ESP-Techpedia_CN

Table of contents

Table of contents i

1 ESP-FAQ 3

2 ESP Friends 5

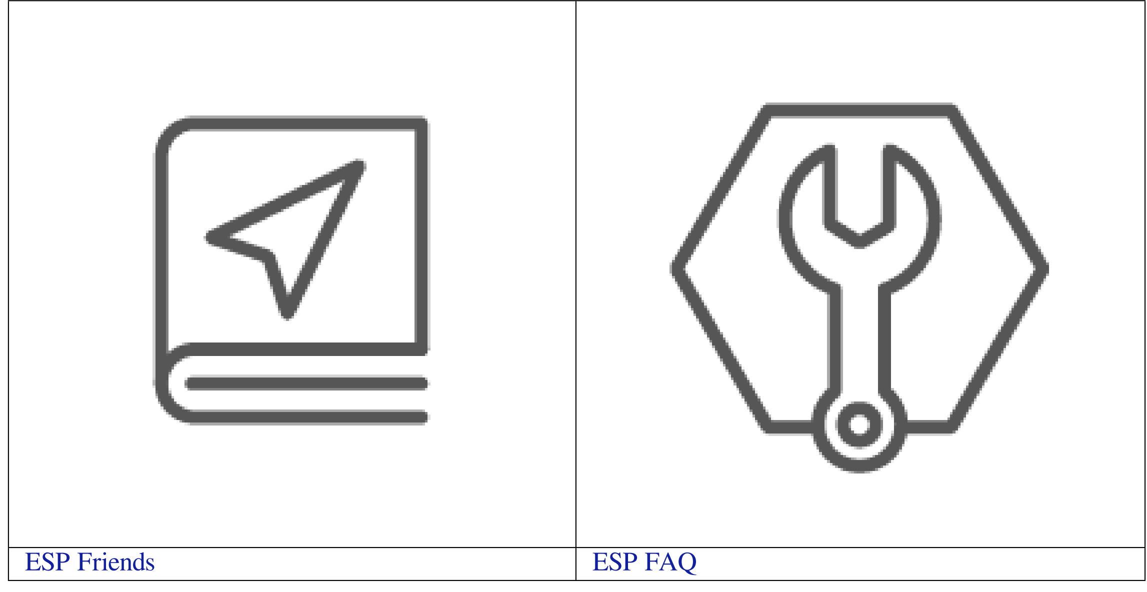
2.1 基础入门 5
2.1.1 硬件选型 5
2.1.2 固件烧录 19
2.1.3 环境搭建 35
2.1.4 项目开发入门 39
2.2 进阶开发 42
2.2.1 组件管理和使用 42
2.2.2 进阶代码调试 46
2.2.3 性能优化 83
2.3 方案介绍 84
2.3.1 模板 84
2.4 工具推荐 84
2.4.1 Windows 上的 Wireshark 抓包教程 84

ESP 技术百科是乐鑫（Espressif Systems）汇总归纳各类技术文档的综合平台，致力于为开发者提供全面、准确的乐鑫相关技术信息。我们不仅涵盖上手软件开发所需的文档，还收集了常见技术问答，为开发者解答在使用乐鑫芯片、模组或开发板过程中的疑问。

在 ESP 技术百科上，您可以轻松访问到各类乐鑫技术文档，如 ESP Friends 文档，可以帮助您快速上手乐鑫开发；而在 ESP-FAQ 中，您可以找到常见问题的解决方案。

除了汇总乐鑫各类技术文档，ESP-Techpedia 还欢迎来自世界各地开发者的新文档需求。我们希望通过紧密的互动，为开发者提供更多有价值的技术文档，让您轻松进行开发。

无论您是初学者还是经验丰富的开发者，ESP-Techpedia 都将成为您不可或缺的技术参考，助力您更高效地实现项目目标。



ESP Friends	ESP FAQ

Chapter 1

ESP-FAQ

请参考 ESP-FAQ 文档。

Chapter 2

ESP Friends

ESP Friends 文档可以帮助开发者快速上手乐鑫开发，如指导如何进行 ESP 硬件选型，环境搭建，软件入门等，后续也会新增 ESP 方案推荐等进阶开发文档，敬请期待！

2.1 基础入门

在这一部分，我们提供了针对 ESP 初学者的基础入门指南。无论您是第一次接触还是想要巩固基础知识，这个部分将引导您了解 ESP 基础操作。您将学到关键的基础知识，包括硬件选型、固件烧录、环境搭建等。

2.1.1 硬件选型

在日新月异的物联网市场中，乐鑫推出了一系列各具特色的 ESP 芯片，以全面满足不断变化的需求。选择合适的 ESP 芯片显得尤为重要，因为这将直接影响产品的性能和功能。根据项目的 **应用场景**、**功耗**、**无线通信**、**GPIO 和内存需求**，选择适合的 ESP 芯片。

为了帮助开发者更好地了解 ESP 系列芯片和模组，以下是一个简要的对比图。

表 1: 芯片对比

芯片	发布时间	应用场景	无线功能	GPIO	SRAM	支持 PSRAM
ESP32-C6	2023	具有长久续航能力的超低功耗物联网设备、Thread 边界路由器、Matter 网关、Zigbee 网桥	BLE 5.0 + Wi-Fi 6 + Thread + Zigbee	23	512 KB	×
ESP32-C2	2022	插座、照明、传感器、简单的智能家电设备	BLE 5.0 + Wi-Fi 4	14	272 KB	×
ESP32-H2	2021	Thread 边界路由器、Matter 网关、Zigbee 网桥	BLE 5.0 + Thread + Zigbee	19	230 KB	✓
ESP32-S3	2020	智能摄像头、人脸识别、语音识别、语音唤醒、实时数据采集处理、复杂外设控制	BLE 5.0 + Wi-Fi 4	36	320 KB	✓
ESP32-S2	2020	实时数据采集处理、复杂外设控制	Wi-Fi 4	36	320 KB	✓
ESP32-C3	2020	电工照明、开关插座、智能家电、工控领域	BLE 5.0 + Wi-Fi 4	15	400 KB	×
ESP32	2016	推荐使用最新发布的 ESP32-S3	BLE 4.2 + BT + Wi-Fi 4	26	520 KB	✓
ESP8266	2014	推荐使用最新发布的 ESP32-C2 或 ESP32-C3，ESP8266 即将到达 12 年的供货保证时间	Wi-Fi 4	11	160 KB	×

备注： 以上只是对 ESP 芯片系列的简要介绍。如果想进一步了解每个系列芯片或模组的具体细节和特点，可以使用 ESP 芯片 & 模组选型工具 来轻松获取相关资讯。该工具将根据项目需求和技术规格，选择最适合开发者应用的 ESP 芯片。

芯片，模组，开发板

乐鑫官方提供了芯片、模组和开发板，它们在物联网应用的开发和部署过程中有着不同的用途和特点。

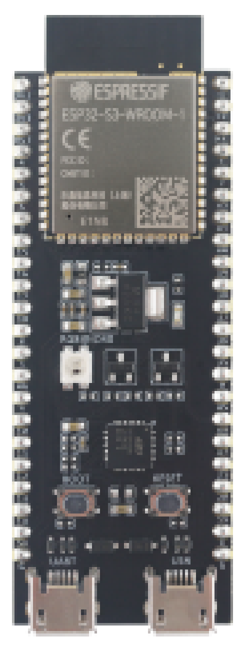
1. 芯片（Chip）：

- 芯片是乐鑫生产的基本集成电路（IC），它是整个 ESP 系列的核心。这些芯片通常包含处理器（CPU）、内存、通信接口、GPIO（通用输入输出）等硬件功能。芯片可以直接嵌入到自定义的电路板中，实现高度定制化的物联网设备，适用于需要紧凑尺寸和特定功能的项目。
- 芯片不能直接上电使用，需要连接其他必要的外部元件。而且使用芯片设计产品需要通过无线通信协议的认证，其过程可能会有些复杂。

2. 模组（Module）：

- 模组是乐鑫芯片的封装，集成了芯片、晶振、天线、flash。乐鑫的模组通常提供预先集成好的无线功能（如 Wi-Fi、蓝牙等），并具备 FCC、CE 等认证，因此开发者可以更加专注于应用程序的开发，而无需关注无线通信的细节，加快产品的上市速度。
- 和芯片相比在硬件设计和项目开发中具有更高的方便性。

3. 开发板（Development Board）：



- 开发板是一个集成了乐鑫模组的完整开发平台。它包含了用于调试、开发和测试的各种接口和资源，可以用于在开发阶段进行软件调试和烧录固件。通常在项目开发初期会通过开发板进行快速的测试和验证，进入到产品量产阶段时再使用模组进行集成。
- 同时，开发板也是为刚接触乐鑫芯片的开发者提供快速入门的工具。开发板可以迅速验证开发者的想法和设计，让创意快速成型。

选择指南 选择合适的芯片、模组或开发板取决于项目的需求、时间、技术能力和预算。下面提供一些在选择时需要考虑的因素：

1. 快速开发和原型验证：

- **开发板**非常有利于在项目初期进行快速的功能开发与验证

2. 硬件自定义设计：

   - 如果需要高度自定义的电路板和硬件设计，**芯片**是更合适的选择

备注： 自定义设计需要通过无线通信协议的认证，可能会增加开发时间和成本。

3. 上市速度：

- **模组**通常能够加速产品的上市速度，因为预先集成了无线功能（如 Wi-Fi、蓝牙）并具备相关认证。开发者可以更专注于应用程序的开发，而不必处理无线通信的细节

4. 成本预算：

- 使用 **芯片**通常成本较低，但自定义设计可能增加时间成本和开发难度。模组具有相对高一些的成本，但可以加速开发过程

5. 团队技术能力：

- 如果是初学者或团队技术资源有限，使用 **模组**更容易上手，加速项目进程并 **降低技术风险**。使用芯片需要更高的技术能力和更多的开发经验

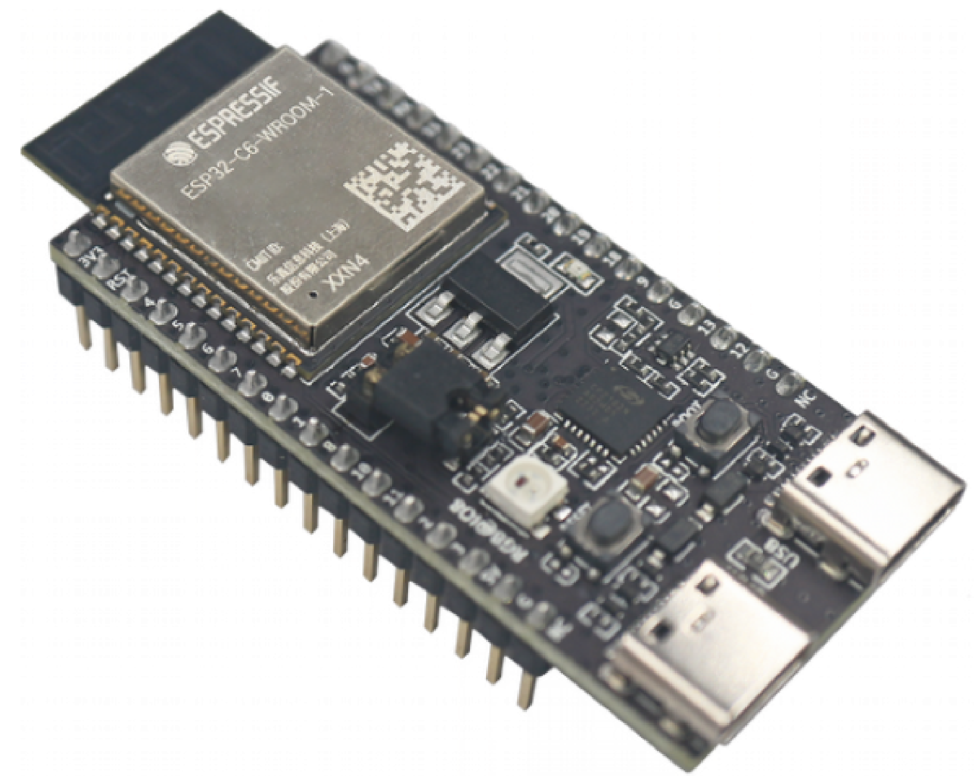
ESP32-C6

支持功能：

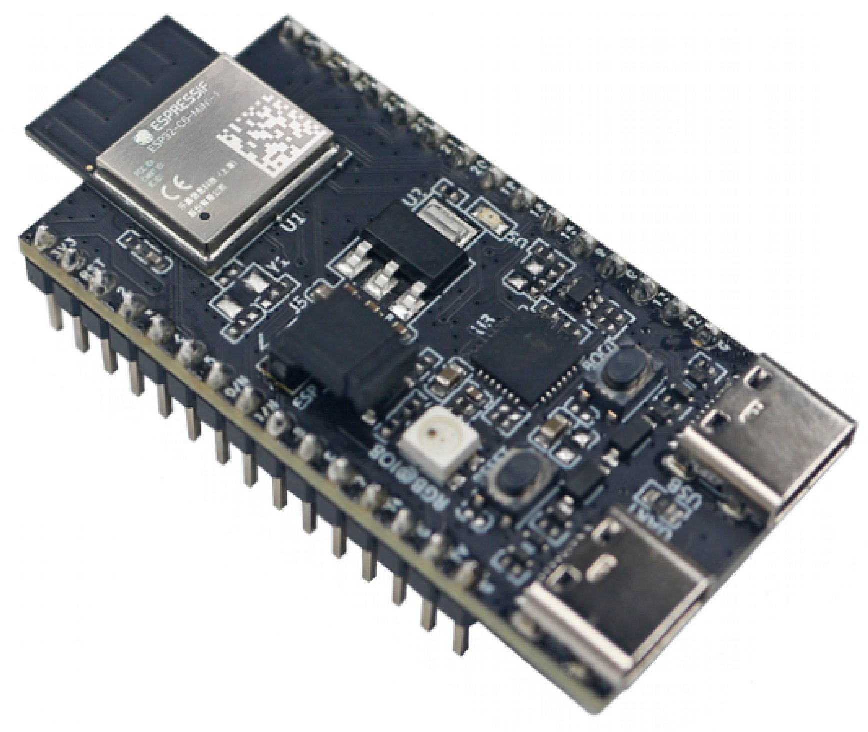
- 30 个 (QFN40) 或 22 个 (QFN32) 可编程 GPIO 管脚，支持 SPI、UART、I2C、I2S、RMT、TWAI 和 PWM
- 可用于开发方案：具有长久续航能力的超低功耗物联网设备、Thread 边界路由器、Matter 网关、Zigbee 网桥

开发板

- ESP32-C6-DevKitC-1 ： ESP32-C6-DevKitC-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



- ESP32-C6-DevKitM-1 ： ESP32-C6-DevKitCM-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



硬件设计指南

- ESP32-C6 硬件设计指南

购买链接：

- ESP32-C6 开发板

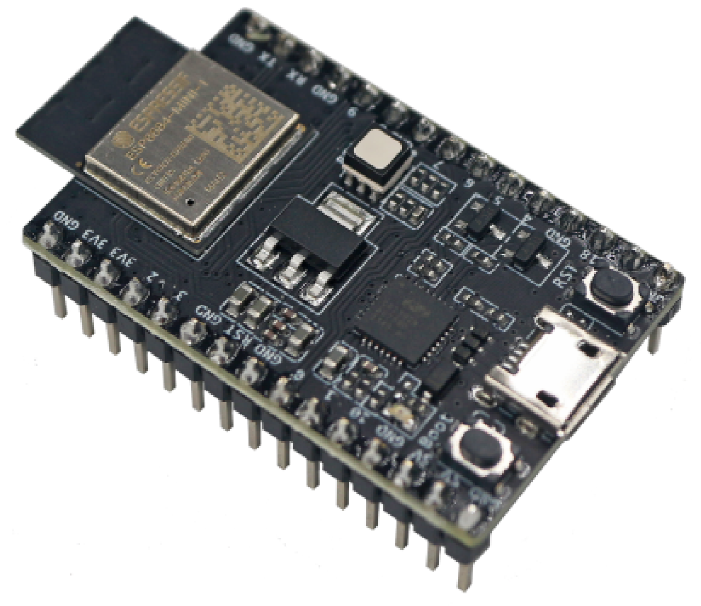
ESP32-C2

支持功能：

- 14 个可编程 GPIO 管脚：SPI、UART、I2C、LED PWM 控制器、SAR 模/数转换器、温度传感器
- 可用于开发方案：插座、照明、传感器、简单的智能家电设备

开发板

- ESP8684-DevKitM-1 ： ESP8684-DevKitM-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



硬件设计指南

- ESP32-C2 硬件设计指南

购买链接：

- ESP32-C2 开发板

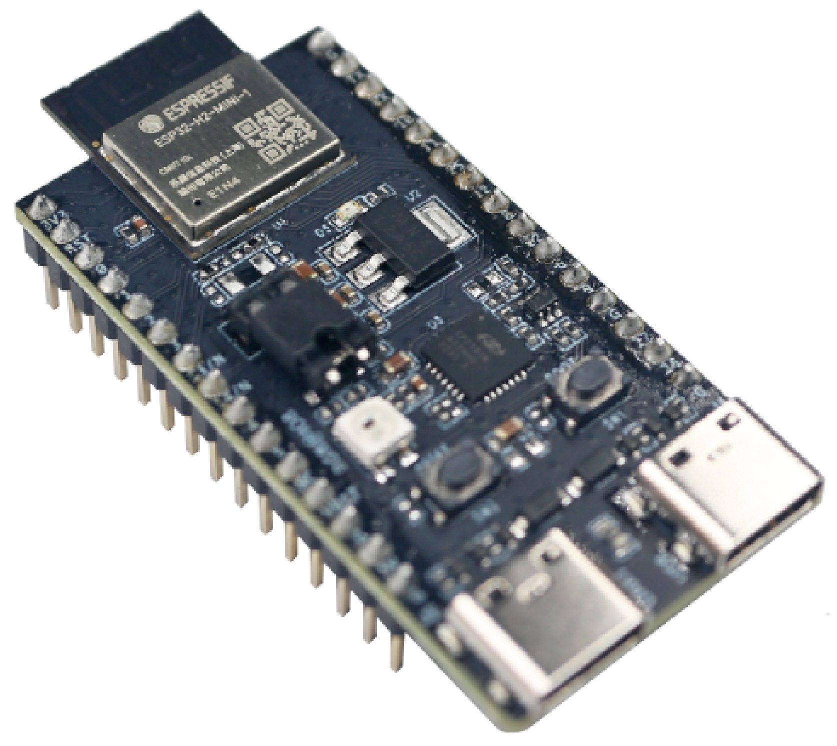
ESP32-H2

支持功能：

- 19 个可编程 GPIO，支持常用外设接口如 UART、SPI、I2C、I2S、红外收发器、LED PWM、全速 USB 串口/JTAG 控制器、GDMA、MCPWM
- 可用于开发方案：Thread 边界路由器、Matter 网关、Zigbee 网桥

开发板

- ESP32-H2-DevKitM-1 ：ESP32-H2-DevKitM-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



硬件设计指南

- ESP32-H2 硬件设计指南

购买链接：

- ESP32-H2 开发板

ESP32-S3

支持功能：

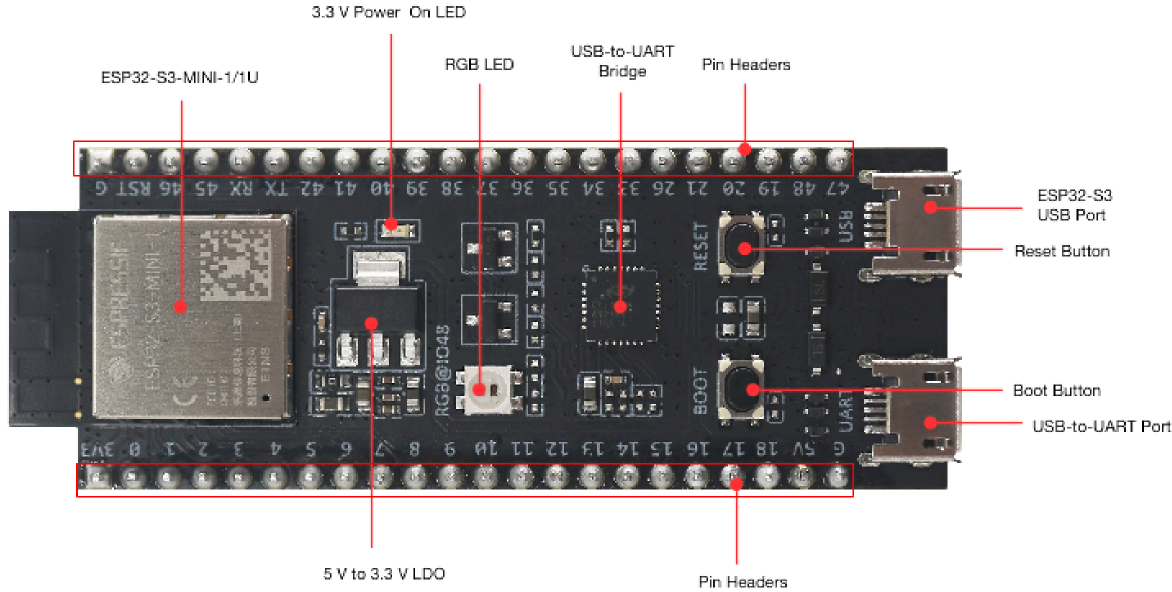
- 常用外设接口如 SPI、I2S、I2C、PWM、RMT、ADC、UART、SD/MMC 主机控制器和 TWAI 控制器等
- 可用于开发方案：智能摄像头、人脸识别、语音识别、语音唤醒、实时数据采集处理、复杂外设控制

开发板：

- EESP32-S3-DevKitC-1 ： ESP32-S3-DevKitC-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。

、

- ESP32-S3-DevKitM-1 ： ESP32-S3-DevKitM-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。
- ESP32-S3-BOX ： ESP-BOX 为用户提供了一个基于语音助手 + 触摸屏控制、传感器、红外控制器和智能 Wi-Fi 网关等功能，开发和控制智能家居设备的平台。



- ESP32-S3-EYE： ESP32-S3-EYE 是乐鑫推出的一款小型 AI（人工智能）开发板。开发板配置一个 2 百万像素的摄像头、一个 LCD 显示屏和一个麦克风，适用于图像识别和音频处理等应用。您可以使用 ESP-WHO 开发各种 AIoT（人工智能物联网）应用，例如智能门铃、监控系统、人脸识别打卡机等

- ESP32-S3-USB-OTG： ESP32-S3-USB-OTG 是一款侧重于 USB-OTG 功能验证和应用开发的开发板，基于 ESP32-S3 SoC，支持 Wi-Fi 和 BLE 5.0 无线功能，支持 USB 主机和 USB 从机功能。可用于开发无线存储设备，Wi-Fi 网卡，LTE MiFi，多媒体设备，虚拟键鼠等应用。
- ESP32-S3-Korvo-1： ESP32-S3-Korvo-1 是乐鑫推出的一款 AI（人工智能）开发板，搭载 ESP32-S3 芯片和乐鑫语音识别 SDK ESP-Skainet。ESP32-S3-Korvo-1 支持中英文语音唤醒和离线语音命令识

别。您可以使用 ESP-Skainet 开发各种语音识别应用，例如智能屏幕、智能插头、智能开关等。

- ESP32-S3-Korvo-2： ESP32-S3-Korvo-2 是一款基于 ESP32-S3 芯片的多媒体开发板，搭载双麦克风阵列，支持语音识别和近/远场语音唤醒。同时它还搭载 LCD、摄像头、microSD 卡等外设，可支持基于 JPEG 的视频流处理，满足用户对低成本、低功耗、联网的音视频产品开发需求。

硬件设计指南

- ESP32-S3 硬件设计指南

购买链接：

- ESP32-S3 开发板

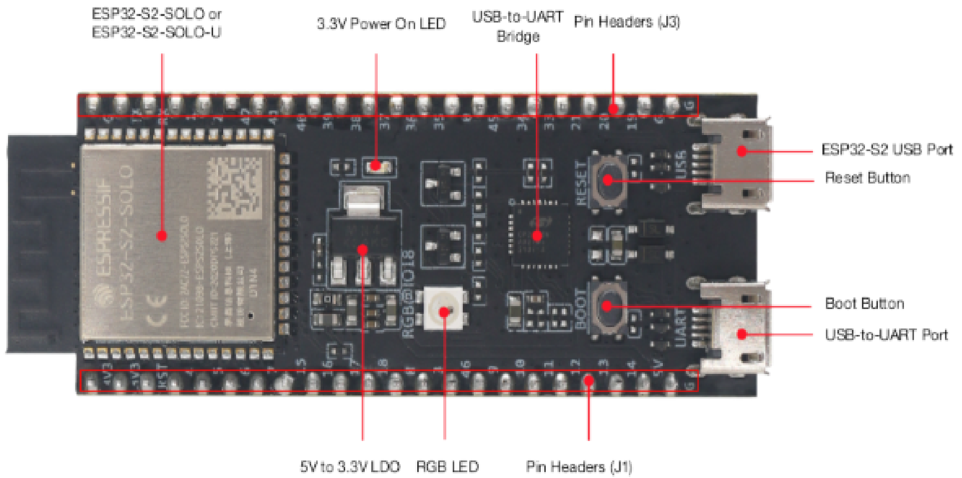
ESP32-S2

支持功能：

- 全速 USB OTG 接口，SPI，I2S，UART，I2C，LED PWM，LCD 接口，Camera 接口，ADC，DAC，触摸传感器
- 可用于开发方案：实时数据采集处理、复杂外设控制

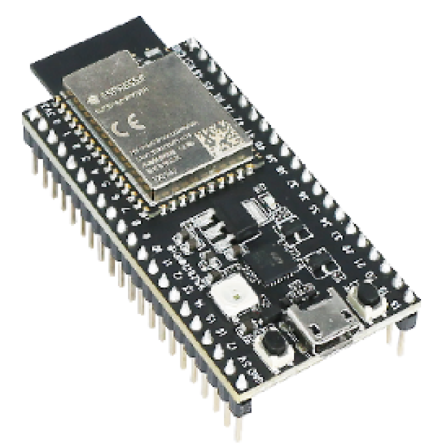
开发板：

- ESP32-S2-DevKitC-1 ： ESP32-S2-DevKitC-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



- ESP32-S2-HMI-DevKit-1：ESP32-S2-HMI-DevKit-1 面向 GUI 应用场景设计，可实现智能家居交互面板，带显示屏的音箱、闹钟等人机交互界面的智能控制。该开发板具有丰富的板载传感器和拓展接口，方便用户快速进行二次开发，实现多样的功能。

- ESP32-S2-Saola-1：ESP32-S2-Saola-1 是乐鑫一款基于 ESP32-S2 的小型开发板，可以用来烧录和体验 IDF 中的 examples。



硬件设计指南

- ESP32-S2 硬件设计指南

购买链接：

- ESP32-S2 开发板

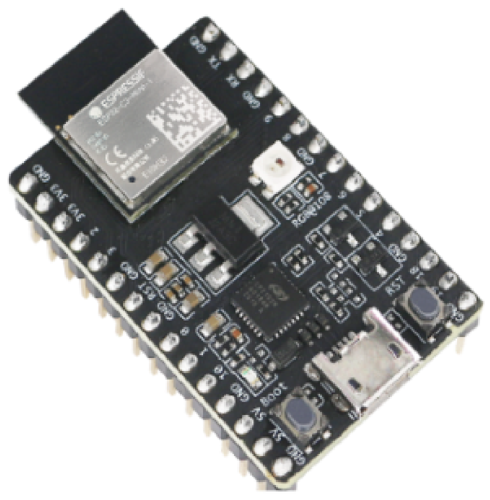
ESP32-C3

支持功能：

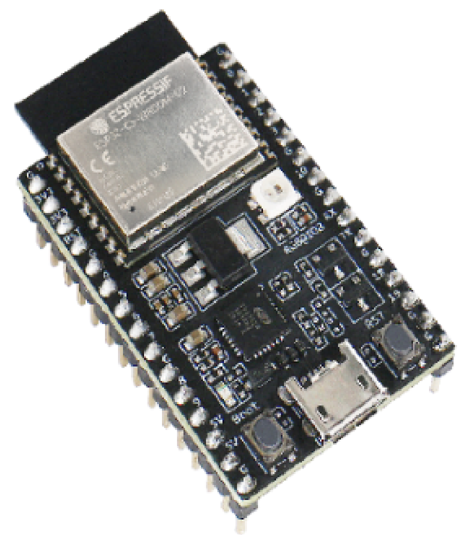
- 丰富的通信接口及 GPIO 管脚，支持多个外部 SPI、Dual SPI、Quad SPI、QPI flash
- 可用于开发方案：电工照明、开关插座、智能家电、工控领域

开发板

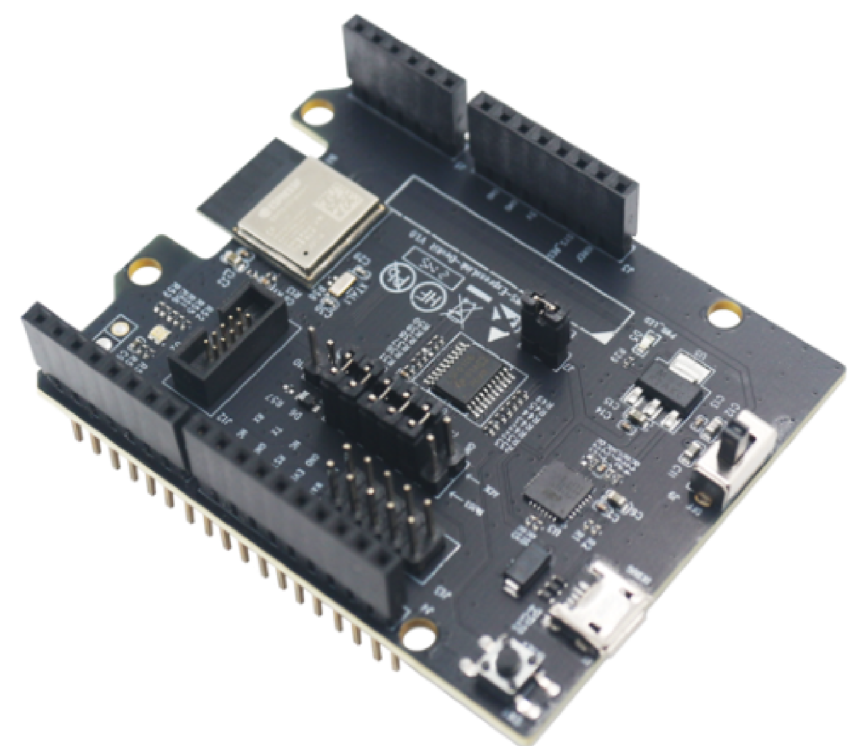
- ESP32-C3-DevKitM-1 ： ESP32-C3-DevKitM-1 是一款入门级开发板，使用以尺寸小而得名的 ESP32-C3-MINI-1 模组。可以用来烧录和体验 IDF 中的 examples。



- ESP32-C3-DevKitC-02 ： ESP32-C3-DevKitC-02 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。



- ESP32-C3-DevKit-RUST-1： ESP32-C3-DevKit-RUST-1 是一款入门级开发板，可以用来烧录和体验 IDF 中的 examples。
- ESP32-C3-AWS-ExpressLink-DevKit： ESP32-C3-AWS-ExpressLink-DevKit 使用抽象的应用程序编程接口（API）将任何主机应用程序连接到 AWS IoT Core 及其服务。它具有 Arduino 扩展板的形状，因此可以直接插在标准 Arduino 上。它也可以与 Raspberry Pi 或任何其他主机一起使用。



硬件设计指南

- ESP32-C3 硬件设计指南

购买链接：

- ESP32-C3 开发板

ESP32

支持功能:

- 提供了多个 GPIO 引脚，包括数字输入/输出、模拟输入、PWM 输出、I2C、SPI、UART 等。
- 开发方案：推荐使用最新发布的 ESP32-S3

开发板

- ESP32-DevKitC ： ESP32-DevKitC V4 是一款基于 ESP32 的小型开发板，可以用来烧录和体验 IDF 中的 examples。
- ESP-EYE ： ESP-EYE 是一款面向人脸识别和语音识别市场的开发板，搭载 200 W 像素摄像头、数字麦克风，可满足各种 AI 应用开发需求。此外，该开发板还支持 Wi-Fi 图像传输、Micro USB 调试和供电，可以实现语音唤醒、人脸检测与识别等功能，可协助用户开发高度集成的 AI 解决方案。
- ESP32-LyraT ： ESP32-LyraT 专为音频应用市场打造。提供了音频编解码芯片，板载双麦克风，耳机输出，2 个 3-watt 扬声器输出，双辅助输入，和锂电池充电管理等硬件支持。
- 此外，还有 ESP32 系列还有其他七款用于音频处理的开发板，不过我们建议开发者使用最新的 ESP32-S3 系列的音频开发板。
- ESP32-LCDKit ： ESP32-LCDKit 是一款以 ESP32-DevKitC 为核心的 HMI（人机交互）开发板，可外接屏幕，并且集成了 SD-Card、DAC-Audio 等外设，主要用于 HMI 相关开发与评估.
- ESP32-Ethernet-Kit ： ESP32-Ethernet-Kit 是一款以太网转 Wi-Fi 开发板，可为以太网设备赋予 Wi-Fi 连接功能。为了提供更灵活的电源选项，ESP32-Ethernet-Kit 同时也支持以太网供电 (PoE)。

硬件设计指南

- ESP32 硬件设计指南

购买链接:

- ESP32 开发板

ESP8266

支持功能:

- 提供了多个 GPIO 引脚，可以用于多种用途，例如 UART、I2C、SPI 等
- 开发方案：推荐使用最新发布的 ESP32-C2 或 ESP32-C3

开发板

- ESP8266-DevKitC ： ESP8266-DevKitC 是一款紧凑型 ESP8266 开发板，可以用来烧录和体验 IDF 中的 examples。

硬件设计指南

- ESP8266 硬件设计指南

购买链接:

- ESP8266 开发板

2.1.2 固件烧录

此文档介绍了各 ESP 芯片进行固件烧录时需要满足的硬件及软件环境。

烧录不同芯片需满足的硬件环境

如果使用的是芯片或者模组，请选择对应的 ESP 系列芯片进行硬件接线配置。

ESP8266

ESP8266 默认是通过 UART0（即 TX0（GPIO1）和 RXD（GPIO3））来下载固件的

硬件接线 需要满足以下接线条件：

```
VDD     -> 3V3
GND     -> GND   (与供电板“共地”)
EN      -> 拉高   (用于上电启动、不可浮空)
GPIO0   -> 拉低   (进入下载模式)
GPIO15  -> 拉低
TXD0    -> RX
RXD0    -> TX
```

启动条件

- ESP8266 芯片工作电压范围为 2.5 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。
- ESP8266 模组工作电压范围为 2.7 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。

更多详情可参考 硬件接线原理总结：

- ESP8266 固件烧录需满足的硬件环境整理

官方文档：

- ESP8266 技术规格书
- ESP8266 硬件设计指南

ESP32

ESP32 默认是通过 UART0（即 TX0（GPIO1）和 RXD（GPIO3））来下载固件的。

硬件接线 需要满足以下接线条件：

```
VDD     -> 3V3
GND     -> GND   (与供电板“共地”)
EN      -> 拉高   (用于上电启动、不可浮空)
GPIO0   -> 拉低   (进入下载模式)
GPIO2   -> 拉低
TXD0(GPIO1)    -> RX
RXD0(GPIO3)    -> TX
```

