Tina Linux 异构通信框架

开发指南

版本号: 1.12

发布日期: 2026.5.31

版本号	日期	制/修订人	内容描述
1.0	2023.04.13	AWA1809	初始版本
1.1	2024.02.27	AWA1982	增加共享中断章节
1.2	2024.05.30	AWA1982	更新文档，增加 AMP 新功能说明
1.3	2024.10.18	AWA1982	参考最新 SDK 更新文档
1.4	2024.11.19	AWA2298	新增修改从核运行内存基地址和空间说明，增加 rpmsg_test/ rpbuf_test 测试结果说明，增加从核 coredump 使用说明
1.5	2025.02.26	AWA2298	增加 trace event 使用说明，增加 FAQ
2025.04.09	AWA2298	OTA 说明	增加 gdb 调试从核 coredump data 使用说明，FAQ 增加从核启动说明以及
1.7	2025.04.18	AWA1982	增加外设复位功能使用说明
1.8	2025.06.13	AWA2298	基于 Tina5.0 优化补充文档
1.9	2025.11.28	AWA1982	更新 AMP 系统相关术语说明并增加 AMP 系统架构框图
1.1	2025.12.23	AWA2241	更新 V861 支持
1.11	2026.5.26	AWA2298	更新 rpmsg v2、share irq 内容说明
1.12	2026.5.31	AWA2298	增加从核各种状态下的 coredump 数据处理说明

言

1.1 文档简介

介绍全志的异构通信框架，该框架主要提供 AMP 场景下处理器的生命周期管理功能和处理器核间通信功能，由于 AMP 场景下从核一般运行 RTOS 系统，因此本文中也有涉及 RTOS 的内容。

1.2 目标读者

异构通信框架的开发、使用人员。

1.3 适用范围

适用于主核运行 Linux 系统，从核运行 RTOS 系统的芯片平台。

1.3.1 标志说明


注意

- 提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。


说明

为准确理解文中指令、正确实施操作而提供的补充或强调信息。


技巧

小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

1.3.2 地址与数据描述方法约定

本文档在描述地址、数据时遵循如下约定：

表 1-1: 地址与数据描述方法约定

符号	例子	说明
0x	0x0200	地址或数据以 16 进制表示。
0b	0b010	数据采用二进制表示 (寄存器描述除外)。
X	00X	数据描述中，X 代表 0 或 1，例如，00X 代表 000 或 001。

位约定

本文档在描述数据容量（如 NAND 容量）时，单位词头代表的是 1024 的倍数；描述频率、数据速率等时则代表的是 1000 的倍数。具体如下：

表 1-2: 数值单位约定

类型	符号	对应数值
数据容量	1 K	1024
数据容量	1 M	1 048 576
数据容量	1 G	1 073 741 824
频率，数据速率等	1 K	1000
频率，数据速率等	1 M	1 000 000

符号		对应数值
频率，数据速率等	1 G	1 000 000 000

1.4 相关术语介绍

表 1-3: 异构框架相关术语

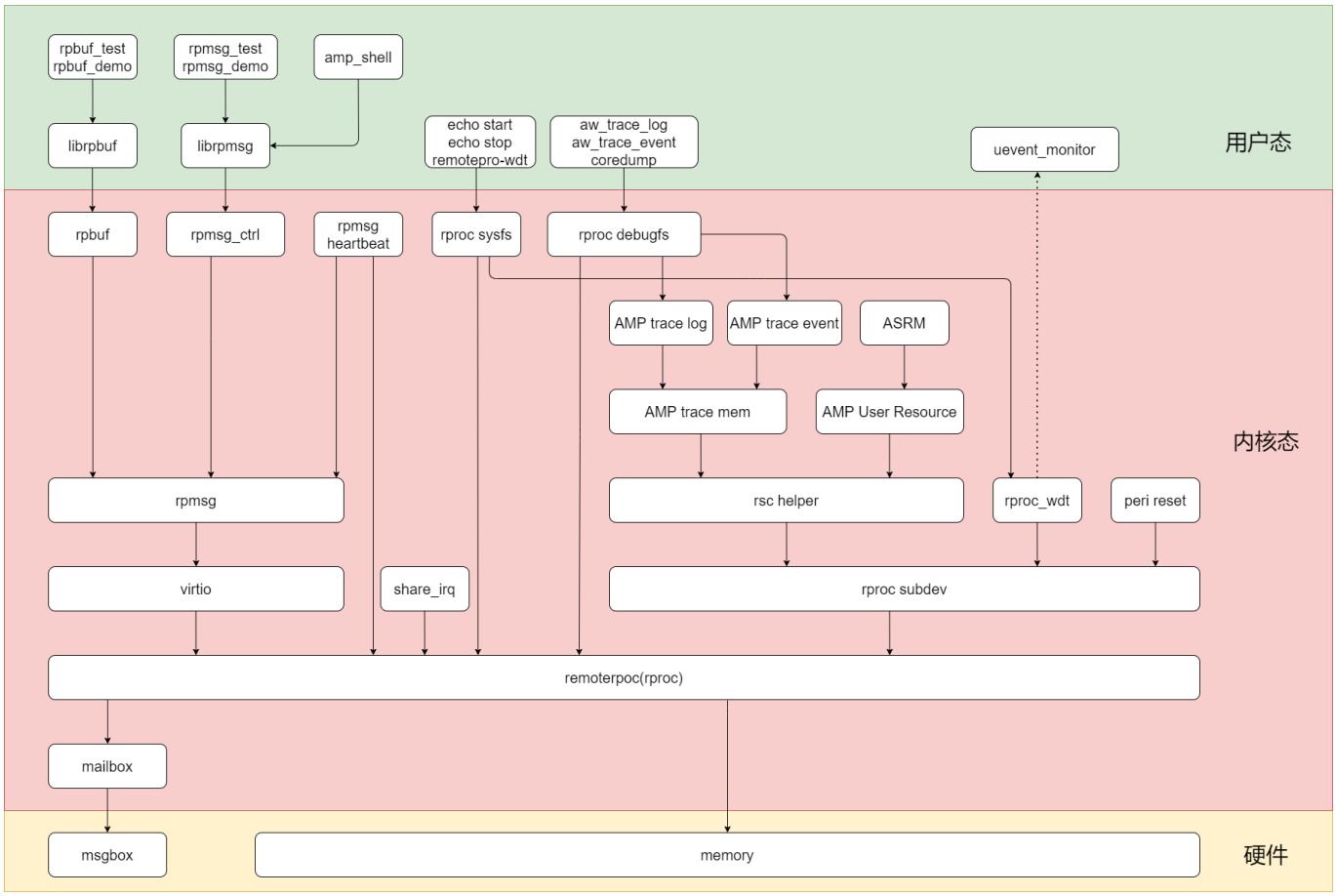
术语	解释说明
SMP	对称多处理 (Symmetric Multi-processing)
AMP	非对称多处理 (Asymmetric Multi-processing)
OpenAMP	可在裸机或 RTOS 上运行的用于 AMP 场景下处理器核间通信的第三方开源软件。全称为 Open Asymmetric Multi-Processing
	CPU 核 (SMP 系统) 的集群组成一个 AMP 用户，其运行的软件可能在整个生命周期内会发生变化，例如早期先运行 bootloader 软件，后期运行操作系统软件
主核	AMP 系统内主要的 AMP 用户 (此 AMP 用户上运行的软件若发生 crash 则必须重启整个 AMP 系统)，一般在整个 AMP 系统硬件复位后最先运行，一般运行 Linux 系统且处理的任务较多
从核	AMP 系统内被主核管理生命周期的 AMP 用户 (此 AMP 用户上运行的软件 crash 后无需重启整个 AMP 系统，可被主核单独重启)，一般在主核之后运行，一般运行 RTOS 系统或裸机软件且处理的任务较少
remoteproc	远端处理器 (Remote Processor)，一般情况下指代从核，在 OpenAMP 框架和 Linux remoteproc 子系统里表示通信时对端的处理器
RPMsg	remoteproc message，AMP 场景下处理器核间的一种标准通信方式
RPBuf	remoteproc buffer，全志实现的 AMP 场景下处理器核间进行大量数据传输的通信方式，可突破 rpmsg 单次传输时的数据长度限制
RV	集成在芯片内部的 RISC-V 架构的处理器核心，部分芯片平台不一定有
DSP	集成在芯片内部的 Xtensa 架构的处理器核心，部分芯片平台不一定有

AMP 系统架构概述

由于 AMP 系统内主核和从核这 2 个 AMP 用户具有较大差异，因此全志目前在主核上运行的 Linux 系统环境和从核上运行 RTOS 系统环境下分别实现了一套 AMP 系统软件。

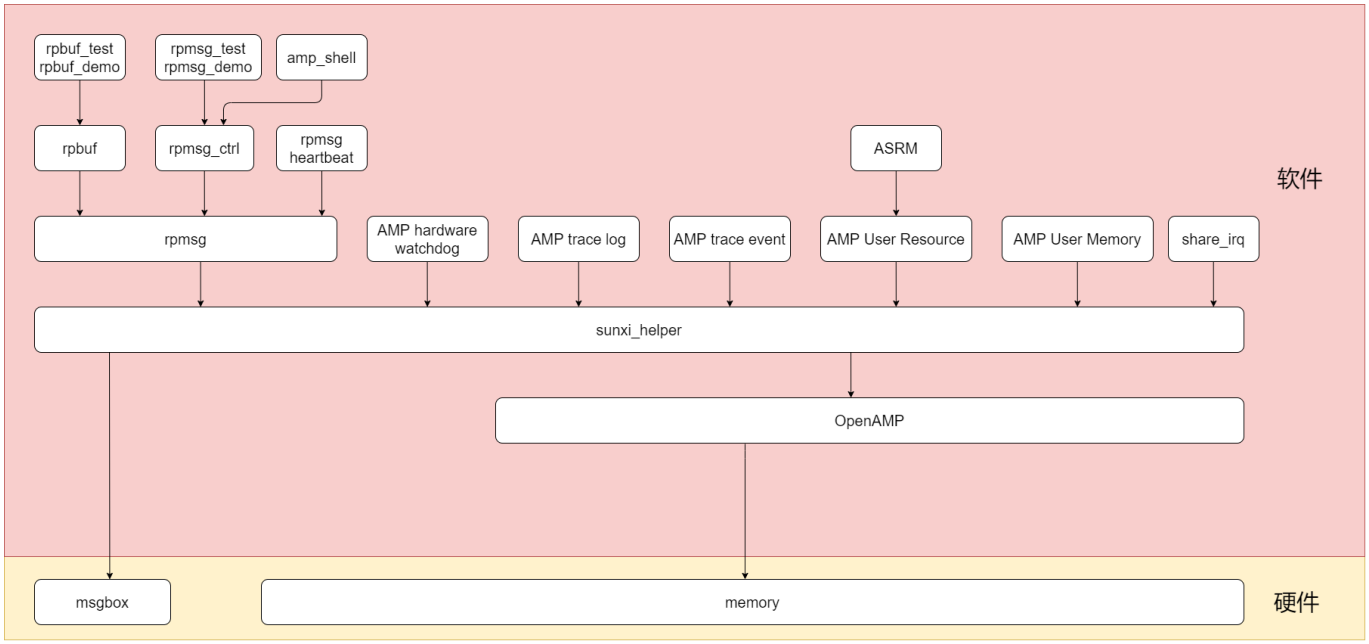
Linux 系统环境下的 AMP 系统架构框图如下：

图 2-1: Linux 系统环境下的 AMP 系统架构架构框图



RTOS 系统环境下的 AMP 系统架构框图如下：

图 2-2: RTOS 系统环境下的 AMP 系统架构架构框图



DK 使用

目前 SDK 里集成了主核 Linux 编译环境与从核 RTOS 编译环境，下面将分别介绍两个编译环境的使用方法。

3.1 Linux 编译环境

在 SDK 下，首先执行 source build/envsetup.sh 配置编译环境，然后输入 lunch 命令选择具体方案。之后就可以使用一些快捷命令来帮助开发，此处介绍 Linux 编译环境下常用的快捷命令。

1. ckernel, m kernel_menuconfig, mkernel：分别对应进入到内核目录，配置内核，单独编译内核。
2. m menuconfig：配置软件包。
3. cboot0, mboot0：进入 boot0 目录，单独编译 boot0。
4. cuboot, muboot：进入 uboot 目录，单独编译 uboot 相关固件。

oot0，系统相关的固件，根文件系统等。

入板级配置目录，这里主要存放板级的设备树，分区等配置文件。

7. p：打包命令，将编译后的东西打包成固件。

注意

m 命令是否编译 RTOS 相关固件由对应方案的 BoardConfig.mk 文件里的 LICHEE_RTOS_PROJECT_NAME 变量决定，可参考“Linux 编译环境下编译 RTOS”章节

3.2 RTOS 编译环境

SDK 中的 rtos 目录即为 RTOS 编译环境，进入 rtos 目录，执行 source envsetup.sh 配置 RTOS 编译环境 (为防止 RTOS 编译环境和 Linux 编译环境中的环境变量冲突，建议单独开一个新的终端来配置 RTOS 编译环境)，然后输入 lunch_rtos 命令选择具体方案。之后就可以使用一些快捷命令来帮助开发，此处介绍 RTOS 编译环境下常用的快捷命令。

切换到 rtos/lichee/rtos

2. cdsp：快速切换到 rtos/lichee/dsp 目录。
3. mrtos menuconfig：RTOS 配置界面。
4. mrtos：编译 RTOS。

在 RTOS 编译环境下编译后生成固件的路径为：

```
RV核：rtos/lichee/rtos/build/<project>/img/rt_system.elf
DSP核：rtos/lichee/dsp/out/<chip>/evb1/<chip>_dsp0_evb1.bin
```

其中 project 为方案名，例如：mr536_e907_evb1，chip 为芯片名，例如：mr536

另外目前在 RTOS 编译环境下单独编译出来的 RV 核固件未被 strip，文件大小比较大，需要对固件进行 strip，可在 rtos/lichee/rtos 目录执行如下命令进行 strip：

```
tools/riscv64-elf-x86_64-20201104/bin/riscv64-unknown-elf-strip build/<project>/img/rt_system.elf
```

若需要打包到最终的烧录固件里则还需要在打包前手动拷贝从核固件到具体方案的 bin 目录，具体 bin 目录的路径请参考“SDK 里的固件”章节。

由于在 RTOS 编译环境下获取最终的 RV 核固件的整个流程至少需要两步：编译、strip，而特殊情况下还需要做拷贝操作，建议不是非常熟悉 RTOS 编译环境的客户使用 Linux 环境来编译 RTOS，其可自动处理 strip 和拷贝操作。

技巧

一般在 RTOS 编译环境里编译固件主要用于运行时单独更新从核固件这种场景，若需要使用烧录方式来更新从核固件，可参考“Linux 编译环境下编译 RTOS”章节。

编译环境下编译

最新 SDK 支持在 Linux 编译环境下直接编译并拷贝 RTOS 固件到对应方案的 bin 目录，前提是需要在对应方案的 BoardConfig.mk 文件里增加 LICHEE_RTOS_PROJECT_NAME 变量，以 mr536_evb1 方案为例，需要在 BoardConfig.mk 文件里增加如下内容：

```
LICHEE_RTOS_PROJECT_NAME:=mr536_e907_evb1
```

增加后在 Linux 编译环境下执行 make 命令或 m 命令时会先编译 RTOS 固件并拷贝到 bin 目录，也可执行如下命令单独编译 RTOS 并拷贝到 bin 目录：

```
./build.sh rtos
```

另外还支持如下命令：

- 修改 RTOS 配置：./build.sh rtos menuconfig

编译中间文件：./build.sh rtos clean

技巧

上述在 Linux 编译环境下单独编译 RTOS 的相关命令还可使用 mrtos 系列命令，比如 mrtos、mrtos menuconfig 和 mrtos clean。

核固件

4.1 固件文件命名

目前 SDK 支持的从核架构有 RISC-V 和 Xtensa 架构，分别对应于 RV 核和 DSP 核，其固件文件分别命名为 amp_rvx.bin 和 amp_dspx.bin，其中 x 表示从核的 ID，此 ID 是架构范围内唯一，并不是整个芯片范围内唯一，也即若芯片内集成了不同架构的从核 (1 个 RV 核和 1 个 DSP 核)，则其固件命名为 amp_rv0.bin 和 amp_dsp0.bin，而不是 amp_rv0.bin 和 amp_dsp1.bin(或是 amp_rv1.bin 和 amp_dsp0.bin)

4.2 固件文件格式

虽然上面提到固件文件的后缀名为.bin，但目前 SDK 里使用从核固件文件实际上是经过 strip 后的 ELF 文件。

4.3 固件保存路径

的固件

SDK 里从核固件保存的路径为如下几个模式 (pattern)：

```
bin/*.bin
configs/*/bin/*.bin
configs/*/*/bin/*.bin
configs/*/*/*/bin/*.bin
```

这些路径模式前面还需要加上 device/config/chips/才是相对于 SDK 根目录的相对路径，其中后面路径模式的文件若存在，则会覆盖前面的文件。

以 MR527 和 MR536 为例，MR527 的从核固件路径为 device/config/chips/mr527/configs/evb/bin/amp_rv0.bin，满足第二种路径模式`configs/*/bin/*.bin`；MR536 从核固件的路径为 device/config/chips/mr536/bin/amp_rv0.bin，满足第一种路径模式`bin/*.bin`。

的固件

设备上的从核固件一般保存在文件系统里，具体为/lib/firmware 目录下，一般用于从核在 Linux 应用阶段启动并初始化的情况。若从核在 bootloader 阶段 (一般是 u-boot) 启动，由于 u-boot 目前不支持读取 Linux 系统使用的文件系统，则需要将固件保存到分区里，例如对与 RV 从核，会从核固件烧录在 riscv0 分区或 riscv0-r 分区。


注意

若使用 OpenWRT 构建 rootfs，不管从核固件是保存在文件系统里还是分区里，都需要在主核 Linux 环境下启用软件包配置 CONFIG_PACKAGE_amp-firmware。

4.4 固件更新

从核固件更新主要有如下 2 种方式

更新

构建时更新固件主要是单独编译 RTOS 固件并将其打包进烧录固件中，然后通过烧录方式来更新从核固件，可参考“Linux 编译环境下编译 RTOS”章节。

4.4.2 运行时更新

运行时更新固件一般在 Linux 系统里操作，使用 adb push 命令将新固件上传到设备文件系统里，根据保存从核固件的方式执行对应的操作：

- 若从核固件保存在文件系统里，直接拷贝新固件到/lib/firmware 目录里替换掉对应的文件即可
- 若从核固件保存在分区里，则可通过 dd 命令更新，例如更新第一个 RV 核固件的命令：dd if=amp_rv0.fex of=/dev/by-name/riscv0 bs=1024。对于 UBI 文件系统，不能使用 dd 命令更新，只能使用 ubi 工具命令更新从核系统镜像，如下是更新 RV 从核系统镜像：

```
nux:/#ls/dev/by-name/riscv0-ahl
    lrwxrwxrwx    1 root     root          11 Jan  1 11:06 /dev/by-name/riscv0 -> /dev/ubi0_7
    root@TinaLinux:/# ubiupdatevol /dev/ubi0_7 /mnt/UDISK/amp_rv0.fex
```

