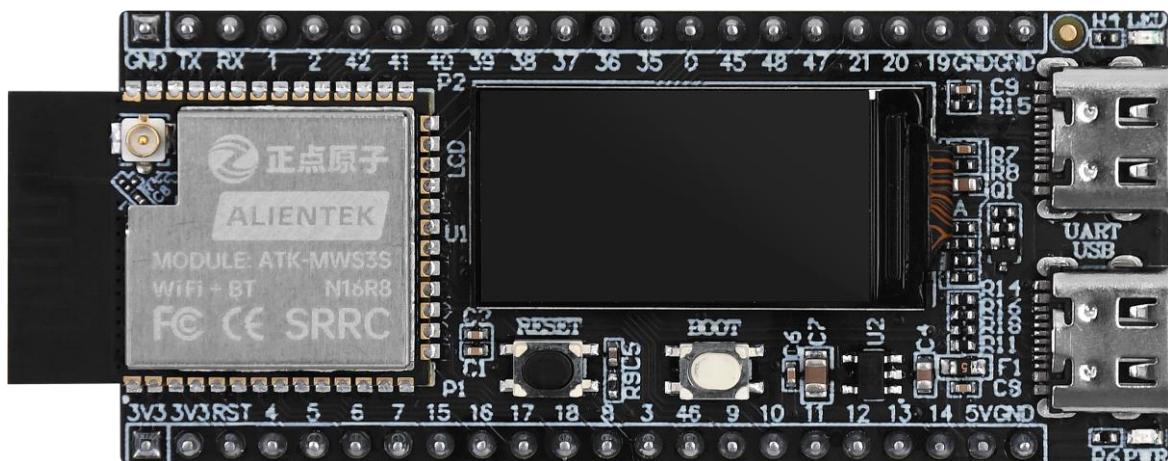


ESP32-S3 使用指南—

MicroPython 版

V1.0



—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:



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
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。
请关注正点原子弹公众号，资料发布更新我们会通知。



扫码下载“原子哥”APP



扫码关注正点原子弹公众号

前言	1
内容简介.....	2
第一篇 基础篇.....	3
第一章 本书学习方法.....	4
1.1 本书学习顺序.....	4
1.2 本书参考资料.....	4
1.3 本书编写规范.....	4
1.4 本书代码规范.....	5
1.5 例程资源说明.....	5
1.6 学习资料查找.....	6
1.7 给初学者的建议.....	7
第二章 MicroPython 与微控制器.....	8
2.1 MicroPython 缘起.....	8
2.2 微控制器软件.....	8
2.2.1 什么是微控制器.....	9
2.2.2 为什么使用 MicroPython.....	9
2.3 MicroPython 优点&应用场景.....	9
2.4 本章小结.....	10
第三章 MicroPython 基础知识.....	11
3.1 Python3 与 MicroPython 关联.....	11
3.2 MicroPython 系统结构.....	11
3.3 MicroPython 的 REPL 交互式环境	12
3.3.1 常用的终端.....	12
3.3.2 REPL 的快捷键	12
3.3.3 help()函数的作用	13
3.3.4 查看模块包含的变量和函数.....	14
第四章 ESP32-S3 基础知识	15
4.1 为什么选择 ESP32-S3	15
4.2 初识 ESP32-S3	16
4.3 ESP32-S3 资源简介	17
4.4 S3 系列型号对比	17
4.5 ESP32-S3 功能概述	19
4.5.1 系统和存储器.....	19
4.5.2 IO MUX 和 GPIO 交换矩阵	24
4.5.3 复位与时钟.....	28
4.5.4 芯片 Boot 控制	31
4.5.5 中断矩阵.....	33
4.6 ESP32-S3 启动流程	34
第五章 MicroPython 环境搭建.....	37
5.1 开发方式的选择.....	37
5.2 开发系统的选择与环境搭建.....	38
5.2.1 开发系统的选择.....	38
5.2.2 环境搭建.....	38
5.2.3 USB 虚拟串口驱动安装	46

5.2.4 ESP-32S3 开发 Thonny 的基本配置	47
5.3 固件的下载与烧录.....	48
第六章 MicroPython 固件编译.....	55
6.1 搭建编译环境.....	55
6.1.1 Linux 环境搭建	55
6.1.2 安装工具链.....	60
6.1.3 ESP-IDF 环境搭建	60
6.1.4 搭建 MicroPython 开发环境.....	63
6.1.5 MicroPython 源文件结构分析.....	65
6.2 编译 ESP32-S3 固件	66
6.3 总结.....	71
第七章 MicroPython 组件扩展.....	73
7.1 组件扩展原理.....	73
7.1.1 组件扩展方式.....	73
7.1.2 组件扩展原理解析.....	74
7.2 组件扩展辅助工具.....	78
第二篇 入门篇.....	79
第八章 LED 实验.....	80
8.1 machine.Pin 类.....	80
8.2 硬件设计.....	81
1. 例程功能.....	81
2. 硬件资源.....	81
3. 原理图.....	81
8.3 软件设计.....	82
8.3.1 程序流程图.....	82
8.3.2 程序解析.....	82
8.4 下载验证.....	84
第九章 KEY 实验.....	85
9.1 独立按键简介.....	85
9.2 硬件设计.....	85
1. 例程功能.....	85
2. 硬件资源.....	85
3. 原理图.....	86
9.3 软件设计.....	86
9.3.1 程序流程图.....	86
9.3.2 程序解析.....	86
9.4 下载验证.....	87
第十章 EXIT 实验.....	88
10.1 外部中断简介.....	88
10.2 machine.Pin 类.....	88
10.3 硬件设计.....	89
1. 例程功能.....	89
2. 硬件资源.....	89
3. 原理图.....	89
10.4 软件设计.....	89

10.4.1 程序流程图.....	89
10.4.2 程序解析.....	89
10.5 下载验证.....	90
第十一章 Timer 实验	91
11.1 Timer 简介	91
11.1 machine.Timer 类.....	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.....	93
第十二章 PWM 实验	94
12.1 PWM 脉宽调制技术	94
12.2 machine.PWM 类.....	95
12.3 硬件设计.....	96
1. 例程功能.....	96
2. 硬件资源.....	96
3. 原理图.....	96
12.4 软件设计.....	96
12.4.1 程序流程图.....	96
12.4.2 程序解析.....	96
12.5 下载验证.....	97
第十三章 SPILCD 实验	98
13.1 SPI 及 LCD 介绍	98
13.1.1 SPI 介绍.....	98
13.1.2 SPI 控制器介绍.....	99
13.1.3 LCD 介绍	100
13.2 SPILCD 驱动解析	103
13.2.1 MicroPython 驱动解析.....	103
13.2.2 LCD 构造与类的方法.....	110
13.3 硬件设计.....	112
1. 例程功能.....	112
2. 硬件资源.....	112
3. 原理图.....	112
13.4 软件设计.....	112
13.3.1 程序流程图.....	112
13.3.2 程序解析.....	113
13.5 下载验证.....	113
第十四章 RTC 实验	115
14.1 machine.RTC 类	115
14.2 硬件设计.....	116
1. 例程功能.....	116
2. 硬件资源.....	116

3. 原理图.....	116
14.3 软件设计.....	116
14.3.1 程序流程图.....	116
14.3.2 程序解析.....	116
14.4 下载验证.....	117
第十五章 看门狗实验.....	118
15.1 WDT 类.....	118
15.2 硬件设计.....	118
1. 例程功能.....	118
2. 硬件资源.....	118
3. 原理图.....	118
15.3 程序设计.....	119
15.3.1 程序流程图.....	119
15.3.2 程序解析.....	119
15.4 下载验证.....	120
第十六章 SD 卡实验.....	121
16.1 SD 卡操作模块.....	121
16.1.1 uos 模块	121
16.1.2 machine.SDCard 对象	122
16.2 硬件设计.....	123
1. 例程功能.....	123
2. 硬件资源.....	123
3. 原理图.....	123
16.3 软件设计.....	124
16.3.1 程序流程图.....	124
16.3.2 程序解析.....	124
16.4 下载验证.....	130
第三篇 高级篇.....	131
第十七章 UDP 实验.....	132
17.1 network 与 socket 库的简介	132
17.1.1 network 与 socket 库	132
17.1.2 network 与 socket 库的构造与方法	133
17.2 硬件设计.....	136
1. 例程功能.....	136
2. 硬件资源.....	136
3. 原理图.....	136
17.3 软件设计.....	136
17.3.1 程序流程图.....	136
17.3.2 程序解析.....	136
17.4 下载验证.....	138
第十八章 TCPClient 实验.....	139
18.1 network 与 socket 库的简介	139
18.2 硬件设计.....	139
1. 例程功能.....	139
2. 硬件资源.....	139
3. 原理图.....	139

18.3 软件设计.....	139
18.3.1 程序流程图.....	139
18.3.2 程序解析.....	140
18.4 下载验证.....	142
第十九章 TCPServer 实验	143
19.1 network 与 socket 库的简介.....	143
19.2 硬件设计.....	143
1. 例程功能.....	143
2. 硬件资源.....	143
3. 原理图.....	143
19.3 软件设计.....	143
19.3.1 程序流程图.....	143
19.3.2 程序解析.....	144
19.4 下载验证.....	146
第二十章 BLE 实验	147
20.1 BLE 特定库	147
20.1.1 BLE 特定库简介	147
20.1.2 BLE 特定库的构造与方法	147
20.2 硬件设计.....	148
1. 例程功能.....	148
2. 硬件资源.....	148
3. 原理图.....	148
20.3 软件设计.....	149
20.3.1 程序流程图.....	149
20.3.2 程序解析.....	149
20.4 下载验证.....	151

前言

MicroPython，当今开源软件领域的热门技术，由英国剑桥大学的教授 Damien George（达米安·乔治）发明。作为计算机工程师，Damien George 的日常工作与 Python 密不可分，同时也是一位机器人项目的爱好者。某日，他突发奇想，能否利用 Python 语言来控制单片机，实现对机器人的操控呢？

Python，这款广受欢迎的脚本语言，因其语法简洁、用法简单、功能强大、容易扩展等特性，备受开发者们的青睐。其强大的社区支持、丰富的库资源、强大的网络功能和计算能力，以及与其他语言良好的兼容性，使得 Python 广泛应用于工程管理、网络编程、科学计算、人工智能、机器人、教育等多个领域。更重要的是，Python 完全开源，不受商业公司的控制和影响，完全靠社区推动和维护，因此受到越来越多开发者的青睐。

然而，早期 Python 在嵌入式系统中的应用并未得到广泛推广，主要受到硬件成本、运行性能、开发习惯等因素的限制。随着半导体技术和制造工艺的快速发展，芯片的升级换代速度日益加快，芯片的功能和存储器容量不断增强，成本逐渐降低，为 Python 在低端嵌入式系统上的应用提供了可能性。

Damien George 投入了六个月的时间研发出了 MicroPython。它使用 GNU C 进行开发，实现在微控制器上 Python3 的基本功能，具备完善的解析器、编译器、虚拟机和类库等。在保留 Python 语言主要特性的基础上，对嵌入式系统的底层进行了出色的封装，将常用功能都封装到库中，甚至为一些常用的传感器和硬件编写了专门的驱动。用户只需通过调用这些库和函数，就能快速控制 LED 小灯、舵机、多种传感器、SD 卡文件系统、UART、I2C、SPI 通信总线等实现各种功能，而不用再去研究底层外设模块的使用方法。这不但降低了开发难度，而且减少了重复开发工作，缩短了开发周期。

MicroPython 最早被应用在 STM32F4 微控制器平台上。随着社区开发者的不断努力，它逐渐被移植到 STM32L4、STM32F7、ESP8266、ESP32、CC3200、dsPIC33FJ256、MK20DX256、microbit、MSP432、XMC4700、RT8195、IMXRT 等众多硬件平台上。而且，不少开发者在不断尝试将 MicroPython 移植到更多的硬件平台上，还有更多的开发者在使用 MicroPython 做嵌入式应用，并将它们在网络上分享。

本书的撰写目的在于梳理作者在基于 ESP32S3 微控制器移植 MicroPython 的过程中总结出的一些开发规范以及一些奇思妙想，整理成文稿后可作为软件组的其他同事在更多平台上移植 MicroPython 和深入开发的参考。重点在于如何移植现有的模块。

内容简介

本书旨在为读者提供一本全面、系统的 MicroPython 和 ESP32-S3 编程指南，从入门到进阶，带领读者逐步掌握 MicroPython 和 ESP32-S3 的开发与应用。

本书分为三个部分：

第一部分：基础篇

本部分将引导读者逐步了解 MicroPython 与微控制器之间的联系，以及如何调试交互环境。此外，我们还将介绍 ESP32-S3 的基本概念以及如何搭建开发环境。通过学习本章节，读者将掌握 MicroPython 和 ESP32-S3 的基本知识和开发环境，为后续的学习和实践打下坚实的基石。

第二部分：入门篇

本部分适合有一定 MicroPython 和 ESP32-S3 编程基础的人，深入介绍 ESP32-S3 的硬件接口和应用开发基础。本部分详细介绍了 ESP32-S3 的 GPIO 接口、I2C 接口、SPI 接口以及应用开发案例等。通过本部分的学习，读者将掌握 ESP32-S3 的基本硬件接口和应用开发基础，能够进行基本的 ESP32-S3 应用程序开发。

第三部分：高级篇

本部分将基于基础篇的知识，进一步讲解 ESP32-S3 MicroPython 的高级应用，包括 WIFI、蓝牙等相关高级应用。

本书结构清晰、内容丰富、实用性强，适合 MicroPython 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

在基础篇中，我们将引领读者逐步了解 MicroPython 与微控制器之间的关联，以及如何为两者之间的交互进行环境调试。此外，我们将向您展示如何构建一个适合开发的平台，涉及安装必要的软件和驱动程序，并指导读者如何自行编译 MicroPython ESP32-S3 固件。同时，我们还将介绍如何编写 MicroPython 扩展组件，并教授您如何下载和烧录固件到 ESP32-S3 芯片中。这些内容将帮助您全面了解 MicroPython 和 ESP32-S3 的基础知识及开发环境，为用户后续的学习和实践提供稳固的基础。

本篇分为以下几个章节：

- 1, 本书学习方法
- 2, MicroPython 与微控制器
- 3, MicroPython 基础知识
- 4, ESP32-S3 基础知识
- 5, MicroPython 环境搭建
- 6, MicroPython 固件编译
- 7, MicroPython 扩展组件

第一章 本书学习方法

为了让读者能够更好地学习和使用本书，本章将介绍本书的学习方法。

本章分为如下几个小节：

- 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

为了让读者更好地学习和使用本书，我们做了以下几点考虑：

1，坚持循序渐进的思路讲解，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（《ESP32-S3 最小系统板硬件参考手册.pdf》）。

因此，读者在学习本书的时候，我们建议：先通读一遍《ESP32-S3 最小系统板硬件参考手册.pdf》，对板卡的硬件资源有个大概的了解，然后从本书的基础篇开始，再到入门篇，最后是提高篇，循序渐进，逐一攻克。

对于初学者，更是要按照以上建议的学习路线进行学习，不要跳跃式学习，因为本书中的知识是环环相扣的，如果没有掌握前面的知识，就去学习后面的知识，就会学的非常吃力。

对于已经有了一定单片机基础的读者，就可以跳跃式地学习，提高学习效率。若是遇到不懂的知识点，也得查阅前面的知识点进行巩固。

1.2 本书参考资料

本书主要参考的资料有以下两份文档：

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

《esp32-s3_technical_reference_manual_cn.pdf 技术规格书》

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

本书通过数十个实验例程为读者详细介绍了 ESP32-S3 几乎所有的功能和外设，按照难易程度以及知识结构，本书分为三大篇章：基础篇、入门篇和提高篇。

基础篇，共包含七章，涵盖了 MicroPython 简介、环境搭建、ESP32-S3 基础知识、编译固件、组件扩展等知识。这些章节内容相互关联，为学习更深入的高级知识打下了坚实的基础。

入门篇，总共二十四章，涵盖了 ESP32-S3 芯片的大多数外设及其对应的应用代码。

高级篇，总共十章，介绍了一些实用的软件例程，包括 WiFi、蓝牙等例程。

入门篇和提高篇，这部分内容占了本书的绝大部分篇幅，并且这些章节在结构上比较有共

性，一般分为四个部分，如下：

- 1, 模块解析与调用
- 2, 硬件设计
- 3, 程序设计
- 4, 下载验证

模块解析与调用，简单介绍具体章节所使用的 `michine` 特定库的类方法以及自定义的模块调用，让读者对该类接口有一个基本的了解，便于后面的程序设计。

硬件设计，包括具体章节实验例程实现的功能说明、使用到的硬件资源及其相关的硬件原理图，从而让读者清楚具体章节的实验例程要做什么？用那些硬件资源来做？这些硬件资源是如何进行连接的？便于在程序设计时编写驱动代码和应用代码。

程序设计，一般包括：程序流程图、关键代码解析、`main` 函数讲解等及部分，一点一点地介绍程序代码是怎么来的和注意事项等，从而让读者掌握整个程序代码。

下载验证，属于实践环节，在程序设计完成之后，下载并验证设计的程序是否能按照预期工作，形成一个闭环的过程。

1.4 本书代码规范

本书严格遵守 Python 规范来编写代码，这意味着代码应该遵循 Python 的语法和风格指南。以下是一些 Python 规范：

- 1, 使用有意义的变量名和函数名：变量名和函数名应该能够清晰地表达它们的含义和作用。
- 2, 使用注释：注释应该简洁明了，能够清晰地解释代码的作用。注释应该是解释代码的功能和目的，而不是解释代码本身。
- 3, 使用空格和缩进：Python 使用空格和缩进来表示代码块和语句的层次结构。正确的空格和缩进可以使代码更加清晰易读。
- 4, 使用简洁的代码：Python 鼓励使用简洁的代码，可以通过使用列表推导式、生成器表达式、`lambda` 表达式等方式来简化代码。
- 5, 遵循 PEP 8 规范：PEP 8 是 Python 的官方编码风格指南，包括代码排版、命名规范、注释规范等方面的建议。本书遵循 PEP 8 规范来编写代码。

总之，Python 规范旨在确保代码的可读性和可维护性，使代码易于理解和修改。本书严格按照 Python 规范来编写代码，可以让读者更好地掌握 Python 编程的基础和技巧。

1.5 例程资源说明

ESP32-S3 最小系统板的配套资料中，除了《00_Basic》之外，还提供了 14 个标准例程。这些例程都是基于 MicroPython 库编写的，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 最小系统板配套的例程如下表所示。

编号	实验名称	编号	实验名称
基础应用			
00	00_basic	05	05_led_pwm
01	01_led	06	06_spilcd
02	02_key	07	07 rtc
03	03_exit	08	08_wdt
04	04_timer_it	09	09_sd
WIFI、BLE 应用			
01	01_UDP	04	04_WebCAM
02	02_TCPCClient	05	05_BLE
03	03_TCPServer		

表 1.5.1 ESP32-S3 最小系统板配套的例程表

从上表可以看出，该开发板提供了基于 MicroPython 的标准例程。这些例程大多数是原创的，并具有详细注释，有利于初学者入门学习。这些例程涉及了多种应用场景，包括基础入门实验和物联网等。通过学习这些例程，开发人员可以快速掌握 ESP32-S3 最小系统板的编程和调试技巧，为开发物联网和嵌入式应用打下基础。

1.6 学习资料查找

如果读者想查找有关使用 MicroPython 进行 ESP32-S3 开发的资料，可以尝试以下方法：

1, MicroPython 开发文档

[MicroPython 开发文档](#)在其官网上提供的是 MicroPython 的通用开发文档，包括语言特性、库函数、调试等方面的信息。对于 MicroPython ESP32 开发及使用，可以参考该文档中的相关内容，如下图所示。

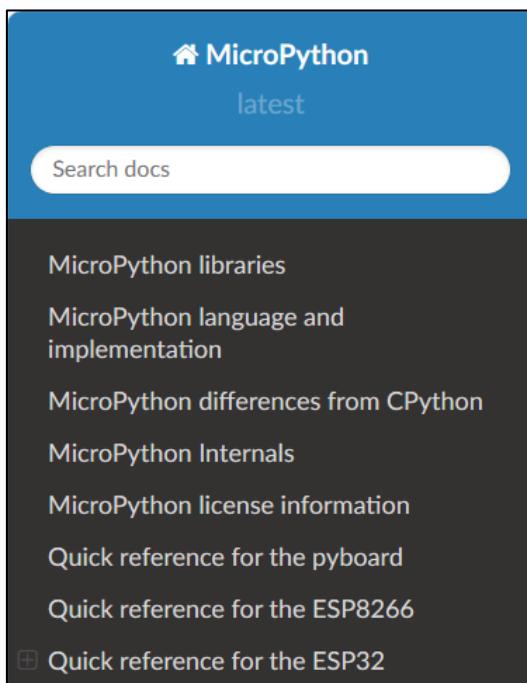


图 1.6.1 MicroPython 官网 ESP32 芯片资料

如果读者想进一步深入了解 ESP32 的 MicroPython 开发，可以点击该页面上的“Quick reference for the ESP32”，进入 ESP32 MicroPython 开发教程。

该教程是针对 ESP32 芯片及 MicroPython 开发的详细指南，包括硬件准备、MicroPython 编程、库函数使用等方面。通过该教程的学习和实践，开发者可以掌握使用 MicroPython 进行 ESP32 开发的基本技能和方法，并能更好地应用 ESP32 进行物联网应用开发。

请注意，不同版本的 ESP32 芯片和 MicroPython 可能存在差异，因此在实际开发中可能需要根据具体情况进行调整和适配。同时，也可以参考 ESP32 和 MicroPython 的官方文档和社区资源，获取更详细和最新的开发指南和技术支持。

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 1.6.4 正点原子文档中心（部分截图）

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、产品等的详细资料下载链接。

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 MicroPython 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，帮助初学者逐步掌握 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 MicroPython 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 MicroPython 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，读者可以找到大量与 ESP32-S3 和 MicroPython 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

学习 ESP32-S3 的三点建议：

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 数据手册》、《esp32-s3_technical_reference_manual_cn.pdf 技术规格书》和《isa-summary.pdf》是学习 ESP32 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。

第二章 MicroPython 与微控制器

本章节主要讲解 Python 与微控制器的发展历程，让读者了解 MicroPython 和微控制器芯片的缘起和发展。

本章分为如下几个小节：

- 2.1 MicroPython 缘起
- 2.2 微控制器软件
- 2.3 MicroPython 优点&应用场景
- 2.4 本章小结

2.1 MicroPython 缘起

“人生苦短，我学 Python”这句话表达了一种对学习 Python 的热情和决心。确实，Python 是一种简单易学的编程语言，它具有简洁清晰的语法和丰富的库，使得开发者可以快速地编写出高质量的代码。然而，在 MicroPython 出现之前，Python 从未在微控制器系统软件开发过程中作为主要的开发语言。

这一切在 2013 年发生了改变，Damien George 在网上发起了一项名为“Kickstarter”的众筹活动。Damien 是一名狂热的机器人程序员，他想把 Python 开发过程从使用 GB 基本容量的机器转移到 KB 基本资源微控制器平台上。最终，他将这个概念转化为一个完整的解决方案。

在机缘巧合下，许多开发者抓住了这个机会赞助 Damien 的项目，这使得他们不仅可以在微控制器上使用 Python，还可以获得 Damien 自己设计的参考硬件和早期版本。

2.2 微控制器软件

微控制器软件是指运行在微控制器上的程序代码，这些代码通常使用嵌入式 C 语言或 MicroPython 等语言编写。这些软件用于控制微控制器的输入、处理和输出，以实现特定的功能或任务。

在应用程序中，通常包含三类基本元件：输入单元、处理单元和输出单元。

1，输入单元：输入单元负责接收外部输入信号或数据，例如按键、传感器、网络通信等。这些输入可以作为程序执行的触发条件或提供程序所需的数据。

2，处理单元：处理单元是应用程序的核心部分，负责处理输入信息并根据预设逻辑更新输出信息。处理单元通常包括一系列函数、循环等，用于实现数据处理、计算和控制功能。

3，输出单元：输出单元负责将处理后的信息以适当的形式输出，例如发送信息、驱动电机等。输出单元可以是各种形式，如数字信号、模拟信号、网络通信等，取决于具体应用需求。

微控制器软件的主要任务是根据输入单元接收到的信号或数据，在处理单元中进行处理和计算，然后通过输出单元控制外部设备或实现特定功能。微控制器软件的设计和编写需要考虑特定的硬件平台、开发工具和语言环境等因素，以确保程序的正确性、可靠性和性能。下图是微控制器整体框图。

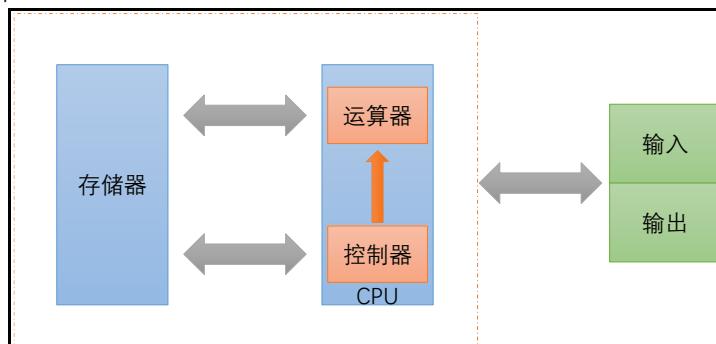


图 2.2.1 微控制器整体框图

2.2.1 什么是微控制器

微控制器是一种将微型计算机的主要部分集成在一个芯片上的单芯片微型计算机。它包含电子计算单元和逻辑单元（统称为CPU）、存储器（程序存储器和数据存储器）、I/O端口（输入/输出端口）以及很少的其他组件集成在单个芯片上。微控制器通常被嵌入在其控制的设备中，因此也称为嵌入式控制器。微控制器具有成本低、性能强大、应用广泛等特点，被广泛应用于电机控制、条码阅读器/扫描器、消费类电子、游戏设备、电话、HVAC、楼宇安全与门禁控制、工业控制与自动化和白色家电等领域。

2.2.2 为什么使用 MicroPython

MicroPython 早于微控制器的普及而诞生，它为开发者和用户带来了以下好处：

首先，MicroPython 降低了微控制器开发的门槛。相对于其他传统的开发语言，MicroPython 的语法更加简单易懂，使得初学者可以快速上手并实现基本功能。同时，MicroPython 在处理复杂任务时也表现出强大的能力，满足了工业应用的需求。

其次，MicroPython 提供了更快的开发反馈机制。通过交互式解释器（REPL），开发者可以实时输入命令并查看结果，快速调整代码并进行测试。这种开发方式减少了重复的编码、编译和上传过程，提高了开发效率。

最后，MicroPython 充分利用了 Python 的丰富资源。Python 作为一种成熟的编程语言，拥有庞大的样例代码库和知识积累。这使得开发者可以更快、更容易地实现特定功能，例如处理字符串、解析 JSON 等。在 MicroPython 的世界中，这些工具同样适用，并比使用 C++ 等语言更加便捷。

综上所述，MicroPython 为早于它诞生的微控制器普及带来了诸多好处，包括降低开发门槛、提高开发效率和利用丰富的资源库等。这些优势使得 MicroPython 成为微控制器开发的重要选择之一。

2.3 MicroPython 优点&应用场景

MicroPython 的优势主要包括以下几点：

1，易学易用：MicroPython 的语法与 Python 类似，易于学习和使用。对于已经熟悉 Python 语言的开发者来说，MicroPython 可以让他们快速上手嵌入式系统和物联网技术的开发。

2，开发效率高：MicroPython 具有丰富的库和模块，可以方便地实现各种功能，缩短了开发周期。同时，MicroPython 还支持多种开发工具和平台，方便开发者进行开发和调试。

3，适合嵌入式系统的资源限制：MicroPython 经过优化，适合在资源有限的嵌入式系统中运行。它占用内存小、运行效率高，能够满足嵌入式系统在实时性和功耗等方面的要求。

4，社区支持：MicroPython 有一个活跃的社区，提供了大量的学习资源和示例代码，方便开发者学习和交流。同时，MicroPython 还在不断发展和完善，能够满足嵌入式系统和物联网技术开发的多种需求。

MicroPython 的应用场景非常广泛，主要包括以下几个方面：

1，物联网设备开发：MicroPython 可以用于各种物联网设备开发，如智能家居、农业、物流等。在这些领域，MicroPython 可以通过串口或 WiFi 连接进行编程和传输，控制各种传感器、执行各种操作和处理各种数据。

2，嵌入式系统开发：MicroPython 是一种嵌入式 Python 编程语言，适用于各种嵌入式系统的开发，如微控制器、树莓派、智能手表等。它可以在这些设备上运行，并通过 I/O 端口与外部设备进行通信和控制。

3，自动化控制：MicroPython 可以用于各种自动化控制领域，如工业、家庭自动化等。在这些领域，MicroPython 可以通过控制各种执行器、传感器等设备，实现自动化控制和智能管理。

4，教育领域：MicroPython 可以用于教育领域，让学生更容易地了解和学习嵌入式系统和物联网技术。通过 MicroPython，学生可以快速实现各种实验和项目，提高他们的实践能力和创新思维。

2.4 本章小结

本章节主要介绍了 MicroPython 的缘起、微控制器软件的基本概念、MicroPython 的优势和应用场景。首先，介绍了 MicroPython 如何改变了微控制器开发的语言格局，使得 Python 成为这类系统的重要编程语言。其次，介绍了微控制器软件的基本组成和特点，为理解 MicroPython 在其中的应用打下基础。再者，详述了 MicroPython 相对于其他语言的优点，如降低开发门槛、提供更快的开发反馈和利用丰富的 Python 资源。最后，列举了 MicroPython 在教育、物联网、微控制器应用、实时系统、嵌入式系统开发、可穿戴设备、移动设备和科学研究等多个领域的广泛应用。总之，MicroPython 凭借其独特的优势和广泛的应用场景，已经成为嵌入式系统开发的重要选择之一。

第三章 MicroPython 基础知识

虽然 MicroPython 易于使用，但对于初学者（特别是 Windows 平台下的开发者）来说，为了避免因平台差异带来的影响，更好地学习和使用 MicroPython，我们还是需要做一些准备工作。这些准备工作包括了解 MicroPython 的基础知识和使用方法、交互式环境、常见问题及常用的开发工具等。通过这些准备，我们可以更快地掌握 MicroPython 的开发技巧，更好地发挥其优势。

本章分为以下几个小节：

- 3.1 Python3 与 MicroPython 关联
- 3.2 MicroPython 系统结构
- 3.3 MicroPython 的 REPL 交互式环境

3.1 Python3 与 MicroPython 关联

本书并非专注于 Python3 语言的指南，因此建议读者通过专门的书籍或在线教程来学习 Python 相关的知识。值得庆幸的是，Python 语言以其简单易学的特点而著称，即使之前没有接触过 Python，大部分读者也能在几天内初步掌握。在此，我们假设读者已经对 Python 语言有了初步的了解。

读者即使没有学过 Python 语言，掌握起来也是非常快。Python3 的参考资料非常多，这里作者推荐了两个免费学习 Python 网站：

- 1, 菜鸟教程
- 2, 黑马程序员

为了更好地帮助初学者，特别是对 Python 还不熟悉的读者，让大家可以更好地理解和使用 MicroPython，下面列出了几个基本要素：

- 1, MicroPython 的语法是完全基础 Python3 的，使用者需了解 Python3 语言的基础知识。
- 2, Python3 语言是使用缩进来表示代码的层次（如果缩进不正确，会导致语法错误或者运行错误），而不是用大括号。缩进可以使用 TAB 键或者空格键，但 TAB 最好设置为 4 个空格。
- 3, 位操作和 C 语言一样。
- 4, 逻辑操作使用 “and”、“or” 和 “^”，而不是 C 语言的 “||”、“&&”。
- 5, 在控制流语句（for、if...else...、match...case）中使用冒号。
- 6, 注意除法的区别，一个除号 “/” 代表浮点计算。两个除号 “//” 代表整数除法。

3.2 MicroPython 系统结构

一个 MicroPython 系统的典型构造如下图：它是由微控制器（系统底层硬件）、MicroPython 固件和用户程序三大部分组成。其中，硬件和 MicroPython 固件是最基础且相对不变的部分，而用户程序则可以随时变更，可以存放多个用户程序到系统中，随时调用或切换，这是使用 MicroPython 的一个特色。

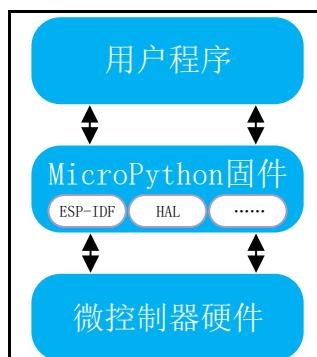


图 3.2.1 典型的 MicroPython 系统架构

微控制器硬件就如同一个没有安装操作系统的计算机，只有下载了程序后才能实现其他的

功能。MicroPython 的功能就像嵌入式系统的操作系统（它不同于 FreeRTOS、ucOS 这样的实时系统，用户程序不能单独修改，因为系统和用户程序是一体的，需要编译后运行）。只有先安装了 MicroPython 系统（固件），才能运行各种 MicroPython 程序。

目前 MicroPython 源代码支持不同微控制器平台，这需要开发者自己编译对应 MCU 的 MicroPython 固件，并将固件下载到微控制器中才能运行 MicroPython。

3.3 MicroPython 的 REPL 交互式环境

REPL 是 Read-Evaluate-Print Loop（读取-求值-输出循环）的缩写。许多编程语言都带有某种形式的 REPL，它们就像是一个小型 Shell，使得程序员能够与解释器（或内核）进行交互，方便地输入各种命令，观察运行状态，因此在程序调试时能够发挥非常大的作用。

Python 语言的 REPL 功能非常强大，而 MicroPython 虽然是一个缩减版的 Python，但它的 REPL 功能同样强大。通过 REPL 交互环境，我们可以访问 pyboard、输入程序、测试代码、查找问题、查看帮助、查看磁盘文件等。由于 MicroPython 是面向嵌入式应用的，因此 MicroPython 的 REPL 与 Python 的标准 REPL 相比，还存在一些差异，例如快捷键不同，还提供了额外的功能和用法。如果能够熟练掌握这些功能，将有助于我们更好地使用 MicroPython。

3.3.1 常用的终端

在 MicroPython 开发中，我们通常使用串口终端软件和 MicroPython 的 REPL（Read-Eval-Print Loop）进行交互，发送命令。通过串口终端软件，我们可以方便地在 REPL 中输入代码，运行和调试程序，以及打印输出结果。在开发 MicroPython 程序时，掌握串口终端软件的使用是非常必要的，因为它可以帮助我们更高效地进行开发和调试。

需要注意的是，不建议使用 Windows 下的串口调试助手等类似软件进行一般的串口调试和发送数据。这些软件通常不支持输入命令，也不支持粘贴功能，无法与 REPL 进行交互操作，因此不适合一般的开发需求。建议使用专业的串口终端软件，例如 PuTTY、Minicom 等，这些软件通常具有更丰富的功能和更好的操作体验。

另外，本书籍使用的是 Thonny 软件中的 Shell 交互窗口，这个交互窗口可以与 MicroPython 的 REPL（Read-Eval-Print Loop）进行交互，发送命令。该软件安装流程及使用，请用户参考第四章的内容。

3.3.2 REPL 的快捷键

在 REPL 提供了不少快捷键，使用这些快捷键可以有效减少按键的次数，提供代码输入效率。REPL 常用的快捷键如下表所示。

键盘按键	功能
上下方向键	切换以前输入的命令
左右方向键	移动光标，编辑当前命令行中的输入的内容
Tab 键	代码补全。如果只有一种操作，会自动补全；在有多种操作时，列出可能的选项让用户参考
CTRL+D	在空命令行下按下 CTRL+D 键，将执行软件复位功能

表 3.3.2.1 RERL 快捷键列表

按下 CTRL+D，将执行软件复位，复位完成后显示以下内容：

```
MPY: soft reboot
MicroPython v1.22.0-preview.31.g3883f2948.dirty on 2023-11-15; Generic ESP32S3
module with Octal-SPIRAM with ESP32S3
Type "help()" for more information.
>>>
```

Tab 键在输入代码时是最常用的功能之一，它可以帮助我们快速补全代码，提高输入效率，比如在 Thonny 工具的终端 Shell 中，输入“ma”，然后按下 Tab 键，它就会提示相关信息，如下图所示。

```

MPY: soft reboot
MicroPython v1.22.0-preview.31.g3883f2948.dirty on 2023-11-15;
Type "help()" for more information.

>>> ma
    map(
        max(

```

map(func, iter1, ...) -> iterator[S]
map(func, iter1, iter2, ...) -> iterator[S]
map(func, iter1, iter2, iter3, ...) -> iterator[S]
map(func, iter1, iter2, iter3, iter4, ...) ->
iterator[S]
map(func, iter1, iter2, iter3, iter4, iter5, ...) ->
iterator[S]

图 3.3.2.1 Tab 键列出可能的选项

3.3.3 help()函数的作用

在 Python 中，我们可以使用 `help()` 函数来获取关于特定对象、模块、函数或类的详细帮助信息。在 MicroPython 中，这个功能同样得到了支持。在 REPL（交互式命令行）环境下，直接输入 `help()`，将会显示一个基本的帮助界面。这个界面提供了关于 MicroPython 中 `machine` 特定库以及 REPL 用法的信息，可以帮助我们更好地了解基本的命令和函数。

Welcome to MicroPython on the ESP32!

```

For online docs please visit http://docs.micropython.org/

For access to the hardware use the 'machine' module:
# machine 特定库的示例
import machine
pin12 = machine.Pin(12, machine.Pin.OUT)
pin12.value(1)
pin13 = machine.Pin(13, machine.Pin.IN, machine.Pin.PULL_UP)
print(pin13.value())
i2c = machine.I2C(scl=machine.Pin(21), sda=machine.Pin(22))
i2c.scan()
i2c.writeto(addr, b'1234')
i2c.readfrom(addr, 4)

Basic WiFi configuration:

import network
sta_if = network.WLAN(network.STA_IF); sta_if.active(True)
sta_if.scan()                                     # Scan for available access points
sta_if.connect("<AP_name>", "<password>")      # Connect to an AP
sta_if.isconnected()                            # Check for successful connection
# REPL 快捷键功能描述
Control commands:
CTRL-A      -- on a blank line, enter raw REPL mode
CTRL-B      -- on a blank line, enter normal REPL mode
CTRL-C      -- interrupt a running program
CTRL-D      -- on a blank line, do a soft reset of the board
CTRL-E      -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)
For a list of available modules, type help('modules')
```

从上述可知，我们可通过 `help`（模块名或函数名）查看更详细的帮助，如查看 `machine` 模块中的 `Pin` 类接口等详细帮助：

```

>>> import machine
>>> help(machine.Pin)
object <class 'Pin'> is of type type
  init -- <function>
  value -- <function>
  off -- <function>
  on -- <function>
  irq -- <function>
  board -- <class 'board'>
```

```

IN -- 1
OUT -- 3
OPEN_DRAIN -- 7
PULL_UP -- 2
PULL_DOWN -- 1
IRQ_RISING -- 1
IRQ_FALLING -- 2
WAKE_LOW -- 4
WAKE_HIGH -- 5
DRIVE_0 -- 0
DRIVE_1 -- 1
DRIVE_2 -- 2
DRIVE_3 -- 3
>>>

```

首先，我们使用 import 导入 machine 模块，然后可以通过 help()函数来获取 machine 模块中 Pin 类的接口等相关帮助信息。根据上述内容可知，<function>代表函数，<class>代表类，而其他则代表宏的数值。

help()函数可以一级一级深入查看，如：

```

>>> help(machine.Pin.init)
object <function> is of type function
>>> help(machine.Pin.value)
object <function> is of type function
>>>

```

上述内容可知，machine 特定库的 Pin 类中的 init 和 value 方法是一个函数类型。如果开发者能够灵活利用 help()函数，就可以查看大部分函数的基本用法和很多常量定义，甚至不需要看手册就能够知道该接口的用法。

3.3.4 查看模块包含的变量和函数

在 Python 中，使用 dir()函数可以查看一个模块内部包含的所有变量和函数。同样，MicroPython 也支持这个函数。通过使用 dir()函数，我们可以快速查看当前系统已经导入了哪些模块：

```

>>> import machine
>>> dir()
['__name__', 'os', '__thonny_helper', 'gc', 'bdev', 'machine']
>>>

```

在 MicroPython 中，我们首先需要导入 machine 模块，然后使用 dir()函数来查看系统已经导入了哪些模块。在系统信息提示中，我们可以看到已经包含了 machine 模块的信息。

也可以通过 dir()函数查看模块内部的变量和函数，与 help()函数一样，dir()函数也可以深入到模块内部：

```

>>> import machine
>>> dir()
['__name__', 'os', '__thonny_helper', 'gc', 'bdev', 'machine']
>>> dir(machine)
['__class__', '__name__', '__dict__', 'ADC', 'ADCBlock', 'DEEPSLEEP',
'DEEPSLEEP_RESET', 'EXT0_WAKE', 'EXT1_WAKE', 'HARD_RESET', 'I2C', 'I2S',
'PIN_WAKE', 'PWM', 'PWRON_RESET', 'Pin', 'RTC', 'SDCard', 'SLEEP', 'SOFT_RESET',
'SPI', 'Signal', 'SoftI2C', 'SoftSPI', 'TIMER_WAKE', 'TOUCHPAD_WAKE', 'Timer',
'TouchPad', 'UART', 'ULP_WAKE', 'WDT', 'WDT_RESET', 'bitstream', 'bootloader',
'deepsleep', 'dht_readinto', 'disable_irq', 'enable_irq', 'freq', 'idle',
'lightsleep', 'mem16', 'mem32', 'mem8', 'reset', 'reset_cause', 'sleep',
'soft_reset', 'time_pulse_us', 'unique_id', 'wake_reason']
>>>

```

作者希望开发者善于利用 help()和 dir()函数，在调试与输入代码时，更够帮助我们查看内部变量和函数名称，了解模块的功能和用法。

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 MicroPython 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 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

4.1 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(S2)/双核(S3)	单核	单核	单/双核
时钟频率	240MHz	120MHz	96MHz	80~240MHz
引出编程 IO	43~45	14	19	25
神经网络加速	有	无	无	无
通讯协议	2.4G\Wi-Fi\BLE	2.4G\Wi-Fi\BLE	BLE	2.4G\Wi-Fi\BLE
SRAM(KB)	320~512	272~512	320	520
ROM(KB)	128~384	348~576	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，在通讯协议方面，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](#)，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。

The screenshot shows the ESP Product Selector interface with the following sections:

- ① 工具选择**: Shows the main navigation bar with 'ESP Product Selector' and tabs for '产品选型' and '产品对比'.
- ② 功能筛选**: Shows a sidebar with dropdown menus for filtering by '型号' (Model), 'Wi-Fi', '蓝牙', '工作温度' (Working Temperature), 'Thread/Zigbee', '类别' (Category), '系列' (Series), '产品状态' (Product Status), '单/双核' (Single/Dual Core), '天线' (Antenna), '封装' (Packaging), '存储' (Storage), and '外设' (Peripherals). A red box highlights this section.
- ③ 筛选结果的选择**: Shows a list of 237 items with two filter buttons: '芯片/模组' (Chip/Moudle) and '开发板' (Development Board). A red box highlights this section.
- ④ 筛选结果**: Shows a table with three rows of results. The first row is highlighted with a red box. The columns are: Index, Name, MPN, and Marketing Status.

