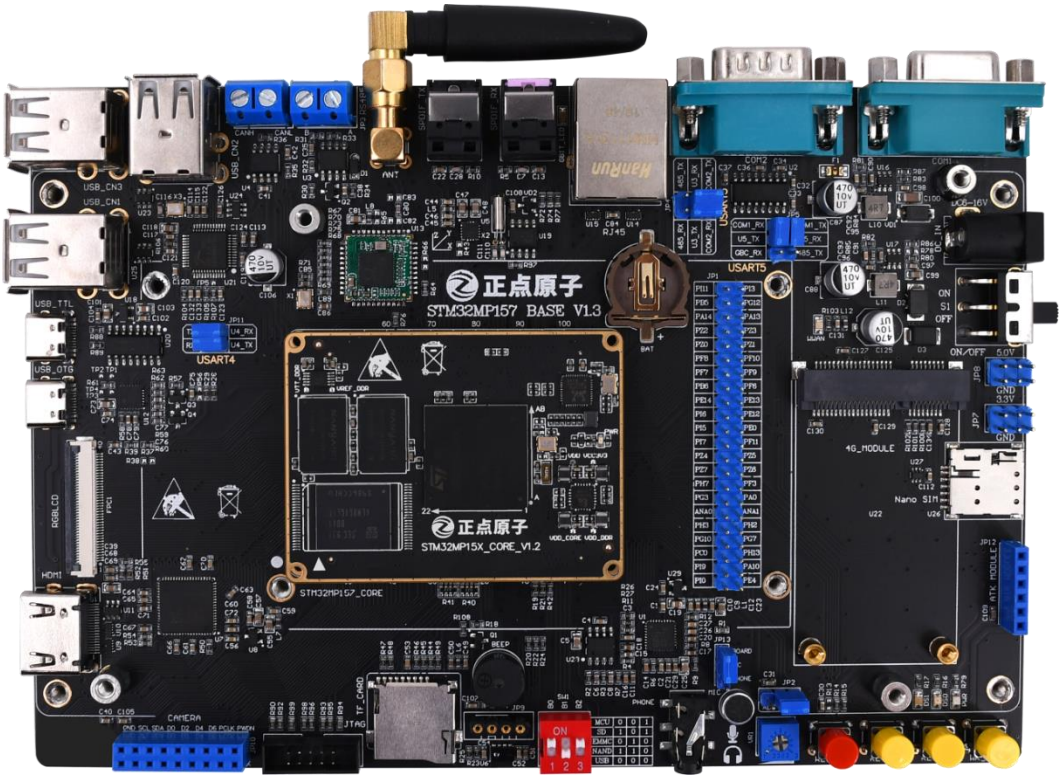
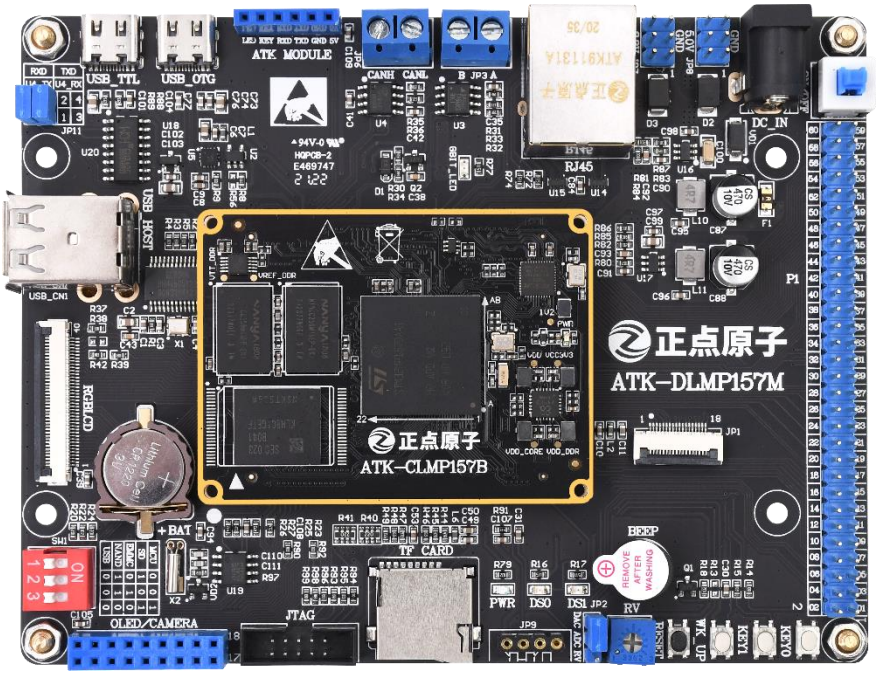
【正点原子】STM32MP1 异核通讯 V1.0





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

目录

前言

早期的计算机程序是硬件化，程序都是由各种门电路通过组装固定在一个电路板，数据就放在存储器中，这样做我们一旦要修改程序功能，就要重新组装电路，这做效率低，灵活性较差。这时候冯.诺依曼就提出一个理论就是把程序和数据都当作数据来看待，把程序进行编码，然后与数据一同存放在存储器中，这样存储器的数据可以被计算机读取当作为程序来处理。所有的程序都会转化为数据的形式存储在存储器中，我们的计算机要运行相应的程序只需要从存储器中依次取出指令、执行，冯.诺依曼结构的核心：硬件和软件分离，即硬件设计和程序设计可以分开!!!就是这种结构催生了程序员这个职业的诞生!

在 1971 年， Intel 公司设计制造一款名为 4004 微处理器，这款芯片研发成功，它严格准守冯诺依曼体系结构，因此第一款 CPU(第一款商用的微处理器)诞生了。那时候的 4004 微处理器只有一个核，慢慢的发现一个核就不够用了，就向多核处理器发展了。多核处理器可以分为 SMP(Symmetric multiprocessing，对称多处理器结构)和 AMP(Asymmetric Multi-Processing, 非对称多处理器结构)。

1. 对称多处理器结构(SMP)

SMP 的特征：多个 CPU 结构都是一样的，它们平等地访问内存、外设、操作系统，简单来说就是共享所有资源。如果两个 CPU 同时访问一个资源，就由硬件、软件的锁机制去解决资源争用问题。

2. 非对称多处理器结构(AMP)

AMP 的特征：有多个 CPU，每个 CPU 在架构上不一样，各个 CPU 内核运行一个独立的操作系统或同一个操作系统的独立实例，每个 CPU 拥有自己的独立资源。这种结构最大的特点在于不共享资源。

AMP 这种结构虽然目前还不是嵌入式的主流，肯定是未来的趋势。AMP 比 SMP 的优点在于物尽其用、以最佳地平衡成本、性能和功耗。AMP 的编程难度也更低。

STM32MP157 就是用 AMP 架构的，MCU+MPU(A7+M4)的组合，这样做 MPU 可以去跑 Linux，MCU 可以跑 FreeRTOS。对于 STM32MP1，不同的核心是如何启动运行，又是如何进行通讯？带着这个疑问我们学习这一篇的内容。

第一章 STM32MP1 的 M4

1.1 Remoteproc 框架简介

1.1.1 Remoteproc 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 内核可以对远程处理器及其相关软件环境进行生命周期管理和通讯。该框架为用户提供了简化的应用级接口，从而消除了管理异构硬件和软件环境的复杂性。

在 Linux3.4.X 版本以后，加入了 remoteproc 和 rpmsg API 的核间基础通讯架构，此基础架构允许主处理器上的 linux 操作系统管理远程处理器上远程软件环境的生命周期和通讯。我们以 STM32MP157 为例来看下 M4 和 A7 的 Remoteproc 框架如下所示：

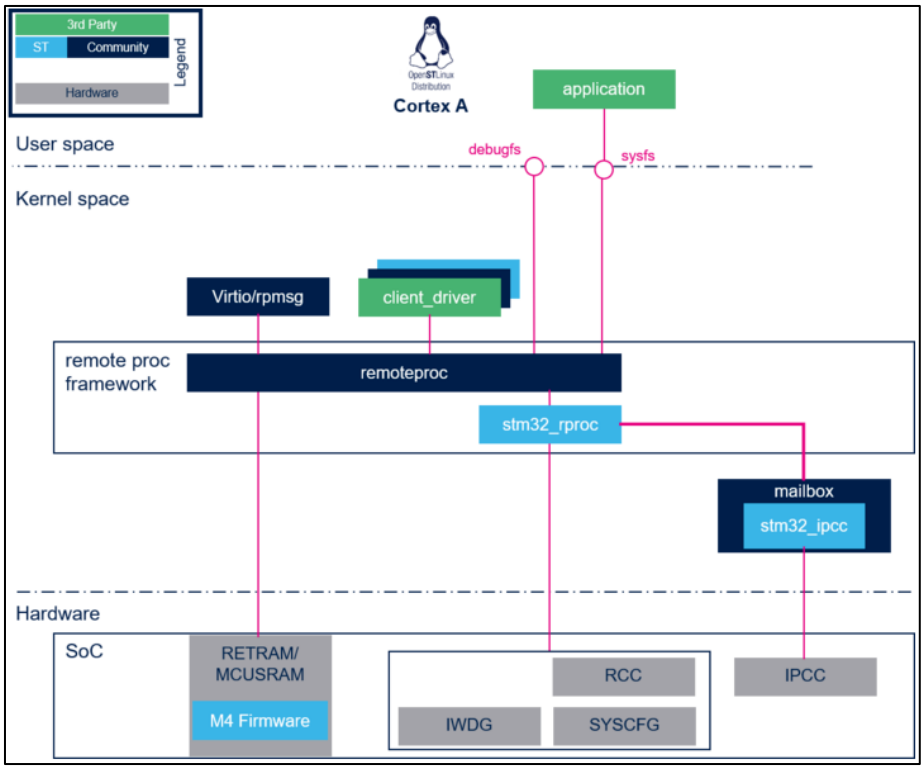


图 1.1.1.1 Remoteproc 框架

remoteproc 是通用远程处理框架部分，其作用是：

1）将.elf 文件加载到 Cortex-M4 内核中（在 MDK 下是.axf 文件）；

2）解释.elf 文件资源表以设置关联的资源（例如 IPC 和内存分割等）；

3）控制 Cortex-M4 内核启动、关闭；

4）提供监视和调试远程服务

stm32_rproc 是远程处理器平台（即 M4）驱动程序，其作用是：

1）向 Remoteproc 框架注册供应商特定的功能（如回调部分）；

2）处理 Cortex-A7 和 Cortex-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）通过邮箱框架将通知转发到 M4；

