

瑞芯微多核异构系统开发指南

文件标识: RK-KF-YF-160

发布版本: V1.0.0

日期: 2024-03-15

文件密级: ☐绝密 ☐秘密 ☐内部资料 ☒公开

免责声明

本文档按“现状”提供, 瑞芯微电子股份有限公司(“本公司”, 下同)不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因, 本文档将可能在未经任何通知的情况下, 不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标, 归本公司所有。

本文档可能提及的其他所有注册商标或商标, 由其各自拥有者所有。

版权所有 © 2024 瑞芯微电子股份有限公司

超越合理使用范畴, 非经本公司书面许可, 任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部, 并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址: 福建省福州市铜盘路软件园A区18号

网址: www.rock-chips.com

客户服务电话: +86-4007-700-590

客户服务传真: +86-591-83951833

客户服务邮箱: fae@rock-chips.com

前言

概述

本文档主要指导工程师基于瑞芯微多核异构系统进行项目开发。

平台支持

芯片名称	处理器核心	运行平台		
		Linux	RTOS	Bare-metal
RK3588	4 x ARM Cortex-A76	Kernel 5.10	N/A	N/A
	4 x ARM Cortex-A55	Kernel 5.10	RTT 3.1-32 RTT 4.1-32	HAL-32
	1 x ARM Cortex-M0	N/A	RTT 3.1 RTT 4.1	HAL
RK3576	4 x ARM Cortex-A72	Kernel 6.1	N/A	N/A
	4 x ARM Cortex-A53	Kernel 6.1	RTT 4.1-32	HAL-32
	1 x ARM Cortex-M0	N/A	RTT 4.1	HAL
RK3568	4 x ARM Cortex-A55	Kernel 4.19 Kernel 5.10	RTT 3.1-32	HAL-32
	1 x RISC-V	N/A	RTT 3.1	HAL
RK3562	4 x ARM Cortex-A53	Kernel 5.10	RTT 4.1-32	HAL-32
	1 x ARM Cortex-M0	N/A	RTT 4.1	HAL
RK3358	4 x ARM Cortex-A35	N/A	RTT 3.1-32	HAL-32
RK3308	4 x ARM Cortex-A35	Kernel 5.10	RTT 3.1-32 RTT 4.1-32	HAL-32

读者对象

本文档（本指南）主要适用于以下工程师：

软件开发工程师

技术支持工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	刘诗舫、邹鸿名、王征增、郑嘉航、杨汉兴	2024-03-15	初始版本

目录

瑞芯微多核异构系统开发指南

1. 第1章 多核异构系统
 - 1.1 概述
 - 1.1.1 多核异构系统简介
 - 1.1.2 瑞芯微多核异构系统
 - 1.2 平台支持
 - 1.2.1 RK3588
 - 1.2.1.1 处理器核心
 - 1.2.1.2 运行平台支持
 - 1.2.2 RK3576
 - 1.2.2.1 处理器核心
 - 1.2.2.2 运行平台支持
 - 1.2.3 RK3568
 - 1.2.3.1 处理器核心
 - 1.2.3.2 运行平台支持
 - 1.2.4 RK3562
 - 1.2.4.1 处理器核心
 - 1.2.4.2 运行平台支持
 - 1.2.5 RK3358
 - 1.2.5.1 处理器核心
 - 1.2.5.2 运行平台支持
 - 1.2.6 RK3308
 - 1.2.6.1 处理器核心
 - 1.2.6.2 运行平台支持
 - 1.3 产品案例介绍
 - 1.3.1 AP + AP 案例：电力继电保护装置
 - 1.3.2 AP + MCU 案例：扫地机器人
2. 第2章 AMP SDK
 - 2.1 目录结构
 - 2.2 device 目录
 - 2.3 kernel 目录
 - 2.4 hal 目录
 - 2.5 rtos 目录
 - 2.6 u-boot 目录
 - 2.7 rkbin 目录
3. 第3章 编译配置
 - 3.1 配置文件
 - 3.1.1 统一编译配置文件
 - 3.1.2 AMP 固件打包配置文件
 - 3.1.2.1 amp_linux.its
 - 3.1.2.2 amp.its
 - 3.1.2.3 amp_mcu.its
 - 3.1.3 辅助配置文件
 - 3.1.4 分区表配置文件
 - 3.2 编译命令
 - 3.2.1 统一编译命令
 - 3.2.2 单独编译命令
 - 3.2.2.1 Linux Kernel 编译命令
 - 3.2.2.2 RT-Thread 编译命令
 - 3.2.2.3 RK HAL 编译命令
 - 3.2.2.4 U-Boot 编译命令
4. 第4章 资源划分
 - 4.1 系统架构
 - 4.1.1 AP + AP 系统架构
 - 4.1.2 AP + MCU 系统架构

- 4.2 Linux Kernel 资源配置
 - 4.2.1 Linux Kernel 配置文件
 - 4.2.1.1 DTS 相关文件
 - 4.2.1.2 AMP 驱动相关文件
 - 4.2.2 Linux Kernel 内存资源
 - 4.2.2.1 运行内存配置
 - 4.2.2.2 共享内存配置
 - 4.2.3 Linux Kernel 外设资源
 - 4.2.3.1 中断配置
 - 4.2.3.2 引脚配置
 - 4.2.3.3 时钟配置
- 4.3 RTOS 资源配置
 - 4.3.1 RT-Thread 配置文件
 - 4.3.1.1 板级相关文件
 - 4.3.1.2 编译相关文件
 - 4.3.2 RT-Thread 内存资源
 - 4.3.2.1 AP 运行内存配置
 - 4.3.2.2 AP 共享内存配置
 - 4.3.2.3 MCU 运行内存配置
 - 4.3.2.4 MCU 共享内存配置
 - 4.3.3 RT-Thread 外设资源
 - 4.3.3.1 AP 中断配置
 - 4.3.3.2 MCU 中断配置
 - 4.3.3.3 引脚配置
 - 4.3.3.4 时钟配置
- 4.4 Bare-metal 资源配置
 - 4.4.1 RK HAL 配置文件
 - 4.4.1.1 板级相关文件
 - 4.4.1.2 编译相关文件
 - 4.4.2 RK_HAL 内存资源
 - 4.4.2.1 AP 运行内存配置
 - 4.4.2.2 AP 共享内存配置
 - 4.4.2.3 MCU 运行内存配置
 - 4.4.2.4 MCU 共享内存配置
 - 4.4.3 RK HAL 外设资源
 - 4.4.3.1 AP 中断配置
 - 4.4.3.2 MCU 中断配置
 - 4.4.3.3 引脚配置
 - 4.4.3.4 时钟配置
- 5. 第5章 启动方案
 - 5.1 Rockchip SoC 处理器架构
 - 5.2 AP + AP 启动方案
 - 5.2.1 Linux + RTOS / Bare-metal
 - 5.2.1.1 示例固件: Kernel + RT-Thread / HAL
 - 5.2.1.2 示例固件: Kernel + 3 * HAL
 - 5.2.2 RTOS + Bare-metal
 - 5.2.2.1 示例固件: HAL + HAL
 - 5.2.2.2 示例固件: RT-Thread + HAL
 - 5.3 AP + MCU 启动方案
 - 5.3.1 Linux + MCU RTOS / Bare-metal
 - 5.3.1.1 示例固件: Kernel + mcu RT-Thread / HAL
 - 5.4 不同存储的启动方案
 - 5.4.1 eMMC / Flash 启动
 - 5.5 快速启动方案
 - 5.5.1 SD卡启动
 - 5.6 快速启动方案
 - 5.6.1 SPL启动方案
 - 5.6.2 双存储启动方案

- 6. 第6章 通信方案
 - 6.1 核间中断触发
 - 6.1.1 Mailbox 中断触发
 - 6.1.2 软件中断触发
 - 6.1.3 SGI 触发
 - 6.2 底层接口方案
 - 6.3 RPMsg 协议方案
 - 6.3.1 标准框架
 - 6.3.2 通信流程
 - 6.3.3 RPMsg 适配
 - 6.3.3.1 Linux Kernel 适配 RPMsg
 - 6.3.3.2 RPMsg-Lite 适配
 - 6.3.3.3 MCU RPMsg-Lite 适配
 - 6.4 RPMsg 编译配置
 - 6.4.1 Kernel + RT-Thread
 - 6.4.2 Kernel + HAL
 - 6.4.3 RT-Thread + HAL
 - 6.5 RPMsg 测试示例
 - 6.5.1 Kernel + RT-Thread
 - 6.5.1.1 共享内存
 - 6.5.1.2 测试 Demo
 - 6.5.1.2.1 Kernel Demo
 - 6.5.1.2.2 RTT Demo
 - 6.5.1.2.3 测试成功log
 - 6.5.2 RT-Thread + HAL
 - 6.5.2.1 共享内存
 - 6.5.2.2 测试demo
 - 6.5.2.2.1 RTT Demo
 - 6.5.2.2.2 HAL Demo
 - 6.5.2.2.3 测试结果
- 7. 第7章 中断
 - 7.1 Cortex-A GIC
 - 7.1.1 GIC 中断配置
 - 7.1.2 GIC 中断服务程序
 - 7.1.2.1 Bare-metal GIC 中断服务程序
 - 7.1.2.2 RTOS GIC 中断服务程序
 - 7.2 Cortex-M NVIC
 - 7.2.1 NVIC 中断初始化
 - 7.2.2 NVIC 中断服务程序
 - 7.2.2.1 Bare-metal GIC 中断服务程序
 - 7.2.2.2 RTOS NVIC 中断服务程序
 - 7.3 RISC-V中断控制器
 - 7.3.1 IPIC 中断初始化
 - 7.3.2 IPIC 中断服务程序
 - 7.3.2.1 Bare-metal GIC 中断服务程序
 - 7.3.2.2 RTOS IPIC 中断服务程序
- 8. 第8章 模块
 - 8.1 eMMC
 - 8.1.1 HAL
 - 8.1.2 RT-Thread
 - 8.1.3 Kernel
 - 8.2 UART
 - 8.2.1 HAL
 - 8.2.2 RT-Thread
 - 8.2.3 Kernel
 - 8.3 SPI FLASH
 - 8.3.1 HAL
 - 8.3.2 RT-Thread

- 8.3.3 Kernel
 - 8.4 GMAC
 - 8.4.1 HAL
 - 8.4.2 RT-Thread
 - 8.4.3 Kernel
 - 8.5 PCIE
 - 8.5.1 HAL / RT-Thread
 - 8.6 CPU Cache ECC
 - 8.6.1 RT-Thread
 - 8.7 DDR ECC
 - 8.7.1 HAL
 - 8.7.2 Kernel
- 9. 第9章 调试
 - 9.1 串口调试
 - 9.1.1 U-Boot 启动输出
 - 9.1.2 RK HAL 启动输出
 - 9.1.3 RT-Thread 启动输出
 - 9.2 AP 使用 OpenOCD 调试
 - 9.2.1 Windows 环境搭建
 - 9.2.1.1 软件安装
 - 9.2.1.2 硬件连接
 - 9.2.2 使用示例
 - 9.3 MCU 使用 Ozone 调试
 - 9.3.1 Windows 环境搭建
- 10. 第10章 演示
 - 10.1 性能测试
 - 10.1.1 测试整型
 - 10.1.2 测试浮点型
 - 10.1.3 测试内存
 - 10.1.4 测试中断响应时间
 - 10.2 实时性演示
 - 10.2.1 测试方法
 - 10.2.2 测试原理
 - 10.2.3 测试结果
- 11. 第11章 附录
 - 11.1 术语
 - 11.2 文档索引

1. 第1章 多核异构系统

1.1 概述

1.1.1 多核异构系统简介

多核异构系统是一种将同一颗 SoC 芯片中不同处理器核心分别独立运行不同平台的计算系统。同时支持 SMP (Symmetric Multi-Processing) 对称多处理系统和 AMP (Asymmetric Multi-Processing) 非对称多处理系统。

多核异构系统将传统平台两套系统合二为一。在传统平台中，Linux 系统和实时系统往往是完全独立的两套系统，需要完整的两颗处理器和两套外围电路。而在多核异构系统中，通过合理的处理器核心、外设等资源划分，同一颗 SoC 芯片就能够独立运行 Linux 系统和实时系统。在满足系统软件功能和硬件外设的丰富性要求的同时，满足系统的实时性要求。

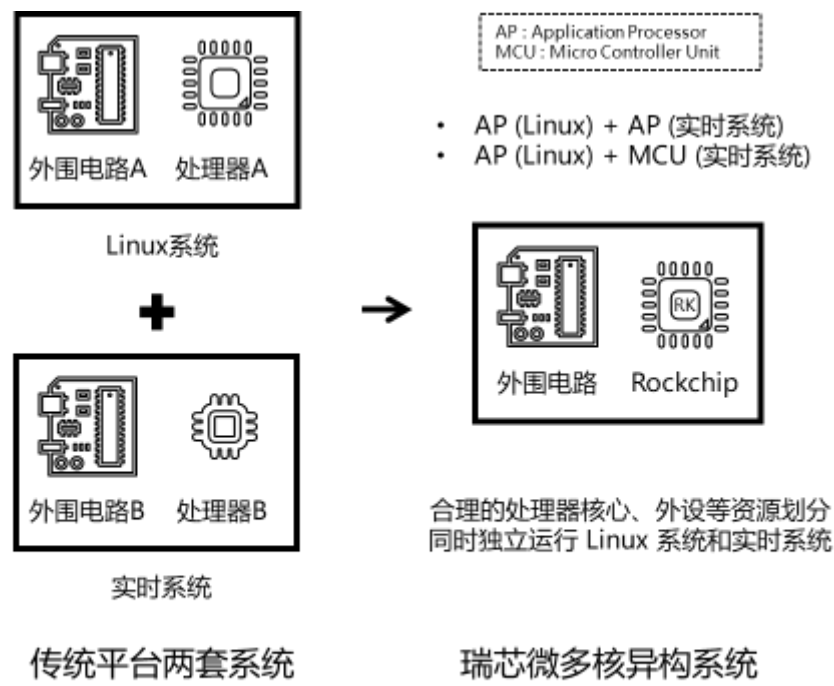


图 1-1-1 多核异构系统将传统平台两套系统合二为一

多核异构系统也支持同一颗 SoC 芯片同时独立运行多个实时系统。

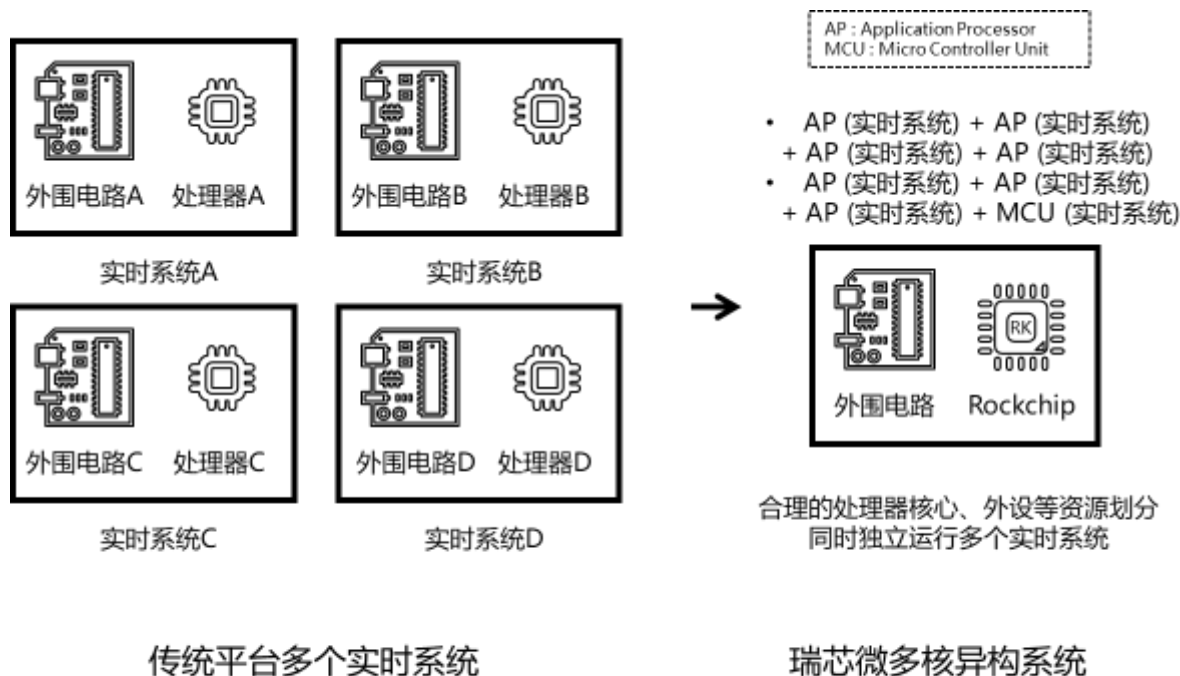


图 1-1-2 同时独立运行多个实时系统

多核异构系统还支持同一颗 SoC 芯片以 SMP + AMP 的方式运行。

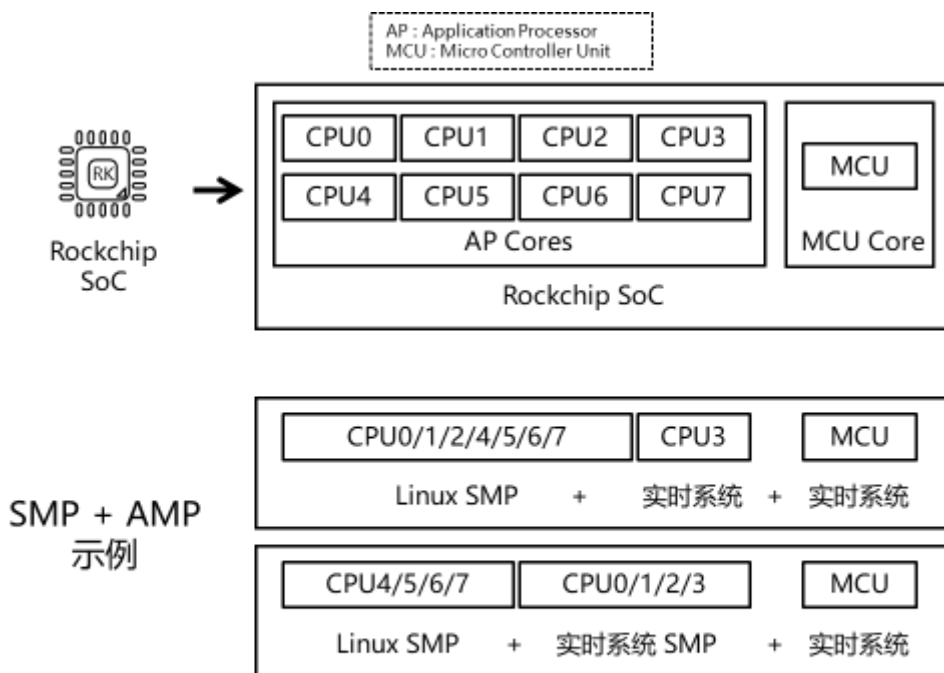


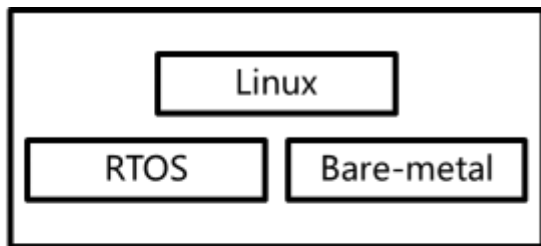
图 1-1-3 以 SMP + AMP 的方式运行

多核异构系统应用于产品设计中，还具有明显的性价比优势和产品体积优势。目前已经广泛应用于电力行业、工控行业、消费电子、汽车电子等产品中。

1.1.2 瑞芯微多核异构系统

瑞芯微多核异构系统是瑞芯微提供的一套通用多核异构系统解决方案。在现有 SMP 对称多处理系统的基础上，增加对 AMP 非对称多处理系统的支持。本文主要对 AMP 方案进行说明。AMP 方案主要包括 AMP 启动方案和 AMP 通信方案。具体实现原理参考本文相关章节。

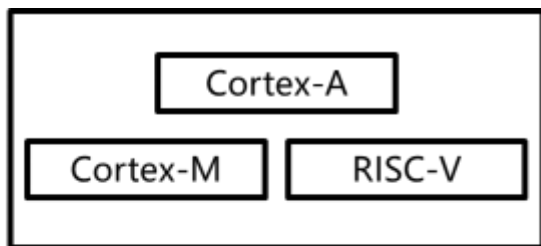
在运行平台方面，Linux 提供标准的 Linux Kernel，RTOS 提供开源的 RT-Thread，Bare-metal 提供基于 RK HAL 硬件抽象层的裸机开发库。同时，瑞芯微多核异构系统支持客户自行适配更多的运行平台，例如可以基于 RK HAL 硬件抽象层适配指定的 RTOS 等。



- Linux：提供标准的 Linux Kernel
- RTOS：提供开源的 RT-Thread
- Bare-metal：提供基于 RK HAL 硬件抽象层的裸机开发库

图 1-2-1 运行平台

在处理器核心方面，瑞芯微多核异构系统支持 SoC 中同构的 ARM Cortex-A 每个处理器核心独立运行。也支持 SoC 中异构的 ARM Cortex-M 或 RISC-V 处理器核心独立运行。瑞芯微多核异构系统通过合理的处理器核心资源划分，将特定的任务分配到最适合的处理器核心进行处理，从而使 SoC 发挥出更优秀的性能和能效表现。



- 支持 SoC 中同构的 ARM Cortex-A 每个处理器核心独立运行
- 支持 SoC 中异构的 ARM Cortex-M 或 RISC-V 核心独立运行

图 1-2-2 处理器核心

目前，瑞芯微多核异构系统主要采用无监督的 AMP 方案。不使用虚拟化管理，从而在运行实时系统时获得更快的中断响应，以满足电力、工控等行业应用中严苛的硬实时性要求。

未来，瑞芯微多核异构系统也会将虚拟化管理作为可选特性。基于 RPMsg 和 RemoteProc 框架，支持标准的 OpenAMP。针对工控行业应用，支持 Type-1 型 Hypervisor 的 Jailhouse。在瑞芯微后续推出的 SoC 芯片中，还将进一步支持硬件资源隔离，增强瑞芯微多核异构系统的灵活性和可靠性。

1.2 平台支持

1.2.1 RK3588

1.2.1.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A76 + 4 x ARM Cortex-A55
MCU	1 x ARM Cortex-M0

1.2.1.2 运行平台支持

处理器核心	ARM Cortex-A76	ARM Cortex-A55	ARM Cortex-M0
Linux 支持	Kernel 5.10	Kernel 5.10	N/A
RTOS 支持	N/A	RTT 3.1-32 RTT 4.1-32	RTT 3.1 RTT 4.1
Bare-metal 支持	N/A	HAL-32	HAL

1.2.2 RK3576

1.2.2.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A72 + 4 x ARM Cortex-A53
MCU	1 x ARM Cortex-M0

1.2.2.2 运行平台支持

处理器核心	ARM Cortex-A72	ARM Cortex-A53	ARM Cortex-M0
Linux 支持	Kernel 6.1	Kernel 6.1	N/A
RTOS 支持	N/A	RTT 4.1-32	RTT 4.1
Bare-metal 支持	N/A	HAL-32	HAL

1.2.3 RK3568

1.2.3.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A55
MCU	1 x RISC-V

1.2.3.2 运行平台支持

处理器核心	ARM Cortex-A55	RISC-V
Linux 支持	Kernel 4.19 Kernel 5.10	N/A
RTOS 支持	RTT 3.1-32	RTT 3.1
Bare-metal 支持	HAL-32	HAL

1.2.4 RK3562

1.2.4.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A53
MCU	1 x ARM Cortex-M0

1.2.4.2 运行平台支持

处理器核心	ARM Cortex-A53	ARM Cortex-M0
Linux 支持	Kernel 5.10	N/A
RTOS 支持	RTT 4.1-32	RTT 4.1
Bare-metal 支持	HAL-32	HAL

1.2.5 RK3358

1.2.5.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A35

1.2.5.2 运行平台支持

处理器核心	ARM Cortex-A35
Linux 支持	N/A
RTOS 支持	RTT 3.1-32
Bare-metal 支持	HAL-32

1.2.6 RK3308

1.2.6.1 处理器核心

处理器类型	处理器核心
AP	4 x ARM Cortex-A35

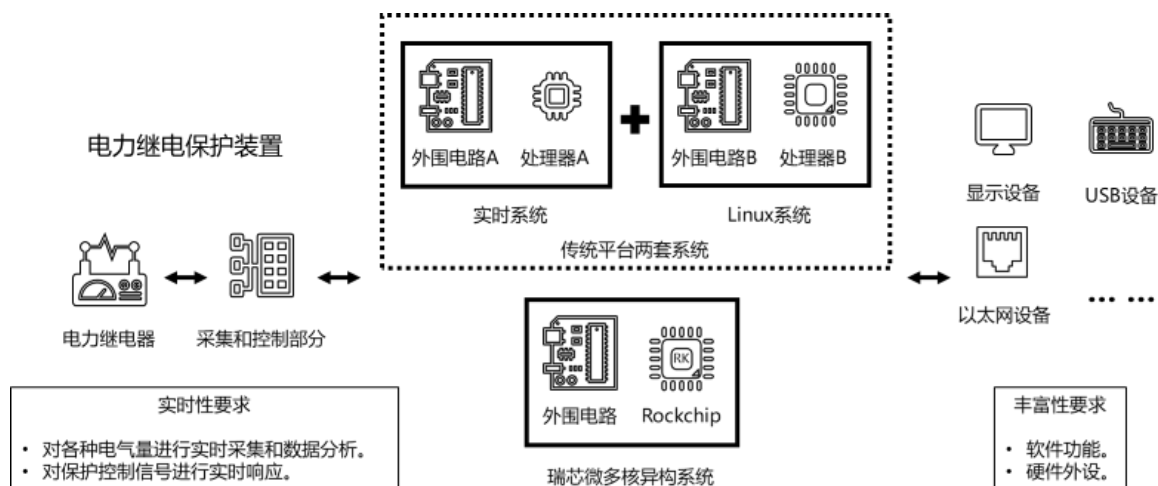
1.2.6.2 运行平台支持

处理器核心	ARM Cortex-A35
Linux 支持	Kernel 5.10
RTOS 支持	RTT 3.1-32 RTT 4.1-32
Bare-metal 支持	HAL-32

1.3 产品案例介绍

1.3.1 AP + AP 案例：电力继电保护装置

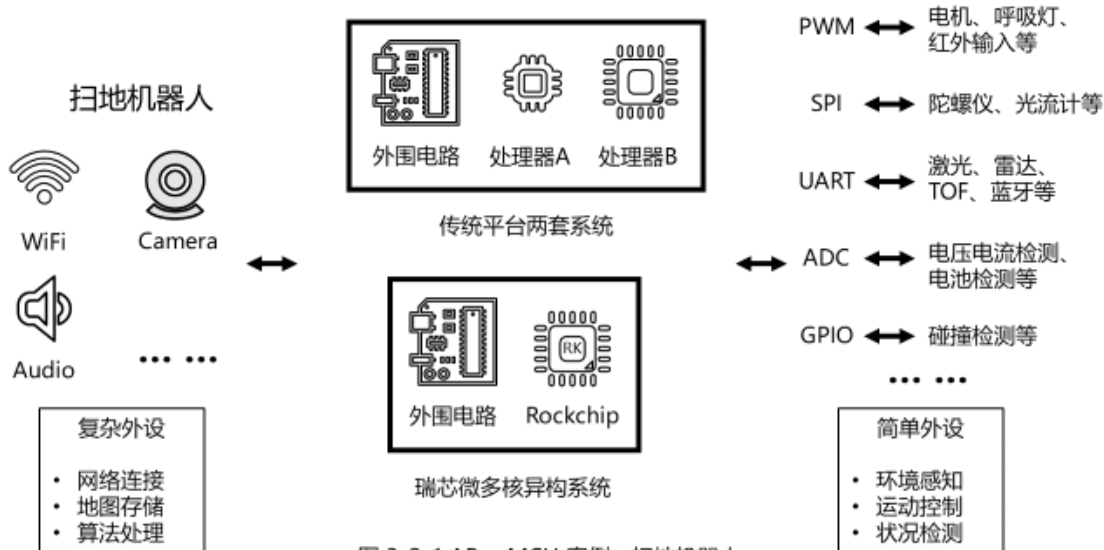
在电力继电保护装置中，既对系统的实时性有要求，例如对各种电气量进行实时采集和数据分析、对保护控制信号进行实时响应等。又对系统的丰富性有要求，需要使用复杂的软件功能和硬件外设，例如显示设备、USB 设备、以太网设备等。使用瑞芯微多核异构系统，将传统平台两套系统合二为一，一套板卡就能同时独立运行 Linux 系统和实时系统，实现上述所有功能。



并且，得益于 AP 的高性能特性，在用于实时系统处理任务时，也能获得运行更高效、算力更强劲的使用体验。

1.3.2 AP + MCU 案例：扫地机器人

在扫地机器人中，既需要运行 Linux 系统，使用复杂外设，例如 WiFi、Camera、Audio 等，实现网络连接、地图存储、算法处理等功能。又需要运行实时系统，使用简单外设，例如 PWM、SPI、UART、ADC、GPIO 等，实现环境感知、运动控制、状况检测等功能。使用瑞芯微多核异构系统，将传统平台两套系统合二为一，省去外挂的 MCU，实现上述所有功能。



并且，使用 SoC 内部的这颗 MCU 作为实时处理器或协处理器，也能让 Linux 系统获得更完整的性能释放。

2. 第2章 AMP SDK

2.1 目录结构

瑞芯微多核异构系统提供的 AMP SDK 源码结构如下。注释中标记 * 的部分为 AMP SDK 主要目录。

```
.
├── app                # Linux 系统用户界面示例
├── buildroot          # Buildroot 系统编译目录
├── debian             # Debian 系统编译目录
├── device             # * AMP SDK 编译脚本和配置文件
├── docs               # * AMP SDK 文档
├── external           # Buildroot 系统补充应用
├── hal                # * Bare-metal 系统编译目录
├── kernel-*           # * 不同版本的 Linux Kernel 编译目录
├── prebuilts          # * 预置的编译工具链
├── rkbin              # * AMP SDK 使用的二进制文件
├── rtos               # * RTOS 系统编译目录
├── tools              # AMP SDK 使用的工具集
├── u-boot             # * U-Boot 编译目录
├── uefi               # uefi 编译目录
└── yocto              # yocto 系统编译目录
```

2.2 device 目录

`device` 目录存放 AMP SDK 的编译脚本和默认配置。以 RK3562 为例，主要文件包括：

```
device/rockchip
├── common
│   ├── build.sh      # AMP SDK 统一编译脚本
│   └── scripts
│       └── mk-amp.sh  # RTOS / Bare-metal 统一编译脚本
└── rk3562
    ├── amp.its        # RTOS + Bare-metal 固件打包配置文件
    ├── amp_linux.its  # Linux + RTOS / Bare-metal 固件打包配置文件
    ├── amp_mcu.its    # Linux + RTOS / Bare-metal MCU 固件打包配置
文件
    └── rockchip_rk3562_xxx_defconfig # 板级编译配置文件
```

相关章节：

[第3章 编译配置](#)

2.3 kernel 目录

`kernel` 目录存放 Linux 系统源码。AMP SDK 提供标准的 Linux Kernel。

各芯片平台支持情况如下：

芯片名称	Kernel 4.19	Kernel 5.10	Kernel 6.1
RK3588		✓	
RK3576			✓
RK3568	✓	✓	
RK3562		✓	
RK3308		✓	

以 RK3562 为例，主要文件包括：

```
kernel
├── arch/arm64/boot/dts/rockchip
│   ├── rk3562-amp.dtsi           # AMP dtsti 文件
│   └── rk3562-xxx-amp.dts       # AMP dts 板级配置文件
├── drivers
│   ├── mailbox                 # Mailbox 核间通信方案
│   ├── rpmsg                  # RPSMsg 核间通信方案
│   └── soc/rockchip
│       └── rockchip_amp.c      # 资源划分、从核生命周期管理等
└── include
```

对于 Linux + RTOS / Bare-metal 的运行方式，运行 Linux 系统的核心必须作为主核心（ Master Core ），负责整个多核异构系统的资源划分和从核心（ Remote Core ）管理。

相关章节：

[第3章 编译配置](#)

[第4章 资源划分](#)

[第5章 启动方案](#)