备注： Strapping 管脚 GPIO12 用于选择 VDD_SPI Flash 电压。

- 当使用 1.8V VDD_SPI Flash 时，GPIO12 需要在芯片上电时拉高。
- 当使用 3.3V VDD_SPI Flash 时，GPIO12 需要在芯片上电时拉低。

启动条件

- ESP32 芯片工作电压范围为 2.3 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。
- ESP32 模组工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。

参考资料 硬件接线原理总结：

- ESP32 固件烧录需满足的硬件环境整理

官方文档：

- ESP32 技术规格书
- ESP32 硬件设计指南

ESP32-S2

ESP32-S2 支持 UART0 和 USB 两种固件下载方式。

硬件接线 当使用 UART0 管脚下载固件时，需要满足以下接线条件：

```
VDD     -> 3V3
GND     -> GND   (与供电板“共地”)
EN      -> 拉高   (用于上电启动、不可浮空)
GPIO0   -> 拉低   (进入下载模式)
GPIO46  -> 拉低
TXD0(GPIO43)    -> RX
RXD0(GPIO44)    -> TX
```

当使用 USB 管脚下载固件时，需要满足以下接线条件：

```
VDD     -> 3V3
GND     -> GND   (与供电板“共地”)
EN      -> 拉高   (用于上电启动、不可浮空)
GPIO0   -> 拉低   (进入下载模式)
GPIO46  -> 拉低
GPIO19    -> USB_D-
GPIO20    -> USB_D+
```

备注： Strapping 管脚 GPIO45 用于选择 VDD_SPI Flash 电压。

- 当使用 1.8V VDD_SPI Flash 时，GPIO45 需要在芯片上电时拉高。
- 当使用 3.3V VDD_SPI Flash 时，GPIO45 需要在芯片上电时拉低。

启动条件

- ESP32-S2 芯片的工作电压范围为 2.8 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。
- ESP32-S2 模组的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。

参考资料 硬件接线原理总结：

- ESP32-S2 固件烧录需满足的硬件环境整理
- ESP32-S2 USB & UART 下载总结

官方文档：

- ESP32-S2 技术规格书
- ESP32-S2 硬件设计指南

ESP32-C3

ESP32-C3 支持 UART0 和 USB 两种固件下载方式。

硬件接线 当使用 UART0 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO2    -> 拉高   (控制 SPI 启动模式)
GPIO8    -> 拉高
GPIO9    -> 拉低   (进入下载模式)
TXD0(GPIO21)    -> RX
RXD0(GPIO20)    -> TX
```

当使用 USB 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO2    -> 拉高   (控制 SPI 启动模式)
GPIO8    -> 拉高
GPIO9    -> 拉低   (进入下载模式)
GPIO18   -> USB_D-
GPIO19   -> USB_D+
```

备注：

- 给芯片/模组上电后，可通过 UART0 串口查看是否进入 Download Boot 模式。
- 在芯片上电启动时，GPIO8 和 GPIO9 不可以同时为低电平。

启动条件

- ESP32-C3 芯片工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上
- ESP32-C3 模组工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上

参考资料 硬件接线原理总结：

- ESP32C3 固件烧录需满足的硬件环境整理
- ESP32C3 USB & UART 下载总结

官方文档：

- ESP32-C3 技术规格书
- ESP32-C3 硬件设计指南

ESP32-S3

ESP32-S3 支持 UART0 和 USB 两种固件下载方式。

硬件接线 当使用 UART0 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   （与供电板“共地”）
EN       -> 拉高   （用于上电启动、不可浮空）
GPIO0    -> 拉低   （进入下载模式）
GPIO46   -> 拉低
TXD0(GPIO43)     -> RX
RXD0(GPIO44)     -> TX
```

当使用 USB 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   （与供电板“共地”）
EN       -> 拉高   （用于上电启动、不可浮空）
GPIO0    -> 拉低   （进入下载模式）
GPIO46   -> 拉低
GPIO19     -> USB_D-
GPIO20     -> USB_D+
```

备注： Strapping 管脚 GPIO45 用于选择 VDD_SPI Flash 电压。

- 当使用 1.8V VDD_SPI Flash 时，GPIO45 需要在芯片上电时拉高。
- 当使用 3.3V VDD_SPI Flash 时，GPIO45 需要在芯片上电时拉低。

启动条件

- ESP32-S3 芯片的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。
- ESP32-S3 模组的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议电源电压为 3.3 V，输出电流需要达到 500 mA 及以上。

参考资料 硬件接线原理总结：

- ESP32S3 固件烧录需满足的硬件环境整理
- ESP32-S3 USB & UART 下载总结

官方文档：

- ESP32-S3 技术规格书
- ESP32-S3 硬件设计指南

ESP32-C2

ESP32-C2 默认通过 UART0 （即 TXD（GPIO20）和 RXD（GPIO19））下载固件。

硬件接线 需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   （与供电板“共地”）
EN       -> 拉高   （用于上电启动、不可浮空）
GPIO8    -> 拉高   （默认浮空）
GPIO9    -> 拉低   （默认为高）
TXD0(GPIO20)     -> RX
RXD0(GPIO19)     -> TX
```

备注：

- 给芯片/模组上电后，可通过 UART0 串口查看是否进入 Download Boot 模式。
- 在芯片上电启动时，GPIO8 和 GPIO9 不可以同时为低电平。

启动条件

- ESP32-C2 芯片的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-C2 系列芯片的电源电压为 3.3 V，额定输出电流最好在 500 mA 及以上。
- ESP32-C2 模组的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-C2 系列芯片的电源电压为 3.3 V，额定输出电流最好在 500 mA 及以上。

参考资料 硬件接线原理总结：

- ESP32C2 固件烧录需满足的硬件环境整理

官方文档：

- ESP32-C2 技术规格书
- ESP32-C2 硬件设计指南

ESP32-H2

ESP32-H2 支持 UART0 和 USB 两种固件下载方式。

硬件接线 当使用 UART0 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO8    -> 拉高   (默认浮空)
GPIO9    -> 拉低   (默认为高)
TXD0(GPIO24)     -> RX
RXD0(GPIO23)     -> TX
```

当使用 USB 管脚下载固件时，需要满足以下接线条件：

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO8    -> 拉高   (默认浮空)
GPIO9    -> 拉低   (默认为高)
GPIO26   -> USB_D-
GPIO27   -> USB_D+
```

备注：

- 给芯片/模组上电后，可通过 UART0 串口查看是否进入 Download Boot 模式。
- 在芯片上电启动时，GPIO8 和 GPIO9 不可以同时为低电平。

启动条件

- ESP32-H2 芯片的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-H2 系列芯片的电源电压为 3.3 V，额定输出电流最好在 350 mA 及以上。
- ESP32-H2 模组的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-H2 系列芯片的电源电压为 3.3 V，额定输出电流最好在 350 mA 及以上。

参考资料 硬件接线原理总结:

- ESP32H2 固件烧录需满足的硬件环境整理

官方文档:

- ESP32-H2 技术规格书
- ESP32-H2 硬件设计指南

ESP32-C6

ESP32-C6 支持 UART0 和 USB 两种固件下载方式。

硬件接线 需要满足以下接线条件:

当使用 UART0 管脚下载固件时，需要满足以下接线条件:

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO8    -> 拉高   (默认浮空)
GPIO9    -> 拉低   (默认是``高电平``)
TXD0(GPIO16)     -> RX
RXD0(GPIO17)     -> TX
```

当使用 USB 管脚下载固件时，需要满足以下接线条件:

```
VDD      -> 3V3
GND      -> GND   (与供电板“共地”)
EN       -> 拉高   (用于上电启动、不可浮空)
GPIO8    -> 拉高   (默认浮空)
GPIO9    -> 拉低   (默认是``高电平``)
GPIO12   -> USB_D-
GPIO13   -> USB_D+
```

备注:

- 给芯片/模组上电后，可通过 UART0 串口查看是否进入 Download Boot 模式。
- 在芯片上电启动时，GPIO8 和 GPIO9 不可以同时为低电平。

启动条件

- ESP32-C6 芯片的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-C6 系列芯片的电源电压为 3.3 V，额定输出电流最好在 500 mA 及以上。
- ESP32-C6 模组的工作电压范围为 3.0 V ~ 3.6 V；使用单电源供电时，建议供给 ESP32-C6 系列芯片的电源电压为 3.3 V，额定输出电流最好在 500 mA 及以上。

参考资料 硬件接线原理总结:

- ESP32C6 固件烧录需满足的硬件环境整理

官方文档:

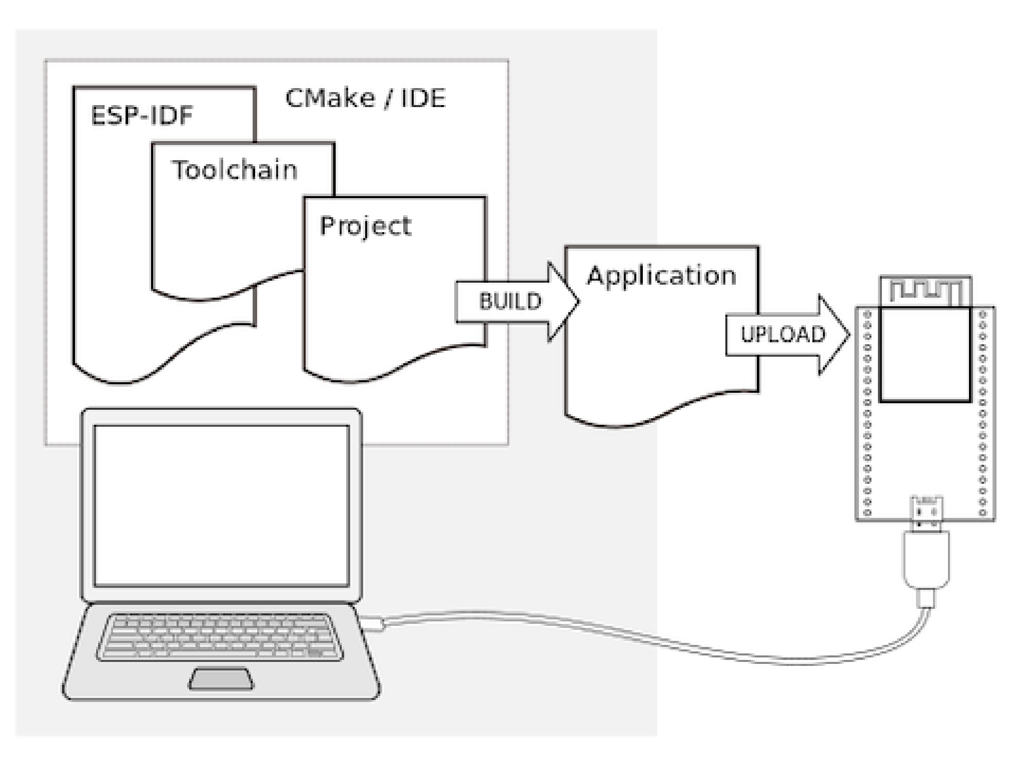
- ESP32-C6 技术规格书
- ESP32-C6 硬件设计指南

选择合适的烧录平台

乐鑫官方针对不同需求提供以下几种烧录方式：

1. **快速固件测试：** *ESP LAUNCHPAD 固件快速烧录*
   - 优势：简单快捷，无需搭建环境或额外工具，通过 Web 界面轻松烧录固件
2. **非开发人员执行烧录：** *Flash Download Tool 烧录*
   - 优势：提供了图形化界面，无需深入了解命令行操作，同时也适用于自动化部署和批量生产烧录
3. **专业开发人员**进行全面的 **固件开发：** *IDE 烧录*
   - 优势：集成开发环境（IDE）提供了直观的用户界面和工具集，支持固件开发、调试和烧录一体化
4. **专业开发人员**进行全面的 **固件开发和更详细的配置：** *IDF 终端烧录*
   - 优势：允许开发人员精确管理项目，包括固件参数设置。方便在开发和调试过程中进行烧录，以快速验证更改

准备工作



备注： 如果还没为项目选好合适的开发板，请参考*ESP 硬件选型*

芯片，模组，和开发板的区别请参考*芯片，模组，开发板*

使用开发板 用数据线连接 **USB** 或者 **UART** 接口便可进行烧录。

备注： 目前一些开发板使用的是 USB Type C 接口。请确保使用合适的数据线来连接开发板！

使用芯片或模组 乐鑫芯片一般具备两种模式：

1. 固件烧录模式：芯片将等待从串口接收固件，以进行烧录。通常需要使用烧录工具发送固件数据
2. 正常运行模式：这是芯片的正常工作模式，它会从 Flash 存储中加载固件并运行

使用芯片或者模组需要通过设置 **特定的引脚**的状态进入烧录模式，通过 **UART0** 或者 **USB** 外设来进行烧录。

备注： 具体的引脚信息需要参考对应芯片的技术手册，在 乐鑫官网 上可以找到。

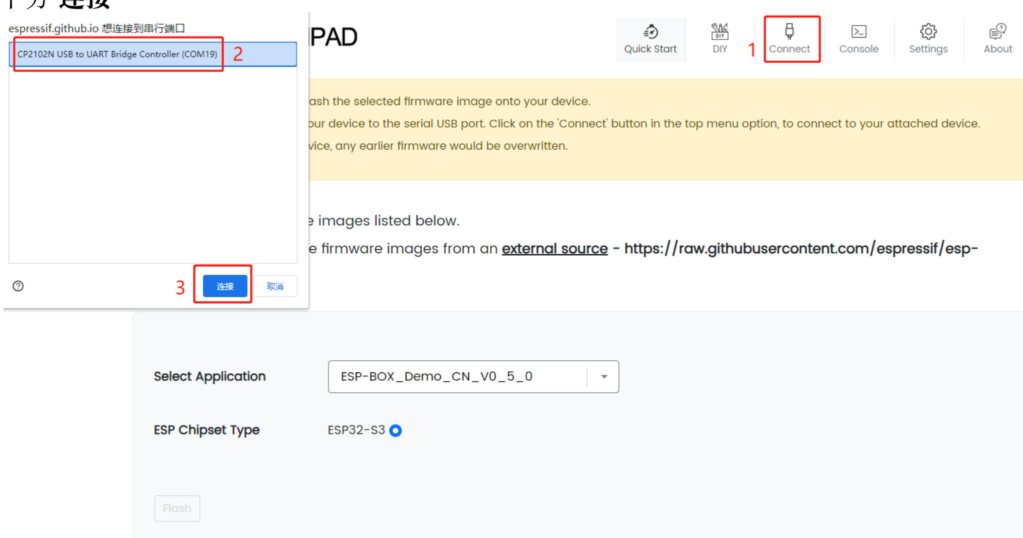
ESP LAUNCHPAD 固件快速烧录 ESP LAUNCHPAD 是一个让开发者快速体验乐鑫芯片功能平台。

- 选择乐鑫官方提供的一些示例固件后可直接在网页上进行烧录
- 开发者还可以上传自己编译好的 .bin 文件后在网页上进行烧录

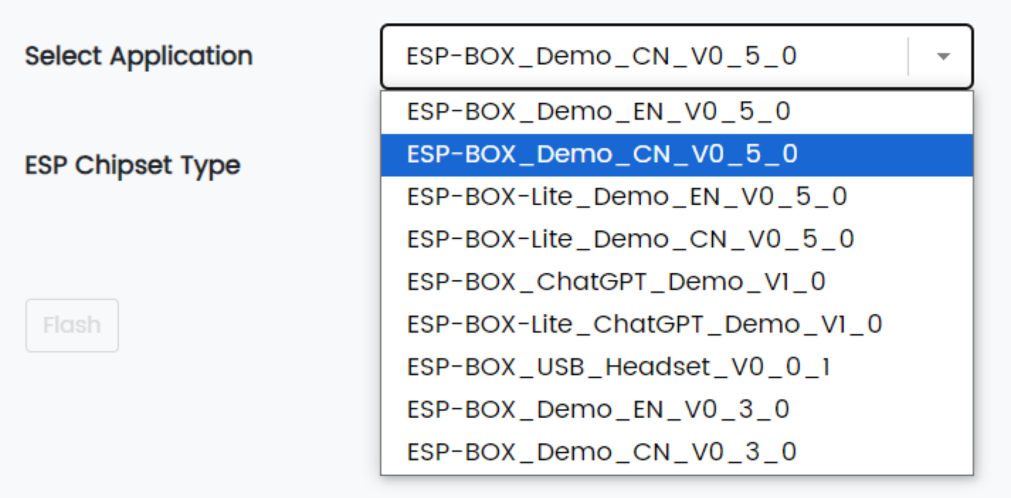
ESP LAUNCHPAD 烧录过程 下面用 ESP-BOX 为例：

打开网站 ESP LAUCHPAD

1. 点击右上方 **Connect**
2. 在弹出的窗口中选择连接 ESP-BOX 的 **端口**
3. 点击下方 **连接**

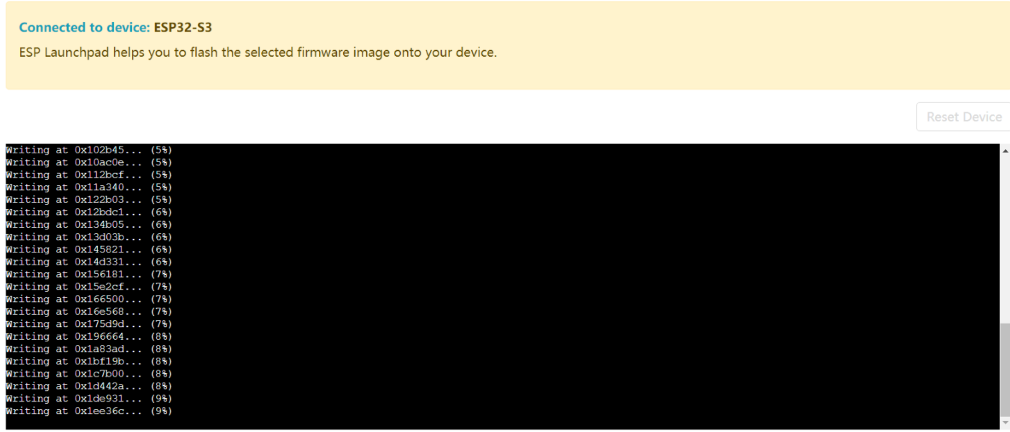


4. 选择要烧录到 ESP-BOX 的固件



5. 选择 **ESP Chipset Type** 为 ESP32-S3

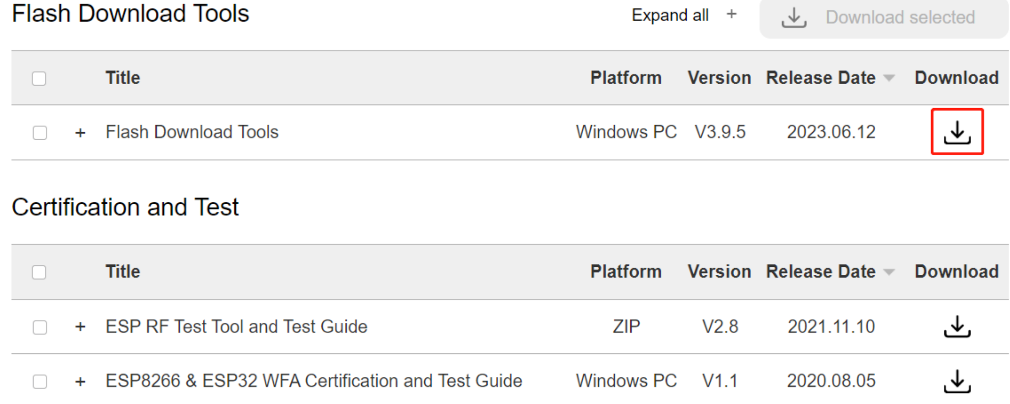
6. 点击左下方 **Flash** 键开始烧录
7. 烧录开始后会跳转到 console 界面，正确的话会显示烧录进度



8. 烧录完成后按下左上方 **Reset Device** 键，或者点击右上角 **Disconnect**。

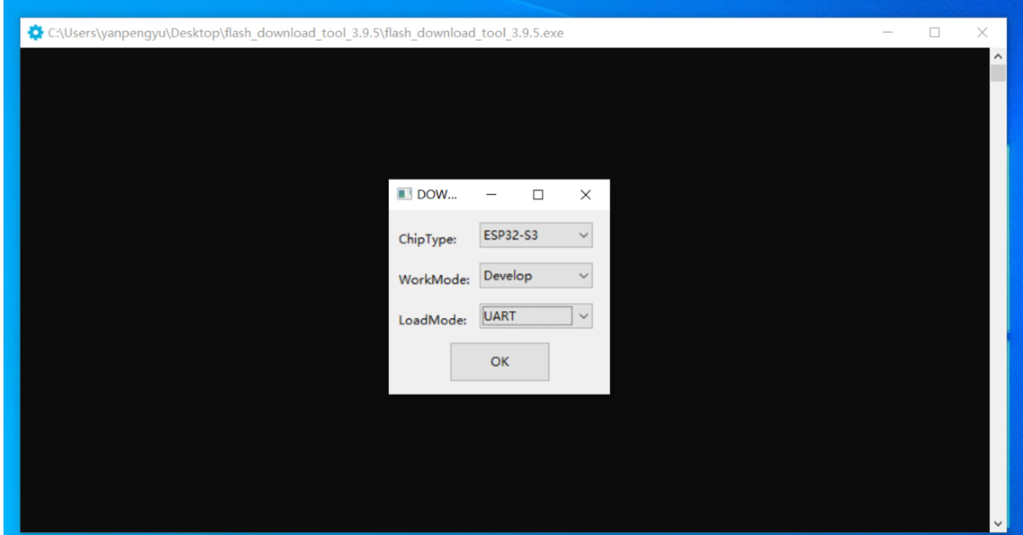
Flash Download Tool 烧录 Flash Download Tool 是由乐鑫开发的一款用于将固件烧录到其 ESP 系列芯片的工具，用于将预先编译好的固件文件加载到芯片的闪存（Flash）中，从而使芯片能够正确地运行应用程序或固件。

- 下载 Flash download tool

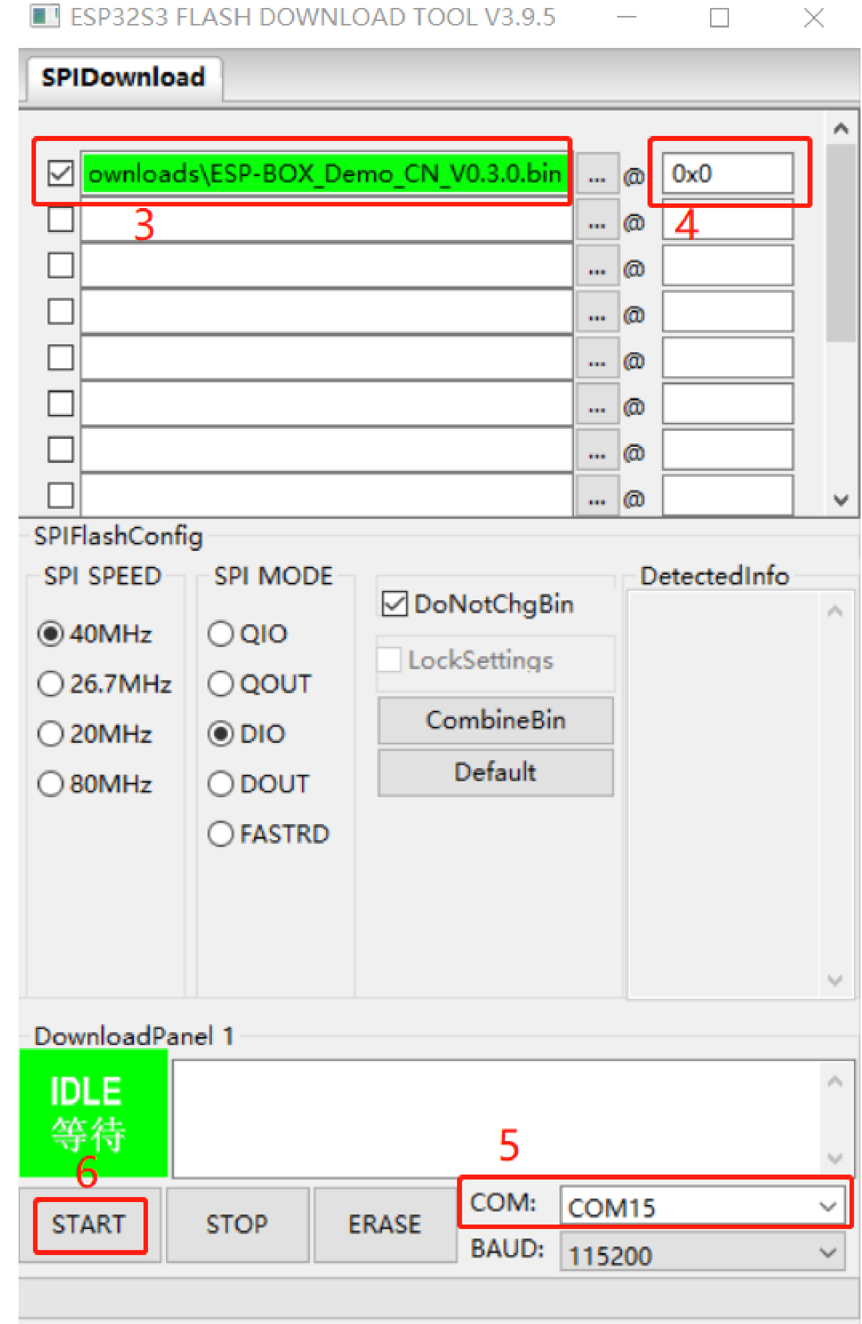


使用 Flash download tool 烧录过程

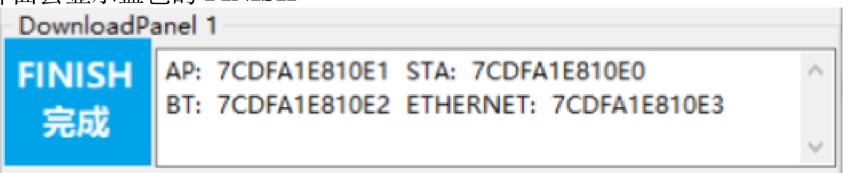
1. 运行 **flash_download_tool** 的 `.exe` 文件
2. 选择 **开发板（ChipType）** 为对应的开发板，一般芯片上会有标识。选择 **烧录模式（LoadMode）** 为 `UART` 后点击下方 **OK**。



3. 点击选择要烧录的 `.bin` 文件
4. 输入要烧录的 **地址偏移量**（**address offset**），可以默认输入起始位 `0x0`。
5. 选择开发板连接的 **端口 COM**
6. 点击左下方 **START** 开始烧录



7. 烧录完成后界面会显示蓝色的 **FINISH**



使用 Flash download tool 进行 Flsah 加密和安全启动

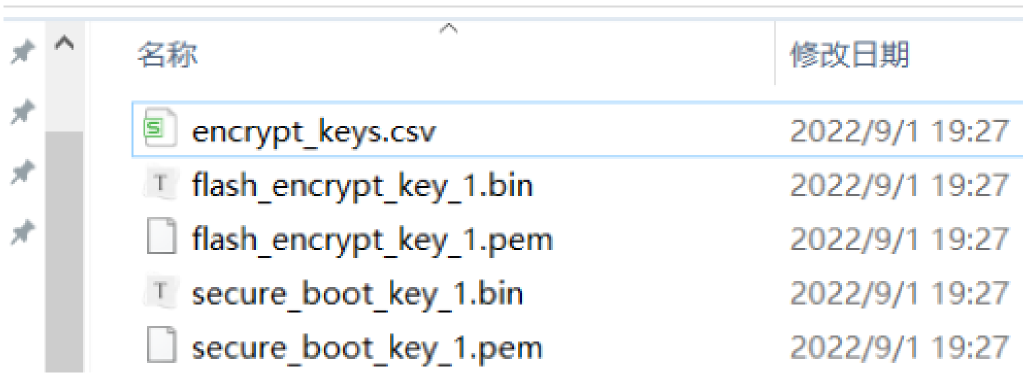
1. 在 **menuconfig** 中禁用所有安全启动和 Flash 加密配置，确保编译后可以生成明文固件
2. 调整 **默认分区表（Offset of partition table）**的 **偏移量**，从 `0x8000` 调整为 `0xa000`

> **备注：** Flash 加密会增加引导加载程序（bootloader）.bin 固件的大小，所以必须为偏移地址留了足够大的空间

3. 打开 **Flash Download Tool** 的文件夹，打开 **configure -> esp 芯片类型 -> security.conf** 。启用以下配置并保存文件：

```
[SECURE BOOT]
secure_boot_en = True
[FLASH ENCRYPTION]
flash_encryption_en = True
reserved_burn_times = 3
[ENCRYPTION KEYS SAVE]
keys_save_enable = True
encrypt_keys_enable = False
encrypt_keys_aeskey_path =
[DISABLE FUNC]
jtag_disable = False
dl_encrypt_disable = False
dl_decrypt_disable = False
dl_cache_disable = False
```

4. 重新启动 **Flash Download Tool**
5. 将生成的明文固件添加到 **Flash Download Tool** 中，并设置相应的 **偏移地址**
6. 使用 **Flash Download Tool** 将 `.bin` 明文固件按照上方*使用 Flash download tool 烧录过程* 进行烧录即可，**加密密钥**将被保存在本地



IDE 烧录 搭建好开发环境后可以在 IDE 中使用 ESP-IDF 插件来进行编译和烧录。

- 在 esp-iot-solution 或者 ESP-IDF 自带的 `examples` 中选择感兴趣的实例项目。

备注： 环境搭建可以参考：VSCode ESP IDF Extension

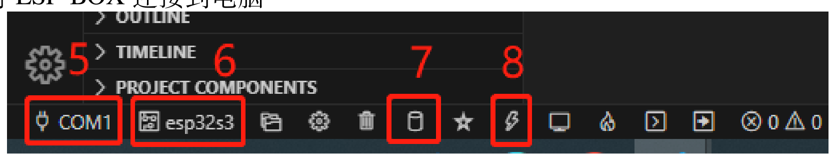
我们可以体验一下 ESP32-S3-BOX 的烧录过程，此项目使用 ESP32-S3-BOX 作为一个 USB 音箱。

以 VSCode 为例：

1. 在 VSCode 中进入想保存项目的路径，在左上角的窗口中选择 **Termina -> New Termianl**
2. 使用 `git clone` 下载本项目的代码

```
git clone --recursive https://github.com/espressif/esp-box.git
```

3. 打开 VSCode，在左上角窗口选择 **file -> open folder** (或者使用快捷键 Ctrl + K Ctrl + O) 进入项目保存的路径 -> **esp-box -> examples -> usb_headset**。点击选择文件夹。
4. 通过数据线将 ESP-BOX 连接到电脑



5. 在左下角选择对应的 `COM 端口`

> **备注：** 如果没有检测到 COM 端口请按住开发板 **Boot** 键，同时按下 **Reset** 键进入下载模式

6. 选择 **Target** 为 `esp32s3`, VSCode 在界面上方选择 **ESP32-S3 chip (via ESP-PROG)**
7. 点击 **ESP-IDF Build Project** 图标，编译成功后会显示：

```
Total sizes:
Used stat D/IRAM:   91594 bytes ( 254262 remain, 26.5% used)
      .data size:   11392 bytes
      .bss  size:    5408 bytes
      .text size:   73767 bytes
   .vectors size:    1027 bytes
Used Flash size :  290627 bytes
      .text     :  216459 bytes
      .rodata   :   73912 bytes
Total image size:  376813 bytes (.bin may be padded larger)
```

8. 点击 **ESP-IDF Flash device** 图标进行烧录，然后在 VSCode 界面上方选择 **UART**。烧录成功后会在终端显示：Flash Done ⚡
9. 按下开发板 **Reset** 按钮后，就可以通过 USB 来用 ESP-BOX 播放声音了

IDF 终端烧录　使用 IDF 终端具有环境稳定，版本切换简单和高级功能齐全等优点。

> **备注：** 环境搭建可以参考：Windows 安装教程。

下面将介绍如何使用 IDF 终端烧录一个示例项目：

1. 打开 `ESP-IDF CMD` 终端窗口，ESP-IDF SDK 编译环境安装成功的界面显示如下：

```
    C:\Espressif\tools\cmake\3.24.0\bin
    C:\Espressif\tools\openocd-esp32\v0.12.0-esp32-20230419\openocd-esp32\bin
    C:\Espressif\tools\ninja\1.10.2\
    C:\Espressif\tools\idf-exe\1.0.3\
    C:\Espressif\tools\ccache\4.8\ccache-4.8-windows-x86_64
    C:\Espressif\tools\dfu-util\0.11\dfu-util-0.11-win64
    C:\Espressif\frameworks\esp-idf-v5.1\tools

Checking if Python packages are up to date...
Constraint file: C:\Espressif\espidf.constraints.v5.1.txt
Requirement files:
 - C:\Espressif\frameworks\esp-idf-v5.1\tools\requirements\requirements.core.txt
Python being checked: C:\Espressif\python_env\idf5.1_py3.11_env\Scripts\python.exe
C:\Espressif\frameworks\esp-idf-v5.1\tools\check_python_dependencies.py:12: DeprecationWarning: pkg_resources is deprecated as an API. See https://s
etuptools.pypa.io/en/latest/pkg_resources.html
  import pkg_resources
Python requirements are satisfied.

Detected installed tools that are not currently used by active ESP-IDF version.
For removing old versions of ccache, cmake, dfu-util, dist, esp-rom-elfs, esp32s2ulp-elf, esp32ulp-elf, idf-driver, idf-exe, idf-python-wheels, ninj
a, openocd-esp32, python_env, riscv32-esp-elf, riscv32-esp-elf-gdb, tools, xtensa-clang, xtensa-esp-elf-gdb, xtensa-esp32-elf, xtensa-esp32s2-elf, x
tensa-esp32s3-elf use command 'python.exe C:\Espressif\frameworks\esp-idf-v5.1\tools\idf_tools.py uninstall'
For free up even more space, remove installation packages of those tools. Use option 'python.exe C:\Espressif\frameworks\esp-idf-v5.1\tools\idf_tool
s.py uninstall --remove-archives'.


Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build


C:\Espressif\frameworks\esp-idf-v5.1>
```

2. 使用 `cd` 指令进入一个工程，例如：

```
C:\Espressif\frameworks\esp-idf-v5.1>cd examples

C:\Espressif\frameworks\esp-idf-v5.1\examples>cd get-started

C:\Espressif\frameworks\esp-idf-v5.1\examples\get-started>cd hello_world
```