1.1.2 Linux 下 Remoteproc 源码简析

打开 include/linux/remoteproc.h，文件找到如下示例代码：

示例代码 1.1.2.1 rproc 结构体

```
1  struct rproc {
2      struct list_head node;
3      struct iommu_domain *domain;
4      const char *name;
5      char *firmware;          //加载固件的名字
6      void *priv;             /*********************
                               *芯片厂商保存自己私有数
                               *据的指针
                               **********************/
7      struct rproc_ops *ops;  /*********************
                               *rproc_ops 结构体是芯片
                               *厂商做好的，主要作用是
                               *加载、检验和控制固件等
                               *等
                               **********************/
8      struct device dev;
9      atomic_t power;
10     unsigned int state;
11     struct mutex lock;
12     struct dentry *dbg_dir;
13     struct list_head traces;
14     int num_traces;
15     struct list_head carveouts;
16     struct list_head mappings;
17     u32 bootaddr;       //加载的首地址
18     struct list_head rvdevs;
19     struct list_head subdevs;
20     struct idr notifyids;
21     int index;
22     struct work_struct crash_handler;
23     unsigned int crash_cnt;
24     bool recovery_disabled;
25     int max_notifyid;
26     struct resource_table *table_ptr;
27     struct resource_table *cached_table;
28     size_t table_sz;
29     bool has_iommu;
30     bool auto_boot;
31     struct list_head dump_segments;
```

```
32      int nb_vdev;
33      bool early_boot;    //固件加载标志位(0 表示加载，1 表示没有加载)
34  };
```

对于写驱动的人来说，rproc 结构体是 Remoteproc 框架的一个重要的结构体。只要实现此结构体加上 rproc_add 函数把 rproc 结构体注册进 Remoteproc 核心，就能够为应用层提供接口去加载固件。接着我们看下 ST 官方是不是这样做的。

首先要找到设备树的 M4 配置，打开 arch/arm/boot/dts/stm32mp151.dtsi 文件，

示例代码 1.1.2.2 mlahb 节点

```
1970    mlahb {
1971        compatible = "simple-bus";
1972        #address-cells = <1>;
1973        #size-cells = <1>;
1974        dma-ranges = <0x00000000 0x38000000 0x10000>,
1975                <0x10000000 0x10000000 0x60000>,
1976                <0x30000000 0x30000000 0x60000>;
1977
1978        m4_rproc: m4@10000000 {
1979            compatible = "st,stm32mp1-m4";
1980            reg = <0x10000000 0x40000>,
1981                  <0x30000000 0x40000>,
1982                  <0x38000000 0x10000>;
1983            resets = <&scmi0_reset RST_SCMI0_MCU>;
1984            st,syscfg-holdboot = <&rcc 0x10C 0x1>;
1985            st,syscfg-tz = <&rcc 0x000 0x1>;
1986            st,syscfg-rsc-tbl = <&tamp 0x144 0xFFFFFFFF>;
1987            st,syscfg-copro-state = <&tamp 0x148 0xFFFFFFFF>;
1988            st,syscfg-pdds = <&pwr_mcu 0x0 0x1>;
1989            status = "disabled";
1990
1991            m4_system_resources {
1992                compatible = "rproc-srm-core";
1993                status = "disabled";
1994            };
1995        };
1996    };
```

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 节点就是加载和管理 M4 的固件配置信息。m4_system_resources 节点就是 M4 和 A7 的资源分配配置信息。第 1979 行，compatible 属性值为“st,stm32mp1-m4”，在 Linux 源码中搜索此属性值找到对应的驱动文件。加载和管理 M4 固件的驱动文件为 drivers/remoteproc/stm32_rproc.c，打开此文件找到如下内容：

示例代码 1.1.2.3 stm32_rproc.c 文件代码段

```
623 static const struct of_device_id stm32_rproc_match[] = {
624     { .compatible = "st,stm32mp1-m4" },
```

```
625     {},
626 };
......
893 static struct platform_driver stm32_rproc_driver = {
894     .probe = stm32_rproc_probe,
895     .remove = stm32_rproc_remove,
896     .shutdown = stm32_rproc_shutdown,
897     .driver = {
898         .name = "stm32-rproc",
899         .pm = &stm32_rproc_pm_ops,
900         .of_match_table = of_match_ptr(stm32_rproc_match),
901     },
902 };
903 module_platform_driver(stm32_rproc_driver);
904
905 MODULE_DESCRIPTION("STM32 Remote Processor Control Driver");
906 MODULE_AUTHOR("Ludovic Barre <ludovic.barre@st.com>");
907 MODULE_AUTHOR("Fabien Dessenne <fabien.dessenne@st.com>");
908 MODULE_LICENSE("GPL v2");
```

上面的驱动代码是一个标准的 platform 驱动，我们直接去看 stm32_rproc_probe 函数吧。stm32_rproc_probe 函数定义如下所示：

示例代码 1.1.2.4 stm32_rproc_probe 函数

```
778 static int stm32_rproc_probe(struct platform_device *pdev)
779 {
780     struct device *dev = &pdev->dev;
781     struct stm32_rproc *ddata;
782     struct device_node *np = dev->of_node;
783     struct rproc *rproc;
784     int ret;
785
786     ret = dma_coerce_mask_and_coherent(dev, DMA_BIT_MASK(32));
787     if (ret)
788         return ret;
789
790     rproc = rproc_alloc(dev, np->name, &st_rproc_ops, NULL,
sizeof(*ddata));
791     if (!rproc)
792         return -ENOMEM;
793
794     rproc->has_iommu = false;
795     ddata = rproc->priv;
796     ddata->workqueue = create_workqueue(dev_name(dev));
797     if (!ddata->workqueue) {
```

```
798         dev_err(dev, "cannot create workqueue\n");
799         ret = -ENOMEM;
800         goto free_rproc;
801     }
802
803     platform_set_drvdata(pdev, rproc);
804
805     ret = stm32_rproc_parse_dt(pdev);
806     if (ret)
807         goto free_wkq;
808
809     if (!rproc->early_boot) {
810         ret = stm32_rproc_stop(rproc);
811         if (ret)
812             goto free_wkq;
813     }
814
815     ret = stm32_rproc_request_mbox(rproc);
816     if (ret)
817         goto free_wkq;
818
819     ret = rproc_add(rproc);
820     if (ret)
821         goto free_mb;
822
823     return 0;
824
825 free_mb:
826     stm32_rproc_free_mbox(rproc);
827 free_wkq:
828     destroy_workqueue(ddata->workqueue);
829 free_rproc:
830     if (device_may_wakeup(dev)) {
831         dev_pm_clear_wake_irq(dev);
832         device_init_wakeup(dev, false);
833     }
834     rproc_free(rproc);
835     return ret;
836 }
```

第 781 行，ddata 指针是 ST 官方的私有数据 stm32_rproc 结构体。

第 783 行，声明了一个 rproc 类型的结构体。

第 790 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（前面说过了 ops 主要实现如何加载固件、解析固件和控制固件），后面在分析 st_rproc_ops 结构体。

第 795 行，保存 ST 官方的私有数据到 rproc 结构体里。

第 796 行，创建工作队列。

第 805 行，stm32_rproc_parse_dt 函数主要作用是根据设备树去设置 ST 官方的 ddata 私有数据。

第 815 行，stm32_rproc_request_mbox 函数主要作用是异核通讯相关的，本章的任务为了解如何加载 M4 固件，所以就不去分析通讯相关的函数。

第 819 行，rproc_add 函数，从名字上就知道把 STM32MP1 异核相关的一些信息添加到 remoteproc 核心框架里进行注册。这样说明和我们前面猜想的一样。

写到这里笔者发现了一个细节，在驱动文件分析各种源码都有 XXX_ops(uart_ops，pinctrl_ops)，这样命名都是芯片厂商自己实现的回调函数，核心驱动就会回调这些函数进行一系列的初始化或者根据这些结构体给核心框架提供接口。

接着我们去看基于 remoteproc 框架下，ST 官方提供的 XXX_ops 结构体的实现，还是在 stm32_rproc.c 代码里，找到如下代码：

示例代码 1.1.2.4 st_rproc_ops 结构体

```
612 static struct rproc_ops st_rproc_ops = {
613     .start      = stm32_rproc_start,
614     .stop       = stm32_rproc_stop,
615     .kick       = stm32_rproc_kick,
616     .load       = stm32_rproc_elf_load_segments,
617     .parse_fw   = stm32_rproc_parse_fw,
618     .find_loaded_rsc_table = stm32_rproc_elf_find_loaded_rsc_table,
619     .sanity_check   = stm32_rproc_elf_sanity_check,
620     .get_boot_addr  = stm32_rproc_elf_get_boot_addr,
621 };
```