	Index	Name	MPN	Marketing Status
<input type="checkbox"/>	1	ESP32-S3	ESP32-S3	Mass Production
<input type="checkbox"/>	2	ESP32-S3	ESP32-S3R2	Mass Production

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 2 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/BLUE	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。

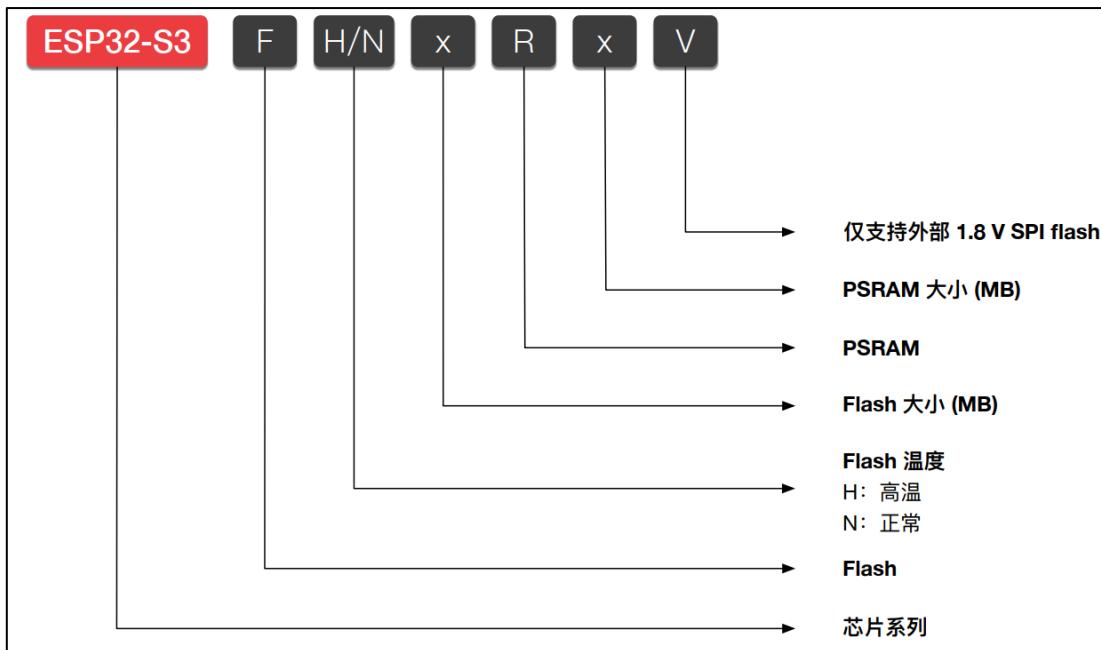


图 4.4.1 ESP32-S3 系列芯片命名规则

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。

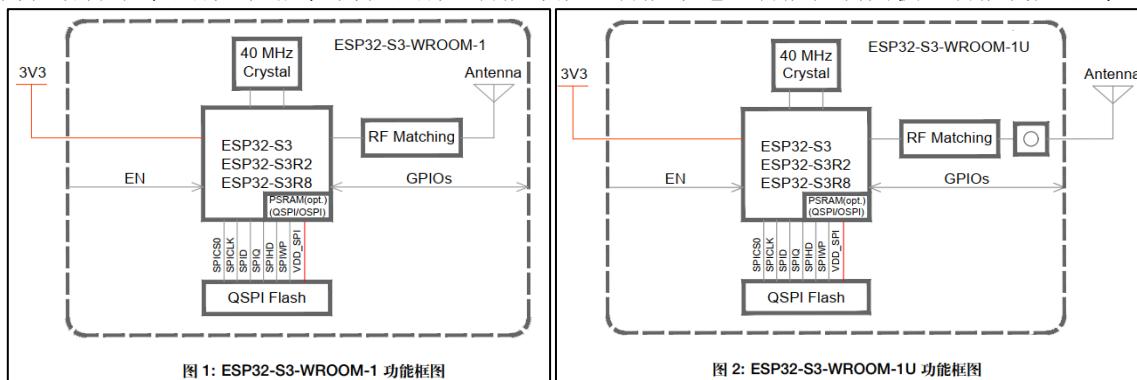


图 4.4.3 ESP32-S3-WROOM-1 和 ESP32-S3-WROOM-1U 的功能框图

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（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。

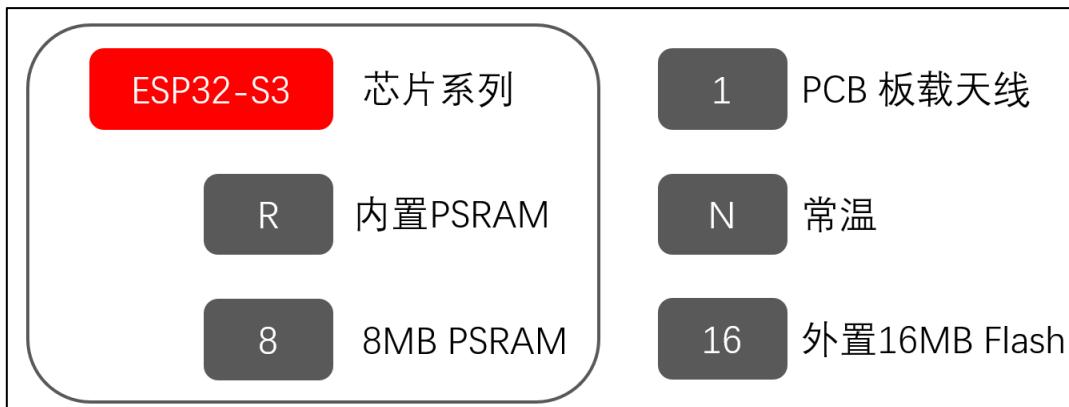


图 4.4.3 模组的命名规则

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 最小系统板是以 ESP32-S3-WROOM-1-N16R8 模组（非乐鑫原厂的模组，实际上是正点原子 ATK-MWS3S 模组，它是根据乐鑫官方 ESP32-S3-WROOM-1-N16R8 模组参考设计的。这种模组与乐鑫官方模组之间实现了 P2P 兼容）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。

下图是 ESP32-S3 地址空间映射结构图，阐述了内部存储器地址空间映射、外部存储器地址空间映射和模块/外设地址映射的系统结构图，以及 GDMA 与各部分的联系示意图。

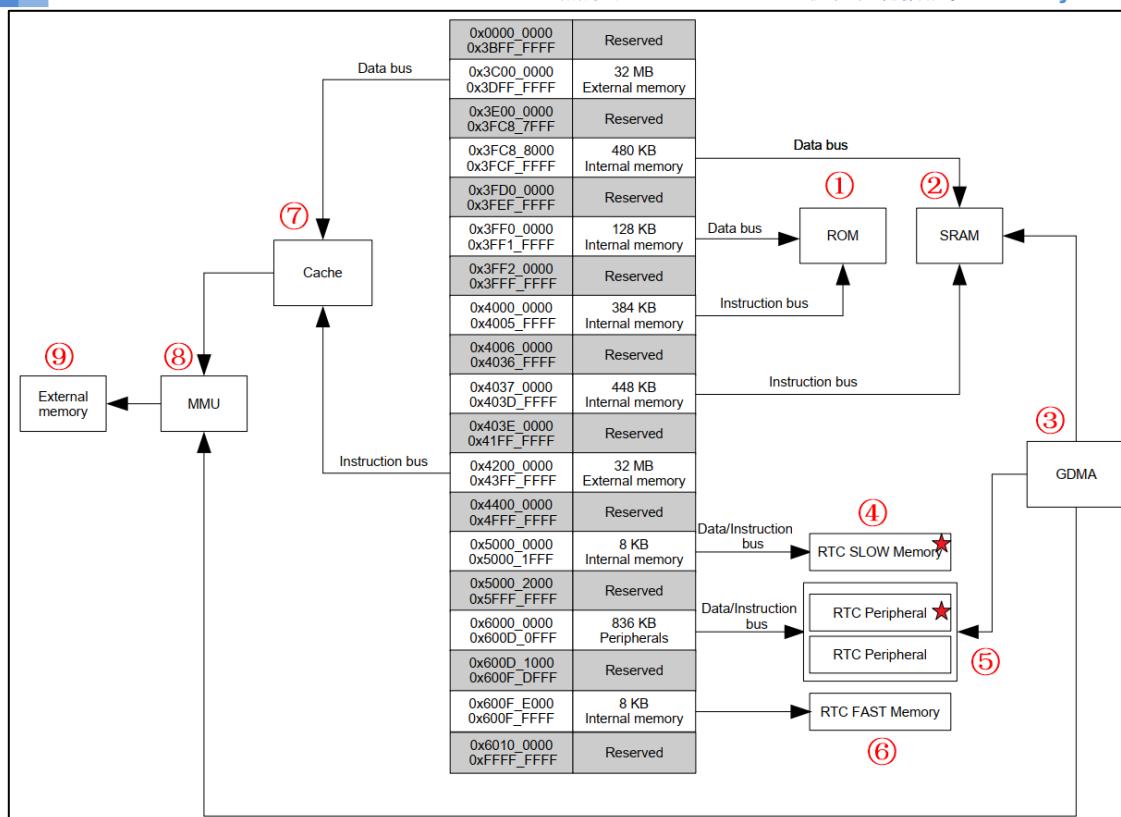


图 4.5.1 系统结构与地址映射结构

上图中，灰色背景的地址空间不可用，红色五角星表示对应存储器和外设可以被协调器访问。由于 ESP32-S3 系统是由两个哈佛结构 Xtensa® LX7 CPU 构成，这两个 CPU 能够访问的地址空间范围是完全一致的。上图中，地址 0x40000000 以下部分属于数据总线的地址范围；地址 0x40000000~4FFFFFFF 部分位指令总线的地址范围，其他是数据总线与指令总线的地址范围，即内部存储器、外部存储器和外设等映射的内存地址。

CPU 的数据总线与指令总线都为小端序（将多字节数据的低位放在较小的地址处，高位放在较大的地址处）。CPU 可以通过数据总线进行单字节、双字节、4 字节、16 字节的数据访问。CPU 也可以通过指令总线进行数据访问，但必须是 4 字节对齐方式；非对齐数据访问会导致 CPU 工作异常。CPU 的工作如下：

- ① 通过数据总线与指令总线直接访问内部存储器。
- ② 通过 Cache 直接访问映射到地址空间的外部存储器。
- ③ 通过数据总线直接访问模块/外设。

系统中部分内部存储器与部分外部存储器既可以被数据总线访问也可以被指令总线访问，这种情况下，CPU 可以通过多个地址访问到同一目标。

4.5.1.1 内部存储器

图 4.5.1 中的①、②、④和⑥部分组成 ESP32-S3 内部存储器。

上图①：Internal ROM (384KB) 是只读存储器、不可编程，用来存放系统底层的固件（程序指令和一些只读数据）。

上图②：Internal SRAM (512 KB) 是易失性存储器，可以快速响应 CPU 的访问请求，通常只需一个 CPU 时钟周期。其中，SRAM 的一部分可以被配置成外部存储器访问的缓存（Cache），但这种情况下无法被 CPU 访问；另外，某些部分只可以被 CPU 的指令总线访问；某些部分只可以被 CPU 的数据总线访问；还有某些部分既可被 CPU 的指令总线访问，也可被 CPU 的数据总线访问。

上图④和⑥：RTC Memory (16 KB) RTC 存储器以静态 RAM (SRAM) 方式实现，因此也是易失性存储器。但是，在 deep sleep 模式下，存放在 RTC 存储器中的数据不会丢失。其中，

RTC FAST memory (8 KB) 只可以被 CPU 访问，不可以被协处理器访问，通常用来存放一些在 Deep Sleep 模式下仍需保持的程序指令和数据。而 RTC SLOW memory (8KB) 既可以被 CPU 访问，又可以被协处理器访问，因此通常用来存放一些 CPU 和协处理器需要共享的程序指令和数据。

注意：所有的内部存储器都接受权限管理。只有获取到访问内部存储器的访问权限，才可以执行正常的访问操作，CPU 访问内部存储器时才可以被响应。关于权限管理的更多信息，请参考《esp32-s3_technical_reference_manual_cn.pdf》章节 15 权限控制 (PMS)。

4.5.1.2 外部存储器

图 4.5.1 中的⑦、⑧和⑨可见。CPU 借助高速缓存 (Cache) 来访问外部存储器。Cache 将根据内存管理单元 (MMU) 中的信息把 CPU 指令总线或数据总线的地址变换为访问片外 flash 与片外 RAM 的实地址。经过变换的实地址所组成的实地址空间最大支持 1 GB 的片外 flash 与 1 GB 的片外 RAM。前面我们讨论知道，ESP32-S3 采用双核共享 ICache 和 DCache 结构，以便当 CPU 的指令总线和数据总线同时发起请求时，也可以迅速响应。当双核同时访问 ICache 时，系统会做以下判断，如下图所示。

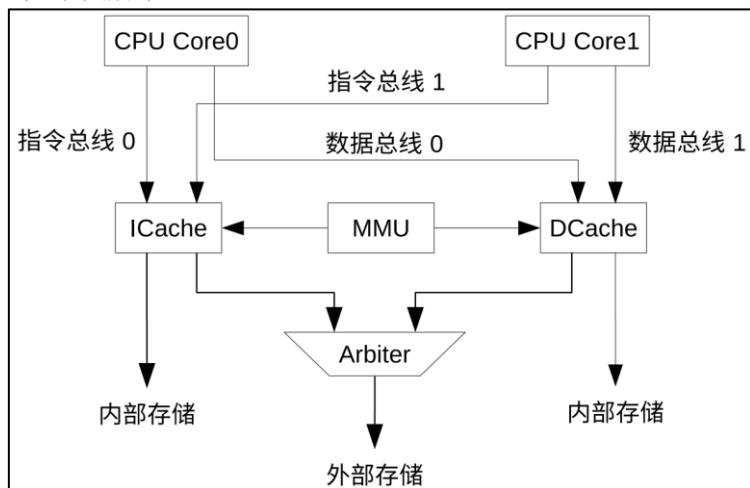


图 4.5.1.2.1 Cache 系统框图

当两个核的指令总线同时访问 ICache/DCache 时，由仲裁器决定谁先获得访问 ICache/DCache 的权限。当 Cache 缺失（处理器所要访问的存储块不在高速缓存中的现象）时，Cache 控制器会向外部存储器发起请求，当 ICache 和 DCache 同时发起外部存储器请求时，由仲裁器决定谁先获得外部存储器的使用权。

①：ICache 的缓存大小可配置为 16 KB 或 32KB，块大小可以配置为 16B 或 32B，当 ICache 缓存大小配置为 32KB 时禁用 16B 块大小模式。

②：DCache 的缓存大小可配置为 32 KB 或 64 KB，块大小可以配置为 16B、32B 或 64B，当 DCache 缓存大小配置为 64 KB 时禁用 16B 块大小模式。

返回到图 4.5.1，外部存储器通过高速缓存 (Cache)，ESP32-S3 一次最多可以同时访问 32MB 的指令总线地址空间和 32MB 的数据总线地址空间。32 MB 的指令总线地址空间，通过指令缓存 (ICache) 以 64 KB 为单位映射到片外 flash 或片外 RAM，支持 4 字节对齐的读访问或取指访问，而 32 MB 的数据总线地址空间，是通过数据缓存 (DCache) 以 64 KB 为单位映射到片外 RAM，支持单字节、双字节、4 字节、16 字节的读写访问。这部分地址空间也可以用作只读数据空间，映射到片外 flash 或片外 RAM。

下表列出了访问外部存储器时 CPU 的数据总线和指令总线与 Cache 的对应关系。

总线类型	边界地址		容量 (MB)	目标
	低位地址	高位地址		
数据	0x3C000000	0x3DFFFFFF	32	DCache
指令	0x42000000	0x43FFFFFF	32	ICache

表 4.5.1.2.1 外部存储器地址映射

同样，想要操作外部存储器的读写，需获取访问权限，CPU 访问外部存储器时才能被响应。关于权限管理的更多信息，请参考《esp32-s3_technical_reference_manual_cn.pdf》15 权限控制（PMS）。

4.5.1.3 模块/外设

图 4.5.1 中的⑤就是模块/外设地址空间地址，CPU 就是通过该空间地址来访问模块/外设的。下表是模块/外设地址空间的各段地址与其能访问到的模块/外设映射关系。

目标	边界地址		容量 (MB)	说明
	低位地址	高位地址		
UART0	0x60000000	0x60000FFF	4	UART0 地址
保留	0x60001000	0x60001FFF		
SPI 控制器 1	0x60002000	0x60002FFF	4	SPI1 地址
SPI 控制器 2	0x60003000	0x60003FFF	4	SPI2 地址
GPIO	0x60004000	0x60004FFF	4	GPIO 地址
保留	0x60005000	0x60006FFF		
eFuse 控制器	0x60007000	0x60007FFF	4	eFuse 地址
低功耗管理	0x60008000	0x60008FFF	4	低功耗地址
IO MUX	0x60009000	0x60009FFF	4	IO MUX 地址
保留	0x6000A000	0x6000EFFF		
其他外设的地址，请参考《esp32-s3_technical_reference_manual_cn.pdf》技术手册表 4-3				
World 控制器	0x600D0000	0x600D0FFF	4	World 地址

表 4.5.1.3.1 模块/外设地址空间映射部分表

从上表可以得知，要操作外设的寄存器，首先需要知道该外设的首地址。然后，我们可以使用一些底层的编程语言，如 C 语言或汇编语言，来编写程序以设置外设寄存器的值，从而控制外设的行为。例如，通过设置 GPIO 寄存器的值，我们可以控制某个 LED 灯的亮灭；同样地，设置 UART 寄存器的值可以用来发送和接收数据。

与内部存储器和外部存储器访问类似，CPU 要想访问某一个模块/外设，需要先获取该模块/外设的访问权限，否则访问将不会被响应。关于权限管理的更多信息，请参考《esp32-s3_technical_reference_manual_cn.pdf》章节 15 权限控制（PMS）。

4.5.1.4 通用 GDMA 控制器

通用直接存储访问（General Direct Memory Access, GDMA）用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。软件可以在无需任何 CPU 操作的情况下通过 GDMA 快速搬移数据，从而降低了 CPU 的工作负载，提高了效率。ESP32-S3 的 GDMA 控制器采用 AHB 总线架构，以字节为单位进行数据传输，支持软件编程控制传输数据量，支持链表传输，同时支持访问内部 RAM 时的 INCR burst 传输。其能够访问的最大内部 RAM 地址空间为 480 KB，最大外部 RAM 地址空间为 32 MB。该控制器包含 5 个接收通道和 5 个发送通道，每个通道都可以访问内部和外部 RAM，并且支持可配置的外设选择。最后，GDMA 控制器采用固定优先级及轮询仲裁机制来管理通道间的传输。

正如前文所述，GDMA 共有 10 个独立的通道，其中包括 5 个接收通道和 5 个发送通道。这 10 个通道被支持 GDMA 功能的外设所共享，也就是说用户可以将通道分配给任何支持 GDMA 功能的外设。这些外设包括 SPI2、SPI3、UHCI0、I2S0、I2S1、LCD/CAM、AES、SHA、ADC 和 RMT 等。根据图 4.3.1 的③所示的连接关系再一次验证了，这些外设都可以使用 GDMA 传输数据。此外，每个 GDMA 通道都具备访问内部 RAM 或外部 RAM 的能力，这使得 ESP32-S3 在处理复杂的数据传输任务时具有显著优势。

下图是 GDMA 功能模块和 GDMA 通道示意图。

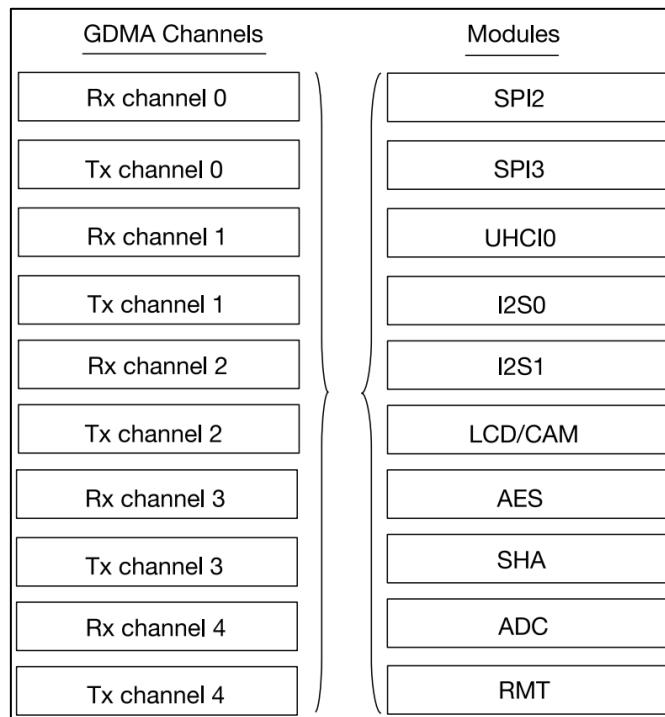


图 4.5.1.4.1 具有 GDMA 功能的模块和 GDMA 通道示意图

从上图可知，每一个外设都可以使用任意一条通道进行数据传输。然而，这些通道分为不同的类型。当使用 GDMA 接收数据时，可以选择任意的 RXn 通道（n:0~4）；相反，当使用 GDMA 发送数据时，则需要选择任务的 TXn 通道（n:0~4）。这种通道的分类和选择方式使得数据传输更加高效和灵活。

ESP32-S3 中有 11 个外设/模块可以和 GDMA 联合工作，如下图所示。其中的 11 根竖线依次对应这 11 个具有 GDMA 功能的外设/模块，横线表示 GDMA 的某一个通道（可以是任意一个通道），竖线与横线的交点表示对应外设/模块可以访问 GDMA 的某一个通道。同一行上有多个交点则表示这几个外设/模块不可以同时开启 GDMA 功能。

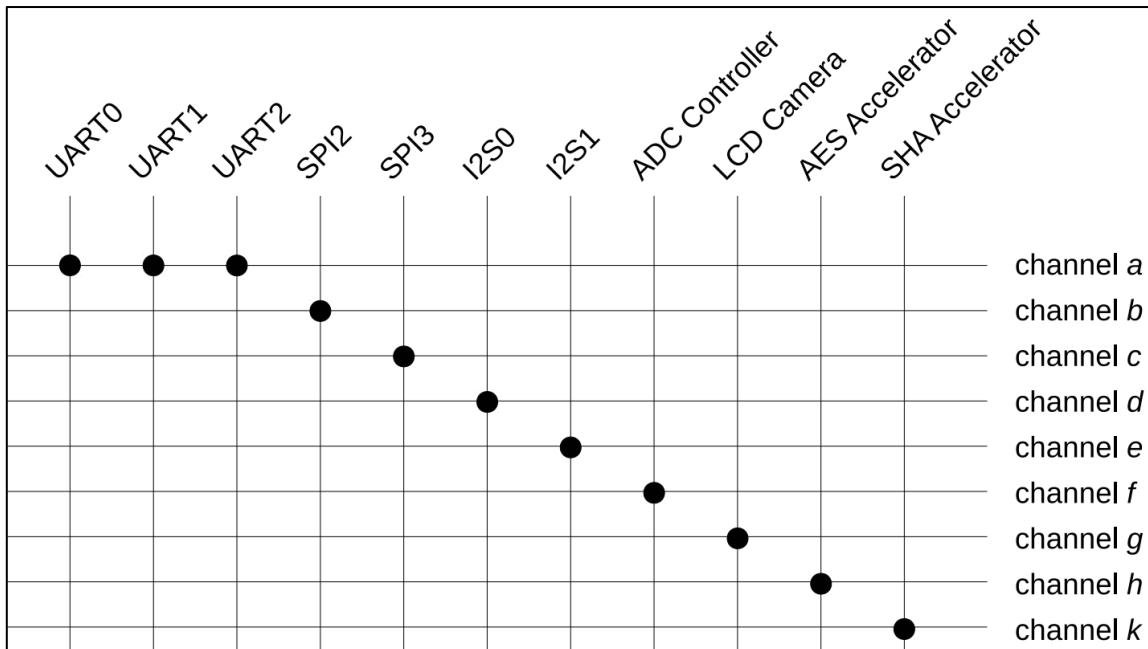


图 4.5.1.4.2 具有 GDMA 功能的外设

具有 GDMA 功能的模块/外设通过 GDMA 可以访问任何 GDMA 可以访问到的存储器。更多关于 GDMA 的信息，请参考《esp32-s3_technical_reference_manual_cn.pdf》章节 3 通用 DMA 控

制器 (GDMA)。

与前面小节一样，当使用 GDMA 访问任何存储器时，都需要获取对应的访问权限，否则访问将会失败。

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又可作为输出管脚。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	SPICSO 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 最小系统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。

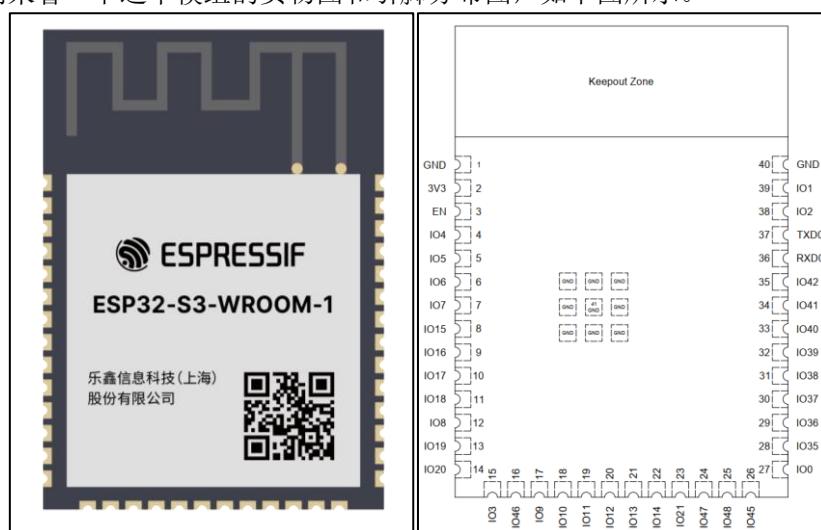


图 4.5.2.2 ESP32-S3-WROOM-1 实物图和引脚分布图

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,FSPID,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	SPIIDQS\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

上表是 ESP32-S3-WROOM-1-N16R8 模组的管脚定义，下面作者根据这个表格来讲解 GPIO

交互矩阵及 IO MUX 复用的知识。

下图为 GPIO 交换矩阵、IO MUX 和 RTC IO MUX 将信号引入外设和引出至管脚的具体过程。

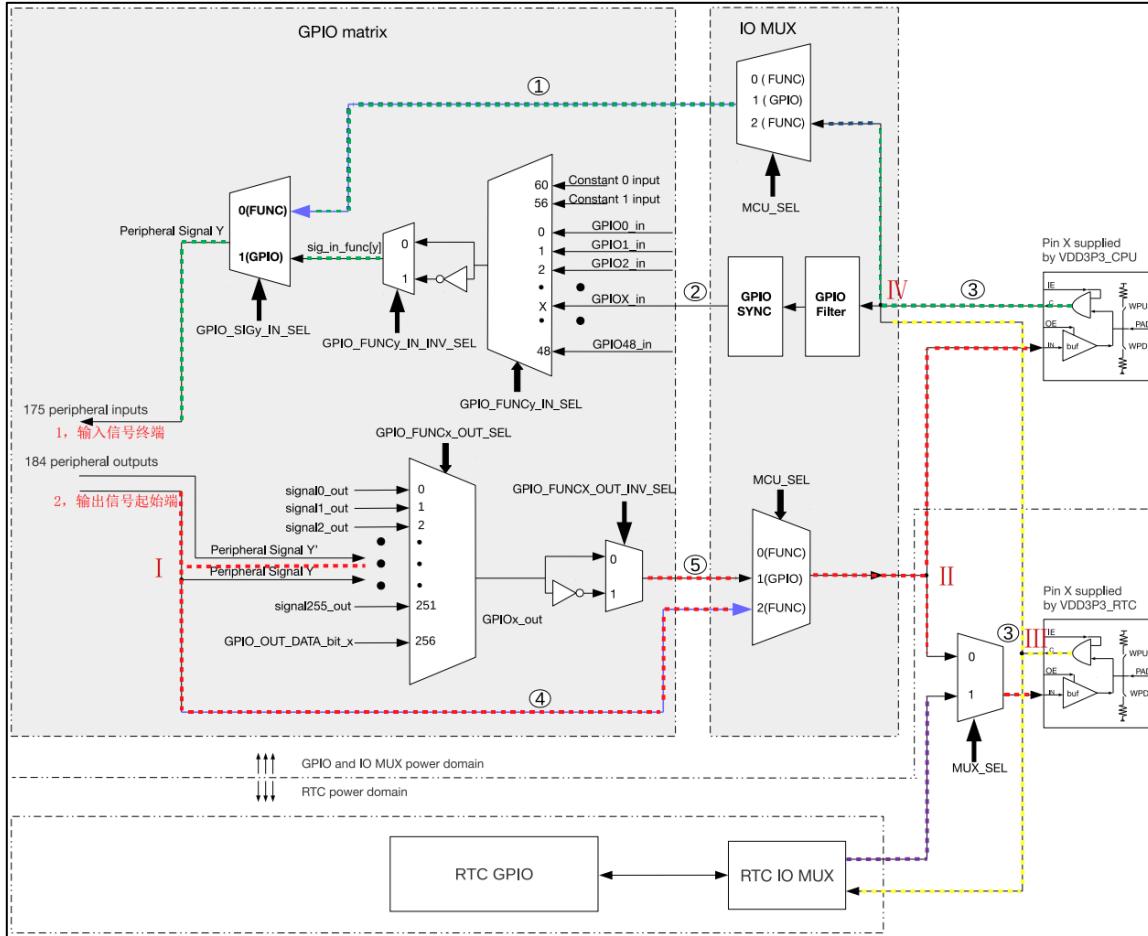


图 4.5.2.3 IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图

首先，作者说明一下上图带有颜色线条和标签（I、II、III、IV）的作用，红色线条表示输出方向；紫色线条代表 RTC IO 管脚的输出方向；黄色线条代表 RTC IO 管脚输入方向；标签代表输入/输出分支的节点。

从上图可知，ESP32-S3 管脚具有预设功能，即每个 IO 管脚直接连接至一组特定的片上外设。在运动时，可通过 IO MUX 和 IO 矩阵配置连接管脚外设。从上表 4.5.2.1 可知，有些 IO 管脚预设了 RTC 和模拟功能，有些 IO 管脚预设了 SPI、IIC 等功能。

上图右边两个“Pin X supplied by VDD3P3_CPU/RTC”框图为芯片焊盘(PAD)的内部结构，即芯片逻辑与 GPIO 管脚之间的电气接口。ESP32-S3 的 45 个 GPIO 管脚均采用此结构，如下图所示。

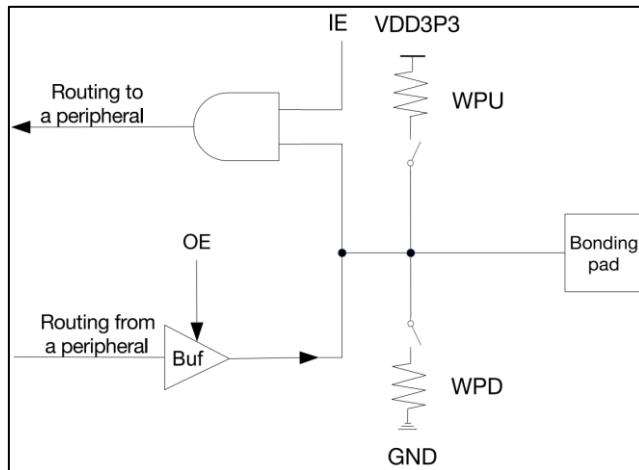


图 4.5.2.4 焊盘内部结构

上图中的 IE 表示输入使能；OE 表示输出使能；WPU 表示内部弱上拉；WPD 表示内部弱下拉，它们实现了芯片封装内晶片与 GPIO 管脚之间的物理连接。

一、“Pin X supplied by VDD3P3_CPU” 芯片焊盘输入流程（图 4.5.1.5.3 中的绿色线条）

从上图 4.5.2.3 可知，输入信号通过两条通道（IV 处）到达输入信号终端。第一条通道（①）无需经过 GPIO SYNC 模块的同步处理，而是通过 IO_MUX_n_REGIO 寄存器（该寄存器的 IO_MUX MCU_SEL 位作用为信号选择 IO MUX 功能，为 0 选择 Function 0，为 1 选择 Function 1（GPIO），Function 功能请看《esp32-s3_technical_reference_manual_cn.pdf》章节 6.12 IO MUX 管脚功能列表）配置进入 GPIO 交换矩阵，然后输入信号进入旁路 GPIO 交换矩阵（GPIO_SIMy_IN_SET）。另一方面，另一条通道经过 GPIO SYNC 模块的同步处理，然后将信号时钟同步 APB 总线时钟，随后进入 GPIO 交换矩阵。在这个交换矩阵中，通道的开通是由寄存器 GPIO_FUNCy_IN_SEL_CFG_REG 进行配置的。这个寄存器的描述如下。

Register 6.19. GPIO_FUNCy_IN_SEL_CFG_REG (y: 0-255) (0x0154+0x4*y)							
(reserved)							
31				8	7	6	5
0	0	0	0	0	0	0	0
Reset							
0x0							
GPIO_FUNCy_IN_SEL							
GPIO_FUNCy_IN_INV_SEL							
GPIO_SIGy_IN_SEL							

GPIO_FUNCy_IN_SEL 外设输入信号 Y 的选择控制位。此位选择 1 个 GPIO 交换矩阵输入管脚与信号连接；或者选择 0x38，则输入信号恒为高电平；或者选择 0x3C，则输入信号恒为低电平。（读/写）

GPIO_FUNCy_IN_INV_SEL 反转输入值。1：反转；0：不反转。（读/写）

GPIO_SIGy_IN_SEL 旁路 GPIO 交换矩阵。1：通过 GPIO 交换矩阵；0：直接通过 IO MUX 连接信号与外设。（读/写）

图 4.5.2.5 GPIO_FUNCy_IN_SEL_CFG_REG 描述

从上图可知，GPIO_FUNCy_IN_SEL（其中 y 为 GPIO 的管脚号）是外设输入信号控制位。如果 GPIO_FUNCy_IN_SEL 的值为 0x38，则输入信号被视为高电平；如果 GPIO_FUNCy_IN_SEL 的值为 0x3C，则输入信号被视为低电平。GPIO_FUNCy_IN_INV_SEL（其中 y 为 GPIO 的管脚号）是反转输入值的控制位。如果输入是高电平，经过反转操作后变为低电平；否则，保持高电平。GPIO_SIMy_IN_SET（其中 y 为 GPIO 的管脚号）是旁路 GPIO 交换矩阵，它的作用是提高高频数字信号的特性。如果 GPIO_SIMy_IN_SET 的值为 1，则选择 GPIO 交换矩阵作为输

入；否则，选择 IO MUX 作为输入，最终信号到达输入信号终端。

二、“Pin X supplied by VDD3P3_CPU” 芯片焊盘输出流程（图 4.5.1.5.3 中的红色线条）

从上图 4.5.2.3 可知，输出信号也是分为两个通道传输（I 处），如果输出信号是普通的 GPIO 输出，则该信号经过 GPIO 矩阵，再由该矩阵输出到 IO MUX，再到输出管脚，这个流程由 GPIO_FUNCy_OUT_SEL_CFG_REG 寄存器配置，如下所示：

Register 6.20. GPIO_FUNCx_OUT_SEL_CFG_REG ($x: 0-48$) (0x0554+0x4*x)											
(reserved)											
31							12	11	10	9	8
0	0	0	0	0	0	0	0	0	0	0	0
0x100 Reset											
GPIO_FUNCx_OUT_SEL GPIO_FUNCx_OUT_INV_SEL GPIO_FUNCx_OEN_SEL GPIO_FUNCx_OEN_INV_SEL											

GPIO_FUNCx_OUT_SEL GPIO 管脚 x 的输出信号选择控制位。值为 y ($0 \leq y < 256$) 连接外设输出 y 与 GPIO 输出 x 。值为 256 选择 GPIO_OUT_REG/GPIO_OUT1_REG[x] 和 GPIO_ENABLE_REG/GPIO_ENABLE1_REG [x] 作为输出值和输出使能。（读/写）

GPIO_FUNCx_OUT_INV_SEL 0：不反转输出值；1：反转输出值。（读/写）

GPIO_FUNCx_OEN_SEL 0：采用外设的输出使能信号；1：强制使用 GPIO_ENABLE_REG[x] 用作输出使能信号。（读/写）

GPIO_FUNCx_OEN_INV_SEL 0：不反转输出使能信号；1：反转输出使能信号。（读/写）

图 4.5.2.6 GPIO_FUNCy_OUT_SEL_CFG_REG 描述

从上图可知，GPIO_FUNCx_OUT_SEL（其中 x 为 GPIO 的管脚号）是外设输出信号控制位。当 GPIO0 管脚输出信号时，该值为 0。GPIO_FUNCx_OUT_INV_SEL（其中 x 为 GPIO 的管脚号）是反转输出值的控制位。如果输出是高电平，经过反转操作后变为低电平；否则，保持高电平。然后通过 IO_MUX_n_REGIO 寄存器（该寄存器的 IO_MUX MCU SEL 位的作用是信号选择 IO MUX 功能，为 0 选择 Function 0，为 1 选择 Function 1 (GPIO)。有关 Function 的详细信息，请参阅《esp32-s3_technical_reference_manual_cn.pdf》第 6.12 节中的 IO MUX 管脚功能列表）配置，信号最终到达输出管脚。

另一条通道是复用功能输出的通道。该通道由输出信号的起始端到 IO MUX 复用电路，然后 IO_MUX_n_REGIO 寄存器的 IO_MUX MCU SEL 位不为 1 (GPIO 模式)，为复用功能，最后经过 II 处输出到输出端 (GPIO 和 RTC IO)。

三、“Pin X supplied by VDD3P3_RTC” 芯片焊盘输入流程（图 4.5.1.5.3 中的黄色线条）

根据表 4.5.2.1 和图 4.5.2.3 所示，ESP32-S3 中有 22 个 GPIO 管脚具有低功耗 (RTC) 性能和模拟功能，由 RTC 子系统控制。这些功能不使用 IO MUX 和 GPIO 交换矩阵，而是使用 RTC IO MUX 将 22 个 RTC 输入输出信号引入 RTC 子系统。当这些管脚被配置为 RTC GPIO 管脚，作为输出管脚时仍然能够在芯片处于 Deep-sleep 模式下保持输出电平值或者作为输入管脚使用时可以将芯片从 Deep-sleep 中唤醒。

如果它们被用作普通输入，则输入流程与“Pin X supplied by VDD3P3_CPU”的输入流程相同。如果它们作为 RTC 复用功能，则输入信号会进入 RTC IO MUX 复用电路，并最终到达 RTC GPIO 矩阵。

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。

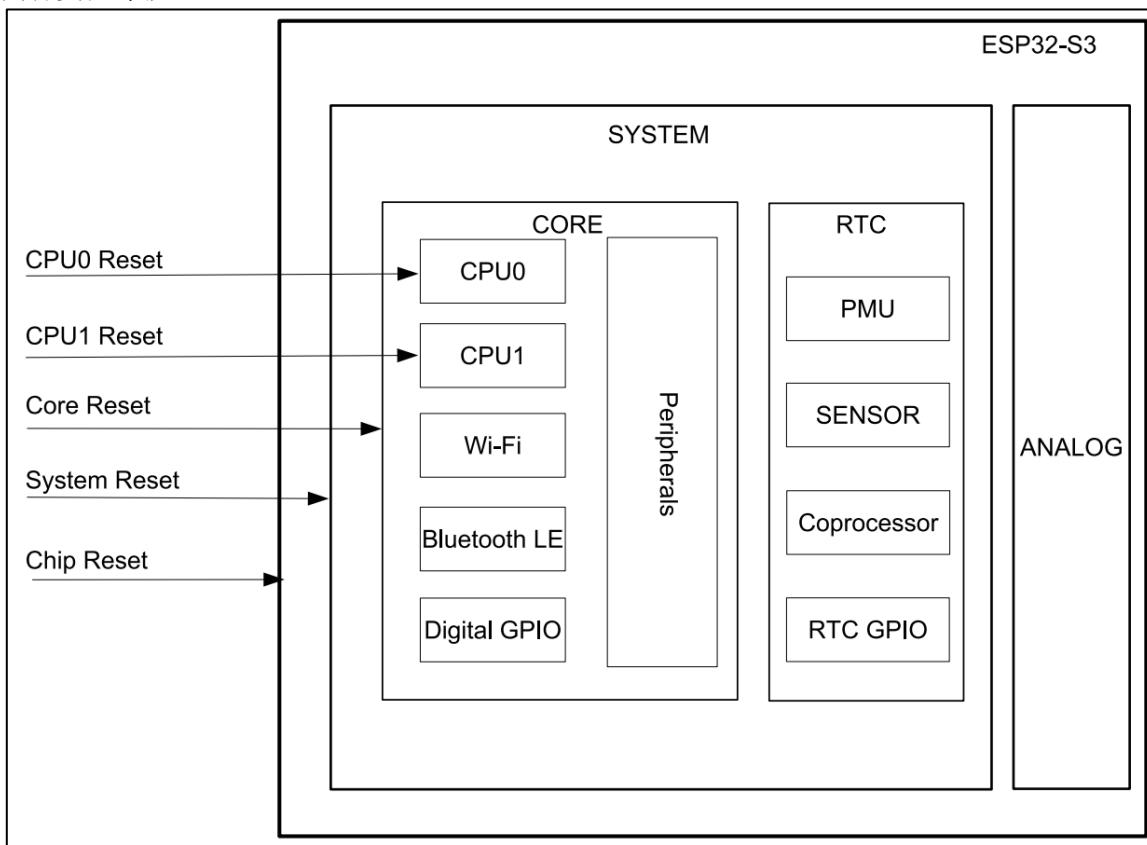


图 4.5.3.1.1 四种复位等级

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_APPC 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 CPUx 复位	CPU 复位	见技术手册章节 13
0x0C	软件 CPUx 复位	CPU 复位	配置 RTC_CNTL_SW_PROCPU_RST 寄存器触发
0x0D	RWDT CPUx 复位	CPU 复位	见技术手册章节 13
0x11	MWDT1 CPUx 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。

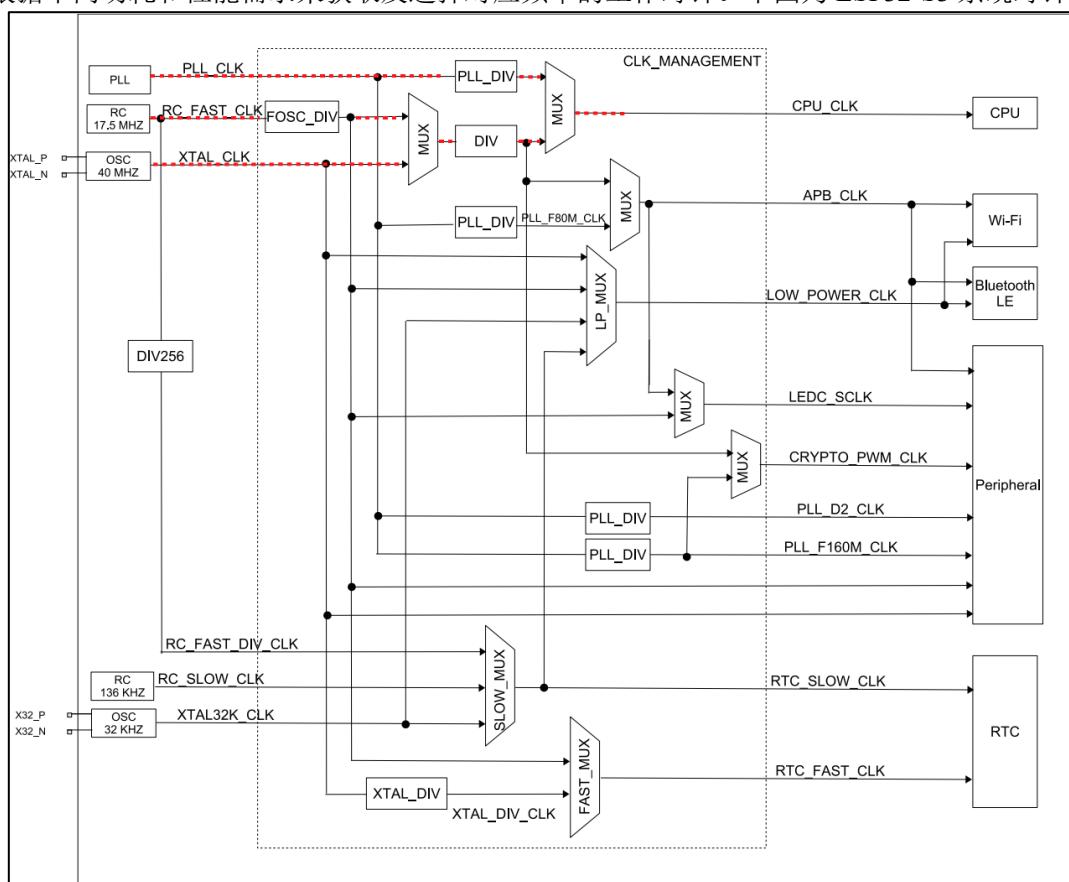


图 4.5.3.2.1 ESP32-S3 时钟树

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)

④: 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_CPUTERIOD_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、BLUE、RTC 等时钟配置及选择源，请读者参考《esp32-s3_technical_reference_manual_cn.pdf》技术手册章节 7 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_TIEH

² 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。

③ 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》->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 4-Kbit 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->章节 eFuse 控制器。

4.5.5 中断矩阵

ESP32-S3 的中断矩阵将任意外部中断源单独分配到双核 CPU 的任意外部中断上，以便在外部设备中断信号产生后，能够及时通知 CPU0 或 CPU1 进行处理。外部中断源必须经过中断矩阵分配至 CPU0/CPU1 外部中断，主要是因为 ESP32-S3 具有 99 个外部中断源，但每个 CPU 只有 32 个中断。通过使用中断矩阵，可以根据应用需求将一个外部中断源映射到多个 CPU0 中断或 CPU1 中断。实际上，CPU0 和 CPU1 的外部中断只有 26 个，剩下的 6 个中断均为内部中断。

下图是双核中断矩阵结构。

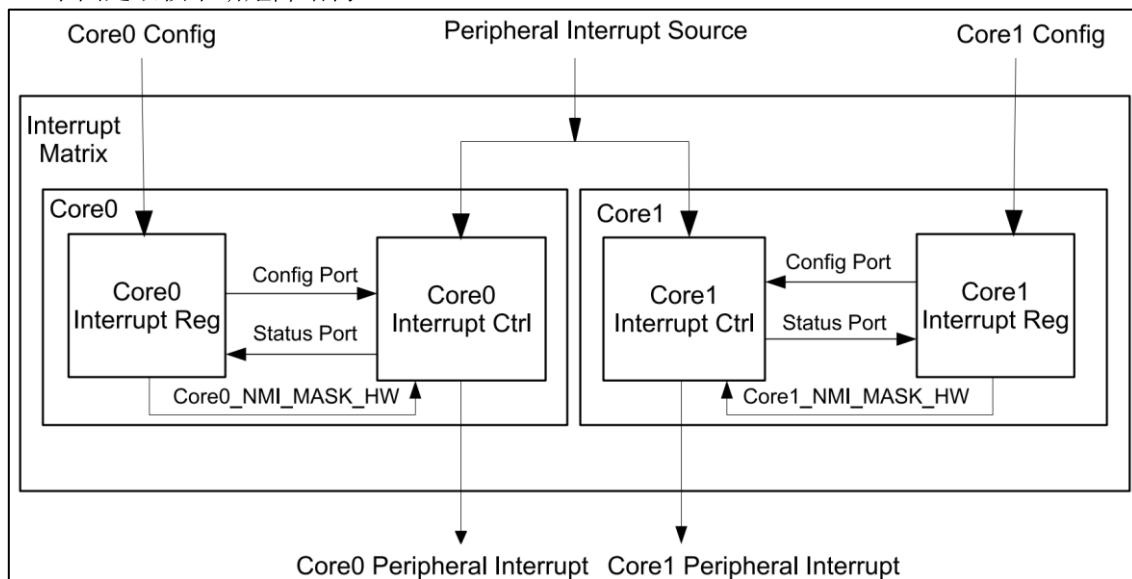


图 4.5.5.1 中断矩阵结构图

这种设计使得 ESP32S3 能够适应不同的应用需求，提供更大的灵活性和控制力。在硬件配置上，用户需要确保中断矩阵的正确配置，以便能够正确地接收和处理外部中断。同时，用户也需要通过编程方式，根据实际需求对中断矩阵进行适当的配置和操作。

