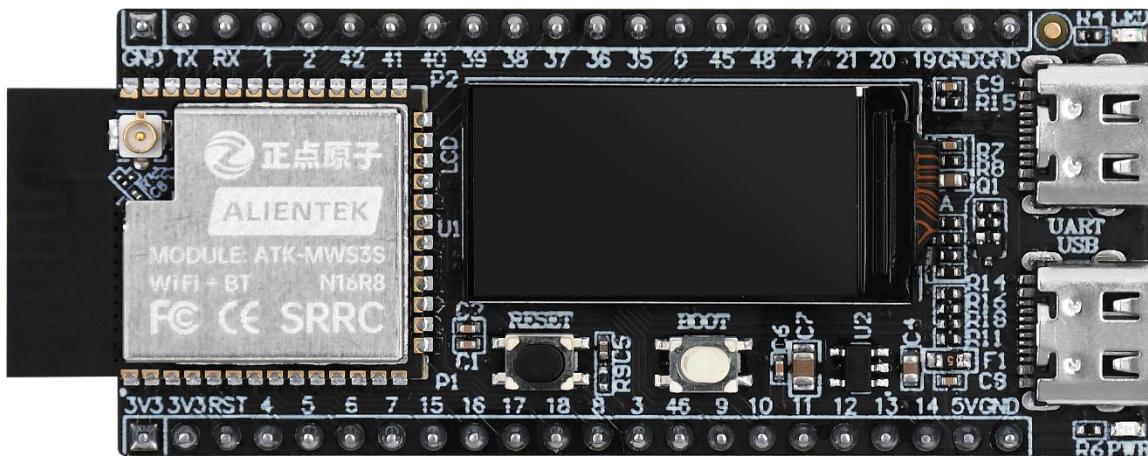


ESP32-S3 使用指南 - Arduino 版本



- 正点原子 DNESP32S3M 最小系统板教程

注：本教程仅适用于 DNESP32S3M 最小系统板

修订历史:

版本	日期	修改内容
V1.0	2024/6/15	第一次发布



正点原子公司名称 : 广州市星翼电子科技有限公司
原子哥在线教学平台 : www.yuanzige.com
开源电子网 / 论坛 : www.openedv.com/forum.php
正点原子官方网站 : www.alientek.com
正点原子淘宝店铺 : <https://openedv.taobao.com>
正点原子 B 站视频 : <https://space.bilibili.com/394620890>

电话: 020-38271790 传真: 020-36773971

请下载原子哥 APP, 数千讲视频免费学习, 更快更流畅。
请关注正点原原子公众号, 资料发布更新我们会通知。



扫码下载“原子哥”APP



扫码关注正点原原子公众号

内容简介	12
第一篇 基础篇	13
第一章 本书学习方法	14
1.1 本书学习顺序	14
1.2 本书参考资料	14
1.3 本书编写规范	14
1.4 本书代码规范	15
1.5 例程资源说明	15
1.6 学习资料查找	16
1.7 给初学者的建议	17
第二章 Arduino 基础知识	19
2.1 什么是 Arduino	19
2.2 Arduino 的由来	20
2.3 Arduino 的优势	20
2.4 Arduino 语言	21
2.5 Arduino 程序结构	21
第三章 C/C++语言基础	23
3.1 数据类型	23
3.2 运算符	24
3.3 表达式	24
3.4 数组	24
3.5 字符串	25
3.6 注释	25
3.7 顺序结构	25
3.8 选择结构	26
3.9 循环结构	27
第四章 ESP32-S3 基础知识	29
4.1 为什么选择 ESP32-S3	29
4.2 初识 ESP32-S3	30
4.3 ESP32-S3 资源简介	31
4.4 S3 系列型号对比	31
4.5 ESP32-S3 功能概述	33
4.5.1 系统和存储器	33
4.5.2 IO MUX 和 GPIO 交换矩阵	33
4.5.3 复位与时钟	35

4.5.4 芯片 Boot 控制	38
4.6 ESP32-S3 启动流程	40
第五章 Arduino 开发环境搭建	43
5.1 开发方式的选择	43
5.2 环境搭建	44
5.2.1 Arduino IDE2 软件安装包下载	44
5.2.2 Arduino IDE2 软件安装	46
5.2.3 认识 Arduino IDE2	49
5.2.4 arduino-esp32 库介绍	53
5.2.5 安装 arduino-esp32 库	53
第六章 新建 Arduino 工程	57
6.1 使用 Arduino IDE2 新建工程	57
6.2 ESP32-S3 Arduino 工程设置	59
6.3 运行 ESP32-S3 Arduino 第一个工程	62
第二篇 入门篇	66
第七章 LED 实验	67
7.1 GPIO 介绍	67
7.1.1 ESP32-S3 的 GPIO 简介	67
7.1.2 GPIO 函数介绍	68
7.2 硬件设计	69
1. 例程功能	69
2. 硬件资源	69
3. 原理图	69
7.3 软件设计	69
7.3.1 程序流程图	69
7.3.2 程序解析	70
7.4 下载验证	71
第八章 KEY 实验	72
8.1 GPIO 输入功能使用	72
8.1.1 GPIO 输入模式介绍	72
8.1.2 独立按键简介	74
8.2 硬件设计	74
1. 例程功能	74
2. 硬件资源	75
3. 原理图	75

8.3 软件设计	75
8.3.1 程序流程图	75
8.3.2 程序解析	75
8.4 下载验证	77
第九章 EXTI 实验	78
9.1 外部中断介绍	78
9.1.1 中断程序	78
9.1.2 ESP32-S3 的中断介绍	78
9.1.3 中断触发模式	79
9.1.4 中断触发函数介绍	80
9.2 硬件设计	80
1. 例程功能	80
2. 硬件资源	80
3. 原理图	80
9.3 软件设计	80
9.3.1 程序流程图	80
9.3.2 程序解析	81
9.4 下载验证	82
第十章 UART 实验	83
10.1 串口介绍	83
10.1.1 数据通信的基本概念	83
10.1.2 UART 介绍	84
10.1.3 串口相关函数介绍	85
10.2 硬件设计	87
1. 例程功能	87
2. 硬件资源	88
3. 原理图	88
10.3 软件设计	88
10.3.1 程序流程图	88
10.3.2 程序解析	89
10.4 下载验证	90
第十一章 TIMER_IT 实验	91
11.1 定时器简介	91
11.1.1 定时器介绍	91

11.1.2 定时器函数介绍	91
11.2 硬件设计	92
1. 例程功能	92
2. 硬件资源	92
3. 原理图	92
11.3 软件设计	92
11.3.1 程序流程图	92
11.3.2 程序解析	93
11.4 下载验证	94
第十二章 LED_PWM 实验	95
12.1 LED PWM 控制器介绍	95
12.1.1 PWM 介绍	95
12.1.2 LED_PWM 控制器介绍	96
12.1.3 LED_PWM 函数介绍	96
12.2 硬件设计	97
1. 例程功能	97
2. 硬件资源	97
3. 原理图	97
12.3 软件设计	97
12.3.1 程序流程图	97
12.3.2 程序解析	98
12.4 下载验证	99
第十三章 SPI_LCD 实验	100
13.1 SPI 及 LCD 介绍	100
13.1.1 SPI 介绍	100
13.1.2 SPI 控制器介绍	101
13.1.3 LCD 介绍	102
13.1.4 TFT_eSPI 库介绍	105
13.2 硬件设计	109
1. 例程功能	109
2. 硬件资源	109
3. 原理图	109
13.3 软件设计	110
13.3.1 程序流程图	110

13.3.2 程序解析	110
13.4 下载验证	112
第十四章 RTC 实验	113
14.1 RTC 介绍	113
14.2 硬件设计	115
1. 例程功能	115
2. 硬件资源	115
3. 原理图	115
14.3 软件设计	115
14.3.1 程序流程图	115
14.3.2 程序解析	116
14.4 下载验证	117
第十五章 INTERNAL_TEMPERATURE 实验	118
15.1 内部温度传感器介绍	118
15.1.1 内部温度传感器简介	118
15.1.2 内部温度传感器接口函数介绍	118
15.2 硬件设计	119
1. 例程功能	119
2. 硬件资源	119
3. 原理图	119
15.3 软件设计	119
15.3.1 程序流程图	119
15.3.2 程序解析	120
15.4 下载验证	121
第十六章 SPI_SDCARD 实验	122
16.1 SD 卡介绍	122
16.1.1 SD 卡介绍	122
16.1.2 SD 卡库介绍	122
16.2 硬件设计	124
1. 例程功能	124
2. 硬件资源	124
3. 原理图	124
16.3 软件设计	125
16.3.1 程序流程图	125

16.3.2 程序解析	125
16.4 下载验证	128
第三篇 高级篇	130
第十七章 WIFI_SCAN 实验	131
17.1 WiFi 工作模式和网络扫描介绍	131
17.1.1 ESP32-S3 WiFi 工作模式	131
17.1.2 ESP32-S3 网络扫描介绍	132
17.1.3 WiFiScan 库介绍	132
17.2 硬件设计	134
1. 例程功能	134
2. 硬件资源	134
3. 原理图	134
17.3 软件设计	134
17.3.1 程序流程图	134
17.3.2 程序解析	135
17.4 下载验证	136
第十八章 WIFI_WEB SERVER 实验	138
18.1 网络基础知识和 WEB SERVER 函数介绍	138
18.1.1 互联网络和 TCP/IP 协议	138
18.1.2 IP 地址	138
18.1.3 端口号	139
18.1.4 客户端-服务器模式	139
18.1.5 HTTP 协议	140
18.1.6 ESP32-S3 Web 服务器	140
18.1.7 URL 和域名、IP 之间的关系	141
18.1.8 ESP32-S3 WEB SERVER 介绍	141
18.2 HTML 基础	143
18.2.1 HTML 文档基本结构	143
18.2.2 HTML 标签	144
18.3 硬件设计	144
1. 例程功能	144
2. 硬件资源	145
3. 原理图	145
18.4 软件设计	145

18.4.1 程序流程图	145
18.4.2 程序解析	146
18.5 下载验证	148
第十九章 WIFI_CLIENT 实验	149
19.1 CLIENT 函数介绍	149
19.2 硬件设计	149
1. 例程功能	149
2. 硬件资源	149
3. 原理图	150
19.3 软件设计	150
19.3.1 程序流程图	150
19.3.2 程序解析	150
19.4 下载验证	152
第二十章 BLE_SCAN 实验	153
20.1 蓝牙基础知识和 BLEScan 介绍	153
20.1.1 蓝牙介绍	153
20.1.2 蓝牙协议介绍	153
20.1.3 工作状态和工作角色介绍	154
20.1.4 蓝牙设备链接建立过程	155
20.1.5 蓝牙扫描介绍	156
20.1.6 BLEScan 库函数介绍	157
20.2 硬件设计	158
1. 例程功能	158
2. 硬件资源	158
3. 原理图	158
20.3 软件设计	158
20.3.1 程序流程图	158
20.3.2 程序解析	159
20.4 下载验证	160
第二十一章 BLE_UART 实验	162
21.1 蓝牙基础知识和蓝牙通信函数介绍	162
21.1.1 属性介绍	162
21.1.2 属性协议 AAT 介绍	163
21.1.3 通用属性协议 GAAT 介绍	163

21.1.4 ESP32 GATT 介绍	164
21.1.5 蓝牙通信函数介绍	164
21.2 硬件设计	166
1. 例程功能	166
2. 硬件资源	166
3. 原理图	167
21.3 软件设计	167
21.3.1 程序流程图	167
21.3.2 程序解析	167
21.4 下载验证	170

内容简介

本教程旨在为读者提供一本基于 Arduino IDE 环境，开发 ESP32-S3 的编程指南，从入门到进阶，带领读者逐步掌握使用 Arduino IDE 去进行 ESP32-S3 开发与应用。

本书总共分为三个部分：

第一部分：基础篇

本部分将引导读者逐步了解 Arduino 与微控制器 ESP32 之间的联系。此外，我们还将介绍 ESP32-S3 的基本知识、C/C++语言基础以及搭建 Arduino IDE 开发环境。通过学习本篇，读者将掌握 ESP32-S3 在 Arduino IDE 上的使用，为后续的学习和实践打下坚实的基石。

第二部分：入门篇

本部分适合接触过 Arduino 编程基础的人，将会介绍如何使用乐鑫提供的 arduino-esp32 库对 ESP32-S3 的硬件接口进行驱动。本篇的学习会涉及到 ESP32-S3 的 GPIO 接口、SPI 接口。通过本篇的学习，读者将掌握 Arduino 库包的使用，以及如何快速在 ESP32-S3 上实现外设驱动。

第三部分：提高篇

入门篇的内容是把 ESP32-S3 当做一个普通的 MCU 进行使用，而提高篇专门讲解芯片的 WIFI 和蓝牙外设功能，让大家了解 ESP32-S3 的通信功能。

本教程结构清晰、内容丰富、实用性强，适合创客和 ESP32-S3 的初学者、进阶者和开发者阅读参考。同时，本书还提供了完整的配套资源，包括视频教程和代码示例等，方便读者学习和实践。

第一篇 基础篇

万事开头难，如果打好了基础，那么后面的学习就事半功倍了！本篇将详细介绍 Arduino 相关知识，让大家捋清楚 Arduino 与微控制器 ESP32 的关系。此外还会讲解 Arduino 语言编程基础以及 ESP32-S3 基础知识，了解如何遵循 Arduino IDE 编程规则对 ESP32-S3 进行开发。

如果您是初学者，建议好好学习本并理解这些知识，手脑并用，不要漏过任何内容，一遍学不会的可以多学几遍，总之这些知识点都要掌握。

如果您已经学过 Arduino 和 ESP32-S3 了，本篇内容则可以挑选着学习。

本篇将分为如下章节：

- 1, 本书学习方法
- 2, Arduino 基础知识
- 3, C/C++语言基础
- 4, ESP32-S3 基础知识
- 5, Arduino 开发环境搭建
- 6, 新建 Arduino 工程

第一章 本书学习方法

为了让大家更好的学习和使用本书，本章将给大家介绍一下本书的学习方法，包括：本书的学习顺序、编写规范、代码规范、资料查找、学习建议等内容。

本章将分为如下几个小节：

- 1.1 本书学习顺序
- 1.2 本书参考资料
- 1.3 本书编写规范
- 1.4 本书代码规范
- 1.5 例程资源说明
- 1.6 学习资料查找
- 1.7 给初学者的建议

1.1 本书学习顺序

为了让大家更好的学习和使用本书，我们做了以下几点考虑：

- 1，坚持循序渐进的思路编写，从基础到入门，从简单到复杂。
- 2，将知识进行分类介绍，简化学习过程。
- 3，将板卡硬件资源介绍独立成一个文档《DNESP32S3M 硬件参考手册.pdf》。

因此，读者在学习本书的时候，我们建议：先通读一遍《DNESP32S3M 硬件参考手册.pdf》，对板卡的硬件资源有个大概了解，然后从本书的基础开始，再到入门篇，最后学习提高篇，循序渐进，逐一攻克。

对初学者来说，尤其要按照以上顺序学习，不要跳跃式学习，因为我们书本的知识都是一环扣一环的，如果前面的知识没学好，后面的知识学起来就会很困难。

对于已经有了一定单片机基础的读者，就可以跳跃式学习了。当然了，若是遇到不懂的知识点，也得翻阅前面的知识点进行巩固。

1.2 本书参考资料

本书的主要参考资料有以下两份文档：

数据手册：《esp32-s3-wroom-1_wroom-1u_datasheet_cn.pdf》

技术规格书：《esp32-s3_technical_reference_manual_cn.pdf》

前者是乐鑫官方针对 S3 系列 ESP32 提供的数据手册，该数据手册提供了关于这些微控制器的详细信息，包括它们的特性、性能指标、引脚布局、电路原理图以及其他相关的技术文档。这对于开发人员、工程师和爱好者来说是非常有用的，可以帮助他们了解和使用这些微控制器，以及设计相关的嵌入式和物联网应用。

后者是乐鑫官方针对 S3 系列 ESP32 提供的技术参考手册，该技术参考手册包含了对 Xtensa32 位双内核和其使用的指令集、寄存器、外设描述等的知识。

以上提及的两份文档也是读者在学习本书的过程中必不可少的参考资料，读者可以在“A 盘→8，ESP32-S3 参考资料”中找到这两份文档。

1.3 本书编写规范

本书通过数十个例程，给大家详细介绍 ESP32-S3 的常用功能和外设，按难易程度以及知识结构，我们将本书分为三个篇章：基础篇、入门篇和提高篇。

基础篇，共 6 章，主要是一些基础知识介绍，包括芯片介绍以及开发环境搭建等，这些章节在结构上没有共性，但互相有关联，有一个集成的关系在里面，即：必须先学了前面的只是，才好学习后面的知识点。

入门篇和提高篇，共 15 章，介绍了如何使用乐鑫提供 arduino-esp32 库对 ESP32-S3 的外设

进行快速开发。这些章节在结构上都比较有共性，一般分为 4 个部分，如下：

- 1, 外设介绍
- 2, 硬件设计
- 3, 程序设计
- 4, 下载验证

外设介绍，简单介绍具体章节所使用的外设，让读者对该外设有一个基本了解，便于后面的程序设计。

硬件设计，包括实验例程实现的功能说明、使用的硬件资源和其原理图。读者可以清晰了解实验例程要做什么？用哪些硬件资源以及如何连接这些硬件资源。这样有利于编写驱动代码和应用代码的人程序设计。

程序设计，通常包括简洁的介绍驱动、配置步骤、关键代码解析和 main 函数讲解等，以逐步介绍程序代码和构建和注意事项，让读者深入了解整个程序代码。

下载验证，是验证程序设计的实践步骤，通过下载并验证程序是否按照预期工作形成一个闭环。

1.4 本书代码规范

为了方便大家编写高质量代码，我们对本书的代码风格进行了统一。

总结几个规范的关键点：

- 1, 所有函数/变量名字非特殊情况，一般使用小写字母；
- 2, 注释风格使用 doxygen 风格，除屏蔽外，一律使用 `/* */` 方式进行注释；
- 3, TAB 键统一使用 4 个空格对齐，不使用默认的方式进行对齐；
- 4, 每两个函数之间，一般有且只有一个空行；
- 5, 相对独立的程序块之间，使用一个空行隔开；
- 6, 全局变量命名一般用 `g_` 开头，全局指针命名一般用 `p_` 开头；
- 7, `if`、`for`、`while`、`do`、`case`、`switch`、`default` 等语句单独占一行，一般无论有多少行执行语句，都要用加括号：`{}`。

1.5 例程资源说明

在 ESP32-S3 最小系统板的配套资料中，提供了 10 个基础例程和 5 个通信例程。这些例程都是基于 arduino-esp32 库和一些 ESP32-S3 相关的库进行编写的。这些例程大部分是原创的，并附有详细的注释，代码风格统一，内容循序渐进，非常适合初学者入门。

ESP32-S3 最小系统板的基础例程列表如表 1.5.1 所示：

编号	实验名字	编号	实验名字
1	01_led	6	06_led_pwm
2	02_key	7	07_spi_lcd
3	03_exit	8	08_rtc
4	04_uart	9	09_internal_temperature
5	05_timer_it	10	10_spi_sdcard

表 1.5.1 ESP32-S3 最小系统板基础例程表

ESP32-S3 最小系统板的通信例程列表如表 1.5.2 所示：

编号	实验名字	编号	实验名字
1	01_wifi_scan	4	04_ble_scan
2	02_wifi_webserver	5	05_ble_uart
3	03_wifi_client		

表 1.5.2 ESP32-S3 最小系统板通信例程表

从上表可以看出，正点原子 ESP32-S3 最小系统板的例程基本上涵盖了 ESP32-S3 芯片的大

部分外设包括 WIFI 和 BLE。而且从上表可以看出，例程安排是循序渐进的，首先从最基础的跑马灯开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握。

所以，正点原子 ESP32-S3 最小系统板是非常适合初学者以及 DIY 爱好者。

1.6 学习资料查找

如果您想查找有关使用 Arduino 进行 ESP32-S3 开发的资料，可以尝试以下方法：

1, Arduino 官方学习资料

(1) Arduino 官方组织的资料

Arduino 网址：www.arduino.cc，在其官网上提供了 Arduino 开发板的使用办法以及 Arduino 函数说明。具体查找方式如下方 2 个图所示。首先通过点击“DOCUMENTATION”选项卡里面的“ARDUINO DOCS”，进入到文档界面，然后可通过“TUTORIALS”选项卡进入到搜索界面进行 ESP32 搜索，搜索结果就是 Arduino 官方的 Nano ESP32 开发板一些资料。由于 Nano ESP32 开发板也是使用到 ESP32S3，所以也就可以参考。

注意：由于 arduino 官网经常性更新，当我们点击“DOCUMENTATION”选项卡进入到的是文档页面，网址为 <https://docs.arduino.cc/>，页面可能不是下图所展示的。不过没有关系，直接在该页面下找到“TUTORIALS”选项，点击即可进入到图 1.6.2 展示的页面。

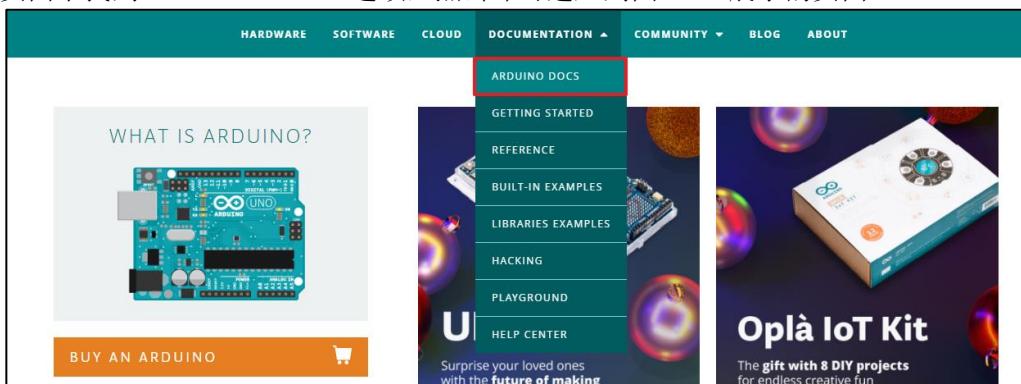


图 1.6.1 Arduino 官网进入到文档界面步骤

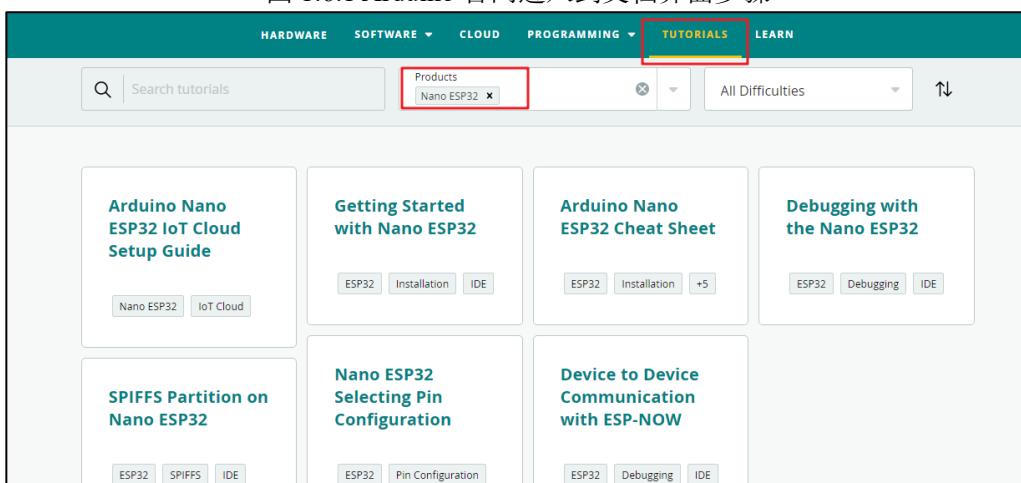


图 1.6.2 Arduino 官网 Nano ESP32 开发板资料

注意：资料中的“Arduino Nano ESP32 Cheat Sheet”就是对该款开发板的介绍。当然，在官网上能搜索到众多 Arduino 官方开发板，大家有兴趣的可以自行去查看。

(2) 乐鑫提供的 arduino_esp32 库的资料

arduino_esp32 库包的网址：<https://github.com/espressif/arduino-esp32>，这是乐鑫公司专门为 ESP32 能在 Arduino IDE 软件上运行专门编写的库包。后面在 Arduino IDE 上开发 ESP32 是依靠该库包，并不是 Arduino 官方提供的 Arduino ESP32 Boards 软件包。Arduino 语言函数接口具有统一性，所以 Arduino 官方资料也有一定的参考价值。

2, 正点原子的学习资料

正点原子提供了大量的学习资料，为方便读者下载所有正点原子最新最全的学习资料，这些资料都放在[正点原子文档中心](#)，如下图所示（正点原子文档中心会不时地更新，以保证为读者提供最新的学习资料）：



图 1.6.3 正点原子文档中心

在正点原子文档中心中，可以找到正点原子所有开发板、模块、产品等详细资料下载链接。

3, 正点原子论坛

[正点原子论坛](#)，即开源电子网，该论坛从 2010 年成立至今，已有十多年的时间，拥有数十万的注册用户和大量嵌入式相关的帖子，每天有数百人互动，是一个非常好的嵌入式学习交流平台。

4, 博客和教程网站

在互联网上搜索与 ESP32-S3 和 Arduino 相关的博客和教程网站。这些网站通常会提供详细的步骤和示例代码，帮助您逐步掌握 ESP32-S3 的开发技巧。

5, 视频教程

在 B 站等视频平台上搜索与 ESP32-S3 和 Arduino 相关的教程视频。这些视频可以直观地展示开发过程和示例代码的执行效果，有助于初学者快速入门。

6, 在线课程和教育资源

寻找与 ESP32-S3 和 Arduino 相关的在线课程和教育资源，例如在线教程、视频课程、教科书等。这些资源通常由教育机构、专业网站或个人开发者提供。

总之，通过以上方法，您可以找到大量与 ESP32-S3 和 Arduino 开发相关的资料。在查找和学习过程中，请注意选择可靠和最新的资源，并根据自己的需求和水平进行选择和学习。

1.7 给初学者的建议

对于学习 ESP32，这里我给大家提以下三点建议：

1, 准备开发板：选择适合的开发板，并配备调试接口，以便在实际开发板上运行和调试程序。这有助于加深对程序执行过程的理解，并方便查找和解决错误。

2, 阅读参考资料：《esp32-s3-wroom-1_wroom-1u_datasheet_cn.pdf》、《esp32-s3_technical_reference_manual_cn.pdf》和《isa-summary.pdf》是学习 ESP32-S3 的重要参考资料。这些手册对于理解 ESP32-S3 和 Xtensa® LX7 内核有很大帮助，尤其是对于初学者，需要多看多了解。

3, 保持耐心和积极态度：学习 ESP32-S3 需要时间和耐心，遇到问题和难点时不能气馁或

逃避。尝试自己解决问题，掌握解决问题的技巧和方法。同时要勤于思考和实践，举一反三，通过实践来加深理解和掌握知识。如果 Arduino 和 C 语言基础不够扎实，建议先学习 Arduino 和 C 语言基础，以便更好理解和掌握 ESP32-S3 相关知识。

第二章 Arduino 基础知识

本章，我们将向大家介绍 Arduino 是一个什么东西？让大家对 Arduino 有一个大概了解。本章将分为如下几个小节：

- 2.1 什么是 Arduino
- 2.2 Arduino 的由来
- 2.3 Arduino 的优势
- 2.4 Arduino 语言
- 2.5 Arduino 程序结构

2.1 什么是 Arduino

Arduino 即为开源硬件。可以说 Arduino 从真正意义上推动了开源硬件的发展，在 Arduino 出现以前，虽然也有很多公司在推广一些简单易用的可编程控制器，但是由于开发平台种类繁多，而且使用这些控制器基本上都需要对电子技术、数字逻辑、寄存器等内容进行多方面的了解和学习，才能完成一个电子产品的制作。这就给开源硬件的推广和普及设定了一个很高的门槛，电子爱好者需要花很多时间和精力才能开始开发和制作自己的作品。而使用 Arduino 能很快地完成一个电子产品的制作，这是由于 Arduino 提供了一个开放易学，进入门槛相对较低的开发平台，让电子爱好者对于开源硬件的广泛使用变成了可能。

广泛来说，Arduino 指的是一个生态，这里就包括 Arduino 开发板、Arduino IDE 以及周边资料，其中包括社区、驱动库以及示例代码等，如下图 2.1.1 所示：

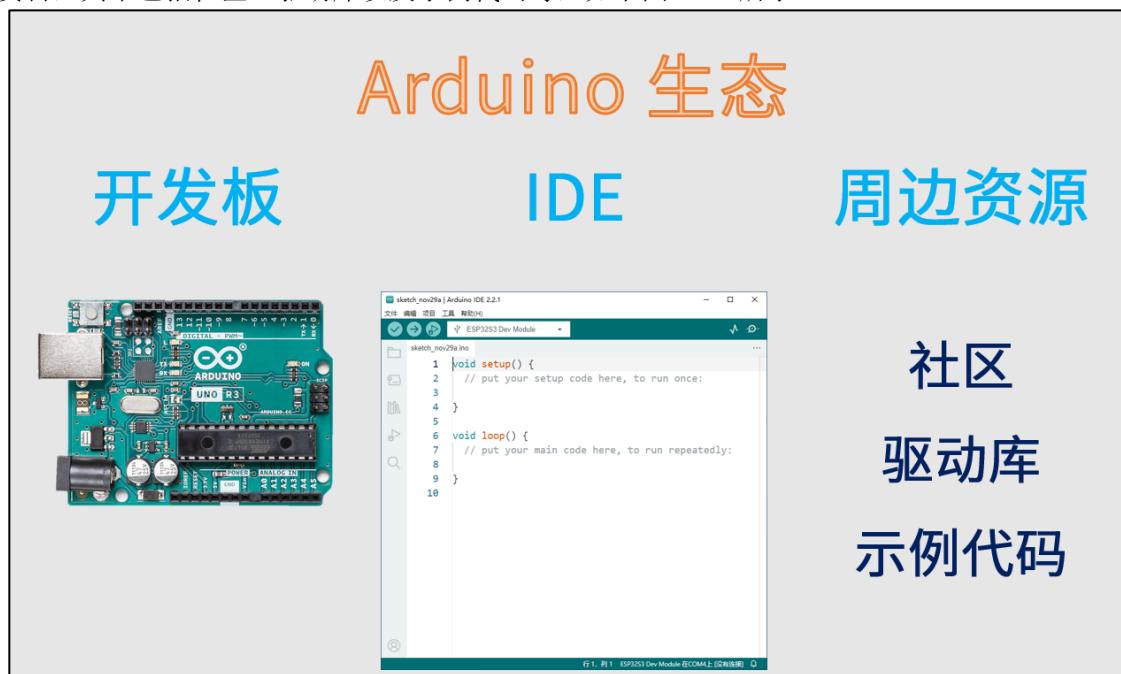


图 2.1.1 Arduino 生态图

现今，Arduino 开发板有很多，正点原子 ESP32-S3 最小系统板也是属于 Arduino 开发板，简单来说，只要是搭载 Arduino 支持芯片的开发板都可以称为 Arduino 开发板。目前支持 Arduino 开发的芯片有很多，比如 Mega 系列芯片(Mega328p/Mega2560/Mega32u4 等)、STM32 系列芯片(STM32F0/F1/F2/F3/F4/F7/H7 等)、ESP 系列芯片(esp32/esp32s2/esp32c3/esp32s3 等)以及树莓派系列芯片等。这里的实质就是有每个系列芯片对应要有一个 Arduino 库，比如乐鑫官方的 arduino-esp32 库，ST 的 stm32duino 库，只要在 Arduino 安装这种芯片库便可以使用 Arduino 的语法在 Arduino IDE 上对芯片进行开发。

Arduino IDE 是 Arduino Integrated Development Environment 集成开发环境的简称。本教程

就是基于该软件对 ESP32-S3 进行开发。该 IDE 比较遗憾的是没有调试功能，在后面“Arduino 开发环境搭建”章节会对这个软件进行讲解。

得益于广大的电子爱好者对于开源硬件的贡献，Arduino 的社区活跃度很高，驱动库种类繁多而且有些一直存在更新迭代，不断完善，示例代码功能也越来越健全。使用 Arduino IDE 进行开发，由于驱动库是相互兼容的，很多代码不需要修改就可以直接使用，这就可以让从芯片替代成本变得很低，也让越来越多电子爱好者参与 Arduino 生态的建设。

2.2 Arduino 的由来

Arduino 是一种基于开源硬件和软件的电子原型平台，它由一个意大利的团队于 2005 年创建。该团队的成员包括 Massimo Banzi、David Cuartielles、Carlo Galoppini 和 Michael Margolis，如下图 2.2.1 所示。



图 2.2.1 Arduino 创始团队

在 2005 年冬天，Massimo Banzi 和 David Cuartielles 为了解决找不到既便宜又好用的微控制器的现状，两人决定设计自己的电路板，并且在这期间吸收了 Banzi 学生 David Mellis 为电路板设计编程语言。两天以后，David Mellis 就写出了程序代码。又过了三天，电路板就完工了。

据说这个 Arduino 名字的由来是，Massimo Banzi 喜欢去一家名叫 di Re Arduino 的酒吧，这酒吧是以 1000 年前意大利国王 Arduin 的名字命名的，为了纪念这个地方，他将这块电路板命名为 Arduino。

随后，Banzi、Cuartielles 和 Mellis 把设计图放到了网上。版权法可以监管开源软件，却很难用在硬件上，为了保持设计的开放源码理念，他们决定采用 Creative Commons(CC)的授权方式公开硬件设计图。在这样的授权下，任何人都可以生产电路板的复制品，甚至还能重新设计和销售原设计的复制品。人们不需要支付任何费用，甚至不用取得 Arduino 团队的许可。然而，如果重新发布了引用设计，就必须声明原始 Arduino 团队的贡献。如果修改了电路板，则最新设计必须使用相同或类似的 Creative Commons(CC)的授权方式，以保证新版本的 Arduino 电路板也会一样是自由和开放的。唯一被保留的只有 Arduino 这个名字，它被注册成了商标，在没有官方授权的情况下不能使用它。短短的几年时间，Arduino 在全球积累了大量用户，推动了开源硬件、创客运动，甚至是硬件创业领域的发展，越来越多的芯片厂商和开发公司宣布自己的硬件支持 Arduino。

2.3 Arduino 的优势

使用 Arduino 去开发硬件，技术门槛很低。几乎任何人，即使不懂电脑编程也能用 Arduino 做出很酷的东西，比如点灯，控制马达，对传感器进行回应。

简单来说，使用 Arduino 去开发优势非常明显，可以归纳为一下 5 点：

1、跨平台

Arduino IDE 可以在 Windows、Mac OS 和 Linux 三大主流操作系统上运行，而其他的大多数控制器只能在 Windows 上开发。

2、简单清晰的开发

Arduino IDE 基于 Processing IDE 开发，这对于开发板来说极易掌握，同时又有足够的灵活

性。Arduino 语言是基于 Wiring 语言开发的，是对 AVR-GCC 库的二次封装，并不需要太多的单片机基础和编程基础，只要简单的学习后就可以快速地进行可开发。

3、开放性

Arduino 的硬件原理图、电路图、IDE 软件及核心库文件都是开源的，在开源协议范围内可以任意修改原始设计及相应代码。

4、社区和第三方支持

Arduino 有着众多的开发者和用户，因此可以找到他们提供的众多开源的示例代码和硬件设计。例如，可以在 Github.com、Arduino.cc、Arduino.me 等网站上找到 Arduino 的第三方硬件、外设和类库等支持，以便更快、更简单地扩展自己的 Arduino 项目。

5、硬件开发趋势

Arduino 不仅仅是全球最流行的开源硬件，也是一个优秀的硬件开发平台，更是硬件开发的趋势。Arduino 简单的开发方式使得开发者更关注于创意和实现，可以更快地完成自己的项目开发，大大节约学习的成本，缩短开发周期。

鉴于 Arduino 的种种优势，越来越多的专业硬件开发者已经或开始使用 Arduino 来开发项目和产品；越来越多的软件开发者使用 Arduino 进入硬件、物联网等开发领域；在大学里，自动化、软件专业，甚至艺术专业，也纷纷开设了 Arduino 相关课程。

2.4 Arduino 语言

Arduino 使用 C/C++语言编写程序，虽然 C++兼容 C 语言，但是这两种语言又有所区别。C 语言跟 C++语言最大的区别在于：C 语言是一种面向过程的编程语言，而 C++是一种面向对象的编程语言。早期的 Arduino 核心库使用 C 语言编写，后面引进了面向对象的思想，目前最新的 Arduino 核心库采用 C 与 C++混合编程。

通常所说的 Arduino 语言，是指 Arduino 核心库文件提供的各种应用程序编程接口（Application Programming Interface，简称 API）的集合。这些 API 是对更底层的单片机支持库进行二次封装所形成的。例如，使用 ESP 单片机的 Arduino 核心库是对 ESP-IDF 库的二次封装。

在使用 ESP-IDF 对 ESP32 进行开发中，将一个 I/O 口设置为输出高电平状态需要以下操作：

```
gpio_set_direction(GPIO_NUM_1, GPIO_MODE_OUTPUT);
gpio_set_level(GPIO_NUM_1, PIN_SET);
```

在 ESP-IDF 中，这个代码架构是可以减少对底层的理解要求，不需要明白寄存器的意义及其之间的关系，来达到配置多个寄存器来达到目的。但是在别的芯片中，比如传统的 AVR 芯片以及 STM32 芯片中，这是需要去理解寄存器的。

在 Arduino 中，我们采用的是如下代码去设置 IO 口输出高电平。

```
pinMode(1, OUTPUT);
digitalWrite(1, HIGH);
```

这里的 pinMode 即是设置引脚的模式，这里设定了 1 脚为输出模式；而 digitalWrite(1, HIGH) 则是使 1 脚输出高电平数字信号。这些封装好的 API 使得程序中的语句更容易被理解，因此可以不用理会单片机中繁杂的寄存器配置就能直观地控制 Arduino，在增强了程序可读性的同时，也提高了开发效率。

若在使用时存在 arduino 库无法实现对芯片的外设驱动时，也可以考虑包含 esp-idf 函数所在头文件对里面函数进行调用，来完成该驱动的实现。

2.5 Arduino 程序结构

在 Arduino 程序中，是没有 main 函数的，这跟传统的 C/C++程序结构有所不同。

其实并不是 Arduino 程序中没有 main 函数，而是 main 函数的定义隐藏在了 Arduino 的核心库文件中，这个是需要我们去到核心库中才能找到 main 函数的踪影。在进行 Arduino 开发时一般不直接操作 main 函数，而使用 setup()和 loop()这两个函数。

接下来，看一下 Arduino 程序的基本结构，如下：

```
void setup() {
```

```
// 在这里填写 setup 函数代码, 它只会运行一次
}

void loop() {
    // 在这里编写 loop 函数代码, 它会不断重复运行
}
```

Arduino 程序的基本结构由 setup()和 loop()两个函数组成。

1、setup()

Arduino 控制器通电后或复位后, 会开始执行 setup()函数中的程序, 该程序只会执行一次。

通常是在 setup()函数中完成 Arduino 的初始化设置, 如配置 I/O 口状态和初始化串口等操作。

2、loop()

setup()函数中的程序执行完毕后, Arduino 会接着执行 loop()函数中的程序。而 loop()函数是一个死循环, 其中的程序会不断地重复运行。

通常在 loop()函数中完成程序的主要功能, 如驱动各种传感器获取数据等。

第三章 C/C++语言基础

C/C++语言是国际上广泛流行的计算机高级语言。在进行绝大多数的硬件开发时，均使用C/C++语言，Arduino也不例外。使用Arduino时需要有一定的C/C++基础，由于篇幅有限，在此仅对C/C++语言基础进行简单介绍。本章将分为如下9个小节：

- 3.1 数据类型
- 3.2 运算符
- 3.3 表达式
- 3.4 数组
- 3.5 字符串
- 3.6 注释
- 3.7 顺序结构
- 3.8 选择结构
- 3.9 循环结构

3.1 数据类型

在C/C++语言程序中，对所有数据都必须指定其数据类型。

数据有常量和变量。需要注意的是，Arduino中的部分数据类型与计算机中的有所不同。

1. 常量

在程序运行过程中，其值不能改变的量称为常量。常量可以是字符，也可以是数字，通常使用以下方式去使用。

方式1：

```
#define 常量名 常量值
如在Arduino核心库中已经定义的常量PI，即是使用以下语句定义的。
```

```
#define PI 3.1415926535897932384626433832795
```

方式2：

```
const 数据类型 常量名 = 常量值;
如定义一个常量PI，可以使用以下语句定义。
```

```
const float PI = 3.14159;
```

2. 变量

程序中可变的值称为变量。其定义方法是：

数据类型 变量名；

例如，定义一个整型变量i的语句是：

```
int i;
```

可以在定义变量的同时为其赋值，也可以在定义之后，再对其赋值，例如：

```
int i = 88;
```

和以下的两条语句是一样的效果，这两者是等效的。

```
int i; i = 95;
```

(1) 整型

整型即整数类型。Arduino可使用的数据类型及其取值范围如表3.1.1所示。

类型	关键字(简写)	占用内存	取值范围
短整型	int16_t (short)	2字节	-32768 ~ 32767
无符号短整型	uint16_t (unsigned short)	2字节	0 ~ 65535
整型	int32_t (int)	4字节	-2147483648 ~ 2147483647
无符号整型	uint32_t (unsigned int)	4字节	0 ~ 4294967295
长整型	int64_t (long long)	8字节	-9223372036854775808 ~ 9223372036854775807
无符号长整型	uint64_t (unsigned long long)	8字节	0 ~ 18446744073709551615

表3.1.1 数据类型表

(2) 浮点型

浮点数其实就是平常所说的实数。在 Arduino 中有 float 和 double 两种浮点类型，在 ESP32-S3 作为主控芯片的 Arduino 开发板上，float 类型占用 4 字节内存空间，double 类型占用 8 字节内存空间。但在某些 Arduino 开发板中，这两种类型占用内存空间可能存在差异。

浮点型数据的运算较慢且有一定误差，因此，通常会把浮点型转换为整型来处理相关运算。比如，9.8cm，通常会换算成 98mm 来计算。

(3) 字符型

字符型，即为 char 类型，其占用 1 字节的内存空间，主要用于存储字符变量。在存储字符时，字符需要用单引号引用，如下所示：

```
char g_char = 'A';
```

字符都是以数字形式存储在 char 类型变量中的，数字与字符的对应关系即平常说的 ASCII 码表，如下图所示。

(4) 布尔型

布尔型变量即 bool 类型。它的值只有两个：false(假)和 true(真)。bool 类型会占用 1 字节的内存空间。

3.2 运算符

C/C++语言中有多种类型的运算符，常见运算符如表 3.2.1 所示：

运算符类型	运算符	说明
算法运算符	=	赋值
	+	加
	-	减
	*	乘
	/	除
	%	取模
比较运算符	==	等于
	!=	不等于
	<	小于
	>	大于
	<=	小于或等于
	>=	大于或等于
逻辑运算符	&&	逻辑“与”运算
		逻辑“或”运算
	!	逻辑“非”运算
复合运算符	++	自加
	--	自减
	+=	复合加
	-=	复合减

表 3.2.1 运算符表

3.3 表达式

通过运算符将运算对象连接起来的式子称为表达式，如 $1+2$ 、 $a-b$ 、 $3<4$ 等。表达式后面加上一个分号，就构成一条 C 语言表达式语句。

3.4 数组

数组是由一组具有相同数据类型的数据构成的集合。数组概念的引入，使得在处理多个相同类型的数据时程序更加清晰和简洁。

定义方式如下：

数据类型 数组名称[数组元素个数];

例如，定义一个有 5 个 int 型元素数组的语句为：

```
int a[5];
```

如果要访问一个数组中的某个元素，则需要使用以下语句。

数组名称[下标]

需要注意的是，数组下标是从 0 开始编号的。例如，将数组 a 中的第一个元素赋值为 1 的语句为：

```
a[0] = 1;
```

除了使用以上方法对数组赋值外，也可以在数组定义时对数组进行复制，如下语句：

```
int a[5] = {1, 2, 3, 4, 5};
```

或者使用以下语句实现，是同等效果。

```
int a[5];
```

```
a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4; a[4] = 5;
```

3.5 字符串

字符串的定义方式有两种，一种是以字符型数组方式定义，另一种是使用 String 类型定义。

以字符型数组方式定义的语句为：

```
char 字符串名称[字符个数];
```

使用字符型数组方式定义的字符串，其使用方法跟数组的使用方式一致，有多少个字符便占用多少字节的存储空间。

而在 Arduino 中，大多数情况下是使用 String 类型来定义字符串，该类型提供了一些操作字符串的成员函数，使得字符串使用起来更为灵活。定义语句是：

```
String 字符串名称;
```

具体使用，如下语句

```
String abc;
```

```
abc = "Arduino";
```

或者使用如下语句，实现同样的效果。

```
String abc = "Arduino";
```

相较于数组形式的定义方式，使用 String 类型定义字符串会占用更多的存储空间。

3.6 注释

“/*” 与 “*/” 之间的内容以及 “//” 之后的内容均为程序注释，使用他们可以更好地管理代码。注释不会被编译到程序中，因此不影响程序的运行。

为程序添加注释的方式有两种。

① 单行注释，语句为

```
// 注释内容
```

② 多行注释，语句为

```
/*
```

```
注释内容 1
```

```
注释内容 2
```

```
.....
```

```
*/
```

3.7 顺序结构

顺序结构是三种基本结构之一，也是最简单的一种流程结构，它采用自上而下的方式逐条执行各语句。简易顺序结构如下图所示：

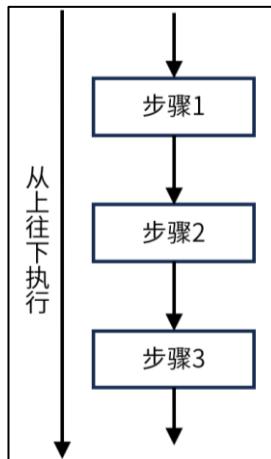


图 3.7.1 顺序结构图

框中的步骤 1、步骤 2 和步骤 3 都是按顺序执行的，先执行步骤 1，然后再执行步骤 2，最后执行步骤 3。

3.8 选择结构

选择结构，又称分支结构，可以控制程序的部分流程是否被执行，或者是从多条执行路径中选择一条来执行。在 C 语言中有两种选择语句：① if 语句，用来实现两个分支的选择结构
② switch 语句，用来实现多分支的选择结构。

选择结构 if，具体用法如下表所示：

if(表达式) {语句}	if(表达式) {语句 1} else {语句 2}
if(表达式) { if(表达式) {语句} } else {语句}	if(表达式) {语句} else if(表达式) {语句} else {语句}

表 3.8.1 选择结构 if 用法

if 语句执行流程图如下图所示。

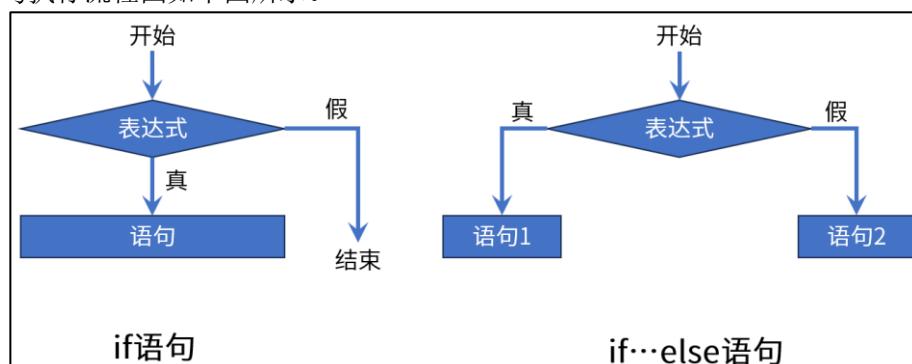


图 3.8.1 if 语句执行流程图

选择结构 switch，具体使用格式如下：

```

switch (整型表达式)
{
    case 整型常量表达式 1:
        [语句 1; break;]
    case 整型常量表达式 2:
        [语句 2; break;]
    case 整型常量表达式 3:
        [语句 3; break;]
    ...
    [default:

```

```
语句 n; break;
}
```

在程序运行时，首先会计算整型表达式的值，然后用该值与后面的所有 case 标签进行一一匹配（即查看该值是否与 case 标签的表达式的值相同）。如果有匹配的 case 标签，则从该 case 标签起，执行后续的语句，直至遇到 break 语句为止。如果没有匹配到 case 标签，则从 default 标签起，执行后续的语句，直到遇到 break 为止。

switch 语句程序流程图如下图所示：

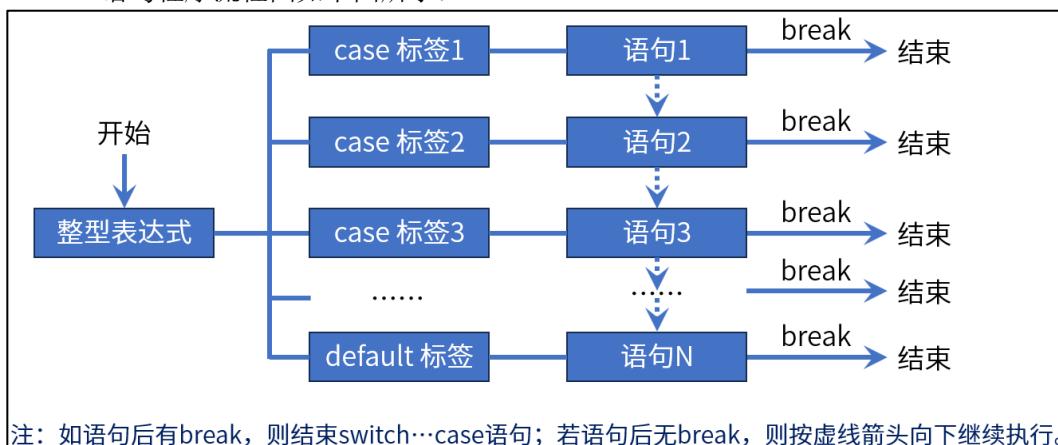


图 3.8.2 switch 语句执行流程图

这里特别需要注意：① case 后带的表达式一定要是常数，并且是整型。②在语句后面通常是有 break 关键字。③ 最后一般有 default，即前面 case 没有一个符合的情况，就会调到 default 里执行。虽然说语法上可以允许没有 default，但是为了完整，建议是需要的。

那么 if 和 switch 这两种选择结构有啥区别呢？

if 语句比较适用于对比条件比较复杂的且分支比较少的情况下使用；switch 语句则是适用于对比条件比较简单且分支比较多的情况下使用；正常情况下，先考虑能不能用 switch 语句，假如不满足使用需求，则使用 if 语句。当然，不管黑猫白猫捉到老鼠的就是好猫，能实现效果即可。

3.9 循环结构

循环结构就是重复执行一个语句块，直到不满足某个条件为止。在 C 语言中有三种循环语句：① while 语句 ② do while 语句 ③ for 语句

while 语句也称 while 循环，具体使用格式如下：

```
while (表达式)
```

语句

while 语句根据小括号内表达式的值来决定是否执行语句，当表达式的值为假时，循环结束，语句不会被执行；假如表达式的值为真，语句被执行，然后会再次判断表达式的值，如此反复，直至表达式的值为假。

do while 语句也称 do while 循环，具体使用格式如下：

```
do
```

语句

```
while (表达式)
```

从结构上，由“do”关键字开头，紧接着是语句也就是循环体了，最后是 while 关键字和表达式。do while 语句是先执行循环体，然后再检查条件是否成立，若成立，再执行循环体，这就是跟 while 语句不同。

for 语句，相对前面两个循环语句复杂一点，具体使用格式如下：

```
for (表达式 1; 表达式 2; 表达式 3)
```

语句

相比前面的 while 和 do while 语句，for 语句的表达式有三个，表达式之间用‘；’隔开，这三个表达式的作用和执行时期是不一样的，具体情况如下表：

表达式	作用	执行时期
表达式 1	通常在这个地方对循环变量进行初始化或赋值	开始时执行 1 次
表达式 2	表达式的值为真时执行循环体，为假时结束 for 语句	循环体执行前
表达式 3	通常在这个地方对循环变量进行修改，通常是加减操作	循环体执行后

表 3.9.1 for 语句表达式情况

具体执行流程：在 for 语句被执行前，首先会执行表达式 1，然后再检查表达式 2 的值：假如为假，不执行循环体，结束 for 语句；假如为真，则执行循环体语句。执行完循环语句后，再去执行表达式 3，并再次去检查表达式 2 的值，如此反复，直到表达式 2 的值为假，结束 for 语句。

这三个循环语句，执行流程图如下所示：

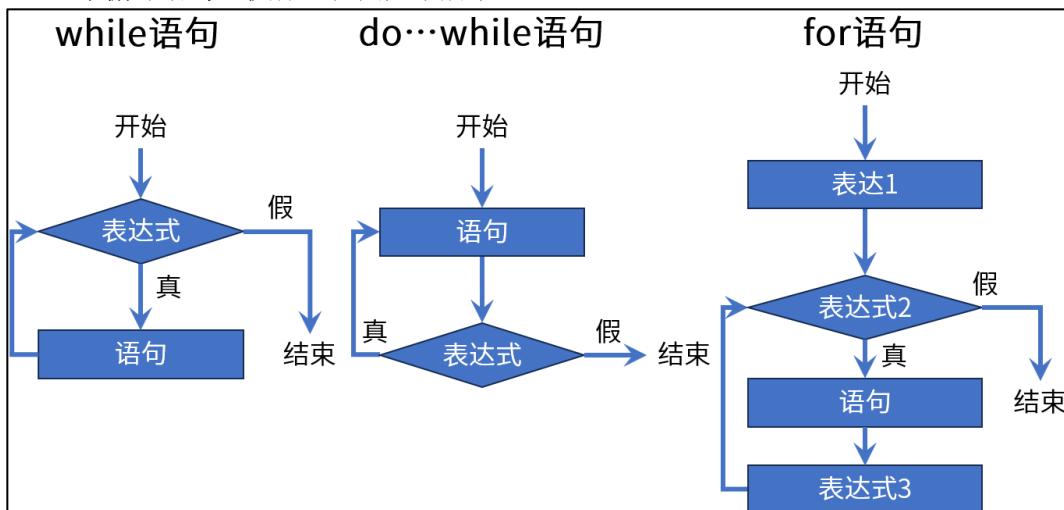


图 3.9.1 三种循环语句执行流程图

在实际编写程序中，哪种情况用哪个好呢？

如果程序中固定了循环的次数，那么就可以采用 for 语句。如果程序中必须执行一次程序，那么就可以使用 do while 语句。除了以上两种情况，就可以使用 while 语句。

第四章 ESP32-S3 基础知识

欲先善其事，必先利其器，我们不仅要具备软件的实力，更要具备硬件的实力，前面作者已经介绍了 Arduino 基础知识，它是能让 MCU 根据开发者的意愿来执行相关的操作，而本章节主要讲解 ESP32-S3 这一块 MCU 的硬件知识，它是执行开发者意愿的工具，那么了解它更能让我们掌握它的使用。

本章分为以下几个小节：

- 4.1 为什么选择 ESP32-S3
- 4.2 初识 ESP32-S3
- 4.3 ESP32-S3 功能描述
- 4.4 S3 系列型号对比
- 4.5 ESP32-S3 功能概述
- 4.6 ESP32-S3 启动流程

4.1 为什么选择 ESP32-S3

在研发之初，作者也对比过乐鑫官方推出的几款 MCU 系列，经过它们各自的功能及应用场景来分析，最终作者选择 S 系列的 S3 型号。

下面，作者比较一下乐鑫推出的芯片有哪些特点：

硬件比较	S系列	C系列	H系列	ESP32系列
内核数量	单核(S2) / 双核(S3)	单核	单核	单/双核
时钟频率	240MHz	120MHz(C2(ESP8685)) 160MHz(C3和C6)	96MHz	240MHz
引出编程IO	43(S2) / 45(S3)	14(C2) 22或16或15(C3) 30或22(C6)	19	34
神经网络加速	S2无 / S3有	无	无	无
通信协议	2.4GHz Wi-Fi(S2和S3有)、 BLE(S3有)	2.4GHz Wi-Fi(C2和C3有)、 BLE(C2、C3和C6有)、 2.4GHz Wi-Fi6(C6有)、 Zigbee和Thread(C6有)	BLE、Zigbee、 Thread	2.4GHz Wi-Fi、BT、 BLE
SRAM(KB)	320(S2) / 512(S3)	272(C2) / 400(C3) / 512(C6)	320	520
ROM(KB)	128(S2) / 384(S3)	576(C2) / 384(C3) / 320(C6)	128	448

表 4.1.1 乐鑫各系列 MCU 硬件区别

在上述表格中，我们可以看到乐鑫推出的各系列 MCU 在硬件方面存在一些差异。下面我将继续分析这些差异及其对应用场景的影响。

1，在内核数量方面：S 系列和 ESP32 系列支持单核和双核处理器，而 C 系列和 H 系列仅支持单核处理器。这意味着 S 系列和 ESP32 系列在处理多任务和高强度计算方面具有更强的性能。对于需要高效能、多任务处理的应用场景，如复杂算法处理、大数据分析等，S 系列和 ESP32 系列可能更合适。

2，在时钟频率方面，S 系列和 ESP32 系列的时钟频率范围为 80~240MHz，而 C 系列和 H 系列的时钟频率分别为 120MHz 和 96MHz。较高的时钟频率意味着更快的处理速度和更高的性能。对于需要高速处理的应用场景，如实时信号处理、高速数据采集等，S 系列和 ESP32 系列可能更合适。

3，在引出编程 IO 方面，S 系列和 ESP32 系列的引出编程 IO 数量较多，而 C 系列和 H 系列的引出编程 IO 数量较少。这表明 S 系列和 ESP32 系列在编程接口的多样性和灵活性方面具有优势。对于需要连接多种外设和传感器的应用场景，S 系列和 ESP32 系列可能更合适。