上面的 st_rproc_ops 结构体就不去分析，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。

1.2 在 Linux 下实现 M4 控制跑马灯

1.2.1 STM32MP1 的资源分配表

表 1.2.1.1 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。

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	TIM	TIM1		☐	☐	单选

器		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 1.2.1.1 STM32MP1 A7 和 M4 外设资源分配

有了 STM32MP157 的资源分配表后，我们可以按照此表进行 STM32MP1 的资源分配。

1.2.2 设备树的配置

本章实验用出厂设备树来讲的，出厂的设备树已经使能加载 M4 的 remoteproc 框架。打开 arch/arm/boot/stm32mp157d-atk.dtsi 文件，找到如下代码所示：

示例代码 1.2.2.1 m4_rproc 节点

```
1   &m4_rproc {
2       memory-region = <&retram>, <&mcuram>, <&mcuram2>, <&vdev0vring0>,
3               <&vdev0vring1>, <&vdev0buffer>;
4       mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
5       mbox-names = "vq0", "vq1", "shutdown";
6       interrupt-parent = <&exti>;
7       interrupts = <68 1>;
8       wakeup-source;
9       status = "okay";
10  };
```

这段代码的 ST 官方给 m4_rproc 节点追加一些配置去使能 remoteproc 框架。

当 M4 核要使用资源的时候要考虑两个问题：M4 能否控制外设、外设是否和 A7 核共享。根据资源表可以得知 LED0 和 LED1 使用的 GPIO 口是可以共享的，最好把 A7 核控制此两个 GPIO 屏蔽掉。关闭 A7 的 LED 灯如下所示：

```
102     leds {
103         compatible = "gpio-leds";
104
105         led1 {
106             label = "sys-led";
107             gpios = <&gpioi 0 GPIO_ACTIVE_LOW>;
108             linux,default-trigger = "heartbeat";
109             default-state = "on";
110             status = "disabled";
111         };
112
113         led2 {
114             label = "user-led";
115             gpios = <&gpiof 3 GPIO_ACTIVE_LOW>;
116             linux,default-trigger = "none";
117             default-state = "on";
118             status = "disabled";
119         };
```

图 1.2.2.1 关闭 A7 控制 LED 灯

我们只要把两个 LED 的 status 修改为“disabled”即可。编译设备树，使用新的设备树去启动系统。有如下图打印信息：

```
remoteproc remoteproc0: releasing m4
Unable to get STM32 DDR PMU clock
debugfs: Directory 'cpu0' with parent 'opp' already present!
remoteproc remoteproc0: releasing m4
Unable to get STM32 DDR PMU clock
```

图 1.2.2.1 remoteproc 框架打印信息

1.2.3 M4 跑马灯测试

测试代码直接使用《【正点原子】STM32MP1 M4 裸机 CubeIDE 开发指南 V1.5.1.pdf》文档的第 10 章跑马灯实验，把编译好的 LED_CM4.elf 文件，拷贝到文件系统/lib/firmware/目录下，拷贝结果如下：

```
[root@ATK-stm32mp1]:/lib/firmware$ ls
LED_CM4.elf         regulatory.db       regulatory.db.p7s   rtlwifi
[root@ATK-stm32mp1]:/lib/firmware$
```

图 1.2.3.1 M4 固件位置

要加载 M4 固件必须把固件放到/lib/firmware/目录里(这是内核的源码决定的,可以修改)，remoteproc 框架是用 sysfs 文件系统给用户层提供接口。加载 M4 固件步骤如下：

- 固件拷贝到 sysfs 文件系统里

```
echo LED_CM4.elf >/sys/class/remoteproc/remoteproc0/firmware
```

- 启动固件

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

运行上面的命令后，LED0 和 LED1 实现跑马灯功能(A7 核也在控制这两个灯)。有以下打印信息输出：

```
[root@ATK-stm32mp1]:/lib/firmware$ echo start > /sys/class/remoteproc/remoteproc
0/state
remoteproc remoteproc0: powering up m4
remoteproc remoteproc0: Booting fw image LED_CM4.elf, size 1913056
remoteproc remoteproc0: header-less resource table
remoteproc remoteproc0: no resource table found for this firmware
remoteproc remoteproc0: header-less resource table
remoteproc remoteproc0: remote processor m4 is now up
[root@ATK-stm32mp1]:/lib/firmware$
```

图 1.2.3.2 启动 M4 的打印信息

- 关闭 M4 命令

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

关闭 M4 后有以下输出信息：

```
[root@ATK-stm32mp1]:/lib/firmware$ echo stop > /sys/class/remoteproc/remoteproc0
/state
remoteproc remoteproc0: warning: remote FW shutdown without ack
remoteproc remoteproc0: stopped remote processor m4
[root@ATK-stm32mp1]:/lib/firmware$
```

图 1.2.3.3 关闭 M4 的打印信息

第二章 通讯框架

2.1 IPCC、Mailbox 和 RPMsg 简介

IPCC 是 STM32MP157 硬件层的异核通讯接口，Mailbox 就是软件层的异核通讯框架，RPMsg 是异核间共享内存的框架。那么它们三者有啥关系？在 STM32MP157 里通讯只有一种就是共享内存，RPMsg 给内存提供好数据后，调用 mailbox 去设置 IPCC 相关寄存器，对应的处理器就会去内存里读取数据。

2.1.1 IPCC 简介

IPCC 是 STM32MP1 的硬件外设，主要作用是管理两个处理器实例之间的处理器间通讯。每个处理器都拥有特定的寄存器和中断。

IPCC 提供了六个双向通道，每个通道分为两个子通道，每个子通道提供从“发送端”处理器到“接收端”处理器的单向通讯：

- P1_TO_P2 子通道(P1 就是发送端处理器，P2 就是接收端处理器，P1 发到 P2)
- P2_TO_P1 子通道(P2 就是发送端处理器，P1 就是接收端处理器，P2 发到 P1)

子通道中还包含如下功能：

- 一个标志位，用来标识通道处于空闲状态或者占用状态，当发送端处理器发送数据就设置标志位为占用状态，接收端处理器接收到数据就设置为空闲状态。
- 两个相关的中断(与所有通道共享)：RXO 中断(RX 通道被占用，连接到接收端处理器)和 TXF 中断(TX 通道空闲，连接到发送端处理器)。
- 两个相关中断带多路复用的中断掩码功能。

IPCC 支持以下通道的操作模式：

- 单工通讯方式：
  1. 仅使用一个子通道。
  2. 单向消息：发送端处理器将通讯数据发布到内存中后，它将通道状态标志设置为已占用。当消息被处理时，接收端处理器清除该标志。
- 半双工通讯方式：
  1. 仅使用一个子信道。
  2. 双向消息：发送端处理器将通讯数据发布到内存中后，它将通道状态标志设置为已占用。当消息被处理并且响应在共享内存中可用时，接收端处理器将清除该标志。
- 全双工通讯方式：
  1. 子通道用于异步模式。
  2. 通过将子通道状态标志设置为占用，任何处理器都可以异步发布消息。当消息被处理时，接收端处理器清除该标志。可以将这种模式视为给定通道上两个单工模式的组合。

IPCC 的核间通讯模型如下：

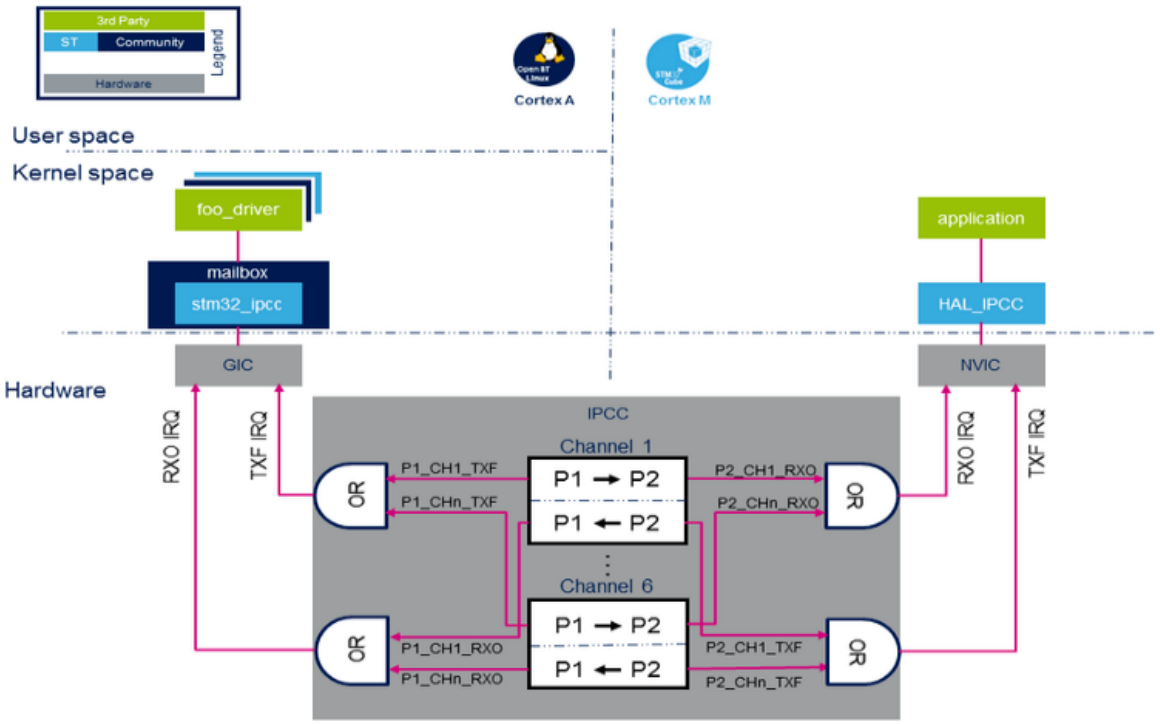


图 2.1.1.1 IPCC 核间通讯模型

图 2.1.1.1 是一个通道的模型，从芯片手册可以知道 IPCC 有六个通道，前面 3 个通道是给 ST 官方分配给 RPMsg 框架和 Remoteproc 框架，后面 3 个是自由分配。前 3 个通道的分配如下表所示：