当某个外部中断源满足触发条件时（例如 GPIO 引脚信号状态发生变化），该中断信号将被送入中断矩阵进行处理。中断矩阵将根据中断信号的特性，将其映射到一个特定的 CPU 外部中

断上。当 CPU 接收到这个外部中断信号时，会执行与该中断相关联的 ISR 函数。

总的来说，ESP32S3 的中断矩阵是一种高效的中断处理机制，它能够将多个外部中断源映射到两个 CPU 的外部中断上进行处理，并能够查询外部中断源当前的中断状态。

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。

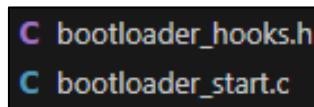


图 4.6.2 `bootloader_start.c` 文件路径

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。因此开发者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` 文件下定义的）。

同理，ESP32S3 的 MicroPython 固件也是以这种方式启动的。此外，MicroPython 的 `app_main` 函数是在 `ports/esp32/main.c` 文件中定义。在该文件中，会进行 UART、USB 和特定库的初始化操作。

第五章 MicroPython 环境搭建

本章，我们将进入实际操作阶段，逐步搭建 MicroPython 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

5.3 固件的下载与烧录

5.1 开发方式的选择

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



1, 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，本书籍选择此开发方式。

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。

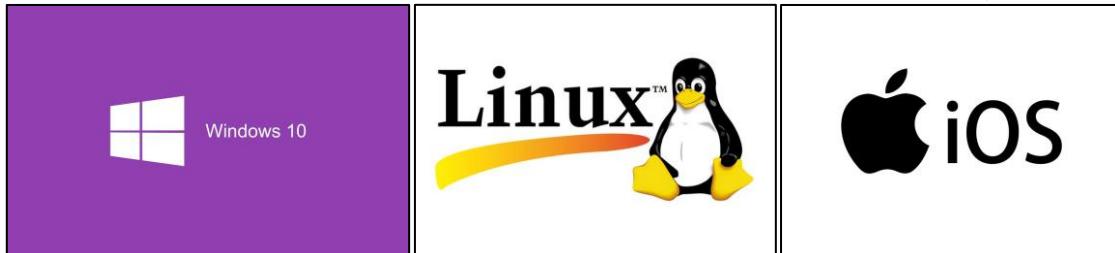
3, ESP-IDF: ESP-IDF 是乐鑫官方推出的开发框架，专门为 ESP32 和其他一些 ESP 系列芯片设计。它提供了一套完整的开发工具和库，可以帮助开发者快速地开发和调试 ESP32 应用程序。ESP-IDF 支持 C/C++ 语言，并提供了一套完整的 API，可以控制 ESP32 的各种功能和外设。此外，ESP-IDF 还提供了一个在线编译器和调试器，可以让开发者在云端进行开发和调试。

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，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，Arduino 和 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 开发系统的选择与环境搭建

5.2.1 开发系统的选择

选择 MicroPython 进行 ESP32-S3 开发，确实需要准备相应的开发环境和工具。根据开发系统不同，需选择适合的开发工具。例如，在 Windows 系统下开发，可以选择 Thonny 或 VS Code 等开发工具，这些工具都支持 MicroPython 插件，可以方便地编写和调试 MicroPython 代码。同样，在 Linux 和 MAC 系统下开发，也可以选择相应的开发工具进行 MicroPython 开发。



Linux 系统开发对于初学者来说也是不容易的，虽然目前的 Linux 系统像 Windows 系统一样都具备图形化配置，但对于经常使用电脑的人来说，Windows 系统还是他们日常的首选，因此本书都是基于 Windows 系统下开发的。

5.2.2 环境搭建

如果您是一位 Python 初学者小白，那最适合 Thonny 它了，如果不是初学者，请选择 PyDev 和 Pycharm 等工具。Thonny 是一款面向 Python 初学者的小型集成开发环境（IDE）。它的特点是轻量级，易于使用，且完全基于 Python 的内置图形库 tkinter 开发。这款 IDE 可以帮助初学者更快地上手 Python，避免在环境设置上浪费过多的时间。它能够让初学者更好地理解每一行代码的运行细节，并且解决了初学者可能会遇到的一些繁杂的环境问题。本书就是使用 Thonny 来开发的，所以下面作者会详细介绍这个软件的安装与软件框架。

一、下载&安装 Thonny

下载和安装 Thonny 的步骤如下：

1，下载：首先，您可以在 [Thonny 的官方网站](#) 上直接下载安装包，如下图所示。



图 5.2.2.1 Thonny 工具的下载主界面

如果您的电脑是 Windows 系统，则选择 Windows 版本；如果是苹果系统，则选择 macOS。然后再根据自己的电脑配置选择适合的 Python 版本，如下图所示。



图 5.2.2.2 选择 Thonny 和 Python 版本

这里作者选择“Installer with 64-bit Python 3.10”这个版本的 Thonny 工具（本书编写时最新

的 Thonny 版本，如有新版本，请下载最新版本），点击下载即可，最后在电脑下载目录下找到该安装文件。

2，安装：双击下好的 thonny-4.1.3.exe 文件（也可以在正点原子提供的软件资料找到安装文件），如下图所示。

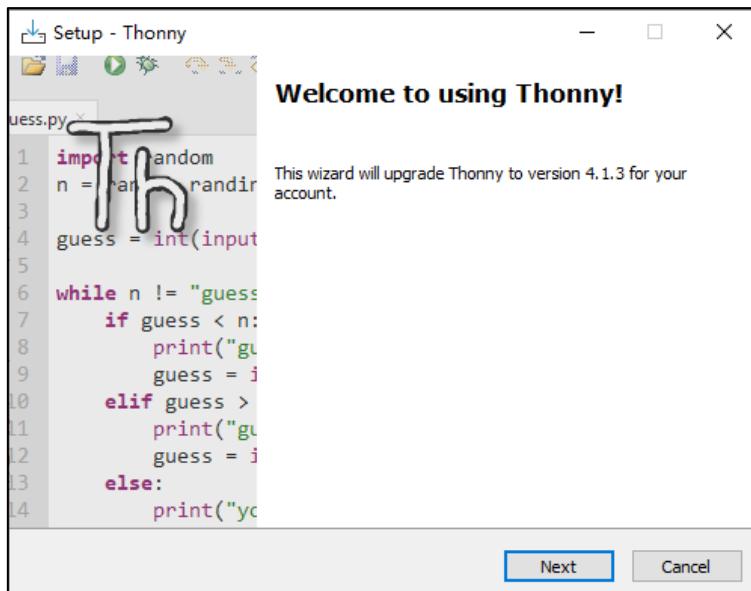


图 5.2.2.3 打开安装文件界面

点击上图中的“Next”按键，进入许可协议界面，如下图所示。

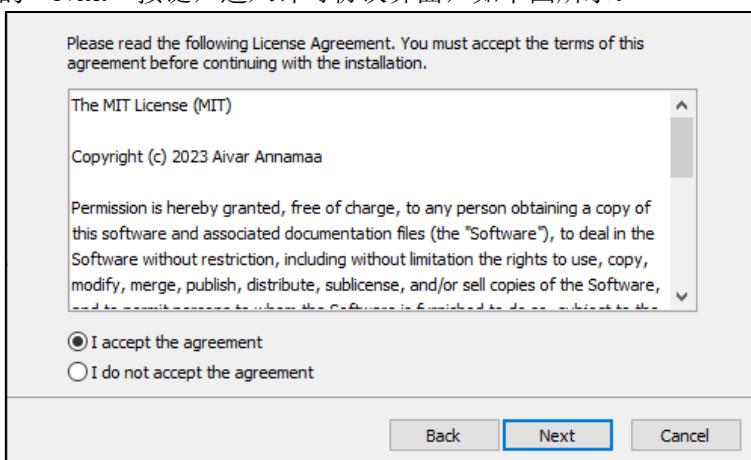


图 5.2.2.4 许可协议界面

选择“*I accept the agreement*”接受许可协议，接着点击“Next”按键，进入选择创建桌面图标界面，如下图所示。

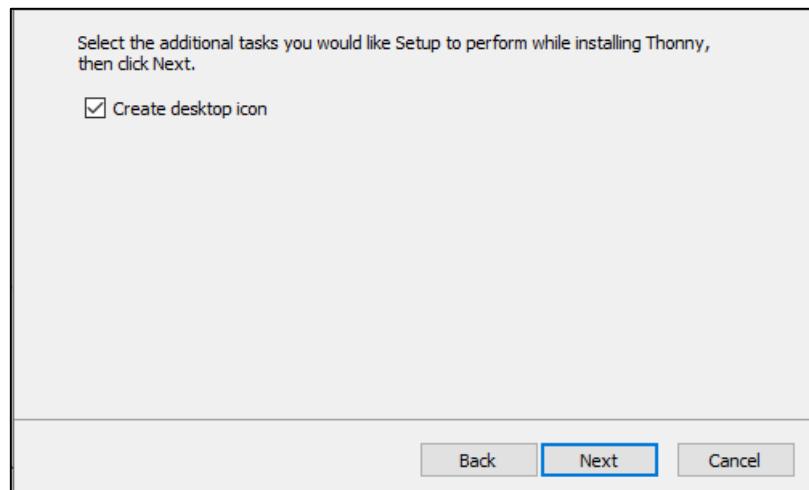


图 5.2.2.5 创建桌面图标界面

我们选择“create desktop icon”选项，接着继续点击“Next”按键，进入安装界面，如下图所示。

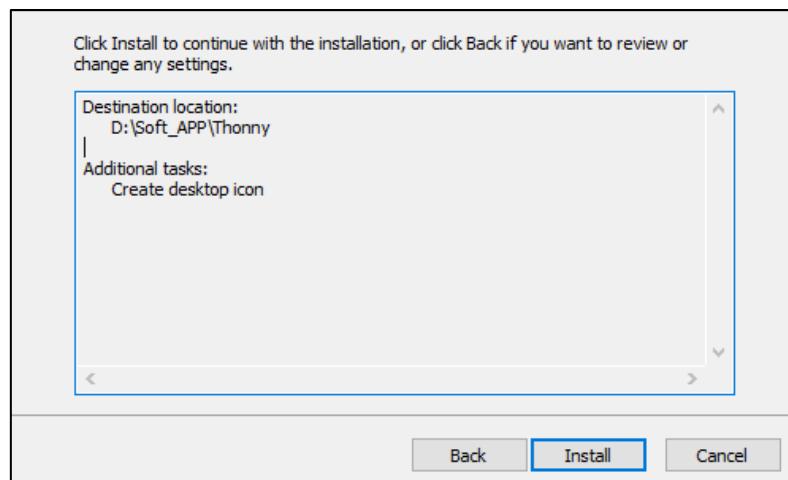


图 5.2.2.6 安装界面

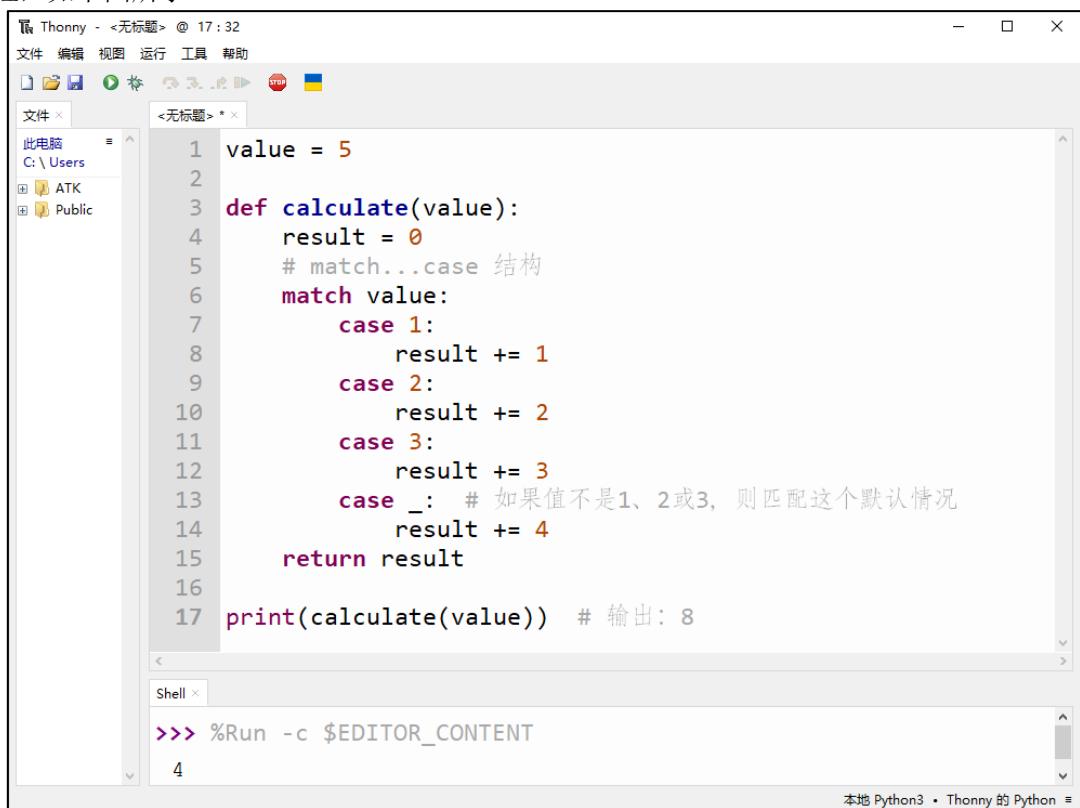
点击“Install”按键安装 Thonny 工具，安装成功后，便会在 Windows 桌面上自动生成 Thonny 程序图标。点击该图标便可以运行该软件。在第一次运行的时候，它会让您选择界面语言和初始化设置。这里就选择简体中文和标准（Standard）设置。最后进入程序标准运行界面。打开 Thonny 主界面如下所示。



图 5.2.2.7 Thonny 主界面

上图提示 Thonny 的 Python 版本是 3.10.11，我们可在 Shell 交互界面或者文件下测试 Python

例程，如下图所示。



```

1 value = 5
2
3 def calculate(value):
4     result = 0
5     # match...case 结构
6     match value:
7         case 1:
8             result += 1
9         case 2:
10            result += 2
11        case 3:
12            result += 3
13        case _: # 如果值不是1、2或3，则匹配这个默认情况
14            result += 4
15    return result
16
17 print(calculate(value)) # 输出: 8

```

本地 Python3 • Thonny 的 Python =

图 5.2.2.8 测试 Python 例程

可以看到，Shell 交互界面显示 Python 例程的输出结果，到了这里，Windows 系统下的 Thonny 工具安装完成。

二、Thonny 工具的框架解析

下面作者来讲解一下 Thonny 工具的界面框架，首先我们把 Thonny 工具主界面划分为几个板块，这些板块如下图所示。

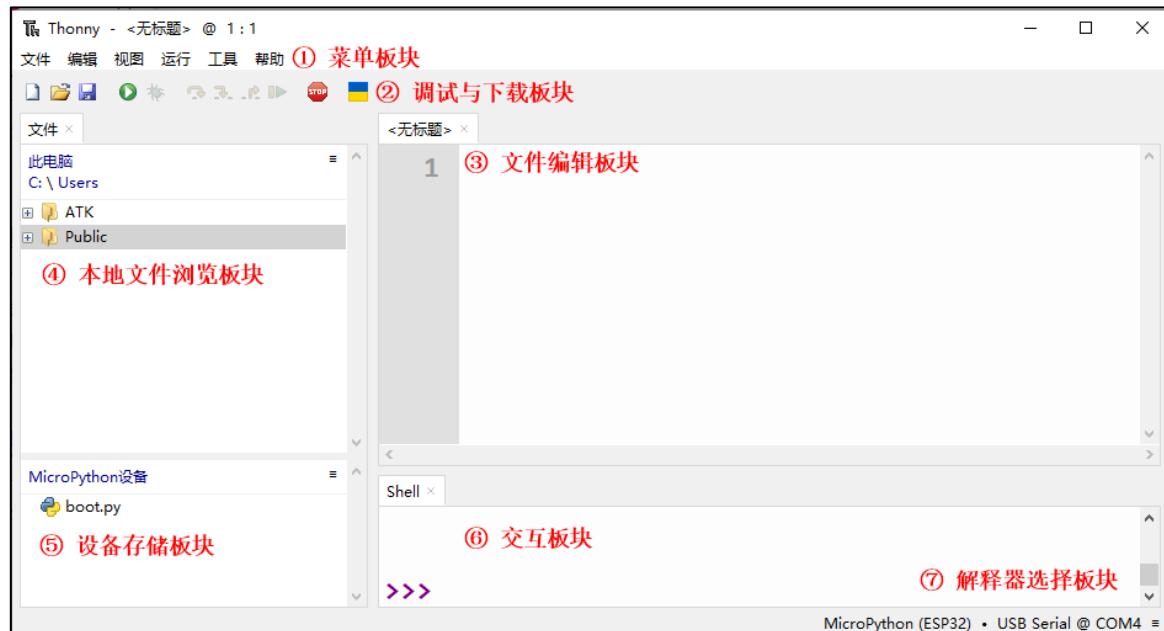


图 5.2.2.9 Thonny 工具的界面框架

从上图可知，作者把 Thonny 主界面划分为 7 个板块，这些板块介绍如下所示：

1，菜单板块

在这个板块下包含了文件、编辑、视图、运行、工具和帮助选项。

① 文件选项：

Thonny 工具的文件功能主要是指在程序运行过程中，对文件输入/输出的处理。具体功能包括：新建文件、打开新建、关闭文件等操作，文本界面如下图所示。

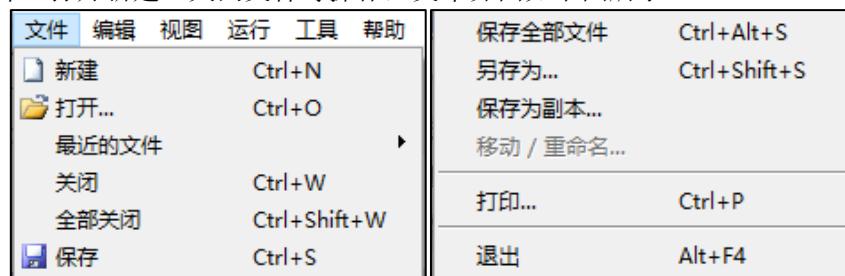


图 5.2.2.10 文件选项界面

② 编辑选项：

Thonny 工具提供了丰富的编辑功能，包括复制/剪切/粘贴、撤销/重做、查找/替换、自动补全、缩进与反缩进选择行、注释代码、折等操作，可以帮助用户更方便地编辑代码文本。编辑界面如下图所示。



图 5.2.2.11 编辑界面

③ 视图选项：

Thonny 工具的视图功能主要是指在程序运行过程中，通过勾选不同的选项来显示和关闭不同的视图窗口。有些视图窗口会显示程序运行过程中变量的值，这对于调试程序非常有帮助。特别是对于初学者来说，可以非常直观地看到变量的值。视图界面如下图所示。



图 5.2.2.12 视图界面

④ 运行选项：

Thonny 工具提供了运行 Python 脚本、单步执行、中断执行、停止/重启后端进程、运行脚本等功能，帮助用户控制和调试 Python 脚本的执行过程。运行界面如下图所示。



图 5.2.2.13 运行界面

⑤ 工具选项:

在“工具”选项中，用户可以管理插件和软件。例如，可以安装和卸载插件，配置解释器和编辑器属性等。工具界面如下图所示。



图 5.2.2.14 工具界面

对于管理插件，用户可以浏览可用的插件列表，选择需要安装的插件，并查看已安装的插件列表。如果需要卸载某个插件，只需选择该插件并点击“卸载”按钮即可。下面作者演示一下插件的安装与卸载，如下步骤所示：

打开“工具”选项，接着点击管理插件进入插件管理界面，如下图所示。



图 5.2.2.15 Thonny 插件管理界面

然后，我们在文本框输入要安装的插件，这里我们安装“thonny-black-format”插件，接着，点击“在 PyPI 上搜索”，搜索成功后的界面如下所示：



图 5.2.2.16 插件搜索结果

上图中，我们选择“thonny-black-format”插件，然后进入安装界面，如下图所示。

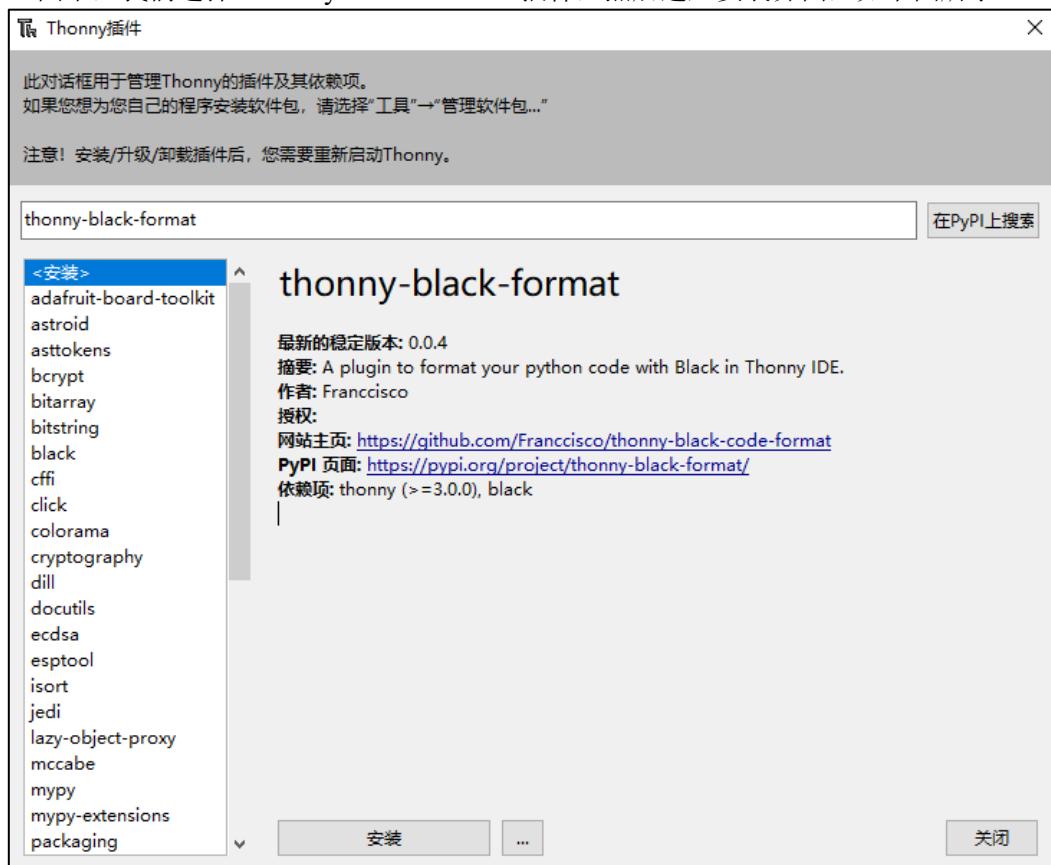


图 5.2.2.17 插件安装界面

到了这里，我们点击“安装”选项，即可安装这个插件，安装成功后需重启 Thonny 软件，在“工具”选项下会多出“thonny-black-format”插件，如下图所示。



图 5.2.2.18 安装“thonny-black-format”插件成功

卸载插件也是一样的操作，首先进入插件管理，然后选择要卸载的插件，最后点击“卸载”选项即可卸载，但卸载成功之后需重启 Thonny 软件。卸载插件步骤如下图所示。

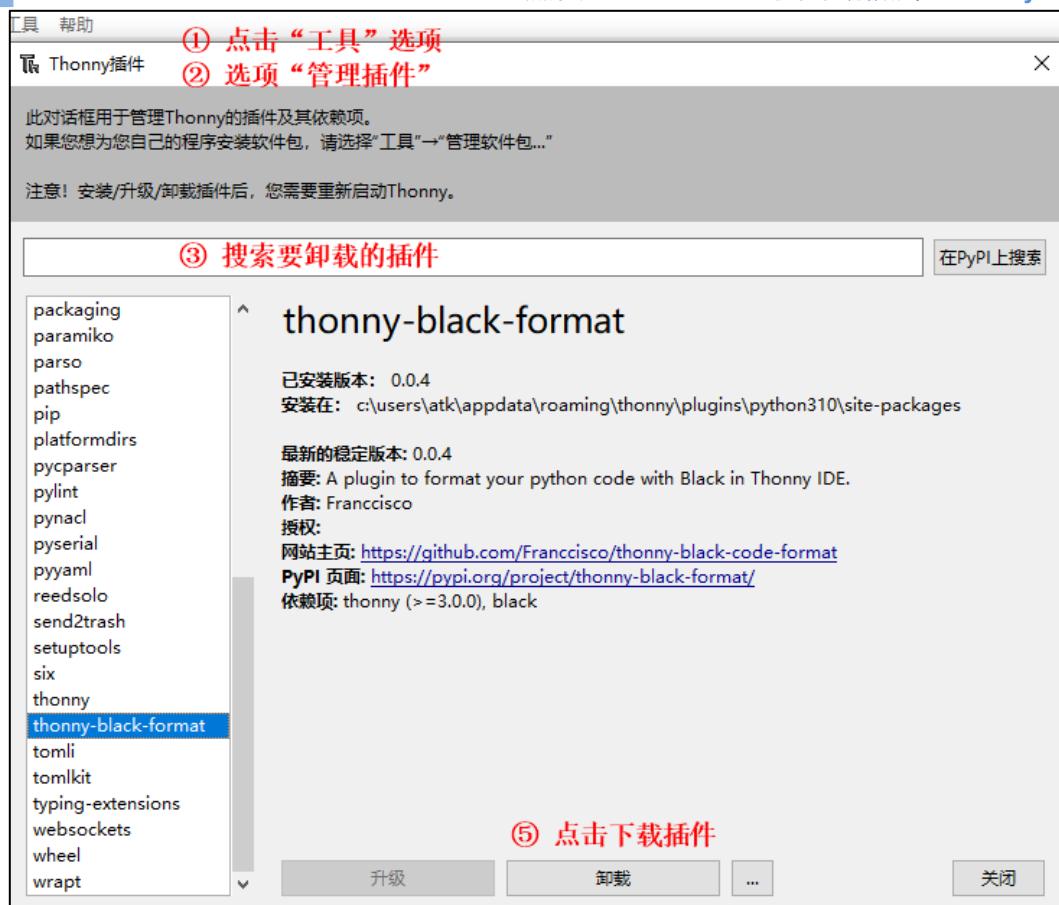


图 5.2.2.19 卸载插件的流程

对于解释器配置，用户可以在设置界面中选择和配置 Python 解释器。例如，可以选择本地安装的解释器，配置解释器的路径和其他参数。

对于编辑器属性配置，用户可以自定义 Thonny 工具的界面外观和行为。例如，可以更改工具栏的外观、设置代码编辑器的字体和颜色等。

⑥ 帮助选项

这个选项就是查看 Thonny 软件的版本，历史更新日记等信息，没什么好讲解的。

2. 调试与下载板块

这个板块的功能只不过在菜单板块下截取一些非常实用的功能，如下载调试、新建文件和另存文件等操作。

3. 文本编辑板块

这一个板块是 Python 代码的编辑器，用来编写代码用的。

4、5. 本地浏览板块和设备存储板块

本地浏览板块可以用来浏览电脑磁盘的内容，而设备存储板块则可以显示 ESP32 开发板的内部文件系统。这些功能使得我们可以在开发过程中，方便地查看和编辑文件。若用户在本地编写好的 py 脚本，可直接保存到 MicroPython 设备（ESP32-S3 内部文件系统）当中。

6、7. 交互板块和解释器选择板块

交互板块是指 Python 解释器与用户进行交互的界面，它可以用于输出程序结果、错误信息以及其他相关信息，如下图 5.2.2.20 所示。而解释器选择板块则是指选择哪个 Python 解释器来编译和运行 Python 脚本，如下图 5.2.2.21 所示。

```

43 if __name__ == '__main__':
44
45     esp.osdebug(1)          # 开启原厂 O/S 调试信息
46     esp.osdebug(0)          # 将原厂 O/S 调试信息重定向到 UART(0)
47     freq = machine.freq()   # 获取CPU当前工作频率
48     print(f'当前系统时钟{freq}') # 打印CPU工作频率
49     print(f'内部flash大小{esp.flash_size()/1024/1024}MB') # 打印FLASH大小
50     micropython.mem_info()  # 打印PSRAM信息
51
52     while True:
53
54         time.sleep_ms(10)    # 延时10ms
55

```

Shell < >>> %Run -c \$EDITOR_CONTENT

MPY: soft reboot
当前系统时钟2400000000
内部flash大小16.0MB
stack: 736 out of 15360
GC: total: 64000, used: 2992, free: 61008, max new split: 8257536
No. of 1-blocks: 33, 2-blocks: 8, max blk sz: 57, max free sz: 3723

图 5.2.2.20 Shell 交互式窗口



图 5.2.2.21 选择 py 解释器

在 ESP32 开发板上，通常会选择 MicroPython 解释器来编译和运行 Python 脚本。这是因为 MicroPython 是专门为微控制器和嵌入式系统设计的 Python 3 解释器，它具有轻量级、高效和易于使用等特点，非常适合在 ESP32 这样的低功耗设备上运行。

在本地计算机上运行 Python 程序时，则通常会选择本地 Python 3 解释器。这是因为 Python 是一种跨平台的语言，可以在不同的操作系统和设备上运行，而本地 Python 解释器则是针对特定平台和操作系统的。

5.2.3 USB 虚拟串口驱动安装

ESP32-S3 的 USB 串口可用于下载程序和 MicroPython 的 REPL 之间的交互。通过 USB 连接 ESP32-S3 最小系统板，在项目文件夹中执行特定指令，然后可以使用如 idf.py 等工具编译程序并下载到开发板中。此外，Thonny 工具也是通过 ESP32-S3 USB 串口把程序或者固件下载到开发板中的。

正点原子的 ESP32-S3 最小系统板的串口信号是通过 CH340C 芯片进行转换，才能与 PC 端进行通信。CH340C 芯片能够将 ESP32-S3 的串口信号转换为 USB 信号，并通过 USB 接口与 PC 进行连接。在 PC 端安装相关的串口调试助手软件，再安装 CH340C 芯片的驱动程序，就可以在 PC 端实现通过虚拟串口与 ESP32-S3 进行通信了。

接下来需要在电脑上安装 CH340C 芯片的驱动程序。CH340C 的官方厂商沁恒提供了该驱动程序的下载选项，您可以前往[沁恒的官方网站](#)下载并安装 CH340C 的驱动程序，也可在 6，软件资料→1，软件→CH340 驱动文件夹下找到 CH340C 的驱动安装程序，如下图所示。

6, 软件资料/1, 软件/CH340驱动/
`-- CH341SER.EXE

图 5.2.3.1 CH340C 驱动安装程序

打开 CH340C 驱动安装程序后，点击安装程序中的“安装”按钮，若提示“驱动安装成功”，则说明 CH340C 驱动已经安装成功了，如下图所示。

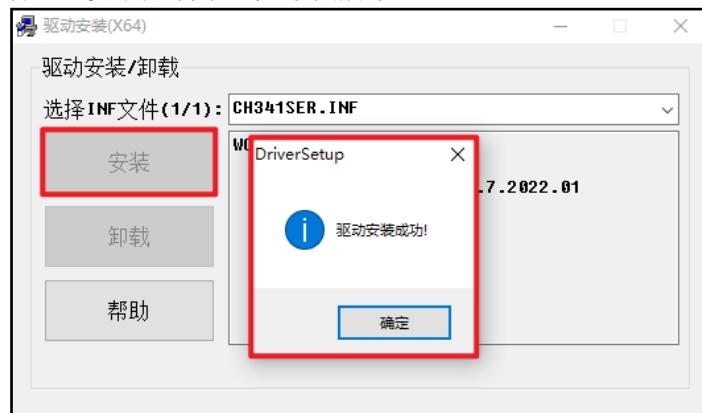


图 5.2.3.2 CH340C 驱动安装成功

安装完 CH340C 驱动后，使用跳线帽将正点原子 ESP32-S3 最小系统板 P4 排针的 U0TX-RXD 和 U0RX-TXD 接上，如下图所示。

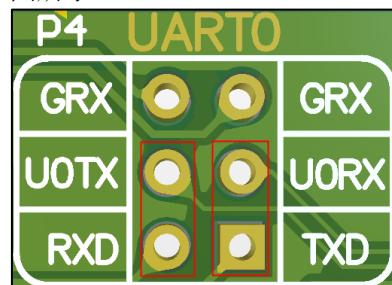


图 5.2.3.3 连接 USB-UART0

接下来，使用 USB 线将开发板 UART 接口与 PC 的 USB 端口相连接即可。此时，PC 端的设备管理器中查看到 CH340C 虚拟出的串口，如下图所示。

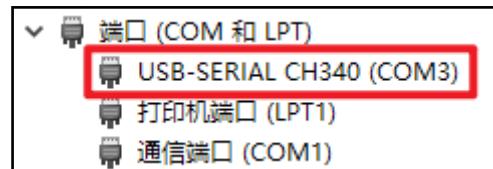


图 5.2.3.4 PC 端显示的虚拟串口

从上图可以看出，CH340C 虚拟出的串口被 PC 分配了 COM3 的端口号。这个端口号用于串口调试助手等上位机确定与之通信的串口端。需要注意的是，当 CH340C 与不同的 PC 连接，甚至是与同一台 PC 的不同 USB 端口连接后，虚拟出的串口被 PC 分配到的端口号可能是不同的，例如 COM4 或 COM5。读者可以根据设备管理器中端口设备的名称来判断具体是哪个端口号。如果同时连接了多个 CH340 系列的芯片，则需要逐个测试端口号。

安装完 USB 虚拟串口驱动后，就可以使用串口调试助手，如正点原子开发的 ATK-XCOM 软件，与板卡通过串口进行通信了。

5.2.4 ESP-32S3 开发 Thonny 的基本配置

使用 Thonny 工具进行 ESP32-S3 应用开发，需要完成以下配置：

打开 Thonny 工具，点击“运行”->“配置解释器”选项，如下图所示。



图 5.2.4.1 打开配置解释器

在配置解释器界面下配置解释器和串口端口，如下图所示。



图 5.2.4.2 配置解释器参数

配置完成后，就可以在 Thonny 工具中开发 ESP32-S3 芯片了。然而，在进行开发之前，需要用户将 MicroPython 固件下载到 ESP32-S3 芯片中，以便在 Thonny 工具中进行开发。固件的下载和烧录将在下一小节中讲解。

5.3 固件的下载与烧录

[MicroPython 官方](#)提供了多种固件，可直接下载后烧录至开发板运行。

官方的固件大致分为 pyboard、ESP 系列、STM32 系列等几类，下面分别说明：

1, pyboard 固件

从 v1.9.1 版本开始，pyboard 的固件分为下面几个不同版本：

- ① 标准版（standard）
- ② 双精度浮点版（double FP）
- ③ 线程版（threading）
- ④ 双进度浮点+线程版（double FP + threading）
- ⑤ 网络版（network）

这些版本具有相同的底层功能和包含的模块，但具有不同的专项优化。双精度浮点版本提升了浮点运算的精度，一般版本使用 32 位单精度浮点，而双精度浮点版本则使用 64 位。线程版本支持多线程功能，适合处理并行任务。双精度浮点+线程版本同时具备高精度浮点运算和多

线程功能。网络版本通过 SPI 接口连接外部网络模块（CC3000 等），为 pyboad 增加网络功能。

2, ESP 系列固件

ESP 系列包括 ES8266、ESP32、ESP32S3 等多款芯片，每款芯片都有其特定的固件。虽然 MicroPython 为这些芯片提供了相应的固件，但有时候这些固件可能不能满足开发需求。因为 MicroPython 的固件是针对特定型号的，所以一个芯片的固件资源无法与其他类型的芯片匹配。

在后续的教程中，作者将教大家如何编译适合资源匹配的固件。通过学习这些教程，读者可以了解到如何针对不同的 ESP 芯片编译 MicroPython 固件，从而满足开发需求。

3, STM32 系列固件

除了前面几种固件外，MicroPython 官方还提供了下面几种芯片的固件，它们分别为：stm32f0、stm32f4、stm32f7、stm32g0、stm32g4、stm32h5、stm32h7、stm32l0、stm32l1、stm32l4、stm32wb 和 stm32wl。

4, MicroPython 中文社区固件

虽然 MicroPython 官方提供了多种固件，但仍然有许多开发板没有官方支持的固件。对于许多使用者来说，编译源码是一项复杂和具有挑战性的任务。为了解决这个问题，MicroPython 中文社区特别提供了多种常见开发板的预编译固件，以便大家能够轻松地使用和测试这些开发板。

MicroPython 中文社区地址是：<https://git.oschina.net/shaoziyang/>。

以下，是在 MicroPython 官方下载对应芯片的固件并烧录至开发板上运行的步骤：

首先打开浏览器，然后打开‘百度一下’网页，接着，在文本框中输入‘MicroPython’，找到 MicroPython 官方网址，如下图所示。



图 5.3.1 搜索 MicroPython 官方网站

点击“MicroPython - Python for microcontrollers”网页链接，进入 MicroPython 官方网站，该网页如下所示。

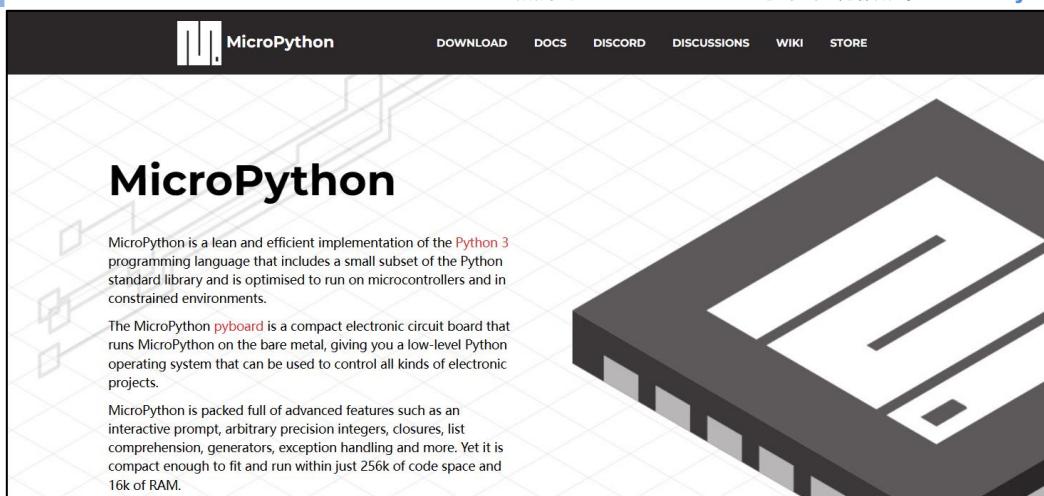


图 5.3.2 MicroPython 官方网页的主界面

在这个主界面下，点击“DOWNLOAD”下载选项，然后找到“ESP32”端口，操作流程如下所示。

MicroPython downloads

MicroPython is developed using git for source code management, and the master repository can be found on GitHub at github.com/micropython/micropython.

The full source-code distribution of the latest version is available for download here:

- [micropython-1.21.0.tar.xz](#) (78MiB)
- [micropython-1.21.0.zip](#) (165MiB)

Daily snapshots of the GitHub repository (not including submodules) are available from this server:

- [micropython-master.zip](#)
- [pyboard-master.zip](#)

Firmware for various microcontroller ports and boards are built automatically on a daily basis and can be found below.

