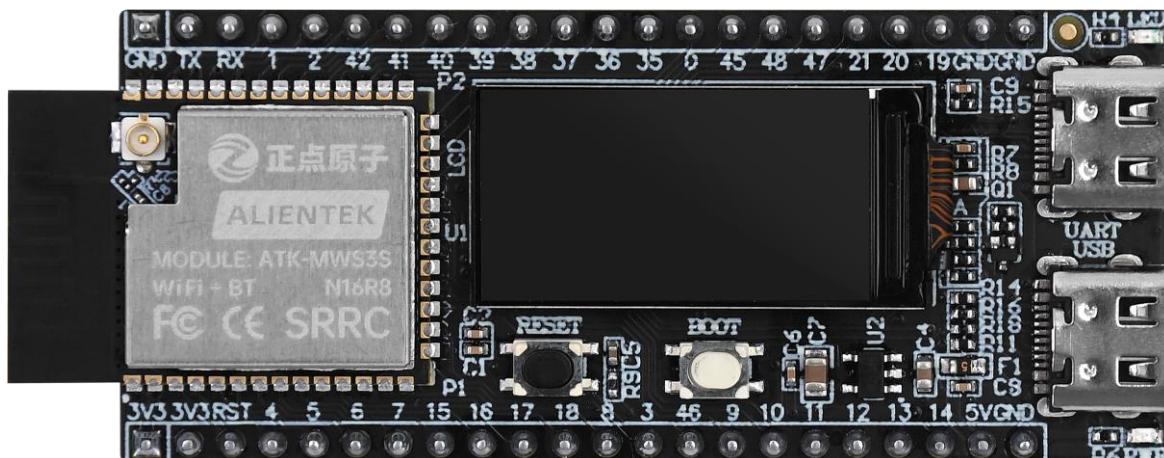


# ESP32-S3 使用指南—IDF 版

V1.1



—正点原子 DNESP32S3M 最小系统板教程

注：本教程仅适用于 DNESP32S3M 最小系统板

**修订历史:**

版本	日期	修改内容
V1.0	2024/6/15	第一次发布
V1.1	2024/7/1	更新 launch.json 文件内容



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](http://www.yuanzige.com)  
开源电子网 / 论坛 : [www.openedv.com](http://www.openedv.com)  
正点原子官方网站 : [www.alientek.com](http://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



扫码关注正点原子公众号

前言 .....	1
内容简介 .....	2
注意事项 .....	3
第一篇 基础篇 .....	4
第一章 本书学习方法 .....	5
1.1 本书学习顺序 .....	5
1.2 本书参考资料 .....	5
1.3 本书编写规范 .....	5
1.4 本书代码规范 .....	6
1.5 例程资源说明 .....	6
1.6 学习资料查找 .....	7
1.7 给初学者的建议 .....	9
第二章 常用的 C 语言知识点 .....	11
5.1 位操作 .....	11
5.2 define 宏定义 .....	11
5.3 ifdef 条件编译 .....	12
5.4 extern 外部申明 .....	13
5.5 typedef 类型别名 .....	14
5.6 struct 结构体 .....	14
5.7 指针 .....	14
第三章 ESP32-S3 基础知识 .....	17
3.1 为什么选择 ESP32-S3 .....	17
3.2 初识 ESP32-S3 .....	18
3.3 ESP32-S3 资源简介 .....	19
3.4 S3 系列型号对比 .....	20
3.5 ESP32-S3 功能概述 .....	22
3.5.1 系统和存储器 .....	22
3.5.2 IO MUX 和 GPIO 交换矩阵 .....	27
3.5.3 复位与时钟 .....	31
3.5.4 芯片 Boot 控制 .....	34
3.5.5 中断矩阵 .....	36
3.6 ESP32-S3 启动流程 .....	37
第四章 认识 ESP-IDF .....	40
4.1 ESP-IDF 简介 .....	40
4.2 ESP-IDF 库框架结构解析 .....	42
4.3 ESP-IDF 与乐鑫芯片 .....	43
4.4 IDF 工程简介 .....	43
第五章 搭建开发环境 .....	45
5.1 安装 ESP-IDF 物联网开发框架 .....	45
5.2 IDF 前端工具 .....	51
5.3 搭建集成开发环境 .....	56
5.3.1 VS Code 安装 .....	56
5.3.2 安装与配置 ESP-IDF 插件 .....	60
5.3.3 个性化配置和工作环境配置 .....	63

第六章 新建基础工程.....	64
6.1 搭建基础工程.....	64
6.2 基础工程的文件架构解析.....	66
6.3 调试相关工具介绍.....	67
6.3.1 串口下载.....	68
6.3.2 JTAG 下载与调试.....	69
6.4 原子工程的文件架构解析.....	73
6.5 基础工程配置.....	75
第七章 分区表.....	81
7.1 分区表概述.....	81
7.2 分区表 API 函数 .....	82
第八章 MENUCONFIG 菜单配置 .....	84
8.1 menuconfig 概述.....	84
8.2 menuconfig 实现原理.....	85
8.3 配置项解析.....	85
第九章 IDF 组件注册表 .....	93
9.1 IDF 组件注册表简介 .....	93
9.2 项目工程如何添加组件.....	96
9.2.1 命令式添加组件.....	96
9.2.2 VS Code 工程添加组件 .....	97
第二篇 入门篇.....	100
第十章 LED 实验 .....	101
10.1 GPIO&LED 简介.....	101
10.1.1 GPIO 简介.....	101
10.1.2 LED 简介 .....	101
10.2 硬件设计.....	102
10.2.1 例程功能.....	102
10.2.2 硬件资源.....	102
10.2.3 原理图.....	102
10.3 程序设计.....	103
10.3.1 程序流程图.....	103
10.3.2 GPIO 函数解析.....	103
10.3.3 LED 驱动解析 .....	105
10.3.4 CMakeLists.txt 文件 .....	106
10.3.5 实验应用代码.....	106
10.4 下载验证.....	107
第十一章 KEY 实验.....	108
11.1 独立按键基础知识.....	108
11.2 硬件设计.....	109
11.2.1 例程功能.....	109
11.2.2 硬件资源.....	109
11.2.3 原理图.....	109
11.3 程序设计.....	109
11.3.1 程序流程图.....	109

11.3.2 GPIO 函数解析.....	110
11.3.3 KEY 驱动解析.....	110
11.3.4 CMakeLists.txt 文件 .....	111
11.3.5 实验应用代码.....	111
11.4 下载验证.....	112
第十二章 EXIT 实验.....	113
12.1 中断简介.....	113
12.2 硬件设计.....	115
12.2.1 例程功能.....	115
12.2.2 硬件资源.....	115
12.2.3 原理图.....	115
12.3 程序设计.....	115
12.3.1 程序流程图.....	115
12.3.2 EXIT 函数解析.....	115
12.3.3 EXIT 驱动解析.....	116
12.3.4 CMakeLists.txt 文件 .....	117
12.3.5 实验应用代码.....	117
12.4 下载验证.....	118
第十三章 UART 实验.....	119
13.1 串口简介.....	119
13.1.1 数据通信的基础概念.....	119
13.1.2 串口通信协议简介.....	120
13.1.3 ESP32-S3 的 UART 简介 .....	122
13.1.4 ESP32-S3 的 UART 框架图介绍.....	122
13.2 硬件设计.....	123
13.2.1 例程功能.....	123
13.2.2 硬件资源.....	123
13.2.3 原理图.....	123
13.3 程序设计.....	124
13.3.1 程序流程图.....	124
13.3.2 UART 函数解析 .....	124
13.3.3 UART 驱动解析 .....	127
13.3.4 CMakeLists.txt 文件 .....	128
13.3.5 实验应用代码.....	128
13.4 下载验证.....	129
第十四章 ESPTIMER 实验.....	131
14.1 定时器简介.....	131
14.2 硬件设计.....	132
14.2.1 例程功能.....	132
14.2.2 硬件资源.....	133
14.2.3 原理图.....	133
14.3 程序设计.....	133
14.3.1 程序流程图.....	133
14.3.2 ESPTIMER 函数解析 .....	133
14.3.3 ESPTIMER 驱动解析 .....	134
14.3.4 CMakeLists.txt 文件 .....	135
14.3.5 实验应用代码.....	135

14.4 下载验证.....	136
第十五章 SW_PWM 实验 .....	137
15.1 PWM 简介 .....	137
15.2 硬件设计.....	138
15.2.1 例程功能.....	138
15.2.2 硬件资源.....	138
15.2.3 原理图.....	138
15.3 程序设计.....	139
15.3.1 程序流程图.....	139
15.3.2 SW_PWM 函数解析 .....	139
15.3.3 SW_PWM 驱动解析 .....	142
15.3.4 CMakeLists.txt 文件 .....	143
15.3.5 实验应用代码.....	143
15.4 下载验证.....	144
第十六章 HW_PWM 实验.....	145
16.1 PWM 简介 .....	145
16.2 硬件设计.....	145
16.2.1 例程功能.....	145
16.2.2 硬件资源.....	145
16.2.3 原理图.....	145
16.3 程序设计.....	145
16.3.1 程序流程图.....	145
16.3.2 HW_PWM 函数解析 .....	146
16.3.3 HW_PWM 驱动解析 .....	147
16.3.4 CMakeLists.txt 文件 .....	148
16.3.5 实验应用代码.....	148
16.4 下载验证.....	149
第十七章 SPI_LCD 实验.....	150
17.1 SPI 及 LCD 介绍 .....	150
17.1.1 SPI 介绍 .....	150
17.1.2 SPI 控制器介绍 .....	151
17.1.3 LCD 介绍 .....	152
17.2 硬件设计.....	155
17.2.1 例程功能.....	155
17.2.2 硬件资源.....	155
17.2.3 原理图.....	156
17.3 程序设计.....	156
17.3.1 程序流程图.....	156
17.3.2 SPI_LCD 函数解析 .....	156
17.3.3 SPI_LCD 驱动解析 .....	158
17.3.4 CMakeLists.txt 文件 .....	163
17.3.5 实验应用代码.....	164
17.4 下载验证.....	165
第十八章 RTC 实验 .....	166
18.1 RTC 时钟简介 .....	166
18.2 硬件设计.....	166

18.2.1 例程功能.....	166
18.2.2 硬件资源.....	166
18.2.3 原理图.....	166
18.3 程序设计.....	167
18.3.1 程序流程图.....	167
18.3.2 RTC 函数解析 .....	167
18.3.3 RTC 驱动解析 .....	168
18.3.4 CMakeLists.txt 文件 .....	169
18.3.5 实验应用代码.....	170
18.4 下载验证.....	171
第十九章 内部温度传感器实验.....	172
19.1 内部温度传感器简介.....	172
19.2 硬件设计.....	172
19.2.1 例程功能.....	172
19.2.2 硬件资源.....	172
19.2.3 原理图.....	172
19.3 程序设计.....	172
19.3.1 程序流程图.....	172
19.3.2 内部温度传感器函数解析 .....	173
19.3.3 内部温度传感器驱动解析 .....	174
19.3.4 CMakeLists.txt 文件 .....	175
19.3.5 实验应用代码.....	175
19.4 下载验证.....	176
第二十章 RNG 实验 .....	177
20.1 随机数发生器简介.....	177
20.1.1 RNG 功能描述 .....	177
20.1.2 RNG 随机数寄存器 .....	177
20.2 硬件设计.....	178
20.2.1 例程功能.....	178
20.2.2 硬件资源.....	178
20.2.3 原理图.....	178
20.3 程序设计.....	178
20.3.1 程序流程图.....	178
20.3.2 RNG 函数解析 .....	179
20.3.3 RNG 驱动解析 .....	179
20.3.4 CMakeLists.txt 文件 .....	180
20.3.5 实验应用代码.....	180
20.4 下载验证.....	181
第二十一章 SPI_SDCARD 实验 .....	182
22.1 SD 卡简介.....	182
22.1.1 SD 物理结构.....	182
21.1.2 命令和响应.....	184
21.1.3 卡模式.....	186
21.1.4 数据模式.....	187
21.1.5 SD 卡初始化流程.....	189
21.2 硬件设计.....	194
21.2.1 例程功能.....	194

21.2.2 硬件资源.....	194
21.2.3 原理图.....	194
21.3 程序设计.....	195
21.3.1 程序流程图.....	195
21.3.2 SD 卡函数解析.....	195
21.3.3 SD 卡驱动解析.....	197
21.3.4 CMakeLists.txt 文件 .....	200
21.3.5 实验应用代码.....	200
21.4 下载验证.....	201
第二十二章 SPIFFS 实验 .....	202
22.1 SPIFFS 介绍 .....	202
22.2 硬件设计.....	202
22.2.1 例程功能.....	202
22.2.2 硬件资源.....	202
22.2.3 原理图.....	202
22.3 程序设计.....	203
22.3.1 程序流程图.....	203
22.3.2 SPIFFS 函数解析 .....	203
22.3.3 SPIFFS 驱动解析 .....	204
22.3.4 CMakeLists.txt 文件 .....	204
22.3.5 实验应用代码.....	204
22.4 下载验证.....	207
第二十三章 汉字显示实验.....	208
23.1 汉字显示原理简介.....	208
23.1.1 字符编码介绍.....	208
23.1.2 汉字字库简介.....	209
23.1.3 汉字显示原理.....	212
23.2 硬件设计.....	212
23.2.1 例程功能.....	212
23.2.2 硬件资源.....	212
23.2.3 原理图.....	213
23.3 程序设计.....	213
23.3.1 程序流程图.....	213
23.3.2 汉字显示函数解析.....	213
23.3.3 汉字显示驱动解析.....	213
23.3.4 CMakeLists.txt 文件 .....	220
23.3.5 实验应用代码.....	221
23.4 下载验证.....	222
第二十四章 图片显示实验.....	224
24.1 图片格式介绍.....	224
24.1.1 BMP 编码简介 .....	224
24.1.2 JPEG 编码简介 .....	224
24.1.3 PNG 编码简介 .....	225
24.1.4 GIF 编码简介 .....	225
24.2 硬件设计.....	226
24.2.1 例程功能.....	226
24.2.2 硬件资源.....	226

24.3 程序设计.....	226
24.3.1 程序流程图.....	226
24.3.2 图片显示函数解析.....	226
24.3.3 图片显示函数驱动解析.....	227
24.3.4 CMakeLists.txt 文件 .....	229
24.3.5 实验应用代码.....	230
24.4 下载验证.....	233
第二十五章 USB 虚拟串口(Slave)实验.....	234
25.1 USB 虚拟串口简介 .....	234
25.2 硬件设计.....	234
25.2.1 例程功能.....	234
25.2.2 硬件资源.....	234
25.2.3 原理图.....	234
25.3 程序设计.....	235
25.3.1 程序流程图.....	235
25.3.2 USB 虚拟串口函数解析 .....	236
25.3.3 USB 虚拟串口驱动解析 .....	237
25.3.4 CMakeLists.txt 文件 .....	238
25.3.5 实验应用代码.....	238
25.4 下载验证.....	239
第二十六章 Flash 模拟 U 盘实验.....	241
26.1 Flash 模拟 U 盘简介.....	241
26.2 硬件设计.....	241
26.2.1 例程功能.....	241
26.2.2 硬件资源.....	241
26.2.3 原理图.....	241
26.3 程序设计.....	241
26.3.1 程序流程图.....	241
26.3.2 Flash 模拟 U 盘函数解析 .....	242
26.3.3 USB 虚拟串口驱动解析 .....	243
26.3.4 CMakeLists.txt 文件 .....	244
26.3.5 实验应用代码.....	244
26.4 下载验证.....	245
第二十七章 SD 卡模拟 U 盘实验 .....	247
27.1 Flash 模拟 U 盘简介.....	247
27.2 硬件设计.....	247
27.2.1 例程功能.....	247
27.2.2 硬件资源.....	247
27.2.3 原理图.....	247
27.3 程序设计.....	247
27.3.1 程序流程图.....	247
27.3.2 SD 卡模拟 U 盘函数解析 .....	248
27.3.3 SD 卡模拟 U 盘驱动解析 .....	249
27.3.4 CMakeLists.txt 文件 .....	250
27.3.5 实验应用代码.....	250
27.4 下载验证.....	251
第三篇 高级篇.....	253

第二十八章 lwIP 初探 .....	254
28.1 TCP/IP 协议栈是什么 .....	254
28.1.1 TCP/IP 协议栈架构 .....	254
28.1.2 TCP/IP 协议栈的封包和拆包 .....	255
28.2 lwIP 简介 .....	256
28.3 WiFi MAC 内核简介 .....	257
28.4 lwIP Socket 编程接口 .....	258
第二十九章 扫描 WiFi 实验 .....	262
29.1 WiFi 模式概述 .....	262
29.2 硬件设计 .....	263
1. 例程功能 .....	263
2. 硬件资源 .....	263
3. 原理图 .....	263
29.3 软件设计 .....	263
29.3.1 程序流程图 .....	263
29.3.2 程序解析 .....	264
29.4 下载验证 .....	266
第三十章 WiFi 路由实验 .....	267
30.1 硬件设计 .....	267
1. 例程功能 .....	267
2. 硬件资源 .....	267
3. 原理图 .....	267
30.2 软件设计 .....	267
30.2.1 程序流程图 .....	267
30.2.2 程序解析 .....	268
30.3 下载验证 .....	270
第三十一章 WiFi 热点实验 .....	271
31.1 硬件设计 .....	271
1. 例程功能 .....	271
2. 硬件资源 .....	271
3. 原理图 .....	271
31.2 软件设计 .....	271
31.2.1 程序流程图 .....	271
31.2.2 程序解析 .....	272
31.3 下载验证 .....	274
第三十二章 UDP 实验 .....	275
32.1 Socket 编程 UDP 连接流程 .....	275
32.2 硬件设计 .....	275
1. 例程功能 .....	275
2. 硬件资源 .....	275
3. 原理图 .....	275
32.3 软件设计 .....	276
32.3.1 程序流程图 .....	276
32.3.2 程序解析 .....	276
32.4 下载验证 .....	278

第三十三章 TCPClient 实验.....	279
33.1 Socket 编程 TCPClient 连接流程 .....	279
33.2 硬件设计.....	279
1. 例程功能.....	279
2. 硬件资源.....	279
3. 原理图.....	280
33.3 软件设计.....	280
33.3.1 程序流程图.....	280
33.3.2 程序解析.....	280
33.4 下载验证.....	283
第三十四章 TCPServer 实验 .....	284
34.1 Socket 编程 TCPServer 连接流程.....	284
34.2 硬件设计.....	284
1. 例程功能.....	284
2. 硬件资源.....	284
3. 原理图.....	285
34.3 软件设计.....	285
34.3.1 程序流程图.....	285
34.3.2 程序解析.....	285
34.4 下载验证.....	288
第三十五章 WiFi 一键配网.....	289
35.1 主流 WIFI 配网方式简介 .....	289
35.2 硬件设计.....	290
1. 例程功能.....	290
2. 硬件资源.....	290
3. 原理图.....	290
35.3 软件设计.....	290
35.3.1 程序流程图.....	290
35.3.2 程序解析.....	290
35.4 下载验证.....	294

## 前言

随着物联网（IoT）的迅猛发展，嵌入式系统开发成为了连接物理世界与数字世界的桥梁。在这个变革的时代，乐鑫（Espressif）的 ESP 系列微控制器，凭借其低功耗、高性能和广泛的应用领域，已经成为物联网领域中的明星产品。为了帮助开发者更好地掌握 ESP 系列微控制器的开发技术，我们编写了这本 ESP-IDF 编程教程。

ESP-IDF，即 Espressif IoT Development Framework，是乐鑫为 ESP 系列微控制器提供的官方开发框架。它提供了丰富的 API 和库，使得开发者能够更加便捷地进行硬件抽象、驱动开发、网络通信和应用程序设计。同时，ESP-IDF 还具备高度的可配置性和可扩展性，能够满足不同应用场景的需求。

本教程旨在帮助读者从零开始掌握 ESP-IDF 开发框架的使用技巧。通过本教程的学习，读者将能够了解 ESP 系列微控制器的基本架构和特性，熟悉 ESP-IDF 开发环境的搭建和配置，掌握常用的 API 和库的使用方法，以及学会如何开发和调试基于 ESP-IDF 的物联网应用程序。

在编写本教程的过程中，我们力求内容详实、结构清晰、实例丰富，使读者能够在实践中逐步掌握 ESP-IDF 开发技术。同时，我们也充分考虑了初学者的学习需求，通过大量的图表和实例来辅助说明，使学习过程更加直观和易于理解。

## 内容简介

为了更好地发挥 ESP32-S3 芯片的性能和功能，我们特别编写了这本教程。教程以 C 语言为编程语言，旨在帮助读者快速掌握 ESP32-S3 的开发技巧。C 语言以其简单易学、高效灵活的特点，在物联网设备开发和调试过程中具有显著优势。通过 C 语言编程，读者不仅能够深入了解 ESP32-S3 的工作原理，还能够实现多种有趣且实用的物联网应用。

本教程特别针对初学者设计，分为入门篇、基础篇和高级篇三个部分。

### 第一部分：入门篇

入门篇将引导读者了解 C 语言和 ESP32-S3 的基础知识，包括安装和配置、基本语法以及模块和库等。

### 第二部分：基础篇

基础篇则深入介绍 ESP32-S3 的硬件接口和应用开发基础，包括 GPIO 接口、I2C 接口、SPI 接口等。

### 第三部分：高级篇

高级篇则将带领读者探索基于 C 语言的综合应用开发，如人脸识别、猫脸识别、颜色识别等高级应用。

我们相信，通过本教程的学习，读者将能够全面掌握 ESP32-S3 的开发技术，为物联网领域的创新和发展贡献自己的力量。让我们携手共进，开启物联网编程之旅！

# 注意事项

在深入学习例程之前，有几个与之相关的关键问题是我们必须理解的。这些问题通常也是读者在后续章节中可能会遭遇的编译错误。为此，作者特意设置了一个特殊的章节，专门介绍学习例程前的注意事项，以帮助读者更好地掌握相关知识。

## 1, dir\_sdi()函数未找到

由于 ESP-IDF 的 ff.c 文件把该函数设置为 static 类型，所以需要读者打开 ff.c 文件把该函数设置为动态函数，然后在 ff.h 文件声明以提供其他文件调用。下面是 VS Code 提示错误信息和解决方案：

VS Code 提示错误：

```
error: implicit declaration of function 'dir_sdi' [-Werror=implicit-function-declaration]
```

解决方案：

在 VS Code 找到这个函数按 F12 进去 ff.c 文件，或者在 Espressif\frameworks\esp-idf-v5.1.2\components\fatfs\src\路径下找到 ff.c 文件，然后把“static FRESULT dir\_sdi”函数修改为“FRESULT dir\_sdi”，最后在 ff.h 文件中声明此函数，如下图所示：

```
338 FRESULT dir_sdi(FF_DIR* dp, DWORD ofs);
```

作者在 ff.h 文件中的第 338 行声明了此函数，以提供外部文件调用。注意：此时工程最好先擦除 flash(垃圾桶图标)，再去编译工程，不然可能会发生某些错误。

## 2, SD 卡读取问题

作者在 DNESP32S3M 最小系统板上测试闪迪 16G\32G、雷克沙 32G 和金士顿 32G 等多款 TF 卡，都是可以进行读写操作。至于 32G 以上或者其他类型的 TF 卡支不支持（作者也不知道），需要读者查找相关资料或者根据乐鑫官方要求修改相应的代码。

## 3, 中文路径下编译错误

例程仅支持全英文路径下编译。

## 4, 已生成 build 文件的例程，拷贝到其他路径会出现编译错误。

首先先擦除 flash，然后再编译工程。

## 5, 关于调试问题

有时在 VS Code 中成功调试完一个工程后，再次尝试调试可能会出现不成功的情况。为解决此问题，你可以尝试关闭工程并重新打开，清除 Flash 内容，并重新编译例程。这样做通常能够恢复正常调试功能。

# 第一篇 基础篇

万事开头难，但只要打好基础，后续的学习将会事半功倍。本篇将为大家详细解析 ESP32-S3 的基础知识，涵盖环境搭建、入门知识、新建工程、ESP32-IDF 介绍、启动过程分析、时钟系统以及 components 文件夹等多个方面。深入理解并掌握这些知识，将为后续的例程学习奠定坚实基础，极大提升学习效率。

特别提醒初学者，务必认真学习并充分理解这些基础知识，边学边做，不遗漏任何细节。一遍没掌握也不要气馁，多重复几遍，确保每个知识点都能熟练掌握。这将为您在 ESP32-S3 的学习旅程中提供坚实的支撑。

本篇将分为如下章节：

- 1, 本书学习方法
- 2, 常用的 C 语言知识点
- 3, ESP32-S3 基础知识
- 4, 认识 ESP-IDF
- 5, 搭建开发环境
- 6, 新建基础工程
- 7, 分区表
- 8, MENUCONFIG 菜单配置解析
- 9, IDF 组件注册表
- 10, 入门篇的注意事项

# 第一章 本书学习方法

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

本章将分为如下几个小节：

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

## 1.1 本书学习顺序

为了让大家更好的学习和使用本书，我们做了以下几点考虑：

- 1，坚持循序渐进的思路编写，从基础到入门，从简单到复杂。
- 2，将知识进行分类介绍，简化学习过程，包括：基础篇、入门篇、提高篇。
- 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，开发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开发者，就可以跳跃式地学习，学习效率，当然了，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 技术手册》

前者是乐鑫官方针对 S3 系列 ESP32-S3 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-S3 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、外设描述等的支持文档。

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

## 1.3 本书编写规范

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

基础篇，共 9 章，主要是一些基础知识介绍，包括开发环境搭建、新建工程、ESP32-IDF 介绍和 menuconfig 介绍等，这些章节在结构上没有共性，但是互相有关联，有一个集成的关系在里面，即：必须先学了前面的知识，才好学习后面的知识点。

入门篇和提高篇，共五十五章，详细介绍了 ESP32-S3 每一个外设的使用方法及驱动代码，

并且还介绍了一些非常实用的程序代码（纯软件例程），如：内存管理、文件系统读写、SD 卡读取、图片解码、音频解码、视频解码、USB、Wi-Fi、AI 以及人脸识别等。这两篇内容占了本书的绝大部分篇幅，而且这些章节在结构上都比较有共性，一般分为 4 个部分，如下：

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

外设功能介绍，简单介绍具体章节所要用到的外设功能、框图和寄存器等，让大家对所用外设的功能有一个基本了解，方便后面的程序设计。

硬件设计，包括具体章节的实验具体功能说明、所用到的硬件资源及原理图连接方式，从而知道要做什么？需要用到哪些 IO 口？是怎么接线的？方便程序设计的时候编写驱动代码。

程序设计，一般包括：驱动介绍、配置步骤、程序流程图、关键代码分析、main 函数讲解等三部分。一点点介绍程序代码是怎么来的，注意事项等，从而学会整个代码。

下载验证，属于实践环节，在完成程序设计后，教大家如何下载并验证我们的例程是否正确？完成一个闭环过程。

## 1.4 本书代码规范

为了方便大家编写高质量代码，我们对本书的代码风格进行了统一，详细的代码规范说明文档，见光盘：A 盘→1，入门资料→《嵌入式单片机 C 代码规范与风格.pdf》，初学者务必好好学习一下这个文档。

总结几个规范的关键点：

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

## 1.5 例程资源说明

ESP32-S3 最小系统板的配套资料中，除了《00\_basic》之外，还提供了 37 个标准例程。这些例程都是基于 C 语言和 ESP32-IDF 进行编写的。这些例程大部分是原创的，并附有详细的注释，代码风格统一，内容循序渐进，非常适合初学者入门。

ESP32-S3 最小系统板配套的例程如下表所示：

编号	实验名字	编号	实验名字
基础应用			
00	00_basic	19	19_ds18b20
01	01_led	20	20_dht11
02	02_key	21	21_rng
03	03_exit	22	22_qma6100p
04	04_uart	23	23_rgb
05	05_esp_timer	24	24_touch
06	06_gp_timer	25-1	25_1_camera
07	07_wdt	25-2	25_2_camera_photograph
08-1	08-1_sw_pwm	26	26_sd

08-2	08-2_hw_pwm	27	27_spiffs
09	09_iic_exio	28	28_chinese_display
10	10_iic_eeprom	29	29_pictures
11	11_oled	30	30_music
12	12_spilcd	31	31_recoding
13	13 rtc	32	32_videoplayer
14	14_adc	33	33_usb_uart
15	15_ap3216c	34	34_usb_flash_u
16	16_infrared_reception	35	35_usb_sd_u
17	17_infrared_transmission	36	36_bootloader
18	18_internal_Temperature		
WiFi 例程			
01	01_WiFi_SCAN	05	05_WiFi_UDP
02	02_WiFi_STA	06	06_WiFi_TCPCClient
03	03_WiFi_AP	07	07_WiFi_TCPServer
04	04_WiFi_SmartConfig		

表 1.5.1 DNESP32S3M 最小系统板基础例程表

从上表可以看出，正点原子 DNESP32S3M 最小系统板的例程基本上涵盖了 ESP32-S3 芯片的多个内部资源使用，并且外扩展了很多有价值的例程，比如：基础入门实验、物联网等。

而且从上表可以看出，例程安排是循序渐进的，首先从最基础的新建一个工程开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握。所以，正点原子 ESP32-S3 最小系统板是非常适合初学者的。当然，对于想深入了解 ESP32-S3 内部资源的朋友，正点原子 DNESP32S3M 最小系统板也绝对是一个不错的选择。

## 1.6 学习资料查找

### 1. ESP-IDF 编程指南

ESP-IDF 编程指南包含了 ESP 官方推出的几款芯片的编程指南，这里我们选择 ESP32-S 系列的 ESP32-S3 即可，如下图所示：



图 1.6.1 ESP32-S3 相关资料

ESP-IDF 编程指南包含了 IDF 的快速入门、API 参考、H/W 硬件参考以及 API 指南等，我们在开发过程中主要用到 API 参考，如下图示：



图 1.6.2 ESP32-S3 相关资料

该教程是针对 ESP32-S3 芯片及 IDF 开发的详细指南，包括硬件准备、IDF 编程及使用等方面。通过该教程的学习和实践，读者可以掌握使用 IDF 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。

### 2, ESP 官方论坛

ESP 官方论坛网址，读者可在该论坛上获取相应的 ESP 资料。为了方便开发者学习，ESP 官方论坛分别提供了中、英文两种语言的论坛，如下图所示：

The screenshot shows the ESPRESSIF forum interface. At the top, there's a red header with the ESPRESSIF logo and a white dove icon. Below the header, there are navigation links for 'home', 'quick links', 'forum', and other system icons. The main content area has two sections: 'FORUM' and 'WHO IS ONLINE'. The 'FORUM' section displays two threads: 'English Forum' (Subforum: General Discussion) with 28855 topics and 104133 posts, and 'Chinese Forum 中文社区' with 4455 topics and 12991 posts. The 'WHO IS ONLINE' section shows a list of users. On the right, there's a 'LOGIN' form with fields for email and password, and options for 'I forgot my password', 'Remember me', and 'Hide my online status this session'. There are 'Login' and 'Register' buttons at the bottom of the form.

图 1.6.3 ESP 官方论坛

读者可以根据您的需求进行选择使用哪一种语言的社区来进行学习。

### 3, 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

在文档中心下面，我们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地址。

#### 4, 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。

#### 5, 博客和教程网站

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

#### 6, 视频教程

在 B 站等视频平台上搜索与 ESP32-S3 和 IDF 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。

#### 7, 在线课程和教育资源

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

总之，通过以上方法，您可以找到大量与 ESP32-S3 和 IDF 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。

在学习过程中，我们难免会遇到一些问题，有任何问题，大家都可以先去开源电子网搜索一下，基本上你能遇到的问题，我们论坛都有人问过了，所以可以很方便的找到一些参考解决方法。如果实在找不到，你也可以在论坛提问，每天原子哥都会在上面给大家做解答。

### 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。如果 C 语言基础不够扎实，建议先学习 C 语言基础，以便更好地理解和掌握 ESP32-S3 的相关知识。

## 第二章 常用的 C 语言知识点

### 5.1 位操作

位操作是直接在整数的二进制位上进行操作的一种技术。C 语言提供了多种位操作符，这些操作符允许程序员对整数的二进制位进行读取、设置、清除或翻转。位操作在处理底层硬件、优化代码、处理二进制数据等方面非常有用。C 语言支持如下 6 种位操作：

运算符	含义	运算符	含义
&	按位与	~	按位取反
	按位或	<<	左移
^	按位异或	>>	右移

表 5.1.1 六种位操作

以下是 C 语言中的一些位操作符及其描述：

1, 按位与(&): 如果两个相应的二进制位都为 1，则结果为 1，否则为 0。

```
int a = 60; /* 60 = 0011 1100 */
int b = 13; /* 13 = 0000 1101 */
int c = a & b; /* c = 0000 1100 */
```

2, 按位或(|): 如果两个相应的二进制位中至少有一个为 1，则结果为 1，否则为 0。

```
int a = 60; /* 60 = 0011 1100 */
int b = 13; /* 13 = 0000 1101 */
int c = a | b; /* c = 0011 1101 */
```

3, 按位异或(^): 如果两个相应的二进制位不同，则结果为 1，否则为 0。

```
int a = 60; /* 60 = 0011 1100 */
int b = 13; /* 13 = 0000 1101 */
int c = a ^ b; /* c = 0011 0001 */
```

4, 按位取反(~): 将整数的二进制位翻转。

```
int a = 60; /* 60 = 0011 1100 */
int b = ~a; /* b = 1100 0011 */
```

5, 左移(<<): 将整数的二进制位向左移动指定的位数，右侧用 0 填充。

```
int a = 60; /* 60 = 0011 1100 */
int b = a << 2; /* b = 0111 1000 = 240 */
```

6, 右移(>>): 将整数的二进制位向右移动指定的位数，左侧根据整数的符号位填充 0 或 1。

```
int a = 60; /* 60 = 0011 1100 */
int b = a >> 2; /* b = 0000 1111 = 15 */
```

使用位操作可以执行许多有用的任务，例如检查一个数的特定位是否为 1，设置或清除一个数的特定位，快速地进行整数乘法和除法，等等。然而，使用位操作时需要特别小心，因为错误的操作可能会导致不可预测的结果。

### 5.2 define 宏定义

在 C 语言中，#define 是预处理指令的一部分，用于定义宏。宏可以是一个简单的常量、一个带有参数的表达式或是一个代码块。使用宏可以在编译前替换代码中的特定部分，从而实现代码的重用、简化和提高可读性。

#### 1. 无参数宏定义

无参数宏定义是最简单的形式，它只是一个标识符和一个值的组合。

```
#define PI 3.14159
```

在代码中，每次出现 PI，预处理器都会将其替换为 3.14159。

#### 2. 带参数宏定义

带参数宏定义允许宏接受一个或多个参数，并返回一个表达式。

```
#define SQUARE(x) ((x) * (x))
```

这里，SQUARE 是一个宏，它接受一个参数 x，并返回 x 的平方。注意，宏中的参数应该用括号括起来，以避免因为运算符优先级导致的问题。

#### 3. 宏定义的代码块

虽然不常见，但宏定义还可以包含多个语句。这通常用于实现复杂的操作或内联函数。

```
#define SWAP(a, b) do { \
    typeof(a) temp = a; \
    a = b; \
    b = temp; \
} while (0)
```

SWAP 宏交换两个变量的值。由于宏是文本替换，因此需要用 do ... while(0) 来确保宏内部的多条语句被当作一个单独的语句块处理。

#### 4. #undef

可以使用#undef 指令来取消一个宏的定义。

```
#define PI 3.14159
/* ... 使用 PI... */
#undef PI
/* 之后 PI 不再是一个宏 */
```

#### 5. 宏定义的注意事项

- 宏定义只是简单的文本替换，没有类型检查或作用域限制。
- 宏可能会因为参数的运算符优先级导致预期之外的行为，所以使用时要特别小心。
- 宏定义可能会导致代码膨胀，因为每个宏的使用都会导致相同代码的重复插入。
- 避免在宏中使用复杂的表达式或逻辑，因为这会增加代码阅读和维护的难度。
- 使用宏时要谨慎，以避免出现意外的副作用或难以调试的错误。

### 5.3 ifdef 条件编译

在 C 语言中，#ifdef 是预处理指令的一部分，用于条件编译。条件编译允许你在编译时根据特定的条件来决定是否包含某些代码段。这对于编写跨平台或可配置的代码非常有用。

#ifdef 检查是否定义了一个宏（使用#define 指令）。如果宏已经定义，那么#endif 和紧随其后的#endif 之间的代码将被包含在编译中。如果宏没有定义，那么这部分代码将被忽略。

下面是一个简单的例子：

```
#define FEATURE_A

int main() {
    #ifdef FEATURE_A
        /* 这段代码将被编译，因为 FEATURE_A 已经定义 */
        printf("Feature A is enabled.\n");
    #else
        /* 这段代码不会被编译 */
        printf("Feature A is disabled.\n");
    #endif

    #ifndef FEATURE_B
        /* 这段代码将被编译，因为 FEATURE_B 没有被定义 */
        printf("Feature B is not defined.\n");
    #else
        /* 这段代码不会被编译 */
        printf("Feature B is defined.\n");
    #endif

    return 0;
}
```

在这个例子中，FEATURE\_A 被定义了，所以#ifdef FEATURE\_A 和#endif 之间的代码会被编译。而 FEATURE\_B 没有被定义，所以#ifndef FEATURE\_B 和#endif 之间的代码会被编译。

除了#endif，还有其他的条件编译指令：

- #ifndef：如果宏没有定义，则包含代码。
- #if：用于检查宏是否定义以及它的值是否为真（非零）。
- #elif：与#if 和#else 结合使用，用于检查多个条件。
- #else：如果前面的#if 或#ifndef 条件不满足，则包含代码。
- #endif：标记条件编译块的结束。

这些指令通常在源代码的顶部使用，以根据特定的配置或平台条件包含或排除代码段。例

如，你可能想要在不同的操作系统上使用不同的系统调用，或者在调试和发布版本中包含或排除调试代码。

## 5.4 extern 外部申明

在 C 语言中，extern 关键字用于声明一个变量或函数，而不是定义它。extern 告诉编译器，变量的定义或函数的实现在其他地方，可能是在另一个源文件中。这允许程序的不同部分共享同一个变量或函数，而无需在每个文件中都重复定义它。

### 1. 变量外部声明

当你想在一个源文件中使用另一个源文件中定义的变量时，你需要使用 extern 来声明这个变量。例如，假设你有一个名为 variables.c 的文件，它定义了一个名为 globalVar 的全局变量：

```
/* variables.c */
#include <stdio.h>

int globalVar = 100; /* 定义全局变量 */
```

现在，如果你想在另一个源文件 main.c 中使用这个变量，你可以这样声明它：

```
/* main.c */
#include <stdio.h>

extern int globalVar; /* 外部声明全局变量 */

int main()
{
    printf("The value of globalVar is: %d\n", globalVar);
    return 0;
}
```

在这个例子中，main.c 并没有定义 globalVar，而是使用了 extern 关键字来声明它。这告诉编译器，globalVar 的定义存在于其他地方，并且在链接阶段，链接器会找到这个定义。

### 2. 函数外部声明

同样，当你想在一个源文件中调用另一个源文件中定义的函数时，你需要使用 extern 来声明这个函数。例如，假设你有一个名为 functions.c 的文件，它定义了一个名为 myFunction 的函数：

```
/* functions.c */
#include <stdio.h>

void myFunction() {
    printf("This is my function!\n");
}
```

然后，在 main.c 中，你可以这样声明并使用这个函数：

```
/* main.c */
#include <stdio.h>

extern void myFunction(); /* 外部声明函数 */

int main()
{
    myFunction(); /* 调用函数 */
    return 0;
}
```

在这个例子中，main.c 中并没有定义 myFunction，而是通过 extern 关键字进行了声明。在编译和链接阶段，链接器会找到 myFunction 的定义，并将其与 main.c 中的调用关联起来。

### 3. 注意事项

- extern 只能用于声明变量或函数，不能用于定义。定义会分配内存空间，而声明不会。
- 当你在一个源文件中定义了一个变量或函数，并想在另一个源文件中使用它时，你需要在第二个源文件中使用 extern 进行声明。
- 外部声明必须在使用变量或函数之前进行。
- 在链接阶段，链接器会查找所有 extern 声明的定义，如果找不到，就会出现链接错误。
- 如果多个源文件定义了同一个 extern 变量，那么链接器会将其视为错误，因为同一个变

量只能有一个定义。但是，多个源文件可以包含同一个 `extern` 函数的声明，因为这个函数可能在不同的地方有不同的实现。

## 5.5 `typedef` 类型别名

`typedef` 用于为现有类型创建一个新的名字，或称为类型别名，用来简化变量的定义。例如在编写程序时经常使用到的 `uint8_t`、`uint16_t` 和 `uint32_t` 等都是由 `typedef` 定义的类型别名，其定义如下：

```
typedef unsigned char      uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int       uint32_t;
```

这么一来就可以在编写程序代码的时候使用 `uint8_t` 等代替 `unsigned char` 等，极大地提高了代码的可读性可编写代码的效率。

## 5.6 `struct` 结构体

`struct` 用于定义结构体，结构体就是一堆变量的集合，结构体中的成员变量的作用一般都是相互关联的，定义结构体的形式如下：

```
struct 结构体名
```

```
{
```

```
    成员变量 1 的定义;
```

```
    成员变量 2 的定义;
```

```
    ....
```

```
};
```

例如：

```
struct lcd_device_struct
{
    uint16_t width;
    uint16_t height;
};
```

如上举例的结构体定义，一堆描述 LCD 屏幕的变量的集合，其中包含了 LCD 屏幕的宽度和高度。

结构体变量的定义如下：

```
struct lcd_device_struct lcd_device;
```

如上，就定义了一个名为 `lcd_device` 的结构体变量，那么怎么访问这个结构体变量中的成员变量呢？如下：

```
lcd_device.width = 240;
printf("LCD Height: %d\n", lcd_device.height);
```

如上就展示了结构体变量中成员变量的访问操作。

任何时候，我们只需要修改结构体成员变量，往结构体中间加入新的成员变量，而不需要修改函数定义就可以达到修改入口参数同样的目的了。这样的好处是不用修改任何函数定义就可以达到增加变量的目的。

理解了结构体在这个例子中间的作用吗？在以后的开发过程中，如果你的变量定义过多，如果某几个变量是用来描述某一个对象，你可以考虑将这些变量定义在结构体中，这样也许可以提高你的代码的可读性。

使用结构体组合参数，可以提高代码的可读性，不会觉得变量定义混乱。当然结构体的作用就远远不止这个了，同时，VSCode 中用结构体来定义外设也不仅仅只是这个作用，这里我们只是举一个例子，通过最常用的场景，让大家理解结构体的一个作用而已。后面一节我们还会讲解结构体的一些其他知识。

## 5.7 指针

指针是一个值指向地址的变量（或常量），其本质是指向一个地址，从而可以访问一片内存区域。在编写 ESP32 代码的时候，或多或少都要用到指针，它可以使不同代码共享同一片内存数据，也可以用作复杂的链接性的数据结构的构建，比如链表，链式二叉树等，而且，有些地方必须使用指针才能实现，比如内存管理等。

申明指针我们一般以 p 开头，如：

```
char * p_str = "This is a test!";
```

如上，就定义一个名为 p\_str 的指针变量，并将 p\_str 指针指向了字符串 “This is a string!” 保存在内存中首地址，对于 ESP32 来说，此时 p\_str 的值就是一个 32 位的数，这个数就是一个内存地址，这个内存地址就是上述字符串保存在内存中的首地址。

通过 p\_str 指针就可以访问到字符串 “This is a string!”，那具体是如何访问的呢？前面说 p\_str 保存的是一个内存地址，那么就可以通过这个内存地址去内存中读取数据，通过\*p\_str 就可以访问地址为 p\_str 的内存数据，\*(p\_str + 1)可以访问下一个内存地址中的数据。

知道了如何访问内存中的数据，但是读取到的数据要如何解析呢？这就有 p\_str 指针的类型决定了。在这个例子中 p\_str 是一个 char 类型的指针，那么访问\*p\_str 就是访问地址为 p\_str，大小为 sizeof(char)（一般为一个字节）的一段内存数据，在这个例子中就可以读取到字符 “T”，读取\*(p\_str + 1)就是 “h”，以此类推。

为了更加直观的演示，我们试着编写如下代码并观察输出结果的变化：

```
/** @brief 程序入口
 * @param 无
 * @retval 无
 */
void app_main(void)
{
    esp_err_t ret;
    uint8_t temp = 0x88; /* 定义变量 temp */
    uint8_t *p_num = &temp; /* 定义指针 p_num，指向 temp 的地址 */

    ret = nvs_flash_init(); /* 初始化 NVS */
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    printf("temp:0X%X\r\n", temp); /* 打印 temp 的值 */
    printf("*p_num:0X%X\r\n", *p_num); /* 打印*p_num 的值 */
    printf("p_num:0X%x\r\n", (unsigned int)(long)p_num); /* 打印 p_num 的值 */
    printf("&p_num:0X%x\r\n", (unsigned int)(long)&p_num); /* 打印&p_num 的值 */
}
```

此代码的输出结果为：

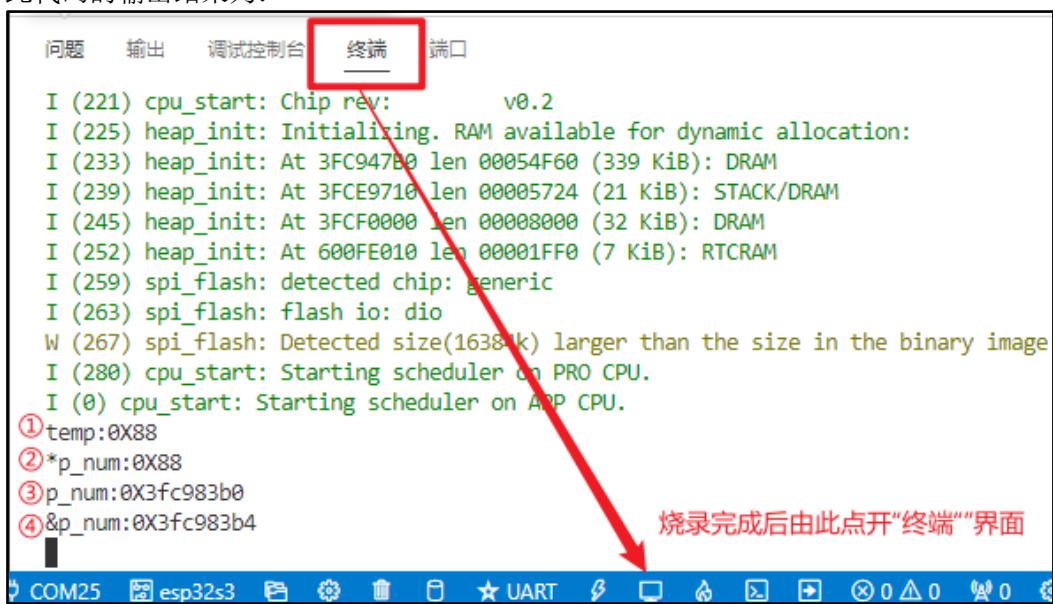


图 5.7.1 终端输出结果

①: p\_num: 是 uint8\_t 类型指针, 指向 temp 变量的地址, 其值等于 temp 变量的地址。

②: \*p\_num: 取 p\_num 指向的地址所存储的值, 即 temp 的值。

③: p\_num: 指针的地址。

④: &p\_num: 取 p\_num 指针的地址, 即指针自身的地址。

以上, 就是指针的简单使用和基本概念说明, 指针的详细知识和使用范例大家可以百度学习, 网上有非常多的资料可供参考。指针是 C 语言的精髓, 在后续的代码中我们将会大量用到各种指针, 大家务必好好学习和了解指针的使用。

## 第三章 ESP32-S3 基础知识

在本章中，我们将深入探索 ESP32-S3 这款备受瞩目的微控制器。我们将详细阐述其定义、核心资源、功能应用，以及如何选择适合您项目的 ESP32-S3 型号。通过本章的学习，您将全面了解 ESP32-S3，为您的物联网项目选择合适的硬件平台奠定坚实基础。

本章分为如下几个小节：

- 3.1 为什么选择 ESP32-S3
- 3.2 初识 ESP32-S3
- 3.3 ESP32-S3 资源简介
- 3.4 S3 系列型号对比
- 3.5 ESP32-S3 功能概述
- 3.6 ESP32-S3 启动例程

### 3.1 为什么选择 ESP32-S3

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

下面，作者比较一下乐鑫推出的芯片有哪些特点：

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

表 3.1.1 乐鑫各系列 MCU 硬件区别

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

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

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

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

4，在神经网络加速方面，只有 S 系列支持神经网络加速功能。这意味着选择 S 系列可以更

好地满足深度学习、图像识别等应用场景的需求。对于需要加速神经网络运算的应用场景，如智能家居控制、智能安防等，S 系列可能更合适。

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

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

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

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

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

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

- ① 工具选择**: Shows the main navigation bar with tabs for "产品选型" and "产品对比".
- ② 功能筛选**: A sidebar on the left containing dropdown menus for filtering products based on various parameters like Model, Wi-Fi, Bluetooth, Temperature, Category, Series, Status, Dual-core, Antenna, Packaging, Storage, and Peripherals.
- ③ 筛选结果的选择**: A preview section for the selected product, showing the "ESP32-S3-BOX-3 新一代 AIoT 开发工具" and its details.
- ④ 筛选结果**: A table showing the filtered results, with two entries for the ESP32-S3 chip/module.

	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 3.1.1 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。

## 3.2 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。下图为芯片的功能框图。

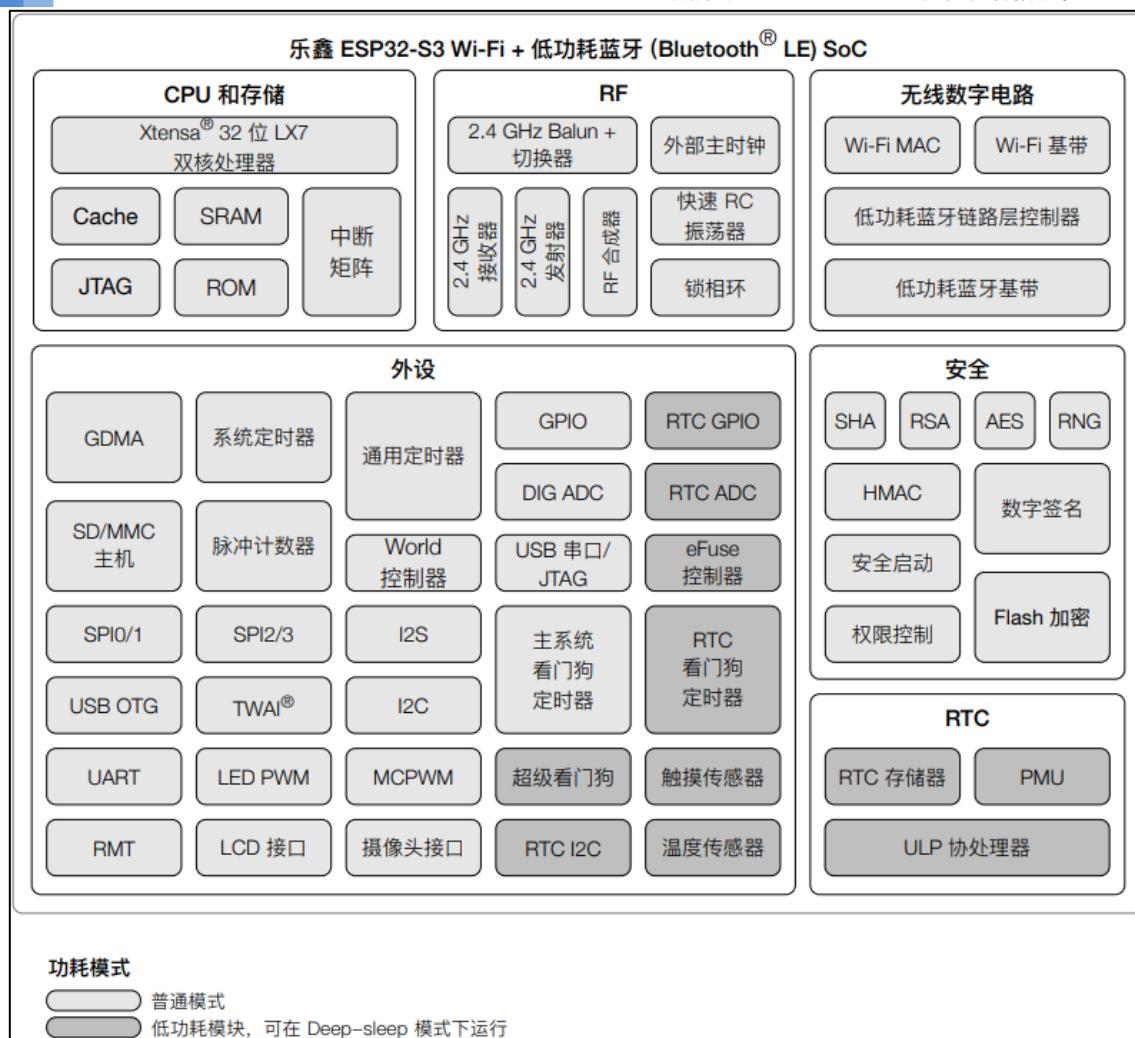


图 3.2.1 ESP32-S3 功能框图

结合《esp32-s3\_datasheet\_cn.pdf》数据手册和上图的内容，简单归纳 5 个内容。

1，架构和性能：ESP32-S3 采用 Xtensa® LX7 CPU，这是一个哈佛结构的双核系统。它具有独立的指令总线和数据总线，所有的内部存储器、外部存储器以及外设都分布在这两条总线上。这种架构使得 CPU 可以同时读取指令和数据，从而提高了处理速度。

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

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

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

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

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

### 3.3 ESP32-S3 资源简介

下面来看看 ESP32-S3 具体的内部资源，如下表所示。

ESP32-S3 资源						
内核	Xtensa® LX7 CPU	系统定时器	1	UART	3	

主频	240MHz	定时器组	2	RNG	1
ROM	384KB	LEDC	1	I2C	2
SRAM	512KB	RMT	1	I2S	2
编程 IO	45GPIO	PCNT	1	SPI	4 (0、1 禁用)
工作电压	3.3	TWAI	1	RGB	1
Wi-Fi/BLE	1/1	USB OTG	1	SD/MMC	1

表 4.3.1 ESP32-S3 内部资源表

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

关于 ESP32-S3 内部资源的详细介绍，请大家参考“光盘→A 盘→7，硬件资料→2，芯片资料→esp32-s3\_technical\_reference\_manual\_cn.pdf”，该文档即《ESP32-S3 的技术手册》，里面有 ESP32-S3 详细的资源说明和相关性能参数。

### 3.4 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。

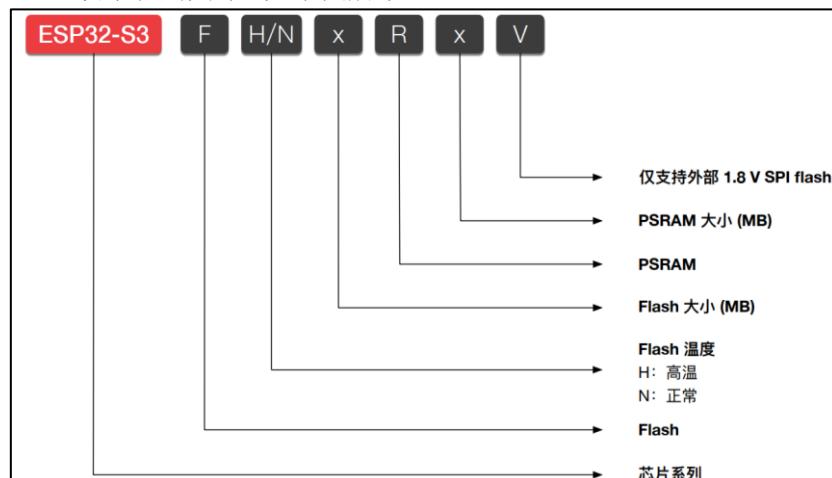


图 3.4.1 ESP32-S3 系列芯片命名规则

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 3.4.2 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。

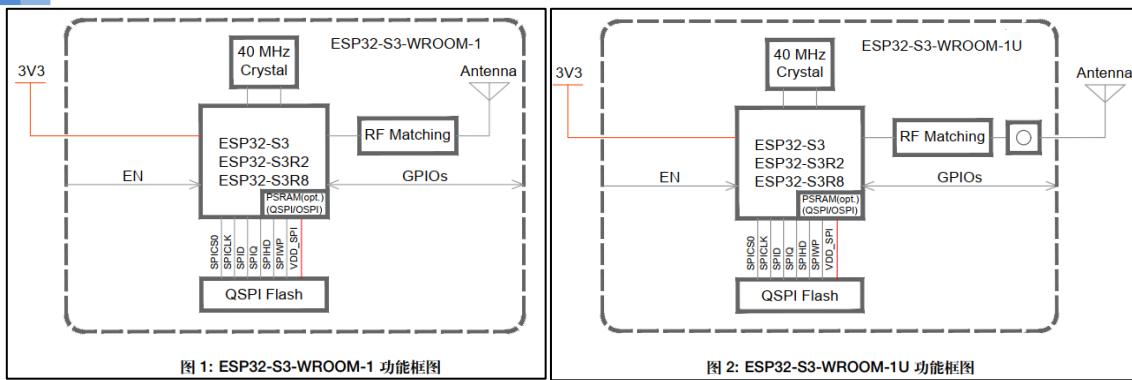


图 3.4.3 ESP32-S3-WROOM-1 和 ESP32-S3-WROOM-1U 的功能框图

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 3.4.1 通用型模组的命名

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

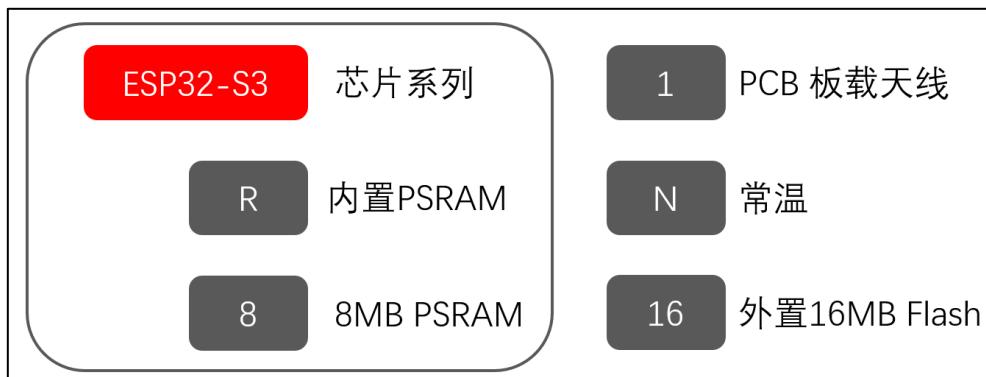


图 3.4.3 模组的命令规则

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

## 3.5 ESP32-S3 功能概述

### 3.5.1 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。

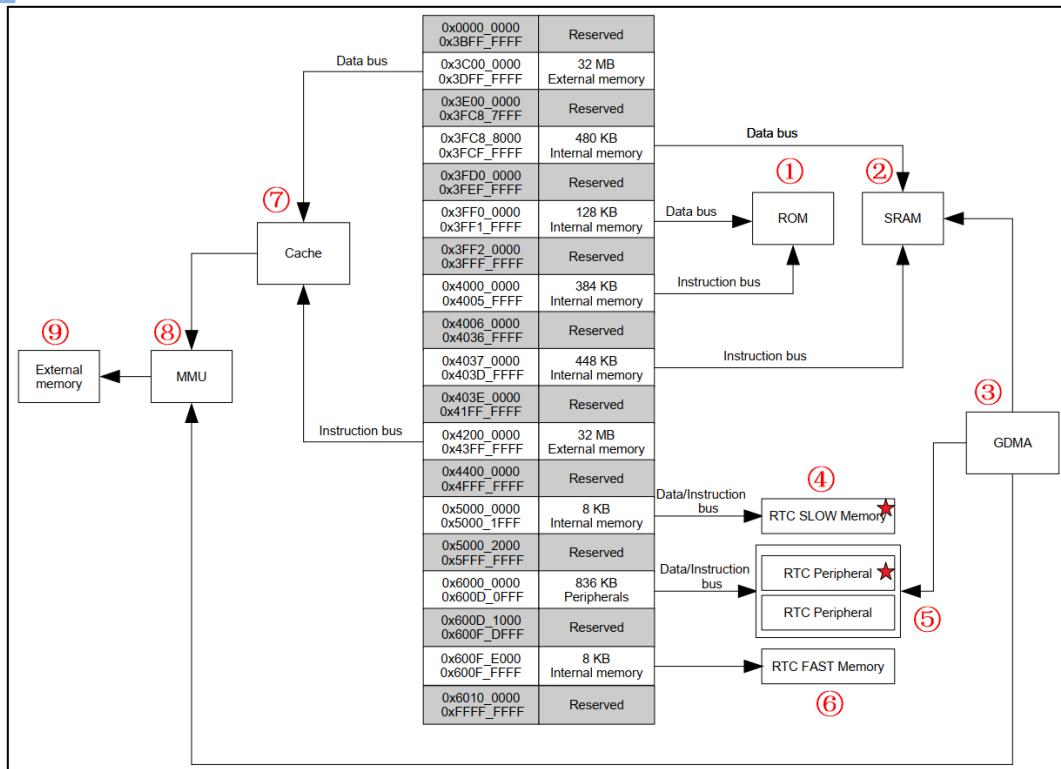


图 3.5.1 系统结构与地址映射结构

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。

### 3.5.1.1 内部存储器

图 3.5.1 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)。

### 3.5.1.2 外部存储器

图 3.5.1 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。

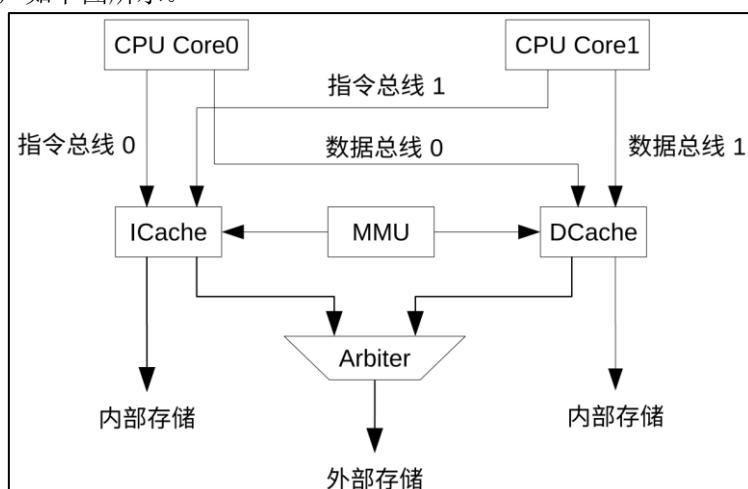


图 3.5.1.2.1 Cache 系统框图

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 3.5.1.2.1 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）。

### 3.5.1.3 模块/外设

图 3.5.1 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 3.5.1.3.1 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）。

### 3.5.1.4 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 3.3.1 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。

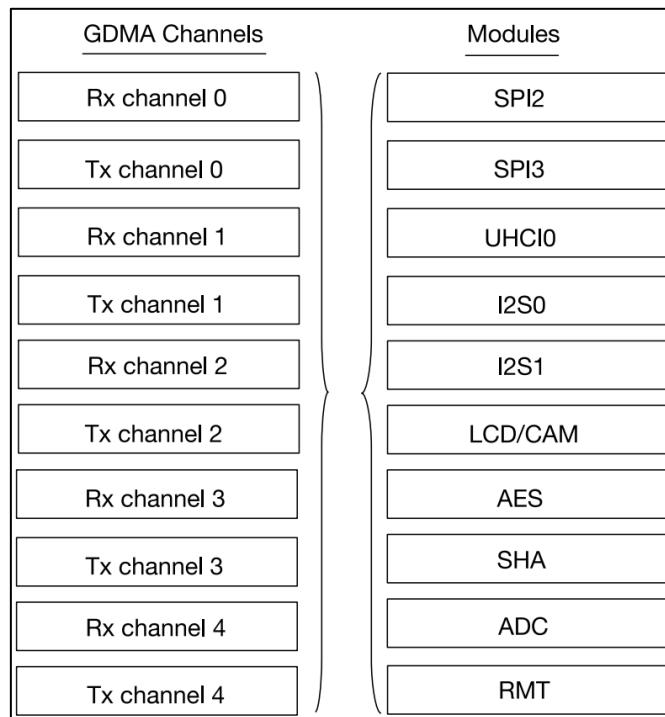


图 3.5.1.4.1 具有 GDMA 功能的模块和 GDMA 通道示意图

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。

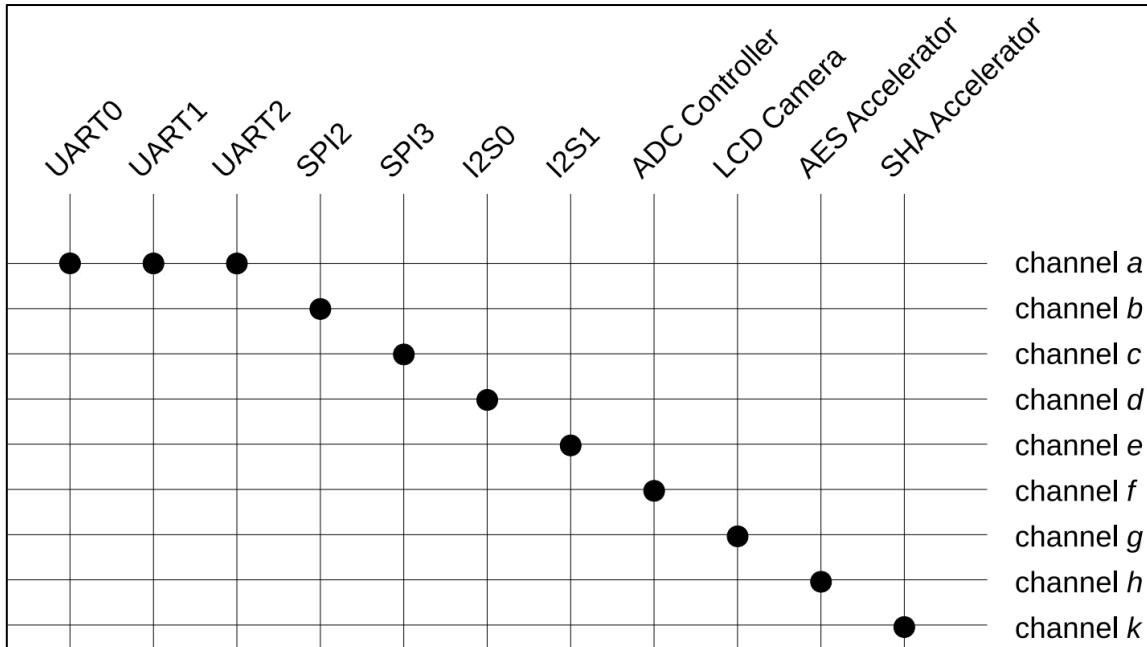


图 3.5.1.4.2 具有 GDMA 功能的外设

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。

### 3.5.2 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 <sup>1</sup>	CS#		CS#		CS#		CS#	
28	SPICS1 <sup>2</sup>		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 3.5.2.1 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。

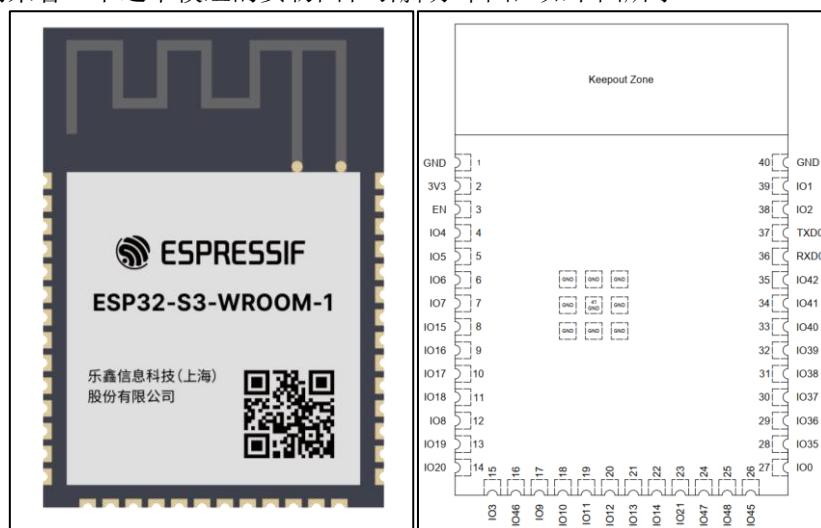


图 3.5.2.2 ESP32-S3-WROOM-1 实物图和引脚分布图

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 3.5.2.1 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。

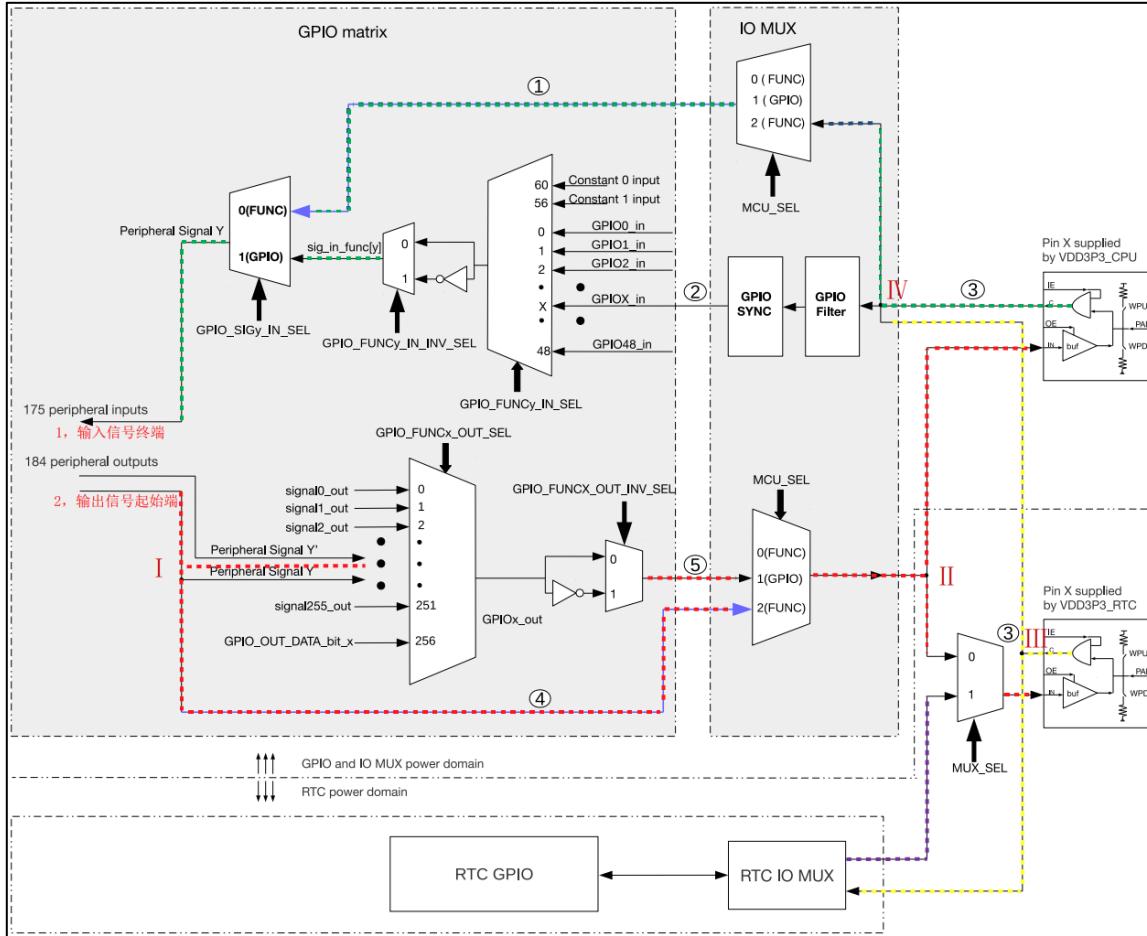


图 3.5.2.3 IO MUX、RTC IO MUX 和 GPIO 交换矩阵结构框图

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。

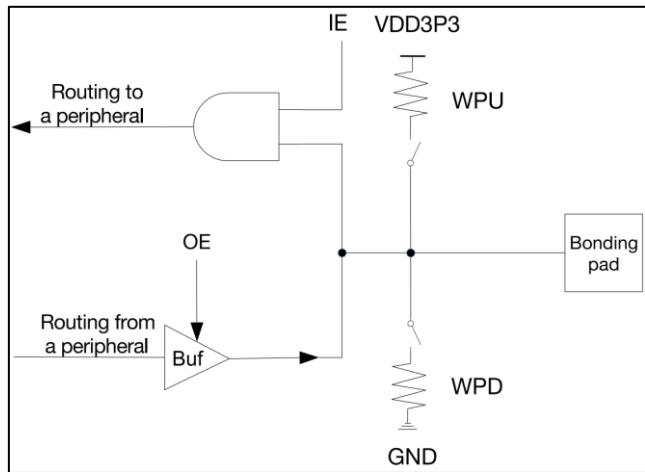


图 3.5.2.4 焊盘内部结构

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 3.5.2.3 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\_FUN\_Cy\_IN\_SEL\_CFG\_REG 进行配置的。这个寄存器的描述如下。

**GPIO\_FUNC<sub>y</sub>\_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\_FUNC<sub>y</sub>\_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 3.5.2.5 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

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 3.5.2.3 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
<b>GPIO_FUNCx_OUT_SEL</b> 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 <b>GPIO_OUT_REG/GPIO_OUT1_REG[x]</b> 和 <b>GPIO_ENABLE_REG/GPIO_ENABLE1_REG</b> [x] 作为输出值和输出使能。(读/写) <b>GPIO_FUNCx_OUT_INV_SEL</b> 0: 不反转输出值；1: 反转输出值。(读/写) <b>GPIO_FUNCx_OEN_SEL</b> 0: 采用外设的输出使能信号；1: 强制使用 <b>GPIO_ENABLE_REG[x]</b> 用作输出使能信号。(读/写) <b>GPIO_FUNCx_OEN_INV_SEL</b> 0: 不反转输出使能信号；1: 反转输出使能信号。(读/写)											

图 3.5.2.6 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 3.5.2.1 和图 3.5.2.3 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。

## 3.5.3 复位与时钟

### 3.5.3.1 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。

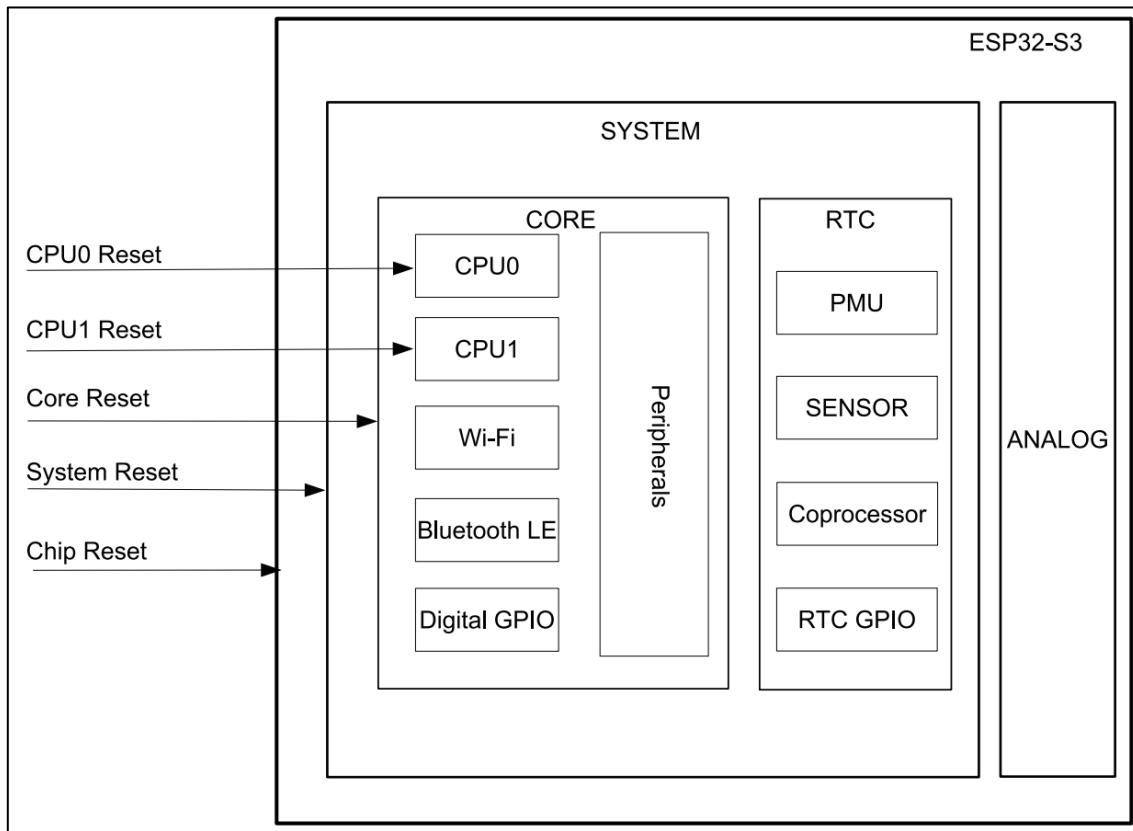


图 3.5.3.1.1 四种复位等级

**CPU 复位：**只复位 CPU<sub>x</sub> 内核，这里的 CPU<sub>x</sub> 代表 CPU0 和 CPU1。复位释放后，程序将从 CPU<sub>x</sub> 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 PU 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<sub>x</sub> 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 3.5.3.1.1 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。

### 3.5.3.2 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。

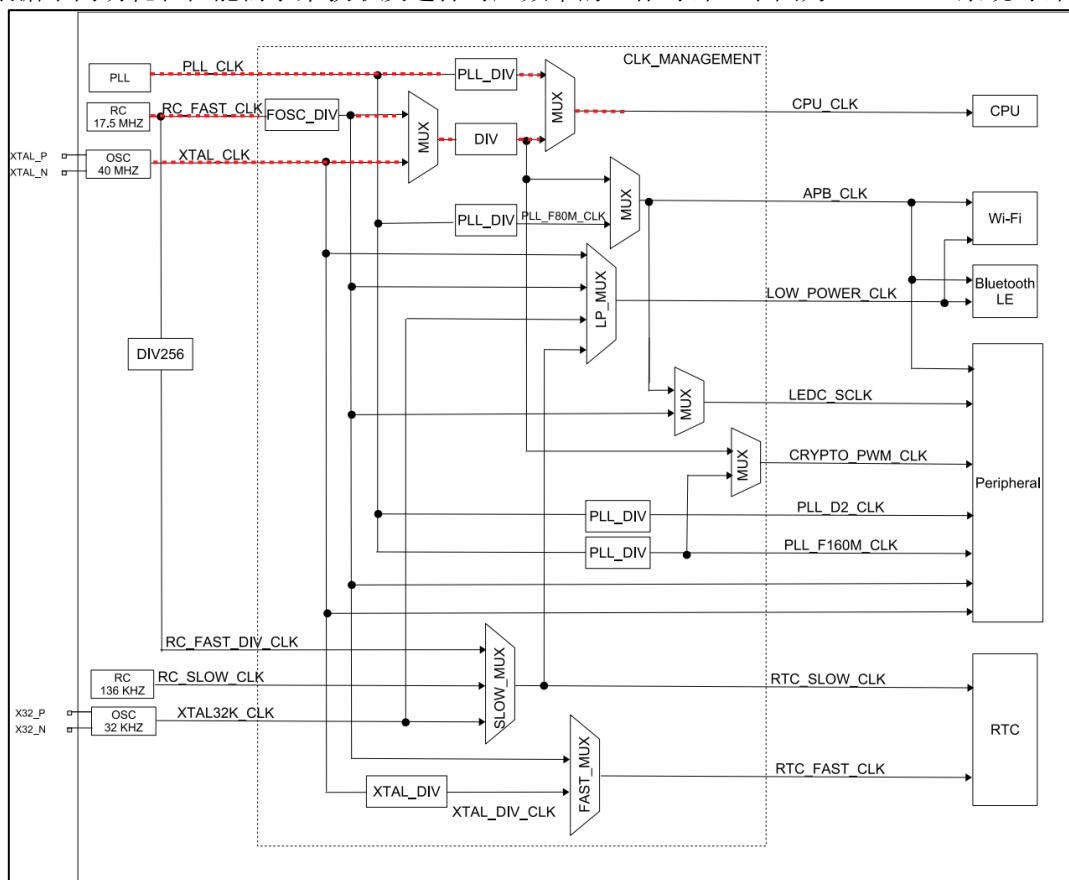


图 3.5.3.2.1 ESP32-S3 时钟树

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 3.5.3.2.1 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。

### 3.5.4 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 3.5.4.1 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 3.5.4.2 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 <sup>1</sup>	封装内 Flash	封装内 PSRAM	环境温度 <sup>2</sup> (°C)	VDD_SPI 电压 <sup>3</sup>
ESP32-S3	—	—	-40 ~ 105	3.3 V/1.8 V
ESP32-S3FN8	8 MB (Quad SPI) <sup>4</sup>	—	-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 3.5.4.1 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 最小系统板的模组选择的是 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 <sup>1</sup>	电压	VDD_SPI 电源 <sup>2</sup>
0	0	忽略	3.3 V	VDD3P3_RTC 通过 R <sub>SPI</sub> 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 <sub>SPI</sub> 供电

<sup>1</sup> eFuse: EFUSE\_VDD\_SPI\_TIEH

<sup>2</sup> 请参考章节 2.5.2 电源管理

图 3.5.4.2 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 <sup>a</sup>	eFuse 2 <sup>b</sup>	eFuse 3 <sup>c</sup>	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

<sup>a</sup> eFuse 1：表示是否永久禁用 JTAG 功能

<sup>b</sup> eFuse 2：表示是否禁用 usb\_serial\_jtag 模块的 usb 转 jtag 功能

<sup>c</sup> 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 3.5.4.3 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。

### 3.5.5 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。

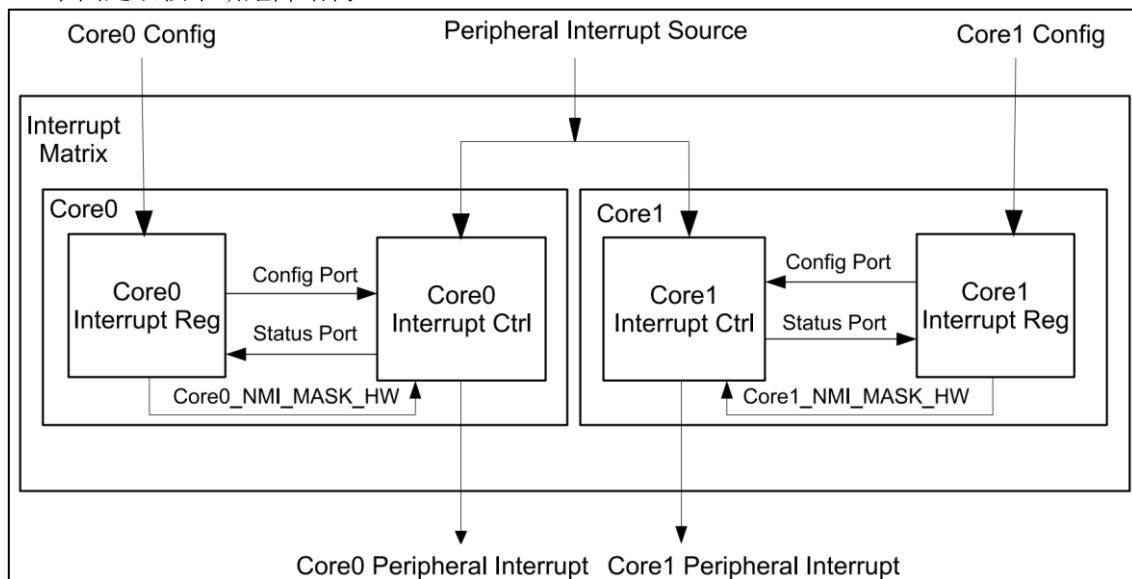


图 3.5.5.1 中断矩阵结构图

这种设计使得 ESP32-S3 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。

### 3.6 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 3.6.1 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。

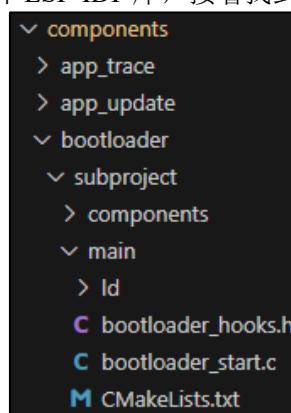


图 3.6.2 bootloader\_start.c 文件路径

在这个文件下，找到 call\_start\_cpu0 函数，此函数是 bootloader 程序，如下是 bootloader 程序的部分代码。

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

```

if (bootloader_before_init) {
    bootloader_before_init();
}

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

if (bootloader_after_init) {
    bootloader_after_init();
}

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

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

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

```

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

### 三、三级引导程序

应用程序的入口是在 esp-idf/components/esp\_system/port 路径下的 cpu\_star.c 文件，在此文件下找到 call\_start\_cpu0 函数（端口层初始化函数）。这个函数由二级引导加载程序执行，并且从不返回。因此你看不到是哪个函数调用了它，它是从汇编的最底层直接调用的。

这个函数会初始化基本的 C 运行环境（“CRT”），并对 SOC 的内部硬件进行了初始配置。执行 call\_start\_cpu0 函数完成之后，在 components\esp\_system\startup.c 文件下调用 start\_cpu0(在 110 行中，弱关联 start\_cpu0\_default 函数)系统层初始化函数，如下 start\_cpu0\_default 函数的部分代码。

```

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

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

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

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

```

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

```

/* components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c */
/* 开启 APP 程序 */
void esp_startup_start_app(void)

```

```
{  /* 省略部分代码 */
/* 新建 main 任务函数 */
esp_startup_start_app_common();

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

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

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

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

## 第四章 认识 ESP-IDF

ESP-IDF 是乐鑫科技为其 ESP32 系列芯片提供的官方开发框架。这个框架主要用于开发、构建和部署基于 ESP32 的物联网（IoT）应用。我们要写程序控制 ESP32 芯片，其实最终就是控制它的寄存器，使之工作在我们需要的模式下，ESP-IDF 库将大部分寄存器的操作封装成了函数，我们只需要学习和掌握 ESP-IDF 库函数的结构和用法，就能方便地驱动 ESP32 工作，以节省开发时间。

本章将分为以下几个小节：

- 4.1 ESP-IDF 简介
- 4.2 ESP-IDF 库框架结构解析
- 4.3 ESP-IDF 与乐鑫芯片
- 4.4 IDF 工程简介

### 4.1 ESP-IDF 简介

ESP-IDF（Espressif IoT Development Framework）是乐鑫（Espressif Systems）为 ESP 系列芯片开发的物联网开发框架。它支持 ESP32、ESP32-S、ESP32-C 和 ESP32-H 系列 SoC，基于 C/C++语言提供了一个自给自足的 SDK，方便用户在这些平台上开发通用应用程序。

#### 一、ESP-IDF 特点：

- (1) 免费开源：ESP-IDF 相关资源已在 GitHub 上免费开放。
- (2) 专业稳定：ESP-IDF 发布的版本均经过严格的测试流程，以确保版本稳定，客户可快速实现量产。
- (3) 功能丰富的软件组件：ESP-IDF 集成了大量的软件组件，包括 RTOS、外设驱动程序、网络栈、多种协议实现技术以及常见应用程序的使用助手。
- (4) 支持多种 IDE 开发：Eclipse 和 VSCode 等 IDE，确保其易于开发人员使用。
- (5) 丰富的文档和示例资源：ESP-IDF 提供详尽的软件组件使用和设计文档，有助于开发人员充分理解 ESP-IDF 功能，并从中挑选最适合构建其应用程序的模块。
- (6) 支持 Windows、Linux 和 macOS 系统平台上开发 ESP32 应用程序提供工具链、API、组件和工作流程（本教程选择 Windows 系统下开发）。

#### 二、自动化构建系统：

开发者只需要通过简单的命令即可触发整个编译流程。下图为 ESP-IDF 编译系统流程：

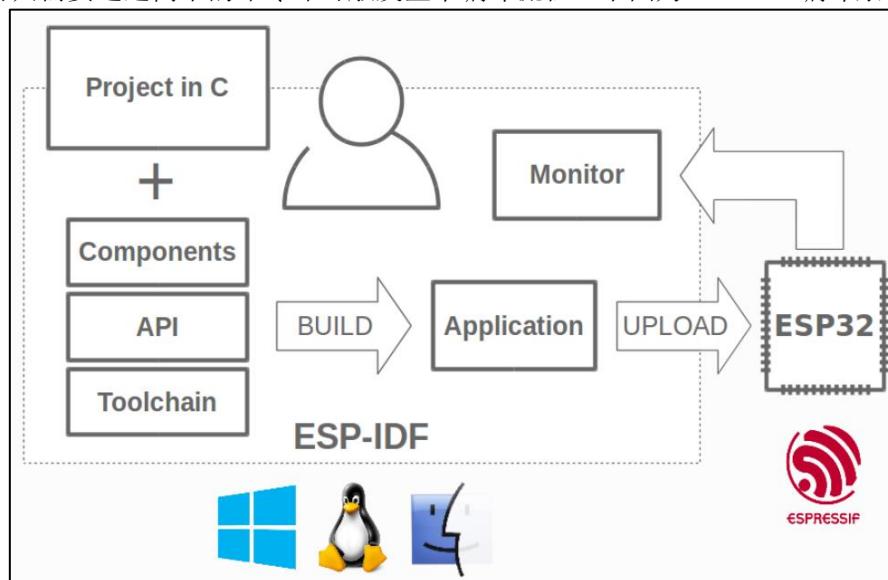


图 4.1.1 ESP-IDF 编译系统流程

从上述图示可见，项目的工程文件通过集成 C 项目、中间组件以及工具链，共同编译生成可执行文件。随后，这个可执行文件被下载到 ESP32 芯片中。ESP32 芯片能够通过其监控器功能，向开发者提供实时的反馈信息。这一流程使得开发者能够更有效地监控和管理 ESP32 芯片的运行状态，从而优化项目的开发过程。

ESP-IDF 的编译过程主要基于 Make 或 CMake 构建系统，它自动化地处理源代码的编译、链接和生成最终的可执行文件或固件镜像。简单来讲，ESP-IDF 可以使用命令的形式进行配置、编译、链接和构建项目，类似于 linux 的开发方式。整个编译过程通过构建系统自动化完成，开发者只需要通过简单的命令即可触发整个编译流程。

ESP-IDF 虽然提供了强大的功能和灵活性，但使用命令编译和构造等操作也存在一些缺点。

①：学习曲线陡峭。对于初学者来说，ESP-IDF 可能具有一定的学习难度。它需要一定的时间来理解其目录结构、编译系统、配置文件以及 API 等

②：编译过程繁琐。虽然 ESP-IDF 提供了命令来编译和构造项目，但编译过程可能会相对繁琐。特别是当项目规模较大或者依赖多个组件时，编译时间可能会延长，这可能会影响开发效率。

③：缺乏图形化界面。虽然 ESP-IDF 提供了命令行工具来进行项目构建和调试，但缺乏图形化界面的支持可能会使得某些操作相对不便。对于一些开发者来说，图形化界面可以提供更直观和易于使用的操作体验。

④：配置复杂。ESP-IDF 的配置文件（如 sdkconfig）可能相对复杂，特别是对于初学者来说。需要正确配置各种选项和参数，以确保项目能够正确编译和运行。配置错误可能导致编译失败或运行时问题。

对于上述的问题，乐鑫科技提供了一个解决方案，那就是在集成开发环境下开发 ESP-IDF，使得开发者更加快速的开发 ESP32 芯片。下面是乐鑫官方 ESP-IDF 支持的集成开发环境。

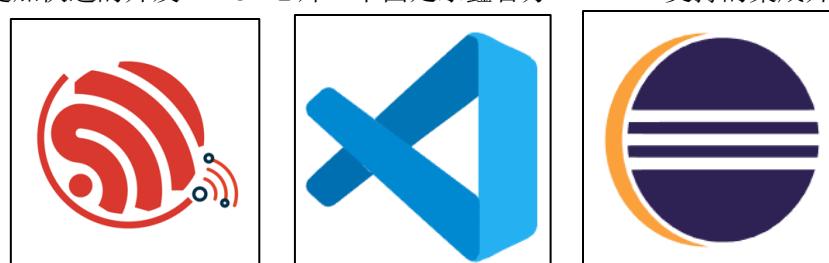


图 4.1.2 ESP-IDF 支持的集成开发环境

下图为 VS Code/Eclipse/Espressif-IDE 等 IDE 的开发 ESP-IDF 程序流程图。

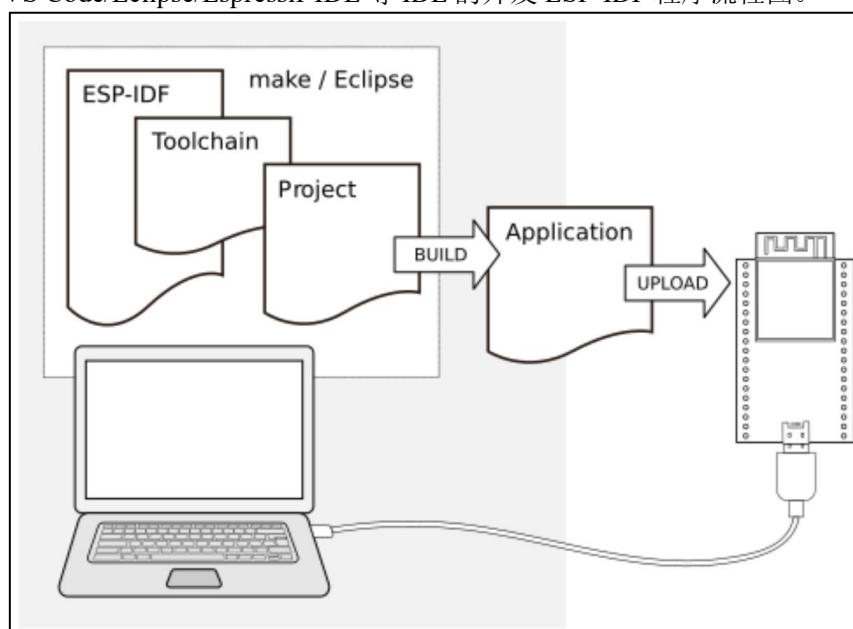


图 4.1.3 开发应用程序流程图

不难发现，图 4.1.1 的流程和图 4.1.3 的流程极为相似，唯一不同的是图 4.1.3 是基于 IDE 集成开发环境下开发的，但是它们的编译流程是一致的。

本教程中的所有例程都是基于 VS Code IDE 进行开发的。虽然并未提供基于 Eclipse IDE 的开发例程，但上图中展示的开发应用程序流程同样适用于 VS Code IDE 的开发流程。这意味着，无论使用哪种 IDE 开发 ESP-IDF，开发者都可以遵循这一流程来有效地进行应用程序的开发。

### 三、ESP-IDF 在 VS Code 集成开发环境开发具备那些特点

①：代码编辑与智能提示。安装了 ESP-IDF 扩展插件后，VSCode 将提供代码高亮、语法检查、自动补全等 IDE 功能，极大地提高了代码编辑的效率和准确性。插件还提供了对 ESP-IDF 特定函数和 API 的智能提示，帮助开发者更快地编写和调试代码。

②：构建与调试。ESP-IDF 扩展插件集成了构建和调试功能，开发者可以直接在 VSCode 中执行 idf.py build 等命令来构建项目。还可以通过配置 launch.json 文件，开发者还可以在 VSCode 中设置断点、单步执行、查看变量值等调试功能，从而方便地进行程序调试。

③：项目管理与配置。VSCode 支持多项项目管理，开发者可以方便地切换和管理不同的 ESP-IDF 项目。还通过 VSCode 中的配置文件（如.vscode/settings.json），开发者可以针对每个项目设置特定的构建和调试选项。

④：集成终端与日志查看。VSCode 内置了终端功能，开发者可以直接在 VSCode 中打开终端并执行 ESP-IDF 相关的命令。还可以通过集成终端，开发者可以方便地查看构建日志、调试输出等信息，从而快速定位和解决问题。

VSCode 与 ESP-IDF 之间的关系主要体现在 VSCode 为 ESP-IDF 提供了一个功能强大的集成开发环境，通过安装和使用 ESP-IDF 扩展插件，开发者可以更加高效地在 VSCode 中编写、构建、调试和部署 ESP32 等 Espressif 芯片的应用程序。

## 4.2 ESP-IDF 库框架结构解析

下面我们从 gitee 仓库下克隆 ESP-IDF 物联网开发框架的源代码，并在此分析各个文件的作用。克隆 ESP-IDF 源码库流程如下图所示。