通道	模式	用法	A7(Linux)	M4(STM32cube)
1	全双工通讯	从 M4 到 A7 的 RPMsg 传输	RPMsg	OpenAMP
2	全双工通讯	从 A7 到 M4 的 RPMsg 传输	RPMsg	OpenAMP
3	单工通讯	控制 M4 相关	RemoteProc	CprocSync cube utility
4		未定义		
5		未定义		
6		未定义		

2.1.2 Mailbox 简介

Mailbox 是内核一种架构，通过消息队列和中断驱动信号处理多处理器间的通讯。

Mailbox 的实现分为 contoller 和 client。简单的说就是 client 可以通过 controller 提供 channel 发送信息给 controller。STM32MP1 的 Mailbox 框架如下图所示：

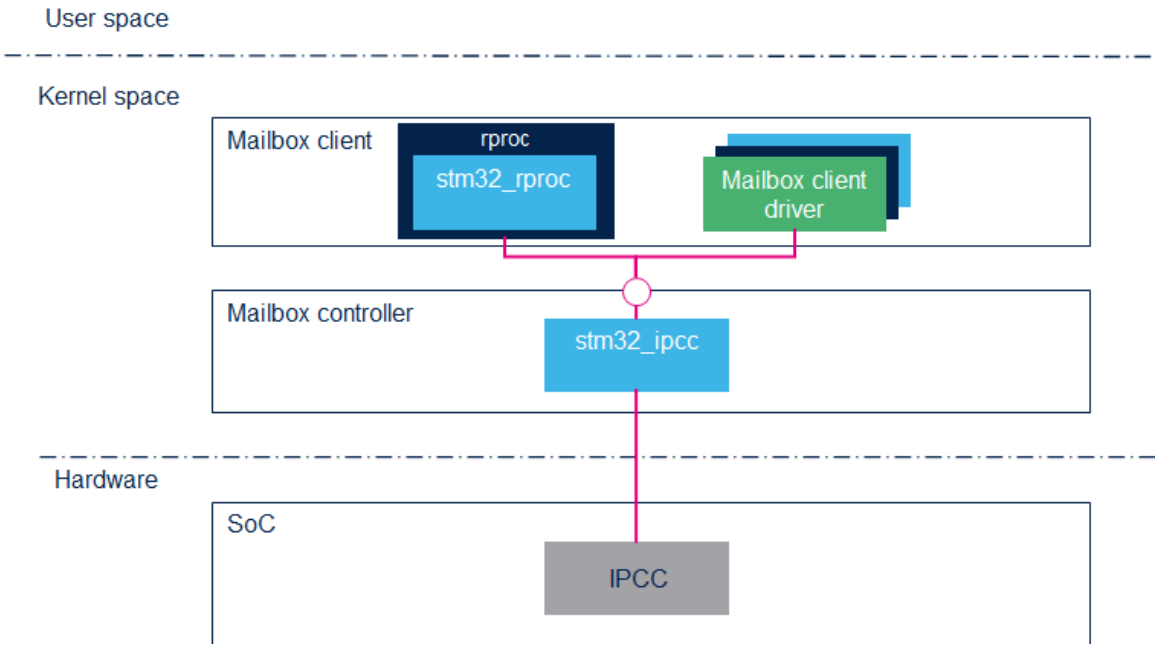


图 2.1.2.1 Mailbox 框架

邮箱控制器(mailbox controller)，依赖于硬件平台实现，比如 STM32MP1 的 IPCC 外设：

- controller 负责处理来自 IPCC 外围设备的 IRQ
- controller 为邮箱客户端提供了通用 API。

邮箱客户端(mailbox client)，负责发送或接收。

2.1.3 RPMsg 简介

Linux RPMsg 框架是基于 virtio 框架的基础上实现的本地处理器与系统上的远程处理器通讯。RPMsg 框架在异核通讯的位置如下所示：

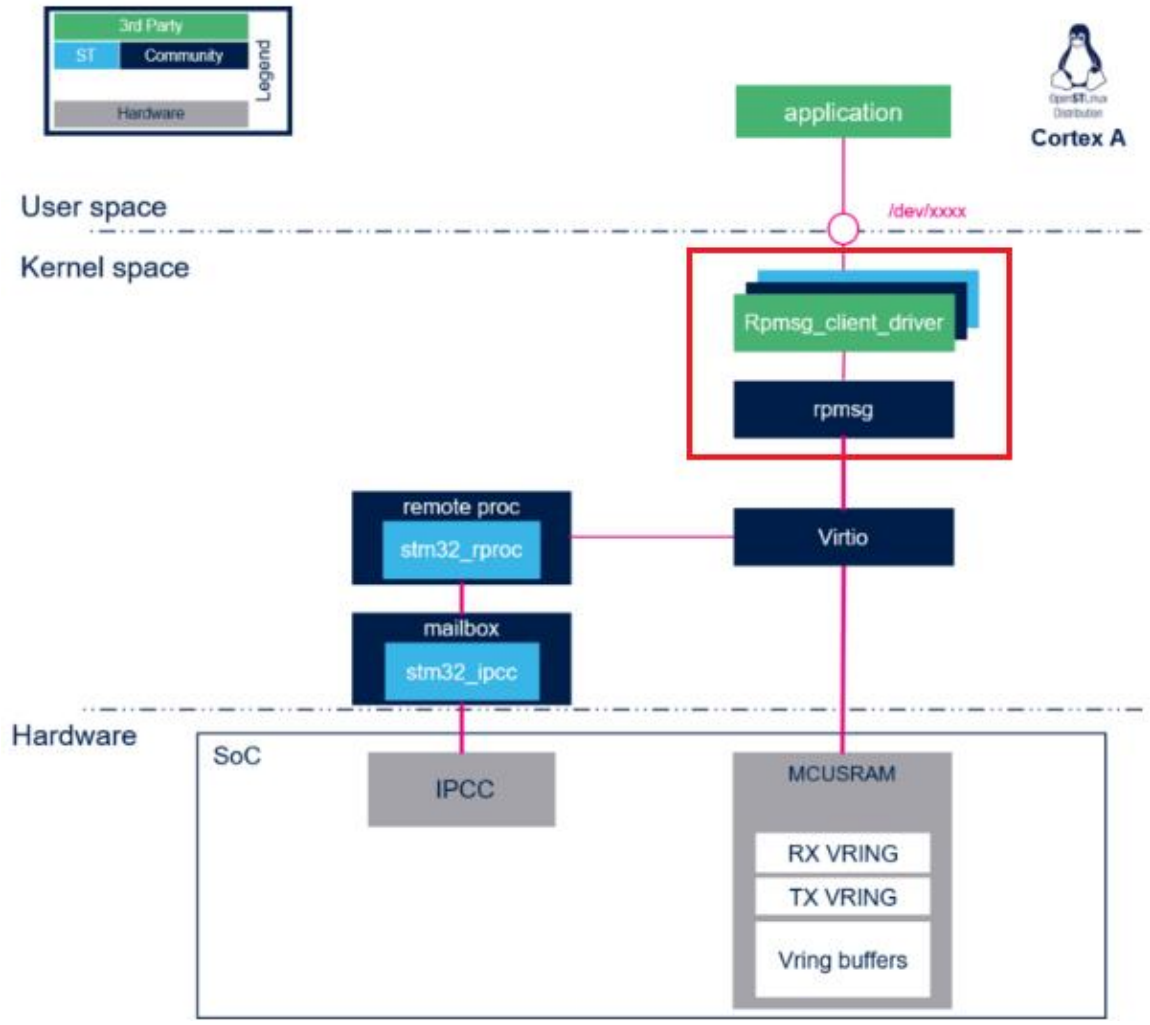


图 2.1.3.1 RPMsg 框架

图 2.1.3.1 红色框就是 RPMsg 框架。virtio 框架支持虚拟化，它提供一个共享环形缓冲区(vring)的高效传输层。图 2.1.3.1 中有两个 vring，这些 virng 是单向的，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。vring 的缓冲区是两处理器都能使用的，所以 STM32MP1 的 viring 空间在 RETRAM 和 MCUSRAM 里。这两个空间的地址是在我们 stm32mp157d-atk.dtsi 文件里定义，找到如下示例代码：