Filter by:
Port: cc3200, esp32, esp8266, mimxrt, nrf, renesas-ra, rp2, samd, stm32

图 5.3.3 MicroPython 固件下载主界面

在上图中，点击“esp32”选项进入 ESP32 MicroPython 固件下载网页，接着往下找到“ESP32-S3 Espressif”选择，如下图所示。



图 5.3.4 ESP32S3 固件网页链接

此时，点击“ESP32-S3 Espressif”选项进入 ESP32-S3 MicroPython 固件下载界面，在此界

面下找到 Firmware (Support for Octal-SPIRAM) 标签，如下所示。

Firmware (Support for Octal-SPIRAM)

Releases

[v1.21.0 \(2023-10-05\) .uf2 / \[.app-bin\] / \[.bin\] / \[.elf\] / \[.map\] / \[Release notes\] \(latest\)](#)
[v1.20.0 \(2023-04-26\) .uf2 / \[.bin\] / \[.elf\] / \[.map\] / \[Release notes\]](#)

图 5.3.5 ESP32-S3 MicroPython 固件下载

这里我们必须根据自己的模组类型来选择固件，由于正点原子 ESP32-S3 最小系统板的模组为 ESP32-S3-WROOM-1-N16R8 型号，从《esp32-s3-wroom-1_wroom-1u_datasheet_cn.pdf》模组数据手册->章节 1.2 描述会发现，ESP32-S3-WROOM-1-N16R8 模组的 PSRAM 使用的是 Octal SPI，所以上图的固件，我们选择“v1.21.0(2023-10-05)/bin”固件，也就是上图红色框框的 bin 网页链接。

下载固件成功之后，我们得到了 ESP32_GENERIC_S3-20231005-v1.21.0.bin 固件文件，下面我们把这个固件烧录到 ESP32-S3 最小系统板中，这里会涉及到两种烧录方式。第一种是使用 Thonny 工具烧录，另一种是使用乐鑫官方提供的烧录固件软件 flash_download_tool。

下面作者重点讲解这两种烧录方式，希望读者掌握其中一种即可。

1. 使用 Thonny 工具烧录：

在 Thonny 工具中，选择“运行”菜单中的“配置解释器”选项，然后点击“安装或者更新 microPython (esptool)”如下图 5.3.6 所示，进入固件烧录界面，如下如 5.3.7 所示。



图 5.3.6 打开烧录界面

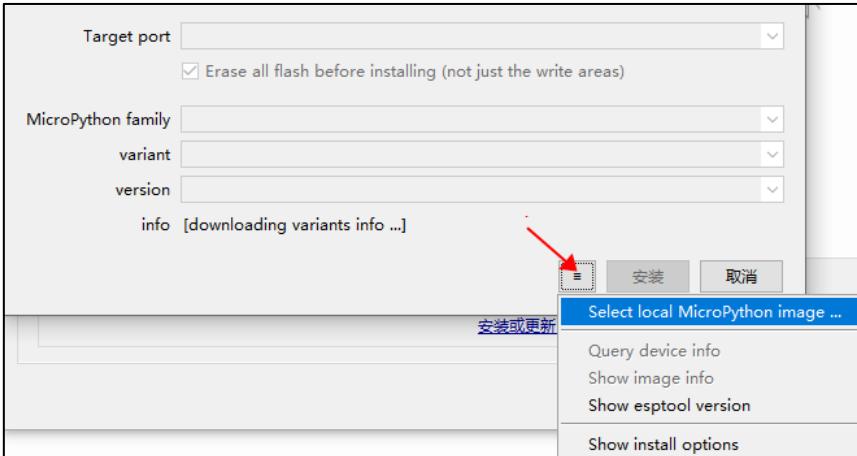


图 5.3.7 esptool 烧录固件工具

在上图中，我们点击红色箭头按键，在它弹出的下拉列表上选择“Select local microPython image...”选项，接着选择刚刚下载的 ESP32_GENERIC_S3-20231005-v1.21.0.bin 固件文件，此时，esptool 烧录界面变成可编辑界面，如下图所示。

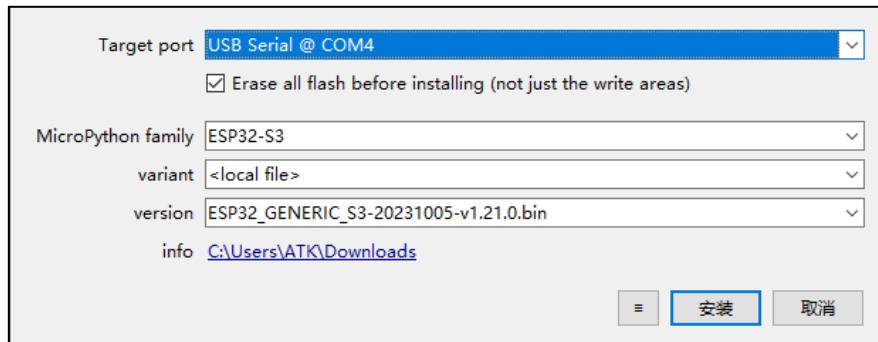


图 5.3.8 可编辑的 esptool 烧录界面

点击“安装”按钮就执行烧录操作了，此时点击“烧录”进度条即可看到烧录进度信息，如下图所示。

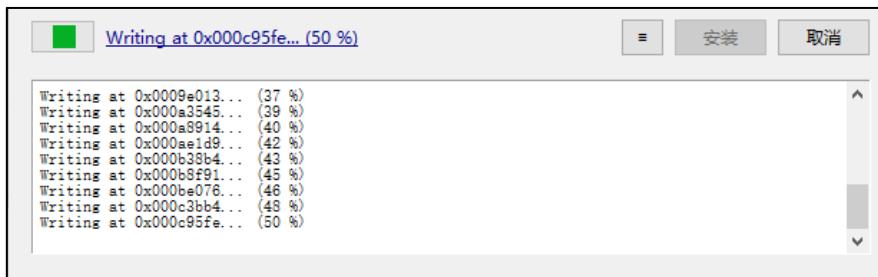


图 5.3.9 显示烧录进度

等待烧录进度提示“Hard resetting via RTS Pin ... Done”表示烧录完成，接着关闭 esptool 工具，重新选择解释器（MicroPython ESP32 * USB Serial @ COM4），并按下开发板上的复位按键，即可启动 MicroPython 固件了。最后，在 Shell 交互窗口上看到固件的信息，如下图所示。

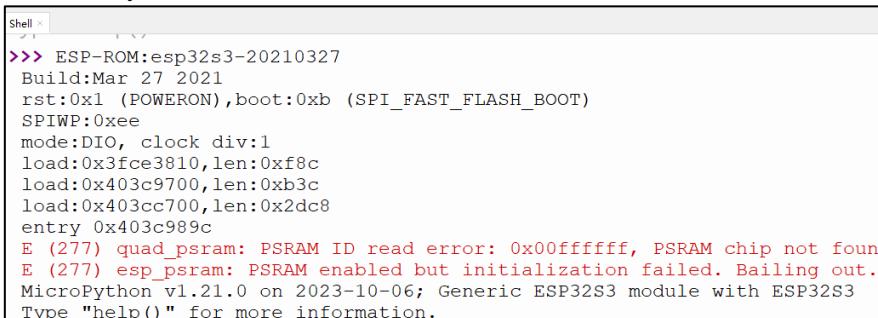


图 5.3.10 提示固件信息

从上图可以看到，Shell 交互式窗口提示“MicroPython v1.21.0 on 2023-10-06;Generic ESP32S3 module with ESP32S3”信息，表示已成功烧录 V1.21.0 版本 MicroPython。

注：上图提示两个错误，这是由于这个固件配置 PSRAM 参数，和本模组需配置 PSRAM 参数不匹配才导致的，所以在第五章的时候，作者会教大家如何制作自己的固件。

2, flash_download_tool 烧录：

首先打开[乐鑫下载工具](#)网页，接着下载 Flash 下载工具，如下图所示。

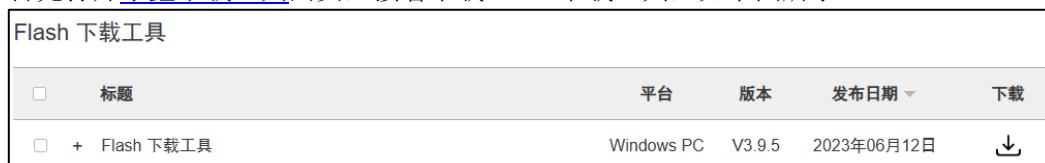


图 5.3.11 下载 flash_download_tool 工具

在上图中，点击下载 Flash 下载工具的链接，当前版本为 3.9.5，下载完成之后双击打开程序文件 flash_download_tool_3.9.5.exe，然后在“DOWNLOAD TOOL MODE”界面下选择芯片信息（Flash 下载工具相关介绍，请参考 flash_download_tool_3.9.5\doc 路径下的《Flash_Download_Tool_cn.pdf 使用手册》），如下图所示。

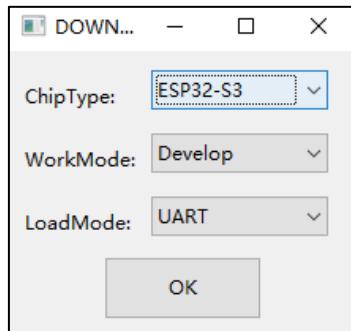


图 5.3.12 Flash 工具选择下载芯片

上图中，ChipType 选择 ESP32S3，WorkMode 选择 Develop LoadMode 选择 UART，点击 OK，进入下载页面，如下图所示。

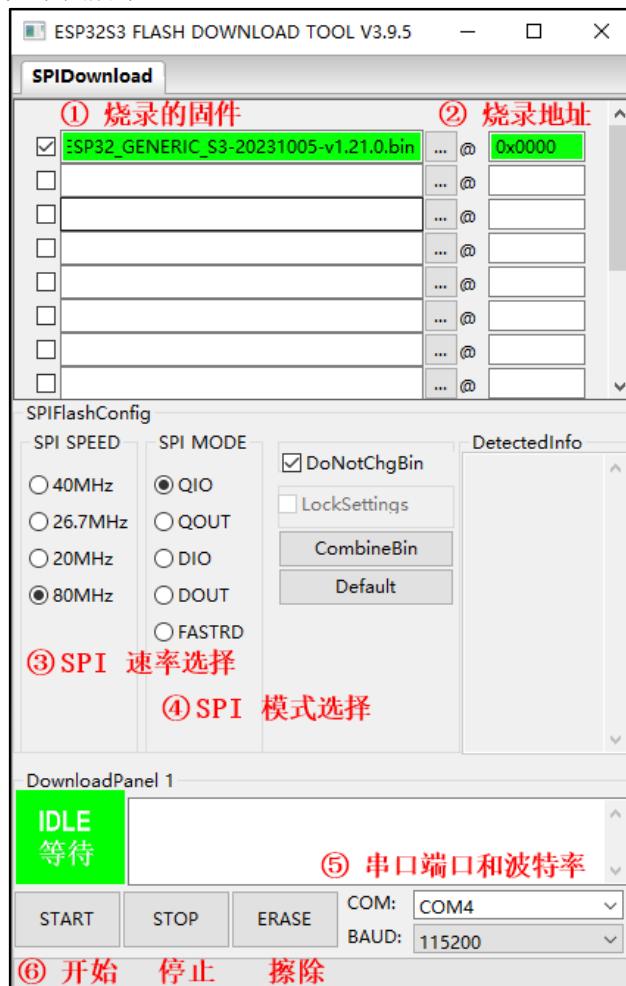


图 5.3.13 Flash 下载界面

上图①表示选择下载的固件；上图②表示该固件将被下载到 ESP32-S3 的 Flash 哪个地址，这里我们选择 0x0000；上图③和④表示 SPI Flash 速率及模式，这里我们选择 80MHz 和 QIO 模式；上图⑤表示串口端口和波特率，这里我们选择 COM4（不同电脑插入，端口号也不同）和 115200；上图⑥表示 Flash 固件下载开始、停止和擦除按键。

首先点击“ERASE”擦除按键，等待 ESP32-S3 擦除完成，然后点击“STOP”按键停止操作，接着点击“START”开始烧录操作，等待烧录完成后，就关闭 flash_download_tool 软件。

需要注意的是，在烧录固件之前，需要确认 ESP32-S3 最小系统板已经正确连接到电脑，并且端口号正确选择。同时，在烧录过程中要避免断电或断开连接等操作，以免对开发板造成损坏。

另外，正点原子提供了两个 MicroPython 出厂固件，它们唯一区别是是否携带 AI 接口，读者可在资料 A 盘\6，软件资料\1，软件\2，MicroPython 开发工具\01-Windows\4，相关固件路径下找到，如下图所示：

【正点原子】DNESP32-S3开发板(A盘) > 6. 软件资料 > 1. 软件 > 2. MicroPython开发工具 > 01-Windows > 4. 相关固件		
名称	类型	大小
ATK-ESP32_S3_8Mpsram(with_AI)-2024-01-02.bin	BIN 文件	5,241 KB
ATK-ESP32_S3_8Mpsram-2024-01-02.bin	BIN 文件	1,724 KB

图 5.3.14 ESP32-S3 固件

若读者想测试正点原子提供的 MicroPython 例程，可把这两个固件其中一个烧录到 ESP32-S3 最小系统板中，即可运行相关的例程。

第六章 MicroPython 固件编译

本章将向读者介绍 MicroPython ESP32 固件编译的详细步骤和过程。我们将从准备工作开始，逐步指导开发者完成固件编译的各个环节。通过阅读本章，开发者将了解 MicroPython 固件编译的基本原理、所需工具和文件，以及编译过程中的注意事项和技巧。

本章分为如下几个小节：

- 6.1 搭建编译环境
- 6.2 编译 ESP32-S3 固件
- 6.3 总结

6.1 搭建编译环境

自编译 MicroPython ESP32-S3 固件需要搭建两个开发环境，一个是 Linux 环境，另一个是 ESP-IDF 环境。这两个环境是必需的，因为 MicroPython ESP32 固件依赖于它们进行编译。下面是对这两个环境的搭建流程的介绍。

6.1.1 Linux 环境搭建

Linux 环境的搭建有三种方式。第一种是使用 VMware 虚拟机并安装 Ubuntu 系统来搭建，第二种是安装多个操作系统，每次重启电脑时根据需要进行选择进入，第三种是在 Windows 系统下安装 WSL Ubuntu 子系统。前两种方式属于比较完整的安装方式，都可以有图形用户界面，适合对图形界面有需求的同学。而最后一种 Linux 子系统的方式，安装和运行最快，但是只有命令行，适合 Linux 老手或者对图形界面需求不大的用户。

考虑到不需要安装 VMware 虚拟机且安装多个操作系统，本文介绍的是最后一种方式，如何在 Windows 上安装 ubuntu linux 子系统。本机是 Windows10 系统。

一、下载 ubuntu 子系统

进入 Windows store 微软商城，输入关键词 ubuntu，即可找到多个版本的 ubuntu 子系统，如下图所示。

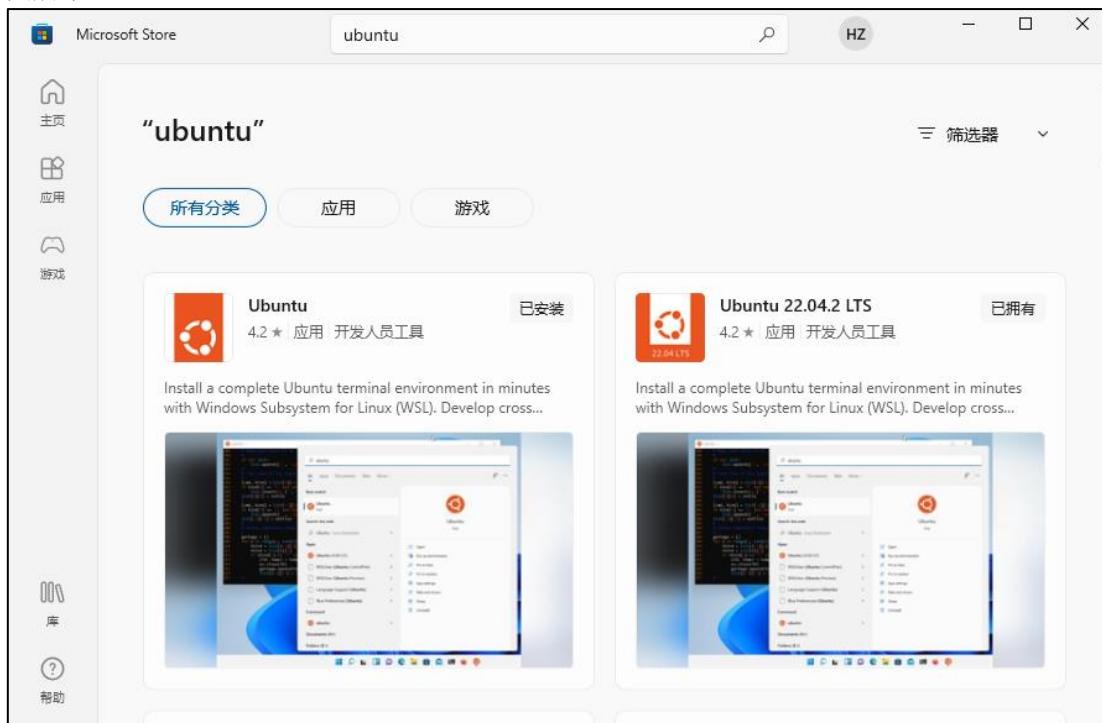


图 6.1.1.1 下载 Ubuntu

这里作者选的是 Ubuntu22.04.2 LTS 版本。点进去下载安装。接着我们打开 linux 子系统功能，如下步骤所示：

打开电脑设置（快捷键为 WIN + i），如下图所示。



图 6.1.1.2 Window 设置

在上图中，点击“应用”选项，进入应用和功能界面，如下图所示。



图 6.1.1.3 应用和功能界面

在上图中，点击可选功能，进入可选功能界面，接着往下找到更多 Windows 功能选项，并且点击进入，如下图所示。

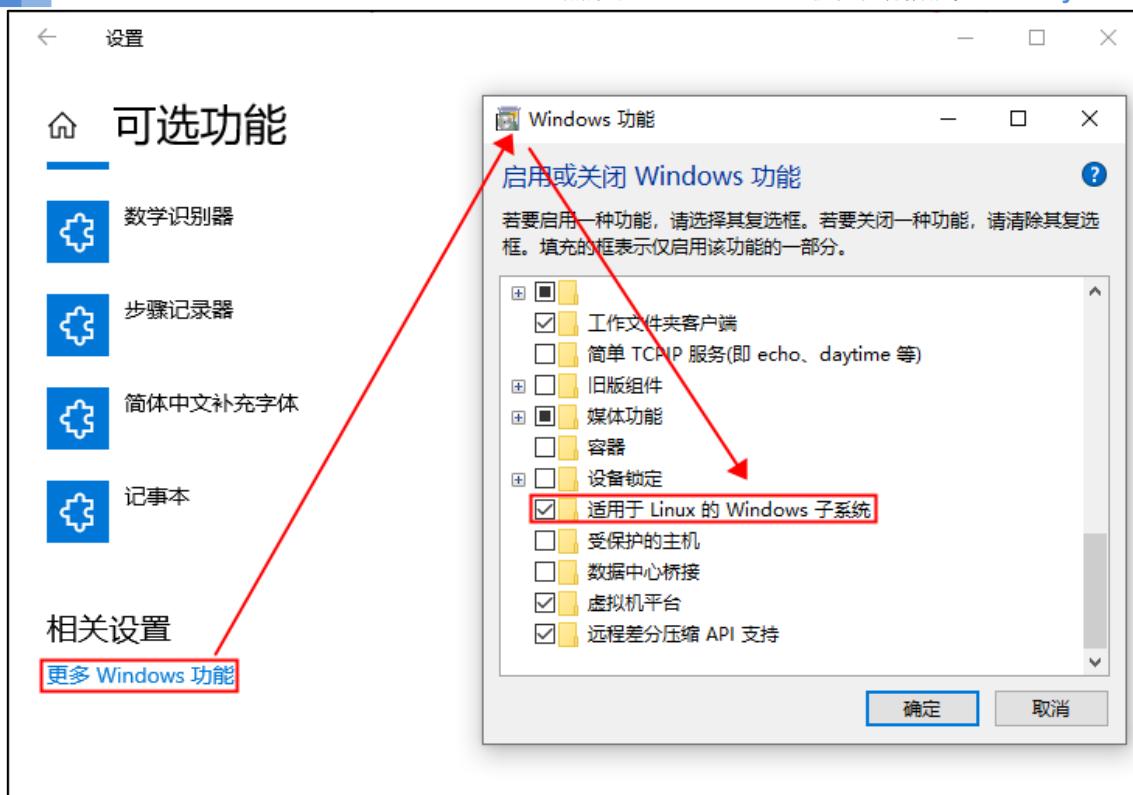


图 6.1.1.4 打开 Windows 功能

在此界面下找到“适用 Linux 的 window 子系统”选项，我们必须勾选这个选项，否则无法启动子系统，最后重启系统即可。

重启完毕后，需要进行一下账户配置。我们还是从微软商城中进入，找到我们刚刚安装的 Ubuntu22.04.2 LTS，点击“打开”。此时会提示正在安装，安装完毕后，会让您输入一个用户名，并配置密码，按照提示进行配置，最后进入 Ubuntu22.04.2 LTS 系统，如下图所示。

```
caixuefeng@DESKTOP-QH7611H: ~
Please create a default UNIX user account. The username does not need to match
your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: caixuefeng
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 4.4.0-19041-Microsoft x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

This message is shown once a day. To disable it please create the
/home/caixuefeng/.hushlogin file.
caixuefeng@DESKTOP-QH7611H: ~ $
```

6.1.1.5 配置用户名和密码

二、WSL 子系统搬迁

Windows Store 微软商城安装的 APP 都默认被安装在 C 盘，然而 C 盘是我们电脑重要的磁盘，如果 C 盘空间满了，会影响电脑运行。因此，我们最好将子系统搬离 C 盘，也就是说，不使用 C 盘来存放文件。在这里，作者将子系统搬到 D 盘存储。

WSL 子系统搬迁流程，如下所示：

1，查询 WSL 系统状态。使用管理员身份打开 CMD 命令行，并在命令行输入“wsl -l -v”查看 wsl 虚拟机的名称和状态，如下图所示。

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.3570]
(c) Microsoft Corporation. 保留所有权利。

C:\Windows\system32>
C:\Windows\system32>wsl -l -v
  NAME          STATE           VERSION
* Ubuntu        Stopped        1
  Ubuntu-22.04  Running       1

C:\Windows\system32>
```

图 6.1.1.6 wsl 虚拟机的名称与状态

从上图可知，作者安装了两个 WSL Ubuntu 子系统，它们分别为 Ubuntu 和 Ubuntu-22.04，目前它们的状态分别为停止状态和运行状态。下面，作者将以 Ubuntu-22.04 为例，说明如何将其搬迁到 D 盘。

2，停止 WSL 子系统。输入 wsl --shutdown 使 Ubuntu-22.04 停止运行，再次使用 wsl -l -v 确保其处于 stopped 状态，如下图所示。

```
C:\Windows\System32\cmd.exe
C:\Windows\system32>wsl --shutdown

C:\Windows\system32>wsl -l -v
  NAME          STATE           VERSION
* Ubuntu        Stopped        1
  Ubuntu-22.04  Stopped       1

C:\Windows\system32>
```

图 6.1.1.7 Ubuntu-22.04 停止运行

3，导出/恢复备份。在 D 盘创建一个文件夹用来存放新的 WSL，比如作者创建了一个 D:\Ubuntu_WSL 文件夹。

①：导出它的备份（比如命名为 Ubuntu.tar）。在 CMD 命令行下输入“wsl --export Ubuntu-22.04 D:\Ubuntu_WSL\Ubuntu.tar”。

②：确定在 D:\Ubuntu_WSL 下是否可看见备份 Ubuntu.tar 文件，如下图所示那样，之后注销原有的 WSL。在 CMD 命令行输入“wsl --unregister Ubuntu-22.04”注销原本的 WSL。



图 6.1.1.8 在 D:\Ubuntu_WSL 目录下生成 Ubuntu.tar 文件

③：将备份文件恢复到 D:\Ubuntu_WSL 中去。在 CMD 命令行下输入“wsl --import Ubuntu-22.04 D:\Ubuntu_WSL D:\Ubuntu_WSL\Ubuntu.tar”恢复备份文件，如下图所示。



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.3570]
(c) Microsoft Corporation. 保留所有权利。

C:\Windows\system32>wsl -l -v
  NAME      STATE      VERSION
* Ubuntu    Stopped    1
  Ubuntu-22.04 Stopped    1

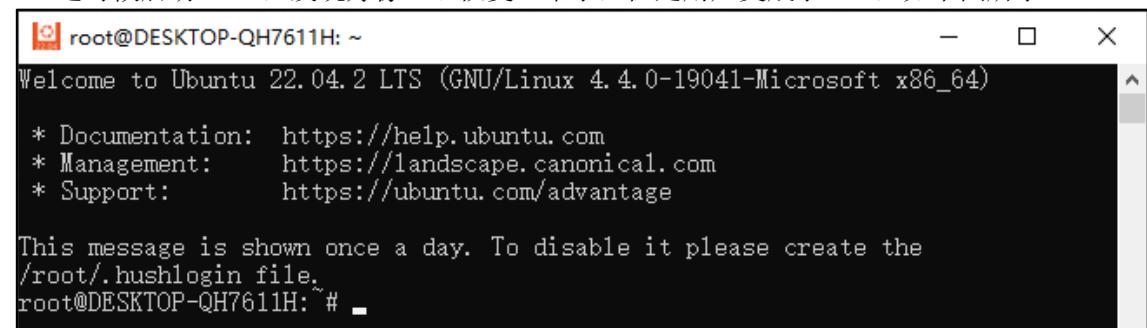
C:\Windows\system32>wsl --export Ubuntu-22.04 D:\Ubuntu_WSL\Ubuntu.tar

C:\Windows\system32>wsl --unregister Ubuntu-22.04
正在注销...

C:\Windows\system32>
```

图 6.1.1.9 恢复 WSL 备份

这时候启动 WSL，发现好像已经恢复正常了，但是用户变成了 root，如下图所示。



```
root@DESKTOP-QH7611H: ~
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 4.4.0-19041-Microsoft x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This message is shown once a day. To disable it please create the
/root/.hushlogin file.
root@DESKTOP-QH7611H: #
```

图 6.1.1.10 启动 Ubuntu-22.04

在使用 git 工具进行代码版本管理时，特别是在克隆开源市场中的 Github 仓库时，可能会遇到超时或失败的问题。这主要是因为国内网络连接国外服务器可能会显得冗余。为了解决这个问题，作者介绍了一种加速方法，可以帮助大家更好地搭建 MicroPython 环境。这种方法的具体操作步骤如下：

①：在 Ubuntu 子系统上输入以下命令来解析 github.com 的 IP 地址。

```
nslookup github.com
解析完成之后，我们得到了 Github 仓库的 IP 地址，如下图所示。
```

```
Non-authoritative answer:
Name: github.com
Address: 20.205.243.166
```

图 6.1.1.11 解析 github 仓库

②：修改 hosts 文件，该文件位于路径是：C:\Windows\System32\drivers\etc。添加内容如下。

```
# GitHub520 Host Start
140.82.112.4           alive.github.com
140.82.112.4           live.github.com
18.207.134.67          github.githubassets.com
140.82.112.4           central.github.com
3.220.169.176          desktop.githubusercontent.com
140.82.112.4           assets-cdn.github.com
3.238.132.1            camo.githubusercontent.com
151.101.1.6             github.map.fastly.net
151.101.1.6             github.global.ssl.fastly.net
140.82.112.4           gist.github.com
185.199.108.153         github.io
20.205.243.166          github.com
192.0.66.2              github.blog
140.82.112.4           api.github.com
18.212.214.173          raw.githubusercontent.com
```

```

44.204.227.149          user-images.githubusercontent.com
18.208.143.190          favicons.githubusercontent.com
3.227.11.2               avatars5.githubusercontent.com
44.192.25.139          avatars4.githubusercontent.com
54.197.35.64            avatars3.githubusercontent.com
44.211.83.233          avatars2.githubusercontent.com
3.238.132.1             avatars1.githubusercontent.com
3.238.199.124          avatars0.githubusercontent.com
44.210.19.120          avatars.githubusercontent.com
140.82.112.4            codeload.github.com
72.21.206.80            github-cloud.s3.amazonaws.com
72.21.206.80            github-com.s3.amazonaws.com
72.21.206.80            github-production-release-asset-
2e65be.s3.amazonaws.com  github-production-user-asset-
6210df.s3.amazonaws.com  github-production-repository-file-
72.21.206.80            githubstatus.com
5c1aeb.s3.amazonaws.com  github.community
185.199.108.153         github.dev
140.82.113.18           collector.github.com
52.224.38.193           pipelines.actions.githubusercontent.com
140.82.112.4            media.githubusercontent.com
3.238.235.79            cloud.githubusercontent.com
18.207.249.155          objects.githubusercontent.com
35.172.164.199          vscode.dev
54.198.215.57
13.107.219.40

```

大部分情况下是直接生效，如未生效可尝试刷新 DNS，在 CMD 窗口输入“ipconfig /flushdns”命令。

6.1.2 安装工具链

到了这里，我们将开始在 Ubuntu 下工作。首先，我们需要安装 Linux 常用的工具链，例如 gcc、cmake、python3、source、git 等。为了安装这些工具链，我们首先需要打开 Ubuntu-22.04 子系统。然后，以命令的形式安装工具链，如下安装命令：

- 1, 更新工具: sudo apt-get update
- 2, 安装 GCC: sudo apt-get install gcc
- 3, 安装 Cmake: sudo apt-get install cmake
- 4, 安装 python3.10: sudo apt-get install python3.10
- 5, 安装 python pip: sudo apt-get install python3-pip
- 6, 安装 virtualenv: sudo apt-get install virtualenv
- 7, 安装 git: sudo apt-get install git

一旦这些工具链成功安装，我们就可以开始搭建 ESP-IDF 环境并编译 MicroPython 固件。请注意，这些工具链的安装必须全部安装完成，否则在编译 MicroPython 时可能会出现问题。

6.1.3 ESP-IDF 环境搭建

ESP-IDF（Espressif IoT Development Framework）是用于开发 ESP 系列芯片的官方开发环境，它支持 Windows、Linux 和 macOS 操作系统，方便用户在不同系统下开发。在开发之前，我们先了解一下 ESP-IDF 和乐鑫芯片的版本关系，不同版本的 IDF 所支持的乐鑫芯片都不一样，下图总结了乐鑫芯片在 ESP-IDF 各版本中的支持状态，其中支持代表已支持，预览代表目前处于预览支持状态。预览支持状态通常有时间限制，而且仅适用于测试版芯片。请确保使用与芯片相匹配的 ESP-IDF 版本。

芯片	v4.3	v4.4	v5.0	v5.1	v5.2	
ESP32	支持	支持	支持	支持	支持	
ESP32-S2	支持	支持	支持	支持	支持	
ESP32-C3	支持	支持	支持	支持	支持	
ESP32-S3		支持	支持	支持	支持	芯片发布公告
ESP32-C2			支持	支持	支持	芯片发布公告
ESP32-C6				支持	支持	芯片发布公告
ESP32-H2				支持	支持	芯片发布公告
ESP32-P4					预览	芯片发布公告

图 6.1.3.1 不同版本的 IDF 支持乐鑫芯片关系表

从上图可以看到，ESP32-S3 芯片只能在 IDF v4.4 版本及以上运行，因此我们在使用 MicroPython 搭建 IDF 开发环境时，需要选择 IDF v4.4 版本及以上。在这里，作者选择的是 V5.0.4 版本，因为 MicroPython 目前仅支持 IDF v5.0.4 和 V5.1.2 版本（[MicroPython IDF 版本要求](#)：MicroPython 目前支持的 IDF 版本如下图所示）。然而，不排除以后可能会支持更高版本的可能性。

Setting up ESP-IDF and the build environment

MicroPython on ESP32 requires the Espressif IDF version 5 (IoT development framework, aka SDK). The ESP-IDF includes the libraries and RTOS needed to manage the ESP32 microcontroller, as well as a way to manage the required build environment and toolchains needed to build the firmware.

The ESP-IDF changes quickly and MicroPython only supports certain versions. Currently MicroPython supports v5.0.4, v5.1.2.

图 6.1.3.2 MicroPython 支持的 IDF 版本

到了这里，我们已经了解了 MicroPython 目前支持的 IDF 版本，下面作者来讲解 Micropython ESP-IDF 环境搭建的基本步骤：

1，克隆 IDF 库

首先打开 ubuntu 22.04.2 LTS 子系统，然后在命令行下输出“git clone -b v5.0.4 --recursive https://github.com/espressif/esp-idf.git（目前官方建议使用 5.0.4 版本）”克隆 Github 仓库中的 ESP-IDF 源码库，如下图所示。

```
root@DESKTOP-QH7611H:~/micropython/ports/esp32
root@DESKTOP-QH7611H:~# git clone -b v5.0.4 --recursive https://github.com/espressif/esp-idf.git
Cloning into 'esp-idf'...
remote: Enumerating objects: 551660, done.
remote: Counting objects: 100% (3026/3026), done.
remote: Compressing objects: 100% (1464/1464), done.
remote: Total 551660 (delta 1605), reused 2828 (delta 1449), pack-reused 548634
Receiving objects: 100% (551660/551660), 255.27 MiB | 8.13 MiB/s, done.
Resolving deltas: 100% (403362/403362), done.
Note: switching to '8fbf4ba6058bcf736317d8a7aa75d0578563c38b'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>
```

图 6.1.3.3 克隆 ESP32 IDF 库

当读者成功克隆 IDF 库后，我们会在子系统（D:\Ubuntu_WSL\rootfs\root）根目录下找到一个名为“esp-idf”的文件夹。这个文件夹包含了 ESP-IDF 开发环境的相关文件和工具（相关文件的描述，请用户观看 IDF 版的教程）。

2, 切换 IDF 5.0.4 版本

在 root@DESKTOP-QH7611H:~/根目录下输入“cd esp-idf”进入 esp-idf 文件夹，然后在 root@DESKTOP-QH7611H:~/esp-idf# 目录下输入“git checkout v5.0.4”命令，切换 IDF 版本为 v5.0.4。如下图所示。

```
root@DESKTOP-QH7611H: ~/micropython/ports/esp32
root@DESKTOP-QH7611H: ~/esp-idf# git checkout v5.0.4
HEAD is now at 8fbf4ba605 bugfix(usb/host): Fix transfer direction determination during argument check
ing for regular EP transfer
```

图 6.1.3.4 切换 IDF 版本

3, 更新 IDF 版本子模块

在 root@DESKTOP-QH7611H:~/esp-idf# 目录下输入“git submodule update --init --recursive”更新子模块，如下图所示。

```
root@DESKTOP-QH7611H: ~/esp-idf
Cloning into '/root/esp-idf/components/bt/controller/lib_esp32c2/esp32c2_bt-lib'...
Cloning into '/root/esp-idf/components/bt/controller/lib_esp32c3_family'...
Cloning into '/root/esp-idf/components/bt/controller/lib_esp32h2/esp32h2_bt-lib'...
Cloning into '/root/esp-idf/components/bt/host/nimble/nimble'...
Cloning into '/root/esp-idf/components/cmock/CMock'...
Cloning into '/root/esp-idf/components/esp_phy/lib'...
Cloning into '/root/esp-idf/components/esp_wifi/lib'...
```

图 6.1.3.5 更新子模块

一般来讲，在克隆 IDF 时会自动更新子模块，为了保险起见，最好重新更新子模块。

4, 运行安装脚本和设置环境变量

root@DESKTOP-QH7611H:~/esp-idf# 目录下输入两条命令“./install.sh”和“source export.sh”，它们分别是安装 IDF 环境和设置环境变量，设置成功之后系统提示“../export.sh”信息表示安装及设置环境变量成功如下图所示。

```
root@DESKTOP-QH7611H: ~/esp-idf
Successfully installed idf-component-manager-1.4.1
All done! You can now run:
    ./export.sh

root@DESKTOP-QH7611H: ~/esp-idf#
root@DESKTOP-QH7611H: ~/esp-idf#
```

图 6.1.3.6 IDF 安装环境

上图提示“../export.sh”信息表示安装成功，此时我们在子系统目录下发现了.espressif 隐藏文件夹，这是 IDF 环境搭建时生成的文件。

接着输入“source export.sh”设置环境变量，如下图所示。

```
root@DESKTOP-QH7611H: ~/esp-idf
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
    idf.py build

root@DESKTOP-QH7611H: ~/esp-idf#
```

图 6.1.3.7 设置环境变量

上图提示“idf.py build”信息表示设置环境变量成功。**请注意，每次进入 Ubuntu 22.04.2 LTS 子系统时，用户必须在 root@DESKTOP-QH7611H:~/esp-idf# 目录下输入“source export.sh”**

来操作环境变量，否则在生成固件时可能会报错。

为了解决上述的问题，作者在启动 Ubuntu 22.04.2 LTS 子系统时，在`~/.bashrc`直接执行`export.sh`文件，来设置 ESP-IDF 环境变量。

`~/.bashrc`是一个在 Unix 和 Linux 系统中非常重要的 Shell 初始化文件。这个文件在 bash shell（Bourne Again SHell）启动时被读取（系统启动时），用户可以在这里定义别名、函数、环境变量、路径等，所有这些在以后启动的 bash shell 中都可用。

这个文件通常位于系统的 home 目录下（`~/.bashrc`），用户可以使用文本编辑器（如 vim、nano 或 emacs）来编辑它。在此文件最后一行命令下添加以下命令，如下所示：

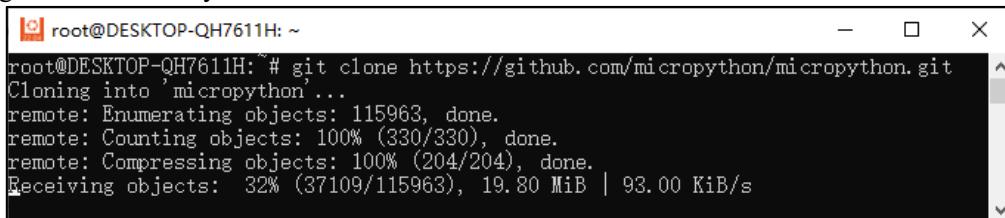
```
source ./esp-idf/export.sh
```

这样我们在子系统启动时，就无需进去 ESP-IDF 源码库设置环境变量，启动时直接运行`export.sh`文件设置 ESP-IDF 的环境变量了。

到了这里，总共花费了作者一天的时间，因为 Github 是国外的服务器，国内的网路想要从 Github 下载资料，必须有更良好的网络才行，否则即使知道 ESP-IDF 搭建环境步骤，也未必能搭建成功。另外，作者还提供了第二种 ESP-IDF 环境搭建方法，请参考这个[博客的文章](#)，他是从国内的 gitee 仓库克隆下来的，这样我们下载时就比较快速了。但在子模块更新时可能会遇到“gitee Username for 'https://gitee.com':”这样的提示信息，需要大家自行解决。

6.1.4 搭建 MicroPython 开发环境

在`root@DESKTOP-QH7611H:~/`目录下输入“`git clone https://github.com/micropython/micropython.git`”克隆 MicroPython 库，如下图所示。

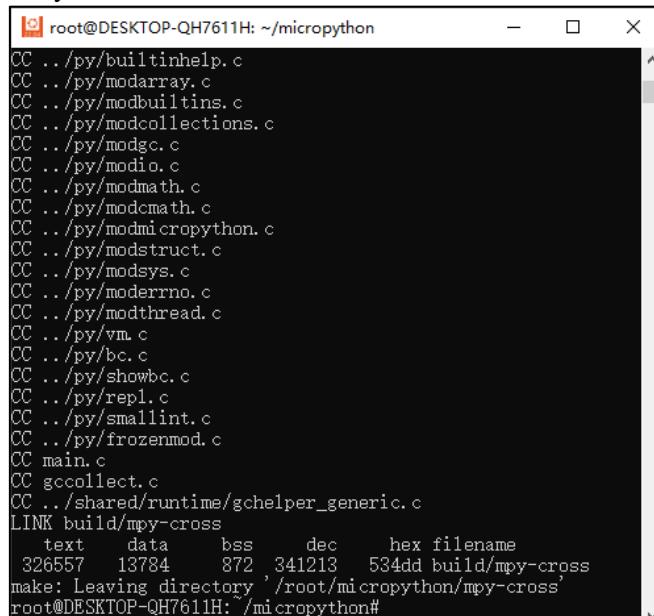


```
root@DESKTOP-QH7611H: ~
root@DESKTOP-QH7611H: ~# git clone https://github.com/micropython/micropython.git
Cloning into 'micropython'...
remote: Enumerating objects: 115963, done.
remote: Counting objects: 100% (330/330), done.
remote: Compressing objects: 100% (204/204), done.
Receiving objects: 32% (37109/115963), 19.80 MiB | 93.00 KiB/s
```

图 6.1.4.1 克隆 MicroPython 库

克隆成功之后，在此目录下输入“`cd micropython`”进入`micropython`库，然后输入“`git submodule update --init --recursive`”更新 MicroPython 子模块。

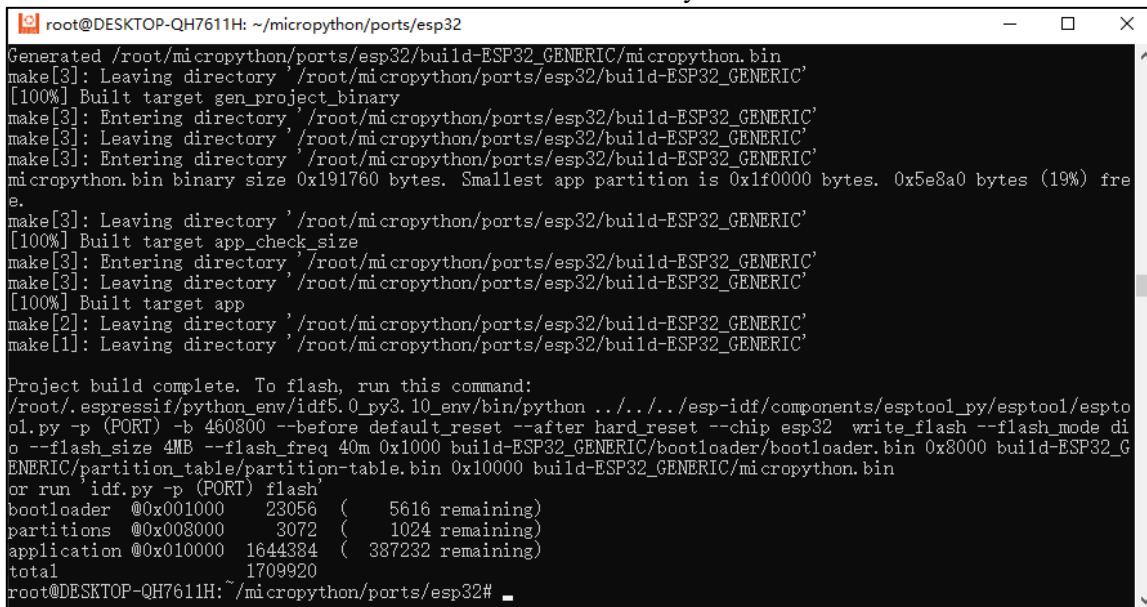
子模块更新成功之后，在`root@DESKTOP-QH7611H:~/micropython#`目录下输入“`make -C mpy-cross`”构建 MicroPython 交叉编译器，以便将一些内置脚本预编译为字节码，如下图所示。



```
root@DESKTOP-QH7611H: ~/micropython
CC .. /py/builtinhelp.c
CC .. /py/modarray.c
CC .. /py/modbuiltins.c
CC .. /py/modcollections.c
CC .. /py/modgc.c
CC .. /py/modio.c
CC .. /py/modmath.c
CC .. /py/modmath.c
CC .. /py/modmicropython.c
CC .. /py/modstruct.c
CC .. /py/modsys.c
CC .. /py/moderrno.c
CC .. /py/modthread.c
CC .. /py/vm.c
CC .. /py/bc.c
CC .. /py/showbc.c
CC .. /py/repl.c
CC .. /py/smallint.c
CC .. /py/frozenmod.c
CC main.c
CC gccollect.c
CC .. /shared/runtime/gchelper_generic.c
LINK build/mpy-cross
      text     data     bss     dec     hex filename
 326557    13784     872   341213   534dd build/mpy-cross
make: Leaving directory '/root/micropython/mpy-cross'
root@DESKTOP-QH7611H: ~/micropython#
```

图 6.1.4.2 构建 MicroPython 交叉编译

构建 MicroPython 交叉编译器完成之后，就可以构建 MicroPython ESP32 固件了。首先在 root@DESKTOP-QH7611H:~/micropython 目录下输入“cd ports/esp32”进入 MicroPython ESP32 编译工程，在此目录下输入“make”构建 ESP32 MicroPython 固件，如下图所示。



```

root@DESKTOP-QH7611H: ~/micropython/ports/esp32
Generated /root/micropython/ports/esp32/build-ESP32_GENERIC/micropython.bin
make[3]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
[100%] Built target gen_project_binary
make[3]: Entering directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
make[3]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
make[3]: Entering directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
micropython.bin binary size 0x191760 bytes. Smallest app partition is 0x1f0000 bytes. 0x5e8a0 bytes (19%) free.
make[3]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
[100%] Built target app_check_size
make[3]: Entering directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
make[3]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
[100%] Built target app
make[2]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'
make[1]: Leaving directory '/root/micropython/ports/esp32/build-ESP32_GENERIC'

Project build complete. To flash, run this command:
/root/.espressif/python_env/idf5.0_py3.10_env/bin/python ../../../../esp-idf/components/esptool_py/esptool/esptool.py -p (PORT) -b 460800 --before default_reset --after hard_reset --chip esp32 write_flash --flash_mode dio --flash_size 4MB --flash_freq 40m 0x1000 bootloader/bootloader.bin 0x8000 build-ESP32_GENERIC/partition_table/partition-table.bin 0x10000 build-ESP32_GENERIC/micropython.bin
or run idf.py -p (PORT) flash
bootloader @0x001000 23056 ( 5616 remaining)
partitions @0x008000 3072 ( 1024 remaining)
application @0x010000 1644384 ( 387232 remaining)
total 1709920
root@DESKTOP-QH7611H: ~/micropython/ports/esp32#

```

图 6.1.4.3 编译 ESP32 MicroPython 固件

根据上图所示，系统在 esp32\build-ESP32_GENERIC 文件夹下生成了三个 bin 文件，它们分别为 bootloader.bin、partition-table.bin 和 micropython.bin。然后，我们使用 flash_download_tool_3.9.5.exe 工具将这三个 bin 文件烧录到 ESP32 芯片中，设置方法如下所示：

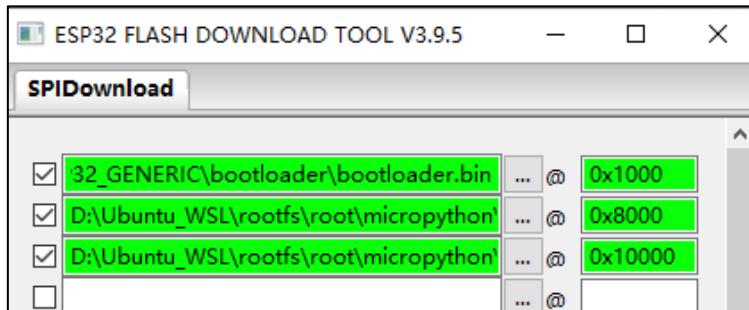


图 6.1.4.4 烧录 bin 文件的地址设置

根据上图中的信息，我们可以得知烧录 bin 文件的地址是根据图 5.1.4.4 编译固件成功后的系统提示信息来进行的。根据提示信息，我们需要将生成的 bin 文件烧录到对应的地址中。另外，我们也可以使用 esp32\build-ESP32_GENERIC 文件夹下的 firmware.bin 文件，这个文件是 bootloader.bin、partition-table.bin 和 micropython.bin 三个文件的总 bin 文件：

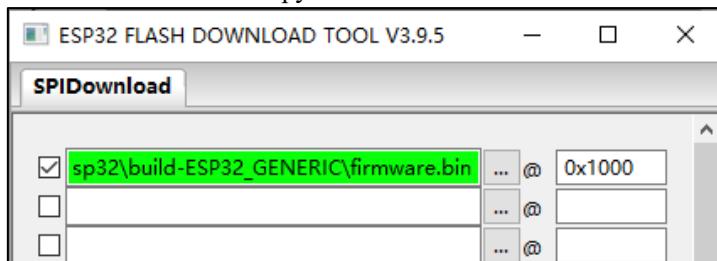


图 6.1.4.5 烧录总 bin 文件

注意：此固件虽然能满足 ESP32 在 MicroPython 下的运作，但它是有缺陷的。它并没有适配芯片的资源，如，是否挂载 PSRAM、Flash/PSRAM 的大小、时钟是否调节到 240MHz 等等。如果我们不去配置的话，系统可能选择默认的配置，如时钟只能达到 160MHz、Flash 大小为 2MB 等。这种默认的配置无法体现该芯片的价值，所以作者在后续的教程中，会详细讲解如何

配置一个与手上芯片相匹配的固件。

6.1.5 MicroPython 源文件结构分析

在上小节中，我们在子系统的 Ubuntu 下克隆了 MicroPython 源码包，并且在这个源码包下编译出了 ESP32 MicroPython 固件。下面，作者重点介绍这个源码包文件架构及文件功能说明。

MicroPython 的基础源程序基本由 C 语言编写的，在编译过程中，使用了少量的 Py 脚本，用于在预编译过程中处理字符串。如下表中展示了 MicroPython 项目的源码文件结构。

一级子目录	功能说明
docs	MicroPython 参考文档，需配置到 Sphinx 的整个文档网站
drivers	通过软实现的一个硬件驱动，基于 py 的架构，使用标准 C 实现的 Python 模块（C+Python）
examples	一些用 Py 编写的实验脚本，自编写的 C 模块可在此处实现
extmod	一些不需要在内核中实现的扩展模块，如硬件的 machine_spi，解析 json 等
lib	给 ports 目录下各平台提供的第三方库
logo	一些关于 Micro Python 的图片素材
mpy-cross	MicroPython 解释器的项目目录，将脚本文件编译成 Py 可直接执行的字节码
ports	MicroPython 针对具体运行平台的移植工程，包含 ESP32、STM32 等
py	Python 解释器内核实现的抽象代码
tests	MicroPython 框架测试脚本
tools	各类脚本辅助工具，如，mpy-tools.py 和 conf.py 编译 MicroPython 过程中调用

表 6.1.5.1 MicroPython 项目的源文件结构

docs 目录：可生成 MicroPython 文档，可以在 GitHub 仓库中找到 MicroPython/docs 目录，并安装相关的工具来生成 HTML，如下图所示。

Building the documentation locally

If you're making changes to the documentation, you may want to build the documentation locally so that you can preview your changes.

Install Sphinx, and optionally (for the RTD-styling), sphinx_rtd_theme, preferably in a virtualenv:

```
pip install sphinx
pip install sphinx_rtd_theme
```

In `micropython/docs`, build the docs:

```
make html
```

You'll find the index page at `micropython/docs/build/html/index.html`.

