

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 资源分配
 - 4.1.1 内存资源
 - 4.1.1.1 SRAM内存分区配置

- 4.1.1.2 SDRAM 内存分区配置
 - 4.1.1.3 共享内存分区配置
 - 4.1.2 Cache和MMU配置
 - 4.1.2.1 Kernel
 - 4.1.2.2 HAL
 - 4.1.2.3 RT-Thread
 - 4.1.3 中断资源
 - 4.1.4 外设资源
- 4.2 资源保护
 - 4.2.1 时钟资源
 - 4.2.2 引脚资源
 - 4.2.3 电源域资源
- 5. 第5章 启动方案
 - 5.1 AP启动方案
 - 5.1.1 U-Boot 阶段
 - 5.1.2 Linux + HAL
 - 5.1.3 Linux + RTOS
 - 5.1.4 RTOS + HAL
 - 5.1.5 4 * RTOS
 - 5.1.6 4 * HAL
 - 5.2 MCU启动方案
 - 5.2.1 U-Boot阶段启动
 - 5.2.2 MCU启动阶段
 - 5.3 AMP固件打包
- 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 Linux Kernel适配RPMsg
 - 6.3.3.1 代码结构
 - 6.3.3.2 RTOS端适配RPMsg-Lite
 - 6.4 RPMsg测试示例
 - 6.4.1 RK3308
 - 6.4.1.1 kernel+rtt
 - 6.4.1.1.1 共享内存
 - 6.4.1.1.2 测试demo
 - 6.4.1.2 RTT+HAL
 - 6.4.1.2.1 共享内存
 - 6.4.1.2.2 测试demo
- 7. 第7章 中断
 - 7.1 ARM GIC v2
 - 7.1.1 HAL 中断使用示例
 - 7.1.2 HAL GIC中断配置表
 - 7.1.3 HAL GPIO中断示例
 - 7.1.4 HAL TIMER中断示例
 - 7.1.5 HAL 软中断使用方法
 - 7.1.6 HAL 中断用例小结
 - 7.2 ARM GIC v3
 - 7.3 ARM NVIC
 - 7.4 RISC-V中断控制器
- 8. 第8章 模块
 - 8.1 UART
 - 8.1.1 RK3308

- 8.1.1.1 U-Boot
 - 8.1.1.2 Kernel
 - 8.1.1.3 RT-Thread
 - 8.1.1.4 HAL
 - 8.2 EMMC
 - 8.2.1 RK3308
 - 8.2.1.1 Kernel
 - 8.2.1.2 RT-Thread
 - 8.2.1.2.1 RT-Thread配置
 - 8.2.1.2.2 Flash分区表配置
 - 8.2.1.2.3 执行结果验证
 - 8.3 SNOR
 - 8.3.1 RK3308
 - 8.3.1.1 RT-Thread
 - 8.3.1.1.1 RT-Thread配置
 - 8.3.1.1.2 Flash分区表配置
 - 8.3.1.1.3 执行结果验证
 - 8.4 GMAC
 - 8.4.1 RK3308
 - 8.4.1.1 RT-Thread
 - 8.4.1.1.1 RT-Thread配置
 - 8.4.1.1.2 执行结果验证
 - 8.5 PCIE
- 9. 第10章 调试
 - 9.1 串口调试
 - 9.2 OpenOCD调试
 - 9.2.1 环境搭建说明
 - 9.3 Ozone调试
- 10. 第11章 演示
 - 10.1 RK3568
 - 10.2 RK3308
 - 10.3 RK3358
 - 10.4 RK3562
- 11. 第12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。