[第6章 通信方案](#)

2.4 hal 目录

`hal` 目录存放 Bare-metal 系统源码。AMP SDK 提供基于 RK HAL 硬件抽象层的裸机开发库。

RK HAL 是一套基于 [ARM CMSIS](#) （ Cortex Microcontroller Software Interface Standard ）微控制器软件接口标准的硬件抽象层。用户可以直接使用 RK HAL 进行 Bare-metal 裸机系统开发，也可以基于 RK HAL 适配指定的 RTOS。

各芯片平台支持情况如下：

芯片名称	AP HAL-32	MCU HAL
RK3588	✓	✓
RK3576	✓	✓
RK3568	✓	✓
RK3562	✓	✓
RK3358	✓	N/A
RK3308	✓	N/A

以 RK3562 为例，主要文件包括：

```

├── doc
│   └── Rockchip_User_Guide_HAL_CN.md    # RK HAL 开发文档
├── lib
│   ├── bsp                             # 板级支持配置文件
│   ├── CMSIS                           # ARM 微控制器软件接口标准库
│   │   ├── Core                         # MCU 相关文件
│   │   ├── Core_A                       # AP 32位相关文件
│   │   ├── Core_A_64                   # AP 64位相关文件
│   │   └── Device/RK3562
│   │       ├── Include
│   │       │   ├── rk3562.h            # RK3562 寄存器定义
│   │       │   ├── soc.h              # RK3562 芯片相关定义
│   │       │   └── system_rk3562.h
│   │       └── Source/Templates
│   │           ├── GCC                 # 使用 GCC 交叉编译工具链
│   │           │   ├── gcc_arm.ld      # 链接脚本示例
│   │           │   ├── start_m0.S      # MCU 启动文件
│   │           │   └── startup_rk3562.c # AP 启动文件
│   │           ├── mmu_rk3562.c        # AP MMU Map 配置文件
│   │           ├── system_rk3562.c
│   │           └── system_rk3562_mcu.c
│   ├── DSP                             # DSP 相关文件
│   └── RISC-V                           # RISC-V 相关文件
├── hal
│   ├── inc                             # 模块驱动头文件
│   └── src                             # 模块驱动文件
├── middleware                           # Bare-metal 系统中间件
│   ├── benchmark                       # 性能测试
│   ├── rpmsg-lite                      # RPMsg 核间通信方案
│   └── simple_console                  # Console
├── project                              # Bare-metal 系统工程示例
│   ├── common
│   │   └── GCC
│   │       ├── Cortex-A.mk            # AP 通用编译文件
│   │       ├── Cortex-M.mk            # MCU 通用编译文件
│   │       └── riscv.mk                # RISC-V 通用编译文件
│   └── rk3562
│       ├── GCC
│       │   ├── build.sh                # AP 编译脚本
│       │   ├── gcc_arm.ld.S            # AP 链接脚本
│       │   └── Makefile                 # AP 编译文件

```


芯片名称	AP RTT 3.1-32	AP RTT 4.1-32	MCU RTT 3.1	MCU RTT 4.1
RK3588	✓	✓	✓	✓
RK3576		✓		✓
RK3568	✓		✓	
RK3562		✓		✓
RK3358	✓		N/A	N/A
RK3308	✓	✓	N/A	N/A

以 RK3562 为例，RT-Thread V4.1 主要文件包括：

└─ applications	# 公共的应用文件
└─ bsp/rockchip	# 板级支持包
└─ common	# 公共的模块与功能相关文件
└─ drivers	# RTOS Device Driver 文件
└─ fwmgr	# 固件管理相关文件
└─ hal	# RK HAL 文件
└─ test	# 模块与功能测试文件
└─ rk3562-32	# RK3562 AP 32位
└─ applications	# 应用文件
└─ board	# 板级配置文件
└─ driver	# 驱动文件
└─ Image	
└─ amp.img	# 打包后的 RTOS / Bare-metal 固件
└─ amp.its	# RTOS + Bare-metal 固件打包配置文件
└─ amp_linux.its	# Linux + RTOS / Bare-metal 固件打包配置文
件	
└─ rttx.bin	
└─ rttx.elf	
└─ parameter.txt	# 分区表配置文件
└─ smp.its	# RTOS SMP 固件打包配置文件
└─ test	# 测试文件
└─ build.sh	# AP 编译脚本
└─ gcc_arm.ld.S	# AP 链接脚本
└─ hal_conf.h	# RK HAL 配置文件
└─ Kconfig	
└─ mkimage.sh	# AP 固件打包脚本
└─ rtconfig.h	
└─ rtconfig.py	
└─ SConscript	
└─ SConstruct	
└─ rk3562-mcu	# RK3562 MCU
└─ applications	# 应用文件
└─ board	# 板级配置文件
└─ driver	# 驱动文件
└─ Image	
└─ amp.img	# 打包后的 RTOS 系统固件
└─ amp_mcu.its	# Linux + RTOS / Bare-metal MCU 固件打包
配置文件	
└─ mcu.bin	
└─ mcu.elf	
└─ gcc_arm.ld.S	# MCU 链接脚本

```

|   |   |   | hal_conf.h           # RK HAL 配置文件
|   |   |   | Kconfig
|   |   |   | mkimage.sh          # MCU 固件打包脚本
|   |   |   | rtconfig.h
|   |   |   | rtconfig.py
|   |   |   | SConscript
|   |   |   | SConstruct
|   |   |   | tools                # 工具集
|   |   |   |
|   |   |   | components
|   |   |   | examples
|   |   |   | include
|   |   |   | libcpu
|   |   |   | src
|   |   |   | third_party
|   |   |   | tools

```

相关章节：

[第3章 编译配置](#)

[第4章 资源划分](#)

[第5章 启动方案](#)

[第6章 通信方案](#)

2.6 u-boot 目录

`u-boot` 目录存放 U-Boot 源码。AMP SDK 需要增加 `rk-amp.config` 打开如下配置：

```

CONFIG_AMP=y
CONFIG_ROCKCHIP_AMP=y

```

以 RK3562 为例，AMP SDK 相关的主要文件包括：

```

arch/arm/mach-rockchip/rk3562/rk3562.c  # 芯片初始化文件
drivers/cpu/rockchip_amp.c              # AMP 启动方案

include/configs/rk3562_common.h         # 芯片通用配置文件

```

相关章节：

[第3章 编译配置](#)

[第5章 启动方案](#)

2.7 rkbin 目录

`rkbin` 目录存放 AMP SDK 使用的二进制文件，需要与 `u-boot` 目录配合使用。

以 RK3562 为例，AMP SDK 相关的主要文件包括：

bin/rk35/rk3562_b131_xxx.elf	# 通用 b131 文件
bin/rk35/rk3562_b131_cpu3_xxx.elf	# 前级运行在 CPU3 的 b131 文件
RKTRUST/RK3562TRUST.ini	# 通用配置文件
RKTRUST/RK3562TRUST_CPU3.ini	# 前级运行在 CPU3 的配置文件

相关章节：

[第3章 编译配置](#)

[第5章 启动方案](#)

3. 第3章 编译配置

3.1 配置文件

在编译 AMP SDK 前，请先选择并修改相关配置文件。

3.1.1 统一编译配置文件

AMP SDK 编译配置：主要告诉编译脚本，RTOS 和 Bare-metal 的工程位置，以及配套工程的具体配置。AMP SDK 默认配置位于 `<AMP_SDK>/device/rockchip/.chip/rockchip_xxx_defconfig`，相关配置如下：

```
RK_AMP=y                                # AMP RTOS / Bare-metal 支持
RK_AMP_ARCH="arm"                       # RTOS / Bare-metal 使用32位
                                          # 64位使用 "arm64"
RK_AMP_HAL_TARGET="rk3562"              # AP Bare-metal 对应的工程目录名
RK_AMP_RTT_TARGET="rk3562-32"           # AP RTOS 对应的工程目录名
RK_AMP_MCU_HAL_TARGET="rk3562-mcu"      # MCU Bare-metal 对应的工程目录名
RK_AMP_MCU_RTT_TARGET="rk3562-mcu"     # MCU RTOS 对应的工程目录名
RK_AMP_CFG is not set                  # 辅助配置文件
RK_AMP_FIT_ITS="amp_linux.its"          # AMP 固件打包配置文件

RK_UBOOT_CFG_FRAGMENTS="rk-amp"         # 增加 AMP U-Boot config 配置文件
RK_UBOOT_TRUST_INI is not set           # 使用特殊 rkbin 时需要指定的配置文件
件
RK_PARAMETER="parameter.txt"           # 分区表配置文件
```

AMP SDK 配置建议使用 `make menuconfig` 的方式，这种方式可以自动整理编译宏之间的依赖关系，避免生成无效编译配置。

```
-- AMP (Asymmetric Multi-Processing System)
    arch (arm) --->
    (rk3562) HAL target
    (${RK_CHIP_FAMILY}-32) RT-Thread target
    (${RK_CHIP_FAMILY}-mcu) MCU HAL target
    (${RK_CHIP_FAMILY}-mcu) MCU RT-Thread target
    () config file
    (amp.its) FIT ITS file
```

3.1.2 AMP 固件打包配置文件

AMP 固件打包使用标准的 FIT 格式。FIT 格式是 U-Boot 原生支持并且主推的。通过修改 ITS 配置文件，对于每个运行 RTOS / Bare-metal 的 AMP 处理器核心，可以支持丰富的配置。以 RK3562 为例，RK3562 默认有三种 AMP 配置方案：

- `amp_linux.its` : Linux + RTOS / Bare-metal 混合部署方案。
- `amp.its` : RTOS + Bare-metal 混合部署方案。
- `amp_mcu.its` : Linux + MCU RTOS / Bare-metal 混合部署方案。

3.1.2.1 amp_linux.its

Linux + RTOS / Bare-metal 混合部署方案。以下示例为 RK3562 CPU3 独立运行 RTOS，其余处理器核心运行 Linux SMP。

```
/dts-v1/;
/ {
    description = "FIT source file for rockchip AMP";
    #address-cells = <1>;
    images {
        # amp3 节点配置 CPU3，其它处理器核心类似。
        amp3 {
            description = "rtt-core3";           # 固件描述信息
            data = /incbin/("rtt3.bin");         # 指定待打包的固件位置（带路径）
            type = "firmware";                   # AP 设置为 firmware
            compression = "none";                # 保持默认 none
            arch = "arm";                        # 指定处理器的架构
            cpu = <0x3>;                         # 指定处理器的硬件ID
            thumb = <0>;                          # 指定处理器的指令集
            hyp = <0>;                            # 指定处理器是否运行 Hypervisor
            load = <0x01800000>;                  # 指定固件加载和运行地址
            compile {                             # 编译时的配置，编译后自动清除
                size = <0x00800000>;              # 运行内存大小
                sys = "rtt";                      # RTOS (rtt) or Bare-metal (hal)
                core = "ap";                     # 处理器核心类型: ap or mcu
                rtt_config = "board/rk3562_evbl_lp4x/amp_defconfig"
            };
            udelay = <10000>;                     # 启动下一个处理器核心的延时
            hash {
                algo = "sha256";                 # 指定固件完整性校验的算法
            };
        };
    };

    share {                                     # 编译时的共享内存配置，编译后自
动清除
        shm_base = <0x07800000>;                # 共享内存起始地址
        shm_size = <0x00400000>;                # 共享内存大小
        rpmsg_base = <0x07c00000>;              # RPMsg 共享内存起始地址
        rpmsg_size = <0x00500000>;              # RPMsg 共享内存大小
    };

    configurations {
        default = "conf";
        conf {
            description = "Rockchip AMP images";
            rollback-index = <0x0>;
            loadables = "amp3";                 # 指定被加载的固件，以及加载和启动顺
序
            signature {
                algo = "sha256,rsa2048";
            }
        }
    }
};
```

```

        padding = "pss";
        key-name-hint = "dev";
        sign-images = "loadables";
    };
    /* - run linux on cpu0
    * - it is brought up by amp(that run on U-Boot)
    * - it is boot entry depends on U-Boot
    */
    linux {
        # 支持 Linux 混合部署
        description = "linux-os";
        arch = "arm64";
        cpu = <0x000>;
        thumb = <0>;
        hyp = <0>;
        udelay = <0>;
        # AMP 固件加载位置如果与 Linux Kernel 加载位置冲突, 需要进行调整
        load = <0x2000000>; # Linux Kernal 加载位置
        load_c = <0x4880000>; # 压缩的 Linux Kernel 加载位置
    };
};

};

};

```

3.1.2.2 amp.its

RTOS + Bare-metal 混合部署方案。以下示例为 RK3562 CPU1 独立运行 RTOS，其余处理器核心独立运行三个 Bare-metal。

```

/dts-v1/;
/ {
    description = "FIT source file for rockchip AMP";
    #address-cells = <1>;
    images {
        amp0 {
            description = "bare-metal-core0";
            data = /incbin/("rtt0.bin");
            type = "firmware";
            compression = "none";
            arch = "arm";
            cpu = <0x0>;
            thumb = <0>;
            hyp = <0>;
            load = <0x02000000>;
            udelay = <10000>;
            compile {
                size = <0x00800000>;
                sys = "hal";
            };
            hash {
                algo = "sha256";
            };
        };
        amp1 {
            description = "rtt-core1";
            data = /incbin/("rtt1.bin");

```

```

        type          = "firmware";
        compression   = "none";
        arch          = "arm";
        cpu           = <0x1>;
        thumb         = <0>;
        hyp           = <0>;
        load          = <0x00800000>;
        udelay        = <10000>;
        compile {
            size       = <0x00800000>;
            sys        = "rtt";
        };
        hash {
            algo = "sha256";
        };
    };
    amp2 {
        description    = "bare-metal-core2";
        data           = /incbin/("rtt2.bin");
        type          = "firmware";
        compression   = "none";
        arch          = "arm";
        cpu           = <0x2>;
        thumb         = <0>;
        hyp           = <0>;
        load          = <0x01000000>;
        udelay        = <10000>;
        compile {
            size       = <0x00800000>;
            sys        = "hal";
        };
        hash {
            algo = "sha256";
        };
    };
    amp3 {
        description    = "bare-metal-core3";
        data           = /incbin/("rtt3.bin");
        type          = "firmware";
        compression   = "none";
        arch          = "arm";
        cpu           = <0x3>;
        thumb         = <0>;
        hyp           = <0>;
        load          = <0x01800000>;
        udelay        = <10000>;
        compile {
            size       = <0x00800000>;
            sys        = "hal";
        };
        hash {
            algo = "sha256";
        };
    };
};

share {
    shm_base          = <0x07800000>;

```

```

    shm_size      = <0x00400000>;
};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;

        loadables = "amp0", "amp1", "amp2", "amp3";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };
    };
};
};
};

```

3.1.2.3 amp_mcu.its

Linux + MCU RTOS / Bare-metal 混合部署方案。以下示例为 RK3562 AP 运行 Linux SMP，MCU 独立运行 Bare-metal AMP。

```

/dts-v1/;
/ {
    description = "Rockchip AMP FIT Image";
    #address-cells = <1>;
    images {
        mcu {
            description = "mcu";
            data = /incbin/("./rtt.bin");
            type = "standalone";          # MCU 设置为 standalone
            compression = "none";
            arch = "arm";
            load = <0x08200000>;
            udelay = <1000000>;
            compile {
                size = <0x00800000>;
                sys = "hal";
                core = "mcu";             # 处理器核心类型: ap or mcu
            };
            hash {
                algo = "sha256";
            };
        };
    };

    share {
        shm_base = <0x07800000>;
        shm_size = <0x00400000>;
        rpmsg_base = <0x07c00000>;
        rpmsg_size = <0x00500000>;
    };
};

```

```

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;
        loadables = "mcu";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };
    };
};
};
};

```

注意：在使用瑞芯微多核异构系统时，需要对运行内存和共享内存进行合理规划，以避免冲突。

相关章节：

[第4章 资源划分](#)

3.1.3 辅助配置文件

某些 SoC 平台需要额外使用 `<AMP_SDK>/device/rockchip/.chip/$RK_AMP_CFG` 辅助配置文件。辅助配置文件中的参数会被直接 `export` 到环境变量中，辅助进行编译。

例如在 RK3308 中，使用共享 LOG 功能需要增加以下配置：

```

## 第3章 RTT config
## 第3章 Share Memory config
RTT_SHLOG0_SIZE=0x00001000
RTT_SHLOG1_SIZE=0x00001000
RTT_SHLOG2_SIZE=0x00001000
RTT_SHLOG3_SIZE=0x00001000

## 第3章 HAL config
## 第3章 Share Memory config same as RTT
SHLOG0_SIZE=$RTT_SHLOG0_SIZE
SHLOG1_SIZE=$RTT_SHLOG1_SIZE
SHLOG2_SIZE=$RTT_SHLOG2_SIZE
SHLOG3_SIZE=$RTT_SHLOG3_SIZE

```

3.1.4 分区表配置文件

在开发过程中，需要根据实际使用的存储介质容量大小和 AMP 固件大小等调整分区表配置。

手动修改分区表配置文件 `parameter.txt`。分区表配置文件在：

`<AMP_SDK>/device/rockchip/.chip/$RK_PARAMETER`。添加分区格式为 `start@size(part_name)`，单位为 sector（512 Byte）。例如增加一个 2M 的 amp 分区：

```
CMDLINE:
mtdparts=:0x00002000@0x00004000 (uboot),0x00002000@0x00006000 (misc),0x00020000@0x00008000 (boot),0x00001000@0x00028000 (amp),0x00040000@0x00029000 (recovery),0x00010000@0x00069000 (backup),0x01c00000@0x00079000 (rootfs),0x00040000@0x01c79000 (oem),-@0x01cb9000 (userdata:grow)
```

使用脚本插入新增的 **amp** 分区:

```
./build.sh list-parts
=====
Partition table
=====
1:      uboot at 0x00004000 size=0x00002000 (4M)
2:      misc at 0x00006000 size=0x00002000 (4M)
3:      boot at 0x00008000 size=0x00020000 (64M)
4:      recovery at 0x00028000 size=0x00040000 (128M)
5:      backup at 0x00068000 size=0x00010000 (32M)
6:      rootfs at 0x00078000 size=0x01c00000 (14G)
7:      oem at 0x01c78000 size=0x00040000 (128M)
8:      userdata at 0x01cb8000 size=- (grow)

./build.sh insert-part:4:amp:2M
./build.sh list-parts
=====
Partition table
=====
1:      uboot at 0x00004000 size=0x00002000 (4M)
2:      misc at 0x00006000 size=0x00002000 (4M)
3:      boot at 0x00008000 size=0x00020000 (64M)
4:      amp at 0x00028000 size=0x00001000 (2M)
5:      recovery at 0x00029000 size=0x00040000 (128M)
6:      backup at 0x00069000 size=0x00010000 (32M)
7:      rootfs at 0x00079000 size=0x01c00000 (14G)
8:      oem at 0x01c79000 size=0x00040000 (128M)
9:      userdata at 0x01cb9000 size=- (grow)
```

3.2 编译命令

3.2.1 统一编译命令

AMP SDK 统一编译命令如下, 支持一键编译和打包等功能:

```
./build.sh chip      # 选择 SoC 平台
./build.sh lunch     # 选择默认配置文件
./build.sh           # 一键编译打包
./build.sh uboot     # 单独编译 U-Boot
./build.sh kernel    # 单独编译 Linux Kernel
./build.sh amp       # 单独编译 RTOS / Bare-metal
./build.sh cleanall  # 清除所有
./build.sh help      # 获取帮助
```

`./build.sh amp` 会读取 AMP 固件打包配置文件, 自动完成 **amp.img** 的编译和打包。

3.2.2 单独编译命令

可以从 AMP SDK 附带的基础固件中的 `build_info.txt` 获取各个组件的单独编译命令。

3.2.2.1 Linux Kernel 编译命令

Linux Kernel 作为独立组件单独编译时，以 RK3562 AP 为例，参考命令如下：

```
cd <AMP_SDK>/kernel
export ARCH=arm64 # 指定处理器的架构
export CROSS_COMPILE="path to compiler" # 指定编译工具链
## 第3章 例如：
export CROSS_COMPILE=../prebuilts/gcc/linux-x86/aarch64/gcc-arm-10.3-2021.07-
x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-

make rockchip_linux_defconfig # 指定编译配置
make rk3562-evbl-lp4x-v10-linux-amp.img -j8 # 编译指定 dts 板级配置
```

3.2.2.2 RT-Thread 编译命令

RT-Thread 作为独立组件单独编译时，以 RK3562 AP 为例，参考命令如下：

```
cd <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/
./build.sh <cpu_id 0~3 or all>
./mkimage.sh

scons -j8
scons -c
```

3.2.2.3 RK HAL 编译命令

RK HAL 作为独立组件单独编译时，以 RK3562 AP 为例，参考命令如下：

```
cd <AMP_SDK>/hal/project/rk3562/GCC
./build.sh <cpu_id 0~3 or all>
cd ..
./mkimage.sh

make -j8
make clean
```

3.2.2.4 U-Boot 编译命令

U-Boot 作为独立组件单独编译时，以 RK3562 为例，参考命令如下：

```
cd <AMP_SDK>/u-boot/
make rk3562_defconfig rk-amp.config
./make.sh
```

如果需要使用特殊的 rkbin，例如前级运行在 CPU3 上的 rkbin，参考命令如下：

```
cd <AMP_SDK>/u-boot/  
make rk3562_defconfig rk-amp.config  
./make.sh ../rkbin/RKTRUST/RK3562TRUST_CPU3.ini
```

相关章节：

[第5章 启动方案](#)

4. 第4章 资源划分

4.1 系统架构

4.1.1 AP + AP 系统架构

在瑞芯微多核异构系统中，将 AP + AP 系统架构分为 Linux + RTOS / Bare-metal 和 RTOS + Bare-metal 两种。在 Linux + RTOS / Bare-metal 系统架构中，运行 Linux 的处理器核心作为主核（Master Core）。运行 RTOS / Bare-metal 的处理器核心作为从核（Remote Core）。在 RTOS + Bare-metal 系统架构中，第一个启动的处理器核心作为主核（Master Core）。其它处理器核心作为从核（Remote Core）。主核负责整个多核异构系统中共享资源的划分和管理，并运行主站服务程序。

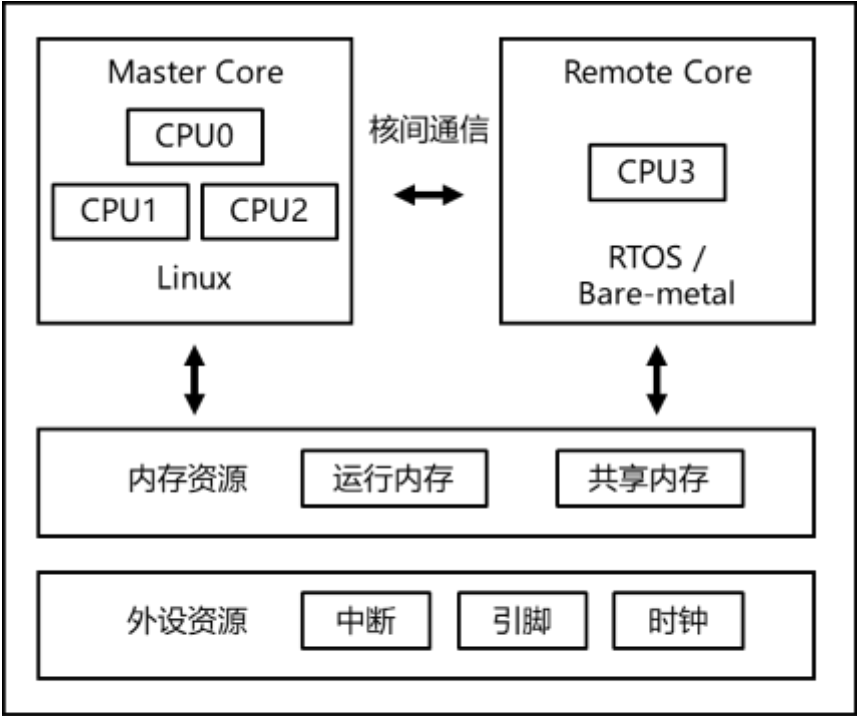


图 4-1-1 Linux + RTOS / Bare-metal 系统架构

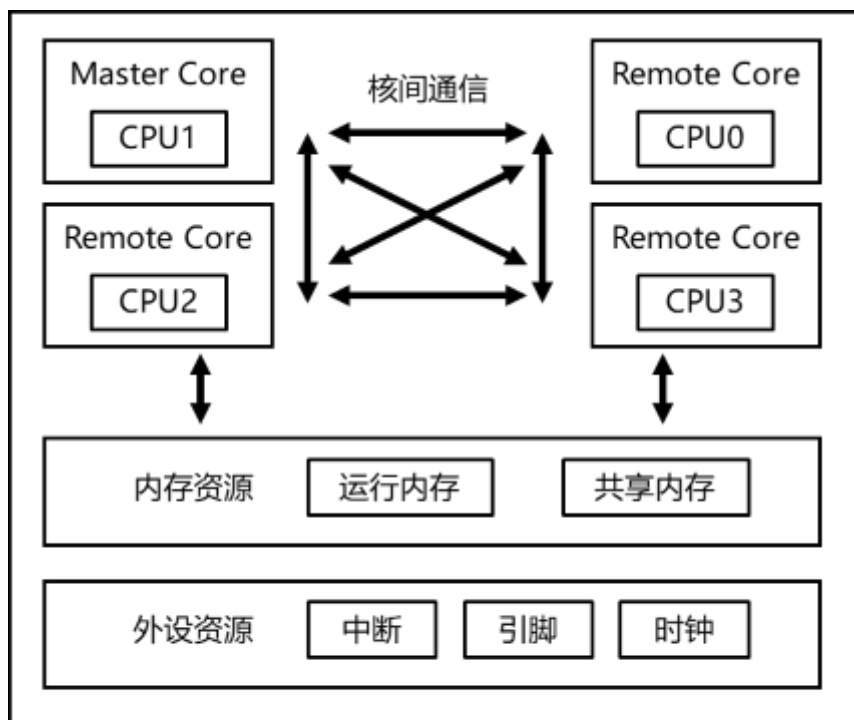


图 4-1-2 RTOS + Bare-metal 系统架构

4.1.2 AP + MCU 系统架构

在瑞芯微多核异构系统中，AP + MCU 系统架构为 Linux + MCU RTOS / Bare-metal。运行 Linux 的 AP 处理器核心作为主核（Master Core）。运行 RTOS / Bare-metal 的 MCU 处理器核心作为从核（Remote Core）。主核负责整个多核异构系统中共享资源的划分和管理，并运行主站服务程序。

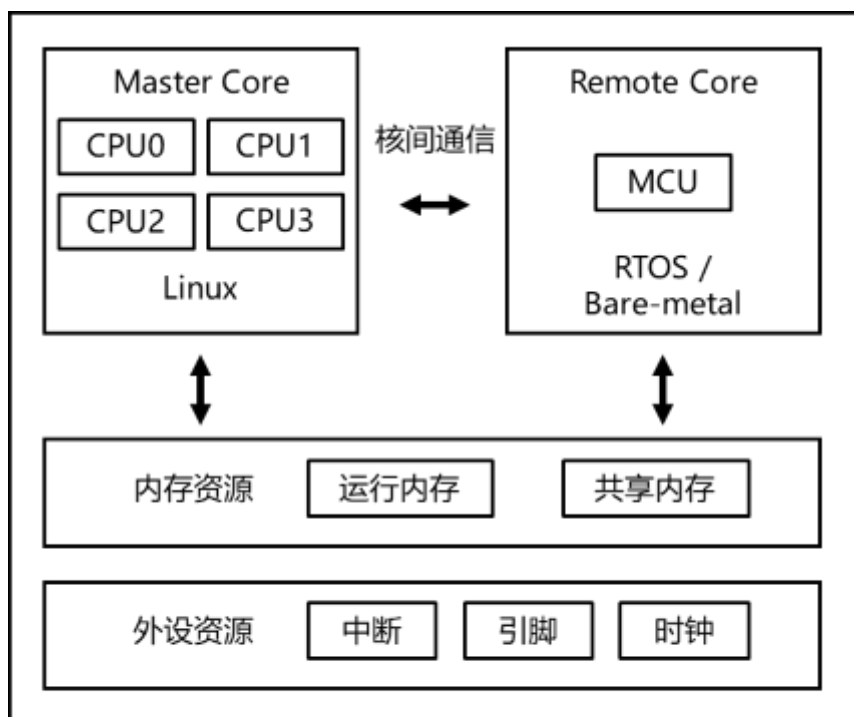


图 4-1-3 Linux + MCU RTOS / Bare-metal 系统架构

4.2 Linux Kernel 资源配置

4.2.1 Linux Kernel 配置文件

4.2.1.1 DTS 相关文件

Linux Kernel 的资源配置在 DTS 文件中，瑞芯微多核异构系统的 DTS 文件命名为 `rkxxxx-evbxxxxx-amp.dts`，并包含 `rkxxxx-amp.dtsi`，例如：

```
arch/arm64/boot/dts/rockchip/rk3562-amp.dtsi
arch/arm64/boot/dts/rockchip/rk3562-evb1-lp4x-v10-linux-amp.dts
```

`rk3562-amp.dtsi` 是 Linux Kernel 用来集中管理多核异构系统的共享资源的。

```
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        /* AMP 用到的时钟资源 */
        clocks = <&cru FCLK_BUS_CM0_CORE>, <&cru CLK_BUS_CM0_RTC>,
                <&cru PCLK_MAILBOX>, <&cru PCLK_INTC>,
                <&cru SCLK_UART7>, <&cru PCLK_UART7>,
                <&cru PCLK_TIMER>, <&cru CLK_TIMER4>, <&cru CLK_TIMER5>;

        /* AMP 用到的引脚资源 */
        pinctrl-names = "default";
        pinctrl-0 = <&uart7m1_xfer>;

        /* AMP 用到的中断资源 */
        amp-cpu-aff-maskbits = /bits/ 64 <0x0 0x1 0x1 0x2 0x2 0x4 0x3
0x8>;

        amp-irqs = /bits/ 64 <GIC_AMP_IRQ_CFG_ROUTE(147, 0xd0,
CPU_GET_AFFINITY(3, 0))>;

        status = "okay";
    };
};
```

4.2.1.2 AMP 驱动相关文件

AMP 驱动相关文件：`<AMP_SDK>/kernel/drivers/soc/rockchip/rockchip_amp.c`

```
// ...

// 集中管理 AMP 的中断资源，由 GIC 驱动调用
static void amp_gic_get_irqs_config(struct device_node *np, struct amp_gic_ctrl_s
*amp_ctrl)
{
    // ....
}

static int rockchip_amp_probe(struct platform_device *pdev)
{

```

```

// ...

// 集中管理 AMP 的时钟资源
rkamp_dev->num_clks = devm_clk_bulk_get_all(&pdev->dev, &rkamp_dev->clks);

// 集中管理 AMP 的供电资源
rkamp_dev->num_pds =
    of_count_phandle_with_args(pdev->dev.of_node, "power-domains",
                                "#power-domain-cells");

// 集中管理 AMP 的核心资源，做核心的生命周期管理，例如开，关，重启等操作
cpus_node = of_get_child_by_name(pdev->dev.of_node, "amp-cpus");
// ...
}

// 引脚资源，在调用 rockchip_amp_probe 前，由 pinctrl 驱动先行处理了。
static const struct of_device_id rockchip_amp_match[] = {
    { .compatible = "rockchip,amp" }, // 和 DTS 文件的属性对应
    // ...
};

```

4.2.2 Linux Kernel 内存资源

4.2.2.1 运行内存配置

DRAM 是系统私有的运行内存。Linux Kernel 默认是将所有 DDR 资源都作为 DRAM 使用，因此多核异构系统中，需要在 Linux Kernel DTS 上，将其他系统的 DRAM 位置保留出来。

以 RK3562 为例：

```

/ {
    reserved-memory {
        /* rk3588-amp.dtsi */
        /* mcu address */
        mcu_reserved: mcu@8200000 {
            reg = <0x0 0x8200000 0x0 0x100000>;
            no-map;
        };

        /* rk3588-evb1-lp4-v10-linux-amp.dts */
        /* ap address */
        amp_reserved: amp@800000 {
            reg = <0x0 0x01800000 0x0 0x00800000>;
            no-map;
        };
    };
};

```

4.2.2.2 共享内存配置

Share Memory 是多系统间交互信息的空间。Linux Kernel 默认是将所有 DDR 资源都作为 DRAM 使用，因此多核异构系统中，需要在 Linux Kernel DTS 上，将 Share Memory 保留出来。

保留的配置操作与 [Linux Kernel DRAM 配置](#) 一致，只要不同名即可。

以 RK3562 为例:

```
/ {
    reserved-memory {
        /* rk3588-amp.dtsi */
        rpmsg_reserved: rpmsg@7c00000 {
            reg = <0x0 0x07c00000 0x0 0x400000>;
            no-map;
        };
    };
};
```

4.2.3 Linux Kernel 外设资源

Linux Kernel 默认将所有芯片资源都定义到了 DTS 中, 因此, 当 AMP 需要在 Linux Kernel 之外使用外设资源, 需要现在 DTS 中关闭对应的模块, 将资源让给 AMP 的其他系统使用。

下面以 RK3562 EVB1 中, Linux Kernel 将 I2C1 资源转让给 RTOS 使用为例:

先找到 `rk3562.dtsi` 中, I2C1 的定义。

```
i2c1: i2c@ffa00000 {                                     /* 模块名字
*/
    compatible = "rockchip,rk3562-i2c", "rockchip,rk3399-i2c";
    reg = <0x0 0xffa00000 0x0 0x1000>;
    clocks = <&cru CLK_I2C1>, <&cru PCLK_I2C1>;           /* 时钟引用
*/
    clock-names = "i2c", "pclk";
    interrupts = <GIC_SPI 13 IRQ_TYPE_LEVEL_HIGH>;      /* 中断引用
*/
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1m0_xfer>;                          /* 引脚引用
*/
    #address-cells = <1>;
    #size-cells = <0>;
    status = "disabled";
};
```

从 DTS 中, I2C1 的资源配置中, 可以看到 I2C1 需要用到:

- 中断资源
- 引脚资源
- 时钟资源

先在 DTS 中将 I2C1 资源关闭:

```
&i2c1 {
    status = "disabled";
};
```

4.2.3.1 中断配置

将 I2C1 的中断资源，加到 `rockchip-amp` 的 `amp-irqs` 节点中：

```
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        // .....
        /* AMP 用到的中断资源 */
        -       amp-irqs = /bits/ 64 <GIC_AMP_IRQ_CFG_ROUTE(147, 0xd0,
CPU_GET_AFFINITY(3, 0))>;
        +       amp-irqs = /bits/ 64 <GIC_AMP_IRQ_CFG_ROUTE(147, 0xd0,
CPU_GET_AFFINITY(3, 0))
        +                               GIC_AMP_IRQ_CFG_ROUTE(45, 0xd0,
CPU_GET_AFFINITY(3, 0))>;
                                                // 新增 I2C1: 45 = I2C1 中断 13 + 固定偏移 32
        // .....
    };
};
```

注意：模块中断号，和 `amp-irqs` 引用中断号差一个固定偏移32。

4.2.3.2 引脚配置

将 I2C1 的引脚资源，加到 `rockchip-amp` 节点中：

```
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        // .....
        /* AMP 用到的引脚资源 */
        pinctrl-names = "default";
        -       pinctrl-0 = <&uart7m1_xfer>;
        +       pinctrl-0 = <&uart7m1_xfer>, <&i2c1m0_xfer>;
        // .....
    };
};
```

注意：模块使用的引脚资源可能有多组，选择实际使用的组别进行添加。

4.2.3.3 时钟配置

将 PWM1 的中断资源，加到 `rockchip-amp` 节点中：

```
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        // .....
        /* AMP 用到的时钟资源 */
        clocks = <&cru FCLK_BUS_CM0_CORE>, <&cru CLK_BUS_CM0_RTC>,
                <&cru PCLK_MAILBOX>, <&cru PCLK_INTCT>,
                <&cru SCLK_UART7>, <&cru PCLK_UART7>,
```

```

-             <&cru PCLK_TIMER>, <&cru CLK_TIMER4>, <&cru CLK_TIMER5>;
+             <&cru PCLK_TIMER>, <&cru CLK_TIMER4>, <&cru CLK_TIMER5>,
+             <&cru CLK_I2C1>, <&cru PCLK_I2C1>;
// .....
};
};

```

4.3 RTOS 资源配置

4.3.1 RT-Thread 配置文件

通常来说，芯片级别的工程中，会预设好几个板级配置，使用工程中，通过 `scons--menuconfig` 指令修改 `CONFIG_RT_BOARD_NAME` 配置，达到同一个工程适配多个板子的需求。

因此，RT-Thread 配置的内容包括：

```

<AMP_SDK>/internal/rk3588/rtos/bsp/rockchip/rk3562-32/
├── applications
├── board
│   ├── common                # 通用配置
│   ├── Kconfig
│   ├── rk3562_evb1_lp4x      # 板级配置
│   └── SConscript
├── build.sh                  # 编译脚本，内含一些编译配置
└── ...