4，在神经网络加速方面，只有 S 系列支持神经网络加速功能。这意味着选择 S 系列可以更好地满足深度学习、图像识别等应用场景的需求。对于需要加速神经网络运算的应用场景，如智能家居控制、智能安防等，S 系列可能更合适。

5，在通信协议方面，所有系列都支持 2.4G Wi-Fi 和蓝牙（BLE），这意味着它们在无线通信方面具有良好的兼容性。

6，在存储器方面，各系列 MCU 的 SRAM 和 ROM 大小有所不同。较大的存储器可以提供更多的程序运行空间和数据存储空间，以满足更复杂的应用需求。对于需要处理大量数据和运行复杂程序的应用场景，如物联网网关、智能仪表等，S 系列和 ESP32 系列可能更合适。

综上所述，乐鑫推出的各系列 MCU 在硬件方面各有特点，选择哪个系列取决于具体的应用场景和需求。对于需要高性能、多核处理和神经网络加速的应用场景，S 系列可能是更好的选择；而对于简单的物联网应用场景，C 系列或 H 系列可能更合适。

正点原子选择 S 系列的 S3 型号作为开发板的核心芯片，是为了读者提供更好的学习资源和开发体验，帮助读者更好地掌握物联网和嵌入式开发的相关技术。

另外，乐鑫科技还提供了一个在线选型工具（<https://products.espressif.com/#/product-selector?language=zh>），名为 ESP Product Selector。它可以帮助用户全面了解乐鑫产品与方案、提高产品选型和开发效率，如下图所示。



图 4.1.1 乐鑫在线选型工具

上图①显示了筛选工具的选择，左边是产品选型，右边是产品对比。上图②表示产品选型的功能筛选，主要根据客户的需求来选择，例如工作温度、单/双核、是否具备天线等条件，来选择自己心仪的芯片/模组或者开发板。上图③表示功能筛选之后的结果选择，例如芯片/模组或者满足条件的开发板。最后，上图④表示筛选的结果，如果筛选结果是芯片/模组，那么它就会显示符合筛选的芯片型号或者模组。

4.2 初识 ESP32-S3

ESP32-S3 是一款由乐鑫公司开发的物联网芯片，它具有一些非常独特的功能和特点。以下是对 ESP32-S3 的初步介绍：

1，架构和性能：ESP32-S3 采用 Xtensa® LX7 CPU，这是一个哈佛结构的双核系统。它具有独立的指令总线和数据总线，所有的内部存储器、外部存储器以及外设都分布在这两条总线

上。这种架构使得 CPU 可以同时读取指令和数据，从而提高了处理速度。

2, 存储: ESP32-S3 具有丰富的存储空间。它内部有 384 KB 的内部 ROM, 512 KB 的内部 SRAM, 以及 8 KB 的 RTC 快速存储器和 8 KB 的 RTC 慢速存储器。此外, 它还支持最大 1 GB 的片外 FLASH 和最大 1 GB 的片外 RAM。

3, 外设: ESP32-S3 具有许多外设, 总计有 45 个模块/外设。其中 11 个具有 GDMA (Generic DMA) 功能, 可以用来进行数据块的传输, 减轻 CPU 的负担, 提高整体性能。

4, 通信: ESP32-S3 同时支持 WIFI 和蓝牙功能, 应用领域贯穿移动设备、可穿戴电子设备、智能家居等。在 2.4GHz 频带支持 20MHz 和 40MHz 频宽。

5, 向量指令: ESP32-S3 增加了用于加速神经网络计算和信号处理等工作的向量指令。这些向量指令可以大大提高芯片在 AI 方面的计算速度和效率。

ESP32-S3 是一款功能强大、性能丰富的物联网芯片, 适用于各种物联网应用场景。以上信息仅供参考, 如需了解更多信息, 请访问乐鑫公司官网查询相关资料。

4.3 ESP32-S3 资源简介

下面来看看 ESP32-S3 具体的内部资源, 如下表所示。

ESP32-S3 资源					
内核	Xtensa® LX7 CPU	系统定时器	1	UART	3
主频	240MHz	定时器组	2	RNG	1
ROM	384KB	LEDC	1	I2C	2
SRAM	512KB	RMT	1	I2S	2
编程 IO	45GPIO	PCNT	1	SPI	4 (0、1 禁用)
工作电压	3.3	TWAI	1	RGB	1
Wi-Fi/BLE	1/1	USB OTG	1	SD/MMC	1

表 4.3.1 ESP32-S3 内部资源表

由表可知, ESP32 内部资源还是非常丰富的, 本书将针对这些资源进行详细的使用介绍, 并提供丰富的例程, 供大家参考学习, 相信经过本书的学习, 您会对 ESP32-S3 系列芯片有一个全面的了解和掌握。

关于 ESP32-S3 内部资源的详细介绍, 请大家参考“光盘→A 盘→7, 硬件资料→2, 芯片资料→esp32-s3_technical_reference_manual_cn.pdf”, 该文档即《ESP32-S3 的技术手册》, 里面有 ESP32-S3 详细的资源说明和相关性能参数。

4.4 S3 系列型号对比

乐鑫 S3 系列型号包括 ESP32-S3、ESP32-S3R2、ESP32-S3R8 和 ESP32-S3FN8 等。这些型号在硬件配置、功能和应用场景方面略有不同。不同型号的 MCU 都有不同的应用场景, 下面我们来看一下这些型号的命名规则, 如下图所示。

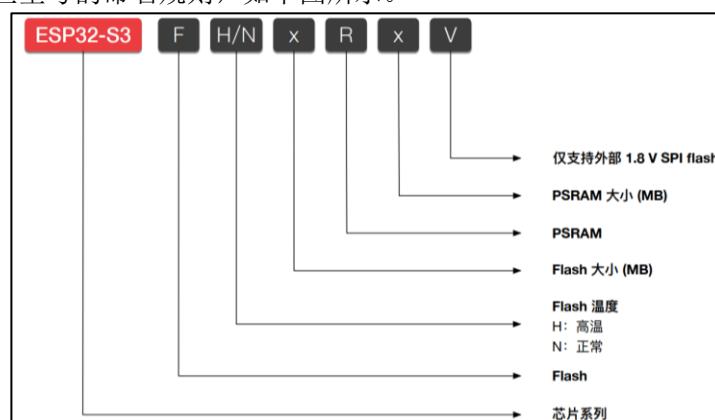


图 4.4.1 ESP32-S3 系列芯片命名规则

从上图可以看到, F 表示内置 FLASH; H/N 表示 FLASH 温度(H: 高温, N: 常温); x 表示内置 FLASH 大小 (MB); R 表示内置 PSRAM; x 表示内置 PSRAM 大小 (MB); V 表示仅

支持外部 1.8v spi flash。为了让读者更清晰了解 ESP32-S3 命名规则,这里作者以 ESP32-S3FH4R2 这一款芯片为例,绘画一副清晰的命名示意图,如下图所示。



图 4.4.2 ESP32-S3FH4R2 命名解析

根据上述两张图的分析,我们可以了解到乐鑫 S3 系列的命名规则和特点。除了 S3 系列的芯片之外,乐鑫还推出了 S3 系列的模组,它是 S3 系列芯片的简易系统。

乐鑫 S3 系列模组是基于 S3 系列芯片的子系统,它已经设计好了外围电路,简化了开发过程,让开发者可以更快速地使用 S3 系列芯片进行开发。通过使用 S3 系列模组,开发者可以更容易地实现特定功能,缩短开发周期,提高开发效率。

乐鑫推出了 ESP32-S3-WROOM-1 和 ESP32-S3-WROOM-1U 两款通用型 Wi-Fi+低功耗蓝牙 MCU 模组,如下图所示,它们搭载 ESP32-S3 系列芯片。除具有丰富的外设接口外,模组还拥有强大的神经网络运算能力和信号处理能力,适用于 AIoT 领域的多种应用场景,例如唤醒词检测和语音命令识别、人脸检测和识别、智能家居、智能家电、智能控制面板、智能扬声器等。

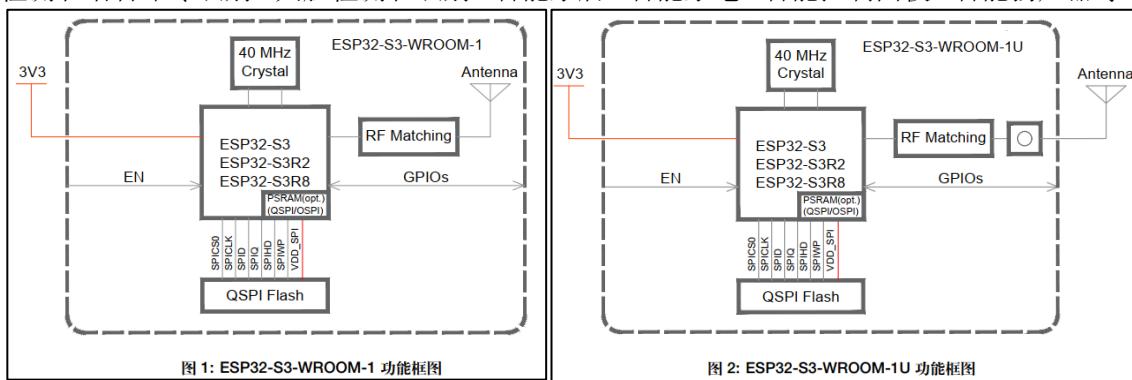


图 4.4.3 ESP32-S3-WROOM-1 和 ESP32-S3-WROOM-1U 的功能框图

从上图可知,ESP32-S3-WROOM-1 采用 PCB 板载天线,而 ESP32-S3-WROOM-1U 采用连接器连接外部天线。两款模组均有多种芯片型号可供选择,具体见下表所示:

模组型号	内置芯片	外置 FLASH	内置 PSRAM
ESP32-S3-WROOM-1-N4	ESP32-S3	4	0
ESP32-S3-WROOM-1-N8	ESP32-S3	8	0
ESP32-S3-WROOM-1-N16	ESP32-S3	16	0
ESP32-S3-WROOM-1-H4	ESP32-S3	4	0
ESP32-S3-WROOM-1-N4R2	ESP32-S3R2	4	2 (Quad SPI)
ESP32-S3-WROOM-1-N8R2	ESP32-S3R2	8	2 (Quad SPI)
ESP32-S3-WROOM-1-N16R2	ESP32-S3R2	16	2 (Quad SPI)
ESP32-S3-WROOM-1-N4R8	ESP32-S3R8	4	8 (Octal SPI)
ESP32-S3-WROOM-1-N8R8	ESP32-S3R8	8	8 (Octal SPI)
ESP32-S3-WROOM-1-N16R8	ESP32-S3R8	16	8 (Octal SPI)
ESP32-S3-WROOM-1U-N4	ESP32-S3	4	0
ESP32-S3-WROOM-1U-N8	ESP32-S3	8	0
ESP32-S3-WROOM-1U-N16	ESP32-S3	16	0
ESP32-S3-WROOM-1U-H4	ESP32-S3	4	0
ESP32-S3-WROOM-1U-N4R2	ESP32-S3R2	4	2 (Quad SPI)
ESP32-S3-WROOM-1U-N8R2	ESP32-S3R2	8	2 (Quad SPI)
ESP32-S3-WROOM-1U-N16R2	ESP32-S3R2	16	2 (Quad SPI)
ESP32-S3-WROOM-1U-N4R8	ESP32-S3R8	4	8 (Octal SPI)
ESP32-S3-WROOM-1U-N8R8	ESP32-S3R8	8	8 (Octal SPI)
ESP32-S3-WROOM-1U-N16R8	ESP32-S3R8	16	8 (Octal SPI)

表 4.4.1 通用型模组的命名

根据上表，可以看出这两款模组的主控芯片是 ESP32-S3 和 ESP32-S3Rx，它们都属于乐鑫的 ESP32-S3 系列芯片。之前作者已经详细讲解了 ESP32-S3 系列芯片的命令规则，可以得出这两款通用模组都是外接 Flash 存储器，并且内置有 PSRAM（主控芯片 ESP32-S3 没有内置 PSRAM）。下面我们以 ESP32-S3-WROOM-1-N16R8 模组为例，来讲解模组的命名规则，如下图所示。

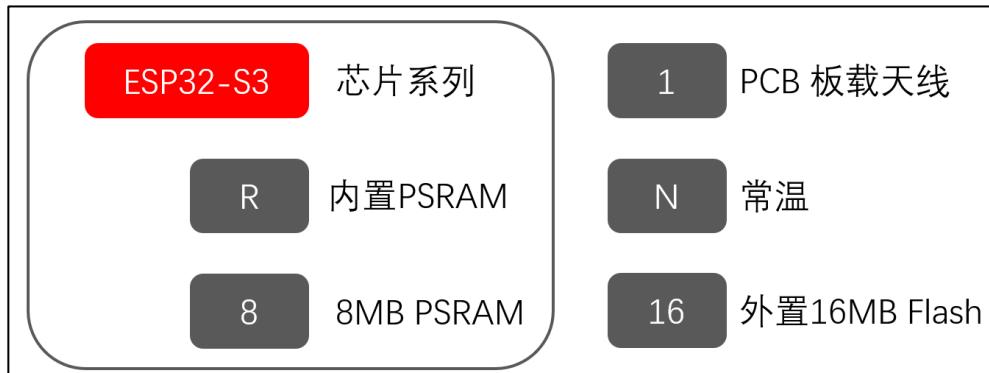


图 4.4.3 模组的命令规则

通过了解模组内置的主控芯片类型，开发者可以更好地理解该模组的功能和特点，并根据需要进行相应的开发和应用。正点原子 ESP32-S3 开发板是以 ESP32-S3-WROOM-1-N16R8 模组作为主控，它可以提供稳定的控制系统和高效的数据处理能力，同时引出的 IO 可以满足各种应用需求。

4.5 ESP32-S3 功能概述

4.5.1 系统和存储器

ESP32-S3 采用哈佛结构 Xtensa® LX7 CPU 构成双核系统。所有的内部存储器、外部存储器以及外设都分布在 CPU 的总线上。

以下是 ESP32-S3 的主要特性：

1, 地址空间：ESP32-S3 拥有丰富的地址空间，包括内部存储器指令地址空间、内部存储器数据地址空间、外设地址空间、外部存储器指令虚地址空间、外部存储器数据虚地址空间、内部 DMA 地址空间和外部 DMA 地址空间。这些地址空间为芯片的各个部分提供了独立的存储空间。

2, 内部存储器：ESP32-S3 内部存储器包括 384 KB 的内部 ROM、512 KB 的内部 SRAM、8 KB 的 RTC 快速存储器和 8 KB 的 RTC 慢速存储器。这些存储器为芯片提供了存储和读取数据的能力。

3, 外部存储器：ESP32-S3 支持最大 1 GB 的片外 FLASH 和最大 1 GB 的片外 RAM。这些外部存储器可以用来存储大量的程序代码和数据，以满足复杂应用的需求。

4, 外设空间：ESP32-S3 总计有 45 个模块/外设，这些外设为芯片提供了丰富的输入输出接口和特殊功能。

5, GDMA (Generic DMA)：ESP32-S3 具有 11 个具有 GDMA 功能的模块/外设，这些 GDMA 外设可以用来进行数据块的传输，从而减轻 CPU 的负担，提高整体性能。

4.5.2 IO MUX 和 GPIO 交换矩阵

ESP32-S3 芯片有 45 个物理通用输入输出管脚 (GPIO Pin)。每个管脚都可用作一个通用输入输出，或连接一个内部外设信号。利用 GPIO 交换矩阵、IO MUX (IO 复用选择器) 和 RTC IO MUX (RTC 复用选择器)，可配置外设模块的输入信号来源于任何的 GPIO 管脚，并且外设模块的输出信号也可连接到任意 GPIO 管脚。这些模块共同组成了芯片的输入输出控制。值得注意的是，这 45 个物理 GPIO 管脚的编号为 0~21、26~48。这些管脚即可作为输入也可作为输出管脚。正如前文所述，正点原子选择 ESP32-S3-WROOM-1-N16R8 模组作为主控，但由于该模组只有 36 个实际引脚的物理 GPIO 管脚。这是因为该模组的 Flash 和 PSRAM 使用了八线 SPI

即 Octal SPI 模式，这些模式共占用了 12 个 GPIO 管脚。而且，该模组还将 IO35、IO36、IO37 引出，所以最终的管脚数量为 45-12+3，即 36 个 GPIO 管脚。

下图是从《esp32-s3_datasheet_cn.pdf》数据手册截取下来的，主要描述 FLASH 和 PSRAM 使用八线 SPI 模式下的管脚。

管脚序号	管脚名称	单线 SPI		双线 SPI		四线 SPI		八线 SPI	
		Flash	PSRAM	Flash	PSRAM	Flash	PSRAM	Flash	PSRAM
33	SPICLK	CLK	CLK	CLK	CLK	CLK	CLK	CLK	CLK
32	SPICS0 ¹	CS#		CS#		CS#		CS#	
28	SPICS1 ²		CE#		CE#		CE#		CE#
35	SPIID	DI	SI/SIO0	DI	SI/SIO0	DI	SI/SIO0	DQ0	DQ0
34	SPIQ	DO	SO/SIO1	DO	SO/SIO1	DO	SO/SIO1	DQ1	DQ1
31	SPIWP	WP#	SIO2	WP#	SIO2	WP#	SIO2	DQ2	DQ2
30	SPIHD	HOLD#	SIO3	HOLD#	SIO3	HOLD#	SIO3	DQ3	DQ3
38	GPIO33							DQ4	DQ4
39	GPIO34							DQ5	DQ5
40	GPIO35							DQ6	DQ6
41	GPIO36							DQ7	DQ7
42	GPIO37							DQS/DM	DQS/DM

图 4.5.2.1 芯片与封装内 flash/PSRAM 的管脚对应关系

需要注意的是，正点原子 ESP32-S3 开发板的原理图并没有使用 IO35-IO37 号管脚，所以不存在共用 FLASH 和 PSRAM 管脚。

下面我们来看一下这个模组的实物图和引脚分布图，如下图所示。

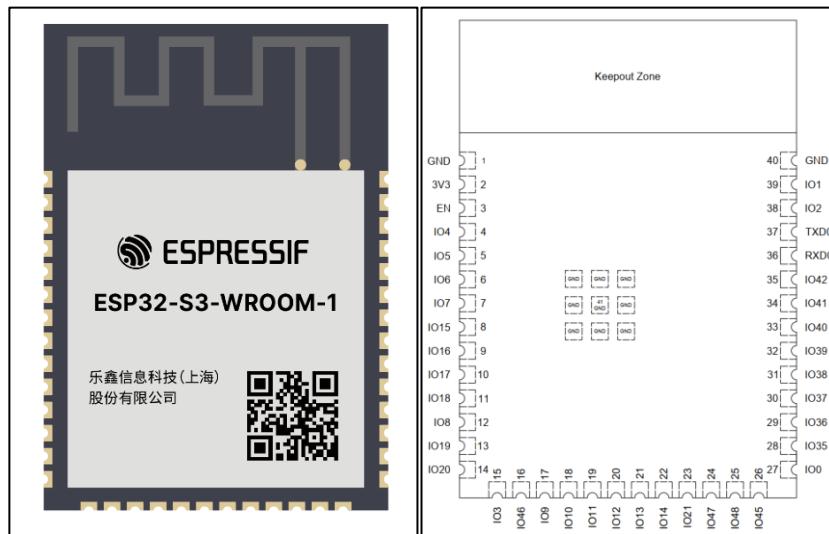


图 4.5.2.2 ESP32-S3-WROOM-1 实物图和引脚分布图

从上图可以得知，左边的图片是该模组的 3D 实物图，而右边的图片是该模组的管脚分布图。虽然这些管脚是无序的，但它们都可以被复用为其他功能（除个别功能外），例如 SPI、串口、IIC 等协议。这是 ESP32 相比其他 MCU 的优势之一，它具有更多的可复用管脚，可以支持更多的外设和协议。

接下来，我们来看一下模组管脚默认复用管脚和管脚功能释义，如下表所示：

管脚名称	序号	类型	描述
GND	1	P	接地
3V3	2	P	供电
EN	3	I	高电平：使能芯片 低电平：关闭芯片 注：不能悬空

IO4	4	I/O/T	RTC GPIO4\GPIO4\TOUCH4\ADC1 CH3
IO5	5	I/O/T	RTC GPIO5\GPIO5\TOUCH5\ADC1 CH4
IO6	6	I/O/T	RTC GPIO6\GPIO6\TOUCH6\ADC1 CH5
IO7	7	I/O/T	RTC GPIO7\GPIO7\TOUCH7\ADC1 CH6
IO15	8	I/O/T	RTC GPIO15\GPIO15\U0RTS\ADC2 CH4\XTAL 32K P
IO16	9	I/O/T	RTC GPIO16\GPIO16\U0CTS\ADC2 CH5\XTAL 32K N
IO17	10	I/O/T	RTC GPIO17\GPIO17\U1TXD\ADC2 CH6
IO18	11	I/O/T	RTC GPIO18\GPIO18\U1RXD\ADC2 CH7\CLK_OUT3
IO8	12	I/O/T	RTC GPIO8\GPIO8\TOUCH8\ADC1 CH7\SUBSPICS1
IO19	13	I/O/T	RTC GPIO19\GPIO19\U1RTS\ADC2 CH8\CLK_OUT2\USB D-
IO20	14	I/O/T	RTC GPIO20\GPIO20\U1CTS\ADC2 CH9\CLK_OUT1\USB D+
IO3	15	I/O/T	RTC GPIO3\GPIO3\TOUCH3\ADC1 CH2
IO46	16	I/O/T	GPIO46
IO9	17	I/O/T	RTC GPIO9,GPIO9,TOUCH9,ADC1 CH8,FSPIHD,SUBSPIHD
IO10	18	I/O/T	RTC GPIO10,GPIO10,TOUCH10,ADC1 CH9,FSPICS0,FSPIIO4
IO11	19	I/O/T	RTC GPIO11,GPIO11,TOUCH11,ADC2 CH0,FSPIID,FSPIIO5
IO12	20	I/O/T	RTC GPIO12,GPIO12,TOUCH12,ADC2 CH1,FSPICLK,FSPIIO6
IO13	21	I/O/T	RTC GPIO13,GPIO13,TOUCH13,ADC2 CH2,FSPIQ,FSPIIO7
IO14	22	I/O/T	RTC GPIO14,GPIO14,TOUCH14,ADC2 CH3,FSPIWP,FSPIDQS
IO21	23	I/O/T	RTC GPIO21,GPIO21
IO47	24	I/O/T	SPICLK_P DIFF,GPIO47,SUBSPICLK_P DIFF
IO48	25	I/O/T	SPICLK_N DIFF\GPIO48\SUBSPICLK_N DIFF
IO45	26	I/O/T	GPIO45
IO0	27	I/O/T	RTC GPIO0\GPIO0
IO35	28	I/O/T	SPIIO6\GPIO35\FSPID\SUBSPID
IO36	29	I/O/T	SPIIO7\GPIO36\FSPICLK\SUBSPICLK
IO37	30	I/O/T	SPIDQS\GPIO37\FSPIQ\SUBSPIQ
IO38	31	I/O/T	GPIO38\FSPIWP\SUBSPIWP
IO39	32	I/O/T	MTCK\GPIO39\CLK_OUT3\SUBSPICS1
IO40	33	I/O/T	MTDO\GPIO40\CLK_OUT2
IO41	34	I/O/T	MTDI\GPIO41\CLK_OUT1
IO42	35	I/O/T	MTMS\GPIO42
RXD0	36	I/O/T	U0RXD\GPIO44\CLK_OUT2
TXD0	37	I/O/T	U0TXD\GPIO43\CLK_OUT1
IO2	38	I/O/T	RTC GPIO2\GPIO2\TOUCH2\ADC1 CH1
IO1	39	I/O/T	RTC GPIO1\GPIO1\TOUCH1\ADC1 CH0
GND	40	P	接地
EPAD	41	P	接地

表 4.5.2.1 管脚定义

4.5.3 复位与时钟

4.5.3.1 ESP32-S3 复位等级

ESP32-S3 提供四种级别的复位方式，分别是 CPU 复位、内核复位、系统复位和芯片复位。除芯片复位外其它复位方式不影响片上内存存储的数据。下图展示了整个芯片系统的结构以及四种复位等级。

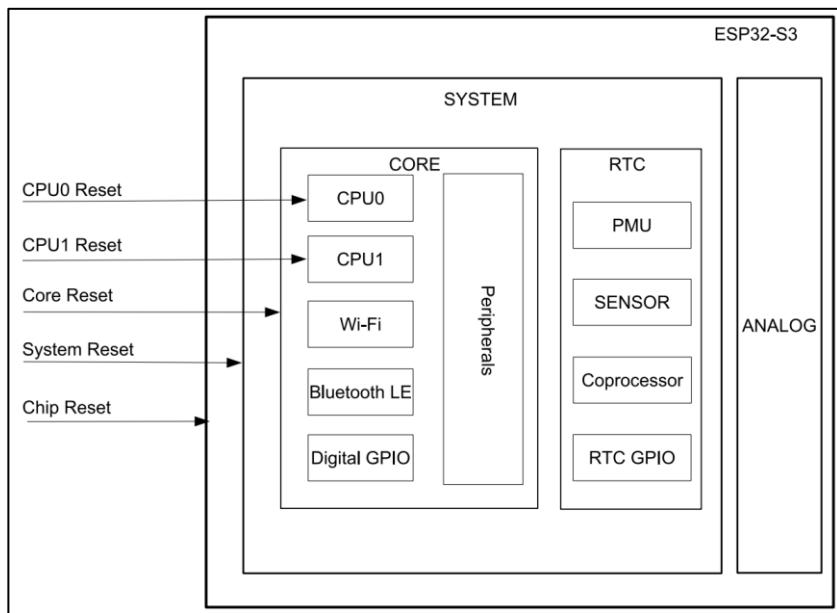


图 4.5.3.1.1 四种复位等级

CPU 复位：只复位 CPU_x 内核，这里的 CPU_x 代表 CPU0 和 CPU1。复位释放后，程序将从 CPU_x Reset Vector 开始执行。

内核复位：复位除了 RTC 以外的数字系统，包括 CPU0、CPU1、外设、WiFi、Bluetooth® LE 及数字 GPIO。

系统复位：复位包括 RTC 在内的整个数字系统。

芯片复位：复位整个芯片。

上述任意复位源产生时，CPU0 和 CPU1 均将立刻复位。复位释放后，CPU0 和 CPU1 可分别通过读取寄存器 RTC_CNTL_RESET_CAUSE_PROCPU 和 RTC_CNTL_RESET_CAUSE_APP CPU 获取复位源。这两个寄存器记录的复位源除了复位级别为 CPU 复位的复位源分别对应自身的 CPU_x 以外，其余的复位源保持一致。下表列出了从上述两个寄存器中可能读出的复位源。

复位编码	复位源	复位等级	描述
0x01	芯片复位	芯片复位	-
0x0F	欠压系统复位	系统复位或芯片复位	欠压检测器触发的系统复位
0x10	RWDT 系统复位	系统复位	见技术手册章节 13
0x12	Super Watchdog 复位	系统复位	见技术手册章节 13
0x13	GLITCH 复位	系统复位	见技术手册章节 24
0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器触发
0x05	Deep-sleep 复位	内核复位	见技术手册章节 10
0x07	MWDT0 内核复位	内核复位	见技术手册章节 13
0x08	MWDT1 内核复位	内核复位	见技术手册章节 13
0x09	RWDT 内核复位	内核复位	见技术手册章节 13
0x14	eFuse 复位	内核复位	eFuse CRC 校验错误触发复位
0x15	USB (UART) 复位	内核复位	见技术手册章节 33
0x16	USB (JTAG) 复位	内核复位	见技术手册章节 33
0x0B	MWDT0 CPU _x 复位	CPU 复位	见技术手册章节 13
0x0C	软件 CPU _x 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPU _x 复位	CPU 复位	见技术手册章节 13
0x11	MWDT1 CPU _x 复位	CPU 复位	见技术手册章节 13

表 4.5.3.1.1 相关复位的复位源

上表描述了不同的复位对应的复位源，在 ESP32-S3 上电复位时，它的复位源为芯片复位，如下信息所示：

```
ESP-ROM:esp32s3-20210327
Build:Mar 27 2021
rst:0x1 (POWERON),boot:0xb (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fce3810,len:0x17c0
load:0x403c9700,len:0xd7c
load:0x403cc700,len:0x300c
entry 0x403c992c
```

从上述内容可以看到，rst 为 0x01（复位编码），根据上表的对应关系，可得芯片上电时的复位源为芯片复位。

4.5.3.2 系统时钟

ESP32-S3 的时钟主要来源于振荡器（oscillator, OSC）、RC 振荡电路和 PLL 时钟生成电路。上述时钟源产生的时钟经时钟分频器或时钟选择器等时钟模块的处理，使得大部分功能模块可以根据不同功耗和性能需求来获取及选择对应频率的工作时钟。下图为 ESP32-S3 系统时钟结构。

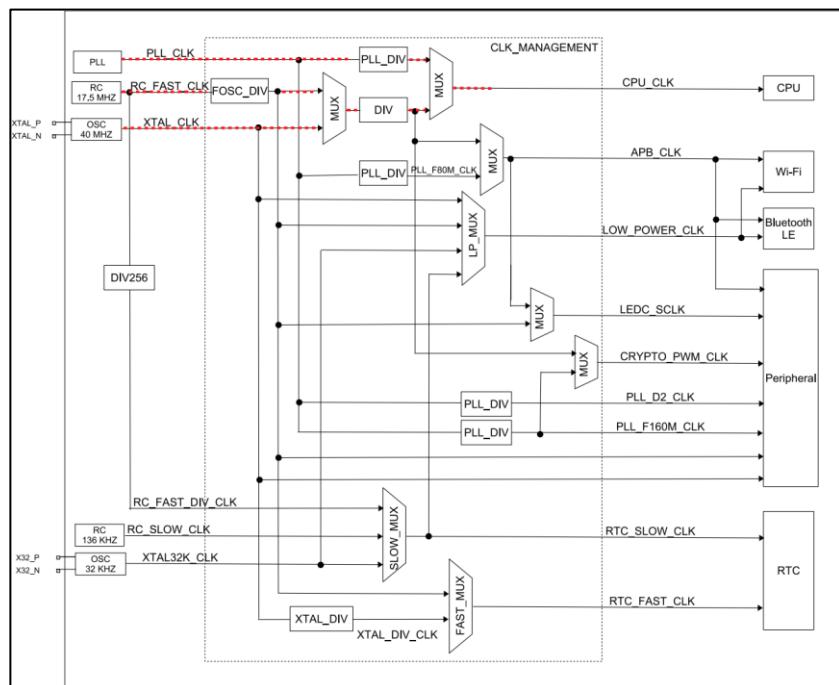


图 4.5.3.2.1 ESP32-S3 时钟树

从上图可知，ESP32-S3 时钟频率，可划分为：

(1)，高性能时钟，主要为 CPU 和数字外设提供工作时钟。

- ①：PLL_CLK：320MHz 或者 480MHz 内部 PLL 时钟
- ②：XTAL_CLK：40MHz 外部晶振时钟

(2)，低功耗时钟，主要为 RTC 模块以及部分处于低功耗模式的外设提供工作时钟。

- ①：XTAL32K_CLK：32kHz 外部晶振时钟
- ②：RC_FAST_CLK：内置快速 RC 振荡器时钟，频率可调节（通常为 17.5MHz）
- ③：RC_FAST_DIV_CLK：内置快速 RC 振荡器分频时钟（RC_FAST_CLK/256）

①：RC_SLOW_CLK：内置慢速 RC 振荡器，频率可调节（通常为 136 kHz）

从上图红色线条所示，CPU_CLK 代表 CPU 的主时钟。在 CPU 最高效的工作模式下，主频可以达到 240MHz。主频频率是由寄存器 SYSTEM_SOC_CLK_SEL(SEL_0: 选择 SOC 时钟源)、SYSTEM_PLL_FREQ_SEL(SEL_2: 选择 PLL 时钟频率)和 SYSTEM_CPUPERIOD_SEL(SEL_3: 选择 CPU 时钟频率)共同确定的，具体如下表所示：

时钟源	SEL_0	SEL_2	SEL_3	CPU 时钟频率
XTAL_CLK	0	-	-	CPU_CLK = XTAL_CLK/(SYSTEM_PRE_DIV_CNT + 1)
PLL_CLK (480 MHz)	1	1	0	CPU_CLK = PLL_CLK/6 CPU_CLK 频率为 80 MHz。
PLL_CLK (480 MHz)	1	1	1	CPU_CLK = PLL_CLK/3 CPU_CLK 频率为 160 MHz
PLL_CLK (480 MHz)	1	1	2	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 240 MHz
PLL_CLK (320 MHz)	1	0	0	CPU_CLK = PLL_CLK/4 CPU_CLK 频率为 80 MHz
PLL_CLK (320 MHz)	1	0	1	CPU_CLK = PLL_CLK/2 CPU_CLK 频率为 160 MHz
RC_FAST_CLK	2	-	-	CPU_CLK = RC_FAST_CLK/(SYSTEM_PRE_DIV_CNT + 1)

表 4.5.3.2.1 CPU_CLK 时钟频率配置

从上表可以得知, 如果用户想要将 ESP32-S3 的主频设置为 240MHz, 那么我们应该选择 PLL_CLK 作为输入源, 然后通过二分频得到 240MHz 的时钟频率。

外设、WiFi、BLE、RTC 等时钟配置及选择源, 请读者参考《esp32-s3_technical_reference_manual_cn.pdf》技术手册→第 7 章节复位和时钟。

4.5.4 芯片 Boot 控制

在上电复位、RTC 看门狗复位、欠压复位、模拟超级看门狗 (analog super watchdog) 复位、晶振时钟毛刺检测复位过程中, 硬件将采样 Strapping 管脚电平存储到锁存器中, 并一直保持到芯片掉电或关闭。GPIO0、GPIO3、GPIO45 和 GPIO46 锁存的状态可以通过软件从寄存器 GPIO_STRAPPING 中读取。GPIO0、GPIO45 和 GPIO46 默认连接内部上拉/下拉。如果这些管脚没有外部连接或者连接的外部线路处于高阻抗状态, 内部弱上拉/下拉将决定这几个管脚输入电平的默认值, 如下表所示。

功能	Strapping 管脚	默认配置
芯片启动模式	GPIO0 和 GPIO46	上拉
VDD_SPI 电压	GPIO45	下拉
ROM 代码日志打印	GPIO46	下拉
JTAG 信号源	GPIO3	浮空

表 4.5.4.1 Strapping 管脚默认配置

GPIO0、GPIO45 和 GPIO46 在芯片复位时连接芯片内部的弱上拉/下拉电阻。如果 strapping 管脚没有外部连接或者连接的外部线路处于高阻抗状态, 这些电阻将决定 strapping 管脚的默认值。所有 strapping 管脚都有锁存器。系统复位时, 锁存器采样并存储相应 strapping 管脚的值, 一直保持到芯片掉电或关闭。锁存器的状态无法用其他方式更改。因此, strapping 管脚的值在芯片工作时一直可读取, 并可在芯片复位后作为普通 IO 管脚使用。

① 芯片启动模式控制

复位释放后, GPIO0 和 GPIO46 共同决定启动模式。详见下表。

启动模式	GPIO0	GPIO46
默认配值	1	0
SPI BOOT	1	任意值
Download Boot	0	0
无效组合	0	1

表 4.5.4.2 芯片启动模式控制

正常情况下, ESP32 启动模式为“SPI BOOT”, 当我们按下开发板的 BOOT 按键时才能进入“Download Boot”模式启动。

② VDD_SPI 电压控制

ESP32-S3 系列芯片所需的 VDD_SPI 电压请参考《esp32-s3_datasheet_cn.pdf》数据手册的 1.2 型号对比表格，如下图所示：

表 1-1. ESP32-S3 系列芯片对比				
订购代码 ¹	封装内 Flash	封装内 PSRAM	环境温度 ² (°C)	VDD_SPI 电压 ³
ESP32-S3	—	—	-40 ~ 105	3.3 V/1.8 V
ESP32-S3FN8	8 MB (Quad SPI) ⁴	—	-40 ~ 85	3.3 V
ESP32-S3R2	—	2 MB (Quad SPI)	-40 ~ 85	3.3 V
ESP32-S3R8	—	8 MB (Octal SPI)	-40 ~ 65	3.3 V
ESP32-S3R8V	—	8 MB (Octal SPI)	-40 ~ 65	1.8 V
ESP32-S3FH4R2	4 MB (Quad SPI)	2 MB (Quad SPI)	-40 ~ 85	3.3 V

图 4.5.4.1 芯片型号对比

这个表格下定义了每个芯片型号 VDD_SPI 电压。由于正点原子 ESP32S3 开发板的模组选择的是 ESP32-S3-WROOM-1-N16R8，而它的主控芯片为 ESP32R8，所以根据上图的内容，我们会发现 ESP32R8 芯片的 VDD_SPI 电压为 3.3V。接着我们来看一下 GPIO45 号管脚的定义，如下图所示：

表 2-12. VDD_SPI 电压控制					
EFUSE_VDD_SPI_FORCE	GPIO45	eFuse ¹	电压	VDD_SPI 电源 ²	
0	0	忽略	3.3 V	VDD3P3_RTC 通过 R _{SPI} 供电	
	1		1.8 V	Flash 稳压器	
1	忽略	0	1.8 V	Flash 稳压器	
		1	3.3 V	VDD3P3_RTC 通过 R _{SPI} 供电	

¹ eFuse: EFUSE_VDD_SPI_TIEH² 请参考章节 2.5.2 电源管理

图 4.5.4.2 VDD_SPI 电压控制

从上图可以看到，电压有两种控制方式，具体取决于 EFUSE_VDD_SPI_FORCE 的值。如果这个值为 0，那么 VDD_SPI 电压取决于 GPIO45 的电平值。如果 GPIO45 的电平值为 0，VDD_SPI 电压为 3.3V；否则为 1.8V。相反，如果 EFUSE_VDD_SPI_FORCE 为 1，VDD_SPI 电压取决于 eFuse（表示 flash 电压调节器是否短接至 VDD_RTC_IO）。如果 eFuse 为 0，VDD_SPI 电压值为 1.8V；否则为 3.3V。

② ROM 日记打印控制

系统启动过程中，ROM 代码日志可打印至 UART 和 USB 串口/JTAG 控制器。我们可通过配置寄存器和 eFuse 可分别关闭 UART 和 USB 串口/JTAG 控制器的 ROM 代码日志打印功能。详细信息请参考《ESP32-S3 技术参考手册》→第 8 章节芯片 Boot 控制。

④ JTAG 信号源控制

在系统启动早期阶段，GPIO3 可用于控制 JTAG 信号源。该管脚没有内部上下拉电阻，strapping 的值必须由不处于高阻抗状态的外部电路控制。如图所示，GPIO3 与 EFUSE_DIS_PAD_JTAG、EFUSE_DIS_USB_JTAG 和 EFUSE_STRAP_JTAG_SEL 共同控制 JTAG 信号源。

表 2-13. JTAG 信号源控制				
eFuse 1 ^a	eFuse 2 ^b	eFuse 3 ^c	GPIO3	JTAG 信号源
0	0	0	忽略	USB 串口/JTAG 控制器
		1	0	JTAG 管脚 MTDI、MTCK、MTMS 和 MTDO
		1	1	USB 串口/JTAG 控制器
0	1	忽略	忽略	JTAG 管脚 MTDI、MTCK、MTMS 和 MTDO
1	0	忽略	忽略	USB 串口/JTAG 控制器
1	1	忽略	忽略	JTAG 关闭

^a eFuse 1: 表示是否永久禁用 JTAG 功能^b eFuse 2: 表示是否禁用 usb_serial_jtag 模块的 usb 转 jtag 功能^c eFuse 3: 表示是否使能使用 strapping GPIO3 选择 usb_to_jtag 或 pad_to_jtag 的功能

图 4.5.4.3 JTAG 信号源控制

注意: ESP32-S3 系统中有一块 4Kbit 的 eFuse, 其中存储着参数内容。相关内容请看《esp32-s3_technical_reference_manual_cn.pdf》技术参考手册→第 5 章节 eFuse 控制器。

4.6 ESP32-S3 启动流程

本文将会介绍 ESP32-S3 从上电到运行 app_main 函数中间所经历的步骤 (即启动流程)。从宏观上, 该启动流程可分为如下 3 个步骤。

①: 一级引导程序, 它被固化在 ESP32-S3 内部的 ROM 中, 它会从 flash 的 0x00 处地址加载二级引导程序至 RAM 中。

②: 二级引导程序从 flash 中加载分区表和主程序镜像至内存中, 主程序中包含了 RAM 段和通过 flash 高速缓存映射的只读段。

③: 应用程序启动阶段运行, 这时第二个 CPU 和 freeRTOS 的调度器启动, 最后进入 app_main 函数执行用户代码。

下面作者根据 IDF 库相关的代码来讲解这三个引导流程, 如下:

一、一级引导程序

该部分程序是直接存储在 ESP32-S3 内部 ROM 中, 所以普通开发者无法直接查看, 它主要是做一些前期的准备工作 (复位向量代码), 然后从 flash 0x00 偏移地址中读取二级引导程序文件头中的配置信息, 并使用这些信息来加载剩余的二级引导程序。

二、二级引导程序

该程序是可以查看且可被修改, 在搭建 ESP-IDF 环境完成后, 可在 esp-idf\components\bootloader\subproject\main\路径下找到 bootloader_start.c 文件, 此文件就是二级引导程序启动处。首先我们克隆 ESP-IDF 库, 克隆过程如下所示。

```
root@DESKTOP-QH7611H:~# git clone https://github.com/espressif/esp-idf.git
Cloning into 'esp-idf'...
remote: Enumerating objects: 527128, done.
remote: Counting objects: 100% (84773/84773), done.
remote: Compressing objects: 100% (2741/2741), done.
remote: Total 527128 (delta 82783), reused 82032 (delta 82032), pack-reused 442355
Receiving objects: 100% (527128/527128), 238.92 MiB | 1.19 MiB/s, done.
Resolving deltas: 100% (392209/392209), done.
Updating files: 100% (13058/13058), done.
root@DESKTOP-QH7611H:~#
```

图 4.6.1 克隆 ESP-IDF 库

克隆完成后, 使用 VSCode 打开 ESP-IDF 库, 接着找到 bootloader_start.c, 如下图所示。

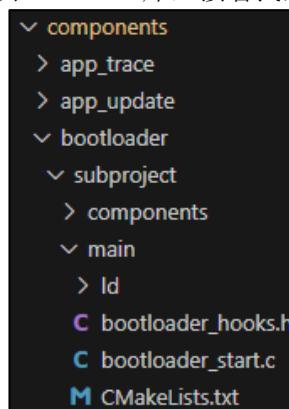


图 4.6.2 bootloader_start.c 文件路径

在这个文件下, 找到 call_start_cpu0 函数, 此函数是 bootloader 程序, 如下是 bootloader 程序的部分代码。

```
/*
ROM 引导加载程序完成从闪存加载第二阶段引导加载程序之后到达这里
*/
void __attribute__((noreturn)) call_start_cpu0(void)
{
    if (bootloader_before_init) {
```

```

        bootloader_before_init();
    }

    /* 1. 硬件初始化:清楚 bss 段、开启 cache、复位 mmc 等操作
       bootloader_support/src/esp32s3/bootloader_esp32s3.c */
    if (bootloader_init() != ESP_OK) {
        bootloader_reset();
    }

    if (bootloader_after_init) {
        bootloader_after_init();
    }

    /* 2. 选择启动分区的数量: 加载分区表, 选择 boot 分区 */
    bootloader_state_t bs = {0};
    int boot_index = select_partition_number(&bs);

    if (boot_index == INVALID_INDEX) {
        bootloader_reset();
    }

    /* 3. 加载应用程序映像并启动
       bootloader_support/src/esp32s3/bootloader_utility.c */
    bootloader_utility_load_boot_image(&bs, boot_index);
}

```

ESP-IDF 使用二级引导程序可以增加 FLASH 分区的灵活性 (使用分区表), 并且方便实现 FLASH 加密, 安全引导和空中升级 (OTA) 等功能。主要的作用是从 flash 的 0x8000 处加载分区表 (请看在线 ESP32-IDF 编程指南分区表章节)。根据分区表运行应用程序。

三、三级引导程序

应用程序的入口是在 esp-idf/components/esp_system/port/路径下的 `cpu_star.c` 文件, 在此文件下找到 `call_start_cpu0` 函数 (端口层初始化函数)。这个函数由二级引导加载程序执行, 并且从不返回。因此你看不到是哪个函数调用了它, 它是从汇编的最底层直接调用的。

这个函数会初始化基本的 C 运行环境 (“CRT”), 并对 SOC 的内部硬件进行了初始配置。执行 `call_start_cpu0` 函数完成之后, 在 `components\esp_system\startup.c` 文件下调用 `start_cpu0`(在 110 行中, 弱关联 `start_cpu0_default` 函数)系统层初始化函数, 如下 `start_cpu0_default` 函数的部分代码。

```

static void start_cpu0_default(void)
{
    ESP_EARLY_LOGI(TAG, "Pro cpu start user code");
    /* 获取 CPU 时钟 */
    int cpu_freq = esp_clk_cpu_freq();
    ESP_EARLY_LOGI(TAG, "cpu freq: %d Hz", cpu_freq);

    /* 初始化核心组件和服务 */
    do_core_init();

    /* 执行构造函数 */
    do_global_ctors();

    /* 执行其他组件的 init 函数 */
    do_secondary_init();
    /* 开启 APP 程序 */
    esp_startup_start_app();
    while (1);
}

```

到了这里, 就完成了二级程序引导, 并调用 `esp_startup_start_app` 函数进入三级引导程序, 该函数的源码如下:

```

/* components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c */
/* 开启 APP 程序 */
void esp_startup_start_app(void)
{ /* 省略部分代码 */ }

```

```
/* 新建 main 任务函数 */
esp_startup_start_app_common();

/* 开启 FreeRTOS 任务调度 */
vTaskStartScheduler();
}

/* components/freertos/FreeRTOS-Kernel/portable/port_common.c */
/* 新建 main 任务函数 */
void esp_startup_start_app_common(void)
{
    /* 省略部分代码 */
    /* 创建 main 任务 */
    portBASE_TYPE res = xTaskCreatePinnedToCore(&main_task, "main",
                                                ESP_TASK_MAIN_STACK, NULL,
                                                ESP_TASK_MAIN_PRIO, NULL,
                                                ESP_TASK_MAIN_CORE);
    assert(res == pdTRUE);
    (void)res;
}

/* main 任务函数 */
static void main_task(void* args)
{
    /* 省略部分代码 */
    /* 执行 app_main 函数 */
    app_main();
    vTaskDelete(NULL);
}
```

从上述源码可知，首先在 `esp_startup_start_app_common` 函数调用 FreeRTOS API 创建 `main` 任务，然后开启 `FreeRTOS` 任务调度器，最后在 `main` 任务下调用 `app_main` 函数（此函数在创建工程时，在 `main.c` 下定义的）。

同理，ESP32S3 的 Arduino 也是以这种方式启动的。此外，Arduino 的 `app_main` 函数是在 `Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\main.cpp` 文件中定义。在函数中，会进行 `UART`、`USB` 和特定库的初始化操作。

第五章 Arduino 开发环境搭建

本章，我们将进入实际操作阶段，逐步搭建 Arduino 的开发环境。

本章分为如下几个小节：

5.1 开发方式的选择

5.2 开发系统的选择与环境搭建

5.1 开发方式的选择

ESP32 的开发方式主要有三种：ESP-IDF、Arduino 和 MicroPython。



图 5.1.1 开发 ESP32 方式

1, ESP-IDF: ESP-IDF 是乐鑫官方推出的开发框架，专门为 ESP32 和其他一些 ESP 系列芯片设计。它提供了一套完整的开发工具和库，可以帮助开发者快速地开发和调试 ESP32 应用程序。ESP-IDF 支持 C/C++ 语言，并提供了一套完整的 API，可以控制 ESP32 的各种功能和外设。此外，ESP-IDF 还提供了一个在线编译器和调试器，可以让开发者在云端进行开发和调试。

2, Arduino: Arduino 是一种流行的开源电子原型平台，包括一系列的开发板和开发环境。Arduino 提供了一种基于 C/C++ 的语言，使得开发者可以更容易地控制和编程 ESP32。Arduino 开发环境还提供了大量的库和函数，可以帮助开发者快速地构建和测试他们的代码。Arduino 还支持图形化编程，使得初学者和非专业人士也可以轻松地进行开发。本教程选择此开发方式。

3, MicroPython: MicroPython 是一种精简的 Python 3 语言，可以运行在 ESP32 和其他一些微控制器上。它提供了一种简单的方式来编程和控制 ESP32，而且由于 Python 是一种高级语言，它使得开发过程相对快速和简单。开发者可以使用 MicroPython 进行快速原型设计和开发，并且由于 Python 是一种解释型语言，所以可以直接在 ESP32 上运行代码，无需进行编译。

这三种开发方式各有其优点，开发者可以根据自己的需求和技能水平选择适合自己的开发方式。对于初学者和非专业人士来说，Arduino 和 MicroPython 是一种很好的选择，因为它简单易学，可以快速上手。对于专业人士和对性能有更高要求的开发者来说，ESP-IDF 可能是更好的选择，因为它们提供了更高级的开发工具和更强大的控制能力。另外，正点原子 ESP32-S3 最小系统板为开发者提供了 MicroPython、Arduino 和 ESP-IDF 三种开发方式的相关例程和教程，这使得开发者可以根据自己的需求和技能水平选择适合自己的开发方式。

5.2 环境搭建

5.2.1 Arduino IDE2 软件安装包下载

Arduino 集成开发环境，即 Arduino IDE 有两个版本，Arduino IDE1 和 Arduino IDE2。本教程是采用 Arduino IDE2 去编写例程源码的，所以在这里需要下载 Arduino IDE2 软件安装包，可直接到官网地址获取，地址为：<https://www.arduino.cc>。在这里我们将一步一步带大家把软件安装包下载下来，首先进入官网如下所示：

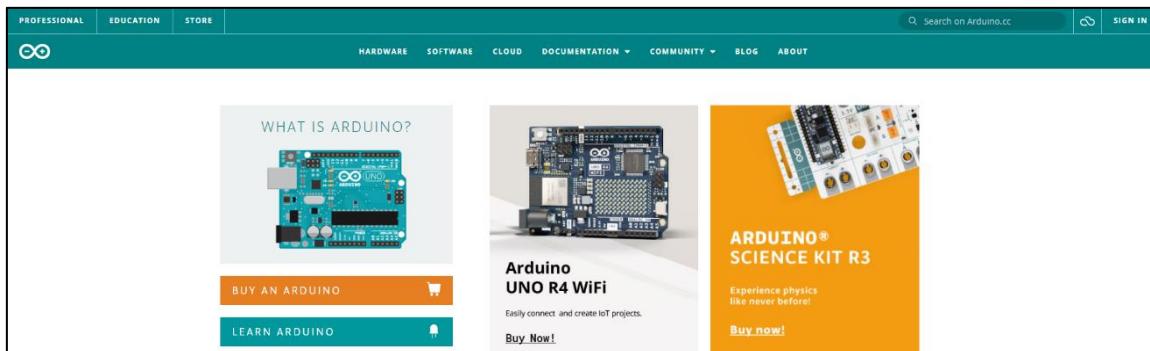


图 5.2.1.1 Arduino 官网

点击“SOFTWARE”选项卡，选择“IDE2”，如下图所示：



图 5.2.1.2 打开 IDE2 界面过程

然后就来到 Arduino IDE2 介绍页面，这个页面除了有下载按钮还有不少文字介绍信息。

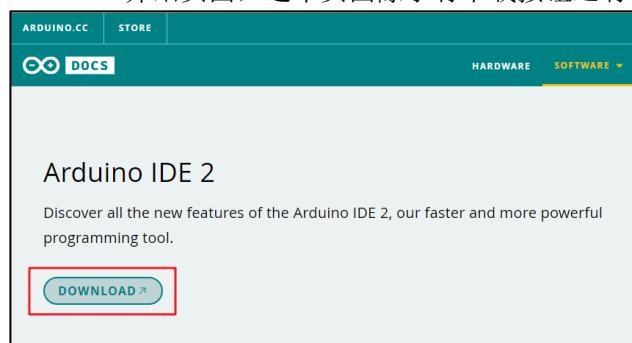


图 5.2.1.3 Arduino IDE2 介绍页面

按下“DOWNLOAD”按钮进行跳转，来到 Arduino IDE2 下载页面，如下图所示：

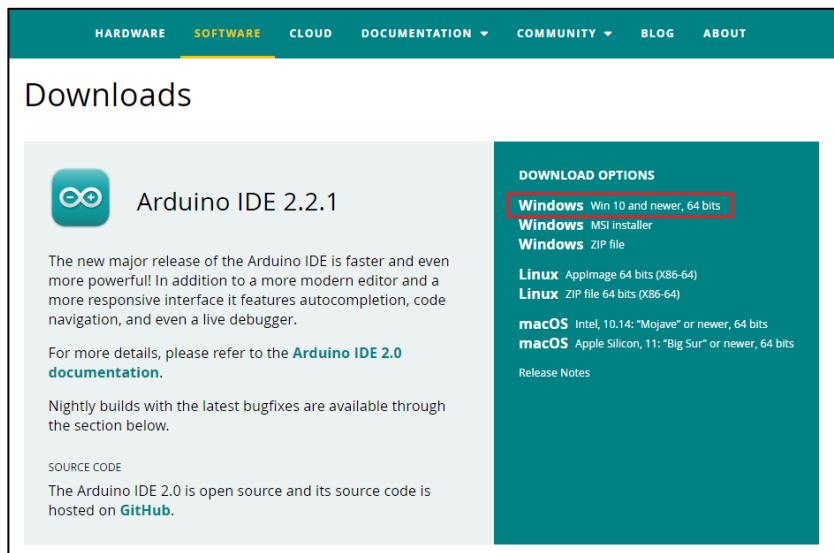


图 5.2.1.4 Arduino IDE2 下载界面

根据自己电脑系统选择对应版本下载，我们使用 Windows，因此选择 Windows 版本（Win 10 and newer,64bits），除此之外还有 ZIP 压缩包下载以及 MSI 下载。

点击“Windows（Win 10 and newer,64bits）”，即可准备免费下载。这时候会弹出一个界面表示是否需要资金支持以下该团队，如下图所示，当然你可以忽略，直接点击“JUST DOWNLOAD”。



图 5.2.1.5 Arduino IDE2 下载界面 2

当点击“JUST DOWNLOAD”按钮后，这时候还没有正式到下载，还会弹出一个页面表示是否需要订阅 Arduino 信息，如下图所示，你可以继续点击“JUST DOWNLOAD”。

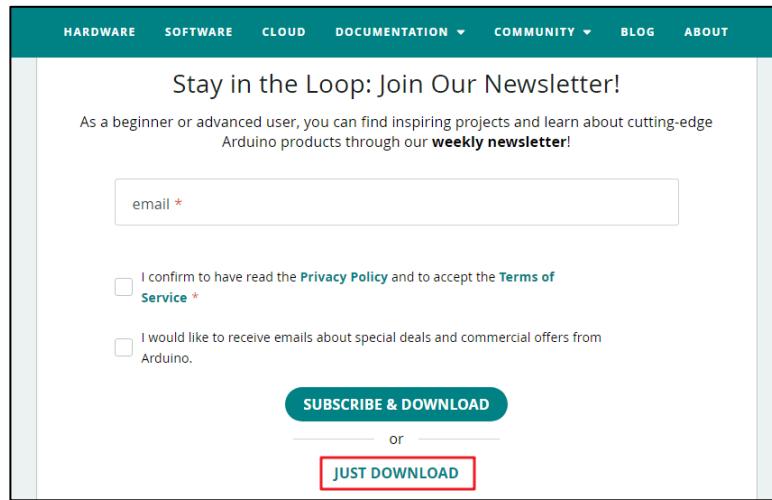


图 5.2.1.6 Arduino IDE2 下载界面 3

当点击“JUST DOWNLOAD”按钮后，这时候就会有一个弹窗“新建下载任务”，如下图所示，这时可选择该软件安装包在电脑的存放路径，然后点击“下载”后，开始下载。

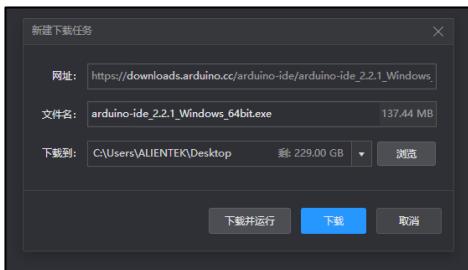


图 5.2.1.7 新建 Arduino IDE2 下载任务

如果觉得官方下载麻烦，也可以到我们提供的开发板资料内获取 Arduino 软件安装包，软件存放路径“光盘→6，软件资料→1，软件→3，Arduino 开发工具”。

在该目录下还有一个“Arduino IDE 添加 ESP32 的软件包”文件夹，顾名思义，这个文件夹是解决的是如何在 Arduino IDE 上开发 ESP32 的问题。Arduino IDE 只是一个开发环境，默认支持的是 AVR-Arduino 硬件平台，如果需要在 ESP32 上使用 Arduino，则需要在 Arduino IDE 上安装 ESP32 的库。

5.2.2 Arduino IDE2 软件安装

从前面选择电脑存放路径下找到“arduino-ide_2.2.1_Windows_64bit.exe”文件（也可以在正点原子提供的软件资料找到安装文件），双击该 exe 文件，如下图所示。

