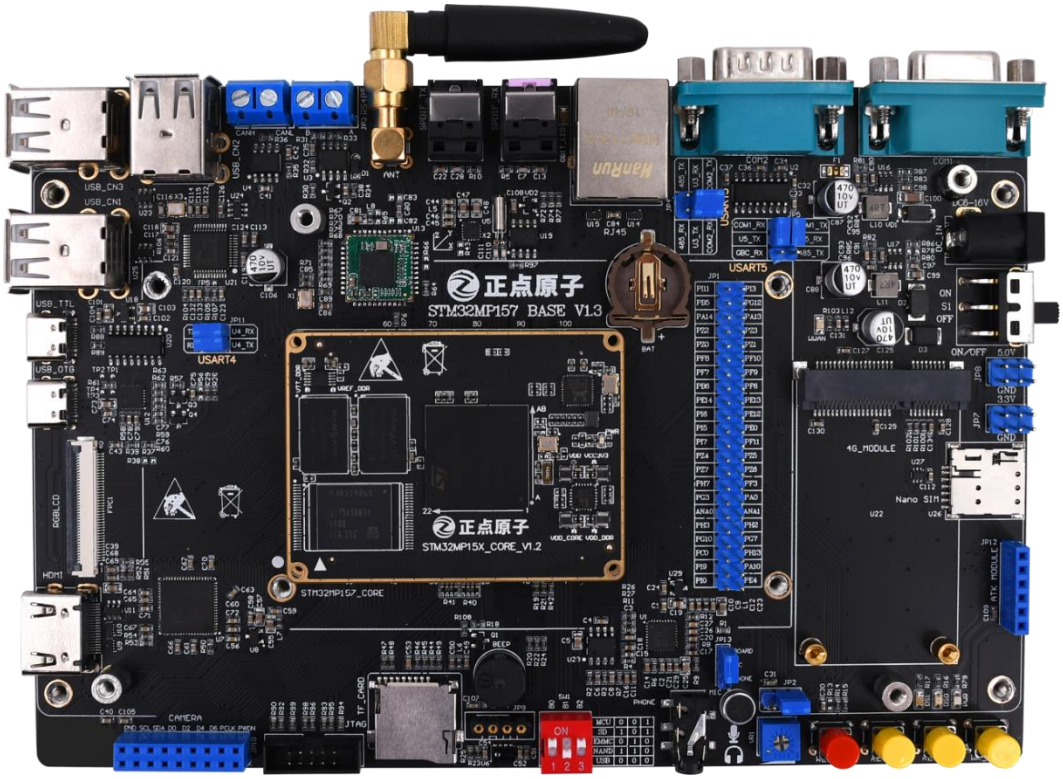
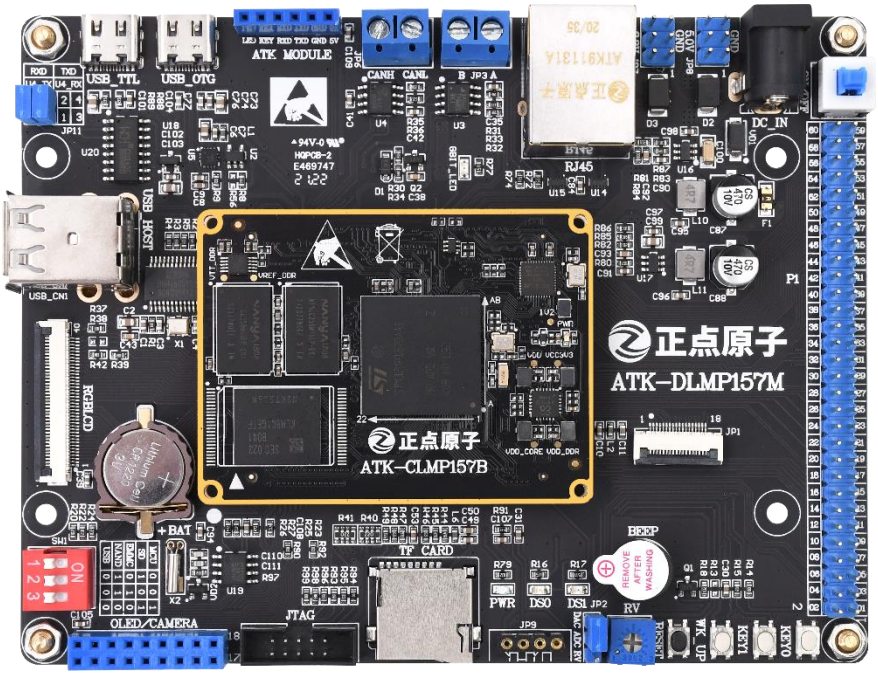
【正点原子】STM32MP1 异核通信(基于 CubeIDE) V1.1





正点原子

正点原子公司名称 ：广州市星翼电子科技有限公司

原子哥在线教学平台：www.yuanzige.com

开源电子网 / 论坛 ：http://www.openedv.com/forum.php

正点原子淘宝店铺 ：https://openedv.taobao.com

正点原子官方网站 ：www.alientek.com

正点原子 B 站视频 ：https://space.bilibili.com/394620890

电话：020-38271790 传真：020-36773971

请关注正点原子公众号，资料发布更新我们会通知。

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。

扫码关注正点原子公众号

扫码下载“原子哥”APP

文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿：	梁文聪	正点原子 Linux 团队	2021.08.05
V1.1	1、文档名字将《STM32MP1 异核通讯》修改为《STM32MP1 异核通信》 2、修改文档格式和描述，添加部分函数说明，添加部分例程	梁欢燕	正点原子 Linux 团队	2022.01.21

目录

前言

以一种有效的方式在一颗芯片中集成多个具有相同或者不同功能、不同结构的微处理器核，我们称之为多核异构，例如，一颗芯片里会有多个不同架构的 CPU 或者 DSP，这种就是多核异构处理器，此时芯片中属于不同架构的核，我们就称它们为“异核”，“异”，即不同，例如一颗芯片中的 DSP 和 ARM 属于不同的架构，它们是异核。

ST 推出的 STM32MP157 主控芯片有两个 Cortex-A7 内核和一个 Cortex-M4 内核，并集成 3D GPU，它属于多核异构，其中，Cortex-A7 内核可以运行 Linux 操作系统，Cortex-M4 内核可以运行裸机程序或者 RTOS，例如，可以运行 FreeRTOS、UCOS、RT-Thread 和 OneOS 等等这样的实时操作系统。

随着集成电路工艺技术的不断发展，工艺设计尺寸不断刷新更小的极限，加之性能需求的提高和功耗限制问题，采用大容量缓存、多核异构处理器已经成为一种更加有效降低成本和提高性能的嵌入式处理器系统设计方案。那么，不同的核在运行不同的操作系统时，不同的核心是如何启动运行的，它们之间又是如何进行异核通信的？

以 STM32MP157 为例，为解决前面提到的这两个问题，在裸机或者 RTOS 下（Cortex-M4 内核）会用到 OpenAMP 这个开发框架，在 Linux 下（Cortex-A7 内核）也会用到一些框架，这些框架都是服务于共享内存的，即 Cortex-A7 内核和 Cortex-M4 内核之间进行核间通信，本质上是通过共享内存来完成的。本篇的内容将围绕共享内存来进行阐述如何实现异核通信，我们以正点原子的 STM32MP157 开发板为开发平台，了解异核通信的实现原理并完成异核通信的实验。

现在，很多半导体公司在开发异核架构的芯片时，都会用到 OpenAMP 开发框架，例如 Xilinx、TI、NXP、ST 和 NORDIC（北欧）等等，这些厂商的一些异核架构芯片都有用到 OpenAMP 开发框架，那么，在理解了 STM32MP157 这个平台的异核通信实现方式以后，再切换到其它平台时，只要使用的是 OpenAMP 开发框架，其实现原理很多都是相似的。

关于异核通信部分的内容，如果要理解其实现的原理，我们建议您首先要掌握一些 Linux 操作系统相关的基础知识，如设备树相关的基础。如果您没有 Linux 操作系统基础，那也没有关系，只要按照实验步骤进行操作，最后也是可以完成实验的。

Linux 下的开发环境使用的是 Ubuntu，如果您要搭建开发环境，或者需要入门 Linux 驱动、Linux QT 和 Linux 应用，您可以参考正点原子的 Linux 系列教程：

《【正点原子】STM32MP1 嵌入式 Linux 驱动开发指南》

《【正点原子】STM32MP157 嵌入式 Qt 开发指南》

《【正点原子】STM32MP1 嵌入式 Linux C 应用编程指南》

《【正点原子】STM32MP157 快速体验》

如果您需要入门 STM32MP157 的 Cortex-M4 内核裸机驱动，您可以参考：

《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南》

《【正点原子】STM32MP1 M4 裸机 HAL 库开发指南》

如果您想入门 FreeRTOS、UCOS 和 OneOS 实时操作系统，您可以参考正点原子的 ST 系列的教程。以上提到的教程都可以在如下链接进行下载：

http://www.openedv.com/docs/boards/arm-linux/zdyzmp157.html

好，废话不多说，我们开启本篇的学习之旅。

第一章 配置 OpenAMP

STM32MP157的Cortex-A7可以运行Linux操作系统，Cortex-M4可以运行RTOS或者裸机程序，Cortex-A7和Cortex-M4进行异核通信，我们采用的是OpenAMP软件框架，OpenAMP是一个针对非对称多处理（AMP）系统开发应用程序所需的软件框架，它提供了所需的软件组件，用户通过调用这些组件中的API，就可以实现核间通信。本章，我们先不讲解OpenAMP，我们先来配置Cortex-M4的工程，主要开启IPCC和OpenAMP。

本章分为如下三个部分：

1.1 配置 M4 工程
1.2 导出 M4 工程
1.3 添加 LED1 代码
1.4 编译 M4 工程
1.5 导出 MDK 工程

1.1 配置 M4 工程

关于 STM32CubeIDE 和 STM32CubeMX 的详细使用，大家可以查看《【正点原子】STM32 MP1 M4 裸机 CubeIDE 开发指南》，下面我们会跳过 STM32CubeIDE 的一些细节，直接讲解怎么配置工程。本工程可参考参考开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch1\RPMsg_TEST。

1.1.1 配置 LED1 和 USART3

（1）在STM32CubeID上创建一个工程，工程名字为RPMsg_TEST。

（2）开发板底板的DS0灯被A7占用为心跳灯了，M4不能直接使用DS0灯，M4可以使用DS1灯，DS1灯占用的IO口是PF3，我们配置PF3为推挽输出、上拉模式，设置Pin Reserved为Cortex-M4，即给M4使用，如图1.1.1.1所示。

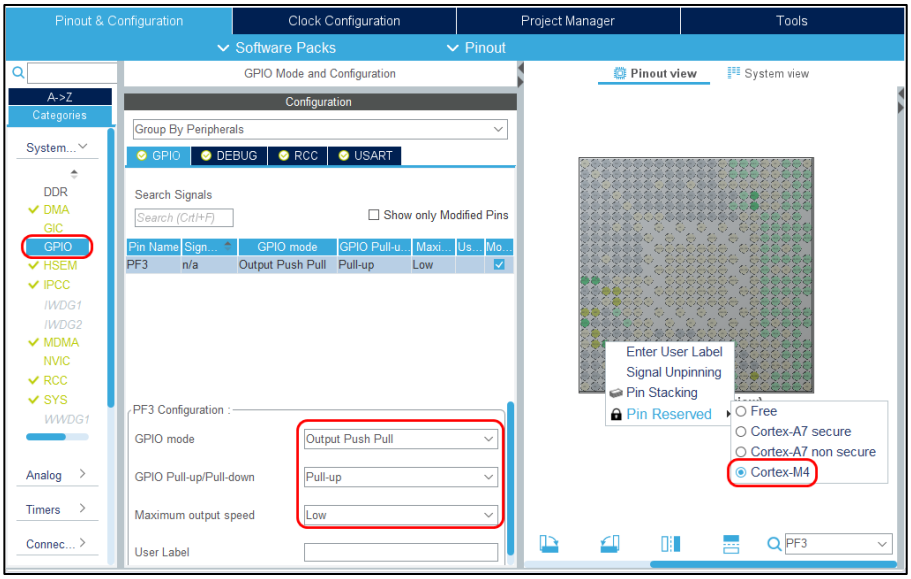


图1.1.1.1 配置DS1

（3）配置USART3给M4，M4通过USART3打印信息，如图1.1.1.2所示，开启M4的USART3，并配置工作模式为异步通信模式，参数采用默认的配置：

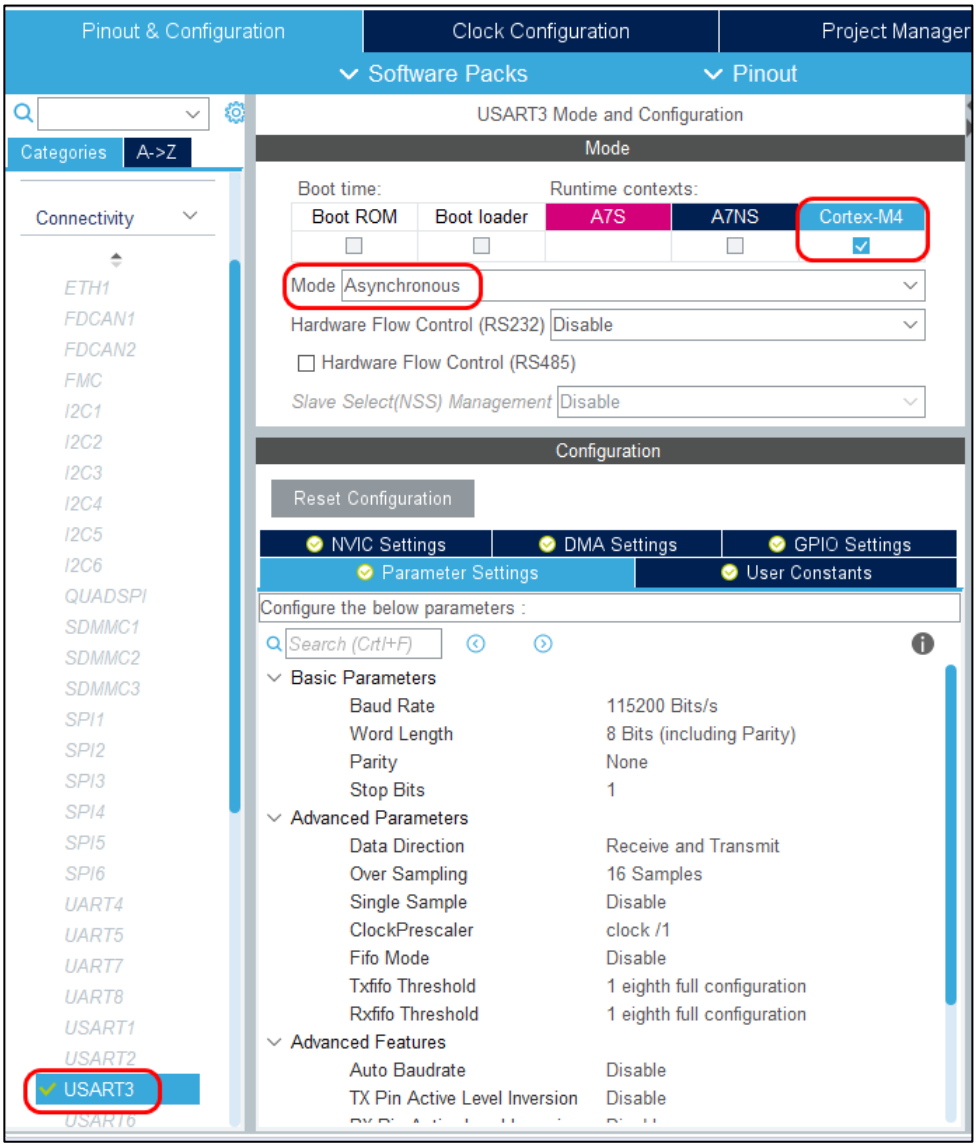


图1.1.1.2 开启M4的USART3

如图1.1.1.3所示，根据开发板的原理图，注意USART3的两个引脚为PD8和PD9：

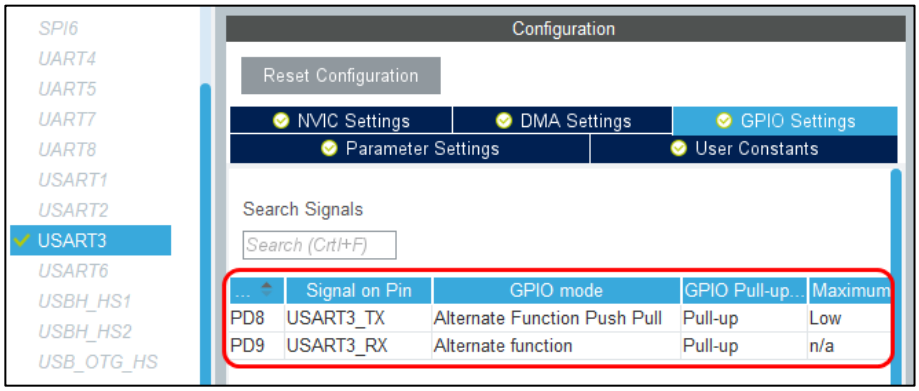


图1.1.1.3 USART3的两个引脚配置

1.1.2 配置 IPCC 和 HSEM

如下图所示，配置IPCC ，A7内核和M4内核是通过IPCC中断机制来通知对方在共享内存中有新的数据或者数据已经被处理，在A7这边已经默认开启了IPCC中断了，如下图1.1.2.1所示，M4这边也要开启IPCC中断，如图1.1.2.2所示。注意，M4这边一定要开启IPCC中断，否则M4就无法使能OpenAMP。

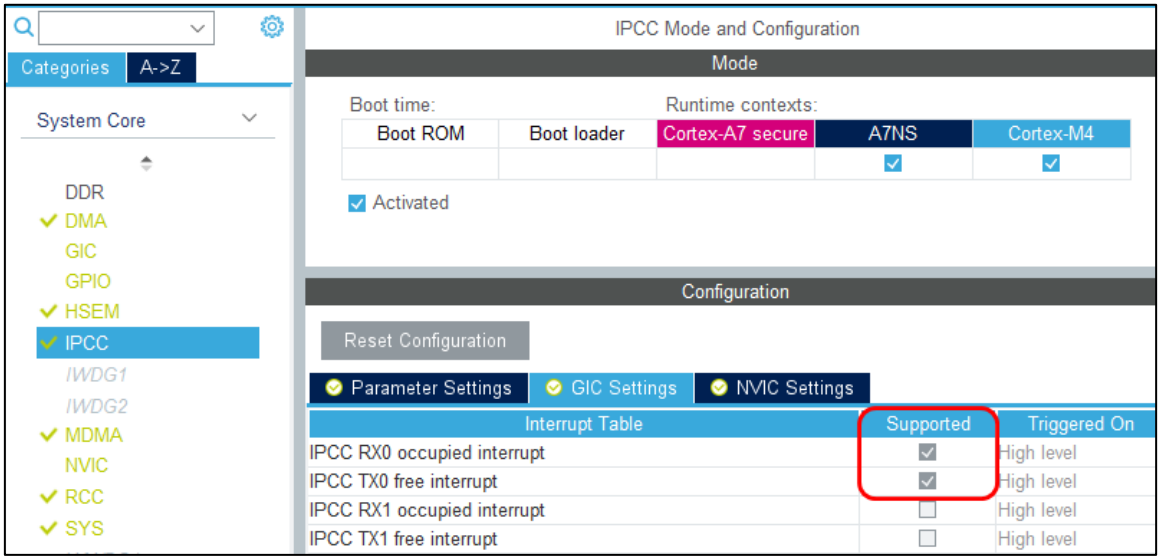


图1.1.2.1 A7默认开启了IPCC中断

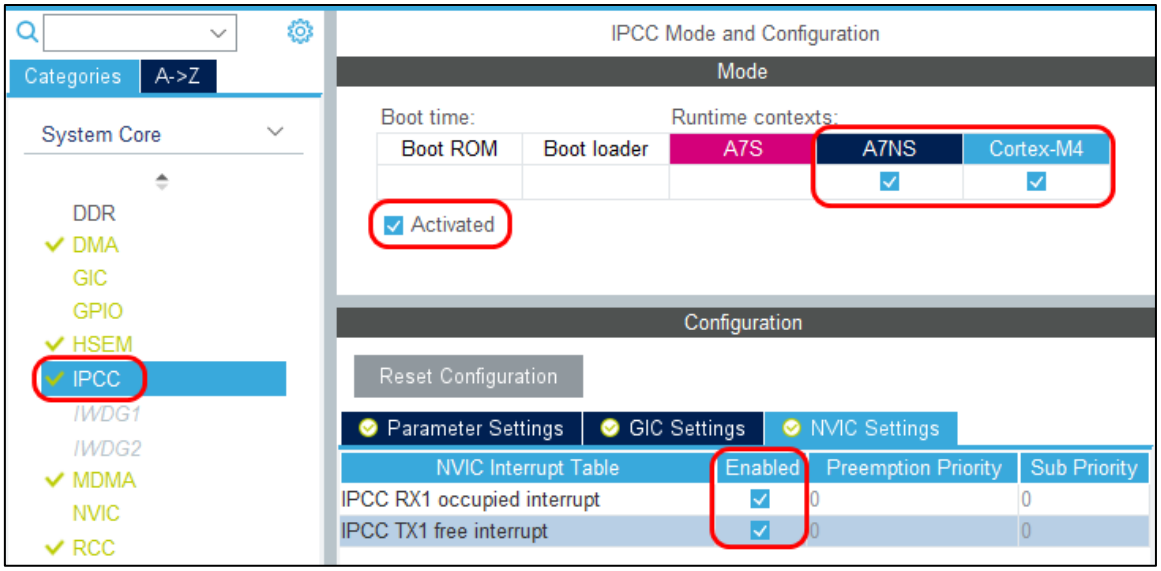


图 1.1.2.2 开启 M4 的 IPCC 中断

HSEM（硬件信号量）默认开启了，HSEM 是为了所有内核可以有序地访问公共资源，内核可以使用信号量来确保对外围设备的独占访问和信息交换。如图 1.1.2.3 所示：

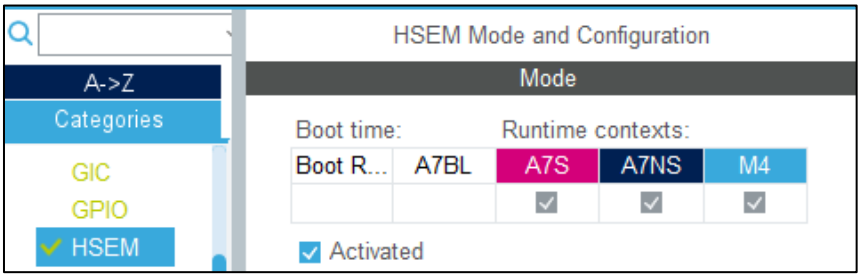


图 1.1.2.3 开启 HSEM

1.1.3 配置 OpenAMP

在异核通信开发中，M4 的工程要用到 OpenAMP 开发框架，所以我们还需要开启 OpenAMP，如下图 1.1.3.1 所示，进入中间件 Middleware→OPENAMP 处，开启 OpenAMP。开启 OpenAMP 以后，在参数一栏可以看到默认的配置，目前我使用的 STM32Cube 固件包为 V1.2.0 版本，所以提示 OpenAMP 的版本为 v2018.10 版本，通信模式提示的是远程通信模式，共享内存的起始地址是 0x10040000，共享的内存大小为 0x00008000，这段地址处于 SRAM3 中，每个 Vring 的缓冲区数量是 16 个，即 RPMsg Buffer 数量默认为 16 个。关于共享内存和 Vring，后面的章节会进行介绍。

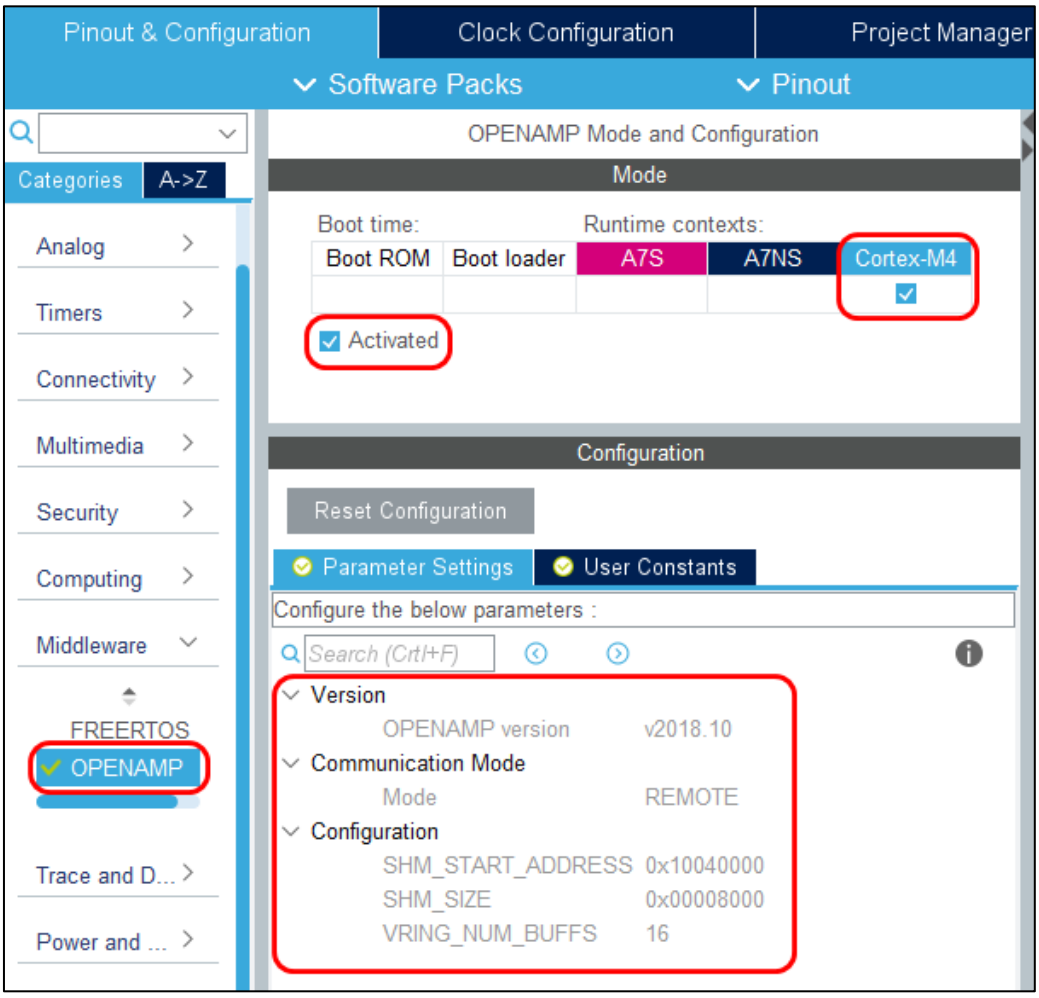


图 1.1.3.1 配置 OpenAMP

1.2 导出 M4 工程

关于时钟配置，可以采用默认的配置，也可以根据需要来配置，然后选择如下图1.2.1中的选项，让已经配置的每个外设生成独立的'.c/.h'文件：

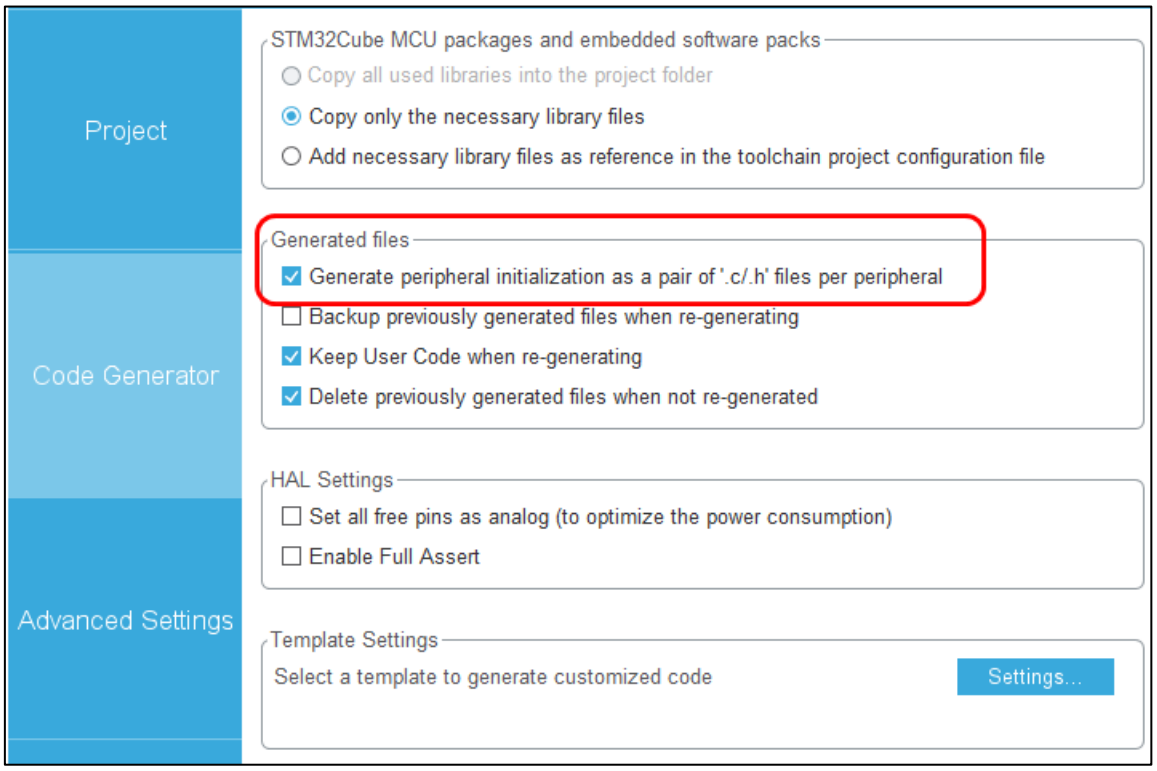


图1.2.1 选择生成独立文件

配置完毕后，保存配置，默认情况下系统会提示是否生成工程，直接点击下图 1.2.2 所示的“Yes”选择生成工程：

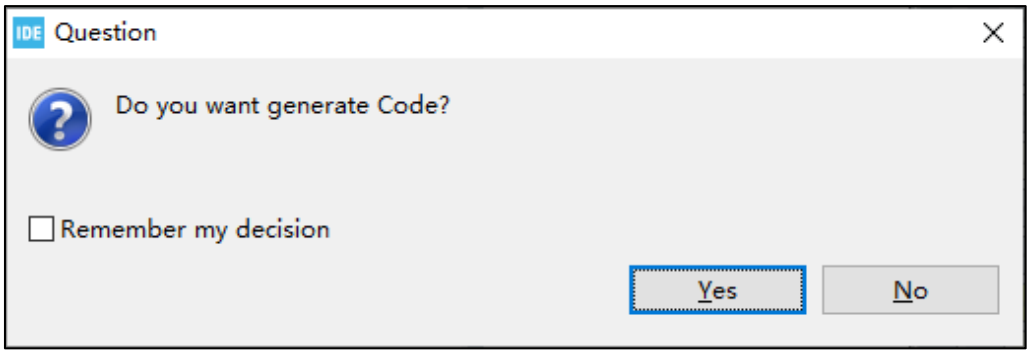


图 1.2.2 选择生成工程

如果没有提示生成工程，可以点击 Project→Generate Code 选择生成工程，如图 1.2.3 所示：

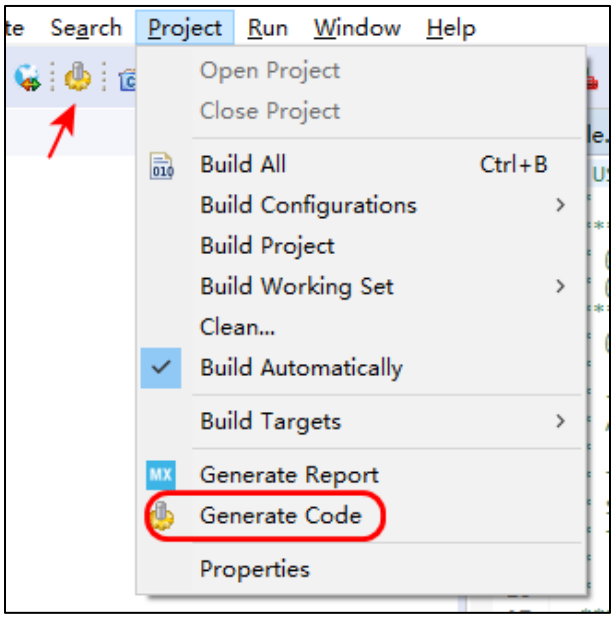


图 1.2.3 配置生成工程

如下图 1.2.4 所示，可以看到 OpenAMP 相关的库自动编译进工程了，生成的这个工程，我们也可以称为 M4 的工程：

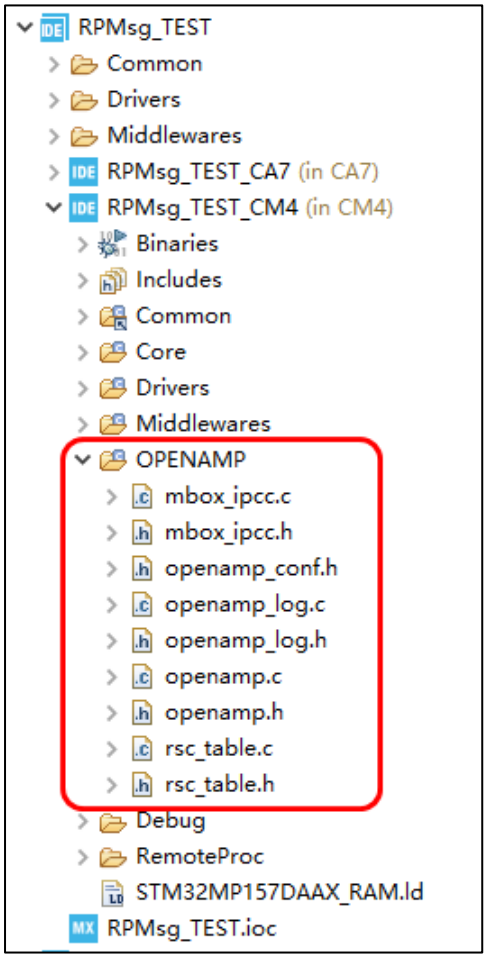


图 1.2.4 生成工程

后续的异核通信实验，我们就是围绕这个工程来进行的。

1.3 添加 LED1 代码

在main函数中的while循环中添加如下红色区域的代码，即循环点亮LED1灯，添加这段代码的目的就是通过观察LED1灯的现象来判断程序的运行情况。

```
int main(void)
{
  HAL_Init();
  if(IS_ENGINEERING_BOOT_MODE())
  {
    SystemClock_Config();
  }
  MX_IPCC_Init();
  MX_OPENAMP_Init(RPMSG_REMOTE, NULL);
  MX_GPIO_Init();
  MX_USART3_UART_Init();
  while (1)
  {
    HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_3);
    HAL_Delay(500);
    HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_3);
    HAL_Delay(500);
  }
}
```

1.4 编译 M4 工程

直接选择 M4 工程，然后选择编译工程，如图 1.4.1 所示：

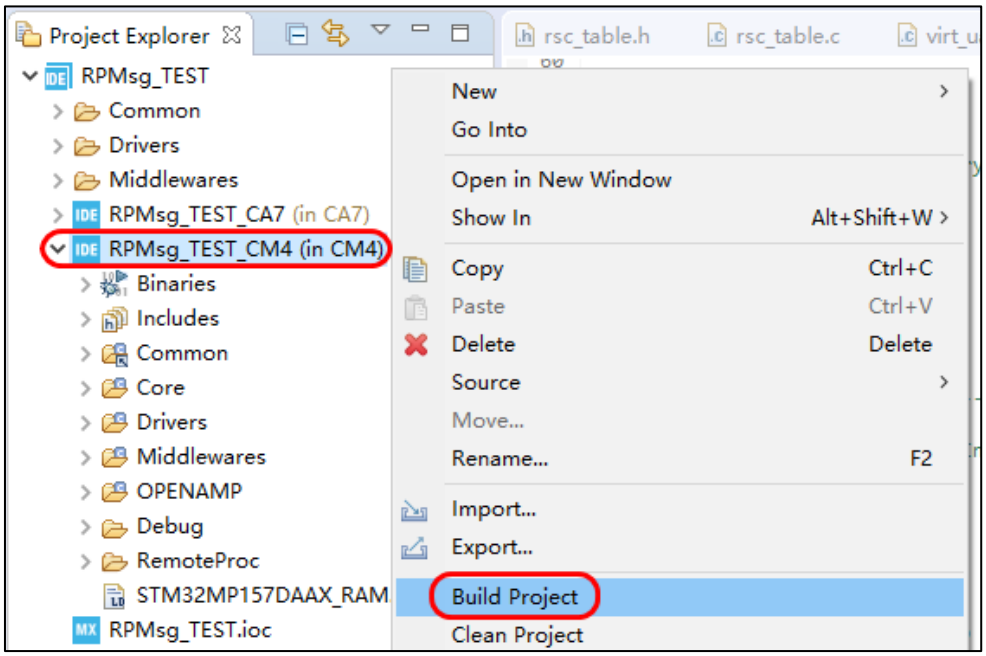


图 1.4.1 编译工程

工程编译不报错，并生成二进制文件 RPMsg_TEST_CM4.elf，如下图 1.4.2 所示，“.elf”格式的文件就是我们最终要用到的 M4 固件。

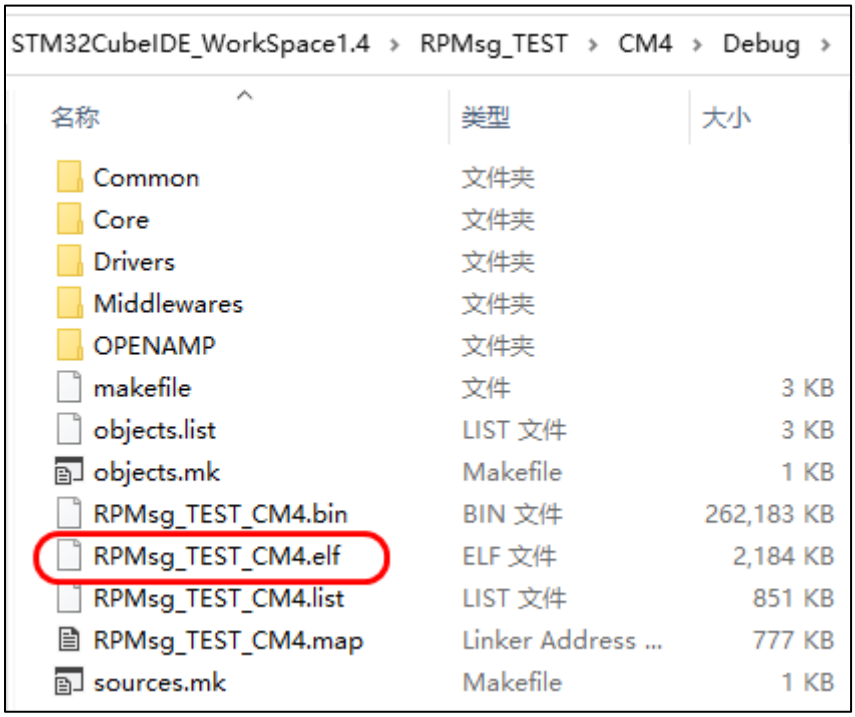


图 1.3.2 编译生成的文件

至此，工程生成成功，后续我们可以在此工程的基础上分析和添加代码，进行异核通信实验。

1.5 导出 MDK 工程

如果小伙伴想在 MDK 下进行开发，可以将以上配置好的工程导出为 MDK 工程，详细的操作方法可以参考《【正点原子】STM32MP1 M4 裸机 HAL 库开发指南》的第十章。下面我们介绍基于 STM32CubeIDE 如何导出 MDK 工程。

导出的 MDK 工程可参考参考开发板光盘 A-基础资料\ 01、程序源码\15、异核通信例程源码\MDK\ch1。

1.5.1 配置工程

接下来我们学习怎么导出一个 MDK 工程，在 STM32CubeIDE 上无法导出 MDK 工程，需要通过 STM32CubeMX 来导出，所以要提前安装 STM32CubeMX，具体可参考《【正点原子】STM32MP1 M4 裸机 HAL 库开发指南》的第十章。

首先打开 STM32CubeIDE 工程的目录，如下图 1.5.1.1 所示，找到 RPMsg_TEST.ioc 文件，这个就是 STM32CubeMX 工程文件。

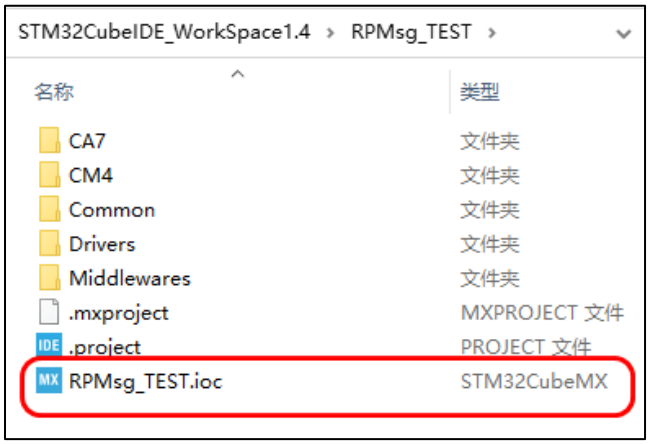


图 1.5.1.1 打开 STM32CubeMX 工程文件

打开后页面如下图 1.5.1.2 所示，初次安装和使用 STM32CubeMX 的小伙伴，在打开 STM32CubeMX 时可能需要在线下载 STM32Cube 固件包，耐心等待即可，或者可以手动添加固件包，具体做法在《【正点原子】STM32MP1 M4 裸机 HAL 库开发指南》的第十章有讲解。

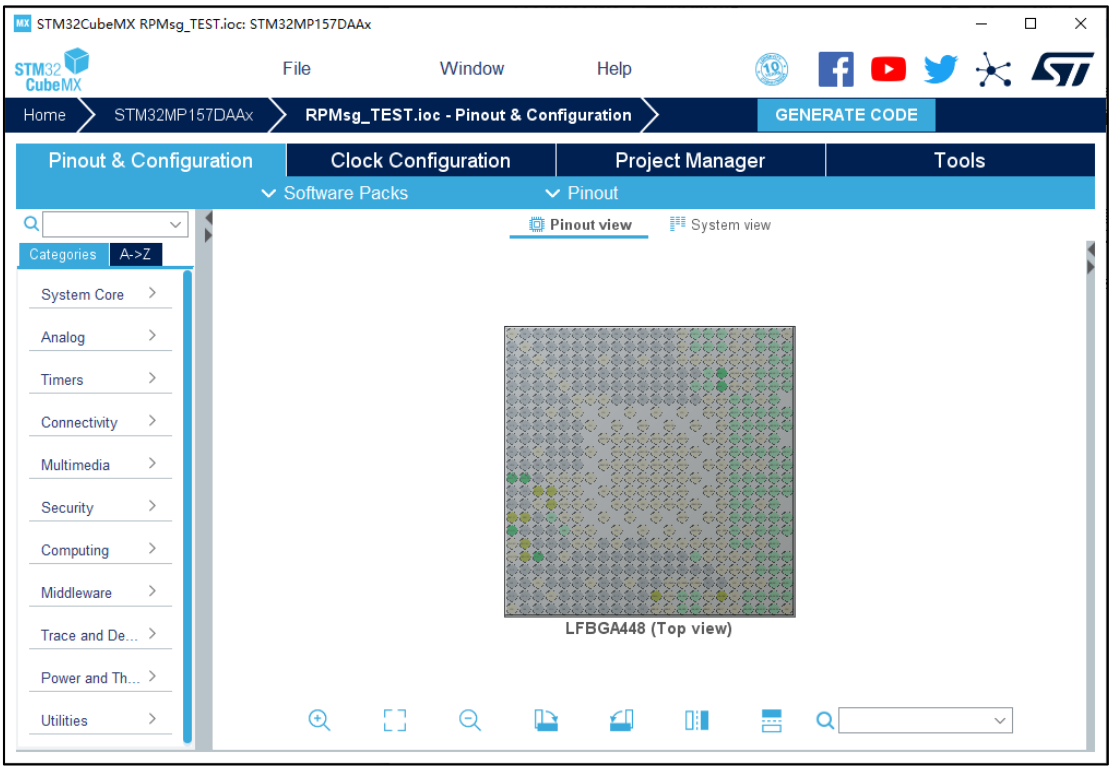


图 1.5.1.2 STM32CubeMX 工程页面

如下图 1.5.1.3 所示：

1）Toolchain/IDE 是工具链/集成开发环境，我们使用 Keil，因此选择 MDK-ARM，Min Version 选择 V5.27（最新，本教程的 MDK 是 V5.31 版本的，高版本的 MDK 可以打开低版本的 MDK 工程）；

2）勾选 Use Default Firmware Location，文本框里面的路径就是固件包的存储地址，我们使用默认地址即可。（这里如果有两个版本的固件包，它会默认使用最新版本的固件）。

3）工程保存路径可以采用默认路径，如要自动选择保存路径，请选择非中文路径，本例程默认保存在 E:\STM32CubeIDE_WorkSpace1.4\RPMsg_TEST 下。

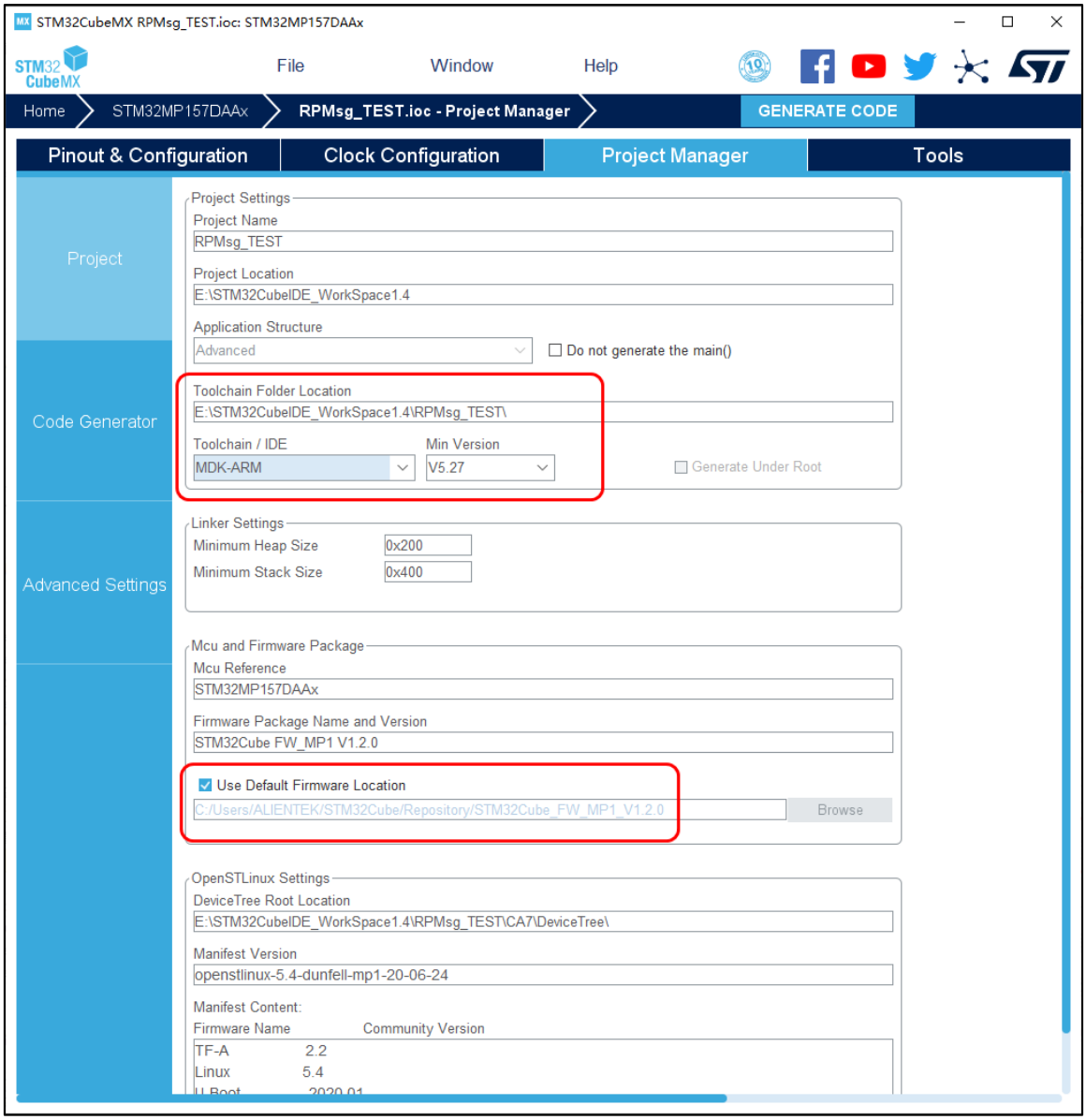
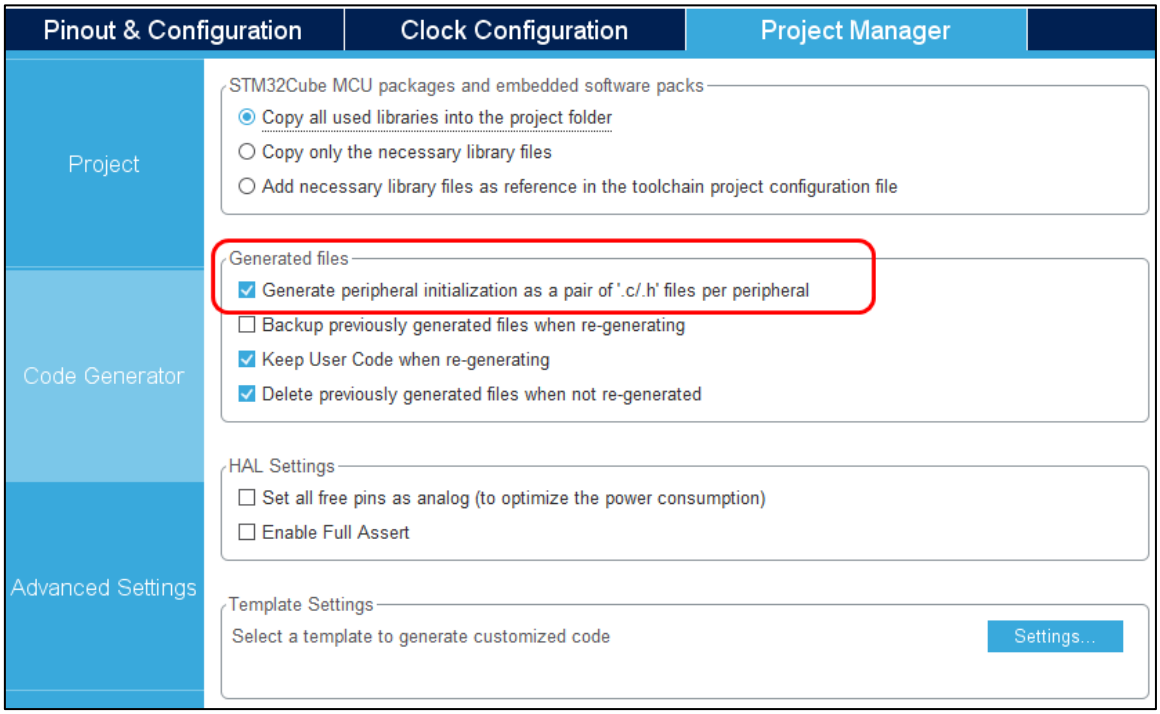


图 1.5.1.3 导出配置

打开 Project Manager-> Code Generator 选项，Generated files 生成文件选项，勾选 Generate peripheral initialization as a pair of '.c/.h'files per peripheral，勾选这个选项的话将会将每个外设单独生成一组.c、和.h 文件，这使得代码结构更加的清晰，如下图 1.5.1.4 所示。



1.5.1.4 代码生成器设置

先按下 Ctrl+S 保存配置，再点击蓝色按钮(SENERATE CODE)就可以生成工程，如下图 1.5.1.5 所示：

图 1.5.1.5 选择生成工程

如下图 1.5.1.6 所示，STM32CubeMX 会从之前我们关联的固件包中拷贝本工程需要使用的文件到要生成的工程目录中：

图 15.1.6 STM32CubeMX 从固件包中拷贝文件

如下图 1.5.1.7 所示，如果在弹出来的窗口中点击 Open Project 就可以打开 MDK 工程（已经按照本教程安装了 MDK），我们选择点击 Close 即可：

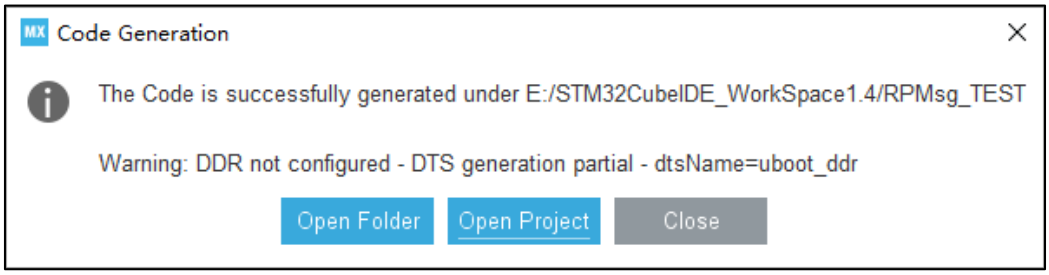


图 1.5.1.7 导出 MDK 工程成功

1.5.2 编译测试

在 E:\STM32CubeIDE_WorkSpace1.4\RPMsg_TEST 下可以看到多了一个 MDK-ARM 文件夹，这就是我们刚导出的 MDK 工程，如下图 1.5.2.1 所示，双击 RPMsg_TEST.uvprojx 即可打开工程：

图 1.5.2.1 打开 MDK 工程

打开工程以后，鼠标选中文件，可以看到文件的保存路径，如 main.c 文件和 STM32CubeIDE 使用的是同一个文件，如果在 STM32CubeIDE 上修改了 main.c 文件，那么 MDK 下的 main.c 文件也就被修改了，如下图 1.5.2.2 所示：

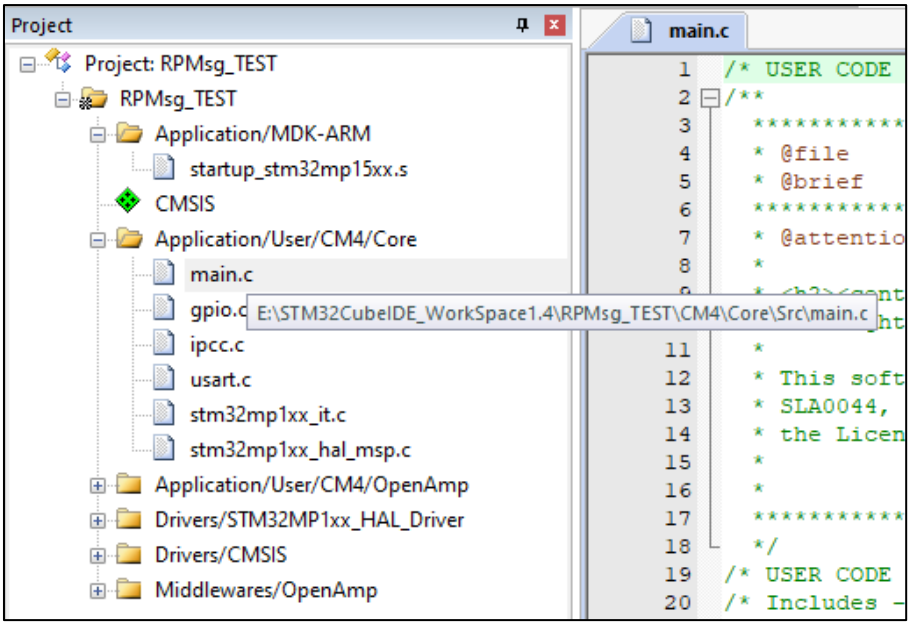


图 1.5.2.2 打开工程

可以尝试编译工程，一般都可以顺利编译通过，我们可以基于此工程完成后面的实验。

第二章STM32MP157 资源分配

当前，在大型 SOC（system on chip，系统芯片）设计当中，采用多核异构已经成为了一种趋势，在多核异构处理器的系统开发中往往需要面临一些问题，例如，系统中的资源是如何分配的，处理器启动的顺序是怎样的，还有，不同的核心之间是怎样进行通信的。本章，我们就以 STM32MP157 为例先来讲解系统中的资源是如何分配的。

本章分为如下三个部分：

2.1 STM32MP157 资源介绍

2.2 STM32MP157 内核外设资源分配

2.3 STM32MP157 存储资源分配

2.1 STM32MP157 资源介绍

正点原子 STM32MP157 开发板的主控芯片使用的是 STM32MP157DAA1，这是一款 448 脚、LFBGA 封装的芯片，它具有双核 32 位 Cortex-A7+单核 Cortex-M4，A7 主频为 800MHz，M4 主频为 209MHz。A7 内核包含 32KB 的 L1 I/D Cache，256KB 的 L2 Cache，支持 NEON 以及 TrustZone，M4 内核包含 FPU 单元。

STM32MP1 系列芯片支持 16/32bit 的 LPDDR2/LPDDR3-1066、DDR3/DDR3L-1066 内存，内存最大支持 1GB，另外 STM32MP157 内部包含 708KB 的 SRAM，同时，芯片内部有一个 3D GPU，支持 OpenGL ES2.0，RGB LCD 接口支持 24bit，RGB888 格式，分辨率最高支持 1366*768 60fps。其它详细的资源配置如下表 2.1.1：

STM32MP157DAA1 资源			
Cortex-A7	800M×2	32 位定时器	2 个
Cortex-M4	209M×1	电机控制定时器	2 个
3D GPU	1	16 位 ADC	2 个
SRAM	708KB	SPI	6 个
封装	LFBGA448	QSPI	2 个
通用 IO	176	I2S	3 个
16 位定时器	12 个	I2C	6 个
U(S)ART	4+4=8 个	FMC	1 个
CAN FD	2 个	USB OTG FS	1 个
SDIO	3 个	USB OTG HS	2 个
千 M 网络 MAC	1 个	RGB LCD	1 个
DSI	1 个	SAI	4 个
SPDIF RX	4 个	DFSDM	8 个
DCMI	1 个	TRNG	1 个

表 2.1.1 STM32MP157DAA1 内部资源表

Cortex-A7 内核可以运行 Linux 操作系统，Cortex-M4 内核可以运行裸机程序或者运行 RTOS，在这两个 Cortex-A7 内核中，其中一个 Cortex-A7 内核具有特殊的用途，可以用于开发和安全相关的系统或者应用，如 OP-TEE，所以我们就称该内核具有安全模式，另外一个 Cortex-A7

内核可当做普通的内核来使用，相对的，我们就称该内核具有非安全模式。Cortex-M4 内核可以运行裸机或者其它实时操作系统，如 FreeRTOS、UCOS、RT-Thread 和 OneOS 实时操作系统。

为了方便后面章节内容的讲解，在后面，我们就将 Cortex-A7 简称为 A7，将 Cortex-M4 简称为 M4。

2.2 STM32MP157 内核外设分配

2.2.1 STM32MP157 内核外设分配图

在多核异构处理器开发中需要面临的问题之一，就是资源是如何进行分配，在一颗 SOC 中会有很多不同的外设，例如 DDR，SRAM，视频类的输入输出接口，各种高速的外设接口（如 USB、PCIE 等），各种低速的外设接口（如 UART、I2C、I2S 和 SPI 等），还有各种内置的模块等。在这些资源中，哪个 CPU 可以访问哪些外设和模块呢，每个 CPU 可以占用哪些内存呢？下面，我们就先来了解 STM32MP157 的外设资源分配情况。

如下图 2.2.1.1，是 STM32MP157 的资源分配图，在图中的资源里，有红色区域的表示此外设可以被 A7 内核（安全模式）访问，有藏蓝色区域的表示此外设可以被 A7 内核（非安全模式）访问，有蓝色区域的表示此外设可以被 M4 内核访问。

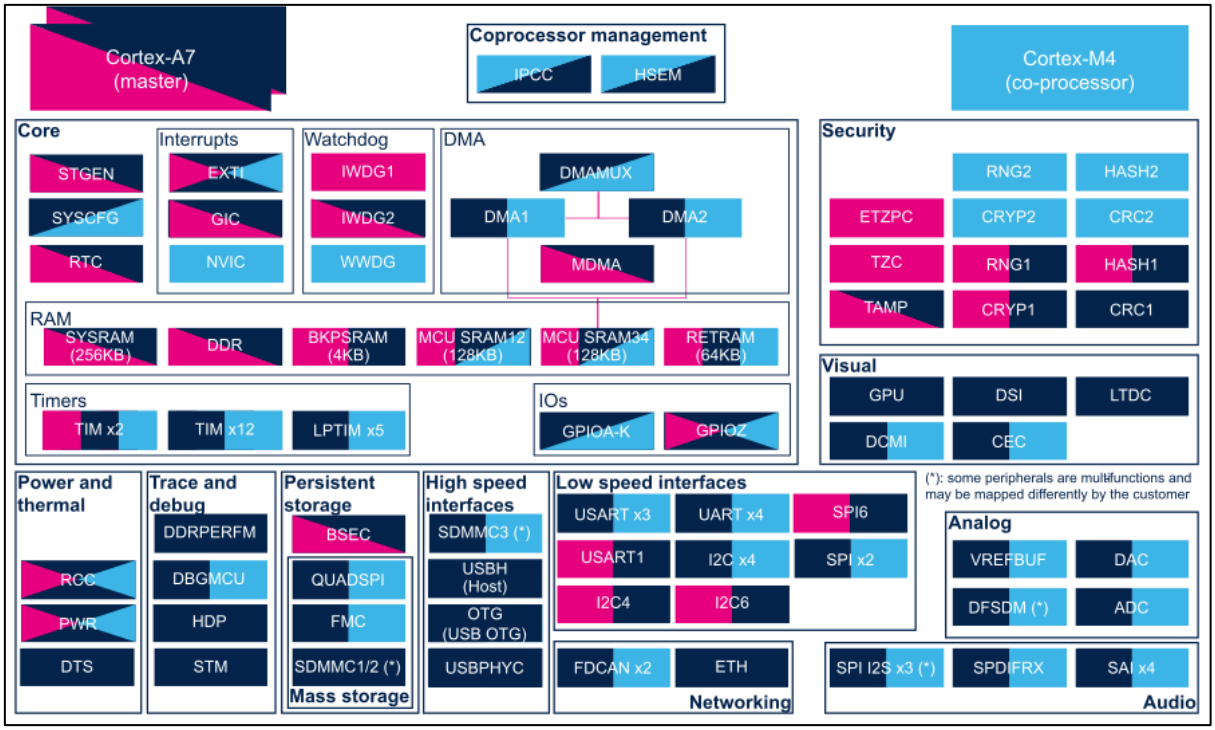


图 2.2.1.1 STM32MP157DAA1 内部资源分配图

从图中可以看到有的外设区域有多种颜色，例如 GPIOA-K 这几组 GPIO 有两种颜色，分别是藏蓝色和蓝色，表示此外设可以被 A7 内核（非安全模式）和 M4 内核访问，有的外设区域只有一种颜色，例如 NVIC 只有蓝色，说明 NVIC 只能 M4 内核访问，实际上，NVIC 是 M4 内核独有的外设，它是嵌套向量中断控制器。有多个颜色的外设，在这些颜色的划分中，有的是以竖线“|”来划分的，有的是以斜线“/”来划分的，它们的区别就是：

以竖线“|”来划分的外设，表示此外设在同一个时刻只能是单选的，例如 ADC 这个外设，它有两个颜色，分别是藏蓝色和蓝色，表示在同一个时刻，要么只能是 A7 内核（非安全模式）访问，要么只能是 M4 内核访问，不能在同一个时刻两个内核都去访问，否则 ADC 采集到的数

据就不会准确了，在这种情况下，一般是 M4 内核采集到的数据不准确，因为 A7 内核是主处理器，M4 内核是协处理器，关于这点我们后面的课程会介绍。以斜线“/”来划分的外设，表示此外设可以是共享的，例如前面提到的 GPIOA-K 这几组 GPIO，可以同时被 A7 内核（非安全模式）和 M4 内核共同访问。

以上图 2.2.1.1 的资源分配图可以在 ST 官方的 Wiki 上找到：

https://wiki.st.com/stm32mpu/wiki/STM32MP15_peripherals_overview

2.2.2 STM32MP157 内核外设分配表

根据上图 2.2.1.1 以及 ST 的参考资料（参考手册和数据手册），STM32MP157 的资源详细分配情况如下表 2.2.2.1 所示，表中‘☐’表示外设可以分配(☑)给此运行时上下文，‘✓’表示此外设只能用于某些运行时上下文，“单选”则表示某外设只能某个内核独享，“共享”，表示某个资源是 M4 和 A7 可以一起共享的。

域	外设	运行时分配				描述
		外设实例	A7 安全模式 (OP-TEE)	A7 非安全模式 (Linux)	M4 模式	
模拟	ADC	ADC		☐	☐	单选
模拟	DAC	DAC		☐	☐	单选
模拟	DFSDM	DFSDM		☐	☐	单选
模拟	VREFBUF	VREFBUF		☐		单选
音频	SAI	SAI1		☐	☐	单选
		SAI2		☐	☐	单选
		SAI3		☐	☐	单选
		SAI4		☐	☐	单选
音频	SPDIFRX	SPDIFRX		☐	☐	单选
协处理	IPCC	IPCC		☑	☑	共享
协处理	HSEM	HSEM	✓	✓	✓	
内核	RTC	RTC	✓	✓		
内核	STGEN	STGEN	✓	✓		
内核	SYSCFG	SYSCFG		✓	✓	
内核/DMA	DMA	DMA1		☐	☐	单选
内核/DMA		DMA2		☐	☐	单选
内核/DMA	DMAMUX	DMAMUX		☐	☐	可共享
内核/DMA	MDMA	MDMA	☐	☐		可共享
内核/中断	EXTI	EXTI		☐	☐	可共享
内核/中断	GIC	GIC	✓	✓		
内核/中断	NVIC	NVIC			✓	
内核/IO	GPIO	GPIOA		☐	☐	可共享
		GPIOB		☐	☐	可共享
		GPIOC		☐	☐	可共享
		GPIOD		☐	☐	可共享
		GPIOE		☐	☐	可共享

		GPIOF		☐	☐	可共享
		GPIOG		☐	☐	可共享
		GPIOH		☐	☐	可共享
		GPIOI		☐	☐	可共享
		GPIOJ		☐	☐	可共享
		GPIOK		☐	☐	可共享
		GPIOZ	☐	☐	☐	可共享
内核/RAM	BKPSRAM	BKPSRAM	☐	☐		单选
内核/RAM	DDR 控制器	DDR	✓	✓		
内核/RAM	MCU SRAM	SRAM1	☐	☐	☐	可共享
		SRAM2	☐	☐	☐	可共享
		SRAM3	☐	☐	☐	可共享
		SRAM4	☐	☐	☐	可共享
内核/RAM	RETRAM	RETRAM	☐	☐	☐	单选
内核/RAM	SYSRAM	SYSRAM	☐	☐	☐	可共享
内核/定时器	LPTIM	LPTIM1		☐	☐	单选
		LPTIM2		☐	☐	单选
		LPTIM3		☐	☐	单选
		LPTIM4		☐	☐	单选
		LPTIM5		☐	☐	单选
内核/定时器	TIM	TIM1		☐	☐	单选
		TIM2		☐	☐	单选
		TIM3		☐	☐	单选
		TIM4		☐	☐	单选
		TIM5		☐	☐	单选
		TIM6		☐	☐	单选
		TIM7		☐	☐	单选
		TIM8		☐	☐	单选
		TIM12	☐	☐	☐	单选
		TIM13		☐	☐	单选
		TIM14		☐	☐	单选
		TIM15	☐	☐	☐	单选
		TIM16		☐	☐	单选
		TIM17		☐	☐	单选
内核/看门狗	IWDG	IWDG1	☐			
		IWDG2	☐	☐		共享
内核/看门狗	WWDG	WWDG			☐	
高速接口	OTG(USB OTG)	OTG(USB OTG)		☐		
高速接口	USBH(USB Host)	USBH(USB Host)		☐		
高速接口	USBPHYC(USB HS PHY 控制器)	USBPHYC(USB HS PHY 控制器)		☐		
低速接口	I2C	I2C1		☐	☐	单选

		I2C2		☐	☐	单选
		I2C3		☐	☐	单选
		I2C4	☐	☐		单选
		I2C5		☐	☐	单选
		I2C6	☐	☐		单选
低速接口	SPI	SPI2S1		☐	☐	单选
		SPI2S2		☐	☐	单选
		SPI2S3		☐	☐	单选
		SPI4		☐	☐	单选
		SPI5		☐	☐	单选
		SPI6	☐	☐		单选
低速接口	USART	USART1	☐	☐		单选
		USART2		☐	☐	单选
		USART3		☐	☐	单选
		UART4		☐	☐	单选
		UART5		☐	☐	单选
		USART6		☐	☐	单选
		UART7		☐	☐	单选
		UART8		☐	☐	单选
大容量存储	FMC	FMC		☐		
大容量存储	QUADSPI	QUADSPI		☐	☐	单选
大容量存储	SDMMC	SDMMC1		☐		
		SDMMC2		☐		
		SDMMC3		☐	☐	单选
网络	ETH	ETH		☐		单选
网络	FDCAN	FDCAN1		☐	☐	单选
		FDCAN2		☐	☐	单选
电源&热量	DTS	DTS		☐		
电源&热量	PWR	PWR	✓	✓	✓	
电源&热量	RCC	RCC	✓	✓	✓	
安全	BSEC	BSEC	✓	✓		
安全	CRC	CRC1		☐		
安全		CRC2			☐	
安全	CRYP	CRYP1	☐	☐		单选
安全		CRYP2			☐	
安全	ETZPC	ETZPC	✓	✓	✓	
安全	HAS	HASH1	☐	☐		单选
安全		HASH2			☐	
安全	RNG	RNG1	☐	☐		单选
安全		RNG2			☐	
安全	TZC	TZC	✓			
安全	TAMP	TAMP	✓	✓		

跟踪&调试	DBGMCU	DBGMCU				
跟踪&调试	HDP	HDP		□		
跟踪&调试	STM	STM		□		
视觉	CEC	CEC		□	□	单选
视觉	DCMI	DCMI		□	□	单选
视觉	DSI	DSI		□		
视觉	GPU	GPU		□		
视觉	LTDC	LTDC		□		

表 2.2.2.1 STM32MP1 A7 和 M4 外设资源分配

在以上的外设资源中，FMC 可以用来外扩 SRAM、SDRAM、NAND Flash 等，也可以通过 FMC 来连接 8080 接口的 MCU 屏幕。LTDC 可以用来连接 RGB 接口屏幕，配置 DMA2D，可以实现绚丽的 UI 界面设计。

从以上的外设资源分配来看，除了 DDR、RTC、STGEN、MDMA、GIC、IWDG、USB、DSI、GPU、FMC、ETH 和 LTDC 等资源 M4 不可访问之外，其它大部分资源 M4 是可以进行访问的。值得注意的是，M4 内核没有内部 Flash，不能保存 M4 的程序。而 DDR 内存只有 A7 可以访问，M4 不能够访问，M4 可以访问（A7 也可以访问）的是 RETRAM、SYSRAM 和 SRAM1~SRAM4，这是芯片内置的存储，M4 程序可以在其中运行，程序掉电后会丢失。