技巧

可通过内核日志确认从核固件是保存在分区里还是文件系统里，若内核日志中打印 “load fw from filesystem, filename: amp_rv0.bin” 类似日志，则表明从核固件是保存在文件系统里的，否则是保存在分区里。

核启动和初始化

首先说明下从核启动和从核初始化的概念，从核的启动指的是让从核开始运行代码，而从核的初始化则指的是 Linux remoteproc 框架初始化 remoteproc 的操作，完成初始化后才可以和从核通信。

目前 SDK 里支持如下几种从核启动和初始化方式：

1. 从核在 bootloader 阶段启动，在 Linux **内核**阶段初始化
2. 从核在 bootloader 阶段启动，在 Linux **应用**阶段初始化
3. 从核在 Linux 应用阶段启动并初始化

一般主要使用第 1 种方式和第 3 种方式，当前 SDK 默认使用第 1 种方式。

5.1 bootloader 启动从核

当前 SD[illegible]方案会在
实现了快启功能的平台则会在 boot0 阶段启动从核 (部分平台甚至 boot0 直接启动 Linux 内核，不会有 uboot 阶段)。

uboot 阶段启动从核主要需要配置 u-boot 启动命令、从核固件分区信息以及从核 RTOS 端 OpenAMP 框架等待主核 Linux 端初始化从核，而在 boot0 阶段启动从核时一般会从 boot package 里获取从核固件，且平台对应的 boot0 默认会启动从核，故此处不详细展开说明 boot0 相关的配置。

5.1.1 配置信息

5.1.1.1 主核 u-boot 的 menuconfig 配置

采用 u-boot 启动从核的方法，主核 Linux 环境下 u-boot 需要选中对应配置来提供启动从核的命令，SDK 中已默认选中，此章节对所需配置进行介绍。

[illegible] menuconfig

```
CONFIG_CMD_SUNXI_BOOTRV=y
CONFIG_BOOT_RISCV=y
```

5.1.1.2 从核固件的分区信息和从核启动命令

以 MR536 平台为例，执行 cconfigs 进入方案配置目录，修改该目录中的两个文件 openwrt/sys_partition.fex 和 openwrt/env-5.15.cfg。部分旧平台可能没有 openwrt 目录，可修改对应 kernel 的版本目录下的对应文件，例如：MR527 对应修改 linux-5.15/sys_partition.fex 和 linux-5.15/env-5.15.cfg。

1. sys_partition.fex 中添加从核固件的分区信息，内容如下：

```
[partition]
    name         = riscv0
= 2048
    downloadfile = "amp_rv0.fex"
    user_type    = 0x8000
```

技巧

其中 size 以 512 对齐，可根据实际的从核固件文件大小调整。当设备使用 ubi 文件系统时，riscv0 分区需要以 496 对齐。

2. env-5.15.cfg 文件中添加启动从核的命令并修改 bootcmd 环境变量，内容如下：

```
boot_riscv=bootrv 46000000 200000 0 riscv0 riscv0-r
bootcmd=run boot_riscv sunxi_pre_cmd boot_normal
```

u-boot 中 bootrv 命令的参数分别为固件文件加载内存地址、加载内存区域大小、从核 ID、从核固件分区名、从核固件备份分区名，其中加载内存区域大小需要大于等于从核固件大小。

3. 对于 MR153/MR536 等使用了安全 env 的平台，只能在 env-xx.cfg 文件中修改 RV 的运行加载地址和加载内存区域大小，boot_riscv、bootcmd 启动配置则定义在 uboot 的 bsp/configs 对应的平台的 CONFIG_SUNXI_BOOT_ENV_STRING 配置中。如下是 MR153 安全 env 的配置：

```
~/workspace/tina_mr153/brandy/brandy-2.0/u-boot-2023/bsp/configs$ grep -rnwHI CONFIG_SUNXI_BOOT_ENV_STRING ./
./sun8iw22p1_mr153_nor_defconfig:117:CONFIG_SUNXI_BOOT_ENV_STRING="preboot=sunxi_get_kern_env;sunxi_preboot\n bootcmd=run
     boot_riscv sunxi_pre_cmd boot_normal\n sunxi_pre_cmd=${sunxicmd};sunxi_update\n sunxicmd=run setargs_nand\n boot_normal=
     sunxi_flash read 0x40007800 ${boot_partition};bootm 0x40007800 ${sunxi_ramd_addr} ${sunxi_fdt_addr}\n boot_riscv=bootrv ${
     rv_fw_load_addr} ${rv_fw_load_size} 0 ${riscv_partition} ${riscv-r_partition}\n boot_recovery=\n boot_fastboot=fastboot\n
     altbootcmd=sunxi_switch_system\n"

ina_mr153/device/config/chips/mr153/configs/evb1$grep -rnEHI "rv_fw_load_size=|rv_fw_load_addr=|riscv_partition=|
     riscv-r_partition=" ./
./openwrt/env-5.15.cfg:20:riscv_partition=riscv0
./openwrt/env-5.15.cfg:21:riscv-r_partition=riscv0-r
./openwrt/env-5.15.cfg:22:rv_fw_load_addr=0x44000000
./openwrt/env-5.15.cfg:23:rv_fw_load_size=0x300000
```

5.1.1.3 从核 RTOS 环境配置

从核在 u-boot 阶段启动时，RTOS 端 OpenAMP 框架中部分逻辑是需要等待 Linux 端初始化从核后才能继续执行的，因此从核 RTOS 环境需要启用配置 CONFIG_SLAVE_EARLY_BOOT。

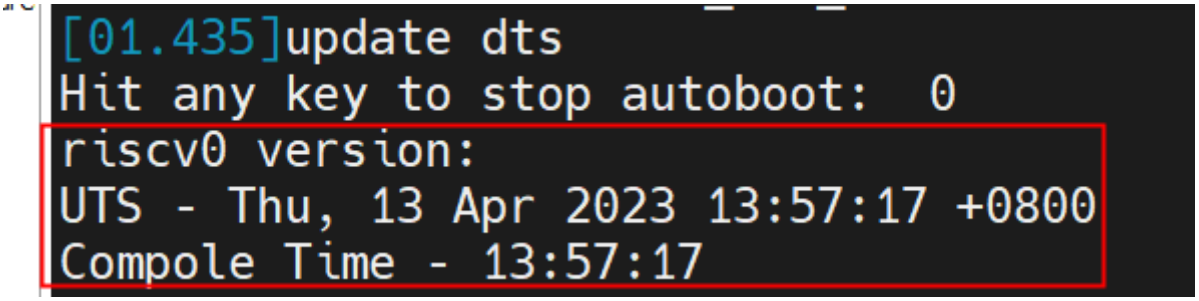
 注意

从核 RTOS 端 OpenAMP 框架默认是放在线程中初始化的，在开启上述配置后会导致用户程序先于 OpenAMP 框架初始化完成前运行 (不开启可能，但概率较低)，用户编写代码时需要注意这一点。

5.1.2 u-boot 启动从核现象

u-boot 启动从核时，会打印如下从核固件编译时间的日志：

图 5-1: uboot 启动 RV 的 logs



 注意

目前 boot0 正常启动从核时不会有任何打印，只能根据从核串口是否在 u-boot 之前或 kernel 运行之前有打印来判断 boot0 是否有启动从核。

5.2 Linux 启动和初始化从核

Linux 里启动和初始化从核的操作可分为如下 2 种组合：

- 启动和初始化从核，一般用于 bootloader 阶段未启动从核的场景
- 初始化从核，一般用于 bootloader 阶段已启动从核的场景

nux 内核只启动从核但不初始化从核的情况。

5.2.1 remoteproc 状态

Linux remoteproc 框架里目前规定 remoteproc 有 7 种状态，分别为如下：

- offline：关机状态
- suspended：休眠状态
- running：运行状态
- crashed：crash 状态，也即 remoteproc 进入了异常状态，需要恢复
- deleted：删除状态，一般不会出现这种情况
- attached：attached 状态，表明 remoteproc 已在更早的阶段启动，且 Linux 已初始化 remoteproc
- detached：detached 状态，表明 remoteproc 已在更早的阶段启动，比如 bootloader 阶段

到 offline、running、attached 以及 detached 这 4 种状态，由于目前 SDK 默认在 bootloader 阶段启动从核，在 Linux 内核阶段初始化从核，因此在主核 Linux 系统完全启动后，从核的状态为 attached。

5.2.2 remoteproc 生命周期管理相关命令

remoteproc 生命周期管理一般会用到 2 个命令，分别用于 start 和 stop remoteproc，具体如下：

- start remoteproc：echo start > /sys/class/remoteproc/remoteprocx/state
- stop remoteproc：echo stop > /sys/class/remoteproc/remoteprocx/state

上述命令里的 x 表示 remoteproc 的 ID，在 Linux 系统 remoteproc 框架范围内唯一，和前面 “固件文件命名” 章节里提到的从核 ID 有区别，也即若某个具体芯片平台有多个不同架构的从核，此 remoteproc ID 也不会重复。

需要注意的是在 start remoteproc 时，会根据 remoteproc 当前处于的状态执行不同的操作，若 remoteproc 处于 offline 状态，则会执行从核启动和从核初始化操作，执行完成后 remoteproc 处于 running 状态，若 remoteproc 处于 detached 状态 (表明从核已在 bootloader 阶段启动和从核初始化操作，执行完成后从核处于

另外查看 remoteproc 状态可执行命令 cat /sys/class/remoteproc/remoteprocx/state，还可通过命令 cat /sys/class/remoteproc/remoteprocx/name 查看对应的 remoteproc 在 dts 中的节点名，根据节点名来识别具体的从核。

5.2.3 bootloader 阶段已启动从核

此种情况下 Linux 只需要初始化从核即可，在内核阶段初始化从核只需要对应的 remoteproc 节点里增加 auto-boot 空属性，应用阶段初始化从核则只需要 start remoteproc 即可。

5.2.4 bootloader 阶段未启动从核

此种情况下一般从核固件会保存在文件系统里，在应用阶段启动并初始化从核，只需执行 start remoteproc 对应的命令即可。

此种情况下虽然增加 auto-boot 属性也可在内核阶段尝试启动并初始化从核，但由于 remoteproc 框架初始化比较早，此时文件系统还无法正常使用，所以会出现启动失败的情况，一般不会在此种情况下让内核去启动和初始化从核。

6 模块节点的使用说明

在 remoteproc 框架中的 remoteproc_debugfs.c 和 remoteproc_sysfs.c 中分别创建了 debugfs 和 sysfs 节点。

6.1 sysfs 节点

在 remoteproc 框架中向用户提供了 **coredump、recovery、firmware、state、name** 节点，其路径如下：

```
/sys/class/remoteproc/remoteproc0/state
/sys/class/remoteproc/remoteproc0/name
/sys/class/remoteproc/remoteproc0/firmware
/sys/class/remoteproc/remoteproc0/recovery
/sys/class/remoteproc/remoteproc0/coredump
```

其作用分别为：

- **state**：通过输入不同的命令来控制远程处理器的状态；
- **name**：显示远程处理器的名称；
- **firmware**：通过输入来修改固件名称，在修改了固件名称后，需要将同名的固件放置在主核 Linux 端的/lib/firmware 目录下，才可以 start 启动改名后固件；
- **recovery**：recovery 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）。
- **coredump**：coredump 的内容可以是 inline、disabled 或 default 三者之一，用于控制核心转储机制的行为。
- inline 表示 coredump 不会被复制到单独的缓冲区，恢复过程必须等到数据被用户空间读取；
- disabled 表示这是默认的核心转储机制。恢复将在不收集任何转储的情况下继续进行；

当 remoteproc 崩溃时，整个核心转储将被复制到一个单独的缓冲区并暴露给用户空间。

6.2 debugfs 节点

在 remoteproc 框架中向用户提供了 **name、recovery、crash、resource_table、carveout_memories、coredump、aw_trace_log、aw_trace_event** 节点，其路径如下：

```
/sys/kernel/debug/remoteproc/remoteproc0/name
/sys/kernel/debug/remoteproc/remoteproc0/recovery
/sys/kernel/debug/remoteproc/remoteproc0/crash
/sys/kernel/debug/remoteproc/remoteproc0/resource_table
/sys/kernel/debug/remoteproc/remoteproc0/carveout_memories
/sys/kernel/debug/remoteproc/remoteproc0/coredump
/sys/kernel/debug/remoteproc/remoteproc0/aw_trace_log
/sys/kernel/debug/remoteproc/remoteproc0/aw_trace_event
```

其作用分别为：

- **name**：显示远程处理器的名称；
- **recovery**：recovery 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）。
- **crash**：记录系统崩溃时候的有关信息；
- **resource_table**：记录了从核的资源信息（资源表）；
- **carveout_memories**：记录从核的内存分配情况；
- **coredump**：coredump 的内容可以是 inline、disabled 或 default 三者之一，用于控制核心转储机制的行为。
- inline 表示 coredump 不会被复制到单独的缓冲区，恢复过程必须等到数据被用户空间读取；

示这是默认的核心转储机制。恢复将在不收集任何转储的情况下继续进行；

- enabled 表示当 remoteproc 崩溃时，整个核心转储将被复制到一个单独的缓冲区并暴露给用户空间。
- **aw_trace_log**：查看从核系统运行过程中的日志。
- **aw_trace_event**：查看从核的 trace event

调试技巧 coredump

当从核运行过程中发生异常（内存越界、堆栈溢出、非法指针等）导致 crash 时，主核 Linux 系统会把从核使用到的内存区域的数据存储在一个文件中，这个过程称作 coredump，产生的文件即为 coredump 文件 (ELF 格式)，coredump 一般译为核心转储。

7.1 coredump 的使用步骤

1. 修改内核配置，将下面的配置设置为 y

```
CONFIG_COREDUMP=y
CONFIG_WANT_DEV_COREDUMP=y
CONFIG_ALLOW_DEV_COREDUMP=y
CONFIG_DEV_COREDUMP=y
```

2. 使能从核 coredump

方法一：永久启用，修改设备树中 remoteproc 对应的节点，增加 coredump 属性：

```
coredump;
```

以 MR536 平台为例，修改如下：

```
#tina/bsp$ git diff configs/linux-5.15-origin/sun55iw6p1.dtsi
  diff --git a/configs/linux-5.15-origin/sun55iw6p1.dtsi b/configs/linux-5.15-origin/sun55iw6p1.dtsi
  index 835331c9f..3ed5402fa 100644
  --- a/configs/linux-5.15-origin/sun55iw6p1.dtsi
  +++ b/configs/linux-5.15-origin/sun55iw6p1.dtsi
                  e907_rproc: e907_rproc@1a00000 {
  +                       coredump;
                          reg = <0x0 0x1a00000 0x0 0x1000>,
                                <0x0 0x01a02000 0x0 0x400>;
                          compatible="allwinner,e907-rproc";
```

方法二：临时启用（掉电丢失）

```
echo enabled > /sys/class/remoteproc/remoteproc0/coredump
```

3. 编译 linux 内核，打包镜像，烧录更新。
4. 在从核执行 panic 命令，产生 crash。

以 MR536 平台为例，在从核执行 panic 命令后的输出如下：

```
cpu0>panic
================================================================================
                        EXC_BREAKPOINT
================================================================================

gprs:
 x0:0x00000000   ra:0x7000e3c6   sp:0x7008a410   gp:0x70045130
 tp:0x00000000   t0:0xa5a5a5a5   t1:0x00000002   t2:0xa5a5a5a5
 s0:0x00000000   s1:0x00000001   a0:0x00000001   a1:0x7008a420
 a2:0x00000000   a3:0x00000000   a4:0x00000000   a5:0x7000de2c
 a6:0x00000000   a7:0x7008a424   s2:0x7008a420   s3:0x7002ed90
 s5:0x70050d50   s5:0x7008a6f0   s6:0xa5a5a5a5   s7:0xa5a5a5a5
 s8:0xa5a5a5a5   s9:0xa5a5a5a5  s10:0xa5a5a5a5  s11:0xa5a5a5a5
 t3:0x00000022   t4:0x0000003b   t5:0x00000020   t6:0x00000009

other:
mepc    :0x7000de2c
mcause  :0x38000003
mtval   :0x00000000
mstatus :0x00003880
```

a5a5a5

```
-------backtrace-----------
backtrace : 0X7000DE2C #某条指令的地址，从核因为执行这条指令而导致crash，通过这个地址，可以定位到此指令对应函数的哪一行
backtrace : invalid lr(00000000)
backtrace : 0X7000E3C4
backtrace : 0X7000D9E8
backtrace : 0X7000DD8C
backtrace : 0X70025DF4

省略
```

5. 在主核 Linux 系统里可以查看从核生成的 coredump 文件，coredump 文件路径为：`/sys/class/remoteproc/remoteproc0/devcoredump/data`

⚠ 注意

coredump，不会生成 coredump 文件，coredump 文件只有在从核 crash 时才生成。

- 读取完成后需要对/sys/class/remoteproc/remoteproc0/devcoredump/data 写入任意数据，否则新产生的 coredump 不会保存，除非旧的 coredump 的生命周期结束 (一般是 5 分钟后自动释放掉)

6. 将 coredump 文件从主核 Linux 系统拷贝出来

以 MR536 为例，使用 adb pull 拉取 coredump 文件：

```
C:\Users\xxx\Desktop\temp>adb pull /sys/class/remoteproc/remoteproc0/devcoredump/data coredump.data
/sys/class/remoteproc/remoteproc0/devcoredump/data: 1 file pulled. 3.8 MB/s (2117812 bytes in 0.534s)
```

7. 将 coredump 文件上传到到 Tina SDK 的服务器的 rtos 目录

```
~/workspace/{SDK_ROOT}/rtos$ ls -ahl coredump.data
-rwxr--r-- 1 user user 3.1M 4月   9 10:17 coredump.data
```

7.2 gdb 查看状态、全局变量

用户可以将从核的奔溃时生成的 coredump.data 拷贝到 SDK 中，然后执行命令将带有调试信息的 rtos elf 原始固件中的 section 数据替换为 coredump.data 中相应的 section 数据，重新生成一个带有奔溃时内存数据且带有调试信息的 elf 固件，执行命令后会自动进入 gdb 终端，用户可以在 gdb 终端中查看全局变量、奔溃时的状态信息、内存等，具体操作如下：

1. 把奔溃时，从核的 coredump 数据拷贝到 SDK 中的 rtos 的目录下：

```
~/workspace/{SDK_ROOT}/rtos$ ls
board  coredump.data  envsetup.sh  lichee  text  tools
~/workspace/{SDK_ROOT}/rtos$ ls -ahl coredump.data
-rwxr--r-- 1 user user 3.1M 4月   9 10:17 coredump.data
```

nch执行 env 设置以及进行

```
~/workspace/{SDK_ROOT}/rtos$ source envsetup.sh
~/workspace/{SDK_ROOT}/rtos$ lunch_rtos
```

3. 在 rtos 目录下执行命令，将带有调试信息的 rtos elf 原始固件中的 section 数据替换为 coredump.data 中相应的 section 数据，并重新生成一个带有奔溃时内存数据且带有调试信息的 elf 固件。执行命令成功后，会自动进入 gdb 终端。

```
~/workspace/{SDK_ROOT}/rtos$ rv_coredump_debug coredump.data
```