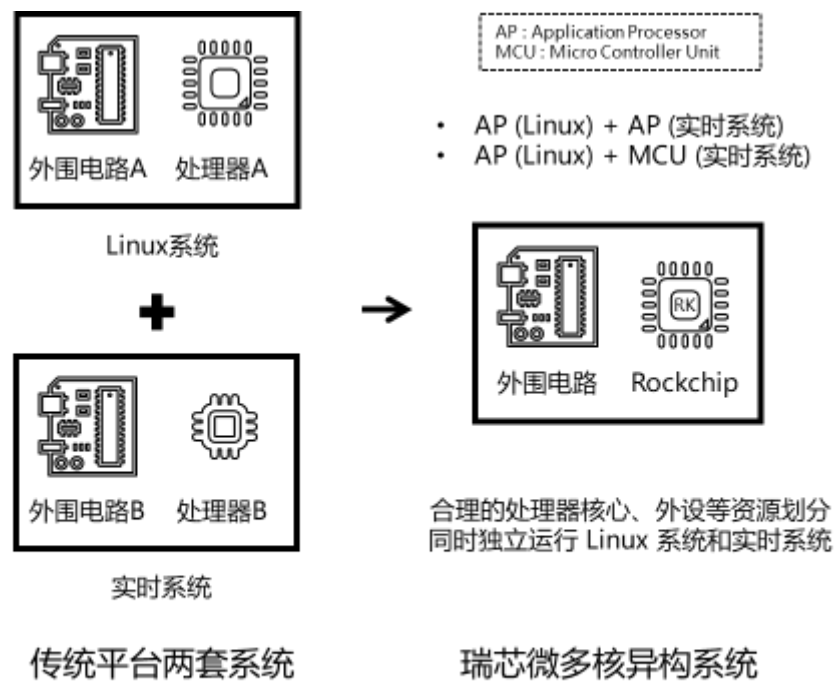


图 1-1-1 多核异构系统将传统平台两套系统合二为一

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。

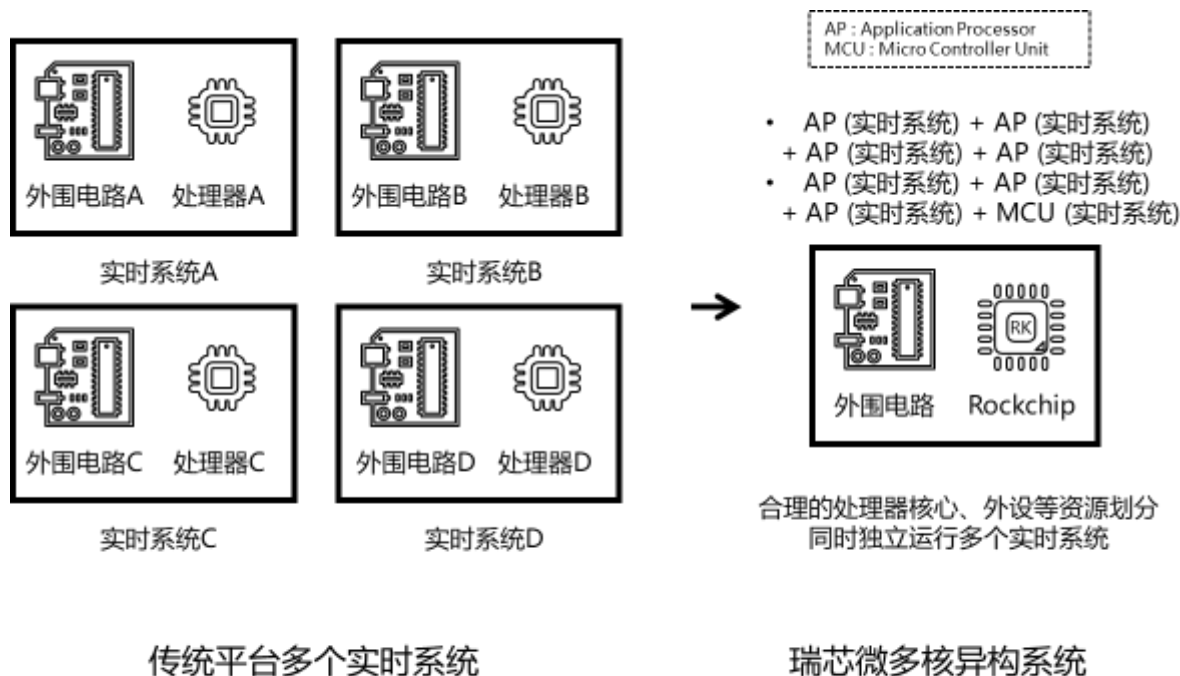


图 1-1-2 同时独立运行多个实时系统

多核异构系统还支持同一颗 SoC 芯片以 SMP + AMP 的方式运行。

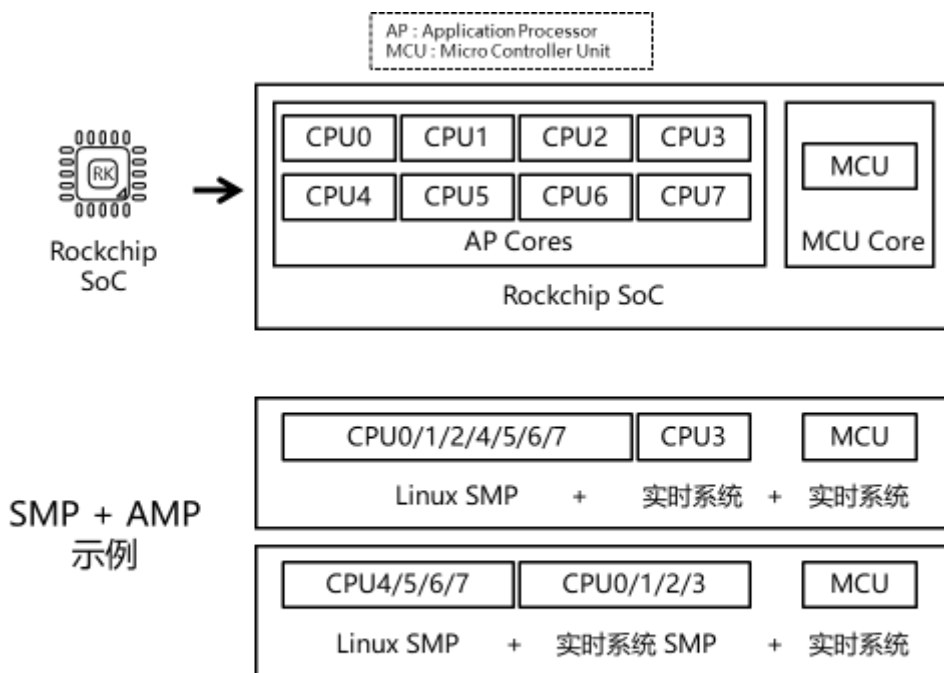


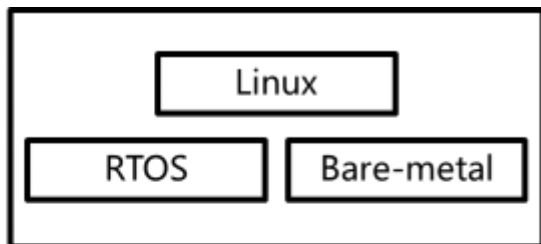
图 1-1-3 以 SMP + AMP 的方式运行

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。

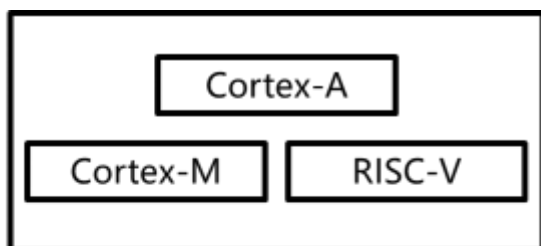
在运行平台方面，Linux 提供标准的 Linux Kernel，RTOS 提供开源的 RT-Thread，Bare-metal 提供基于 RK HAL 硬件抽象层的裸机开发库。同时，瑞芯微多核异构系统支持客户自行适配更多的运行平台，例如可以基于 RK HAL 硬件抽象层适配指定的 RTOS 等。



- 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

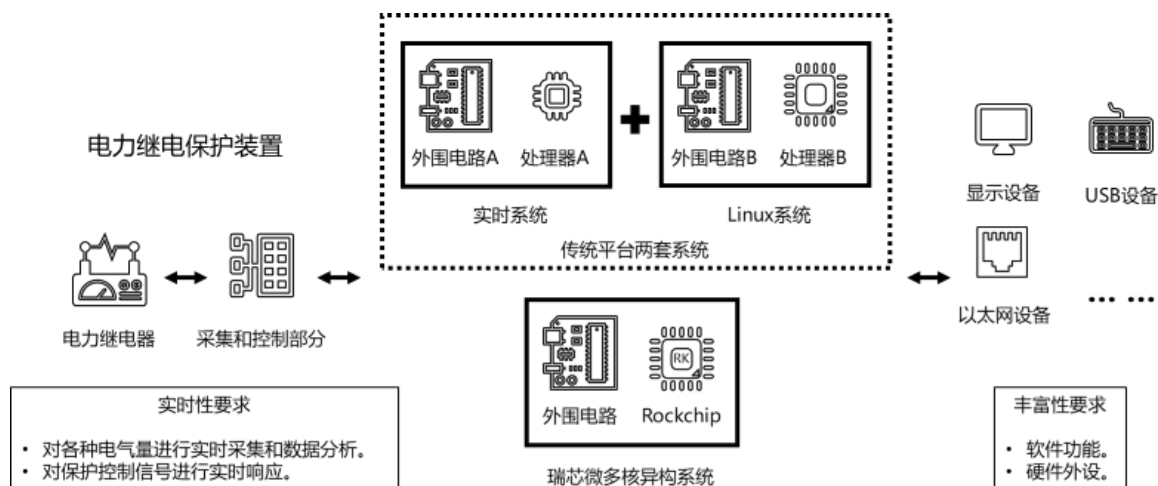
1.2.6.2 运行平台支持

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

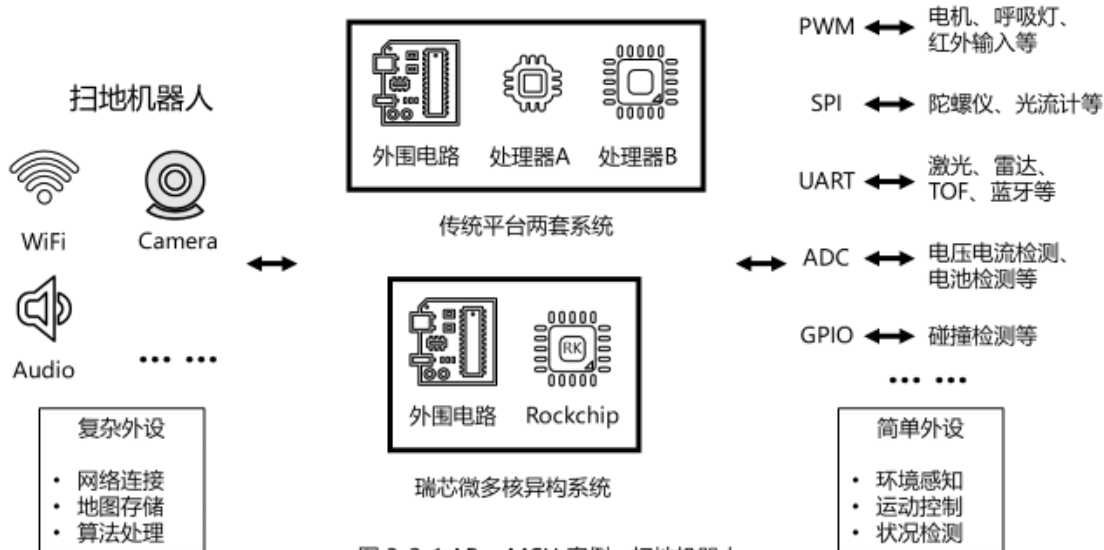
在电力继电保护装置中，既对系统的实时性有要求，例如对各种电气量进行实时采集和数据分析、对保护控制信号进行实时响应等。又对系统的丰富性有要求，需要使用复杂的软件功能和硬件外设，例如显示设备、USB 设备、以太网设备等。使用瑞芯微多核异构系统，将传统平台两套系统合二为一，一套板卡就能同时独立运行 Linux 系统和实时系统，实现上述所有功能。



