ESP32−C3 教程

尚硅谷

一　概述

ESP32-C3 SoC 芯片支持以下功能：

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

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

此芯片由乐鑫公司开发。

二　安装开发工具 ESP−IDF

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

在本入门指南中，我们通过 命令行 进行有关操作。

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

2.1　离线安装 ESP−IDF

点击链接下载离线安装包。

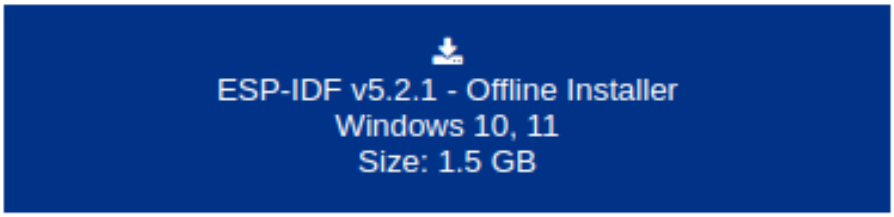

图 1　离线安装包示意图

2.2　安装内容

安装程序会安装以下组件：

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

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

2.3　启动 ESP–IDF 环境

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

`Run ESP-IDF PowerShell Environment`：

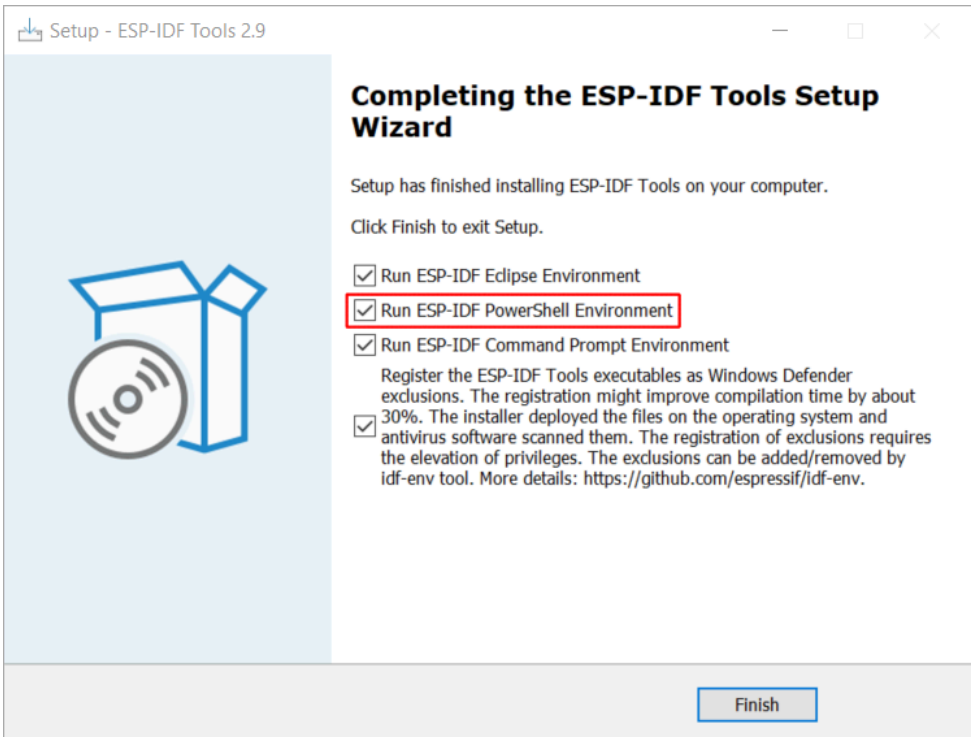

图 2　PowerShell

三　开始创建工程

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

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

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

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

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

3.1 连接设备

现在，请将 ESP32 开发板连接到 PC，并查看开发板使用的串口。

在 Windows 操作系统中，串口名称通常以 COM 开头。

3.2 配置工程

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

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

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

正确操作上述步骤后，系统将显示以下菜单：

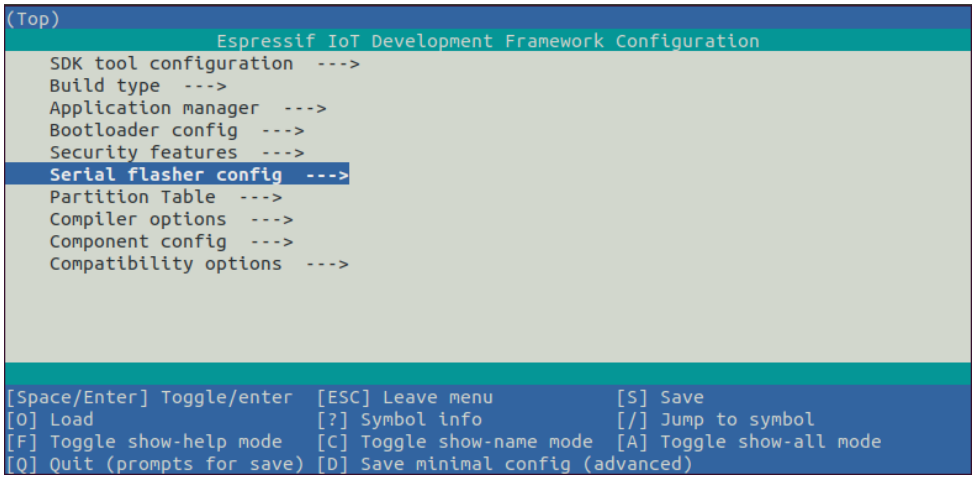

图 3　配置界面示意图

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

3.3 编译工程

请使用以下命令，编译烧录工程：

```
idf.py build
```

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

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

... (more lines of build system output)

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

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

如果一切正常，编译完成后将生成 `.bin` 文件。

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：

```
idf.py flash
```

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

3.5 常规操作

在烧录过程中，会看到类似如下的输出日志：

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

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

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

3.6 监视输出

使用 串口助手 监视输出和调试。

> ⚠ Warning
>
> 当要进行烧写时，请关闭串口助手！

四 基本 GPIO 操作

4.1 GPIO 配置

普通配置

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

有关中断的配置方法

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

操作 GPIO 的 API

```

```

```c
// 将 GPIO 口设置为输入模式
gpio_set_direction(GPIO_NUM_2, GPIO_MODE_INPUT);
// 设置输出模式
gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);
// 输出高低电平
gpio_set_level(GPIO_NUM_1, 1);
gpio_set_level(GPIO_NUM_1, 0);
// 获取 GPIO 的电平
gpio_get_level(GPIO_NUM_2);
```

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

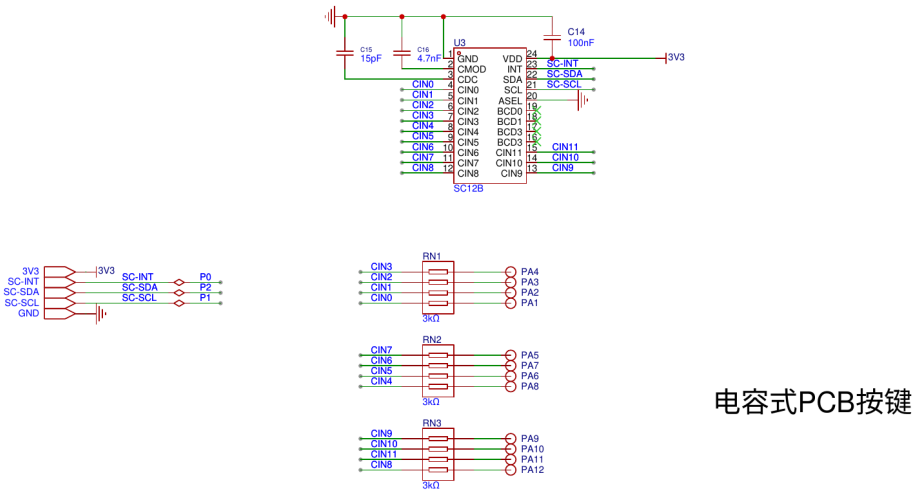

图 4　键盘模块电路图

为了方便操作，我们先来定义一组宏定义以及声明头文件。

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

```c
#ifndef __KEYBOARD_DRIVER_H_
#define __KEYBOARD_DRIVER_H_

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

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

#define I2C_SDA_IN gpio_set_direction(SC12B_SDA, GPIO_MODE_INPUT)
#define I2C_SDA_OUT gpio_set_direction(SC12B_SDA, GPIO_MODE_OUTPUT)

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

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

#define I2C_READ_SDA gpio_get_level(SC12B_SDA)

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

#endif
```