从以上的外设资源分配情况还可以看出，M4 一般访问的是低速接口，如 UART、SPI 和 I2C，而 A7 可以访问 ETH 和高速接口（如 USB）以及一些大功耗的外设，如 LTDC 接口上接 RGB 屏幕，在运行 UI 界面时需要的功耗是比较大的。M4 还可以访问电源控制模块（PWR），PWR 用于实现电源监控和电源管理，可使 CPU 系统进入睡眠/停止/待机模式而降低功耗，也可以唤醒系统。

在以上的外设资源中，HSEM 和 IPCC 这两个外设比较特殊，A7（非安全模式）和 M4 内核都可以访问这两个外设，这两个外设我们做如下简单介绍：

1. HSEM

HSEM 是硬件信号量，用于共享资源的管理，两个 CPU 可访问的所有外设受到信号量的保护，当某个 CPU 在访问某个共享的外设之前，应先获取相关的信号量，在使用完外设后再释放信号量。通过信号量，可防止共享资源访问冲突，所有内核可以有序地访问公共资源。打个比方，共享的资源就像是一条路，所有的内核就像是行人和车，信号量就像是交通的信号灯，这条路是人走还是车走，就需要信号灯来指示，这样交通就变得井然有序了。

2. IPCC

IPCC 是核间通信控制模块，可用于 A7 和 M4 实现核间通信，它可以提供中断信令，允许处理器以非阻塞的方式交换信息。IPCC 最多有 6 个双向通道，每个通道支持单工、半双工和全双工模式，通道的数据（即核间通信的数据）存储在共享的内存中。关于 IPCC 和共享内存，我们在后面的章节再进行介绍。

如果 A7 跑 Linux 操作系统，M4 跑裸机或者 RTOS 时，大家一定要根据外设资源的实际分配情况来对 A7 和 M4 做好分配，不然实验可能无法得到正常现象，或者出现系统崩溃的情况。

2.2.3 STM32MP157 的应用

下面，我们根据以上 STM32MP157 的资源分配情况来了解这款芯片的应用。

从 STM32MP157 的资源分配上看，通常 Linux 运行在 A7 上，在 Linux 下就具有完整的网络、UI 显示（包括 GPU）、内存管理和安全功能，那么，A7 可用于高性能的任务处理、智能网关、图形显示和人机交互界面（HMI）应用中。例如：

① 在白色家电、工业控制和医疗设备中，可用于图形化人机交互控制终端；

② 在家居和工业控制中，可用于家庭智能网关和工业网关；

③ 在其它设备或领域中，可用于图像处理或远程界面，例如，在办公自动化以及系统监控设备中，可用于智能打印机、二维码扫描枪、智能点钞机、语音设备、电话会议设备以及系统监控的物体/人脸识别设备等。

M4 通常可以运行 RTOS，可以运行特定的软件应用，如：

① 运行实时传感器处理程序，例如，在 M4 可以访问的 SPI 和 I2C 等这些外设上接一些传感器，实时采集数据；

② 系统管理，基于 PWR，M4 可以让设备在 Linux 处于待机状态时唤醒；

③ 系统监控，基于 PWR，当整个系统发生故障的时候，M4 能够重置或者恢复整个系统。

④ 马达驱动，M4 结合 A7 的性能，可以用于电机控制中，例如双电机控制方案。

随着工业控制、消费电子、智能家居和医疗保健等领域的互联越来越密切，就需要特定的嵌入式设计，其在实现实时控制的同时，还能够实现高性能应用控制和图像处理，针对这些需求，使用 STM32MP157 这样的多核异构处理器就很合适，此时，M4 可用于低功耗的实时控制，A7 可用于管理较高的处理负载和开发具有丰富人机界面（HMI）的复杂应用。目前，STM32MP157 已经广泛应用于 HMI（人机界面）设计、工业控制和 IoT 应用开发中。

2.3 STM32MP157 存储分配

在多核异构处理器的开发中，除了要关注外设资源怎么分配以外，我们还要关注存储资源怎么分配，特别是当存储资源有限的时候，该怎么分配、分配多大才能保证系统得以正常运行，本小节我们就来了解 STM32MP157 的存储资源分配情况。

2.3.1 内存映射关系

STM32MP157 是一款 32 位的 ARM 芯片，有 4GB 的存储空间，被划分为了 13 块，如下表 2.3.1.1 所示，不同的外设占用不同的范围。其中，数据字节是以小端模式存放在存储器中的，即数据的低字节保存在内存的低地址中，数据的高字节保存在内存的高地址中。

存储块	功能	地址范围
BOOT	BOOT ROM 区域	0x00000000~0x0FFFFFFF(256MB)
SRAMs	SRAM 区域	0x10000000~0x1FFFFFFF(256MB)
SYSRAM	SYSRAM 区域	0x20000000~0x2FFFFFFF(256MB)
RAM aliases	SRAMs 的别名区，作用同 SRAMs	0x30000000~0x3FFFFFFF(256MB)
Peripherals 1	外设内存区域 1	0x40000000~0x4FFFFFFF(256MB)
Peripherals 2	外设内存区域 2	0x50000000~0x5FFFFFFF(256MB)
FMC NOR	FMC 接口 NOR Flash 映射后的内存区域	0x60000000~0x6FFFFFFF(256MB)
QUADSPI	QUAD SPI Flash 映射后的内存区域	0x70000000~0x7FFFFFFF(256MB)
FMC NAND	FMC 接口 NAND Flash 映射后的内存区域	0x80000000~0x8FFFFFFF(256MB)
STM	STM 寄存器区域	0x90000000~0x9FFFFFFF(256MB)
CA7	Cortex-A7 内核区域	0xA0000000~0xBFFFFFF(512MB)
DDR	DDR 内存区域	0xC0000000~0xDFFFFFF(512MB)
DDR 扩展(仅仅 CA7)或调试	DDR 扩展或调试区域	0xE0000000~0xFFFFFFF(512MB)

表 2.3.1.1 存储块功能及地址范围

下面，我们来看 STM32MP157 的内存映射图，如下图 2.3.1.1 所示：

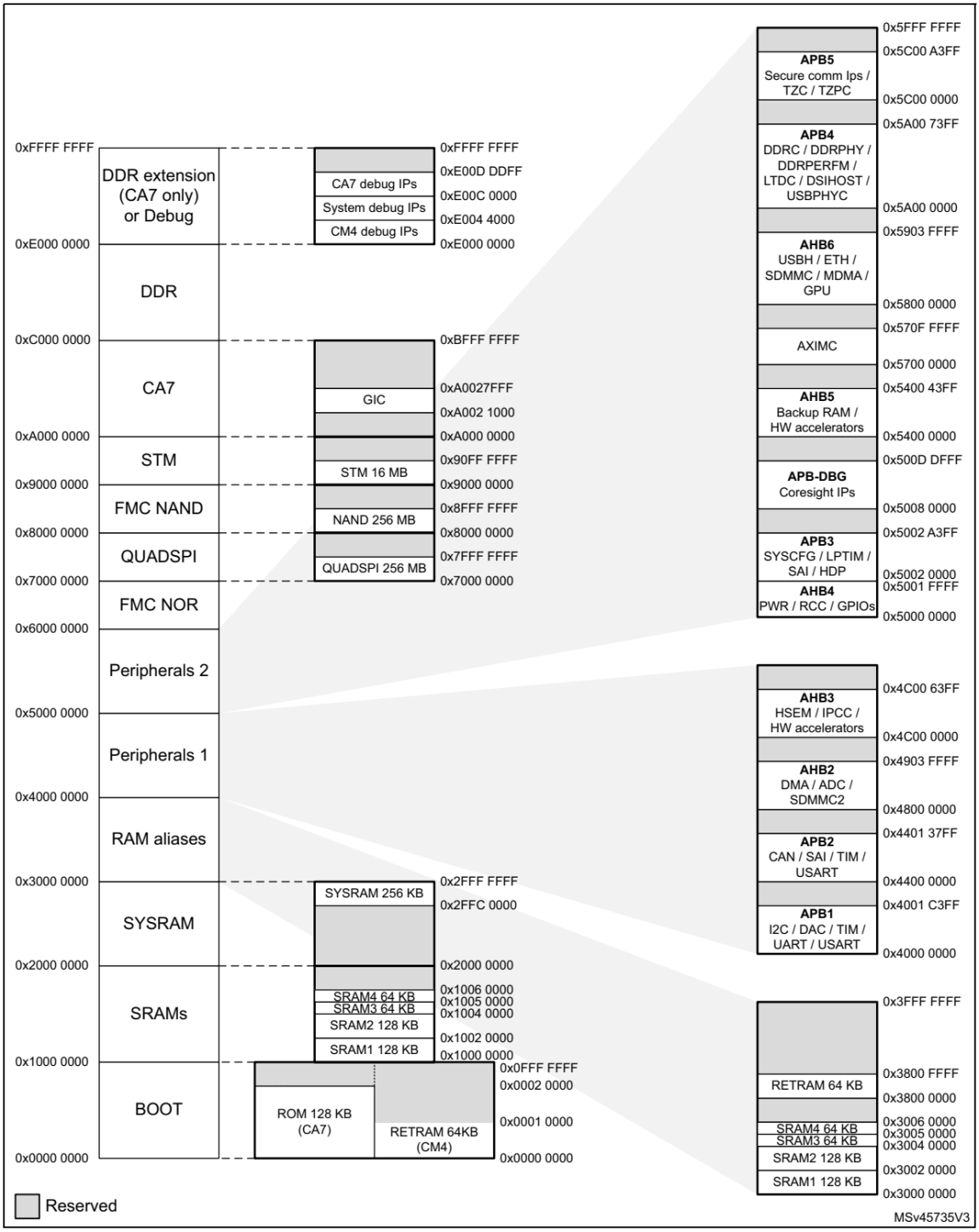


图 2.3.1.1 STM32MP157 的内存映射图

从上图的左侧部分可以看到，STM32MP157 的整个 4GB 内存空间被分为了 13 块，地址区间范围对应上表 2.3.1.1 的 13 块区域。上图的右侧部分则是某块存储区域的地址划分情况，例如 0x40000000~0x50000000 这个区域是外设内存区域 1，从图的右侧可以看出这段区域是被划分给了 APB1、APB2、AHB2 和 AHB3。图中的灰色区域部分是未使用的保留（Reserved）区域，是为了后续芯片型号升级而预留的存储块。

2.3.2 SRAM 存储区域

下面我们来了解几个比较重要的存储块，即 SRAM 存储区域，包括 BOOT、SRAMs、SYS RAM、RAM aliases 存储区域，它们和 M4 有关，如下图 2.3.2.1 所示：

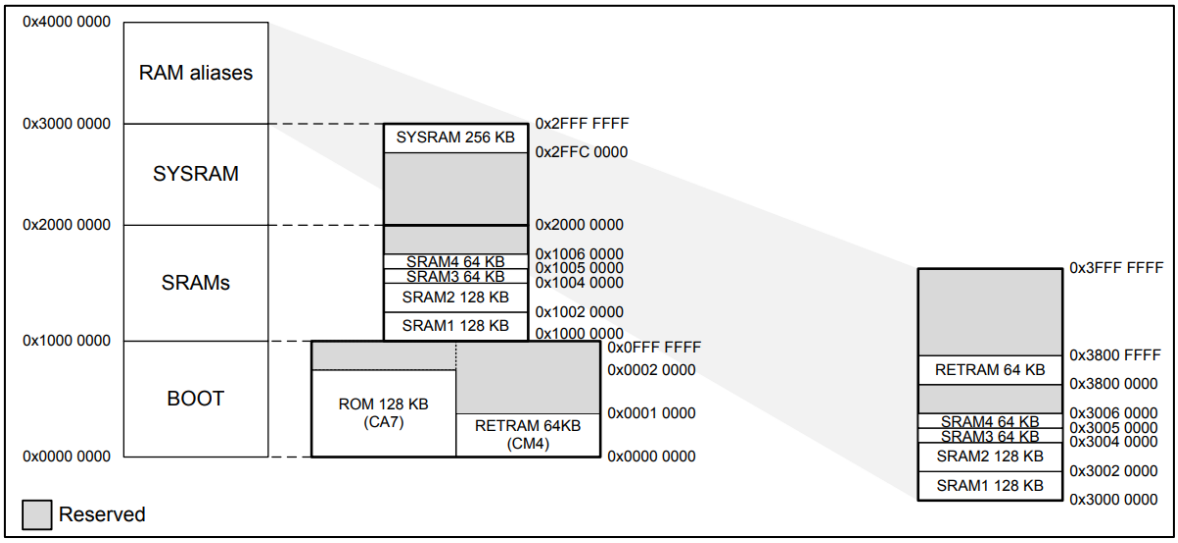


图 2.3.2.1 SRAM 存储区域

1. BOOT 存储区域

BOOT 存储区域从地址 0x00000000 开始，这是 STM32MP157 内部 BOOT ROM 区域，用于存储 ST 自己编写的启动代码，用户不可以使用该区域。BOOT 存储区域中的 RETRAM（64 KB）仅用于存放 M4 内核的中断向量表，注意，M4 内核的中断向量表是从地址 0x0000 0000 开始的。

2. SRAMs 存储区域

从内存映射图中看出，SRAMs 区域范围是 0x10000000~0x20000000，除了预留存储块，还有 4 块 SRAM：SRAM1~SRAM4，这四块 SRAM 的地址空间是连续的，地址范围为 0x10000000~0x1005FFFF，总大小为 384KB，每块的地址范围如下表 2.3.2.1 所示：

存储块	内存区域	大小
SRAM1	0x10000000~0x1001FFFF	128KB
SRAM2	0x10020000~0x1003FFFF	128KB
SRAM3	0x10040000~0x1004FFFF	64KB
SRAM4	0x10050000~0x1005FFFF	64KB

表 2.3.2.1 SRAMs 区域地址范围

M4 可以访问 BOOT 区域里的 RETRAM 和 SRAMs 区域里的 SRAM1~SRAM4，所以可以称它们为 MCURAM，其中，RETRAM 用于存放 M4 内核的中断向量表，而 SRAM1~SRAM4 可以用于存放 M4 的代码段和数据段，如果 A7 和 M4 之间要进行核间通信，还要存放资源表和双核之间通信的数据，关于核间通信时怎么分配 SRAM1~SRAM4，后面的章节会进行讲解。

如果 STM32MP157 不运行 A7 内核，只运行 M4 内核，那么这 SRAM1~SRAM4 的全部存储空间可以全部分配给 M4 使用，即 M4 独享这 384KB 的存储区域，RAM 和 ROM 全部都在这 384KB 内存范围内，注意的是，只能在线仿真，程序再掉电以后就丢失了。如果 STM32MP157 要运行 A7 和 M4 内核，且 A7 和 M4 之间又要进行核间通信，那么 M4 就不能独享这 384KB 的存储区域了，其中要划出一片区域用于核间通信时的共享内存，另外一小片区域用于存放资源表，资源表中包含了核间通信的一些配置，关于资源表，后面的章节再进行介绍。

下面，我们先来看 ST 给的参考设计，查看在 ST 给的参考设计中，MCURAM 如何使用了，如下图 2.3.2.2 是 ST 给的参考设计：

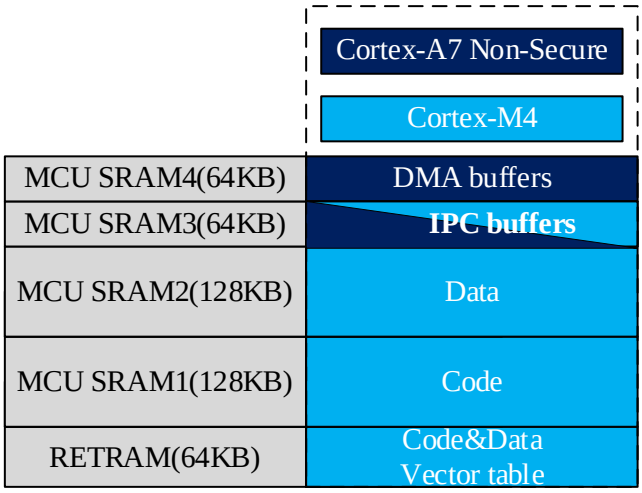


图2.3.2.2 SRAM1~SRAM4的内存分配

从上图ST的参考设计中，RETRAM和SRAM1以及SRAM2是单独给M4使用的，其中，RETRAM用于存放M4的中断向量表，这部分的区域是固定的，用户不能够使用，SRAM1用于存放M4的代码段，SRAM2用于存放M4的数据段，SRAM1和SRAM2加起来共256KB。

SRAM3是A7和M4共同使用的，这部分区域用于IPC缓冲区，即A7和M4的共享内存在SRAM3中。SRAM4被A7单独使用了，用于Linux下的DMA缓冲区。

以上SRAM1~SRAM4的分配情况只是ST给的参考设计，当然了，用户也可以根据自己的项目开发情况来重新规划这部分的区域，例如，如果A7下不需要使用SRAM4作为DMA 缓冲区了，可以将A7占用的SRAM4释放出来给M4使用。又例如，A7和M4的共享内存是在SRAM3中，也可以重新规划，将共享内存设置在SRAM1、SRAM2和SRAM4中。如果需要重新规划SRAM1~SRAM4，在Linux下就需要修改设备树，在M4的工程中就需要修改分散加载文件或者链接脚本，关于怎么修改，我们后面的课程会进行介绍。

3. RAM aliases存储区域

RAM aliases的地址范围是0x30000000~0x40000000，除了预留存储块，还有和SRAMs一样大小的SRAM1~SRAM4以及和BOOT区域一样大小的RETRAM。RAM aliases里的RETRAM、SRAM1~SRAM4我们称之为A7可以访问的RAM，实际上，RAM aliases里的SRAM1~SRAM4和RETRAM与SRAMs里的SRAM1~SRAM4和BOOT里的RETRAM对应的是同一个物理地址，即同一块物理地址映射到了两段内存区域中，一个区域是A7“可见”的，另一个区域是M4“可见”的，A7“可见”的区域和M4“可见”的区域本质上对应的是同一块物理内存。

我们举个例子来进行分析，例如有一块物理地址A映射到了RAM aliases里的RETRAM，同时也映射到了BOOT里的RETRAM，即物理地址A映射到了两个内存区域，一个区域是A7可以访问的，另外一个内存区域是M4可以访问的。又比如，物理地址B映射到RAM aliases里的SRAM1，也映射到了SRAMs里的SRAM1，依次类推（此处的物理地址A和B只是用于举例说明）。

这么一分析，RAM aliases中的aliases恰好有“别名”之意，那么RAM aliases的存储区域可以当做SRAMs的别名区，本质上它们对应的是同一段物理地址（保留区域除外），所以，如果要手动配置设备树和分散加载文件或者链接脚本时，设备树中RAM aliases的地址范围最好要和SRAMs里的地址范围对应，关于这点，我们后面配置设备树时会进行讲解。

第三章 异核通信框架

为了加深理解异核通信的实现过程，以及为了解释后面的内容出现的一些陌生的概念，本章我们就先来了解异核通信相关的框架，通过框架去了解异核通信的实现。异核通信的代码量是非常多的，在 Linux 内核下有大量的代码，在 M4 工程中又有大量的代码，如果想通过阅读这些代码来了解异核通信的实现方式，整个过程有点吃力，所以，我们直接从框架入手，如果理解了它的框架，那么我们再去分析代码的时候，就轻松多了。

本章分为如下几部分：

3.1 SMP 和 AMP 架构

3.2 IPCC 通信框架

3.3 OpenAMP 框架

3.4 驱动文件介绍

3.1 SMP 和 AMP 架构

1971 年， Intel 公司设计出一款 4 位的 4004 微处理器，它是第一款商用处理器，很快 Intel 又推出了 8 位的 8008 处理器和 16 位的 8086 处理器，那时候的 4004 芯片、8008 芯片和 8086 芯片上都只有一个核(单核 CPU)，随着需求的提高和功耗问题，慢慢的发现一个核不够用了，于是就在一个芯片上建造两个或者多个核，进而转向多核处理器发展了。多核 CPU 具有更高的计算密度和更强的并行处理能力，多核化趋势改变了 IT 计算的面貌。

3.1.1 同构和异构

从硬件的角度来分，多核处理器可以分为同构和异构，如下图 3.1.1.1 所示。

图 3.1.1.1 同构（左图）和异构（右图）

1. 同构

如果所有的 CPU 或核心的架构都一样，那么称为同构。例如，三星的 Exynos4210、飞思卡尔的 I.MX6D 以及 TI 的 OMAP4460，它们有两个架构相同的 Cortex-A9 内核，都属于同构。

2. 异构

如果所有 CPU 或核心的架构有不一样的，那么就称为异构。例如，ST 推出的 STM32MP157，有两个 Cortex-A7 核和一个 Cortex-M4 核，Xilinx 的 ZYNQ7000 系列，有两个 Cortex-A9 核和 FPGA，TI 的达芬奇系列 TMS320DM8127 有一个 DSP C674x 核和一个 Cortex-A8 核，这些处理器有不一样结构的核，所以都属于异构。

3.1.2 SMP 和 AMP

从软件的角度来分，多核处理器平台的操作系统体系有：SMP（Symmetric multiprocessing，对称多处理）结构、AMP（Asymmetric Multi-Processing，非对称多处理）结构和 BMP（bound multi-processing，边界多处理）结构。

1. 对称多处理结构(SMP)

SMP 结构是指只有一个操作系统（OS）实例运行在多个 CPU 上，一个 OS 同等的管理各个内核，为各个内核分配工作负载，系统中所有的内核平等地访问内存资源和外设资源。因为异构处理器的各个内核结构不同，如果一个 OS 去管理不同的内核，这种情况实现起来比较复杂，所以一般运行在 SMP 结构下的通常都是同构处理器。Windows、Linux 和 Vxworks 等多种操作系统都支持 SMP 结构。

如下图 3.1.2.1 所示，在 SMP 结构下，一个 OS 负责协调两个处理器，两个处理器共享内存，每个核心运行的应用程序（APP1 和 APP2）的地址是相同的，通过 MMU（Memory Management Unit，内存管理单元）把它们映射到主存的不同位置上。

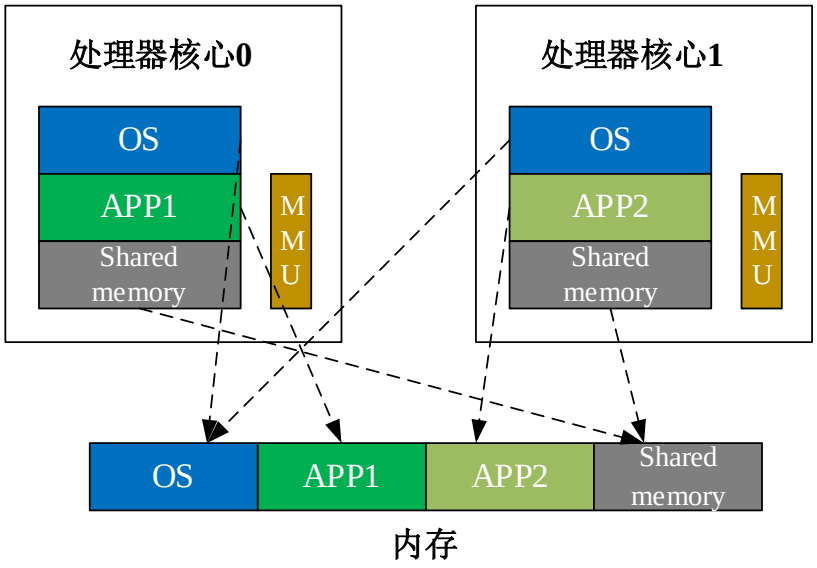


图 3.1.2.1 SMP 结构

2. 非对称多处理结构(AMP)

AMP 结构是指每个内核运行自己的 OS 或同一 OS 的独立实例，或者说不运行 OS，如运行裸机，每个内核有自己独立的内存空间，也可以和其它内核共享部分内存空间，每个核心相对独立地运行不同的任务，但是有一个核心为主要核心，它负责控制其它核心以及整个系统的运行，而其它核心负责“配合”主核心来完成特定的任务。这里，主核心我们就称为主处理器，其它核心我们就称为协处理器或者远程处理器。这种结构最大的特点在于各个操作系统都有本身独占的资源，其它资源由用户来指定多个系统共享或者专门分配给某一个系统来使用，系统之间可以通过共享的内存来完成通信。

如下图 3.1.2.2 是 STM32MP157 的资源简图，STM32MP157 的 Cortex-A7 内核可以运行 Linux 操作系统，Cortex-M4 内核可以运行裸机或者其它 RTOS（实时操作系统），RTOS 如 OneOS、FreeRTOS、RT-Thread 和 UCOS 等。Cortex-A7 和 Cortex-M4 都有自己独占的资源，也有共享的资源，这些资源由用户来分配，双核之间可通过共享内存来进行通信。

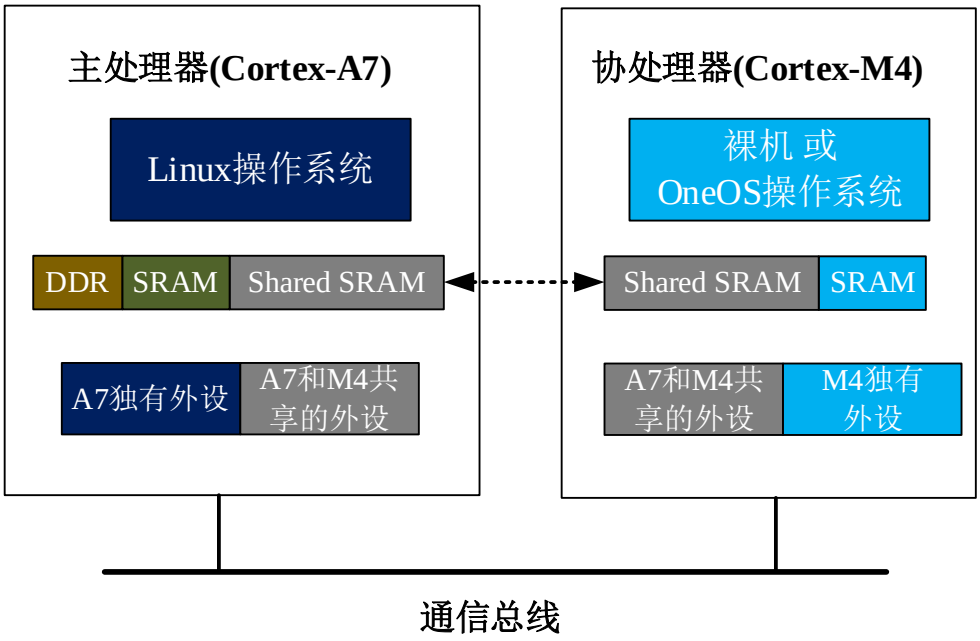


图 3.1.2.2 STM32MP157 资源简图

在 AMP 系统设计中，一般需要解决两个问题：

（1）生命周期管理（内核启动顺序）问题；

（2）内核间通信问题。

配置 AMP 系统最好的方法是使用一个既满足控制又满足通信要求的统一框架，而 OpenAMP 就是当前多核架构使用的最多的标准框架，许多芯片供应商都提供了 OpenAMP 的实现。基于 OpenAMP，生命周期管理是通过 Remoteproc 来实现的，内核间通信是通过 RPMsg 来管理的，关于两者的实现方式是我们后面章节重点介绍的内容。

3. 边界多处理结构(BMP)

BMP 和 SMP 类似，也是由一个操作系统同时管理所有 CPU 内核，但是开发者可以指定某个任务在某个核中执行，这里就不再详细介绍 BMP 结构了。

前面讲解的 AMP 和 SMP 有着明显的差别，但两者之间也有着联系，例如，在一个芯片上，可能多个架构相同的内核被配置为一个 SMP 子系统，而此时另外的内核跑的是其它的操作系统，从整体来看就是 AMP 结构了，从逻辑上来分，这个 SMP 子系统看起来像是一个单核，可以看做包含在这个大的 AMP 系统中。例如 STM32MP157 的两个 Cortex-A7 内核跑的是同一个 Linux 操作系统，这两个 Cortex-A7 内核就可以看做是一个 SMP 子系统，而 Cortex-M4 内核可以跑裸机或者 RTOS，那么 STM32MP157 这款芯片从整体上看就是 AMP 结构了。

3.2 IPCC 通信框架

当多个处理器位于不同的设备中时，这种情况我们称之为多个 SOC（片上系统），如果这两个设备的处理器要进行核间通信，可以通过物理串行链路来完成（其它的如 I2C、SPI、PCIE 等，此处只考虑串行链路的情况），如下图 3.2.1 所示，两个 SOC 可以通过串口进行核间通信，例如开发板和我们的 PC 属于两个设备，它们之间可以通过串口来互发数据。

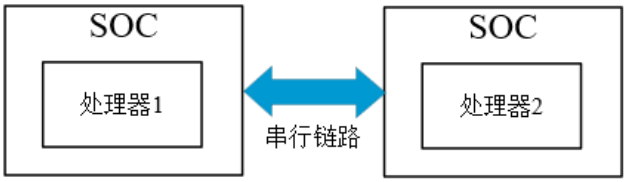


图 3.2.1 多 SOC 解决方案

如果是多核 SOC（一颗芯片中有多个处理器）呢，一般可以通过基于 IPC（进程间通信）的硬件邮箱（Mailbox）和可选的共享内存来完成，如下图 3.2.2 所示，在一个 SOC 上有两个处理器，我们称之为多处理器 SOC（或多核 SOC），这两个处理器通过硬件邮箱和共享内存来完成核间通信。关于硬件邮箱和共享内存，我们会在后面进行介绍，下面我们先来了解硬件 IPCC。

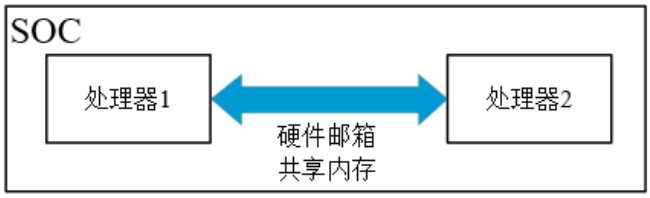


图 3.2.2 多处理器 SOC

3.2.1 IPCC 框架

IPC（Inter-Process Communication，进程间通信）是指两个进程的数据之间发生交互，它是通过处理器间通信控制器 IPCC（Inter-Process Communication controller ，IPC 控制器）来实现的，IPCC 属于硬件部分，用于在两个 CPU 之间进行信号交换，邮箱依赖于 IPCC，STM32MP157 的 IPCC 硬件模块有 6 个双向通道，每个通道分为两个子通道，6 个双向通道则共有 12 个子通道。如下图 3.2.1.1 所示是 A7 内核和 M4 内核通过 IPCC 进行通信的结构框图。

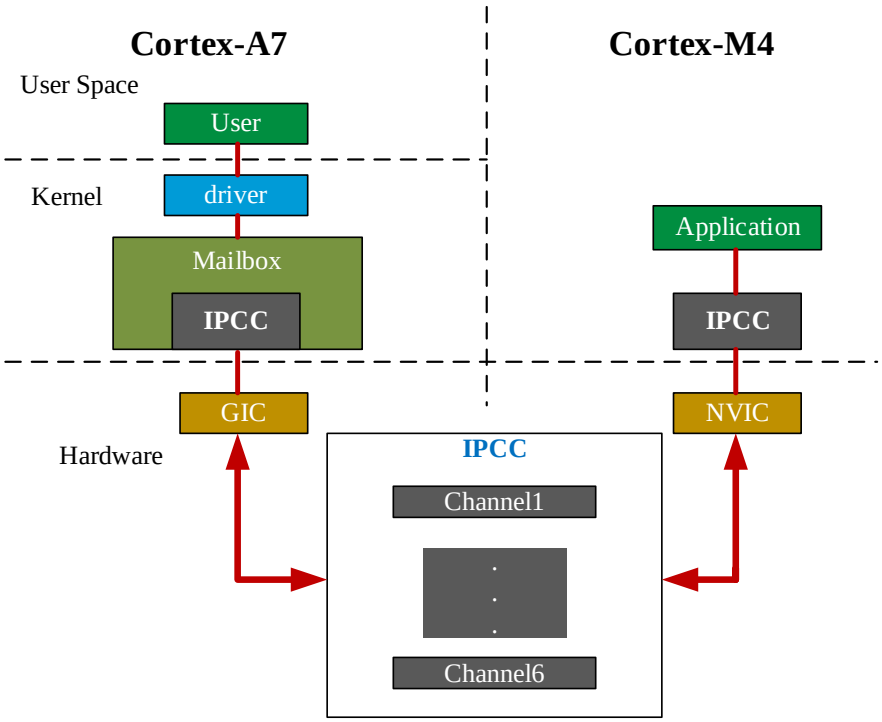


图 3.2.1.1 IPC 通信结构框图

对于上图 3.2.1.1，其通信过程为：

① 处理器之间交互的数据存放于共享内存中（上图 3.2.1.1 并未画出共享的内存），共享的内存对参与核间通信的 CPU 都是可见的。对于 STM32MP157 来说，在 ST 给的参考设计中，共享内存是在 SRAM3 中，众所周知，SRAM3 并不是 IPCC 的一部分，即共享的内存并不是 IPCC 的一部分，这点要注意。

② 当一个数据包被放入共享内存时，CPU 需要中断或“通知”另一个 CPU 有新的数据包在共享内存中要处理，另外一个 CPU 收到信号后就去处理了，这是使用 IPCC 硬件中断机制来完成的，A7 内核的中断控制器是 GIC，M4 内核的中断控制器是 NVIC。IPCC 的作用就好比放哨员，什么时候有数据可以接收，什么时候可以发送数据，整个过程是由 IPCC 来控制的。IPCC 只是提供了处理器之间信息交换的机制，它并不具有数据传输功能，也就是说，处理器之间要交互的数据并不是在 IPCC 中传输的，是在共享内存中传输的。

③ 从图中看到在 IPCC 处封装了邮箱（Mailbox）框架，邮箱框架主要用于通知 CPU 在共享内存中有数据要处理，即转发通知，我们在后面会进行介绍。

3.2.2 IPCC 通道

IPCC 的 6 个通信的工作模式可分为单工、半双工和全双工，下面，我们来看 IPCC 的这 6 个双向通道都有哪些用途：

- 从 CPU1 到 CPU2 的方向有 6 个通道（P1_TO_P2 子通道，P1 代表 CPU1，P2 代表 CPU2）；
- 从 CPU2 到 CPU1 的方向有 6 个通道（P2_TO_P1 子通道，P1 代表 CPU1，P2 代表 CPU2）；

如下表 3.2.2.1 所示，在 ST 官方配置的 IPCC 通信模型中，这 6 个通道被当做不同的软件框架来使用，其中：

① 通道 3 为单工模式，被当做 RemoteProc 框架，主处理器可通过该框架来加载协处理器的固件以及控制协处理器的生命周期；

② 通道 2 为全双工模式，被当做 RPMsg 框架，用于将 A7 的消息传输到 M4；

③ 通道 1 为全双工模式，被当做 RPMsg 框架，用于将 M4 的消息传输到 A7。

通道	模式	用法	软件客户端框架	
			Cortex-A7(非安全)	Cortex-M4
通道 1	全双工	从 Cortex–M4 到 Cortex–A7 的 RPMsg 传输： 1. Cortex-M4 使用该信道来指示一个消息可用 2. Cortex–A7 使用该信道以指示该消息被处理	RPMsg 框架	OpenAMP
通道 2	全双工	从 Cortex–A7 到 Cortex–M4 的 RPMsg 传输： 1. Cortex–A7 使用该信道来指示一个消息可用 2. Cortex-M4 使用该信道以指示该消息被处理	RPMsg 框架	OpenAMP
通道 3	单工	Cortex-M4 关闭请求	RemoteProc 框架	CprocSync cube utility
通道 4/5/6		未使用		

表3.2.2.1 IPCC的6个双向通道

可以看到，在ST配置好的框架中，RemoteProc软件框架解决了远程处理器生命周期的问题，RPMsg软件框架解决了核间通信的问题，这两个就是在多核处理器通信中关心的问题！在Linux内核下已经有ST移植好的RemoteProc和RPMsg软件框架，在M4使用的OpenAMP库中也有对应的RemoteProc和RPMsg软件框架，M4主要依赖OpenAMP库中已有的软件框架来实现和A7通信。

如下图3.2.2.1所示，STM32MP157的A7和M4实际上是通过通道1和通道2来传递数据的：

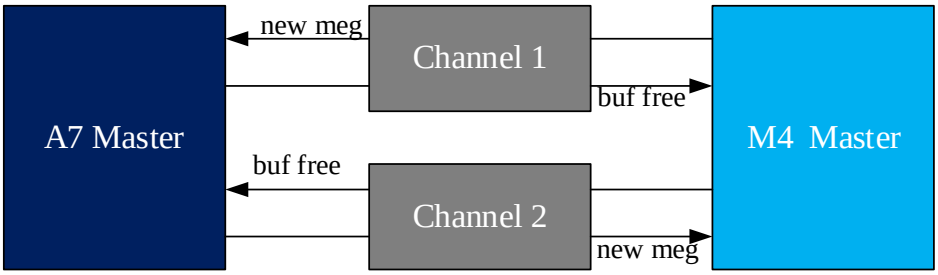


图3.2.2.1 Cortex-A7和Cortex-M4之间传递数据

3.2.3 IPCC 寄存器

下面，我们先了解如下表3.2.3.1所示和IPCC相关的8个寄存器，寄存器区被分成两个区域，每个处理器占用一个区域，防止读——写访问冲突，如下表中，前4个寄存器被处理器1占用，后4个寄存器被处理器2占用。IPCC为处理器提供专用的中断，每个处理器都有自己的状态屏蔽寄存器位、每个通道的设置或清除寄存器位。

寄存器	说明	位（n表示1~6）
IPCC_C1CR	使能/关闭 处理器1 TX通道空闲中断/RX通道占用中断	TXFIE、RXOIE
IPCC_C1MR	屏蔽/不屏蔽 处理器1 TX通道X空闲中断/RX通道X占用中断	CHnFM、CHnOM
IPCC_C1SCR	处理器1 TX/RX 通道X状态位 设置/清零	CHnS、CHnC
IPCC_C1TOC2SR	通道被占用/空闲	CHnF
IPCC_C2CR	使能/关闭 处理器2 TX通道空闲中断/RX通道占用中断	TXFIE、RXOIE
IPCC_C2MR	屏蔽/不屏蔽 处理器2 TX通道X空闲中断/RX通道X占用中断	CHnFM、CHnOM
IPCC_C2SCR	处理器2 TX/RX 通道X状态位 设置/清零	CHnS、CHnC
IPCC_C2TOC1SR	通道被占用/空闲	CHnF

表3.2.3.1 IPCC相关寄存器

对上表3.2.3.1中的寄存器说明如下：

1、IPCC_C1CR（IPCC处理器1控制寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	TXFIE
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	RXOIE
															rw

第17~31和第1~15位，保留位，必须保持在复位值。

第16位，TXFIE：处理器1发送（TX）通道空闲中断使能位，与 IPCC_C1TOC2SR关联，当该位为：

1：使能处理器1发送（TX）通道空闲中断；

0：关闭处理器1发送（TX）通道空闲中断。

第0位，RXOIE：处理器1接收（RX）通道占用中断使能位，与IPCC_C2TOC1SR关联，当该位为：

1：使能处理器1接收（RX）通道占用中断；

0：关闭处理器1接收（RX）通道占用中断。

2、IPCC_C1MR（IPCC处理器1屏蔽寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6FM	CH5FM	CH4FM	CH3FM	CH2FM	CH1FM
										rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6OM	CH5OM	CH4OM	CH3OM	CH2OM	CH1OM
										rw	rw	rw	rw	rw	rw

第22~31位和第6~15位，保留位，必须保持在复位值。

第16~21位，CH1FM~CH6FM，处理器1发送（TX）通道X空闲中断屏蔽位，与 IPCC_C1TOC2SR.CHxF关联，当该位为：

1：发送（TX）通道X空闲中断被屏蔽；

0：发送（TX）通道X空闲中断未被屏蔽。

第0~5位，CH1OM~ CH6OM，处理器1接收（RX）通道X占用中断屏蔽位，与IPCC_C2TOC1SR.CHxF相关，当该位为：

1：接收（RX）通道X占用中断被屏蔽；

0：接收（RX）通道X占用中断未被屏蔽。

3、IPCC_C1SCR（IPCC处理器1状态设置清除寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6S	CH5S	CH4S	CH3S	CH2S	CH1S
										rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6C	CH5C	CH4C	CH3C	CH2C	CH1C
										rw	rw	rw	rw	rw	rw

第22~31位和第6~15位，保留位，必须保持在复位值。

第16~21位，CH1S~CH6S，处理器1发送（TX）通道X状态设置位，与IPCC_C1TOC2SR.CHxF 相关，当设置该位：

1：处理器1发送（TX）通道X状态位被设置，即将CHnF设置为1，表示通道被占用；

0：无动作。

第0~5位，CH1C~CH6C，处理器1接收（RX）通道X状态清除位，与IPCC_C2TOC1SR.CHxF关联，当设置该位：

1：处理器1接收（RX）通道X状态位清零；

0：无动作。

4、IPCC_C1TOC2SR（处理器1到处理器2状态寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R	R	R	CH6F	CH5F	CH4F	CH3F	CH2F	CH1F

es	es	es	es	es	es	es	es	es	es						
										r	r	r	r	r	r

第6~31位，保留位，必须保持在复位值。

第0~5位，CH1F~CH6F，处理器1在屏蔽前向处理器2发送（TX）接收（RX）通道X状态标志位，当该位：

1：通道被占用，处理器2可以接收数据；

0：通道空闲，数据可由发送处理器1写入。

5、IPCC_C2CR（IPCC处理器2控制寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	TXFIE
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	RXOIE
															rw

第17~31和第1~15位，保留位，必须保持在复位值。

第16位，TXFIE：处理器2发送（TX）通道空闲中断使能位，与 IPCC_C2TOC1SR关联，当该位为：

1：使能处理器2发送（TX）通道空闲中断；

0：关闭处理器2发送（TX）通道空闲中断。

第0位，RXOIE：处理器2接收（RX）通道占用中断使能位，与IPCC_C1TOC2SR关联，当该位为：

1：使能处理器2接收（RX）通道占用中断；

0：关闭处理器2接收（RX）通道占用中断。

6、IPCC_C2MR（IPCC处理器2屏蔽寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6FM	CH5FM	CH4FM	CH3FM	CH2FM	CH1FM
										rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6OM	CH5OM	CH4OM	CH3OM	CH2OM	CH1OM
										rw	rw	rw	rw	rw	rw

第22~31位和第6~15位，保留位，必须保持在复位值。

第16~21位，CH1FM~CH6FM，处理器2发送（TX）通道X空闲中断屏蔽位，与 IPCC_C2TOC1SR.CHxF关联，当该位为：

1：发送（TX）通道X空闲中断被屏蔽；

0：发送（TX）通道X空闲中断未被屏蔽。

第0~5位，CH1OM~CH6OM，处理器2接收（RX）通道X占用中断屏蔽位，与IPCC_C1TOC2SR.CHxF相关，当该位为：

1：接收（RX）通道X占用中断被屏蔽；

0：接收（RX）通道X占用中断未被屏蔽。

7、IPCC_C2SCR（IPCC处理器2状态设置清除寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6S	CH5S	CH4S	CH3S	CH2S	CH1S
										rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6C	CH5C	CH4C	CH3C	CH2C	CH1C
										rw	rw	rw	rw	rw	rw

第22~31位和第6~15位，保留位，必须保持在复位值。

第16~21位，CH1S~CH6S，处理器2发送（TX）通道X状态设置位，与IPCC_C2TOC1SR.CHxF 相关，当设置该位：

1：处理器2发送（TX）通道X状态位被设置，即将CHnF设置为1，表示通道被占用；

0：无动作。

第0~5位，CH1C~CH6C，处理器2接收（RX）通道X状态清除位，与IPCC_C1TOC2SR.CHxF关联，当设置该位：

1：处理器2接收（RX）通道X状态位清零；

0：无动作。

8、IPCC_C2TOC1SR（处理器2到处理器1状态寄存器）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	CH6F	CH5F	CH4F	CH3F	CH2F	CH1F
										r	r	r	r	r	r

第6~31位，保留位，必须保持在复位值。

第0~5位，CH1F~CH6F，处理器2在屏蔽前向处理器1发送（TX）接收（RX）通道X状态标志位，当该位：

1：通道被占用，处理器1可以接收数据；

0：通道空闲，数据可由发送处理器2写入。

3.2.4 IPCC 功能描述

1、IPCC 概述

- IPCC 对处理器通信的控制，其实是通过中断的方式来实现的，称为核间中断（IPI），在每个处理器上都有两条中断线，分别是：
  ① 一条用于 RX 通道占用（发送处理器发布数据）；
  ② 一条用于TX通道空闲（接收处理器读取通信数据）。
- 每个通道都有中断屏蔽：
  ① 通道占用屏蔽（位 CHnOM）；
  ② 通道空闲屏蔽（位CHnFM）。
- 每个子通道都有两种工作模式：
  ① 单工模式（每个通道都有自己的通信数据存储位置）；
  ② 半双工模式（与双向通信数据信息存储位置关联的单个通道）。

- 通信的数据位于共享内存中（注意，共享内存不是 IPCC 的一部分），每次通信，IPCC 模块提供一个通道状态标志位 CHnF（n 可取值 1~6，表示对应的通道）：

① 当 CHnF=0 时，表示相关的通道是空闲的（也可以认为通信的数据已经被接收处理器读取了，此时通道处于空闲状态了），这个时候发送处理器可以占用这个通道用于发送数据；

② 当 CHnF=1 时，表示相关的通道已经被占用了（即通信数据已经被发送处理器发布了），这个时候接收处理器可以去访问该通道以读取数据。

- IPCC 为通道管理提供非阻塞信令机制：

① 消息可用性中断；

② 通道流控制（如生成一个 TX 通道空闲中断，则称为流开启，当通道被占用时，则称为流关闭）。

- 应用程序受益于基于非阻塞中断的消息交换和通道流控制来进行核间通信，每个子通道会有一个传输方向：

① 从 CPU1 发送并由 CPU2 接收；

② 或者从 CPU2 发送并由 CPU1 接收。

2、IPCC 通信模式

下面我们来看 IPCC 是如何协调主处理器和协处理器进行核间通信的，也就是如何控制处理器有序地访问共享内存。关于该部分，了解即可。

（1）IPCC 单工发送模式

如下图 3.2.4.1 所示为 IPCC 单工发送模式示意图。当通道状态标志为通道占用时（CHnF=1），这是因为接收方没有从先前的消息中释放通道，这个时候，通道空闲中断被取消屏蔽，然后等待 TX 空闲中断。一旦接收方释放了通道，就会产生通道空闲中断（流开启）。当产生通道空闲中断时，通道空闲中断被屏蔽，消息可以写入数据缓冲区。随后，通道状态标志被设置为已占用，这会触发接收端的通道占用中断。

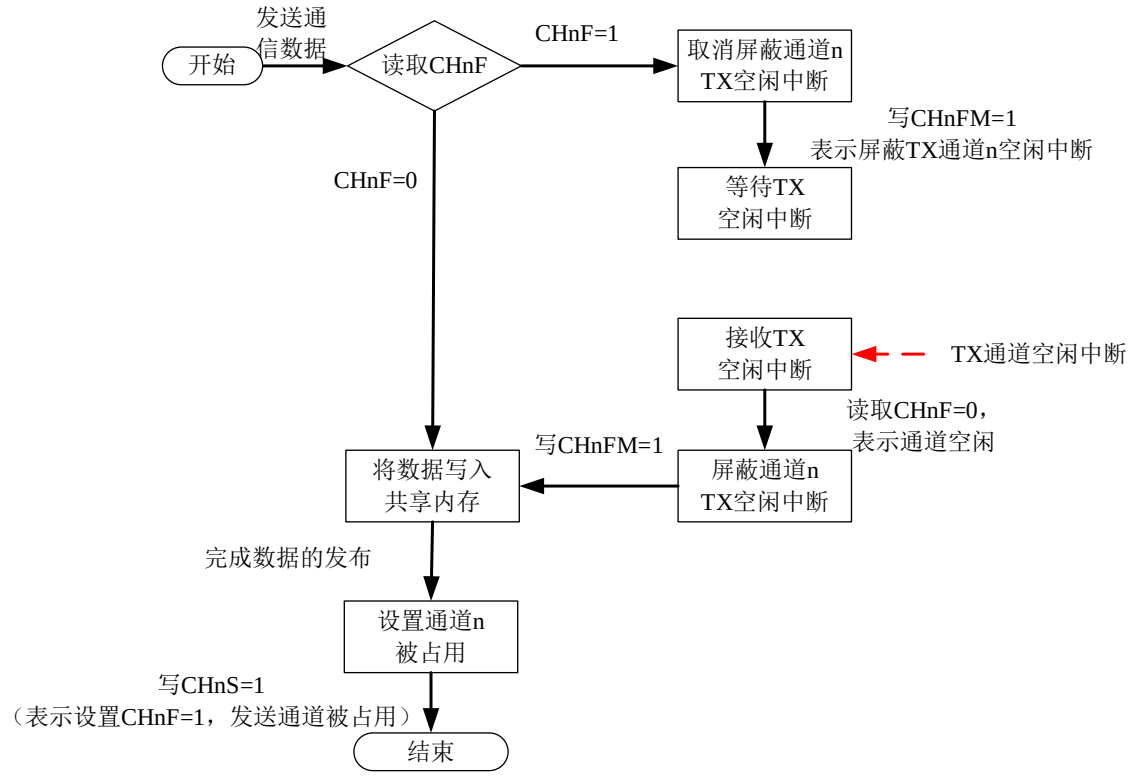


图 3.2.4.1 IPCC 单工发送模式

（2）IPCC 单工接收模式

如下图 3.2.4.2 所示为 IPCC 单工发送模式示意图。当产生通道占用中断时，接收端确定是哪个通道被占用，并屏蔽相应的通道占用中断，随后，可以从共享内存中读取数据，一旦读取数据，通道状态标志 CHnF 将被清除（CHnF=0），通道占用中断被取消屏蔽。

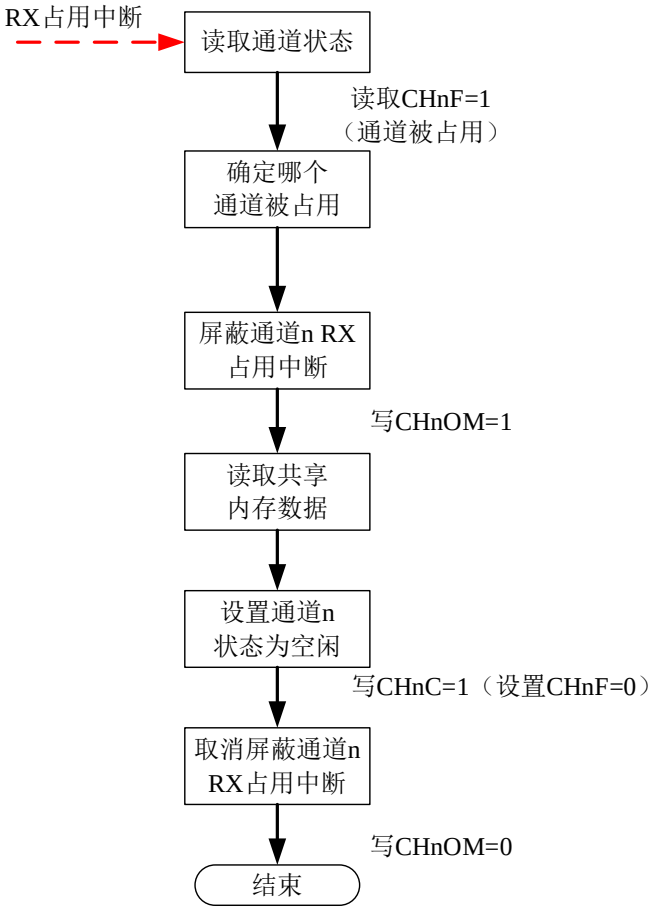


图 3.2.4.2 IPCC 单工接收模式

（3）IPCC 半双工发送模式

如下图 3.2.4.3 所示为 IPCC 半双工发送模式示意图。

半双工过程允许使用单个共享缓冲区将消息从发送方传输到接收方，然后将响应（Response pending）从接收方发送回发送方。

发送端发送数据过程：软件变量 response pending=1 表示发送端等待接收端的应答，response pending=0 表示获得接收端的应答。

首先，发送方检查通道状态标志，如果通道状态标志指示通道被占用（CHnF=1），即由于接收方尚未发送对先前消息的响应，则发送方等待响应，如果通道空闲，发送方可以将消息写入共享内存中。随后，通道状态标志被设置为被占用，这会触发接收端的 RX 通道占用中断，一旦通道状态标志 CHnF=1，通道空闲中断被取消屏蔽。通道空闲中断表示接收方发送的响应是否可用，当产生通道空闲中断时（响应就绪），发送方确定释放哪个通道并屏蔽相应的通道空闲中断，随后，可以从共享内存中读取响应。

接收端的响应过程：接收处理器等待软件变量 response pending 是否等于 1，如果等于 1，则接收端将响应写入共享内存中（发布响应），一旦发布了响应，通道状态标志 CHnF 被 CHnC 清除，则 CHnF 为 0，且接收端取消屏蔽通道占用中断（CHnOM = 0）。

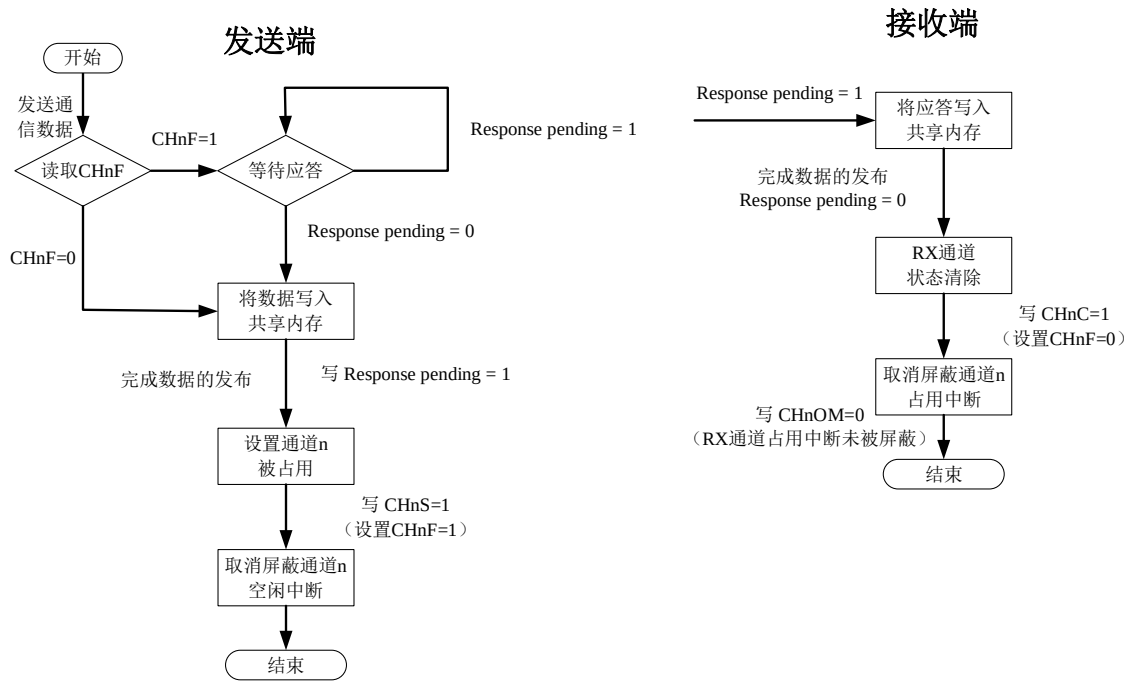


图 3.2.4.3 IPCC 半双工发送模式

（4）IPCC 半双工接收模式

如下图 3.2.4.4 所示为 IPCC 半双工接收模式示意图。

接收端读取数据：当产生通道占用中断时，接收端判断哪个通道被占用，并屏蔽响应的通道占用中断，随后从共享内存中接收数据，只有在接收方将响应发送到共享内存中后，通道才会被释放。

发送端读取响应：为了接收响应，通道空闲中断被取消屏蔽，发送处理器会检查那个通道变为空闲，屏蔽相关的通道空闲中断，然后从共享内存中读取响应。

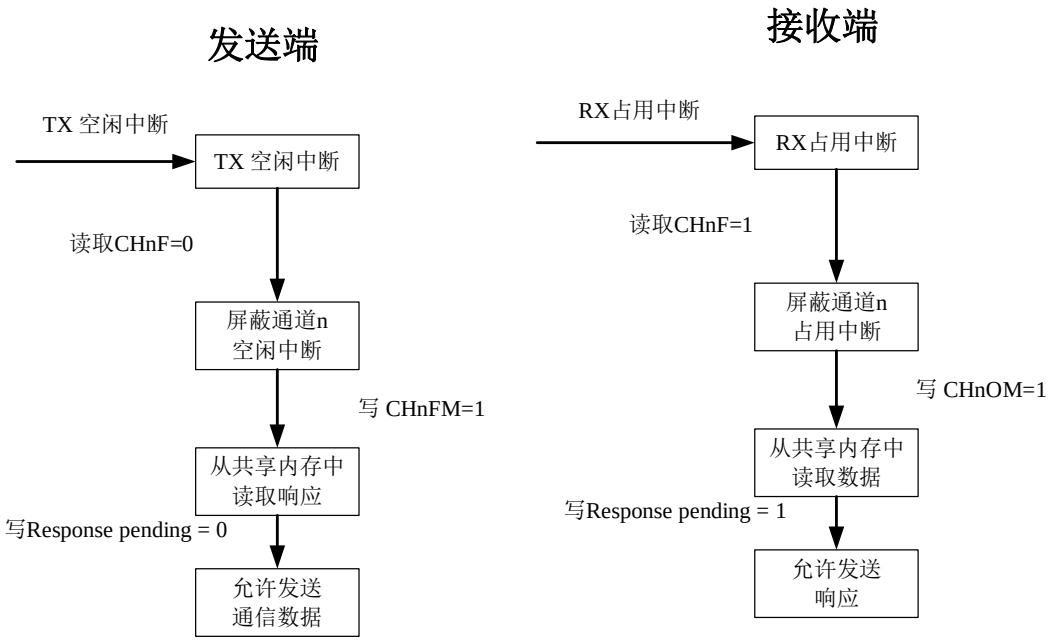


图 3.2.4.4 IPCC 半双工接收模式

IPCC 为 6 个双向通道提供了一种非阻塞信令机制，以原子方式发布和检索通信数据。非阻塞方式允许处理器以非阻塞的方式交换信息，例如读取数据，当不满足条件、未能立刻得到结果时，先挂起进程，然后每隔一段时间去检查是否满足条件了，若满足，则进行读取操作，若不满足，则不读取，这就是非阻塞方式。以上描述的通信过程可以用如下图 3.2.4.5 来表示，通信的处理器都可以访问共享的内存，围绕共享内存，从主处理器到协处理器以及从协处理器到主处理器的方向可以配置一根中断线，即内核间中断（PPI，简称核中断），核中断发起方首先将消息数据写到共享内存中，然后发起核间中断，被中断的核线程在中断服务程序中读取该共享内存，以获得发起方通知的数据。

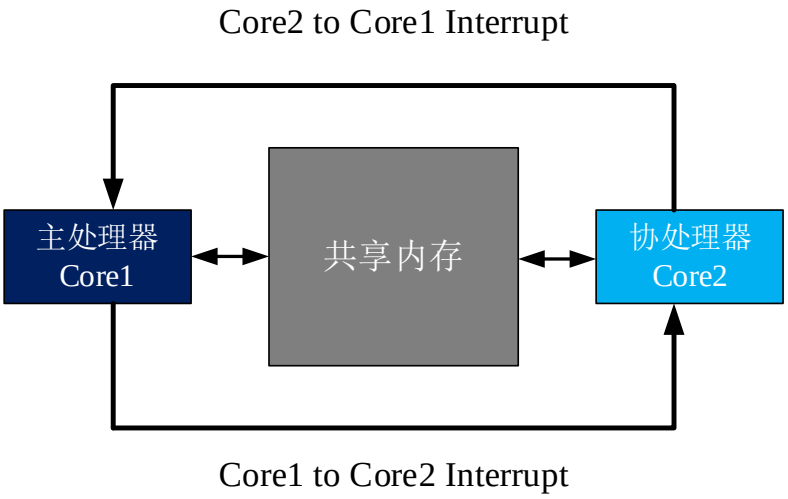


图3.2.4.5 共享内存和核间中断

在第一章配置OpenAMP时，我们有开启了M4的IPCC中断，如下图3.2.4.6所示：

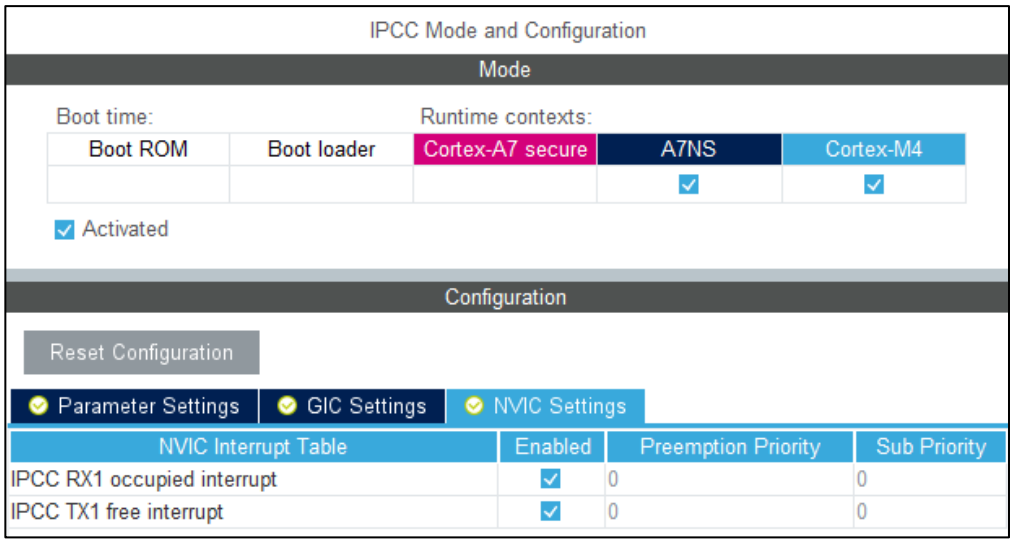


图3.2.4.6 M4开启的IPCC中断

在M4工程的stm32mp1xx_it.c文件中可以找到IPCC中断处理程序，如下所示，其处理过程和以上的过程分析的差不多，此处就不再重复分析了。

```
/**
 * @brief       IPCC RX1 占用中断
 * @param       无
```

```
 * @retval      无
 */
void IPCC_RX1_IRQHandler(void)
{
  /* 该函数处理 IPCC Rx 占用中断请求 */
  HAL_IPCC_RX_IRQHandler(&hipcc);
}
/**
 * @brief       IPCC TX1 空闲中断
 * @param       无
 * @retval      无
 */
void IPCC_TX1_IRQHandler(void)
{
  /* 该函数处理 IPCC Tx 空闲中断请求 */
  HAL_IPCC_TX_IRQHandler(&hipcc);
}
```

以上的处理过程在 M4 工程中都已经自动为我们配置好了，其实我们不用关注这部分的实现，我们更应该关注的是 RemoteProc 框架和 RPMsg 框架部分。

3.2.5 Mailbox 框架

我们来看看和 IPCC 紧密联系的 Mailbox（邮箱）框架，Mailbox 是一种驱动架构，它依赖于硬件平台来实现，例如，STM32MP157 平台的 Mailbox 依赖 IPCC 外设，其 Mailbox 框架如下图 3.2.5.1 所示：

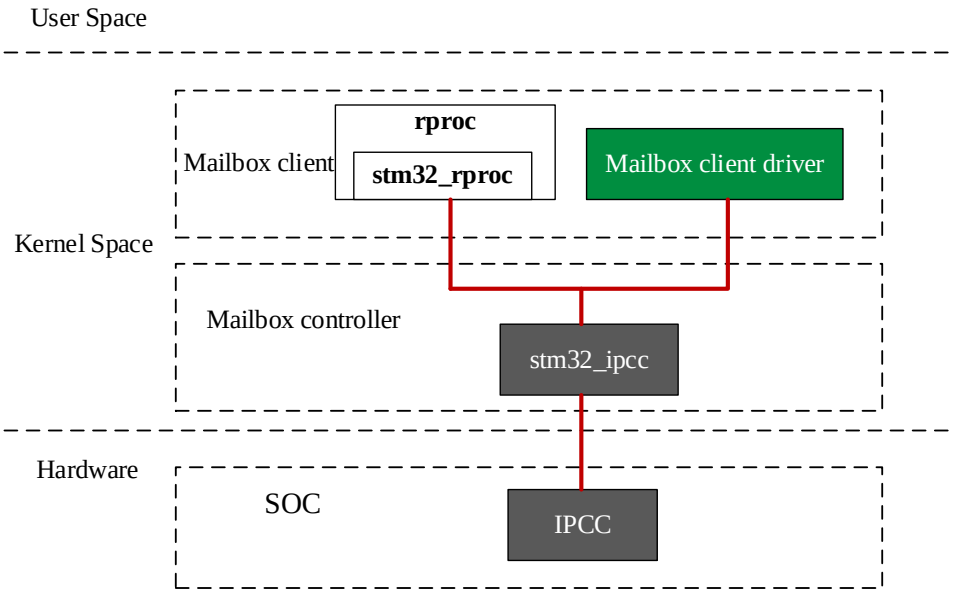


图 3.2.5.1 Mailbox 框架

上层应用如何知道数据已经发送完毕或者已经接收完全了呢？这个工作就交给 Mailbox 框架来处理，Mailbox 的实现分为邮箱控制器（Mailbox contoller）和邮箱客户端（Mailbox client）：

（1）邮箱控制器主要负责配置和处理来自 IPCC 外设的消息队列或者中断请求（IRQ），并为负责发送或接收通知消息的邮箱客户端提供一个通用的 API。

（2）邮箱客户端主要负责发送或接收通知消息，它通过邮箱控制器提供的通道来发送或接收通知消息，这个通道就是 IPCC 通道。

用户可以自己定义邮箱客户端，如上图 3.2.5.1 所示，Linux 下的邮箱控制器是 stm32_ipcc，它配置和控制 IPCC 外设，可以提供邮箱服务。stm32_rproc 是远程处理器平台驱动程序，它主要处理与远程处理器关联的平台资源（例如寄存器、看门狗、复位、时钟和存储器），可将对应回调函数注册到 Remoteproc 框架中，还可以通过邮箱框架将通知消息转发到远程处理器。

邮箱框架的大概工作流程是：

① 先注册邮箱控制器；② 邮箱客户端发送数据前，先申请通道；③ 客户端发送数据；④ 邮箱客户端记录数据；⑤ 邮箱控制器将底层接收到的数据回调给上层应用；⑥ 当数据发完时，邮箱控制器通知上层当前数据已经发送完成；⑦ 邮箱客户端释放通道。

关于 Mailbox 框架，在 Linux 内核源码 drivers/mailbox 目录下有相关的驱动，感兴趣的小伙伴可以自行研究，本篇，我们重点讲解的是 RPMsg 框架和 Remoteproc 框架。

3.3 OpenAMP 框架

在前面我们提到 M4 使用的是 OpenAMP 框架，下面我们来了解这个框架的作用。

在具有多个处理器的分布式系统中会面临一系列难题，例如内存怎么分配、系统资源怎么分配、如何为处理器之间的信息交换设置共享内存以及中断的分配和管理等等，而开源框架 OpenAMP 就是为解决这些难题而产生的。

OpenAMP（开放非对称多处理）最初是由 Mentor Graphics 与 Xilinx 公司为解决 AMP 系统中的 RTOS 或者裸机程序与 Linux 接口能进行通信而开发的一个软件框架。目前 OpenAMP 可用在 ST、NXP、TI 和 Xilinx 等平台上，这些厂商已经提供了移植好的 OpenAMP 的开发实例，用户可参考实例来进行开发。如果你打开 STM32Cube 固件包，在 STM32Cube_FW_MP1_V1.2.0\Projects\STM32MP157C-DK2\Applications\OpenAMP 下就有参考示例，我们后面的例程也是参考了这些示例。如下图 3.3.1 所示。

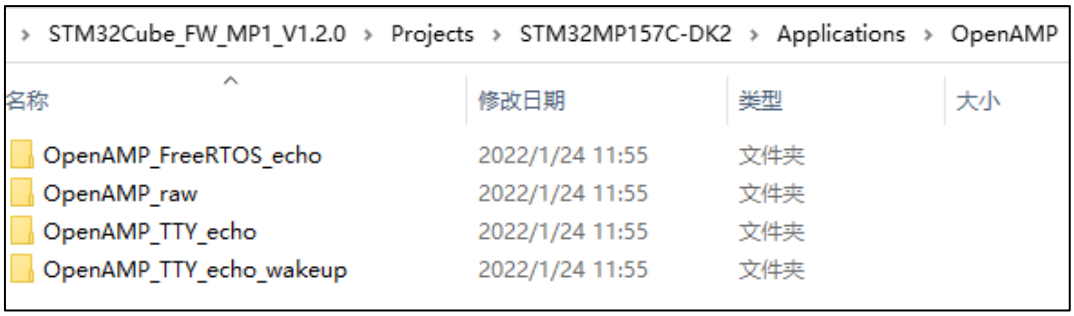


图 3.3.1 ST 提供的参考示例

OpenAMP 为 AMP 系统开发应用程序提供了三个重要组件：Virtio、RPMsg 和 Remoteproc，下面我们来了解这三个组件。

3.3.1 Virtio(虚拟化模块)

Virtio 是一个提供共享内存管理的虚拟设备框架，Virtio 中的 vring 是指向数据缓冲区指针的 FIFO 队列，有两个单向的 vring，一个 vring 专用于发送到远程处理器的消息，另一个 vring 用于从远程处理器接收的消息，两个 vring 组成一个环形，A7 和 M4 的数据通过 vring 的缓冲

区来共享，vring 的缓冲区就是两个处理器的共享内存（Shared memory，也可称为 IPC Buffers 或者 Vring buffers），在 ST 给的参考配置中，STM32MP157 共享的内存就在 SRAM3 中。

关于 Virtio 此处不做深入研究，本小节我们只需要了解到 Virtio 有两个 vring，一个 vring 用于发送，一个 vring 用于接收，数据就存放于共享的内存中，即 Vring buffers，因为有两个 vring，所以共享的内存一半用于发送，一半用于接收。如下图 3.3.1.1 所示，处理器通过 vring 环形缓冲区完成数据流转发。

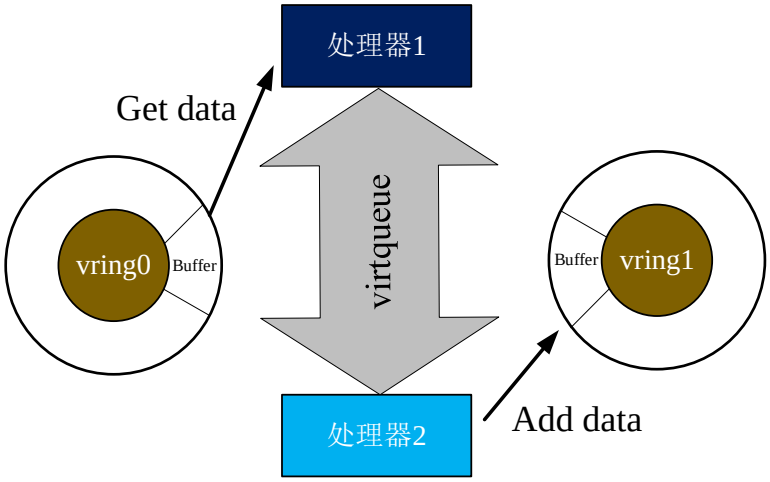


图 3.3.1.1 vring 转发数据

3.3.2 RPMsg(远程处理器消息传递)

RPMsg 框架如下图 3.3.2.1 所示，可以看到 RPMsg 框架位于 Virtio 的上层，RPMsg（Remote Processor Messaging）框架是一种基于 Virtio 的消息总线。