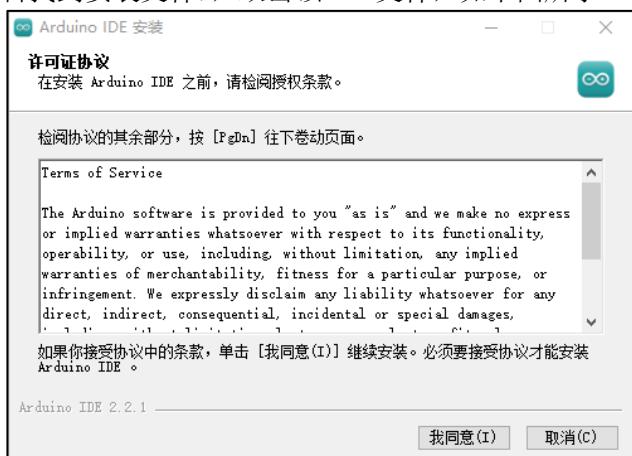


图 5.2.2.1 Arduino IDE 安装-许可证协议

点击“我同意”，然后就是 Arduino IDE 安装选项，具体界面如下图所示：

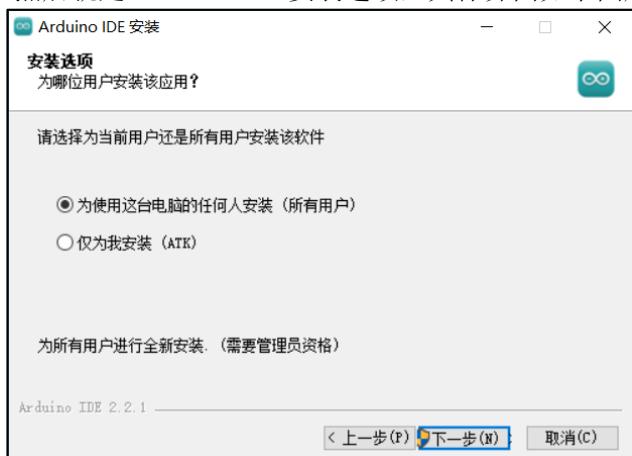


图 5.2.2.2 Arduino IDE 安装-安装选项-为哪位用户安装应用

通常情况下，选择“为使用这台电脑的任何人安装（所有用户）”，然后点击“下一步”。进入许可协议界面，如下图所示。

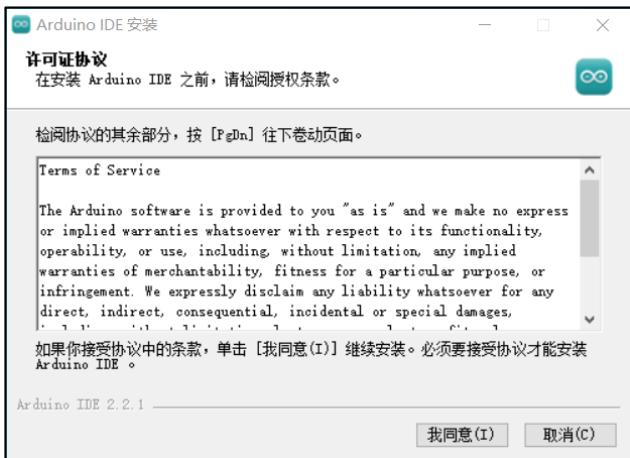


图 5.2.2.3 Arduino IDE 安装-许可证协议界面

在这里需要点击“我同意”，不然无法进行下一步的。然后就是选择安装位置，在这里就需要用户自己去选择安装的目标文件夹，如下图所示。



图 5.2.2.4 Arduino IDE 安装-选定安装位置

然后点击“安装”即可进入软件安装流程，如下图所示。



图 5.2.2.5 Arduino IDE 安装-正在安装

当安装完成，即可见到如下安装完成界面，点击“完成”即可，然后就会启动 Arduino IDE2 软件。



图 5.2.2.6 Arduino IDE 安装-安装完成界面

初次打开 Arduino IDE 软件, 需要等待一段时间, 这期间会安装一些库以及安装一些驱动, 比如: Adafruit Industries 的 LLC 端口、Arduino srl 的 Arduino USB Driver、Arduino SA 的 Arduino USB Driver 以及 Arduino LLC 的 Genuino USB Driver, 如下图所示。



图 5.2.2.7 首次打开安装的一些库包



图 5.2.2.8 驱动安装图

5.2.3 认识 Arduino IDE2

Arudino IDE2, Arduino 的集成开发环境, 具有程序编辑、调试、编译、上传、库管理等功能。Arduino IDE2 的主界面如图所示:



图 5.2.3.1 Arduino IDE2 主界面

首次打开 Arduino IDE2 软件, 语言默认为英文, 怎么设置成上图的中文呢? 点击“File→Preferences”, 进入到首选项界面, 对 Language 选项设为中文即可, 如下图所示:

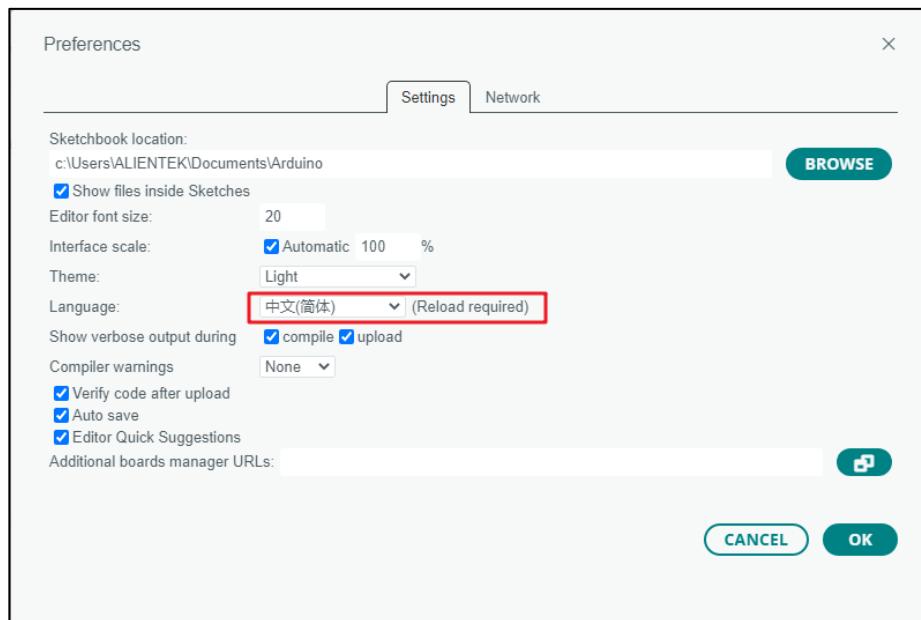


图 5.2.3.2 Arduino IDE2 设置为中文界面

在首选项中，我们进行如下设置：

- ① 项目文件夹地址，即工程存放的位置，当我们新建工程时，默认推荐存放的地方，当然你也可以自行设置存放文件夹。这里有一个选项是是否在 IDE 中进行显示项目中的文件夹，假如打勾，这时候通过左侧工具栏的第一个按钮即可显示该新项目文件夹的文件情况。
- ② 设置编辑器字体大小，以及设置界面比例，这些都是默认操作即可。当你觉得字体太小了，自行调整编辑器字体大小参数即可，觉得编辑器界面调小，也可以适当调整界面比例参数。
- ③ 设置颜色主题，可选为：明亮、暗黑、明亮对比和暗黑对比。
- ④ 设置编辑器语言，多种语言可以设置，设置后会进行软件重启。
- ⑤ 显示详细输出，这里我把编译和上传都勾选了，即项目编译和上传时，在信息显示窗口都会打印出相关信息，便于了解整个过程的执行情况。
- ⑥ 编译器警告，这里设置的是无。
- ⑦ 上传后验证代码，这里可打勾。
- ⑧ 自动保存，这里是要打勾的，以防电脑蓝屏等突发情况导致工程代码丢失。
- ⑨ 编辑快速建议，这里是要打勾的。比如我要用 `Serial.printf` 函数，这时候会提示函数的传参是什么，便于我们更好更快的编写对的函数。
- ⑩ 其他开发板管理器地址，当你要添加其他开发板时，假如是 Arduino 非官方的开发板，这时候你在 IDE 上是搜索不到，所以我们就需要把这个开发板的地址添加进来。打个比方，要添加树莓派 pico，就需要获取对应的地址呢？点击旁边的编辑按钮，进入到“其他开发板管理器地址页面”，然后点击下图中的①“获取支持的非官方开发板地址列表”，进入到一个网页 <https://github.com/arduino/Arduino/wiki/Unofficial-list-of-3rd-party-boards-support-urls>，在该网页中查询你要添加的芯片的 json 文件地址：https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json，然后把它复制到“其他开发板管理器地址页面”的②输入框，软件会自动检测，然后下载对应的 json 文件到“C:\Users\ALIENTEK\AppData\Local\Arduino15”文件夹下，如下图所示：

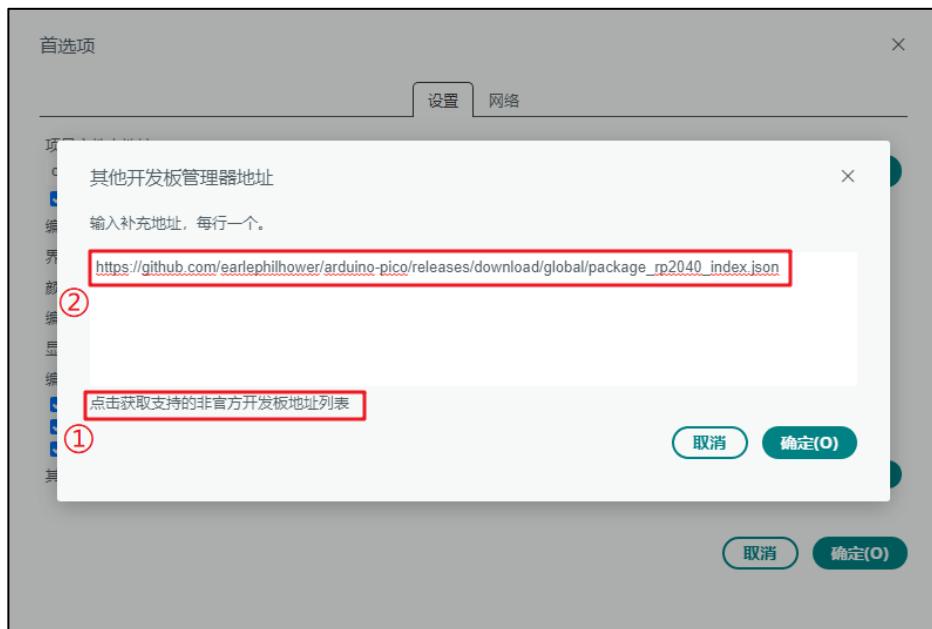


图 5.2.3.3 Arduino IDE2 添加非官方开发板步骤

注意：假如 json 文件下载不成功，就可以手动下载，把它放置于“C:\Users\ALIENTEK\AppData\Local\Arduino15”文件夹下就可以了。后面就可以在开发板管理器中搜索 rp2040 进行安装开发板，具体操作过程可以参考后面的“5.2.5 安装 arduino-esp32 库”小节的说明。

接下来对 Arduino IDE2 的工具栏（橙色框框）进行说明，如下表所示：

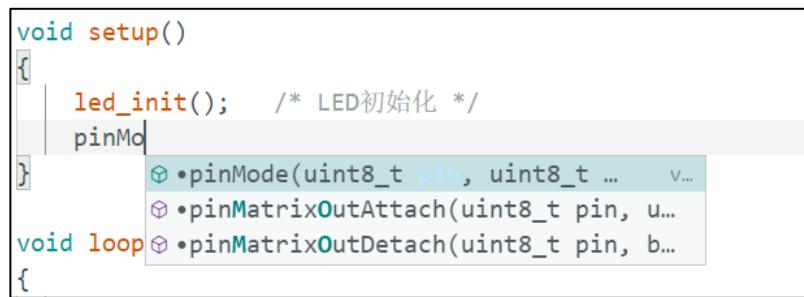
按钮	功能描述
	验证：检查程序是否有错误，如没有错误就将程序编译成二进制文件（每次编译都会自动保存文件）
	上传：编译程序，并将编译后的二进制文件上传到 Arduino 控制器的 MCU 中
	开始调试：目前官方仅支持 10 个板子调试，不支持 esp32，所以该功能未用到，并且调试还需要硬件的调试器
选择开发板	选择开发板：设置开发板
	串口绘图仪：用于跟踪从你的 Arduino 板接收的不同数据和变量，常用于测试和校准传感器，比较数值等
	串口监视器：打开串口监视器窗口，通过串口监视器，可以查看串口接收和发送的数据
	项目文件夹：显示“图 5.2.3.2 的项目文件夹地址”的文件情况
	开发板管理器：Arduino IDE 可以选择安装的芯片/板子
	库管理：用于安装一些器件驱动库
	调试：调试时，可看到 Debug 过程
	搜索：用于搜索函数等

表 5.2.3.1 Arduino IDE2 的工具栏

接下来，看一下 Arduino IDE2 相对 Arduino IDE1 新增功能：

- ① 代码自动填充功能

输入时，编辑器可以根据你的代码和你包含的库建议变量和函数自动完成：



```

void setup()
{
    led_init(); /* LED初始化 */
    pinMode
}

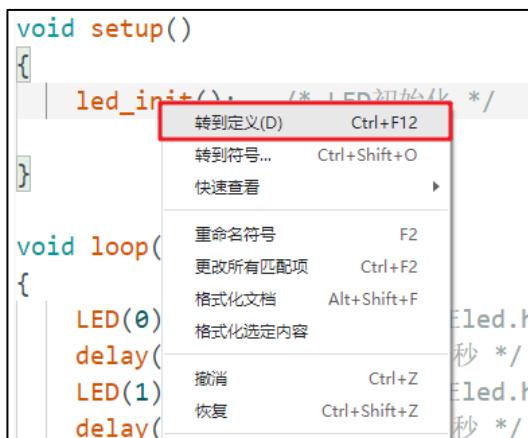
void loop
{
}

```

图 5.2.3.4 代码补全功能

注意：要对“文件→首选项→编辑快速建议”的功能打钩才会有这个功能。

② 变量或函数跳转功能



```

void setup()
{
    led_init(); /* LED初始化 */
}

void loop
{
    LED(0)
    delay()
    LED(1)
    delay()
}

```

图 5.2.3.5 代码跳转功能

右键单击变量或函数时，上下文菜单将提供导航快捷方式以跳转到声明它们的行（和文件），当然也可以通过快捷键“Ctrl+F12”达到目的。

基于这个功能，我们可以比较轻松了解函数的调用关系，可以了解到底层。

③ Arduino Cloud 云端保存

对于拥有多台工作电脑的人来说，云端保存是一个非常有用的功能。这个功能我们并没有用到，有兴趣的小伙伴可以自行尝试使用以下。

④ 串行绘图仪

IDE2 的串行绘图仪，如下图所示，用于跟踪从 Arduino 板接收的不同数据和变量，可以让我们更直观看到数据。通常用于测试和校准传感器、比较值和其他类似场景。



图 5.2.3.6 串行绘图仪

在 IDE1，串行绘图仪和串行监控器只能有其一个工作，而 IDE2 支持两个功能同时工作。

5.2.4 arduino-esp32 库介绍

Arduino-esp32 库就是一个在 Arduino 平台上开发 ESP32 的插件，它为 Arduino 环境下的 ESP32 芯片提供了支持。它允许使用熟悉的 Arduino 函数和库编写代码，并直接在 ESP32 上运行。

Arduino-esp32 库支持对 ESP32、ESP32-S2、ESP32-S3、ESP32-C3、ESP32-C6 和 ESP32-H2 进行开发，还提供了很多基础库。这些基础库涵盖了芯片的所有外设，如下表所示。

基础库	基础库说明
EEPROM	EEPROM 库，管理内部的 FLASH
Preferences	ESP32 NVS 加密 flash
ESP32	ESP32 外设的示例
Wire	I2C 库，与 IIC 接口相关
SPI	SPI 库，与 SPI 接口相关
I2S	I2S 库，与 I2S 接口相关
SD	SD 卡驱动库，SD 卡使用的是 SPI 接口
SD_MMC	SD 卡驱动库，SD 卡使用的是 4 线接口
USB	USB 库
FS	文件系统虚拟框架
FFat	文件系统，FAT 文件系统
LittleFS	文件系统，LittleFS 文件系统
SPIFFS	文件系统，SPIFFS 文件系统
Ticker	定时任务库
BLE	蓝牙相关库，低功耗蓝牙 V4.2 客户端/服务器 框架
BluetoothSerial	蓝牙相关库，经典蓝牙，适用 ESP32，不适用 ESP32-S2/C3/S3
SimpleBLE	蓝牙相关库，低功耗蓝牙广播
ArduinoOTA	无线更新-OTA 固件更新
Update	无线更新-OTA 固件更新
HTTPUpdate	无线更新-OTA 固件更新
HTTPUpdateServer	无线更新-OTA 固件更新
Insights	ESP Insights 功能，通过 WiFi 把设备固件运行状态和日报上报云
NetBIOS	解析 NetBIOS 名称
WiFi	WiFi 基础功能，包括 AP、STA、SCAN、TCPClient、TCPServer、UDP
WebServer	局域网 Web 服务器功能
WiFiClientSecure	加密 WiFi 客户端
HTTPClient	http 客户端功能，兼容 WiFiClientSecure
WiFiProv	WiFi 配网功能
AsyncUDP	异步任务驱动的 UDP 数据客户端/服务器
DNSServer	真正的 DNS 域名服务
ESPmDNS	局域网本地发现功能
Ethernet	以太网
RainMaker	ESP RainMaker AIoT 平台服务

表 5.2.4.1 arduino-esp32 基础库

5.2.5 安装 arduino-esp32 库

前面也提及到 Arduino IDE 默认支持的是 AVR-Arduino 硬件平台，如果想在 IDE 上对 ESP32 进行开发，就必须安装 arduino-esp32 库。

下面提供两种方式安装 esp32 到 Arduino IDE。

安装方式一：使用 Arduino IDE 的开发板管理器来安装

安装步骤如下：

- ① 打开 Arduino IDE，选择“工具→开发板→开发板管理器”菜单项或直接点击主界面左侧的开发板管理器 LOGO，然后在文本框中输入 esp32，选择 esp32 by Espressif System，点击安装，具体操作如下图所示。

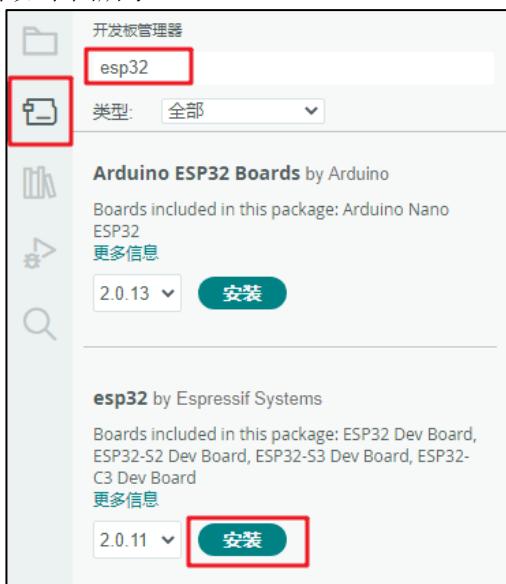


图 5.2.5.1 安装 ESP32 芯片包

(注：教程中版本为 V2.0.11，若版本更新，还是选择 V2.0.11 即可)

由于这里面的文件存放于 github 中，而服务器在国外，所以这种方式很大程度会失败，当然网上也有方法教如何可以成功，具体可以自行上网查阅。

注意：若在开发板管理器找不到 esp32，这时候就需要在“文件→首选项→其他开发板管理器地址”添加 esp32 地址“https://espressif.github.io/arduino-esp32/package_esp32_index.json”进去，具体操作如下：



图 5.2.5.2 添加 esp32 开发板地址

- ② 重新启动 Arduino，可到“工具→开发板→esp32”进行查看，这里面就有 ESP32 众多开发板在，针对正点原子的 ESP32-S3 开发板，直接选择“ESP32S3 Dev Module”即可，具体操作如下图。

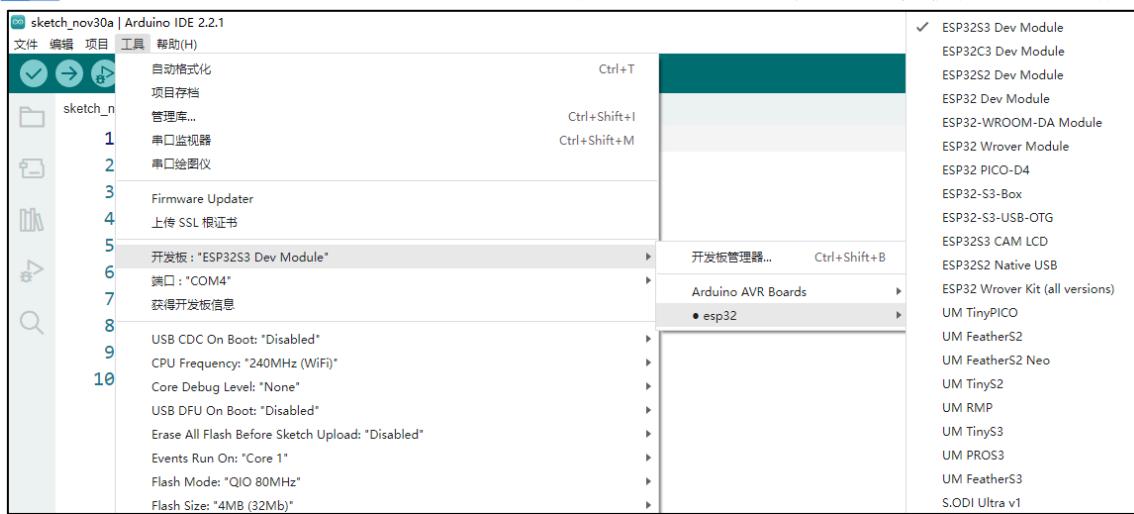


图 5.2.5.3 正点原子 ESP32-S3 开发板选择芯片

安装方式二：离线安装

这种方式，主要解决的是方式一中的第一步，Arduino IDE 很大可能从 github 中下载相关软件包失败，所以我们这里直接准备好了所需要的“Arduino_IDE 添加 ESP32 的软件包”，可以在“光盘→6，软件资料→1，软件→3，Arduino 开发工具”找到，其文件夹的文件如下图所示：



图 5.2.5.3 ESP32 的软件包

- ① 到 C:\Users\ALIENTEK\AppData\Local\Arduino15\staging\packages 路径下，把“Arduino_IDE 添加 ESP32 的软件包”文件夹下所有文件拷贝到该文件夹下，packages 文件夹拷贝后如下图所示。

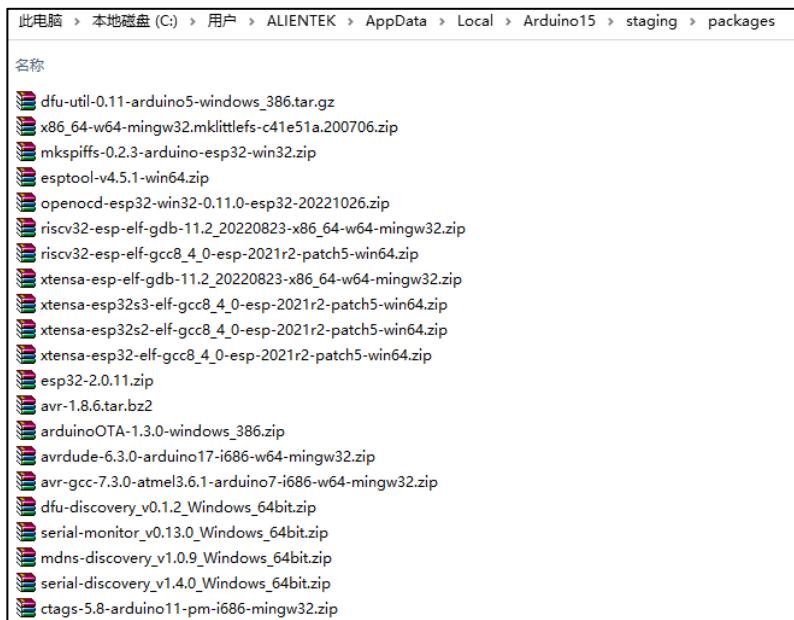


图 5.2.5.4 packages 文件夹

这里有两个注意点：①标红的 ALIENTEK 为本机的用户名，根据自己本机的用户名进行替换即可 ②假如看到发现没有 AppData 文件夹，需要在查看的菜单栏中对“隐藏的项目”前面的框框打钩，显示隐藏的项目

② 重复方式一的①操作即可，要选 V2.0.11 版本。因为这时候相关软件包已经被我们存放到 packages 中，这时候 Arduino IDE 就免去了从 github 中下载的步骤，直接对这些软件包进行安装，安装过程如下图所示。



图 5.2.5.5 Arduino-ESP32 安装过程

③ 与方式①的②步骤一致，这里不再赘述。

安装好 arduino-esp32 库，这时候就算搭建环境完成了。

第六章 新建 Arduino 工程

本章我们将讲解新建 ESP32S3 的 Arduino 工程的详细步骤，同时介绍一下对正点原子 ESP32-S3 开发板 Arduino 工程的设置。

本章将分为如下几个小节：

- 6.1 使用 Arduino IDE2 新建工程
- 6.2 ESP32-S3 Arduino 工程设置
- 6.3 运行 ESP32-S3 Arduino 第一个工程

6.1 使用 Arduino IDE2 新建工程

1、打开 Arduino IDE2 软件，通过点击“文件→新建项目”，具体操作如下：

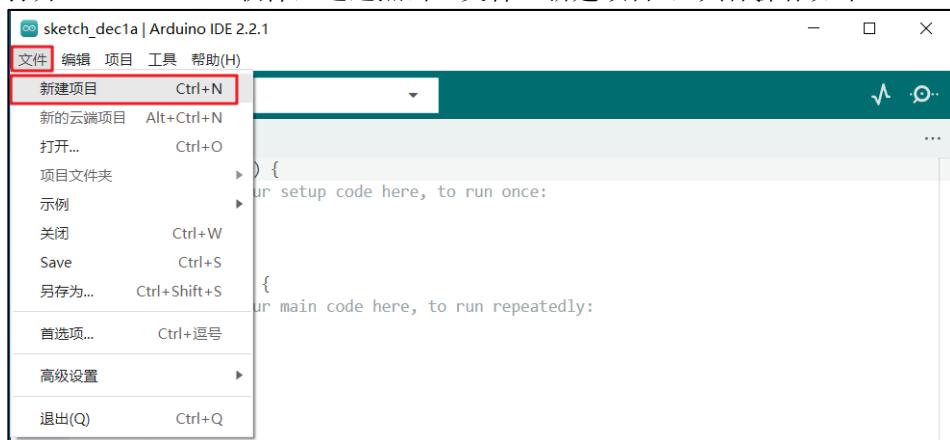


图 6.1.1 新建 Arduino 项目

通过以上操作后，我们得到一个干净的 Arduino 初始项目，界面如下：

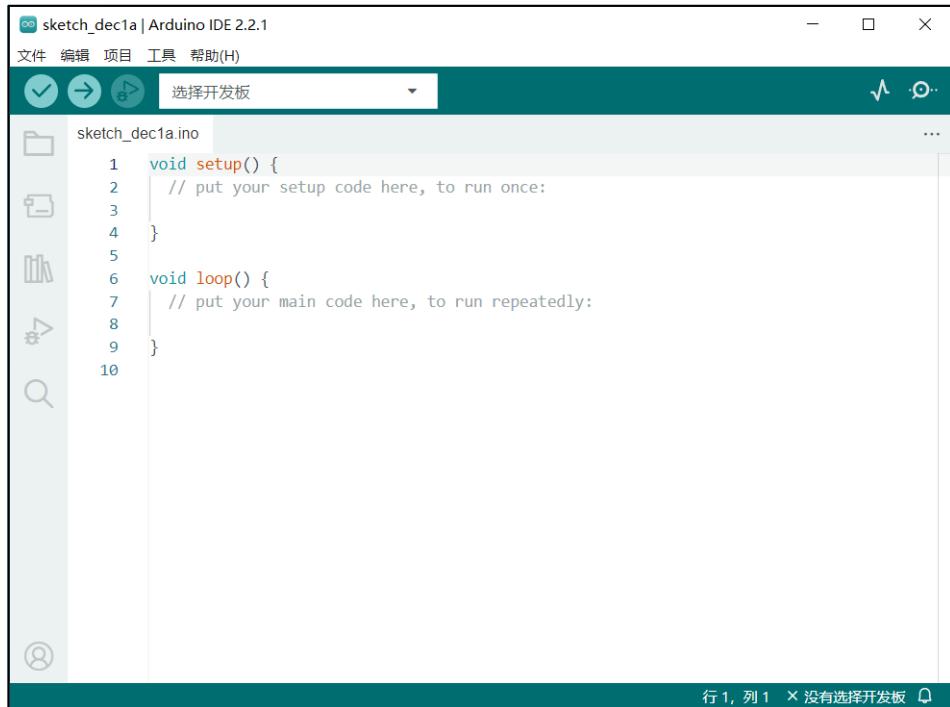


图 6.1.2 Arduino 初始项目界面

2、通过“文件→Save”操作或直接通过“Ctrl+S”快捷键，将工程保存到某个文件夹下，

然后再选择存放路径并对工程进行重命名，具体操作如下：

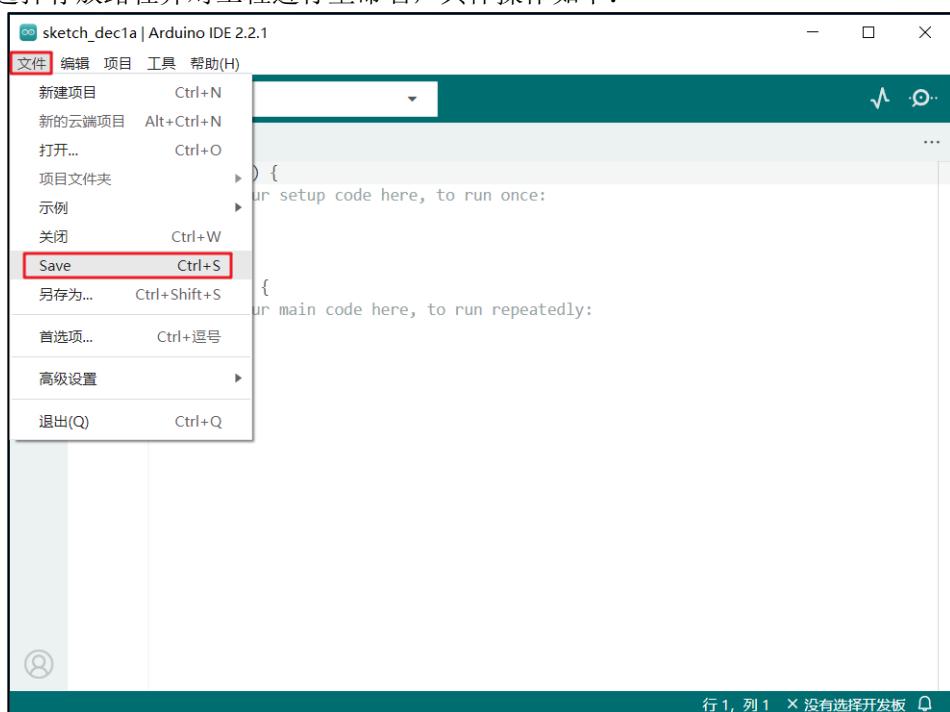


图 6.1.3 保存工程操作



图 6.1.4 保存到文件夹下

当我们设置好存放路径以及工程的命名时，按下保存，这时候软件进行刷新，这样子工程已经算是在你的电脑上存档好了。

注意：保存的工程命名，其实是有两重含义，第一重就是项目文件夹的命名，即在 Arduino_Project 文件夹下可看到有一个文件夹命名为 hello_world；第二重就是 hello_world 文件夹里面的 ino 文件的命名也是 hello_world，这两个名字一定要一致才可以。而 ino 文件是 Arduino IDE 产生的工程文件。

6.2 ESP32-S3 Arduino 工程设置

按照 6.1 小节的操作，我们已经新建好 Arduino 工程，但是还不是 ESP32-S3 工程，需要针对正点原子 ESP32-S3 开发板用的 ESP32-S3 模组进行设置。

在前面第四章中提及到 ESP32S3 模组因 PSRAM 和 FLASH 容量的差别，有好几个种类，而正点原子 ESP32-S3 开发板采用的模组型号为 ESP32-S3-WROOM-1-N16R8。

首先我们得选择该项目工程的开发板，这个操作在前面 5.2.5 也有介绍到，就是通过“工具 → 开发板 → esp32 → ESP32S3 Dev Module”，如下图所示：

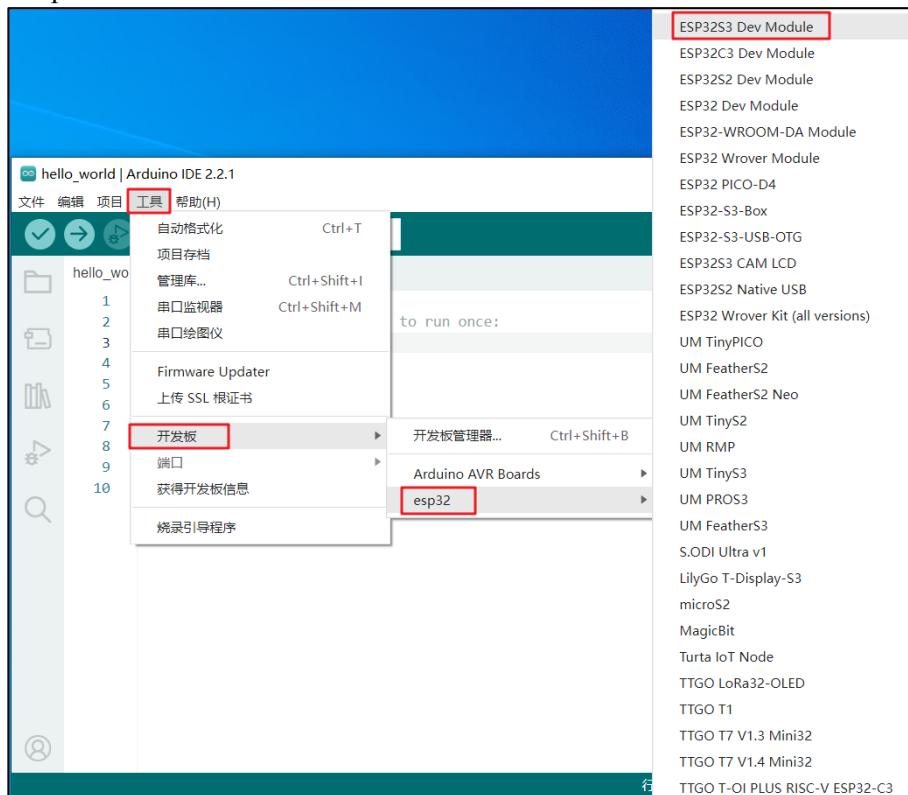


图 6.2.1 选择 ESP32S3 Dev Module

选择好 ESP32-S3 模组后，在“工具”界面下，出现开发板的一些配置项，如下图所示：

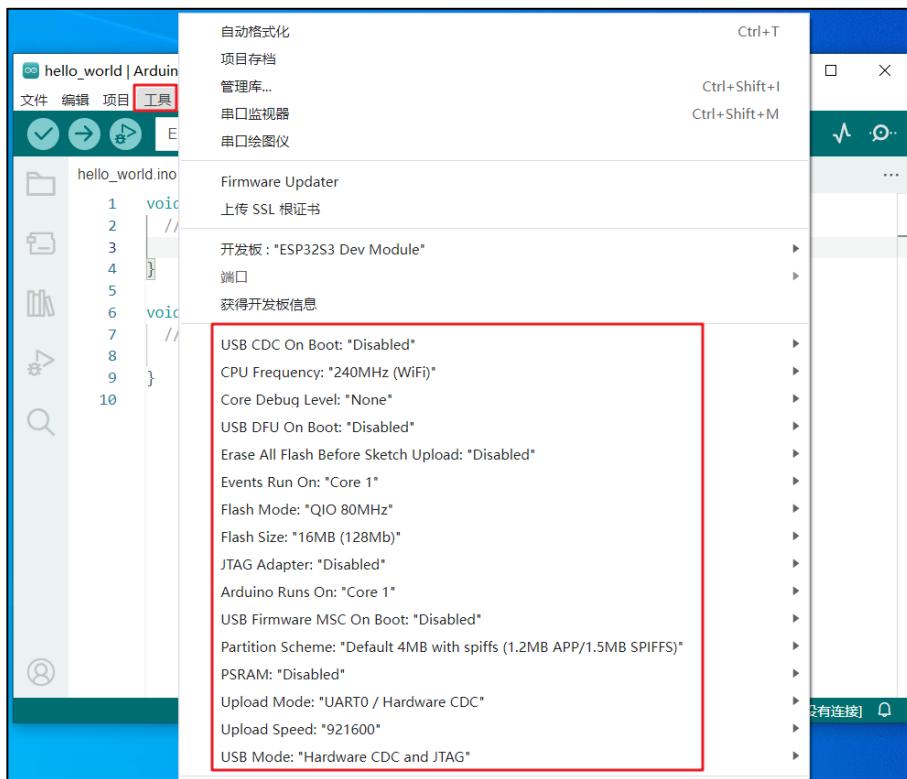


图 6.2.2 ESP32S3 模组参数

在这里，我们一一做一个科普。

1、USB CDC On Boot (USB Communications Device Class On Boot)

ESP32S3 芯片内置 USB 接口，支持 USB CDC，说明可以跳过 USB 转串口芯片对模组进行下载。有两个选项，**Enable**: 这种选项是要在没有串口芯片的情况下选择；**Disable**: 有串口芯片的情况下选择。我们的 ESP32-S3 开发板直接是通过带有串口芯片的电路进行下载程序的，所以这里直接选择 **Disable** 即可。

2、CPU Frequency

该选项顾名思义就是让你设置 CPU 的时钟频率，有 6 个频率可供选择：240MHz(WiFi)、160MHz(WiFi)、80MHz(WiFi)、40MHz、20MHz、10MHz。

想让 CPU 性能拉满，肯定是全速的 240MHz，这时候耗能也就越多，所以想降低功耗的话，降低 CPU 主频也是一个好办法，这里选择的依据就是要看你的使用场景。学习时，不考虑耗能，直接选择 240MHz 即可，在最佳性能下运行。

注意：

① 240MHz(WiFi)、160MHz(WiFi)、80MHz(WiFi)三个选项是可以保证 ESP32-S3 的 WIFI/BT 功能正常运行的 CPU 频率

② 40MHz、20MHz、10MHz 频率有点低，无法使 WIFI/BT 功能正常运行，只能保证 CPU 运行基本功能

3、Core Debug Level

该选项是用于选择要打印到串口的 Arduino 内核调试日志等级，有 6 个选择可供选择：None、Error、Warn、Info、Debug、Verbose。

None: 不打印输出任何内核调试日志

Error: 仅打印内核调试中错误级别的日志

Warn: 仅打印内核调试中警告级别及其以上等级的日志

Info: 仅打印内核调试中信息级别及其以上等级的日志

Debug: 仅打印内核调试中调试级别及其以上等级的日志

Verbose: 打印内核调试中所有的日志

我们一般情况无需关注内核中日志信息，除非是自己开发一些底层代码时，与内核中相关

功能有关，才需要打开内核调试日志，一般情况下建议直接选择 None 即可。

4、USB DFU On Boot(The USB Device Firmware Upgrade On Boot)

该选项是配置是否在 ESP32-S3 启动时，通过 USB 接口来升级固件。如果需要就选择 Enable，否则就 Disable。一般情况不需要升级固件，平时选择 Disable 即可。

5、Erase All Flash Before Sketch Upload

该选项是决定在 Arduino IDE 中上传代码时，是否需要把整个 FLASH 空间擦除。Enable 就是需要全面擦除 FLASH；Disable 就是不需要。这两者最直观的感觉就是下载时间，flash 需要全面擦除的话，慢了一倍多，另外，FLASH 的擦写次数是有限的，建议是选择 Disable。

6、Events Run On

该选项是配置 Arduino 中断事件在 ESP32-S3 的哪一个内核上运行，由于 ESP32-S3 有两个内核，分别为 Core 0 和 Core 1，所以我们可以将涉及到中断相关的事件处理代码放到单独的内核上运行，这样就可以防止正常业务代码运行，不用再被打断，这样程序运行时效率更高，可以做到实时处理外部中断事件。

这个选项跟后面的 Event Run On 选项就决定了双核的运行情况

7、Flash Mode

该选项是配置 ESP32-S3 芯片与 Flash 通信的模式，不同的 Flash 芯片，这里可以配置不同的通信模式和速度，这里有四种选择：QIO 80MHz、QIO 120MHz、DIO 80MHz 和 OPI 80MHz。

QIO (Quad I/O Fast Read): 使用四根 SPI 逻辑线用于 Flash 的读取和写入；

DIO (Dual I/O Fast Read): 使用两根 SPI 逻辑线用于 Flash 的读取和写入；

OPI (Octal I/O): 使用八根 SPI 逻辑线用于 Flash 的读取和写入

这里选择的依据，就是我们用的模组，其内部 FLASH 是通过什么接口跟 ESP32-S3 芯片进行通信的。我们采用的模组 ESP32-S3-WROOM-1-N16R8，模组内部通过 QSPI 接口跟芯片进行通信的，当然，选择普通 SPI 也可以。只不过为了提高速度，配置选择为 QIO 120MHz。

8、Flash Size

该选项是选择当前 ESP32-S3 上挂载的 Flash 容量大小，要根据实际的大小来选择，可以配置的参数有 4MB(32Mb)、8MB(64Mb) 和 16MB(128Mb)。我们采用的模组 FLASH 容量为 16MB，所以选择为 16MB(128b)。

9、JTAG Adapter

该选项是用来配置 JTAG Adapter 的，使用 JTAG 调试程序会更加的精细，我们可以设置短点、单步调试、查看变量等来调试程序，能够帮助我们迅速的定位问题，可以配置的参数有 Disabled、Integrated USB JTAG、FTDI Adapter、ESP USB Bridge。前面也有提及到 Arduino IDE 不支持 ESP32-S3 调试的，所以这里直接选择 Disabled 即可，在程序用比较见效的“printf 大法”调试程序。

10、Arduino Run On

该选项是为了配置 Arduino Core 任务代码所运行的 ESP32-S3 内核，这里的配置可以跟前面 Events Run On 所配置的内核不同，那么就可以在 ESP32-S3 两个内核上同时运行中断处理函数和 Arduino Core 任务代码。当然也可以选择成一样的内核，那么这样就可以节约一个内核的电量消耗，可以一定程度上降低功耗。

11、USB Firmware MSC On Boot

该选项是配置当使用 USB 连接到 ESP32-S3 开发板后，电脑上会弹出一个类似 U 盘的存储盘，这样我们就可以很方便的拉一个新的固件到这个存储盘里，这就是更新固件的一种新方法。有玩过 STM32 的小伙伴应该不陌生，就是读卡器实验，简单来说，就是在 FLASH 中移植了文件系统了。当你用 USB 线连接开发板的 USB 口，你的电脑就会出现 SPI FLASH 模拟的磁盘。

还是新手的话，建议直接 Disable 即可，后面用到文件系统的时候，再开就行了。

12、Partition Scheme

该选项配置磁盘分区的方案，就是将 FLASH 空间合理规划。在 Arduino IDE 中为我们设定了几个预设好的方案，我们根据实际情况选择对应的方案即可，这里我们选择的是 16MB(3MB APP/9.9MB FATFS)。

当然也可以自行创建自己的分区表方案，这部分内容大家自行去学习就好了。基于现有例程，16MB(3MB APP/9.9MB FATFS) 的分区表方案已经够用了。

13、PSRAM

该选项为了配置外接的 PSRAM 的 SPI 连接方式。有些 ESP32-S3 芯片只有内部 512KB SRAM, 没有外挂 PSRAM 对内存扩容, 那这里直接选择 Disabled 即可。如果是外挂了 PSRAM, 那就要根据 PSRAM 的容量大小来选择对应的 SPI 通信方式, 我们模组的 PSRAM 是通过 OPI 接口跟芯片通信, 所以这里选择为 OPI PSRAM。

14、Upload Mode

该选项就是为了配置使用哪种接口来上传 Arduino 代码到 ESP32-S3 开发板中, 具体配置参数为 UART0/Hardware CDC、USB-OTG CDC(TinyUSB)。

在这里我们选择的是 UART0/Hardware CDC, 结合前面 USB CDC On Boot 设置为 Disable 的情况, 所以在这里就确定了要使用 UART0 去上传代码到开发板中, 在后面我们就需要用 USB 线连接电脑 USB 口和开发板的串口进行下载程序。

15、Upload Speed

该选项是配置上传代码到开发板时的通信速率, 可以选择配置的参数有: 921600、115200、25600、230400、512000。我们串口电路用到的 CH340C 支持通信波特率为 50bps~2Mbps, 所以为了提高速度, 我们直接选择 921600。

16、USB Mode

该选项是为了配置, 当使用 ESP32-S3 开发板的 USB 接口时配置的模式, 如果我们一直使用的是 ESP32 S3 的串口上传代码模式, 配置 USB CDC On Boot 为 Disabled, 那么这里的 USB Mode 其实是用不到的。

最后推荐设置为如下图配置:

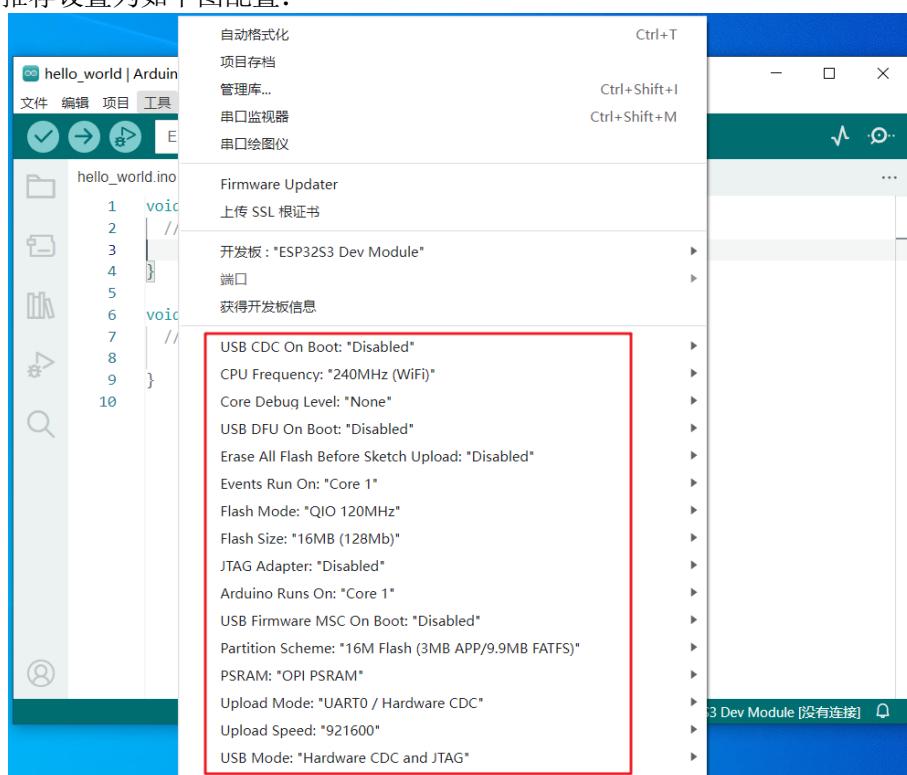


图 6.2.3 正点原子 ESP32S3 Arduino 工程配置参数

注意: 当我们把 ESP32S3 Arduino 工程参数配置好之后, 会自动保存下来。当你选择的开发板为 ESP32S3 Dev Module 时, 会自动配置好。只不过当我们把工程移动到别的地方, 这时候工程就需要重新进行开发板选择以及端口选择。

6.3 运行 ESP32-S3 Arduino 第一个工程

经过 6.1 和 6.2 小节操作, 我们接下来就是见证奇迹的时候了, 向我们 Arduino 说个 hello。首先, 用串口线连接电脑以及开发板的 UART 口, 并且注意查看跳线帽有没有接对, 如下

图所示。

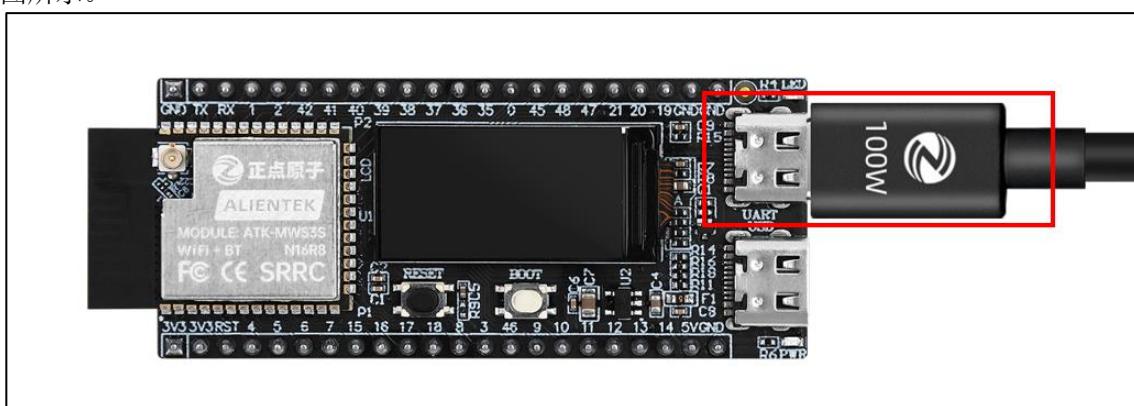


图 6.3.1 开发板连接电脑图

连接好开发板后，我们需要查看一下用到的是哪个端口，这时候可以通过 Arduino IDE 软件“工具→端口”进行查看，如下图所示。

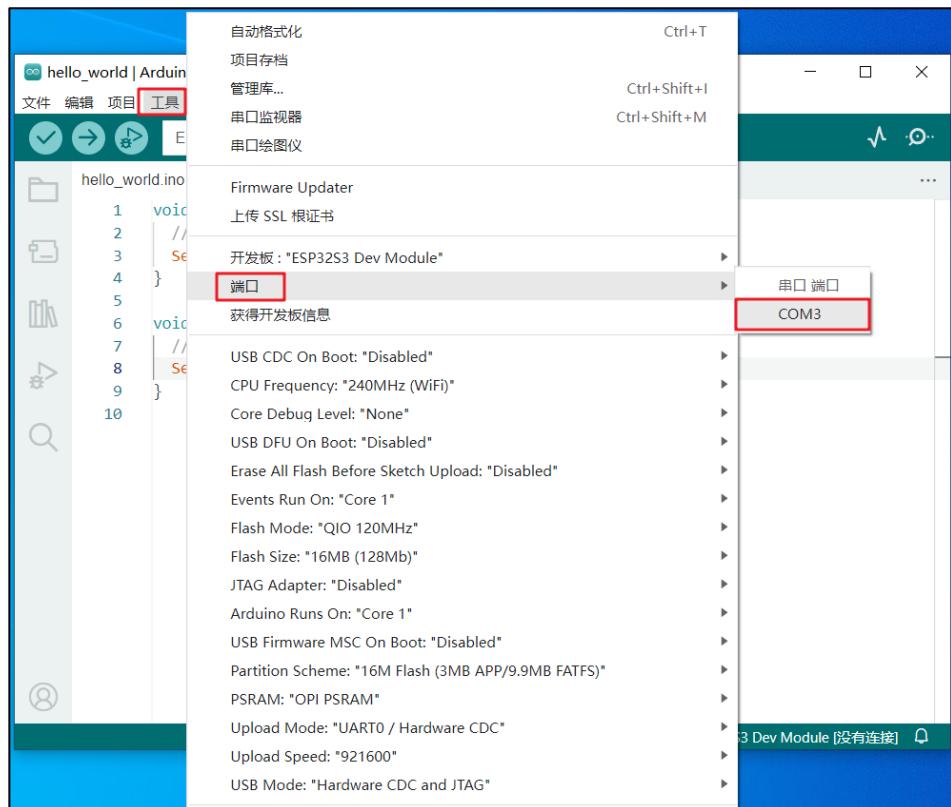


图 6.3.2 查看开发板的串口

可以看到开发板连接到的是电脑的 COM3，我们点击一下 COM3 即可。然后，可以看到软件的主界面右下角会显示“ESP32S3 Dev Module 在 COM3 上”，同时我们再简单写几句代码，大家现在看不懂没有关系，后面会一一讲解到，这时候主界面如下图所示。

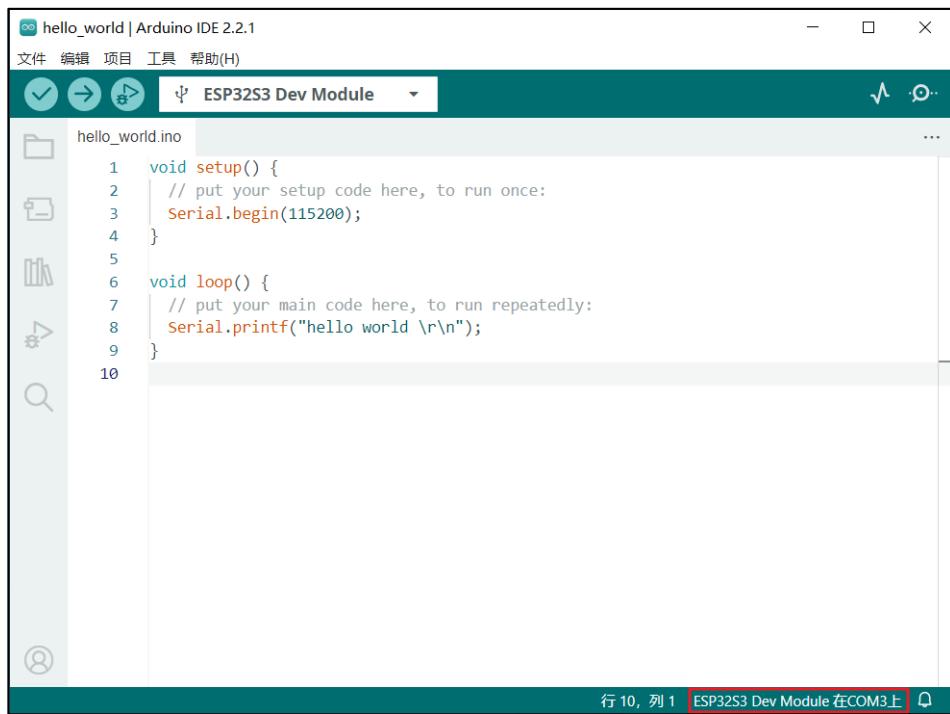


图 6.3.3 Arduino IDE2 主界面

接下来，我们就要按一下编译按钮，编译一下代码，通过信息显示窗口可以看到编译的过程，编译完成后，会显示项目占用的内存大小，如下图所示。

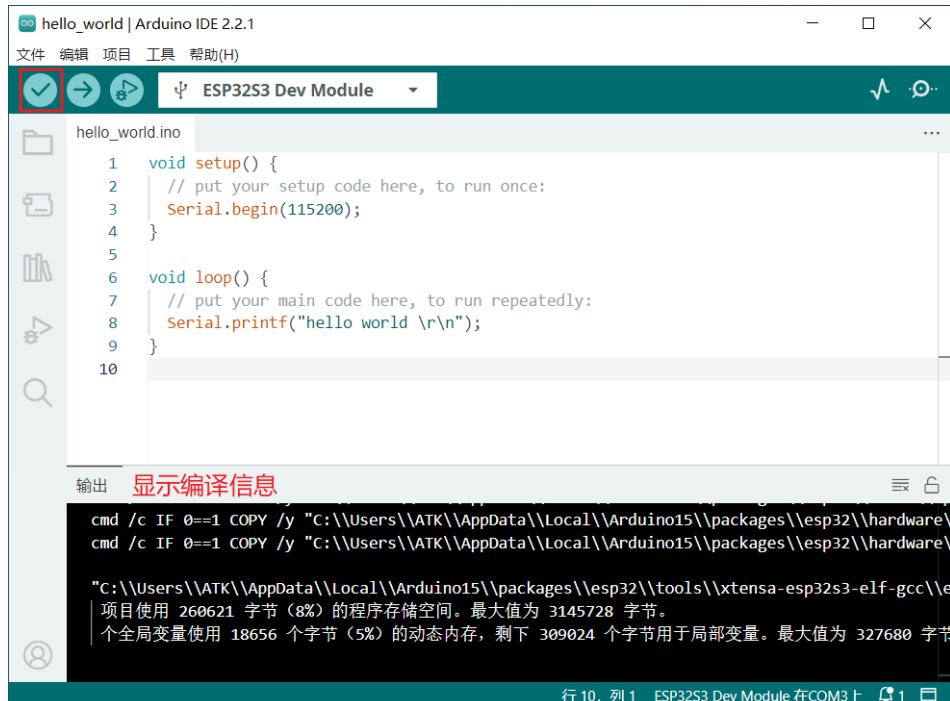


图 6.3.4 对代码进行编译

代码编译成功后，我们按下上传按钮，把程序烧录进开发板，这时候会从信息显示窗口看到具体的烧录过程，如下图所示。

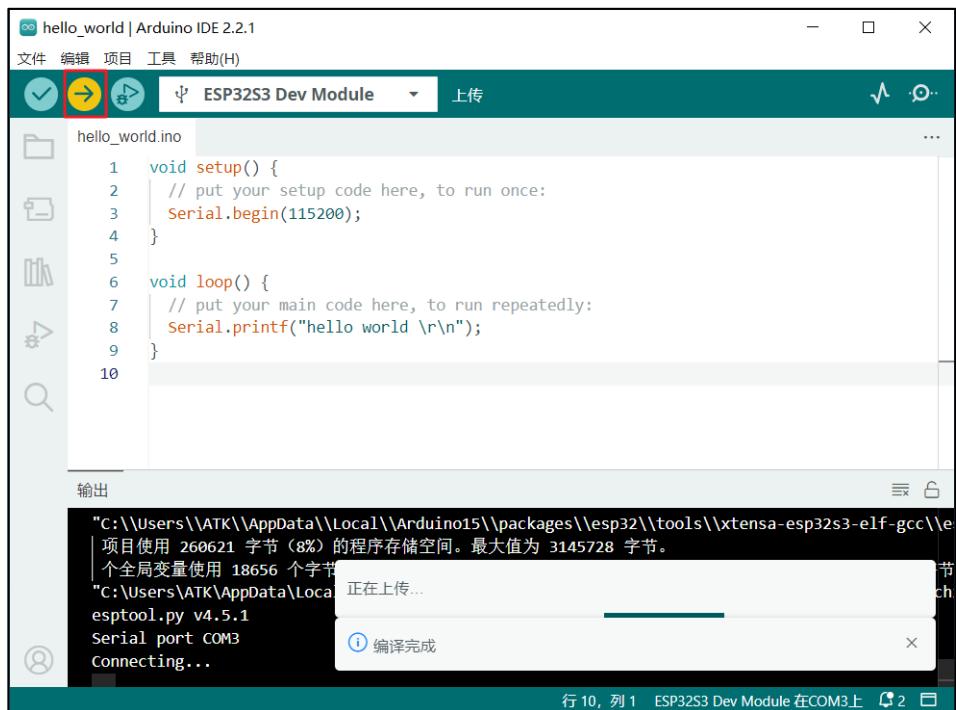


图 6.3.4 对代码进行上传

当我们程序下载成功后，如下图所示。

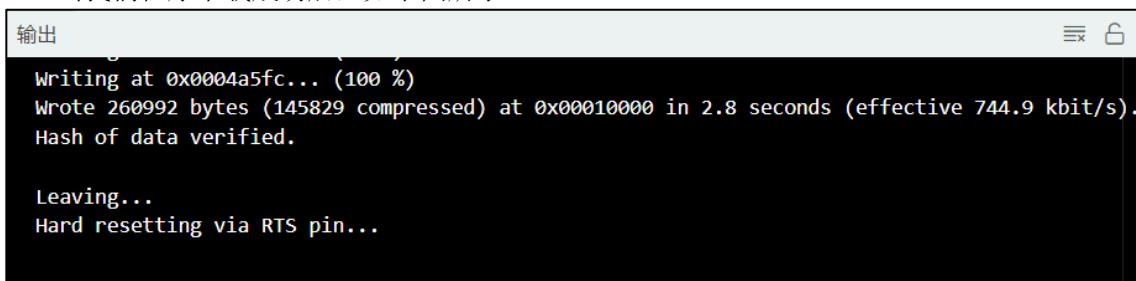


图 6.3.5 程序下载成功

最后，打开 Arduino IDE2 软件的串口助手，查看打印内容，如下图所示。



图 6.3.6 程序效果图

程序的功能是通过串口打印出“hello world”。

第二篇 入门篇

功夫不负有心人，相信学习至此你已经掌握了基础篇介绍的知识。我们希望通过前面的章节你已经掌握了 ESP32-S3 开发的工具和方法。本篇我们将和大家一起来学习 ESP32-S3 的一些基础外设，这些外设实际项目中经常会用到，希望大家认真学习和掌握，以便将来更好、更快的完成实际项目开发。