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

```
#include "keyboard_driver.h"

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

/// 产生起始信号
void I2C_Start(void)
{
    I2C_SDA_OUT; // sda 线输出
    I2C_SDA_H;
    I2C_SCL_H;
```

```
    Delay_ms(1);
    I2C_SDA_L; // START:when CLK is high,DATA change form high to
low
    Delay_ms(1);
    I2C_SCL_L; // 钳住 I2C 总线，准备发送或接收数据
    Delay_ms(1);
}

/// 产生停止信号
void I2C_Stop(void)
{
    I2C_SCL_L;
    I2C_SDA_OUT; // sda 线输出
    I2C_SDA_L;   // STOP:when CLK is high DATA change form low to
high
    Delay_ms(1);
    I2C_SCL_H;
    Delay_ms(1);
    I2C_SDA_H; // 发送 I2C 总线结束信号
}

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

/// 等待应答信号到来，成功返回 0 。
uint8_t I2C_Wait_Ack(void)
{
    uint8_t ucErrTime = 0;
    I2C_SCL_L;
    I2C_SDA_IN; // SDA 设置为输入
```

```
    Delay_ms(1);
    I2C_SCL_H;
    Delay_ms(1);
    while (I2C_READ_SDA)
    {
        if (ucErrTime++ > 250)
        {
            // I2C_Stop();
            // printf("接受应答失败\n");
            return 1;
        }
    }
    I2C_SCL_L;
    // printf("接受应答成功\n");
    return 0;
}

/// 发送一个字节
void I2C_Send_Byte(uint8_t d)
{
    uint8_t t = 0;
    I2C_SDA_OUT;
    while (8 > t++)
    {
        I2C_SCL_L;
        Delay_ms(1);
        if (d & 0x80)
        {
            I2C_SDA_H;
        }
        else
        {
            I2C_SDA_L;
        }
        Delay_ms(1); // 对 TEA5767 这三个延时都是必须的
        I2C_SCL_H;
        Delay_ms(1);
        d <<= 1;
    }
}

/// 读 1 个字节
uint8_t I2C_Read_Byte(uint8_t ack)
{
    uint8_t i = 0;
    uint8_t receive = 0;
```

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

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

/// SC12B 简易读取按键值函数（默认直接读取）
/// 此函数只有初始化配置默认的情况下，直接调用，如果在操作前有写入或者其他读取不能调用默认
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; // SDA 设置为输入
    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; // SDA 设置为输入
    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;
}

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;
}

/// GPIO 初始化
void KEYBORAD_init(void)
{
    gpio_config_t io_conf;
    // disable interrupt
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // set as output mode
    io_conf.mode = GPIO_MODE_OUTPUT;
    // bit mask of the pins that you want to set,e.g.SDA
    io_conf.pin_bit_mask = ((1ULL << SC12B_SCL) | (1ULL <<
SC12B_SDA));
```

```
    // disable pull-down mode
    io_conf.pull_down_en = 0;
    // disable pull-up mode
    io_conf.pull_up_en = 1;
    // configure GPIO with the given settings
    gpio_config(&io_conf);

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

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

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

`smart-lock.c` 文件内容如下。

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

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

// 轮训中断事件队列，然后挨个处理
static void process_isr(void *arg)
{
  uint32_t io_num;
  for (;;)
  {
    if (xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY))
    {
      if (io_num == 0)
      {
        uint8_t key = KEYBOARD_read_key();
        printf("按下的键: %d\r\n", key);
      }
```

```
    }
  }
}

static void ISR_QUEUE_Init(void)
{
  // 创建一个队列来处理来自 GPIO 的中断事件
  gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));
  // 开启 process_isr 任务。这个任务的作用是轮训存储中断事件的队列，将队列中的
事件
  // 挨个出队列并进行处理。
  xTaskCreate(process_isr, "process_isr", 2048, NULL, 10, NULL);

  gpio_install_isr_service(0);
  // 将 SC12B_INT 引脚产生的中断交由 gpio_isr_handler 处理。
  // 也就是说一旦 SC12B_INT 产生中断，则调用 gpio_isr_handler 函数。
  gio_isr_handler_add(SC12B_INT, gpio_isr_handler, (void
*)SC12B_INT);
}

// 主程序
void app_main(void)
{
  ISR_QUEUE_Init();
}
```

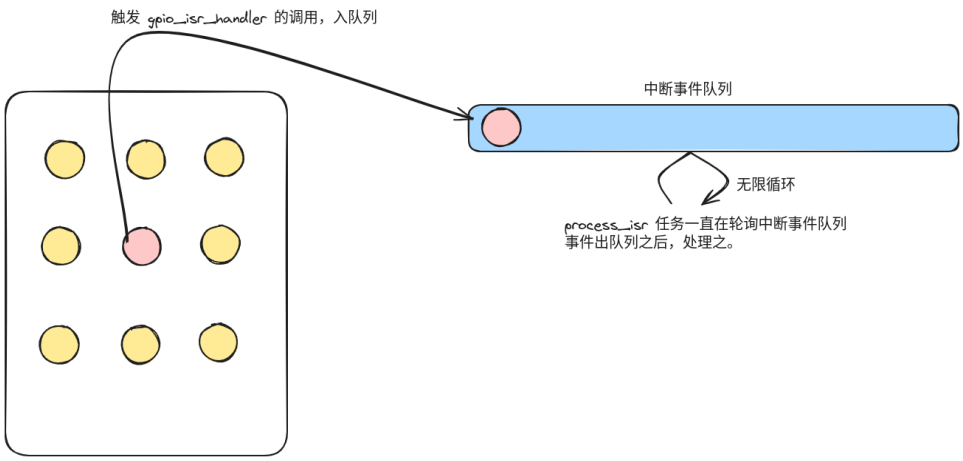

图 5　处理中断示意图

五　红外遥控(RMT)

5.1　简介

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

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

RMT 外设通常支持以下场景：

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

5.2　RMT 符号的内存布局

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

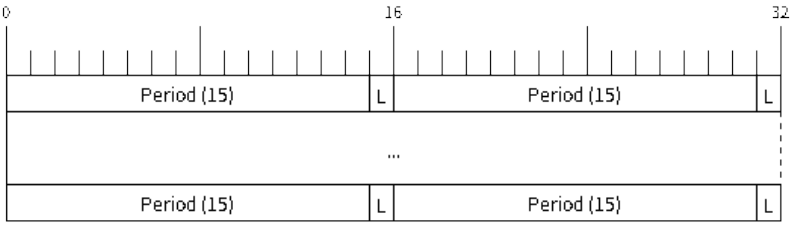

图 6　RMT 符号结构(L−信号电平)

5.3　RMT 发射器概述

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

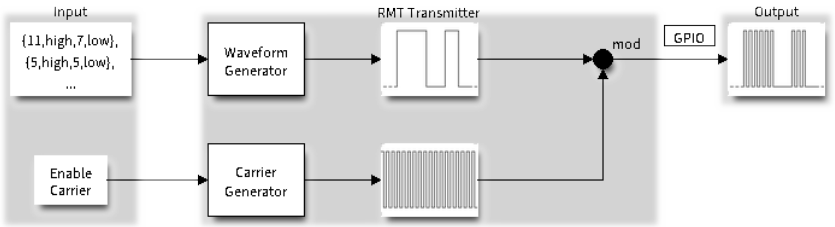

图 7　RMT 发射器概述

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

5.4　RMT 接收器概述

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

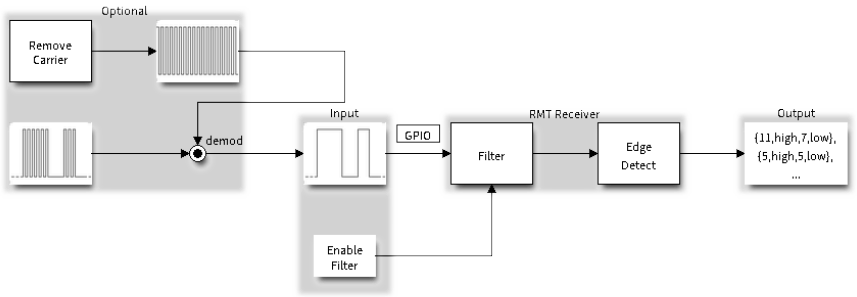

图 8　RMT 接收器概述

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

5.5　补充知识：数字调制

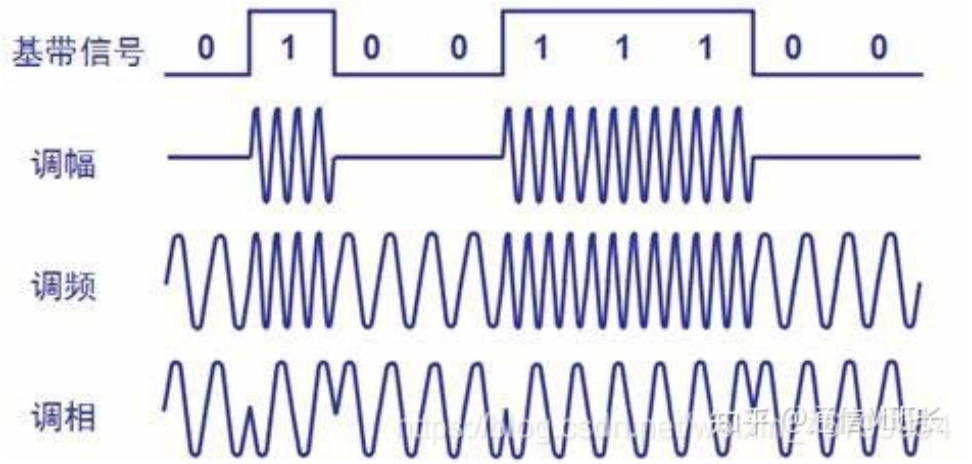

图 9　数字信号调制示意图

5.6　ws2812

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

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

原理图如下：

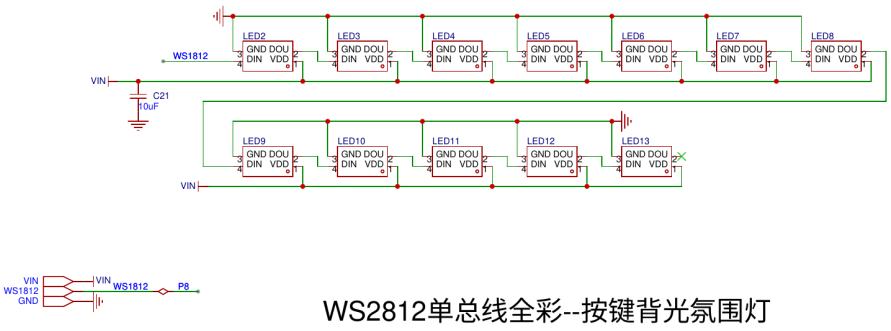

图 10　LED 灯原理图

驱动大部分外设来说，几乎是通过 GPIO 的高低电平来处理，而 ws2812 正是需要这样的电平；RMT（远程控制）模块驱动程序可用于发送和接收红外遥控信号。由于 RMT 灵活性，驱动程序还可用于生成或接收许多其他类型的信号。由一系列脉冲组成的信号由 RMT 的发射器根据值列表生成。这些值定义脉冲持续时间和二进制级别。发射器还可以提供载波并用提供的

脉冲对其进行调制；总的来说它就是一个中间件，就是通过 RMT 模块可以生成解码成包含脉冲持续时间和二进制电平的值的高低电平，从而实现发送和接收我们想要的信号。

关于这个灯珠的资料网上多的是，我总的概述：

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

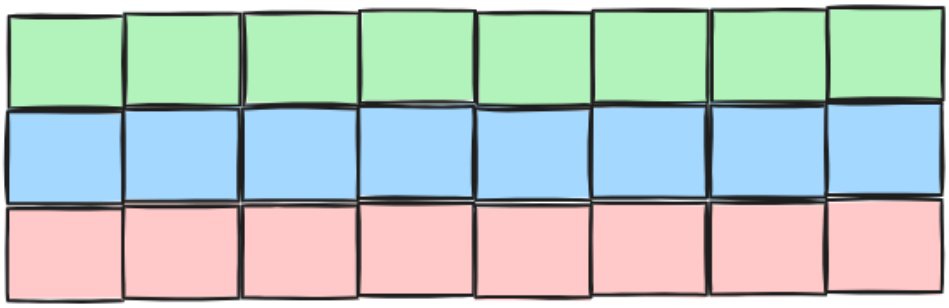

图 11　发送颜色的顺序

5.7　代码

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

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