以 MR153 为例，执行时的关键日志如下：

```
ina_mr153/rtos$rv_coredump_debug coredump.data
  /home/user/workspace/tina_mr153/rtos
  ~/workspace/tina_mr153/rtos/tools/tool/rv_coredump_debug ~/workspace/tina_mr153/rtos

  warning: Couldn't find general-purpose registers in core file.

  warning: Couldn't find general-purpose registers in core file.
  #0  <unavailable> in ?? ()
  rv elf file path = /home/user/workspace/tina_mr153/rtos/lichee/rtos/build/mr153_e907_evb1/img/rt_system.elf
  rv mem file path = coredump.data.mem
  rv mem file addr = 421c5000
  rv out file path = /home/user/workspace/tina_mr153/rtos/lichee/rtos/build/mr153_e907_evb1/img/rt_system_fix.elf
  elf file size: 2181564
  mem file size: 3145728
  replace memory
  update .text
  update .data
  update .version_table
ce_table
  update .digest
  update .FSymTab
  update .amp_user_rsc later
  update .bss later
  add memory
  idx: 0
  idx: 1
  idx: 2
  idx: 3
  idx: 4
  idx: 5
  idx: 6
  idx: 7
  update .amp_user_rsc
  idx: 8
  update .bss
  idx: 9

  idx: b
  idx: c
  idx: d
  idx: e
  idx: f
  idx: 10
  idx: 11
  idx: 12
  idx: 13
  idx: 14
  idx: 15
  done
  add mem to elf success
  GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
  Copyright (C) 2020 Free Software Foundation, Inc.
  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
  This is free software: you are free to change and redistribute it.
RANTY, to the extent permitted by law.
  Type "show copying" and "show warranty" for details.
  This GDB was configured as "x86_64-linux-gnu".
  Type "show configuration" for configuration details.
  For bug reporting instructions, please see:
  <http://www.gnu.org/software/gdb/bugs/>.
  Find the GDB manual and other documentation resources online at:
      <http://www.gnu.org/software/gdb/documentation/>.

  For help, type "help".
  Type "apropos word" to search for commands related to "word"...
  Reading symbols from /home/user/workspace/tina_mr153/rtos/lichee/rtos/build/mr153_e907_evb1/img/rt_system_fix.elf...

  warning: Loadable section ".bss" outside of ELF segments
  (gdb)
```

1. 命令执行日志中的`#0 <unavailable> in ?? ()`以及前两行`warning`打印可以忽略
2. 命令执行分为两个阶段，第一个阶段是生成带有崩溃时运行内存以及调试信息的 elf 固件，这个阶段执行成功会打印`add mem to elf success`。第二阶段时，进入 gdb 终端，进入 gdb 终端时，提示的 `warning:Loadablesection".bss"outside ofELFsegments` 可以忽略，gdb 终端进入成功后，会打印提示`(gdb)`，等待用户输入命令。gdb 如果无法正常进入，用户需要根据 gdb 报错提示检查自己服务器的是否安装了 gdb 以及是否缺失 gdb 正常运行所需的依赖库。

进入 gdb 时，默认加载了全志自定义的 GDB 命令脚本文件，允许用户在 gdb 终端中使用`stat`命令查看调度器、堆、以及任务状态：

```
(gdb) stat
scheduler stat:
         uxTaskNumber: 17
         uxCurrentNumberOfTasks: 9
         uxTopReadyPriority: 40001

         xSchedulerRunning: 1
         xPendedTicks: 0
         xTickCount: 342569

heap stat:
         total:         2837976 (    2771 KB)
         free:          2707712 (    2644 KB)
         free(mini):    2632688 (    2570 KB)

status         tcb    id prio nest runtimec      stack-free-base-cur name
running 0x4222bc10     9   18    1        8   3542 42227c00 4222b358 CLI
ready   0x4221aed0     4    0    0   342219    978 42219ec0 4221ae08 IDLE
delay2  0x42219530    12    6    0        0   4006 422342e0 42238178 ctrldev
delay2  0x42218200     2   31    0       13    206 42212b80 42212eb8 amp_hw_wdog_feed
delay2  0x4221d080     5   31    0        1   1988 4221b070 4221cf80 Tmr Svc
delay2  0x42218060     1   15    0        2   4434 42213050 42217598 uart
suspend 0x4221f750     7   17    0        0    926 4221e740 4221f5b8 pm_client
suspend 0x4221e410     6   16    0        0   4004 4221e2a8 pm_suspend
suspend 0x42234080    11   31    0        0   8118 4222c070 42233f48 cpu-vring-ipi
end
```

用户可以打印全局变量：

```
(gdb) print rtos_version_msg
$1 = "UTS - Tue, 08 Apr 2025 09:29:21 +0800\nCompile Time - 09:29:21"

(gdb) print g_is_irq_core_init
$2 = 1

(gdb) print uart_msg
$3 = {{base = 39845888, irqn = 1800, clk_id = 117, rst_id = 27, uart_tx = 41, uart_rx = 42, func = 2, drqdst_tx = -1, drqsrc_rx =
      -1, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 39849984, irqn = 1900, clk_id = 118,
    rst_id = 28, uart_tx = 160, uart_rx = 161, func = 4, drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false, used_dma = false,
     rsc = 0x0}, {base = 39854080, irqn = 2000, clk_id = 119, rst_id = 29, uart_tx = 32, uart_rx = 33,
    drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 39858176, irqn = 2100,
     clk_id = 120, rst_id = 30, uart_tx = 110, uart_rx = 111, func = 6, drqdst_tx = -1, drqsrc_rx = -1,
    no_suspend = false, used_dma = false, rsc = 0x0}, {base = 39862272, irqn = 2200, clk_id = 121, rst_id = 31, uart_tx = 102,
     uart_rx = 103, func = 6, drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false, used_dma = false,
    rsc = 0x0}, {base = 39866368, irqn = 2300, clk_id = 122, rst_id = 32, uart_tx = 288, uart_rx = 289, func = 7, drqdst_tx = -1,
      drqsrc_rx = -1, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 39870464, irqn = 2400,
    clk_id = 123, rst_id = 33, uart_tx = 100, uart_rx = 101, func = 7, drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false,
     used_dma = false, rsc = 0x0}, {base = 39874560, irqn = 2500, clk_id = 124, rst_id = 34, uart_tx = 45,
    uart_rx = 46, func = 3, drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false, used_dma = false, rsc = 0xa0dead55}, {base =
     39878656, irqn = 16400, clk_id = 125, rst_id = 35, uart_tx = 106, uart_rx = 107, func = 7, drqdst_tx = -1,
    drqsrc_rx = -1, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 39882752, irqn = 16500, clk_id = 126, rst_id = 36,
     uart_tx = 96, uart_rx = 97, func = 7, drqdst_tx = -1, drqsrc_rx = -1, no_suspend = false,
    used_dma = false, rsc = 0x0}, {base = 0, irqn = 0, clk_id = 0, rst_id = 0, uart_tx = 0, uart_rx = 0, func = 0, drqdst_tx = 0,
      drqsrc_rx = 0, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 0, irqn = 0, clk_id = 0,
    rst_id = 0, uart_tx = 0, uart_rx = 0, func = 0, drqdst_tx = 0, drqsrc_rx = 0, no_suspend = false, used_dma = false, rsc = 0x0
```

```
=0, irqn=0, clk_id=0, rst_id=0, uart_tx=0, uart_rx=0, func=0, drqdst_tx=0,
      drqsrc_rx = 0, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 0, irqn = 0, clk_id = 0, rst_id = 0, uart_tx = 0,
       uart_rx = 0, func = 0, drqdst_tx = 0, drqsrc_rx = 0, no_suspend = false, used_dma = false, rsc = 0x0}, {
      base = 0, irqn = 0, clk_id = 0, rst_id = 0, uart_tx = 0, uart_rx = 0, func = 0, drqdst_tx = 0, drqsrc_rx = 0, no_suspend =
       false, used_dma = false, rsc = 0x0}, {base = 0, irqn = 0, clk_id = 0, rst_id = 0, uart_tx = 0,
      uart_rx = 0, func = 0, drqdst_tx = 0, drqsrc_rx = 0, no_suspend = false, used_dma = false, rsc = 0x0}, {base = 0, irqn = 0,
       clk_id = 0, rst_id = 0, uart_tx = 0, uart_rx = 0, func = 0, drqdst_tx = 0, drqsrc_rx = 0,
      no_suspend = false, used_dma = false, rsc = 0x0}}
```

用户可以查看内存：

```
(gdb) x/4x 0x421d1532  #从内存地址 0x421d1532 开始，查看 4 个内存单元，每个单元以十六进制格式显示
0x421d1532 <cmd_panic>: 0x45019002      0x72f98082      0x110102c1      0x6709ce06


解析：
0x421d1532：这是查看的起始内存地址。
这个地址对应的符号是cmd_panic，在这个地址处，程序正在执行cmd_panic函数。
37980820xce0611010x0814911a：这些是从地址0x421d1532开始的四个4字节（32位）值，以十六进制格式显示。
```

7.3 gdb 查看 trace event

若当用户配置打开 rtos 的 trace event 功能（具体请查阅“从核跟踪事件获取”章节），从核崩溃时，从设备中导出的 coredump data 包含有记录了 trace event 日志的全局数组，用户使用全志自定义的 gdb 命令 `event_dump` 可以把 trace event 日志的全局数组中的 event log 导出，具体如下：

```
(gdb) event_dump
$1 = {magic = 4092878443, stat = 4223233, pid = 4, name = 0x421f4fac "unlock:7:mutex:", time = 5142563967105, args = 0x42211b38 <
     g_events_buffer+24>}
[42211b20 +1c                          Tmr Svc-4 [0] .0 5142.563967105 tracing_mark_write: E|4|mutex|unlock:7:mutex:{0x42254690}]
[42211b3c +20                          Tmr Svc-4 [0] .0 5142.563972271 tracing_mark_write: B|4|mutex|lock:7:mutex:4:ticks:{0
     x42254690}{-1}]
[42211b5c +1c                          Tmr Svc-4 [0] .0 5142.563985146 tracing_mark_write: E|4|mutex|unlock:7:mutex:{0x42254690}]
rSvc-4[0].05143.554379427tracing_mark_write:B|4|mutex|lock:7:mutex:4:ticks:{0
     x42254690}{-1}]
[42211b98 +1c                          Tmr Svc-4 [0] .0 5143.554386468 tracing_mark_write: E|4|mutex|unlock:7:mutex:{0x42254690}]
[42211bb4 +20                          Tmr Svc-4 [0] .0 5143.554391385 tracing_mark_write: B|4|mutex|lock:7:mutex:4:ticks:{0
     x42254690}{-1}]
[42211bd4 +1c                          Tmr Svc-4 [0] .0 5143.554403677 tracing_mark_write: E|4|mutex|unlock:7:mutex:{0x42254690}]
[42211bf0 +20                          Tmr Svc-4 [0] .0 5144.544798207 tracing_mark_write: B|4|mutex|lock:7:mutex:4:ticks:{0
     x42254690}{-1}]
....略...
```

7.4 从核各种状态下的 coredump 数据处理

从核在 uevent-monitor-for-rproc_wdt 程序运行前就已经发生崩溃，会导致 uevent-monitor-for-rproc_wdt 程序无法拿到运行前产生的看门狗超时事件，丢失看门狗事件，用户就无法保存 coreudmp 文件，影响问题调试。为了避免这种情况，用户在导出从核 coredump 数据前，应主动检查从核状态，以获取正确的 coredump 数据，具体如下：

```
lass/remoteproc/remoteprocX/state文件，根据状态进行处理：
  ├─ attached --> 这种状态是快启平台的从核正常运行状态，无需处理读取devcoredump/data
  ├─ running --> 说明从核已发生过一次 recovery，用户应检查 devcoredump/data 是否存在
  |    ├─ 存在 --> 此为recovery过程中自动生成的有效coredump数据，应保存
  |    └─ 不存在 --> 说明此时coredump数据已被内核5分钟超时清理了
  ├─ crashed --> 这种状态表示recovery失败，更具体来说是主核stop从核失败。recovery过程中如果stop从核成功，在没有重新start从核前，从核的状态
       都应该为offline。用户应检查 devcoredump/data 是否存在
  |    ├─ 存在 --> 这部分数据不可用，因为从核状态是在切换成offline之后，才会进行coredump，所以此时有数据则是上一次从核崩溃的残留数据，用户应
       主动去清除残留的数据，清理方式：向 devcoredump/data 写入任意内容（如 "1"），触发内核调度删除，然后再手动 trigger 产生新的 coredump
  |    └─ 不存在 --> 手动 trigger coredump --> 等待 data --> 保存
  └─ offline --> stop从核成功但start从核失败，用户应检查 devcoredump/data 是否存在
     ├─ 存在 --> 此为有效的coredump数据，应保存
     └─ 不存在 --> 手动 trigger coredump --> 等待 data --> 保存
```

手动 trigger coredump 的方式：通过写 “trigger” 到/sys/class/remoteproc/remoteprocX/device/rproc_dump 文件，触发内核调用 `rproc->ops->coredump()`，生成新的 devcoredump 数据。
具体源码可参考全志提供的 uevent-monitor-for-rproc_wdt 程序（路径：`openwrt/package/allwinner/testtools/testapk/openamp_robustness_test/src/uevent-monitor`）。如旧的 SDK 包中不存在如上目录，请找对应 fae 获取。

核 AMP 控制台

当前 SDK 中从核的控制台分为 2 个，一个是通过串口进入的串口控制台，另外一个是在 Linux 端通过 amp_shell 命令进入的 AMP 控制台，AMP 控制台主要提供在 Linux 端远程执行从核固件里的命令并查看命令的输出结果，一般在实际设备未引出从核串口的情况下使用。而串口控制台除了执行命令和查看命令输出结果外还可查看从核系统运行过程中打印的一些日志。AMP 控制台配置步骤如下。

8.1 配置信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPMSG_CTRL、软件包配置 CONFIG_PACKAGE_amp_shell。

从核 RTOS 环境需要启用如下配置：

- CONFIG_RPMSG_CLIENT
- CONFIG_MULTI_CONSOLE

RT_MULTI_CONSOLE

- CONFIG_UART_MULTI_CONSOLE_AS_MAIN
- CONFIG_RPMSG_MULTI_CONSOLE
- CONFIG_RPMSG_CONSOLE_CACHE

8.2 使用示例

以 MR527 为例，在 Linux 终端运行 amp_shell -d /dev/rpmsg_ctrl-e906_rproc@7130000 可以进入 RV 核 AMP 控制台。

图 8-1: amp 控制台现象

```
root@TinaLinux:/# amp_shell -d dev/rpmsg_ctrl-e906_rproc@7130000
opt = ?
[  895.739030] virtio_rpmsg_bus virtio0: creating channel sunxi,rpmsg_client addr 0x402
rpmsg opt
opt = d
Creating Endpoint...
Success Create /dev/rpmsg1
============================
          AMP Shell
============================
msh >
msh >ls
msh >Command not recognised.  Enter 'help' to view a list of available commands.


msh >ts
msh >Task          State  Priority  Stack       #
*********************************************
Name            State   Pri     HWM     Idx     StkCur          StkBot
rpmsg1-console                  X       15      4322    15      0x40023410      0x4001ef20
IDLE                            R       0       891     2       0x40003f00      0x40003040
Tmr Svc                         B       31      1907    3       0x40005f90      0x400040f0
ctrldev                         B       6       3880    8       0x4001ed20      0x4001af10
vring-ipi                       B       16      8084    7       0x4001a410      0x400125c0
CLI                             B       18      3696    5       0x40011c90      0x4000e2d0

msh >
```

amp_shell 命令参数说明：

- -d devname：指定要与哪个远端处理器建立连接，可使用 ls /dev/rpmsg_ctrl* 查看可用的设备文件。

技巧

可使用命令 amp_exit 退出 AMP 控制台

核日志获取

由于 AMP 控制台只能用来执行命令并查看命令输出结果，因此在未引出从核串口的情况下若需要获取从核系统运行过程中打印的日志则需要使用 AMP trace log 功能。

AMP trace log 的实现机制说明：

- 异构系统会从 DDR SDRAM 中开辟（carve out）出一段空间用作从核的运行内存，代码段、数据段、BSS 段、堆栈都在这段内存当中；
- 从核在 BSS 段中定义一个全局的数组 amp_log_buffer，将这个 trace buffer 的 address 和 len，通过 resource table 传递给主核
- 从核的 printf 打印日志接口会同步将日志写入 amp_log_buffer
- 主核解析 resource table 获得 trace buffer 的 address 和 len，然后从相应的内存中，读取 trace log

9.1 配置信息

需要启用内核配置 CONFIG_AW_RPROC_ENHANCED_TRACE

从核 RTOS 环境需要启用配置 CONFIG_AMP_TRACE_SUPPORT。

技巧

RTOS 环境下可通过配置 CONFIG_AMP_TRACE_BUF_SIZE 来修改 trace buffer 的大小，此 buffer 对应的内存来自 RTOS 环境的一个全局数组，定义在 rtos/lichee/rtos-components/thirdparty/openamp/trace_log/openamp_log.c 文件里，因此配置 CONFIG_AMP_TRACE_BUF_SIZE 的值决定了此全局数组占用的内存大小。

9.2 使用示例

在 Linux 系统控制台下执行命令 cat /sys/kernel/debug/remoteproc/remoteproc0/aw_trace_log 即可看到从核系统运行过程中的日志：

```
x:/# cat /sys/kernel/debug/remoteproc/remoteproc0/aw_trace_log
it


 info: writer init
 [0.108]sunxi_dma_clk_init()412 enter dma clk init
 [0.126]
 [0.127] ****************************************
 [0.128] ** Welcome to MR536_E907 FreeRTOS V1.6.0    **
 [0.132] ** Copyright (C) 2019-2021 AllwinnerTech **
 [0.136] **                                        **
 [0.141] **      starting riscv FreeRTOS           **
 [0.146] ****************************************
 [0.151]
 [0.151]Date:May 31 2024, Time:10:10:43
 [0.155]init Icache
 [0.157]init Dcache
 [0.158]AMP timestamp device 0 probe success, count_freq: 24000000Hz
 [0.165]amp_hw_wdog_feed_thread enter, feed_interval: 100ms, feed_interval_tick: 100
dware watchdog's new timeout: 2s
 [0.176]Begin to feed AMP hardware watchdog!
 [0.180]pm_client_init end!!!
 [0.190][RV] [AMP_INFO][openamp_platform_init:112]rproc0 init
 [0.197][RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0x60039850, da: 0xffffffff, size = 0xc8
 [0.206][RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x60039850) to va(0x60039850)
 [0.214][RV] [AMP_INFO][openamp_sunxi_create_rproc:238]Wait master update resource_table
 [4.124][RV] [AMP_INFO][openamp_sunxi_create_rproc:241]resource_table ready
 [4.126][RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0x42400000, da: 0x42400000, size = 0x40000
 [4.136][RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42400000) to va(0x42400000)
 [4.144][RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0xffffffff, da: 0x42440000, size = 0x1406
 [4.154][RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42440000) to va(0x42440000)
 [4.162][RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0xffffffff, da: 0x42442000, size = 0x1406
 [4.172][RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42442000) to va(0x42442000)
 [4.180][RV] [AMP_INFO][openamp_sunxi_create_rpmsg_vdev:365]Wait connected to remote master
 [4.189][RV] [AMP_INFO][openamp_sunxi_create_rpmsg_vdev:371]Connecte to remote master Successed
```

```
MP_INFO][openamp_platform_init:157]rproc0 init done
  [4.205][RV] [AMP_INFO][openamp_platform_init:112]rproc0 init
  [4.211][RV] [AMP_INFO][openamp_platform_init:122]rproc0 already init.
  [4.218][RV] [AMP_INFO][openamp_ept_open:321]Waiting for ept('sunxi,rpmsg_ctrl') ready
  [4.237][RV] [AMP_INFO][openamp_ept_open:326]ept('sunxi,rpmsg_ctrl') ready! src: 0x400, dst: 0x400
  [4.241]rpmsg ctrldev: Start Running...
  [4.245][RPBUF_INFO][rpbuf_init_service:154]Waiting for rpbuf ept('rpbuf-service') ready.
  [4.253][RV] [AMP_INFO][openamp_platform_init:112]rproc0 init
  [4.259][RV] [AMP_INFO][openamp_platform_init:122]rproc0 already init.
  [4.266][RV] [AMP_INFO][openamp_ept_open:321]Waiting for ept('sunxi,rpmsg_heartbeat') ready
  [4.297][RPBUF_INFO][rpbuf_init_service:160]rpbuf ept('rpbuf-service') ready! src: 0x401, dst: 0x401
  [4.315][RV] [AMP_INFO][openamp_ept_open:326]ept('sunxi,rpmsg_heartbeat') ready! src: 0x402, dst: 0x402
  [4.319]Start Rpmsg Hearbeat Timer
```