示例代码 2.1.3.1 stm32mp157d-atk.dtsi 代码段

```
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      };
```

mcuram2 节点主要用来保存 M4 固件代码和数据，起始地址为：0x10000000，大小为 256 KB。

vdev0vring0 节点和 vdev0vring1 节点就前面说的两个 vring，两个起始地址分别为：0x10040000 和 0x10041000，它们的大小都为 4KB。

vdev0buffer 节点，起始地址为：0x10042000，大小为 16KB，不知道有啥作用。

mcuram 节点不知道有啥用，笔者猜测应该是用来保存 M4 固件代码和数据，这是一个 mcuram2 的备份空间。

retram 节点保存 M4 的异常向量表。起始地址为：0x38000000，大小为 64KB。

总结一下 IPCC、Mailbox 和 RPMsg 如何实现异核通讯，如下图所示：

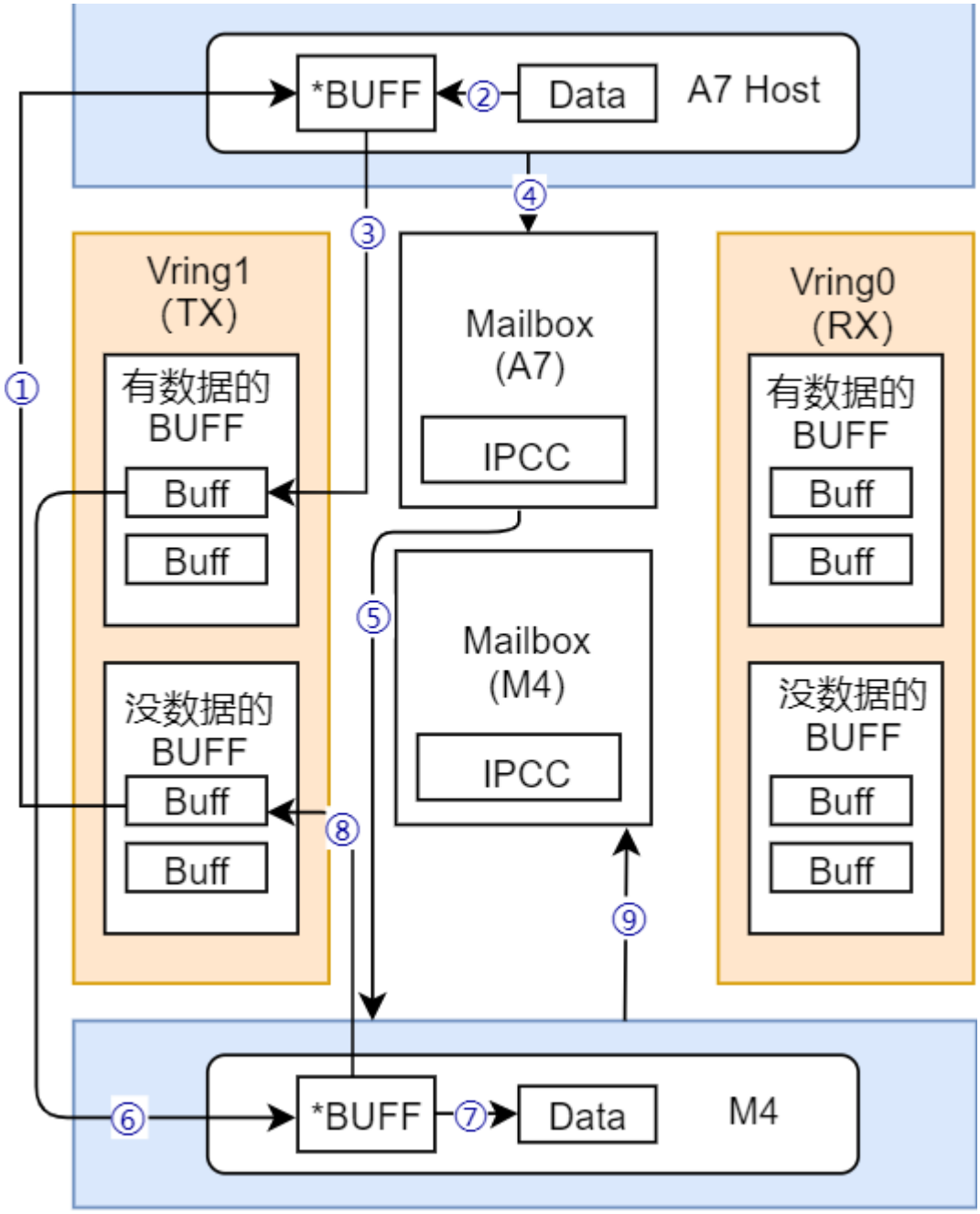


图 2.1.3.1 A7 发送数据 M4 图

注：有数据的 BUFF 表示要发送数据的队列，没数据的 BUFF 表示已经发送完数据的队列或者新建的空 BUFF。每个 BUFF 大小为 512 字节。每一次给 M4 发送数据最大为 512 字节，实际能接收字节为 496 字节，还剩下 16 字节应该是头部信息或者结构体(笔者猜测)。

如图 2.1.3.1 是 A7 给 M4 发送数据步骤如下：

1. 从 vring1 里分配一个 Buff(空 Buff)。
2. 用户把数据拷贝到此 Buff 里。
3. 把(1)的 Buff 移动到有数据的 BUFF 队列中。
4. A7 Host 去调用 Mailbox(A7)，Mailbox(A7)就去设置 IPCC 相关的寄存器。
5. Mailbox(M4)中发现来自 A7 Host 的信号，M4 处理器被告知有新数据可读。
6. 从 vring1 中读取数据。

7. 把数据转移到自己的 DATA 中。
8. 把空的 Buff 放回到 vring1 中的没有数据队列中。
9. 使用 Mailbox(M4)去控制 IPCC 寄存器，告诉 A7 处理器已经处理完数据了。