3. 切换到正确的的芯片环境，例如：

```
idf.py set-target esp32
```

> **备注：** 将 `esp32` 替换为正确的芯片目标

4. 运行下方指令编译项目：

```
idf.py build
```

5. 运行下方指令烧录项目，将 PORT 替换为开发板的串口名称：

```
idf.py -p PORT flash
```

> **备注：** 使用 `idf.py flash` 将尝试使用可用的串口自动连接

烧录成功的界面显示如下：

```
MAC: a8:03:2a:ec:06:58
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Flash will be erased from 0x00001000 to 0x00007fff...
Flash will be erased from 0x00010000 to 0x0003afff...
Flash will be erased from 0x00008000 to 0x00008fff...
Compressed 26640 bytes to 16668...
Writing at 0x00001000... (50 %)
Writing at 0x0000769f... (100 %)
Wrote 26640 bytes (16668 compressed) at 0x00001000 in 0.7 seconds (effective 290.2 kbit/s)...
Hash of data verified.
Compressed 174688 bytes to 97229...
Writing at 0x00010000... (16 %)
Writing at 0x0001c055... (33 %)
Writing at 0x00021a33... (50 %)
Writing at 0x0002713d... (66 %)
Writing at 0x0002d3fa... (83 %)
Writing at 0x0003517a... (100 %)
Wrote 174688 bytes (97229 compressed) at 0x00010000 in 2.5 seconds (effective 550.6 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.1 seconds (effective 454.2 kbit/s)...
Hash of data verified.

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

进行量产烧录

使用*Flash Download Tool 烧录* 配合 乐鑫官方烧录配件 可以进行自动化部署和批量生产烧录。

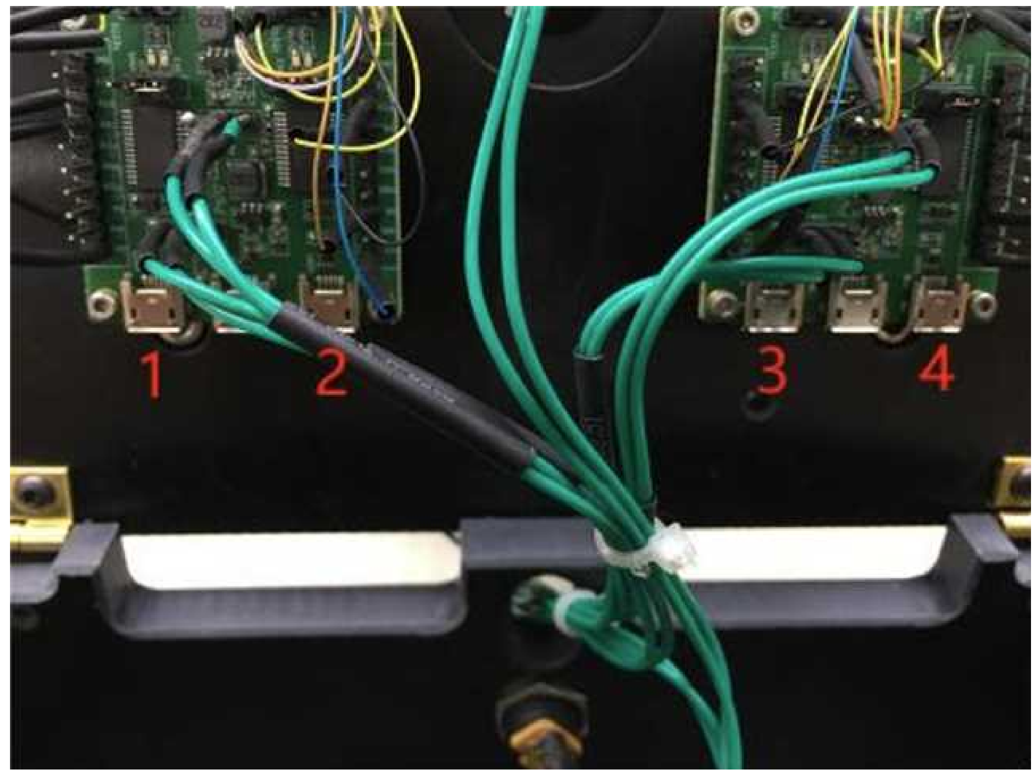
- 烧录底板（一次烧录一个模组）
- 治具（1 次烧录 4 个模组）

使用乐鑫官方治具的烧录过程

1. 继电器供电

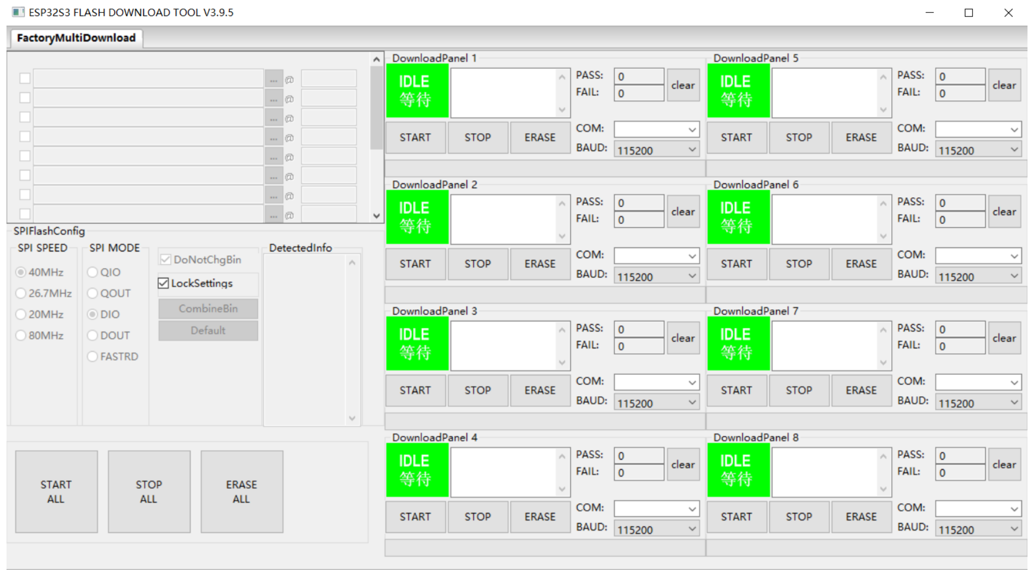
   适配器插接如图所示，给继电器供电；继电器起延时作用，确保治具把手下压稳定后再给模组上电。

2. 治具底板串口通过 HUB 连接电脑
   使用 USB 线将治具底箱下两块底板上的 4 个 mini USB 连接到 HUB，HUB 需要独立供电，确认每个串口给模组供电充足；然后将 HUB 连接到电脑上。（单个串口下载可治具连接电脑）



3. 下载

   选择要下载的固件，设置串口；治具上四个位置放上模组，压下把手，点击 START 开始下载。详细的步骤可以参考：*使用 Flash download tool 烧录过程*



产测资料参考

- 产测指南
- 产测工具

2.1.3 环境搭建

乐鑫官方支持的开发环境有两类：ESP-IDF 和 Arduino IDE。

ESP-IDF

- ESP-IDF 是由乐鑫官方推出的开发框架，支持 Windows、Linux 和 macOS 操作系统。
- 使用终端命令行进行开发，比如运行配置工具，编译，烧录，串口监视。
- 如果开发者更习惯使用图形化的界面，乐鑫官方还提供 **VSCode 和 Eclipse 的插件**可以在 **IDE 的图形化界面**中进行开发。

Arduino IDE

- Arduino IDE 是一个通用的集成开发环境。它针对 ESP 系列的芯片提供了一个专用的核心（core），该核心封装了 ESP-IDF 的 **部分**功能。
- 允许开发者使用 Arduino **更简单的 C++ 编程语言**对 ESP 系列芯片进行开发。
- 不需要深入了解 IDF 的细节。让嵌入式编程 **更加容易**，尤其是对于 **初学者**。

选择合适的开发环境 初学者和寻求简单快速开发的开发者：

- Arduino IDE 是一个绝佳的选择。其简单易用的界面和基本功能可以轻松上手。

有一定编程经验的开发者：

- Visual Studio Code 搭配 ESP-IDF 插件是一个不错的选择。这个组合提供了更友好的界面和自动化配置，使得复杂项目开发变得更加轻松。

熟悉 Eclipse IDE 的高级开发者或专业开发者：

- Eclipse 的 ESP-IDF 插件将提供更多灵活性和强大的功能。
- Espressif-IDE 提供便捷的 ESP 开发环境。

熟悉使用命令行和寻求更丰富功能的开发者：

- ESP-IDF 命令行是很合适的选择。它是乐鑫官方提供的框架，具备最完善的功能，满足开发者复杂嵌入式方案的需求。

ESP-IDF 与乐鑫芯片

下表总结了乐鑫芯片在 ESP-IDF 各版本中的支持状态，其中 支持 代表已支持，预览 代表目前处于预览支持状态。预览支持状态通常有时间限制，而且仅适用于测试版芯片。请确保使用与芯片相匹配的 ESP-IDF 版本。

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

点击标题链接查看最新 IDF 版本支持信息

官方开发环境对比

Arduino IDE

适合人群：

- 初学者、入门级开发者和追求简单快速开发的用户

优点：

- 配置简单，搭建过程相对容易
- 提供了直观的用户界面和简化的工作流程，使得编写和上传代码变得简单易用，方便快速原型开发。
- 提供了大量的库和现成代码，这对于快速原型验证和简单的项目非常方便。
- Arduino IDE 在多个操作系统上操作简单，包括 Windows、Mac 和 Linux。
- Arduino IDE 有大量的文档，教程可供参考。遇到问题更方便解决。

缺点：

- 高级功能有限，不能修改底层代码。在复杂的 ESP-IDF 项目中，使用原始的 ESP-IDF 开发工具链和命令行工具可能更为灵活。
- 调试功能有限。在复杂的 ESP-IDF 项目中，需要其他的调试工具。

如何安装：

- 【文档】基于 Windows 安装 ESP32 Arduino 软件开发环境

ESP-IDF 命令行

适合人群：

- 高级开发者。有丰富嵌入式系统开发经验的开发人员，对嵌入式系统的概念、低级编程和硬件相关知识有一定了解。喜欢或对命令行开发环境感兴趣的开发人员。

优点：

- 提供了基于命令行的开发环境，其中使用 CMD 或 PowerShell 进行开发，有更高的稳定性和灵活性。
- 开发人员可以在命令行开发环境中使用脚本和命令行工具来批量构建、测试和部署应用程序，从而提高开发效率。
- 命令行开发环境通常更加轻量级，对系统资源的消耗较低。这使得它在资源受限的嵌入式系统开发中更加适用。
- 安装简单，官方提供 Windows 一键安装工具，轻松搭建环境。

缺点：

- 命令行开发环境需要开发人员具备一定的命令行和脚本编程的知识。这对新手来说存在较高的学习曲线。
- 缺少图形化界面，可能不如图形界面友好和直观。不方便进行直接的开发。
- 需要在其他 IDE 中编辑代码。

如何安装：

- Windows 一键安装工具
- 【文档】Windows 安装教程
- 【文档】Linux 和 macOS 平台教程
- 【视频教程】Windows 使用一键安装工具快速搭建 ESP-IDF 开发环境

Visual Studio Code Extension

适合人群：

- 有一定嵌入式开发经验的开发人员，已经使用或倾向于使用 VSCode 作为主要开发环境的开发人员。

优点：

- 将 ESP-IDF 框架与 VSCode 集成在一起，为开发人员提供了在同一界面下开发、构建和调试 ESP-IDF 项目的便利性。
- 提供了简化配置的功能，开发人员可以通过插件中的设置界面轻松配置项目参数、编译选项和调试设置等。配置 ESP-IDF 环境和项目变得更加简单和直观。
- VSCode 是一个相对轻量级的编辑器，启动速度快，并且对系统资源消耗相对较低。

缺点：

- 对于不熟悉 VSCode 或插件开发的开发人员来说，可能需要额外的学习和适应。
- 可能会受到插件本身更新和与新版本 ESP-IDF 兼容性的限制。开发人员可能需要关注插件的更新和相关文档。
- 许多功能依赖插件。

如何安装：

- 【英文文档】Visual Studio Code Extension 安装教程
- 【视频教程】使用 VS Code 快速搭建 ESP-IDF 开发环境 (Windows、Linux、MacOS)

Espressif-IDE 或 Eclipse Plugin

备注： Espressif-IDE 附带 IDF Eclipse Plugin、基本 Eclipse CDT 插件以及 Eclipse 平台的其他第三方插件，以支持构建 ESP-IDF 应用程序。

适合人群：

- 高级开发者和专业开发者。有嵌入式系统开发经验并且已经使用或倾向于使用 Eclipse 的的开发人员。

优点：

- 将 ESP-IDF 框架与 Eclipse 集成在一起，为开发人员提供了在同一界面下开发、构建和调试 ESP-IDF 项目的便利性。
- 对复杂的大型项目支持更加出色。
- Eclipse IDE 有庞大的插件生态系统，开发人员可以根据项目需求安装和使用其他有用的插件，提高开发效率。
- 拥有强大的图形化调试功能。

缺点：

- 相对于上述轻量级的开发环境或纯命令行环境，Eclipse IDE 本身可能会消耗较多的系统资源，尤其在较低性能的计算机上可能会感觉较慢。
- Eclipse 在配置环境方面相对复杂，对新手不太友好。
- Eclipse 有相当大的用户社区，但相对于其他环境来说较小。

如何安装：

- 【文档】Espressif-IDE
- 【文档】Eclipse Plugin 教程
- 【英文文档】Eclipse Plugin Windows 安装教程
- 【视频教程】Eclipse Plugin 教程（英语）

备注： 环境搭建问题可以参考 csdn 博客

选择合适的开发环境取决于你的经验水平、项目复杂性和个人偏好。初学者可以从 Arduino IDE 开始，然后逐渐转向带插件的 VSCode 环境。有经验的开发者可以直接使用 ESP-IDF 或在 Eclipse 中使用 ESP-IDF 插件进行更复杂的项目开发。

其他开发环境

1. MicroPython

   `MicroPython` 是一种基于 `Python` 的高度精简版解释器，专为嵌入式系统设计而成。它允许开发者使用熟悉的 Python 语言进行嵌入式开发，使得编程变得更加简单和高效。MicroPython 可以运行在多种硬件平台上，包括 ESP 系列的芯片。

2. PlatformIO IDE

`PlatformIO` 是一个跨平台的开源生态系统，它支持多种硬件平台和开发板，并提供了一个统一的开发环境，允许开发者使用不同的硬件和开发板进行嵌入式开发。PlatformIO 集成了多个开发工具，包括编译器、调试器和上传工具。

3. CLion

`CLion` 是一款由 JetBrains 公司开发的 IDE，专为 C 和 C++ 编程语言开发者设计。它提供了智能代码补全、语法高亮、代码导航、强大的重构功能和调试工具，以提高开发者的效率。CLion 还集成了版本控制系统，如 Git，使团队协作更加便捷。

注意： MicroPython 、PlatformIO 和 CLion 不是乐鑫直接开发的平台，故乐鑫官方不会维护与这些开发环境相关的问题。因此在使用过程中可能遇到一些困难，开发者需要去相关社区和论坛来获取支持和解决问题。

第三方参考资料

Arduino Windows

- 【视频】Arduino 开发 ESP32 环境搭建指南

Arduino Ubuntu

- 【视频】ESP32 开发环境搭建（Arduino）

VSCode

- 【视频】ESP32-IDF 环境搭建之 vscode 环境
- 【视频】vscode+espidf 开发环境搭建 (实现单步调试)

Eclipse

- 【英文视频】Install ESP32 + Eclipse IDE 2020 (English)

2.1.4 项目开发入门

当开发一个 ESP 项目时，通常的流程是先根据项目的内容去了解对应的 ESP 的示例和 SDK，挑选和目标项目最契合的作为模板，然后在其基础上进行修改并创建自己的项目。

本文将总结一些通用嵌入式项目在乐鑫环境下开发所需要的知识点:

- *通用知识*
- *了解 ESP 系列芯片和 ESP-IDF 框架*
- *项目开发知识点总结*
- *学习方式*

通用知识

在开发 ESP 项目之前需要一些关键的通用知识和技能，以确保开发能够顺利进行:

- Git:

  Git 是一个开源的分布式版本控制系统，用于跟踪代码的更改和协作开发。在项目中使用 Git 可以帮助你有效地管理代码，进行版本控制，并与团队成员协同工作。学习如何使用 Git 进行代码提交、分支管理、合并等操作是开发 ESP 项目的基本要求。这里推荐开发者需掌握一些 Git 的常用指令

- FreeRTOS：
  FreeRTOS 是一种开源的实时操作系统，广泛应用于嵌入式系统和微控制器项目中。ESP 系列芯片都支持 FreeRTOS，并且常常使用它来实现多任务和实时调度。了解 FreeRTOS 的任务调度、消息队列、信号量等概念是开发 ESP 项目的必备知识。
  – 开发者可以学习 FreeRTOS 的官方入门指南。
  – 在 ESP-IDF 编程指南的 FreeRTOS 章节 中学习如何在 ESP-IDF 环境下使用 FreeRTOS。
- Linux（可选）：
  ESP 项目的开发很多情况下在 Linux 操作系统上进行，因为 Linux 提供了丰富的工具和命令行环境，适合嵌入式开发。开发者应该了解 Linux 的基本命令和使用方式，例如文件操作、目录管理、进程管理等，以便在开发过程中进行调试和配置。推荐开发者需掌握一些 Linux 文件与目录管理常用指令

```
- ls： 列出当前目录的文件和文件夹。
- cd <directory>： 切换到指定目录。
- pwd： 显示当前工作目录的路径。
- cp <source> <destination>： 复制文件或目录。
- rm <file_name>： 删除文件。
- mkdir <directory>： 创建新目录。
- rmdir <directory>： 删除空目录。
- cat <file_name>： 显示文件内容。
```

了解 ESP 系列芯片和 ESP-IDF 框架

- ESP 芯片的特点和功能，参考：硬件选型
- ESP-IDF 的版本简介和管理
- ESP-IDF 框架的组成和架构
- 开发环境的设置和配置
- ESP-IDF 工具的使用
- ESP-IDF API 的使用 和 ESP-IDF API 用法的指南
- ESP 芯片的安全功能

推荐网站：

- ESP-IDF 快速入门
- ESP-IDF 编程指南

项目开发知识点总结

C 语言

1. 数据类型
   - **基本数据类型**
     – 整数、字符、浮点
   - **标准库数据类型**
     – 布尔型（bool）、字符串（string）
   - 指针类型
   - **复合数据类型**
     – 数组、结构体（struct）、枚举（enum）
   - 自定义数据类型
   - **数据类型限定符**
     – 常量限定符（const）、易变限定符（volatile）
2. 函数、指针和内存管理
3. 工程项目编译、链接和执行过程
4. 算法
   - **数据结构**
     – 链表

  - 栈和队列
  - 树和图
- **排序算法**
  - 冒泡排序（Bubble Sort）
  - 选择排序（Selection Sort）
  - 插入排序（Insertion Sort）
  - 快速排序（Quick Sort）
  - 归并排序（Merge Sort）
- **查找算法**
  - 线性查找（Linear Search）
  - 二分查找（Binary Search）
  - 哈希查找（Hashing）
- **递归算法**
  - 分治
  - 回溯
- 动态规划
- **编码和解码算法**
  - 处理数据的压缩、加密、编码和解码
- **信号处理算法**
  - 音频、图像的处理
- **算法优化**
  - 时间复杂度
  - 空间复杂度

通信协议

1. 掌握各种通信协议，如 **UART**、**SPI**、**I2C**、**CAN** 等，以便与外部设备进行通信
2. 学习网络协议栈，如 **TCP/UDP**。以及上层应用协议，如 **HTTP**、**MQTT** 等

工程构建和管理

1. 构建系统
2. 分区表
3. 组件管理和使用

应用编程

1. GPIO（General Purpose Input/Output），通用输入/输出
   - GPIO 初始化、模式、读取、写入
   - **GPIO 配置选项**
     - 根据 **技术手册**和 **引脚图**来确定每个引脚的功能
   - GPIO 中断
   - Strapping 管脚
2. JTAG（Joint Test Action Group）调试
3. 内存管理
   - 动态内存分配和释放，如 `malloc` 和 `free`
   - 内存布局和堆栈管理
4. 中断处理
   - 理解和处理硬件中断
   - 实现中断服务程序（ISR）来响应外部事件
5. 时钟和定时器
   - 使用定时器和时钟源来实现时间控制和定时任务
   - 处理延时和定时操作
6. 异常处理
   - 处理硬件和软件异常
7. 低功耗模式设计
   - 实施功耗优化策略，用来延长电池使用寿命或减少能源消耗

8. 进程
   - 进程的创建和终止
   - 进程的管理和调度
   - 进程的资源分配和使用
   - **进程间的通信方式**
     - **管道（Pipe）、命名管道（Named Pipe，FIFO）、消息队列（Message Queue）、信号（Signal）、共享内存（Shared Memory）、信号量（Semaphore）、套接字（Socket）**
9. 线程
   - 线程的创建和销毁
   - 线程的调度和同步
   - 多线程编程的并发控制
   - **线程间通信方式**
     - **互斥锁（Mutex）、条件变量（Condition Variable）、信号量（Semaphore）、屏障（Barrier）、消息队列（Message Queue）、共享内存（Shared Memory）、自旋锁（Spin Lock）**
10. 驱动开发

- 驱动程序可以针对各种硬件设备编写，包括但不限于 **传感器、执行器、存储设备（如闪存和 SD 卡）、通信接口（如 UART、SPI、I2C）、显示屏、网络接口卡（NIC）** 等

学习方式

- 在线资源和文档：
  - 利用 ESP-IDF 官方文档、教程和示例代码，深入了解框架和 API 的使用方法。
  - 在 ESP IoT Solution 库中找到基于 ESP-IDF 的解决方案、应用实例、组件和驱动等内容。多数文档均提供中英文版本。
- 在线课程和视频教程：
  - 通过参加 ESP 芯片和 ESP-IDF 开发的 在线课程和视频教程，学习相关知识和实践技巧。
- 实验和项目：
  - 通过实际使用 ESP 开发板 进行实验和项目开发，加深对硬件和软件的理解。刚入门可以多参考其他已经做出来的项目或者 乐鑫发布的解决方案 来学习。
- 社区和论坛：
  - 加入 ESP32 官方论坛 ，CSDN 论坛，或者其他开发者社区。与其他开发者交流经验、寻求帮助并分享项目和解决方案。
  - 通过 GitHub 的 Issues 版块提交 bug 或功能请求。在提交新 Issues 之前，请先查看现有的 Issues。
- 参考书籍：
  - 阅读与嵌入式系统、C 语言和 ESP 芯片开发相关的书籍，扩展知识广度和深度。参考：ESP32 书籍列表。

持续实践和项目开发是学习过程中最重要的部分。开发者可以通过不断的练习和实际应用，逐渐掌握 ESP 芯片的嵌入式项目开发技能。

2.2 进阶开发

这一部分适合那些希望深入了解和掌握更高级编程技巧的开发者。我们将介绍一些更复杂的开发技巧，包括组件管理器，代码调试等，这个部分都会提供深入的指导。

2.2.1 组件管理和使用

想象你正在开发一个复杂的应用程序，它需要涉及多个功能模块，例如网络通信、传感器读取、屏幕驱动。如果你将所有功能都写在一个大文件中，代码会变得冗长且难以维护。这时，组件就起到了模块化的作用。

每个组件专注于一个特定的功能领域，这样，每个组件都可以单独开发、测试和维护。通过将功能拆分为组件，你可以在多个项目中重复使用这些功能，无需重复编写代码。这样既节省了开发时间，又提高了开发效率。同时项目的结构也会变得更加的清晰。

- 寻找组件
- 使用组件管理器添加组件
- 使用本地的组件
- 从 *Git* 仓库拉取组件
- 更新组件
- 删除组件
- 开发组件（进阶）
- 发布组件

IDF 组件

ESP-IDF 组件是一个可重用的代码包，组件在构建时会被编译成静态库，可以被其他组件或者应用程序链接。这样就可以在多个项目中使用。

一个完整的组件一般包含：

- 源代码
- 头文件
- CMakeLists.txt
  - 定义源代码和头文件
  - 定义依赖关系
  - 注册组件
  - 配置可选特性
  - CMake 构建描述文件，用于描述组件的构建配置和依赖关系，用于指示编译器如何编译、链接和构建组件。
- idf_component.yml
  - 组件管理器描述文件，它列出了需要引用的其他组件及其版本信息。这样，在构建项目时，ESP-IDF 的组件管理器会根据这些描述，自动下载并集成所需的组件，以确保项目的依赖关系得到满足。

组件依赖

编译各个组件时，ESP-IDF 系统会递归评估其依赖项。这意味着每个组件都需要声明它所依赖的组件，即“requires”。

依赖是指一个软件项目所依赖的其他软件库、模块或组件，这些依赖项是构建和运行项目所必需的。软件项目通常会使用其他已经存在的代码或库来实现特定功能或提供某些服务，而不是从头开始编写所有的代码。这些外部的代码或库就是项目的依赖项。

依赖的作用是为了促进代码的重用、降低开发成本、加快开发速度和提高软件质量。使用现有的依赖项可以避免重复造轮子

常见的依赖项包括：

- 第三方库
- 框架和组件
- 运行时环境

寻找组件

1. IDF 中包含了一些基础功能组件，可以在 components 目录下找到
2. idf-extra-components 包含了一些 IDF 补充组件
3. esp-iot-solution 提供了一些外设驱动组件
4. esp-registry 中可以搜索到乐鑫维护的组件和第三方上传的组件
5. 寻找第三方组件库

组件管理器（Component Manager）

组件管理器 是指用于管理组件的工具或系统。在 IDF 中，组件管理器提供了一组命令 用于添加、移除、下载、管理组件，以便在项目中重用已有的功能模块。

使用组件管理器添加组件

1. 在 esp-registry 找到工程所需的组件
2. 使用终端进入工程目录的根文件夹，执行 `idf.py add-dependency "[命名空间]/<组件名>[版本号]"`
   - 这条指令将自动添加一个条目到 `main/idf_component.yml`，若文件不存在将自动创建
   - 命名空间：可选，默认从 espressif (大小写不敏感)
   - 组件名：必须，组件的名称，例如 usb_stream (大小写不敏感)
   - 版本号：可选，默认为 * 代表任意版本。首次添加组件时，会自动下载最新版本
3. 运行 `idf.py build`

备注:

- 无需对工程 CMakeLists.txt 进行额外修改，可以直接在工程源代码中 include 组件的公共头文件、编译并链接组件的静态库
- 用户无法直接修改 managed_components 中的组件，因为组件管理器会自动检测组件 Hash 值、并恢复更改。
- 除了使用指令 idf.py add-dependency ，用户也可以直接修改 idf_component.yml 文件，手动修改组件条目。

使用本地的组件

在组件管理器描述文件 `idf_component.yml` 中，按照下方示例，添加本地组件地址

```yaml
dependencies:
    idf: ">=4.4.1"
    cmake_utilities: "0.*"
    iot_usbh_modem:
        version: "0.1.*"
        override_path: "../../../../../components/usb/iot_usbh_modem"
```

从 Git 仓库拉取组件

在组件管理器描述文件 `idf_component.yml` 中，按照下方示例，添加组件在 Github 的地址

```yaml
dependencies:
    esp-gsl:
        git: https://github.com/leeebo/esp-gsl.git
        version: "*"
    button:
        git: https://github.com/espressif/esp-iot-solution.git
        path: components/button
        version: "*"
```

更新组件

1. 将组件管理器描述文件 `idf_component.yml` 中的组件版本号修改为要更新到的目标版本
2. 删除工程文件中的 `managed_components` 和 `dependencies.lock`
3. 执行 `idf.py build` 或手动执行 `idf.py reconfigure`

删除组件

1. 删除 `idf_component.yml` 中的组件条目
2. 删除 `build`, `managed_components` 和 `dependencies.lock`
3. 执行 `idf.py build`

开发组件（进阶）

1. 创建一个组件目录，和 ESP-IDF 组件要求一致，编写对应的 `CMakeLists.txt` 文件
2. 添加 `idf_component.yml` 文件，添加组件信息，IDF 版本要求，指定依赖的其他组件
3. 添加 `license.txt` 文件，添加组件的许可证信息。许可证信息用于明确告知其他开发者或用户，在使用该组件时需要遵守的条款和条件。这样可以确保组件的使用和分发是合法的，并明确组件作者对其作品的版权和授权情况。许可证信息一般包含在一个名为 license.txt 或 LICENSE 的文件中，其中包含了以下内容：
   - 许可证类型：明确指定组件的许可证类型，例如 MIT 许可证、Apache 许可证、GPL 许可证等。不同类型的许可证有不同的条款和条件，开发者和用户需要遵守相应的许可证规定。
   - 版权信息：注明组件的版权归属权，即组件的作者或版权持有人。版权信息通常包括作者的姓名、组织或公司名称，以及版权年份。
   - 免责声明：可能包含免责声明，声明组件的作者或版权持有人对组件使用可能引起的任何损失或责任不承担责任。
   - 权限和限制：明确指定在何种条件下可以使用和分发该组件，以及禁止的行为或限制。
   - 开源许可：如果组件是开源软件，许可证信息应该包含相应的开源许可条款，如开源代码的共享、修改、分发等规则。
4. 添加 `README.md` 文件，添加组件的简介、简单的使用说明
5. 添加 `CHANGELOG.md` 文件，添加组件的版本更新记录
   - 用于记录组件版本更新记录的文档。它通常包含了组件每个版本的变更内容、改进、修复和新增功能等信息。开发者和用户可以通过查阅 CHANGELOG.md 文件了解组件的版本更新历史，以便更好地了解组件的发展和使用。
6. 添加 `test_apps` 目录，添加组件的测试用例
   - 组件的自检工具，可以确保组件的质量和稳定性，对于组件的开发和维护非常重要
   - 里面包含了应用程序都针对组件的不同方面或功能进行测试
7. 添加 `examples` 目录
   - 为组件提供示例代码，以展示组件的基本用法和功能
   - 或在 idf_component.yml 中指定 examples 路径，添加组件的示例代码

发布组件

发布在 ESP-Registry ESP-Registry 是乐鑫提供的中央仓库，为开发者提供了一个便捷的方式来发现和下载用于工程项目中的组件

1. 在 ESP-Registry 注册账号 - 通过 Github 账号授权就可以在 ESP-Registry 上完成注册
2. 注册账号后，点击右上角用户名，选择 `Tokens`
3. 在这个界面点击 `Create` 就可以创建一个 `Token`
4. 把刚才创建的 `Token` 注册到环境变量 `IDF_COMPONENT_API_TOKEN`
5. 使用 `idf.py upload-component` 命令上传组件到 ESP 组件注册库 - `idf.py upload-component --namespace [YOUR_NAMESPACE] --name test_cmp` - `YOUR_NAMESPACE`：esp-registry 用户名，在 profile 中可以看到 - `test_cmp`：上传组件的名称
6. 上传完成后，组件就可以在 ESP-Registry 上查看了。

发布在 Github 仓库 如果只想发布到 Github 仓库，可以按照组件开发规范，直接将组件发布到自己的 Github 仓库。其他开发者可以用前面*从 Git 仓库拉取组件* 的方式来拉取这个组件。

相关链接

- 组件管理器 Github

- ESP-Registry 网站
- ESP-Registry 文档
- upload-components-ci-action

2.2.2 进阶代码调试

这一部分介绍了 ESP 芯片常见的软件异常 crash ，以及出现类似问题时使用的一些常用的进阶代码调试方法。

综述：常见的 Panic & Exception 介绍

在进行代码调试前，了解常见的 Panic & Exception 也很有必要，常见的情况包括以下几种：

- *Watchdog Interrupt*
- *Brownout Interrupt*
- *Assert / Abort*
- *Stack Overflow*
- *Cache Access Error*
- *Invalid Memory / Instruction Address*

Watchdog Interrupt 此部分往往常见为两种情况：中断看门狗和任务看门狗，详细的文档请参考 看门狗，以下是这两个看门狗的简要说明：

中断看门狗 中断看门狗定时器的目的是，确保中断服务例程 (ISR) 运行不会受到长时间阻塞（即 IWDT 超时）。阻塞 ISR 及时运行会增加 ISR 延迟，也会阻止任务切换（因为任务切换是从 ISR 执行的）。阻止 ISR 运行的事项包括：

- 禁用中断
- 临界区（也会禁用中断）
- 其他相同或更高优先级的 ISR，在完成前会阻止相同或较低优先级的 ISR

当 IWDT 超时后，默认操作是调用紧急处理程序 (Panic Handler)，并显示出错原因（Interrupt wdt timeout on CPU0 或 Interrupt wdt timeout on CPU1，视情况而定）。

此时需要排查以上三点内容，尤其是要注意是否在 ISR 里放置了太多代码，导致 ISR 的时间过长导致中断看门狗触发。

任务看门狗 任务看门狗定时器用于监控 FreeRTOS 里的任务在规定的时间内是否得到调度。如果默认存在的优先级最低的 IDLE 任务没有在规定的时间内喂狗（即重置看门狗定时器），就会触发看门狗中断。这通常表示高优先级的任务因为无限循环且无延时的调用某个 API 等原因一直占用着 CPU。

此问题的调试方法简要举例如下，在 *app_main* 里刻意编写一个不带有延时的无限循环来打印 log ：

```
void app_main(void)
{
    while (1) {
        printf("Hello world!-----------------------------------------------------\n
↪");
    }
}
```

在 ESP 上电运行 CONFIG_ESP_TASK_WDT_TIMEOUT_S 秒后，可以看到以下 log ：

```
E (10303) task_wdt: Task watchdog got triggered. The following tasks/users did not
↪reset the watchdog in time:
E (10303) task_wdt:  - IDLE (CPU 0)
E (10303) task_wdt: Tasks currently running:
E (10303) task_wdt: CPU 0: main
E (10303) task_wdt: CPU 1: IDLE
E (10303) task_wdt: Print CPU 0 (current core) backtrace
```

上述 log 可简析为：

- *E (10303) task_wdt: - IDLE (CPU 0)* –> CPU0 上的 IDLE0 没有及时喂狗
- *E (10303) task_wdt: CPU 0: main* –> CPU0 上此时运行的任务是 main 任务（main 任务对应的即是 app_main 函数）
- *E (10303) task_wdt: CPU 1: IDLE* –> CPU1 上此时运行的任务是 IDLE1 任务

此时则说明 IDLE0 任务因为在 CONFIG_ESP_TASK_WDT_TIMEOUT_S 秒内没有及时喂狗而触发了看门狗复位，然后通过 CPU0 上运行的任务为 main 任务可以推断应该是 main 任务一直在占用 CPU 导致任务看门狗复位，应重点排查此任务，添加适当的延时以让其他低优先级任务有得到调度的机会。

Brownout Interrupt 此部分为掉电中断，ESP 芯片内部集成掉电检测电路，并且会默认启用。当掉电检测器被触发时，会打印如下信息:

```
Brownout detector was triggered
```

芯片会在该打印信息结束后复位。此时应检查硬件供电电压是否满足设定的阈值。更多信息请参考 掉电 文档。

备注： 在电池供电场景时，比如 2xAA 电池供电，电压为 3.1 V，此时在 Wi-Fi 建立连接等场景下会因较大的瞬时电流导致电压短时下降，触发掉电检测导致芯片重启。

此时建议使用峰值电流更大的稳压芯片。或更换能提供大电流的电池，或尝试增加电源的电容。

Assert / Abort 此部分为触发断言或中止，断言通常用于检查程序中的假设是否为真。如果某个断言失败（即假设不成立），就会触发中断。中断后，程序通常会中止（abort），并在日志中记录错误信息，以帮助开发人员调试。

因此可以查看 log 中此时哪个 API 触发了断言或中止，从这个 API 入手来调试代码，例如：