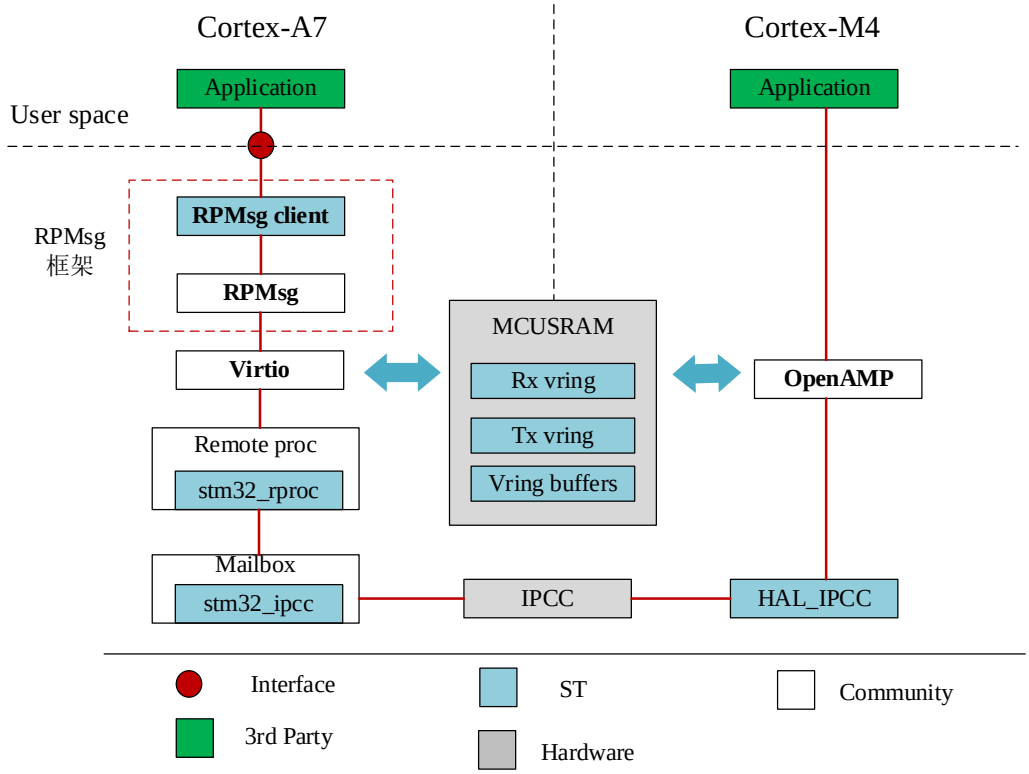


图 3.3.2.1 RPMsg 框架示意图

图中的 Mailbox 框架在前面我们已经了解了，位于 Mailbox 框架上层的是 Remoteproc（远程处理器）框架，该框架我们后面会介绍到。在 Cortex-A7 上，RemoteProc 框架根据协处理器的固件资源表中的可用信息来激活基于 Linux 的进程间通信（IPC），RPMsg 服务是通过 RPMsg 框架实现的，邮箱服务由邮箱驱动程序 stm32_ipcc 实现。在 Cortex-M4 上，RPMsg 服务由 OpenAMP 库实现，邮箱服务由 HAL_IPCC 驱动程序实现。

在 ST 给的默认配置中，MCUSRAM 里的 SRAM3 区域有两个 vring，分别用于发送和接收的消息，Vring buffers 缓冲区就是共享的内存。RPMsg 框架基于 Virtio 的 vrings，它通过 Virtio 的 vrings 向远程处理器发送消息，或者从远程处理器接收消息，当新消息已经在共享内存中时，邮箱框架就会通知处理器已经有消息可以接收。

RPMsg 实际上是一种基于 Virtio 的消息总线，用于实现的是消息传递（传递核间数据），可以认为 RPMsg 是一个与远程处理器通信的通道，这个通道，我们也可以称它为 RPMsg 设备，每个通道都有一个本地源地址和远程目标地址，消息就可以在源地址和目标地址之间进行传输，关于 RPMsg，我们会在后面的章节进行讲解。基于 RPMsg，最终的通信过程变化如下图 3.3.2.2 所示。

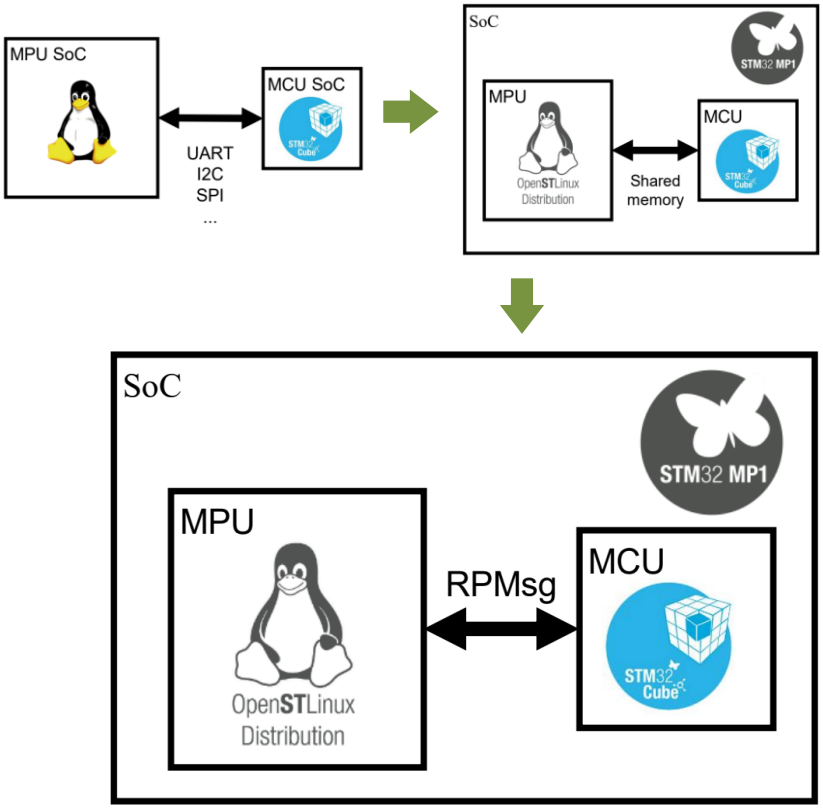


图 3.3.2.2 通信过程转化图

基于这些软件框架，所有的数据都在 RPMsg 上传递，RPMsg 将数据传输到内核层的 RPMsg 客户端，再通过内核下的设备节点传输给用户空间。结合网络 TCP/IP 结构层级关系，共享的内存和核间中断相当于物理硬件层，Virtio 相当于 MAC 子层，而 RPMsg 相当于传输层，如下图 3.3.2.3 所示：

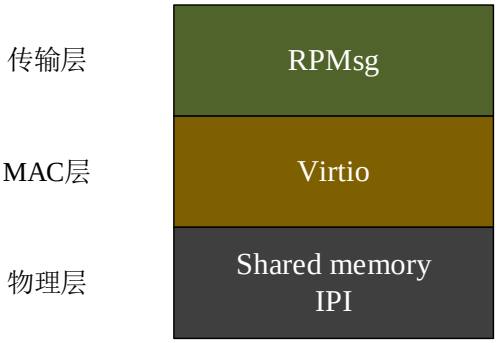


图 3.3.2.3 共享内存、Virtio 和 RPMsg 的层级关系

3.3.3 RemoteProc(远程处理)

对于具有非对称多处理的 SOC，不同的核心可能跑不同的操作系统，例如 STM32MP157 的 Cortex-A7 运行 Linux 操作系统，Cortex-M4 可以运行 OneOS 操作系统或者裸机程序。为了使运行 Linux 的主处理器与协处理器之间能够轻松通信，在 Linux3.4.X 版本以后就引入了 Remoteproc 核间通信框架，Remoteproc 框架是由 Texas Instrument 开发的，在此基础上 Mentor Graphics 公司开发了一种软件框架 OpenAMP，在这个框架下，主处理器上的 Linux 操作系统可以对远程处理器及其相关软件环境进行生命周期管理，即启动或关闭远程处理器。

我们以 STM32MP157 为例来看下 M4 和 A7 的 Remoteproc 框架，如下图 3.3.3.1 所示：

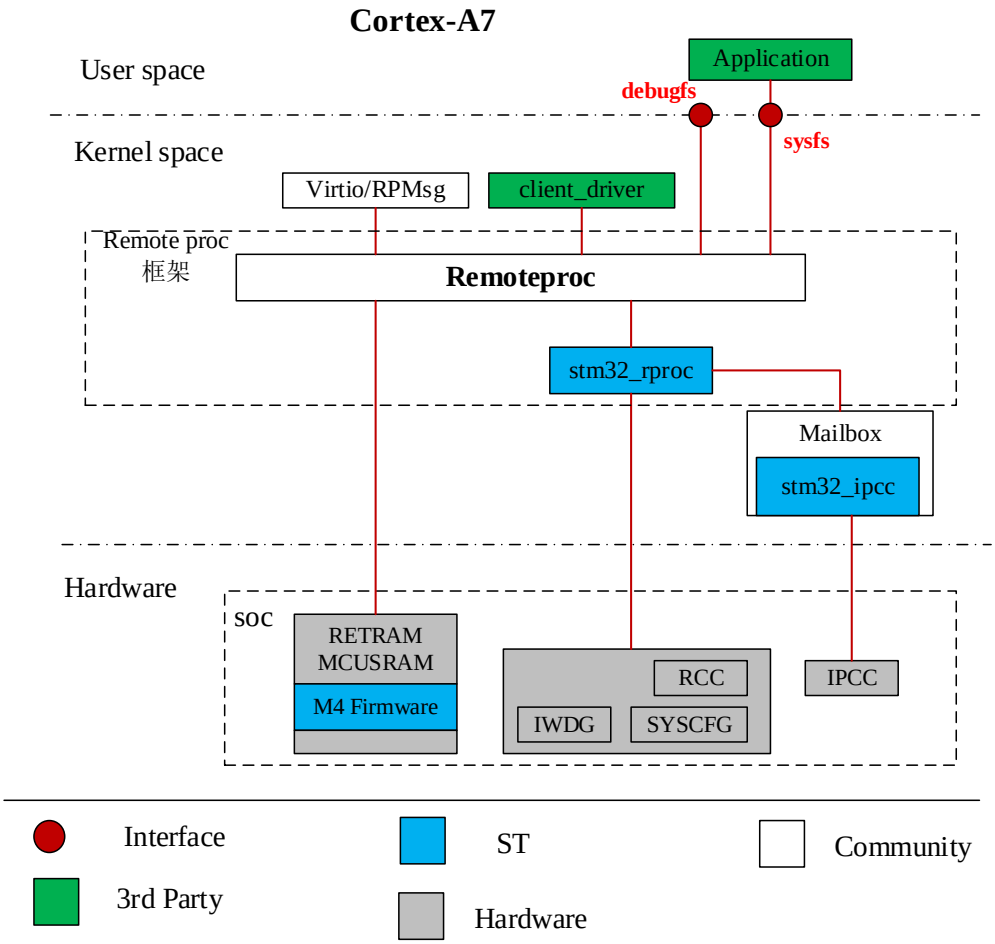


图 3.3.3.1 Remoteproc 框架

Remoteproc 是通用远程处理框架部分，其作用是：

1）A7 将 M4 固件映像的代码段和数据段加载到 M4 内存中，以便就地执行程序；

2）解析固件资源表以设置关联的资源（固件中各个段的起始地址和大小等信息，Virtio 设备特性、vring 地址、大小和对齐信息）；

3）控制 M4 内核固件的启动和关闭；

4）为与 M4 的通信建立 RPMsg 通信通道；

5）提供监视和调试远程服务（使用 sysfs 和 debugfs 文件系统，这两个文件系统在开发板的 Linux 文件系统中已经默认配置好了，可开机即用）。

stm32_rproc 是远程处理器（M4 内核）的驱动程序，其作用是：

1）向 Remoteproc 框架注册供应商特定的功能（如回调函数部分）；

2）处理和 M4 关联的平台资源（例如寄存器，看门狗，复位，时钟和存储器）；

3）通过邮箱框架(Mailbox)将通知转发到 M4。

以上所说的固件就是 M4 的可执行文件，如 MDK 下编译好的.axf 文件或 STM32CubeIDE 下编译好的.elf 文件。A7 称为主处理器，M4 称为协处理器或远程处理器，主处理器先启动，再引导协处理器启动。主处理器可以通过 Remoteproc 框架先加载协处理器的固件，然后再解析固件资源表发布的信息来配置协处理器系统资源并创建 Virtio 设备，一旦主处理器上的 Remoteproc 加载并启动远程处理器的固件后，协处理器就得到运行。协处理器启动后，此时双核之间还不能进行核间通信，这需要先创建 RPMsg 通道，并由主处理器发送第一条消息以后，双核之间才可以进行核间通信，才可以互发数据。如果需要停止运行固件，主处理器也可以关闭固件，协处理器程序随即停止运行。总之，Remoteproc 框架实现了对远程协处理器生命周期的管理(LCM)和控制，如下图 3.3.3.2 所示：

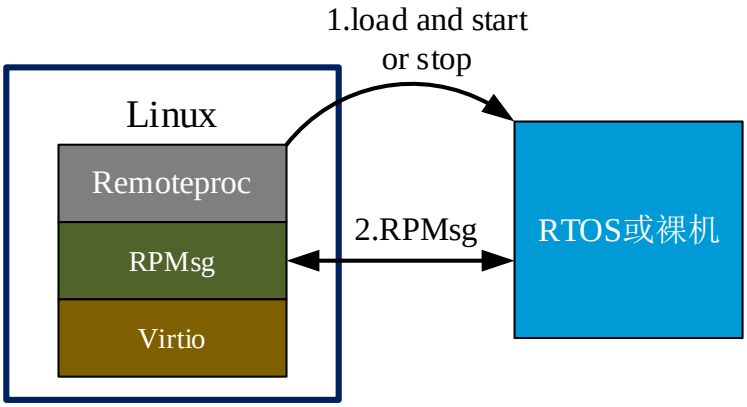


图 3.3.3.2 Remoteproc 控制协处理器

如下图 3.3.3.3 所示，在 STM32MP157 的 Linux 操作系统下执行以下指令可以查看此时运行的 CPU：

```
cat /proc/cpuinfo
```

```
root@ATK-MP157:~# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 48.00
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

processor       : 1
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 48.00
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

Hardware        : STM32 (Device Tree Support)
Revision        : 0000
Serial          : 004800353030511739383538
root@ATK-MP157:~#
```

图 3.3.3.3 查看 CPU 信息

其中 A7(非安全模式)是 CPU0，也就是主处理器，A7(安全模式)是 CPU1，M4 是协处理器。CPU1 和 CPU0 运行 Linux 操作系统，属于 SMP 子系统。

经过以上分析，OpenAMP 开源软件框架具有如下功能和特点：

1）提供处理器生命周期管理和处理器间通信功能；

2）提供可用于 RTOS 和裸机软件环境的独立库；

3）与上游 Linux Remoteproc、RPMsg 和 VirtIO 组件具有兼容性。

使用 OpenAMP 实现核间通信的过程：

假定主处理器已经在运行并且远程处理器处于某种状态，如待机或断电状态，主处理器基于 Remoteproc 框架，先将远程处理器的固件加载到内存中，然后主处理器启动远程处理器（运行固件）并等待它初始化完成，例如唤醒、解除复位、上电等，远程处理器初始化完成后通知主处理器，主处理器和远程处理器建立起 RPMsg 通信通道，通过 RPMsg 通道两者可进行核间通信。

3.4 驱动文件介绍

3.4.1 Linux 驱动编译配置

Linux 下的 IPCC、Mailbox、Remoteproc、Virtio 和 RPMsg 相关驱动在 Linux 内核下已经默认使能了，打开 Linux 内核源码的 arch/arm/configs/stm32mp1_atk_defconfig 配置文件，可以找到如下配置：

```
/* 使能 Mailbox */
CONFIG_ARM_SMC_MBOX=y
CONFIG_PL320_MBOX=y
/* 使能 stm32_ipcc */
CONFIG_STM32_IPCC=y
/* 使能 Remoteproc */
CONFIG_REMOTEPROC=y
/* 使能 stm32_rproc */
CONFIG_STM32_RPROC=y
/* 使能 rpmsg_vrtio */
```

```
CONFIG_RPMSG_VIRTIO=y
/* 使能 rpmsg_tty，可用于虚拟串口 */
CONFIG_RPMSG_TTY=y
```

可以看到，以上选项默认配置为“y”，即默认将相关驱动编译进内核中，我们启动 Linux 系统后，以上驱动已经在内核启动时自行加载了，不需要我们手动去加载相关驱动了。也可以通过 Linux 内核的 menuconfig 配置界面去查看以上驱动的配置选项，例如，执行如下 menuconfig 配置指令，在 Device drivers→Mailbox Hardware Support 下可以看到已经默认使能了邮箱驱动，如下图 3.4.1.1 所示：

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig
```

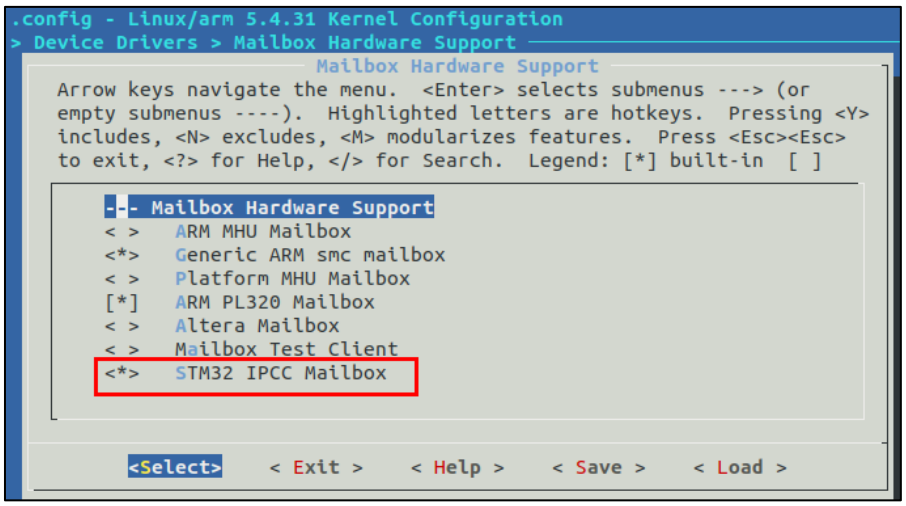


图3.4.1.1 Linux内核源码开启了Mailbox

其它的驱动配置选项，也可以在 menuconfig 下找到，这里就不一一列出了。

3.4.2 Linux 驱动文件

以上是 Linux 下的驱动编译配置选项，之所以可以编译出以上驱动，是因为在 Linux 内核源码下已经有芯片厂商移植好的对应的驱动文件了，无需我们编写驱动，只要编译内核后就可以使用这些驱动了，而我们更应该关注的是如何去使用这些驱动。

1. IPCC 和 Mailbox 驱动源码

Mailbox 相关说明文档或示例代码如下表 3.4.2.1 所示：

文件	描述
通用 Mailbox 框架说明文档	Documentation/mailbox.txt
通用 Mailbox 设备树说明文档	Documentation/devicetree/bindings/mailbox/mailbox.txt
ST 官方 Mailbox 设备树说明文档	Documentation/devicetree/bindings/mailbox/sti-mailbox.txt
ST 提供的通用 Mailbox 测试示例代码文件	drivers/mailbox/mailbox-test.c
ST 实现的创建 Mailbox client 的驱动文件	drivers/remoteproc/stm32_rproc.c
Mailbox 和 IPCC 相关的驱动源码	drivers/mailbox
ST 实现的创建 Mailbox contoller 的驱动文件	drivers/mailbox/stm32-ipcc.c

表 3.4.2.1 Mailbox 相关说明文档或示例代码

在 Linux 内核源码 drivers/mailbox 目录下的是 Mailbox 和 IPCC 相关的驱动源码文件，如下图 3.4.2.1 所示：

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/mailbox$ ls
armada-37xx-rwtm-mailbox.c  mailbox-altera.c          pcc.c
arm_mhu.c                   mailbox.c                 pl320-ipc.c
arm-smc-mailbox.c           mailbox.h                 pl320-ipc.o
arm-smc-mailbox.o           mailbox.o                 platform_mhu.c
bcm2835-mailbox.c           mailbox-sti.c             qcom-apcs-ipc-mailbox.c
bcm-flexrm-mailbox.c        mailbox-test.c            rockchip-mailbox.c
bcm-pdc-mailbox.c           mailbox-xgene-slimpro.c   stm32-ipcc.c
built-in.a                  Makefile                  stm32-ipcc.o
hi3660-mailbox.c            modules.builtin           tegra-hsp.c
hi6220-mailbox.c            modules.order             ti-msgmgr.c
imx-mailbox.c               mtk-cmdq-mailbox.c        zynqmp-ipi-mailbox.c
Kconfig                     omap-mailbox.c
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/mailbox$
```

图 3.4.2.1 Mailbox 和 IPCC 相关驱动源码

2. Remoteproc 驱动源码

Remoteproc 相关说明文档或示例代码如下表 3.4.2.2 所示：

文件	描述
Remoteproc 框架说明文档	Documentation/remoteproc.txt
ST32MP 系列芯片 Remoteproc 设备树说明文档	Documentation/devicetree/bindings/remoteproc/stm32-rproc.txt
Remoteproc 系统资源管理器说明文档	Documentation/devicetree/bindings/remoteproc/rproc-srm.txt
Remoteproc 驱动程序	drivers/remoteproc

表 3.4.2.2 Remoteproc 相关说明文档或示例代码

在 Linux 内核源码的 drivers/remoteproc 目录下就有 Remoteproc 相关的驱动文件，如下图 3.4.2.2 所示：

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/remoteproc$ ls
built-in.a              qcom_common.h       remoteproc_core.c       rproc_srm_core.c
da8xx_remoteproc.c      qcom_q6v5_adsp.c    remoteproc_core.o       rproc_srm_core.h
imx_rproc.c             qcom_q6v5.c         remoteproc_debugfs.c    rproc_srm_core.o
Kconfig                 qcom_q6v5.h         remoteproc_debugfs.o    rproc_srm_dev.c
keystone_remoteproc.c   qcom_q6v5_mss.c     remoteproc_elf_loader.c rproc_srm_dev.o
Makefile                qcom_q6v5_pas.c     remoteproc_elf_loader.o stm32_rproc.c
modules.builtin         qcom_q6v5_wcss.c    remoteproc_internal.h   stm32_rproc.o
modules.order           qcom_sysmon.c       remoteproc_sysfs.c      st_remoteproc.c
omap_remoteproc.c       qcom_wcnss.c        remoteproc_sysfs.o      st_slim_rproc.c
omap_remoteproc.h       qcom_wcnss.h        remoteproc_virtio.c     wkup_m3_rproc.c
qcom_common.c           qcom_wcnss_iris.c   remoteproc_virtio.o
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/remoteproc$
```

图 3.4.2.2 Remoteproc 相关驱动源码

3. Virtio 驱动源码

在 Linux 内核源码的 drivers/virtio 目录下就有 Virtio 相关的驱动文件，如下图 3.4.2.3 所示：

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/virtio$ ls
built-in.a       virtio_balloon.c  virtio.o             virtio_ring.c
Kconfig          virtio.c          virtio_pci_common.c  virtio_ring.o
Makefile         virtio_input.c    virtio_pci_common.h
modules.builtin  virtio_mmio.c     virtio_pci_legacy.c
modules.order    virtio_mmio.o     virtio_pci_modern.c
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/virtio$
```

图 3.4.2.3 Virtio 相关驱动源码

4. RPMsg 驱动源码

RPMsg 相关说明文档或示例代码如下表 3.4.2.3 所示：

文件	描述
RPMsg 框架说明文档	Documentation/rpmsg.txt
RPMsg 客户端示例代码文件	samples/rpmsg/rpmsg_client_sample.c
RPMsg 驱动程序	drivers/rpmsg

表 3.4.2.3 RPMsg 相关说明文档或示例代码

在 Linux 内核源码的 drivers/rpmsg 目录下就有 RPMsg 相关的驱动文件，如下图 3.4.2.4 所示：

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/rpmsg$ ls
built-in.a       qcom_glink_native.c  rpmsg_char.c       rpmsg_tty.o
Kconfig          qcom_glink_native.h  rpmsg_core.c       virtio_rpmsg_bus.c
Makefile         qcom_glink_rpm.c     rpmsg_core.o       virtio_rpmsg_bus.o
modules.builtin  qcom_glink_smem.c    rpmsg_internal.h
modules.order    qcom_smd.c           rpmsg_tty.c
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/rpmsg$
```

图 3.4.2.4 RPMsg 相关驱动源码

3.4.3 M4 工程驱动文件

1. OpenAMP 库文件

在前面新建的 M4 工程 RPMsg_TEST 的 OPENAMP 目录以及 Middlewares\Third_Party\OpenAMP 目录下可以找到 OpenAMP 库有关的驱动文件，其实这些文件就是从 STM32Cube 的固件包 STM32Cube_FW_MP1_V1.2.0\Middlewares\Third_Party\OpenAMP 下拷贝得来的，如下图 3.4.3.1 所示：

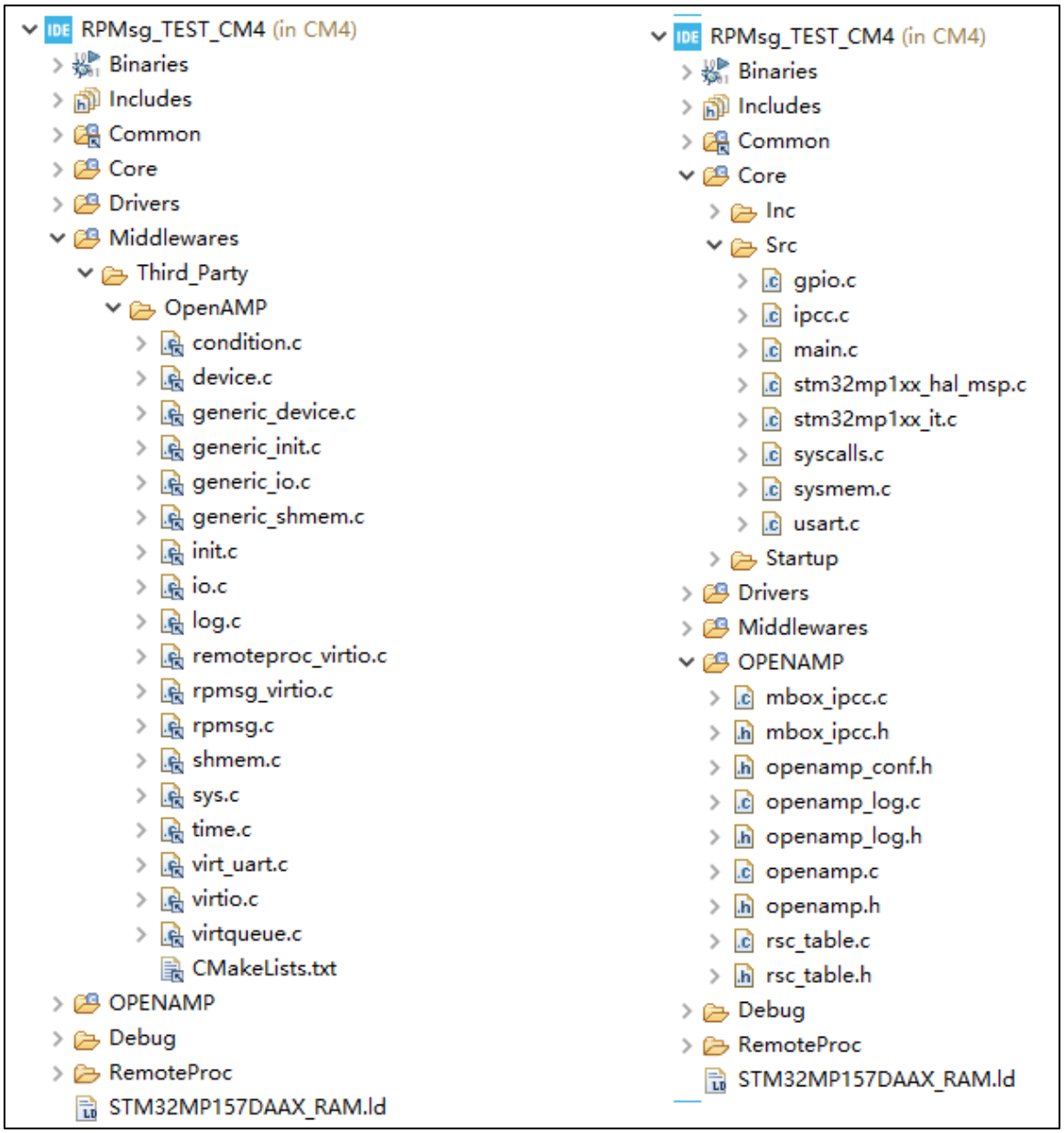


图 3.4.3.1 M4 工程目录

我们简单介绍几个重要的文件，如下表 3.4.3.1 所示是

文件	描述
virtio.c	M4 内核使用的 virtio 驱动。
mbox_ipcc.c	M4 内核使用的邮箱驱动。
rpmsg.c rpmsg_virtio.c	RPMsg 相关驱动文件。
remoteproc.c remoteproc_virtio.c	和 Remoteproc 相关的驱动文件。
rsc_table.c rsc_table_parser.c	M4 的固件资源表配置相关的文件。
openamp.c	OpenAMP 相关应用文件，OpenAMP 的初始化就是由此文件中的 MX_OPENAMP_Init()函数完成的，包括 Mailbox、共享内存、RPMsg、V

	irtio 和 vring 相关的初始化。
virtqueue.c	虚拟队列（virtqueue）相关的驱动文件，用于管理 Virtio 队列和 vring 队列。每个 RPMsg 通道包含两个与之关联的虚拟队列：一个 tx virtqueue 用于主处理器单向传输数据到远程处理器，另一个 rx virtqueue 用于远程处理器单向传输数据到主处理器。
virt_uart.c	虚拟串口相关驱动文件，双核之间是通过 RPMsg 来进行消息传递的，基于 RPMsg，OpenAMP 向上又进行了一层封装，并抽象出了虚拟串口（可以理解为一个普通的串口），主处理器和远程处理器可通过虚拟串口来传输数据。类似的，基于 RPMsg 还可以再继续封装抽象出其它更多的通信接口，例如，还可以封装成虚拟 I2C、虚拟 SPI、虚拟 GPIO、虚拟网口等等。虚拟串口是一种典型的应用，ST 已经在 STM32Cube 中提供示例，在后面的章节中，我们会使用 OpenAMP 封装好的虚拟串口来实现异核通信。

表 3.4.3.1 OpenAMP 库文件说明

其它文件我们就不一一进行介绍了，感兴趣的小伙伴可以自行研究这些文件。

OpenAMP 旨在通过 AMP 的开源解决方案来标准化嵌入式异构系统中操作环境之间的交互，基于现有的 RemoteProc、RPMsg 和 Virtio 的实现，涵盖了生命周期管理、消息传递和底层抽象等操作，OpenAMP 的标准化工作还在进行中，未来还会继续优化并添加更多的功能，OpenAMP 源码的结构还是会有变动的。

Linux 下的驱动代码和 OpenAMP 库的代码是比较多的，也比较复杂和不易于理解，对于我们来说，先理解其框架，再基于代码分析就更加容易理解其实现的过程。采用库开发的好处就是不需要我们从 0 到 1 去编写底层的实现，我们只需要在上层调用对应的 API 即可完成对应的功能，这将加快产品应用程序迁移的速度，大大节省了开发时间和开发成本。

第四章 Remoteproc 相关驱动简析

远程处理器（Remoteproc）框架负责根据固件资源表中可用的信息激活 Linux 端的进程间通信（IPC），本节我们来分析 Linux 下的 Remoteproc 相关驱动，了解 Remoteproc 是怎样控制远程处理器的。本小节的内容会涉及到分析代码，会比较枯燥乏味，如果只想了解 Remoteproc 使用方法，可以直接看本章的最后一小节。

本章分为如下几部分：

4.1 资源表
4.2 存储和系统资源分配
4.3 Linux 下 Remoteproc 相关 API
4.4 链接脚本
4.5 Remoteproc 的使用

4.1 资源表

远程处理器的固件映像一般包括资源表（resource_table）、用户应用程序、RTOS 或者裸机（Bare Metal ，简称 BM）相关代码以及 OpenAMP 的库。主处理器将远程处理器的固件加载到远程处理器的内核中后，会去解码固件映像以获取关联的资源，并为固件代码段和数据留出内存，在启动远程处理器后，主处理器创建 RPMsg 通道，RPMsg 通道用于远程通信。

图 4.1.1 远程固件映像组成

远程处理器的资源包含远程处理器在上电之前需要的系统资源，例如分配给远程处理器的连续物理内存，还有分配给远程处理器的外围设备（这些设备可以是保留的、未使用的），这些资源在 stm32mp157-m4-srm.dtsi 设备树文件的资源管理器（即 m4_system_resources 节点）中配置，我们后面会讲到。除了系统资源之外，远程处理器都会有一个资源表（resource_table），资源表还可能包含资源条目，这些条目发布远程处理器支持的功能或存在的配置，如 Virtio 设备中的 vring 地址、vring 的大小等。只有在满足所有资源的要求后，Remotecore 才会启动设备。打开内核源码下的 include/linux/remoteproc.h 文件，找到如下结构体 resource_table，它是固件资源表头：

```
truct resource_table {     /* 固件资源表头 */
        u32 ver;           /* 版本号 */
        u32 num;           /* 可用的资源条目数量 */
        u32 reserved[2];   /* 保留（必须为零） */
        u32 offset[0];     /* 指向各种资源条目的偏移量的数组 */
} __packed;
```

紧随此固件资源表头的是资源条目本身，每个条目都以“struct fw_rsc_hdr”标头开头，条目本身的内容会紧跟在这个头之后，根据资源类型来解析。以下是资源条目标头开头：

```
struct fw_rsc_hdr {   /* 固件资源入口头 */
       u32 type;      /* 资源类型 */
       u8 data[0];    /* 资源数据 */
} __packed
```

一些资源条目仅仅是公告，如其中主机被告知特定的 Remoteproc 配置，其它条目需要主机做一些事情（例如分配系统资源）。有时需要协商固件请求资源，一旦分配资源，主机应提供其详细信息（如分配的内存区域的地址）。以下是当前支持的各种资源类型：

```
enum fw_resource_type {  /* 资源条目的类型 */
  RSC_CARVEOUT     = 0,   /* 请求分配物理上连续的内存区域 */
  RSC_DEVMEM       = 1,   /* 请求 iommu_map 一个基于内存的外设 */
  RSC_TRACE        = 2,   /* 宣布跟踪缓冲区的可用性，远程处理器将在其中写入日志 */
  RSC_VDEV         = 3,   /* 声明对 Virtio 设备的支持，并作为其 Virtio 标头。 */
  RSC_LAST         = 4,     /* 把这个放在标准资源的末尾 */
  RSC_VENDOR_START = 128, /* 供应商特定资源类型范围的开始 */
  RSC_VENDOR_END   = 512, /* 供应商特定资源类型范围的末尾 */
};
```

以上这些值用作 rproc_handle_rsc()函数（在 remoteproc_internal.h 文件中）查找表的索引。当注册一个新的远程处理器时，Remoteproc 框架将查找其资源表并注册它支持的 Virtio 设备，固件应提供有关其支持的 Virtio 设备及其配置的 Remoteproc 信息，RSC_VDEV 资源条目应指定 Virtio 设备 ID（如 virtio_ids.h）、Virtio 功能、Virtio 配置空间、vrings 信息等，我们可以通过查看 Linux 文件系统/sys/kernel/debug/remoteproc/remoteproc0/resource_table 文件得到 RSC_VDEV 信息。

资源表信息在 OpenAMP 库的 rsc_table.h 和 rsc_table.c 下，打开前面创建的工程，打开 OpenAMP 目录下的 rsc_table.c 文件，如下图 4.1.2 所示：

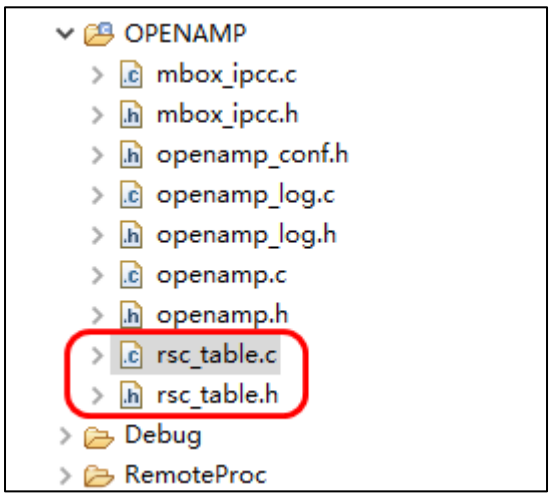


图 4.1.2 rsc_table.c 文件

下面贴出 rsc_table.c 文件的代码：

```
1  #if defined(__ICCARM__) || defined (__CC_ARM)
2  #include <stddef.h>              /* offsetof 定义所需 */
3  #endif
4  #include "rsc_table.h"
5  #include "openamp/open_amp.h"
```

```
6
7   /* 将资源表放在专门的 ELF 部分 */
8   #if defined(__GNUC__)
9   #define __section_t(S)          __attribute__((__section__(#S)))
10  #define __resource               __section_t(.resource_table)
11  #endif
12
13  #if defined (LINUX_RPROC_MASTER)
14   #ifdef VIRTIO_MASTER_ONLY
15    #define CONST
16   #else
17    #define CONST const
18   #endif
19  #else
20   #define CONST
21  #endif
22
23  #define RPMSG_IPU_C0_FEATURES       1
24  #define VRING_COUNT                 2       /* vring 个数 */
25
26  /* VirtIO RPMsg 设备 ID */
27  #define VIRTIO_ID_RPMSG_            7
28
29  #if defined (__LOG_TRACE_IO_)
30  extern char system_log_buf[];
31  #endif
32
33  #if defined(__GNUC__)
34  #if !defined (__CC_ARM) && !defined (LINUX_RPROC_MASTER)
35
36  /*
37   *由于 GCC 在启动时没有初始化资源_表，因此将其声明为 volatile，
38   *以避免对 CM4 进行编译器优化（请参阅下面的 resource_table_init()）
39   */
40  volatile struct shared_resource_table __resource __attribute__((used))
                                                          resource_table;
41  #else
42  CONST struct shared_resource_table __resource __attribute__((used))
       resource_table = {
43  #endif
44  #elif defined(__ICCARM__)
45  __root CONST struct shared_resource_table resource_table @
     ".resource_table" = {
```

```
46  #endif
47
48  #if defined(__ICCARM__) || defined (__CC_ARM) || defined (LINUX_RPROC_MASTER)
49    .version = 1,
50  #if defined (__LOG_TRACE_IO_)
51    .num = 2,
52  #else
53    .num = 1,
54  #endif
55    .reserved = {0, 0},
56    .offset = {
57      offsetof(struct shared_resource_table, vdev),
58      offsetof(struct shared_resource_table, cm_trace),
59    },
60
61    /* Virtio 设备入口*/
62    .vdev= {
63      RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0, 0,
64      VRING_COUNT, {0, 0},
65    },
66
67    /* Vring rsc 条目 - vdev rsc 条目的一部分 */
68    .vring0 = {VRING_TX_ADDRESS, VRING_ALIGNMENT, VRING_NUM_BUFFS, VRING0_ID, 0},
69    .vring1 = {VRING_RX_ADDRESS, VRING_ALIGNMENT, VRING_NUM_BUFFS, VRING1_ID, 0},
70
71  #if defined (__LOG_TRACE_IO_)
72    .cm_trace = {
73      RSC_TRACE,
74      (uint32_t)system_log_buf, SYSTEM_TRACE_BUF_SZ, 0, "cm4_log",
75    },
76  #endif
77  } ;
78  #endif
79
80  void resource_table_init(int RPMsgRole, void **table_ptr, int *length)
81  {
82
83  #if !defined (LINUX_RPROC_MASTER)
84  #if defined (__GNUC__) && ! defined (__CC_ARM)
85  #ifdef VIRTIO_MASTER_ONLY
86    /*
87     * 当前，GCC 链接器在启动时不会初始化 resource_table
88     * 全局变量，它是由主设备应用程序在此处完成的。
```

```
89     */
90    memset(&resource_table, '\0', sizeof(struct shared_resource_table));
91    resource_table.num = 1;
92    resource_table.version = 1;
93    resource_table.offset[0] = offsetof(struct shared_resource_table,
                                                                  vdev);
94
95    resource_table.vring0.da = VRING_TX_ADDRESS;
96    resource_table.vring0.align = VRING_ALIGNMENT;
97    resource_table.vring0.num = VRING_NUM_BUFFS;
98    resource_table.vring0.notifyid = VRING0_ID;
99
100   resource_table.vring1.da = VRING_RX_ADDRESS;
101   resource_table.vring1.align = VRING_ALIGNMENT;
102   resource_table.vring1.num = VRING_NUM_BUFFS;
103   resource_table.vring1.notifyid = VRING1_ID;
104
105   resource_table.vdev.type = RSC_VDEV;
106   resource_table.vdev.id = VIRTIO_ID_RPMSG_;
107   resource_table.vdev.num_of_vrings=VRING_COUNT;
108   resource_table.vdev.dfeatures = RPMSG_IPU_C0_FEATURES;
109 #else
110
111   /* 对于从设备应用程序，等待资源表正确初始化 */
112   while(resource_table.vring1.da != VRING_RX_ADDRESS)
113   {
114
115   }
116 #endif
117 #endif
118 #endif
119
120   (void)RPMsgRole;
121   *length = sizeof(resource_table);
122   *table_ptr = (void *)&resource_table;
123 }
```

第 24 行，vring 个数是 2；

第 27 行，Virtio RPMsg 设备 ID 为 7，此 ID 和 Linux 内核源码的 include/uapi/linux/virtio_ids.h 文件中的宏 VIRTIO_ID_RPMSG 的值是一样，都是 7，目的就是实现主处理器的 Virtio 匹配到远程处理器的 Virtio，然后注册注册 Virtio 设备；

第 55 行，固件资源表头的 Reserved 为 0；

第 56~59 行，资源条目的偏移量；

第 62~65 行，Virtio 设备入口，是 RSC_VDEV 资源条目，此条目支持 Virtio 设备；

第 68~69 行，vring 的条目，包括 vring 的发送端和接收端地址、地址对齐方式、RPMsg Buffer 数量以及 vring ID 等信息，打开 openamp_conf.h 头文件，找到如下代码：

```
#if defined LINUX_RPROC_MASTER
#define VRING_RX_ADDRESS              -1       /* 由主处理器分配：A7 */
#define VRING_TX_ADDRESS              -1       /* 由主处理器分配：A7 */
#define VRING_BUFF_ADDRESS            -1       /* 由主处理器分配：A7 */
#define VRING_ALIGNMENT               16       /* 数据对齐 */
#define VRING_NUM_BUFFS               16       /* RPMsg Buffer 数量 */
#else

#define VRING_RX_ADDRESS     0x10040000         /* 由主处理器分配：A7 */
#define VRING_TX_ADDRESS     0x10040400         /* 由主处理器分配：A7 */
#define VRING_BUFF_ADDRESS   0x10040800         /* 由主处理器分配：A7 */
#define VRING_ALIGNMENT      16                 /* 数据对其 */
#define VRING_NUM_BUFFS      16                 /* RPMsg Buffer 数量 */
#endif
```

VRING_RX_ADDRESS 和 VRING_TX_ADDRESS 分别是收、发 vring 的地址，VRING_BUFF_ADDRESS 是 Vring Buffer 的地址，这些地址是在 Linux 内核的设备树下配置的，即由主处理器 A7 来分配。vring 是 16 位数据对齐，VRING_NUM_BUFFS 的个数是 16 个，即每个 vring（Vring 0 或 Vring 1，表示 TX 和 RX 各有 16 个缓冲区）有 16 个缓冲区（RPMsg Buffer）。

第 83~118 行，LINUX_RPROC_MASTER 宏已经定义了，这部分的代码主要是初始化资源表，M4 不会执行，实际资源表的初始化是由 A7 来完成的。

4.2 存储和系统资源分配

在前面，我们介绍了 ST 对 STM32MP157 资源的分配情况，下面我们来看在设备树下是怎么配置的，我们重点关注 M4 的存储和外设资源分配。

4.2.1 存储分配

我们先来了解在设备树下是怎么分配 M4 的存储地址的，在 stm32mp157d-atk.dtsi 设备树下可以看到这段代码：

```
1   reserved-memory {
2       #address-cells = <1>;
3       #size-cells = <1>;
4       ranges;
5
6       mcuram2: mcuram2@10000000 {
7           compatible = "shared-dma-pool";
8           reg = <0x10000000 0x40000>;
9           no-map;
10      };
11
12      vdev0vring0: vdev0vring0@10040000 {
13          compatible = "shared-dma-pool";
```

```
14          reg = <0x10040000 0x1000>;
15          no-map;
16      };
17
18      vdev0vring1: vdev0vring1@10041000 {
19          compatible = "shared-dma-pool";
20          reg = <0x10041000 0x1000>;
21          no-map;
22      };
23
24      vdev0buffer: vdev0buffer@10042000 {
25          compatible = "shared-dma-pool";
26          reg = <0x10042000 0x4000>;
27          no-map;
28      };
29
30      mcuram: mcuram@30000000 {
31          compatible = "shared-dma-pool";
32          reg = <0x30000000 0x40000>;
33          no-map;
34      };
35
36      retram: retram@38000000 {
37          compatible = "shared-dma-pool";
38          reg = <0x38000000 0x10000>;
39          no-map;
40      };
```

第 1 行的 reserved-memory 表示该节点下分配的内存都是预留的内存，预留的内存区域一般是给特定的驱动程序使用的，它和 Linux 内核使用的内存区域不同，一般预留的内存功能和 Linux 内核的 DMA 或者 CMA 紧密相关：

如果某个节点的 compatible 属性为 shared-dma-pool，则表示该节点内存区域用作一组设备的 DMA 缓冲区共享池，此时，如果看到节点属性中有 no-map，则表示该内存不能被 Linux 内核映射为系统内存的一部分，需要从系统内存中分离出来，如果看到节点属性中有 reusable 属性，则表示该内存不用从系统内存分离出来，当特定驱动不使用这些内存的时候，OS 可以使用这些内存。注意的是，一个节点不能同时有 no-map 和 reusable 属性，因为它们是逻辑上的矛盾关系。

我们看到 reserved-memory 节点下的每个子节点都包含 no-map 属性，说明这些内存不能被 Linux 内核用作系统内存，实际上是留给 M4 的系统使用的。

设备树中设备节点的名称格式为 node-name@unit-address 例如：vdev0vring0 子节点：

```
vdev0vring0: vdev0vring0@10040000 {
          compatible = "shared-dma-pool";
          reg = <0x10040000 0x1000>;
          no-map;
```

```
};
```

vdev0vring0 是子节点的名字（node-name），@后面的 10040000 表示 vdev0vring0 节点的地址为 0x10040000。reg 后面是设备的起始地址和地址的长度。

下面我们来看看 reserved-memory 节点下的每个子节点：

- mcuram2 子节点

0x10000000 是 SRAM1 的起始地址，0x40000 的大小刚好 256KB，这段区域是 SRAM1+SRAM2 区域，这段区域主要用来保存 M4 固件的代码段和数据段：SRAM1（代码）和 SRAM2（数据），可以从 M4 工程的链接脚本（分散加载文件）看出，不过也可以通过修改链接脚本，重新划分地址范围，但是要注意的是，链接脚本的地址范围要和设备树配置的范围一致。

- vdev0vring0、vdev0vring1 和 vdev0buffer 子节点

vdev0vring0、vdev0vring1 和 vdev0buffer 子节点刚好在 SRAM3 处，即 IPC 缓冲区，我们来看看这三个节点怎么分配的。

1）vdev0vring0 子节点，0x10040000 是 vring0 的起始地址，地址长度为 0x1000，即 4KB，同样的。

2）vdev0vring1 是 vring1 的起始地址，地址长度也是 0x1000，即 4KB。这两个节点就是我们前面说的用于发送和接收消息的 vring。

3）vdev0buffer 子节点，起始地址为 0x10042000，地址长度为 0x4000，大小为 16KB，这段地址刚好落在 SRAM3 中，这就是设置的共享的内存区域。

vdev0vring0、vdev0vring1 和 vdev0buffer 只占用了 SRAM3 的前 24KB，SRAM3 有 64KB，并未使用完，所以如果有需要，也可以通过修改设备树和链接脚本来将 SRAM3 未使用到的地址用作其它功能。

- mcuram 子节点

mcuram 子节点起始地址是 0x30000000，地址长度为 0x40000，大小为 256KB，这段地址是 RAM aliases 里的 SRAM1 和 SRAM2 区域，因为 RAM aliases 和 SRAMs 的物理地址是一样的，所以也需要配置对于 A7“可见”的对应区域，这段区域对应的是 M4“可见”的 mcuram2 区域。因为 mcuram 和 mcuram2 的物理地址一样，所以这两段存储区域的功能是一样的，可以说 mcuram 是 mcuram2 的别名存储区域，这可能就是 RAM aliases 中 aliases（别名）的由来，要注意的是，在设备树中，mcuram 和 mcuram2 内存段定义必须是一致的。

- retram 子节点

retram 子节点的起始地址是 0x38000000，地址长度 0x10000 为 64KB，属于 RAM aliases 里的 RETRAM 区域，此区域和 BOOT 存储区域的 RETRAM 区域对应，它们是同一个物理地址。RETRAM 用于存放 M4 内核的中断向量表（中断向量表从 0x00000000 开始），默认情况下，RETRAM 起始地址为 0x38000000，它会重新映射到 0x00000000 以执行 M4 的代码。

以上 SRAM 地址配置，如下表 4.2.1.1 所示。

子节点	地址	大小	区域
mcuram2	0x10000000~0x10040000	256KB	M4 可见的 SRAM1+SRAM2
vdev0vring0	0x10040000~0x10041000	4KB	SRAM3
vdev0vring1	0x10041000~0x10042000	4KB	
vdev0buffer	0x10042000~0x10046000	16KB	
mcuram	0x30000000~0x30040000	256KB	A7 可见的 SRAM1+SRAM2
retram	0x38000000~0x38010000	64KB	A7 可见的 RETRAM

表 4.2.1.1 SRAM 地址划分

4.2.2 系统资源分配

打开内核源码的 stm32mp151.dtsi 设备树文件，找到如下地方：

```
mlahb {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        dma-ranges = <0x00000000 0x38000000 0x10000>,
                     <0x10000000 0x10000000 0x60000>,
                     <0x30000000 0x30000000 0x60000>;

        m4_rproc: m4@10000000 {
                compatible = "st,stm32mp1-m4";
                reg = <0x10000000 0x40000>,
                      <0x30000000 0x40000>,
                      <0x38000000 0x10000>;
                resets = <&scmi0_reset RST_SCMI0_MCU>;
                st,syscfg-holdboot = <&rcc 0x10C 0x1>;
                st,syscfg-tz = <&rcc 0x000 0x1>;
                st,syscfg-rsc-tbl = <&tamp 0x144 0xFFFFFFFF>;
                st,syscfg-copro-state = <&tamp 0x148 0xFFFFFFFF>;
                st,syscfg-pdds = <&pwr_mcu 0x0 0x1>;
                status = "disabled";

                m4_system_resources {
                        compatible = "rproc-srm-core";
                        status = "disabled";
                };
        };
};
```

以上设备树节点中有个 m4_system_resources 子节点（也就是 M4 的资源管理器），它是用于配置 M4 的外设资源的，其中 compatible 属性中的 rproc-srm-core 会匹配到内核源码源码的 drivers/remoteproc/rproc_srm_core.c 驱动文件，如下是 rproc_srm_core.c 文件的部分代码：

```
static const struct of_device_id rproc_srm_core_match[] = {
        { .compatible = "rproc-srm-core", },
        {},
};

MODULE_DEVICE_TABLE(of, rproc_srm_core_match);

static struct platform_driver rproc_srm_core_driver = {
        .probe = rproc_srm_core_probe,
        .remove = rproc_srm_core_remove,
```

```
        .driver = {
                .name = "rproc-srm-core",
                .of_match_table = of_match_ptr(rproc_srm_core_match),
        },
};

module_platform_driver(rproc_srm_core_driver);
```

当设备和驱动匹配成功以后，platform_driver 的 probe 成员变量所代表的函数 rproc_srm_core_probe()被执行，通过该函数实现注册 rproc 子设备（rproc 代表一个物理远程处理器设备，可以说是一个外设）。

打开内核源码的 stm32mp157-m4-srm.dtsi 设备树文件，如下，看到这部分代码，&m4_rproc 表示在前面的 m4_rproc 节点下追加内容，我们看追加了哪些内容：

```
&m4_rproc {
  m4_system_resources {
    #address-cells = <1>;
    #size-cells = <0>;

    m4_timers2: timer@40000000 {
            compatible = "rproc-srm-dev";
            reg = <0x40000000 0x400>;
            clocks = <&rcc TIM2_K>;
            clock-names = "int";
            status = "disabled";
       };
    /* 省略部分代码 */
     m4_adc: adc@48003000 {
            compatible = "rproc-srm-dev";
            reg = <0x48003000 0x400>;
            clocks = <&rcc ADC12>, <&rcc ADC12_K>;
            clock-names = "bus", "adc";
            status = "disabled";
      };

    /* 省略部分代码 */
    m4_ethernet0: ethernet@5800a000 {
            compatible = "rproc-srm-dev";
            reg = <0x5800a000 0x2000>;
            clock-names = "stmmaceth",
                          "mac-clk-tx",
                          "mac-clk-rx",
                          "ethstp",
                          "syscfg-clk";
            clocks = <&rcc ETHMAC>,
```

```
                    <&rcc ETHTX>,
                    <&rcc ETHRX>,
                    <&rcc ETHSTP>,
                    <&rcc SYSCFG>;
            status = "disabled";
    };
  };
};
```

以上代码配置的就是 M4 的系统资源，即 M4 配置了哪些外设，不过 status 属性都是 disabled，也就是虽然配置了外设，但是不使能外设。因为 stm32mp157d-atk.dtsi 文件 include 了 stm32mp157-m4-srm.dtsi 文件，stm32mp157d-atk.dts 文件又 include 了 stm32mp157d-atk.dtsi 文件，所以我们可以直接在 stm32mp157d-atk.dtsi 或 stm32mp157d-atk.dts 设备树文件中选择使能 M4 的某个外设。

这里说明一下，M4 和 A7 有些外设是共享的，例如 GPIO 是共享的资源，如果此 GPIO 没有复用做其它功能，只是单纯当做普通的 IO 使用，那么 A7 和 M4 都可以访问这些资源。例如在 stm32mp157d-atk.dtsi 下有配置了一个蜂鸣器和两个 led 节点，它们使用的是 GPIO 功能，这些节点是给 A7 用的，但 M4 也可以使用：

```
leds {
   compatible = "gpio-leds";

   led1 {
       label = "sys-led";
       gpios = <&gpioi 0 GPIO_ACTIVE_LOW>;
       linux,default-trigger = "heartbeat";
       default-state = "on";
       status = "okay";
   };

   led2 {
       label = "user-led";
       gpios = <&gpiof 3 GPIO_ACTIVE_LOW>;
       linux,default-trigger = "none";
       default-state = "on";
       status = "okay";
   };

   beep {
       label = "beep";
       gpios = <&gpioc 7 GPIO_ACTIVE_LOW>;
       default-state = "off";
   };
};
```

如果是具有单选功能的外设，也就是这些外设要么只能单独给 A7 使用，要么只能单独给 M4 使用，如果 A7 和 M4 都一起使用该外设，就会存在资源争用问题，某一方就会出现异常（主处理器有一定的优先权，一般是协处理器这边出现异常），例如，如果 A7 和 M4 一起占用 ADC1 来采集数据，这个时候 M4 这边采集到的数据会不准确，可能采集不到数据而显示 0。

对于单选的外设，如果 A7 要使用该外设的话，在设备树下一定要配置 A7 对应的外设节点，如果该外设要给 M4 使用的话，设备树下可以不必配置 M4 相关的节点，只需要在固件中配置该外设即可（也就是在裸机程序中配置），当 A7 加载和启动固件后，M4 就可以使用该外设了。如果已经在设备树下配置 A7 对应的某个外设节点，M4 想使用此外设的话，是否需要将设备树下 A7 占用的相关节点注释掉呢？例如在 stm32mp157d-atk.dtsi 设备树下有如下节点：

```
adc1_in6_pins_b: adc1-in6 {
   pins {
       pinmux = <STM32_PINMUX('A', 5, ANALOG)>;
   };
};

&adc {
  /* ADC1 & ADC2 common resources */
  pinctrl-names = "default";
  pinctrl-0 = <&adc1_in6_pins_b>;
      vdd-supply = <&vdd>;
      vdda-supply = <&vdd>;
      vref-supply = <&vdd>;

  status = "okay";

  adc1: adc@0 {
      /* private resources for ADC1 */
      st,adc-channels = <19>;
      st,min-sample-time-nsecs = <10000>;
      status = "okay";
  };
};
```

以上代码段中表示将 ADC1 分配给 A7 使用，如果 M4 要使用的话，这段代码可以不用注释掉，只要确保 Linux 系统运行以后 A7 不去操作 ADC1，那么，当加载和运行 M4 的固件后（固件中已经配置了 ADC1），M4 就可以使用 ADC1 来采集数据了。但是，如果此时 A7 去操作 ADC1 的话，M4 这边 ADC1 采集到的数据就会不准确。所以，要么将设备树下 A7 占用 ADC1 的节点注释掉，这样 A7 就永远无法使用 ADC1 了，要么保留 A7 占用的 ADC1 节点，只需要保证 Linux 系统启动后，A7 不去操作 ADC1，这样 M4 就可以正常使用 ADC1 了。如果采用前者的方法，将 A7 占用 ADC1 的相关节点注释掉，可以修改如下：

```
/*
   adc1_in6_pins_b: adc1-in6 {
       pins {
           pinmux = <STM32_PINMUX('A', 5, ANALOG)>;
```

```
        };
    };
*/
/*
&adc {
  //  * ADC1 & ADC2 common resources *
    pinctrl-names = "default";
    pinctrl-0 = <&adc1_in6_pins_b>;
        vdd-supply = <&vdd>;
        vdda-supply = <&vdd>;
        vref-supply = <&vdd>;

    status = "okay";

    adc1: adc@0 {
    //    * private resources for ADC1 *
        st,adc-channels = <19>;
        st,min-sample-time-nsecs = <10000>;
        status = "okay";
    };
};
*/
&m4_adc {
        vref-supply = <&vrefbuf>;
        status = "okay";            /* 使能 M4 的 ADC */
};
```

也就是将 A7 占用的 ADC1 部分注释掉。后面&m4_adc 节点部分是手动添加的，这段可以添加也可以不添加，不过按照 ST 的标准来，最好要添加，设备树 stm32mp157c-dk2-m4-examples.dts 是模板文件，我们修改设备树的时候，可以参考模板文件，以上修改的&m4_adc 节点就是参考此文件来写的。修改好设备树以后，执行如下指令重新编译设备树：

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- dtbs
```

再将编译出来的 stm32mp157d-atk.dtb 文件拷贝到开发板文件系统的/boot 目录下，替换掉以前的设备树二进制文件，再执行 sync 指令以同步缓存，然后重启开发板，重新进入 Linux 操作系统后，A7 就不能再去操作 ADC1 了，当加载和启动 M4 固件后，M4 可单独访问 ADC1。关于这些操作，大家在后期操作的时候可以多进行实践。

4.3 Linux 下 Remoteproc 相关 API

内核源码根目录的 Documentation/remoteproc.txt 文本有对 Remoteproc 的说明，可以查阅此帮助文档获得一些信息。在 Linux 内核源码的 drivers/remoteproc 目录下就是 Remoteproc 驱动文件，如下图 4.3.1 所示，在这些文件中，一般比较关心的是 remoteproc_core.c、remoteproc_elf_loader.c、remoteproc_core.c、remoteproc_sysfs.c、remoteproc_virtio.c 和 stm32_rproc.c 这几个文件。stm32_rproc.c 文件是 ST 官方编写的部分驱动程序，而其它文件主要就是注册设备、初始化调试目录、为 stm32_rproc.c 文件驱动提供接口等。

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/remoteproc$ ls
built-in.a              qcom_common.h       remoteproc_core.c        rproc_srm_core.c
da8xx_remoteproc.c      qcom_q6v5_adsp.c    remoteproc_core.o        rproc_srm_core.h
imx_rproc.c             qcom_q6v5.c         remoteproc_debugfs.c     rproc_srm_core.o
Kconfig                 qcom_q6v5.h         remoteproc_debugfs.o     rproc_srm_dev.c
keystone_remoteproc.c   qcom_q6v5_mss.c     remoteproc_elf_loader.c  rproc_srm_dev.o
Makefile                qcom_q6v5_pas.c     remoteproc_elf_loader.o  stm32_rproc.c
modules.builtin         qcom_q6v5_wcss.c    remoteproc_internal.h    stm32_rproc.o
modules.order           qcom_sysmon.c       remoteproc_sysfs.c       st_remoteproc.c
omap_remoteproc.c       qcom_wcnss.c        remoteproc_sysfs.o       st_slim_rproc.c
omap_remoteproc.h       qcom_wcnss.h        remoteproc_virtio.c      wkup_m3_rproc.c
qcom_common.c           qcom_wcnss_iris.c   remoteproc_virtio.o
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/remoteproc$
```

图 4.3.1 Remoteproc 驱动源码

在 M4 工程的 openampMiddlewares\Third_Party\OpenAMP 目录下也可以看到 Remoteproc 驱动相关的文件：remoteproc_virtio.c，大家感兴趣的也可以分析这部分的代码。如下图 4.3.2 所示。

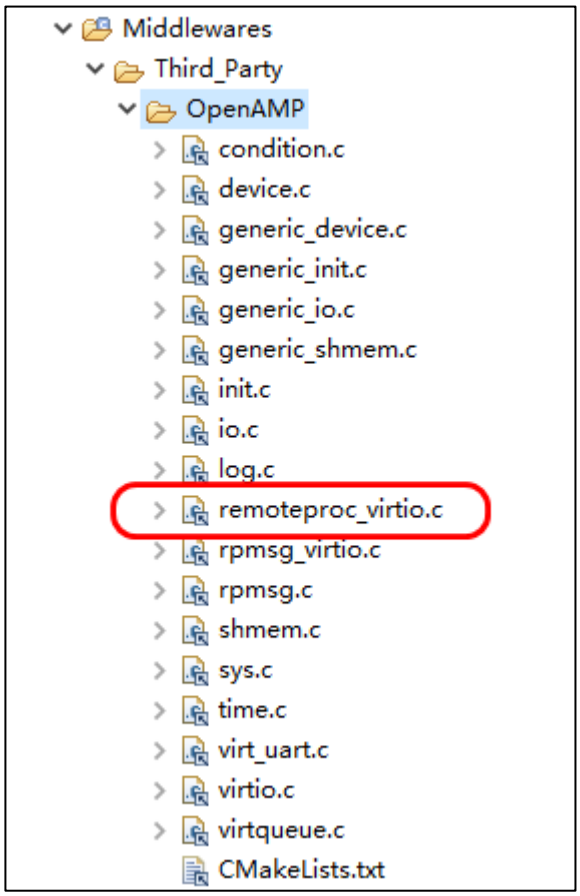


图4.3.2 M4工程目录

下面我们来了解几个和 Remoteproc 框架相关的函数，首先打开 Linux 内核源码 drivers/remoteproc/remoteproc_core.c 文件，分别查看如下表 4.3.1.1 所示函数：

函数	描述
remoteproc_init()	初始化 Remoteproc 实例
remoteproc_exit()	退出 Remoteproc 实例
rproc_alloc()	分配远程处理器句柄
stm32_rproc_parse_dt()	获取设备树中的属性

rproc_boot()	启动远程处理器
rproc_shutdown()	关闭远程处理器
rproc_add()	注册远程处理器
stm32_rproc_request_mbox()	为远程处理器申请邮箱
stm32_rproc_probe()	在 platform 驱动中，设备和驱动匹配成功后，此函数会执行

表 4.3.1.1 几个重要的 API 函数

Remoteproc 框架通过 remoteproc_init()函数初始化远程处理器环境，其根据固件资源表的配置信息来建立远程固件所需的运行环境，如配置固件所需的物理存储地址、注册所支持的 Virtio 设备等。

rproc_alloc()函数根据远程处理器（即 M4）的名称和固件来分配一个远程处理器句柄，并完成远程设备的部分初始化，此时远程处理器的初始状态处于离线状态。stm32_rproc_parse_dt()函数用于获取设备树中的属性，目的是完成远程处理器的配置。接下来调用 rproc_add()函数来注册远程处理器，rproc_add()函数内部会调用 rproc_boot()函数来启动远程处理器，此时远程处理器为在线状态。stm32_rproc_request_mbox()函数用于为远程处理器申请邮箱，A7 和 M4 的核间通信，通过邮箱机制来通知已经有数据在共享内存中了，可以进行读取操作了。以上 API 的关系，我们可以使用以下图 4.3.3 来表示：

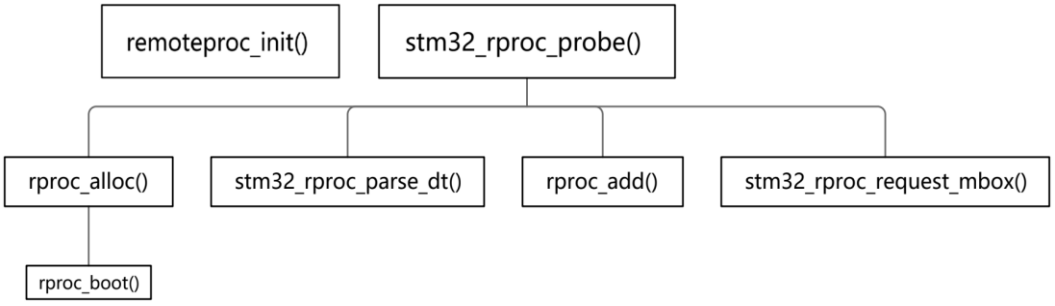


图 4.3.3 API 函数的调用关系

下面我们来分析其中的几个 API 函数。

4.3.1 rproc 结构体

关于这部分驱动，可以做简单了解即可，下面我们先看和远程处理器相关的 rproc 结构体，打开 include/linux/remoteproc.h 文件，找到如下示例代码：

```
struct rproc {                    /* rproc 代表一个物理远程处理器设备 */
  struct list_head node;          /* rproc 对象的节点 */
  struct iommu_domain *domain;    /* iommu 域 */
  const char *name;               /* rproc 的可读名称，即远程处理器的名称 */
  char *firmware;                 /* 要加载的固件文件名字 */
  void *priv;                     /* 芯片厂商保存自己私有数据的指针 */
/* rproc_ops 结构体是芯片厂商做好的，用于加载/启动/停止固件 */
  struct rproc_ops *ops;
  struct device dev;              /* 用于引用计数和常见 Remoteproc 行为的虚拟设备 */
  atomic_t power;                 /* 需要此 rproc 启动的用户的引用计数 */
  unsigned int state;             /* 设备状态 */
  struct mutex lock;              /* 保护 rproc 并发操作的锁 */
```

```
  struct dentry *dbg_dir;    /* 这个 rproc 设备的 debugfs 目录 */
  struct list_head traces;   /* 跟踪缓冲区列表 */
  int num_traces;            /* 跟踪缓冲区的数量 */
  struct list_head carveouts;/* 物理连续内存分配列表 */
  struct list_head mappings; /* 我们启动的 iommu 映射列表，关闭时需要 */
  u32 bootaddr;              /* 加载的首地址 */
  struct list_head rvdevs;   /* 远程 Virtio 设备列表 */
  struct list_head subdevs;  /* 子设备列表，跟踪运行状态 */
  struct idr notifyids;      /* idr 用于动态分配 rproc 范围内的唯一通知 ID */
  int index;                 /* 这个 rproc 设备的索引 */
  struct work_struct crash_handler;   /* 处理崩溃的工作队列 */
  unsigned int crash_cnt;             /* 崩溃计数器 */
  bool recovery_disabled;             /* 如果恢复被禁用，则标记该状态 */
  int max_notifyid;                   /* 最大分配的通知 ID */
  struct resource_table *table_ptr;   /* 指向有效资源表的指针 */
  struct resource_table *cached_table;/* 资源表的副本 */
  size_t table_sz;                /* cached_table 的大小 */
  bool has_iommu;                 /* 指示远程处理器是否在 MMU 后面的标志 */
  bool auto_boot;                 /* 指示是否应自动启动远程处理器的标志 */
  struct list_head dump_segments;/* 固件中的段列表 */
  int nb_vdev;                    /* 当前由 rproc 处理的 Virtio 设备数量 */
  bool early_boot;               /* 固件加载标志位(0 表示加载，1 表示没有加载)*/
};
```

对于写驱动的人来说，rproc 是 Remoteproc 框架的一个重要的结构体，后面的函数只需要定义一个结构体指针，然后通过指针来配置这个结构体里面的成员变量，就可以达到一定的目的。

4.3.2 初始化 Remoteproc 实例

remoteproc_init()函数用于初始化 Remoteproc 实例，即初始化远程处理器环境，其定义如下：

```
static int __init remoteproc_init(void)
{
       rproc_init_sysfs();
       rproc_init_debugfs();

       return 0;
}
subsys_initcall(remoteproc_init);
```

rproc_init_sysfs()函数在 remoteproc_sysfs.c 文件中定义，它为 sysfs 文件系统注册 Remoteproc 设备类，它提供了一个用于控制远程处理器的 sysfs 接口，sysfs 其实也是个文件系统，挂载在/sys 下，如对此文件系统感兴趣的可自行了解，这里就不占用过多篇幅介绍了。

rproc_init_debugfs()函数在 remoteproc_debugfs.c 文件中定义，用于创建调试（debugfs）目录，debugfs 其实也是一个虚拟文件系统，它是内核空间与用户空间的接口，方便开发人员调试和向用户空间导出内核空间数据，一般发行版的内核都已经默认将 debugfs 和 sysfs 编译到了内

核，并将 debugfs 自动挂载到文件系统的/sys/kernel/debug 目录下，我们进入到开发板 Linux 文件系统该目录下，如下图 4.3.2.1 所示：

```
root@ATK-MP157:/sys/kernel/debug# ls
49000000.usb-otg  cec                dmaengine           gc          memcg_slabinfo
allocators        clear_warn_once    dri                 gpio        mmc0
asoc              clk                dynamic_debug       hid         mmc1
bdi               device_component   edt_ft5x06          ieee80211   mmc2
block             devices_deferred   extfrag             iio         mtd
bluetooth         dma_buf            fault_around_bytes  memblock    opp
root@ATK-MP157:/sys/kernel/debug#
```

图 4.3.2.1 debugfs 文件系统

可以看到有 remoteproc 文件夹，进入文件夹发现如下有一个 remoteproc0 目录，如下图 4.3.2.2 所示。