M4 给 A7 发送数据步骤也是一样的，M4 负责拷贝数据到 vring0 中，A7 从 vring0 读取数据。

第三章 基于 RPMsg 的异核通讯例程

3.1 M4 核

本章要用到的接口：IPCC、OPENAMP 和 USART3(用作 M4 的串口打印)。

3.1.1 IPCC 的配置

使用 STM32CubeIDE 新建一个工程，配置 IPCC 如下所示：

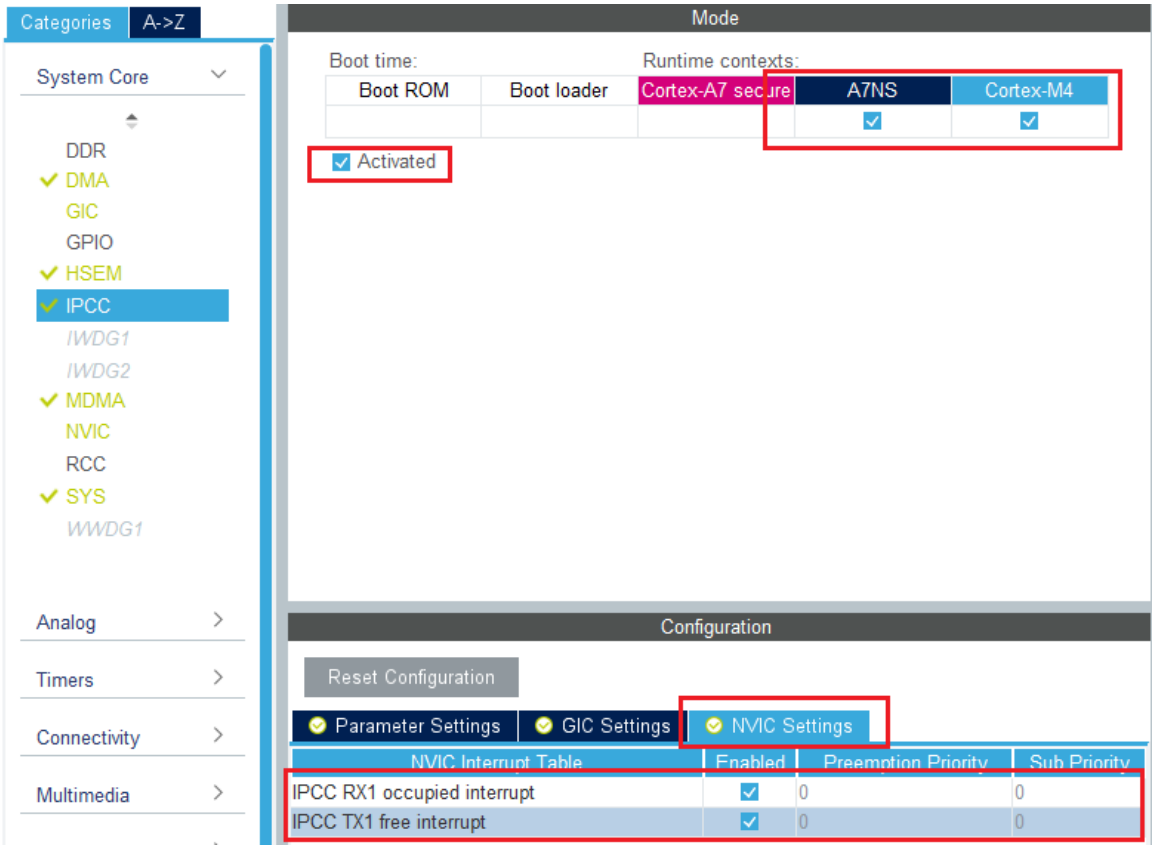


图 3.1.1.1 IPCC 配置

3.1.2 OPENAMP 的配置

OPENAMP 配置如下所示：

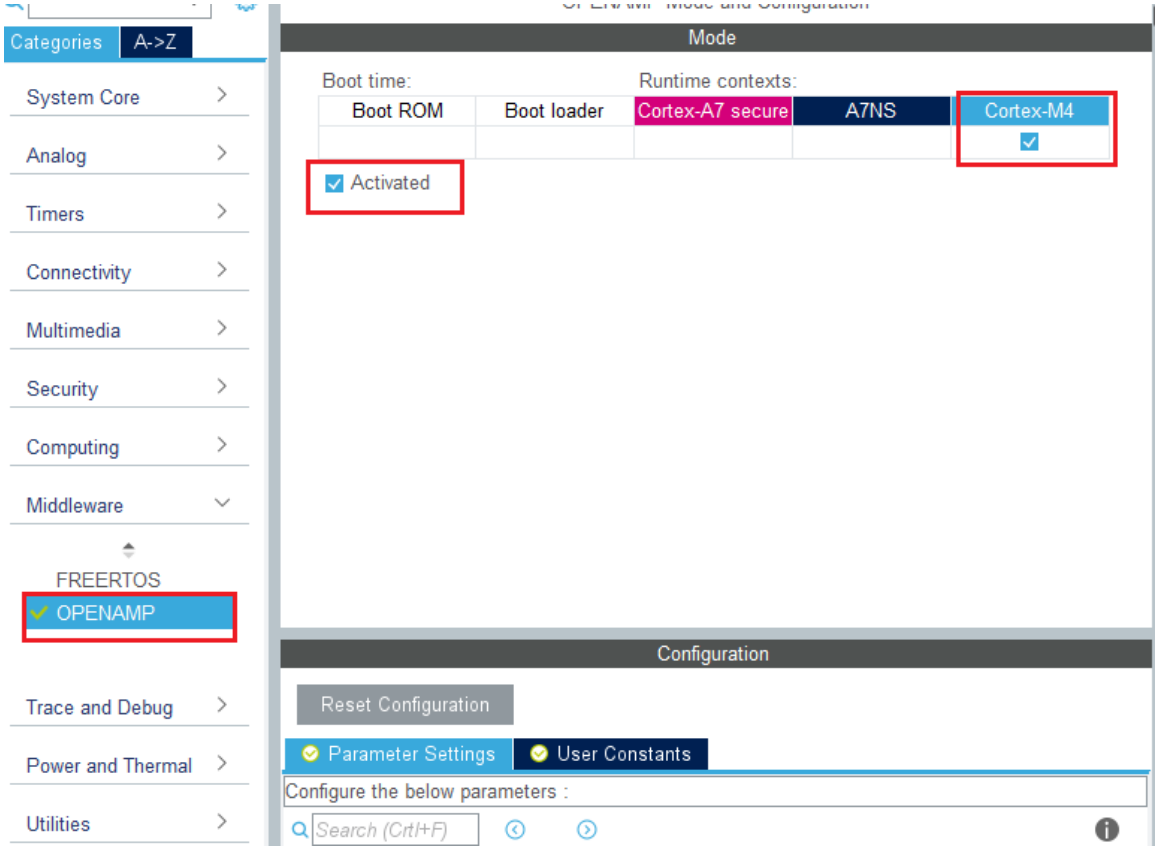


图 3.1.2 OPENAMP 配置

3.1.3 USART3 的配置

USART3 配置如下所示：

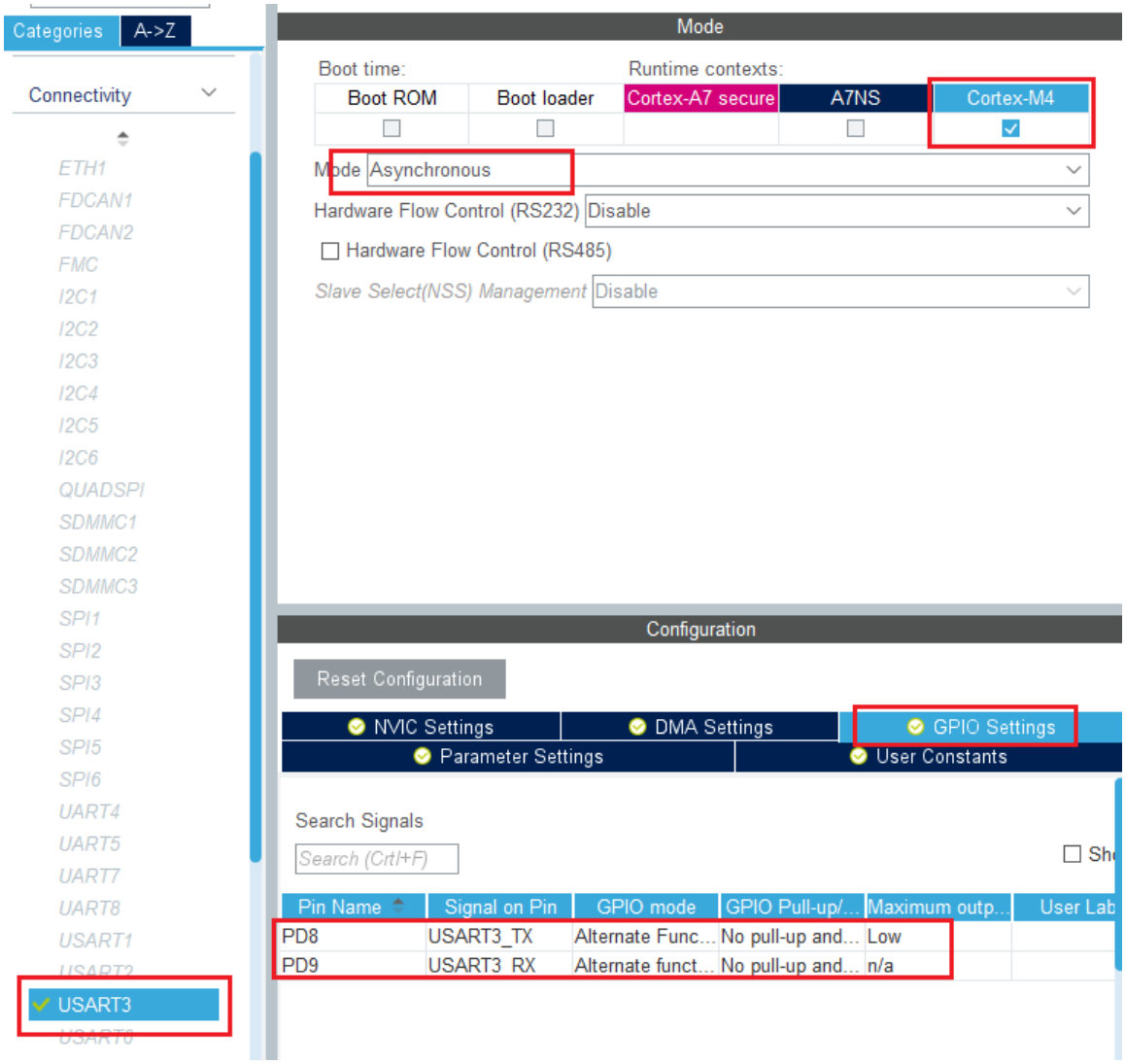


图 3.1.3.1 USART3 配置

配置保存生成对应的代码即可。

3.1.4 代码的编写

1. M4 通讯编写步骤

- 初始化 IPCC

```
static void MX_IPCC_Init(void)
```

函数参数和返回值都没有。

- 初始化 OPENAMP

```
int MX_OPENAMP_Init(int RPMsgRole, rpmsg_ns_bind_cb ns_bind_cb)
```

函数参数和返回值含义如下：

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：通常直接给 NULL。

返回值：负数，失败；0，成功。

- 设置回调函数，用作接收 A7 发过来的数据

```
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：

rp_chnl：rpmsg_endpoint 类型的结构体。

data：存放 A7 发过来的数据 BUFF。

len：data 数据的长度。

src：不知道有啥作用。

priv：可以设置私有数据，笔者没有用过。

返回值：负数，失败；0，成功。

- 把回调函数注册到 openamp 框架，

```
int OPENAMP_create_endpoint(struct rpmsg_endpoint *ept, const char *name,
                            uint32_t dest, rpmsg_ept_cb cb,
                            rpmsg_ns_unbind_cb unbind_cb)
```

函数参数和返回值含义如下：

ept：rpmsg_endpoint 类型的结构体。

name：此参数是用作和 A7 rpmsg 驱动进行匹配，和 A7 的名字一样。

dest：rpmsg 地址直接给 0xFFFFFFFF 就行。

cb：给回调函数的地址，把回调函数注册到 rpmsg 框架。

unbind_cb：直接给 NULL 就行。

返回值：负数，失败；0，成功。

- 轮询函数

```
void OPENAMP_check_for_message(void)
```

函数参数和返回值都没有。

- M4 给 A7 发送数据

```
static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
                             int len)
```

函数参数和返回值含义如下：

ept：rpmsg_endpoint 类型的结构体。

data：要发送数据的指针。

len：发送数据的长度。

返回值：负数，失败；0，成功。

```
#define OPENAMP_send  rpmsg_send
```

还可以使用 OPENAMP_send 宏，来发送数据。

2. 真正的代码编写

下面的代码添加都是在 main.c 文件修改。

```
/* USER CODE BEGIN PD */
#define RPMSG_SERVICE_NAE            "rpmsg-client-sample"
/* USER CODE END PD */
/* USER CODE BEGIN PV */
__IO FlagStatus rx_status = RESET;
uint8_t received_rpmsg[128];
/* USER CODE END PV */
```

RPMSG_SERVICE_NAME 宏用作和 A7 romsg 匹配。rx_status 用作标志位，有数据的时候设置 SET，没有数据的时候设置为 RESET。received_rpmsg 数组用作接收数据。

设置 usart3 当作打印串口：

```
/* USER CODE BEGIN 0 */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif
PUTCHAR_PROTOTYPE
{
    while ((USART3->ISR & 0X40) == 0);
    USART3->TDR = (uint8_t) ch;
    return ch;
}
/* USER CODE END 0 */
```

设置 rpmsg 的回调函数，用作处理数据 A7 数据，就是图 2.1.3.1 的第 7 步。

```
/* USER CODE BEGIN PFP */
static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data, size_t
len, uint32_t src, void *priv);
/* USER CODE END PFP */
/* USER CODE BEGIN 4 */
static int rx_callback(struct rpmsg_endpoint *rp_chnl, void *data, size_t
len, uint32_t src, void *priv)
{
  /* copy received msg, and raise a flag  */
  memcpy(received_rpmsg, data, len > sizeof(received_rpmsg) ?
sizeof(received_rpmsg) : len);
  printf("received_rpmsg=%s\r\n", received_rpmsg);
  rx_status = SET;
  return 0;
}
/* USER CODE END 4 */
```

最后我们设置 M4 给 A7 发送数据：