图 6.1.5.1 命令生成 MicroPython 文档

从上述可知，首先使用 pip 命令安装 sphinx 和 sphinx_rtd_theme 工具，然后在子系统上使用 cd 命令跳到 `micropython/docs` 目录下，最后在此目录下输入“make html”命令构建 html 文档。

py 目录：在 MicroPython 的开发过程中，为了确保与 Python 标准库的一致性，我们不会修改 py 目录中的任何源代码。这是因为 py 目录存放着 Python 内核的全部源代码，是实现 Python 功能的重要基础。

lib 目录和 **extmod** 目录：它们都包含了 MicroPython 扩展功能的组件，但它们的分工有所不同。**lib** 目录中的内容是与 MicroPython 无关的，它们针对特定的微控制器平台，并可以独立地在这些平台上运行，以支持 MicroPython 功能的实现。它们位于 MicroPython 的下层，例如，包括各个微控制器平台的固件库、基本的文件系统协议栈和 USB 协议栈等。而 **extmod** 目录中实现的是扩展模块，这些模块都是基于 MicroPython 实现的，并在 MicroPython 内部实现扩展功能，例如，`machine_spi` 模块就是在 MicroPython 的 `machine` 类中实现的 SPI 类模块。

对于微控制器开发者最有用也是最直接操作的目录是 **ports**，里面存在了 MicroPython 支持的所有微控制器的移植代码，如下图所示。

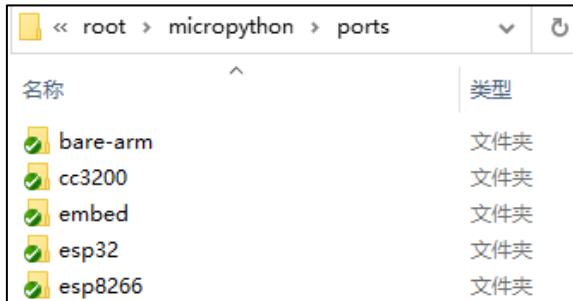


图 6.1.5.2 MicroPython 已经支持的移植平台

在具体微控制器平台的目录下，存放了各自对 MicroPython 的底层移植源文件以及移植工程的 `Makefile` 文件。因此，在 6.1.4 小节编译固件时，作者只输入“make”命令即可完成 ESP32 MicroPython 固件的编译和构建。

6.2 编译 ESP32-S3 固件

上小节中，我们直接 `make` 就可以构建 ESP32 固件了，`make` 这是 Make 的基本命令，用于生成目标文件。执行 `make` 命令时，Make 会自动查找当前目录下的 `Makefile` 文件（如果真的要细说 `make` 命令和 `Makefile`，可能需要客户了解 linux 相关知识了）。

`Makefile` 是用于自动编译和构建软件项目的工具。它使用简单的规则来告诉编译器如何构建软件。`Makefile` 包含一组命令，这些命令用于编译、链接和构建源代码文件。

在 ESP32-S3 的固件编译过程中，可以在 `ports/esp32` 目录中找到一个 `Makefile` 文件。这个 `Makefile` 文件包含了编译 ESP32-S3 固件的规则和指令。通过编辑 `Makefile` 文件，可以设置要编译的型号，例如 `GENERIC_S3`，然后保存文件后在终端使用 `make` 命令开始编译适用于对应板型的固件。

为了用户能够了解 MicroPython 固件编译，下面作者简单讲解一下 `ports/esp32` 目录 `Makefile` 文件到底有什么作用，该文件的部分内容如下所示：

```

ifdef BOARD_DIR
# (1) 自定义板路径
BOARD ?= $(notdir $(BOARD_DIR:/=))
else
# (2) 如果命令行中未给出，则默认为 ESP32_GENERIC.
BOARD ?= ESP32_GENERIC
# (3) 得到对应芯片的编译路径
BOARD_DIR ?= boards/$(BOARD)
endif

ifeq ($(wildcard $(BOARD_DIR)/.),)
ifeq ($(findstring boards/GENERIC,$(BOARD_DIR)),boards/GENERIC)
$(warning The GENERIC* boards have been renamed to ESP32_GENERIC*)
endif
$(error Invalid BOARD specified: $(BOARD_DIR))
endif
# (5) 如果没有给出构建目录，使用默认文件名
ifneq ($(BOARD_VARIANT),)
BUILD ?= build-$(BOARD)-$(BOARD_VARIANT)
else
BUILD ?= build-$(BOARD)

```

```
endif

# (6) 串口设备波特率
PORT ?= /dev/ttyUSB0
BAUD ?= 460800
# Python3
PYTHON ?= python3

.PHONY: all clean deploy erase submodules FORCE

CMAKE_ARGS =
# (5) 使用 C 模块，把自己的 C 驱动编译成 MicroPython 调用的 API
ifdef USER_C_MODULES
    CMAKE_ARGS += -DUSER_C_MODULES=${USER_C_MODULES}
endif

IDFPY_FLAGS += -D MICROPY_BOARD=${BOARD} -D MICROPY_BOARD_DIR=${abspath
${BOARD_DIR})} ${CMAKE_ARGS}

ifdef FROZEN_MANIFEST
    IDFPY_FLAGS += -D MICROPY_FROZEN_MANIFEST=${FROZEN_MANIFEST}
endif

# (6) 设置 PSRAM SPI 模式、D2WD 等
ifdef BOARD_VARIANT
    IDFPY_FLAGS += -D MICROPY_BOARD_VARIANT=${BOARD_VARIANT}
endif

HELP_BUILD_ERROR ?= "See
\033[1;31mhttps://github.com/micropython/micropython/wiki/Build-
Troubleshooting\033[0m"

define RUN_IDF_PY
    idf.py ${IDFPY_FLAGS} -B ${BUILD} -p ${PORT} -b ${BAUD} ${1}
endef

all:
    idf.py ${IDFPY_FLAGS} -B ${BUILD} build || (echo -e ${HELP_BUILD_ERROR});;
false)
@${PYTHON} makeimg.py \
    ${BUILD}/sdkconfig \
    ${BUILD}/bootloader/bootloader.bin \
    ${BUILD}/partition_table/partition-table.bin \
    ${BUILD}/micropython.bin \
    ${BUILD}/firmware.bin \
    ${BUILD}/micropython.uf2

${BUILD}/bootloader/bootloader.bin ${BUILD}/partition_table/partition-table.bin
${BUILD}/micropython.bin: FORCE

clean:
    $(call RUN_IDF_PY,fullclean)
# 自动烧录
deploy:
    $(call RUN_IDF_PY,flash)
# 擦除指令
erase:
    $(call RUN_IDF_PY,erase-flash)
monitor:
    $(call RUN_IDF_PY,monitor)
size:
    $(call RUN_IDF_PY,size)
size-components:
    $(call RUN_IDF_PY,size-components)
size-files:
    $(call RUN_IDF_PY,size-files)
# 更新子模块
submodules:
```

```

@GIT_SUBMODULES=$$(idf.py $(IDFPY_FLAGS) -B ${BUILD} /submodules -D
ECHO_SUBMODULES=1 build 2>&1 | \
    grep '^GIT_SUBMODULES=' | cut -d= -f2); \
${MAKE} -f ../../py/mkrules.mk GIT_SUBMODULES="${GIT_SUBMODULES}"
submodules

```

上述源码中的（1）表示要编译型号（如 make BOARD=GENERIC_S3 就是编译 S3 的 MicroPython 固件），如果在命令行中没有给出这个变量，则默认为 ESP32_GENERIC（2），即编译 ESP32 这款芯片；（3）表示编译路径，也就是在这个路径下编译哪些文件，如果（1）中没有设置 BOARD 变量，则系统会编译 boards/ESP32_GENERIC 路径下的文件（如下图所示）；（4）表示构建文件夹名称，这个文件夹包含系统编译产生的文件都会保存在这个文件夹中；（5）表示是否使用 C 模块，也就是说，使用 C 驱动编译成 MicroPython 可调用的 API，这个方法我们到第六章时候讲解；（6）表示编译条件，如挂在 PSRAM 的 SPI 模式为 SPIRAM_OCT。

名称	类型	大小
board.json	JSON 文件	1 KB
board.md	Markdown File	1 KB
mpconfigboard.cmake	CMAKE 文件	2 KB
mpconfigboard.h	C/C++ Header F...	1 KB
sdkconfig.d2wd	D2WD 文件	1 KB
sdkconfig.ota	OTA 文件	1 KB
sdkconfig.unicore	UNICORE 文件	1 KB

图 6.2.1 boards/ESP32_GENERIC 路径下的文件

上图中的 mpconfigboard.cmake 文件用于指定编译哪些 sdkconfig 文件。sdkconfig 文件决定了系统应该编译和构建哪些功能（即加载项目的具体变量，通过具体变量让系统执行相应的代码），而 mpconfigboard.h 文件则用于定义全局宏定义，系统在运行时会根据这些宏定义的数值来执行相应的条件判断。

打开 mpconfigboard.cmake 文件，如下：

```

# 需执行的文件
set(SDKCONFIG_DEFAULTS
    boards/sdkconfig.base
    boards/sdkconfig.ble
)
# 判断是否开启 D2WD
if(MICROPY_BOARD_VARIANT STREQUAL "D2WD")
    set(SDKCONFIG_DEFAULTS
        ${SDKCONFIG_DEFAULTS}
        boards/ESP32_GENERIC/sdkconfig.d2wd
    )

    list(APPEND MICROPY_DEF_BOARD
        MICROPY_HW MCU_NAME="ESP32-D2WD"
    )
endif()
# 省略以下代码

```

从上述内容可知，编译 ESP32 MicroPython 固件时，需要执行 boards/ 路径下的 sdkconfig.base 和 sdkconfig.ble 文件以及条件判断下的 sdkconfig.x 文件。根据这些项目的具体变量，系统将有意识地编译哪些代码。下面，作者将以 sdkconfig.spiram_sx 为例来讲解这个文件的作用。

打开 ports/esp32/boards/sdkconfig.spiram_sx 文件，如下：

```

# PSRAM 模式
CONFIG_SPIRAM_MODE_QUAD=Y
.....
# PSRAM 读取速率
CONFIG_SPIRAM_SPEED_80M=Y
# 使能 PSRAM

```

```
CONFIG_SPIRAM=y
```

此文件用于配置 ESP32 芯片是否挂载 PSRAM，以及设置 SPI 模式、速率等相关项目的具体变量。例如，正点原子 ESP32-S3 最小系统板的主控使用的是 ESP32-S3-WROOM-1-N16R8 模组，根据《esp32-s3-wroom-1_wroom-1u_datasheet_cn.pdf》模组数据手册第三页表 1 所示，该模组的 PSRAM 使用 Octal SPI 模式，因此在本章节编译 ESP32-S3 固件时，需要开启 PSRAM，并将 SPI 模式设置为 Octal SPI。

ESP32-S3-WROOM-1-N16R8 模组内置 16MB Flash（Octal SPI）和 8MB PSRAM（Octal SPI），可运行最高时钟频率为 240MHz。这些信息来自《esp32-s3_datasheet_cn.pdf》和《esp32-s3-wroom-1_wroom-1u_datasheet_cn.pdf》这两份数据手册。

根据上述的要求，作者划分几个步骤来讲解。

1. 修改 sdkconfig.board 文件

打开 ports/esp32/borads/ESP32_GENERIC_S3 路径下打开 sdkconfig.board 文件，此文件的内容如下：

```
CONFIG_ESPTOOLPY_FLASHMODE_QIO=y
CONFIG_ESPTOOLPY_FLASHFREQ_80M=y
CONFIG_ESPTOOLPY_AFTER_NORESET=y
# Flash 大小
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=
CONFIG_ESPTOOLPY_FLASHSIZE_8MB=y
CONFIG_ESPTOOLPY_FLASHSIZE_16MB=
CONFIG_PARTITION_TABLE_CUSTOM=y
# 分区表名称
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions-8MiB.csv"
```

此文件默认设置 Flash 大小为 8MB，显然不符合 ESP32-S3-WROOM-1-N16R8 模组内置 16MB Flash 大小，因此，把 CONFIG_ESPTOOLPY_FLASHSIZE_16MB 配置为 y，并且修改分区表名称为“partitions-16MiB.csv”，修改后如下：

```
CONFIG_ESPTOOLPY_FLASHMODE_QIO=y
CONFIG_ESPTOOLPY_FLASHFREQ_80M=y
CONFIG_ESPTOOLPY_AFTER_NORESET=y
# Flash 大小为 16MB
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=
CONFIG_ESPTOOLPY_FLASHSIZE_8MB=
CONFIG_ESPTOOLPY_FLASHSIZE_16MB=y
CONFIG_PARTITION_TABLE_CUSTOM=y
# 分区表名称
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions-16MiB.csv"
```

2. 修改 partitions-16MiB.csv 分区表

因为正点原子 ESP32-S3 MicroPython 固件使用了乐鑫 AI 库，因此这个 AI 库存储在 Flash 内需要大量的内存。因此，我们将分区表中的 factory 子分区（存储代码的区域）设置为 6M。

修改前的分区表：

```
# Notes: the offset of the partition table itself is set in
# $IDF_PATH/components/partition_table/Kconfig.projbuild.
# Name,      Type,    SubType,    Offset,     Size,   Flags
nvs,        data,      nvs,       0x9000,    0x6000, ,
phy_init,   data,      phy,       0xf000,    0x1000, ,
factory,   app,      factory,   0x10000,   0x1F0000, # 1M 内存
vfs,        data,      fat,       0x200000,  0xE00000, # 14M 内存
```

修改后的分区表

```
# Notes: the offset of the partition table itself is set in
# $IDF_PATH/components/partition_table/Kconfig.projbuild.
# Name,      Type,    SubType,    Offset,     Size,   Flags
nvs,        data,      nvs,       0x9000,    0x6000, ,
phy_init,   data,      phy,       0xf000,    0x1000, ,
factory,   app,      factory,   0x10000,   0x600000, # 6M 内存
vfs,        data,      fat,       0x700000,  0x900000, # 9M 内存
```

到了这里，我们成功让 MicroPython 源代码适配了 ESP32-S3-WROOM-1-N16R8 模组。有小伙伴会问，为什么 CONFIG_ESPTOOLPY_FLASHMODE_QIO 还是为 y 呢？其实，在编译固件

时，通过在 make 命令后面加上“BOARD_VARIANT=SPIRAM_OCT”命令就可以覆盖这个项目具体变量。大家可以打开 ports/esp32/boards/ESP32_GENERIC_S3 路径下的 mpconfigboard.cmake 文件，如下所示：

```
set(IDF_TARGET esp32s3)

set(SDKCONFIG_DEFAULTS
    boards/sdkconfig.base
    boards/sdkconfig.usb
    boards/sdkconfig.ble
    boards/sdkconfig.spiram_sx
    boards/ESP32_GENERIC_S3/sdkconfig.board
)

if(MICROPY_BOARD_VARIANT STREQUAL "SPIRAM_OCT")
    set(SDKCONFIG_DEFAULTS
        ${SDKCONFIG_DEFAULTS}
        boards/sdkconfig.240mhz
        boards/sdkconfig.spiram_oct
    )

    list(APPEND MICROPY_DEF_BOARD
        MICROPY_HW_BOARD_NAME="Generic ESP32S3 module with Octal-SPIRAM"
    )
endif()
```

根据上述文件内容，当 MICROPY_BOARD_VARIANT 变量等于 SPIRAM_OCT 时，将编译 sdkconfig.240mhz 和 sdkconfig.spiram_oct 文件。sdkconfig.240mhz 文件用于将芯片的时钟频率配置为 240MHz，而 sdkconfig.spiram_oct 文件则用于将外挂 PSRAM 的 SPI 模式配置为 Octal SPI。因此，在系统编译 sdkconfig.spiram_oct 时，将禁用 CONFIG_SPIRAM_MODE_QUAD 这个项目具体变量，并将 CONFIG_SPIRAM_MODE_OCT 这个项目具体变量设置为 y。以下是 sdkconfig.spiram_oct 文件的内容：

```
# MicroPython on ESP32-S2 and ESP32-PAD1_subscript_3, ESP IDF configuration with
# SPIRAM support in Octal mode
CONFIG_SPIRAM_MODE_QUAD=
CONFIG_SPIRAM_MODE_OCT=y
```

至此，MicroPython 源代码适配 ESP32-S3-WROOM-1-N16R8 模组完成，接下来，在 ports/esp32/ 路径下输入“make BOARD=ESP32_GENERIC_S3 BOARD_VARIANT=SPIRAM_OCT”命令构建 ESP32-S3-WROOM-1-N16R8 模组 MicroPython 固件，如下图所示。

```
Project build complete. To flash, run this command:
/root/.espressif/python_env/idf5.0_py3.10_env/bin/python ../../components/esptool_py/esptool/esptool.py -p (P0
RT) -b 460800 --before default_reset --after no_reset --chip esp32s3 write_flash --flash_mode dio --flash_size 16MB --fl
ash_freq 80m 0x0 build-ESP32_GENERIC_S3-SPIRAM_OCT/bootloader/bootloader.bin 0x8000 build-ESP32_GENERIC_S3-SPIRAM_OCT/part
ition_table/partition-table.bin 0x10000 build-ESP32_GENERIC_S3-SPIRAM_OCT/micropython.bin
or run idf.py -p (PORT) flash
bootloader @0x00000000 18672 ( 14096 remaining)
partitions @0x0080000 3072 ( 1024 remaining)
application @0x0100000 1557056 ( 474560 remaining)
total 1622592
root@DESKTOP-QH7611H:~/micropython/ports/esp32#
```

图 6.2.2 ESP32-S3-WROOM-1-N16R8 模组 MiroPython 固件

根据上图所示，系统会在 build-ESP32_GENERIC_S3-SPIRAM_OCT 文件夹下生成四个 bin 文件，它们分别是 bootloader.bin、partition-table.bin、micropython.bin 和 firmware.bin（总 bin）。接着，我们使用 flash_download_tool.exe 工具将这三个 bin 文件烧录到 ESP32-S3-WROOM-1-N16R8 模组中。设置 flash_download_tool 的方法请参考图 5.1.4.5 所示。需要注意的是，bootloader.bin、partition-table.bin 和 micropython.bin 设置的下载地址分别为 0x0000、0x8000 和 0x10000。如果烧录的是 firmware.bin（总 bin），则将烧录地址设置为 0x0000。当然，我们也可以借助 Thonny 软件来烧录这个固件，如下图所示。

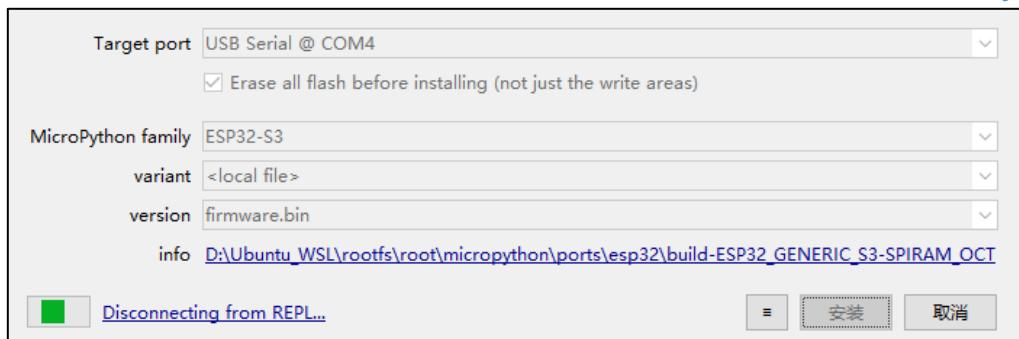


图 6.2.3 烧录 ESP32-S3 MicroPython 固件

烧录完成后，关闭烧录界面，切换 Thonny 软件的解释器。在 Thonny 软件的 Shell 下提示固件信息，如下图所示。

```
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:0xf8c
load:0x403c9700,len:0xb3c
load:0x403cc700,len:0x2dc8
entry 0x403c989c
MicroPython v1.22.0-preview.164.gfce8d9fd5.dirty on 2023-11-23; Generic ESP32S3 module with Octal-SPIRAM with ESP32S3
Type "help()" for more information.
MicroPython v1.22.0-preview.164.gfce8d9fd5.dirty on 2023-11-23; Generic ESP32S3 module with Octal-SPIRAM with ESP32S3
Type "help()" for more information.
>>>
```

图 6.2.4 固件提示信息

在 Shell 交互环境下输入以下代码，查看固件资源信息，如下图所示。

```
>>> import machine
import esp
import gc
import micropython

if __name__ == '__main__':

    freq = machine.freq()                                     # 获取CPU当前工作频率
    print(f'当前系统时钟{freq}')                            # 打印CPU工作频率
    print(f'内部flash大小{esp.flash_size()/1024/1024}MB') # 打印FLASH大小
    gc.mem_alloc()
    psram_tocal = gc.mem_free()
    print(f'挂在PSRAM大小{psram_tocal/1024/1024}MB')      # 打印PSRAM剩余大小

当前系统时钟240000000
内部flash大小16.0MB
6160
挂在PSRAM大小7.930084MB
>>> |
```

图 6.2.5 提示固件资源信息

从上图中可知，这个固件的系统时钟被设置为 240MHz，内部 Flash 大小为 16MB，PSRAM 经过申请和释放操作，最后打印剩余空间为 7.930084MB 约等于 8MB。

6.3 总结

不妨读者思考一下 MicroPython 的 ESP32-S3 固件为什么能驱动 ESP32-S3 芯片，并且实现相关应用。

在编译 MicroPython 的 ESP32-S3 固件之前，需要先搭建 ESP-IDF 环境。这是因为 ESP-IDF 库为开发者提供了一套完善的 API，使得开发者可以使用高级语言（如 C/C++）来操作硬件，而不需要直接对硬件寄存器进行操作。这一设计大大降低了开发的难度，使得开发者可以更加专注于实现应用功能，而不必花费大量时间在底层的硬件操作上。

MicroPython 的 ESP32-S3 固件编译也是基于这一思想进行的。它在 ESP-IDF 库的基础上进

行了 Python 的封装，使得用户可以使用 Python 的语法来实现对 ESP32-S3 芯片的控制。这种封装方式不仅使得使用 MicroPython 进行开发变得更加简单和方便，同时也提高了 MicroPython 的可读性和易用性。

总的来说，MicroPython 的 ESP32-S3 固件编译是基于 ESP-IDF 库进行的，它为开发者提供了一种更加高级、更加方便的方式来控制 ESP32-S3 芯片，使得开发者可以更加专注于实现应用功能。

第七章 MicroPython 组件扩展

使用原生的 MicroPython 进行开发时，可能会感觉束手束脚，就像在狭窄的小路上行走，前方有着各种陡峭的山坡和深邃的沟渠。有时，你会觉得官方提供的功能不足以满足你的需求，或者你发现这些功能并不能完全符合你的工作场景。这时，你就可以选择亲手打造自己的 C 模块，将其融入 MicroPython 中，仿佛在狭窄的小路上开辟出一条自己的道路，前方是开阔的平原和明亮的阳光。你可以按照自己的想法和需求，设计和实现适合自己的 Python 函数调用，让它们像你手中的利剑一样，挥舞在代码的世界中。

本章分为如下几个小节：

- 7.1 组件扩展原理
- 7.2 组件扩展辅助工具

7.1 组件扩展原理

7.1.1 组件扩展方式

很多人会疑惑，C 语言与 Python 是两种不同的语言，MicroPython 如何调用 C 语言实现的函数在 Python 下调用的呢？这个问题关键在于，如何使用 C 语言的形式在 MicroPython 源代码中表述函数的入参与出参，比如 Python 实现一个 A 变量与 B 变量相加函数，它的实现代码如下：

```
def add(a, b):
    return a + b
```

这个函数有两个入参和一个返回参数，此时如果使用 C 语言表示该函数的输入输出参数，就可以使用 C 函数对接到 MicroPython 当中。在我们讲解 C 模块原理之前，我们先了解 MicroPython C 模块的组件扩展方式。

MicroPython C 模块有两种组件扩展方式，它们分别为模块扩展，模块+类扩展，这两种形式有什么区别呢？下面作者使用一个示意图来讲解。

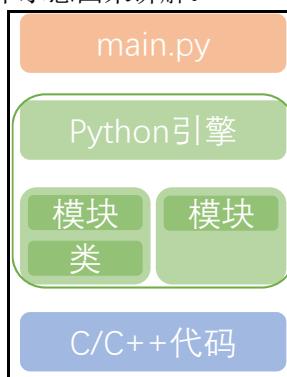


图 7.1.1.1 组件扩展方式

从上图可知，在 main.py 文件下可调用两种类型的 Python 引擎，它们分别为模块扩展，模块+类扩展，下面作者详细说明这两种组件扩展有何区别，如下所示：

①：模块扩展：在这种方式下，扩展组件以模块的形式提供，用户在使用时直接导入模块即可。这种方式比较简单，只需要在模块中定义所需的函数和变量。模块扩展调用示例如下所示：

```
import cexample
cexample.cppfunc() # 调用模块的方法
```

这种 Python 引擎类似于在 Python 环境下新建一个 cexample.py 脚本，然后，在脚本下利用“de cppfunc”方式（Python 定义函数的流程）定义模块的方法，接着，在 main.py 脚本下导入 cexample 模块，最后，引导这个模块的方法（函数）。

②：模块+类扩展：在这种方式下，扩展组件通过模块和类的组合进行扩展。用户在使用时需要通过模块导入特定的类，然后使用这个类提供的功能。这种方式相对于模块扩展更加灵活，

可以更好地组织代码和提供面向对象的功能。模块+类扩展调用示例如下所示：

```
from cexample import Timer
tr = Timer()      # 实例化 Timer 类对象
tr.time()        # 调用 Timer 对象的方法
```

这种 Python 引擎类似于在 Python 环境下新建一个 cexample.py 脚本。然后，在脚本下定义一个 Timer 类对象，在这个类中定义属性与方法，接着，在 main.py 脚本下引入 cexample 模块中的 Timer 类，并实例化 Timer 类对象，最后，使用实例化 Timer 对象调用该类的方法与属性。

这两种扩展方式的主要区别在于代码组织和使用方式上。模块扩展方式更简单直接，适合提供一些基础的功能和函数。而模块+类扩展方式更加灵活，可以更好地组织和封装代码，提供更高级的功能和接口。需要注意的是，这两种扩展方式并不是互斥的，可以根据需要同时使用。例如，可以在一个扩展模块中同时使用模块扩展和模块+类扩展方式，以提供更加丰富和灵活的功能。

7.1.2 组件扩展原理解析

到了这里，我们已经了解了 C 模块组件的扩展方式。接下来，将重点讲解 MicroPython 提供的三个 C 模块实例，以便用户将来编写自己的 C 模块组件。这些实例位于 micropython\examples\usercmodule 目录下，它们分别是 cexample、cppexample 和 subpackage C 模块。

下面，就以 cexample C 模块为例。首先，打开 micropython\examples\usercmodule\cexample 目录。该目录包含三个文件，它们共同构成了一个简单的 MicroPython C 模块。如下图所示。

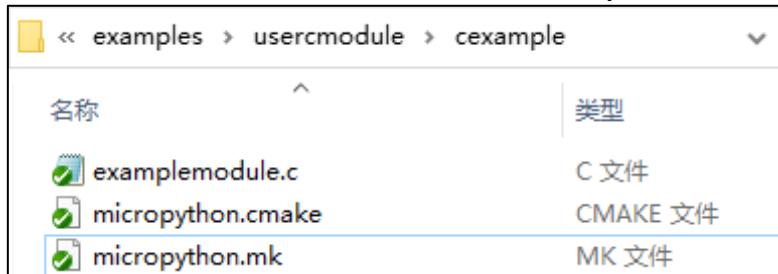


图 7.1.2.1 cexample 文件夹目录

上图中的 micropython.mk 是一个包含此模块的 Makefile 片段的脚本文件。其中，USERMOD_DIR 可用作 micropython.mk 模块目录的路径。在 Makefile 中，它应该被扩展为本地 make 变量，例如：CEXAMPLE_MOD_DIR:=\$(USERMOD_DIR)。同时，需要将模块源文件添加到 SRC_USERMOD 的扩展副本中，例如：SRC_USERMOD += \$(CEXAMPLE_MOD_DIR)/examplemodule.c。如果存在自定义的“CFLAGS”设置或包括文件夹来定义，这些应该添加到“CFLAGS_USERMOD”中。

```
# USERMOD_DIR 模块目录的路径,如 cexample/
CEXAMPLE_MOD_DIR := $(USERMOD_DIR)

# 将所有 C 文件添加到 SRC_USERMOD
SRC_USERMOD += $(CEXAMPLE_MOD_DIR)/examplemodule.c

# 如果有自定义编译器选项（例如 -I 添加目录以搜索头文件）,
# 则应将这些选项添加到 C 代码的 CFLAGS_USERMOD 和 C++ 代码的 CXXFLAGS_USERMOD
CFLAGS_USERMOD += -I$(CEXAMPLE_MOD_DIR)
```

在上面的源码中，我们看到了一个链接命令，它链接了 cexample 对象文件，最终生成了一个共享库。这个共享库可以被 MicroPython 解释器加载和使用。总的来说，micropython.mk 对于 MicroPython C 模块的作用主要是定义构建规则和编译选项，以用于构建和编译 C 模块。

micropython.cmake 是一个 CMake 配置文件，用于构建 MicroPython 模块。它包含了构建模块所需的 CMake 配置指令，例如定义库、添加源文件、设置编译选项等。通过使用 CMake，可以方便地构建和管理 MicroPython 模块的构建过程。如下代码所示：

```
# 创建一个 INTERFACE 库
add_library(usermod_cexample INTERFACE)

# 源文件添加到库中
```

```

target_sources(usermod_cexample INTERFACE
    ${CMAKE_CURRENT_LIST_DIR}/examplemodule.c
)

# 将当前目录添加为包含目录
target_include_directories(usermod_cexample INTERFACE
    ${CMAKE_CURRENT_LIST_DIR}
)

# 将我们的 INTERFACE 库链接到 usermod 目标
target_link_libraries(usermod INTERFACE usermod_cexample)

```

在 CMake 配置文件 micropython.cmake 中，需要定义一个 INTERFACE 库并将源文件关联起来，然后将其链接到 usermod 目标。该文件对于 MicroPython C 模块的作用是定义 CMake 配置，以确保 C 模块能够正确地编译和链接成可执行文件或库，并能够在 MicroPython 环境中正确地运行。

examplemodule.c 文件使用了模块和模块+类两种组件扩展方式。下面，作者分别来讲解这两种方式的实现原理。

1. 模块扩展实现原理

examplemodule.c 文件模块扩展部分代码如下所示：

```

/* 第一部分：添加所需 API 的头文件 */
#include "py/runtime.h"
#include "py/mphal.h"

/* 第二部分：实现功能 */
STATIC mp_obj_t example_add_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    /* 通过 Python 获取的第一个整形参数 arg_1 */
    int a = mp_obj_get_int(a_obj);
    /* 通过 Python 获取的第二个整形参数 arg_2 */
    int b = mp_obj_get_int(b_obj);

    /* 处理入参 arg_1 和 arg_2，并向 python 返回整形参数 ret_val */
    return mp_obj_new_int(a + b);
}

/* 使用 MP_DEFINE_CONST_OBJ_2 宏将函数添加到模块中 */
STATIC MP_DEFINE_CONST_OBJ_2(example_add_ints_obj, example_add_ints);

/* 第三部分：将模块注册到模块列表 */
STATIC const mp_rom_map_elem_t example_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR__name__), MP_ROM_QSTR(MP_QSTR_cexample) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&example_add_ints_obj) },
};

/* 把模块注册列表注册到 example_module_globals 字典对象当中 */
STATIC MP_DEFINE_CONST_DICT(example_module_globals,
                           example_module_globals_table);

/* 第四部分：定义模块对象 */
const mp_obj_module_t example_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&example_module_globals, /* 指向字典对象 */
};

/* 第五部分：注册模块以使其在 Python 中可用 */
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule);

```

上述是 examplemodule.c 文件的部分代码，它主要阐述了简单 MicroPython C 模块的模块扩展架构，首先作者把它这个架构划分为四个部分，如下解析：

第一部分：在.c 文件下添加 microPython 相关头文件，可用来引用相关的 API 函数。

第二部分：加法函数，使用 C 语言的方式实现功能，但需要使用 MicroPython API 函数对入参和出参进行转换，入参转换完成后，才能让 C 语言识别，出参转换成功后才能被 MicroPython 调用。最后在 MP_DEFINE_CONST_FUN_OBJ_2 宏的作用下，将函数添加到模块中，以便在 MicroPython 环境中调用。

第三部分：将模块注册到模块列表，然后在 MP_DEFINE_CONST_DICT 宏的作用下在 C 语言中创建一个常量字典，并将其添加到 MicroPython 模块中，以便在 Python 环境中直接访问和使用这些字典。这对于需要在 MicroPython 模块中定义和共享常量字典的情况非常有用。大家不妨回顾一下字典的作用，它通过键来访问字典中的值（这个值可用来指向某个函数的地址，这样我们根据这个地址调用函数了）。

第四部分：创建一个模块对象，然后对象的成员变量 globals 指向字典对象，接着在 MP_REGISTER_MODULE 宏的作用下将模块注册到 MicroPython 系统中，使其可以在 Python 环境中被导入和使用。

上述组件扩展方式是模块扩展，我们可在 Py 脚本下使用模块扩展的形式调用加法函数，如下示例所示：

```
# 导入模块
import cexample

"""

* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == "__main__":
    a = 5
    b = 10
    c = cexample.add_ints(a,b) # 调用自定义模块的加法函数
    print(c) # 输出 c 为 15
```

这种模块扩展类似于自己定义一个 cexample.py 文件，然后，在此文件下实现多个函数，最后，在 main.py 文件下导入 cexample 模块，并以模块名引用函数。

2. 模块+类扩展原理

examplemodule.c 文件模块+类扩展部分代码如下所示：

```
/* 添加所需 API 的头文件 */
#include "py/runtime.h"
#include "py/mphal.h"

/* (2) 定义 Timer 对象实例 */
typedef struct _example_Timer_obj_t
{
    /* 所有对象的基础地址 */
    mp_obj_base_t base;
    /* 开始时间的变量 */
    mp_uint_t start_time;
} example_Timer_obj_t;

/* (3) 对象的实现方法（函数） */
STATIC mp_obj_t example_Timer_time(mp_obj_t self_in)
{
    /* 获取 Timer 句柄 */
    example_Timer_obj_t *self = MP_OBJ_TO_PTR(self_in);

    /* 获取经过的时间，并将其作为 MicroPython 整数返回 */
    mp_uint_t elapsed = mp_hal_ticks_ms() - self->start_time;
    return mp_obj_new_int_from_uint(elapsed);
}
/* 使用 MP_DEFINE_CONST_FUN_OBJ_1 宏将函数添加到模块中 */
STATIC MP_DEFINE_CONST_FUN_OBJ_1(example_Timer_time_obj, example_Timer_time);

/* 构造函数 */
STATIC mp_obj_t example_Timer_make_new(const mp_obj_type_t *type, size_t n_args,
                                         size_t n_kw, const mp_obj_t *args) {
```

```

/* 分配新对象并设置类型 */
example_Timer_obj_t *self = mp_obj_malloc(example_Timer_obj_t, type);

/* 获取开始时间 */
self->start_time = mp_hal_ticks_ms();

/* 返回 Timer 句柄 */
return MP_OBJ_FROM_PTR(self);
}

/* 将模块注册到对象的模块列表 */
STATIC const mp_rom_map_elem_t example_Timer_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_time), MP_ROM_PTR(&example_Timer_time_obj) },
};

/* 把字典转换成对象 */
STATIC MP_DEFINE_CONST_DICT(example_Timer_locals_dict,
                           example_Timer_locals_dict_table);

/* (1) 定义了 Timer 类 */
MP_DEFINE_CONST_OBJ_TYPE(
    example_type_Timer,
    MP_QSTR_Timer,
    MP_TYPE_FLAG_NONE,
    make_new, example_Timer_make_new,
    locals_dict, &example_Timer_locals_dict
);

/* 将模块注册到模块列表 */
STATIC const mp_rom_map_elem_t example_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR__name__), MP_ROM_QSTR(MP_QSTR_cexample) },
    { MP_ROM_QSTR(MP_QSTR_Timer), MP_ROM_PTR(&example_type_Timer) },
};
/* 把模块注册列表注册到 example_module_globals 字典对象当中 */
STATIC MP_DEFINE_CONST_DICT(example_module_globals,
                           example_module_globals_table);

/* 定义模块对象 */
const mp_obj_module_t example_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&example_module_globals, /* 指向字典对象 */
};

/* 注册模块以使其在 Python 中可用 */
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule);

```

上述代码也不难理解，我们知道 Python 语言是面向对象的语言。正因为如此，在 Python 中创建一个类和对象是很容易的。类是用来描述具有相同的属性（变量）和方法（函数）的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。明白了这点，我们可以去看上述的代码到底怎么实现一个 Python 类的。

(1)它定义了一个名为“Timer”的类，这个类包含了一个对实例对象的操作 example_Timer_make_new，以及一个类的方法（函数）example_Timer_locals_dict；(2)处定义了 Timer 对象实例，它用来获取开始的运行时间；(3)处就是类的实现方法 example_Timer_time 即运行的函数。如下是模组+类扩展调用示例，如下代码所示：

```

from cexample import Timer

"""

* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == "__main__":

```

```

tr = Timer()          # 对象实例（调用 example_Timer_make_new 函数）
timer = tr.time()     # 调用对象的方法 example_Timer_time 获取运行时间
print(timer)          # 打印运行的时间

```

从上述源代码可以看出，C 模块的模块+类扩展实现方式类似于在定义一个 cexample.py 文件，并在该文件中实现一个 Timer 类，然后在该类中实现计算运行时间的方法，最后在 main.py 文件中导入 cexample 模块下的 Timer 类，并使用类名来引用该函数。

7.2 组件扩展辅助工具

RT-Thread 提供的 MicroPython [C 绑定代码自动生成器](#)是一个非常有用的工具，可以帮助开发者快速将 C 语言函数或模块集成到 MicroPython 环境中。通过这个工具，开发者可以轻松添加自己编写的 C 语言函数或者模块到 MicroPython 中，并被 Python 脚本调用。这大大扩展了原版 MicroPython 的能力，并且可以快速实现任何功能。

使用这个工具，开发者只需要简单几步操作，即可实现添加 C 绑定的功能。自动生成的 C 代码形式可以在 RT-Thread 的官方文档中找到。这个工具已经经过多次迭代，变得越来越完善，可以轻松加入到工程中。C 绑定代码自动生成器如下所示：

The screenshot shows the 'RT-Thread MicroPython C 绑定代码自动生成器' (RTT MicroPython C Binding Code Generator) interface. It has a header with a star icon and '186' reviews. Below the header is a message: '如果觉得有帮助，请帮我们 [点亮小星星](#)，您的支持是我们前进的动力！'. The main form has three input fields: '函数名' (Function Name) set to 'my_function', '输入参数数量' (Input Parameter Count) set to '0', and '函数返回值类型' (Function Return Type) set to '空-->None'. There is a checkbox '添加提示代码' (Add Hint Code) which is unchecked. At the bottom are two buttons: '点击生成 C 绑定代码' (Click to generate C binding code) in blue and '点击复制代码' (Click to copy code) in white. Below the form is a note: '添加一个 C 函数到 MicroPython 中有如下三个步骤，分别对应 Output 中的三个动作：'. A numbered list follows: 1. 将自动生成的同名 C 语言函数拷贝到 port/modules/user/moduserfunc.c 文件 2. 将该函数注册到用户模块列表 (_globals_table[]) 3. 追加关联该函数的 QSTR 到 port/genhdr/qstrdefs.generated.h 文件.