```
root@ATK-MP157:/sys/kernel/debug/remoteproc# ls
remoteproc0
root@ATK-MP157:/sys/kernel/debug/remoteproc# cd remoteproc0/
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# ls
carveout_memories  crash  name  recovery  resource_table
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0#
```

图 4.3.2.2

进入到 remoteproc0 目录下，可以看到有如下文件：carveout_memories、crash、name、recovery 和 resource_table。这几个文件说明如下：

1） name 中的内容是处理器的名字，内容为 m4，也就是 M4 协处理器；

2） resource_table 记录了 M4 的资源信息（资源表）；

3） carveout_memories 记录了 M4 的内存分配情况（内存是在设备树下配置的）；

4） recovery 文件的内容可以是 enabled、disabled 或 recovery 三者之一，用于控制恢复机制的行为。enabled 表示远程处理器将在崩溃时可以自动恢复，disabled 表示远程处理器在崩溃时将保持崩溃状态，recovery 表示如果远程处理器处于崩溃状态，此功能将触发立即恢复，而不用手动更改或检查恢复状态（启用/禁用）。这三种模式在调试的时候很有用，默认为 enabled，如果要修改为 disabled，执行如下指令即可：

```
echo disabled >/sys/kernel/debug/remoteproc/remoteproc0/recovery
```

5） crash 是记录系统崩溃时候的有关信息。

如果 A7 没有使用 Remoteproc 加载固件，在 carveout_memories 和 resource_table 文件中是没有什么内容的，如下操作所示，使用 cat 命令查看文件是没有什么内容的，如下图 4.3.2.3 所示：

```
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat name
m4
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat carveout_memories
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat resource_table
No resource table found
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0#
```

图 4.3.2.3 查看文件内容

如果 A7 使用 Remoteproc 加载固件后，固件资源被解析，关联的资源信息会被记录到以上文件中，后面我们在加载固件后再来查看这些文件的内容。

4.3.3 退出 Remoteproc 实例

remoteproc_exit()函数用于退出 Remoteproc 示例，即退出远程处理器环境，其定义如下：

```
static void __exit remoteproc_exit(void)
{
	ida_destroy(&rproc_dev_index);

	rproc_exit_debugfs();
	rproc_exit_sysfs();
}
module_exit(remoteproc_exit);
```

释放 Remoteproc 和初始化 Remoteproc 相反，也就是退出 debugfs 和 sysfs 目录，并删除关联的资源。

4.3.4 启动远程处理器

rproc_boot() 函数用于启动远程处理器，其定义如下：

```
1  /**
2   * @brief       启动远程处理器（即加载其固件，打开电源，...）
3   * @param       rproc:指针变量，远程处理器的结构体
4   * @retval      成功返回 0，否则返回适当的错误值。
5   */
6  int rproc_boot(struct rproc *rproc)
7  {
8      const struct firmware *firmware_p = NULL;
9      struct device *dev;
10     int ret;
11
12     if (!rproc) {
13           pr_err("invalid rproc handle\n");
14           return -EINVAL;
15     }
16
17     dev = &rproc->dev;
18     /* 获取互斥锁，可中断 */
19     ret = mutex_lock_interruptible(&rproc->lock);
20     if (ret) {
21          dev_err(dev, "can't lock rproc %s: %d\n", rproc->name, ret);
22           return ret;
23     }
24    /* 如果处理器的状态是已经删除，则解锁互斥锁 */
25     if (rproc->state == RPROC_DELETED) {
26           ret = -ENODEV;
27           dev_err(dev, "can't boot deleted rproc %s\n", rproc->name);
28           goto unlock_mutex;
29     }
30   /* 如果 rproc 已经通电，则跳过引导过程 */
```

```
31      if (atomic_inc_return(&rproc->power) > 1) {
32              ret = 0;
33              goto unlock_mutex;
34      }
35
36      dev_info(dev, "powering up %s\n", rproc->name);
37
38      if (!rproc->early_boot) {
39              /* 如果没有加载固件的话，则加载固件到内存 */
40              ret = request_firmware(&firmware_p, rproc->firmware, dev);
41              if (ret < 0) {
42                      dev_err(dev, "request_firmware failed: %d\n", ret);
43                      goto downref_rproc;
44              }
45      } else {
46              /* 如果已经加载了固件，则将固件名称设置为 null 作为未知 */
47              kfree(rproc->firmware);
48              rproc->firmware = NULL;
49      }
50    /* 获取固件并用固件启动远程处理器。 */
51     ret = rproc_fw_boot(rproc, firmware_p);
52    /* 释放固件 */
53     release_firmware(firmware_p);
54
55    downref_rproc:
56          if (ret)
57                  atomic_dec(&rproc->power); /* 递减原子变量 */
58    unlock_mutex:
59          mutex_unlock(&rproc->lock);       /* 释放互斥锁 */
60          return ret;
61  }
62  EXPORT_SYMBOL(rproc_boot);
```

启动远程处理器的代码比较简单，我们来简单分析这段代码：

第 12~15 行，判断远程处理器的结构体是否为真，其实也就是检查远程处理器是否是准备好了；

第 19 行，获取互斥锁操作，目的就是操作设备的时候，先上锁，其它线程将阻塞等待而不能同时操作设备（防止多个线程同时访问共享数据）；

第 24~29 行，如果远程处理器状态是已经删除，则解锁互斥锁；

第 30~34 行，如果远程处理器已经通电，则跳过引导过程，则该函数立即返回 0（成功）。

第 36 行，打印“powering up m4”，这里处理器的名字是 m4，启动固件后会打印这句话。

第 38~45，early_boot 是固件加载标记，如果没有加载固件的话，则加载固件，firmware_p 是指向固件名字的指针变量；

第 51 行，rproc_fw_boot 函数会检查固件是否合法、使能 iommu（地址映射）、从固件中获取到 boot 启动地址、从固件加载资源表、核心转储段列表等信息，然后启动远程处理器的固件资源（包括 RSC_CARVEOUT、RSC_DEVMEM、RSC_VDEV 和 RSC_TRACE 等资源条目），并将内存分配情况记录在 carveout_memories 文件中，将资源表信息记录在 resource_table 文件中。如果获取到固件的话，会打印固件的名字和大小。例如测试的时候，会打印类似语句：“Booting fw image RPMsg_TEST_CM4.elf, size 5582740”，这里获取固件的名字是 RPMsg_TEST_CM4.elf，文件大小为 5582740bit（约为 5.4MB）。如果未获取到固件的话，则打印“Synchronizing with early booted co-processor”。

第 52 行，释放固件，一旦驱动程序处理完固件，它就可以调用 release_firmware 来释放固件映像和任何相关资源。

4.3.5 关闭远程处理器

前面使用 rproc_boot()启动远程处理器，这里 rproc_shutdown()函数用于关闭远程处理器，其定义如下：

```
1   /**
2   * @brief       关闭远程处理器
3   * @param       rproc: 指针变量，远程处理器的结构体
4   * @retval      无。
5   * @note        如果 rproc 仍在被其它用户使用，则此函数只会
6   *              减少电源引用计数并退出，无需真正关闭设备电源。
7   */
8   void rproc_shutdown(struct rproc *rproc)
9   {
10      struct device *dev = &rproc->dev;
11      int ret;
12      /* 获取互斥锁，可中断 */
13      ret = mutex_lock_interruptible(&rproc->lock);
14      if (ret) {
15            dev_err(dev, "can't lock rproc %s: %d\n", rproc->name, ret);
16            return;
17      }
18
19      /* 如果仍然需要远程过程，请退出 */
20      if (!atomic_dec_and_test(&rproc->power))
21            goto out;
22      /* Virtio 设备被 rproc_stop() 销毁 */
23      ret = rproc_stop(rproc, false);
24      if (ret) {
25            atomic_inc(&rproc->power);
26            goto out;
27      }
28
29      /* 清理所有获得的资源 */
```

```
30      rproc_resource_cleanup(rproc);
31
32      rproc_disable_iommu(rproc);
33
34      /* 释放资源表 */
35      kfree(rproc->cached_table);
36      rproc->cached_table = NULL;
37      rproc->table_ptr = NULL;
38  out:
39      mutex_unlock(&rproc->lock); /* 释放互斥锁 */
40  }
41  EXPORT_SYMBOL(rproc_shutdown);
```

关闭远程处理器的过程和启动远程处理器的过程相反，我们看看上面代码做了哪些操作：

第 23 行，关闭远程处理器，其中 rproc_stop()函数会停止远程处理器的任何子设备，不访问已关联的资源表，关闭远程处理器，然后打印:“stopped remote processor m4”,在后面调试的时候我们可以多关注打印的信息。

第 30 行，清理所有获得的资源，清理调试跟踪条目，清除 carveout_memories 文件等。

第 32 行，这里补充一下，IOMMU 主要是将虚拟内存地址映射为物理内存地址，让实体设备可以在虚拟的内存环境中工作，在加载和获取固件的时候已经做了这步操作了。这里是注销设备、注销远程处理器的操作域（iommu 域）。

第 35~37，释放资源表。

这里要注意的是，如果远程处理器仍在被其它用户使用，则此函数只会减少电源引用计数并退出，而不会真正关闭设备电源。

4.3.6 分配远程处理器句柄

在远程处理器初始化期间会调用 rproc_alloc()函数，该函数会根据远程处理器的名称和固件来分配一个远程处理器句柄，并完成远程设备的部分初始化。在使用此函数分配新的远程处理器句柄后，远程处理器是还未注册的，后期应该调用 rproc_add()函数以完成远程处理器的注册。运行此函数后，远程处理器的初始状态处于离线状态。

```
/**
* @brief       分配一个远程处理器句柄
* @dev         dev:底层设备
*              name:此远程处理器的名称
*              ops:处理程序（主要是启动/停止）
*              firmware:要加载的固件文件的名称，可以为空
*              len:rproc 驱动程序所需的私有数据长度（字节）
* @retval      成功时返回新的 rproc，失败时返回 NULL。
* @note        注意：如果要释放 rproc_add()函数分配的处理器，使用 rproc_free()
*/

struct rproc *rproc_alloc(struct device *dev, const char *name,
                          const struct rproc_ops *ops,
                          const char *firmware, int len)
```

```
{
  struct rproc *rproc;
  char *p, *template = "rproc-%s-fw";
  int name_len;

  if (!dev || !name || !ops)
  return NULL;
  /* 如果调用方没有传入固件名称，则构造一个默认名称 */
  if (!firmware)
  {
    name_len = strlen(name) + strlen(template) - 2 + 1;
    p = kmalloc(name_len, GFP_KERNEL);
    if (!p)
         return NULL;
    snprintf(p, name_len, template, name);
  }
  /*
   * 如果调用方有传入固件名称,则分配内存空间,
   * 并将该固件名称字符串拷贝到所分配的地址空间中
   */
  else
  {
    p = kstrdup(firmware, GFP_KERNEL);
    if (!p)
         return NULL;
  }
  /* 调用 kzalloc 分配一个内存空间 */
  rproc = kzalloc(sizeof(struct rproc) + len, GFP_KERNEL);
  if (!rproc)
  {
    kfree(p);
    return NULL;
  }
  /* 根据给定的 ops 来分配一个内存空间 */
  rproc->ops = kmemdup(ops, sizeof(*ops), GFP_KERNEL);
  if (!rproc->ops)
  {
    kfree(p);
    kfree(rproc);
    return NULL;
  }
  rproc->firmware = p;    /* 要加载的固件文件名字 */
  rproc->name = name;     /* 远程处理器器的名称 */
```

```
rproc->priv = &rproc[1];/* 私有数据 */
rproc->auto_boot = true;/* 自动启动远程处理器 */
/* 对设备进行初始化，主要是设备引用计数器、信号量、设备访问锁等字段的初始化 */
device_initialize(&rproc->dev);
rproc->dev.parent = dev;
rproc->dev.type = &rproc_type;
rproc->dev.class = &rproc_class;
rproc->dev.driver_data = rproc;

/* 分配唯一的设备索引和名称 */
rproc->index = ida_simple_get(&rproc_dev_index, 0, 0, GFP_KERNEL);
if (rproc->index < 0)
{
  dev_err(dev, "ida_simple_get failed: %d\n", rproc->index);
  put_device(&rproc->dev); /* 对设备引用次数减一 */
  return NULL;
}

dev_set_name(&rproc->dev, "remoteproc%d", rproc->index);

atomic_set(&rproc->power, 0);

/* Default to ELF loader if no load function is specified */
/* 若未指定加载的文件名，则默认是 ELF 文件 */
if (!rproc->ops->load)
{
  /* 加载 ELF 文件，即加载固件盗内存中 */
  rproc->ops->load = rproc_elf_load_segments;
  /* 加载固件资源表 */
  rproc->ops->parse_fw = rproc_elf_load_rsc_table;
  /* 查找已经加载的资源表 */
  rproc->ops->find_loaded_rsc_table = rproc_elf_find_loaded_rsc_table;
  /* 检查 ELF 固件映像 */
  rproc->ops->sanity_check = rproc_elf_sanity_check;
  /* 获取处理器的启动地址 */
  rproc->ops->get_boot_addr = rproc_elf_get_boot_addr;
}

mutex_init(&rproc->lock);

idr_init(&rproc->notifyids);
/* 初始化对应的双向链表 */
INIT_LIST_HEAD(&rproc->carveouts);
```

```
  INIT_LIST_HEAD(&rproc->mappings);
  INIT_LIST_HEAD(&rproc->traces);
  INIT_LIST_HEAD(&rproc->rvdevs);
  INIT_LIST_HEAD(&rproc->subdevs);
  INIT_LIST_HEAD(&rproc->dump_segments);
  /* 初始化一个工作队列 */
  INIT_WORK(&rproc->crash_handler, rproc_crash_handler_work);
  /* 远程处理器的初始状态是离线的状态 */
  rproc->state = RPROC_OFFLINE;

  return rproc;
}
  EXPORT_SYMBOL(rproc_alloc);
```

4.3.7 注册远程处理器

rproc_add()函数用于注册一个远程处理器（rproc），rproc_add()函数把 rproc 结构体注册进 Remoteproc 框架，就能够为上一层提供接口去加载固件。使用 rproc_add()函数注册 rproc 结构体后，当需要删除时可以使用 rproc_del()函数。

```
1   /**
2     * @brief       注册一个远程处理器
3     * @param       rproc:远程处理器的结构体
4     * @retval      成功返回 0，否则返回适当的错误代码。
5     * @note        注意：此函数会启动一个异步固件加载上下文，
6     *              它将寻找 rproc 的固件支持的 virtio 设备。
7     */
8   int rproc_add(struct rproc *rproc)
9   {
10    struct device *dev = &rproc->dev;
11    int ret;
12    /* 调用 device_add 增加一个设备对象 */
13    ret = device_add(dev);
14    if (ret < 0)
15          return ret;
16
17    dev_info(dev, "%s is available\n", rproc->name);
18
19    /* 创建 debugfs 条目 */
20    rproc_create_debug_dir(rproc);
21
22    /* 添加资源管理器设备 */
23    ret = devm_of_platform_populate(dev->parent);
24    if (ret < 0)
25          return ret;
```

```
26    if (rproc->early_boot) {
27    /* 如果已经加载固件，则无需等待固件,只需处理关联资源并启动子设备 */
28          ret = rproc_boot(rproc);
29          if (ret < 0)
30                return ret;
31    } else if (rproc->auto_boot) {
32    /* 如果 rproc 被标记为永远在线，则请求它启动 */
33          ret = rproc_trigger_auto_boot(rproc);
34          if (ret < 0)
35                return ret;
36    }
37
38    /* 暴露给 rproc_get_by_phandle 用户 */
39    mutex_lock(&rproc_list_mutex);
40    list_add(&rproc->node, &rproc_list);
41    mutex_unlock(&rproc_list_mutex);
42
43    return 0;
44  }
45  EXPORT_SYMBOL(rproc_add);
```

rproc_add()函数调用了 rproc_boot()函数，我们分析其启动过程：

第 12 行，调用 device_add()增加一个设备对象，注册一个远程处理器；

第 23 行，添加资源管理器设备，dev 是从设备树请求的设备，devm_of_platform_populate()函数会调用 of_platform_populate()函数，of_platform_populate()函数会遍历对应的设备树节点，并将设备树节点转化为一个 platform device，因为在 platform driver 中，最终是 platform device 和 platform driver 匹配，最后可以完成 probe 过程；

第 26~36 行，如果已经加载了固件，则执行 rproc_boot()函数，表示处理关联资源并启动处理器，如果远程处理器被标记为永远在线，则请求启动它；

第 39~41 行，先调用互斥锁锁定 rproc_list_mutex 互斥对象，再使用 list_add 将处理器设备节点添加到列表中，然后再解锁 rproc_list_mutex 互斥对象，这样操作后，rproc_get_by_phandle()可通过设备节点的 phandle 查找远程处理器。

前面的第 20 行，创建 debugfs 条目，rproc_create_debug_dir()函数在 remoteproc_debugfs.c 文件中定义，如下：

```
void rproc_create_debug_dir(struct rproc *rproc)
{
  struct device *dev = &rproc->dev;

  if (!rproc_dbg)
        return;

  rproc->dbg_dir = debugfs_create_dir(dev_name(dev), rproc_dbg);
  if (!rproc->dbg_dir)
        return;
```

```
    debugfs_create_file("name", 0400, rproc->dbg_dir,
                        rproc, &rproc_name_ops);
    debugfs_create_file("recovery", 0400, rproc->dbg_dir,
                        rproc, &rproc_recovery_ops);
    debugfs_create_file("crash", 0200, rproc->dbg_dir,
                        rproc, &rproc_crash_ops);
    debugfs_create_file("resource_table", 0400, rproc->dbg_dir,
                        rproc, &rproc_rsc_table_ops);
    debugfs_create_file("carveout_memories", 0400, rproc->dbg_dir,
                        rproc, &rproc_carveouts_ops);
}
```

可以看到，rproc_create_debug_dir()函数通过 debugfs_create_file()创建了 4 个 debugfs 文件：name、recovery、crash、resource_table 和 carveout_memories，这 4 个文件我们在前面已经了解过了。看到的 rproc_name_ops、rproc_recovery_ops 等类似 XXX_ops 结构的其实是回调函数结构体，这些回调函数结构体中的函数会对文件进行打开、释放、读文件等操作。例如 rproc_rsc_table_ops 如下：

```
static const struct file_operations rproc_rsc_table_ops = {
        .open           = rproc_rsc_table_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = single_release,
};
```

open 处理程序会执行 rproc_rsc_table_open()函数：

```
static int rproc_rsc_table_open(struct inode *inode, struct file *file)
{
    return single_open(file, rproc_rsc_table_show, inode->i_private);
}
```

rproc_rsc_table_open()函数会执行 single_open()函数，single_open()调用 rproc_rsc_table_show()函数，rproc_rsc_table_show()函数的内容，如下：

```
1   /**
2       * @brief      通过 debugfs 打印出资源表内容。
3       * @param      seq:序列文件接口指针
4       * @retval     成功返回 0，否则返回适当的错误代码。
5       */
6   static int rproc_rsc_table_show(struct seq_file *seq, void *p)
7   {
8     static const char * const types[] = {"carveout","devmem","trace", "vdev"};
9     struct rproc *rproc = seq->private;
10    struct resource_table *table = rproc->table_ptr;
11    struct fw_rsc_carveout *c;
12    struct fw_rsc_devmem *d;
13    struct fw_rsc_trace *t;
```

```
14    struct fw_rsc_vdev *v;
15    int i, j;
16
17    if (!table) {
18    seq_puts(seq, "No resource table found\n");
19    return 0;
20    }
21
22    for (i = 0; i < table->num; i++) {
23    int offset = table->offset[i];
24    struct fw_rsc_hdr *hdr = (void *)table + offset;
25    void *rsc = (void *)hdr + sizeof(*hdr);
26
27    switch (hdr->type) {
28      case RSC_CARVEOUT:
29          /*省略部分代码 */
30          break;
31      case RSC_DEVMEM:
32          /*省略部分代码 */
33          break;
34      case RSC_TRACE:
35          /*省略部分代码 */
36          break;
37      case RSC_VDEV:
38       v = rsc;
39       seq_printf(seq, "Entry %d is of type %s\n", i, types[hdr->type]);
40
41          seq_printf(seq, "  ID %d\n", v->id);
42          seq_printf(seq, "  Notify ID %d\n", v->notifyid);
43          seq_printf(seq, "  Device features 0x%x\n", v->dfeatures);
44          seq_printf(seq, "  Guest features 0x%x\n", v->gfeatures);
45          seq_printf(seq, "  Config length 0x%x\n", v->config_len);
46          seq_printf(seq, "  Status 0x%x\n", v->status);
47          seq_printf(seq, "  Number of vrings %d\n", v->num_of_vrings);
48          seq_printf(seq, "  Reserved (should be zero) [%d][%d]\n\n",
49                      v->reserved[0], v->reserved[1]);
50
51          for (j = 0; j < v->num_of_vrings; j++) {
52           seq_printf(seq, "  Vring %d\n", j);
53           seq_printf(seq, "    Device Address 0x%x\n", v->vring[j].da);
54           seq_printf(seq, "    Alignment %d\n", v->vring[j].align);
55         seq_printf(seq, "    Number of buffers %d\n", v->vring[j].num);
56          seq_printf(seq, "    Notify ID %d\n", v->vring[j].notifyid);
```

```
57          seq_printf(seq, "    Physical Address 0x%x\n\n",
58                     v->vring[j].pa);
59          }
60          break;
61      default:
62         seq_printf(seq, "Unknown resource type found: %d [hdr: %pK]\n",
63                     hdr->type, hdr);
64          break;
65      }
66    }
67    return 0;
68  }
```

seq_file 是序列文件接口，seq_file 可以将 Linux 内核里面常用的数据结构通过文件导出到用户空间，如果我们读取或者打开 Linux 文件系统/sys/kernel/debug/remoteproc/remoteproc0 下的 resource_table 文件，那么就会执行第 37 行~60 行的代码，即打印 RSC_VDEV 资源条目信息（包括 Virtio 设备 ID、Virtio 功能、Virtio 配置空间、vrings 信息等）。假设此时已经加载固件了，在 Linux 文件系统的/sys/kernel/debug/remoteproc/remoteproc0 目录下执行如下指令：

```
cat resource_table
```

打印如下图 4.3.7.1 所示信息，可以看到 Virtio RPMsg 设备 ID 为 7，vring 个数是 2，每个 vring 有 16 个 rpmsg buffer，这些信息我们在前面分析资源表的时候已经确定下来了，其中 vring0 和 vring1 的地址和前面我们分析 stm32mp157d-atk.dtsi 设备树的时候看到的配置一样：

```
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat resource_table
Entry 0 is of type vdev
  ID 7
  Notify ID 0
  Device features 0x1
  Guest features 0x1
  Config length 0x0
  Status 0x7
  Number of vrings 2
  Reserved (should be zero) [0][0]

  Vring 0
    Device Address 0x10040000
    Alignment 16
    Number of buffers 16
    Notify ID 0
    Physical Address 0x0

  Vring 1
    Device Address 0x10041000
    Alignment 16
    Number of buffers 16
    Notify ID 1
    Physical Address 0x0

root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0#
```

图 4.3.7.1 查看资源表信息

又比如 rproc_carveouts_ops 如下：

```
static const struct file_operations rproc_carveouts_ops = {
        .open           = rproc_carveouts_open,
```

```
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = single_release,
};
```

open 处理程序会执行 rproc_carveouts_open()函数，rproc_carveouts_open()函数如下：

```
static int rproc_carveouts_open(struct inode *inode, struct file *file)
{
        return single_open(file, rproc_carveouts_show, inode->i_private);
}
```

我们再打开 rproc_carveouts_show()函数如下：

```
/**
  * @brief      通过 debugfs 打印 carveout 内容。
  * @param      seq:序列文件接口指针
  * @retval     成功返回 0，否则返回适当的错误代码。
  */
static int rproc_carveouts_show(struct seq_file *seq, void *p)
{
  struct rproc *rproc = seq->private;
  struct rproc_mem_entry *carveout;

  list_for_each_entry(carveout, &rproc->carveouts, node) {
      seq_puts(seq, "Carveout memory entry:\n");
      seq_printf(seq, "\tName: %s\n", carveout->name);
      seq_printf(seq, "\tVirtual address: %pK\n", carveout->va);
      seq_printf(seq, "\tDMA address: %pad\n", &carveout->dma);
      seq_printf(seq, "\tDevice address: 0x%x\n", carveout->da);
      seq_printf(seq, "\tLength: 0x%x Bytes\n\n", carveout->len);
  }
  return 0;
}
```

可以看到，最终打印的是 carveout_memories 的信息，即 M4 内存相关配置信息，如名字 Name 、虚拟地址 Virtual address、DMA 地址 DMA address、设备地址 Device address 和地址长度 Length 等。假设此时已经加载了固件，在 Linux 文件系统的/sys/kernel/debug/remoteproc/remoteproc0 目录下执行如下指令：

```
cat carveout_memories
```

如下图 4.3.7.2 操作所示，这是笔者加载固件后查看的信息，这些信息就是 M4 的内存分配信息，分别有 retram、mcuram、mcuram2、vdev0vring0、vdev0vring1 和 vdev0buffer，这些地址范围和前面我们讲解 stm32mp157d-atk.dtsi 设备树下的存储配置一样：

```
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat carveout_memories
Carveout memory entry:
        Name: retram
        Virtual address: fbf95399
        DMA address: 0x38000000
        Device address: 0x0
        Length: 0x10000 Bytes

Carveout memory entry:
        Name: mcuram
        Virtual address: e7fd2427
        DMA address: 0x30000000
        Device address: 0x30000000
        Length: 0x40000 Bytes

Carveout memory entry:
        Name: mcuram2
        Virtual address: ace459da
        DMA address: 0x10000000
        Device address: 0x10000000
        Length: 0x40000 Bytes

Carveout memory entry:
        Name: vdev0vring0
        Virtual address: a4205634
        DMA address: 0x10040000
        Device address: 0x10040000
        Length: 0x1000 Bytes

Carveout memory entry:
        Name: vdev0vring1
        Virtual address: 6bad96af
        DMA address: 0x10041000
        Device address: 0x10041000
        Length: 0x1000 Bytes

Carveout memory entry:
        Name: vdev0buffer
        Virtual address: 9dd2dc9c
        DMA address: 0x00000000
        Device address: 0x10042000
        Length: 0x4000 Bytes
```

图 4.3.7.2 查看 M4 内存分配信息

大家在后面实验操作的时候可以多查看这些文件的信息。其它回调函数这里就不再一一分析了，感兴趣的小伙伴可以自行阅读代码。

4.3.8 rproc 设备树节点

打开 stm32mp151.dtsi 设备树文件，找到设备树中和 M4 相关的 Remoteproc 配置，如下：

```
1    mlahb {
2                compatible = "simple-bus";
3                #address-cells = <1>;
4                #size-cells = <1>;
5                dma-ranges = <0x00000000 0x38000000 0x10000>,
6                             <0x10000000 0x10000000 0x60000>,
7                             <0x30000000 0x30000000 0x60000>;
8
```

```
9                m4_rproc: m4@10000000 {
10                      compatible = "st,stm32mp1-m4";
11                      reg = <0x10000000 0x40000>,
12                            <0x30000000 0x40000>,
13                            <0x38000000 0x10000>;
14                      resets = <&scmi0_reset RST_SCMI0_MCU>;
15                      st,syscfg-holdboot = <&rcc 0x10C 0x1>;
16                      st,syscfg-tz = <&rcc 0x000 0x1>;
17                      st,syscfg-rsc-tbl = <&tamp 0x144 0xFFFFFFFF>;
18                      st,syscfg-copro-state = <&tamp 0x148 0xFFFFFFFF>;
19                      st,syscfg-pdds = <&pwr_mcu 0x0 0x1>;
20                      status = "disabled";
21
22                      m4_system_resources {
23                              compatible = "rproc-srm-core";
24                              status = "disabled";
25                       };
26               };
27       };
```

在上面的代码段里，有三个节点：mlahb 节点、m4_rproc 节点和 m4_system_resources 节点。m4_rproc 节点下就是加载和管理 M4 固件的配置信息。m4_system_resources 节点（也就是 M4 的资源管理器）下就是 M4 的资源分配配置信息。第 10 行，compatible 属性值为“st,stm32mp1-m4”，在 Linux 内核源码中搜索此属性值找到对应的驱动文件为 drivers/remoteproc/stm32_rproc.c，打开此文件找到如下内容：

```
1   static const struct of_device_id stm32_rproc_match[] = {
2           { .compatible = "st,stm32mp1-m4" },
3           {},
4   };
5   MODULE_DEVICE_TABLE(of, stm32_rproc_match);
6   /* 此处省略部分代码 */
7   static SIMPLE_DEV_PM_OPS(stm32_rproc_pm_ops,
8                            stm32_rproc_suspend, stm32_rproc_resume);
9
10  static struct platform_driver stm32_rproc_driver = {
11          .probe = stm32_rproc_probe,
12          .remove = stm32_rproc_remove,
13          .shutdown = stm32_rproc_shutdown,
14          .driver = {
15                  .name = "stm32-rproc",
16                  .pm = &stm32_rproc_pm_ops,
17                  .of_match_table = of_match_ptr(stm32_rproc_match),
18          },
19  };
```

```
20  /* 向 Linux 内核注册 stm32_rproc_driver 驱动 */
21  module_platform_driver(stm32_rproc_driver);
22  MODULE_DESCRIPTION("STM32 Remote Processor Control Driver");
23  MODULE_AUTHOR("Ludovic Barre <ludovic.barre@st.com>");
24  MODULE_AUTHOR("Fabien Dessenne <fabien.dessenne@st.com>");
25  MODULE_LICENSE("GPL v2");
```

上面的驱动代码是一个标准的 platform 驱动，第 2 行就是设备树下匹配到驱动程序的地方，在 stm32_rproc_match 中。第 15 行，成员 name 属性是 stm32-rproc，即定义了驱动的名字是 stm32-rproc，它是用于驱动与设备匹配的。

第 17 行，用于匹配对应的 device，即匹配设备树中的节点。第 21 行 module_platform_driver()函数向 Linux 内核注册 stm32_rproc_driver 这个 platform 驱动。当设备和驱动匹配成功以后，stm32_rproc_driver->probe 函数（即 stm32_rproc_probe()函数）就会被执行。下面，我们直接去看 stm32_rproc_probe()函数做了哪些操作。

stm32_rproc_probe()函数在 Linux 内核源码的 drivers/remoteproc/stm32_rproc.c 文件下，其函数定义如下所示：

```
1   static int stm32_rproc_probe(struct platform_device *pdev)
2   {
3     struct device *dev = &pdev->dev;
4     struct stm32_rproc *ddata;
5     struct device_node *np = dev->of_node;
6     struct rproc *rproc;
7     int ret;
8
9     ret = dma_coerce_mask_and_coherent(dev, DMA_BIT_MASK(32));
10    if (ret)
11          return ret;
12
13    rproc=rproc_alloc(dev,np->name,&st_rproc_ops,NULL,sizeof(*ddata));
14    if (!rproc)
15          return -ENOMEM;
16
17    rproc->has_iommu = false;
18    ddata = rproc->priv;
19    ddata->workqueue = create_workqueue(dev_name(dev));
20    if (!ddata->workqueue) {
21          dev_err(dev, "cannot create workqueue\n");
22          ret = -ENOMEM;
23          goto free_rproc;
24    }
25
26    platform_set_drvdata(pdev, rproc);
27
28    ret = stm32_rproc_parse_dt(pdev);
```

```
29    if (ret)
30          goto free_wkq;
31  if (!rproc->early_boot) {
32          ret = stm32_rproc_stop(rproc);
33          if (ret)
34                goto free_wkq;
35    }
36
37    ret = stm32_rproc_request_mbox(rproc);
38    if (ret)
39          goto free_wkq;
40
41    ret = rproc_add(rproc);
42    if (ret)
43          goto free_mb;
44
45    return 0;
46
47  free_mb:
48    stm32_rproc_free_mbox(rproc);
49  free_wkq:
50    destroy_workqueue(ddata->workqueue);
51  free_rproc:
52    if (device_may_wakeup(dev)) {
53          dev_pm_clear_wake_irq(dev);
54          device_init_wakeup(dev, false);
55    }
56    rproc_free(rproc);
57    return ret;
58  }
```

第 4 行，ddata 指针是 ST 官方的私有数据 stm32_rproc 结构体。

第 6 行，声明了一个 rproc 类型的结构体。

第 9 行，DMA 相关配置。

第 13 行，rproc_alloc 函数主要作用是分配一个新的 rproc 结构体空间，同时 st_rproc_ops 地址赋值给 rproc->ops 参数，目的是调用回调函数完成对应的功能（类似 XXX_ops 的结构体中的成员变量就是一些回调函数），这行的目的就是分配一个新的远程处理器（rproc）结构体，使用此函数创建 rproc 结构体后，应调用 rproc_add() 以完成远程处理器的注册，在后面我们会看到调用此函数。

第 18 行，保存 ST 官方的私有数据到 rproc 结构体里。

第 19 行，创建工作队列，workqueue 的名称是设备的名字，每个 workqueue 就是一个内核进程，为系统创建一个内核线程。

第 28 行，调用 stm32_rproc_parse_dt()函数来获取设备树中的属性，目的就是完成 M4 设备的配置。

第 31~35 行，如果没有加载固件则调用 stm32_rproc_stop 函数，该函数会做一些操作，如请求关闭远程处理器，此时传输阻塞，会打印“warning: remote FW shutdown without ack”并将远程处理器状态设置为离线状态等。

第 37 行，stm32_rproc_request_mbox()函数为远程处理器申请邮箱，A7 和 M4 可通过邮箱发布数据。

第 41 行，rproc_add()函数在前面的介绍中我们已经了解了，就是注册一个远程处理器。

第 56 行，rproc_free()函数释放由 rproc_alloc ()分配的 rproc 。

下面我们来看看 st_rproc_ops 结构体，在 stm32_rproc.c 找到如下代码：

```
static struct rproc_ops st_rproc_ops = {
        .start          = stm32_rproc_start,
        .stop           = stm32_rproc_stop,
        .kick           = stm32_rproc_kick,
        .load           = stm32_rproc_elf_load_segments,
        .parse_fw        = stm32_rproc_parse_fw,
        .find_loaded_rsc_table = stm32_rproc_elf_find_loaded_rsc_table,
        .sanity_check    = stm32_rproc_elf_sanity_check,
        .get_boot_addr   = stm32_rproc_elf_get_boot_addr,
};
```

st_rproc_ops 结构体中有几个处理程序，每个处理程序对应一个回调函数，如 start 处理程序接受一个 rproc 结构体，然后打开设备电源并启动它。stop 处理程序采用 rproc 并关闭远程处理器。kick 处理程序接受一个 rproc 和一个放置新消息的虚拟队列的索引，调用此函数时会中断远程处理器并让它知道它有待处理的消息。find_loaded_rsc_table 就是查找已经加载的固件资源表，执行的是 stm32_rproc_elf_find_loaded_rsc_table()函数。

每个 Remoteproc 的实现至少应该提供 start 和 stop 处理程序，关于这些函数我们不必深入分析，只要我们知道这是 ST 官方实现操作 M4 相关的接口函数就行了。

4.4 链接脚本

关于链接脚本，在《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南》的第六章的 6.3 小节有讲解，如想深入了解链接脚本，可参考这部分内容。我们在第一章创建的 RPMsg_TEST 工程，在 M4 工程下可以看到链接脚本 STM32MP157DAAX_RAM.ld，该文件就是从 STM32MP157 的固件包 STM32Cube_FW_MP1_V1.2.0\Drivers\CMSIS\Device\ST\STM32MP1xx\Source\Templates\gcc\linker\stm32mp15xx_m4.ld 拷贝得来的。

4.4.1 链接脚本地址分配

下面我们直接分析 STM32MP157DAAX_RAM.ld，了解代码段、数据段以及共享内存分别链接到了内存的那个区域，如下是链接脚本的部分代码所示：

```
1   /* 程序入口，程序将从 Reset_Handler 开始执行 */
2   ENTRY(Reset_Handler)
3
4   /* 用户模式栈的最高地址，声明内存末尾地址 */
5   _estack = 0x10040000;          /* 堆栈末尾 = RAM 起始地址 + RAM 空间大小 */
6   /* 定义了堆和栈的最小空间大小 */
7   _Min_Heap_Size = 0x200 ;       /* 堆大小 */
```

```
8   _Min_Stack_Size = 0x400 ;      /* 栈大小 */
9
10  /* 该部分是内存定义，'MEMORY' 命令描述目标平台上内存块的位置与长度。
11   * m_interrupts : M4 的中断向量表， 地址 0x00000000~0x00000298
12   * m_text       : M4 的代码段，     地址 0x10000000~0x10020000
13   * m_data       : M4 的数据段，     地址 0x10020000~0x10040000
14   * m_ipc_shm    : 共享内存和 vring，地址 0x10040000~0x10048000
15   */
16  MEMORY
17  {/* 名称        权限（读 R/写 W/执行 X）    起始地址            地址长度 */
18    m_interrupts (RX)    :  ORIGIN = 0x00000000, LENGTH = 0x00000298
19    m_text       (RX)    : ORIGIN = 0x10000000, LENGTH = 0x00020000
20    m_data       (RW)    : ORIGIN = 0x10020000, LENGTH = 0x00020000
21    m_ipc_shm    (RW)    : ORIGIN = 0x10040000, LENGTH = 0x00008000
22  }
23
24  /* OpenAMP 启用 RPMsg 所需的符号 */
25  __OPENAMP_region_start__   = ORIGIN(m_ipc_shm);
26  __OPENAMP_region_end__ = ORIGIN(m_ipc_shm)+LENGTH(m_ipc_shm);
27
28  /* SECTIONS 关键字，用来描述输出文件各个 section 的布局 */
29  SECTIONS
30  {
31      /* 此处省略该部分代码 */
32      .resource_table :     /* 资源表 */
33      {
34      . = ALIGN(4);
35      KEEP (*(.resource_table*))
36      . = ALIGN(4);
37      } > m_data
38      /* 此处省略该部分代码 */
39  }
```

第 2 行，程序入口，程序将从 Reset Handler 函数开始执行，该函数在启动文件 startup_stm32mp15xx.s 中有定义。

第 5 行，设置堆栈的最高地址为 0x10040000。

第 7、8 行，定义了堆和栈的最小空间大小，其中，设置堆大小为 512B，栈大小为 1KB。

第 16~22 行，以 MEMORY 命令定义了系统中可用于放置代码和数据的内存区域：

① 区域名为 m_interrupts 的地址范围是 0x00000000~0x00000298，这个范围也就是 M4 内核中断向量表的范围，对应前面内存映射关系图中的 RETRAM 区域；

② m_text 区域的地址范围是 0x10000000~0x10020000，刚好对应内存映射关系图中的 SRAM1 区域，链接的是代码段（Code）；

③ m_data 区域的范围是 0x10020000~0x10040000，对应内存映射关系图中的 SRAM2 区域，链接的是数据段（Data）；

④ m_ipc_shm 区域的范围是 0x10040000~0x10048000，此范围落在了 SRAM3 中，ipc（Inter Process Communicaton）即进程通信，这个区域可以作为 IPC 缓冲区（IPC Buffers），即共享的内存就在这个区域中。

第 29~39 行省略了大部分代码，该部分代码指定了程序的各个内容该如何放置在 SRAM 上，其中第 32~37 行是资源表，资源表也在 m_data 区域中，即占用 SRAM2。

以上配置如下表 4.4.1.1 所示：

段	地址	大小	内存区域
中断向量表	0x00000000~0x00000298	664B	M4 可见的 RETRAM
代码段	0x10000000~0x10020000	128KB	M4 可见的 SRAM1
数据段（包含资源表）	0x10020000~0x10040000	128KB	M4 可见的 SRAM2
共享内存和 vring	0x10040000~0x10048000	32KB	SRAM3

表 4.4.1.1 链接脚本的地址分配

以上的链接地址就是程序的执行地址，找到链接地址就知道程序是在哪里执行了。根据 MEMORY 命令定义的地址范围，我们知道 m_text、m_data 和 m_ipc_shm 占用了 SRAM1~SRAM3，可以看到，M4 内核占用的是 RETRAM（M4 可见）和 SRAM1~SRAM2，SRAM3 用于 A7 和 M4 内核交换数据，还有个 SRAM4 怎么分配呢？

在第二章的内存映射图中可以看到，SRAM1~SRAM4 的地址范围是 0x10000000~0x1005FFFF，共 384KB，如果 STM32MP157 主控芯片只是运行 M4 内核（跑裸机或者 RTOS），不运行 A7 内核的话，这 SRAM1~SRAM4 可以全部分配给 M4 使用，但是，如果要同时运行 M4 内核和 A7 内核的话，这些地址分配就要注意了：

如果要运行 A7 的话，M4 并不是完全占用 SRAM3，具体占用多少需要根据 Linux 下的设备树配置来决定，在 A7 和 M4 双核通信中，默认 A7 和 M4 共同占用 SRAM3 的 0x10040000~0x10046000，这部分地址作为 A7 和 M4 通信的内存交换区（共享内存），这点在 4.2.1 小节我们已经介绍过。内核下的设备树如下图 4.4.1.1 所示。

图 4.4.1.1 设备树部分截图

剩下的 SRAM4 用于做什么呢？如果不跑 A7，只是跑 M4 的话，M4 内核完全可以使用这部分区域，由用户来指定。如果 A7 跑 Linux 操作系统，在 Linux 设备树下已经默认将 SRAM4 当做了 Linux 功能的 DMA 了，如果要释放这部分区域，在设备树下将对应节点删除释放资源即可。

根据上述描述，这几个区域对应关系如下图 4.4.1.2，从图中可以明显看出 SRAM 的地址分配情况：

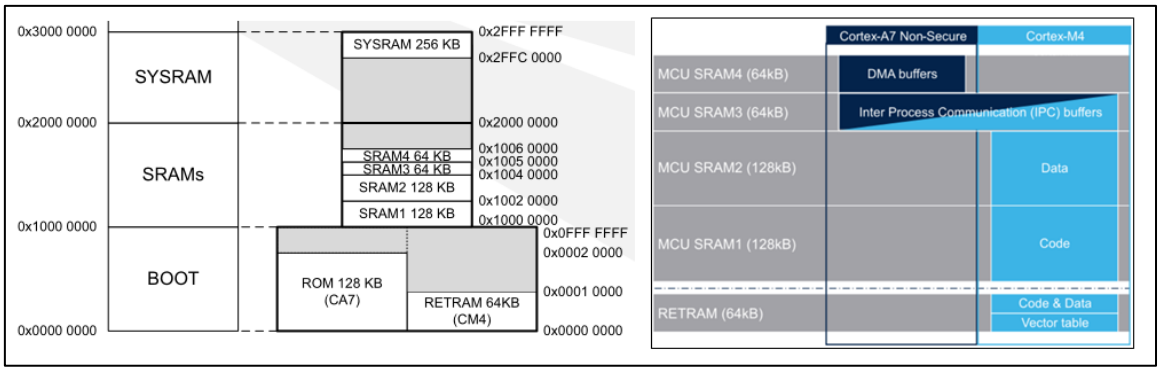


图 4.4.1.2 ST 的参考设计

综合以上分析，在 ST 的参考设计中，总结如下：

如果不跑 A7，只运行 M4（M4 可以跑裸机、RTOS）：SRAM1~SRAM4 可以完全分配给 M4；

如果同时跑 A7 和 M4（例如进行双核通信）：SRAM1 和 SRAM2 是单独给 M4 用的，SRAM3 的部分地址是 M4 和 A7 一起使用的，SRAM4 在 Linux 下单独配置了 DMA，即被 A7 占用了。此时，如果要修改 MEMORY 中的地址区域范围，一定要联系内存映射表的地址范围来修改。

4.4.2 重新划分存储区域

本小节主要讲解如何重新规划 MCURAM 的存储区域，用户可以根据自己的实际情况来划分存储区域，假设用户的代码段已经超出了原来链接脚本配置的范围了，可以通过手动调整链接脚本和设备树来解决。

本小节的内容只是为了讲解如何重新划分存储区域，大家可以不必按照本节的讲解来修改原来工程中链接脚本的配置以及内核源码下设备树的配置，因为默认的配置已经满足我们后续双核通信例程测试了，当自己开发的时候有需要时再根据个人的实际情况来进行修改。

1. 修改链接脚本

前面我们说过，SRAM4 在 Linux 下默认当做了 DMA 的区域了，如果 Linux 下不使用 DMA，那么可以将 SRAM4 当做其它用途，例如，我们可以将 SRAM4 分配给 M4 使用，对于 SRAM3 中未使用的区域，我们也可以将其用作其它功能，假设刚好我们有这个需求，那么，我们可以将以上链接脚本修改如下：

```
/* Memories definition */
/*
MEMORY
{
  m_interrupts (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000298
  m_text       (RX)  : ORIGIN = 0x10000000, LENGTH = 0x00020000
  m_data       (RW)  : ORIGIN = 0x10020000, LENGTH = 0x00020000
```

```
  m_ipc_shm    (RW)  : ORIGIN = 0x10040000, LENGTH = 0x00008000
}
*/
/* 以上部分修改如下，其它代码保持不变 */
MEMORY
{
  m_interrupts (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000298
  m_text       (RX)  : ORIGIN = 0x10000000, LENGTH = 0x0003A800
  m_data       (RW)  : ORIGIN = 0x1003A800, LENGTH = 0x00015800
  m_ipc_shm    (RW)  : ORIGIN = 0x10050000, LENGTH = 0x00010000
}
```

以上的配置如表 4.4.2.1 中所示：

段	地址	大小	区域
中断向量表	0x00000000~0x00000298	664B	M4 可见的 RETRAM
代码段	0x10000000~0x1003A800	234KB	M4 可见的 SRAM1+SRAM2
数据段 （包含资源表）	0x1003A800~0x10050000	86KB	M4 可见的 SRAM2+SRAM3
共享内存和 vring	0x10050000~0x10060000	64KB	SRAM4

表 4.4.2.1 新的链接脚本地址划分

以上的地址范围是笔者随机分配的，我们看到，在以上的地址配置中，共享的内存占用了 SRAM4，而在 ST 官方默认的配置中，共享的内存在 SRAM3 中。可以发现，这些地址分配并不是固定的，只要在合理的范围内就行，在实际项目开发中，可根据个人情况进行配置。

2. 修改设备树

在链接脚本中对存储区域重新划分后，还需要同步修改设备树，设备树的地址也要和链接脚本的对应，否则 A7 启动 M4 固件后无法建立通信，会报地址请求错误或者其它错误，如下图 4.4.2.1 所示，报错提示”unable to acquire memory-region”，即提示无法获取内存区域，这种错误就是因为链接脚本和设备树的地址不一致导致的：

```
root@ATK-MP157:/boot# cd /lib/firmware/
root@ATK-MP157:/lib/firmware# ./s0.sh
[   39.782397] remoteproc remoteproc0: powering up m4
[   40.045978] remoteproc remoteproc0: Booting fw image oneos.axf, size 5545712
[   40.052370] stm32-rproc mlahb:m4@10000000: unable to acquire memory-region
[   40.068881] remoteproc remoteproc0: Boot failed: -22
sh: write error: Invalid argument
root@ATK-MP157:/lib/firmware# start
```

图 4.4.2.1 典型报错

stm32mp157d-atk.dtsi 设备树需要修改如下地方，首先，将分配给 A7 当做 DMA 的相关节点注释掉以释放资源，如下图 4.4.2.2 所示：

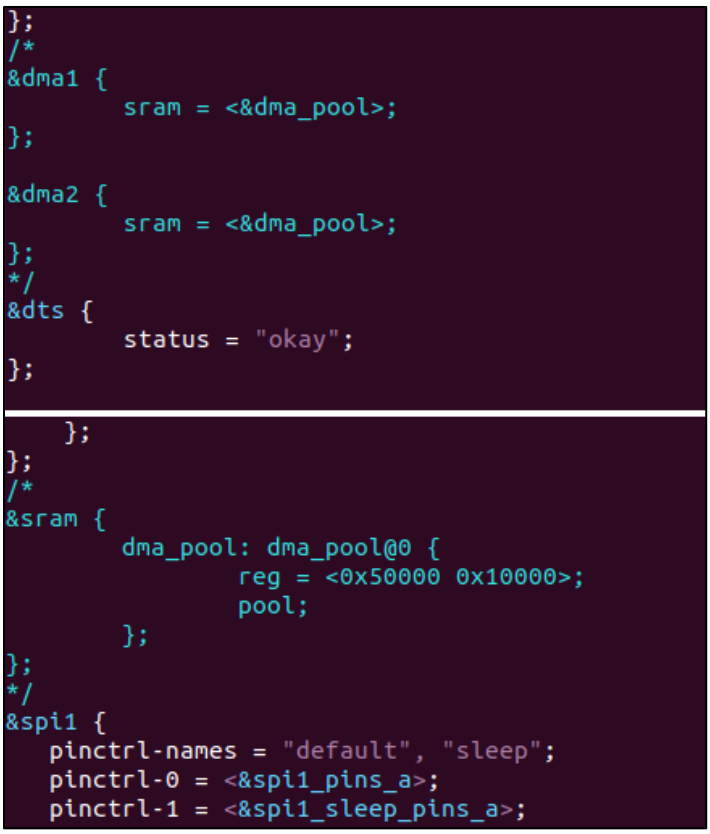


图 4.4.2.2 屏蔽 A7 占用的 SRAM4

屏蔽的代码如下所示：

```
/*
&dma1 {
        sram = <&dma_pool>;
};

&dma2 {
        sram = <&dma_pool>;
};
*/

/*
&sram {
        dma_pool: dma_pool@0 {
                reg = <0x50000 0x10000>;
                pool;
        };
};
*/
```

然后在 stm32mp157d-atk.dtsi 下修改 MCURAM 的地址范围，如下：

```
1   reserved-memory {
2               #address-cells = <1>;
```

```
3               #size-cells = <1>;
4               ranges;
5       mcuram2: mcuram2@10000000 {
6                       compatible = "shared-dma-pool";
7                       reg = <0x10000000 0x50000>;
8
9                       no-map;
10                      };
11
12      vdev0vring0: vdev0vring0@10050000 {
13                      compatible = "shared-dma-pool";
14                      reg = <0x10050000 0x1000>;
15
16                      no-map;
17                      };
18
19      vdev0vring1: vdev0vring1@10051000 {
20                      compatible = "shared-dma-pool";
21                      reg = <0x10051000 0x1000>;
22
23                      no-map;
24                      };
25
26      vdev0buffer: vdev0buffer@10052000 {
27                      compatible = "shared-dma-pool";
28                      reg = <0x10052000 0x4000>;
29
30                      no-map;
31                      };
32
33      mcuram: mcuram@30000000 {
34                      compatible = "shared-dma-pool";
35                      reg = <0x30000000 0x50000>;
36
37                      no-map;
38                      };
39
40      retram: retram@38000000 {
41                      compatible = "shared-dma-pool";
42                      reg = <0x38000000 0x10000>;
43
44                      no-map;
45                      };
```

```
46          };
```

以上地址配置如下表 4.4.2.2 所示：

子节点	地址	大小	区域
mcuram2	0x10000000~0x10050000	320KB	M4 可见的 SRAM1+SRAM2+SRAM3
vdev0vring0	0x10050000~0x10051000	4KB	SRAM4
vdev0vring1	0x10051000~0x10052000	4KB	
vdev0buffer	0x10052000~0x10056000	16KB	
mcuram	0x30000000~0x30050000	320KB	A7 可见的 SRAM1+SRAM2+SRAM3
retram	0x38000000~0x38010000	64KB	A7 可见的 RETRAM

表 4.4.2.2 SRAM 地址划分

第 33~39 行，前面我们说过，RAM aliases 和 SRAMs 的物理地址是一样的，所以在配置设备树的时候它们的地址范围最好是一致的，这里 mcuram 子节点和 mcuram2 子节点的地址范围设置一致。

在前面链接脚本的配置中，代码段、数据段和资源表加起来刚好是 320KB，也就是等于此处 mcuram2 区域的地址范围。此处两个 vring 和共享内存（vdev0buffer）占用了 24KB，在前面的链接脚本中，我们配置了 32KB，但实际上只能用这 24KB，两个 vring 和 vdev0buffer 怎么使用呢，我们会在后面介绍。

注意，设备节点@后面的地址一定不要忘了同步修改。修改好后，重新编译设备树并生成 stm32mp157d-atk.dtb 文件，然后将开发板 Linux 文件系统/boot 目录下的 stm32mp157d-atk.dtb 文件替换掉。接下来再编译 M4 工程，生成.elf 文件，此.elf 文件就是 M4 的固件，然后我们可以按照下文的测试方法来进行测试。

4.5 Remoteproc 的使用

本节实验主要是介绍如何使用 Remoteproc 来加载和启动 M4 固件，实验中会使用到前面 M4 工程编译出来的 RPMsg_TEST_CM4.elf 文件。

本工程可参考参考开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch1\RPMsg_TEST。

4.5.1 硬件连接

如下图 4.5.1.1 所示，Type-C 线接在开发板的 USB_TTL 接口，用于打印 A7 信息，网线的另一端接的路由器，后面可以通过网络的方式将编译好的.elf 文件传输到开发板。屏幕可用于显示 Linux 的 APP 界面，如果用不到屏幕的话，屏幕可以选择接或者不接。开发板要通过电源线来供电，要注意的是，开发板的拨码开关一定要设置成 EMMC 启动方式，也就是启动开发板 EMMC 中已经烧录好的出厂 Linux 操作系统。

硬件连接如下图 4.5.1.1 所示：

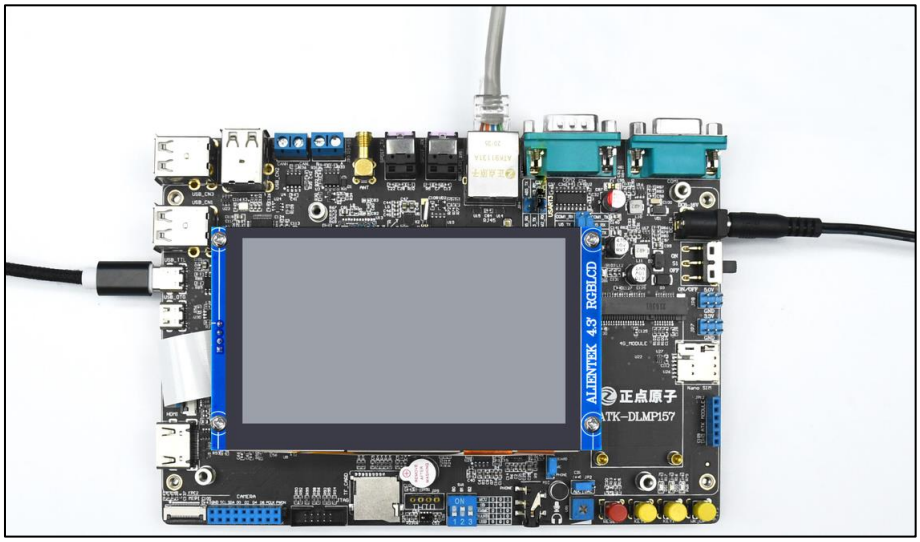


图 4.5.1.1 硬件连接

4.5.2 传输固件

本实验需要将编译出来的.elf 文件拷贝到开发板 Linux 文件系统的/lib/firmware 目录下，拷贝的方式有多种，例如可以使用 U 盘、TF 卡等存储设备拷贝到开发板上，或者可以使用网络的方式将文件传输到开发板上，下面使用网络传输方式。

1. 检查开发板和 PC 是否可以通过网络通信

启动开发板，进入 Linux 操作系统后，执行如下 ifconfig 指令查看开发板的 IP 地址，如下图 4.5.2.1 所示，开发板的 IP 地址是 192.168.1.193。

```
ifconfig
```

```
root@ATK-MP157:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 7E:E4:72:21:5E:8E
          inet addr:192.168.1.193  Bcast:192.168.3.255  Mask:255.255.252.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5106 errors:0 dropped:58 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:472852 (461.7 KiB)  TX bytes:1859 (1.8 KiB)
          Interrupt:53 Base address:0x8000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:140 (140.0 B)  TX bytes:140 (140.0 B)

usb0      Link encap:Ethernet  HWaddr FA:32:D6:7B:59:0B
          inet addr:192.168.7.1  Bcast:192.168.7.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:215 errors:0 dropped:86 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20127 (19.6 KiB)  TX bytes:2027 (1.9 KiB)
```

图 4.5.2.1 查询开发板的 IP 地址

在 windows 的 cmd 下执行 ipconfig 指令查看 PC 的 IP 地址，如下图 4.5.2.2 所示，IP 地址为 192.168.1.156。

```
ipconfig
```

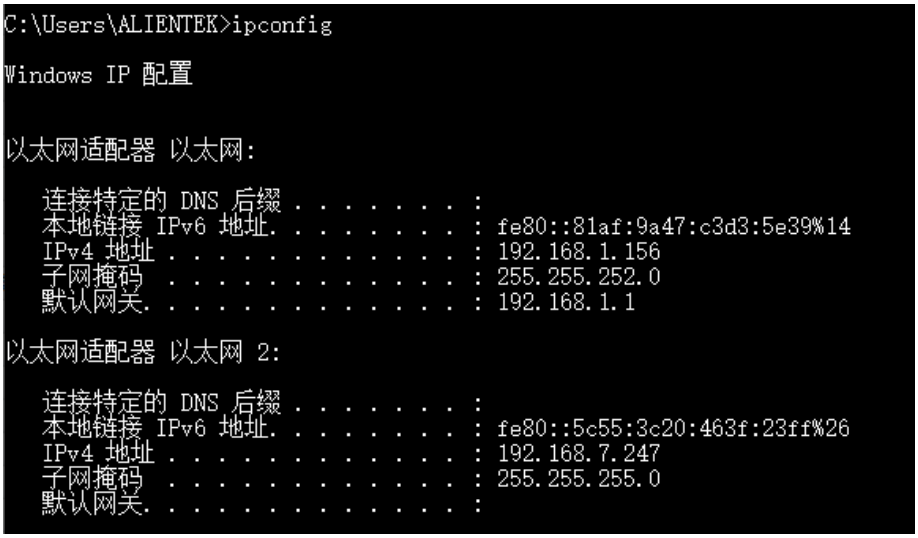


图 4.5.2.2 查看 PC 的 IP 地址

开发板的 IP 地址是 192.168.1.193，PC 的 IP 地址是 192.168.1.156，两者 IP 地址在同一个网段内，我们在开发板中执行如下指令 ping 测试 PC 看看是否可以 ping 通，如下图 4.5.2.3 所示，ping 了几个包后，按下 Ctrl+C 组合键终止 ping 操作，从结果中看到 ping 了 6 个包，没有丢包情况，可以 ping 通，说明开发板和 PC 可以通过网络来通信，所以我们可以通过网络的方式将第一章编译出的 RPMsg_TEST_CM4.elf 文件传输到开发板的文件系统中。

```
ping 192.168.1.156
```

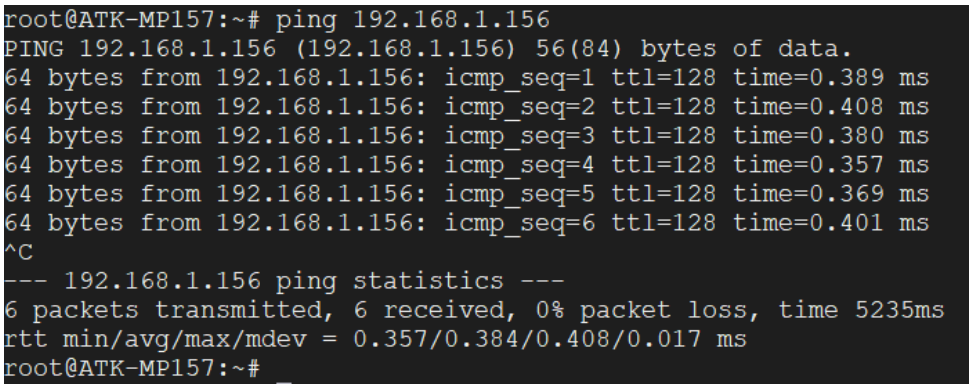


图 4.5.2.3 开发板和 PC ping 测试

2.进入 M4 工程编译目录

进入第一章工程 E:\STM32CubeIDE_WorkSpace1.4\RPMsg_TEST\CM4\Debug 目录下，可以看到编译生成的 RPMsg_TEST_CM4.elf 文件，在该目录下按下 Shift 键，同时点击鼠标右键，打开一个选项界面，如下图 4.5.2.4 所示，可以看到 Powershell 选项，或者如果电脑安装了 Git 的话，可以看到有 Git Bash Here，这两个也就是 Shell 终端，打开 Shell 终端后可以操作 Shell 指令，两者选其中一个就可以，下面我们就选择使用 Powershell 吧。

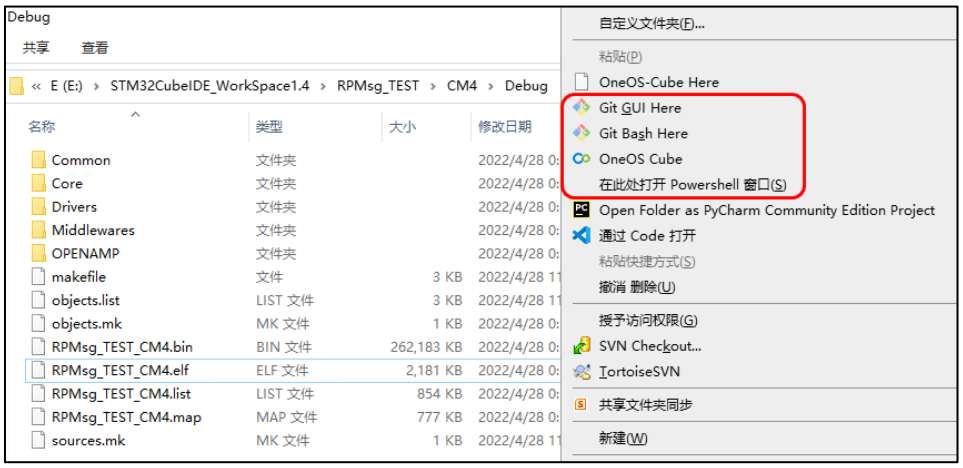


图 4.5.2.4 打开 Shell 终端

打开 Powershell 界面后如下，输入如下命令后按下回车键，可以将固件 RPMsg_TEST_CM4.elf 传输到开发板的/lib/firmware 目录下，如下图 4.5.2.5 所示。

```
scp RPMsg_TEST_CM4.elf root@192.168.1.193:/lib/firmware
```

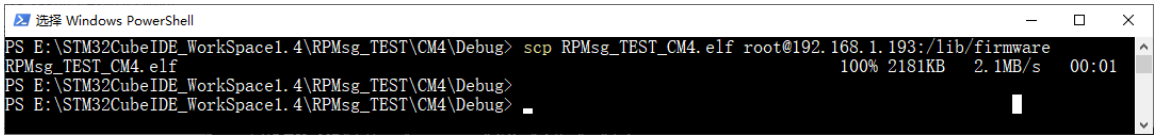


图 4.5.2.5 拷贝固件到开发板中

如下图 4.5.2.6 所示，在开发板的/lib/firmware 下可以看到已经有传输的固件 RPMsg_TEST_CM4.elf 了。

```
root@ATK-MP157:/lib/firmware# sync
root@ATK-MP157:/lib/firmware# ls
brcm                    OpenAMP_TTY_echo_CM4.elf  regulatory.db.p7s  RPMsg_TEST_CM4.elf  rtlbt
LICENCE.cypress_bcm4343  regulatory.db            RPMsg_CM4.elf      RPMsg_UART_CM4.elf
root@ATK-MP157:/lib/firmware#
```

图 4.5.2.6 固件拷贝成功

4.5.3 加载和运行固件

在开发板下执行如下指令，可以加载和启动固件 RPMsg_TEST_CM4.elf，如下图 4.5.3.1 所示：

```
/* 加载固件 */
echo RPMsg_TEST_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware
/* 启动固件 */
echo start >/sys/class/remoteproc/remoteproc0/state
```

```
root@ATK-MP157:/lib/firmware# ls
brcm                    OpenAMP_TTY_echo_CM4.elf  regulatory.db.p7s  RPMsg_TEST_CM4.elf  rtlbt
LICENCE.cypress_bcm4343  regulatory.db            RPMsg_CM4.elf      RPMsg_UART_CM4.elf
root@ATK-MP157:/lib/firmware# echo RPMsg_TEST_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware
root@ATK-MP157:/lib/firmware# echo start >/sys/class/remoteproc/remoteproc0/state
[  676.451901] remoteproc remoteproc0: powering up m4
[  676.568464] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2232980
[  676.575267]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[  676.588469] virtio_rpmsg_bus virtio0: rpmsg host is online
[  676.593505]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[  676.601649] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:/lib/firmware#
```

图 4.5.3.1 加载固件

对以上打印的 log 说明如下：

- powering up m4
  提示已经启动了 M4；
- Booting fw image RPMsg_TEST_CM4.elf, size 2232980
  提示固件包 RPMsg_TEST_CM4.elf 大小约 2232980 个字节；
- mlahb:m4@10000000#vdev0buffer:assigned reserved memory node vdev0buffer@10042000
  分配的共享内存 vdev0buffer 起始地址为 0x10042000，该地址位于 SRAM3 中；
- virtio_rpmsg_bus virtio0: rpmsg host is online
  提示 RPMsg 主处理器在线，但是此时 A7 和 M4 还不能进行核间通信，为什么呢？我们会在下一章进行介绍。
- mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
  成功注册 Virtio 设备；
- remoteproc remoteproc0: remote processor m4 is now up
  提示现在远程处理器 M4 已经启动。

固件加载成功后，M4 已经成功启动了，观察到开发板底板的 DS1 灯在闪烁（黄色的灯），说明主处理器 A7 通过 Remoteproc 加载和启动协处理器 M4 成功。

进入/sys/kernel/debug/remoteproc/remoteproc0 目录下查看相关文件的内容，例如查看资源表 resource_table，如下图 4.5.3.2 所示：

```
cd /sys/kernel/debug/remoteproc/remoteproc0
cat resource_table
```

```
root@ATK-MP157:/lib/firmware# cd /sys/kernel/debug/remoteproc/remoteproc0
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# ls
carveout_memories  crash  name  recovery  resource_table
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat resource_table
Entry 0 is of type vdev
  ID 7
  Notify ID 0
  Device features 0x1
  Guest features 0x1
  Config length 0x0
  Status 0x7
  Number of vrings 2
  Reserved (should be zero) [0][0]

  Vring 0
    Device Address 0x10040000
    Alignment 16
    Number of buffers 16
    Notify ID 0
    Physical Address 0x0

  Vring 1
    Device Address 0x10041000
    Alignment 16
    Number of buffers 16
    Notify ID 1
    Physical Address 0x0

root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0#
```

图 4.5.3.2 查看资源表

可以看到 Virtio RPMsg 设备 ID 为 7，vring 个数是 2，每个 vring 有 16 个 rpmsg buffer，vring 0 的设备地址为 0x10040000，vring 1 的设备地址为 0x10041000。

其它文件的内容和我们前面讲解的一样，如 name 下的是 m4，也就是远程处理器设备是 M4，如下图 4.5.3.3 所示：

```
cat name
cat recovery
cat crash
```

```
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat name
m4
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat recovery
enabled
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat crash
cat: crash: Invalid argument
```

图 4.5.3.3 查看其它文件

如下图 4.5.3.4 所示，查看 carveout_memories 可以知道 M4 的内存分配情况，retram、mcuram、mcuram2、vdev0vring0、vdev0vring1 和 vdev0buffer 的设备地址以及地址长度，和前面第 4.2.1 小节分析 stm32mp157d-atk.dtsi 设备树的时候看到的一致。

```
cat carveout_memories
```

```
root@ATK-MP157:/sys/kernel/debug/remoteproc/remoteproc0# cat carveout_memories
Carveout memory entry:
        Name: retram
        Virtual address: 7a8bdcc3
        DMA address: 0x38000000
        Device address: 0x0
        Length: 0x10000 Bytes

Carveout memory entry:
        Name: mcuram
        Virtual address: 3082364d
        DMA address: 0x30000000
        Device address: 0x30000000
        Length: 0x40000 Bytes

Carveout memory entry:
        Name: mcuram2
        Virtual address: 16188ccd
        DMA address: 0x10000000
        Device address: 0x10000000
        Length: 0x40000 Bytes

Carveout memory entry:
        Name: vdev0vring0
        Virtual address: 4adc61d3
        DMA address: 0x10040000
        Device address: 0x10040000
        Length: 0x1000 Bytes

Carveout memory entry:
        Name: vdev0vring1
        Virtual address: 960302b1
        DMA address: 0x10041000
        Device address: 0x10041000
        Length: 0x1000 Bytes

Carveout memory entry:
        Name: vdev0buffer
        Virtual address: ead2df54
        DMA address: 0x00000000
        Device address: 0x10042000
        Length: 0x4000 Bytes
```

图 4.5.3.4 查看 carveout_memories

4.5.4 关闭固件

如果要关闭固件的话，执行如下指令即可，如下图 4.5.4.1 所示：

```
/* 关闭，停止固件运行 */
echo stop > /sys/class/remoteproc/remoteproc0/state
```

```
root@ATK-MP157:~# echo stop > /sys/class/remoteproc/remoteproc0/state
[  238.957792] remoteproc remoteproc0: warning: remote FW shutdown without ack
[  238.963385] remoteproc remoteproc0: stopped remote processor m4
root@ATK-MP157:~#
```

图 4.5.4.1 关闭固件

4.5.4 编写脚本

以上的操作都是通过手动输入指令来完成的，如果我们需要经过很多次测试，每次都要手动输入指令的话，这样效率会低很多，所以我们可以将以上指令写入到一个脚本里，通过执行脚本可以大大提高效率。如下图 4.5.4.1，在/lib/firmware 目录下执行如下指令可以新建一个 test1.sh 脚本：

```
vi test1.sh                /* 新建 test1.sh 脚本 */
```

在脚本中添加内容如下：

```
#!/bin/sh

echo RPMsg_TEST_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware

echo start >/sys/class/remoteproc/remoteproc0/state
```

添加完脚本后，保存修改退出，再执行如下指令以修改 test1.sh 为可执行权限：

```
chmod a+x test1.sh
```

```
root@ATK-MP157:/lib/firmware# ls
brcm                      regulatory.db      RPMsg_TEST_CM4.elf
LICENCE.cypress_bcm4343   regulatory.db.p7s  RPMsg_UART_CM4.elf
OpenAMP_TTY_echo_CM4.elf  RPMsg_CM4.elf      rtlbt
root@ATK-MP157:/lib/firmware# vi test1.sh
root@ATK-MP157:/lib/firmware# sync
root@ATK-MP157:/lib/firmware# ls
brcm                      regulatory.db      RPMsg_TEST_CM4.elf  test1.sh
LICENCE.cypress_bcm4343   regulatory.db.p7s  RPMsg_UART_CM4.elf
OpenAMP_TTY_echo_CM4.elf  RPMsg_CM4.elf      rtlbt
root@ATK-MP157:/lib/firmware# chmod a+x test1.sh
root@ATK-MP157:/lib/firmware# sync
root@ATK-MP157:/lib/firmware# ls -l test1.sh
-rwxr-xr-x 1 root root 124 Feb  8 00:03 test1.sh
root@ATK-MP157:/lib/firmware#
```

图 4.5.4.1 新建 test1.sh 脚本

以后，只要将 RPMsg_TEST_CM4.elf 传输到/lib/firmware 目录下后，都可以通过执行./test1.sh 指令来加载和启动固件，如下图 4.5.4.2 所示，运行后可以看到开发板底板的 DS1 灯在闪烁：

```
./test1.sh
```

```
root@ATK-MP157:/lib/firmware# ./test1.sh
[ 1000.430364] remoteproc remoteproc0: powering up m4
[ 1000.442973] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2232980
[ 1000.450026]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[ 1000.458671] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 1000.463687]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[ 1000.470422] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:/lib/firmware#
```

图 4.5.4.2 通过脚本加载和启动固件

我们也可以进一步修改以上脚本，如在/lib/firmware 目录下新建一个 test2.sh 文件，文件内容如下：