```
1   while (1)
2   {
3     OPENAMP_check_for_message();
4     /* USER CODE END WHILE */
5     if (rx_status == SET)
6       {
7         /* Message received: send back a message anwser */
8         rx_status = RESET;
9
10        if (++count < 100)
11        {
```

```
12          sprintf((char *)msg, "M4->A7 %02ld", count);
13        }
14        else
15        {
16          strcpy((char *)msg, "goodbye!");
17        }
18
19        if (OPENAMP_send(&resmgr_ept, msg, strlen((char *)msg) + 1) < 0)
20        {
21          printf("Failed to send message\r\n");
22          Error_Handler();
23        }
24
25        /* Print also the message to the trace */
26        printf("%s\r\n", msg);
27      }
28    /* USER CODE BEGIN 3 */
29  }
```

第 3 行，一直检查 A7 是否有数据发给 M4。

第 5 行，判断 rx_status 标志位是否等于 SET，当有数据的时候 rx_callback 回调函数就会设置 rx_status 为 SET。

第 19 行，调用 OPENAMP_send 发送数据给 A7。

详细的例程源码在：开发板光盘 A-基础资料→1、程序源码→2、Linux 驱动例程→30_RPMsg→ RPMsg.rar 压缩包里。

3.2 A7 核

3.2.1 设备树配置

在 M4 里已经使用了 USART3 所以要在 A7 核里做相关的配置，去设置 A7 的 usart3 为关闭状态，新建一个 stm32mp157d-atk-m4.dts 设置如下所示：

示例代码 3.2.1.1 stm32mp157d-atk-m4.dts 文件内容

```
1  // SPDX-License-Identifier: (GPL-2.0+ OR BSD-3-Clause)
2
3  #include "stm32mp157d-atk.dts"
4
5  &usart3 {
6      status = "disabled";
7  };
8
9  &m4_rproc {
10     m4_system_resourcec {
11         status = "okay";
12     };
```

```
13  };
14
15  &m4_usart3 {
16      status = "okay";
17  };
```

第 3 行，引用正点原子的出厂的设备树，出厂的设备树里开启了 m4_rproc 和 ipcc。

第 5~7 行，关闭 A7 核的 usart3。只要把 status 改为“disabled”。

第 9~13 行，m4_system_resourcec 是控制 M4 的资源，只要把 status 改为“disabled”。

第 15~17 行，把 usart3 资源分配给 M4，给 m4_usart3 节点追加 status 属性为“okay”，此节点定义在 stm32mp157-m4-srm.dtsi 文件里定义了。

3.2.2 驱动的编写

创建一个 RPMsg 目录把内核下的 samples/rpmsg/rpmsg_client_sample.c 文件拷贝到这个目录下。

接着去修改此文件，打开 rpmsg_client_sample.c，找到 rpmsg_sample_cb 函数，添加如下代码：

```
printk("MSG: %s rx_count = %d \n", (char *)data, ++idata->rx_count);
```

修改 MSG 宏定义为：

```
#define MSG       "A7->M4"
```

还要注释一部分打印信息代码，最后的结果如下所示：

```
16 #define MSG      "A7->M4"
17
18 static int count = 100;
19 module_param(count, int, 0644);
20
21 struct instance_data {
22     int rx_count;
23 };
24
25 static int rpmsg_sample_cb(struct rpmsg_device *rpdev, void *data, int len,
26                            void *priv, u32 src)
27 {
28     int ret;
29     struct instance_data *idata = dev_get_drvdata(&rpdev->dev);
30
31 //  dev_info(&rpdev->dev, "incoming msg %d (src: 0x%x)\n",
32 //           ++idata->rx_count, src);
33
34 //  print_hex_dump_debug(__func__, DUMP_PREFIX_NONE, 16, 1, data, len,
35 //                       true);
36     printk("MSG: %s rx_count = %d \n", (char *)data, ++idata->rx_count);
```

rpmsg_sample_cb 就是负责接收 M4 数据的，这个驱动很简单，没啥好讲的。

3.2.3 添加 Makefile 文件

新建 Makefile 文件，把下面内容拷贝进去：

示例代码 3.2.3.1 Makefile 文件

```
1  KERNELDIR := /home/liangwencong/linux/atk-mp1/kernel
2  CURRENT_PATH := $(shell pwd)
3
```

```
4   obj-m := rpmsg_client_sample.o
5
6   build: kernel_modules
7
8   kernel_modules:
9       $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) modules
10
11  clean:
12      $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

3.3 运行测试

把编译好的 RPMsg_CM4.elf、设备树和 rpmsg_client_sample.ko 文件拷贝到对应的目录下。创建一个 m4.sh 的脚本，方便以后调试 M4。脚本内容如下所示：

示例代码 3.3.1 m4.sh 文件内容

```
1   #!/bin/sh
2
3   rproc_class_dir="/sys/class/remoteproc/remoteproc0"
4   fmw_dir="/lib/firmware"
5
6   cd /sys/class/remoteproc/remoteproc0
7
8   if [ $1 == "start" ]
9   then
10          /bin/echo -n $2 > $rproc_class_dir/firmware
11          /bin/echo -n start > $rproc_class_dir/state
12  fi
13
14  if [ $1 == "stop" ]
15  then
16          /bin/echo -n stop > $rproc_class_dir/state
17  fi
```

还要给脚本对应的执行权限，运行以下代码即可：

```
chmod +x m4.sh
```

下面运行以下命令去测试 STM32MP157 的异核通讯：

```
insmod rpmsg_client_sample.ko //加载 RPMsg 驱动
./m4.sh start RPMsg_CM4.elf //加载 M4 固件和启动 M4
```

运行结果如下所示：


[root@ATK-stm32mp1:~$
[root@ATK-stm32mp1:~$ insmod rpmsg_client_sample.ko
[root@ATK-stm32mp1:~$ ./m4.sh start RPMsg_CM4.elf
这里是A7运行步骤
remoteproc remoteproc0: powering up m4
remoteproc remoteproc0: Booting fw image RPMsg_CM4.elf, size 2237588
mlahb:m4@10000000#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
virtio_rpmsg_bus virtio0: rpmsg host is online
virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x0
mlahb:m4@10000000#vdev0buffer: registered virtio0 (type 7)
rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: new channel: 0x400 -> 0x0!
remoteproc remoteproc0: remote processor m4 is now up
MSG: M4->A7 01 rx_count = 1
[root@ATK-stm32mp1:~$ MSG: M4->A7 02 rx_count = 2
MSG: M4->A7 03 rx_count = 3
MSG: M4->A7 04 rx_count = 4
MSG: M4->A7 05 rx_count = 5
MSG: M4->A7 06 rx_count = 6
MSG: M4->A7 07 rx_count = 7
MSG: M4->A7 08 rx_count = 8
A7接收到M4的数据
Cortex-M4 boot successful with STM32Cube FW version: v1.2.0
received_rpmsg=A7->M4
M4->A7 01
received_rpmsg=A7->M4
M4->A7 02
received_rpmsg=A7->M4
M4->A7 03
received_rpmsg=A7->M4
M4->A7 04
received_rpmsg=A7->M4
M4->A7 05
received_rpmsg=A7->M4
M4->A7 06
received_rpmsg=A7->M4
M4->A7 07
received_rpmsg=A7->M4
M4->A7 08
received_rpmsg=A7->M4
M4->A7 09
received_rpmsg=A7->M4
M4->A7 10
M4接收到A7的数据

图 3.3.1 异核通讯结果

第四章 虚拟串口的异核通讯

本章实验是使用 Linux 自带的驱动，此驱动代码在：drivers/rpmsg/rpmsg_tty.c。此驱动也是很简单的，在 rpmsg_client_sample 驱动下添加了一个 UART 框架，UART 框架把异核通讯封装了一层，用户层就可以通过调用 UART 进行异核通讯。实验目的，通过 UART 发送数据进行控制蜂鸣器。

4.1 M4 核

4.1.1 IDE 配置

本章实验还是按照第 3.1.1~3 小节配置，还要配置蜂鸣器。

4.1.2 代码的编写

1. M4 的 UART 异核通讯步骤

- IPCC 和 OPENAMP 初始化

  参考 3.1.4 小节

- UART 初始化

```
VIRT_UART_StatusTypeDef VIRT_UART_Init(VIRT_UART_HandleTypeDef *huart)
```

函数参数和返回值含义如下：

huart：VIRT_UART_HandleTypeDef 类型的结构体

返回值：0 表示成功，1 表示失败。

- 回调函数

```
void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
```

函数参数和返回值含义如下：

huart：VIRT_UART_HandleTypeDef 类型的结构体

返回值：0 表示成功，1 表示失败。

用作处理 A7 发送过来的数据。

- 把回调函数注册到 UART

```
VIRT_UART_StatusTypeDef VIRT_UART_RegisterCallback(
                                VIRT_UART_HandleTypeDef *huart,
                                VIRT_UART_CallbackIDTypeDef CallbackID,
                                void (* pCallback)(VIRT_UART_HandleTypeDef *_huart))
```

函数参数和返回值含义如下：

huart：VIRT_UART_HandleTypeDef 类型的结构体。

CallbackID：直接给 VIRT_UART_RXCPLT_CB_ID。

pCallback：回调函数

返回值：0 表示成功，1 表示失败。

- 检查函数

```
void OPENAMP_check_for_message(void)
```

函数参数和返回值都没有。

- M4->A7 数据

```
VIRT_UART_StatusTypeDef VIRT_UART_Transmit(VIRT_UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size)
```