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
```

		└─ Image	
		└─ amp.img	# 打包后的 Bare-metal 系统固件
		└─ 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			
		└─ halx.bin	
		└─ halx.elf	
		└─ parameter.txt	# 分区表配置文件
		└─ mkimage.sh	# AP 固件打包脚本
		└─ src	
		└─ hal_conf.h	# RK HAL 配置文件
		└─ main.c	# Bare-metal 系统示例
		└─ test_demo.c	# Bare-metal 系统模块与功能示例
		└─ rk3562-mcu	
		└─ GCC	
		└─ gcc_bus_m0.ld	# MCU 链接脚本
		└─ Makefile	# MCU 编译文件
		└─ Image	
		└─ amp.img	# 打包后的 Bare-metal 系统固件
		└─ amp_mcu.its	# Linux + RTOS / Bare-metal MCU 固件打包
配置文件			
		└─ mcu.bin	
		└─ mcu.elf	
		└─ parameter.txt	# 分区表配置文件
		└─ mkimage.sh	# MCU 固件打包脚本
		└─ src	
		└─ hal_conf.h	# RK HAL 配置文件
		└─ main.c	# Bare-metal 系统示例
		└─ test_demo.c	# Bare-metal 系统模块与功能示例
		└─ test	# Bare-metal 系统模块测试文件
		└─ tools	# Bare-metal 系统工具集

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](#)

2.5 rtos 目录

rtos 目录存放 RTOS 系统源码。AMP SDK 提供开源的 [RT-Thread](#)。

RT-Thread 是中国自主开发、国内最成熟稳定、装机量最大的开源 RTOS。RT-Thread 平台是一个集实时操作系统内核、中间件组件、开源开发者社区等于一体的技术平台。可以访问 [RT-Thread 文档中心](#) 查看在线技术文档。

瑞芯微多核异构系统中提供的 RT-Thread 基于 RK HAL 进行适配。RK HAL 文件在 `bsp/rockchip/common/hal/lib`。RTOS Device Driver 文件在 `bsp/rockchip/common/drivers`。

各芯片平台支持情况如下：

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
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_memory {                               # 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_memory {
    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_memory {
        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 资源分配

4.1.1 内存资源

以RK3308为例，RK3308支持针对不同CPU配置不同大小的内存，用户可根据自身需求，通过修改配置文件进行配置。

```
device/rockchip/rk3308/rockchip_rk3308*_cfg
device/rockchip/rk3308/***.its
```

注意：各个CPU所占用的内存之间，以及与共享内存之间，不能有内存冲突。

4.1.1.1 SRAM内存分区配置

SRAM内存分区包括CPU0到CPU3的独占内存，可用于对SDRAM的扩展使用。

以RTOS+Bare-metal为例，SRAM内存的默认配置如下：

File: device/rockchip/rk3308/rockchip_rk3308_rtos_amp.its

```
amp0 {
    # .....
    srambase      = <0xffffb0000>;    # CPU0 SRAM起始地址
    sramsize      = <0x00010000>;    # CPU0 SRAM分配大小
    # .....
};

amp1 {
    # .....
    srambase      = <0xffff88000>;    # CPU1 SRAM起始地址
    sramsize      = <0x00008000>;    # CPU1 SRAM分配大小
    # .....
};

amp2 {
    # .....
    srambase      = <0xffff90000>;    # CPU2 SRAM起始地址
    sramsize      = <0x00010000>;    # CPU2 SRAM分配大小
    # .....
};

amp3 {
    # .....
    srambase      = <0xffffa0000>;    # CPU3 SRAM起始地址
    sramsize      = <0x00010000>;    # CPU3 SRAM分配大小
    # .....
};
```

以Linux+RTOS/Bare-metal为例，SRAM内存的默认配置如下：

File: device/rockchip/rk3308/rockchip_rk3308_rtos_linux.its

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

    images {
        amp3 {
            # .....
            sys          = "rtt";          # 系统: "rtt" or "hal"
            cpu          = <0x3>;          # CPU ID
            # .....
            srambase     = <0>;            # SRAM起始地址（根据需要分配，不需要为0）
            sramsize     = <0>;            # SRAM分配大小（根据需要分配，不需要为0）
            # .....
        };
    };
    # .....
};
# .....
};
```

4.1.1.2 SDRAM 内存分区配置

SDRAM内存分区包括CPU0到CPU3的独占内存，以及CPU0到CPU3共享内存部分，用户根据不同的CPU使用情况分配其大小。

以RTOS+Bare-metal为例，SDRAM内存的默认配置如下：

File: device/rockchip/rk3308/rockchip_rk3308_rtos_amp.its

```
amp0 {
    # .....
    load          = <0x02600000>;        # CPU0 SDRAM起始地址
    size          = <0x00900000>;        # CPU0 SDRAM分配大小
    # .....
};

amp1 {
    # .....
    load          = <0x00800000>;        # CPU1 SDRAM起始地址
    size          = <0x00a00000>;        # CPU1 SDRAM分配大小
    # .....
};

amp2 {
    # .....
    load          = <0x01200000>;        # CPU2 SDRAM起始地址
    size          = <0x00a00000>;        # CPU2 SDRAM分配大小
    # .....
};
```

```

    amp3 {
        # .....
        load      = <0x01c00000>;    # CPU3 SDRAM起始地址
        size      = <0x00a00000>;    # CPU3 SDRAM分配大小
        # .....
    };

# .....
    share_memory {
        base      = <0x02f00000>;    # 共享SDRAM内存起始地址
        size      = <0x00100000>;    # 共享SDRAM内存分配大小
    };

# .....

```

以Linux+RTOS/Bare-metal为例，SDRAM内存的默认配置如下：

File: device/rockchip/rk3308/rockchip_rk3308_rtos_linux.its

```

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

    images {
        amp3 {
            # .....
            sys      = "rtt";          # 系统: "rtt" or "hal"
            cpu      = <0x3>;          # CPU ID
            # .....
            load     = <0x02e00000>;    # SDRAM起始地址
            size     = <0x00400000>;    # SDRAM分配大小
            # .....
        };
    };

    # .....
};

# 共享内存信息
share_memory {
    base      = <0x03200000>;    # 共享SDRAM内存起始地址
    size      = <0x00100000>;    # 共享SDRAM内存分配大小
};

# .....
};

```

在该示例中，RTOS的内核配置为CPU3，运行RT-Thread系统。RTOS(CPU3)内存分配起始地址为0x02e00000。Kernel+RTT(HAL)的共享内存起始地址0x03200000，紧接在RTOS(CPU3)的内存分配空间之后。RTOS的内存与共享内存，都必须分配在Kernel预留的空间之内。

4.1.1.3 共享内存分区配置

共享内存为SDRAM中CPU0到CPU3独占内存之外的部分，按照上述SDRAM内存分区所定义，起始地址为SHMEM_BASE，大小为SHMEM_SIZE。

共享内存用于CPU0到CPU3之间的通信、数据共享等使用。CPU0到CPU3都可访问该区间数据。SRAM内存的默认配置如下：

File: device/rockchip/rk3308/rockchip_rk3308_rtos_amp_32bit_cfg

```
## 第4章 Share Memory config
RTT_SHRPMMSG_SIZE=0x00080000
RTT_SHRAMFS_SIZE=0x00020000
RTT_SHLOG0_SIZE=0x00001000
RTT_SHLOG1_SIZE=0x00001000
RTT_SHLOG2_SIZE=0x00001000
RTT_SHLOG3_SIZE=0x00001000

## 第4章 HAL config
## 第4章 Share memory config same as RTOS
SHRPMMSG_SIZE=$RTT_SHRPMMSG_SIZE
SHRAMFS_SIZE=$RTT_SHRAMFS_SIZE
SHLOG0_SIZE=$RTT_SHLOG0_SIZE
SHLOG1_SIZE=$RTT_SHLOG1_SIZE
SHLOG2_SIZE=$RTT_SHLOG2_SIZE
SHLOG3_SIZE=$RTT_SHLOG3_SIZE
```

4.1.2 Cache和MMU配置

以RK3308为例，分别介绍不同系统下的Cache和MMU配置。

4.1.2.1 Kernel

/to-do/

4.1.2.2 HAL

HAL程序中增加自定义内存，需要在hal/lib/CMSIS/Device/RK3308/Source/Templates/mmu_rk3308.c

中配置对应内存属性。默认配置如下：

```
/*
 * Define MMU flat-map regions and attributes
 */
// Define dram address space
#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);

    //----- PERIPHERALS -----
    MMU_TTSection(MMUTable, 0xFF000000, 15U, Sect_Device_RW);

    //----- INTERNAL SRAM -----
    MMU_TTSection(MMUTable, 0xFFFF0000, 1, Sect_Normal);