本篇分为以下几个章节：

- 1, LED 实验
- 2, KEY 实验
- 3, EXIT 实验
- 4, UART 实验
- 5, TIMER_IT 实验
- 6, LEDC_PWM 实验
- 7, SPI_LCD 实验
- 8, RTC 实验
- 9, INTERNAL_TEMPERATURE 实验
- 10, SPI_SDCARD 实验

第七章 LED 实验

本章将通过一个经典的点灯实验，带大家开启 ESP32-S3 Arduino 开发之旅。通过本章学习，我们将会学习到如何实现 ESP32-S3 的 IO 作为输出功能。

本章分为如下 4 个小节：

7.1 GPIO 介绍

7.2 硬件设计

7.3 软件设计

7.4 下载验证

7.1 GPIO 介绍

在前面的第四章中，有对 ESP32-S3 的 IO 进行说明。ESP32-S3 芯片具有 45 个物理 GPIO 管脚（GPIO0 ~ GPIO21 和 GPIO26 ~ GPIO48）。对于 ESP32-S3 模组引出的 IO 会变得更少些，只有 36 个。每个管脚都可用作一个通用 IO，或连接一个内部外设信号，可见 ESP32-S3 的管脚的强大。

“通用 IO”，官方会称之为“GPIO”，英文详称为 General-Purpose Input/output Port，通用输入/输出接口，当设置成输入模式可用于感知外界信号，设置成输出模式可用于控制外部设备。在开发中，我们经常会用到一些比较简单的外部模块：LED、按键等，使用它们其实很简单，只需要把它们跟芯片的 GPIO 连接，控制 GPIO 输出/读取高低电平即可。

当管脚作为一个内部外设信号去使用，这种情况可以称之为引脚复用，通常就是作为某个协议的引脚。

本章节就是介绍 ESP32-S3 管脚作为一个 GPIO 去点亮 LED。

7.1.1 ESP32-S3 的 GPIO 简介

ESP32-S3 模组可作为 GPIO 的管脚情况如下图所示。

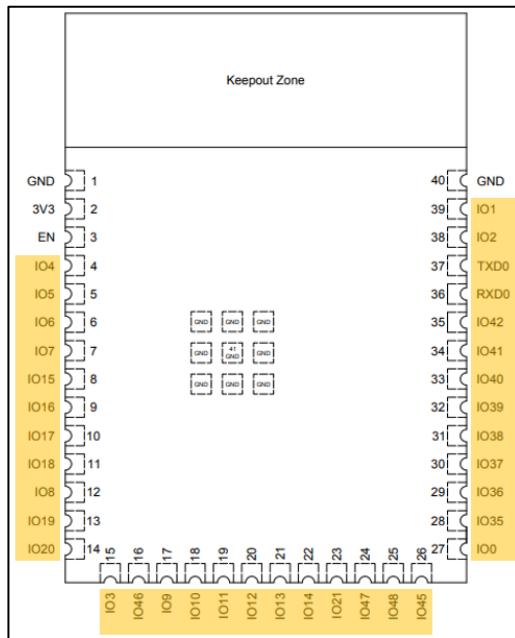


图 7.1.1.1 ESP32-S3 模组可做 GPIO 功能管脚图

GPIO 口输出来的信号是数字信号。数字信号是以 0, 1 表示的不连续信号，也就是以二进制形式表示的信号。在 Arduino 中数字信号用高低电平来表示，高电平为数字信号 1，低电平为数字信号 0。如下图所示：

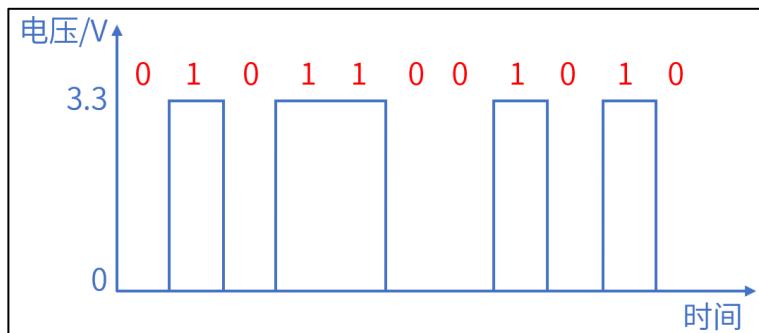


图 7.1.1.2 数字信号

图中 Arduino 输出的低电平为 0V，输出的高电平为当前 Arduino 的工作电压，对于 ESP32-S3 模组的工作电压为 3.3V，则其高电平为 3.3V。

7.1.2 GPIO 函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\esp32-hal-gpio.c

当我们调用 Arduino 库的函数时，实际上调用的是 arduino-esp32 库里的函数。换句话来说，每个芯片的 arduino 库都会有相同的函数，当工程设定好开发板参数，就会调用该芯片的 arduino 库下的函数接口，如下图所示。



图 7.1.2.1 不同芯片的 Arduino 库情况

Arduino 库有 pinMode 函数，同样的，其他芯片 Arduino 库也会有该函数的存在。唯一不同的就是该函数的实现会因芯片不同而不同。

接下来，我们就介绍一下本章节所用到的 GPIO 函数。

第一个函数：pinMode 函数，该函数功能是设置引脚的工作模式。在使用输入或输出功能前，需要先通过 pinMode 函数配置引脚的模式为输入模式或输出模式。

`void pinMode(uint8_t pin, uint8_t mode);`

参数 pin 为指定配置的引脚编号，比如要配置 IO1，那么 pin 设置为 1 即可。

参数 mode 为指定的配置模式，如下表所示。

模式名称	说明
OUTPUT	输出模式
INPUT	输入模式
INPUT_PULLUP	输入上拉模式
INPUT_PULLDOWN	输入下拉模式

表 7.1.2.1 引脚配置的模式

无返回值。

第二个函数：digitalWrite 函数，该函数功能是向指定引脚输出高低电平数字信号。

`void digitalWrite(uint8_t pin, uint8_t value);`

参数 pin 为指定输出的引脚编号。

参数 value 为要指定的输出电平，使用 HIGH 指定输出高电平，使用 LOW 指定输出低电平。HIGH 和 LOW 是 Arduino 核心库定义的关键字，分别代表 1 和 0，目的就是方便阅读、提高编程效率。当代码中使用了 HIGH 和 LOW，在编译时，会分别替换为 1 和 0。所以有时候，

我们直接用 1 和 0 而不用 HIGH 和 LOW。

无返回值。

第三个函数: `digitalRead` 函数, 该函数的功能是从指定引脚读取外部输入的数字信号。

```
int digitalRead(uint8_t pin);
```

参数 `pin` 为指定读取状态的引脚编号。

返回值: 当外部输入高电平时, 返回值为 1; 当外部输入低电平时, 返回值为 0。

7.2 硬件设计

1. 例程功能

LED 每过 500ms 一次交替闪烁, 实现类似跑马灯的效果。

2. 硬件资源

1) LED 灯

LED-IO1

3. 原理图

本章用到的硬件有 LED 灯。电路在开发板上已经连接好, 所以在硬件上不需要动任何东西, 直接下载代码就可以测试使用。其连接原理图如下图所示。

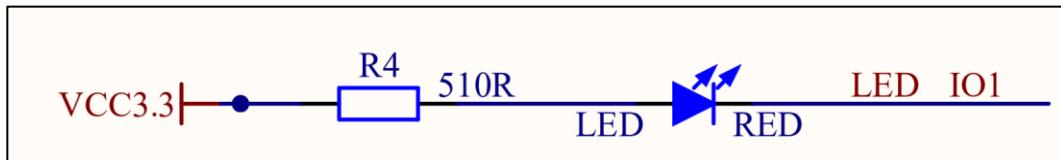


图 7.2.1 LED 与 ESP32-S3 模组连接原理图

从上图可知, 若 IO1 输出低电平时, 则 LED 亮起, 反之, 熄灭。

7.3 软件设计

7.3.1 程序流程图

程序流程图能帮助我们更好的理解一个工程的功能和实现的过程, 对学习和设计工程有很好的主导作用。

下面看看本实验的程序流程图:

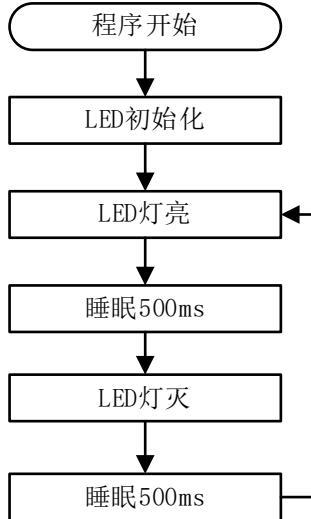


图 7.3.1.1 程序流程图

7.3.2 程序解析

1. led 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。LED 驱动源码包括两个文件：led.cpp 和 led.h。

下面我们先解析 led.h 的程序，我们把它分两部分功能进行讲解。

由硬件设计小节，我们知道 LED 灯在硬件上连接到 IO1，再结合 Arduino 库，我们做了下面的引脚定义。

```
/* 引脚定义 */
#define LED_PIN 1 /* 开发板上 LED 连接到 GPIO1 引脚 */
```

这样的好处是移植更加方便，函数命名更亲近实际的开发板。比如：当我们看到 LED_PIN 这个宏定义，我们就知道这是 LED 灯的端口号。大家后面学习时间长了就会慢慢熟悉这样的命名方式。

为了后续对 LED 灯进行便捷的操作，我们为 LED 灯操作函数做了下面的定义。

```
/* 宏函数定义 */
#define LED(x) digitalWrite(LED_PIN, x)
#define LED_TOGGLE() digitalWrite(LED_PIN, !digitalRead(LED_PIN))
```

LED 宏定义，控制 LED 亮灭。如果要设置 LED 输出低电平，那么调用宏定义 LED(0)即可，如果要设置 LED 输出高电平，调用宏定义 LED(1)即可。

LED_TOGGLE 宏定义，分别是控制 LED 的翻转。这里利用 digitalWrite 和 digitalRead 函数组合实现 IO 口输出电平取反操作。

下面我们再解析 led.cpp 的程序，这里只有一个函数 led_init，这是 LED 灯的初始化函数，其定义如下：

```
/**
 * @brief 初始化 LED 相关 IO 口
 * @param 无
 * @retval 无
 */
void led_init(void)
{
    pinMode(LED_PIN, OUTPUT); /* 设置 led 引脚为输出模式 */
    digitalWrite(LED_PIN, HIGH); /* 结合原理图设计，实物 LED 获得高电平会熄灭 */
}
```

LED 灯的引脚设置为输出模式。最后关闭 LED，防止没有操作的情况下，灯就亮了。

2. 01_led.ino 代码

在 01_led.ino 里面编写如下代码：

```
#include "led.h"

/**
 * @brief 当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param 无
 * @retval 无
 */
void setup()
{
    led_init(); /* LED 初始化 */
}

/**
 * @brief 循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param 无
 * @retval 无
 */
```

```
void loop()
{
    LED(0);
    /* 该宏函数在 led.h 有定义, 等同于 digitalWrite(LED, LOW), LED 引脚输出接低电平, 点亮 */
    delay(500); /* 延时 500 毫秒 */
    LED(1);
    /* 该宏函数在 led.h 有定义, 等同于 digitalWrite(LED, HIGH), LED 引脚输出接高电平, 熄灭 */
    delay(500); /* 延时 500 毫秒 */

    /* 以上 4 行代码实现的效果也可以用以下两行代码实现
    LED_TOGGLE(); // led 状态翻转
    delay(500); // 延时 500 毫秒
    */
}
```

在 `setup` 函数中, 调用 `led_init` 函数对 LED 进行初始化。接下来, 在 `loop` 函数中, 提供了两种方法实现 LED 灯的闪烁效果, 第一种: 先调用 `LED(0)` 宏函数实现 LED 点亮, 调用 `delay(500)` 实现 500ms 延时, 这时候 LED 就相当于亮了 500ms, 然后调用 `LED(1)` 宏函数实现 LED 熄灭, 调用 `delay(500)` 实现 500ms 延时, 这时候就相当于 LED 熄灭 500ms, 循环这个过程。第二种: 调用 `LED_TOGGLE()` 函数实现 LED 灯状态的变化并延时 500ms, 循环这个过程。

`delay()` 为毫秒延时函数, 在本程序用来控制开关 LED 的间隔时间。大家可以修改 `delay` 函数里面的参数, 观察运行效果的变化。

可能很多小伙伴会提出疑问, 这里怎么不是 `main` 函数, `main` 函数是有的, 但是其实已经隐藏起来了, 对于 Arduino 这种代码架构, 提供出 `setup` 函数和 `loop` 函数给用户使用。大家遵循 Arduino 规则去开发即可, 有兴趣的也可以自己去搜索。

7.4 下载验证

下载完之后, 可以看到 LED 灯每隔 500ms 闪烁一下。

第八章 KEY 实验

上一章，我们介绍了 ESP32-S3 的 IO 如何作为输出功能，实现 LED 闪烁。本章，我们将向大家介绍如何使用 ESP32-S3 的 IO 口作为输入。我们将利用板载的 boot 按键，来控制板载的 LED 灯亮灭。通过本章的学习，我们将了解到 ESP32-S3 的 IO 口作为输入口的使用方法。

本章分为如下 4 个小节：

- 8.1 GPIO 输入功能使用
- 8.2 硬件设计
- 8.3 软件设计
- 8.4 下载验证

8.1 GPIO 输入功能使用

8.1.1 GPIO 输入模式介绍

首先回顾一下：在上一章也提及到 `pinMode` 函数，要对数字 I/O 进行检测，首先把 I/O 设置为输入模式，然后使用数字 I/O 检测函数为 `digitalRead` 函数检测外部电平状态。当外部输入高电平时，返回值为 1；当外部输入低电平时，返回值为 0。

`pinMode` 函数设置 I/O 为输入，有三种选择：`INPUT`、`INPUT_PULLUP`、`INPUT_PULLDOWN`。这里简单设置的依据就跟这个按键电路相关。当我们存在上拉或者下拉按键电路的情况时，可以直接选择 `INPUT`；如果当前的按键电路需要内部上拉电阻，这时候选择 `INPUT_PULLUP`；如果当前的按键电路需要内部下拉电阻时，这时候选择 `INPUT_PULLDOWN`。

在这里，简单介绍一下上拉电阻电路、下拉电阻电路、内部上拉电路和内部下拉电路。

上拉电阻电路，如下图所示。

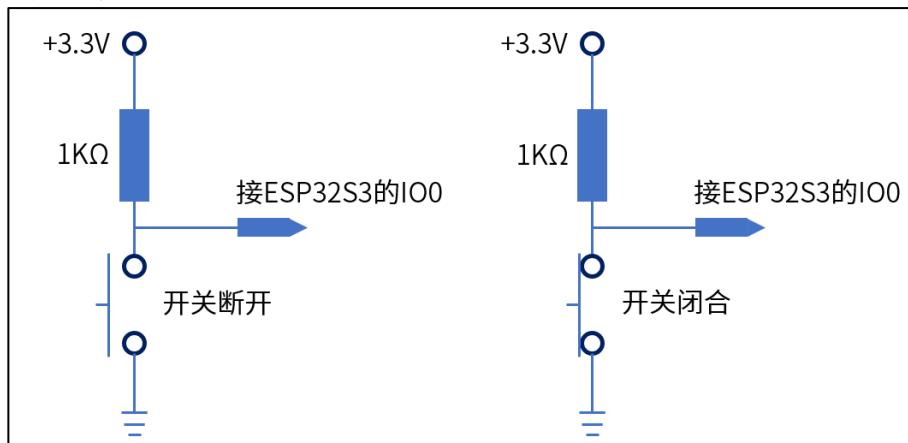
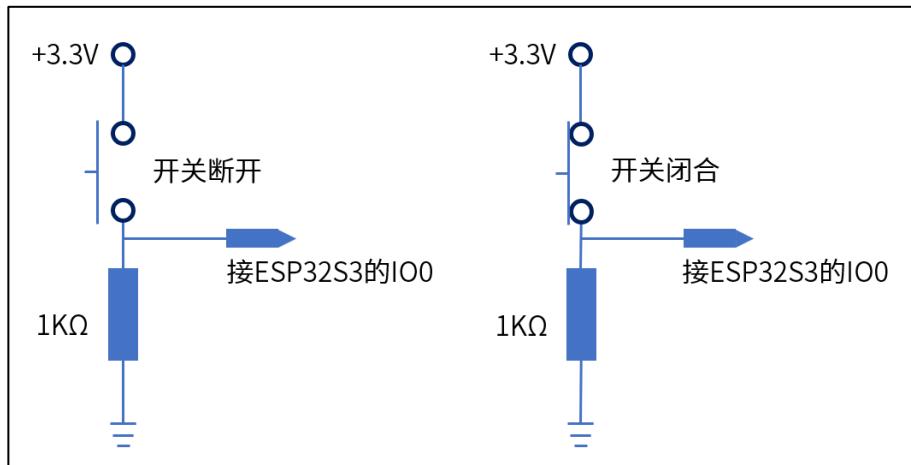


图 8.1.1.1 上拉电阻电路图

当按键开关断开时，即没有被按下时，ESP32S3 的 IO0 通过电阻和 3.3V 电源相连接，产生高电平，`digitalRead(0)` 函数的返回值为 1。当按键开关闭合时，即按下按键时，ESP32S3 的 IO0 的电压和地相连接，产生低电平，`digitalRead(0)` 函数的返回值为 0。电路中的 $1K\Omega$ 电阻成为上拉电阻。

下拉电阻电路，如下图所示。



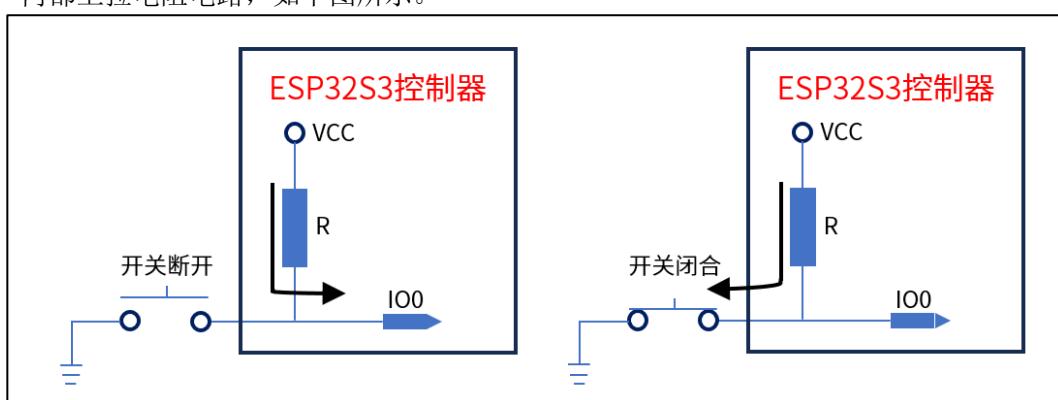
当按键开关断开时，即没有被按下时，ESP32S3 的 IO0 通过电阻和地相连接，产生低电平，`digitalRead(0)`函数的返回值为 0。当按键开关闭合时，即按下按键时，ESP32S3 的 IO0 的电压和电源 3.3V 相连接，产生高电平，`digitalRead(0)`函数的返回值为 1。电路中的 $1K\Omega$ 电阻成为下拉电阻。

当数字输入引脚的工作模式设置为 INPUT 时，读取按键值，一定要在电路中设置一个上拉电阻或者下拉电阻，电阻的阻值一般可以为 $1\sim10K\Omega$ 。采用上拉电阻时，当按键断开时 `digitalRead(0)`函数的返回值为 1。采用下拉电阻时，当按键断开时，`digitalRead(0)`函数的返回值为 0。

内部上拉电阻电路，除了上述的两种电路外，在 ESP32S3 控制器内部还集成了上拉电阻，通过在 `pinMode()` 函数中设置 `mode` 参数为 `INPUT_PULLUP` 来启动内部上拉电阻。

启动控制器内部的上拉电阻后，按键开关电路就可以省略外接电阻。

内部上拉电阻电路，如下图所示。



从上图可以看出，当开关断开时，`digitalRead(0)`函数的返回值为 1；当开关闭合时，`digitalRead(0)`函数的返回值为 0。

当采用内部上拉电阻电路时，按键的一端和数字引脚相连，另外一端和地相连。

内部下拉电阻电路，在 ESP32S3 控制器内部除了集成上拉电阻，还有下拉电阻，通过在 `pinMode()` 函数中设置 `mode` 参数为 `INPUT_PULLDOWN` 来启动内部下拉电阻。

启动控制器内部的下拉电阻后，按键开关电路就可以省略外接电阻。

内部下拉电阻电路，如下图所示。

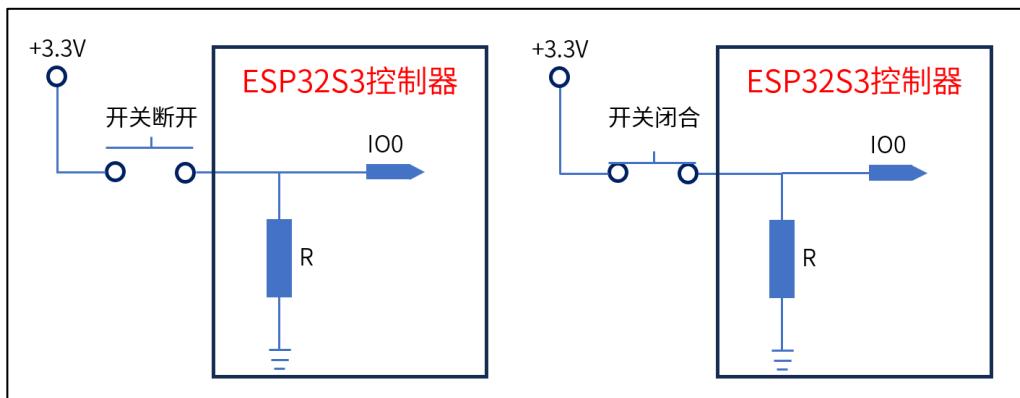


图 8.1.1.4 内部下拉电阻电路

从上图可以看出，当开关断开时，`digitalRead(0)`函数的返回值为 0；当开关闭合时，`digitalRead(0)`函数的返回值为 1。

当采用内部下拉电阻电路时，按键的一端和数字引脚相连，另外一端和 VCC 相连。

8.1.2 独立按键简介

几乎每个开发板都会板载有独立按键，因为按键用处很多。常态下，独立按键是断开的，按下的时候才闭合。每个独立按键会单独占用一个 IO 口，通过 IO 口的高低电平判断按键的状态。但是按键在闭合和断开的时候，都存在抖动现象，即按键在闭合时不会马上就稳定的连接，断开时也不会马上断开。这是机械触点，无法避免。独立按键抖动波形图如下：

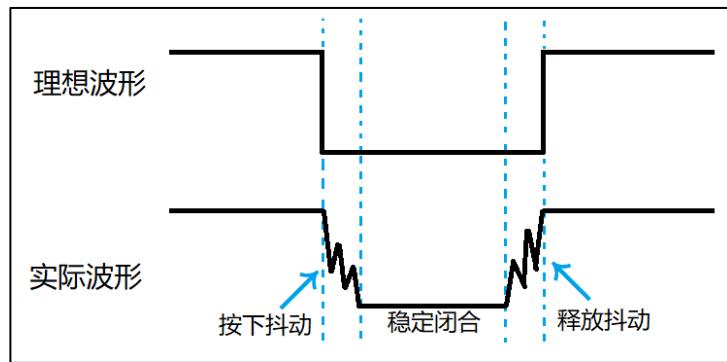


图 8.1.2.1 独立按键抖动波形图

图中的按下抖动和释放抖动的时间一般为 5~10ms，如果在抖动阶段采样，其不稳定状态可能出现一次按键动作被认为是多次按下的情况。为了避免抖动可能带来的误操作，我们要做的措施就是给按键消抖（即采样稳定闭合阶段）。消抖方法分为硬件消抖和软件消抖，我们常用软件的方法消抖。

软件消抖：方法很多，我们例程中使用最简单的延时消抖。检测到按键按下后，一般进行 10ms 延时，用于跳过抖动的时间段，如果消抖效果不好可以调整这个 10ms 延时，因为不同类型的按键抖动时间可能有偏差。待延时过后再检测按键状态，如果没有按下，那我们就判断这是抖动或者干扰造成的；如果还是按下，那么我们就认为这是按键真的按下了。对按键释放的判断同理。

硬件消抖：利用 RC 电路的电容充放电特性来对抖动产生的电压毛刺进行平滑出来，从而实现消抖，但是成本会更高一点，本着能省则省的原则，我们推荐使用软件消抖即可。

8.2 硬件设计

1. 例程功能

通过开发板上的 boot 独立按键实现 LED 的亮灭。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) 独立按键
BOOT-IO0

3. 原理图

独立按键硬件部分的原理图，如下图所示。

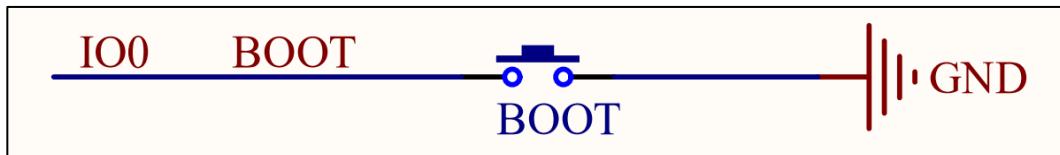


图 8.2.1 独立按键与 ESP32-S3 连接原理图

这里需要注意的是：BOOT 设计为采样到按键另一端的低电平为有效电平。

8.3 软件设计

8.3.1 程序流程图

下面看看本实验的程序流程图：

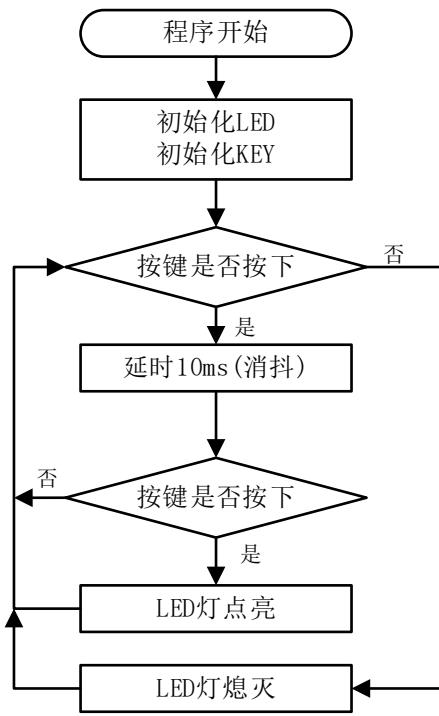


图 8.3.1.1 程序流程图

8.3.2 程序解析

1. key 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。KEY 驱动源码包括两个文件：key.cpp 和 key.h。

下面我们先解析 key.h 的程序，我们把它分两部分功能进行讲解。

由硬件设计小节，我们知道 KEY 按键在硬件上连接到 IO0，我们做了下面的引脚定义。

```
/* 引脚定义 */
#define KEY_PIN      0 /* 开发板上 KEY 连接到 GPIO0 引脚 */
为了后续对 KEY 按键进行便捷的操作，我们为 KEY 按键操作函数做了下面的定义。
/* 宏函数定义 */
#define KEY          digitalRead(KEY_PIN) /* 读取 KEY 引脚的状态 */

KEY 是读取对应按键状态的宏定义。用 digitalRead 函数实现，该函数返回值就是 IO 口的状态，0 或 1，代表的是低电平或高电平。
下面我们再解析 key.cpp 的程序，这里只有一个函数 key_init，这是 KEY 按键的初始化函数，其定义如下：
```

```
/**
* @brief    初始化 KEY 相关 IO 口
* @param    无
* @retval   无
*/
void key_init(void)
{
    /* 结合原理图设计，按键没有按下时，KEY 引脚检测到的是高电平 */
    pinMode(KEY_PIN, INPUT_PULLUP); /* 设置 key 引脚为上拉输入模式 */
}
```

KEY 按键的引脚设置为上拉输入模式。默认情况，读取到的是高电平。

2. 02_key.ino 代码

在 02_key.ino 里面编写如下代码：

```
#include "led.h"
#include "key.h"

/**
* @brief    当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
* @param    无
* @retval   无
*/
void setup()
{
    led_init(); /* LED 初始化 */
    key_init(); /* KEY 初始化 */
}

/**
* @brief    循环函数，通常放程序的主体或者需要不断刷新的语句
* @param    无
* @retval   无
*/
void loop()
{
    if (KEY == 0) /* 读取 KEY 状态，如果按下 KEY */
    {
        delay(10);

        if (KEY == 0)
        {
            LED(0); /* LED 引脚输出接低电平，点亮 */
        }
    }
    else          /* 读取 KEY 状态，如果 KEY 没有按下 */
    {
        LED(1); /* LED 引脚输出接高电平，熄灭 */
    }
}
```

在 setup 函数中，除了要调用 key_init 函数对 KEY 进行初始化，还要调用 led_init 对 LED

进行初始化。接下来，在 `loop` 函数中，当按键被按下时，会调用 `delay` 函数等待 10 毫秒实现消抖作用，然后再次确认按键状态，如果按键确实被按下，点亮 LED。如果按键没有被按下时，就会熄灭 LED。然后程序进入检测按键是否按下的循环中。

8.4 下载验证

下载完之后，通过 BOOT 按键来控制 LED 灯的亮灭状态。

第九章 EXTI 实验

在前面两章的学习中，我们掌握了 ESP32-S3 的 IO 口最基本的操作。本章我们将介绍如何把 ESP32-S3 的 IO 口作为外部中断输入来使用。在本章中，我们将以中断的方式，实现 boot 按键控制板载的 LED 灯状态翻转。

本章分为如下 4 个小节：

- 9.1 外部中断介绍
- 9.2 硬件设计
- 9.3 软件设计
- 9.4 下载验证

9.1 外部中断介绍

很多时候，我们程序采集一个传感器的数据，采集后就要进行分析判断，若符合某个条件就会做出处理。为了随时根据传感器的变化做出反应，所以程序需要一直重复这个过程。这种方式称为轮询，这种方式是最简单的。

但轮询有时候并不能很好完成一些实际场景的应用，比如我某个时刻按下按键，但这时候程序执行的是采集传感器数据的过程，这就意味着没有检测到按键按下的动作，此时该系统就成了无法正常响应的系统了。通过对该按键配置外部中断功能，这时候就能很好解决上述问题。

9.1.1 中断程序

外部中断是由外部设备发起请求的中断。每个中断对应一个中断程序，中断可以看作一段独立于主程序之外的程序，也称为中断回调函数。当中断被触发时，控制器会暂停当前正在运行的主程序，而跳转去运行中断程序。当中断程序运行完毕，则返回到先前主程序暂停的位置，继续运行主程序，如此便可达到实时响应处理事件的效果。中断程序运行示意图如下图所示。

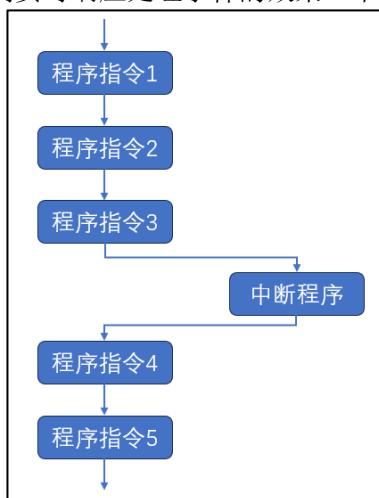


图 9.1.1.1 中断程序执行示意图

9.1.2 ESP32-S3 的中断介绍

便于大家了解 ESP32-S3 芯片的中断知识，这里简单介绍一下。

ESP32-S3 有 99 个外部中断源，但是 CPU0 或 CPU1 只能够处理 32 个中断。ESP32-S3 将外部中断映射到 CPU0 或 CPU1 中断就需要用到中断矩阵。中断映射的过程如下图所示。

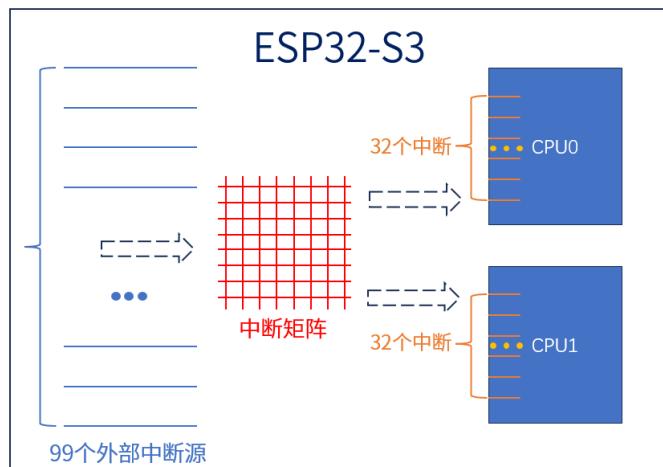


图 9.1.2.1 ESP32-S3 芯片中断源到 CPU 中断过程图

ESP32-S3 中断矩阵会将任一外部中断源单独分配到双核 CPU 的任一外部中断上，以便在外设中断信号产生后，及时通知 CPU0 或 CPU1 进行处理。

每个 CPU 都有 32 个中断号(0~31)，其中包括 26 个外部中断，6 个内部中断。

外部中断为外部中断源引发的中断，包括下面三种类型：

- 1) 电平触发类型中断：高电平触发，要求保持中断的电平状态直到 CPU 响应
- 2) 边沿触发类型中断：上升沿触发，此中断一旦产生，CPU 即可响应
- 3) NMI 中断：不可屏蔽中断，产生该中断时，表示系统发生了致命错误

内部中断为 CPU 内部自己产生的中断，包括以下三种类型：

- 1) 定时器中断：由内部定时器触发，可用于产生周期性的中断
- 2) 软件中断：软件写特殊寄存器时将触发此中断
- 3) 解析中断：用于性能监测和分析

ESP32-S3 的外部中断是很强大的，每个引脚都可设置成外部中断触发引脚。但在大部分的 Arduino 控制器上，并非所有引脚都有中断功能。只有少数带外部中断功能的引脚上，Arduino 控制器才能捕获到该中断信号并做出反应。在这过程中，难免需要通过中断引脚找中断的编号。

9.1.3 中断触发模式

ESP32-S3 的中断触发模式有 5 种，即前面所提及的电平触发以及边沿触发，如下表所示。

中断触发模式	说明	示意图
RISING	上升沿触发，即由低电平变高电平时触发	
FALLING	下降沿触发，即由高电平变低电平时触发	
CHANGE	电平变化触发，即由高电平变低电平或由低电平变高电平时触发	
ONLOW	低电平触发，即当前为低电平时触发	
ONHIGH	高电平触发，即当前为高电平时触发	

表 9.1.3.1 ESP32-S3 中断模式说明表

9.1.4 中断触发函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\esp32-hal-gpio.c

Arduino-esp32 库提供了两个中断函数，一个用于对中断引脚进行初始化设置，另一个是关闭外部中断。

attachInterrupt 函数，该函数功能是指定中断引脚，并对中断引脚进行初始化设置。

```
void attachInterrupt(uint8_t pin, voidFuncPtr handler, int mode);
```

参数 pin 为要设置中断触发输入的引脚，ESP32-S3 所有引脚均可以配置为外部中断引脚。

参数 handler 为中断回调函数，当引脚中断触发时，会终止当前运行的程序，转而执行该程序。

参数 mode 为 5 种中断触发模式，如 9.1.3.1 表所示。

注意：中断回调函数不能有参数，且没有返回值。

假如不需要监测某个引脚的信号变化，可通过 detachInterrupt 函数关闭外部中断。

```
void detachInterrupt(uint8_t pin);
```

其中参数 pin 为已经设置中断触发输入的引脚。在本例程中没有用到该函数。

另外，很多时候，我们会见到 attachInterrupt 函数的第一参数会使用 digitalPinToInterrupt(pin) 函数，这里简单解释一下为什么？

其实前面也有所提及，在一些 Arduino 开发板中比如 Arduino Uno、Leonardo，只有 2 和 3 引脚有外部中断功能，而中断编号对应为 0 和 1。在 Arduino Uno 开发板中，attachInterrupt 函数第一个参数为中断编号，第二个参数为中断回调函数，第三个参数为触发模式，所以为了避免硬件引脚和中断编号，直接通过 digitalPinToInterrupt 函数解决。很多时候，attachInterrupt 函数的使用也采用以下方式。

```
attachInterrupt(digitalPinToInterrupt(pin), handler, mode);
```

ESP32-S3 的中断引脚跟中断编号一致，也可不必用 digitalPinToInterrupt 函数，但是你在程序中也没有问题。

9.2 硬件设计

1. 例程功能

通过外部中断的方式让开发板上的 BOOT 独立按键控制 LED 灯翻转。

2. 硬件资源

1) LED 灯

LED-IO1

2) 独立按键

BOOT-IO0

3. 原理图

本实验的原理图跟第八章 KEY 实验的原理图是一致的，所以不再列出。

9.3 软件设计

9.3.1 程序流程图

下面看看本实验的程序流程图：

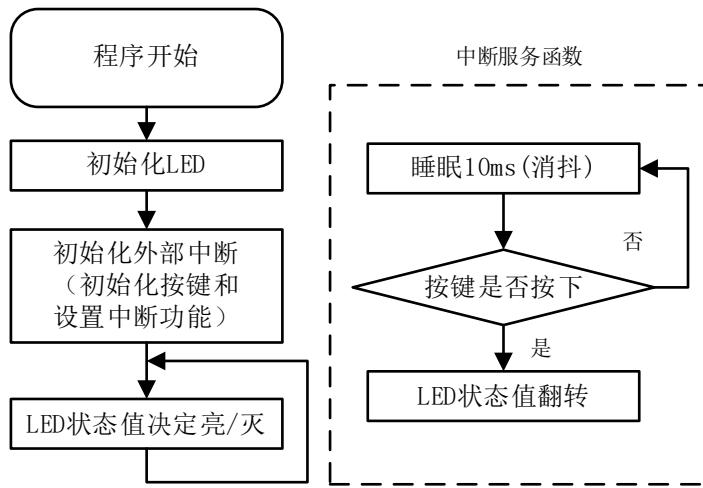


图 9.3.1.1 程序流程图

9.3.2 程序解析

1. exti 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。EXTI 驱动源码包括两个文件：exti.cpp 和 exti.h。

下面我们先解析 exti.h 的程序。由硬件设计小节，我们知道 KEY 按键在硬件上连接到 IO0，为了与按键实验进行区分，我们做了下面的引脚定义。

```
/* 引脚定义 */
#define KEY_INT_PIN 0 /* 外部中断引脚 IO0 */
```

下面我们再解析 exti.cpp 的程序，这里有两个函数 exti_init 和 key_isr，其定义如下：

```
uint8_t led_state = 0; /* 决定灯亮灭状态变量 */

/**
 * @brief 初始化外部中断相关 IO 口
 * @param 无
 * @retval 无
 */
void exti_init(void)
{
    key_init(); /* KEY 初始化 */
    attachInterrupt(digitalPinToInterrupt(KEY_INT_PIN), key_isr, FALLING);
    /* 设置 KEY 引脚为中断引脚,下降沿触发 */
}

/**
 * @brief KEY 外部中断回调函数
 * @param 无
 * @retval 无
 */
void key_isr(void)
{
    delay(10);
    if (KEY == 0)
    {
        led_state = ! led_state; /* 两种情况：从 0 变 1，从 1 变为 0 */
    }
}
```

exti_init 函数是初始化外部中断引脚，首先调用 key_init 设置 KEY 引脚为上拉输入模式，然后调用 attachInterrupt 函数设置中断回调函数和设置中断引脚为下降沿触发。

`key_isr` 函数就是中断引脚的中断回调函数。当按键被按下时，出现了下降沿即高电平变为低电平时，这时候就会跳到该函数去执行。函数内部很简单，首先调用 `delay` 函数进行按键消抖，然后判断按键是否真的被按下。当按键是按下情况，就对全局变量 `led_state` 进行取反操作，即 `led_state` 就有两种情况 0 和 1，最终在 `loop` 函数中作为 `LED(x)` 宏函数的参数 `x` 决定了 LED 灯的状态。

2. 03_exti.ino 代码

在 03_exti.ino 里面编写如下代码：

```
#include "led.h"
#include "exti.h"

/**
 * @brief  当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    led_init();      /* LED 初始化 */
    exti_init();    /* 外部中断引脚初始化 */
}

/**
 * @brief  循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param  无
 * @retval 无
 */
void loop()
{
    LED(led_state); /* 灯的亮灭由 led_state 值决定，led_state 变化在 key_isr 函数中实现 */
}
```

在 `setup` 函数中，除了要调用 `led_init` 函数对 LED 灯进行初始化，还要调用 `exti_init` 函数对按键 KEY 进行初始化以及配置 KEY 所在引脚的下降沿触发中断功能。接下来，在 `loop` 函数中，LED 的状态通过全局变量 `led_state` 决定，若 `led_state` 为 0，即运行 `LED(0)` 代码，这时候 LED 灯就会亮起；若 `led_state` 为 1，即运行 `LED(1)` 代码，这时候 LED 灯就会熄灭。

当按键被按下时，这时候出现了下降沿，就会触发中断，进入到中断回调函数中。在该函数中，会对全局变量 `led_state` 进行取反操作，执行完又回到 `loop` 函数暂停的地方继续往下执行。最终，实现的效果就是：通过按下按键，LED 灯的状态会进行翻转。

9.4 下载验证

下载完之后，通过 BOOT 按键来控制 LED 灯的亮灭状态。

第十章 UART 实验

本章,我们将学习 ESP32-S3 的串口,教大家如何使用 ESP32-S3 的串口来发送和接收数据。本章将实现如下功能: ESP32-S3 通过串口和上位机的对话, ESP32-S3 在收到上位机发过来的字符串后,原原本本的返回给上位机。

本章分为如下 4 个小节:

- 10.1 串口简介
- 10.2 硬件设计
- 10.3 软件设计
- 10.4 下载验证

10.1 串口介绍

由于大部分 Arduino 开发板都没有调试功能,所以在开发中,最常用的就是串口打印信息到串口助手去调试程序。而串口打印,正规一点来讲就是串口通信,开发板通过串口外设把数据通过串口线传输到电脑的串口助手上位机。

在开发板上存在各种各样的通信,所以在这里花一点篇幅来讲解一下数据通信方面知识,方便大家对通信有点概念,知其然知其所以然。

10.1.1 数据通信的基本概念

这里将会简单讲解一下“串行/并行通信”、“单工/半双工/全双工通信”、“同步/异步通信”、“波特率”等知识。

串行/并行通信

平常经常会听到串行通信和并行通信,这两者其实很好去区分。而串口属于串行通信,特点就是数据是逐位按顺序依次传输,如下图所示。

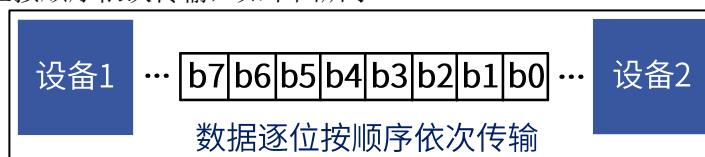


图 10.1.1.1 串行通信

并行通信,特点是数据各位通过多条线同时传输,如下图所示。



图 10.1.1.2 并行通信

简单理解,这两者就是单车道和多车道的概念,同一个时刻,单车道只能通行一辆车,好比串行通信只能传递 1 位数据;多车道可以通行多辆车好比并行通信传递多位数据。

这两者的对比,我们这里也做了一点归纳,如下图所示。

特点	传输速率	抗干扰能力	通信距离	IO 资源占用	成本
串行通信	较低	较强	较长	较少	较低
并行通信	较高	较弱	较短	较多	较高

表 10.1.1.1 串行通信和并行通信的对比表

单工/半双工/全双工通信

按数据传输方向分类：单工通信、半双工通信和全双工通信，他们的通信图如下图所示。

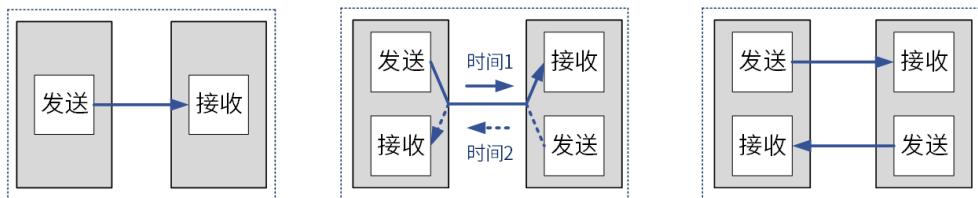


图 10.1.1.3 3 种数据传输方向图

单工通信：数据只能沿一个方向传输

半双工通信：数据可以沿两个方向传输，但需要分时进行

全双工通信：数据可以同时进行双向传输

串行通信属于以上的全双工通信。

同步/异步通信

按数据同步方式分类：同步通信、异步通信，他们的区别如下图所示。

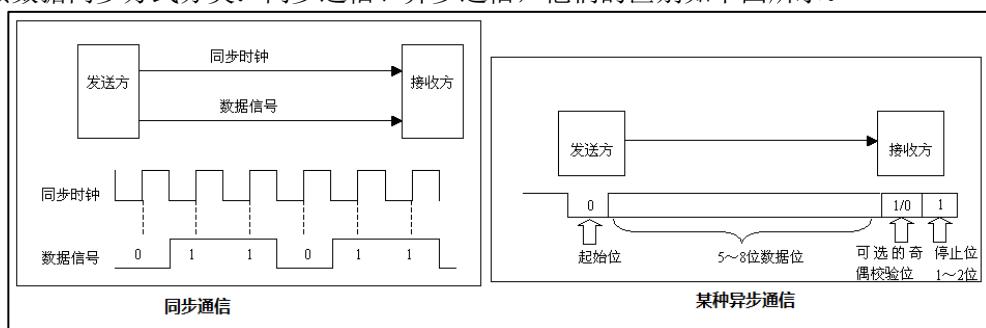


图 10.1.1.4 同步和异步通信区别图

同步通信：共用同一时钟信号

异步通信：没有时钟信号，通过在数据信号中加入起始位和停止位等一些同步信号

串行通信属于以上的异步通信。

波特率

双方进行通信，假如不存在时钟信号去同步数据的传输的过程，就需要双方设置一样的数据传输速度，也就是波特率。

在二进制系统中，波特率的含义就是每秒传输多少位，当我们设置波特率为 115200，这里的含义就是 1 秒钟传送 9600 位(bit)数据。串口通信默认的传输方式除了数据本身 8 位外，还需要加上起始位和停止位，所以传输 1 字节数据就需要 10 位。那 1 秒钟传输的数据量就是 960 字节，即一秒钟传输 9600 位 / 一字节需要传输 10 位 = 960 字节。

电脑的串口助手或者 Arduino IDE 自带的串口监视器波特率需要跟程序设置的一样，才可以正确接收数据。

10.1.2 UART 介绍

UART，Universal asynchronous receiver transmitter，是通用异步接收器/发送器，通常集成在主控器中。UART 控制器设有一定容量的数据缓冲区，用于存储通信时的数据。

通常，UART 使用两条信号线传输数据，分别为数据发送端 TX 和数据接收端 RX。通信时，一端的数据发送端(TX)连接到另一端的数据接收端(RX)，连接形式如下图所示：

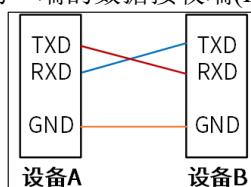


图 10.1.2.1 UART 通信连接形式

在 ESP32-S3 中，是有 3 个 UART 控制器，即 UART0、UART1 和 UART2。3 个 UART 端

口对应的引脚如下表所示。

UART 端口	RX	TX
UART0	GPIO44	GPIO43
UART1	GPIO18	GPIO17
UART2	任意 GPIO 管脚	任意 GPIO 管脚

表 10.1.2.1 UART 端口引脚

上表带有具体 IO 口是默认使用 IO，但是 ESP32-S3 有 IO MUX，所以是可以选择任意 GPIO 管脚作为 UART 的引脚。使用 Arduino，调用串口初始化函数时，可以指定发送引脚和接收引脚。

ESP32-S3 开发板上有一个 TYPEC 接口是通过一个 USB 转 UART 接口芯片连接 UART0，使用该串口上传程序或与计算机交互。注意：要识别串口，就得安装串口驱动。

10.1.3 串口相关函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\HardwareSerial.cpp

在 HardwareSerial.cpp 中已经定义好了三个 UART 对象 Serial、Serial1 和 Serial2，对应的就是 UART0、UART1 和 UART2，直接使用它们即可。

串口初始化函数介绍

在 Arduino 中，是要使用 begin 函数初始化串口功能，即

```
void HardwareSerial::begin(unsigned long baud, uint32_t config, int8_t rxPin, int8_t txPin, bool invert, unsigned long timeout_ms, uint8_t rxfifo_full_thrd);
```

由于参数比较多，所以这里以表格形式说明参数，如下表所示。

参数	参数说明
baud	串口波特率，此处写 0 会进入自动检测波特率程序
config	串口参数：设置数据位、奇偶校验位和停止位，默认为 SERIAL_8N1 即数据位 8 位，不使用奇偶检验，停止位 1 位
rxPin	接收引脚的编号
txPin	发送引脚的编号
invert	翻转逻辑电平，串口默认高电平为 1，低电平为 0
timeout_ms	自动检测波特率超时时间，如果超过该时间还没有获得波特率就不会使能串口（若设置 baud 为 0，该参数有意义）
rxfifo_full_thrd	接收缓冲区的阈值，当接收器接收到的比阈值多的数据时，产生中断

表 10.1.3.1 begin 函数参数说明表

通常情况下，通过如下语句便可以初始化串口 0，默认使用 IO43 作为串口 0 的发送引脚，使用 IO44 作为串口 0 的接收引脚，8 位数据位，无奇偶检验位，1 位停止位。

```
Serial.begin(115200);
```

当然，我们也可以通过 Serial1.begin 或 Serial2.begin 接口去设置串口 2 和串口 3，带上参数即可自由设置发送和接收引脚。

串口发送函数介绍

串口初始化完成后，便可以 Serial.print、Serial.println 和 Serial.printf 函数向串口助手发送数据。

Serial.print 函数用法：

```
Serial.print(val);
```

其中参数 val 是要输出的数据，各种类型数据都可以。

Serial.println 函数用法：

```
Serial.println(val);
```

Serial.println(val)函数也是使用串口输出数据，不同于 Serial.print 函数，该函数输出完指定数据后，再输出回车换行符。

下面测试一下这两个函数，在 UART 例程的 loop 函数中加入这四句代码。

```
Serial.print(1);
Serial.println("hello world");
```

```
Serial.print(2);
Serial.println("hello world");
```

通过点击 Arduino IDE 右上角的串口监视器图标打开串口监视器，可以看到如下效果。



图 10.1.3.1 print 和 println 函数区别

通过上图可以清楚看到 print 是直接输出，而 println 是输出数据后，还要进行换行。

需要注意，串口监视器窗口有一个选择波特率的设置，要选择与程序一样的设置，才能正常发送/接收数据。

Serial.printf 函数用法：

```
Serial.printf(char * format, ...);
```

该函数功能是输出一个字符串，或者按指定格式和数据类型输出若干变量的值，函数返回值为输出字符的个数。

在 Serial.printf() 函数使用中，会涉及到比较多的格式字符 “%d、%c、%f”，“\n、\r” 等为转义字符，这里整理了一个表格来说明这些常用的格式字符和转义字符，如下表所示。

格式字符/转义字符	说明
%o	八进制整数输出
%d	十进制整数输出
%x	十六进制整数输出
%f	浮点输出，默认小数点 6 位
%c	单个字符输出
%s	字符串输出
\n	换行
\r	回车
\t	Tab 符

表 10.1.3.2 常用的格式字符和转义字符

串口接收函数介绍

除了输出，串口同样可以接收由串口助手发出的数据。接收串口数据需要使用 Serial.read() 函数，函数用法：

```
Serial.read();
```

调用该函数，每次都会返回 1 字节数据，该返回值便是当前串口读取到的数据。

下面测试一下这个函数，在 UART 例程的 loop 函数中加入这两句代码。

```
char c = Serial.read();
Serial.print(c);
```

程序执行情况如下图所示：



图 10.1.3.2 串口监视器显示接收到的数据

在图 10.1.2.1 中圈红框的发送数据框写入“hello_world”回车进行发送，然后在串口监视器界面是可以见到“hello_world”字样，除此之外，还有一些乱码。这些乱码数据是因为没有可读数据造成的。当我们用 `Serial.print(Serial.read())` 程序，若没有可读数据，返回的是 `int` 型数据-1，对应到 `char` 型数据就是乱码。

在使用串口时，Arduino 会在 SRAM 中开辟一段大小为 64 字节的空间，窗口接收到的数据都会被暂时存放在该空间中，称这个存储空间为缓冲区。当调用 `Serial.read()` 函数时，Arduino 便会从缓冲区中取出 1 字节的数据。

通常在使用串口读取数据时，需要搭配使用 `Serial.available()` 函数，用法是：

```
Serial.available();
```

`Serial.available` 函数的返回值为当前缓冲区中接收到的数据字节数。通常该函数会搭配 `if` 或者 `while` 语句来使用，先检测缓冲区中是否有可读数据，如果有数据，再读取；如果没有数据，跳过读取或等待读取，如下所示。

```

if (Serial.available() > 0)
    或
while (Serial.available() > 0)
    下面改进一下上面的代码。
while (Serial.available() > 0)
{
    char c = Serial.read();
    Serial.print(c);
}