```
MINGW64:/c/Users/ATK/Desktop/test
ATK@DESKTOP-QH7611H MINGW64 ~/Desktop/test
$ git clone https://gitee.com/EspressifSystems/esp-idf.git
```

图 4.2.1 克隆 ESP-IDF 源码库

克隆成功后，在上图路径下找到 esp-idf 文件夹，此文件夹就是 ESP-IDF 物联网开发框架的源码库，如下图所示。



图 4.2.2 ESP-IDF 源码库部分截图

下面作者来讲解一下这些文件夹的作用及特点，如下表所示：

文件名称	说明
components	提供了模块化、可配置、可重用和可扩展的代码组织方式，使得 ESP32 和其他 Espressif 芯片的开发变得更加高效和灵活
docs	ESP-IDF 相关的文档和指南，旨在帮助开发者更好地理解和使用 ESP-IDF
examples	为开发者提供了丰富的学习资源、原型开发工具和功能演示。

tools	提供了开发 ESP-IDF 项目所需的各种工具和脚本，帮助开发者更高效地完成开发、构建和部署任务
-------	--

表 4.2.1 esp-idf 源码库的框架结构解析

正如图 4.1.1 所示那样，在编译 ESP32 的例程时，确保 components 和 tools 目录的完整性是非常重要的。components 目录包含了项目所需的所有源代码和库文件，而 tools 目录则提供了编译和链接这些代码所需的工具链，这样才能做到自给自足的构建项目。

### 4.3 ESP-IDF 与乐鑫芯片

每款乐鑫芯片都可能有不同版本，下表总结了乐鑫芯片在 ESP-IDF 各版本中的支持状态，其中“√”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。

芯片型号	V4.3	V4.4	V5.0	V5.1	V5.2
ESP32	√	√	√	√	√
ESP32-S2	√	√	√	√	√
ESP32-C3	√	√	√	√	√
ESP32-S3		√	√	√	√
ESP32-C2			√	√	√
ESP32-C6				√	√
ESP32-H2				√	√
ESP32-P4					预览

表 4.3.1 ESP-IDF 与乐鑫芯片关系表

从上表可知：每款乐鑫芯片都可能有不同版本。建议参考 [\[ESP-IDF 版本与乐鑫芯片版本兼容性\]](#)，了解 ESP-IDF 版本与各芯片版本之间的兼容性，作者建议读者最好找到合适的 ESP-IDF 版本来开发自己的 SoC 芯片。

对于 2016 年之前发布的乐鑫芯片（包括 ESP8266 和 ESP8285），请参考 [\[RTOS SDK\]](#)。

### 4.4 IDF 工程简介

ESP-IDF 工程，特别是 ESP-IDF 项目，可以看作是由多个不同组件集合而成的工程。这些组件包括 ESP-IDF 基础库（如 libc、ROM bindings 等）、Wi-Fi 驱动、TCP/IP 协议栈、FreeRTOS 操作系统、网页服务器、各种传感器驱动（如湿度传感器驱动）以及负责将上述组件整合到一起的主程序等。

ESP-IDF 工程借助 CMake 的可自定义性，创新地采用了“组件”式的设计。整个工程由多个组件组成，每个组件都像是一块积木，共同构建起完整的工程结构。在这些组件中，有一个特殊的组件被称为“main”组件，它包含了用户应用程序的入口函数。

组件之间的关系主要以依赖关系为主，确保了它们之间的有序协作。每个组件都可以拥有单独的配置，这进一步增加了工程的灵活性和可定制性。在 ESP-IDF 中，用户可以明确地指定和配置每个组件。构建系统会在 ESP-IDF 目录、项目目录以及用户自定义的组件目录（如存在）中查找所有的组件。这一过程允许用户通过文本菜单系统对 ESP-IDF 项目中使用的每个组件进行配置。当所有组件的配置完成后，构建系统便会开始编译整个项目，确保所有组件能够按照预期的方式协同工作，从而生成最终的工程成果。这种设计不仅简化了工程的开发和维护，也提高了开发效率。

#### 1. ESP32 工程有如下重要概念：

①：项目（Project）特指一个目录，其中包含了构建可执行应用程序所需的全部文件和配置，以及其他支持型文件，例如分区表、数据/文件系统分区和引导程序。

②：项目配置保存在项目根目录下名为 sdkconfig 的文件中，可以通过 idf.py menuconfig 进行修改，且一个项目只能包含一个项目配置。

③：应用程序是由 ESP-IDF 构建得到的可执行文件。一个项目通常会构建两个应用程序：项目应用程序（可执行的主文件，即用户自定义的固件）和引导程序（启动并初始化项目应用程序）。

④：组件（Components）是模块化且独立的代码，会被编译成静态库（.a 文件）并链接到应用程序。部分组件由 ESP-IDF 官方提供，其他组件则来源于其它开源项目。

⑤：目标（Target）特指运行构建后应用程序的硬件设备。ESP-IDF 当前仅支持 esp32 和 esp32s2 以及 esp32s3 这三个硬件目标。

请注意，以下内容并不属于项目的组成部分：

①：ESP-IDF 并不是项目的一部分，它独立于项目，通过 IDF\_PATH 环境变量（保存 esp-idf 目录的路径）链接到项目，从而将 IDF 框架与项目分离。

②：交叉编译工具链并不是项目的组成部分，它应该被安装在系统 PATH 环境变量中。

## 2. ESP32 项目工程分析

下面作者以 sample\_project 示例（D:\ESP32\Espressif\frameworks\esp-idf-v5.1.2\examples\get-started\sample\_project 路径下找到）为例，来讲解基础工程的构建项目原理。sample\_project 工程目录如下图所示：

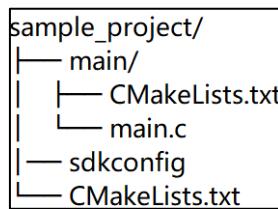


图 5.3.9 test\_1 项目的文件结构

①：顶层项目 CMakeLists.txt 文件，这是 CMake 用于学习如何构建项目的主要文件，可以在这个文件中设置项目全局的 CMake 变量。顶层项目 CMakeLists.txt 文件会导入 /tools/cmake/project.cmake 文件，**由它负责实现构建系统的其余部分**。该文件最后会设置项目的名称，并定义该项目。每个项目都有一个顶层 CMakeLists.txt 文件，包含整个项目的构建设置。默认情况下，项目 CMakeLists。文件会非常小，如下代码所示：

```

/* 必须放在 CMakeLists.txt 文件的第一行,
   它会告诉 CMake 构建该项目所需要的最小版本号。ESP-IDF 支持 CMake 3.16 或更高的版本 */
cmake_minimum_required(VERSION 3.16)
/* 会导入 CMake 的其余功能来完成配置项目、检索组件等任务 */
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
/* 会创建项目本身，并指定项目名称 */
project(myProject)
  
```

②：main 目录是一个特殊的组件，它包含两个文件，它们分别为 CMakeLists.txt 和 main.c 文件。其中 main.c 定义了程序入口函数 app\_main()，而 CMakeLists.txt 文件是将组件添加到构建系统中，如下图所示：

```

idf_component_register(SRCS "main.c"
                      INCLUDE_DIRS ".")
  
```

通过调用 idf\_component\_register 函数，开发者可以将组件添加到构建系统中。在此过程中，SRCS 代表源文件列表，其中包括 .c、.cpp、.cc、.S 等类型的文件，这些源文件都将被编译并整合进组件库中。另外，INCLUDE\_DIRS 指的是目录列表，这些目录中的路径将被添加到所有需要该组件（包括主组件）的全局 include 搜索路径中，确保在编译过程中能够正确找到相关的头文件。注意：若该组件需要依赖其他的驱动代码，可使用 REQUIRES 设置依赖库，具体内容请参考 ESP-IDF 编程指南的组件依赖章节。

③：“sdkconfig” 项目配置文件，执行 idf.py menuconfig 时会创建或更新此文件，文件中保存了项目中所有组件（包括 ESP-IDF 本身）的配置信息。sdkconfig 文件可能会也可能不会被添加到项目的源码管理系统中。

从上述内容可以得知以下总结：子层的 CMakeLists.txt 文件负责将对应层的组件整合进构建系统当中，而 sdkconfig 文件则用于设置构建过程中所需的多种配置选项（该编译哪些代码，不编译哪些代码）。同时，顶层的 CMakeLists.txt 文件不仅指定了 CMake 的版本，还通过引用 ESP-IDF 路径下的 project.cmake 文件来指导整个项目的构建流程。这些文件共同协作，确保项目能够顺利构建和配置。最终，项目构建完成后会生成一个 build 文件夹，其中包含了临时目标文件、库文件以及最终输出的二进制文件。

## 第五章 搭建开发环境

在前面章节中，我们已经大致了解了 ESP32-S3 基础知识及 ESP-IDF 物联网开发框架的编译与链接原理，现在，我们将进入实际操作阶段，逐步搭建 ESP-IDF 的开发环境。

本章分为以下几个小节：

- 5.1 安装 ESP-IDF 物联网开发框架
- 5.2 IDF 前端工具
- 5.3 搭建集成开发环境

### 5.1 安装 ESP-IDF 物联网开发框架

为了安装 ESP32 的开发环境，我们需要前往[乐鑫官方的 Windows 安装下载中心](#)下载 ESP32-IDF 安装包。在这里，我们推荐下载离线的安装包，虽然安装速度可能会稍慢一些，但它能够确保安装的成功率。相比之下，在线的安装包需要稳定的网络支持，如果网络状况不佳，可能会导致安装失败。当然，也可以在 A 盘→6，软件资料→1，软件→1，IDF 开发工具→04-ESP32-IDF Offline Installer 路径下找到 5.1.2 离线安装包。ESP\_IDF 物联网开发框架安装包如下图所示：

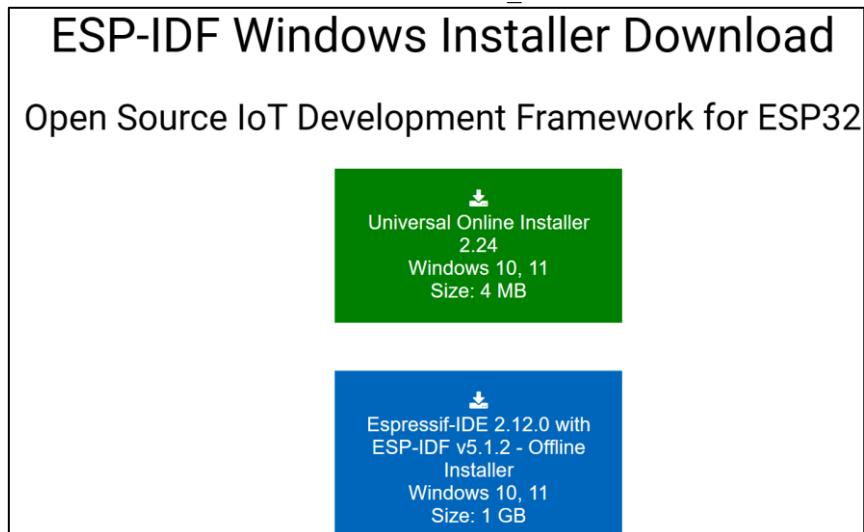


图 5.1.1 下载 V5.1.2 离线安装包（教程编写时最新版本）

下载成功后，在安装程序上单击右键选择<以管理员身份运行>运行 `esp-idf-tools-setup-offline-5.1.2.exe` 文件，如下图所示：

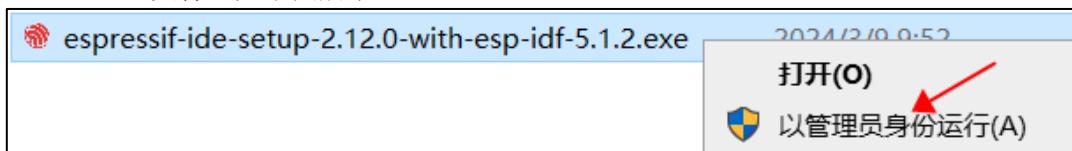


图 5.1.2 以管理员身份运行安装包

打开安装程序后选择简体中文安装，如下图所示：

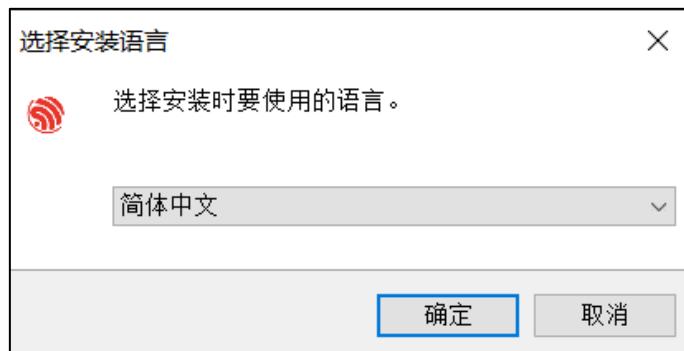


图 5.1.3 选择安装语言

往下走就是许可协议，勾选“我同意此协议”，单击下一步，如下图所示：



图 5.1.4 勾选“我同意此协议”选择

点击下一步之后，会跳出安装前系统检查界面，如下图所示：

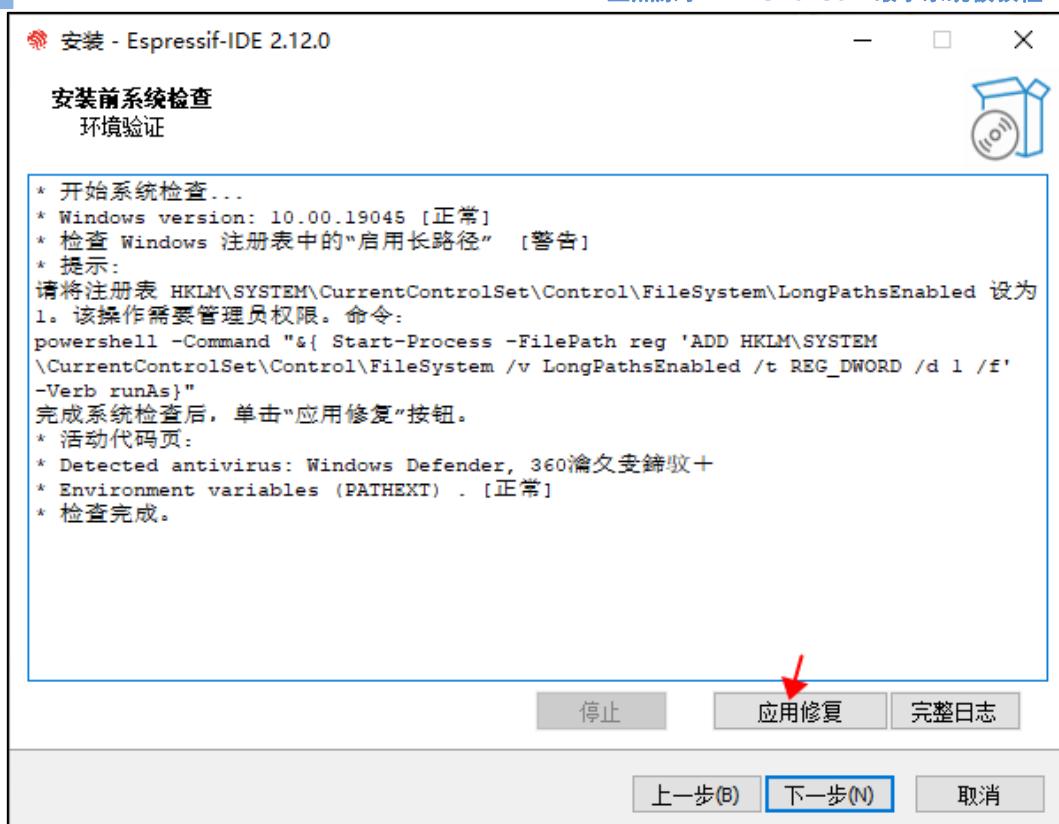


图 5.1.5 安装界面

安装程序会检查你当前系统有没有打开"长路径支持", 因为 GNU 编译器产生的编译文件会有非常深的目录结构, 如果不支持长路径, 编译可能出现文件不存在, 目录不存在等奇怪的错误。这里单击**应用修复**按钮, 可以修复这个问题。在弹出的确认对话框中, 选择是, 开始修复。

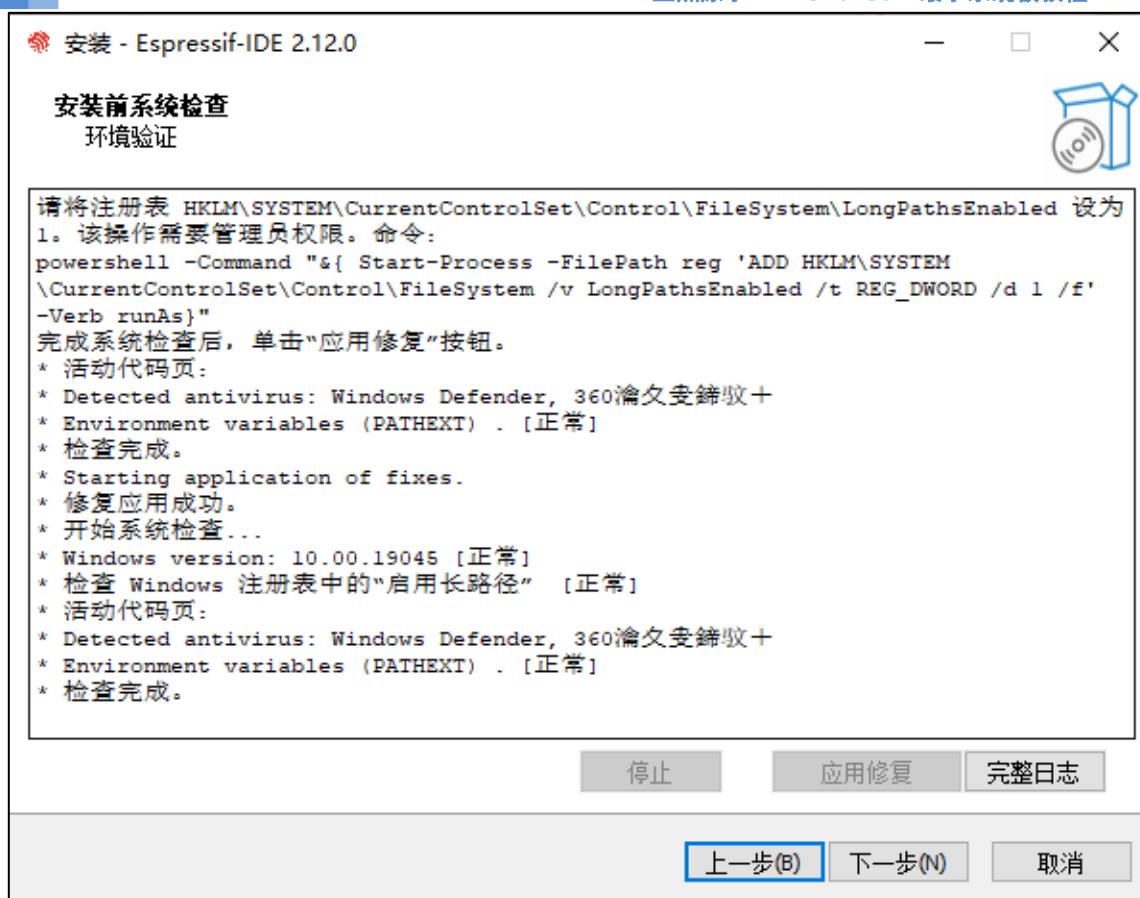


图 5.1.6 在注册表中启用长路径

如果修复不成功，一般情况是安装软件打开时没有使用管理员权限打开，可以手动修改注册表来支持长路径：打开注册表 HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\LongPathEnabled 设置为 1。如下图所示：



图 5.1.7 手动启用长路径

图 3.1.6 提示修复完成后，点击下一步进入配置安装路径，如下图所示：

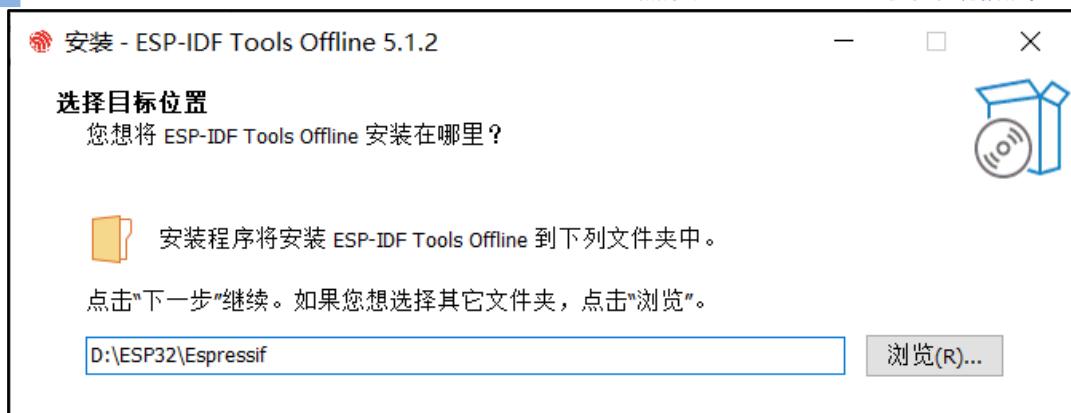


图 5.1.8 配置安装路径

安装程序默认的安装位置为 C:\Espressif，但这里我是安装在 D 盘，如果全部源码编译后可能产生几十 G 的大小占用，我们在 D 盘下创建 ESP32\Espressif 文件夹来保存 ESP32-IDF 库安装过程中生成的文件。注意：这个安装路径非常重要，因为 VS Code 软件的 IDF 插件需要此路径来获取相关文件，所以开发者务必牢记该路径。

设置安装路径后点击“下一步”选项，进入确认安装组件界面，这里全部打勾，默认完全安装时 ESP32C2 是不打勾的（如下图所示），看需要自己选择。然后单击下一步。

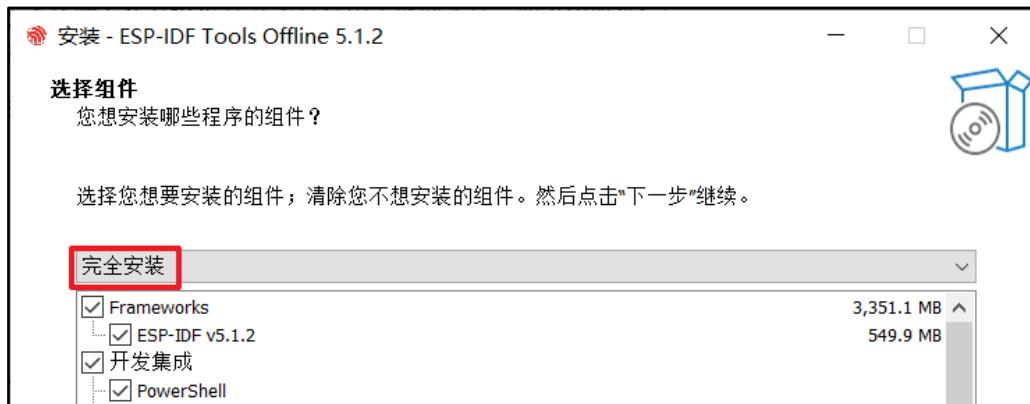


图 5.1.9 选择安装组件

我们选择全部安装。点击下一步再次确认安装目录信息，然后单击安装。安装完成，三个全部勾选，1、2 用于测试环境安装是否成功，3 是将 ESP-IDF 工具链加入杀毒工具排除项，以加快编译速度，如下图所示：



图 5.1.10 ESP32-IDF 库安装完成

安装成功后，桌面自动生成 ESP-IDF 5.1 PowerShell 和 ESP-IDF 5.1 CMD 命令提示符终端，PowerShell 提供了更强大的脚本和自动化功能，适合需要执行复杂任务或管理复杂环境的用户；而 CMD 则更适合进行基础的命令行操作和简单的脚本执行。用户可以根据自己的需求和偏好选择使用其中一个工具。

打开其中一个终端，如果终端提示“`idf.py build`”指令时，说明我们的环境已经安装成功了。如下图所示：

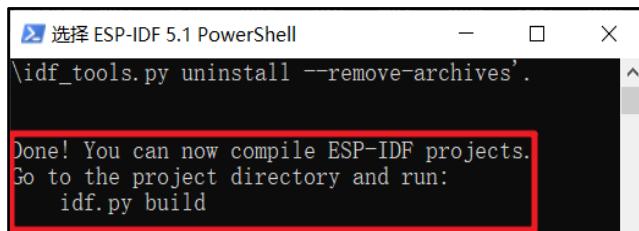


图 5.1.11 PowerShell 窗口

从上图中可以看到，当出现红色方框内的内容时，可以初步证明安装是没有问题的。**在这终端下，我们可以采用命令形式进行配置、编译、链接和构建项目，这与在 Linux 中的开发方式颇为相似。下一个小节，将详细讲解 ESP-IDF 常用的命令。**

为了让系统能够找到和识别 ESP-IDF 的相关工具和库，从而能够顺利地进行编译、构建和调试 ESP32 或其他 Espressif 芯片的项目，我们必须设置 ESP-IDF 的环境变量，设置方法如下：

按照此过程（此电脑→属性→高级系统→环境变量）打开，如下图所示：

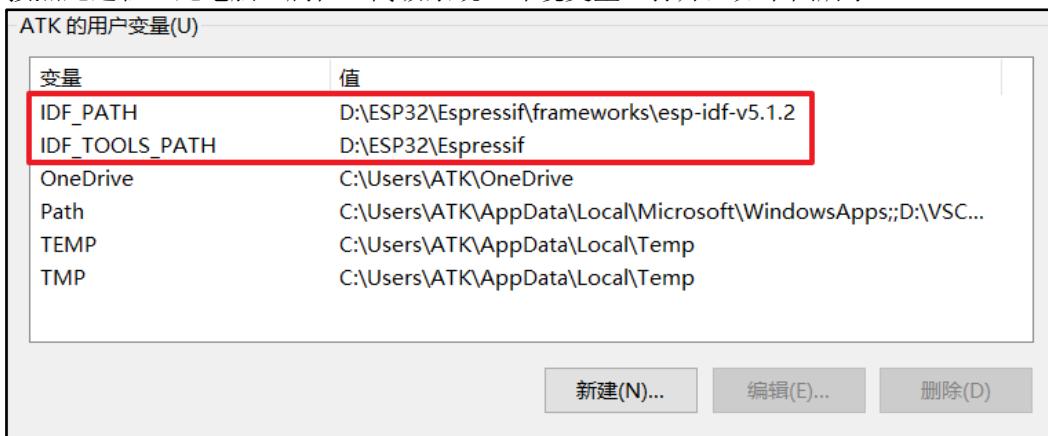


图 5.1.12 添加环境变量

如果 ESP-IDF 库安装成功，则系统自动为我们添加上图中的 `IDF_TOOLS_PATH` 和 `IDF_PATH` 环境变量，否则手动添加这两个环境变量。

安装完成后，系统会自动安装 Espressif-IDE 集成开发环境，这是专为乐鑫 SoC 芯片开发的集成开发环境。鉴于该软件在国内发布时间尚短，且国内开发者多倾向于使用 VS Code IDE 进行开发，因此本教程的例程主要基于 VS Code IDE 展开。然而，我们正点原子也致力于推广 Espressif-IDE 集成开发环境，故决定额外编写一个关于 Espressif-IDE 集成开发环境的使用教程，以更好地帮助国内开发者熟悉并利用这一强大的集成开发环境（请读者查看《Espressif-IDE 集成开发环境使用指南》）。

### 安装 USB 虚拟串口驱动

ESP32-S3 的 USB 串口可用于下载程序和 ESP 监控器之间的交互。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。正点原子的 ESP32-S3 最小系统板的串口信号是通过 CH343P 芯片进行转换，才能与 PC 端进行通信。CH343P 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 CH343P 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 CH343P 芯片的驱动程序。CH343P 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 CH343P 的驱动程序，也可在 6，软件

资料→1，软件→CH343P 驱动文件夹下找到 CH343P 的驱动安装程序，如下图所示。



图 5.1.13 CH343P 驱动安装程序

打开 CH343P 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 CH343P 驱动已经安装成功了，如下图所示。

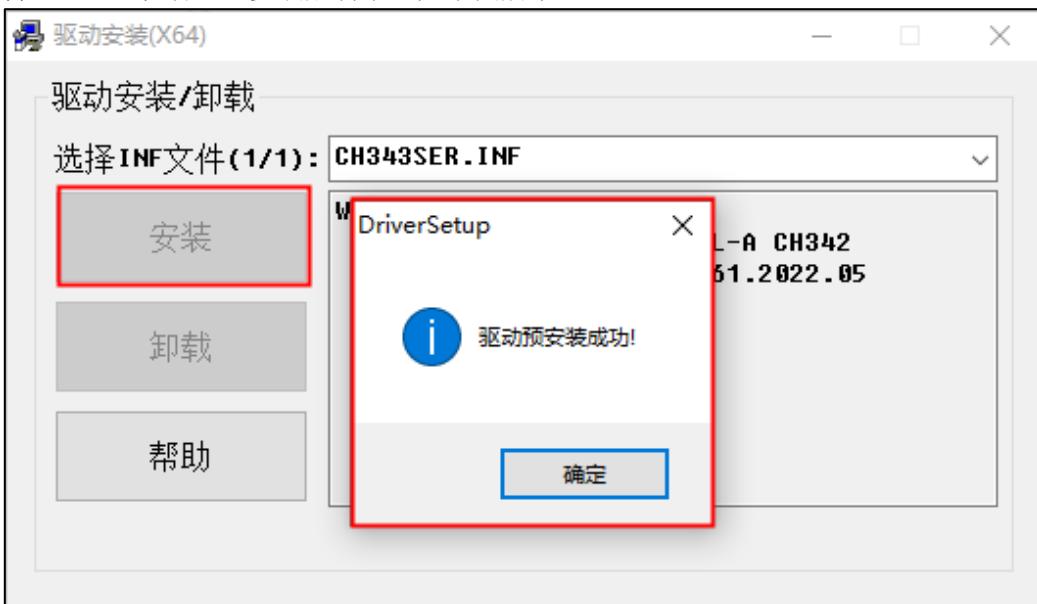


图 5.1.14 CH343P 驱动安装成功

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 CH343P 虚拟出的串口，如下图所示。



图 5.1.15 PC 端显示的虚拟串口

从上图可以看出，CH343P 虚拟出的串口被 PC 分配了 COM5 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 CH343P 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 COM6 或 COM7。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 CH343 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。

接下来，作者将详细阐述 ESP-IDF 的两种开发方式。这两种方式分别是命令式开发和基于 IDE 集成开发环境下的开发。为了让读者能够更深入地理解和掌握这两种开发方式，作者将分别用两个小节来讲解这两部分的内容。

## 5.2 IDF 前端工具

在上一章节中，作者已经介绍过，ESP-IDF 能够通过命令的形式来构建或编译系统，这一命令被称为 ESP-IDF 的前端工具。此工具旨在简化 ESP32 以及其他 Espressif 芯片系列的开发流程。它提供了诸多实用功能，从而协助开发者更加高效地构建、烧录以及调试项目。

以下是 idf.py 的一些主要功能：

(1) 构建系统: idf.py 是一个构建系统, 它使用 CMake 来生成适用于不同目标平台的构建文件。你可以使用 idf.py 来构建你的项目, 它会处理所有必要的编译和链接步骤。

(2) 菜单配置: idf.py menuconfig 命令提供了一个文本用户界面, 用于配置项目的各种选项。你可以通过这个界面选择目标硬件、设置编译选项、启用或禁用组件等。

(3) 烧录和调试: idf.py 支持将构建好的二进制文件烧录到目标设备上。你可以使用 idf.py -p PORT flash 命令来烧录固件, 其中 PORT 是你的设备的串口或 USB 端口。此外, idf.py 还可以与调试器配合使用, 例如 GDB, 以便在目标设备上调试代码。

(4) 清理和重新构建: idf.py 提供了清理构建文件的功能, 以确保每次构建都是从头开始的。这对于在修改配置或更新源代码后重新构建项目非常有用。

(5) 项目模板和示例: idf.py 通常与 ESP-IDF 提供的项目模板和示例代码一起使用。这些模板和示例为你提供了一个良好的起点, 帮助你快速设置和开始你的项目。

(6) 扩展性: idf.py 是可扩展的, 允许开发者添加自定义的构建步骤和脚本。这使得开发者可以根据需要定制构建过程, 以满足特定项目的需求。

下面作者来讲解一下 IDF 前端工具的常用命令。

### 1, 创建新工程 (create-project)

“idf.py create-project --path <project name>” 是 ESP-IDF 提供的一个命令, 用于创建一个新的项目目录结构, 并将必要的文件和模板复制到该目录中。这个命令的目的是帮助开发者快速设置一个新的 IoT 项目, 而无需手动创建所有必要的文件和目录。

下面是这个命令的参数解析和使用方法。

#### ①: 参数解析

--path: 指定创建工程的位置 (必须在文件夹路径下)。

<project name>: 项目工程名称。

#### ②: 使用方法

在桌面新建 test\_1 文件夹, 用来保存 ESP-IDF 新建的工程, 然后打开 ESP-IDF CMD 终端输入 “idf.py create-project --path C:\Users\ATK\Desktop\test\_1 led” 命令创建项目工程, 如下图所示:

```
PS D:\ESP32\Espressif\frameworks\esp-idf-v5.1.2> cd C:\Users\ATK\Desktop\test_1
PS C:\Users\ATK\Desktop\test_1> idf.py create-project --path C:\Users\ATK\Desktop\test_1 led
PS C:\Users\ATK\Desktop\test_1>
```

图 5.2.1 指定路径创建工程

此时桌面会创建 test\_1 项目, 而工程名称为 led。如下图所示:

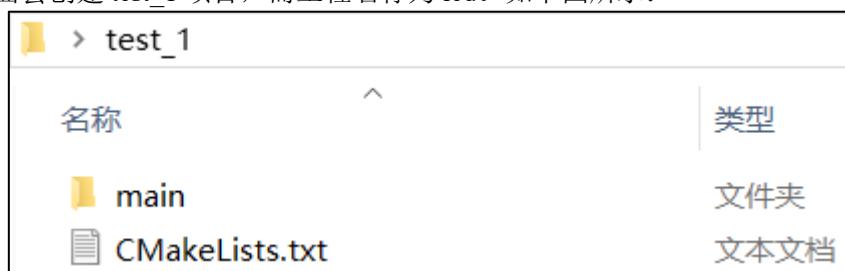


图 5.2.2 新建项目工程

### 2, 创建新组件 (create-component)

“idf.py create-component <component name>” 创建一个新的组件, 包含构建所需的最基本文件集。一般存放第三方组件, 如编写的驱动程序等。

下面是这个命令的参数解析和使用方法。

#### ①: 参数解析

<component name>: 组件名称

#### ②: 使用方法

首先使用 “cd” 命令进入新建工程路径, 然后在此路径下新建组件文件夹, 如下图所示:

```

PS D:\ESP32\Espressif\frameworks\esp-idf-v5.1.2> cd C:\Users\ATK\Desktop\test_1
PS C:\Users\ATK\Desktop\test_1> idf.py create-component component
Executing action: create-component
The component was created in C:\Users\ATK\Desktop\test_1\component
PS C:\Users\ATK\Desktop\test_1>

```

名称	类型	大小
component	文件夹	
main	文件夹	
CMakeLists.txt	文本文档	1 KB

图 5.2.3 新建组件

component 文件夹一般用来存储第三方组件或者用户的程序驱动代码。

### 3，设置目标芯片

“idf.py set-target <target>” 命令用于设置工程的目标芯片。由于 ESP-IDF 支持多款乐鑫 SoC 芯片，新建工程时默认会选择 ESP32 类型的芯片。因此，如果我们希望创建一个针对 ESP32-S3 类型的工程，就必须使用此命令来指定该工程的目标芯片为 ESP32-S3。

下面是这个命令的参数解析和使用方法。

#### ①：参数解析

< target >: 目标芯片，可使用 “idf.py --list-targets” 命令查看支持的芯片类型。

#### ②：使用方法

首先输入 “idf.py --list-targets” 命令查看支持的芯片类型，然后输入 “idf.py set-target esp32s3” 命令设置工程的目标芯片，如下图所示：

```

选择 ESP-IDF 5.1 PowerShell
The component was created in C:\Users\ATK\Desktop\test_1\component
PS C:\Users\ATK\Desktop\test_1> idf.py --list-targets
esp32
esp32s2
esp32c3
esp32s3
esp32c2
esp32c6
esp32h2
PS C:\Users\ATK\Desktop\test_1> idf.py set-target esp32s3
Adding "set-target"'s dependency "fullclean" to list of commands with default set of options.
Executing action: fullclean

```

名称	类型	大小
build	文件夹	
component	文件夹	
main	文件夹	
CMakeLists.txt	文本文档	1 KB
sdkconfig	文件	59 KB

图 5.2.4 设置工程目标芯片

注意：“idf.py set-target” 命令将清除构建目录，并从头开始重新生成 sdkconfig 文件。旧的 sdkconfig 文件将保存为 sdkconfig.old。

### 4，编译工程

“idf.py build” 命令用来编译当前项目工程。如下图所示：

```

ESP-IDF 5.1 PowerShell
Running ninja in directory C:\Users\ATK\Desktop\test_1\build
Executing "ninja all"...
[5/943] Generating ../../partition_table/partition-table.bin
Partition table binary generated. Contents:
*****
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 24K,
phy_init, data, phy, 0xf000, 4K,
factory, app, factory, 0x10000, 1M,
*****
[219/943] Building C object esp-idf/mbedtls/library/CMakeFiles/mbedx509.dir/x509_crl.c.obj

```

图 5.2.5 编译当前项目工程

## 5，监控项目

“idf.py monitor” 命令用来监控当前项目。监控之前必须安装 USB 虚拟串口驱动以及开发板上的 USB 串口接入到电脑当中，才能监控当前项目工程。

```

PS C:\Users\ATK\Desktop\test_1> idf.py monitor
Executing action: monitor
Serial port COM9
Connecting...
Detecting chip type... ESP32-S3
Running idf_monitor in directory C:\Users\ATK\Desktop\test_1
Executing "D:\ESP32\Espressif\python_env\idf5.1_py3.11_env\Scripts\python.exe D:\ESP32\Esspressif\frameworks\esp-idf-v5.1.2\tools\idf_monitor.py -p COM9 -b 115200 --toolchain-prefix xtensa-esp32s3-elf" --target esp32s3 --revision 0 C:\Users\ATK\Desktop\test_1\build\led.elf --force-color -m 'D:\ESP32\Espressif\python_env\idf5.1_py3.11_env\Scripts\python.exe' D:\ESP32\Esspressif\frameworks\esp-idf-v5.1.2\tools\idf.py'...
--- WARNING: GDB cannot open serial ports accessed as COMx
--- Using \\.\COM9 instead...
--- esp-idf-monitor 1.3.4 on \\.\COM9 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ESP-ROM:esp32s3
Build:Mar 27 2021
rst:0x1 (POWERON),boot:0x2b (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fce3818, len:0x1758
load:0x403c9700, len:0x4
load:0x403c9704, len:0xc00
load:0x403cc700, len:0x2e04
entry 0x403c9908
I (27) boot: ESP-IDF v5.1.2-dirty 2nd stage bootloader
I (27) boot: compile time Mar 9 2024 11:39:36
I (27) boot: Multicore bootloader
I (30) boot: chip revision: v0.2
I (34) boot.esp32s3: Boot SPI Speed : 80MHz

```

图 5.2.6 监控项目工程（类似串口打印系统数据）

注意：请按“Ctrl + ]”快捷键退出监控器。

## 6，配置项目

“idf.py menuconfig” 这个命令会启动一个文本用户界面，允许开发者为他们的 ESP32 或其他 Espressif SoC 芯片系列的项目配置各种选项。如下图所示：

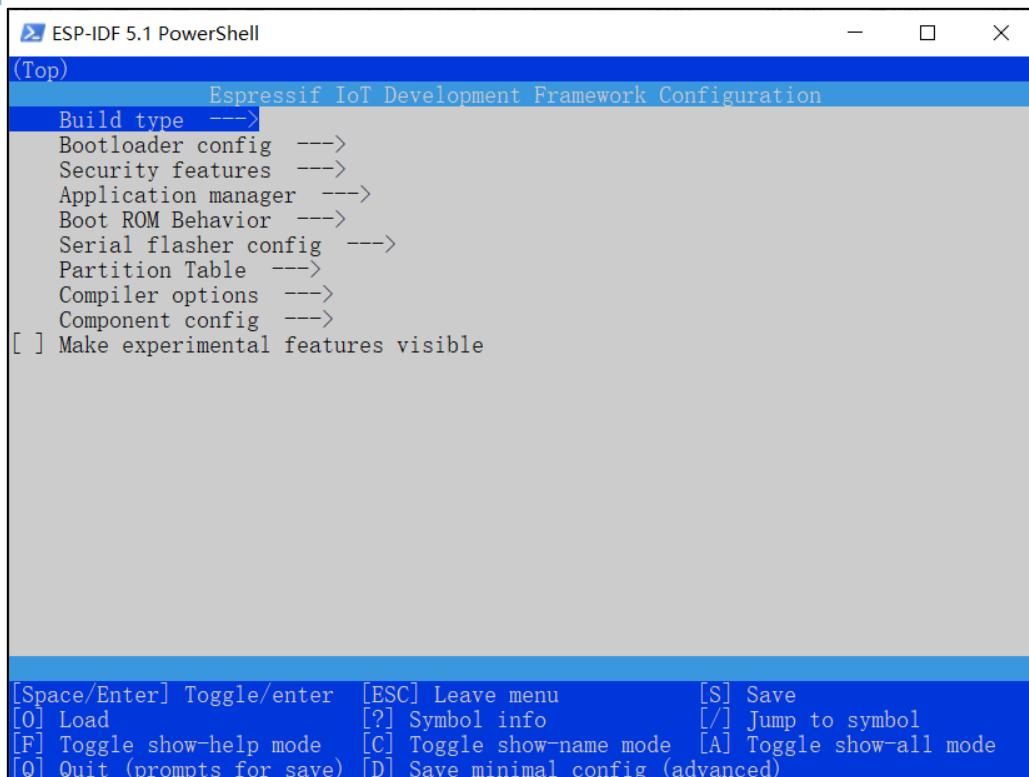


图 5.2.7 配置项目

menuconfig 菜单项的内容解析，作者会在以后的章节中讲解这部分的内容，读者先知道有这么一回事即可。

## 7. 下载代码

“idf.py -p COM9 flash”这个命令用来把编译出来的可执行文件烧录到 ESP32-S3 芯片当中。注意：烧录之前必须调用“idf.py build”命令编译项目工程，编译完成后方能烧录代码。如下图所示：

```
ESP-IDF 5.1 PowerShell
Compressed 20928 bytes to 13295...
Writing at 0x00000000... (100 %)
Wrote 20928 bytes (13295 compressed) at 0x00000000 in 0.7 seconds (effective 250.3 kbit/s).
Hash of data verified.
Compressed 201360 bytes to 109293...
Writing at 0x00010000... (14 %)
Writing at 0x0001cd24... (28 %)
Writing at 0x00022780... (42 %)
Writing at 0x00028e22... (57 %)
Writing at 0x0002f386... (71 %)
Writing at 0x000371d1... (85 %)
```

图 5.2.8 烧录代码

上述 COM9 端口需要根据自己的开发板识别出来的虚拟串口端口。

## 8. 清除编译文件

“idf.py clean”和“idf.py fulldclean”是 ESP-IDF (Espressif IoT Development Framework) 中用于清理构建目录和输出文件的两个命令，它们的主要区别在于清理的彻底程度和范围。

### ① “idf.py clean”命令：

这个命令主要用于清理构建目录中的构建输出文件。它会删除 build 文件夹中的某些文件，但不会删除 CMake 的配置输出和其他相关文件。这意味着下次构建时，CMake 将基于现有的配置信息重新生成所需的构建输出，但不需要从头开始配置整个项目。这通常用于在不需要更改项目配置的情况下，重新构建项目以解决可能存在的构建问题或更新代码。

### ② “idf.py fulldclean”命令：

这个命令则更为彻底，它会删除整个 build 目录下的所有内容，包括所有的 CMake 配置输出文件。这意味着下次构建项目时，CMake 将需要从头开始配置项目，重新生成所有的构建输

出和配置文件。这个命令通常用于在需要完全重置项目构建环境的情况下使用，例如在更改了项目的硬件配置或需要确保从头开始全新构建。

上述提到的命令是 IDF 前端工具中极为常用的指令，掌握了这些命令，您就可以开始着手开发 ESP32 项目了。至于其他命令的详细信息，建议读者查阅 [ESP-IDF 编程指南中的 IDF 前端工具](#) 章节内容。

看到此处，许多读者对命令式开发感到束手束脚，毕竟命令式开发难以轻松调试和编写代码，更何况在大型项目中，开发效率会大大降低。因此，为了让开发者能够更顺畅地使用 ESP-IDF，作者建议开发者选择基于集成开发环境（IDE）的开发方式。这种方式能更好地支持代码的调试和编写。至于 IDE 的选择，作者特别推荐 VS Code IDE，因为它是一款免费且开源的 IDE 软件，非常适合用于 ESP-IDF 的开发工作。

## 5.3 搭建集成开发环境

在上一小节中，作者详细阐述了命令式开发的常用命令。然而，对于初学者来说，他们更倾向于使用图形界面式的开发方式，因为图形界面能更直观地展示整个开发过程。因此，作者在这里推荐使用 VS Code IDE 作为开发工具，该软件支持下载 ESP-IDF 插件，从而方便开发者进行项目开发和调试。ESP-IDF Eclipse 插件可便利开发人员在 VS Code 开发环境中开发基于 ESP32 的 IoT 应用程序。本插件集成了编辑、编译、烧录和调试等基础功能，还有安装工具、SDK 配置和 CMake 编辑器等附加功能，可简化并增强开发人员在使用标准 VS Code 开发和调试 ESP32 IoT 应用程序时的开发体验。

### 5.3.1 VS Code 安装

鉴于我们使用的是 VSCode IDE 搭配乐鑫 ESP-IDF 的方式进行开发，我们接下来便介绍一下 VSCode 的安装过程。

首先，进入 [VSCode 官方下载](#) 页面，根据系统需求选择下载安装包，也可以在 A 盘→6，软件资料→1，软件→1，IDF 开发工具→01-Windows 路径下找到 VSCode 安装包。下图是 VSCode 官方下载页面。

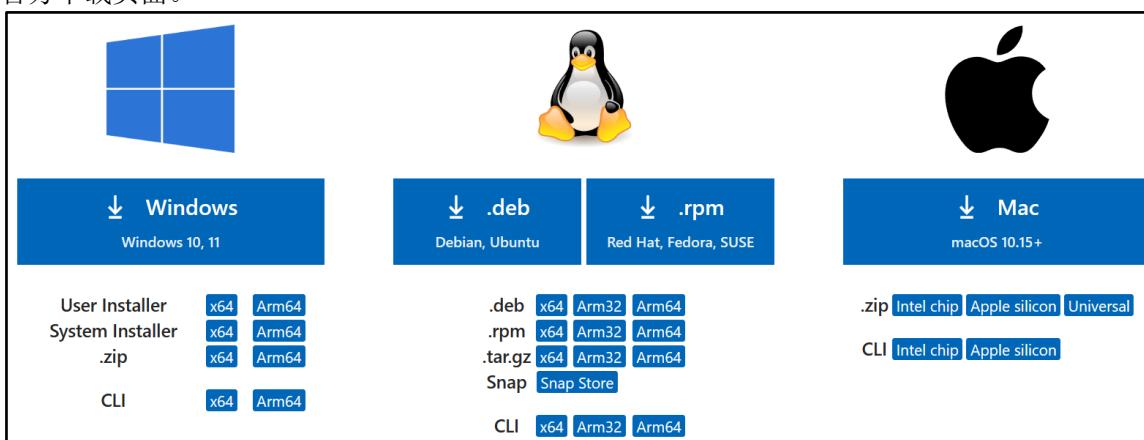


图 5.3.1.1 VSCode 官网下载界面

这里，我们选择 Windows 版本进行下载。因为，我们是在 Windows 环境下进行的开发，故在此介绍 Windows 版本的下载步骤。不出意外，其它版本的下载方式应该也是一样的。这里我们不多废话，直接点击下载。

下载完后，我们按照如下所示步骤进行即可：

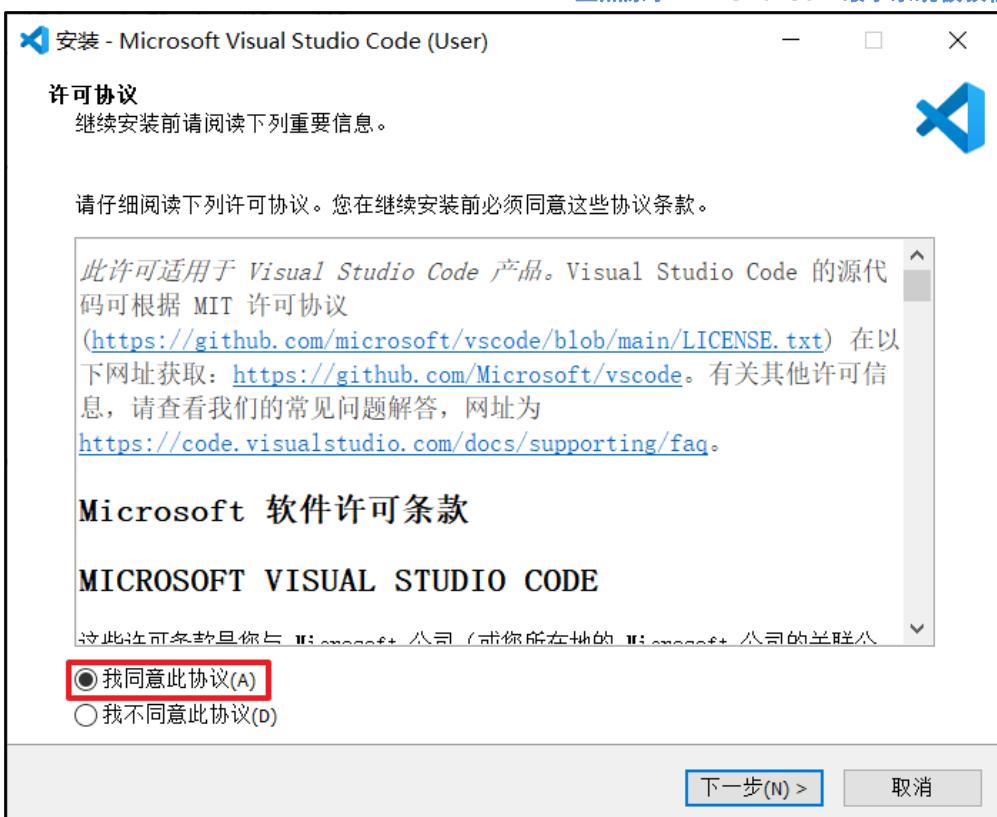


图 5.3.1.2 VSCode 下载步骤一



图 5.3.1.3 VSCode 下载步骤二

在该步骤中，路径如需更改的，请您点击“浏览”进行更改，**但请注意：修改的路径最好不要出现中文，以避免在往后的开发过程中遇到问题而导致重装软件，这对您来说就得不偿失了。**

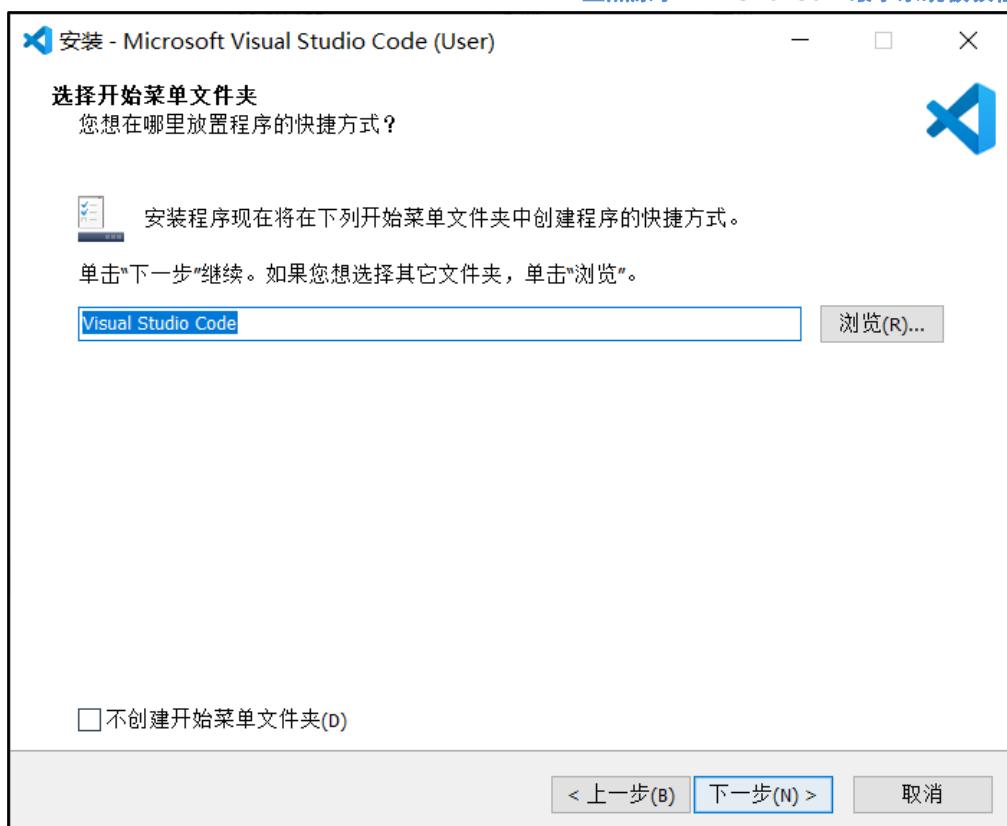


图 5.3.1.4 VSCode 下载步骤三

如需修改，同样点击“浏览”进行设置，无需修改的话直接点击“下一步”即可。

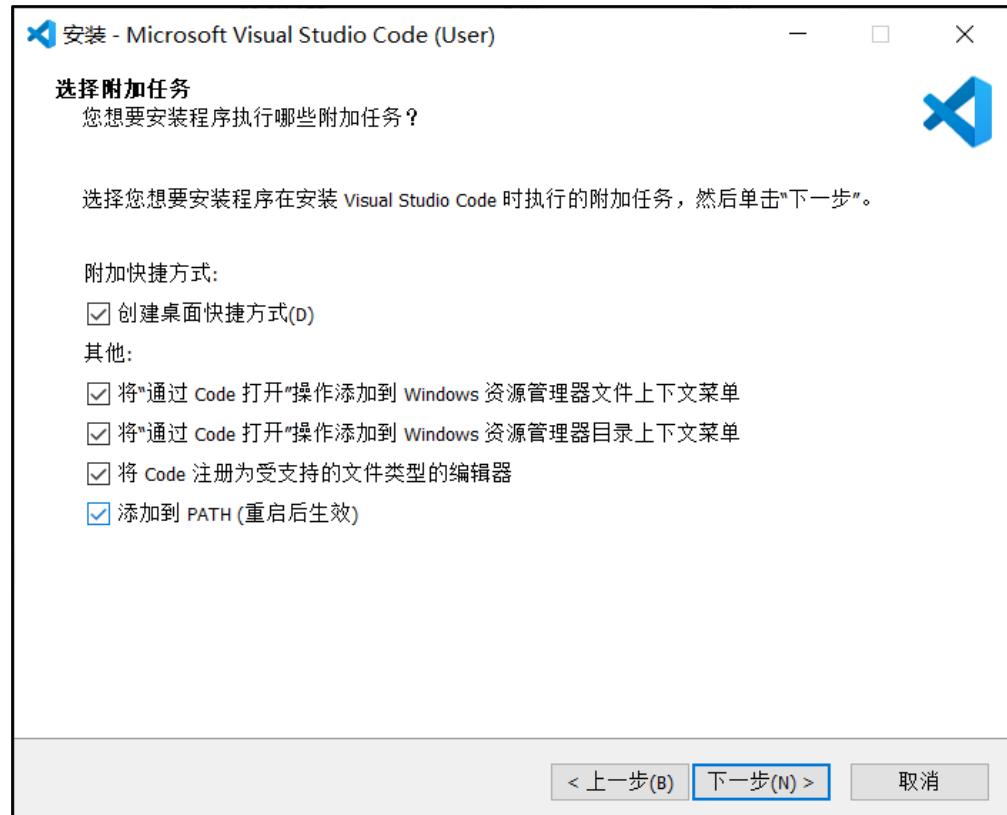


图 5.3.1.5 VSCode 下载步骤四

这一步骤同样是有需求的都勾上，我们建议是都勾上。

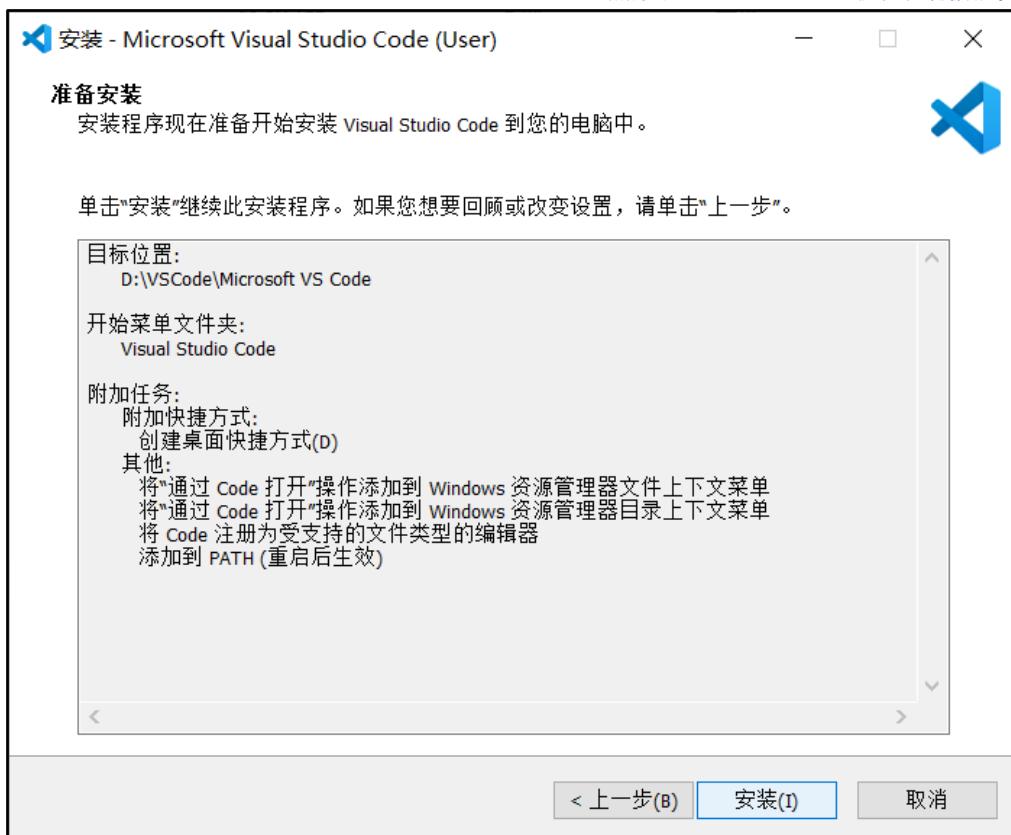


图 5.3.1.6 VSCode 下载步骤五

点击“安装”后，您只需静候佳音即可。



图 5.3.1.7 VSCode 下载步骤六

到这一步便可以开始运行 VSCode 了。打开 VS Code，在扩展商店的搜索区域输入“Chinese”安装中文插件，如下图所示：

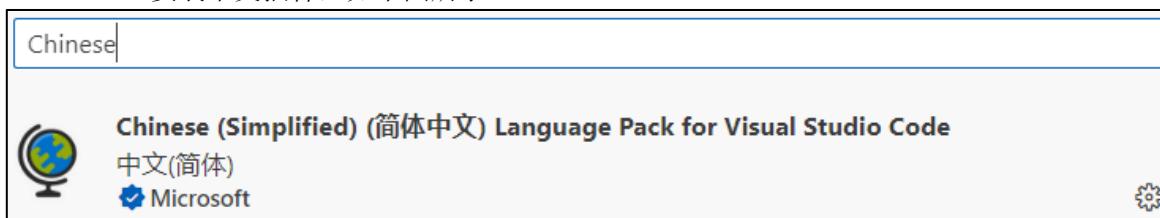


图 5.3.1.8 VSCode 中文界面设置

至此，VSCode 的安装与配置便算是大功告成了。感谢您能耐心看到此处。

### 5.3.2 安装与配置 ESP-IDF 插件

打开 VS Code 软件，然后按下快捷键“Ctrl+Shift+X”进入应用商城，在搜索栏下搜索 Espressif IDF 插件，点击安装即可：

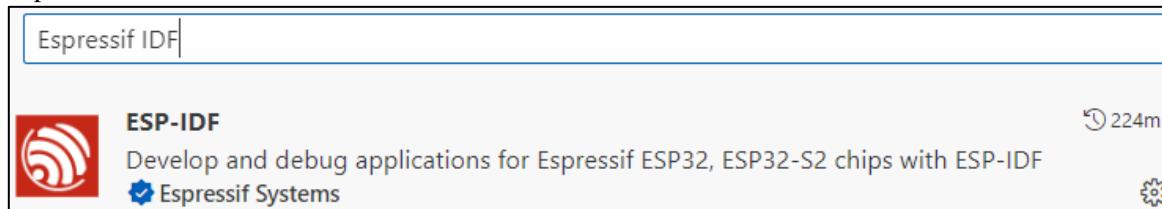


图 5.3.2.1 Espressif IDF 插件安装

至此 Espressif IDF 插件就算安装好了，接下来我们来看看插件的配置。

快捷键 `ctrl+shift+p` 呼出命令栏，在弹如下提示框后，搜索“配置 ESP-IDF 插件”，或者在使用快捷键 `ctrl+shift+p` 呼出命令栏后，在搜索框输入配置命令：Configure ESP-IDF。



图 5.3.2.2 配置 ESP-IDF 插件

回车后，进入配置 ESP-IDF 插件界面，如下图所示：

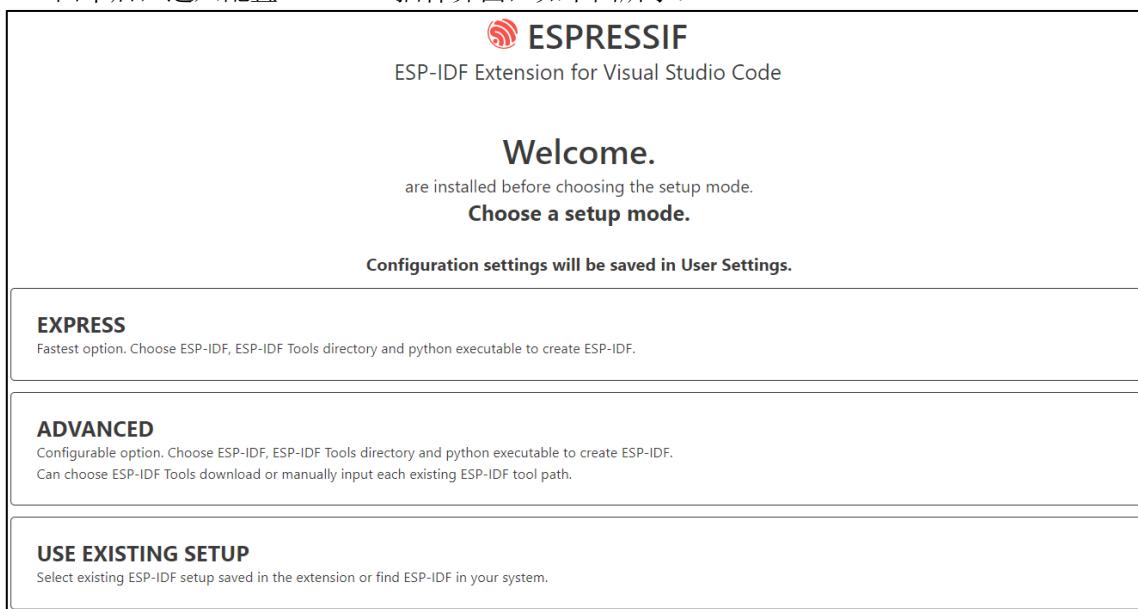


图 5.3.2.3 配置 ESP-IDF 插件界面

在上图中，点击“ADVANCED”进入高级配置界面，如下图所示：



图 5.3.2.4 配置 ESP-IDF 插件

配置 ESP-IDF 插件完成后，点击上图“Configure Tools”选项执行配置操作，此时需要等待系统配置成功，如下图所示：

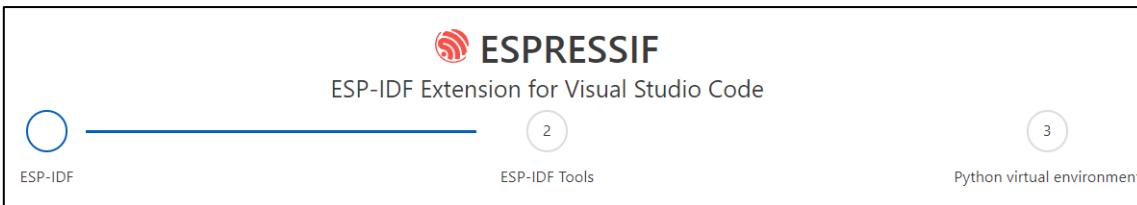


图 5.3.2.5 配置操作

注意：如果出现“python.exe -m pip is not valid”错误，大概是 python 环境搭建原因。请参考这位博主的[解决方案](#)。

从上图可以看到，配置 ESP-IDF 插件需要进行三个流程，等待第一个流程配置完成，此时进入 ESP-IDF Tools 配置流程，如下图所示：



图 5.3.2.6 下载 ESP-IDF Tools

接着点击“Download Tools”选项下载工具（**需要网络加持**），如下图所示：

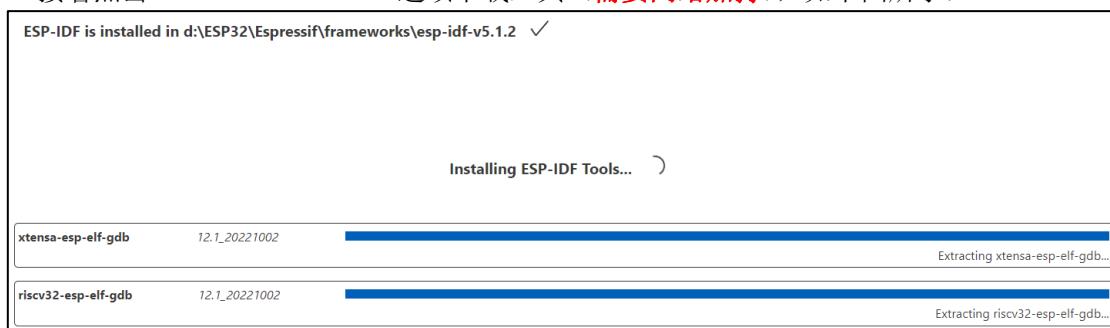


图 5.3.2.7 下载 ESP-IDF 工具

下载成功后，系统进入第三个流程 Python 环境搭建，如下图所示：

### Installing Python virtual environment for ESP-IDF...

图 5.3.2.8 安装 Python 环境

三个流程完成后，系统提示如下信息，如下图所示：



图 5.3.2.9 ESP-IDF 配置完成

接下来，作者将讲解插件默认的配置参数，如串口下载的波特率和下载方式，配置流程如下所示：

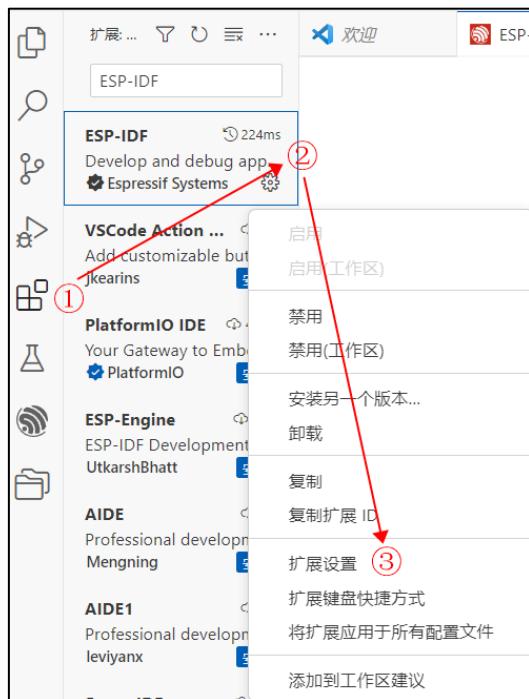


图 5.3.2.10 进入 ESP-IDF 插件配置

点击“扩展设置”选项，进入配置插件界面，然后找到“Flash Baud Rate”和“Flash Type”选项，如下图所示：



图 5.3.2.11 配置下载默认参数

到了这里，我们已经配置 ESP-IDF 插件完成。

### 5.3.3 个性化配置和工作环境配置

以 Visual Studio Code (VS Code) 为例，settings.json 文件存储了用户的个性化配置和工作环境设置，包括编辑器的外观、语言、代码格式化风格、自动补全行为、调试配置、扩展设置等。在 VS Code 中，可以通过用户设置（全局设置）和工作区设置（针对特定项目或文件夹的设置）来区分不同类型的配置。settings.json 文件配置流程如下。

#### ① 打开设置界面

我们按住“Ctrl +”快捷键，进入 settings.json 文件编写内容。

#### ② 添加个性化配置和工作环境设置参数

添加内容如下：

```
{  
    /* 上面的部分是我自己创建的一些设置 */  
    "editor.insertSpaces": false, /* 自动插入空格禁用 */  
    "editor.detectIndentation": false, /* 启用时根据文件内容进行重写 */  
    "editor.renderControlCharacters": true, /* 是否显示控制字符：启用 */  
    "editor.renderWhitespace": "all", /* 显示 4 个空格是.... */  
    "editor.tabSize": 4, /* tab 设置为 4 个空格 */  
    "editor.fontSize": 18, /* 代码字体大小 */  
    "editor.fontFamily": "Monaco, 'Courier New', monospace", /* 代码字体 */  
    "update.mode": "manual", /* 设置不自动更新 */  
}
```

至此，我们已成功安装 VS Code 集成开发环境的 ESP-IDF 插件，并完成了 VS Code 工作环境的配置。接下来，在下一章节中，作者将详细阐述如何在 VS Code IDE 中新建一个 ESP-IDF 工程，并进一步探讨工程的调试工具等相关内容。

## 第六章 新建基础工程

在前面的章节中，我们已经简要介绍了 ESP32-S3 的基础知识和 ESP-IDF 的基本概念，并详细阐述了 VS Code IDE 环境的搭建以及 Espressif 插件的安装流程。现在，基于这些前期准备，我们将在本章搭建一个 ESP-IDF 基础工程，以后的例程都是基于此基础例程为模版来编写的。

本章将分为如下几个小节：

- 6.1 搭建基础工程
- 6.2 基础工程的文件架构解析
- 6.3 乐鑫工程的文件架构解析
- 6.4 原子工程的文件架构解析
- 6.5 基础工程配置

### 6.1 搭建基础工程

在 VS Code 中新建 ESP-IDF 基础工程的步骤如下：

#### 1, 启动 VS Code 并打开命令面板

按下“Ctrl+Shift+P”快捷键打开命令面板，并在搜索栏内输入“新建项目”，如下图所示：

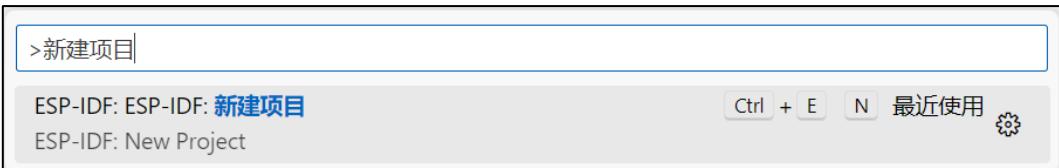
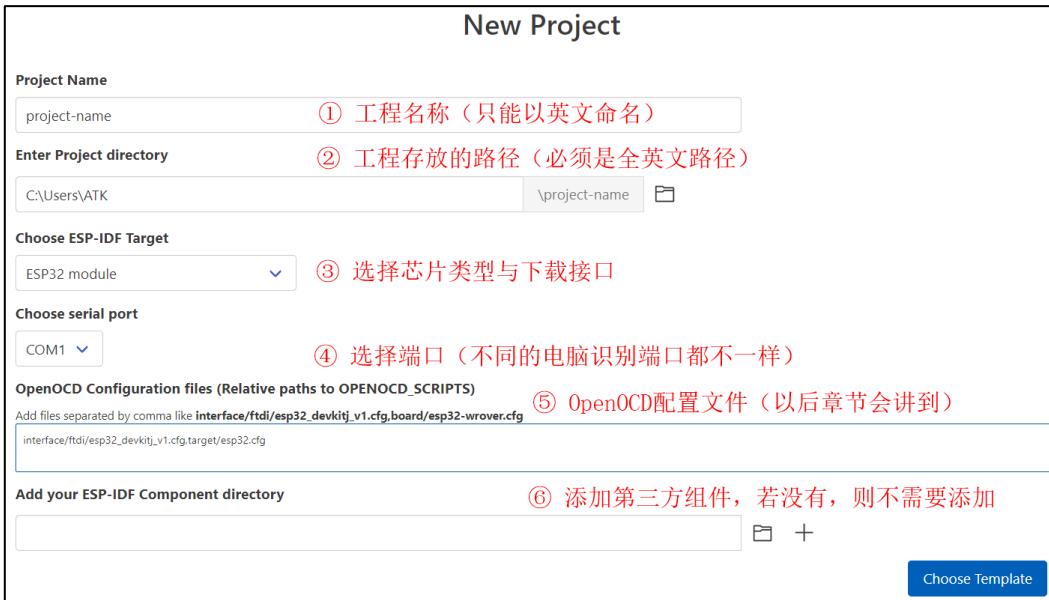


图 6.1.1 新建项目工程

#### 2, 配置工程参数

图 6.1.1 回车进入新建工程配置界面，如下图所示：



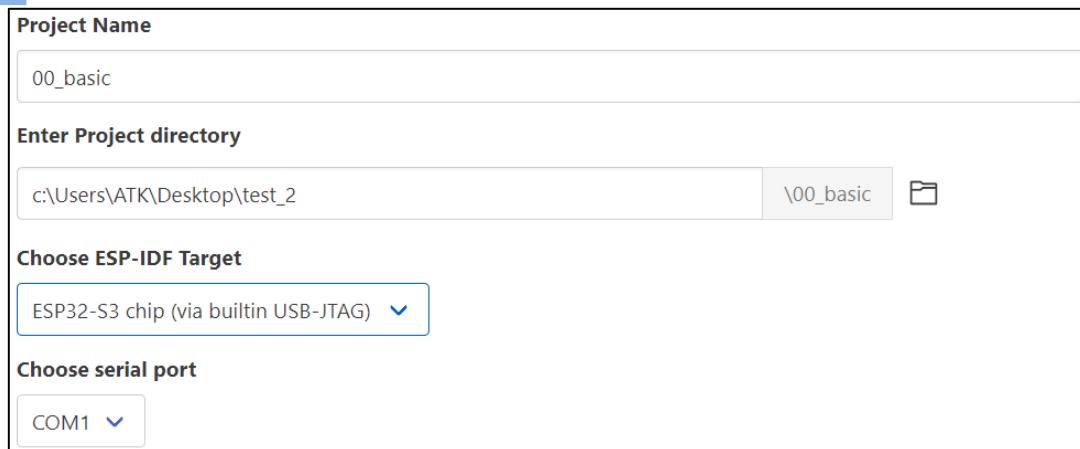


图 6.1.2 新建工程界面（下图是作者配置的参数）

### 3，选择模版工程

配置参数填写完成后，点击上图中的“Choose Template”选项，将进入选择模版界面。在这个界面上，您会发现许多应用实例可供选择（这些示例可以在路径 D:\ESP32\Espressif\frameworks\esp-idf-v5.1.2\examples 下找到）。为了新建工程，您可以采用某个应用示例作为模版，例如，若以乐鑫 Blink 跑马灯实验为模版，那么新创建的工程将具备 LED 使用功能。下面作者以 sample\_project 为模版新建工程（因为此工程是乐鑫官方提供的基本模板，所以我们使用此模板来新建工程），如下图所示：



图 6.1.3 以 blink 模版新建工程

点击“Create project using template sample\_project”选项新建工程，点击完成后在此界面的右下角跳出以下信息，如下所示：

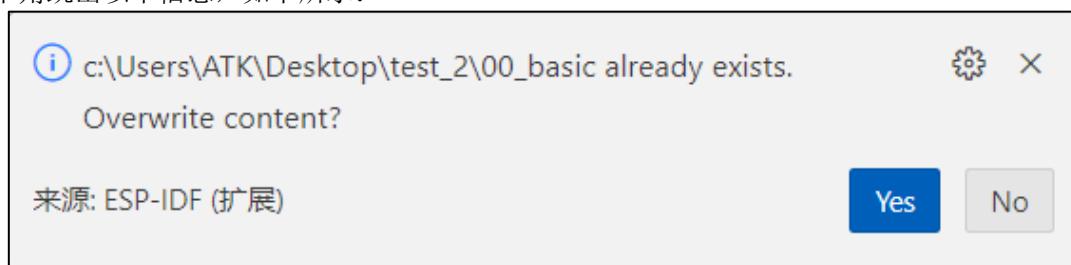


图 6.1.4 是否覆盖内容

我们点击“Yes”选项即可完成新建工程。此时 VS Code 的资源管理器区域内显示我们的新建工程，如下图所示：

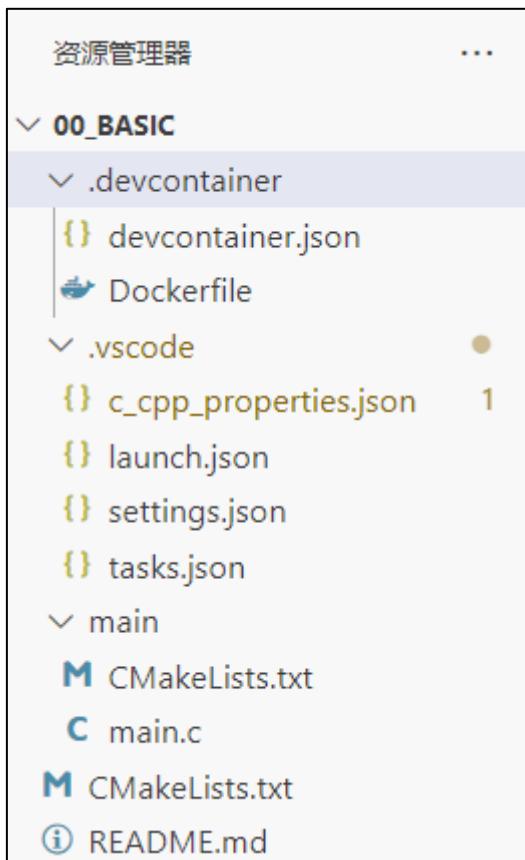


图 6.1.5 新建 00\_BASIC 工程完成

00\_BASIC 工程与乐鑫官方提供的 sample\_project 示例是一样的内容和工程结构，读者不妨对比一下这两个工程。

在下小节中，作者将讲述 00\_BASIC 工程的文件作用。

## 6.2 基础工程的文件架构解析

上图 6.1.5 中，.devcontainer 和.vscode 文件夹在 VS Code 的使用场景中各自扮演着不同的角色。

.devcontainer 文件夹通常与 VS Code 的 Remote - Containers 扩展一起使用，用于定义开发容器环境的配置。这个文件夹包含了用于创建和管理容器化开发环境的所有必需文件。

.vscode 文件夹通常位于项目的根目录下，用于存放 VS Code 的项目级设置和扩展配置。这个文件夹中的文件不会影响其他用户或全局的 VS Code 设置，它们只针对当前项目有效。在.vscode 文件夹中，常见的文件包括：

- ①：settings.json。用于定义项目特定的 VS Code 设置（个性配置和工作环境配置）。
- ②：tasks.json。用于定义任务，这些任务可以在 VS Code 的终端中运行，或者与编辑器中的其他功能（如代码片段）结合使用。
- ③：launch.json。用于配置调试器，包括启动配置和断点等。
- ④：c\_cpp\_properties.json。这个文件用于定义 C 和 C++ 项目的编译器路径、包含路径、编译器定义以及其他与 IntelliSense 相关的设置。

上述文件都是 VSCode 自动生成的，不需要人为去编写。但是在某种特殊情况下需要人为介入，如代码调试（需要修改 launch.json）、编译错误（需要修改 c\_cpp\_properties.json）和个性配置（需要修改 settings.json）等。剩下的文件就是 sample\_project (00\_BASIC) 工程的文件结构，如下图所示：

```

sample_project/
├── CMakeLists.txt      # 项目的主CMake文件，定义了项目的基本设置和组件
└── main/
    ├── CMakeLists.txt  # 主应用程序目录
    └── main.c          # 主应用程序的C源文件，通常包含app_main()入口函数
└── sdkconfig           # 项目的配置文件，记录了menuconfig中的配置选项

```

图 6.2.1 00\_BASIC 工程的文件结构和说明

乐鑫为开发者提供了最基础的项目工程，即 sample\_project。正点原子则以这一工程为模板去编写其他的应用示例。因此，关于这一基础工程文件的具体作用，读者可在第四章的 4.4 小节中详细了解。为避免重复，此处不再赘述。

### 6.3 调试相关工具介绍

在 6.1 小节中，作者已经在 VS Code 环境下新建一个 00\_BASIC 工程，下面我们重点来讲解 VS Code 软件提供用户调试相关的工具有哪些，如下所示：



图 6.3.1 调试相关工具

1，选择串口（插头）：即连接开发板的下载串口号，VS 会列出当前连接电脑的所有串口让你选择，这个会记录，再新打开 VSCode 不用重新选择，开发过程中尽量不要更换 USB 线的电脑插口，否则串口号会变。

2，选择目标芯片：对应 idf 命令 idf.py set-target xxxx。即你当前这个工程是要下载到什么芯片上面，如 ESP32 S2,S3,C2,C3 等等，工程要与芯片相匹配，这个选择是写入当前工程配置的，一般不用更改，工程下配置文件基本已经选择好的。

3，选择当前工程目录（文件夹）：也不用修改，一般打开工程时会默认操作都在这个工程目录下。

4，工程配置菜单（齿轮）：对应 idf 命令 idf.py menuconifg，用来配置当前工程的一些设置，配置项非常多，建议使用到再修改。一般代码工程都是配置好的，且不用修改。

5，清除工程（垃圾桶）：清除工程编译文件，一般用于压缩拷贝工程文件时用到，清除后工程目录占用空间会占用非常小，KB 级，编译后为百 MB 级，还有一些编译过程中奇奇怪怪的问题也可以先清除编译后再编译。

6，编译工程（圆柱体）：编译当前工程，只是编译，没有下载功能。

7，选择下载模式（五角星）：一般都是选择串口 UART 方式下载。

8，下载（闪电）：下载编译好的固件到设备芯片上，这里只是下载，没有编译功能，修改代码后要先编译再点这个下载，所做的修改才有效。

9，串口监控（小电视）：打开与设备连接的串口，打印设备串口信息。

10，编译/下载/监控（一团火）：最常用的一个，它将编译下载和打开串口监控做在了一起，点一次全部搞定。

11，打开命令行：打开命令行窗口，且会定位在当前项目路径下，可以执行 idf 的一些命令。

12，执行自定义任务：不使用。

13，工程的错误与警告提示。

以上是 ESP-IDF 插件提供的调试工具，一般我们只用到 1、2、4、5、6 和 8 即可完成程序开发。ESP32-S3 有两种下载方式。首先是 USB 串口下载，这种方式主要用于代码下载，但无法用于代码调试。另一种则是 JTAG（USB 口）下载，它不仅能用于下载代码，还支持代码调试。接下来，作者将详细解释这两种下载方式。

### 6.3.1 串口下载

下面，作者以 6.1 小节新建工程（00\_BASIC）为例，简单讲解一下工程的下载流程，如下流程所示。

①：使用 USB 线的 Type-C 接口连接 DNESP32S3M 最小系统板的 USB 串口，并 USB A 口连接到电脑，使得电脑与 DNESP32S3M 最小系统板建立连接。

②：在设备管理器中，查看 USB 串口的端口号，并在 VS Code 软件左下角调试区域设置端口号（插头）。

③：点击“Set Espressif Device Target”选择目标芯片，这里我们选择 ESP32-S3。

④：选择“select flash Method”下载方式（五角星），如 UART 或者 JTAG

⑤：点击“Full Clean”擦除工程（垃圾桶）。

⑥：点击“Build Project”编译工程（圆柱形）。

⑦：编译完成后，点击“ESP-IDF: Flash Device”下载代码（闪电）。

编译工程成功后，工程目录下出现 build 文件夹，这个文件夹是由 ESP-IDF 编译器产生的文件，如 log、固件、map 等下载和调试文件。如下图所示：

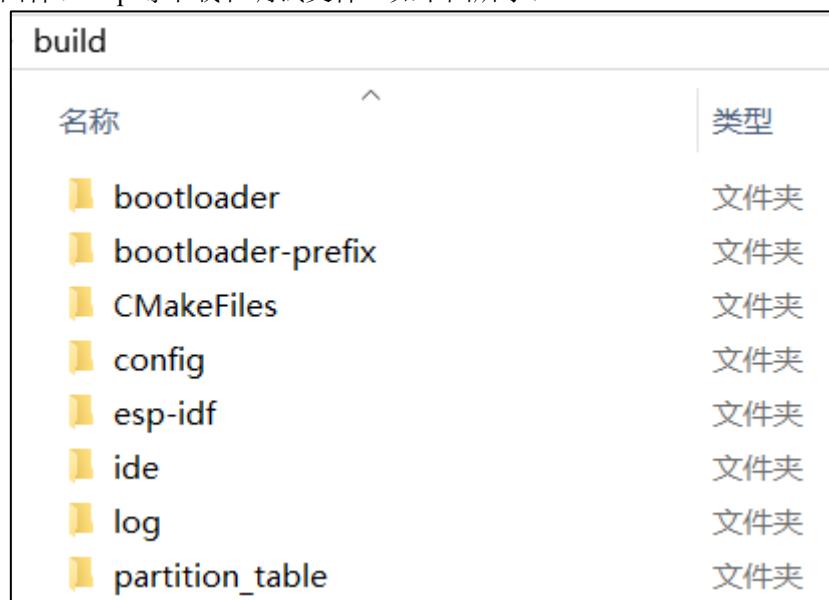


图 6.3.1.1 编译输出文件（部分截图）

编译完成后，VSCode 软件底部的 Build Output 窗口会输出编译信息，如下图所示：

```
Total sizes:
Used static IRAM:      54146 bytes ( 308094 remain, 14.9% used)
    .text size:   53119 bytes
    .vectors size: 1027 bytes
Used stat D/IRAM:     12500 bytes ( 333356 remain, 3.6% used)
    .data size:   10364 bytes
    .bss size:    2136 bytes
Used Flash size : 136735 bytes
    .text:        96311 bytes
    .rodata:      40168 bytes
Total image size: 201245 bytes (.bin may be padded larger)
```

图 6.3.1.2 编译 00\_BASIC 工程

由上图可以看到，当出现以上信息后便证明工程编译成功，这个过程可能会持续 2~3 分钟，快的话 1 分钟也是可以的（电脑配置越高，编译就越快），请您耐心等待。

上图中也可以查到本次编译的许多信息，这里介绍终端界面下的几个重要信息：

①: Used static IRAM: 为了与 ESP32-S3 目标兼容, 保留了这些选项, 当前读取为 54146 bytes (.text size + .vectors size)

②: .text size: 文本占用空间大小 (53119bytes)

③: .vectors size: 矢量大小 (1027 bytes)

④: Used stat D/IRAM: 这是内部 RAM 的总使用量, 静态 DRAM .data + .bss 的总和, 以及应用程序用于可执行代码的静态 IRAM (指令 RAM)。可用大小是运行时作为堆内存可用的 DRAM 的估计量 (由于元数据开销和实现限制, 以及 ESP-IDF 在启动期间完成的堆分配, 启动时的实际可用堆将低于此值)。

⑤: .data size: 是静态分配的 RAM, 在启动时分配给非零值。这在运行时使用 RAM (DRAM), 并且还使用二进制文件中的空间。

⑥: .bss size: 是静态分配的 RAM, 在启动时分配为零。这在运行时使用 RAM (DRAM), 但不使用二进制文件中的任何空间。

⑦: Used Flash size: 这表示项目在编译后将使用的 Flash 内存的大小, 但不包括 DRAM 和 IRAM 的使用量。

⑧: .text: 这部分用于表示.text 的 Flash 大小。

⑨: .rodata: 这部分用于表示.rodata 的 Flash 大小。

⑩: Total image size: 是二进制文件的预估总大小。

编译成功后, 点击图 6.3.1 闪电图标就可以把编译出来的可执行文件烧录至 ESP32S3 最小系统板上, 如下图所示:

```
esptool.py v4.7.0
Serial port COM9
Connecting....
Chip is ESP32-S3 (QFN56) (revision v0.2)
Features: WiFi, BLE, Embedded PSRAM 8MB (AP_3v3)
Crystal is 40MHz
MAC: dc:da:0c:18:d2:e0
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
```

图 6.3.1.3 下载程序至开发板中

### 6.3.2 JTAG 下载与调试

本小节的内容是参考乐鑫 ESP-IDF 编程指南的 JTAG 调试章节。

ESP32-S3 内置 JTAG 电路, 因此无需额外芯片即可实现调试功能。通过简单地将 USB 线连接到其 D+/D- 引脚, 即可轻松完成下载与调试操作。利用 JTAG 接口, 开发人员能够运用开源工具 OpenOCD 对 ESP32-S3 进行调试。OpenOCD 专为嵌入式系统开发和调试设计, 可连接到目标硬件的调试接口 (如 JTAG 或 SWD), 支持调试、固件烧写等硬件相关任务。根据乐鑫官方资料, JTAG 下载提供两种方式: 一种为直接利用内置的 JTAG 电路进行调试; 另一种则是借助乐鑫官方推出的 ESP-PROG 调试器, 它集成了自动下载固件、串口通信以及 JTAG 在线调试等多项功能。此外, 在 VS Code 中安装 OpenOCD 时, 它会自动为我们配置好所需环境, 使得我们可以直接使用内置的 JTAG 电路和 VS Code 中的 OpenOCD 来下载与调试 ESP32-S3 芯片, 进一步简化了开发流程。

#### 1. 前期准备

JTAG 下载与调试之前, 我们必须修改 launch.json 和 settings.json 这两个文件。其中 launch.json 文件用来配置调试器。

①: 进入乐鑫官方提供的 [VS Code 调试配置文件 launch.json 网址](#)。在该网址下, 有详细的说明指导我们如何根据使用 JTAG 调试器或使用 VS Code 进行调试, 来相应地修改 launch.json 的

内容。由于我们选择使用 VS Code 进行调试，因此 launch.json 的修改内容如下。

```
{
  "configurations": [
    {
      "name": "GDB",
      "type": "cppdbg",
      "request": "launch",
      "MIMode": "gdb",
      "miDebuggerPath": "${command:espIdf.getToolchainGdb}",
      "program":
        "${workspaceFolder}/build/${command:espIdf.getProjectName}.elf",
      "windows": {
        "program":
          "${workspaceFolder}\\build\\${command:espIdf.getProjectName}.elf"
      },
      "cwd": "${workspaceFolder}",
      "environment": [{ "name": "PATH", "value": "${config:idf.customExtraPaths}" }],
      "setupCommands": [
        { "text": "target remote :3333" },
        { "text": "set remotetimeout 20" },
      ],
      "postRemoteConnectCommands": [
        { "text": "mon reset halt" },
        { "text": "maintenance flush register-cache" },
      ],
      "externalConsole": false,
      "logging": {
        "engineLogging": true
      }
    }
  ]
}
```

②：在 settings.json 文件下找到“idf.openOcdConfigs”配置选项，我们把该选项的内容修改为一下内容。

```
"idf.openOcdConfigs": [
  "interface/ftdi/esp32_devkitj_v1.cfg",
  "target/esp32.cfg"
],
```

修改为：

```
"idf.openOcdConfigs": [
  "board/esp32s3-builtin.cfg",
],
```

到了这里，我们已经配置 VS Code OpenOCD 完成，下面作者来讲解使用 JTAG 下载与调试操作。

③：cpp\_properties.json 文件下添加以下红色内容，解决编译过程中总会出现某些文件无法找到头文件的错误信息提醒（有时会是警告）。

```
{
  "configurations": [
    {
      "name": "ESP-IDF",
      "includePath": [
        "${config:idf.espIdfPath}/components/**",
        "${config:idf.espIdfPathWin}/components/**",
        "${config:idf.espAdfPath}/components/**",
        "${config:idf.espAdfPathWin}/components/**",
        "${config:idf.espAdfPathWin}/components/**",
        "${workspaceFolder}/**"
      ],
      "browse": {
        "path": [
          "${config:idf.espIdfPath}/components",
          "${config:idf.espIdfPathWin}/components",
          "${config:idf.espAdfPath}/components/**",
        ]
      }
    }
  ]
}
```