```

HAL中使用CMSIS标准方案，内存属性参考hal/lib/CMSIS/Core_A/Include/core_ca.h

4.1.2.3 RT-Thread

RT-Thread程序中增加自定义内存，需要在rtos/bsp/rockchip/rk3568-32/board/common/board_base.c

中配置对应内存属性。默认配置如下：

```

struct mem_desc platform_mem_desc[] =
{
#ifdef RT_USING_UNCACHE_HEA
    {FIRMWARE_BASE, FIRMWARE_BASE + FIRMWARE_SIZE - 1, FIRMWARE_BASE,
    NORMAL_MEM},
    {RT_UNCACHE_HEAP_BASE, RT_UNCACHE_HEAP_BASE + RT_UNCACHE_HEAP_SIZE - 1,
    RT_UNCACHE_HEAP_BASE, UNCACHED_MEM},
#else
    {FIRMWARE_BASE, FIRMWARE_BASE + DRAM_SIZE - 1, FIRMWARE_BASE, NORMAL_MEM},
#endif
    {SHMEM_BASE, SHMEM_BASE + SHMEM_SIZE - 1, SHMEM_BASE, SHARED_MEM},
#ifdef LINUX_RPMSG_BASE
    {LINUX_RPMSG_BASE, LINUX_RPMSG_BASE + LINUX_RPMSG_SIZE - 1, LINUX_RPMSG_BASE,
    UNCACHED_MEM},
#endif
    {0xFF000000, 0xFFFF0000 - 1, 0xFF000000, DEVICE_MEM}, /* DEVICE */
    {0xFFFF0000, 0xFFFFFFFF, 0xFFFF0000, NORMAL_MEM} /* SRAM */
};

```

4.1.3 中断资源

（说明主核做完整初始化，从核做配置覆盖）

4.1.4 外设资源

（说明软件需要自行划分外设，说明未来的硬件资源划分）

4.2 资源保护

在Linux + RTOS/Bare-metal模式下，不同系统间会存在资源的竞争。所以在RTOS/Bare-metal中使用到的一些外设、时钟等资源时需要在kernel/arch/arm(64)/rockchip/rkxxx-amp.dtsi文件中保护一下，避免和Linux那边的资源冲突。

以RK3308的uart1为例，在Linux + RTOS/Bare-metal模式下，默认Linux系统运行的核心为主核，RTOS/Bare-metal为从核。

4.2.1 时钟资源

从核这边的系统使用uart1定时器时，需要在Linux系统中进行时钟相关配置。

File: SDK/ernel/arch/arm64/rockchip/rk3308b-amp.dtsi

```
/ {
    rockchip_amp: rockchip-amp {
        compatible = "rockchip,amp";

        # 将uart1等使用到的外设的时钟资源分配给从核端的系统
        clocks = <&cru SCLK_UART1>, <&cru PCLK_UART1>,
                <&cru PCLK_TIMER>, <&cru SCLK_TIMER4>, <&cru SCLK_TIMER5>;

        #参考rk308.dtsi中的uart1节点配置uart1的pinctrl属性
        pinctrl-names = "default";
        pinctrl-0 = <&uart1_xfer>;
        status = "okay";

        # .....
```

4.2.2 引脚资源

（说明Linux中的保护）

4.2.3 电源域资源

当运行Linux + RTOS/Bare-metal的AMP模式时，可以根据项目开发的需求自定义Linux端和 HAL/RT-Thread端各占核心的数量。以Linux端占CPU0到CPU2三个核心，HAL/RT-Thread端占据CPU3一个核心为例，配置电源域资源。

File: SDK/kernel/arch/arm64/boot/dts/rockchip/rk3308b-amp.dtsi

```
## 第4章 .....
## 第4章 关闭Linux CPU3的节点，使Linux启动时不会运行在CPU3上
&cpu3 {
    status = "disabled";
};
```

5. 第5章 启动方案

5.1 AP启动方案

SDK 支持 AMP 混合架构设计，使得不同的 CPU 可以运行不同的系统，以满足灵活的产品设计需求。目前支持 RTT、Linux、HAL的混合结构模型，允许这三种系统相互组合或者独立运行，以RK3562为例：

5.1.1 U-Boot 阶段

如果有使用到Linux，在U-Boot中需要指定加载 Linux 内核镜像的内存起始地址，用于启动 Linux 内核，需要依据实际RAM情况进行配置，如下为RK3562 配置示例：

```
diff --git a/include/configs/rk3562_common.h b/include/configs/rk3562_common.h
index b076f90d6b..4e0b333357 100644
--- a/include/configs/rk3562_common.h
+++ b/include/configs/rk3562_common.h
@@ -62,8 +62,8 @@
     "scriptaddr=0x00c00000\0" \
     "pxefile_addr_r=0x00e00000\0" \
     "fdt_addr_r=0x08300000\0" \
-    "kernel_addr_r=0x00400000\0" \
-    "kernel_addr_c=0x04080000\0" \
+    "kernel_addr_r=0x02000000\0" \
+    "kernel_addr_c=0x04880000\0" \
     "ramdisk_addr_r=0x0a200000\0"

#include <config_distro_bootcmd.h>
--
2.38.0
```

5.1.2 Linux + HAL

AMP 混合架构设计支持Linux + HAL 的模式，在一般情况下CPU0~CPU2运行Linux系统，CPU3运行HAL

系统	CPU	说明
Linux	CPU0 CPU1 CPU2	执行 Linux 系统
HAL	CPU3	执行裸核系统

5.1.3 Linux + RTOS

AMP 混合架构设计支持Linux + RTOS 的模式，在一般情况下CPU0~CPU2运行Linux系统，CPU3运行RTOS

系统	CPU	说明
Linux	CPU0 CPU1 CPU2	执行 Linux 系统
RT-Thread	CPU3	执行 RT-Thread 系统

5.1.4 RTOS + HAL

AMP 混合架构设计支持RTOS + HAL 的模式，组合较为灵活，可以运行如RTOS + 3 * HAL、2 * RTOS + 2 * HAL等

系统	CPU	说明
RTT	CPU1	执行 RT-Thread 系统
HAL	CPU2	执行裸核系统
HAL	CPU3	执行裸核系统
HAL	CPU0	执行裸核系统

5.1.5 4 * RTOS

4.1.x分支的RT-Thread同时支持AMP和SMP，及4个核同时跑4个RTOS或者只跑一个RTOS

系统	CPU	说明
RTT	CPU1	执行 RT-Thread 系统
RTT	CPU2	执行 RT-Thread 系统
RTT	CPU3	执行 RT-Thread 系统
RTT	CPU0	执行 RT-Thread 系统

如果需要SMP系统，可以添加如下配置

```
CONFIG_RT_USING_SMP=Y
CONFIG_RT_CPUS_NR=4
```

ITS可以参考如下配置：

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

    images {

        amp0 {
            description = "bare-mental-core0";
            data = /incbin/("rtt0.bin");
```

```

        type          = "firmware";
        compression   = "none";
        arch          = "arm";      // "arm64" or "arm"
        cpu           = <0x000>;    // mpidr
        thumb         = <0>;        // 0: arm or thumb2; 1: thumb
        hyp           = <0>;        // 0: el1/svc; 1: el2/hyp
        load          = <0x02600000>;
        udelay        = <10000>;
        hash {
            algo = "sha256";
        };
    };

};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;
        loadables = "amp0";

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

5.1.6 4 * HAL

AMP 系统可以支持所有核都运行在HAL上

系统	CPU	说明
HAL	CPU1	执行裸核系统
HAL	CPU2	执行裸核系统
HAL	CPU3	执行裸核系统
HAL	CPU0	执行裸核系统

5.2 MCU启动方案

利用AMP特性，将AP作为主要核心，MCU作为辅助核心跑裸核系统，辅助AMP系统实现快速响应和控制，在启动MCU时需要为其分配内存空间，以RK3562 MCU为例介绍。