```
// Function to check if input is equal to 1
int check_input(int input) {
    if (input == 1) {
        return 1; // Return 1 if input is equal to 1
    } else {
        return 0; // Return 0 if input is not equal to 1
    }
}

void app_main(void)
{
    // Test case for input not equal to 1
    int input1 = 2;
    assert(check_input(input1) == 1); // Assert that the function returns 1 for
↪input 1
}
```

对应的异常 log 为：

```
assert failed: app_main hello_world_main.c:26 (check_input(input1) == 1)
```

可以看出原因是 *assert(check_input(input1) == 1)* 因为条件得不到满足而触发断言，此时可以重点排查此断言对应的代码。

备注： 由于断言可以在 ESP-IDF menuconfig 中的 CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL 选项来禁用，故不建议在断言中做计算或进行函数调用。

Stack Overflow 此部分为栈溢出，往往实际使用的任务堆栈超出了预分配的任务堆栈时，会产生此报错，错误示例代码如下：

```
static void test_task(void *pvParameters)
{
    uint32_t large_array[4096] = {0};
    while (1) {
        // This task obstruct a setting tx_done_sem semaphore in the UART␣
↪interrupt.
        // It leads to waiting the ticks_to_wait time in uart_wait_tx_done()␣
↪function.
        vTaskDelay(200 / portTICK_PERIOD_MS);
    }
    vTaskDelete(NULL);
}


void app_main(void)
{
    xTaskCreate(test_task, "test_task", 1024, NULL,  5, NULL);
}
```

对应的错误 log 如下：

```
***ERROR*** A stack overflow in task test_task has been detected.
```

从 log 里可以看到是 test_task 任务发生了栈溢出，检查代码可以得知在 *xTaskCreate* 时仅仅分配了 1024 字节的任务堆栈，但是任务里却在使用一个大小为 4096 字节的数组，因此造成了堆栈溢出，此时需要减少任务函数里的数组大小，或者增加 *xTaskCreate* 分配的任务堆栈。

Cache Access Error 此部分为缓存访问错误，目前可以先参考 Cache disabled but cached memory region accessed 文档，在*利用 Guru Meditation 错误打印定位问题* 中会进行对此错误进行更详细的讲解。

Invalid Memory / Instruction Address 当程序试图访问不存在的内存地址或者执行无效的指令时，会触发此类异常。这通常是由于指针错误或者内存损坏引起的。出现此类错误时，可以参考*使用 Backtrace & Coredump 定位问题* 和*定位内存问题（踩内存、内存泄漏等）* 章节来进一步代码调试。

综述：Panic Handler 与常用代码调试方法介绍

Panic Handler（紧急处理程序） 已在 ESP-IDF 编程指南 里详细描述，在此不做赘述。

常用的代码调试方法如下。

日志打印 / App Trace ESP-IDF 中提供了便捷易用的日志打印库 ESP Logging 和 App trace（应用层跟踪库）。

Backtrace / Core Dump 此部分的详细指导请参考*使用 Backtrace & Coredump 定位问题*。

GDB Stub through UART 请参考 GDB Stub，可简要总结为如下。

GDB Stub Postmortem

1. 使能 *CONFIG_ESP_SYSTEM_PANIC_GDBSTUB* 在错误发生时，通过 UART esp_monitor 会自动进入 GDB
2. 使用 ELF + Core dump 记录，通过 *idf.py coredump-debug* 进入 GDB

GDB 常用指令：

1. *backtrace* 追溯调用栈（包含传入参数）
2. *list* 打印代码位置
3. *info threads* 打印所有 Task 信息
4. *info locals* 打印当前 Task 局部变量
5. *print <var>* 打印当前 Task 局部变量/全局变量
6. *thread <id>* 切换 Task

也可以进一步参考 GDB Cheat Sheet。

GDB Stub Debugging

1. 使能 *CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME*，在运行时可通过 UART esp_monitor 自动进入 GDB（*Ctrl'+'C*）

GDB 常用指令：

1. *x/1xw <addr>* 打印对应地址 32 bit 内存/寄存器
2. *watch <var>* 监视变量，当变量值发生变化时，GDB 会自动停下来
3. *break <where>* 设置断点，可以是函数名、行号、地址
4. *continue* 继续运行
5. *step* 单步运行，进入函数
6. *next* 单步运行，不进入函数
7. *print <var>* 打印变量值
8. *set <var> = <value>* 修改变量值

也可以进一步参考 GDB Cheat Sheet。

OpenOCD & GDB 请参考 JTAG Debugging。

GPIO tracing GPIO tracing（GPIO 跟踪）往往通过翻转 IO 标识事件，比如在代码里自行实现出现某个事件时进行 GPIO 翻转，这样就能通过查询 GPIO 的翻转状态来判断事件是否触发。

SystemView through UART/JTAG 此部分的详细指导请参考*使用 SystemView 进行系统分析和调优*。

利用 Guru Meditation 错误打印定位问题

目前可先参考 ESP-IDF 编程指南里的 Guru Meditation 错误 来了解常见的错误及对应的原因。此文档包含常见错误的进一步说明及展示一些可行的分析思路。

备注： 以下主要以 ESP32-C3 所代表的 RISC-V 架构出现的错误进行分析，ESP32 等 Xtensa 架构对应的错误在命名上会有区别，但错误本质相同，某种架构的一些专属错误会单独分析。

Guru Meditation 错误分为两种类型：ARCH 异常和 SOC 异常。

ARCH 常见异常主要包含以下类型：

- Load access fault
- Store Access Fault
- Instruction Access Fault
- IllegalInstruction

SOC 常见异常主要包含以下类型：

- Interrupt wdt timeout on CPU0/CPU1
- Cache disable but cached memory region accessed

很多时候，Guru Meditation 错误并不能很快的确定原因，因为这些 Guru Meditation crash 只是表象，实际问题会和 FreeRTOS，memory, flash, interrupt 等模块相关，需要具体问题具体分析。

Load access fault（LoadProhibited）

测试用例 使用加载指令（如 lw）加载无效的内存就会发生此类异常。触发此类 crash 的示例代码如下：

```
uint8_t *buff = NULL;
load_fault = buff[1];
```

异常现象

- RISC-V

```
Guru Meditation Error: Core  0 panic'ed (Load access fault). Exception was␣
↪unhandled.

Core  0 register dump:
Stack dump detected
MEPC    : 0x4200696a  RA      : 0x4200696a  SP      : 0x3fc8f7c0  GP      :␣
↪0x3fc8b400
TP      : 0x3fc87d74  T0      : 0x4005890e  T1      : 0x3fc8f41c  T2      :␣
↪0x00000000
S0/FP   : 0x3c023000  S1      : 0x00000000  A0      : 0x0000000f  A1      :␣
↪0x3fc8f3f8
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x3fc8c000  A5      :␣
↪0x00000000
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :␣
↪0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000005  MTVAL   :␣
↪0x00000001
```

- XTENSA

```
Guru Meditation Error: Core  0 panic'ed (LoadProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x42007c2f  PS      : 0x00060a30  A0      : 0x82007c42  A1      :␣
↪0x3fc97d70
A2      : 0x42011934  A3      : 0x3c023f48  A4      : 0x3c023fc8  A5      :␣
↪0x3fc97d90
A6      : 0x3fc97d70  A7      : 0x0000000c  A8      : 0x00000000  A9      :␣
↪0x3fc97d20
A10     : 0x0000000f  A11     : 0x3fc97fac  A12     : 0x3fc97d70  A13     :␣
↪0x0000000c
```

（下页继续）

(续上页)

```
A14      : 0x3fc941e4  A15      : 0x00000000  SAR      : 0x00000004  EXCCAUSE:␣
↪0x0000001c

Backtrace: 0x42007c2c:0x3fc97d70 0x42007c3f:0x3fc97d90 0x42019247:0x3fc97db0␣
↪0x40379e71:0x3fc97de0
```

定位方法 如果此时进一步分析汇编代码（步骤可参考*附录 1 - ESP 反汇编对应的指令*），可以发现能对应到以下指令：

```
lw a1, 1(zero)              // RSIC-V
l8ui    a11, a8, 1          // XTENSA
```

在触发此类异常时，`MTVAL/EXCVADDR` 会自动更新为与异常有关的数据，从 RISC-V 异常信息中可以看到 `MTVAL : 0x00000001`。

一般来说 MTVAL 为 NULL(0x00000000) 代表的是从 NULL 地址取值，而接近 NULL 代表的是尝试从 NULL 地址访问某个数组或者结构体的成员。

- 对于 RSIC-V 架构来说，PC 和 RA 寄存器包含了异常的函数指针，从 elf 中找出对应的函数排查即可
- 对于 Xtensa 架构来说，异常时会打印 backtrace，从 elf 中也可以找出对应的函数

实际案例 下图是一个典型的 ESP32-C3 Load access fault 问题。

从上图可以看出，出问题的地方在于 `lw a5, 20(s0)` ，`s0` 是 0x0, 这对应了一个 NULL 指针，此时可以继续分析前一句汇编 `lw s0, 24(sp)`，对应的代码段为 `pxTimer = xMessage.u.xTimerParameters.pxTimer;`，此时问题可变成 `pxTimer` 为什么为 NULL。

通过添加如下调试日志来进一步定位问题：

```
if (pxTimer == NULL) {
    ets_printf("xMessageID: %d\n", xMessage.xMessageID);
}
```

最终发现问题在于并未成功创建 timer ，此时 timer 对应的指针为 NULL，此时再执行后续 timer 操作就会因空指针导致此 `Load access fault (LoadProhibited)` 错误。

```
    }

    // If there's no callback, we just need a delay.
    if(pTimerCallback == NULL) {
        vTaskDelay(period);
    } else {
        // If it doesn't exist, create it.
        if(g_timer != NULL) {
            g_timer = xTimerCreate("NFCTimer", 10, pdFALSE, pTimerCallback, timer_expiry_cb);
        }

        // Set the period, then start/restart it.
        xTimerChangePeriod(g_timer, period, BLOCK_TIME);
        xTimerReset(g_timer, BLOCK_TIME);
    }
```

```
36      }
37
38      // If there's no callback, we just need a delay.
39      if(pTimerCallback == NULL) {
40          vTaskDelay(period);
41      } else {
42          // If it doesn't exist, create it.
43+         if(g_timer == NULL) {
44              g_timer = xTimerCreate("NFCTimer", 10, pdFALSE, pTimerCallback, timer_expiry_cb);
45          }
46
47          // Set the period, then start/restart it.
48          xTimerChangePeriod(g_timer, period, BLOCK_TIME);
49          xTimerReset(g_timer, BLOCK_TIME);
50      }
```

注意事项 并不是所有的 `Load access fault (LoadProhibited)` 问题都可以简单通过上述方法确定出问题的位置，如下图所示：

```
ELF file SHA256: a25f43e6908a36ed

Rebooting...
Guru Meditation Error: Core  1 panic'ed (LoadProhibited). Exception was unhandled.

Core  1 register dump:
PC      : 0x400358e6  PS      : 0x00060533  A0      : 0x80376ae6  A1      : 0x3fca2590
0x400358e6: rom_i2c_writeReg_Mask in ROM

A2      : 0x0000006d  A3      : 0x00000001  A4      : 0x00000004  A5      : 0x00000004
A6      : 0x00000000  A7      : 0x0000001d  A8      : 0x3fcef3d4  A9      : 0x00000000
A10     : 0x00060523  A11     : 0x00000004  A12     : 0x00060524  A13     : 0x00000001
A14     : 0x00000001  A15     : 0x3fc98824  SAR     : 0x0000000a  EXCCAUSE: 0x0000001c
EXCVADDR: 0x00000190  LBEG    : 0x40056f5c  LEND    : 0x40056f72  LCOUNT  : 0x00000000
0x40056f5c: memcpy in ROM

0x40056f72: memcpy in ROM



Backtrace: 0x400358e3:0x3fca2590 |<-CORRUPTED
0x400358e3: rom_i2c_writeReg_Mask in ROM
```

PC 地址指向了 ROM 函数 `rom_i2c_writeReg_mask` 函数，而且 Backtrace 没有足够的有效信息，这种情况下，一般需要能够稳定复现来排查，如果此问题本身很难复现，那么可能是硬件或者野指针问题导致的。

Store access fault（StoreProhibited）

测试用例 当应用程序尝试在无效的内存位置进行写入操作时，会发生此类 CPU 异常，测试代码如下所示：

```c
typedef struct {
    uint8_t buf[10];
    uint8_t data;
} store_test_t;

void store_access_fault(store_test_t * store_test)
{
    store_test->data = 0x5F;
    printf("Data: %d\n", store_test->data);
}
void app_main(void)
{
    store_test_t * store_test = NULL;
    store_access_fault(store_test);
}
```

异常现象

- RISC-V

在 RISC-V 架构的芯片中，可以看到如下异常的打印，通过观察汇编指令（步骤可参考*附录 1 - ESP 反汇编对应的指令*）可发现出现这种异常时 PC 地址都指向了 `sb/sw` 等 store 相关指令。此时 MTVAL 存储的异常值 `0x0000000a` 说明想要访问的是 NULL 地址开始的某个数组或者结构体里的内容。

```
Guru Meditation Error: Core  0 panic'ed (Store access fault). Exception was
↪unhandled.

Core  0 register dump:
Stack dump detected
MEPC    : 0x42006960  RA      : 0x42006984  SP      : 0x3fc8f7c0  GP      :
↪0x3fc8b400
TP      : 0x3fc87d7c  T0      : 0x4005890e  T1      : 0x20000000  T2      :
↪0x00000000
S0/FP   : 0x3c023000  S1      : 0x00000000  A0      : 0x00000000  A1      :
↪0x3fc8f3f8
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x3fc8c000  A5      :
↪0x0000005f
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :
↪0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :
↪0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000007  MTVAL   :
↪0x0000000a
```

- XTENSA

在 XTENSA 架构的芯片中，可以看到下列异常打印，通过观察汇编指令（步骤可参考*附录 1 - ESP 反汇编对应的指令*）可发现出现这种异常时 PC 地址都指向了 `s8i/s32i` 等 store 相关的指令。此时 `EXCVADDR` 存储的异常值 `0x0000000a` 说明想要访问的是 NULL 地址开始的某个数组或者结构体里的内容。

```
Guru Meditation Error: Core  0 panic'ed (StoreProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x42007c21  PS      : 0x00060630  A0      : 0x82007c38  A1      :
↪0x3fc97d70
A2      : 0x00000000  A3      : 0x3c023f48  A4      : 0x3c023fc8  A5      :
↪0x3fc97d90
A6      : 0x3fc97d70  A7      : 0x0000000c  A8      : 0x0000005f  A9      :
↪0x3fc97d00
A10     : 0x00000031  A11     : 0x3fc97fac  A12     : 0x3fc97d70  A13     :
↪0x0000000c
A14     : 0x3fc941e4  A15     : 0x00000000  SAR     : 0x00000004  EXCCAUSE:
↪0x0000001d
EXCVADDR: 0x0000000a  LBEG    : 0x400556d5  LEND    : 0x400556e5  LCOUNT  :
↪0xfffffffd

Backtrace: 0x42007c1e:0x3fc97d70 0x42007c35:0x3fc97d90 0x4201923f:0x3fc97db0
↪0x40379e71:0x3fc97de0
```

定位方法 定位方法与*Load access fault*（*LoadProhibited*）类似，如果此时观察汇编代码（步骤可参考*附录 1 - ESP 反汇编对应的指令*）会看到如下指令：

```
sb      a5,10(a0)           // RSIC-V
s8i     a8, a2, 10         // XTENSA
```

在触发此类异常时，MTVAL/EXCVADDR 会自动更新为与异常有关的数据，从 RISC-V 异常信息中可以看到 MTVAL : 0x0000000a。一般来说 MTVAL 为 NULL 代表 CPU 尝试向 NULL 地址写数据，而接近 NULL 代表的是尝试向 NULL 地址开始的某个数组或者结构体的成员中写数据。

- 对于 RSIC-V 架构，PC 和 RA 寄存器包含了异常的函数指针，从 elf 中找出对应的函数排查即可
- 对于 Xtensa 架构，异常时会打印 backtrace，从 elf 中也可以找出对应的函数排查

实际案例 请参考测试用例。

注意事项 与*Load access fault（LoadProhibited）* 问题类似，存在野指针导致无法解析的情况，此时需要根据实际应用代码进一步分析。

Instruction Access Fault(InstrFetchProhibited)

测试用例 当应用程序尝试从无效的地址读取指令时，会发生此类 CPU 异常，此时 PC 寄存器往往会指向无效的内存地址，测试代码如下所示：

```
typedef void (*fptr_t)(void);
volatile fptr_t fptr = (fptr_t) 0x4;
fptr();
```

对应的 RISC-V 汇编指令（步骤可参考附录 *1 - ESP 反汇编对应的指令*）如下：

```
42006958:       1101                    addi    sp,sp,-32
4200695a:       ce06                    sw      ra,28(sp)
4200695c:       4791                    li      a5,4
4200695e:       c63e                    sw      a5,12(sp)
42006960:       47b2                    lw      a5,12(sp)
42006962:       9782                    jalr    a5
42006964:       40f2                    lw      ra,28(sp)
42006966:       6105                    addi    sp,sp,32
42006968:       8082                    ret
```

异常现象

- RISC-V

```
Guru Meditation Error: Core  0 panic'ed (Instruction access fault). Exception was␣
↪unhandled.

Core  0 register dump:
Stack dump detected
MEPC    : 0x00000004  RA      : 0x42006964  SP      : 0x3fc8f7b0  GP      :␣
↪0x3fc8b400
TP      : 0x3fc87d8c  T0      : 0x4005890e  T1      : 0x20000000  T2      :␣
↪0x00000000
S0/FP   : 0x3c023000  S1      : 0x00000000  A0      : 0x00000031  A1      :␣
↪0x3fc8f3f8
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x3fc8c000  A5      :␣
↪0x00000004
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :␣
↪0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
```

（下页继续）

(续上页)

```
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000001  MTVAL   :␣
↪0x00000004
```

- Xtensa

```
Guru Meditation Error: Core  0 panic'ed (InstrFetchProhibited). Exception was␣
↪unhandled.

Core  0 register dump:
PC      : 0x00000004  PS      : 0x00060630  A0      : 0x800d539a  A1      :␣
↪0x3ffb4a30
A2      : 0x400de738  A3      : 0x3f40379c  A4      : 0x3f40381c  A5      :␣
↪0x3ffb4a60
A6      : 0x3ffb4a40  A7      : 0x0000000c  A8      : 0x800e4dbe  A9      :␣
↪0x3ffb49d0
A10     : 0x00000031  A11     : 0x3ffafc64  A12     : 0x3ffb4a40  A13     :␣
↪0x0000000c
A14     : 0x3ffb2770  A15     : 0x0000cdcd  SAR     : 0x00000004  EXCCAUSE:␣
↪0x00000014
EXCVADDR: 0x00000004  LBEG    : 0x400014fd  LEND    : 0x4000150d  LCOUNT  :␣
↪0xfffffffd

Backtrace: 0x00000001:0x3ffb4a30 0x400d5397:0x3ffb4a60 0x400e54f8:0x3ffb4a80␣
↪0x400858e9:0x3ffb4ab0
```

定位方法 一般来说，此类问题都是由野指针导致的，此时 PC 寄存器已经没有价值，其他寄存器是否值得分析取决于野指针运行的时间，对于此用例来说，因为在当前函数已经触发 CPU 异常，因此 RA 寄存器中包含了上层函数指针（Xtensa 架构通过 Backtrace），此时可以分析出来异常。但是大多数时间，RA 寄存器也会被破坏，此时如果需要分析，可以根据如下步骤进行：

1. 稳定复现问题，保证每次出现的问题相同
2. 找寻问题的规律，比如执行哪些操作，或者打印哪些日志后开始异常
3. 根据程序异常的位置来结合代码分析，通过加调试日志以及逐步减少工程的代码量来看是否仍能复现问题

实际案例 如下图所示，异常时的现场已经完全被破坏，通过这些信息，没有办法去反推问题的原因，不过大多数寄存器都被破坏成一个固定值本身也是一个规律，像这种寄存器全被破坏的情况，一般是内存踩踏导致的问题，可以用 堆内存调试 的方法进行分析。

```
Anim->Dir 1
Guru Meditation Error: Core  0 panic'ed (InstrFetchProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x08a408a4  PS      : 0x00050033  A0      : 0x08a408a4  A1      : 0x3fcada60
A2      : 0x08a408a4  A3      : 0x08a408a4  A4      : 0x08a408a4  A5      : 0x08a408a4
A6      : 0x08a408a4  A7      : 0x08a408a4  A8      : 0x08a408a4  A9      : 0x08a408a4
A10     : 0x08a408a4  A11     : 0x08a408a4  A12     : 0x08a408a4  A13     : 0x08a408a4
A14     : 0x08a408a4  A15     : 0x08a408a4  SAR     : 0x00000024  EXCCAUSE: 0x00000014
EXCVADDR: 0x08a408a4  LBEG    : 0x08a408a4  LEND    : 0x08a408a4  LCOUNT  : 0x08a408a4


Backtrace: 0x08a408a1:0x3fcada60 0x08a408a1:0x08a408a4 |<-CORRUPTED



ELF file SHA256: 73673d3b6
```

Illegal Instruction

测试用例 CPU 取指阶段，取到非法指令（RISC-V spec 中定义为 16 bit 全为 0 的指令）会打印此错误。测试代码如下所示：

```
uint32_t instr_dram_addr = 0x0;
intptr_t instr_addr = MAP_DRAM_TO_IRAM((intptr_t)&instr_dram_addr);
typedef void (*illegal_instr_t)(void);
illegal_instr_t illegal_instr = (uint32_t *)instr_addr; // 确保 instr_addr
↪内存数据为 0x0
illegal_instr();
```

备注： Xtensa 平台中的 ESP32 不支持上述测试用例，其他芯片如要运行此测试用例，需要先关闭内存保护功能。

异常现象

- RISC-V

```
Guru Meditation Error: Core  0 panic'ed (Illegal instruction). Exception was
↪unhandled.

Core  0 register dump:
Stack dump detected
MEPC    : 0x4005f002  RA      : 0x42006962  SP      : 0x3fc8f7c0  GP      :
↪0x3fc8b400
0x42006962: illegal_instr_fault at /home/jacques/useful_example/crash_sim/store_
↪access_fault/main/hello_world_main.c:79

TP      : 0x3fc87d8c  T0      : 0x4005890e  T1      : 0x20000000  T2      :
↪0x00000000
S0/FP   : 0x3c023000  S1      : 0x00000000  A0      : 0x3fc8f840  A1      :
↪0x3fc8f3f8
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x3fc8c000  A5      :
↪0x4005f000
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :
↪0x00000000
```

（下页继续）

(续上页)

```
S4       : 0x00000000  S5       : 0x00000000  S6       : 0x00000000  S7       :␣
↪0x00000000
S8       : 0x00000000  S9       : 0x00000000  S10      : 0x00000000  S11      :␣
↪0x00000000
T3       : 0x00000000  T4       : 0x00000000  T5       : 0x00000000  T6       :␣
↪0x00000000
MSTATUS  : 0x00001881  MTVEC    : 0x40380001  MCAUSE   : 0x00000002  MTVAL    :␣
↪0x00000000
```

- Xtensa

```
Guru Meditation Error: Core  0 panic'ed (IllegalInstruction). Exception was␣
↪unhandled.
Memory dump at 0x403e298c: 00000000 00000000 00000000
Core  0 register dump:
PC       : 0x403e2990  PS       : 0x00060130  A0       : 0x820059fe  A1       :␣
↪0x3fcf2990
A2       : 0x4200c030  A3       : 0x3c0232fc  A4       : 0x3c02337c  A5       :␣
↪0x3fcf29c0
A6       : 0x3fcf29a0  A7       : 0x0000000c  A8       : 0x820059f4  A9       :␣
↪0x3fcf2930
A10      : 0x00000031  A11      : 0x3fcf2be4  A12      : 0x3fcf29a0  A13      :␣
↪0x0000000c
A14      : 0x3fc922c4  A15      : 0x00000000  SAR      : 0x00000004  EXCCAUSE:␣
↪0x00000000
EXCVADDR: 0x00000000  LBEG     : 0x400556d5  LEND     : 0x400556e5  LCOUNT   :␣
↪0xfffffffd

Backtrace: 0x403e298d:0x3fcf2990 |<-CORRUPTED
```

定位方法 虽然在 ESP-IDF 文档中有 Illegal instruction 说明，其中提到可能有 3 类情况触发此问题，但是实际使用中，大多数都是 CPU 从 flash 取指令异常触发此问题，还有一部分是类似测试用例这种野指针导致的从内存取值异常，从 PC 地址一般可以看出是从 IRAM 还是从 flash 取址异常。

从 flash 取指出现问题可能的原因包括 ESP 芯片休眠时 flash 下电、CS 上拉不稳定，flash 供电不稳定，以及 flash suspend 功能等。

实际案例 下图是 ESP 芯片在低功耗场景下的一种异常情况，从日志中可以看到异常前有 power save 的打印，可以怀疑问题跟芯片休眠有关。后在 menuconfig 配置项中关闭休眠时给 flash 掉电 (CONFIG_ESP_SLEEP_POWER_DOWN_FLASH)，以及开启休眠时 CS 软件上拉 (CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND) 这两个选项，问题不再复现。

```
battery_read_key:3
is_wifi_enable=0 is_standalone=0
test sleep UPM_init before
test sleep UPM_init before
I (11323) pm: Frequency switching config: CPU_MAX: 160, APB_MAX: 80, APB_MIN: 10, Light sleep: ENABLED
I (11324) sleep: Enable automatic switching of GPIO sleep configuration
test sleep UPM_init behind
Guru Meditation Error: Core  0 panic'ed (Illegal instruction). Exception was unhandled.

Stack dump detected
Core  0 register dump:
MEPC     : 0x42002f7c  RA       : 0x42004f82  SP       : 0x3fca48e0  GP       : 0x3fc94400
0x42002f7c: heap_caps_init at E:/utec/esp-idf0829/components/heap/heap_caps_init.c:94 (discriminator 1)

0x42004f82: app_main at E:/utec/lock/esp_lock/main/main.cpp:315

TP       : 0x3fc7ef88  T0       : 0x4005890e  T1       : 0x42001a3e  T2       : 0xffffffff
0x42001a3e: console_write at E:/utec/esp-idf0829/components/vfs/vfs_console.c:73

S0/FP    : 0x3c0e6000  S1       : 0x3c0e6000  A0       : 0x3c0e5d8c  A1       : 0x3fca4958
A2       : 0x00000000  A3       : 0x00000001  A4       : 0x600c0000  A5       : 0x00000000
A6       : 0x00000010  A7       : 0x000000c8  S2       : 0x00000000  S3       : 0x0007a000
```

Cache disable but cached memory region accessed（Cache error）

异常原理 在 flash 操作（读写擦）时会关闭 cache ，如果此时中断触发（使用 ESP_INTR_FLAG_IRAM 修饰的中断可以在 cache 关闭的时候触发），如中断函数中存在函数需要通过 cache 从 flash 中取指或取数据（包括 mmap），会因为 cache 被关闭导致异常。

测试用例 在关闭 cache 后，访问并打印 flash 中的数据。测试代码如下所示：

```
static const uint32_t s_in_rodata[8] = { 0x12345678, 0xfedcba98 };
spi_flash_disable_interrupts_caches_and_other_cpu();
volatile uint32_t* src = (volatile uint32_t*) s_in_rodata;
uint32_t v1 = src[0];
uint32_t v2 = src[1];
ets_printf("%lx %lx\n", v1, v2);
```

异常现象

- RISC-V

```
Guru Meditation Error: Core  0 panic'ed (Cache error).
access to cache while dbus or cache is disabled

Stack dump detected
Core  0 register dump:
MEPC    : 0x40000040  RA      : 0x40381e82  SP      : 0x3fc8f7c0  GP      :␣
↪0x3fc8b400
0x40381e82: cache_access_test_func at /home/jacques/useful_example/crash_sim/store_
↪access_fault/main/hello_world_main.c:38

TP      : 0x3fc87d5c  T0      : 0x4005890e  T1      : 0x20000000  T2      :␣
↪0x00000000
S0/FP   : 0x3c023000  S1      : 0x00000000  A0      : 0x3c025610  A1      :␣
↪0x00000000
A2      : 0x00000000  A3      : 0x00000200  A4      : 0x600c4000  A5      :␣
↪0x3c02561c
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :␣
↪0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000019  MTVAL   :␣
↪0x6944806f
```

- Xtensa

```
Guru Meditation Error: Core  0 panic'ed (Cache disabled but cached memory region␣
↪accessed).

Core  0 register dump:
PC      : 0x400830ed  PS      : 0x00060a34  A0      : 0x800d539a  A1      :␣
↪0x3ffb4a40
0x400830ed: cache_access_test_func at /home/jacques/useful_example/crash_sim/store_
↪access_fault/main/hello_world_main.c:32

A2      : 0x400de738  A3      : 0x3f40379c  A4      : 0x3f40381c  A5      :␣
↪0x3ffb4a60
0x400de738: vprintf at /builds/idf/crosstool-NG/.build/xtensa-esp32-elf/src/newlib/
↪newlib/libc/stdio/vprintf.c:30
```

(下页继续)

(续上页)

```
A6       : 0x3ffb4a40  A7       : 0x0000000c  A8       : 0x3f4040c4  A9       :␣
↪0x3ffb4a10
A10      : 0x00000001  A11      : 0xbad00bad  A12      : 0x3ffb2608  A13      :␣
↪0x0000000c
A14      : 0x3ffb2770  A15      : 0x0000cdcd  SAR      : 0x00000004  EXCCAUSE:␣
↪0x00000007
EXCVADDR: 0x00000000  LBEG     : 0x400014fd  LEND     : 0x4000150d  LCOUNT   :␣
↪0xfffffffd

Backtrace: 0x400830ea:0x3ffb4a40 0x400d5397:0x3ffb4a60 0x400e54e4:0x3ffb4a80␣
↪0x40085915:0x3ffb4ab0
```

分析方法　在出现此类问题，主要观察异常时的 PC 地址，如果地址在 flash 中，说明在关闭 cache 的时候从 flash 读指令，此时只需要把对应的函数使用 `IRAM_ATTR` 修饰即可，需要注意的是 `IRAM_ATTR` 只针对当前函数，如果此函数还有子函数，那么子函数也需要使用 `IRAM_ATTR` 进行修饰。

如果地址在 IRAM 中，说明此函数本身没问题，可能是此函数对应的输入参数等需要从 flash 中读取。

实际案例　在 flash 读写操作（导致 cache 关闭）时已放入 IRAM 的 gptimer 中断也触发，gptimer ISR 里执行了一些需要从 flash 读取内容的操作触发此问题：

```
IDiff Time: 180
 (00:00:03.131) NVM_TASK: InitDiff Time: 179
ializing NVS Flash
Diff Time: 179
Diff Time: 183
Diff Time: 183
Diff Time: 183
Guru Meditation Error: Core  0 panic'ed (Cache disabled but cached memory region accessed).

Core  0 register dump:
PC      : 0x40086728  PS      : 0x00060035  A0      : 0x800868a0  A1      : 0x3ffbfb40
0x40086728: gpio_set_level at /home/jacques/sdk/esp-idf/components/driver/gpio/gpio.c:237

A2      : 0x3ffbeff8  A3      : 0x3ffc1126  A4      : 0x3ffc111c  A5      : 0x3ffc1126
A6      : 0x00000000  A7      : 0x0006a0d9  A8      : 0x800842d8  A9      : 0x00000002
A10     : 0x000000d0  A11     : 0x00000001  A12     : 0x00000000  A13     : 0x80000000
A14     : 0x3ffc0a20  A15     : 0x00000000  SAR     : 0x00000005  EXCCAUSE: 0x00000007
EXCVADDR: 0x00000000  LBEG    : 0x40082e88  LEND    : 0x40082e90  LCOUNT  : 0x00000026
0x40082e88: esp_timer_impl_get_counter_reg at /home/jacques/sdk/esp-idf/components/esp_timer/src/esp_timer_impl_lac.c:118
0x40082e90: esp_timer_impl_get_counter_reg at /home/jacques/sdk/esp-idf/components/esp_timer/src/esp_timer_impl_lac.c:128

Core  0 was running in ISR context:
EPC1    : 0x400d3633  EPC2    : 0x00000000  EPC3    : 0x4008b20c  EPC4    : 0x00000000
0x400d3633: uart_hal_write_txfifo at /home/jacques/sdk/esp-idf/components/hal/uart_hal_iram.c:26
0x4008b20c: spi_flash_ll_cmd_is_done at /home/jacques/sdk/esp-idf/components/hal/esp32/include/hal/spi_flash_ll.h:77
 (inlined by) spi_flash_hal_poll_cmd_done at /home/jacques/sdk/esp-idf/components/hal/spi_flash_hal_common.inc:38



Backtrace: 0x40086725:0x3ffbfb40 0x4008689d:0x3ffbfb70 0x400846bd:0x3ffbfbb0 0x4008b206:0x3ffd6700 0x4008b78e:0x3ffd6730 0x40
6850 0x400ef6f5:0x3ffd6880 0x400efaa3:0x3ffd68d0 0x400ed6d5:0x3ffd6940 0x400ee08b:0x3ffd69a0 0x400ee14f:0x3ffd69e0 0x400ec32
0x40086725: gpio_intr_disable at /home/jacques/sdk/esp-idf/components/driver/gpio/gpio.c:191
0x4008689d: gptimer_default_isr at /home/jacques/sdk/esp-idf/components/driver/gptimer/gptimer.c:543
0x400846bd: _xt_lowint1 at /home/jacques/sdk/esp-idf/components/xtensa/xtensa_vectors.S:1240
0x4008b206: spi_flash_hal_poll_cmd_done at /home/jacques/sdk/esp-idf/components/hal/spi_flash_hal_common.inc:37
0x4008b78e: spi_flash_hal_read at /home/jacques/sdk/esp-idf/components/hal/spi_flash_hal_common.inc:194
0x4008df15: spi_flash_chip_generic_read at /home/jacques/sdk/esp-idf/components/spi_flash/spi_flash_chip_generic.c:246
0x40085a22: esp_flash_read at /home/jacques/sdk/esp-idf/components/spi_flash/esp_flash_api.c:899
0x400e96ff: esp_partition_read_raw at /home/jacques/sdk/esp-idf/components/esp_partition/partition_target.c:102
0x400edcf8: nvs::NVSPartition::read_raw(unsigned int, void*, unsigned int) at /home/jacques/sdk/esp-idf/components/nvs_flash/
0x400ef6f5: nvs::Page::load(nvs::Partition*, unsigned long) at /home/jacques/sdk/esp-idf/components/nvs_flash/src/nvs_page.c
0x400efaa3: nvs::PageManager::load(nvs::Partition*, unsigned long, unsigned long) at /home/jacques/sdk/esp-idf/components/nv
```

从上述截图我们可以观察到几个问题点

1. 可确定问题为在 flash 读写操作的时候触发了 gptimer 中断，且 ISR 里有存在于 flash 里的内容
2. 异常时 PC 指向的函数 `gpio_set_level` 已放入 IRAM，而不是在 flash 中，因此不需要经过 cache，在 cache 关闭时也应正常运行

经过分析，最终排查到是函数 `gpio_set_level` 的输入参数 `LED_matrix` 使用 const 修饰导致的，被 const 修饰的参数会放置在 flash 的.rodata 段，因此读取此参数需通过 cache。

```
//static bool LEDsTimer_on_alarm(gptimer_handle_t LEDtimer_hdl, const gptimer_alarm_ev
static bool IRAM_ATTR LEDsTimer_on_alarm(gptimer_handle_t LEDtimer_hdl, const gptimer_
{
    int64_t start, diff;
    start = esp_timer_get_time();
    BaseType_t high_task_awoken = pdFALSE;
    /***LLL TEST 12.04.2024
    QueueHandle_t queue = (QueueHandle_t)user_data;
    *******/
    uint8_t count;
    static bool pin_level = 0;

    gpio_set_level( IO0_PIN, 1 );
    ledc_timer_rst( ledc_timer.speed_mode, ledc_timer.timer_num );  //To synchronized

#if 1
    /*--- Switch OFF all COMMON */
    for ( count = 0; count < GPIO_NUM_MAX_ROW; count++ )
    {
        gpio_set_level( LED_matrix.row_GPIO[count], 0 );
    }

    /*--- Set the LED's duty cycle */
    for ( count = 0; count < GPIO_NUM_MAX_COL; count++ )
    {
        //uint8_t index =  row_count * GPIO_NUM_MAX_COL + count;
        /***** Not SAFE */
        ledc_set_duty(ledc_channel[count].speed_mode, ledc_channel[count].channel, dut
        ledc_update_duty(ledc_channel[count].speed_mode, ledc_channel[count].channel);
        /*********/
        /**** SAFE but doesn't work properly and not putted in IRAM *
        ledc_set_duty_and_update(ledc_channel[count].speed_mode, ledc_channel[count].
```

注意事项 Cache disable 的异常在 Xtensa 平台上捕捉的更好，在 RISC-V 平台上可能会表现为因 cache 读不到数据导致的其他错误，如 Illegal instruction. 代码示例如下：

```
gpio_set_level(12, 1);
spi_flash_disable_interrupts_caches_and_other_cpu();
gpio_set_level(12, 0);
```

在 ESP32-C3（RISC-V 平台）中运行时，出现的 crash 如下

```
Guru Meditation Error: Core  0 panic'ed (Illegal instruction). Exception was␣
↪unhandled.