```
    }

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

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

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

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

尝试编写代码调用点灯方法，将灯点亮。

六　语音模块

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

交互语音播放电路

图 12　语音模块电路图

我们来编写初始化 GPIO 引脚的代码。

```
void AUDIO_Init(void)
{
    ESP_LOGI(AUDIO_TAG, "WTN6170P20_Init");

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

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

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

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

```
        {
            AUDIO_SDA_H;
            DELAY_US(200); // 延时 200us
            AUDIO_SDA_L;
            DELAY_US(600); // 延时 600us
        }
        S_DATA = S_DATA >> 1;
        B_DATA = S_DATA & 0X01;
    }
    AUDIO_SDA_H;
    DELAY_MS(2);
}
```

七　电机驱动

电机用来开关锁。也就是通过驱动电机进行正转反转来开关锁。

当然我们还是通过 GPIO 的拉高拉低来驱动电机。比较简单。

电路图如下：

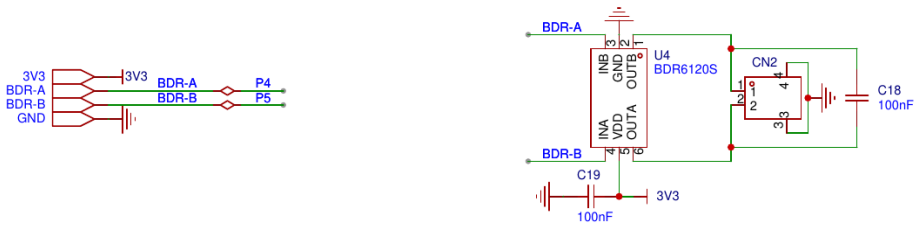

图 13　电机模块电路图

初始化 GPIO 引脚代码

```
void MOTOR_Init(void)
{
    gpio_config_t io_conf;
    // 禁用中断
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // 设置为输出模式
    io_conf.mode = GPIO_MODE_OUTPUT;
```

```
    // 设置要用的两个引脚
    io_conf.pin_bit_mask = ((1ULL << MOTOR_DRIVER_NUM_0) | (1ULL <<
MOTOR_DRIVER_NUM_1));
    gpio_config(&io_conf);

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

开锁代码

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

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

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

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

八　指纹模块

MCU 使用串口和指纹模块进行通信。电路图如下：

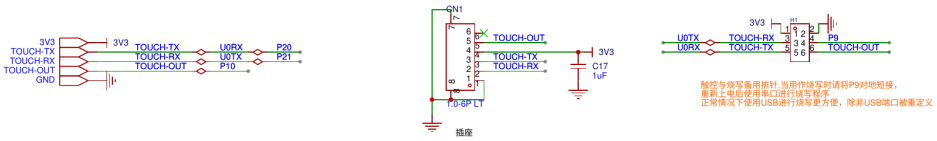

图 14　指纹模块电路图

我们先来写头文件

```c
#ifndef __FINGERPRINT_DRIVER_H_
#define __FINGERPRINT_DRIVER_H_

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

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

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

#define BUF_SIZE (1024)

#define TOUCH_INT GPIO_NUM_8

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

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

/// 获取指纹图像
int get_image(void);

/// 获取指纹特征
int gen_char(void);

/// 搜索指纹
int search(void);
```

```
/// 读取指纹芯片配置参数
void read_sys_params(void);

#endif
```

然后编写头文件中接口的实现

```
#include "fingerprint_driver.h"

void FINGERPRINT_Init(void)
{
    printf("hahahahahahah\r\n");

    /* Configure parameters of an UART driver,
     * communication pins and install the driver */
    uart_config_t uart_config = {
        .baud_rate = ECHO_UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    int intr_alloc_flags = 0;

    ESP_ERROR_CHECK(uart_driver_install(ECHO_UART_PORT_NUM, BUF_SIZE
* 2, 0, 0, NULL, intr_alloc_flags));
    ESP_ERROR_CHECK(uart_param_config(ECHO_UART_PORT_NUM,
&uart_config));
    ESP_ERROR_CHECK(uart_set_pin(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);
}

// 检测是否有手指放在模组上面
int get_image(void)
{
    uint8_t *data = (uint8_t *)malloc(BUF_SIZE);

    // 验证用获取图像 0x01，验证指纹时，探测手指，探测到后录入指纹图像存于图像缓
冲区。返回确认码表示：录入成功、无手指等。
    uint8_t PS_GetImageBuffer[12] = {0xEF, 0x01, 0xFF, 0xFF, 0xFF,
0xFF, 0x01, 0x00, 0x03, 0x01, 0x00, 0x05};

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

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

    int result = 0xFF;

    if (len)
    {
        if (data[6] == 0x07)
```

```
        {
            if (data[9] == 0x00)
            {
                result = 0;
            }
            else if (data[9] == 0x01)
            {
                result = 1;
            }
            else if (data[9] == 0x02)
            {
                result = 2;
            }
        }
    }

    free(data);

    return result;
}

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

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

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

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

    int result = 0xFF;

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

    free(data);
```

```
    return result;
}

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

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

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

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

    int result = 0xFF;

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

    free(data);

    return result;
}

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

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

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

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

    if (len)
    {
        if (data[6] == 0x07)
        {
            if (data[9] == 0x00)
            {
                int register_count = (data[10] << 8) | data[11];
                printf("register count ==> %d\r\n", register_count);
                int fingerprint_template_size = (data[12] << 8) |
data[13];
                printf("finger print template size ==> %d\r\n",
fingerprint_template_size);
                int fingerprint_library_size = (data[14] << 8) |
data[15];
                printf("finger print library size ==> %d\r\n",
fingerprint_library_size);
                int score_level = (data[16] << 8) | data[17];
                printf("score level ==> %d\r\n", score_level);
                // device address
                printf("device address ==> 0x");
                for (int i = 0; i < 4; i++)
                {
                    printf("%02X ", data[18 + i]);
                }
                printf("\r\n");
                // data packet size
                int packet_size = (data[22] << 8) | data[23];
                if (packet_size == 0)
                {
                    printf("packet size ==> 32 bytes\r\n");
                }
                else if (packet_size == 1)
                {
                    printf("packet size ==> 64 bytes\r\n");
                }
                else if (packet_size == 2)
                {
                    printf("packet size ==> 128 bytes\r\n");
                }
                else if (packet_size == 3)
                {
                    printf("packet size ==> 256 bytes\r\n");
                }
                // baud rate
```

```
                int baud_rate = (data[24] << 8) | data[25];
                printf("baud rate ==> %d\r\n", 9600 * baud_rate);
            }
            else if (data[9] == 0x01)
            {
                printf("send packet error\r\n");
            }
        }
    }

    free(data);
}
```