```
#!/bin/sh

rproc_class_dir="/sys/class/remoteproc/remoteproc0"
fmw_dir="/lib/firmware"

cd /sys/class/remoteproc/remoteproc0

if [ $1 == "start" ]
then
    /bin/echo -n $2 > $rproc_class_dir/firmware
    /bin/echo -n start > $rproc_class_dir/state
fi

if [ $1 == "stop" ]
then
    /bin/echo -n stop > $rproc_class_dir/state
fi
```

保存修改好的 test2.sh，然后同样设置 test2.sh 为可执行权限，修改好以后，可以通过执行如下指令来加载和启动 RPMsg_TEST_CM4.elf 固件，或者关闭 RPMsg_TEST_CM4.elf 固件：

```
/* 加载、启动 M4 固件 */
./test2.sh start RPMsg_TEST_CM4.elf
/* 停止 M4 固件 */
./test2.sh stop RPMsg_TEST_CM4.elf
```

操作过程如下所示：

```
root@ATK-MP157:/lib/firmware# ls
brcm                    regulatory.db       RPMsg_TEST_CM4.elf  test1.sh
LICENCE.cypress_bcm4343 regulatory.db.p7s   RPMsg_UART_CM4.elf  test2.sh
OpenAMP_TTY_echo_CM4.elf RPMsg_CM4.elf      rtlbt
root@ATK-MP157:/lib/firmware# ./test2.sh start RPMsg_TEST_CM4.elf
[ 7442.991261] remoteproc remoteproc0: powering up m4
[ 7443.001113] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2232980
[ 7443.008010]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[ 7443.016672] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 7443.022984]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[ 7443.028399] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:/lib/firmware#
root@ATK-MP157:/lib/firmware# ./test2.sh stop RPMsg_TEST_CM4.elf
[ 7454.507725] remoteproc remoteproc0: warning: remote FW shutdown without ack
[ 7454.513307] remoteproc remoteproc0: stopped remote processor m4
root@ATK-MP157:/lib/firmware#
```

图 4.5.4.3 加载和启动固件

我们也可以新建一个 q.sh 文件，在该文件中添加如下内容：

```
#!/bin/sh

echo stop >/sys/class/remoteproc/remoteproc0/state
```

同样的，设置 q.sh 文件为可执行权限，下次要关闭 M4，可以直接执行./q.sh 命令即可，如下图 4.5.4.4 所示：

```
root@ATK-MP157:/lib/firmware# ls -l q.sh
-rwxr-xr-x 1 root root 52 Feb  7 23:52 q.sh
root@ATK-MP157:/lib/firmware# cat q.sh
echo stop >/sys/class/remoteproc/remoteproc0/state

root@ATK-MP157:/lib/firmware#
```

图 4.5.4.4 q.sh 文件

4.5.5 通过 STM32CubeIDE 进行调试

前面我们是将编译生成的.elf 文件手动传输到了开发板中进行测试，如果使用 STM32CubeIDE 来调试，不需要手动拷贝，可由 STM32CubeIDE 自动完成，硬件连接以及具体操作步骤可参考《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南》第 28.2 小节。

注意，通过 STM32CubeIDE 进行调试时，笔者建议使用 1.4 版本的 STM32CubeIDE，高版本的可能会有问题，这也许是高版本软件的问题，在《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南》第 28.1.2 小节有提到。

4.5.6 设置开机自动运行固件

假设希望开机以后自动加载 M4 固件，不需要我们手动运行以上脚本，可以设置开机自动加载和运行固件。

进入开发板文件系统的/lib/systemd/system/目录下，可以看到在该目录下存放了很多文件，如以 XXX.service 形式的文件，这些文件是和启动有关的服务，如下图 4.5.6.1 所示。一般一个服务都有一个配置文件告诉 Systemd 怎么去启动这个服务，我们先打开其中一个文件，并参考该文件来编写一个自启动服务。打开 vsftpd.service 文件，其内容如下所示：

```
1  [Unit]
2  Description=Vsftpd ftp daemon
3  After=network.target
4
5  [Service]
6  ExecStart=/usr/sbin/vsftpd
7
8  [Install]
9  WantedBy=multi-user.target
10
```

第 1 行，[Unit]字段表示一个单元，代表一个服务，该字段后面通常是对一个服务的描述；

第 2 行，Description 字段是对这个服务的简单描述；

第 3 行，表示当 network.target 这个模块或者服务启动完成以后，才会启动 vsftpd.service 这个服务；

第 5 行，[Service]字段表示针对.service 文件的配置，对于一个服务来说，该字段是最重要的。

第 6 行，ExecStart 表示启动当前服务的命令；

第 8 行，[Install]字段之后的配置需要通过 systemctl enable 命令来激活，通过通过 systemctl disable 命令来禁用，该部分的目标依赖模块一般是特定启动级别的 .target 文件；

第 9 行，表示依赖当前服务的模块，此处为 multi-user.target。

```
root@ATK-MP157:/lib/systemd/system# ls
alsa-restore.service                    runlevel5.target.wants
alsa-state.service                      runlevel6.target
apt-daily.service                       run-postinsts.service
apt-daily.timer                         serial-getty@.service
atk-qtapp-start.service                 shutdown.target
autovt@.service                         sigpwr.target
avahi-daemon.service                    sleep.target
avahi-daemon.socket                     slices.target
basic.target                            smartcard.target
bluetooth-brcmfmac-sleep.service        sockets.target
bluetooth.service                       sockets.target.wants
bluetooth.target                        sound.target
boot-complete.target                    sound.target.wants
busybox-klogd.service                   st-m4firmware-load.service
busybox-syslog.service                  suspend.target
```

图 4.5.6.1 启动服务文件

有关于 Systemd 的详细介绍大家可以在网上查询，下面我们直接参考 vsftpd.service 文件，照葫芦画瓢，在/lib/systemd/system 下也编写一个启动服务文件，该服务文件名称为 start_m4.service，内容如下：

```
[unit]
Description=start M4
After=multi-user.target

[Service]
Type=simple
user=root
ExecStart=/lib/firmware/test1.sh

[Install]
WantedBy=multi-user.target
```

Type=simple 表示 ExecStart 字段启动的进程为主进程，user=root 指定运行服务的用户为 root 用户，ExecStart=/lib/firmware/test1.sh 表示启动当前的服务命令为/lib/firmware/test1.sh，也就是说，运行 start_m4.service 这个服务，实际上就是执行/lib/firmware 下的 test1.sh 脚本。

保存编写的 start_m4.service 文件，然后执行如下命令，先设置文件为可执行权限，再使能服务，只有使能服务以后，添加的服务才会生效，如下图 4.5.6.2 所示：

```
vi start_m4.service          /* 新建 start_m4.service 文件 */
sync                         /* 修改好 start_m4.service 文件以后，同步一下缓存 */
chmod a+x start_m4.service          /* 修改 start_m4.service 为可执行权限 */
systemctl enable start_m4.service   /* 使能服务 start_m4.service */
```

```
root@ATK-MP157:/lib/systemd/system# vi start_m4.service
root@ATK-MP157:/lib/systemd/system# sync
root@ATK-MP157:/lib/systemd/system# chmod a+x start_m4.service
root@ATK-MP157:/lib/systemd/system# systemctl enable start_m4.service
Created symlink /etc/systemd/system/multi-user.target.wants/start_m4.service → /lib/systemd/system/start_m4.service.
root@ATK-MP157:/lib/systemd/system# sync
```

图 4.5.6.2

执行启动服务命令以后，系统自动在/etc/systemd/system/multi-user.target.wants 下创建了链接文件 start_m4.service，该文件软链接到了/lib/systemd/system/start_m4.service，系统运行以后会执行/etc/systemd/system/multi-user.target.wants/start_m4.service，本质上是执行

/lib/systemd/system/start_m4.service，服务使能成功。接下来需要重启开发板，开发板重启后打印信息如下图 4.5.6.3 所示，可以看到固件已经自动被加载和运行了，观察到开发板底板的 DS1 灯也在闪烁了。

```
[    8.250291] EXT4-fs (mmcblk2p2): mounted filesystem with ordered data mode. Opts: (null)
[    8.256993] ext4 filesystem being mounted at /boot supports timestamps until 2038 (0x7fffffff)
[    8.645058] systemd-journald[320]: Received client request to flush runtime journal.
[   10.872182] remoteproc remoteproc0: powering up m4
[   11.036992] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2232980
[   11.058862] ds18b20: module verification failed: signature and/or required key missing - tainting kernel
[   11.069261] dht11 dht11: dht11 device and driver matched successfully!
[   11.094301]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[   11.128860] ds18b20 ds18b20: ds18b20 device and driver matched successfully!
[   11.134593] ds18b20 ds18b20: Failed to request gpio
[   11.187435] virtio_rpmsg_bus virtio0: rpmsg host is online
[   11.230930]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[   11.236412] ds18b20: probe of ds18b20 failed with error -16
[   11.281326] remoteproc remoteproc0: remote processor m4 is now up
[   11.500971] mc: Linux media interface: v0.10
[   11.528094] using random self ethernet address
[   11.531092] using random host ethernet address
```

图 4.5.6.3 固件运行信息

经过以上设置，只要每次重启开发板，都可以看到固件 RPMsg_TEST_CM4.elf 被加载和运行，如果想关掉自动运行程序，可以执行如下指令，如下图 4.5.6.4 所示：

```
systemctl disable start_m4.service  /* 关闭服务 start_m4.service */
sync
```

```
root@ATK-MP157:~# systemctl disable start_m4.service
Removed /etc/systemd/system/multi-user.target.wants/start_m4.service.
root@ATK-MP157:~# sync
root@ATK-MP157:~# reboot
```

图 4.5.6.4 关闭 start_m4.service 服务

关闭服务后，系统自动将/etc/systemd/system/multi-user.target.wants/start_m4.service 文件删除，也只是删除了软连接文件，并未删除前面我们新建的/lib/systemd/system/start_m4.service 文件，所以如果想再次设置开机自动运行固件，直接执行 systemctl enable start_m4.service 这个命令来再次使能服务即可。

4.5.7 uboot 启动 M4 固件

1. rproc 命令的使用

下面我们来讲解如何通过 uboot 来启动 M4，我们先来了解 uboot 下的 rproc 命令的使用，首先，启动开发板以后，在 autoboot 变量变为 0 之前马上按下回车键，进入 uboot 命令行模式，然后，在 uboot 下执行 rproc 命令，或者执行 rproc --help 命令，可以看到 uboot 下弹出命令的使用方法，如下图 4.5.7.1 所示。

```
/* 查看 rproc 命令的使用 */
rproc
/* 或者执行如下命令查看命令的帮助信息 */
rproc --help
```

```
STM32MP> rproc
rproc - Control operation of remote processors in an SoC

Usage:
rproc  [init|list|load|start|stop|reset|is_running|ping]
                Where:
               [addr] is a memory address
               <id> is a numerical identifier for the remote processor
                    provided by 'list' command.
               Note: Remote processors must be initalized prior to usage
               Note: Services are dependent on the driver capability
                     'list' command shows the capability of each device

        Subcommands:
        init <id> - Enumerate and initalize the remote processor.
                          if id is not passed, initialize all the remote prcessors
        list    - list available remote processors
        load <id> [addr] [size]- Load the remote processor with binary
                          image stored at address [addr] in memory
        start <id>      - Start the remote processor(must be loaded)
        stop <id>       - Stop the remote processor
        reset <id>      - Reset the remote processor
        is_running <id> - Reports if the remote processor is running
        ping <id>       - Ping the remote processor for communication

STM32MP> rproc --help
rproc - Control operation of remote processors in an SoC

Usage:
rproc  [init|list|load|start|stop|reset|is_running|ping]
                Where:
               [addr] is a memory address
               <id> is a numerical identifier for the remote processor
                    provided by 'list' command.
               Note: Remote processors must be initalized prior to usage
               Note: Services are dependent on the driver capability
                     'list' command shows the capability of each device

        Subcommands:
```

图 4.5.7.1 查看 rproc 命令的使用

对以上命令的使用方法提示，我们介绍一下，如下表 4.5.7.1 所示。

命令	说明
rproc list	列出当前存在的协处理器
rproc load <id> [addr] [size]	加载协处理器的固件
rproc init	初始化协处理器
rproc start <id>	启动协处理器
rproc stop <id>	关闭协处理器
rproc reset <id>	复位协处理器
rproc is_running <id>	查看协处理器是否在运行
rproc ping <id>	ping 协处理器

表 4.5.7.1 rproc 命令的使用方法

<id>表示协处理器的 ID，在执行 rproc init 以后，可以初始化协处理器，然后执行 rproc list 命令可以列出有哪些协处理器，并可以看到对应协处理器的 ID。

2. 手动加载 M4 固件和启动 M4

以上命令的使用步骤，一般是如下（注意，以下操作，笔者是从开发板的 EMMC 启动的出厂系统来进行测试的，是在 uboot 的命令行模式下执行的命令）：

① 使用 ext4load 命令将 SD 卡或者 EMMC 中的命令读取到 DDR 中

没错，是读取到 DDR 中，毕竟启动 uboot 的时候此时 SRAM1~SRAM4 还不能够使用，M4 的固件暂时是放在 DDR 中。这里注意，如果出厂的系统是烧录到 EMMC 中的，文件系统分区是在 EMMC 里的，我们可以通过 uboot 下的 mmc 命令来查看，关于 uboot 下的命令，大家可以查看《【正点原子】STM32MP1 嵌入式 Linux 驱动开发指南》。

```
/* 切换到 EMMC，后面的参数，1 表示 EMMC，0 表示 SD 卡 */
mmc dev 1
mmc part          /* 查看 MMC 的分区 */

/* 如果是 SD 卡，则命令如下 */
mmc dev 0
mmc part          /* 查看 SD 卡的分区 */
```

如下图 4.5.7.2 所示查看的是 EMMC，EMMC 的第 3 个分区是文件系统，第 2 个分区是 boot 区域，boot 区域存放的是内核 uImage 和设备树文件。

```
STM32MP> mmc dev 1
switch to partitions #0, OK
mmc1(part 0) is current device
STM32MP> mmc part

Partition Map for MMC device 1  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000400      0x000013ff      "ssbl"
        attrs:  0x0000000000000000
        type:   8da63339-0007-60c0-c436-083ac8230908
        guid:   36743650-ae96-4f48-b379-3bf5d1caafda
  2     0x00001400      0x000213ff      "boot"
        attrs:  0x0000000000000004
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   f164934f-97fb-4742-9dea-7d070eeee4fe
  3     0x00021400      0x00e8fbff      "rootfs"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   491f6117-415d-4f53-88c9-6e0de54deac6
STM32MP>
```

图 4.5.7.2 查看 EMMC 的分区信息

如下图 4.5.7.3 所示，如果是 SD 卡，则 SD 卡的第 5 分区是文件系统，第 4 分区是 boot 区域。

```
STM32MP> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
STM32MP> mmc part

Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000022      0x00000221      "fsbl1"
        attrs:  0x0000000000000000
        type:   8da63339-0007-60c0-c436-083ac8230908
        guid:   152a2069-a438-45e0-b4fd-12a3267f2468
  2     0x00000222      0x00000421      "fsbl2"
        attrs:  0x0000000000000000
        type:   8da63339-0007-60c0-c436-083ac8230908
        guid:   35b5bc53-b880-4466-b988-3c3eac1a544a
  3     0x00000422      0x00001421      "ssbl"
        attrs:  0x0000000000000000
        type:   8da63339-0007-60c0-c436-083ac8230908
        guid:   4b65e3ae-aee9-468b-9754-53905beee0a1
  4     0x00001422      0x00021421      "boot"
        attrs:  0x0000000000000004
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   f0640705-bc17-4bda-9ab8-d5b11d8c11df
  5     0x00021422      0x01dacbdc      "rootfs"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   e91c4e10-16e6-4c0e-bd0e-77becf4a3582
```

图 4.5.7.3 查看 SD 卡的分区信息

内核 uImage 和设备树 stm32mp157d-atk.dtb 在 EMMC 的 2 分区，不信你可以执行如下命令看看，如下图 4.5.7.4 所示。

```
ext4ls mmc 1:2        /* 查看 EMMC 的 2 分区有什么 */
ext4ls mmc 0:4        /* 如果是 SD 卡，则查看 SD 卡的 4 分区有什么 */
```

```
STM32MP> ext4ls mmc 1:2
<DIR>         1024 .
<DIR>         1024 ..
              2943 boot.scr.uimg
<DIR>         1024 lost+found
<DIR>         1024 mmc0_extlinux
<DIR>         1024 mmc1_extlinux
             74892 stm32mp157d-atk.dtb
             74336 stm32mp157d-atk-hdmi.dtb
             75199 stm32mp157d-atk-mipi.dtb
           3632241 uInitrd
           1228872 alientek_1024x600.bmp
           2048072 alientek_1280x800.bmp
            261192 alientek_480x272.bmp
            768072 alientek_800x480.bmp
           8321176 uImage
<DIR>         1024 5.4.31-g8c3068500
             74935 stm32mp157d-atk-spdif.dtb
STM32MP>
```

图 4.5.7.4 查看 EMMC 的 2 分区

笔者事先将 M4 的固件 RPMsg_TEST_CM4.elf 传输到了开发板文件系统的/lib/firmware/目录下了。执行如下命令将此固件读取到 DDR 的地址${kernel_addr_r}处，可以将 kernel_addr_r 打印出来，其就是内核 uImage 的加载地址。如下图 4.5.7.5 所示，读取成功后，提示固件的大小为 2237256 字节。

```
/* 可以执行如下命令查看变量 kernel_addr_r 表示的地址 */
print kernel_addr_r
/*
 * 通过 ext4load 命令将 EMMC 的第 3 个分区（即文件系统分区）的
 * /lib/firmware/RPMsg_TEST_CM4.elf 文件读取到 DDR 的地址 kernel_addr_r
 * 地址 kernel_addr_r=0xc2000000 也就是 DDR 的地址，其实也是内核的加载地址
 */
ext4load mmc 1:3 ${kernel_addr_r} /lib/firmware/RPMsg_TEST_CM4.elf

/* 如果是 SD 卡，则是执行如下命令 */
ext4load mmc 0:5 ${kernel_addr_r} /lib/firmware/RPMsg_TEST_CM4.elf
```

```
STM32MP> print kernel_addr_r
kernel_addr_r=0xc2000000
STM32MP> ext4load mmc 1:3 ${kernel_addr_r} /lib/firmware/RPMsg_TEST_CM4.elf
2237256 bytes read in 78 ms (27.4 MiB/s)
STM32MP>
```

图 4.5.7.5 加载 M4 的固件

上图中，我们打印的地址 kernel_addr_r 等于 0xc2000000，该地址就是 DDR 的地址，可以查看参考手册的内存映射图，如下图 4.5.7.6 所示，这个地址就是在 DDR 中！

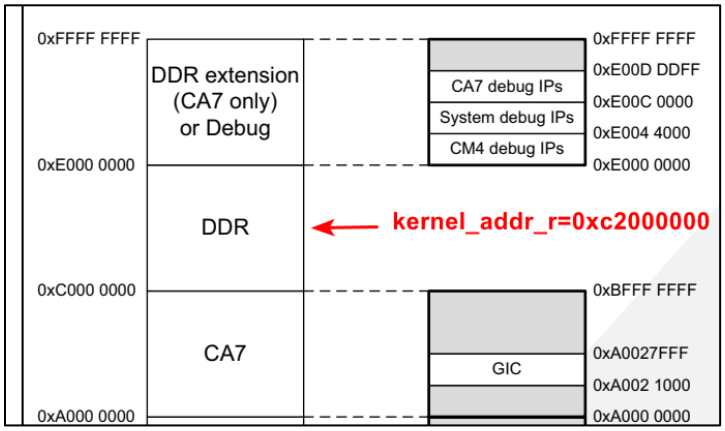


图 4.5.7.6 内存映射部分截图

kernel_addr_r 变量表示内核的加载地址，该变量在 include/configs/stm32mp1.h 中有进行定义，如下图 4.5.7.7 所示。

```
#define CONFIG_EXTRA_ENV_SETTINGS \
        "bootdelay=1\0" \
        "kernel_addr_r=0xc2000000\0" \
        "fdt_addr_r=0xc4000000\0" \
        "scriptaddr=0xc4100000\0" \
        "pxefile_addr_r=0xc4200000\0" \
        "splashimage=0xc4300000\0"  \
```

图 4.5.7.7 kernel_addr_r 是内核加载地址

② 初始化协处理器

可以执行 rproc list 命令列出当前有哪些协处理器，在没有执行 rproc init 命令初始化时，此时是看不到有任何协处理器的，如下图 4.5.7.8 所示，打印“0 - Name:'m4@10000000' type:'internal memory mapped' supports: load start stop reset is_running”，即此时有一个协处理器，ID 是 0。

```
rproc list                          /* 列出此时有哪些协处理器 */
rproc init                          /* 初始化协处理器 */
rproc list                          /* 再次列出协处理器 */
```

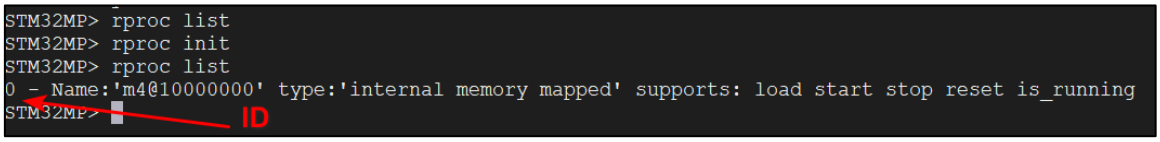


图 4.5.7.8 初始化协处理器

③ 加载固件

执行如下命令来加载 M4 的固件，其中变量 filesize 表示固件的大小，也就是上面 RPMsg_TEST_CM4.elf 文件的大小，其数值为 16 进制，由系统进行计算，不需要我们手动去计算。如下图 4.5.7.9 所示，打印出来的变量 filesize 的大小为 222348，将前面读取固件时提示的 10 进制 2237256 转化为 16 进制，刚好等于 222348，所以，以后加载 M4 固件，真的不需要你手动计算固件的大小，变量 filesize 已经帮你计算好！

```
/* 加载 M4 的固件到地址 kernel_addr_r，加载的长度是 filesize */
rproc load 0 ${kernel_addr_r} ${filesize}
print filesize                  /* 尝试打印变量 filesize 的值 */
```

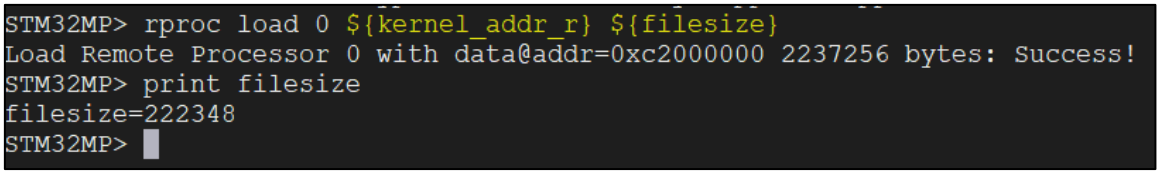


图 4.5.7.9 加载 M4 的固件

④ 启动协处理器

执行如下命令启动协处理器，然后再去查看协处理器是否已经启动了，如下图 4.5.7.10 所示，提示协处理器已经启动了。

```
rproc is_running 0              /* 查看协处理器是否已经启动了 */
rproc start 0                   /* 启动协处理器 */
rproc is_running 0              /* 查看协处理器是否已经启动了 */
```

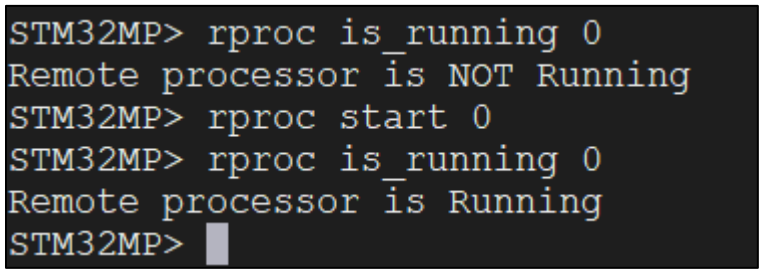


图 4.5.7.10 启动协处理器

自此，手动加载 M4 固件和启动 M4 已经讲解完毕。此时，你会留意到，为什么已经启动 M4 了，开发板底板的 DS1 灯没有闪烁呢？在第一章创建 M4 工程的时候，不是在 main.c 文件

中添加了控制 DS1 灯的代码了吗？其实，之所以灯没闪烁，这是因为我们在 M4 工程中开启了 OpenAMP，开启了 OpenAMP，就意味着要进行核间通信，而且 OpenAMP 内部也进行了不少操作，再者，咱们现在可是处于 uboot 的命令行模式下，在 uboot 下看不到现象也正常，如果将 OpenAMP 关闭了，再重新编译工程，再在 uboot 下重新加载和启动 M4 的固件，你会发现 DS1 灯在闪烁了。

当然了，想在 uboot 下看到现象，也可以直接编译我们的裸机例程来进行测试，在资料里有提供裸机例程的源码，如下图 4.5.7.11 所示，其中一个是基于 CubeIDE 的，另一个是基于 MDK 的，直接拿一个按键、LED 或者蜂鸣器的工程来重新编译，再去测试，就可以看到现象了。

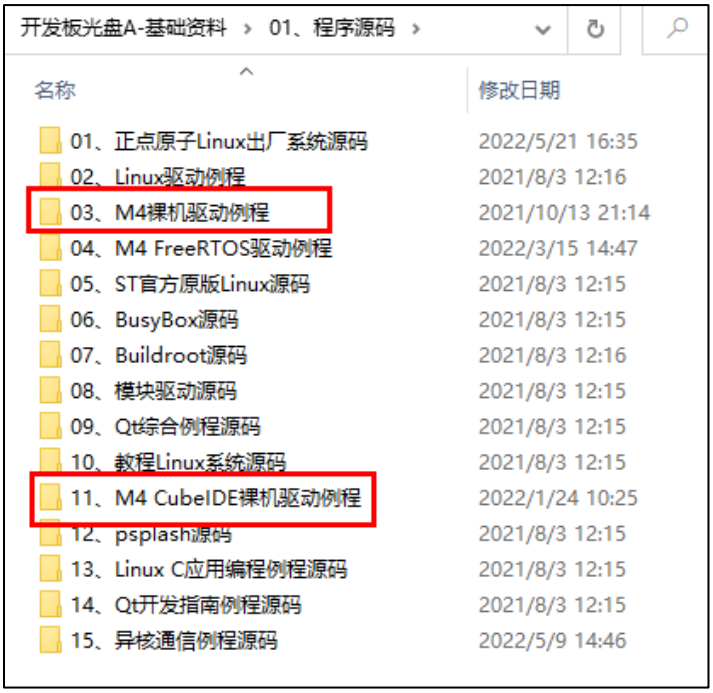


图 4.5.7.11 正点原子的 M4 工程代码

3. uboot 自动加载固件和启动 M4

以上的操作是通过手动执行 uboot 的命令方式来自动加载 M4 固件和启动 M4 的，可不可以设置 uboot 自动加载 M4 的固件并自动启动 M4？当然可以啦！下面我们讲解如何实现。

具体操作很简单，以上的方式是通过手动执行命令去实现的，我们也可以让 uboot 内部自动执行以上的命令。如果之前看了《【正点原子】STM32MP1 嵌入式 Linux 驱动开发指南》这个文档，或者看了《【正点原子】I.MX6U 嵌入式 Linux 驱动开发指南》这个文档，就应该对 Linux 启动的流程有个大概的理解。其中，uboot 下的 bootcm 这个环境变量起到引导内核启动的作用，即加载内核和设备树，并启动内核。uboot 下的 bootargs 这个环境变量保存着传递给内核的的一些重要参数，通过 bootargs 可以实现挂载文件系统。好，我们不妨将以上加载 M 固件、启动 M4 固件的命令，还有 bootcmd 以及 bootargs 这些环境变量相关的命令，把它“写死”在 uboot 源码里，然后重新编译 uboot，并替换开发板的 uboot，实现 uboot“永久”自动加载 M4 固件、启动 M4，并启动 Linux 系统？说干就干！

打开出厂 uboot 源码的 include/configs/stm32mp1.h 文件，在宏 CONFIG_EXTRA_ENV_SETTINGS 的后面添加如下代码，大约在 257 行之后，如下图 4.5.7.12 所示（注意，如下代码不建议复制粘贴来修改，会有格式问题！）。

```
/* 采用 EMMC 启动方式的修改方法： */
"boot_net_usb_start=true\0"\
"boot_m4_load=ext4load mmc 1:3 0xc2000000
/lib/firmware/RPMsg_TEST_CM4.elf\0"\
"boot_m4fw=rproc init;rproc load 0 0xc2000000 ${filesize};rproc start 0\0"
\
"now_boot=ext4load mmc 1:2 0xc2000000 uImage;ext4load mmc 1:2 0xc4000000
stm32mp157d-atk.dtb;bootm 0xc2000000 - 0xc4000000;boot\0"\
"bootcmd=run boot_m4_load;run boot_m4fw;sleep 5;setenv bootargs
'console=ttySTM0,115200 root=/dev/mmcblk2p3 rootwait rw';run now_boot\0"\


/* 采用 SD 卡启动方式的修改方法： */
"boot_net_usb_start=true\0"\
"boot_m4_load=ext4load mmc 0:5 0xc2000000
/lib/firmware/RPMsg_TEST_CM4.elf\0"\
"boot_m4fw=rproc init;rproc load 0 0xc2000000 ${filesize};rproc start 0\0"
\
"now_boot=ext4load mmc 0:4 0xc2000000 uImage;ext4load mmc 0:4 0xc4000000
stm32mp157d-atk.dtb;bootm 0xc2000000 - 0xc4000000;boot\0"\
"bootcmd=run boot_m4_load;run boot_m4fw;sleep 5;setenv bootargs
'console=ttySTM0,115200 root=/dev/mmcblk1p5 rootwait rw';run now_boot\0"\
```

```
#define CONFIG_EXTRA_ENV_SETTINGS \
        "bootdelay=1\0" \
        "kernel_addr_r=0xc2000000\0" \
        "fdt_addr_r=0xc4000000\0" \
        "scriptaddr=0xc4100000\0" \
        "pxefile_addr_r=0xc4200000\0" \
        "splashimage=0xc4300000\0" \
        "ramdisk_addr_r=0xc4400000\0" \
        "altbootcmd=run bootcmd\0" \
        "env_check=" \
                "env exists env_ver || env set env_ver ${ver};" \
                "if env info -p -d -q; then env save; fi;" \
                "if test \"$env_ver\" != \"$ver\"; then" \
                " echo \"*** Warning: old environment ${env_ver}\";" \
                " echo '* set default: env default -a; env save; reset';" \
                " echo '* update current: env set env_ver ${ver}; env save';" \
                "fi;\0" \
        STM32MP_BOOTCMD \
        STM32MP_ANDROID \
        PARTS_DEFAULT \
        BOOTENV \
        "boot_net_usb_start=true\0"\
        "boot_m4_load=ext4load mmc 1:3 0xc2000000 /lib/firmware/RPMsg_TEST_CM4.elf\0"\
        "boot_m4fw=rproc init;rproc load 0 0xc2000000 ${filesize};rproc start 0\0" \
        "now_boot=ext4load mmc 1:2 0xc2000000 uImage;ext4load mmc 1:2 0xc4000000 stm32mp157d-atk.dtb;bootm 0xc2000000 - 0xc4000000;boot\0"\
        "bootcmd=run boot_m4_load;run boot_m4fw;sleep 5;setenv bootargs 'console=ttySTM0,115200 root=/dev/mmcblk2p3 rootwait rw';run now_boot\0"\


#endif /* ifndef CONFIG_SPL_BUILD */
#endif /* ifdef CONFIG_DISTRO_DEFAULTS*/
```

图 4.5.7.12 添加代码

我们分析一下以上代码：

① boot_m4_load 这个环境变量表示将文件系统/lib/firmware/目录下的固件 RPMsg_TEST_CM4.elf 加载到 DDR 的地址 0xc2000000 处。

② boot_m4fw 这个环境变量用于初始化协处理器、加载协处理器和启动协处理器。

③ now_boot 这个环境变量用于从 EMMC 的分区 3 中读取内核 uImage、从 EMMC 的分区 2 中读取设备树 stm32mp157d-atk.dtb，最后通过 bootm 来启动内核和设备树。

④ bootcmd 是 uboot 启动以后会自动运行的环境变量，此处，我们设置它先运行①→再运行②→然后延时 5s 的时间→接着设置 bootargs 这个环境变量，即指定加载 EMMC 里的文件系

统→最后运行③。关于延时 5s 的时间，笔者的本意只是想让 uboot 暂停一会，然后可以观察一下现象，或者有的用户想在这个时候显示一些信息，我们可以通过暂停 5s 的时间来实现。如不需要暂停 5s，也可以删掉以上的代码“sleep 5”。

以上的修改方法，就是直接运行对应的环境变量就行了，简单粗暴，效果显著！

修改好之后，保存并退出，执行如下命令重新编译出厂的 uboot，如下图 4.5.7.13 所示。build.sh 文件是出厂 uboot 源码编写好的编译脚本，用于使能交叉编译器、配置 uboot 和编译 uboot。

```
./build.sh            /* 使能交叉编译器、配置 uboot、编译 uboot */
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/uboot$ vi include/configs/stm32mp1.h
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/uboot$ sync
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/uboot$ vi include/configs/stm32mp1.h
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/uboot$ sync
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/uboot$ ./build.sh
  CLEAN   dts/../arch/arm/dts
  CLEAN   dts
  CLEAN   examples/standalone
  CLEAN   tools
  CLEAN   tools/lib tools/common
  CLEAN   u-boot-nodtb.bin u-boot.lds u-boot.srec u-boot.cfg.configs u-boot.stm32.log u-boot.map u-boot.bin u-boot.cfg
m.map
  CLEAN   scripts/basic
  CLEAN   scripts/dtc
  CLEAN   scripts/kconfig
  CLEAN   include/config include/generated
  CLEAN   .config include/autoconf.mk.dep include/autoconf.mk include/config.h
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  YACC    scripts/kconfig/zconf.tab.c
  LEX     scripts/kconfig/zconf.lex.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
```

图 4.5.7.13 编译 uboot

耐心等待编译完成，最终生成 u-boot.stm32 这个文件，我们就是要用到这个文件！如下图 4.5.7.14 所示。

```
DTC     arch/arm/dts/stm32mp157d-dk1.dtb
DTC     arch/arm/dts/stm32mp157d-ed1.dtb
DTC     arch/arm/dts/stm32mp157d-ev1.dtb
DTC     arch/arm/dts/stm32mp157f-dk2.dtb
DTC     arch/arm/dts/stm32mp157f-ed1.dtb
DTC     arch/arm/dts/stm32mp157f-ev1.dtb
DTC     arch/arm/dts/stm32mp15xx-dhcom-pdk2.dtb
FDTGREP dts/dt-spl.dtb
SHIPPED dts/dt.dtb
CAT     u-boot-dtb.bin
COPY    u-boot.dtb
COPY    u-boot.bin
MKIMAGE u-boot.stm32
CFGCHK  u-boot.cfg
```

4.5.7.14 生成 u-boot.stm32 文件

下面我们通过“开发板光盘 A-基础资料\08、系统镜像\02、出厂系统镜像\01、STM32CubeProg 烧录固件包”来烧录系统，首先，我们复制一份 01、STM32CubeProg 烧录固件包（建议复制一份出来，以区别出厂的和自己修改的），如下图 4.5.7.15 所示是笔者复制出来的。

图 4.5.7.15 复制一份烧录包

进入到复制的 01、STM32CubeProg 烧录固件包目录下，新建一个 uboot1 文件夹，这个文件夹下存放的是上面我们编译出来的 u-boot.stm32 文件，如下图 4.5.7.16 所示。

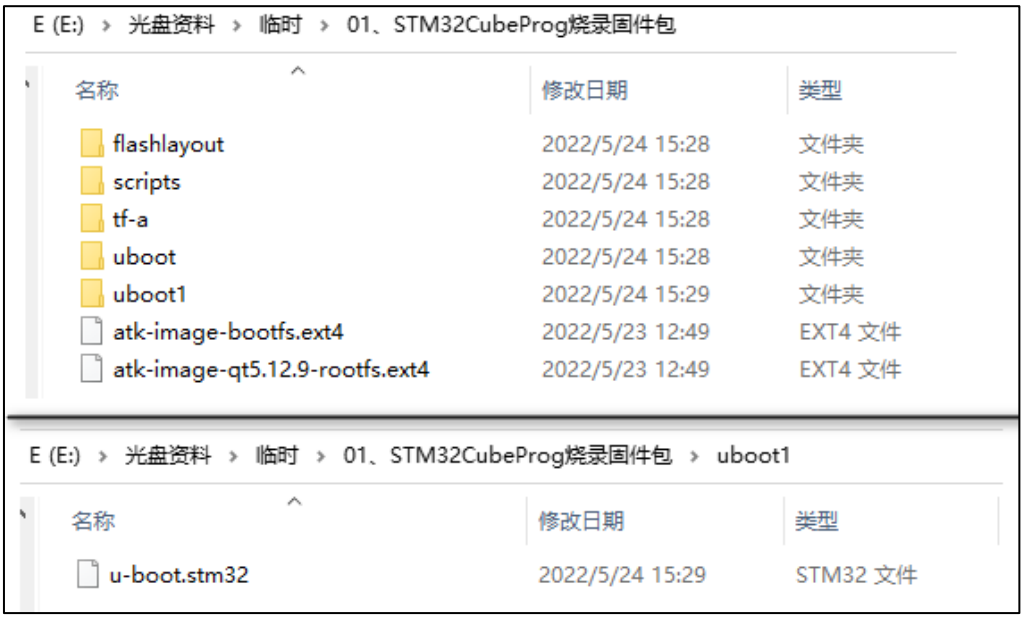


图 4.5.7.16 拷贝 u-boot.stm32 文件

打开 E:\光盘资料\临时\01、STM32CubeProg 烧录固件包\flashlayoutatk_emmc-stm32mp157d-atk-qt.tsv 文件，这个文件是用于烧录镜像到 EMMC 中的，如下图 4.5.7.17 所示是该文件原来的配置。

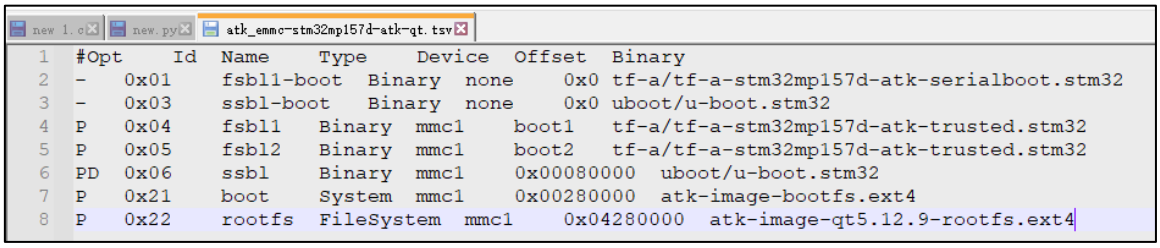


图 4.5.7.17 原来的配置

我们修改为如下图 4.5.7.18 所示。

图 4.5.7.18 修改后的

PD 表示删除并更新，1 处的 PD 作用就是删除原来的 uboot（包括之前保存的 uboot 的环境变量！）并更新（重新烧录），那么，烧录以后，uboot 里的环境变量就是 uboot 源码默认的环

境变量。PE 表示不更新，2 处的 PE 就表示不会更新文件系统，也就是说原来 EMMC 里的文件系统保持不变。如果你想重新烧录 EMMC，也可以将以上 2 处的 PE 改为 P。注意 3 处，此时烧录的是 uboot1 里的 u-boot.stm32 文件，别搞错了，要烧录的是我们自己编译出来的 uboot 文件。

关于以上“.tsv”文件的内容，相信看过《【正点原子】STM32MP1 嵌入式 Linux 驱动开发指南》的就知道它的作用，笔者这里不再花篇幅去重复分析，大家可以去看教程，去全局搜索对应的内容，正点原子的教程内容还是比较丰富的，看完教程，可以入门和提高，很多疑难的问题都是由很多个小问题集成的，基础扎实了，问题解决起来就快了。如下图 4.5.7.19 所示，是教程里的截图。

- **‘P’**：向分区或者设备烧写固件。

STM32CubeProgrammer 本质是通过 uboot 来烧写系统的，也就是先把 uboot 加载到板子的 DDR 里面并运行，然后使用 uboot 来烧写系统。uboot 会请求需要烧写的二进制文件，然后将其烧写到指定的分区或者 Falsh 设备里面。

针对‘P’选项，还有另外两个可以搭配使用的小伙伴：

- **‘E’**：空分区或设备，表示对应的分区或设备不更新，相关的 Id 项会被跳过。
- **‘D’**：删除分区或设备。

允许的组合选项如下所示：

- **‘-’**：空选型。
- **‘P’**：更新分区或设备，也就是向分区或设备烧写固件。
- **‘PE’**：不更新，也就是指定某个分区或者设备不需要烧写固件，这样我们就可以单独只更新 tf-a、uboot、kernel 或者 rootfs。
- **‘PD’**：删除并更新，也可以写作 DP。
- **‘PDE’**：删除并且保持为空，也可以写作 PED/DPE/DEP/EPD/EDP。

②、Id 域

STM32CubeProgrammer 通过 Id 域来确定烧写方法，会通过 Id 域来识别下一个要烧写到设备里面的二进制文件：

- ROM 或者 FSBL：二进制文件要加载到 RAM 中
- SSBL(uboot)：二进制文件要烧写到 Flash 中。

FlashLayout 支持的 Id 范围如表 6.2.3.2 所示：

范围	分区
0x01~0x0F	带有 STM32 头部信息的 Boot 分区，如 SSBL、FSBL、其他(TEE 或 M4 固件)
0x10~0xF0	不带头部的用户编程分区，如 uiamge、dtb、rootfs、vendorfs、userfs

表 6.2.3.2 FlashLayout 支持的 Id 范围

图 4.5.7.19 Linux 驱动教程部分截图

保存以上修改，接下来开始烧录，烧录的步骤请参考开发板光盘 A-基础资料\10、用户手册\《【正点原子】STM32MP157 快速体验》的第二章。如下图 4.5.7.20 所示。

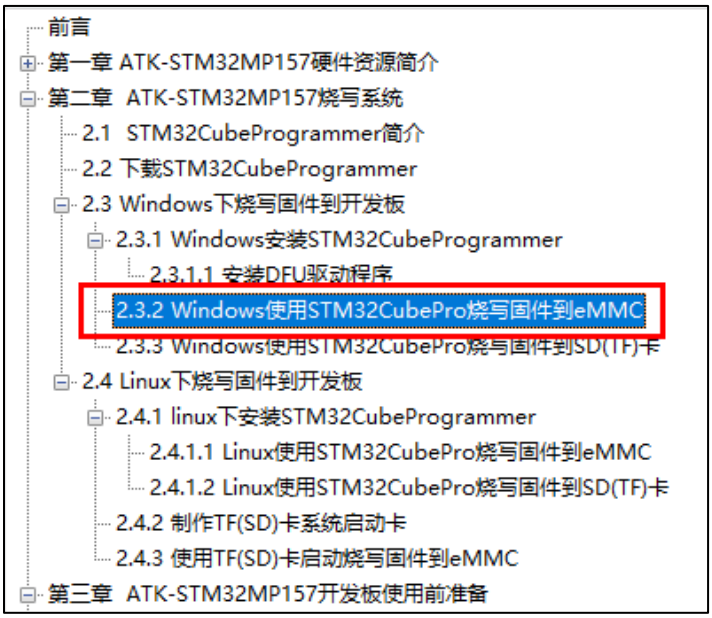


图 4.5.7.20 烧录参考文档

烧录完成后，开发板重新从 EMMC 启动，在启动 uboot 时，打印语句“Load Remote Processor 0 with data@addr=0xc2000000 2249860 bytes: Success!”，然后停留 5s 的时间后就继续启动内核，如下图 4.5.7.21 所示。

```
U-Boot 2020.01-stm32mp-r1 (May 24 2022 - 15:15:41 +0800)

CPU: STM32MP157DAA Rev.Z
Model: STMicroelectronics STM32MP157D eval daughter
Board: stm32mp1 in trusted mode (st,stm32mp157d-atk)
DRAM:  1 GiB
Clocks:
- MPU : 800 MHz
- MCU : 208.878 MHz
- AXI : 266.500 MHz
- PER : 24 MHz
- DDR : 533 MHz
WDT:   Started with servicing (32s timeout)
NAND:  0 MiB
MMC:   STM32 SD/MMC: 0, STM32 SD/MMC: 1
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
invalid MAC address in OTP 00:00:00:00:00:00
Net:
Error: ethernet@5800a000 address not set.
No ethernet found.

lcd_id  = 02
Hit any key to stop autoboot:  0                    停留5s
2249860 bytes read in 78 ms (27.5 MiB/s)
Load Remote Processor 0 with data@addr=0xc2000000 2249860 bytes: Success!
```

图 4.5.7.21 启动在 uboot 阶段停留 5s

成功进入文件系统中以后，可以看到开发板底板的 DS1 灯在闪烁了，程序完美运行。我们也可以执行如下命令，手动搜索一下和“m4”有关的 log，如下图 4.5.7.22 所示，提示已经成运行了 M4！这种方式其实和 4.5.6 小节的很相似，都实现了自启动，只不过一个是进入文件系统中后才开始执行，一个是在 uboot 下开始执行。

```
dmesg | grep m4                    /* 筛选出带有“m4”字眼的 log 信息 */
```

```
root@ATK-MP157:~# dmesg | grep m4
[    1.105939] remoteproc remoteproc0: releasing m4
[    1.157394] remoteproc remoteproc0: releasing m4
[    1.192691] remoteproc remoteproc0: releasing m4
[    1.282956] remoteproc remoteproc0: releasing m4
[    3.792024] stm32-rproc mlahb:m4@10000000: wdg irq registered
[    3.797519] remoteproc remoteproc0: m4 is available
[    3.801543] remoteproc remoteproc0: powering up m4
[    3.815339]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[    3.829561]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[    3.842838] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:~#
```

图 4.5.7.22 筛选出关键的 log

如果下次我们想加载另一个文件，例如加载第六章的 RPMsg_UART_CM4.elf 文件，此时不需要重新去修改、编译和替换 uboot 了，我们可以直接设置 uboot 的环境变量来解决。首先，重启开发板，进入 uboot 命令行，执行如下命令就可以了，如下图 4.5.23 所示。

```
print bootcmd                      /* 打印 bootcmd 这个环境变量 */
print boot_m4_load                 /* 打印 print boot_m4_load 这个环境变量 */
/* 重新设置 boot_m4_load 这个环境变量 */
setenv boot_m4_load "ext4load mmc 1:3 0xc2000000
/lib/firmware/RPMsg_UART_CM4.elf"
saveenv                            /* 保存环境变量 */
```

```
STM32MP> print bootcmd
bootcmd=run boot_m4_load;run boot_m4fw;sleep 5;setenv bootargs 'console=ttySTM0,115200 root=/dev/mmcblk2p3 rootwait rw';run now_boot
STM32MP> print boot_m4_load
boot_m4_load=ext4load mmc 1:3 0xc2000000 /lib/firmware/RPMsg_TEST_CM4.elf
STM32MP> setenv boot_m4_load "ext4load mmc 1:3 0xc2000000 /lib/firmware/RPMsg_UART_CM4.elf"
STM32MP> saveenv
Saving Environment to MMC... Writing to redundant MMC(1)... OK
STM32MP>
```

图 4.5.23 设置和保存环境变量

保存好环境变量以后，重启开发板，进入文件系统中以后，可以看到 M4 成功运行了，如下图 4.5.24 所示。现象也如第六章的实验一样，接下来 A7 和 M4 之间就可以通过虚拟串口 /dev/ttyRPMSG0 来互发数据了（注意，必须 A7 先发数据），具体测试方法可参考第六章的实验。

```
dmesg | grep m4                   /* 筛选出带有“m4”字眼的 log 信息 */
ls /dev/ttyRPMSG0                  /* 列出/dev/ttyRPMSG0 */
```

```
root@ATK-MP157:~# dmesg | grep m4
[    1.090098] remoteproc remoteproc0: releasing m4
[    1.141366] remoteproc remoteproc0: releasing m4
[    1.176747] remoteproc remoteproc0: releasing m4
[    1.273053] remoteproc remoteproc0: releasing m4
[    3.768597] stm32-rproc mlahb:m4@10000000: wdg irq registered
[    3.774960] remoteproc remoteproc0: m4 is available
[    3.779803] remoteproc remoteproc0: powering up m4
[    3.791146]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[    3.804931]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[    3.819088] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:~# ls /dev/ttyRPMSG0
/dev/ttyRPMSG0
root@ATK-MP157:~#
```

图 4.5.24 M4 成功运行了

4. 使用启动脚本 boot.scr.uimg

以上自动加载 M4 固件和启动启动 M4 的操作，需要编译 uboot，如果不想编译 uboot，也可以使用 DISTRO 启动脚本 mmc1_boot.scr.uimg（用于 EMMC 启动）或者 mmc0_boot.scr.uimg（用于 SD 卡启动），下面，我们就来介绍这种不需要编译 uboot 的方式。

在 Ubuntu 中用 vi 命令新建一个文件，文件的名字是 boot.scr.cmd，如果是 EMMC 启动方式，该文件的内容如下：

```
# EMMC 启动方式
#load M4 firmware
if ext4load mmc 1:3 ${kernel_addr_r} /lib/firmware/RPMsg_TEST_CM4.elf
then
  rproc init
  rproc load 0 ${kernel_addr_r} ${filesize}
  rproc start 0
fi

#load kernel and device tree
ext4load mmc 1:2 ${kernel_addr_r} uImage
ext4load mmc 1:2 ${fdt_addr_r} stm32mp157d-atk.dtb

# rootfs
env set bootargs root=/dev/mmcblk2p3 rootwait rw console=ttySTM0,115200

# start kernel
bootm ${kernel_addr_r} - ${fdt_addr_r}
```

如果是 SD 卡启动方式，文件 boot.scr.cmd 的内容如下：

```
# SD 卡启动方式
#load M4 firmware
if ext4load mmc 0:5 ${kernel_addr_r} /lib/firmware/RPMsg_TEST_CM4.elf
then
  rproc init
  rproc load 0 ${kernel_addr_r} ${filesize}
  rproc start 0
fi

#load kernel and device tree
ext4load mmc 0:4 ${kernel_addr_r} uImage
ext4load mmc 0:4 ${fdt_addr_r} stm32mp157d-atk.dtb

# rootfs
env set bootargs root=/dev/mmcblk1p5 rootwait rw console=ttySTM0,115200

# start kernel
```

```
bootm ${kernel_addr_r} - ${fdt_addr_r}
```

执行如下命令来编译生成对应的文件，若是 EMMC 启动方式，则编译生成 mmc1_boot.scr.uimg 文件，若是 SD 卡启动方式，则编译生成 mmc0_boot.scr.uimg 文件，如下图 4.5.25 所示。注意，根据是 EMMC 启动还是 SD 卡启动，boot.scr.cmd 文件的内容不同。

```
mkimage -T script -C none -A arm -d boot.scr.cmd mmc1_boot.scr.uimg
mkimage -T script -C none -A arm -d boot.scr.cmd mmc0_boot.scr.uimg
```

```
alientek@ubuntu:~/linshi$ ls
boot.scr.cmd
alientek@ubuntu:~/linshi$ mkimage -T script -C none -A arm -d boot.scr.cmd mmc1_boot.scr.uimg
Image Name:
Created:      Tue May 24 19:52:39 2022
Image Type:   ARM Linux Script (uncompressed)
Data Size:    451 Bytes = 0.44 KiB = 0.00 MiB
Load Address: 00000000
Entry Point:  00000000
Contents:
   Image 0: 443 Bytes = 0.43 KiB = 0.00 MiB
alientek@ubuntu:~/linshi$ ls
boot.scr.cmd  mmc1_boot.scr.uimg
alientek@ubuntu:~/linshi$
```

图 4.5.25 编译文件

如果是 EMMC 启动方式，将编译出来的 mmc1_boot.scr.uimg 文件拷贝到开发板文件系统的/boot 目录下，如果是 SD 卡启动方式，则将 mmc0_boot.scr.uimg 文件拷贝到开发板文件系统的/boot 目录下。如下图 4.5.26 所示，笔者是从 EMMC 启动的开发板，拷贝的是 mmc1_boot.scr.uimg 文件。

```
alientek@ubuntu:~/linshi$ scp mmc1_boot.scr.uimg root@192.168.1.15:/boot
mmc1_boot.scr.uimg
                      100%  515   260.4KB/s   00:00
alientek@ubuntu:~/linshi$
```

图 4.5.26 使用 scp 命令拷贝文件

如果是 EMMC 启动，则将文件系统/boot 目录下 mmc1_extlinux 文件夹删除，或者可以不删除，可以改为别的名字来进行备份，笔者是将名字改为 mmc1 了。若是 SD 卡启动，同样的可以删除文件系统/boot 目录下的 mmc0_extlinux 文件夹，或者可以改一个名字。操作完成后，执行 sync 命令同步一下缓存，再重启开发板，如下图 4.5.27 所示。

```
mv mmc1_extlinux mmc1          /* 修改 mmc1_extlinux 名字为 mmc1，相当于备份 */
sync                           /* 同步缓存 */
```

```
root@ATK-MP157:/boot# ls
5.4.31-g8c3068500       lost+found                  stm32mp157d-atk-mipi.dtb
alientek_1024x600.bmp   mmc0_extlinux               stm32mp157d-atk-spdif.dtb
alientek_1280x800.bmp   mmc1_boot.scr.uimg          uImage
alientek_480x272.bmp    mmc1_extlinux               uInitrd
alientek_800x480.bmp    stm32mp157d-atk.dtb
boot.scr.uimg           stm32mp157d-atk-hdmi.dtb
root@ATK-MP157:/boot# mv mmc1_extlinux mmc1
root@ATK-MP157:/boot# ls
5.4.31-g8c3068500       lost+found                  stm32mp157d-atk-mipi.dtb
alientek_1024x600.bmp   mmc0_extlinux               stm32mp157d-atk-spdif.dtb
alientek_1280x800.bmp   mmc1                        uImage
alientek_480x272.bmp    mmc1_boot.scr.uimg          uInitrd
alientek_800x480.bmp    stm32mp157d-atk.dtb
boot.scr.uimg           stm32mp157d-atk-hdmi.dtb
root@ATK-MP157:/boot#
root@ATK-MP157:/boot# sync
root@ATK-MP157:/boot#
```

图 4.5.27 修改 mmc1_extlinux 目录名字

重启开发板以后，成功进入文件系统，可以发现开发板底板的 DS1 灯在闪烁，M4 成功启动了，查看关键的 log 信息，如图 4.5.28 所示。

```
root@ATK-MP157:~# dmesg | grep m4
[    1.096394] remoteproc remoteproc0: releasing m4
[    1.147711] remoteproc remoteproc0: releasing m4
[    1.182994] remoteproc remoteproc0: releasing m4
[    1.273248] remoteproc remoteproc0: releasing m4
[    3.778704] stm32-rproc mlahb:m4@10000000: wdg irq registered
[    3.785026] remoteproc remoteproc0: m4 is available
[    3.789909] remoteproc remoteproc0: powering up m4
[    3.801228]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[    3.815008]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[    3.829135] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:~#
```

图 4.5.28 查看 log

默认情况下，开发板的启动由 mmc1_extlinux（EMMC 启动）或者 mmc0_extlinux 里的 stm32mp157d-atk_extlinux.conf 文件控制，uboot 启动脚本会去读取配置该文件，如果找不到 mmc1_extlinux 或者 mmc0_extlinux 目录（相当于找不到 stm32mp157d-atk_extlinux.conf 文件），那么 uboot 启动脚本会去读取 mmc1_boot.scr.uimg（EMMC 启动）或 mmc0_boot.scr.uimg（SD 卡启动）文件，以完成启动。

关于通用 DISTRO 引导脚本，感兴趣的小伙伴可以去自行研究，此处我们只是讲解怎么使用，怎么修改。

第五章 基于 RPMsg 实现异核通信

远程处理器消息传递 (RPMsg)是 OpenAMP 的一部分，RPMsg 是一种基于 Virtio 的消息总线，用于实现的是消息传递，在 RPMsg 通道建立后就可使用 RPMsg API 在主处理器与远程处理器软件环境之间进行处理器间通信（IPC）。本章我们来了解 RPMsg 组件相关的 API，本章分为如下几部分：

5.1 Linux 下 RPMsg 相关驱动文件
5.2 OpenAMP 库中的 API
5.3 基于 RPMsg 的异核通信实验

5.1 Linux 下 RPMsg 相关驱动文件

在 Linux 内核源码的 Documentation/rpmsg.txt 下有 RPMsg 驱动相关 API 介绍，在 Linux 内核源码的 drivers/rpmsg 目录下就是 RPMsg 驱动，如下图 5.1.1 所示，文件名中带有“glink”、“smd”字眼的一般是用于高通的平台。我们主要关注的是 rpmsg_core.c、virtio_rpmsg_bus.c、rpmsg_char.c 和 rpmsg_tty.c 这几个文件，运行 Linux 操作系统的主处理器可以通过调用这些驱动文件中的 API 来给协处理器发送消息。

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/rpmsg$ ls
built-in.a        qcom_glink_native.c  rpmsg_char.c       rpmsg_tty.o
Kconfig           qcom_glink_native.h  rpmsg_core.c       virtio_rpmsg_bus.c
Makefile          qcom_glink_rpm.c     rpmsg_core.o       virtio_rpmsg_bus.o
modules.builtin   qcom_glink_smem.c    rpmsg_internal.h
modules.order     qcom_smd.c           rpmsg_tty.c
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/drivers/rpmsg$
```

图 5.1.1 RPMsg 驱动源码

在 M4 工程的 Middlewares\Third_Party\OpenAMP 目录下有 rpmsg.c 和 rpmsg_virtio.c 文件，如下图 5.1.2 所示，协处理器可以通过调用这些文件中的 API 来给主处理器发送消息。

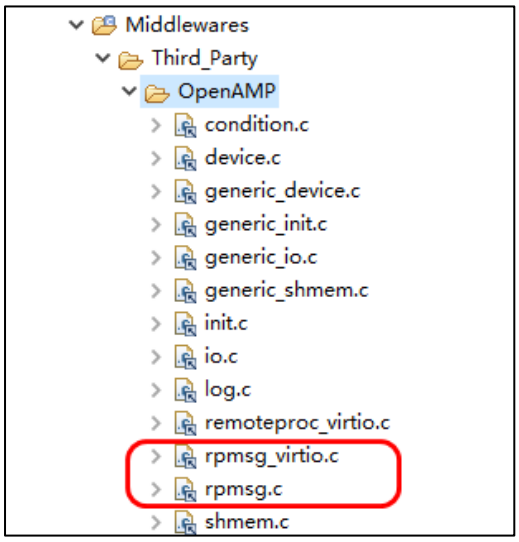


图 5.1.2 M4 工程目录

在 Linux 内核源码的 include/uapi/linux/virtio_ids.h 文件中可以找到如下定义，这些是 Virtio 设备的设备 ID，可以看到，Virtio RPMsg 设备 ID 为 7，另外，虚拟串口的设备 ID 为 11，系统通过这些 ID 来识别这是哪一种 Virtio 设备：

```
#define VIRTIO_ID_NET           1    /* virtio net */
#define VIRTIO_ID_BLOCK         2    /* virtio block */
#define VIRTIO_ID_CONSOLE       3    /* virtio console */
#define VIRTIO_ID_RNG           4    /* virtio rng */
#define VIRTIO_ID_BALLOON       5    /* virtio balloon */
#define VIRTIO_ID_RPMSG         7    /* virtio remote processor messaging */
#define VIRTIO_ID_SCSI          8    /* virtio scsi */
#define VIRTIO_ID_9P            9    /* 9p virtio console */
#define VIRTIO_ID_RPROC_SERIAL 11    /* virtio remoteproc serial link */
#define VIRTIO_ID_CAIF          12   /* Virtio caif */
#define VIRTIO_ID_GPU           16   /* virtio GPU */
#define VIRTIO_ID_INPUT         18   /* virtio input */
#define VIRTIO_ID_VSOCK         19   /* virtio vsock transport */
#define VIRTIO_ID_CRYPTO        20   /* virtio crypto */
#define VIRTIO_ID_IOMMU         23   /* virtio IOMMU */
#define VIRTIO_ID_FS            26   /* virtio filesystem */
#define VIRTIO_ID_PMEM          27   /* virtio pmem */
```

OpenAMP 库下有如下定义：

```
/* VirtIO rpmsg device id */
#define VIRTIO_ID_RPMSG_          7
```

定义 VIRTIO_ID_RPMSG 为 7，此值和 OpenAMP 库中的 VIRTIO_ID_RPMSG_值必须一样，两者为 7。在 Linux 内核源码的 drivers/rpmsg/virtio_rpmsg_bus.c 文件中，rpmsg_init()函数会注册 Virtio 设备驱动，通过 id_table 中的 Virtio 设备 ID（VIRTIO_ID_RPMSG，值为 7）和 OpenAMP 库中的 VIRTIO_ID_RPMSG_值匹配，这样主处理器的 Virtio 匹配到远程处理器，如匹配成功，则注册 Virtio 设备，注册成功后，A7 内核打印“registered virtio0 (type 7)”。

Virtio 设备注册成功后，通过 rpmsg_probe()函数配置缓冲区（Vring Buffer），一半缓冲区用于发送消息数据，一半缓冲区用于接收消息数据，如下图 5.1.3 所示。当 virtqueue 和 Virtio 设备准备就绪后，A7 内核则打印“rpmsg host is online”，提示 RPMsg 主处理器在线。接下来就可以通过 rpmsg_ns_cb()函数创建 RPMsg 通道了。RPMsg Virtio 的初始化过程如下图 5.1.4 所示。

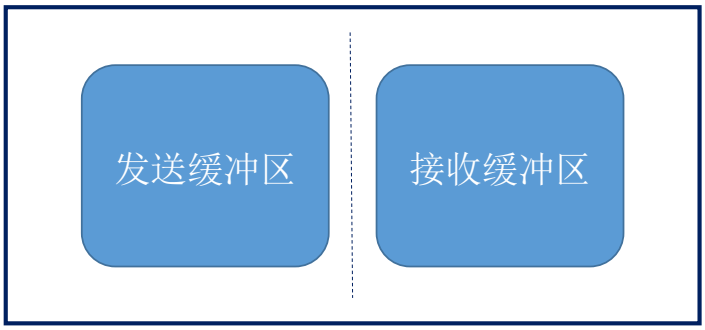


图 5.1.3 Vring Buffer

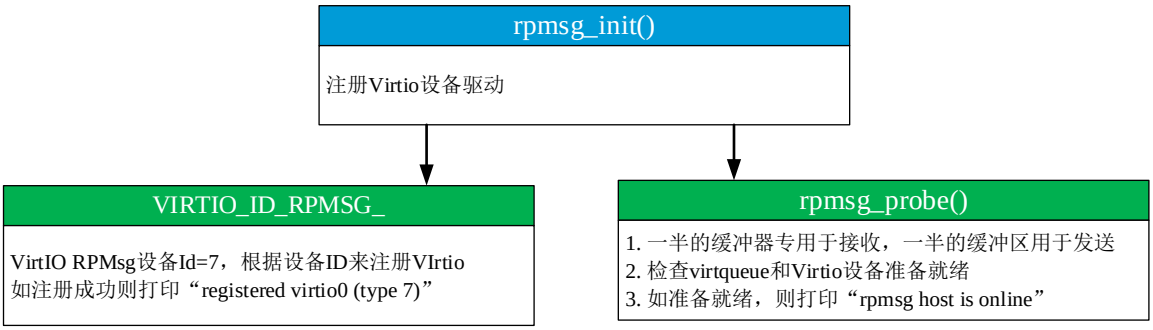


图 5.1.4 RPMsg Virtio 初始化

5.1.1 相关的结构体

首先来看看几个重要的结构体，打开 Linux 内核源码的 include/linux/rpmsg.h 文件，找到 rpmsg_device 和 rpmsg_endpoint 结构体。

1. rpmsg_device 结构体

rpmsg_device 是属于 RPMsg 总线的设备的结构体，其中，成员变量 id 是设备 ID，设备 ID 用于 RPMsg 驱动程序和设备之间的匹配。

```
struct rpmsg_device {          /* 属于 RPMsg 总线的设备 */
  struct device dev;           /* 设备结构体 */
  struct rpmsg_device_id id;   /* 设备 ID（用于匹配 RPMsg 驱动程序和设备） */
  char *driver_override;       /* 驱动程序名称以强制匹配 */
  u32 src;                     /* 本地地址 */
  u32 dst;                     /* 目的地址 */
  struct rpmsg_endpoint *ept;  /* RPMsg 通道的端点 */
  bool announce;               /* 如果设置，RPMsg 将宣布此通道的创建/删除 */
  /* RPMsg 设备回调函数结构体 */
  const struct rpmsg_device_ops *ops;
};
```

rpmsg_device_ops 是 RPMsg 设备回调函数结构体，在 Linux 内核源码的 drivers/rpmsg/rpmsg_internal.h 文件下有定义。

```
struct rpmsg_device_ops { /* rpmsg_device 回调函数 */
  /* 创建端点（必须） */
  struct rpmsg_endpoint *(*create_ept)(struct rpmsg_device *rpdev,
                           rpmsg_rx_cb_t cb, void *priv,
                           struct rpmsg_channel_info chinfo);
  /* 宣布新通道的存在，可选 */
  int (*announce_create)(struct rpmsg_device *ept);
  /* 宣布通道销毁，可选 */
  int (*announce_destroy)(struct rpmsg_device *ept);
};
```

其中 announce_create 和和 announce_destroy 是可选的，因为后端可能会通过创建端点隐式地通告新通道。

在 virtio_rpmsg_bus.c 文件中找到如下定义，通过指定 virtio_rpmsg_ops 结构体成员即可后续调用对应的回调函数：

```
static const struct rpmsg_device_ops virtio_rpmsg_ops = {
        .create_ept = virtio_rpmsg_create_ept,
        .announce_create = virtio_rpmsg_announce_create,
        .announce_destroy = virtio_rpmsg_announce_destroy,
};
```

如下表 5.1.1.1 是回调函数说明：

函数	描述
virtio_rpmsg_create_ept ()	创建一个 rpmsg 端点
virtio_rpmsg_announce_create ()	创建新通道，且告知远程处理器通道存在
virtio_rpmsg_announce_destroy ()	销毁通道，且告知远程处理器通道被销毁

表 5.1.1.1 回调函数说明

2. rpmsg_endpoint 结构体

rpmsg_endpoint 结构体是 RPMsg 端点结构体，用于将本地 RPMsg 地址绑定到对应的用户中，其指定了 RPMsg 通道的设备，对应的回调函数，其定义如下：

```
struct rpmsg_endpoint {          /* 将本地 RPMsg 地址绑定到其用户 */
  struct rpmsg_device *rpdev;    /* RPMsg 通道设备 */
  struct kref refcount;          /* 当此值为零时，ept 端点被释放 */
  rpmsg_rx_cb_t cb;              /* rx 回调处理程序 */
  struct mutex cb_lock;          /* 必须在访问/更改 cb 之前进行 */
  u32 addr;                     /* 本地 RPMsg 地址 */
  void *priv;                    /* 供驱动使用的私人数据 */
  /*  rpmsg 端点回调函数结构体 */
  const struct rpmsg_endpoint_ops *ops;
};
```

rpmsg_endpoint_ops 是 RPMsg 端点回调函数结构体，在 Linux 内核源码的 drivers/rpmsg/rpmsg_internal.h 文件下有定义。

```
struct rpmsg_endpoint_ops {          /* rpmsg_endpoint 处理程序结构体 */
  /* destroy_ept 处理程序 */
  void (*destroy_ept)(struct rpmsg_endpoint *ept);
  /* send 处理器程序*/
  int (*send)(struct rpmsg_endpoint *ept, void *data, int len);
  /* sendto 处理程序 */
  int (*sendto)(struct rpmsg_endpoint *ept, void *data, int len, u32 dst);
  /* send_offchannel 处理程序 */
  int (*send_offchannel)(struct rpmsg_endpoint *ept,
                         u32 src, u32 dst,
                         void *data, int len);
  /* trysend 处理程序 */
  int (*trysend)(struct rpmsg_endpoint *ept, void *data, int len);
  /* trysendto 处理程序 */
  int (*trysendto)(struct rpmsg_endpoint *ept, void *data, int len, u32 dst);
  /* trysend_offchannel 处理程序 */
  int (*trysend_offchannel)(struct rpmsg_endpoint *ept,
```

```
                                u32 src, u32 dst,
                                void *data, int len);
  /* poll 处理程序 */
  __poll_t (*poll)(struct rpmsg_endpoint *ept,
                   struct file *filp,
                   poll_table *wait);
  /* get_buffer_size 处理程序 */
  int (*get_buffer_size)(struct rpmsg_endpoint *ept);
};
```

在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 中找到如下结构体，此结构体成员指定对应的回调函数，当在程序中调用 rpmsg_endpoint_ops 处理程序的某个成员时，会执行对应的回调函数。

```
static const struct rpmsg_endpoint_ops virtio_endpoint_ops = {
  .destroy_ept = virtio_rpmsg_destroy_ept,
  .send = virtio_rpmsg_send,
  .sendto = virtio_rpmsg_sendto,
  .send_offchannel = virtio_rpmsg_send_offchannel,
  .trysend = virtio_rpmsg_trysend,
  .trysendto = virtio_rpmsg_trysendto,
  .trysend_offchannel = virtio_rpmsg_trysend_offchannel,
  .get_buffer_size = virtio_get_buffer_size,
};
```

如下表 5.1.1.2 是回调函数说明：

函数	描述
virtio_rpmsg_destroy_ept()	销毁现有的 rpmsg 端点
virtio_rpmsg_send()	通过默认端点给远程处理器默认端点发送数据（阻塞）
virtio_rpmsg_sendto()	通过默认端点和 dts 给远程处理器发送数据（阻塞）
virtio_rpmsg_send_offchannel()	通过指定端点（src 和 dts）给远程处理器发送数据
virtio_rpmsg_trysend()	通过默认端点给远程处理器默认端点发送数据（非阻塞）
virtio_rpmsg_trysendto()	通过默认端点和 dts 给远程处理器发送数据（非阻塞）
virtio_rpmsg_trysend_offchannel()	通过指定端点（src 和 dts）给远程处理器发送数据（非阻塞）
virtio_get_buffer_size()	获取消息的有效负载（除去消息头后的长度）

表 5.1.1.2 回调函数说明

以上的发送函数都调用了 virtio_rpmsg_send_offchannel()函数来实现发送功能，我们后面会重点讲解此函数。

3. virtio_rpmsg_channel

在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 中找到 virtio_rpmsg_channel 结构体，此结构体用于指定是哪个通道，以及使用的是此通道的那个处理器。

```
struct virtio_rpmsg_channel {
  struct rpmsg_device rpdev; /* RPMsg 通道 */
  struct virtproc_info *vrp; /* 此通道所属的远程处理器 */
};
```

4. virtproc_info

每个物理远程处理器可能有多个 virtio proc 设备，virtproc_info 结构体用于指定 virtio 远程处理器设备的 RPMsg 状态，virtproc_info 在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下定义：

```
struct virtproc_info {                  /* 远程处理器状态 */
  struct virtio_device *vdev;           /* virtio 设备 */
  struct virtqueue *rvq, *svq;          /* rx 和 tx 的 virtqueue */
  void *rbufs, *sbufs;                  /* rx 和 tx 缓冲区的内核地址 */
  unsigned int num_bufs;                /* rx 和 tx 的缓冲区总数 */
  unsigned int buf_size;                /* 一个 rx 或 tx 缓冲区的大小 */
  int last_sbuf;                        /* 上次使用的 tx 缓冲区的索引 */
  dma_addr_t bufs_dma;                  /* 缓冲区的 dma 基地址 */
  struct mutex tx_lock;                 /* 保护 svq、sbufs 和 sleepers，以允许并发发送者 */
  struct idr endpoints;                 /* 本地端点的 idr，允许快速检索 */
  struct mutex endpoints_lock;          /* 端点集的锁 */
  wait_queue_head_t sendq;              /* 发送上下文的等待队列等待 tx 缓冲区 */
  atomic_t sleepers;                    /* 等待 tx 缓冲区的发送者数量 */
  struct rpmsg_endpoint *ns_ept;/* 总线的名称服务端点 */
};
```