图 7.2.1 RTT 提供的 MicroPython C 绑定代码自动生成器

这个辅助工具很简单，只需填写要实现的函数名、传入的参数和函数的返回值类型，即可生成一个 MicroPython 函数模板。当然，核心代码需要开发者自行编写。

第二篇 入门篇

入门篇主要讲解了 ESP32S3 MicroPython 开发中 MicroPython machine 特定库的使用，包括 GPIO、RTC、PWM 和传感器数据读取等。

本篇分为如下几个章节：

- 1, LED 实验
- 2, KEY 实验
- 3, EXIT 实验
- 4, TIMER 实验
- 5, PWM 实验
- 6, SPILCD 实验
- 7, RTC 实验
- 8, SD 卡实验

第八章 LED 实验

本章使用 `machine.Pin` 类实现 ESP32-S3 IO 操作。`machine.Pin` 类是 `machine` 模块下的一个硬件类，它为 IO 提供配置和控制功能，为实现 IO 设备的操作方法。`Pin` 对象主要被用来控制输入/输出引脚，这些引脚也被称为 GPIO。每个 `Pin` 对象都与一个物理引脚相对应，它能驱动输出电压并读取输入电压。

本章分为以下几个小节：

- 8.1 `machine.Pin` 类
- 8.2 硬件设计
- 8.3 软件设计
- 8.4 下载验证

8.1 `machine.Pin` 类

`Pin` 类中提供了设置引脚模式（输入/输出）的方法，以及获取和设置数字逻辑值（0 或 1）的方法。要创建一个 `Pin` 对象，我们需要提供一个标识符，这个标识符能够明确地指定一个特定的输入/输出引脚。这个标识符可以是整数、字符串，或者是一个包含端口和引脚号码的元组。在 MicroPython 中，引脚标识符是一个由代号和引脚号组成的元组，例如在 `Pin("IO1", 1)`, `Pin.OUT` 中的 ("IO1", 1)。

1, `machine.Pin` 类的构造对象

`Pin` 的构造对象方法如下：

```
class machine.Pin(id, mode=-1, pull=-1, value=None)
```

使用示例： led = `machine.Pin(1, Pin.OUT, value = 1)`

该构造方法的参数描述，如下表所示。

参数	描述	
<code>id</code>	由用户自定义的引脚名和 <code>Pin</code> 设备引脚号组成，如 ("IO1", 1), "IO1" 为用户自定义的引脚名，1 为 ESP32S3 开发板的 LED 引脚号	
<code>mode</code>	输入输出模式	
	<code>Pin.IN</code>	输入模式
	<code>Pin.OUT</code>	输出模式
	<code>Pin.OPEN_DRAIN</code>	开漏模式
<code>pull</code>	如果指定的引脚连接了上拉下拉电阻	
	<code>None</code>	没有上拉或者下拉电阻
	<code>Pin.PULL_UP</code>	使能上拉电阻
	<code>Pin.PULL_DOWN</code>	使能下拉电阻
<code>value</code>	引脚电平值。输出：0 为低电平，1 为高电平；输入：无须参数，返回电平	

表 8.1.1 `Pin` 类构造方法的参数描述

返回值：指定引脚的 `Pin` 对象。

2, `machine.Pin` 类的方法

①：使用给定参数重新初始化引脚。

其方法原型如下：

```
led.init(mode=-1, pull=-1, value=None)
```

此方法的参数描述如下：

参数	描述	
<code>mode</code>	输入输出模式	
	<code>Pin.IN</code>	输入模式
	<code>Pin.OUT</code>	输出模式

	Pin.OPEN_DRAIN	开漏模式
pull	如果指定的引脚连接了上拉下拉电阻	
	None	无上拉或者下拉电阻
	Pin.PULL_UP	使能上拉电阻
	Pin.PULL_DOWN	使能下拉电阻
value	引脚电平值。输出：0 为低电平，1 为高电平；输入：无须参数，返回电平	

表 8.1.2 led.init 方法的参数描述

返回值：None。

②：获取或设置管脚输出电平。

其方法原型如下：

```
led.value([x])
```

此方法的参数描述如下：

参数	描述
x	有参数：设置输出电平
	无参数：获取管脚电平

表 8.1.3 led.value 方法的参数描述

返回值：管脚电平。

③：设置管脚输出高电平。

其方法原型如下：

```
led.on()
```

返回值：无。

④：设置管脚输出低电平。

其方法原型如下：

```
led.off()
```

返回值：无。

除了上述方法之外，其余方法不能在 ESP32 芯片上使用。相关知识，请看 MicroPython 在线文档。在 MicroPython 中，系统延时需要用到 time/utime 模块。该模块下有三个 sleep 方法：

- time.sleep(seconds)：秒级的延时
- time.sleep_ms(ms)：毫秒的延时
- time.sleep_us(us)：微秒的延时

需要注意的是：调用这些方法之前，必须在前面导入 time/utime 模块，才能使用这些模块的方法。

8.2 硬件设计

1. 例程功能

本章实验功能简介：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。

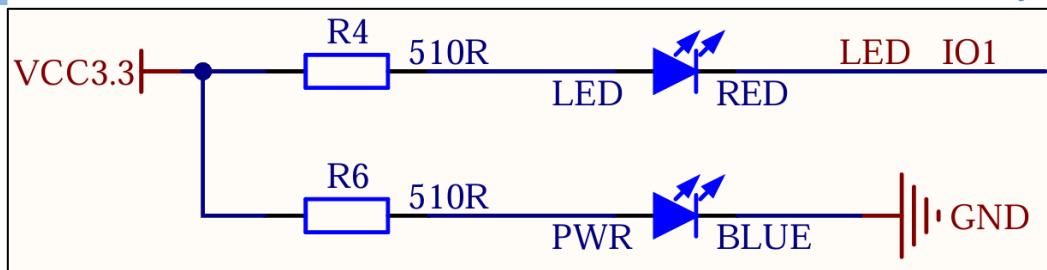


图 8.2.1 LED 与 ESP32-S3 模组连接原理图

从上图可知，若 IO1 输出低电平时，则 LED 亮起，反之，熄灭。

8.3 软件设计

8.3.1 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：

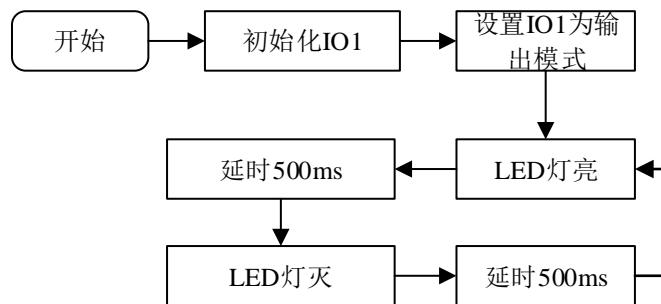


图 8.3.1.1 程序流程图

8.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。LED 实验 main.py 源码如下：

```

from machine import Pin
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1,Pin.OUT,value = 1)
    while True:

        led.value(0)                      # 设置 GPIO1 输出低电平
        time.sleep_ms(500)                 # 延时 500ms
        led.value(1)                      # 设置 GPIO1 输出高电平
        time.sleep_ms(500)                 # 延时 500ms

```

在这个示例中，作者首先导入了 machine 特定库的 Pin 类和 time 两个库。然后，作者使用 Pin()方法实例化了一个 Pin 对象，该对象与 ESP32-S3 物理引脚 1 相对应，并将其配置为输出模式且初始电平为高电平。

在 while 循环中，我们使用 led.value()方法来翻转 LED 的状态。首先，我们将引脚设置为低电平，从而使 LED 亮起，然后等待 0.5 秒。接着，我们将引脚设置为高电平，从而使 LED 熄灭，

并再次等待 0.5 秒。这个过程会不断重复，从而实现 LED 的闪烁效果。

注意：“`if __name__ == '__main__':`”是 Python 的一种常见模式，用于确保该代码块只在直接运行此脚本时执行，而不是在其他脚本中导入时执行。

在 Thonny 软件上，导入 `main.py` 文件至 MicroPython 设备中，有两种方法。第一种是在本地上存到 MicroPython 设备。另一种是新建文本，然后把正点原子提供的实验内容复制粘贴，并把新建文本命名为“`main.py`”保存至 MicroPython 设备。下面作者详细介绍这两种方法的使用流程。

第一种：本地上存

第一种方法很简单，只需把 Thonny 本地目录设置为正点原子例程目录，然后右键把 `01_led` 实验的 `main.py` 脚本上存到 MicroPython 设备当中，如下图所示。



图 8.3.2.1 main.py 脚本上存流程

上存完成后，按下键盘上的“F5”运行程序。

第二种：新建文本

在 Thonny 软件上，新建文档，然后复制上述的源代码到文档当中，如下图所示。



图 8.3.2.2 新建文件

接着，按下“Ctrl+S”保存至 MicroPython 设备当中，并命名为 `main.py` 文件，如下图所示。

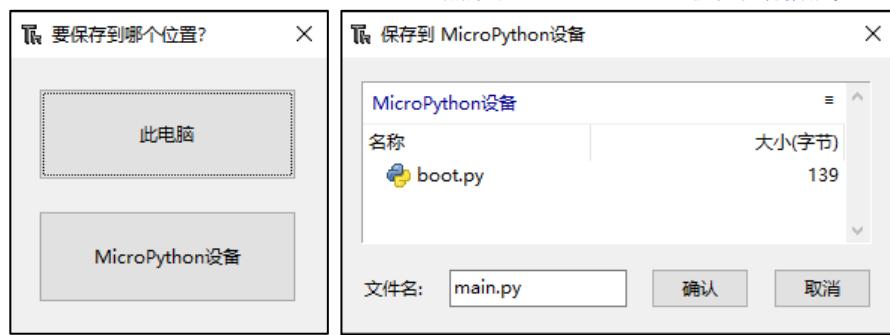


图 8.3.2.3 命名为 main.py，且保存至 MicroPython 设备当中
最后，按下“F5”运行代码。

8.4 下载验证

下载完之后，可以看到 LED 以每次 500ms 闪烁。

第九章 KEY 实验

上一章，我们介绍了 ESP32-S3 的 IO 口作为输出的使用。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。

本章分为如下几个小节：

- 9.1 独立按键简介
- 9.2 硬件设计
- 9.3 软件设计
- 9.4 下载验证

9.1 独立按键简介

`machine.Pin` 类操作方法相关内容，请参考章节七 LED 实验的 7.1 小节。

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：

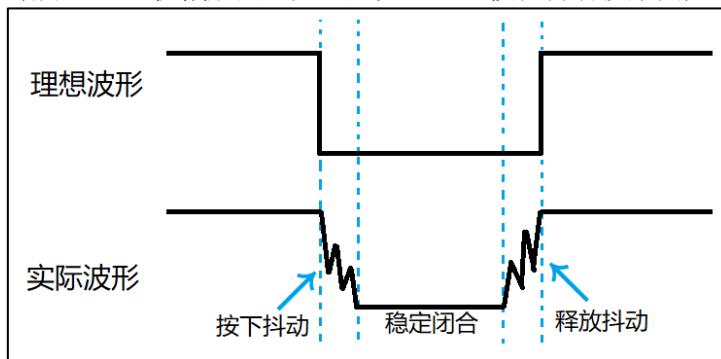


图 9.1.1 独立按键抖动波形图

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。

9.2 硬件设计

1. 例程功能

本章实验功能简介：通过开发板上的 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

独立按键硬件部分的原理图，如下图所示。

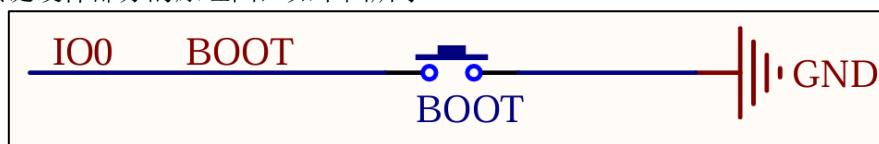


图 9.2.1 独立按键与 ESP32-S3 连接原理图

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。

9.3 软件设计

9.3.1 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：

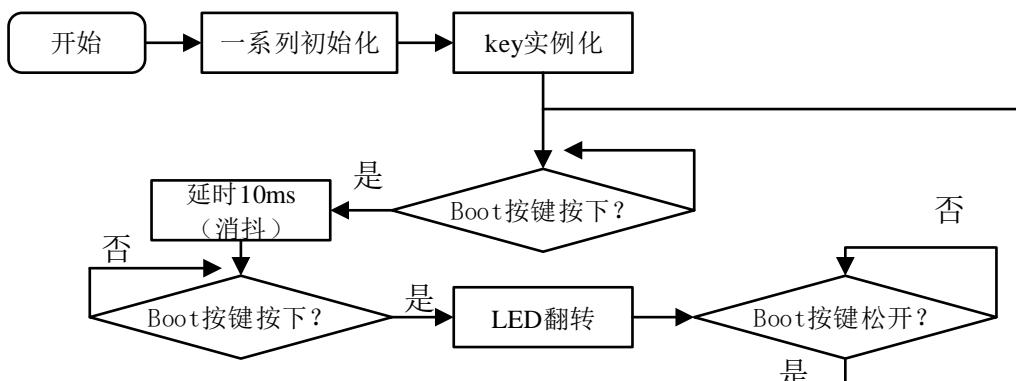


图 9.3.1.1 程序流程图

9.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。KEY 实验 main.py 源码如下：

```

from machine import Pin
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1, Pin.OUT, value = 1)          # 配置 led 引脚模式
    key = Pin(0, Pin.IN, Pin.PULL_UP)         # 配置 key 引脚模式及上拉

    while True:
        if key.value() == 0:                  # 判断 KEY 是否按下
  
```

```
time.sleep_ms(10)           # 该延时为按键消抖

if key.value() == 0:         # 再一次判断是否按下
    led_state = led.value()
    led.value(not led_state)

while not key.value():      # 检测按键是否松开
    pass
```

这段 MicroPython 代码通过按键控制 LED 灯的开关状态。当按键被按下时，程序会等待 10 毫秒消除抖动，然后再次确认按键状态，如果按键确实被按下，则获取当前 LED 状态并反转。然后程序进入等待按键松开的循环。

9.4 下载验证

下载完之后，通过 BOOT 按键来控制 LED 灯的开关状态。

第十章 EXIT 实验

在前面几章的学习中，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，实现我们在第九章所实现的功能。

本章分为以下几个小节：

- 10.1 外部中断简介
- 10.2 machine.Pin 类
- 10.3 硬件设计
- 10.4 软件设计
- 10.5 下载验证

10.1 外部中断简介

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，比如在某个时刻按下按键，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。

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 10.1.1 中断程序执行示意图

10.2 machine.Pin 类

1, machine.Pin 类的构造方法

machine.Pin 类操作方法相关内容，请参考章节七 LED 实验的 7.1 小节。

2, machine.Pin 类的方法

①：设置外部中断

其函数原型如下：

```
Pin.irq(handler=None, trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING))
```

此方法的参数描述如下：

参数	描述
handler	中断回调函数
trigger	触发中断，上升沿或者下降沿，可 OR

表 10.2.1.1 Pin.irq 方法的参数描述

返回值：一个回调对象。

10.3 硬件设计

1. 例程功能

本章实验功能简介：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

独立按键硬件部分的原理图，如下图所示。

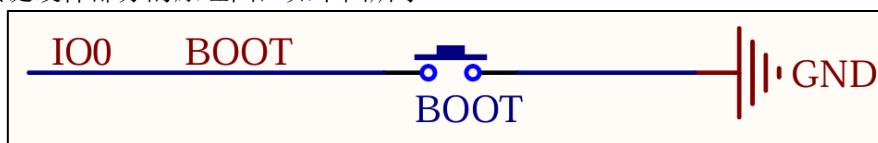


图 10.3.1 独立按键与 ESP32-S3 连接原理图

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。

10.4 软件设计

10.4.1 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：

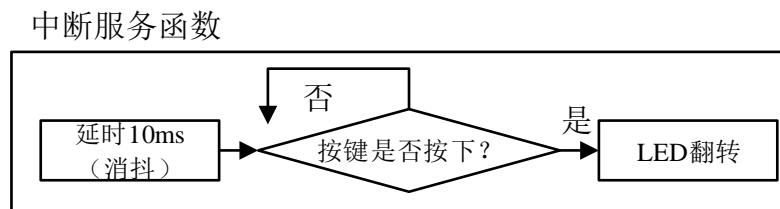
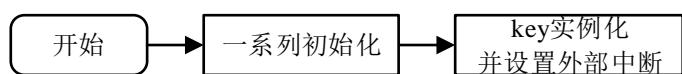


图 10.4.1.1 程序流程图

10.4.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。EXIT 实验 main.py 源码如下：

```
from machine import Pin
import time

# 定义全局变量
global led, key

"""
* @brief      按键中断服务函数
* @param      key:定时器句柄

```

```
* @retval      无
"""
def KEY_INT_IRQHandler(key):
    global led, key
    time.sleep_ms(10)                                # 按键消抖

    if key.value() == 0:
        global led_state
        led_state = led.value()
        led.value(not led_state)

"""
* @brief      程序入口
* @param      无
* @retval     无
"""
if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1,Pin.OUT,value = 1)
    # 配置 key 引脚模式及上下拉
    key = Pin(0,Pin.IN,Pin.PULL_UP)
    # 定义中断，下降沿触发
    key.irq(KEY_INT_IRQHandler,Pin.IRQ_FALLING)
    # 主循环，防止程序退出
    while True:
        pass
```

该示例主要用于通过按键来控制 LED 灯的状态。具体来说，当按键被按下时，LED 灯的状态会被切换（如果灯原本是亮的，则会熄灭；如果灯原本是熄灭的，则会亮起）。为了实现这个功能，程序首先初始化了 LED 灯和按键，并为按键配置了一个中断。当中断被触发（即按键被按下）时，程序会先等待 10 毫秒以消除按键抖动，然后检查按键的状态。如果按键状态为低（即按键被按下），程序会获取当前 LED 的状态并将其反转。

10.5 下载验证

下载完之后，通过 BOOT 按键来控制 LED 灯的开关状态。

第十一章 Timer 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。

本章分为以下几个小节：

- 11.1 Timer 简介
- 11.2 machine.Timer 类
- 11.3 硬件设计
- 11.4 软件设计
- 11.5 下载验证

11.1 Timer 简介

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： $10\text{ms} / 1\mu\text{s} = 10000$ ，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。

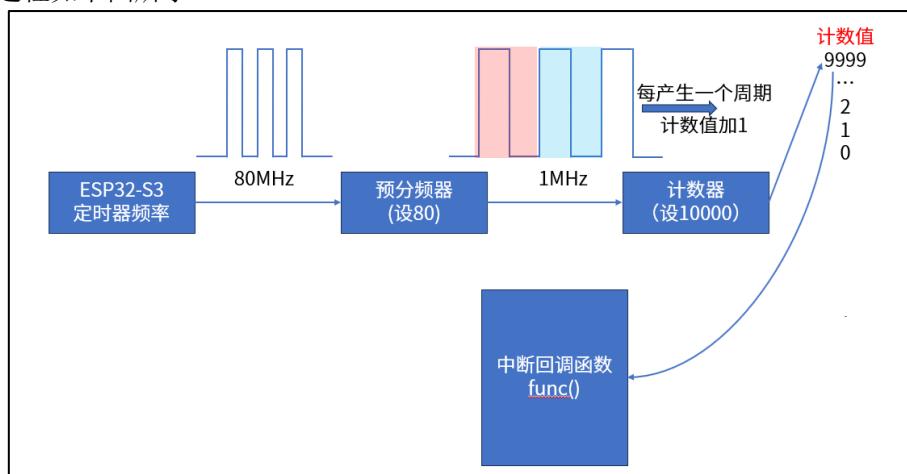


图 11.1.1 定时器定时配置过程

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 machine.Timer 类

`machine.Timer` 类是 `machine` 模块下的一个硬件类，用于 `Timer` 设备的配置和控制，提供对 `Timer` 设备的操作方法。`Timer`（硬件定时器）是一种用于处理周期性和定时性事件的设备，主要通过内部计数器模块对脉冲信号进行计数，实现周期性设备控制的功能。同时，`Timer` 硬件定时器可以自定义超时时间和超时回调函数，并且提供两种定时器模式：`ONE_SHOT` 和 `PERIOD`。打印 `Timer` 对象会打印出配置的信息。

1, `machine.Timer` 类的构造函数

`Timer` 的构造对象方法如下：

```
class machine.Timer(id, /, ...)
```

该构造函数的参数描述，如下表所示。

参数	描述
id	构造给定 id 的 Timer 对象

表 11.1.1 machine.Timer 构造函数参数描述

返回值：返回指定引脚的 Timer 对象。

2, machine.Timer 类的方法

①：对象初始化

其函数原型如下：

```
Timer.init(*, mode=Timer.PERIODIC, freq=-1, period=-1, callback=None)
```

该函数的参数描述，如下表所示。

参数	描述
mode	Timer 定时器模式
	ONE_SHOT（执行一次）
	PERIODIC（周期性执行）
freq	设置定时器频率。当给出 freq 和 period 参数时，freq 具有更高的优先级，而 period 被忽略
period	设置 Timer 定时器定时周期(ms)
callback	设置 Timer 定义器超时回调函数

表 11.1.2 UART.init 函数参数描述

②：关闭 Timer 设备

其函数原型如下：

```
Timer.deinit()
```

11.2 硬件设计

1. 例程功能

本章实验功能简介：程序启动后配置定时器的定时时间为 1 秒，定时到来时执行中断服务函数翻转 LED 状态。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) 独立按键
BOOT-IO0
- 3) Timer1

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

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：

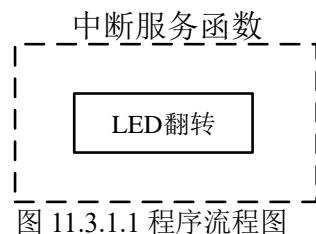
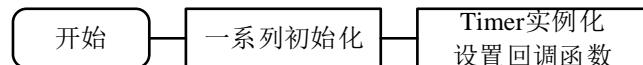


图 11.3.1.1 程序流程图

11.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。Timer 实验 main.py 源码如下：

```

from machine import Pin,Timer
import time

"""
* @brief      基本定时器 TIMEX 中断服务函数
* @param      tim: 定时器句柄
* @retval     无
"""

def BTMR_TIMEX_INT_IRQHandler(tim):
    led_state = led.value()
    led.value(not led_state)
"""

* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1,Pin.OUT,value = 1)
    # 开启定时器 1
    tim = Timer(1)
    # 配置定时器 1: 1000ms 中断、循环模式及中断回调函数 BTMR_TIMEX_INT_IRQHandler
    tim.init(period = 1000, mode = Timer.PERIODIC,
              callback = BTMR_TIMEX_INT_IRQHandler)

```

这示例代码的主要部分包括一个定时器和一个 LED 灯。

首先，通过 Pin 类初始化 LED 灯，设置其引脚为输出模式，并初始化为高电平（也就是 LED 灯初始为亮的状态）。

然后，通过 Timer 类开启一个定时器，设置其编号为 1。

接下来，配置这个定时器。设置其周期为 1000 毫秒（也就是 1 秒），模式为 Timer.PERIODIC，表示这是一个循环定时器，即每隔设定的周期就会触发一次中断。同时，设置中断回调函数为 BTMR_TIMEX_INT_IRQHandler，这意味着每当定时器触发中断时，都会执行这个函数。

最后，BTMR_TIMEX_INT_IRQHandler 函数的作用是在每次定时器中断时切换 LED 灯的状态。它首先读取当前 LED 灯的状态，然后反转这个状态，最后将新的状态写入 LED 灯。因此，每当定时器触发中断时，LED 灯的状态（亮或灭）就会被反转。

11.4 下载验证

下载代码完成后，ESP32-S3 最小系统板每隔 1000 毫秒触发一次中断，然后在中断服务函数中切换 LED 灯的状态。

第十二章 PWM 实验

在 ESP32-S3 上，PWM 功能主要由 machine 模块提供，通过设置 PWM 频率（每秒高电平脉冲数）和占空比（一个周期内高电平时间比例）实现对特定设备的控制。例如，使用 PWM 控制 LED 灯亮度时，需要先初始化 PWM 对象并设置频率和占空比。频率影响 PWM 信号扫描速度，占空比则控制高电平脉冲时间长度，从而影响 LED 灯亮度。

本章分为以下几个小节：

12.1 PWM 脉宽调制技术

12.2 machine.PWM 类

12.3 硬件设计

12.4 软件设计

12.5 下载验证

12.1 PWM 脉宽调制技术

PWM 是脉冲宽度调制的缩写，它是通过对一些列脉冲的宽度进行调制，等效出所需要的波形，对模拟信号电平进行数字编码，也就是说，通过调节占空比的变化调节信号、能量等编号。占空比就是指在一个周期内，信号处于高电平的时间占据整个信号周期的百分比，如方波的占空比为 50%。

在第八章实现的 LED 灯闪烁，500ms 亮与 500ms 灭循环，信号的波形图如下所示。

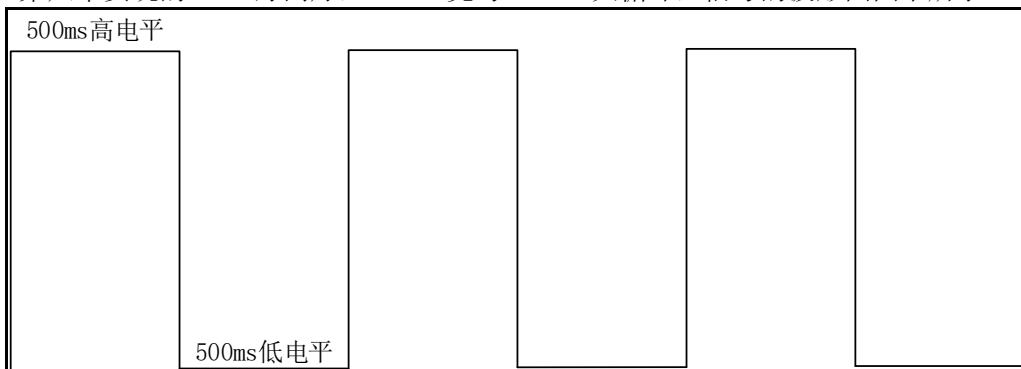


图 12.1.1 信号波形图

上图中，占空比为 50% 的方波。如果把延时时间缩短，时间越短 LED 灯闪烁的越快。当时间足够短的时候，从人眼的角度来看，LED 保持常亮，并不会感觉到 LED 闪烁。如果我们在一个时间足够短的周期内调节高低电平的时间比例，也就是控制高低电平的占空比，就可以达到控制 LED 灯亮度的目的了，这就是 PWM 控制。不同占空比效果如下图所示。

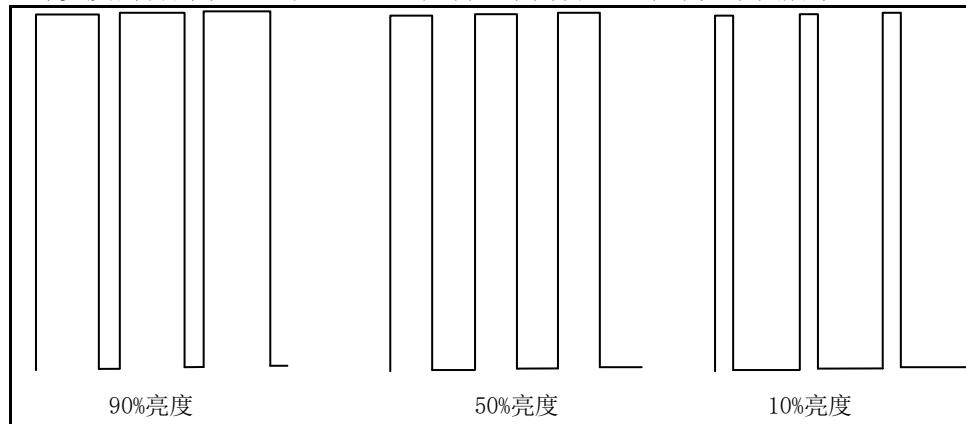


图 12.1.2 PWM 控制

PWM 占空比受到两个因素影响，频率和占空比。例如 PWM 的控制周期为 100ms，其中 25ms 被设置为高电平，75ms 被设置为低电平，则占空比就是 $25/100=25\%$ 。值得注意的是，在 MicroPython 中，ESP32-S3 的占空比（duty）并不是百分比取值，而是一个分辨率的形式，其取值范围为 0~1023。

12.2 machine.PWM 类

machine.PWM 类是 machine 模块下的一个硬件类，用于指定 PWM 设备的配置和控制，提供对 PWM 设备的操作方法。PWM 是一种对模拟信号电平进行数字编码的方式。PWM 设备可以通过调节有效电平在一个周期信号中的比例时间来操作设备。PWM 设备有两个重要的参数：频率（freq）和占空比（duty）。频率是指从一个上升沿（下降沿）到下一个上升沿（下降沿）的时间周期，单位为 Hz。占空比是有效电平（通常为电平）在一个周期内的时间比例。

1, machine.PWM 类的构造对象

在 MicroPython 中 PWM 的构造对象方法如下：

```
class machine.PWM(pin, freq, duty)
```

使用示例：pwm = machine.PWM(Pin(1), freq = 1000)

该构造函数的参数描述，如下表所示。

参数	描述
pin	Pin 对象
freq	设置 PWM 周期的频率（0~78.125KHz）
duty	占空比（0~1023）

表 12.2.1 machine.PWM 构造函数参数描述

返回值：PWM 对象。

2, machine.PWM 类的方法

①：PWM 对象初始化。

其方法原型如下：

```
pwm.init(freq, duty)
```

该方法的参数描述，如下表所示。

参数	描述
freq	设置 PWM 周期的频率（0~78.125KHz）
duty	占空比（0~1023）

表 12.2.2 UART.init 方法参数描述

②：关闭 PWM 设备。

其方法原型如下：

```
pwm.deinit()
```

③：设置频率。

其方法原型如下：

```
pwm.freq([value])
```

该方法的参数描述，如下表所示。

参数	描述
value	设置 PWM 周期的频率（0~78.125KHz）

表 12.2.2 PWM.freq 方法参数描述

④：设置占空比。

其方法原型如下：

```
pwm.duty ([value])
```

该方法的参数描述，如下表所示。

参数	描述
value	占空比（0~1023）

表 12.2.2 PWM.duty_u16 函数参数描述

12.3 硬件设计

1. 例程功能

本章实验功能简介：实现 LED 由暗变亮，再从亮变暗，依次循环。

2. 硬件资源

- 1) LED 灯
LED-IO1
- 2) PWM

3. 原理图

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

12.4 软件设计

12.4.1 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：

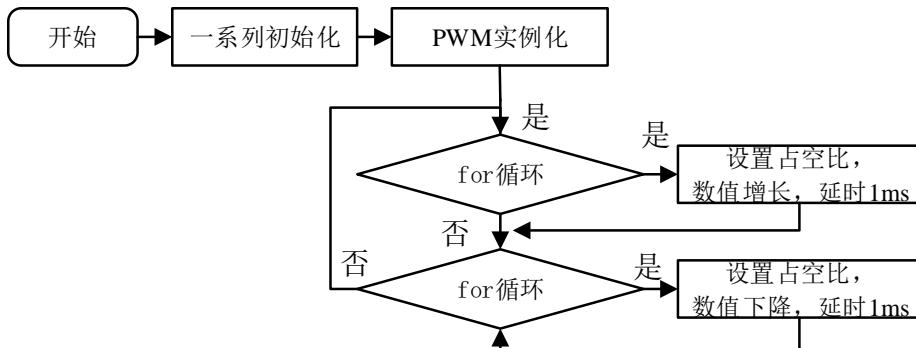


图 12.4.1.1 程序流程图

12.4.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。PWM 实验 main.py 源码如下：

```

from machine import Pin, PWM
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    pwm = PWM(Pin(1), freq = 1000)

    while True:
        # 渐亮
        for i in range(0,1024):
            pwm.duty(i)
            time.sleep_ms(1)
  
```

```
# 演示  
for i in range(1023,0,-1):  
    pwm.duty(i)  
    time.sleep_ms(1)
```

这示例代码使用 ESP32 的 PWM 功能，通过循环改变 PWM 占空比的方式，控制 LED 灯的亮度在 0-1023 之间变化，然后从 1023-0 之间变化，实现 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。

12.5 下载验证

下载代码完成后，ESP32-S3 最小系统板上 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。

第十三章 SPILCD 实验

在本章实验中，我们将通过编写 MicroPython 驱动程序来实现 SPILCD 显示。在开发板上，我们已经预留了 SPILCS 模块接口，因此需要准备一个 SPILCD 显示模块。我们将一起点亮 SPILCD，并实现字符的显示。

- 13.1 SPILCD 模块简介
- 13.2 SPILCD C 模块解析
- 13.3 硬件设计
- 13.4 软件设计
- 13.5 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。

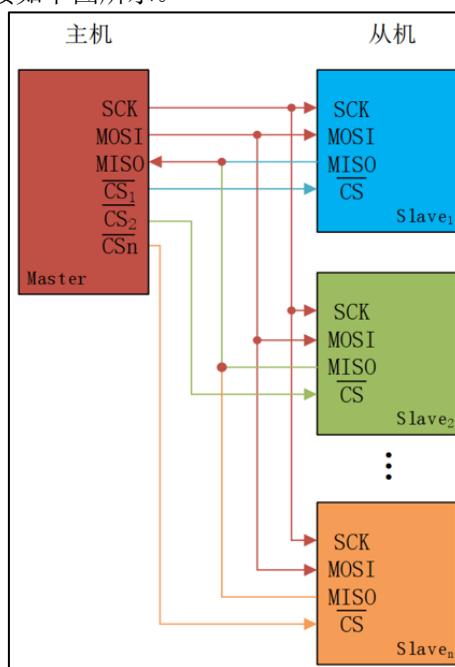


图 13.1.1.1 多从机 SPI 通信网络图

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。

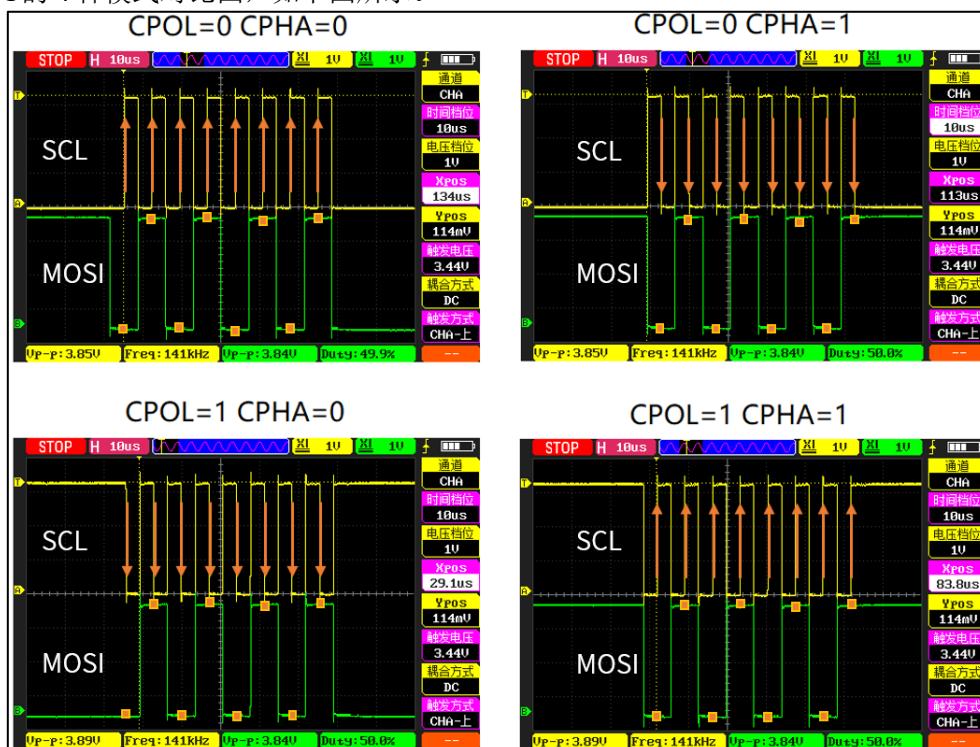


图 13.1.1.2 SPI 的 4 种模式对比图

- 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, 数据采样在下降沿, 数据发送在上升沿。
- 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, 数据采样在下降沿, 数据发送在上升沿。
- 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。

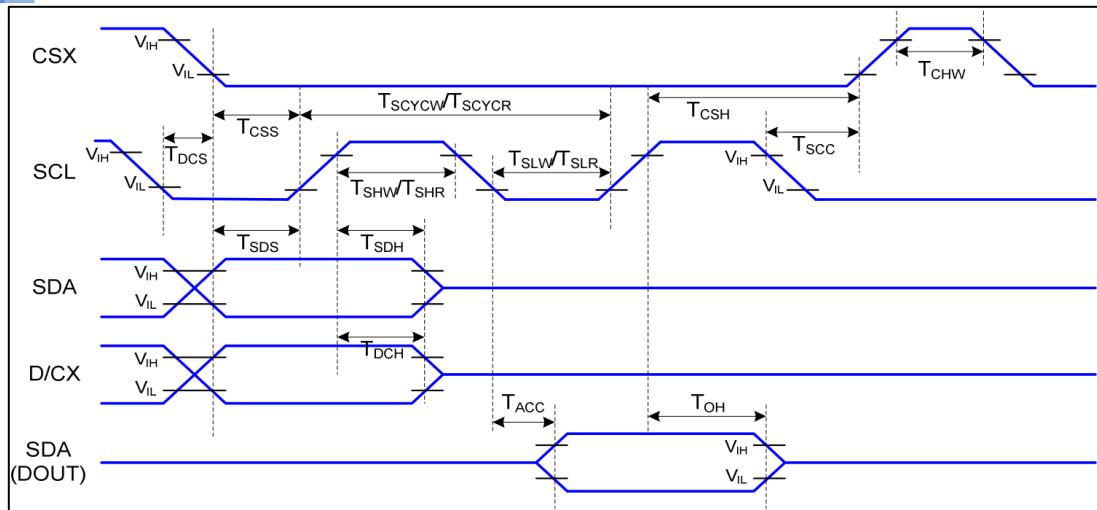


图 13.1.3.2 四线 SPI 接口时序图

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	
	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	-Read Command & Data Ram
	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 $132 \times 162 \times 18\text{-bits}$ ，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。

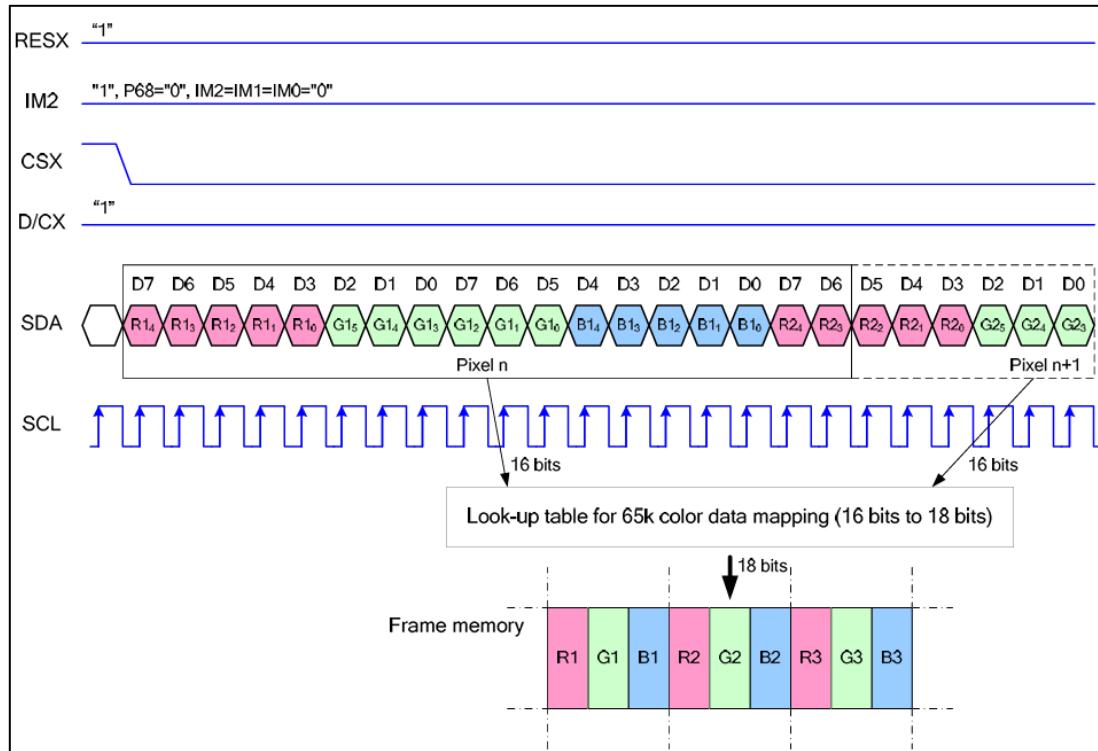


图 13.1.3.3 16 位色深模式 (RGB565) 传输颜色数据

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)													
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。

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。

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该区域的颜色数据。

13.2 SPILCD 驱动解析

13.2.1 MicroPython 驱动解析

作者将简要介绍正点原子 SPILCD MicroPython 模块驱动。

```
"""
* @brief      tft 类
* @param      无
* @retval     实例化对象
"""

class atk_tft(object):

    """
    * @brief      复位 TFT
    """


```

```
* @param      无
* @retval     无
"""
def resrt(self):

    self.rst(1)
    time.sleep_ms(200)
    self.rst(0)
    time.sleep_ms(200)
    self.rst(1)
    time.sleep_ms(200)
"""

* @brief      写命令
* @param      cmd_data:命令值
* @retval     无
"""
def write_cmd(self,cmd):

    self.dc(0)
    self.cs(0)
    self.spi.write(bytes([cmd]))
    self.cs(1)
"""

* @brief      写数据
* @param      data:数据
* @retval     无
"""
def write_data(self,data):

    self.dc(1)
    self.cs(0)
    self.spi.write(data)
    self.cs(1)
"""

* @brief      设置窗口大小
* @param      xstar : 左上角x坐标
* @param      ystar : 左上角y坐标
* @param      xend  : 右上角x坐标
* @param      yend  : 右上角y坐标
* @retval     无
"""
def set_window(self,xstar,ystar,xend,yend):

    if self.rotate == 1:
        self.window_data[0] = ((xstar + 1) >> 8)
        self.window_data[1] = (0xFF & (xstar + 1))
        self.window_data[2] = ((xend + 1) >> 8)
        self.window_data[3] = (0xFF & (xend + 1))
        self.write_cmd(self.setxcmd)
        self.write_data(self.window_data)

        self.window_data[0] = ((ystar + 26) >> 8)
        self.window_data[1] = (0xFF & (ystar + 26))
        self.window_data[2] = ((yend + 26) >> 8)
        self.window_data[3] = (0xFF & (yend + 26))
        self.write_cmd(self.setycmd)
        self.write_data(self.window_data)
    else:
        self.window_data[0] = (xstar + 26) >> 8
        self.window_data[1] = 0xFF & (xstar + 26)
        self.window_data[2] = (xend + 26) >> 8
        self.window_data[3] = 0xFF & (xend + 26)
        self.write_cmd(self.setxcmd)
        self.write_data(self.window_data)

```

```
    self.window_data[0] = (ystar + 1) >> 8
    self.window_data[1] = 0xFF & (ystar + 1)
    self.window_data[2] = (yend + 26) >> 8
    self.window_data[3] = 0xFF & (yend + 26)
    self.write_cmd(self.setycmd)
    self.write_data(self.window_data)
    self.write_cmd(self.wramcmd)
"""

* @brief      设置屏幕方向
* @param      tf_dir:0 为竖屏,1 为横屏
* @retval     无
"""

def set_scan_dir(self,tf_dir):
    regval = 0
    dirreg = 0
    temp = 0

    if tf_dir == 1:
        regval |= 0xA8

    else:
        regval |= 0xC8

    dirreg = 0x36
    data_reg = bytearray([regval])

    self.write_cmd(dirreg)
    self.write_data(data_reg)

    if (regval & 0x20) == 1:
        if self.width < self.height :
            temp = self.width
            self.width = self.height
            self.height = temp
        else:
            if self.width > self.height:
                temp = self.width
                self.width = self.height
                self.height = temp
        self.set_window(0,0,self.width,self.height)
"""

* @brief      垂直线
* @param      x,y      : 坐标
* @param      length   : 长度
* @param      color    : 颜色
* @retval     无
"""

def vline(self, x, y, length, color):
    self.fill_rect(x, y, 1, length, color)
"""

* @brief      水平线
* @param      x,y      : 坐标
* @param      length   : 长度
* @param      color    : 颜色
* @retval     无
"""

def hline(self, x, y, length, color):
    self.fill_rect(x, y, length, 1, color)
```

```

"""
* @brief      画点
* @param      x,y      : 坐标
* @param      color    : 颜色
* @retval     无
"""

def pixel(self, x, y, color):
    self.set_window(x, y, x, y)
    self.write_data(self._encode_pixel(color))
"""

* @brief      矩形
* @param      x,y      : 坐标
* @param      w,h      : 长宽
* @param      color    : 颜色
* @retval     无
"""

def rect(self, x, y, w, h, color):
    self.hline(x, y, w, color)
    self.vline(x, y, h, color)
    self.vline(x + w - 1, y, h, color)
    self.hline(x, y + h - 1, w, color)
"""

* @brief      将像素颜色编码为字节
* @param      color    : 颜色值
* @retval     无
"""

def _encode_pixel(self, color):
    """Encode a pixel color into bytes."""
    return struct.pack(_ENCODE_PIXEL, color)
"""

* @brief      填充矩形
* @param      x,y      : 坐标
* @param      w,h      : 长宽
* @param      color    : 颜色
* @retval     无
"""

def fill_rect(self, x, y, width, height, color):
    self.set_window(x, y, x + width - 1, y + height - 1)
    chunks, rest = divmod(width * height, _BUFFER_SIZE)
    pixel = self._encode_pixel(color)
    self.dc(1)
    if chunks:
        data = pixel * _BUFFER_SIZE
        for _ in range(chunks):
            self.write_data(data)
    if rest:
        self.write_data(pixel * rest)
"""

* @brief      清屏
* @param      color    : 颜色
* @retval     无
"""

def clear(self, color):
    self.fill_rect(0, 0, self.width, self.height, color)
"""

* @brief      绘画 bmp
* @param      x,y      : 坐标
* @param      w,h      : 长宽
* @param      buffer   : bmp 数据
* @retval     无
"""

def draw_bmp(self,x,y,w,h,buffer):
    if((x >= self.width) or (y >= self.height)):

```

```

        return
    if (x + w - 1) >= self.width:
        w = self.width - x
    if (y + h - 1) >= self.height:
        h = self.height - y
    self.set_window(x,y,x+w-1,y+h-1)
    self.dc(1)
    self.cs(0)
    self.write_data(buffer)      # write bytes on MOSI
    self.cs(1)
"""
* @brief      绘画一个字符
* @param      x,y      : 坐标
* @param      ch       : 字符
* @retval     无
"""
def p_char(self, x, y, ch):
    fp = (ord(ch)-0x20) * 5
    f = open('font5x7.fnt', 'rb')
    f.seek(fp)
    b = f.read(5)
    char_buf = bytearray(b)
    char_buf.append(0)

    # make 8x6 image
    char_image = bytearray()
    for bit in range(8):
        for c in range(6):
            if ((char_buf[c]>>bit) & 1)>0:
                char_image.append(self._color >> 8)
                char_image.append(self._color & 0xff)
            else:
                char_image.append(self._bground >> 8)
                char_image.append(self._bground & 0xff)
    self.draw_bmp(x,y,6,8,char_image)
"""
* @brief      绘画一个字符串
* @param      x,y      : 坐标
* @param      str      : 字符串
* @retval     无
"""
def p_string(self, x, y, str):
    for ch in (str):
        self.p_char(x, y, ch)
        x += 6
"""
* @brief      显示一张 BMP 图片
* @param      fname   : .bmp 文件名称
* @param      x,y      : 坐标
* @param      color   : 颜色
* @retval     无
"""
def bmp(self,fname, x, y, color = 0):

    f = open(fname, 'rb')
    b = bytearray(54)
    b = f.read(54)
    # header check
    if b[0] == 0x42 and b[1] == 0x4D:
        # is bitmap
        size = b[2] + (b[3]<<8) + (b[4]<<16) +(b[5]<<24)
        offset = b[10] + (b[11]<<8) + (b[12]<<16) +(b[13]<<24)
        width = b[18] + (b[19]<<8) + (b[20]<<16) +(b[21]<<24)

```

```

height = b[22] + (b[23]<<8) + (b[24]<<16) +(b[25]<<24)
color_planes = b[26] + (b[27]<<8)
bits_per_pixel = b[28] + (b[29]<<8)
compression = b[30] + (b[31]<<8) + (b[32]<<16) +(b[33]<<24)
image_size = b[34] + (b[35]<<8) + (b[36]<<16) +(b[37]<<24)

f.seek(offset)

row_bytes = int(bits_per_pixel/8) * width
# Add up to multiple of 4
if row_bytes % 4 > 0:
    row_bytes += 4 - row_bytes % 4

buffer = bytearray(row_bytes)
for row in range(height):
    # print(row)
    # read in a whole row
    buffer=f.read(row_bytes)
    d_buffer = bytearray(width*2)
    index = 0
    for index in range(width):
        y1 = (height-1) - row + y
        if color:
            b = buffer[index*3]
            g = buffer[index*3+1]
            r = buffer[index*3+2]
            c = TFTColor(r,g,b)
            d_buffer[index*2] = c >> 8
            d_buffer[index*2+1] = c & 0xff
        else:
            if buffer[index*3]!=0xff:
                self.pixel(x,y,1)
        if color:
            self.draw_bmp(x,y1,width,1,d_buffer)
    f.close()
"""

* @brief      TFT 初始化序列
* @param      无
* @retval     无
"""

def tft_init(self):

    self.write_cmd(0x11)
    time.sleep_ms(120)
    self.write_cmd(0x21)
    time.sleep_ms(120)
    data1 = bytearray([0x05,0x3A,0x3A])
    self.write_cmd(0xB1)
    self.write_data(data1)
    self.write_cmd(0xB2)
    self.write_data(data1)
    data2 = bytearray([0x05,0x3A,0x3A,0x05,0x3A,0x3A])
    self.write_cmd(0xB3)
    self.write_data(data2)
    self.write_cmd(0xB4)
    data_reg = bytearray([0x03])
    self.write_data(data_reg)
    data3 = bytearray([0x62,0x02,0x04])
    self.write_cmd(0xC0)
    self.write_data(data3)
    self.write_cmd(0xC1)
    data_reg[0] = 0xC0
    self.write_data(data_reg)
    data4 = bytearray([0x0D,0x00])
    self.write_cmd(0xC2)
    self.write_data(data4)

```

```
data5 = bytearray([0x8D, 0x6A])
self.write_cmd(0xC3)
self.write_data(data5)
data6 = bytearray([0x8D, 0xEE])
self.write_cmd(0xC4)
self.write_data(data6)
self.write_cmd(0xC5)
data_reg[0] = 0x0E
self.write_data(data_reg)
data7 = bytearray([0x10, 0x0E, 0x02, 0x03, 0x0E, 0x07, 0x02,
                  0x07, 0x0A, 0x12, 0x27, 0x37, 0x00, 0x0D, 0x0E, 0x10])
self.write_cmd(0xE0)
self.write_data(data7)
data8 = bytearray([0x10, 0x0E, 0x03, 0x03, 0x0F, 0x06, 0x02,
                  0x08, 0x0A, 0x13, 0x26, 0x36, 0x00, 0x0D, 0x0E, 0x10])
self.write_cmd(0xE1)
self.write_data(data8)
self.write_cmd(0x3A)
data_reg[0] = 0x05
self.write_data(data_reg)
self.write_cmd(0x36)
data_reg[0] = 0xA8
self.write_data(data_reg)
self.write_cmd(0x29)
time.sleep_ms(120)
"""
* @brief      构造方法
* @param      spi      : 句柄
* @param      dc       : 命令与数据管脚
* @param      rst     : 复位管脚
* @param      cs      : 片选管脚
* @param      bl      : 背光管脚
* @param      rotate   : 方向
* @retval     无
"""
def __init__(self, spi, dc, rst, cs, bl, rotate = 0):

    self.rotate = rotate    # 默认为竖屏
    self.window_data = bytearray(4)
    self.spi = spi
    self._offset = bytearray([0, 0])
    self._color = RED
    self._background = WHITE
    # 管脚初始化
    self.dc = machine.Pin(dc, machine.Pin.OUT)
    self.rst = machine.Pin(rst, machine.Pin.OUT)
    self.cs = machine.Pin(cs, machine.Pin.OUT)
    self.bl = machine.Pin(bl, machine.Pin.OUT)
    self.reset()

    if self.rotate == 0:    # 默认为竖屏

        self.height = 160
        self.width = 80
        self.wramcmd = 0x2C
        self.setxcmd = 0x2A
        self.setycmd = 0x2B
    else:

        self.height = 80
        self.width = 160
        self.wramcmd = 0x2C
        self.setxcmd = 0x2A
        self.setycmd = 0x2B
```

```

self.tft_init()
self.set_scan_dir(self.rotate)
self.bl(1)
self.clear(WHITE)