九　蓝牙功能

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

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

Bluedroid

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

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

BLE（Bluetooth Low Energy）

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

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

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

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 Includes

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

这些包含文件是运行 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。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 BT 控制器和 VHCI 配置程序。
- `esp_bt_main.h`：实现 Bluedroid 堆栈的初始化和启用。
- `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。

9.2 入口函数

入口函数是 `app_main()` 函数。

```
```

```
 void app_main()
{
    esp_err_t ret;

    // Initialize NVS.
    ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    esp_bt_controller_config_t bt_cfg =
BT_CONTROLLER_INIT_CONFIG_DEFAULT();
    ret = esp_bt_controller_init(&bt_cfg);
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s initialize controller failed\n",
__func__);
        return;
    }

    ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s enable controller failed\n",
__func__);
        return;
    }
    ret = esp_bluedroid_init();
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s init bluetooth failed\n", __func__);
        return;
    }
    ret = esp_bluedroid_enable();
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s enable bluetooth failed\n",
__func__);
        return;
    }

    ret = esp_ble_gatts_register_callback(gatts_event_handler);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts register error, error code = %x",
ret);
        return;
    }
    ret = esp_ble_gap_register_callback(gap_event_handler);
```

```
    if (ret){
        ESP_LOGE(GATTS_TAG, "gap register error, error code = %x",
ret);
        return;
    }
    ret = esp_ble_gatts_app_register(PROFILE_A_APP_ID);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts app register error, error code =
%x", ret);
        return;
    }
    ret = esp_ble_gatts_app_register(PROFILE_B_APP_ID);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts app register error, error code =
%x", ret);
        return;
    }
    esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(512);
    if (local_mtu_ret){
        ESP_LOGE(GATTS_TAG, "set local  MTU failed, error code =
%x", local_mtu_ret);
    }
    return;
}
```

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

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

9.3 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 BT 控制器配置结构体来初始化 BT 控制器，该结构体使用 `BT_CONTROLLER_INIT_CONFIG_DEFAULT()` 宏生成的默认设置。BT 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）。BT 控制器对用户应用程序是不可见的，它处理 BLE 堆栈的底层。控制器配置包括设置 BT 控制器堆栈大小、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 BT 控制器：

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

接下来，控制器使能为 BLE 模式。

```
ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
```

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

支持四种蓝牙模式：

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

在 BT 控制器初始化之后，Bluedroid 堆栈（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：

```
ret = esp_bluedroid_init();
ret = esp_bluedroid_enable();
```

此时程序流程中的蓝牙堆栈已经启动并运行，但应用程序的功能尚未定义。功能是通过响应事件来定义的，例如当另一个设备尝试读取或写入参数并建立连接时会发生什么。两个主要的事件管理器是 GAP 和 GATT 事件处理器。应用程序需要为每个事件处理器注册一个回调函数，以便让应用程序知道哪些函数将处理 GAP 和 GATT 事件：

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

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

9.4 应用程序配置文件(APPLICATION PROFILES)

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

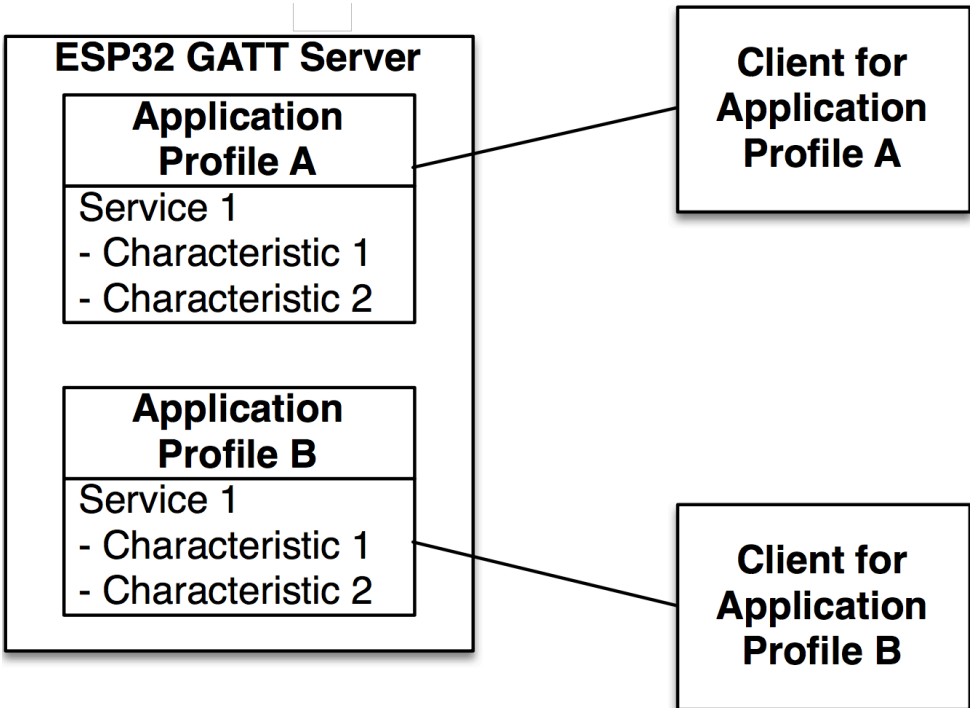

图 15　GATT 服务器

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

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

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

结构实现是：

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

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

```
static struct gatts_profile_inst gl_profile_tab[PROFILE_NUM] = {
    [PROFILE_A_APP_ID] = {
        .gatts_cb = gatts_profile_a_event_handler,
        .gatts_if = ESP_GATT_IF_NONE,
    [PROFILE_B_APP_ID] = {
        .gatts_cb = gatts_profile_b_event_handler,
        .gatts_if = ESP_GATT_IF_NONE,
    },
};
```

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

```
esp_ble_gatts_app_register(PROFILE_A_APP_ID);
esp_ble_gatts_app_register(PROFILE_B_APP_ID);
```

9.5 配置 GAP 参数

注册应用程序事件是在程序生命周期中首先触发的事件，这个示例使用 Profile A GATT 事件句柄在注册时配置广告参数。这个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 iBeacons、Eddystone 或其他专有和自定义帧类型，如用于室内定位服务的那些，这些与标准规范不同。

用于配置标准蓝牙规范广告参数的函数是 `esp_ble_gap_config_adv_data()`，它接受一个指向 `esp_ble_adv_data_t` 结构的指针。广告数据的 `esp_ble_adv_data_t` 数据结构定义如下：

```
typedef struct {
    bool set_scan_rsp;            /*!< Set this advertising data as
scan response or not*/
    bool include_name;            /*!< Advertising data include
device name or not */
    bool include_txpower;         /*!< Advertising data include TX
power */
    int min_interval;             /*!< Advertising data show slave
preferred connection min interval */
    int max_interval;             /*!< Advertising data show slave
preferred connection max interval */
    int appearance;               /*!< External appearance of device
*/
    uint16_t manufacturer_len;    /*!< Manufacturer data length */
    uint8_t *p_manufacturer_data; /*!< Manufacturer data point */
    uint16_t service_data_len;    /*!< Service data length */
    uint8_t *p_service_data;      /*!< Service data point */
    uint16_t service_uuid_len;    /*!< Service uuid length */
    uint8_t *p_service_uuid;      /*!< Service uuid array point */
    uint8_t flag;                 /*!< Advertising flag of discovery
mode, see BLE_ADV_DATA_FLAG detail */
} esp_ble_adv_data_t;
```

在本示例程序中，参数被初始化为以下：

```
static esp_ble_adv_data_t adv_data = {
    .set_scan_rsp = false,
    .include_name = true,
    .include_txpower = true,
    .min_interval = 0x0006,
    .max_interval = 0x0010,
    .appearance = 0x00,
    .manufacturer_len = 0, //TEST_MANUFACTURER_DATA_LEN,
    .p_manufacturer_data =  NULL, //&test_manufacturer[0],
    .service_data_len = 0,
    .p_service_data = NULL,
    .service_uuid_len = 32,
    .p_service_uuid = test_service_uuid128,
```

```
    .flag = (ESP_BLE_ADV_FLAG_GEN_DISC |
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT),
};
```