```

程序执行情况如下图所示：



图 10.1.3.3 串口监视器显示接收到的数据

可以看到 `Serial.available()` 函数和 `Serial.read()` 函数搭配使用下，不会出现乱码现象。

10.2 硬件设计

1. 例程功能

1. 回显串口接收到的数据
2. 每间隔一定时间，串口发送一段提示信息

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44

3. 原理图

USB 转串口硬件部分的原理图，如下图所示。

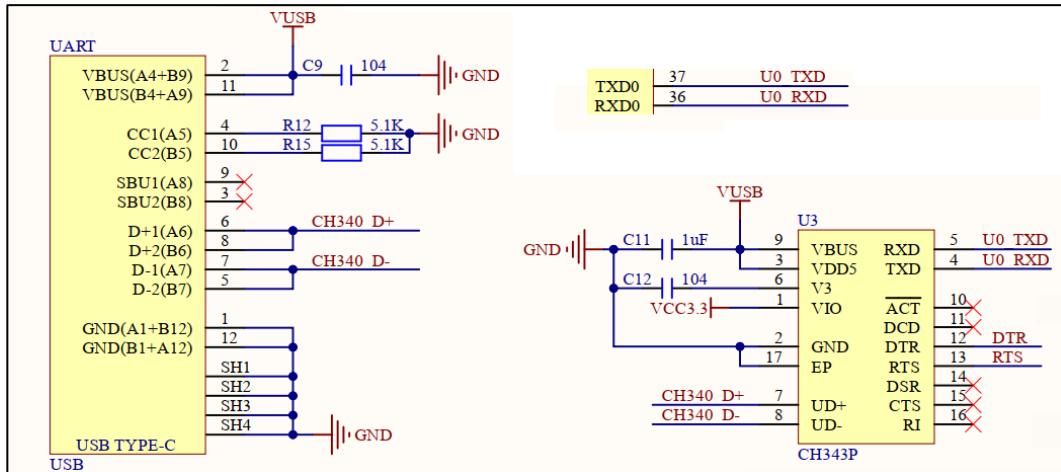


图 10.2.1 USB 转串口原理图

10.3 软件设计

10.3.1 程序流程图

下面看看本实验的程序流程图：

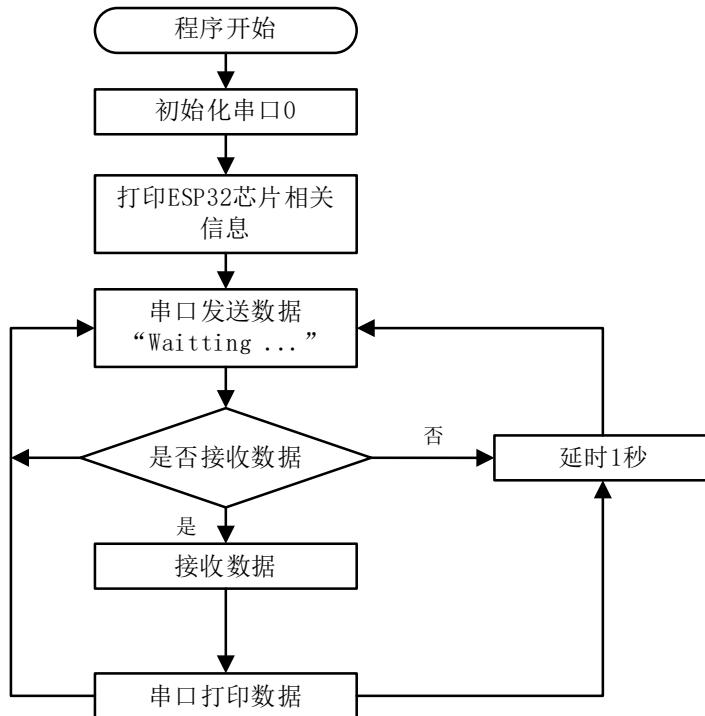


图 10.3.1.1 程序流程图

10.3.2 程序解析

1. uart 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。UART 驱动源码包括两个文件：uart.cpp 和 uart.h。

下面我们先解析 uart.h 的程序。在 uart 头文件中，我们做了下面的引脚定义。

```
/* 引脚定义 */
/* 串口 0 默认已经使用了固定 IO(GPIO43 为 U0TXD, GPIO44 为 U0RXD)
 * 以下两个宏为串口 1 或串口 2 使用到的 IO 口(例程未使用)
 */
#define TXD_PIN      19
#define RXD_PIN      20
```

下面我们再解析 uart.cpp 的程序，这里只有一个函数 uart_init，其定义如下：

```
/***
 * @brief    初始化 UART
 * @param    uartx: 串口 x
 * @param    baud: 波特率
 * @retval   无
 */
void uart_init(uint8_t uartx, uint32_t baud)
{
    if (uartx == 0)
    {
        Serial.begin(baud);                                /* 串口 0 初始化 */
    }
    else if (uartx == 1)
    {
        Serial1.begin(baud, SERIAL_8N1, RXD_PIN, TXD_PIN); /* 串口 1 初始化 */
    }
    else if (uartx == 2)
    {
        Serial2.begin(baud, SERIAL_8N1, RXD_PIN, TXD_PIN); /* 串口 2 初始化 */
    }
}
```

为了方便大家使用不同的串口，所以特定封装了一个 uart_init 函数。该函数是根据传参去初始化对应串口，而在 uart.h 中的 TXD_PIN 和 RXD_PIN 是专门给串口 1 或者串口 2 指定引脚的。初始化串口就是用到 Serial 库的 begin 函数。由于 Serial 库属于系统的核心库，所以使用时不需要导入库的头文件。

需要注意，串口 0 在硬件上已经固定好了，而使用其他串口实现与例程同样的效果，得需要串口电路即 10.2.3 原理图中的 CH343P 芯片或其他串口芯片，这里也可以选择购买正点原子的 USB 转串口模块 MO340P。

2. 04_uart.ino 代码

在 04_uart.ino 里面编写如下代码：

```
#include "uart.h"

uint32_t chip_id = 0;      /* 芯片 ID */
/***
 * @brief    当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param    无
 * @retval   无
 */
void setup()
{
    uart_init(0, 115200);    /* 串口 0 初始化 */

    for(int i = 0; i < 17; i = i + 8)
```

```

    { /* 获取 ESP32 芯片 MAC 地址 (6Byte), 该地址也可作为芯片 ID */
    chip_id |= ((ESP.getEfuseMac() >> (40 - i)) & 0xff) << i;
}

Serial.printf("ESP32 Chip model = %s Rev %d \n", ESP.getChipModel(),
ESP.getChipRevision()); /* 芯片类型和芯片版本号 */
Serial.printf("This chip has %d cores \n", ESP.getChipCores()); /* 内核数 */
Serial.print("Chip ID: "); Serial.println(chip_id); /* 芯片 ID */
Serial.printf("CpuFreqMHz: %d MHz\n", ESP.getCpuFreqMHz()); /* 主频 */
Serial.printf("SdkVersion: %s \n", ESP.getSdkVersion()); /* SDK 版本 */
}

/**
 * @brief 循环函数, 通常放程序的主体或者需要不断刷新的语句
 * @param 无
 * @retval 无
 */
void loop()
{
    Serial.println("Waiting for Serial Data \n"); /* 等待串口助手发过来的串口数据 */
    while (Serial.available() > 0) /* 当串口 0 接收到数据 */
    {
        Serial.println("Serial Data Available..."); /* 通过串口监视器通知用户 */
        String serial_data; /* 存放接收到的串口数据 */

        int c = Serial.read(); /* 读取一字节串口数据 */
        while (c >= 0)
        {
            serial_data += (char)c; /* 存放到 serial_data 变量中 */
            c = Serial.read(); /* 继续读取一字节串口数据 */
        }
        /* 将接收到的信息使用 readString() 存储于 serial_data 变量(跟前面 4 行代码具有同样效果) */
        // serial_data = Serial.readString();
        Serial.print("Received Serial Data: "); /* 串口监视器输出 serial_data 内容 */
        Serial.println(serial_data); /* 查看 serial_data 变量的信息 */
    }

    delay(1000);
}

```

在 setup 函数中, 调用 uart_init 函数对串口 0 进行初始化。

接下来, 在 loop 函数中, 就通过 Serial.available 函数去查询是否接收到串口数据, 假如有接收到, 那么就通过 Serial.read 函数去读取串口数据, 把这些串口数据保存到 serial_data 变量中, 最终打印 serial_data 变量就为串口接收到的数据。程序中提供了两种方式去读取完整串口数据, 大家可以自行去选择使用。

10.4 下载验证

下载完之后, 打开串口监视器, 看到不断打印出“Waiting for Serial Data”信息, 当我们在消息输入框写入“hello esp32”, 回车发送, “hello esp32”将会回显出来, 可以看到如下效果。

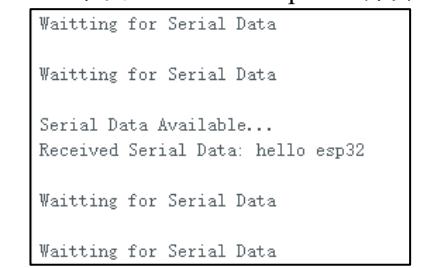


图 10.4.1 串口发送数据和接收数据

第十一章 TIMER_IT 实验

本章，我们将学习 ESP32-S3 的定时器，教会大家如何使用 ESP32-S3 的定时器实现定时功能。在本章中，我们将实现如下功能：开启 ESP32-S3 的定时器，并在定时器的回调函数中，翻转 LED 灯的状态。

本章分为如下 4 个小节：

- 11.1 定时器介绍
- 11.2 硬件设计
- 11.3 软件设计
- 11.4 下载验证

11.1 定时器简介

11.1.1 定时器介绍

定时器，顾名思义，用于设置定时。平常我们设定计时或闹钟，时间到了就告诉我们要做什么了。而这里的定时器同样也是如此，也需要设定定时到后的操作。

ESP32-S3 有通用定时器、系统定时器和看门狗定时器，本章主要讲解的是通用定时器。

ESP32-S3 有两个硬件定时器组，定时器组 0 和定时器组 1，每组有两个硬件通用定时器，所以总共是有 4 个硬件通用定时器。它们都是基于 16 位预分频器和 54 位可自动重载的向上/向下计数器实现定时功能。

ESP32-S3 的计数频率为 80MHz，假如对 16 位预分频器设置预分频系数为 80，那么可得到 1MHz 的计数信号，每个计数信号的周期为 1us，即每个计数单位为 1us。基于要设定的时间，就可以对计数器进行设置。打个比方，要定时 10ms，而每个计数周期为 1us，这里得计算 10ms 需要多少个这样的 1us 周期： $10ms / 1us = 10000$ ，计数器就需要设置为 10000，实现 10ms 定时，这个举例过程如下图所示。

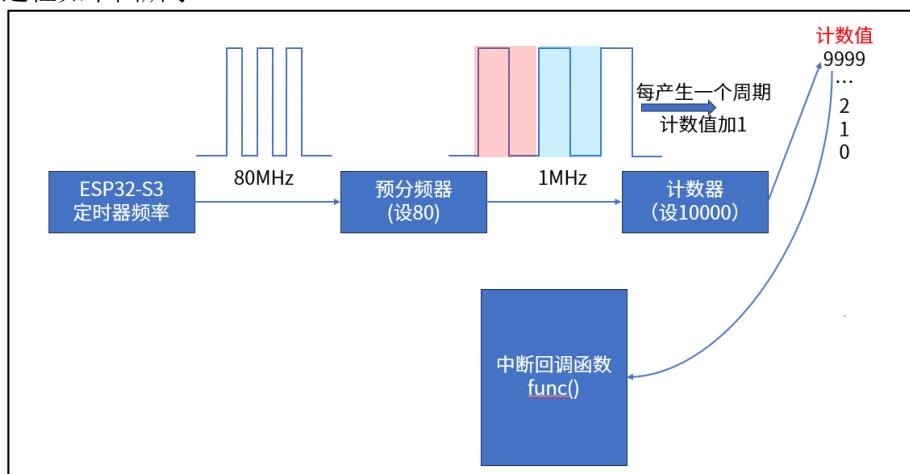


图 11.1.1.1 定时器定时配置过程

当设置好定时器的预分频器以及计数器以及开启定时器，这时候定时开始，当计数值达到 9999 时，即到达设定时间，就会跳进中断回调函数中执行，执行完毕再回到主程序中运行。

11.1.2 定时器函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\esp32-hal-timer.c

接下来，我们介绍一下本章节所用到的 TIMER 相关函数。

第一个函数：timerBegin 函数，该函数功能是初始化一个定时器对象。

```
hw_timer_t * timerBegin(uint8_t num, uint16_t divider, bool countUp);
```

参数 num 为定时器编号, 0 到 3, 对应 4 个硬件通用定时器;

参数 divider 为预分频系数;

参数 countUp 为计数器计数方向标志, true: 向上计数; false: 向下计数

返回值: 定时器结构体指针

第二个函数: timerAttachInterrupt 函数, 该函数功能是为目标定时器绑定一个中断回调函数, 配置定时器中断。

```
void timerAttachInterrupt(hw_timer_t *timer, void (*fn)(), bool edge);
```

参数*timer 为已初始化的目标定时器结构体指针;

参数(*fn)() 为定时器中断回调函数的函数指针;

参数 edge 为中断触发类型, true: 边沿触发, false: 电平触发;

返回值

第三个函数: timerAlarmWrite 函数, 该函数功能是为目标定时器设置间隔定时参数和是否自动重装载。

```
void timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload);
```

参数*timer 为已初始化的目标定时器结构体指针;

参数 alarm_value 为最大计数值。向上计数到达该数值溢出, 触发中断;

参数 autoreload 为定时器在产生中断时是否重新加载的标志。true: 自动加载, 循环间隔定时, false: 不自动加载, 只进行一次间隔定时。

无返回值。

第四个函数: timerAlarmEnable 函数, 该函数功能是使能定时器, 开始间隔定时。

```
void timerAlarmEnable(hw_timer_t *timer);
```

参数*timer 为已初始化的目标定时器结构体指针;

无返回值。

11.2 硬件设计

1. 例程功能

程序启动后配置定时器的定时时间为 500 毫秒, 定时到来时执行中断回调函数翻转 LED 状态。

2. 硬件资源

1) LED 灯

LED-IO1

2) Timer0

3. 原理图

本章实验使用的定时器为 ESP32-S3 的片上资源, 因此并没有相应的连接原理图。

11.3 软件设计

11.3.1 程序流程图

下面看看本实验的程序流程图:

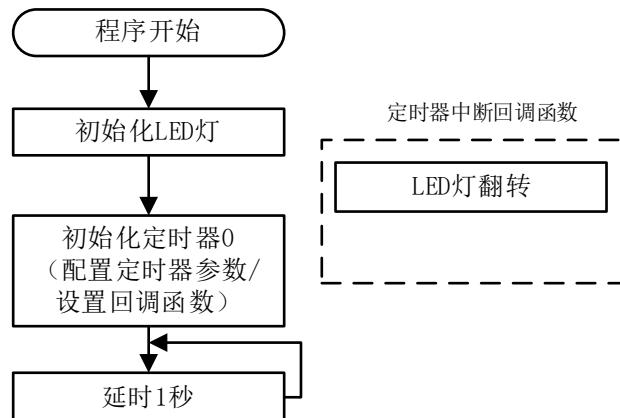


图 11.3.1.1 程序流程图

11.3.2 程序解析

1. timer 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。EXTI 驱动源码包括两个文件：tim.cpp 和 tim.h。

下面我们先解析 tim.h 的程序。我们做了定时器的相关定义。

```
#define TIMx_INT          0
#define TIMx_ISR          tim0_ISR
```

我们选择使用通用定时器 0，当然你也可以通过改变 TIMx_INT 的值进行初始化别的定时器。TIMx_ISR 宏是定时器 0 的中断回调函数，采用该宏移植性更加强。

下面我们再解析 tim.cpp 的程序，这里有两个函数 timx_int_init 和 TIMX_ISR，其定义如下：

```
hw_timer_t *timer = NULL;

/**
 * @brief    定时器 TIMX 定时中断初始化函数
 * @note
 *          定时器的时钟来自 APB，而 APB 为 80M
 *          所以定时器时钟 = (80/psc)Mhz, 单位时间为 1 / (80 / psc) = x us
 *          定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us
 *          Ft=定时器工作频率, 单位:Mhz
 *
 * @param    arr: 自动重装值
 * @param    psc: 时钟预分频数
 * @retval   无
 */
void timx_int_init(uint16_t arr, uint16_t psc)
{
    timer = timerBegin(TIMx_INT, psc, true);           /* 初始化定时器 0 */
    timerAlarmWrite(timer, arr, true);                  /* 设置中断时间 */
    timerAttachInterrupt(timer, &TIMx_ISR, true);      /* 配置定时器中断回调函数 */
    timerAlarmEnable(timer);                           /* 使能定时器中断 */
}

/**
 * @brief    定时器 TIMX 中断回调函数
 * @param    无
 * @retval   无
 */
void TIMx_ISR(void)
{
```

```
    LED_TOGGLE();
}
```

timx_int_init 函数是初始化定时器 0，首先调用 timerBegin 函数初始化一个定时器对象并采用向上计数方式，然后调用 timerAlarmWrite 函数配置中断时间并设置自动加载，后面再调用 timerAttachInterrupt 函数设置定时器边沿触发中断，最后调用 timerAlarmEnable 函数使能开启定时器。

注意：定时器的定时时间是由 timx_int_init 函数的参数决定，这里我们定义了一个全局的 hw_timer_t 类型的结构体指针 timer； timerAlarmWrite 函数的 autoreload 参数需要设置成 true，才能一直循环间隔定时，否则只会执行一次就停下来。

TIMx_ISR 函数就是定时器 0 中断回调函数。函数内部很简单，就是对 LED 灯的状态进行翻转。

2. 05_timer_it.ino 代码

在 05_timer_it.ino 里面编写如下代码：

```
#include "tim.h"
#include "led.h"

/***
 * @brief  当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    led_init();           /* LED 初始化 */
    timx_int_init(5000, 8000); /* 定时器初始化，定时时间为 500ms */
}

/***
 * @brief  循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param  无
 * @retval 无
 */
void loop()
{
    /* 死循环，不做事情，等待定时器中断触发 */
    delay(1000);
}
```

在 setup 函数中，除了要调用 led_init 函数对 LED 灯进行初始化，还要调用 timx_int_init 函数初始化定时器 0，通过参数我们可以算出定时时间为 500 毫秒。启动定时器后，每隔 500 毫秒进入我们编写好的定时器中断回调函数中，执行 LED 灯状态翻转操作。

由于该实验是用来测试定时器中断的，所以在 loop 函数中，不需要做事情，调用 delay 函数即可。

11.4 下载验证

下载代码完成后，ESP32-S3 开发板每隔 500 毫秒触发一次中断，然后在中断回调函数中切换 LED 灯的状态。

第十二章 LED_PWM 实验

本章，我们将学习 ESP32-S3 的 LED PWM 控制器，教会大家如何使用 LED PWM 控制器实现 PWM 信号输出。在本章中，驱动 LED PWM 控制器，输出 PWM 信号去控制 LED 的亮度，实现呼吸灯效果。

本章分为如下 4 个小节：

- 12.1 LED PWM 控制器介绍
- 12.2 硬件设计
- 12.3 软件设计
- 12.4 下载验证

12.1 LED PWM 控制器介绍

12.1.1 PWM 介绍

PWM (Pulse Width Modulation)，简称脉宽调制，是一种将模拟信号变为脉冲信号的技术。PWM 可以控制 LED 亮度、直流电机的转速等。

PWM 的主要参数如下：

- 1) PWM 频率。PWM 频率是 PWM 信号在 1s 内从高电平到低电平再回到高电平的次数，也就是说 1s 内有多少个 PWM 周期，单位为 Hz。
- 2) PWM 周期。PWM 周期是 PWM 频率的倒数，即 $T=1/f$ ， T 是 PWM 周期， f 是 PWM 频率。如果 PWM 频率为 50Hz，也就是说 PWM 周期为 20ms，即 1s 由 50 个 PWM 周期。
- 3) PWM 占空比。PWM 占空比是指在一个 PWM 周期内，高电平的时间与整个周期时间的比例，取值范围为 0%~100%。PWM 占空比如下图所示。

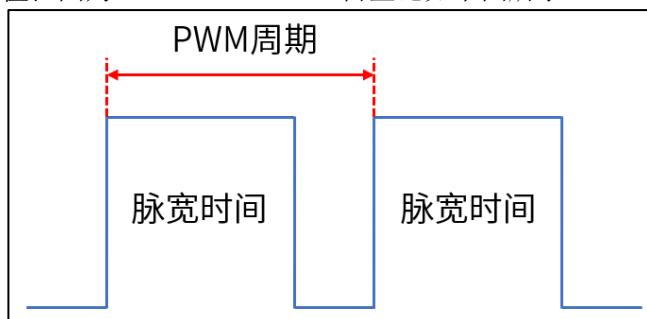


图 12.1.1.1 PWM 占空比

PWM 周期是一个 PWM 信号的时间：脉宽时间是指高电平时间；脉宽时间占 PWM 周期的比例就是占空比。例如，如果 PWM 周期是 10ms，而脉宽时间为 8ms，那么 PWM 占空比就是 $8/10=80\%$ ，此时的 PWM 信号就是占空比为 80% 的 PWM 信号。PWM 名为脉冲宽度调制，顾名思义，就是通过调节 PWM 占空比来调节 PWM 脉宽时间。

在使用 PWM 控制 LED 时，亮 1s 后灭 1s，往复循环，就可以看到 LED 在闪烁。如果把这个周期缩小到 200ms，亮 100ms 后灭 100ms，往复循环，就可以看到 LED 灯在高频闪烁。继续把这个周期持续缩小，总有一个临界值使人眼分辨不出 LED 在闪烁，此时 LED 的亮度处于灭与亮之间亮度的中间值，达到了 1/2 亮度。PWM 占空比和亮度的关系如下图所示。

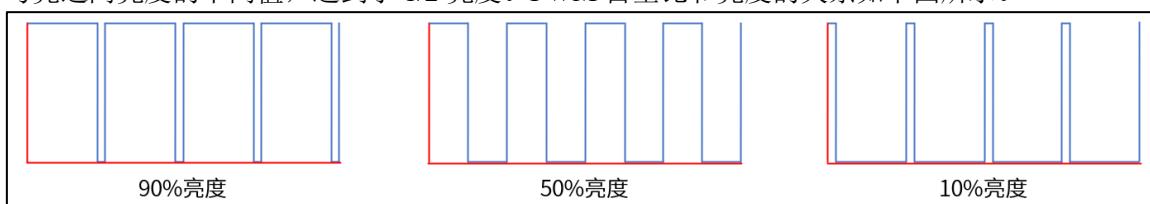


图 12.1.1.2 PWM 占空比和亮度的关系

12.1.2 LED_PWM 控制器介绍

ESP32-S3 的 LED PWM 控制器，简写为 LEDC，用于生成控制 LED 的脉冲宽度调制信号。

LED PWM 控制器具有八个独立的 PWM 生成器（即八个通道）。每个 PWM 生成器会从四个通用定时器中选择一个，以该定时器的计数值作为基准生成 PWM 信号。LED PWM 定时器如下图所示。

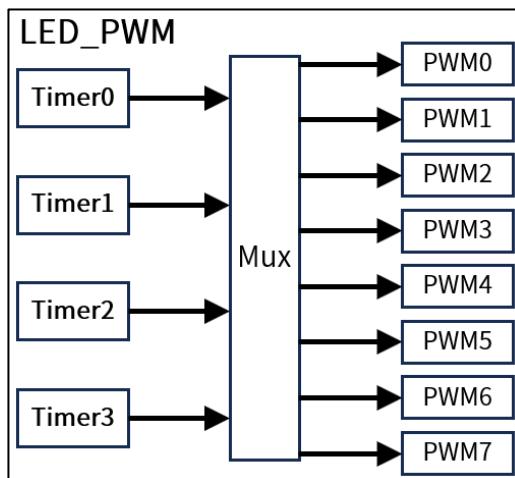


图 12.1.2.1 LED_PWM 的定时器

为了实现 PWM 输出，先需要设置指定通道的 PWM 参数：频率、分辨率、占空比，然后将该通道映射到指定引脚，该引脚输出对应通道的 PWM 信号，通道和引脚的关系所下图所示。

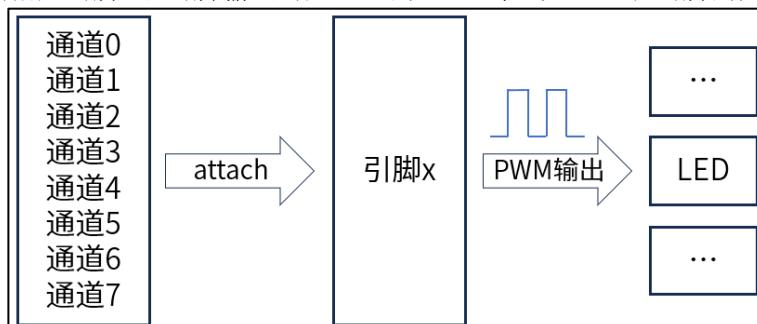


图 12.1.2.2 LED_PWM 输出示意图

12.1.3 LED_PWM 函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\cores\esp32\esp32-hal-ledc.c

接下来，我们介绍一下本章节所用到的 LED PWM 相关函数。

第一个函数：ledcSetup 函数，该函数功能是指定 LEDC 通道的 PWM 信号频率和占空比分辨率。

```
double ledcSetup(uint8_t chan, double freq, uint8_t bit_num);
```

参数 chan 为 LEDC 通道号，取值为 0~7，共 8 个通道；

参数 freq 为待设置的 PWM 脉宽信号的频率；

参数 bit_num 为计数位数，即 PWM 信号占空比的分辨率；

返回值：通道 PWM 信号的频率。

第二个函数：ledcAttachPin 函数，该函数功能是将指定的 LEDC 通道绑定到指定 GPIO 引脚上，即由该引脚输出 LEDC 的 PWM 信号。

```
void ledcAttachPin(uint8_t pin, uint8_t chan);
```

参数 pin 为数字引脚编号；

参数 chan 为 LEDC 通道号，取值为 0~7，共 8 个通道；

无返回值。

第三个函数: ledcWrite 函数, 该函数功能是设置指定通道输出的占空比数值。

```
void ledcWrite(uint8_t chan, uint32_t duty);
```

参数 chan 为 LEDC 通道号, 取值为 0~7, 共 8 个通道;

参数 duty 为待设置的 PWM 占空比数值。该数值的范围由通道初始化设置函数 ledcSetup() 中的计数位数决定。例如, 计数位数为 8, 那么占空比设置值的范围就为 0~255。要输出占空比 50% 的 PWM 信号, 该参数应设置为 128。

无返回值。

12.2 硬件设计

1. 例程功能

实现 LED0 由暗变亮, 再从亮变暗, 依次循环。

2. 硬件资源

- 1) LED PWM
LED-IO1

3. 原理图

本章实验使用的 LED PWM 为 ESP32-S3 的片上资源, 因此并没有相应的连接原理图。

12.3 软件设计

12.3.1 程序流程图

下面看看本实验的程序流程图:

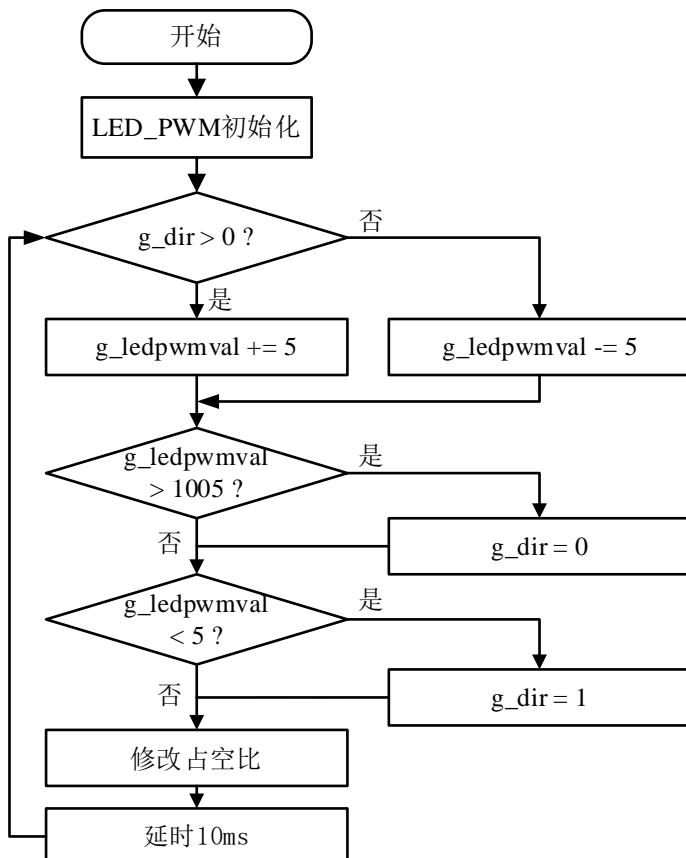


图 12.3.1.1 程序流程图

12.3.2 程序解析

1. pwm 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWM 驱动源码包括两个文件：pwm.cpp 和 pwm.h。

下面我们先解析 pwm.h 的程序。对 LED PWM 做了相关定义。

```
#define LED_PWM_PIN 1 /* PWM 信号输出的引脚 */
#define LED_PWM_CHANNEL 0 /* LED PWM 通道号 */
```

我们选择使用 LED PWM 的通道 0，将该通道映射到 IO1，最终 IO1 引脚会输出通道 0 的 PWM 信号。

下面我们再解析 pwm.cpp 的程序，这里有两个函数 led_pwm_init 和 pwm_set_duty，其定义如下：

```
/***
 * @brief LED PWM 初始化函数
 * @param frequency: PWM 输出频率，单位 HZ
 * @param resolution: PWM 占空比的分辨率 1-16，比如设置 8，分辨率范围 0~255
 * @retval 无
 */
void led_pwm_init(uint16_t frequency, uint8_t resolution)
{
    ledcSetup(LED_PWM_CHANNEL, frequency, resolution);
    /* PWM 初始化，引脚和通道由 pwm.h 的 LED_PWM_PIN 和 LED_PWM_CHANNEL 宏修改 */
    ledcAttachPin(LED_PWM_PIN, LED_PWM_CHANNEL);
    /* 绑定 PWM 通道到 LED_PWM_PIN 上 */
}

/***
 * @brief PWM 占空比设置
 * @param duty: PWM 占空比
 * @retval 无
 */
void pwm_set_duty(uint16_t duty)
{
    ledcWrite(LED_PWM_CHANNEL, duty);
    /* 改变 PWM 的占空比，通道由 pwm.h 的 LED_PWM_CHANNEL 宏修改 */
}
```

led_pwm_init 函数是初始化 PWM 输出功能，首先调用 ledcSetup 函数设置 LEDC 通道参数：PWM 通道、PWM 频率和占空比的分辨率。然后调用 ledcAttachPin 函数绑定 PWM 通道到 IO1 引脚。

pwm_set_duty 函数就是调用 ledcWrite 函数去完成 PWM 占空比的修改。

2. 07_led_pwm.ino 代码

在 07_led_pwm.ino 里面编写如下代码：

```
#include "pwm.h"

uint16_t g_ledpwmval = 0; /* 占空比值 */
uint8_t g_dir = 1; /* 变化方向 (1 增大 0 减小) */

/***
 * @brief 当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param 无
 * @retval 无
 */
void setup()
{
```

```
led_pwm_init(1000, 10); /* LED PWM 初始化, PWM 输出频率为 1000HZ, 占空比分辨率为 10 */  
}  
  
/**  
 * @brief 循环函数, 通常放程序的主体或者需要不断刷新的语句  
 * @param 无  
 * @retval 无  
 */  
void loop()  
{  
    if (g_dir)  
    {  
        g_ledpwmval += 5;  
    }  
    else  
    {  
        g_ledpwmval -= 5;  
    }  
  
    if (g_ledpwmval > 1005)  
    {  
        g_dir = 0;  
    }  
  
    if (g_ledpwmval < 5)  
    {  
        g_dir = 1;  
    }  
  
    pwm_set_duty(g_ledpwmval);  
    delay(10);  
}
```

在 setup 函数中，调用 led_pwm_init 函数完成 PWM 输出的初始化。

在 loop 函数中，通过简单的逻辑判断，让 g_ledpwmval 变量可从小变大，又从大变小，最终通过 pwm_set_duty 函数传递 g_ledpwmval 参数去修改占空比，达到修改 LED 亮度效果。为了让效果更为明显，添加了延时 10ms 操作。

12.4 下载验证

下载代码完成后，ESP32-S3 开发板上 LED 灯的亮度会以一个平稳的速度从最暗变到最亮，然后再从最亮变到最暗，实现呼吸灯的效果。

第十三章 SPI_LCD 实验

本章, 我们将学习 ESP32-S3 的硬件 SPI 接口, 将会大家如何使用 SPI 接口去驱动 LCD 屏。在本章中, 实现和 LCD 屏之间的通信, 实现 ASCII 字符、彩色、图片和图形的显示。

本章分为如下 4 个小节:

13.1 SPI 及 LCD 介绍

13.2 硬件设计

13.3 软件设计

13.4 下载验证

13.1 SPI 及 LCD 介绍

13.1.1 SPI 介绍

SPI, Serial Peripheral interface, 顾名思义, 就是串行外围设备接口, 是由原摩托罗拉公司在其 MC68HCXX 系列处理器上定义的。SPI 是一种高速的全双工、同步、串行的通信总线, 已经广泛应用在众多 MCU、存储芯片、AD 转换器和 LCD 之间。

SPI 通信跟 IIC 通信一样, 通信总线上允许挂载一个主设备和一个或者多个从设备。为了跟从设备进行通信, 一个主设备至少需要 4 跟数据线, 分别为:

MOSI (Master Out / Slave In): 主数据输出, 从数据输入, 用于主机向从机发送数据。

MISO (Master In / Slave Out): 主数据输入, 从数据输出, 用于从机向主机发送数据。

SCLK (Serial Clock): 时钟信号, 由主设备产生, 决定通信的速率。

CS (Chip Select): 从设备片选信号, 由主设备产生, 低电平时选中从设备。

多从机 SPI 通信网络连接如下图所示。

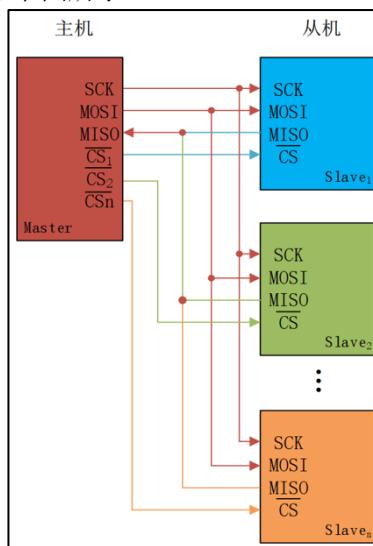


图 13.1.1.1 多从机 SPI 通信网络图

从上图可以知道, MOSI、MISO、SCLK 引脚连接 SPI 总线上每一个设备, 如果 CS 引脚为低电平, 则从设备只侦听主机并与主机通信。SPI 主设备一次只能和一个从设备进行通信。如果主设备要和另外一个从设备通信, 必须先终止和当前从设备通信, 否则不能通信。

SPI 通信有 4 种不同的模式, 不同的从机可能在出厂时就配置为某种模式, 这是不能改变的。通信双方必须工作在同一模式下, 才能正常进行通信, 所以可以对主机的 SPI 模式进行配置。SPI 通信模式是通过配置 CPOL (时钟极性) 和 CPHA (时钟相位) 来选择的。

CPOL, 详称 Clock Polarity, 就是时钟极性, 当主从机没有数据传输的时候即空闲状态, SCL 线的电平状态, 假如空闲状态是高电平, CPOL=1; 若空闲状态时低电平, 那么 CPOL=0。

CPHA, 详称 Clock Phase, 就是时钟相位, 实质指的是数据的采样时刻。CPHA=0 表示数据的采样是从第 1 个边沿信号上即奇数边沿, 具体是上升沿还是下降沿的问题, 是由 CPOL 决定的。CPHA=1 表示数据采样是从第 2 个边沿即偶数边沿。

SPI 的 4 种模式对比图, 如下图所示。

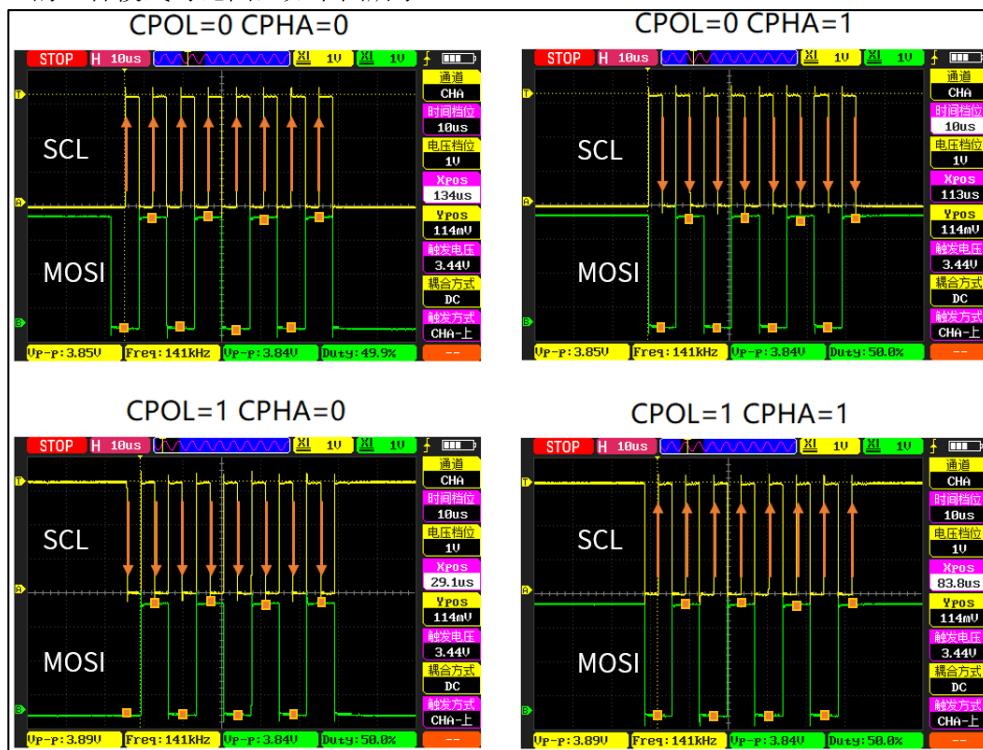


图 13.1.1.2 SPI 的 4 种模式对比图

- 1) 模式 0, CPOL=0, CPHA=0; 空闲时, SCL 处于低电平, 数据采样在第 1 个边沿, 即 SCL 由低电平到高电平的跳变, 数据采样在上升沿, 数据发送在下降沿。
- 2) 模式 1, CPOL=0, CPHA=1; 空闲时, SCL 处于低电平, 数据采样在第 2 个边沿, 即 SCL 由高电平到低电平的跳变, 数据采样在下升沿, 数据发送在上降沿。
- 3) 模式 2, CPOL=1, CPHA=0; 空闲时, SCL 处于高电平, 数据采样在第 1 个边沿, 即 SCL 由高电平到低电平的跳变, 数据采样在下升沿, 数据发送在上降沿。
- 4) 模式 3, CPOL=1, CPHA=1; 空闲时, SCL 处于高电平, 数据采样在第 2 个边沿, 即 SCL 由低电平到高电平的跳变, 数据采样在上升沿, 数据发送在下降沿。

13.1.2 SPI 控制器介绍

ESP32-S3 芯片集成了四个 SPI 控制器, 分别为 SPI0、SPI1、SPI2 和 SPI3。SPI0 和 SPI1 控制器主要供内部使用以访问外部 FLASH 和 PSRAM, 所以只能使用 SPI2 和 SPI3。SPI2 又称为 HSPI, 而 SPI3 又称为 VSPI, 这两个属于 GP-SPI。

GP-SPI 特性:

- 支持主机模式和从机模式
- 支持半双工通信和全双工通信
- 支持多种数据模式:

SPI2: 1-bit SPI 模式、2-bit Dual SPI 模式、4-bit Quad SPI 模式、QPI 模式、8-bit Octal 模式、OPI 模式

SPI3: 1-bit SPI 模式、2-bit Dual SPI 模式、4-bit Quad SPI 模式、QPI 模式

时钟频率可配置:

在主机模式下: 时钟频率可达 80MHz

在从机模式下: 时钟频率可达 60MHz

数据位的读写顺序可配置

时钟极性和相位可配置

四种 SPI 时钟模式：模式 0 ~ 模式 3

在主机模式下，提供多条 CS 线

SPI2: CS0 ~ CS5

SPI3: CS0 ~ CS2

支持访问 SPI 接口的传感器、显示屏控制器、flash 或 RAM 芯片

SPI2 和 SPI3 接口相关信号线可以经过 GPIO 交换矩阵和 IO_MUX 实现与芯片引脚的映射，IO 使用起来非常灵活。

13.1.3 LCD 介绍

ESP32S3 最小系统板板载 0.96 英寸高清 IPS LCD 显示屏，其分辨率为 160x80，支持 16 位真彩色显示。该显示屏采用 ST7735S 作为驱动芯片，其内置 RAM 无需外部驱动器或存储器。ESP32S3 芯片仅需通过 SPI 接口即可轻松驱动此显示屏。

显示屏的外观，如下图所示。



图 13.1.3.1 显示屏实物图

该屏幕通过 13 个引脚与 PCB 电路连接。引脚详细描述，如下表所示。

序号	名称	说明
1	TP0	NC
2	TP1	NC
3	SDA	SPI 通讯 MOSI 信号线
4	SCL	SPI 通讯 SCK 信号线
5	RS	写命令/数据信号线（低电平：写命令；高电平：写数据）
6	RES	硬件复位引脚（低电平有效）
7	CS	SPI 通讯片选信号（低电平有效）
8	GND	电源地
9	NC	NC
10	VCC	3.3V 电源供电
11	LEDK	LCD 背光控制引脚（阴极）
12	LEDA	LCD 背光控制引脚（阳极）
13	GND	电源地

表 13.1.3.1 0.96 寸 LCD 引脚说明

0.96 寸 LCD 屏在四线 SPI 通讯模式下，仅需四根信号线 (CS、SCL、SDA、RS (DC)) 就能够驱动。

四线 SPI 接口时序如下图所示。

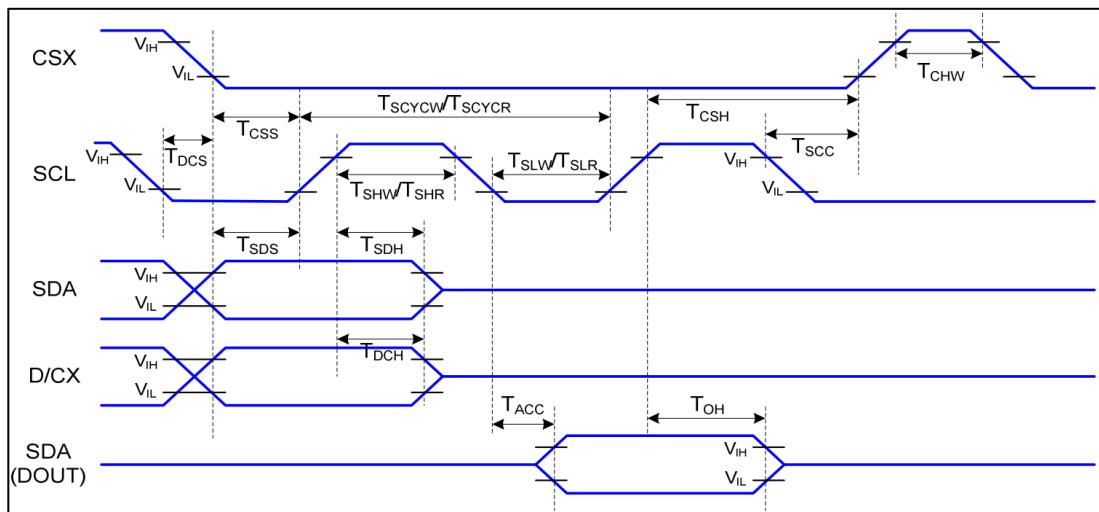


图 13.1.3.2 四线 SPI 接口时序图

上图中各个时间参数，如下表所示。

Signal	Symbol	Parameter	MIN	MAX	Unit	Description
CSX	TCSS	Chip Select Setup Time (Write)	45		ns	-Write Command & Data Ram
	TCSH	Chip Select Hold Time (Write)	45		ns	
	TCSS	Chip Select Setup Time (Read)	60		ns	
	TSCC	Chip Select Hold Time (Read)	65		ns	
	TCHW	Chip Select "H" Pulse Width	40		ns	
SCL	TSCYCW	Serial Clock Cycle (Write)	66		ns	-Write Command & Data Ram
	TSHW	SCL "H" Pulse Width (Write)	15		ns	
	TSLW	SCL "L" Pulse Width (Write)	15		ns	
	TSCYCR	Serial Clock Cycle (Read)	150		ns	
	TSHR	SCL "H" Pulse Width (Read)	60		ns	
	TSLR	SCL "L" Pulse Width (Read)	60		ns	
D/CX	TDCS	D/CX Setup Time	10		ns	
	TDCH	D/CX Hold Time	10		ns	
SDA (DIN) (DOUT)	TSDS	Data Setup Time	10		ns	For Maximum CL=30pF
	TSDH	Data Hold Time	10		ns	
	TACC	Access Time	10	50	ns	For Minimum CL=8pF
	TOH	Output Disable Time	15	50	ns	

表 13.1.3.2 四线 SPI 接口时序参数表

从上图中可以看出，0.96 寸 LCD 模块四线 SPI 的写周期是非常快的 ($T_{SCYCW} = 66\text{ns}$)，而读周期就相对慢了很多 ($T_{SCYCR} = 150\text{ns}$)。

更详细的时序介绍，可以参考 ST7735S 的数据手册《ST7735S_V1.1_20111121.pdf》。

0.96 寸 LCD 屏采用 ST7735S 作为 LCD 驱动器，LCD 的显存可直接存放在 ST7735S 的片上 RAM 中，ST7735S 的片上 RAM 有 $132*162*18\text{-bits}$ ，并且 ST7735S 会在没有外部时钟的情况下，自动将其片上 RAM 的数据显示至 LCD 上，以最小化功耗。

在每次初始化显示模块之前，必须先通过 RST 引脚对显示模块进行硬件复位，硬件复位要求 RST 至少被拉低 10 微秒，拉高 RST 结束硬件复位后，须延时 120 毫秒等待复位完成后，才能够往显示模块传输数据。

LEDK 引脚用于控制显示模块的 LCD 背光，该引脚自带下拉电阻，当 LEDK 引脚被拉高或悬空时，0.96 寸 LCD 模块的 LCD 背光都处于关闭状态，当 LEDK 引脚被拉低时，显示模块的 LCD 背光才会点亮。

ST7735S 最高支持 18 位色深 (262K 色), 不过一般使用 16 位颜色深度 (65K 色), RGB565 格式, 这样可以在 16 位色深下达到最快的速度。在 16 位色深模式下, ST7789V 采用 RGB565 格式传输、存储颜色数据, 如下图所示。

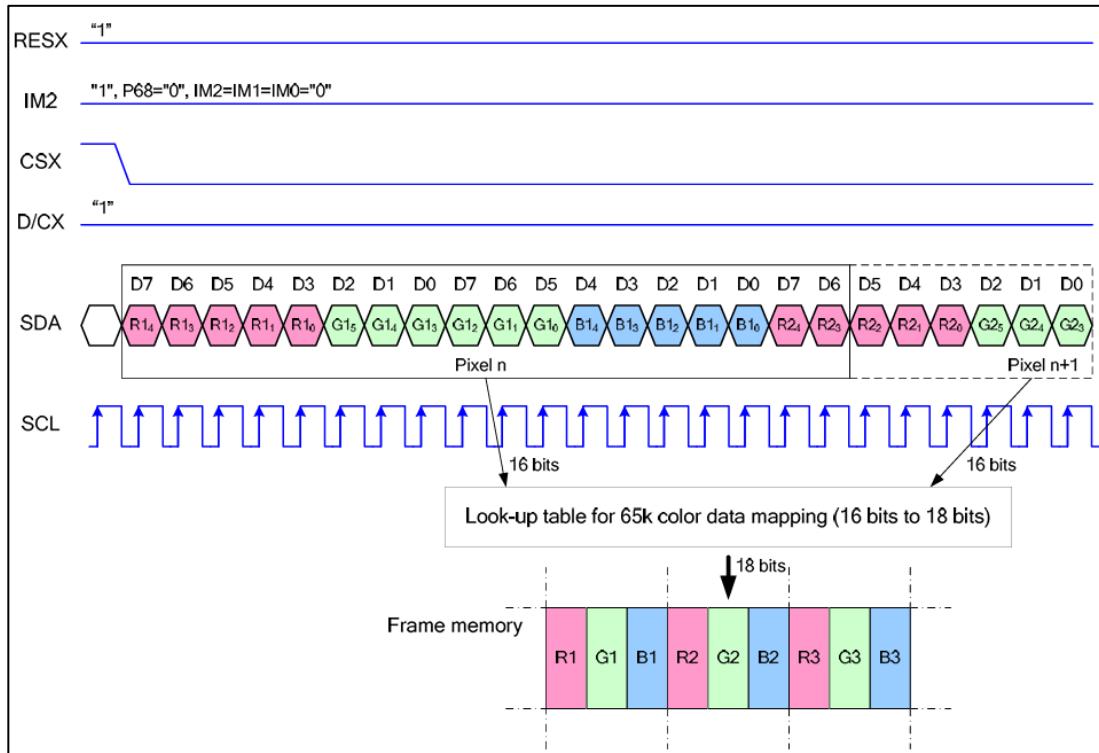


图 13.1.3.3 16 位色深模式 (RGB565) 传输颜色数据

上图是一个传输像素数据的时序过程, D/CX 线需要拉高, 表示传输的是数据。一个像素的颜色数据需要使用 16 比特来传输, 这 16 比特数据中, 高 5 比特用于表示红色, 低 5 比特用于表示蓝色, 中间的 6 比特用于表示绿色。数据的数值越大, 对应表示的颜色就越深。

ST7735S 支持连续读写 RAM 中存放的 LCD 上颜色对应的数据, 并且连续读写的方向 (LCD 的扫描方向) 是可以通过命令 0x36 进行配置的, 如下图所示。

MADCTL (Memory Data Access Control)													HEX
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
MADCTL	0	↑	1	-	0	0	1	1	0	1	1	0	(36h)
parameter	1	↑	1	-	MY	MX	MV	ML	RGB	MH	-	-	
-This command defines read/ write scanning direction of frame memory.													
Bit		NAME				DESCRIPTION							
D7		MY				Page Address Order							
D6		MX				Column Address Order							
D5		MV				Page/Column Order							
D4		ML				Line Address Order							
D3		RGB				RGB/BGR Order							
D2		MH				Display Data Latch Order							

图 13.1.3.4 命令 0x36 描述

从上图中可以看出, 命令 0x36 可以配置 6 个参数, 但对于配置 LCD 的扫描方向, 仅需关心 MY、MX 和 MV 这三个参数, 如下表所示。

参数			LCD 扫描方向 (RAM 自增方向)		
MY	MX	MY	MX		

0	0	0	从左到右, 从上到下
1	0	0	从左到右, 从下到上
0	1	0	从右到左, 从上到下
1	1	0	从右到左, 从下到上
0	0	1	从上到下, 从左到右
1	0	1	从上到下, 从右到左
0	1	1	从下到上, 从左到右
1	1	1	从下到上, 从右到左

表 13.1.3.3 命令 0x36 配置 LCD 扫描方向

这样, 我们在使用 ST7735S 显示内容的时候, 就有很大灵活性了, 比如显示 BMP 图片, BMP 解码数据, 就是从图片的左下角开始, 慢慢显示到右上角, 如果设置 LCD 扫描方向为从左到右, 从下到上, 那么我们只需要设置一次坐标, 然后就不停的往 LCD 填充颜色数据即可, 这样可以大大提高显示速度。

在往 ST7735S 写入颜色数据前, 还需要设置地址, 以确定随后写入的颜色数据对应 LCD 上的哪一个像素, 通过命令 0x2A 和命令 0x2B 可以分别设置 ST7735S 显示颜色数据的列地址和行地址, 命令 0x2A 的描述, 如下图所示。

2AH		CASET(Column Address Set)_											
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
CASET(2Ah)	0	↑	1	-	0	0	1	0	1	0	1	0	(2Ah)
1 st Parameter	1	↑	1	-	XS15	XS14	XS13	XS12	XS11	XS10	XS9	XS8	
2 nd Parameter	1	↑	1	-	XS7	XS6	XS5	XS4	XS3	XS2	XS1	XS0	
3 rd Parameter	1	↑	1	-	XE15	XE14	XE13	XE12	XE11	XE10	XE9	XE8	
4 th Parameter	1	↑	1	-	XE7	XE6	XE5	XE4	XE3	XE2	XE1	XE0	

图 13.1.3.5 命令 0x2A 描述

命令 0x2B 的描述, 如下图所示。

2BH		RASET (Row Address Set)											
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
RASET (2Bh)	0	↑	1	-	0	0	1	0	1	0	1	1	(2Bh)
1 st Parameter	1	↑	1	-	YS15	YS14	YS13	YS12	YS11	YS10	YS9	YS8	
2 nd Parameter	1	↑	1	-	YS7	YS6	YS5	YS4	YS3	YS2	YS1	YS0	
3 rd Parameter	1	↑	1	-	YE15	YE14	YE13	YE12	YE11	YE10	YE9	YE8	
4 th Parameter	1	↑	1	-	YE7	YE6	YE5	YE4	YE3	YE2	YE1	YE0	

图 13.1.3.6 命令 0x2B 描述

以默认的 LCD 扫描方式 (从左到右, 从上到下) 为例, 命令 0x2A 的参数 XS 和 XE 和命令 0x2B 的参数 YS 和 YE 就在 LCD 上确定了一个区域, 在连读读写颜色数据时, ST7735S 就会按照从左到右, 从上到下的扫描方式读写设个区域的颜色数据。

13.1.4 TFT_eSPI 库介绍

前面已经对 LCD 的驱动做了说明, 简单来说, 使用 SPI 接口跟 LCD 驱动芯片 ST7735S 进行通信, 向 ST7735S 写入配置数据进行初始化, 然后就是画点。画点的流程就是: 设置坐标 → 写 GRAM 指令 → 写入颜色数据。

LCD 驱动, 基于 SPI 库函数自行实现, 可以参考正点原子 ESP32-S3 开发板的 SPI_LCD 实验。在这里, 我们直接使用别人写好的库进行实现 LCD 驱动。这里采用的库是 TFT_eSPI, 它是一个高性能、跨平台的库, 旨在为 Arduino 平台上的各种 TFT 显示器提供简单而强大的支持。这个库的出现, 极大简化了 Arduino 用户在处理彩色图形显示任务时的复杂性, 使得硬件爱好者和开发者可以更专注于他们的创意应用, 而不是底层的驱动程序编写。

接下来, 我们就来看一下如何下载 TFT_eSPI 库包, 步骤如下图所示:



图 13.1.4.1 安装 TFT_eSPI 库包步骤

在线下载过程可能会失败，不用担心，我们采用第二种方式：手动安装。在“资料盘→6，软件资料→2，Arduino 软件包”下，有“TFT_eSPI-master.zip”压缩包，我们就是要安装这个软件包，操作如下：

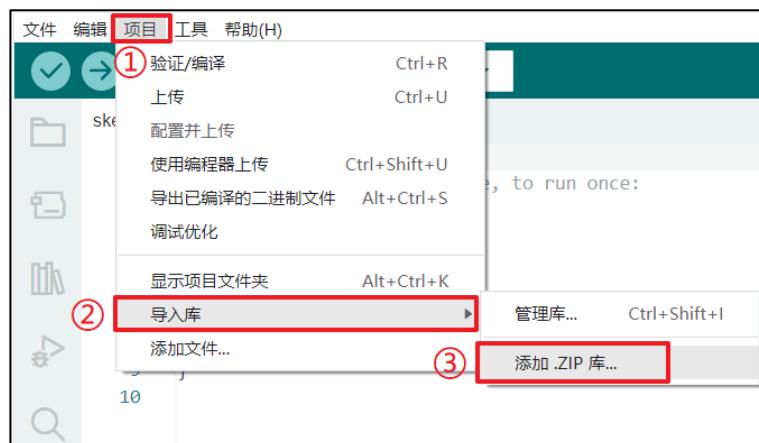


图 13.1.4.2 手动添加软件库

通过操作“项目→导入库→添加.ZIP 库”，后面会跳出一个页面，我们找到并选中“TFT_eSPI-master.zip”压缩包，这时候 IDE 就会安装这个库。安装成功如下图所示。

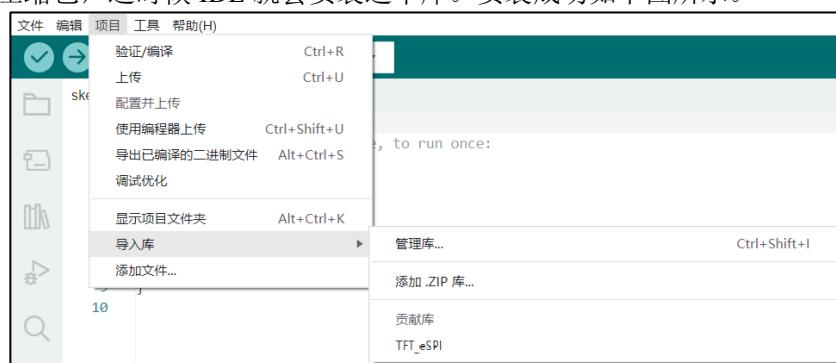


图 13.1.4.3 TFT_eSPI 库安装成功

当我们下载好“TFT_eSPI”库后，可以从“C:\Users\用户名\Documents\Arduino\libraries\”路径下找到该文件，该文件夹内容如下图所示。

名称	修改日期	类型	大小
docs	2024/5/7 18:05	文件夹	
examples	2024/5/7 18:05	文件夹	
Extensions	2024/5/7 18:05	文件夹	
Fonts	2024/5/7 18:05	文件夹	
Processors	2024/5/7 18:05	文件夹	
TFT_Drivers	2024/5/7 18:05	文件夹	
Tools	2024/5/7 18:05	文件夹	
User_Setups	2024/5/7 18:05	文件夹	
CMakeLists.txt	2024/5/7 18:05	文本文档	1 KB
Kconfig	2024/5/7 18:05	文件	13 KB
keywords.txt	2024/5/7 18:05	文本文档	4 KB
library.json	2024/5/7 18:05	JSON 源文件	1 KB
library.properties	2024/5/7 18:05	Properties 源文件	1 KB
license.txt	2024/5/7 18:05	文本文档	7 KB
README.md	2024/5/7 18:05	Markdown File	20 KB
README.txt	2024/5/7 18:05	文本文档	1 KB
TFT_config.h	2024/5/7 18:05	C Header 源文件	10 KB
TFT_eSPI.cpp	2024/5/7 18:05	C++ 源文件	192 KB
TFT_eSPI.h	2024/5/7 18:05	C Header 源文件	47 KB
User_Setup.h	2024/5/7 18:05	C Header 源文件	19 KB
User_Setup_Select.h	2024/5/7 18:05	C Header 源文件	18 KB