```

- 通用配置部分：芯片的基础配置，为工程必选代码，为所有板级工程服务。
- 板级配置部分：板级配置，为具体板子配置相关功能，比如 GPIO、UART、I2C 等。
- 编译命令：编译命令可以实现编译时的系统传参，提供灵活的编译指令。有 `build.sh` 脚本时，参数可以直接定义在 `build.sh` 中。或者 `export` 特定编译参数后，使用 `scons` 命令编译。

4.3.1.1 板级相关文件

RT-Thread 工程资源配置，包括通用配置，以及板级配置。

通用配置文件包括：

```

## 第4章 通用配置
<AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/common/board_base.c
<AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/common/iomux_base.c

```

通用配置，定义了硬件启动的初始化顺序和默认硬件资源同时提供大量的 `RT_WEAK` 定义，方便用户在板级配置中进行对应的替换。

```
RT_WEAK const struct clk_init clk_inits[];           // 弱定义结构体，可以在板级配置
中重定义
RT_WEAK void rt_hw_iomux_config(void);              // 弱定义函数，可以在板级配置中
重定义
// ...                                             // 不同芯片的弱定义资源不一致，
不——列举

void rt_hw_board_init(void)
{
    // ... 初始化顺序，不要修改
}
```

板级配置文件包括：

```
## 第4章 板级配置
<AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/rk3562_evb1_lp4x/board.c
<AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/rk3562_evb1_lp4x/iomux.c
```

板级配置中，对需要的 `RT_WEAK` 资源进行重定义。

```
// <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/rk3562_evb1_lp4x/iomux.c

void rt_hw_iomux_config(void)
{
    // ...
}
```

4.3.1.2 编译相关文件

RT-Thread 使用 `SCONS` 编译，编译涉及的文件有：

```
cd <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/

rk3562-32/rtconfig.h      # 编译使用的配置文件
rtconfig.py               # 编译脚本，决定编译链位置，编译命令的传入参数等
SConscript                # SCONS 链接文件
SConstruct                # SCONS 链接文件

build.sh                  # RT-Thread AP Core 快捷编译脚本，MCU Core 没有此
脚本
```

AP Core 使用 `./build.sh` 完成编译，MCU Core，使用 `make` 命令完成编译。

4.3.2 RT-Thread 内存资源

RT-Thread 的内存分配，由编译的链接文件分配。

```
<AMP_SDK>/rtos/bsp/rockchip/rk3562-32/gcc_arm.ld.S
<AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/gcc_link.ld.S
```

4.3.2.1 AP 运行内存配置

```
/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/gcc_arm.ld.S */

MEMORY
{
    SRAM    (rxw) : ORIGIN = 0xfe480000, LENGTH = 64K           /* SYSTEM SRAM */
    DRAM    (rxw) : ORIGIN = FIRMWARE_BASE, LENGTH = DRAM_SIZE /* DRAM */
    SHMEM   (rxw) : ORIGIN = SHMEM_BASE, LENGTH = SHMEM_SIZE    /* shared memory
for all cpu */
    LINUX_RPMSG (rxw) : ORIGIN = LINUX_RPMSG_BASE, LENGTH = LINUX_RPMSG_SIZE
}

## 第4章 cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/rtconfig.py
## 第4章 做变量转换, 连接 build.sh 脚本和 gcc_arm.ld.S 文件
CFLAGS += ' -DFIRMWARE_BASE={a} -DDRAM_SIZE={b} -DSHMEM_BASE={c} -DSHMEM_SIZE={d}
-DLINUX_RPMSG_BASE={e} -DLINUX_RPMSG_SIZE={f}'.format(a=PRMEM_BASE, b=PRMEM_SIZE,
c=SHMEM_BASE, d=SHMEM_SIZE, e=LINUX_RPMSG_BASE, f=LINUX_RPMSG_SIZE)

## 第4章 cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/build.sh
export RTT_PRMEM_BASE=$(eval echo \${CPU$1}_MEM_BASE)           /* DRAM 起始位置 */
export RTT_PRMEM_SIZE=$(eval echo \${CPU$1}_MEM_SIZE)           /* DRAM 容量大小 */
export RTT_SHMEM_BASE=0x07800000                                /* shared memory
起始位置 */
export RTT_SHMEM_SIZE=0x00400000                                /* shared memory
容量大小 */
export LINUX_RPMSG_BASE=0x07c00000                              /* rpmsg 起始位置
*/
export LINUX_RPMSG_SIZE=0x00500000                              /* rpmsg 容量大小
*/
```

RT-Thread AP Core 的内存配置, 处于 `gcc_arm.ld.S` 中, 为了方便编译配置, 部分参数, 通过 `rtconfig.py` 转化到 `build.sh` 中, 方便编译修改。

AP Core 中配置的地址信息, 即使真实 DDR 的物理地址信息。

DRAM 可以通过 `build.sh` 中的参数进行配置:

```
CPU3_MEM_BASE=0x01800000
CPU3_MEM_SIZE=0x00800000
export RTT_PRMEM_BASE=$(eval echo \${CPU$1}_MEM_BASE)           /* DRAM 起始位置
*/
export RTT_PRMEM_SIZE=$(eval echo \${CPU$1}_MEM_SIZE)           /* DRAM 容量大小
*/
```

以上变量, 对应 `gcc_arm.ld.S` 的 DRAM 区域:

```
DRAM    (rxw) : ORIGIN = FIRMWARE_BASE, LENGTH = DRAM_SIZE    /* DRAM */
```

变量名的不同, 通过 `rtconfig.py` 转化。

4.3.2.2 AP 共享内存配置

以共享内存 LINUX_RPMSG 为例子，需要在 `gcc_arm.ld.S` 中定义以下内容：

```
MEMORY {
    LINUX_RPMSG (rxw) : ORIGIN = LINUX_RPMSG_BASE, LENGTH = LINUX_RPMSG_SIZE
}

.linux_share_rpmsg (NOLOAD):
{
    PROVIDE(__linux_share_rpmsg_start__ = .);
    . += LINUX_RPMSG_SIZE;
    PROVIDE(__linux_share_rpmsg_end__ = .);
} > LINUX_RPMSG
```

可以在代码中，直接获取到 LINUX_RPMSG 的物理地址指针

```
/* <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/common/rpmsg_base.h */

/* RPMSG share memory infomation */
extern uint32_t __share_rpmsg_start__[];
extern uint32_t __share_rpmsg_end__[];
#define RPMSG_MEM_BASE ((uint32_t)&__share_rpmsg_start__)
#define RPMSG_MEM_END ((uint32_t)&__share_rpmsg_end__)
```

4.3.2.3 MCU 运行内存配置

```
/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/gcc_link.ld.S */

MEMORY
{
    DDR (rxw) : ORIGIN = 0x00000000, LENGTH = 512K /* DRAM */
}

/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/Image/amp.its */
/ {
    images {
        mcu {
            //...
            load = <0x08200000>;
            //...
        };
    };
};
```

RT-Thread MCU Core 的内存配置，由 `gcc_link.ld.S` 和 `amp.its` 共同完成。

MCU Core 较 AP Core 不同的地方在于，MCU Core 的启动位置，就是 MCU Core 的 0 地址。所以，MCU Core 看到的地址，和真实物理地址间存在一个固定偏移。

```
/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/gcc_link.ld.S */

MEMORY
```

```

{
    DDR    (rxw) : ORIGIN = 0x00000000, LENGTH = 512K          /* DRAM */
}

/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/Image/amp.its */
/ {
    images {
        mcu {
            //...
            load          = <0x08200000>;
            //...
        };
    };
};
};

```

以上展示的，就是将 RT-Thread MCU DRAM 设置在物理地址 `0x08200000`，容量大小位 512K 的例子。

4.3.2.4 MCU 共享内存配置

以向 RK3562 RT-Thread MCU 添加共享内存 LINUX_RPMSG 为例子，需要在 `gcc_arm.ld.S` 中定义以下内容：

```

/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/gcc_link.ld.S */

MEMORY {
    // ...
    LINUX_RPMSG (rxw) : ORIGIN = 0x00100000, LENGTH = 0x00500000
}

//...
.linux_share_rpmsg (NOLOAD):
{
    PROVIDE(__linux_share_rpmsg_start__ = .);
    . += LINUX_RPMSG_SIZE;
    PROVIDE(__linux_share_rpmsg_end__ = .);
} > LINUX_RPMSG

/* cat <AMP_SDK>/rtos/bsp/rockchip/rk3562-mcu/Image/amp.its */
/ {
    images {
        mcu {
            //...
            load          = <0x08200000>;
            //...
        };
    };
};
};

```

以上例子中，LINUX_RPMSG 的物理地址应该是 `0x08300000 = 0x08200000 + 0x00100000`。容量大小为 0x00500000 bytes (5M bytes)。

代码中，同样可以获取 LINUX_RPMSG 信息。需要注意，MCU 中，所有的地址信息，都是以自生加载地址为偏移的。

```

/* RPMSG share memory infomation */
extern uint32_t __share_rpmsg_start__[];
extern uint32_t __share_rpmsg_end__[];
#define RPMSG_MEM_BASE      ((uint32_t)&__share_rpmsg_start__)      /* 0x00100000
*/
#define RPMSG_MEM_END       ((uint32_t)&__share_rpmsg_end__)        /* 0x00500000
*/

```

Zain: Isf: RISC-V 是否一致。

4.3.3 RT-Thread 外设资源

以 RK3562 添加 I2C1 资源为例，说明如何在RT-Thread中添加一个外设模块。

4.3.3.1 AP 中断配置

Zain: Isf: 增加 中断章节 跳转

如果使用的是 **MCU** 核心，请调过此章节。**MCU** 使用的是独立的 **NVIC** 控制器，不需要配置此项。

向 `irqsConfig` 中，声明需要响应 I2C1 的中断。

```

// <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/common/board_base.c

// 系统运行在 CPU3 上，则 CPU3 需要响应 I2C1 中断。
#define CUR_CPU      3
static struct GIC_AMP_IRQ_INIT_CFG irqConfig[] =
{
    // ...
    GIC_AMP_IRQ_CFG_ROUTE(I2C1_IRQn, 0xd0, CPU_GET_AFFINITY(CUR_CPU, 0)),
    // ...
}

```

4.3.3.2 MCU 中断配置

与 MCU 直连的中断走 NVIC 的接口，其余的中断需要走 INTMUX 接口。

```

<AMP_SDK>/hal/project/rk3562/src/test_demo.c

// NVIC 接口示例
/*****
/*
/*                               */
/*          SOFTIRQ_TEST          */
/*                               */
*****/
#ifdef SOFTIRQ_TEST
static void soft_isr(void)
{
    printf("softirq_test: enter isr\n");
}

```

```

static void softirq_test(void)
{
    printf("softirq_test start\n");
    HAL_NVIC_SetIRQHandler(RSVD0_MCU_IRQn, soft_isr);
    HAL_NVIC_EnableIRQ(RSVD0_MCU_IRQn);

    HAL_DelayMs(4000);
    HAL_NVIC_SetPendingIRQ(RSVD0_MCU_IRQn);
}
#endif

// INTMUX 接口示例
/*****
/*
/*          GPIO_TEST          */
/*
/*          */
*****/
#ifdef GPIO_TEST

    //.....

static void gpio_test(void)
{
    //.....

    /* Test GPIO interrupt */
    HAL_GPIO_SetPinDirection(GPIO1, GPIO_PIN_B7, GPIO_IN);
    HAL_INTMUX_SetIRQHandler(GPIO1_IRQn, gpio1_isr, NULL);
    HAL_IRQ_HANDLER_SetGpioIRQHandler(GPIO_BANK1, GPIO_PIN_B7, b7_call_back,
    NULL);
    HAL_INTMUX_EnableIRQ(GPIO1_IRQn);
    HAL_GPIO_SetIntType(GPIO1, GPIO_PIN_B7, GPIO_INT_TYPE_EDGE_BOTH);
    HAL_GPIO_EnableIRQ(GPIO1, GPIO_PIN_B7);
    printf("test_gpio interrupt ready\n");
}
#endif

```

4.3.3.3 引脚配置

向 `rt_hw_iomux_config` 中，初始化 I2C1 引脚配置。

```

// <AMP_SDK>/rtos/bsp/rockchip/rk3562-32/board/rk3562_evb1_lp4x/iomux.c

void i2c1_m0_iomux_config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK0,
                          GPIO_PIN_B3 | GPIO_PIN_B4,
                          PIN_CONFIG_MUX_FUNC1);
}

void rt_hw_iomux_config(void)
{
    // ...
}

```

```
i2c1_m0_iomux_config();  
// ...  
}
```

4.3.3.4 时钟配置

RT-Thread 中自带 `CRU` 模块，不需要再进行时钟开关配置。

至此，即可使用 RT-Thread 的 I2C 接口，对 I2C1 进行操作。

4.4 Bare-metal 资源配置

4.4.1 RK HAL 配置文件

4.4.1.1 板级相关文件

RK HAL 所有资源都是直接定义于 `main.c` 中，用户可以自行划分配置方式。

4.4.1.2 编译相关文件

RT-Thread 使用 `SCONS` 编译，编译涉及的文件有：

```
cd <AMP_SDK>/hal/project/rk3562/GCC  
  
Makefile                # 编译使用的配置文件  
build.sh                # RT-Thread AP Core 快捷编译脚本，MCU Core 没有此  
脚本
```

AP Core 使用 `./build.sh` 完成编译，MCU Core，使用 `make` 命令完成编译。

4.4.2 RK_HAL 内存资源

HAL 的内存分配，由编译的链接文件分配。

```
<AMP_SDK>/hal/project/rk3562/GCC/gcc_arm.ld.S  
<AMP_SDK>/hal/project/rk3562-mcu/GCC/gcc_bus_m0.ld
```

4.4.2.1 AP 运行内存配置

```
/* cat <AMP_SDK>/hal/project/rk3562/GCC/gcc_arm.ld.S */  
  
MEMORY  
{  
    SRAM    (rxw) : ORIGIN = SRAM_BASE, LENGTH = SRAM_SIZE           /* SYSTEM  
SRAM */  
    DRAM    (rxw) : ORIGIN = FIRMWARE_BASE, LENGTH = DRAM_SIZE      /* DRAM */
```

```

    SHMEM (rxw) : ORIGIN = SHMEM_BASE, LENGTH = SHMEM_SIZE      /* shared memory
for all cpu */
    LINUX_RPMSG (rxw) : ORIGIN = LINUX_RPMSG_BASE, LENGTH = LINUX_RPMSG_SIZE
}

## 第4章 cat <AMP_SDK>/hal/lib/CMSIS/Device/RK3562/Source/Templates/mmu_rk3562.c
## 第4章 做内存映射, 连接 build.sh 脚本和 gcc_arm.ld.S 文件
#if defined(NC_MEM_BASE) && defined(NC_MEM_SIZE)
    MMU_TTSection(MMUTable, FIRMWARE_BASE, (DRAM_SIZE - NC_MEM_SIZE) >> 20,
Sect_Normal);
    MMU_TTSection(MMUTable, NC_MEM_BASE, NC_MEM_SIZE >> 20, Sect_Normal_NC);
#else
    MMU_TTSection(MMUTable, FIRMWARE_BASE, DRAM_SIZE >> 20, Sect_Normal);
#endif
    MMU_TTSection(MMUTable, SHMEM_BASE, SHMEM_SIZE >> 20, Sect_Normal_SH);
//    MMU_TTSection(MMUTable, SHMEM_BASE, SHMEM_SIZE >> 20, Sect_Normal_NC_SH);
#ifdef LINUX_RPMSG_BASE
    MMU_TTSection(MMUTable, LINUX_RPMSG_BASE, LINUX_RPMSG_SIZE >> 20,
Sect_Normal_NC_SH);
#endif

## 第4章 cat <AMP_SDK>/hal/project/rk3562/GCC/build.sh
export FIRMWARE_CPU_BASE=$(eval echo \${CPU$1_MEM_BASE})      /* DRAM 起始位置
*/
export DRAM_SIZE=$(eval echo \${CPU$1_MEM_SIZE})              /* DRAM 容量大小
*/
export SHMEM_BASE=0x07800000                                  /* shared
memory 起始位*/
export SHMEM_SIZE=0x00400000                                  /* shared memory
容量大小 */
export LINUX_RPMSG_BASE=0x07c00000                            /* rpmsg 起始位置
*/
export LINUX_RPMSG_SIZE=0x00500000                            /* rpmsg 容量大小
*/

```

HAL AP Core 的内存配置, 处于 `gcc_arm.ld.S` 中, 为了方便编译配置, 部分参数, 通过 `mmu_rk3562.c` 转化到 `build.sh` 中, 方便编译修改。

AP Core 中配置的地址信息, 即使真实 DDR 的物理地址信息。

DRAM 可以通过 `build.sh` 中的参数进行配置:

```

CPU3_MEM_BASE=0x01800000
CPU3_MEM_SIZE=0x00800000
export FIRMWARE_CPU_BASE=$(eval echo \${CPU$1_MEM_BASE})      /* DRAM 起始位
置 */
export DRAM_SIZE=$(eval echo \${CPU$1_MEM_SIZE})              /* DRAM 容量大
小 */

```

以上变量, 对应 `gcc_arm.ld.S` 的 DRAM 区域:

```

DRAM (rxw) : ORIGIN = FIRMWARE_BASE, LENGTH = DRAM_SIZE      /* DRAM */

```

变量名的不同, 在 `mmu_rk3562.c` 中转化。

4.4.2.2 AP 共享内存配置

以共享内存 LINUX_RPMSG 为例子，需要在 `gcc_arm.ld.s` 中定义以下内容：

```
MEMORY {
    LINUX_RPMSG (rxw) : ORIGIN = LINUX_RPMSG_BASE, LENGTH = LINUX_RPMSG_SIZE
}

.linux_share_rpmsg (NOLOAD):
{
    PROVIDE(__linux_share_rpmsg_start__ = .);
    . += LINUX_RPMSG_SIZE;
    PROVIDE(__linux_share_rpmsg_end__ = .);
} > LINUX_RPMSG
```

可以在代码中，直接获取到 LINUX_RPMSG 的物理地址指针

```
/* <AMP_SDK>/hal/project/rk3562/src/test_demo.c */

extern uint32_t __linux_share_rpmsg_start__[];
extern uint32_t __linux_share_rpmsg_end__[];

#define RPMSG_LINUX_MEM_BASE ((uint32_t)&__linux_share_rpmsg_start__)
#define RPMSG_LINUX_MEM_END  ((uint32_t)&__linux_share_rpmsg_end__)
#define RPMSG_LINUX_MEM_SIZE (2UL * RL_VRING_OVERHEAD)
```

4.4.2.3 MCU 运行内存配置

```
/* cat <AMP_SDK>/hal/project/rk3562-mcu/GCC/gcc_bus_m0.ld */

MEMORY
{
    DDR (rxw) : ORIGIN = 0x00000000, LENGTH = 512K /* DRAM */
}

/* cat <AMP_SDK>/hal/project/rk3562-mcu/Image/amp.its */
/ {
    images {
        mcu {
            //...
            load = <0x08200000>;
            //...
        };
    };
};
```

RT-Thread MCU Core 的内存配置，由 `gcc_bus_m0.ld` 和 `amp.its` 共同完成。

MCU Core 较 AP Core 不同的地方在于，MCU Core 的启动位置，就是 MCU Core 的 0 地址。所以，MCU Core 看到的地址，和真实物理地址间存在一个固定偏移。

4.4.2.4 MCU 共享内存配置

以向 RK3562 HAL MCU 添加共享内存 LINUX_RPMSG 为例子，需要在 gcc_bus_m0.ld 中定义以下内容：

```
/* cat <AMP_SDK>/hal/project/rk3562-mcu/GCC/gcc_bus_m0.ld */

MEMORY {
    // ...
    LINUX_RPMSG (rxw) : ORIGIN = 0x00100000, LENGTH = 0x00500000
}

//...
.linux_share_rpmsg (NOLOAD):
{
    PROVIDE(__linux_share_rpmsg_start__ = .);
    . += LINUX_RPMSG_SIZE;
    PROVIDE(__linux_share_rpmsg_end__ = .);
} > LINUX_RPMSG

/* cat <AMP_SDK>/hal/project/rk3562-mcu/Image/amp.its */
/ {
    images {
        mcu {
            //...
            load          = <0x08200000>;
            //...
        };
    };
};
```

以上例子中，LINUX_RPMSG 的物理地址应该是 `0x08300000 = 0x08200000 + 0x00100000`。容量大小为 0x00500000 bytes (5M bytes)。

代码中，同样可以获取 LINUX_RPMSG 信息。需要注意，MCU 中，所有的地址信息，都是以自生加载地址为偏移的。

4.4.3 RK HAL 外设资源

4.4.3.1 AP 中断配置

RK HAL 所有资源都是直接定义于 `main.c` 中。以 RK3562 添加 I2C1 资源为例，说明如何在 RK HAL 中添加一个外设模块。

```
/* ----- I2C1 中断配置 ----- */
/* RK HAL bare CORE*/
static struct GIC_AMP_IRQ_INIT_CFG irqConfig[] = {
    GIC_AMP_IRQ_CFG_ROUTE(I2C1_IRQn, 0xd0, CPU_GET_AFFINITY(3, 0)),
    GIC_AMP_IRQ_CFG_ROUTE(0, 0, CPU_GET_AFFINITY(1, 0)), /* sentinel */
};

static struct GIC_IRQ_AMP_CTRL irqConfig = {
    .cpuAff = CPU_GET_AFFINITY(1, 0),
    .defPrio = 0xd0,
```

```

        .defRouteAff = CPU_GET_AFFINITY(1, 0),
        .irqsCfg = &irqsConfig[0],
};

/* ----- I2C1 引脚资源 ----- */
static void HAL_IOMUX_I2C1M0_Config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK0,
                          GPIO_PIN_B3 | GPIO_PIN_B4,
                          PIN_CONFIG_MUX_FUNC1);
}

void main(void)
{
    uint32_t freq;
    struct I2C_HANDLE instance;

    /* HAL BASE Init */
    HAL_Init();
    /* BSP Init */
    BSP_Init();
    /* Interrupt Init */
    HAL_GIC_Init(&irqConfig);

    HAL_IOMUX_I2C1M0_Config();
    freq = HAL_CRU_ClkGetFreq(g_i2c1dev.clkID);
    HAL_I2C_Init(&instance, g_i2c1dev.pReg, freq, I2C_100K);
    // i2c operations ...

    while (1);
}

```

4.4.3.2 MCU 中断配置

RK HAL 所有资源都是直接定义于 `main.c` 中。以 RK3562 MCU 添加 I2C1 资源为例，说明如何在 RK HAL 中添加一个外设模块。

```

/* ----- I2C1 引脚资源 ----- */
static void HAL_IOMUX_I2C1M0_Config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK0,
                          GPIO_PIN_B3 | GPIO_PIN_B4,
                          PIN_CONFIG_MUX_FUNC1);
}

void main(void)
{
    uint32_t freq;
    struct I2C_HANDLE instance;

    /* HAL BASE Init, MCU Core 使用 NVIC 控制器, HAL_Init 完成 NVIC 初始化 */
    HAL_Init();
    /* BSP Init */
    BSP_Init();

```

```

    HAL_IOMUX_I2C1M0_Config();
    freq = HAL_CRU_ClkGetFreq(g_i2c1dev.clkID);
    HAL_I2C_Init(&instance, g_i2c1dev.pReg, freq, I2C_100K);
    // i2c operations ...

    while (1);
}

```

4.4.3.3 引脚配置

I2C1 引脚配置在 MCU 及 AP 上相同，使用 HAL_PINCTRL_SetIOMUX 函数配置需要的引脚功能。

```

/* ----- I2C1 引脚资源 ----- */
static void HAL_IOMUX_I2C1M0_Config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK0,
                          GPIO_PIN_B3 | GPIO_PIN_B4,
                          PIN_CONFIG_MUX_FUNC1);
}

void main(void)
{
    //.....
    HAL_IOMUX_I2C1M0_Config();
    //.....
}

```

4.4.3.4 时钟配置

I2C1 时钟配置在 MCU 及 AP 上相同，使用 HAL_CRU_ClkGetFreq 和 HAL_I2C_Init 函数获取时钟频率及初始化 I2C 设备。

```

void main(void)
{
    uint32_t freq;
    struct I2C_HANDLE instance;

    //.....
    HAL_CRU_ClkGetFreq(g_i2c1dev.clkID);
    HAL_I2C_Init(&instance, g_i2c1dev.pReg, freq, I2C_100K);
    //.....
}

```

5. 第5章 启动方案

5.1 Rockchip SoC 处理器架构

在瑞芯微多核异构系统中，Rockchip SoC 处理器架构可以抽象为下图所示。

AP Cores （Application Processor），一般为 ARM Cortex-A 处理器核心。

MCU Core （Micro Controller Unit），一般为 ARM Cortex-M 或 RISC-V 处理器核心。

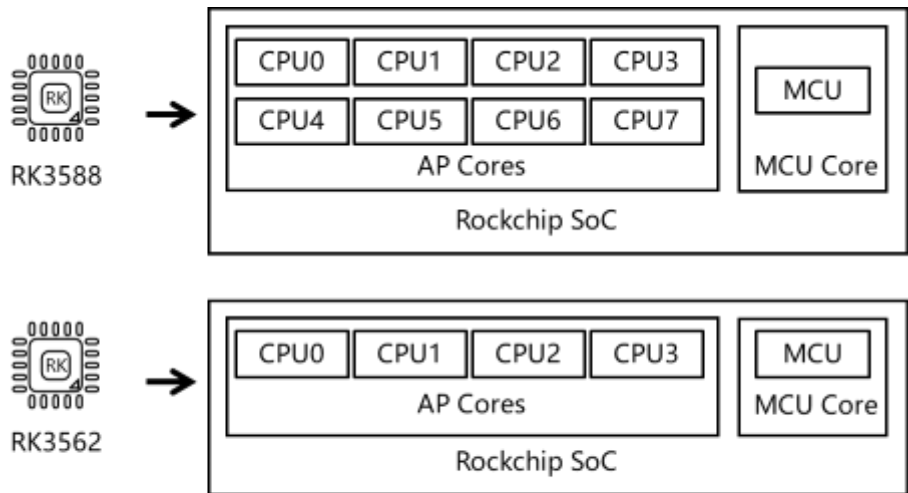


图 5-1-1 Rockchip SoC 处理器架构

5.2 AP + AP 启动方案

以 RK3562 为例，RK3562 是一颗四核 ARM Cortex-A53 处理器，我们将其抽象成 CPU0、CPU1、CPU2、CPU3。在 RK3562 运行瑞芯微多核异构系统时，支持多种组合运行方式。

5.2.1 Linux + RTOS / Bare-metal

5.2.1.1 示例固件：Kernel + RT-Thread / HAL

使用 Kernel + RT-Thread / HAL 的启动方案时默认 RTOS 运行在 CPU3 上。启动时，Bootloader 运行在 CPU0 上，先加载到 U-Boot。U-Boot 中，读取并启动 CPU3 运行 AMP 固件。U-Boot 继续运行在 CPU0 中，继续加载启动 Linux Kernel，Linux Kernel 再将 CPU1 和 CPU2 启动起来。流程框图如下：