5.2.1 U-Boot阶段启动

在MCU启动时需要在uboot中为MCU指定对应的RAM运行大小和起始地址，如Cache映射范围，MCU启动地址，SDK发布时会提供一套默认的配置，以下为RK3562的uboot配置：

```
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 */
    writel(0xfc000000, SYS_GRF_BASE + SYS_GRF_SOC_CON5);
    writel(0xffb40000, SYS_GRF_BASE + SYS_GRF_SOC_CON6);

    sip_smc_mcu_config(ROCKCHIP_SIP_CONFIG_BUSMCU_0_ID,
                      ROCKCHIP_SIP_CONFIG_MCU_CODE_START_ADDR,
                      0xffff0000 | (entry_point >> 16));
    /* 0x07c00000 is mapped to 0xa0000000 and used as shared memory for rpmsg */
    sip_smc_mcu_config(ROCKCHIP_SIP_CONFIG_BUSMCU_0_ID,
                      ROCKCHIP_SIP_CONFIG_MCU_EXPERI_START_ADDR, 0xffff07c0);

    /* release dcache / icache / bus m0 jtag / bus m0 */
    writel(0x03280000, TOP_CRU_BASE + TOP_CRU_SOFTTRST_CON23);

    /* release pmu m0 jtag / pmu m0 */
    /* writel(0x00050000, PMU1_CRU_BASE + PMU1_CRU_SOFTTRST_CON02); */

    return 0;
}
```

5.2.2 MCU启动阶段

MCU的RAM启动地址可以通过ITS配置去指定，在uboot中会进行解析，以下为RK3562 MCU的ITS配置示例：

```
/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 = <0x08200000>; //MCU 程序RAM启动地址
        udelay = <1000000>; //启动延时时间
    }
}
```

```

        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";
            };
        };
    };
};
};
};

```

5.3 AMP固件打包

SDK 发布时会提供一套默认的打包编译方式，一般在对应的工程目录下(project/rkXXX/mkImage.sh)。

使用者也可以自行编写自定义的打包方式：

1. 将 mkimage_for_windows\usr\bin 的路径添加至系统环境变量的 Path 变量中（打包工具由Rockchip提供，一般在SDK的tools/windows/RKImageMaker目录下）
2. 将编译生成的TestDemo.bin文件和 同级目录下的amp.its 文件拷贝和 mkimage.exe 放到同级目录下
3. 在 windows 下执行 mkimage.exe -f amp.its -E -p 0xe00 amp.img 命令就会在同级目录下生成 amp.img

AMP 固件打包需要依赖ITS 配置文件通过打包工具进行打包，主要需要配置不同CPU的RAM加载地址以及启动延迟时间，RAM的加载地址需要与内存资源划分相匹配，由于AMP 组合的灵活性，以下我们以RK3562为例介绍配置：

4*HAL或者4*RTOS的ITS配置如下：

```

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

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

    images {

        amp0 {
            description = "bare-mental-core0";
            data = /incbin/("hal0.bin");
            type = "firmware";

```

```

        compression = "none";
        arch         = "arm";      // "arm64" or "arm"
        cpu          = <0x0>;      // mpidr
        thumb        = <0>;        // 0: arm or thumb2; 1: thumb
        hyp           = <0>;        // 0: el1/svc; 1: el2/hyp
        load          = <0x02000000>;
        udelay        = <10000>;
        hash {
            algo = "sha256";
        };
};

amp1 {
    description = "bare-mental-core1";
    data        = /incbin/("hal1.bin");
    type        = "firmware";
    compression = "none";
    arch        = "arm";
    cpu         = <0x1>;
    thumb       = <0>;
    hyp         = <0>;
    load        = <0x00800000>;
    udelay      = <10000>;
    hash {
        algo = "sha256";
    };
};

amp2 {
    description = "bare-mental-core2";
    data        = /incbin/("hal2.bin");
    type        = "firmware";
    compression = "none";
    arch        = "arm";
    cpu         = <0x2>;
    thumb       = <0>;
    hyp         = <0>;
    load        = <0x01000000>;
    udelay      = <10000>;
    hash {
        algo = "sha256";
    };
};

amp3 {
    description = "bare-mental-core3";
    data        = /incbin/("hal3.bin");
    type        = "firmware";
    compression = "none";
    arch        = "arm";
    cpu         = <0x3>;
    thumb       = <0>;
    hyp         = <0>;
    load        = <0x01800000>;
    udelay      = <10000>;
    hash {
        algo = "sha256";
    };
};

```



```

    };

};

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";
        };
    };
};
};
};

```

Linux + HAL 或者 Linux + RTOS 示例：

```

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

    images {
        amp3 {
            description = "bare-mental-core3";
            data = /incbin/("rtt3.bin");
            type = "firmware";
            compression = "none";
            arch = "arm";
            cpu = <0x3>;
            thumb = <0>;
            hyp = <0>;
            load = <0x01800000>;
            udelay = <10000>;
            hash {
                algo = "sha256";
            };
        };
    };

};

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

6. 第6章 通信方案

6.1 核间中断触发

瑞芯微AMP通信方案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。这是一种兼容异构处理器的通用核间中断触发方式。

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。

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。

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。

目前核间中断触发方式支持mailbox、软件中断，共享内存Linux与仅支持uncache。

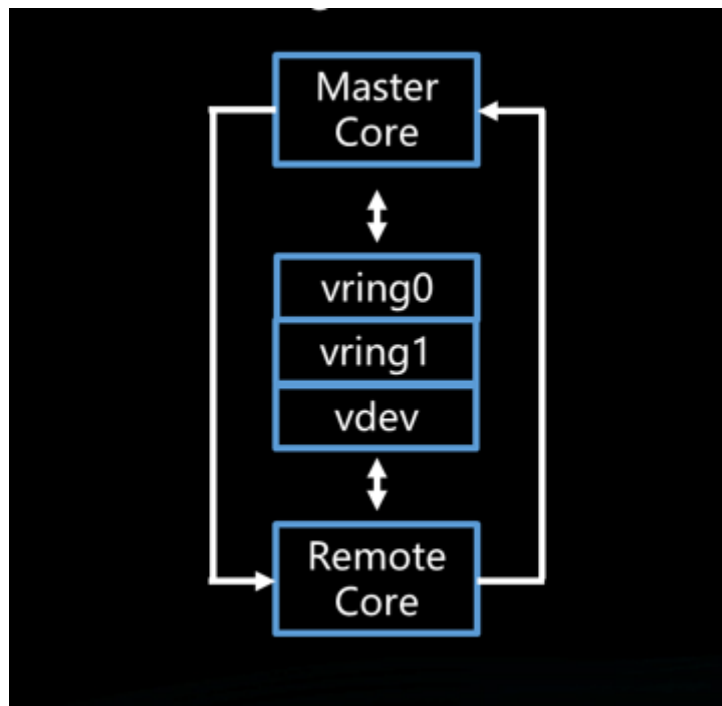
Linux下mailbox中断的方式参考如下路径的代码：drivers/rpmsg/rockchip_rpmsg_mbox.c。

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

6.3 RPMsg协议方案


6.3.1 标准框架


瑞芯微多核异构系统提供RPMsg协议标准框架方案。支持RPMsg Name Service功能。Linux Kernel适配RPMsg，RTOS和Bare-metal适配RPMsg-Lite。从如下框图可以看到，RPMsg也是由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

主核发送时，主核从vring0中取得一块buffer，再将消息按照RPMsg协议填充。消息通过队列发送到vring1中，触发从核中断，通知从核有消息待处理。从核根据队列从vring1中取得对应的buffer，并将消息传递给注册的endpoint callback。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，则与主核发送流程相反。通信过程中的共享数据放在vdev buffer中。以下两个图片为主核发送和从核发送的流程框图。

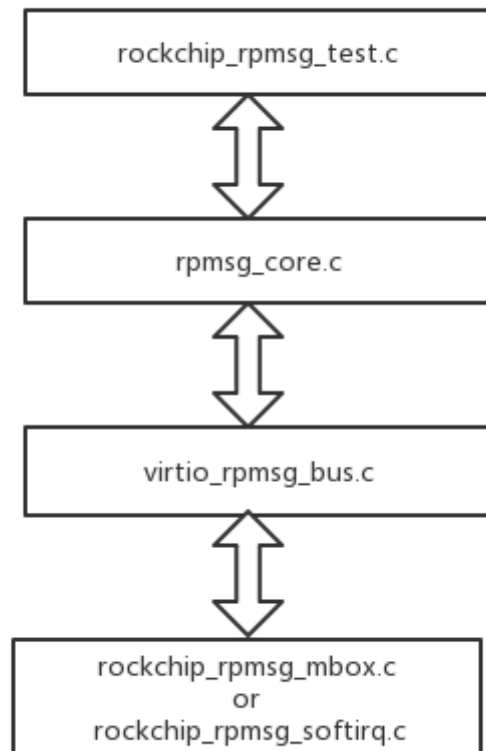
 rpmmsg-communication-process1

 rpmmsg-communication-process2

6.3.3 Linux Kernel适配RPMsg

6.3.3.1 代码结构

Linux Kerne RPMsg主要代码结构如下图。



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+Shared Memory底层驱动接口实现的物理层（Physical Layer）。

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+Shared Memory底层驱动接口实现的物理层（Physical Layer）。

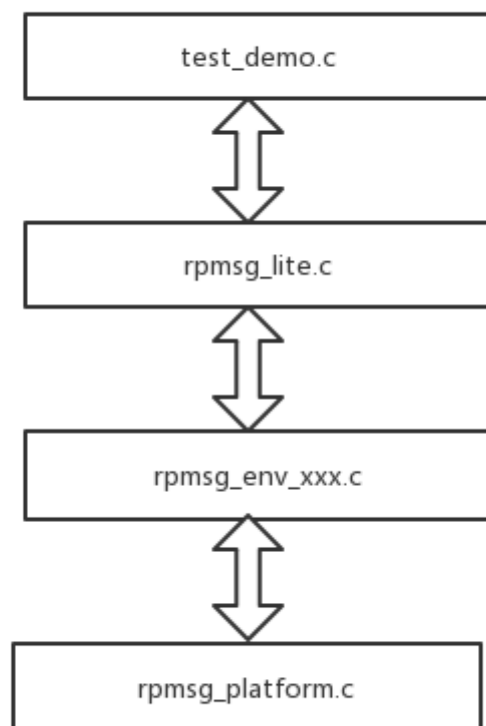
virtio_rpmsg_bus.c是注册在VirtIO Bus上的driver，同时向RPMsg Bus注册device。VirtIO和Virtqueue是通用RPMsg协议选择的MAC层（MAC Layer）。

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

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。

6.3.3.2 RTOS端适配RPMsg-Lite

RTOS RPMsg主要代码结构如下图。



RPMsg-Lite由恩智浦（NXP）提供，结构与Linux RPMsg类似。RPMsg-Lite提供不同RTOS的支持，即框图中的rpmsg_env_xxx.c。瑞芯微多核异构系统适配了Bare-metal和RT-Thread，客户可以直接使用。当然，也可以参考这部分代码，实现对指定RTOS的支持。

6.4 RPMsg测试示例

6.4.1 RK3308

6.4.1.1 kernel+rtt

RK3308使用标准的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
```