从核进入异常状态后，主核 Linux 端自动重启从核前也会主动获取从核系统运行过程中的日志并打印出来。如下所示，从核 panic 异常时，主核 Linux 端既输出了内核日志，也输出了从核的日志。用户可配置主核 Linux 端的 CONFIG_AW_RPROC_ENHANCED_TRACE_CRASH_DUMP 为 n，可禁止在从核异常时输出从核日志到内核日志中。

```
  [   36.053987] axp2202-cldo4: disabling
  [   36.059619] axp2202-vmid: disabling
  [   39.377556] remoteproc remoteproc0: crash detected in e907_rproc: type watchdog
  [   39.386901] remoteproc remoteproc0: handling crash #1 in e907_rproc
  [   39.394063] remoteproc remoteproc0: recovering e907_rproc
  [   39.404001] rpmsg_heartbeat virtio0.sunxi,rpmsg_heartbeat.-1.1026: virtio0.sunxi,rpmsg_heartbeat.-1.1026 is removed
  [   39.417880] sunxi-rproc 1a00000.e907_rproc: exception cause: EXC_BREAKPOINT
  [   39.425826] sunxi-rproc 1a00000.e907_rproc: begin dump crash log before stop remoteproc!
  [   39.435806] sunxi-rproc 1a00000.e907_rproc: trace mem: 0x60043780, len: 32768
  [   39.444356] trace_mem:     ffffffc009843780
  [   39.449378] trace_mem_len: 32768
  [   39.452993] r->pos:        0
  [   39.456240] r->area_size:  0
  [   39.459482] w->pos:        21346
  [   39.463116] w->size:       32640
  [   39.466743] w->area_size:  64
  [   39.470079] w->overrun:    0
fo:reader init
  [   39.476743]
  [   39.478524] info: writer init
  [   39.481937] sunxi_dma_clk_init()406 enter dma clk init
  [   39.487676]
  [   39.489440]  ****************************************
  [   39.495476]  ** Welcome to MR536_E907 FreeRTOS V1.6.0    **
  [   39.501810]  ** Copyright (C) 2019-2021 AllwinnerTech **
  [   39.507859]  **                                        **
  [   39.513892]  **      starting riscv FreeRTOS           **
  [   39.519915]  ****************************************
  [   39.525938]
  [   39.527690] Date:Feb 24 2025, Time:16:41:03
  [   39.532352] AMP timestamp device 0 probe success, count_freq: 24000000Hz
  [   39.539831] amp_hw_wdog_feed_thread enter, feed_interval: 100ms, feed_interval_tick: 100
  [   39.548854] AMP hardware watchdog's new timeout: 2s
  [   39.554295] Begin to feed AMP hardware watchdog!
  [   39.559446] pm_client_init end!!!
V][AMP_INFO][openamp_platform_init:217]rproc0 init
  [   39.570813] [RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0x6003f448, da: 0xffffffff, size = 0x138
  [   39.582840] [RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x6003f448) to va(0x6003f448)
  [   39.592644] [RV] [AMP_INFO][openamp_sunxi_create_rproc:238]Wait master update resource_table
  [   39.603023] [RV] [AMP_INFO][openamp_sunxi_create_rproc:241]resource_table ready
  [   39.612146] [RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0x42400000, da: 0x42400000, size = 0x40000
  [   39.624368] [RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42400000) to va(0x42400000)
  [   39.634171] [RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0x4244c000, da: 0x4244c000, size = 0x1000
  [   39.646300] [RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x4244c000) to va(0x4244c000)
  [   39.656102] [RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0xffffffff, da: 0x42440000, size = 0x1406
  [   39.668225] [RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42440000) to va(0x42440000)
  [   39.678030] [RV] [AMP_INFO][rproc_common_mmap:128]remoteproc mmap pa: 0xffffffff, da: 0x42442000, size = 0x1406
  [   39.690156] [RV] [AMP_INFO][rproc_common_mmap:199]map pa(0x42442000) to va(0x42442000)
  [   39.699957] [RV] [AMP_INFO][openamp_sunxi_create_rpmsg_vdev:365]Wait connected to remote master
  [   39.710626] [RV] [AMP_INFO][openamp_sunxi_create_rpmsg_vdev:371]Connecte to remote master Successed
```

```
penamp_share_irq_init:139]share_irq_table(addr:4244C000)invalidtag:0x0,shouldbe0x52454459
[   39.733037] [RV] [AMP_ERR][amp_share_irq_init:167]Init Share Interrupt Table Failed,ret=-14.
[   39.743323] [RV] [AMP_ERR][openamp_platform_init:256]Init AMP share interrupt failed, ret: -4
[   39.753707] [RV] [AMP_INFO][openamp_platform_init:268]rproc0 init done
[   39.761858] [RV] [AMP_INFO][openamp_platform_init:217]rproc0 init
[   39.769532] [RV] [AMP_INFO][openamp_platform_init:227]rproc0 already init.
[   39.778169] [RV] [AMP_INFO][openamp_ept_open:439]Waiting for ept('sunxi,rpmsg_ctrl') ready
[   39.788363] [RV] [AMP_INFO][openamp_ept_open:444]ept('sunxi,rpmsg_ctrl') ready! src: 0x400, dst: 0x400
[   39.799717] rpmsg ctrldev: Start Running...
[   39.804382] [RPBUF_INFO][rpbuf_init_service:154]Waiting for rpbuf ept('rpbuf-service') ready.
[   39.814476] [RV] [AMP_INFO][openamp_platform_init:217]rproc0 init
[   39.822532] [RV] [AMP_INFO][openamp_platform_init:227]rproc0 already init.
[   39.831167] [RV] [AMP_INFO][openamp_ept_open:439]Waiting for ept('sunxi,rpmsg_heartbeat') ready
[   39.841842] [RPBUF_INFO][rpbuf_init_service:160]rpbuf ept('rpbuf-service') ready! src: 0x401, dst: 0x401
[   39.852995] [RV] [AMP_INFO][openamp_ept_open:444]ept('sunxi,rpmsg_heartbeat') ready! src: 0x402, dst: 0x402
[   39.865214] Start Rpmsg Hearbeat Timer
[   39.869493] Warning: no shared GPIO interrupt info table!
[   39.875512] ================================================================================================
EXC_BREAKPOINT
[   39.895309] ================================================================================================
[   39.906759]
[   39.908413] gprs:
[   39.910560]  x0:0x00000000   ra:0x6000caf4   sp:0x6007aef0   gp:0x6003f800
[   39.918230]  tp:0x00000000   t0:0xa5a5a5a5   t1:0x00000002   t2:0xa5a5a5a5
[   39.925902]  s0:0x00000000   s1:0x00000001   a0:0x00000001   a1:0x6007af00
[   39.933570]  a2:0x00000000   a3:0x00000000   a4:0x00000000   a5:0x6000c55a
[   39.941237]  a6:0x00000000   a7:0x6007af04   s2:0x6007af00   s3:0x6002ab10
[   39.948902]  s5:0x60042dd8   s5:0x6007b1d0   s6:0xa5a5a5a5   s7:0xa5a5a5a5
[   39.956571]  s8:0xa5a5a5a5   s9:0xa5a5a5a5  s10:0xa5a5a5a5  s11:0xa5a5a5a5
[   39.964239]  t3:0x00000022   t4:0x0000003b   t5:0x00000020   t6:0x00000009
[   39.971906]
[   39.973564] other:
[   39.975800] mepc    :0x6000c55a
[   39.979299] mcause  :0x38000003
[   39.982801] mtval   :0x00000000
[   39.986302] mstatus :0x00003880
cratch:0xa5a5a5a5
[   39.993303]
[   39.995058] -------backtrace-----------
[   39.999332] backtrace : 0X6000C55A
[   40.003318] backtrace : invalid lr(00000000)
[   40.008174] backtrace : 0X6000CAF2
[   40.012161] backtrace : 0X6000C116
[   40.016146] backtrace : 0X6000C4BA
[   40.020138] backtrace : 0X60022790
省略
```

当 CONFIG_AW_RPROC_ENHANCED_TRACE_CRASH_DUMP 配置为 n 时，从核异常时的详细日志不会输出到内核日志中，从核异常原因可通过类似如下的内核日志来判断，下面这个日志是从核 panic 时，主核 Linux 端打印的异常原因：

```
[   39.417880] sunxi-rproc 1a00000.e907_rproc: exception cause: EXC_BREAKPOINT
```

从核跟踪事件获取

10.1 概述

trace event（跟踪事件）作用如下：

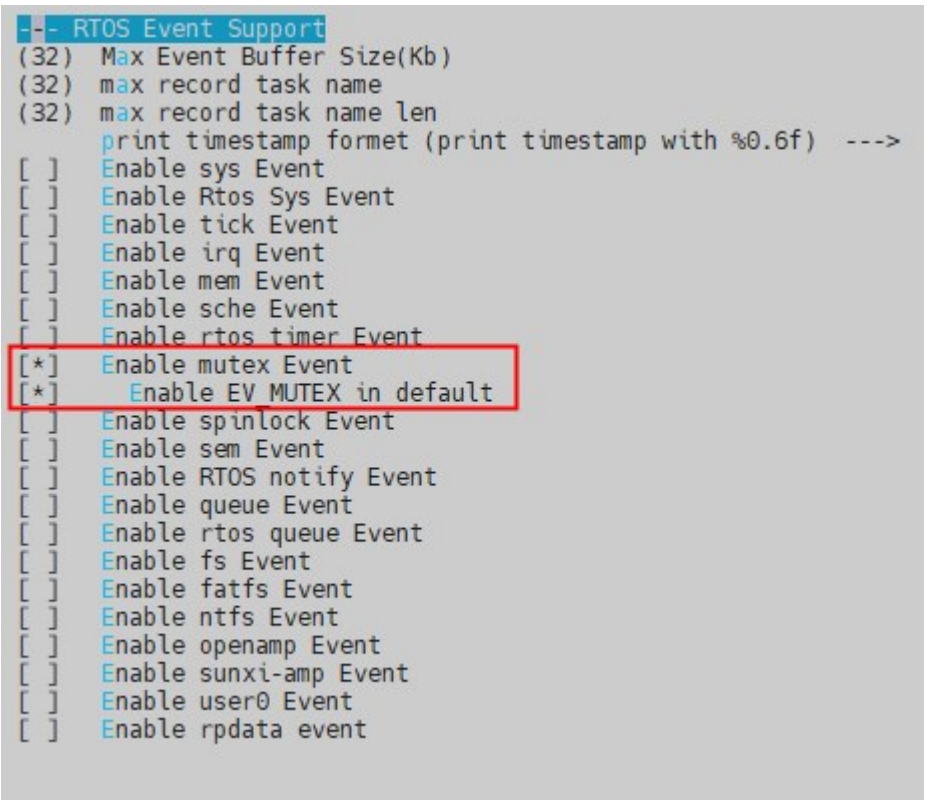
1. 用于性能监控和调试的一个重要特性，通过记录子系统的事件，可以帮助分析系统的性能瓶颈。例如，可以跟踪 CPU 调度器的行为、内存分配情况、I/O 操作等，trace_events 可以帮助开发者快速定位问题。例如，如果发现某个模块有问题，可以通过启用相关的跟踪事件来查看具体的执行过程。
2. 它允许用户**启用**和**自定义**各种跟踪事件
3. 定义跟踪点：跟踪点（Tracepoints）是在代码中预定义的位置，当执行到这些位置时，相关的信息会被记录下来。
4. 用户可以通过配置文件或命令行工具来启用或禁用特定的跟踪事件。

从核支持用户配置各个子系统是否启用 trace event，相关的的配置层级如下：

```
System components
  ---> aw components
    ---> RTOS Event Support
```

以使能 mutex 的 trace event 为例，配置结果如下：

图 10-1: mutex trace_events 配置



用户如果需要启用其他模块的 trace event，启用相应的配置即可。

如下图所示，是 mutex 的 trace_event 日志输出到 event traces buffer 的部分情况。

图 10-2: hal_mutex 调用 trace_events 接口情况

```
lichee/rtos/drivers/osal/src/hal_mutex.c:3:#include <trace_event.h>
lichee/rtos/drivers/osal/src/hal_mutex.c:7:        trace_event_count(EV_MUTEX, "", mutex, ARG_INT_RENAME(mutex, 1));
lichee/rtos/drivers/osal/src/hal_mutex.c:41:       trace_event_count(EV_MUTEX, "", mutex, ARG_INT_RENAME(mutex, 0));
lichee/rtos/drivers/osal/src/hal_mutex.c:59:       trace_event_end(EV_MUTEX, "unlock", ARG_PTR(mutex));
lichee/rtos/drivers/osal/src/hal_mutex.c:87:       trace_event_begin(EV_MUTEX, "lock", ARG_PTR(mutex), ARG_INT(ticks));
```

10.3 从核 trace event 命令使用说明

```
cpu0>trace_events
useage:
  #导出所有支持的trace event的状态
  trace_events events        :dump all support subsys
  #把记录了trace evnent的buffer内容全部打印出来
  :dump events buffer
  #单独控制某个子系统的trace event的开关
  trace_events ctrl name 0/1 :ctrl wether enable sys event
  #禁用所有子系统trace events
  trace_events disable       :disable all sys event
  #使能所有子系统trace events
  trace_events enable        :enable all sys event
  #把记录了trace evnent的buffer内容全部清除掉
  trace_events clear         :clear event buffer
```

以 mutex 为例，演示 trace_events 令的使用：

```
cpu0>trace_events events                              #dump支持的trace events的子系统的trace event状态
        mutex                           [enable]      #支持mutex产生trace event日志，其状态为enabled
cpu0>trace_events ctrl mutex 0                        #disable mutex产生trace event
cpu0>trace_events events
        mutex                           [disable]     #支持mutex产生trace eventr日志，其状态为disabled
cpu0>trace_events ctrl mutex 1                        #disable mutex产生trace event
cpu0>trace_events events
        mutex                           [enable]
cpu0>trace_events disable                             #禁用所有子系统trace events
cpu0>trace_events events
        mutex                           [disable]
cpu0>trace_events enable                              #使能所有子系统trace events
cpu0>trace_events events
        mutex                           [enable]
```

10.4 日志格式说明

下面的伪代码中，记录了两条 mutex 的 trace event 日志

```
int a = 1;
//产生一条begin日志
trace_event_begin(EV_MUTEX, "foo1", ARG_PTR(&a));

trace_event_end(EV_MUTEX, "foo2", ARG_PTR(&a));
```

使用 trace event dump 可以得到如下的日志：

```
cpu0>trace_events dump

# tracer: nop
#
#                      _---=> irqs-off (.: irq enable, d:irq disabled)
#                     /  _--=> preempt-depth
#                     | /
#       TASK-PID   CPU# || TIMESTAMP
#          | |       |  ||     |
       hellowold-6  [0] .0 52.915813: tracing_mark_write: B|6|mutex: foo1 &a=6008DA18
```

```
old-6[0].052.915928:tracing_mark_write:E|6|mutex:foo2&a=6008DA18
```

日志分析如下，其中括号内的注解为日志的每个字段注解：

```
hellowold（任务名称）-6（任务pid） [0(cpu的id)] .0 52.915813(时间戳): tracing_mark_write（EV_MUTEX属于非EV_SCHE类event）: B(表示调用
    trace_event_begin输出的打印)|6（任务pid）|mutex（subsys_str，定义在trace_event_def.h的字符串）: foo1（trace_event_begin接口的第
    二个参数） &a（trace_event_begin第三个参数ARG_PTR的括号里面的字符）=6008DA18（&a的指针值）
hellowold（任务名称）-6（任务pid） [0(cpu的id)] .0 52.915928(时间戳): tracing_mark_write（EV_MUTEX属于非EV_SCHE类event）: E(表示调用
    trace_event_end输出的打印)|6（任务pid）|mutex（subsys_str，定义在trace_event_def.h的字符串）: foo2（trace_event_end接口的第二个
    参数） &a（trace_event_begin第三个参数ARG_PTR的括号里面的字符）=6008DA18（&a的指针值）
```

10.5 trace event 接口说明

10.5.1 trace_event

一个完整的事件，通常用于简单事件的跟踪。

语法：

```
trace_event(subsys, name, [arg1, arg2, ...])
```

- subsys：事件所属的子系统标识符，如 EV_SPIN、EV_SEM 等。
- name：事件名称，描述事件的类型或操作。
- arg：与事件相关的参数，可选，使用宏（如 ARG_PTR、RET_ARG）包装。

使用示例：

```
trace_event(EV_MUTEX, "lock", ARG_PTR(mutex));
```

件，M传递指针中的utex作为参数。

10.5.2 trace_event_begin

功能：用于记录事件的开始，通常与 trace_event_end 配对使用，用于跟踪事件的持续时间。

语法：

```
trace_event_begin(subsys, name, [arg1, arg2, ...])
```

- subsys：事件所属的子系统标识符。
- name：事件名称。
- arg：与事件相关的参数，可选。

使用示例：

```
trace_event_begin(EV_SPIN, "", ARG_PTR(lock), ARG_ULONG_RENAME(flags, __cpsr));
```

- 记录子系统 EV_SPIN 中的事件开始，传递指针 lock 和重命名为 __cpsr 的 flags 作为参数。

10.5.3 trace_event_end

功能：用于记录事件的结束，通常与 trace_event_begin 配对使用。

语法：

```
trace_event_end(subsys, name, [arg1, arg2, ...])
```

件所属的子系统标识符。

- name：事件名称。
- arg：与事件相关的参数，可选。

使用示例：

```
trace_event_end(EV_SPIN, "", ARG_PTR(lock));
```

- 记录子系统 EV_SPIN 中的事件结束，传递指针 lock 作为参数。

10.5.4 trace_event_count

功能：一个用于记录计数器事件的接口。它通常用于跟踪资源的使用情况，例如信号量、互斥锁等。

语法：

```
trace_event_count(subsys, _name, p, v)
```

- subsys：事件所属的子系统标识符，例如 EV_SEM、EV_MUTEX 等。
- _name：事件名称，通常留空。
- p：指针。
- v：计数器值，可以是当前计数或变化量。