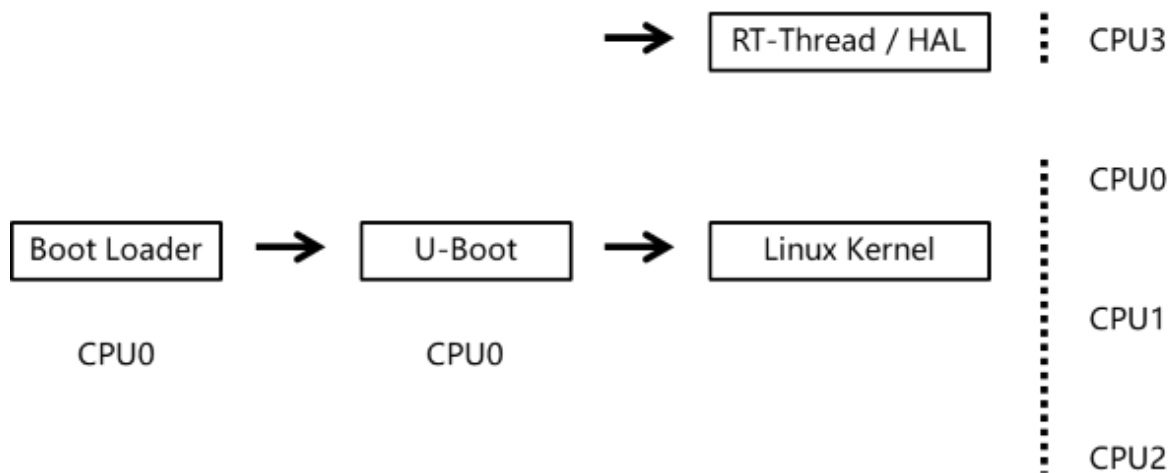


图 5-1-2 kernel + rtt / hal

Kernel + RT-Thread / HAL 的打包文件 amp_linux.its 配置如下：

```

/dts-v1/;
/ {
    description = "FIT source file for rockchip AMP";
    #address-cells = <1>;

    images {
        amp3 {
            description = "bare-mental-core3";
            data = /incbin/"cpu3.bin";
            type = "firmware";
            compression = "none";
            arch = "arm"; # arm or arm64, 默认arm
            cpu = <0x3>; # CPU ID
            thumb = <0>;
            hyp = <0>;
            load = <0x01800000>; # DRAM 起始位置
            # 编译配置，编译解释后，编译脚本自动清除
            compile {
                size = <0x00800000>; # DRAM 大小
                srambase = <0xfe480000>; # SRAM 起始位置
                sramsize = <0x00010000>; # SRAM 大小
                sys = "rtt"; # CPU 运行系统: HAL or RT-Thread

                # RT-Thread 编译配置文件
                rtt_config = "board/rk3562_evbl_lp4x/defconfig"
            };
            udelay = <10000>; # CPU 启动延时，多个 CPU 依次启动时的延时时间

            hash {
                algo = "sha256";
            };
        };
        # 一个 images 下，可以包含多个子节点，SDK 会遍历所有 images 下的节点，根据节点信息，自动编译打包
    };

    # 共享内存信息，编译配置，编译解释后，编译脚本自动清除
    share {

```

```

shm_base      = <0x07800000>; # 多核 CPU 共享 SDRAM 内存起始地址
shm_size      = <0x00400000>; # 多核 CPU 共享 SDRAM 内存分配大小
rpmsg_base    = <0x07C00000>; # RPMSG 共享内存起始地址
rpmsg_size    = <0x00500000>; # RPMSG 共享内存分配大小
# 主系统核心 ID, 当多个 AMP 包含多个系统时, 该参数设定主系统的 CPU ID
# 纯 RTOS AMP 中默认为"0x01"
# 当 AMP 系统中包含 linux 时, linux cpu0 默认配置为主核
primary = <0x0>;

};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;

        loadables = "amp3"; # 加载镜像, 该示例中只有"amp3"
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };

        /* - run linux on cpu0
         * - it is brought up by amp(that run on U-Boot)
         * - it is boot entry depends on U-Boot
         */
        linux {
            description = "linux-os";
            arch = "arm64";
            cpu = <0x000>;
            thumb = <0>;
            hyp = <0>;
            udelay = <0>;

            # AMP 系统如果和 Kernel 加载位置冲突, 需要调整 Kernel
            load = <0x2000000>; # Kernel 加载位置
            load_c = <0x4880000>; # 压缩 Kernel 加载位置
        };
    };
};
};

```

加载位置

5.2.1.2 示例固件：Kernel + 3 * HAL

使用 Kernel + 3 * HAL 的启动方案时默认 Linux 运行在 CPU0 上。启动时，Bootloader 运行在 CPU0 上，先加载到 U-Boot。U-Boot 中，读取 amp.img 固件，按 `loadables = "amp1", "amp2", "amp3"`；顺序，加载 Bare-metal 固件，并启动响应 CPU1，CPU2，CPU3 核心。U-Boot 继续运行在 CPU0 中，继续加载启动 Linux Kernel。流程框图如下：

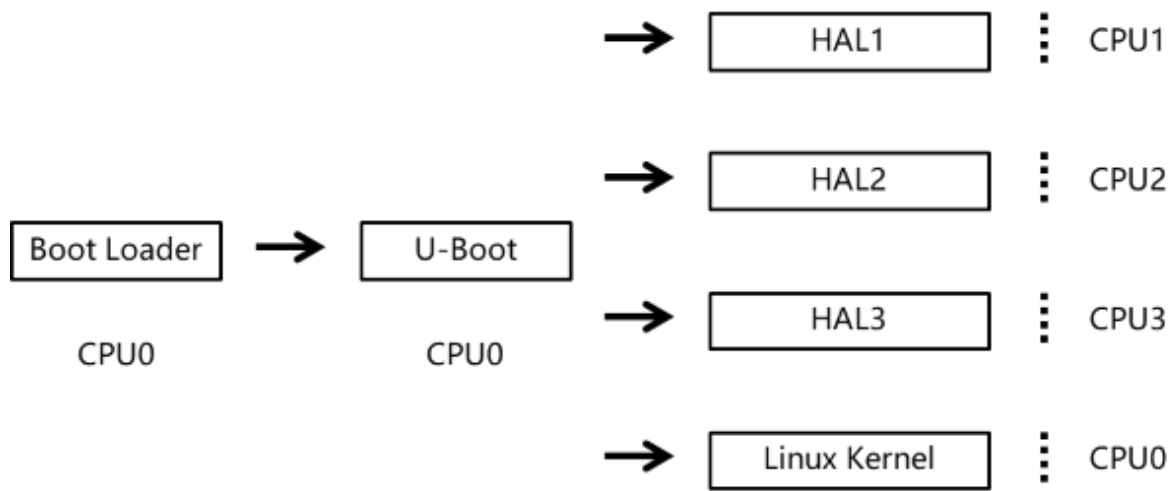


图 5-1-3 kernel + 3 * hal

Kernel + 3 * HAL / RT-Thread 的打包文件 amp_linux.its 配置如下：

```

description = "FIT source file for rockchip AMP";
#address-cells = <1>;

images {
    amp1 {
        # .....
    }

    amp2 {
        # .....
    }

    amp3 {
        description = "bare-mental-core3";
        data        = /incbin/("cpu3.bin");
        type        = "firmware";
        compression = "none";
        arch         = "arm";           # arm or arm64, 默认arm
        cpu          = <0x3>;          # CPU ID
        thumb        = <0>;
        hyp          = <0>;
        load         = <0x01800000>;    # DRAM 起始位置
        # 编译配置，编译解释后，编译脚本自动清除
        compile {
            size      = <0x00800000>;    # DRAM 大小
            srambase  = <0xfe480000>;    # SRAM 起始位置
            sramsize  = <0x00010000>;    # SRAM 大小
            sys       = "hal";          # CPU 运行系统: HAL or RT-
Thread

            # RT-Thread 编译配置文件
            rtt_config = "board/rk3562_evbl_lp4x/defconfig"
        };
        udelay      = <10000>;          # CPU 启动延时，多个 CPU 依次启动时的
延时时间

        hash {
            algo = "sha256";
        };
    };
};
  
```

一个 images 下,可以包含多个子节点,SDK 会遍历所有 images 下的节点,根据节点信息,自动编译打包

```
};
```

共享内存信息,编译配置,编译解释后,编译脚本自动清除

```
share {
```

```
    shm_base          = <0x07800000>; # 多核 CPU 共享 SDRAM 内存起始地址
```

```
    shm_size          = <0x00400000>; # 多核 CPU 共享 SDRAM 内存分配大小
```

```
    rpmsg_base        = <0x07C00000>; # RPMSG 共享内存起始地址
```

```
    rpmsg_size        = <0x00500000>; # RPMSG 共享内存分配大小
```

```
    # 主系统核心 ID,当多个 AMP 包含多个系统时,该参数设定主系统的 CPU ID
```

```
    # 纯 RTOS AMP 中默认为"0x01"
```

```
    # 当AMP系统中包含 linux 时,linux cpu0 默认配置为主核
```

```
    primary = <0x0>;
```

```
};
```

```
configurations {
```

```
    default = "conf";
```

```
    conf {
```

```
        description = "Rockchip AMP images";
```

```
        rollback-index = <0x0>;
```

```
        # 镜像加载列表:当存在多个 amp 镜像时,
```

```
        # 如: loadables = "amp0", "amp1", "amp2", "amp3"...
```

```
        # 当 CPU0 为启动核时,实际加载顺序为: amp1-->amp2-->amp3->...-
```

>amp0

```
        # 依次类推,启动核由于在 boot 阶段被占用,在 AMP 中将是最后一个启动
```

动

```
        # 加载镜像,该示例中有"amp1"、"amp2"、"amp3"
```

```
        loadables = "amp1", "amp2", "amp3";
```

```
        signature {
```

```
            algo = "sha256,rsa2048";
```

```
            padding = "pss";
```

```
            key-name-hint = "dev";
```

```
            sign-images = "loadables";
```

```
        };
```

```
        /* - run linux on cpu0
```

```
        * - it is brought up by amp(that run on U-Boot)
```

```
        * - it is boot entry depends on U-Boot
```

```
        */
```

```
        linux {
```

```
            description = "linux-os";
```

```
            arch        = "arm64";
```

```
            cpu         = <0x000>;
```

```
            thumb       = <0>;
```

```
            hyp         = <0>;
```

```
            udelay      = <0>;
```

加载位置

```
            # AMP 系统如果和 Kernel 加载位置冲突,需要调整 Kernel
```

```
            load        = <0x2000000>; # Kernel 加载位置
```

```
            load_c      = <0x4880000>; # 压缩 Kernel 加载位置
```

```
        };
```

```
    };
```

```
};
```

```
};
```

5.2.2 RTOS + Bare-metal

5.2.2.1 示例固件：HAL + HAL

使用 HAL + HAL 的启动方案时，前级 Bootloader 运行在 CPU0 上，先加载到 U-Boot 中，读取 amp.img 固件，加载 cpu1.bin、cpu2.bin、cpu3.bin 3 个 Bare-metal 固件，并启动响应 CPU1，CPU2，CPU3 核心。U-Boot 仍然运行在 CPU0 中，继续加载剩余的 Bare-metal cpu0.bin 固件。流程框图如下：

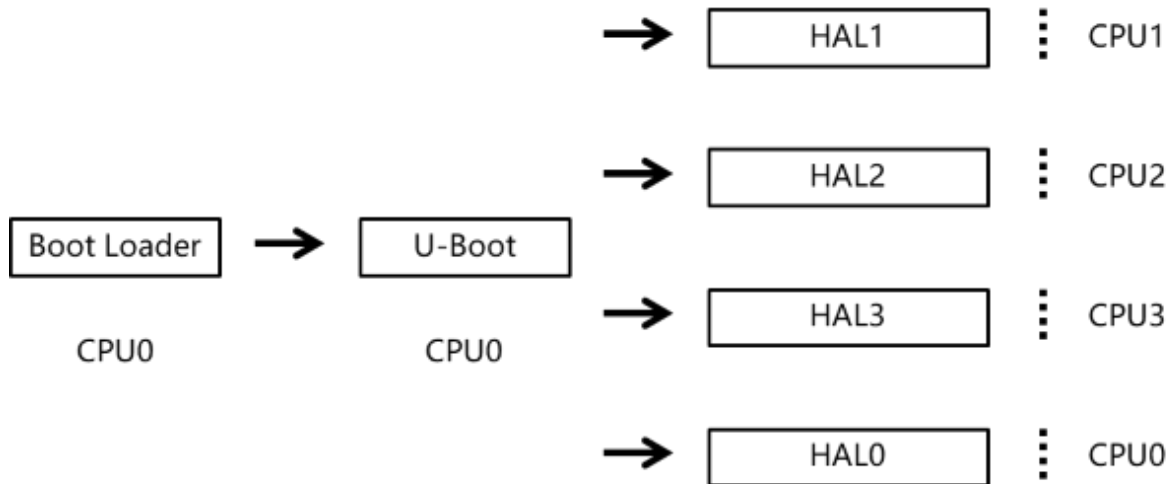


图 5-1-4 hal + hal + hal + hal

HAL + HAL 的打包文件 amp_linux.its 配置如下：

```
description = "FIT source file for rockchip AMP";
#address-cells = <1>;

images {
    amp0 {
        # .....
    }

    amp1 {
        description = "bare-mental-core3";
        data = /incbin/("cpu1.bin");
        type = "firmware";
        compression = "none";
        arch = "arm"; # arm or arm64, 默认arm
        cpu = <0x1>; # CPU ID
        thumb = <0>;
        hyp = <0>;
        load = <0x00800000>; # DRAM 起始位置
        # 编译配置，编译解释后，编译脚本自动清除
        compile {
            size = <0x00800000>; # DRAM 大小
            srambase = <0xfe480000>; # SRAM 起始位置
            sramsize = <0x00010000>; # SRAM 大小
            sys = "hal"; # CPU 运行系统: HAL or RT-Thread

            # RT-Thread 编译配置文件
            rtt_config = "board/rk3562_evbl_lp4x/defconfig"
        };
    }
};
```

```

        udelay          = <10000>; # CPU 启动延时，多个 CPU 依次启动时的
延时时间

        hash {
            algo = "sha256";
        };
    }

    amp2 {
        # .....
    }

    amp3 {
        # .....
    };
    # 一个 images 下，可以包含多个子节点，SDK 会遍历所有 images 下的节点，根据
节点信息，自动编译打包
};

# 共享内存信息，编译配置，编译解释后，编译脚本自动清除
share {
    shm_base          = <0x07800000>; # 多核 CPU 共享 SDRAM 内存起始地址
    shm_size          = <0x00400000>; # 多核 CPU 共享 SDRAM 内存分配大小
    rpmsg_base        = <0x07C00000>; # RPMSG 共享内存起始地址
    rpmsg_size        = <0x00500000>; # RPMSG 共享内存分配大小
    # 主系统核心 ID，当多个 AMP 包含多个系统时，该参数设定主系统的 CPU ID
    # 纯 RTOS AMP 中默认为"0x01"
    # 当 AMP 系统中包含 linux 时，linux cpu0 默认配置为主核
    primary = <0x0>;
};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;

        # 镜像加载列表：当存在多个 amp 镜像时，
        # 如：loadables = "amp0", "amp1", "amp2", "amp3"...
        # 当 CPU0 为启动核时，实际加载顺序为：amp1-->amp2-->amp3->...-
>amp0

        # 依次类推，启动核由于在 boot 阶段被占用，在 AMP 中将是最后一个启
动

        # 加载镜像，该示例中有"amp1"、"amp2"、"amp3"
        loadables = "amp0", "amp1", "amp2", "amp3";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };
    };
};
};

```

5.2.2.2 示例固件： RT-Thread + HAL

使用 RTOS + Bare-metal 的启动方案时，前级 Bootloader 运行在 CPU0 上，先加载到 U-Boot 中，读取 amp.img 固件，加载 cpu1.bin（RTOS固件）和 cpu2.bin、cpu3.bin 2 个 Bare-metal 固件，并启动响应 CPU1，CPU2，CPU3 核心。U-Boot 继续运行在 CPU0 中，继续加载启动 Bare-metal 固件，即 cpu0.bin。流程框图如下：

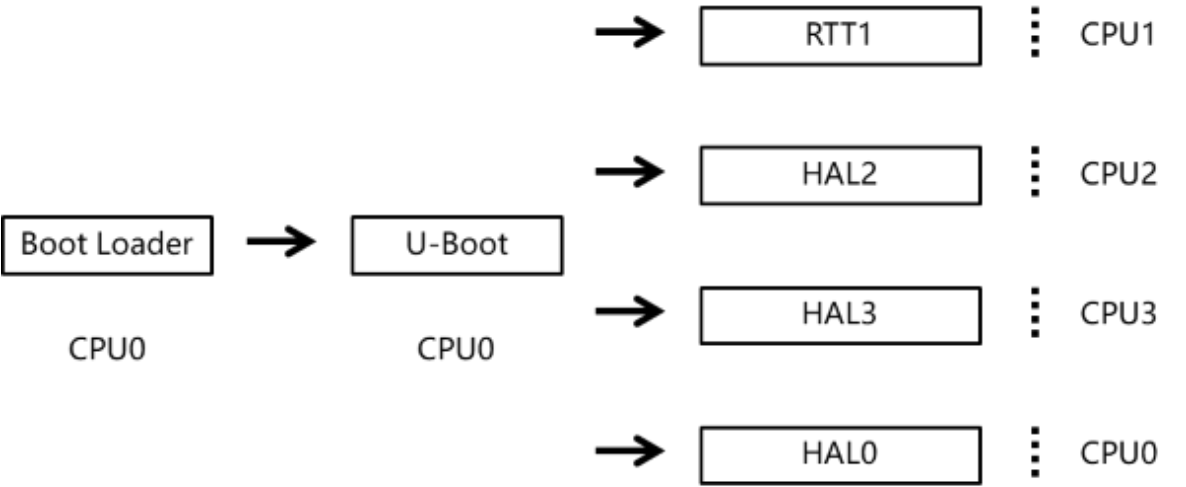


图 5-1-5 rtt + hal + hal + hal

RT-Thread + HAL 的打包文件 amp_linux.its 配置如下：

```
description = "FIT source file for rockchip AMP";
#address-cells = <1>;

images {
    amp0 {
        # .....
    }

    amp1 {
        description = "bare-metal-core3";
        data = /incbin/("cpu1.bin");
        type = "firmware";
        compression = "none";
        arch = "arm"; # arm or arm64, 默认arm
        cpu = <0x1>; # CPU ID
        thumb = <0>;
        hyp = <0>;
        load = <0x00800000>; # DRAM 起始位置
        # 编译配置，编译解释后，编译脚本自动清除
        compile {
            size = <0x00800000>; # DRAM 大小
            srambase = <0xfe480000>; # SRAM 起始位置
            sramsize = <0x00010000>; # SRAM 大小
            sys = "rtt"; # CPU 运行系统: HAL or RT-Thread

            # RT-Thread 编译配置文件
            rtt_config = "board/rk3562_evb1_lp4x/defconfig"
        };
        udelay = <10000>; # CPU 启动延时，多个 CPU 依次启动时的延时时间
    }
}
```

```

        hash {
            algo = "sha256";
        };
    }

    amp2 {
        # .....
    }

    amp3 {
        # .....
    };
    # 一个 images 下，可以包含多个子节点，SDK 会遍历所有 images 下的节点，根据
节点信息，自动编译打包
};

# 共享内存信息，编译配置，编译解释后，编译脚本自动清除
share {
    shm_base          = <0x07800000>; # 多核 CPU 共享 SDRAM 内存起始地址
    shm_size          = <0x00400000>; # 多核 CPU 共享 SDRAM 内存分配大小
    rpmsg_base        = <0x07C00000>; # RPMSG 共享内存起始地址
    rpmsg_size        = <0x00500000>; # RPMSG 共享内存分配大小
    # 主系统核心 ID，当多个 AMP 包含多个系统时，该参数设定主系统的 CPU ID
    # 纯 RTOS AMP 中默认为"0x01"
    # 当 AMP 系统中包含 linux 时，linux cpu0 默认配置为主核
    primary = <0x0>;
};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;

        # 镜像加载列表：当存在多个 amp 镜像时，
        # 如：loadables = "amp0", "amp1", "amp2", "amp3"...
        # 当 CPU0 为启动核时，实际加载顺序为：amp1-->amp2-->amp3->...-
>amp0
动

        # 依次类推，启动核由于在 boot 阶段被占用，在 AMP 中将是最后一个启

        # 加载镜像，该示例中有"amp1"、"amp2"、"amp3"
        loadables = "amp0", "amp1", "amp2", "amp3";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };
    };
};
};

```

5.3 AP + MCU 启动方案

5.3.1 Linux + MCU RTOS / Bare-metal

5.3.1.1 示例固件：Kernel + mcu RT-Thread / HAL

使用 Linux + 1个MCU 的启动方案，启动时，Bootloader 运行在 CPU0 上，先加载到 U-Boot。U-Boot 中，读取并启动 MCU 运行AMP 固件。U-Boot 继续运行在 CPU0 中，继续加载启动 Linux Kernel，Linux Kernel 再将 CPU1、CPU2、CPU3 启动起来。流程框图如下：

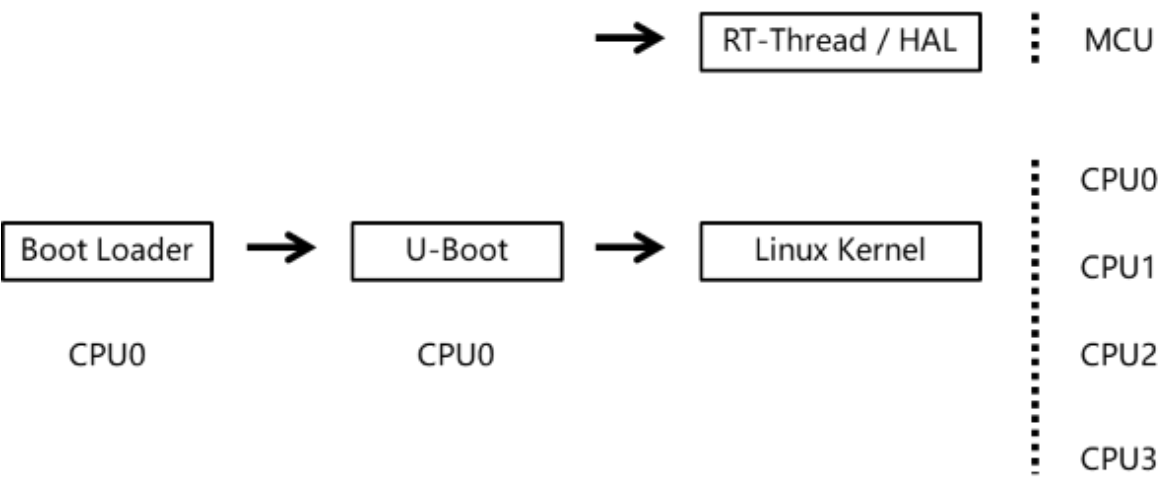


图 5-1-6 kernel + mcu rtt / hal

Kernel + mcu RT-Thread / HAL 的打包文件 amp_linux.its 配置如下：

```
/dts-v1/;
/ {
    description = "Rockchip AMP FIT Image";
    #address-cells = <1>;

    images {
        mcu {
            description = "mcu";
            data = /incbin/("./mcu.bin"); // hal、rtt 系统编译
            type = "standalone"; // must be
            compression = "none";
            arch = "arm"; // "arm64" or
            load = <0x08200000>; // MCU 程序 RAM 启动地址
            udelay = <1000000>; // 启动延时时间
            hash {
                algo = "sha256";
            };
        };
    };

    configurations {
        default = "conf";
        conf {
```


5.5 快速启动方案

5.5.1 SD卡启动

SD 启动卡是通过 RK 的工具制作，实现直接从 SD 卡启动，极大的方便用户更新启动新固件而不用重新烧写固件到设备存储内。具体实现是将固件烧写到 SD 卡中，把 SD 卡当作主存储使用。主控从 SD 卡启动时，固件以及临时文件都存放在 SD 卡上，有没有本地主存储都可以正常工作。

具体启动流程及相关细节可以参考《Rockchip_Developer_Guide_UBoot_Nextdev_CN.pdf》和《Rockchip_Developer_Guide_SD_Boot_CN.pdf》两份文档中SD卡启动部分的描述。

5.6 快速启动方案

SDK 支持系统快速启动方案。通过利用SoC内置的一个高性能的 MCU，在启动阶段协助 AP 端进行一些简单的线程操作，从而实现操作任务的快速进行。

5.6.1 SPL启动方案

SPL 方案指的是通过将 MCU 固件提前到SPL阶段释放和加载，减少了上电到MCU固件加载的时间。

以 RK3562 为例，从上电到 MCU 上第一个控制业务完成时间最快可以达到 200ms。

SPL启动方案具体配置如下：

SPL 阶段启动需要在 rkbin 仓库下，修改 RKTRUST/RK3562TRUST.ini，启用 MCU。默认固件运行在 0x08200000 上。

```
MCU=bin/rk35/rk3562_mcu_v1.00.bin,0x08200000,okay
```

在 uboot 仓库下，使用以下命令编译：

```
./make.sh rk3562 --spl-new
```

MCU 固件打包到 uboot.img 中，由 SPL 负责加载和释放。

5.6.2 双存储启动方案

双存储方案指的是 Nor + eMMC 搭配的方案，将 MCU 固件存放到一个小内存的 Flash 中，AP 固件放在 eMMC 固件中。由于 Flash 上电初始化的时间相比 eMMC 上电初始化的时间少很多，MCU 固件就能够更快速的加载起来。

双存储方案具体配置请参考文档《Rockchip_Developer_Guide_Dual_Storage_CN.pdf》。

6. 第6章 通信方案

6.1 核间中断触发

瑞芯微 AMP 通信方案采用中断 + 共享内存的方式实现，发送端在更新共享内存中数据后，通过触发中断通知接收端进行处理，目前提供三种核间中断触发方式，分别是 Mailbox 中断触发、软件中断触发及 SGI 触发。另外，瑞芯微多核异构系统通常还会提供 Hardware Spinlock 来进行可靠的原子操作。

6.1.1 Mailbox 中断触发

使用 RK Mailbox 模块进行核间通信，在触发 Mailbox 中断的同时，可以传输一个 32 bit 的 Command 寄存器数据和一个 32 bit 的 Data 寄存器数据。

```
#ifdef PRIMARY_CPU
// Master中断处理，会在此处回调函数
static void mbox_master_isr(int vector, void *param)
{
    HAL_MBOX_IrqHandler(vector, pMBox);
    HAL_GIC_EndOfInterrupt(vector);
}
// 在 Mailbox 中断中回调
static void mbox_master_cb(struct MBOX_CMD_DAT *msg, void *args)
{
    uint32_t cpu_id;
    struct MBOX_CMD_DAT rx_msg = *msg;

    cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
    // 处理接收到的 32 bit 的 Command 数据和一个 32 bit 的 Data 数据
    printf("mbox master: recieve cpu-%ld cmd=0x%lx data=0x%lx\n", cpu_id,
rx_msg.CMD, rx_msg.DATA);
}
#else
// Remote 中断处理，会在此处回调函数
static void mbox_remote_isr(int vector, void *param)
{
    HAL_MBOX_IrqHandler(vector, pMBox);
    HAL_GIC_EndOfInterrupt(vector);
}

static void mbox_remote_cb(struct MBOX_CMD_DAT *msg, void *args)
{
    uint32_t cpu_id;
    struct MBOX_CMD_DAT rx_msg = *msg;

    cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
    // 处理接收到的 32 bit 的 Command 数据和一个 32 bit 的 Data 数据
    printf("mbox remote: recieve cpu-%ld cmd=0x%lx data=0x%lx\n", cpu_id,
rx_msg.CMD, rx_msg.DATA);
}
#endif
```

```

#ifdef PRIMARY_CPU
// Master 端通道注册
static struct MBOX_CLIENT mbox_client2_m = { "mbox-cl2m", MBOX0_CH2_B2A_IRQn,
mbox_master_cb, (void *)MBOX_CH_2 };

static void mbox_master_test(void)
{
    struct MBOX_CLIENT *mbox_cl2m;
    struct MBOX_CMD_DAT tx_msg;
    uint32_t cpu_id;
    int ret = 0;

    cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
    mbox_cl2m = &mbox_client2_m;
    tx_msg.CMD = cpu_id & 0xFU;
    tx_msg.DATA = 0x12345678;
    /* master core uses MBOX_A2B and remote core uses MBOX_B2A */
    HAL_MBOX_Init(pMBox, MBOX_A2B);
    ret = HAL_MBOX_RegisterClient(pMBox, MBOX_CH_2, mbox_cl2m);
    if (ret) {
        printf("mbox_cl2m register failed, ret=%d\n", ret);
    }
    HAL_IRQ_HANDLER_SetIRQHandler(MBOX0_CH2_B2A_IRQn, mbox_master_isr, NULL);
    HAL_GIC_Enable(MBOX0_CH2_B2A_IRQn);
    HAL_DelayMs(4000);
    printf("mbox master: send cmd=0x%x data=0x%x\n", tx_msg.CMD, tx_msg.DATA);
    // 发送数据, 32 bit 的 Command 数据和一个 32 bit 的 Data 数据
    HAL_MBOX_SendMsg(pMBox, MBOX_CH_2, &tx_msg);
}
#endif

#ifdef CPU2
// Remote 端通道注册
static struct MBOX_CLIENT mbox_client2_r = { "mbox-cl2r", MBOX0_CH2_A2B_IRQn,
mbox_remote_cb, (void *)MBOX_CH_2 };

static void mbox_remote_test(void)
{
    struct MBOX_CLIENT *mbox_cl2r;
    struct MBOX_CMD_DAT tx_msg;
    uint32_t cpu_id;
    int ret = 0;

    cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
    mbox_cl2r = &mbox_client2_r;
    tx_msg.CMD = cpu_id & 0xFU;
    tx_msg.DATA = 0x98765432;

    /* master core uses MBOX_A2B and remote core uses MBOX_B2A */
    HAL_MBOX_Init(pMBox, MBOX_B2A);
    ret = HAL_MBOX_RegisterClient(pMBox, MBOX_CH_2, mbox_cl2r);
    if (ret) {
        printf("mbox_cl2r register failed, ret=%d\n", ret);
    }
    HAL_IRQ_HANDLER_SetIRQHandler(MBOX0_CH2_A2B_IRQn, mbox_remote_isr, NULL);
    HAL_GIC_Enable(MBOX0_CH2_A2B_IRQn);
    HAL_DelayMs(2000);
}
#endif

```

```

printf("mbox remote: send cmd=0x%x data=0x%x\n", tx_msg.CMD, tx_msg.DATA);
// 发送数据, 32 bit 的 Command 数据和一个 32 bit 的 Data 数据
HAL_MBOX_SendMsg(pMBox, MBOX_CH_2, &tx_msg);
}
#endif

```

6.1.2 软件中断触发

使用 GIC SPI 中断, 即共享外设中断中的 reserved irq, 通过主动 Send Pending 触发。

```

static void soft_isr(int vector, void *param)
{
    printf("soft_isr, vector = %d\n", vector);
    HAL_GIC_EndOfInterrupt(vector);
}

static void softirq_test(void)
{
    HAL_IRQ_HANDLER_SetIRQHandler(RSVD0_IRQn, soft_isr, NULL);
    HAL_GIC_Enable(RSVD0_IRQn);
    // 触发软中断
    HAL_GIC_SetPending(RSVD0_IRQn);
}

```

6.1.3 SGI 触发

使用 GIC SGI, 即软中断触发。由于 Linux SMP 占用了8个 non-secure SGI 中断号, 而另外8个 secure 的 SGI 中断号需要特殊申请。因此, SGI 触发的方式常用于多个从核进行同步。

```

#define IPI_CPU0          0x01
#define IPI_CPU1          0x02
#define IPI_CPU2          0x04
#define IPI_CPU3          0x08
#define IPI_TO_TARGETLIST 0
#define IPI_TO_ALL_EXCEPT_SELF 1

static void ipi_sgi_isr(int vector, void *param)
{
    uint32_t cpu_id;

    cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
    if (cpu_id == 2) {
        HAL_DelayMs(1000);
    } else if (cpu_id == 3) {
        HAL_DelayMs(2000);
    }
    printf("ipi sgi: cpu_id=%ld vector = %d\n", cpu_id, vector);
    HAL_GIC_EndOfInterrupt(vector);
}

static void ipi_sgi_test(void)
{
    uint32_t cpu_id;

```

```

cpu_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
HAL_IRQ_HANDLER_SetIRQHandler(IPI_SGI7, ipi_sgi_isr, NULL);
HAL_GIC_Enable(IPI_SGI7);

if (cpu_id == 1) {
    printf("ipi sgi: cpu_id=%ld test start\n", cpu_id);
    HAL_DelayMs(2000);
    // 触发 CPU0 SGI7中断
    HAL_GIC_SendSGI(IPI_SGI7, 0, IPI_TO_ALL_EXCEPT_SELF);
    HAL_DelayMs(4000);
    HAL_GIC_SendSGI(IPI_SGI7, IPI_CPU3, IPI_TO_TARGETLIST);
    HAL_DelayMs(4000);
    HAL_GIC_SendSGI(IPI_SGI7, IPI_CPU3 | IPI_CPU2, IPI_TO_TARGETLIST);
    HAL_DelayMs(4000);
    HAL_GIC_SendSGI(IPI_SGI7, IPI_CPU3 | IPI_CPU2 | IPI_CPU0,
IPI_TO_TARGETLIST);
}
}