```

        "${config:idf.espAdfPathWin}/components/**",
        "${workspaceFolder}"
    ],
    "limitSymbolsToIncludedHeaders": false
},
/* 解决 browse.path 中未找到包含文件 */
"configurationProvider": "ms-vscode.cmake-tools"
}
],
"version": 4
}

```

## 2, JTAG 下载程序

JTAG 下载程序流程如下：

使用 USB 线的 Type-C 接口连接 DNESP32S3M 最小系统板右下角的 USB 口，并 USB A 口连接到电脑，使得电脑与 DNESP32S3M 最小系统板建立连接。

- 设置 USB JTAG 接口的端口号（电脑自动识别）。
- 选择“select flash Method”下载方式（五角星），这里我们选择 JTAG 下载。
- 点击“Full Clean”擦除工程（垃圾桶）。
- 点击“Build Project”编译工程（圆柱形）。

编译成功后，点击“Flash Device”下载程序至 DNESP32S3M 最小系统板中，此时，VS Code 提示是否运行 OpenOCD，如下图所示：

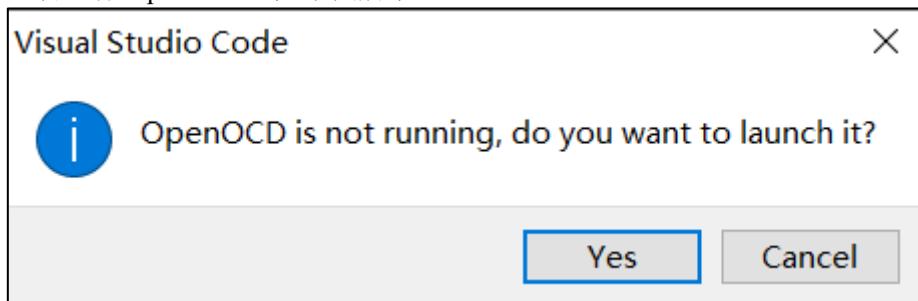


图 6.3.2.1 运行 OpenOCD

下载成功后，VS Code 右下角提示如下信息。



图 6.3.2.2 提示下载成功

## 3, JTAG 调试

JTAG 调试非常简单，先把代码下载至 DNESP32S3M 最小系统板，然后点击“运行与调试”，如下图所示：

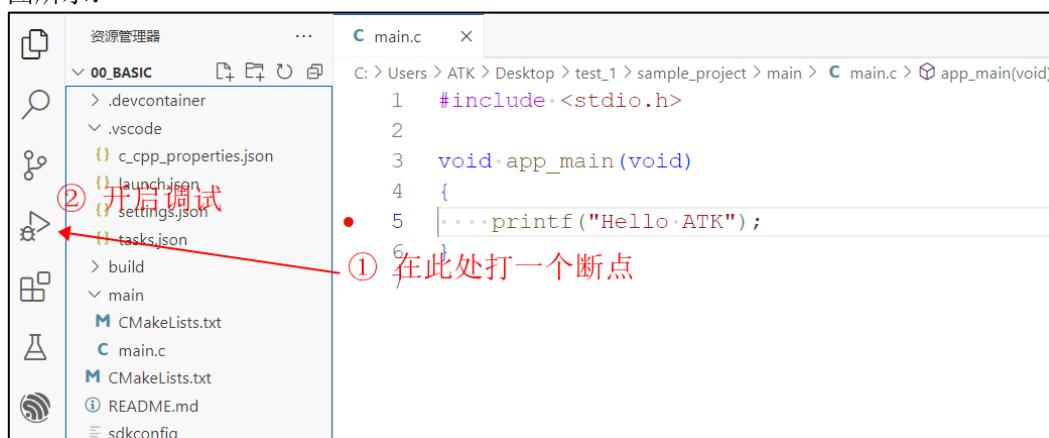


图 6.3.2.3 点击运行与调试

打开运行与调试界面如下图所示：



图 6.3.2.4 点击运行调试

此时，系统执行到我们刚刚定义的断电处，如下图所示：

```
main > C main.c > app_main(void)
1 #include <stdio.h>
2
3 void app_main(void)
4 {
    // 程序当前执行的位置
5     printf("Hello ATK");
6 }
7
```

图 6.3.2.5 调试效果

在上图中，右上角提供了用于调试代码的选项。由于这些功能在学习 MDK 时都有所涉及并经常使用，因此作者在此不再过多介绍这些调试选项。

#### 4. 调试方法

下面，作者简单介绍一下 VS Code 调试方法，例如变量怎么加载至监视，反汇编如何打开等实用操作。

##### 变量监控：

打开调试时，选择某个变量右键选择“添加到监视”，如下图所示：

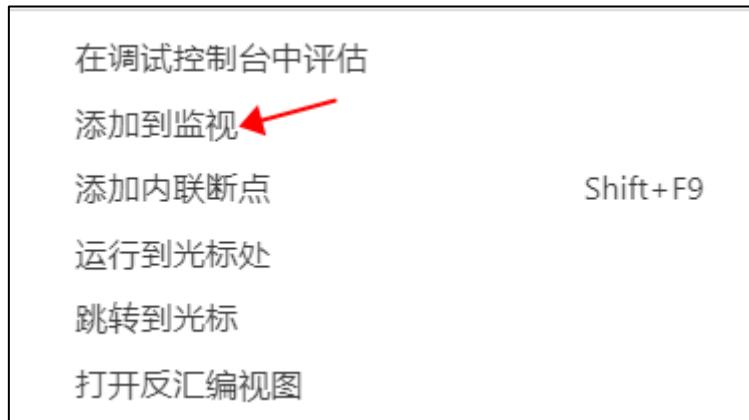


图 6.3.2.6 变量添加到监视器中

此时，运行与调试界面显示该变量的当前数值，如下图所示：

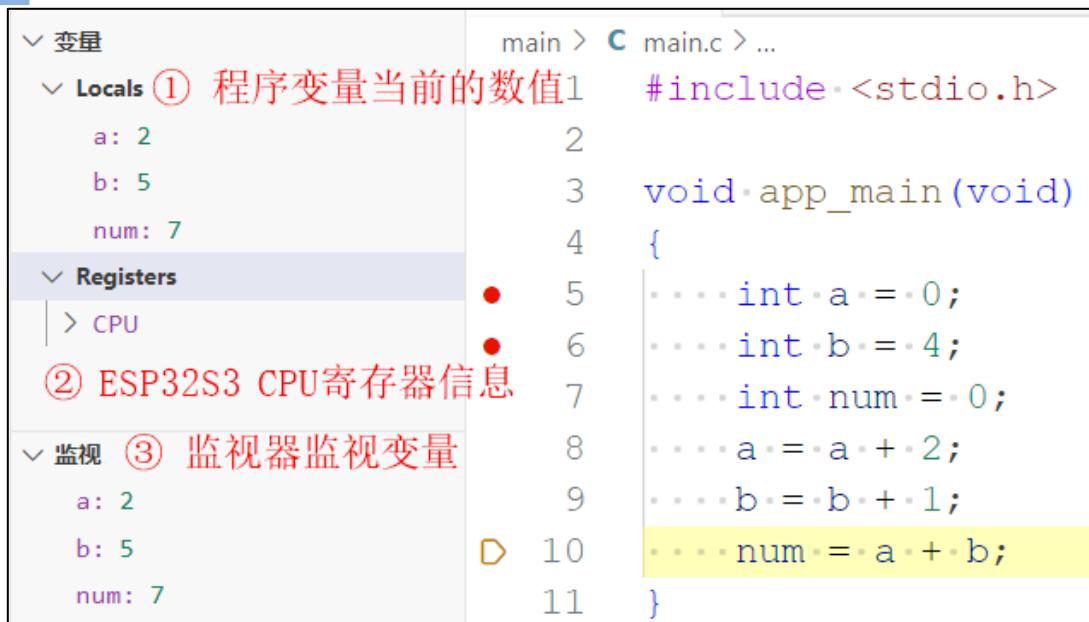


图 6.3.2.7 监视变量和监视 CPU 寄存器

在图 6.3.2.6 中，我们可以打开“反汇编视图”，用户可以查看程序的反汇编指令，理解程序的执行流程，以及分析程序的结构和逻辑，如下图所示：

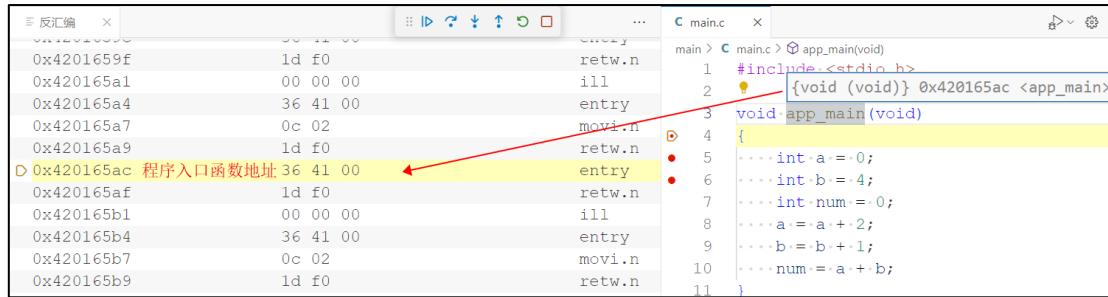


图 66.3.2.8 反汇视图查看函数地址

除了上述的调试方法，大家还需亲自动手实践，才能真正理解 VS Code 的调试流程。请注意，在使用 VS Code 对 ESP32 进行调试时，可能会遇到一些问题。例如，如果调试失败，可能是因为需要先使用 JTAG 接口下载程序才能进行调试。另外，调试结束后，有时需要重新编译代码才能再次成功调试。最后，调试过程中可能会出现不稳定的情况，如自动断开，这可能是由于 VS Code 的 OpenOCD 与 ESP32-S3 芯片内置的 JTAG 连接不稳定所致。为了解决这些问题，建议使用乐鑫官方的 JTAG 调试器。

## 6.4 原子工程的文件架构解析

在讲解正点原子 ESP32 工程架构之前，我们先了解一下乐鑫 ESP32 工程的工程架构，如下所示：

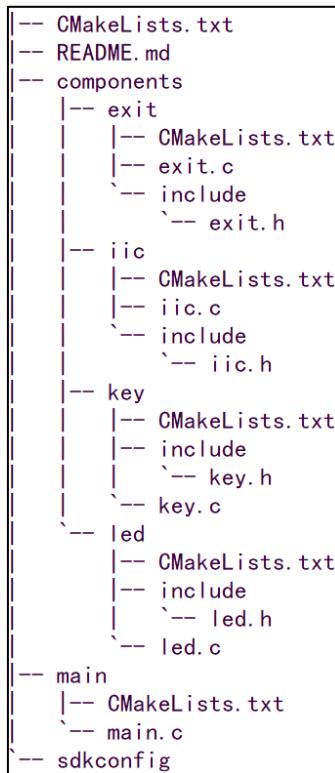


图 6.4.1 乐鑫 ESP32 工程文件架构

从上图可以清晰地看出，除了 components 文件夹外，该工程架构与我们在 6.1 小节中新建的架构是一致的。components 文件夹主要用于存放第三方驱动库和开发者编写的驱动库。然而，如果我们采用类似于乐鑫 ESP32 的工程架构进行开发，这可能会导致我们的工程架构变得相当混乱。因此，正点原子采取了一个不同的做法，将开发者编写的驱动文件整理到 components 文件夹下的 BSP 文件夹中。此外，我们还简化了 CMakeLists.txt 文件的管理，通过保留一个 CMakeLists.txt 文件，就能够加载多个驱动库，从而提高了工程的整洁性和可维护性。

下图是正点原子 ESP32 工程的架构。

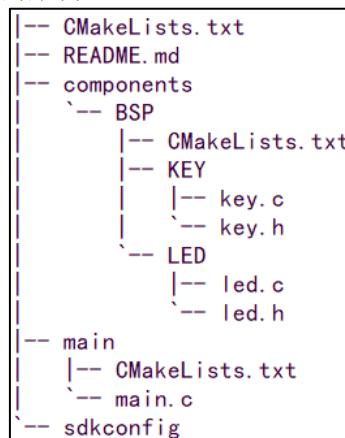


图 6.4.2 正点原子 ESP32 文件架构

从上图中我们可以清晰地观察到，正点原子将所有开发者编写的程序驱动都统一地放置在 BSP 文件夹下。为了对这些程序驱动实施高效的统一管理，它引入了一个 CMakeLists.txt 文件。这种做法的显著优势在于，当需要创建新的驱动文件时，我们无需像乐鑫 ESP32 工程那样，为每一个驱动代码都单独创建一个 CMakeLists.txt 文件。相反，我们只需使用一个 CMakeLists.txt 文件即可完成所有相关操作，极大地简化了整个开发流程。接下来，我们将深入探究我们提供的 CMakeLists.txt 文件，其代码内容如下：

① 源文件路径，指本目录下的所有代码驱动

```
set(src_dirs
```

```
IIC
LCD
LED
SPI
XL9555
KEY
24CXX
ADC
AP3216C
QMA6100P)
```

② 头文件路径，指本目录下的所有代码驱动

```
set(include_dirs
    IIC
    LCD
    LED
    SPI
    XL9555
    KEY
    24CXX
    ADC
    AP3216C
    QMA6100P)
```

③ 设置依赖库

```
set(requires
    driver
    fatfs
    esp_adc
    esp32-camera
    newlib
    esp_timer)
```

④ 注册组件到构建系统的函数

```
idf_component_register(SRC_DIRS ${src_dirs} INCLUDE_DIRS
    ${include_dirs} REQUIRES ${requires})
```

⑤ 设置特定组件编译选项的函数

```
component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
-ffast-math: 允许编译器进行某些可能减少数学运算精度的优化，以提高性能。
-O3: 这是一个优化级别选项，指示编译器尽可能地进行高级优化以生成更高效的代码。
-Wno-error=format: 这将编译器关于格式字符串不匹配的警告从错误降级为警告。
-Wno-format: 这将完全禁用关于格式字符串的警告。
```

在开发过程中，④和⑤是固定不变的设定。而①和②的确定则依赖于项目所需的驱动文件数量。如果当前目录下缺少某个特定的驱动文件（例如 LED 驱动文件），但在 CMakeLists.txt 文件中却指定了需要编译该 LED 驱动文件，那么在系统编译时将会遇到以下错误：

```
CMake Error at D:/Soft_APP/ESP32/Espressif/frameworks/esp-idf-v5.1.2/tools/cmake/
SRC_DIRS entry 'LED' does not exist.
Call Stack (most recent call first):
D:/Soft_APP/ESP32/Espressif/frameworks/esp-idf-v5.1.2/tools/cmake/component.cma
components/BSP/CMakeLists.txt:12 (idf_component_register)
```

图 6.4.3 未发现 LED 驱动文件

此时，需要我们添加 LED 驱动文件，并且清除编译工程文件才能再一次编译工程。

③表示驱动程序需要依赖的库，例如 CAMERA 驱动程序，它依赖的是 esp32-camera 这一个摄像头驱动库。下图是 CMakeLists.txt 未添加依赖库的错误提示。

```
In file included from C:/Users/ATK/Desktop/25_1_camera/components/BSP/CAMERA/camera.c:22:
C:/Users/ATK/Desktop/25_1_camera/components/BSP/CAMERA/camera.h:29:10: fatal error: esp_camera.h: No such file or directo
29 | #include "esp_camera.h"
| ^~~~~~
```

图 6.4.4 未添加依赖库的提示错误（CAMERA 驱动程序未添加依赖库）

## 6.5 基础工程配置

在第三章节中，我们得知正点原子是以 ATK-MWS3S 模组作为主控，该模组的设计参考了乐鑫的 ESP32-S3-WOOD-1 的 N16R8 型号，因此它与乐鑫产品可实现 P2P 兼容。然而，在 6.1 小节新建工程时，我们并未立刻适配这款模组的内部资源，例如时钟频率是否设定为 240MHz、

Flash 配置是否为 16MB，以及 PSRAM 的模式是否为 OCT 和 8MB 等。这些参数需要由开发者自行配置，否则我们的工程只能使用乐鑫的默认配置进行开发，这样模组便无法充分发挥其应有的性能。下面，作者将逐步引导读者进行基础工程的配置，确保其与正点原子发布的 DNESP32S3M 最小系统板的主控模组内部资源相匹配。配置流程如下：

1，使用 VS Code 打开 6.1 小节新建的工程，并点击左下角齿轮（ESP-IDF: SDK Configuration Editor (menuconfig)）进去 menuconfig 菜单配置界面，如下图所示：

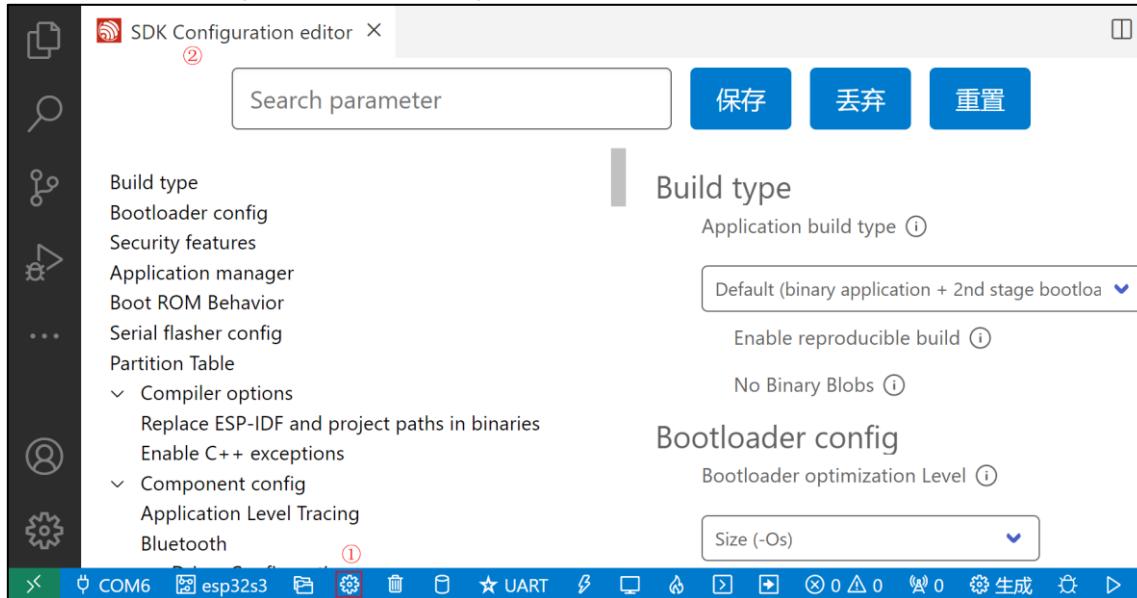


图 6.5.1 进入 menuconfig 配置界面

关于 menuconfig 配置界面的具体内容，作者将在后续的章节中进行详细讲解。目前，我们的主要目标是配置与 ATK-MWS3S 模组相匹配的资源。

2，在上图“Search parameter”搜索框下输入“Flash”进去 flash 配置界面，配置内容如下所示：



图 6.5.2 配置 Flash 资源

上图中的①“Flash SPI mode”支持四种不同的 SPI flash 访问模式，它们分别为 DIO、DOUT、QIO 和 QOUT。下面我们来看一下这几种模式到底有哪些区别，这些模式的对比如下表所示：

可选项	模式名称	引脚	速度
QIO	Quad I/O	地址和数据 4pins	最快

QOUT	Quad Output	数据 4pins	约比 qio 模式下慢 15%
DIO	Dual I/O	地址和数据 2pins	约比 qio 模式下慢 45%
DOUT	Dual Output	数据 2pins	约比 qio 模式下慢 50%

表 6.5.1 四种 SPI 模式的对比

从上表可知，QIO 模式的速率为最快，所以我们把基础工程的 Flash SPI mode 设置为 QIO。

上图中的②“Flash SPI speed”提供了 120、80、40 和 20MHz 的配置选项。在选择具体速度时，我们需要考虑 Flash 和 PSRAM 的 SPI 接口共享情况。为了优化模组性能，我们最好将 flash 和 PSRAM 的 SPI 速率设置为一致，这样分时访问这两个存储设备时，就不必切换时钟频率了。鉴于 PSRAM 的 SPI 速率最高可设置为 80MHz，因此我们将 flash 的 SPI 速率也设置为 80MHz，以确保最佳性能。

上图的③是根据模组挂载的 flash 来确定的，这里我们选择 16MB 大小，是毫无争议的。

3，在搜索框中输入“Partition Table”来设置分区表。分区表的主要功能是将 flash 划分为多个功能各异的区域，包括存储启动文件、代码区域和文件系统区域等子分区，以满足不同的应用需求。后续章节将详细解释分区表的作用和配置方法。下图是基础工程分区表配置参数。

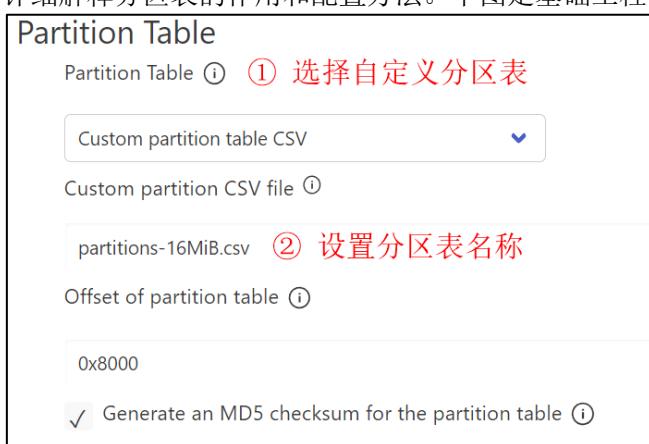


图 6.5.3 配置分区表

上图中，我们选择“Custom partition table CSV”自定义分区表，然后设置分区表的名称为 partitions-16MiB.csv。稍后我们会设置分区表各个子分区的管理大小。

4，在搜索框中输入“PSRAM”来设置 PSRAM 参数，配置参数如下图所示：

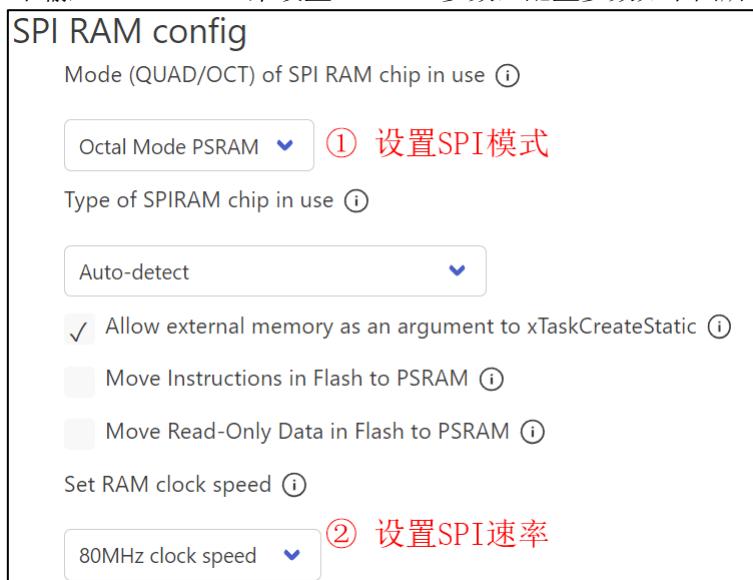


图 6.5.4 设置 PSRAM 参数

上图的①选项是基于模组内部芯片的选择来确定的。为了正确配置，读者可以查阅《esp32-s3-wroom-1\_wroom-1u\_datasheet\_cn》数据手册的第三页。在该页中，我们可以看到

ESP32-S3-WROOM-1-N16R8 模组所挂载的 PSRAM 使用的是 Octal SPI 模式。因此，在配置过程中，我们应该选择“Octal Mode PSRAM”这一选项。

上图的②选项选择最该的速率即可，并且与 Flash SPI 速率一致。

5，在搜索框中输入“CPU frequency”来设置 CPU 的时钟频率，如下图所示：



图 6.5.5 配置 CPU 主频

6，在搜索框中输入“FreeRTOS”来配置系统节拍时钟（tick clock）的频率。默认情况下，“configTICK\_RATE\_HZ”的值为 100，意味着节拍时钟的周期为 10ms。因此，调用 vTaskDelay(1000)将会导致延时 10 秒。为了提高定时精度和方便性，建议将该值设置为 1000，这样节拍时钟的周期就变为 1ms，从而使得 vTaskDelay(1000)代表延时 1 秒。如下图所示：

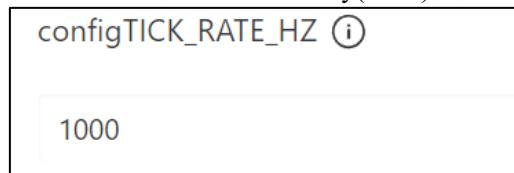


图 6.5.6 配置系统节拍时钟（tick clock）的频率

7，配置分区表各个子分区，我们按下“Ctrl+Shift+P”快捷键打开命令面板，并在搜索栏内输入“打开分区表编辑器”，按以下图配置各个分区的管理大小。

Name	Type	Sub Type	Offset	Size	Encrypted
nvs	data	nvs	0x9000	0x6000	<input type="checkbox"/>
phy_init	data	phy	0xf000	0x1000	<input type="checkbox"/>
factory	app	factory	0x10000	0x1F0000	<input type="checkbox"/>
vfs	data	fat	0x200000	0xA00000	<input type="checkbox"/>
storage	data	spiffs	0xc00000	0x400000	<input type="checkbox"/>

图 6.5.7 设置分区表

首先我们按下“Add New Row”选项添加子分区条目，然后设置条目的类型、偏移和大小，最后按下“Save”选项保存退出。

至此，我们已经完成了工程配置，确保其与 DNESP32S3M 最小系统板所搭载的 ATK-MWS3S 模组的内部资源相匹配。现在，我们可以利用这个工程在 DNESP32S3M 最小系统板上进行开发工作。下面我们来看一下当前工程架构，如下图所示：

```

|-- CMakeLists.txt
|-- README.md
|-- main
|   |-- CMakeLists.txt
|   `-- main.c
|-- partitions-16MiB.csv
|-- sdkconfig
`-- sdkconfig.old

```

图 6.5.8 正点原子 ESP32S3 基础工程架构

上图中的 `sdkconfig.old` 是之前的基础工程所使用的旧版系统配置文件，用于记录之前的配置信息。而当前的 `sdkconfig` 则是系统最新生成的系统配置文件，包含了最新的配置设置。此外，`partitions-16MiB.csv` 是系统配置保存后自动生成的分区表文件，该文件可供用户查看，以便了解当前的分区配置情况。这些文件共同构成了 ESP32 开发环境的配置体系。

接下来，作者在 `app_main` 函数中编写了代码，以获取 DNESP32S3M 最小系统板上 ATK-MWS3S 模组的内部资源信息。这些信息包括时钟频率、flash 大小以及 PSRAM 大小等。具体的代码实现如下所示：

```

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "nvs_flash.h"
#include "esp_system.h"
#include "esp_chip_info.h"
#include "esp_psram.h"
#include "esp_flash.h"

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;
    uint32_t flash_size;
    esp_chip_info_t chip_info; /* 定义芯片信息结构体变量 */
    ret = nvs_flash_init(); /* 初始化 NVS */
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    esp_flash_get_size(NULL, &flash_size); /* 获取 FLASH 大小 */
    esp_chip_info(&chip_info);
    printf("内核: cup 数量%d\n", chip_info.cores); /* 获取 CPU 内核数并显示 */
    /* 获取 FLASH 大小并显示 */
    printf("FLASH size:%ld MB flash\n", flash_size / (1024 * 1024));
    /* 获取 PARAM 大小并显示 */
    printf("PSRAM size: %d bytes\n", esp_psram_get_size());
    while(1)
    {
        printf("Hello-ESP32\r\n");
        vTaskDelay(1000);
    }
}

```

上述代码是获取 DNESP32S3M 最小系统板上 ATK-MWS3S 模组的内部资源信息，并打印到

监控器上。首先我们编译基础工程，然后下载至 DNESP32S3M 最小系统板中，最后打开监控器查看串口打印内容，如下图所示：

```
内核：cup数量 2
FLASH size:16 MB flash
PSRAM size: 8388608 bytes
Hello-ESP32
Hello-ESP32
Hello-ESP32
```

图 6.5.9 打印内部资源信息

在以后的例程中，我们是以这个基础工程来延申扩展，所以本章节的内容非常重要，望读者好好理解。

## 第七章 分区表

分区表，主要是用来对 ESP32 外挂的 SPI FLASH 进行区域划分的一个表格，通过一个表格，可以根据多个不同的功能，将 SPI Flash 划分为不同的区域，方便开发者存储数据。本章，作者带大家来了解一下 ESP32 分区表的妙用。

本章将分为以下几个小节：

7.1 分区表概述

7.2 分区表 API 函数

### 7.1 分区表概述

#### 一、分区表简介

ESP32 分区表在 ESP32 的 flash 存储管理中扮演着至关重要的角色。其主要意义体现在以下几个方面：

1，存储空间优化：通过将 flash 划分为不同的区域，每个区域根据其功能进行定义，可以更有效地利用存储空间。开发者可以根据实际需求为不同的应用或数据分配适当的存储空间，确保资源得到最大化利用。

2，数据管理和维护：每个分区都有其特定的作用，比如存储应用程序、文件系统数据、校准数据等。这种分区的设置使得数据的管理和维护变得更加方便。当需要对特定数据进行更新或修复时，只需关注相关的分区，而无需对整个 flash 进行操作。

3，应用程序更新和回滚：ESP32 支持在线升级（OTA）功能，这意味着应用程序可以在运行时进行更新。通过分区表，可以设置一个或多个用于 OTA 的分区，使得新版本的应用程序可以安全地下载并存储在 flash 中，然后在需要时切换到新版本。同时，如果新版本存在问题，还可以轻松回滚到旧版本，确保系统的稳定性。

4，灵活性：分区表可以根据用户的需求进行自定义。这意味着开发者可以根据其项目的特定需求来定义分区的数量、大小和功能。这种灵活性使得 ESP32 能够适应各种不同的应用场景。

在 ESP32 中，分区表固定长度为 0xc000 字节，也就是说一个分区表只能创建 95 条子分区（条目），并且 MD5 校验位和附加在分区表之后，用于在运行时验证分区表的完整性。

#### 二、分区表格式

ESP32 的分区表主要有两种格式，一种是.csv 格式，它方便开发人员进行更改和设置各个子分区的偏移与空间大小；另一种则是用于烧录设备的.bin 文件格式。在系统编译时，系统会将.csv 文件转化为.bin 文件格式的分区表。下面展示了基础例程的分区表文件的内容。

```
# Name,.....Type,.....SubType,.....Offset,.....Size,.....Flags
· nvs,.....data,.....nvs,.....0x9000,.....0x6000,....,
· phy_init,.....data,.....phy,.....0xf000,.....0x1000,....,
· factory,.....app,.....factory,.....0x10000,.....0x1F0000,..,
· vfs,.....data,.....fat,.....0x200000,.....0xA00000,..,
· storage,.....data,.....spiffs,.....0xc00000,.....0x400000,..,
```

图 7.1.1 基础例程的分区表文件

上图中展示了 ESP32 的多个子分区及其功能。其中，nvs 子分区是专为开发者设计的非易失性存储（NVS）设备区域；phy\_init 子分区用于存放 PHY 初始化数据，确保每个设备都能单独配置其 PHY；factory 子分区则专门用于存储应用程序区域；vfs 子分区作为虚拟文件系统的存储区域；而 storage 子分区则是自定义的 SPIFFS 文件系统区域。这些子分区共同构成了 ESP32 的分区结构，满足了不同功能的需求。

#### 三、分区表条目结构

从上图中可以看到，每一个子分区都由以下几个部分组成：

1，name：子分区名称。该字段对 ESP32-S3 并不是特别重要。

- 2, Type: 子分区的存储类型。设置子分区的存储格式, app (0x00) 和 data (0x01)。
- 3, SubType: 进一步描述或分类分区表的条目。如果这个子分区 Type 为 app, 则 SubType 只能设置 factory、ota\_0、ota\_15 和 test; 如果这个子分区 Type 为 data, 则 SubType 只能设置 ota、phy、nvs 和 nvs\_keys。
- 4, Offset: 偏移地址。编译地址必须是 4KB 的倍数
- 5, Size: 大小。子分区的大小
- 6, Flags: 标志位。一般不设置该字段。

#### 四、分区表的类型

分区表的类型具有四种, 如下表所示:

分区表类型	描述
Single factory app no OTA	小型的应用程序, 但没有 OTA 升级区域
Single factory app (large) no OTA	大型的应用程序, 但没有 OTA 升级区域
Factory app two OTA definitions	大型的应用程序, 且具备两个 OTA 升级区域
Custom partition table CSV	自定义分区表

表 7.1.1 分区表的类型

根据上述内容, 如果不打算使用 OTA 升级功能, 可以根据项目的大小选择第一种或第二种分区类型。而想要使用 OTA 升级功能时, 应选择第三种分区类型。作者推荐采用最后一种分区类型, 因为它允许自定义分区表, 能更精确地满足项目工程的需求, 从而实现完美的契合。

从上述内容可以了解到, 分区表是对 flash 闪存进行区域划分, 以便根据不同的功能需求将 flash 划分为不同的区域。这些区域可以包括应用、数据等不同类型, 例如应用可以进一步细分为 Factory 程序、OTA 程序等, 数据则可以包括校准数据、文件系统数据、参数存储数据等。

每个分区都有其特定的作用, 开发者可以根据自己的需求进行配置以及修改。这样的设计使得开发者能够更方便地管理和使用 flash 空间, 同时也有助于提高系统的稳定性和安全性。

## 7.2 分区表 API 函数

esp\_partition 组件是 ESP-IDF 中用于管理 ESP32 及其系列芯片上 flash 分区的一个关键组件。它提供了一组高层次的 API 函数, 允许开发者方便地访问和操作定义在分区表中的各个分区。这些高层次的 API 函数为开发者提供了简洁和易用的接口, 以进行诸如读取、写入、擦除分区内容等操作。这些函数可在 components/esp\_partition/include/esp\_partition.h 路径下找到这些分区表 API 函数。

### 1, esp\_partition\_find 函数

该函数查找子分区, 该函数原型如下所示:

```
const esp_partition_t *esp_partition_find_first(esp_partition_type_t type,
                                              esp_partition_subtype_t subtype,
                                              const char *label)
```

该函数的形参描述如下表所示:

参数	描述
type	子分区类型
	ESP_PARTITION_TYPE_APP 应用程序分区类型
	ESP_PARTITION_TYPE_DATA 数据分区类型
	ESP_PARTITION_TYPE_ANY 搜索随意类型分区
subtype	子类型 (请看 esp_partition_subtype_t 结构体)
label	子分区名称

表 7.2.1 esp\_partition\_find 函数函数描述

该函数返回值如下:

NULL: 未找到子分区。esp\_partition\_t 指针: 返回子分区。

### 2, esp\_partition\_read 函数

该函数用于读取子分区的某个地址的数据, 该函数原型如下所示:

```
esp_err_t esp_partition_read( const esp_partition_t* partition,
                             size_t src_offset,
```

`void* dst, size_t size)`

该函数的形参描述如下表所示：

参数	描述
partition	分区结构指针
src_offset	读取数据的地址
dst	存储数据的指针
size	读取数据大小

表 7.2.2 esp\_partition\_read 函数函数描述

该函数返回值如下：

ESP\_OK：读取成功。其他：失败。

### 3, esp\_partition\_write 函数

该函数用于写入子分区的某个地址的数据，该函数原型如下所示：

```
esp_err_t esp_partition_write( const esp_partition_t* partition,
                               size_t dst_offset,
                               void* src, size_t size)
```

该函数的形参描述如下表所示：

参数	描述
partition	分区结构指针
dst_offset	写入数据的地址
src	写入数据
size	写入数据大小

表 7.2.3 esp\_partition\_write 函数函数描述

该函数返回值如下：

ESP\_OK：读取成功。其他：失败。

### 4, esp\_partition\_range 函数

该函数用于擦除子分区的某个地址的数据，该函数原型如下所示：

```
esp_err_t esp_partition_erase_range(const esp_partition_t *partition,
                                    size_t offset, size_t size)
```

该函数的形参描述如下表所示：

参数	描述
partition	分区结构指针
offset	擦除数据的地址
size	擦除数据大小

表 7.2.4 esp\_partition\_range 函数函数描述

该函数返回值如下：

ESP\_OK：读取成功。其他：失败。

上述列举的函数是访问和操作分区表时较为常用的 API 函数。若需进一步了解或学习其他剩余的分区表 API 函数，可查阅 esp\_partition.h 头文件。

注意：上述函数的使用方法，可打开 12\_chinese\_display 实验下的 fonts.c 文件，我们使用这些函数把 GBK 字库更新至 storage 分区表，然后完成了汉字实验。

## 第八章 MENUCONFIG 菜单配置

ESP-IDF menuconfig 菜单配置在 ESP32 及其系列芯片的开发过程中起着至关重要的作用。它允许开发者通过友好的图形界面，对项目的各种配置选项进行细致入微的调整。这些配置选项涵盖了从硬件设置到软件功能的各个方面，确保了项目能够按照开发者的需求进行定制化开发。通过本章的学习，让读者更能熟悉和使用 menuconfig 菜单配置。

本章将分为如下几个小节：

- 8.1 menuconfig 概述
- 8.2 menuconfig 实现原理
- 8.3 配置项解析

### 8.1 menuconfig 概述

menuconfig 是 Linux 平台用于管理代码工程、模块及功能的实用工具。上至决定某一程序模块是否编译，下到某一行具体的代码是否需要编译以及某个项的值在本次编译时该是什么都可由 menuconfig 来定义。menuconfig 的使用方式通常是在编译系统之前在系统源代码根目录下执行 make menuconfig 命令从而打开一个图形化配置界面，再通过对各项的值按需配置从而达到影响系统编译结果的目的。乐鑫的 ESP-IDF 采用了这种便捷的方式来迅速配置项目中的多样化选项，它的功能类似于 STM32 的 CubeMX，虽然它不能直接构建工程，但却能够高效配置所需功能。下图是使用命令打开 ESP-IDF 项目的 menuconfig 菜单示意图。

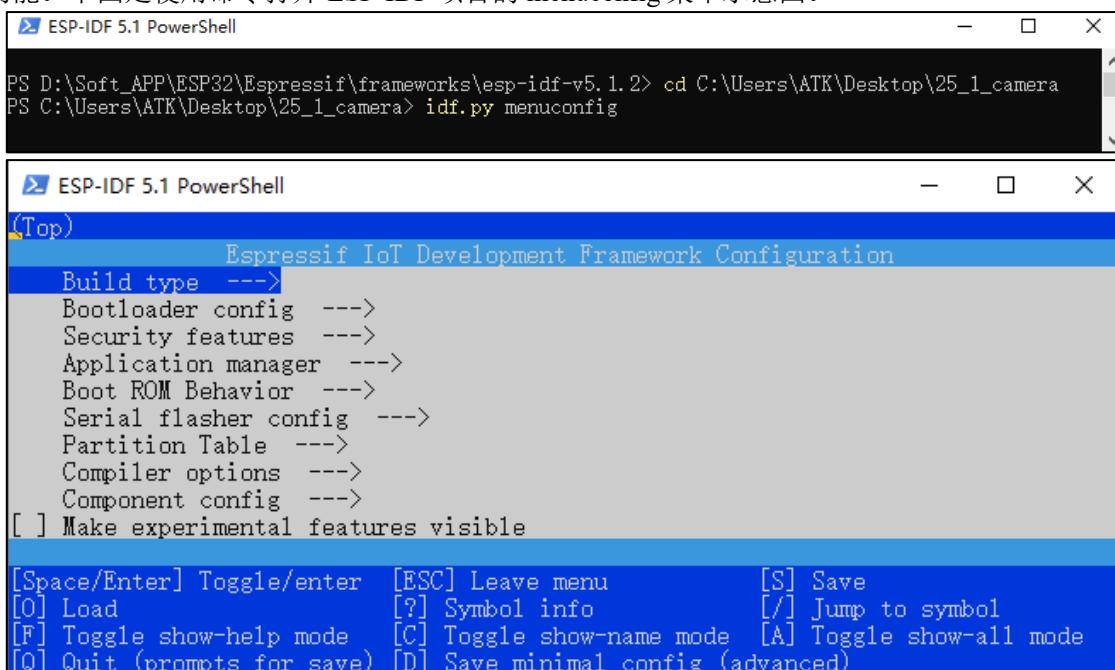


图 8.1.1 打开配置菜单

通过配置菜单上的各个选项，从而达到影响系统编译的结果的目的。

menuconfig 的配置优点体现在以下几个方面：

1，硬件资源配置：开发者可以通过 menuconfig 配置 ESP32 的硬件资源，如 SPIRAM 的启用与大小、Wi-Fi 和蓝牙的参数设置等。这些配置能够确保硬件资源得到合理的利用，提高系统的性能和稳定性。

2，分区表设置：menuconfig 提供了对 flash 分区表的详细配置功能。开发者可以自定义分区的大小、类型和属性，以满足不同项目对 flash 存储的需求。这对于固件升级、数据存储等功能至关重要。

3，组件功能配置：ESP-IDF 包含了众多组件，如网络、驱动、工具等。menuconfig 允许开

开发者根据项目的需求，启用或禁用这些组件，以及调整它们的配置参数。这有助于精简项目代码，提高开发效率。

4. 调试与日志设置：通过 menuconfig，开发者可以配置调试选项和日志级别，以便在开发过程中更好地跟踪和定位问题。这对于调试和优化项目性能非常有帮助。

menuconfig 菜单配置为开发者提供了一个强大而灵活的工具，使他们能够根据项目需求进行定制化开发，优化系统性能，提高开发效率。因此，在进行 ESP32 项目开发时，熟练掌握 menuconfig 的使用是非常必要的。在基础工程中，我们可以通过命令的形式打开 menuconfig 菜单界面，也可以通过 VS Code 软件的方式打开菜单界面（请参考 6.5 章节）。

## 8.2 menuconfig 实现原理

menuconfig 可以被视为一个“前端”工具，它提供了一个用户友好的界面来配置项目选项。而决定 menuconfig 拥有哪些配置项并管理这些配置项的“后端”则是 Kconfig 语言和相关的 Kconfig 文件。Kconfig 严格来讲是一种编程语言，它拥有自己的语法及结构。正是这些语法和结构组成了 menuconfig 在用户眼前不同的表现形式。下图是 ESP-IDF 项目工程菜单配置流程。

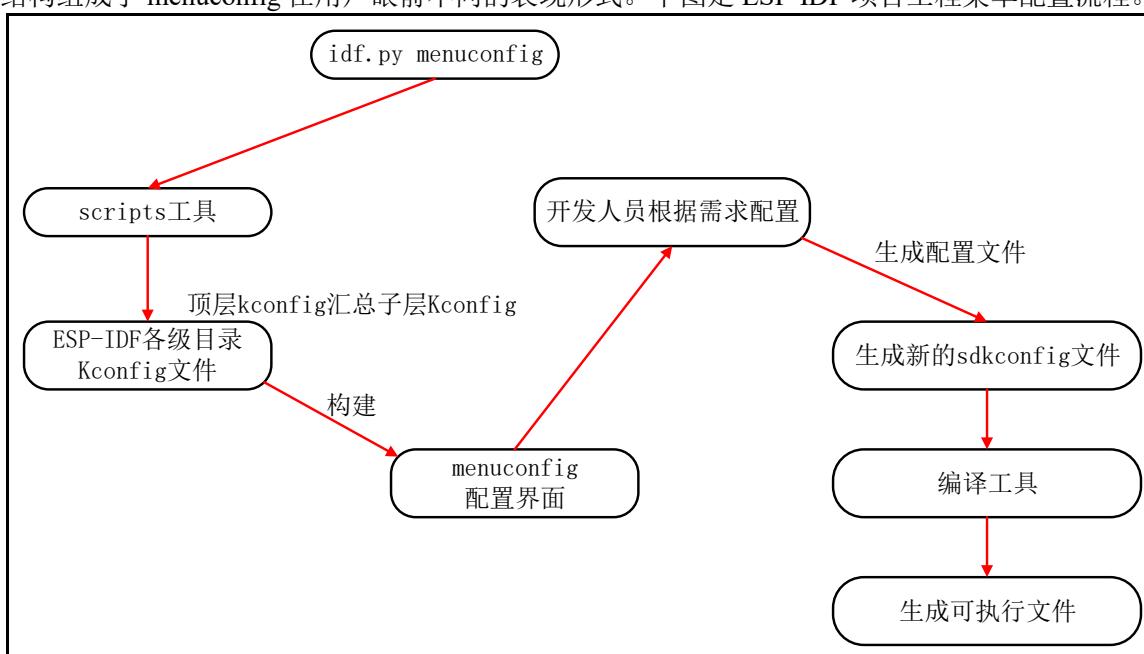


图 8.2.1 ESP-IDF 项目工程菜单配置流程

从上图中可见，当输入“`idf.py menuconfig`”命令时，顶层的 Kconfig 会调用多个子层 Kconfig 文件，共同构建出完整的配置选项集。这些选项集随后被转化为直观的可视化 menuconfig 配置界面，方便开发人员根据需求配置各个选项。完成配置并保存退出后，系统会自动生成 `sdkconfig` 文件。这个文件在项目中扮演着核心角色，它向构建系统提供编译和链接项目的指令，包括启用或禁用特定功能、选择硬件接口、设置网络参数等。这些配置信息对于确保生成正确的固件至关重要。在构建过程中，CMake 会读取 `sdkconfig` 文件，并根据其中的配置选项来精确配置构建过程。这样，最终生成的固件会精确匹配用户在 menuconfig 中所做的选择。

## 8.3 配置项解析

接下来，作者将为大家详细讲解 ESP-IDF 项目工程中常用的菜单配置项及其描述与作用。这些配置项允许开发者根据项目需求进行灵活定制。同时，附上的是 ESP-IDF 项目工程的菜单目录图，方便大家直观了解各项配置的位置和分类。通过合理配置这些选项，可以优化项目的开发流程，提高开发效率。

<b>Build type</b>	① 编译类型
Bootloader config	② Bootloader 配置
Security features	③ 安全特性
Application manager	④ 应用程序管理
Boot ROM Behavior	⑤ Boot ROM
Serial flasher config	⑥ Flash 配置
Partition Table	⑦ 分区表配置
> Compiler options	⑧ 编译器选项
> Component config	⑨ 组件配置

图 8.3.1 ESP-IDF 项目工程菜单目录

下面，作者将重点介绍上图中①、②、⑥和⑦分组的选项及其作用。对于其他分组的选项介绍与作用，建议读者查阅乐鑫官方 ESP-IDF 编程指南中的 menuconfig 菜单章节，以获取更详细的信息。请注意，以下截图均来自 VS Code 菜单界面。

### 1. 编译类型 (Build type)

用于配置项目工程编译时生成哪个应用程序类型，下图是编译类型分组的选项。

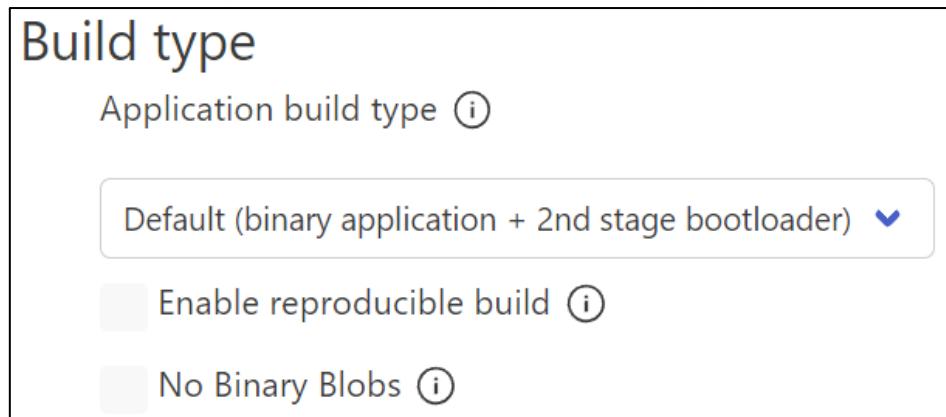


图 8.3.2 编译类型分组的选项

上图中，“Application build type”用于选择应用程序的构建方式。该选项为开发者提供了两种选择：其一为系统默认的“Default (binary application + 2nd stage bootloader)”，这表示应用程序将以与 ESP-IDF 引导加载程序兼容的二进制格式构建；另一种则是“Build app runs entirely in RAM (EXPERIMENTAL)”，它仅适用于非常小和有限的应用程序，通过仅链接应用程序的 elf 文件，使其能够通过 JTAG 直接加载到 RAM 中。但请注意，由于 IRAM 和 DRAM 的大小非常有限，因此采用这种方式构建复杂应用程序是不可行的。

上图中的“Enable reproducible build”选项表示可复制构建；上图中的“No Binary Blobs”选项表示禁用应用程序构建中的二进制库链接，但注意，此项开启后，WiFi 与蓝牙无法工作。

### 2. Bootloader 配置 (Bootloader config)

用于配置引导加载程序编译器优化等级、日记等，下图为 Bootloader 配置分组的选项。

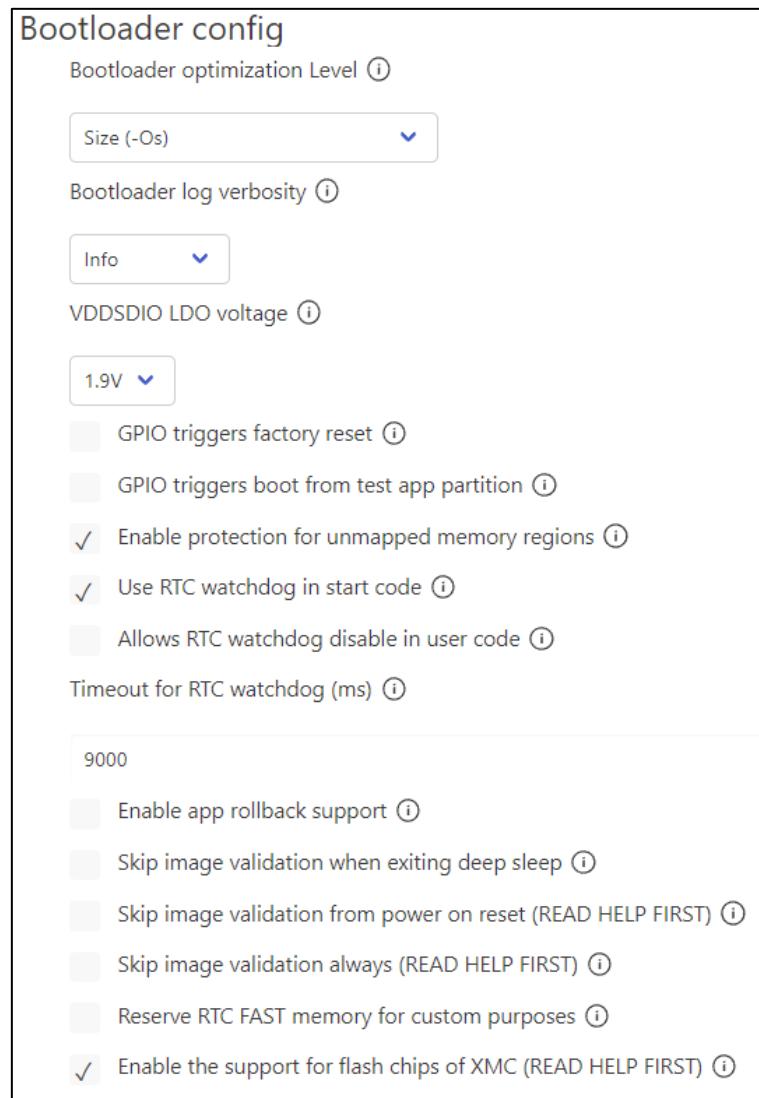


图 8.3.3 Bootloader 配置分组的选项

下面，我们来看一下上图各个选项的描述与作用。

### (1) Bootloader optimization Level 选项

这一选项为引导加载程序设置编译器优化级别（gcc-O 参数）。

- **Size(-Os)**: 配置引导加载程序编译器优化大小。
- **Debug(-Og)**: 配置引导加载程序编译器优化调试。
- **Optimize for performance (-O2)**: 配置引导加载程序编译器优化性能。
- **Debug without optimization (-O0)**: 无配置引导加载程序编译器优化。

### (2) Bootloader log verbosity 选项

日志输出等级设置，以下为可供配置的项目：

- **No output**: 日志输出等级为 NONE。
- **Error**: 日志输出等级为 ERROR。
- **Warning**: 日志输出等级为 WARN。
- **Info**: 日志输出等级为 INFO。
- **Debug**: 日志输出等级为 DEBUG。
- **Verbose**: 日志输出等级为 VERBOSE。

### (3) VDDSDIO LDO voltage 选项

配置 LDO 输出电压，这有助于防止闪存芯片在闪存编程操作过程中发生故障。

- **1.9V**: 配置引导加载程序 VDDSDIO BOOST 1.9V

**(4) GPIO triggers factory reset 选项**

允许用户将设备重置为出厂设置，操作包括清除一个或多个数据分区，并从“工厂”分区启动。若在设备启动时，GPIO 输入保持在预设的配置水平，则将自动触发出厂重置功能。默认情况下，该选项配置为“No”，即出厂重置功能处于关闭状态。

**(5) GPIO triggers boot from test app partition 选项**

允许用户直接从“TEST”分区运行测试应用程序。在设备启动时，如果检测到 GPIO 输入被拉低，系统会自动从“TEST”分区加载并启动测试程序，无需额外的操作或配置。

**(6) Enable protection for unmapped memory regions 选项**

为保护地址空间中未映射的内存区域免受意外访问，我们已启用该选项，确保 CPU 在执行涉及未映射区域的内存操作时触发异常。此选项已设置为“enable”，从而提供更强的内存安全保护。

**(7) Use RTC watchdog in start code 选项**

启用跟踪启动代码执行时间的功能，以确保在超过预定执行时间时，RTC\_WDT（实时时钟看门狗）能够重新启动系统，从而有效避免电源不稳定时启动代码可能发生的锁定问题。此跟踪过程将从引导加载程序代码启动，重新设置超时时长，并选择 slow\_clk 的时钟源，直至 app\_main 被调用结束。由于看门狗使用 SLOW\_CLK 作为时钟源，因此在调整 slow\_clk 频率后，我们必须相应地为新的频率重新设置看门狗的时间。值得注意的是，slow\_clk 的具体时钟源依赖于 RTC\_clk\_SRC 的设置，即是否选择 INTERNAL\_RC（内部时钟源）或 EXTERNAL\_CRYSTAL（外部晶体时钟源）。为了确保系统稳定性和可靠性，我们强烈推荐启用该功能，并将其配置为“enable”状态。

**(8) Allows RTC watchdog disable in user code 选项**

如果启用了该选项，ESP-IDF 应用程序需要在其代码中自行处理 RTC\_WDT 的重置、喂狗或禁用操作。而若未启用此选项（默认为禁用状态），ESP-IDF 会在调用 app\_main() 函数之前自动禁用 RTC\_WDT。如需手动重置 RTC\_WDT 的计数器，请使用函数 rtc\_wdt\_feed()；若需禁用 RTC\_WDT，则使用函数 rtc\_wdt\_disable()。为确保应用程序的稳定运行，我们推荐保持该选项为禁用状态，即选择“No (disable)”。

**(9) Timeout for RTC watchdog (ms) 选项**

设置 RTC 看门狗的超时时间，其范围为 0 至 120000 毫秒 (ms)，而默认的超时时间已设置为 9000 毫秒 (ms)。

**(10) Enable app rollback support 选项**

更新应用程序后，引导加载程序将启动新应用并设置“ESP\_OTA\_IMG\_PENDING\_VERIFY”状态，此状态会防止该应用被重复运行。用户代码首次启动新应用后，应调用相关函数以确认应用的可操作性。若应用运行正常，则将其标记为有效；否则，将其标记为无效，并回滚到前一个工作应用。随后将执行重启操作，并在软件更新前启动相应应用。请注意，新应用在首次启动时若发生电源中断或 WDT 触发，将触发回调函数。此回调仅在具有相同安全版本的应用之间有效。为确保系统稳定性，此选项默认设置为禁用，即“No(disable)”。

**(11) Skip image validation when exiting deep sleep 选项**

此选项禁用从深度睡眠唤醒时对图像的常规验证（包括校验和、SHA256 和签名），这是为了权衡深度睡眠唤醒的性能与图像完整性检查。请仅在明确知道操作后果的情况下启用此选项。同时，不建议将此选项与 deep\_sleep() 调用以及活动 OTA 分区的更改一同使用，因为这将会跳过对新 OTA 分区加载的首次验证。虽然启用了“允许不安全选项”后，安全引导功能仍可使用，但强烈建议禁用此选项，因为它可能允许绕过安全引导机制，从而带来安全风险。

**(12) Skip image validation from power on reset (READ HELP FIRST) 选项**

一些应用程序需要迅速开机启动。默认情况下，应用程序的完整二进制文件会从闪存读取并验证，这一过程占据了大量的引导时间。

**(13) Skip image validation always (READ HELP FIRST) 选项**

选择此选项可防止引导加载程序在引导应用程序映像之前对其进行验证。所选应用程序分区的任何闪存损坏都将使整个 SoC 无法启动。

**(14) Reserve RTC FAST memory for custom purposes 选项**

此选项允许客户将数据放入 RTC FAST 内存，当重启时此区域保持有效，电源丢失除外。

此内存位于固定地址，可用于引导加载程序和应用程序。（应用程序和引导程序必须使用编译相同的选项）。RTC FAST 内存只能通过 PRO\_CPU 进行访问。

#### (15) Enable the support for flash chips of XMC (READ HELP FIRST) 选项

执行 XMC 建议的启动流程。

### 3, Flash 配置 (Serial flasher config)

用来设置挂载的 Flash 通信模式、大小、速率等操作。下图是 Flash 配置选项。

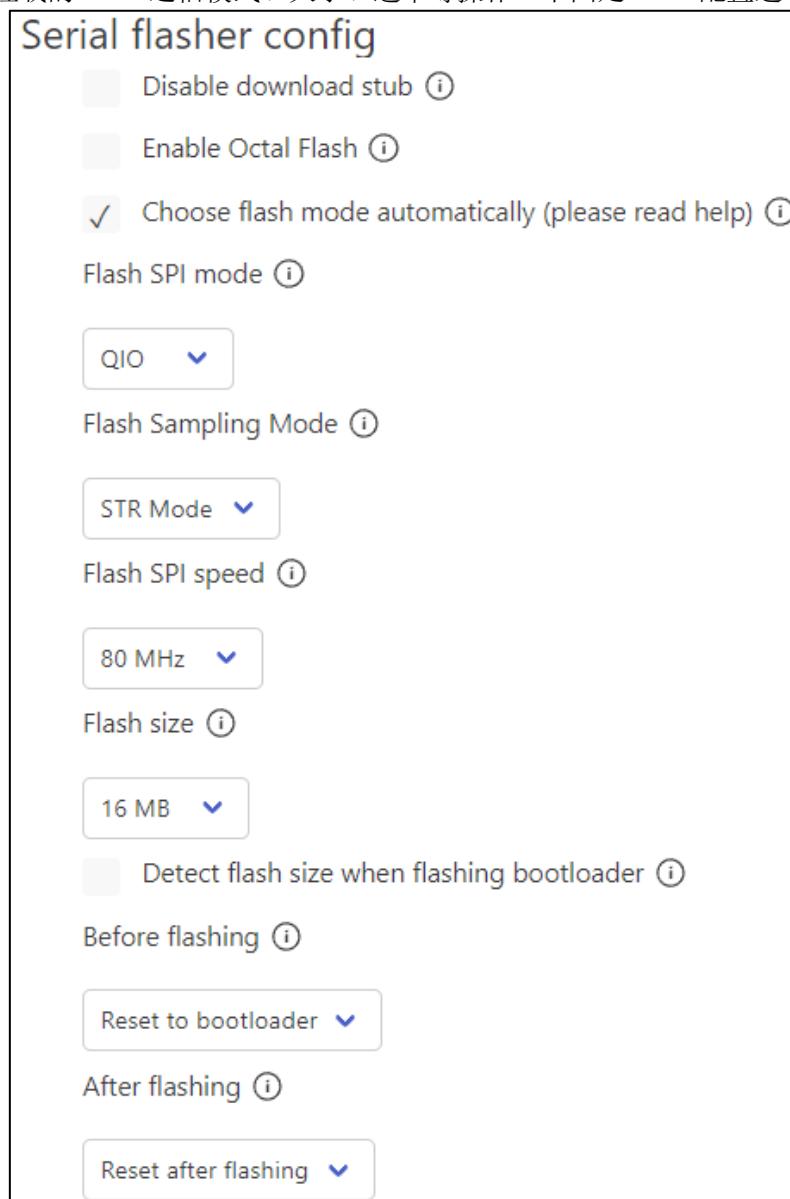


图 8.3.4 Flash 配置选项

下面，作者讲解一下上图中各个选项的描述与作用。

#### (1) Disable download stub 选项

禁用下载存根。通常是指在某些嵌入式系统或固件更新过程中，关闭或禁用一个用于下载或接收新固件代码的小程序或模块。存根（stub）是一个占位符或简化版的程序，用于在开发或测试阶段模拟某个功能或接口的行为。

#### (2) Enable Octal Flash 选项

启用八通道闪存（Octal Flash）功能。八通道闪存是一种高性能的存储器技术，它利用多个通道（在这里是八个通道）并行访问闪存芯片，从而提高了数据读写速度。注意：请根据模组挂载 Flash 是否使用 Octal SPI。

#### (3) Choose flash mode automatically (please read help) 选项

让系统或烧录工具自动选择适合的闪存（flash memory）模式。

#### (4) Flash SPI mode 选项

设置当前挂载的 Flash SPI 模式。下表是系统提供的配置项描述。

可选项	模式名称	引脚	速度
QIO	Quad I/O	地址和数据 4pins	最快
QOUT	Quad Output	数据 4pins	约比 qio 模式下慢 15%
DIO	Dual I/O	地址和数据 2pins	约比 qio 模式下慢 45%
DOUT	Dual Output	数据 2pins	约比 qio 模式下慢 50%

表 8.3.1 四种 SPI 模式的对比

#### (5) Flash Sampling Mode 选项

闪存（Flash Memory）操作中的一种特定模式，用于读取或采样闪存中的数据。

#### (6) Flash SPI speed 选项

配置 SPI Flash 速率，系统提供开发者配置项分别为 120、80、40 和 20MHz。

#### (7) Flash size 选项

配置 Flash 大小，系统提供开发者配置项分别为 1、2、4、8、16、32、64 和 128MB。

#### (8) Detect flash size when flashing bootloader 选项

在烧录引导加载程序（bootloader）时检测闪存（flash）大小，通常是为了确保引导加载程序能够正确适配目标设备的闪存容量，并避免潜在的溢出或空间不足的问题。

#### (9) Before flashing 选项

固件烧录后的设备复位行为。可用配置：

- Reset to bootloader: 复位 bootloader
- No reset: 无需复位

#### (10) After flashing 选项

官方并没有描述此选项。

- Reset after flashing: 复位 Flash
- Stay in bootloader: 停留在引导程序中

### 4. 分区表配置（Partition Table）

分区表的介绍，请读者查看第七章节的内容。下图是分区表菜单示意图。

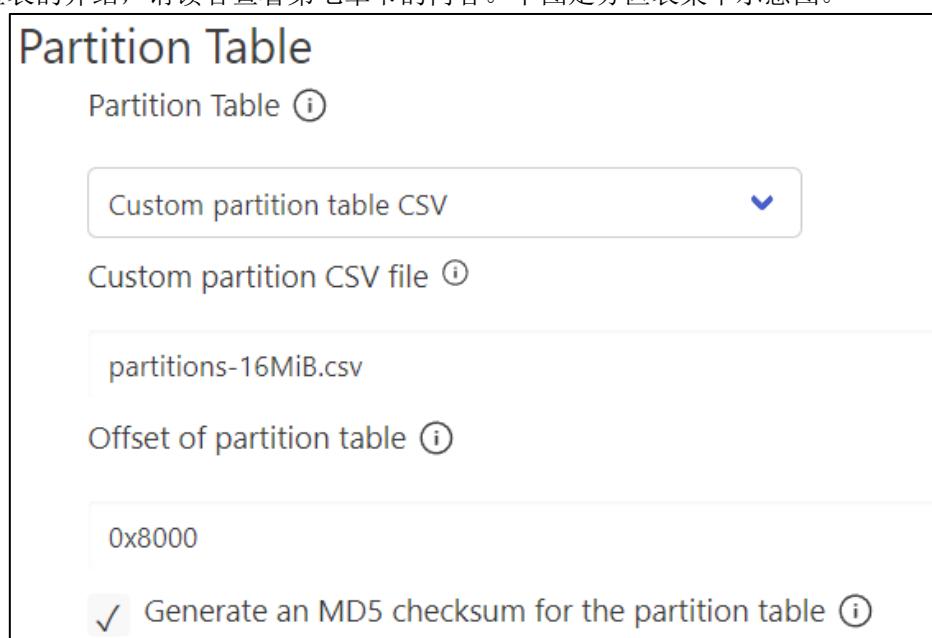


图 8.3.5 分区表菜单选项

#### (1) Partition Table 选项

设置分区表的类型，下表是分区表的类型描述：

分区表类型	描述
Single factory app no OTA	小型的应用程序，但没有 OTA 升级
Single factory app (large) no OTA	大型的应用程序，但没有 OTA 升级
Factory app two OTA definitions	大型的应用程序，且具备两个 OTA 升级
Custom partition table CSV	自定义分区表

表 8.3.2 分区表的类型

**(2) Custom partition CSV file 选项**

设置分区表名称，如 partitions-16MiB.csv。

**(3) Offset of partition table 选项**

程序烧录时，分区表下载至哪个偏移地址。默认为 0x8000。

**(4) Generate an MD5 checksum for the partition table 选项**

在分区表后是否添加 MD5 校验位和。

**5. PSRAM 配置 (Component config/ESP PSRAM)**

下图为 PSRAM 配置界面。

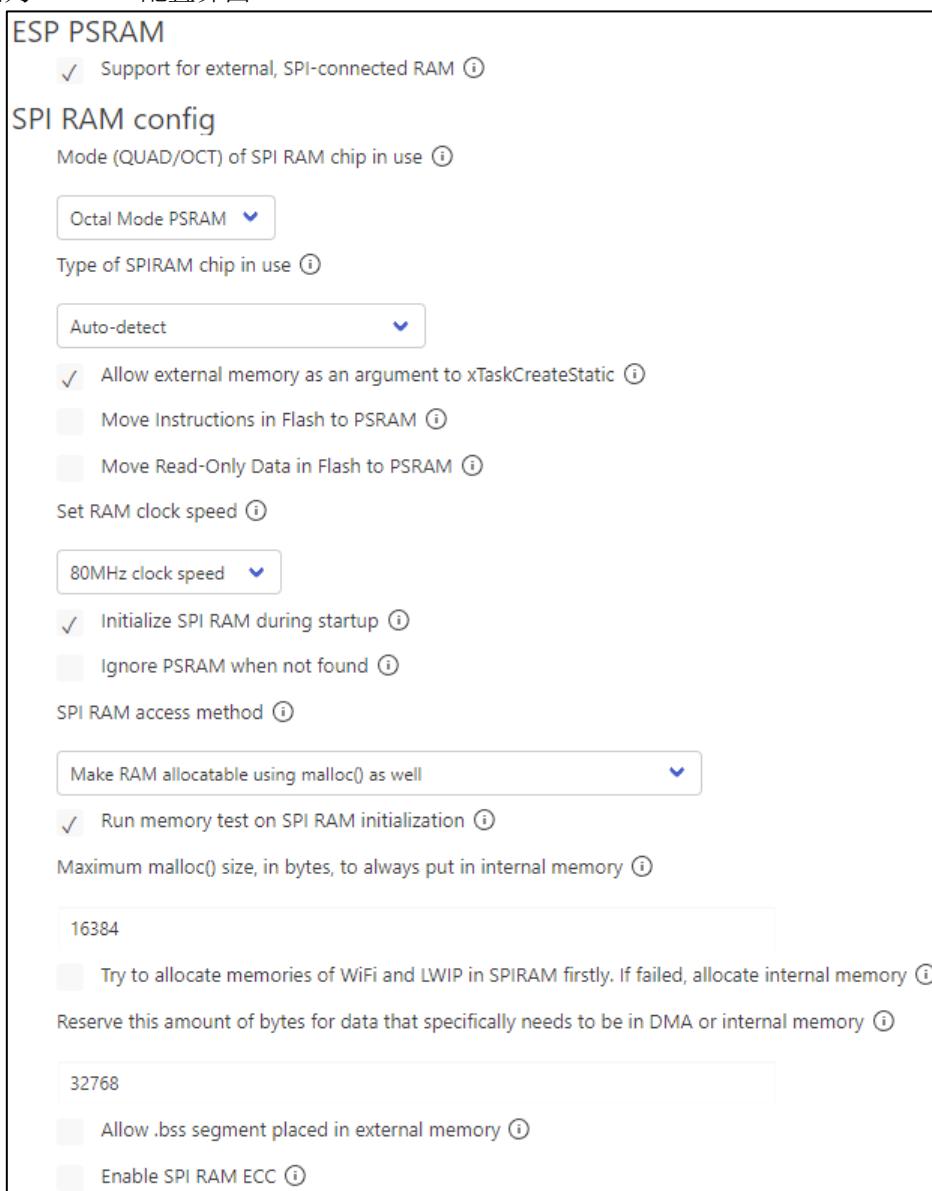


图 8.3.6 PSRAM 配置界面

**(1) Support for external, SPI-connected RAM 选项**

是否支持扩展 PSRAM，如挂载了 PSRAM，则需要启动该选项。

**(2) Mode (QUAD/OCT) of SPI RAM chip in use 选项**

设置 PSRAM 的 SPI 模式，需开发者查看当前的模组数据手册，才能确定 PSRAM 的 SPI 模式。该选项的可用配置：

- Quad Mode PSRAM: 四线
- Octal Mode PSRAM: 八线

**(3) Type of SPIRAM chip in use 选项**

设置 PSRAM 芯片类型。该选项的可用配置：

- Auto-detect : 自动检测。
- ESP-PSRAM64,LY68L6400 or APS6408: 固定的 PSRAM 芯片。

**(4) Allow external memory as an argument to xTaskCreateStatic 选项**

是否支持 PSRAM 内存用作于 FreeRTOS 的静态创建函数使用。

**(5) Move Instructions in Flash to PSRAM 选项**

是否支持将 Flash 中的指令移动到 PSRAM。

**(6) Move Read-Only Data in Flash to PSRAM 选项**

是否支持将 Flash 中的只读数据移动到 PSRAM。

**(7) Set RAM clock speed 选项**

设置 SPI 时钟速率，可选配置项：

- 80MHz clock speed: 80MHz
- 40Mhz clock speed: 40MHz

**(8) Initialize SPI RAM during startup 选项**

是否启动时初始化 SPI RAM。

**(9) Ignore PSRAM when not found 选项**

是否启动当没有找到 PSRAM 时，忽略 PSRAM。

**(10) SPI RAM access method 选项**

SPI RAM 存储方法，可选配置项：

- Integrate RAM into memory map: 将 PSRAM 集成到内存映射当中。
- Make RAM allocatable using heap\_caps\_malloc(..., MALLOC\_CAP\_SPIRAM): 支持使用 heap\_caps\_malloc 等函数分配内存。
- Make RAM allocatable using malloc as well: 支持使用 malloc 等函数分配内存。

**(11) Run memory test on SPI RAM initialization 选项**

是否启动在 SPI RAM 初始化上运行内存测试。

**(12) Maximum malloc() size, in bytes, to always put in internal memory 选项**

设置最大分配内存。

**(13) Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, allocate internal memory 选项**

尝试先在 SPIRAM 中分配 WiFi 和 LWIP 的内存。如果失败，再分配内部内存。

**(14) Reserve this amount of bytes for data that specifically needs to be in DMA or internal memory 选项**

为那些特别需要在 DMA 或内部内存中的数据保留这些字节。

**(15) Allow .bss segment placed in external memory 选项**

允许.bss 端放置在外部存储器中。

**(16) Enable SPI RAM ECC 选项**

使能 SPI RAM ECC。

其他配置项描述与作用，请读者参考乐鑫官方的 ESP-IDF 编程指南 MENUCONFIG 章节。

## 第九章 IDF 组件注册表

IDF 组件注册表（IDF Component Registry）是一个为 ESP-IDF（Espressif IoT Development Framework）开发框架提供的官方组件搜索和添加平台。开发者可以通过访问 IDF 组件注册表，搜索并找到所需的组件，然后按照提供的指南将组件添加到自己的 ESP-IDF 项目中。例如，如果开发者需要 LVGL 组件和 USB 组件，开发者可以在 IDF 组件注册表中搜索这两个组件，然后找到每个组件的添加代码。在终端窗口中，开发者可以输入这些添加命令来将组件添加到他们的项目中。一旦添加成功，这些组件就可以在项目中使用，从而简化了组件的集成过程。

本章将分为如下几个小节：

9.1 IDF 组件注册表简介

9.2 项目工程如何添加组件

### 9.1 IDF 组件注册表简介

IDF 组件注册表是乐鑫官方专为 ESP-IDF 开发框架打造的一个组件搜索与添加平台。尽管 ESP-IDF 物联网开发框架的 components 文件夹内已包含诸如 lwIP、MQTT 等丰富的驱动程序和第三方组件，但这些组件主要服务于物联网领域，具有特定的专一性。部分第三方库并未直接集成于 ESP-IDF 中，为了提升开发者的工作效率，乐鑫官方推出了 IDF 组件注册表，其中存储了已适配的第三方组件供开发者使用。开发者只需联网，即可轻松下载并移植这些组件到自己的项目工程中。以下是 IDF 组件注册表的特点。

(1) 简化组件搜索和集成：IDF 组件注册表为开发者提供了一个集中化的平台，用于搜索和查找 ESP-IDF 项目中所需的组件。通过注册表，开发者可以轻松地浏览和筛选组件，根据项目的需求选择适合的组件，并快速集成到我们的项目中。

(2) 提高开发效率：有了 IDF 组件注册表，开发者无需花费大量时间在互联网上搜索和评估各种组件。我们可以直接在注册表中找到经过验证和官方推荐的组件，这大大减少了寻找和测试组件所需的时间，从而提高了开发效率。

(3) 确保组件兼容性和质量：IDF 组件注册表中的组件都经过严格的测试和验证，以确保它们与 ESP-IDF 框架的兼容性和稳定性。这有助于减少项目中可能出现的兼容性问题，提高软件的质量。

(4) 促进组件共享和复用：通过 IDF 组件注册表，开发者可以轻松地共享和复用他们创建的组件。这有助于推动 ESP-IDF 社区的发展，鼓励更多的开发者参与到组件的开发和贡献中，从而形成一个良性的生态循环。

(5) 提供官方支持和文档：IDF 组件注册表不仅提供了组件的搜索和添加功能，还为每个组件提供了详细的文档和官方支持。这有助于开发者更好地理解和使用组件，解决在使用过程中可能遇到的问题。

乐鑫官方的 IDF 组件注册表打开地址为：<https://components.espressif.com/>。打开以后如下图所示：

## Find the most exciting ESP-IDF components

Kickstart your next IoT Project with the open-source components, you can also easily integrate the components into your existing IDF projects.



图 9.1.1 IDF 组件注册表

从上图可清晰看出，我们可以通过②或③选项对组件进行筛选，以排除那些不支持 ESP-IDF 特定版本或特定芯片的第三方库。同时，我们也可以在①处直接搜索需要添加的组件。例如，作者就通过搜索功能找到了 esp\_jpeg 这一 JPEG 解码库（**作者以 esp\_jpeg 解码库组件为例，其他的组件是类似的**），如下图所示。

README

## JPEG Decoder: TJpgDec - Tiny JPEG Decompressor

latest version 1.0.5~2

TJpgDec is a generic JPEG image decompressor that is highly optimized for small embedded systems. It works with very low memory consumption.

Some microcontrollers have TJpg decoder in ROM, it is used, if there is.  
[^1] Using ROM code can be disabled in menuconfig.

[^1]: **NOTE:** When the ROM decoder is used, the configuration can't be changed. The configuration is fixed.

**Links**

- [Homepage](#)

**Supports all targets**

**License:** Apache-2.0

To add this component to your project, run:

```
idf.py add-dependency "espressif/esp_jpeg^1.0.5~2"
```

or [download archive](#)

**Dependencies**

- ESP-IDF >=4.4

**Examples:**

[get\\_started](#) [more details](#)

图 9.1.2 esp\_jpeg 解码库

在上图的页面中，我们可以轻松找到关于 esp\_jpeg 的使用方法、芯片支持情况、使用示例以及添加组件的详细方法等内容介绍。这些内容具体如下：

### 1. 芯片支持

esp\_jpeg 解码库可支持 ESP32、ESP32-S3、ESP32-C3 和 ESP32-C6 这些芯片型号，如下图所示：

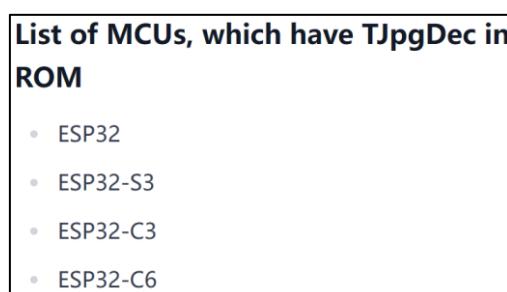


图 9.1.3 esp\_jpeg 解码库可支持芯片型号

## 2. 使用要求

开发时，需要查看 `esp_jpeg` 组件的使用要求，才能添加到项目当中。下图是官方提供的 `esp_jpeg` 解码库组件的使用要求。

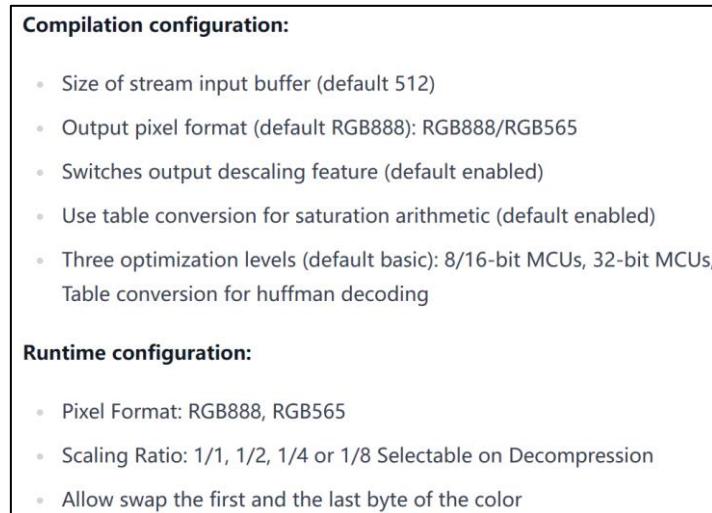


图 9.1.4 `esp_jpeg` 解码库组件使用要求

这个解码库组件对 MCU、存储内存以及 LCD 都有明确的使用要求。如果设备未能满足这些要求，那么将无法顺利使用该解码库组件。因此，在选择和使用此组件时，务必确保设备满足相应的硬件条件。

## 3. 使用示例

通过命令的形式就可以下载 `esp_jpeg` 解码库组件的示例工程，如下图所示：

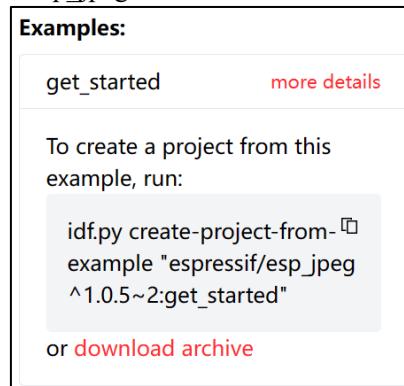


图 9.1.5 下载使用实例命令

在桌面上创建名为“test”的文件夹后，我们打开“ESP-IDF 5.1 PowerShell”窗口，并导航至该文件夹。在“test”文件夹内，我们执行命令“`idf.py create-project-from-example 'espressif/esp_jpeg^1.0.5~2:get_started'`”来下载 `esp_jpeg` 的示例工程。如下图所示：

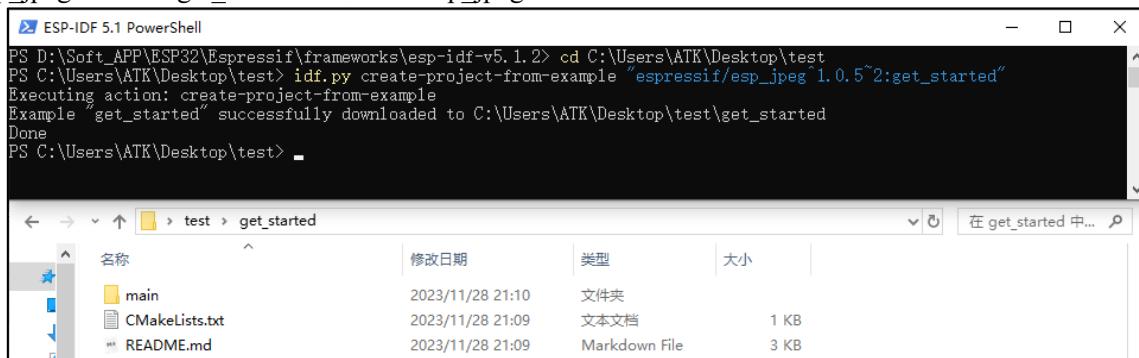


图 9.1.6 下载 `esp_jpeg` 组件示例

这样我们就可以参考这个示例来编写自己的 jpeg 解码实验了。

#### 4. ESP-IDF 版本要求

esp\_jpeg 解码库组件只能在 v4.4 以上版本运行，如下图所示：

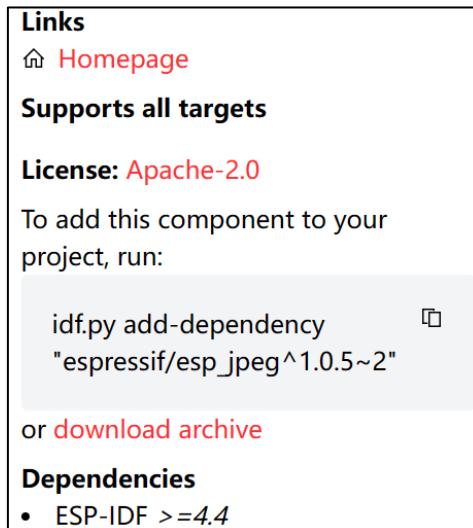


图 9.1.5 ESP-IDF 版本要求

从上图可以看到，本教程安装的 ESP-IDF5.1.2 可支持这一个组件库。

## 9.2 项目工程如何添加组件

### 9.2.1 命令式添加组件

在项目工程中添加 IDF 组件注册表的组件是非常简单的，我们只需要输入“`idf.py add-dependency "espressif/esp_jpeg^1.0.5~2"`”命令就可以把组件下载到项目工程当中了。注意：执行这条命令时，系统会在工程的 main 文件夹下生成 `idf_component.yml` 文件，工程编译时，系统从 IDF 组件注册表 Git 仓库获取组件。在 ESP-IDF 中，组件是代码和资源的集合，这些代码和资源可以单独编译并与其他组件一起链接，以构建最终的应用程序。每个组件都可以有自己的 `idf_component.yml` 文件，该文件提供了关于该组件的元数据信息，以及构建和集成该组件所需的指令。下面，作者以基础工程 `00_basic` 为例，在此工程下添加 `esp_jpeg` 解码库组件。

(1) 使用“ESP-IDF 5.1 PowerShell”进入 `00_basic` 工程目录下，如下图所示：

```
PS D:\Soft_APP\ESP32\Espresif\frameworks\esp-idf-v5.1.2> cd C:\Users\ATK\Desktop\00_basic
```

图 9.2.1.1 命令进入 `00_basic` 工程目录下

(2) 输入“`idf.py add-dependency "espressif/esp_jpeg^1.0.5~2"`”命令，如下图所示：

```
PS D:\Soft_APP\ESP32\Espresif\frameworks\esp-idf-v5.1.2> cd C:\Users\ATK\Desktop\00_basic
PS C:\Users\ATK\Desktop\00_basic> idf.py add-dependency "espressif/esp_jpeg^1.0.5~2"
Executing action: add-dependency
Created "C:\Users\ATK\Desktop\00_basic\main\idf_component.yml"
Successfully added dependency "espressif/esp_jpeg 1.0.5~2" to component "main"
Done
PS C:\Users\ATK\Desktop\00_basic>
```

图 9.2.1.2 下载 `esp_jpeg` 解码库 `idf_component.yml` 文件

此时，我们会发现基础工程 `00_basic` 的 `main` 文件夹下多了一个 `idf_component.yml`，如下图所示：

00_basic > main		
名称	类型	大小
main.c	C 文件	2 KB
idf_component.yml	YML 文件	1 KB
CMakeLists.txt	文本文档	1 KB

图 9.2.1.3 生成 idf\_component.yml

(3) 使用 VS Code 打开 00\_basic 基础工程，然后线清除工程再编译此工程，编译成功之后，如下图所示：

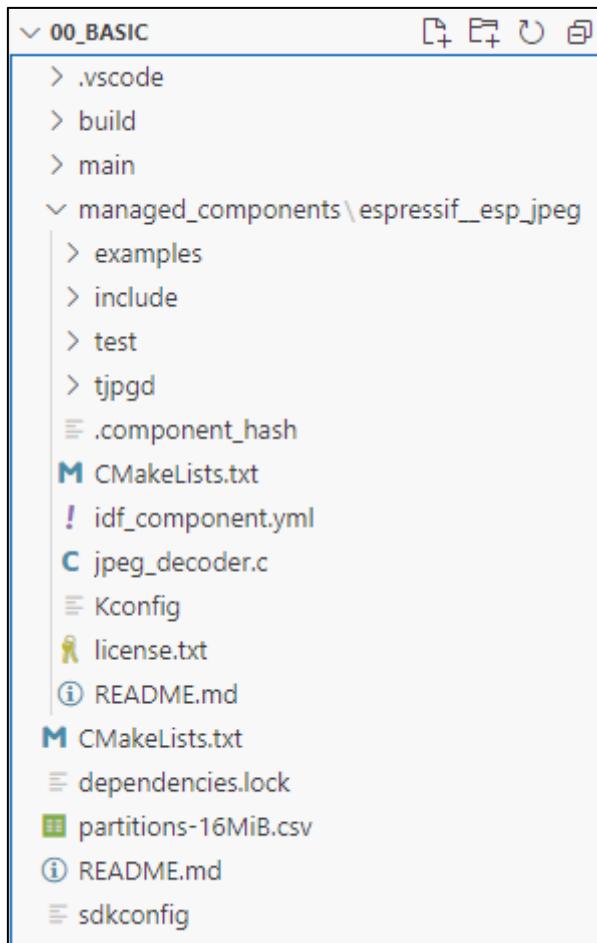


图 9.3.1.3 自动下载 esp\_jpeg 解码库

从上图可见，在编译 00\_basic 基础工程时，系统会自动从 IDF 组件注册表中下载 esp\_jpeg 解码库组件，并将其集成到我们的工程项目中。这就是 IDF 组件注册表的添加与使用机制。但请注意，若要通过 IDF 组件注册表添加组件，必须确保设备处于联网状态，否则下载操作将失败。因此，在 USB 实验中，作者选择直接将 USB 库放置到 components 文件夹中，一来为了保持工程结构清晰，二来避免在清除编译时误删 managed\_components 文件夹。这样的处理方式确保了项目的结构清晰和编译的顺畅进行。

最后，我们可在.c 文件下引用 esp\_jpeg 解码库组件了。其他组件也是一样的操作流程。望读者可好好利用 IDF 组件注册表这一牛 X 的功能。

## 9.2.2 VS Code 工程添加组件

在 VS Code 项目工程中添加 IDF 组件注册表中的组件十分便捷。您只需按下“Ctrl+Shift+P”快捷键快速进入命令面板，或者通过菜单栏的“查看”选项，选择“命令面板”来打开它。随后，在命令面板中输入“ESP-IDF: Show Component Registry”即可展示出组件注册表，具体操

作如图所示。



图 9.2.2.1 ESP-IDF 注册表

在搜索框中，我们可以方便地查找开发者所需的组件。请注意，下载 ESP-IDF 注册表中的组件需要在联网环境下进行，因此请确保您的电脑已连接到网络，以便顺利下载所需的组件。下面作者以 qrcode 为例来讲解。

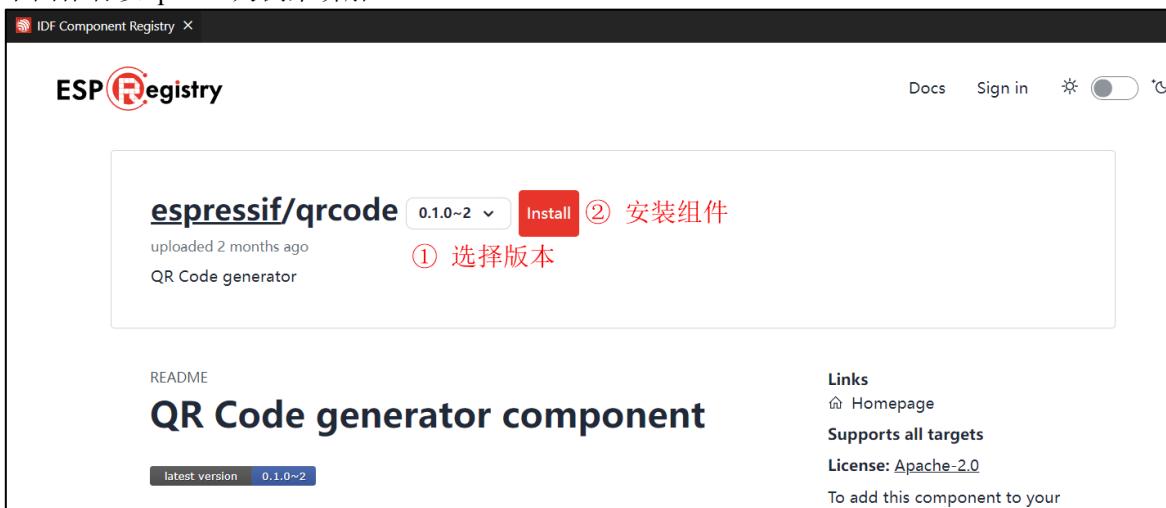


图 9.2.2.2 选择版本并安装组件

在上图中，首先选择适合的组件版本，随后点击“install”按钮以安装 qrcode 组件。请务必注意，选择与 ESP32S3 芯片适配的组件至关重要，因为并非所有组件都兼容这款芯片。为了筛选出与 ESP32S3 兼容的组件，开发者可以在图 9.2.2.1 中选择“By target”选项，并将其设置为 ESP32-S3。这样，您就能清晰地查看到这款芯片所支持的组件列表。

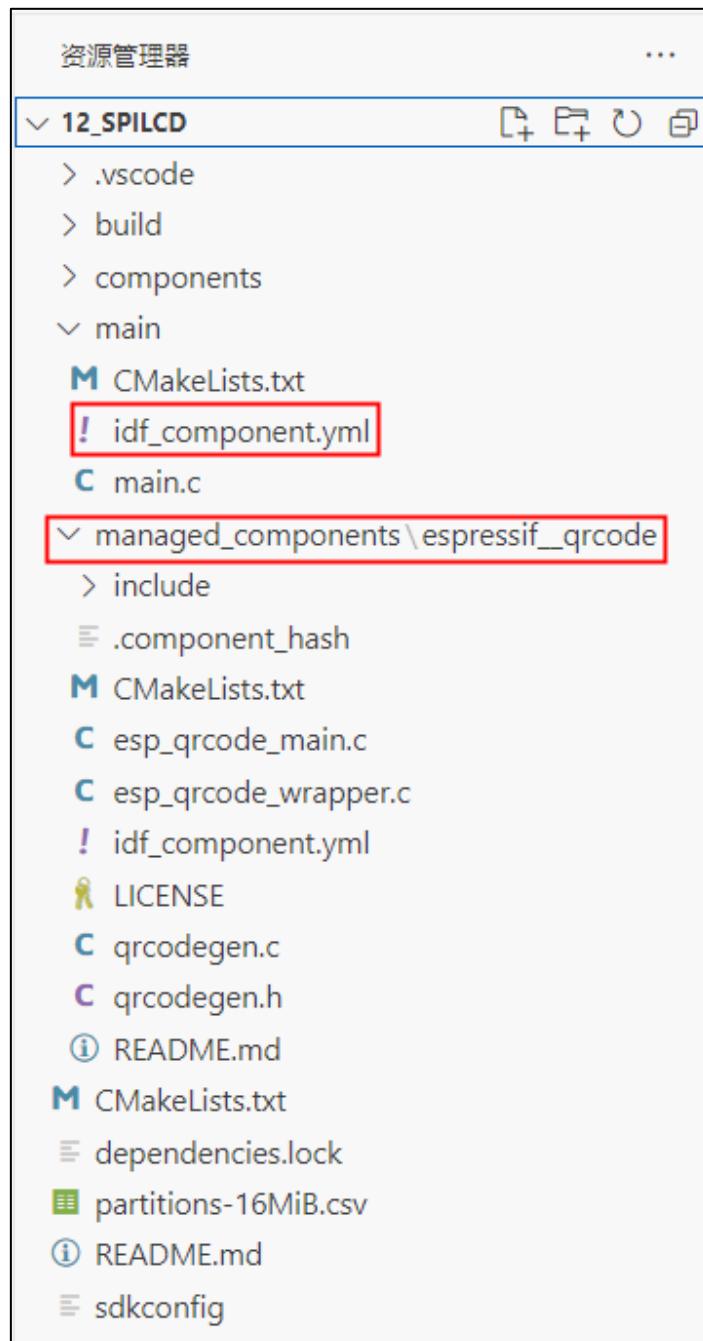


图 9.2.2.3 下载 qrcode 组件

从上图可见，我们只需点击图 9.2.2.2 中的“install”按钮，即可为项目工程安装 qrcode 组件。随后，espressif-qrcode 组件将被下载至项目工程目录下，并在 main 文件夹中自动生成 idf\_component.yml 文件。完成这些步骤后，我们便可以在 c 文件中直接调用 qrcode 的相关函数了。

# 第二篇 入门篇

入门篇主要讲解了 ESP32-S3 IDF 开发中 IDF 库的使用，包括 GPIO、RTC、PWM 和传感器数据读取等。通过学习这些内容，我们可以了解到如何使用 IDF 库来控制 ESP32-S3 的硬件接口和传感器，并实现各种应用。

本篇分为以下几个章节：

- 1, LED 实验
- 2, KEY 实验
- 3, EXIT 实验
- 4, UART 实验
- 5, 高分辨率定时器（ESP\_TIMER）实验
- 6, 软件设置 PWM 实验
- 6, 硬件设置 PWM 实验
- 7, SPILCD 实验
- 8, RTC 实验
- 9, 内部温度传感器实验
- 10, RNG 随机数实验
- 11, SD 卡实验
- 12, SPI 文件系统实验
- 13, 汉字显示实验
- 14, 图片显示实验
- 15, USB 虚拟串口（Slave）实验
- 16, FLASH 模拟 U 盘实验
- 17, SD 卡模拟 U 盘实验

## 第十章 LED 实验

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

本章分为以下几个小节：

10.1 GPIO&LED 简介

10.2 硬件设计

10.3 程序设计

10.4 下载验证

### 10.1 GPIO&LED 简介

#### 10.1.1 GPIO 简介

GPIO 是负责控制或采集外部器件信息的外设，主要负责输入输出功能。在第三章节中，作者详细阐述了 ESP32-S3 芯片具有 45 个物理 GPIO 管脚，涵盖 GPIO0 至 GPIO21 以及 GPIO26 至 GPIO28 的广泛范围。然而，相较于 ESP32-S3 芯片，ATK-MWS3S 模组引出的 GPIO 管脚数量较少，仅有 36 个。尽管如此，这些管脚均具备通用 IO 功能，并且可以通过内部 IO MUX（复用矩阵）灵活复用为其他功能，这充分展现了 ESP32-S3 芯片的强大和灵活性。关于 ESP32-S3 的 IO MUX 和 GPIO 交换矩阵的内容已在第三章详细阐述，为避免重复，此处不再赘述。以下是 ATK-MWS3S 模组的 GPIO 分布图。

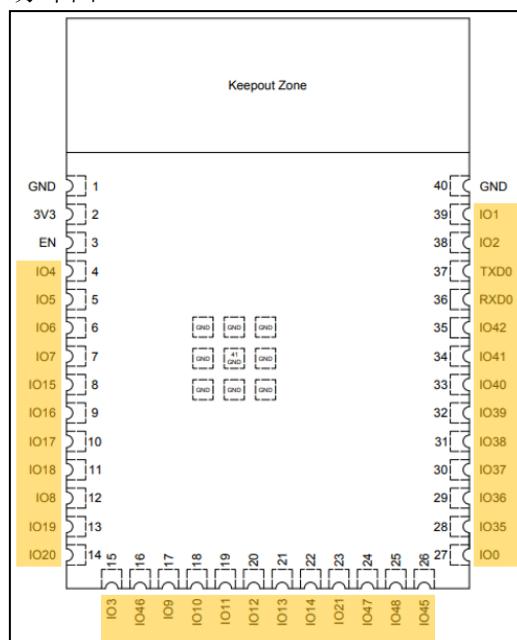


图 10.1.1.1 ATK-MWS3S 模组 GPIO 分布图

从上面的图示中可见，黄色区域的管脚均可作为普通的 IO 端口使用。因此，在控制 LED 灯时，我们可以自由选择任意一个管脚进行操作。但请注意，部分 IO 端口可能与 Flash 或 PSRAM 等元件的管脚相关联，这就需要开发者在操作过程中参考相关技术手册，以避免潜在的问题。在正点原子的 DNESP32S3M 最小系统板中，模组的 IO1 被用来连接 LED 的负极，因此在本章的实验中，我们将主要对 IO1 进行操作。

#### 10.1.2 LED 简介

LED，即发光二极管，其发光原理基于半导体的特性。在半导体中，有两类重要的载流子：

电子，主要存在于 n 型半导体中；而空穴，则主要存在于 p 型半导体中。当 n 型半导体与 p 型半导体材料接触时，它们的交界处会形成一个特殊的层结。当对这个层结施加适当的电压时，层结中的空穴与电子会发生重组，并释放出能量。这些能量会以光子的形式被释放出来，从而产生可见光。这就是 LED 发光的基本原理。

### 1, LED 灯驱动原理

LED 驱动是指通过稳定的电源为 LED 提供适宜的电流和电压，确保其正常发光。LED 驱动方式主要有恒流和恒压两种，其中，恒流驱动因其能限定电流而备受青睐。由于 LED 灯对电流变化极为敏感，一旦电流超过其额定值，可能导致损坏。因此，恒流驱动通过确保电流的稳定性，进而保障 LED 的安全运行。

### 2, LED 灯驱动方式

下面，我们来看一下 LED 两种驱动方式。

(1) 灌入电流接法。指的是 LED 的供电电流是由外部提供电流，将电流灌入我们的 MCU；风险是当外部电源出现变化时，会导致 MCU 的引脚烧坏。其接法如下图所示。

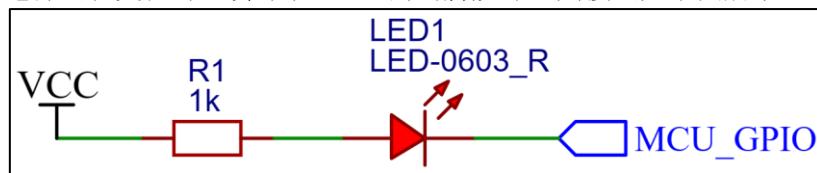


图 10.1.2.1 灌入电流接法

(2) 输出电流接法。指的是由 MCU 提供电压电流，将电流输出给 LED；如果使用 MCU 的 GPIO 直接驱动 LED，则驱动能力较弱，可能无法提供足够的电流驱动 LED。其接法如下图所示。

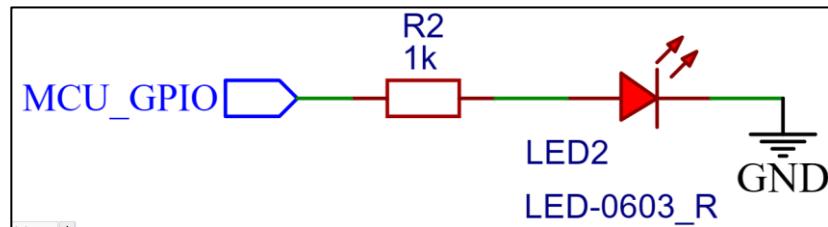


图 10.1.2.2 输出电流接法

DNESP32S3M 最小系统板上的 LED 采用灌入电流接法，这种方式避免了 MCU 直接提供电压电流来驱动 LED，从而有效减轻了 MCU 的负载。这使得 MCU 能够更加专注于执行其他核心任务，进而提升了整体系统的性能和稳定性。

## 10.2 硬件设计

### 10.2.1 例程功能

实验现象： LED 灯以 500ms 的频率交替闪烁。

### 10.2.2 硬件资源

1.LED:

LED-IO1

### 10.2.3 原理图

本章用到的硬件有 LED 灯。电路在 DNESP32S3M 最小系统板上已经连接好，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。

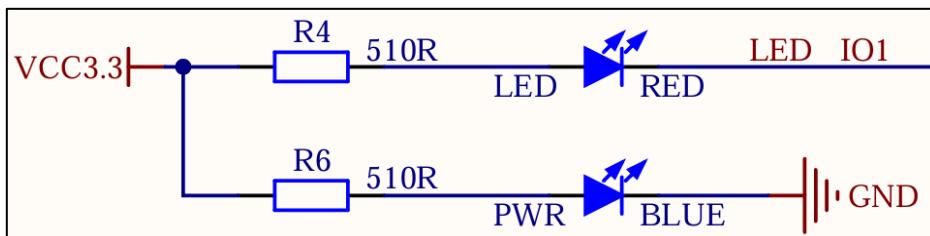


图 10.2.3.1 LED 连接原理图

从上图可知，若 IO1 输出低电平时，则 LED 亮起，反之，熄灭。

## 10.3 程序设计

了解了 ESP32 的 GPIO 结构以及我们的实验功能，下面开始设计程序。

### 10.3.1 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：

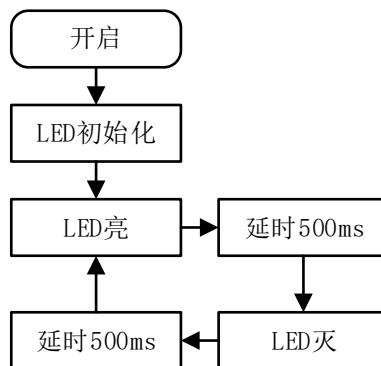


图 10.3.1.1 LED 实验程序流程图

### 10.3.2 GPIO 函数解析

ESP-IDF 提供了丰富的 GPIO 操作函数，开发者可以在 `esp-idf-v5.1.2\components\driver\gpio` 路径下找到相关的 `gpio.c` 和 `gpio.h` 文件。在 `gpio.h` 头文件中，你可以找到 ESP32-S3 的所有 GPIO 函数定义。接下来，作者将介绍一些常用的 GPIO 函数，这些函数的描述及其作用如下：

#### 1. GPIO 配置函数

该函数用来配置 GPIO 的模式、上下拉等功能，其函数原型如下所示：