6.4.1.1.1 共享内存

以SDK提供的demo为例，划分5M共享内存给RPMMSG，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/rk3308b-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},

```

6.4.1.1.2 测试demo

Kernel Demo

在kernel工程中修改配置文件kernel/arch/arm64/configs/rk3308_linux_defconfig。

配置菜单配置:

```

make ARCH=arm64 rk3308_linux_defconfig
make ARCH=arm64 menuconfig
# 打开以下宏开关
CONFIG_RPMSG_ROCKCHIP_TEST
make ARCH=arm64 savedefconfig
cp defconfig arch/arm64/configs/rk3308_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。

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%x shmem_end-0x%x\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_recv(info->instance, remote_queue,
                    (uint32_t *)&master_ept_id, rx_msg,
                    RL_BUFFER_PAYLOAD_SIZE, RL_NULL, RL_BLOCK);
    //    rpmsg_queue_recv_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,
RPMMSG_RTT_TEST_MSG, strlen(RPMMSG_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

测试成功log

Linux master core RPMMSG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)
```

6.4.1.2 RTT+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。

6.4.1.2.1 共享内存

RTT共享内存开始的地址及大小

Path;

/* to-do*/

共享内存区域具体分配

Path: rtos/bsp/rockchip/rk3308-32/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

Path:

/* to-do*/

共享内存区域具体分配

Path: hal/project/rk3308/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

```

6.4.1.2.2 测试demo

RTT Demo

Path: SDK/rtos/bsp/rockchip/rk3308-32

配置菜单配置: scones --menuconfig

```

CONFIG_RT_USING_RPMMSG_LITE=y
CONFIG_RT_USING_COMMON_TEST_RPMMSG_LITE=y

```

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

RPMsmsg-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()	RPMsmsg-lite master 端初始化
rpmsg_lite_remote_init()	RPMsmsg-lite remote 端初始化
rpmsg_lite_wait_for_link_up()	RPMsmsg-lite remote 端等待初始化链接成功
rpmsg_queue_create()	RPMsmsg-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

HAL Demo

File: SDK/hal/project/rk3308/src/main.c

```
#define TEST_DEMO
#define TEST_USE_RPMMSG_INIT
```

File: SDK/hal/project/rk3308/src/test_demo.c

```
#define RPMMSG_TEST
```

HAL Demo Path: hal/project/rk3568/src/test_demo.c

RPMMsg-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()	RPMMsg-lite master 端初始化
rpmsg_lite_remote_init()	RPMMsg-lite remote 端初始化
rpmsg_lite_wait_for_link_up()	RPMMsg-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

测试结果

在串口终端输入串口命令查看log信息。

```
## 第6章 RPMMSG 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_recv: 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_recv: 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_recv: 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_recv: 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 ARM GIC v2

RK3308 用的 GIC400，走的是 GICv2 的接口，支持优先级抢占和 CPU route。对于 GICv2 来说，有三种中断：

1. SGI: 中断号 0-15，这是软件产生的中断，每个 CPU 私有。
2. PPI: 中断号 16-31，这是私有的外设中断，也是每个 CPU 私有。
3. SPI: 中断号 32-1019，这是所有 CPU 共享的外设中断。

7.1.1 HAL 中断使用示例

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。
4. 模块中断的配置与使能：每个模块有独立的中断配置与使能，并且因模块的功能设计不同，其配置使用方法也不尽相同。具体使用方法请参考对应的模块代码。

下面以 GPIO0 和 TIMER0 为例，来简单介绍 RK3308 HAL 中断的使用方法。

7.1.2 HAL GIC中断配置表

在该 SDK 中，GIC 的中断配置位于 `hal/project/rk3308/src/main.c` 文件中。参考如下：

```
## 第7章 自定义 GIC 中断信息表
static struct GIC_AMP_IRQ_INIT_CFG irqConfig[] = {

    # The priority higher than 0x80 is non-secure interrupt.

#ifdef AMP_LINUX_ENABLE
    GIC_AMP_IRQ_CFG_ROUTE(RPMMSG_03_IRQn, 0xd0, CPU_GET_AFFINITY(3, 0)),
#endif
#ifdef TEST_USE_UART1M0
    GIC_AMP_IRQ_CFG_ROUTE(UART1_IRQn, 0xd0, CPU_GET_AFFINITY(3, 0)),
#endif
#else // #ifdef AMP_LINUX_ENABLE
    GIC_AMP_IRQ_CFG_ROUTE(AMP0_IRQn, 0xd0, CPU_GET_AFFINITY(0, 0)),

    #.....

    # GIC 中断配置结束标志，不能删，并且所有有效配置都要放在这个配置前面
    GIC_AMP_IRQ_CFG_ROUTE(0, 0, CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0)), /*
sentinel */
};

## 第7章 默认 GIC 中断信息表
```



```
static struct GIC_IRQAMP_CTRL irqConfig = {
    #默认使用DEFAULT_IRQ_CPU来初始化 GIC 配置
    .cpuAff = CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0),
    # 默认中断优先级
    .defPrio = 0xd0,
    # 默认使用DEFAULT_IRQ_CPU来处理中断服务程序
    .defRouteAff = CPU_GET_AFFINITY(DEFAULT_IRQ_CPU, 0),
    # 用户自定义 GIC 中断信息配置表。通过该表更改以上默认配
    .irqsCfg = &irqsConfig[0],
};

void main(void)
{
    #.....