```

根据上述源代码可以看出，作者定义了 `tft` 类，在类中编写了多个类的方法，如 2D 图形绘画、BMP 图片绘画、设置窗口和屏幕初始化序列等方法。当我们实例化一个 `tft` 对象时，可调用类的方法实现 TFT 相关功能。

13.2.2 LCD 构造与类的方法

1, 实例化对象

实例化一个 LCD 对象：

```

class atk_tft.init(spi,dc,rst,cs,bl,rotate = 0)
使用示例：
import st7735
spi = SPI(2, baudrate=80000000, sck=Pin(12), mosi=Pin(11), miso=Pin(13))
lcd = st7735.atk_tft(spi,40,38,39,41,rotate = 1) #DC, Reset, CS, BL, rotate

```

该构造方法的参数描述，如下表所示。

参数	描述
spi	SPI 控制块
dc	数据/命令控制管脚
Rst	复位管脚
cs	片选管脚
bl	背光管脚
rotate	方向管脚。0：竖屏；1：横屏

表 13.2.2.1 atk_tft.init 构造函数参数描述

返回值：LCD 对象。

2, lcd 类的方法

①：复位 LCD。

其函数原型如下：

```
lcd.resrt()
```

②：设置屏幕方向。

其函数原型如下：

```
lcd.set_scan_dir(tf_dir)
```

tf_dir：0 为竖屏，1 为横屏。

③：垂直线。

其函数原型如下：

```
lcd.vline(x, y, length, color)
```

该函数的参数描述，如下表所示。

参数	描述
x	X 坐标
y	Y 坐标
length	长度
color	线的颜色

表 13.2.2.2 lcd.vline 函数参数描述

④：水平线。

其函数原型如下：

```
lcd.hline(x, y, length, color)
```

该函数的参数描述，如下表所示。

参数	描述
x	X 坐标
y	Y 坐标
length	长度

color	线的颜色
-------	------

表 13.2.2.3 lcd.hline 函数参数描述

⑤: LCD 画线。

其函数原型如下:

```
lcd.line(x1,y1,x2,y2,color)
```

该函数的参数描述, 如下表所示。

参数	描述
x1	左上角 X 坐标
y1	左上角 Y 坐标
x2	右上角 X 坐标
y2	右上角 Y 坐标
color	线条的颜色

表 16.2.2.4 lcd.line 函数参数描述

⑥: LCD 画点。

其函数原型如下:

```
lcd.pixel(x,y,color)
```

该函数的参数描述, 如下表所示。

参数	描述
x	左上角 X 坐标
y	左上角 Y 坐标
color	线条的颜色

表 13.2.2.5 lcd.pixel 函数参数描述

⑦: LCD 画矩形。

其函数原型如下:

```
lcd.rect(x,y,w,h,color)
```

该函数的参数描述, 如下表所示。

参数	描述
x	左上角 X 坐标
y	左上角 Y 坐标
w	宽度
h	高度
color	线条的颜色

表 13.2.2.6 lcd.circle 函数参数描述

⑧: LCD 清屏。

其函数原型如下:

```
lcd.clear(color)
```

该函数的参数描述, 如下表所示。

参数	描述
color	清屏颜色

表 13.2.2.7 lcd.clear 函数参数描述

⑨: LCD 绘画 BMP 图片。

其函数原型如下:

```
lcd.draw_bmp(x,y,w,h,buffer)
```

该函数的参数描述, 如下表所示。

参数	描述
x	X 坐标
y	Y 坐标
w	宽度
h	高度

buffer	图像数据
--------	------

表 13.2.2.7 lcd.draw_bmp 函数参数描述

⑩: LCD 显示字符串。

其函数原型如下:

```
lcd.p_string(x,y,str)
```

该函数的参数描述, 如下表所示。

参数	描述
x	X 坐标
y	Y 坐标
str	字符串

表 13.2.2.8 lcd.p_string 函数参数描述

13.3 硬件设计

1. 例程功能

本章实验功能简介: 按下复位之后, 就可以看到 SPILCD 模块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) 0.96 寸 LCD

3. 原理图

SPILCD 硬件部分的原理图, 如下图所示。

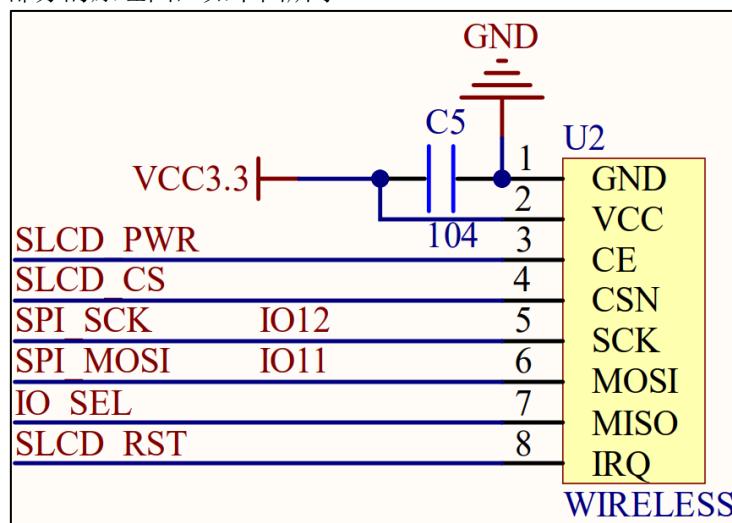


图 13.3.1 SPILCD 接口原理图

13.4 软件设计

13.3.1 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。

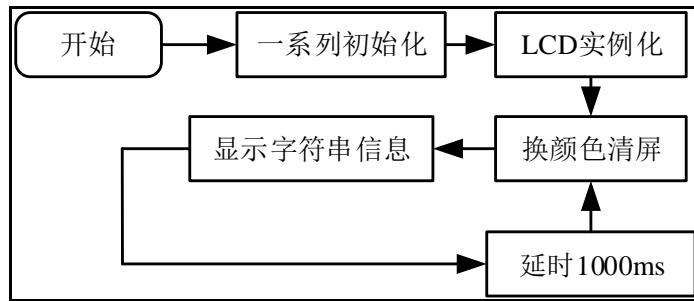


图 13.3.1.1 程序流程图

13.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。SPILCD 实验 main.py 源码如下：

```

from machine import Pin, SPI
import st7735
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    x = 0 # Initialize variable x to 0
    spi = SPI(2, baudrate=80000000, sck=Pin(12), mosi=Pin(11), miso=Pin(13))
    tft = st7735.atk_tft(spi, 40, 38, 39, 41, rotate = 1) #DC, Reset, CS, BL, rotate
    #tft.bmp('flower64x48.bmp',10,10,1)
    #time.sleep(3) # Sleep execution for 3 second

    while True: # Start an infinite loop

        # Create a dictionary for different LCD colors
        seasondict = {
            0: st7735.BLACK,
            1: st7735.BLUE,
            2: st7735.RED,
            3: st7735.GREEN,
            4: st7735.CYAN,
            5: st7735.YELLOW,
            6: st7735.WHITE}

        # Refresh the LCD display with the current color
        tft.clear(seasondict[x])
        tft.p_string(0,0,"ESP32-S3")
        tft.p_string(0,12,"ATOM@ALIENTEK")
        x += 1 # Increment x by 1

        if x == 7: # If x reaches 7, reset it to 0
            x = 0
        time.sleep(1) # Sleep execution for 1 second

```

在循环中，使用一个变量 x 来循环显示不同的颜色，并在屏幕上显示固定的文本。当变量 x 的值达到 7 时，它会被重置为 0，从而循环显示。

13.5 下载验证

下载代码后，LED 不停的闪烁，提示程序已经在运行了。同时可以看到 SPILCD 模块的显示实验信息，并且背景色不停切换，如下图所示。

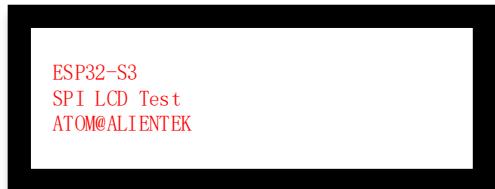


图 13.5.1 SPILCD 显示效果图

第十四章 RTC 实验

machine.RTC 类是 machine 模块下面的一个硬件类，它用于控制实时时钟（RTC）。实时时钟是一种计算机内部或外部的硬件设备，用于提供计算机系统的当前时间和日期。本章，我们将介绍 ESP32-S3 的内部实时时钟（RTC）。我们将使用 SPILCD 模块来显示日期和时间，实现一个简单的实时时钟。

本章分为以下几个小节：

14.1 machine.RTC 类

14.2 硬件设计

14.3 软件设计

14.4 下载验证

14.1 machine.RTC 类

machine.RTC 是一个硬件类，用于控制实时时钟（RTC）设备。RTC 设备是一种计算机内部或外部的硬件设备，它独立于其他系统操作，能够提供计算机系统的当前时间和日期。实时时钟的主要功能是提供精确的实时时间，可以用于产生年、月、日、时、分、秒等信息。

通过 Machine.RTC 类，用户可以在 MicroPython 环境下获取和设置 RTC 的当前时间，以及读取和写入 RTC 的寄存器。

1, machine.RTC 类的构造方法

RTC 的构造对象方法如下：

```
class machine.RTC()
```

使用示例： rtc = RTC()

返回值： RTC 对象。

2, machine.RTC 类的方法

①： 初始化 RTC 设备起始时间。

其方法原型如下：

```
rtc.init([datatimetuple])
```

该方法的参数描述，如下表所示。

参数	描述	
datatimetuple	时间元组 (year, month, day, wday, hour, minute, second, yday)	
	year	年份
	month	月份，范围 [1, 12]
	day	日期，范围 [1, 31]
	wday	星期，范围 [0, 6]，0 表示星期一
	hour	小时，范围 [0, 23]
	minute	分钟，范围[0, 59]
	second	秒，范围[0, 59]
	yday	从年份 1 月 1 日开始，范围 [0, 365]

表 14.1.2 RTC.init 方法参数描述

返回值： RTC 对象。

②： 关闭 RTC 设备。

其方法原型如下：

```
rtc.deinit()
```

返回值： 无。

③： 获取当前时间。

其方法原型如下：

```
rtc.datetime([datatimetuple])
```

返回值： 当前时间元组。

14.2 硬件设计

1. 例程功能

本章实验功能简介：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。

2. 硬件资源

- 1) 0.96 寸 LCD
- 2) RTC

3. 原理图

本章实验使用的 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

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：

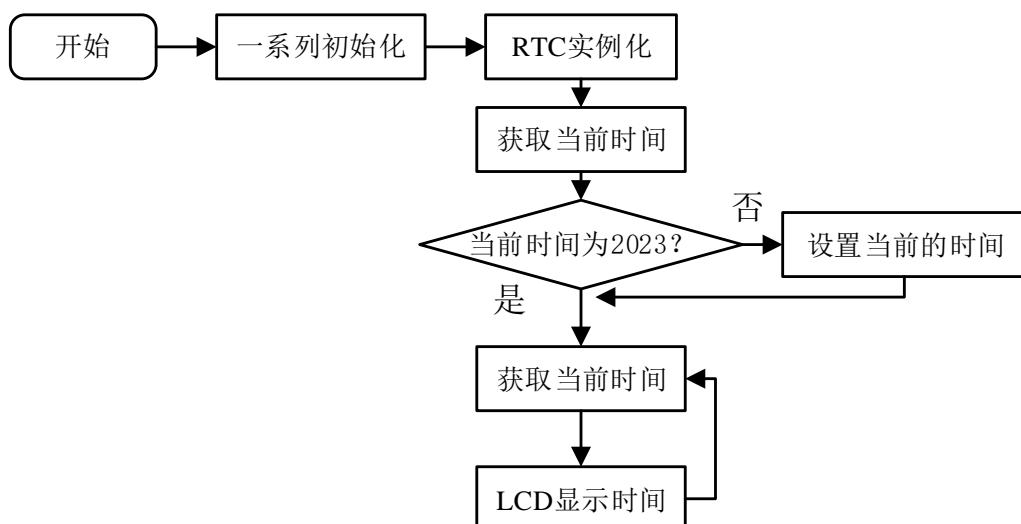


图 14.3.1.1 程序流程图

14.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。RTC 实验 main.py 源码如下：

```

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1,Pin.OUT,value = 1)
    spi = SPI(2, baudrate=80000000, sck=Pin(12), mosi=Pin(11), miso=Pin(13))
    tft = st7735.atk_tft(spi,40,38,39,41,rotate = 1) #DC, Reset, CS, BL, rotate
    time.sleep_ms(100)

```

```

# 显示实验信息
tft.p_string(0, 0, "ESP32-S3")
tft.p_string(0, 12, "ATOM@ALIENTEK")
tft.p_string(0, 24, "Specific:")
tft.p_string(0, 36, "Time:")
tft.p_string(0, 48, "Date:")
# 初始化 RTC
rtc = RTC()

if rtc.datetime()[0] != 2023:
    rtc.datetime((2023, 8, 15, 2, 0, 0, 0))

while True:

    data_time = rtc.datetime()
    tft.p_string(60, 24, str(data_time[0:3]))
    tft.p_string(30, 36, str(data_time[4:7]))
    tft.p_string(30, 48, str(int(data_time[3]) + 1))

```

这示例代码主要是在 ESP32-S3 微控制器上初始化各种硬件接口和设备，包括 LED、I2C、SPI、LCD 显示屏，以及 XL9555 芯片和 RTC（实时时钟）。

首先，代码初始化了 LED 灯并使其输出高电平。然后，它初始化了 I2C 接口，并使用这个接口初始化了 XL9555 芯片。接着，代码复位了 LCD 显示屏，然后初始化了 SPI 接口和 LCD 显示屏。在 LCD 显示屏上，它显示了一些实验信息。

此后，代码初始化了 RTC 芯片，如果获取的 RTC 时间不是 2023，那么就设置 RTC 时间为 2023 年 12 月 7 日。

最后，代码进入一个无限循环，在这个循环中，它获取 RTC 的时间，并将其显示在 LCD 显示屏上。这个显示包括年、月、日等信息。需要注意的是，获取的 RTC 时间是一个 Python 元组类型，所以需要使用索引来获取年、月、日等具体的时间信息。同时，这个代码还使用了 time.sleep_ms() 函数来进行延时操作，以确保操作的正确性和稳定性。

14.4 下载验证

将程序下载到开发板后，系统首先获取当前的时间（年、月、日、时、分和秒），然后在 SPILCD 显示屏上显示当前时间，如下图所示。

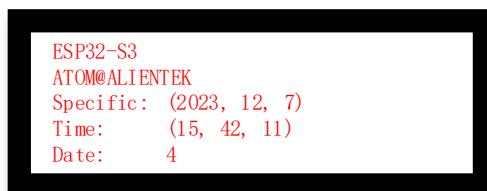


图 14.5.1 SPILCD 显示效果图

第十五章 看门狗实验

MicroPython 看门狗通常指的是在 MicroPython 环境中使用看门狗（Watchdog）来确保系统稳定运行的一种技术。看门狗（Watchdog）是一种硬件或软件机制，用于监视系统状态并在出现故障时重置系统。在 MicroPython 环境中，看门狗可以用来检测和解决系统故障，例如软件死循环、系统资源耗尽等问题。

本章分为以下几个小节：

15.1 WDT 类

15.2 硬件设计

15.3 程序设计

15.4 下载验证

15.1 WDT 类

在 MicroPython 中，可以使用 `machine.WDT` 类来创建一个看门狗对象，并设置其超时时间。当系统正常运行时，需要在看门狗允许的时间间隔内对计数器进行清零操作，以避免触发系统重启。如果系统出现故障或死机，看门狗计数器溢出，就会产生一个复位信号使系统重启。

1, machine.WDT 类的构造方法

```
class machine.WDT(id=0, timeout=5000)
```

该构造方法的参数描述，如下表所示。

参数	描述
id	看门狗编号
mode	设置看门狗超时时间

表 15.1.1 Pin 类构造方法的参数描述

返回值：看门狗对象。

2, machine.WDT 类的方法

①：喂狗

```
WDT.feed()
```

向看门狗提供输入以防止它重置系统。

15.2 硬件设计

1. 例程功能

本章实验功能简介：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。

2. 硬件资源

1) 独立按键

BOOT-IO0

2) WDT

3. 原理图

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

15.3 程序设计

15.3.1 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：

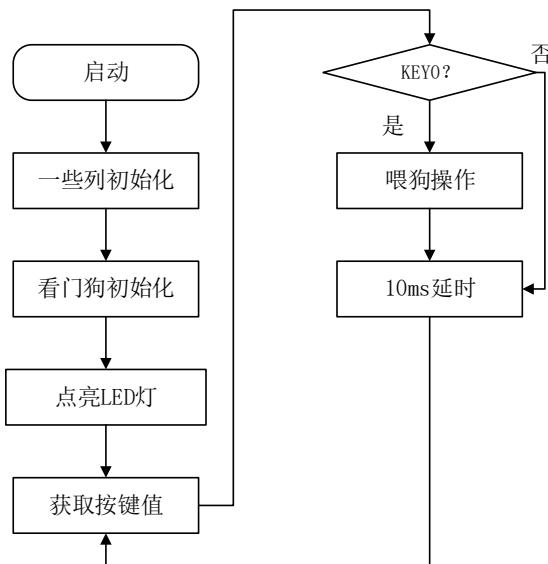


图 15.3.1.1 程序流程图

15.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。RTC 实验 main.py 源码如下：

```

from machine import Pin,I2C,WDT
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # 初始化 LED 并输出高电平
    led = Pin(1,Pin.OUT,value = 1)
    time.sleep_ms(1000)
    # 看门狗初始化，定时时间为 3s
    wdt = WDT(timeout=3000)
    led.value(0)

    while True:
        if key.value() == 0:                      # 判断 KEY 是否按下
            time.sleep_ms(10)                     # 该延时为按键消抖
            if key.value() == 0:                  # 再一次判断是否按下
                # 喂狗
                wdt.feed()

        while not key.value():                  # 检测按键是否松开
    
```

pass

```
time.sleep_ms(10)          # 延时 10ms
```

该示例代码非常简单，当我们在 3 秒钟内按下 KEY0 按键，表示喂狗成功，反次，系统复位。

15.4 下载验证

程序下载成功后，首先开启看门狗和点亮 LED 灯，当 3 秒钟内按下 KEY0 按键对看门狗进行喂狗，反次，系统复位操作。

第十六章 SD 卡实验

SD 卡是最常见的小型可移动存储介质之一，它有多种尺寸和物理外形。MMC 卡是类似的可移动存储设备，而 eMMC 设备是电气类似的存储设备，旨在嵌入其他系统。所有这三种形式共享一个通用协议来与它们的主机系统进行通信，并且高级支持对它们来说看起来都是一样的。因此，在 MicroPython 中，它们被抽象为一个名为 `machine.SDCard` 的对象。在本章中，我们将通过 `machine.SDCard` 对象来驱动这些存储设备。

本章分为以下几个小节：

16.1 SD 卡操作模块

16.2 硬件设计

16.3 软件设计

16.4 下载验证

16.1 SD 卡操作模块

在 MicroPython 中，`uos` 模块和 `machine.SDCard` 对象都与 SD 卡操作有关。`uos` 模块提供了一些用于操作文件系统的函数，例如 `listdir`（列出目录中的文件）、`mkdir`（创建目录）、`chdir`（切换当前工作目录）、`getcwd`（获取当前工作目录）、`remove`（删除文件）、`rename`（重命名文件或文件夹）、`stat`（获取文件或文件夹的状态信息）、`chmod`（改变文件或文件夹的权限）、`utime`（修改文件或文件夹的访问时间和修改时间）以及 `system`（执行系统命令）。

而 `machine.SDCard` 对象是 MicroPython 中用于表示 SD 卡的抽象对象。它提供了一组统一的 API 来与 SD 卡进行通信，包括初始化 SD 卡、获取 SD 卡信息、读取和写入块等操作。在使用 MicroPython 进行 SD 卡操作时，通常会结合使用 `uos` 模块和 `machine.SDCard` 对象。

现在，作者将对这些模块的构造函数和对象的使用方法进行讲解。

16.1.1 uos 模块

MicroPython 的内置模块 `uos` 主要提供文件系统操作服务。该模块实现了 CPython 模块的一个子集，同时具有一些特殊的特点和限制。以下是其主要特点和应用场景，以及需要注意的事项。

一、主要特点：

1，提供基本的“操作系统”服务，如获取系统信息、生成随机数、更改或获取当前目录、列出或创建或删除或重命名文件或目录等。

2，提供了文件系统访问和挂载的功能，例如在虚拟文件系统中挂载多个“真实”文件系统，或者使用不同类型的文件系统格式，如 FAT、littlefs 等。

3，提供终端重定向和复制的功能，例如在给定的类似流对象上复制或切换 MicroPython 终端（REPL）。

4，`uos` 模块使用的文件系统操作语法是 CPython `os` 模块的一个子集，也是 POSIX 标准文件系统操作的一个子集。支持的函数和方法有：`uname`、`urandom`、`chdir`、`getcwd`、`ilistdir`、`listdir`、`mkdir`、`remove`、`rmdir`、`rename`、`stat`、`statvfs`、`sync`、`mount`、`umount`、`dupterm` 等。不支持的有：`chown`、`chmod`、`link`、`symlink` 等。

二、应用场景：

1，对文件系统进行管理和操作，例如创建或删除或修改文件或目录，或者获取文件或目录的属性或状态。

2，对不同类型的存储设备进行访问和挂载，例如使用 SD 卡或 SPI 闪存等外部存储设备扩展内部存储空间，或者使用不同的文件系统格式适应不同的性能和兼容性需求。

3，对终端进行重定向和复制，例如在不同的通信接口上使用 MicroPython 终端（REPL），或者在多个终端上同时输出或输入数据。

三、需注意事项：

1, uos 模块的功能和性能可能因不同的端口而异, 因此在开发可移植的 MicroPython 应用程序时, 应该尽量避免依赖特定的文件系统操作语法或结果。

2, uos 模块在编译和执行文件系统操作时可能会消耗较多的内存和时间, 因此在处理大量或复杂的文件或目录时, 应该注意优化代码和资源管理。

3, uos 模块在编译和执行文件系统操作时可能会遇到无效或不合法的路径、参数、数据等, 或者文件错误、权限错误、设备错误等异常情况。这些情况会引发 OSError 或 ValueError 异常, 并给出错误信息。应该使用 try-except 语句来捕获并处理这些异常。

下面我们打开 Thonny 软件, 在 Shell 交互窗口下使用 dir 命令获取 uos 模块提供的函数, 如下所示:

```
>>> import machine, uos
>>> dir(uos)
['__class__', '__name__', 'remove', 'VfsFat', 'VfsLfs2', '__dict__', 'chdir',
'dupterm', 'dupterm_notify', 'getcwd', 'ilistdir', 'listdir', 'mkdir', 'mount',
'rename', 'rmdir', 'stat', 'statvfs', 'sync', 'umount', 'uname', 'unlink',
'urandom']
```

可以看到。MicroPython 的 uos 模块提供了十几种方法(函数), 这些方法足够我们去操作 SD 卡了。

16.1.2 machine.SDCard 对象

machine.SDCard 是一个用于表示 SD 卡的抽象对象, 它提供了一组统一的 API 来访问和操作 SD 卡。通过使用 machine.SDCard 对象, 开发人员可以轻松地读取和写入 SD 卡上块的地址、获取 SD 卡的容量和可用空间等操作。下面讲解的是 SDCard 对象的构造函数与方法。

一、SDCard 构造函数

SDCard 的构造对象方法如下:

```
class machine.SDCard(slot=1, width=1, cd=None, wp=None, sck=None,
                      miso=None, mosi=None, cs=None, freq=20000000)
```

使用示例: sd = machine.SDCard(slot=2, width=8, sck=12, miso=13, mosi=11, cs=2)

该构造方法的参数描述, 如下表所示。

参数	描述
slot	slot 选择要使用的可用接口
width	width 选择 SD/MMC 接口的总线宽度
cd	cd 可用于指定卡检测引脚
wp	wp 可用于指定写保护引脚
sck	sck 可用于指定 SPI 时钟引脚
miso	miso 可用于指定 SPI miso 引脚
mosi	mosi 可用于指定 SPI mosi 引脚
cs	cs 可用于指定 SPI 片选引脚
freq	freq 以 Hz 为单位选择 SD/MMC 接口频率

表 16.1.1 machine.SDCard 构造方法参数描述

返回值: SDCard 对象。

上表的 slot 参数选择接口如下表所示。

Slot	0	1	2	3
Signal	Pin (默认值)	Pin (默认值)	Pin (默认值)	Pin (默认值)
sck	6	14	18	14
cmd	11	15		
cs			5	15
miso			19	12
mosi			23	13
D0	7	2		

D1	8	4		
D2	9	12		
D3	10	13		
D4	16			
D5	17			
D6	5			
D7	18			

表 16.1.2 Slot 端号分配

从上表可知，如果我们使用 SPI 驱动 SD 卡，则选择 2 与 3 号端口。

注意：正点原子 ESP32-S3 最小系统板的 SD 卡使用的是 SPI 协议进行通信，所以 SD 卡构造函数中，我们仅使用上表绿色的参数。

二、SDCard 类的方法

打开 Thonny 软件，在 Shell 交互窗口下使用 dir 命令获取 SDCard 类的方法，如下所示：

```
>>> import machine
>>> dir(machine.SDCard)
['__class__', '__name__', '__bases__', '__del__', '__dict__', 'deinit', 'info',
'ioctl', 'readblocks', 'writeblocks']
>>>
```

可以看到。通过使用 machine.SDCard 对象，开发人员可以轻松地读取和写入 SD 卡上块的地址、获取 SD 卡的容量和可用空间等操作。

16.2 硬件设计

1. 例程功能

本章实验功能简介：系统打开 SD 卡根目录下的 test.txt 文件，然后在此文件下写入“Hello ALIENTEK”字符串数据，写入完成后读取此文件的内容，并输出至 SPILCD 显示屏上。

2. 硬件资源

1) 0.96 寸 LCD

2) SD

CS-IO2

SCK-IO12

MOSI-IO11

MISO-IO13

3. 原理图

SD 接口与 ESP32-S3 的连接关系，如下图所示：

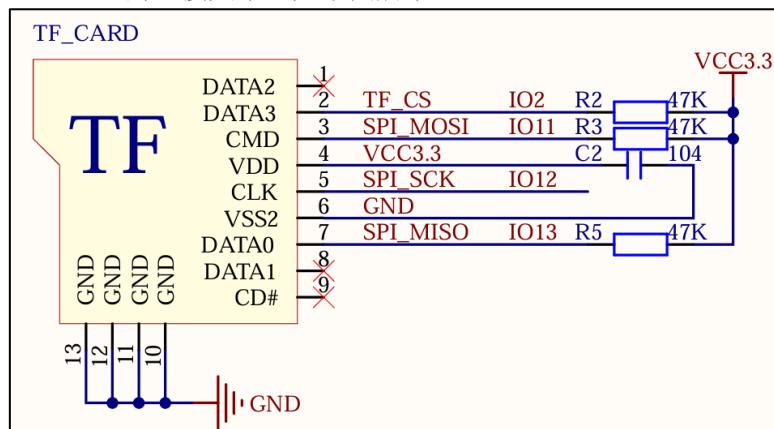


图 16.2.1 SD 接口与 ESP32-S3 的连接电路图

16.3 软件设计

16.3.1 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。

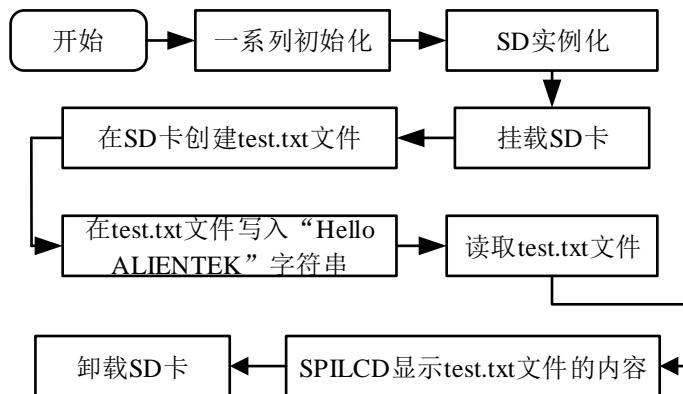


图 16.3.1.1 程序流程图

16.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。SD 卡实验 main.py 源码如下：

```

import machine, uos
import time

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    # Slot 2 uses pins sck = 12, cs = 2, miso = 13, mosi = 11
    sd = machine.SDCard(slot=2, width=8, sck=12, miso = 13, mosi=11, cs=2)
    # 重新当前系统文件目录
    print('挂载 SD 前的系统目录:{}\n'.format(uos.listdir()))
    # 使用 uos.VfsFat 类创建一个 FAT 文件系统对象
    vfs = uos.VfsFat(sd)
    # 挂载到 SD/sd
    uos.mount(vfs, '/sd')
    # 重新查询系统文件目录
    print('挂载 SD 后的系统目录:{}\n'.format(uos.listdir()))
    # 打开 test.txt 文件，如果 SD 卡目录没有会重新创建 test.txt 文件
    with open("/sd/test.txt", "w") as f:
        for i in range(1, 50):
            # 从这个文件写数据
            f.write(str(i)+"\n")
    # 从 sd 卡目录下读取 test.txt 文件内容
    with open("/sd/test.txt", "r") as f:
        # 打印读取的内容
        print(f.read())
    # 卸载 SD 卡
    uos.umount('/sd')
  
```

首先，我们需要实例化 SD 卡对象，并配置 SPI 协议通信和 SPI 管脚。然后，使用 uos.VfsFat

类创建一个 FAT 文件系统对象，并调用 `uos.mount` 方法挂载 SD 卡。接着，我们需要检查 SD 卡的挂载是否成功。如果挂载失败，系统目录将不显示 SD 卡目录。最后，我们需要打开 SD 卡根目录下的 `test.txt` 文件，在此文件下写入 1 到 50 的数字，写入完成后，读取此文件的内容，并在串口上输出读取的内容。

在使用这种方法驱动 SD 卡的情况下，可能会引发一个问题，导致 SPI LCD 显示屏无法正常显示。问题的根源在于正点原子 ESP32-S3 最小系统板的 SD 卡与 SPI LCD 共享一个 SPI 接口。在这种情况下，MicroPython 提供的 `machine_sdcard.c` 驱动并没有返回 SPI 控制块，这使得 LCD 驱动无法获取到 SPI 控制块以调用 SPI 收发函数发送相关的数据和命令。

因此，为了实现 SD 卡与 SPI LCD 的兼容性，作者引用了别人的一个用于驱动 SD 卡 Python 脚本（SPI 接口）。通过该脚本，可以在实例化 SPI 的情况下驱动 SD 卡和 SPI LCD 显示屏。下面是 `sdcard.py` 脚本的代码示例：

```
from micropython import const
import time

_CMD_TIMEOUT = const(100)

_R1_IDLE_STATE = const(1 << 0)
# R1_ERASE_RESET = const(1 << 1)
_R1_ILLEGAL_COMMAND = const(1 << 2)
# R1_COM_CRC_ERROR = const(1 << 3)
# R1_ERASE_SEQUENCE_ERROR = const(1 << 4)
# R1_ADDRESS_ERROR = const(1 << 5)
# R1_PARAMETER_ERROR = const(1 << 6)
_TOKEN_CMD25 = const(0xFC)
_TOKEN_STOP_TRAN = const(0xFD)
_TOKEN_DATA = const(0xFE)

class SDCard:
    def __init__(self, spi, cs, baudrate=20000000):
        self.spi = spi
        self.cs = cs

        self.cmdbuf = bytearray(6)
        self.dummybuf = bytearray(512)
        self.tokenbuf = bytearray(1)
        for i in range(512):
            self.dummybuf[i] = 0xFF
        self.dummybuf_memoryview = memoryview(self.dummybuf)

        # initialise the card
        self.init_card(baudrate)

    def init_spi(self, baudrate):
        try:
            master = self.spi.MASTER
        except AttributeError:
            # on ESP8266
            self.spi.init(baudrate=baudrate, phase=0, polarity=0)
        else:
            # on pyboard
            self.spi.init(master, baudrate=baudrate, phase=0, polarity=0)

    def init_card(self, baudrate):
        # init CS pin
        self.cs.init(self.cs.OUT, value=1)

        # init SPI bus; use low data rate for initialisation
        self.init_spi(100000)

        # clock card at least 100 cycles with cs high
```

```

for i in range(16):
    self.spi.write(b"\xff")

# CMD0: init card; should return _R1_IDLE_STATE (allow 5 attempts)
for _ in range(5):
    if self.cmd(0, 0, 0x95) == _R1_IDLE_STATE:
        break
    else:
        raise OSError("no SD card")

# CMD8: determine card version
r = self.cmd(8, 0x01AA, 0x87, 4)
if r == _R1_IDLE_STATE:
    self.init_card_v2()
elif r == (_R1_IDLE_STATE | _R1_ILLEGAL_COMMAND):
    self.init_card_v1()
else:
    raise OSError("couldn't determine SD card version")

# get the number of sectors
# CMD9: response R2 (R1 byte + 16-byte block read)
if self.cmd(9, 0, 0, 0, False) != 0:
    raise OSError("no response from SD card")
csd = bytearray(16)
self.readinto(csd)
if csd[0] & 0xC0 == 0x40: # CSD version 2.0
    self.sectors = ((csd[8] << 8 | csd[9]) + 1) * 1024
elif csd[0] & 0x00 == 0x00: # CSD version 1.0 (old, <=2GB)
    c_size = (csd[6] & 0b11) << 10 | csd[7] << 2 | csd[8] >> 6
    c_size_mult = (csd[9] & 0b11) << 1 | csd[10] >> 7
    read_bl_len = csd[5] & 0b1111
    capacity = (c_size + 1) * (2 ** (c_size_mult + 2))
    * (2**read_bl_len)
    self.sectors = capacity // 512
else:
    raise OSError("SD card CSD format not supported")
# print('sectors', self.sectors)

# CMD16: set block length to 512 bytes
if self.cmd(16, 512, 0) != 0:
    raise OSError("can't set 512 block size")

# set to high data rate now that it's initialised
self.init_spi(baudrate)

def init_card_v1(self):
    for i in range(_CMD_TIMEOUT):
        self.cmd(55, 0, 0)
        if self.cmd(41, 0, 0) == 0:
            # SDSC card, uses byte addressing in read/write/erase commands
            self.cdv = 512
            # print("[SDCard] v1 card")
            return
    raise OSError("timeout waiting for v1 card")

def init_card_v2(self):
    for i in range(_CMD_TIMEOUT):
        time.sleep_ms(50)
        self.cmd(58, 0, 0, 4)
        self.cmd(55, 0, 0)
        if self.cmd(41, 0x40000000, 0) == 0:
            # 4-byte response, negative means keep the first byte
            self.cmd(58, 0, 0, -4)
            # get first byte of response, which is OCR
            ocr = self.tokenbuf[0]
            if not ocr & 0x40:
                self.cdv = 512

```

```
        else:
            self.cdv = 1
            # print("[SDCard] v2 card")
            return
    raise OSError("timeout waiting for v2 card")

def cmd(self, cmd, arg, crc, final=0, release=True, skip1=False):
    self.cs(0)

    # create and send the command
    buf = self.cmdbuf
    buf[0] = 0x40 | cmd
    buf[1] = arg >> 24
    buf[2] = arg >> 16
    buf[3] = arg >> 8
    buf[4] = arg
    buf[5] = crc
    self.spi.write(buf)

    if skip1:
        self.spi.readinto(self.tokenbuf, 0xFF)

    # wait for the response (response[7] == 0)
    for i in range(_CMD_TIMEOUT):
        self.spi.readinto(self.tokenbuf, 0xFF)
        response = self.tokenbuf[0]
        if not (response & 0x80):
            if final < 0:
                self.spi.readinto(self.tokenbuf, 0xFF)
                final = -1 - final
            for j in range(final):
                self.spi.write(b"\xff")
            if release:
                self.cs(1)
                self.spi.write(b"\xff")
    return response

    # timeout
    self.cs(1)
    self.spi.write(b"\xff")
    return -1

def readinto(self, buf):
    self.cs(0)

    # read until start byte (0xff)
    for i in range(_CMD_TIMEOUT):
        self.spi.readinto(self.tokenbuf, 0xFF)
        if self.tokenbuf[0] == _TOKEN_DATA:
            break
        time.sleep_ms(1)
    else:
        self.cs(1)
        raise OSError("timeout waiting for response")

    # read data
    mv = self.dummybuf_memoryview
    if len(buf) != len(mv):
        mv = mv[: len(buf)]
    self.spi.write_readinto(mv, buf)

    # read checksum
    self.spi.write(b"\xff")
    self.spi.write(b"\xff")

    self.cs(1)
    self.spi.write(b"\xff")
```

```
def write(self, token, buf):
    self.cs(0)

    # send: start of block, data, checksum
    self.spi.read(1, token)
    self.spi.write(buf)
    self.spi.write(b"\xff")
    self.spi.write(b"\xff")

    # check the response
    if (self.spi.read(1, 0xFF)[0] & 0x1F) != 0x05:
        self.cs(1)
        self.spi.write(b"\xff")
        return

    # wait for write to finish
    while self.spi.read(1, 0xFF)[0] == 0:
        pass

    self.cs(1)
    self.spi.write(b"\xff")

def write_token(self, token):
    self.cs(0)
    self.spi.read(1, token)
    self.spi.write(b"\xff")
    # wait for write to finish
    while self.spi.read(1, 0xFF)[0] == 0x00:
        pass

    self.cs(1)
    self.spi.write(b"\xff")

def readblocks(self, block_num, buf):
    nbblocks = len(buf) // 512
    assert nbblocks and not len(buf) % 512, "Buffer length is invalid"
    if nbblocks == 1:
        # CMD17: set read address for single block
        if self.cmd(17, block_num * self.cdv, 0, release=False) != 0:
            # release the card
            self.cs(1)
            raise OSError(5) # EIO
        # receive the data and release card
        self.readinto(buf)
    else:
        # CMD18: set read address for multiple blocks
        if self.cmd(18, block_num * self.cdv, 0, release=False) != 0:
            # release the card
            self.cs(1)
            raise OSError(5) # EIO
        offset = 0
        mv = memoryview(buf)
        while nbblocks:
            # receive the data and release card
            self.readinto(mv[offset : offset + 512])
            offset += 512
            nbblocks -= 1
        if self.cmd(12, 0, 0xFF, skip1=True):
            raise OSError(5) # EIO

def writeblocks(self, block_num, buf):
    nbblocks, err = divmod(len(buf), 512)
    assert nbblocks and not err, "Buffer length is invalid"
    if nbblocks == 1:
        # CMD24: set write address for single block
        if self.cmd(24, block_num * self.cdv, 0) != 0:
```

```

        raise OSError(5) # EIO

    # send the data
    self.write(_TOKEN_DATA, buf)
else:
    # CMD25: set write address for first block
    if self.cmd(25, block_num * self.cdv, 0) != 0:
        raise OSError(5) # EIO
    # send the data
    offset = 0
    mv = memoryview(buf)
    while nbblocks:
        self.write(_TOKEN_CMD25, mv[offset : offset + 512])
        offset += 512
        nbblocks -= 1
    self.write_token(_TOKEN_STOP_TRAN)

def ioctl(self, op, arg):
    if op == 4: # get number of blocks
        return self.sectors
    if op == 5: # get block size in bytes
        return 512

```

可以看到，这个脚本通过使用 SD 命令来实现对 SD 卡的读写操作。在脚本的构造函数中，需要传入三个参数来实例化 SD 卡对象，分别是 SPI 控制块、CS 片选管脚以及 SPI 的速率。这些参数在实例化 SD 卡对象时是必需的。此外，值得注意的是，驱动 SD 卡的速率一般不能超过 24M，相关内容可以参考 SD 卡数据手册。

首先，作者在 Thonny 软件中新建了一个文本文件。然后，在此文本文件中粘贴了上述的 SD 卡驱动代码，并将其命名为 sdcard.py 文件。接着，将这个文件保存到了 MicroPython 设备中，如下图所示：



图 16.3.2.1 添加 sdcard.py 脚本

在 main.py 脚本的测试代码，如下所示：

```

from machine import Pin, SPI, I2C
from sdcard import SDCard
import st7735
import time
import uos

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':
    x = 0
    # IIC 初始化
    i2c0 = I2C(0, scl=Pin(42), sda=Pin(41), freq=400000)
    spi = SPI(2, baudrate=8000000, sck=Pin(12), mosi=Pin(11), miso=Pin(13))
    tft = st7735.atk_tft(spi, 40, 38, 39, 41, rotate=1) #DC, Reset, CS, BL, rotate
    sd = SDCard(spi, Pin(2, Pin.OUT))
    # 实验信息
    tft.p_string(0, 0, "ESP32-S3")
    tft.p_string(0, 12, "ATOM@ALIENTEK")
    tft.p_string(0, 24, "File Read:")

```

```
# 挂载到 SD/sd
uos.mount(sd, '/sd')
# 重新查询系统文件目录
print('挂载 SD 后的系统目录:{}' .format(uos.listdir()))
with open("/sd/test.txt", "w") as f:
    f.write(str("Hello ALIENTEK"))

# 从 sd 卡目录下读取 hello.txt 文件内容
with open("/sd/test.txt", "r") as f:
    # 打印读取的内容
    data = f.read()

tft.p_string(60, 24, str(data))
# 卸载 SD 卡
uos.umount('/sd')
```

在上述源代码中，首先实例化了 LCD 和 SD 卡对象。接着，系统显示了实验信息。随后，使用了 uos 模块的方法来挂载 SD 卡。最后，在 SD 卡的目录下创建了一个名为 test.txt 的文件，并将字符串数据“Hello ALIENTEK”写入该文件。同时，读取了该文件的内容，并将其显示在 LCD 显示屏上。

16.4 下载验证

程序下载到开发板后，可以看到 SPILCD 显示 SD 卡目录下的 test.txt 文件的内容，如下图所示。



图 16.4.1 RGBLCD 显示效果图

第三篇 高级篇

经过基础篇与入门篇的学习，我们已经对 MicroPython 的 machine 特定库的使用，有了一定的了解。从本篇开始，将迎来非常重要的实用的内容，那就是 WiFi 和蓝牙的应用。通过本篇的学习，我们可以体会到基于 MicroPython 的 WiFi、蓝牙等开发是多么简单而美妙。

- 1, UDP 实验
- 2, TCPClient 实验
- 3, TCPServer 实验
- 4, WebServer 实验
- 5, BLE 实验
- 6, AI 实验

第十七章 UDP 实验

在本章中，我们将使用 network 和 socket 第三方模块来实现网络连接。network 库用于实现 WiFi 连接，而 socket 库则用于实现基于 lwIP 协议栈的网络协议连接。在本章实验中，我们将使用 ESP32-S3 最小系统板连接远程网络，并在此基础上实现 UDP 连接。在本实验中，我们将实现一个简单的 UDP 连接，通过网络调试助手发送数据，并将接收到的数据原原本本地返回。

- 17.1 network 与 socket 库的简介
- 17.2 硬件设计
- 17.3 软件设计
- 17.4 下载验证

17.1 network 与 socket 库的简介

17.1.1 network 与 socket 库

MicroPython 的 network 和 socket 库都是用于实现网络连接的工具，它们具有一些共同的特点和不同的功能。

network 库是 MicroPython 的第三方库之一，它提供了 WiFi 和网络连接的功能。通过使用 network 库，我们可以连接到无线网络，获取网络状态，发送和接收数据等。在 MicroPython 的系统中，network 库通常用于实现设备的网络连接和通信功能。

socket，一个在网络世界中广泛使用的术语，它是我们程序与网络协议之间沟通的桥梁。想象一下，如果没有这座桥梁，我们的程序就需要直接与复杂的网络协议打交道，这无疑会增加程序设计的难度和复杂性。幸运的是，socket 为我们提供了一个优雅而简洁的解决方案。在众多的网络协议中，如 TCP、UDP、IP、ICMP 等，每一种都有其特定的通信规则和格式。而 socket，则像是为这些协议量身定制的一套适配器，让我们的程序可以无需关心底层的协议细节，就能轻松地进行网络通信。每一个网络通信的端点，都有一个 socket 对象，它们就像是网络通信的“门牌号”，让我们的程序知道应该将数据发送到哪个地址，同时也让网络知道应该将哪些数据发送到我们的程序。

在 Python 中，我们有一个名为 socket 的模块，它提供了一套简单易用的接口，让我们可以在程序中创建和使用 Socket 对象。通过这个模块，我们可以轻松地实现客户机和服务器的通信。socket 的背后，其实隐藏了一套复杂的系统。它不仅仅是一组接口，更是一种设计模式，一种门面模式。它将底层的 TCP/IP 协议族的复杂性隐藏起来，只暴露出一组简单的接口供我们使用。这样，我们无需深入理解网络协议的细节，只需要按照 socket 的接口规范进行编程，就能写出符合 TCP/UDP 标准的程序。下图是 socket 建立连接流程，如下所示。