5. rpmsg_channel_info

在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 中找到 rpmsg_channel_info 结构体，rpmsg_channel_info 是通道信息，如下：

```
struct rpmsg_channel_info {
        char name[RPMSG_NAME_SIZE];  /* 远程服务名字 */
        u32 src;                     /* 本地地址 */
        u32 dst;                     /* 目的地址 */
};
```

6. rpmsg_ns_msg

在 Linux 内核源码 drivers/rpmsg/ virtio_rpmsg_bus.c 下有如下定义：

```
struct rpmsg_ns_msg {               /* 动态服务公告名称消息 */
    char name[RPMSG_NAME_SIZE];     /* 已发布的远程服务的名称 */
    u32 addr;                       /* 已发布的远程服务的地址 */
    u32 flags;                      /* 用于指示是创建服务还是销毁服务 */
} __packed;

/* flags 的取值枚举类型 */
enum rpmsg_ns_flags {             /* 动态名称服务公告标志取值 */
    RPMSG_NS_CREATE     = 0,      /* 刚刚创建了一个新的远程服务 */
    RPMSG_NS_DESTROY    = 1,      /* 一个已知的远程服务刚刚被销毁 */
};
```

rpmsg_ns_msg 是名称服务公告消息结构体，其中 name 就是和端点关联的服务的名字（service name），addr 是已经发布服务公告名称的远程处理器的地址，flags 标志位用于指示是创建服务还是销毁服务。

5.1.2 缓冲区

在 Linux 内核下，驱动为每个通信分配 512 字节的缓冲区，且每个缓冲区将 16 个字节用作消息头，所以每次最大只能发送 496 字节的数据，即，消息数据的有效负载大小为 496 字节。缓冲区的数量是根据 vring 支持的缓冲区数量来计算的，最多 512 个缓冲区（每个方向 256 个）。如下所示是 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的定义：

```
#define MAX_RPMSG_NUM_BUFS   (512) /* 最多 512 个缓冲区（收和发方向各 256 个） */
#define MAX_RPMSG_BUF_SIZE   (512) /* 缓冲区大小为 512 字节，16 个字节用于消息头 */
```

在 Linux 内核源码的 drivers/rpmsg/virtio_rpmsg_bus.c 文件中定义了 rpmsg_hdr 结构体，此结构体用于表示在 RPMsg 通道上发送/接收的每条消息的格式，每条消息都以此结构体为开头。

```
struct rpmsg_hdr {/* 所有 RPMsg 消息的公共标头 */
  u32 src;          /* 源地址 */
  u32 dst;          /* 目的地址 */
  u32 reserved;     /* 保留字段 */
  u16 len;          /* 消息有效负载的长度（以字节为单位）*/
  u16 flags;       /* 消息标志 */
  u8 data[0];      /* 长度为字节的消息有效负载数据 */
} __packed;
```

根据以上定义，RPMsg 中的消息格式如下图 5.1.2.1 所示，前面是消息头，共占用了 16 个字节，由 RPMsg 内部使用，消息头组成是：第一个字（32 位）用作发送方或源端点的地址，下一个字是接收方或目标端点的地址，第三个字是保留字段，最后一个字是有效负载的长度（16 位）和一个(16 位)标志字段。紧跟消息头后面的才是用户消息的有效负载，即用户发送的有效消息数据，以字节为单位。

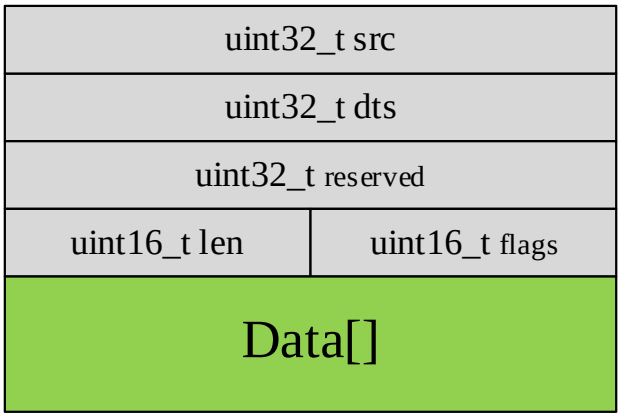


图 5.1.2.1 RPMsg 中的消息格式

5.1.3 创建 RPMsg 通道 API 函数

RPMsg 通道（RPMsg channel）是主处理器与远程处理器(也称为 RPMsg 设备)之间的双向通信通道，所有的消息都在 RPMsg 上传递，消息包括消息头和用户实际要发送的数据。RPMsg 通道创建的过程大概如下：

1）一旦主处理器通过 Remoteproc 加载和启动协处理器的固件后，协处理器就会发送服务公告名称消息；

2）主处理器收到这个消息后，会根据消息里的服务公告名称（或者称为服务的名称）来创建一个 RPMsg 通道，通道的名字和服务公告名称是一样的；

3）接下来，主处理器必须先给协处理器发送第一条消息（可以是任何的消息）后，通道“才算真正激活”；

4）通道“激活”以后，协处理器才可以通过此 RPMsg 通道给主处理器发送消息，用户代码可以通过调用 RPMsg 相关的 API 函数来实现消息的发送操作，例如，可以使用 rpmsg_send()函数发送消息，关于此函数我们后面会进行介绍。

RPMsg 是基于 Virtio 的，Virtio 有两个 vring，分别用于发送和接收消息，还有一个 Vring buffers，Vring buffers 就是共享的内存，所以 RPMsg 框架，其本质上也是通过共享内存来实现核间通信。

下图 5.1.3.1 是主处理器和协处理器通信的过程，对于 M4，用户调用 OpenAMP 库中的 OPENAMP_create_endpoint()函数创建端点时，会指定服务的名称， A7 接收到服务公告名称消息后，根据消息中的服务名称，通过调用 Linux 内核下的 rpmsg_ns_cb()函数来创建和协处理器通信的 RPMsg 通道设备，此 RPMsg 通道设备的名字和服务的名称一样（例如，服务名称是 rpmsg-client-sample，那么此 RPMsg 通道设备的名称也是 rpmsg-client-sample）。

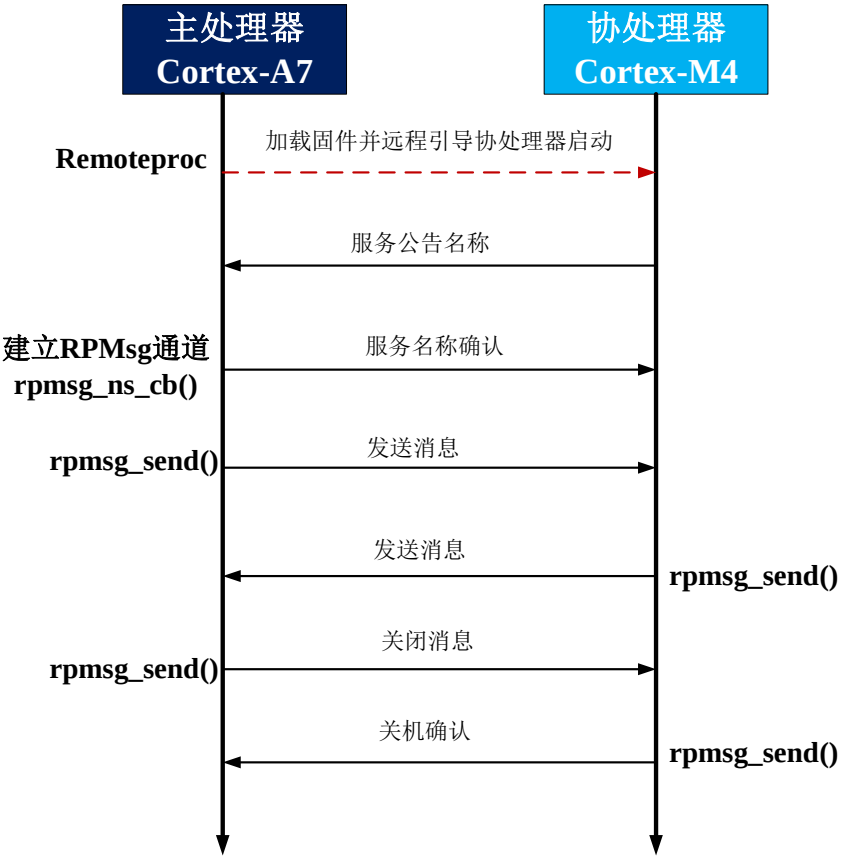


图 5.1.3.1 主处理器和远程处理器通信过程

Linux 内核会根据接收到的服务公告消息内容来创建或者销毁一个 RPMsg 通道，如果创建此通道，则通过此通道来和远程处理器进行通信，此通道的名字和服务名称一样。在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下找到如下函数：

```
/**
 * @brief       在服务公告名称消息到达时，此函数被调用
 * @param       rpdev：RPMsg 通道设备
 *              data：接收的服务公告名称消息
```

```
 *              len:  接收的服务公告名称消息长度
 *              priv: RPMsg 驱动私有数据
 *              src: 接收到消息的远程源地址
 * @retval     返回 0: 成功; 其它值: 失败
 */
static int rpmsg_ns_cb(struct rpmsg_device *rpdev, void *data, int len,
                       void *priv, u32 src)
{
  struct rpmsg_ns_msg *msg = data;
  struct rpmsg_device *newch;
  struct rpmsg_channel_info chinfo;
  struct virtproc_info *vrp = priv;
  struct device *dev = &vrp->vdev->dev;
  int ret;
/* Linux 内核动态调试功能, 以十六进制打印 */
#if defined(CONFIG_DYNAMIC_DEBUG)
  dynamic_hex_dump("NS announcement:",DUMP_PREFIX_NONE,16,1,data,len, true);
#endif
  /* 检查接收到的消息的长度 */
  if (len != sizeof(*msg)) {
        dev_err(dev, "malformed ns msg (%d)\n", len);
        return -EINVAL;
  }
   /* 名称服务 ept 不属于真正的 RPMsg 通道, 由 RPMsg 总线本身处理 */
  if (rpdev) {
        dev_err(dev, "anomaly: ns ept has an rpdev handle\n");
        return -EINVAL;
  }
  msg->name[RPMSG_NAME_SIZE - 1] = '\0';
  /* Linux 内核打印通道建立或者销毁信息 */
  dev_info(dev, "%sing channel %s addr 0x%x\n",
         msg->flags & RPMSG_NS_DESTROY ? "destroy" : "creat",
         msg->name, msg->addr);
  /* 获取服务的名字  */
  strncpy(chinfo.name, msg->name, sizeof(chinfo.name));
  /* 基于服务名字设置通道源地址, 系统自动分配一个地址 */
  chinfo.src = RPMSG_ADDR_ANY;
  chinfo.dst = msg->addr;   /* 根据已发布的远程服务的地址设置目的地址  */

  if (msg->flags & RPMSG_NS_DESTROY) { /* 销毁一个已有的通道 */
        ret = rpmsg_unregister_device(&vrp->vdev->dev, &chinfo);
        if (ret)
              dev_err(dev, "rpmsg_destroy_channel failed: %d\n", ret);
```

```
  } else {/* 创建一个新的通道，通道名字是服务的名字 */
        newch = rpmsg_create_channel(vrp, &chinfo);
        if (!newch)
              dev_err(dev, "rpmsg_create_channel failed\n");
  }

  return 0;
}
```

rpmsg_ns_cb()函数会根据接收到的服务公告名称消息来创建和远程处理器通信的 RPMsg 通道，如果服务公告名称中的消息 flags 是 RPMSG_NS_CREATE，则表示新建一个 RPMsg 通道，如果 flags 为 RPMSG_NS_DESTROY，则删除 RPMsg 通道。

通过调用 rpmsg_create_channel()函数来创建一个名字为服务名称的通道，此服务名称由 M4 这边决定。

5.1.4 创建 RPMsg 端点 API 函数

主处理器和远程处理器是通过 RPMsg 通道来发送消息的，当主处理器和远程处理器建立起 RPMsg 通道后，基于这个通道，用户可以在主处理器或者远程处理器端创建一个或者多个 RPMsg 端点（RPMsg endpoint），即一个通道可以有多个 RPMsg 端点，RPMsg 端点是可出现在 RPMsg 通道任意一侧的逻辑抽象，RPMsg 端点是主处理器与远程处理器之间发送消息的基础架构。

值得注意的是，在没有手动创建端点前，每个通道其实都有一个默认的端点，即使应用程序没有创建新的 RPMsg 端点，也可以使用默认的端点来进行核间通信。每个 RPMsg 端点都有一个唯一的本地（src）地址和关联的接收回调函数，接收回调函数由用户定义，当接收到针对给定端点索引的消息时，和这个端点关联的接收回调函数就会被执行。而那些没有明确指定目标端点索引的消息就会到达与 RPMsg 通道相关联的默认端点。

如下图是 RPMsg 端点、RPMsg 通道和主/远程处理器联系示意图，主处理器上的一个 RPMsg 端点的本地地址 src=1，要发送消息的远程处理器的目标端点的地址 dts=2。在远程处理器（也叫协处理器）这边的另外一个端点，其本地地址 src=2，目的地址 dts=1。

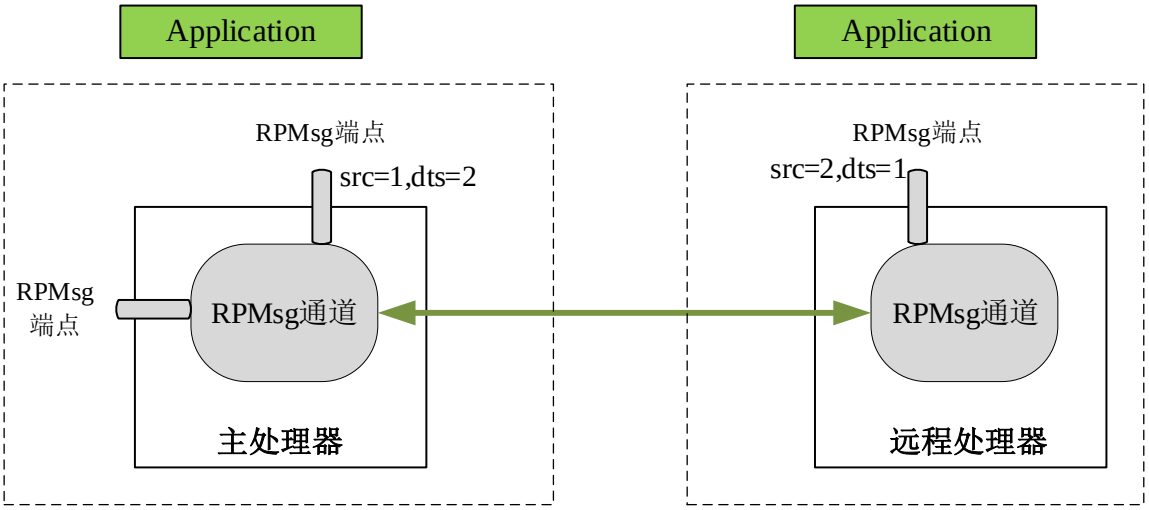


图 5.1.4.1 RPMsg 端点示意图

以上结构图说明：

一个 RPMsg 通道可以创建多个 RPMsg 端点；

通道依靠一个或多个 RPMsg 端点实现服务；

RPMsg 端点通过通道提供逻辑连接。

共享的内存和核间中断，我们可以称为物理链路，或者称为通信链路，一条物理链路可以提供多个通信通道，如下图 5.1.4.2 所示。

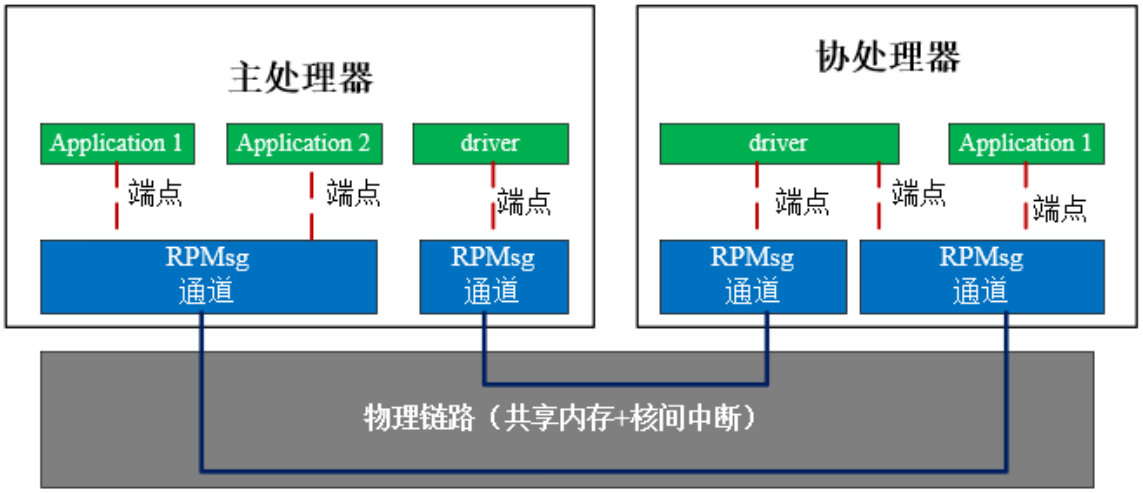


图 5.1.4.2 基于共享内存实现核间通信

可以通过调用 rpmsg_create_ept()函数来创建 RPMsg 端点，此函数在 Linux 内核源码的 drivers/rpmsg/rpmsg_core.c 下有定义，实际上是通过调用 virtio_rpmsg_create_ept()函数完成端点的创建。默认情况下 RPMsg 通道是已经有一个默认的端点了，用户无需创建就可以使用，如需要，用户可通过此函数来创建其它端点。

```
/**
 * @brief       创建新的 RPMsg 端点
 * @param       rpdev：RPMsg 通道设备
 *              cb：rx 回调处理程序
 *              priv：私有数据
 *              chinfo：与 cb 绑定的通道信息
 * @retval      成功时返回指向端点的指针，错误时返回 NULL
 */
struct rpmsg_endpoint *rpmsg_create_ept(struct rpmsg_device *rpdev,
                                        rpmsg_rx_cb_t cb, void *priv,
                                        struct rpmsg_channel_info chinfo)
{
  if (WARN_ON(!rpdev))
        return NULL;
  /* 创建特定于后端的端点（必须） */
  return rpdev->ops->create_ept(rpdev, cb, priv, chinfo);
}
EXPORT_SYMBOL(rpmsg_create_ept);
```

5.1.5 发送消息 API 函数

RPMsg 的发送 API 说明如下表 5.1.5.1 所示：

API	说明
rpmsg_send_offchannel_raw()	wait=0 表示非阻塞，wait=1 表示阻塞
rpmsg_send()	阻塞方式发送（wait=1）
rpmsg_sendto()	
rpmsg_send_offchannel()	
rpmsg_trysend()	非阻塞方式发送（wait=0）
rpmsg_trysendto()	
rpmsg_trysend_offchannel()	

表 5.1.5.1 RPMsg 相关的发送 API 汇总

以上的 API 函数，大多数是通过调用 rpmsg_send_offchannel_raw()函数来完成的，下面，我们就以 rpmsg_send_offchannel_raw()函数和 rpmsg_send()函数为例子进行分析，其它的函数分析方法也类似。

1. 函数 rpmsg_send_offchannel_raw()

rpmsg_send_offchannel_raw()函数在 drivers/rpmsg/virtio_rpmsg_bus.c 中定义，该函数使用用户提供的 src（源地址） 和 dst （目标地址）地址向远程处理器发送消息。调用者应该指定通道、它想要发送的数据、数据的长度（以字节为单位）以及明确的源地址和目标地址，然后将消息发送到通道所属的远程处理器。很多 API 函数是通过调用该函数来完成消息发送的。

当参数 wait 为 1 时表示等待，即调用者被阻塞，直到有可用的 TX 缓冲区或者 15s 的等待时间超时后，如果还没等到 TX 缓冲区，则返回-ERESTARTSYS，当 wait 为 0 时，表示非阻塞，当调用该函数时，只要没有可用的 TX 缓冲区，该函数将立即失败，并返回-ENOMEM。

```
1  /**
2    * @brief       向远程处理器发送消息
3    * @param       rpdev: RPMsg 通道
4    *              src  : 源地址
5    *              dst  : 目的地址
6    *              data : 有效的消息数据
7    *              len  : 有效的消息长度
8    *              wait : 指示在没有可用的 TX 缓冲区的情况下调用者是否应该阻塞
9    * @retval      成功返回 0，否则返回适当的错误代码。
10   */
11 static int rpmsg_send_offchannel_raw(struct rpmsg_device *rpdev,
12                                       u32 src, u32 dst,
13                                       void *data, int len, bool wait)
14 {
15     /* 获取指定的 RPMsg 通道 */
16    struct virtio_rpmsg_channel *vch = to_virtio_rpmsg_channel(rpdev);
17     /* 获取指定的 RPMsg 通道上的处理器 */
18     struct virtproc_info *vrp = vch->vrp;
19     struct device *dev = &rpdev->dev;  /*获取设备 */
20     struct scatterlist sg; /* 要填充的散点列表 */
```

```
21      struct rpmsg_hdr *msg; /* 在 RPMsg 上发送/接收的消息的格式 */
22      int err;               /* 返回值 */
23
24      /* 当源地址或者目的地址为 RPMSG_ADDR_ANY 时，不允许广播 */
25      if (src == RPMSG_ADDR_ANY || dst == RPMSG_ADDR_ANY) {
26        dev_err(dev, "invalid addr (src 0x%x, dst 0x%x)\n", src, dst);
27        return -EINVAL;
28      }
29
30      /* 检查有效的消息的长度是否大于 rx 或 tx 缓冲区的大小 */
31      if (len > vrp->buf_size - sizeof(struct rpmsg_hdr)) {
32            dev_err(dev, "message is too big (%d)\n", len);
33            return -EMSGSIZE;
34      }
35
36      /* 分配一个 tx 缓冲区*/
37      msg = get_a_tx_buf(vrp);
38      if (!msg && !wait)
39            return -ENOMEM;
40
41    /* 没有空闲缓冲区？则等待一个（但在 15 秒后释放） */
42      while (!msg)
43      {
44        /* 如果需要，启用“tx-complete”中断 */
45        rpmsg_upref_sleepers(vrp);
46
47        /* 休眠，直到有空闲缓冲区可用或 15 秒超时 */
48        err = wait_event_interruptible_timeout(vrp->sendq,
49                        (msg = get_a_tx_buf(vrp)),
50                        msecs_to_jiffies(15000));
51
52        /* 如果是最后一个休眠，则禁用“tx-complete”中断 */
53        rpmsg_downref_sleepers(vrp);
54
55        /*检查是否超时 */
56        if (!err)
57        {
58          dev_err(dev, "timeout waiting for a tx buffer\n");
59          return -ERESTARTSYS;
60        }
61      }
62
63      msg->len = len;     /* 有效消息的长度（以字节为单位）*/
```

```
64      msg->flags = 0;     /* 消息标志 */
65      msg->src = src;     /* 消息的源地址 */
66      msg->dst = dst;     /* 消息的目的地址 */
67      msg->reserved = 0;/* 保留 */
68      memcpy(msg->data, data, len);/* 将长度为 len 的消息拷贝到 data 数组中 */
69     dev_dbg(dev, "TX From 0x%x,To 0x%x,Len %d,Flags %d,Reserved %d\n",
70              msg->src, msg->dst, msg->len, msg->flags, msg->reserved);
71
72     #if defined(CONFIG_DYNAMIC_DEBUG)
73      /* 使用 dynamic_hex_dump 进行十六进制转储跟踪 */
74      dynamic_hex_dump("rpmsg_virtio TX: ", DUMP_PREFIX_NONE, 16, 1,
75                       msg, sizeof(*msg) + msg->len, true);
76      #endif
77      /* 根据 cpu 地址位置初始化 scatterlist */
78      rpmsg_sg_init(&sg, msg, sizeof(*msg) + len);
79      /* 加锁 */
80      mutex_lock(&vrp->tx_lock);
81
82      /* 将消息添加到远程处理器的虚拟队列 */
83      err = virtqueue_add_outbuf(vrp->svq, &sg, 1, msg, GFP_KERNEL);
84      if (err)
85        {
86  /* 这里需要回收缓冲区，否则会丢失（内存不会泄漏，RPMsg 不会再次将其用于 TX） */
87          dev_err(dev, "virtqueue_add_outbuf failed: %d\n", err);
88          goto out;
89        }
90
91      /* 告诉远程处理器它有一个待读取的消息 */
92      virtqueue_kick(vrp->svq);
93    out:
94      /* 解锁 */
95      mutex_unlock(&vrp->tx_lock);
96      return err;
97  }
```

第 16~18 行，获取指定 RPMsg 通道上的远程处理器；

第 25~28 行，判断源地址和目的地址是否是 RPMSG_ADDR_ANY，如果是的话，则提示无效地址。RPMSG_ADDR_ANY 在 include/linux/rpmsg.h 文件中定义为 0xFFFFFFFF，这里是判断定义的 src 和 dts 是否有效，即消息将在指定的通道上发送，其源地址和目标地址字段将设置为通道关联的端点的 src 和 dst 地址；

第 31~34 行，判断消息的长度是否大于 TX 或 RX 缓冲区的大小，如果大于，则返回报错值；

第 37~39 行，选择下一个未使用的 TX 缓冲区，或者回收一个用过的 TX 缓冲区。

第 42~61 行，等待一个空闲的 TX 缓冲区，如果没有可用的 TX 缓冲区，该函数将阻塞，直到有一个可用的 TX 缓冲区（即，直到远程处理器消耗 TX 缓冲区并将其放回 virtio 使用的描述符上），或者经过 15 秒的超时，当超时发生时，返回-ERESTARTSYS。

第 63~68，如果等到了一个空闲的 TX 缓冲区，则指定消息的长度、源地址、目的地址和标志为 0，然后将消息拷贝到 data 数组；

第 72~76 行，宏 CONFIG_DYNAMIC_DEBUG 用于启动动态 printk()支持，这是一种调试手段，如果有定义此宏，则使用 dynamic_hex_dump 进行十六进制转储跟踪调试；

第 78 行，根据 CPU 地址位置初始化 scatterlist，sg 是要填充的散点列表，cpu_addr 是缓冲区的虚拟地址，len 是缓冲区长度；

第 80~95 行，将消息添加到远程处理器的虚拟队列，然后告诉远程处理器它有一个待读取的消息。

以上就是将消息数据以消息队列的方式发给 RPMsg 通道上的处理器的过程。

2. 函数 rpmsg_send()

rpmsg_send()函数在 drivers/rpmsg/rpmsg_core.c 文件中定义，此函数在 ept 端点上发送长度为 len 的 data，消息将使用 ept 端点的地址及其关联的 RPMsg 通道的目标地址发送到 ept 端点所属的远程处理器。调用者应指定端点、要发送的数据及其长度（以字节为单位）。如果没有可用的 TX 缓冲区，同样的，该函数会产生阻塞。

```
1   /**
2     * @brief       向远程处理器发送消息
3     * @param       ept: RPMsg 端点
4     *              data: 消息的有效载荷
5     *              len:  有效载荷长度
6     * @retval      成功返回 0，否则返回适当的错误代码。
7     */
8   int rpmsg_send(struct rpmsg_endpoint *ept, void *data, int len)
9   {
10    if (WARN_ON(!ept))
11          return -EINVAL;
12    if (!ept->ops->send)
13          return -ENXIO;
14
15    return ept->ops->send(ept, data, len);
16  }
17  EXPORT_SYMBOL(rpmsg_send);
```

第 10 行，WARN_ON 相当于内核运行时的断言，当括号中的条件成立时，内核会抛出一个栈回溯，常用于调试，也就是判断 ept（RPMsg 端点）是否存在。

第 12 行，如果 RPMsg 端点存在的话，则调用 rpmsg_endpoint_ops 中的 rpmsg_send 处理程序，即最终调用 virtio_rpmsg_send()回调函数完成发送功能。下面是 virtio_rpmsg_send 回调函数：

```
static int virtio_rpmsg_send(struct rpmsg_endpoint *ept,
                             void *data, int len)
{
  struct rpmsg_device *rpdev = ept->rpdev;
  u32 src = ept->addr, dst = rpdev->dst;
```

```
  return rpmsg_send_offchannel_raw(rpdev, src, dst, data, len, true);
}
```

可以看到，virtio_rpmsg_send()回调函数是通过调用前面的 rpmsg_send_offchannel_raw()函数来完成发送功能的，如果没有可用的 TX 缓冲区，该函数将阻塞直到远程处理器消耗 TX 缓冲区并将其释放，或者经过 15 秒的超时，当超时发生时，返回-ERESTARTSYS。

3. 其它发送函数

前面 virtio_endpoint_ops 里指定的回调函数大多数都会调用 rpmsg_send_offchannel_raw()函数来完成数据发送功能，这些函数在 drivers/rpmsg/rpmsg_core.c 中定义，函数的实现过程和 rpmsg_send()类似，我们就不再一一进行分析了，下面我们直接将这些函数列出来，其中 rpmsg_send_offchannel_raw()函数的形参如下表 5.1.5.2 所示：

函数	描述
ept	RPMsg 端点
src	源地址
dst	目的地址
data	发送的数据
len	数据长度
wait	wait 为 0 时，表示非阻塞；为 1 是表示阻塞

表 5.1.5.2 参数说明

1. rpmsg_sendto()

rpmsg_sendto()函数是以 ept 端点的地址作为源地址，将长度为 len 的 data 发送到远程 dts 地址中，消息将被发送到 ept 端点所属的远程处理器。调用者应指定端点、要发送的数据及其长度（以字节为单位），还有目标地址 dts。如果没有可用的 TX 缓冲区，该函数会产生阻塞。

```
int rpmsg_sendto(struct rpmsg_endpoint *ept, void *data, int len, u32 dst)
```

2. rpmsg_send_offchannel()

rpmsg_send_offchannel()函数使用 src 作为源地址，将长度为 len（以字节为单位）的数据 data 发送到远程 dst 地址，消息将被发送到 ept 端点所属的远程处理器。调用者应指定端点、要发送的数据及其长度（以字节为单位），还有源地址 src 和目标地址 dts。如果没有可用的 TX 缓冲区，该函数会产生阻塞。

```
int rpmsg_send_offchannel(struct rpmsg_endpoint *ept, u32 src, u32 dst,
                          void *data, int len)
```

3. rpmsg_trysend()

rpmsg_trysend()函数在 ept 端点上发送长度为 len（以字节为单位）的 data，消息将使用 ept 端点的地址作为源地址，其关联的 RPMsg 通道为目标地址，消息将被发送到 ept 端点所属的远程处理器。

```
int rpmsg_trysend(struct rpmsg_endpoint *ept, void *data, int len)
```

4. rpmsg_trysendto()

rpmsg_sendto()函数是以 ept 端点的地址作为源地址，将长度为 len 的 data 发送到远程 dts 地址中，消息将被发送到 ept 端点所属的远程处理器。调用者应指定端点、要发送的数据及其长度（以字节为单位），还有目标地址 dts。

```
int rpmsg_trysendto(struct rpmsg_endpoint *ept, void *data, int len, u32 dst)
```

5. rpmsg_trysend_offchannel()

rpmsg_trysend_offchannel()函数使用 src 作为源地址，将长度为 len（以字节为单位）的数据 data 发送到远程 dst 地址，消息将被发送到 ept 端点所属的远程处理器。调用者应指定端点、要发送的数据及其长度（以字节为单位），还有源地址 src 和目标地址 dts。

```
int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept, u32 src, u32 dst,
                             void *data, int len)
```

5.2 OpenAMP 库中的 API

5.2.1 初始化 IPCC API

MX_IPCC_Init()函数用于初始化 IPCC，函数定义如下：

```
/**
 * @brief       初始化 IPCC
 * @param       无
* @retval      无
 */
static void MX_IPCC_Init(void)
{
  hipcc.Instance = IPCC;
  if (HAL_IPCC_Init(&hipcc) != HAL_OK)
  {
    Error_Handler();
  }
}
```

HAL_IPCC_Init()会初始化 IPCC 的时钟、通道和中断配置等。

5.2.2 初始化 OpenAMP API

MX_OPENAMP_Init()函数用于初始化 OpenAMP，包括邮箱、共享内存、资源表、Virtio、Vring 和 vring Buffers，函数如下所示，值得注意的是，此函数会间接调用 malloc()函数来分配内存，所以，当 M4 还未完成内存初始化时，不能调用 MX_OPENAMP_Init()函数，否则测试会异常，尤其是在 M4 跑 RTOS 时，当在编程的时候一定要注意，最好在进入主函数的以后再去调用此函数。

```
/**
 * @brief   OpenAMP 初始化
 * @param   RPMsgRole：只能给 0 或者 1,0 表示做主机，1 表示做从机。M4 只能给 1 做从机
 *          ns_bind_cb：用于服务公告名称的回调处理程序，通常直接给 NULL
 * @retval  负数，失败；0，成功
 */
int MX_OPENAMP_Init(int RPMsgRole, rpmsg_ns_bind_cb ns_bind_cb)
{
  struct fw_rsc_vdev_vring *vring_rsc = NULL;
  struct virtio_device *vdev = NULL;
  int status = 0;
```

```
/* 初始化邮箱 */
MAILBOX_Init();
/* 初始化公共内存 */
status = OPENAMP_shmem_init(RPMsgRole);
if(status)
{
  return status;
}
/* 初始化 Virtio 环境 */
vdev = rproc_virtio_create_vdev(RPMsgRole, VDEV_ID, &rsc_table->vdev,
                                rsc_io, NULL, MAILBOX_Notify, NULL);
if (vdev == NULL)
{
  return -1;
}
/* 等待 Virtio 环境就绪 */
rproc_virtio_wait_remote_ready(vdev);

/* 初始化 vring0 */
vring_rsc = &rsc_table->vring0;
status = rproc_virtio_init_vring(vdev, 0, vring_rsc->notifyid,
                            (void *)vring_rsc->da, shm_io,
                            vring_rsc->num, vring_rsc->align);
if (status != 0)
{
  return status;
}
 /* 初始化 vring1 */
vring_rsc = &rsc_table->vring1;
status = rproc_virtio_init_vring(vdev, 1, vring_rsc->notifyid,
                                (void *)vring_rsc->da, shm_io,
                                 vring_rsc->num, vring_rsc->align);
if (status != 0)
{
  return status;
}
/* 主核初始化共享缓冲池 */
rpmsg_virtio_init_shm_pool(&shpool, (void *)VRING_BUFF_ADDRESS,
                           (size_t)SHM_SIZE);
/* 初始化 RPMsg virtio 设备 */
rpmsg_init_vdev(&rvdev, vdev, ns_bind_cb, shm_io, &shpool);

return 0;
```

```
}
```

5.2.3 回调函数

OpenAMP 库中的回调函数有固定的形式，如下：

```
static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data, size_t len,
                       uint32_t src, void *priv);
```

函数参数和返回值含义如下所示：

参数	说明
rp_chnl	rpmsg_endpoint 类型的结构体，即指定端点
data	存放 A7 发过来的数据的 Buffer
len	data 数据的长度
src	源地址
priv	私有数据

表 5.2.3.1 rx_callback()函数参数说明

返回值：负数，失败；0，成功。

5.2.4 创建 RPMsg 端点 API

OPENAMP_create_endpoint()函数用于创建 RPMsg 端点，使用指定的服务名称、源地址、目的地址、端点关联的接收回调函数对其进行初始化，注意的是，服务名称要和 Linux 端的驱动名称一样，因为 Linux 下的 rpmsg_ns_cb()函数会创建名字为服务名称的 RPMsg 通道设备，通过此名字和 Linux 下平台的驱动相匹配，驱动才可以加载，关于这点我们会在后面实验进行讲解。

这里注意的是，此处默认将目的地址设置为 RPMSG_ADDR_ANY（为 0xFFFFFFFF），需要主处理器发送第一条消息（可以是任意的消息，可以称为查询消息），根据服务名称来查询对方（指协处理器）的地址，这有点类似于网络协议中的域名解析过程，根据域名解析出 IP 地址。注意的是，此函数没有设置源地址，在 rpmsg_create_ept()函数中会调用 rpmsg_get_address()函数从地址位映射中取出一个没有使用的地址来用。

```
/**
 * @brief     创建 RPMsg 端点
 * @param     ept：RPMsg 端点
 *             name：服务名称
 *             dest：目的地址
 *           cb：和端点绑定的 rx 回调函数
 *           unbind_cb:ns_bind_cb:用于名称服务公告的回调处理程序,通常直接给 NULL
 * @retval    负数，失败；0，成功
 */
int OPENAMP_create_endpoint(struct rpmsg_endpoint *ept, const char *name,
                            uint32_t dest, rpmsg_ept_cb cb,
                            rpmsg_ns_unbind_cb unbind_cb)
{
  int ret = 0;
```

```
  ret = rpmsg_create_ept(ept, &rvdev.rdev, name, RPMSG_ADDR_ANY, dest, cb,
                         unbind_cb);

  return ret;
}
```

上面提到协处理器创建 RPMsg 端点时，端点对应目的地址 dest 设置为 RPMSG_ADDR_ANY，该地址并不是主处理器的地址，如果协处理器调用 RPMsg 发送函数来发送消息，是发不出去的，这个时候需要主处理器先给协处理器发送第一条消息，协处理器收到这第一条消息后，会在初始化 RPMsg 时更新 RPMsg 通道的目的地址，具体可查看 OpenAMP 库下 rpmsg_virtio.c 文件中的 rpmsg_init_vdev()函数，该函数会调用 rpmsg_virtio_rx_callback()函数，如下红色部分的代码，这就是为什么在 5.1.3 小节时提到要主处理器发送第一条消息。

```
/**
 * rpmsg_virtio_rx_callback
 * Rx 回调函数。
 * @param vq - 指向接收消息的 virtqueue 的指针
 */
static void rpmsg_virtio_rx_callback(struct virtqueue *vq)
{
  struct virtio_device *vdev = vq->vq_dev;
  struct rpmsg_virtio_device *rvdev = vdev->priv;
  struct rpmsg_device *rdev = &rvdev->rdev;
  struct rpmsg_endpoint *ept;
  struct rpmsg_hdr *rp_hdr;
  unsigned long len;
  unsigned short idx;
  int status;

  metal_mutex_acquire(&rdev->lock);

  /* 处理从远程节点接收的数据 */
  rp_hdr = (struct rpmsg_hdr *)rpmsg_virtio_get_rx_buffer(rvdev,
              &len, &idx);

  metal_mutex_release(&rdev->lock);

  while (rp_hdr) {
    /* 从远程设备通道列表中获取通道节点 */
    metal_mutex_acquire(&rdev->lock);
    ept = rpmsg_get_ept_from_addr(rdev, rp_hdr->dst);
    metal_mutex_release(&rdev->lock);

    if (!ept)
      /* 致命错误给定 dst 地址没有端点 */
```

```
      return;

    if (ept->dest_addr == RPMSG_ADDR_ANY) {
      /*
       * 从远端收到的第一条信息,
       * 更新通道目标地址
       */
      ept->dest_addr = rp_hdr->src;
    }
    status = ept->cb(ept, (void *)RPMSG_LOCATE_DATA(rp_hdr),
          rp_hdr->len, ept->addr, ept->priv);

    RPMSG_ASSERT(status == RPMSG_SUCCESS,
          "unexpected callback status\n");
    metal_mutex_acquire(&rdev->lock);

    /* 返回使用过的缓冲区 */
    rpmsg_virtio_return_buffer(rvdev, rp_hdr, len, idx);

    rp_hdr = (struct rpmsg_hdr *)
       rpmsg_virtio_get_rx_buffer(rvdev, &len, &idx);
    metal_mutex_release(&rdev->lock);
  }
}
```

5.2.5 轮询 API

函数 OPENAMP_check_for_message()通过邮箱轮询来检查 Vring Buffer 中是否有数据，即检查 M4 是否有收到 A7 发来的数据（包括 A7 发来的第一条数据）。

```
/**
 * @brief       通过邮箱查询是否接收到数据
 * @param       无
 * @retval      无
 */
void OPENAMP_check_for_message(void)
{
  /* 查询 vring0 和 vring1 */
  MAILBOX_Poll(rvdev.vdev);
}
```

5.2.6 发送消息 API

和 Linux 下的 RPMsg 发送消息相关的 API 函数一样，在 OpenAMP 下的 API 函数如表 5.2.6.1 所示：

函数	说明
rpmsg_send_offchannel_raw()	wait=0 表示非阻塞，wait=1 表示阻塞
rpmsg_send()	阻塞方式发送（wait=1）
rpmsg_sendto()	
rpmsg_send_offchannel()	
rpmsg_trysend()	非阻塞方式发送（wait=0）
rpmsg_trysendto()	
rpmsg_trysend_offchannel()	

表 5.2.6.1 RPMsg 相关的发送 API 汇总

在前面我们已经分析了 Linux 下相关的 RPMsg 发送相关的 API 函数，下面我们就来简单了解一下在 OpenAMP 下的这些函数。在 openamp.h 中有如下宏定义，即调用了 OPENAMP_send ()函数的话，相当于调用了 rpmsg_send()函数：

```
#define OPENAMP_send  rpmsg_send
```

rpmsg_send()函数在 rpmsg.h 文件中可以找到，如下：

```
/**
 * @brief     基于 ept 端点发送长度为 len 的 data
 * @param     ept：RPMsg 端点
 *            data：发送的数据
 *            len：  数据长度
 * @retval     负数，失败；0，成功
 */
static inline int rpmsg_send(struct rpmsg_endpoint *ept,
                             const void *data,int len)
{
  if (ept->dest_addr == RPMSG_ADDR_ANY)
    return RPMSG_ERR_ADDR;
  return rpmsg_send_offchannel_raw(ept, ept->addr, ept->dest_addr,
                                   data,len, true);
}
```

rpmsg_send()函数是基于 ept 端点发送长度为 len 的 data，消息将使用 ept 端点的源地址和目标地址发送到通道所属的主处理器，如果没有可用的 TX 缓冲区，该函数会产生阻塞。实际上是调用 rpmsg_send_offchannel_raw()函数来完成发送功能的。

如果调用了 OPENAMP_send()函数，则表示执行了 rpmsg_send()函数来向主处理器发送数据。

其它发送函数和 Linux 下的类似，也都是通过调用 rpmsg_send_offchannel_raw()函数来完成发送功能的，其中函数中的形参如下表 5.2.6.2 所示：

函数	描述
ept	RPMsg 端点
src	源地址
dst	目的地址
data	发送的数据
size	数据长度
len	数据长度
wait	wait 为 0 时，表示非阻塞；为 1 是表示阻塞

表 5.2.6.2 参数说明

1. rpmsg_send_offchannel_raw()

此函数从源 src 地址向远程 dst 地址发送长度为 size 的 data。消息将被发送到通道所属的远程处理器。

```
int rpmsg_send_offchannel_raw(struct rpmsg_endpoint *ept,
                              uint32_t src,uint32_t dst,
                              const void *data, int size,
                              int wait);
```

2. rpmsg_sendto()

此函数将长度为 len 的 data 发送到远程 dst 地址。消息将使用 ept 端点的源地址发送到 ept 通道所属的远程处理器。如果没有可用的 TX 缓冲区，该函数会产生阻塞。

```
static inline int rpmsg_sendto(struct rpmsg_endpoint *ept,
                               const void *data,int len, uint32_t dst)
```

3. rpmsg_send_offchannel()

此函数将 len 长度的 data 发送到远程 dst 地址，并使用 src 作为源地址。该消息将被发送到 ept 端点对应通道所属的远程处理器。如果没有可用的 TX 缓冲区，该函数会产生阻塞。

```
static inline int rpmsg_send_offchannel(struct rpmsg_endpoint *ept,
                                        uint32_t src, uint32_t dst,
                                        const void *data, int len)
```

4. rpmsg_trysend()

此函数在 ept 通道上发送长度为 len 的 data。消息将使用 ept 的源地址和目标地址发送到 ept 通道所属的远程处理器。

```
static inline int rpmsg_trysend(struct rpmsg_endpoint *ept,
                                const void *data,int len)
```

5. rpmsg_trysendto()

此函数将长度为 len 的 data 发送到远程 dst 地址。消息将使用 ept 的源地址发送到 ept 通道所属的远程处理器。

```
static inline int rpmsg_trysendto(struct rpmsg_endpoint *ept,
                                  const void *data,int len, uint32_t dst)
```

6. rpmsg_trysend_offchannel()

此函数将 len 长度的 data 发送到远程 dst 地址，并使用 src 作为源地址。该消息将被发送到 ept 通道所属的远程处理器。

```
static inline int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept,
                                           uint32_t src, uint32_t dst,
                                           const void *data, int len)
```

5.3 单次接收的数据量

5.3.1 默认单次接收 512B

值得注意的是，我们在第五章 RPMsg 相关驱动简介章节介绍到：在 RPMsg 的消息格式中，前 16 字节是消息头，在消息头的后面才是用户要发送的消息数据，而且在 Linux 下默认定义了 RPMsg 通信的最大 buffers 大小为 512 字节，512 字节减去 16 字节的消息头以后剩下 496 字节的用户数据，那么实际上一次最多只能传输 496 字节（加上最后的换行符）的用户数据。

如果 A7 一次发送 495 字节的用户数据，后面加上一个换行字符就组合成了 496 字节，用户数据的前面加上 16 字节的消息头就刚好是 512 字节，这 512 字节的数据可以一次性发送出去，如下图 5.3.1.1 所示。

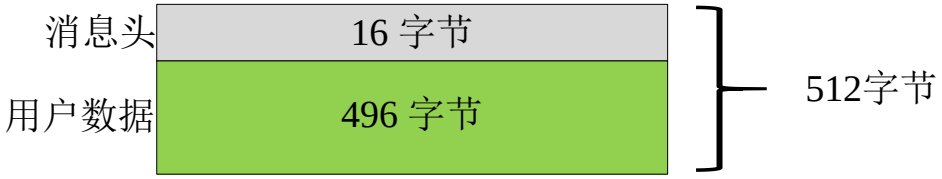


图 5.3.1.1 A7 默认最大发送 512 字节数据

如果 A7 一次发送的用户数据大于 496 字节，那么 M4 将分多次接收这些数据：假设 A7 一次发送 498 字节的用户数据（算上换行符），加上 16 字节的消息头，则用户一次发送的数据是 514 字节，那么 M4 将分为两次接收这些数据，M4 第一次接收到的是前面的 512 个字节的数据（包括消息头），第二次再接收第 513 个字节的用户数据和最后一个换行符，即第二次接收到最后的两个字节数据。

5.3.2 增大单次接收的数据量

注意：本小节的内容是为了第六章做铺垫。

如果实现 M4 一次能接收大于 512 字节的数据？假设 A7 一次发送 1024 字节的数据，如何实现 M4 一次性接收 1024 字节的数据？这个时候就需要修改 stm32mp157d-atk.dtsi 设备树下的 vdev0vring0、vdev0vring1 和 vdev0buffer 以及 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的宏 MAX_RPMSG_BUF_SIZE，下面我们来分析它们之间的关系。

我们先看设备树 stm32mp157d-atk.dtsi 下的配置：

```
vdev0vring0: vdev0vring0@10040000 {
            compatible = "shared-dma-pool";
            reg = <0x10040000 0x1000>;

            no-map;
    };

vdev0vring1: vdev0vring1@10041000 {
            compatible = "shared-dma-pool";
            reg = <0x10041000 0x1000>;

            no-map;
    };

vdev0buffer: vdev0buffer@10042000 {
            compatible = "shared-dma-pool";
            reg = <0x10042000 0x4000>;

            no-map;
};
```

vdev0vring0 配置的地址长度是 0x1000，即 4096 字节；

vdev0vring1 配置的地址长度是 0x1000，即 4096 字节；

vdev0buffer 配置的地址长度是 0x4000，即 16384 字节。

其中 vdev0vring0 和 vdev0vring1，一个是用于发送，一个是用于接收。vdev0vring0+vdev0vring1+vdev0buffer=24576 字节。M4 工程的链接脚本 STM32MP157DAAX_RAM.ld 配置的 m_ipc_shm（即 IPC 缓冲区）大小为 0x00008000，为 32768 字节：

```
MEMORY
{
  m_interrupts (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000298
  m_text       (RX)  : ORIGIN = 0x10000000, LENGTH = 0x00020000
  m_data       (RW)  : ORIGIN = 0x10020000, LENGTH = 0x00020000
  m_ipc_shm    (RW)  : ORIGIN = 0x10040000, LENGTH = 0x00008000
}
```

实际上，它们的关系是：vdev0vring0+vdev0vring1+vdev0buffer ≤ m_ipc_shm，也就是说，配置的 m_ipc_shm 最小值是（vdev0vring0+vdev0vring1+vdev0buffer），如果配置 m_ipc_shm 比（vdev0vring0+vdev0vring1+vdev0buffer）小，A7 加载和启动 M4 固件以后，A7 和 M4 无法正常进行核间通信。

我们再来看 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的宏 MAX_RPMSG_BUF_SIZE 表示配置每个缓冲区的大小（buffer_size）为 512 字节：

```
#define MAX_RPMSG_NUM_BUFS   (512)/* 最多 512 个缓冲区（收和发方向各 256 个） */
#define MAX_RPMSG_BUF_SIZE   (512)/* 缓冲区大小为 512 字节，16 个字节用于消息头 */
```

M4 工程 RPMsg_TEST\CM4\OPENAMP\openamp_conf.h 文件中的 VRING_NUM_BUFFS 表示每个 vring 的缓冲区的数量，默认配置有 16 个缓冲区，其中共有两个 vring，分别用于收和发，即 Vring 0（对应设备树下配置的 vdev0vring0）和 Vring 1（对应设备树下配置的 vdev0vring1）都各有 16 个缓冲区：

```
#if defined LINUX_RPROC_MASTER
#define VRING_RX_ADDRESS        -1      /* 主处理器分配：CA7 */
#define VRING_TX_ADDRESS        -1      /* 主处理器分配：CA7 */
#define VRING_BUFF_ADDRESS      -1      /* 主处理器分配：CA7 */
#define VRING_ALIGNMENT         16      /* 固定以匹配 linux 约束 */
#define VRING_NUM_BUFFS         16      /* RPMsg 缓冲区的数量 */
#else
```

那么每个 vring（发送或者接收）占用的缓冲区大小= buffer_size * 缓冲区数=512*16=8192 字节，两个 vring 占用的缓冲区大小为 8192*2=16384，这就是为什么以上 vdev0buffer 配置为 16384 字节，vdev0buffer 的大小仅和缓冲区的大小和缓冲区的个数有关。

vring（发送和接收，即 vdev0vring0 和 vdev0vring1）和 M4 工程 RPMsg_TEST\CM4\OPENAMP\openamp_conf.h 文件中的 VRING_NUM_BUFFS 值有关，一般不会修改该宏的值（默认为 16），如果要修改该值，可能设备树下 vdev0vring0 和 vdev0vring1 的值要进行修改，具体要不要修改，应根据 vring 需要多大的内存来配置。

内核源码 include/uapi/linux/virtio_ring.h 下的函数 vring_size()用于计算 vring（即 vdev0vring0 和 vdev0vring1）需要的内存大小，如下所示，其中 num 是 vring 的描述符数量，也就是 VRING_NUM_BUFFS 的值，注意，如果要修改该值，该值一般是 2^N，如修改为 16、32、64 和 128 等等，align 也就是 VRING_ALIGNMENT，为 16：

```
static inline unsigned vring_size(unsigned int num, unsigned long align)
```

```
{
  return ((sizeof(struct vring_desc) * num + sizeof(__virtio16) * (3 + num)
         + align - 1) & ~(align - 1))
        + sizeof(__virtio16) * 3 + sizeof(struct vring_used_elem) * num;
}
```

以上 vring_size()的返回值只是一个中间值，我们可以加上 printk()将其打印出来，如下所示，红色字体的代码是我们手动添加的，打印了 vring 的个数（num）以及 vring_size()的返回值（tmp）：

```
static inline unsigned vring_size(unsigned int num, unsigned long align)
{
  unsigned tmp;
  /*return ((sizeof(struct vring_desc) * num + sizeof(__virtio16) * (3 + num)
         + align - 1) & ~(align - 1))
        + sizeof(__virtio16) * 3 + sizeof(struct vring_used_elem) * num;*/
  tmp =  ((sizeof(struct vring_desc) * num + sizeof(__virtio16) * (3 + num)
         + align - 1) & ~(align - 1))
        + sizeof(__virtio16) * 3 + sizeof(struct vring_used_elem) * num;
  printk("wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww%d, %d\n", num, tmp);
  return tmp;
}
```

在 drivers/remoteproc/remoteproc_core.c 下找到 rproc_alloc_vring()函数，该函数调用了以上 vring_size()函数，size 是最终计算出的 vring 需要的内存大小（字节），我们可以将 size 打印出来，通过 size 知道 vdev0vring0 和 vdev0vring1 应该占用多大内存空间，如下代码，红色部分是我们手动添加的：

```
int rproc_alloc_vring(struct rproc_vdev *rvdev, int i)
{
  struct rproc *rproc = rvdev->rproc;
  struct device *dev = &rproc->dev;
  struct rproc_vring *rvring = &rvdev->vring[i];
  struct fw_rsc_vdev *rsc;
  int ret, size, notifyid;
  struct rproc_mem_entry *mem;

  /* vring 的实际大小（字节）*/
  size = PAGE_ALIGN(vring_size(rvring->len, rvring->align));
  printk("**************************size %d",size);
  rsc = (void *)rproc->table_ptr + rvdev->rsc_offset;

  /* 搜索预先注册的 carveout */
  mem = rproc_find_carveout_by_name(rproc, "vdev%dvring%d",
                                    rvdev->index,i);
  /* 省略部分代码 */
}
```

在以上代码中，函数 rproc_alloc_vring()通过调用 PAGE_ALIGN(addr)来将 addr 加大对齐到 4096 字节的边界（即 4096 的整数倍），假设 addr=4096+3，则执行 PAGE_ALIGN(addr)以后，size=4096+4096，没错，最终要给 vring 分配这么多的地址，即使用不完这些分配的地址，未使用的地址也在那里空着。从这点你也看出了，如果我们配置设备树下的 vdev0vring0 和 vdev0vring1 的值，应该是 4K 对齐！也就是说配置的值是 4096 的整数倍，这点一定要注意了，设备树下的地址可不是随便配置的！

在内核源码的 drivers/remoteproc/remoteproc_core.c 下找到如下函数 rproc_check_carveout_da()，其中标红色的 printk()也是我们手动添加的，len 表示 vring 需要的内存大小，其值应该等于前面打印的 size 的值，mem->len 则表示设备树下配置的 vring（即 vdev0vring0 和 vdev0vring1）的地址长度：

```
static int rproc_check_carveout_da(struct rproc *rproc,
                                   struct rproc_mem_entry *mem, u32 da, u32 len)
{
    struct device *dev = &rproc->dev;
    int delta;
    printk("****************************************len %d",len);
    printk("****************************************men %d",mem->len);

    /* 检查请求的资源长度 */
    if (len > mem->len) {
            dev_err(dev, "Registered carveout doesn't fit len request\n");
            return -EINVAL;
    }

    if (da != FW_RSC_ADDR_ANY && mem->da == FW_RSC_ADDR_ANY) {
            /* 地址与注册的 carveout 不匹配 */
            return -EINVAL;
    } else if (da != FW_RSC_ADDR_ANY && mem->da != FW_RSC_ADDR_ANY) {
            delta = da - mem->da;

      /*检查请求的资源是否属于已注册的 Carvout */
      if (delta < 0) {
              dev_err(dev,
                      "Registered carveout doesn't fit da request\n");
              return -EINVAL;
      }

      if (delta + len > mem->len) {
              dev_err(dev,
                      "Registered carveout doesn't fit len request\n");
              return -EINVAL;
      }
    }
```

```
    return 0;
}
```

通过打印 len（或者 size）的值，我们就知道 vdev0vring0 和 vdev0vring1 应该占用多大内存空间，通过打印 mem->len 的值，我们就知道在设备树下实际配置的 vdev0vring0 和 vdev0vring1 的地址大小。所以，在加载和启动固件时，通过比较 len 和 mem->len 的值，我们就知道如何调整设备树下 vdev0vring0 和 vdev0vring1 的地址范围了。

以上就是几个内存配置的对应关系，如果要修改某一个的值，相应的其它值也要根据计算进行修改，我们将其对应关系汇总如下表 5.3.2.1 所示：

	关系描述
1	vdev0vring0+vdev0vring1+vdev0buffer ≤ m_ipc_shm
2	vdev0buffer= MAX_RPMSG_BUF_SIZE * VRING_NUM_BUFFS
3	vdev0vring0 和 vdev0vring1 可根据以上打印的 len（或者 size）、mem->len 的值来确定： len（或者 size）---应该占用多大内存 mem->len---设备树下实际配置的 vdev0vring0 和 vdev0vring1 的地址大小

表 5.3.2.1 vdev0vring0、vdev0vring1、vdev0buffer 的配置要求

5.3.3 单次接收 1024B

注意：本小节的内容是为了第六章做铺垫。

我们回到前面提到的问题，如何实现 M4 单次接收 1024 字节的数据，此时 M4 工程 RPMsg_TEST\CM4\OPENAMP\openamp_conf.h 下的 VRING_NUM_BUFFS 保持默认的配置，为 16，将 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的宏 MAX_RPMSG_BUF_SIZE 改为 1024：

```
#define MAX_RPMSG_NUM_BUFS   (512)/* 最多 512 个缓冲区（收和发方向各 256 个） */
/* 缓冲区大小为 512 字节，16 个字节用于消息头 */
#define MAX_RPMSG_BUF_SIZE   (1024)
```

接下来我们先计算一下怎么修改设备树下 vdev0buffer 的大小：

1024*16*2=32768 字节，转化为 16 进制为 0x8000，即设备树 stm32mp157d-atk.dtsi 下 vdev0buffer 的地址长度配置为 8000。vdev0vring0 和 vdev0vring1 保持默认值，修改后的设备树配置如下：

```
mcuram2: mcuram2@10000000 {
      compatible = "shared-dma-pool";
      reg = <0x10000000 0x40000>;

      no-map;
};

vdev0vring0: vdev0vring0@10040000 {
      compatible = "shared-dma-pool";
      reg = <0x10040000 0x1000>;

      no-map;
};
```

```
vdev0vring1: vdev0vring1@10041000 {
      compatible = "shared-dma-pool";
      reg = <0x10041000 0x1000>;

      no-map;
};

vdev0buffer: vdev0buffer@10042000 {
      compatible = "shared-dma-pool";
      reg = <0x10042000 0x8000>;

      no-map;
};

mcuram: mcuram@30000000 {
    compatible = "shared-dma-pool";
    reg = <0x30000000 0x40000>;

    no-map;
};
retram: retram@38000000 {
      compatible = "shared-dma-pool";
      reg = <0x38000000 0x10000>;

      no-map;
};
```

根据以上配置，我们计算一下需要配置的 m_ipc_shm 应该多大：vdev0vring0+ vdev0vring1+ vdev0buffer= 0x1000+ 0x1000+ 0x8000=0xA000 。 下 面 我 们 将 M4 工 程 STM32MP157DAAX_RAM.ld 文件中内存区域定义改写如下：

```
MEMORY
{
  m_interrupts (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000298
  m_text       (RX)  : ORIGIN = 0x10000000, LENGTH = 0x00020000
  m_data       (RW)  : ORIGIN = 0x10020000, LENGTH = 0x00020000
  m_ipc_shm    (RW)  : ORIGIN = 0x10040000, LENGTH = 0x0000A000
}
```

修改好以后，重新编译 M4 工程生成最新的 RPMsg_TEST_CM4.elf 文件，并将该文件拷贝到开发板的/lib/firmware 目录下。

由于笔者前面在内核源码中添加了 printk()语句，所以要重新编译内核和设备树（如果只是修改了设备树，没有修改内核源码的话，一般编译设备树就可以了）。

笔者使用的是出厂的 Linux 系统来测试，所以使用的是资料盘中 01、程序源码\01、正点原子 Linux 出厂系统源码下的内核源码。在内核源码的根目录下，按照顺序执行如下命令去清理

内核、配置内核和编译内核，如果之前配置过内核，可以直接执行最后的一条命令来编译内核，如果之前未配置过内核，请按照顺序全部执行如下命令：

```
/* 清理内核 */
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- distclean
/* 配置内核 */
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf-
stm32mp1_atk_defconfig
/* 编译内核和设备树 */
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- uImage dtbs
LOADADDR=0XC2000040 -j4
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- uImage dtbs LOADADDR=0XC2000040 -j4
  DTC     arch/arm/boot/dts/stm32mp157d-atk-mipi.dtb
  DTC     arch/arm/boot/dts/stm32mp157d-atk-hdmi.dtb
  DTC     arch/arm/boot/dts/stm32mp157d-atk.dtb
  CALL    scripts/atomic/check-atomics.sh
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
  CC      drivers/rpmsg/virtio_rpmsg_bus.o
  AR      drivers/rpmsg/built-in.a
  AR      drivers/built-in.a
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  AR      init/built-in.a
  LD      vmlinux.o
  MODPOST vmlinux.o
  MODINFO modules.builtin.modinfo
  LD      .tmp_vmlinux1
  KSYM    .tmp_kallsyms1.o
  LD      .tmp_vmlinux2
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  GZIP    arch/arm/boot/compressed/piggy_data
  AS      arch/arm/boot/compressed/piggy.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-5.4.31
Created:      Thu May  5 12:31:53 2022
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    8324016 Bytes = 8128.92 KiB = 7.94 MiB
Load Address: c2000040
Entry Point:  c2000040
  Kernel: arch/arm/boot/uImage is ready
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$
```

图 5.3.3.1 编译内核

将编译出来的内核 uImage 和设备树 stm32mp157d-atk.dtb 拷贝到开发板的/boot 目录下，此处笔者的开发板的 IP 地址是 192.168.1.12，开发板和 ubuntu 是能够互相 ping 通的，笔者使用 scp 命令来传输文件，文件宝贝到开发板中以后，需要重启开发板，新替换的内核和设备树才会生效：

```
scp arch/arm/boot/uImage arch/arm/boot/dts/stm32mp157d-atk.dtb
root@192.168.1.12:/boot
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ scp arch/arm/boot/uImage arch/arm/boot/dts/stm32mp157d-atk.dtb root@192.168.1.12:/boot
uImage                                                                    100% 8129KB   4.6MB/s   00:01
stm32mp157d-atk.dtb                                                       100%   73KB   1.0MB/s   00:00
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$
```

图 5.3.3.2 拷贝内核和设备树到开发板的/boot 目录

开发板的/lib/modules 目录下有很多文件，包括和驱动相关的依赖文件，以及一些内核下配置为编译成模块（M）的驱动.ko 文件，一般加载模块（驱动）时用到，我们称该目录下的文件是内核模块。例如，在内核下的 uvc 驱动会默认编译成模块.ko 文件，并存放于/lib/modules/$(uname -r)/kernel/drivers/media/usb/uvc 下，如果/lib/modules/下没有任何文件（即没有内核模块），那么开发板无法使用 USB 摄像头。

```
root@ATK-MP157:/lib/modules/5.4.31/kernel/drivers/media/usb/uvc# ls
uvcvideo.ko
```

图 5.3.3.3 uvc 驱动

编译并替换了开发板的内核以后，接下来还需要编译内核模块，并把内核模块安装到开发板的/lib/modules 目录下。为什么要做这一步操作呢，可能很多人使用出厂的 Linux 系统（即出厂的内核、出厂的设备树和出厂的文件系统）来测试例程，开发板/lib/modules 目录下的文件是出厂系统自带的，当开发板中使用自己编译出来的内核时，由于自己的内核版本和出厂系统/lib/modules 目录下的内核模块的版本不匹配，可能导致有些驱动无法加载，此时的解决办法：

（1）修改开发板/lib/modules 目录下的内核模块版本为当前开发板使用的内核版本；

（2）自己编译和安装内核模块。

一般尝试方法（1）无法解决时，我们会使用（2）的方法来解决，笔者也比较推荐大家使用方法（2），因为方法（2）能彻底解决内核版本和内核模块版本不匹配的问题。

下面我们先试试方法（1），在开发板中执行 uname -r 可以查看开发板的内核版本，如下图 5.3.3.4 所示，查看笔者编译的内核版本是 5.4.31：

```
root@ATK-MP157:~# uname -r
5.4.31
root@ATK-MP157:~#
```

图 5.3.3.4 查看内核版本

进入到/lib/modules 目录下，该目录下只有一个目录 5.4.31-g8c3068500，这就是出厂系统自带的内核模块版本，并不是笔者自己编译的，里边就是内核模块相关的文件，可以看到内核模块版本是 5.4.31-g8c3068500，然而前面我们查询到的内核版本为 5.4.31，很明显内核版本和内核模块版本不一样，可能会出现一些驱动无法加载的情况。

```
root@ATK-MP157:~# cd /lib/modules/
root@ATK-MP157:/lib/modules# ls
5.4.31-g8c3068500
root@ATK-MP157:/lib/modules#
```

图 5.3.3.5 查看内核模块版本

执行如下命令，将内核模块版本和内核版本修改为一致，如下图 5.3.3.6 所示：

```
root@ATK-MP157:/lib/modules# mv 5.4.31-g8c3068500 $(uname -r)
root@ATK-MP157:/lib/modules# ls
5.4.31
root@ATK-MP157:/lib/modules#
```

图 5.3.3.6 修改内核模块版本等于内核版本

修改好以后，如果后期测试无法加载某个驱动，例如报错提示“version magic”、“ disagrees about version of symbol device_create”、“ modprobe: can't load module xxx.ko (xxx.ko): Invalid argument”等等，一般都是内核版本和内核模块版本不匹配导致的，此时建议采用方法（2）。

方法（2）是先使用内核源码编译内核和设备树，然后再使用同一个内核源码去编译内核模块，再把内核、设备树和内核模块放到开发板对应的目录下，前面我们已经完成了内核和设备树的编译，并将内核和设备树放到了开发板的/boot 目录下了，下面我们还需要编译和安装内核模块。

首先先在内核源码的根目录下新建一个目录 mymodules，再依次执行如下命令编译内核模块并将内核模块安装到新建的 mymodules 目录下，

```
/* 在内核源码的根目录下新建 mymodules 目录 */
```

```
mkdir mymodules
/* 编译内核模块 modules */
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- modules
/* 将内核模块 modules 安装到新建的 mymodules 目录下 */
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules_install
INSTALL_MOD_PATH=mymodules
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ mkdir mymodules
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- modules
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
  CC [M]  drivers/crypto/virtio/virtio_crypto_algs.o
  CC [M]  drivers/crypto/virtio/virtio_crypto_mgr.o
  CC [M]  drivers/crypto/virtio/virtio_crypto_core.o
  LD [M]  drivers/crypto/virtio/virtio_crypto.o
  CC [M]  drivers/net/macvtap.o
  CC [M]  drivers/net/tun.o
  CC [M]  drivers/net/tap.o
  CC [M]  net/vmw_vsock/virtio_transport.o
  LD [M]  net/vmw_vsock/vmw_vsock_virtio_transport.o
  CC [M]  net/vmw_vsock/virtio_transport_common.o
  LD [M]  net/vmw_vsock/vmw_vsock_virtio_transport_common.o
  Building modules, stage 2.
  MODPOST 531 modules
  LD [M]  drivers/crypto/virtio/virtio_crypto.ko
  LD [M]  drivers/net/macvtap.ko
  LD [M]  drivers/net/tap.ko
  LD [M]  drivers/net/tun.ko
  LD [M]  net/vmw_vsock/vmw_vsock_virtio_transport.ko
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules_install INSTALL_MOD_PATH=mymodules
  INSTALL arch/arm/crypto/aes-arm-bs.ko
  INSTALL arch/arm/crypto/aes-arm-ce.ko
  INSTALL arch/arm/crypto/aes-arm.ko
  INSTALL arch/arm/crypto/chacha-neon.ko
  INSTALL arch/arm/crypto/crc32-arm-ce.ko
  INSTALL arch/arm/crypto/ghash-arm-ce.ko
  INSTALL arch/arm/crypto/sha1-arm-ce.ko
  INSTALL arch/arm/crypto/sha1-arm-neon.ko
  INSTALL arch/arm/crypto/sha1-arm.ko
```

图 5.3.3.7 编译和安装内核模块

进入到刚才新建的 mymodules 目录下，可以看到有安装好的内核模块，如下图 5.3.3.8 所示：

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel$ cd mymodules;ls
lib
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules$ cd lib;ls
modules
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib$ cd modules;ls
5.4.31
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ cd 5.4.31;ls
build   modules.alias      modules.builtin      modules.builtin.modinfo  modules.dep.bin  modules.order    modules.symbols      source
kernel  modules.alias.bin  modules.builtin.bin  modules.dep              modules.devname  modules.softdep  modules.symbols.bin
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules/5.4.31$
```

图 5.3.3.8 安装好的内核模块

mymodules/lib/modules/下的文件就是我们要安装（拷贝）到开发板的/lib/modules 目录下的文件，在 mymodules/lib/modules 目录下执行如下命令，将 mymodules/lib/modules 下的 5.4.31 文件夹打包成 modules.tar.bz2，如下图 5.3.3.9 所示，一定要注意，以下命令是在 mymodules/lib/modules 目录下执行的！

```
tar -vcjf modules.tar.bz2 *        /* 打包内核模块 */
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ ls
5.4.31
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ pwd
/home/alientek/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ ls
5.4.31
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ tar -vcjf modules.tar.bz2 *
5.4.31/
5.4.31/modules.softdep
5.4.31/modules.dep.bin
5.4.31/build
5.4.31/modules.order
5.4.31/modules.alias
5.4.31/modules.devname
5.4.31/modules.builtin.modinfo
5.4.31/modules.alias.bin
5.4.31/source
5.4.31/modules.builtin.bin
5.4.31/modules.dep
5.4.31/kernel/
5.4.31/kernel/sound/
5.4.31/kernel/sound/core/
5.4.31/kernel/sound/core/snd-hwdep.ko
```

图 5.3.3.9 打包内核模块

成功打包后，在 mymodules/lib/modules 目录下多了一个压缩包 modules.tar.bz2，我们把该压缩包拷贝到开发板的/lib/modules 目录下，如下图 5.3.3.10 所示：

```
/* 拷贝内核模块到开发板中 */
scp modules.tar.bz2 root@192.168.1.12:/lib/modules
```

```
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ ls
5.4.31  modules.tar.bz2
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ pwd
/home/alientek/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$ scp modules.tar.bz2 root@192.168.1.12:/lib/modules
modules.tar.bz2                                                                                100%  117MB
4.2MB/s   00:28
alientek@ubuntu:~/STM32MP157/kernel_and_uboot/kernel/mymodules/lib/modules$
```

图 5.3.3.10 拷贝内核模块到开发板中

在开发板中可以看到拷贝成功的压缩包，下面我们解压此压缩包，在解压之前可以将原来的内核模块删除掉，然后执行如下命令进行解压，如下图 5.3.3.11 所示：

```
rm -rf 5.4.31                  /* 删除原有的内核模块 */
tar -xf modules.tar.bz2        /* 解压内核模块 */
```