```

6.2 底层接口方案

瑞芯微多核异构系统开放核间中断 + Shared Memory 底层驱动接口给客户，对于已经在使用多核异构系统的客户，可以直接替换相应底层驱动接口，完成平台移植工作。

RK AMP 目前核间中断触发方式支持 Mailbox、软件中断，共享内存 Linux 仅支持 uncache，其余默认支持 cacheable

Linux 下 Mailbox 中断的方式参考如下路径的代码：`drivers/rpmsg/rockchip_rpmsg_mbox.c`

Linux 下软件中断的方式参考如下路径的代码：`drivers/rpmsg/rockchip_rpmsg_softirq.c`

Bare-metal 下 RPMsg-Lite 参考如下路径的代码：

`hal/middleware/rpmsg-lite/lib/rpmsg_lite/porting/platform/RKXX/rpmsg_platform.c`

RTOS 下 RPMsg-Lite 参考如下路径的代码：

`rtos/rockchip/common/drivers/rpmsg-lite/lib/rpmsg_lite/porting/platform/RKXX/rpmsg_platform.c`

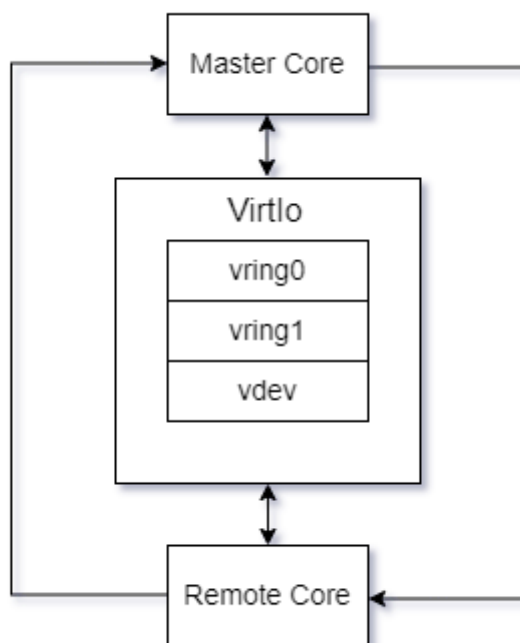
6.3 RPMsg 协议方案

6.3.1 标准框架

瑞芯微为多核异构系统提供了 RPMsg 协议标准框架方案，Linux Kernel 适配 RPMsg，RTOS 和 Bare-metal 适配 RPMsg-Lite。它定义了 AMP 系统中核与核之间进行通信时所使用的标准二进制接口。

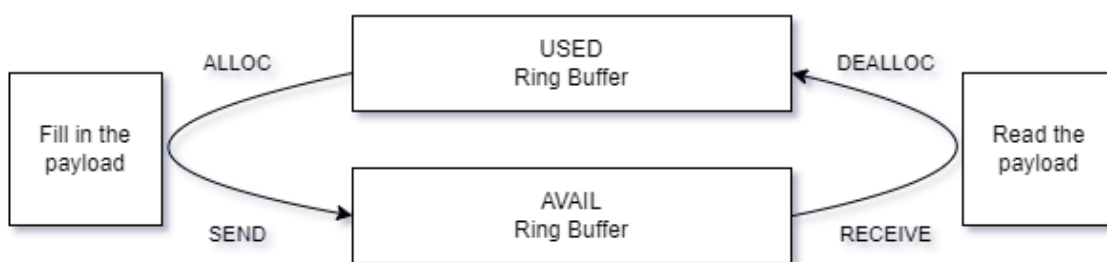
RPMsg 是在 VirtIo 上实现的一个消息传递机制，VirtIo 是一种用来实现虚拟化 IO 的通用架构，类似的虚拟网卡，虚拟磁盘等都是用这种技术。VirtIo 中基于 VirtIo-Ring，通过共享内存实现数据的发送/接收，vring 是单向的，一个 vring 专用于发送数据到 Remote Core，另一个 vring 用于从 Remote Core 接收数据。

因此从整体框架上看，RPMsg 是由 Master Core 和 Remote Core 的核间中断，以及 vring0、vring1、vdev buffer 三段 Shared Memory 构成。



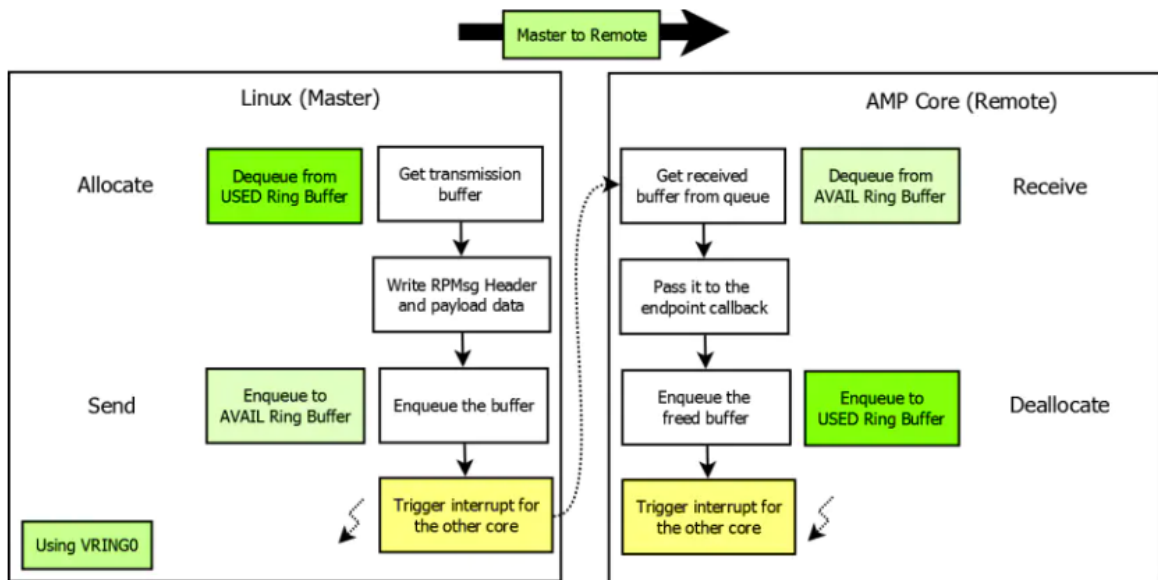
6.3.2 通信流程

在 RPMsg 中，主-从核心通过中断和共享内存的方式进行通信，内存的管理由主核负责，在每个通信方向上都有 USED 和 AVAIL 两个缓冲区，这两个缓冲区可以按照 RPMsg 的消息格式分成一块一块，由这些内存块可以链接成一个环。



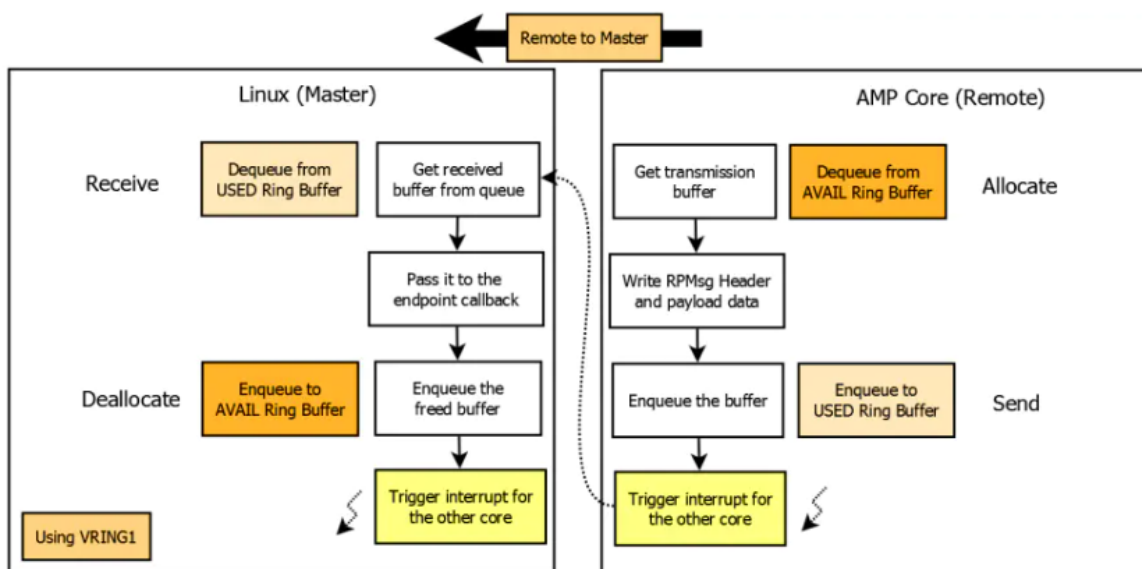
因此当主核（Master Core）和从核（Remote Core）进行通信时：

1. Master Core 发送时，从 vring0(USED) 中取得一块 buffer，再将消息按照 RPMsg 协议填充
2. 将处理好的内存 buffer 链接到 vring1(AVAIL)
3. 触发中断通知 Remote Core 有数据处理待处理



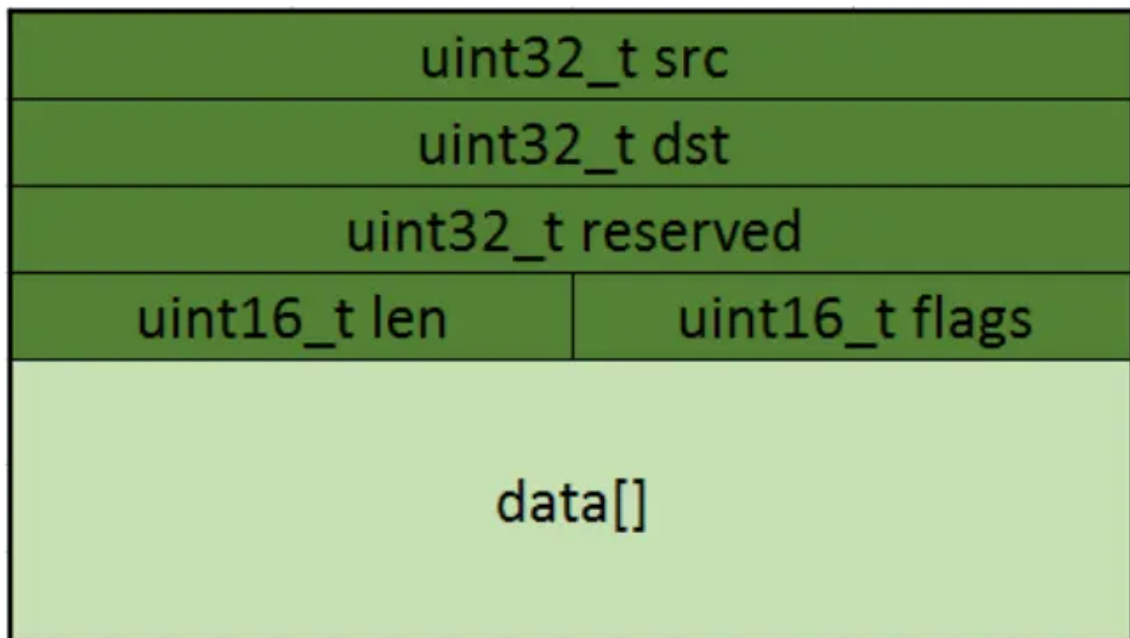
类似的，当从核需要和主核进行通信时：

1. 从核根据队列从 vring1(AVAIL) 中取得一块 buffer，再将消息按照 RPMsg 协议填充
2. 将处理好的内存 buffer 链接到 vring0(USED)
3. 触发中断通知 Master Core 有数据处理待处理



完成消息传递后，释放使用的 buffer，并等待下一笔数据发送。从核发送时，与主核发送流程相反。通信过程中的共享数据放在 vdev buffer 中。

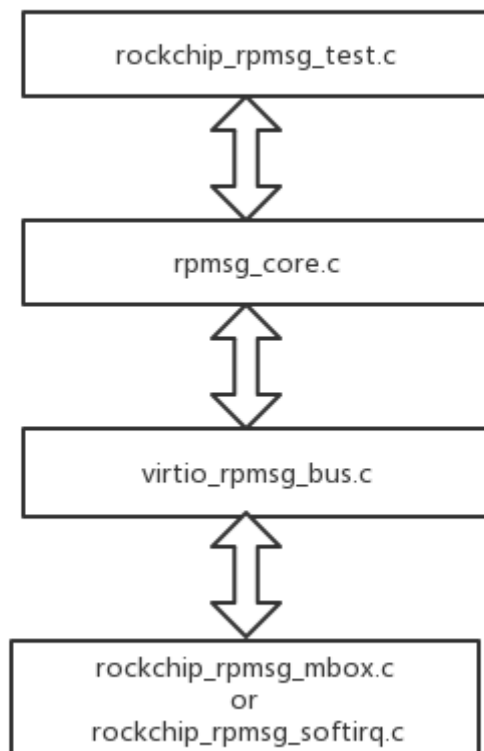
RPMsg 每次发送的最大数据长度取决于 payload 长度，这个长度在 SDK 中默认为 512 Bytes，由于 RPMsg 还带有 16 Bytes 的数据头，因此一次性传输的最大数据量为 496 Bytes，详细可以参考 `drivers/rpmsg/virtio_rpmsg_bus.c`



6.3.3 RPMsg 适配

6.3.3.1 Linux Kernel 适配 RPMsg

Linux Kernel RPMsg 主要代码结构:



kernel/drivers/rpmsg/rockchip_rpmsg_mbox.c 是注册在 Platform Bus 上的 driver，同时向 VirtIO Bus 注册 device。它是基于 mailbox 核间中断加 Shared Memory 底层驱动接口实现的物理层（Physical Layer）。

kernel/drivers/rpmsg/rockchip_rpmsg_softirq.c 也是注册在 Platform Bus 上的 driver，同时向 VirtIO Bus 注册 device。它是基于 softirq 核间中断加 Shared Memory 底层驱动接口实现的物理层（Physical Layer）。

kernel/drivers/rpmsg/virtio_rpmsg_bus.c 是注册在 VirtIO Bus 上的 driver，同时向 RPMsg Bus 注册 device。VirtIO 和 Virtqueue 是通用 RPMsg 协议选择的MAC层（MAC Layer）。

kernel/drivers/rpmsg/rpmsg_core.c 则是创建 RPMsg Bus，并提供传输层（Transport Layer）接口。

kernel/drivers/rpmsg/rockchip_rpmsg_test.c 提供一个简单的核间通信通道创建和数据收发的示例。

RPMsg TTY支持

Linux kernel 同时支持将 RPMsg 挂载成 TTY 设备，其原理和软件结构和上述的 Linux 一致，Linux 端 RPMsg 一般作为 Master，在与 Remote 进行连接时需要创建终端(endpoint)，一个通道(channel)允许创建多个终端，通过服务器名称(service name)来创建的不同终端，也就是说在 Linux(Master) 的本地服务器名称(local service name)和远程 Remote 的发送的服务器名称相匹配时，在通道(channel)的两端就创建了两个可以相互通信的终端。

如kernel/drivers/rpmsg/rockchip_rpmsg_test.c:

```
static struct rpmsg_device_id rockchip_rpmsg_test_id_table[] = {
    { .name = "rpmsg-ap3-ch0" },
    { .name = "rpmsg-mcu0-test" },
    { .name = "rpmsg-perf-bw-test" },
    { .name = "rpmsg-perf-pingpong-bw-test" },
    { .name = "rpmsg-latency-test" },
    { /* sentinel */ },
};
```

在 kernel/drivers/rpmsg/rockchip_rpmsg_test.c 声明了如上几个服务器名称(service name)，而 Remote 端在 Link 完成后，可以通过发送对应的服务器名称来创建链接，在名称匹配时调用 probe 函数。

RPMsg TTY 原理也是如此，在开启配置后，Remote 端发送匹配的服务器名称，Linux RPMsg 在收到匹配的名称后，调用 probe 函数，同时将 RPMsg 注册成 TTY 设备，注册成功后可以发现 /dev/ttyRPMMSG 节点。

TTY 创建示例如下：

Remote 端在创建终端后，使用 rpmsg_ns_announce 发送"rpmsg-tty"服务器名称：

```
ept = rpmsg_lite_create_ept(instance, RPMMSG_HAL_REMOTE_TEST3_EPT, remote_ept_cb,
info);
rpmsg_ns_announce(instance, ept, "rpmsg-tty", RL_NS_CREATE);
```

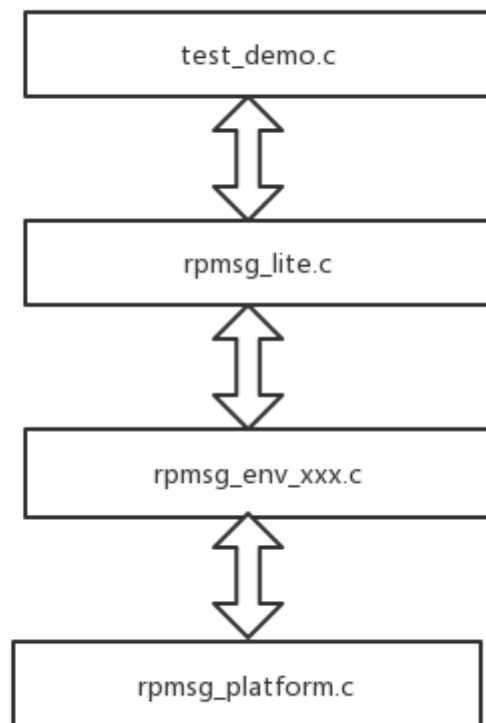
kernel/drivers/tty/rpmsg_tty.c

```
static struct rpmsg_device_id rpmsg_driver_tty_id_table[] = {
    { .name = "rpmsg-tty" },
    { },
};
```

6.3.3.2 RPMsg-Lite 适配

RPMsg-Lite 是第三方开源方案，结构与 Linux RPMsg 类似。

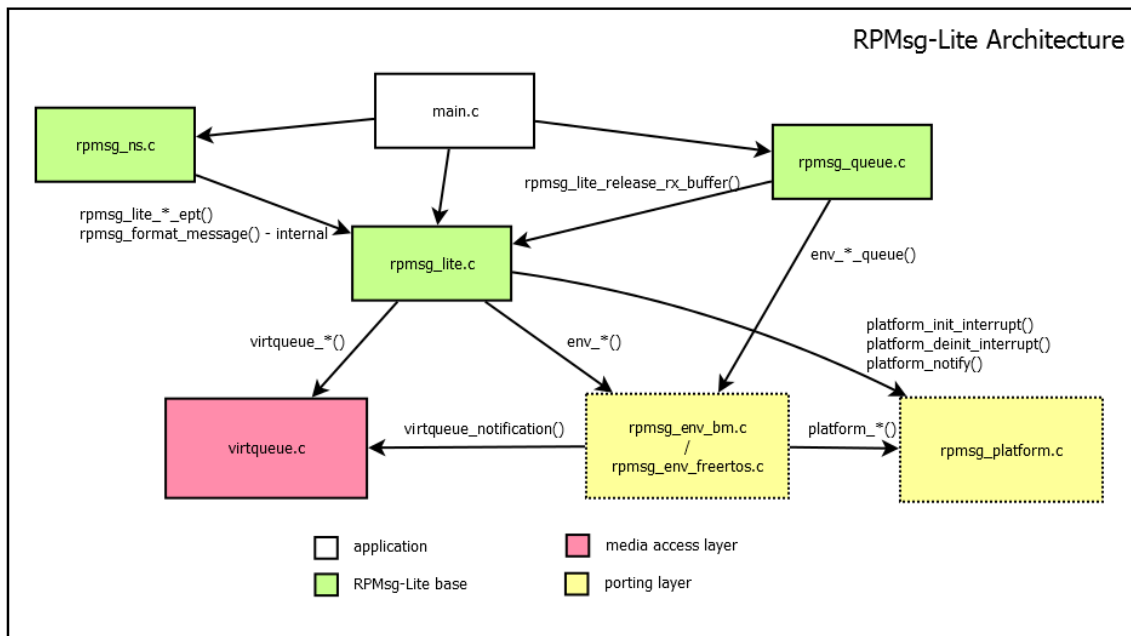
RPMsg-Lite 主要代码结构：



RPMsg-Lite 的实现可以分为三个组件，其中两个是可选的。核心组件位于 `rpmsg_lite.c`。两个可选组件用于实现接收阻塞 API（在 `rpmsg_queue.c` 中）和 Name Service endpoint，用来创建和声明终端节点（在 `rpmsg_ns.c` 中）。

实际的内存访问是在 `virtqueue.c` 中实现的，主要定义用来管理共享内存使用的数据结构，例如 `vring` 或 `virtqueue`。

移植部分为两个层：`environment` 和 `platform`。`environment` 将针对每个环境单独实现，如裸机环境 Bare-metal 在 `rpmsg_env_bm.c` 中实现，RT-Thread 环境在 `rpmsg_env_rt_thread.c` 中实现，当然，开发者也可以参考这部分代码，实现对指定 RTOS 的支持。`platform` 在 `rpmsg_platform.c` 中实现，主要实现中断配置和触发，如上文介绍，可以在这一阶段使用不同的中断（MailBox或软中断）。如果存在 `cache` 或者共享内存地址的偏移，也需要在 `platform` 中进行具体的操作，RPMsg-Lite 可以参考：



6.3.3.3 MCU RPMsg-Lite 适配

MCU 同样使用 RPMsg-Lite，需要注意的是由于 MCU 存在内存的映射，如果想要改变共享内存的地址，需要重新配置 uboot 和 rpmsg_platform.c。目前 RPMsg 的共享内存都处于 unCache 下，所以不需要刷新 Cache 来保证内存数据一致，对于有 MCU 的平台只需要注意共享内存地址的映射，一般 MCU 的地址映射会在 Uboot 中进行初始化。

在 MCU 端 RAM 的起始地址由 ITS 配置传入，和实际的物理地址存在偏移，所以 MCU 看到的共享内存地址和真实的物理地址也存在偏移，这里建议将共享内存的地址放在 MCU 之后，方便后续操作。

以 RK3562 为例，寄存器说明可以参考 TRM 文档，目前 SDK 共享地址配置在 0x07C00000，大小为 0x500000，MCU RAM 起始地址由 ITS 传给 Uboot，配置为 0x7b00000，所以在 u-boot/arch/arm/mach-rockchip/rk3562/rk3562.c 中：

```

int fit_standalone_release(char *id, uintptr_t entry_point)
{
    /* bus m0 configuration: */
    /* open hclk_dcache / hclk_icache / clk_bus m0 rtc / fclk_bus_m0_core */
    writel(0x03180000, TOP_CRU_BASE + TOP_CRU_GATE_CON23);

    /* open bus m0 sclk / bus m0 hclk / bus m0 dclk */
    writel(0x00070000, TOP_CRU_BASE + TOP_CRU_CM0_GATEMASK);

    /*
     * mcu_cache_peripheral_addr
     * The uncache area ranges from 0x7c00000 to 0xffb40000
     * and contains rpmsg shared memory
     */
    /*这里将0x07c00000到0xffb40000都设置为了uncache，确保共享内存处于uncache*/
    writel(0x07c00000, SYS_GRF_BASE + SYS_GRF_SOC_CON5);
    writel(0xffb40000, SYS_GRF_BASE + SYS_GRF_SOC_CON6);
    /*配置MCU RAM的起始地址*/
    sip_smc_mcu_config(ROCKCHIP_SIP_CONFIG_BUSMCU_0_ID,
        ROCKCHIP_SIP_CONFIG_MCU_CODE_START_ADDR,
        0xffff0000 | (entry_point >> 16));
}

```

```

/* release dcache / icache / bus m0 jtag / bus m0 */
writel(0x03280000, TOP_CRU_BASE + TOP_CRU_SOFTRST_CON23);

/* release pmu m0 jtag / pmu m0 */
/* writel(0x00050000, PMU1_CRU_BASE + PMU1_CRU_SOFTRST_CON02); */

return 0;
}

```

MCU 的 amp.its 中:

```

/*
 * Copyright (C) 2023 Rockchip Electronics Co., Ltd
 *
 * SPDX-License-Identifier: GPL-2.0
 */

/dts-v1/;
/ {
    description = "Rockchip AMP FIT Image";
    #address-cells = <1>;

    images {
        mcu {
            description = "mcu";
            data = /incbin/("./mcu.bin");
            type = "standalone"; // must be "standalone"
            compression = "none";
            arch = "arm"; // "arm64" or "arm", the same as U-Boot
state
            load = <0x07b00000>; //MCU RAM 起始地址
            udelay = <1000000>;
            hash {
                algo = "sha256";
            };
        };
    };

    configurations {
        default = "conf";
        conf {
            description = "Rockchip AMP images";
            rollback-index = <0x0>;
            loadables = "mcu";

            signature {
                algo = "sha256,rsa2048";
                padding = "pss";
                key-name-hint = "dev";
                sign-images = "loadables";
            };
        };
    };
};
};
};

```

gcc_bus_m0.ld 配置中可以看到，共享内存的地址是一个偏移地址：

```

/* SPDX-License-Identifier: BSD-3-Clause */
/*
 * Copyright (c) 2023 Rockchip Electronics Co., Ltd.
 */

MEMORY
{
    # RAM 起始地址，真实物理地址 0x07b00000
    RAM(rxw) : ORIGIN = 0x00000000, LENGTH = 0x100000
    # RPMsg 共享内存地址，真实物理地址 0x07c00000
    LINUX_RPMMSG (rxw) : ORIGIN = 0x00100000, LENGTH = 0x00500000
}

```

同时内存的管理由 Master 负责，MCU 作为 Remote 端，从 vring 中取到的 buffer 地址为真实物理地址，所以在发送/接收时都需要对申请到地址进行偏移处理，参考 `rpmsg_platform.c`

```

#ifdef HAL_MCU_CORE
/* MCU offset address */
#ifdef HAL_CACHE_DECODED_ADDR_BASE
#define RL_PHY_MCU_OFFSET HAL_CACHE_DECODED_ADDR_BASE
#else
#define RL_PHY_MCU_OFFSET (0U)
#endif
#endif

/**
 * platform_patova
 *
 * Dummy implementation
 *
 */
void *platform_patova(uint32_t addr)
{
#ifdef HAL_MCU_CORE
    addr -= RL_PHY_MCU_OFFSET;
#endif
    return ((void *) (char *)addr);
}

```

6.4 RPMsg 编译配置

6.4.1 Kernel + RT-Thread

Kernel 配置:

```

## 第6章 开启MailBox支持
CONFIG_MAILBOX=y
CONFIG_ROCKCHIP_MBOX=y
## 第6章 MailBox 中断触发
CONFIG_RPMMSG_ROCKCHIP_MBOX=y
CONFIG_RPMMSG_VIRTIO=y

```

开启TTY的支持:

```
CONFIG_RPMMSG_TTY=y
```

RT-Thread 配置:

运行 `scons --menuconfig`

```
## 第6章 开启RPMMsg-Lite 支持
CONFIG_RT_USING_RPMMSG_LITE=y
## 第6章 开启Linux+RTT RPMMsg
CONFIG_RT_USING_LINUX_RPMMSG=y
```

6.4.2 Kernel + HAL

Kernel 配置:

```
## 第6章 开启MailBox支持
CONFIG_MAILBOX=y
CONFIG_ROCKCHIP_MBOX=y
## 第6章 MailBox 中断触发
CONFIG_RPMMSG_ROCKCHIP_MBOX=y
CONFIG_RPMMSG_VIRTIO=y
```

HAL 配置:

HAL 默认开启 RPMMSG 的支持,可以参考**RPMMsg** 测试示例章节,开启使用 HAL 的测试 demo。

```
// 在test_demo.c 中开启
#define RPMMSG_LINUX_TEST
```

6.4.3 RT-Thread + HAL

RT-Thread 配置:

运行 `scons --menuconfig`

```
## 第6章 开启RPMMsg-Lite 支持
CONFIG_RT_USING_RPMMSG_LITE=y
```

6.5 RPMMsg 测试示例

6.5.1 Kernel + RT-Thread

Kernel RPMMSG 核间通信框架,底层适配使用 VirtIO 方案。主要代码路径如下:

```
kernel/drivers/rpmsg/rpmsg_core.c
kernel/drivers/rpmsg/virtio_rpmsg_bus.c
kernel/drivers/rpmsg/rockchip_rpmsg_softirq.c
kernel/include/linux/rpmsg/rockchip_rpmsg.h
```

6.5.1.1 共享内存

以 SDK 提供的 demo 为例，划分 5M 共享内存给 RPMSG，其中 4M 为 VRING BUFFER，1M 为 VDEV BUFFER。目前共享内存仅支持 unCache。

Kernel Path: SDK/kernel/arch/arm64/boot/dts/rockchip/rkxxxx-amp.dtsi

```
reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    /* remote amp core address */
    amp_reserved: amp@2e00000 {
        reg = <0x0 0x2e00000 0x0 0x1200000>;
        no-map;
    };

    rpmsg_reserved: rpmsg@7c00000 {
        reg = <0x0 0x07c00000 0x0 0x400000>;
        no-map;
    };

    rpmsg_dma_reserved: rpmsg-dma@8000000 {
        compatible = "shared-dma-pool";
        reg = <0x0 0x08000000 0x0 0x100000>;
        no-map;
    };
};
```

RTT Path: SDK/rtos/bsp/rockchip/rk3308-32/board/common/board_base.c

1.MMU 映射为 unCache

```
{LINUX_SHMEM_BASE, LINUX_SHMEM_BASE + LINUX_SHMEM_SIZE - 1, LINUX_SHMEM_BASE,
UNCACHED_MEM},
```

6.5.1.2 测试 Demo

6.5.1.2.1 Kernel Demo

在 Kernel 工程中修改配置文件 kernel/arch/arm64/configs/xxxx_defconfig

```
CONFIG_RPMSG_ROCKCHIP_TEST
```

或者 menuconfig 配置菜单配置:

```
## 第6章 打开配置界面
make ARCH=arm64 rkxxxx_linux_defconfig
make ARCH=arm64 menuconfig

# 打开以下宏开关
CONFIG_RPMMSG_ROCKCHIP_TEST

## 第6章 保存配置
make ARCH=arm64 savedefconfig
cp defconfig arch/arm64/configs/rkxxxx_linux_defconfig
```

Kernel Demo Path: kernel/drivers/rpmsg/rockchip_rpmsg_test.c

```
1.demo主要流程
static struct rpmsg_driver rockchip_rpmsg_test = {
    .drv.name     = KBUILD_MODNAME,
    .drv.owner    = THIS_MODULE,
    .id_table     = rockchip_rpmsg_test_id_table,
    .probe        = rockchip_rpmsg_test_probe,
    .callback     = rockchip_rpmsg_test_cb,
    .remove       = rockchip_rpmsg_test_remove,
};

2.rockchip_rpmsg_test_id_table
/* 等待从核announce完声明一个新的ept name,如果和下面链表中的name对应则进入probe函数中 */
static struct rpmsg_device_id rockchip_rpmsg_test_id_table[] = {
    { .name = "rpmsg-ap3-ch0" },
    { .name = "rpmsg-mcu0-test" },
    { /* sentinel */ },
};

3.rockchip_rpmsg_test_probe
static int rockchip_rpmsg_test_probe(struct rpmsg_device *rp)
{
    int ret, size;
    uint32_t master_ept_id, remote_ept_id;
    struct instance_data *idata;

    master_ept_id = rp->src;
    remote_ept_id = rp->dst;
    dev_info(&rp->dev, "new channel: 0x%x -> 0x%x!\n", master_ept_id,
            remote_ept_id);

    /*probe发一笔数据过去给remote,让remote知道master ept id*/
    ret = rpmsg_send(rp->ept, LINUX_TEST_MSG_1, strlen(LINUX_TEST_MSG_1));
    if (ret) {
        dev_err(&rp->dev, "rpmsg_send failed: %d\n", ret);
        return ret;
    }

    /*运行测试*/
    ret = rpmsg_sendto(rp->ept, LINUX_TEST_MSG_2, strlen(LINUX_TEST_MSG_2),
            remote_ept_id);
    if (ret) {
        dev_err(&rp->dev, "rpmsg_send failed: %d\n", ret);
        return ret;
    }
}
```

```

    }

    return 0;
}

4.rockchip_rpmsg_test_cb
static int rockchip_rpmsg_test_cb(struct rpmsg_device *rp, void *payload,
                                   int payload_len, void *priv, u32 src)
{
    int ret, size;
    uint32_t remote_ept_id;
    struct instance_data *idata = dev_get_drvdata(&rp->dev);

    /* master发完一笔数据给remote后, remote也会发一笔数据过来 */
    remote_ept_id = src;
    dev_info(&rp->dev, "rx msg %s rx_count %d(remote_ept_id: 0x%x)\n",
             (char *)payload, ++idata->rx_count, remote_ept_id);

    /* 测试来回收发10000后退出 */
    if (idata->rx_count >= MSG_LIMIT) {
        dev_info(&rp->dev, "Rockchip rpmsg test exit!\n");
        return 0;
    }

    /* 收到数据后再次发送一个数据给对端 */
    ret = rpmsg_sendto(rp->ept, LINUX_TEST_MSG_2, strlen(LINUX_TEST_MSG_2),
                      remote_ept_id);

    if (ret)
        dev_err(&rp->dev, "rpmsg_send failed: %d\n", ret);
    return ret;
}