```
esp_err_t gpio_config(const gpio_config_t *pGPIOConfig)
```

该函数的形参描述如下表所示：

参数	描述
pGPIOConfig	GPIO 结构体

表 10.3.2.1 `gpio_config` 函数形参描述

返回值：ESP\_OK 表示配置成功，ESP\_FAIL 表示配置失败。

`pGPIOConfig` 为 GPIO 配置结构体指针，下面来看一下 `gpio_config_t` 结构体中的变量。

```
/* GPIO 配置参数 */
typedef struct {
    uint64_t pin_bit_mask;          /* 配置引脚位 */
    gpio_mode_t mode;              /* 设置引脚模式 */
    gpio_pullup_t pull_up_en;      /* 设置上拉 */
    gpio_pulldown_t pull_down_en;  /* 设置下拉 */
    gpio_int_type_t intr_type;     /* 中断配置 */
} gpio_config_t;
```

关于各个参数有哪些看下表说明：

类型	类型说明	可填参数	参数说明
.pin_bit_mask	引脚位	(1 << x)其中 x 为 ESP32S3 中可用 GPIO	设置的引脚位，比如本实验用到的 IO1 引脚，则写为：(1ull << GPIO_NUM_1)
.mode	引脚模式	GPIO_MODE_DISABLE	失能输入输出模式
		GPIO_MODE_INPUT	仅输入模式
		GPIO_MODE_OUTPUT	仅输出模式
		GPIO_MODE_OUTPUT_OD	输出开漏模式
		GPIO_MODE_INPUT_OUTPUT_OD	输入输出开漏模式
		GPIO_MODE_INPUT_OUTPUT	输入输出模式
.pull_up_en	配置上拉	GPIO_PULLUP_DISABLE	失能上拉
		GPIO_PULLUP_ENABLE	使能上拉
.pull_down_en	配置下拉	GPIO_PULLDOWN_DISABLE	失能下拉
		GPIO_PULLDOWN_ENABLE	使能下拉
其它		GPIO_PULLUP_ONLY	仅上拉
		GPIO_PULLDOWN_ONLY	仅下拉
		GPIO_PULLUP_PULLDOWN	上拉+下拉
		GPIO_FLOATING	浮空
.intr_type	中断配置	GPIO_INTR_DISABLE	失能中断
		GPIO_INTR_POSEDGE	上升沿
		GPIO_INTR_NEGEDGE	下降沿
		GPIO_INTR_ANYEDGE	上升沿和下降沿
		GPIO_INTR_LOW_LEVEL	输入低电平触发
		GPIO_INTR_HIGH_LEVEL	输入高电平触发
		GPIO_INTR_DISABLE	失能中断
		GPIO_INTR_POSEDGE	上升沿

表 10.3.2.2 gpio\_config\_t 结构体的参数描述

在上表中，可填参数均可在 `gpio_types.h` 文件中找到。这些参数通常是通过枚举类型（enum）定义的，它们为特定的 GPIO 模式或配置提供了预定义的数值。当我们需要为结构体变量（如 `gpio_mode_t`）设置参数时，我们可以查阅 `gpio_types.h` 文件，找到对应的枚举类型，并从中选择适当的数值。这样，我们可以确保为 GPIO 接口设置的模式或配置是准确和有效的。

## 2. 设置管脚输出电平

该函数用于配置某个管脚输出电平，该函数原型如下所示：

```
esp_err_t gpio_set_level(gpio_num_t gpio_num, uint32_t level);
```

该函数的形参描述如下表所示：

参数	描述
gpio_num	GPIO 引脚号。（在 <code>gpio_types.h</code> 文件中枚举 <code>gpio_num_t</code> 有定义）
level	GPIO 引脚输出电平。0：低电平，1：高电平

表 10.3.2.3 gpio\_set\_level 函数形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

## 3. 获取管脚电平

该函数用于获取某个管脚的电平，该函数原型如下所示：

```
esp_err_t gpio_get_level(gpio_num_t gpio_num);
```

该函数的形参描述如下表所示：

参数	描述
----	----

gpio_num	GPIO 引脚号。(在 gpio_types.h 文件中枚举 gpio_num_t 有定义)
----------	--

表 10.3.2.4 gpio\_get\_level 函数形参描述

返回值: ESP\_OK 表示获取成功, ESP\_FAIL 表示获取失败。

上述函数, 便是本实验所需的核心 GPIO 函数。对于其他未提及的 GPIO 函数, 我们用到了再去了解。

### 10.3.3 LED 驱动解析

在 IDF 版的 01\_led 例程中, 作者在 01\_led\components\BSP 路径下新增了一个 LED 文件夹, 用于存放 led.c 和 led.h 这两个文件。其中, led.h 文件负责声明 LED 相关的函数和变量, 而 led.c 文件则实现了 LED 的驱动代码。下面, 我们将详细解析这两个文件的实现内容。

#### 1, led.h 文件

```
/* 引脚定义 */
#define LED_GPIO_PIN      GPIO_NUM_1 /* LED 连接的 GPIO 端口 */

/* 引脚的输出的电平状态 */
enum GPIO_OUTPUT_STATE
{
    PIN_RESET,
    PIN_SET
};

/* LED 端口定义 */
#define LED(x)          do { x ? \
                           gpio_set_level(LED_GPIO_PIN, PIN_SET) : \
                           gpio_set_level(LED_GPIO_PIN, PIN_RESET); \
                           } while(0) /* LED 翻转 */

/* LED 取反定义 */
#define LED_TOGGLE()     do { \
                           gpio_set_level(LED_GPIO_PIN, !gpio_get_level(LED_GPIO_PIN)); \
                           } while(0) /* LED 翻转 */

/* 函数声明 */
void led_init(void); /* 初始化 LED */
```

作者巧妙地编写了 LED(x) 宏, 用于控制 IO1 管脚的电平状态。当 x 为 1 时, 该宏会设置 IO1 管脚输出高电平; 反之, 则输出低电平。此外, 作者还定义了 LED\_TOGGLE() 宏, 它能够实现 IO1 管脚电平状态的快速翻转。这些宏的实现均基于之前小节所介绍的函数, 使得对 LED 的控制变得简洁而高效。

#### 2, led.c 文件

```
/***
 * @brief      初始化 LED
 * @param      无
 * @retval     无
 */
void led_init(void)
{
    gpio_config_t gpio_init_struct = {0};
    gpio_init_struct.intr_type = GPIO_INTR_DISABLE;           /* 失能引脚中断 */
    gpio_init_struct.mode = GPIO_MODE_INPUT_OUTPUT;          /* 输入输出模式 */
    gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;        /* 使能上拉 */
    gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE;   /* 失能下拉 */
    gpio_init_struct.pin_bit_mask = 1ull << LED_GPIO_PIN;   /* 设置的引脚的位掩码 */
    gpio_config(&gpio_init_struct);                          /* 配置 GPIO */

    LED(1);                                                 /* 关闭 LED */
}
```

在.c 文件中, 作者首先对 gpio\_init\_struct 结构体变量进行了参数配置。接着, 调用

gpio\_config 函数，利用该配置变量完成了 GPIO 的初始化工作。最后，通过调用 LED(x)宏定义，实现了 LED 灯的关闭操作，即输出了高电平信号。整个流程清晰、简洁，有效地实现了对 LED 灯的控制。

#### 10.3.4 CMakeLists.txt 文件

打开本章节的实验（01\_LED），我们惊奇地发现，在 components/BSP 路径下定义了 CMakeLists.txt 文件（该文件的作用，作者已经在第六章中阐述过）。此文件的作用是将 BSP 文件夹下的驱动程序添加到构建系统中，确保在编译项目工程时能够调用这些驱动程序。下面展示了该驱动 CMakeLists.txt 文件的具体内容。

①源文件路径，指本目录下的所有代码驱动

```
set(src_dirs
    LED)
```

②头文件路径，指本目录下的所有代码驱动

```
set(include_dirs
    LED)
```

③设置依赖库

```
set(requirements
    driver)
```

```
idf_component_register(SRC_DIRS ${src_dirs}
                        INCLUDE_DIRS ${include_dirs} REQUIREMENTS ${requirements})
```

```
component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

从上述描述中，我们了解到①和②处均需要添加相应的驱动程序。以本章节实验为例，在 BSP 文件夹下仅有一个 LED 驱动，因此我们需要在①和②的位置引入该 LED 驱动。而③处所提到的，是 LED 驱动所依赖的 ESP-IDF 中的具体驱动库。本实验中的 LED 驱动调用 io.c 和 gpio.h 文件下的函数实现的，这些文件均位于 esp-idf-v5.1.2\components\driver 文件夹中，因此我们可以确定 LED 驱动所依赖的是 driver 库。通过这样的依赖关系，我们可以确保 LED 驱动的正确集成和调用。

#### 10.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/** @brief 程序入口
 * @param 无
 * @retval 无
 */
void app_main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init(); /* 初始化 NVS */
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    led_init(); /* 初始化 LED */

    while(1)
    {
        LED_TOGGLE();
        vTaskDelay(500); /* 延时 500ms */
    }
}
```

上述应用代码中，作者首先通过调用 `nvs_flash_init` 函数来初始化 NVS。若初始化时遇到没有足够空闲页面或检测到新版本的情况，代码会先擦除整个 NVS 分区，并随后重新进行初始化。这种处理方式旨在确保 NVS 在特定错误条件下能够被重置并重新使用。紧接着，代码调用 `led_init` 函数来初始化 LED。在随后的 `while` 循环中，利用 `LED_TOGGLE()` 宏定义来定期翻转 LED 的电平状态，每次翻转间隔为 500 毫秒，从而实现了 LED 的闪烁效果。

## 10.4 下载验证

下载完之后，可以看到 LED 以每次 500ms 闪烁。

## 第十一章 KEY 实验

在上一章，我们详细讲解了 GPIO 的输出模式，并演示了如何利用它来控制 LED 的亮灭。而在本章中，我们将重点关注 GPIO 的输入模式配置，学会如何获取外部的输入信号，例如检测按键的状态。通过学习本章内容，开发者将能够掌握 GPIO 作为输入模式的使用方法，进一步扩展其在嵌入式系统开发中的应用能力。

本章分为以下几个小节：

11.1 独立按键基础知识

11.2 硬件设计

11.3 程序设计

11.4 下载验证

### 11.1 独立按键基础知识

独立按键是一种简洁高效的输入设备，广泛应用于各类电子设备中，实现基础的用户交互功能。其工作原理主要基于机械开关的触发机制，当用户按下按键时，便能执行相应的操作。独立按键在尺寸、形状和颜色上都具有多样性，便于用户进行辨识和使用，满足不同场景下的需求。

#### 1. 独立按键原理

独立按键的原理主要依赖于机械触点和电气触点之间的相互作用。在未被按下时，触点保持分离状态，电路处于断开状态。然而，当用户按下按键时，在弹簧和导电片的共同作用下，触点会闭合，从而使电路连通。此时，微控制器能够检测到按键触发的信号，进而执行相应的操作。这种基于物理触点的设计使得独立按键既稳定又可靠，广泛应用于各种电子设备中。

#### 2. 消抖措施

机械按键在闭合与分开的过程中，由于机械振动（类似于弹簧效应）的存在，可能导致开关状态在短时间内频繁切换，这种现象被称为按键抖动。下图是独立按键抖动波形图。

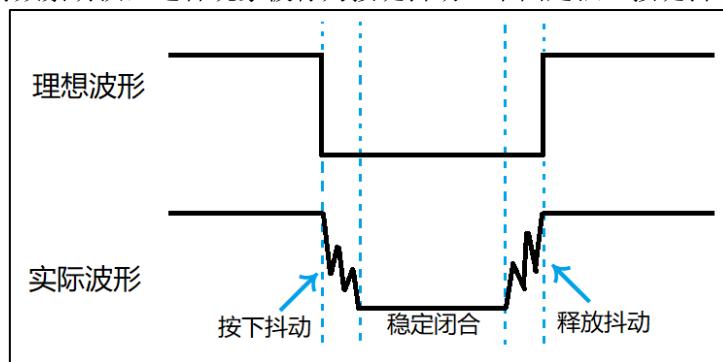


图 11.1.1 独立按键抖动波形图

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）。

为了消除这种抖动，我们通常采用软件消抖和硬件消抖两种主要方法：

(1) 软件消抖：主要是通过编程的方法，设定一个延迟或计时器，确保在一定的时间内只读取一次按键状态，避免抖动对程序的影响。

(2) 硬件消抖：在按键电路中加入元器件如电阻、电容组成的 RC 低通滤波器，对按键信号进行平滑处理，降低抖动的影响。

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。

## 11.2 硬件设计

### 11.2.1 例程功能

实验现象：按下 BOOT 按键可控制 LED 状态翻转。

### 11.2.2 硬件资源

1. LED  
LED-IO1
2. 按键  
BOOT-IO0

### 11.2.3 原理图

本章实验使用的一个 ESP32-S3 最小系统板板载按键：BOOT 按键，其于板载 MCU 的连接原理图，如下图所示：

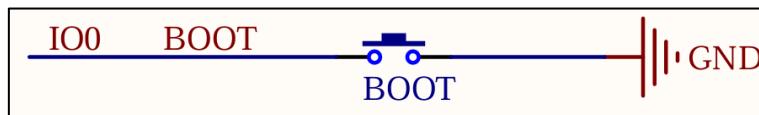


图 11.2.3.1 按键与 MCU 的连接原理图

从上面的原理图中可以看出，BOOT 按键的一端连接到了电源负极，而另一端分别与 MCU 的 BOOT 引脚相连接，因此当按键被按下时，MCU 对应的引脚都能够读取到低电平的状态，而当松开按键后，MCU 对应的引脚读取到的电平状态却是不确定的，因此用于读取 BOOT 按键的 BOOT 引脚不仅要配置为输入模式，还需要配置成上拉。

## 11.3 程序设计

### 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：

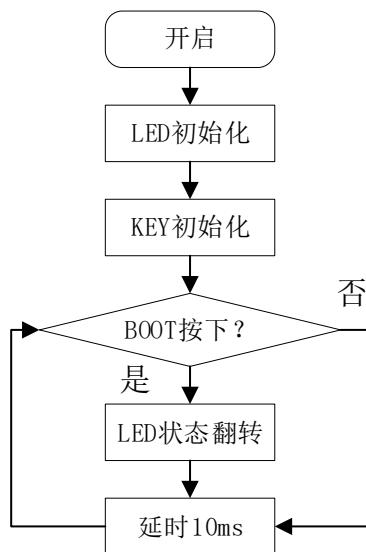


图 11.3.1.1 KEY 实验程序流程图

### 11.3.2 GPIO 函数解析

GPIO 函数已在 10.3.2 章节中详细阐述，为避免重复，此处不再赘述。建议读者查阅第十章的函数解析章节，以获取更多关于 GPIO 函数的信息。

### 11.3.3 KEY 驱动解析

在 IDF 版的 02\_key 例程中，作者在 02\_key\components\BSP 路径下新增了一个 KEY 文件夹，用于存放 key.c 和 key.h 这两个文件。其中，key.h 文件负责声明 KEY 相关的函数和变量，而 key.c 文件则实现了 KEY 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

#### 1, key.h 文件

```
/* 引脚定义 */
#define BOOT_GPIO_PIN    GPIO_NUM_0

/* IO 操作 */
#define BOOT             gpio_get_level(BOOT_GPIO_PIN)

/* 按键按下定义 */
#define BOOT_PRES        1           /* BOOT 按键按下 */

/* 函数声明 */
void key_init(void);           /* 初始化按键 */
uint8_t key_scan(uint8_t mode); /* 按键扫描函数 */
```

此文件的核心内容已较为明确，无需过多阐述。它主要定义了 BOOT 宏，用于获取 IO0 的状态，并声明了 key\_init 和 key\_scan 函数，以便外部文件能够调用这些函数。通过这些声明和定义，该文件为其他部分的代码提供了必要的接口和功能支持。

#### 2, key.c 文件

此文件中定义了两个函数，分别为 key\_init 和 key\_scan。接下来，我将对这两个函数进行详细的解析。

##### (1) key\_init 函数

```
/***
 * @brief      初始化按键
 * @param      无
 * @retval     无
 */
void key_init(void)
{
    gpio_config_t gpio_init_struct;

    gpio_init_struct.intr_type = GPIO_INTR_DISABLE;          /* 失能引脚中断 */
    gpio_init_struct.mode = GPIO_MODE_INPUT;                 /* 输入模式 */
    gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;       /* 使能上拉 */
    gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE; /* 失能下拉 */
    gpio_init_struct.pin_bit_mask = 1ull << BOOT_GPIO_PIN; /* BOOT 按键引脚 */
    gpio_config(&gpio_init_struct);                         /* 配置使能 */
}
```

该函数主要配置 IO0 管脚为输入模式，这样就可以获取 IO0 的电平状态了。

##### (2) key\_scan 函数

```
/***
 * @brief      按键扫描函数
 * @param      mode:0 / 1, 具体含义如下:
 *            0, 不支持连续按(当按键按下不放时, 只有第一次调用会返回键值,
 *            必须松开以后, 再次按下才会返回其他键值)
 *            1, 支持连续按(当按键按下不放时, 每次调用该函数都会返回键值)
 * @retval     键值, 定义如下:
 *            BOOT_PRES, 1, BOOT 按下
 */

```

```

uint8_t key_scan(uint8_t mode)
{
    uint8_t keyval = 0;
    static uint8_t key_boot = 1; /* 按键松开标志 */

    if (mode)
    {
        key_boot = 1;
    }

    if (key_boot && (BOOT == 0)) /* 按键松开标志为 1，且有任意一个按键按下了 */
    {
        vTaskDelay(10); /* 去抖动 */
        key_boot = 0;

        if (BOOT == 0)
        {
            keyval = BOOT_PRES;
        }
    }
    else if (BOOT == 1)
    {
        key_boot = 1;
    }

    return keyval; /* 返回键值 */
}

```

此函数只有一个形参 mode，用于设置按键是否支持连续按下模式。当 mode 为 0 时，表示按键不支持连续按下；反之，则支持连续按下。值得注意的是，该函数内部已经对按键进行了消抖延时处理，因此，在其他地方调用此函数时，无需再进行额外的按键消抖操作。

#### 11.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    KEY
    LED)

set(include_dirs
    KEY
    LED)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIREMENTS ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 KEY 驱动需要由开发者自行添加，以确保 KEY 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 KEY 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 11.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/** 
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)

```

```
{  
    uint8_t key;  
    esp_err_t ret;  
  
    ret = nvs_flash_init(); /* 初始化 NVS */  
  
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES  
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)  
    {  
        ESP_ERROR_CHECK(nvs_flash_erase());  
        ret = nvs_flash_init();  
    }  
  
    led_init(); /* 初始化 LED */  
    key_init(); /* KEY 初始化 */  
  
    while(1)  
    {  
        key = key_scan(0); /* 获取键值 */  
  
        switch (key)  
        {  
            case BOOT_PRES: /* BOOT 被按下 */  
            {  
                LED_TOGGLE(); /* LED 状态翻转 */  
                break;  
            }  
            default:  
            {  
                break;  
            }  
        }  
  
        vTaskDelay(10);  
    }  
}
```

可以看到应用代码中，在初始化完 LED 和按键后，就进入了一个 while 循环，在循环中，每间隔 10 毫秒就调用 key\_scan() 函数扫描以此按键的状态，如果扫描到 BOOT 按键被按下，则反转对应 LED 的亮灭状态。

## 11.4 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED 是处于亮起的状态，若此时按下并释放一次 BOOT 按键，则能够看到 LED 的亮灭状态发生了一次翻转，与预期的实验现象效果相符。

## 第十二章 EXIT 实验

本章将介绍如何将 GPIO 引脚作为外部中断输入来使用。通过本章的学习，开发者将学习到 GPIO 作为外部中断输入的使用。

本章分为如下几个小节：

- 12.1 外部中断简介
- 12.2 硬件设计
- 12.3 程序设计
- 12.4 下载验证

### 12.1 中断简介

在上一章节中，我们虽然实现了 GPIO 口输入功能的读取，但代码始终在检测 IO 输入口的变化，导致在代码量增加时，按键检测部分的轮询效率降低。尤其在某些特定场合，如按键可能一天才被按下一次，实时检测将造成大量时间浪费。为优化此问题，我们引入了外部中断的概念。外部中断即在按键被按下（触发中断）时执行相关功能，从而显著节省 CPU 资源，因此在实际项目中应用广泛。

#### 1. 什么是外部中断

外部中断属于硬件中断，由微控制器外部事件触发。微控制器的特定引脚被设计为对特定事件（如按钮按压、传感器信号变化等）作出响应，这些引脚通常称为“外部中断引脚”。一旦外部中断事件发生，当前程序执行将立即暂停，并跳转到相应的中断服务程序（ISR）进行处理。处理完毕后，程序会恢复执行，从被中断的地方继续。

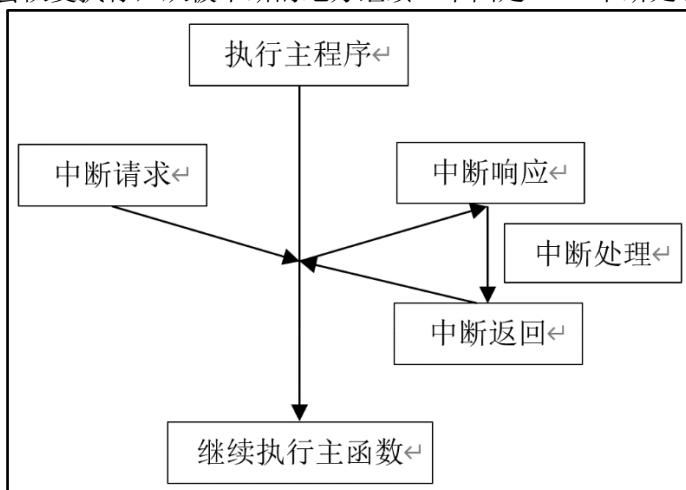


图 12.1.1 CPU 中断处理过程

对于嵌入式和实时系统而言，外部中断的使用至关重要，它能使系统对外部事件作出即时响应，极大提升系统效率和实时性。

#### 2. ESP32-S3 外部中断

ESP32 S3 的外部中断具备两种触摸类型：

- (1) 电平触发：高、低电平触发，要求保持中断的电平状态直到 CPU 响应。
- (2) 边沿触发：上升沿和下降沿触发，这类型的中断一旦触发，CPU 即可响应。

ESP32S3 的外部中断功能能够以非常精确的方式捕捉外部事件的触发。开发者可以通过配置中断触发方式（如上升沿、下降沿、任意电平、低电平保持、高电平保持等）来适应不同的外部事件，并在事件发生时立即中断当前程序的执行，转而执行中断服务函数。

ESP32-S3 支持六级中断，同时支持中断嵌套，也就是优先级中断可以被高优先级中断打断。如下表 12.1.2 中的优先级一栏，数字越大表明该中断的优先级越高。其中，NMI 中断拥有最高优先级，此类中断已经触发，CPU 必须处理。

中断号	类别	种类	优先级
0	外部中断	电平触发	1
1	外部中断	电平触发	1
2	外部中断	电平触发	1
3	外部中断	电平触发	1
4	外部中断	电平触发	1
5	外部中断	电平触发	1
6	内部中断	定时器 0	1
7	内部中断	软件	1
8	外部中断	电平触发	1
9	外部中断	电平触发	1
10	外部中断	边沿触发	1
11	内部中断	解析	3
12	外部中断	电平触发	1
13	外部中断	电平触发	1
14	外部中断	NMI	NMI
15	内部中断	定时器 1	3
16	内部中断	定时器 2	5
17	外部中断	电平触发	1
18	外部中断	电平触发	1
19	外部中断	电平触发	2
20	外部中断	电平触发	2
21	外部中断	电平触发	2
22	外部中断	边沿触发	3
23	外部中断	电平触发	3
24	外部中断	电平触发	4
25	外部中断	电平触发	4
26	外部中断	电平触发	5
27	外部中断	电平触发	3
28	外部中断	边沿	4
29	内部中断	软件	3
30	外部中断	边沿	4
31	外部中断	电平触发	5

表 12.1.1 ESP32S3 中断

在 ESP32S3 中，中断系统被用于响应各种内部和外部事件。这些中断按照其触发方式和优先级进行分类。上表详细列出了 ESP32S3 的中断号、类别、种类以及相应的优先级。通过配置这些中断，开发者可以实现对各种事件的及时响应和处理，提高系统的效率和实时性。

(1) 中断号：每个中断的唯一标识符，用于在程序中引用和配置特定的中断。

(2) 类别：中断的来源类型，分为外部中断和内部中断。外部中断由外部设备或信号触发，如按键、传感器等；内部中断则由微控制器内部的硬件事件触发，如定时器溢出、软件中断等。

(3) 种类：中断的触发方式，包括电平触发和边沿触发。电平触发是在输入信号达到特定电平（如高电平或低电平）时触发中断；边沿触发则是在输入信号从一种电平状态变化到另一种状态时触发中断。

(4) 优先级：中断的响应优先级。当多个中断同时发生时，微控制器会根据中断的优先级来决定先处理哪个中断。较高的优先级意味着中断将优先得到处理。

在开发过程中，开发者可以根据实际需求配置中断的触发方式、优先级等参数，以实现高效、可靠的事件处理机制。

## 12.2 硬件设计

### 12.2.1 例程功能

实验现象：按下 BOOT 按键可控制 LED 状态翻转。

### 12.2.2 硬件资源

1. LED  
LED-IO1
2. 按键  
BOOT-IO0

### 12.2.3 原理图

BOOT 原理图已在 11.2.3 章节中详细阐述，为避免重复，此处不再赘述。

## 12.3 程序设计

### 12.3.1 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：

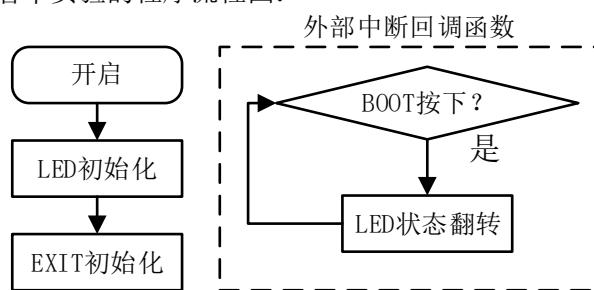


图 12.3.1.1 EXIT 实验程序流程图

### 12.3.2 EXIT 函数解析

接下来，作者将介绍一些常用的 GPIO EXIT 函数，这些函数的描述及其作用如下：

#### 1, 注册中断函数

该函数用来注册中断服务，该函数原型如下所示：

```
void gpio_install_isr_service(esp_intr_alloc_flag_t flags);
```

该函数的形参描述如下表所示：

参数	描述	
flags	中断标志位	
	ESP_INTR_FLAG_LEVEL1	使用 Level 1 中断级别。在中断服务程序执行期间禁用同级别的中断
	ESP_INTR_FLAG_LEVEL2	使用 Level 2 中断级别。在中断服务程序执行期间禁用同级别和 Level 1 的中断
	ESP_INTR_FLAG_EDGE	使用边沿触发方式
	ESP_INTR_FLAG_LOWMED	使用中低水平触发方式
	ESP_INTR_FLAG_HIGH	使用高电平触发方式

表 12.3.2.1 gpio\_install\_isr\_service 函数形参描述

返回值：无。

#### 2, 分配中断函数

该函数设置某个管脚的中断服务函数，该函数原型如下所示：

```
esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num,
                               gpio_isr_t isr_handler, void* args);
```

该函数的形参描述如下表所示：

参数	描述
gpio_num	GPIO 引脚号，指定要分配中断处理程序的 GPIO 引脚
isr_handler	指向中断处理函数的函数指针。中断处理函数是一个用户定义的回调函数，将在中断发生时执行
args	传递给中断处理程序的参数。这是一个指向用户特定数据的指针，可以在中断处理程序中使用

表 12.3.2.2 gpio\_isr\_handler\_add 函数形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

实现一个中断服务程序的回调函数，在函数中处理中断响应。（其中函数名可以随意起名，但是要符合 C 语言标准）中断处理函数需要声明为 IRAM\_ATTR，以确保其运行在内存中的可执行区域。下面是中断函数的模板。

```
void IRAM_ATTR gpio_isr_handler(void* arg) {
    /* 处理中断响应 */
}
```

### 3. 开启外部中断函数

该函数用来配置某个管脚开启外部中断，该函数原型如下所示：

```
void gpio_intr_enable(gpio_num_t gpio_num)
```

该函数的形参描述如下表所示：

参数	描述
gpio_num	GPIO 引脚号，指定要分配中断处理程序的 GPIO 引脚

表 12.3.2.3 gpio\_intr\_enable 函数形参描述

返回值：无。

注意：在使用 gpio\_intr\_enable() 函数之前，开发者需要先通过 gpio\_install\_isr\_service() 函数和 gpio\_isr\_handler\_add() 函数来安装和注册中断处理程序。

### 12.3.3 EXIT 驱动解析

在 IDF 版的 03\_exit 例程中，作者在 03\_exit\components\BSP 路径下新增了一个 EXIT 文件夹，用于存放 exit.c 和 exit.h 这两个文件。其中，exit.h 文件负责声明 EXIT 相关的函数和变量，而 exit.c 文件则实现了 EXIT 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

#### 1. exit.h 文件

```
/* 引脚定义 */
#define BOOT_INT_GPIO_PIN    GPIO_NUM_0

/* IO 操作 */
#define BOOT                  gpio_get_level(BOOT_INT_GPIO_PIN)

/* 函数声明 */
void exit_init(void); /* 外部中断初始化程序 */
```

#### 2. exit.c 文件

```
/**
 * @brief      外部中断服务函数
 * @param      arg: 中断引脚号
 * @note       IRAM_ATTR: 这里的 IRAM_ATTR 属性用于将中断处理函数存储在内部 RAM 中，  
        目的在于减少延迟
 * @retval     无
 */
static void IRAM_ATTR exit_gpio_isr_handler(void *arg)
{
    uint32_t gpio_num = (uint32_t) arg;

    if (gpio_num == BOOT_INT_GPIO_PIN)
    {
```

```

        LED_TOGGLE();
    }

}

/***
 * @brief      外部中断初始化程序
 * @param      无
 * @retval     无
 */
void exit_init(void)
{
    gpio_config_t gpio_init_struct;

    /* 配置 BOOT 引脚和外部中断 */
    gpio_init_struct.mode = GPIO_MODE_INPUT;                                /* 选择为输入模式 */
    gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;                      /* 上拉使能 */
    gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE;                 /* 下拉失能 */
    gpio_init_struct.intr_type = GPIO_INTR_NEGEDGE;                         /* 下降沿触发 */
    gpio_init_struct.pin_bit_mask = 1ull << BOOT_INT_GPIO_PIN;
    gpio_config(&gpio_init_struct);                                         /* 配置使能 */

    /* 注册中断服务 */
    gpio_install_isr_service(ESP_INTR_FLAG_EDGE);

    /* 设置 GPIO 的中断回调函数 */
    gpio_isr_handler_add(BOOT_INT_GPIO_PIN, exit_gpio_isr_handler,
                        (void*) BOOT_INT_GPIO_PIN);
    /* 使能 GPIO 模块中断信号 */
    gpio_intr_enable(BOOT_INT_GPIO_PIN);
}

```

开启管脚的外部中断操作相对简便。首先，需要将管脚配置为下降沿触发（GPIO\_INTR\_NEGEDGE）和输入模式（GPIO\_MODE\_INPUT）。完成配置后，需要调用 `gpio_install_isr_service` 函数来注册中断服务，并调用 `gpio_isr_handler_add` 函数来注册外部中断的回调函数。最后，调用 `gpio_intr_enable` 函数启用外部中断功能。其中，`exit_gpio_isr_handler` 回调函数负责实现 LED 灯状态的切换。

#### 12.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    EXIT
    LED)

set(include_dirs
    EXIT
    LED)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 EXIT 驱动需要由开发者自行添加，以确保 EXIT 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 EXIT 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 12.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 `app_main`。该函数代码如下。

```
/**  
 * @brief      程序入口  
 * @param      无  
 * @retval     无  
 */  
void app_main(void)  
{  
    esp_err_t ret;  
  
    ret = nvs_flash_init(); /* 初始化 NVS */  
  
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES  
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)  
    {  
        ESP_ERROR_CHECK(nvs_flash_erase());  
        ret = nvs_flash_init();  
    }  
  
    led_init();           /* 初始化 LED */  
    exit_init();          /* 初始化按键 */  
  
    while(1)  
    {  
        vTaskDelay(10);  
    }  
}
```

本实验的应用代码很简单，在初始化完 LED 和外部中断后，就进入一个 while 循环，以便通过串口助手查看单片机运行状态。另外，按键控制 LED 亮灭状态翻转的操作都在对应 EXTI 的中断服务函数中完成了。

## 12.4 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED 处于亮起状态，若此时按下 KEY 按键，则能够看到 LED 处于熄灭状态。如此往复该操作，将会看到 LED 处在熄灭与亮起的状态之间转换，与预期的实验现象效果相符。

## 第十三章 UART 实验

本章将介绍使用串口进行数据的收发操作，具体实现 ESP32-S3 与上位机软件的数据通信，ESP32-S3 将接受自上位机软件的数据原原本本地发送回给上位机软件。通过本章的学习，开发者将学习到 UART 和 GPIO 引脚的使用。

本章分为如下几个小节：

- 13.1 串口简介
- 13.2 硬件设计
- 13.3 程序设计
- 13.4 下载验证

### 13.1 串口简介

学习串口前，我们先来了解一下数据通信的一些基础概念。

#### 13.1.1 数据通信的基础概念

在单片机的应用中，数据通信是必不可少的一部分，比如：单片机和上位机、单片机和外围器件之间，它们都有数据通信的需求。由于设备之间的电气特性、传输速率、可靠性要求各不相同，于是就有了各种通信类型、通信协议，我们最常的有：USART、IIC、SPI、CAN、USB 等。下面，我们先来学习数据通信的一些基础概念。

##### 1. 数据通信方式

按数据通信方式分类，可分为串行通信和并行通信两种。串行和并行的对比如下图所示：

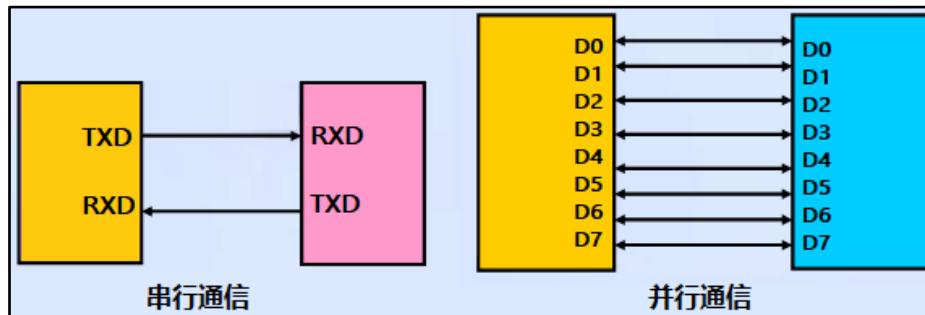


图 13.1.1.1 数据传输方式

串行通信的基本特征是数据逐位顺序依次传输，优点是传输线少、布线成本低、灵活度高等优点，一般用于近距离人机交互，特殊处理后也可以用于远距离，缺点就是传输速率低。

而并行通信是数据各位可以通过多条线同时传输，优点是传输速率高，缺点就是布线成本高，抗干扰能力差因而适用于短距离、高速率的通信。

##### 2. 数据传输方向

根据数据传输方向，通信又可分为全双工、半双工和单工通信。全双工、半双工和单工通信的比较如下图所示：

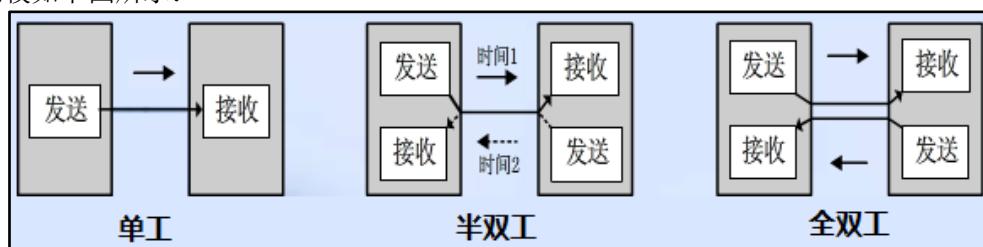


图 13.1.1.2 数据传输方式

单工是指数据传输仅能沿一个方向，不能实现反方向传输，如校园广播。

半双工是指数据传输可以沿着两个方向，但是需要分时进行，如对讲机。

全双工是指数据可以同时进行双向传输，日常的打电话属于这种情形。

这里注意全双工和半双工通信的区别：半双工通信是共用一条线路实现双向通信，而全双工是利用两条线路，一条用于发送数据，另一条用于接收数据。

### 3. 数据同步方式

根据数据同步方式，通信又可分为同步通信和异步通信。同步通信和异步通信比较如下图所示：

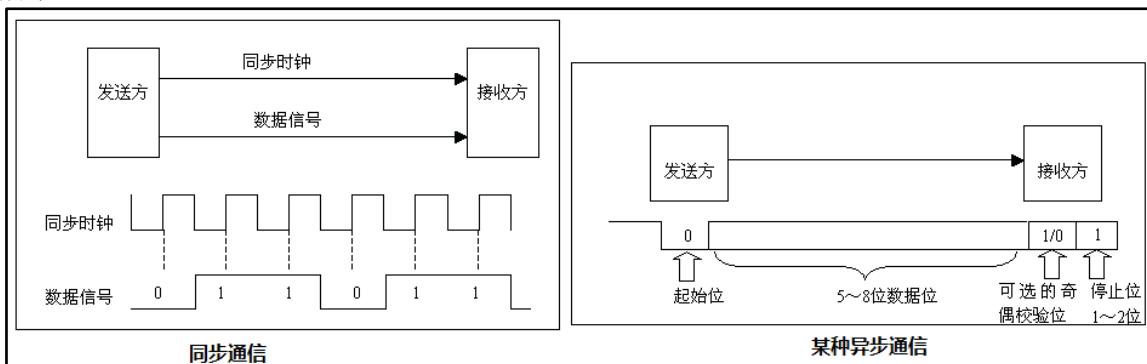


图 13.1.1.3 数据同步方式

同步通信要求通信双方共用同一时钟信号，在总线上保持统一的时序和周期完成信息传输。优点：可以实现高速率、大容量的数据传输，以及点对多点传输。缺点：要求发送时钟和接收时钟保持严格同步，收发双方时钟允许的误差较小，同时硬件复杂。

异步通信不需要时钟信号，而是在数据信号中加入开始位和停止位等一些同步信号，以便使接收端能够正确地将每一个字符接收下来，某些通信中还需要双方约定传输速率。优点：没有时钟信号硬件简单，双方时钟可允许一定误差。缺点：通信速率较低，只适用点对点传输。

### 4. 通信速率

在数字通信系统中，通信速率（传输速率）指数据在信道中传输的速度，它分为两种：传信率和传码率。

传信率：每秒钟传输的信息量，即每秒钟传输的二进制位数，单位为 bit/s（即比特每秒），因而又称为**比特率**。

传码率：每秒钟传输的码元个数，单位为 Baud（即波特每秒），因而又称为**波特率**。

**比特率**和**波特率**这两个概念又常常被人们混淆。比特率很好理解，我们来看看波特率，波特率被传输的是码元，码元是信号被调制后的概念，每个码元都可以表示一定 bit 的数据信息量。举个例子，在 TTL 电平标准的通信中，用 0V 表示逻辑 0，5V 表示逻辑 1，这时候这个码元就可以表示两种状态。如果电平信号 0V、2V、4V 和 6V 分别表示二进制数 00、01、10、11，这时候每一个码元就可以表示四种状态。

由上述可以看出，码元携带一定的比特信息，所以比特率和波特率也是有一定的关系的。

比特率和波特率的关系可以用以下式子表示：

$$\text{比特率} = \text{波特率} * \log_2 M$$

其中 M 表示码元承载的信息量。我们也可以理解 M 为码元的进制数。

举个例子：波特率为 100 Baud，即每秒传输 100 个码元，如果码元采用十六进制编码（即 M=16，代入上述式子），那么这时候的比特率就是 400 bit/s。如果码元采用二进制编码（即 M=2，代入上述式子），那么这时候的比特率就是 100 bit/s。

可以看出采用二进制的时候，波特率和比特率数值上相等。但是这里要注意，它们的相等只是数值相等，其意义上不同，看波特率和比特率单位就知道。由于我们的所用的数字系统都是二进制的，所以有部分人久而久之就直接把波特率和比特率混淆了。

### 13.1.2 串口通信协议简介

串口通信是一种设备间常用的串行通信方式，串口按位（bit）发送和接收字节。尽管比特字节（byte）的串行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。

**串口通信协议**是指规定了数据包的内容，内容包含了起始位、主体数据、校验位及停止位，双方需要约定一致的数据包格式才能正常收发数据的有关规范。在串口通信中，常用的协议包括 RS-232、RS-422 和 RS-485 等。

随着科技的发展，RS-232 在工业上还有广泛的使用，但是在商业技术上，已经慢慢的使用 USB 转串口取代了 RS-232 串口。我们只需要在电路中添加一个 USB 转串口芯片，就可以实现 USB 通信协议和标准 UART 串行通信协议的转换，而我们 DNESP32S3M 最小系统板上的 USB 转串口芯片是 CH343P 这个芯片。

下面我们来学习串口通信协议，这里主要学习串口通信的协议层。

串口通信的数据包由发送设备的 TXD 接口传输到接收设备的 RXD 接口。在串口通信的协议层中，规定了数据包的内容，它由起始位、主体数据、校验位以及停止位组成，通讯双方的数据包格式要约定一致才能正常收发数据，其组成如图 13.1.2.1 所示。

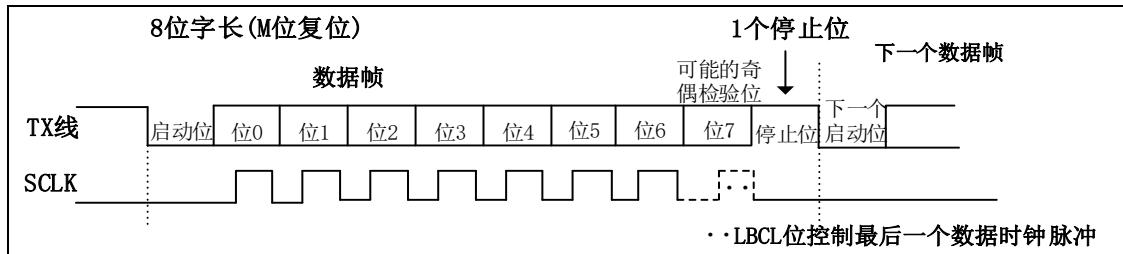


图 13.1.2.1 串口通信协议数据帧格式

串口通信协议数据包组成可以分为波特率和数据帧格式两部分。

### 1. 波特率

本章主要讲解的是串口异步通信，异步通信是不需要时钟信号的，但是这里需要我们约定好两个设备的波特率。波特率表示每秒钟传送的码元符号的个数，所以它决定了数据帧里面每一个位的时间长度。两个要通信的设备的波特率一定要设置相同，我们常见的波特率是 4800、9600、115200 等。

### 2. 数据帧格式

数据帧格式需要我们提前约定好，串口通信的数据帧包括起始位、停止位、有效数据位以及校验位。

#### ①：起始位和停止位

串口通信的一个数据帧是从起始位开始，直到停止位。数据帧中的起始位是由一个逻辑 0 的数据位表示，而数据帧的停止位可以是 0.5、1、1.5 或 2 个逻辑 1 的数据位表示，只要双方约定一致即可。

#### ②：有效数据位

数据帧的起始位之后，就接着是数据位，也称有效数据位，这就是我们真正需要的数据，有效数据位通常会被约定为 5、6、7 或者 8 个位长。有效数据位是低位（LSB）在前，高位（MSB）在后。

#### ③：校验位

校验位可以认为是一个特殊的数据位。校验位一般用来判断接收的数据位有无错误，检验方法有：奇检验、偶检验、0 检验、1 检验以及无检验。下面分别介绍一下：

**奇校验**是指有效数据为和校验位中“1”的个数为奇数，比如一个 8 位长的有效数据为：10101001，总共有 4 个“1”，为达到奇校验效果，校验位设置为“1”，最后传输的数据是 8 位的有效数据加上 1 位的校验位总共 9 位。

**偶校验**与奇校验要求刚好相反，要求帧数据和校验位中“1”的个数为偶数，比如数据帧：11001010，此时数据帧“1”的个数为 4 个，所以偶校验位为“0”。

**0 校验**是指不管有效数据中的内容是什么，校验位总为“0”，**1 校验**是校验位总为“1”。

**无校验**是指数据帧中不包含校验位。

我们一般是使用无校验的情况。

### 13.1.3 ESP32-S3 的 UART 简介

ESP32-S3 芯片中有三个 UART 控制器可供使用，并且兼容不同的 UART 设备。此外，UART 还可以用作红外数据交换(IrDA)或 RS485 调制解调器。三个 UART 控制器分别有一组功能相同的寄存器，分别为 UART0、UART1、UART2，在该实验中我们用到了 UART0。

UART 是一种以字符为导向的通用数据链，可以实现设备间的通信。异步通信不需要在发送数据的过程中添加时钟信息，但这也要求发送端和接收端的速率、停止位以及奇偶校验位等参数的配置要相同，唯有如此通信才能成功。

UART 数据帧始于一个起始位，接着是有效数据，然后是奇偶校验位，最后才是停止位。ESP32-S3 芯片上的 UART 控制器支持多种字符长度和停止位。另外，控制器还支持软、硬件控制流和 GDMA，可以实现无缝高速的数据传输。

### 13.1.4 ESP32-S3 的 UART 框架图介绍

下面先来学习如图 13.1.4.1 所示的 UART 框架图。通过框架图引出 UART 的相关知识，从而有一个很好的整体掌握，对之后的代码开发也会有一个清晰的思路。

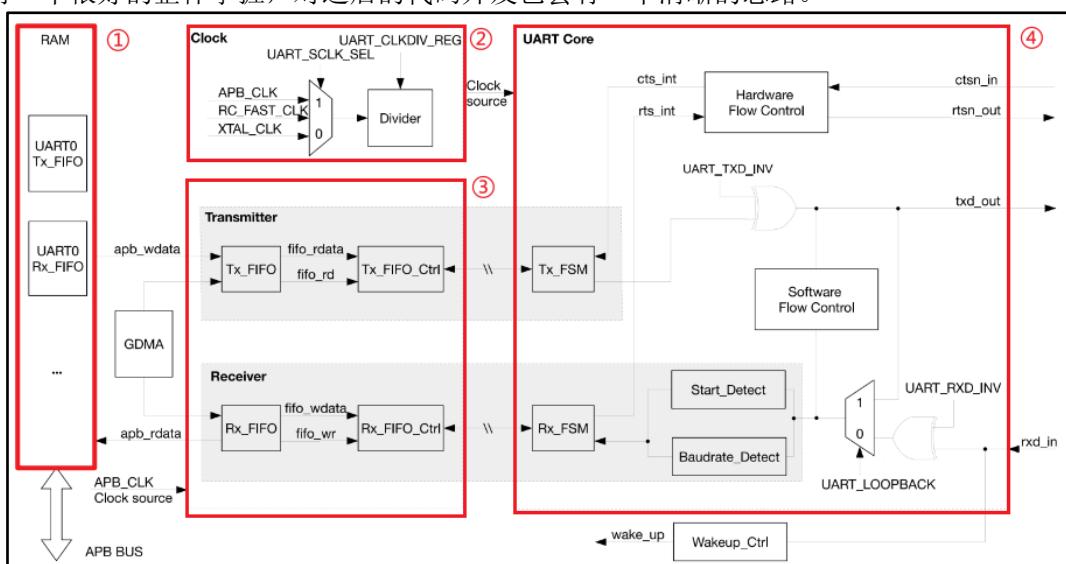


图 13.1.4.1 UART 框架图

为了方便大家理解，我们把整个框图分成几个部分来介绍。

#### ①：RAM

ESP32-S3 芯片中三个 UART 控制器(UART0、UART1、UART2)共用  $1024 \times 8\text{-bit}$  的 RAM 空间，图中①处仅列出了 UART0 的情况。通过配置 `UART_TX_SIZE` 可以对三个 UART 控制器中的其中一个的 Tx\_FIFO 以 1block 为单位进行扩展。同理，配置 `UART_RX_SIZE` 也是一样的。具体的请参考《esp32-s3\_technical\_reference\_manual\_cn》。

#### ②：Clock

UART 作为异步通信的外设，它的寄存器配置模块与 TX/RX/FIFO 都工作在 APB\_CLK 时钟域内，而控制 UART 接收与发送的 Core 模块工作在 UARTCore 时钟域。Clock 有三个时钟源，如图 13.1.4.1 中的②所示，分别为：APB\_CLK、RC\_FAST\_CLK 以及晶振时钟 XTAL\_CLK，他们可以通过配置寄存器 `UART_SCLK_SEL` 来选择使用哪个时钟作为时钟源。选择后的时钟源通过预分频器(Divider)分频后进入 UART Core 模块。该分频器支持小数分频，分频系数为：

$$\text{UART\_SCLK\_DIV\_NUM} + \frac{\text{UART\_SCLK\_DIV\_B}}{\text{UART\_SCLK\_DIV\_A}}$$

支持的分频范围为：1~256。

#### ③：UART 控制器模块

UART 控制器可以分为两个功能块，分别为：发送块(Transmitter)以及接收块(Receiver)。

发送块包含一个发送 FIFO 用于缓存待发送的数据。软件可以通过 APB 总线向 Tx\_FIFO 写数据，也可以通过 GDMA 将数据传入 Tx\_FIFO。Tx\_FIFO\_Ctrl，用于控制 Tx\_FIFO 的读写过程，当 Tx\_FIFO 非空时，Tx\_FSM 通过 Tx\_FIFO\_Ctrl 读取数据，并将数据按照配置的帧格式转化成比特流。比特流输出信号 txd\_out 可以通过配置 UART TXD INV 寄存器实现取反功能。

接收块包含一个接收 FIFO 用于缓存待处理的数据。输入比特流 rxd\_in 可以输入到 UART 控制器。可以通过 UART\_RXD\_INV 寄存器实现取反。Baudrate\_Detect 通过检测最小比特流输入信号的脉宽来测量输入信号的波特率。Start\_Detect 用于检测数据的 START 位，当检测到 START 位之后，Rx\_FSM 通过 Rx\_FIFO\_Ctrl 将帧解析后的数据存入 Rx\_FIFO 中。软件可以通过 APB 总线读取 Rx\_FIFO 中的数据也可以使用 GDMA 方式进行数据接收。

## ④: UART Core

HW\_Flow\_Ctrl 通过标准 UART RTS 和 CTS (rtsn\_out 和 ctsn\_in) 流控信号来控制 rxd\_in 和 txd\_out 的数据流。SW\_Flow\_Ctrl 通过在发送数据流中插入特殊字符以及在接收数据流中检测特殊字符来进行数据流的控制。

## 13.2 硬件设计

### 13.2.1 例程功能

1. 回显串口接收到的数据
  2. 每间隔一定时间，串口发送一段提示信息
  3. LED 闪烁，提示程序正在运行

### 13.2.2 硬件资源

1. LED  
LED - IO1
  2. UART\_NUM\_0  
U0TXD-IO43  
U0RXD-IO44

### 13.2.3 原理图

板载的 USB 转串口芯片的 USB 接口通过板载的 USB UART 端口引出，其原理图如下图。

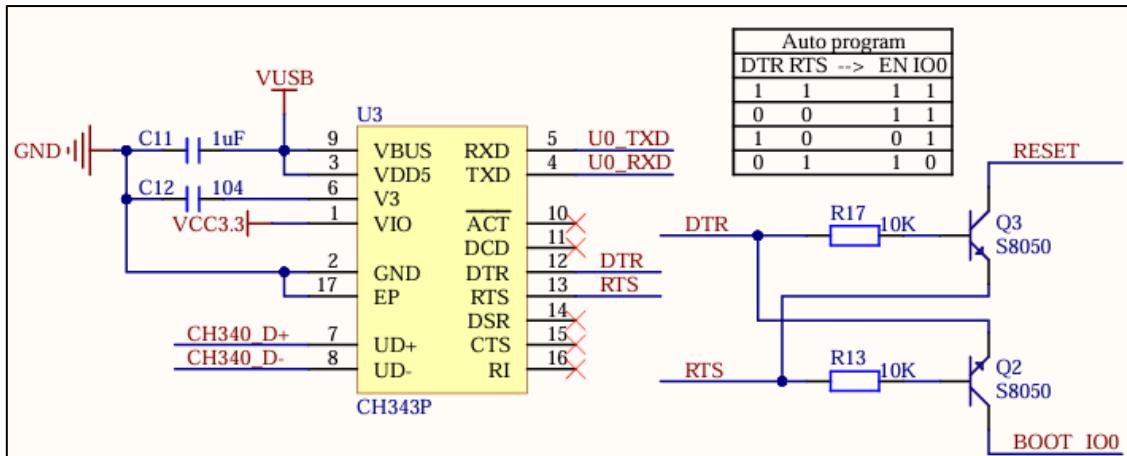


图 13.2.3.3 USB 转串口芯片相关原理图

从以上原理图可以看出，TXD 引脚和 RXD 引脚分别作为发送和接收引脚分别与 USB 转串口芯片的接收和发送引脚进行连接，USB 转串口芯片再通过一对 USB 差分信号连接至 USB UART 的接口，这样一来，ESP32-S3 就可以通过 USB 与 PC 上位机软件进行串口通信了。

另外 ESP32S3 有三个串口，即 UART0、UART1、UART2。其中，DNESP32S3M 最小系统板的串口 0 已经用于自动下载与调试部分，故在实际应用中不建议使用串口 0 与其他设备通信。

### 13.3 程序设计

#### 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：

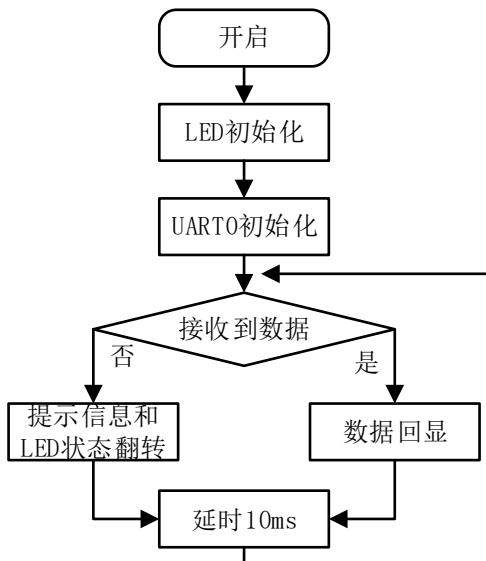


图 13.3.1.1 UART 实验程序流程图

针对本章的实验要求，需要将串口发送引脚配置成复用推挽输出模式，而串口接收引脚配置成上拉输入模式。

#### 13.3.2 UART 函数解析

ESP-IDF 提供了一套 API 来配置串口。要使用串口功能，需要导入必要的头文件：

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

接下来，作者将介绍一些常用的 UART 函数，这些函数的描述及其作用如下：

##### 1. 配置 UART 端口

该函数用来设置指定 UART 端口的通信参数，该函数原型如下所示：

```
esp_err_t uart_param_config(uart_port_t uart_num,
                           const uart_config_t *uart_config)
```

该函数的形参描述如下表所示：

形参	描述
uart_num	UART 外设端口号 例如：UART_NUM_0、UART_NUM_1 等（在 uart.h 文件中有定义）
uart_config	指向 uart_config_t 结构体的指针，包含了 UART 的参数配置信息，需自行定义，并根据 UART 的参数配置填充结构体中的成员变量

表 13.3.2.1 函数 uart\_param\_config() 形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

该函数使用 uart\_config\_t 类型的结构体变量传入 uart 外设的配置参数，该结构体的定义如下所示：

结构体	参数 1	可选值
uart_config_t	.baud_rate： 它代表串口的波特率，即数据传输速率，具体指的是每秒传输的位数。	常见的波特率值包括 9600 和 115200 等。
		UART_DATA_5_BITS

<b>.data_bits:</b> 数据位的数量，也就是每个字节中的位数。	UART_DATA_6_BITS
	UART_DATA_7_BITS
	UART_DATA_8_BITS
<b>.parity:</b> 奇偶校验位的设置。	UART_PARITY_DISABLE: 禁用校验位
	UART_PARITY_EVEN: 偶校验位
	UART_PARITY_ODD: 奇校验位
<b>.stop_bits:</b> 停止位的数量。	UART_STOP_BITS_1: 1个停止位
	UART_STOP_BITS_1_5: 1.5个停止位，仅适用于5数据位
	UART_STOP_BITS_2: 2个停止位
<b>.flow_ctrl:</b> 硬件流控制的设置	UART_HW_FLOWCTRL_DISABLE: 禁用流控制
	UART_HW_FLOWCTRL_RTS: 只启用 RTS 信号流控制
	UART_HW_FLOWCTRL_CTS: 只启用 CTS 信号流控制
<b>.source_clk:</b> 配置时钟源	UART_SCLK_APB: 选择 APB 作为时钟源
	UART_SCLK_RTC: 选择 RTC 作为时钟源
	UART_SCLK_XTAL: 选择 XTAL 作为时钟源
<b>.rx_flow_ctrl_thresh:</b> 硬件控制流阈值	UART_SCLK_DEFAULT: 选择 APB 时钟源为默认选项
	例程中我们设置为：122

表 13.3.2.2 uart\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 `uart_param_config()` 函数，用以实例化串口并返回串口句柄。

## 2. 配置 UART 引脚

该函数设置某个管脚的中断服务函数，该函数原型如下所示：

```
esp_err_t uart_set_pin(uart_port_t uart_num,
                      int tx_io_num,
                      int rx_io_num, int rts_io_num,
                      int cts_io_num);
```

该函数的形参描述如下表所示：

函数的形参描述如下表所示：

形参	描述
uart_num	UART 外设端口号例如：UART_NUM_0、UART_NUM_1 等（在 <code>uart.h</code> 文件中有定义）
tx_io_num	UART 发送引脚的 GPIO 号。若不需要此功能，可将此参数设为-1。
rx_io_num	UART 接收引脚的 GPIO 号。若不需要此功能，可将此参数设为-1。

rts_io_num	UART 请求发送(RTS)引脚的 GPIO 号。若不需要此功能，可将此参数设为-1，例如：UART_PIN_NO_CHANGE = -1
cts_io_num	UART 清除发送(CTS)引脚的 GPIO 号。若不需要此功能，可将此参数设为-1，例如：UART_PIN_NO_CHANGE = -1

表 13.3.2.3 uart\_set\_pin()函数形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

该函数可以将 UART 的发送、接收、RTS 以及 CTS 引脚与指定的 GPIO 引脚进行连接。

### 3. 安装驱动程序

该函数用于安装 UART 驱动程序，并指定发送和接收缓冲区的大小，其函数原型如下所示：

```
esp_err_t uart_driver_install(uart_port_t uart_num,
                             int rx_buffer_size,
                             int tx_buffer_size,
                             int event_queue_size,
                             QueueHandle_t *uart_queue,
                             int intr_alloc_flags)
```

该函数的形参描述，如下表所示：

形参	描述
uart_num	UART 外设端口号例如：UART_NUM_0、UART_NUM_1 等（在 uart.h 文件中有定义）。
rx_buffer_size	UART 接收环形缓冲区大小，用于存储接收到的数据。
tx_buffer_size	UART 发送环形缓冲区大小，用于存储有待发送的数据。
queue_size	UART 驱动程序内部缓冲队列的大小，用于存储待处理的接收和发送数据。
uart_queue	指向用户定义的用于接收数据的队列句柄，在接收数据时，接收到的数据会存储在这个队列中。
intr_alloc_flags	UART 中断分配标志，用于配置中断分配策略。

表 13.3.2.4 函数 uart\_driver\_install()形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

使用 uart\_driver\_install()函数可以方便地初始化 UART，并且指定相应的缓冲区和队列大小以及其他参数。

### 4. 获取数据长度

该函数用于获取接收环形缓冲区中缓存的数据长度，其函数原型如下所示：

```
esp_err_t uart_get_buffered_data_len(uart_port_t uart_num, size_t* size);
```

该函数的形参描述，如下表所示：

形参	描述
uart_num	UART 外设端口号例如：UART_NUM_0、UART_NUM_1 等
size	结构体 size_t 指针所接受缓存的数据长度

表 13.3.2.5 函数 uart\_get\_buffered\_data\_len()形参描述

返回值：ESP\_OK 表示设置成功，ESP\_FAIL 表示设置失败。

### 5. 接收数据

该函数从 UART 接收缓冲区中读取数据，其函数原型如下所示：

```
int uart_read_bytes(uart_port_t uart_num,
                    void *buf,
                    uint32_t length,
                    TickType_t ticks_to_wait)
```

该函数的形参描述，如下表所示：

形参	描述
uart_num	UART 外设端口号
buf	指向缓冲区的指针
length	数据长度
ticks_to_wait	超时等待，RTOS 节拍计数

表 13.3.2.6 函数 uart\_read\_bytes()形参描述

返回值: ESP\_OK 表示设置成功, ESP\_FAIL 表示设置失败。

## 6, 发送数据

该函数将指定的数据写入到 UART 发送缓冲区, 并触发数据的发送, 其函数原型如下所示:

```
int uart_write_bytes(uart_port_t uart_num, const void *src, size_t size)
```

该函数的形参描述, 如下表所示:

形参	描述
uart_num	UART 外设端口号 例如: UART_NUM_0、UART_NUM_1 等
src	指向源数据缓冲区的指针, 包含要发送的数据
size	要发送的数据长度

表 13.3.2.7 函数 uart\_write\_bytes()形参描述

返回值: ESP\_OK 表示设置成功, ESP\_FAIL 表示设置失败。

在使用 uart\_write\_bytes()函数发送数据时, 重要的是要理解该函数的执行机制: 数据首先被复制到 UART 发送缓冲区, 随后函数会返回, 并不会等待数据完全发送完成。因此, 若需确保数据完整无误地发送成功, 应当调用 uart\_wait\_tx\_done()函数进行同步等待, 直至发送过程完全结束。在确认 UART 已成功初始化, 并且已经配置了正确的波特率及其他相关参数之后, 即可调用 uart\_write\_bytes()函数, 将数据准确无误地发送至 UART 设备。

### 13.3.3 UART 驱动解析

在 IDF 版的 04\_uart 例程中, 作者在 04\_uart\components\BSP 路径下新增了一个 UART 文件夹, 用于存放 uart.c 和 uart.h 这两个文件。其中, uart.h 文件负责声明 UART 相关的函数和变量, 而 uart.c 文件则实现了 UART 的驱动代码。下面, 我们将详细解析这两个文件的实现内容。

#### 1, uart.h 文件

```
/* 引脚和串口定义 */
#define USART_UX                      UART_NUM_0
#define USART_TX_GPIO_PIN              GPIO_NUM_43
#define USART_RX_GPIO_PIN              GPIO_NUM_44

/* 串口接收相关定义 */
#define RX_BUF_SIZE                   1024      /* 环形缓冲区大小 */
```

#### 2, uart.c 文件

```
/**
 * @brief      初始化串口
 * @param      baudrate: 波特率, 根据自己需要设置波特率值
 * @note       注意: 必须设置正确的时钟源, 否则串口波特率就会设置异常.
 * @retval     无
 */
void usart_init(uint32_t baudrate)
{
    uart_config_t uart_config;                                /* 串口配置句柄 */

    uart_config.baud_rate = baudrate;                         /* 波特率 */
    uart_config.data_bits = UART_DATA_8_BITS;                  /* 字长为 8 位数据格式 */
    uart_config.parity = UART_PARITY_DISABLE;                 /* 无奇偶校验位 */
    uart_config.stop_bits = UART_STOP_BITS_1;                  /* 一个停止位 */
    uart_config.flow_ctrl = UART_HW_FLOWCTRL_DISABLE;         /* 无硬件控制流 */
    uart_config.source_clk = UART_SCLK_APB;                   /* 配置时钟源 */
    uart_config.rx_flow_ctrl_thresh = 122;                     /* 硬件控制流阈值 */
    uart_param_config(USART_UX, &uart_config);                /* 配置 uart 端口 */

    /* 配置 uart 引脚 */
    uart_set_pin(USART_UX,
                USART_TX_GPIO_PIN,
                USART_RX_GPIO_PIN,
                UART_PIN_NO_CHANGE,
```

```

    UART_PIN_NO_CHANGE);

/* 安装串口驱动 */
uart_driver_install(USART_UX,
                    RX_BUF_SIZE * 2,
                    RX_BUF_SIZE * 2,
                    20,
                    NULL,
                    0);
}

```

uart\_config 是结构体 uart\_config\_t 类型的全局变量，关于 uart\_config\_t 结构体成员的含义以及可选的参数请回到本章节的 13.3.2 小节进行回顾。波特率我们通过传参的方式，赋值给 uart\_config.baud\_rate 这个成员，其他成员也可以通过相应的步骤进行配置。接着，在配置 UART0 作为串口通信端口，并且将 IO44 与 IO43 引脚作为收发引脚后，调用安装串口驱动函数，便完成了 UART 的初始化配置。

ESP32-S3 的串口通讯驱动不需要为 UART0 编写中断回调函数，因为在 ESP32-S3 IDF 库中已经封装了数据读写函数。UART0 通过函数获取 RX 环形缓冲区缓存的数据长度，并判断该数据长度非空后，将其逐一通过读写函数进行操作。

### 13.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    USART
    LED)

set(include_dirs
    USART
    LED)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 USART 驱动需要由开发者自行添加，以确保 USART 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 USART 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

### 13.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main，该函数代码如下：

```

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;
    uint8_t len = 0;
    uint16_t times = 0;
    unsigned char data[RX_BUF_SIZE] = {0};

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {

```

```

ESP_ERROR_CHECK(nvs_flash_erase());
ret = nvs_flash_init();
}

led_init();                                     /* 初始化 LED */
uart_init(115200);                            /* 初始化串口 */

while(1)
{
    /* 获取环形缓冲区数据长度 */
    uart_get_buffered_data_len(USART_UX, (size_t*) &len);

    if (len > 0)                                /* 判断数据长度 */
    {
        memset(data, 0, RX_BUF_SIZE);           /* 对缓冲区清零 */
        printf("\n 您发送的消息为:\n");
        uart_read_bytes(USART_UX, data, len, 100);      /* 读数据 */
        uart_write_bytes(USART_UX,
                          (const char*)data,
                          strlen((const char*)data));          /* 写数据 */
    }
    else
    {
        times++;

        if (times % 5000 == 0)
        {
            printf("\n 正点原子 ATK-DNESP32-S3M 最小系统板 串口实验\n");
            printf("正点原子@ALIENTEK\n\n\n");
        }

        if (times % 200 == 0)
        {
            printf("请输入数据, 以回车键结束\n");
        }

        if (times % 30 == 0)
        {
            LED_TOGGLE();
        }
        vTaskDelay(10);
    }
}
}

```

本实验的实验代码很简单，在完成初始化后，就不断地通过串口通信驱动提供的数据接收并判断数据长度大小，若还未完成数据接收，则每间隔一段时间就使用 printf 函数通过 UART0 打印一段提示信息，若数据接收完毕，则将数据原原本本地使用 printf 函数通过 UART0 打印出去，实现数据的回显功能。

### 13.4 下载验证

在完成编译和烧录操作后，需要将 DNESP32S3M 最小系统板的 USB UART 接口与 PC 的 USB 接口通过具有数据传输功能的数据线进行连接。接着打开 PC 上的 ATK-XCOM 串口调试助手软件，选择好正确的 COM 端口和相关的配置后，就能看到串口调试助手上每间隔一段时间就打印一次“请输入数据，以回车键结束”，接下来就可以根据提示通过串口调试助手发送一段任意的数据（以回车换行结束），随后立马就能看到串口调试助手上显示发送出去的数据，这就是本实验实现的数据回显功能。

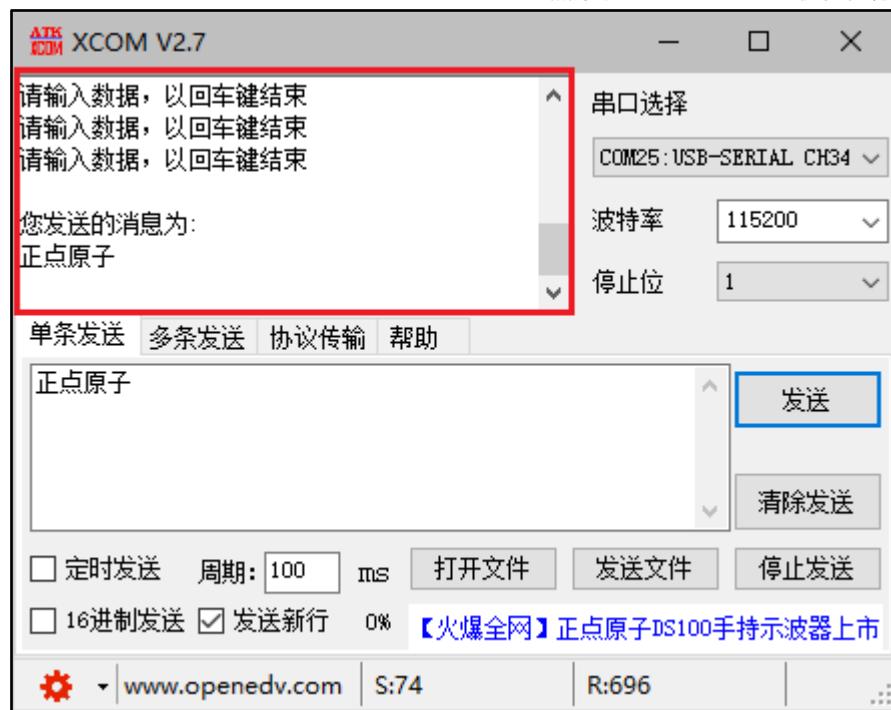


图 13.4.1 串口调试助手显示的信息

## 第十四章 ESPTIMER 实验

ESP32-S3 芯片内置一组 52 位系统定时器。该定时器可用于生成操作系统所需的滴答定时中断，也可用作普通定时器生成周期或单次延时中断。通过本章的学习，开发者将学习到高分辨率定时器的使用。

本章分为如下几个小节：

- 14.1 定时器简介
- 14.2 硬件设计
- 14.3 程序设计
- 14.4 下载验证

### 14.1 定时器简介

定时器是单片机内部集成的功能，它能够通过编程进行灵活控制。单片机的定时功能依赖于内部的计数器实现，每当单片机经历一个机器周期并产生一个脉冲时，计数器就会递增。定时器的主要作用在于计时，当设定的时间到达后，它会触发中断，从而通知系统计时完成。在中断服务函数中，我们可以编写特定的程序以实现所需的功能。例如，若要让 LED 灯每秒钟切换一次状态，我们只需将定时器配置为每秒触发一次中断，并在中断处理程序中编写 LED 状态切换的逻辑。

#### 1，定时器能做什么

定时器的主要作用包括但不限于：

- (1) 执行定时任务：定时器常用于周期性执行特定任务。例如，若需要每 500 毫秒执行某项任务，定时器能够精准地满足这一需求。
- (2) 时间测量：定时器能够精确测量时间，无论是代码段的执行时间还是事件发生的间隔时间，都能通过定时器进行准确的计量。
- (3) 精确延时：对于需要微秒级精度的延时场景，定时器能够提供可靠的解决方案，确保延时的精确性。
- (4) PWM 信号生成：通过定时器的精确控制，我们可以生成 PWM（脉宽调制）信号，这对于驱动电机、调节 LED 亮度等应用至关重要。
- (5) 事件触发与监控：定时器不仅用于触发中断，实现事件驱动的逻辑，还可用于实现看门狗功能，监控系统状态，并在必要时进行复位操作，确保系统的稳定运行。

#### 2，硬件定时器与软件定时器

定时器既可通过硬件实现，也可基于软件进行设计，二者各具特色，适用于不同场景：

硬件定时器，依托微控制器的内置硬件机制，通过专门的计时/计数器电路达成定时功能。其显著优势在于高精度与高可靠性，这是因为硬件定时器的工作独立于软件任务和操作系统调度，故而不受它们的影响。在追求极高定时精确度的场合，如生成 PWM 信号或进行精确时间测量时，硬件定时器无疑是最佳选择。其工作原理确保即便主 CPU 忙于其他任务，定时器也能在预设时间准确触发相应操作。

而软件定时器，则是通过操作系统或软件库模拟实现的定时功能。这类定时器的性能受系统当前负载和任务调度策略制约，因此在精度上较硬件定时器稍逊一筹。然而，软件定时器在灵活性方面更胜一筹，允许创建大量定时器，适用于对时间控制要求不那么严格的场景。

值得注意的是，软件定时器在某些情况下可能面临定时精度问题，特别是在系统负载较重或存在众多高优先级任务时。不过，对于简单的非高精度延时需求，软件定时器通常已经足够应对。

#### 3，ESP32-S3 的定时器整体框架介绍

下面先来简单了解一下系统定时器结构图，通过学习系统定时器结构图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

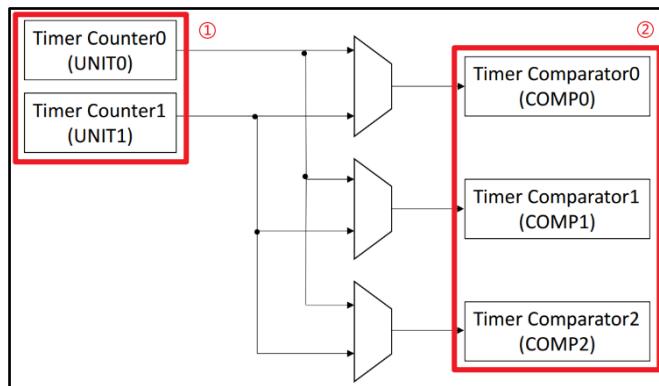


图 14.1.1 系统定时器结构图

系统定时器内置两个计数器 UNIT0 和 UNIT1(如图 14.1.1 中①所示)以及三个比较器 COMP0、COMP1、COMP2(如图 14.1.1 中②所示)。比较器用于监控计数器的计数值是否达到报警值。

### (1) 计数器

UNIT0、UNIT1 均为 ESP32-S3 系统定时器内置的 52 位计数器。计数器使用 XTAL\_CLK 作为时钟源(40MHz)。XTAL\_CLK 经分频后，在一个计数周期生成频率为  $f_{XTAL\_CLK}/3$  的时钟信号，然后在另一个计数周期生成频率为  $f_{XTAL\_CLK}/2$  的时钟信号。因此，计数器使用的时钟 CNT\_CLK，其实际平均频率为  $f_{XTAL\_CLK}/2.5$ ，即 16MHz，见图 14.1.2。每个 CNT\_CLK 时钟周期，计数递增  $1/16\mu s$ ，即 16 个周期递增  $1\mu s$ 。

用户可以通过配置寄存器 SYSTIMER\_CONF\_REG 中下面三个位来控制计数器 UNITn，这三个位分别是：

- ①: SYSTIMER\_TIMER\_UNITn\_WORK\_EN
- ②: SYSTIMER\_TIMER\_UNITn\_CORE0\_STALL\_EN
- ③: SYSTIMER\_TIMER\_UNITn\_CORE1\_STALL\_EN

关于这三位的配置请参考《esp32-s3\_technical\_reference\_manual\_cn》。

### (2) 比较器

COMP0、COMP1、COMP2 均为 ESP32-S3 系统定时器内置的 52 位比较器。比较器同样使用 XTAL\_CLK 作为时钟源(40MHz)。

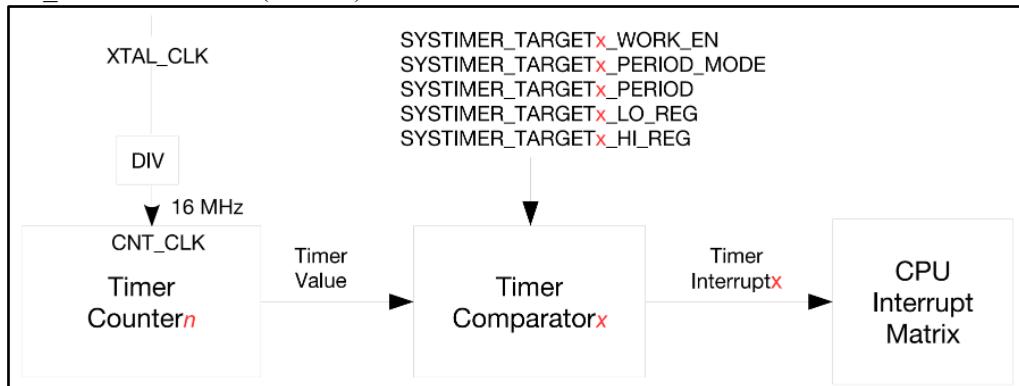


图 14.1.2 系统定时器生成报警

图 14.1.2 展示了系统定时器生成报警的过程。在上述过程中用到一个计数器(Timer Counter $n$ )和一个比较器(Timer Comparator $x$ )，比较器将根据比较结果，生成报警中断。

## 14.2 硬件设计

### 14.2.1 例程功能

实验现象：程序运行后配置高分辨率定时器，并开启中断，在中断回调函数中翻转 LED 的状态。

### 14.2.2 硬件资源

1. LED  
LED - IO1
2. 高分辨率定时器(ESP 定时器)

### 14.2.3 原理图

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

## 14.3 程序设计

### 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：

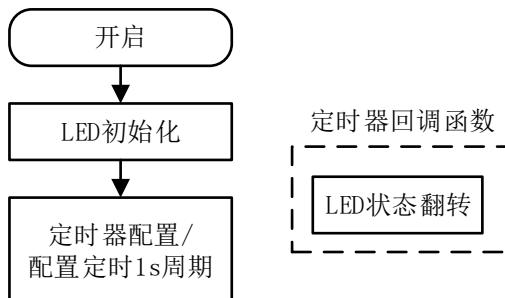


图 14.3.1.1 ESPTIMER 实验程序流程图

### 14.3.2 ESPTIMER 函数解析

ESP-IDF 提供了一套 API 来配置高精度定时器。要使用此功能，需要导入必要的头文件：

```
#include "esp_timer.h"
```

接下来，作者将介绍一些常用的 ESPTIMER 函数，这些函数的描述及其作用如下：

#### 1，创建一个事件

该函数用于创建 ESPTIMER 实例，其函数原型如下所示：

```
esp_err_t esp_timer_create(const esp_timer_create_args_t* create_args,  
                           esp_timer_handle_t* out_handle);
```

该函数的形参描述如下表所示：

形参	描述
arg	指向 arg 外设结构体的指针
esp_tim_handle	指定使能的中断

表 14.3.2.1 函数 esp\_timer\_create()形参描述

返回值：ESP\_OK 表示创建成功。其他表示创建失败。

下面是 esp\_timer\_create\_args\_t 结构体的成员变量描述。

结构体	参数 1	可选值
esp_timer_create_args_t	<b>callback:</b> 定时器在周期内调用的函数。	无，函数需由开发者自行编写
	<b>void* arg:</b> 一个指针类型，将参数传递给回调函数。	一般设置为：NULL，即不携带参数

	<b>dispatch_method:</b> 从 task 或 ISR 调用回调。	无
	<b>const char* name:</b> 定时器名称, 用于 ESPTIMER 转储功能。	无
	<b>skipUnhandledEvents:</b> 跳过周期计时器的未处理事件。	无

表 14.3.2.2 结构体 tim\_periodic\_arg 描述

**2, 每个周期内触发一次**

该函数用于使能定时器的指定中断, 其函数原型如下所示:

```
esp_err_t IRAM_ATTR esp_timer_start_periodic(esp_timer_handle_t timer,
                                              uint64_t period_us);
```

该函数的形参描述, 如下表所示:

形参	描述
timer	使用 esp_timer_create 创建的定时器句柄
period_us	计时器周期, 以微秒为单位

表 14.3.2.3 函数 esp\_timer\_start\_periodict()形参描述

返回值: ESP\_OK 表示开启定时器成功。其他表示开启失败。

**14.3.3 ESPTIMER 驱动解析**

在 IDF 版的 05\_esp\_timer 例程中, 作者在 05\_esp\_timer \components\BSP 路径下新增了一个 ESPTIM 文件夹, 用于存放 esptim.c 和 esptim.h 这两个文件。其中, esptim.h 文件负责声明 ESPTIMER 相关的函数和变量, 而 esptim.c 文件则实现了 ESPTIMER 的驱动代码。下面, 我们将详细解析这两个文件的实现内容。

**1, esptim.h 文件**

```
/* 函数声明 */
void esptim_int_init(uint64_t tps); /* 初始化初始化高分辨率定时器 */
void esptim_callback(void *arg); /* 定时器回调函数 */
```

**2, esptim.c 文件**

```
/**
 * @brief      初始化高精度定时器 (ESP_TIMER)
 * @param      tps: 定时器周期, 以微妙为单位 (μs), 以一秒为定时器周期来执行一次定时器中断, 那此处 tps = 1s = 1000000μs
 * @retval     无
 */

void esptim_int_init(uint64_t tps)
{
    esp_timer_handle_t esp_tim_handle; /* 定时器回调函数句柄 */

    /* 定义一个定时器结构体 */
    esp_timer_create_args_t tim_periodic_arg = {
        .callback = &esptim_callback, /* 设置回调函数 */
        .arg = NULL, /* 不携带参数 */
    };

    esp_timer_create(&tim_periodic_arg, &esp_tim_handle); /* 创建一个事件 */
    esp_timer_start_periodic(esp_tim_handle, tps); /* 每周期内触发一次 */
}

/**
 * @brief      定时器回调函数
 */
```

```

* @param      arg: 不携带参数
* @retval     无
*/
void esptim_callback(void *arg)
{
    LED_TOGGLE();
}

```

从 ESPTIMER 的初始化代码中可以看到，结构体 esp\_timer\_create\_args\_t 通过其中两个结构体成员，以指针的形式调用定时器回调函数。传入的参数 tim\_periodic\_arg，目的在于方便后续的调用，而 esp\_timer\_create() 函数便是通过指针的方式完成对该结构体的调用，之后再通过 esp\_timer\_start\_periodic() 函数设定周期，最终完成 ESPTIMER 的初始化配置。

在定时器回调函数中，我们调用了 LED 状态翻转函数，并设定，定时器的每一次计数溢出都会翻转一次 LED 的状态。关于 ESPTIMER 结构体的介绍，请读者回顾本章节的 14.3.2 小节内容。

#### 14.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    ESPTIM
    LED)

set(include_dirs
    ESPTIM
    LED)

set(requires
    driver
    esp_timer)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRE ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 ESPTIM 驱动需要由开发者自行添加，以确保 ESPTIM 驱动能够顺利集成到构建系统中，需要注意的是我们的依赖库（requires）需要添加上 ESP32-S3 定时器的库。这一步骤是必不可少的，它确保了 ESPTIM 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 14.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/***
* @brief      程序入口
* @param      无
* @retval     无
*/
void app_main(void)
{
    esp_err_t ret;
    uint8_t len = 0;
    uint16_t times = 0;
    unsigned char data[RX_BUF_SIZE] = {0};

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
}

```

}

```
led_init();          /* 初始化 LED */  
esptim_int_init(1000000); /* 初始化高分辨率定时器，此处设置定时器周期为 1 秒，  
                           但该函数以微秒为单位进行计算，  
                           故而 1 秒钟换算为 1000000 微秒 */  
}
```

从上面的代码中可以看到，ESP 定时器的周期值配置为 1000000，因为 ESP32-S3 高分辨率定时器的计数周期是以微秒作为基础单位进行运算，所以当我们设定计数周期为 1 秒时需要将单位换算为微秒。因此 LED 的闪烁周期为 1 秒。

## 14.4 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED 在闪烁，LED 在 ESP 定时器的中断回调函数中被改变状态，LED 每间隔 1000 毫秒改变一次状态。

## 第十五章 SW\_PWM 实验

本章将介绍使用 ESP32-S3 LED 控制器(LEDC)。LEDC 主要用于控制 LED，也可产生 PWM 信号用于其他设备的控制。该控制器有 8 路通道，可以产生独立的波形，驱动 RGB LED 等设备。LED PWM 控制器可在无需 CPU 干预的情况下自动改变占空比，实现亮度渐变。ESP32-S3 IDF 提供了两种方式改变 PWM，一种是通过软件改变 PWM 占空比，另一种是通过硬件改变 PWM 占空比，这两种方式我们都会一一进行讲解。通过本章的学习，开发者将学习到软件改变 PWM 占空比的运用。本章分为如下几个小节：

15.1 PWM 简介

15.2 硬件设计

15.3 程序设计

15.4 下载验证

### 15.1 PWM 简介

#### 1. PWM 原理解析

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

PWM 的主要参数如下：

①：PWM 频率。PWM 频率是 PWM 信号在 1s 内从高电平到低电平再回到高电平的次数，也就是说 1s 内有多少个 PWM 周期，单位为 Hz。

②：PWM 周期。PWM 周期是 PWM 频率的倒数，即  $T=1/f$ ，T 是 PWM 周期，f 是 PWM 频率。如果 PWM 频率为 50Hz，也就是说 PWM 周期为 20ms，即 1s 由 50 个 PWM 周期。

③：PWM 占空比。PWM 占空比是指在一个 PWM 周期内，高电平的时间与整个周期时间的比例，取值范围为 0%~100%。PWM 占空比如下图所示。

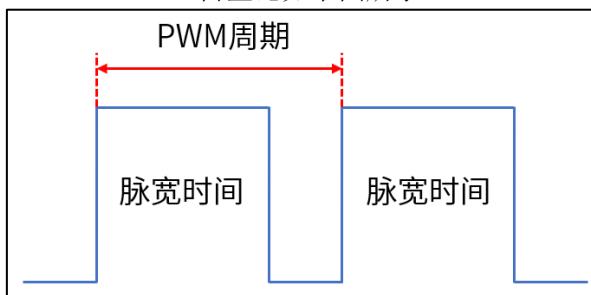


图 15.1.1 PWM 占空比

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

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

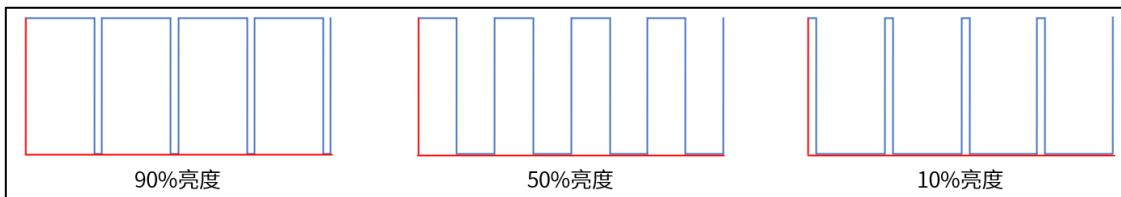


图 15.1.2 PWM 占空比和亮度的关系

## 2. ESP32 的 LED PWM 控制器介绍

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

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

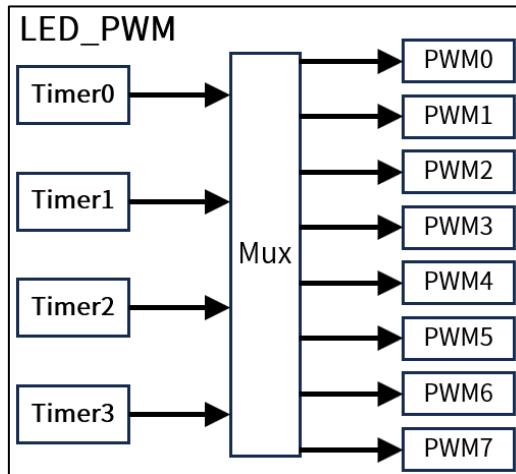


图 15.1.1.1 LED\_PWM 的定时器

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

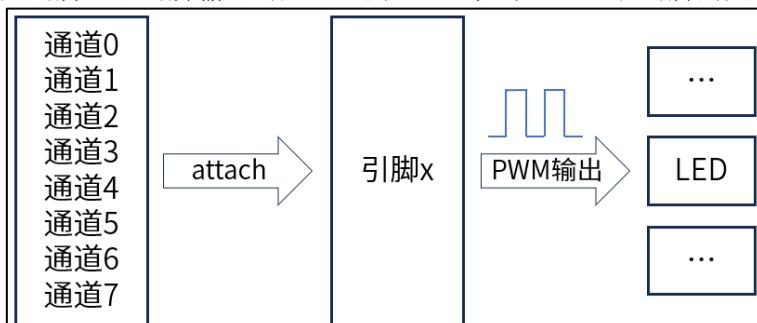


图 15.1.1.2 LED PWM 输出示意图

另外，LED PWM 控制器可在没有 CPU 干预的情况下自动改变占空比，实现亮度以及颜色渐变。

## 15.2 硬件设计

### 15.2.1 例程功能

- 通过软件改变 PWM 的形式使得 LED 由亮到暗，再由暗到亮，依次循环。

### 15.2.2 硬件资源

- LED  
LED-IO1
- 定时器 1  
通道 1 - IO1

### 15.2.3 原理图

本章实验使用的定时器 1 为 ESP32-S3 的片上资源，因此没有对应的连接原理图。

## 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：

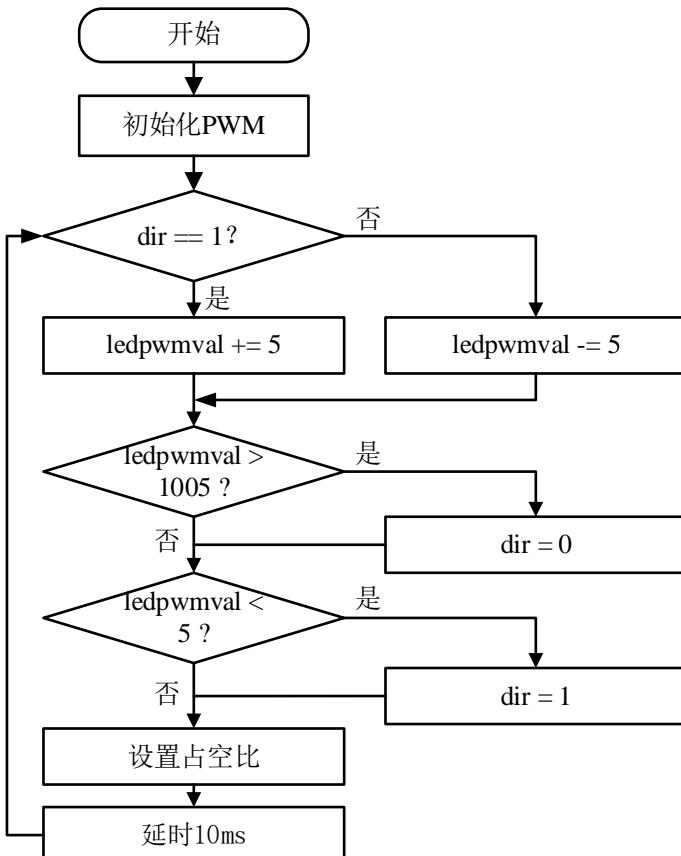


图 15.3.1.1 SW\_PWM 实验程序流程图

### 15.3.2 SW\_PWM 函数解析

ESP-IDF 提供了一套 API 来配置 PWM。要使用此功能，需要导入必要的头文件：