Core  0 register dump:
Stack dump detected
MEPC     : 0x42006ef2  RA       : 0x40381e7e  SP       : 0x3fc8f7f0  GP       :␣
↪0x3fc8b400
0x42006ef2: gpio_set_level at /home/jacques/sdk/esp-idf/components/driver/gpio/
↪gpio.c:233
0x40381e7e: cache_access_test_func at /home/jacques/useful_example/crash_test/main/
↪hello_world_main.c:50
TP       : 0x3fc87d4c  T0       : 0x4005890e  T1       : 0x20000000  T2       :␣
↪0x00000000
S0/FP    : 0x3c023000  S1       : 0x00000000  A0       : 0x0000000c  A1       :␣
↪0x00000000
A2       : 0x00000200  A3       : 0x00000200  A4       : 0x600c4000  A5       :␣
↪0x00000000
A6       : 0x60023000  A7       : 0x0000000a  S2       : 0x00000000  S3       :␣
↪0x00000000
```

（下页继续）

(续上页)

```
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000002  MTVAL   :␣
↪0x00000000
```

Interrupt wdt timeout on CPU0/CPU1

异常原理 中断看门狗使用 Timer group 1 上的硬件看门狗定时器，通过在滴答定时器 SysTick 中断中添加喂狗操作来监控系统任务调度是否正常。当长时间（默认 300 ms）未在 SysTick 中断中做喂狗处理时，中断看门狗中断触发。更多信息请参考 Interrupt Watchdog Timer (IWDT)。

测试用例 可以通过关中断的方式来触发此问题，测试代码如下所示：

```
static portMUX_TYPE s_init_spinlock = portMUX_INITIALIZER_UNLOCKED;
portENTER_CRITICAL(&s_init_spinlock);
while (1) {
}
```

异常现象

- RISC-V

```
Guru Meditation Error: Core  0 panic'ed (Interrupt wdt timeout on CPU0).

Core  0 register dump:
Stack dump detected
MEPC    : 0x42006964  RA      : 0x42006964  SP      : 0x3fc911a0  GP      :␣
↪0x3fc8b400
0x42006964: interrupt_wdt_fault_task at /home/jacques/useful_example/crash_test/
↪main/hello_world_main.c:121 (discriminator 1)

TP      : 0x3fc8971c  T0      : 0x00000000  T1      : 0x00000000  T2      :␣
↪0x00000000
S0/FP   : 0x00000000  S1      : 0x00000000  A0      : 0x00000001  A1      :␣
↪0x00000000
A2      : 0x00000000  A3      : 0x00000004  A4      : 0x00000001  A5      :␣
↪0x3fc8c000
A6      : 0x00000000  A7      : 0x00000000  S2      : 0x00000000  S3      :␣
↪0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000018  MTVAL   :␣
↪0x0000a001
```

- Xtensa

```
Guru Meditation Error: Core  0 panic'ed (Interrupt wdt timeout on CPU0).

Core  0 register dump:
```

(下页继续)

(续上页)

```
PC      : 0x40378b06  PS      : 0x00060634  A0      : 0x82002239  A1      :␣
↪0x3fc984f0
0x40378b06: esp_cpu_wait_for_intr at /home/jacques/sdk/esp-idf/components/esp_hw_
↪support/cpu.c:110

A2      : 0x00000000  A3      : 0x00000000  A4      : 0x3fc96c70  A5      :␣
↪0x3fc96c50
A6      : 0x42005e10  A7      : 0x00000000  A8      : 0x82009452  A9      :␣
↪0x3fc984b0
0x42005e10: timer_task at /home/jacques/sdk/esp-idf/components/esp_timer/src/esp_
↪timer.c:470

A10     : 0x00000000  A11     : 0x00000000  A12     : 0x3fc96c50  A13     :␣
↪0x3fc96c30
A14     : 0x00000000  A15     : 0x00000000  SAR     : 0x00000000  EXCCAUSE:␣
↪0x00000005
EXCVADDR: 0x00000000  LBEG    : 0x00000000  LEND    : 0x00000000  LCOUNT  :␣
↪0x00000000


Backtrace: 0x40378b03:0x3fc984f0 0x42002236:0x3fc98510 0x4037b5a9:0x3fc98530␣
↪0x40379e71:0x3fc98550
0x40378b03: xt_utils_wait_for_intr at /home/jacques/sdk/esp-idf/components/xtensa/
↪include/xt_utils.h:81
 (inlined by) esp_cpu_wait_for_intr at /home/jacques/sdk/esp-idf/components/esp_hw_
↪support/cpu.c:101
0x42002236: esp_vApplicationIdleHook at /home/jacques/sdk/esp-idf/components/esp_
↪system/freertos_hooks.c:59
0x4037b5a9: prvIdleTask at /home/jacques/sdk/esp-idf/components/freertos/FreeRTOS-
↪Kernel/tasks.c:4269 (discriminator 1)
0x40379e71: vPortTaskWrapper at /home/jacques/sdk/esp-idf/components/freertos/
↪FreeRTOS-Kernel/portable/xtensa/port.c:149

Core  1 register dump:
PC      : 0x42007c32  PS      : 0x00060034  A0      : 0x80379e74  A1      :␣
↪0x3fc99f20
0x42007c32: interrupt_wdt_fault_task at /home/jacques/useful_example/crash_test/
↪main/hello_world_main.c:121 (discriminator 1)

A2      : 0x00000000  A3      : 0x00000000  A4      : 0x00000001  A5      :␣
↪0x00000000
A6      : 0x00000001  A7      : 0x00000000  A8      : 0x82007c32  A9      :␣
↪0x3fc99ef0
A10     : 0x00000001  A11     : 0x3fc93a54  A12     : 0x00060020  A13     :␣
↪0x00060023
A14     : 0x00060323  A15     : 0x0000abab  SAR     : 0x00000000  EXCCAUSE:␣
↪0x00000005
EXCVADDR: 0x00000000  LBEG    : 0x00000000  LEND    : 0x00000000  LCOUNT  :␣
↪0x00000000


Backtrace: 0x42007c2f:0x3fc99f20 0x40379e71:0x3fc99f40
0x42007c2f: vPortEnterCritical at /home/jacques/sdk/esp-idf/components/freertos/
↪FreeRTOS-Kernel/portable/xtensa/include/freertos/portmacro.h:573
 (inlined by) interrupt_wdt_fault_task at /home/jacques/useful_example/crash_test/
↪main/hello_world_main.c:120
0x40379e71: vPortTaskWrapper at /home/jacques/sdk/esp-idf/components/freertos/
↪FreeRTOS-Kernel/portable/xtensa/port.c:149
```

分析方法 在出现此类问题，主要观察异常时的 PC 地址，即可确定触发 crash 的位置。对于多核架构，需要观察几个核心的寄存器，分析其中触发问题的方法。

可能的原因及具体解释如下：

1. 长时间关中断
2. ISR 函数中存在阻塞
3. 没有清中断

长时间关中断 关中断操作本身是为了保护一些不能中途停止执行的程序而设计的，它用来禁止系统响应中断源的中断请求。长时间关中断是导致中断看门狗触发的最主要原因。

如测试用例 中进入临界区的操作会关闭中断。关中断之后，SysTick 中断无法触发来及时进行喂狗操作，这后续会导致中断看门狗超时（关中断无法屏蔽看门狗中断）并触发看门狗异常处理程序，此时需要用户自己排查上层关中断的逻辑，依据出问题的代码位置进一步分析。

ISR 函数中存在阻塞 在 ESP 芯片产生某一中断之后，会调用对应的 ISR 进行处理，在 ISR 函数中应做的操作应尽可能少，可以将需要耗时的操作转移到其他非中断函数中处理。如 ISR 函数需耗费大量时间来运行或存在阻塞导致未在中断看门狗触发前退出 ISR，就会触发异常。

以下是 GPIO ISR 阻塞的一个简单示例：

```
#define GPIO_INPUT_IO_0     4

static void IRAM_ATTR gpio_isr_handler(void* arg)
{
    ets_printf("ISR\n");
    while(1){}
}

void app_main(void)
{
    gpio_config_t io_conf;
    io_conf.intr_type = GPIO_INTR_POSEDGE;
    io_conf.pin_bit_mask = (1ULL<<GPIO_INPUT_IO_0);
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.pull_up_en = 1;
    gpio_config(&io_conf);

    gpio_install_isr_service(0);
    gpio_isr_handler_add(GPIO_INPUT_IO_0, gpio_isr_handler, (void*) GPIO_INPUT_IO_
↪0);
}
```

将 GPIO 4 手动拉高，触发 GPIO 中断，打印如下 crash 信息：

```
ISR
Guru Meditation Error: Core  0 panic'ed (Interrupt wdt timeout on CPU0).
Core  0 register dump:
PC      : 0x40082580  PS      : 0x00060034  A0      : 0x80081560  A1      :␣
↪0x3ffb07c0
0x40082580: gpio_isr_handler at /home/jacques/problem/wdt/int_wdt/build/../main/
↪hello_world_main.c:40 (discriminator 1)

A2      : 0x00000004  A3      : 0x3ff0015c  A4      : 0x800d6d82  A5      :␣
↪0x3ffb3400
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x80082580  A9      :␣
↪0x3ffb0730
A10     : 0x00000004  A11     : 0x00000004  A12     : 0x00000004  A13     :␣
↪0x00000000
A14     : 0x00000000  A15     : 0x3ffb3400  SAR     : 0x0000001c  EXCCAUSE:␣
↪0x00000005
```

（下页继续）

(续上页)

```
EXCVADDR: 0x00000000  LBEG    : 0x00000000  LEND    : 0x00000000  LCOUNT  :␣
↪0x00000000
Core  0 was running in ISR context:
EPC1    : 0x400d1097  EPC2    : 0x00000000  EPC3    : 0x00000000  EPC4    :␣
↪0x40082580
0x400d1097: uart_hal_write_txfifo at /home/jacques/sdk/esp-idf/components/hal/uart_
↪hal_iram.c:35

0x40082580: gpio_isr_handler at /home/jacques/problem/wdt/int_wdt/build/../main/
↪hello_world_main.c:40 (discriminator 1)

Backtrace:0x4008257d:0x3ffb07c0 0x4008155d:0x3ffb07e0 0x40082102:0x3ffb0800␣
↪0x400d6702:0x3ffb3450 0x400862a2:0x3ffb3470 0x40087865:0x3ffb3490
```

通过 PC 和 Backtrace，我们可以确认问题出现在 gpio_isr_handle 函数中。此外，`Core 0 was running in ISR context` 这行日志也表明当前在中断函数中运行。

没有清中断 在 ISR 函数结束后，需要手动清除相应的中断，如果此时不做清除，中断将会一直触发，从而使得 Systick 中断无法响应。

如*ISR 函数中存在阻塞* 里的示例，如果把 `gpio_isr_handler` 中的 while(1) 循环去掉，然后将 ESP-IDF GPIO 驱动中的 gpio_hal_clear_intr_status_bit 注释后重新编译烧写运行上述程序，可以发现同样触发了中断看门狗的 crash。

如果用户没有自行修改默认驱动，清中断一般来说会由驱动底层自行完成，若此时出现了指向内部驱动的中断看门狗异常，则可以进一步查看驱动底层是否正常做了清中断操作。

实际案例 通过上述描述，可能会认为中断看门狗问题极易分析，但实际的应用中中断看门狗问题往往非独立存在，一般会跟 FreeRTOS 相结合，FreeRTOS 很多操作都需要关中断处理，一旦 FreeRTOS 出现异常则容易卡住并表现为中断看门狗问题，实际 crash 案例如下：

```
Guru Meditation Error: Core  0 panic'ed (Interrupt wdt timeout on CPU0)
Core 0 register dump:
PC      : 0x400915d2  PS      : 0x00060b34  A0      : 0x800903ed  A1      :␣
↪0x3ffd4e80
A2      : 0x3ffd3598  A3      : 0x3ffd4ff0  A4      : 0x00000c9f  A5      :␣
↪0x3f408990
A6      : 0x3ffbf580  A7      : 0x00060023  A8      : 0x3ffd4ff0  A9      :␣
↪0x0000000f
A10     : 0x3ffd4ff0  A11     : 0x0000000f  A12     : 0x00060b20  A13     :␣
↪0x00000001
A14     : 0x3ffeddf0  A15     : 0x3ffea57c  SAR     : 0x00000018  EXCCAUSE:␣
↪0x00000005
EXCVADDR: 0x00000000  LBEG    : 0x400014fd  LEND    : 0x4000150d  LCOUNT  :␣
↪0xfffffffe

ELF file SHA256: eadf38922e335ed1

Backtrace: 0x400915cf:0x3ffd4e80 0x400903ea:0x3ffd4ea0 0x4008f514:0x3ffd4ec0␣
↪0x400ffb91:0x3ffd4f00 0x40136612:0x3ffd4f30
```

解析 Backtrace 信息，如下：

```
~$ xtensa-esp32-elf-addr2line -afe esp32_interrupt_wdt.elf 0x400915cf:0x3ffd4e80␣
↪0x400903ea:0x3ffd4ea0 0x4008f514:0x3ffd4ec0 0x400ffb91:0x3ffd4f00␣
↪0x40136612:0x3ffd4f30
0x400915cf
vListInsert
/arm/esp-idf/components/freertos/list.c:205
```

(下页继续)

(续上页)

```
0x400903ea
vTaskPlaceOnEventList
/arm/esp-idf/components/freertos/tasks.c:2901 (discriminator 2)
0x4008f514
xQueueGenericReceive
/arm/esp-idf/components/freertos/queue.c:1596
0x400ffb91
arch_os_mbox_fetch
/arm/components/esp32/arch_os.c:352 (discriminator 4)
0x40136612
msg_fetch_block
/arm/components/libs/d0/delayzero.c:631
```

通过上述日志可看出问题并不是 crash 在中断里，而是指向了 FreeRTOS 本身，通过查找 vListInsert 可发现在以下 for 循环里出现问题：

```
if( xValueOfInsertion == portMAX_DELAY )
{
    pxIterator = pxList->xListEnd.pxPrevious;
}
else
{
    for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->
↪xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
    {
        /* There is nothing to do here, just iterating to the wanted
        insertion position. */
    }
}
```

此时可进一步检查对应上层代码在此刻是否有关中断操作，最后在 vTaskPlaceOnEventList 中可看到确实是先关中断，再去执行的链表插入操作。

```
void vTaskPlaceOnEventList( List_t * const pxEventList, const TickType_t␣
↪xTicksToWait )
{
    configASSERT( pxEventList );
    taskENTER_CRITICAL(&xTaskQueueMutex);

    vListInsert( pxEventList, &( pxCurrentTCB[xPortGetCoreID()]->xEventListItem )␣
↪);

    prvAddCurrentTaskToDelayedList( xPortGetCoreID(), xTicksToWait);
    taskEXIT_CRITICAL(&xTaskQueueMutex);
}
```

至此分析完成，此问题是一个表现为中断看门狗的 FreeRTOS 问题，因此排查这个问题需要分析为什么会在 for 循环中不再退出。FreeRTOS 本身出现问题的概率极低，此时大多数是由于上层应用代码使用不当导致的，可能的原因包括内存踩踏，关中断后调用阻塞操作等问题。

附录 1 - ESP 反汇编对应的指令

- Xtensa ESP32 : `xtensa-esp32-elf-objdump -S xxx.elf > a.txt`
- Xtensa ESP32-S2 : `xtensa-esp32s2-elf-objdump -S xxx.elf > a.txt`
- Xtensa ESP32-S3 : `xtensa-esp32s3-elf-objdump -S xxx.elf > a.txt`
- RISC-V : `riscv32-esp-elf-objdump -S xxx.elf > a.txt`

其中 `xxx.elf` 为工程 elf 文件，此文件往往在工程目录下的 build 文件夹里。`a.txt` 为反汇编的输出问题，里面包含了实际的汇编指令。

使用 Backtrace & Coredump 定位问题

Backtrace 综述 默认情况下 Panic 发生后，除非启用了 *CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT*，否则紧急处理程序会将 CPU 寄存器和回溯打印到控制台来供开发者进一步调试分析，这部分在 ESP-IDF 编程指南的 寄存器转储与回溯 章节有进一步描述。

不同 CPU 架构 Backtrace 信息不同，目前 ESP 主要存在以 ESP32-S3 为例的 Xtensa 架构和以 ESP32-C3 为例的 RISC-V 架构：

- Xtensa 主要关注 PC 和 EXCVADDR，以及 Backtrace 地址
- RISC-V 主要关注 MEPC 和 MTVAL，以及堆栈信息（默认不打印 Backtrace 地址）

寄存器信息和 Backtrace 地址可以通过第三方串口工具打印并手动通过 *addr2line* 工具解析，也可以通过 *idf.py monitor* 打印并自动解析。以下我们会分别详述这两种 CPU 架构如何使用 Backtrace & Coredump 定位问题。

xTensa Backtrace ESP 目前使用 xTensa 架构的芯片有 ESP8266、ESP32、ESP32-S2、ESP32-S3。

以下是 xTensa 对应的错误日志示例：

```
Guru Meditation Error: Core  0 panic'ed (LoadProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x42016d2f  PS      : 0x00060c30  A0      : 0x820088ac  A1      :␣
↪0x3fc98d00
A2      : 0x00000000  A3      : 0x3fc98f34  A4      : 0x0000000a  A5      :␣
↪0x3fc98d20
A6      : 0x3fc98d00  A7      : 0x0000000c  A8      : 0x8200ca00  A9      :␣
↪0x3fc989b0
A10     : 0x00000002  A11     : 0xffffffff  A12     : 0x00000002  A13     :␣
↪0x3fc98bc0
A14     : 0x3fc989c0  A15     : 0x00000001  SAR     : 0x00000019  EXCCAUSE:␣
↪0x0000001c
EXCVADDR: 0x00000008  LBEG    : 0x400556d5  LEND    : 0x400556e5  LCOUNT  :␣
↪0xfffffff9


Backtrace: 0x42016d2c:0x3fc98d00 0x420088a9:0x3fc98d20 0x420180ff:0x3fc98d40␣
↪0x4037a2dd:0x3fc98d70
```

此时可以进一步使用 *addr2line* 命令，根据 *.elf* 文件解析 Backtrace 为可读函数名，指令示例如下：

```
xtensa-esp32s3-elf-addr2line -pfiaC -e path.elf 0x42016d2c:0x3fc98d00
```

对应结果如下：

```
? xtensa-esp32s3-elf-addr2line -pfiaC -e ./build/idf_debug_method.elf␣
↪0x42016d2c:0x3fc98d00 0x420088a9:0x3fc98d20 0x420180ff:0x3fc98d40␣
↪0x4037a2dd:0x3fc98d70
0x42016d2c: new_monkey_born at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:18
0x420088a9: app_main at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:70
0x420180ff: main_task at /home/libo/esp/github_master/components/freertos/app_
↪startup.c:208 (discriminator 13)
0x4037a2dd: vPortTaskWrapper at /home/libo/esp/github_master/components/freertos/
↪FreeRTOS-Kernel/portable/xtensa/port.c:162
```

如果使用 IDF monitor 来打印日志，它会自动调用上述 addr2line 并打印解析后的结果，如下：

```
Guru Meditation Error: Core  0 panic'ed (LoadProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x42016d2f  PS      : 0x00060c30  A0      : 0x820088ac  A1      :␣
↪0x3fc98d00
0x42016d2f: new_monkey_born at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:19

A2      : 0x00000000  A3      : 0x3fc98f34  A4      : 0x0000000a  A5      :␣
↪0x3fc98d20
A6      : 0x3fc98d00  A7      : 0x0000000c  A8      : 0x8200ca00  A9      :␣
↪0x3fc989b0
A10     : 0x00000002  A11     : 0xffffffff  A12     : 0x00000002  A13     :␣
↪0x3fc98bc0
A14     : 0x3fc989c0  A15     : 0x00000001  SAR     : 0x00000019  EXCCAUSE:␣
↪0x0000001c
EXCVADDR: 0x00000008  LBEG    : 0x400556d5  LEND    : 0x400556e5  LCOUNT  :␣
↪0xfffffff9
0x400556d5: strlen in ROM

0x400556e5: strlen in ROM

Backtrace: 0x42016d2c:0x3fc98d00 0x420088a9:0x3fc98d20 0x420180ff:0x3fc98d40␣
↪0x4037a2dd:0x3fc98d70
0x42016d2c: new_monkey_born at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:18

0x420088a9: app_main at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:70

0x420180ff: main_task at /home/libo/esp/github_master/components/freertos/app_
↪startup.c:208 (discriminator 13)

0x4037a2dd: vPortTaskWrapper at /home/libo/esp/github_master/components/freertos/
↪FreeRTOS-Kernel/portable/xtensa/port.c:162
```

此时可以根据详细的 backtrace 日志、Guru Meditation 错误 和以下：

- PC: 程序计数器（Program Counter）
- EXCVADDR: 异常向量地址（Exception Vector Address）

指针来进一步调试分析，Guru Meditation 错误的常见原因也可参考*利用 Guru Meditation 错误打印定位问题* 章节。

RISC-V Backtrace ESP 目前使用 RISC-V 架构的芯片有 ESP32-C3、ESP32-C2、ESP32-C6、ESP32-H2。以下是 RISC-V 对应的错误日志示例：

```
Guru Meditation Error: Core  0 panic'ed (Load access fault). Exception was␣
↪unhandled.

Core  0 register dump:
MEPC    : 0x42007988  RA      : 0x42007a4e  SP      : 0x3fc8fed0  GP      :␣
↪0x3fc8b200
TP      : 0x3fc870f8  T0      : 0x4005890e  T1      : 0x3fc8fb2c  T2      :␣
↪0x00000000
S0/FP   : 0x0000000a  S1      : 0x3fc90948  A0      : 0x00000000  A1      :␣
↪0x3fc8fb08
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000000  A5      :␣
↪0x00000009
A6      : 0x60023000  A7      : 0x0000000a  S2      : 0x00000000  S3      :␣
↪0x00000000
```

（下页继续）

(续上页)

```
S4       : 0x00000000  S5       : 0x00000000  S6       : 0x00000000  S7       :␣
↪0x00000000
S8       : 0x00000000  S9       : 0x00000000  S10      : 0x00000000  S11      :␣
↪0x00000000
T3       : 0x00000000  T4       : 0x00000000  T5       : 0x00000000  T6       :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC    : 0x40380001  MCAUSE   : 0x00000005  MTVAL    :␣
↪0x00000008
MHARTID : 0x00000000

Stack memory:
3fc8fed0: 0x3c021bd0 0x00000000 0x3c022000 0x42015de4 0x00000000 0x00001388␣
↪0x00000001 0x00000000
3fc8fef0: 0x00000000 0x00000000 0x00000000 0x40384538 0x00000000 0x00000000␣
↪0x00000000 0x00000000
```

此时可以进一步使用 *addr2line* 命令，根据 *.elf* 文件解析 Backtrace 为可读函数名，指令示例如下：

```
riscv32-esp-elf-addr2line -pfiaC -e path.elf 0x42007988 0x42007a4e
```

对应结果如下：

```
? riscv32-esp-elf-addr2line  -pfiaC -e ./build/idf_debug_method.elf 0x42007988␣
↪0x42007a4e
0x42007988: new_monkey_born at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:19
0x42007a4e: app_main at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:71
```

如果使用 IDF monitor 来打印日志，它会自动调用上述 addr2line 并打印解析后的结果，如下：

```
Guru Meditation Error: Core  0 panic'ed (Load access fault). Exception was␣
↪unhandled.

Stack dump detected
Core  0 register dump:
MEPC     : 0x42007988  RA       : 0x42007a4e  SP       : 0x3fc8fed0  GP       :␣
↪0x3fc8b200
0x42007988: new_monkey_born at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:19

0x42007a4e: app_main at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:71

TP       : 0x3fc870f8  T0       : 0x4005890e  T1       : 0x3fc8fb2c  T2       :␣
↪0x00000000
0x4005890e: memset in ROM

S0/FP    : 0x0000000a  S1       : 0x3fc90948  A0       : 0x00000000  A1       :␣
↪0x3fc8fb08
A2       : 0x00000000  A3       : 0x00000001  A4       : 0x00000000  A5       :␣
↪0x00000009
A6       : 0x60023000  A7       : 0x0000000a  S2       : 0x00000000  S3       :␣
↪0x00000000
S4       : 0x00000000  S5       : 0x00000000  S6       : 0x00000000  S7       :␣
↪0x00000000
S8       : 0x00000000  S9       : 0x00000000  S10      : 0x00000000  S11      :␣
↪0x00000000
T3       : 0x00000000  T4       : 0x00000000  T5       : 0x00000000  T6       :␣
↪0x00000000
MSTATUS : 0x00001881  MTVEC    : 0x40380001  MCAUSE   : 0x00000005  MTVAL    :␣
↪0x00000008
```

(下页继续)

(续上页)

```
0x40380001: _vector_table at ??:?

MHARTID : 0x00000000

Backtrace:

new_monkey_born (zoo=zoo@entry=0x0) at /home/libo/test_github/idf_debug_method/
↪main/idf_debug_method.c:19
19          zoo->monkey++;
#0  new_monkey_born (zoo=zoo@entry=0x0) at /home/libo/test_github/idf_debug_method/
↪main/idf_debug_method.c:19
#1  0x42007a4e in app_main () at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:70
#2  0x42015de4 in main_task (args=<error reading variable: value has been␣
↪optimized out>) at /home/libo/esp/github_master/components/freertos/app_startup.
↪c:208
#3  0x40384538 in vPortTaskWrapper (pxCode=<optimized out>, pvParameters=
↪<optimized out>) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/riscv/port.c:234
```

此时可以根据详细的 backtrace 日志、Guru Meditation 错误 和以下：

- MEPC：机器异常程序计数器（Machine Exception Program Counter）
- MTVAL：机器陷阱值（Machine Trap Value）

指针来进一步调试分析，Guru Meditation 错误的常见原因也可参考*利用 Guru Meditation 错误打印定位问题* 章节。

常见的 Backtrace 错误 常见的 Backtrace 错误如下表，具体细节也会在*利用 Guru Meditation 错误打印定位问题* 章节描述。

表 2: Xtensa and RISC-V Exception Mapping

Xtensa	RISC-V	Why	Where
IllegalInstruction	IllegalInstruction	SPI Flash IO Broken / task return without vTaskDelete / non-void function return void	Hardware or Software
•	Instruction Address Misaligned	not 2-byte aligned	PC(*0x4_*), MEPC(*0x3_-0x6_*)
InstrFetchProhibited	Instruction Access Fault	not in IRAM/RAM range	PC(*0x4_*), MEPC(*0x3_-0x6_*)
•	Memory Protection Fault	write to IRAM or execute from DRAM	MEPC(*0x3_-0x6_*)
LoadProhibited	Load Access Fault	Read from NULL/invalid pointer	EXCVADDR, MTVAL
StoreProhibited	Store Access Fault	Save to NULL/invalid pointer	EXCVADDR, MTVAL
LoadStoreAlignment	Load/Store Address Misaligned	IRAM use as DRAM, but not 4-byte aligned	EXCVADDR, MTVAL
IntegerDivideByZero	•	calculate n/0	PC(*0x4_*), MEPC(*0x3_-0x6_*)
LoadStoreError	•	write to read-only IROM DROM	EXCVADDR, MTVAL
Interrupt Watchdog Timeout on CPU0/CPU1	Interrupt Watchdog Timeout on CPU0/CPU1	Interrupt timeout/ disabled	•
Cache disabled but cached memory region accessed	Cache error	set FLAG_IRAM but not all data/functions in IRAM/DRAM	PC(*0x4_*), MEPC(*0x3_-0x6_*)
Brownout	Brownout	Supply voltage lower than threshold	Hardware
rst:0x10 (RTCWDT_RTC_RESET)	rst:0x10 (RTCWDT_RTC_RESET)	no valid program in flash	Hardware
rst:0x7 (TG0WDT_SYS_RST)	rst:0x7 (TG0WDT_SYS_RST)	Watchdog timeout	Flash or PSRAM Pin / software
Corrupt Heap	Corrupt Heap	Heap corruption detected	Need further analysis
•	Stack protection fault	Stack overflow	Need further analysis
Stack smashing protect failure	Stack smashing protect failure	Stack overflow	Need further analysis
Core 0 panic' ed Exception was unhandled	Stack canary watchpoint triggered (task_name)	Stack overflow	Need further analysis
UBSAN	UBSAN	Undefined behavior	Need further analysis

Core Dump 简述 这部分可参考 Core Dump 文档。简要总结如下：

- Core Dump 比 backtrace 更加丰富，包括每个 task 的 stack dump
- Core Dump 可以打印到终端，也可以保存到 flash （type: data, subtype: coredump）
- Core Dump 保存到 flash 以后，可以直接通过指令 *idf.py coredump-info* 分析
- Core Dump 串口数据为 *BASE64* 格式二进制字符串，复制并保存到 text 后，调用 *idf.py coredump-info -c </path/to/saved/base64/text>* 指令来进行分析
- *esp monitor* 可以解析 Core Dump 信息，显示为可读格式
- *idf.py coredump-debug* 可以 "还原现场"，GDB debug 将被打开，可以用于查看变量值

Xtensa Core Dump 一般如下：

```
Initiating core dump!
I (423) esp_core_dump_uart: Press Enter to print core dump to UART...
I (431) esp_core_dump_uart: Print core dump to uart...
Core dump started (further output muted)
Received  13 kB...
Core dump finished!
===============================================================
==================== ESP32 CORE DUMP START ====================
The ROM ELF file won't load automatically since it was not found for the provided␣
↪chip type.