最小和最大从设备首选连接间隔以 1.25 毫秒为单位设置。在这个示例中，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` 。

广告负载可以包含多达 31 字节的数据。参数数据可能足够大，以至于超过 31 字节的广告包限制，这会导致栈切割广告包并留下部分参数。如果取消注释制造商长度和数据，这种行为可以在这个示例中展示，这会导致服务在重新编译和测试后不被广告。

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，这要求创建并传递一个缓冲区，用于广告数据和扫描响应数据。在这个示例中，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()` 函数设置设备名称。注册事件处理程序如下所示：

```
static void gatts_profile_a_event_handler(esp_gatts_cb_event_t
event, esp_gatt_if_t gatts_if, esp_ble_gatts_cb_param_t *param) {
    switch (event) {
    case ESP_GATTS_REG_EVT:
         ESP_LOGI(GATTS_TAG, "REGISTER_APP_EVT, status %d, app_id
%d\n", param->reg.status, param->reg.app_id);
         gl_profile_tab[PROFILE_A_APP_ID].service_id.is_primary =
true;
         gl_profile_tab[PROFILE_A_APP_ID].service_id.id.inst_id =
0x00;
         gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.len =
ESP_UUID_LEN_16;

gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.uuid.uuid16 =
GATTS_SERVICE_UUID_TEST_A;

         esp_ble_gap_set_device_name(TEST_DEVICE_NAME);
#ifdef CONFIG_SET_RAW_ADV_DATA
        esp_err_t raw_adv_ret =
esp_ble_gap_config_adv_data_raw(raw_adv_data, sizeof(raw_adv_data));
        if (raw_adv_ret){
            ESP_LOGE(GATTS_TAG, "config raw adv data failed, error
code = %x ", raw_adv_ret);
        }
```

```
        adv_config_done |= adv_config_flag;
        esp_err_t raw_scan_ret =
esp_ble_gap_config_scan_rsp_data_raw(raw_scan_rsp_data,
sizeof(raw_scan_rsp_data));
        if (raw_scan_ret){
            ESP_LOGE(GATTS_TAG, "config raw scan rsp data failed,
error code = %x", raw_scan_ret);
        }
        adv_config_done |= scan_rsp_config_flag;
#else
        //config adv data
        esp_err_t ret = esp_ble_gap_config_adv_data(&adv_data);
        if (ret){
            ESP_LOGE(GATTS_TAG, "config adv data failed, error code
= %x", ret);
        }
        adv_config_done |= adv_config_flag;
        //config scan response data
        ret = esp_ble_gap_config_adv_data(&scan_rsp_data);
        if (ret){
            ESP_LOGE(GATTS_TAG, "config scan response data failed,
error code = %x", ret);
        }
        adv_config_done |= scan_rsp_config_flag;
#endif
```

9.6 GAP Event Handler(GAP 事件句柄)

一旦设置了广告数据，GAP 事件 `ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT` 将被触发。对于设置原始广告数据的情况，触发的事件是 `ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT`。另外，当设置了原始扫描响应数据时，将触发 `ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT` 事件。

```
static void gap_event_handler(esp_gap_ble_cb_event_t event,
esp_ble_gap_cb_param_t *param)
{
    switch (event) {
#ifdef CONFIG_SET_RAW_ADV_DATA
    case ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT:
         adv_config_done &= (~adv_config_flag);
         if (adv_config_done==0){
             esp_ble_gap_start_advertising(&adv_params);
         }
         break;
```

```
    case ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT:
         adv_config_done &= (~scan_rsp_config_flag);
         if (adv_config_done==0){
             esp_ble_gap_start_advertising(&adv_params);
         }
         break;
#else
    case ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT:
         adv_config_done &= (~adv_config_flag);
         if (adv_config_done == 0){
             esp_ble_gap_start_advertising(&adv_params);
         }
         break;
    case ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT:
         adv_config_done &= (~scan_rsp_config_flag);
         if (adv_config_done == 0){
             esp_ble_gap_start_advertising(&adv_params);
         }
         break;
#endif
...
...
```

无论哪种情况，服务器都可以使用 `esp_ble_gap_start_advertising()` 函数开始广告，该函数接受一个类型为 `esp_ble_adv_params_t` 的结构体，其中包含了堆栈操作所需的广告参数：

```
/// Advertising parameters
typedef struct {
    uint16_t adv_int_min;
    /*!< Minimum advertising interval for undirected and low duty
cycle directed advertising.
          Range: 0x0020 to 0x4000
          Default: N = 0x0800 (1.28 second)
          Time = N * 0.625 msec
          Time Range: 20 ms to 10.24 sec */
    uint16_t adv_int_max;
    /*!< Maximum advertising interval for undirected and low duty
cycle directed advertising.
          Range: 0x0020 to 0x4000
          Default: N = 0x0800 (1.28 second)
          Time = N * 0.625 msec
          Time Range: 20 ms to 10.24 sec */
    esp_ble_adv_type_t adv_type;            /*!< Advertising type */
```

```
    esp_ble_addr_type_t own_addr_type;      /*!< Owner bluetooth
device address type */
    esp_bd_addr_t peer_addr;                /*!< Peer device
bluetooth device address */
    esp_ble_addr_type_t peer_addr_type;     /*!< Peer device
bluetooth device address type */
    esp_ble_adv_channel_t channel_map;      /*!< Advertising channel
map */
    esp_ble_adv_filter_t adv_filter_policy; /*!< Advertising filter
policy */
}
esp_ble_adv_params_t;
```

请注意，`esp_ble_gap_config_adv_data()` 配置将要广告给客户端的数据，并接受一个 `esp_ble_adv_data_t` 结构体，而 `esp_ble_gap_start_advertising()` 使服务器实际开始广告，并接受一个 `esp_ble_adv_params_t` 结构体。广告数据是显示给客户端的信息，而广告参数是 GAP 执行所需的配置。

对于这个示例，广告参数如下初始化：

```
static esp_ble_adv_params_t test_adv_params = {
    .adv_int_min        = 0x20,
    .adv_int_max        = 0x40,
    .adv_type           = ADV_TYPE_IND,
    .own_addr_type      = BLE_ADDR_TYPE_PUBLIC,
    //.peer_addr        =
    //.peer_addr_type   =
    .channel_map        = ADV_CHNL_ALL,
    .adv_filter_policy  = ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY,
};
```

这些参数将广告间隔配置在 40 毫秒到 80 毫秒之间。广告类型为`ADV_IND`，这是一种通用的、不针对特定中央设备的可连接类型。地址类型为公开，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，在这个示例中用于检查广告状态是否确实为正在广告。否则，将打印一个错误消息。

```
...
...
    case ESP_GAP_BLE_ADV_START_COMPLETE_EVT:
    //advertising start complete event to indicate advertising start
```

```
successfully or failed
         if (param->adv_start_cmpl.status != ESP_BT_STATUS_SUCCESS)
{
             ESP_LOGE(GATTS_TAG, "Advertising start failed\n");
         }
         break;
...
...
```

9.7 GATT Event Handler

当一个应用程序配置文件被注册时，将触发一个 `ESP_GATTS_REG_EVT` 事件。`ESP_GATTS_REG_EVT` 的参数有：

```
esp_gatt_status_t status;  /*!< Operation status */`
uint16_t app_id;           /*!< Application id which input in
register API */`
```

除了前面的参数外，该事件还包含由 BLE 堆栈分配的 GATT 接口。该事件被 `gatts_event_handler()` 捕获，该处理器用于将生成的接口存储在配置文件表中，然后将事件转发到相应的配置文件事件处理器。

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

/* If the gatts_if equal to profile A, call profile A cb handler,
 * so here call each profile's callback */
    do {
```

```
        int idx;
        for (idx = 0; idx < PROFILE_NUM; idx++) {
            if (gatts_if == ESP_GATT_IF_NONE||gatts_if ==
gl_profile_tab[idx].gatts_if)       {
                if (gl_profile_tab[idx].gatts_cb) {
                    gl_profile_tab[idx].gatts_cb(event, gatts_if,
param);
                }
            }
        }
    } while (0);
}
```

9.8 创建服务

注册事件还用于通过使用`esp_ble_gatts_create_service()`创建服务。当服务创建完成时，会调用回调事件 ESP_GATTS_CREATE_EVT 来向配置文件报告状态和服务 ID。创建服务的方式是：

```
...
...
esp_ble_gatts_create_service(gatts_if,
&gl_profile_tab[PROFILE_A_APP_ID].service_id,
GATTS_NUM_HANDLE_TEST_A);
break;
...
...
```

句柄的数量定义为 4：