```
#include "driver/ledc.h"
```

接下来，作者将介绍一些常用的 SW\_PWM 函数，这些函数的描述及其作用如下：

#### 1. 配置 LEDC 使用的定时器为定时器 1

需要注意的一点是，在首次配置 LEDC 时，建议先配置定时器(调用函数 `ledc_timer_config()`)，再配置通道(调用函数 `ledc_channel_config()`)。这样可以确保 IO 引脚上的 PWM 信号自输出开始那一刻起，其频率就是正确的。

方才提到，要设置定时器，可调用函数 `ledc_timer_config()`，需要将一些参数的数据结构传递给该函数，该函数原型如下所示：

```
esp_err_t ledc_timer_config(const ledc_timer_config_t *timer_conf);
```

该函数的形参描述，如下表所示：

形参	描述
timer_conf	指向配置 LEDC 定时器的结构体指针

表 15.3.2.1 函数 ledc\_timer\_config() 形参描述

返回值：ESP\_OK 表示配置成功，其他配置失败。

该函数使用 `ledc_timer_config_t` 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
<b>ledc_timer_config_t</b>	<b>speed_mode :</b> 速度模式。	<b>LEDC_LOW_SPEED_MODE:</b> 低速模式 <b>LEDC_HIGH_SPEED_MODE:</b> 高速模式 需要注意的是，ESP32-S3 仅支持低速模式
	<b>timer_num :</b> 通道的定时器源，定时器索引 ledc_timer_t	LEDC_TIMER_0 LEDC_TIMER_1 LEDC_TIMER_2 LEDC_TIMER_3 LEDC_TIMER_MAX
	<b>freq_hz :</b> PWM 信号频率，表示 LEDC 模块的定时器时钟频率设置，单位为 Hz	无
	<b>duty_resolution :</b> PWM 占空比分辨率。占空比分分辨率通常用 ledc_timer_bit_t 设置，范围是 10 至 15 位。如需较低的占空比分分辨率（上至 10，下至 1），可直接输入相应数值。相关参数请参考 ledc_timer_bit_t。	无
	<b>clk_cfg:</b> LEDPWM 的时钟来源	<b>LEDC_AUTO_CLK:</b> 启动定时器时，将根据给定的分辨率和占空率参数自动选择 ledc 源时钟； <b>LEDC_USE_APB_CLK:</b> 选择 APB 作为源时钟； <b>LEDC_USE_RC_FAST_CLK:</b> 选择 “RC_FAST” 作为源时钟； <b>LEDC_USE_XTAL_CLK:</b> 选择 XTAL 作为源时钟； <b>LEDC_USE_RTC8M_CLK:</b> ”LEDC_USE_RC_FAST_CLK”的别名

表 15.3.2.2 ledc\_timer\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 ledc\_timer\_config() 函数，用以实例化 SW\_PWM 并返回 SW\_PWM 句柄。另外值得一提的是，时钟源同样可以限制 PWM 频率。选择的时钟源频率越高，可以配置的 PWM 频率上限就越高。

## 2. 通道配置函数

该函数原型如下所示：

```
esp_err_t ledc_channel_config(const ledc_channel_config_t *ledc_conf);
```

该函数的形参描述，如下表所示：

形参	描述
ledc_conf	指向配置 LEDC 通道的结构体指针

表 15.3.2.3 函数 ledc\_channel\_config() 形参描述

返回值：ESP\_OK 表示配置成功，其他配置失败。

该函数使用 ledc\_channel\_config\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
<b>ledc_channel_config_t</b>	<b>gpio_num :</b> 配置输出引脚	本例程使用的引脚是 GPIO_NUM_1
<b>speed_mode :</b> 速度模式。	<b>LEDC_LOW_SPEED_MODE:</b> 低速模式	
	<b>LEDC_HIGH_SPEED_MODE:</b> 高速模式	
	<b>channel :</b> LEDC 的输出通道 (PWM 的输出通道)。	0~7。本例程配置为通道 1
	<b>intr_type:</b> 配置中断	使能中断: LEDC_INTR_FADE_END 失能中断: LEDC_INTR_DISABLE
	<b>timer_sel:</b> 选择通道的定时器源。 定时器索引 ledc_timer_t	LEDC_TIMER_0 LEDC_TIMER_1 LEDC_TIMER_2 LEDC_TIMER_3 LEDC_TIMER_MAX
	<b>duty :</b> LEDC 通道的占空比设置	占空比设定范围为 0~ $2^{\text{duty_resolution}}$
	<b>hpoint :</b> led 通道 hpoint 值。表示占空比对应的时钟计数值	无
	<b>output_invert:</b> 启用(1)或禁用(0)gpio 输出反相	无

表 15.3.2.4 ledc\_channel\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 ledc\_channel\_config() 函数，用以实例化 PWM 通道。

### 3, 改变 PWM 占空比

调用函数 ledc\_set\_duty() 可以设置新的占空比。之后，调用函数 ledc\_update\_duty() 使新配置生效。要查看当前设置的占空比，可使用 get 函数 ledc\_get\_duty()，该函数原型如下所示：

```
esp_err_t ledc_set_duty(ledc_mode_t speed_mode,
                        ledc_channel_t channel,
                        uint32_t duty);
```

该函数的形参描述，如下表所示：

形参	描述
speed_mode	速度模式选择： LEDC_HIGH_SPEED_MODE LEDC_LOW_SPEED_MODE
channel	LEDC 通道： (0 - LEDC_CHANNEL_MAX-1)，从 ledc_channel_t 中选择
duty	设置 led 的负载，负载设置范围为：0~ $[2^{\text{duty_resolution}}] - 1]$

表 15.3.2.5 函数 ledc\_set\_dut() 形参描述

返回值：ESP\_OK 表示配置成功，其他配置失败。

### 4, 改变 PWM 占空比

在上一步调用 ledc\_set\_duty() 设置新的占空比后，调用函数 ledc\_update\_duty() 使新配置生效，

该函数原型如下所示：

```
esp_err_t ledc_update_duty(ledc_mode_t speed_mode, ledc_channel_t channel);
```

该函数的形参描述，如下表所示：

形参	描述
speed_mode	速度模式选择： LEDC_HIGH_SPEED_MODE LEDC_LOW_SPEED_MODE
channel	LEDC 通道： (0 - LEDC_CHANNEL_MAX-1), 从 ledc_channel_t 中选择

表 15.3.2.6 函数 ledc\_update\_duty() 形参描述

返回值：ESP\_OK 表示配置成功，其他配置失败。

### 15.3.3 SW\_PWM 驱动解析

在 IDF 版的 06-1\_sw\_pwm 例程中，作者在 06-1\_sw\_pwm\components\BSP 路径下新增了一个 PWM 文件夹，用于存放 pwm.c 和 pwm.h 这两个文件。其中，pwm.h 文件负责声明 SW\_PWM 相关的函数和变量，而 pwm.c 文件则实现了 SW\_PWM 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

#### 1, pwm.h 文件

```
/* 引脚以及重要参数定义 */
#define LEDC_PWM_TIMER           LEDC_TIMER_1      /* 使用定时器 1 */
#define LEDC_PWM_CH0_GPIO         GPIO_NUM_1       /* LED 控制器通道对应 GPIO */
#define LEDC_PWM_CH0_CHANNEL     LEDC_CHANNEL_1    /* LED 控制器通道号 */
```

#### 2, pwm.c 文件

```
/**
 * @brief      初始化 PWM
 * @param      resolution:  PWM 占空比分率
 * @param      freq:      PWM 信号频率
 * @retval     无
 */
void pwm_init(uint8_t resolution, uint16_t freq)
{
    ledc_timer_config_t ledc_timer;                      /* LEDC 定时器句柄 */
    ledc_channel_config_t ledc_channel;                  /* LEDC 通道配置句柄 */

    /* 配置 LEDC 定时器 */
    ledc_timer.duty_resolution = resolution;             /* PWM 占空比分率 */
    ledc_timer.freq_hz = freq;                           /* PWM 信号频率 */
    ledc_timer.speed_mode = LEDC_LOW_SPEED_MODE;        /* 定时器模式 */
    ledc_timer.timer_num = LEDC_PWM_TIMER;              /* 定时器序号 */
    ledc_timer.clk_cfg = LEDC_AUTO_CLK;                 /* LEDC 时钟源 */
    ledc_timer_config(&ledc_timer);                     /* 配置定时器 */

    /* 配置 LEDC 通道 */
    ledc_channel gpio_num = LEDC_PWM_CH0_GPIO;          /* LED 控制器通道对应引脚 */
    ledc_channel.speed_mode = LEDC_LOW_SPEED_MODE;       /* LEDC 高速模式 */
    ledc_channel.channel = LEDC_PWM_CH0_CHANNEL;        /* LEDC 控制器通道号 */
    ledc_channel.intr_type = LEDC_INTR_DISABLE;          /* LEDC 失能中断 */
    ledc_channel.timer_sel = LEDC_PWM_TIMER;             /* 定时器序号 */
    ledc_channel.duty = 0;                               /* 占空比值 */
    ledc_channel_config(&ledc_channel);                /* 配置 LEDC 通道 */

}

/**
 * @brief      PWM 占空比设置
 * @param      duty:  PWM 占空比
 * @retval     无
 */
```

```

/*
void pwm_set_duty(uint16_t duty)
{
    ledc_set_duty(LEDC_LOW_SPEED_MODE,
                  LEDC_PWM_CH0_CHANNEL,
                  duty); /* 设置占空比 */
    ledc_update_duty(LEDC_LOW_SPEED_MODE,
                      LEDC_PWM_CH0_CHANNEL); /* 更新占空比 */
}

```

LEDC 定时器设置好后，请配置所需的通道（ledc\_channel\_t）。配置通道需调用函数 ledc\_channel\_config()。通道的配置与定时器设置类似，需向通道配置函数传递相应的结构体 ledc\_channel\_config\_t。此时，通道会按照 ledc\_channel\_config\_t 的配置开始运作，并在选定的 GPIO 上生成由定时器指定的频率和占空比的 PWM 信号。

调用函数 ledc\_set\_duty()可以设置新的占空比。之后，调用函数 ledc\_update\_duty()使新配置生效。为了方便使用，笔者将这两个函数进行了“封装”，通过传参的形式来配置 PWM 占空比。

关于配置过程中所涉及到的结构体成员变量的含义以及用法，请读者们回顾本章节前面的内容。

### 15.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    PWM)

set(include_dirs
    PWM)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 SW\_PWM 驱动需要由开发者自行添加，以确保 SW\_PWM 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 SW\_PWM 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

### 15.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

int main(void)
{
    esp_err_t ret;
    uint8_t dir = 1;
    uint16_t ledrpwmval = 0;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    pwm_init(10, 1000); /* 初始化 PWM */

    while (1)
    {
        vTaskDelay(10);
    }
}

```

```
if (dir == 1)
{
    /* dir==1,ledrpwmval 递增 */
    ledrpwmval += 5;
}
else
{
    /* dir==0,ledrpwmval 递减 */
    ledrpwmval -= 5;
}

if (ledrpwmval > 1005)
{
    /* ledrpwmval 到达 1005 后, 方向改为递减 */
    dir = 0;
}

if (ledrpwmval < 5)
{
    /* ledrpwmval 递减到 5 后, 方向改为递增 */
    dir = 1;
}
/* 设置占空比 */
pwm_set_duty(ledpwmval);
}
```

从上面的代码中可以看到，在初始化 LEDC 定时器，并输出 PWM 后，就不断地改变定时器 0 的值，以达到改变 PWM 占功比的目的。又因为 PWM 由 IO1 引脚输出，IO1 引脚连接至 LED，所以 LED 的亮度也会随之发生变化，从而实现呼吸灯的效果。

#### 15.4 下载验证

在完成编译和烧录后，可以看到板子上的 LED 先由暗再逐渐变亮，以此循环，实现了呼吸灯的效果。

## 第十六章 HW\_PWM 实验

本章将介绍使用 ESP32-S3 LED 控制器（LEDC）。上一章节我们介绍了通过软件改变 PWM 占空比。那么这一章节我们将和开发者们一起来学习硬件改变 PWM 占空比的运用。

本章分为如下几个小节：

- 16.1 PWM 简介
- 16.2 硬件设计
- 16.3 程序设计
- 16.4 下载验证

### 16.1 PWM 简介

关于 PWM 的一些知识，我们在第十五章已经介绍过了，在此便不做赘述。使用硬件的方式改变 PWM 占空比与使用软件的方式改变 PWM 占空比的不同之处在于，LED PWM 控制器硬件可逐渐改变占空比的数值，要使用此功能，需用函数 `ledc_fade_func_install()` 使能渐变，之后用下列可用渐变函数之一配置：

- `ledc_set_fade_with_time()`
- `ledc_set_fade_with_step()`
- `ledc_set_fade()`

最后需要调用 `ledc_fade_start()` 开启渐变。

### 16.2 硬件设计

#### 16.2.1 例程功能

1. 通过硬件改变 PWM 的形式使得 LED 由亮到暗，再由暗到亮，依次循环。

#### 16.2.2 硬件资源

1. LED  
    LED - IO1
2. 定时器 1  
    通道 1 - IO1

#### 16.2.3 原理图

本章实验使用的定时器 1 为 ESP32-S3 的片上资源，因此没有对应的连接原理图。

### 16.3 程序设计

#### 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：

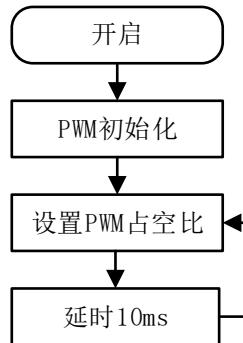


图 16.3.1.1 HW\_PWM 实验程序流程图

### 16.3.2 HW\_PWM 函数解析

ESP-IDF 提供了一套 API 来配置 PWM。要使用此功能，需要导入必要的头文件：

```
#include "driver/ledc.h"
```

接下来，作者将介绍一些常用的 HW\_PWM 函数，这些函数的描述及其作用如下：

#### 1, 使能渐变

LED PWM 控制器硬件可逐渐改变占空比的数值。开启此功能，需要用函数 ledc\_fade\_func\_install() 使能渐变，该函数原型如下所示：

```
esp_err_t ledc_fade_func_install(int intr_alloc_flags);
```

该函数的形参描述，如下表所示：

形参	描述
intr_alloc_flags	用于分配中断的标志

表 16.3.2.1 函数 ledc\_fade\_func\_install() 形参描述

返回值：ESP\_OK 表示配置成功，其他表示配置失败。

#### 2, 设置 LEDC 渐变功能

经过上一步渐变功能的配置后，需要设置占空比以及渐变时长，该函数原型如下所示：

```
esp_err_t ledc_set_fade_with_time(ledc_mode_t speed_mode, ledc_channel_t channel,
                                    uint32_t target_duty, int max_fade_time_ms);
```

该函数的形参描述，如下表所示：

形参	描述
speed_mode	速度模式选择： LEDC_HIGH_SPEED_MODE LEDC_LOW_SPEED_MODE
channel	LEDC 通道： (0 - LEDC_CHANNEL_MAX-1)，从 ledc_channel_t 中选择
target_duty	目标占空比
max_fade_time_ms	最大渐变时间

表 16.3.2.2 函数 ledc\_set\_fade\_with\_time() 形参描述

返回值：ESP\_OK 表示配置成功，其他表示配置失败。

#### 3, 开启渐变

设置占空比以及渐变时长后，便可开启渐变功能，该函数原型如下所示：

```
esp_err_t ledc_fade_start(ledc_mode_t speed_mode,
                           ledc_channel_t channel,
                           ledc_fade_mode_t fade_mode);
```

该函数的形参描述，如下表所示：

形参	描述
speed_mode	速度模式选择： LEDC_HIGH_SPEED_MODE LEDC_LOW_SPEED_MODE

channel	LEDC 通道: (0 - LEDC_CHANNEL_MAX-1), 从 ledc_channel_t 中选择
fade_mode	渐变模式, ledc_fade_mode_t 为索引, 有几个模式可选: LEDC_FADE_NO_WAIT LEDC_FADE_WAIT_DONE LEDC_FADE_MAX

表 16.3.2.3 函数 ledc\_fade\_start() 形参描述

返回值: ESP\_OK 表示配置成功, 其他表示配置失败。

### 16.3.3 HW\_PWM 驱动解析

在 IDF 版的 06-2\_hw\_pwm 例程中, 作者在 06-2\_hw\_pwm\components\BSP 路径下新增了一个 PWM 文件夹, 用于存放 pwm.c 和 pwm.h 这两个文件。其中, pwm.h 文件负责声明 HW\_PWM 相关的函数和变量, 而 pwm.c 文件则实现了 HW\_PWM 的驱动代码。下面, 我们将详细解析这两个文件的实现内容。

#### 1, pwm.h 文件

```
/* 引脚以及重要参数定义 */
#define LEDC_PWM_TIMER          LEDC_TIMER_0      /* 使用定时器 0 */
#define LEDC_PWM_MODE            LEDC_LOW_SPEED_MODE /* 模式设定必须使用 LEDC 低速模式 */
#define LEDC_PWM_CH0_GPIO        GPIO_NUM_1       /* LED 控制器通道对应 GPIO */
#define LEDC_PWM_CH0_CHANNEL    LEDC_CHANNEL_0     /* LED 控制器通道号 */
#define LEDC_PWM_DUTY            8000              /* 渐变的变大最终目标占空比 */
#define LEDC_PWM_FADE_TIME      3000              /* 变化时长 */

/* 函数声明 */
void pwm_init(uint8_t resolution, uint16_t freq); /* 初始化 PWM */
void pwm_set_duty(uint16_t duty);                  /* PWM 占空比设置 */
```

#### 2, pwm.c 文件

```
/**
 * @brief      初始化 PWM
 * @param      resolution: PWM 占空比分辨率
 *             freq: PWM 信号频率
 * @retval     无
 */
void pwm_init(uint8_t resolution, uint16_t freq)
{
    ledc_timer_config_t ledc_timer;           /* LEDC 定时器句柄 */
    ledc_channel_config_t ledc_channel;      /* LEDC 通道配置句柄 */

    /* 配置 LEDC 定时器 */
    ledc_timer.duty_resolution = resolution; /* PWM 占空比分辨率 */
    ledc_timer.freq_hz = freq;                /* PWM 信号频率 */
    ledc_timer.speed_mode = LEDC_PWM_MODE;   /* 定时器模式 */
    ledc_timer.timer_num = LEDC_PWM_TIMER;   /* 定时器序号 */
    ledc_timer.clk_cfg = LEDC_AUTO_CLK;      /* LEDC 时钟源 */
    ledc_timer_config(&ledc_timer);          /* 配置定时器 */

    /* 配置 LEDC 通道 */
    ledc_channel gpio_num = LEDC_PWM_CH0_GPIO; /* LED 控制器通道对应引脚 */
    ledc_channel.speed_mode = LEDC_PWM_MODE;   /* LEDC 高速模式 */
    ledc_channel.channel = LEDC_PWM_CH0_CHANNEL; /* LEDC 控制器通道号 */
    ledc_channel.intr_type = LEDC_INTR_DISABLE; /* LEDC 失能中断 */
    ledc_channel.timer_sel = LEDC_PWM_TIMER;   /* 定时器序号 */
    ledc_channel.duty = 0;                     /* 占空比值 */
    ledc_channel_config(&ledc_channel);       /* 配置 LEDC 通道 */

    ledc_fade_func_install(0);                 /* 使能渐变 (该函数不可或缺) */
}
```

```

}

/***
 * @brief      PWM 占空比设置
 * @param      duty: PWM 占空比
 * @retval     无
 */
void pwm_set_duty(uint16_t duty)
{
    ledc_set_fade_with_time(LEDC_PWM_MODE,
                            LEDC_PWM_CH0_CHANNEL,
                            duty,
                            LEDC_PWM_FADE_TIME); /* 设置占空比以及渐变时长 */
    ledc_fade_start(LEDC_PWM_MODE,
                    LEDC_PWM_CH0_CHANNEL,
                    LEDC_FADE_NO_WAIT); /* 开始渐变 */

    ledc_set_fade_with_time(LEDC_PWM_MODE,
                            LEDC_PWM_CH0_CHANNEL,
                            0,
                            LEDC_PWM_FADE_TIME); /* 设置占空比以及渐变时长 */
    ledc_fade_start(LEDC_PWM_MODE,
                    LEDC_PWM_CH0_CHANNEL,
                    LEDC_FADE_NO_WAIT); /* 开始渐变 */
}

```

在 PWM 初始化函数中，我们配置好 LEDC 定时器的频率、占空比、定时器模式以及定时器通道，并在 `pwm_set_duty()` 函数中，调用函数 `ledc_set_fade_with_time()` 用以设置占空比和渐变时长。此时，我们需要再次调用该函数，将占空比配置为 0，最后调用函数 `ledc_fade_start()` 启开渐变。为了方便使用，笔者将这两个函数进行了“封装”，通过传参的形式来配置 PWM 占空比。关于 PWM 初始化函数中涉及到的结构体含义，请读者们回顾第十五章节的内容。

#### 16.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    PWM)

set(include_dirs
    PWM)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIREMENTS ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 `HW_PWM` 驱动需要由开发者自行添加，以确保 `HW_PWM` 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 `HW_PWM` 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 16.3.5 实验应用代码

打开 `main/main.c` 文件，该文件定义了工程入口函数，名为 `app_main`。该函数代码如下。

```

int main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND)

```

```
{  
    ESP_ERROR_CHECK(nvs_flash_erase());  
    ret = nvs_flash_init();  
}  
  
pwm_init(13, 5000); /* 初始化 PWM */  
  
while (1)  
{  
    vTaskDelay(10);  
    pwm_set_duty(LEDC_PWM_DUTY); /* 设置占空比 */  
}  
}
```

在主函数中，我们首先对 PWM 进行初始化，设置了 PWM 占空比分辨率为 13，PWM 信号频率为 5000，最后在 while(1)循环中配置了占空比。通过与上一个实验的对比，硬件改变占空比的方式与软件改变占空比的方式虽有不同，但一样可以实现呼吸灯的效果。开发者们可以这两个章节进行对比学习。

## 16.4 下载验证

在完成编译和烧录后，可以看到板子上的 LED 先由暗再逐渐变亮，以此循环，实现了呼吸灯的效果。

## 第十七章 SPI\_LCD 实验

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

本章分为以下几个小节：

17.1 SPI 与 LCD 简介

17.2 硬件设计

17.3 程序设计

17.4 下载验证

### 17.1 SPI 及 LCD 介绍

#### 17.1.1 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。

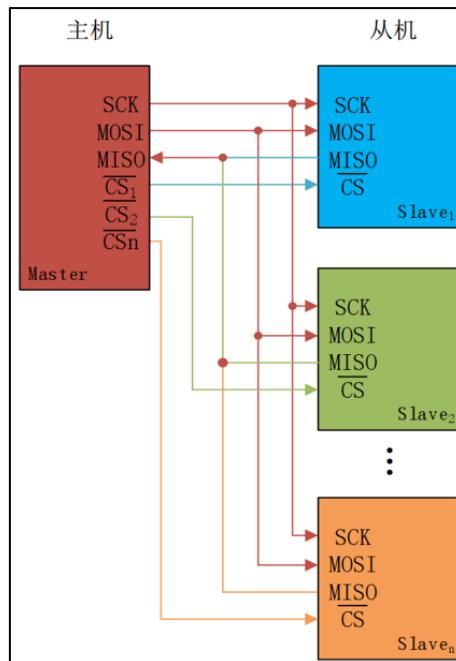


图 17.1.1.1 多从机 SPI 通信网络图

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。

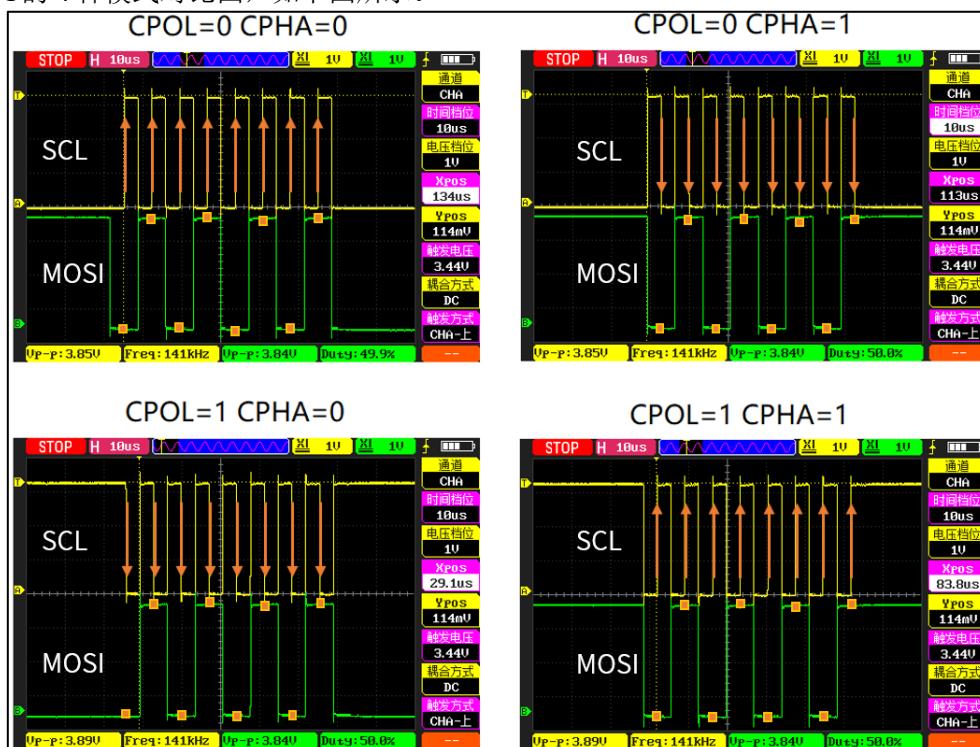


图 17.1.1.2 SPI 的 4 种模式对比图

- 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。

### 17.1.2 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。

### 17.1.3 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 17.1.3.1 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 17.1.3.1 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。

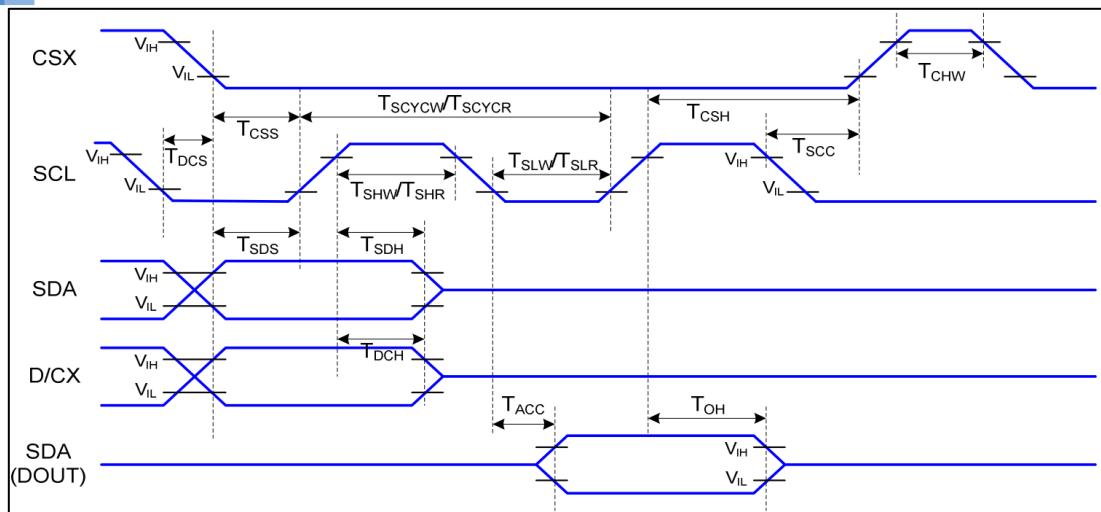


图 17.1.3.2 四线 SPI 接口时序图

上图中各个时间参数，如下表所示。

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

表 17.1.3.2 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。

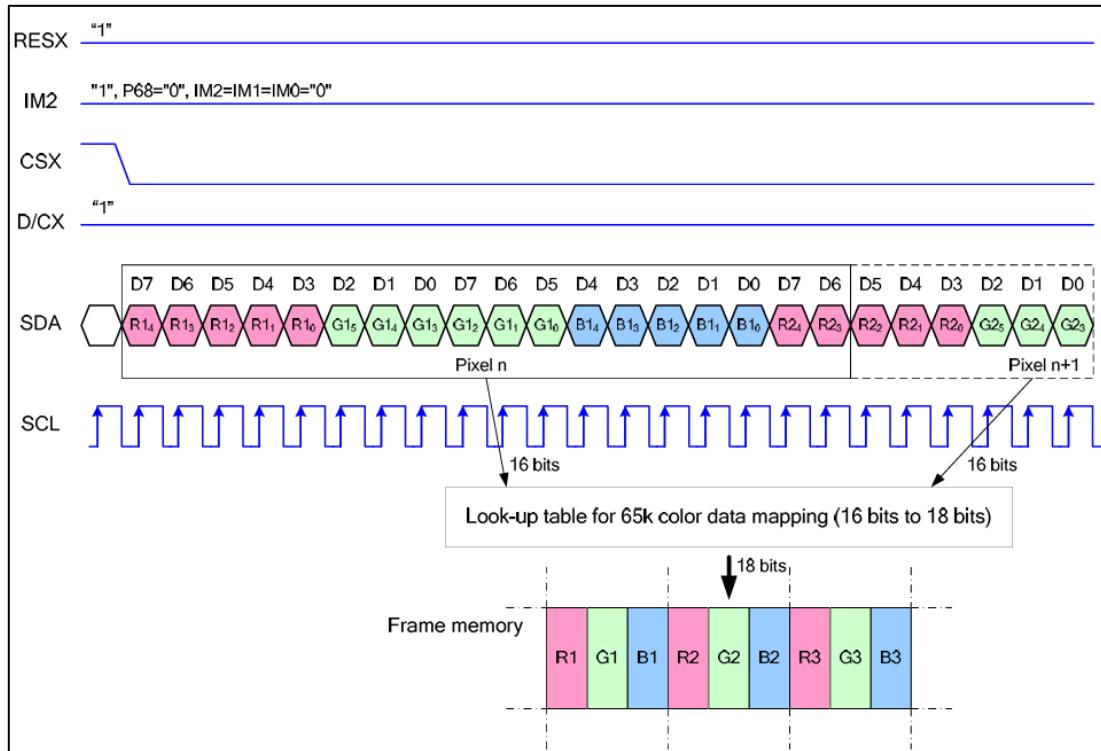


图 17.1.3.3 16 位色深模式 (RGB565) 传输颜色数据

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.																																																																																																	
<table border="1"> <thead> <tr> <th>Bit</th> <th colspan="3">NAME</th> <th colspan="8">DESCRIPTION</th> </tr> </thead> <tbody> <tr> <td>D7</td> <td colspan="3">MY</td> <td colspan="8">Page Address Order</td> </tr> <tr> <td>D6</td> <td colspan="3">MX</td> <td colspan="8">Column Address Order</td> </tr> <tr> <td>D5</td> <td colspan="3">MV</td> <td colspan="8">Page/Column Order</td> </tr> <tr> <td>D4</td> <td colspan="3">ML</td> <td colspan="8">Line Address Order</td> </tr> <tr> <td>D3</td> <td colspan="3">RGB</td> <td colspan="8">RGB/BGR Order</td> </tr> <tr> <td>D2</td> <td colspan="3">MH</td> <td colspan="8">Display Data Latch Order</td> </tr> </tbody> </table>														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							
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 17.1.3.4 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 17.1.3.3 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)													
2AH	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 <sup>st</sup> Parameter	1	↑	1	-	XS15	XS14	XS13	XS12	XS11	XS10	XS9	XS8	
2 <sup>nd</sup> Parameter	1	↑	1	-	XS7	XS6	XS5	XS4	XS3	XS2	XS1	XS0	
3 <sup>rd</sup> Parameter	1	↑	1	-	XE15	XE14	XE13	XE12	XE11	XE10	XE9	XE8	
4 <sup>th</sup> Parameter	1	↑	1	-	XE7	XE6	XE5	XE4	XE3	XE2	XE1	XE0	

图 17.1.3.5 命令 0x2A 描述

命令 0x2B 的描述，如下图所示。

RASET (Row Address Set)													
2BH	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 <sup>st</sup> Parameter	1	↑	1	-	YS15	YS14	YS13	YS12	YS11	YS10	YS9	YS8	
2 <sup>nd</sup> Parameter	1	↑	1	-	YS7	YS6	YS5	YS4	YS3	YS2	YS1	YS0	
3 <sup>rd</sup> Parameter	1	↑	1	-	YE15	YE14	YE13	YE12	YE11	YE10	YE9	YE8	
4 <sup>th</sup> Parameter	1	↑	1	-	YE7	YE6	YE5	YE4	YE3	YE2	YE1	YE0	

图 17.1.3.6 命令 0x2B 描述

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

## 17.2 硬件设计

### 17.2.1 例程功能

本章实验功能简介：按下复位之后，就可以看到 SPI LCD 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。

### 17.2.2 硬件资源

#### 1. LED

LED - IO1

#### 2. 正点原子 0.96 寸 SPI LCD 模块

### 17.2.3 原理图

0.96 寸 LCD 与板载 MCU 的连接原理图，如下图所示：

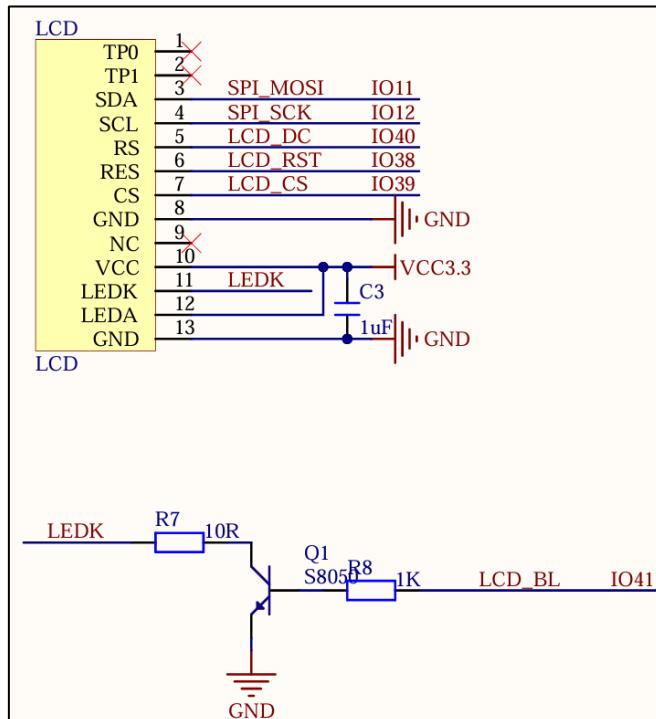


图 17.2.3.1 SPILCD 模块与 MCU 的连接原理图

## 17.3 程序设计

### 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：

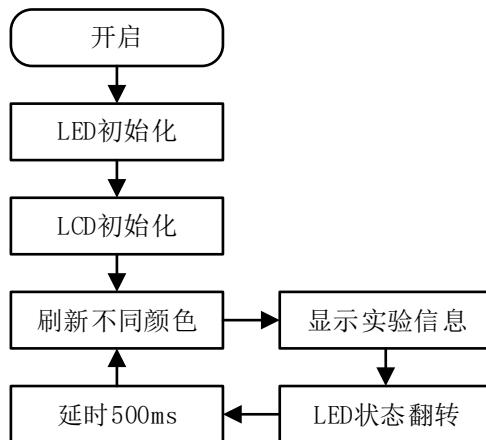


图 17.3.1.1 SPI\_LCD 实验程序流程图

### 17.3.2 SPI\_LCD 函数解析

ESP-IDF 提供了一套 API 来配置 SPI。要使用此功能，需要导入必要的头文件：

```
#include "driver/spi_master.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 SPI 函数，以及 IO 扩展芯片中用到的函数，

这些函数的描述及其作用如下：

### 1. 初始化和配置

该函数用于初始化 SPI 总线，并配置其 GPIO 引脚和主模式下的时钟等参数，该函数原型如下所示：

```
esp_err_t spi_bus_initialize(spi_host_device_t host_id,
                             const spi_bus_config_t *bus_config,
                             spi_dma_chan_t dma_chan);
```

该函数的形参描述如下表所示：

参数	描述
host_id	指定 SPI 总线的主机设备 ID
bus_config	指向 spi_bus_config_t 结构体的指针，用于配置 SPI 总线的 SCLK、MISO、MOSI 等引脚以及其他参数
dma_chan	指定使用哪个 DMA 通道。 有效值为： SPI_DMA_CH_AUTO, SPI_DMA_DISABLED 或 1 至 2 之间的数字

表 17.3.2.1 spi\_bus\_initialize() 函数形参描述

返回值：ESP\_OK 配置成功。其他配置失败。

该函数使用 spi\_bus\_config\_t 类型的结构体变量传入，笔者此处列举了我们需要用到的结构体，该结构体的定义如下所示：

```
typedef struct {
    int miso_io_num;          /* MISO 引脚号 */
    int mosi_io_num;          /* MOSI 引脚号 */
    int sclk_io_num;          /* 时钟引脚号 */
    int quadwp_io_num;        /* 用于 Quad 模式的 WP 引脚号，未使用时设置为-1 */
    int quadhd_io_num;        /* 用于 Quad 模式的 HD 引脚号，未使用时设置为-1 */
    int max_transfer_sz;      /* 最大传输大小 */
    ...
} spi_bus_config_t;
```

完成上述结构体参数配置之后，可以将结构传递给 spi\_bus\_initialize 函数，用以实例化 SPI。

### 2. 设备配置

该函数用于在 SPI 总线上分配设备，函数原型如下所示：

```
esp_err_t spi_bus_add_device(spi_host_device_t host_id,
                             const spi_device_interface_config_t *dev_config,
                             spi_device_handle_t *handle);
```

该函数的形参描述，如下表所示：

形参	描述
host_id	指定 SPI 总线的主机设备 ID
dev_config	指向 spi_device_interface_config_t 结构体的指针，用于配置 SPI 设备的通信参数，如时钟速率、SPI 模式等。
handle	返回创建的设备句柄

表 17.3.2.2 函数 spi\_bus\_add\_device() 形参描述

返回值：ESP\_OK 配置成功。其他配置失败。

该函数使用 spi\_host\_device\_t 类型以及 spi\_device\_interface\_config\_t 类型的结构体变量传入 SPI 外围设备的配置参数，该结构体的定义如下所示：

```
/**
 * @brief 带有三个 SPI 外围设备的枚举，这些外围设备可通过软件访问
 */
typedef enum {
    /* SPI1 只能在 ESP32 上用作 GPSPI */
    SPI1_HOST = 0,      /* SPI1 */
    SPI2_HOST = 1,      /* SPI2 */
#if SOC_SPI_PERIPH_NUM > 2
    SPI3_HOST = 2,      /* SPI3 */
#endif
}
```

```

#endif
    SPI_HOST_MAX, /* 无效的主机值 */
} spi_host_device_t

typedef struct {
    uint32_t command_bits; /* 命令阶段的位数 */
    uint32_t address_bits; /* 地址阶段的位数 */
    uint32_t dummy_bits; /* 虚拟阶段的位数 */
    int clock_speed_hz; /* 时钟速率 */
    uint32_t mode; /* SPI 模式 (0-3) */
    int spics_io_num; /* CS 引脚号 */
    ...
} spi_device_interface_config_t;

```

### 3, 数据传输

根据函数功能，以下函数可以归为一类进行讲解，下面将以表格的形式逐个介绍这些函数的作用与参数。

函数	描述
spi_device_transmit()	该函数用于发送一个 SPI 事务，等待它完成，并返回结果。 <b>handle:</b> 设备的句柄。 <b>trans_desc:</b> 指向 spi_transaction_t 结构体的指针，描述了要发送的事务详情。
spi_device_polling_transmit()	该函数用于发送一个轮询事务，等待它完成，并返回结果。 <b>handle:</b> 设备的句柄。 <b>trans_desc:</b> 指向 spi_transaction_t 结构体的指针，描述了要发送的事务详情。

表 17.3.2.3 SPI 数据传输函数描述

### 17.3.3 SPI\_LCD 驱动解析

在 IDF 版的 08\_spilcd 例程中，作者在 8\_spilcd\components\BSP 路径下新增了一个 SPI 文件夹和一个 LCD 文件夹，分别用于存放 spi.c、spi.h 和 lcd.c 以及 lcd.h 这四个文件。其中，spi.h 和 lcd.h 文件负责声明 SPI 以及 LCD 相关的函数和变量，而 spi.c 和 lcd.c 文件则实现了 SPI 以及 LCD 的驱动代码。下面，我们将详细解析这四个文件的实现内容。

#### 1, spi.h 文件

```

/* 引脚定义 */
#define SPI_MOSI_GPIO_PIN    GPIO_NUM_11      /* SPI2_MOSI */
#define SPI_CLK_GPIO_PIN     GPIO_NUM_12      /* SPI2_CLK */
#define SPI_MISO_GPIO_PIN    GPIO_NUM_13      /* SPI2_MISO */

```

该文件下定义了 SPI 的时钟引脚与通讯引脚。

#### 2, spi.c 文件

```

/**
 * @brief      初始化 SPI
 * @param      无
 * @retval     无
 */
void spi2_init(void)
{
    esp_err_t ret = 0;
    spi_bus_config_t spi_bus_conf = {0};

    /* SPI 总线配置 */
    spi_bus_conf.miso_io_num = SPI_MISO_GPIO_PIN; /* SPI_MISO 引脚 */
    spi_bus_conf.mosi_io_num = SPI_MOSI_GPIO_PIN; /* SPI_MOSI 引脚 */
    spi_bus_conf.sclk_io_num = SPI_CLK_GPIO_PIN; /* SPI_SCLK 引脚 */
}

```

```
spi_bus_conf.quadwp_io_num = -1; /* SPI 写保护信号引脚，该引脚未使能 */
spi_bus_conf.quadhd_io_num = -1; /* SPI 保持信号引脚，该引脚未使能 */
spi_bus_conf.max_transfer_sz = 160 * 80 * 2; /* 配置最大传输大小，以字节为单位 */

/* 初始化 SPI 总线 */
ret = spi_bus_initialize(SPI2_HOST, &spi_bus_conf, SPI_DMA_CH_AUTO);
ESP_ERROR_CHECK(ret); /* 校验参数值 */
}

/***
 * @brief      SPI 发送命令
 * @param      handle : SPI 句柄
 * @param      cmd    : 要发送命令
 * @retval     无
 */
void spi2_write_cmd(spi_device_handle_t handle, uint8_t cmd)
{
    esp_err_t ret;
    spi_transaction_t t = {0};

    t.length = 8; /* 要传输的位数 一个字节 8 位 */
    t.tx_buffer = &cmd; /* 将命令填充进去 */
    ret = spi_device_polling_transmit(handle, &t); /* 开始传输 */
    ESP_ERROR_CHECK(ret); /* 一般不会有问题 */
}

/***
 * @brief      SPI 发送数据
 * @param      handle : SPI 句柄
 * @param      data   : 要发送的数据
 * @param      len    : 要发送的数据长度
 * @retval     无
 */
void spi2_write_data(spi_device_handle_t handle, const uint8_t *data, int len)
{
    esp_err_t ret;
    spi_transaction_t t = {0};

    if (len == 0)
    {
        return; /* 长度为 0 没有数据要传输 */
    }
    t.length = len * 8; /* 要传输的位数 一个字节 8 位 */
    t.tx_buffer = data; /* 将命令填充进去 */
    ret = spi_device_polling_transmit(handle, &t); /* 开始传输 */
    ESP_ERROR_CHECK(ret); /* 一般不会有问题 */
}

/***
 * @brief      SPI 处理数据
 * @param      handle      : SPI 句柄
 * @param      data        : 要发送的数据
 * @param      t.rx_data[0] : 接收到的数据
 */
uint8_t spi2_transfer_byte(spi_device_handle_t handle, uint8_t data)
{
    spi_transaction_t t;

    memset(&t, 0, sizeof(t));

    t.flags = SPI_TRANS_USE_TXDATA | SPI_TRANS_USE_RXDATA;
    t.length = 8;
```

```

        t.tx_data[0] = data;
        spi_device_transmit(handle, &t);

        return t.rx_data[0];
    }
}

```

在 spi2\_init() 函数中主要工作就是对于 SPI 参数的配置，如 SPI 管脚配置和数据传输大小以及 SPI 总线配置等，通过该函数就可以完成 SPI 初始化。

SPI 驱动中对 SPI 的各种操作，请读者结合 SPI 的时序规定查看本实验的配套实验源码。

### 3, lcd.h 文件

lcd.c 和 lcd.h 文件是驱动函数和引脚接口宏定义以及函数声明等。lcdfont.h 头文件存放了 4 种字体大小不一样的 ASCII 字符集 (12\*12、16\*16、24\*24 和 32\*32)。这个跟 oledfont.h 头文件一样的，只是这里多了 32\*32 的 ASCII 字符集，制作方法请回顾 OLED 实验。下面我们还是先介绍 lcd.h 文件，首先是 LCD 的引脚定义：

```

/* 引脚定义 */
#define LCD_NUM_BL      GPIO_NUM_41
#define LCD_NUM_WR      GPIO_NUM_40
#define LCD_NUM_CS      GPIO_NUM_39
#define LCD_NUM_RST     GPIO_NUM_38

/* IO 操作 */
#define LCD_WR(x)        do{ x ? \
                           (gpio_set_level(LCD_NUM_WR, 1)): \
                           (gpio_set_level(LCD_NUM_WR, 0)); \
                     }while(0)

#define LCD_CS(x)        do{ x ? \
                           (gpio_set_level(LCD_NUM_CS, 1)): \
                           (gpio_set_level(LCD_NUM_CS, 0)); \
                     }while(0)

#define LCD_PWR(x)       do{ x ? \
                           (gpio_set_level(LCD_NUM_BL, 1)): \
                           (gpio_set_level(LCD_NUM_BL, 0)); \
                     }while(0)

#define LCD_RST(x)       do{ x ? \
                           (gpio_set_level(LCD_NUM_RST, 1)): \
                           (gpio_set_level(LCD_NUM_RST, 0)); \
                     }while(0)

/* 常用颜色值 */
#define WHITE            0xFFFF /* 白色 */
#define BLACK            0x0000 /* 黑色 */
#define RED              0xF800 /* 红色 */
#define GREEN            0x07E0 /* 绿色 */
#define BLUE             0x001F /* 蓝色 */
#define MAGENTA          0XF81F /* 品红色/紫红色 = BLUE + RED */
#define YELLOW           0XFFE0 /* 黄色 = GREEN + RED */
#define CYAN             0X07FF /* 青色 = GREEN + BLUE */

/* 非常用颜色 */
#define BROWN            0XB4C0 /* 棕色 */
#define BRRED             0XFC07 /* 棕红色 */
#define GRAY              0X8430 /* 灰色 */
#define DARKBLUE          0X01CF /* 深蓝色 */
#define LIGHTBLUE         0X7D7C /* 浅蓝色 */
#define GRAYBLUE          0X5458 /* 灰蓝色 */
#define LIGHTGREEN        0X841F /* 浅绿色 */
#define LGRAY             0XC618 /* 浅灰色(PANEL), 窗体背景色 */
#define LGRAYBLUE         0XA651 /* 浅灰蓝色(中间层颜色) */

```

```

#define LBBLUE          0XB12 /* 浅棕蓝色(选择条目的反色) */

/* 扫描方向定义 */
#define L2R_U2D          0      /* 从左到右,从上到下 */
#define L2R_D2U          1      /* 从左到右,从下到上 */
#define R2L_U2D          2      /* 从右到左,从上到下 */
#define R2L_D2U          3      /* 从右到左,从下到上 */
#define U2D_L2R          4      /* 从上到下,从左到右 */
#define U2D_R2L          5      /* 从上到下,从右到左 */
#define D2U_L2R          6      /* 从下到上,从左到右 */
#define D2U_R2L          7      /* 从下到上,从右到左 */

#define DFT_SCAN_DIR     L2R_U2D /* 默认的扫描方向 */

```

```

/* LCD 信息结构体 */
typedef struct _lcd_obj_t
{
    uint16_t width;           /* 宽度 */
    uint16_t height;          /* 高度 */
    uint8_t dir;              /* 横屏还是竖屏控制: 0, 竖屏; 1, 横屏。 */
    uint16_t wramcmd;         /* 开始写 gram 指令 */
    uint16_t setxcmd;         /* 设置 x 坐标指令 */
    uint16_t setycmd;         /* 设置 y 坐标指令 */
    uint16_t wr;               /* 命令/数据 IO */
    uint16_t cs;               /* 片选 IO */
    uint16_t bl;               /* 背光 */
    uint16_t rst;              /* 复位 */
} lcd_obj_t;

/* LCD 缓存大小设置, 修改此值时请注意!!!! 修改这两个值时可能会影响以下函数
lcd_clear/lcd_fill/lcd_draw_line */
#define LCD_TOTAL_BUF_SIZE      (160 * 80 * 2)
#define LCD_BUF_SIZE             2560

```

```

/* 导出相关变量 */
extern lcd_obj_t lcd_self;
extern uint8_t lcd_buf[LCD_TOTAL_BUF_SIZE];

```

第一部分的宏定义是对 CS、RST、BL 和 WR (DC) 引脚的定义，第二部分宏定义是 WR/CS/PWR/RST 引脚操作的定义，接下来的部分是对一些常用颜色的 RGB 数值以及 LCD 信息结构体的定义。

#### 4, lcd.c 文件

```

/**
 * @brief      LCD 初始化
 * @param      无
 * @retval     无
 */
void lcd_init(void)
{
    int cmd = 0;
    esp_err_t ret = 0;

    lcd_self.dir = 0;                                /* 配置 WR 引脚 */
    lcd_self.wr = LCD_NUM_WR;                         /* 配置 CS 引脚 */
    lcd_self.cs = LCD_NUM_CS;                         /* 配置 BL 引脚 */
    lcd_self.bl = LCD_NUM_BL;                         /* 配置 RST 引脚 */
    lcd_self.rst = LCD_NUM_RST;
}

gpio_config_t gpio_init_struct;

```

```

/* SPI 驱动接口配置 */
spi_device_interface_config_t devcfg = {
    .clock_speed_hz = 60 * 1000 * 1000,           /* SPI 时钟 */
    .mode = 0,                                     /* SPI 模式 0 */
    .spics_io_num = lcd_self.cs,                   /* SPI 设备引脚 */
    .queue_size = 7,
};

/* 添加 SPI 总线设备 */
ret = spi_bus_add_device(SPI2_HOST, &devcfg, &MY_LCD_Handle);
ESP_ERROR_CHECK(ret);

/* WR 管脚 */
gpio_init_struct.intr_type = GPIO_INTR_DISABLE;   /* 失能引脚中断 */
gpio_init_struct.mode = GPIO_MODE_OUTPUT;          /* 配置输出模式 */
gpio_init_struct.pin_bit_mask = 1ull << lcd_self.wr; /* 配置引脚位掩码 */
gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE; /* 失能下拉 */
gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;   /* 使能下拉 */
gpio_config(&gpio_init_struct);                  /* 引脚配置 */

/* BL 管脚 */
gpio_init_struct.intr_type = GPIO_INTR_DISABLE;   /* 失能引脚中断 */
gpio_init_struct.mode = GPIO_MODE_OUTPUT;          /* 配置输出模式 */
gpio_init_struct.pin_bit_mask = 1ull << lcd_self.bl; /* 配置引脚位掩码 */
gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE; /* 失能下拉 */
gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;   /* 使能下拉 */
gpio_config(&gpio_init_struct);                  /* 引脚配置 */

/* RST 管脚 */
gpio_init_struct.intr_type = GPIO_INTR_DISABLE;   /* 失能引脚中断 */
gpio_init_struct.mode = GPIO_MODE_OUTPUT;          /* 配置输出模式 */
gpio_init_struct.pin_bit_mask = 1ull << lcd_self.rst; /* 配置引脚位掩码 */
gpio_init_struct.pull_down_en = GPIO_PULLDOWN_DISABLE; /* 失能下拉 */
gpio_init_struct.pull_up_en = GPIO_PULLUP_ENABLE;   /* 使能下拉 */
gpio_config(&gpio_init_struct);                  /* 引脚配置 */

/* LCD 硬件复位 */
lcd_hard_reset();                                /* LCD 硬件复位 */

/* 0.96 寸 lcd 屏幕初始化序列 */
lcd_init_cmd_t ili_init_cmds[] =
{
    /* 此处忽略 */
};

/* 发送初始化序列 */
while (ili_init_cmds[cmd].databytes != 0xff)
{
    lcd_write_cmd(ili_init_cmds[cmd].cmd);
    lcd_write_data(ili_init_cmds[cmd].data, ili_init_cmds[cmd].bytes&0x1F);

    if (ili_init_cmds[cmd].databytes & 0x80)
    {
        vTaskDelay(120);
    }

    cmd++;
}

lcd_display_dir(1);                             /* 设置屏幕方向 */
LCD_PWR(1);                                    /* 清屏 */
lcd_clear(WHITE);
}

```

从上的代码中可以看出，本章实验的 SPILCD 驱动是兼容了正点原子的 1.3 寸与 2.4 寸 SPILCD 模块的，因此在加载完 SPI 设备后，会与 SPILCD 进行通讯，确定 SPILCD 的型号，然后根据型号针对性地对 SPILCD 模块进行配置。

SPILCD 驱动中与 SPILCD 模块通讯的函数，如下所示：

```
/***
 * @brief      发送命令到 LCD，使用轮询方式阻塞等待传输完成(由于数据传输量很少，因此在轮询方式处理可提高速度。使用中断方式的开销要超过轮询方式)
 * @param      cmd 传输的 8 位命令数据
 * @retval     无
 */
void lcd_write_cmd(const uint8_t cmd)
{
    LCD_WR(0);
    spi2_write_cmd(MY_LCD_Handle, cmd);
}

/***
 * @brief      发送数据到 LCD，使用轮询方式阻塞等待传输完成(由于数据传输量很少，因此在轮询方式处理可提高速度。使用中断方式的开销要超过轮询方式)
 * @param      data 传输的 8 位数据
 * @retval     无
 */
void lcd_write_data(const uint8_t *data, int len)
{
    LCD_WR(1);
    spi2_write_data(MY_LCD_Handle, data, len);
}

/***
 * @brief      发送数据到 LCD，使用轮询方式阻塞等待传输完成(由于数据传输量很少，因此在轮询方式处理可提高速度。使用中断方式的开销要超过轮询方式)
 * @param      data 传输的 16 位数据
 * @retval     无
 */
void lcd_write_data16(uint16_t data)
{
    uint8_t dataBuf[2] = {0, 0};
    dataBuf[0] = data >> 8;
    dataBuf[1] = data & 0xFF;
    LCD_WR(1);
    spi2_write_data(MY_LCD_Handle, dataBuf, 2);
}
```

在上述代码中，`lcd_write_cmd()`和`lcd_write_data()`在调用 SPI 的驱动函数前，按照 LCD 时序图，前者需要先将 WR 引脚电平信号置 0，后者则需要置 1。

通过上面介绍的驱动函数就能够与 SPILCD 模块进行通讯了，而在 SPILCD 模块的显示屏上显示出特定的图案或字符或设置 SPILCD 模块的显示方向等等的操作都是能够通过 SPILCD 模块规定的特定命令来完成的，想深究的读者可以产看正点原子 SPILCD 模块的用户手册或查看实际使用的 SPILCD 模块的相关文档。

#### 17.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```
set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)
```

```

set(requires
      driver)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 LCD 与 SPI 驱动需要由开发者自行添加，以确保 SPI\_LCD 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 SPI\_LCD 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

### 17.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void){
    uint8_t x = 0;
    esp_err_t ret;
    ret = nvs_flash_init(); /* 初始化 NVS */
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    led_init(); /* 初始化 LED */
    spi2_init(); /* 初始化 SPI2 */
    lcd_init(); /* 初始化 LCD */
    while (1){
        switch (x)
        {
            case 0:
            {
                lcd_clear(WHITE);
                break;
            }
            case 1:
            {
                lcd_clear(BLACK);
                break;
            }
            case 2:
            {
                lcd_clear(BLUE);
                break;
            }
            case 3:
            {
                lcd_clear(RED);
                break;
            }
            case 4:
            {
                lcd_clear(MAGENTA);
                break;
            }
            case 5:
            {
                lcd_clear(GREEN);
                break;
            }
        }
    }
}

```

```
        }
    case 6:
    {
        lcd_clear(CYAN);
        break;
    }
    case 7:
    {
        lcd_clear(YELLOW);
        break;
    }
    case 8:
    {
        lcd_clear(BRRED);
        break;
    }
    case 9:
    {
        lcd_clear(GRAY);
        break;
    }
    case 10:
    {
        lcd_clear(LGRAY);
        break;
    }
    case 11:
    {
        lcd_clear(BROWN);
        break;
    }
}
lcd_show_string(0, 0, 240, 32, 32, "ESP32", RED);
lcd_show_string(0, 33, 240, 24, 24, "SPILCD TEST", RED);
lcd_show_string(0, 60, 240, 16, 16, "ATOM@ALIENTEK", RED);
x++;
if (x == 12) {
    x = 0;
}
LED_TOGGLE();
vTaskDelay(500);
}
}
```

从上面的代码中可以看出，在初始化完 LCD 后，便在 LCD 上显示一些本实验的相关信息，随后便每间隔 500 毫秒就更换一次 LCD 屏幕显示的背景色。

## 17.4 下载验证

在完成编译和烧录操作后，可以看到 SPI LCD 上不断变换着不同的颜色，LED 灯闪烁。

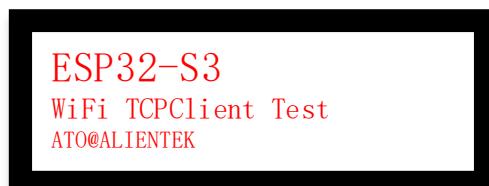


图 17.4.1 SPI LCD 显示效果图

## 第十八章 RTC 实验

本章介绍 ESP32-S3 实时时钟（RTC）的使用，实时时钟能为系统提供一个准确的时间，即时系统复位或主电源断电，RTC 依然能够运行，因此 RTC 也经常用于各种低功耗场景。通过本章的学习，读者将学习到 RTC 的使用。

本章分为如下几个小节：

18.1 RTC 时钟简介

18.2 硬件设计

18.3 程序设计

18.4 下载验证

### 18.1 RTC 时钟简介

RTC（实时时钟）是指安装在电子设备或实现其功能的 IC（集成电路）上的时钟。当您在数字电路中称其为“时钟”时，您可能会想到周期信号，但在英语中，clock 也意味着“时钟”。

那为什么我们需要一个单独的 RTC？

原因是 CPU 的定时器时钟功能只在“启动”即“通电时”运行，断电时停止。当然，如果时钟不能连续跟踪时间，则必须手动设置时间。

通常，RTC 配备一个单独分离的电源，如纽扣电池（备用电池），即使 DNESP32S3M 最小系统板电源关闭，它也能保持运作，随时可以实时显示时间。然后，当 DNESP32S3M 最小系统板再次打开时，计算机内置的定时器时钟从 RTC 读取当前时间，并在此基础上供电的同时，时间在其自身机制下显示。顺便说一句，由于纽扣电池相对便宜且使用寿命长，因此 RTC 可以以极低的成本运行。基于此这个作用，它也可以用作内存。

#### 1. ESP32-S3 的 RTC

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

ESP32-S3 使用两种硬件时钟源建立和保持系统时间。根据应用目的及对系统时间的精度要求，既可以仅使用其中一种时钟源，也可以同时使用两种时钟源。这两种硬件时钟源为 RTC 定时器和高分辨率定时器。默认情况下，是使用这两种定时器。下面我们将逐一介绍。

### 18.2 硬件设计

#### 18.2.1 例程功能

1. 通过 LCD 实时显示 RTC 时间
2. LED 闪烁，指示程序正在运行

#### 18.2.2 硬件资源

1. LED  
LED0 - IO1
2. 0.96 寸 LCD
3. RTC

#### 18.2.3 原理图

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

## 18.3 程序设计

### 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：

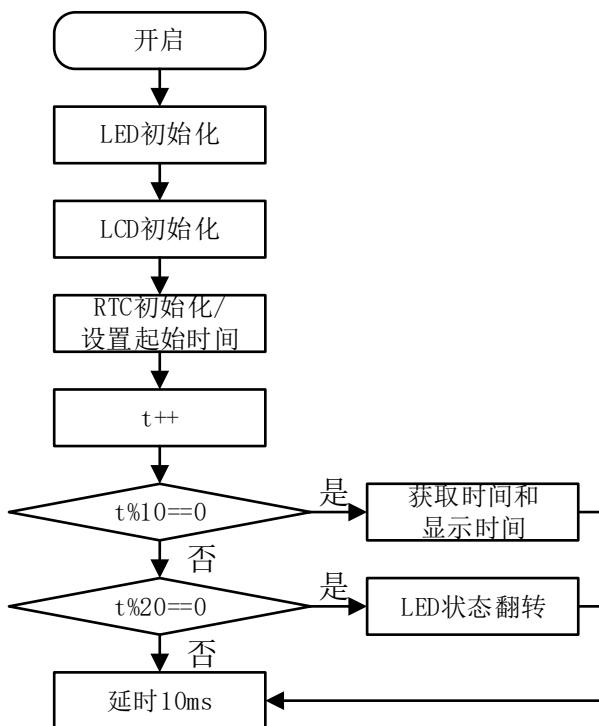


图 18.3.1.1 RTC 实验程序流程图

### 18.3.2 RTC 函数解析

由于 ESP32 并未给出 RTC 相关的 API 函数，因而笔者在设计例程时调用了 C 库中的一些函数来配置 RTC 时钟，这些函数的描述及其作用如下：

#### 1, 获取当前时间

该函数用于获取当前时间，其函数原型如下所示：

```
struct tm *localtime(const time_t *timer);
```

该函数的形参描述，如下表所示：

形参	描述
timer	这是指向表示日历时间的 time_t 值的指针

表 18.3.2.1 函数 localtime() 形参描述

返回值：无。

#### 2, 设置当前时间

该函数用于设置当前时间，其函数原型如下所示：

```
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

该函数的形参描述，如下表所示：

形参	描述
tv	设置当前时间结构体
tz	设置时区信息

表 18.3.2.2 函数 settimeofday() 形参描述

返回值：无。

### 18.3.3 RTC 驱动解析

在 IDF 版的 9\_RTC 例程中，作者在 9\_RTC\components\BSP 路径下新增了一个 RTC 文件夹，分别用于存放 esp\_rtc.c、esp\_rtc.h 两个文件。其中，esp\_rtc.h 文件负责声明 RTC，而 esp\_rtc.c 文件则实现了 RTC 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

#### 1, esp\_rtc.h 文件

```
/* 时间结构体，包括年月日周时分秒等信息 */
typedef struct
{
    uint8_t hour;          /* 时 */
    uint8_t min;           /* 分 */
    uint8_t sec;           /* 秒 */
    /* 公历年月日周 */
    uint16_t year;         /* 年 */
    uint8_t month;         /* 月 */
    uint8_t date;          /* 日 */
    uint8_t week;          /* 周 */
} _calendar_obj;

extern _calendar_obj calendar;      /* 时间结构体 */
```

#### 2, esp\_rtc.c 文件

```
calendar_obj calendar;      /* 时间结构体 */

/***
 * @brief      RTC 设置时间
 * @param      year    :年
 * @param      mon     :月
 * @param      mday    :日
 * @param      hour    :时
 * @param      min     :分
 * @param      sec     :秒
 * @retval     无
 */
void rtc_set_time(int year,int mon,int mday,int hour,int min,int sec)
{
    struct tm datetime;
    /* 设置时间 */
    datetime.tm_year = year - 1900;
    datetime.tm_mon = mon - 1;
    datetime.tm_mday = mday;
    datetime.tm_hour = hour;
    datetime.tm_min = min;
    datetime.tm_sec = sec;
    datetime.tm_isdst = -1;
    /* 获取 1970.1.1 以来的总秒数 */
    time_t second = mktime(&datetime);
    struct timeval val = { .tv_sec = second, .tv_usec = 0 };
    /* 设置当前时间 */
    settimeofday(&val, NULL);
}

/***
 * @brief      获取当前的时间
 * @param      无
 * @param      无
 */
void rtc_get_time(void)
{
    struct tm *datetime;
    time_t second;
```

```

/* 返回自 (1970.1.1 00:00:00 UTC) 经过的时间(秒) */
time(&second);
datetime = localtime(&second);

calendar.hour = datetime->tm_hour;           /* 时 */
calendar.min = datetime->tm_min;             /* 分 */
calendar.sec = datetime->tm_sec;              /* 秒 */

/* 公历年月日周 */
calendar.year = datetime->tm_year + 1900;    /* 年 */
calendar.month = datetime->tm_mon + 1;         /* 月 */
calendar.date = datetime->tm_mday;             /* 日 */

/* 周 */
calendar.week = rtc_get_week(calendar.year, calendar.month, calendar.date);

}

/**
 * @brief      将年月日时分秒转换成秒数
 * @note       输入公历日期得到星期(起始时间为：公元 0 年 3 月 1 日开始，输入往后的任何日期，都可以获取正确的星期)
 *            使用 基姆拉尔森计算公式 计算，原理说明见此贴：
 *            https://www.cnblogs.com/fengbohello/p/3264300.html
 * @param      syear : 年份
 * @param      smon : 月份
 * @param      sday : 日期
 * @retval     0, 星期天；1 ~ 6: 星期一 ~ 星期六
 */
uint8_t rtc_get_week(uint16_t year, uint8_t month, uint8_t day)
{
    uint8_t week = 0;

    if (month < 3)
    {
        month += 12;
        --year;
    }
    week = (day + 1 + 2 * month + 3 * (month + 1) /
            5 + year + (year >> 2) - year /
            100 + year / 400) % 7;
    return week;
}

```

以上三个获取、设置 RTC 时间、日期的函数，均是对 ESP IDF 中 RTC 驱动的简单封装。

#### 18.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    LCD
    LED
    SPI
    RTC)

set(include_dirs
    LCD
    LED
    SPI
    RTC)

set(requires
    driver
    newlib)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

```

```
component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

上述的红色 RTC 驱动以及 newlib 依赖库需要由开发者自行添加，以确保 RTC 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 RTC 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

### 18.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/* 定义字符数组用于显示周 */
char* weekdays[]={"Sunday", "Monday", "Tuesday", "Wednesday",
                   "Thursday", "Friday", "Saturday"};

/**
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;
    uint8_t tbuf[40];
    uint8_t t = 0;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    led_init(); /* 初始化 LED */
    spi2_init(); /* 初始化 SPI2 */
    lcd_init(); /* 初始化 LCD */
    rtc_set_time(2023,8,26,00,00,00); /* 设置 RTC 时间 */

    lcd_show_string(10, 40, 240, 32, 32, "ESP32",RED);
    lcd_show_string(10, 80, 240, 24, 24, "RTC Test",RED);
    lcd_show_string(10, 110, 240, 16, 16, "ATOM@ALIENTEK",RED);

    while (1)
    {
        t++;

        if ((t % 10) == 0) /* 每 100ms 更新一次显示数据 */
        {
            rtc_get_time();
            sprintf((char *)tbuf, "Time:%02d:%02d:%02d",
                    calendar.hour, calendar.min, calendar.sec);
            printf("Time:%02d:%02d:%02d\r\n", calendar.hour,
                   calendar.min, calendar.sec);
            lcd_show_string(10, 130, 210, 16, 16, (char *)tbuf,BLUE);
            sprintf((char *)tbuf, "Date:%04d-%02d-%02d",
                    calendar.year, calendar.month, calendar.date);
            printf("Date:%02d-%02d-%02d\r\n", calendar.year,
                   calendar.month, calendar.date);
            lcd_show_string(10, 150, 210, 16, 16, (char *)tbuf,BLUE);
            sprintf((char *)tbuf, "Week:%s", weekdays[calendar.week]);
            lcd_show_string(10, 170, 210, 16, 16, (char *)tbuf,BLUE);
        }
    }
}
```

```
if ((t % 20) == 0)
{
    LED_TOGGLE(); /* 每 200ms, 翻转一次 LED */
}

vTaskDelay(10);
}
```

从上面的代码中可以看到，在初始化完 RTC 后便每间隔 100 毫秒获取一次 RTC 的时间和日期，并在 LCD 上进行显示。

## 18.4 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时地显示着 RTC 的时间。



图 18.3.1 SPI LCD 显示效果图

## 第十九章 内部温度传感器实验

本章，我们将介绍 ESP32-S3 的内部温度传感器并使用它来读取温度值，然后在 LCD 模块上显示出来。本章分为如下几个小节：

- 19.1 IIC 简介
- 19.2 硬件设计
- 19.3 程序设计
- 19.4 下载验证

### 19.1 内部温度传感器简介

温度传感器生成一个随温度变化的电压。内部 ADC 将传感器电压转化为一个数字量。温度传感器的测量范围为  $-20^{\circ}\text{C}$  到  $110^{\circ}\text{C}$ 。温度传感器适用于监测芯片内部温度的变化，该温度值会随着微控制器时钟频率或 IO 负载的变化而变化。一般来讲，芯片内部温度会高于外部温度。ESP32-S3 温度传感器相关内容，请看《esp32-s3\_technical\_reference\_manual\_cn.pdf》技术手册 39.4 章节。

温度传感器的输出值需要使用转换公式转换成实际的温度值 ( $^{\circ}\text{C}$ )。转换公式如下：

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

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

测量范围 ( $^{\circ}\text{C}$ )	温度偏移 ( $^{\circ}\text{C}$ )
50 ~ 110	-2
20 ~ 100	-1
-10 ~ 80	0
-15 ~ 50	1
-20 ~ 20	2

表 19.1.1 温度传感器的温度偏移

### 19.2 硬件设计

#### 19.2.1 例程功能

本章实验功能简介：通过 ADC 的通道读取 ESP32-S3 内部温度传感器的电压值，并将其转换为温度值，显示在 SPILCD 屏上。

#### 19.2.2 硬件资源

1. 0.96 寸 LCD
2. 内部温度传感器

#### 19.2.3 原理图

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

### 19.3 程序设计

#### 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：

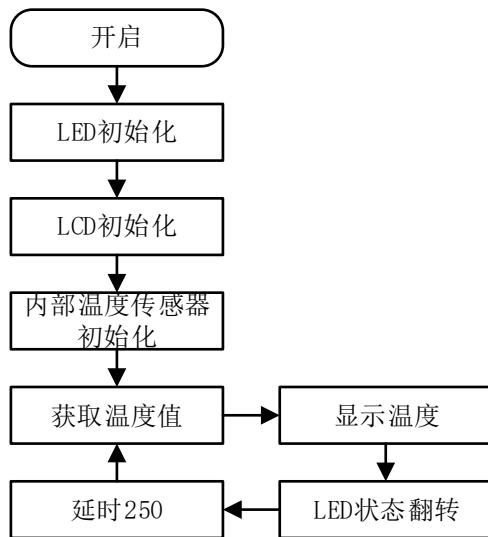


图 19.3.1.1 温度传感器实验程序流程图

### 19.3.2 内部温度传感器函数解析

ESP-IDF 提供了一套 API 来配置温度传感器。要使用此功能，需要导入必要的头文件：

```
#include "driver/temperature_sensor.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的温度传感器函数，这些函数的描述及其作用如下：

#### 1. 设置测试温度的最大与最小值

该函数用于配置测试温度的大小范围，其函数原型如下：

```
esp_err_t temperature_sensor_install(const temperature_sensor_config_t
                                      *tsens_config,
                                      temperature_sensor_handle_t
                                      *ret_tsens);
```

该函数的形参描述，如下表所示：

形参	描述
tsens_config	指向温度配置结构的指针
ret_tsens	返回温度传感器句柄的指针

表 19.3.2.1 函数 temperature\_sensor\_install()形参描述

返回值：ESP\_OK 表示配置成功，其他表示配置失败。

#### 2. 使能温度传感器

该函数用于使能温度传感器，其函数原型如下：

```
esp_err_t temperature_sensor_enable(temperature_sensor_handle_t tsens);
```

该函数的形参描述，如下表所示：

形参	描述
tsens	由 temperature_sensor_install() 创建的句柄

表 19.3.2.2 函数 temperature\_sensor\_enable()形参描述

返回值：ESP\_OK 表示使能成功，其他表示使能失败。

#### 3. 获取传输的传感器数据

该函数用于获取传输的传感器数据，其函数原型如下：

```
esp_err_t temperature_sensor_get_celsius(temperature_sensor_handle_t tsens,
                                         float *out_celsius);
```

该函数的形参描述，如下表所示：

形参	描述
tsens	由 temperature_sensor_install() 创建的句柄
out_celsius	度量值输出值

表 19.3.2.3 函数 temperature\_sensor\_get\_celsius()形参描述

返回值: ESP\_OK 表示获取数据成功, 其他表示获取数据失败。

#### 4. 失能温度传感器

该函数用于获取传输的传感器数据, 其函数原型如下:

```
esp_err_t temperature_sensor_disable(temperature_sensor_handle_t tsens);
```

该函数的形参描述, 如下表所示:

形参	描述
tsens	由 temperature_sensor_install() 创建的句柄

表 19.3.2.7 函数 temperature\_sensor\_disable()形参描述

返回值: ESP\_OK 表示失能成功, 其他表示失能失败。

### 19.3.3 内部温度传感器驱动解析

在 IDF 版 9\_internal\_temperature 例程中, 作者在 9\_internal\_temperature\components\BSP 路径下新增了一个 SENSOR 文件夹, 分别用于存放 sensor.c、sensor.h 这两个文件。其中, sensor.h 文件负责声明温度传感器相关的函数和变量, 而 sensor.c 文件则实现了温度传感器的驱动代码。下面, 我们将详细解析这两个文件的实现内容。

#### 1, sensor.h 文件

```
/* 参数定义 */
#define SENSOR_RANGE_MIN    20      /* 要测试温度的最小值 */
#define SENSOR_RANGE_MAX    50      /* 要测试温度的最大值 */
```

#### 2, sensor.c 文件

在上述 sensor.h 文件中我们通过宏定义的方式定义了待测试温度的最大与最小值, 该值在不超过理论值的基础上, 开发者可以自行定义。

```
esp_err_t rev_flag;
temperature_sensor_handle_t temp_handle = NULL; /* 温度传感器句柄 */

/**
 * @brief      初始化内部温度传感器
 * @param      无
 * @retval     无
 */
void temperature_sensor_init(void)
{
    temperature_sensor_config_t temp_sensor;

    temp_sensor.range_min = SENSOR_RANGE_MIN; /* 要测试温度的最小值 */
    temp_sensor.range_max = SENSOR_RANGE_MAX; /* 要测试温度的最大值 */

    rev_flag |= temperature_sensor_install(&temp_sensor, &temp_handle);
    ESP_ERROR_CHECK(rev_flag);
}

/**
 * @brief      获取内部温度传感器温度值
 * @param      无
 * @retval     返回内部温度值
 */
short sensor_get_temperature(void)
{
    float temp;

    /* 启用温度传感器 */
    rev_flag |= temperature_sensor_enable(temp_handle);

    /* 获取传输的传感器数据 */
    rev_flag |= temperature_sensor_get_celsius(temp_handle, &temp);
```

```

/* 温度传感器使用完毕后，禁用温度传感器，节约功耗 */
rev_flag |= temperature_sensor_disable(temp_handle);
ESP_ERROR_CHECK(rev_flag);

return temp;
}

```

初始化内部温度传感器后，再将温度传感器使能以获取传感器数据，最终以返回值的形式将数据返回到数据处理的函数。

#### 19.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    LCD
    LED
    SENSOR
    SPI)

set(include_dirs
    LCD
    LED
    SENSOR
    SPI)

set(requirements
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
    INCLUDE_DIRS ${include_dirs} REQUIRES ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 SENSOR 驱动需要由开发者自行添加，以确保温度传感器驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了温度传感器驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 19.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/**
 * @brief 程序入口
 * @param 无
 * @retval 无
 */
void app_main(void)
{
    int16_t temp;
    esp_err_t ret;

    /* 初始化 NVS */
    ret = nvs_flash_init();

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    /* 初始化 LED */
    led_init();

    /* 初始化 SPI2 */
    spi2_init();
}

```

```
/* 初始化 LCD */
lcd_init();

/* 初始化内部温度传感器 */
temperature_sensor_init();

lcd_show_string(30, 120, 200, 16, 16, "TEMPERATE: 00.00C", BLUE);

while(1)
{
    /* 得到温度值 */
    temp = sensor_get_temperature();

    if (temp < 0)
    {
        temp = -temp;

        /* 显示符号 */
        lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, "-", BLUE);
    }
    else
    {
        /* 无符号 */
        lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, " ", BLUE);
    }

    /* 显示整数部分 */
    lcd_show_xnum(30 + 11 * 8, 120, temp, 2, 16, 0, BLUE);

    /* 显示小数部分 */
    lcd_show_xnum(30 + 14 * 8, 120, temp * 100 % 100, 2, 16, 0x80, BLUE);

    /* LED 闪烁, 提示程序运行 */
    LED_TOGGLE();
    vTaskDelay(250);
}
}
```

main 函数代码比较简单，主要是通过 `sensor_get_temperature()` 函数读取 ESP32-S3 内部温度值，最后在 SPILCD 上显示。

## 19.4 下载验证

将程序下载到 DNESP32S3M 最小系统板后，LCD 显示的内容如下图所示：



图 19.4.1 内部温度传感器实验测试图

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

## 第二十章 RNG 实验

本章，我们将介绍 ESP32-S3 的硬件随机数发生器。我们使用 BOOT 按键来获取硬件随机数，并且将获取到的随机数值显示在 LCD 上面。同时，使用 LED 指示程序运行状态。

本章分为以下几个小节：

20.1 随机数发生器简介

20.2 硬件设计

20.3 程序设计

20.4 下载验证

### 20.1 随机数发生器简介

ESP32-S3 内置一个真随机数发生器（RNG），其生成的 32 位随机数可作为加密等操作的基础。ESP32-S3 的随机数发生器可通过物理过程而非算法生成真随机数，所有生成的随机数在特定范围内出现的概率完全一致。

#### 20.1.1 RNG 功能描述

下面先来了解噪声源，通过学习噪声源会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。EDP32-S3 的随机数发生器噪声源如下图所示：

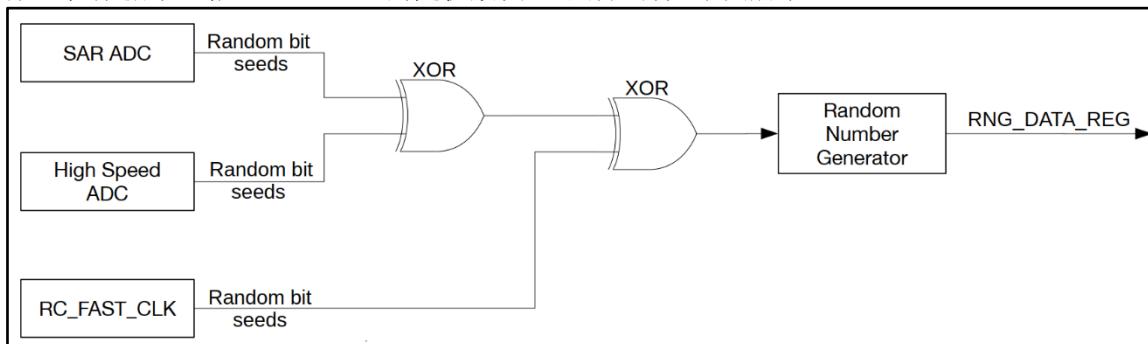


图 20.1.1.1 ESP32-S3 随机数发生器噪声源

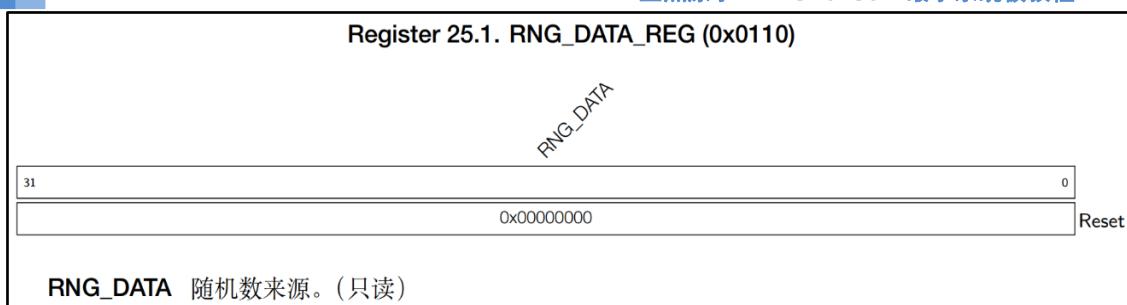
系统可以从随机数发生器的寄存器 RNG\_DATA\_REG 中读取随机数，每个读到的 32 位随机数都是真随机数，噪声源为系统中的热噪声和异步时钟。具体来说，这些热噪声可以来自 SAR ADC 或高速 ADC 或两者兼有。当芯片的 SAR ADC 或高速 ADC 工作时，就会产生比特流，并通过异或(XOR)逻辑运算作为随机数种子进入随机数生成器。当为数字内核使能 RC\_FAST\_CLK 时钟时，随机数发生器也会对 RC\_FAST\_CLK(20MHz) 进行采样，作为随机数种子。RC\_FAST\_CLK 是一种异步时钟源，由于存在电路亚稳态，因此可以提高随机数发生器的熵值。然而，为了保证随机数发生器可以获得最大熵值，仍建议在使用随机数发生器时至少使能一路 ADC 作为随机数种子。

#### 20.1.2 RNG 随机数寄存器

RNG 随机数寄存器描述如下所示：

名称	描述	地址	访问
RNG_DATA_REG	随机数数据	0x0110	只读

表 20.1.2.1 RNG 寄存器列表



**RNG\_DATA** 随机数来源。(只读)

图 20.1.2.1 RNG\_CR 寄存器

需要注意的是，这里的地址都是相对于随机数发生器基地址的地址偏移量。

## 20.2 硬件设计

### 20.2.1 例程功能

本实验使用 ESP32-S3 自带的硬件随机数生成器 (RNG)，获取随机数，并显示在 LCD 屏幕上。按 BOOT 按键可以获取一次随机数。同时程序自动获取 0~9 范围内的随机数，显示在屏幕上。LED 闪烁用于提示程序正在运行。

### 20.2.2 硬件资源

1. LED  
    LED - IO1
2. 独立按键
3. 0.96 寸 LCD
4. RNG (硬件随机数生成器)

### 20.2.3 原理图

RNG 属于 ESP32-S3 内部资源，通过软件设置好就可以了。本实验通过配合按键获取随机数和通过 LCD 显示。

## 20.3 程序设计

### 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：

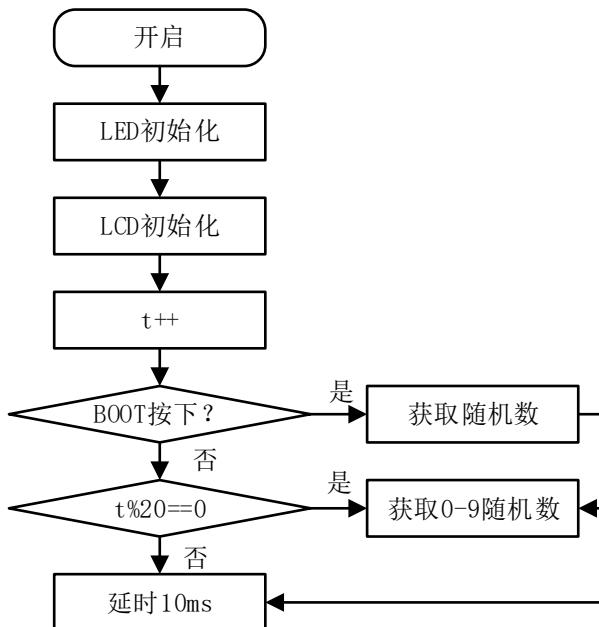


图 20.3.1.1 随机数实验程序流程图

### 20.3.2 RNG 函数解析

ESP-IDF 提供了一套 API 来配置和使用 RNG。要使用此功能，需要导入必要的头文件：

```
#include "esp_random.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 RNG 函数，这些函数的描述及其作用如下：

#### 1, 得到随机数

该函数用于获取一个 32 位的硬件随机数，其函数原型如下：

```
uint32_t esp_random(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 21.3.2.1 函数 esp\_random() 形参描述

返回值：0 和 UINT32\_MAX 之间的随机值。

### 20.3.3 RNG 驱动解析

在 IDF 版 10\_rng 例程中，作者在 10\_rng\components\BSP 路径下新增了一个 RNG 文件夹，分别用于存放 rng.c、rng.h 这两个文件。其中，rng.h 文件负责声明 RNG 相关的函数和变量，而 rng.c 文件则实现了 RNG 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

#### 1, rng.h 文件

```
/* 函数声明 */
uint32_t rng_get_random_num(void); /* 得到随机数 */
int rng_get_random_range(int min, int max); /* 得到某个范围内的随机数 */
```

#### 2, rng.c 文件

```
/**
 * @brief      得到随机数
 * @param      无
 * @retval     获取到的随机数(32bit)
 */
uint32_t rng_get_random_num(void)
{
    uint32_t randomnum;
    randomnum = esp_random();
```

```

        return randomnum;
    }

/***
 * @brief      得到某个范围内的随机数
 * @param      min,max: 最小,最大值.
 * @retval     得到的随机数(rval),满足:min<=rval<=max
 */
int rng_get_random_range(int min, int max)
{
    uint32_t randomnum;

    randomnum = esp_random();

    return randomnum % (max - min + 1) + min;
}

```

从上述代码中，我们不难看出，对于 RNG 我们并没有相应的初始化函数，ESP32 IDF 提供了相应的 API 函数，我们只需调用即可，具体的在上一小节我们已经介绍过了，在此不做出赘述。

#### 20.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    KEY
    LCD
    LED
    RNG
    SPI)

set(include_dirs
    KEY
    LCD
    LED
    RNG
    SPI)

set(requires
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
                       INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 RNG 驱动需要由开发者自行添加，以确保 RNG 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 RNG 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 20.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    uint8_t key;
    uint32_t random;
    uint8_t t = 0;
    esp_err_t ret;

```