使用示例：

```
trace_event_count(EV_SEM, "", sem, ARG_INT_RENAME(sem, cnt));
trace_event_count(EV_MUTEX, "", mutex, ARG_INT_RENAME(mutex, 1));
```

- 第一个示例记录信号量 sem 的计数变化。

录互斥锁的状态变化。(

10.6 自定义 trace event 与使用 trace event

1. 在 trace_event_def.h 中添加新的子系统枚举

- 在 event_subsys 枚举中，添加新的子系统枚举项，例如：

```
EV_NEW,
```

- 确保在定义时，新的子系统枚举项具有一个对应的条件编译宏（如 CONFIG_EVENTS_TRACE_EV_NEW）。

2. 添加新的子系统字符串定义

- 在 trace_event_def.h 文件中，添加新的子系统字符串宏定义，例如：

```
#define EV_NEW_STRING                "new"
```

3. 在 Kconfig 文件中添加配置选项

- 在 Kconfig 文件中，添加新的子系统配置选项，格式如下：

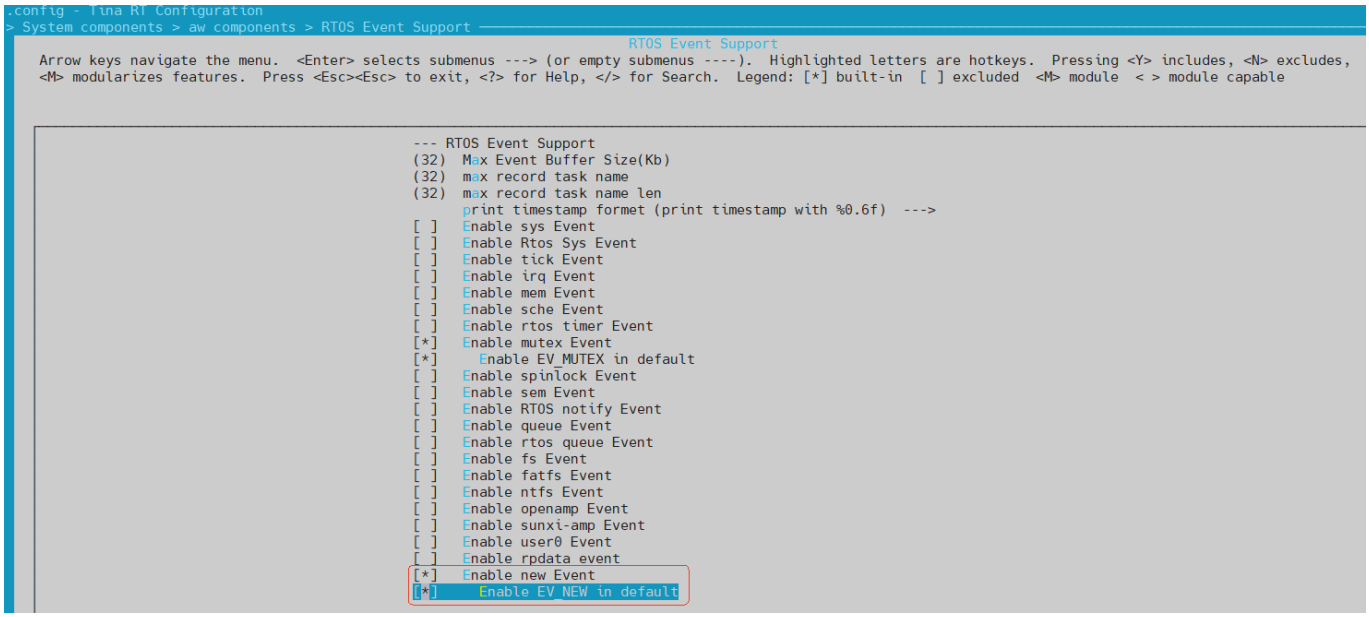
```
config EVENTS_TRACE_EV_NEW
       bool "Enable new Event"
       default n
config EVENTS_TRACE_EV_NEW_DEFAULT
       bool "Enable EV_NEW in default"
       depends on EVENTS_TRACE_EV_NEW
       default n
```

数组中增加新表ubsys_class

```
[EV_NEW] = {
   .name = EV_NEW_STRING,
   .enable = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_NEW_DEFAULT),
   .buildin = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_NEW)
},
```

5. 然后打开 new event 配置

图 10-3: 配置 new event



6. 使用 new event :

```
static void helloworld(void * param)
{
    int a = 1;
    int *p = &a;
    trace_event(EV_NEW, "init", ARG_PTR(p));
    trace_event_begin(EV_NEW, "foo1", ARG_PTR(p));
    trace_event_end(EV_NEW, "foo2", ARG_PTR(p));
    trace_event_count(EV_NEW, "", p, ARG_INT_RENAME(p, 1));
    trace_event_count(EV_NEW, "", p, ARG_INT_RENAME(p, 0));
    trace_event(EV_NEW, "deinit", ARG_PTR(p));
}
```

ent 日志:

```
cpu0>trace_events dump
# tracer: nop
#
#                       _---=> irqs-off (.: irq enable, d:irq disabled)
#                      /  _--=> preempt-depth
#                      | /
#       TASK-PID   CPU# || TIMESTAMP
#          | |       |  ||     |
       hellowold-0  [0] .0 45.020855: tracing_mark_write: I|0|new: init p=60074e0c
       hellowold-0  [0] .0 45.020866: tracing_mark_write: B|0|new: foo1 p=60074e0c
       hellowold-0  [0] .0 45.020866: tracing_mark_write: E|0|new: foo2 p=60074e0c
       hellowold-0  [0] .0 45.020870: tracing_mark_write: C|1611091468|p|1
       hellowold-0  [0] .0 45.020870: tracing_mark_write: C|1611091468|p|0
       hellowold-0  [0] .0 45.020870: tracing_mark_write: I|0|new: deinit p=60074e0c
```

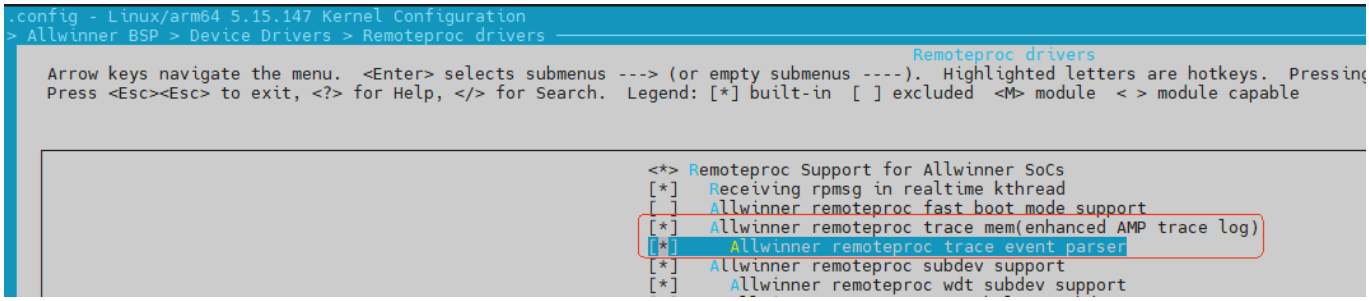
的补丁：

```
user@AW:~/tina_sdk/rtos/lichee/rtos-components$ git diff
diff --git a/aw/trace_event/event.Kconfig b/aw/trace_event/event.Kconfig
index f491c9e6..9864c9a3 100644
--- a/aw/trace_event/event.Kconfig
+++ b/aw/trace_event/event.Kconfig
@@ -158,3 +158,11 @@ config EVENTS_TRACE_EV_RPD_DEFAULT
        depends on EVENTS_TRACE_EV_RPD
        default n

+config EVENTS_TRACE_EV_NEW
+       bool "Enable new Event"
+       default n
+config EVENTS_TRACE_EV_NEW_DEFAULT
+       bool "Enable EV_NEW in default"
+       depends on EVENTS_TRACE_EV_NEW
+       default n

diff --git a/aw/trace_event/include/trace_event_def.h b/aw/trace_event/include/trace_event_def.h
index d7b3424a..82c7dae3 100644
--- a/aw/trace_event/include/trace_event_def.h
+++ b/aw/trace_event/include/trace_event_def.h
@@ -23,7 +23,8 @@ typedef enum {
        EV_SUNXIAMP,
        EV_USR0,
        EV_RPD,
-       EV_NUM_SUBSYS
+       EV_NEW,
+       EV_NUM_SUBSYS,
 } event_subsys;

 #define EV_SYS_STRING                  "sys"
@@ -46,5 +47,6 @@ typedef enum {
 #define EV_SUNXIAMP_STRING             "amp"
 #define EV_USR0_STRING                 "usr0"
#define EV_RPD_STRING
+#define EV_NEW_STRING                  "new"

 #endif
diff --git a/aw/trace_event/trace_event_def.c b/aw/trace_event/trace_event_def.c
index 818aaa8a..3c49fedf 100644
--- a/aw/trace_event/trace_event_def.c
+++ b/aw/trace_event/trace_event_def.c
@@ -101,6 +101,12 @@ struct subsys_entry subsys_class[EV_NUM_SUBSYS] = {
        [EV_RPD] = {
                .name = EV_RPD_STRING,
                .enable = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_RPD_DEFAULT),
-               .buildin = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_RPD)
+               .buildin = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_RPD),
        },
+
+       [EV_NEW] = {
+               .name = EV_NEW_STRING,
.enable=EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_NEW_DEFAULT),
+               .buildin = EV_MACRO_IS_ENABLED(CONFIG_EVENTS_TRACE_EV_NEW)
        },
 };
```

10.7 主核获取从核的 trace events

从核在支持了 trace event 的基础上，主核配置 CONFIG_AW_RPROC_TRACE_EVENT_PARSER 为 y，允许用户在主核 Linux 命令行获取 trace event。配置如下图所示

图 10-4: 配置支持主核通过命令行获取 trace event



主核在命令行获取 trace event 日志如下图所示，用户可以执行 `cat/sys/kernel/debug/remoteproc/remoteproc0/aw_trace_event` 获取从核日志

图 10-5: 主核获取从核的 trace event 日志

```
root@TinaLinux:/sys/kernel/debug/remoteproc/remoteproc0# cat aw_trace_event
abnormal read loop cnt! reader: pos=99, loop=0, writer: pos=4818, loop=0
# tracer: nop
#
#                              _---=> irqs-off (.: irq enable, d:irq disabled)
#                             /  _--=> preempt-depth
#                             | /
#       TASK-PID    CPU#  || TIMESTAMP
#         | |         |   ||    |
       amp_init-0   [0] .0 127921462: tracing_mark_write: C|1611130208|mutex|-202088853
       amp_init-0   [0] .0 1053919115: tracing_mark_write: B|0|mutex: lock mutex=0000000000000000 ticks=-202088853
       amp_init-0   [0] .0 1056758323: tracing_mark_write: E|0|mutex: unlock mutex=0000000000000000
       amp_init-0   [0] .0 1057429532: tracing_mark_write: C|1611035136|mutex|-202088853
       amp_init-0   [0] .0 1058611365: tracing_mark_write: C|1611035328|mutex|-202088853
       amp_init-0   [0] .0 1059570699: tracing_mark_write: B|0|mutex: lock mutex=0000000000000000 ticks=-202088853
       amp_init-0   [0] .0 1059577282: tracing_mark_write: E|0|mutex: unlock mutex=0000000000000000
       amp_init-0   [0] .0 1059585990: tracing_mark_write: B|0|mutex: lock mutex=0000000000000000 ticks=-202088853
       amp_init-0   [0] .0 1059591949: tracing_mark_write: E|0|mutex: unlock mutex=0000000000000000
       amp_init-0   [0] .0 1059598490: tracing_mark_write: B|0|mutex: lock mutex=0000000000000000 ticks=-202088853
       amp_init-0   [0] .0 1059623615: tracing_mark_write: E|0|mutex: unlock mutex=0000000000000000
  cpu-vring-ipi-1   [0] .0 1061413157: tracing_mark_write: B|1|mutex: lock mutex=0000000000000000 ticks=-202088853
  cpu-vring-ipi-1   [0] .0 1061424907: tracing_mark_write: E|1|mutex: unlock mutex=0000000000000000
  cpu-vring-ipi-1   [0] .0 1061429865: tracing_mark_write: B|1|mutex: lock mutex=0000000000000000 ticks=-202088853
  cpu-vring-ipi-1   [0] .0 1061433032: tracing_mark_write: E|1|mutex: unlock mutex=0000000000000000
  cpu-vring-ipi-1   [0] .0 1061440657: tracing_mark_write: B|1|mutex: lock mutex=0000000000000000 ticks=-202088853
  cpu-vring-ipi-1   [0] .0 1061444949: tracing_mark_write: E|1|mutex: unlock mutex=0000000000000000
  cpu-vring-ipi-1   [0] .0 1061446740: tracing_mark_write: B|1|mutex: lock mutex=0000000000000000 ticks=-202088853
  cpu-vring-ipi-1   [0] .0 1061452032: tracing_mark_write: E|1|mutex: unlock mutex=0000000000000000
```

从核异常状态自动恢复

从核异常状态自动恢复支持使用 AMP 软件看门狗或 AMP 硬件看门狗。

11.1 AMP 软件看门狗

异构通信框架支持 AMP 软件看门狗功能，即 rpmsg heartbeat 功能，早期也被称为心跳保活。当从核因为异常或跑飞等原因导致无法继续发送心跳数据给主核后，主核可以检测到从核此时未正常运行，并重新启动从核，配置步骤如下：

11.1.1 配置信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPMSG_HEARTBEAT。

从核 RTOS 环境需要启用配置 CONFIG_RPMSG_HEARBEAT。

需要注意的是从核[illegible]TE_NAME
dts 节点的名字保持一致，如下：

图 11-1: RTOS 环境的 remoteproc 名字

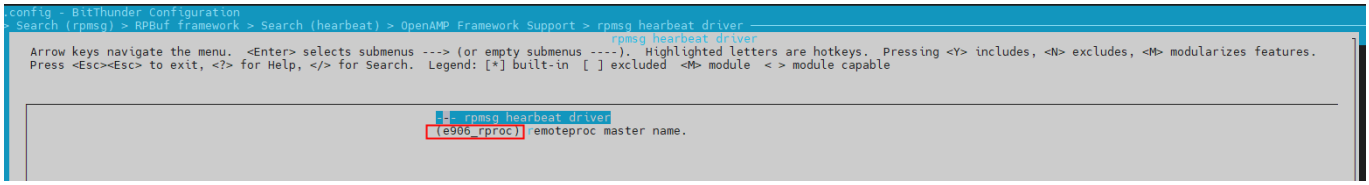
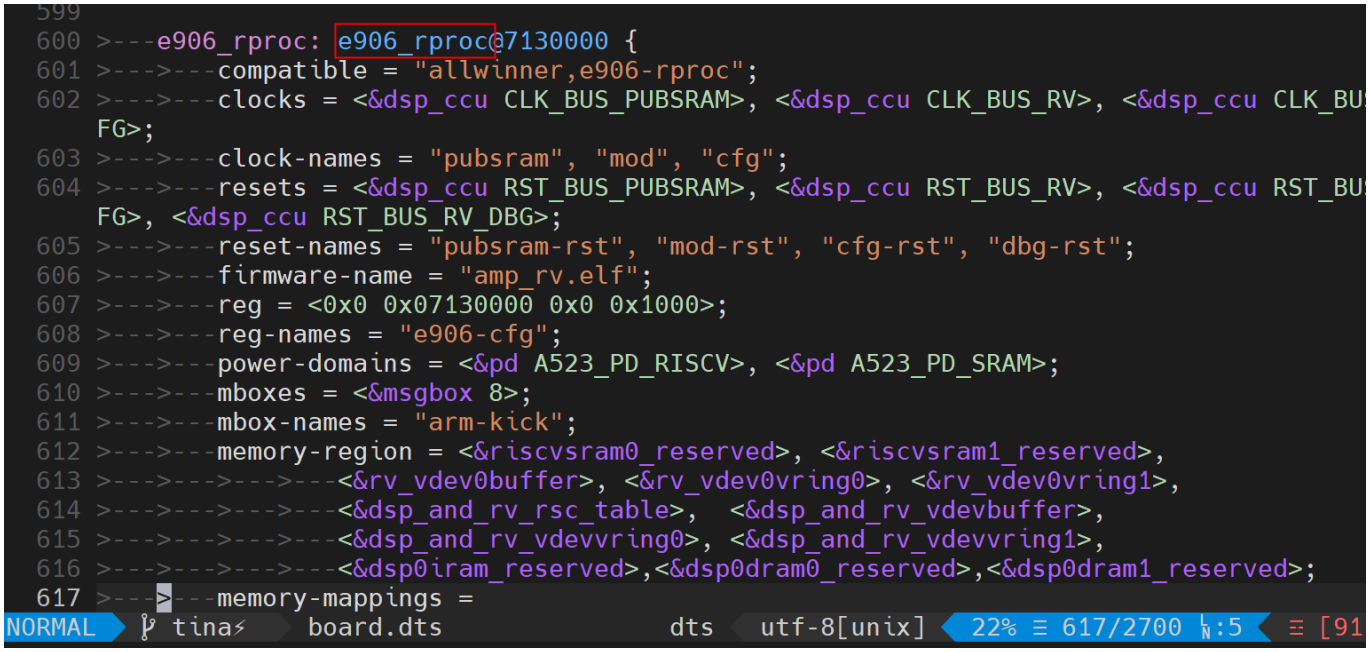


图 11-2: Linux 环境的 dts 节点名字



11.1.2 使用示例

在从核系统控制台执行 panic 命令让从核 crash，之后主核 Linux 端会自动重新启动从核，如下：

图 11-3: Linux 终端检测从核状态异常并重启从核现象

```
root@TinaLinux:/#
root@TinaLinux:/# [  391.213157] remoteproc remoteproc1: crash detected in e906_rproc: type watchdog
[  391.221389] remoteproc remoteproc1: handling crash #2 in e906_rproc
[  391.228460] remoteproc remoteproc1: recovering e906_rproc
[  391.235331] rpmsg_heartbeat virtio0.sunxi,rpmsg_heartbeat.-1.1025: virtio0.sunxi,rpmsg_heartbeat.-1.1025 is removed
[  391.247308] remoteproc remoteproc1: stopped remote processor e906_rproc
[  391.256217] remoteproc remoteproc1: sunxi_rproc_elf_load_segments,502 find segments err
[  391.266407]  remoteproc1#vdev0buffer: assigned reserved memory node vdev0buffer@42400000
[  391.276259] virtio_rpmsg_bus virtio0: rpmsg host is online
[  391.282452]  remoteproc1#vdev0buffer: registered virtio0 (type 7)
[  391.289292] remoteproc remoteproc1: remote processor e906_rproc is now up
[  391.366852] virtio_rpmsg_bus virtio0: creating channel sunxi,rpmsg_ctrl addr 0x400
[  391.378293] virtio_rpmsg_bus virtio0: creating channel sunxi,rpmsg_heartbeat addr 0x401

root@TinaLinux:/#
```