```

具体接口函数说明如下：

函数	说明
rpmsg_send()	向远程处理器发送消息
rpmsg_sendto()	向远程处理器发送消息，指定 remote ept id

注意：在主从核发送消息的函数中均设置了dst和src参数表示master ept id和remote ept id，对于master端来说dst和src代表master ept id和remote ept id，对于remote端来说dst和src代表remote ept id和master ept id。

6.5.1.2.2 RTT Demo

配置菜单配置：scons --menuconfig

```

CONFIG_RT_USING_RPMMSG_LITE=y
CONFIG_RT_USING_LINUX_RPMMSG=y
CONFIG_RT_USING_COMMON_TEST_LINUX_RPMMSG_LITE=y

```

RTT Demo Path: rtos/bsp/rockchip/common/tests/rpmsg_test.c

```

static void rpmsg_linux_test(void)
{

```

```

int j;
uint32_t master_id, remote_id;
struct rpmsg_info_t *info;
struct rpmsg_block_t *block;
rpmsg_queue_handle remote_queue;
char *rx_msg = (char *)rt_malloc(RL_BUFFER_PAYLOAD_SIZE);
uint32_t master_ept_id;
uint32_t ept_flags;
void *ns_cb_data;

rpmsg_share_mem_check();
master_id = MASTER_ID;
remote_id = HAL_CPU_TOPOLOGY_GetCurrentCpuId();
rt_kprintf("rpmsg remote: remote core cpu_id-%ld\n", remote_id);
rt_kprintf("rpmsg remote: shmem_base-0x%lx shmem_end-%lx\n",
RPMMSG_LINUX_MEM_BASE, RPMMSG_LINUX_MEM_END);

info = malloc(sizeof(struct rpmsg_info_t));
if (info == NULL) {
    rt_kprintf("info malloc error!\n");
    while (1) {
        ;
    }
}
info->private = malloc(sizeof(struct rpmsg_block_t));
if (info->private == NULL) {
    rt_kprintf("info malloc error!\n");
    while (1) {
        ;
    }
}

/*初始化rpmsg ept*/
info->instance = rpmsg_lite_remote_init((void *)RPMMSG_LINUX_MEM_BASE,
RL_PLATFORM_SET_LINK_ID(master_id, remote_id), RL_NO_FLAGS);
rpmsg_lite_wait_for_link_up(info->instance);
rt_kprintf("rpmsg remote: link up! link_id-0x%lx\n",
info->instance->link_id);
rpmsg_ns_bind(info->instance, rpmsg_ns_cb, &ns_cb_data);
remote_queue = rpmsg_queue_create(info->instance);
info->ept = rpmsg_lite_create_ept(info->instance,
RPMMSG_RTT_REMOTE_TEST3_EPT_ID, rpmsg_queue_rx_cb, remote_queue);

/*从核announce完声明一个新的ept name与master端对应*/
ept_flags = RL_NS_CREATE;
rpmsg_ns_announce(info->instance, info->ept,
RPMMSG_RTT_REMOTE_TEST_EPT3_NAME, ept_flags);

/***** rpmsg test run *****/
for (j = 0; j < 100; j++)
{
    rpmsg_queue_rcv(info->instance, remote_queue,
        (uint32_t *)&master_ept_id, rx_msg,
        RL_BUFFER_PAYLOAD_SIZE, RL_NULL, RL_BLOCK);
    // rpmsg_queue_rcv_nocopy(remote_rpmsg, remote_queue, (uint32_t *)&src,
        (char **)&rx_msg, RL_NULL, RL_BLOCK);
    rt_kprintf("rpmsg remote: master_ept_id-0x%lx rx_msg: %s\n",
        master_ept_id, rx_msg);
}

```

```

        rpmsg_lite_send(info->instance, info->ept, master_ept_id,
        RPMSG_RTT_TEST_MSG, strlen(RPMSG_RTT_TEST_MSG), RL_BLOCK);
    }
}

```

具体接口函数说明如下：

函数	说明
rpmsg_lite_remote_init()	RPMMsg-lite remote 端初始化
rpmsg_queue_create()	RPMMsg-lite 创建队列
rpmsg_lite_create_ept()	创建端点
rpmsg_queue_recv()	接收到的数据自动复制到缓存区
rpmsg_ns_bind()	绑定 name service ept（0x35这个ept id是专门给name service用于传新通道的名字）
rpmsg_ns_announce()	声明 remote new ept name
rpmsg_lite_send()	发送消息

6.5.1.2.3 测试成功log

Linux master core RPMSG 成功挂载能看到如下打印：

```

[ 1.105178] rockchip-rpmsg 7c00000.rpmsg: rockchip rpmsg platform probe.
[ 1.105228] rockchip-rpmsg 7c00000.rpmsg: assigned reserved memory node
rpmsg_dma@8000000
[ 1.105239] rockchip-rpmsg 7c00000.rpmsg: rpdev vdev0: vring0 0x7c00000,
vring1 0x7c08000
[ 1.105720] virtio_rpmsg_bus virtio0: rpmsg host is online

```

remote core 发起 name service announce 后，Linux master core 能看到如下打印：

```

[ 1.105808] virtio_rpmsg_bus virtio0: creating channel rpmsg-ap3-ch0 addr 0xc3
[ 1.105980] rockchip_rpmsg_test virtio0.rpmsg-ap3-ch0.-1.195: rpmsg master:
new channel: 0x400 -> 0xc3!

```

其中，rpmsg-ap3-ch0 为 ept name，0x400 为 Master ept id，0xc3 为 Remote ept id。

RT-Thread 测试结果，开机 log 信息如下：

```

[(3)0.101.712] rpmsg remote: remote core cpu_id-3
[(3)0.101.890] rpmsg remote: shmem_base-0x7c00000 shmem_end-8100000
[(3)0.506.840] rpmsg remote: link up! link_id-0x3

```

RPMSG FLAG 定义如下

```

/* rpmsg flag bit definition
 * bit 0: Set 1 to indicate remote processor is ready
 * bit 1: Set 1 to use reserved memory region as shared DMA pool
 * bit 2: Set 1 to use cached share memory as vring buffer
 */
#define RPMSG_REMOTE_IS_READY          BIT(0)
#define RPMSG_SHARED_DMA_POOL         BIT(1)
#define RPMSG_CACHED_VRING            BIT(2)

```

6.5.2 RT-Thread + HAL

RTOS 的 RPMsg-lite 多核通信是建立在核间中断和共享内存的基础上。通过标准化的框架，实现多核之间的通信。默认配置 CPU 1 为 Master，其他 CPU 为 Remote。

6.5.2.1 共享内存

RT-Thread 共享内存开始的地址及大小

共享内存区域具体分配

Path: rtos/bsp/rockchip/rkxxxx-32/gcc_arm.ld.S

```

.share_lock (NOLOAD):
{
    . = ALIGN(64);
    PROVIDE(__spinlock_mem_start__ = .);
    . += __SPINLOCK_MEM_SIZE;
    PROVIDE(__spinlock_mem_end__ = .);
    . = ALIGN(64);
} > SHMEM

.share_rpmsg (NOLOAD):
{
    . = ALIGN(0x1000);
    PROVIDE(__share_rpmsg_start__ = .);
    . += __SHARE_RPMSG_SIZE;
    PROVIDE(__share_rpmsg_end__ = .);
    . = ALIGN(0x1000);
} > SHMEM

.share_data :
{
    . = ALIGN(64);
    PROVIDE(__share_data_start__ = .);
    KEEP(*(.share_data))
    PROVIDE(__share_data_end__ = .);
    . = ALIGN(64);
} > SHMEM AT > DRAM

```

HAL 共享内存开始的地址及大小

共享内存区域具体分配

Path: hal/project/rkxxxx/GCC/gcc_arm.ld.S

```
.share_lock (NOLOAD) :
{
    . = ALIGN(64);
    PROVIDE(__spinlock_mem_start__ = .);
    . += __SPINLOCK_MEM_SIZE;
    PROVIDE(__spinlock_mem_end__ = .);
    . = ALIGN(64);
} > SHMEM

.share_rpmsg (NOLOAD):
{
    . = ALIGN(0x1000);
    PROVIDE(__share_rpmsg_start__ = .);
    . += SHRPMSG_SIZE;
    PROVIDE(__share_rpmsg_end__ = .);
    . = ALIGN(0x1000);
} > SHMEM

.share_ramfs (NOLOAD):
{
    . = ALIGN(0x1000);
    PROVIDE(__share_ramfs_start__ = .);
    . += SHRAMFS_SIZE;
    PROVIDE(__share_ramfs_end__ = .);
    . = ALIGN(0x1000);
} > SHMEM

.share_log (NOLOAD):
{
    . = ALIGN(64);
    PROVIDE(__share_log0_start__ = .);
    . += SHLOG0_SIZE;
    PROVIDE(__share_log0_end__ = .);

    . = ALIGN(64);
    PROVIDE(__share_log1_start__ = .);
    . += SHLOG1_SIZE;
    PROVIDE(__share_log1_end__ = .);

    . = ALIGN(64);
    PROVIDE(__share_log2_start__ = .);
    . += SHLOG2_SIZE;
    PROVIDE(__share_log2_end__ = .);

    . = ALIGN(64);
    PROVIDE(__share_log3_start__ = .);
    . += SHLOG3_SIZE;
    PROVIDE(__share_log4_end__ = .);
    . = ALIGN(64);
} > SHMEM
```

6.5.2.2 测试demo

6.5.2.2.1 RTT Demo

Path: SDK/rtos/bsp/rockchip/rkxxxx-32

配置菜单配置: scones --menuconfig

```
CONFIG_RT_USING_RPMSG_LITE=y
CONFIG_RT_USING_COMMON_TEST_RPMSG_LITE=y
```

RTT Demo Path: rtos/bsp/rockchip/common/tests/rpmsg_test.c

RPMsg-lite 的核心代码位于: rtos/bsp/rockchip/common/drivers/rpmsg-lite 目录下。其具体接口函数说明如下:

函数	说明
rpmsg_lite_master_init()	RPMsg-lite master 端初始化
rpmsg_lite_remote_init()	RPMsg-lite remote 端初始化
rpmsg_lite_wait_for_link_up()	RPMsg-lite remote 端等待初始化链接成功
rpmsg_queue_create()	RPMsg-lite创建队列
rpmsg_lite_create_ept()	创建端点
rpmsg_queue_recv()	接收到的数据复制到本地buffer
rpmsg_queue_recv_nocopy()	接收到的数据直接传递指针
rpmsg_lite_send()	发送消息

6.5.2.2.2 HAL Demo

File: SDK/hal/project/rkxxxx/src/main.c

```
#define TEST_DEMO
#define TEST_USE_RPMSG_INIT
```

File: SDK/hal/project/rkxxxx/src/test_demo.c

```
#define RPMSG_TEST
```

HAL Demo Path: hal/project/rkxxxx/src/test_demo.c

RPMsg-lite 的核心代码位于: hal/middleware/rpmsg-lite/ 目录下。其具体接口函数说明如下:

函数	说明
rpmsg_lite_master_init()	RPMsg-lite master 端初始化
rpmsg_lite_remote_init()	RPMsg-lite remote 端初始化
rpmsg_lite_wait_for_link_up()	RPMsg-lite remote 端等待初始化链接成功
rpmsg_lite_create_ept()	创建端点
rpmsg_lite_send()	发送消息

6.5.2.2.3 测试结果

在串口终端输入串口命令查看 log 信息。

```
## 第6章 RPMSG 测试命令
msh >rpmsg_master_test

## 第6章 测试结果
[ (1)21.952.622] rpmsg probe remote cpu(0) ept(0x80008000) sucess!
[ (1)22.031.235] rpmsg probe remote cpu(2) ept(0x80008002) sucess!
[ (1)22.086.368] rpmsg probe remote cpu(3) ept(0x80008003) sucess!
[ (1)22.086.410] rpmsg_master_send: master[1]-->remote[0], remote ept addr =
0x80008000
[ (0)22.152.780]rpmsg_remote_rcv: remote[0]<--master[1], master ept addr =
0x80000000
[ (0)22.152.959]rpmsg_remote_send: remote[0]-->master[1], master ept addr =
0x80000000
[ (1)22.153.616] rpmsg_master_rcv: master[1]<--remote[0], remote ept addr =
0x80008000
[ (1)22.154.272] rpmsg_master_send: master[1]-->remote[2], remote ept addr =
0x80008002
[ (2)22.231.397]rpmsg_remote_rcv: remote[2]<--master[1], master ept addr =
0x80000002
[ (2)22.231.580]rpmsg_remote_send: remote[2]-->master[1], master ept addr =
0x80000002
[ (1)22.232.237] rpmsg_master_rcv: master[1]<--remote[2], remote ept addr =
0x80008002
[ (1)22.232.893] rpmsg_master_send: master[1]-->remote[3], remote ept addr =
0x80008003
[ (3)22.286.525]rpmsg_remote_rcv: remote[3]<--master[1], master ept addr =
0x80000003
[ (3)22.286.706]rpmsg_remote_send: remote[3]-->master[1], master ept addr =
0x80000003
[ (1)22.287.363] rpmsg_master_rcv: master[1]<--remote[3], remote ept addr =
0x80008003
[ (1)22.288.017] rpmsg test OK!
```

7. 第7章 中断

7.1 Cortex-A GIC

Cortex-A GIC（Generic Interrupt Controller）是一种用于处理中断请求的模块。它的作用是管理和分配各种类型的中断，并将它们发送给处理器核心以执行相应的中断服务程序。

GIC 可以分为 GICv2、GICv3 版本，常用芯片平台的支持情况如下：

Chip	GICv2	GICv3
RK3588		✓
Rk3576	✓	
RK3568		✓
RK3562	✓	
RK3358	✓	
RK3308	✓	

GIC中包含三种不同类型的中断：

1. SGI: 中断号 0-15，软件产生的中断，每个 CPU 私有。
2. PPI: 中断号 16-31，私有的外设中断，每个 CPU 私有。
3. SPI: 中断号 32 起，公共的外设中断，所有 CPU 共享。

三种中断配合，实现了各种丰富的应用。同时，各种配置文件中，也按不同的中断分组给予不同的分组偏移。

中断使用步骤包含以下几个方面：

1. GIC 中断配置：配置指定中断号对应的中断优先级，以及中断服务程序由哪个 CPU 来运行。
2. GIC 中断服务程序注册：注册指定中断号对应的中断服务程序。
3. GIC 中断使能：使能中断，系统能收到中断，并响应。
4. 配置外设模块中断，使模块能产生中断。

以在 RK3562 Bare-metal 和 RTOS 中配置 GPIO 中断为例，简要说明要如何配置。

7.1.1 GIC 中断配置

在 Bare-metal 和 RTOS 中，找到 `irqsConfig` 结构体的定义位置：

- RTOS: `<AMP_SDK>/rtos/bsp/rockchip/rk3562/board/common/board_base.c`
- Bare-metal: `<AMP_SDK>/hal/project/rk3562/src/main.c` 或者 `<AMP_SDK>/hal/project/rk3308/src/main.c`，依照编译的宏，使用不同的配置

代码结构为：

```
#define DEFAULT_IRQ_CPU 1 /* 指向主系统核心，依据实际修改 */
```

```

struct GIC_AMP_IRQ_INIT_CFG irqConfig[] = {
    GIC_AMP_IRQ_CFG_ROUTE(GPIO0_IRQn, 0xd0, CPU_GET_AFFINITY(0, 0)), /* 添加 CPU0
响应 CPU0 中断的路径 */
    GIC_AMP_IRQ_CFG_ROUTE(0, 0, CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0)), /*
sentinel */
};

struct GIC_IRQ_AMP_CTRL irqConfig = {
    .cpuAff = CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0),
    .defPrio = 0xd0,
    .defRouteAff = CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0),
    .irqsCfg = &irqConfig[0],
};

int main()
{
    HAL_GIC_Init(&irqConfig);
    // .....
}

```

在多系统 Bare-metal 和 RTOS 配置中，所有系统共享这张表格。

用户可修改的位置：

- DEFAULT_IRQ_CPU: 默认响应所有中断的 CPU。irqConfig 未配置到的中断，都由 DEFAULT_IRQ_CPU 响应。
- GIC_AMP_IRQ_CFG_ROUTE(irqNum, Priority, CPU_GET_AFFINITY(cpuID, cpuCluster)) 参数说明：

参数	说明
irqNum	中断号
Priority	优先级 (使用默认值，不需要修改)
cpuID	响应中断的 CPU 号
cpuCluster	cpu 集群，多见于大小核系统中。RK3562，始终为 0

7.1.2 GIC 中断服务程序

7.1.2.1 Bare-metal GIC 中断服务程序

下面以 GPIO0 的 C4 pin 脚设置上升沿触发为例，简单说明 GPIO 中断使用方法。

```

// GPIO0 中断服务程序总入口
static void gpio_isr(int vector, void *param)
{
    // .....
    HAL_GPIO_IRQHandler(GPIO0, GPIO_BANK0);
    // .....
}

```

```

// GPIO0 C4 pin 脚中断回调函数
static HAL_Status c4_call_back(eGPIO_bankId bank, uint32_t pin, void *args)
{
    // .....
    return HAL_OK;
}

// GPIO 脚中断使用示例
static void gpio_test(void)
{
    // .....

    /* Step 1: GIC 设置 */
    /* 设置 GIC (GPIO0) 中断服务程序, 使能中断, 使系统能收到模块中断 */
    HAL_IRQ_HANDLER_SetIRQHandler(GPIO0_IRQn, gpio_isr, NULL);
    HAL_GIC_Enable(GPIO0_IRQn);

    /* Step 2: 模块设置 */
    /* 设置 GPIO0 C4 为输入口 */
    HAL_GPIO_SetPinDirection(GPIO0, GPIO_PIN_C4, GPIO_IN);
    /* 设置 GPIO0 C4 中断类型、回调函数, 并且使能 GPIO0 C4 的 IO 中断 */
    HAL_IRQ_HANDLER_SetGpioIRQHandler(GPIO_BANK0, GPIO_PIN_C4, c4_call_back,
    NULL);
    HAL_GPIO_SetIntType(GPIO0, GPIO_PIN_C4, GPIO_INT_TYPE_EDGE_RISING);
    /* 使能 GPIO 中断, 使模块能产生中断 */
    HAL_GPIO_EnableIRQ(GPIO0, GPIO_PIN_C4);
}

```

示例中, 中断配置分为两部分:

- GIC 设置: 配置中断响应函数 `gpio_isr`, 并使能系统中断接收。该部分内容, 对所有中断通用, 统一配置接口。
- 模块设置: 配置模块中断, 使模块能产生中断信号给GIC模块。该部分内容, 主要依照模块自己的规则, 有较大的不一样, 需要详细参考模块说明, 编写代码。

因为 GPIO 的中断是 32 个引脚共用一个 GPIO 中断信号, 所以是示例中, 中断服务中增加了 `HAL_GPIO_IRQHandler` 函数, 用来配发具体引脚的回调函数。

7.1.2.2 RTOS GIC 中断服务程序

RTOS 中, 可以使用 Bare-metal 接口, 使用Bare-metal GIC 中断服务程序注册的例子。也可以使用 RTOS 官方封装的接口。同样使用使用 GPIO0 的 C4 pin 脚设置上升沿触发的例子, RTOS 的设置为:

```

// GPIO 脚中断使用示例
void irq_callback(void *args)
{
    // .....
}

static void gpio_test(void)
{
    struct rt_device_pin_mode pin_mode;
    rt_device_t pin_dev = rt_device_find("pin");

    rt_device_open(pin_dev, RT_DEVICE_FLAG_RDWR);
}

```

```

pin_mode.pin = BANK_PIN(0, GPIO_PIN_C4); /* GPIO0_C4 */
pin_mode.mode = PIN_MODE_INPUT;
rt_device_control(pin_dev, 0, &pin_mode);
rt_pin_attach_irq(pin_mode.pin, PIN_IRQ_MODE_RISING, irq_callback, RT_NULL);
rt_pin_irq_enable(pin_mode.pin, PIN_IRQ_ENABLE);
}

```

模块不一样，差异较大，具体参考RT-Thread官方文档。

7.2 Cortex-M NVIC

Cortex-M NVIC（Nested Vectored Interrupt Controller）是处理器中用于管理中断的关键组件。它负责管理和分配来自外部和内部源的中断请求，并将它们发送给适当的处理器核心进行处理。

支持的平台有：RK3588、RK3576、RK3562，主要核心是 Cortex-M0 系列。

Cortex-M0 系列中断最大接入为 32 个中断。意味着多出的中断需要进行二级轮询。为此，引入 INTMUX 机制，尽可能的引入更多中断，方便软件开发。

NVIC 中断使用步骤包含以下几个方面：

1. NVIC 中断初始化：初始化中断向量和NVIC控制器。
2. NVIC 中断服务程序注册：注册指定中断号对应的中断服务程序。
3. NVIC 中断使能：使能中断，系统能收到中断，并响应。
4. 配置外设模块中断，使模块能产生中断。

以在 RK3562 Bare-metal MCU 和 RTOS MCU 中配置 GPIO 中断为例，简要说明要如何配置。

7.2.1 NVIC 中断初始化

在 Bare-metal 和 RTOS 中，NVIC 中断初始化已经被包含在 `HAL_Init()` 中了，直接调用 `HAL_Init` 或单独提取 NVIC 操作都可以实现初始化。

```

/* <AMP_SDK>/hal/lib/hal/src/hal_base.c */

HAL_Status HAL_Init(void)
{
#ifdef __CORTEX_M
#ifdef HAL_NVIC_MODULE_ENABLED
    /* Set Interrupt Group Priority */
    HAL_NVIC_Init();

    /* Set Interrupt Group Priority */
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_DEFAULT);
#endif
#endif
    // .....
    return HAL_OK;
}

```

7.2.2 NVIC 中断服务程序

7.2.2.1 Bare-metal GIC 中断服务程序

下面以 GPIO0 的 C4 pin 脚设置上升沿触发为例，简单说明 GPIO 中断使用方法。

```
// GPIO0 中断服务程序总入口
static void gpio_isr(int vector, void *param)
{
    // .....
    HAL_GPIO_IRQHandler(GPIO0, GPIO_BANK0);
    // .....
}

// GPIO0 C4 pin 脚中断回调函数
static HAL_Status c4_call_back(eGPIO_bankId bank, uint32_t pin, void *args)
{
    // .....
    return HAL_OK;
}

// GPIO 脚中断使用示例
static void gpio_test(void)
{
    // .....

    /* Step 1: GIC 设置 */
    /* 设置 NVIC (GPIO0) 中断服务程序，使能中断，使系统能收到模块中断 */
    HAL_INTMUX_SetIRQHandler(GPIO0_IRQn, gpio_isr, NULL);
    HAL_INTMUX_EnableIRQ(GPIO0_IRQn);

    /* Step 2: 模块设置 */
    /* 设置 GPIO0 C4 为输入口 */
    HAL_GPIO_SetPinDirection(GPIO0, GPIO_PIN_C4, GPIO_IN);
    /* 设置 GPIO0 C4 中断类型、回调函数，并且使能 GPIO0 C4 的 IO 中断 */
    HAL_IRQ_HANDLER_SetGpioIRQHandler(GPIO_BANK0, GPIO_PIN_C4, c4_call_back,
    NULL);
    HAL_GPIO_SetIntType(GPIO0, GPIO_PIN_C4, GPIO_INT_TYPE_EDGE_RISING);
    /* 使能 GPIO 中断，使模块能产生中断 */
    HAL_GPIO_EnableIRQ(GPIO0, GPIO_PIN_C4);
}
```

示例中，中断配置分为两部分：

- NVIC 设置: 配置中断响应函数 `gpio_isr`，并使能系统中断接收。该部分内容，对所有中断通用，统一配置接口。该函数中，因为使用了 INTMUX 机制，所以使用 `HAL_INTMUX***` 对原来 NVIC 接口进行了一次封装，使用上和 NVIC 原函数无异。
- 模块设置: 配置模块中断，使模块能产生中断信号给 NVIC 模块。该部分内容，主要依照模块自己的规则，有较大的不一样，需要详细参考模块说明，编写代码。

因为 GPIO 的中断是 32 个引脚共用一个 GPIO 中断信号，所以是示例中，中断服务中增加了

`HAL_GPIO_IRQHandler` 函数，用来配发具体引脚的回调函数。

7.2.2.2 RTOS NVIC 中断服务程序

参考[RTOS GIC 中断服务程序](#)

7.3 RISC-V中断控制器

RISC-V IPIC (Integrated Programmable Interrupt Controller) 是处理器中用于管理中断的关键组件。它负责管理和分配来自外部和内部源的中断请求，并将它们发送给适当的处理器核心进行处理。

支持的平台有：RK3568，主要核心是 RISC-V 系列。

RISC-V 系列中断最大接入为 32 个中断。意味着多出的中断需要进行二级轮询。为此，引入 INTMUX 机制，尽可能的引入更多中断，方便软件开发。

IPIC 中断使用步骤包含以下几个方面：

1. IPIC 中断初始化：初始化中断向量表和IPIC控制器。
2. IPIC 中断服务程序注册：注册指定中断号对应的中断服务程序。
3. IPIC 中断使能：使能中断，系统能收到中断，并响应。
4. 配置外设模块中断，使模块能产生中断。

以在 RK3568 Bare-metal RISC-V和 RTOS RISC-V中配置 GPIO 中断为例，简要说明要如何配置。

7.3.1 IPIC 中断初始化

在 Bare-metal 和 RTOS 中，IPIC 中断初始化已经被包含在 `HAL_INTMUX_Init()` 中了，直接调用 `HAL_INTMUX_Init()` 或单独提取 IPIC 操作都可以实现初始化。

```
/* <AMP_SDK>/hal/lib/hal/src/hal_intmux.c */

HAL_Status HAL_INTMUX_Init(void)
{
    // .....
#ifdef HAL_RISCVIC_MODULE_ENABLED
    HAL_RISCVIC_Init();
#endif
    // .....
    return HAL_OK;
}
```

7.3.2 IPIC 中断服务程序

7.3.2.1 Bare-metal GIC 中断服务程序

下面以 GPIO4 的 C5 pin 脚设置上升沿触发为例，简单说明 GPIO 中断使用方法。

```
// GPIO0 中断服务程序总入口
static void gpio_isr(int vector, void *param)
{
    // .....
```

```

    HAL_GPIO_IRQHandler(GPIO4, GPIO_BANK4);
    // .....
}

// GPIO0 C4 pin 脚中断回调函数
static HAL_Status c5_call_back(eGPIO_bankId bank, uint32_t pin, void *args)
{
    // .....
    return HAL_OK;
}

// GPIO 脚中断使用示例
static void gpio_test(void)
{
    // .....

    /* Step 1: GIC 设置 */
    /* 设置 IPIC (GPIO4) 中断服务程序, 使能中断, 使系统能收到模块中断 */
    HAL_INTMUX_SetIRQHandler(GPIO4_IRQn, gpio_isr, NULL);
    HAL_INTMUX_EnableIRQ(GPIO4_IRQn);

    /* Step 2: 模块设置 */
    /* 设置 GPIO4 C5 为输入口 */
    HAL_GPIO_SetPinDirection(GPIO4, GPIO_PIN_C5, GPIO_IN);
    /* 设置 GPIO4 C5 中断类型、回调函数, 并且使能 GPIO4 C5 的 IO 中断 */
    HAL_IRQ_HANDLER_SetGpioIRQHandler(GPIO_BANK4, GPIO_PIN_C5, c5_call_back,
    NULL);
    HAL_GPIO_SetIntType(GPIO4, GPIO_PIN_C5, GPIO_INT_TYPE_EDGE_RISING);
    /* 使能 GPIO 中断, 使模块能产生中断 */
    HAL_GPIO_EnableIRQ(GPIO4, GPIO_PIN_C5);
}

```

示例中, 中断配置分为两部分:

- IPIC 设置: 配置中断响应函数 `gpio_isr`, 并使能系统中断接收。该部分内容, 对所有中断通用, 统一配置接口。该函数中, 因为使用了 INTMUX 机制, 所以使用 `HAL_INTMUX_Init` 对原来 IPIC 接口进行了一次封装, 使用上和 IPIC 原函数无异。
- 模块设置: 配置模块中断, 使模块能产生中断信号给 IPIC 模块。该部分内容, 主要依照模块自己的规则, 有较大的不一样, 需要详细参考模块说明, 编写代码。

因为 GPIO 的中断是 32 个引脚共用一个 GPIO 中断信号, 所以是示例中, 中断服务中增加了

`HAL_GPIO_IRQHandler` 函数, 用来配发具体引脚的回调函数。

7.3.2.2 RTOS IPIC 中断服务程序

参考[RTOS GIC 中断服务程序](#)

8. 第8章 模块

8.1 eMMC

8.1.1 HAL

根据硬件控制器不同，eMMC 驱动分为 SDIO 和 SDHCI，源码位于 `hal/lib/src/` 和 `hal/middleware/sdhci/`，HAL 提供基础的读写接口。

以 RK3568 为例：

```
#include "mmc_api.h"

#define TestSector 8
#define maxTestSector (TestSector * 4)
static int pWriteBuf[maxTestSector * 128];
static int pReadBuf[maxTestSector * 128];
static int userCapSize;

static int SdhciInit(void)
{
    int ioctlParam[5] = {0, 0, 0, 0, 0};
    int ret;

    sdmmc_init((void *)0xFE310000);
    ret = sdmmc_ioctl(SDM_IOCTL_REGISTER_CARD, ioctlParam);
    if (ret) {
        printf("emmc init error!\n");
        return -1;
    }

    ret = sdmmc_ioctl(SDM_IOCTL_GET_CAPABILITY, ioctlParam);
    if (ret) {
        printf("emmc get capability error!\n");
        return -1;
    }

    userCapSize = ioctlParam[1];
}

static int SdhciTest(void)
{
    int i, j, loop = 0;
    int testEndLBA;
    int testLBA = 0;
    int testSecCount = 1;
    int printFlag;

    testEndLBA = userCapSize / 32;

    for (i = 0; i < (maxTestSector * 128); i++)
        pWriteBuf[i] = i;
```

```

for (loop = 0; loop < 2; loop++) {
    HAL_DBG("-----Test loop = %d-----\n", loop);
    HAL_DBG("-----Test ftl write %s-----\n", "");
    testSecCount = 1;
    HAL_DBG("testEndLBA = %x\n", testEndLBA);
    HAL_DBG("testLBA = %x\n", testLBA);

    for (testLBA = 0x10000 + loop; (testLBA + testSecCount) < testEndLBA;) {
        sdmmc_write(testLBA, testSecCount, pWriteBuf);
        sdmmc_read(testLBA, testSecCount, pReadBuf);
        printFlag = testLBA & 0x1FF;

        if (printFlag < testSecCount)
            HAL_DBG("testLBA = %x\n", testLBA);

        for (j = 0; j < testSecCount * 128; j++) {
            if (pWriteBuf[j] != pReadBuf[j]) {
                printf("write not match:row=%x, num=%x, write=%x, read=%x\n",
testLBA, j, pWriteBuf[j], pReadBuf[j]);
                while (1);
            }
        }

        testLBA += testSecCount;
        testSecCount++;

        if (testSecCount > maxTestSector)
            testSecCount = 1;
    }

    HAL_DBG("-----Test ftl check-----%s\n", "");
    testSecCount = 1;

    for (testLBA = 0x10000 + loop; (testLBA + testSecCount) < testEndLBA;) {
        sdmmc_read(testLBA, testSecCount, pReadBuf);
        printFlag = testLBA & 0x7FF;

        if (printFlag < testSecCount)
            HAL_DBG("testLBA = %x\n", testLBA);

        for (j = 0; j < testSecCount * 128; j++) {
            if (pWriteBuf[j] != pReadBuf[j]) {
                printf("check not match:row=%x, num=%x, write=%x,
read=%x\n", testLBA, j, pWriteBuf[j], pReadBuf[j]);
                while (1);
            }
        }

        testLBA += testSecCount;
        testSecCount++;

        if (testSecCount > maxTestSector)
            testSecCount = 1;
    }
}

HAL_DBG("-----Test end---%s-----\n", "");