```
ret = nvs_flash_init(); /* 初始化 NVS */

if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||  
    ret == ESP_ERR_NVS_NEW_VERSION_FOUND)  
{  
    ESP_ERROR_CHECK(nvs_flash_erase());  
    ret = nvs_flash_init();  
}

led_init(); /* 初始化 LED */  
spi2_init(); /* 初始化 SPI2 */  
lcd_init(); /* 初始化 LCD */  
key_init(); /* 初始化按键 */  
lcd_show_string(0, 0, lcd_self.width, 16, 16, "Random Num:", RED);  
lcd_show_string(0, 20, lcd_self.width, 16, 16, "Random Num[0-9]:", RED);

while(1)
{
    key = key_scan();

    if (key == BOOT) /* 获取随机数并显示至 LCD */
    {
        random = rng_get_random_num();
        lcd_show_num(30 + 8 * 11, 150, random, 10, 16, BLUE);
    }
    if ((t % 20) == 0) /* 获取 0~9 间的随机数并显示至 LCD */
    {
        LED_TOGGLE(); /* 每 200ms, 翻转一次 LED */
        random = rng_get_random_range(0, 9); /* 取[0,9]区间的随机数 */
        lcd_show_num(30 + 8 * 16, 180, random, 1, 16, BLUE); /* 显示随机数 */
    }
    vTaskDelay(10);
    t++;
}
}
```

该部分代码也比较简单，在所有外设初始化成功后，进入死循环，等待 BOOT 按键按下，如果按下，则调用 `rng_get_random_num` 函数，读取随机数值，并将读到的随机数显示在 LCD 上面。每隔 200ms 获取一次区间[0,9]的随机数，并实时显示在液晶上。同时 LED，周期性闪烁，400ms 闪烁一次。

## 20.4 下载验证

将程序下载到 DNESP32S3M 最小系统板后，可以看到 LED 不停的闪烁，提示程序已经在运行了。然后我们按下 BOOT，就可以在屏幕上看到获取到的随机数。同时，就算不按 BOOT，程序也会自动的获取 0~9 区间的随机数显示在 LCD 上面。

## 第二十一章 SPI\_SDCARD 实验

本很多单片机系统都需要大容量存储设备，以存储数据。目前常用的有 U 盘，FLASH 芯片，SD 卡等。他们各有优点，综合比较，最适合单片机系统的莫过于 SD 卡了，它不仅容量可以做到很大（32GB 以上），支持 SPI/SDIO 驱动，而且有多种体积的尺寸可供选择（标准的 SD 卡尺寸及 Micro SD 卡尺寸等），能满足不同应用的要求。

只需要少数几个 IO 口即可外扩一个高达 32GB 或以上的外部存储器，容量从几十 M 到几十 G 选择范围很大，更换也很方便，编程也简单，是单片机大容量外部存储器的首选。

正点原子 ESP32-S3 最小系统板使用的接口是 Micro SD 卡接口，卡座带自锁功能，SD SPI 主机驱动程序基于 SPI Master Driver 实现。借助 SPI 主控驱动程序，SD 卡及其他 SPI 设备可以共享同一 SPI 总线。SPI 主机驱动程序将处理来自不同任务的独占访问。在本章中，我们将向大家介绍，如何在正点原子 ESP32-S3 最小系统板上实现 Micro SD 卡的读取。本章分为如下几个部分：

本章分为如下几个小节：

21.1 SD 卡简介

21.2 硬件设计

21.3 程序设计

21.4 下载验证

### 22.1 SD 卡简介

#### 22.1.1 SD 物理结构

SD 卡的规范由 SD 卡协会明确，可以访问 <https://www.sdcard.org> 查阅更多标准。SD 卡主要有 SD、Mini SD 和 microSD(原名 TF 卡，2004 年正式更名为 Micro SD Card，为方便本文用 microSD 表示)三种类型，Mini SD 已经被 microSD 取代，使用得不多，根据最新的 SD 卡规格列出的参数如下表所示：

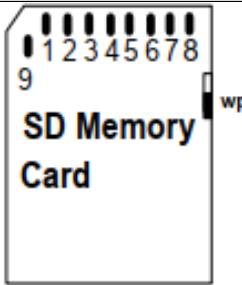
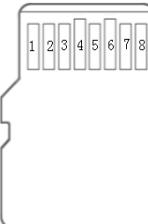
形状	SD	microSD
尺寸	 32 x 24 x 2.1 mm, 32 x 24 x 1.4 mm, 约重 1.2g~2.5g	 11 x 15 x 1.0 mm, 约重 0.5g
卡片种类 (容量范围)	SD(≤2GB)、SDHC(2GB~32GB)、SDXC(32GB ~2TB)、SDUC(2TB ~128TB)	
硬件规格	脚位数	High Speed and UHS-I : 9 pins UHS-II and UHS-III: 17 pins SD Express 1-lane: 17-19 pins SD Express 2-lane: 25-27 pins
	电压范围	3.3V 版本: 2.7V - 3.6V 1.8V 低电压版: 1.70V-1.95V
	防写开关	是
		否

表 21.1.1.1 SD 卡的主要规格参数

上述表格的“脚位数”，对应于实卡上的“金手指”数，不同类型的卡的触点数量不同，访问的速度也不相同。SD 卡允许了不同的接口来访问它的内部存储单元。最常见的是 SDIO 模式和 SPI 模式，根据这两种接口模式，我们也列出 SD 卡引脚对应于这两种不同的电路模式的引脚功能定义，如下表所示。

SD 卡		SD Mode			SPI Mode		
引脚编号	引脚名	引脚类型	功能描述	引脚名	引脚类型	功能描述	
1	CD/DAT3	I/O/PP	卡识别/数据线位 3	CS	I3	片选, 低电平有效	
2	CMD	I/O/PP	命令/响应	DI	I	数据输入	
3	VSS1	S	电源地	VSS	S	电源地	
4	VDD	S	DC 电源正极	VDD	S	DC 电源正极	
5	CLK	I	Clock	SCLK	I	时钟	
6	VSS2	S	电源地	VSS2	S	电源地	
7	DAT0	I/O/PP	数据线位 0	DO	O/PP	数据输出	
8	DAT1	I/O/PP	数据线位 1	RSV			
9	DAT2	I/O/PP	数据线位 2	RSV			

表 21.1.1.2 SD 卡引脚编号（注：S:电源 I: 输入 O: 推挽输出 PP: 推挽）

我们对比着来看一下 microSD 引脚，可见只比 SD 卡少了一个电源引脚 VSS2，其它的引脚功能类似。

microSD		SD Mode			SPI Mode		
引脚编号	引脚名	引脚类型	功能描述	引脚名	引脚类型	功能描述	
1	DAT2	I/O/PP	数据线位 2	RSV			
2	CD/DAT3	I/O/PP	卡识别/数据线位 3	CS	I	片选低电平有效	
3	CMD	PP	命令/响应	DI	I	数据输入	
4	VDD	S	DC 电源正极	VDD	S	DC 电源正极	
5	CLK	I	时钟	SCLK	I	时钟	
6	VSS	S	电源地	VSS	S	电源地	
7	DAT0	I/O/PP	数据线位 0	DO	O/PP	数据输出	
8	DAT1	I/O/PP	数据线位 1	RSV			

表 21.1.1.3 microSD 卡引脚编号（注：S:电源 I: 输入 O: 推挽输出 PP: 推挽）

SD 卡和 Micro SD 只有引脚和形状大小不同，内部结构类似，操作时序完全相同，可以使用完全相同的代码驱动，下面以 9'Pin SD 卡的内部结构为例，展示 SD 卡的存储结构。

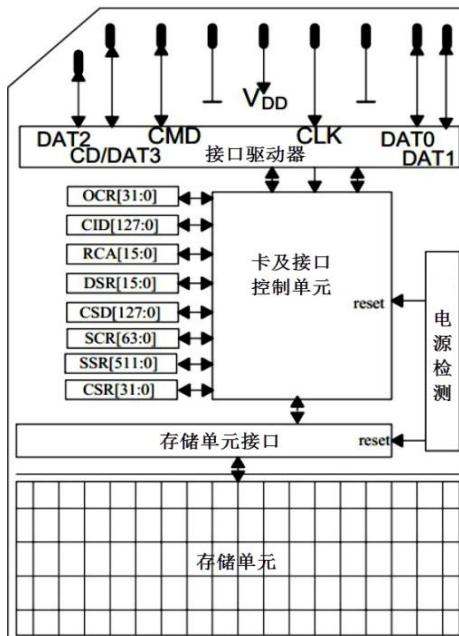


图 21.1.1.1 SD 卡内部物理结构 (RCA 寄存器在 SPI 模式下不可访问)

SD 卡有自己的寄存器，但它不能直接进行读写操作，需要通过命令来控制，SDIO 协议定义了一些命令用于实现某一特定功能，SD 卡根据收到的命令要求对内部寄存器进行修改。图 37.1.1.4 中描述的 SD 卡的寄存器是我们和 SD 卡进行数据通讯的主要通道，如下：

名称	位宽	描述
CID	128	卡标识(Card identification):每个卡都是唯一的
RCA	16	相对地址(Relative card address):卡的本地系统地址，初始化时，动态地由卡建议，经主机核准。
DSR	16	驱动级寄存器(Driver Stage Register):配置卡的输出驱动
CSD	128	卡的特定数据(Card Specific Data):卡的操作条件信息
SCR	64	SD 配置寄存器(SD Configuration Register):SD 卡特殊特性信息
OCR	32	操作条件寄存器(Operation conditions register): 卡电源和状态标识
SSR	512	SD 状态(SD Status):SD 卡专有特征的信息
CSR	32	卡状态(Card Status):卡状态信息

表 21.1.1.4 SD 卡寄存器信息

关于 SD 卡的更多信息和硬件设计规范可以参考 SD 卡协议《Physical Layer Simplified Specification Version 2.00》的相关章节。

## 21.1.2 命令和响应

一个完整的 SD 卡操作过程是：主机(单片机等)发起“命令”，SD 卡根据命令的内容决定是否发送响应信息及数据等，如果是数据读/写操作，主机还需要发送停止读/写数据的命令来结束本次操作，这意味着主机发起命令指令后，SD 卡可以没有响应、数据等过程，这取决于命令的含义。这一过程如下图所示。

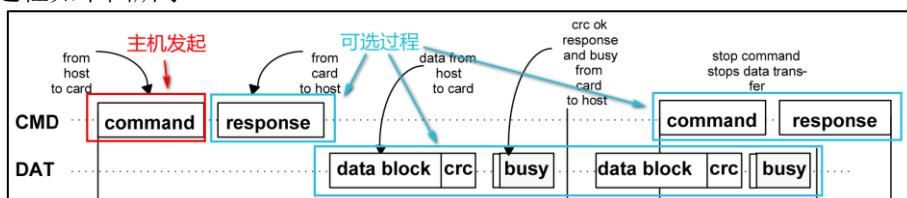


图 21.1.2.1 SD 卡命令格式

SD 卡有多种命令和响应，它们的格式定义及含义在《SD 卡协议 V2.0》的第三和第四章有详细介绍，发送命令时主机只能通过 CMD 引脚发送给 SD 卡，串行逐位发送时先发送最高位 (MSB)，然后是次高位这样类推……接下来，我们看看 SD 卡的命令格式，如下表所示：

字节	字节1			字节2~5	字节6	
位	47	46	45:40	39:8	7:1	0
描述	0	1	command	命令参数	CRC7	1

表 21.1.2.1 SD 卡控制命令格式

SD 卡的命令固定为 48 位，由 6 个字节组成，字节 1 的最高 2 位固定为 01，低 6 位为命令号（比如 CMD16，为 10000B 即 16 进制的 0X10，完整的 CMD16，第一个字节为 01010000，即 0X10+0X40）。字节 2~5 为命令参数，有些命令是没有参数的。字节 6 的高七位为 CRC 值，最低位恒定为 1。

SD 卡的命令总共有 12 类，分为 Class0~Class11，本章，我们仅介绍几个比较重要的命令，如下表所示：

命令	参数	响应	描述
CMD0(0X00)	NONE	R1	复位SD卡
CMD8(0X08)	VHS+Checkpattern	R7	发送接口状态命令
CMD9(0X09)	NONE	R1	读取卡特定数据寄存器
CMD10(0X0A)	NONE	R1	读取卡标志数据寄存器
CMD16(0X10)	块大小	R1	设置块大小（字节数）
CMD17(0X11)	地址	R1	读取一个块的数据
CMD24(0X18)	地址	R1	写入一个块的数据
CMD41(0X29)	NONE	R3	发送给主机容量支持信息和激活卡初始化过程
CMD55(0X37)	NONE	R1	告诉SD卡，下一个特定应用命令
CMD58(0X3A)	NONE	R3	读取OCR寄存器

表 21.1.2.2 SD 卡部分命令

上表中，大部分的命令是初始化的时候用的。表中的 R1、R3 和 R7 等是 SD 卡的应答信号，每个响应也有规定好的格式，如下图所示：

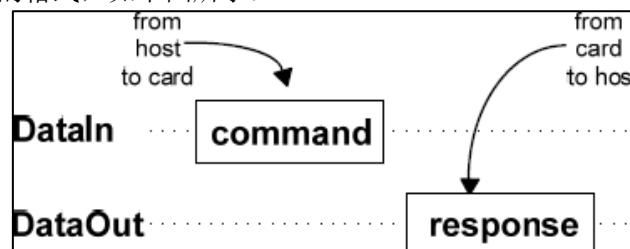


图 21.1.2.2 SD 卡命令传输过程

在规定为有响应的命令下，每发送一个命令，SD 卡都会给出一个应答，以告知主机该命令的执行情况，或者返回主机需要获取的数据，应答可以是 R1~R7，R1 的应答，各位描述如下表所示：

R1 响应	描述	起始位	传输位	命令号	卡状态	CRC7	终止位
	Bit	47	46	[45:40]	[39:8]	[7:1]	0
	位宽	1	1	6	32	7	1
	值	"0"	"0"	x	x	x	"1"

表 21.1.2.3 R1 响应

R2~R7 的响应，限于篇幅，我们就不介绍了，但需要注意的是除了 R2 响应是 128 位外，其它的响应都是 48 位，请大家参考 SD 卡 2.0 协议。

### 21.1.3 卡模式

SD 卡系统(包括主机和 SD 卡)定义了 SD 卡的工作模式，在每个操作模式下，SD 卡都有几种状态，参考下表，状态之间通过命令控制实现卡状态的切换。

无效模式 (Inactive)	无效状态(Inactive State)
卡识别模 (Card identification mode)	空闲状态(Idle State)
	准备状态(Ready State)
	识别状态(Identification State)
数据传输模式 (Data transfer mode)	待机状态(Stand-by State)
	传输状态(Transfer State)
	发送数据状态(Sending-data State)
	接收数据状态(Receive-data State)
	编程状态(Programming State)
	断开连接状态(Disconnect State)

表 21.1.3.1 SD 卡状态与操作模式

对于我们来说两种有效操作模式：卡识别模式和数据传输模式。在系统复位后，主机处于卡识别模式，寻找总线上可用的 SDIO 设备，对 SD 卡进行数据读写之前需要识别卡的种类：V1.0 标准卡、V2.0 标准卡、V2.0 高容量卡或者不被识别卡；同时，SD 卡也处于卡识别模式，直到被主机识别到，即当 SD 卡在卡识别状态接收到 CMD3 (SEND\_RCA)命令后，SD 卡就进入数据传输模式，而主机在总线上所有卡被识别后也进入数据传输模式。

在卡识别模式下，主机会复位所有处于“卡识别模式”的 SD 卡，确认其工作电压范围，识别 SD 卡类型，并且获取 SD 卡的相对地址(卡相对地址较短，便于寻址)。在卡识别过程中，要求 SD 卡工作在识别时钟频率 FOD 的状态下。卡识别模式下 SD 卡状态转换如图 37.1.3.1。

主机上电后，所有卡处于空闲状态，包括当前处于无效状态的卡。主机也可以发送 GO\_IDLE\_STATE(CMD0)让所有卡软复位从而进入空闲状态，但当前处于无效状态的卡并不会复位。

主机在开始与卡通信前，需要先确定双方在互相支持的电压范围内。SD 卡有一个电压支持范围，主机当前电压必须在该范围可能才能与卡正常通信。SEND\_IF\_COND(CMD8)命令就是用于验证卡接口操作条件的(主要是电压支持)。卡会根据命令的参数来检测操作条件匹配性，如果卡支持主机电压就产生响应，否则不响应。而主机则根据响应内容确定卡的电压匹配性。CMD8 是 SD 卡标准 V2.0 版本才有的新命令，所以如果主机有接收到响应，可以判断卡为 V2.0 或更高版本 SD 卡。

SD\_SEND\_OP\_COND(ACMD41)命令可以识别或拒绝不匹配它的电压范围的卡。ACMD41 命令的 VDD 电压参数用于设置主机支持电压范围，卡响应会返回卡支持的电压范围。对于对 CMD8 有响应的卡，把 ACMD41 命令的 HCS 位设置为 1，可以测试卡的容量类型，如果卡响应的 CCS 位为 1 说明为高容量 SD 卡，否则为标准卡。卡在响应 ACMD41 之后进入准备状态，不响应 ACMD41 的卡为不可用卡，进入无效状态。ACMD41 是应用特定命令，发送该命令之前必须先发 CMD55。

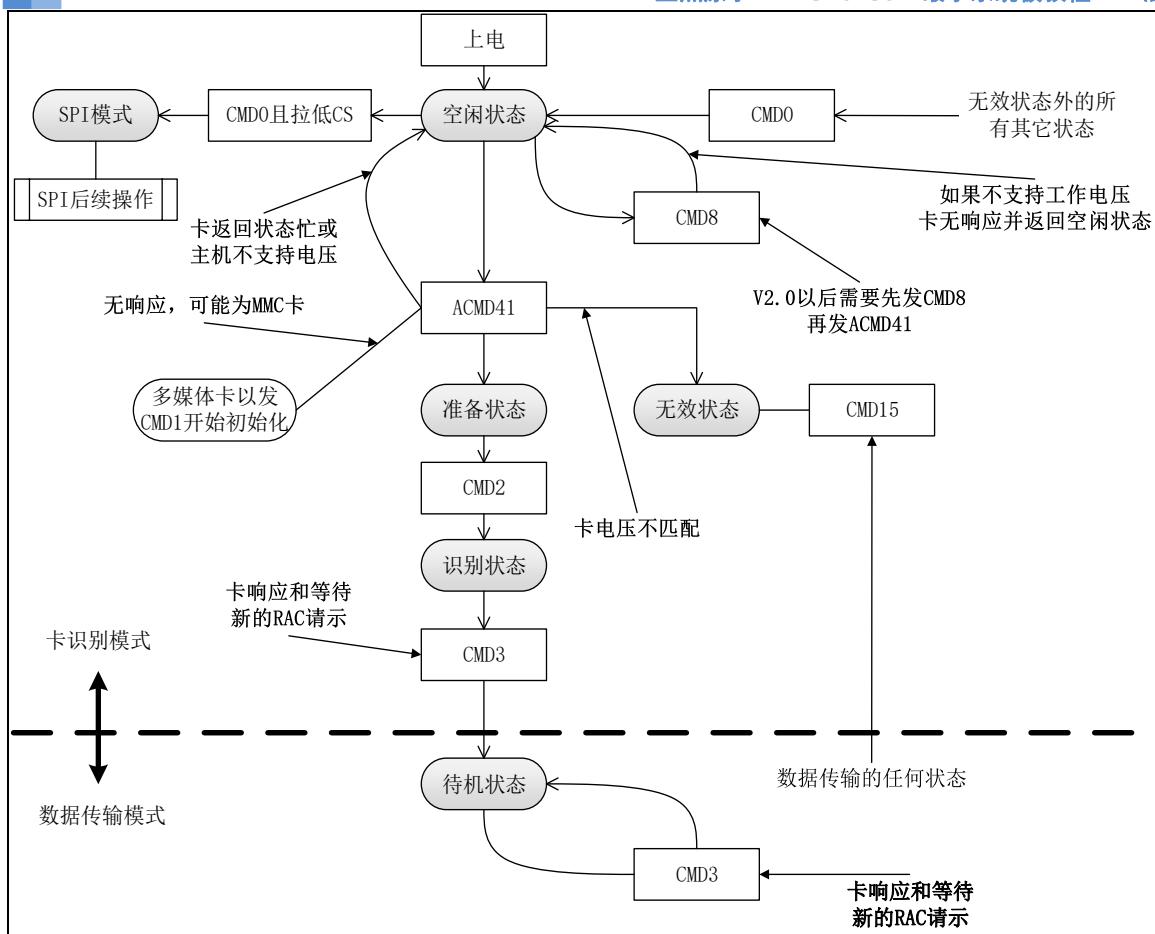


图 21.1.3.1 卡识别模式状态转换图

ALL\_SEND\_CID(CMD2)用来控制所有卡返回它们的卡识别号(CID)，处于准备状态的卡在发送 CID 之后就进入识别状态。之后主机就发送 SEND\_RELATIVE\_ADDR(CMD3)命令，让卡自己推荐一个相对地址(RCA)并响应命令。这个 RCA 是 16bit 地址，而 CID 是 128bit 地址，使用 RCA 简化通信。卡在接收到 CMD3 并发出响应后就进入数据传输模式，并处于待机状态，主机在获取所有卡 RCA 之后也进入数据传输模式。

#### 21.1.4 数据模式

在数据模式下我们可以对 SD 卡的存储块进行读写访问操作。SD 卡上电后默认以一位数据总线访问，可以通过指令设置为宽总线模式，可以同时使有 4 位总线并行读写数据，这样对于支持宽总线模式的接口（如：SDIO 和 QSPI 等）都能加快数据操作速度。

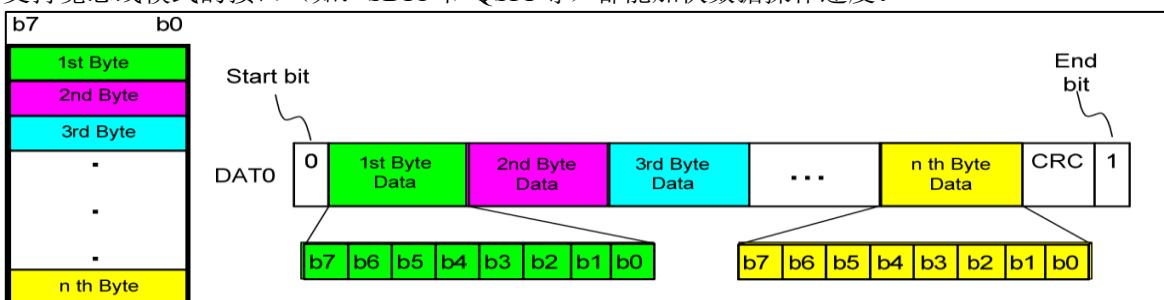


图 21.1.4.1 1 位数据线传输 8bit 的数据流格式

SD 卡有两种数据模式，一种是常规的 8 位宽，即一次按一字节传输，另一种是一次按 512 字节传输，我们只介绍前面一种。当按 8-bit 连续传输时，每次传输从最低字节开始，每字节从最高位(MSB)开始发送，当使用一条数据线时，只能通过 DAT0 进行数据传输，那它的数据传输

结构如下图所示。

当使用 4 线模式传输 8-bit 结构的数据时，数据仍按 MSB 先发送的原则，DAT[3:0]的高位发送高数据位，低位发送低数据位。硬件支持的情况下，使用 4 线传输可以提升传输速率。

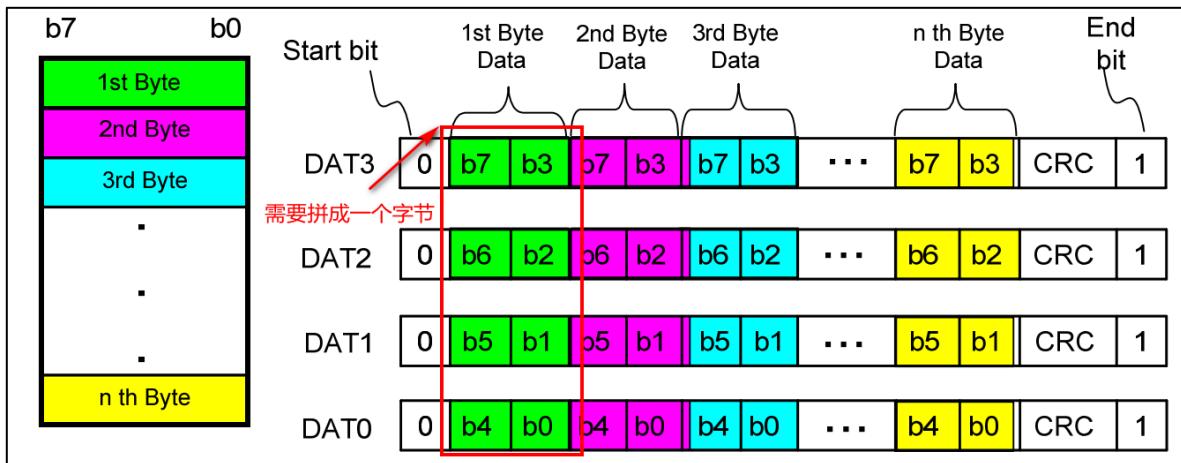


图 21.1.4.2 4 位数据线传输 8bit 格式的数据流格式

只有 SD 卡系统处于数据传输模式下才可以进行数据读写操作。数据传输模式下可以将主机 SD 时钟频率设置为 FPP，默认最高为 25MHz，频率切换可以通过 CMD4 命令来实现。数据传输模式下，SD 卡状态转换过程见下图。

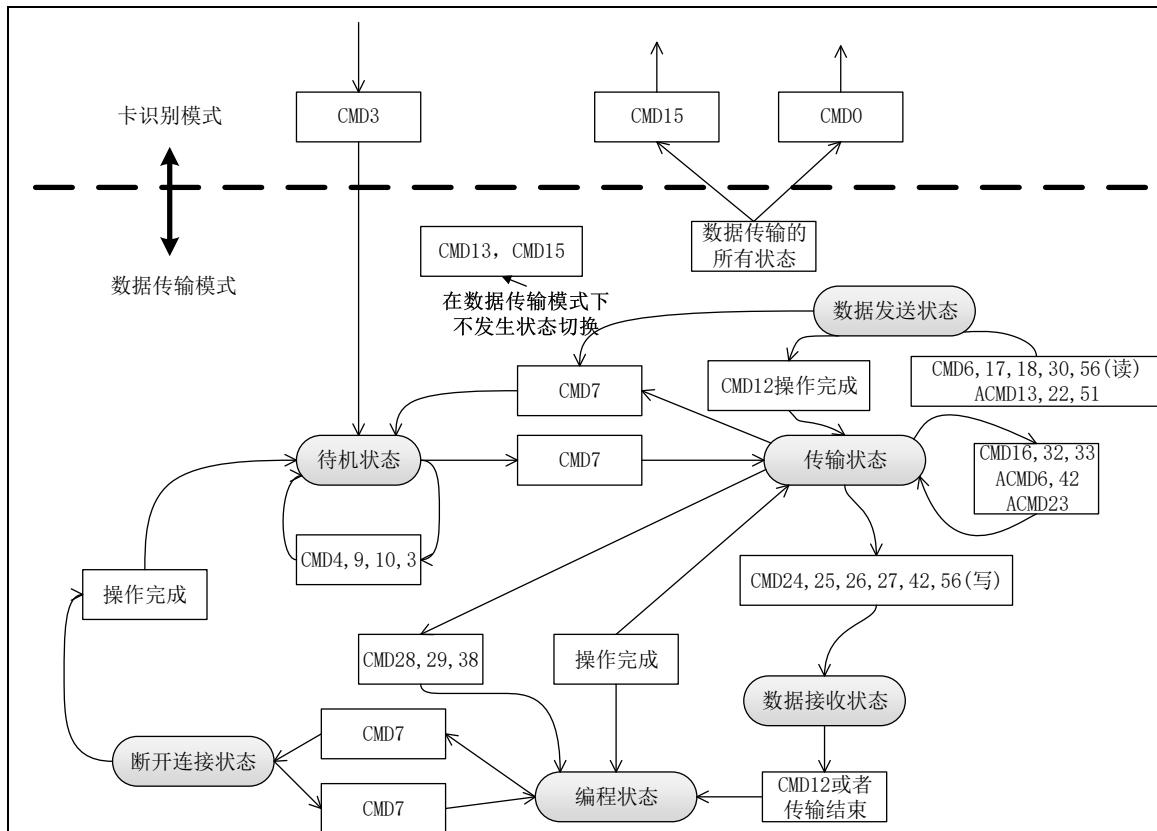


图 21.1.4.3 数据传输模式卡状态转换

CMD7 用来选定和取消指定的卡，卡在待机状态下还不能进行数据通信，因为总线上可能有多个卡都是出于待机状态，必须选择一个 RCA 地址目标卡使其进入传输状态才可以进行数据通信。同时通过 CMD7 命令也可以让已经被选择的目标卡返回到待机状态。

数据传输模式下的数据通信都是主机和目标卡之间通过寻址命令点对点进行的。卡处于传输状态下可以通过命令对卡进行数据读写、擦除。CMD12 可以中断正在进行的数据通信，让卡返回到传输状态。CMD0 和 CMD15 会中止任何数据编程操作，返回卡识别模式，注意谨慎使

用，不当操作可能导致卡数据被损坏。

至此，我们已经介绍了 SD 卡操作的一些知识，并知道了 SD 卡操作的命令、响应和数据传输等状态，接下来我们来分析实际的硬件接口如何向 SD 卡发送我们需要的数据。

### 21.1.5 SD 卡初始化流程

#### 1. 模式下的 SD 卡初始化

这一节，我们来看看 SD 卡的初始化流程，要实现 SDIO 驱动 SD 卡，最重要的步骤就是 SD 卡的初始化，只要 SD 卡初始化完成了，那么剩下的（读写操作）就简单了，所以我们这里重点介绍 SD 卡的初始化。从《SD 卡 2.0 协议》（见光盘资料）文档，我们得到 SD 卡初始化流程图如下图所示：

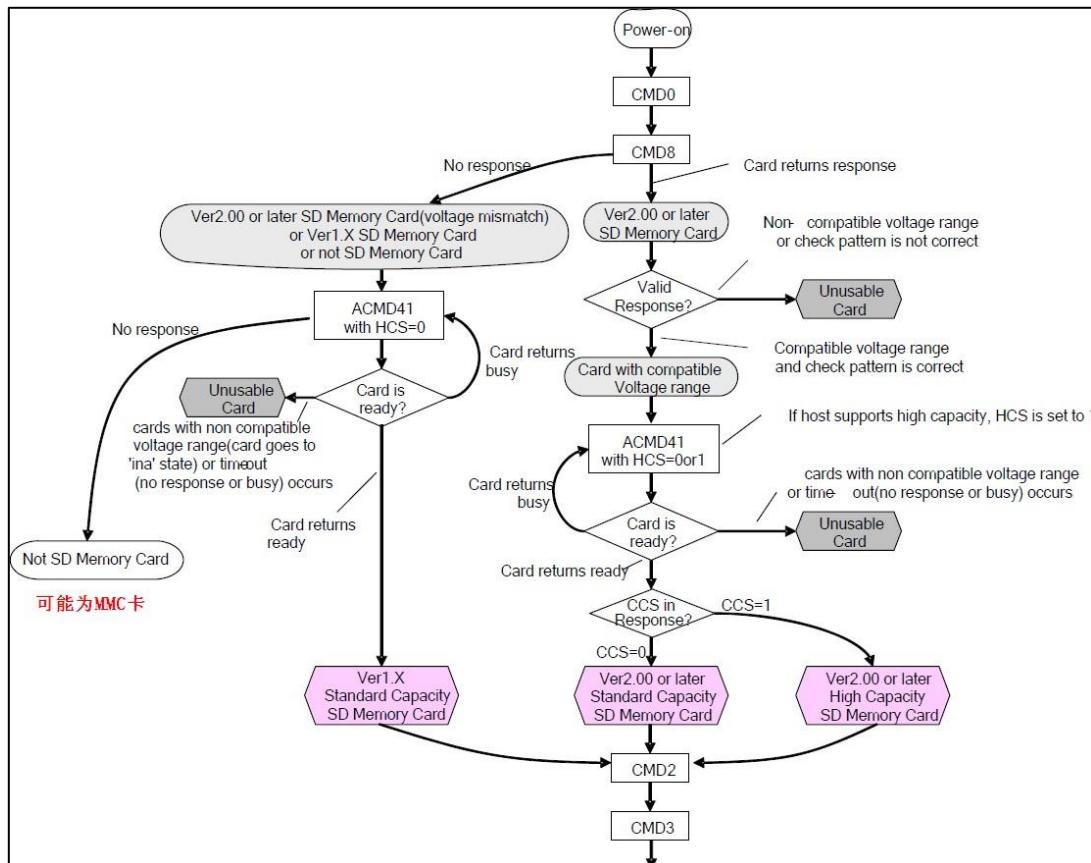


图 21.1.5.1 SD 卡初始化流程 (Card Initialization and Identification Flow (SD mode))

从图中，我们看到，不管什么卡（这里我们将卡分为 4 类：SD2.0 高容量卡（SDHC，最大 32G），SDv2.0 标准容量卡（SDSC，最大 2G），SD1.x 卡和 MMC 卡），首先我们要执行的是卡上电（需要设置 SDIO\_POWER[1:0]=11），上电后发送 CMD0，对卡进行软复位，之后发送 CMD8 命令，用于区分 SD 卡 2.0，只有 2.0 及以后的卡才支持 CMD8 命令，MMC 卡和 V1.x 的卡，是不支持该命令的。CMD8 的格式如下表所示：

位序	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
占用位	1	1	6	20	4	8	7	1
命令值	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
描述	起始位	传输位	命令索引	保留位	电源(VHS)	校验	CRC7	结束位

表 21.1.5.1 CMD8 命令格式

这里，我们需要在发送 CMD8 的时候，通过其带的参数我们可以设置 VHS 位，以告诉 SD 卡，主机的供电情况，VHS 位定义如下表所示：

Voltage Supplied	Value Definition
0000b	Not Defined
0001b	2.7-3.6V
0010b	Reserved for Low Voltage Range
0100b	Reserved
1000b	Reserved
Others	Not Defined

表 21.1.5.2 VHS 位定义

这里我们使用参数 0X1AA，即告诉 SD 卡，主机供电为 2.7~3.6V 之间，如果 SD 卡支持 CMD8，且支持该电压范围，则会通过 CMD8 的响应 (R7) 将参数部分原本返回给主机，如果不支持 CMD8，或者不支持这个电压范围，则不响应。

在发送 CMD8 后，发送 ACMD41（注意发送 ACMD41 之前要先发送 CMD55），来进一步确认卡的操作电压范围，并通过 HCS 位来告诉 SD 卡，主机是不是支持高容量卡 (SDHC)。ACMD41 的命令格式如下所示：

ACMD INDEX	type	argument	resp	abbreviation	command description
ACMD41	bcr	[31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V <sub>DD</sub> Voltage Window(OCR[23:0])	R3	SD_SEND_OP_COND	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30].

表 21.1.5.3 ACMD41 命令格式

ACMD41 得到的响应 (R3) 包含 SD 卡 OCR 寄存器内容，OCR 寄存器内容定义如下表所示：

OCR bit position	OCR Fields Definition
0-6	reserved
7	Reserved for Low Voltage Range
8-14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-29	reserved
30	Card Capacity Status (CCS) <sup>1</sup>
31	Card power up status bit (busy) <sup>2</sup>

1) This bit is valid only when the card power up status bit is set.  
2) This bit is set to LOW if the card has not finished the power up routine.

} VDD Voltage Window

表 21.1.5.4 OCR 寄存器定义

对于支持 CMD8 指令的卡，主机通过 ACMD41 的参数设置 HCS 位为 1，来告诉 SD 卡主机支持 SDHC 卡，如果设置为 0，则表示主机不支持 SDHC 卡，SDHC 卡如果接收到 HCS 为 0，则永远不会反回卡就绪状态。对于不支持 CMD8 的卡，HCS 位设置为 0 即可。

SD 卡在接收到 ACMD41 后，返回 OCR 寄存器内容，如果是 2.0 的卡，主机可以通过判断 OCR 的 CCS 位来判断是 SDHC 还是 SDSC；如果是 1.x 的卡，则忽略该位。OCR 寄存器的最后一个位用于告诉主机 SD 卡是否上电完成，如果上电完成，该位将会被置 1。

对于 MMC 卡，则不支持 ACMD41，不响应 CMD55，对 MMC 卡，我们只需要在发送 CMD0 后，在发送 CMD1（作用同 ACMD41），检查 MMC 卡的 OCR 寄存器，实现 MMC 卡的初始化。

至此，我们便实现了对 SD 卡的类型区分，图 21.1.5.1 中，最后发送了 CMD2 和 CMD3 命令，用于获得卡 CID 寄存器数据和卡相对地址（RCA）。

CMD2，用于获得 CID 寄存器的数据，CID 寄存器数据各位定义如下表所示：

Name	Field	Width	CID-slice
Manufacturer ID	MID	8	[127:120]
OEM/Application ID	OID	16	[119:104]
Product name	PNM	40	[103:64]
Product revision	PRV	8	[63:56]
Product serial number	PSN	32	[55:24]
reserved	--	4	[23:20]
Manufacturing date	MDT	12	[19:8]
CRC7 checksum	CRC	7	[7:1]
not used, always 1	-	1	[0:0]

表 21.1.5.5 卡 CID 寄存器位定义

SD 卡在收到 CMD2 后，将返回 R2 长响应（136 位），其中包含 128 位有效数据（CID 寄存器内容），存放在 SDIO\_RESP1~4 等 4 个寄存器里面。通过读取这四个寄存器，就可以获得 SD 卡的 CID 信息。

CMD3，用于设置卡相对地址（RCA，必须为非 0），对于 SD 卡（非 MMC 卡），在收到 CMD3 后，将返回一个新的 RCA 给主机，方便主机寻址。RCA 的存在允许一个 SDIO 接口挂多个 SD 卡，通过 RCA 来区分主机要操作的是哪个卡。而对于 MMC 卡，则不是由 SD 卡自动返回 RCA，而是主机主动设置 MMC 卡的 RCA，即通过 CMD3 带参数（高 16 位用于 RCA 设置），实现 RCA 设置。同样 MMC 卡也支持一个 SDIO 接口挂多个 MMC 卡，不同于 SD 卡的是所有的 RCA 都是由主机主动设置的，而 SD 卡的 RCA 则是 SD 卡发给主机的。

在获得卡 RCA 之后，我们便可以发送 CMD9（带 RCA 参数），获得 SD 卡的 CSD 寄存器内容，从 CSD 寄存器，我们可以得到 SD 卡的容量和扇区大小等十分重要的信息。CSD 寄存器我们在这里就不详细介绍了，关于 CSD 寄存器的详细介绍，请大家参考《SD 卡 2.0 协议.pdf》。

至此，我们的 SD 卡初始化基本就结束了，最后通过 CMD7 命令，选中我们要操作的 SD 卡，即可开始对 SD 卡的读写操作了，SD 卡的其他命令和参数，我们这里就不再介绍了，请大家参考《SD 卡 2.0 协议.pdf》，里面有非常详细的介绍。

## 2. SPI 模式下的 SD 卡初始化

ESP32 的 SDIO 驱动模式和 SPI 模式不兼容，二者使用时需要区分开来。《SD 卡 2.0 协议.pdf》中提供了 SD 卡的 SPI 初始化时序，我们可以按它建议的流程进行 SD 卡的初始化，如下图所示。

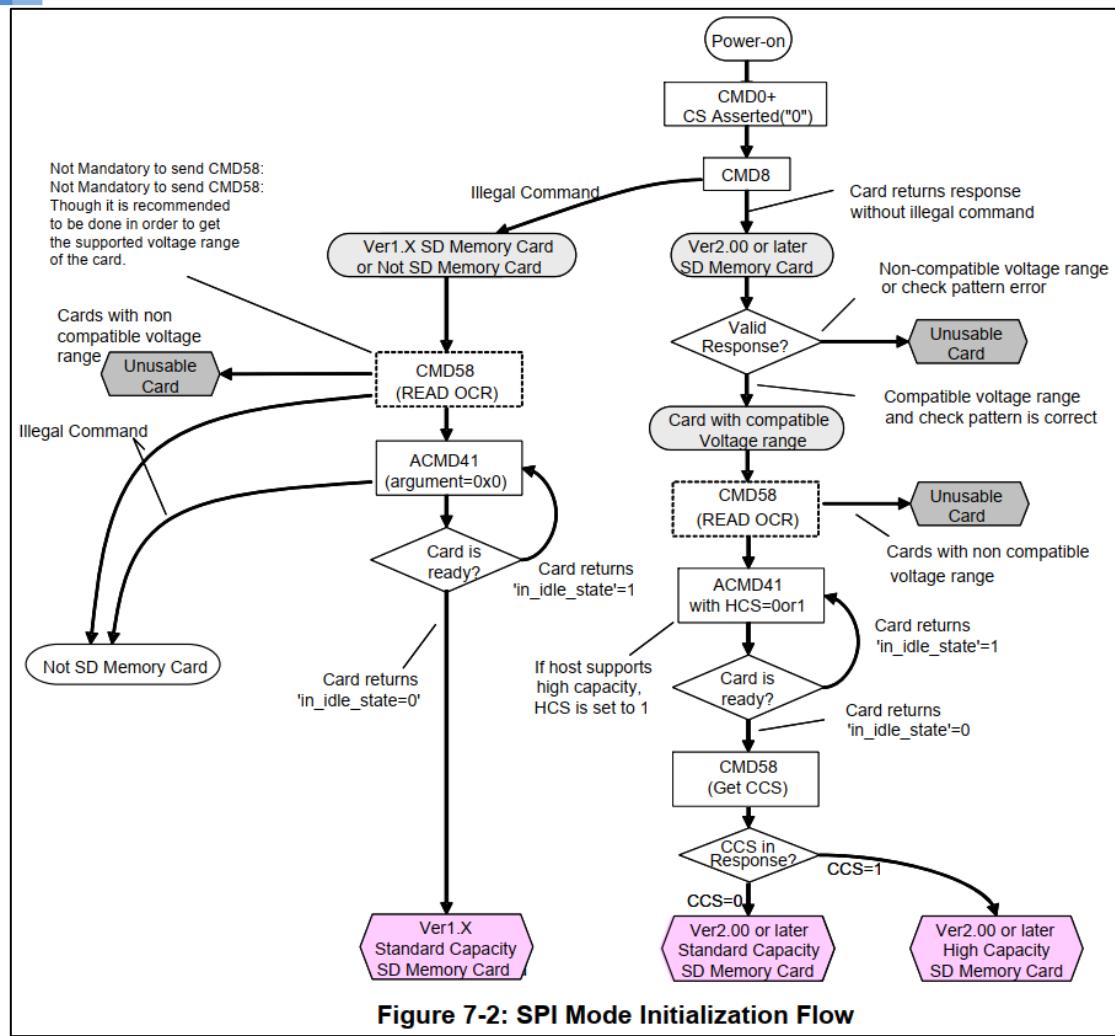


图 21.1.5.2 SD 卡的 SPI 初始化流程 (SPI Mode Initialization Flow)

要使用 SPI 模式驱动 SD 卡，先得让 SD 卡进入 SPI 模式。方法如下：在 SD 卡收到复位命令 (CMD0) 时，CS 为有效电平 (低电平) 则 SPI 模式被启用。不过在发送 CMD0 之前，要发送 >74 个时钟，这是因为 SD 卡内部有个供电电压上升时间，大概为 64 个 CLK，剩下的 10 个 CLK 用于 SD 卡同步，之后才能开始 CMD0 的操作，在卡初始化的时候，CLK 时钟最大不能超过 400Khz！

接着我们看看 SD 卡的初始化，由前面 SD 卡的基本介绍，我们知道 SD 卡是先发送数据高位的，SD 卡的典型初始化过程如下：

- 1、初始化与 SD 卡连接的硬件条件 (MCU 的 SPI 配置，IO 口配置)；
- 2、拉低片选信号，上电延时 (>74 个 CLK)；
- 3、复位卡 (CMD0)，进入 IDLE 状态；
- 4、发送 CMD8，检查是否支持 2.0 协议；
- 5、根据不同协议检查 SD 卡 (命令包括：CMD55、ACMD41、CMD58 和 CMD1 等)；
- 6、取消片选，发多 8 个 CLK，结束初始化

这样我们就完成了对 SD 卡的初始化，注意末尾发送的 8 个 CLK 是提供 SD 卡额外的时钟，完成某些操作。通过 SD 卡初始化，我们可以知道 SD 卡的类型 (V1、V2、V2HC 或者 MMC)，在完成了初始化之后，就可以开始读写数据了。

SD 卡单扇区读取数据，这里通过 CMD17 来实现，具体过程如下：

- 1、发送 CMD17；
- 2、接收卡响应 R1；
- 3、接收数据起始令牌 0XFE；

- 4、接收数据；
  - 5、接收 2 个字节的 CRC，如果不使用 CRC，这两个字节在读取后可以丢掉。
  - 6、禁止片选之后，发多 8 个 CLK；
- 以上就是一个典型的读取 SD 卡数据过程，SD 卡的写于读数据差不多，写数据通过 CMD24 来实现，具体过程如下：

- 1、发送 CMD24；
- 2、接收卡响应 R1；
- 3、发送写数据起始令牌 0XFE；
- 4、发送数据；
- 5、发送 2 字节的伪 CRC；
- 6、禁止片选之后，发多 8 个 CLK；

以上就是一个典型的写 SD 卡过程。关于 SD 卡的介绍，我们就介绍到这里，更详细的关于 SPI 操作 SD 卡方法可以参考我们《ESP32F1 Mini 板》关于 SD 卡的章节，我们这里就不再展开了

### 3.32-S3 的 SD/MMC 概述

ESP32-S3 存储卡接口控制器提供了一个访问安全数字输入输出卡、MMC 卡以及 CD-ATA 设备的硬件接口，用于连接高级外设总线和外部存储设备。该控制器支持两个外部卡，分别为：卡 0 和卡 1。所有 SD/MMC 模块接口信号都必须通过 GPIO 矩阵传输至 GPIO pad。SD/MMC 控制器支持两组外设工作，但不支持同时工作，其连接的拓扑结构如下所示：

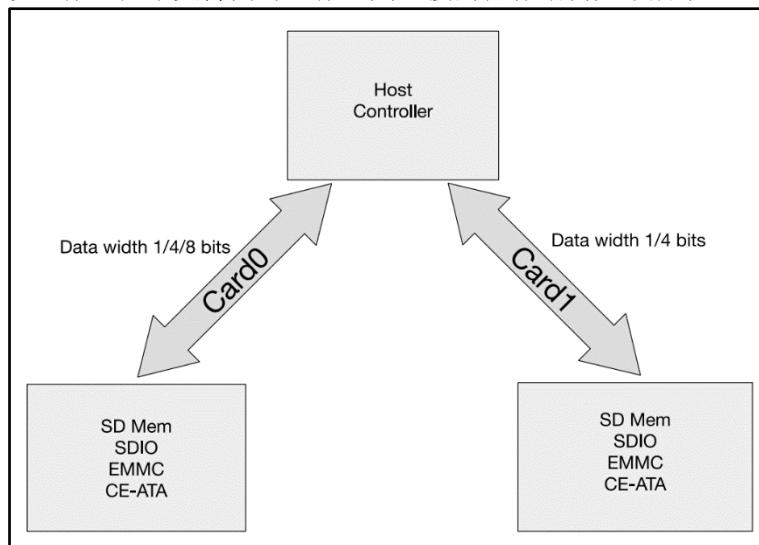


图 21.1.5.3 ESP32-S3 的 SD/MMC 控制器连接的拓扑结构

SD/MMC 的外部接口信号主要为时钟信号(sdhost\_cclk\_out\_1,eg:card1)、命令信号(sdhost\_ccmd\_out\_1)、数据信号(sdhost\_cdata\_in\_1[7:0]/sdhost\_cdata\_out\_1[7:0])，SD/MMC 控制器可通过这些外部接口信号与外部设备通信。其它信号还包括卡中断信号、卡检测信号和写保护信号等。

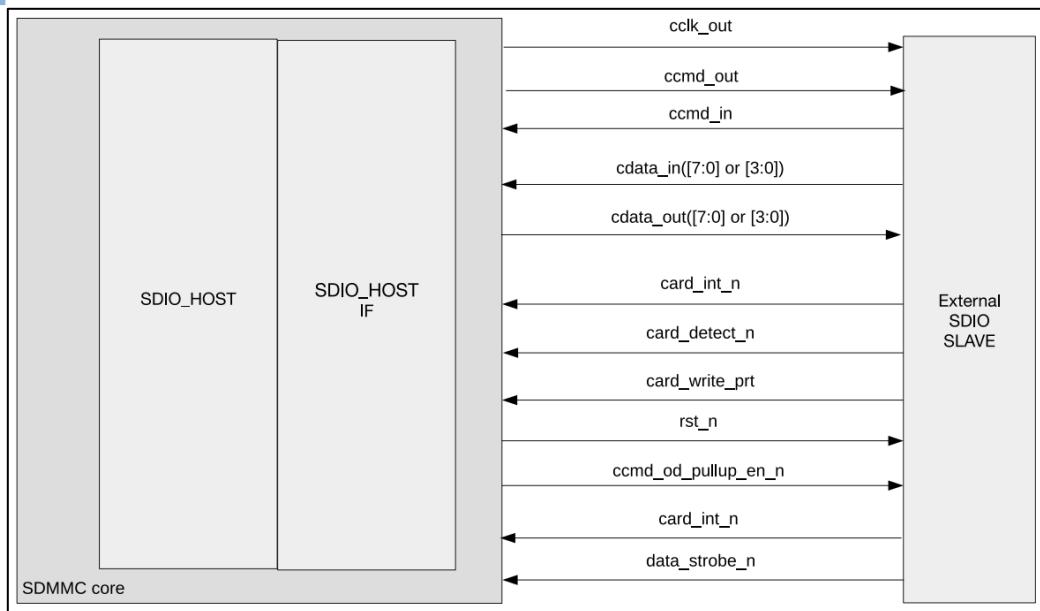


图 21.1.5.4 P32-S3 的 SD/MMC 控制器外部接口信号

管脚	方向	描述
sdhost_cclk_out	输出	向从机发送信号
sdhost_ccmd	双向	双向命令/响应线
sdhost_cdata	双向	双向数据读/写线
sdhost_card_detect_n	输入	卡检测输入线
sdhost_card_write_prt	输入	卡写保护状态输入

图 21.1.5.6 ESP32-S3 的 SD/MMC 信号描述

## 21.2 硬件设计

### 21.2.1 例程功能

本章实验功能简介：经过一系列初始化之后，通过一个 while 循环以 SD 卡初始化为条件，以检测 SD 卡是否初始化成功，若初始化 SD 卡成功，则会通过串口或者 VSCode 终端输出 SD 卡的相关参数，并在 LCD 上显示 SD 卡的总容量以及剩余容量。此时 LED 闪烁，表示程序正在运行。

### 21.2.2 硬件资源

1. LED 灯  
LED -IO0
2. 0.96 寸 LCD
3. SD  
CS-IO2  
SCK-IO12  
MOSI-IO11  
MISO-IO13

### 21.2.3 原理图

本章实验使用 SPI 接口与 SD 卡进行连接，DNESP32S3M 最小系统板板载了一个 Micro SD 卡座用于连接 SD 卡，SD 卡与 ESP32-S3 的连接原理图，如下图所示：

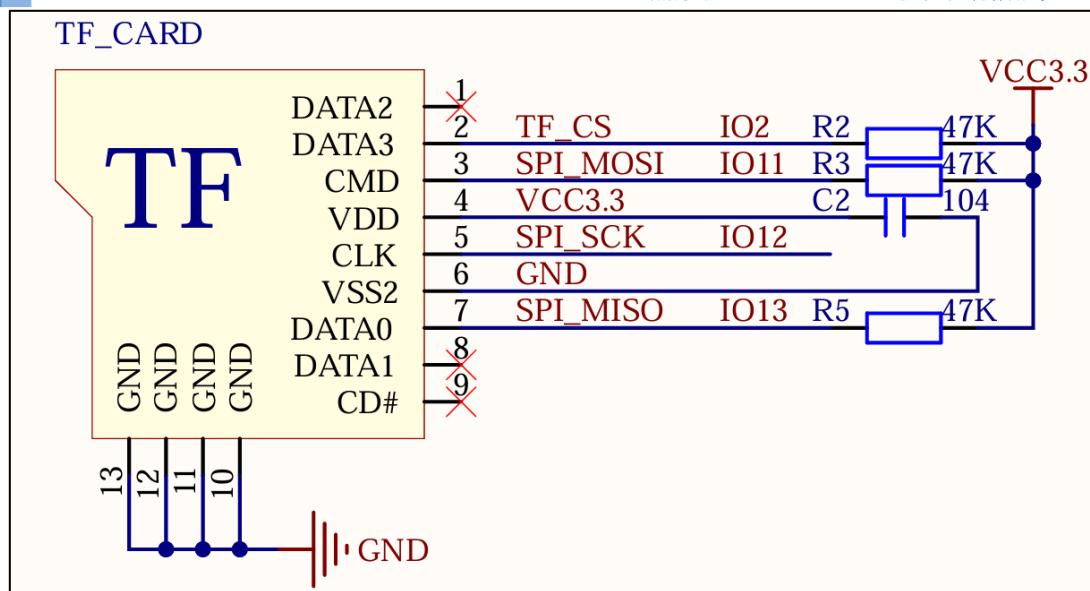


图 21.2.3.1 SD 卡接口与 ESP32-S3 的连接电路图

## 21.3 程序设计

### 21.3.1 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：

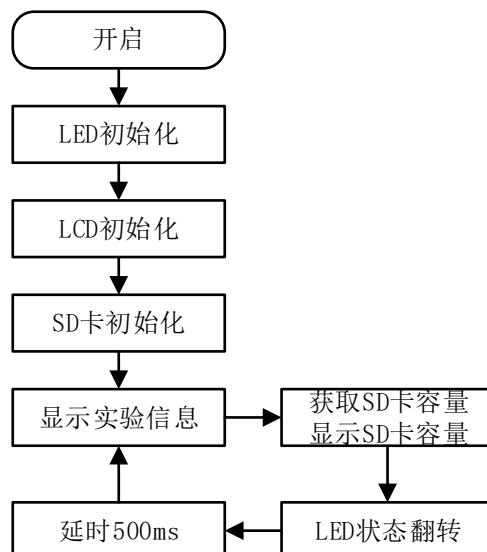


图 21.3.1.1 SD 卡实验程序流程图

### 21.3.2 SD 卡函数解析

ESP-IDF 提供了一套 API 来配置 SD 卡。要使用此功能，需要导入必要的头文件：

```
#include "driver/sdspi_host.h"
#include "driver/spi_common.h"
#include "sdmmc_cmd.h"
#include "driver/sdmmc_host.h"
#include "spi.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 SD 卡函数，这些函数的描述及其作用如下：

#### 1. 挂载 SD 卡

该函数用给定的配置，挂载 SD 卡，该函数原型如下所示：

```
esp_err_t esp_vfs_fat_sdspi_mount(const char* base_path,
                                    const sdmmc_host_t* host_config_input,
                                    const sdspi_device_config_t* slot_config,
                                    const esp_vfs_fat_mount_config_t* mount_config,
                                    sdmmc_card_t** out_card);
```

该函数的形参描述如下表所示：

参数	描述
base_path	应该注册分区的路径（例如“/sdcard”）
host_config_input	指向描述 SDMMC 主机的结构的指针。此结构可以使用 SDSPI_HOST_DEFAULT 宏初始化。
slot_config	指向具有插槽配置的结构的指针，对于 SPI 外设，将指针传递到使用 sdspi_device_config_DEFAULT 初始化的 sdspi_device_config_t 结构。
mount config	指向具有用于安装 FATFS 的额外参数的结构的指针
out_card	如果不是 NULL，指向卡片信息结构的指针将通过此参数返回。

表 21.3.2.1 esp\_vfs\_fat\_sdspi\_mount() 函数形参描述

该函数的返回值描述，如下表所示：

返回值	描述
ESP_OK	返回：0，配置成功
ESP_ERR_INVALID_STATE	如果已经调用了 esp_vfs_fat_sdmmc_mount
ESP_ERR_NO_MEM	如果无法分配内存
ESP_FAIL	如果分区无法安装，则来自 SDMMC 或 SPI 驱动程序、SDMMC 协议或 FATFS 驱动程序的其他错误代码

表 21.3.2.2 函数 esp\_vfs\_fat\_sdspi\_mount() 返回值描述

该函数使用 sdmmc\_host\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
sdmmc_host_t	flags	定义主机属性的标志：SPI 协议且可调用 deinit 函数，有如下值可以配置： SDMMC_HOST_FLAG_1BIT SDMMC_HOST_FLAG_4BIT SDMMC_HOST_FLAG_8BIT SDMMC_HOST_FLAG_SPI SDMMC_HOST_FLAG_DDR SDMMC_HOST_FLAG_DEINIT_ARG
	slot	使用 SPI2 端口，有如下值可以配置： SPI1_HOST SPI2_HOST
	max_freq_khz	主机支持的最大频率：20000
	io_voltage	控制器使用的 I/O 电压
	init	用于初始化驱动程序的主机函数
	set_bus_width	设置总线宽度的主机功能，例程设置为 NULL
	get_bus_width	取总线宽度的主机函数，例程设置为 NULL

	set_bus_ddr_mode	设置 DDR 模式的主机功能，例程设置为 NULL
	set_card_clk	设置板卡时钟频率的主机函数
	do_transaction	执行事务的主机函数
	deinit_p	用于取消初始化驱动程序的主机函数
	io_int_enable	启用 SDIO 中断线的主机功能
	io_int_wait	等待 SDIO 中断线路激活的主机功能
	command_timeout_ms	超时， 默认为 0

表 21.3.2.3 sdmmc\_host\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 esp\_vfs\_fat\_sdspi\_mount() 函数，用以实例化 SD 卡，挂载文件系统。

## 2. 取消挂载 SD 卡

该函数用于取消挂载 SD 卡，该函数原型如下所示：

```
esp_err_t esp_vfs_fat_sdcard_unmount(const char* base_path, sdmmc_card_t *card);
```

该函数的形参描述如下表所示：

参数	描述
base_path	应该注册分区的路径（例如“/sdcard”）
card	SD / MMC 卡结构

表 21.3.2.4 esp\_vfs\_fat\_sdcard\_unmount() 函数形参描述

该函数的返回值描述，如下表所示：

返回值	描述
ESP_OK	返回：0，配置成功
ESP_ERR_INVALID_ARG	如果 card 参数未注册
ESP_ERR_INVALID_STATE	如果尚未调用 esp_vfs_fat_sdmmc_mount

表 21.3.2.5 函数 esp\_vfs\_fat\_sdcard\_unmount() 返回值描述

## 21.3.3 SD 卡驱动解析

在 IDF 版的 11\_sd 例程中，作者在 11\_sd\components\BSP 路径下新增了一个 SDIO 文件夹，用于存放 spi\_sdcard.c、spi\_sdcard.h 这两个文件。其中，spi\_sdcard.h 文件负责声明 SDIO 相关的函数和变量，而 spi\_sdcard.c 文件则实现了 SDIO 的驱动代码。下面，我们将详细解析这两个文件的实现内容。

### 1. spi\_sdcard.h 文件

```
/* 引脚定义 */
#define SD_NUM_CS      GPIO_NUM_2
```

```
#define MOUNT_POINT      "/0:"
```

## 2, spi\_sdcard.c 文件

sd\_pi\_init 的设计就比较简单了，我们只需要填充 SPI 结构体的控制句柄，然后添加 SPI 总线设备并配置文件系统挂载即可，根据外设的情况，我们可以设置 SPI 的使用端口为 SPI2 端口。

```
spi_device_handle_t MY_SD_Handle;
```

```
/** @brief      SD 卡初始化
 * @param      无
 * @retval     esp_err_t
 */
esp_err_t sd_spi_init(void)
{
    esp_err_t ret = ESP_OK;

    /* SD / MMC 卡结构 */
    sdmmc_card_t *card;

    /* 挂载点/根目录 */
    const char mount_point[] = MOUNT_POINT;

    /* SPI 驱动接口配置,SPISD 卡时钟是 20-25MHz */
    spi_device_interface_config_t devcfg = {

        /* SPI 时钟 */
        .clock_speed_hz = 20 * 1000 * 1000,

        /* SPI 模式 0 */
        .mode = 0,

        /* 片选引脚 */
        .spics_io_num = SD_NUM_CS,

        /* 事务队列尺寸 7 个 */
        .queue_size = 7,
    };

    /* 添加 SPI 总线设备 */
    ret = spi_bus_add_device(SPI2_HOST, &devcfg, &MY_SD_Handle);

    /* 文件系统挂载配置 */
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {

        /* 如果挂载失败: true 会重新分区和格式化/false 不会重新分区和格式化 */
        .format_if_mount_failed = false,

        /* 打开文件最大数量 */
        .max_files = 5,

        /* 硬盘分区簇的大小 */
        .allocation_unit_size = 16 * 1024,
    };

    /* SD 卡参数配置 */
    sdmmc_host_t host = {0};

    /* 定义主机属性的标志: SPI 协议且可调用 deinit 函数 */
    host.flags = SDMMC_HOST_FLAG_SPI | SDMMC_HOST_FLAG_DEINIT_ARG;

    /* 使用 SPI2 端口 */
    host.slot = SPI2_HOST;

    /* 主机支持的最大频率: 20000 */
```

```

host.max_freq_khz = SDMMC_FREQ_DEFAULT;

/* 控制器使用的 I/O 电压 */
host.io_voltage = 3.3f;

/* 用于初始化驱动程序的主机函数 */
host.init = &sdspi_host_init;

/* 设置总线宽度的主机功能 */
host.set_bus_width = NULL;

/* 取总线宽度的主机函数 */
host.get_bus_width = NULL;

/* 设置 DDR 模式的主机功能 */
host.set_bus_ddr_mode = NULL;

/* 设置板卡时钟频率的主机函数 */
host.set_card_clk = &sdspi_host_set_card_clk;

/* 执行事务的主机函数 */
host.do_transaction = &sdspi_host_do_transaction;

/* 用于取消初始化驱动程序的主机函数 */
host.deinit_p = &sdspi_host_remove_device;

/* 启用 SDIO 中断线的主机功能 */
host.io_int_enable = &sdspi_host_io_int_enable;

/* 等待 SDIO 中断线路激活的主机功能 */
host.io_int_wait = &sdspi_host_io_int_wait;

/* 超时，默认为 0 */
host.command_timeout_ms = 0;

/* SD 卡引脚配置 */
sdspi_device_config_t slot_config = {0};
slot_config.host_id = host.slot;
slot_config.gpio_cs = SD_NUM_CS;
slot_config.gpio_cd = GPIO_NUM_NC;
slot_config.gpio_wp = GPIO_NUM_NC;
slot_config.gpio_int = GPIO_NUM_NC;

/* 挂载文件系统 */
ret = esp_vfs_fat_sdspi_mount(mount_point,
                               &host,
                               &slot_config,
                               &mount_config,
                               &card);

if (ret != ESP_OK)
{
    spi_bus_remove_device(MY_SD_Handle); /* 移除 SPI 上的 SD 卡设备 */
    return ret;
}

return ret;
}

```

接下来这个函数用于获取 SD 卡相关信息，比如容量大小，以及剩余容量，如下所示：

```

/**
 * @brief      获取 SD 卡相关信息
 * @param      out_total_bytes: 总大小
 * @param      out_free_bytes: 剩余大小
 * @retval     无

```

```

/*
void sd_get_fatfs_usage(size_t *out_total_bytes, size_t *out_free_bytes)
{
    FATFS *fs;
    size_t free_clusters;
    int res = f_getfree("0:", (size_t *)&free_clusters, &fs);
    assert(res == FR_OK);
    size_t total_sectors = (fs->n_fatent - 2) * fs->csize;
    size_t free_sectors = free_clusters * fs->csize;

    size_t sd_total = total_sectors / 1024;
    size_t sd_total_KB = sd_total * fs->ssize;
    size_t sd_free = free_sectors / 1024;
    size_t sd_free_KB = sd_free*fs->ssize;

    /* 假设总大小小于 4GiB, 对于 SPI Flash 应该为 true */
    if (out_total_bytes != NULL)
    {
        *out_total_bytes = sd_total_KB;
    }

    if (out_free_bytes != NULL)
    {
        *out_free_bytes = sd_free_KB;
    }
}

```

#### 21.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    LCD
    LED
    SDIO
    SPI)

set(include_dirs
    LCD
    LED
    SDIO
    SPI)

set(requires
    driver
    fatfs)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 SDIO 驱动以及 fatfs 依赖库需要由开发者自行添加，以确保 SDIO 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 SDIO 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 21.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{

```

```

esp_err_t ret;
size_t bytes_total, bytes_free; /* SD 卡的总空间与剩余空间 */

ret = nvs_flash_init(); /* 初始化 NVS */

if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
{
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
}

led_init(); /* 初始化 LED */
spi2_init(); /* 初始化 SPI */
lcd_init(); /* 初始化 LCD */

while (sd_spi_init()) /* 检测不到 SD 卡 */
{
    lcd_show_string(0, 0, 200, 16, 16, "SD Card Error!", RED);
    vTaskDelay(500);
    lcd_show_string(0, 20, 200, 16, 16, "Please Check! ", RED);
    vTaskDelay(500);
}

lcd_show_string(0, 0, 200, 16, 16, "SD Card OK!", RED);
lcd_show_string(0, 20, 200, 16, 16, "Total:      MB", RED);
lcd_show_string(0, 40, 200, 16, 16, "Free :      MB", RED);
sd_get_fatfs_usage(&bytes_total, &bytes_free);

lcd_show_num(60, 20, (int)bytes_total / 1024, 5, 16, BLUE);
lcd_show_num(60, 40, (int)bytes_free / 1024, 5, 16, BLUE);

while (1)
{
    LED_TOGGLE();
    vTaskDelay(500);
}
}

```

可以看到，本实验的应用代码中，通过初始化 SD 卡判断与 SD 卡的连接是否有误，SD 卡初始化成功后便通过函数 `sd_get_fatfs_usage()` 获取容量等信息，同时也在 LCD 上显示了 SD 的容量信息。

## 21.4 下载验证

程序下载到 DNESP32S3M 最小系统板后，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 21.4.1 程序运行效果图

## 第二十二章 SPIFFS 实验

上一章实验中已经成功驱动 SD 卡，并可对 SD 卡进行读写操作，但读写 SD 卡时都是直接读出或写入二进制数据，这样使用起来显得十分不方便，因此本章将介绍 SPIFFS，SPIFFS 是一个用于 SPI NOR flash 设备的嵌入式文件系统，支持磨损均衡以及文件系统一致性检查等功能。通过本章的学习，读者将学习到 SPIFFS 的基本使用。

本章分为以下几个小节：

22.1 SPIFFS 简介

22.2 硬件设计

22.3 程序设计

22.4 下载验证

### 22.1 SPIFFS 介绍

SPIFFS 是一个用于嵌入式目标上的 SPI NOR flash 设备的文件系统，并且有如下特点：

- 小目标，没有堆的少量 RAM
- 只有大范围的数据块才能被擦除
- 擦除会将块中的所有位重置为 1
- 写操作将 1 置 0
- 0 只能被擦除成 1
- 磨损均衡
- 以上几点是 SPIFFS 的特点，下面则说明了 SPIFFS 具体能做些什么：
- 专门为低 ram 使用而设计
- 使用静态大小的 ram 缓冲区，与文件的数量无关
- 类可移植操作系统接口：打开、关闭、读、写、查找、统计等
- 它可以在任何 NOR 闪存上运行，不仅是 SPI 闪存，理论上也可以在微处理器的嵌入式闪存上运行
- 多个 spiffs 配置可以在相同的目标上运行—甚至可以在相同的 SPI 闪存设备上运行
- 实现静态磨损调平（也就是 flash 的寿命维护）
- 内置文件系统一致性检查
- 高度可配置的

### 22.2 硬件设计

#### 22.2.1 例程功能

1. 在 nor flash 指定区域新建 holle.txt 文件，然后对这文件进行读写操作
2. LED 闪烁，指示程序正在运行

#### 22.2.2 硬件资源

1.LED 灯

LED -IO0

2.0.96 寸 LCD

3.SPIFFS

#### 22.2.3 原理图

本章实验使用的 SPIFFS 为软件库，因此没有对应的连接原理图。

## 22.3 程序设计

### 22.3.1 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 22.3.1.1 spiffs 实验流程图

### 22.3.2 SPIFFS 函数解析

SPIFFS 涉及到的文件并不算多，主要调用到了 C 库的函数，关于 C 库的函数我们在这里就不过多介绍了，主要介绍一下调用到 ESP32 IDF 库中的函数。

#### 1. 注册装载 SPIFFS

该函数使用给定的路径前缀将 SPIFFS 注册并装载到 VFS，其函数原型如下所示：

```
esp_err_t esp_vfs_spiffs_register(const esp_vfs_spiffs_conf_t * conf);
```

该函数的形参描述，如下表所示：

形参	描述
conf	指向 esp_vfs_spiffs_conf_t 配置结构的指针

表 22.3.2.1 函数 esp\_vfs\_spiffs\_register() 形参描述

该函数的返回值描述，如下表所示：

返回值	描述
ESP_OK	返回：0，配置成功
ESP_ERR_NO_MEM	如果无法分配对象
ESP_ERR_INVALID_STATE	如果已安装或分区已加密
ESP_ERR_NOT_FOUND	如果找不到 SPIFFS 的分区
ESP_FAIL	如果装载或格式化失败

表 22.3.2.2 函数 esp\_vfs\_spiffs\_register() 返回值描述

该函数使用 esp\_vfs\_spiffs\_conf\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
esp_vfs_spiffs_conf_t	base_path	与文件系统关联的文件路径前缀。
	partition_label	可选，要使用的 SPIFFS 分区的标签。如果设置为 NULL，则 f
	max_files	可以同时打开的最大文件数。
	format_if_mount_failed	如果为 true，则在装载失败时将格式化文件系统。

表 22.3.2.3 esp\_vfs\_spiffs\_conf\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 esp\_vfs\_spiffs\_register 函数，用以实例化 SPIFFS。

#### 2. 获取 SPIFFS 的信息

该函数用于获取 SPIFFS 的信息，其函数原型如下所示：

```
esp_err_t esp_spiffs_info(const char* partition_label,
                           size_t *total_bytes, size_t *used_bytes);
```

该函数的形参描述，如下表所示：

形参	描述
param partition_label	指向分区标签的指针，分区表名称
total_bytes	文件系统的大小
used_bytes	文件系统中当前使用的字节数

表 22.3.2.4 函数 esp\_spiffs\_info() 形参描述

返回值: ESP\_OK 表示获取成功, 其他表示获取失败。

### 3, 注销和卸载 SPIFFS

该函数从 VFS 注销和卸载 SPIFFS, 其函数原型如下所示:

```
esp_err_t esp_vfs_spiffs_unregister(const char* partition_label);
```

该函数的形参描述, 如下表所示:

形参	描述
param partition_label	指向分区表的指针, 分区表名称

表 22.3.2.5 函数 esp\_vfs\_spiffs\_unregister()形参描述

返回值: ESP\_OK 表示注销成功, 其他表示注销失败。

### 22.3.3 SPIFFS 驱动解析

在 IDF 版的 12\_spiffs 例程中, 作者在分区表中添加了 SPIFFS 的分区, 12\_spiffs\components\BSP 路径下并无新的驱动文件增加。分区表内容如下:

```
# ESP-IDF Partition Table
# Name,      Type,      SubType,      Offset,      Size,      Flags
nvs,        data,      nvs,          0x9000,     0x6000,      ,
phy_init,   data,      phy,          0xf000,     0x1000,      ,
factory,    app,       factory,      0x10000,    0x1F0000,    ,
vfs,        data,      fat,          0x200000,   0xA00000,    ,
storage,    data,      spiffs,       0xc00000,   0x400000,    ,
```

### 22.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件, 其内容如下所示:

```
set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)

set(requires
    driver
)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

该例程驱动文件与依赖库并没有新的文件添加。

### 22.3.5 实验应用代码

打开 main/main.c 文件, 该文件定义了工程入口函数, 名为 app\_main。该函数代码如下。

```
#define DEFAULT_FD_NUM          5
#define DEFAULT_MOUNT_POINT      "/spiffs"
#define WRITE_DATA                "ALIENTEK ESP32-S3\r\n"
static const char              *TAG = "spiffs";

/**
 * @brief      spiffs 初始化
 * @param      partition_label: 分区表的分区名称
 * @param      mount_point: 文件系统关联的文件路径前缀
 * @param      max_files: 可以同时打开的最大文件数
 * @retval
 */
esp_err_t spiffs_init(char *partition_label, char *mount_point, size_t max_files)
```

```
{  
    /* 配置 spiffs 文件系统各个参数 */  
    esp_vfs_spiffs_conf_t conf = {  
        .base_path = mount_point,  
        .partition_label = partition_label,  
        .max_files = max_files,  
        .format_if_mount_failed = true,  
    };  
  
    /* 使用上面定义的设置来初始化和挂载 SPIFFS 文件系统 */  
    esp_err_t ret_val = esp_vfs_spiffs_register(&conf);  
  
    /* 判断 SPIFFS 挂载及初始化是否成功 */  
    if (ret_val != ESP_OK)  
    {  
        if (ret_val == ESP_FAIL)  
        {  
            printf("Failed to mount or format filesystem\n");  
        }  
        else if (ret_val == ESP_ERR_NOT_FOUND)  
        {  
            printf("Failed to find SPIFFS partition\n");  
        }  
        else  
        {  
            printf("Failed to initialize SPIFFS(%s)\n", esp_err_to_name(ret_val));  
        }  
  
        return ESP_FAIL;  
    }  
  
    /* 打印 SPIFFS 存储信息 */  
    size_t total = 0, used = 0;  
    ret_val = esp_spiffs_info(conf.partition_label, &total, &used);  
  
    if (ret_val != ESP_OK)  
    {  
        ESP_LOGE(TAG,  
        "Failed to get SPIFFS partition information(%s)",  
        esp_err_to_name(ret_val));  
    }  
    else  
    {  
        ESP_LOGE(TAG, "Partition size: total: %d, used: %d", total, used);  
    }  
  
    return ret_val;  
}  
  
/**  
 * @brief      注销 spiffs 初始化  
 * @param      partition_label: 分区表标识  
 * @retval     无  
 */  
esp_err_t spiffs_deinit(char *partition_label)  
{  
    return esp_vfs_spiffs_unregister(partition_label);  
}  
  
/**  
 * @brief      测试 spiffs  
 * @param      无  
 * @retval     无  
 */  
void spiffs_test(void)
```

{

```
ESP_LOGI(TAG, "Opening file");
/* 建立一个名为/spiffs/hello.txt 的只写文件 */
FILE* f = fopen("/spiffs/hello.txt", "w");

if (f == NULL)
{
    ESP_LOGE(TAG, "Failed to open file for writing");
    return;
}

/* 写入字符 */
fprintf(f, WRITE_DATA);

fclose(f);
ESP_LOGI(TAG, "File written");

/* 重命名之前检查目标文件是否存在 */
struct stat st;

if (stat("/spiffs/foo.txt", &st) == 0) /* 获取文件信息，获取成功返回 0 */
{
    /* 从文件系统中删除一个名称。
       如果名称是文件的最后一个连接，并且没有其它进程将文件打开，
       名称对应的文件会实际被删除。 */
    unlink("/spiffs/foo.txt");
}

/* 重命名创建的文件 */
ESP_LOGI(TAG, "Renaming file");

if (rename("/spiffs/hello.txt", "/spiffs/foo.txt") != 0)
{
    ESP_LOGE(TAG, "Rename failed");
    return;
}

/* 打开重命名的文件并读取 */
ESP_LOGI(TAG, "Reading file");
f = fopen("/spiffs/foo.txt", "r");

if (f == NULL)
{
    ESP_LOGE(TAG, "Failed to open file for reading");
    return;
}

char line[64];
fgets(line, sizeof(line), f);
fclose(f);

char* pos = strchr(line, '\n'); /* 指针 pos 指向第一个找到'\n' */

if (pos)
{
    *pos = '\0';                  /* 将'\n'替换为'\0' */
}

ESP_LOGI(TAG, "Read from file: '%s'", line);
lcd_show_string(90, 110, 200, 16, 16, line, RED);
}
```

在 SPIFFS 驱动中，首先初始化并挂载了一个 SPIFFS 分区，然后使用 POSIX 和 C 库 API 写入和读取数据。

```
/**  
 * @brief      程序入口  
 * @param      无  
 * @retval     无  
 */  
void app_main(void)  
{  
    esp_err_t ret;  
    ret = nvs_flash_init();  
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret==ESP_ERR_NVS_NEW_VERSION_FOUND)  
    {  
        ESP_ERROR_CHECK(nvs_flash_erase());  
        ret = nvs_flash_init();  
    }  
    ESP_ERROR_CHECK(ret);  
    led_init();  
    spi2_init();  
    lcd_init();  
    spiffs_init("storage", DEFAULT_MOUNT_POINT, DEFAULT_FD_NUM); /*SPIFFS 初始化*/  
    /* 显示实验信息 */  
    lcd_show_string(10, 70, 200, 16, 16, "SPIFFS TEST", RED);  
    lcd_show_string(10, 110, 200, 16, 16, "Read file:", BLUE);  
    spiffs_test();  
    while (1)  
    {  
        LED_TOGGLE();  
        vTaskDelay(500);  
    }  
}
```

可以看到，本实验的应用代码中，在一系列初始化之后，配置 spiffs 文件系统各个参数，再建立一个名为/spiffs/hello.txt 的只写文件，LED 闪烁表明程序正在运行。

## 22.4 下载验证

在完成编译和烧录操作后，在指定区域新建 hello.txt 文件，然后对这文件进行读写操作。

## 第二十三章 汉字显示实验

本章，我们将介绍如何使用 ESP32 控制 SPILCD 显示汉字。在本章中，我们将使用通过 SD 卡更新字库。ESP32 读取存在 SD 卡里面的字库，然后将汉字显示在 LCD 上面。

本章分为以下几个小节：

- 23.1 汉字显示简介
- 23.2 硬件设计
- 23.3 程序设计
- 23.4 下载验证

### 23.1 汉字显示原理简介

汉字的显示和 ASCII 显示其实是一样的原理，如下图所示：

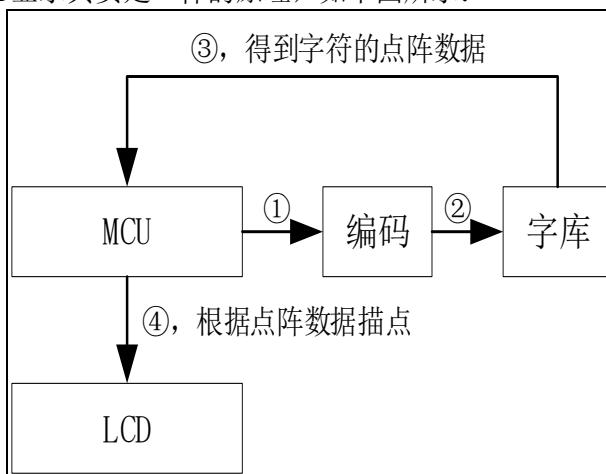


图 23.1.1 单个汉字显示原理框图

上图显示了单个汉字显示的原理框图，单片机（MCU）先根据汉字编码（①，②）从字库里面找到该汉字的点阵数据（③），然后通过描点函数，按字库取模方式，将点阵数据在 LCD 上画出来（④），就可以实现一个汉字的显示。

接下来，重点介绍一下汉字的：编码、字库及显示等相关知识。

#### 23.1.1 字符编码介绍

单片机只能识别 0 和 1，所有信息都是以 0 和 1 的形式存储的，单片机本身并不能识别字符，所以我们需要对字符进行编码（也叫内码，特定的编码对应特定的字符），单片机通过编码来识别具体的汉字。常见的字符集编码如下表所示：

字符集	编码长度	说明
ASCII	1 个字节	拉丁字母编码，仅 128 个编码，最简单
GB2312	2 个字节	简体中文字符编码，包含约 6000 多汉字编码
GBK	2 个字节	对 GB2312 的扩充，支持繁体中文，约 2W 多汉字编码
BIG5	2 个字节	繁体中文字符编码，在台湾、香港用的多
UNICODE	一般 2 个字节	国际标准编码，支持各国文字

表 23.1.1.1 常见字符集编码

其中 ASCII 编码最简单，采用单字节编码，在前面的 OLED 和 LCD 实验，我们已经有所接触。ASCII 是基于拉丁字母的一套电脑编码系统，仅包括 128 个编码，其中 95 个显示字符，使用一个字节即可编码完所有字符，我们常见的英文字母和数字，就是使用 ASCII 字符编码，另外 ASCII 字符显示所占宽度为汉字宽度的一半！也可以理解成：ASCII 字符的宽度 = 高度的一半。

GB2312、GBK 和 BIG5 都是汉字编码，GBK 码是 GB2312 的扩充，是国内计算机系统默认的汉字编码，而 BIG5 则是繁体汉字字符集编码，在香港和台湾的计算机系统汉字编码一般默认使用 BIG5 编码。一般来说，汉字显示所占的宽度等于高度，即宽度和高度相等。

UNICODE 是国际标准编码，支持各国文字，一般是 2 字节编码（也可以是 3 字节），这里不做讨论。想详细了解的可以执行百度学习。

接下来，我们重点介绍一下 GBK 编码。

GBK 是一套汉字编码规则，采用双字节编码，共 23940 个码位，收录汉字和图形符号 21886 个，其中汉字（含繁体字和构件）21003 个，图形符号 883 个。

每个 GBK 码由 2 个字节组成，第一个字节范围：0X81~0XFE，第二个字节分为两部分，一是：0X40~0X7E，二是：0X80~0XFE。其中与 GB2312 相同的区域，字完全相同。GBK 编码规则如表 23.1.1.2 所示：

字节	范围	说明
第一字节 (高)	0X81~0XFE	共 126 个区（不包括 0X00~0X80，以及 0xFF）
第二字节 (低)	0X40~0X7E	63 个编码（不包括 0X00~0X39，以及 0X7F）
	0X80~0XFE	127 个编码（不包括 0xFF）

表 23.1.1.2 GBK 编码规则

我们把第一个字节（高字节）代表的意义称为区，那么 GBK 里面总共有 126 个区（0XFE - 0X81 + 1），每个区内有 190 个汉字（0XFE - 0X80 + 0X7E - 0X40 + 2），总共就有  $126 * 190 = 23940$  个汉字。

第一个编码：0X8140，对应汉字：乚；

第二个编码：0X8141，对应汉字：乚；

第三个编码：0X8142，对应汉字：乚；

第四个编码：0X8143，对应汉字：乚；

依次对所有汉字进行编码，详见：[www.qqxiuzi.cn/zh/hanzi-gbk-bianma.php](http://www.qqxiuzi.cn/zh/hanzi-gbk-bianma.php)。

### 23.1.2 汉字字库简介

光有汉字编码，单片机还是无法在 LCD 上显示这个汉字的，必须有对应汉字编码的点阵数据，才可以通过描点的方式，将汉字显示在 LCD 上。所有汉字点阵数据的集合，就叫做汉字字库。而不同大小的汉字，其字库大小也不一样，因此又有不同大小汉字的字库（如：12\*12 汉字字库、16\*16 汉字字库、24\*24 汉字字库等）。

单个汉字的点阵数据，也称之为字模。汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画。为了方便取模和描点，我们一般规定一个取模方向，当取模和描点都按取模方向来操作，就可以实现一个汉字的点阵数据提取和显示。

以 12\*12 大小的“好”字为例，假设我们规定取模方向为：从上到下，从左到右，且高位在前，则其取模原理如下图所示：

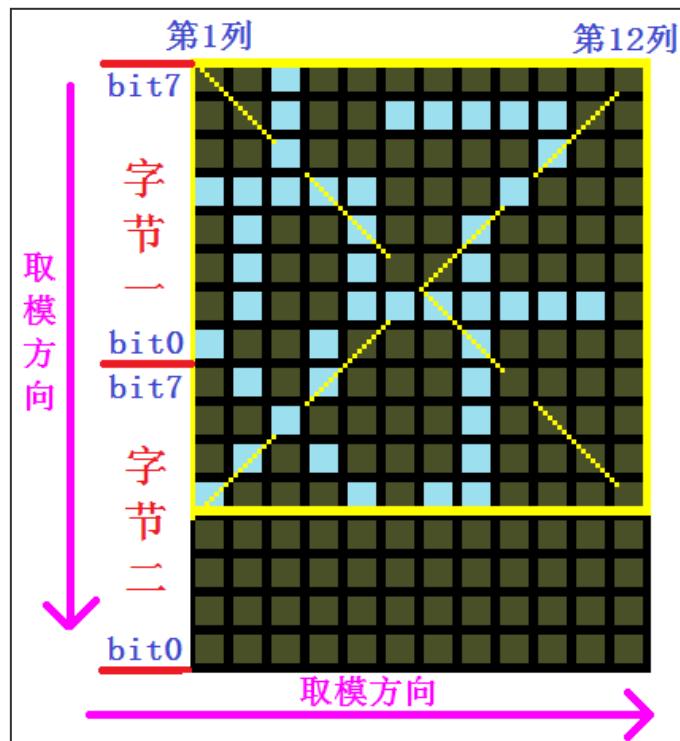


图 23.1.2.1 从上到下，从左到右取模原理

图中，我们取模的时候，从最左上方的点开始取（从上到下，从左到右），且高位在前（bit7 在表示第一个位），那么：

第一个字节是：0X11（1，表示浅蓝色的点，即要画出来的点，0 则表示不要画出来）；

第二个字节是：0X10；

第三个字节是：0X1E（到第二列了，每列 2 个字节）；

第四个字节是：0XA0；

以此类推，共 12 列，每列 2 个字节，总共 24 字节，12\*12 “好” 字完整的字模如下：

```
uint8_t hzm_1212[24]={  
0x11,0x10,0x1E,0xA0,0xF0,0x40,0x11,0xA0,0x1E,0x10,0x42,0x00,  
0x42,0x10,0x4F,0xF0,0x52,0x00,0x62,0x00,0x02,0x00,0x00,0x00}; /* 好字字模 */
```

在显示的时候，我们只需要读取这个汉字的点阵数据（12\*12 字体，一个汉字的点阵数据为 24 个字节），然后将这些数据，按取模方式，反向解析出来（坐标要处理好），每个字节，是 1 的位，就画出来，不是 1 的位，就忽略，这样，就可以显示出这个汉字了。

知道显示一个汉字的原理，就可以推及整个汉字库了，要显示任意汉字，我们首先要知道该汉字的点阵数据，整个 GBK 字库是比较大的（2W 多个汉字），这些数据可以由专门的软件来生成。

### 字库的制作

字库的制作，我们要用到一款软件，由正点原子设计的字模生成软件。该软件可以在 Windows 2000/XP/7/8/10 等操作系统下生成任意点阵大小的 ASCII、GB2312（简体中文）、GBK（简繁体中文）、BIG5（繁体中文）、HANGUL(韩文)、SJIS(日文)、Unicode 以及泰文，越南文、俄文、乌克兰文，拉丁文，8859 系列等共二十几种编码的字库，不但支持生成二进制文件格式的文件，也可以生成 BDF 文件，还支持生成图片功能，并支持横向，纵向等多种扫描方式，且扫描方式可以根据用户的需求进行增加。软件主界面如图 39.1.2.2 所示：

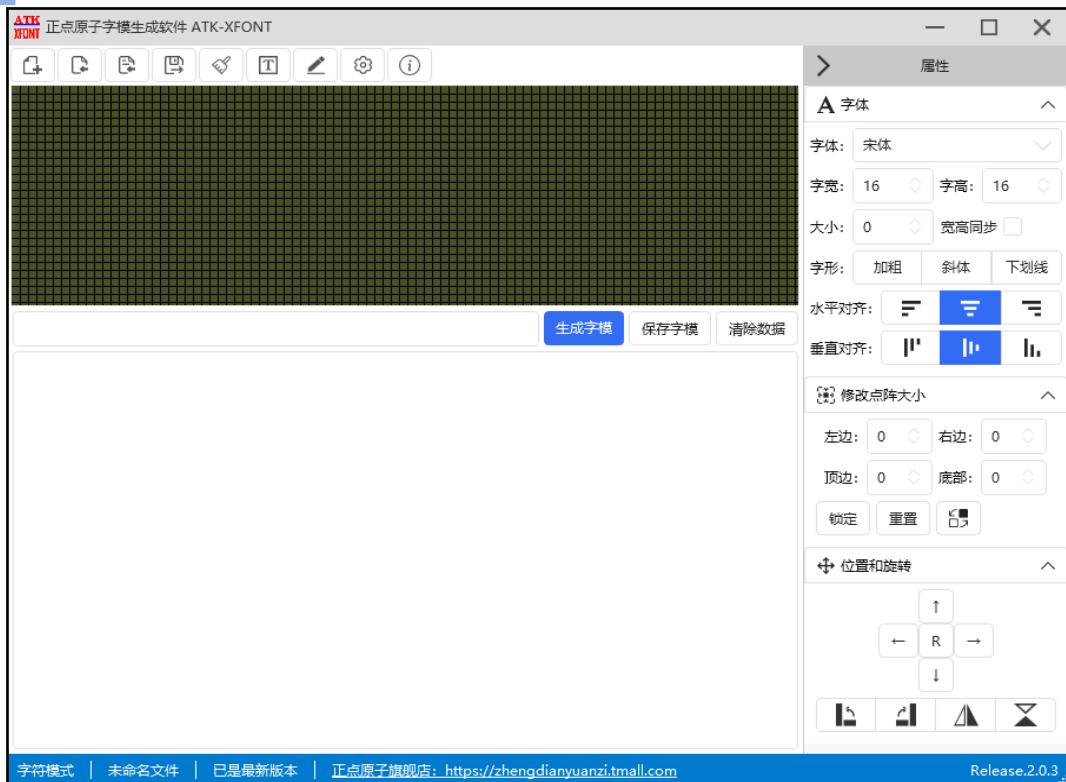


图 23.1.2.2 点阵字库生成器默认界面

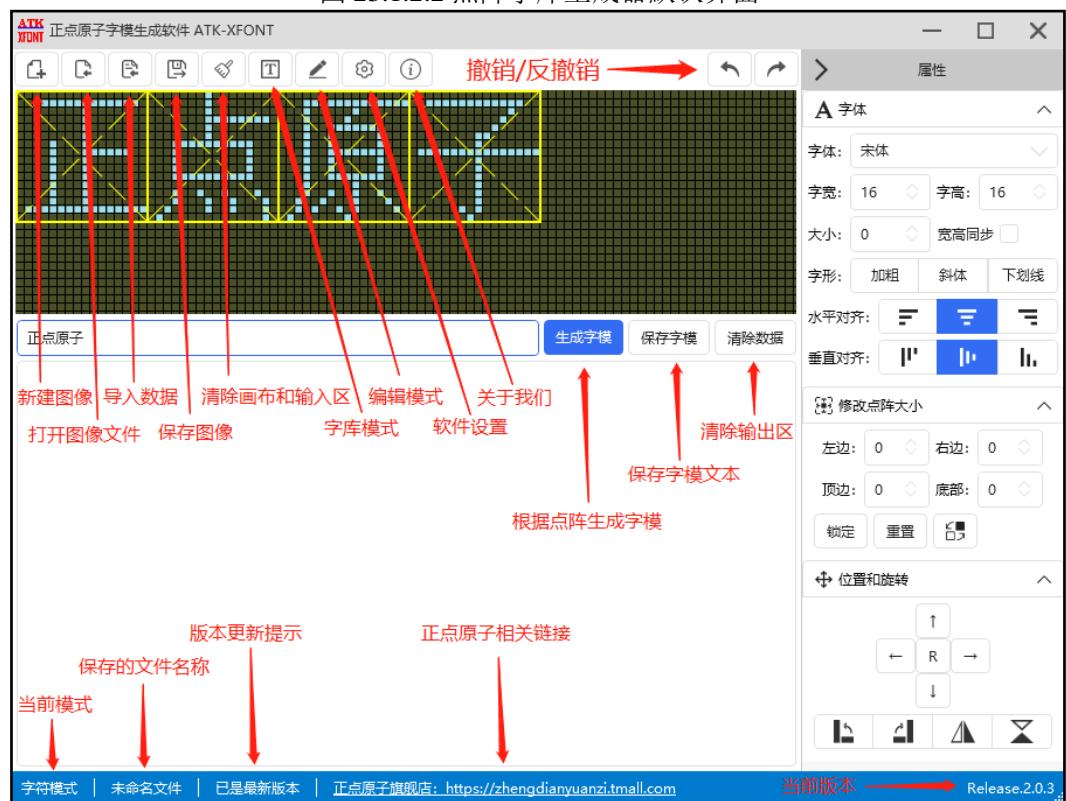


图 23.1.2.3 生成 GBK16\*16 字库的设置方法

注意：电脑端的字体大小与我们生成点阵大小的关系为：

$$\text{fsize} = \text{dsize} * 6 / 8$$

其中，fsize 是电脑端字体的大小，dsize 是点阵大小（12、16、24 等）。所以 16\*16 点阵大小对应的是 12 号字体。

生成完以后，我们把文件名和后缀改成：GBK16.FON（这里是手动修改后缀!!）。用类似的方法，生成 12\*12 的点阵库（GBK12.FON）和 24\*24 的点阵库（GBK24.FON），总共制作 3 个字库。

另外，该软件还可以生成其他很多字库，字体也可选，大家可以根据自己的需要按照上面的方法生成即可。该软件的详细介绍请看《ATK-XFONT 软件用户手册》。

最后，我们生成的字库，要先放入 TF 卡，然后 TF 卡中的字库文件通过分区表进行加载，我们在分区表（partitions-16MiB）中定义了一个分区专用于 SPIFFS 以及存储。使用的时候，分区表读取传入的字库存放的起始地址。

### 23.1.3 汉字显示原理

经过以上两个小节的学习，我们可以归纳出汉字显示的过程：

MCU → 汉字编码 → 汉字字库 → 汉字点阵数据 → 描点

编码和字库的制作我们已经学会了，所以只剩下一个问题：如何通过汉字编码在汉字字库里面查找对应汉字的点阵数据？

根据 GBK 编码规则，我们的汉字点阵字库只要按照这个编码规则从 0X8140 开始，逐一建立，每个区的点阵大小为每个汉字所用的字节数\*190。这样，我们就可以得到在这个字库里定位汉字的方法：

当  $GBKL < 0X7F$  时： $Hp = ((GBKH - 0x81) * 190 + GBKL - 0X40) * csize;$

当  $GBKL > 0X80$  时： $Hp = ((GBKH - 0x81) * 190 + GBKL - 0X41) * csize;$

其中 GBKH、GBKL 分别代表 GBK 的第一个字节和第二个字节(也就是高字节和低字节)，csize 代表单个汉字点阵数据的大小（字节数），Hp 则为对应汉字点阵数据在字库里面的起始地址(假设是从 0 开始存放，如果是非 0 开始，则加上对应偏移量即可)。

单个汉字点阵数据大小（csize）计算公式如下：

$$csize = (size / 8 + ((size \% 8) ? 1 : 0)) * (size);$$

其中 size 为汉字点阵长宽尺寸，如：12（对应 12\*12 字体）、16（对应 16\*16 字体）、24（对应 24\*24 字体）。对于 12\*12 字体，csize 大小为 24 字节，对于 16\*16 字体，csize 大小为 32 字节。

通过以上方法，从字库里面获取到某个汉字点阵数据后，按取模方式（我们使用：从上到下、从左到右，高位在前）进行描点还原即可将汉字显示在 LCD 上面。这就是汉字显示的原理。

## 23.2 硬件设计

### 23.2.1 例程功能

本实验开机的时候程序通过预设值的标记位检测分区表中是否已经存在字库，如果存在，则按次序显示汉字（三种字体都显示）。如果没有，则检测 SD 卡和文件系统，并查找 SYSTEM 文件夹下的 FONT 文件夹，在该文件夹内查找 UNIGBK.BIN、GBK12.FON、GBK16.FON 和 GBK24.FON 这几个文件的由来，我们在前面已经介绍过了。在检测到这些文件之后，就开始更新字库，更新完毕才开始显示汉字。通过按按键 KEY0，可以强制更新字库。

LED 闪烁，提示程序运行。

### 23.2.2 硬件资源

1. LED 灯  
LED -IO0
2. 0.96 寸 LCD  
RST- IO1\_2 (XL9555)
3. SD  
CS-IO2  
SCK-IO12  
MOSI-IO11

### 23.2.3 原理图

本章实验使用的字库管理库为软件库，因此没有对应的连接原理图。

## 23.3 程序设计

### 23.3.1 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：

### 23.3.2 汉字显示函数解析

正点原子提供的字库管理库包含了六个文件，分别为：convert.c、convert.h、fonts.c、fonts.h、text.c 和 text.h，本章实验配套实验例程中已经提供了这六个文件，并且已经针对正点原子 ESP32-S3 软硬件进行了移植适配，用户在使用时，仅需将这六个文件添加到自己的工程中即可，如下图所示：

```
./28_chinese_display/main/APP
|-- convert.c
|-- convert.h
|-- fonts.c
|-- fonts.h
|-- text.c
`-- text.h
```

图 23.3.2.1 正点原子字库文件

字库管理库中 fonts.c 和 fonts.h 两个文件提供了字库更新和初始化的函数，test.c 和 test.h 文件中提供了在 LCD 上显示中文字符的函数，convert.c 和 convert.h 提供了 UTF8 与 GBK 互转代码。

字库管理库在显示中文字符至 LCD 上时会使用 SD 卡或者分区表中的中文字库，因此需要确保这两者中的中文字库无误，若分区表中没有中文字库的数据，那么在进行字库初始化时就会提示失败，这时就需要使用字库管理库中提供的字库更新函数更新分区表中的中文字库数据，更新字库是读取 SD 卡中的字库文件将其写入分区表，因此需确保 SD 卡中有对应的中文字库文件。本章实验所需的中文字库文件可在 A 盘→5，SD 卡根目录文件→SYSTEM→FONT 中找到，建议将 A 盘→5，SD 卡根目录文件中的所有文件按照该目录的目录结构复制到 SD 卡中，方便后续实验的使用。

### 23.3.3 汉字显示驱动解析

在 IDF 版的 13\_chinese\_display 例程中，作者在 13\_chinese\_display\components\BSP 路径下并未添加新的内容，而是在 28\_chinese\_display\main\APP 路径下面，新增了一个 APP 文件，我们将详细解析这四个文件的实现内容。

#### 1, font.h 文件

```
/* 字库信息结构体定义
 * 用来保存字库基本信息，地址，大小等
 */
__packed typedef struct
{
    uint8_t  fontok;          /* 字库存在标志，0XAA，字库正常；其他，字库不存在 */
    uint32_t ugbkaddr;        /* unigbk 的地址 */
    uint32_t ugbksize;        /* unigbk 的大小 */
```

```

    uint32_t f12addr;           /* gbk12 地址 */
    uint32_t gbk12size;         /* gbk12 的大小 */
    uint32_t f16addr;           /* gbk16 地址 */
    uint32_t gbk16size;         /* gbk16 的大小 */
    uint32_t f24addr;           /* gbk24 地址 */
    uint32_t gbk24size;         /* gbk24 的大小 */
} _font_info;

```

这个结构体占用 33 字节，用于记录字库的地址和大小等信息。其中，首字节指示字库状态，其余字节记录地址和文件大小。用户字库等文件存储在分区表后的 12M 存储 storage 分区。前 33 字节保留给 \_font\_info，以保存其结构体数据；随后是 UNIGBK.BIN、GBK12.FON、GBK16.FON 和 GBK24.FON 文件。

## 2, font.c 文件

字库初始化函数也是利用其存储顺序，进行检查字库，其定义如下：

```

/**
 * @brief      初始化字体
 * @param      无
 * @retval     0, 字库完好；其他，字库丢失；
 */
uint8_t fonts_init(void)
{
    uint8_t t = 0;

    storage_partition = esp_partition_find_first(ESP_PARTITION_TYPE_DATA,
                                                ESP_PARTITION_SUBTYPE_ANY,
                                                "storage");

    if (storage_partition == NULL)
    {
        ESP_LOGE(TAG, "Flash partition not found.");
        return 1;
    }

    while (t < 10) /* 连续读取 10 次，都是错误，说明确实是存在问题，得更新字库了 */
    {
        t++;

        /* 读出 ftinfo 结构体数据 */
        fonts_partition_read((uint8_t *)&ftinfo,
                            FONTINFOADDR,
                            sizeof(ftinfo));

        if (ftinfo.fontok == 0XAA)
        {
            break;
        }

        vTaskDelay(20);
    }

    if (ftinfo.fontok != 0XAA)
    {
        return 1;
    }
    return 0;
}