    # GIC 中断初始化。
    HAL_GIC_Init(&irqConfig);
    #.....
}
```

在以上的示例中，irqConfig 为初始化默认的中断信息配置表。当用户没有自定义中断信息时，系统将按照该表的规则进行中断处理。如果用户需要修改中断配置信息，可在 irqsConfig[] 结构体中进行修改。

GIC_AMP_IRQ_CFG_ROUTE(p1, p2, p3) 参数说明如下：

参数	说明
p1	中断号
p2	优先级
p3	route

CPU_GET_AFFINITY(p1, p2) 参数说明如下：

参数	说明	备注
p1	cpu_id	分别对应cpu 0~3
p2	cluster_id	对于 RK3308，始终为 0

7.1.3 HAL GPIO中断示例

下面以 GPIO0 的 C4 pin 脚为例，简单说明 GPIO 中断使用方法。

```
## 第7章 GPIO0 中断服务程序总入口
static void gpio_isr(int vector, void *param)
{
    #.....
    HAL_GPIO_IRQHandler(GPIO0, GPIO_BANK0);
    #.....
}

## 第7章 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;
}

## 第7章 GPIO脚中断使用示例
static void gpio_test(void)
{
    #.....

    # 设置 GPIO0 C4 为输入口
    HAL_GPIO_SetPinDirection(GPIO0, GPIO_PIN_C4, GPIO_IN);

    # 设置 GIC (GPIO0) 中断服务程序并使能中断
    HAL_IRQ_HANDLER_SetIRQHandler(GPIO0_IRQn, gpio_isr, NULL);
    HAL_GIC_Enable(GPIO0_IRQn);

    # 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);
    HAL_GPIO_EnableIRQ(GPIO0, GPIO_PIN_C4);
}

```

在该示例中，gpio_isr() 为 GPIO0 的 GIC 中断服务程序，该函数中通过 HAL_GPIO_IRQHandler() 来回调 GPIO0 C4 pin对应的回调函数 c4_call_back() 来处理 GPIO0 C4 pin脚所产生的中断服务程序。

7.1.4 HAL TIMER中断示例

下面以 TIMER0 为例，简单介绍 TIMER 中断的使用方法。

```

static void timer_isr(int vector, void *param)
{
    #.....
}

static void timer_test(void)
{
    #.....

    # 设置 GIC (TIMER0) 中断服务程序并使能中断
    HAL_IRQ_HANDLER_SetIRQHandler(TIMER0_IRQn, timer_isr, NULL);
    HAL_GIC_Enable(TIMER0_IRQn);

    # 设置 TIMER0 配置信息，并以中断方式启动 TIMER0
    HAL_TIMER_Init(TIMER0, TIMER_FREE_RUNNING);
    HAL_TIMER_SetCount(TIMER0, 24000000);
    HAL_TIMER_Start_IT(TIMER0);
}

```

7.1.5 HAL 软中断使用方法

在 RK3308 中，不但可以通过硬件进行中断触发，也可以通过软件的方式进行中断触发。

通过软件进行中断触发之前，首先需要参考前面小结用例的介绍，进行相应的中断配置以及初始化工作。在用户需要进行软件触发的位置，通过调用以下接口函数，触发软件中断。

```
HAL_GIC_SetPending(IRQn); // IRQn: 要触发的中断号
```

7.1.6 HAL 中断用例小结

在以上的几个示例中，我们简单的介绍了 RK3308 AMP HAL 中断的使用方法，总结如下：

1. 在 main.c 系统初始化时，通过调用 HAL_GIC_Init(&irqConfig) 函数进行中断信息的初始化配置。其中包含对 GPIO0、TIMER0 的中断信息初始化配置。
2. 在 gpio_test()、timer_test() 等中断测试用例中，分别通过 HAL_IRQ_HANDLER_SetIRQHandler() 注册对应的 GIC 中断服务程序，并通过 HAL_GIC_Enable() 函数使能对应的中断。
3. 根据 GPIO 和 TIMER 等不同的硬件模块，调用其对应的接口进行硬件、中断等信息配置，并使能对应的模块中断。

7.2 ARM GIC v3

7.3 ARM NVIC

7.4 RISC-V中断控制器

8. 第8章 模块

(模块章节按照：简介、示例芯片、运行平台、测试、详细文档等部分编写，字母排序)

8.1 UART

8.1.1 RK3308

8.1.1.1 U-Boot

File: SDK/device/rockchip/.chips/rk3308/rockchip_rk3308_xxx_defconfig

```
RK_UBOOT_INI="RK3308MINIALL.ini"          # UART2
or
RK_UBOOT_INI="RK3308MINIALL_UART4.ini"    # UART4
```

8.1.1.2 Kernel

File: SDK/kernel/arch/arm64/boot/dts/rockchip/rk3308-evb.dts

```
## 第8章 UART2
&uart2 {
    pinctrl-names = "default";
    pinctrl-0 = <&uart2m0_xfer>;
    status = "okay";
};
or
## 第8章 UART4
&uart4 {
    pinctrl-names = "default";
    pinctrl-0 = <&uart4_xfer>;
    status = "okay";
};
```

File: SDK/kernel/arch/arm64/boot/dts/rockchip/rk3308b-evb-v10.dtsi

```
## 第8章 UART2
&fiq_debugger {
    rockchip,serial-id = <2>;
    status = "okay";
};
or
## 第8章 UART4
&fiq_debugger {
    rockchip,serial-id = <4>;
    status = "okay";
};
```

8.1.1.3 RT-Thread

Path: SDK/rtos/bsp/rockchip/rk3308-32

配置菜单配置: `scons --menuconfig`

```
## 第8章 UART2
CONFIG_RT_CONSOLE_DEVICE_NAME="uart2"
CONFIG_RT_USING_UART=y
CONFIG_RT_USING_UART2=y
or
## 第8章 UART4
CONFIG_RT_CONSOLE_DEVICE_NAME="uart4"
CONFIG_RT_USING_UART=y
CONFIG_RT_USING_UART4=y
```

8.1.1.4 HAL

File: SDK/hal/project/rk3308/src/main.c

```
## 第8章 UART2
static struct UART_REG *pUart = UART4;
or
## 第8章 UART4
static struct UART_REG *pUart = UART2;
```

8.2 EMMC

8.2.1 RK3308

8.2.1.1 Kernel

kernel默认配置为SFC Flash。如果用户需要在kernel下使用eMMC Flash，按照以下方式修改：

File: SDK/arch/arm64/boot/dts/rockchip/rk3308b-evb-v10.dtsi

```
## 第8章 .....
&emmc {
    # .....
    status = "okay";    # 打开EMMC
};

## 第8章 .....
&sfc {
    status = "disabled";    # 关闭 SFC
};
```

注意：kernel下使用eMMC Flash时，在固件升级时需要选择“rootfs”选项，参考如下：

#	<input type="checkbox"/>	存储	地址	名字	路径	...
1	<input checked="" type="checkbox"/>		0x00000000	Loader	F:\img\rk3308\rk3308_loader_v...	
2	<input checked="" type="checkbox"/>		0x00000000	Parameter	F:\img\rk3308\parameters-rk33...	
3	<input checked="" type="checkbox"/>		0x00003000	Trust	F:\img\rk3308\trust.img	
4	<input checked="" type="checkbox"/>		0x00002000	Uboot	F:\img\rk3308\uboot.img	
5	<input checked="" type="checkbox"/>		0x0000C800	Boot	F:\img\rk3308\zboot.img	
6	<input checked="" type="checkbox"/>		0x00023000	amp	F:\img\rk3308\amp.img	
7	<input checked="" type="checkbox"/>		0x00011000	rootfs	F:\img\rk3308\rootfs.img	...

8.2.1.2 RT-Thread

RT-Thread默认配置关闭eMMC Flash。如果用户需要在RT-Thread下使用eMMC Flash，按照以下方式修改：

Path: SDK/rtos/bsp/rockchip/rk3308-32

配置菜单配置：scons --menuconfig

8.2.1.2.1 RT-Thread配置

eMMC Flash配置：

```
CONFIG_RT_USING_SDIO=y
CONFIG_RT_SDCARD_MOUNT_POINT="/"
CONFIG_RT_USING_SDIO0=y

CONFIG_RT_USING_DMA=y
CONFIG_RT_USING_DMA_PL330=y
CONFIG_RT_USING_DMA0=y
```

elm文件系统配置：

```
CONFIG_RT_USING_DFS=y
CONFIG_DFS_FILESYSTEMS_MAX=4
CONFIG_DFS_FILESYSTEM_TYPES_MAX=4
CONFIG_RT_USING_DFS_MNTTABLE=y
CONFIG_RT_USING_DFS_ELMFAT=y
CONFIG_RT_DFS_ELM_MAX_SECTOR_SIZE=4096
```

8.2.1.2.2 Flash分区表配置

File: SDK/device/rockchip/.chips/rk3308/parameters-rtos-amp.txt