```
#define GATTS_NUM_HANDLE_TEST_A     4
```

句柄如下：

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 初始化，并通过`GATTS_SERVICE_UUID_TEST_A`定义 UUID。

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，使用相同 UUID 的两个服务，则需要使用不同的实例 ID 来区分这两个服务。

应用程序配置文件 B 以与配置文件 A 相同的方式创建服务：

```
static void gatts_profile_b_event_handler(esp_gatts_cb_event_t
event, esp_gatt_if_t gatts_if, esp_ble_gatts_cb_param_t *param) {
    switch (event) {
    case ESP_GATTS_REG_EVT:
         ESP_LOGI(GATTS_TAG, "REGISTER_APP_EVT, status %d, app_id
%d\n", param->reg.status, param->reg.app_id);
         gl_profile_tab[PROFILE_B_APP_ID].service_id.is_primary =
true;
         gl_profile_tab[PROFILE_B_APP_ID].service_id.id.inst_id =
0x00;
         gl_profile_tab[PROFILE_B_APP_ID].service_id.id.uuid.len =
ESP_UUID_LEN_16;

gl_profile_tab[PROFILE_B_APP_ID].service_id.id.uuid.uuid.uuid16 =
GATTS_SERVICE_UUID_TEST_B;

         esp_ble_gatts_create_service(gatts_if,
&gl_profile_tab[PROFILE_B_APP_ID].service_id,
GATTS_NUM_HANDLE_TEST_B);
         break;
...
...
}
```

9.9 启动服务和创建特征(Characteristics)

当一个服务成功创建时，由配置文件 GATT 处理器管理的 `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性。对于配置文件 A 的情况，服务被启动并且特性被添加如下：

```
case ESP_GATTS_CREATE_EVT:
     ESP_LOGI(GATTS_TAG, "CREATE_SERVICE_EVT, status %d,
service_handle %d\n", param->create.status, param-
>create.service_handle);
     gl_profile_tab[PROFILE_A_APP_ID].service_handle = param-
>create.service_handle;
     gl_profile_tab[PROFILE_A_APP_ID].char_uuid.len =
```

```
ESP_UUID_LEN_16;
     gl_profile_tab[PROFILE_A_APP_ID].char_uuid.uuid.uuid16 =
GATTS_CHAR_UUID_TEST_A;


esp_ble_gatts_start_service(gl_profile_tab[PROFILE_A_APP_ID].service_handle);
     a_property = ESP_GATT_CHAR_PROP_BIT_READ |
ESP_GATT_CHAR_PROP_BIT_WRITE | ESP_GATT_CHAR_PROP_BIT_NOTIFY;
     esp_err_t add_char_ret =

esp_ble_gatts_add_char(gl_profile_tab[PROFILE_A_APP_ID].service_handle,

&gl_profile_tab[PROFILE_A_APP_ID].char_uuid,
                            ESP_GATT_PERM_READ |
ESP_GATT_PERM_WRITE,
                            a_property,
                            &gatts_demo_char1_val,
                            NULL);
    if (add_char_ret){
        ESP_LOGE(GATTS_TAG, "add char failed, error code
=%x",add_char_ret);
    }
    break;
```

首先，由 BLE 堆栈生成的服务句柄被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，设置特性的 UUID 及其 UUID 长度。特性 UUID 的长度再次为 16 位。使用之前生成的服务句柄，通过`esp_ble_gatts_start_service()`函数启动服务。触发了一个`ESP_GATTS_START_EVT`事件，用于打印信息。特性通过`esp_ble_gatts_add_char()`函数添加到服务中，结合特性的权限和属性。在这个示例中，两个配置文件中的特性都按以下方式设置：

权限：

- `ESP_GATT_PERM_READ`：允许读取特性值
- `ESP_GATT_PERM_WRITE`：允许写入特性值

属性：

- `ESP_GATT_CHAR_PROP_BIT_READ`：特性可以被读取
- `ESP_GATT_CHAR_PROP_BIT_WRITE`：特性可以被写入
- `ESP_GATT_CHAR_PROP_BIT_NOTIFY`：特性可以通知值变化

对于读和写拥有权限和属性可能看起来有些多余。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。从这个意义上讲，属性作为一个提示，帮助客户端正确访问服务器资源。另一方面，权限是授予客户端读取或写入该属性的授权。例如，如果客户端尝试写入一个它没有写权限的属性，服务器将拒绝该请求，即使设置了写属性。

此外，这个示例给特性赋予了一个初始值，由 `gatts_demo_char1_val` 表示。初始值定义如下：

```
esp_attr_value_t gatts_demo_char1_val =
{
    .attr_max_len = GATTS_DEMO_CHAR_VAL_LEN_MAX,
    .attr_len     = sizeof(char1_str),
    .attr_value   = char1_str,
};
```

这里 `char1_str` 是占位数据，没啥用

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

and the characteristic length is defined as:

```
#define GATTS_DEMO_CHAR_VAL_LEN_MAX 0x40
```

特性的初始值必须是一个非空对象，并且特性长度必须始终大于零，否则堆栈将返回错误。

最后，特性被配置为每次读取或写入特性时都需要手动发送响应，而不是让堆栈自动响应。这是通过设置 `esp_ble_gatts_add_char()` 函数的最后一个参数，代表属性响应控制参数，为 `ESP_GATT_RSP_BY_APP` 或 NULL 来配置的。

9.10 创建特征描述符

向服务添加特性会触发一个 `ESP_GATTS_ADD_CHAR_EVT` 事件，该事件返回堆栈为刚添加的特性生成的句柄。该事件包括以下参数：

```
esp_gatt_status_t status;          /*!< Operation status */
uint16_t attr_handle;             /*!< Characteristic attribute
handle */
uint16_t service_handle;           /*!< Service attribute handle */
esp_bt_uuid_t char_uuid;           /*!< Characteristic uuid */
```

事件返回的属性句柄被存储在配置文件表中，同时也设置了特性描述符的长度和 UUID。使用 `esp_ble_gatts_get_attr_value()` 函数读取特性的长度和值，然后出于信息目的打印出来。最后，使用 `esp_ble_gatts_add_char_descr()` 函数添加特性描述符。使用的参数包括服务句柄、描述符 UUID、写和读权限、一个初始值和自动响应设置。特性描述符的初始值可以是一个 NULL 指针，自动响应参数也设置为 NULL，这意味着需要响应的请求必须手动回复。

```
    case ESP_GATTS_ADD_CHAR_EVT: {
         uint16_t length = 0;
         const uint8_t *prf_char;

         ESP_LOGI(GATTS_TAG, "ADD_CHAR_EVT, status %d,  attr_handle
%d, service_handle %d\n",
                  param->add_char.status, param-
>add_char.attr_handle, param->add_char.service_handle);
                  gl_profile_tab[PROFILE_A_APP_ID].char_handle =
param->add_char.attr_handle;
                  gl_profile_tab[PROFILE_A_APP_ID].descr_uuid.len =
ESP_UUID_LEN_16;

gl_profile_tab[PROFILE_A_APP_ID].descr_uuid.uuid.uuid16 =
ESP_GATT_UUID_CHAR_CLIENT_CONFIG;
                  esp_err_t get_attr_ret =
esp_ble_gatts_get_attr_value(param->add_char.attr_handle, &length,
&prf_char);
         if (get_attr_ret == ESP_FAIL){
             ESP_LOGE(GATTS_TAG, "ILLEGAL HANDLE");
         }
         ESP_LOGI(GATTS_TAG, "the gatts demo char length = %x\n",
length);
         for(int i = 0; i < length; i++){
             ESP_LOGI(GATTS_TAG, "prf_char[%x] =
%x\n",i,prf_char[i]);
         }
         esp_err_t add_descr_ret = esp_ble_gatts_add_char_descr(

gl_profile_tab[PROFILE_A_APP_ID].service_handle,

&gl_profile_tab[PROFILE_A_APP_ID].descr_uuid,
                                    ESP_GATT_PERM_READ |
ESP_GATT_PERM_WRITE,
                                    NULL,NULL);
         if (add_descr_ret){
            ESP_LOGE(GATTS_TAG, "add char descr failed, error code =
%x", add_descr_ret);
         }
         break;
    }
```

一旦添加了描述符，就会触发 `ESP_GATTS_ADD_CHAR_DESCR_EVT` 事件，在本示例中用于打印一条信息消息。

```
    case ESP_GATTS_ADD_CHAR_DESCR_EVT:
        ESP_LOGI(GATTS_TAG, "ADD_DESCR_EVT, status %d, attr_handle
%d, service_handle %d\n",
                 param->add_char.status, param-
>add_char.attr_handle,
                 param->add_char.service_handle);
        break;
```

这个过程在配置文件 B 的事件处理器中重复，以便为该配置文件创建服务和特性。

9.11 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，如延迟、最小连接间隔、最大连接间隔和超时。连接参数被存储到一个`esp_ble_conn_update_params_t`结构体中，然后传递给`esp_ble_gap_update_conn_params()`函数。更新连接参数的过程只需要做一次，因此配置文件 B 的连接事件处理器不包括`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。

配置文件 A 连接事件：