图 13.1.4.4 TFT_eSPI 库包文件夹

在该文件夹下，特别需要注意的是 User_Setup.h 和 User_Setup_Select.h 两个文件，这两个文件属于 TFT_eSPI 库的配置文件，一个支持自定义参数，一个支持使用已有配置驱动 TFT 屏幕。本例程选择直接对 User_Setup.h 进行配置，配置 TFT_eSPI 库适配 ST7735S，从而驱动好板载的 0.96 寸 LCD 屏。对 User_Setup.h 的具体配置，将会在 13.3.2 程序解析章节说明。

该库下面的 examples 文件夹下存放的是示例工程，可以快速了解库的使用。打开方式有两种：从文件夹中打开，从 IDE 进行打开。从 IDE 打开成功安装的库包示例工程操作如下图所示。

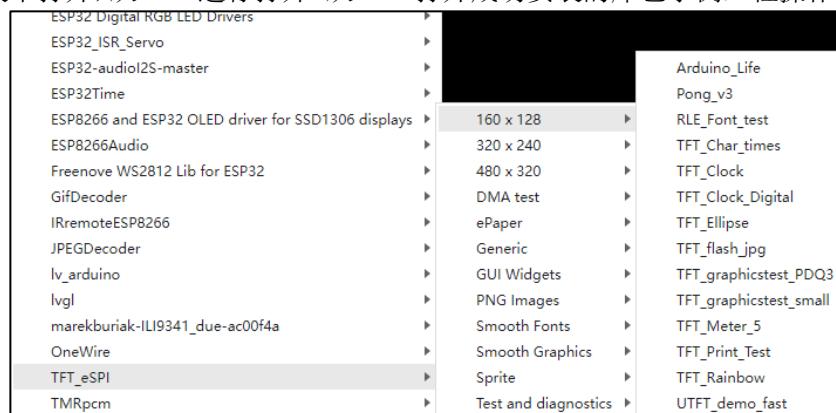


图 13.1.4.5 TFT_eSPI 库示例工程

TFT_Drivers 文件夹存放的是各个 LCD 驱动芯片的驱动代码，而 User_Setups 文件夹存放的是一些厂家用的 LCD 的引脚、字体、速度的配置，大家有兴趣去研究的可以自行查看。

在 TFT_eSPI 库中，主要分为初始化、文字、绘制文字、绘制几何图形和图片显示等功能函数，这里就简单介绍一下比较常用的，其余函数可以自行去看源文件。

初始化相关函数如下：

```
void init(uint8_t tc = TAB_COLOUR);           /* 初始化 */
void begin(uint8_t tc = TAB_COLOUR);           /* 初始化 */
void fillScreen(uint32_t color);                /* 清屏 */
void setRotation(uint8_t r);                    /* 设置图像方向 */
```

注：begin 与 init 是相同的，可以在源码中看到，在 begin 直接调用了 init 函数。

文字相关功能函数如下：

```
void setCursor(uint16_t x, uint16_t y);        /* 设置文本显示坐标 */
void setTextColor(uint16_t color);               /* 设置文本颜色 */
void setTextColor(uint16_t fgcolor, uint16_t bgcolor); /* 设置文本颜色与背景色 */
```

```
void setTextSize(uint8_t size);           /* 设置文本大小 */
tft.print("hello world");                /* 显示字体 */
tft.printf("hello world");               /* 显示字体 */
```

绘制文字相关功能函数如下：

```
/* 绘制字符串(居左) */
int16_t drawString(const char *string, int32_t x, int32_t y, uint8_t font);
int16_t drawString(const char *string, int32_t x, int32_t y);
int16_t drawString(const String& string, int32_t x, int32_t y, uint8_t font);
int16_t drawString(const String& string, int32_t x, int32_t y);

/* 绘制字符串(居左) */
int16_t drawCentreString(const char *string, int32_t x, int32_t y, uint8_t font);
int16_t drawCentreString(const String& string, int32_t x, int32_t y, uint8_t font);

/* 绘制字符串(居右) */
int16_t drawRightString(const char *string, int32_t x, int32_t y, uint8_t font);
int16_t drawRightString(const String& string, int32_t x, int32_t y, uint8_t font);

/* 绘制字符 */
int16_t drawChar(uint16_t uniCode, int32_t x, int32_t y);
int16_t drawChar(uint16_t uniCode, int32_t x, int32_t y, uint8_t font);
void drawChar(int32_t x, int32_t y, uint16_t c, uint32_t color, uint32_t bg,
uint8_t size);

/* 绘制浮点数 */
int16_t drawFloat(float floatNumber, uint8_t decimal, int32_t x, int32_t y);
int16_t drawFloat(float floatNumber, uint8_t decimal, int32_t x, int32_t y, uint8_t font);

/* 绘制数字 */
int16_t drawNumber(long intNumber, int32_t x, int32_t y, uint8_t font);
int16_t drawNumber(long intNumber, int32_t x, int32_t y);
```

绘制几何图形相关功能函数如下：

```
void drawPixel(int32_t x, int32_t y, uint32_t color);           /* 画点 */
/* 画线 */
void drawLine(int32_t xs, int32_t ys, int32_t xe, int32_t ye, uint32_t color);
void drawFastVLine(int32_t x, int32_t y, int32_t h, uint32_t color); /* 画竖线 */
void drawFastHLine(int32_t x, int32_t y, int32_t w, uint32_t color); /* 画横线 */
void drawCircle(int32_t x, int32_t y, int32_t r, uint32_t color); /* 画空心圆 */
void fillCircle(int32_t x, int32_t y, int32_t r, uint32_t color); /* 画实心圆 */
/* 画空心椭圆 */
void drawEllipse(int16_t x, int16_t y, int32_t rx, int32_t ry, uint16_t color);
/* 画实心椭圆 */
void fillEllipse(int16_t x, int16_t y, int32_t rx, int32_t ry, uint16_t color);
/* 画空心矩形 */
void drawRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color);
/* 画实心矩形 */
void fillRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color);
/* 画空心三角形 */
void drawTriangle(int32_t x1, int32_t y1, int32_t x2, int32_t y2, int32_t
x3, int32_t y3, uint32_t color);
/* 画实心三角形 */
void fillTriangle(int32_t x1, int32_t y1, int32_t x2, int32_t y2, int32_t
x3, int32_t y3, uint32_t color);
```

图片显示相关功能函数如下：

```
void drawBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w,
int16_t h, uint16_t fgcolor);
void drawXBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w,
int16_t h, uint16_t fgcolor);
void pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data);
```

在使用前面罗列的函数前，首先实例化一个 TFT_eSPI 对象 myGLCD，因为前面的函数都是 TFT_eSPI 类中的成员函数，操作如下：

```
TFT_eSPI myGLCD = TFT_eSPI();
```

有了 TFT_eSPI 对象就可以调用前面函数，使用以下格式：
`myGLCD.init();`

13.2 硬件设计

1. 例程功能

按下复位之后，就可以看到 LCD 屏不停的显示一些信息并不断切换底色。LED 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11
DC-IO40
BL-IO41
RST-IO38

3. 原理图

LCD 原理图，如下图所示。

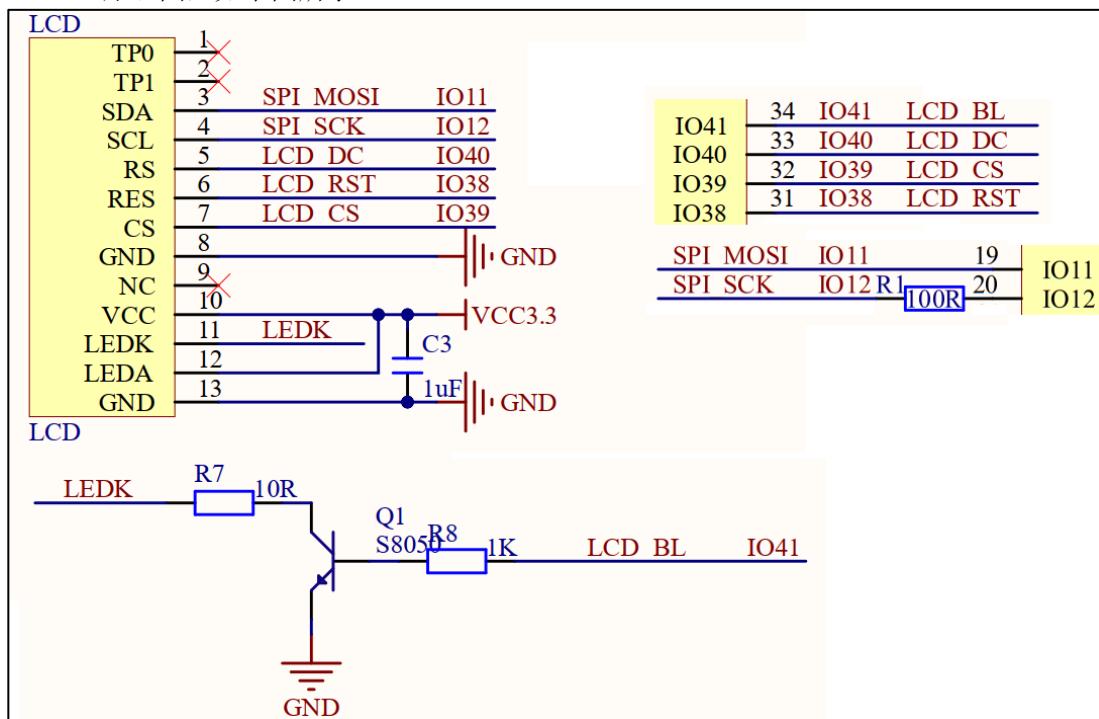


图 13.2.1 LCD 原理图

13.3 软件设计

13.3.1 程序流程图

下面看看本实验的程序流程图：

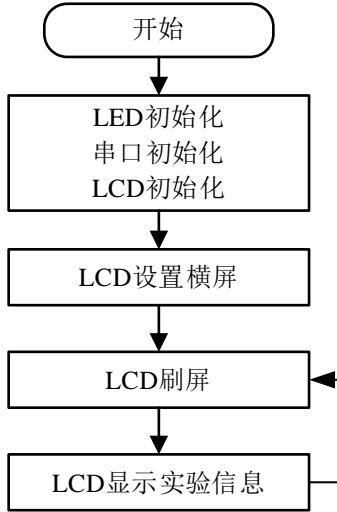


图 13.3.1.1 程序流程图

13.3.2 程序解析

1. TFT_eSPI 库 User_Setup.h 文件修改

这里针对板载的 0.96 寸 LCD 屏主要就是做了如下修改。

1、设置屏幕的驱动（注释掉默认的驱动，将 ST7735 驱动取消注释）

```

44 // Only define one driver, the other ones must be commented out
45 //#define ILI9341_DRIVER 注释 // Generic driver for common displays
46 //#define ILI9341_2_DRIVER // Alternative ILI9341 driver, see https://github.com
47 #define ST7735_DRIVER 取消注释 注释 Define additional parameters below for this display
48 //#define ILI9163_DRIVER // Define additional parameters below for this display
49 //#define S6D02A1_DRIVER
50 //#define RPI_ILI9486_DRIVER // 20MHz maximum SPI
51 //#define HX8357D_DRIVER
52 //#define ILI9481_DRIVER
53 //#define ILI9486_DRIVER
  
```

2、颜色顺序修改（需要看 LCD 屏驱动好显示的颜色是否有问题）

```

76 // #define TFT_RGB_ORDER TFT_RGB // Colour order Red-Green-Blue
77 #define TFT_RGB_ORDER TFT_BGR // Colour order Blue-Green-Red
  
```

3、LCD 屏的分辨率（将合适的注释掉，0.96 寸 LCD 分辨率是 160*80）

```

83 // For ST7789, ST7735, ILI9163 and GC9A01 ONLY, define the pixel width and height in portrait orientation
84 #define TFT_WIDTH 80
85 // #define TFT_WIDTH 128
86 // #define TFT_WIDTH 172 // ST7789 172 x 320
87 // #define TFT_WIDTH 170 // ST7789 170 x 320
88 // #define TFT_WIDTH 240 // ST7789 240 x 240 and 240 x 320
89 #define TFT_HEIGHT 160
90 // #define TFT_HEIGHT 128
91 // #define TFT_HEIGHT 240 // ST7789 240 x 240
92 // #define TFT_HEIGHT 320 // ST7789 240 x 320
93 // #define TFT_HEIGHT 240 // GC9A01 240 x 240
  
```

4、设置 ST7735 的主体和反色（ST7735 特有的配置选项，显示异常如颜色反色、图像镜像等，可逐条解开注释并编译查看）

```

102 // #define ST7735_INITB
103 // #define ST7735_GREENTAB
104 // #define ST7735_GREENTAB2
105 // #define ST7735_GREENTAB3
106 // #define ST7735_GREENTAB128 // For 128 x 128 display
107 #define ST7735_GREENTAB160x80 // For 160 x 80 display (BGR, inverted, 26 offset)
108 // #define ST7735_ROBOTLCD // For some RobotLCD Arduino shields (128x160, BGR,
109 // #define ST7735_REDTAB
110 // #define ST7735_BLACKTAB
111 // #define ST7735_REDTAB160x80 // For 160 x 80 display with 24 pixel offset

```

5、颜色反转（颜色颠倒就得使用）

```

116 #define TFT_INVERSION_ON
117 // #define TFT_INVERSION_OFF

```

6、配置背光控制引脚（这里配置的才生效）

```

132 #define TFT_BL 41 // LED back-light control pin
133 #define TFT_BACKLIGHT_ON HIGH // Level to turn ON back-light (HIGH or LOW)

```

7、设置 LCD 屏相关引脚（注释掉默认的引脚定义）

```

223 #define TFT_MOSI 11 // In some display driver board, it might be written as "SDA" and so on.
224 #define TFT_SCLK 12
225 #define TFT_CS 39 // Chip select control pin
226 #define TFT_DC 40 // Data Command control pin
227 #define TFT_RST 38 // Reset pin (could connect to Arduino RESET pin)
228 #define TFT_BL 41 // LED back-light

```

8、使用 HSPI 驱动（默认使用的是 VSPI，这里选择使用 HSPI）

```

374 // The ESP32 has 2 free SPI ports i.e. VSPI and HSPI, the VSPI is the default.
375 // If the VSPI port is in use and pins are not accessible (e.g. TTGO T-Beam)
376 // then uncomment the following line:
377 #define USE_HSPI_PORT

```

2.07_spi_lcd.ino 代码

在 07_spi_lcd.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"

TFT_eSPI myGLCD = TFT_eSPI(); /* 定义 TFT_eSPI 对象 myGLCD */
uint8_t x = 0; /* 刷屏颜色索引 */

/**
 * @brief 当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param 无
 * @retval 无
 */
void setup()
{
    led_init(); /* LED 初始化 */
    uart_init(0, 115200); /* 串口 0 初始化 */
    myGLCD.init(); /* LCD 初始化 */
    myGLCD.setRotation(1); /* 设置屏幕的方向 (横屏) */
}

/**
 * @brief 循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param 无
 * @retval 无
 */
void loop()
{
}

```

```

switch (x)
{
    case 0:
        myGLCD.fillRect(TFT_WHITE);
        break;
    case 1:
        myGLCD.fillRect(TFT_BLACK);
        break;
    case 2:
        myGLCD.fillRect(TFT_BLUE);
        break;
    case 3:
        myGLCD.fillRect(TFT_RED);
        break;
    case 4:
        myGLCD.fillRect(TFT_GREEN);
        break;
    case 5:
        myGLCD.fillRect(TFT_YELLOW);
        break;
    case 6:
        myGLCD.fillRect(TFT_CYAN);
        break;
    case 7:
        myGLCD.fillRect(TFT_LIGHTGREY);
        break;
}
myGLCD.setTextColor(TFT_RED, TFT_WHITE);
myGLCD.drawString("ESP32-S3", 10, 0, 2);
myGLCD.drawString("LCD Test", 10, 16, 2);
myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);
x++;

if (x == 8)
{
    x = 0;
}

LED_TOGGLE();
delay(1000);
}

```

首先，定义一个全局变量 TFT_eSPI 对象 myGLCD。

在 setup 函数中，调用 led_init 函数完成 LED 初始化，调用 uart_init 函数完成串口初始化，然后调用 myGLCD.init 函数完成 LCD 初始化，调用 myGLCD.setRotation(1)设置屏幕的方向为横屏。

在 loop 函数中，调用 myGLCD.fillRect 函数进行刷屏，调用 myGLCD.drawString 函数显示实验信息。注意：setTextColor 函数是设置字体显示的颜色以及底色。

13.4 下载验证

下载代码后，可以看到 LCD 屏不停的显示一些信息并不断切换底色。



图 13.4.1 LCD 显示效果图

第十四章 RTC 实验

本章，我们将介绍如何使用 ESP32-S3 的系统时间，实现一个简单的实时时钟。

本章分为如下 4 个小节：

14.1 RTC 介绍

14.2 硬件设计

14.3 软件设计

14.4 下载验证

14.1 RTC 介绍

RTC，Real Time Clock，实时时钟，专门用来记录时间的。

在 ESP32-S3 中，并没有像 STM32 芯片一样，具有 RTC 外设，但是存在一个系统时间，利用系统时间，也可以实现实时时钟的功能效果。

ESP32-S3 使用两种硬件时钟源建立和保持系统时间。根据应用目的及对系统时间的精度要求，既可以仅使用其中一种时钟源，也可以同时使用两种时钟源。这两种硬件时钟源为 RTC 定时器和高分辨率定时器。默认情况下，是使用这两种定时器。

在 Arduino 开发中，我们需要下载一个“ESP32Time”软件库，该库用于设置和读取 ESP32-S3 内部 RTC 时间，我们就是基于这个库进行开发 RTC 功能。这个库，可以在 Arduino IDE 中库管理搜索到，具体下载操作如下图所示。



图 14.1.1 ESP32Time 库下载过程

假如在线安装不成功，也可以尝试使用手动安装方法，跟 16.1.2 小节描述的步骤是一样的。“ESP32Time.zip”压缩包在“资料盘→6，软件资料→2，Arduino 软件包”文件夹下，下载成功后，便可以在“C:\Users\用户名\Documents\Arduino\libraries\”路径下找到该文件。

ESP32Time 库提供有三个示例工程，如下图所示：



图 14.1.2 ESP32Time 库示例工程

最快速了解库使用方法莫过于查看示例工程了，大家可以自行查看学习，下载看现象。

在 ESP32Time 库中，总的来说，分为三类函数：设置时间、获取日期时间、提取日期时间的某个数据（年/月/日/时/分/秒）。

要使用该库的函数，所以需要先定义 ESP32Time 对象实例，操作如下：

```
ESP32Time rtc;
ESP32Time rtc(offset); // create an instance with a specified offset in seconds
```

这里有两种方式，一种是不带参数，另一种是带参数的。参数 offset 的意思是时间的偏移，单位为秒。这两种有什么区别？在我的理解看来，不带参数的操作属于以自我为参照，带参数的操作属于以其他为参照。若只需要显示一个时间时，就可以使用不带参数的，即“ESP32Time rtc”；若要显示多个时间，则参考的还是使用“ESP32Time rtc”进行操作，而第二个就需要使用“ESP32Time rtc1(offset)”，offset 表明跟参考时间的关系。

以上的设计考虑，其实是因为：每个国家都会有自己的时间时，比如中国的北京时间，英国的伦敦时间，日本的东京时间。它们处于不同的时区，但是都可以以格林尼治时间(GMT)为参考，向东的时区依次为 GMT+1, GMT+2, ……, GMT+12；向西的时区依次为 GMT-1, GMT-2, ……, GMT-12。中国通用的北京时间处于 GMT+8。

打个比方，我们显示两个时区，rtc 一开始以格林尼治时间为标准进行设置，然后 rtc1 设置为北京时间。

```
ESP32Time rtc; // 格林尼治时间（伦敦时间）
ESP32Time rtc1(28800); // 北京时间，跟 rtc 相差 8 小时，8*60*60=28800 秒
```

这样定义 ESP32Time 对象实例，只需要 rtc 通过 setTime 函数设置好时间，rtc1 就会在它的基础上+8 小时，而不需要 rtc1 自己设置时间。简单理解，该功能就是可以设置相对时间。

接下来介绍一下，ESP32Time 库的函数，如下所示。

```
setTime(30, 24, 15, 17, 1, 2021); // 17th Jan 2021 15:24:30
setTime(1609459200); // 1st Jan 2021 00:00:00
setTimeStruct(time); // set with time struct

getTime() // (String) 15:24:38
getDate() // (String) Sun, Jan 17 2021
getDate(true) // (String) Sunday, January 17 2021
getDateTime() // (String) Sun, Jan 17 2021 15:24:38
getDateTime(true) // (String) Sunday, January 17 2021 15:24:38
getTimeDate() // (String) 15:24:38 Sun, Jan 17 2021
getTimeDate(true) // (String) 15:24:38 Sunday, January 17 2021

getMicros() // (unsigned long) 723546
getMillis() // (unsigned long) 723
getEpoch() // (unsigned long) 1609459200
getLocalEpoch() // (unsigned long) 1609459200
getSecond() // (int) 38 (0-59)
getMinute() // (int) 24 (0-59)
getHour() // (int) 3 (0-12)
getHour(true) // (int) 15 (0-23)
getAmPm() // (String) PM
getAmPm(true) // (String) pm
getDay() // (int) 17 (1-31)
getDayOfWeek() // (int) 0 (0-6)
getDayOfYear() // (int) 16 (0-365)
getMonth() // (int) 0 (0-11)
getYear() // (int) 2021
getTime("%A, %B %d %Y %H:%M:%S") // (String) returns time with specified format
```

通过看前面注释的说明，我们也比较清楚函数的使用了。

本例程主要用到 setTime 函数和 getTimeStruct 函数，使用方法如下：

```
rtc.setTime(00, 51, 17, 1, 12, 2023); /* 2023 年 12 月 1 日 17:52:00 */
struct tm timeinfo = rtc.getTimeStruct();
```

使用 getTimeStruct 函数，把时间信息存进 timeinfo 结构体。通过这个结构体就可以获取年、月、日、时、分、秒等信息，该结构体类型如下：

```
struct tm
{
    int tm_sec;
```

```

int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};


```

14.2 硬件设计

1. 例程功能

通过 LCD 显示模块实时显示 RTC 时间，包括年、月、日、时、分、秒等信息。LED 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43 U0RXD-IO44
- 3) LCD
CS-IO39 SCK-IO12 SDA-IO11
DC-IO40 BL-IO41 RST-IO38

3. 原理图

RTC 为 ESP32-S3 的片上资源，因此并没有相应的连接原理图。

14.3 软件设计

14.3.1 程序流程图

下面看看本实验的程序流程图：

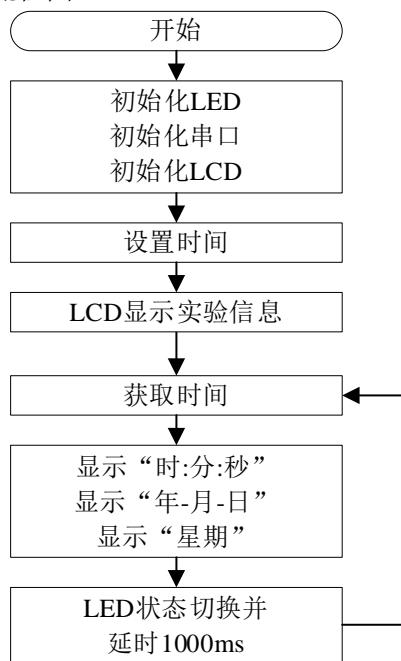


图 14.3.1 程序流程图

14.3.2 程序解析

1. 08_rtc.ino 代码

ESP32Time 库的函数已经满足了例程需求，所以没有另外写 rtc.cpp 和 rtc.h，所以这里直接介绍 08_rtc.ino 文件。

在 08_rtc.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include <ESP32Time.h>           /* 需要安装 ESP32Time 库 */

TFT_eSPI myGLCD = TFT_eSPI();      /* 定义 TFT_eSPI 对象 myGLCD */
ESP32Time rtc;
uint8_t tbuf[100];                 /* 存放 RTC 信息 */

/**
 * @brief  当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    led_init();                      /* LED 初始化 */
    uart_init(0, 115200);           /* 串口 0 初始化 */
    myGLCD.init();                  /* LCD 初始化 */
    myGLCD.setRotation(1);          /* 设置屏幕的方向 (横屏) */
    myGLCD.fillRect(TFT_WHITE);     /* 清屏 */
    rtc.setTime(00, 51, 17, 1, 12, 2023); /* 2023 年 12 月 1 日 17:52:00 */

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("RTC TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);
}

/**
 * @brief  循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param  无
 * @retval 无
 */
void loop()
{
    struct tm timeinfo = rtc.getTimeStruct();
    /* 根据 time.h 头文件中 tm 结构体的定义进行调整显示 */
    sprintf((char *)tbuf, "Time:%02d:%02d:%02d Week:%d", timeinfo.tm_hour - 1,
            timeinfo.tm_min, timeinfo.tm_sec, timeinfo.tm_wday);
    myGLCD.drawString((char *)tbuf, 0, 48, 2);
    sprintf((char *)tbuf, "Date:%04d-%02d-%02d", timeinfo.tm_year + 1900,
            timeinfo.tm_mon + 1, timeinfo.tm_mday);
    myGLCD.drawString((char *)tbuf, 0, 64, 2);

    LED_TOGGLE();
    delay(1000);
}

```

在 setup 函数中，调用 led_init 函数完成 LED 初始化，调用 uart_init 函数完成串口初始化，调用 myGLCD.init 函数完成 LCD 初始化，调用 myGLCD.setRotation(1) 设置屏幕的方向为横屏，调用 myGLCD.fillRect 函数清屏，然后调用 rtc.setTime 函数设置年月日时分秒信息，然后 LCD 显示实验信息。

在 loop 函数中，调用 rtc.getTimeStruct 函数实现时间信息的获取，通过 sprintf 函数组合成字符串，然后调用 myGLCD.drawString 函数进行显示，这个过程是间隔一秒进行。

14.4 下载验证

下载代码后，LCD 显示 RTC 实验信息，然后间隔一秒更新时间信息。



图 14.4.1 RTC 实验测试图

第十五章 INTERNAL_TEMPERATURE 实验

本章，我们将介绍 ESP32-S3 的内部温度传感器并使用它来读取温度值，然后在 LCD 模块上显示出来。

本章分为以下几个小节：

- 15.1 内部温度传感器介绍
- 15.2 硬件设计
- 15.3 软件设计
- 15.4 下载验证

15.1 内部温度传感器介绍

15.1.1 内部温度传感器简介

温度传感器生成一个随温度变化的电压，内部 ADC 将传感器电压转化为一个数字量。温度传感器的测量范围为-20°C到110°C。温度传感器适用于监测芯片内部温度的变化，该温度值会随着微控制器时钟频率或IO负载的变化而变化。一般来讲，芯片内部温度会高于外部温度。ESP32-S3 温度传感器相关内容，请看《esp32-s3_technical_reference_manual_cn.pdf》技术手册39.4 章节。

温度传感器的输出值需要使用转换公式转换成实际的温度值(°C)。转换公式如下：

$$T(^\circ\text{C}) = 0.4386 * \text{VALUE} - 27.88 * \text{offset} - 20.52$$

其中 VALUE 即温度传感器的输出值，offset 由温度偏移决定。温度传感器在不同的实际使用环境（测量温度范围）下，温度偏移不同，见下表所示。

测量范围 (°C)	温度偏移 (°C)
50 ~ 110	-2
20 ~ 100	-1
-10 ~ 80	0
-15 ~ 50	1
-20 ~ 20	2

表 15.1.1.1 温度传感器的温度偏移

15.1.2 内部温度传感器接口函数介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\tools\sdk\esp32s3\include\driver\esp32s3\include\driver\temp_sensor.h

接下来，我们介绍一下本实验所用到的相关函数。

第一个函数：temp_sensor_set_config 函数，该函数功能是设置温度传感器的配置参数。

```
esp_err_t temp_sensor_set_config (temp_sensor_config_t tsens);
```

参数 tsens 为温度传感器读取数据的配置参数，该结构体类型有两个成员：dac_offset 和 clk_div，前者用于决定温度读取范围和误差，而后者为时钟分频系数（默认为 6）。

返回值：ESP_OK 表示配置成功，其他表示配置失败。

第二个函数：temp_sensor_start 函数，该函数功能是启动温度传感器进行测量。

```
esp_err_t temp_sensor_start (void);
```

返回值：ESP_OK 表示启动成功，ESP_ERR_INVALID_ARG 表示无效参数。

第三个函数：temp_sensor_stop 函数，该函数功能是停止温度传感器测量。

```
esp_err_t temp_sensor_stop (void);
```

返回值：ESP_OK 表示获取数据成功，其他表示失败。

第四个函数：temp_sensor_read_celsius 函数，该函数功能用于读取转换为摄氏度的温度传感器数据。

```
esp_err_t temp_sensor_read_celsius(float *celsius);
```

参数 celsius 为测量后的输出值；

返回值：ESP_OK 表示读取数据成功，其他表示读取有问题。

15.2 硬件设计

1. 例程功能

通过对应接口函数读取 ESP32-S3 内部温度传感器的温度值，显示在 LCD 屏上。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11
DC-IO40
BL-IO41
RST-IO38

3. 原理图

内部温度传感器属于内部资源，因此没有对应的连接原理图。

15.3 软件设计

15.3.1 程序流程图

下面看看本实验的程序流程图：

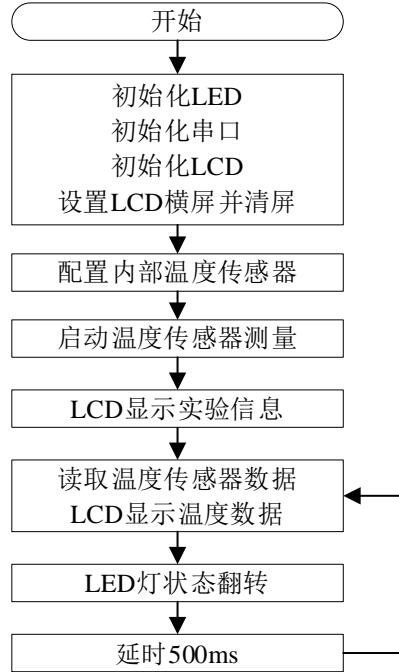


图 15.3.1 程序流程图

15.3.2 程序解析

1. 09_internal_temperature.ino 代码

在 09_internal_temperature.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include "driver/temp_sensor.h"

TFT_eSPI myGLCD = TFT_eSPI(); /* 定义 TFT_eSPI 对象 myGLCD */

/***
* @brief 当程序开始执行时, 将调用 setup() 函数, 通常用来初始化变量、函数等
* @param 无
* @retval 无
*/
void setup()
{
    led_init(); /* LED 初始化 */
    uart_init(0, 115200); /* 串口 0 初始化 */
    myGLCD.init(); /* LCD 初始化 */
    myGLCD.setRotation(1); /* 设置屏幕的方向(横屏) */
    myGLCD.fillScreen(TFT_WHITE); /* 清屏 */

    temp_sensor_config_t temp_sensor = {
        .dac_offset = TSENS_DAC_L2, /* 测量范围: -10°C ~ 80°C, 误差<1°C */
        .clk_div = 6,
    };
    temp_sensor_set_config(temp_sensor); /* 温度传感器配置 */
    temp_sensor_start(); /* 启动温度传感器, 开始温度测量 */

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("TEMP TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

    myGLCD.drawString("temperature:      °C", 10, 48, 2);
}

/***
* @brief 循环函数, 通常放程序的主体或者需要不断刷新的语句
* @param 无
* @retval 无
*/
void loop()
{
    float tsens_out;
    temp_sensor_read_celsius(&tsens_out); /* 获取当前温度 */
    // Serial.printf("%.2f\r\n", tsens_out);

    myGLCD.setTextColor(TFT_BLUE, TFT_WHITE);
    myGLCD.drawString(tsens_out, 2, 96, 48, 2);

    LED_TOGGLE();
    delay(500);
}

```

在 setup 函数中, 调用 led_init 函数完成 LED 初始化, 调用 uart_init 函数完成串口初始化, 调用 myGLCD.init 函数完成 LCD 初始化, 调用 myGLCD.setRotation(1) 设置横屏, 调用 myGLCD.fillScreen 清屏, 然后定义温度传感器的配置变量, 调用 temp_sensor_set_config 配置

温度传感器测量范围和误差，调用 `temp_sensor_start` 函数启动温度传感器进行测量，最后 LCD 显示实验信息。

在 `loop` 函数中，间隔 500 毫秒调用 `temp_sensor_read_celsius` 函数获取温度数据并在 LCD 上显示。LED 灯也是每隔 500 毫秒状态翻转，实现闪烁效果。

15.4 下载验证

将程序下载到开发板后，LCD 显示的内容如下图所示：



图 15.4.1 内部温度传感器实验测试图

大家可以看看温度值与实际是否相符合（因为芯片会发热，所以一般会比实际温度偏高）？

第十六章 SPI_SDCARD 实验

本章，我们将介绍 ESP32-S3 如何驱动 SD 卡，在 LCD 上显示 SD 卡容量，通过按键对 SD 卡进行文件操作。

本章分为如下 4 个小节：

- 16.1 SD 卡介绍
- 16.2 硬件设计
- 16.3 软件设计
- 16.4 下载验证

16.1 SD 卡介绍

16.1.1 SD 卡介绍

如果需要使用大量数据或保存大量数据，Arduino 自带的 EEPROM 和 FLASH 存储空间就显得捉襟见肘了。虽然说，模组内有 SPI FLASH 以及开发板上板载 EEPROM 芯片，但是对于大容量存储来说，还是不够用的，所以在这里推荐使用 SD 卡来解决大量数据存储的需求。

SD 卡，Secure Digital Card，称为安全数字卡。SD 卡系列主要有三种：SD 卡、Mini SD 卡和 Micro SD 卡。目前，Mini SD 卡已经被 micro SD 卡取代，micro SD 卡又被称为 TF 卡。ESP32-S3 开发板上板载的卡槽为 TF 卡，所以本例程主要就是讲解 TF 卡。

接下来，来看一下 TF 卡实物图以及引脚定义，如下图所示。

TF卡引脚定义	引脚编号	引脚名称	功能 (SDIO模式)	功能 (SPI模式)
1	Pin 1	DAT2	数据线2	保留
8	Pin 2	DAT3/CS	数据线3	片选信号
	Pin 3	CMD/MOSI	命令线	主机输出，从机输入
	Pin 4	VDD	电源	电源
	Pin 5	CLK	时钟	时钟
	Pin 6	VSS	电源地	电源地
	Pin 7	DAT0/MISO	数据线0	主机输入，从机输出
	Pin 8	DAT1	数据线1	保留

图 16.1.1.1 TF 引脚定义

TF 卡支持两种驱动方式：SPI 或 SDIO，由于引脚资源十分紧缺，所以开发板给 TF 卡设计的是 SPI 接口。使用 SPI 去驱动 TF 卡，只需要用到 4 根数据线：CS、MOSI、CLK 和 MISO。

TF 的驱动会涉及到比较多的知识，但使用 Arduino 去驱动 TF 卡会比较简单，在这里我们只介绍一下使用方法，原理性东西就不多讲了。

16.1.2 SD 卡库介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\SD\src\SD.cpp
Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\FSsrc\FS.cpp

SD.cpp 主要是 SD 卡的操作相关函数，而 FS.cpp 是提供了读/写文件功能，与串口相关函数有点相似。

在 SD.cpp 中已经定义好了 SD 对象，直接使用 SD 即可。

接下来，我们介绍一下本章节所用到的 SD 卡相关函数。

第一个函数：begin 函数，该函数功能是初始化 SD 卡。

```
bool SDFS::begin(uint8_t ssPin, SPIClass &spi, uint32_t frequency, const char *mountpoint, uint8_t max_files, bool format_if_empty)
```

参数 ssPin 为片选引脚；

参数 spi 为 SPI 对象，因为用到 SPI 接口；

参数 frequency 为时钟频率；

参数 mountpoint 为挂载点；

参数 max_files 为文件最大同时打开数；

参数 format_if_empty 为存储卡空时是否格式化；

返回值：布尔类型。初始化成功返回 true，否则返回 false。

第二个函数：cardType 函数，该函数功能是返回存储卡类型。

```
sdcard_type_t SDFS::cardType()
```

无参数；

返回值：0 为 CARD_NONE 未连接存储卡；1 为 CARD_MMC 即 MMC 卡；2 为 CARD_SD 即 SD 卡（最大 2G）；3 为 CARD_SDHC 即 SDHC 卡（最大 32G）；4 为 CARD_UNKNOWN 即未知存储卡。

第三个函数：cardSize 函数，该函数功能是返回存储卡大小字节数。

```
uint64_t SDFS::cardSize()
```

无参数；

返回值为存储卡大小的字节数。

跟 cardSize 函数相似的，还有 totalBytes 和 usedBytes 函数，前者是获取文件系统大小，后者是获取文件系统已使用的大小。

从第四个函数开始介绍文件操作函数，open 函数，该函数功能是打开 SD 卡上的一个文件。

```
File FS::open(const String& path, const char* mode, const bool create)
```

参数 path 指的是需要打开的文件命名。其中可以包含路径，路径用 “/” 分隔。

参数 mode 为打开文件的方式，FILE_READ 为只读方式打开文件，FILE_WRITE 为写入方式打开文件，FILE_APPEND 为以添加内容方式打开文件。

参数 create 不用管，可以不用传递参数。

返回值：一个 File 类型的对象，其实就是打开的文件对象。

第五个函数：close 函数，该函数功能是关闭文件并确保数据已经被完全写入到 SD 卡中。

注意：打开文件后，操作完成，最后一定需要调用 close 函数关闭文件，这样子向文件操作的内容才会被保存。

```
void File::close()
```

无参数；

无返回值。

第六个函数：write 函数，该函数功能是向文件中写入 1 字节的内容。

```
size_t File::write(uint8_t c)
```

参数 c 为写入到文件的 1 字节数据

返回值为写入的字节数。

注意：write 函数还有另外一种样式 write(const uint8_t *buf, size_t size)，其中 buf 为要写入数据的缓冲区，而 size 为要写入的长度。

除了以上两种方式向文件写入内容，还可以使用 file.print 或者 file.println，前者是输出数据到文件，后者是输出数据到文件并回车换行，具体使用方法请参考例程。

在使用写入函数前，我们要确保要写入的文件已经被打开。

第七个函数：read 函数，该函数功能是从文件中读取 1 字节数据。

```
int File::read()
```

无参数；

返回值为读取到的数据，当为-1 即没有数据可被读取。

注意：read 函数还有另外一种样式 read(uint8_t* buf, size_t size)，其中 buf 为要读取数据的缓冲区，而 size 为要读取的长度。

在使用读取函数前，我们要确保要读取的文件已经被打开。

第八个函数：available 函数，该函数的功能为检查当前文件中可读数据的字节数。

```
int File::available()
```

无参数.

返回值为可用的字节数。

例程中还会涉及到比较多的文件操作函数，大家可以自行查看。

16.2 硬件设计

1. 例程功能

程序下载完成，向 TF 卡槽插入 TF 卡，LCD 显示屏上显示 SD 卡容量大小。当按下 BOOT 键时，进行 SD 卡测试。LED 灯用来指示程序正在运行中。

2. 硬件资源

3. 原理图

TF 卡原理图，如下图所示。

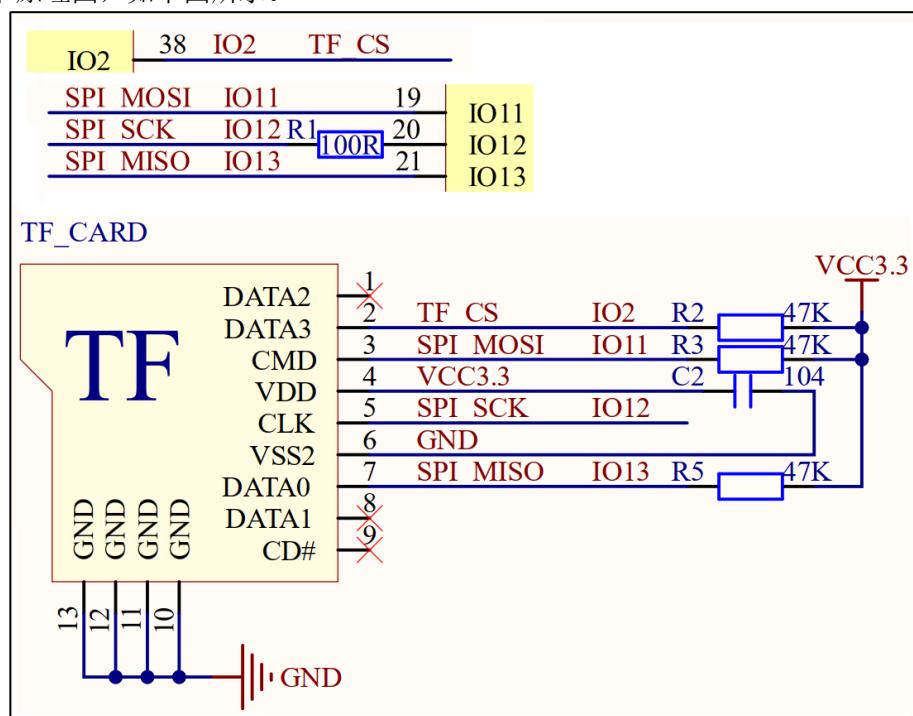


图 16.2.1 TF 卡原理图

16.3 软件设计

16.3.1 程序流程图

下面看看本实验的程序流程图：

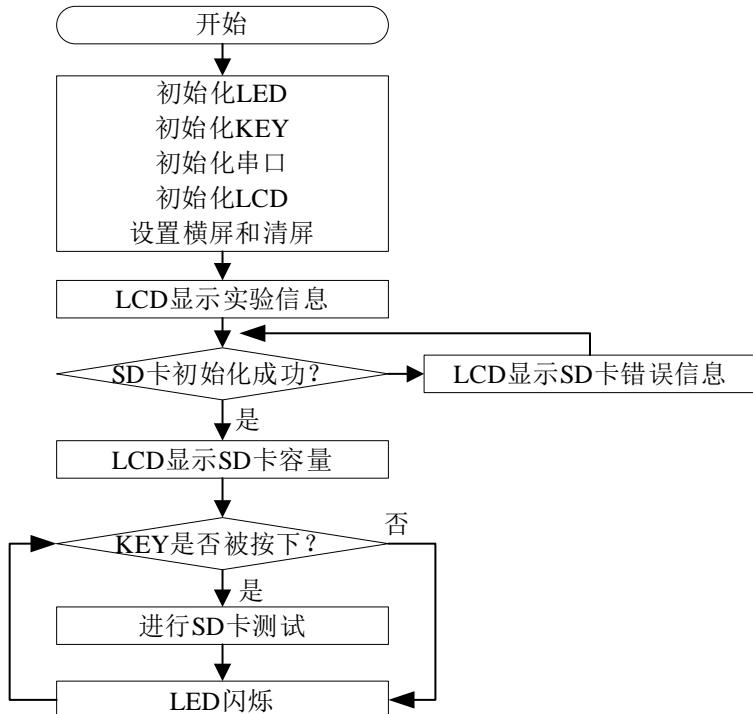


图 16.3.1 程序流程图

16.3.2 程序解析

1. SPI_SDCARD 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SPI_SDCARD 驱动源码包括两个文件：spi_sdcard.cpp、spi_sdcard.h。另外还写了文件系统操作源码 myfs.cpp 和 myfs.h，这部分代码就是对文件系统操作接口进行封装，测试使用，大家自行查看。

下面我们先解析 spi_sdcard.h 的程序。对 SD 卡相关引脚做了相关定义。

```

#define SD_CS_PIN 2
#define SD_MISO_PIN 13
#define SD_MOSI_PIN 11
#define SD_SCK_PIN 12

```

我们选择使用 IO2 作为 SD 卡的片选信号线，IO13 作为 SD 卡 SPI 接口的 MOSI 线，IO11 作为 SD 卡 SPI 接口的 SCL 线，IO12 作为 SD 卡 SPI 接口的 SCK 线。

下面我们再解析 spi_sdcard.cpp 的程序，首先先来看一下初始化函数 sdcard_init，代码如下：

```


/**
 * @brief 初始化 SD 卡
 * @param 无
 * @retval 返回值:0 初始化正确; 其他值, 初始化错误
 */
uint8_t sdcard_init(void)
{
    spi_sdcard.begin(SD_SCK_PIN, SD_MISO_PIN, SD_MOSI_PIN, SD_CS_PIN);
    /* 设置 SD 卡的 SPI 用到的引脚 */

    pinMode(SD_CS_PIN, OUTPUT); /* 片选引脚设置为输出 */
}


```

```

if (!SD.begin(SD_CS_PIN, spi_sdcard))          /* SD 卡初始化 */
{
    Serial.println("SD 卡初始化失败!");

    if (!SD.begin(SD_CS_PIN, spi_sdcard))      /* SD 卡初始化失败再次初始化 */
    {
        Serial.println("SD 卡初始化失败!");
        return 1;
    }
}
Serial.println("SD 卡初始化成功");

uint8_t cardType = SD.cardType();      /* 获取卡的类型 */
if (cardType == CARD_NONE)           /* 未连接存储卡 */
{
    Serial.println("没有卡连接");
    return 2;
}

Serial.print("SD Card Type: ");
if (cardType == CARD_MMC)           /* mmc 卡 */
{
    Serial.println("MMC");
}
else if (cardType == CARD_SD)       /* sd 卡, 最大 2G */
{
    Serial.println("SDSC");
}
else if (cardType == CARD_SDHC)     /* sdhc 卡, 最大 32G */
{
    Serial.println("SDHC");
}
else                                /* 未知存储卡 */
{
    Serial.println("UNKNOWN");
}
return 0;
}

```

在 SDCARD 初始化函数中,首先调用 spi 接口的 begin 函数初始化 SD 卡用到的 SPI 接口,然后用 SD 接口的 begin 函数初始化 SD 卡,后面就调用 cardType 函数获取存储卡的类型。

下面介绍的是 SD 卡的测试函数, 定义如下:

```

/**
 * @brief SD 卡测试代码
 * @param 无
 * @retval 无
 */
void sd_test(void)
{
    myFile = SD.open("/test.txt", FILE_WRITE);      /* 打开创建文件 */
    if (myFile)                                     /* 文件成功被打开 */
    {
        Serial.print("Writing to test.txt...");
        myFile.println("testing 1, 2, 3.");          /* 写数据到文件中 */

        myFile.close();                            /* 关闭文件 */
        Serial.println("done.");
    }
    else                                              /* 文件打开失败 */
    {
        Serial.println("error opening test.txt");
    }
}

```

```

myFile = SD.open("/test.txt");           /* 打开前面操作的文件 */
if (myFile)
{
    Serial.println("test.txt:");
    while (myFile.available())
    {
        Serial.write(myFile.read());      /* 读取文件中全部内容 */
    }
    myFile.close();                   /* 关闭文件 */
}
else                                     /* 文件打开失败 */
{
    Serial.println("error opening test.txt");
}

listDir(SD, "/");                      /* 遍历目录下的文件 */
createDir(SD, "/mydir");                /* 创建文件夹 */
listDir(SD, "/");                      /* 遍历目录下的文件 */
removeDir(SD, "/mydir");                /* 移除文件夹 */
listDir(SD, "/");                      /* 遍历目录下的文件 */
writeFile(SD, "/hello.txt", "Hello ");  /* 写数据到文件中 */
appendFile(SD, "/hello.txt", "World!\n"); /* 在文件后追加数据 */
readFile(SD, "/hello.txt");             /* 从文件中读取数据 */
deleteFile(SD, "/foo.txt");             /* 删除文件 */
renameFile(SD, "/hello.txt", "/foo.txt"); /* 修改文件名字 */
readFile(SD, "/foo.txt");               /* 从文件中读取数据 */
testFileIO(SD, "/test.txt");            /* 测试文件 IO 性能 */
Serial.printf("Total space: %lluMB\n", SD.totalBytes() / (1024 * 1024));
/* 打印 SD 卡文件系统总容量大小 */
Serial.printf("Used space: %lluMB\n", SD.usedBytes() / (1024 * 1024));
/* 打印 SD 卡文件系统已用容量大小 */
}

```

该函数创建了 test.txt 文件并进行写入数据以及读取测试，后面就是调用 myfs.cpp 文件中的一些文件操作接口进行目录遍历、创建文件夹，创建文件，删除文件的测试。

2. 10_spi_sdcard.ino 代码

在 10_spi_sdcard.ino 里面编写如下代码：

```

#include "led.h"
#include "key.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include "spi_sdcard.h"
#include <SD.h>

TFT_eSPI myGLCD = TFT_eSPI(); /* 定义 TFT_eSPI 对象 myGLCD */

/**
 * @brief 当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param 无
 * @retval 无
 */
void setup()
{
    led_init();                      /* LED 初始化 */
    key_init();                      /* KEY 初始化 */
    uart_init(0, 115200);            /* 串口 0 初始化 */
}

```

```

myGLCD.init();           /* LCD 初始化 */
myGLCD.setRotation(1);   /* 设置屏幕的方向(横屏) */
myGLCD.fillRect(TFT_WHITE); /* 清屏 */
myGLCD.setTextColor(TFT_RED, TFT_WHITE);
myGLCD.drawString("ESP32-S3", 10, 0, 2);
myGLCD.drawString("SDCARD TEST", 10, 16, 2);
myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

while (sdcard_init()) /* 检测不到 SD 卡 */
{
    myGLCD.drawString("SD Card Error!", 10, 48, 2);
    delay(500);
    myGLCD.drawString("Please Check! ", 10, 48, 2);
    delay(500);
    LED_TOGGLE(); /* 红灯闪烁 */
}

myGLCD.drawString("SD Card OK      ", 10, 48, 2);
myGLCD.drawString("SD Card Size:    MB", 10, 64, 2);
myGLCD.setTextColor(TFT_BLUE, TFT_WHITE);
myGLCD.drawString("SD cardSize() / (1024 * 1024), 13 * 8, 64, 2);
}

/***
 * @brief    循环函数, 通常放程序的主体或者需要不断刷新的语句
 * @param    无
 * @retval   无
 */
void loop()
{
    if (KEY == 0)
    {
        sd_test();
    }

    LED_TOGGLE();
    delay(500);
}

```

在 setup 函数中, 调用 led_init 函数完成 LED 初始化, 调用 key_init 函数完成 KEY 初始化, 调用 uart_init 函数完成串口初始化, 调用 myGLCD.init 函数完成 LCD 初始化, 调用 myGLCD.setRotation(1) 设置横屏, 调用 myGLCD.fillRect 清屏, 调用 sdcard_init 函数完成 SDCARD 初始化, LCD 显示实验信息和 SD 卡容量。

在 loop 函数中, 通过按下 KEY 触发 SD 卡测试。LED 灯每隔 500 毫秒状态翻转, 实现闪烁效果。

16.4 下载验证

假定 TF 卡已经接上去正确的位置, 将程序下载到开发板后, 可以看到 LED0 不停的闪烁, 提示程序已经在运行了。LCD 显示实验信息和 SD 卡容量信息如下图所示:

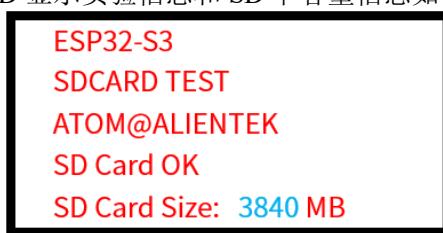


图 16.4.1 SD 卡实验测试图

然后按下 BOOT 按键进行 SD 卡测试功能，通过串口助手查看操作信息，如下图所示。

```
Writing to test.txt...done.
test.txt:
testing 1, 2, 3.
Listing directory: /
  DIR : System Volume Information
  FILE: test.txt  SIZE: 18
Creating Dir: /mydir
Dir created
Listing directory: /
  DIR : System Volume Information
  FILE: test.txt  SIZE: 18
  DIR : mydir
Removing Dir: /mydir
Dir removed
Listing directory: /
  DIR : System Volume Information
Listing directory: /System Volume Information
  FILE: IndexerVolumeGuid  SIZE: 76
  FILE: WPSettings.dat  SIZE: 12
  FILE: test.txt  SIZE: 18
Writing file: /hello.txt
File written
Appending to file: /hello.txt
Message appended
Reading file: /hello.txt
Read from file: Hello World!
Deleting file: /foo.txt
Delete failed
Renaming file /hello.txt to /foo.txt
File renamed
Reading file: /foo.txt
Read from file: Hello World!
18 bytes read for 2 ms
1048576 bytes written for 2897 ms
Total space: 3839MB
Used space: 1MB
```

图 16.4.2 SD 卡测试过程

把 SD 卡通过读卡器插入计算机进行查看，如下图所示。



图 16.4.3 SD 卡存储情况

可以看到，SD 卡的实际情况与程序执行的效果是一致的。大家也可以调用测试一下别的接口函数。

第三篇 高级篇

经过基础篇与入门篇的学习，我们已经对 ESP32-S3 的外设有了一定了解。从本篇开始，将迎来非常重要的实用的内容，那就是 WiFi 和蓝牙应用。通过本篇的学习，我们可以体会到基于 ESP32-S3 的 WiFi、蓝牙等开发是多么简单而美妙。

- 1, wifi_scan 实验
- 2, wifi_webserver 实验
- 3, wifi_client 实验
- 4, ble_scan 实验
- 5, ble_uart 实验

第十七章 WIFI_SCAN 实验

本章，我们首先介绍基本的网络互联知识为提高篇的学习奠定基础，然后介绍如何使用 ESP32-S3 把附近的网络扫描出来。

本章分为如下 4 个小节：

- 17.1 WiFi 工作模式和网络扫描介绍
- 17.2 硬件设计
- 17.3 软件设计
- 17.4 下载验证

17.1 WiFi 工作模式和网络扫描介绍

17.1.1 ESP32-S3 WiFi 工作模式

前面章节讲到，ESP32-S3 是一款集成了 WiFi+ 蓝牙功能的双模芯片，ESP32-S3 提供了 3 种 WiFi 工作模式，分别是 Station (STA)、Access Point (AP)、STA+AP。

1, STA 模式（站点模式）

当 ESP32-S3 工作于 STA 模式时，ESP32-S3 作为一个站点接入到接入点，最常见的接入点就是路由器，如下图所示。



图 17.1.1.1 STA 模式示意图

2, AP 模式（WiFi 热点模式）

当 ESP32-S3 工作于 AP 模式时，ESP32-S3 就是接入点，类似于路由器。手机和计算机都可以连接到该 ESP32-S3，如下图所示。



图 17.1.1.2 AP 模式示意图

ESP32-S3 的 AP 功能是通过软件来实现的，所以也称为 softAP。

3, STA+AP 模式

当 ESP32-S3 工作于 STA+AP 模式时，首先，作为接入点，手机等其他终端可以连接到该 ESP32-S3，同时该 ESP32-S3 还可以作为断点连接到其他接入点，如路由器，如下图所示。



图 17.1.1.3 STA+AP 模式示意图

STA+AP 模式通常适用于 MESH 网络，本书不再讲述。

17.1.2 ESP32-S3 网络扫描介绍

现在，手机连上一个 WiFi 热点，基本上都是打开手机里面的 WiFi 设置功能，然后就会看到有个 WiFi 热点列表，选择你要连接的输入密码即可。那手机怎么知道附近的 WiFi 呢？

通常，无线网络提供的 WiFi 热点，大部分都是开放了网络名称（SSID）广播，而手机打开 WiFi 设置功能，这时候就会传送到手机端即可显示出 WiFi 热点列表。WiFi 的 Scan 功能就是扫描出附近的 WiFi 热点的 SSID 信息。

扫描附近的热点在产品产测过程中也经常用到，因为工厂工序比较多，不可能每个工序都测试所有功能，通过测试扫描热点的强度可以判断模块 WiFi 性能是否符合要求。

ESP32-S3 实现 WiFi Scan 功能，首先得把模式设置为 STA 模式，然后再进行扫描网络。

一般的扫描网络需要几百毫秒才能完成。而扫描网络的过程就包括：触发扫描过程、等待完成和提供结果。

WiFi 的 Scan 库提供了两种方式实现上面的扫描过程：

- 1、同步扫描：通过单个函数在一次运行中完成，需要等待完成所有操作才能继续运行下面的操作。
- 2、异步扫描：把上面的过程分为几个步骤，每个步骤由单个函数完成。

17.1.3 WiFiScan 库介绍

本实验实现的网络扫描主要依赖的是 WiFiScan 库，还会涉及到 WiFiGeneric 库和 WiFiSTA 库。WiFiGeneric 库为 WiFi 基础功能库，而 WiFiSTA 库为 station 模式专用库。

使用 WiFiGeneric 库中的 mode 函数设置 WiFi 模式；使用 WiFiSTA 库中的 disconnect 函数断开之前的 WiFi 连接。

WiFiScan 库的函数主要分为两类：扫描操作和获取扫描结果，如下图所示。

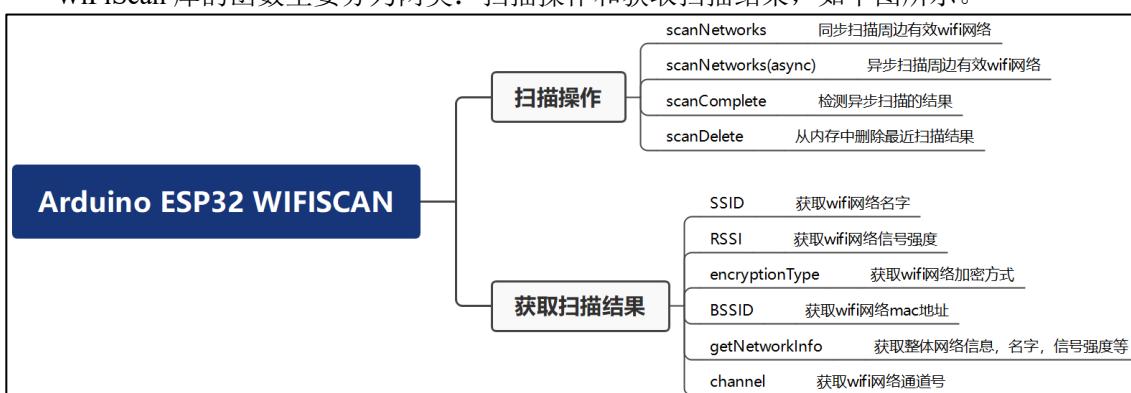


图 17.1.3.1 WIFISCAN 库介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFi.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiScan.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiGeneric.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiSTA.cpp

在 WiFi.cpp 中已经定义好了 WiFi 对象，直接使用 WiFi 即可。WiFi 对象继承了 WiFiSTAClass 对象、WiFiAPClass 对象、WiFiScanClass 对象和 WiFiGenericClass 对象。

接下来，我们介绍一下本章节所用到的函数。

第一个函数：scanNetworks 函数，该函数功能是扫描可用的 WiFi 网络。