```

```
    return 0;
}
```

8.1.2 RT-Thread

RT-Thread 为 SDIO 驱动提供了文件系统的支持，SDHCI 提供基础的块读写接口。

SDIO 配置：

Menuconfig 配置入口：

```
RT-Thread Components --->
  Device Drivers --->
    [*] Using SD/MMC device drivers
```

```
RT_USING_SDIO=y
RT_USING_SDIO0=y

RT_USING_DMA=y
RT_USING_DMA_PL330=y
RT_USING_DMA0=y
```

SDHCI 配置：

```
RT_USING_SDHCI=y
```

RT-Thread elm-fat 文件支持：

Menuconfig 配置入口：

```
RT-Thread Components --->
  Device virtual file system --->
    [*] Using device virtual file system
    [*] Using mount table for file system /* 实现相应注册分区表，可实现分区上电自动挂
载 */
    [*] Enable elm-chan fatfs /* fat 文件系统 */
      elm-chan's FatFs, Generic FAT Filesystem Module --->
        (4096) Maximum sector size to be handled. /*对于 SPI Nor 产品必须修改为 4096
*/
```

```
RT_USING_DFS=y
RT_SDCARD_MOUNT_POINT="/"
DFS_FILESYSTEMS_MAX=4
DFS_FILESYSTEM_TYPES_MAX=4
RT_USING_DFS_MNNTABLE=y
RT_USING_DFS_ELMFAT=y
RT_DFS_ELM_MAX_SECTOR_SIZE=4096
```

RT-Thread 会根据 mount_table 来挂载存储中的文件系统，如开启分区自动挂载文件系统，可在 mnt.c 中添加相应分区注册信息，例如“root”分区到“/”目录：

```
const struct dfs_mount_tbl mount_table[] =
{
    {"root", "/", "elm", 0, 0},
    {0}
};
```

如希望自行设计文件系统挂载流程，也可以通过以下代码实现文件系统挂载：

```
dfs_mount("root", "/", "elm", 0, 0)
```

如果 eMMC 中没有对应的文件系统，可以对文件系统格式化，通过 mount 挂载：

```
mkfs -t elm sd0      # sd0 格式化为 elm-FAT 文件系统
mount sd0 / elm      # 在 / 目录挂载 sd0 为 elm-FAT
```

文件系统挂载成功后，通过文件系统串口命令操作，验证文件系统功能：

```
## 第8章 在根目录下创建一个文件
echo "This is a test!" /test.txt

## 第8章 查看目录
ls
Directory /:
test.txt

## 第8章 查看文件内容
cat test.txt
This is a test!
```

8.1.3 Kernel

Kernel eMMC 详细使用方法可以参考《Rockchip_Developer_Guide_SDMMC_SDIO_eMMC_CN.pdf》

8.2 UART

8.2.1 HAL

HAL 中 UART 配置主要分为如下几步：

1. 配置 IOMUX
2. 在中断表中配置 UART 中断
3. 调用初始化接口

以RK3562为例：

```
static void HAL_IOMUX_Uart7M1Config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK1,
                          GPIO_PIN_B3,
                          PIN_CONFIG_MUX_FUNC3);
```

```

        HAL_PINCTRL_SetIOMUX(GPIO_BANK1,
                               GPIO_PIN_B4,
                               PIN_CONFIG_MUX_FUNC3);
    }

    static struct GIC_AMP_IRQ_INIT_CFG irqsConfig[] = {
/* TODO: Config the irqs here.
 * GIC version: GICv2
 */
        GIC_AMP_IRQ_CFG_ROUTE(UART7_IRQn, 0xd0, CPU_GET_AFFINITY(1, 0)),

        GIC_AMP_IRQ_CFG_ROUTE(0, 0, CPU_GET_AFFINITY(1, 0)),    /* sentinel */
    };

    void main(void)
    {
        struct HAL_UART_CONFIG hal_uart_config = {
            .baudRate = UART_BR_1500000,
            .dataBit = UART_DATA_8B,
            .stopBit = UART_ONE_STOPBIT,
            .parity = UART_PARITY_DISABLE,
        };

        HAL_IOMUX_Uart7M1Config();
        HAL_UART_Init(&g_uart7Dev, &hal_uart_config);
    }

```

8.2.2 RT-Thread

RT-Thread 中 UART 配置主要分为如下几步：

1. `scons --menuconfig` 打开UART支持
2. 配置对应的 IOMUX
3. 配置 `g_uart_board` 信息，包括波特率等
4. 在中断表中配置 UART 中断

RT-Thread 中已经对部分 UART 进行上面所诉的完整配置，可以直接通过 `scons --menuconfig` 打开使用，以 RK3562 为例：

Menuconfig 配置入口：

```

RT-Thread rockchip RK3562 drivers  --->
  Enable UART  --->
    [*] Enable UART
    [*] Enable UART0

```

```

## 第8章 UART0
RT_CONSOLE_DEVICE_NAME="uart0"
RT_USING_UART=y
RT_USING_UART0=y

## 第8章 UART7
RT_CONSOLE_DEVICE_NAME="uart7"
RT_USING_UART=y
RT_USING_UART7=y

```

```

/* 配置对应的 IOMUX */
#ifdef RT_USING_UART0
RT_WEAK void uart0_m0_iomux_config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK0,
                          GPIO_PIN_D0 |
                          GPIO_PIN_D1,
                          PIN_CONFIG_MUX_FUNC1);
}
#endif

#ifdef RT_USING_UART7
RT_WEAK void uart7_m1_iomux_config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK1,
                          GPIO_PIN_B3 |
                          GPIO_PIN_B4,
                          PIN_CONFIG_MUX_FUNC3);
}
#endif

/* 调用 IOMUX */
void rt_hw_iomux_config(void)
{
    rt_hw_iodomain_config();

#ifdef RT_USING_UART0
    uart0_m0_iomux_config();
#endif

#ifdef RT_USING_UART7
    uart7_m1_iomux_config();
#endif

#ifdef RT_USING_GMAC
#ifdef RT_USING_GMAC0
    gmac0_m0_iomux_config();
#endif
#endif

/* 配置 g_uart_board 信息 */
#if defined(RT_USING_UART0)
RT_WEAK const struct uart_board g_uart0_board =
{
    .baud_rate = UART_BR_1500000,
    .dev_flag = ROCKCHIP_UART_SUPPORT_FLAG_DEFAULT,
    .bufer_size = RT_SERIAL_RB_BUFSZ,
    .name = "uart0",
};
#endif /* RT_USING_UART0 */

/* 在中断表中配置UART中断 */
static struct GIC_AMP_IRQ_INIT_CFG irqsConfig[] =
{
    /* Config the irqs here. */
    // todo...
    GIC_AMP_IRQ_CFG_ROUTE(UART0_IRQn, 0xd0, CPU_GET_AFFINITY(3, 0)),

```

```
GIC_AMP_IRQ_CFG_ROUTE(0, 0, CPU_GET_AFFINITY(1, 0)), /* sentinel */
};
```

8.2.3 Kernel

Kernel 的 DTS 对应的平台 `kernel/arch/arm64/boot/dts/rockchip/rkxxxx.dtsi` 中具备所有的 UART 配置，在使用时开启即可，Kernel UART 详细细节请参考《Rockchip_Developer_Guide_UART_CN.pdf》。

在 AMP 系统中为了防住出现外设资源冲突的情况，Kernel DTS 中需要对其他系统使用的资源进行隔离关闭，UART 也是如此，所以在 `rkxxxx-amp.dtsi` 中需要屏蔽被其他系统使用的 UART，同时将 UART 中断配置给其使用的 CPU

```
/* DTS 中将UART7中断(69)route到CPU3，同时声明UART7的时钟，如果其他 DTS 使用时钟会报错，以此做资源隔离 */
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        clocks = <&cru FCLK_BUS_CM0_CORE>, <&cru CLK_BUS_CM0_RTC>,
                <&cru PCLK_MAILBOX>, <&cru PCLK_INTC>,
                <&cru SCLK_UART7>, <&cru PCLK_UART7>,
                <&cru PCLK_TIMER>, <&cru CLK_TIMER4>, <&cru CLK_TIMER5>;

        pinctrl-names = "default";
        pinctrl-0 = <&uart7m1_xfer>;

        amp-cpu-aff-maskbits = /bits/ 64 <0x0 0x1 0x1 0x2 0x2 0x4 0x3 0x8>;
        amp-irqs = /bits/ 64 <GIC_AMP_IRQ_CFG_ROUTE(147, 0xd0,
CPU_GET_AFFINITY(3, 0))
                                GIC_AMP_IRQ_CFG_ROUTE(69, 0xd0, CPU_GET_AFFINITY(3,
0))>;

        status = "okay";
    };
    /* 在 DTS 中禁用 UART7 */
    &uart7 {
        status = "disabled";
    }
}
```

8.3 SPI FLASH

SPI FLASH 详细使用方法可以参考《Rockchip_Developer_Guide_RT-Thread_SPIFLASH_CN.pdf》

RK 平台 SPI Flash 可选用的控制器包括 FSPI、SFC、SPI 三种方案。

FSPI (Flexible Serial Peripheral Interface) 是一个灵活的串行传输控制器，有以下主要特性：

- 支持 SPI Nor、SPI Nand、SPI 协议的 Psram 和 SRAM
- 支持 Standard SPI（单线）、Dual SPI、Quad SPI，部分版本支持 Octal SPI
- 支持 SDR（单沿传输），部分版本支持 DTR（双沿传输）
- XIP 技术
- DMA 传输（内置 DMA）

SFC (Serial Flash Controller) 是串行传输控制器，有以下主要特性：

- 支持 SPI Nor、SPI Nand、SPI 协议的 Psram 和 SRAM
- 支持 Standard SPI（单线）、Dual SPI、Quad SPI
- 支持 SDR（单沿传输）
- DMA 传输（内置 DMA）

SPI（Serial Peripheral Interface）为通用的串行传输控制器，有以下主要特性：

- 支持 SPI Nor、SPI Nand、SPI 协议的 Psram
- 支持 Standard SPI（单线）
- 支持 SDR（单沿传输）
- DMA 传输（外部 DMA）

8.3.1 HAL

RK HAL 提供基于 SPI Nor 传输协议的 HAL_SNOR 协议层，HAL 源码录在 `hal/lib/hal/src/`
RK HAL 在 `hal/test/hal/` 下提供了接口使用的 Demo，用户可以参考 HAL 下如何读写 SPI FLASH

由于 SPI FLASH 型号种类繁多，软件通过 flash id 识别特定颗粒，已支持的 SPI FLASH 颗粒可以查询源码，如下所示：

```
HAL_SECTION_SRAM_CODE static const struct FLASH_INFO s_spiFlashbl[] = {
    /* GD25LQ16E */
    { 0xc86015, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0D, 12, 9, 0 },
    /* GD25Q32B */
    { 0xc84016, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0D, 13, 9, 0 },
    /* GD25Q64B */
    { 0xc84017, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0D, 14, 9, 0 },
    /* GD25Q127C and GD25Q128C */
    { 0xc84018, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0C, 15, 9, 0 },
    /* GD25Q256B/C/D */
    { 0xc84019, 128, 8, 0x13, 0x12, 0x6C, 0x3E, 0x21, 0xDC, 0x1C, 16, 6, 0 },
    /* GD55LT01GE */
    { 0xc8661b, 128, 8, 0x13, 0x12, 0x6B, 0x32, 0x20, 0xD8, 0x3C, 18, 0, 0 },
    /* GD25LQ64C */
    { 0xc86017, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0D, 14, 9, 0 },
    /* GD25LQ32E */
    { 0xc86016, 128, 8, 0x03, 0x02, 0x6B, 0x32, 0x20, 0xD8, 0x0D, 13, 9, 0 },
    /* GD25LX256E */
    { 0xc86819, 128, 8, 0x13, 0x12, 0x00, 0x00, 0x21, 0xDC, 0x10, 16, 0, 0x0D },
}
```

配置说明：

```
#define HAL_SNOR_MODULE_ENABLED /* SNOR 支持 */
#define HAL_SFC_MODULE_ENABLED /* SFC 控制器支持 */
#define HAL_SNOR_SFC_HOST /* SFC 控制器支持 */
```

8.3.2 RT-Thread

基础配置

Menuconfig 配置入口：

```

RT-Thread rockchip common drivers --->
  [*] Enable ROCKCHIP SPI NOR Flash
  (80000000) Reset the speed of SPI Nor flash in Hz
  [ ] Set SPI Host DUAL IO Lines /* 如果 FSPI 主控仅预留 IO0~1,应使用 Dual mode
*/
      Choose SPI Nor Flash Adapter (Attach FSPI controller to SNOR) --->

```

配置说明：

```

RT_USING_MTD_NOR=y
RT_USING_SNOR=y
RT_SNOR_SPEED=80000000 /* IO 接口速率 */
## 第8章 RT_SNOR_DUAL_IO=n /* 默认不配置, Quad SPI 限制为 Duad SPI 使用 */
## 第8章 RT_SNOR_XIP_DATA_BEGIN=0 /* 默认不配置, XIP 读接口实现起始地址, 详细参考 “Nor
Flash XIP 技术”说明 */
RT_USING_SNOR_FSPI_HOST=y /* FSPI 控制器方案 */
## 第8章 RT_USING_SNOR_SFC_HOST=y /* SFC 控制器方案 */
## 第8章 RT_USING_SNOR_SPI_HOST=y /* SPI 控制器方案 */
## 第8章 RT_SNOR_SPI_DEVICE_NAME="spi2_0" /* SPI 控制器方案, 指定目标控制器 */

```

RT-Thread elm-fat 文件支持

```

RT-Thread Components --->
  Device virtual file system --->
  [*] Using device virtual file system
  [*] Using mount table for file system /* 实现相应注册分区表, 可实现分区上电自动挂
载 */
  [*] Enable elm-chan fatfs /* fat 文件系统 */
      elm-chan's FatFs, Generic FAT Filesystem Module --->
      (4096) Maximum sector size to be handled. /*对于 SPI Nor 产品必须修改为 4096
*/

```

如开启分区自动挂载文件系统，可在 mnt.c 中添加相应分区注册信息，例如“root”分区到“/”目录：

```

const struct dfs_mount_tbl mount_table[] =
{
    {"root", "/", "elm", 0, 0},
    {0}
};

```

如希望自行设计文件系统挂载流程，也可以通过以下代码实现文件系统挂载：

```
dfs_mount("root", "/", "elm", 0, 0)
```

可以通过以下命令查看相应分区是否注册成功：

```

msh />list_device
device          type          ref count
-----
root            Block Device      1          /* 分区名 root, 分区类型 block 设备, Nor
flash 支持 setting.ini 修改设定为 MTD 设备 */
snor            MTD Device       0          /* SPI Nor 根存储设备, 分区读写最终接入到该
节点完成读写擦除 */

```



```

        GPIO_PIN_A6 | /* RGMII_RXD1_M0 */
        GPIO_PIN_A7 | /* RGMII_RXDV_M0 */
        GPIO_PIN_B1 | /* ETH_CLK_25M_OUT_M0 */
        GPIO_PIN_B2 | /* RGMII_MDC_M0 */
        GPIO_PIN_B3 | /* RGMII_MDIO_M0 */
        GPIO_PIN_B7 , /* RGMII_CLK_M0 */
        PIN_CONFIG_MUX_FUNC2);
}

/* 配置 GMAC_ETH_CONFIG 表 */
static struct GMAC_ETH_CONFIG ethConfigTable[] =
{
    {
        .halDev = &g_gmac0Dev,
        .mode = PHY_INTERFACE_MODE_RGMII,
        .maxSpeed = 1000,
        .phyAddr = 0, /* PHY 地址 */
        .extClk = false, /* true 由 PHY 提供时钟输入 */
        .resetGpioBank = GPIO3, /* PHY reset 引脚 */
        .resetGpioNum = GPIO_PIN_A0,
        .resetDelayMs = { 0, 20, 100 }, /* PHY reset 时序 */
        .txDelay = 0x3C,
        .rxDelay = 0,
    },
};

/* 在中断表中配置 GMAC0_IRQn 中断 */
static struct GIC_AMP_IRQ_INIT_CFG irqConfig[] = {
/* TODO: Config the irqs here.
 * GIC version: GICv2
 * The priority higher than 0x80 is non-secure interrupt.
 */
    GIC_AMP_IRQ_CFG_ROUTE(GMAC0_IRQn, 0xd0, CPU_GET_AFFINITY(0, 0)),
};

```

8.4.2 RT-Thread

配置说明：

Menuconfig 配置入口：

```

RT-Thread rockchip RK3562 drivers  --->
  Enable GMAC  --->
    [*] Enable GMAC
    [*]   Enable GMAC0

```

```

RT_USING_GMAC=y      /* 打开 GMAC 配置 */
RT_USING_GMAC0=y     /* 打开 GMAC0 配置*/

```

RT_Thread 中 GMAC 主要分为如下几步：

1. 配置 IOMUX
2. 配置 rockchip_eth_config 表
3. 在中断表中配置 GMAC0_IRQn 中断（SMP 系统不需要配置）

以 RK3562 为例：

```
/* 配置 IOMUX */
RT_WEAK void gmac0_m0_iomux_config(void)
{
    /* GMAC0 iomux */
    HAL_PINCTRL_SetIOMUX(GPIO_BANK3,
                          GPIO_PIN_A0 | /* RGMII_RSTn */
                          GPIO_PIN_A1, /* RGMII_INT/PMEB_M0 */
                          PIN_CONFIG_MUX_FUNC0);

    HAL_GPIO_SetPinDirection(GPIO3, GPIO_PIN_A1, GPIO_IN);
    HAL_GPIO_SetPinDirection(GPIO3, GPIO_PIN_A0, GPIO_OUT);
    HAL_GPIO_SetPinLevel(GPIO3, GPIO_PIN_A0, GPIO_HIGH);

    HAL_PINCTRL_SetIOMUX(GPIO_BANK3,
                          GPIO_PIN_D4 | /* RGMII_TXD2_M0 */
                          GPIO_PIN_D5 | /* RGMII_TXD3_M0 */
                          GPIO_PIN_D6 | /* RGMII_TXCLK_M0 */
                          GPIO_PIN_D7, /* RGMII_RXD2_M0 */
                          PIN_CONFIG_MUX_FUNC2);

    HAL_PINCTRL_SetIOMUX(GPIO_BANK4,
                          GPIO_PIN_A0 | /* RGMII_RXD3_M0 */
                          GPIO_PIN_A1 | /* RGMII_RXCLK_M0 */
                          GPIO_PIN_A2 | /* RGMII_TXD0_M0 */
                          GPIO_PIN_A3 | /* RGMII_TXD1_M0 */
                          GPIO_PIN_A4 | /* RGMII_TXEN_M0 */
                          GPIO_PIN_A5 | /* RGMII_RXD0_M0 */
                          GPIO_PIN_A6 | /* RGMII_RXD1_M0 */
                          GPIO_PIN_A7 | /* RGMII_RXDV_M0 */
                          GPIO_PIN_B1 | /* ETH_CLK_25M_OUT_M0 */
                          GPIO_PIN_B2 | /* RGMII_MDC_M0 */
                          GPIO_PIN_B3 | /* RGMII_MDIO_M0 */
                          GPIO_PIN_B7, /* RGMII_CLK_M0 */
                          PIN_CONFIG_MUX_FUNC2);
}

/* 配置 rockchip_eth_config 表 */
const struct rockchip_eth_config rockchip_eth_config_table[] =
{
    {
        .halDev = &g_gmac0Dev,
        .mode = PHY_INTERFACE_MODE_RGMII,
        .maxSpeed = 1000,
        .phyAddr = 0, /* PHY 地址 */
        .extClk = false, /* true 由 PHY 提供时钟输入 */
        .resetGpioBank = GPIO3, /* PHY reset 引脚 */
        .resetGpioNum = GPIO_PIN_A0,
        .resetDelayMs = { 0, 20, 100 }, /* PHY reset 时序 */
        .txDelay = 0x3C,
        .rxDelay = 0,
    },
};
```

RT-Thread 提供 LWIP 的支持，可以通过 `scons --menuconfig`，打开对应功能的支持

Menuconfig 配置入口:

```
RT-Thread Components  --->
  Network  --->
    [*] LwIP: light weight TCP/IP stack  --->
    --- LwIP: light weight TCP/IP stack
    [ ] Use LwIP local version only (NEW)
    lwIP version (lwIP v2.0.3)  --->
    [ ] IPV6 protocol (NEW)
    (4) Memory alignment (NEW)
    [*] IGMP protocol (NEW)
    -* ICMP protocol
    [ ] SNMP protocol (NEW)
    [*] Enable DNS for name resolution (NEW)
    [*] Enable alloc ip address through DHCP (NEW)
    (1) SOF broadcast (NEW)
    (1) SOF broadcast recv (NEW)
    Static IPv4 Address  --->
```

```
NETDEV_USING_PING=y
RT_USING_LWIP=y
RT_USING_LWIP203=y
RT_USING_LWIP_VER_NUM=0x20003
RT_LWIP_MEM_ALIGNMENT=4
RT_LWIP_IGMP=y
RT_LWIP_ICMP=y
RT_LWIP_DNS=y
#开启DHCP, 静态IP可以不用配置
RT_LWIP_DHCP=y
IP_SOF_BROADCAST=1
IP_SOF_BROADCAST_RECV=1

/* Static IPv4 Address */
RT_LWIP_IPADDR="XXX.XXX.XXX.XXX"
RT_LWIP_GWADDR="XXX.XXX.XXX.XXX"
RT_LWIP_MSKADDR="XXX.XXX.XXX.XXX"
RT_LWIP_UDP=y
RT_LWIP_TCP=y
RT_LWIP_RAW=y

RT_MEMP_NUM_NETCONN=8
RT_LWIP_PBUF_NUM=16
RT_LWIP_RAW_PCB_NUM=4
RT_LWIP_UDP_PCB_NUM=4
RT_LWIP_TCP_PCB_NUM=4
RT_LWIP_TCP_SEG_NUM=40
RT_LWIP_TCP_SND_BUF=8196
RT_LWIP_TCP_WND=8196
RT_LWIP_TCPTHREAD_PRIORITY=10
RT_LWIP_TCPTHREAD_MBOX_SIZE=8
RT_LWIP_TCPTHREAD_STACKSIZE=1024
RT_LWIP_ETHTHREAD_PRIORITY=12
RT_LWIP_ETHTHREAD_STACKSIZE=1024
RT_LWIP_ETHTHREAD_MBOX_SIZE=8
LWIP_NETIF_STATUS_CALLBACK=1
LWIP_NETIF_LINK_CALLBACK=1
```

```

SO_REUSE=1
LWIP_SO_RCVTIMEO=1
LWIP_SO_SNDTIMEO=1
LWIP_SO_RCVBUF=1
LWIP_SO_LINGER=0
LWIP_NETIF_LOOPBACK=0
RT_LWIP_USING_PING=y

```

开启ping后，验证前首先确认网线已经连接，再通过“ping”命令进行操作，参考如下：

```

## 第8章 网络连接成功log信息
[ (1)3.357.573] e0: 100M
[ (1)3.357.592] e0: full duplex
[ (1)3.357.610] e0: flow control off
[ (1)3.357.811] e0: link up.

## 第8章 向网关发送ping包和执行结果
msh >ping 192.168.31.1
[ (1)52.351.270] 60 bytes from 192.168.31.1 icmp_seq=0 ttl=64 time=0 ms
[ (1)53.355.786] 60 bytes from 192.168.31.1 icmp_seq=1 ttl=64 time=0 ms
[ (1)54.361.215] 60 bytes from 192.168.31.1 icmp_seq=2 ttl=64 time=0 ms
[ (1)55.366.645] 60 bytes from 192.168.31.1 icmp_seq=3 ttl=64 time=0 ms

```

8.4.3 Kernel

Kernel GMAC 详细使用方法可以参考

《Rockchip_Developer_Guide_Linux_GMAC_Mode_Configuration_CN.pdf》

如使用 Linux + RT_Thread 的方式，并且在RT_Thread 下使用 GMAC，Kernel 需要在 DTS 中关闭对应的 GMAC，同时声明 GMAC 的时钟。

以 RK3562 为例：

```

/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";
        clocks = <&cru FCLK_BUS_CM0_CORE>, <&cru CLK_BUS_CM0_RTC>,
        # GMAC 时钟声明
        <&cru PCLK_GMAC>, <&cru ACLK_GMAC>, <&cru CLK_GMAC_125M_CRU_I>,
        <&cru CLK_GMAC_50M_CRU_I>, <&cru CLK_GMAC_ETH_OUT2IO>,
        <&cru SCLK_UART7>, <&cru PCLK_UART7>, <&cru PCLK_TIMER>,
        <&cru CLK_TIMER4>, <&cru CLK_TIMER5>;

        pinctrl-names = "default";
        pinctrl-0 = <&uart7m1_xfer>;

        amp-cpu-aff-maskbits = /bits/ 64 <0x0 0x1 0x1 0x2 0x2 0x4 0x3 0x8>;
        amp-irqs = /bits/ 64 <GIC AMP_IRQ_CFG_ROUTE(147, 0xd0,
CPU_GET_AFFINITY(3, 0))
                                # GMAC 中断配置
                                GIC AMP_IRQ_CFG_ROUTE(105, 0xd0, CPU_GET_AFFINITY(3,
0))>;

        status = "okay";
    };
};

```

```
&gmac0 {  
    status = "disabled";  
};
```

8.5 PCIE

Bare-metal 或 RT_Thread 下仅支持以下简单功能：

- 控制器寄存器访问
- CPU 访问外设，主要包括 Bar、CFG 空间
- uDMA 传输
- INTx legacy 中断

8.5.1 HAL / RT-Thread

测试代码的路径为 `hal/test/hal/test_pcie.c`

具体介绍可以参考 `hal/doc/guides/Rockchip_User_Guide_HAL_PCIE_CN.md` 文档。

8.6 CPU Cache ECC

RK3568平台上支持Cache ECC功能，支持单bit错误可检测纠正，双bit错误可检测不可纠正可记录。并支持手动注入错误用于功能的验证。

8.6.1 RT-Thread

Cache ECC 操作需要在安全环境下进行，需要客户进行评估。具体的操作及配置请参考文档《Rockchip_Developer_Guide_RT-Thread_CacheECC_CN.pdf》。

8.7 DDR ECC

RK3568平台上支持DDR ECC（Sideband ECC），可以对DDR数据进行错误检查和纠正，支持SEC/DED ECC。支持单bit错误可检测纠正，双bit错误可检测不可纠正可记录。并可以在指定的内存区域中手动注入错误用于功能的验证。

8.7.1 HAL

HAL中具体的操作及配置请参考文档《Rockchip_Developer_Guide_HAL_DDR_ECC_CN.pdf》；

8.7.2 Kernel

Linux中具体的操作及配置请参考文档《Rockchip-Developer-Guide-DDR-CN.pdf》。

9. 第9章 调试

9.1 串口调试

瑞芯微多核异构系统中默认调试串口配置如下：

波特率	数据位	停止位	奇偶校验	流控
1500000	8	1	none	none

相关章节：

[第8章 编译配置](#)

9.1.1 U-Boot 启动输出

以 RK3562 为例，瑞芯微多核异构系统启动时，CPU3 固件 Ram 加载地址为 0x01800000：

```
AMP: Brought up cpu[3] with state 0x10, entry 0x01800000 ...OK
```

9.1.2 RK HAL 启动输出

```
*****
Hello RK3562 Bare-metal using RK_HAL!
Rockchip Electronics Co.Ltd
CPI_ID(3)
*****
[(3)0.671.983] CPU(3) Initial OK!
```

9.1.3 RT-Thread 启动输出

```
\ | /
- RT -      Thread Operating System
/ | \      4.1.1 build Apr 12 2024 20:28:35
2006 - 2022 Copyright by RT-Thread team
```

9.2 AP 使用 OpenOCD 调试

AP 支持使用 OpenOCD 调试。调试使用的硬件工具为瑞芯微设计的 JTAG 小板。支持单步运行、断点追踪、data watch 和 寄存器、memory dump 等常用功能，详细资料请参考 [JTAG & SWD连接开发和调试](#)

9.2.1 Windows 环境搭建

9.2.1.1 软件安装

1. 安装 OpenOCD 开发环境

OpenOCD 开发包为 `openocd_eclipse-2020-09.zip`，将该压缩文件解压至指定目录。

RK压缩包目录：

```
.
├── eclipse-workspace      # 工作目录，eclipse 已默认把工作目录设到该文件夹
├── OpenOCD                # OpenOCD 相关文件
│   ├── bat               # Windows 批处理文件，双击可以直接连接芯片
│   ├── bin               # 存放 openocd.exe 和 *gdb.exe
│   ├── doc               # 驱动安装文档和使用文档
│   └── tcl               # 脚本文件
└── SVD                   # 主要用来查看芯片寄存器
```

2. 安装运行 eclipse 需要的 JRE 工具包

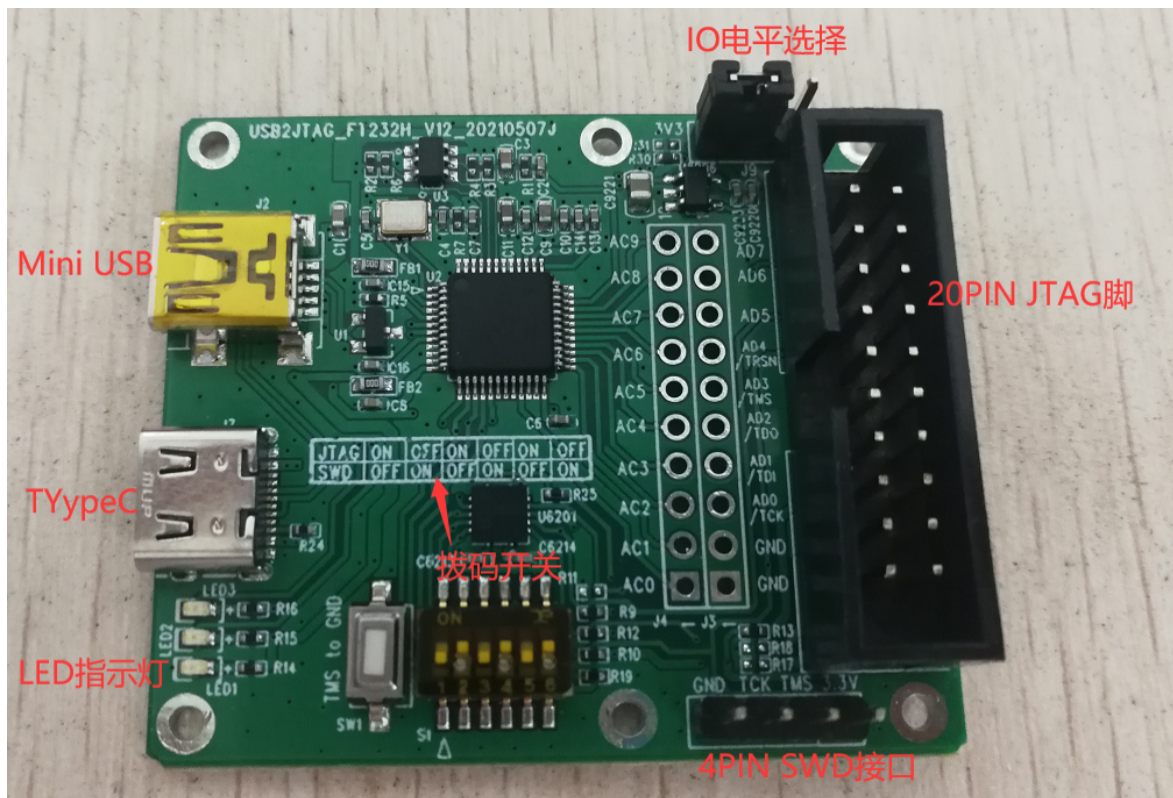
JRE 工具包为下载资料包目录下的 `/环境搭建软件/jdk_8.0.1310.11_64.exe`。具体安装及配置步骤参见资料包下的《Rockchip_Developer_Guide_GNU_MCU_Eclipse_OpenOCD_CN.pdf》文档说明。