```

这里就是把分区表的 12M 地址的 33 个字节数据读取出来，进而判断字库结构体 ftinfo 的字库标记 fontok 是否为 AA，确定字库是否完好。

有人会有疑问：ftinfo.fontok 是在哪里赋值 AA 呢？肯定是字库更新完毕后，给该标记赋值的，那下面就来看一下是不是这样子，字库更新函数定义如下：

```
/**
```

```
* @brief      更新字体文件
* @note       所有字库一起更新(UNIGBK,GBK12,GBK16,GBK24)
* @param      x, y      : 提示信息的显示地址
* @param      size     : 提示信息字体大小
* @param      src      : 字库来源磁盘
* @arg        "0:", SD 卡;
* @Arg        "1:", FLASH 盘
* @param      color    : 字体颜色
* @retval     0, 成功; 其他, 错误代码;
*/
uint8_t fonts_update_font(uint16_t x,
                           uint16_t y,
                           uint8_t size,
                           uint8_t *src,
                           uint16_t color)
{
    uint8_t *pname;
    uint32_t *buf;
    uint8_t res = 0;
    uint16_t i, j;
    FIL *fftemp;
    uint8_t rval = 0;
    res = 0xFF;
    ftinfo.fontok = 0xFF;
    pname = malloc(100); /* 申请 100 字节内存 */
    buf = malloc(4096); /* 申请 4K 字节内存 */
    fftemp = (FIL *)malloc(sizeof(FIL)); /* 分配内存 */

    if (buf == NULL || pname == NULL || fftemp == NULL)
    {
        free(fftemp);
        free(pname);
        free(buf);
        return 5; /* 内存申请失败 */
    }

    for (i = 0; i < 4; i++) /* 先查找文件UNIGBK,GBK12,GBK16,GBK24是否正常 */
    {
        strcpy((char *)pname, (char *)src); /* copy src 内容到 pname */
        strcat((char *)pname, (char *)FONT_GBK_PATH[i]); /* 追加具体文件路径 */
        res = f_open(fftemp, (const TCHAR *)pname, FA_READ); /* 尝试打开 */

        if (res)
        {
            rval |= 1 << 7; /* 标记打开文件失败 */
            break; /* 出错了,直接退出 */
        }
    }

    free(fftemp); /* 释放内存 */

    if (rval == 0) /* 字库文件都存在. */
    {
        /* 提示正在擦除扇区 */
        lcd_show_string(x, y, 240, 320, size, "Erasing sectors... ", color);

        /* 先擦除字库区域,提高写入速度 */
        for (i = 0; i < FONTSECSIZE; i++)
        {
            /* 进度显示 */
            fonts_progress_show(x+20*size/2, y, size, FONTSECSIZE, i, color);
        }

        /* 读出整个扇区的内容 */
    }
}
```

```

fonts_partition_read((uint8_t *)buf,
                     ((FONTINFOADDR/4096)+i)*4096,
                     4096);

for (j = 0; j < 1024; j++) /* 校验数据 */
{
    if (buf[j] != 0xFFFFFFFF) break; /* 需要擦除 */
}

if (j != 1024)
{
    /* 需要擦除的扇区 */
    fonts_partition_erase_sector(((FONTINFOADDR / 4096) + i)*4096);
}
}

for (i = 0; i < 4; i++) /* 依次更新 UNIGBK,GBK12,GBK16,GBK24 */
{
    lcd_show_string(x,y,240,320,size,FONT_UPDATE_REMIND_TBL[i],color);
    strcpy((char *)pname, (char *)src); /* copy src 内容到 pname */
    strcat((char *)pname, (char *)FONT_GBK_PATH[i]);/* 追加具体文件路径 */

    /* 更新字库 */
    res = fonts_update_fontx(x + 20 * size / 2,y,size,pname,i,color);

    if (res)
    {
        free(buf);
        free(pname);
        return 1 + i;
    }
}

/* 全部更新好了 */
ftinfo.fontok = 0XAA;

/* 保存字库信息 */
fonts_partition_write((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo));
}

free(pname); /* 释放内存 */
free(buf);
return rval; /* 无错误 */
}

```

函数的实现：动态申请内存→尝试打开文件(UNIGBK、GBK12、GBK16 和 GBK24)，确定文件是否存在→擦除字库→依次更新 UNIGBK、GBK12、GBK16 和 GBK24→写入 ftinfo 结构体信息。

在字库更新函数中能直接看到的是 ftinfo.fontok 成员被赋值，而其他成员在单个字库更新函数中被赋值，接下来分析一下更新某个字库函数，其代码如下：

```

/**
 * @brief      更新某一个字库
 * @param      x, y      : 提示信息的显示地址
 * @param      size       : 提示信息字体大小
 * @param      fpath     : 字体路径
 * @param      fx        : 更新的内容
 * @arg          0, ungbk;
 * @Arg         1, gbk12;
 * @arg         2, gbk16;
 * @arg         3, gbk24;
 * @param      color     : 字体颜色
 * @retval      0, 成功; 其他, 错误代码;
 */

```

```
static uint8_t fonts_update_fontx(uint16_t x, uint16_t y, uint8_t size,
                                  uint8_t *fpath, uint8_t fx, uint16_t color)
{
    uint32_t flashaddr = 0;
    FIL *fftemp;
    uint8_t *tempbuf;
    uint8_t res;
    uint16_t bread;
    uint32_t offx = 0;
    uint8_t rval = 0;
    ftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 分配内存 */
    if (fftemp == NULL) rval = 1;

    tempbuf = mymalloc(SRAMIN, 4096); /* 分配 4096 个字节空间 */

    if (tempbuf == NULL) rval = 1;

    res = f_open(ftemp, (const TCHAR *)fpath, FA_READ);

    if (res) rval = 2; /* 打开文件失败 */

    if (rval == 0)
    {
        switch (fx)
        {
            case 0: /* 更新 UNIGBK.BIN */
                /* 信息头之后，紧跟 UNIGBK 转换码表 */
                ftinfo.ugbkaddr = FONTINFOADDR + sizeof(ftinfo);
                ftinfo.ugbksize = ftemp->obj.objsize; /* UNIGBK 大小 */
                flashaddr = ftinfo.ugbkaddr;
                break;

            case 1: /* 更新 GBK12.FONT */
                /* UNIGBK 之后，紧跟 GBK12 字库 */
                ftinfo.f12addr = ftinfo.ugbkaddr + ftinfo.ugbksize;
                ftinfo.gbk12size = ftemp->obj.objsize; /* GBK12 字库大小 */
                flashaddr = ftinfo.f12addr; /* GBK12 的起始地址 */
                break;

            case 2: /* 更新 GBK16.FONT */
                /* GBK12 之后，紧跟 GBK16 字库 */
                ftinfo.f16addr = ftinfo.f12addr + ftinfo.gbk12size;
                ftinfo.gbk16size = ftemp->obj.objsize; /* GBK16 字库大小 */
                flashaddr = ftinfo.f16addr; /* GBK16 的起始地址 */
                break;

            case 3: /* 更新 GBK24.FONT */
                /* GBK16 之后，紧跟 GBK24 字库 */
                ftinfo.f24addr = ftinfo.f16addr + ftinfo.gbk16size;
                ftinfo.gbk24size = ftemp->obj.objsize; /* GBK24 字库大小 */
                flashaddr = ftinfo.f24addr; /* GBK24 的起始地址 */
                break;
        }
    }

    while (res == FR_OK) /* 死循环执行 */
    {
        res = f_read(ftemp, tempbuf, 4096, (UINT *)&bread); /* 读取数据 */
        if (res != FR_OK) break; /* 执行错误 */

        /* 从 0 开始写入 bread 个数据 */
        fonts_partition_write(tempbuf, offx + flashaddr, bread);

        offx += bread;
    }
}
```

```

        fonts_progress_show(x,y,size,fftemp->obj.objsize,offx,color); /*进度显示*/
        if (bread != 4096)break; /* 读完了. */
    }

    f_close(ftemp);
}

free(ftemp); /* 释放内存 */
free(tempbuf); /* 释放内存 */
return res;
}

```

单个字库更新函数，主要是对把字库从 SD 卡中读取出数据，写入分区表。同时把字库大小和起始地址保存在 ftinfo 结构体里，在前面的整个字库更新函数中使用函数：

```

/*保存字库信息*/
fonts_partition_write((uint8_t *)&ftinfo,FONTINFOADDR,sizeof(ftinfo));

```

结构体的所有成员一并写入到那 33 字节。有了这个字库信息结构体，就能很容易进行定位。结合前面的说到的根据地址偏移寻找汉字的点阵数据，我们就可以开始真正把汉字搬上屏幕中去了。

在这里可以看出 VSCode 识别汉字的方式是 GBK 码，换句话来说就是 VSCode 自动会把汉字看成是两个字节表示的东西。知道了要表示的汉字和其 GBK 码，那么就可以去找对应的点阵数据。在这里我们就定义了一个获取汉字点阵数据的函数，其定义如下：

```

/**
 * @brief      获取汉字点阵数据
 * @param      code   : 当前汉字编码 (GBK 码)
 * @param      mat    : 当前汉字点阵数据存放地址
 * @param      size   : 字体大小
 * @note       size 大小的字体, 其点阵数据大小为:
 *             (size / 8 + ((size % 8) ? 1 : 0)) * (size) 字节
 * @retval     无
 */
static void text_get_hz_mat(unsigned char *code,unsigned char *mat,uint8_t size)
{
    unsigned char qh, ql;
    unsigned char i;
    unsigned long foffset;
    uint8_t csize;

    /* 计算字体一个字符对应点阵集所占的字节数 */
    csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size);
    qh = *code;
    ql = *(++code);

    /* 非常用汉字 */
    if ((qh < 0x81) || (ql < 0x40) || (ql == 0xFF) || (qh == 0xFF))
    {
        for (i = 0; i < csize; i++)
        {
            *mat++ = 0x00; /* 填充满格 */
        }
        return;
    }

    if (ql < 0x7F)
    {
        ql -= 0x40;
    }
    else
    {
        ql -= 0x41;
    }
}

```



```
qh -= 0x81;
foffset = ((unsigned long)190 * qh + ql) * csize; /* 得到字库中的字节偏移量 */

switch (size)
{
    case 12:
    {
        fonts_partition_read(mat, foffset + ftinfo.f12addr, csize);
        break;
    }
    case 16:
    {
        fonts_partition_read(mat, foffset + ftinfo.f16addr, csize);
        break;
    }
    case 24:
    {
        fonts_partition_read(mat, foffset + ftinfo.f24addr, csize);
        break;
    }
}
```

函数实现的依据就是前面 23.1.3 小节讲到的两条公式：

当 GBKL < 0X7F 时： Hp = ((GBKH - 0x81) \* 190 + GBKL - 0X40) \* csiz;

当 GBKL > 0X80 时: Hp = ((GBKH - 0x81) \* 190 + GBKL - 0X41) \* csiz;

目标汉字的 GBK 码满足上面两条公式其一，就会得出与一个 GBK 对应的汉字点阵数据的偏移。在这个基础上，通过汉字点阵的大小，就可以从对应的字库提取目标汉字点阵数据。

在获取到点阵数据后，接下来就可以进行汉字显示，下面看一下汉字显示函数，其定义如下：

```
/***
 * @brief      显示一个指定大小的汉字
 * @param      x,y      : 汉字的坐标
 * @param      font     : 汉字 GBK 码
 * @param      size     : 字体大小
 * @param      mode     : 显示模式
 * @note       0, 正常显示(不需要显示的点,用 LCD 背景色填充,即 g_back_color)
 * @note       1, 叠加显示(仅显示需要显示的点, 不需要显示的点, 不做处理)
 * @param      color   : 字体颜色
 * @retval     无
 */
void text_show_font(uint16_t x,
                    uint16_t y,
                    uint8_t *font,
                    uint8_t size,
                    uint8_t mode,
                    uint16_t color)
{
    uint8_t temp, t, t1;
    uint16_t y0 = y;
    uint8_t *dzk;
    uint8_t csize;
    uint8_t font_size = size;
```

```
/* 计算字体一个字符对应点阵集所占的字节数 */
```

```
cscale = (font_size / 8 + ((font_size % 8) ? 1 : 0)) * (font_size);
```

```
if ((font_size != 12) && (font_size != 16) && (font_size != 24))
```

{

```
    return;
```

}

```
dzk = (uint8_t *)malloc(font_size * 5); /* 申请内存 */
```

```

if (dzk == NULL)
{
    return;
}

text_get_hz_mat(font, dzk, font_size);           /* 得到相应大小的点阵数据 */

for (t = 0; t < csize; t++)
{
    temp = dzk[t];                            /* 得到点阵数据 */

    for (t1 = 0; t1 < 8; t1++)
    {
        if (temp & 0x80)
        {
            lcd_draw_pixel(x, y, color);      /* 画需要显示的点 */
        }
    }

    /* 如果非叠加模式，不需要显示的点用背景色填充 */
    else if (mode == 0)
    {
        lcd_draw_pixel(x, y, 0xffff);        /* 填充背景色 */
    }

    temp <<= 1;
    y++;
    if ((y - y0) == font_size)
    {
        y = y0;
        x++;
        break;
    }
}
}

free(dzk);                                     /* 释放内存 */
}

```

汉字显示函数通过调用获取汉字点阵数据函数 `text_get_hz_mat` 就获取到点阵数据，使用 `lcd` 画点函数把点阵数据中“1”的点都画出来，最终会在 LCD 显示你所要表示的汉字。

其他函数就不多讲解，大家可以自行消化。

### 23.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 `CMakeLists.txt` 文件，其内容如下所示：

```

set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)

set(requirements
    driver
    fatfs)

idf_component_register(SRC_DIRS ${src_dirs}
                      INCLUDE_DIRS ${include_dirs} REQUIREMENTS ${requirements})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 `fatfs` 依赖库需要由开发者自行添加，以确保汉字显示驱动能够顺利集成到构建

系统中。这一步骤是必不可少的，它确保了汉字显示驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

打开本实验 main 文件下的 CMakeLists.txt 文件，其内容如下所示：

```
idf_component_register(
    SRC_DIRS
        "."
        "APP"
    INCLUDE_DIRS
        "."
        "APP")
```

上述的红色 APP 驱动需要由开发者自行添加，在此便不做赘述了。

### 23.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/** @brief 程序入口
 * @param 无
 * @retval 无
 */
void app_main(void)
{
    esp_err_t ret;
    uint8_t t;
    uint8_t key;
    uint32_t fontcnt;
    uint8_t i;
    uint8_t j;
    uint8_t fontx[2];

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    ESP_ERROR_CHECK(ret);

    led_init(); /* 初始化 LED */
    spi2_init(); /* 初始化 SPI */
    lcd_init(); /* 初始化 LCD */

    while (sd_spi_init()) /* 检测不到 SD 卡 */
    {
        lcd_show_string(0, 0, 200, 16, 16, "SD Card Failed!", RED);
        vTaskDelay(200);
        lcd_fill(0, 0, 200 + 30, 50 + 16, WHITE);
        vTaskDelay(200);
    }

    while (fonts_init()) /* 检查字库 */
    {
        UPD:
        lcd_clear(WHITE); /* 清屏 */
        lcd_show_string(0, 0, 200, 16, 16, "ESP32-S3", RED);

        key = fonts_update_font(0, 20, 16, (uint8_t *)"0:", RED); /* 更新字库 */

        while (key) /* 更新失败 */
        {
```

```

    lcd_show_string(0, 50, 200, 16, 16, "Font Update Failed!", RED);
    vTaskDelay(200);
    lcd_fill(0, 50, 200 + 20, 90 + 16, WHITE);
    vTaskDelay(200);
}

lcd_show_string(0, 20, 200, 16, 16, "Font Update Success!      ", RED);
vTaskDelay(1500);
lcd_clear(WHITE);                                /* 清屏 */
}

text_show_string(0, 0, 200, 16, "GBK 字库测试程序", 16, 0, RED);
text_show_string(0, 20, 200, 16, "BOOT: 更新字库", 16, 0, RED);
text_show_string(0, 40, 200, 16, "汉字计数器:", 16, 0, BLUE);
text_show_string(0, 60, 200, 12, "对应汉字为:", 12, 0, BLUE);

while (1)
{
    fontcnt = 0;

    for (i = 0x81; i < 0xFF; i++)           /* GBK 内码高字节范围为 0x81~0xFE */
    {
        fontx[0] = i;
        /* GBK 内码低字节范围为 0x40~0x7E、0x80~0xFE） */
        for (j = 0x40; j < 0xFE; j++)
        {
            if (j == 0x7F)
            {
                continue;
            }

            fontcnt++;
            lcd_show_num(108, 40, fontcnt, 5, 16, BLUE); /* 汉字计数显示 */
            fontx[1] = j;
            text_show_font(108, 60, fontx, 12, 0, BLUE);

            t = 200;

            while ((t--) != 0)          /* 延时，同时扫描按键 */
            {
                vTaskDelay(1);

                key = key_scan(0);

                if (key == BOOT)
                {
                    goto UPD; /* 跳转到 UPD 位置（强制更新字库） */
                }
            }

            LED_TOGGLE();
        }
    }
}
}

```

程序很简单，当系统从 SD 卡更新字库完成时，会在 LCD 上显示汉字。按下 BOOT 按键可重新更新汉字字库。

## 23.4 下载验证

本例程支持 12\*12、16\*16 和 24\*24 等三种字体的显示，将程序下载到 DNESP32S3M 最小系统板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 开始显示三种大小的汉字及内码如下图所示：

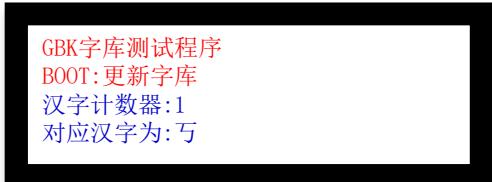


图 23.4.1 汉字显示实验显示效果

## 第二十四章 图片显示实验

在开发产品的时候，很多时候，我们都会用到图片解码，在本章中，我们将向大家介绍如何通过 ESP32-S3 来解码 BMP/JPG/JPEG/PNG/GIF 等图片，并在 SPILCD 上显示出来。

本章分为以下几个小节：

- 24.1 图片格式简介
- 24.2 硬件设计
- 24.3 程序设计
- 24.4 下载验证

### 24.1 图片格式介绍

我们常用的图片格式有很多，一般最常用的有三种：JPEG（或 JPG）、BMP、PNG 和 GIF。其中 JPEG（或 JPG）、PNG 和 BMP 是静态图片，而 GIF 则是可以实现动态图片。下面，我们简单介绍一下这三种图片格式。

#### 24.1.1 BMP 编码简介

我们常用的图片格式有很多，一般最常用的有三种：JPEG（或 JPG）、BMP 和 GIF。其中 JPEG（或 JPG）和 BMP 是静态图片，而 GIF 则是可以实现动态图片。下面，我们简单介绍一下这三种图片格式。

首先，我们来看看 BMP 图片格式。BMP（全称 Bitmap）是 Window 操作系统中的标准图像文件格式，文件后缀名为“.bmp”，使用非常广。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，BMP 文件所占用的空间很大，但是没有失真。BMP 文件的图像深度可选 1bit、4bit、8bit、16bit、24bit 及 32bit。BMP 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

典型的 BMP 图像文件由四部分组成：

- ①：位图头文件数据结构，它包含 BMP 图像文件的类型、显示内容等信息；
- ②：位图信息数据结构，它包含有 BMP 图像的宽、高、压缩方法，以及定义颜色等信息
- ③：调色板，这个部分是可选的，有些位图需要调色板，有些位图，比如真彩色图（24 位的 BMP）就不需要调色板；
- ④：位图数据，这部分的内容根据 BMP 位图使用的位数不同而不同，在 24 位图中直接使用 RGB，而其他的小于 24 位的使用调色板中颜色索引值。

关于 BMP 的详细介绍，请参考光盘的《BMP 图片文件详解.pdf》。

#### 24.1.2 JPEG 编码简介

JPEG 是 Joint Photographic Experts Group(联合图像专家组)的缩写，文件后缀名为“.jpg”或“.jpeg”，是最常用的图像文件格式，由一个软件开发联合会组织制定，同 BMP 格式不同，JPEG 是一种有损压缩格式，能够将图像压缩在很小的储存空间，图像中重复或不重要的资料会被丢失，因此容易造成图像数据的损伤（BMP 不会，但是 BMP 占用空间大）。尤其是使用过高的压缩比例，将使最终解压缩后恢复的图像质量明显降低，如果追求高品质图像，不宜采用过高压缩比例。但是 JPEG 压缩技术十分先进，它用有损压缩方式去除冗余的图像数据，在获得极高的压缩率的同时能展现十分丰富生动的图像，换句话说，就是可以用最少的磁盘空间得到较好的图像品质。而且 JPEG 是一种很灵活的格式，具有调节图像质量的功能，允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在 10:1 到 40:1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好。比如可以把 1.37Mb 的 BMP 位图文件压缩至 20.3KB。当然也可以在图像质量和文件尺寸之间找到平衡点。JPEG 格式压缩的主要信息是高频信息，对色彩的信息保留较好，适合应用于互联网，可减少图像的传输时间，可以支持 24bit 真彩色，

也普遍应用于需要连续色调的图像。

JPEG/JPG 的解码过程可以简单的概述为如下几个部分：

①：从文件头读出文件的相关信息。

JPEG 文件数据分为文件头和图像数据两大部分，其中文件头记录了图像的版本、长宽、采样因子、量化表、哈夫曼表等重要信息。所以解码前必须将文件头信息读出，以备图像数据解码过程之用。

②：从图像数据流读取一个最小编码单元(MCU)，并提取出里边的各个颜色分量单元。

③：将颜色分量单元从数据流恢复成矩阵数据。

使用文件头给出的哈夫曼表，对分割出来的颜色分量单元进行解码，把其恢复成  $8 \times 8$  的数据矩阵。

④： $8 \times 8$  的数据矩阵进一步解码。

此部分解码工作以  $8 \times 8$  的数据矩阵为单位，其中包括相邻矩阵的直流系数差分解码、使用文件头给出的量化表反量化数据、反 Zig-zag 编码、隔行正负纠正、反向离散余弦变换等 5 个步骤，最终输出仍然是一个  $8 \times 8$  的数据矩阵。

⑤：颜色系统 YCrCb 向 RGB 转换。

将一个 MCU 的各个颜色分量单元解码结果整合起来，将图像颜色系统从 YCrCb 向 RGB 转换。

⑥：排列整合各个 MCU 的解码数据。

不断读取数据流中的 MCU 并对其解码，直至读完所有 MCU 为止，将各 MCU 解码后的数据正确排列成完整的图像。JPEG 的解码本身是比较复杂的，这里 FATFS 的作者，提供了一个轻量级的 JPG/JPEG 解码库：TjpgDec，最少仅需 3KB 的 RAM 和 3.5KB 的 FLASH 即可实现 JPG/JPEG 解码，本例程采用 TjpgDec 作为 JPG/JPEG 的解码库，关于 TjpgDec 的详细使用，请参考“A 盘→4，软件资料→图片编解码→TjpgDec 技术手册”这个文档。

### 24.1.3 PNG 编码简介

PNG (Portable Network Graphics) 是一种无损的位图图像格式，旨在替代 GIF 格式并增加一些 GIF 文件格式所不具备的特性。PNG 图像使用一种称为 DEFLATE 的无损数据压缩算法来减小文件大小，不会损失图像质量。这种压缩算法结合了 LZ77 算法和哈夫曼编码，能够有效地压缩数据并减小文件大小。

在 PNG 编码过程中，预滤器编码格式被用来先对图像数据进行预处理，以便更好地利用 Deflate 算法进行压缩。PNG 定义了五种不同的预滤器，分别是 None、Sub、Up、Average 和 Paeth。这些预滤器根据图像像素的特性对每一行的像素进行编码，以更好地利用 Deflate 算法进行压缩。

PNG 还支持多种颜色模式，包括 8 位灰度图像、索引彩色图像和 24 位真彩色图像，并且可以支持 Alpha 通道透明度，这意味着可以在图像中创建半透明的效果。此外，PNG 还支持多层图像，可以将多个图像组合在一起，每个图像可以具有自己的透明度和颜色。

典型的 PNG 图像文件由以下几部分组成：

①：文件署名域 (File Signature)：这是文件的开头部分，由 8 个字节组成，用于标识该文件是一个 PNG 文件。其值固定为"89 50 4E 47 0D 0A 1A 0A"。

②：关键数据块 (Critical Chunk)：这是 PNG 文件必须包含的数据块，包括 IHDR、IDAT、IEND 等。IHDR 块包含了图像的基本信息，如宽度、高度、像素格式等；IDAT 块包含了图像的实际数据；IEND 块标记了图像数据的结束。

③：辅助数据块 (Ancillary Chunk)：这是可选的数据块，用于存储与图像相关的其他信息，如文本注释、时间戳等。这些数据块对于解码图像是可选的，但如果存在，解码器应当对其进行解析。

### 24.1.4 GIF 编码简介

GIF(Graphics Interchange Format)是 CompuServe 公司开发的图像文件存储格式，1987 年开发的 GIF 文件格式版本号是 GIF87a，1989 年进行了扩充，扩充后的版本号定义为 GIF89a。GIF

图像文件以数据块(block)为单位来存储图像的相关信息。一个 GIF 文件由表示图形/图像的数据块、数据子块以及显示图形/图像的控制信息块组成，称为 GIF 数据流(DataStream)。数据流中的所有控制信息块和数据块都必须在文件头(Header)和文件结束块(Trailer)之间。

GIF 文件格式采用了 LZW(Lempel-ZivWalch)压缩算法来存储图像数据，定义了允许用户为图像设置背景的透明(transparency)属性。此外，GIF 文件格式可在同一个文件中存放多幅彩色图形/图像。如果在 GIF 文件中存放有多幅图，它们可以像演幻灯片那样显示或者像动画那样演示。

一个 GIF 文件的结构可分为文件头(FileHeader)、GIF 数据流(GIFDataStream)和文件终结器(Trailer)三个部分。文件头包含 GIF 文件署名(Signature)和版本号(Version)；GIF 数据流由控制标识符、图象块(ImageBlock)和其他的一些扩展块组成；文件终结器只有一个值为 0x3B 的字符(';) 表示文件结束。

关于 GIF 的详细介绍，请参考光盘 GIF 解码相关资料。图片格式简介，我们就介绍到这里。

## 24.2 硬件设计

### 24.2.1 例程功能

开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg、png 和 gif 格式），循环显示，通过按 KEY0 和 KEY2 可以快速浏览下一张和上一张，KEY\_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。如果未找到 PICTURE 文件夹/任何图片文件，则提示错误。同样我们也是用 DS0 来指示程序正在运行。

### 24.2.2 硬件资源

- 1.LED  
LED-IO1
- 2.0.96 寸 LCD
- 3.SD
  - CS-IO2
  - SCK-IO12
  - MOSI-IO11
  - MISO-IO13

## 24.3 程序设计

### 24.3.1 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：

### 24.3.2 图片显示函数解析

正点原子提供的 PICTURE 驱动源码包括以下文件，并且已经针对正点原子 ESP32-S3 软硬件进行了移植适配，用户在使用时，仅需将这以下文件添加到自己的工程中即可，如下图所示：

```

./main/APP          |-- jpeg.h
|-- bmp.c           |-- piclib.c
|-- bmp.h           |-- piclib.h
|-- convert.c       |-- png.c
|-- convert.h       |-- png.h
|-- fonts.c         |-- pngle.c
|-- fonts.h         |-- pngle.h
|-- gif.c           |-- text.c
|-- gif.h           |-- text.h
|-- jpeg.c          |-- tjpgd.c
|-- jpeg.h          `-- tjpgd.h

```

图 24.3.2.1 正点原子 PICTURE 驱动源码文件

其中：

bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；  
tjpgd.c 和 tjpgd.h 用于实现对 jpeg/jpg 文件的解码；  
gif.c 和 gif.h 用于实现对 gif 文件的解码；

这几个代码太长了，而且也有规定的标准，需要结合各个图片编码的格式来编写，所以我们在这里不贴出来，大家查看光盘中的源码的实现过程即可。

### 24.3.3 图片显示函数驱动解析

在 IDF 版的 14\_picture 例程中，作者在 14\_picture\components\BSP 路径下并未添加新的内容，而是在 14\_picture\main\APP 路径下面，新增了一个 APP 文件，我们将详细解析这四个文件的实现内容。

#### 1. 解码库的控制句柄(pic\_phy 和 pic\_info)

我们使用这个接口，把解码后的图形数据与 LCD 的实际操作对应起来。为了方便去显示图片，我们需要将图片的信息与我们的 LCD 联系上。这里我们定义了(pic\_phy 和(pic\_info 分别用于定义图片解码库的 LCD 操作和存放解码后的图片尺寸颜色信息。它们的定义如下：

```

/* 图片显示物理层接口 */
/* 在移植的时候，必须由用户自己实现这几个函数 */
typedef struct
{
    /* 画点函数 */
    void(*draw_point)(uint16_t, uint16_t, uint16_t);

    /* 单色填充函数 */
    void(*fill)(uint16_t, uint16_t, uint16_t, uint16_t, uint16_t);

    /* 画水平线函数 */
    void(*draw_hline)(uint16_t, uint16_t, uint16_t, uint16_t);

    /* 多点填充 */
    void(*multicolor)(uint16_t, uint16_t, uint16_t, uint16_t *);
} _pic_phy;

/* 图像信息 */
typedef struct
{
    uint16_t lcdwidth;           /* LCD 的宽度 */
    uint16_t lcdheight;          /* LCD 的高度 */
} _pic_info;

```

在 piclib.c 文件中，我们用上述类型定义了两个结构体，声明如下：

```
_pic_info picinfo;      /* 图片信息 */
```

```
_pic_phys.pic_phys; /* 图片显示物理接口 */
```

### 2, piclib\_init 函数

piclib\_init 函数，该函数用于初始化图片解码的相关信息，用于定义解码后的 LCD 操作。具体定义如下：

```
/***
 * @brief      画图初始化
 * @note       在画图之前, 必须先调用此函数, 指定相关函数
 * @param      无
 * @retval     无
 */
void piclib_init(void)
{
    pic_phys.draw_point = lcd_draw_pixel; /* 画点函数实现, 仅 GIF 需要 */
    pic_phys.fill = lcd_fill; /* 填充函数实现, 仅 GIF 需要 */
    pic_phys.draw_hline = lcd_draw_hline; /* 画线函数实现, 仅 GIF 需要 */
    pic_phys.multicolor = piclib_multi_color; /* 颜色填充函数实现, JPEG、BMP、PNG */

    picinfo.lcdwidth = lcd_self.width; /* 得到 LCD 的宽度像素 */
    picinfo.lcdheight = lcd_self.height; /* 得到 LCD 的高度像素 */
}
```

初始化图片解码的相关信息，这些函数必须由用户在外部实现。我们使用之前 LCD 的操作函数对这个结构体中的绘制操作：画点、画线、画圆等定义与我们的 LCD 操作对应起来。

### 3, piclib\_ai\_load\_picfile 函数

piclib\_ai\_load\_picfile 帮助我们得到需要显示的图片信息并有助于下一步的绘制。本函数需要结合文件系统来操作，图片根据后缀区分并且在文件夹在保存是我们在 PC 下的习分类，也是我们处理和分类图片的最方便的方式。

```
/***
 * @brief      智能画图
 * @note       图片仅在 x,y 和 width, height 限定的区域内显示.
 *
 * @param      filename : 包含路径的文件名(.bmp/.jpg/.jpeg/.png/.gif 等)
 * @param      x, y      : 起始坐标
 * @param      width, height : 显示区域
 * @param      fast       : 使能快速解码
 * @arg        0, 不使能
 * @arg        1, 使能
 * @note      图片尺寸小于等于液晶分辨率, 才支持快速解码
 * @retval     无
 */
uint8_t piclib_ai_load_picfile(char *filename,
                               uint16_t x,
                               uint16_t y,
                               uint16_t width,
                               uint16_t height)
{
    uint8_t res = 0; /* 返回值 */
    uint8_t temp;

    if ((x + width) > picinfo.lcdwidth) return PIC_WINDOW_ERR; /* x 坐标超范围了 */
    if ((y + height) > picinfo.lcdheight) return PIC_WINDOW_ERR; /* y 坐标超范围了 */

    /* 得到显示方框大小 */
    if (width == 0 || height == 0) return PIC_WINDOW_ERR; /* 窗口设定错误 */

    /* 文件名传递 */
    temp = exfun_file_type(filename); /* 得到文件的类型 */

    switch (temp)
```

```

{
    case T_BMP:
        res = bmp_decode(filename, width, height);           /* 解码 BMP */
        break;

    case T_JPG:
    case T_JPEG:
        res = jpeg_decode(filename, width, height);         /* 解码 JPG/JPEG */
        break;

    case T_GIF:
        res = gif_decode(filename, x, y, width, height);   /* 解码 gif */
        break;

    case T_PNG:
        res = png_decode(filename, width, height);          /* 解码 PNG */
        break;

    default:
        res = PIC_FORMAT_ERR;                            /* 非图片格式!!! */
        break;
}

return res;
}

```

该函数的形参描述，如下表所示：

形参	描述
filename	filename 是文件的路径名，具体可以参考 FATFS 一节的描述，为字符口，我们的例程采用的是 SD 卡存图片，故一般为”0:/PICTURE/*.GIF”等类似格式。
x	为画图的起始 x 坐标
y	为画图的起始 y 坐标
width	形成了以 x、y 为起点的(x,y)~(x+width,y+height)的矩形显示区域，对屏幕坐标不理解的可能参考我们的 SPILCD 一节的描述。
height	形成了以 x、y 为起点的(x,y)~(x+width,y+height)的矩形显示区域，对屏幕坐标不理解的可能参考我们的 SPILCD 一节的描述。

表 24.3.1.1 函数 piclib\_ai\_load\_picfile()形参描述

返回值：0 表示成功，其他表示失败。

piclib\_ai\_load\_picfile 函数，整个图片显示的对外接口，外部程序，通过调用该函数，可以实现 bmp、jpg/jpeg、png 和 gif 的显示，该函数根据输入文件的后缀名，判断文件格式，然后交给相应的解码程序（bmp 解码/jpeg 解码/gif 解码/png 解码），执行解码，完成图片显示。

这里用到的 exfun\_file\_type() 函数是我们用来判断文件类型，方便我们进行程序设计。由于图片显示需要用到大内存，我们使用动态内存分配来实现，我们仍使用我们自定的内存管理函数来管理程序内存。申请内存函数 piclib\_mem\_malloc() 和 内存释放函数 piclib\_mem\_free() 的实现就比较简单了，大家参考光盘的源码即可。

#### 24.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```

set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD

```

```

LED
SPI)

set(requires
    driver
    fatfs)

idf_component_register(SRC_DIRS ${src_dirs}
                       INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)

```

上述的红色 **fatfs** 依赖库需要由开发者自行添加，以确保图片显示驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了图片显示驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

打开本实验 main 文件下的 CMakeLists.txt 文件，其内容如下所示：

```

idf_component_register(
    SRC_DIRS
        "."
        "APP"
    INCLUDE_DIRS
        "."
        "APP")

```

上述的红色 APP 驱动需要由开发者自行添加，在此便不做赘述了。

#### 24.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 `app_main`。该函数代码如下。

```

/** @brief      得到 path 路径下, 目标文件的总个数
 * @param      path : 路径
 * @retval     总有效文件数
 */
uint16_t pic_get_tnum(char *path)
{
    uint8_t res;
    uint16_t rval = 0;
    FF_DIR tdir;                                /* 临时目录 */
    FILINFO *tfileinfo;                          /* 临时文件信息 */
    tfileinfo = (FILINFO *)malloc(sizeof(FILINFO)); /* 申请内存 */
    res = f_opendir(&tdir, (const TCHAR *)path); /* 打开目录 */

    if (res == FR_OK && tfileinfo)
    {
        while (1)                                /* 查询总的有效文件数 */
        {
            res = f_readdir(&tdir, tfileinfo);    /* 读取目录下的一个文件 */

            if (res != FR_OK || tfileinfo->fname[0] == 0) break;
            res = exfun_file_type(tfileinfo->fname);

            if ((res & 0X0F) != 0X00)              /* 取低四位,看看是不是图片文件 */
            {
                rval++;                           /* 有效文件数增加 1 */
            }
        }

        free(tfileinfo);                         /* 释放内存 */
        return rval;
    }
}

```

```
* @brief      程序入口
* @param      无
* @retval     无
*/
void app_main(void)
{
    esp_err_t ret = 0;
    uint8_t res = 0;
    FF_DIR picdir;                                /* 图片目录 */
    FILINFO *picfileinfo;                          /* 文件信息 */
    char *pname;                                  /* 带路径的文件名 */
    uint16_t totopicnum;                           /* 图片文件总数 */
    uint16_t curindex = 0;                          /* 图片当前索引 */
    uint8_t key = 0;                               /* 键值 */
    uint8_t pause = 0;                             /* 暂停标记 */
    uint8_t t;
    uint16_t temp;
    uint32_t *picoffsettbl;                        /* 图片文件 offset 索引表 */

    ret = nvs_flash_init();                         /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
        || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    led_init();                                    /* 初始化 LED */
    spi2_init();                                   /* 初始化 SPI */
    lcd_init();                                   /* 初始化 LCD */

    while (sd_spi_init())                         /* 检测不到 SD 卡 */
    {
        lcd_show_string(0, 0, 200, 16, 16, "SD Card Error!", RED);
        vTaskDelay(500);
        lcd_show_string(0, 0, 200, 16, 16, "Please Check! ", RED);
        vTaskDelay(500);
    }

    res = exfuns_init();

    while (fonts_init())                          /* 检查字库 */
    {
        lcd_clear(WHITE);                         /* 清屏 */
        lcd_show_string(0, 0, 200, 16, 16, "ESP32-S3", RED);

        key = fonts_update_font(0, 50, 16, (uint8_t *)"0:", RED); /* 更新字库 */

        while (key)                                /* 更新失败 */
        {
            lcd_show_string(0, 0, 200, 16, 16, "Font Update Failed!", RED);
            vTaskDelay(200);
            lcd_fill(0, 0, 200 + 20, 90 + 16, WHITE);
            vTaskDelay(200);
        }

        lcd_show_string(0, 50, 200, 16, 16, "Font Update Success! ", RED);
        vTaskDelay(1500);
        lcd_clear(WHITE);                         /* 清屏 */
    }

    text_show_string(0, 0, 200, 16, "图片显示实验", 16, 0, RED);
}
```

```
text_show_string(0, 20, 200, 16, "BOOT:NEXT", 16, 0, RED);

while (f_opendir(&picdir, "0:/PICTURE"))           /* 打开图片文件夹 */
{
    text_show_string(0, 60, 240, 16, "PICTURE 文件夹错误!", 16, 0, RED);
    vTaskDelay(200);
    lcd_fill(0, 60, 240, 186, WHITE);                /* 清除显示 */
    vTaskDelay(200);
}

totpicnum = pic_get_tnum("0:/PICTURE");             /* 得到总有效文件数 */

while (totpicnum == NULL)                           /* 图片文件为 0 */
{
    text_show_string(0, 60, 240, 16, "没有图片文件!", 16, 0, RED);
    vTaskDelay(200);
    lcd_fill(0, 60, 240, 186, WHITE);                /* 清除显示 */
    vTaskDelay(200);
}

picfileinfo = (FILINFO *)malloc(sizeof(FILINFO));   /* 申请内存 */
pname = malloc(255 * 2 + 1);
picoffsettbl = malloc(4 * totpicnum);

while (!picfileinfo || !pname || !picoffsettbl)      /* 内存分配出错 */
{
    text_show_string(0, 60, 240, 16, "内存分配失败!", 16, 0, RED);
    vTaskDelay(200);
    lcd_fill(0, 60, 240, 186, WHITE);                /* 清除显示 */
    vTaskDelay(200);
}

/* 记录索引 */
res = f_opendir(&picdir, "0:/PICTURE");           /* 打开目录 */

if (res == FR_OK)
{
    curindex = 0;                                     /* 当前索引为 0 */

    while (1)                                         /* 全部查询一遍 */
    {
        temp = picdir.dptr;                          /* 记录当前 dptr 偏移 */
        res = f_readdir(&picdir, picfileinfo);       /* 读取目录下的一个文件 */
        if (res != FR_OK || picfileinfo->fname[0] == 0) break;

        res = exfuns_file_type(picfileinfo->fname);

        if ((res & 0XF) != 0X0)                      /* 取高四位,看看是不是图片文件 */
        {
            picoffsettbl[curindex] = temp;           /* 记录索引 */
            curindex++;
        }
    }
}

vTaskDelay(1500);                                    /* 初始化画图 */
piclib_init();                                      /* 从 0 开始显示 */
curindex = 0;
res = f_opendir(&picdir, (const TCHAR *)"0:/PICTURE"); /* 打开目录 */

while (res == FR_OK)                                /* 打开成功 */
{
    dir_sdi(&picdir, picoffsettbl[curindex]);
}
```

```

res = f_readdir(&picdir, picfileinfo);
if (res != FR_OK || picfileinfo->fname[0] == 0)break;

strcpy((char *)pname, "0:/PICTURE/");
strcat((char *)pname, (const char *)picfileinfo->fname);
lcd_clear(BLACK);
piclib_ai_load_picfile(pname, 0, 0, lcd_self.width, lcd_self.height);
text_show_string(2, 2, lcd_self.width, 16, (char *)pname, 16, 0, RED);
t = 0;

while (1)
{
    if (t > 250)key = 1; /* 模拟一次按下 KEY0 */

    if ((t % 20) == 0)
    {
        LED_TOGGLE(); /* LED 闪烁, 提示程序正在运行. */
    }

    key = key_scan(0);

    if (key == BOOT)
    {
        curindex++;
        if (curindex >= totpicnum)
        {
            curindex = 0;
        }
        break;
    }
    if (pause == 0)t++;
    vTaskDelay(10);
}

res = 0;
}

free(picfileinfo); /* 释放内存 */
free(pname); /* 释放内存 */
free(picoffsettbl); /* 释放内存 */
}

```

可以看到整个设计思路是根据图片解码库来设计的，`piclib_ai_load_picfile()`是这套代码的核心，其它的交互是围绕它和图片解码后的图片信息作的显示。大家再仔细对照光盘中的源码进一步了解整个设置思路。另外，我们的程序中只分配了 4 个文件索引，故更多数量的图片无法直接在本程序下演示，大家根据自己的需要再进行修改即可。

## 24.4 下载验证

在代码编译成功之后，我们下载代码到 DNESP32S3M 最小系统板上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了，即：在 SD 卡根目录新建：PICTURE 文件夹，并存放一些图片文件(.bmp/.jpg/.gif/.png)在该文件夹内），如下图所示：



图 24.4.1 图片显示实验显示效果

## 第二十五章 USB 虚拟串口(Slave)实验

本章，我们将向大家介绍如何利用 USB 在 DNESP32S3M 最小系统板实现一个 USB 虚拟串口，通过 USB 与电脑数据交互。

本章分为以下几个小节：

25.1 USB 虚拟串口简介

25.2 硬件设计

25.3 程序设计

25.4 下载验证

### 25.1 USB 虚拟串口简介

USB 虚拟串口，简称 VCP，是 Virtual COM Port 的简写，它是利用 USB 的 CDC 类来实现的一种通信接口。CDC(Communication Device Class)类是 USB2.0 标准下的一个设备类，定义了通信相关设备的抽象集合。

我们可以利用 ESP32 自带的 USB 功能，来实现一个 USB 虚拟串口，从而通过 USB，实现电脑与 ESP32 的数据互传。上位机无需编写专门的 USB 程序，只需要一个串口调试助手即可调试，非常实用。

### 25.2 硬件设计

#### 25.2.1 例程功能

本实验利用 ESP32 自带的 USB 功能，连接电脑 USB，虚拟出一个 USB 串口，实现电脑和 DNESP32S3M 最小系统板的数据通信。本例程功能完全同实验 4（串口通信实验），只不过串口变成了 ESP32 的 USB 虚拟串口。当 USB 连接电脑（USB 线插入 USB\_SLAVE 接口），DNESP32S3M 最小系统板将通过 USB 和电脑建立连接，并虚拟出一个串口。

LED 闪烁，提示程序运行。USB 和电脑连接成功后。

#### 25.2.2 硬件资源

1. LED 灯  
LED -IO0
2. 0.96 寸 LCD
3. USB

#### 25.2.3 原理图

本章实验使用 USB 接口与 PC 进行连接，DNESP32S3M 最小系统板板载了一个 USB 接口，用于连接其他 USB 设备，USB 接口与 MCU 的连接原理图，如下图所示：

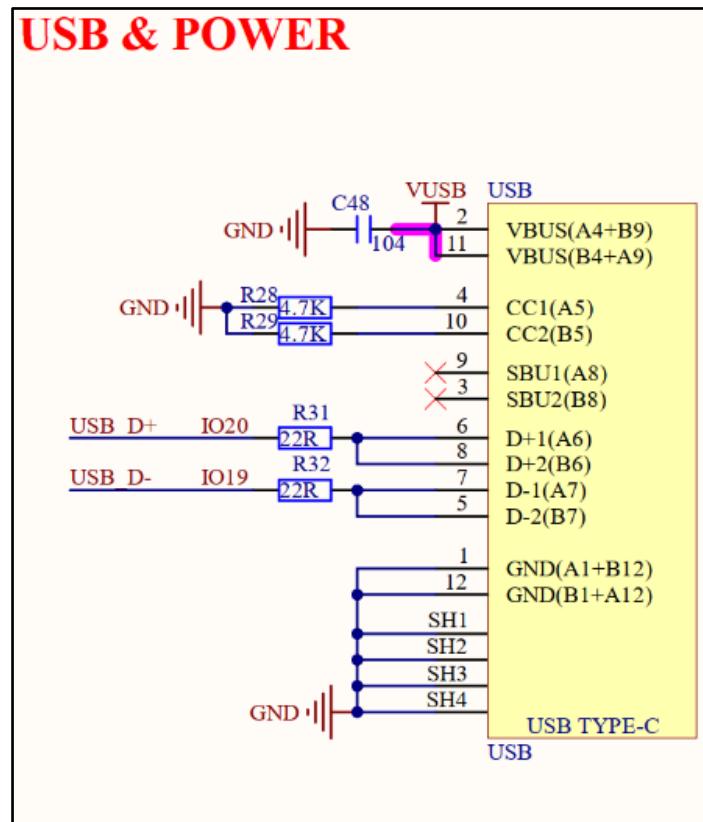


图 25.2.3.1 USB 接口与 MCU 的连接原理图

## 25.3 程序设计

### 25.3.1 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：

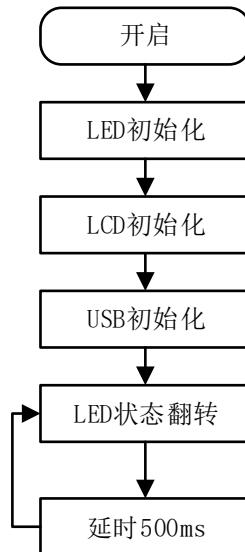


图 25.3.1.1 USB 虚拟串口实验程序流程图

### 25.3.2 USB 虚拟串口函数解析

ESP-IDF 提供了一套 API 来配置 USB。要使用此功能，需要导入必要的头文件：

```
#include "tinyusb.h"
#include "tusb_cdc_acm.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 USB 函数，这些函数的描述及其作用如下：

#### 1, USB 设备登记

该函数用给定的配置，来配置 USB 设备，该函数原型如下所示：

```
esp_err_t tinyusb_driver_install(const tinyusb_config_t *config);
```

该函数的形参描述如下表所示：

参数	描述
config	Tinyusb 堆栈特定配置

表 25.3.2.1 tinyusb\_driver\_install() 函数形参描述

返回值：ESP\_OK 表示设备登记成功，其他值表示设备登记失败。

该函数使用 tinyusb\_config\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
tinyusb_config_t	device_descriptor	指向设备描述符的指针，例程设置为 NULL
	string_descriptor	指向字符串描述符数组的指针，例程设置为 NULL
	external_phy	USB 应该使用外部 PHY，例程设置为 false
	configuration_descriptor	指向配置描述符的指针，例程设置为 NULL

表 25.3.2.2 i2s\_pin\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 tinyusb\_driver\_install() 函数，用以安装 USB 驱动。

#### 2, USB 设备初始化

该函数用给定的配置，来初始化 USB 设备，该函数原型如下所示：

```
esp_err_t tusb_cdc_acm_init(const tinyusb_config_cdcacm_t *cfg);
```

该函数的形参描述如下表所示：

参数	描述
cfg	指向 USB 设备初始化配置结构体

表 25.3.2.3 tusb\_cdc\_acm\_init() 函数形参描述

返回值：ESP\_OK 表示设备初始化成功，其他值表示设备初始化失败。

该函数使用 tinyusb\_config\_cdcacm\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
tinyusb_config_cdcacm_t	usb_dev	USB 设备
	cdc_port	CDC 端口
	rx_unread_buf_sz	配置 RX 缓冲区大小
	callback_rx	接收回调函数
	callback_rx_wanted_char	指向 'tusb_cdcacm_callback_t' 类型的函数的指针，该函数将作为回调处理，例程中配置为 NULL
	callback_line_state_changed	同上，例程中配置为 NULL

callback_line_coding_changed	同上，例程中配置为 NULL
------------------------------	----------------

表 25.3.2.4 tinyusb\_config\_cdcacm\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 `tusb_cdc_acm_init()` 函数，用以实例化 USB。

### 3. 注册回调函数

该步骤用以注册回调函数，该函数原型如下所示：

```
esp_err_t tinyusb_cdcacm_unregister_callback(tinyusb_cdcacm_if_t ift,
                                              cdcacm_event_type_t event_type);
```

该函数的形参描述如下表所示：

参数	描述
ift	CDC 对象的编号
event_type	回调所注册事件的类型

表 25.3.2.5 tinyusb\_cdcacm\_unregister\_callback() 函数形参描述

返回值：ESP\_OK 表示设备注册成功，其他值表示设备注册失败。

### 25.3.3 USB 虚拟串口驱动解析

USB 驱动库移植，我们可参考第九章 IDF 组件注册表的形式添加 USB 驱动库。下图为添加 USB 驱动库的工程结构。

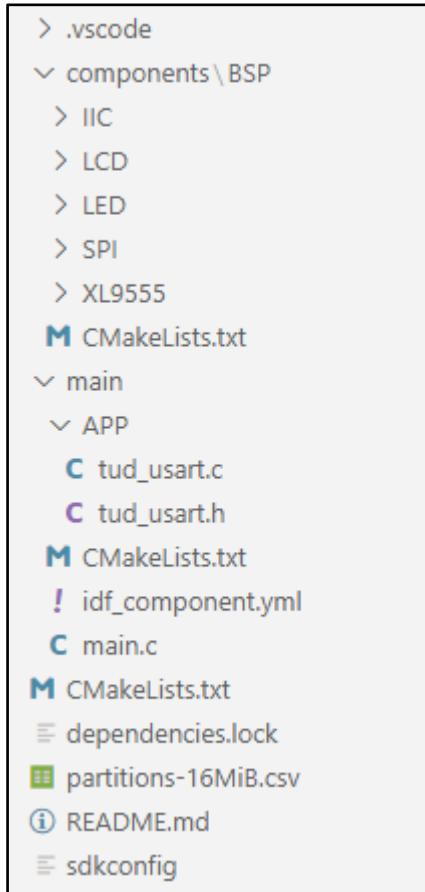


图 25.3.3.1 USB 虚拟串口工程分组

上图中位于 `components` 文件夹下的是我们自己编写的一些外设驱动，`main` 文件夹下包含了一个 `APP` 文件与一个后缀为 `.yml` 的文件。`APP` 文件夹下包含的是 USB 模拟串口代码，而后缀为 `.yml` 的文件其主要作用是将项目中各组件的依赖项定义在单独的清单文件中，并以上图所示的方式进行命名。在我们的例程中提现出来的作用就是简化了整个工程结构。我们在编译的过程中，系统便会帮我们自动生成 USB 外设所需要的依赖库：`espressif_esp_tinyusb` 以及 `espressif_tinyusb`。做到了既能简化项目工程，又能有效规避了在编译中遇到的错误，但前提是

运行时得确保个人的电脑处于联网状态。

#### 25.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```
set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)

set(requires
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
    INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

该路径下的 CmakeList 文件并没有新增内容，主要变化在于 main 文件。

打开本实验 main 文件下的 CMakeLists.txt 文件，其内容如下所示：

```
idf_component_register(
    SRC_DIRS
        "."
        "app"
    INCLUDE_DIRS
        "."
        "app")
```

上述的红色 app 驱动需要由开发者自行添加，以确保 USB 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 USB 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 25.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    ESP_ERROR_CHECK(ret); /* 初始化 LED */
    /* 初始化 SPI */
    /* 初始化 LCD */

    led_init();
    spi2_init();
    lcd_init();
```

```

/* 显示实验信息 */
lcd_show_string(0, 0, 200, 16, 16, "ESP32-S3", RED);
lcd_show_string(0, 20, 200, 16, 16, "USB USART TEST", RED);
lcd_show_string(0, 40, 200, 16, 16, "ATOM@ALIENTEK", RED);

tud_usb_usart(); /* USB 初始化 */

while(1)
{
    LED_TOGGLE();
    vTaskDelay(500);
}
}

```

此部分代码比较简单，通过 `tud_usb_usart()` 等函数初始化 USB，在该函数中需要注册一个调用 CDC 事件的回调函数。此时，如果回调已经注册，那么它将会被覆盖。同时，LCD 显示实验信息，LED 闪烁以示程序正在运行。

## 25.4 下载验证

本例程的测试，不需要安装特定的 USB 驱动，开发者只需用数据线将 USB 接口（**不是 UART 接口**）与 PC 端连接起来即可，并打开串口助手，选择对应的端口号进行数据发送操作。我们打开设备管理器（我用的是 WIN10），在端口（COM 和 LPT）里面可以发现多出了一个 COM8 的设备，这就是 USB 虚拟的串口设备端口，如下图所示：



图 25.4.1 通过设备管理器查看 USB 虚拟的串口设备端口

上图中，ESP32 通过 USB 虚拟的串口，被电脑识别了，端口号为：COM8（可变），字符串名字为：USB 串行设备（COM8）。此时，DNESP32S3M 最小系统板的 LED 闪烁，提示程序运行，如图 44.4.2 所示：



图 25.4.2 USB 虚拟串口连接成功

然后我们打开 XCOM，选择 COM8（需根据自己的电脑识别到的串口号选择），并打开串

口（注意：波特率可以随意设置），就可以进行测试了，如下图所示：

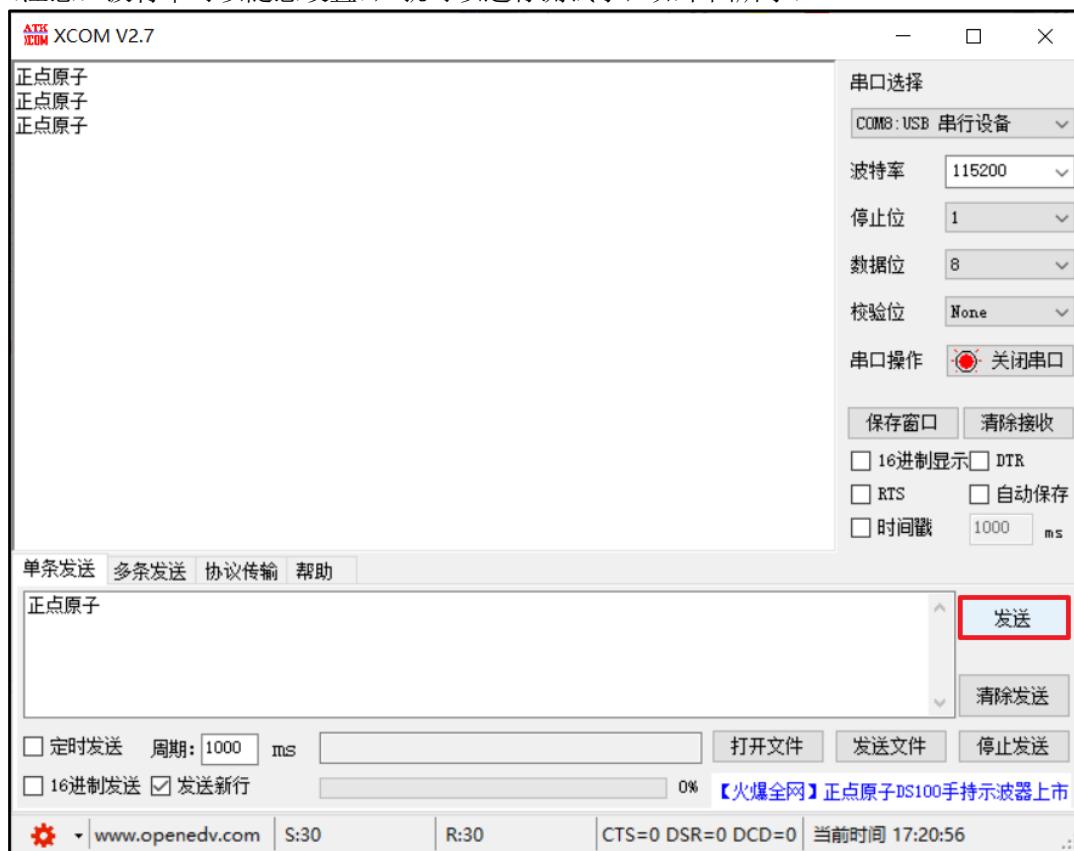


图 25.4.3 ESP32 虚拟串口通信测试

可以看到，我们的串口调试助手，按发送按钮，可以收到电脑发送给 ESP32 的数据（原样返回），说明我们的实验是成功的。

至此，USB 虚拟串口实验就完成了，通过本实验，我们就可以利用 ESP32 的 USB，直接和电脑进行数据互传了，具有广泛的应用前景。

## 第二十六章 Flash 模拟 U 盘实验

本章我们介绍 ESP32S3 的 USB HOST 应用，即通过 USB HOST 功能，将某个分区表实现模拟 U 盘/读卡器等大容量 USB 存储设备。

本章分为以下几个小节：

26.1 Flash 模拟 U 盘简介

26.2 硬件设计

26.3 程序设计

26.4 下载验证

### 26.1 Flash 模拟 U 盘简介

U 盘，全称 USB 闪存盘，英文名“USB flash disk”。它是一种使用 USB 接口的无需物理驱动器的微型高容量移动存储产品，通过 USB 接口与主机连接，实现即插即用，是最常用的移动存储设备之一。

ESP32-S3 的 USB OTG FS 支持 U 盘，并且乐鑫官方提供了 esp\_tinyusb 开发库，我们可使用 IDF 组件注册的方式把该库移植至项目工程中。

### 26.2 硬件设计

#### 26.2.1 例程功能

本实验利用 ESP32 自带的 USB 功能，通过 USB 连接电脑后，子分区会在电脑上进行加载，并显示该子分区的容量，我们可测试子分区数据的读写了。

LED 闪烁，提示程序运行，USB 和电脑连接成功。

#### 26.2.2 硬件资源

1. LED 灯

    LED -IO0

2. 0.96 寸 LCD

3. USB

#### 26.2.3 原理图

USB 原理图已在 25.2.3 章节中详细阐述，为避免重复，此处不再赘述

### 26.3 程序设计

#### 26.3.1 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：

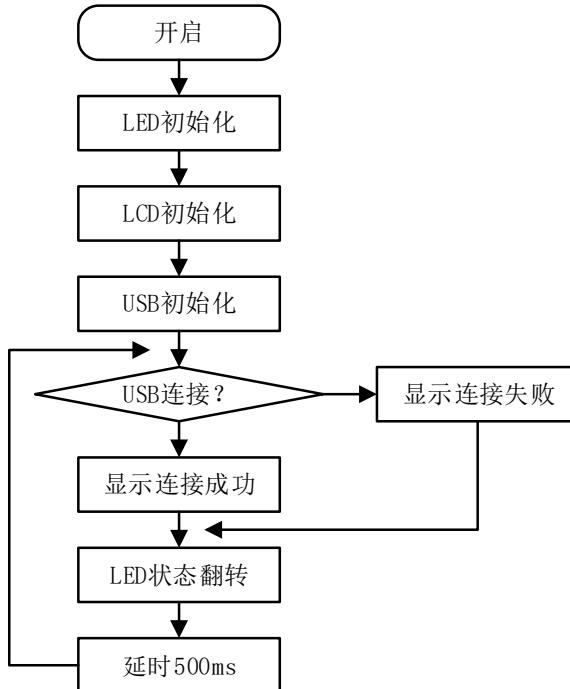


图 26.3.1.1 Flash 模拟 U 盘实验程序流程图

### 26.3.2 Flash 模拟 U 盘函数解析

ESP-IDF 提供了一套 API 来配置 Flash。要使用此功能，需要导入必要的头文件：

```
#include "ff.h"
#include "esp_vfs_fat.h"
#include "tinyusb.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 Flash 函数，这些函数的描述及其作用如下：

#### 1. 挂载分区函数

该函数用给定的配置，来挂载分区，该函数原型如下所示：

```
esp_err_t esp_vfs_fat_spiflash_mount_rw_wl(const char* base_path,
                                             const char* partition_label,
                                             const esp_vfs_fat_mount_config_t* mount_config,
                                             wl_handle_t* wl_handle);
```

该函数的形参描述如下表所示：

参数	描述
base_path	应该挂载 FATFS 分区的路径
partition_label	应该使用的分区的标签
mount_config	指向带有附加参数的结构的指针，用于挂载 FATFS
wl_handle	磨损均衡驱动句柄

表 27.3.2.1 esp\_vfs\_fat\_spiflash\_mount\_rw\_wl() 函数形参描述

该函数的返回值描述，如下表所示：

返回值	描述
ESP_OK	返回：0，配置成功
ESP_ERR_INVALID_ARG	参数错误
ESP_FAIL	配置错误
ESP_ERR_NO_MEM	如果无法分配内存
ESP_ERR_NOT_FOUND	如果分区表不包含带有给定标签的 FATFS 分区

表 27.3.2.2 函数 esp\_vfs\_fat\_spiflash\_mount\_rw\_wl() 返回值描述

该函数使用 `esp_vfs_fat_mount_config_t` 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
<b>esp_vfs_fat_mount_config_t</b>	format_if_mount_failed	如果不能挂载 FAT 分区，此参数为 true，创建分区表并格式化文件系统。
	max_files	打开文件的最大数目
	allocation_unit_size	将此字段设置为 0 将导致分配单元设置为扇区大小

表 27.3.2.3 esp\_vfs\_fat\_mount\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 `esp_vfs_fat_spiflash_mount_rw_wl()` 函数，用以挂载分区。

更多有关 USB 函数的介绍，请读者们回顾上一章节的内容。

### 26.3.3 USB 虚拟串口驱动解析

USB 驱动库移植，我们可参考第九章 IDF 组件注册表的形式添加 USB 驱动库。下图为添加 USB 驱动库的工程结构。

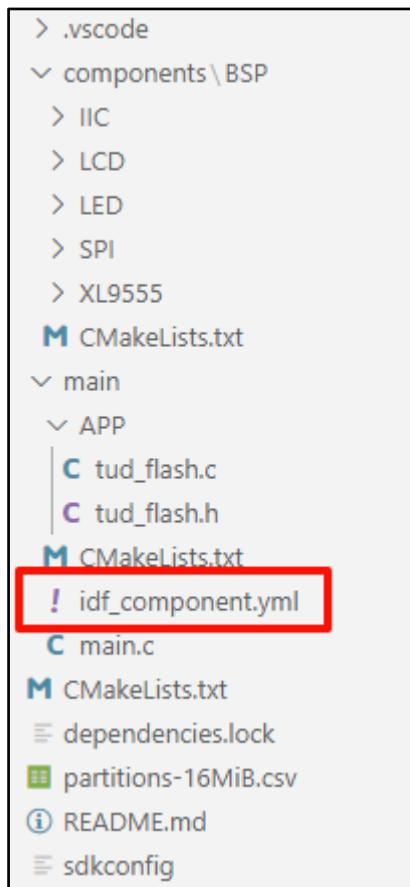


图 25.3.3.1 Flash 模拟 U 盘工程分组

上图中位于 components 文件夹下的是我们自己编写的一些外设驱动，main 文件夹下包含了一个 APP 文件与一个后缀为.yml 的文件。APP 文件夹下包含的是 FLASH 模拟 U 盘（USB）代码，而后缀为.yml 的文件其主要作用是将项目中各组件的依赖项定义在单独的清单文件中，并以上图所示的方式进行命名。在我们的例程中呈现出的作用就是简化了整个工程结构。我们在编译的过程中，系统便会帮我们自动生成 USB 外设所需要的依赖库：`espressif_esp_tinyusb` 以及 `espressif_tinyusb`。做到了既能简化项目工程，又能有效规避了在编译中遇到的错误，但前提是运行时得确保个人的电脑处于联网状态。

### 26.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```
set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)

set(requires
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
    INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

该路径下的 CmakeList 文件并没有新增内容，主要变化在于 main 文件。

打开本实验 main 文件下的 CMakeLists.txt 文件，其内容如下所示：

```
idf_component_register(
    SRC_DIRS
        "."
        "app"
    INCLUDE_DIRS
        "."
        "app")
```

上述的红色 app 驱动需要由开发者自行添加，以确保 USB 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 USB 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

### 26.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    ESP_ERROR_CHECK(ret);

    led_init(); /* 初始化 LED */
    spi2_init(); /* 初始化 SPI */
    lcd_init(); /* 初始化 LCD */

    /* 显示实验信息 */
    lcd_show_string(0, 0, 200, 16, 16, "ESP32-S3", RED);
```

```

lcd_show_string(0, 20, 200, 16, 16, "USB FLASH TEST", RED);
lcd_show_string(0, 40, 200, 16, 16, "ATOM@ALIENTEK", RED);
tud_usb_flash(); /* USB 初始化 */

while(1)
{
    if ((g_usbdev.status & 0x0f) == 0x01)
    {
        lcd_show_string(0, 60, lcd_self.width, 16, 16,
        "connect success.....", BLUE);
    }
    else if ((g_usbdev.status & 0x0f) == 0x00)
    {
        lcd_show_string(0, 60, lcd_self.width, 16, 16,
        "connect fail.....", BLUE);
    }
    LED_TOGGLE();
    vTaskDelay(500);
}
}

```

此部分代码比较简单，通过 `tud_usb_flash()` 等函数初始化 USB。由于该实验例程需要系统将 storage 分区模拟成 U 盘，所以在该函数中需要初始化 SPIFFS 分区，其次是用 USB 设备安装函数，用以 USB 设备登记。同时，LCD 显示实验信息，LED 闪烁以示程序正在运行。

## 26.4 下载验证

将程序下载到 DNESP32S3M 最小系统板后（注意：USB 数据线，要插在 USB 端口！而不是 UART 端口！），我们打开设备管理器（我用的是 WIN10），在端口（COM 和 LPT）里面可以发现多出了一个 COM25 的设备，这就是 USB 虚拟的串口设备端口，如下图所示：



图 26.4.1 通过设备管理器查看 USB 虚拟的串口设备端口

如图 26.4.1，ESP32 通过 Flash 模拟 U 盘，被电脑识别了，通用串行总线控制器显示的是：USB 大容量存储设备（其实也不算大，也就差不多 4MB...）。此时，DNESP32S3M 最小系统板的 LED 在闪烁，提示程序运行。DNESP32S3M 最小系统板的 LCD 显示“connect success.....”，如下图所示：

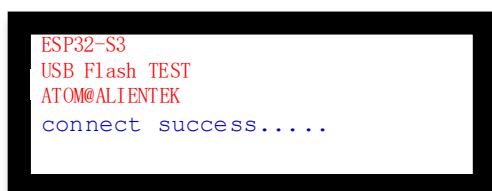


图 26.4.2 USB 虚拟串口连接成功

然后我们打开“我的电脑”，可以看见界面显示了通过 Flash 模拟 U 盘后的容量大小，如下图所示：

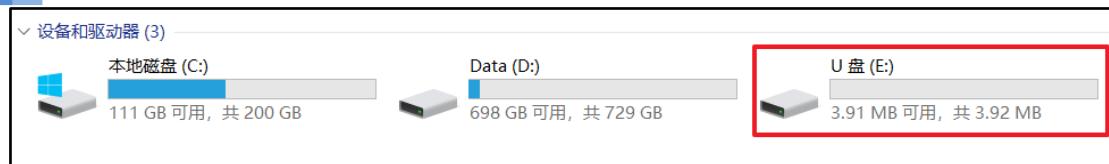


图 26.4.3 ESP32 Flash 模拟 U 盘测试

至此，Flash 模拟 U 盘实验就完成了，通过本实验，我们就可以利用 ESP32 的 Flash 进行 U 盘模拟。

## 第二十七章 SD 卡模拟 U 盘实验

本章我们介绍 ESP32S3 的 USB HOST 应用，即通过 USB HOST 功能，将某个分区表实现模拟 U 盘/读卡器等大容量 USB 存储设备。

本章分为以下几个小节：

27.1 Flash 模拟 U 盘简介

27.2 硬件设计

27.3 程序设计

27.4 下载验证

### 27.1 Flash 模拟 U 盘简介

U 盘，全称 USB 闪存盘，英文名“USB flash disk”。它是一种使用 USB 接口的无需物理驱动器的微型高容量移动存储产品，通过 USB 接口与主机连接，实现即插即用，是最常用的移动存储设备之一。

ESP32-S3 的 USB OTG FS 支持 U 盘，并且乐鑫官方提供了 esp\_tinyusb 开发库，我们可使用 IDF 组件注册的方式把该库移植至项目工程中。

### 27.2 硬件设计

#### 27.2.1 例程功能

本实验通过 USB 把 SD 卡模拟成 U 盘，并实现 U 盘读写操作。

LED 闪烁，提示程序运行，USB 和电脑连接成功。

#### 27.2.2 硬件资源

1. LED 灯  
LED -IO0
2. 0.96 寸 LCD
3. SD  
CS-IO2  
SCK-IO12  
MOSI-IO11  
MISO-IO13
4. USB

#### 27.2.3 原理图

USB 原理图已在 25.2.3 章节中详细阐述，为避免重复，此处不再赘述。

### 27.3 程序设计

#### 27.3.1 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：

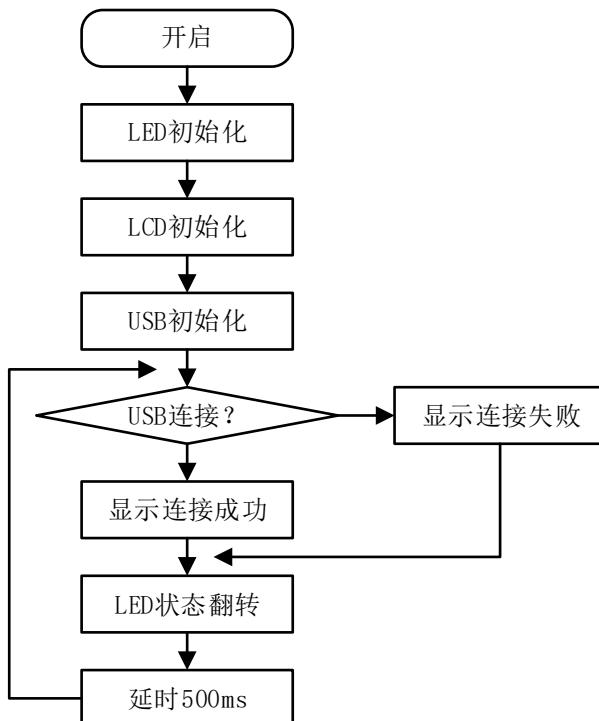


图 27.3.1.1 SD 卡模拟 U 盘实验程序流程图

### 27.3.2 SD 卡模拟 U 盘函数解析

ESP-IDF 提供了一套 API 来配置 Flash。要使用此功能，需要导入必要的头文件：

```
#include "ff.h"
#include "diskio.h"
#include "esp_vfs_fat.h"
#include "tinyusb.h"
```

接下来，作者将介绍一些常用的 ESP32-S3 中的 Flash 函数，这些函数的描述及其作用如下：

#### 1. 挂载分区函数

该函数用给定的配置，来挂载分区，该函数原型如下所示：

```
esp_err_t esp_vfs_fat_spiflash_mount_rw_wl(const char* base_path,
                                              const char* partition_label,
                                              const esp_vfs_fat_mount_config_t* mount_config,
                                              wl_handle_t* wl_handle);
```

该函数的形参描述如下表所示：

参数	描述
base_path	应该挂载 FATFS 分区的路径
partition_label	应该使用的分区的标签
mount_config	指向带有附加参数的结构的指针，用于挂载 FATFS
wl_handle	磨损均衡驱动句柄

表 27.3.2.1 esp\_vfs\_fat\_spiflash\_mount\_rw\_wl() 函数形参描述

该函数的返回值描述，如下表所示：

返回值	描述
ESP_OK	返回：0，配置成功
ESP_ERR_INVALID_ARG	参数错误
ESP_FAIL	配置错误
ESP_ERR_NO_MEM	如果无法分配内存
ESP_ERR_NOT_FOUND	如果分区表不包含带有给定标签的 FATFS 分区

表 27.3.2.2 函数 esp\_vfs\_fat\_spiflash\_mount\_rw\_wl() 返回值描述

该函数使用 esp\_vfs\_fat\_mount\_config\_t 类型的结构体变量传入，该结构体的定义如下所示：

结构体	成员变量	可选参数
esp_vfs_fat_mount_config_t	format_if_mount_failed	如果不能挂载 FAT 分区，此参数为 true，创建分区表并格式化文件系统。
	max_files	打开文件的最大数目
	allocation_unit_size	将此字段设置为 0 将导致分配单元设置为扇区大小

表 27.3.2.3 esp\_vfs\_fat\_mount\_config\_t 结构体参数值描述

完成上述结构体参数配置之后，可以将结构传递给 esp\_vfs\_fat\_spiflash\_mount\_rw\_wl() 函数，用以挂载分区。

更多有关 USB 函数的介绍，请读者们回顾上一章节的内容。

### 27.3.3 SD 卡模拟 U 盘驱动解析

USB 驱动库移植，我们可参考第九章 IDF 组件注册表的形式添加 USB 驱动库。下图为添加 USB 驱动库的工程结构。

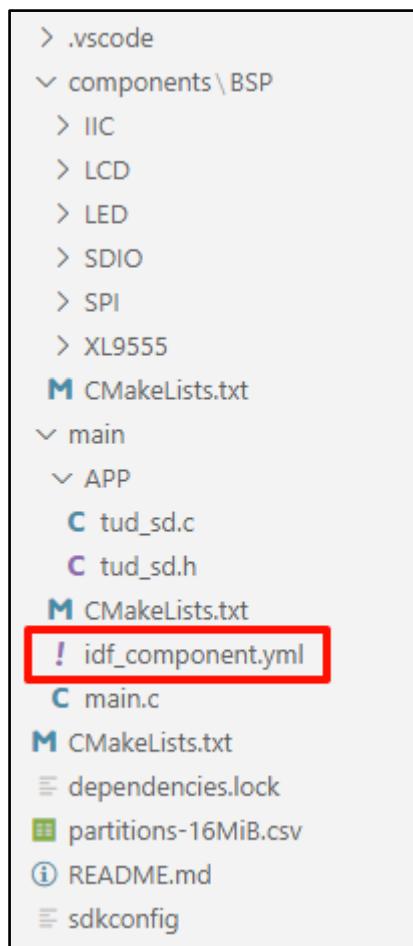


图 27.3.3.1 SD 卡模拟 U 盘工程分组

上图中位于 components 文件夹下的是我们自己编写的一些外设驱动，main 文件夹下包含了一个 APP 文件与一个后缀为.yml 的文件。APP 文件夹下包含的是 SD 卡模拟 U 盘（USB）代码，而后缀为.yml 的文件其主要作用是将项目中各组件的依赖项定义在单独的清单文件中，并以上

图所示的方式进行命名。在我们的例程中提现出来的作用就是简化了整个工程结构。我们在编译的过程中，系统便会帮我们自动生成 USB 外设所需要的依赖库：espressif\_esp\_tinyusb 以及 espressif\_tinyusb。做到了即能简化项目工程，又能有效规避了在编译中遇到的错误，但前提是运行时得确保个人的电脑处于联网状态。

#### 27.3.4 CMakeLists.txt 文件

打开本实验 BSP 下的 CMakeLists.txt 文件，其内容如下所示：

```
set(src_dirs
    LCD
    LED
    SPI)

set(include_dirs
    LCD
    LED
    SPI)

set(requires
    driver)

idf_component_register(SRC_DIRS ${src_dirs}
    INCLUDE_DIRS ${include_dirs} REQUIRES ${requires})

component_compile_options(-ffast-math -O3 -Wno-error=format=-Wno-format)
```

该路径下的 CmakeList 文件并没有新增内容，主要变化在于 main 文件。

打开本实验 main 文件下的 CMakeLists.txt 文件，其内容如下所示：

```
idf_component_register(
    SRC_DIRS
    "."
    "app"
    INCLUDE_DIRS
    "."
    "app")
```

上述的红色 app 驱动需要由开发者自行添加，以确保 USB 驱动能够顺利集成到构建系统中。这一步骤是必不可少的，它确保了 USB 驱动的正确性和可用性，为后续的开发工作提供了坚实的基础。

#### 27.3.5 实验应用代码

打开 main/main.c 文件，该文件定义了工程入口函数，名为 app\_main。该函数代码如下。

```
/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    esp_err_t ret;

    ret = nvs_flash_init(); /* 初始化 NVS */

    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }

    ESP_ERROR_CHECK(ret);

    led_init(); /* 初始化 LED */
```

```

spi2_init();                                /* 初始化 SPI */
lcd_init();                                 /* 初始化 LCD */

/* 显示实验信息 */
lcd_show_string(30, 0, 200, 16, 16, "ESP32-S3", RED);
lcd_show_string(30, 20, 200, 16, 16, "USB SD TEST", RED);
lcd_show_string(30, 40, 200, 16, 16, "ATOM@ALIENTEK", RED);

while (sd_spi_init())                      /* 检测不到 SD 卡 */
{
    lcd_show_string(0, 60, 200, 16, 16, "SD Card Error!", RED);
    vTaskDelay(500);
    lcd_show_string(0, 60, 200, 16, 16, "Please Check! ", RED);
    vTaskDelay(500);
}

tud_usb_sd();                             /* USB 初始化 */

while(1)
{
    if ((g_usbdev.status & 0x0f) == 0x01)
    {
        lcd_show_string(0, 60, lcd_self.width, 16, 16,
"connect success.....", BLUE);
    }
    else if ((g_usbdev.status & 0x0f) == 0x00)
    {
        lcd_show_string(0, 60, lcd_self.width, 16, 16,
"connect fail.....", BLUE);
    }

    LED_TOGGLE();
    vTaskDelay(500);
}
}

```

此部分代码比较简单，通过 `tud_usb_sd()` 等函数初始化 USB。由于该实验例程需要系统将 `storage` 分区模拟成 U 盘，所以在该函数中需要初始化 SPIFFS 分区，其次是用 USB 设备安装函数，用以 USB 设备登记。同时，LCD 显示实验信息，LED 闪烁以示程序正在运行。

## 27.4 下载验证

将程序下载到 DNESP32S3M 最小系统板后（注意：USB 数据线，要插在 USB 端口！而不是 UART 端口！），我们打开设备管理器（我用的是 WIN10），在端口（COM 和 LPT）里面可以发现多出了一个 COM25 的设备，这就是 USB 虚拟的串口设备端口，如下图所示：



图 27.4.1 通过设备管理器查看 USB 虚拟的串口设备端口

上图中，ESP32 通过 SD 卡模拟 U 盘，被电脑识别了，通用串行总线控制器显示的是：USB 大容量存储设备（其实也不算大，也就差不多 4MB...）。此时，DNESP32S3M 最小系统板的 LED 在闪烁，提示程序运行。DNESP32S3M 最小系统板的 LCD 显示“connect success.....”，如下图所示：



图 27.4.2 USB 虚拟串口连接成功

至此，SD 卡模拟 U 盘实验就完成了，通过本实验，我们就可以利用 ESP32 的 USB，直接和电脑进行数据互传了，具有广泛的应用前景。

# 第三篇 高级篇

通过基础篇和入门篇的学习，我们掌握了在 IDF 版 VSCode 环境下开发 ESP32-S3 芯片的基本知识。接下来，我们将进入更加实用的内容，包括 WiFi 和 AI 的应用。通过本篇的学习，我们将深入了解 ESP32-S3 芯片在 IOT 和 AI 开发中的简单易用性。

- 1, lwIP 初探
- 2, 扫描 WiFi 实验
- 3, WiFi 路由实验
- 4, WiFi 热点实验
- 5, WiFi 一键配网
- 6, UDP 实验
- 7, TCPClient 实验
- 8, TCPServer 实验

## 第二十八章 lwIP 初探

ESP32-S3 是一款集成了 Wi-Fi 和蓝牙功能的微控制器，而 lwIP（轻量级 IP）是一个为嵌入式系统设计的开源 TCP/IP 协议栈。通过使用 lwIP 库，ESP32-S3 可以实现与外部网络的通信，包括发送和接收数据包、处理网络连接等。因此，ESP32-S3 是基于 lwIP 来实现网络功能的。

本章将分为如下几个部分：

28.1 TCP/IP 协议栈是什么

28.2 lwIP 简介

28.3 WiFi MAC 内核简介

28.4 lwIP Socket 编程接口

### 28.1 TCP/IP 协议栈是什么

TCP/IP 协议栈是一系列网络协议的总和，构成网络通信的核心骨架，定义了电子设备如何连入因特网以及数据如何在它们之间进行传输。该协议采用 4 层结构，分别是应用层、传输层、网络层和网络接口层，每一层都使用下一层提供的协议来完成自己的需求。由于大部分时间都在应用层工作，下层的事情不用操心。此外，网络协议体系本身很复杂庞大，入门门槛高，因此很难搞清楚 TCP/IP 的工作原理。如果读者想深入了解 TCP/IP 协议栈的工作原理，建议阅读《计算机网络书籍》。

#### 28.1.1 TCP/IP 协议栈架构

TCP/IP 协议栈是一个分层结构的模型，每一层负责不同的网络功能。整个协议栈可以被分为四层，从上到下分别是：应用层、传输层、网络层和网络接口层。

1，应用层：这是最顶层，负责处理特定的应用程序细节。在这一层，用户的数据被处理和解释。一些常见的应用层协议包括 HTTP、FTP、SMTP 和 DNS 等。

2，传输层：这一层负责数据包的分割、打包以及传输控制，确保数据能够可靠、有序地到达目的地。主要的传输层协议有 TCP 和 UDP。

3，网络层：负责确定数据包的路径从源到目的地。这一层的主要协议是 IP (Internet Protocol)，它负责在主机之间发送和接收数据包。

4，网络接口层：这是最底层，负责将数据转换为可以在物理媒介上发送的信号。这一层的协议涉及到如何将数据帧封装在数据链路层，以便在网络上进行传输。

每一层都使用下一层提供的服务，同时对上一层提供服务。这种分层结构使得协议栈更加灵活，易于扩展和维护。不同层次上的协议一起工作，协调数据在计算机网络中的传输，使得不同的计算机能够相互通信。

需要注意的是，TCP/IP 协议栈和传统的 OSI 模型并不完全对应。TCP/IP 协议栈是一个简化的模型，强调了实际的协议实现和因特网的实际运作方式。相比之下，OSI 模型更加全面和理想化，它提供了一个框架来描述不同系统之间的交互方式。下图是 IOS 协议栈与 TCP/IP 协议栈分层架构的对比图。

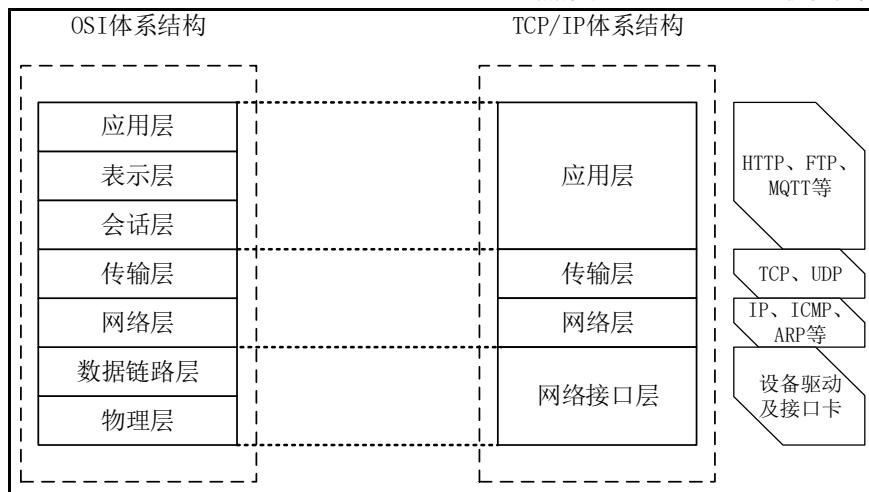


图 28.1.1.1 TCP/IP 协议栈的分层结构

ISO/OSI 分层模型也是一个分层结构，包括七个层次，从上到下分别是：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。虽然 ISO/OSI 模型为不同的系统之间的通信提供了一个理论框架，但 TCP/IP 协议栈更侧重于实际的协议实现和因特网的实际运作方式。注意：网络技术的发展并不是遵循严格的 ISO/OSI 分层概念。实际上现在的互联网使用的是 TCP/IP 体系结构，有时已经演变成为图 1.1.1.2 所示那样，即某些应用程序可以直接使用 IP 层，或甚至直接使用最下面的网络接口层。

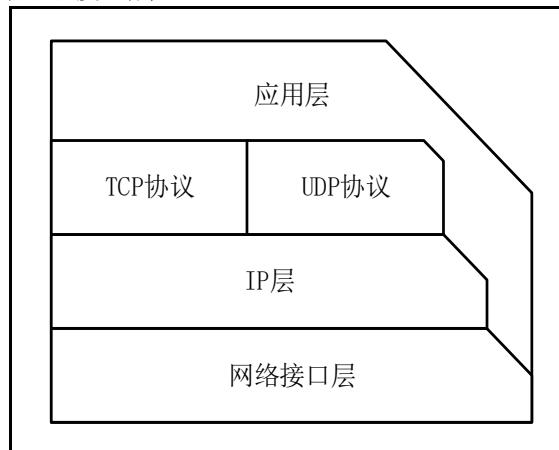


图 28.1.1.2 TCP/IP 体系结构另一种表示方法

无论那种表示方法，TCP/IP 模型各个层次都分别对应于不同的协议。TCP/IP 协议栈负责确保网络设备之间能够通信。它是一组规则，规定了信息如何在网络中传输。其中，这些协议都分布在应用层，传输层和网络层，网络接口层是由硬件来实现。如 Windows 操作系统包含了 CBISC 协议栈，该协议栈就是实现了 TCP/IP 协议栈的应用层，传输层和网络层的功能，网络接口层由网卡实现，所以 CBISC 协议栈和网卡构建了网络通信的核心骨架。因此，无论哪一款以太网产品，都必须符合 TCP/IP 体系结构，才能实现网络通信。注意：路由器和交换机等相关网络设备只实现网络层和网络接口层的功能。

## 28.1.2 TCP/IP 协议栈的封装和拆包

TCP/IP 协议栈的封装和拆包是指在网络通信中，将数据按照一定的协议和格式进行封装和解析的过程。

在 TCP/IP 协议栈中，数据封装是指在发送端将数据按照协议规定的格式进行打包，以便在网络中进行传输。在应用层的数据被封装后，会经过传输层、网络层和网络接口层的处理，最终转换成可以在物理网络上传输的帧格式。数据封装的过程涉及到对数据的分段、压缩、加密等操作，以确保数据能够可靠、安全地传输到目的地，下图描述的是封装处理流程。

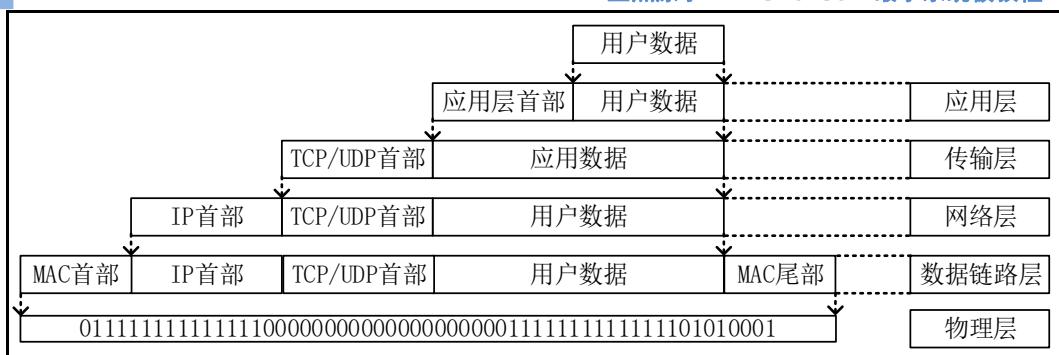


图 28.1.2.1 TCP/IP 协议栈的封包

数据拆包是指接收端收到数据后，按照协议规定的格式对数据进行解析和处理，还原出原始的数据。在接收到数据后，接收端会按照协议规定的层次从下往上逐层处理数据，最终将应用层的数据还原出来。数据拆包的过程涉及到对数据的重组、解压缩、解密等操作，以确保数据能够被正确地解析和处理，下图描述的是拆包处理流程。

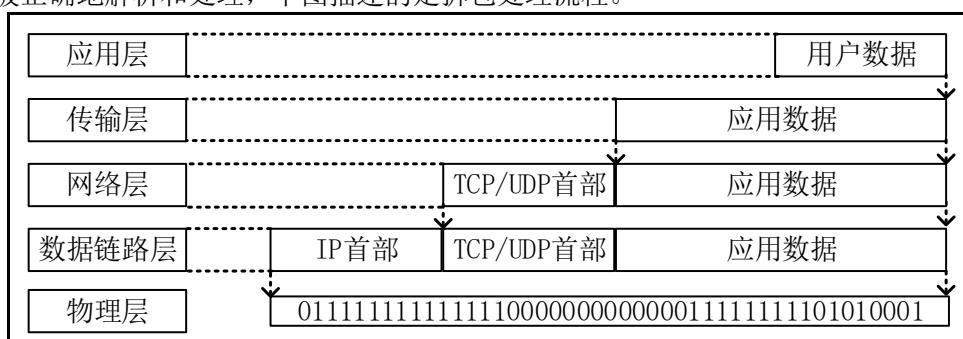


图 28.1.2.2 TCP/IP 协议栈的拆包

需要注意的是，TCP/IP 协议栈的封包和拆包过程涉及到多个层次和协议的处理，需要按照协议规定的格式和顺序进行操作。在实际应用中，需要根据具体的情况选择合适的协议和格式来满足不同的需求。同时，为了保证数据的安全和可靠性，还需要采取相应的加密、压缩等措施，以避免数据被篡改或损坏。

## 28.2 lwIP 简介

lwIP，全称为 Lightweight IP 协议，是一种专为嵌入式系统设计的轻量级 TCP/IP 协议栈。它可以在无操作系统或带操作系统环境下运行，支持多线程或无线程，适用于 8 位和 32 位微处理器，同时兼容大端和小端系统。它的设计核心理念在于保持 TCP/IP 协议的主要功能同时尽量减少对 RAM 的占用。这意味着，尽管它的体积小巧，但它能够实现完整的 TCP/IP 通信功能。通常，lwIP 只需十几 KB 的 RAM 和大约 40K 的 ROM 即可运行，使其成为资源受限的嵌入式系统的理想选择。lwIP 的灵活性使其既可以在无操作系统环境下工作，也可以与各种操作系统配合使用。这为开发者提供了更大的自由度，可以根据具体的应用需求和硬件配置进行优化。无论是在云台接入、无线网关、远程模块还是工控控制器等场景中，lwIP 都能提供强大的网络支持。

### 一、lwIP 特性参数

lwIP 的各项特性，如下表所示：

项目	说明	
内存占用	lwIP 在内存占用方面进行了优化，使其适用于资源受限的嵌入式系统。具体内存占用取决于配置和版本，但通常来说，它可以仅使用十几 KB 的 RAM 和大约 40K 的 ROM 即可运行	
协议实现	应用层	未实现，需要开发者自行集成或实现特定应用程序
	传输层	实现了 TCP/UDP 协议功能
	网络层	实现了 IP 协议、ARP 协议和 ICMP 协议功能
扩展性	lwIP 支持一些实验性的扩展，例如 IPv6 协议	
硬件接口	WIFI 网络接口、以太网网络接口...	

用户编程接口	RAW、NETCONN、SOCKET 用户编程接口
硬件支持	网络接口层的实现依赖于底层硬件和驱动程序。开发者需要根据具体的硬件平台和需求进行相应的配置和优化
其他特性	支持 PPP 协议（点对点通信协议）和 DHCP 协议（动态分配 IP 地址）

表 28.2.1 lwIP 基本特性

## 二、lwIP 与 TCP/IP 体系结构的对应关系

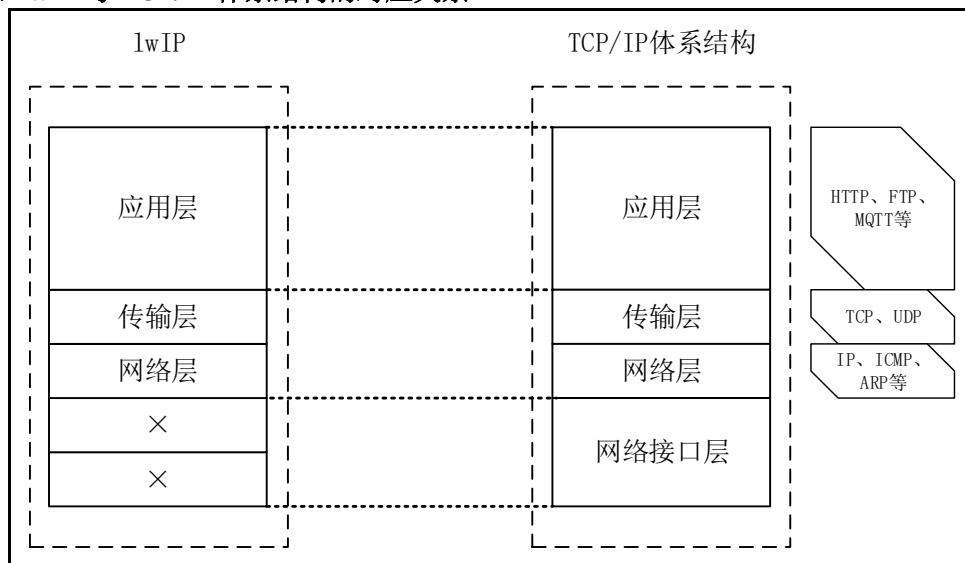


图 28.2.1 lwIP 与 TCP/IP 体系结构的对应图解

从上图可以清晰地看到，lwIP 软件库主要实现了 TCP/IP 体系结构中的三个层次：应用层、传输层和网络层。这些层次共同处理和传输数据包，确保了数据在网络中的可靠传输。然而，网络接口层作为 TCP/IP 协议栈的最底层，其功能并无法通过软件方式完全实现。这一层的主要任务是将数据包转换为光电模拟信号，以便能够在物理媒介上传输。这个过程涉及到与硬件的直接交互，包括数据的调制解调、信号的转换等，这些都是软件难以模拟或实现的。因此，虽然 lwIP 软件库没有实现网络接口层的功能，但通过与底层硬件的紧密配合，它仍然能够提供完整且高效的 TCP/IP 通信功能。这也使得 lwIP 成为一个适用于资源受限的嵌入式系统的理想选择。

在开发过程中，开发者需要根据具体的硬件平台和需求进行相应的配置和优化，以确保 lwIP 能够与硬件完美配合，发挥出最佳的性能。

## 28.3 WiFi MAC 内核简介

ESP32-S3 完全遵循 802.11b/g/n WiFi MAC 协议栈，支持分布式控制功能（DCF）下的基本服务集（BSS）STA 和 SoftAP 操作。支持通过最小化主机交互来优化有效工作时长，以实现功耗管理。ESP32-S3 WiFi MAC 支持 4 个虚拟 WiFi 接口，同时支持基础结构型网络、SoftAP 模式和 Station+SoftAP 混杂模式。它还具备 RTS 保护、CTS 保护、立即块确认、分片和重组、TX/RX A-MPDU 和 TX/RX A-MSDU 等高级功能。此外，ESP32-S3 还支持无线多媒体、GCMP、CCMP、TKIP、WAPI 等安全协议，并提供自动 Beacon 监测和 802.11mc FTM 支持。

关于 ESP32 的 WiFi MAC 内核，官方没有提供更多学习资料。读者只需了解它扮演 TCP/IP 协议的网络接口层角色即可。下面作者使用一张示意图来介绍 WiFi 通讯示意图，如下图所示。

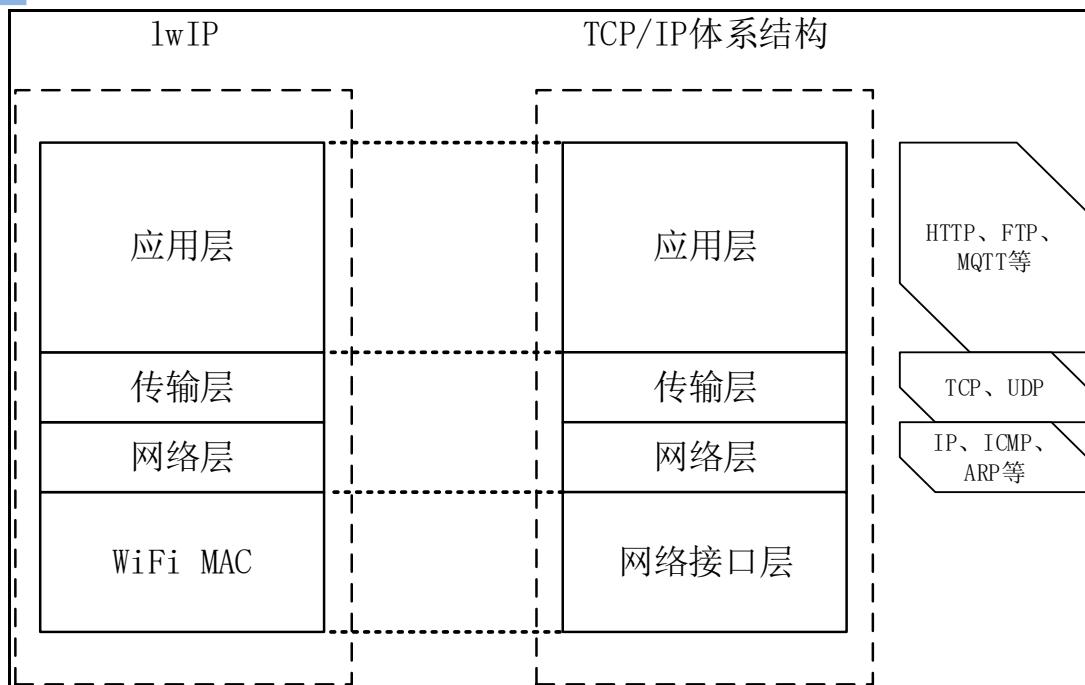


图 28.3.1 ESP32-S3 网络层次示意图

从上图可以看出，ESP32-S3 芯片内置 WiFi MAC 内核。当我们发送数据到网络时，数据首先被转化为无线信号，然后发送到该设备连接的 WiFi 路由器中。接着，路由器通过网线将数据传输到目标主机，从而完成数据传输操作。以下是作者对于无线网络传输的描述。

1，数据转化为无线信号：当 ESP32-S3 想要发送数据到网络时，它首先会将数据封装到一个无线传输帧中。这一过程涉及到将数据转化为可以在无线介质上传输的格式。

2，发送到 WiFi 路由器：封装后的无线信号然后被发送到 ESP32-S3 连接的 WiFi 路由器。WiFi 路由器充当一个中间设备，负责将无线信号转换为有线网络信号（如果目标主机是通过有线网络连接的）或直接转发无线信号（如果目标主机也是通过 WiFi 连接的）。

3，路由器传输数据：WiFi 路由器接收到无线信号后，会进一步处理它。如果目标主机是通过有线网络连接的，路由器会将无线信号转换为有线网络信号，并通过网线将其传输到目标主机。如果目标主机也是通过 WiFi 连接的，路由器会直接转发无线信号到目标主机。

4，完成数据传输：最终，目标主机接收到路由器发送的有线网络信号或无线信号，并将其解析为原始数据。这样，整个数据传输过程就完成了。

在整个过程中，ESP32-S3 的 WiFi MAC 内核起着核心的作用，它负责管理无线连接、封装和解封装数据以及与 WiFi 路由器进行通信。

## 28.4 lwIP Socket 编程接口

lwIP 作者为了方便开发者将其他平台上的网络应用程序移植到 lwIP 上，并让更多开发者快速上手 lwIP，作者设计了三种应用程序编程接口：RAW 编程接口、NETCONN 编程接口和 Socket 编程接口。然而，由于 RAW 编程接口只能在无操作系统环境下运行，因此对于内嵌 FreeRTOS 操作系统的 ESP32 来说，无法使用这个编程接口。尽管 Socket 编程接口是由 NETCONN 编程接口封装而成，但是该接口非常简易的实现网络连接（作者推荐使用此接口）。需要注意的是，由于受到嵌入式处理器资源和性能的限制，部分 Socket 接口并未在 lwIP 中完全实现。因此，为了实现网络连接，推荐使用 Socket API。

下面作者简单介绍一下 lwIP Socket 编程接口常用的 API 函数。这些 API 函数如下所示。

### (1) socket 函数

该函数的原型，如下源码所示：