图 11-4: RTOS 终端异常状态自动恢复现象

```
dump_memory:x3

0x3FFEF8F0: 0x00000000 0x00000000 0x00000000 0x00000003
0x3FFEF900: 0x00000040 0x74726175 0x00000038 0x00000000
0x3FFEF910: 0x00000000 0x00000000 0x00000000 0x00000000
0x3FFEF920: 0x00000000 0x00000002 0x0000005e 0x74726175
0x3FFEF930: 0x00000039 0x00000000 0x00000000 0x00000000
0x3FFEF940: 0x00000000 0x00000000 0x00000000 0x00000002
0x3FFEF950: 0x00000072 0x73726576 0x006e6f69 0x00000000
0x3FFEF960: 0x00000000 0x00000000 0x00000000 0x00000000

dump_memory:x4

0xFFFFFFF0: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x00000000:
 ****************************************
 ** Welcome to MR527_E906 FreeRTOS V1.6.0    **
 ** Copyright (C) 2019-2021 AllwinnerTech **
 **                                      **
 **      starting riscv FreeRTOS         **
 ****************************************

Date:Mar 22 2023, Time:10:16:02
init Icache
init Dcache


FreeRTOS command server.
Type Help to view a list of registered commands.

>[AMP_INFO][rproc_arch_init:79]Init proc ok.
[AMP_INFO][rproc_arch_mmap:181]map pa(0x072af6b4) to va(0x3ffef6b4)
[AMP_INFO][openamp_sunxi_create_rproc:173]Wait master update resource_table
[AMP_INFO][openamp_sunxi_create_rproc:176]resource_table ready
[AMP_INFO][rproc_arch_mmap:181]map pa(0x42400000) to va(0x42400000)
[AMP_INFO][rproc_arch_mmap:181]map pa(0x42440000) to va(0x42440000)
[AMP_INFO][rproc_arch_mmap:181]map pa(0x42442000) to va(0x42442000)
[AMP_INFO][openamp_sunxi_create_rpmsg_vdev:286]Wait connected to remote master
[AMP_INFO][openamp_sunxi_create_rpmsg_vdev:292]Connecte to remote master Successed
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_ctrl ept ready
rpmsg ctrldev: Start Running...
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_heartbeat ept ready
Start Rpmsg Hearbeat Timer


cpu0>
```

11.2 AMP 硬件看门狗

由于 AMP 软件看门狗使用了 rpmsg，若 RTOS 端 rpmsg 通信出现问题会导致 Linux 端认为从核运行异常，因此使用硬件看门狗来检测从核运行状态会更加可靠。

信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPROC_SUBDEV 和 CONFIG_AW_RPROC_SUBDEV_WDT。

主核 Linux 环境 dts 配置中 remoteproc 节点需要增加 reg、reg-names、interrputs 以及 interrupt-names 这些 Linux 内核支持的标准属性：

```
diff --git a/configs/linux-5.15-origin/sun55iw6p1.dtsi b/configs/linux-5.15-origin/sun55iw6p1.dtsi
index 168801a72..e88c0b126 100644
--- a/configs/linux-5.15-origin/sun55iw6p1.dtsi
+++ b/configs/linux-5.15-origin/sun55iw6p1.dtsi
@@ -1870,7 +1870,7 @@

                e907_rproc: e907_rproc@1a00000 {
                        reg = <0x0 0x1a00000 0x0 0x1000>,
-                             <0x0 0x02002000 0x0 0x2000>;
+                             <0x0 0x01a02000 0x0 0x400>;
                        compatible = "allwinner,e907-rproc";
                        core-clock-frequency=<600000000>;
                        axi-clock-frequency = <300000000>;
@@ -1880,8 +1880,10 @@
                        reset-names = "rv-cfg-rst", "core-rst", "rv-sys-rst";

                        firmware-name = "amp_rv0.bin";
-                       reg-names = "rv-cfg", "ccmu";
+                       reg-names = "rv-cfg", "rproc_wdt_reg";

+                       interrupts = <GIC_SPI 185 IRQ_TYPE_LEVEL_HIGH>;
+                       interrupt-names = "rproc_wdt_irq";
                        status = "disabled";
                };
```

另外 remoteproc 节点内还需要增加 rproc_wdt 子节点以及相关属性：

```
diff --git a/configs/evb1/linux-5.15-origin/board.dts b/configs/evb1/linux-5.15-origin/board.dts
index 76fe48c..4d5bbce 100755
--- a/configs/evb1/linux-5.15-origin/board.dts
+++ b/configs/evb1/linux-5.15-origin/board.dts
@@ -771,6 +771,13 @@
                standby-ctrl-en = <0x1>;
                standby-record-reg = <0x07090114>;
                status = "okay";
+
+               rproc_wdt@0 {
+                       timeout_ms = <6000>;
+                       try_times = <3>;
+                       reset_type = <0x2>;
+                       status="okay";
+               };
        };
 };
```

rproc_wdt 子节点内各属性说明如下：

- timeout_ms：Linux 端初始配置 AMP 硬件看门狗时的超时时间 (单位为 ms)
- try_times：AMP 硬件看门狗检测到 remoteproc 状态异常后 Linux 端尝试重启 remoteproc 的次数
- reset_type：AMP 硬件看门狗复位类型，一般设置为 2，表示复位后发送中断，1 表示复位整个系统 (AMP 场景下一般不使用)
- status：rproc_wdt 的状态，设置为 “okay” 或 “ok” 时表示 AMP 硬件看门狗生效，其他值表示不生效

从核 RTOS 环境需要启用配置 CONFIG_COMPONENTS_AMP_HW_WATCHDOG。

且从核 RTOS 环境有如下配置可修改 AMP 硬件看门狗的相关参数：

- CONFIG_AMP_HW_WATCHDOG_TIMEOUT：RTOS 端启动后配置的 AMP 硬件看门狗超时时间 (单位为 s)，默认为 2s
- CONFIG_AMP_HW_WATCHDOG_FEED_INTERVAL：RTOS 端喂 AMP 硬件看门狗的间隔时间 (单位为 ms)，默认为 100ms
- CONFIG_AMP_HW_WATCHDOG_THREAD_PRIORITY：RTOS 端用于喂狗的线程的线程优先级，默认为 31

示例

在从核系统控制台执行 panic 命令让从核 crash，之后主核 Linux 端会自动重新启动从核，打印 “detected timeout & report crash, timeout_cnt: 1, try_times: 0” 类似日志：

```
[  350.039031] rproc_wdt remoteproc0-wdt: detected timeout & report crash, timeout_cnt: 1, try_times: 0
[  350.049421] remoteproc remoteproc0: crash detected in e907_rproc: type watchdog
[  350.057961] remoteproc remoteproc0: handling crash #4 in e907_rproc
[  350.065102] remoteproc remoteproc0: recovering e907_rproc
[  354.892586] sunxi-rproc 1a00000.e907_rproc: timeout return stop ack, e907 maybe crash
[  355.914205] sunxi-rproc 1a00000.e907_rproc: timeout return stop dead
[  355.922028] remoteproc remoteproc0: stopped remote processor e907_rproc
[  355.929886] remoteproc remoteproc0: firmware version: UTS - Mon, 03 Jun 2024 14:42:41 +0800
[  355.929886] Compile Time - 14:42:41
[  355.943513] sunxi-rproc 1a00000.e907_rproc: boot address: 0x60000000
[  355.950759] sunxi-rproc 1a00000.e907_rproc: core freq: 600000000Hz, axi freq: 300000000Hz
[  355.959979]  remoteproc0#vdev0buffer: assigned reserved memory node vdev0buffer@42400000
irtio_rpmsg_bus virtio0: rpmsg host is online
[  355.975880]  remoteproc0#vdev0buffer: registered virtio0 (type 7)
[  355.982729] remoteproc remoteproc0: remote processor e907_rproc is now up
[  356.247916] virtio_rpmsg_bus virtio0: creating channel sunxi,rpmsg_ctrl addr 0x400
[  356.270420] virtio_rpmsg_bus virtio0: creating channel rpbuf-service addr 0x401
[  356.279222] rpbuf_service_rpmsg virtio0.rpbuf-service.-1.1025: rpmsg device parent 0: virtio0
[  356.288841] rpbuf_service_rpmsg virtio0.rpbuf-service.-1.1025: rpmsg device parent 1: remoteproc0#vdev0buffer
[  356.299976] rpbuf_service_rpmsg virtio0.rpbuf-service.-1.1025: rpmsg device parent 2: remoteproc0
[  356.309943] rpbuf_service_rpmsg virtio0.rpbuf-service.-1.1025: rpmsg device parent 3: 1a00000.e907_rproc
```

另外在让从核 crash 之前使用命令 uevent-monitor-for-rproc_wdt rproc_wdt 可以监测到 rproc_wdt 设备上传的 uevent：

```
root@TinaLinux:/# uevent-monitor-for-rproc_wdt rproc_wdt
addr.nl_groups: ffffffff
socket_fd = 3
 1749: remoteproc0-wdt                , rproc_wdt , change  , WDT_TIMEOUT , (type: rst_core  , timeout_ms:   6000, try_times:
     0, timeout_cnt:   4)
 1760: remoteproc0-wdt                , rproc_wdt , change  , WDT_STOPPED , (type: rst_core  , timeout_ms:   6000, try_times:
     0, timeout_cnt:   4)
 1761: remoteproc0-wdt                , rproc_wdt , change  , WDT_RUNNING , (type: rst_core  , timeout_ms:   6000, try_times:
     0, timeout_cnt:   4)
```

技巧

使用 uevent-monitor-for-rproc_wdt 命令需要启用软件包配置 CONFIG_PACKAGE_uevent-monitor-for-rproc_wdt

从核资源回收

从核使用的一些外设资源可能需要在从核重启 (从核异常状态重启，主核主动重启) 时回收 (一般是恢复为默认状态，不会将这些从核专用的资源又分配给主核使用)，目前外设复位功能支持回收从核使用的硬件外设资源。

12.1 外设复位功能

外设复位功能基于 Linux remoteproc 框架的子设备实现，主核会在从核 crash 后复位其使用到的外设。

12.1.1 配置信息

由于是主核负责回收从核使用的外设资源，故只需要在主核做相关配置。

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPROC_SUBDEV、CONFIG_AW_RPROC_SUBDEV_PERI_RST。

dts 配置中需要增加从核使用的外设的复位信息：

```
diff --git a/configs/evb1/linux-5.15-origin/board.dts b/configs/evb1/linux-5.15-origin/board.dts
index b12ce8b..2b54357 100644
--- a/configs/evb1/linux-5.15-origin/board.dts
+++ b/configs/evb1/linux-5.15-origin/board.dts
@@ -471,6 +471,9 @@
                standby-record-reg = <0x07090114>;
                share-irq = "e907";
                status = "okay";
+               peri_reset {
+                       resets = <&ccu RST_BUS_TIMER0>, <&ccu RST_BUS_UART7>;
+               };
        };
 };
```

外设的复位信息可以在对应芯片平台的 dtsi 文件里查找，比如从核使用了 uart7，可以在 dtsi 里查找 uart7 的节点，其节点内一般会有 resets 属性包含其复位信息。

从核获取用户资源

从核在运行过程中可能需要根据一些用户资源 (用户自定义的数据信息) 来执行对应的操作，这些用户资源一般存放在存储器 (Flash 或 eMMC 等) 里，但目前在从核 RTOS 系统上是无法直接去读取存储器，因为对应的控制器外设一般都是分配给了主核 Linux 端，从核不能去操作对应的控制器，否则会导致 Linux 端操作控制器时出现异常。一般推荐的做法是在主核 Linux 端使用 rpmsg 等核间通信方式将存储器上的数据传给从核使用。但如果从核在 bootloader 阶段启动，且从核启动后需要立即使用到存储器里的数据，此时 Linux 还未启动完成，故推荐做法就无法满足这种需求了，需要使用 AMP User Resource 功能。

技巧

AMP User Resource 功能目前只支持保存在存储器分区上的用户资源。

13.1 配置信息

主核 Linux 环境需要启用配置 CONFIG_AW_RPROC_USER_RESOURCE
置 CONFIG_AW_RPROC_SUBDEV 和 CONFIG_AW_RPROC_SUBDEV_RSC_HELPER。

从核 RTOS 环境需要启用配置 CONFIG_COMPONENTS_AMP_USER_RESOURCE。

且从核 RTOS 环境目前最多可指定 4 个用户资源，每个用户资源有 3 个配置，这 3 个配置的相关说明如下 (配置名中的 X 表示用户资源的 ID，目前可取 0 到 3)：

- CONFIG_AMP_USER_RESOURCE_X：启用用户资源 X
- CONFIG_AMP_USER_RESOURCE_X_PARTITION_NAME：保存用户资源 X 的存储器分区名
- CONFIG_AMP_USER_RESOURCE_X_DATA_LEN：用户资源 X 的数据长度

注意

源的 buffer 来自于 RTOS 环境的一个全局数组，定义在 rtos/lichee/rtos-components/aw/amp_app/amp_user_resource/urce.c 文件里，故配置 CONFIG_AMP_USER_RESOURCE_X_DATA_LEN 的值决定了此全局数组占用的内存大小。

13.2 API 说明

由于用户资源主要是从核 RTOS 端使用，故提供如下用于从核 RTOS 环境的 API。

13.2.1 amp_user_resource_t 数据结构

amp_user_resource_t 表示 AMP 用户资源：

```
typedef struct amp_user_resource
{
        uint32_t id;
        int type;
        const char *part_name;

        uint32_t len;
        unsigned char *buf;
} amp_user_resource_t;
```

成员说明：

- id：用户资源的 ID
- type：用户资源的类型，目前只支持保存在存储器分区上的用户资源，对应类型值为 255
- part_name：保存用户资源的存储器分区名
- len：用户资源的数据长度
- buf：保存用户资源的 buffer 地址

指定 ID 对应的用户资源

```
int get_amp_user_resource(uint32_t id, amp_user_resource_t *user_rsc);
```

参数：

- id：用户资源 ID
- user_rsc：指向获取到的用户资源的指针

返回值：

- 0：获取成功
- 其他值：获取失败

指定 ID 对应的用户资源信息

```
void show_user_resource(uint32_t id);
```

参数：- id：用户资源 ID

13.2.4 打印当前系统所有的用户资源信息

```
void show_all_user_resource(void);
```

13.3 API 使用方法

AMP 用户资源使用起来比较简单，步骤如下：

p_user_resource 用户资源的想要获取的

2. 根据具体需求使用 AMP 用户资源

13.4 使用示例

在从核启动后，会打印出当前系统所有的用户资源信息：

```
AMP User Resource:
ID: 0
Type: 255
Partition Name: 'riscv0'
Buffer Addr: 6003B648
Data Length: 256
Data:
0x6003B648: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
0x6003B658: 02 00 F3 00 01 00 00 00 00 00 00 60 34 00 00 00 |...........`4...|
C703000500000034002000010028 00|,.......4....(.|
0x6003B678: 0D 00 0C 00 01 00 00 00 00 10 00 00 00 00 00 60 |...............`|
0x6003B688: 00 00 00 60 48 B6 03 00 64 81 04 00 07 00 00 00 |...`H...d.......|
0x6003B698: 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6A8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6B8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6C8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6E8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B6F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B708: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B718: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B728: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
0x6003B738: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

主核 Linux 端若在运行时动态修改了用户资源对应的分区里的数据，则可通过重启从核来让从核获取最新的用户资源数据。

主核和从核获取相同时基的时间戳

AMP 场景下主核和从核在有些情况下需要获取相同时间基准的时间戳，比如通信耗时测量，故 SDK 中提供了 AMP 时间戳功能来满足这种需求。

14.1 配置信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_AMP_TIMESTAMP。

主核 Linux 环境 dts 配置中需要增加 AMP 时间戳设备对应的 amp_ts 节点以及相关属性：

```
diff --git a/configs/linux-5.15-origin/sun55iw6p1.dtsi b/configs/linux-5.15-origin/sun55iw6p1.dtsi
index ef7294fb6..302c95b95 100644
--- a/configs/linux-5.15-origin/sun55iw6p1.dtsi
+++ b/configs/linux-5.15-origin/sun55iw6p1.dtsi
@@ -1947,6 +1947,15 @@
                        status = "disabled";
 };


+               amp_ts0: amp_ts@8020000 {
+                       compatible = "sunxi,amp_timestamp";
+                       reg = <0x0 0x08020000 0x0 0x1000>;
+                       id = <0>;
+                       counter_type = <ARM_ARCH_COUNTER_TYPE>;
+                       count_freq = <24000000>;
+                       status = "disabled";
+               };
+
                hwspinlock: hwspinlock@3005000 {
                        compatible = "allwinner,sunxi-hwspinlock";
                        reg = <0x0 0x3005000 0x0 0x1000>;
```

amp_ts 节点内各属性 (除 Linux 内核支持的标准属性外) 说明如下：

间戳设备 ID

- counter_type：计数器类型，ARM 架构的类型值为 ARM_ARCH_COUNTER_TYPE，RV 架构的类型值为 THEAD_RISCV_ARCH_COUNTER_TYPE
- count_freq：计数频率，单位 Hz

另外若需要使用下面章节提到的 of_amp_ts_get_dev API 来获取时间戳设备，则需要在使用到 AMP 时间戳设备的设备树节点里增加如下属性：

```
amp_ts_dev = <&amp_ts0>;
```

从核 RTOS 环境需要启用配置 CONFIG_COMPONENTS_AMP_TIMESTAMP。

且从核 RTOS 环境有如下配置可配置 AMP 时间戳设备的相关参数：

- CONFIG_AMP_TS_DEV_0：启用 AMP 时间戳设备 0

P_TS_DEV_0_COUNTER_REG_ADDR：AMP 时间戳设备 0 对应的计数器的寄存器地址，可在芯片用户手册内存映射章节里找到带 TIMESTAMP 的模块，例如对于 ARM 架构计数器，其模块名为 TIMESTAMP_CTRL)

- CONFIG_AMP_TS_DEV_0_COUNT_FREQ：AMP 时间戳设备 0 的计数频率
- CONFIG_AMP_TS_DEV_0_COUNTER_TYPE：AMP 时间戳设备 0 对应的计数器类型，ARM 架构计数器对应的值为 0，RISCV 架构计数器对应的值为 1

注意

要获取相同时基的时间戳，主核和从核配置中 AMP 时间戳设备的参数要一致（也即使用同一个时间戳硬件），也就是 amp_ts0 设备树节点中的 reg 节点描述的地址范围要和 CONFIG_AMP_TS_DEV_0_COUNTER_REG_ADDR 配置一致，设备树中的 counter_type 节点要和 CONFIG_AMP_TS_DEV_0_COUNTER_TYPE 配置一致，设备树中的 count_freq 节点要和 CONFIG_AMP_TS_DEV_0_COUNT_FREQ 一致。

说明

目前只提供了 Linux 环境内核态 API 和 RTOS 环境 API，暂未提供 Linux 环境用户态 API。

14.2.1 Linux 环境内核态 API

14.2.1.1 amp_ts_dev_t 数据结构

amp_ts_dev_t 标识一个 AMP 时间戳设备：

```
typedef void *amp_ts_dev_t;
```

其实际上是一个指针类型，方便快速获取到 AMP 时间戳设备对象，减少获取时间戳过程中的耗时。

14.2.1.2 获取指定 ID 对应的 AMP 时间戳设备

```
int amp_ts_get_dev(uint32_t id, amp_ts_dev_t *dev);
```

参数：

- id：AMP 时间戳设备 ID
- dev：指向获取到的 AMP 时间戳设备的指针

返回值：

- 0：获取成功
- 其他值：获取失败

14.2.1.3 获取指定设备树节点对应的 AMP 时间戳设备