3. 安装 JTAG 驱动

JTAG 驱动为下载资料包目录下的 `/环境搭建软件/zadig-2.7.exe`。具体安装及配置步骤参见资料包下的《Rockchip_Developer_Guide_FT232H_USB2JTAG.pdf》文档说明。

9.2.1.2 硬件连接

FT232H 是“Future Technology Devices International Ltd”的一款芯片，它可以通过 USB 接口与计算机通信，提供 JTAG 和 SWD 的扩展能力。



FT232H 小板如上图所示：

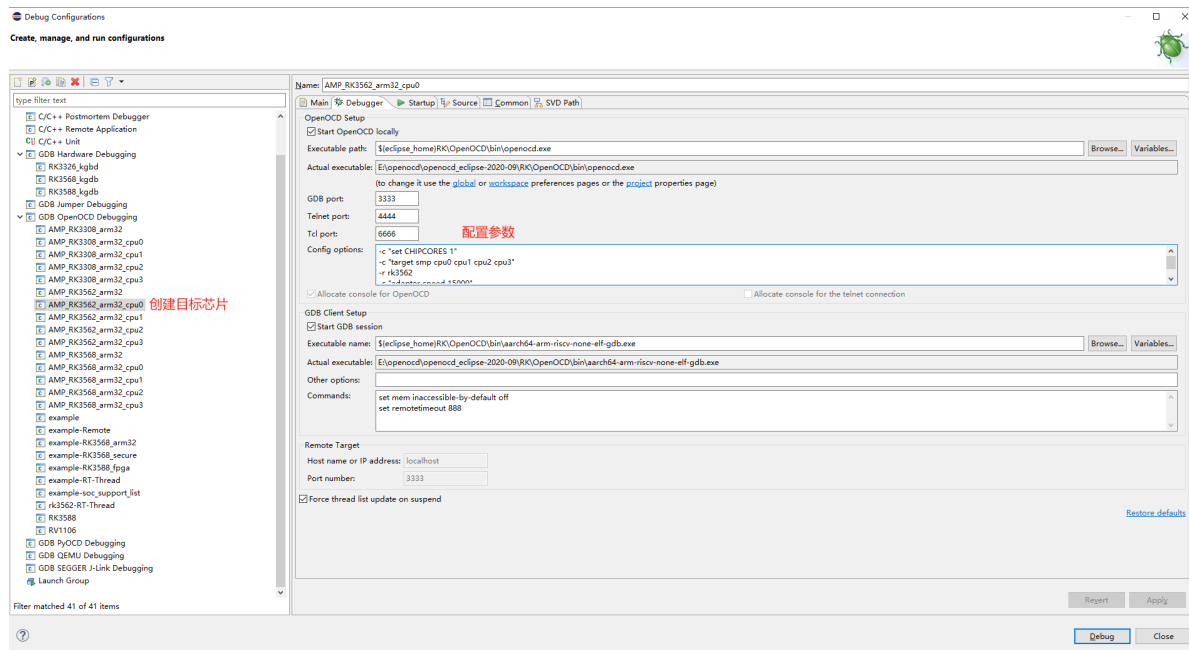
- LED 指示灯，LED1：电源指示灯；LED2：灭：未连接，闪：连接；LED3：暂时未定义
 - ARM 20PIN JTAG 接口
 - USB 接口：有 TYPEC 接口和 mini USB 接口两种
 - 拨码开关
- SWD 模式， 1、3、5 off，2、4、6 on
- JTAG 模式， 1、3、5 on，2、4、6 off
- 排针，VCC、TCS、TCK、GND，可以和板子飞线连接
 - 排针，3.3V、VCCIO、1.8V，可以用跳冒连接 VCCIO 到 3.3V 或 1.8V，这个一定要接，不然 JTAG 通讯会失败

9.2.2 使用示例

以 RK3562 为列，搭建 OpenOCD 开发环境：

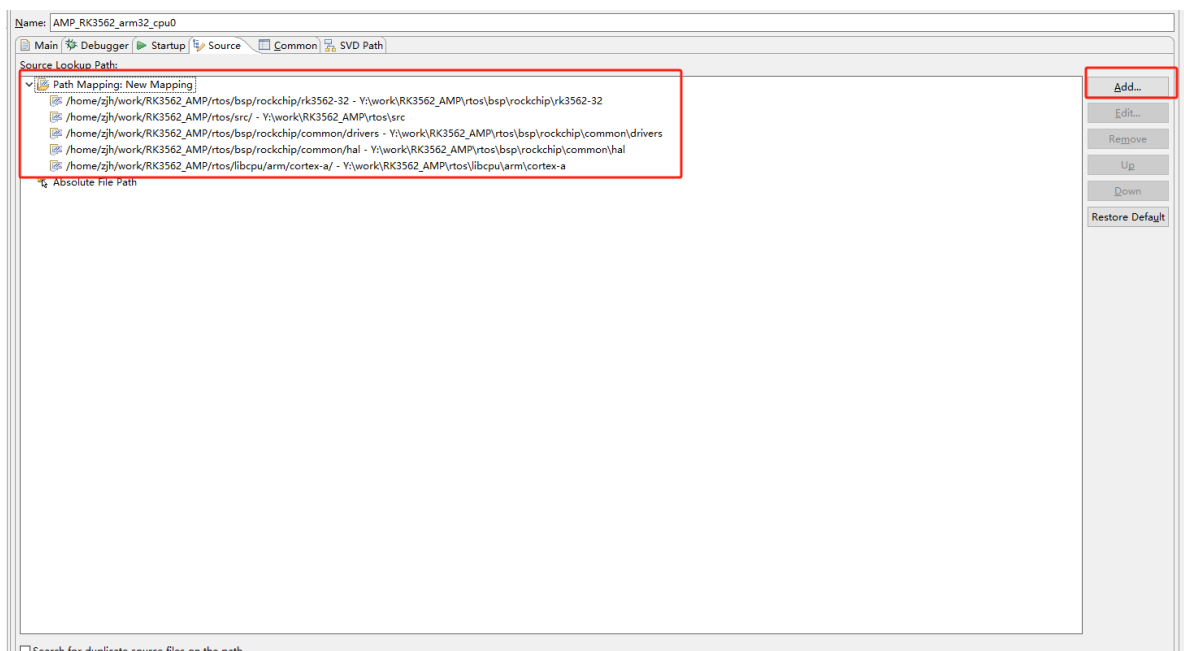
1. 参照《Rockchip_Developer_Guide_GNU_MCU_Eclipse_OpenOCD_CN.pdf》文档，创建目标芯片的配置项。
2. 运行 eclipse.exe 进入 “Debug Configurations” 配置项，打开 “Debugger” 标签页，在 “Config options” 项目中加入：

```
-c "set SMPMASK 0x8"           # 0x8表示cpu3，配置cpu3运行RT_Thread
-r rk3562                      # 指定芯片
-c "cpu3 configure -rtos RT_Thread" # 指定 cpu3 运行 RT_Thread
-c "adapter speed 15000"        # JTAG TCK 速率，单位KHz
```



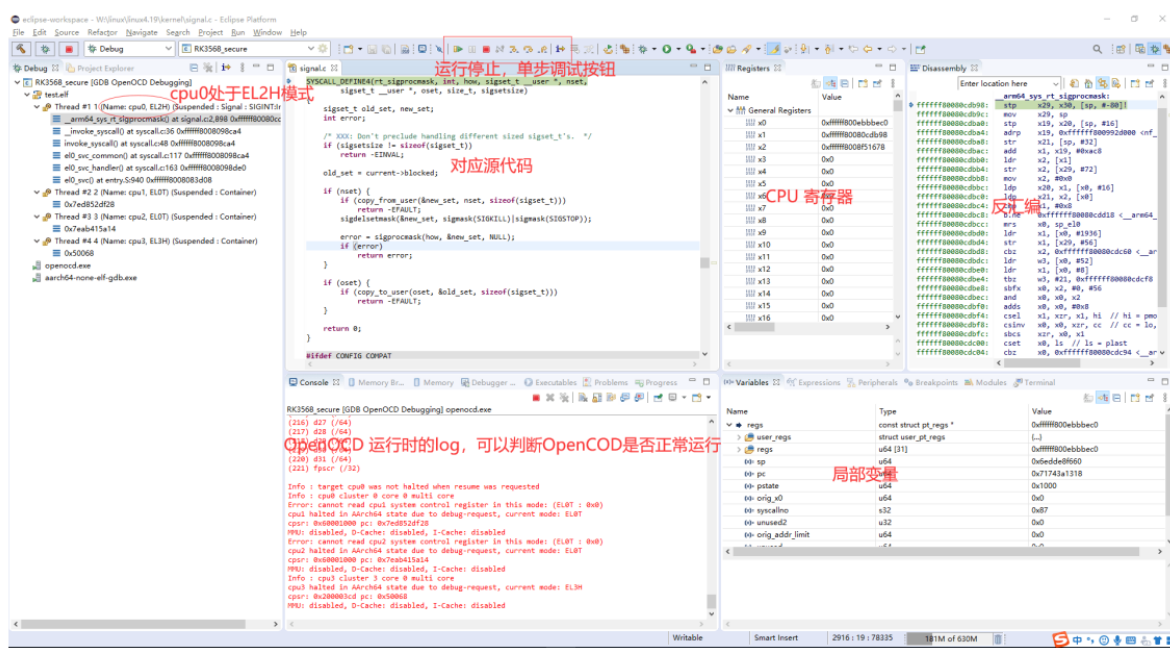
3. 进入 “Debug Configurations” 配置项，打开 Source 标签页，编辑 “Path Mapping: New Mapping” 项目，加入或修改 RK356x AMP SDK 的工程路径：

```
<AMP_SDK>/hal/                # GCC 编译时的工程路径
D:<AMP_SDK>\hal                # Windows 下用于 Debug 追踪的源代码的工程路径
```



以上两个路径实际为同一路径，<AMP_SDK>/rk3562/hal/ 为解析符号表时需要的路径信息。D:\<AMP_SDK>\hal\ 为 Windows 下加载工程源代码的路径。同时需要保证下载到开发板的固件，与调试代码一致。

4. 以上配置完成后，通过“Debug Configurations”下的“Debug”按钮开始进行调试。此时会在“Debugger Console”窗口显示调试信息：



在 Console 窗口下通过以下命令分别加入 4 个 CPU 的 *.elf 文件：

```
# .....
For help, type "help".
Type "apropos word" to search for commands related to "word".
# .....
add-symbol-file D:/rk3562/hal/project/rk3562/GCC/0_TestDemo.elf
add-symbol-file D:/rk3562/hal/project/rk3562/GCC/1_TestDemo.elf
add-symbol-file D:/rk3562/hal/project/rk3562/GCC/2_TestDemo.elf
add-symbol-file D:/rk3562/hal/project/rk3562/GCC/3_TestDemo.elf
```

9.3 MCU 使用 Ozone 调试

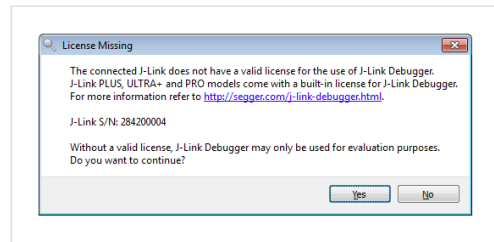
9.3.1 Windows 环境搭建

Ozone 工具是一款常用的，且带有便捷图像界面的嵌入式调试工具，借助J-Link硬件，可以实现对代码的实时跟踪，分步运行以及多断点触发等功能。官方地址：[Ozone – The Performance Analyzer \(segger.com\)](http://www.segger.com/Ozone-The-Performance-Analyzer)。官方提供商业使用许可和非商业使用许可两种模式，请用户根据实际需求选择合适的许可模式，确保合法使用。

Licensing

Commercial use

Ozone can be used in a commercial environment as part of the licence for **J-Link PLUS, ULTRA+, PRO** and **J-Trace**. With **J-Link BASE**, Ozone can be used commercially after purchasing the **J-Link BASE to PLUS upgrade** bundle, that includes the Ozone license. With other J-Link models, Ozone remains in evaluation mode and presents the following screen each time a debug session is started:



Free for non-commercial use

Like most of our software, Ozone can easily be downloaded and installed without any registration process. For non-commercial use or evaluation, Ozone is available to be used free of charge. With a J-Link PLUS, PRO, ULTRA+, or with a J-Trace, Ozone is available for free.



以 RK3562 为例：

1. 连接好 J-Link 设备和调试板子，打开 Ozone 软件，默认跳出工程配置选项，或者点击 `File->New->New Project Wizard`

 New Project Wizard ✕

Target Device
Choose a Target Device

Device

Cortex-M0 ...

Register Set

Cortex-M0 ▼ ...

Peripherals (optional)

...

< Back Next > Cancel

 New Project Wizard ✕

Connection Settings
Choose a Target and Host Interface

Target Interface
SWD

Target Interface Speed
4 MHz

Host Interface
USB

Serial No (optional)

Emulators connected via USB

Product	Nickname	Serial No

< Back

Next >

Cancel

在红框位置中，选中连接的 J-Link 设备。



New Project Wizard



Program File

Choose the Program to be debugged

ELF, Motorola S-record, Intel Hex, or Binary file (optional)

Z:/work/hal-amp/project/rk3562-mcu/GCC/TestDemo.elf

...

< Back

Next >

Cancel

New Project Wizard

Optional Settings
Set optional project settings, such as the initial PC

Initial PC (after download and reset)

☐ ELF Entry Point

☐ Read from Base Address Vector Table

☐ Read from Location

☐ Location

☒ Do not set

Initial Stack Pointer

☐ Read from Base Address Vector Table

☐ Read from Location

☐ Location

☒ Do not set

J-Link Script File

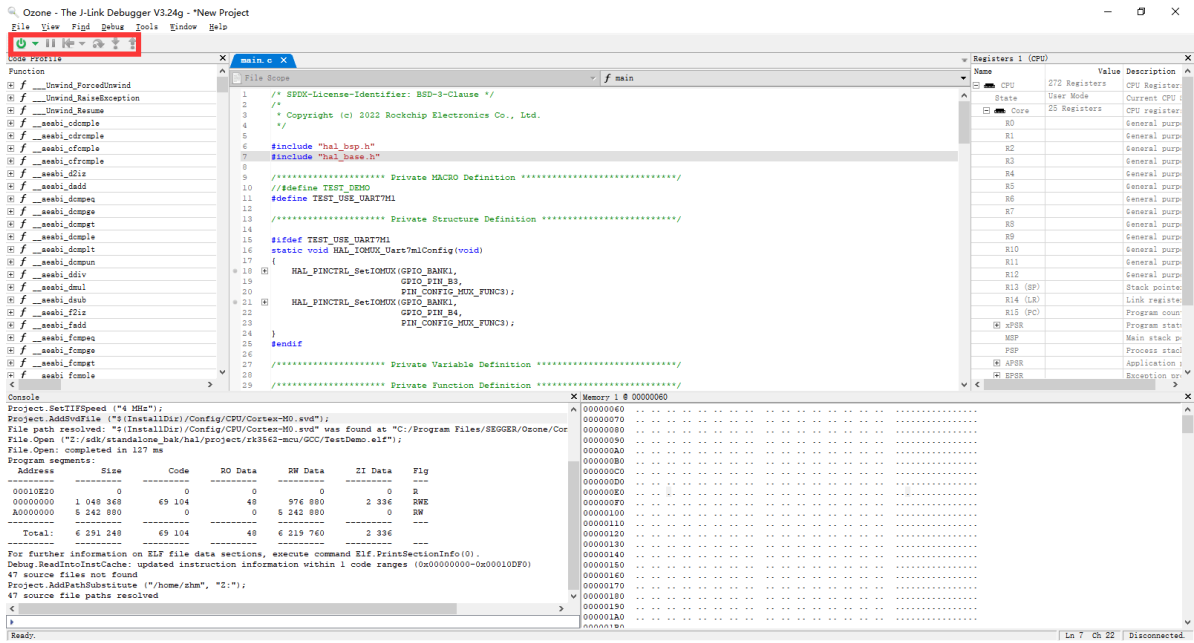
J-Link Log File

< Back Finish Cancel

2. 加载目标文件：使用 Ozone 的加载功能将目标文件（通常是生成的可执行文件）加载到调试器中。如果 RK HAL 代码仓库在 Linux 环境下，Ozone 调试工具安装在 Windows 环境下，需要先在 Windows 系统中对 Linux 路径做网络磁盘映射，例如将 Linux 系统中的 `"/home/xxx"` 映射到 Windows 系统中的 `"Z:"`。再在 Ozone 软件界面左下角命令行使用以下命令进行工程路径映射，其中参数 `"/home/xxx"` 为 Linux 环境挂载到 Windows 上使用的 Linux 路径，参数 `"Z:"` 则为对应的 Windows 路径。

```
Project.AddPathSubstitute "/home/xxx" "Z:"
```

完成这一系列操作后能得到如下 Ozone 界面。



红框位置是 Debug 开关，可以实现分步调试功能，也能在代码窗上直接加断点进行调试。

10. 第10章 演示

10.1 性能测试

10.1.1 测试整型

使用 Coremark 测试整型性能。Coremark 是一个专门用于测试处理器整型性能的基准测试。运行 Coremark 会产生一个单一数字分数，使用户能够快速比较不同的处理器。用于衡量嵌入式系统中微控制器（MCU）和中央处理器（CPU）的性能。下表为各平台 Coremark 测试的数据汇总：

处理器	RK3568 AP HAL	RK3562 AP HAL	RK3562 MCU	RK3576 MCU
主频	816MHZ	816MHZ		
运行方式	DDR4 1560MHZ	DDR4 1332MHZ		
Cache	开	开		
TCM	无	无		
Coremark	3273	2387		
Coremark /MHz	4.0	2.92		

裸机端测试方法如下：

使能测试代码，打开如下几个宏开关。

代码路径：hal/project/rkxxx/src/main.c

```
-//#define TEST_DEMO
+#define TEST_DEMO
```

代码路径：hal/project/rkxxx/src/test_demo.c

```
-//#define PERF_TEST
+#define PERF_TEST
```

代码路径：/hal/middleware/benchmark/benchmark

```
INCLUDES += \
-I"$ (BENCHMARK_PATH) " \
-I"$ (BENCHMARK_PATH)/coremark" \
-I"$ (BENCHMARK_PATH)/coremark/barebones" \

SRC_DIRS += \
$(BENCHMARK_PATH) \
$(BENCHMARK_PATH)/coremark \
$(BENCHMARK_PATH)/coremark/barebones \
```

代码路径：/hal/middleware/benchmark/benchmark.h

```
#define HAL_BENCHMARK_COREMARK
//#define HAL_BENCHMARK_LINPACK
//#define HAL_BENCHMARK_TINYMEMBENCH
```

10.1.2 测试浮点型

使用 Linpack 测试浮点型性能。Linpack 测试的主要指标是每秒浮点操作次数（FLOPS），即系统每秒能够执行的浮点运算次数。通常以 MFLOPS（百万次每秒浮点操作）为单位进行测量。下表为各平台 Linpack 测试的数据汇总：

处理器	RK3568 AP HAL	RK3562 AP HAL	RK3562 MCU	RK3576 MCU
主频	816MHZ	816MHZ		
运行方式	DDR4 1560MHZ	DDR4 1332MHZ		
Cache	开	开		
TCM	无	无		
Linpack MFLOPS	154.7	79.38		

裸机端测试方法如下：

使能测试代码，打开如下几个宏开关。

代码路径：hal/project/rkxxx/src/main.c

```
-// #define TEST_DEMO
+ #define TEST_DEMO
```

代码路径：hal/project/rkxxx/src/test_demo.c

```
-// #define PERF_TEST
+ #define PERF_TEST
```

代码路径：/hal/middleware/benchmark/benchmark.mk

```
INCLUDES += \
-I"$ (BENCHMARK_PATH) " \
-I"$ (BENCHMARK_PATH)/linpack" \

SRC_DIRS += \
$(BENCHMARK_PATH) \
$(BENCHMARK_PATH)/linpack \
```

代码路径: /hal/middleware/benchmark/benchmark.h

```
// #define HAL_BENCHMARK_COREMARK
#define HAL_BENCHMARK_LINPACK
// #define HAL_BENCHMARK_TINYMEMBENCH
```

10.1.3 测试内存

RTOS / Bare-metal 使用 tinymembench 测试内存性能。tinymembench 是一个简单的内存基准测试工具，用于评估计算机系统的内存性能。它测试了内存带宽、延迟和随机访问性能等关键指标。下表为各平台 tinymembench 测试的数据汇总：

处理器	RK3568 AP HAL	RK3562 AP HAL	RK3562 MCU	RK3576 MCU
主频	816MHZ	816MHZ		
运行方式	DDR4 1560MHZ	DDR4 1332MHZ		
Cache	开	开		
TCM	无	无		
内存带宽测试	参考下方详细数据	无		
内存延迟测试	参考下方详细数据	无		

RK3568 AP HAL 数据

```
=====
== Memory bandwidth tests ==
==
== Note 1: 1MB = 1000000 bytes ==
== Note 2: Results for 'copy' tests show how many bytes can be ==
==         copied per second (adding together read and writen ==
==         bytes would have provided twice higher numbers) ==
== Note 3: 2-pass copy means that we are using a small temporary buffer ==
==         to first fetch data into it, and only then write it to the ==
==         destination (source -> L1 cache, L1 cache -> destination) ==
== Note 4: If sample standard deviation exceeds 0.1%, it is shown in ==
==         brackets ==
=====

C copy backwards                : 1673.8 MB/s
C copy backwards (32 byte blocks) : 1687.0 MB/s
C copy backwards (64 byte blocks) : 1673.8 MB/s
```

```

C copy : 1920.2 MB/s
C copy prefetched (32 bytes step) : 1563.7 MB/s
C copy prefetched (64 bytes step) : 1941.1 MB/s (0.1%)
C 2-pass copy : 995.7 MB/s
C 2-pass copy prefetched (32 bytes step) : 1036.3 MB/s
C 2-pass copy prefetched (64 bytes step) : 1007.0 MB/s
C fill : 3297.4 MB/s
C fill (shuffle within 16 byte blocks) : 3297.4 MB/s
C fill (shuffle within 32 byte blocks) : 3297.4 MB/s
C fill (shuffle within 64 byte blocks) : 3292.3 MB/s
---
standard memcpy : 1165.1 MB/s
standard memset : 3322.9 MB/s
---
ARM fill (STM with 8 registers) : 3343.7 MB/s
ARM fill (STM with 4 registers) : 3322.9 MB/s

```

```

=====
== Memory latency test ==
==
== Average time is measured for random memory accesses in the buffers ==
== of different sizes. The larger is the buffer, the more significant ==
== are relative contributions of TLB, L1/L2 cache misses and SDRAM ==
== accesses. For extremely large buffer sizes we are expecting to see ==
== page table walk with several requests to SDRAM for almost every ==
== memory access (though 64MiB is not nearly large enough to experience ==
== this effect to its fullest). ==
==
== Note 1: All the numbers are representing extra time, which needs to ==
== be added to L1 cache latency. The cycle timings for L1 cache ==
== latency can be usually found in the processor documentation. ==
== Note 2: Dual random read means that we are simultaneously performing ==
== two independent memory accesses at a time. In the case if ==
== the memory subsystem can't handle multiple outstanding ==
== requests, dual random read has the same timings as two ==
== single reads performed one after another. ==
=====

block size : single random read / dual random read
1024 : 0.0 ns / 0.0 ns
2048 : 0.0 ns / 0.0 ns
4096 : 0.0 ns / 0.0 ns
8192 : 0.0 ns / 0.0 ns
16384 : 0.0 ns / 0.0 ns
32768 : 11.2 ns / 0.3 ns
65536 : 22.4 ns / 32.7 ns
131072 : 33.3 ns / 43.9 ns
262144 : 39.4 ns / 47.2 ns
524288 : 48.8 ns / 56.1 ns
1048576 : 176.3 ns / 253.8 ns
2097152 : 240.5 ns / 317.2 ns
4194304 : 272.0 ns / 339.1 ns
8388608 : 285.9 ns / 328.5 ns

```

裸机端测试方法如下：

使能测试代码，打开如下几个宏开关。

代码路径：hal/project/rkxxx/src/main.c

```
-//#define TEST_DEMO
+#define TEST_DEMO
```

代码路径：hal/project/rkxxx/src/test_demo.c

```
-//#define PERF_TEST
+#define PERF_TEST
```

代码路径：/hal/middleware/benchmark/benchmark.mk

```
INCLUDES += \
-I"$ (BENCHMARK_PATH) " \
-I"$ (BENCHMARK_PATH) /tinymembench" \

SRC_DIRS += \
$(BENCHMARK_PATH) \
$(BENCHMARK_PATH) /tinymembench \
```

代码路径：/hal/middleware/benchmark/benchmark.h

```
//#define HAL_BENCHMARK_COREMARK
//#define HAL_BENCHMARK_LINPACK
#define HAL_BENCHMARK_TINYMEMBENCH
```

10.1.4 测试中断响应时间

使用 HAL 中自带的中断延迟测试 demo 进行测试。中断响应延迟是指从中断事件发生到系统开始处理中断的时间间隔，用于评估计算机系统中断处理性能的测试方法。下表为各平台中断响应延迟测试的数据汇总：

处理器	RK3568 AP HAL	RK3562 AP HAL	RK3562 MCU	RK3576 MCU
主频	816MHZ	816MHZ		
运行方式	DDR4 1560MHZ	DDR4 1332MHZ		
Cache	开	开		
TCM	无	无		
Irq Latency Test	avg = 3.42 us max = 4.13 us min = 3.21 us	avg = 1.543296 us max = 2.916667 us min = 0.875000 us		

裸机端测试方法如下：

裸机系统测试中断延时响应数据需要打开以下宏开关。

hal/project/rkxxx/src/main.c

```
-//#define TEST_DEMO
+#define TEST_DEMO
```

hal/project/rkxxx/src/test_demo.c

```
-//#define IRQ_LATENCY_TEST
+#define IRQ_LATENCY_TEST
```

10.2 实时性演示

以 RK3568 平台为例，展示 AMP 方案下 Linux 操作系统及 RTOS 的实时性能。通过这个演示，便于了解 AMP 方案在实时数据处理方面的能力以及相关的特性和工具。

10.2.1 测试方法

Linux 系统使用cyclicttest 测试评估系统的响应时间和延迟。可以从Linux发行版的软件仓库或cyclicttest的官方网站下载并安装cyclicttest工具。

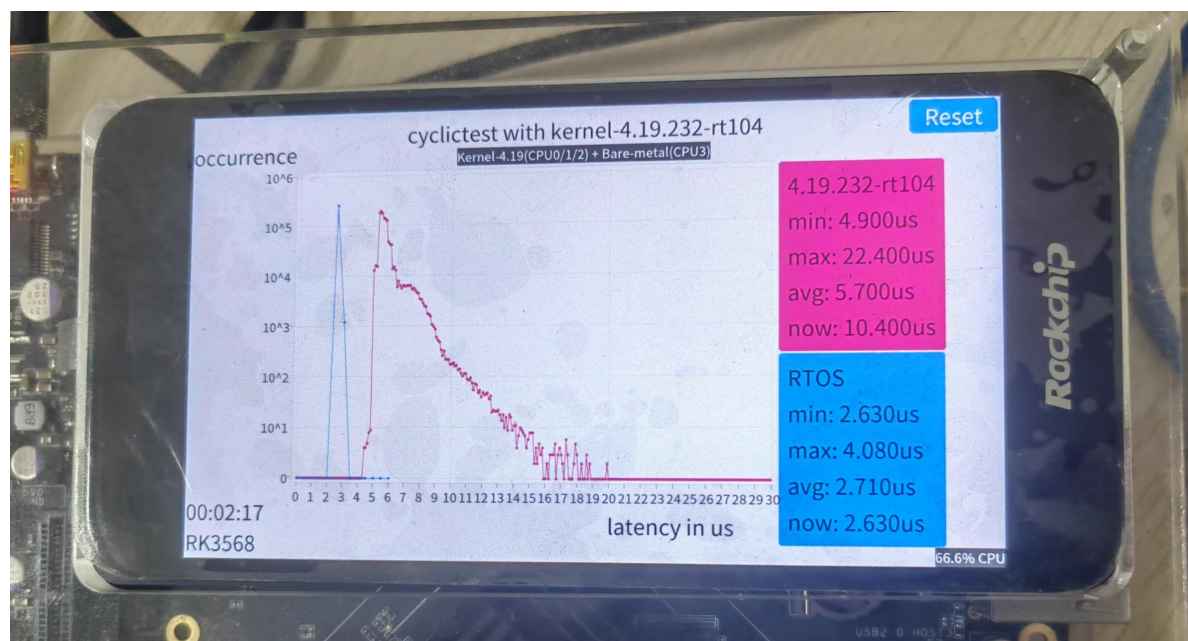
RTOS 使用中断延迟测试评估系统的响应时间和延迟。示例代码路径：hal/project/rk3568/src/test_demo.c

10.2.2 测试原理

测试原理为创建一个或多个实时线程，这些线程以固定的循环时间运行。每个线程在每个循环中都会记录时间，然后计算出实际的循环时间和偏差。这样可以测量系统的响应时间和延迟。

10.2.3 测试结果

测试将输出测试结果，包括每个循环的当前延迟时间和最大延迟时间等。测试结果如下图：



测试的输出包含了一些关键指标，如以下几个例子：

- **Min:** 测量的最小循环时间。
- **Avg:** 测量的平均循环时间。
- **Max:** 测量的最大循环时间。
- **Now:** 记录当前时间值。
- **Occurrence:** 记录延迟耗时区间发生的次数
- **Latency in us:** 延迟耗时区间

这些指标可用于评估系统的实时性能。最大的延迟时间越小表示系统具有较好的实时性能。

11. 第11章 附录

11.1 术语

缩写	全称	定义
OpenAMP	Open Asymmetric Multi-Processing	开源非对称多处理系统
AMP	Asymmetric Multi-Processing	非对称多处理系统
HAL	Hardware abstraction layer	硬件抽象层
Bare-metal	Bare-metal	提供基于硬件抽象层的裸机开发库
MailBox	MailBox	一个简单的 APB 外设，允许 CPU、MCU 核心通过写操作产生中断来相互通信
RPMsg	Remote Processor Messaging	一种用于多核处理器之间通信的协议
RTOS	Real-time operating system	实时操作系统
RTT	RT-Thread	一款主要由中国开源社区主导开发的开源实时操作系统
SDK	Software Development Kit	软件开发工具包
Linux	Linux	一种自由和开放源代码的类UNIX操作系统
Kernel	Linux Kernel	Linux 内核，是 Linux 操作系统的核心部分，它负责管理计算机的硬件资源，并提供基本的系统服务。
Hypervisor	Virtual Machine Monitor	虚拟机监视器用于创建和运行虚拟机的软件、固件或硬件，允许它们共享物理硬件资源
Jailhouse	Jailhouse	一个针对创建工业级应用程序的小型虚拟机监视器

11.2 文档索引

引用文档	说明	文档路径
Rockchip_Developer_Guide_FT232H_USB2JTAG.pdf	FHT232 小板介绍	openocd_eclipse-2020-09\RK\OpenOCD\doc
Rockchip_Developer_Guide_GNU_MCU_Eclipse_OpenOCD_CN.pdf	OpenOCD 使用说明	openocd_eclipse-2020-09\RK\OpenOCD\doc
Rockchip_Developer_Guide_UBoot_Nextdev_CN.pdf	U-boot 开发文档	docs\cn\Common\UBOOT
Rockchip_Developer_Guide_Linux_AB_System_CN.pdf	AB 双分区说明	docs\cn\Common\UBOOT
Rockchip_Developer_Guide_SDMMC_SDIO_eMMC_CN.pdf	eMMC 使用说明	docs\cn\Common\MMC
Rockchip_Developer_Guide_UART_CN.pdf	UART 使用说明	docs\cn\Common\UART
Rockchip_Developer_Guide_RT-Thread_SPIFLASH_CN.pdf	SPI FLASH 使用说明	docs\cn\Common\NVM