```
... 0x00008000@0x00004000 (amp), 0x00800000@0x0000c000 (rootfs), -  
@0x0080c000 (userdata:grow)
```

Flash分区表中的rootfs字段，配置了Flash中文件系统分区的地址与大小，单位为512Byte。

8.2.1.2.3 执行结果验证

固件升级首次启动后，需要先对文件系统格式化，操作如下：

```
mkfs -t elm sd0
```

重新启动后，文件系统挂载成功。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!
```

8.3 SNOR

8.3.1 RK3308

8.3.1.1 RT-Thread

RT-Thread默认配置关闭SNOR Flash。如果用户需要在RT-Thread下使用SNOR Flash，按照以下方式修改：

Path: SDK/rtos/bsp/rockchip/rk3308-32

配置菜单配置：scons --menuconfig

8.3.1.1.1 RT-Thread配置

SNOR Flash配置：

```
CONFIG_RT_USING_MTD_NOR=y  
CONFIG_RT_USING_SNOR=y  
CONFIG_RT_USING_SNOR_SFC_HOST=y
```

elm文件系统配置：

```
CONFIG_RT_USING_DFS=y
CONFIG_DFS_FILESYSTEMS_MAX=4
CONFIG_DFS_FILESYSTEM_TYPES_MAX=4
CONFIG_RT_USING_DFS_MNTTABLE=y
CONFIG_RT_USING_DFS_ELMFAT=y
CONFIG_RT_DFS_ELM_MAX_SECTOR_SIZE=4096
```

8.3.1.1.2 Flash分区表配置

File: SDK/device/rockchip/.chips/rk3308/parameters-rtos-amp.txt

```
... 0x00002000@0x00004000 (amp), 0x00000800@0x00006000 (rootfs), -
@0x00006800 (userdata:grow)
```

Flash分区表中的rootfs字段，配置了Flash中文件系统分区的地址与大小，单位为512Byte。

8.3.1.1.3 执行结果验证

系统启动后，文件系统挂载成功。通过文件系统串口命令操作，验证文件系统功能：

```
## 第8章 查看目录
ls
Directory /:
demo.txt

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

8.4 GMAC

8.4.1 RK3308

8.4.1.1 RT-Thread

RT-Thread默认配置关闭GMAC。如果用户需要在RT-Thread下使用GMAC，按照以下方式修改：

Path: SDK/rtos/bsp/rockchip/rk3308-32

配置菜单配置：scons --menuconfig

8.4.1.1.1 RT-Thread配置

GMAC配置：


```
## 第8章 打开GMAC配置
CONFIG_RT_USING_GMAC=y
CONFIG_RT_USING_GMAC0=y

## 第8章 配置网关等信息
CONFIG_RT_LWIP_IPADDR="xx.xx.xx.xx"
CONFIG_RT_LWIP_GWADDR="xx.xx.xx.xx"
CONFIG_RT_LWIP_MSKADDR="255.255.255.0"

## 第8章 打开PING测试
CONFIG_RT_USING_NETUTILS=y
CONFIG_RT_NETUTILS_PING=y
```

8.4.1.1.2 执行结果验证

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
```

8.5 PCIE

9. 第10章 调试

9.1 串口调试

RK356X AMP SDK中主核默认用UART2M0打印，从核默认用UART4M1打印。

串口通信配置信息如下：

波特率：1500000

数据位：8

停止位：1

奇偶校验：none

流控：none

AMP系统启动后，以CPU1为例，能够看到如下u-boot打印：

```
AMP: Brought up cpu[100] with state 0x10, entry 0x01800000 ...OK
```

能够看到如下HAL打印：

```
*****
Hello RK3568 Bare-metal using RK_HAL!
      Rockchip Electronics Co.Ltd
      CPI_ID(1)
*****
[(1) 0.002.772] CPU(1) Initial OK!
```

如果CPU运行的是RTT系统则能够看到如下RTT打印：

```
\ | /
- RT -      Thread Operating System
/ | \      3.1.3 build Aug  24 2022
2006 - 2019 Copyright by rt-thread team
Hello RK3568 RT-Thread! CPU_ID(1)
```

9.2 OpenOCD调试

RK356X AMP SDK 可以支持 JTAG 调试。当遇到复杂问题并且串口调试无法满足需求时，可以使用 JTAG 进行单步运行、断点追踪、data watch 和 寄存器、memory dump 等。

RK356X AMP SDK 的 JTAG 调试相关资料包可通过以下链接获取。

```
https://redmine.rock-chips.com/documents/111
```

9.2.1 环境搭建说明

这里简单介绍 Windows 平台下的环境搭建说明。具体操作步骤及说明文档，请参考下载资料包下相关文档及说明。

- 安装 OpenOCD 开发环境

OpenOCD 开发包为 `openocd_eclipse-2020-09.zip`，用户获取到资料包后，将该压缩文件解压至指定目录。

- 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。

- 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。

- RK3568 开发环境配置

运行 `openocd_eclipse-2020-09.zip` 解压出的 `eclipse.exe` 文件，并参考

《Rockchip_Developer_Guide_GNU_MCU_Eclipse_OpenOCD_CN.pdf》文档进行配置。

这里对于部分配置简单说明，详细信息请参考以上文档：

- 首先参照《Rockchip_Developer_Guide_GNU_MCU_Eclipse_OpenOCD_CN.pdf》文档，创建目标芯片的配置项。
- 进入“Debug Configurations”配置项，打开“Debugger”标签页，在“Config options”项目中加入以下内容：

```
-r rk3568
-c "cpu0 aarch64_32"
```

- 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，参考如下：

```
/home/user/rk3568/hal/ # GCC 编译时的工程路径
D:\rk3568\hal          # 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，`/home/user/rk3568/hal/` 为解析符号表时需要的路径信息。
`D:\rk3568\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。

- 添加 elf 文件。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，如下：

```
GNU gdb (GDB) 11.1
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-w64-mingw32 --target=aarch64-
none-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
```

```
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".

# .....
```

在该窗口下通过以下命令分别加入 4 个 CPU 的 *.elf 文件，参考如下：

```
# .....
For help, type "help".
Type "apropos word" to search for commands related to "word".
# .....
add-symbol-file D:/rk3568/hal/project/rk3568/GCC/0_TestDemo.elf
add-symbol-file D:/rk3568/hal/project/rk3568/GCC/1_TestDemo.elf
add-symbol-file D:/rk3568/hal/project/rk3568/GCC/2_TestDemo.elf
add-symbol-file D:/rk3568/hal/project/rk3568/GCC/3_TestDemo.elf
```

9.3 Ozone调试

Ozone工具是一款常用的，功能强大，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实用功能。官方地址：[Ozone – The Performance Analyzer \(segger.com\)](https://www.segger.com/products/debug-probes/ozone/)。官方支持两种使用许可模式（商业使用许可和非商业使用许可），请按实际需求，选择合适的许可模式合法使用。

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`
2. 配置目标设备：在项目设置中配置目标设备和调试器选项。
3. 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:"
```

4. 配置调试会话：在调试会话设置中选择断点、观察窗口等调试选项。
5. 启动调试会话：点击Ozone的调试按钮启动调试会话。
6. 调试应用程序：使用Ozone的调试功能来单步执行代码、查看变量值等。
7. 分析问题：如果遇到问题，可以使用Ozone的调试工具和功能来分析问题并找到解决方案。

Ozone调试器功能非常强大，除了常见的代码追踪功能外，还能查看CPU寄存器、直接读写Memory、通过Call Stack查看函数调用关系、跟踪变量等操作。详细使用说明可以参考Ozone官方手册[Getting started with Ozone \(segger.com\)](https://www.segger.com/documentation/getting-started/)。

10. 第11章 演示

（需要提供演示固件）

10.1 RK3568

10.2 RK3308

10.3 RK3358

10.4 RK3562

11. 第12章 附录

11.1 术语

11.2 文档索引