```
#define socket(domain,type,protocol)          lwip_socket(domain,type,protocol)
向内核申请一个套接字，本质上该函数调用了函数 lwip_socket，该函数的参数如下表所示：
```

参数	描述
domain	创建的套接字指定使用的协议簇 AF_INET 表示 IPv4 网络协议 AF_INET6 表示 IPv6 AF_UNIX 表示本地套接字（使用一个文件）
	协议簇中的具体服务类型 SOCK_STREAM（可靠数据流交付服务，比如 TCP） SOCK_DGRAM（无连接数据报交付服务，比如 UDP） SOCK_RAW（原始套接字，比如 RAW）
protocol	实际使用的具体协议（常见的有 IPPROTO_TCP、IPPROTO_UDP 等，若设置为"0"，表示根据前两个参数使用缺省协议）

表 28.4.1 函数 Socket() 相关形成描述

## (2) bind 函数

该函数的原型，如下源码所示：

```
#define bind(s,name,namelen)      lwip_bind(s,name,namelen)
int bind(int s, const struct sockaddr *name, socklen_t namelen)
```

该函数与 netconn\_bind 函数一样，用于服务器端绑定套接字与网卡信息，本质上就是对函数 netconn\_bind 再一次封装，从上述源码可以知道参数 name 指向一个 sockaddr 结构体，它包含了本地 IP 地址和端口号等信息；参数 namelen 指出结构体的长度。结构体 sockaddr 定义如下源码所示：

```
struct sockaddr {
    u8_t          sa_len;           /* 长度 */
    sa_family_t   sa_family;        /* 协议簇 */
    char          sa_data[14];       /* 连续的 14 字节信息 */
};

struct sockaddr_in {
    u8_t          sin_len;           /* 长度 */
    u8_t          sin_family;        /* 协议簇 */
    u16_t         sin_port;          /* 端口号 */
    struct in_addr sin_addr;        /* IP 地址 */
    char          sin_zero[8];
};
```

可以看出，lwIP 作者定义了两个结构体，结构体 sockaddr 中的 sa\_family 指向该套接字所使用的协议簇，本地 IP 地址和端口号等信息在 sa\_data 数组里面定义，这里暂未用到。由于 sa\_data 以连续空间的方式存在，所以用户要填写其中的 IP 地段和端口 port 地段，这样会比较麻烦，因此 lwIP 定义了另一个结构体 sockaddr\_in，它与 sockaddr 结构对等，只是从中抽出 IP 地址和端口号 port，方便于用于的编程操作。

## (3) connect 函数

该函数与 netconn 接口的 netconn\_connect 函数作用是一样的，因此它是被 netconn\_connect 函数封装了，该函数的作用是将 Socket 与远程 IP 地址和端口号绑定，如果 DNESP32S3M 最小系统板作为客户端，通常使用这个函数来绑定服务器的 IP 地址和端口号，对于 TCP 连接，调用这个函数会使客户端与服务器之间发生连接握手过程，并建立稳定的连接；如果是 UDP 连接，该函数调用不会有任何数据包被发送，只是在连接结构中记录下服务器的地址信息。当调用成功时，函数返回 0；否则返回-1。该函数的原型如下源码所示：

```
#define connect(s,name,namelen)    lwip_connect(s,name,namelen)
int lwip_connect(int s, const struct sockaddr *name, socklen_t namelen);
```

## (4) listen 函数

该函数和 netconn 的函数 netconn\_listen 作用一样，它是由函数 netconn\_listen 封装得来的，内核同时接收到多个连接请求时，需要对这些请求进行排队处理，参数 backlog 指明了该套接字上连接请求队列的最大长度。当调用成功时，函数返回 0；否则返回-1。该函数的原型如下源码所示：

```
#define listen(s,backlog)    lwip_listen(s,backlog)
int lwip_listen(int s, int backlog);
```

**注意：**该函数作用于 TCP 服务器程序。

### (5) accept 函数

该函数与 netconn\_accept 作用是一样的，当接收到新连接后，连接另一端（客户端）的地址信息会被填入到地址结构 addr 中，而对应地址信息的长度被记录到 addrlen 中。函数返回新连接的套接字描述符，若调用失败，函数返回-1。该函数的原型如下源码所示：

```
#define accept(s,addr,addrlen) lwip_accept(s,addr,addrlen)
int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

**注意：**该函数作用于 TCP 服务器程序。

### (6) send()/sendto()函数

该函数是被 netconn\_send 封装的，其作用是向另一端发送 UDP 报文，这两个函数的原型如下源码所示：

```
#define send(s,dataptr,size,flags) lwip_send(s,dataptr,size,flags)
#define sendto(s,dataptr,size,flags,to,tolen)
                           lwip_sendto(s,dataptr,size,flags,to,tolen)
ssize_t lwip_send(int s, const void *dataptr, size_t size, int flags);
ssize_t lwip_sendto(int s, const void *dataptr, size_t size, int flags,
                   const struct sockaddr *to, socklen_t tolen);
```

可以看出，函数 sendto 比函数 send 多了两个参数，该函数如下表所示：

参数	描述
s	Socket 接口
dataptr	发送数据的起始地址
size	长度
flags	数据发送时的特殊处理，例如带外数据、紧急数据等，通常设置为 0
to(sendto())	目的地址信息
tolen(sendto())	信息的长度

表 28.4.2 函数 sendto() 和 send() 形参描述

### (7) write 函数

该函数用于在一条已经建立的连接上发送数据，通常使用在 TCP 程序中，但在 UDP 程序中也能使用。该函数本质上是基于前面介绍的 send 函数来实现的，其参数的意义与 send 也相同。当函数调用成功时，返回成功发送的字节数；否则返回-1。

### (8) read()/recv()/recvfrom()函数

函数 recvfrom 和 recv 用来从一个套接字中接收数据，该函数可以在 UDP 程序使用，也可在 TCP 程序中使用。该函数本质上是被函数 netconn\_recv 的封装，其参数与函数 sendto 的参数完全相似，如下表所示，数据发送方的地址信息会被填写到 from 中，fromlen 指明了缓存 from 的长度；mem 和 len 分别记录了接收数据的缓存起始地址和缓存长度，flags 指明用户控制接收的方式，通常设置为 0。两个函数的原型如下源码所示：

```
#define recv(s,mem,len,flags) lwip_recv(s,mem,len,flags)
#define recvfrom(s,mem,len,flags,from,fromlen)
                           lwip_recvfrom(s,mem,len,flags,from,fromlen)
ssize_t lwip_readv(int s, const struct iovec *iov, int iovcnt);
ssize_t lwip_recvfrom(int s, void *mem, size_t len, int flags,
                     struct sockaddr *from, socklen_t *fromlen);
#define read(s,mem,len) lwip_read(s,mem,len)
ssize_t lwip_read(
    int s,
    void *mem, size_t len);
```

参数	描述
s	Socket 接口
mem	接收数据的缓存起始地址
len	缓存长度
flags	用户控制接收的方式，通常设置为 0
from(recvfrom())	发送方的地址信息
fromlen(recvfrom())	缓存 from 的长度

表 28.4.3 函数 recv() 和 recvfrom() 形参描述

### (9) close 函数

函数 `close` 作用是关闭套接字，对应的套接字描述符不再有效，与描述符对应的内核结构 `lwip_socket` 也将被全部复位。该函数本质上是被 `netconn_delete` 的封装，对于 TCP 连接来说，该函数将导致断开握手过程的发生。若调用成功，该函数返回 0；否则返回-1。该函数的原型如下源码所示：

```
#define close(s)      lwip_close(s)
int lwip_close(int s);
```

## 第二十九章 扫描 WiFi 实验

ESP32-S3 的 WiFi 库支持配置及监控 ESP32-S3 的 Wi-Fi 连网功能。它支持配置基站模式（即 STA 模式或 WiFi 客户端模式），此时 ESP32-S3 连接到接入点（AP）。还支持 AP 模式（即 Soft-AP 模式或接入点模式），此时基站连接到 ESP32-S3。同时，支持 AP-STA 共存模式，此时 ESP32-S3 既是接入点，同时又作为 STA。本章节的实验是基于乐鑫官方提供的 WiFi 库来实现的，但遗憾的是乐鑫并没有公开 WiFi 库的源码，所以我们只能调用 API 函数实现。

本章分为以下几个小节：

29.1 WiFi 模式概述

29.2 硬件设计

29.3 软件设计

29.4 下载验证

### 29.1 WiFi 模式概述

WiFi 主要有两种模式：STA 和 AP 模式。AP 模式即无线接入点，是我们常说的手机热点，被其他设备连接；STA 模式即 Station，是连接热点的设备。另外，ESP32S3 可支持 STA 和 AP 两种模式共存，就像手机那样可以开热点，也可以连接其他热点。

WiFi 库支持配置及监控 ESP32S3 WiFi 连网功能。支持配置：

- ①：Station 模式（即 STA 模式或 WiFi 客户端模式），此时 ESP32 S3 连接到接入点（AP）。
- ②：AP 模式（即 Soft-AP 模式或接入点模式），此时基站连接到 ESP32S3 设备。
- ③：Station/AP 共存模式（ESP32S3 既是接入点，同时又作为基站连接到另外一个接入点）。
- ④：上述模式的各种安全模式（WPA、WPA2 及 WEP 等）。
- ⑤：扫描接入点（包括主动扫描及被动扫描）。
- ⑥：使用混杂模式监控 IEEE802.11 Wi-Fi 数据包。

下面作者讲解 ESP32S3 开启 WiFi 两种模式启动流程，如下：

#### 一、WiFi-AP 启动流程

WiFi-AP 启动流程如下。

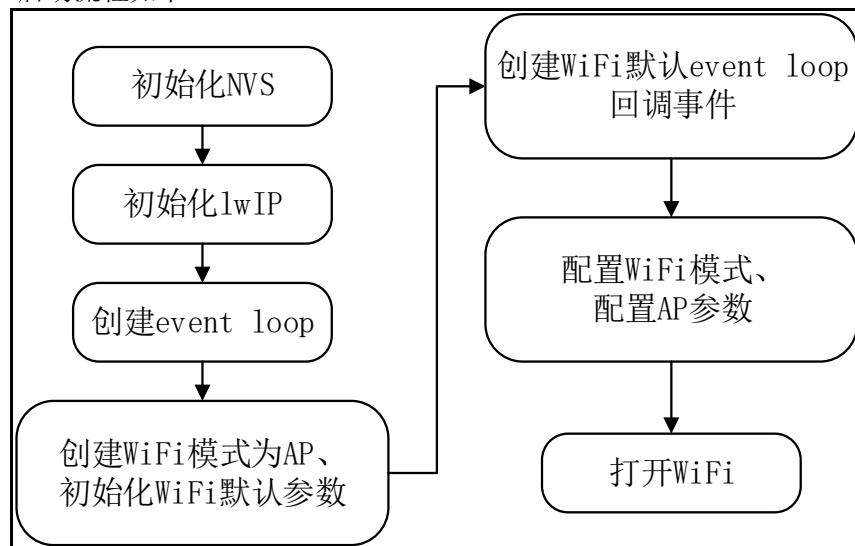


图 29.1.1 AP 启动流程

上图展示了 ESP32 系列芯片以 AP 模式开启 WiFi 的启动流程。首先，系统需要对 lwIP 协议栈进行初始化。接着，创建一个任务，该任务将用于触发相应的事件。然后，配置 WiFi 参数和 AP 模式参数。最后，启动 WiFi，从而完成以 AP 模式开启 WiFi 的操作。

#### 二、WiFi-STA 启动流程

WiFi-STA 启动流程如下。

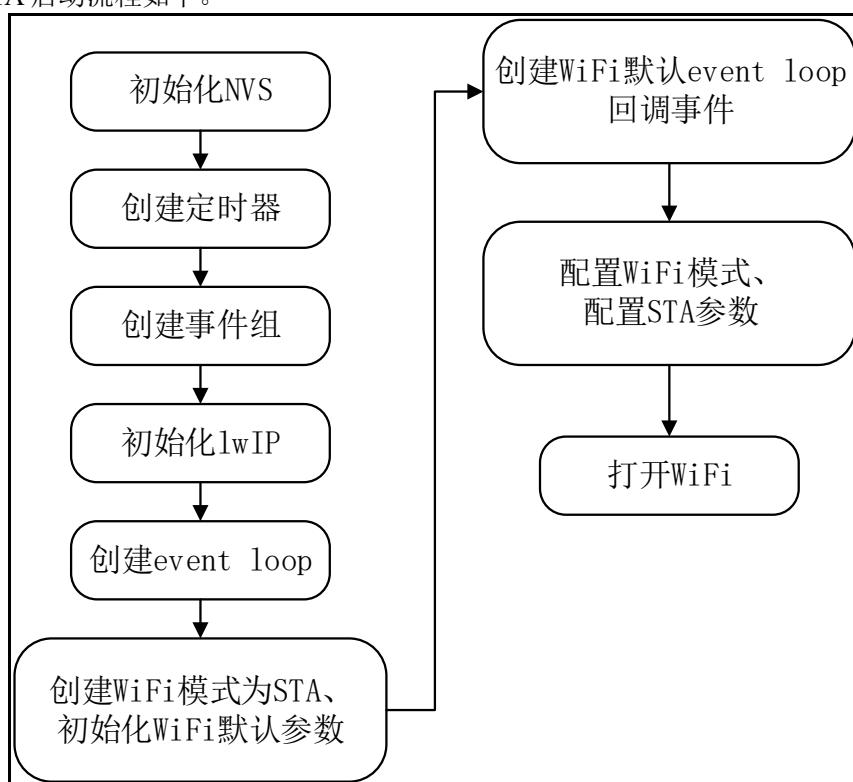


图 29.1.2 STA 模式启动流程

上图展示了 ESP32 系列芯片以 STA 模式开启 WiFi 的启动流程。首先，系统需要创建定时器和事件组，并对 lwIP 协议栈进行初始化。接着，创建一个任务，该任务将用于触发相应的事件。然后，配置 WiFi 参数和 STA 模式参数。最后，启动 WiFi，从而完成以 STA 模式开启 WiFi 的操作。

## 29.2 硬件设计

### 1. 例程功能

本章实验功能简介：扫描附近的 WIFI 信号，并在 LCD 显示屏右侧显示 3 个 WIFI 名称。

### 2. 硬件资源

1. LED 灯  
LED-IO1
2. 0.96 寸 LCD
3. 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。

## 29.3 软件设计

### 29.3.1 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。

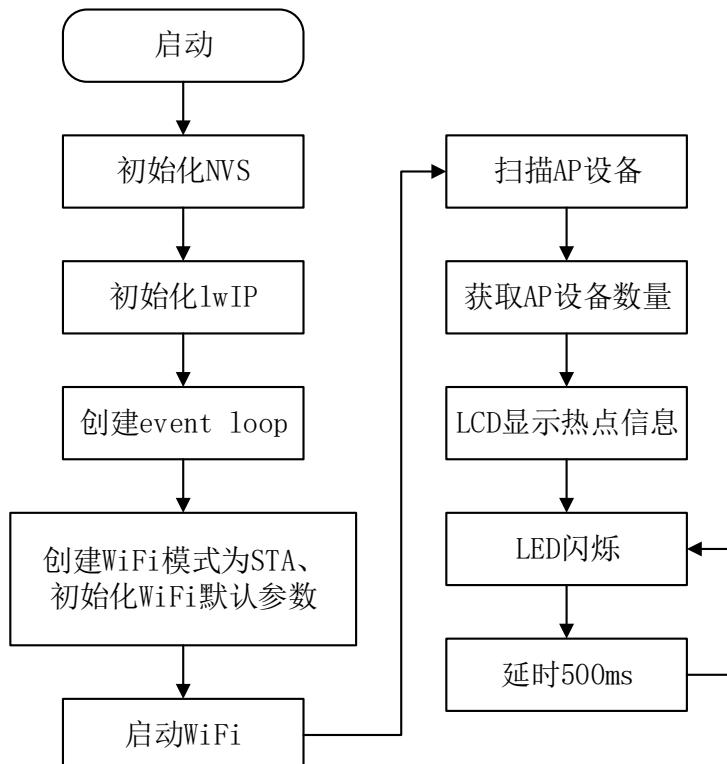


图 29.3.1 程序流程图

### 29.3.2 程序解析

在本章节实验中，我们只关心 main.c 文件内容即可，该文件内容如下：

```

/* 存储 12 个 WIFI 名称 */
#define DEFAULT_SCAN_LIST_SIZE 3
i2c_obj_t i2c0_master;
static const char *TAG = "scan";

/**
 * @brief      身份认证模式
 * @param      authmode :身份验证模式
 * @retval     无
 */
static void print_auth_mode(int authmode)
{
    switch (authmode)
    {
        /* 省略身份认证模式代码 */
    }
}

/**
 * @brief      打印 WIFI 密码类型
 * @param      pairwise_cipher :密码类型
 * @param      group_cipher    :群密码类型
 * @retval     无
 */
static void print_cipher_type(int pairwise_cipher, int group_cipher)
{
    switch (pairwise_cipher)
    {
        /* 省略 WIFI 密码类型代码 */
    }
}

```

```
switch (group_cipher)
{
    /* 省略 WIFI 密码类型代码 */
}
}

/***
 * @brief      将 Wi-Fi 初始化为 sta 并设置扫描方法
 * @param      无
 * @retval     无
 */
void wifi_scan(void)
{
    char lcd_buff[100] = {0};
    /* 网卡初始化 */
    ESP_ERROR_CHECK(esp_netif_init());
    /* 创建新的事件循环 */
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    /* 用户初始化 STA 模式 */
    esp_netif_t *sta_netif = esp_netif_create_default_wifi_sta();
    assert(sta_netif);
    /* wifi 配置初始化 */
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    uint16_t number = DEFAULT_SCAN_LIST_SIZE;
    wifi_ap_record_t ap_info[DEFAULT_SCAN_LIST_SIZE];
    uint16_t ap_count = 0;
    memset(ap_info, 0, sizeof(ap_info));
    /* 设置 WIFI 为 STA 模式 */
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    /* 启动 WIFI */
    ESP_ERROR_CHECK(esp_wifi_start());
    /* 开始扫描附件的 WIFI */
    esp_wifi_scan_start(NULL, true);
    /* 获取上次扫描中找到的 AP 列表 */
    ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&number, ap_info));
    /* 获取上次扫描中找到的 AP 数量 */
    ESP_ERROR_CHECK(esp_wifi_scan_get_ap_num(&ap_count));
    ESP_LOGI(TAG, "Total APs scanned = %u", ap_count);
    /* 下面是打印附件的 WIFI 信息 */
    for (int i = 0; (i < DEFAULT_SCAN_LIST_SIZE) && (i < ap_count); i++)
    {
        sprintf(lcd_buff, "%s", ap_info[i].ssid);
        lcd_show_string(0, 20 * i, lcd_self.width, 16, 16, lcd_buff, BLUE);
        ESP_LOGI(TAG, "SSID \t\t%s", ap_info[i].ssid);
        ESP_LOGI(TAG, "RSSI \t\t%d", ap_info[i].rssi);
        print_auth_mode(ap_info[i].authmode);

        if (ap_info[i].authmode != WIFI_AUTH_WEP)
        {
            print_cipher_type(ap_info[i].pairwise_cipher
                            , ap_info[i].group_cipher);
        }
        ESP_LOGI(TAG, "Channel \t\t%d\n", ap_info[i].primary);
    }
}
/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
```

```
esp_err_t ret;
ret = nvs_flash_init(); /* 初始化 NVS */
if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
    ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
{
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
}
led_init(); /* 初始化 LED */
spi2_init(); /* 初始化 SPI2 */
lcd_init(); /* 初始化 LCD */
lcd_show_string(0, 0, lcd_self.width, 16, 16, "WiFi SCAN Test", RED);
wifi_scan();
while (1)
{
    LED_TOGGLE();
    vTaskDelay(500);
}
}
```

在上述源码中，作者从 `wifi_scan` 函数开始讲解，该函数首先创建了 `event loop` 事件回调，即任务事件处理机制。接着，配置 WiFi 为 STA 模式（设备连接热点模式）并设置相应的参数，然后启动 WiFi。最后，程序会扫描附近的 12 个热点，并在 LCD 上显示热点的名称。此外，读者还可以通过串口查看热点的安全模式等身份认证信息。

## 29.4 下载验证

程序下载成功后，我们可以看到 LCD 显示附近 3 个热点名称，如下图所示：



图 29.4.1 SPILCD 显示效果图

## 第三十章 WiFi 路由实验

本章节实验作者把 ESP32-S3 配置为 STA 模式，即连接附近的热点。STA 模式相关知识请读者查看上一章节的内容。

本章分为如下几个小节：

30.1 硬件设计

30.2 软件设计

30.3 下载验证

### 30.1 硬件设计

#### 1. 例程功能

本章实验功能简介：扫描附近的 WIFI 信号，并连接到一个真实存在的 WIFI 热点。

#### 2. 硬件资源

1. LED 灯

LED-IO1

2. 0.96 寸 LCD

3. 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。

### 30.2 软件设计

#### 30.2.1 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：

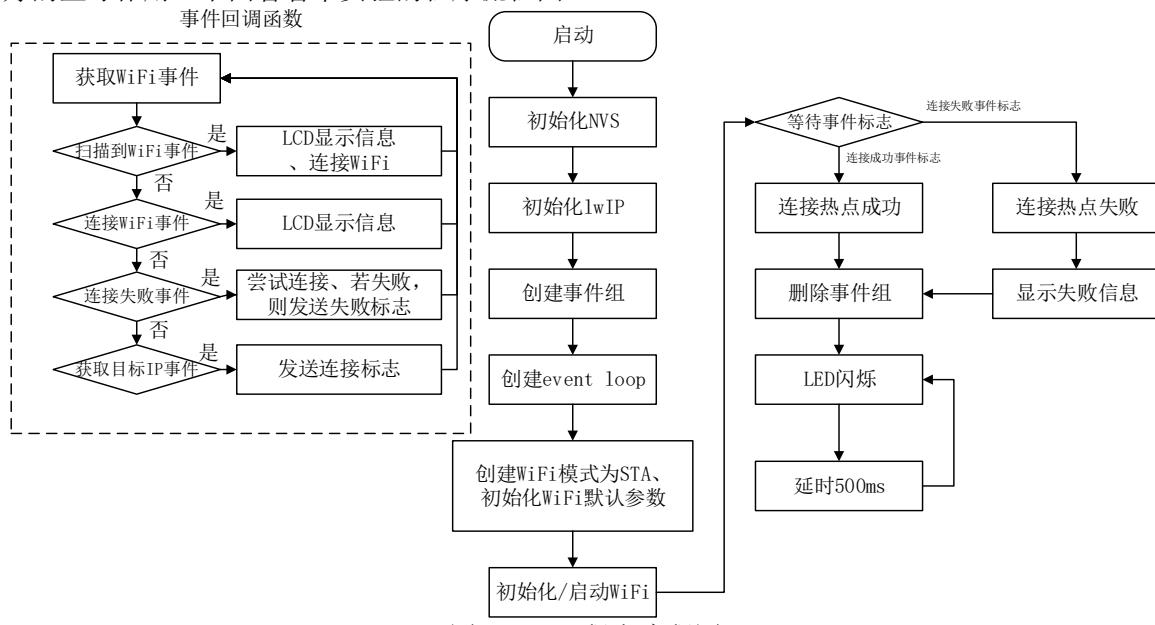


图 30.2.1.1 程序流程图

### 30.2.2 程序解析

在本章节实验中，我们只关心 main.c 文件内容即可，该文件内容如下：

```
i2c_obj_t i2c0_master;
/* 链接 wifi 名称 */
#define DEFAULT_SSID          "123"
/* wifi 密码 */
#define DEFAULT_PWD            "aa1234567"
/* 事件标志 */
static EventGroupHandle_t  wifi_event;
#define WIFI_CONNECTED_BIT    BIT0
#define WIFI_FAIL_BIT         BIT1
static const char *TAG = "static_ip";
char lcd_buff[100] = {0};

/* WIFI 默认配置 */
#define WIFICONFIG()           \
{                           \
    .sta = {                \
        .ssid = DEFAULT_SSID, \
        .password = DEFAULT_PWD, \
        .threshold.authmode = WIFI_AUTH_WPA2_PSK, \
    },                      \
}

/***
 * @brief      链接显示
 * @param      flag:2->链接;1->链接失败;0->再链接中
 * @retval     无
 */
void connet_display(uint8_t flag)
{
    if(flag == 2)
    {
        lcd_fill(0,40,lcd_self.width,lcd_self.height,WHITE);
        sprintf(lcd_buff, "SSID:%s",DEFAULT_SSID);
        lcd_show_string(0, 40, lcd_self.width, 16, 16, lcd_buff, BLUE);
        sprintf(lcd_buff, "PSW:%s",DEFAULT_PWD);
        lcd_show_string(0, 60, lcd_self.width, 16, 16, lcd_buff, BLUE);
    }
    else if (flag == 1)
    {
        lcd_show_string(0,40, lcd_self.width,16,16,"wifi connecting fail", BLUE);
    }
    else
    {
        lcd_show_string(0,40, lcd_self.width,16,16,"wifi connecting....", BLUE);
    }
}

/***
 * @brief      WIFI 链接糊掉函数
 * @param      arg:传入网卡控制块
 * @param      event_base:WIFI 事件
 * @param      event_id:事件 ID
 * @param      event_data:事件数据
 * @retval     无
 */
static void wifi_event_handler(void *arg, esp_event_base_t event_base,
                               int32_t event_id, void *event_data)
{
    static int s_retry_num = 0;

    /* 扫描到要连接的 WIFI 事件 */
```

```
if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START)
{
    connet_display(0);
    esp_wifi_connect();
}
/* 连接 WIFI 事件 */
else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_CONNECTED)
{
    connet_display(2);
}
/* 连接 WIFI 失败事件 */
else if (event_base == WIFI_EVENT &&
         event_id == WIFI_EVENT_STA_DISCONNECTED)
{
    /* 尝试连接 */
    if (s_retry_num < 20)
    {
        esp_wifi_connect();
        s_retry_num++;
        ESP_LOGI(TAG, "retry to connect to the AP");
    }
    else
    {
        xEventGroupSetBits(wifi_event, WIFI_FAIL_BIT);
    }

    ESP_LOGI(TAG, "connect to the AP fail");
}
/* 工作站从连接的 AP 获得 IP */
else if(event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP)
{
    ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
    ESP_LOGI(TAG, "static ip:" IPSTR, IP2STR(&event->ip_info.ip));
    s_retry_num = 0;
    xEventGroupSetBits(wifi_event, WIFI_CONNECTED_BIT);
}
}

/**
 * @brief      WIFI 初始化
 * @param      无
 * @retval     无
 */
void wifi_sta_init(void)
{
    static esp_netif_t *sta_netif = NULL;
    wifi_event= xEventGroupCreate();      /* 创建一个事件标志组 */
    /* 网卡初始化 */
    ESP_ERROR_CHECK(esp_netif_init());
    /* 创建新的事件循环 */
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    sta_netif= esp_netif_create_default_wifi_sta();
    assert(sta_netif);
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK( esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID,
                                                &wifi_event_handler, NULL) );
    ESP_ERROR_CHECK( esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP,
                                                &wifi_event_handler, NULL) );
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    wifi_config_t wifi_config = WIFICONFIG();
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
    ESP_ERROR_CHECK(esp_wifi_start());

    /* 等待链接成功后、ip 生成 */
}
```

```

EventBits_t bits = xEventGroupWaitBits(wifi_event,
    WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
    pdFALSE,
    pdFALSE,
    portMAX_DELAY);

/* 判断连接事件 */
if (bits & WIFI_CONNECTED_BIT)
{
    ESP_LOGI(TAG, "connected to ap SSID:%s password:%s",
        DEFAULT_SSID, DEFAULT_PWD);
}
else if (bits & WIFI_FAIL_BIT)
{
    connect_display(1);
}
else
{
    ESP_LOGE(TAG, "UNEXPECTED EVENT");
}

vEventGroupDelete(wifi_event);
}

/**
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    /* 省略代码..... */
    wifi_sta_init();

    while (1)
    {
        LED_TOGGLE();
        vTaskDelay(500);
    }
}
}

```

从上述源码中，作者首先创建了事件组、WiFi 事件回调函数，并配置 WiFi 为 STA 模式。当系统搜索到可连接的热点时，它会尝试与该热点建立连接。如果连接成功，则会在 LCD 上显示连接信息，并发送一个连接事件标志。如果连接失败，系统会尝试发送 20 次连接请求，直到没有收到任何连接回复为止。此时，会发送一个连接失败事件标志。通过查看这些连接事件标志，我们可以确定热点是否成功连接。

### 30.3 下载验证

程序下载成功后，需要利用手机或其他设备创建一个 WiFi 热点。在创建热点时，需要注意提供正确的账号名和密码，以确保程序能够成功连接。同时，确保程序中要连接的热点账号与密码与所创建的热点一致。当 LCD 显示热点的账号名和密码时，此时 ESP32-S3 设备已经与热点连接成功了，否则，LCD 提示连接失败，如下图所示：



图 30.3.1 SPILCD 显示效果图

## 第三十一章 WiFi 热点实验

本章节实验作者把 ESP32-S3 配置为 AP 模式，即创建连接热点，读者可使用手机连接该热点。AP 模式相关知识请读者查看第二章节的内容。

本章分为以下几个小节：

31.1 硬件设计

31.2 软件设计

31.3 下载验证

### 31.1 硬件设计

#### 1. 例程功能

本章实验功能简介：当手机连接这个热点时，LCD 显示该连接设备的 MAC 地址，断开时，LCD 显示断开设备的 MAC 地址。

#### 2. 硬件资源

1. LED 灯  
LED-IO1
2. 0.96 寸 LCD
3. 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。

### 31.2 软件设计

#### 31.2.1 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：

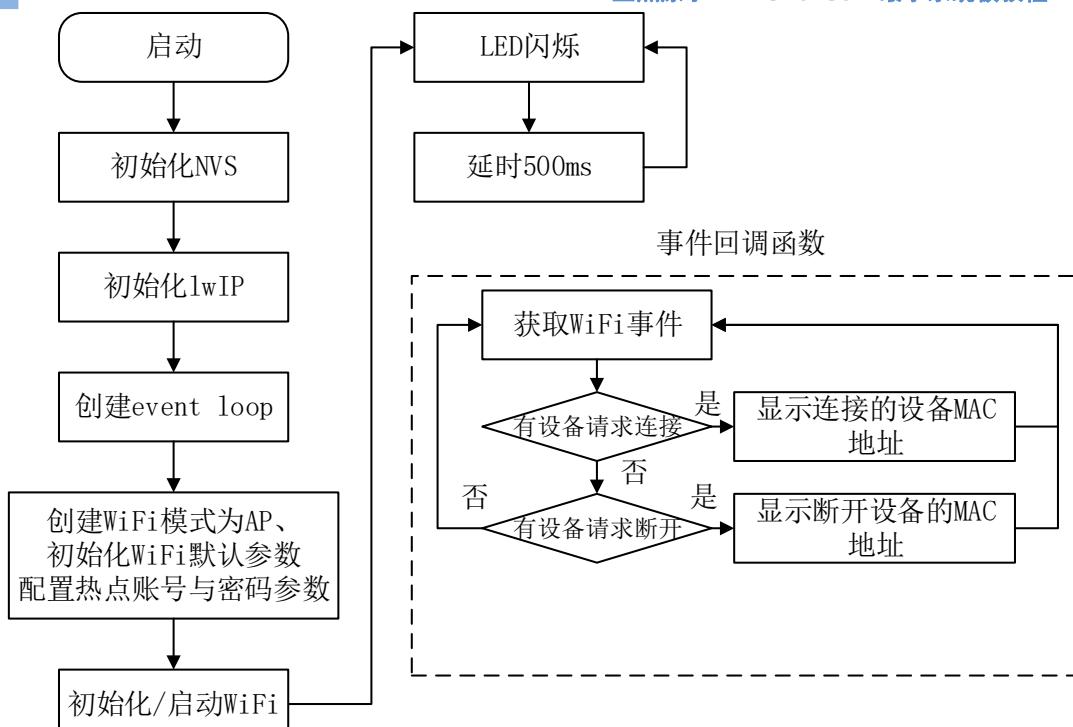


图 31.2.1.1 程序流程图

### 31.2.2 程序解析

在本章节实验中，我们只关心 main.c 文件内容即可，该文件内容如下：

```

i2c_obj_t i2c0_master;
static const char *TAG = "AP";
#define EXAMPLE_ESP_WIFI_SSID    "123"
#define EXAMPLE_ESP_WIFI_PASS    "123456789"
#define EXAMPLE_MAX_STA_CONN    5
#define MAC2STR(a) (a)[0], (a)[1], (a)[2], (a)[3], (a)[4], (a)[5]
#define MACSTR "%02x:%02x:%02x:%02x:%02x:%02x"
static char lcd_buff[100] = {0};

/**
 * @brief      WIFI 链接掉函数
 * @param      arg:传入网卡控制块
 * @param      event_base:WIFI 事件
 * @param      event_id:事件 ID
 * @param      event_data:事件数据
 * @retval     无
 */
static void wifi_event_handler(void *arg, esp_event_base_t event_base,
                               int32_t event_id, void *event_data)
{
    /* 设备连接 */
    if (event_id == WIFI_EVENT_AP_STACONNECTED)
    {
        lcd_fill(0, 40, lcd_self.width, lcd_self.height, WHITE);
        wifi_event_ap_staconnected_t *event =
            (wifi_event_ap_staconnected_t *)event_data;
        ESP_LOGI(TAG, "station " MACSTR " join, AID=%d",
                 MAC2STR(event->mac), event->aid);
        sprintf(lcd_buff, "MACSTR:" MACSTR, MAC2STR(event->mac));
        lcd_show_string(0, 40, lcd_self.width, 16, 16, lcd_buff, BLUE);
        lcd_show_string(0, 60, lcd_self.width, 16, 16, "Withdeviceconnection", BLUE);
    }
}
  
```

```
/* 设备断开 */
else if (event_id == WIFI_EVENT_AP_STADISCONNECTED)
{
    wifi_event_ap_stadisconnected_t *event =
        (wifi_event_ap_stadisconnected_t *)event_data;
    ESP_LOGI(TAG, "station " MACSTR " leave, AID=%d",
             MAC2STR(event->mac), event->aid);
    lcd_fill(0,40,lcd_self.width,lcd_self.height,WHITE);
    sprintf(lcd_buff, "Device disconnected:" MACSTR MAC2STR(event->mac));
    lcd_show_string(0, 40, lcd_self.width, 16, 16, lcd_buff, BLUE);
}
}

/**
 * @brief      WIFI 初始化
 * @param      无
 * @retval     无
 */
static void wifi_init_softap(void)
{
    /* 初始化网卡 */
    ESP_ERROR_CHECK(esp_netif_init());

    /* 创建新的事件循环 */
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    /* 使用默认配置初始化包括 netif 的 Wi-Fi */
    esp_netif_create_default_wifi_ap();
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID,
                                              &wifi_event_handler, NULL));
    /* 配置 WIFI */
    wifi_config_t wifi_config = {
        .ap = {
            .ssid = EXAMPLE_ESP_WIFI_SSID,
            .ssid_len = strlen(EXAMPLE_ESP_WIFI_SSID),
            .password = EXAMPLE_ESP_WIFI_PASS,
            .max_connection = EXAMPLE_MAX_STA_CONN,
            .authmode = WIFI_AUTH_WPA_WPA2_PSK
        },
    };

    if (strlen(EXAMPLE_ESP_WIFI_PASS) == 0)
    {
        wifi_config.ap.authmode = WIFI_AUTH_OPEN;
    }
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());
    esp_netif_ip_info_t ip_info;
    /* 获取当前设备的 IP 地址 */
    esp_netif_get_ip_info(esp_netif_get_handle_from_ifkey("WIFI_AP_DEF"),
                          &ip_info);
    char ip_addr[16];
    inet_ntoa_r(ip_info.ip.addr, ip_addr, 16);
    ESP_LOGI(TAG, "Set up softAP with IP: %s", ip_addr);
    ESP_LOGI(TAG, "wifi_init_softap finished. SSID:'%s' password:'%s'",
             EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);

    lcd_show_string(0,40,lcd_self.width,16,16,"wifi connecting.....", BLUE);
}

/**
```

```
* @brief      程序入口
* @param      无
* @retval     无
*/
void app_main(void)
{
    /* 省略部分代码..... */
    wifi_init_softap();
    while (1)
    {
        LED_TOGGLE();
        vTaskDelay(500);
    }
}
```

上述源码相对简单，主要将 ESP32-S3 设备配置为 AP 模式，即作为热点设备。然后，设置热点设备的账号、密码、安全模式等参数。在 WiFi 事件回调函数中，当有外部设备请求连接时，程序会在 LCD 上显示连接设备的 MAC 地址等信息。而当外部设备从连接状态断开时，LCD 会显示当前断开的外部设备 MAC 地址。

### 31.3 下载验证

程序下载成功后，我们利用手机连接 ESP32-S3 热点设备，当手机连接热点设备成功时，LCD 显示手机的 MAC 地址等信息，当手机从已连接状态断开时，LCD 显示断开的外部设备的 MAC 地址。下图为连接成功的 LCD 显示效果图。

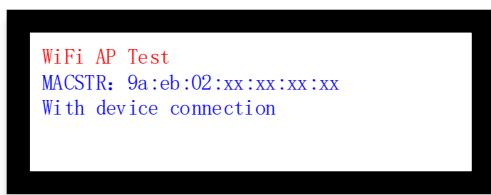


图 31.3.1 外部设备连接热点设备

下图为外部设备从已连接状态断开效果图，如下所示。



图 31.3.2 外部设备断开热点设备

## 第三十二章 UDP 实验

对于 lwIP 的 Socket 的使用方式，它与文件操作非常相似。在文件操作中，我们首先打开文件，然后进行读/写操作，最后关闭文件。在 TCP/IP 网络通信中，也存在着相同的操作流程，但所使用的接口不再是文件描述符或 FILE\*，而是被称为 Socket 的描述符。通过 Socket，我们可以进行读、写、打开和关闭操作来进行网络数据的传输。此外，还有一些辅助函数，如查询域名/IP 地址和设置 Socket 功能等。在本章中，我们将使用 Socket 编程接口来实现 UDP 实验。

本章分为以下几个部分：

32.1 Socket 编程 UDP 连接流程

32.2 硬件设计

32.3 软件设计

32.4 下载验证

### 32.1 Socket 编程 UDP 连接流程

在实现 UDP 协议之前，用户需要按照以下步骤配置结构体 sockaddr\_in 的成员变量，以便建立 UDP 连接：

- ①：配置 ESP32-S3 设备连接网络（必须的，因为 WiFi 是无线通信，所以需搭建通信桥梁）。
- ②：将 sin\_family 设置为 AF\_INET，表示使用 IPv4 网络协议。
- ③：设置 sin\_port 为所需的端口号，例如 8080。
- ④：设置 sin\_addr.s\_addr 为本地 IP 地址。
- ⑤：调用函数 Socket 创建 Socket 连接。请注意，该函数的第二个参数指定连接类型。SOCK\_STREAM 表示 TCP 连接，而 SOCK\_DGRAM 表示 UDP 连接。
- ⑥：调用函数 bind 将本地服务器地址与 Socket 进行绑定。
- ⑦：调用适当的收发函数来接收或发送数据。

通过遵循这些步骤，用户可以成功地配置并建立 UDP 连接，以实现数据的发送和接收。

### 32.2 硬件设计

#### 1. 例程功能

本章实验功能简介：

本实验主要通过 Socket 编程接口实现了一个 UDP 服务器。这个服务器具有以下功能：

- ①：可以通过按键发送 UDP 广播数据给其他 UDP 客户端。
- ②：能够接收其他 UDP 客户端发送的广播数据。
- ③：实时将接收到的数据显示在 LCD 屏幕上。

通过这个实验，用户可深入了解 UDP 协议的工作原理，并掌握如何使用 Socket 编程接口来实现 UDP 通信。这对于开发基于 UDP 的网络应用程序非常有用，例如实时通信、多播应用等。

#### 2. 硬件资源

1. LED 灯  
LED-IO1
2. 0.96 寸 LCD
3. 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。

## 32.3 软件设计

### 32.3.1 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：

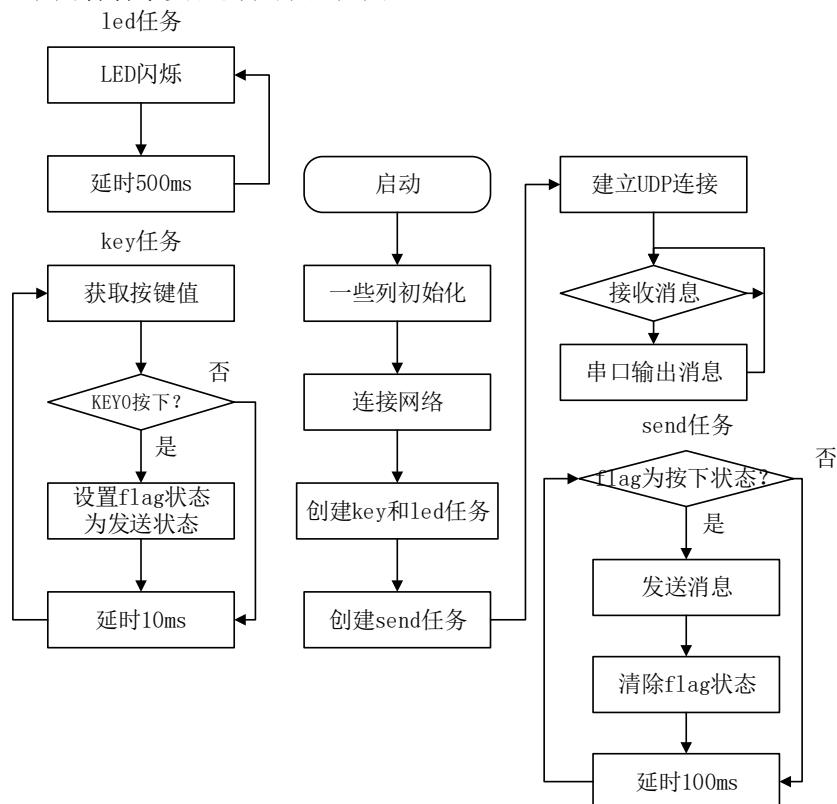


图 32.3.1.1 程序流程图

### 32.3.2 程序解析

在本章节中，我们主要关注两个文件：lwip\_demo.c 和 lwip\_demo.h。lwip\_demo.h 文件主要定义了发送标志位并声明了 lwip\_demo 函数，这部分相对简单，所以我们暂不详细解释。主要关注点是 lwip\_demo.c 文件中的函数。在 lwip\_demo 函数中，我们配置了相关的 UDP 参数，并创建了一个名为 lwip\_send\_thread 的发送数据线程。这个线程通过调用 scokec 函数来发送数据到服务器。接下来，我们将分别详细解释 lwip\_demo 函数和 lwip\_send\_thread 任务。

```

/* 需要自己设置远程 IP 地址 */
#define IP_ADDR "192.168.101.33"

#define LWIP_DEMO_RX_BUFSIZE          200           /* 最大接收数据长度 */
#define LWIP_DEMO_PORT                8080          /* 连接的本地端口号 */
#define LWIP_SEND_THREAD_PRIO   ( tskIDLE_PRIORITY + 3 ) /* 发送数据线程优先级 */

/* 接收数据缓冲区 */
uint8_t g_lwip_demo_recvbuf[LWIP_DEMO_RX_BUFSIZE];
/* 发送数据内容 */
char g_lwip_demo_sendbuf[] = "ALIENTEK DATA \r\n";
/* 数据发送标志位 */
uint8_t g_lwip_send_flag;
static struct sockaddr_in dest_addr;           /* 远端地址 */
struct sockaddr_in g_local_info;
socklen_t g_sock_fd;                          /* 定义一个 Socket 接口 */
static void lwip_send_thread(void *arg);
  
```

```
extern QueueHandle_t g_display_queue; /* 显示消息队列句柄 */

/** @brief      发送数据线程
 * @param      无
 * @retval     无
 */
void lwip_data_send(void)
{
    xTaskCreate(lwip_send_thread, "lwip_send_thread", 4096,
                NULL, LWIP_SEND_THREAD_PRIO, NULL);
}

/** @brief      lwip_demo 实验入口
 * @param      无
 * @retval     无
 */
void lwip_demo(void)
{
    char *tbuf;
    lwip_data_send(); /* 创建发送数据线程 */
    /* 远端参数设置 */
    dest_addr.sin_addr.s_addr = inet_addr(IP_ADDR); /* 目标地址 */
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(LWIP_DEMO_PORT); /* 目标端口 */

    g_local_info.sin_family = AF_INET; /* IPv4 地址 */
    g_local_info.sin_port = htons(LWIP_DEMO_PORT); /* 设置端口号 */
    g_local_info.sin_addr.s_addr = htons(INADDR_ANY); /* 设置本地 IP 地址 */

    g_sock_fd = socket(AF_INET, SOCK_DGRAM, 0); /* 建立一个新的 socket 连接 */
    tbuf = malloc(200); /* 申请内存 */
    sprintf((char *)tbuf, "Port:%d", LWIP_DEMO_PORT); /* 客户端端口号 */
    lcd_show_string(0, 60, 200, 16, 16, tbuf, MAGENTA);

    /* 建立绑定 */
    bind(g_sock_fd, (struct sockaddr *)&g_local_info, sizeof(g_local_info));
    while (1)
    {
        memset(g_lwip_demo_recvbuf, 0, sizeof(g_lwip_demo_recvbuf));
        recv(g_sock_fd, (void *)g_lwip_demo_recvbuf,
              sizeof(g_lwip_demo_recvbuf), 0);
        printf("%s\r\n", g_lwip_demo_recvbuf);
    }
}

/** @brief      发送数据线程函数
 * @param      pvParameters : 传入参数(未用到)
 * @retval     无
 */
void lwip_send_thread(void *pvParameters)
{
    pvParameters = pvParameters;

    while (1)
    { /* 有数据要发送 */
        if ((g_lwip_send_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA)
        {

```

```

printf("send\r\n");
sendto(g_sock_fd,
       (char *)g_lwip_demo_sendbuf,
       sizeof(g_lwip_demo_sendbuf), 0,
       (struct sockaddr *)&dest_addr,
       sizeof(dest_addr));
/* socket */
/* 发送的数据 */
/* 发送的数据大小 */
/* 接收端地址信息 */
/* 接收端地址信息大小 */

g_lwip_send_flag &= ~LWIP_SEND_DATA;
}

vTaskDelay(100);
}
}

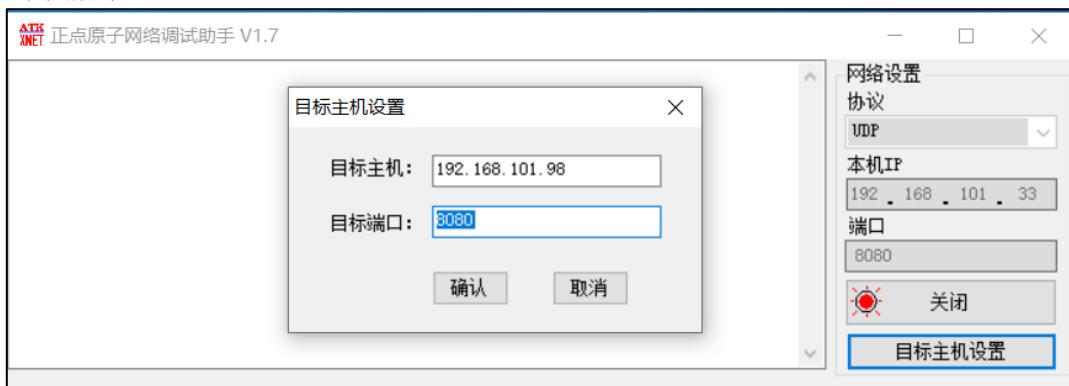
```

在源码中，lwip\_demo 函数通过 lwip\_data\_send 创建了发送数据的线程 lwip\_send\_thread，并配置了 Socket 的 UDP 协议。该线程在发送前会检查标志位，有效时则通过 sendto 发送数据并重置标志位。同时，需设置目标 IP 地址以确保数据正确发送。此外，主函数的循环中不断通过 recv 接收数据并使用串口输出接收的数据。

## 32.4 下载验证

在程序中，首先需要设置好能够连接的网络账号和密码。然后，使用笔记本电脑作为终端，确保它与 ESP32-S3 设备处于同一网络段内。

打开网络调试助手，然后配置网络参数，如 UDP 协议、端口号、目标主机设置等，设置内容如下图所示。



在确保网络连接正常后，可以通过按下 DNESP32S3M 最小系统板上的 BOOT 按键来发送数据至网络调试助手。当网络调试助手接收到“ALIENTEK DATA”字符串时，它会在显示区域展示这个信息。此外，用户还可以在调试助手的发送区域自行输入要发送的数据，然后点击发送键，将数据发送至 ESP32-S3 设备。此时，ESP32-S3 的串口将打印接收到的数据，具体操作和输出如下图所示。

```
I (11482) static_ip: connected to ap SSID:ALIENTEK-YF password:15902020353
I (65605) wifi:<ba-add>idx:0 (ifx:0, e4:0e:ee:f2:11:14), tid:5, ssn:4, winSize:64
www.openedv.com
```

图 32.4.2 接收网络调试助手的数据

## 第三十三章 TCPClient 实验

本章作者重点讲解 lwIP 的 Socket 接口如何配置 TCP 客户端，并在此基础上实现收发功能。

本章分为如下几个部分：

33.1 Socket 编程 TCPClient 连接流程

33.2 硬件设计

33.3 软件设计

33.4 下载验证

### 33.1 Socket 编程 TCPClient 连接流程

在实现 TCP 协议之前，用户需要按照以下步骤配置结构体 sockaddr\_in 的成员变量，以便建立 TCPClient 连接：

①：配置 ESP32-S3 设备连接网络（必须的，因为 WiFi 是无线通信，所以需搭建通信桥梁）。

②：将 sin\_family 设置为 AF\_INET，表示使用 IPv4 网络协议。

③：设置 sin\_port 为所需的端口号，例如 8080。

④：设置 sin\_addr.s\_addr 为远程 IP 地址。

⑤：调用函数 Socket 创建 Socket 连接。请注意，该函数的第二个参数指定连接类型。

SOCK\_STREAM 表示 TCP 连接，而 SOCK\_DGRAM 表示 UDP 连接。

⑥：调用函数 connect 连接远程 IP 地址。

⑦：调用适当的收发函数来接收或发送数据。

通过遵循这些步骤，用户可成功地配置并建立 TCPClient 连接，以实现数据的发送和接收。

### 33.2 硬件设计

#### 1. 例程功能

本章实验功能简介：

本实验主要通过 Socket 编程接口实现了一个 TCPClient 客户端。这个客户端具有以下功能：

①：可以通过按键发送 TCPClient 数据发送至服务器。

②：能够接收服务器发送的数据。

③：实时将接收到的数据显示在 LCD 屏幕上。

通过这个实验，用户可深入了解 TCP 协议的工作原理，并掌握如何使用 Socket 编程接口来实现 TCP 通信。这对于开发基于 TCP 的网络应用程序非常有用，例如实时传输、文件传输等。

#### 2. 硬件资源

1) LED 灯

LED-IO1

2) XL9555

IIC\_INT-IO0 (需在 P5 连接 IO0)

IIC\_SDA-IO41

IIC\_SCL-IO42

3) SPILCD

CS-IO21

SCK-IO12

SDA-IO11

DC-IO40 (在 P5 端口，使用跳线帽将 IO\_SET 和 LCD\_DC 相连)

PWR- IO1\_3 (XL9555)

RST- IO1\_2 (XL9555)

## 4) 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。

## 33.3 软件设计

## 33.3.1 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：

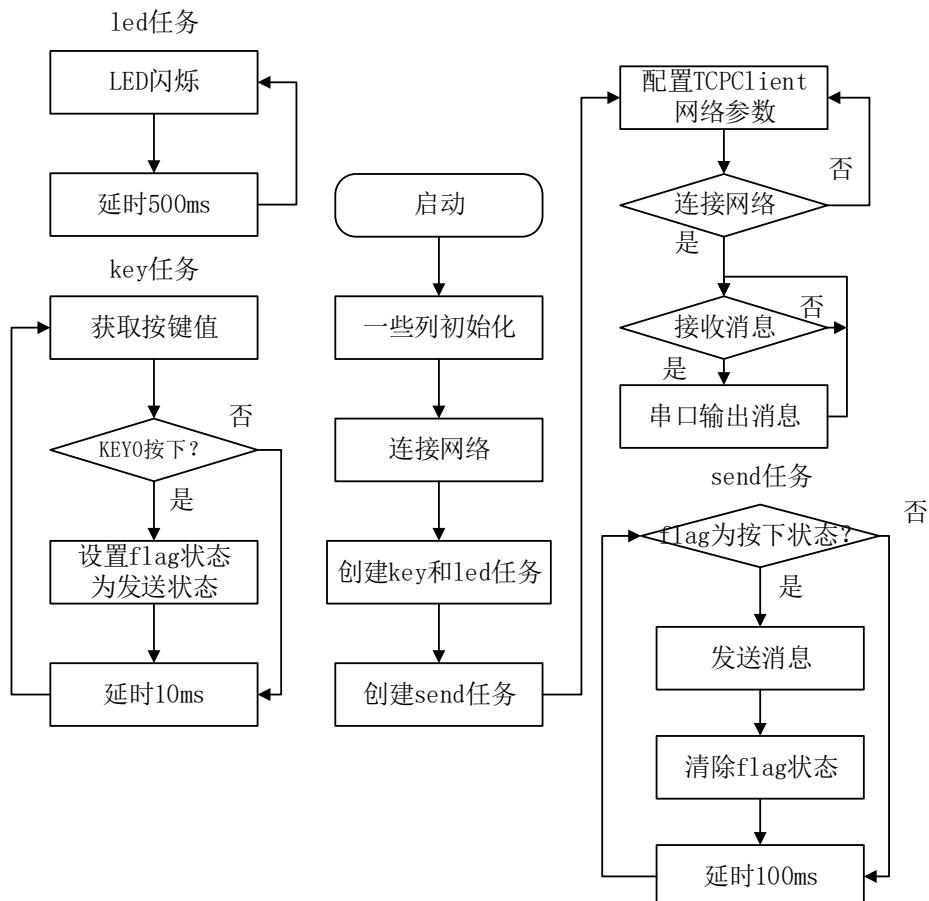


图 33.3.1.1 程序流程图

## 33.3.2 程序解析

在本章节中，我们主要关注两个文件：lwip\_demo.c 和 lwip\_demo.h。lwip\_demo.h 文件主要定义了发送标志位并声明了 lwip\_demo 函数，这部分相对简单，所以我们暂不详细解释。主要关注点是 lwip\_demo.c 文件中的函数。在 lwip\_demo 函数中，我们配置了相关的 TCPClient 参数，并创建了一个名为 lwip\_send\_thread 的发送数据线程。这个线程通过调用 scokec 函数来发送数据到服务器。接下来，我们将分别详细解释 lwip\_demo 函数和 lwip\_send\_thread 任务。

```

/* 需要自己设置远程 IP 地址 */
#define IP_ADDR      "192.168.101.33"

#define LWIP_DEMO_RX_BUFSIZE      100          /* 最大接收数据长度 */
#define LWIP_DEMO_PORT           8080          /* 连接的本地端口号 */
#define LWIP_SEND_THREAD_PRIO    ( tskIDLE_PRIORITY + 3 ) /* 发送数据线程优先级 */
/* 接收数据缓冲区 */

```

```
uint8_t g_lwip_demo_recvbuf[LWIP_DEMO_RX_BUFSIZE];\n\n/* 发送数据内容 */\nuint8_t g_lwip_demo_sendbuf[] = "ALIENTEK DATA \r\n";\n/* 数据发送标志位 */\nuint8_t g_lwip_send_flag;\nint g_sock = -1;\nint g_lwip_connect_state = 0;\nstatic void lwip_send_thread(void *arg);\n\n/**\n * @brief      发送数据线程\n * @param      无\n * @retval     无\n */\nvoid lwip_data_send(void)\n{\n    xTaskCreate(lwip_send_thread, "lwip_send_thread", 4096,\n                NULL, LWIP_SEND_THREAD_PRIO, NULL);\n}\n\n/**\n * @brief      lwip_demo 实验入口\n * @param      无\n * @retval     无\n */\nvoid lwip_demo(void)\n{\n    struct sockaddr_in atk_client_addr;\n    err_t err;\n    int recv_data_len;\n    char *tbuf;\n\n    lwip_data_send();\n\n    /* 创建发送数据线程 */\n\n    while (1)\n    {\n        sock_start:\n            g_lwip_connect_state = 0;\n            atk_client_addr.sin_family = AF_INET; /* 表示 IPv4 网络协议 */\n            atk_client_addr.sin_port = htons(LWIP_DEMO_PORT); /* 端口号 */\n            atk_client_addr.sin_addr.s_addr = inet_addr(IP_ADDR); /* 远程 IP 地址 */\n            g_sock = socket(AF_INET, SOCK_STREAM, 0); /* 可靠数据流交付服务既是 TCP 协议 */\n            memset(&(atk_client_addr.sin_zero), 0,\n                   sizeof(atk_client_addr.sin_zero));\n\n            tbuf = malloc(200);\n\n            /* 申请内存 */\n            sprintf((char *)tbuf, "Port:%d", LWIP_DEMO_PORT);\n\n            /* 客户端端口号 */\n            lcd_show_string(5, 170, 200, 16, 16, tbuf, MAGENTA);\n\n            /* 连接远程 IP 地址 */\n            err = connect(g_sock, (struct sockaddr *)&atk_client_addr,\n                          sizeof(struct sockaddr));\n\n            if (err == -1)\n            {\n                lcd_show_string(5, 190, 200, 16, 16, "State:Disconnect", MAGENTA);\n                g_sock = -1;\n                closesocket(g_sock);\n                free(tbuf);\n                vTaskDelay(10);\n                goto sock_start;\n            }\n    }\n}
```

```
lcd_show_string(5,190,200,16,16,"State:Connection Successful", MAGENTA);
g_lwip_connect_state = 1;

while (1)
{
    recv_data_len = recv(g_sock,g_lwip_demo_recvbuf,
                          LWIP_DEMO_RX_BUFSIZE,0);
    if (recv_data_len <= 0 )
    {
        closesocket(g_sock);
        g_sock = -1;
        lcd_fill(5, 190, lcd_self.width,320, WHITE);
        lcd_show_string(5, 190, 200,16,16,"State:Disconnect", MAGENTA);
        free(tbuf);
        goto sock_start;
    }

    printf("%s\r\n",g_lwip_demo_recvbuf);
    vTaskDelay(10);
}
}

/***
 * @brief      发送数据线程函数
 * @param      pvParameters : 传入参数(未用到)
 * @retval     无
 */
void lwip_send_thread(void *pvParameters)
{
    pvParameters = pvParameters;

    err_t err;

    while (1)
    {
        while (1)
        {
            if(((g_lwip_send_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA)
               && (g_lwip_connect_state == 1)) /* 有数据要发送 */
            {
                err = write(g_sock, g_lwip_demo_sendbuf,
                            sizeof(g_lwip_demo_sendbuf));

                if (err < 0)
                {
                    break;
                }

                g_lwip_send_flag &= ~LWIP_SEND_DATA;
            }

            vTaskDelay(10);
        }

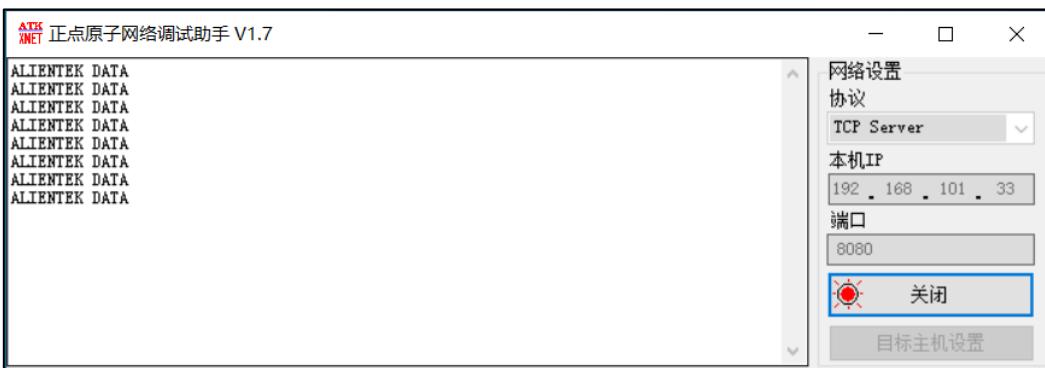
        closesocket(g_sock);
    }
}
```

在上述源码中，首先创建了一个用于发送 ESP32-S3 设备数据的任务。然后，对 TCPClient 进行网络参数配置，并调用 connect 函数来建立与远程服务器的连接。当连接成功时，系统将进入接收轮询任务。如果出现断开连接的情况，系统将尝试重新连接服务器。在发送线程中，发送数据前会检查标志位。如果标志位有效，则通过 write 函数发送数据并重置标志位。

### 33.4 下载验证

在程序中，首先需要设置好能够连接的网络账号和密码。然后，使用笔记本电脑作为终端，确保它与 ESP32-S3 设备处于同一网络段内。

打开网络调试助手，然后配置网络参数，如 TCP Server 协议、端口号等，设置内容如下图所示。



在确保网络连接正常后，可以通过按下 DNESP32S3M 最小系统板上的 BOOT 按键来发送数据至网络调试助手。当网络调试助手接收到“ALIENTEK DATA”字符串时，它会在显示区域展示这个信息。此外，用户还可以在调试助手的发送区域自行输入要发送的数据，然后点击发送键，将数据发送至 ESP32-S3 设备。此时，ESP32-S3 的串口将打印接收到的数据，具体操作和输出如下图所示。

```
I (11482) static_ip: connected to ap SSID:ALIENTEK-YF password:15902020353
I (65605) wifi:<ba-add>idx:0 (ifx:0, e4:0e:ee:f2:11:14), tid:5, ssn:4, winSize:64
www.openedv.com
```

图 33.4.2 接收网络调试助手的数据

## 第三十四章 TCPServer 实验

本章笔者重点讲解 lwIP 的 Socket 接口如何配置 TCP 服务器，并在此基础上实现收发功能。

本章分为如下几个部分：

34.1 Socket 编程 TCPServer 连接流程

34.2 硬件设计

34.3 软件设计

34.4 下载验证

### 34.1 Socket 编程 TCPServer 连接流程

在实现 TCP 协议之前，用户需要按照以下步骤配置结构体 sockaddr\_in 的成员变量，以便建立 TCPServer 连接：

- ①：配置 ESP32-S3 设备连接网络（必须的，因为 WiFi 是无线通信，所以需搭建通信桥梁）。
- ②：将 sin\_family 设置为 AF\_INET，表示使用 IPv4 网络协议。
- ③：设置 sin\_port 为所需的端口号，例如 8080。
- ④：设置 sin\_addr.s\_addr 为本地 IP 地址。
- ⑤：调用函数 Socket 创建 Socket 连接。请注意，该函数的第二个参数指定连接类型。

SOCK\_STREAM 表示 TCP 连接，而 SOCK\_DGRAM 表示 UDP 连接。

⑥：调用函数 bind 绑定本地 IP 地址和端口号。

⑦：调用函数 listen 监听连接请求

⑧：调用函数 accept 监听连接

⑨：调用适当的收发函数来接收或发送数据。

通过遵循这些步骤，用户可成功地配置并建立 TCPServer 连接，以实现数据的发送和接收。

### 34.2 硬件设计

#### 1. 例程功能

本章实验功能简介：

本实验主要通过 Socket 编程接口实现了一个 TCPServer 服务器。这个客户端具有以下功能：

- ①：可以通过按键发送 TCPServer 数据发送至客户端。
- ②：能够接收客户端发送的数据。
- ③：实时将接收到的数据显示在 LCD 屏幕上。

通过这个实验，用户可深入了解 TCP 协议的工作原理，并掌握如何使用 Socket 编程接口来实现 TCP 通信。这对于开发基于 TCP 的网络应用程序非常有用，例如实时传输、文件传输等。

#### 2. 硬件资源

1) LED 灯

LED-IO1

2) XL9555

IIC\_INT-IO0 (需在 P5 连接 IO0)

IIC\_SDA-IO41

IIC\_SCL-IO42

3) SPILCD

CS-IO21

SCK-IO12

SDA-IO11

DC-IO40 (在 P5 端口，使用跳线帽将 IO\_SET 和 LCD\_DC 相连)

- PWR- IO1\_3 (XL9555)  
 RST- IO1\_2 (XL9555)  
 4) 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。

### 34.3 软件设计

#### 34.3.1 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：

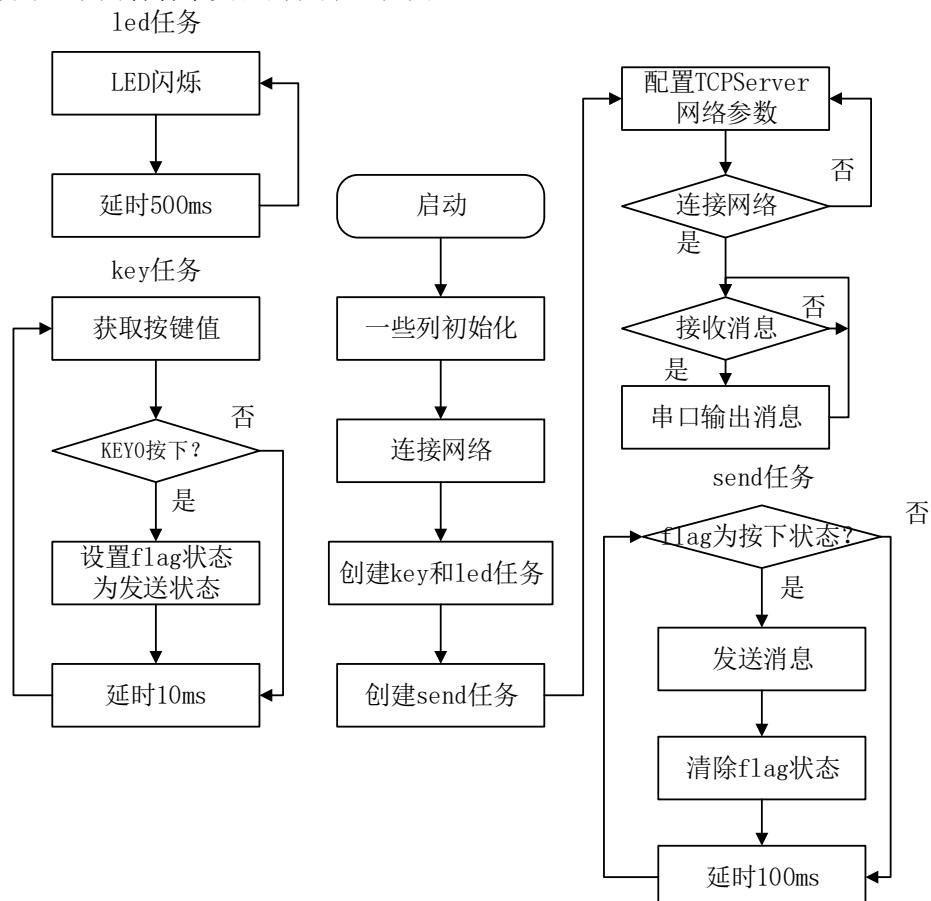


图 34.3.1.1 程序流程图

#### 34.3.2 程序解析

在本章节中，我们主要关注两个文件：lwip\_demo.c 和 lwip\_demo.h。lwip\_demo.h 文件主要定义了发送标志位并声明了 lwip\_demo 函数，这部分相对简单，所以我们暂不详细解释。主要关注点是 lwip\_demo.c 文件中的函数。在 lwip\_demo 函数中，我们配置了相关的 TCPServer 参数，并创建了一个名为 lwip\_send\_thread 的发送数据线程。这个线程通过调用 scokec 函数来发送数据到服务器。接下来，我们将分别详细解释 lwip\_demo 函数和 lwip\_send\_thread 任务。

```

/* 需要自己设置远程 IP 地址 */
#define IP_ADDR      "192.168.101.33"

#define LWIP_DEMO_RX_BUFSIZE    100
#define LWIP_DEMO_PORT          8080
/* 最大接收数据长度 */
/* 连接的本地端口号 */
  
```

```
#define LWIP_SEND_THREAD_PRIO  ( tskIDLE_PRIORITY + 3 ) /* 发送数据线程优先级 */
/* 接收数据缓冲区 */
uint8_t g_lwip_demo_recvbuf[LWIP_DEMO_RX_BUFSIZE];

/* 发送数据内容 */
uint8_t g_lwip_demo_sendbuf[] = "ALIENTEK DATA \r\n";
/* 数据发送标志位 */
uint8_t g_lwip_send_flag;
int g_sock = -1;
int g_lwip_connect_state = 0;
static void lwip_send_thread(void *arg);

/***
 * @brief      发送数据线程
 * @param      无
 * @retval     无
 */
void lwip_data_send(void)
{
    xTaskCreate(lwip_send_thread, "lwip_send_thread", 4096,
                NULL, LWIP_SEND_THREAD_PRIO, NULL);
}

/***
 * @brief      lwip_demo 实验入口
 * @param      无
 * @retval     无
 */
void lwip_demo(void)
{
    struct sockaddr_in atk_client_addr;
    err_t err;
    int recv_data_len;
    char *tbuf;

    lwip_data_send(); /* 创建发送数据线程 */

    while (1)
    {
        sock_start:
        g_lwip_connect_state = 0;
        atk_client_addr.sin_family = AF_INET; /* 表示 IPv4 网络协议 */
        atk_client_addr.sin_port = htons(LWIP_DEMO_PORT); /* 端口号 */
        atk_client_addr.sin_addr.s_addr = inet_addr(IP_ADDR); /* 远程 IP 地址 */
        g_sock = socket(AF_INET, SOCK_STREAM, 0); /* 可靠数据流交付服务既是 TCP 协议 */
        memset(&(atk_client_addr.sin_zero), 0,
               sizeof(atk_client_addr.sin_zero));

        tbuf = malloc(200); /* 申请内存 */
        sprintf((char *)tbuf, "Port:%d", LWIP_DEMO_PORT); /* 客户端端口号 */
        lcd_show_string(5, 170, 200, 16, 16, tbuf, MAGENTA);

        /* 连接远程 IP 地址 */
        err = connect(g_sock, (struct sockaddr *)&atk_client_addr,
                      sizeof(struct sockaddr));

        if (err == -1)
        {
            lcd_show_string(5, 190, 200, 16, 16, "State:Disconnect", MAGENTA);
            g_sock = -1;
            closesocket(g_sock);
            free(tbuf);
            vTaskDelay(10);
        }
    }
}
```

```
        goto sock_start;
    }

    lcd_show_string(5,190,200,16,16,"State:Connection Successful", MAGENTA);
    g_lwip_connect_state = 1;

    while (1)
    {
        recv_data_len = recv(g_sock,g_lwip_demo_recvbuf,
                             LWIP_DEMO_RX_BUFSIZE,0);
        if (recv_data_len <= 0 )
        {
            closesocket(g_sock);
            g_sock = -1;
            lcd_fill(5, 190, lcd_self.width,320, WHITE);
            lcd_show_string(5,190,200,16,16,"State:Disconnect", MAGENTA);
            free(tbuf);
            goto sock_start;
        }

        printf("%s\r\n",g_lwip_demo_recvbuf);
        vTaskDelay(10);
    }
}

/***
 * @brief      发送数据线程函数
 * @param      pvParameters : 传入参数(未用到)
 * @retval     无
 */
void lwip_send_thread(void *pvParameters)
{
    pvParameters = pvParameters;

    err_t err;

    while (1)
    {
        while (1)
        {
            if(((g_lwip_send_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA)
               && (g_lwip_connect_state == 1)) /* 有数据要发送 */
            {
                err = write(g_sock, g_lwip_demo_sendbuf,
                            sizeof(g_lwip_demo_sendbuf));

                if (err < 0)
                {
                    break;
                }

                g_lwip_send_flag &= ~LWIP_SEND_DATA;
            }

            vTaskDelay(10);
        }

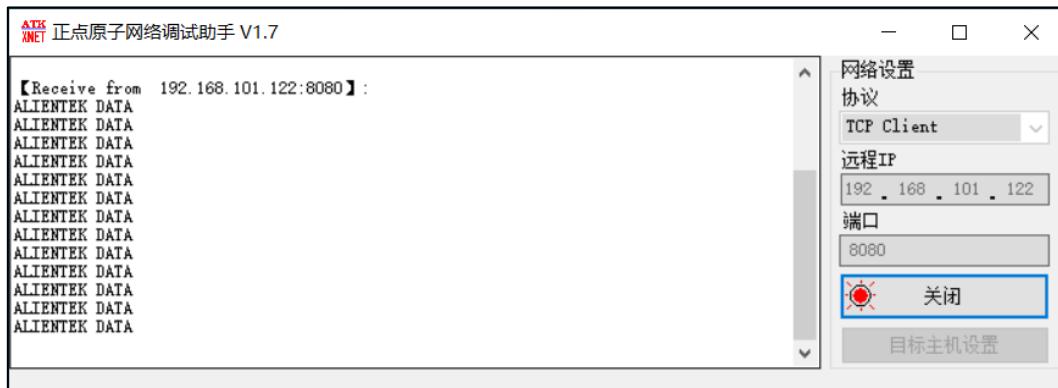
        closesocket(g_sock);
    }
}
```

上述源码中，我们首先创建一个发送任务，用来发送 ESP32-S3 设备的数据，然后配置 TCPServer 网络参数，并调用函数 `connect` 连接远程服务器，当连接成功时，系统进入接收轮询任务，反次，重新连接客户端。发送线程在发送前会检查标志位，有效时则通过 `write` 发送数据并重置标志位。

### 34.4 下载验证

在程序中，首先需要设置好能够连接的网络账号和密码。然后，使用笔记本电脑作为终端，确保它与 ESP32-S3 设备处于同一网络段内。

打开网络调试助手，然后配置网络参数，如 TCPClient 协议、端口号等，设置内容如下图所示。



在确保网络连接正常后，可以通过按下 DNESP32S3M 最小系统板上的 BOOT 按键来发送数据至网络调试助手。当网络调试助手接收到“ALIENTEK DATA”字符串时，它会在显示区域展示这个信息。此外，用户还可以在调试助手的发送区域自行输入要发送的数据，然后点击发送键，将数据发送至 ESP32-S3 设备。此时，ESP32-S3 的串口将打印接收到的数据，具体操作和输出如下图所示。

```
I (11482) static_ip: connected to ap SSID:ALIENTEK-YF password:15902020353
I (65605) wifi:<ba-add>idx:0 (ifx:0, e4:0e:ee:f2:11:14), tid:5, ssn:4, winSize:64
www.openedv.com
```

图 34.4.2 接收网络调试助手的数据

## 第三十五章 WiFi 一键配网

ESP32-S3 的一键配网模式是一种方便快捷的 WiFi 配置方式。在这种模式下，用户无需手动输入 WiFi 的 SSID 和密码等信息，只需要通过一键操作，即可完成 WiFi 的配置和连接。本章节，作者使用乐鑫官方提供的 SmartConfig 软件一键配置 WiFi 账号与密码。

本章分为如下几个小节：

35.1 主流 WIFI 配网方式简介

35.2 硬件设计

35.3 软件设计

35.4 下载验证

### 35.1 主流 WIFI 配网方式简介

目前主流的 WIFI 配网方式主要有以下三种：

#### 一、SoftAP 配网

ESP32-S3 会建立一个 WiFi 热点（AP 模式），用户将手机连接到这个热点后，将要连接的 WiFi 信息发送给 ESP32-S3，ESP32-S3 得到 SSID 和密码。

①：优点：很可靠，成功率基本达到 100%，设备端的代码简单。

②：缺点：需要手动切换手机 WiFi 连接的网络，先连接到 ESP32 的 AP 网络，配置完成后恢复连接正常 WiFi 网络，操作上存在复杂性，可能给用户带来困扰。

③：官方支持：没有提供 Demo。

#### 二、Smartconfig 配网

ESP32-S3 处于混杂模式下，监听网络中的所有报文，手机 APP 将当前连接的 SSID 和密码编码到 UDP 报文中，通过广播或组播的方式发送报文，ESP32-S3 接收到 UDP 报文后解码，得到 SSID 和密码，然后使用该组 SSID 和密码去连接网络。

①：优缺点：简洁，用户容易操作，但配网成功率受环境影响较大。

②：官方支持：提供 Demo 和 smart\_config 例程。

#### 三、Airkiss 配网

AirKiss 是微信硬件平台提供的一种 WIFI 设备快速入网配置技术。要使用微信客户端的方式配置设备入网，需要设备支持 AirKiss 技术。Airkiss 的原理和 Smartconfig 很类似，设备工作在混杂模式下，微信客户端发送包含 SSID 和密码的广播包，设备收到广播包解码得到 SSID 和密码。详细的可以参考微信官方的介绍。

①：优缺点：简洁，用户容易操作，但配网成功率受环境影响较大。

②：官方支持：提供 Demo 和 smart\_config 例程。

本实验以 Smartconfig 软件对 ESP32-S3 设备进行一键配网，该软件的安装包可在乐鑫官方网站的[相关下载网页](#)找到，如下图所示。

+ ESP-TOUCH for Android	<b>安卓手机安装包</b>	Android	V2.0.0
+ ESP-Drone for iOS		IOS	V1.0.1
+ ESP-BluFi for iOS		IOS	V1.2.0
+ ESP-Drone for Android		Android	V0.7.3
+ ESP-MESH for Android		Android	V1.2.3
<b>苹果手机安装包</b>			
+ ESP-TOUCH for iOS		IOS	V2.0.0

图 35.1.1 Smartconfig 软件下载

下载成功后，需把安装包转移到安卓手机或者苹果手机上安装。

## 35.2 硬件设计

### 1. 例程功能

本章实验功能简介：设备进入初始化状态，开启混监听所有网络数据包，此时 LCD 显示 "In the distribution network....."，表示设备已进入混监听模式。手机连上自己的 WiFi，开启 APP（EspTouch）软件，输入手机所在 WiFi 密码，请求配网，发送 UDP 广播包。ESP32 -S3 通过 UDP 包（长度）获取配置信息捕捉到路由 SSID 和 PASSWD，连接路由器，此时 LCD 显示路由的账号与密码，表示连接路由成功。

### 2. 硬件资源

1. LED 灯  
LED-IO1
2. 0.96 寸 LCD
3. 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。

## 35.3 软件设计

### 35.3.1 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：

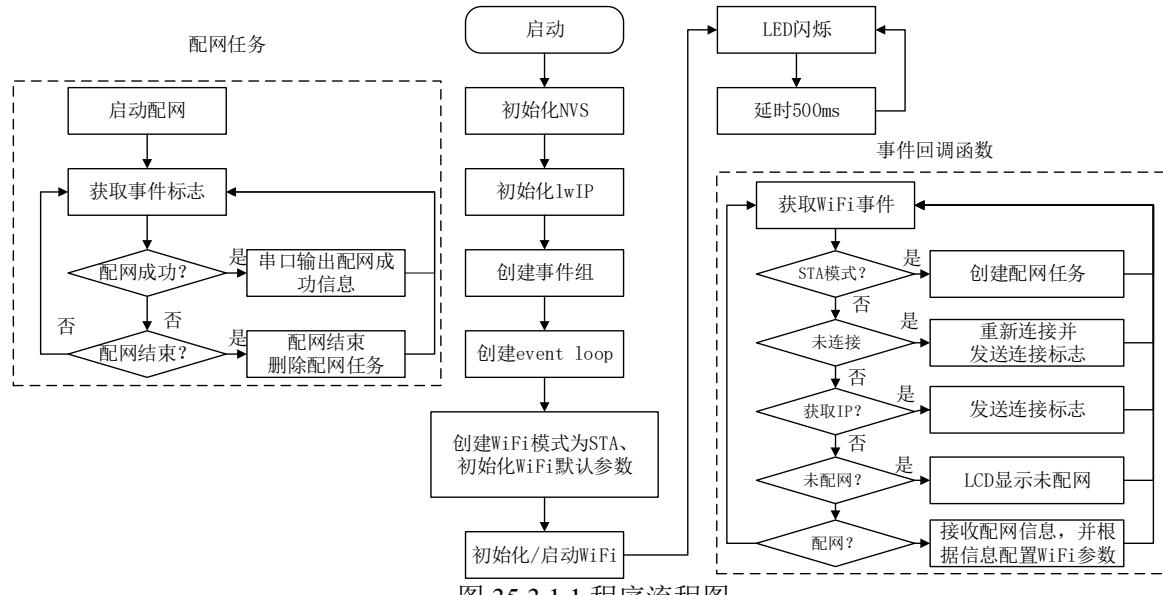


图 35.3.1.1 程序流程图

### 35.3.2 程序解析

在本章节实验中，我们只关心 main.c 文件内容即可，该文件内容如下：

```

/* 定义事件 */
static EventGroupHandle_t s_wifi_event_group;
static const int CONNECTED_BIT = BIT0;
static const int ESPTOUCH_DONE_BIT = BIT1;

```

```
static const char *TAG = "smartconfig_example";
static void smartconfig_task(void * parm);
static char lcd_buff[100] = {0};

/***
 * @brief      WIFI 链接掉函数
 * @param      arg:传入网卡控制块
 * @param      event_base:WIFI 事件
 * @param      event_id:事件 ID
 * @param      event_data:事件数据
 * @retval     无
 */
static void event_handler(void* arg, esp_event_base_t event_base,
                         int32_t event_id, void* event_data)
{
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START)
    {
        xTaskCreate(smartconfig_task, "smartconfig_task", 4096, NULL, 3, NULL);
    }
    else if (event_base == WIFI_EVENT &&
              event_id == WIFI_EVENT_STA_DISCONNECTED)
    {
        esp_wifi_connect();
        xEventGroupClearBits(s_wifi_event_group, CONNECTED_BIT);
    }
    else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP)
    {
        xEventGroupSetBits(s_wifi_event_group, CONNECTED_BIT);
    }
    else if (event_base == SC_EVENT && event_id == SC_EVENT_SCAN_DONE)
    {
        ESP_LOGI(TAG, "Scan done");
        lcd_show_string(0, 90, 320, 16, 16, "In the distribution network", BLUE);
    }
    else if (event_base == SC_EVENT && event_id == SC_EVENT_FOUND_CHANNEL)
    {
        ESP_LOGI(TAG, "Found channel");
    }
    /* 已获取 SSID 和密码, 表示配网成功 */
    else if (event_base == SC_EVENT && event_id == SC_EVENT_GOT_SSID_PSWD)
    {
        ESP_LOGI(TAG, "Got SSID and password");

        smartconfig_event_got_ssid_pswd_t *evt =
            (smartconfig_event_got_ssid_pswd_t *)event_data;
        wifi_config_t wifi_config;
        uint8_t ssid[33] = { 0 };
        uint8_t password[65] = { 0 };
        uint8_t rvd_data[33] = { 0 };

        bzero(&wifi_config, sizeof(wifi_config_t));
        memcpy(wifi_config.sta.ssid, evt->ssid, sizeof(wifi_config.sta.ssid));
        memcpy(wifi_config.sta.password, evt->password,
               sizeof(wifi_config.sta.password));
        wifi_config.sta.bssid_set = evt->bssid_set;

        if (wifi_config.sta.bssid_set == true)
        {
            memcpy(wifi_config.sta.bssid, evt->bssid,
                   sizeof(wifi_config.sta.bssid));
        }

        memcpy(ssid, evt->ssid, sizeof(evt->ssid));
        memcpy(password, evt->password, sizeof(evt->password));
    }
}
```

```
ESP_LOGI(TAG, "SSID:%s", ssid);
ESP_LOGI(TAG, "PASSWORD:%s", password);

lcd_fill(0,90,320,240,WHITE);
sprintf(lcd_buff, "%s",ssid);
lcd_show_string(0, 90, 320, 16, 16, lcd_buff, BLUE);
sprintf(lcd_buff, "%s",password);
lcd_show_string(0, 110, 320, 16, 16, lcd_buff, BLUE);
lcd_show_string(0,130,320,16,16,"distribution network", BLUE);

/* 手机 APPEspTouch 软件使用 ESPTOUCH V2 模式，会执行以下代码 */
if (evt->type == SC_TYPE_ESPTOUCH_V2)
{
    ESP_ERROR_CHECK( esp_smartconfig_get_rvd_data(rvd_data,
        sizeof(rvd_data)) );
    ESP_LOGI(TAG, "RVD_DATA:");
    for (int i = 0; i < 33; i++)
    {
        printf("%02x ", rvd_data[i]);
    }
    printf("\n");
}

ESP_ERROR_CHECK( esp_wifi_disconnect() );
ESP_ERROR_CHECK( esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
esp_wifi_connect();
}
else if (event_base == SC_EVENT && event_id == SC_EVENT_SEND_ACK_DONE)
{
    xEventGroupSetBits(s_wifi_event_group, ESPTOUCH_DONE_BIT);
}
}

/***
 * @brief      WiFi 一键配网
 * @param      无
 * @retval     无
 */
static void wifi_smartconfig_sta(void)
{
    /* 初始化网卡 */
    ESP_ERROR_CHECK(esp_netif_init());
    /* 创建事件 */
    s_wifi_event_group = xEventGroupCreate();
    /* 使用默认配置初始化包括 netif 的 Wi-Fi */
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    /* 把 WIFI 网卡设置为 STA 模式 */
    esp_netif_t *sta_netif = esp_netif_create_default_wifi_sta();
    assert(sta_netif);
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    /* WIFI 初始化 */
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) );

    /* 注册 WIFI 事件 */
    ESP_ERROR_CHECK( esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID,
        &event_handler, NULL) );
    ESP_ERROR_CHECK( esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP,
        &event_handler, NULL) );
    ESP_ERROR_CHECK( esp_event_handler_register(SC_EVENT, ESP_EVENT_ANY_ID,
        &event_handler, NULL) );

    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK( esp_wifi_start() );
}
```

```
}

/***
 * @brief      一键配网回调函数
 * @param      parm:传入的形参(未使用)
 * @retval     无
 */
static void smartconfig_task(void * parm)
{
    parm = parm;
    EventBits_t uxBits;
    /* 设置配网协议 */
    ESP_ERROR_CHECK( esp_smartconfig_set_type(SC_TYPE_ESPTOUCH) );
    /* 设置配网参数 */
    smartconfig_start_config_t cfg = SMARTCONFIG_START_CONFIG_DEFAULT();
    /* 开始配网 */
    ESP_ERROR_CHECK( esp_smartconfig_start(&cfg) );

    while (1)
    {
        /* 获取事件 */
        uxBits = xEventGroupWaitBits(s_wifi_event_group, CONNECTED_BIT
                                     | ESPTOUCH_DONE_BIT, true, false, portMAX_DELAY);

        /* 配网成功 */
        if(uxBits & CONNECTED_BIT)
        {
            ESP_LOGI(TAG, "WiFi Connected to ap");
        }

        /* 智能配置结束 */
        if(uxBits & ESPTOUCH_DONE_BIT)
        {
            /* 配网结束, 删除任务 */
            esp_smartconfig_stop();
            vTaskDelete(NULL);
        }
    }
}

/***
 * @brief      程序入口
 * @param      无
 * @retval     无
 */
void app_main(void)
{
    /* 省略部分代码..... */

    wifi_smartconfig_sta();

    while (1)
    {
        LED_TOGGLE();
        vTaskDelay(500);
    }
}
```

上述源码是把 ESP32-S3 设备配置为 STA 模式，然后开启配网任务并启动配网，此时，ESP32-S3 处于混杂模式下，监听网络中的所有报文，当手机 APP 将当前连接的 SSID 和密码编码到 UDP 报文中，通过广播或组播的方式发送报文，ESP32-S3 接收到 UDP 报文后解码，得到 SSID 和密码，然后使用该组 SSID 和密码去连接当前网络。

### 35.4 下载验证

程序下载成功后，我们打开“EspTouch”软件，在此软件下点击“EspTouch”选项，注意：手机必须连接 WiFi，才能一键配网，如下图所示。



图 35.4.1 手机配置要连接的 WiFi 账号与密码

此时，我们填写好“ALIENTEK-YF\_5G” WiFi 密码和传输方式，可按下确定按键发送 UDP 报文。当 ESP32-S3 设备接收到这个报文时，系统会提取该报文的 SSID 和密码去连接该网络。下图是 ESP32-S3 配网成功效果图。

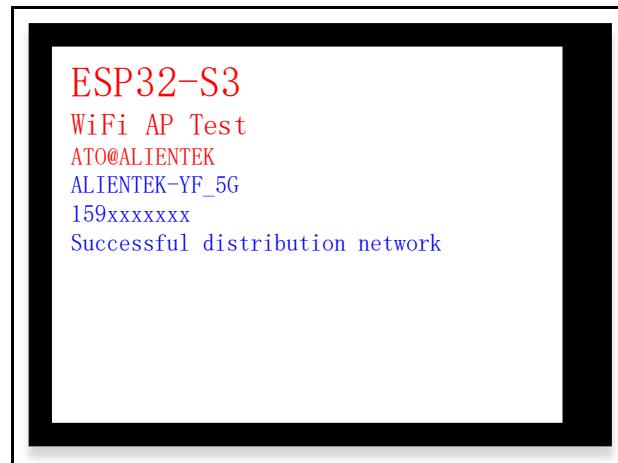


图 35.4.2 配网成功