```
root@ATK-MP157:/lib/modules# ls
5.4.31  modules.tar.bz2
root@ATK-MP157:/lib/modules# rm -rf 5.4.31
root@ATK-MP157:/lib/modules# ls
modules.tar.bz2
root@ATK-MP157:/lib/modules# tar -xf modules.tar.bz2
tar: 5.4.31/modules.softdep: time stamp 2022-05-05 15:04:40 is 70632473.260248723 s in the future
tar: 5.4.31/modules.dep.bin: time stamp 2022-05-05 15:04:39 is 70632472.256430432 s in the future
tar: 5.4.31/modules.order: time stamp 2022-05-05 15:04:18 is 70632451.254301515 s in the future
tar: 5.4.31/modules.alias: time stamp 2022-05-05 15:04:39 is 70632472.248812307 s in the future
tar: 5.4.31/modules.devname: time stamp 2022-05-05 15:04:40 is 70632473.247483931 s in the future
tar: 5.4.31/modules.builtin.modinfo: time stamp 2022-05-05 15:04:18 is 70632451.233682431 s in the
tar: 5.4.31/modules.alias.bin: time stamp 2022-05-05 15:04:40 is 70632473.225091556 s in the futur
tar: 5.4.31/modules.builtin.bin: time stamp 2022-05-05 15:04:40 is 70632473.222516556 s in the fut
tar: 5.4.31/modules.dep: time stamp 2022-05-05 15:04:39 is 70632472.217755056 s in the future
```

图 5.3.3.11 解压内核模块

解压完成后，在/lib/modules 下多了个目录 5.4.31，里边就是内核模块文件，如下图 5.3.3.12 所示：

```
root@ATK-MP157:/lib/modules# ls
5.4.31  modules.tar.bz2
root@ATK-MP157:/lib/modules# cd 5.4.31;ls
build          modules.alias.bin    modules.builtin.modinfo  modules.devname  modules.symbols
kernel         modules.builtin      modules.dep              modules.order    modules.symbols.bin
modules.alias  modules.builtin.bin  modules.dep.bin          modules.softdep  source
root@ATK-MP157:/lib/modules/5.4.31#
root@ATK-MP157:/lib/modules/5.4.31# pwd
/lib/modules/5.4.31
root@ATK-MP157:/lib/modules/5.4.31#
```

图 5.3.3.12 成功将内核模块安装到开发板中

可能有的小伙伴会疑惑，为啥要将内核模块打包以后再拷贝到开发板中解压呢？直接使用 scp 指令将内核模块下的所有文件都拷贝到开发板下不就完了吗？这样子也是可以的，只不过 scp 是以单个文件进行拷贝，拷贝完一个文件再拷贝下一个文件，有些软连接文件也会被拷贝，内核模块下有很多文件，拷贝的时间长不说，可能还会导致个别文件未拷贝成功，所以笔者直接打包后再拷贝，再解压，节省了时间，也能保证文件不丢失。

更新了内核 uImage、设备树 stm32mp157d-atk.dtb、内核模块和 M4 固件 RPMsg_TEST_CM4.elf 以后，下面我们加载和运行固件包，检查固件是否能正常运行，关于 A7 一次发送大于 1024 个字节数据，M4 一次性接收 1024 个字节数据的具体测试方法可参考第六章，本小节，我们只验证程序是否能够正常运行。

进入/lib/firmware/目录下，直接运行第四章我们新建的 test1.sh 脚本，如下图 5.3.3.13 所示，可以看到程序正常运行了，说明前面我们修改的配置成功了。同时也打印了我们在内核源码下添加的打印信息，其中打印 vring 的个数为 16，正好等于 M4 工程下 VRING_NUM_BUFFS 的值，len=size=4096 字节，表示 vdev0vring0 和 vdev0vring1 应该占用 4096 字节的内存，打印的 mem 也为 4096 字节，mem 表示在设备树下实际配置的 vdev0vring0 和 vdev0vring1 的内存大小，前面我们在设备树下配置的 vdev0vring0 和 vdev0vring1 大小刚好等于 4096 字节。

```
root@ATK-MP157:/lib/firmware# ./test1.sh
[   35.893149] remoteproc remoteproc0: powering up m4
[   36.010559] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2237256
[   36.027034] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[   36.030186] **************************size 4096
[   36.030192] ******************************************len 4096
[   36.034708] ******************************************men 4096
[   36.046927] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[   36.056082] **************************size 4096
[   36.056090] ******************************************len 4096
[   36.076892] ******************************************men 4096
[   36.081692]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[   36.120995] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[   36.124163] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[   36.137039] virtio_rpmsg_bus virtio0: rpmsg host is online
[   36.137156] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x0
[   36.144768]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[   36.155355] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:/lib/firmware# [   36.217215] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: new channel
: 0x400 -> 0x0!
[   36.341268] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 1 (src: 0x0)
[   36.546213] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 2 (src: 0x0)
[   36.751208] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 3 (src: 0x0)
[   36.956210] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 4 (src: 0x0)
[   37.161206] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 5 (src: 0x0)
[   37.366210] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 6 (src: 0x0)
[   37.571211] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 7 (src: 0x0)
[   37.776209] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 8 (src: 0x0)
[   37.981209] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 9 (src: 0x0)
[   38.186214] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 10 (src: 0x0)
```

图 5.3.3.13 运行固件

小伙伴们可以尝试修改 MAX_RPMSG_BUF_SIZE 和 VRING_NUM_BUFFS 的值，然后计算应该配置 vdev0vring0、vdev0vring1、vdev0buffer 和 m_ipc_shm 的大小，再去修改设备树和链接脚本，通过运行程序以及观察以上的打印信息来验证自己修改的是否正确。

注意的是，在设备树下配置 vdev0vring0、vdev0vring1 时，地址最好是 4096 的整数倍，关于这点我们也在前面解释了，也正因为如此，当修改 VRING_NUM_BUFFS 的值为 16、32、6

4 和 128 时，len=size 都等于 4096。在调整 vdev0vring0、vdev0vring1、vdev0buffer 和 m_ipc_sh m 的值时，也要根据内存地址范围来修改，SRAM1~SRAM4 的地址是连续的，范围为 0x10000000~0x10060000，地址不要超过该范围，如果要占用 SRAM4 的地址，记得将设备树下 A7 占用的 SRAM4 节点给注释掉，以释放 SRAM4，关于这点我们在前面也讲解过了。

5.4 基于 RPMsg 的异核通信实验

5.4.1 硬件设计

1. 例程功能

通过调用 RPMsg 相关的 API 接口，A7 和 M4 之间互相发送 100 条数据，A7 通过 UART4 打印从 M4 接收到的次数，M4 通过 USART3 打印从 A7 接收到的数据和接收到的次数。注意，RPMsg 通道建立后，必须先由 A7 先发送第一条数据。

该实验工程参考开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch5\RPMsg_TEST。

2. 硬件连接

硬件连接如下图5.4.1.1所示，如果开发板带了屏幕的话，可以在开发板接上屏幕用于测试。USB 转 TTL 串口(CH340)模块的 TX 引脚接在开发板 USART3 的 U3_RX 引脚上，模块的 RX 引脚接在开发板 USART3 的 U3_TX 引脚上，模块另一端通过一根 T 口 USB 线接到电脑的 USB 接口，目的是打印 M4 内核的信息。

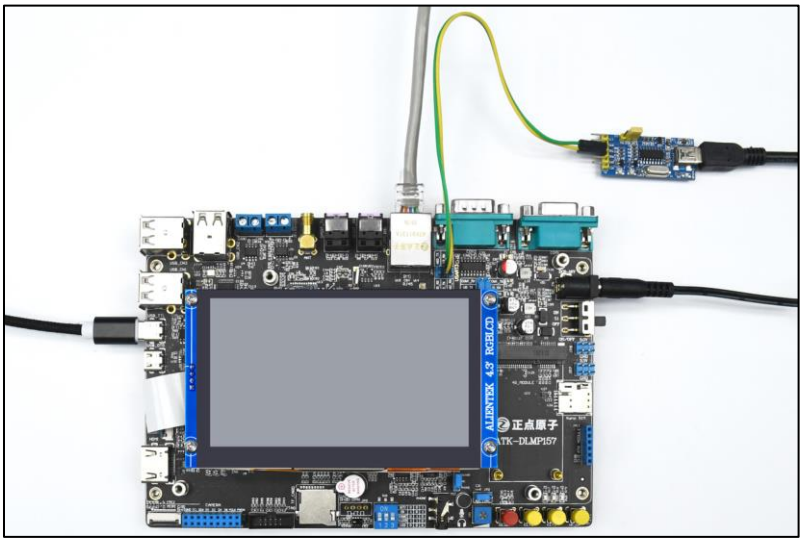


图 5.4.1.1 硬件连接图

5.4.2 软件设计

1. 程序流程图

根据上述例程功能分析，我们得到以下程序流程图，如下图 5.4.2.1 所示：

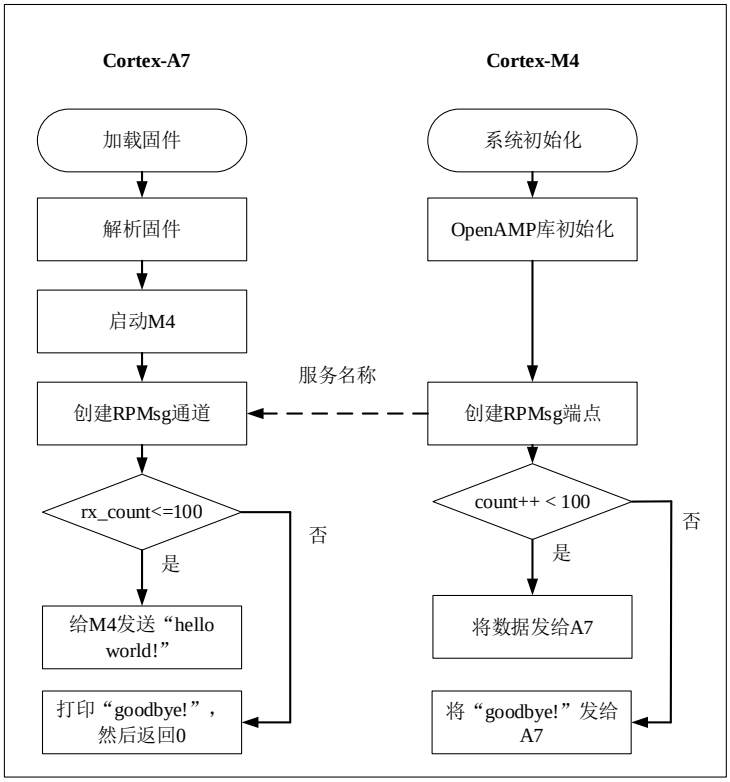


图 5.4.2.1 程序流程图

2. Linux 下程序解析

在 Linux 内核源码下有 ST 已经编写好的 RPMsg 驱动示例文件，即 samples/rpmsg/rpmsg_client_sample.c 文件，实际上 STM32MP157 开发板的出厂 Linux 操作系统已经将此驱动编译成驱动模块 rpmsg_client_sample.ko 文件了，在开发板出厂 Linux 文件系统的 /lib/modules/$(uname -r)/kernel/samples/rpmsg 目录下，所以本实验我们可以直接使用此驱动文件来完成后续的实验，如下图 5.4.2.2 所示：

```
root@ATK-MP157:~# cd /lib/modules/$(uname -r)/kernel/samples/rpmsg
root@ATK-MP157:/lib/modules/5.4.31/kernel/samples/rpmsg# ls
rpmsg_client_sample.ko
root@ATK-MP157:/lib/modules/5.4.31/kernel/samples/rpmsg#
```

图 5.4.2.2 RPMsg 客户端驱动文件

下面，我们来分析 rpmsg_client_sample.c 文件，我们将此文件的程序分为三个部分进行分析。第一部分，函数 rpmsg_sample_probe()用于 A7 创建 RPMsg 设备的私有数据 idata，并且向远程处理器首次发送数据，代码如下：

```
/**
 * @brief       创建私有数据，将数据发给 M4
 * @param       rpdev：RPMsg 通道设备
 * @retval      0：成功，其它：失败
 */
static int rpmsg_sample_probe(struct rpmsg_device *rpdev)
{
  int ret;
```

```
  struct instance_data *idata;
   /* A7 打印 src 和 dst 地址 */
  dev_info(&rpdev->dev, "new channel: 0x%x -> 0x%x!\n",
         rpdev->src, rpdev->dst);
  /* 内核分配内存 */
  idata = devm_kzalloc(&rpdev->dev, sizeof(*idata), GFP_KERNEL);
  if (!idata)
    return -ENOMEM;
  /* 设置 RPMsg 的私有数据，即 idata */
  dev_set_drvdata(&rpdev->dev, idata);

  /* 给远程处理器发送“hello world!”1 次 */
  ret = rpmsg_send(rpdev->ept, MSG, strlen(MSG));
  if (ret) {
    dev_err(&rpdev->dev, "rpmsg_send failed: %d\n", ret);
    return ret;
  }

  return 0;
}
```

rpmsg_sample_probe()函数为 RPMsg 设备的私有数据 idata 分配内存，idata 用于计数发送了多少次消息。第一条信息“hello world!”是由 rpmsg_sample_probe()函数发出去的，即 A7 给 M4 发第一条消息。此外，A7 会打印“new channel: 0xxxx -> 0xxxx!”(此处 xxx 是泛指，代表地址)。

第二部分，当 A7 接收到 M4 发来的数据时，就执行 rpmsg_sample_cb()函数，代码如下所示，此函数做的操作：

1）先获取私有数据 idata，每当 A7 接收到 M4 发来的数据的时候，idata 自加一，idata 也就表示 A7 接收到数据的次数，同时 A7 打印接收到的次数和 src 地址，此处的 src 地址是 M4 端的地址；

2）A7 给 M4 发送字符串“hello world!”，通过调用 Linux 下的 rpmsg_send()函数实现发送功能，当发完第 100 条（第一条由 rpmsg_sample_probe()函数发送）后，A7 打印“goodbye!”，并直接返回 0。

注意，此函数并未将 M4 发来的数据打印出来。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rpmsg.h>

#define MSG   "hello world!"     /* A7 发送的消息 */

static int count = 100;          /* 次数是 100 */
module_param(count, int, 0644);

struct instance_data {
```

```
  int rx_count;                 /* A7 接收到消息的次数 */
};
/**
 * @brief       A7 接收到 M4 发来的数据后，A7 将字符串发给 M4
 * @param       rpdev：RPMsg 通道设备
 *              data：发送的数据
 *              len： 数据长度
 *              priv：私有数据
 * @retval      0：成功，其它：失败
 */
static int rpmsg_sample_cb(struct rpmsg_device *rpdev, void *data,
                           int len,void *priv, u32 src)
{
  int ret;
  /* 获取 RPMsg 的私有数据 idata */
  struct instance_data *idata = dev_get_drvdata(&rpdev->dev);
  /* A7 打印接收到的次数和 src 地址 */
  dev_info(&rpdev->dev, "incoming msg %d (src: 0x%x)\n",\
           ++idata->rx_count, src);
  /* 内核使用动态调试功能，以十六进制打印转储信息 */
  print_hex_dump_debug(__func__, DUMP_PREFIX_NONE, 16, 1, data, len,true);

  /* 当 A7 接收 100 次后，A7 打印"goodbye! " */
  if (idata->rx_count >= count) {
    dev_info(&rpdev->dev, "goodbye!\n");
    return 0;
  }

  /* A7 将"hello world!" 字符串发给 M4 */
  ret = rpmsg_send(rpdev->ept, MSG, strlen(MSG));
  if (ret)
    dev_err(&rpdev->dev, "rpmsg_send failed: %d\n", ret);

  return 0;
}
```

第三部分，注册 rpmsg_sample_client 这个驱动，代码如下：

```
/*  rpmsg_driver_sample_id_table 结构体，用于绑定设备和驱动 */
static struct rpmsg_device_id rpmsg_driver_sample_id_table[] = {
  { .name = "rpmsg-client-sample" },
  { },
};
MODULE_DEVICE_TABLE(rpmsg, rpmsg_driver_sample_id_table);
```

```
static struct rpmsg_driver rpmsg_sample_client = {
  .drv.name = KBUILD_MODNAME,
  .id_table = rpmsg_driver_sample_id_table,
  .probe    = rpmsg_sample_probe,
  .callback = rpmsg_sample_cb,
  .remove   = rpmsg_sample_remove,
};
/* 注册 RPMsg 驱动程序 rpmsg_sample_client */
module_rpmsg_driver(rpmsg_sample_client);

MODULE_DESCRIPTION("Remote processor messaging sample client driver");
MODULE_LICENSE("GPL v2");
```

以上代码是 Linux 下平台设备和驱动匹配的方法之一，即通过 id_table 来匹配。

MODULE_DEVICE_TABLE(rpmsg, rpmsg_driver_sample_id_table)这一行第一个参数 rpmsg 是设备类型（type），为 rpmsg 类型，第二个参数是此驱动所支持的设备列表，从上面的代码可以看到表 rpmsg_driver_sample_id_table 中只有一个设备名字，为 rpmsg-client-sample。这行代码的作用是内核在运行时，根据设备类型（type）和设备列表中的名称（name）的对应关系，能够知道什么驱动对应什么设备，那么，当插入设备时，会检查该设备的类型以及设备 ID 值是否和设备驱动匹配，如果匹配，则可以迅速地加载驱动模块，这种机制通常会在热插拔中用到。

RPMsg 通道的建立是通过前面分析的 rpmsg_ns_cb()函数来完成的，首先由 M4 端创建一个 RPMsg 端点，端点关联的服务名称是 rpmsg-client-sample（在后面 M4 工程的程序讲解部分会进行分析），然后根据这个名称，作为主处理器的 A7 就创建一个名字为 rpmsg-client-sample 的 RPMsg 通道，并打印信息“creating channel rpmsg-client-sample addr 0xxx”(0xxx 代表地址)，只有通道建立以后，A7 和 M4 才可以进行核间通信。

在 rpmsg_driver_sample_id_table[]结构体中有一个 name 成员，其属性为 rpmsg-client-sample，如果 RPMsg 通道的名字也为 rpmsg-client-sample，那么设备和设备驱动就会匹配。当设备和设备驱动匹配时，驱动 rpmsg-client-sample.ko 就会被加载（此时，可以在 Linux 文件系统下执行 lsmod 指令查看加载了哪些驱动），内核就会执行 rpmsg_sample_client 中的 probe 处理程序（probe 是所有驱动的入口），即执行 rpmsg_sample_probe()，此时 A7 建立私有数据，且打印 src 和 dst 地址，并将“hello world!”发给 M4。接下来，M4 就可以使用 RPMsg 相关的 API 来发送数据了。

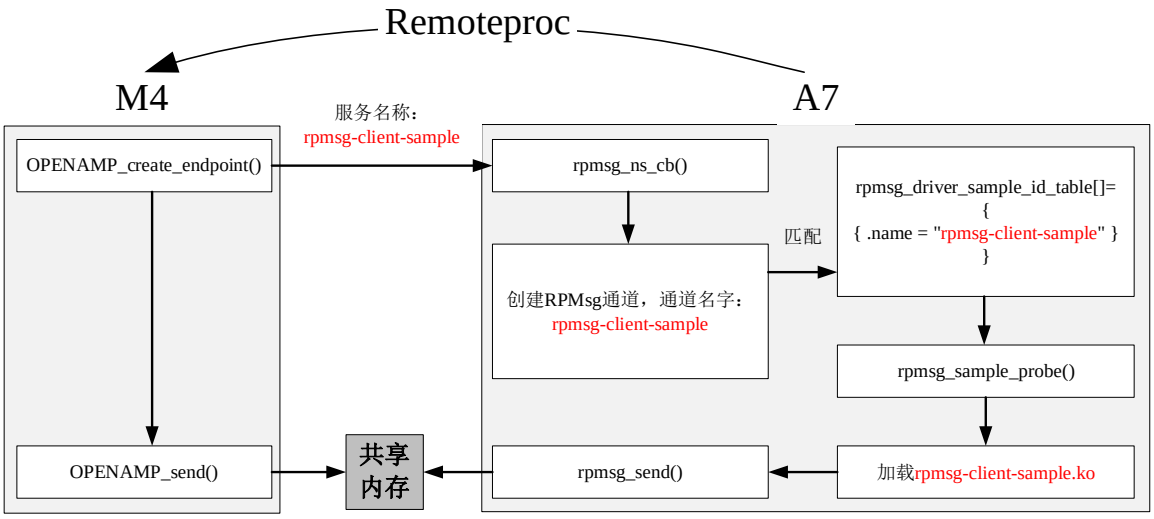


图 5.4.2.3 驱动 rpmsg-client-sample.ko 被加载过程

执行完 probe 处理程序后，接着执行和 probe 关联的回调处理程序 callback，即执行前面分析的 rpmsg_sample_cb()函数，此函数通过 rpmsg_send()函数来将“hello world!”发给 M4，当发送 100 次后随即打印“goodbye!”。

后面的 remove 处理程序是当需要时才会执行，当执行 rmmod 指令来卸载驱动时，rpmsg_sample_remove()函数就会被执行。

3. M4 工程下程序代码

下面我们在 main.c 文件中添加代码，如下所示，其中每组红色字体（在 USER CODE BEGIN XXX 和 USER CODE END XXX 之间）的代码是我们手动添加的：

```
#include "main.h"
#include "ipcc.h"
#include "openamp.h"
#include "usart.h"
#include "gpio.h"

/* USER CODE BEGIN PTD */
#define RPMSG_SERVICE_NAME              "rpmsg-client-sample"
__IO FlagStatus rx_status = RESET;     /* 接收标志位 */
uint8_t received_rpmsg[128];           /* 接收缓冲区 */
 /* 声明接收回调函数 */
static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data,
                       size_t len, uint32_t src, void *priv);
/* USER CODE END PTD */

void SystemClock_Config(void);

/* USER CODE BEGIN 0 */
/* 支持 UART3 通过通过 printf()函数打印信息 */
#ifdef __GNUC__
```

```
20  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
21  #else
22  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
23  #endif
24  PUTCHAR_PROTOTYPE
25  {
26  /* 本实验使用的是 UART3，如果使用的是其它串口，则将 UART3 改为对应的串口即可 */
27   while ((USART3->ISR & 0X40) == 0);/* 等待上一个字符发送完成 */
28   USART3->TDR = (uint8_t) ch;       /* 将要发送的字符 ch 写入到 TDR 寄存器 */
29   return ch;                        /* 返回要发送的字符 */
30  }
31  /* USER CODE END 0 */
32
33
34  int main(void)
35  {
36    /* USER CODE BEGIN 1 */
37    struct rpmsg_endpoint resmgr_ept;/* RPMsg 端点 */
38    uint32_t count = 0;              /* 发送次数 */
39    uint8_t msg[32];                 /* 拷贝缓冲区 */
40    /* USER CODE END 1 */
41
42    HAL_Init(); /* 初始化 HAL 库 */
43
44    if(IS_ENGINEERING_BOOT_MODE())    /* 检查平台是否为工程启动模式 */
45    {
46      /* 配置系统时钟 */
47      SystemClock_Config();
48    }
49
50     MX_IPCC_Init();                      /* 初始化 IPCC */
51     MX_OPENAMP_Init(RPMSG_REMOTE, NULL);/* 初始化 OpenAMP 库 */
52
53    /* 初始化所有已经配置的外设 */
54    MX_GPIO_Init();                       /* 初始化 GPIO */
55    MX_USART3_UART_Init();                /* 初始化 UART3 */
56
57    /* USER CODE BEGIN 2 */
58    /* 创建 RPMsg 端点函数 */
59    OPENAMP_create_endpoint(&resmgr_ept, RPMSG_SERVICE_NAME,
                              RPMSG_ADDR_ANY,rx_callback, NULL);
60    /* USER CODE END 2 */
61
```

```
62    while (1)
63    {
64
65     /* USER CODE BEGIN 3 */
66      OPENAMP_check_for_message();       /* 邮箱轮询,检查 A7 是否有发来数据 */
67      if (rx_status == SET)              /* M4 接收到 A7 发送过来的数据 */
68        {
69          /* 接收标志位复位，以便下次判断通道是否再次接收到数据 */
70          rx_status = RESET;
71
72          if (count++ < 100)
73          {
74           sprintf((char *)msg, "M4->A7 %02ld", count);
75          }
76          else
77          {
78           strcpy((char *)msg, "goodbye!");
79          }
80         /* 调用 OPENAMP_send 发送数据给 A7 */
81          if (OPENAMP_send(&resmgr_ept, msg, strlen((char *)msg)+1) < 0)
82          {
83           printf("Failed to send message\r\n");
84           Error_Handler();
85          }
86          printf("%s\r\n", msg);
87        }
88         HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_3);
89         HAL_Delay(100);
90         HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_3);
91         HAL_Delay(100);
92    }
93    /* USER CODE END 3 */
94  }
95
96  /**
97    * @brief 系统时钟初始化
98    * @retval None
99    */
100 void SystemClock_Config(void)
101 {
102   /* 该部分代码省略 */
103 }
104
```

```
105 /* USER CODE BEGIN 4 */
106 /**
107  * @brief       接收回调函数，用于处理 A7 发送过来的数据
108  * @param
109  *              rp_chnl:  rpmsg_endpoint 类型的结构体
110  *              data:     存放 A7 发过来的数据 BUFF
111  *              len:      data 数据的长度
112  *              src:      源地址（这里没用到）
113  *              priv:     用于设置私有数据（本函数也没用到）
114  * @note       如果 RPMsg 接收到数据，该回调函数将被调用
115  * @retval      无
116  */
117 static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data,
                           size_t len, uint32_t src, void *priv)
118 {
119   /* 将 A7 发送过来的数据拷贝到指定的内存 received_rpmsg 处 */
120   memcpy(received_rpmsg, data, len > sizeof(received_rpmsg) ?
            sizeof(received_rpmsg) : len);
121   /* M4 打印 A7 发送过来的数据 */
122   printf("received_rpmsg=%s\r\n", received_rpmsg);
123   /* 通道标志位置 1，表示接收到了函数 */
124   rx_status = SET;
125   return 0;
126 }
127 /* USER CODE END 4 */
```

第 7~第 13 行：

第 8 行，定义端点关联的服务名称为 rpmsg-client-sample；

第 9 行，定义一个接收标志位 rx_status，初始值为 RESET，表示没有接收到数据，若接收到数据，则该值为 SET；

第 10 行，定义接收缓冲区 received_rpmsg[128]，大小为 128 个字节。

第 17 行~第 31 行：

通过重映射的方式将 printf()函数重映射到 STM32 串口的寄存器上，那么串口可通过 printf()输出流，最终 UART3 可以打印信息。

第 36~40 行：

第 37 行，定义一个 RPMsg 端点；

第 38 行，定义 count 表示发送次数，初始值为 0；

第 39 行，定义一个拷贝缓冲区，大小为 32 个字节。

第 57~第 60 行：

调用 OPENAMP_create_endpoint()函数创建一个端点 resmgr_ept，端点关联的服务名称是 rpmsg-client-sample，目的地址设置为 RPMSG_ADDR_ANY，由系统查询对方的地址，端点关联的回调函数是 rx_callback。

第 65~第 93 行：

第 66 行，调用 OPENAMP_check_for_message()函数进行邮箱轮询操作，检查 Vring Buffer 中是否有数据，即检查 M4 是否有接收到 A7 发来的数据；

第 67~第 87 行：

如果 M4 接收到 A7 发来的数据，先将标志位 rx_status 复位（RESET），然后将拷贝缓冲区 msg 中的数据（字符串“M4->A7 count”，count 表示次数）发送给 A7，最大只能发送 100 次，同时 M4 打印拷贝缓冲区的数据。若 M4 发送失败，则打印“Failed to send message”提示发送失败了。

第 88~第 92 行的代码是第一章添加的，如不需要，可以将这段代码删除，此处是将延迟时间由 500ms 修改为 100ms，那么串口打印的速度会快一些。

第 105 行~127 行：

M4 创建的 RPMsg 端点 resmgr_ept 关联（绑定）的接收回调函数，当 M4 接收到数据时，该回调函数就被执行，该函数先将 M4 接收到的数据拷贝到接收缓冲区 received_rpmsg（最大只能拷贝 128 个字节），然后打印 received_rpmsg 的数据，再把接收标志位 rx_status 设置为 SET。

经过以上分析，可以看到端点关联的服务名称和 Linux 下 rpmsg_driver_sample_id_table 结构体中的 name 一样，那么，当 A7 加载和启动 M4 固件时，A7 收到 M4 发来的服务公告名称消息后，根据消息中的服务名称来创建一个名为 rpmsg-client-sample 的通道（可以理解为一个 RPMsg 设备），A7 和 M4 就可以通过这个通道来进行通信。RPMsg 设备与驱动匹配，驱动 rpmsg-client-sample.ko 被加载，A7 就给 M4 发送数据，其过程和前面分析的 Linux 下程序的过程一样，可参考图 5.4.2.3。

5.4.3 实验验证

1. 不使用屏幕进行测试

启动开发板，进入 Linux 操作系统中，执行 lsmod 指令可以查询此时 Linux 系统加载了哪些模块，可以看到没有 rpmsg_client_sample 这个驱动，如下图 5.4.3.1 所示：

```
lsmod
```

```
root@ATK-MP157:~# lsmod
Module                  Size  Used by
ipv6                  442368  32
nf_defrag_ipv6         20480  1 ipv6
8723ds               1363968  0
galcore               323584  6
stm32_dcmi             32768  0
videobuf2_dma_contig    20480  1 stm32_dcmi
videobuf2_memops       16384  1 videobuf2_dma_contig
videobuf2_v4l2         20480  1 stm32_dcmi
videobuf2_common       40960  2 stm32_dcmi,videobuf2_v4l2
ov5640                 28672  0
v4l2_fwnode            20480  2 ov5640,stm32_dcmi
videodev              176128  5 ov5640,v4l2_fwnode,videobuf2_common,stm32_dcmi,videobuf2_v4l2
spi_stm32              24576  0
mc                     36864  5 ov5640,videobuf2_common,videodev,stm32_dcmi,videobuf2_v4l2
ap3216c                16384  0
stm32_cec              16384  0
dht11                  16384  0
ds18b20                16384  0
root@ATK-MP157:~#
```

图 5.4.3.1 查看系统加载了哪些模块

编译以上 M4 工程后，参考第四章 4.5 小节讲解的操作步骤，将编译出来的 RPMsg_TEST_CM4.elf 文件传输到开发板的/lib/firmware 目录下。打开 A7 和 M4 端的串口终端（或者称为控制台），如下图 5.4.3.2 所示：

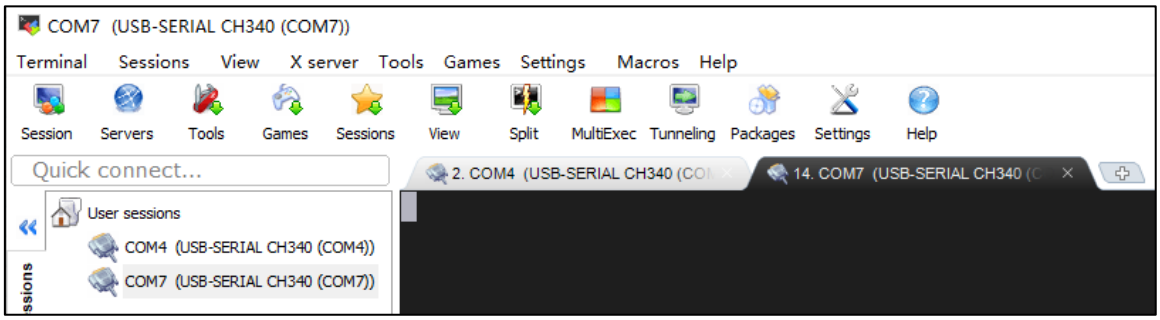


图 5.4.3.2 打开串口终端

在 A7 端的控制台执行如下指令后，可以看到串口打印如下图 5.4.3.3 信息，注意到，打印“virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x0”，表示创建了一个名为 rpmsg-client-sample 通道，0x0 表示 M4 端的 RPMsg 端点地址。在“rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: new channel: 0x400 -> 0x0!”信息中，0x400 是 A7 端 RPMsg 端点的地址，0x0 是 M4 端地址。

```
./test1.sh
```

```
root@ATK-MP157:~# cd /lib/firmware/
root@ATK-MP157:/lib/firmware# ./test1.sh
[  238.103540] remoteproc remoteproc0: powering up m4
[  238.215253] remoteproc remoteproc0: Booting fw image RPMsg_TEST_CM4.elf, size 2237256
[  238.222341]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[  238.234809] virtio_rpmsg_bus virtio0: rpmsg host is online
[  238.235033] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x0
[  238.241315]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[  238.255330] remoteproc remoteproc0: remote processor m4 is now up
root@ATK-MP157:/lib/firmware# [  238.272776] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: new channel: 0x400 -> 0x0!
[  238.438932] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 1 (src: 0x0)
[  238.643921] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 2 (src: 0x0)
[  238.848903] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 3 (src: 0x0)
[  239.053886] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 4 (src: 0x0)
[  239.258895] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 5 (src: 0x0)
[  239.463894] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 6 (src: 0x0)
[  239.668901] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 7 (src: 0x0)
[  239.873900] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 8 (src: 0x0)
[  240.078891] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 9 (src: 0x0)
```

图 5.4.3.3 运行程序

A7 这边打印 100 条信息后，最后打印“goodbye!”，程序即停止打印，如下图 5.4.3.4 所示：

```
[  257.093890] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 92 (src: 0x0)
[  257.298915] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 93 (src: 0x0)
[  257.503897] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 94 (src: 0x0)
[  257.708898] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 95 (src: 0x0)
[  257.913895] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 96 (src: 0x0)
[  258.118900] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 97 (src: 0x0)
[  258.323907] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 98 (src: 0x0)
[  258.528903] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 99 (src: 0x0)
[  258.733906] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 100 (src: 0x0)
[  258.741081] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: goodbye!
```

图 5.4.3.4 A7 打印的数据

在 M4 端的控制台可以看到打印如下图 5.4.3.5 所示信息，也是打印 100 条：

```
M4->A7 96
received_rpmsg=hello world!
M4->A7 97
received_rpmsg=hello world!
M4->A7 98
received_rpmsg=hello world!
M4->A7 99
received_rpmsg=hello world!
M4->A7 100
```

图 5.4.3.5 M4 打印的数据

在 Linux 下再次执行 lsmod 指令，可以看到已经有 rpmsg_client_sample 这个驱动了，如下图 5.4.3.6 所示：

```
[   75.645067] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 99 (src: 0x0)
[   75.850081] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming msg 100 (src: 0x0)
[   75.857175] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: goodbye!

root@ATK-MP157:/lib/firmware# lsmod
Module                  Size  Used by
rpmsg_client_sample    16384  0
ipv6                  442368  32
nf_defrag_ipv6         20480  1 ipv6
8723ds               1363968  0
galcore               323584  6
stm32_dcmi             32768  0
videobuf2_dma_contig   20480  1 stm32_dcmi
videobuf2_memops       16384  1 videobuf2_dma_contig
videobuf2_v4l2         20480  1 stm32_dcmi
videobuf2_common       40960  2 stm32_dcmi,videobuf2_v4l2
ov5640                 28672  0
v4l2_fwnode            20480  2 ov5640,stm32_dcmi
spi_stm32              24576  0
videodev              176128  5 ov5640,v4l2_fwnode,videobuf2_common,stm32_dcmi,videobuf2_v4l2
stm32_cec              16384  0
mc                     36864  5 ov5640,videobuf2_common,videodev,stm32_dcmi,videobuf2_v4l2
ap3216c                16384  0
dht11                  16384  0
ds18b20                16384  0
root@ATK-MP157:/lib/firmware#
```

图 5.4.3.6 加载了 rpmsg_client_sample 驱动

执行如下指令可以卸载 rpmsg_client_sample 驱动，再去执行 lsmod 指令就不会看到 rpmsg_client_sample 驱动了，如下图 5.4.3.7 所示。

```
rmmod rpmsg_client_sample
```

```
root@ATK-MP157:/lib/firmware# rmmod rpmsg_client_sample
[  367.006837] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: rpmsg sample client driver is removed
root@ATK-MP157:/lib/firmware# lsmod
Module                  Size  Used by
ipv6                  442368  32
nf_defrag_ipv6         20480  1 ipv6
8723ds               1363968  0
galcore               323584  6
stm32_dcmi             32768  0
videobuf2_dma_contig   20480  1 stm32_dcmi
videobuf2_memops       16384  1 videobuf2_dma_contig
videobuf2_v4l2         20480  1 stm32_dcmi
videobuf2_common       40960  2 stm32_dcmi,videobuf2_v4l2
ov5640                 28672  0
v4l2_fwnode            20480  2 ov5640,stm32_dcmi
spi_stm32              24576  0
videodev              176128  5 ov5640,v4l2_fwnode,videobuf2_common,stm32_dcmi,videobuf2_v4l2
stm32_cec              16384  0
mc                     36864  5 ov5640,videobuf2_common,videodev,stm32_dcmi,videobuf2_v4l2
ap3216c                16384  0
dht11                  16384  0
ds18b20                16384  0
root@ATK-MP157:/lib/firmware#
```

图 5.4.3.7 卸载 rpmsg_client_sample 驱动

2. 使用屏幕进行测试

如果开发板接了屏幕的话，也可以通过屏幕来测试（开发板需接上配套的屏幕）。注意，如果前面已经启动了 M4，应先关闭 M4 以后再进行如下测试步骤，或者可以重启开发板进行后续的测试。重新启动开发板进入文件系统后，先将 RPMsg_TEST_CM4.elf 文件和实验提供的开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch5\different_core 可执行文件传输到发板的/lib/firmware 目录下，拷贝完成后，记得通过 chmod 指令设置 different_core 文件为可执行权限。如下图 5.4.3.8 所示：

```
chmod a+x different_core
```

```
root@ATK-MP157:/lib/firmware# chmod a+x different_core
root@ATK-MP157:/lib/firmware# ls -l different_core
-rwxr-xr-x 1 root root 1.1M Feb  8  2020 different_core
root@ATK-MP157:/lib/firmware#
```

图 5.4.3.8 拷贝 different_core_com 文件

different_core 文件是一个 Linux QT 编译出来的可执行文件，通过执行该文件可以打开 APP 测试界面，可以通过 APP 来操作实验。

首先，将出厂 Linux 操作系统自带的 APP 桌面关掉，关掉出厂的 APP 界面后，再去执行 different_core 程序打开测试桌面，这样做的目的是避免出厂 Linux 系统的 APP 桌面干扰到 different_core 的显示。如下图 5.4.3.9 所示，点击“设置”：

图 5.4.3.9 点击“设置”

再点击“退出桌面”即可退出出厂的 APP 桌面，退出后屏幕是不再显示其它界面，如下图 5.4.3.10 所示：

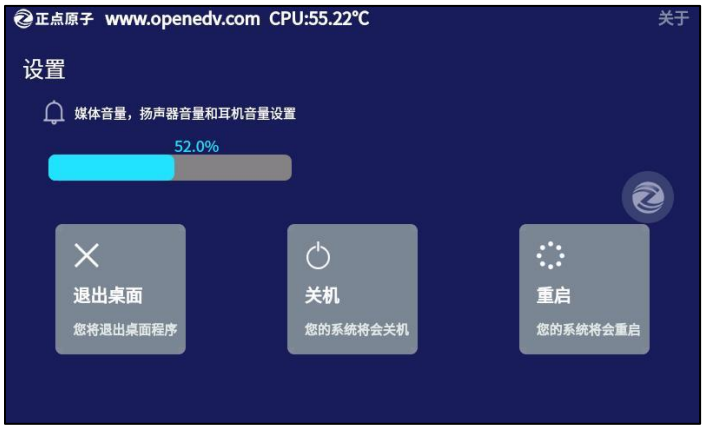


图 5.4.3.10 点击“退出桌面”

退出出厂的 QT 桌面以后，执行如下命令关掉 Wayland，如下图 5.4.3.11 所示，Wayland 是出厂系统自带的一个显示服务器，与 X Window 属于同一级的事物。

```
systemctl stop weston@root.service
```

```
root@ATK-MP157:~# systemctl stop weston@root.service
root@ATK-MP157:~#
```

图 5.4.3.11 关闭 Wayland

执行如下命令可启动测试 APP，如下图 5.4.3.12 所示，可以看到屏幕显示测试 APP 桌面，如下图 5.4.3.13 所示。

```
./different_core RPMsg_TEST_CM4.elf &
```

```
root@ATK-MP157:~# cd /lib/firmware/
root@ATK-MP157:/lib/firmware# ./different_core RPMsg_TEST_CM4.elf &
root@ATK-MP157:/lib/firmware# 固件名字 "RPMsg_TEST_CM4.elf"
```

图 5.4.3.12 运行测试 APP

图 5.4.3.13 测试 APP 桌面

点击屏幕上的“运行固件”按钮即可加载和运行固件，此时 A7 和 M4 的串口打印如前面图 5.4.3.3、图 5.4.3.4 和图 5.4.3.5 所示的信息，说明程序实验运行成功。此时的屏幕如下图 5.4.3.14 所示。

图 5.4.3.14 屏幕显示

点击屏幕上的“卸载固件”按钮即可卸载 M4 固件，A7 这边打印信息如下图 5.4.3.15 所示。

```
root@ATK-MP157:/lib/firmware# [  806.144336] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: rpmsg sampl
e client driver is removed
[  806.656760] remoteproc remoteproc0: warning: remote FW shutdown without ack
[  806.662386] remoteproc remoteproc0: stopped remote processor m4

root@ATK-MP157:/lib/firmware#
```

图 5.4.3.15 卸载固件

该测试 APP 的源代码在开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码目录下，如大家想了解程序的实现方式，可自行研究该代码。

第六章　基于虚拟串口实现异核通信

基于 OpenAMP 库中的 RPMsg，还可以进一步封装，可以抽象出虚拟串口、虚拟 SPI、虚拟 I2C 和虚拟网口等字符设备或块设备。虚拟串口是一个典型的应用，ST 已经提供了实例，本章节，我们基于虚拟串口来实现 Cortex-A7 和 Cortex-M4 之间进行通信。本章分为如下几部分：

6.1 虚拟串口概述
6.2 Linux 下虚拟串口驱动分析
6.3 OpenAMP 库中的 API
6.4 M4 工程下虚拟串口驱动分析
6.5 基于异核通信实现灯光控制
6.6 M4 单次接收 1024B 的数据
6.7 基于异核通信实现阈值报警

6.1 虚拟串口概述

串口是一个字符设备（character device），字符设备是指在 I/O 传输过程中以字符为单位进行传输的设备，我们日常使用的键盘、串口、I2C、SPI 和 LED 等等都是字符设备。

STM32MP157 的 A7 内核和 M4 内核是以共享内存的方式进行通信的，通信的数据放到共享内存中。A7 内核使用 Remoteproc API 控制和管理 M4 内核的生命周期，为 M4 分配系统资源，并创建 Virio 设备。共享内存的数据收发实际上是通过 Virtio 来实现的，Virtio 有两个单向的 vring，分别用于收和发的消息，发送和接收的数据就保存在 Virtio 中的 Vring buffers 中，可以这么说，A7 和 M4 互相通信，实际上就是通过 Virtio 来实现的。

基于 Virtio 框架拓展出了 RPMsg 框架，使用 RPMsg 传递消息的大致操作顺序是：

1. A7 通过 Remoteprco 加载和启动 M4 固件；
2. M4 内核创建端点，并发送服务公告名称；
3. 根据服务公告的名称，A7 内核创建了 RPMsg 的通道，A7 和 M4 可以通过此通道来实现消息传递；
4. A7 和 M4 可通过 RPMsg API 来发送数据，如 rpmsg_send()这个 API；
5. 当有消息接收时，执行 RPMsg 端点绑定的接收回调函数以完成一定的功能；

在调用 RPMsg API 来发送数据时，需要指定 RPMsg 端点、发送的用户数据、数据的长度（字节为单位）以及源地址或者目的地址等，使用起来不是很方便，也不是很灵活，于是，基于 RPMsg 框架，OpenAMP 库提供了虚拟串口的实现方式，即将 RPMsg 再经过一层封装，最后抽象出虚拟串口（virUART）。

其实不仅仅有虚拟串口的这种实现方式，基于 RPMsg 还可以封装为虚拟 SPI、虚拟 I2C 和虚拟网络等等方式，如下图 6.1.1 是基于 RPMsg 封装的虚拟 SPI、虚拟 I2C 和虚拟串口示意图。

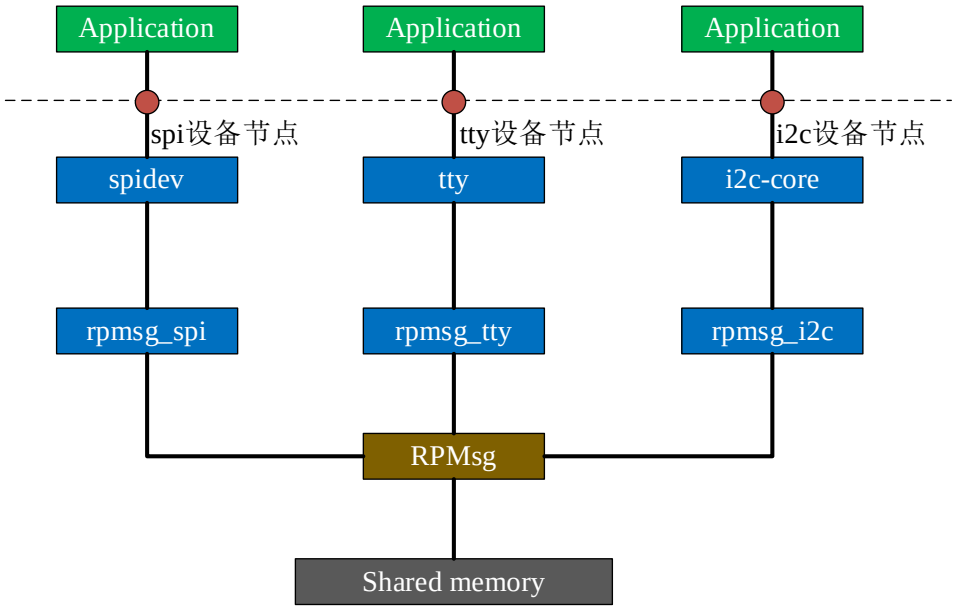


图 6.1.1 基于 RPMsg 的虚拟 SPI、虚拟 I2C 和虚拟串口示意图

本章节，我们重点讲解基于 RPMsg 的虚拟串口的核间通信实现，因为 ST 已经实现了该功能，在 ST 提供的 M4 固件包中已经有示例了，目录为 STM32Cube_FW_MP1_V1.2.0\Projects\STM32MP157C-DK2\Applications\OpenAMP，我们的实验例程也是基于 ST 提供的参考例程来实现的。

前面提到的虚拟串口可以当成普通的串口来使用，通过注册虚拟串口，将复杂的多处理器通信转换为两个串口的通信，使用起来更加灵活和方便。经过层层的封装和抽象，最终的通信方式转化如下图 6.1.2 所示：

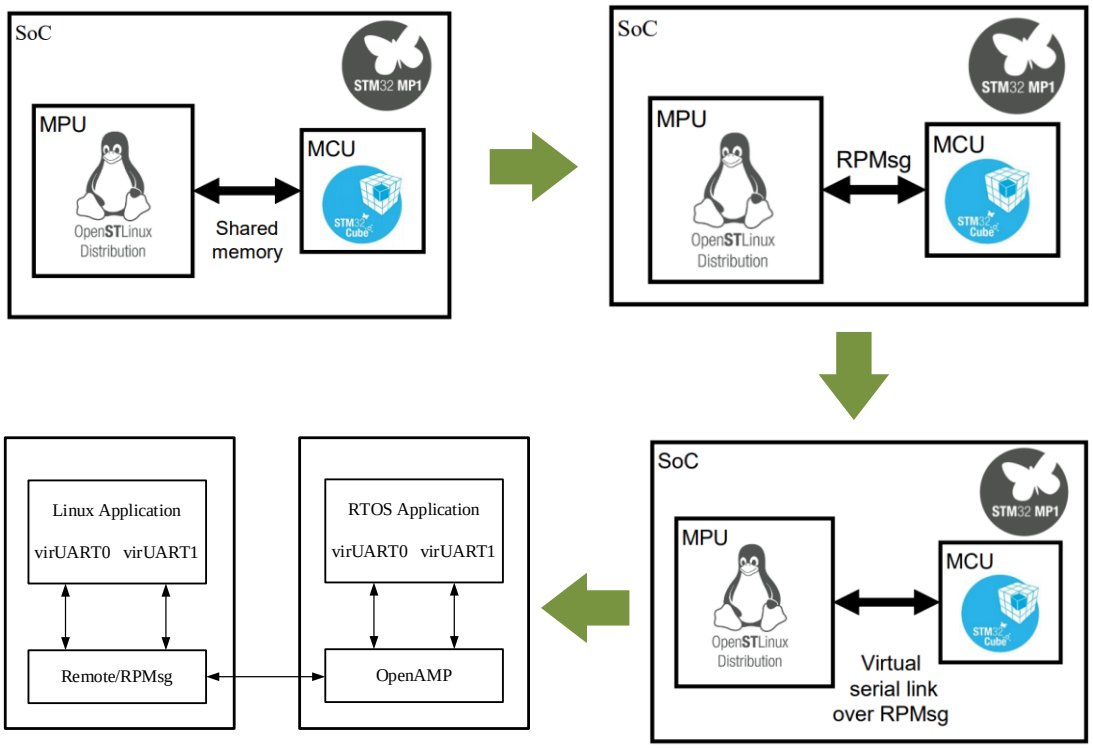


图 6.1.2 通信方式转化关系图

在前面第一章生成的 M4 工程的 RPMsg_TEST\CM4\Middlewares\Third_Party\OpenAMP 目录下，可以看到有 virt_uart.c 文件，此文件是 OpenAMP 库已经封装好的虚拟串口驱动文件，同时，在 Linux 内核源码的 drivers/rpmsg/目录下可以找到 rpmsg_tty.c 文件，通过此文件可以创建一个 ttyRPMSGx 设备（x 可以是 0、1、2 等等），这是一个串行端口终端设备，也就是一个虚拟串口设备，通过虚拟串口设备，A7 可以将数据发给 M4。M4 也可以通过 OpenAMP 库封装的虚拟串口 API 来给 A7 发送数据。

虚拟串口，可以理解为 SOC 内部的一条通信总线，A7 内核和 M4 内核以串口的方式发送和接收数据，发送和接收的数据依然是存放在共享的内存中。既然发送和接收的数据都是在内存中，那么理论上虚拟串口的传输速率会比实际的物理串口速率要快得多。在 Linux 内核下和 STM32Cube 固件包的 OpenAMP 中，ST 已经提供了虚拟串口示例，如下图 6.1.3 所示是虚拟串口实现的框图，下面我们基于此框图来了解虚拟串口的实现过程。

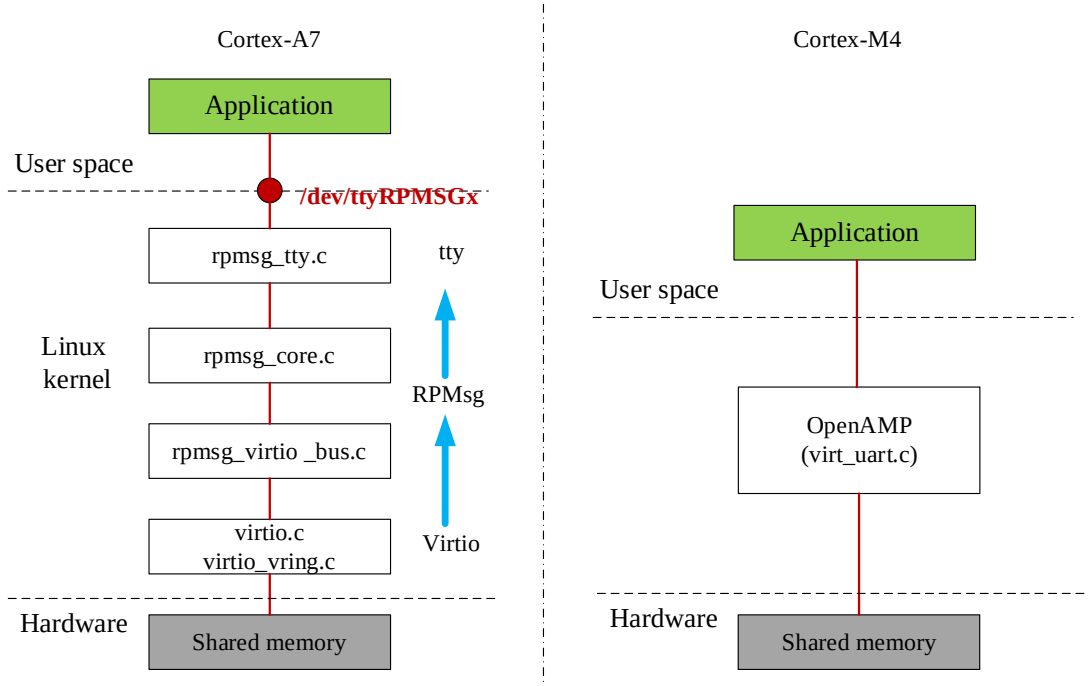


图 6.1.3 基于虚拟串口实现异核通信框图

6.2 Linux 下虚拟串口驱动分析

在 Linux 内核源码的 drivers/rpmsg/rpmsg_tty.c 文件中有和 rpmsg tty 相关的驱动代码，可以找到如下代码：

```
/* rpmsg tty 驱动入口函数，安装 rpmsg tty 驱动程序，驱动的名字是 ttyRPMSG */
static int __init rpmsg_tty_init(void)
{
	int err;

	rpmsg_tty_driver = tty_alloc_driver(MAX_TTY_RPMSG_INDEX, 0);
	if (IS_ERR(rpmsg_tty_driver))
		return PTR_ERR(rpmsg_tty_driver);

	rpmsg_tty_driver->driver_name = "rpmsg_tty";
```

```
        rpmsg_tty_driver->name = "ttyRPMSG";
        rpmsg_tty_driver->major = TTYAUX_MAJOR;
        rpmsg_tty_driver->minor_start = 3;
        rpmsg_tty_driver->type = TTY_DRIVER_TYPE_CONSOLE;
        rpmsg_tty_driver->init_termios = tty_std_termios;
        rpmsg_tty_driver->flags = TTY_DRIVER_REAL_RAW |
                                  TTY_DRIVER_DYNAMIC_DEV;

        tty_set_operations(rpmsg_tty_driver, &rpmsg_tty_ops);

        /* Disable unused mode by default */
        rpmsg_tty_driver->init_termios = tty_std_termios;
        rpmsg_tty_driver->init_termios.c_lflag &= ~(ECHO | ICANON);
        rpmsg_tty_driver->init_termios.c_oflag &= ~(OPOST | ONLCR);

        /* 安装 rpmsg tty 驱动程序 */
        err = tty_register_driver(rpmsg_tty_driver);
        if (err < 0) {
                pr_err("Couldn't install rpmsg tty driver: err %d\n", err);
                goto tty_error;
        }
        /* 在 RPMsg 总线上注册 rpmsg tty 驱动程序 */
        err = register_rpmsg_driver(&rpmsg_tty_rmpsg_drv);

        if (!err)
                return 0;

        tty_unregister_driver(rpmsg_tty_driver);

tty_error:
        put_tty_driver(rpmsg_tty_driver);

        return err;
}
/* 驱动出口函数 */
static void __exit rpmsg_tty_exit(void)
{
  unregister_rpmsg_driver(&rpmsg_tty_rmpsg_drv);
  tty_unregister_driver(rpmsg_tty_driver);
  put_tty_driver(rpmsg_tty_driver);
}

module_init(rpmsg_tty_init);
```

```
module_exit(rpmsg_tty_exit);

MODULE_AUTHOR("Arnaud Pouliquen <arnaud.pouliquen@st.com>");
MODULE_DESCRIPTION("virtio remote processor messaging tty driver");
MODULE_LICENSE("GPL v2");
```

此段代码中的 rpmsg_tty_init()函数负责注册一个 rpmsg_tty_driver 驱动，驱动的名字是 rpmsg_tty，设备名字是 ttyRPMSG，和上一章节讲解的 rpmsg-client-sample 驱动有些不同，本段代码是将驱动 rpmsg_tty.ko 编译进了内核了，可以在开发板的 Linux 操作系统下执行如下的命令查看有哪些和 RPMsg 相关的驱动编译进了内核，如下图 6.2.1 所示：

```
cat /lib/modules/$(uname -r)/modules.builtin | grep rpmsg*
```

```
root@ATK-MP157:~# cat /lib/modules/$(uname -r)/modules.builtin | grep rpmsg*
kernel/drivers/rpmsg/rpmsg_core.ko
kernel/drivers/rpmsg/rpmsg_tty.ko
kernel/drivers/rpmsg/virtio_rpmsg_bus.ko
root@ATK-MP157:~#
```

图 6.2.1 编译进内核的 RPMsg 相关驱动

可以看到有 rpmsg_tty.ko 文件，说明 rpmsg_tty.ko 驱动最终是编译进了内核，在内核启动后此驱动就会被加载了。

rpmsg_driver_tty_id_table[]中的 name 属性为"rpmsg-tty-channel"，名字和 OpenAMP 库中的 virt_uart.c 文件中定义的一样，如下是 M4 工程 OpenAMP 库中 RPMsg_TEST\Middlewares\Third_Party\OpenAMP\virtual_driver\virt_uart.c 文件的部分代码：

```
#define RPMSG_SERVICE_NAME              "rpmsg-tty-channel"

VIRT_UART_StatusTypeDef VIRT_UART_Init(VIRT_UART_HandleTypeDef *huart)
{
  int status;
  /* 为 RPMsg 通信创建端点 */
  status = OPENAMP_create_endpoint(&huart->ept, RPMSG_SERVICE_NAME,
                                   RPMSG_ADDR_ANY,
                                   VIRT_UART_read_cb, NULL);
  if(status < 0) {
    return VIRT_UART_ERROR;
  }

  return VIRT_UART_OK;
}
```

上面的这段代码可以创建 RPMsg 端点（多次调用该函数就可以创建多个端点），端点关联的服务名称是 rpmsg-tty-channel，与前面 Linux 内核源码 drivers/rpmsg/rpmsg_tty.c 下 rpmsg_driver_tty_id_table[]中的 name 属性的名字一样，那么，当加载 M4 固件以后，A7 就会建立一个名字为 rpmsg-tty-channel 的 RPMsg 的通道，A7 和 M4 就可以基于此通道来通信，然后 rpmsg_tty_rmpsg_drv 中的 probe 成员指向的函数 rpmsg_tty_probe()就会被运行，此函数代码如下：

```
static int rpmsg_tty_probe(struct rpmsg_device *rpdev)
{
    struct rpmsg_tty_port *cport, *tmp;
```

```
    unsigned int index;
    struct device *tty_dev;

    cport = devm_kzalloc(&rpdev->dev, sizeof(*cport), GFP_KERNEL);
    if (!cport)
            return -ENOMEM;

    tty_port_init(&cport->port);
    cport->port.ops = &rpmsg_tty_port_ops;
    spin_lock_init(&cport->rx_lock);

    cport->port.low_latency = cport->port.flags | ASYNC_LOW_LATENCY;

    cport->rpdev = rpdev;

    /* get free index */
    mutex_lock(&rpmsg_tty_lock);
    for (index = 0; index < MAX_TTY_RPMSG_INDEX; index++) {
            bool id_found = false;

            list_for_each_entry(tmp, &rpmsg_tty_list, list) {
                    if (index  == tmp->id) {
                            id_found = true;
                            break;
                    }
            }
            if (!id_found)
                    break;
    }
/* 注册 tty 端口，如端口 0 和 1，分别对应的是 ttyRPMSG0 和 ttyRPMSG1 */
tty_dev = tty_port_register_device(&cport->port, rpmsg_tty_driver,
                                   index, &rpdev->dev);
    if (IS_ERR(tty_dev)) {
            dev_err(&rpdev->dev, "failed to register tty port\n");
            tty_port_destroy(&cport->port);
            mutex_unlock(&rpmsg_tty_lock);
            return PTR_ERR(tty_dev);
    }

    cport->id = index;
    list_add_tail(&cport->list, &rpmsg_tty_list);
    mutex_unlock(&rpmsg_tty_lock);
     /* 设置 tty 设备的私有数据 */
```

```
    dev_set_drvdata(&rpdev->dev, cport);

    dev_info(&rpdev->dev, "new channel: 0x%x -> 0x%x : ttyRPMSG%d\n",
             rpdev->src, rpdev->dst, index);

    return 0;
}
```

rpmsg_tty_probe()函数主要是通过 tty_port_register_device()函数来注册 tty 端口，端口的个数和 M4 端初始化的虚拟串口个数有关，例如，假设在 M4 端调用函数 VIRT_UART_Init()初始化了两个虚拟串口，那么就有两个 tty 端口，分别为端口 0 和端口 1，最终 A7 下的 TTY 设备（即虚拟串口）就会有 ttyRPMSG0 和 ttyRPMSG1，那么，A7 就可以通过这两个设备来和 M4 进行串口通信。

6.3 OpenAMP 库中的 API

6.3.1 虚拟串口初始化 API

VIRT_UART_Init()用于初始化虚拟串口，函数原型如下：

```
VIRT_UART_StatusTypeDef VIRT_UART_Init(VIRT_UART_HandleTypeDef *huart)
```

函数 VIRT_UART_Init()的参数如表 6.3.1.1 所示：

参数	描述
huart	虚拟串口设备句柄，VIRT_UART_HandleTypeDef 类型的结构体

表 6.3.1.1 函数 VIRT_UART_Init()相关形参描述

返回值：0 表示成功，1 表示失败。

6.3.2 虚拟串口回调 API

和 RPMsg 类似，用户可自行定义虚拟串口回调函数，用于处理 A7 发送过来的数据，函数名字可自行定义，如下定义一个回调函数，函数的名字是 VIRT_UART0_RxCpltCallback。

```
void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
```

函数 VIRT_UART0_RxCpltCallback()的参数如表 6.3.2 所示：

参数	描述
huart	虚拟串口设备句柄，VIRT_UART_HandleTypeDef 类型的结构体

表 6.3.2 函数 VIRT_UART0_RxCpltCallback()相关形参描述

返回值：无。

6.3.3 注册回调函数

和 RPMsg 创建端点时关联一个回调函数类似，VIRT_UART_RegisterCallback()函数用于给对应的虚拟串口注册回调函数，当串口接收到数据时则执行绑定的回调函数，函数原型如下：

```
VIRT_UART_StatusTypeDef VIRT_UART_RegisterCallback(
                         VIRT_UART_HandleTypeDef *huart,
                        VIRT_UART_CallbackIDTypeDef CallbackID,
                        void(*pCallback)(VIRT_UART_HandleTypeDef*_huart))
```

函数 VIRT_UART_RegisterCallback()的参数如表 6.3.3 所示：

参数	描述
huart	虚拟串口设备句柄，VIRT_UART_HandleTypeDef 类型的结构体
CallbackID	回调 ID，为 VIRT_UART_RXCPLT_CB_ID
pCallback	要注册的回调函数

表 6.3.3 函数 VIRT_UART_RegisterCallback()相关形参描述

返回值：0 表示成功，1 表示失败。

6.3.4 虚拟串口发送 API

VIRT_UART_Transmit()函数用于 M4 通过虚拟串口给 A7 发送数据。函数原型如下：

```
VIRT_UART_StatusTypeDef VIRT_UART_Transmit(
                                    VIRT_UART_HandleTypeDef *huart,
                                    uint8_t *pData, uint16_t Size)
```

函数 VIRT_UART_Transmit()的参数如表 6.3.4 所示：

参数	描述
huart	虚拟串口设备句柄，VIRT_UART_HandleTypeDef 类型的结构体
pData	发送数据的地址
Size	发送数据的长度

表 6.3.4 函数 VIRT_UART_Transmit ()相关形参描述

返回值：0 表示成功，1 表示失败。

6.4 M4 工程下虚拟串口驱动分析

在 M4 工程的 RPMsg_TEST\Middlewares\Third_Party\OpenAMP\virtual_driver 下找到 virt_uart.c 文件，找到如下代码，这段代码我们在前面也有介绍到，函数 VIRT_UART_Init()用于初始化虚拟串口，如下：

```
#define RPMSG_SERVICE_NAME              "rpmsg-tty-channel"
VIRT_UART_StatusTypeDef VIRT_UART_Init(VIRT_UART_HandleTypeDef *huart)
{
  int status;
  /* 创建一个 RPMsg 端点 */
  status = OPENAMP_create_endpoint(&huart->ept, RPMSG_SERVICE_NAME, \
                                   RPMSG_ADDR_ANY,\
                                   VIRT_UART_read_cb, NULL);
  if(status < 0) {
    return VIRT_UART_ERROR;
  }

  return VIRT_UART_OK;
}
```

注意到，创建 RPMsg 端点函数 OPENAMP_create_endpoint()中的服务名称是 rpmsg-tty-channel，和 Linux 下 drivers/rpmsg/rpmsg_tty.c 文件中 rpmsg_driver_tty_id_table[]的 name 属性一样，设备名字和设备驱动名字匹配，内核执行 probe 处理程序，最终 Linux 下创建虚拟串口端口 tty

RPMSGX（X 可以是 0、1、2……），具体有多少个虚拟串口端口，这要看用户通过 VIRT_UART_Init()函数在 M4 下初始化了多少个虚拟串口设备。

6.5 基于异核通信实现灯光控制

6.5.1 硬件设计

1. 例程功能

通过虚拟串口实现 A7 和 M4 互发数据，同时 A7 可控制 M4 开启和关闭开发板的 DS1 灯。

本实验可以基于第一章创建的工程来完成，为了区别不同的工程，我们新建了一个名称为 RPMsg_UART 的工程，工程配置方法可参考第一章。该实验工程参考开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch6\RPMsg_UART。

2. 硬件连接

硬件连接请参考上一章。

6.5.2 软件设计

1. 程序流程图

根据上述例程功能分析，我们得到以下程序流程图，如下图 6.5.2.1 所示，M4 端的 VirtUartx 表示 M4 端初始化的虚拟串口，如 huart0、huart2、huart3……等，ttyRPMSGx 表示 A7 下的虚拟串口设备节点，如 ttyRPMSG0、ttyRPMSG2、ttyRPMSG3……等。

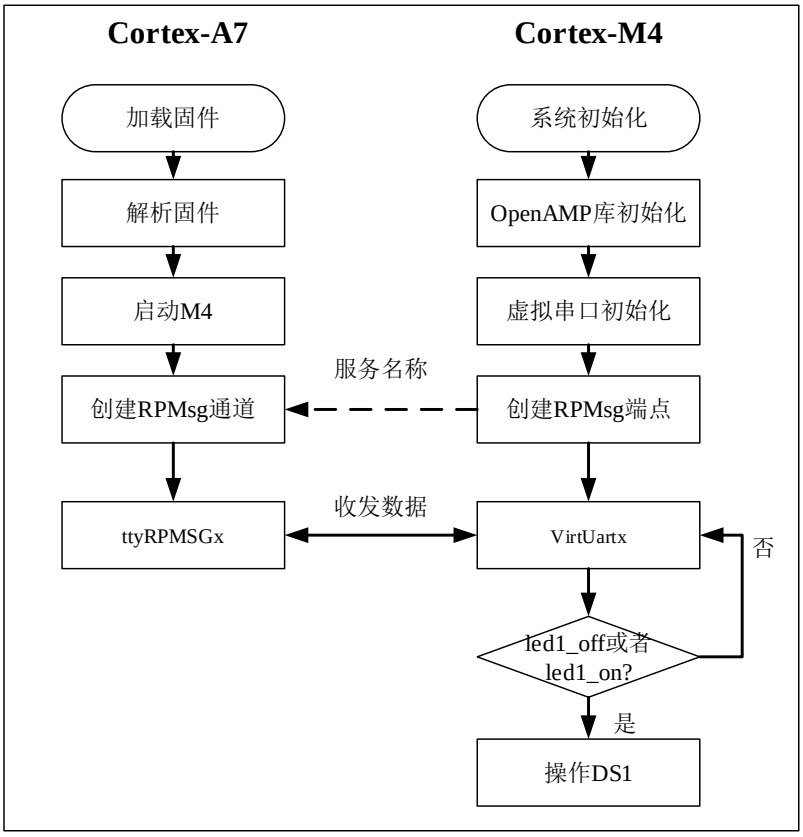


图 6.5.2.1 流程图框图

2. M4 工程下程序解析

Linux 下的驱动分析可参考前面 6.2 小节，下面我们直接在前面创建的 RPMsg_UART 的 M4 工程中添加代码，如下所示，其中成对出现的标红的字体之间的代码（在 USER CODE BEGIN XXX 和 USER CODE END XXX 之间）是我们手动添加的：

```
#include "main.h"
#include "ipcc.h"
#include "openamp.h"
#include "usart.h"
#include "gpio.h"

/* USER CODE BEGIN Includes */
#include "virt_uart.h"                    /* include 虚拟串口头文件 */
/* USER CODE END Includes */

/* USER CODE BEGIN PTD */
#define LED1_ON  "led1_on"
#define LED1_OFF "led1_off"
#define MAX_BUFFER_SIZE RPMSG_BUFFER_SIZE   /* 最大拷贝的字节数 */

uint8_t Copy_Buffer[MAX_BUFFER_SIZE];       /* 拷贝缓冲区 */
uint8_t BuffTx[MAX_BUFFER_SIZE];       /* 发送缓冲区 */

VIRT_UART_HandleTypeDef huart0;        /* 虚拟串口 0 */
uint16_t RxSize = 0;                    /* 拷贝的字节个数 */
__IO FlagStatus flag = RESET;           /* M4 接收数据标志位 */
/* 声明虚拟串口关联的接收回调函数 */
void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);

/* 支持 UART3 通过通过 printf()函数打印信息 */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif
PUTCHAR_PROTOTYPE
{
 /* 本实验使用的是 UART3，如果使用的是其它串口，则将 UART3 改为对应的串口即可 */
  while ((USART3->ISR & 0X40) == 0);/* 等待上一个字符发送完成 */
  USART3->TDR = (uint8_t) ch;/* 将要发送的字符 ch 写入到 TDR 寄存器 */
  return ch;                    /* 返回要发送的字符 */
}
```

```
38  /* USER CODE END PTD */
39  /* 声明配置时钟函数 */
40  void SystemClock_Config(void);
41
42  int main(void)
43  {
44    HAL_Init();                     /* 初始化 HAL 库 */
45    if(IS_ENGINEERING_BOOT_MODE())/* 检查平台是否为工程启动模式 */
46    {
47      /* 配置系统时钟 */
48      SystemClock_Config();
49    }
50    MX_IPCC_Init();                       /* 初始化 IPCC */
51    MX_OPENAMP_Init(RPMSG_REMOTE, NULL);/* 初始化 OpenAMP 库 */
52    MX_GPIO_Init();                        /* 初始化 GPIO */
53    MX_USART3_UART_Init();                 /* 初始化 UART3 */
54
55    /* USER CODE BEGIN 2 */
56    printf("****** Start Initialize Virtual UART0 ******\r\n");
57    if (VIRT_UART_Init(&huart0) != VIRT_UART_OK)/* 初始化虚拟串口 */
58    {
59      printf("****** VIRT_UART_Init UART0 failed. ******\r\n");
60      Error_Handler();
61     }
62    /* 给虚拟串口 huart0 注册回调函数 VIRT_UART0_RxCpltCallback() */
63    if(VIRT_UART_RegisterCallback(&huart0, VIRT_UART_RXCPLT_CB_ID,
                          VIRT_UART0_RxCpltCallback) != VIRT_UART_OK)
64    {
65      Error_Handler();
66    }
67    /* USER CODE END 2 */
68
69    while (1)
70    {
71      /* USER CODE BEGIN 3 */
72      OPENAMP_check_for_message();/* 邮箱轮询，检查 A7 是否有发来数据 */
73      if (flag==SET)                /* M4 接收到 A7 发送过来的数据 */
74      {
75        /* 接收标志位复位，以便下次判断通道是否再次接收到数据 */
76         flag = RESET;
77        /*
78         * 比较 Copy_Buffer 和 LED1_ON，如果相等，则:
79         * 1、给 A7 发送字符串"M4:LED1_ON"
```

```
80          * 2、点亮 LED1
81          */
82         if (!strncmp((char *)Copy_Buffer, LED1_ON, strlen(LED1_ON)))
83         {
84           strcpy((char *)BuffTx, "M4:LED1_ON\n");
85           VIRT_UART_Transmit(&huart0, BuffTx, strlen((const char
                                                        *)BuffTx));
86           HAL_GPIO_WritePin(GPIOF, GPIO_PIN_3, GPIO_PIN_RESET);
87         }
88         /*
89          * 比较 Copy_Buffer 和 LED1_OFF，如果相等，则:
90          * 1、给 A7 发送字符串"M4:LED1_OFF"
91          * 2、关闭 LED1
92          */
93         if (!strncmp((char *)Copy_Buffer, LED1_OFF, strlen(LED1_OFF)))
94         {
95           strcpy((char *)BuffTx, "M4:LED1_OFF\n");
96           VIRT_UART_Transmit(&huart0, BuffTx, strlen((const char
                                                          *)BuffTx));
97           HAL_GPIO_WritePin(GPIOF, GPIO_PIN_3, GPIO_PIN_SET);
98         }
99
100        memset(Copy_Buffer, 0 ,RxSize);     /* 清空 Copy_Buffer */
101        memset(BuffTx,0,strlen((const char *)BuffTx));/* 清空 BuffTx */
102      }
103    }
104    /* USER CODE END 3 */
105  }
106
107  /**
108    * @brief 配置系统时钟
109    * @retval None
110    */
111  void SystemClock_Config(void)
112  {
113    /* 省略此处代码 */
114  }
115
116  /* USER CODE BEGIN 4 */
117  void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart)
118  {
119    /* M4 打印接收到的数据以及数据的长度 */
120    printf("\n\rReceived on Virtual UART0:\n\r%s\n\rSize:%d\n\r",
```

```
                        (char *) huart->pRxBuffPtr,huart->RxXferSize);
121   /*
122    * 取 RxSize 值:
123    * 1、若接收到的数据长度小于 MAX_BUFFER_SIZE，则 RxSize=实际接收到的数据长度
124    * 2、若接收到的数据长度大于 MAX_BUFFER_SIZE，则则 RxSize=MAX_BUFFER_SIZE
125    */
126   RxSize = huart->RxXferSize < MAX_BUFFER_SIZE? huart->RxXferSize :
                                    MAX_BUFFER_SIZE-1;
127   /* 将接收到长度为 RxSize 的数据拷贝到 Copy_Buffer 中 */
128   memcpy(Copy_Buffer, huart->pRxBuffPtr, RxSize);
129   flag = SET; /* 设置标志位 flag=SET，表示 M4 接收到了数据 */
130 }
131 /* USER CODE END 4 */
132
133 /**
134   * @brief  此功能在发生错误时执行
135   * @retval None
136   */
137 void Error_Handler(void)
138 {
139
140 }
```