Crashed task handle: 0x3fc9a790, name: 'main', GDB name: 'process 1070180240'

================== CURRENT THREAD REGISTERS ===================
exccause       0x1c (LoadProhibitedCause)
excvaddr       0x8
epc1           0x40378347
epc2           0x0
epc3           0x0
epc4           0x0
epc5           0x0
epc6           0x0
eps2           0x0
eps3           0x0
eps4           0x0
eps5           0x0
eps6           0x0
pc             0x42018677          0x42018677 <new_monkey_born+7>
lbeg           0x400556d5          1074091733
lend           0x400556e5          1074091749
lcount         0xfffffffc          4294967292
sar            0x1a                26
ps             0x60c20             396320
threadptr      <unavailable>
br             <unavailable>
scompare1      <unavailable>
acclo          <unavailable>
acchi          <unavailable>
m0             <unavailable>
m1             <unavailable>
m2             <unavailable>
m3             <unavailable>
expstate       <unavailable>
f64r_lo        <unavailable>
f64r_hi        <unavailable>
f64s           <unavailable>
fcr            <unavailable>
fsr            <unavailable>
a0             0x8200a136          -2113887946
a1             0x3fc9a5a0          1070179744
a2             0x0                 0
a3             0x3fc9a7ec          1070180332
a4             0x3c023088          1006776456
a5             0x3fc9a5d0          1070179792
a6             0x3fc9a5b0          1070179760
a7             0xc                 12
a8             0x8200e2b8          -2113871176
a9             0x3fc9a250          1070178896
a10            0x10                16
a11            0xffffffff          -1
a12            0x10                16
a13            0x3fc9a460          1070179424
```

(下页继续)

(续上页)

```
a14            0x3fc9a260          1070178912
a15            0x1                 1

==================== CURRENT THREAD STACK =====================
#0  new_monkey_born (zoo=0x0) at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:21
#1  0x4200a136 in app_main () at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:57
#2  0x42019ade in main_task (args=<optimized out>) at /home/libo/esp/github_master/
↪components/freertos/app_startup.c:208
#3  0x4037a2ec in vPortTaskWrapper (pxCode=0x42019a38 <main_task>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162

======================== THREADS INFO =========================
Id   Target Id          Frame
* 1    process 1070180240 new_monkey_born (zoo=0x0) at /home/libo/test_github/idf_
↪debug_method/main/idf_debug_method.c:21
2    process 1070182124 vPortTaskWrapper (pxCode=0x0, pvParameters=0x0) at /home/
↪libo/esp/github_master/components/freertos/FreeRTOS-Kernel/portable/xtensa/port.
↪c:161
3    process 1070184008 0x40378326 in esp_cpu_wait_for_intr () at /home/libo/esp/
↪github_master/components/esp_hw_support/cpu.c:145
4    process 1070169660 0x400559e0 in ?? ()
5    process 1070175732 0x400559e0 in ?? ()
6    process 1070171288 0x400559e0 in ?? ()

==================== THREAD 1 (TCB: 0x3fc9a790, name: 'main') =====================
#0  new_monkey_born (zoo=0x0) at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:21
#1  0x4200a136 in app_main () at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:57
#2  0x42019ade in main_task (args=<optimized out>) at /home/libo/esp/github_master/
↪components/freertos/app_startup.c:208
#3  0x4037a2ec in vPortTaskWrapper (pxCode=0x42019a38 <main_task>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162

==================== THREAD 2 (TCB: 0x3fc9aeec, name: 'IDLE0')␣
↪=====================
#0  vPortTaskWrapper (pxCode=0x0, pvParameters=0x0) at /home/libo/esp/github_
↪master/components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:161
#1  0x40000000 in ?? ()

==================== THREAD 3 (TCB: 0x3fc9b648, name: 'IDLE1')␣
↪=====================
#0  0x40378326 in esp_cpu_wait_for_intr () at /home/libo/esp/github_master/
↪components/esp_hw_support/cpu.c:145
#1  0x42002be5 in esp_vApplicationIdleHook () at /home/libo/esp/github_master/
↪components/esp_system/freertos_hooks.c:59
#2  0x4037b038 in prvIdleTask (pvParameters=0x0) at /home/libo/esp/github_master/
↪components/freertos/FreeRTOS-Kernel/tasks.c:4170
#3  0x4037a2ec in vPortTaskWrapper (pxCode=0x4037b02c <prvIdleTask>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162

==================== THREAD 4 (TCB: 0x3fc97e3c, name: 'ipc0') =====================
#0  0x400559e0 in ?? ()
#1  0x4037a692 in vPortClearInterruptMaskFromISR (prev_level=<optimized out>) at /
↪home/libo/esp/github_master/components/freertos/FreeRTOS-Kernel/portable/xtensa/
↪include/freertos/portmacro.h:582
```

(下页继续)

(续上页)

```
#2  vPortExitCritical (mux=<optimized out>) at /home/libo/esp/github_master/
↪components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:532
#3  0x4037c0c5 in xTaskGenericNotifyWait (uxIndexToWait=0, ulBitsToClearOnEntry=
↪<optimized out>, ulBitsToClearOnExit=4294967295, pulNotificationValue=0x3fc97ca0,
↪ xTicksToWait=4294967295) at /home/libo/esp/github_master/components/freertos/
↪FreeRTOS-Kernel/tasks.c:5644
#4  0x40378104 in ipc_task (arg=0x0) at /home/libo/esp/github_master/components/
↪esp_system/esp_ipc.c:58
#5  0x4037a2ec in vPortTaskWrapper (pxCode=0x403780d4 <ipc_task>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162

==================== THREAD 5 (TCB: 0x3fc995f4, name: 'esp_timer')␣
↪======================
#0  0x400559e0 in ?? ()
#1  0x4037a692 in vPortClearInterruptMaskFromISR (prev_level=<optimized out>) at /
↪home/libo/esp/github_master/components/freertos/FreeRTOS-Kernel/portable/xtensa/
↪include/freertos/portmacro.h:582
#2  vPortExitCritical (mux=<optimized out>) at /home/libo/esp/github_master/
↪components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:532
#3  0x4037bf89 in ulTaskGenericNotifyTake (uxIndexToWait=0, xClearCountOnExit=1,␣
↪xTicksToWait=<optimized out>) at /home/libo/esp/github_master/components/
↪freertos/FreeRTOS-Kernel/tasks.c:5565
#4  0x42006143 in timer_task (arg=0x0) at /home/libo/esp/github_master/components/
↪esp_timer/src/esp_timer.c:477
#5  0x4037a2ec in vPortTaskWrapper (pxCode=0x42006134 <timer_task>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162

==================== THREAD 6 (TCB: 0x3fc98498, name: 'ipc1') =====================
#0  0x400559e0 in ?? ()
#1  0x4037a692 in vPortClearInterruptMaskFromISR (prev_level=<optimized out>) at /
↪home/libo/esp/github_master/components/freertos/FreeRTOS-Kernel/portable/xtensa/
↪include/freertos/portmacro.h:582
#2  vPortExitCritical (mux=<optimized out>) at /home/libo/esp/github_master/
↪components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:532
#3  0x4037c0c5 in xTaskGenericNotifyWait (uxIndexToWait=0, ulBitsToClearOnEntry=
↪<optimized out>, ulBitsToClearOnExit=4294967295, pulNotificationValue=0x3fc98300,
↪ xTicksToWait=4294967295) at /home/libo/esp/github_master/components/freertos/
↪FreeRTOS-Kernel/tasks.c:5644
#4  0x40378104 in ipc_task (arg=0x1) at /home/libo/esp/github_master/components/
↪esp_system/esp_ipc.c:58
#5  0x4037a2ec in vPortTaskWrapper (pxCode=0x403780d4 <ipc_task>,␣
↪pvParameters=0x1) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/xtensa/port.c:162


======================= ALL MEMORY REGIONS ========================
Name   Address   Size   Attrs
.rtc.force_fast 0x600fe010 0x0 RW
.rtc_noinit 0x50000000 0x0 RW
.rtc.force_slow 0x50000000 0x0 RW
.iram0.vectors 0x40374000 0x403 R XA
.iram0.text 0x40374404 0xd56f R XA
.dram0.data 0x3fc91a00 0x3e84 RW A
.flash.text 0x42000020 0x1a41b R XA
.flash.appdesc 0x3c020020 0x100 R  A
.flash.rodata 0x3c020120 0xae5c RW A
.flash.rodata_noload 0x3c02af7c 0x0 RW
.ext_ram.bss 0x3c030000 0x0 RW
.iram0.data 0x40381a00 0x0 RW
```

(下页继续)

(续上页)

```
.iram0.bss 0x40381a00 0x0 RW
.dram0.heap_start 0x3fc96a28 0x0 RW
.coredump.tasks.data 0x3fc9a790 0x154 RW
.coredump.tasks.data 0x3fc9a4e0 0x2a0 RW
.coredump.tasks.data 0x3fc9aeec 0x154 RW
.coredump.tasks.data 0x3fc9ace0 0x200 RW
.coredump.tasks.data 0x3fc9b648 0x154 RW
.coredump.tasks.data 0x3fc9b3c0 0x280 RW
.coredump.tasks.data 0x3fc97e3c 0x154 RW
.coredump.tasks.data 0x3fc97b90 0x2a0 RW
.coredump.tasks.data 0x3fc995f4 0x154 RW
.coredump.tasks.data 0x3fc99350 0x290 RW
.coredump.tasks.data 0x3fc98498 0x154 RW
.coredump.tasks.data 0x3fc981f0 0x2a0 RW

===================== ESP32 CORE DUMP END =====================
===============================================================
Done!
Coredump checksum='7b5945fe'
I (1690) esp_core_dump_uart: Core dump has been written to uart.
```

RISC-V Core Dump 一般如下：

```
Initiating core dump!
I (696) esp_core_dump_uart: Press Enter to print core dump to UART...
I (703) esp_core_dump_uart: Print core dump to uart...
Core dump started (further output muted)
Received   3 kB...
Core dump finished!
===============================================================
==================== ESP32 CORE DUMP START ====================

Crashed task handle: 0x3fc91594, name: 'main', GDB name: 'process 1070142868'

================== CURRENT THREAD REGISTERS ===================
ra             0x42009404       0x42009404 <app_main+56>
sp             0x3fc91510       0x3fc91510
gp             0x3fc8b400       0x3fc8b400 <__c.29+52>
tp             0x3fc88368       0x3fc88368
t0             0x4005890e       1074104590
t1             0x3fc9116c       1070141804
t2             0x0      0
fp             0x3c022000       0x3c022000
s1             0x3fc91f98       1070145432
a0             0x0      0
a1             0x3fc91148       1070141768
a2             0x0      0
a3             0x1      1
a4             0x0      0
a5             0x0      0
a6             0x60023000       1610756096
a7             0xa      10
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
s8             0x0      0
s9             0x0      0
s10            0x0      0
```

(下页继续)

(续上页)

```
s11            0x0      0
t3             0x0      0
t4             0x0      0
t5             0x0      0
t6             0x0      0
pc             0x4200939e       0x4200939e <new_monkey_born+4>

==================== CURRENT THREAD STACK =====================
#0  new_monkey_born (zoo=zoo@entry=0x0) at /home/libo/test_github/idf_debug_method/
↪main/idf_debug_method.c:21
#1  0x42009404 in app_main () at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:55
#2  0x42017976 in main_task (args=<error reading variable: value has been␣
↪optimized out>) at /home/libo/esp/github_master/components/freertos/app_startup.
↪c:208
#3  0x403845bc in vPortTaskWrapper (pxCode=<optimized out>, pvParameters=
↪<optimized out>) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/riscv/port.c:234

======================== THREADS INFO =========================
Id   Target Id          Frame
* 1    process 1070142868 new_monkey_born (zoo=zoo@entry=0x0) at /home/libo/test_
↪github/idf_debug_method/main/idf_debug_method.c:21
2    process 1070144752 vPortTaskWrapper (pxCode=0x40384f20 <prvIdleTask>,␣
↪pvParameters=0x0) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/riscv/port.c:230
3    process 1070138360 0x40384796 in vPortClearInterruptMaskFromISR (prev_int_
↪level=1) at /home/libo/esp/github_master/components/freertos/FreeRTOS-Kernel/
↪portable/riscv/port.c:417

==================== THREAD 1 (TCB: 0x3fc91594, name: 'main') =====================
#0  new_monkey_born (zoo=zoo@entry=0x0) at /home/libo/test_github/idf_debug_method/
↪main/idf_debug_method.c:21
#1  0x42009404 in app_main () at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:55
#2  0x42017976 in main_task (args=<error reading variable: value has been␣
↪optimized out>) at /home/libo/esp/github_master/components/freertos/app_startup.
↪c:208
#3  0x403845bc in vPortTaskWrapper (pxCode=<optimized out>, pvParameters=
↪<optimized out>) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/riscv/port.c:234

==================== THREAD 2 (TCB: 0x3fc91cf0, name: 'IDLE') =====================
#0  vPortTaskWrapper (pxCode=0x40384f20 <prvIdleTask>, pvParameters=0x0) at /home/
↪libo/esp/github_master/components/freertos/FreeRTOS-Kernel/portable/riscv/port.
↪c:230
#1  0x00000000 in ?? ()
Backtrace stopped: frame did not save the PC

==================== THREAD 3 (TCB: 0x3fc903f8, name: 'esp_timer')␣
↪=====================
#0  0x40384796 in vPortClearInterruptMaskFromISR (prev_int_level=1) at /home/libo/
↪esp/github_master/components/freertos/FreeRTOS-Kernel/portable/riscv/port.c:417
#1  0x403847f8 in vPortExitCritical () at /home/libo/esp/github_master/components/
↪freertos/FreeRTOS-Kernel/portable/riscv/port.c:517
#2  0x40385cfa in ulTaskGenericNotifyTake (uxIndexToWait=uxIndexToWait@entry=0,␣
↪xClearCountOnExit=xClearCountOnExit@entry=1,␣
↪xTicksToWait=xTicksToWait@entry=4294967295) at /home/libo/esp/github_master/
↪components/freertos/FreeRTOS-Kernel/tasks.c:5565
#3  0x420046b6 in timer_task (arg=<error reading variable: value has been␣
↪optimized out>) at /home/libo/esp/github_master/components/esp_timer/src/esp_
↪timer.c:477
```

(下页继续)

(续上页)

```
#4  0x403845bc in vPortTaskWrapper (pxCode=<optimized out>, pvParameters=
↪<optimized out>) at /home/libo/esp/github_master/components/freertos/FreeRTOS-
↪Kernel/portable/riscv/port.c:234


======================= ALL MEMORY REGIONS ========================
Name   Address   Size   Attrs
.rtc.force_fast 0x50000010 0x0 RW
.rtc_noinit 0x50000010 0x0 RW
.rtc.force_slow 0x50000010 0x0 RW
.iram0.text 0x40380000 0xab8a R XA
.dram0.data 0x3fc8ac00 0x21c0 RW A
.flash.text 0x42000020 0x18504 R XA
.flash.appdesc 0x3c020020 0x100 R  A
.flash.rodata 0x3c020120 0x9218 RW A
.eh_frame 0x3c029338 0xe0 R  A
.flash.rodata_noload 0x3c029418 0x0 RW
.iram0.data 0x4038ac00 0x0 RW
.iram0.bss 0x4038ac00 0x0 RW
.dram0.heap_start 0x3fc8e490 0x0 RW
.coredump.tasks.data 0x3fc91594 0x154 RW
.coredump.tasks.data 0x3fc91470 0x110 RW
.coredump.tasks.data 0x3fc91cf0 0x154 RW
.coredump.tasks.data 0x3fc91c40 0xa0 RW
.coredump.tasks.data 0x3fc903f8 0x154 RW
.coredump.tasks.data 0x3fc90310 0xe0 RW

===================== ESP32 CORE DUMP END =====================
===============================================================
Done!
Coredump checksum='704e2165'
I (1064) esp_core_dump_uart: Core dump has been written to uart.
```

可以看到 Core Dump 里有各类详细信息，有些信息已在 backtrace 里包含，如当前 Thread/Task 寄存器，当前 Thread 信息。额外需要 Core dump 信息的原因往往是它打印了当前异常时运行的所有任务的状态，有时对解决一些如任务看门狗问题时会提供帮助。

定位内存问题（踩内存、内存泄漏等）

常见的内存问题有以下几种：

- 堆内存泄漏
- 堆内存覆盖
- 栈内存溢出
- 栈内存覆盖
- 指针在释放后继续使用
- 指针在初始化之前被使用
- 双重释放

本节将逐一说明上述内存问题的通用调试方法。

堆内存泄漏 堆内存泄露的表现形式往往为程序分配了一块堆内存，但在使用完毕后未正确释放，导致内存泄漏。这会导致程序运行时消耗的内存逐渐增加，最终可能耗尽系统的可用内存。

这部分可以参考 堆内存调试文档，要点整理如下：

- 可以使用 *heap_caps_get_per_task_info* 获得所有任务的内存申请情况
- 可以使用 *heap_caps_get_free_size* 对比剩余内存情况，大致确定泄露区域

- 使能 *CONFIG_HEAP_TRACING_STANDALONE* 或 *CONFIG_HEAP_TRACING_TOHOST*
- STANDALONE 模式需要分配 buffer，直接在 ESP 上记录、计算、打印结果，但 RISC-V 架构无法定位代码行
- TOHOST 需要 UART/JTAG 使用 app_trace 抓取，在主机上分析，无需额外 buffer，可以定位代码行
- *heap_trace_init_standalone* 初始化 buffer，*heap_trace_start(HEAP_TRACE_LEAKS)* 开始记录
- *heap_trace_stop()* 停止记录，使用 *heap_trace_dump()* 打印分析结果

使用上述堆内存调试方法后的典型日志如下：

1. Xtensa

```
====== Heap Trace: 2 records (100 capacity) ======
36 bytes (@ 0x3fc9c524, Internal) allocated CPU 0 ccount 0x02f204e0 caller␣
↪0x42008cfd:0x42008d73
0x42008cfd: zoo_create at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:68

0x42008d73: mem_leak_task at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:96 (discriminator 3)

    24 bytes (@ 0x3fc9c54c, Internal) allocated CPU 0 ccount 0x02f20c00 caller␣
↪0x42008cfd:0x42008d73
0x42008cfd: zoo_create at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:68

0x42008d73: mem_leak_task at /home/libo/test_github/idf_debug_method/main/idf_
↪debug_method.c:96 (discriminator 3)

====== Heap Trace Summary ======
Mode: Heap Trace Leaks
60 bytes 'leaked' in trace (2 allocations)
records: 2 (100 capacity, 3 high water mark)
total allocations: 3
total frees: 1
================================
```

2. RISC-V

```
====== Heap Trace: 3 records (100 capacity) ======
36 bytes (@ 0x3fc91574, Internal) allocated CPU 0 ccount 0x02cf6d38 caller
36 bytes (@ 0x3fc9159c, Internal) allocated CPU 0 ccount 0x02cf72cc caller
====== Heap Trace Summary ======
Mode: Heap Trace Leaks
72 bytes 'leaked' in trace (2 allocations)
records: 2 (100 capacity, 3 high water mark)
total allocations: 3
total frees: 1
================================
```

堆内存覆盖 堆内存覆盖的表现形式往往为在写入或读取堆内存时，程序访问了超出分配给它的内存范围的区域。这可能导致未定义的行为，破坏了程序的内存结构。该错误对应的日志往往为：

```
assert failed: remove_free_block tlsf.c:331 (next && "next_free field can not be␣
↪null")
```

这部分可以参考 堆内存调试文档，要点整理如下：

1. 定位 Who and Where

- 使能内存调试，将堆内存调试等级 CONFIG_HEAP_CORRUPTION_DETECTION 提高到 *light impact* 或者 *comprehensive*：
- Basic（默认）：使用堆属性检测是否被污染

- Light impact : 给分配的内存前后加上头尾特殊字节 *0xABBA1234 0xBAAD5678*
- Comprehensive : 在 light impact 的基础上增加 uninitialized-access 和 use-after-free bugs 检查。内存申请时，所有内存初始化为 0xce, 内存释放后所有空间赋值为 0xfe

- 开启内存调试以后，等待 crash 或在怀疑踩内存的位置前后主动调用检查内存完整性 *heap_caps_check_integrity_all* 触发 crash。如果已经定位到踩内存的地址，可以直接使用 *heap_caps_check_integrity_addr*。
  - 踩尾巴，当前内存块操作越界 *CORRUPT HEAP: Bad tail at 0x3fc9ad5a. Expected 0xbaad5678 got 0x02020202*
  - 踩头，上一个内存块越界 *CORRUPT HEAP: Bad head at 0x3fc9a94c. Expected 0xabba1234 got 0x00000000*
- 两种方法可以确认内存块前后邻居
  - 使用 heap trace, 调用 *heap_trace_start(HEAP_TRACE_ALL)* 收集信息
  - 使用 *heap_caps_dump_all* 打印收集到的信息 (需要在内存申请之后、踩之前打印)

备注: 上述确认内存块状态的具体方式描述可参考 堆内存跟踪。

---

2. 定位 When

- 可以在代码里通过 *esp_cpu_set_watchpoint(0, (void *)0x3fc9a94c, 4, ESP_CPU_WATCHPOINT_STORE);* 设置 CPU 断点。如果不知道哪个内核，需要两个内核都调用一遍
- CPU 将在该地址写入数据时触发断点，通过 PC 可以定位到代码行，参考日志如下：

```
Guru Meditation Error: Core  0 panic'ed (Unhandled debug exception).
Debug exception reason: Watchpoint 0 triggered
Core  0 register dump:
PC      : 0x400570e8  PS      : 0x00060c36  A0      : 0x82008d43  A1      :␣
↪0x3fc99f10
0x400570e8: memset in ROM

A2      : 0x3fc9b3ac  A3      : 0x00000000  A4      : 0x000003e8  A5      :␣
↪0x3fc9b75c
A6      : 0x00000000  A7      : 0x0000003e  A8      : 0x8200333d  A9      :␣
↪0x3fc99ee0
A10     : 0x00000400  A11     : 0x00060c20  A12     : 0x00000000  A13     :␣
↪0x00060c23
A14     : 0xb33fffff  A15     : 0xb33fffff  SAR     : 0x00000004  EXCCAUSE:␣
↪0x00000001
EXCVADDR: 0x00000000  LBEG    : 0x400570e8  LEND    : 0x400570f3  LCOUNT  :␣
↪0x00000002

Backtrace: 0x400570e5:0x3fc99f10 0x42008d40:0x3fc99f20 0x4201874b:0x3fc99f50␣
↪0x4037a80d:0x3fc99f80
0x400570e5: memset in ROM
0x42008d40: app_main at /home/libo/test_github/idf_debug_method/main/idf_debug_
↪method.c:169 (discriminator 3)
0x4201874b: main_task at /home/libo/esp/github_master/components/freertos/app_
↪startup.c:208 (discriminator 13)
0x4037a80d: vPortTaskWrapper at /home/libo/esp/github_master/components/
↪freertos/FreeRTOS-Kernel/portable/xtensa/port.c:162
```

栈内存溢出 栈内存溢出的表现形式往往为在函数调用过程中使用栈内存时，如果递归调用或局部变量过多，栈的大小可能会超过系统允许的限制，导致栈内存溢出。以下是 ESP-IDF 里支持的栈内存溢出检测机制：

1. ESP-IDF FreeRTOS 默认开启栈溢出检测，如果检测到栈溢出，会触发断言，打印对应栈溢出信息，典型日志如下：

```
***ERROR*** A stack overflow in task test_task has been detected.
```

更多细节可以参考 栈溢出 章节。

2. ESP-IDF 支持开启 End of Stack Watchpoint，在 FreeRTOS 栈溢出触发断言之前，触发断点。
3. RISC-V 平台支持开启硬件栈溢出检测（*Stack protection fault*），具体可参考 硬件堆栈保护。

栈内存覆盖 栈内存覆盖的表现形式往往类似于堆内存覆盖，但发生在程序使用栈内存时。写入或读取超出栈分配的内存范围的数据可能导致程序错误。以下是几个注意点：

1. 可能导致任务堆栈溢出，一般可通过 FreeRTOS 的栈溢出机制检测到。
2. 可能导致局部变量值被覆盖，导致程序不符合预期。
3. 可能导致局部指针变量被修改，访问非法指令/数据地址，导致程序崩溃。
4. 可能导致函数返回地址被覆盖，程序跳转到错误地址，导致程序崩溃。

简单的错误代码示例如下：

```
int vulnerableFunction() {
    int localArray[5];  // Array allocated on the stack

    // Writing beyond the bounds of the array
    for (int i = 0; i <= 5; ++i) {
        localArray[i] = i;
    }

    return localArray[0];
}

void app_main() {
    printf("Before vulnerable function.\n");

    vulnerableFunction();  // Call the function that causes stack memory corruption

    printf("After vulnerable function.\n");
}
```

值得一提的是，ESP-IDF 在编译时就会检查到部分此类型错误并给出警告（但仍能编译通过），编译时的警告日志如下：

```
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c: In function 'vulnerableFunction':
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c:20:23: warning: iteration 5 invokes undefined behavior [-
↪Waggressive-loop-optimizations]
  20 |         localArray[i] = i;
     |          ~~~~~~~~~~~~~~^~~
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c:19:23: note: within this loop
  19 |     for (int i = 0; i <= 5; ++i) {
```

指针在释放后继续使用 指针在释放后继续使用的表现形式往往为程序释放了一块内存，但后续仍然使用了指向该内存的指针。这可能导致访问无效内存，引发崩溃或未定义的行为。

此问题可能会导致各种错误，很难通过实际错误来定位到为此问题，因此在开发过程中需要特别注意指针的使用。简单的错误代码示例如下：

```
void app_main(void)
{
  int *number = (int *)malloc(sizeof(int));  // Allocate memory for an integer

  if (number == NULL) {
      // Handle memory allocation failure
      printf("Memory allocation failed.\n");
```

(下页继续)

(续上页)

```
  }

  *number = 42;  // Assign a value to the allocated memory

  printf("Value before freeing: %d\n", *number);

  free(number);  // Free the allocated memory

  // Attempt to use the pointer after freeing
  // This will result in undefined behavior
  printf("Value after freeing: %d\n", *number);
}
```

值得一提的是，ESP-IDF 在编译时就会检查到部分此类型错误并给出警告（但仍能编译通过），编译时的警告日志如下：

```
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c: In function 'app_main':
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c:32:5: warning: pointer 'number' used after 'free' [-Wuse-
↪after-free]
  32 |     printf("Value after freeing: %d\n", *number);
     |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/home/zhengzhong/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/
↪main/hello_world_main.c:28:5: note: call to 'free' here
  28 |     free(number);  // Free the allocated memory
```

指针在初始化之前被使用　指针在初始化之前被使用的表现形式往往为当程序尝试使用尚未初始化的指针时，可能会访问未知的内存区域，导致不稳定的行为。

此问题可能会导致各种错误，很难通过实际错误来定位到为此问题，因此在开发过程中需要特别注意指针的使用。简单的错误代码示例如下：

```
void app_main(void)
{
    int *number;  // Pointer declared but not initialized

    // Attempt to dereference the uninitialized pointer
    // This will result in undefined behavior
    printf("Value: %d\n", *number);

}
```

值得一提的是，ESP-IDF 往往在编译时就会检查到部分此类型错误并给出报错提示，编译时的错误日志如下：

```
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c: In function 'app_main':
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c:21:5: error: 'number' is used uninitialized [-
↪Werror=uninitialized]
  21 |     printf("Value: %d\n", *number);
     |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/home/user/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/
↪hello_world_main.c:17:10: note: 'number' was declared here
  17 |     int *number;  // Pointer declared but not initialized
     |          ^~~~~~
cc1: some warnings being treated as
```

双重释放 双重释放的表现形式往往为程序释放了已经被释放的内存，这可能导致内存池的破坏，进而导致程序崩溃或产生其他严重问题。错误代码示例如下：

```
void app_main(void)
{
    // Allocate a block of memory
    int *data = (int *)malloc(sizeof(int));

    // Check if memory allocation is successful
    if (data != NULL) {
        // Assign a value to the allocated memory
        *data = 42;

        // First free
        free(data); // Line 26

        // Second free (double-free)
        free(data);  // Line 29, this is incorrect and may lead to undefined␣
↪behavior
    }
}
```

值得一提的是，ESP-IDF 往往在编译时就会检查到部分此类型错误并给出报错提示，编译时的警告日志如下：

```
/home/zhengzhong/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/
↪main/hello_world_main.c: In function 'app_main':
/home/zhengzhong/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/
↪main/hello_world_main.c:29:9: warning: pointer 'data' used after 'free' [-Wuse-
↪after-free]
  29 |         free(data);  // This is incorrect and may lead to undefined behavior
     |         ^~~~~~~~~~
/home/zhengzhong/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/
↪main/hello_world_main.c:26:9: note: call to 'free' here
  26 |         free(data);
     |         ^~~~~~~~~~
```

运行时的错误日志如下：

```
I (285) main_task: Calling app_main()

assert failed: tlsf_free tlsf.c:1119 (!block_is_free(block) && "block already␣
↪marked as free")
Core  0 register dump:
Stack dump detected
MEPC    : 0x403805d8  RA      : 0x403838e8  SP      : 0x3fc8f330  GP      :␣
↪0x3fc8ae00
0x403805d8: panic_abort at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/
↪components/esp_system/panic.c:452

0x403838e8: __ubsan_include at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/
↪components/esp_system/ubsan.c:313

TP      : 0x3fc87110  T0      : 0x37363534  T1      : 0x7271706f  T2      :␣
↪0x33323130
S0/FP   : 0x00000069  S1      : 0x00000001  A0      : 0x3fc8f36c  A1      :␣
↪0x3fc8acd1
A2      : 0x00000001  A3      : 0x00000029  A4      : 0x00000001  A5      :␣
↪0x3fc8c000
A6      : 0x7a797877  A7      : 0x76757473  S2      : 0x00000009  S3      :␣
↪0x3fc8f49e
S4      : 0x3fc8acd0  S5      : 0x00000000  S6      : 0x00000000  S7      :␣
↪0x00000000
```

(下页继续)

(续上页)

```
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     :␣
↪0x00000000
T3      : 0x6e6d6c6b  T4      : 0x6a696867  T5      : 0x66656463  T6      :␣
↪0x62613938
MSTATUS : 0x00001881  MTVEC   : 0x40380001  MCAUSE  : 0x00000007  MTVAL   :␣
↪0x00000000
0x40380001: _vector_table at ??:?

MHARTID : 0x00000000


Backtrace:


panic_abort (details=details@entry=0x3fc8f36c "assert failed: tlsf_free tlsf.
↪c:1119 (!block_is_free(block) && \"block already marked as free\")") at /home/
↪zhengzhong/github/esp-idf/rel5.1/esp-idf/components/esp_system/panic.c:452
452         *((volatile int *) 0) = 0; // NOLINT(clang-analyzer-core.
↪NullDereference) should be an invalid operation on targets
#0  panic_abort (details=details@entry=0x3fc8f36c "assert failed: tlsf_free tlsf.
↪c:1119 (!block_is_free(block) && \"block already marked as free\")") at /home/
↪zhengzhong/github/esp-idf/rel5.1/esp-idf/components/esp_system/panic.c:452
#1  0x403838e8 in esp_system_abort (details=details@entry=0x3fc8f36c "assert␣
↪failed: tlsf_free tlsf.c:1119 (!block_is_free(block) && \"block already marked␣
↪as free\")") at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/components/esp_
↪system/port/esp_system_chip.c:84
#2  0x403890e8 in __assert_func (file=file@entry=0x3c0212f3 "",␣
↪line=line@entry=1119, func=<optimized out>, func@entry=0x3c021984 <__func__.6> "
↪", expr=expr@entry=0x3c0217ec "") at /home/zhengzhong/github/esp-idf/rel5.1/esp-
↪idf/components/newlib/assert.c:81
#3  0x40387e5e in tlsf_free (tlsf=0x3fc8c574, ptr=ptr@entry=0x3fc8ff20) at /home/
↪zhengzhong/github/esp-idf/rel5.1/esp-idf/components/heap/tlsf/tlsf.c:1119
#4  0x40387a8e in multi_heap_free_impl (heap=0x3fc8c560, p=p@entry=0x3fc8ff20) at /
↪home/zhengzhong/github/esp-idf/rel5.1/esp-idf/components/heap/multi_heap.c:231
#5  0x40380b98 in heap_caps_free (ptr=ptr@entry=0x3fc8ff20) at /home/zhengzhong/
↪github/esp-idf/rel5.1/esp-idf/components/heap/heap_caps.c:388
#6  0x4038910e in free (ptr=ptr@entry=0x3fc8ff20) at /home/zhengzhong/github/esp-
↪idf/rel5.1/esp-idf/components/newlib/heap.c:39
#7  0x4200712a in app_main () at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/
↪examples/get-started/hello_world/main/hello_world_main.c:29
#8  0x4201498a in main_task (args=<error reading variable: value has been␣
↪optimized out>) at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/components/
↪freertos/app_startup.c:208
#9  0x40385a2c in vPortTaskWrapper (pxCode=<optimized out>, pvParameters=
↪<optimized out>) at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/components/
↪freertos/FreeRTOS-Kernel/portable/riscv/port.c:202
ELF file SHA256: 1df25094bc6834da
```

可以看到 log 有 *block already marked as free* 提示，以及错误代码定位提示 *app_main () at /home/zhengzhong/github/esp-idf/rel5.1/esp-idf/examples/get-started/hello_world/main/hello_world_main.c:29* 发现在代码 29 行出现了双重释放的情况，需要删除第二次的 free。

使用 SystemView 进行系统分析和调优

这部分可直接参考 ESP-IDF 编程指南里的 应用层跟踪库 章节。

2.2.3 性能优化

这一部分介绍了 ESP 芯片常见的性能优化方式。

ESP 芯片启动时间优化

引言 ESP 芯片启动时间指的是 ESP 芯片从上电到执行 app_main 函数所花费的时间。未经优化的启动流程通常耗时较长（一般在 300 毫秒左右），导致无法满足即时性要求较高的应用场景（比如灯的随时开关）。同时伴随着较高的功耗，进而在 Deep-sleep 低功耗模式下的平均功耗居高不下。为了避免这些问题，必须对启动流程进行精细优化。

硬软件环境

- 硬件：ESP32-S3 和 ESP32-C6
- 软件：ESP-IDF v5.2.1

可选的优化项 通过 *menuconfig* 配置以下选项，即可大幅度减小启动时间：

1. **关闭 Bootloader 日志打印**
   - *CONFIG_BOOTLOADER_LOG_LEVEL_NONE=y*
   - *CONFIG_BOOTLOADER_LOG_LEVEL=0*
2. **跳过 Image 验证**
   - *CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP=y*
   - *CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON=y*
   - *CONFIG_BOOTLOADER_SKIP_VALIDATE_ALWAYS=y*
3. **关闭 Boot ROM 日志打印**
   - *CONFIG_BOOT_ROM_LOG_ALWAYS_OFF=y*

   除了这个配置外，还需要在终端中使用 *espefuse.py* 命令配置相关控制 ROM 日志输出的 eFuse 值（详见 技术参考手册 的 Boot 日志打印控制章节，注：写 eFuse 操作不可逆），具体命令如下：

   ```
   espefuse.py burn_efuse UART_PRINT_CONTROL 3
   espefuse.py burn_efuse DIS_USB_SERIAL_JTAG_ROM_PRINT 1
   ```

   运行以上两个命令后可使用 *espefuse.py summary* 命令进行检查，查看是否写入成功，若看到有如下信息，则表示配置成功：

   ```
   UART_PRINT_CONTROL (BLOCK0)                 Set the default UART boot message output mode       = Disable R/W (0b11)
   ```
4. **修改 SPI Flash 模式和频率**
   - *CONFIG_ESPTOOLPY_FLASHMODE_QIO=y*
   - *CONFIG_ESPTOOLPY_FLASHFREQ_80M=y*
5. **关闭启动校准**
   - *CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE=y*
   - *CONFIG_ESP_PHY_CALIBRATION_MODE=1*
6. **修改 FreeRTOS 配置**
   - *CONFIG_FREERTOS_UNICORE=y*
   - *CONFIG_FREERTOS_HZ=1000*
7. **关闭 Log 打印**
   - *CONFIG_LOG_DEFAULT_LEVEL_NONE=y*
   - *CONFIG_LOG_DEFAULT_LEVEL=0*

如需要在优化启动时间的基础上进一步优化上电后 Wi-Fi 连接 AP 的速度，可以同步进行以下操作：

1. **使能配置项 LWIP_DHCP_RESTORE_LAST_IP**
   - *CONFIG_LWIP_DHCP_RESTORE_LAST_IP=y*

启动时间及启动功耗结果统计 ESP32-S3 在不同主频下的启动时间统计:

CPU 频率	80 M	160 M	240 M
优化前启动时间（ms）	318.8	318.7	318.7
优化前启动功耗（mA）	44.1	51.4	57.3
优化后启动时间（ms）	26.87	26.875	26.965
优化后启动功耗（mA）	29.69	29.77	29.74

ESP32-C6 在不同主频下的启动时间统计:

CPU 频率	80 M	120 M	160 M
优化前启动时间（ms）	328.0	327.9	327.8
优化前启动功耗（mA）	34.1	35.8	37.6
优化后启动时间（ms）	33.31	33.315	33.315
优化后启动功耗（mA）	28.23	28.19	28.16

备注: 启动功耗为整个启动过程的平均电流。

2.3 方案介绍

这个部分我们将介绍乐鑫推出的应用方案。

2.3.1 模板

2.4 工具推荐

在这个栏目，我们将介绍一些常见的工具，虽然它们并非由 ESP 提供，但在 ESP 开发和调试过程中发挥着重要的作用。这些工具包括但不限于 Wireshark、Postman 等。通过掌握这些工具的使用方法，您将能够更加高效地进行 ESP 开发和调试，从而节省宝贵的时间并提高开发效率。

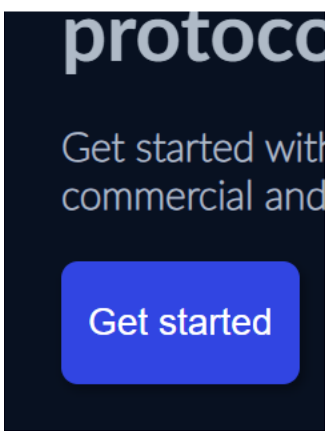
2.4.1 Windows 上的 Wireshark 抓包教程

本文介绍 Windows 10 系统下使用 Wireshark 抓 Wi-Fi 无线空包的详细教程。

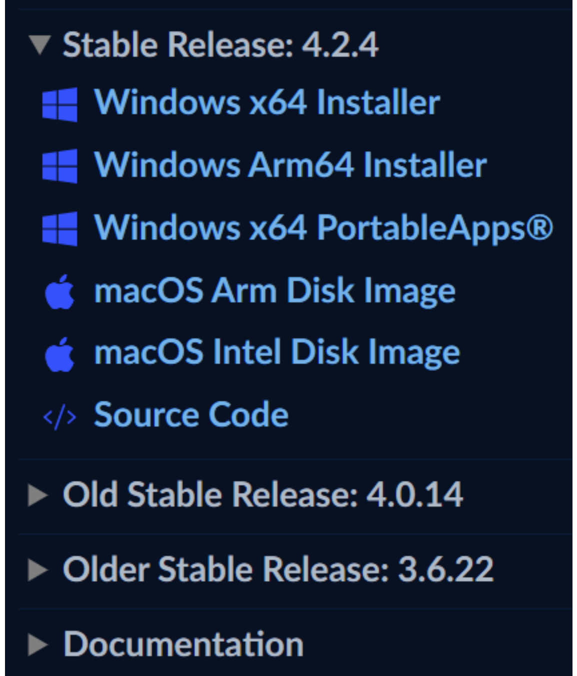
安装 Wireshark

关于 Wireshark Wireshark(前身 Ethereal) 是一个网络包分析工具。该工具主要是用来捕获网络数据包，并自动解析数据包，为用户显示数据包的详细信息，供用户对数据包进行分析。它可以运行在 Windows 和 Linux 操作系统上。可以使用该工具捕获并分析各类协议数据包，本文将讲解该工具的安装及基本使用方法。

下载及安装 Kali Linux 系统自带 Wireshark 工具，而 Windows 系统中默认没有安装该工具。进入 Wireshark 官网。单击 `Get started` 进入下载页面。



在 Stable Release 部分可以看到目前 Wireshark 的最新版本是 4.2.4，并提供了 Windows(32 位和 64 位)、Mac OS 和源码包的下载地址。您可以根据自己的操作系统下载相应的软件包。

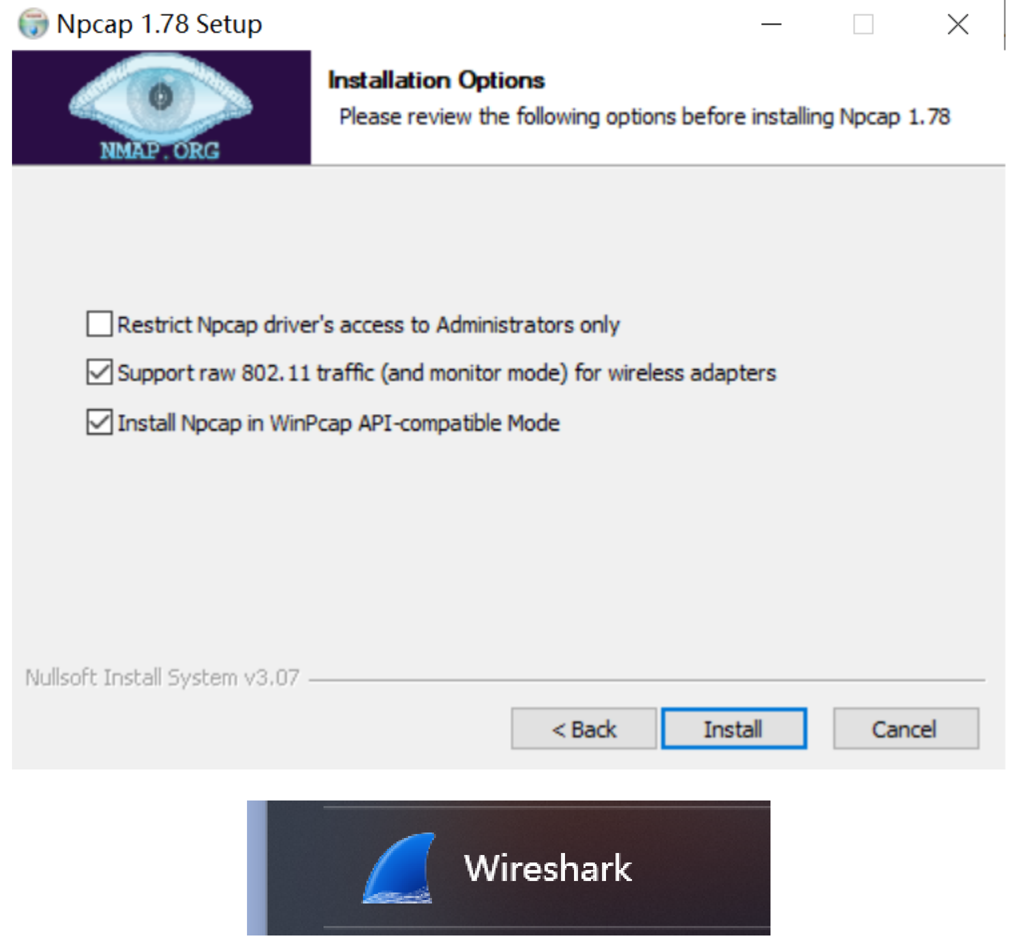


这里下载 Windows64 位的安装包。选择 `Windows Installer(64-bit)`，进行下载，下载后的文件名为 `Wireshark-win64-4.2.4.exe`。双击下载的软件包进行安装。安装使用默认值单击 `Next` 按钮即可。过程中需要注意：Wireshark 会向客户询问是否同步安装 Npcap 插件（默认勾选）。

Npcap 是由 Nmap 项目创建的高级数据包捕获和网络嗅探软件工具，它是一个轻量级但功能强大的平台，使用户能够在 Windows 操作系统上执行实时网络流量分析和监控。该步骤需要注意 Npcap 的版本，早期的 Npcap-1.7.1 版本无法使用 cmd 打开网卡的 monitor mode，在 1.76 版本后已修复。具体可参考 信息来源，目前最新下载的 Wireshark 自带的是 1.78 版本，可以直接安装。

> **注意：** Npcap 会弹窗进入第二个安装页面，需要勾选下图两个选项（**默认没有勾选**），不然无法抓取 802.11 包。

安装好以后，在 Windows 的“开始”菜单中会出现 Wireshark 图标。



抓包网卡

首要考虑的是抓包网卡需要支持监听模式 (Monitor mode)，其次需要考虑的是网卡支持的无线协议和传输速率。

监听模式 (Monitor Mode) 监听模式，或 RFMON(Radio Frequency Monitor)，是指无线网卡可以接收所有经过它的数据流的工作方式，对应于 IEEE 802.11 网卡的其他模式，诸如 Master（路由器）、Managed（普通模式的网卡）、Ad-hoc 等。监听模式不区分所接收数据包的目标 MAC 地址，这点和混杂模式类似。然而和混杂模式不同的是，监听模式**不需要**和无线接入点（AP）或 Ad-hoc 网络建立连接。监听模式是无线网卡特有的特殊模式，而混杂模式应用于有线网卡和无线网卡。监听模式通常用于网络发现、流量监听和分组分析（来源：Wikipedia）。

判断网卡支持的模式 可以通过 Npcap 工具查看当前网卡的模式、支持的模式并打开监听模式。

1. `win+R` 或开始菜单输入 `cmd` 打开 cmd 命令提示符窗口：

2. 查看网卡的 GUID，输入 `netsh wlan show interfaces`
3. 查看当前模式，复制 GUID，然后再输入 `WlanHelper.exe + GUID + modes`，如：