函数参数和返回值含义如下：

huart：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。

返回值：0 表示成功，1 表示失败。

2. 真正的代码编写

下面的代码添加都是在 main.c 文件修改，如下示例所示：

示例代码

```
1  #include "main.h"
2  #include "openamp.h"
3
4  /* Private includes
----------------------------------------------------------*/
5  /* USER CODE BEGIN Includes */
6  #include "virt_uart.h"
7  /* USER CODE END Includes */
8
9  /* Private typedef
-----------------------------------------------------------*/
10 /* USER CODE BEGIN PTD */
11
12 /* USER CODE END PTD */
13
14 /* Private define
------------------------------------------------------------*/
15 /* USER CODE BEGIN PD */
16 #define MAX_BUFFER_SIZE RPMSG_BUFFER_SIZE   //512 字节
17 /* USER CODE END PD */
18
19 /* Private macro
-------------------------------------------------------------*/
20 /* USER CODE BEGIN PM */
21
22 /* USER CODE END PM */
23
24 /* Private variables
---------------------------------------------------------*/
25 IPCC_HandleTypeDef hipcc;
26
27 UART_HandleTypeDef huart3;
28
29 /* USER CODE BEGIN PV */
30 VIRT_UART_HandleTypeDef huart0;
31
```

```
32  __IO FlagStatus VirtUart0RxMsg = RESET;     //标志位，有数据就设置为 SET，
没数据就 RESET
33  uint8_t VirtUart0ChannelBuffRx[MAX_BUFFER_SIZE];    //接收 A7 数据的 buff，
大小为 512 字节
34  uint16_t VirtUart0ChannelRxSize = 0;    //接收 A7 数据长度
35
36  uint8_t BuffTx[MAX_BUFFER_SIZE];    //发送数据给 A7 的 BUFF，大小为 512 字节
37
38  #define MSG_BEEP_ON "beep_on"       //接收到“beep_on”,就打开蜂鸣器
39  #define MSG_BEEP_OFF "beep_off"     //接收到“beep_off”,就关闭蜂鸣器
40  /* USER CODE END PV */
41
42  /* Private function prototypes
-----------------------------------------------*/
43  void SystemClock_Config(void);
44  static void MX_GPIO_Init(void);
45  static void MX_IPCC_Init(void);
46  static void MX_USART3_UART_Init(void);
47  int MX_OPENAMP_Init(int RPMsgRole, rpmsg_ns_bind_cb ns_bind_cb); //
初始化 OPENAMP
48  /* USER CODE BEGIN PFP */
49  void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart);
50  /* USER CODE END PFP */
51
52  /* Private user code
---------------------------------------------------------*/
53  /* USER CODE BEGIN 0 */
54  #ifdef __GNUC__
55  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
56  #else
57  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
58  #endif
59  PUTCHAR_PROTOTYPE
60  {
61      while ((USART3->ISR & 0X40) == 0);
62      USART3->TDR = (uint8_t) ch;
63      return ch;
64  }
65
66  /* USER CODE END 0 */
67
68  /**
69    * @brief  The application entry point.
```

```
70    * @retval int
71    */
72  int main(void)
73  {
74    /* USER CODE BEGIN 1 */
75
76    /* USER CODE END 1 */
77
78    /* MCU
Configuration--------------------------------------------------------*/
79
80    /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
81    HAL_Init();
82
83    /* USER CODE BEGIN Init */
84
85    /* USER CODE END Init */
86
87    if(IS_ENGINEERING_BOOT_MODE())
88    {
89      /* Configure the system clock */
90      SystemClock_Config();
91    }
92
93    /* IPCC initialisation */
94     MX_IPCC_Init();  //初始化 IPCC
95    /* OpenAmp initialisation ---------------------------------*/
96    MX_OPENAMP_Init(RPMSG_REMOTE, NULL); //初始化 OPENAMP
97
98    /* USER CODE BEGIN SysInit */
99
100   /* USER CODE END SysInit */ //初始化 UART
101    printf("Virtual UART0 OpenAMP-rpmsg channel creation\r\n");
102    if (VIRT_UART_Init(&huart0) != VIRT_UART_OK) {
103      printf("VIRT_UART_Init UART0 failed.\r\n");
104      Error_Handler();
105    }
106     /* 把回调函数注册到 UART 里 */
107    if(VIRT_UART_RegisterCallback(&huart0, VIRT_UART_RXCPLT_CB_ID,
VIRT_UART0_RxCpltCallback) != VIRT_UART_OK)
108     {
109      Error_Handler();
```

```
110     }
111   /* Initialize all configured peripherals */
112   MX_GPIO_Init();   //蜂鸣器初始化
113   MX_USART3_UART_Init();    //usart3 初始化
114   /* USER CODE BEGIN 2 */
115
116   /* USER CODE END 2 */
117
118   /* Infinite loop */
119   /* USER CODE BEGIN WHILE */
120   while (1)
121   {
122       OPENAMP_check_for_message(); //查询 MailBox 状态
123     /* USER CODE END WHILE */
124       if (VirtUart0RxMsg) //当标志位为 SET，就去处理 A7 发送过来的数据
125       {
126           VirtUart0RxMsg = RESET; //设置标志位为 RESET
127             /*接收到数据为"beep_on"，就去打开蜂鸣器，发送数据告诉 A7，已经打开了
蜂鸣器。*/
128           if (!strncmp((char *)VirtUart0ChannelBuffRx, MSG_BEEP_ON,
strlen(MSG_BEEP_ON)))
129           {
130               strcpy((char *)BuffTx, "m4:beep on\n");
131               VIRT_UART_Transmit(&huart0, BuffTx, strlen((const char
*)BuffTx));
132               HAL_GPIO_WritePin(beep_GPIO_Port, beep_Pin,
GPIO_PIN_RESET);
133           }
134             /*接收到数据为"beep_off"，就去打开蜂鸣器，发送数据告诉 A7，已经关闭
了蜂鸣器*/
135           if (!strncmp((char *)VirtUart0ChannelBuffRx, MSG_BEEP_OFF,
strlen(MSG_BEEP_OFF)))
136           {
137               strcpy((char *)BuffTx, "m4:beep off\n");
138               VIRT_UART_Transmit(&huart0, BuffTx, strlen((const char
*)BuffTx));
139               HAL_GPIO_WritePin(beep_GPIO_Port, beep_Pin,
GPIO_PIN_SET);
140           }
141
142           memset(VirtUart0ChannelBuffRx, 0 ,VirtUart0ChannelRxSize);
143           memset(BuffTx, 0 ,strlen((const char *)BuffTx));
144       }
```

```
145
146     /* USER CODE BEGIN 3 */
147   }
148   /* USER CODE END 3 */
149 }
```

看看回调函数是如何处理数据的，如下示例代码所示：

示例代码 4.1.2.2 VIRT_UART0_RxCpltCallback 函数

```
1   void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huart)
2   {
3
4       printf("Msg received on VIRTUAL UART0 channel:  %s \n\r", (char *)
huart->pRxBuffPtr);
5
6       /* copy received msg in a variable to sent it back to master processor
in main infinite loop*/
7       VirtUart0ChannelRxSize = huart->RxXferSize < MAX_BUFFER_SIZE?
huart->RxXferSize : MAX_BUFFER_SIZE-1;
8       memcpy(VirtUart0ChannelBuffRx, huart->pRxBuffPtr,
VirtUart0ChannelRxSize);
9       VirtUart0RxMsg = SET;
10  }
```

详细的例程源码在：开发板光盘 A-基础资料→1、程序源码→2、Linux 驱动例程→30_RPMsg→ RPMsg_UART.rar

4.2 A7 核

这里 A7 核没有啥要修改的直接使用第三章的设备树和 uImage 即可。

4.3 运行测试

把 RPMsg_UART_CM4.elf 拷贝到/lib/firmware/目录，运行测试命令如下所示：

```
./m4.sh start RPMsg_UART_CM4.elf //开启 M4 核
cat /dev/ttyRPMSG0 & //接收 M4 发过来的数据
echo 'beep_on' >/dev/ttyRPMSG0 //通过 ttyRPMSG0 来发送数据，启动 M4 控制的蜂鸣器
echo 'beep_off' >/dev/ttyRPMSG0 //关闭 M4 控制的蜂鸣器
```

运行测试结果如下所示：

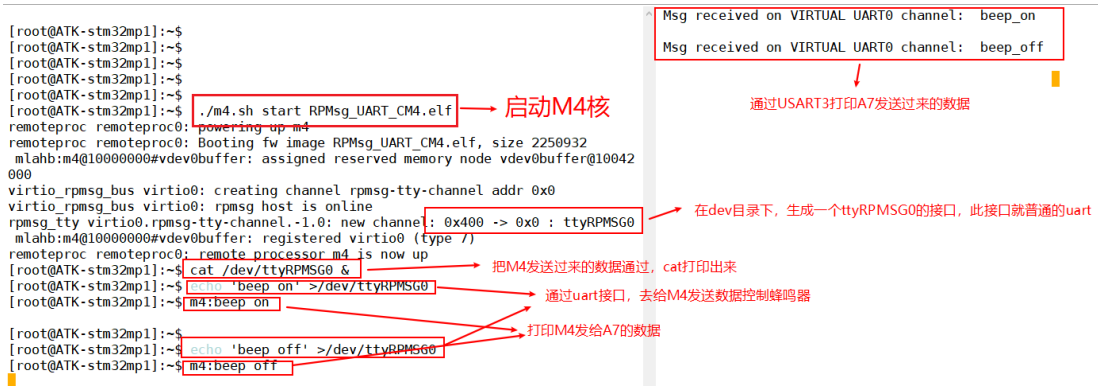


图 4.3.1 运行测试结果

大家可以看下 virt_uart.c 文件，只是封装了一层，底层的收发数据也是和第三章一样，这样我们可以自己封装成网络设备、字符设备(drivers/rpmsg/rpmsg_char.c 文件就是封装成字符设备，笔者没有测试过)和块设备等等。

附录-A