```
int of_amp_ts_get_dev(struct device_node *np, amp_ts_dev_t *dev);
```

参数：

- np：指向指定的设备树节点的指针
- dev：指向获取到的 AMP 时间戳设备的指针

返回值：

- 0：获取成功
- 其他值：获取失败

此 API 为设备树相关的辅助 API，方面用户快速获取 AMP 时间戳设备，只需要在使用到 AMP 时间戳设备的设备树节点里增加如下属性后即可使用此 API 快速获取 AMP 时间戳设备：

```
amp_ts_dev = <&amp_ts0>;
```

14.2.1.4 获取时间戳

```
int amp_ts_get_timestamp(amp_ts_dev_t dev, uint64_t *timestamp);
```

参数：

- dev：AMP 时间戳设备
- timestamp：指向获取到的时间戳的指针，单位：us

- 0：获取成功
- 其他值：获取失败

14.2.1.5 获取 AMP 时间戳设备的原始计数值

```
int amp_ts_get_count_value(amp_ts_dev_t dev, uint64_t *count_value);
```

参数：

- dev：AMP 时间戳设备
- count_value：指向获取到的计数值的指针

返回值：

- 0：获取成功
- 其他值：获取失败

14.2.1.6 获取 AMP 时间戳设备的计数频率

```
int amp_ts_get_count_freq(amp_ts_dev_t dev, uint32_t *freq);
```

参数：

- dev：AMP 时间戳设备
- freq：指向获取到的计数频率的指针，单位：Hz

返回值：

- 0：获取成功
- 其他值：获取失败

14.2.2 RTOS 环境 API

由于没有 Linux 那样的标准驱动模型，RTOS 环境除新增了 1 个用于初始化系统内的 AMP 时间戳设备的 API 外，其余 API 和 Linux 环境下的 API 一致。

14.2.2.1 初始化系统内的 AMP 时间戳设备

```
int amp_timestamp_init(void);
```

参数：无

返回值：

- 0：初始化成功
- 其他值：初始化失败

14.3 API 使用方法

AMP 时间戳使用起来比较简单 (Linux 环境内核态和 RTOS 环境用法一致)，步骤如下：

1. 调用 amp_ts_get_dev 获取 AMP 时间戳设备，传入想要获取的 AMP 时间戳设备的 ID
2. 调用 amp_ts_get_timestamp 获取当前时间戳
3. 根据具体需求使用获取到的 AMP 时间戳

空间获取时间戳

用户可通过 sysfs 文件获取 AMP 时间戳，以 MR536 为例，可执行命令 cat /sys/devices/platform/soc@3000000/8020000.amp_ts/timestamp 获取 AMP 时间戳，执行命令会输出 “timestamp: 201721852us, 201721ms” 类似结果。由于 cat 操作会消耗一定的时间，这样获取的时间戳不是实时的，这个 cat 操作只能用于判断时间戳模块是否正常在计时。

用户如果需要在用户空间获得时间戳，想实现 linux 应用层与 rv 端统一时间轴，可以参考以下的代码，这里的原理是：内核中会计算时间戳和 ktime 的偏移值，用户空间读取偏移值，然在用户空间通过 clock_gettime 获取时间再加上偏移值来反推时间戳的时间。在使用这种方法反推时间戳时，注意要在启动以及休眠唤醒后，用户空间必须重新读取并更新用户空间的 offset 偏移值，并且每隔一段时间更新 offset 偏移值（例如 30s）。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include<errno.h>
#include <string.h>
#include <time.h>
#define IOCTL_GET_S64_VALUE _IOR(0xbb, 1, int64_t) // 用于读取s64值

// 定义一个接口，获取当前时间并转换为微秒
long long get_time_in_microseconds(clockid_t clk_id) {
    struct timespec tp;

    // 调用 clock_gettime 获取当前时间
    if (clock_gettime(clk_id, &tp) == -1) {
        perror("clock_gettime");
        return -1;  // 返回 -1 表示获取时间失败
    }

    // 将 seconds 和 nanoseconds 转换为总的微秒数
    long long total_microseconds = tp.tv_sec * 1000000LL + tp.tv_nsec / 1000;

    return total_microseconds;  // 返回微秒值
}

int main() {
    int fd;
    int64_t value;
    int ret;
    long long microseconds=0;

    // 打开设备文件
    fd = open("/dev/amp_ts_0", O_RDONLY);
    if (fd < 0) {
        perror("Failed to open device file");
        return -1;
    }
    // 执行ioctl操作读取一个int64_t类型的值
    ret = ioctl(fd, IOCTL_GET_S64_VALUE, &value);

        perror("ioctl failed");
        close(fd);
        return -1;
    }


    // 打印读取的s64值
    printf("Read offset: %ld\n", value);

    microseconds = get_time_in_microseconds(CLOCK_MONOTONIC);

    printf("amp ts(us) = %lld\n", microseconds + value);
    // 关闭设备文件
    close(fd);
```

主核和从核共享中断

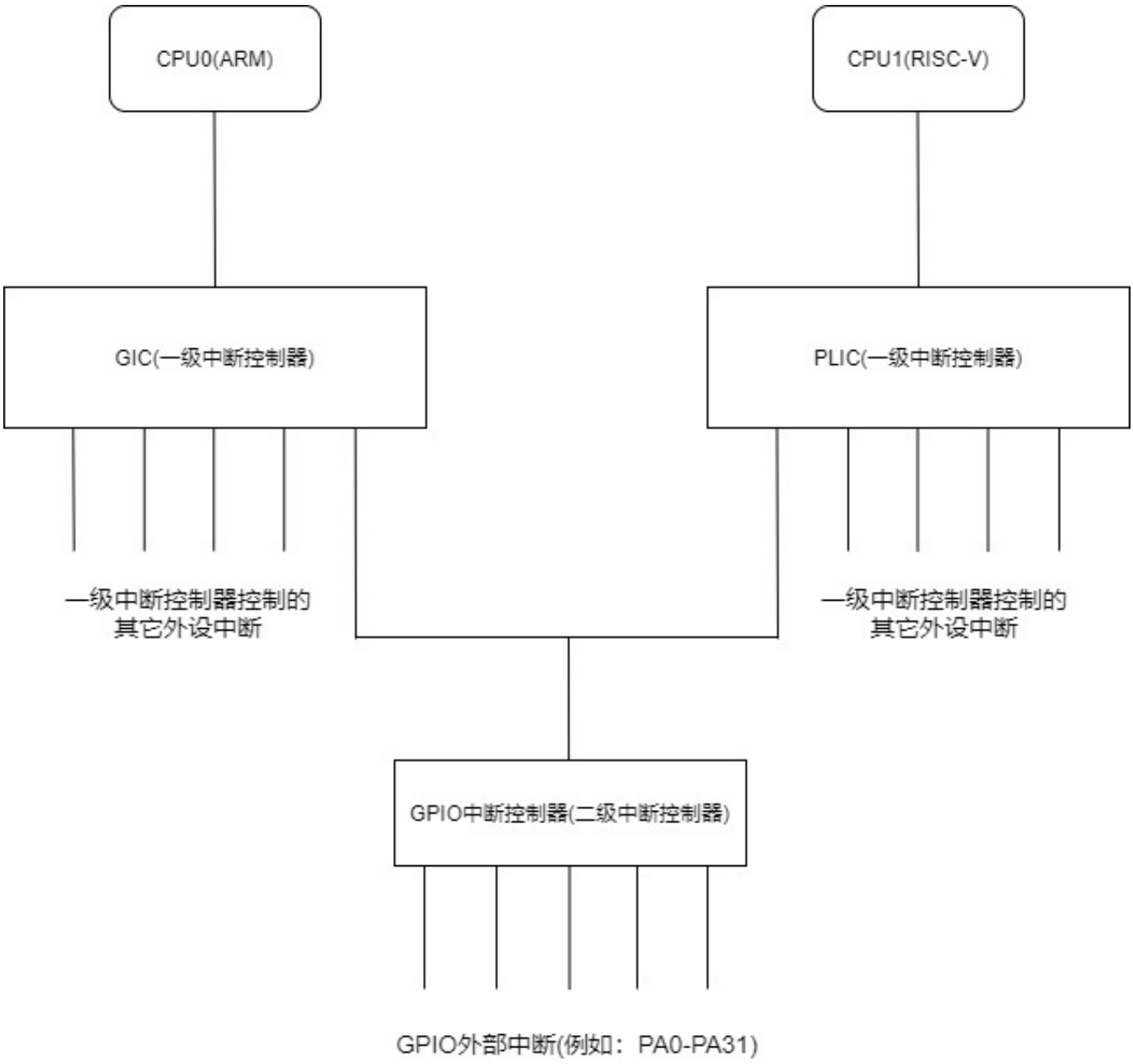
15.1 背景

随着多核异构处理器的流行，外设中断的分配成为 AMP 场景下一个急需解决的问题，一级中断控制器控制的外设中断的分配在 AMP 场景下比较容易解决，由于一级中断控制器一般是每个 CPU 对应一个，不同 CPU 上运行的软件提前协商好某个外设中断在哪个 CPU 上处理，其他 CPU 对应的一级中断控制器禁用此外设中断即可。

但到了二级和二级以上的中断控制器时就无法简单的通过在某个 CPU 上禁用中断控制器控制的某个外设中断解决了，这些中断控制器一般是全局的，并非每个 CPU 一个，以二级中断控制器为例：二级中断控制器汇聚多个中断源为一个中断信号后同时连接到多个 CPU 对应的一级中断控制器上，这些二级中断控制器控制的不同中断只要有其中一个触发，多个 CPU 都会进入对应的中断处理函数，即使触发的中断并不是分配给当前 CPU 的。

二级中断控制器的典型示例为 GPIO 中断控制器：GPIO 控制器内集成了一个比较简单的中断控制器，其将一个 GPIO 组内的所有 GPIO 的外部中断信号然后连接到一级中断控制器上。如下图所示。

图 15-1: 中断控制器级联示例



AMP 场景下外设中断的分配问题，在软件层面上设计了共享中断模块。

15.2 工作原理

共享中断模块使用一块共享内存来记录某个外设中断被分配到哪个核处理，且在对应中断控制器的驱动代码里根据中断分配信息判断当前中断是否由本 CPU 核处理。

15.3 配置说明

使用共享中断功能一般情况下需要配置 Linux 和 RTOS，特殊情况下还需要配置 u-boot。

15.3.1 RTOS

勾选 CONFIG_AMP_SHARE_IRQ。CONFIG_SHARE_IRQ_TABLE_IN_HEADER_FILE 为可选项：

```
nts --->

  thirdparty components  --->
    OpenAMP Support  --->
      [*]     openamp share interrupts support
      [ ]       use share irq table in header file ( 可选 )
```

use share irq table in header file 配置为可选项，若在共享中断资源表未更新前需要使用共享中断，需要把此配置选上。选上此配置后，会在构建阶段，提前解析 dts 配置，并生成 rtos/lichee/rtos-components/thirdparty/openamp/include/openamp/share-irq-dt.h 头文件，文件中包含有中断类型、中断分配信息掩码、中断号这三个关键信息。

15.3.2 Linux

15.3.2.1 内核配置

勾选 CONFIG_SUNXI_RPROC_SHARE_IRQ：

```
Allwinner BSP  --->
  Device Drivers  --->
    Remoteproc drivers  --->
      [*]   SUNXI remote share irq support
```

15.3.2.2 dts 配置

主要需要增加 3 个配置：

1、在保留内存节点 “reserved-memory” 里增加一个节点用于描述保存中断分配信息的保留内存；

```
reserved-memory {
    e906_share_irq_table: share_irq_table@4244c000 {
            reg = <0x0 0x4244c000 0x0 0x1000>;
            no-map;
```

2、根节点下增加 reserved-irq 节点，并在 reserved-irq 节点下增加具体的中断分配信息节点；

```
/{
    reserved-irq {
            share-e906 {
                    arch-name = "e906";
                    memory-region = <&e906_share_irq_table>;
                    /* defined by sun55iw3-share-irq-dt.h */
                    share-irq =
                            <1      0x2     RISCV_IRQn(98, 6)       ARM_IRQn(101) 0x00000000>, /* GPIOB */
                            <2      0x3     RISCV_IRQn(98, 8)       ARM_IRQn(103) 0x00000000>, /* GPIOC */
                            <3      0x4     RISCV_IRQn(99, 2)       ARM_IRQn(105) 0x00000000>, /* GPIOD */
                            <4      0x5     RISCV_IRQn(99, 4)       ARM_IRQn(107) 0x00000000>, /* GPIOE */
                            <5      0x6     RISCV_IRQn(99, 6)       ARM_IRQn(109) 0x00000000>, /* GPIOF */
                            <6      0x7   ARM_IRQn(IRQn(0x00000000>,/*GPIOG*/
                            <7      0x8     RISCV_IRQn(100, 2)      ARM_IRQn(113) 0x00000000>, /* GPIOH */
                            <8      0x9     RISCV_IRQn(100, 4)      ARM_IRQn(115) 0x00000000>, /* GPIOI */
                            <9      0xa     RISCV_IRQn(100, 6)      ARM_IRQn(117) 0x00000000>, /* GPIOJ */
                            <10     0xb     RISCV_IRQn(107, 5)      ARM_IRQn(172) 0x00000000>, /* GPIOK */
                            <11     0xc     RISCV_IRQn(63, 0)       ARM_IRQn(191) 0x00000000>, /* GPIOL */
                            <12     0xd     RISCV_IRQn(65, 0)       ARM_IRQn(193) 0x00000000>; /* GPIOM */
            };
    };
};
```

中断分配信息节点中有 3 个属性：

1). arch-name，用于标识此中断分配信息用于哪个 remoteproc，对应 remoteproc 里的 share-irq 属性值需要和此属性一致；

2). memory-region，用于标识保存此中断分配信息的内存区域；

属性为数组外[illegible]述一个

组内第 1 个数值为 id，需要和 share_irq_table 数组中每个元素的 major 成员的值一致，对于 sun55iw3，share_irq_table 数组可在 bsp/drivers/remoteproc/sunxi_share_interrupt/xxx-share-irq.h 里找到；

组内第 2 个数值为中断类型，从 1 开始为 GPIOA 中断，2 为 GPIOB 中断，依次类推；

组内第 3 个数值为远端处理器 (从核) 的中断号，生成中断号的宏可在 bsp/include/dt-bindings/gpio/xxx-share-irq-dt.h；

组内第 4 个数值为本端处理器 (主核) 的中断号，生成中断号的宏可在 bsp/include/dt-bindings/gpio/xxx-share-irq-dt.h；

组内第 5 个数值为中断分配信息的掩码，表明对应的 GPIO 组内的 GPIO 外部中断是分配给本端处理器 (对应 bit 为 0)，还是分配给远端处理器 (对应 bit 为 1)。

3、相应 remoteproc 节点下增加 share-irq 属性，属性值和中断分配信息节点里的 arch-name 属性值一致，另外还需在原有的 memory-region 属性值的基础上增加保留内存对应的内存区域。

```
&soc {
    e906_rproc: e906_rproc@7130000 {
        memory-region = <&rv_sram_reserved>, <&rv_vdev0buffer>, <&rv_vdev0vring0>, <&rv_vdev0vring1>, <&e906_share_irq_table>;
        share-irq = "e906";
    };
};
```

15.3.3 u-boot

若由 u-boot 启动 remoteproc，则 u-boot 配置里需要勾选 CONFIG_RISCV_UPDATA_IRQ_TAB

```
Device Drivers  --->
    [*] Support RISCV
    [*]   Support RISCV share irq table updata
```

技巧

早期平台并不支持共享中断，目前支持共享中断的平台有 mr536、v821、v821b、r528、r528s2、ai985。MR153 采用 AMP System Resource Manager 对中断进行管理，不使用共享中断。将来共享中断会被 AMP System Resource Manager 替代。

15.4 配置示例

下面给出 PB0、PD1、PE2、PK10、PK12 这 5 个 GPIO 外部中断分配给远端处理器，其他 GPIO 外部中断分配给本端处理器的 dts 配置示例：

```
/{
    reserved-memory {
        e906_share_irq_table: share_irq_table@4244c000 {
                reg = <0x0 0x4244c000 0x0 0x1000>;
                no-map;



    reserved-irq {
            share-e906 {
                    arch-name = "e906";
                    memory-region = <&e906_share_irq_table>;
                    /* defined by sun55iw3-share-irq-dt.h */
                    share-irq =
                            <1      0x2     RISCV_IRQn(98, 6)       ARM_IRQn(101) 0x00000001>, /* GPIOB */
                            <2      0x3     RISCV_IRQn(98, 8)       ARM_IRQn(103) 0x00000000>, /* GPIOC */
                            <3      0x4     RISCV_IRQn(99, 2)       ARM_IRQn(105) 0x00000002>, /* GPIOD */
                            <4      0x5     RISCV_IRQn(99, 4)       ARM_IRQn(107) 0x00000004>, /* GPIOE */
                            <5      0x6     RISCV_IRQn(99, 6)       ARM_IRQn(109) 0x00000000>, /* GPIOF */
                            <6      0x7     RISCV_IRQn(99, 8)       ARM_IRQn(111) 0x00000000>, /* GPIOG */
                            <7      0x8     RISCV_IRQn(100, 2)      ARM_IRQn(113) 0x00000000>, /* GPIOH */
```

```
        0x9 RISCV_IRQn(100, 43)          ARM_IRQn(115) 0x00000000>, /* GPIOI */
        0xa RISCV_IRQn(100, 69)                                     ARM_IRQn(117) 0x00000000>, /* GPIOJ */
                          <10     0xb     RISCV_IRQn(107, 5)       ARM_IRQn(172) 0x00001400>, /* GPIOK */
                          <11     0xc     RISCV_IRQn(63, 0)        ARM_IRQn(191) 0x00000000>, /* GPIOL */
                          <12     0xd     RISCV_IRQn(65, 0)        ARM_IRQn(193) 0x00000000>; /* GPIOM */
           };
    };
};

&soc {
    e906_rproc: e906_rproc@7130000 {
        memory-region = <&rv_sram_reserved>, <&rv_vdev0buffer>, <&rv_vdev0vring0>, <&rv_vdev0vring1>, <&e906_share_irq_table>;
        share-irq = "e906";
    };
};
```

配置完成后在 RTOS 下可通过 dump_amp_share_irq 命令 (早期 SDK 里为 dump_shirq 命令) 查看当前 GPIO 外部中断的分配信息：

```
cpu0>dump_amp_share_irq
     :Status  Type   Flags     IRQ
   69:Enable  GPIO   0x00000001 9806
   71:Enable  GPIO   0x00000000 9808
   73:Enable  GPIO   0x00000002 9902
   75:Enable  GPIO   0x00000004 9904
   77:Enable  GPIO   0x00000000 9906
   79:Enable  GPIO   0x00000000 9908
   81:Enable  GPIO   0x00000000 10002
   83:Enable  GPIO   0x00000000 10004
   85:Enable  GPIO   0x00000000 10006
  140:Enable  GPIO   0x00001400 10705
  159:Enable  GPIO   0x00000000 6300
  161:Enable  GPIO   0x00000000 6500

GPIO group interrupt allocation info:
GPIOB, allocation mask: 0x00000001, IRQ:9806
ion mask: 0x00000000, IRQ:9808
GPIOD, allocation mask: 0x00000002, IRQ:9902
GPIOE, allocation mask: 0x00000004, IRQ:9904
GPIOF, allocation mask: 0x00000000, IRQ:9906
GPIOG, allocation mask: 0x00000000, IRQ:9908
GPIOH, allocation mask: 0x00000000, IRQ:10002
GPIOI, allocation mask: 0x00000000, IRQ:10004
GPIOJ, allocation mask: 0x00000000, IRQ:10006
GPIOK, allocation mask: 0x00001400, IRQ:10705
GPIOL, allocation mask: 0x00000000, IRQ:6300
GPIOM, allocation mask: 0x00000000, IRQ:6500
```