```
D:\>netsh wlan show interfaces

系统上有 1 个接口:

    名称                   : WLAN 2
    描述                   : Linksys WUSB600N Wireless-N USB Network Adapter with Dual-Band ver. 2
    GUID                   :
    物理地址               : 00:25:9c:e2:83:db
    状态                   : 已断开连接
    无线电状态             : 硬件 开
                             软件 开

    承载网络状态  : 未启动
```

```
wlanhelper.exe 00971d40-8f3c-426d-bab3-5a38ce789c45 modes
```

即可查看网卡支持的模式，许多笔记本自带的网卡只会有 managed 模式，此时就需要另外购买支持监听模式的无线网卡。

```
master, managed, monitor
```

4. 为网卡打开监听模式，输入 `WlanHelper.exe + GUID + mode monitor`，如：

```
wlanhelper.exe 00971d40-8f3c-426d-bab3-5a38ce789c45 mode monitor
```

成功会提示 `Success`：

```
D:\>wlanhelper.exe 00971d40-8f3c-4
Success
```

> **注意：** cmd 需要用管理员权限打开，否则无法打开监听模式并报错。也可输入 `mode` 查看当前的模式，更多 WlanHelper 的用法可键入 `WlanHelper.exe -h` 查看。

网卡挑选 根据您的抓包需求挑选合适的无线抓包网卡，如果您需要抓取 Wi-Fi 6 的包，那就需要选择支持 802.11 ax 协议的无线网卡。Wi-Fi 协议版本的对照表参考如下：

Generation	IEEE Standard	Radio Frequency (GHz)
Wi-Fi 6/6E	802.11ax	2.4, 5/6
Wi-Fi 5	802.11ac	5
Wi-Fi 4	802.11n	2.4, 5

您可以在购物网站搜索 无线抓包、抓包网卡等关键字，一般带有抓包功能的网卡都支持监听模式。

如果商品没有详细介绍参数，但有提供所使用的无线芯片型号，可以直接搜索型号查看该芯片的详细参数：

本文以雷凌芯片 RT3572 的思科抓包网卡为例，该芯片支持 802.11n 最高速率 300 Mbps，可以抓取 Wi-Fi 4 协议的空包。

配置网卡驱动

网卡到手后，需要安装对应的驱动程序，现在一些网卡是插上即用免安装，插上网卡后打开 控制面板 – 设备管理器，在 网络适配器栏中找到新插入的网卡，双击可查看详细信息，如驱动不适配，会有黄色三角形：

如果您的网卡插上不能用，本文介绍三种方法获得网卡的驱动：

¥38.88
RT3070L网卡无线usb接收器kali linux cdlinux虚拟机抓包linux RT3070L白色款
81条评价
安翼科技小店
免邮
对比 关注 加入购物车

¥86.00
杰奇洛omnipeek无线抓包网卡\ 空口抓包空中SparkLAN usb3.0双频千兆uos5g 黑
100+条评价
皙语外设产品专营店
对比 关注 加入购物车

¥125.00
EDIMAX usb无线网卡wifi接收器发射器win10免驱ubuntu kali linux抓包 7822UAn
5000+条评价
五星店铺 道恩数码专营店
京东物流 门店有售 免邮
对比 关注 加入购物车

¥48.88
RT3070L网卡无线usb接收器kali linux cdlinux虚拟机抓包linux RT3070L黑色款
81条评价
安翼科技小店
免邮
对比 关注 加入购物车

¥98.88
磊特wifi6无线网卡kali抓包实验测试airmon-ngj模式ax协议双频5g WiFi6-kali
26条评价
安翼科技小店
免邮
对比 关注 加入购物车

¥168.00
omnipeek无线抓包网卡空口抓包空中SparkLAN usb3.0双频千兆uos5g 黑色支
3条评价
京仁堂农资专营店
满199-5

¥90.00
rtl8812au 高功率usb无线网卡5g双频kali linux抓包ubuntu debain rtl8812au-不带延
57条评价
安翼科技小店
免邮

¥49.88
RT3070L网卡无线usb接收器kali linux cdlinux虚拟机抓包linux RT3070L白色外置
81条评价
安翼科技小店
免邮

¥70.00
EDIMAX usb无线网卡wifi接收器发射器win10免驱ubuntu kali linux抓包 7811Un
5000+条评价
五星店铺 道恩数码专营店
门店有售 免邮

¥98.00
rtl8812au 高功率usb无线网卡5g双频kali linux抓包ubuntu debain rtl8812au-
57条评价
安翼科技小店
免邮

Specifications:
FCC ID: RYK-WUBR507N
Antenna: Built-in PCB Antenna, 2T2R
Antenna Connector: U.FL IPEX / IPX Connector
interface: USB 2.0/1.1, type A
Wireless data rate:
- 802.11b: 11, 5.5, 2, 1 Mbps
- 802.11g: 54, 48, 36, 24, 18, 12, 9, 6Mbps
- 802. 11n: up to 300Mbps
- 802. 11a: up to 300Mbps
Frequency: 2.4GHz / 5GHz (NOT works with 802.11AC)
IEEE WLAN Standard: IEEE 802.11 a/b/g/n
Supported Systems: Kali Linux (Kali\ubuntu\Aircrack_ng), Linux, Windows XP/Vista/7/8/8.1/10 32/64-bit etc.
Dimension (L x W x H): 9.50 x 4.00 x 1.60 cm / 3.74 x 1.58 x 0.63 inch
Net Weight: 37.8g/0.08lb

What's in the box:
1x RT3572 USB WiFi Adapter

网络适配器
- Intel(R) Ethernet Connection (17) I219-LM
- Linksys WUSB600N Wireless-N USB Network Adapter with Dual-Band ver. 2
- WAN Miniport (IKEv2)
- WAN Miniport (IP)
- WAN Miniport (IPv6)
- WAN Miniport (L2TP)
- WAN Miniport (Network Monitor)
- WAN Miniport (PPPOE)
- WAN Miniport (PPTP)
- WAN Miniport (SSTP)

系统设备

- 使用购买网卡时商家提供的安装方法，包括但不限于：商家提供下载链接和安装方法，网卡自带储存盘内有对应驱动。
- 使用驱动精灵、驱动大师等第三方一键装机软件，自动为网卡安装合适的驱动。
- 搜索网卡型号或无线芯片的型号，去对应的芯片官网或第三方软件直接下载对应驱动。

安装完毕后，有些网卡需要重启电脑来适配。安装成功后，点开右下角的 `网络适配器 - WLAN` 即可搜索到周围的路由器：

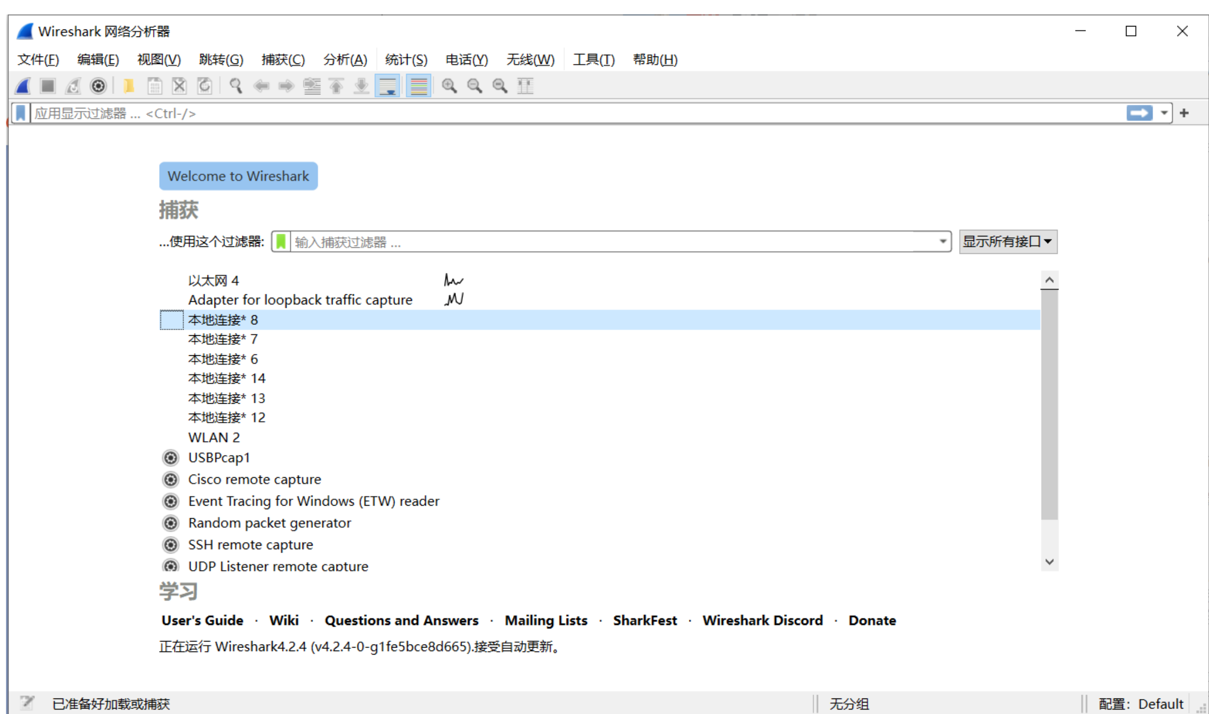
Wireshark 捕获数据包

安装好 Wireshark 以后，就可以运行它来捕获数据包了，启动 Wireshark：

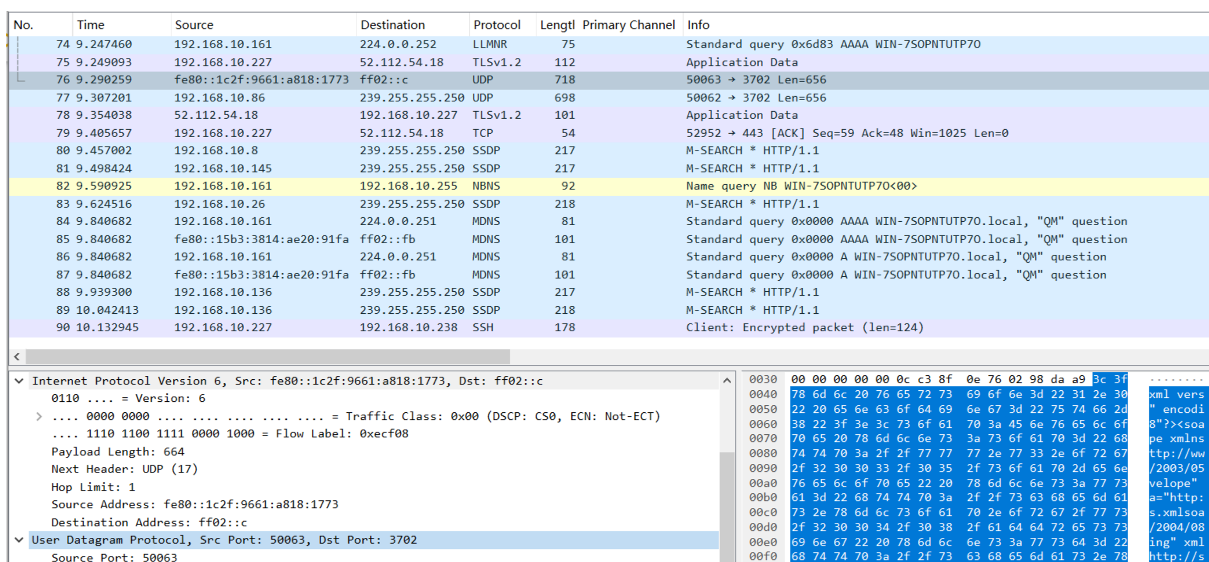
该图为 Wireshark 的主界面，界面中显示了当前可使用的接口，例如 `本地连接 8`、`WLAN 2` 等。要想捕获数据包，必须先选择一个接口，表示捕获该接口上的数据包。关于网络结构和协议的基础概念，可以参考博客：关于 Wi-Fi 网络基本原理了解。

抓取以太网数据包 您可以使用 Wireshark 抓取本机网卡的数据包，并分析抓取的数据包。

> **注意：** 本机访问本机的回环数据 localhost 是不经过网卡的，需要指定回环数据也要先转发到网关，才能使用 Wireshark 抓取。
>
> 也可以使用 Wireshark 打开其他抓包工具生成的抓包文件，使用 Wireshark 分析该抓包文件。



选择捕获 以太网 4 接口上的数据包，然后单击左上角的 开始捕获或直接双击接口名，将进行捕获网络数据，当本地计算机浏览网站时，本地连接接口的数据将会被 Wireshark 捕获到：



Wireshark 将一直捕获 本地连接上的数据。如果不需要再捕获，可以单击左上角的 停止捕获分组按钮，停止捕获。

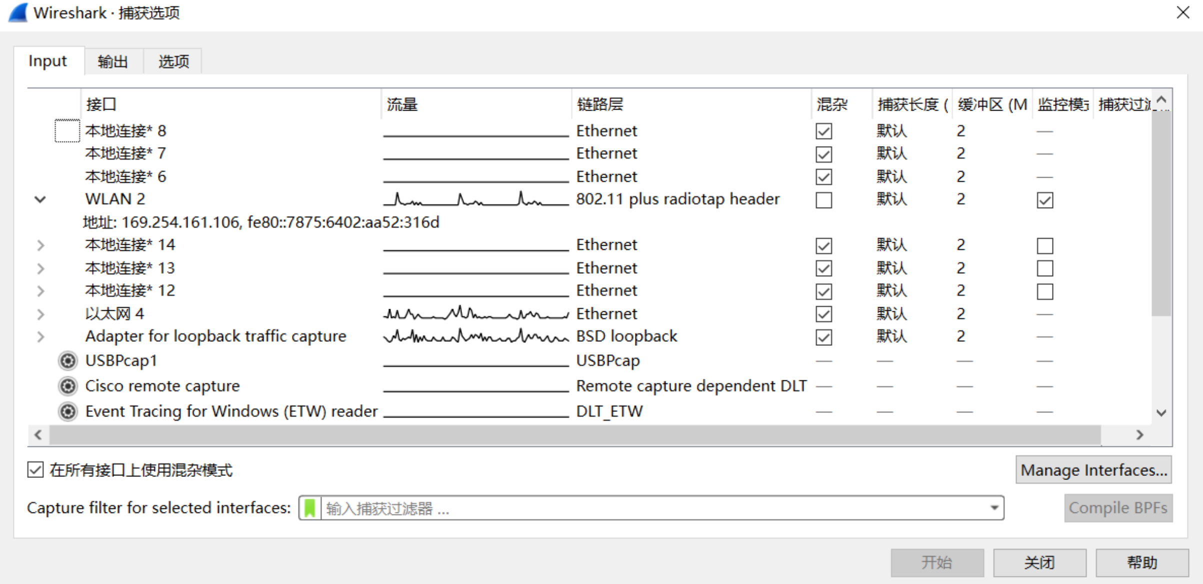
抓取 Wi-Fi 无线数据包

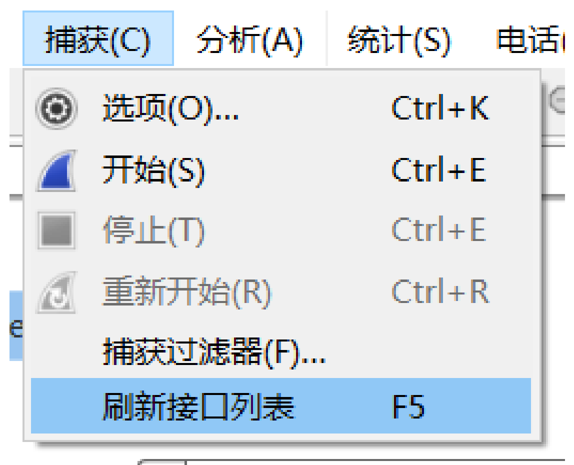
设置捕获选项 菜单栏 捕获-选项（快捷键 `Ctrl + k`）可进入捕获选项设置界面：

需要选择刚才配置好的无线网卡，点开下拉菜单可查看对应 IP 地址：

> **注意：** 连接路由器前后的 IP 地址会变，请通过 捕获-刷新接口列表（快捷键 `F5`）获取最新的接口信息：

停止捕获分组
捕获(C)
分析(A)
统计(S)
电话
选项(O)... Ctrl+K
开始(S) Ctrl+E
停止(T) Ctrl+E
重新开始(R) Ctrl+R
捕获过滤器(F)...
刷新接口列表 F5

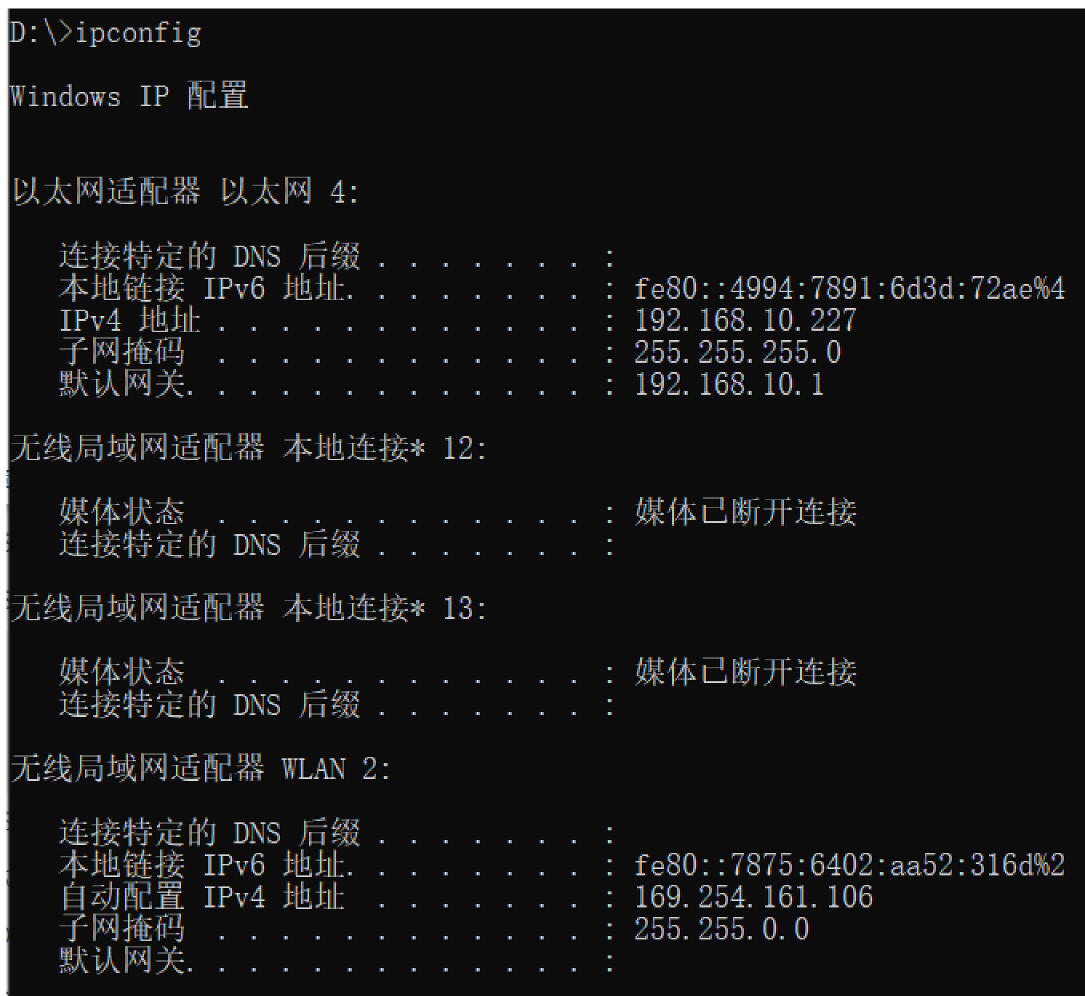

Wireshark · 捕获选项
Input
输出
选项
接口
流量
链路层
混杂
本地连接* 8 Ethernet 默认 2
本地连接* 7 Ethernet 默认 2
本地连接* 6 Ethernet 默认 2
WLAN 2 802.11 plus radiotap header 默认 2
地址: 169.254.161.106, fe80::7875:6402:aa52:316d
本地连接* 14 Ethernet 默认 2
本地连接* 13 Ethernet 默认 2
本地连接* 12 Ethernet 默认 2
以太网 4 Ethernet 默认 2
Adapter for loopback traffic capture BSD loopback 默认 2
USBPcap1 USBPcap
Cisco remote capture Remote capture dependent DLT
Event Tracing for Windows (ETW) reader DLT_ETW
在所有接口上使用混杂模式
Manage Interfaces...
Capture filter for selected interfaces:
输入捕获过滤器 ...
Compile BPFs
开始
关闭
帮助


捕获(C)
分析(A)
统计(S)
选项(O)... Ctrl+K
开始(S) Ctrl+E
停止(T) Ctrl+E
重新开始(R) Ctrl+R
捕获过滤器(F)...
刷新接口列表 F5

选择抓包网卡 不同的电脑配置的网络接口名称可能会不一样，您可以通过以下几个方法确认抓包的无线网卡：

方法 1 连上一个无线网络后，`win+R` 或开始菜单输入 `cmd` 打开 cmd 命令提示符窗口：

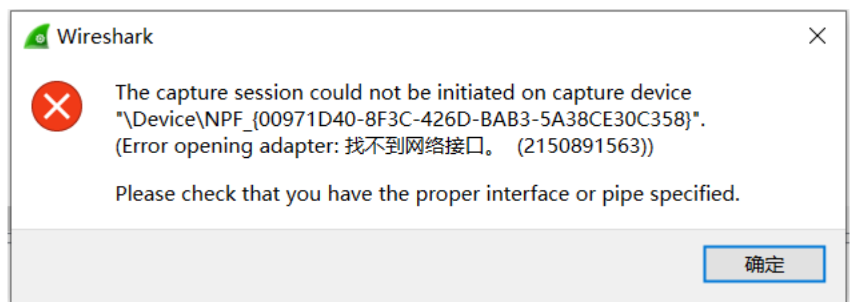
在命令栏输入 `ipconfig` 后回车，可以看到当前网络的接口信息，对应 IP 地址可确定无线网卡为 WLAN 2：



方法 2 您也可以直接拔掉网卡，再在 Wireshark 中尝试使用 `WLAN 2` 捕获，系统提示找不到接口，或 菜单-捕获-刷新接口列表，也可确定该网卡的接口名。

方法 3 打开 cmd 命令提示符窗口，输入 `netsh wlan show interfaces`，即可查看到对应的无线网络接口信息。

确定需要抓取的信道 选择您需要抓取的 Wi-Fi 信道，如果您想抓取被测路由器的交互信息，可以按如下方法确认路由器的信道、带宽等信息：



方法 1 打开 cmd 命令提示符窗口，电脑连接上被测路由器，输入 `netsh wlan show interfaces`，即可查看当前连接的信道。

```
D:\>netsh wlan show interfaces

系统上有 1 个接口:

    名称                   : WLAN 2
    描述                   : Linksys WUSB600N Wireless-N USB Network Adapter with Dual-Band ver. 2
    GUID                   : 00971d40-8f3c-426d
    物理地址               : 00:25:9c:e2
    状态                   : 已连接
    SSID                   :
    BSSID                  : 70:3a:73:84
    网络类型               : 结构
    无线电类型             : 802.11n
    身份验证               : WPA2 - 个人
    密码                   : CCMP
    连接模式               : 配置文件
    信道                   : 60
    接收速率(Mbps)         : 144
    传输速率 (Mbps)        : 144
    信号                   : 80%
    配置文件               :

    承载网络状态  : 未启动
```

方法 2 去路由器管理员设置界面查看，此处不展开。

方法 3 手机下载 Wi-Fi 嗅探软件，如 WiFi 魔盒、wifiman、WiFi Analyzer、inSSIDer、WirelessMon 等，都可查看周围的 AP 信息和信道分析。

设置网卡 monitor 信道 打开 cmd 命令提示符窗口，按照前文*判断网卡支持的模式*章节介绍的方法打开网卡的监听模式。

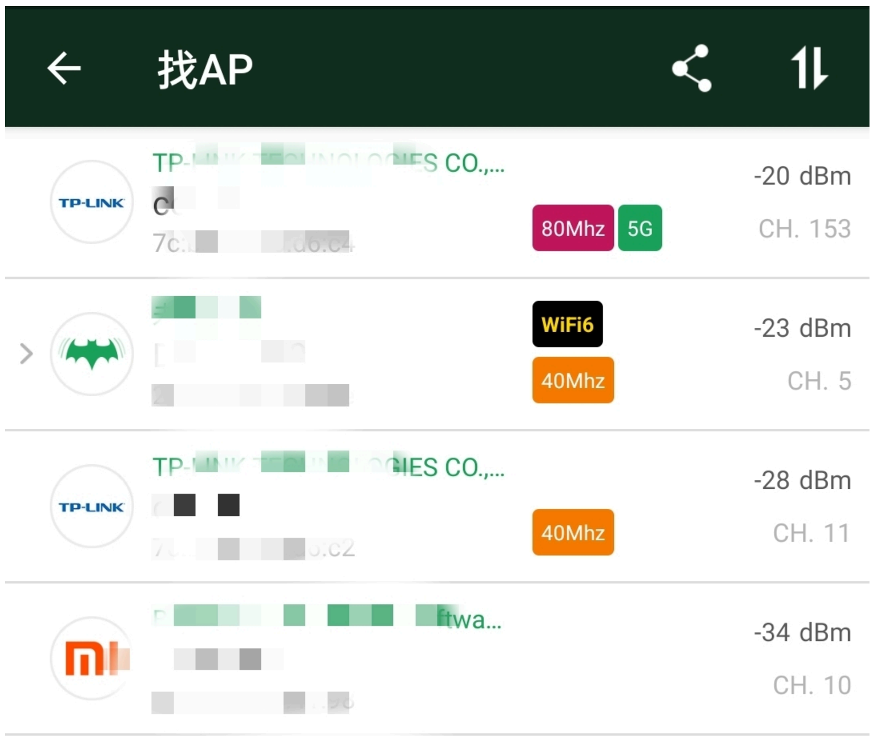
> **注意：** 打开该模式后已连接的网络会断连，这属于正常现象，因为普通网卡模式是 managed，后续抓包完成后可改回来。

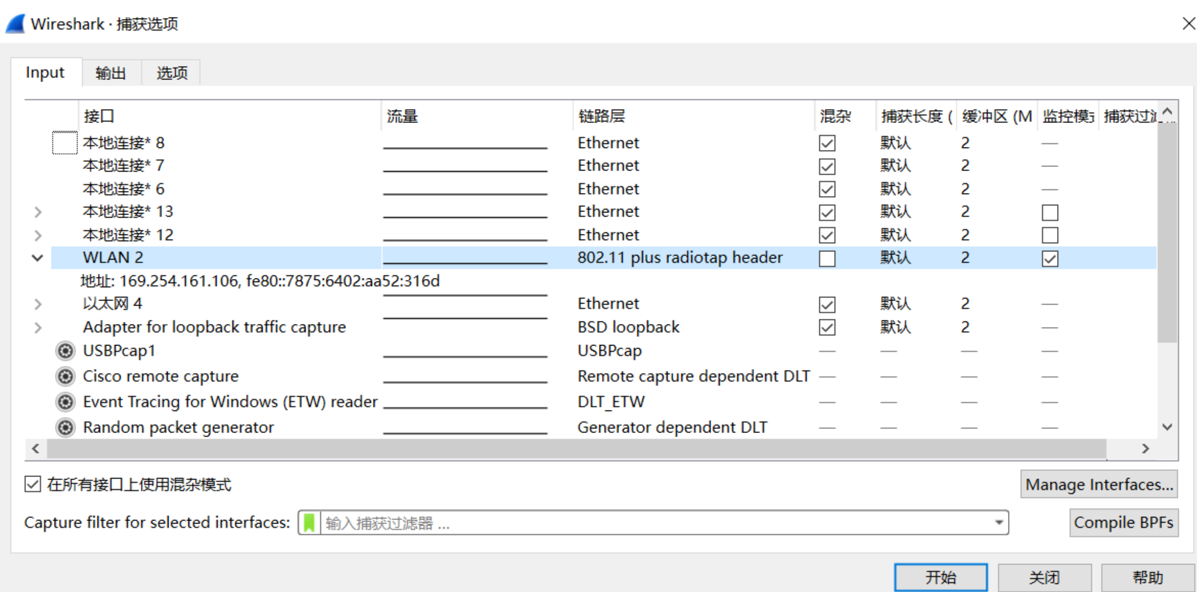
打开监听模式，输入 `WlanHelper.exe + GUID + mode monitor` 如：

```
wlanhelper.exe 00971d40-8f3c-426d-bab3-5a38ce789c45 mode monitor
```

配置需要监听的信道，输入 `WlanHelper.exe + GUID + channel [value]`，如信道 60：