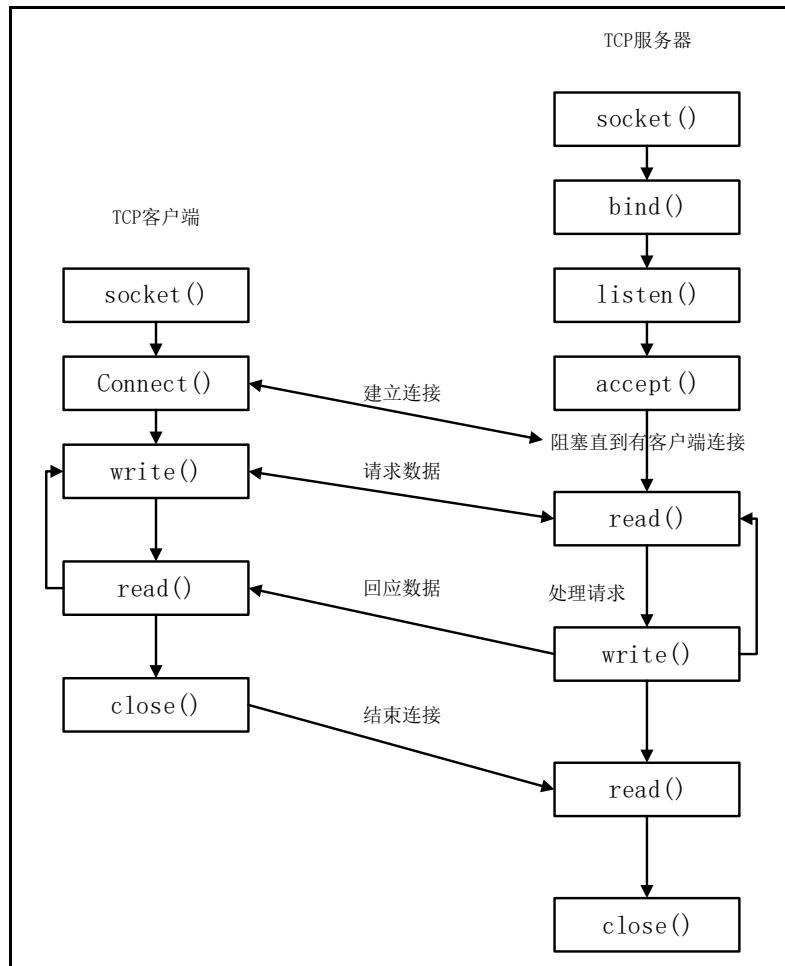


图 17.1.1.1 socket 建立连接流程

上图中，socket 建立连接需要六个步骤。首先，调用 `socket` 方法创建 `socket` 对象，接着，调用 `bind` 方法绑定地址到 `socket` 对象，然后，调用 `listen` 方法监听地址端口并且调用 `accept` 方法阻塞接收连接请求，当连接成功后，利用 `write` 方法和 `read` 方法处理通信数据，最后，如果接收到关闭请求时，就调用 `close` 方法关闭 `socket` 对象。

`network` 库和 `socket` 库的主要区别在于它们的应用场景和功能。`network` 库主要用于实现设备的 WiFi 和网络连接功能，而 `socket` 库则主要用于实现网络协议的连接和数据传输功能。因此，在实现具体的网络连接功能时，需要根据具体的需求和硬件平台来选择合适的库进行使用。

17.1.2 network 与 socket 库的构造与方法

一、network 库构造方法

WLAN 网络接口的构造对象方法如下：

```
class network.WLAN(interface_id)
使用示例: wlan = network.WLAN(network.STA_IF)
```

该构造函数的参数描述，如下表所示。

参数	描述	
interface_id	支持的接口	
	network.STA_IF	作为客户端，连接其他的服务器
	network.AP_IF	作为服务器，其他 WiFi 客户端连接

表 17.1.2.1 network.WLAN 构造方法参数描述

返回值：WLAN 网络接口对象。

二、network 库的方法

①：激活或停用网络接口。

其方法原型如下：

```
wlan.active([is_active])
```

该函数的参数描述，如下表所示。

参数	描述
is_active	True: 激活; False: 停用

表 17.1.2.2 wlan.active 方法参数描述

返回值：无。

②：连接网络。

其方法原型如下：

```
wlan.connect(ssid=None, password=None)
```

该函数的参数描述，如下表所示。

参数	描述
ssid	WiFi 账号
password	WiFi 密码

表 17.1.2.3 wlan.connect 方法参数描述

返回值：true: 连接成功； fail: 连接失败。

③：关闭网络。

其方法原型如下：

```
wlan.disconnect()
```

返回值：无。

④：获取或者设置网络参数。

其方法原型如下：

```
WLAN.ifconfig([(ip, subnet, gateway, dns)])
```

该方法的参数描述，如下表所示。

参数	描述
ip	IP 地址
subnet	子网掩码
gateway	网关
dns	DNS 服务器

表 17.1.2.4 wlan.ifconfig 方法参数描述

返回值：若此函数为无参数传入，则返回连接网络的信息，反次，为设置网络参数。

以上是 network 库常用的方法，还有其他方法可参考 MicroPython 的在线文档。

三、socket 库构造方法

创建套接字的构造对象方法如下：

```
class socket.socket(af=AF_INET, type=SOCK_STREAM, proto=IPPROTO_TCP, /)
```

使用示例：socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

该构造函数的参数描述，如下表所示。

参数	描述	
af	地址族	
	socket.AF_INET	IPv4
	socket.AF_INET6	IPv6
type	套接字类型	
	TCP 类型	socket.SOCK_STREAM
	UDP 类型	socket.SOCK_DGRAM
proto	协议（在大多数情况下不需要指定 proto）	
	socket.IPPROTO_UDP	UDP 协议
	socket.IPPROTO_TCP	TCP 协议

表 17.1.2.5 socket.socket 构造方法参数描述

返回值：套接字对象。

四、socket 库的方法

①: 关闭套接字接口。

其方法原型如下:

```
socket.close()
```

返回值: 无。

②: 套接字绑定到地址。

其方法原型如下:

```
socket.bind(address)
```

该方法的参数描述, 如下表所示。

参数	描述
address	IP 地址 (字符串形式传入, 如'192.168.101.33')

表 17.1.2.6 socket.bind 方法参数描述

返回值: 无。

③: 监听连接, 用作于 TCPServer 连接。

其方法原型如下:

```
socket.listen([backlog])
```

该方法的参数描述, 如下表所示。

参数	描述
backlog	监听连接数量

表 17.1.2.7 socket.listen 方法参数描述

返回值: conn: 新的套接字对象, 用来收发消息; address: 连接到服务器的客户端地址。

④: 接受连接, 用作于 TCPServer 连接。注: 在此之前, 需监听连接。

其函数原型如下:

```
socket.accept()
```

返回值: 无。

⑤: 连接远程 IP 地址

其函数原型如下:

```
socket.connect(address)
```

该函数的参数描述, 如下表所示。

参数	描述
address	IP 地址 (字符串形式传入, 如'192.168.101.33')

表 17.1.2.8 socket.connect 函数参数描述

返回值: 无。

⑥: 发送数据, 返回发送的字节数。

其函数原型如下:

```
socket.send(bytes)
```

该函数的参数描述, 如下表所示。

参数	描述
bytes	需发送的字节数据

表 17.1.2.9 socket.send 函数参数描述

返回值: 无。

⑦: 接收数据, 返回值是一个字节对象。

其函数原型如下:

```
socket.recv(bufsize)
```

该函数的参数描述, 如下表所示。

参数	描述
bufsize	接收数据的存储区

表 17.1.2.10 socket.recv 函数参数描述

返回值: 无。

⑧: UDP 发送数据, 一般用作于 UDP 连接。

其函数原型如下:

```
socket.sendto(bytes, address)
```

该函数的参数描述，如下表所示。

参数	描述
bytes	发送的数据
address	发送至哪个远程 IP 地址

表 17.1.2.11 socket.sendto 函数参数描述

返回值：无。

以上方法列出了本书中常用的套接字方法，对于剩余的套接字方法，请参考 MicroPython 最新的在线文档。

17.2 硬件设计

1. 例程功能

本章实验功能简介：当 UDP 连接成功后，网络调试助手作为服务器，开发板作为客户端，服务器发送数据至开发板时，开发板将数据原原本本的发送至网络调试助手。

2. 硬件资源

1) ESP32-S3 内部 WiFi

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

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。

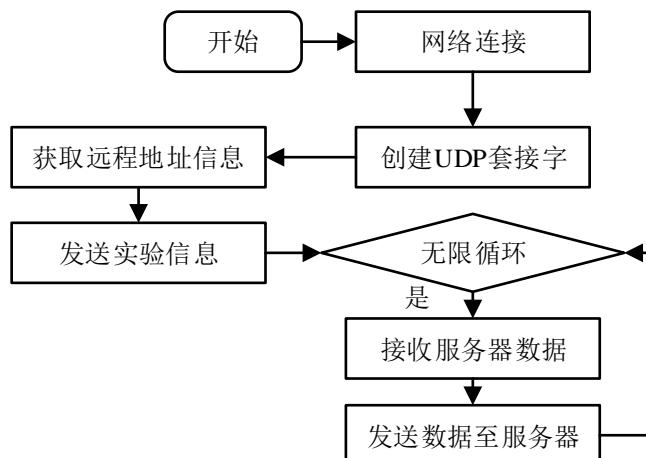


图 17.3.1.1 程序流程图

17.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。UDP 实验 main.py 源码如下：

```

import socket
import network
import time
  
```

```
SSID = "xxx"                      # wifi 名称
PASSWORD = "xxx"                   # wifi 密码
Server_IP = '192.168.101.33'       # 原程 IP 地址
wlan = None

"""
* @brief      连接网络
* @param      无
* @retval     无
"""
def connect():

    global wlan
    # 创建站点接口
    wlan = network.WLAN(network.STA_IF)
    wlan.active(False)
    # 启用站点接口
    wlan.active(True)

    # 判断是否连接
    if not wlan.isconnected():
        print('connecting to network...')
        # 连接 WiFi
        wlan.connect(SSID, PASSWORD)
        while not wlan.isconnected():
            pass
    # 输出网络信息
    print('network config: ', wlan.ifconfig())

"""
* @brief      程序入口
* @param      无
* @retval     无
"""
if __name__ == '__main__':

    # 连接网络
    connect()

    while True:
        # 获取本地 IP
        ip = wlan.ifconfig()[0]
        print('network config:', ip)
        """
        创建 socket 对象, socket.socket(ip 类型, 协议):
        socket.AF_INET 是 ipv4;socket.AF_INET6 是 ipv6
        socket.SOCK_STREAM 表示 tcp;SOCK_DGRAM 表示 UDP
        """
        udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        # ServerIP+ServerPort
        addr = socket.getaddrinfo('192.168.101.33', 8080)[0][-1]
        print(addr)
        # 发送信息
        udp_socket.sendto("*****\r\n",addr)
        udp_socket.sendto(f'正点原子 ESP32-S3 开发板 UDP Test\r\n'.
                           encode('utf-8'),addr)
        udp_socket.sendto("*****\r\n",addr)

        while True:
            try:
                # 每次接收 4096 字节

```

```
    data = udp_socket.recv(4096)
except :
    udp_socket.close()
    break
# 回显操作
udp_socket.sendto(data,addr)
```

在上述代码中，作者首先配置了 WiFi 的账号和密码，并构建了 ESP32-S3 与网络通道。接着，设置了服务器的远程 IP 地址，以便将数据发送到指定的地址。随后，作者定义了一个网络连接函数，该函数使用 network 第三方库的方法来实现网络连接。一旦连接建立，作者配置了连接协议。在本实验中，使用的是 UDP 连接，因此配置了套接字为 UDP 连接。接着，作者发送实验信息至服务器。最后，不断接收服务器发送的数值，并将数据原原本本地发送回服务器。

17.4 下载验证

程序下载到开发板后，需要打开网络调试助手，并在网络调试助手中配置协议为 UDP。在配置过程中，需要输入本机的 IP 地址，即当前电脑的 IP 地址，以及端口号 8080。另外，还需要配置目标主机信息的 IP 地址和端口号（即 ESP32-S3 设备的 IP 地址和端口号）。完成上述配置后，点击“连接”按钮以连接设备。一旦连接成功，可以在网络调试助手的发送框中填写需要发送的数据，并按下“发送按钮”以发送数据。此时，网络调试助手会接收到刚刚发送的数据。



图 17.4.1 网络调试助手配置与收发

第十八章 TCPClient 实验

在本章中，作者将实现一个简单的TCPClient连接，通过网络调试助手发送数据，并将接收到的数据原原本本地返回。

18.1 network 与 socket 库的简介

18.2 硬件设计

18.3 软件设计

18.4 下载验证

18.1 network 与 socket 库的简介

在第三十二章中，作者简要介绍了 network 与 socket 第三方源代码库，并详细阐述了这两个源代码库在 MicroPython 环境中的调用方法和参数传递说明。

18.2 硬件设计

1. 例程功能

本章实验功能简介：当 TCPClient 连接成功后，网络调试助手作为服务器，开发板作为客户端，服务器发送数据至开发板时，开发板将数据原原本本的发送至网络调试助手。

2. 硬件资源

1) ESP32-S3 内部 WiFi

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。

18.3 软件设计

18.3.1 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。

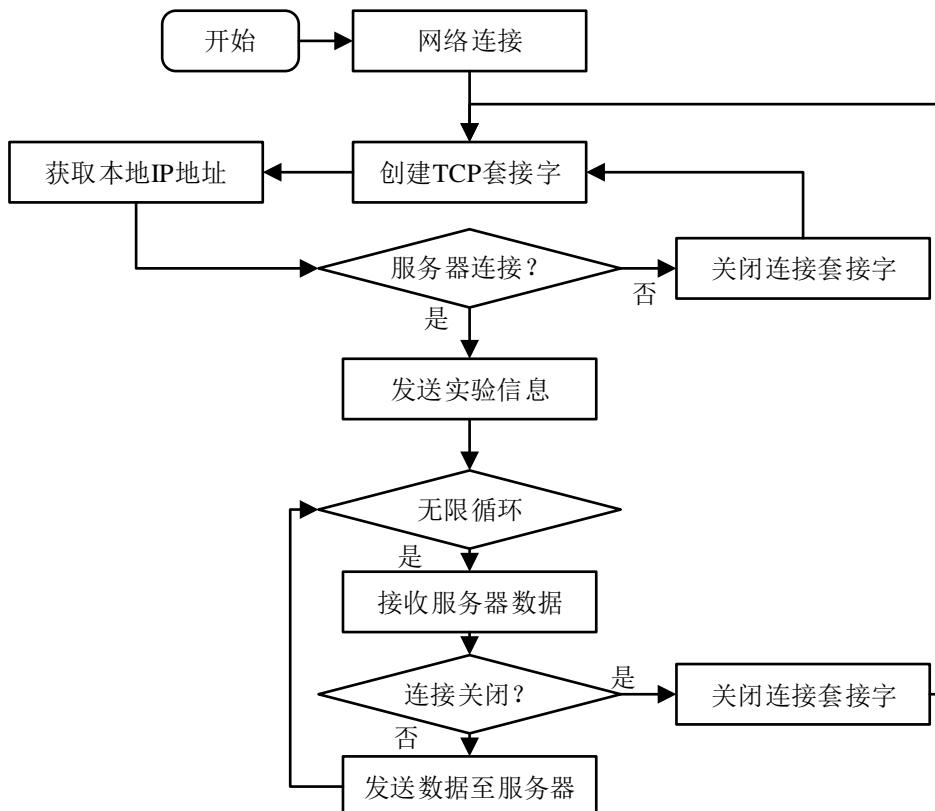


图 18.3.1.1 程序流程图

18.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。
TCPClient 实验 main.py 源码如下：

```

import socket
import network
import time

SSID = "xxx"                      # wifi 名称
PASSWORD = "xxx"                   # wifi 密码
Server_IP = '192.168.101.33'       # 原程 IP 地址
wlan = None
pos = 0

"""
 * @brief      连接网络
 * @param      无
 * @retval     无
"""
def connect():

    global wlan
    wlan = network.WLAN(network.STA_IF)
    wlan.active(False)
    wlan.active(True)

    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect(SSID, PASSWORD)
  
```

```
while not wlan.isconnected():
    pass
print('network config: ', wlan.ifconfig())

"""
* @brief      模拟 goto
* @param      label:标签
* @retval     无
"""
def goto(label):
    global pos
    pos = label

"""
* @brief      程序入口
* @param      无
* @retval     无
"""
if __name__ == '__main__':
    # 连接网络
    connect()

    while True:

        while pos == 0:
            # 获取本地 IP
            ip = wlan.ifconfig()[0]

        """
        创建 socket 对象, socket.socket(ip 类型, 协议):
        socket.AF_INET 是 ipv4;socket.AF_INET6 是 ipv6
        socket.SOCK_STREAM 表示 tcp;SOCK_DGRAM 表示 UDP
        """
        client_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        print('network config:', ip)

        """
        和服务器建立连接
        socket 对象 connect((服务器 ip 地址, 端口号))   类型是元组
        """
        try:
            client_socket.connect((Server_IP,8080))
        except BaseException:
            client_socket.close()
            print('关闭字节套.....')
            goto(0)
            break

        print('连接建立成功.....')
        # 发送信息
        client_socket.send("*****\r\n")
        client_socket.send(f'正点原子 ESP32-S3 开发板 TCPClient\r\n'
                           .encode('utf-8'))
        client_socket.send("*****\r\n")

        while True:
            try:
                # 每次接收 4096 字节
                data = client_socket.recv(4096)
            except BaseException:
                client_socket.close()
                break
```

```

# 如果收到空消息关闭连接
if (len(data) == 0):

    print("close socket")
    client_socket.close()
    break

# 回显操作
client_socket.send(data)

```

在上述代码中，作者首先配置了 WiFi 的账号和密码，并构建了 ESP32-S3 与网络通道。接着，设置了服务器的远程 IP 地址，以便将数据发送到指定的地址。随后，作者定义了一个网络连接函数，该函数使用 network 第三方库的方法来实现网络连接。一旦连接建立，作者配置了连接协议，还定义了模拟 C 语言的 goto 语句，为了重新连接服务器做准备。在本实验中，使用的是 TCP 连接，因此配置了套接字为 TCP 连接。接着，作者调用 client_socket.connect 函数连接远程服务器，如果连接成功，则发送实验信息至服务器，反次，关闭套接字并重新创建 TCP 套接字。最后，不断接收服务器发送的数据，并将数据原原本本地发送回服务器。

温馨提示：当服务器关闭连接时，系统会接收到连接关闭的消息，即上述源码的 len 大小为 0 时，程序执行关闭套接字和跳出第二个 while，最后系统不断尝试连接服务器，直到服务器重新连接。

18.4 下载验证

程序下载到开发板后，需要打开网络调试助手，并在网络调试助手中配置协议为 TCPServer。在配置过程中，需要输入本机的 IP 地址，即当前电脑的 IP 地址，以及端口号 8080。完成上述配置后，点击“连接”按钮以连接设备。一旦连接成功，可以在网络调试助手的发送框中填写需要发送的数据，并按下“发送按钮”以发送数据。此时，网络调试助手会接收到刚刚发送的数据。



图 18.4.1 网络调试助手配置与收发

第十九章 TCPServer 实验

在本章中，作者将实现一个简单的 TCPServer 连接，通过网络调试助手发送数据，并将接收到的数据原原本本地返回。

19.1 network 与 socket 库的简介

19.2 硬件设计

19.3 软件设计

19.4 下载验证

19.1 network 与 socket 库的简介

在第三十二章中，作者简要介绍了 network 与 socket 第三方源代码库，并详细阐述了这两个源代码库在 MicroPython 环境中的调用方法和参数传递说明。

19.2 硬件设计

1. 例程功能

本章实验功能简介：当 TCPServer 连接成功后，网络调试助手作为客户端，开发板作为服务器，客户端发送数据至开发板时，开发板将数据原原本本的发送至网络调试助手。

2. 硬件资源

1) ESP32-S3 内部 WiFi

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

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。

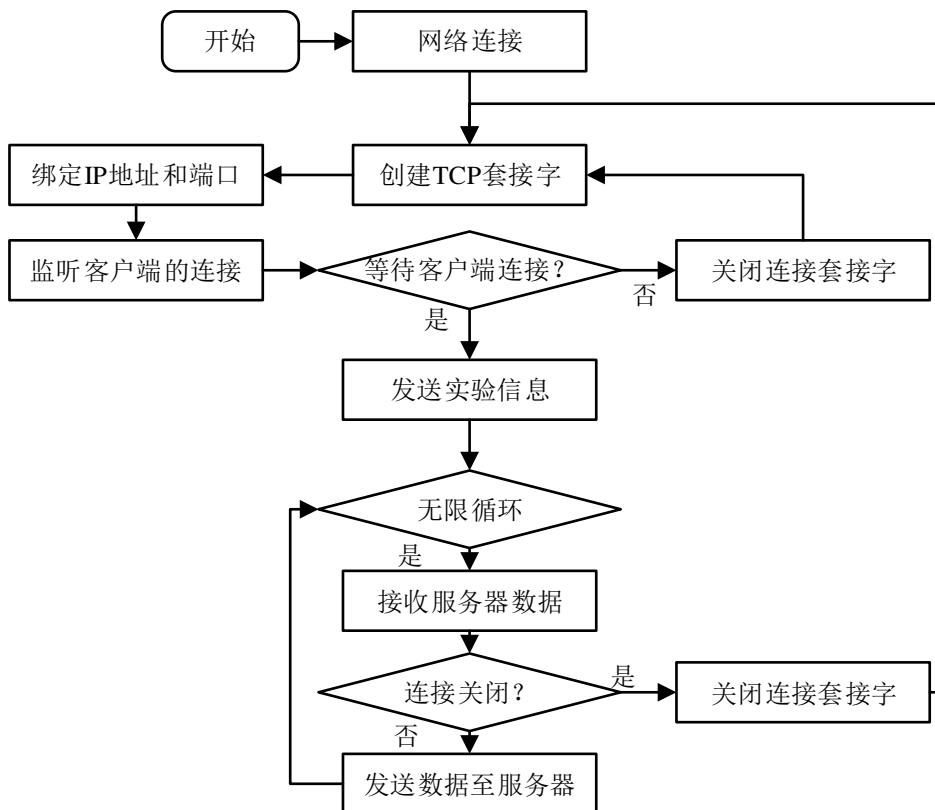


图 19.3.1.1 程序流程图

19.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。
TCPServer 实验 main.py 源码如下：

```

import socket
import network
import time

SSID = "xxx"          # wifi 名称
PASSWORD = "xxx"       # wifi 密码
wlan = None
pos = 0

"""
* @brief      连接网络
* @param      无
* @retval     无
"""
def connect():

    global wlan
    wlan = network.WLAN(network.STA_IF)
    wlan.active(False)
    wlan.active(True)

    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect(SSID, PASSWORD)
        while not wlan.isconnected():
            pass
  
```

```
print('network config: ', wlan.ifconfig())

"""
* @brief      模拟 goto
* @param      label:标签
* @retval     无
"""

def goto(label):

    global pos
    pos = label

"""

* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == '__main__':

    # 连接网络
    connect()

    while True:

        while pos == 0:
            # 获取本地 IP
            ip = wlan.ifconfig()[0]
            print('network config:', ip)
        """

        创建 socket 对象, socket.socket(ip 类型, 协议):
        socket.AF_INET 是 ipv4;socket.AF_INET6 是 ipv6
        socket.SOCK_STREAM 表示 tcp;SOCK_DGRAM 表示 UDP
        """
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # 设置端口复用
        server_socket.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,True)
        # 绑定本地 IP 以及端口
        server_socket.bind(('',8080)) # 前面空, 绑定服务器的任意一个网卡
        # 监听, 参数是同时连接的客户端数量
        server_socket.listen(128)
        # 阻塞等待客户端连接
        # 返回一个元组 (新 socket, (客户端 IP, 端口) )
        try:
            new_socket, client_ip_port = server_socket.accept()
        except:
            new_socket.close()
            server_socket.close()
            print('关闭字节套.....')
            goto(0)
            break

        print('连接建立成功.....')
        # 发送信息
        new_socket.send("*****\r\n")
        new_socket.send(f'正点原子 ESP32-S3 开发板 TCPServer\r\n'
                       '.encode('utf-8'))
        new_socket.send("*****\r\n")

        while True:
            try:
                # 每次接收 4096 字节
                data = new_socket.recv(4096)
            except :
```

```

new_socket.close()
server_socket.close()
break
# 如果收到空消息关闭连接
if (len(data) == 0):

    new_socket.close()
    server_socket.close()
    break

    # 回显操作
    new_socket.send(data)

```

在上述代码中，作者首先配置了 WiFi 的账号和密码，并构建了 ESP32-S3 与网络通道。随后，作者定义了一个网络连接函数，该函数使用 network 第三方库的方法来实现网络连接。一旦连接建立，作者配置了连接协议，还定义了模拟 C 语言的 goto 语句，为了重连做准备。在本实验中，使用的是 TCP 连接，因此配置了套接字为 TCP 连接。接着，作者调用 server_socket.bind、server_socket.listen 绑定 IP 地址和监听客户端，还调用 server_socket.accept 函数判断是否有客户端发送连接请求，如果连接成功，则发送实验信息至服务器，反次，关闭套接字并重新创建 TCP 套接字。最后，不断接收服务器发送的数值，并将数据原原本本地发送回服务器。

温馨提示：当客户端关闭连接时，系统会接收到连接关闭的消息，即上述源码的 len 大小为 0 时，程序执行关闭套接字和跳出第二个 while，最后系统不断监听客户端连接，直到客户端重新连接。

19.4 下载验证

程序下载到开发板后，需要打开网络调试助手，并在网络调试助手中配置协议为 TCPClient。在配置过程中，需要输入 ESP32-S3 的 IP 地址以及端口号 8080。完成上述配置后，点击“连接”按钮以连接设备。一旦连接成功，可以在网络调试助手的发送框中填写需要发送的数据，并按下“发送按钮”以发送数据。此时，网络调试助手会接收到刚刚发送的数据。

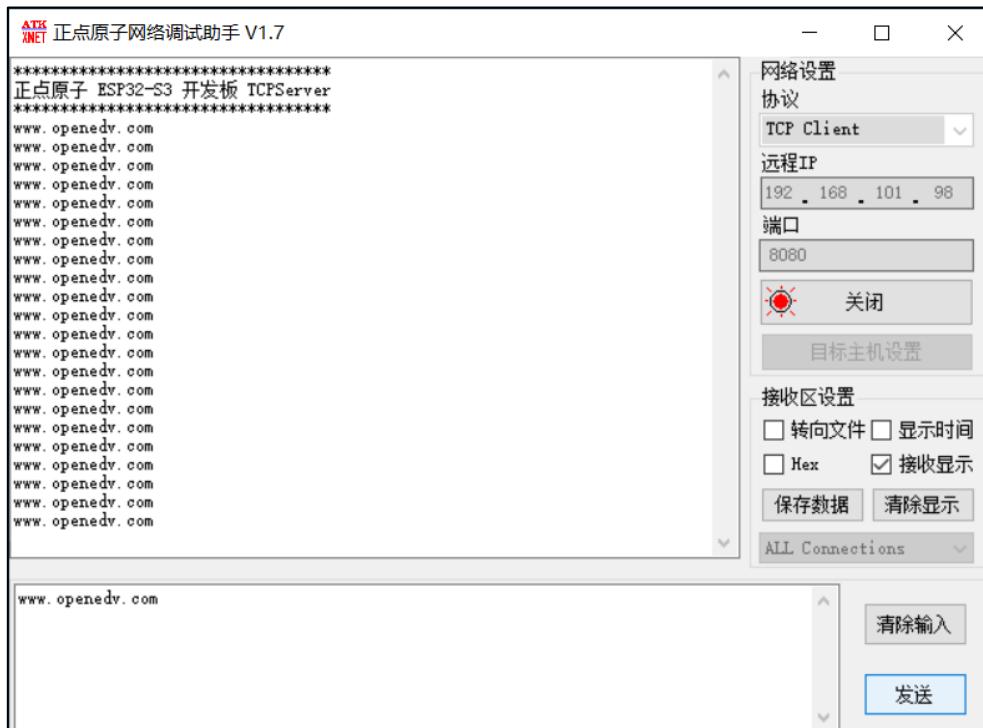


图 19.4.1 网络调试助手配置与收发

第二十章 BLE 实验

前面作者也讲解到，ESP32-S3 是一款专为物联网应用设计的芯片，支持 Wi-Fi 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。本章，作者使用 MicroPython 提供的 BLE 特定库驱动 ESP32-S3 内部的蓝牙设备，并实现控制板载的 LED 灯及接收 ESP32-S3 设备发送的数据。

- 20.1 BLE 特定库
- 20.2 硬件设计
- 20.3 软件设计
- 20.4 下载验证

20.1 BLE 特定库

20.1.1 BLE 特定库简介

MicroPython 的 bluetooth 模块是一种在 MicroPython 环境中用于处理蓝牙连接和通信的模块。它提供了在 MicroPython 设备之间，以及 MicroPython 设备与其他蓝牙设备之间进行无线通信的功能。通过使用 bluetooth 模块，MicroPython 用户可以搜索附近的蓝牙设备、建立与这些设备的连接、发送和接收数据，以及进行其他相关的蓝牙通信操作。该模块支持蓝牙低功耗协议（BLE），也支持经典蓝牙协议。

在 MicroPython 中使用 bluetooth 模块，需要正确配置和设置相应的硬件和软件。具体来说，需要确保 MicroPython 设备支持蓝牙功能，并且已经正确连接到蓝牙模块或适配器。此外，用户还需要了解如何使用 bluetooth 模块提供的函数和方法来进行具体的蓝牙通信操作。

20.1.2 BLE 特定库的构造与方法

一、BLE 库构造方法

BLE 蓝牙的构造对象方法如下：

```
class bluetooth.BLE()
```

使用示例：ble = bluetooth.BLE()

返回值：BLE 蓝牙对象。

二、BLE 库的方法

①：激活或停用蓝牙，返回当前状态。

其方法原型如下：

```
ble.active([is_active])
```

该函数的参数描述，如下表所示。

参数	描述
is_active	True: 激活; False: 停用

表 20.1.2.1 ble.active 方法参数描述

②：配置蓝牙参数。

其方法原型如下：

```
ble.config('gap_name', /)
```

该方法的参数描述，如下表所示。

参数	描述
gap_name	设置蓝牙设备名称

表 20.1.2.2 ble.config 方法参数描述

③：设置蓝牙中断。

其方法原型如下：

```
ble.irq(handler, /)
```

该函数的参数描述，如下表所示。

参数	描述
handler	中断回调函数

表 20.1.2.3 ble.irq 方法参数描述

④：接收蓝牙数据。

其方法原型如下：

```
ble.gatts_read(value_handle, /)
```

该方法的参数描述，如下表所示。

参数	描述
value_handle	接收存储区

表 20.1.2.3 ble.atts_read 方法参数描述

⑤：发送蓝牙数据。

其方法原型如下：

```
ble.gatts_notify(conn_handle, value_handle, data=None, /)
```

该方法的参数描述，如下表所示。

参数	描述
conn_handle	发送通道，0通道已经被串口占用，建议设置为1
value_handle	发送控制块
data	要发送的数据

表 20.1.2.4 ble.gatts_notify 方法参数描述

⑥：注册蓝牙服务器。

其方法原型如下：

```
ble.gatts_register_services(services_definition, /)
```

该方法的参数描述，如下表所示。

参数	描述
services_definition	服务器配置参数

表 20.1.2.5 ble.gatts_register_services 方法参数描述

⑦：允许设备向周围的其他 BLE 设备发送广播信号，以便它们可以发现并连接到该设备。

其方法原型如下：

```
ble.gap_advertise(interval_us, adv_data=None, *)
```

该方法的参数描述，如下表所示。

参数	描述
interval_us	广播间隔
adv_data	广播数据

表 20.1.2.6 ble.gap_advertise 方法参数描述

以上是 MicroPython bluetooth 模块常用的方法（函数），其他 BLE 函数可参看 MicroPython 官方在线文档。

20.2 硬件设计

1. 例程功能

本章实验功能简介：通过 BLUE 调试 APP 来控制 LED，当接收“LED ON”命令时，系统打开 LED；当接收“LED OFF”命令时，系统关闭 LED。

2. 硬件资源

1) ESP32-S3 内部蓝牙

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

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。

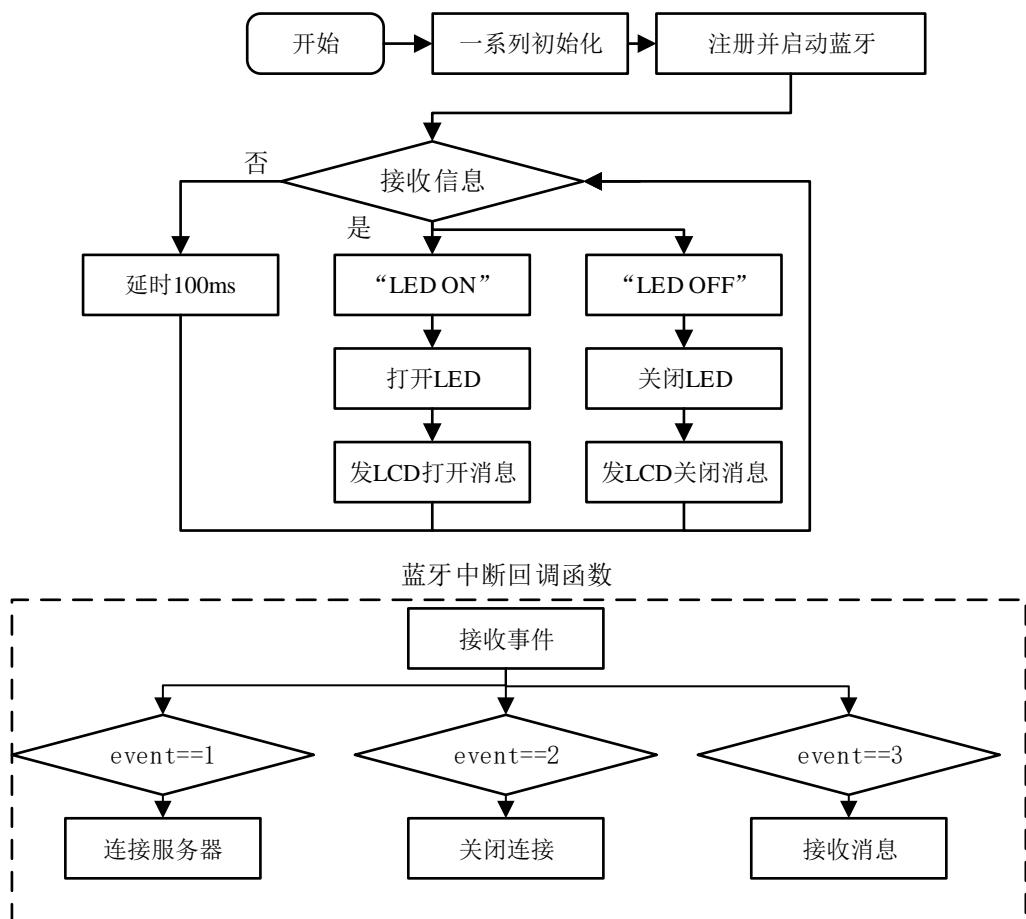


图 20.3.1.1 程序流程图

20.3.2 程序解析

本书籍的代码都在 main.py 脚本下编写的，读者可在光盘资料下找到对应的源码。BLE 实验 main.py 源码如下：

```

from machine import Pin,Timer
import time
import bluetooth

# 定义全局变量
BLE_MESSAGE = ""

"""
 * @brief      蓝牙类
 * @param      无
 * @retval     无
"""

class ESP32S3_BLE():
    def __init__(self, name):

```

```
# 获取定时器 0 句柄
self.timer1 = Timer(0)
self.name = name
# 初始化蓝牙
self.ble = bluetooth.BLE()
# 开启蓝牙
self.ble.active(True)
# 设置蓝牙名称
self.ble.config(gap_name=name)
# 配置定时器 0 且开启定时器
self.disconnected()
# 配置蓝牙回调函数
self.ble.irq(self.ble_irq)
# 注册蓝牙
self.register()
self.advertiser()

# 蓝牙连接时，点亮 LED 且关闭定时器 0
def connected(self):
    led.value(1)
    self.timer1.deinit()

# 蓝牙断开时，闪烁 LED
def disconnected(self):
    self.timer1.init(period=1000, mode=Timer.PERIODIC,
                     callback=lambda t: led.value(not led.value())))

# 蓝牙回调函数
def ble_irq(self, event, data):

    global BLE_MESSAGE
    # _IRQ_CENTRAL_CONNECT 蓝牙终端链接了此设备
    if event == 1:
        self.connected()
    # _IRQ_CENTRAL_DISCONNECT 蓝牙终端断开此设备
    elif event == 2:
        self.advertiser()
        self.disconnected()
    # _IRQ_GATTS_WRITE 蓝牙终端向 ESP32-S3 发送数据，接收数据处理
    elif event == 3:
        buffer = self.ble.gatts_read(self.rx)
        BLE_MESSAGE = buffer.decode('UTF-8').strip()

def register(self):
    service_uuid = '6E400001-B5A3-F393-E0A9-E50E24DCCA9E'
    reader_uuid = '6E400002-B5A3-F393-E0A9-E50E24DCCA9E'
    sender_uuid = '6E400003-B5A3-F393-E0A9-E50E24DCCA9E'
    services = (
        (
            bluetooth.UUID(service_uuid),
            (
                (bluetooth.UUID(sender_uuid), bluetooth.FLAG_NOTIFY),
                (bluetooth.UUID(reader_uuid), bluetooth.FLAG_WRITE),
            )
        ),
        ((self.tx, self.rx,), ) = self.ble.gatts_register_services(services)
    # 向蓝牙 APP 发送数据
    def send(self, data):
        self.ble.gatts_notify(1, self.tx, data + '\n')
    def advertiser(self):
        name = bytes(self.name, 'UTF-8')
        adv_data = bytearray('\x02\x01\x02', 'UTF-8') + bytearray((len(name) + 1, 0x09), 'UTF-8') + name
```

```

self.ble.gap_advertise(100, adv_data)

"""
* @brief      程序入口
* @param      无
* @retval     无
"""

if __name__ == "__main__":
    led = Pin(1, Pin.OUT,value = 1)
    # 创建蓝牙对象且设置蓝牙名称
    ble = ESP32S3_BLE("ESP32-S3 BLE")

    while True:

        # 打开 LED
        if BLE_MESSAGE == 'LED ON':
            led.value(0)
            print('LED is ON.')
            ble.send('LED is ON.')
            BLE_MESSAGE = ""

        # 关闭 LED
        elif BLE_MESSAGE == 'LED OFF':
            led.value(1)
            print('LED is OFF.')
            ble.send('LED is OFF.')
            BLE_MESSAGE = ""

        time.sleep_ms(100)

```

在上述源码中，作者创建了一个名为“ESP32-S3 BLE”的蓝牙对象。通过调用该对象内部的方法，作者成功地构建了一个蓝牙服务器，并向周围的设备发送了广播信号。当某个 APP 或其他蓝牙设备与 ESP32-S3 的蓝牙连接成功建立时，系统会进入等待状态，直到 APP 发送“LED ON”或“LED OFF”命令。一旦接收到这些命令，系统会相应地控制 LCD 的亮灭，并向 APP 发送确认消息。

20.4 下载验证

程序下载至开发板之后，我们可在手机蓝牙管理找到 ESP32-S3 BLE，如下图所示。

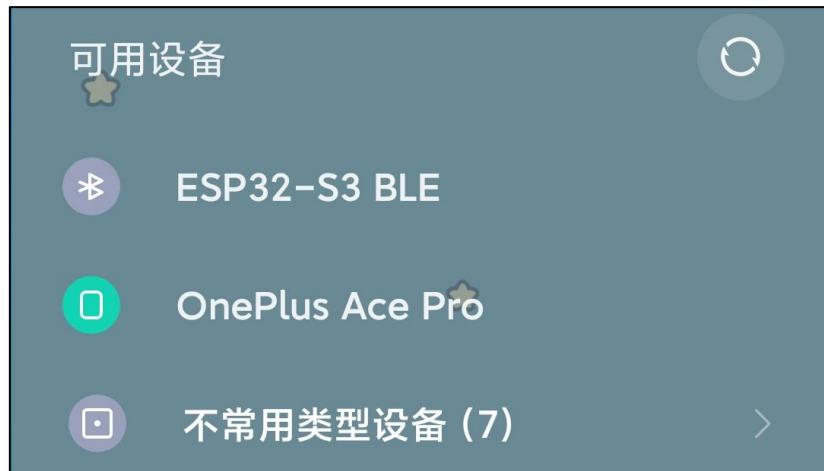


图 20.4.1 搜索蓝牙设备

搜索到 ESP32-S3 蓝牙设备之后，在手机的应用商店 APP 下载 BLE 调试助手，如下图所示。



图 20.4.2 下载 BLE 调试助手

打开 BLE 调试助手，在 Scanner 界面下往下刷新界面（搜索蓝牙设备），接着，找到“ESP32-S3 BLE”蓝牙设备，点击“CONNECT”连接设备，如下图所示。

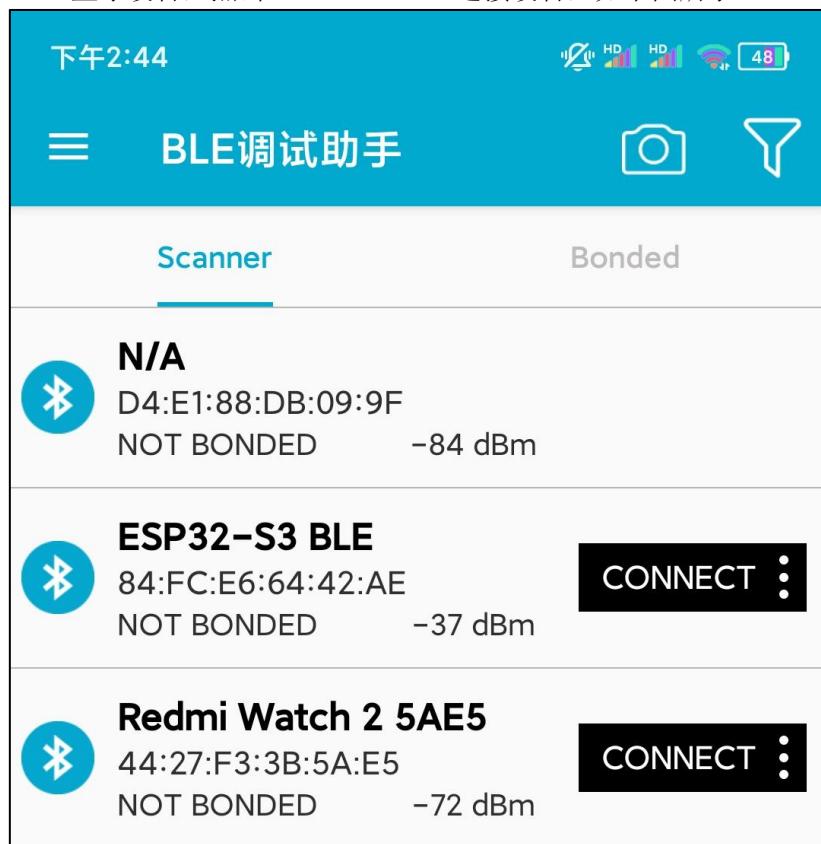


图 20.4.3 连接设备

连接成功之后，我们点击 Unknown Service 项目栏下的 Unknown Characteristic 选项下发数据至 ESP32-S3 最小系统板，如下图所示。

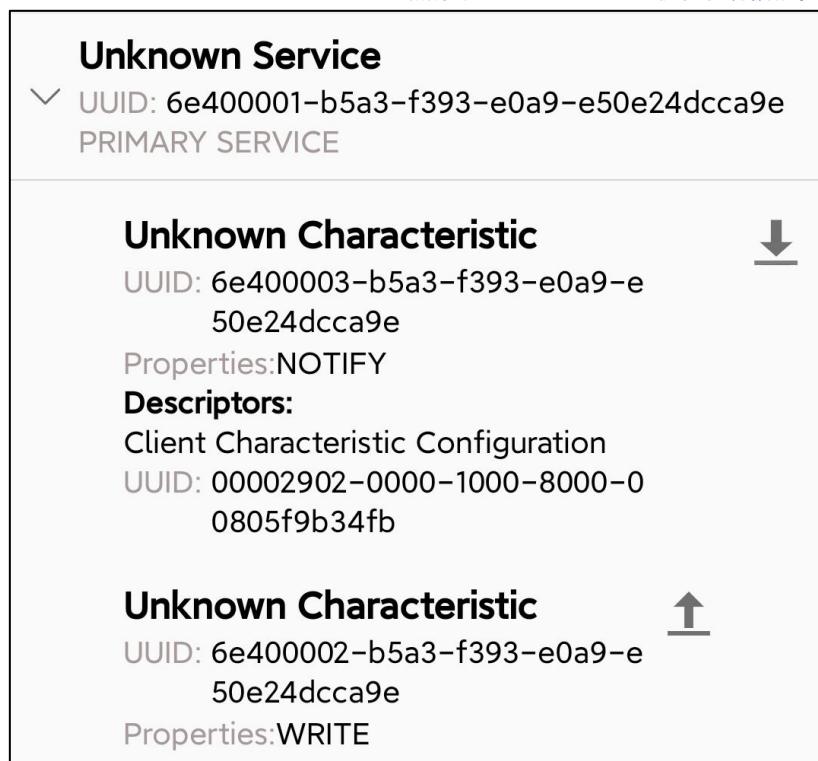


图 20.4.4 下发数据的入口

点击上图中的向上箭头，选择发送数据模式和消息内容，如果我们下发“LED ON”消息时，则点亮开发板的 LED；如果我们下发“LED OFF”消息，则熄灭开发板的 LCD。



图 20.4.5 下发数据

温馨提示：接收数据入口在图 20.4.4 中的“Unknown Characteristic”选项。