```
int16_t WiFiScanClass::scanNetworks(bool async, bool show_hidden, bool passive,
uint32_t max_ms_per_chan, uint8_t channel, const char * ssid, const uint8_t * bssid)
```

参数 async 为是否启动异步扫描，默认是否；

参数 show_hidden 为是否扫描隐藏网络，默认否；

参数 passive 为设置主动扫描还是被动扫描，默认为主动扫描；

参数 max_ms_per_chan 为扫描超时时间，默认为 300；

参数 channel 为是否扫描特定通道，默认为 0；

参数 ssid 为是否扫描特定 SSID，默认不扫描特定 SSID；

参数 bssid 为是否扫描特定 MAC 地址，默认不扫描特定 MAC 地址；

返回值为扫描到的 WiFi 数量。

注意：使用该函数可直接不带参数，即 WiFi.scanNetworks()。

第二个函数：SSID 函数，该函数功能是获取无线网络名称。

```
String WiFiScanClass::SSID(uint8_t i)
```

参数 i 为扫描到的 WiFi 网络信息序号；

返回值为扫描到的 WiFi 网络信息中的无线网络名称。

第三个函数：RSSI 函数，该函数功能是获取无线网络信号强度，以 dBm 为单位。

```
String WiFiScanClass::RSSI(uint8_t i)
```

参数 i 为扫描到的 WiFi 网络信息序号；

返回值为扫描到的 WiFi 网络信息中的无线网络的信号强度。

第四个函数：channel 函数，该函数功能是获取无线网络名称。

```
int32_t WiFiScanClass::channel(uint8_t i)
```

参数 i 为扫描到的 WiFi 网络信息序号；

返回值为扫描到的 WiFi 网络信息中的网络信道号。。

第五个函数：encryptionType 函数，该函数功能是获取无线网络加密类型。

```
wifi_auth_mode_t WiFiScanClass::encryptionType(uint8_t i)
```

参数 i 为扫描到的 WiFi 网络信息序号；

返回值为扫描到的 WiFi 网络信息中的网络加密类型，可为：

WIFI_AUTH_OPEN 为开放网络；

WIFI_AUTH_WEP 为 WEP 加密，属于基础加密方法；

WIFI_AUTH_WPA_PSK 为 WPA_PSK 加密，比 WEP 更强大的加密算法；

WIFI_AUTH_WPA2_PSK 为 WPA2_PSK 加密，安全性很高，设置简单，普通家庭使用；

WIFI_AUTH_WPA_WPA2_PSK 为 WPA_WPA2_PSK 加密；

WIFI_AUTH_WPA2_ENTERPRISE 为 WPA2_ENTERPRISE 加密；

WIFI_AUTH_WPA3_PSK 为 WPA3_PSK 加密；

WIFI_AUTH_WPA2_WPA3_PSK 为 WPA2_WPA3_PSK 加密；

WIFI_AUTH_WAPI_PSK 为 WAPI_PSK 加密。

第六个函数：scanDelete 函数，该函数功能是删除扫描结果。注意：如果不删除，将会叠加上次扫描的结果。

```
void WiFiScanClass::scanDelete()
```

无参数；

无返回值。

第七个函数：mode 函数，该函数功能是设置 WiFi 工作模式。

```
bool WiFiGenericClass::mode(wifi_mode_t m)
```

参数 m 为 WiFi 工作模式，可选：WIFI_MODE_NULL、WIFI_MODE_STA、WIFI_MODE_AP、WIFI_MODE_APSTA 和 WIFI_MODE_MAX；

返回值：布尔类型。设置成功返回 true，否则返回 false。

第八个函数：disconnect 函数，该函数功能是断开 WiFi 连接。

```
bool WiFiSTAclass::disconnect(bool wifioff, bool eraseap)
```

参数 wifioff 为是否关闭网络功能;

参数 eraseap 为是否清空 AP 热点配置信息;

返回值: 布尔类型。返回 wlan 状态。

17.2 硬件设计

1. 例程功能

程序下载完成, 扫描附近的 WIFI 信号, 并在 LCD 显示屏上显示已经搜索到的 WIFI 数量, 通过串口输出 WIFI 的信息包含网络名字、网络强度、通道以及加密类型。

2. 硬件资源

1) LED 灯

LED-IO1

2) USART0

U0TXD-IO43 U0RXD-IO44

3) LCD

CS-IO39

SCK-IO12

SDA-IO11

DC-IO40

BL-IO41

RST-IO38

3. 原理图

本章实验使用的 WiFi 为 ESP32-S3 的片上资源, 因此并没有相应的连接原理图。

17.3 软件设计

17.3.1 程序流程图

下面看看本实验的程序流程图:

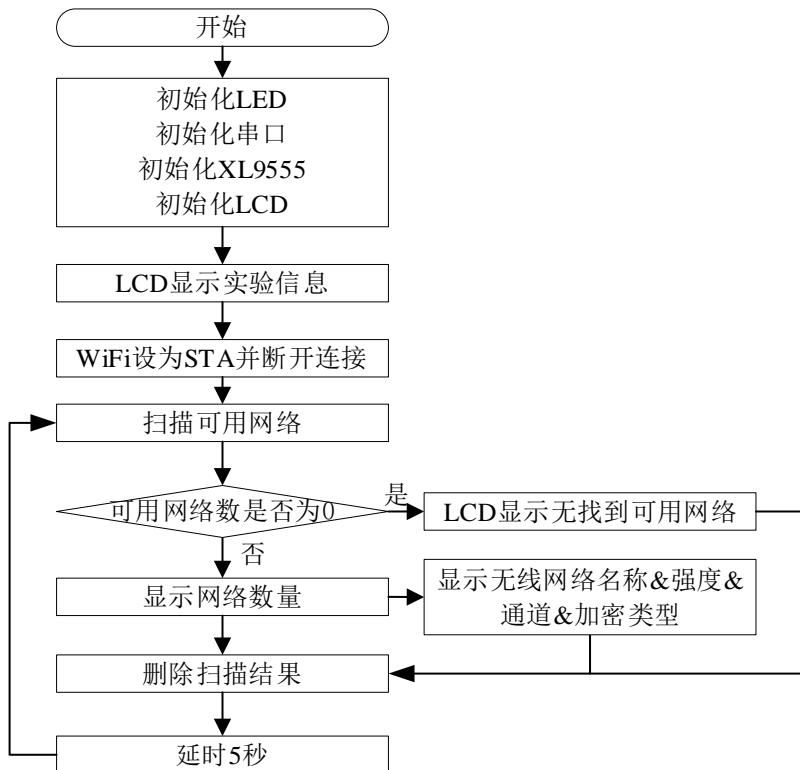


图 17.3.1.1 程序流程图

17.3.2 程序解析

1. 01_wifi_scan.ino 代码

在 01_wifi_scan.ino 里面编写如下代码：

```
#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include "WiFi.h"

TFT_eSPI myGLCD = TFT_eSPI();          /* 定义 TFT_eSPI 对象 myGLCD */

/**
 * @brief  当程序开始执行时, 将调用 setup() 函数, 通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    led_init();                         /* LED 初始化 */
    uart_init(0, 115200);              /* 串口 0 初始化 */
    myGLCD.init();                     /* LCD 初始化 */
    myGLCD.setRotation(1);             /* 设置屏幕的方向(横屏) */
    myGLCD.fillScreen(TFT_WHITE);

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("WIFI SCAN TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

    WiFi.mode(WIFI_STA);             /* WIFI 的模式选择为 STA 模式 */
    WiFi.disconnect();                /* 断开任何之前 WIFI 连接 */
    delay(100);
}

/**
 * @brief  循环函数, 通常放程序的主体或者需要不断刷新的语句
 * @param  无
 * @retval 无
 */
void loop()
{
    myGLCD.setTextColor(TFT_BLUE, TFT_WHITE);
    myGLCD.drawString("Scan start", 10, 48, 2);
    int network_cnt = WiFi.scanNetworks();          /* 开始扫描可用网络 */
    myGLCD.drawString("Scan done ", 10, 48, 2);

    if (network_cnt == 0)
    {
        myGLCD.drawString("no wifi networks found ", 10, 64, 2);
    }
    else
    {
        myGLCD.drawString("networks found", 30, 64, 2);

        Serial.println("Nr | SSID           | RSSI | CH | Encryption");
        for (int i = 0; i < network_cnt; ++i)
        {
            // Print SSID and RSSI for each network found
```

```

Serial.printf("%2d", i + 1);
Serial.print(" | ");
Serial.printf("%-32.32s", WiFi.SSID(i).c_str());
Serial.print(" | ");
Serial.printf("%4d", WiFi.RSSI(i));
Serial.print(" | ");
Serial.printf("%2d", WiFi.channel(i));
Serial.print(" | ");

switch (WiFi.encryptionType(i))
{
    case WIFI_AUTH_OPEN:
        Serial.print("open");
        break;
    case WIFI_AUTH_WEP:
        Serial.print("WEP");
        break;
    case WIFI_AUTH_WPA_PSK:
        Serial.print("WPA");
        break;
    case WIFI_AUTH_WPA2_PSK:
        Serial.print("WPA2");
        break;
    case WIFI_AUTH_WPA_WPA2_PSK:
        Serial.print("WPA+WPA2");
        break;
    case WIFI_AUTH_WPA2_ENTERPRISE:
        Serial.print("WPA2-EAP");
        break;
    case WIFI_AUTH_WPA3_PSK:
        Serial.print("WPA3");
        break;
    case WIFI_AUTH_WPA2_WPA3_PSK:
        Serial.print("WPA2+WPA3");
        break;
    case WIFI_AUTH_WAPI_PSK:
        Serial.print("WAPI");
        break;
    default:
        Serial.print("unknown");
}
Serial.println();
delay(10);
}
}
Serial.println("");

WiFi.scanDelete();
delay(5000);
myGLCD.fillRect(0, 64, 160, 16, TFT_WHITE);
}

```

在 setup 函数中，调用 led_init 函数完成 LED 初始化，调用 uart_init 函数完成串口初始化，调用 myGLCD.init 函数完成 LCD 初始化，调用 myGLCD.setRotation(1) 设置横屏，调用 myGLCD.fillRect 清屏，调用 WiFi.mode 函数设置 WiFi 为 STA 模式，然后调用 WiFi.disconnect 函数断开之前 WiFi 连接。

在 loop 函数中，每隔 5 秒通过调用 WiFi.scanNetworks 函数扫描可用网络，LCD 显示可用网络数，而通过串口把可用网络打印出来，主要调用 WiFi.SSID、WiFi.RSSI 和 WiFi.channel 和 WiFi.encryptionType 函数获取网络的名字、信号强度、通道和加密类型信息。

17.4 下载验证

程序下载成功后，我们可以看到 LCD 显示已经扫描到的无线网络数量，如下图所示：

```

ESP32-S3
WIFI SCAN TEST
ATOM@ALIENTEK
Scan done
10 networks found

```

图 17.4.1 WIFI_SCAN 实验测试图

串口输出内容，如下图所示。

输出 串口监视器 ×			
消息 (按回车将消息发送到"COM53"上的"ESP32S3 Dev Module")			
ESP-ROM: esp32s3-20210327			
Build: Mar 27 2021			
rst:0x1 (POWERON), boot:0xb (SPI_FAST_FLASH_BOOT)			
SPIWP: 0xee			
mode:DIO, clock div: 1			
load: 0x3fce3808, len: 0x44c			
load: 0x403c9700, len: 0xbe8			
load: 0x403cc700, len: 0x2a88			
entry 0x403c98d4			
Nr SSID RSSI CH Encryption			
1 ALIENTEK-YF	-50	11	WPA2+WPA3
2 ZDYZ	-66	9	WPA+WPA2
3 ChinaNet-qrNU	-69	9	WPA+WPA2
4 ALIENTEK-7仓库	-72	13	WPA2
5 ALIENTEK-HYS	-74	4	WPA2
6 zxkj	-74	6	WPA+WPA2
7 ALIENTEK-JS	-79	6	WPA2+WPA3
8 ChinaNet-ZuyC	-81	6	WPA+WPA2
9 EZVIZ_E32800871	-83	3	WPA2
10 DIRECT-E5-HP Smart Tank 530	-83	6	WPA2

图 17.4.2 扫描到的无线网络

第十八章 WIFI_WEB SERVER 实验

本章，我们将介绍 ESP32-S3 通过 WiFi 联网，在 ESP32-S3 上搭建 Web 服务器，然后访问 Web 服务器控制 ESP32-S3 开发板的 LED 灯。

本章分为如下 5 个小节：

- 18.1 网络基础知识和 WEB SERVER 函数介绍
- 18.2 HTML 基础
- 18.3 硬件设计
- 18.4 软件设计
- 18.5 下载验证

18.1 网络基础知识和 WEB SERVER 函数介绍

18.1.1 互联网络和 TCP/IP 协议

在世界各地，各种各样的计算机运行着不同的操作系统为大家服务，这些计算机在表达同一种信息时使用的方法也是千差万别，就好像语言不通会为人们合作带来障碍一样，计算机使用者意识到，使用单台计算机并不能发挥太大的作用，只有把它们连接起来，才能发挥出计算机的巨大潜力。于是人们就想法设法用线路将计算机连接到一起，这就构成了互联网络，也就是常说的 Internet。

通过线路简单将计算机连接到一起是远远不够的，就好像语言不通的两个人见了面，不能完全交流信息，因此需要定义一些共同的规则以方便交流，TCP/IP 就是为此而生的。TCP/IP 是一个协议家族的统称，除了包含了 IP 协议、TCP 协议，及我们比较熟悉的 HTTP、FTP、POP3 协议等。计算机有了这些协议，就好像学会了外语，就可以和其他的计算机进行信息传输和数据交流了。

TCP/IP 协议家族的协议成员是分层次的，我们称之为 TCP/IP 模型，共分为 4 层，从底向上依次是网络接口层、网络层、传输层和应用层，如下图所示。

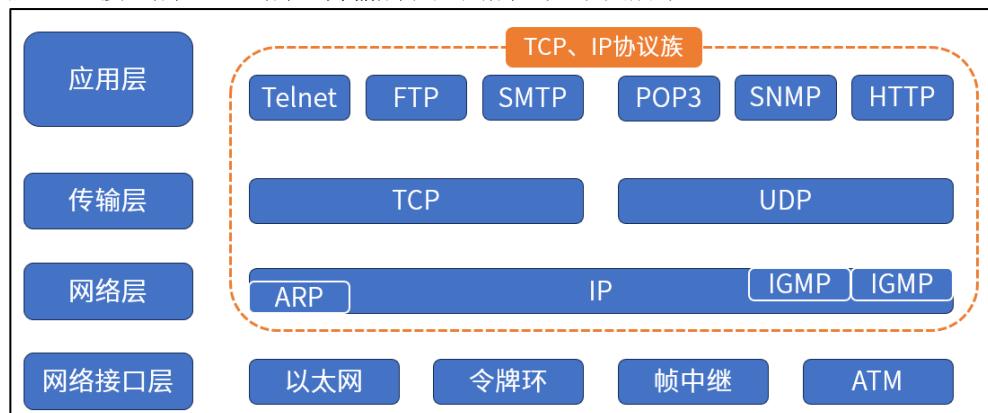


图 18.1.1.1 TCP/IP 模型

18.1.2 IP 地址

IP 地址指互联网协议地址(Internet Protocol Address)，是 IP 协议提供的一种统一地址格式，它为互联网上每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽各种不同类型计算机物理地址的差异。

举个例子，如果要写信给一个人，那就要知道他的地址，这样邮递员才能将信件送到。IP 地址相当于计算机在网络上的“门牌号”，只有通过 IP 地址才能找到对方的计算机或服务器，如下图所示。



图 18.1.2.1 IP 地址

IP 地址的定义不止一个版本，目前的主流版本为 IPv4。IPv4 地址的长度为 32 位，分为 4 段，每段 8 位，用十进制数字表示，每段数字范围为 0 ~ 255，段与段之间用句点隔开，如 202.102.10.1。可见，IPv4 共有 2^{32} 个，接近 43 亿个 IP 地址。在互联网发展的初期这似乎是个很大的数量，但随着互联网的发展，尤其移动互联网的发展，越来越多的服务器和终端（包括手机）连入互联网，即使只满足全球 70 亿人口的手机联网，IP 地址数量就已经不够使用了，更何况现在是物联网时代，万物互联，各种各样的传感器都会联网，此时连入互联网的终端数量将比人口数量至少高 1000 倍。为了满足这种需求，国际标准组织又提出了 IPv6 标准，IPv6 的地址长度为 128 位，其地址数量号称“可以为全世界的每一粒沙子拥有一个 IP 地址”，足以支撑未来相当长时间的发展需要。目前，IPv4 正在向 IPv6 应用的过渡过程中。

如果你的计算机通过路由器连接到互联网，那么路由器会分配一个 IP 地址给计算机。单击快捷键“win+R”，在弹出对话框的文本框中输入 cmd 命令后回车，则可以打开 Windows 命令行界面。在提示符下，输入 ipconfig 命令可以查看当前计算机的 IP 配置，如下图所示。

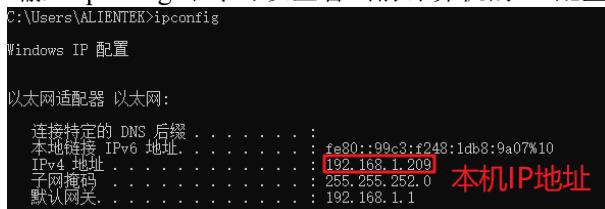


图 18.1.2.2 查看本机 IP 地址

18.1.3 端口号

我们知道，一台拥有 IP 地址的主机可以提供许多服务，比如网页浏览服务、文件传送服务、邮件服务等，那么，主机如何区分不同的网络服务呢？显然不能只靠 IP 地址，因为 IP 地址与网络服务的关系是一对多的关系，现实中是通过“IP 地址+端口号”来区分不同的服务。

常用端口号（Port）对应的应用列表如下表所示。

端口号	基于协议	描述
20	TCP	文件传输协议（FTP），数据端口
21	TCP	文件传输协议（FTP），控制端口
22	TCP	远程登录协议（SSH），用于安全登录及文件传送等
23	TCP	终端仿真协议（Telnet），用于未加密文本通信
25	TCP	简单邮件传输协议（SMTP），用于邮件服务器间的电子邮件传递
53	UDP	域名解析协议（DNS），用于域名解析
80	TCP	超文本传输协议（HTTP），用于传输网页，即 Web 服务
110	TCP	邮局协议第 3 版（POP3），用于接收电子邮件
443	TCP	安全的超文本传输协议（HTTPS），用于加密 HTTP 传输

表 18.1.3.1 常用端口号表

便于理解“IP 地址+端口号”来区分不同服务，可以把一个 IP 地址看作一个医院，医院有内科、外科、牙科、皮肤科等科室。挂号时，会根据你的症状安排到不同的科室进行医治，这些科室就和端口号类似，不同的端口号就对应着不同的网络服务（Web 服务、FTP 服务等）。

18.1.4 客户端-服务器模式

互联网络把计算机连起来的目的主要是向计算机提供信息服务，互联的计算机通常可分为

两类：一类是信息服务的请求者（Requestor），被称为客户端（Client）；另外一类是信息服务的响应者（Responor），被称为服务器（Server）。这种主要由客户端和服务器组成的网络架构称为客户端-服务器模式（简称 C/S 模式），如下图所示。

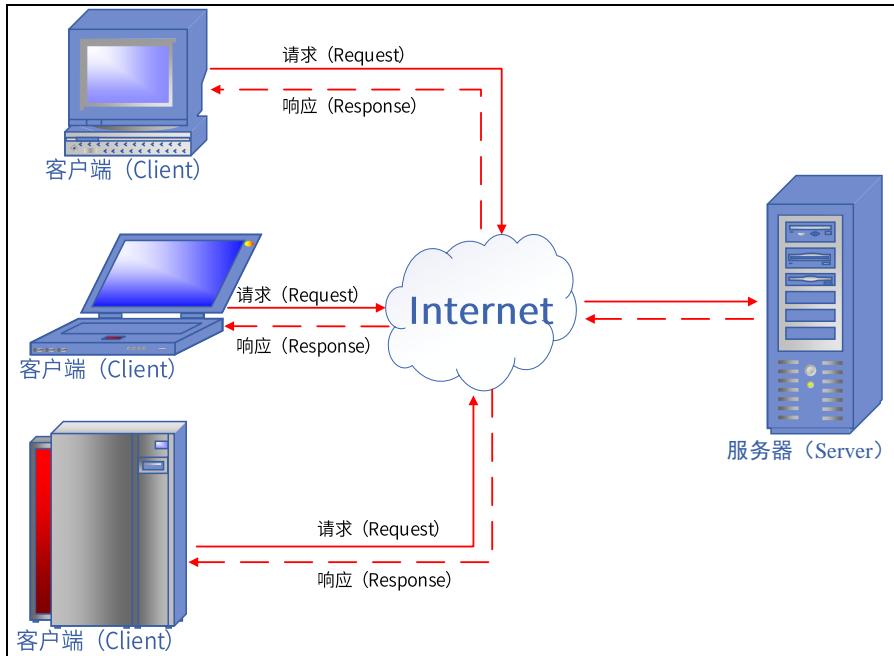


图 18.1.4.1 客户端-服务器模式

在计算机的浏览器输入 www.baidu.com 访问网站，此时计算机上的浏览器就是客户端，而百度网站的计算机和数据库则是服务器。当网页浏览器向百度发送一个查询请求时，百度服务器从百度的数据库中找出该请求所对应的信息，组成一个网页，再发送回浏览器。

18.1.5 HTTP 协议

协议是指计算机通信网络中两台计算机之间进行通信所必须共同遵守的规定或规则。

HTTP 协议是 Hyper Text Transfer Protocol（超文本传输协议）的缩写，是从万维网（WWW: World Wide Web）服务器传输超文本到本地浏览器的传送协议。超文本是指用超链接的方法将各种不同空间的文字信息组织在一起而形成的网状文本。这里的文字信息包含文字、图片、音频、视频、文件等数据。

HTTP 协议是基于客户端/服务端（C/S）的架构模型。

HTTP 客户端一般是一个应用程序（比如 Web 浏览器），通过连接到服务器达到向服务器发送一个或多个 HTTP 的请求的目的。

HTTP 服务器同样也是一个应用程序，通过是一个 Web 服务器程序，如微软 IIS 服务器，服务器通过接收客户端的请求并向客户端发送 HTTP 响应数据。

HTTP 的访问由客户端发起，通过一种叫统一资源定位符（URL: Uniform Resource Locator，比如 www.baidu.com/duty）的标识来找到服务器，建立连接并传输数据。

HTTP 默认端口号为 80。

18.1.6 ESP32-S3 Web 服务器

HTTP 服务器也称为 Web 服务器，用于接收客户端 HTTP 请求消息并回复响应消息，是一个运行在硬件服务器的软件。

ESP32-S3 Web 服务器就是一个运行在 ESP32-S3 硬件平台上的 Web 服务器程序。自底向上与 TCP/IP 的 4 层模型有着对应关系。

客户端与 Web 服务器通过 HTTP 协议进行交互，如下图所示。

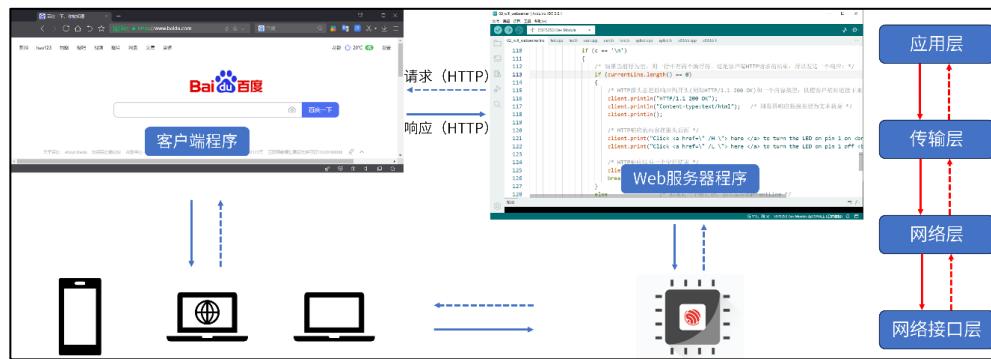


图 18.1.6.1 客户端与 Web 服务器通过 HTTP 交互

18.1.7 URL 和域名、IP 之间的关系

<http://www.baidu.com/duty/> 是 URL; www.baidu.com 就是域名; 183.2.172.42 就是 IP 地址。在计算机的浏览器输入 <http://www.baidu.com/duty/> (URL), 其中, http 为传输协议; www.baidu.com 为域名, 计算机会通过域名解析系统 (DNS: Domain name resolution), 把 www.baidu.com (域名) 解析成 183.2.172.42 (IP 地址), 然后和 183.2.172.42 建立连接, 并告诉 183.2.172.42: “我要看/duty (资源路径下) 网页的内容”。

之所以在 IP 地址之外还引入域名, 主要是为了帮助记忆, IP 地址作为一个抽象的数字不容易记住, 而 www.baidu.com 则更容易记住多了。

18.1.8 ESP32-S3 WEB SERVER 介绍

本实验, 开发板为 Web 服务器。实现 Web 服务器主要依赖 WIFISTA 和 WIFISERVER 库。

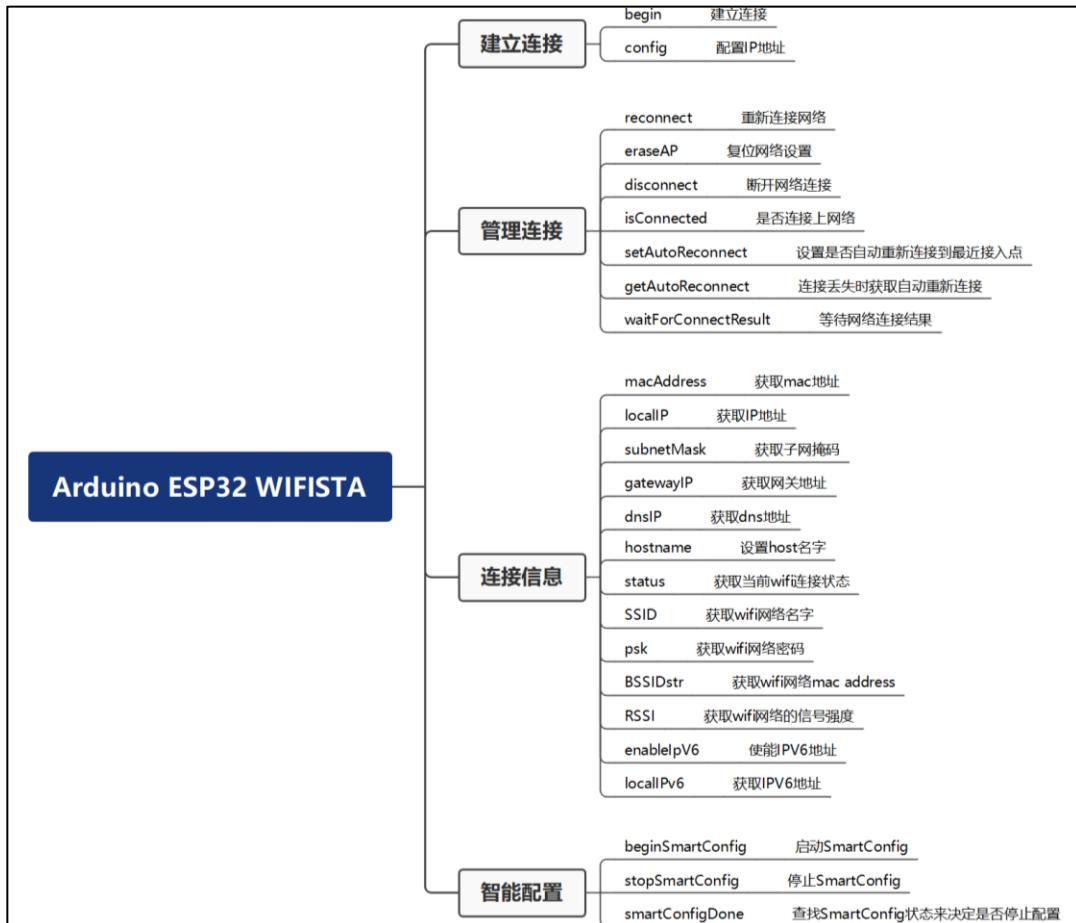


图 18.1.8.1 WIFISTA 库介绍

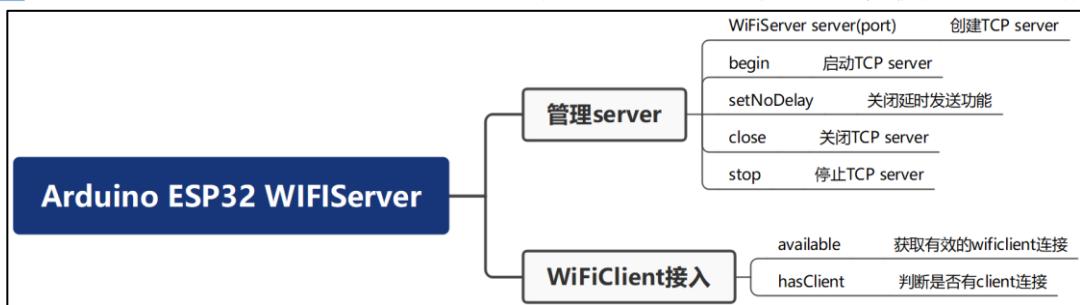


图 18.1.8.2 WiFiServer 库介绍

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiSTA.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiServer.cpp

接下来，我们介绍一下本章节所用到的 WEB SERVER 相关函数。

第一个函数：begin 函数，该函数功能是启动 ESP32-S3 的工作模式。

```
Wl_status_t WiFiSTAClass::begin(char* ssid, char* passphrase, int32_t channel,
const uint8_t* bssid, bool connect)
```

参数 ssid 为要连接的 WiFi 名称；

参数 passphrase 为要连接的 WiFi 密码；

参数 channel 为连接的 WiFi 接入点信道，属于可选参数；

参数 bssid 为连接 WiFi 接入点的 mac 地址，属于可选参数；

参数 connect 为是否连接该 WiFi，属于可选参数；

返回值：wl_status_t 类型。WL_IDLE_STATUS (0) 表示 WIFI 模块处于空闲状态，没有与任何网络连接；WL_NO_SSID_AVAIL (1) 表示找不到指定的 WiFi 网络或者附近没有可用的 WiFi 网络；WL_SCAN_COMPLETED (2) 表示 WiFi 扫描已完成，但尚未连接到任何网络。WL_CONNECTED (3) 表示 WiFi 连接成功；WL_CONNECT_FAILED (4) 表示 WiFi 连接失败，WL_CONNECTION_LOST (5) 表示 WiFi 连接丢失；WL_DISCONNECTED (6) 表示 WiFi 未连接。

第二个函数：status 函数，该函数功能是返回联网状态。

```
Wl_status_t WiFiSTAClass::status()
```

无参数；

返回值：wl_status_t 类型。具体返回值含义查看上一函数返回值说明。

第三个函数：localIP 函数，该函数功能是获取当前 ESP32-S3 的 IP 地址。

```
IPAddress WiFiSTAClass::localIP()
```

无参数；

返回值为 IPAddress 类型，即当前 WiFi 终端分配的 IP 地址。

第四个函数：begin 函数，该函数不同于第一个函数，本函数的功能是用于启动 ESP32-S3 所建立的物联网网络服务器。

```
void WiFiServer::begin(uint16_t port)
```

参数 port 为服务器使用的端口号；

无返回值。

第五个函数：available 函数，本函数的功能是侦听是否有 Client 接入。

```
WiFiClient WiFiServer::available()
```

无参数；

返回值为接入网络的 WiFiClient 对象。

第六个函数：connected 函数，本函数的功能是判断 Client 是否成功连接服务器。

```
uint8_t WiFiClient::connected()
```

无参数；

返回值为 int 类型，0 表示连接失败，1 表示连接成功。

第七个函数：available 函数，本函数的功能是客户端接收缓冲区中的字节数。

```
int WiFiClient::available()
```

无参数；

返回值为 int 类型，返回缓冲区中的字节数。

第八个函数: `read` 函数, 本函数的功能是从客户端接收缓冲区中读取数据。

```
int WiFiClient::read()
```

无参数;

返回值为缓冲区第一个字节数据。

第九个函数: `stop` 函数, 本函数的功能是客户端断开连接。

```
void WiFiClient::stop()
```

无参数;

无返回值。

18.2 HTML 基础

HTML (Hyper Text Markup Language), 即超文本标记语言, 是用于创建网页的主要标记语言。浏览网页时, 每一个网页对应一个 HTML 文档。Web 浏览器是为了解释 HTML 文件而生的, HTML 文件中的标签告诉 Web 浏览器如何在页面上显示内容。HTML 文档的后缀为.html 或者.htm, 这两种后缀都一样, 没有区别。

本节将通过编写简单的 HTML 文档来了解 HTML 文档的基本结构。

HTML 文档是纯文本格式, 可以使用电脑自带的记事本来编辑 HTML 文档, 当然也可以使用 VScode 或者 notepad++ 等等。此外, 微软的 FrontPage 和 Adobe 公司的 Dreamweaver, 是“所见即所得”的可视化网页制作软件, 读者可自行选择。

18.2.1 HTML 文档基本结构

HTML 文档的基本结构如下图所示。

```
<html>
  <head>
    <title> 页面标题 </title>
  </head>

  <body>
    <h1> 这是一个标题 </h1>
    <p> 这是一个段落 </p>
    <p> 这是另一个段落 </p>
  </body>
<html>
```

图 18.2.1.1 HTML 文档结构图示

上图中, 不同的“<>”及其包围的关键词称为 HTML 标记标签, 简称为 HTML 标签, 如 `<html>`、`<head>`、`<title>`、`<body>` 等。需要注意: 绝大多数标签成对出现。

我们手动输入的内容称为文本, 文本和 HTML 标签构成了 HTML 文档。在 HTML 文件中, 使用“`<!--内容-->`”这个格式作为注释。

HTML 文档的基本结构总结如下:

HTML 文档由 HTML 标签以及文本内容组成。

HTML 标签是由尖括号及包围的关键词组成, 比如 `<html>`、`<head>` 等。

HTML 标签通常是成对出现的, 比如 `<p>` 和 `</p>`, 标签对中的第一个标签是开始标签, 第二个标签是结束标签。结束标签比开始标签多一个斜杠“/”。

`<!--内容-->` 为注释标签, 用于在 HTML 插入注释。

`<html>` `</html>` 标签标识网页的开始和结尾, 网页的结构基于两者之间。

HTML 文档分为两个主要部分: 头部和主体。`<head>` 和 `</head>` 标记头部, `<body>` 和 `</body>` 标记主体。头部包含有关 HTML 文档的属性数据, 这些数据对最终用户不可见, 但会向网页添加标题、脚本、样式甚至更多, 这称为元数据。主体是包括类似文本、按钮、表格等页面内容。

基于文档的基本结构, 写了一个简单的页面, 代码如下:

```

<!DOCTYPE html>
<html>  <!--html 文档开始-->
<head>  <!--头部开始-->
    <title>ATK ESP32-S3 Web Server</title>
</head> <!--头部结束-->

<body>  <!--主体页面开始-->
    <h1> ESP32-S3 Web Server </h1>
    <p> LED State </p>
    <p> <button> ON </button> </p>
    <p> <button> OFF </button> </p>
</body> <!--主体页面结束-->
</html> <!--html 文档结束-->

```

网页显示效果如下图所示。

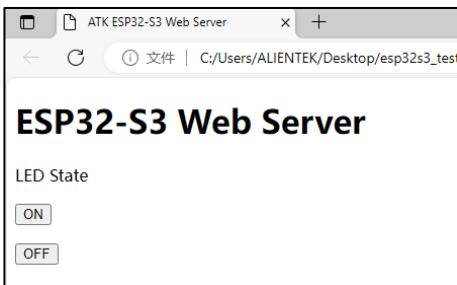


图 18.2.1.2 esp32s3_test.html 网页显示效果

18.2.2 HTML 标签

常用的 HTML 标签如下表所示。

标记	作用	用法	放置位置
<title>	页面标题，用于显示浏览器选项卡中的文本	<title>页面标题名</title>	头部
<h1>~<h6>	文本标题，以“h”开头，后面跟着表示标题等级的数字，数字越大，字体越小	<h1>ESP32-S3 Web Server</h1>	主体
<p>	段落，用于放置文本	<p>段落文本</p>	主体
<button>	按钮	<button>按钮文本</button>	主体
<a>	超链接，用于向文本、图像、按钮类 HTML 元素添加超链接	链接文本	主体
<meta>	元数据，不会显示在页面上，向浏览器提供如何显示内容的有用信息，让页面适应不同类型的 Web 浏览器	<meta charset="UTF-8" http-equiv="refresh" content="2">	头部
 	插入一个简单的换行符	 	主体

表 18.2.2.1 常用 HTML 标签

18.3 硬件设计

1. 例程功能

程序下载完成，ESP32-S3 尝试连接程序设定的 WIFI 网络，连接上后 LCD 显示 ESP32-S3 的 IP。终端浏览器（计算机、手机）也要接入一样的 WIFI 网络，在浏览器中输入 ESP32-S3 的 IP 地址，访问 ESP32-S3 Web 服务器。单击 Web 网页上的按钮，以无线的方式打开或关闭 ESP32-S3 板载的 LED 灯。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11
DC-IO40
BL-IO41
RST-IO38

3. 原理图

本章实验使用的 WiFi 为 ESP32-S3 的片上资源，因此并没有相应的连接原理图。

18.4 软件设计

18.4.1 程序流程图

下面看看本实验的程序流程图：

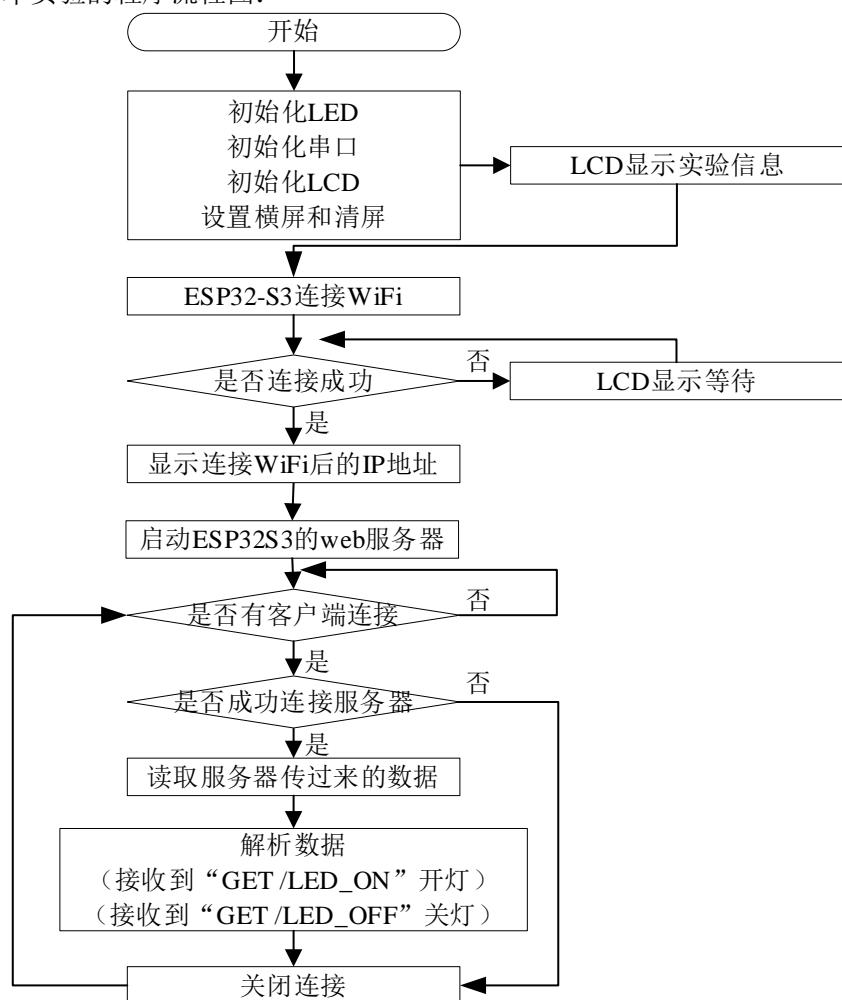


图 18.4.1.1 程序流程图

18.4.2 程序解析

1. 02_wifi_webserver.ino 代码

在 02_wifi_webserver.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include "WiFi.h"

TFT_eSPI myGLCD = TFT_eSPI();      /* 定义 TFT_eSPI 对象 myGLCD */
char* ssid      = "ALIENTEK-YF";   /* 要连接网络名称 */
char* password = "15902020353";    /* 要连接网络密码 */
WiFiServer server(80);           /* 定义 Web 服务器对象实例，端口默认为 80 */
WiFiClient client;                /* 定义客户端对象 */
void webpage_display(void);       /* 网页显示内容 */

/**
 * @brief  当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    char ip_buf[20];

    led_init();                      /* LED 初始化 */
    uart_init(0, 115200);            /* 串口 0 初始化 */
    myGLCD.init();                  /* LCD 初始化 */
    myGLCD.setRotation(1);          /* 设置屏幕的方向 (横屏) */
    myGLCD.fillScreen(TFT_WHITE);

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("WIFI WEBSERVER TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

    WiFi.begin(ssid, password); /* 连接网络 */
    myGLCD.drawString("WIFI_NAME:", 0, 48, 2);
    myGLCD.drawString(ssid, 80, 48, 2);

    while (WiFi.status() != WL_CONNECTED) /* 等待网络连接成功 */
    {
        delay(500);
        myGLCD.drawChar('.', 10 + 8 * i, 64, 2);
        i++;
    }
    myGLCD.fillRect(0, 48, 160, 16, TFT_WHITE);
    myGLCD.drawString("WiFi connected.", 0, 48, 2);
    myGLCD.drawString("IP address:", 0, 64, 2);

    sprintf(ip_buf, "%s", WiFi.localIP().toString().c_str());
    myGLCD.drawString(ip_buf, 68, 64, 2); /* 显示连接 wifi 后的 ip */
}

server.begin(); /* 启动 ESP32S3 所建立的物联网网络服务器 */
}

/**
 * @brief  循环函数，通常放程序的主体或者需要不断刷新的语句
 * @param  无
 */

```

```
* @retval 无
*/
void loop()
{
    client = server.available();      /* 检测服务器端是否有活动的客户端连接 */

    if (client)                      /* 有客户端连接 */
    {
        String currentLine = "";      /* 创建一个字符串来保存来自客户端的传入数据 */
        while (client.connected())    /* 检查设备是否成功连接服务器 */
        {
            // Serial.println("Client is connected");
            if (client.available())    /* 检查网络客户端是否有接收到服务器发来的信息 */
            {
                char c = client.read(); /* 读取服务器发来的信息 */
                //Serial.write(c);
                if (c == '\n')          /* 用于判断接收到得网页访问数据是否结束 */
                {
                    /* 如果当前行为空，则一行中有两个换行符。这是客户端 HTTP 请求的结束，所以发送一个响应： */
                    if (currentLine.length() == 0)
                    {
                        webpage_display();
                        break;
                    }
                    else                  /* 如果有一个换行符，那么清除 currentLine */
                    {
                        currentLine = "";
                    }
                }
                else if (c != '\r')      /* 如果没有回车符 */
                {
                    currentLine += c; /* 将它添加到 currentLine 的末尾 */
                }
            }

            /* 检查客户端请求是"GET /H"还是"GET /L" */
            if (currentLine.endsWith("GET /LED_ON"))
            {
                LED(0); /* GET /H 打开灯 */
                Serial.println("LED ON");
            }

            if (currentLine.endsWith("GET /LED_OFF"))
            {
                LED(1); /* GET /L 关闭灯 */
                Serial.println("LED OFF");
            }
        }
        client.stop(); /* 关闭连接 */
    }
}

/**
 * @brief 网页数据内容
 * @param 无
 * @retval 无
 */
void webpage_display(void)
{
    /* HTTP 报头总是以响应码开头 (例如 HTTP/1.1 200 OK) 和一个内容类型，以便客户端知道接下来
    是什么，然后是一个空行： */
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html"); /* 浏览器响应数据类型为文本数据 */
}
```

```

client.println();

client.print("<!DOCTYPE html>");
client.print("<html>");
client.print("<head>");
client.print("<title>ATK ESP32-S3 Web Server</title>");
client.print("</head>");

client.print("<body>");
client.print("<h1> ESP32-S3 Web Server </h1>");
client.print("<p> LED State </p>");
client.print("<p> <a href=\"/LED_ON\"> <button> ON </button> </p>"); 
client.print("<p> <a href=\"/LED_OFF\"> <button> OFF </button> </p>"); 
client.print("</body>"); 
client.print("</html>");

/* HTTP 响应以另一个空行结束 */
client.println();
}

```

首先定义 WiFiServer 服务器对象实例为 server，服务器端口号为 80。继续定义 WiFiClient 客户端对象实例为 client，用于保存侦听登录到 server 的客户端对象。

在 setup 函数中，调用 led_init 函数完成 LED 初始化，调用 key_init 函数完成 KEY 初始化，调用 uart_init 函数完成串口初始化，调用 myGLCD.init 函数完成 LCD 初始化，调用 myGLCD.setRotation(1) 设置横屏，调用 myGLCD.fillRect 清屏，调用 WiFi.begin 函数连接网络，最后通过 server.begin 函数启动 web 服务器。

在 loop 函数中，通过 server.available 函数侦听是否有客户端连接，若有连接即浏览器登录该 Web 服务器，这就相当于客户端发送给 Web 服务器的 HTTP 请求。HTTP 请求的内容是固定格式的多行文本并以空行结束，就是 webpage_display 函数。

18.5 下载验证

程序下载成功后，我们可以看到 LCD 显示已经连接上 WiFi，显示当前 ESP32-S3 的 IP，如下图所示：



图 18.5.1 WiFi WEB SERVER 实验测试图

将移动端设备，比如手机也连接该 WiFi，然后在浏览器中输入 ESP32-S3 的 IP“192.168.1.247”回车访问网页，需要稍等片刻，如下图所示。



图 18.5.2 网页效果图

通过该网页的 ON 和 OFF 按钮可控制开发板上的 LED 灯。当我们按下 ON 按钮时，ESP32-S3 开发板上的 LED 亮起。

第十九章 WIFI_CLIENT 实验

在上一章中，我们使用手机或者计算机作为客户端，在浏览器中访问 Web 服务器。在本章，我们将介绍 ESP32-S3 作为客户端连接服务器获取数据。

本章分为如下 4 个小节：

19.1 CLIENT 函数介绍

19.2 硬件设计

19.3 软件设计

19.4 下载验证

19.1 CLIENT 函数介绍

在本实验，ESP32-S3 开发板主要作为一个客户端。

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\WiFi\src\WiFiMulti.cpp

接下来，我们介绍一下本章节所用到的 CLIENT 相关函数。

第一个函数：addAP 函数，该函数功能是添加 WiFi 连接信息。

```
bool WiFiMulti::addAP(const char* ssid, const char* passphrase)
```

参数 ssid 为需要添加的 WiFi 网络名称；

参数 passphrase 为需要添加的 WiFi 网络密码；

返回值为 bool 类型，false 表示连接失败，true 表示连接成功。

第二个函数：run 函数，该函数功能是连接网络，前面需要使用 addAP 函数为 ESP32-S3 开发板存储多个 WiFi 网络后。

```
uint8_t WiFiMulti::run(uint32_t connectTimeout)
```

参数 connectTimeout 为连接网络的超时时间；

返回值：WiFi 连接状态。

第三个函数：connect 函数，本函数的功能是成功连接服务器。

```
int WiFiClient::connect(IPAddress ip, uint16_t port)
```

参数 ip 为所要连接服务器的地址；

参数 port 为所要连接服务器端口号；

返回值为 int 类型，0 表示连接失败，1 表示连接成功。

19.2 硬件设计

1. 例程功能

程序下载完成，ESP32-S3 尝试连接程序设定的 WIFI 网络，连接上后 LCD 显示 ESP32-S3 的 IP。ESP32-S3 继续尝试连接上一章的作为 WEB SERVER 的 ESP32-S3 开发板，然后向服务器请求数据，把数据显示出来，然后关闭连接，重复这个过程。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11

DC-IO40
BL-IO41
RST-IO38

3. 原理图

本章实验使用的 WiFi 为 ESP32-S3 的片上资源，因此并没有相应的连接原理图。

19.3 软件设计

19.3.1 程序流程图

下面看看本实验的程序流程图：

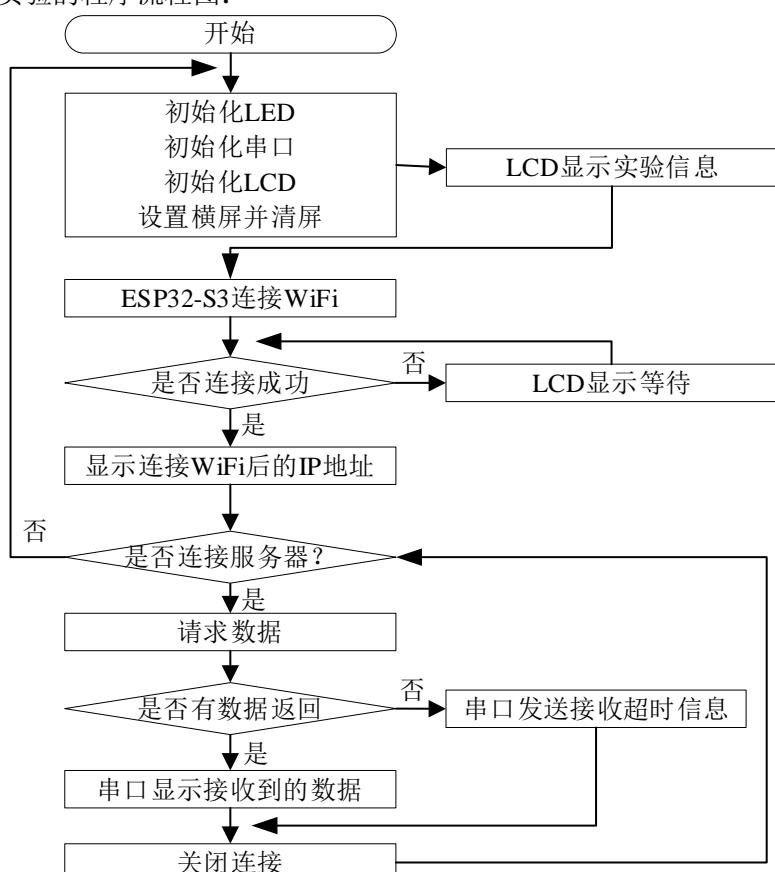


图 29.3.1 程序流程图

19.3.2 程序解析

1. 03_wifi_client.ino 代码

在 03_wifi_client.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include <WiFiMulti.h>

TFT_eSPI myGLCD = TFT_eSPI();      /* 定义 TFT_eSPI 对象 myGLCD */
WiFiMulti WiFiMulti;

char* ssid      = "ALIENTEK-YF";    /* 要连接网络名称 */
  
```

```
char* password = "15902020353";      /* 要连接网络密码 */

/***
 * @brief  当程序开始执行时, 将调用 setup() 函数, 通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */
void setup()
{
    char ip_buf[20];

    led_init();                      /* LED 初始化 */
    uart_init(0, 115200);           /* 串口 0 初始化 */
    myGLCD.init();                  /* LCD 初始化 */
    myGLCD.setRotation(1);          /* 设置屏幕的方向(横屏) */
    myGLCD.fillScreen(TFT_WHITE);

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("WIFI CLIENT TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

    WiFiMulti.addAP(ssid, password);      /* 开始连接网络 */
    myGLCD.drawString("WIFI_NAME:", 0, 48, 2);
    myGLCD.drawString(ssid, 80, 48, 2);

    while(WiFiMulti.run() != WL_CONNECTED) /* 等待网络连接成功 */
    {
        delay(200);
        myGLCD.drawChar('.', 10 + 8 * i, 64, 2);
        i++;
    }

    myGLCD.fillRect(0, 48, 160, 16, TFT_WHITE);
    myGLCD.drawString("WiFi connected.", 0, 48, 2);
    delay(1000);
    myGLCD.fillRect(0, 48, 160, 16, TFT_WHITE);
    myGLCD.drawString("IP address:", 0, 48, 2);

    sprintf(ip_buf, "%s", WiFi.localIP().toString().c_str());
    myGLCD.drawString(ip_buf, 68, 48, 2); /* 显示连接 wifi 后的 ip */
    // Serial.println(ip_buf);
}

/***
 * @brief  循环函数, 通常放程序的主体或者需要不断刷新的语句
 * @param  无
 * @retval 无
 */
void loop()
{
    const uint16_t port = 80;          /* 要连接的端口号 */
    char *host = "192.168.1.247";    /* 要连接服务器的 IP */
    WiFiClient client;

    myGLCD.fillRect(0, 64, 160, 16, TFT_WHITE);
    myGLCD.drawString("Connect to", 0, 64, 2);
    myGLCD.drawString(host, 76, 64, 2);

    if (!client.connect(host, port)) /* 连接网络服务器 */
    {
        myGLCD.fillRect(0, 64, 160, 16, TFT_WHITE);
        myGLCD.drawString("Connection failed.", 0, 64, 2);
        Serial.println("Waiting 5 seconds before retrying...");
```

```

        delay(3000);
        return;
    }

    // client.print("hello server!This is ESP32-S3\n\n"); /* 向服务器发送数据 */
    client.print("GET /index.html HTTP/1.1\n\n"); /* 向服务器发送一个请求 */

    int maxloops = 0;

    while (!client.available() && maxloops < 30000) /* 等待服务器的回复 */
    {
        maxloops++;
        delay(1);
    }

    if (client.available() > 0) /* 服务器是否有数据 */
    {
        String line = client.readStringUntil('\r'); /* 从服务器回读一行 */
        Serial.print("Read: ");
        Serial.println((char *)line.c_str());
    }
    else
    {
        Serial.println("client.available() timed out ");
    }