```
case ESP_GATTS_CONNECT_EVT: {
    esp_ble_conn_update_params_t conn_params = {0};
    memcpy(conn_params.bda, param->connect.remote_bda,
sizeof(esp_bd_addr_t));
    /* For the IOS system, please reference the apple official
documents about the ble connection parameters restrictions. */
    conn_params.latency = 0;
    conn_params.max_int = 0x30;    // max_int = 0x30*1.25ms =
40ms
    conn_params.min_int = 0x10;    // min_int = 0x10*1.25ms =
20ms
    conn_params.timeout = 400;     // timeout = 400*10ms = 4000ms
    ESP_LOGI(GATTS_TAG, "ESP_GATTS_CONNECT_EVT, conn_id %d, remote
%02x:%02x:%02x:%02x:%02x:%02x:, is_conn %d",
             param->connect.conn_id,
             param->connect.remote_bda[0],
             param->connect.remote_bda[1],
             param->connect.remote_bda[2],
             param->connect.remote_bda[3],
             param->connect.remote_bda[4],
             param->connect.remote_bda[5],
             param->connect.is_connected);
   gl_profile_tab[PROFILE_A_APP_ID].conn_id = param-
```

```
>connect.conn_id;
   //start sent the update connection parameters to the peer device.
   esp_ble_gap_update_conn_params(&conn_params);
   break;
    }
```

配置文件 B 连接事件：

```
case ESP_GATTS_CONNECT_EVT:
     ESP_LOGI(GATTS_TAG, "CONNECT_EVT, conn_id %d, remote %02x:%02x:
%02x:%02x:%02x:%02x:, is_conn %d\n",
              param->connect.conn_id,
              param->connect.remote_bda[0],
              param->connect.remote_bda[1],
              param->connect.remote_bda[2],
              param->connect.remote_bda[3],
              param->connect.remote_bda[4],
              param->connect.remote_bda[5],
              param->connect.is_connected);
    gl_profile_tab[PROFILE_B_APP_ID].conn_id = param-
>connect.conn_id;
    break;
```

`esp_ble_gap_update_conn_params()` 函数触发一个 GAP 事件 `ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT`，用于打印连接信息：

```
    case ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT:
         ESP_LOGI(GATTS_TAG, "update connection params status = %d,
min_int = %d, max_int = %d,
                  conn_int = %d,latency = %d, timeout = %d",
                  param->update_conn_params.status,
                  param->update_conn_params.min_int,
                  param->update_conn_params.max_int,
                  param->update_conn_params.conn_int,
                  param->update_conn_params.latency,
                  param->update_conn_params.timeout);
         break;
```

9.12 管理读事件

现在服务和特性已经创建并启动，程序可以接收读和写事件。读操作由 `ESP_GATTS_READ_EVT` 事件表示，该事件有以下参数：

```
uint16_t conn_id;          /*!< Connection id */
uint32_t trans_id;         /*!< Transfer id */
esp_bd_addr_t bda;         /*!< The bluetooth device address which
been read */
uint16_t handle;           /*!< The attribute handle */
uint16_t offset;           /*!< Offset of the value, if the value is
too long */
bool is_long;              /*!< The value is too long or not */
bool need_rsp;             /*!< The read operation need to do
response */
```

在这个示例中，使用事件给定的相同句柄，构造一个带有虚拟数据的响应并发送回主机。除了响应之外，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。如果在创建特性或描述符时将自动响应字节设置为 NULL，则需要此函数。

```
case ESP_GATTS_READ_EVT: {
     ESP_LOGI(GATTS_TAG, "GATT_READ_EVT, conn_id %d, trans_id %d,
handle %d\n",
              param->read.conn_id, param->read.trans_id, param-
>read.handle);
              esp_gatt_rsp_t rsp;
              memset(&rsp, 0, sizeof(esp_gatt_rsp_t));
              rsp.attr_value.handle = param->read.handle;
              rsp.attr_value.len = 4;
              rsp.attr_value.value[0] = 0xde;
              rsp.attr_value.value[1] = 0xed;
              rsp.attr_value.value[2] = 0xbe;
              rsp.attr_value.value[3] = 0xef;
              esp_ble_gatts_send_response(gatts_if,
                                          param->read.conn_id,
                                          param->read.trans_id,
                                          ESP_GATT_OK, &rsp);
     break;
    }
```

9.13 管理写事件

写事件由 `ESP_GATTS_WRITE_EVT` 事件表示，该事件具有以下参数：

```
uint16_t conn_id;          /*!< Connection id */
uint32_t trans_id;         /*!< Transfer id */
```

```
esp_bd_addr_t bda;          /*!< The bluetooth device address which
been written */
uint16_t handle;            /*!< The attribute handle */
uint16_t offset;            /*!< Offset of the value, if the value is
too long */
bool need_rsp;              /*!< The write operation need to do
response */
bool is_prep;               /*!< This write operation is prepare write
*/
uint16_t len;               /*!< The write attribute value length */
uint8_t *value;             /*!< The write attribute value */
```

在这个示例中实现了两种类型的写事件，写特性值和写长特性值。第一种类型的写操作用于当特性值可以适应一个属性协议最大传输单元（ATT MTU），通常是 23 字节长的情况。第二种类型用于当要写入的属性长度超过可以在一个单独的 ATT 消息中发送的长度时，通过使用准备写响应将数据分成多个块，之后使用执行写请求来确认或取消完整的写请求。这种行为在蓝牙规范版本 4.2，卷 3，第 G 部分，第 4.9 节中定义。下图展示了写长特性消息流程。

当触发写事件时，这个示例会打印日志消息，然后执行`example_write_event_env()`函数。

```
case ESP_GATTS_WRITE_EVT: {
     ESP_LOGI(GATTS_TAG, "GATT_WRITE_EVT, conn_id %d, trans_id %d,
handle %d\n", param->write.conn_id, param->write.trans_id, param-
>write.handle);
     if (!param->write.is_prep){
        ESP_LOGI(GATTS_TAG, "GATT_WRITE_EVT, value len %d, value :",
param->write.len);
        esp_log_buffer_hex(GATTS_TAG, param->write.value, param-
>write.len);
        if (gl_profile_tab[PROFILE_B_APP_ID].descr_handle == param-
>write.handle && param->write.len == 2){
            uint16_t descr_value= param->write.value[1]<<8 | param-
>write.value[0];
            if (descr_value == 0x0001){
                if (b_property & ESP_GATT_CHAR_PROP_BIT_NOTIFY){
                    ESP_LOGI(GATTS_TAG, "notify enable");
                    uint8_t notify_data[15];
                    for (int i = 0; i < sizeof(notify_data); ++i)
                    {
                         notify_data[i] = i%0xff;
                     }
                     //the size of notify_data[] need less than MTU
size
```

```
                    esp_ble_gatts_send_indicate(gatts_if, param-
>write.conn_id,
gl_profile_tab[PROFILE_B_APP_ID].char_handle,
sizeof(notify_data),
                                                notify_data,
false);
                }
            }else if (descr_value == 0x0002){
                 if (b_property & ESP_GATT_CHAR_PROP_BIT_INDICATE){
                     ESP_LOGI(GATTS_TAG, "indicate enable");
                     uint8_t indicate_data[15];
                     for (int i = 0; i < sizeof(indicate_data); ++i)
                     {
                         indicate_data[i] = i % 0xff;
                      }
                      //the size of indicate_data[] need less than
MTU size
                     esp_ble_gatts_send_indicate(gatts_if, param-
>write.conn_id,
gl_profile_tab[PROFILE_B_APP_ID].char_handle,
sizeof(indicate_data),
                                                indicate_data,
true);
                 }
             }
             else if (descr_value == 0x0000){
                 ESP_LOGI(GATTS_TAG, "notify/indicate disable ");
             }else{
                 ESP_LOGE(GATTS_TAG, "unknown value");
             }
        }
    }
    example_write_event_env(gatts_if, &a_prepare_write_env, param);
    break;
}
```

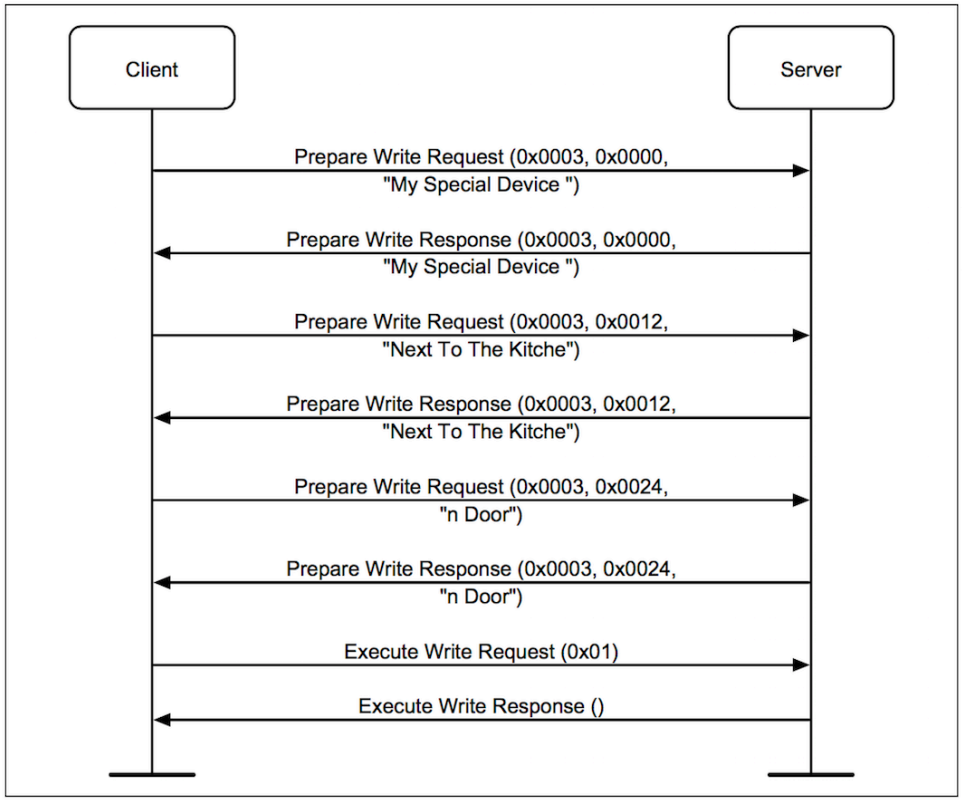

图 16　离线安装包示意图

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