对以上代码进行分析：

第 8 行，我们要用到虚拟串口，此处需要 include 虚拟串口头文件 virt_uart.h。

第 14 行，设置一个缓冲区，将 M4 从虚拟串口中接收到的数据拷贝到该缓冲区中（我们称该缓冲区为拷贝缓冲区），此行是设置拷贝缓冲区的大小为 RPMSG_BUFFER_SIZE 字节，RPMSG_BUFFER_SIZE 的大小为 512。

第 16~第 17 行，Copy_Buffer[]是我们定义的拷贝缓冲区，BuffTx[]是我们定义的发送缓冲区，M4 可以将发送缓冲区的 BuffTx[]数据发给 A7。

第 19 行，定义一个虚拟串口句柄 huart0，我们只用到一个虚拟串口，如果用到多个，可以定义多个虚拟串口句柄，如：

```
VIRT_UART_HandleTypeDef huart1;
```

第 21 行，定义 M4 接收数据标志位，并初始化为 RESET。

第 23 行，声明虚拟串口关联的接收回调函数，当虚拟串口接收到数据时，绑定的该回调函数被执行。

第 25~第 37 行，通过重映射的方式将 printf()函数重映射到 STM32 串口的寄存器上，那么串口可通过 printf()输出流，最终 UART3 可以打印信息。

第 56~第 61 行，打印“****** Start Initialize Virtual UART0 ******”提示开始初始化虚拟串口，通过调用 VIRT_UART_Init()函数完成对虚拟串口 huart0 的初始化，如果初始化失败，则打印提示信息“****** VIRT_UART_Init UART0 failed. ******”。

第 63~第 66 行，给虚拟串口 huart0 注册回调函数 VIRT_UART0_RxCpltCallback()，一般初始化了多少个虚拟串口就注册多少个虚拟串口回调函数。回调函数在第 117~第 130 行有定义，当 M4 从虚拟串口接收到数据时，该回调函数被执行，我们看该回调函数做了哪些操作：

第 120 行，M4 将接收到的数据以及数据的长度打印出来。

第 126 行，计算 RxSize 的值，MAX_BUFFER_SIZE 默认为 512（用户可修改为其它值）当 M4 接收到的数据小于 512 时，RxSize 取 huart->RxXferSize（表示实际接收到的数据长度），当 M4 接收到的数据大于 512 时，RxSize 取 511。

第 128 行，将 M4 从虚拟串口接收到的数据拷贝到拷贝缓冲区 Copy_Buffer 中，拷贝的长度是 RxSize。

第 129 行，每当 M4 从虚拟串口接收到数据的时候，flag 标志位都被设置为 SET。

我们来看 while 循环做的操作：

第 72 行，通过邮箱轮询操作，检查 A7 是否有发来数据了，当有数据发来的时候，M4 端虚拟串口绑定的接收回调函数 VIRT_UART0_RxCpltCallback()就被执行，标志位 flag 被置为 SET，接着第 74~102 行的代码被执行。

第 76 行，在进行数据处理前，先将接收标志位 flag 设置为 RESET。

第 82~第 87 行，对比拷贝缓冲区 Copy_Buffer 的字符串是否等于字符串"led1_on"，若相等，则 M4 将字符串“M4:LED1_ON”发给 A7，同时 M4 点亮开发板的 LED1 灯，也就是说：如果 A7 给 M4 发送了字符串"led1_on"，M4 点亮 LED1。

第 93~第 98 行，同理，若 A7 给 M4 发送了字符串"led1_on"，则 M4 将字符串“M4:LED1_ON”发给 A7，同时 M4 点亮开发板的 LED1 灯。

第 100 和第 101 行，每次处理数据后都要清空缓冲区 Copy_Buffer 和 BuffTx，否则这些缓冲区会保留上次接收到的数据。

6.5.3 实验验证

1. 不使用屏幕进行测试

编译 M4 工程后，参考第四章 4.5 小节讲解的操作步骤，将编译出来的 RPMsg_UART_CM4.elf 文件传输到开发板的/lib/firmware 目录下，新建一个 test3.sh 脚本文件，内容如下：

```
#!/bin/sh

echo RPMsg_UART_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware

echo start >/sys/class/remoteproc/remoteproc0/state

echo "OK" > /dev/ttyRPMSG0
```

即运行 test3.sh 脚本后，加载和运行固件 RPMsg_UART_CM4.elf，同时 A7 通过虚拟串口/dev/ttyRPMSG0 给 M4 发送字符串“OK”，此处的操作非常关键：

在第五章我们也多次强调，A7 和 M4 之间通过 RPMsg 进行核间通信时，需要 A7 发送第一条消息，RPMsg 通道才会被激活，通道激活以后，M4 才能给 A7 发送数据。本章 Linux 下 drivers/rpmsg/rpmsg_tty.c 的驱动和第五章的 samples/rpmsg/rpmsg_client_sample.c 的驱动不同，在第五章中，A7 会给 M4 发送第一条消息"hello world!"，所以通道被激活，在本章中，Linux 下的驱动没有给给 M4 发送第一条消息，驱动下没完成的事我们就在应用下完成，也就是在以上脚本中添加 echo "OK" > /dev/ttyRPMSG0 这句，当运行 test3.sh 脚本时，加载和运行固件后 A7 就立马给 M4 发送第一条消息，RPMsg 通道就被激活了。

修改好 test3.sh 以后，设置该脚本为可执行权限：

```
root@ATK-MP157:/lib/firmware# chmod a+x test3.sh
root@ATK-MP157:/lib/firmware# ls -l test3.sh
-rwxr-xr-x 1 root root 160 Feb  8 00:01 test3.sh
root@ATK-MP157:/lib/firmware#
```

图 6.5.3.1 设置 test3.sh 文件为可执行权限

同时打开 A7 和 M4 端的串口终端，如下图 6.5.3.2 所示：

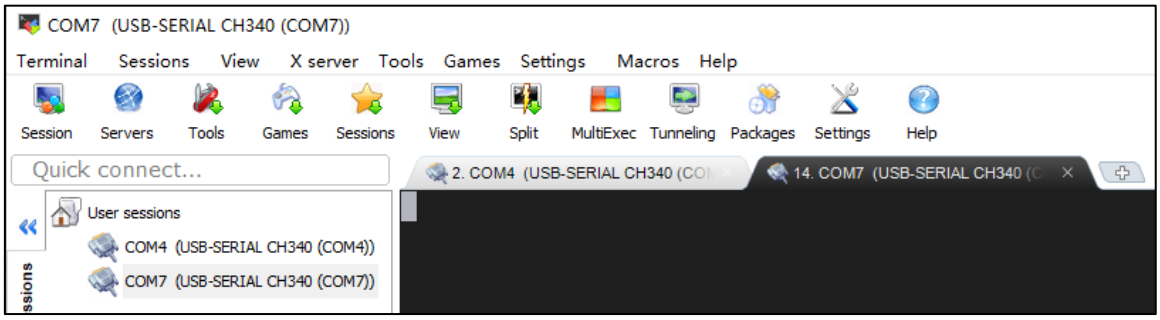


图 6.5.3.2 打开串口终端

运行以上脚本，如下图 6.5.3.3 所示：

```
./test3.sh
ls /dev/ttyR*
```

运行固件后，打印“rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0”，即 A7 下创建了虚拟串口设备 ttyRPMSG0。

```
root@ATK-MP157:/lib/firmware# ./test3.sh
[   53.791975] remoteproc remoteproc0: powering up m4
[   53.896618] remoteproc remoteproc0: Booting fw image RPMsg_UART_CM4.elf, size 2249652
[   53.903460]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[   53.914849] virtio_rpmsg_bus virtio0: rpmsg host is online
[   53.920149] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[   53.930461]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[   53.935647] remoteproc remoteproc0: remote processor m4 is now up
[   53.945233] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
root@ATK-MP157:/lib/firmware# ls /dev/ttyR*
/dev/ttyRPMSG0
root@ATK-MP157:/lib/firmware#
```

图 6.5.3.3 成功加载和运行 M4 固件

查看 M4 端的打印信息，如下图 6.5.3.4 所示，M4 接收到字符串“OK”，共接收到 3 个字节（最后一个字节是换行符）：

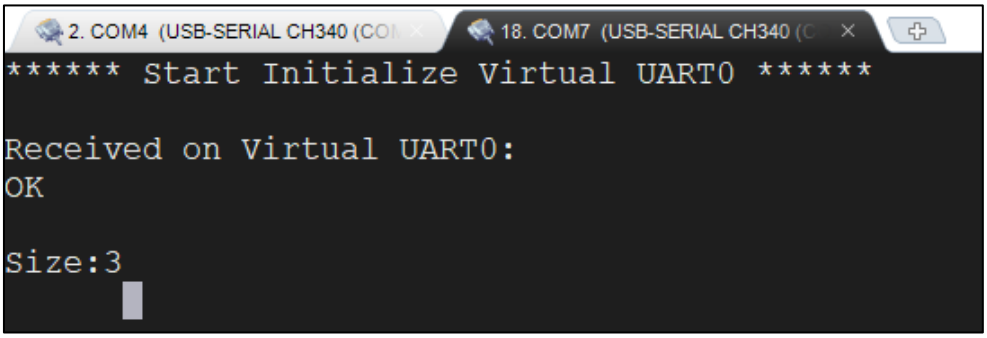


图 6.5.3.4 M4 端打印信息

下面我们给 M4 发送 512 字节的数据，观察 M4 端接收的情况，如下图 6.5.3.5 所示，第 512 个字节的数据是字符“A”，加上换行符，那么 A7 给 M4 发送了 513 个字节的数据：

```
echo
"12345678901234567890123456789012345678901234567890123456789012345678901
12345678901234567890123456789012345678901234567890123456789012345678901
23456789012345678901234567890123456789012345678901234567890123456789012
34567890123456789012345678901234567890123456789012345678901234567890123
45678901234567890123456789012345678901234567890123456789012345678901234
56789012345678901234567890123456789012345678901234567890123456789012345
67890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901A" > /dev/ttyRPMSG0
```

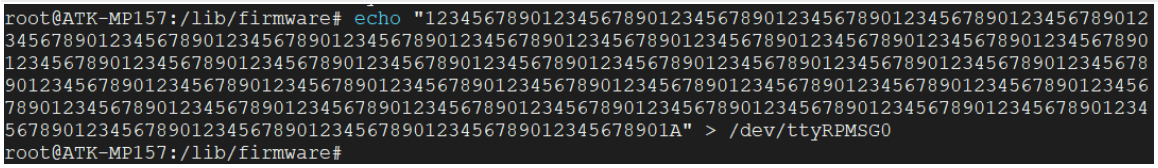


图 6.5.3.5 A7 给 M4 发送 513 字节数据

如下图 6.5.3.6 所示，M4 分两次接收数据，第一次接收 496 个字节，第二次接收 17 个字节，496+17=513 个字节（最后一个字节是换行符），由于串口终端打印的字节数有限，所以没有把所有接收到的字符都打印出来。

在第五章我们分析过，在 RPMsg 的消息格式中，前面 16 个字节是消息头，消息头之后才是用户真正发送的数据，在 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的宏 MAX_RPMSG_BUF_SIZE 默认为 512，所以基于 RPMsg 的虚拟串口，单次最大只能接收 512-16=496 字节的用户数据。如下的程序运行结果和前面第五章分析的一致！

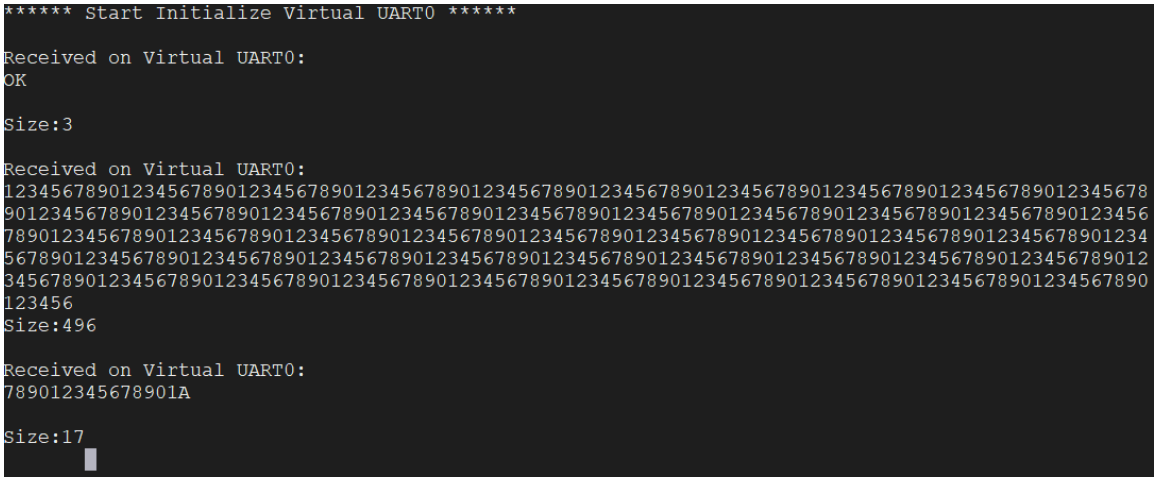


图 6.5.3.6 M4 分次接收数据

如何实现 M4 单次接收 512 字节或者更大的数据？我们在 5.3 小节有进行讲解，关于测试，请看 6.6 小节。

在以上操作中，通过虚拟串口，A7 成功给 M4 发送数据，下面我们验证 M4 给 A7 发送数据。在 A7 的串口终端执行如下命令后，A7 就把 M4 发来的数据回显打印出来，如下图 6.5.3.7 所示：

```
cat /dev/ttyRPMSG0 &
```

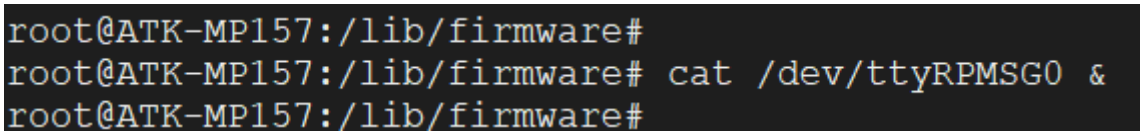


图 6.5.3.7 读取/dev/ttyRPMSG0

在 A7 端分别执行如下命令，当 A7 发送“led1_off”给 M4 以后，M4 关闭开发板的 DS1 灯，同时 M4 给 A7 发送“M4:LED1_OFF”；当 A7 给 M4 发送“led1_on”后，M4 开启 DS1 灯，同时 M4 给 A7 发送“M4:LED1_ON”。A7 串口打印如下图 6.5.3.8 所示，M4 串口打印如下图 6.5.3.9 所示。

```
root@ATK-MP157:/lib/firmware# cat /dev/ttyRPMSG0 &
root@ATK-MP157:/lib/firmware#
root@ATK-MP157:/lib/firmware# echo "led1_off" > /dev/ttyRPMSG0
root@ATK-MP157:/lib/firmware# M4:LED1_OFF

root@ATK-MP157:/lib/firmware# echo "led1_on" > /dev/ttyRPMSG0
root@ATK-MP157:/lib/firmware# M4:LED1_ON

root@ATK-MP157:/lib/firmware#
```

图 6.5.3.8 A7 串口打印

```
Received on Virtual UART0:
led1_off

Size:9

Received on Virtual UART0:
led1_on

Size:8
```

图 6.5.3.9 M4 端串口打印

2. 使用屏幕进行测试

使用屏幕测试的操作步骤和 5.4 小节的一样，先将 M4 固件 RPMsg_UART_CM4.elf 拷贝到开发板的/lib/firmware 目录下，然后先关闭出厂 Linux 操作系统的 APP 桌面，同时再 A7 端执行如下命令关闭 Wayland，如下图 6.5.3.10 所示。

```
systemctl stop weston@root.service
```

```
root@ATK-MP157:~# systemctl stop weston@root.service
root@ATK-MP157:~#
```

图 6.5.3.10 关闭 Wayland

注意，如果前面已经启动了 M4，应先关闭 M4 以后再进行如下测试步骤，将前面执行的 cat /dev/ttyRPMSG0 &命令对应的进程 kill 掉，或者可以重启开发板后再进行后续的测试。执行如下指令运行测试 APP 桌面，如下图 6.5.3.11 所示：

```
./different_core RPMsg_UART_CM4.elf &
```

```
root@ATK-MP157:/lib/firmware# ./different_core RPMsg_UART_CM4.elf &
root@ATK-MP157:/lib/firmware# 固件名字 "RPMsg_UART_CM4.elf"
```

图 6.5.3.11 运行 APP 测试程序

屏幕显示如下图 6.5.3.12 所示，点击屏幕上的“运行固件”

图 6.5.3.12 测试 APP 桌面

点击屏幕上的“运行固件”按钮即可加载和运行固件，此时 A7 打印信息如下图 6.5.3.13 所示：

```
root@ATK-MP157:/lib/firmware# ./different_core RPMsg_UART_CM4.elf &
root@ATK-MP157:/lib/firmware# 固件名字 "RPMsg_UART_CM4.elf"

[    92.820648] remoteproc remoteproc0: powering up m4
[    92.936052] remoteproc remoteproc0: Booting fw image RPMsg_UART_CM4.elf, size 2249652
[    92.943398]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[    92.956413] virtio_rpmsg_bus virtio0: rpmsg host is online
[    92.960690] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[    92.967988]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[    92.968005] remoteproc remoteproc0: remote processor m4 is now up
[    92.988043] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
```

图 6.5.3.13 A7 串口打印信息

M4 串口打印如下图 6.5.3.14 所示，即点击“运行固件”按钮后，A7 会加载、运行 M4 固件，并给 M4 发送第一条消息，即发送“start”字符串，RPMsg 通道被激活。

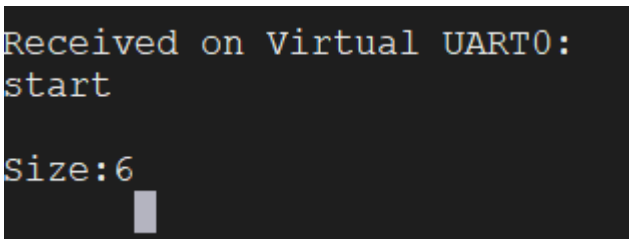


图 6.5.3.14 M4 端打印信息

此时的屏幕如下图 6.5.3.15 所示，串口号默认选择 ttyRPMSG0：

图 6.5.3.15 屏幕显示

依次点击屏幕上的“打开串口”和“发送”按钮，A7 会将屏幕中的 https://www.openedv.com 发送给 M4，M4 端打印如下图 6.5.3.16 所示：

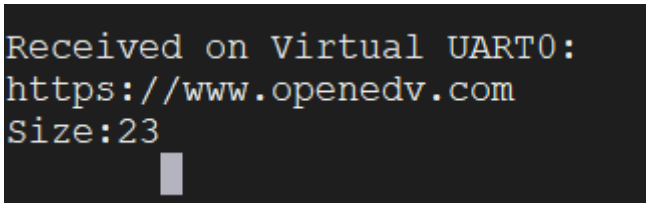


图 6.5.3.16 M4 端接收到的数据

也可以在开发板的 USB 接口接上鼠标和键盘，然后在屏幕上输入其它字符以后再发送给 M4，例如可以输入“led1_off”后再发送，如下图 6.5.3.17 所示，由于篇幅问题，此处笔者就不再进行演示了。

图 6.5.3.17 手动输入字符

再点击“卸载固件”按钮即可关闭固件，实际上 A7 执行了如下指令：

```
/* 关闭，停止固件运行 */
echo stop >/sys/class/remoteproc/remoteproc0/state
```

以上使用的是第五章的 APP 测试程序，如果小伙伴掌握了 Linux QT，也可以自己设计一个功能更加全面、更加美观的 APP 测试桌面，如入门 Linux QT，也可以参考《【正点原子】STM32MP157 嵌入式 Qt 开发指南》。

6.6 M4 单次接收 1024B 的数据

6.6.1 修改代码

若要实现 M4 单次接收 1024B 的数据，则需要修改 Linux 内核源码 drivers/rpmsg/virtio_rpmsg_bus.c 下的宏 MAX_RPMSG_BUF_SIZE、设备树 arch/arm/boot/dts/stm32mp157d-atk.dtsi、M4 工程下的链接脚本 STM32MP157DAAX_RAM.ld，具体修改和操作方法请参考前面 5.3.3 小节，下面我们直接进行测试。

本实验基于前面的 M4 工程 RPMsg_UART_CM4 来进行测试。

6.6.2 编译

编译内核、设备树、内核模块和 M4 工程，将内核 uImage、设备树 stm32mp157d-atk.dtb 拷贝到开发板的/boot 目录下，如果此前已经按照前面的步骤编译和安装过内核模块，则此次可以不用再重新将内核模块安装到开发板中，将新编译出来的 M4 固件 RPMsg_UART_CM4.elf 拷贝到开发板的/lib/firmware 目录下，重启开发板，目的就是使得更新的内核和设备树生效。

查看内核版本和内核模块版本，如下图 6.6.2.1 所示，都是笔者自己编译的：

```
root@ATK-MP157:~# uname -r
5.4.31
root@ATK-MP157:~# cd /lib/modules/
root@ATK-MP157:/lib/modules# ls
5.4.31   modules.tar.bz2
root@ATK-MP157:/lib/modules#
```

图 6.6.2.1 查看内核和内核模块版本

6.6.3 测试

在开发板的/lib/firmware 目录下新建一个 echo.sh 脚本，脚本中实现 A7 给 M4 发送 1024 个字节，其中第 1024 个字节是字符 B，如下图 6.6.3.1 所示。

```
root@ATK-MP157:/lib/firmware# cat echo.sh
echo "12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
34567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
90123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
567890123456789012345678901A1234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
90123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
56789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
345678901234567890123456789012345678901234567890 1B" > /dev/ttyRPMSG0
root@ATK-MP157:/lib/firmware#
```

图 6.6.3.1 查看 echo.sh 脚本的内容

然后在 A7 端执行如下命令进行加载和运行 M4 固件 RPMsg_UART_CM4.elf：

```
./test3.sh
```

如下图 6.6.3.2 所示，打印 len=men=size=4096，vring 的个数位 16（收发各 16 个），同时成功初始化、创建了虚拟串口设备 ttyRPMSG0：

```
root@ATK-MP157:/lib/firmware# ./test3.sh
[  297.496854] remoteproc remoteproc0: powering up m4
[  297.607059] remoteproc remoteproc0: Booting fw image RPMsg_UART_CM4.elf, size 2249652
[  297.613688] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[  297.622297] **************************size 4096
[  297.622312] ************************************************len 4096
[  297.625390] ************************************************men 4096
[  297.633531] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[  297.642714] **************************size 4096
[  297.642730] ************************************************len 4096
[  297.647324] ************************************************men 4096
[  297.653485]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[  297.668358] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[  297.672490] wwwwwwwwwwwwwwwwwwwwwwwwwwwwww16, 438
[  297.678787] virtio_rpmsg_bus virtio0: rpmsg host is online
[  297.682899] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[  297.694348]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[  297.700592] remoteproc remoteproc0: remote processor m4 is now up
[  297.716853] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
root@ATK-MP157:/lib/firmware#
```

图 6.6.3.2 加载和运行固件

M4 端打印如下图 6.6.3.3 所示信息：

```
       ****** Start Initialize Virtual UART0 ******

Received on Virtual UART0:
OK

Size:3
```

图 6.6.3.3 M4 打印的信息

再 A7 端执行如下命令，直接运行 echo.sh 脚本，即 A7 给 M4 端发送 1024 个字节的数据，如下图 6.6.3.4 所示：

```
./echo.sh
```

```
root@ATK-MP157:/lib/firmware#
root@ATK-MP157:/lib/firmware# ./echo.sh
root@ATK-MP157:/lib/firmware#
```

图 6.6.3.4 A7 给 M4 发送 1024 字节数据

M4 端打印信息如下图 6.6.3.5 所示，可以看到 M4 端第一次接收到 1008 个字节的数据，第二次接收到 17 个字节的数据，1008+17=1025（最后一个字节是换行符），因为在 RPMsg 的消息格式中，前 16 个字节是消息头，1008+16=1024，所以经过以上的修改，M4 一次能接收 1024 字节的数据，在这 1024 字节的数据中，1008 个字节的数据是用户真正发送的数据。

```
Received on Virtual UART0:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
5678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
5678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
9012345678901234567890123456
Size:1008

Received on Virtual UART0:
789012345678901B

Size:17
```

图 6.6.3.5 M4 打印的数据

经过以上测试，可以验证我们在 5.3 小节的分析，实验验证成功。如果想修改 vring 个数的小伙伴，也可以参考前面 5.3 小节的分析来进行修改。

6.7 基于异核通信实现阈值报警

6.7.1 硬件设计

1. 例程功能

M4 控制 ADC1 通道 19（IO 口为 PA5）采集数据，并通过虚拟串口将采集到的数据发给 A7，A7 在屏幕中显示 ADC 的数值，当 A7 检测到 ADC 的值超过 2.5V 时，A7 开启蜂鸣器，蜂鸣器发起“嘀嘀嘀……”告警声。该实验工程参考开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码\CubeIDE\ch6\RPMsg_UART_ADC。

2. 硬件连接

硬件连接请参考第四或第五章，此外，本章要用到 ADC1 通道 19，ADC1 通道 19 接的是 PA5 这个 IO 口，开发板上有引出 PA5 引脚，先使用跳线帽将 JP2 排针的 ADC1 和电位器的 RP_AD 连接，这样 PA5 就连接到电位器 VR1 上了，电位器上接的是 3.3V，用户可以通过调节电位器的旋钮改变接入到 PA5 上的电压值为 0~3.3V（实际上就是通过改变电阻来改变电压）。实验前要记住检查跳线帽是否有接好，如下图 6.7.1.1 所示：

图 6.7.1.1 硬件部分

6.7.2 软件设计

1. 程序流程图

根据上述例程功能分析，我们得到以下程序流程图，如下图 6.7.2.1 所示，M4 端的 VirtUartx 表示 M4 端初始化的虚拟串口，如 huart0、huart2、huart3……等，ttyRPMSGx 表示 A7 下的虚拟串口设备节点，如 ttyRPMSG0、ttyRPMSG2、ttyRPMSG3……等。

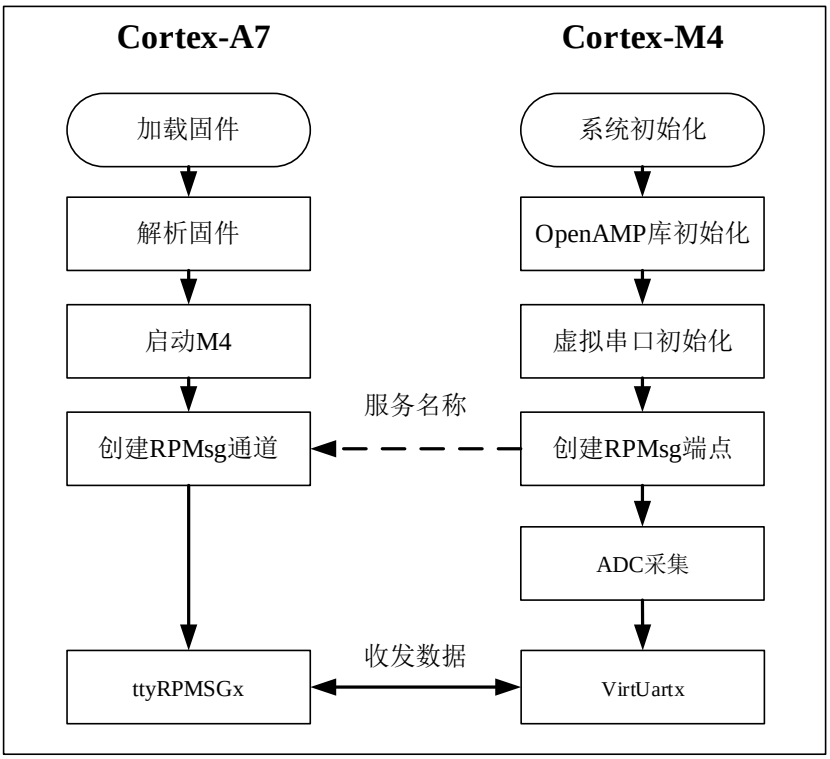


图 6.7.2.1 流程框图

2. 配置工程

请参考前面实验工程的配置，此外，我们要用到 ADC1，需要再配置 ADC1，配置步骤如下：

ADC1 通道 19 占用的 IO 口是 PA5，如下图 6.7.2.2 所示，配置 PA5 复用为 ADC1_ INP19：

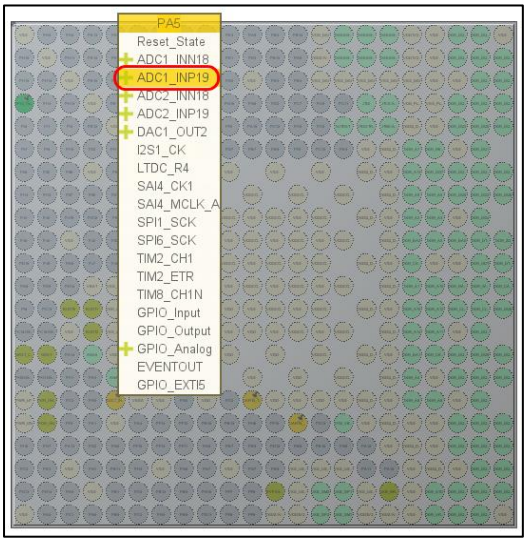


图 6.7.2.2 配置 PA5 引脚

接下来配置 ADC1 的模式，本实验用到 ADC1 的通道 19 的单端输入，所以勾选 IN19 Single-ended，我们没有使用硬件触发，使用的是软件触发，所以 EXTI Conversion Trigger 选项要 Disable，如下图 6.7.2.3 所示：

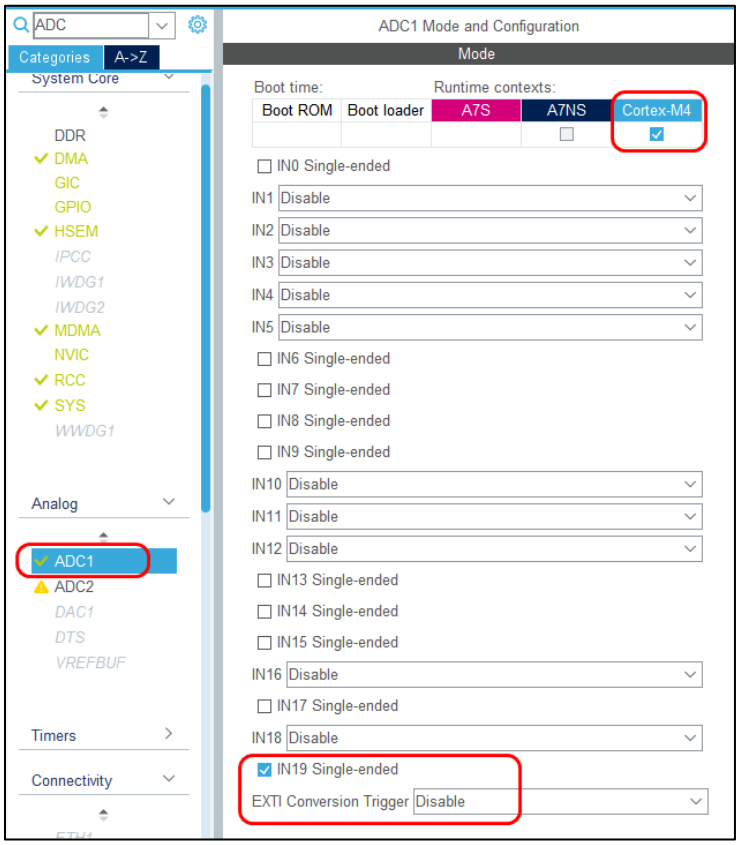


图 6.7.2.3 ADC1 模式配置

按照如下配置 ADC1 通道 19 的参数，如下图 6.7.2.4 所示：

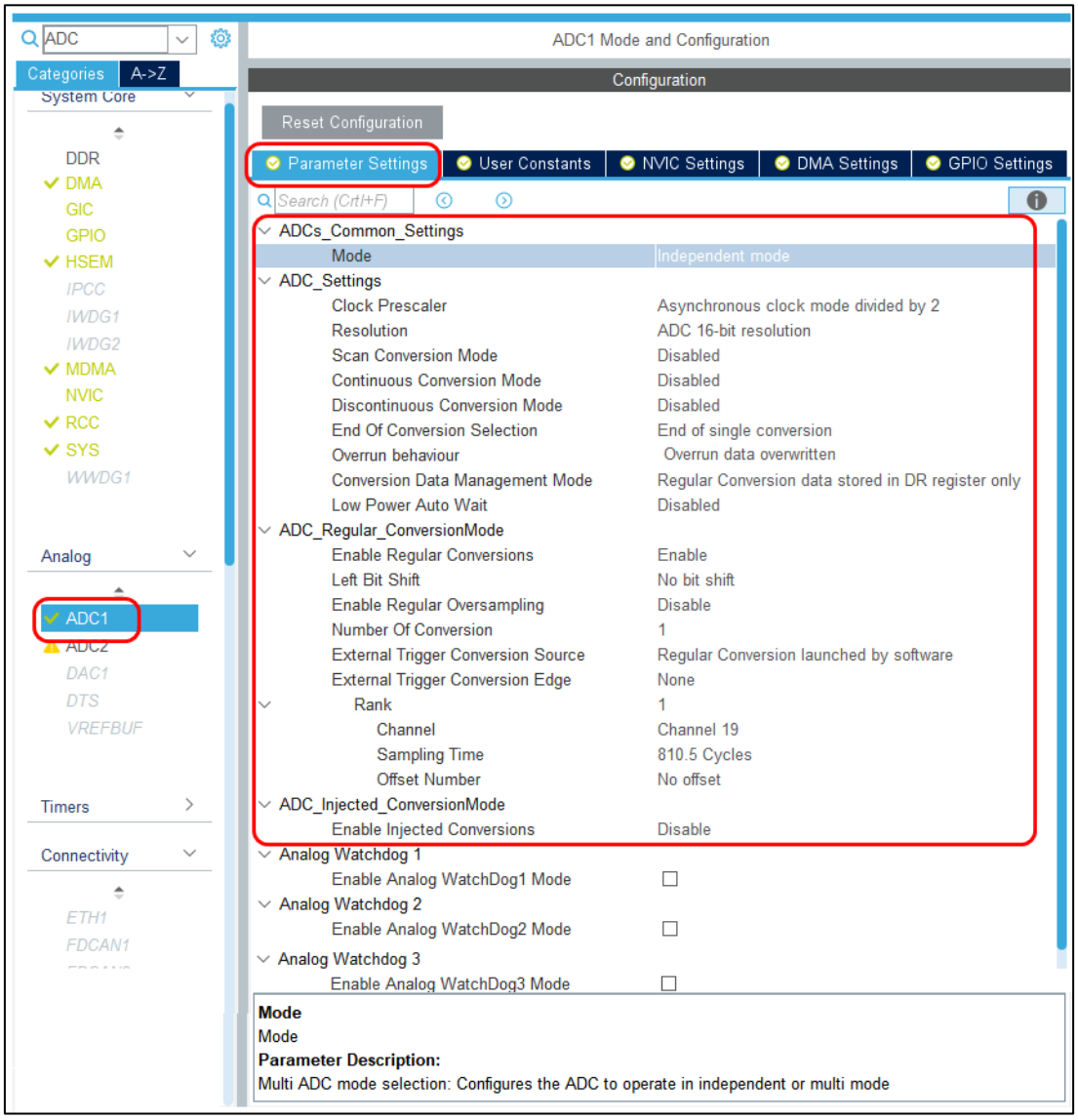


图 6.7.2.4 ADC1 初始化参数配置

对以上的 ADC 参数配置选项进行介绍：

● ADCs_Common_Settings（ADC 工作模式配置）：

这里配置为独立模式，独立模式是指**在同一个管脚上只有一个 ADC 采集该管脚的电压信号。**如果只是用了一个 ADC 的时候就配置为独立模式。

除了独立模式，还有双重模式以及三重模式等多重模式，多重模式是指双 ADC 共同工作，如果需要两个 ADC 同步的话，则使用此模式。

● ADC_Settings（ADC 参数设置）

✧ Clock Prescaler 用于配置时钟分频，这里选择 2 分频；

✧ Resolution 用于配置 ADC 的分辨率，这里选择 16 位；

✧ Scan Conversion Mode 用于配置扫描模式，当有多个通道需要采集信号时必须开启扫描模式，此时 ADC 将会按设定的顺序轮流采集各通道信号，单通道转换不需要使用此功能，这里选择 Dsabled；

✧ Continuous Conversion Mode 用于配置自动连续转换还是单次转换。使用 Enable 配置为使能自动连续转换；使用 Disabled 配置为单次转换，转换一次后停止需要手动控制才重新启

动转换，我们选择 Disabled;

- ✧ Disabled Discontinuous Conversion Mode 用于配置是否使用不连续的转换模式，所谓不连续，比如要转换的通道有 1、2、5、7、8、9，那么第一次触发会进行通道 1 与通道 2，下次触发就是转换通道 5 与通道 7，这样不连续的转换，依次类推。这里我们选择禁用不连续的转换模式 Disabled；
- ✧ End Of Conversion Selection 用于配置转换方式结束选择，可选择单通道转换完成后 EOC 标志位置位或者所有通道转换成后 EOC 置位，也可以选择转换序列结束后 EOS 置位（配置为 End of sequence of conversion），这里配置 End of single conversion，即单通道转换完成后 EOC 置位；
- ✧ Overrun behaviour 用于配置有新的数据溢出时，是覆盖写入还是丢弃新的数据，我们选择覆盖写入新的数据；
- ✧ Conversion Data Management Mode 用于配置转换数据管理模式，我们选择 Regular Conversion data stored in DR register only，即将常规转换的数据存储在DR寄存器中，另外还有DMA相关以及DFSDM，这两个本实验我们不使用；
- ✧ Low Power Auto Wait配置是否使用低功耗自动延迟等待模式，当使能时，仅当一组内所有之前的数据已处理完毕时，才开始新的转换，适用于低频应用，该模式仅用于ADC的轮询模式，不可用于DMA以及中断，这里我们选择关闭。

- ● ADC Regular_ConversionMode（ADC常规通道转换模式）
- ✧ Enable Regular Conversions 选择 Enable，启用常规转换；
- ✧ Left Bit Shit 数据左移位数，最多可支持左移 15 位，这里选择没有位转移 No bit shit；
- ✧ Enable Regular Oversampling 用于配置是否使用常规通道过采样，这里不使用此功能；
- ✧ Number Of Conversion 转换通道数量，此参数会影响可供设置的通道数，按实际使用的通道数来选择即可，这里是 1；
- ✧ External Trigger Conversion Source 外部触发转换模式配置，ADC 在接收到到触发信号后才开始进行模数转换，触发源可以是定时器触发、外部中断触发等硬件触发、也可以是软件控制触发，这里选择软件触发，工程中需要我们添加启动 ADC 的代码；
- ✧ External Trigger Conversion Edge 配置触发边沿，这里选择无触发边沿；
- ✧ Rank 配置模拟信号采集及转换的次序，默认是 1，其中：
- ✧ Channel 用于选择转换的通道，这里选择通道 19；
- ✧ Sampling Time 采样周期选择 810.5Cycles，即设置最大采样周期，要求尽量大以减少误差；
- ✧ Offset Number 配置偏移量的通道选择，这里选择无偏移通道；

- ● ADC_Injected_ConversionMode（ADC注入通道转换模式）
- ✧ Enable Injected Conversions 用于配置注入通道转换模式，实验中我们不需要使用注入通道，所以此项配为 Disable。

后面的是模拟量看门狗的设置，本实验我们不需要，不配置即可。关于 ADC 的介绍，大家可以参考《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南》或者《【正点原子】STM32MP1 M4 裸机 HAL 库开发指南》。

下面，我们配置时钟，配置 MCU 的时钟为 209MHz，使用外部时钟 HSE 或者内部时钟 HSI 都可以，如下图 6.7.2.5 和 6.7.2.6 所示，我们选择外部时钟 HSE：

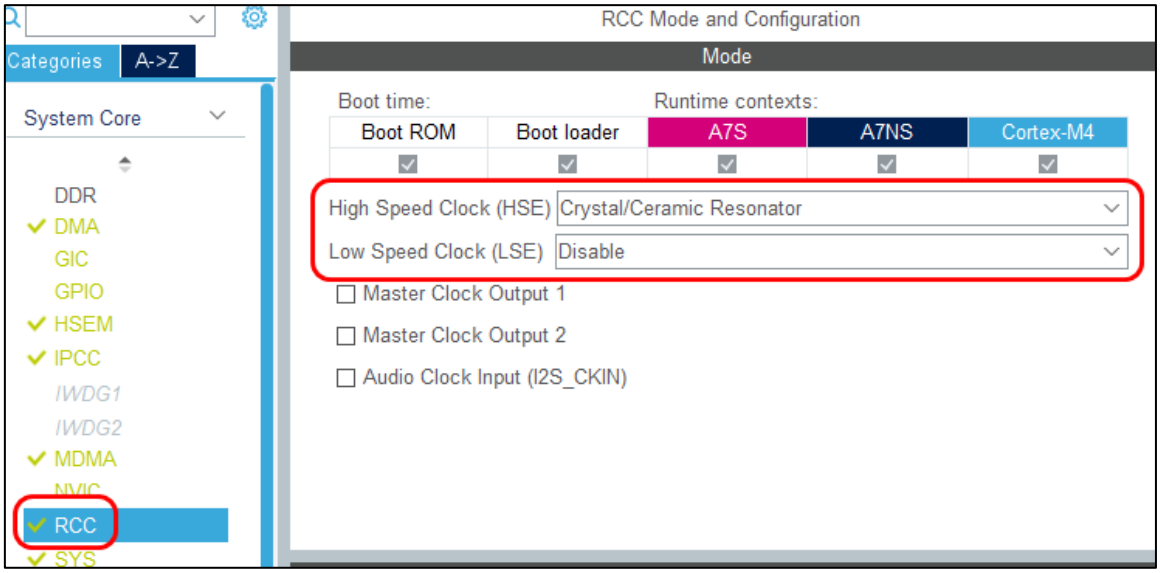


图 6.7.2.5 配置时钟源

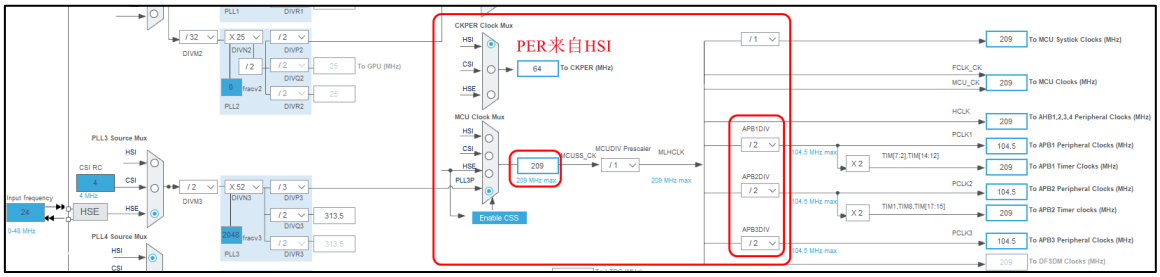


图 6.7.2.6 时钟树配置

ADC1 的时钟使用 PER，如下图 6.7.2.7 所示，其中 PER 时钟源默认使用 HSI，即为 64MHz。上面 ADC 参数配置中，我们配置分频系数为 2，所以实际 ADC 的时钟是 32MHz。

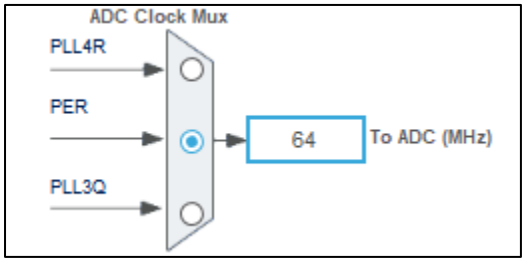


图 6.7.2.7 ADC 时钟源选择

选择将已经配置的每个外设生成独立的'.c/.h'文件，如下图6.7.2.8所示：

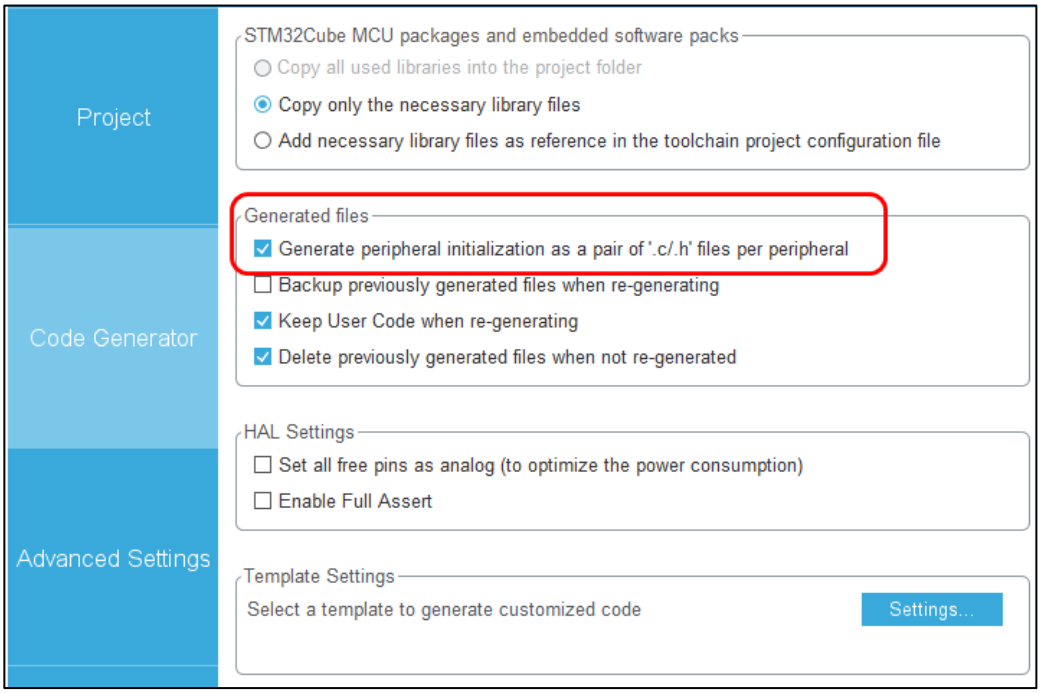


图6.7.2.8 配置生成独立的文件

保存配置，导出工程，下面我们在生成的工程中添加代码。

3. 添加代码

（1）添加 ADC1 相关代码

在 RPMsg_UART_ADC\CM4\Core\Src\adc.c 文件中添加如下代码：

```
/* USER CODE BEGIN 1 */

/**
 * @brief       获取 ADC1 通道 ch 的转换结果，先取 times 次,然后取平均
 * @param       ch      : 通道 ，为 0~19
 * @param       times   : 获取次数
 * @retval      通道 ch 的 times 次转换结果的平均值
 */
uint32_t adc_get_result_average(uint32_t ch, uint8_t times)
{
    uint32_t temp_val = 0;
    uint8_t t;
    /* ADC 校准 */
    HAL_ADCEx_Calibration_Start(&hadc1, ADC_CALIB_OFFSET,
                                ADC_SINGLE_ENDED);
    HAL_ADC_Start(&hadc1);                  /* 启动 ADC */
    HAL_ADC_PollForConversion(&hadc1, 10);/* 轮询转换 */
    for (t = 0; t < times; t++)             /* 获取 times 次数 */
    {
      /* 获取转换结果，并将每次转换结果进行相加 */
```

```
    temp_val += HAL_ADC_GetValue(&hadc1);
    HAL_Delay(5);
  }

  return temp_val / times;            /* 返回转换后的平均值*/
}

/* USER CODE END 1 */
```

首先，进行 ADC1 校准，ADC 校准需在 ADC 开始前或结束后，如果不校验，结果会有偏差。接下来启动 ADC1，我们前面配置的是软件触发模式，也就是需要启动 ADC 以后，ADC 才可以进行转换操作。接下来进行 ADC 轮询转换，这里设置的是转换 10 次，然后将 ADC 10 次的转换的结果进行相加，ADC 转换的值是保存在 ADC_DR 寄存器中的。最后就是返回这 10 次转换结果的平均值。

（2）添加核间通信代码

如下所示，在USER CODE BEGIN XXX和USER CODE END XXX之间手动添加代码：

```
1  #include "main.h"
2  #include "adc.h"
3  #include "ipcc.h"
4  #include "openamp.h"
5  #include "usart.h"
6  #include "gpio.h"
7
8  /* USER CODE BEGIN Includes */
9   #include "virt_uart.h"        /* include 虚拟串口头文件 */
10 /* USER CODE END Includes */
11
12 /* USER CODE BEGIN PTD */
13 uint8_t BuffTx[100];             /* 发送缓冲区 */
14 VIRT_UART_HandleTypeDef huart0; /* 虚拟串口 0 */
15 __IO FlagStatus flag = RESET;    /* M4 接收数据标志位 */
16 /* 声明 ADC 转换函数 */
17 uint32_t adc_get_result_average(uint32_t ch, uint8_t times);
18 /* 声明虚拟串口关联的接收回调函数 */
19 void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
20 /* 支持 UART3 通过通过 printf()函数打印信息 */
21 #ifdef __GNUC__
22 #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
23 #else
24 #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
25 #endif
26 PUTCHAR_PROTOTYPE
27 {
28  /* 本实验使用的是 UART3，如果使用的是其它串口，则将 UART3 改为对应的串口即可 */
```

```
29      while ((USART3->ISR & 0X40) == 0);/* 等待上一个字符发送完成 */
30      USART3->TDR = (uint8_t) ch;/* 将要发送的字符 ch 写入到 TDR 寄存器 */
31      return ch;                  /* 返回要发送的字符 */
32  }
33  /* USER CODE END PTD */
34
35  void SystemClock_Config(void);       /* 声明配置时钟函数 */
36  void PeriphCommonClock_Config(void); /* 声明外设时钟配置 */
37
38  int main(void)
39  {
40    /* USER CODE BEGIN 1 */
41    uint16_t adcx;
42    float temp;
43    /* USER CODE END 1 */
44
45    HAL_Init();                    /* 初始化 HAL 库 */
46    if(IS_ENGINEERING_BOOT_MODE())/* 检查平台是否为工程启动模式 */
47    {
48      SystemClock_Config();        /* 配置系统时钟 */
49    }
50
51    if(IS_ENGINEERING_BOOT_MODE()) /* 检查平台是否为工程启动模式 */
52    {
53      PeriphCommonClock_Config();   /* 外设时钟配置 */
54    }
55    MX_IPCC_Init();                      /* 初始化 IPCC */
56    MX_OPENAMP_Init(RPMSG_REMOTE, NULL);/* 初始化 OpenAMP 库 */
57    MX_GPIO_Init();                      /* 初始化 GPIO */
58    MX_USART3_UART_Init();               /* 初始化 UART3 */
59    MX_ADC1_Init();                      /* 初始化 ADC1 */
60
61    /* USER CODE BEGIN 2 */
62    printf("****** Start Initialize Virtual UART0 ******\r\n");
63      if (VIRT_UART_Init(&huart0) != VIRT_UART_OK)/* 初始化虚拟串口 */
64      {
65        printf("****** VIRT_UART_Init UART0 failed. ******\r\n");
66        Error_Handler();
67       }
68       /* 给虚拟串口 huart0 注册回调函数 VIRT_UART0_RxCpltCallback() */
69      if(VIRT_UART_RegisterCallback(&huart0, VIRT_UART_RXCPLT_CB_ID,
                          VIRT_UART0_RxCpltCallback) != VIRT_UART_OK)
70      {
```

```
71        Error_Handler();
72      }
73    /* USER CODE END 2 */
74
75    while (1)
76    {
77      /* USER CODE BEGIN 3 */
78      OPENAMP_check_for_message();/* 邮箱轮询，检查 A7 是否有发来数据 */
79      /* 获取 ADC1 通道 19 的转换结果 */
80      adcx = adc_get_result_average(ADC_CHANNEL_19, 10);
81      temp = (float)adcx * (3.3 / 65536); /* 将 ADC 值转化为电压值，单位为 v*/
82     // printf("ADC Value = %d, Voltage = %.0f mV\r\n", adcx, temp*1000);
83     /* 电压值转换为 mv，并以字符串的形式保存在 BuffTx 中 */
84      sprintf((char *)BuffTx, "%d\r\n",(int)(temp*1000));
85      /* 如果 M4 接收到 A7 发来的数据 */
86      if (flag==SET)
87      {
88        //flag = RESET;
89        /* M4 将 ADC 对应的电压值发给 A7（单位为 mv—） */
90        VIRT_UART_Transmit(&huart0, BuffTx,
                             strlen((const char *)BuffTx));
91     memset(BuffTx,0,strlen((const char *)BuffTx));/* 清空缓冲区 BuffTx */
92      }
93    HAL_Delay(100);   /* 延时 100ms */
94    }
95    /* USER CODE END 3 */
96  }
97
98  /**
99    * @brief 配置系统时钟
100   * @retval None
101   */
102 void SystemClock_Config(void)
103 {
104   /* 省略该函数的代码 */
105 }
106
107 /**
108   * @brief 外围设备通用时钟配置
109   * @retval None
110   */
111 void PeriphCommonClock_Config(void)
112 {
```

```
113   /* 省略该函数的代码 */
114 }
115
116 /* USER CODE BEGIN 4 */
117 void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart)
118 {
119   /* M4 打印接收到的数据以及数据的长度 */
120   printf("\n\rReceived on Virtual UART0:\n\r%s\n\rSize:%d\n\r",
            (char *) huart->pRxBuffPtr,huart->RxXferSize);
121   flag = SET; /* 设置标志位 flag=SET，表示 M4 接收到了数据 */
122 }
123 /* USER CODE END 4 */
124
125 /**
126   * @brief  此功能在发生错误时执行
127   * @retval None
128   */
129 void Error_Handler(void)
130 {
131
132 }
```

第 9 行，要用到虚拟串口，所以需要 include 头文件 virt_uart.h。

第 13 行，定义一个发送缓冲区，大小为 100 个字节，后面会将该缓冲区将存放 ADC1 通道 19 的数据，并通过虚拟串口发送给 A7。

第 14 行和第 15 行，分别定义一个虚拟串口句柄 huart0，并定义和初始化了虚拟串口接收标志 flag。

第 17 和第 19 行，分别声明 ADC 转换函数和虚拟串口绑定的接收回调函数。

第 21~第 32 行，通过重映射的方式将 printf()函数重映射到 STM32 串口的寄存器上，那么串口可通过 printf()输出流，最终 UART3 可以打印信息。

第 62~第 72 行，主要初始化了虚拟串口 huart0，并给 huart0 注册了虚拟串口接收回调函数。

第 78 行，邮箱轮询操作，检查 A7 是否有发来数据了，如果有发数据，则虚拟串口 huart0 绑定的接收回调函数 VIRT_UART0_RxCpltCallback()会被执行。

第 117~第 122 行，我们来看虚拟串口接收回调函数 VIRT_UART0_RxCpltCallback()做了哪些操作，首先将 A7 发来的数据以及数据的长度打印出来，然后将标志位 flag 设置为 SET，表示 M4 接收到了 A7 发来的数据。

第 80~第 84 行，先获取 ADC1 通道 19 采集到的数据，再将 ADC 值转换为电压值，然后将电压值进行格式化后拷贝到发送缓冲区 BuffTx 中。

第 86~第 92 行，当标志位 flag 为 SET 时，表明 A7 给 M4 发送数据了，此时 RPMsg 通道激活了，这个时候 M4 可以给 A7 发送数据了，所以 M4 将发送缓冲区中的数据（ADC 对应的电压值）发送给 A7，每次发送后都将发送缓冲区清零，否则发送缓冲区将会保留以前的数据。

第 93 行，此处设置延时时间为 100ms，即基本上每隔 100ms 后，M4 给 A7 发送数据。

以上代码比较简单，也是基于虚拟串口来完成数据的发送，下面我们直接编译工程，然后进行测试。

6.7.3 实验测试

1. 不使用屏幕进行测试

和前面的测试方法类似，在/lib/firmware 目录下新建一个 test4.sh 测试脚本，内容如下：

```
#!/bin/sh

echo RPMsg_UART_ADC_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware

echo start >/sys/class/remoteproc/remoteproc0/state

echo "OK" > /dev/ttyRPMSG0

cat /dev/ttyRPMSG0 &
```

给 test4.sh 脚本可执行权限，并将固件 RPMsg_UART_ADC_CM4.elf 拷贝到/lib/firmware 目录下，执行如下命令运行程序，如下图 6.7.3.1 所示，A7 加载和运行固件后，随即将 M4 发来的 ADC 数据打印出来，如果调节电位器的旋钮改变接入到 PA5 上的电压值，则可以看到串口打印的 ADC 值发生变化，范围在 0~3.3V 之间。

```
./test4.sh
```

```
root@ATK-MP157:/lib/firmware# ./test4.sh
[28245.199253] remoteproc remoteproc0: powering up m4
[28245.215223] remoteproc remoteproc0: Booting fw image RPMsg_UART_ADC_CM4.elf, size 2475868
[28245.222448]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[28245.232279] virtio_rpmsg_bus virtio0: rpmsg host is online
[28245.236482] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[28245.237427]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[28245.251242] remoteproc remoteproc0: remote processor m4 is now up
[28245.254353] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
root@ATK-MP157:/lib/firmware# 1226

1223

1229

1225      ADC电压值

1228

1225

1230
```

图 6.7.3.1 A7 端打印信息

M4 端打印信息如下图 6.7.3.2 所示：

```
****** Start Initialize Virtual UART0 ******

Received on Virtual UART0:
OK

Size:3
```

图 6.7.3.2 M4 端打印信息

如果要停止程序运行，可以执行前面我们创建的 q.sh 脚本文件，该文件的内容如下，执行该命令后，M4 随即停止运行程序。

```
echo stop >/sys/class/remoteproc/remoteproc0/state
```

2. 使用屏幕进行测试

使用屏幕进行测试的操作步骤和前面实验差不多，可以使用前面实验使用的测试 APP 程序 different_core 进行测试，也可以使用本实验配套的测试 APP 程序 different_core_adc。

将 different_core_adc 拷贝到开发板的/lib/firmware 目录下，执行如下命令运行测试 APP，如下图 6.7.3.3 所示，可以看到 A7 加载了 M4 固件。

```
chmod a+x different_core_adc                    /* 设置程序为可执行权限 */
/* 如果未关闭 Wayland，请先将其关闭 */
systemctl stop weston@root.service
./different_core_adc RPMsg_UART_ADC_CM4.elf  /* 运行测试 APP */
```

```
root@ATK-MP157:~# cd /lib/firmware/
root@ATK-MP157:/lib/firmware# chmod a+x different_core_adc
root@ATK-MP157:/lib/firmware# ls -l different_core_adc
-rwxr-xr-x 1 root root 1.1M Feb  8  2020 different_core_adc
root@ATK-MP157:/lib/firmware# systemctl stop weston@root.service
root@ATK-MP157:/lib/firmware# ./different_core_adc RPMsg_UART_ADC_CM4.elf
----注意请输入完整的固件路径----

固件路径为: "RPMsg_UART_ADC_CM4.elf"

[  233.216637] remoteproc remoteproc0: powering up m4
[  233.334064] remoteproc remoteproc0: Booting fw image RPMsg_UART_ADC_CM4.elf, size 2475868
[  233.341580]  mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[  233.354883] virtio_rpmsg_bus virtio0: rpmsg host is online
[  233.359108] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[  233.359871]  mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
[  233.373937] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
[  233.378711] remoteproc remoteproc0: remote processor m4 is now up
```

图 6.7.3.3 运行测试 APP

M4 串口打印如下图 6.7.3.4 所示。

```
****** Start Initialize Virtual UART0 ******

Received on Virtual UART0:
start

Size:6
```

图 6.7.3.4 M4 串口打印信息

此时，屏幕显示如下图 6.7.3.5 所示。

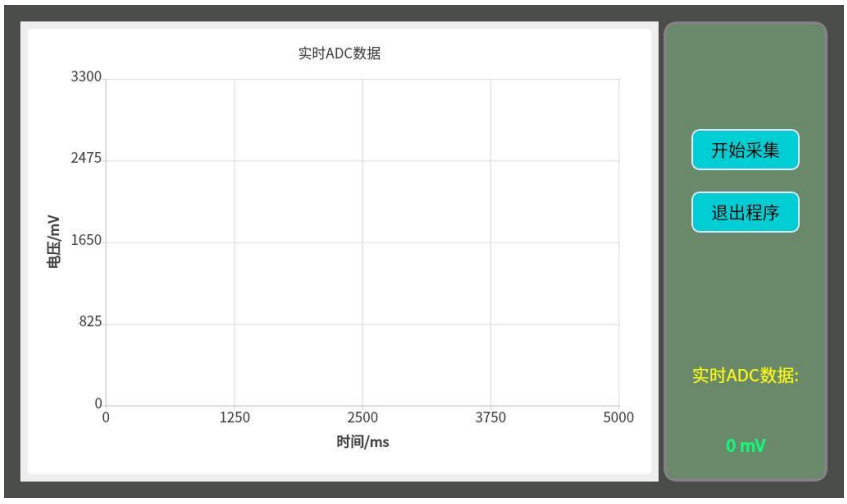


图 6.7.3.5 测试 APP 桌面

点击“开始采集”按钮，A7 运行了 M4 固件，此时屏幕显示实时采集到的 ADC 数据，并以动态实时曲线显示出来，如下图 6.7.3.6 所示，可以看到此时采集到的电压值约 1233mv，在没有旋转电位器时，显示的是一根水平的直线，说明在这段时间内电压值没有发生变化。

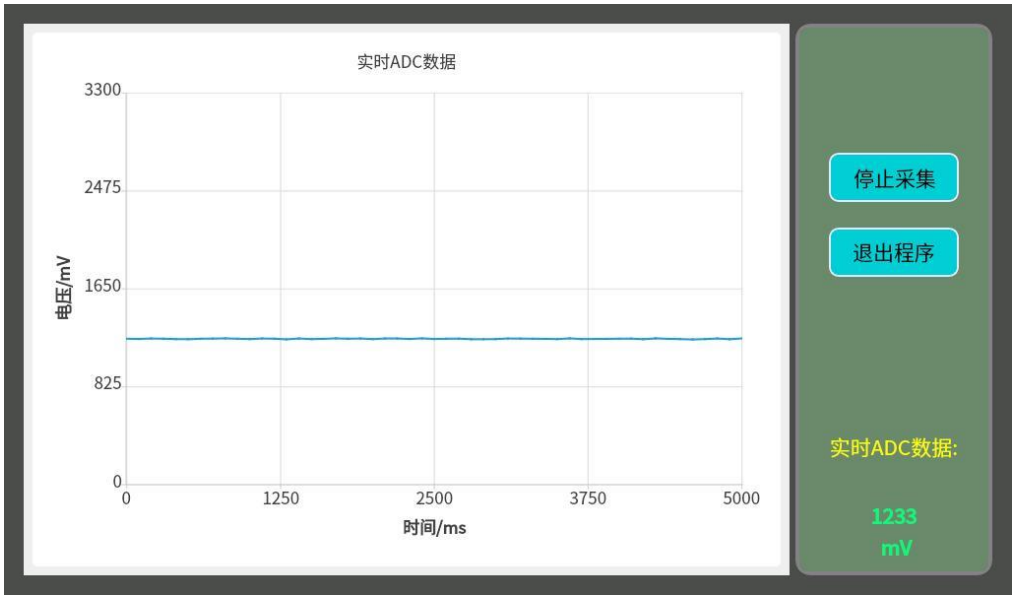


图 6.7.3.6 ADC 数据

我们旋转电位器，改变接入到 PA5 上的电压值，可以看到屏幕上显示的曲线变化，如下图 6.7.3.7 所示。

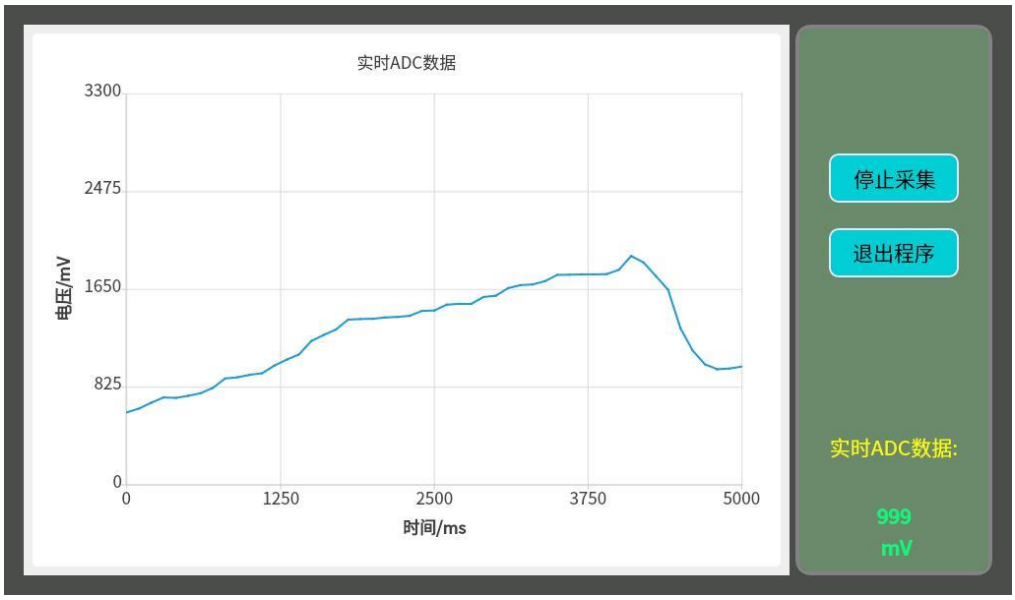


图 6.7.3.7 ADC 数值变化

如下图 6.7.3.8 所示，当电压值达到 2.5V 以后，数据变为红色，且蜂鸣器开始发出“嘀嘀嘀……”告警声。

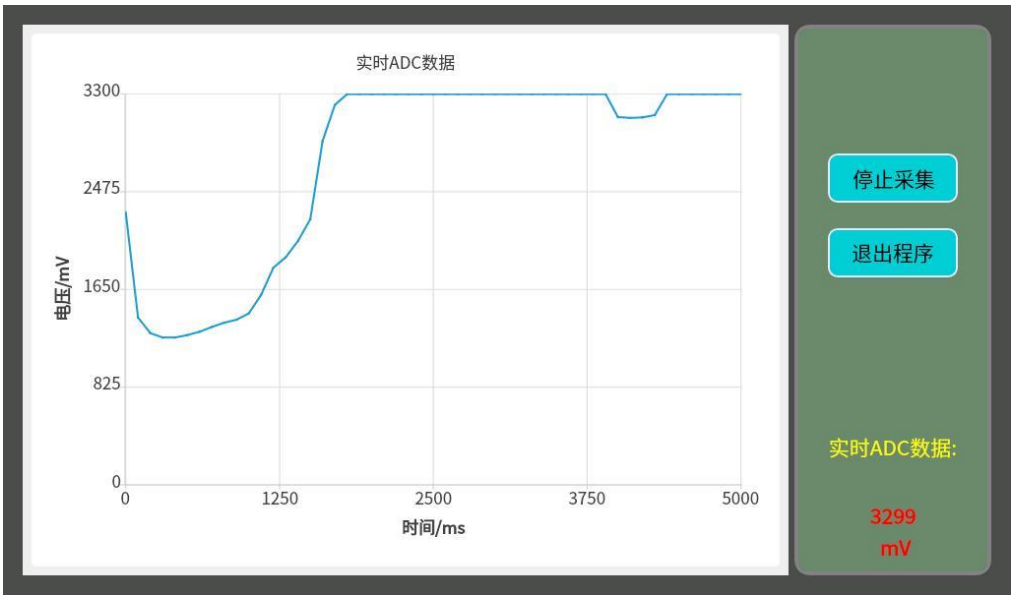


图 6.7.3.8 大于 2.5V 时蜂鸣器报警

点击“停止采集”按钮，即使旋转电位器，屏幕上显示的数据不再发生变化，再点击“开始采集”按钮，屏幕上的数据得到更新。如果要关闭 M4 固件，点击“退出程序”按钮即可。如下图 6.7.3.9 所示，点击“退出程序”后，关闭了 APP 测试程序，同时 A7 串口打印如下信息，表示停止运行 M4 固件。

```
root@ATK-MP157:/lib/firmware# [ 3176.295308] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: rpmsg tty d
evice 0 is removed
[ 3176.807874] remoteproc remoteproc0: warning: remote FW shutdown without ack
[ 3176.813463] remoteproc remoteproc0: stopped remote processor m4
```

图 6.7.3.9 停止运行 M4 固件

自此，本实验验证完毕。以上的工程代码和测试 APP 可执行文件和源码在开发板光盘 A-基础资料\01、程序源码\15、异核通信例程源码下可以找到。

第七章 系统的休眠和唤醒

待更新。

附录-A