```
wlanhelper.exe 00971d40-8f3c-426d-bab3-5a38ce789c45 channel 60
```


找AP
TP-LINK
80Mhz
5G
-20 dBm
CH. 153
WiFi6
40Mhz
-23 dBm
CH. 5
TP-LINK
40Mhz
-28 dBm
CH. 11
-34 dBm
CH. 10


Wireshark · 捕获选项
Input
输出
选项
接口
流量
链路层
混杂
捕获长度 (
缓冲区 (M
监控模
捕获过
本地连接* 8
本地连接* 7
本地连接* 6
本地连接* 13
本地连接* 12
WLAN 2
地址: 169.254.161.106, fe80::7875:6402:aa52:316d
以太网 4
Adapter for loopback traffic capture
USBPcap1
Cisco remote capture
Event Tracing for Windows (ETW) reader
Random packet generator
Ethernet
802.11 plus radiotap header
BSD loopback
USBPcap
Remote capture dependent DLT
DLT_ETW
Generator dependent DLT
默认
2
在所有接口上使用混杂模式
Manage Interfaces...
Capture filter for selected interfaces:
输入捕获过滤器 ...
Compile BPFs
开始
关闭
帮助

配置 Wireshark 捕获选项 在 `input` 栏位选择对应网卡接口，不勾选混杂模式，勾选监控模式。

在 输出栏位选择需要保存的路径和存储的格式。

点击 开始或双击接口名即可抓取空中的无线网络数据包了。

Wireshark 过滤器的使用

Wireshark 设置了两个过滤器：捕获过滤器 (capture filter)、显示过滤器 (display filter)。

- **捕获过滤器：**

  用于在 **开始捕获前**设置过滤条件，设置过滤条件后，抓包工具将仅捕获与条件匹配的数据包；使用捕获过滤器可以减少抓取的网络数据包，减轻抓包软件和储存空间的负担，最终得到的抓包文件也较小，是提升效率必备技能；

- **显示过滤器：**

  用于在 **捕获数据后**设置过滤条件，设置过滤条件后，显示页面上将仅显示与条件匹配的数据包，有助于工程师分析报文。

图片来源

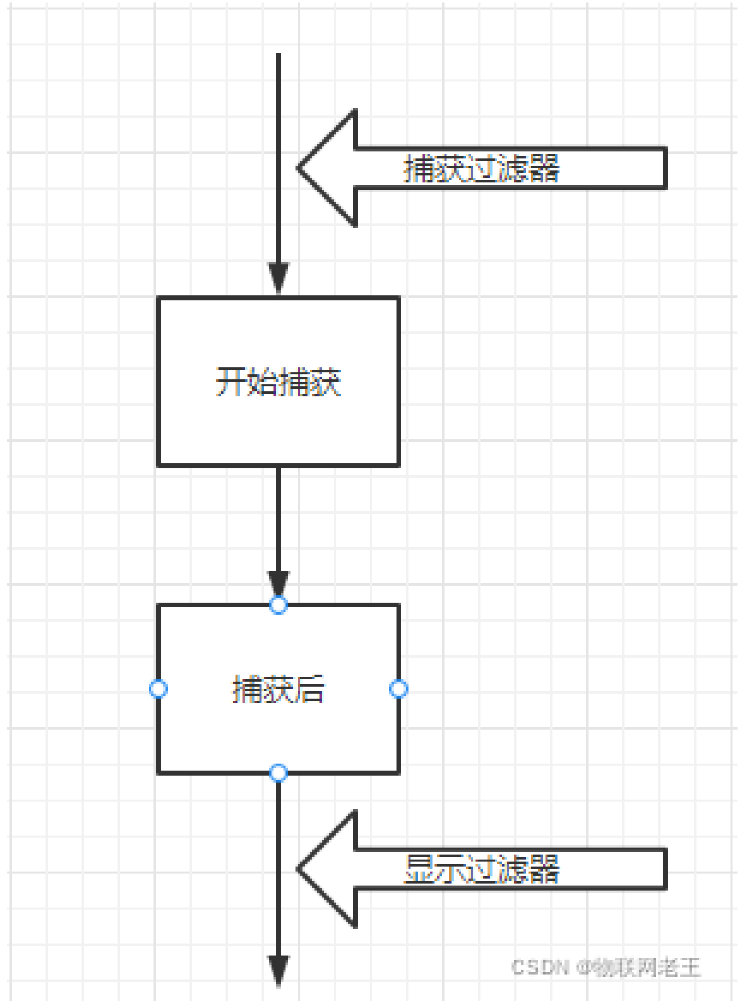
捕获过滤器的基本使用 打开 Wireshark 软件后，如图所示的输入框就是输入捕获过滤条件的地方：

点击上图中绿色的小标签（或者通过菜单：捕获 – 捕获过滤器）即可打开常用的捕获表达式：

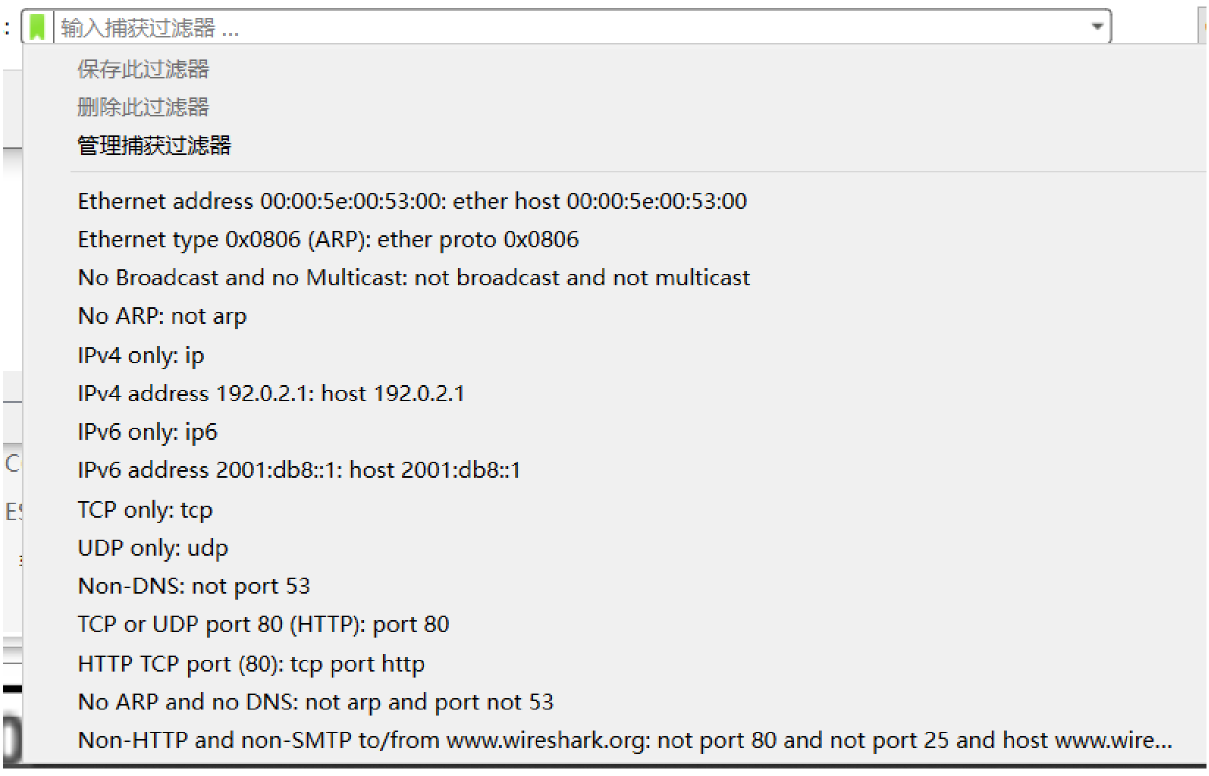
注：捕获表达式中的冒号 `:`，通常起解释说明的意思，无实意。

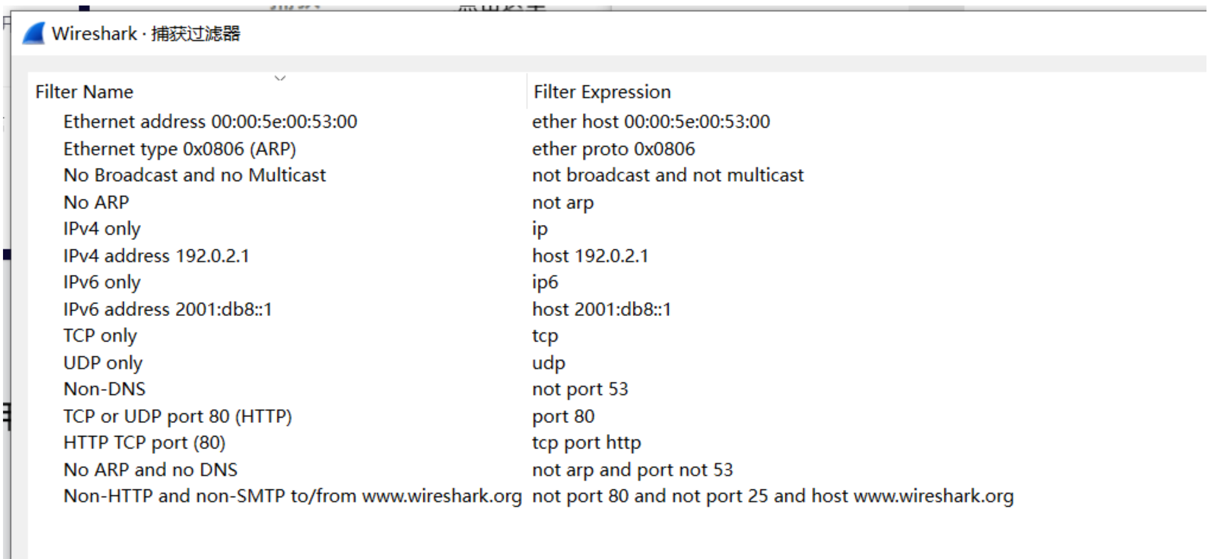
捕获过滤表达式的语法 Wireshark 捕获过滤器表达式遵循 libpcap 语法。过滤器表达式由一个或多个原语组成。原语通常由一个 id (名称或数字) 和一个或多个修饰符组成。

1. 过滤表达式 = 原语 1 + 原语 2 + …
2. 原语 = id + 修饰符 1 + 修饰符 2 + …
3. 原语之间可以通过逻辑连接符和括号 () 进行组合，逻辑连接符包括：
   - 与：可用符号 `&&` 或者文字 `and` 来表示
   - 或：可用符号 `||` 或者文字 `or` 来表示
   - 非：可用符号 `!` 或者文字 `not` 来表示


捕获过滤器
开始捕获
捕获后
显示过滤器
CSDN @物联网老王

在所有接口上使用混杂模式
Capture filter for selected interfaces:
输入捕获过滤器 ...
开始


输入捕获过滤器 ...
保存此过滤器
删除此过滤器
管理捕获过滤器
Ethernet address 00:00:5e:00:53:00: ether host 00:00:5e:00:53:00
Ethernet type 0x0806 (ARP): ether proto 0x0806
No Broadcast and no Multicast: not broadcast and not multicast
No ARP: not arp
IPv4 only: ip
IPv4 address 192.0.2.1: host 192.0.2.1
IPv6 only: ip6
IPv6 address 2001:db8::1: host 2001:db8::1
TCP only: tcp
UDP only: udp
Non-DNS: not port 53
TCP or UDP port 80 (HTTP): port 80
HTTP TCP port (80): tcp port http
No ARP and no DNS: not arp and port not 53
Non-HTTP and non-SMTP to/from www.wireshark.org: not port 80 and not port 25 and host www.wire...


Wireshark · 捕获过滤器
Filter Name
Filter Expression
Ethernet address 00:00:5e:00:53:00
ether host 00:00:5e:00:53:00
Ethernet type 0x0806 (ARP)
ether proto 0x0806
No Broadcast and no Multicast
not broadcast and not multicast
No ARP
not arp
IPv4 only
ip
IPv4 address 192.0.2.1
host 192.0.2.1
IPv6 only
ip6
IPv6 address 2001:db8::1
host 2001:db8::1
TCP only
tcp
UDP only
udp
Non-DNS
not port 53
TCP or UDP port 80 (HTTP)
port 80
HTTP TCP port (80)
tcp port http
No ARP and no DNS
not arp and port not 53
Non-HTTP and non-SMTP to/from www.wireshark.org
not port 80 and not port 25 and host www.wireshark.org

如该示例表示仅捕获通过网关 sup 并且属于 FTP 端口或者数据的数据包：

```
gateway snup and (port ftp or ftp-data)
```

如该示例表示不捕获 arp 类型的数据包：

```
not arp
```

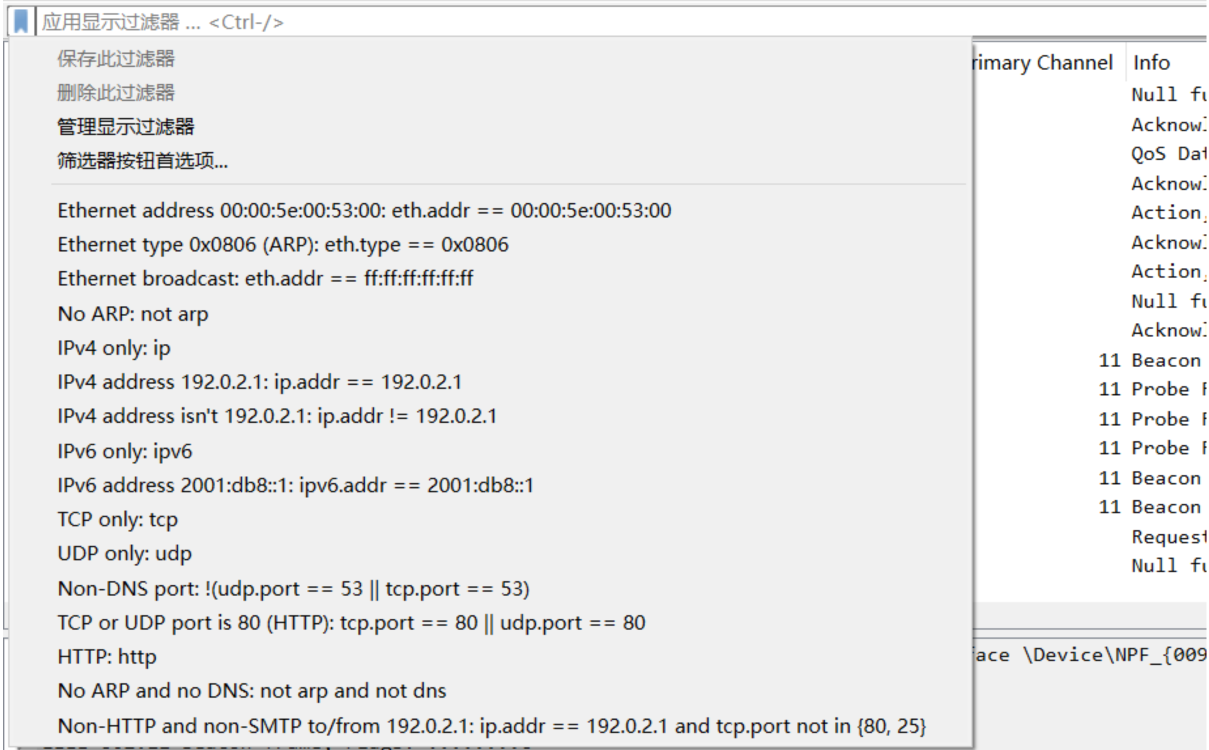
如该示例表示仅捕获 tcp 或 udp 类型的数据包：

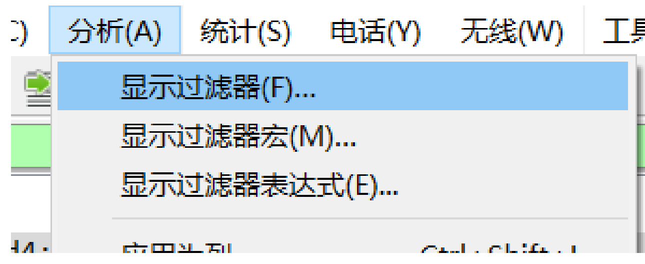
```
tcp || udp
```

值得一提的是，该过滤器自带语法检测，底框绿色即代表语法正确。

备注： 更多捕获过滤器的使用方法与示例可参考 Wireshark 官方说明 capture filters wiki 以及 capture filters Gitlab。

显示过滤器的基本使用 进入到抓包页面后，点击下图中的标签，或者通过菜单 分析- display filters 即可打开常用的显示过滤表达式：





Wireshark · 显示过滤器

Filter Name	Filter Expression
Ethernet address 00:00:5e:00:53:00	eth.addr == 00:00:5e:00:53:00
Ethernet type 0x0806 (ARP)	eth.type == 0x0806
Ethernet broadcast	eth.addr == ff:ff:ff:ff:ff:ff
No ARP	not arp
IPv4 only	ip
IPv4 address 192.0.2.1	ip.addr == 192.0.2.1
IPv4 address isn't 192.0.2.1	ip.addr != 192.0.2.1
IPv6 only	ipv6
IPv6 address 2001:db8::1	ipv6.addr == 2001:db8::1
TCP only	tcp
UDP only	udp
Non-DNS port	!(udp.port == 53 || tcp.port == 53)
TCP or UDP port is 80 (HTTP)	tcp.port == 80 || udp.port == 80
HTTP	http
No ARP and no DNS	not arp and not dns
Non-HTTP and non-SMTP to/from 192.0.2.1	ip.addr == 192.0.2.1 and tcp.port not in {80, 25}

显示过滤表达式的语法 与捕获过滤表达式类似，显示过滤表达式也可看作原语的组合。不同的是，显示过滤表达式的原语由"选项 + 选项关系 + 选项值"组成。如 `tcp.port == 80` 中，`tcp.port` 是选项，`==` 是选项关系，`80` 是选项值，整个表达式表示的是：仅显示 tcp 端口号（包括发送、接收）为 80 的数据包。

再次提醒，显示过滤表达式和捕获过滤表达式不能混淆，捕获过滤表达式的形式为：tcp port80。表达式中三个部分分别介绍如下：

- 选项：选项可以是协议（如 tcp、udp、http 等）、帧 (frame) 等对象，这个在使用中参考已经有的示例或者直接输入协议名字，看是否有提示即可
- 选项关系：用于定义选项与选项值的关系。常见的选项关系如下：

表格来源：Wireshark 官网

- 选项值：选项值可以是数字，如十进制数字 `1500`、十六进制数字 `0x5dc`、二进制数字 `0b10111011100`、布尔值（如 1 代表 `true`，0 代表 `false`）、MAC 地址类的数字（如 `eth.dst == ff:ff:ff:ff:ff:ff`）、ip 地址（如 `ip.addr == 192.168.0.1`）、字符串（如 `http.request.uri == "https://ww.wireshark.org/"`）、时间（如 `ntp.xmt ge "2020-07-0412:34:56"`）等。

多个表达式也可以通过逻辑连接符连接起来，组合出高级的显示过滤表达式。

表格来源：Wireshark 官网

其中，[…] 表示 子串选择符，如 `eth.src[0:3] == 00:00:83` 表示从偏移地址 0 开始的 3 个字节的数据为 `00:00:83` 的数据包。其中 `in` 表示成员选择符，常用于组成选项值的集合。如 `tcp.port in {80,441,8081}` 指的是过滤 tcp 端口为 80、441、8081 的数据包。其等同于 `tcp.port == 80 || tcp.port == 441 || tcp.port == 8081`。

备注： 更多捕获过滤器的使用方法与示例可参考 Wireshark 官方说明 capture filters wiki 以及 capture filters Gitlab。

分析封包

在掌握 Wireshark 抓包方法后，即可对抓取到的包进行初步分析，下面是一些优秀的抓包分析示例：

Table 6.6. Display Filter comparison operators

English	Alias	C-like	Description	Example
eq	any_eq	==	Equal (any if more than one)	`ip.src == 10.0.0.5`
ne	all_ne	!=	Not equal (all if more than one)	`ip.src != 10.0.0.5`
	all_eq	===	Equal (all if more than one)	`ip.src === 10.0.0.5`
	any_ne	!==	Not equal (any if more than one)	`ip.src !== 10.0.0.5`
gt		>	Greater than	`frame.len > 10`
lt		<	Less than	`frame.len < 128`
ge		>=	Greater than or equal to	`frame.len ge 0x100`
le		<=	Less than or equal to	`frame.len <= 0x20`
contains			Protocol, field or slice contains a value	`sip.To contains "a1762"`
matches		~	Protocol or text field matches a Perl-compatible regular expression	`http.host matches "acme\\.(org|com|net)"`

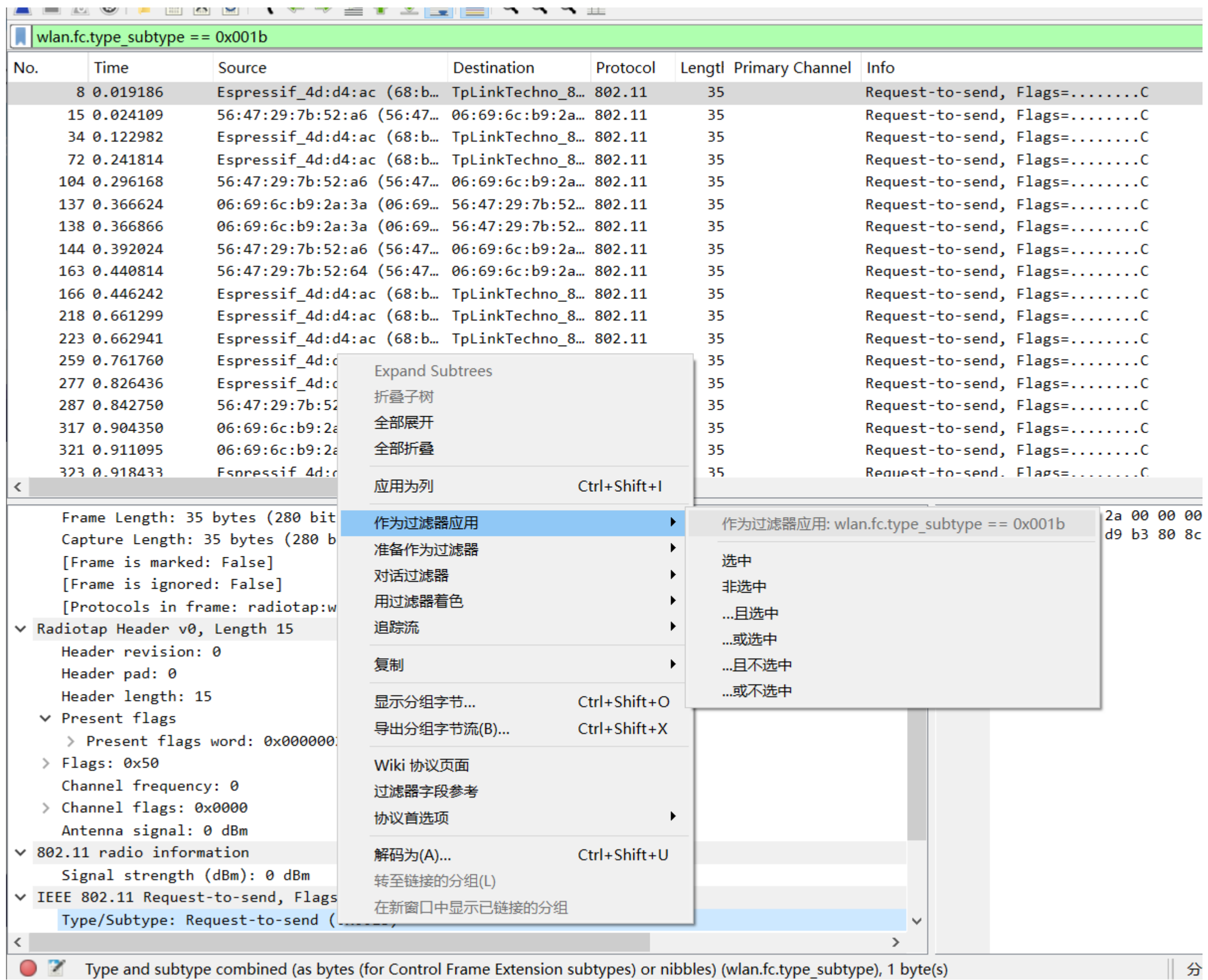
Table 6.7. Display Filter Logical Operations

English	C-like	Description	Example
and	&&	Logical AND	`ip.src==10.0.0.5 and tcp.flags.fin`
or	||	Logical OR	`ip.src==10.0.0.5 or ip.src==192.1.1.1`
xor	^^	Logical XOR	`tr.dst[0:3] == 0.6.29 xor tr.src[0:3] == 0.6.29`
not	!	Logical NOT	`not llc`
[...]		Subsequence	See "Slice Operator" below.
in		Set Membership	http.request.method in {"HEAD", "GET"}. See "Membership Operator" below.

- Wireshark 抓包分析 WLAN 连接过程
- 使用 wireshark 分析 Ping 通信的具体流程
- Wireshark 抓包分析 TCP 三次握手

本文最后再分享一些分析封包中的 Wireshark 使用小技巧：

小技巧 1 除了手动在输入框内输入显示过滤表达式外，还可以选中抓包数据的某个选项，右键选择 作为过滤器应用 (Apply as Filter)，根据需求选择 选中或 非选中或下面更高级的 and or 逻辑操作，如下图过滤了 subtype 为 0x001b 的 RTS 封包，该操作等同于在显示过滤器栏输入 wlan.fc.type_subtype == 0x001b 。



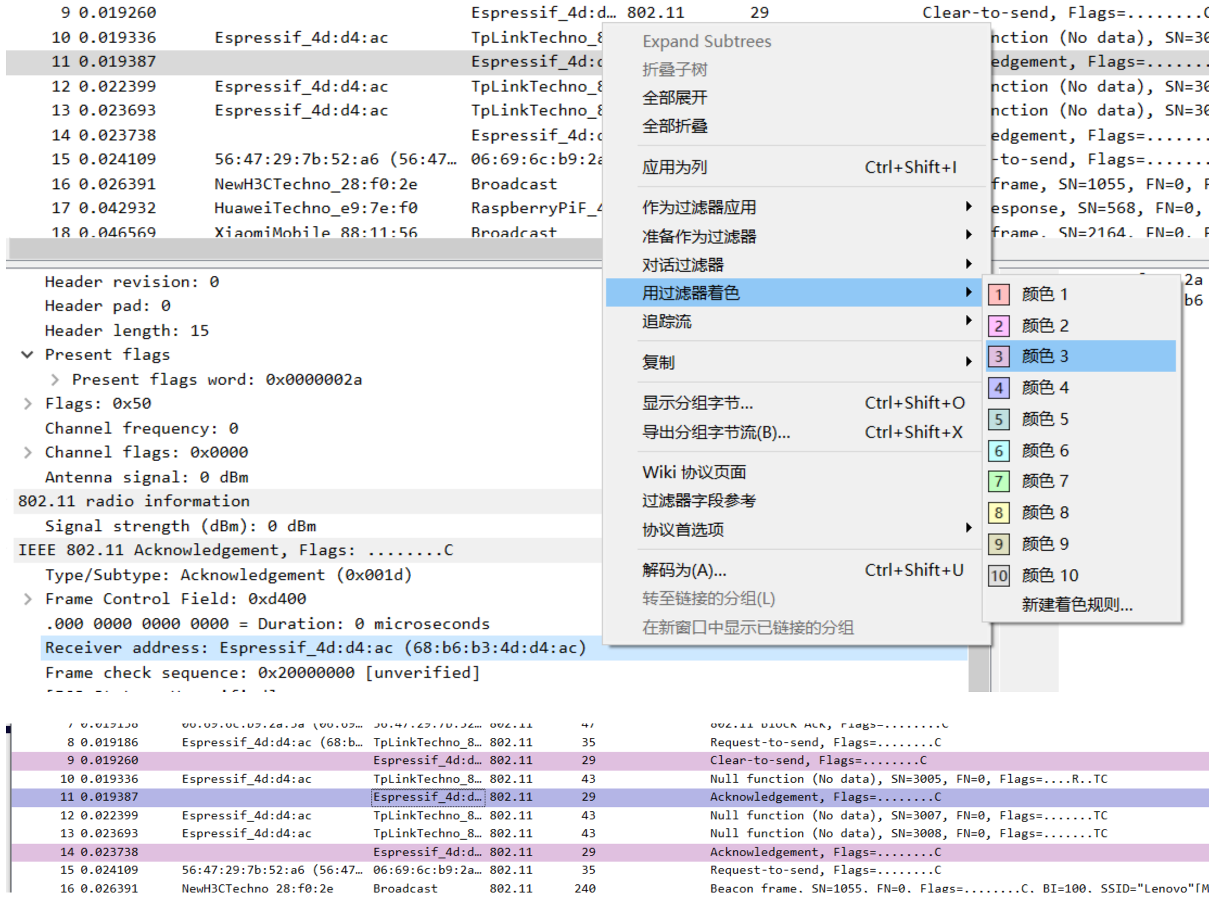
小技巧 2 使用着色器协助进行各类封包的分析，选中抓包数据的某个选项，右键选择 用过滤器应用着色即可对满足特定条件的封包着色，如下图把 Receiver address 为 Espressif_4d:d4:ac (68:b6:b3:4d:d4:ac) 的封包着色为颜色 3：

可在 视图 – 对话着色中管理或重置现有规则：

小技巧 3 您可以将感兴趣的任意一项字段拖动到列表栏进行显示，比如下图把 current channel 栏位拖动到了 Packet List Pane(数据包列表)，这样就可以方便地对每一包的 channel 信息进行查看：

右键列表栏可管理每一个 column 的设置规则：

也可选择 Edit column 进行设置：



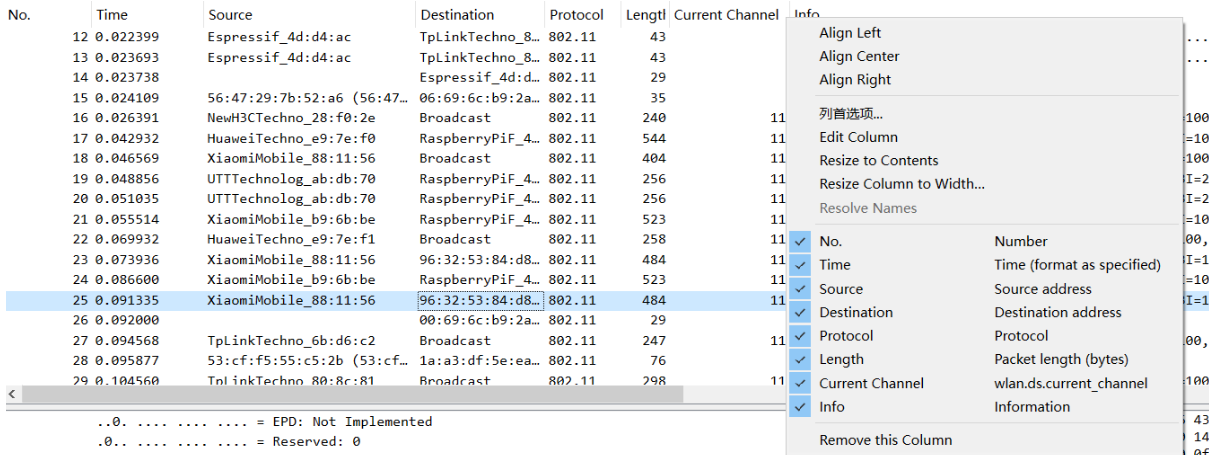
小技巧 4 有时我们需要对复杂问题进行 debug，会在不同设备抓取多份 log，设置绝对时间戳可帮助跨设备定位复杂问题，按小技巧 3 的方法打开 `column` 设置栏，选择 `Absolute date`，可显示每一包的绝对时间：

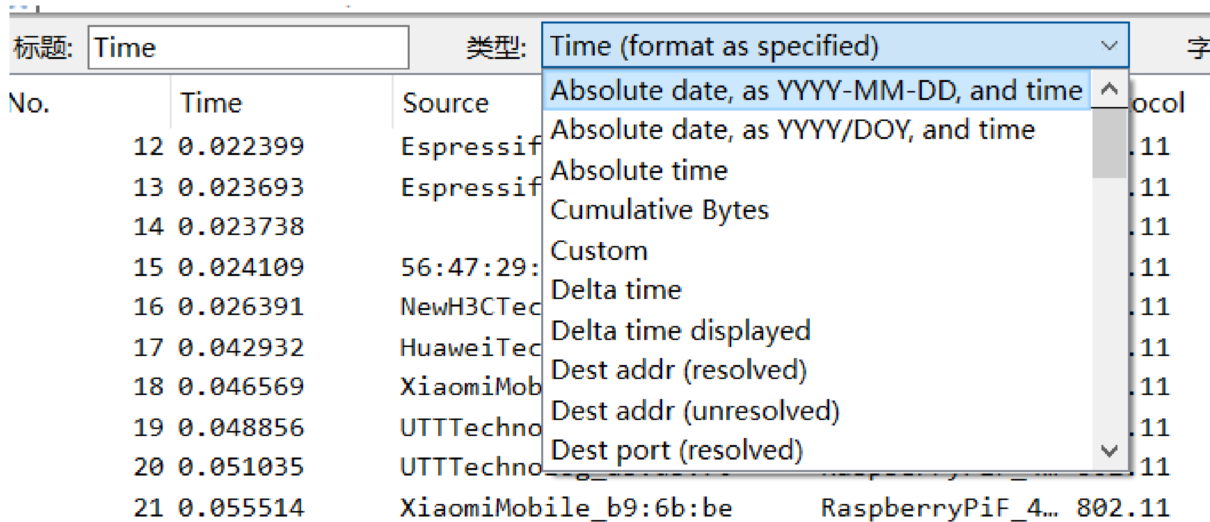
设置好后：

中文参考文档

- 网络抓包工具 Wireshark 下载安装 & 使用详细教程
- Wireshark 基础使用-启用抓包与过滤
- Wireshark 基础使用-过滤并查看抓包数据
- Windows 下 Wi-Fi 抓包教程

```
        0... .... .... .... = Reserved: 0
  ⌄ Tagged parameters (429 bytes)
    > Tag: SSID parameter set: "Audio_CI"
    > Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 6,
    ⌄ Tag: DS Parameter set: Current Channel: 11
        Tag Number: DS Parameter set (3)
        Tag length: 1
        Current Channel: 11
    > Tag: Country Information: Country Code CN, Environm
    > Tag: ERP Information
```





	Time	Sc
12	2024-05-08 11:56:45.093679	Es
13	2024-05-08 11:56:45.094973	Es
14	2024-05-08 11:56:45.095018	
15	2024-05-08 11:56:45.095389	56
16	2024-05-08 11:56:45.097671	Ne
17	2024-05-08 11:56:45.114212	Hu
18	2024-05-08 11:56:45.117849	Xi
19	2024-05-08 11:56:45.120136	UT
20	2024-05-08 11:56:45.122315	UT
21	2024-05-08 11:56:45.126794	Xi
22	2024-05-08 11:56:45.141212	Hu
23	2024-05-08 11:56:45.145216	Xi
24	2024-05-08 11:56:45.157880	Xi
25	2024-05-08 11:56:45.162615	Xi
26	2024-05-08 11:56:45.163280	
27	2024-05-08 11:56:45.165848	Tp
28	2024-05-08 11:56:45.167157	53