```
void example_write_event_env(esp_gatt_if_t gatts_if,
prepare_type_env_t *prepare_write_env, esp_ble_gatts_cb_param_t
*param){
    esp_gatt_status_t status = ESP_GATT_OK;
    if (param->write.need_rsp){
       if (param->write.is_prep){
            if (prepare_write_env->prepare_buf == NULL){
                prepare_write_env->prepare_buf = (uint8_t
*)malloc(PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
                prepare_write_env->prepare_len = 0;
                if (prepare_write_env->prepare_buf == NULL) {
                    ESP_LOGE(GATTS_TAG, "Gatt_server prep no
mem\n");
                    status = ESP_GATT_NO_RESOURCES;
                }
            } else {
                if(param->write.offset > PREPARE_BUF_MAX_SIZE) {
                    status = ESP_GATT_INVALID_OFFSET;
```

```
                }
                else if ((param->write.offset + param->write.len) >
PREPARE_BUF_MAX_SIZE) {
                    status = ESP_GATT_INVALID_ATTR_LEN;
                }
            }

            esp_gatt_rsp_t *gatt_rsp = (esp_gatt_rsp_t
*)malloc(sizeof(esp_gatt_rsp_t));
            gatt_rsp->attr_value.len = param->write.len;
            gatt_rsp->attr_value.handle = param->write.handle;
            gatt_rsp->attr_value.offset = param->write.offset;
            gatt_rsp->attr_value.auth_req = ESP_GATT_AUTH_REQ_NONE;
            memcpy(gatt_rsp->attr_value.value, param->write.value,
param->write.len);
            esp_err_t response_err =
esp_ble_gatts_send_response(gatts_if, param->write.conn_id,

param->write.trans_id, status, gatt_rsp);
            if (response_err != ESP_OK){
               ESP_LOGE(GATTS_TAG, "Send response error\n");
            }
            free(gatt_rsp);
            if (status != ESP_GATT_OK){
                return;
            }
            memcpy(prepare_write_env->prepare_buf + param-
>write.offset,
                   param->write.value,
                   param->write.len);
            prepare_write_env->prepare_len += param->write.len;

        }else{
            esp_ble_gatts_send_response(gatts_if, param-
>write.conn_id, param->write.trans_id, status, NULL);
        }
    }
}
```

当客户端发送写请求或准备写请求时，服务器应当响应。然而，如果客户端发送一个不需要响应的写命令，服务器不需要回复响应。这通过检查`write.need_rsp`参数的值在写过程中进行检查。如果需要响应，程序继续进行响应准备；如果不存在，客户端不需要响应，因此程序结束。

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

然后，函数检查由`write.is_prep`表示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性写操作。如果存在，程序继续准备多个写响应；如果不存在，那么服务器简单地发送回一个单一的写响应。

```
...
if (param->write.is_prep){
...
}else{
  esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param-
>write.trans_id, status, NULL);
}
...
```

为了处理长特性写操作，定义并实例化了一个准备缓冲区结构：

```
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;
```

为了使用准备缓冲区，为其分配了一些内存空间。如果由于内存不足导致分配失败，将打印一个错误：

```
if (prepare_write_env->prepare_buf == NULL) {
    prepare_write_env->prepare_buf =
    (uint8_t*)malloc(PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
    prepare_write_env->prepare_len = 0;
    if (prepare_write_env->prepare_buf == NULL) {
       ESP_LOGE(GATTS_TAG, "Gatt_server prep no mem\n");
       status = ESP_GATT_NO_RESOURCES;
```

```
    }
}
```

如果缓冲区不是 NULL，这意味着初始化完成，程序然后检查传入写操作的偏移量和消息长度是否适合该缓冲区。

```
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;
  }
}
```

程序现在准备要发送回客户端的类型为`esp_gatt_rsp_t`的响应。响应是使用写请求的相同参数构造的，如长度、句柄和偏移量。此外，设置了写入此特性所需的 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性。一旦响应被发送，为其使用而分配的内存被释放。

```
esp_gatt_rsp_t *gatt_rsp = (esp_gatt_rsp_t
*)malloc(sizeof(esp_gatt_rsp_t));
gatt_rsp->attr_value.len = param->write.len;
gatt_rsp->attr_value.handle = param->write.handle;
gatt_rsp->attr_value.offset = param->write.offset;
gatt_rsp->attr_value.auth_req = ESP_GATT_AUTH_REQ_NONE;
memcpy(gatt_rsp->attr_value.value, param->write.value, param-
>write.len);
esp_err_t response_err = esp_ble_gatts_send_response(gatts_if,
param->write.conn_id,
                                                    param-
>write.trans_id, status, gatt_rsp);
if (response_err != ESP_OK){
  ESP_LOGE(GATTS_TAG, "Send response error\n");
}
free(gatt_rsp);
if (status != ESP_GATT_OK){
  return;
}
```

最后，传入的数据被复制到创建的缓冲区中，并且其长度通过偏移量增加：

```
memcpy(prepare_write_env->prepare_buf + param->write.offset,
       param->write.value,
       param->write.len);
prepare_write_env->prepare_len += param->write.len;
```

客户端通过发送执行写请求来完成长写序列。这个命令触发一个 `ESP_GATTS_EXEC_WRITE_EVT` 事件。服务器通过发送响应并执行 `example_exec_write_event_env()` 函数来处理这个事件：

```
case ESP_GATTS_EXEC_WRITE_EVT:
     ESP_LOGI(GATTS_TAG,"ESP_GATTS_EXEC_WRITE_EVT");
     esp_ble_gatts_send_response(gatts_if, param->write.conn_id,
param->write.trans_id, ESP_GATT_OK, NULL);
     example_exec_write_event_env(&a_prepare_write_env, param);
     break;
```

我们看一下写函数的执行函数

```
void example_exec_write_event_env(prepare_type_env_t
*prepare_write_env, esp_ble_gatts_cb_param_t *param){
    if (param->exec_write.exec_write_flag ==
ESP_GATT_PREP_WRITE_EXEC){
        esp_log_buffer_hex(GATTS_TAG, prepare_write_env-
>prepare_buf, prepare_write_env->prepare_len);
    }
    else{
        ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
    }
    if (prepare_write_env->prepare_buf) {
        free(prepare_write_env->prepare_buf);
        prepare_write_env->prepare_buf = NULL;
    }
####     prepare_write_env->prepare_len = 0;
}
```

执行写操作用于通过长特性写过程确认或取消之前完成的写操作。为了做到这一点，函数检查在事件接收到的参数中的 `exec_write_flag` 。如果标志等于由 `exec_write_flag` 表示的执行标志，写操作被确认，并且缓冲区内容将被打印在日志中；如果不是，则意味着写操作被取消，所有已写入的数据将被删除。

```
if (param->exec_write.exec_write_flag == ESP_GATT_PREP_WRITE_EXEC)
{
   esp_log_buffer_hex(GATTS_TAG,
                      prepare_write_env->prepare_buf,
                      prepare_write_env->prepare_len);
 }
else{
    ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
 }
```

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

```
if (prepare_write_env->prepare_buf) {
    free(prepare_write_env->prepare_buf);
    prepare_write_env->prepare_buf = NULL;
}
prepare_write_env->prepare_len = 0;
```

9.14 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性。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处理长特性写入。