    Serial.println("Closing connection.");
    client.stop(); /* 关闭连接 */

    Serial.println("Waiting 3 seconds before restarting.");
    delay(3000);
}

```

在 `setup` 函数中，调用 `led_init` 函数完成 LED 初始化，调用 `key_init` 函数完成 KEY 初始化，调用 `uart_init` 函数完成串口初始化，调用 `myGLCD.init` 函数完成 LCD 初始化，调用 `myGLCD.setRotation(1)` 设置横屏，调用 `myGLCD.fillRect` 清屏，调用 `WiFiMulti.addAP` 函数连接网络，通过 `WiFiMulti.run` 函数，通过 `WiFi.localIP` 函数获取 IP 地址。

在 `loop` 函数中，通过 `client.connect` 函数连接网络服务器，然后通过 `client.print` 函数发送请求，通过 `client.available` 函数查询是否有数据返回，若有数据返回串口打印出来，最后断开连接，三秒后重启。

19.4 下载验证

程序下载成功后，我们可以看到 LCD 显示已经连接上 WiFi，显示当前 ESP32-S3 的 IP，并开始连接服务器，请求数据后显示出来，如下图所示：

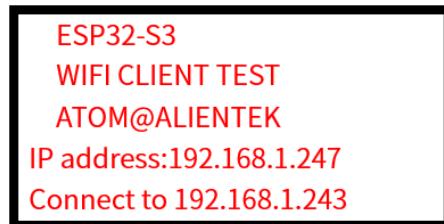


图 19.4.1 WiFi CLIENT 实验测试图

当然，也可以使用正点原子的 XNET 软件创建一个 TCP Server，然后让 ESP32-S3 去连接，也可以看到数据发送到服务器，这种方式大家可以自行尝试。

第二十章 BLE_SCAN 实验

本章，我们首先介绍基本的蓝牙基础知识，然后介绍如何使用 ESP32-S3 把附近的蓝牙设备扫描出来。

本章分为如下 4 个小节：

- 20.1 蓝牙基础知识介绍和 BLEScan 介绍
- 20.2 硬件设计
- 20.3 软件设计
- 20.4 下载验证

20.1 蓝牙基础知识和 BLEScan 介绍

20.1.1 蓝牙介绍

蓝牙，是一种支持设备短距离通信的无线通信技术，最早是由爱立信公司于 1994 年发明。蓝牙的目标是使各类移动设备、嵌入式设备、计算机外设和家用电器等众多设备之间在没有电缆连接的情况下能够在短距离范围内实现信息的自由传输与分享。相比较其他无线通信技术，蓝牙具有安全性高、易于连接等优势。

蓝牙采用分散式网络结构以及快跳频和短包技术，支持点对点及点对多点的通信，工作在全球通用的 2.4 GHz ISM（即工业、科学、医学）频段。蓝牙可分为经典蓝牙和低功耗蓝牙。

（1）经典蓝牙。经典蓝牙，简称 BT，泛指支持蓝牙协议在 4.0 版本以下的模块，一般用于如语音、音乐等大数据量的传输。经典蓝牙的协议包含了个人局域网的各种规范，不同的规范对应于不同的应用场景，比较常用的有：适用于音频的 Advance Audio Distribution Profile、适用于免提设备的 Hands-Free Profile/Head-Set Profile（HFP/HSP）、适用于文本串口透传的 Serial Port Profile（SPP）、适用于无线输入/输出设备的 Human Interface Device（HID）。

（2）低功耗蓝牙。低功耗蓝牙，简称 BLE，是一种新型的超低功耗无线通信技术，主要针对低成本、低复杂度的无线体域网和无线个域网设计，最主要的优点之一是可以用纽扣电池为低功耗蓝牙芯片供电，结合微型传感器构建出各种嵌入式传感器或可穿戴式传感器与传感器网络应用。

总体来说，蓝牙协议版本有两个分支，分别是经典蓝牙和低功耗蓝牙。其中，蓝牙 1.1、1.2、2.0、2.1、3.0 版本属于经典蓝牙，4.0 版本的蓝牙包括经典蓝牙和低功耗蓝牙，4.0 版本以后的蓝牙添加了低功耗蓝牙。

20.1.2 蓝牙协议介绍

蓝牙协议规定了两个层次，分别为蓝牙核心协议和蓝牙应用层协议。蓝牙核心协议是对蓝牙技术本身的规范，主要包括控制器（Controller）和主机（Host），不涉及其应用方式；蓝牙应用层协议是在蓝牙核心协议的基础上，根据具体的应用需求定义出的特定策略。蓝牙协议栈如下图所示。

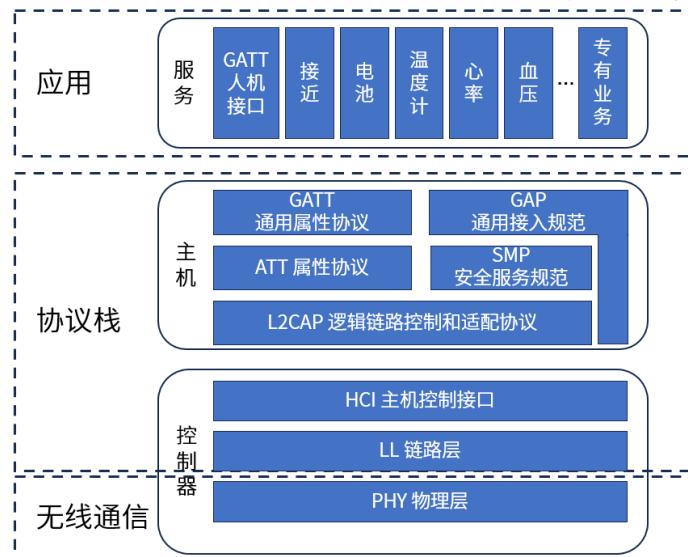


图 20.1.2 蓝牙协议栈图

低功耗蓝牙协议栈包含物理层（Physical Layer, PHY）、链路层（Link Layer, LL）、主机控制接口（Host Controller Interface, HCI）、逻辑链路控制和适配协议（Logical Link Control and Adaptation Protocol, L2CAP）、属性协议（Attribute Protocol, ATT）、安全服务规范（Security Manage Protocol, SMP）、通用属性协议（Generic Attribute Profile, GATT）、通用接入规范（Generic Access Profile, GAP）。

物理层 PHY 工作在 2.4GHz ISM 无线频段，次啊用高斯频移键控 GFSK 的调制方式，负责从物理信道发送和接收数据包。在低功耗蓝牙中，2.4GHz ISM 频段被划分为 40 个信道，单个信道宽度为 2MHz，物理层速率为 1Mb/s。在这 40 个信道中，0~36 号信道采用自适应调频技术收发数据，37~39 号信道负责广播。

链路层 LL 位于物理层之上，为逻辑链路控制和适配协议提供服务，负责广播、扫描、建立和维护链接，选择正确的方式、合适的信道交换数据包，并支持不同的拓扑结构，其操作可描述为就绪、初始、扫描、链接、广播 5 种状态，是整个低功耗蓝牙协议栈的 **核心**。

主机控制接口 HCI 提供了主机与控制器通信的通信方式和命令时间格式。它允许主机将命令和数据发送到控制器，并允许控制器将时间和数据发送到主机。主机控制接口由两部分组成，逻辑接口和物理接口。逻辑接口定义了命令和事件以及相关行为。物理接口定义了命令、事件和数据如何通过不同的链接技术来传输。

逻辑链路控制和适配协议 L2CAP 向上层协议（协议复用、分段、重组操作）提供连接导向和无连接的数据服务，并按通道进行流量控制和重传。

属性协议 ATT 允许蓝牙设备以“属性”（Attribute）的形式向其他蓝牙设备暴露自己的某些数据。通过一个固定的 L2CAP 信道，ATT 客户端与位于远端设备上的 ATT 服务器进行交互。在 ATT 协议种，暴露属性的称为服务器 Server，而另一端就是客户端 Client。

安全服务规范 SMP 负责管理低功耗蓝牙连接的加密和安全，既保证连接的安全性，同时又不影响用户的体验。安全服务规范通过一个固定的 L2CAP 信道，实现设备间的配对、认证和加密功能。

通用属性协议 GATT 位于属性协议之前，定义了属性的类型及其使用方法。通用属性协议 GATT 和属性协议 ATT 被强制安装在低功耗蓝牙上用于支持发现远端设备服务的功能。

通用接入规范 GAP 定义了蓝牙设备如何发现、链接及绑定其他设备，这是其他蓝牙应用得以运用的基础。低功耗蓝牙定义 4 种 GAP 角色：广播设备、观察设备、中心设备和外围设备。中心设备时向外围设备发起链接的设备，需同时具有发射和接收装置，一旦链接成功，它将称为主设备，被链接的外围设备将称为从设备。

20.1.3 工作状态和工作角色介绍

链路层定义了蓝牙的五种状态：就绪态、广播态、扫描态、发起态和链接态，如下图所示。

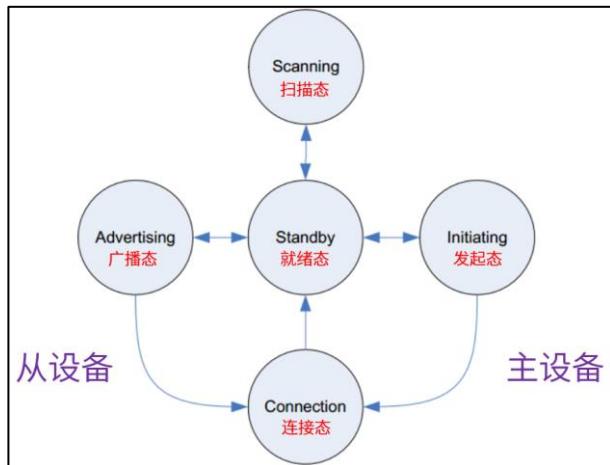


图 20.1.3.1 链路层状态机

处于就绪态（Standby）时，链路层不收发报文，任何状态都可以进入到就绪态。

处于广播态（Advertising）的链路层可以发送广播报文，也可以监听以及响应这些广播报文触发的响应报文。可被发现（也就是可以被扫描到）或可被链接（后面进行数据通信）的设备必须处于广播态，而需要向一定范围内的其他蓝牙设备广播数据的设备也必须处于广播态。

处于扫描态（Scanning）的设备能够接收广播报文。扫描又分为两种，主动扫描和被动扫描。主动扫描可以发送扫描请求给广播态设备，并获取额外的扫描响应数据，而被动扫描仅仅是接收到广播报文。

处于发起态（Initiating）设备可以发起链接请求。如果处于发起态的发起者接收到了来自其他设备的广播报文，链路层会向其发送链接请求并进入链接态。如果发起者不再发起链接，也可进入到就绪态。

处于链接态（Connection）下的设备才是我们经常看到的数据传输状态。并且又会区分为主设备（Master）和从设备（Slave）。由发起态进入链接态的设备是主设备，由广播态进入链接态的设备是从设备。

根据链路层处于状态不同，可分为 5 种不同的工作角色：广播者、扫描者、发起者、主设备和从设备。

链路层不能同时执行主设备和从设备两个角色。执行主设备角色的链路层可以同时执行广播者角色，或者扫描者角色，或者发起者角色。执行从设备角色的链路层可以同时执行广播者角色或扫描者角色。从设备不能发送可链接的广播报文，但可以发送不可链接的广播报文或可发现的广播报文。

20.1.4 蓝牙设备链接建立过程

传统蓝牙支持两种拓扑结构，微微网和分布式网络，而低功耗蓝牙仅支持微微网拓扑结构。对于低功耗蓝牙来说，一个主设备是可以跟多个从设备进行通信的，拓扑结构如下图所示。

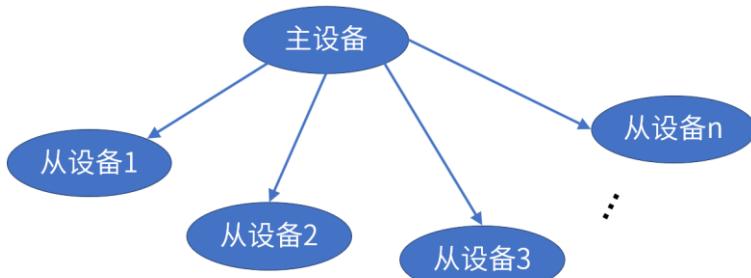


图 20.1.4.1 低功耗蓝牙系统拓扑结构

接下来讲解一下，两个蓝牙设备是如何建立链接并实现通信的。假设有两个蓝牙设备，即一台手机 A 和照相机 B 要进行链接并实现通信，如下图所示。

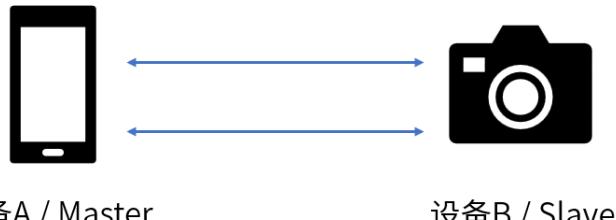


图 20.1.4.2 蓝牙设备链接建立图

这个链接过程如下：

(1) 设备 B 要处在“可见”或“可被发现”模式下。在这种模式下，设备 B 就可被设备 A 或其他蓝牙设备所发现，即“可被发现”状态。

(2) 设备 A 在一定的距离内, 通过在广播信道上发送广播包搜寻可被发现的蓝牙设备, 这个过程称为“询问”。在“询问”的过程中, 设备 A 将定位设备 B。

(3) 定位设备 B 后, 设备 A 将建立与设备 B 的链接。此时, 设备 B 变为“可被链接”状态。

(4) 一切就绪后, 设备 A 建立一个与设备 B 之间的链接, 这个过程称为“扫描”。

(5) 两个设备之间的链接一旦建立,最先发起链接请求的设备A称为主设备,被建立链接的设备B就是从设备。此时,主设备和从设备之间的通信已建立,并可以在同步同频的状态下相互收发数据包。

(6) 当主、从设备不再需要连接时，主、从设备都可以发起断开链接请求，并断开两个设备间的链接。

20.1.5 蓝牙扫描介绍

蓝牙设备想要组建网络传输数据，首先得通过广播或扫描发现周围设备，其次才是创建连接，前者为本实验要实现的内容：蓝牙扫描。蓝牙扫描只是发现周边的蓝牙设备，并没有与任一进行连接，涉及到的过程只是 20.1.4 小节中链接过程的第 1、2、3 和 4 步。

从机想要被主机连接，那么它就必须要先被主机发现。这时候，从机把自身消息以广播形式发送出去。从机需要先进行广播，不断发送广播包，没发送一次广播包，我们称之为一次广播事件。当进行广播事件时，每一个事件包含 3 个广播包，分别在 37、38 和 39 这三个信道上同时广播相同的消息。

扫描是一个在一定范围内用来寻址其他低功耗蓝牙设备广播的过程，扫描者在扫描过程中会使用广播信道。与广播过程不同的是，扫描过程没有严格的事件定义和信道规则，按照主机设定的扫描定时参数进行。

蓝牙扫描被分为被动扫描和主动扫描。

被动扫描。在被动扫描中，扫描者仅仅监听广播包，而不向广播者发送任何数据，被动扫描的过程如下：

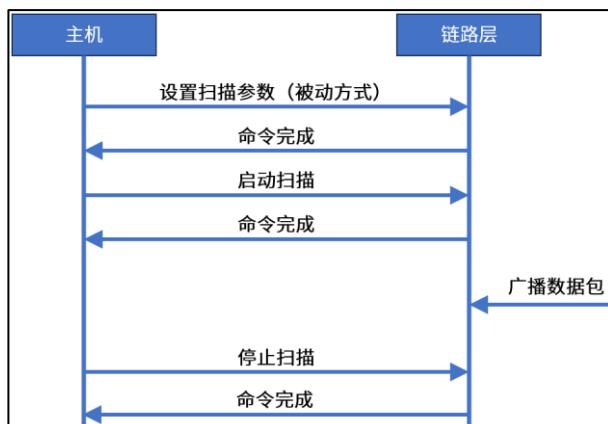


图 20.1.5.1 被动扫描过程

一旦设置好扫描参数，主机就可以在协议栈中发送命令启动扫描。在扫描过程中，如果控制器接收到符合过滤策略或其他规则的广播包，则向主机发送一个报告事件。报告事件除了包括广播者的设备地址，还包括广播包中的数据，以及接收广播包时的信号接收强度。利用信号接收强度以及广播包中的发射功率，共同确定信号的路径损失，从而给出大致的范围，该方面的典型应用就是防丢器和蓝牙定位。

主动扫描。在主动扫描中，主机不仅可以捕获到从机发送的广播包，还可以捕获扫描响应包，并区分广播包和扫描响应包。主动扫描的过程如下图所示。

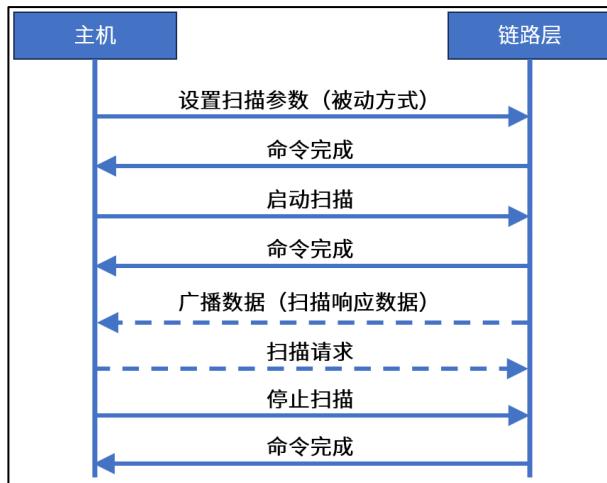


图 20.1.5.2 主动扫描过程

控制器收到数据后将向主机发送一个报告事件，该报告事件包括了链路层数据包的广播类型，因此主机能够判断从机是否可以连接或扫描，并区分广播包和扫描响应包。

蓝牙扫描中获取到的蓝牙设备数据是来自于 ADV_IND 包。ADV_IND 包中包含广播者设备地址、设备名、发送功率和信号强度等。

20.1.6 BLEScan 库函数介绍

ESP32-S3 集成了低功耗蓝牙系统，支持 Bluetooth 5 和 bluetooth Mesh，不支持经典蓝牙。而 ESP32 是同时支持 BT 和 BLE 的。

本实验实现的 BLE 网络扫描主要依赖的是 BLEScan 库，还会涉及到 BLEDevice 库。BLEDevice 库主要用到 init 函数和 getScan 函数。

ESP32-S3 BLESCAN 功能函数主要分为两类：管理扫描和配置扫描，如下图所示。

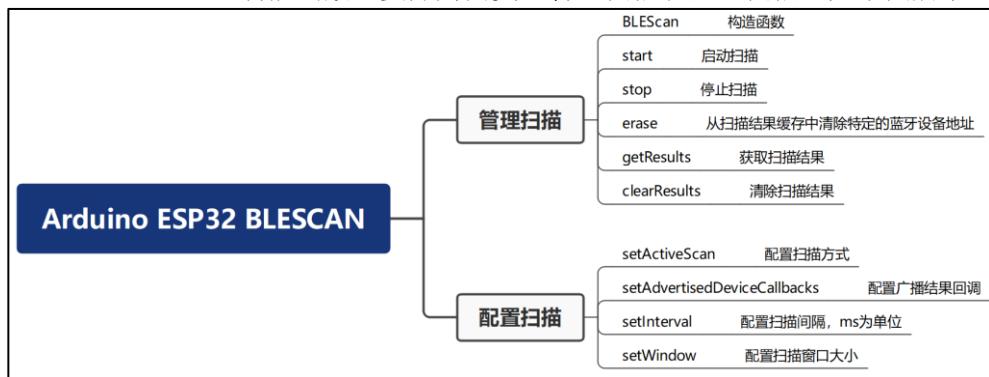


图 20.1.6 Arduino ESP32 BLESCAN 功能的函数列表

管理扫描主要是管理扫描整个扫描过程，包括启动、获取结果、停止等；而配置扫描主要是配置如何进行扫描。

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\BLE\src\BLEDevice.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\BLE\src\BLEScan.cpp

接下来，我们介绍一下本章节所用到的用到的函数。

第一个函数: init 函数, 该函数功能是创建一个 BLE 设备。

```
void BLEDevice::init(std::string deviceName)
```

参数 deviceName 为 BLE 设备名称。

无返回值。

第二个函数: getScan 函数, 该函数功能是创建扫描对象。

```
BLEScan* BLEDevice::getScan()
```

无参数;

返回值为 BLEScan 对象。

第三个函数: start 函数, 该函数功能是启动同步扫描。

```
BLEScanResults BLEScan::start(uint32_t duration, bool is_continue)
```

参数 duration 为扫描时间;

参数 is_continue 为是否清楚扫描设备映射, false 表示清除;

返回值为 BLEScanResults 对象。

第四个函数: getResults 函数, 该函数的功能是获取扫描结果。

```
BLEScanResults BLEScan::getResults()
```

无参数;

返回值为 BLEScanResults 对象。

第五个函数: clearResults 函数, 该函数的功能是清除扫描结果。

```
void BLEScan::clearResults()
```

无参数;

无返回值。

20.2 硬件设计

1. 例程功能

程序下载完成, ESP32-S3 尝试扫描附近的蓝牙设备, 并把扫描到的设备数量显示出来, 设备名字和信号强度通过串口打印出来。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11
DC-IO40
BL-IO41
RST-IO38

3. 原理图

本章实验使用的 BLE 为 ESP32-S3 的片上资源, 因此并没有相应的连接原理图。

20.3 软件设计

20.3.1 程序流程图

下面看看本实验的程序流程图:

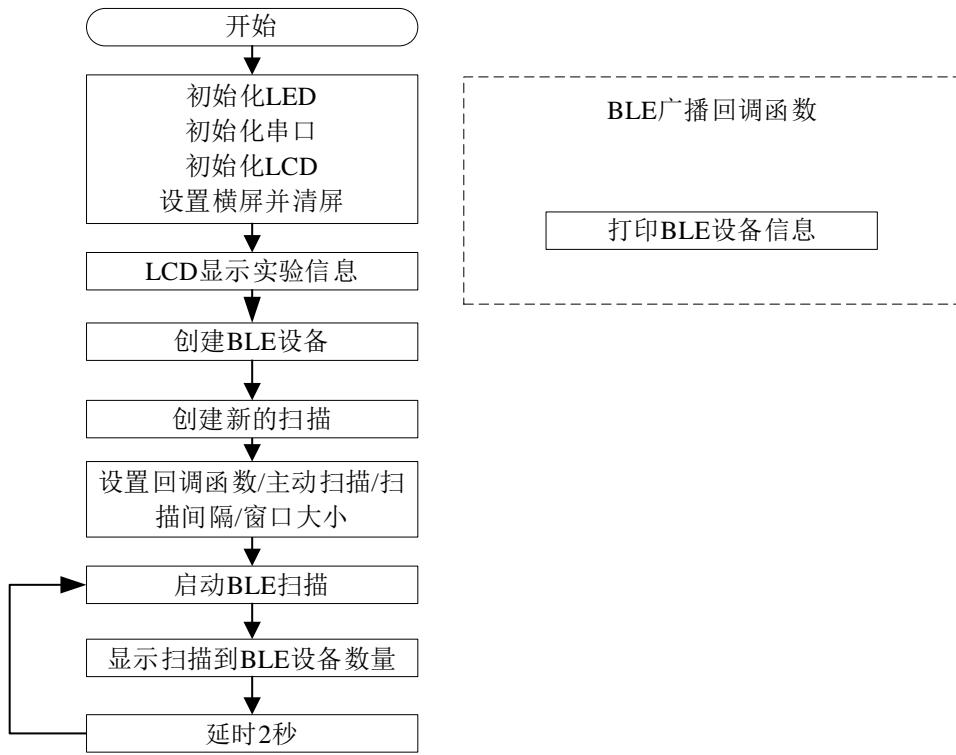


图 20.3.1 程序流程图

20.3.2 程序解析

1. 04_ble_scan.ino 代码

在 04_ble_scan.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include <BLEDevice.h>           /* 蓝牙 BLE 设备库 */
#include <BLEUtils.h>
#include <BLEScan.h>             /* 蓝牙 BLE 设备的扫描功能库 */
#include <BLEAdvertisedDevice.h>  /* 扫描到的蓝牙设备（广播状态） */

TFT_eSPI myGLCD = TFT_eSPI();      /* 定义 TFT_eSPI 对象 myGLCD */
int scanTime = 5;                  /* 蓝牙扫描时间 */
BLEScan* pBLEScan;                /* 扫描对象 */

/* BLE 广播回调函数(每次扫描到广播设备时被调用) */
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks
{
public:
    void onResult(BLEAdvertisedDevice advertisedDevice)
    {
        Serial.printf("Advertised Device: %s \n",
                      advertisedDevice.toString().c_str()); /* 可输出设备的信息 */
    }
};

/**
 * @brief  当程序开始执行时，将调用 setup() 函数，通常用来初始化变量、函数等
 * @param  无
 * @retval 无
 */

```

```

void setup()
{
    led_init();           /* LED 初始化 */
    uart_init(0, 115200); /* 串口 0 初始化 */
    myGLCD.init();       /* LCD 初始化 */
    myGLCD.setRotation(1); /* 设置屏幕的方向(横屏) */
    myGLCD.fillScreen(TFT_WHITE);

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
    myGLCD.drawString("BLE SCAN TEST", 10, 16, 2);
    myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

    myGLCD.drawString("Scanning...", 10, 48, 2);
    BLEDevice::init("ESP BLEDevice"); /* 创建一个 BLE 设备 */
    pBLEScan = BLEDevice::getScan(); /* 创建新的扫描 */
    /* 初始化回调函数 */
    pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pBLEScan->setActiveScan(true); /* 主动扫描消耗更多的能量, 但更快地得到结果 */
    pBLEScan->setInterval(100); /* 设置扫描间隔 */
    pBLEScan->setWindow(99); /* 设置窗口大小 */
}

/**
 * @brief 循环函数, 通常放程序的主体或者需要不断刷新的语句
 * @param 无
 * @retval 无
 */
void loop()
{
    /* 启动 BLE 扫描, 并在扫描到广播设备时调用回调函数 */
    BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
    myGLCD.drawString("Devices found:", 10, 64, 2);
    myGLCD.drawNumber(foundDevices.getCount(), 112, 64, 2);
    myGLCD.drawString("Scan done!", 10, 48, 2);
    pBLEScan->clearResults(); /* 从 BLEScan 缓冲区中删除结果以释放内存 */
    delay(2000);
}

```

首先定义 BLEScan 对象实例为 pBLEScan。编写回调函数 onResult，在 setup 函数中会进行设置。

在 setup 函数中，调用 led_init 函数完成 LED 初始化，调用 key_init 函数完成 KEY 初始化，调用 uart_init 函数完成串口初始化，调用 myGLCD.init 函数完成 LCD 初始化，调用 myGLCD.setRotation(1)设置横屏，调用 myGLCD.fillScreen 清屏，调用 BLEDevice::init 函数创建 BLE 设备，然后通过 BLEDevice::getScan 函数创建新的扫描，通过 setAdvertisedDeviceCallbacks 函数初始化回调函数，通过 setActiveScan 函数设置主动扫描，通过 setInterval 函数设置扫描间隔，通过 setWindow 设置窗口大小。

在 loop 函数中，通过 start 函数启动 BLE 扫描，扫描到广播设备时会进入到回调函数 OnResult 中进行处理，串口打印设备信息，最后调用 clearResults 函数从缓冲区中删除扫描结果释放内存。

20.4 下载验证

程序下载成功后，我们可以看到 LCD 显示扫描到的 BLE 设备数量，如下图所示：

ESP32-S3
BLE SCAN TEST
ATOM@ALIENTEK
Scan done
Devices found: 16

图 20.4.1 BLE SCAN 实验测试图

串口会打印扫描到的蓝牙设备信息，如下图所示：

```
输出 串口监视器 x
消息 (按回车将消息发送到“COM53”上的“ESP32S3 Dev Module”)

ESP-ROM: esp32s3-20210327
Build: Mar 27 2021
rst:0x1 (POWERON), boot:0x2b (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fce36008, len:0x44c
load:0x403c9700, len:0xbe4
load:0x403c9700, len:0x2a68
entry 0x403c98d4

Advertised Device: Name: , Address: 2c:9e:bf:66:23:7a, manufacturer data: 0600010920026c7e9f7dfbb0546cd91df760ce2a7a453ab7169e84ac7, rssi: -89
Advertised Device: Name: RTX_BT_4.1
Advertised Device: Name: , Address: 7e:57:58:de:b3:e5, manufacturer data: 650001c904, serviceUUID: 0000fe70-0000-1000-8000-00005f9b34fb, rssi: -90, serviceData:
Advertised Device: Name: , Address: 11:da:20:45:45:84, manufacturer data: 060001092022779e30c034f922484f15484a2576cf1f7178fafbf1f6d1, rssi: -79
Advertised Device: Name: , Address: 63:21:86:af:3e:b2, manufacturer data: 4c0010063b1a340cbc9, txPower: 12, rssi: -93
Advertised Device: Name: , Address: 3e:ae:00:8d:94:70, manufacturer data: 060001092002e1c832d0338b58084dd7ad84a3c047a0612cf0fc1ed2e, rssi: -88
Advertised Device: Name: , Address: 12:18:31:45:7c:00, manufacturer data: 060001092002426f72d5329e5a81f8d1685ff2903cf4b24e23aa35e46a, rssi: -69
Advertised Device: Name: , Address: 22:29:2f:aa:4f:5c, manufacturer data: 0600010920020cf5b2828990943a2187faf716d9742b925f0d5d523a4ed, rssi: -83
Advertised Device: Name: Xiaomi Watch Color 2 5863, Address: 44:27:f3:4f:58:63, rssi: -52, serviceData: Y cX0 'D, serviceData:
Advertised Device: Name: , Address: 59:b3:a8:ee:34:29, rssi: -72, serviceData:
Advertised Device: Name: Redmi Watch 2 5A5, Address: 44:27:f3:3b:5a:e5, rssi: -63, serviceData: Yg Z: 'D, serviceData:
Advertised Device: Name: , Address: 70:bb:7c:b6:c9:9a, manufacturer data: 4c0016080087b933625bb87b, rssi: -91
Advertised Device: Name: , Address: e6:1e:f1:16:f3:30, manufacturer data: 4c0012020003, rssi: -91
Advertised Device: Name: , Address: 45:94:53:a7:30:d4, manufacturer data: 0600010920021773f4e7cd424b90b1764b54542e618199248104d1df2f, rssi: -90
Advertised Device: Name: , Address: 51:97:ec:44:23:75, manufacturer data: 4c001007731fea0fe3008, txPower: 7, rssi: -96
Advertised Device: Name: midea, Address: b0:96:ea:fb:c6:53, manufacturer data: a806013232323531353231414330303630, rssi: -95
Advertised Device: Name: EDIFIER BLE, Address: 64:68:70:00:e5:dc, manufacturer data: 64687600f34a, serviceUUID: 00003400-0000-1000-8000-00005f9b34fb, rssi: -88
```

图 20.4.2 BLE 设备信息

第二十一章 BLE_UART 实验

本章，我们介绍如何使用 ESP32-S3 开启蓝牙跟手机安装的蓝牙助手进行通信。

本章分为以下几个小节：

- 21.1 蓝牙基础知识和蓝牙通信函数介绍
- 21.2 硬件设计
- 21.3 软件设计
- 21.4 下载验证

21.1 蓝牙基础知识和蓝牙通信函数介绍

在上一章已经讲解了不少蓝牙的基础知识了，本章针对性讲解蓝牙设备连接上之后的数据通信。这部分内容主要涉及到属性协议、通用属性协议的内容。通用属性协议 GATT 就好比一个公司的出纳和库房前台，属性协议 ATT 就是库房。出纳负责处理向上与应用打交道，而库房前台负责向下把检索任务子进程交给 ATT 库房去做，库房则负责数据检索。

属性协议是用于发现、读取和写入对端设备上的属性的规范。属性协议采用了一种客户端服务器模式。服务器暴露一系列属性给客户端。这些属性可以由客户端发现、读取和写入，也可以由服务器指示和通知。

21.1.1 属性介绍

属性代表着数据，设备在任意状态任意事件所对应的相关数据。它可以是位置、尺寸、质量、温度、速度或某个设备向分享给其他设备的任何数据。属性协议用来向远端设备推送或获取数据。此外，属性协议也支持设置通知和指示，当数据发生变化时远端设备可以获得警报。

例如，温度计所测得的温度，温度的单位、设备的名称型号及设备制造商的名称等都是属性。属性由 4 个元素组成：属性句柄、属性类型、属性值和属性许可。

属性句柄：与使用内存地址查找内存中的内容一样，通过属性句柄以找到相应的属性。例如，第一个属性的句柄是 0x0001，第二个属性的句柄是 0x0002，以此类推，属性句柄最大可以为 0xFFFF。

属性类型：每个数据有自己需要代表的意思，例如，表示温度、发射功率、电量等各种各样的信息。蓝牙组织 Bluetooth SIG 对常用的一些数据类型进行了归类，赋予不同的数据类型不同的通用唯一识别码（Universally Unique Identifier，UUID）。例如，0x2A09 表示电池信息，0x2A6E 表示温度信息。UUID 可以是 16 位，也可以是 128 位。

属性值：属性值是每个属性真正要承载的信息，其他 3 个元素都是为了让对方能够更好地获取属性值。有些属性的长度是固定的，例如，电池属性的长度只有 1 个字节，因为需要表示的数据仅为“0~100%”，而 1 字节足以表示 1~100；而有些属性的长度是可变的，如基于 BLE 实现的透明传输模块。

属性许可：每个属性对各自的属性值有相应的访问限制，例如，有些属性是可读的，有些是可写的，有些是可读又可写的等等。拥有数据的一方可以通过属性许可控制本地数据的可读/写属性。

存有数据（即属性）的设备称为服务器端，获取数据的设备称为客户端。

下面是服务器端和客户端间的常用操作：

① 客户端给服务器端发送数据，对服务器端的数据进行写操作（写操作分为两种，一种是写入请求，另一种是写入命令，两者的主要区别是前者需要对方回复，而后者不需要）

② 服务器端给客户端发送数据，主要是通过服务器端指示或者通知的形式，将服务器端更新的数据发送给客户端（与写操作类似，指示和通知的主要区别是前者需要对方回复确认）。

③ 客户端也可以主动通过读操作读取服务器端的数据。服务器端和客户端的交互操作都是通过消息 ATT 的协议数据单元（Protocol Data Unit，PDU）实现的。每个设备可以指定自己支持的最大消息长度。

21.1.2 属性协议 AAT 介绍

属性协议定义了一个设备如何发现、读取和写入另外一台设备的属性。属性服务器暴露一系列属性及其属性值给一个匹配设备。属性客户端可以发现、读取和写入服务器中的属性，也可以被服务器指示或通知。

属性协议支持数据从客户端传向服务器，也支持从服务器传向客户端。客户端可以读写服务器的属性，而服务器也可以给客户端送属性。

属性协议数据单元有 6 种方法类型：

请求：由客户端向服务器发送，并调用响应。

响应：由服务器向客户端发送，响应某个请求。

命令：由客户端向服务器发送，且无响应。

通知：服务器未经请求向客户端发送。

指示：由服务器未经请求向客户端发送，并调用确认。

确认：由客户端向服务器发送，确认收到了一个指示。

21.1.3 通用属性协议 GAAT 介绍

属性协议 ATT 是用于发现、读取和写入远端设备上的属性的规范，而通用属性协议 GATT 则是构建在属性协议 ATT 的基础上，定义了一个基于属性协议的服务框架，定义服务的流程、格式及其包含的特征，该流程包括特征的发现、读取、写入、通知、指示以及广播的配置。

通用属性协议 GATT 定义了一系列程序，用来发现服务、特征、服务之间的关系，以及用来读取和写入特征值。GATT 结构如下图所示。

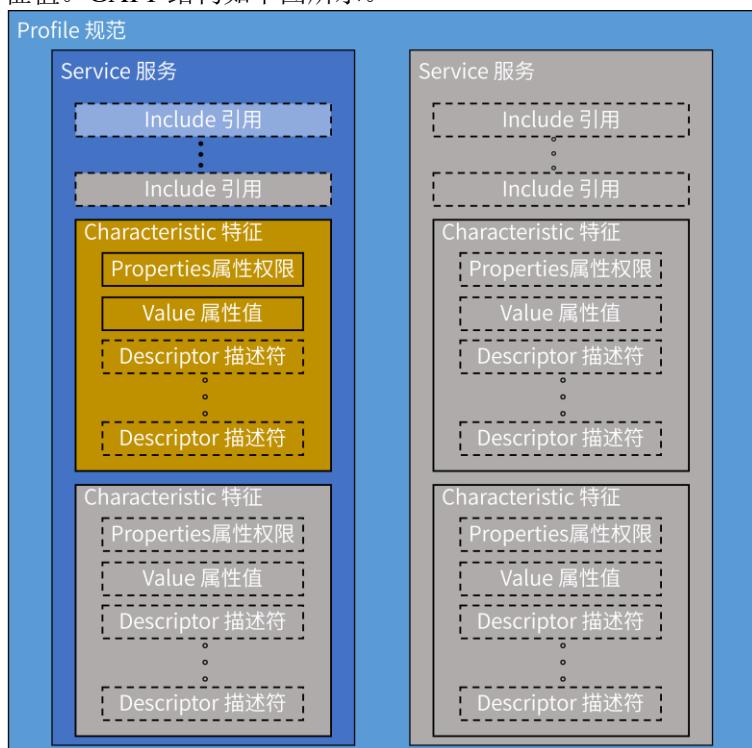


图 21.1.3.1 GAAT 结构图

Profile 规范是一个由一个或多个服务构成，它们结合在一起实现某一个特定功能。

Service 服务是一个数据和相关行为的集合，以实现特定的功能或部分设备的特征。每个 Service 都有一个 UUID 唯一标识。UUID 有 16bit 的，也有 128bit 的。16bit 的 UUID 是官方通过认证的，需要花钱购买，128bit 是自定义的，这个就可以随便设置。为了适应复杂的行为和服务之间的关系，服务分为两种类型：主要服务和次要服务。服务的类型通常不取决于服务本身，而取决于设备如何使用该服务。

Characteristic 特征是一个由属性值、属性权限、描述符组成的集合，是服务的组成部分之

一。与 Service 类似，每个 Characteristic 也会用 16bit 或 128bit 的 UUID 唯一标识。属性和配置信息表明数据是如何访问以及数值如何进行表示和显示，通常一个特性的定义包含属性权限、属性值和描述符 3 个部分。

例如，在一间公司里有 300 名员工，为了使他们的工作更有效率，现在将把他们划分为不同部门、次级部门。属性协议 ATT 规定了众多属性及属性之间的关联操作，可以把这公司里的 300 名员工比喻为属性协议 ATT。通用属性协议 GATT 将这些属性及其相关操作编入不同的规范 Profile、服务 Service 和特征 Characteristic。为了提高工作效率、方便管理，公司将这 300 名员工划分为若干个部门。规范就如同部门，例如后勤部或人力资源部，每个部门都相互独立并有其特定的职能。每个规范可以是一个或多个服务，就像后勤部能够实现物资管理、后勤保障等职能一样。每个服务都包含若干个次级服务或特征。这里把次级服务比作次级部门，例如后勤部的物资管理处等。而特征就如同部门里实现部门职能的个体，例如相关工作人员。

总之，通用属性协议 GATT 将功能类似的属性进行整合，将功能相关的属性整合到一个服务中，例如将和温度检测相关的属性整合为温度检测服务。而一个规范包含一个或多个这样的服务。

21.1.4 ESP32 GATT 介绍

基于 GATT 原理的介绍，ESP32 GATT 的实现会与很多库相关，如下图所示。



图 21.1.4.1 ESP32 GATT 库关联

设备角色：客户端（BLEClient）、服务器（BLEServer）。

GATT 服务相关：本地服务（BLEService、BLEServiceMap）、远端服务（BLERemoteService）。

GATT 服务下特征相关：本地特征（BLECharacteristic、BLECharacteristicMap）、远端特征（BLERemoteCharacteristic）。

GATT 服务下特征描述符相关：本地特征描述符（BLEDescriptor、BLEDescriptorMap）、远端特征描述符（BLERemoteDescriptor）。

GATT 服务下特征下的特征值相关：BLEValue。

以我们本实验要实现的功能来说，ESP32-S3 是作为一个服务器，被手机的蓝牙助手访问，所以这里会涉及到 BLEServer 的学习。作为服务器，其工作内容就是创建服务以及提供服务，所以会涉及到 BLEService、BLEServiceMap、BLECharacteristic、BLECharacteristicMap、BLEDescriptor、BLEDescriptorMap。

21.1.5 蓝牙通信函数介绍

在上一小节已经知道涉及到哪些库，本节主要对 BLEDevice 库做介绍，其他库的介绍可以自行查看一下源文件。

BLEDevice 作为蓝牙 BLE 的中心控制者，主要是四大功能：基础功能、广播（Advertising）、扫描（Scan）、数据通信（GATT）。

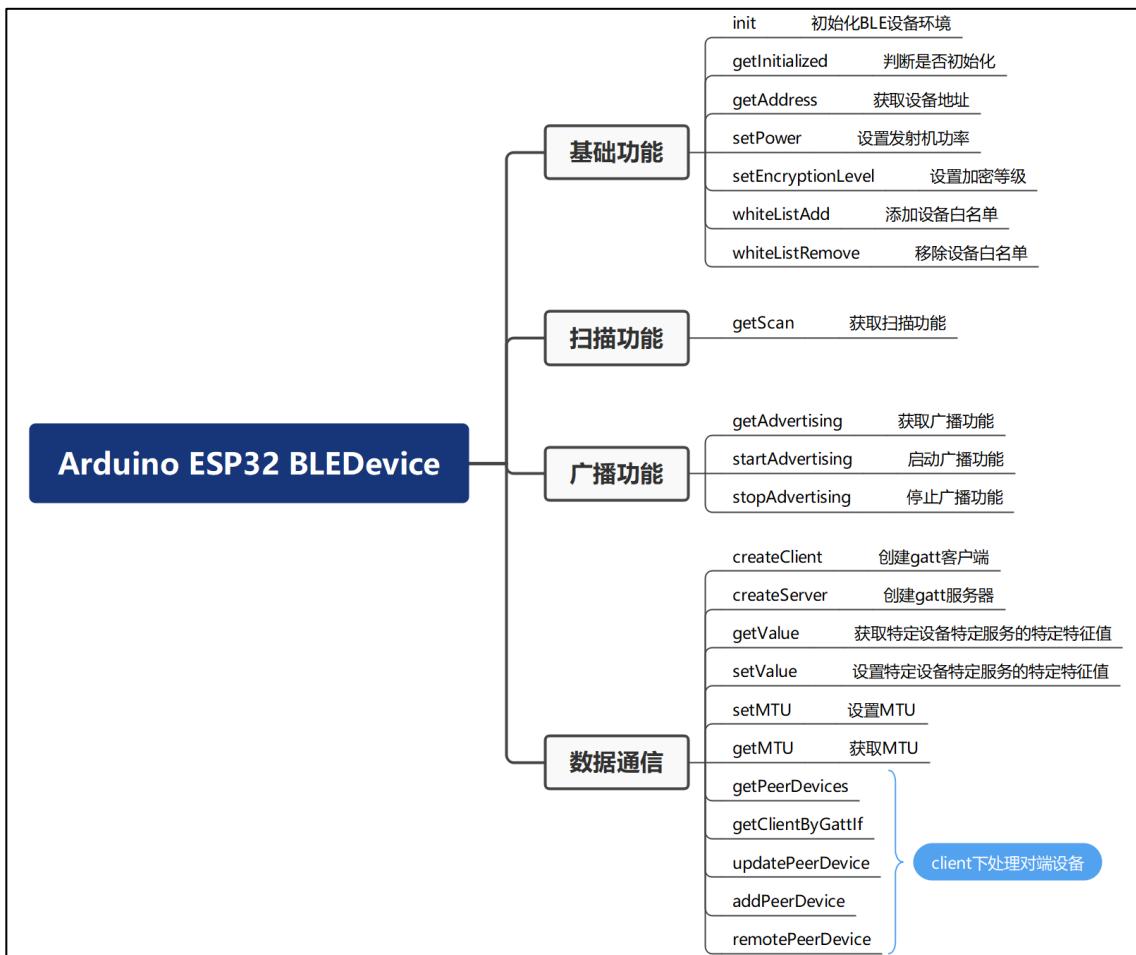


图 21.1.4.1 Arudino ESP32 BLEDevice 库函数

本小节介绍到的函数可在以下文件中找到：

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\BLE\src\BLEDevice.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\BLE\src\BLEServer.cpp

Arduino15\packages\esp32\hardware\esp32\2.0.11\libraries\BLE\src\BLECharacteristic.cpp

接下来，我们介绍一下本章节所用到的用到的函数。

第一个函数：init 函数，该函数功能是创建一个 BLE 设备。

```
void BLEDevice::init(std::string deviceName)
```

参数 deviceName 为 BLE 设备名称。

无返回值。

第二个函数：createServer 函数，该函数功能是创建 GATT 服务。

```
BLEServer* BLEDevice::createServer()
```

无参数；

返回值为 BLEServer 对象。

第三个函数：setCallbacks 函数，该函数功能是设置回调函数。

```
void BLEServer::setCallbacks(BLEServerCallbacks* pCallbacks)
```

参数 pCallbakcs 为要调用的函数；

无返回值。

第四个函数：createService 函数，该函数功能是创建服务。

```
BLEService* BLEServer::createService(const char* uuid)
```

参数 uuid 为通用唯一识别码；

返回值为 BLEService 对象。

第五个函数：createCharacteristic 函数，该函数的功能是创建特征对象。

```
BLECharacteristic* BLEService::createCharacteristic(const char* uuid, uint32_t properties)
```

参数 `uuid` 为通用唯一识别码；

参数 `properties` 为特征的属性；

返回值为 `BLECharacteristic` 对象。

第六个函数： `addDescriptor` 函数，该函数的功能是加入特征描述符。

```
void BLECharacteristic::addDescriptor(BLEDescriptor* pDescriptor)
```

参数 `pDescriptor` 为特征 描述符；

无返回值。

第七个函数： `start` 函数，该函数的功能是启动服务。

```
void BLEService::start()
```

无参数；

无返回值。

第八个函数： `getAdvertising` 函数，该函数的功能是获取广播功能。

```
BLEAdvertising* BLEServer::getAdvertising()
```

无参数；

返回值为 `BLEAdvertising` 对象。

第九个函数： `setValue` 函数，该函数的功能是清除扫描结果。

```
void BLECharacteristic::setValue(uint8_t* data, size_t length)
```

参数 `data` 为特征设置的数据；

参数 `length` 为数据长度；

无返回值。

第十个函数： `notify` 函数，该函数的功能是通知。

```
void BLECharacteristic::notify(bool is_notification)
```

参数 `is_notification` 为是否启动通知功能；

无返回值。

第十一个函数： `startAdvertising` 函数，该函数的功能是启动广播。

```
void BLEDevice::startAdvertising()
```

无参数；

无返回值。

21.2 硬件设计

1. 例程功能

程序下载完成，ESP32-S3 作为蓝牙设备，通过手机蓝牙助手进行连接后，就可以进行蓝牙通信。

手机蓝牙助手可以在手机软件商店下载“BLE 调试宝”或“BLE 调试助手”，当然在“光盘→6，软件资料→1，软件→6，网络调试助手”找到。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) USART0
U0TXD-IO43
U0RXD-IO44
- 3) LCD
CS-IO39
SCK-IO12
SDA-IO11
DC-IO40
BL-IO41
RST-IO38

3. 原理图

本章实验使用的 BLE 为 ESP32-S3 的片上资源，因此并没有相应的连接原理图。

21.3 软件设计

21.3.1 程序流程图

下面看看本实验的程序流程图：

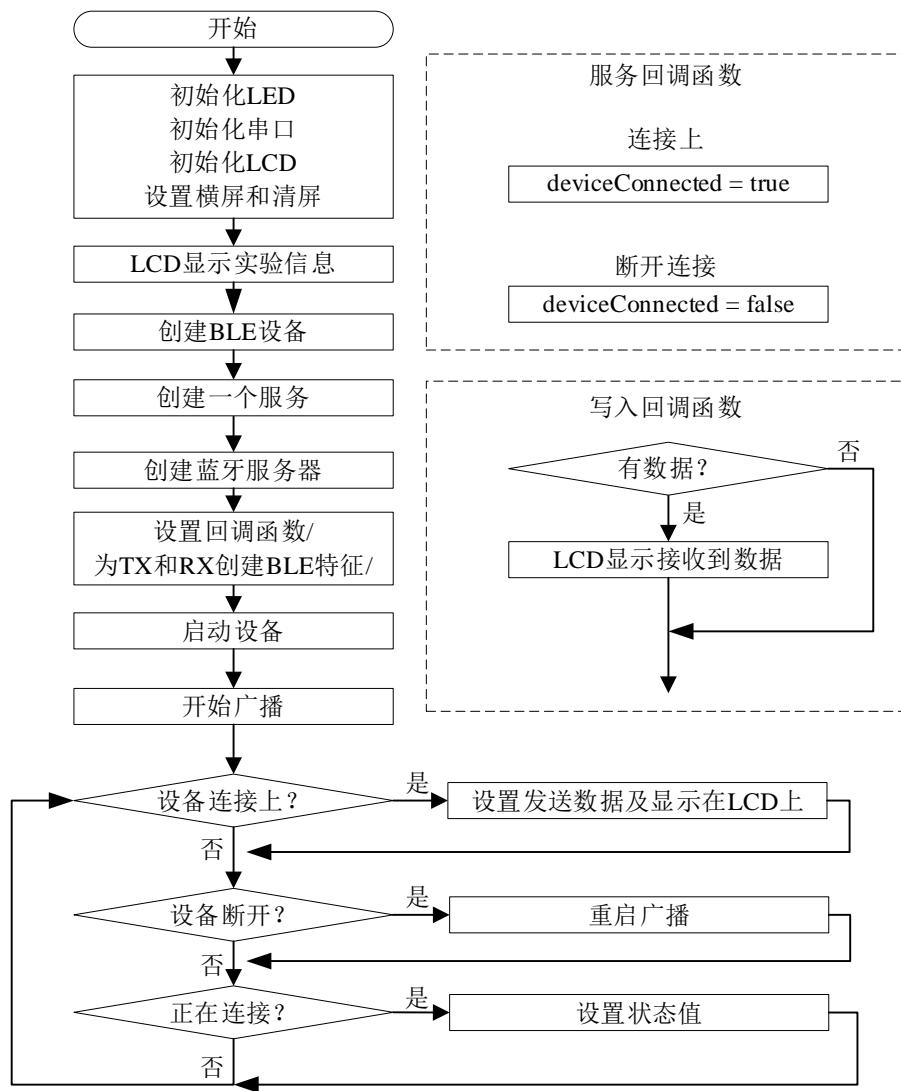


图 21.3.1 程序流程图

21.3.2 程序解析

1. 05_ble_uart.ino 代码

在 05_ble_uart.ino 里面编写如下代码：

```

#include "led.h"
#include "uart.h"
#include <SPI.h>
#include "TFT_eSPI.h"
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>

```

```
#include <BLE2902.h>

TFT_eSPI myGLCD = TFT_eSPI();
BLEServer *pServer = NULL;
BLECharacteristic *pTxCharacteristic;
BLECharacteristic *pRxCharacteristic;
bool deviceConnected = false;
bool oldDeviceConnected = false;
uint8_t txValue = 0;
char rec_buf[20];

#define SERVICE_UUID "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_RX "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"

class MyServerCallbacks: public BLEServerCallbacks
{
    void onConnect(BLEServer* pServer)
    {
        deviceConnected = true;
    }

    void onDisconnect(BLEServer* pServer)
    {
        deviceConnected = false;
    }
};

class MyCallbacks: public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *pCharacteristic)
    {
        std::string rxValue = pCharacteristic->getValue();

        if (rxValue.length() > 0)
        {
            myGLCD.drawString("rec value:", 10, 64, 2);
            myGLCD.fillRect(80, 64, 80, 16, TFT_WHITE);
            for (int i = 0; i < rxValue.length(); i++)
            {
                rec_buf[i] = rxValue[i];
            }
            myGLCD.drawString(rec_buf, 80, 64, 2);
            // Serial.println(rec_buf);
            memset(rec_buf, 0, 20);
        }
    }
};

/**
 * @brief 当程序开始执行时, 将调用 setup() 函数, 通常用来初始化变量、函数等
 * @param 无
 * @retval 无
 */
void setup()
{
    led_init(); /* LED 初始化 */
    uart_init(0, 115200); /* 串口 0 初始化 */
    myGLCD.init(); /* LCD 初始化 */
    myGLCD.setRotation(1); /* 设置屏幕的方向(横屏) */
    myGLCD.fillRect(0, 0, 128, 16, TFT_WHITE);

    myGLCD.setTextColor(TFT_RED, TFT_WHITE);
    myGLCD.drawString("ESP32-S3", 10, 0, 2);
}
```

```

myGLCD.drawString("BLE UART TEST", 10, 16, 2);
myGLCD.drawString("ATOM@ALIENTEK", 10, 32, 2);

BLEDevice::init("ESP32-S3 BLE Service");           /* 创建一个 BLE 设备 */
pServer = BLEDevice::createServer();                /* 创建一个 BLE 服务 */
pServer->setCallbacks (new MyServerCallbacks());    /* 设置回调 */

BLEService *pService = pServer->createService(SERVICE_UUID); /* 创建蓝牙服务器 */

/* 创建发送特征，添加描述符，设置通知权限 / 创建接收特征，设置回调函数，设置可写权限 */
pTxCharacteristic = pService->createCharacteristic(CHARACTERISTIC_UUID_TX,
BLECharacteristic::PROPERTY_NOTIFY);
pTxCharacteristic->addDescriptor(new BLE2902());
pRxCharacteristic = pService->createCharacteristic(CHARACTERISTIC_UUID_RX,
BLECharacteristic::PROPERTY_WRITE);
pRxCharacteristic->setCallbacks (new MyCallbacks());

pService->start(); /* 启动服务 */

pServer->getAdvertising()->start(); /* 开始广播 */
myGLCD.drawString("Waiting client connect", 10, 48, 2);

delay(500);
}

/***
* @brief 循环函数，通常放程序的主体或者需要不断刷新的语句
* @param 无
* @retval 无
*/
void loop()
{
    if (deviceConnected) /* 设备已经连接上 */
    {
        // myGLCD.drawString("client connected      ", 10, 48, 2);
        pTxCharacteristic->setValue(&txValue, 1); /* 设置要发送的值为 1 */
        myGLCD.drawString("notify value:          ", 10, 48, 2);
        pTxCharacteristic->notify();           /* 广播 txValue++ */
        myGLCD.drawString(txValue, 104, 48, 2);
        txValue++;
        delay(100); /* 如果发送的数据包太多，蓝牙堆栈将进入拥塞状态 */
    }

    if (!deviceConnected && oldDeviceConnected) /* 断开连接 */
    {
        delay(500); /* 蓝牙堆栈有机会做好准备 */
        pServer->startAdvertising(); /* 重启广播 */
        myGLCD.drawString("start advertising...", 10, 48, 2);
        oldDeviceConnected = deviceConnected;
    }

    if (deviceConnected && !oldDeviceConnected) /* 正在连接 */
    {
        oldDeviceConnected = deviceConnected;
    }
}
}

```

编写回调函数 `onConnect`、`onDisconnect` 和 `onWrite`，在 `setup` 函数中会进行设置。

在 `setup` 函数中，调用 `led_init` 函数完成 LED 初始化，调用 `uart_init` 函数完成串口初始化，调用 `myGLCD.init` 函数完成 LCD 初始化，调用 `myGLCD.setRotation(1)` 设置横屏，调用 `myGLCD.fillRect` 清屏，调用 `BLEDevice::init` 函数创建一个 BLE 设备，调用 `BLEDevice::createServer` 创建一个 BLE 服务，调用 `createService` 创建蓝牙服务器，调用 `createCharacteristic` 函数创建 2 个 BLE 特征，然后调用 `start` 函数启动设备，最后调用 `getAdvertising`

的 start 函数开始广播。

在 loop 函数中，通过 deviceConnected 变量执行不同的操作，设备已经连接上就往 TX 特征值中发送数据，断开连接就重新尝试链接。发送数据主要通过 setValue 和 notify 函数实现，而接收数据主要通过 getValue 函数实现。

21.4 下载验证

程序下载成功后，我们可以看到 LCD 显示 ESP32-S3 蓝牙等待被连接，如下图所示：



图 21.4.1 BLE UART 实验测试图

通过手机蓝牙助手“BLE 调试助手”连接上后，LCD 显示，如下图所示。



图 21.4.2 手机蓝牙助手连接上 ESP32-S3 并实现通信

手机蓝牙助手，如下图所示。

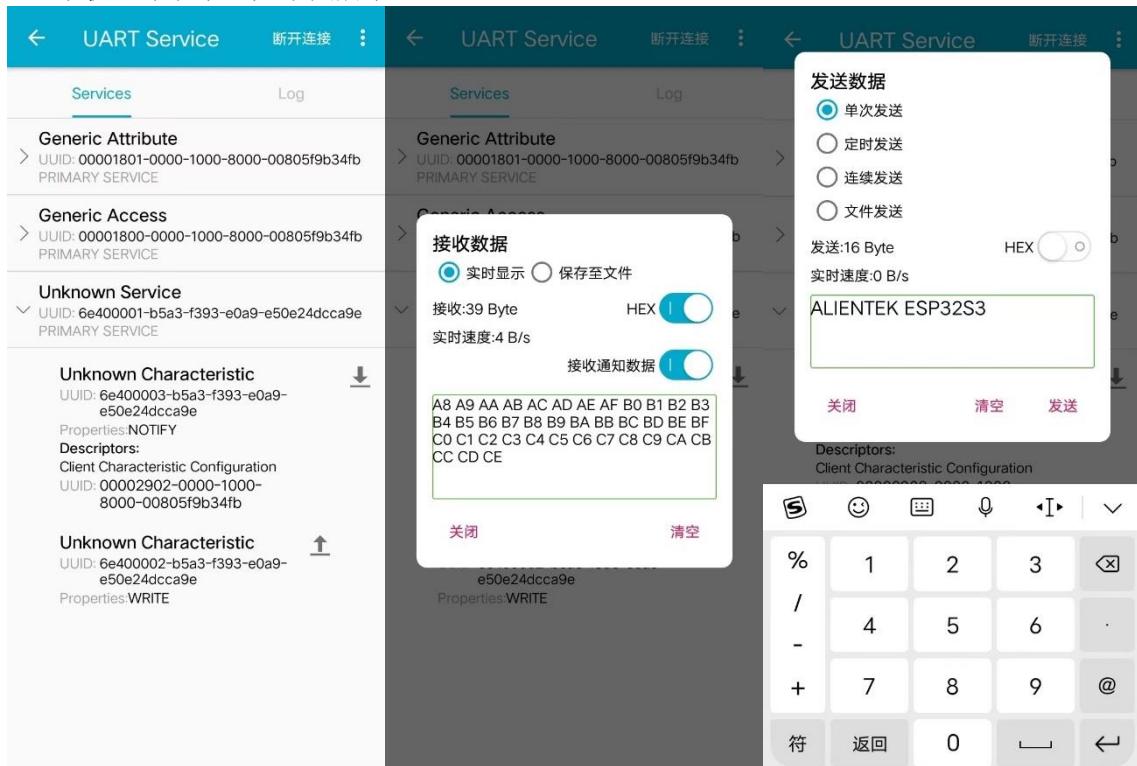


图 21.4.3 手机蓝牙助手集合图

可以看到上图左边部分，显示了蓝牙的名称“UART Service”，创建了一个 UUID 为“6E400001-B5A3-F393-E0A9-E50E24DCCA9E”的服务。在该服务下，创建了两个特征，发送特征的 UUID 为“6E400003-B5A3-F393-E0A9-E50E24DCCA9E”，接收特征的 UUID 为

“6E400002-B5A3-F393-E0A9-E50E24DCCA9E”。服务器的发送功能，对于客户端来说，就需要进行接收，所以这里发送特征的属性权限为 NOTIFY。服务器的接收功能，对于客户端来说，就是需要进行发送，所以这里接收特征的属性权限为 WRITE。

点击上图中左边部分的“向下箭头图标”进入到接收数据界面即上图中中间部分，可以看到数据被传送过来。点击上图中左边部分的“向上箭头图标”进入到发送数据界面即上图中右边部分，在文本输入框中输入“ALIENTEK ESP32S3”，点击发送，最终效果如图 21.4.2 所示，接收到数据。

BLE 涉及到比较多的知识点，需要大家自行去学习。