可以看到分配信息和 dts 配置里的一致。

15.5 注意事项

1、共享中断模块目前只支持 GPIO 中断控制器这个二级中断控制器控制的中断，一级中断控制器控制的外设中断不支持。

2、若同一个 GPIO 组内的不同 GPIO 外部中断分配到超过 2 个 CPU 核上，可通过新增中断分配信息节点并正确设置中断分配信息的掩码来支持这种场景。

3、若一个 GPIO 组内使用到的 GPIO 对应的外部中断只会在某一个 CPU 上处理，可将剩余未复用为外部中断功能的 GPIO 对应的外部中断也分配给这个 CPU 处理，此时另外一个 CPU 不会注册相关的 GPIO 中断处理函数，可避免另外的 CPU 不必要的进入 GPIO 中断处理函数

从核使用的内存区域

当前 SDK 中从核使用的内存区域按使用者分类主要分为专用内存区域和共享内存区域，其中共享内存区域主要用于核间通信、共享中断等功能。

16.1 专用内存区域

从核使用的专用内存区域主要包含从核的运行内存区域，不排除未来可能新增其他专用内存区域。

16.1.1 运行内存区域

运行内存区域是 RTOS 环境下具体方案的链接器脚本文件中 MEMORY 语法里指定的内存区域，以 MR536 为例，其原始链接器脚本文件 (rtos/lichee/rtos/projects/mr536_e907/evb1/freertos.lds.S) 中的运行内存区域指定为如下：

```
/* Linker script to configure memory regions. */
MEMORY
{
    RAM (rwx)   : ORIGIN = CONFIG_ARCH_START_ADDRESS, LENGTH = CONFIG_ARCH_MEM_LENGTH
}
```

可以看到上面定义了一块名为 RAM 的内存，其起始地址为 CONFIG_ARCH_START_ADDRESS，长度为 CONFIG_ARCH_MEM_LENGTH，这 2 个宏来自于 RTOS 配置。

上面提到的链接器脚本文件是原始文件，经过预处理后才是最终的链接器脚本文件（路径为 build/mr536_e907_evb1/projects/mr536_e907/evb1/freertos.lds），如下：

```
MEMORY
{
    RAM (rwx) : ORIGIN = 0x60000000, LENGTH = 0x100000
}
```

可以看到名为 RAM 的内存的起始地址为 0x60000000，长度为 0x100000，根据芯片用户手册，此区域对应 DRAM，由于 DRAM 一般是给主核[illegible]内存给从核使用：

```
reserved-memory {
    e907_dram_reserved: e907_dram@60000000 {
        reg = <0x0 0x60000000 0x0 0x00100000>;
        no-map;
    };
};
```

16.1.1.1 修改从核的运行内存区域

1. 查阅芯片用户手册，确认从核可以访问的 DRAM 区域的地址范围。例如：MR536 平台上，从核可访问 DRAM 区域的地址范围是：0x4000 0000-0xFFFF FFFF
2. 配置 dts，将 DRAM 上的某个内存区域保留下来。保留下来的内存区域是给从核专用的，主核不会将这块内存作为系统内存使用。例如：为 MR536 的从核 E907 预留 2MB 的内存空间，内存起始地址为 0x70000000，大小为 0x00200000。其中，no-map 属性表示这些内存区域不[illegible]间中加载固件时会映射

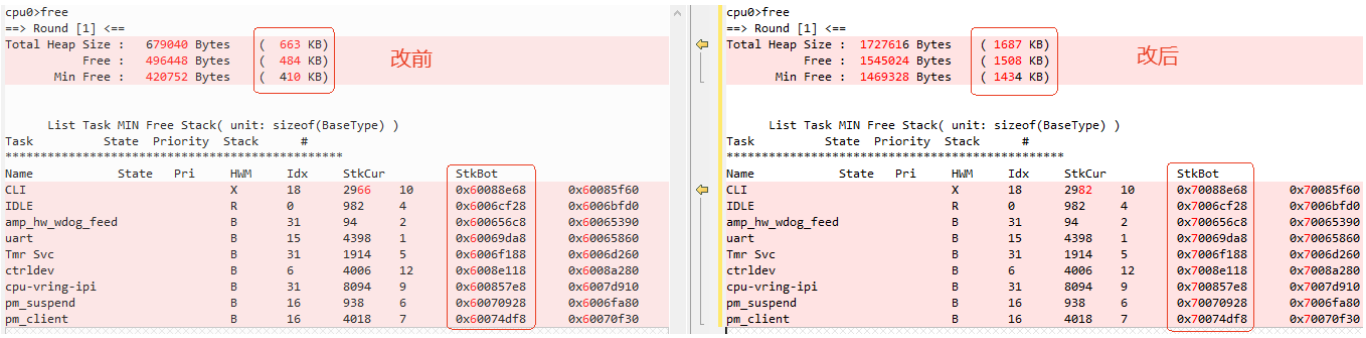
```
reserved-memory {
...
  e907_dram_reserved: e907_dram@60000000 {
   reg = <0x0 0x70000000 0x0 0x00200000>;
   no-map;
  };
...
};
```

3. 修改从核运行内存起始地址和内存长度。在 Tina 环境下，运行 `mrtos menuconfig` ，设置 CONFIG_ARCH_START_ADDRESS 和 CONFIG_ARCH_MEM_LENGTH，与设备树预留内存区域一致。

```
TART_ADDRESS=0x70000000
EM_LENGTH=0x200000
```

4. 在 Tina 环境下，执行 `mrtos` 编译 rtos，执行 `mkernel` 编译 kernel（目的是：编译 dts）。
5. 在 Tina 环境下，执行 `pack`，打包完整镜像。
6. 烧录完整镜像。
7. 在 RTOS 端通过 free 命令查看内存的容量的大小，以及栈地址范围。例如：在 MR536 平台，通过 free 查看剩余内存以及栈地址，对比修改前的 0x60000000-0x60000000+0x100000，可以看到修改后栈地址都是在 0x70000000-0x70000000+0x200000 这段地址范围内，并且剩余内存空间明显变大。

图 16-1: 运行内存修改前后比较图



技巧

用户可以参考这个步骤将运行内存区域从 DRAM 切换到 SRAM。切换运行内存区域到 SRAM，需要在 board.dts 中，需要在设备树中将 SRAM 内存区域 reserved 保留下来，并且在设备树的 remoteproc 节点中的 memory-region 属性中引用这区域 修改如 memory-region = <&rtos_reserved>

RTOS 的 `CONFIG_ARCH_START_ADDRESS` 以及 CONFIG_ARCH_MEM_LENGTH，与设备树预留运行内存区域一致。SRAM 区域和 DRAM 区域可以同时预留，但 CONFIG_ARCH_START_ADDRESS、CONFIG_ARCH_MEM_LENGTH 只能在 SRAM 和 DRAM 之间二选一。

16.2 共享内存区域

从核使用的共享内存区域目前主要包含如下几个：

- 用于 rpmsg 通信的 vdev0buffer、vdev0vring0 和 vdev0vring1 保留内存
- 用于 rpbuf 通信的保留内存
- 用于共享中断功能的 share_irq_table 保留内存

以 MR536 为例，其 Linux 环境的 dts 配置里有如下保留内存区域：

```
ry {

    e907_rpbuf_reserved: e907_rpbuf@42380000 {
        compatible = "shared-dma-pool";
        no-map;
        reg = <0x0 0x42380000 0x0 0x80000>;
    };

    rv_vdev0buffer: vdev0buffer@42400000 {
        compatible = "shared-dma-pool";
        reg = <0x0 0x42400000 0x0 0x40000>;
        no-map;
    };

    rv_vdev0vring0: vdev0vring0@42440000 {
        reg = <0x0 0x42440000 0x0 0x2000>;
        no-map;
```

```
    rv_vdev0vring1: vdev0vring1@42442000 {
        reg = <0x0 0x42442000 0x0 0x2000>;
        no-map;
    };

    e907_share_irq_table: share_irq_table@4244c000 {
        reg = <0x0 0x4244c000 0x0 0x1000>;
        no-map;
    };
};
```

可以看到 vdev0buffer@42400000、vdev0vring0@42440000、vdev0vring1@42442000 是用于 rpmsg 通信的保留内存；e907_rpbuf@42380000 是用于 rpbuf 通信的保留内存；share_irq_table@4244c000 是用于共享中断的保留内存。

16.3 修改内存区域的地址

上述从核使用的内存区域的地址都确保其他用途的保留内存冲突即可，其中从核的运行内存区域被修改时是需要同时更改 Linux 环境的 dts 配置和 RTOS 环境的配置的。

另外由于这些共享内存区域的数量比较多 (除了共享中断使用的内存区域不会被从核修改外其他都会被从核修改)，而从核的 MPU(RISC-V 架构里叫 PMP) 配置条目的数量可能比较少 (比如 MR527 的 RV 核只有 8 个配置条目)，因此建议尽量将这些会被从核修改的内存区域配置为相邻内存区域，也即不同内存区域中间不要有空洞 (前面内存区域的结束地址 +1 后等于后面内存区域的起始地址)，这样的话从核只需要使用一个 MPU 配置条目来配置这些共享内存区域的访问权限信息，防止从核由于 MPU 配置条目数量不够导致系统运行异常。

上面“共享内存区域”章节里的 e907_rpbuf@42380000、vdev0buffer@42400000、vdev0vring0@42440000 和 vdev0vring1@42442000 这 4 个内存区域就是相邻内存区域。下面以 MR536 为例，给出将用于 rpmsg 通信的共享内存和用于 rpbuf 通信的共享内存修改到 0x70000000 开始的连续内存区域的示例：

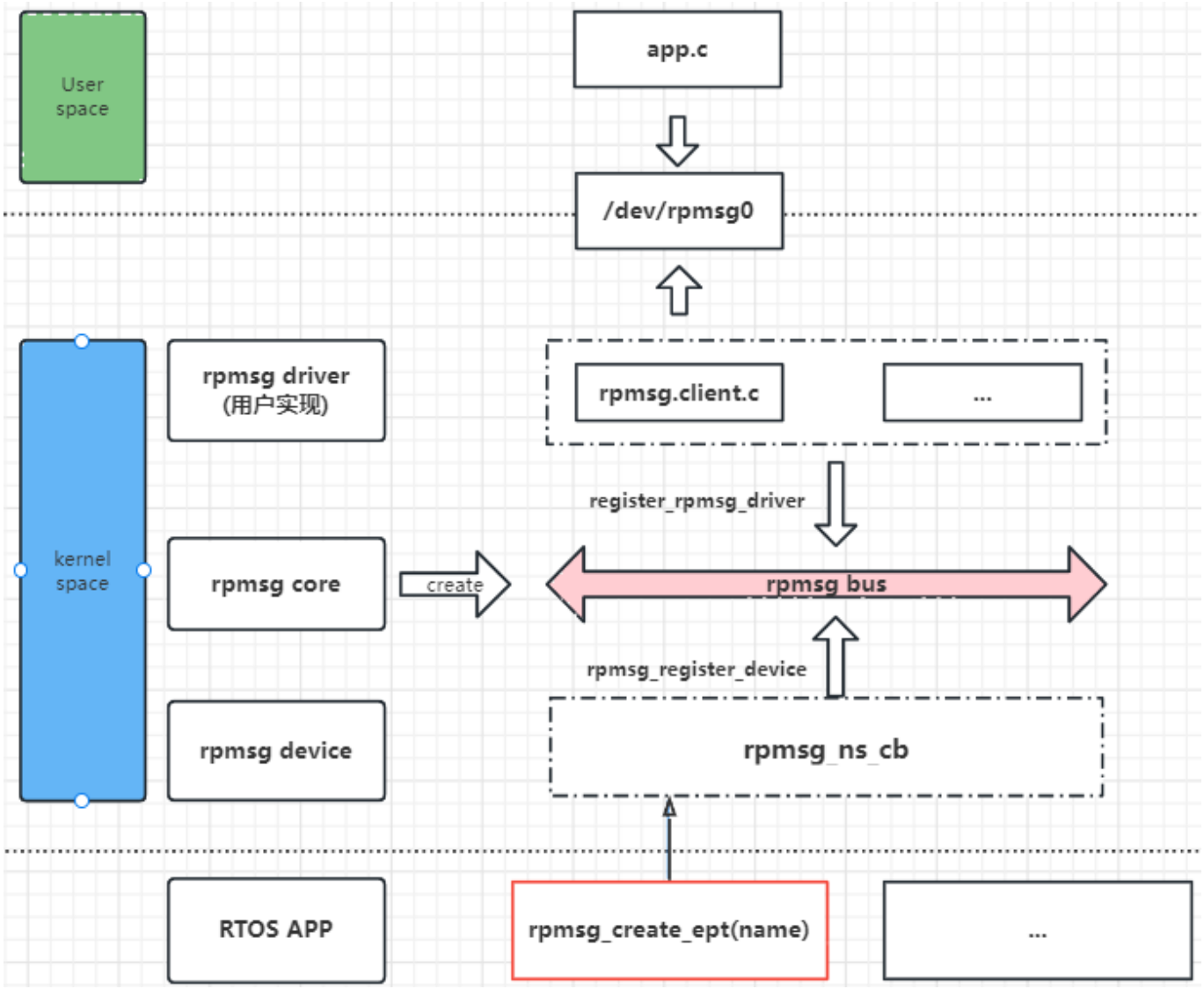
```
reserved-memory {
    e907_rpbuf_reserved: e907_rpbuf@70000000 {
        compatible = "shared-dma-pool";

        reg = <0x0 0x70000000 0x0 0x80000>;
    };

    rv_vdev0buffer: vdev0buffer@70080000 {
        compatible = "shared-dma-pool";
        reg = <0x0 0x70080000 0x0 0x40000>;
        no-map;
    };

    rv_vdev0vring0: vdev0vring0@700c0000 {
        reg = <0x0 0x700c0000 0x0 0x2000>;
        no-map;
    };
    rv_vdev0vring1: vdev0vring1@700c2000 {
        reg = <0x0 0x700c2000 0x0 0x2000>;
        no-map;
    };

    e907_share_irq_table: share_irq_table@4244c000 {
        reg = <0x0 0x4244c000 0x0 0x1000>;
        no-map;
    };
};
```

第十七章 核间通信 rpmsg

全志平台基于 OpenAMP RPmsg 框架以及 Linux 内核的 RPMsg 框架，实现了两套框架（Rpmsg V1/Rpmsg V2），方便用户在 AMP 场景下进行核间通信。Rpmsg V1 与 V2 向用户提供了兼容的 API 接口。

17.1 Rpmsg V1

图 17-1: Rpmsg 整体框图

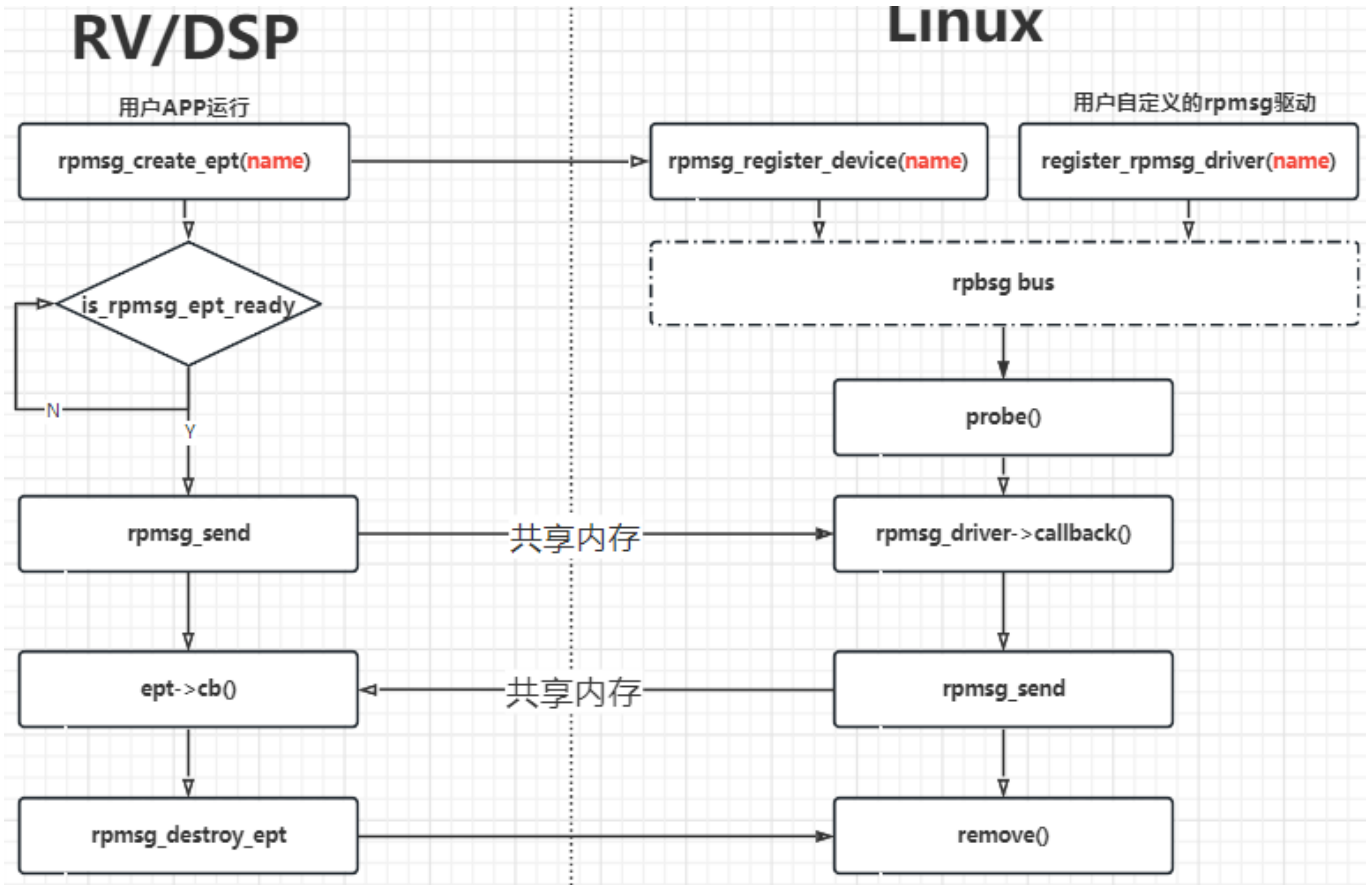


其通信链路建立流程如下：

1. RTOS 端调用 rpmsg_create_ept 创建指定 name 的端点。
2. Linux 端 rpmsg core 层收到端点创建消息，调用 rpmsg_register_device 将其作为一个设备注册到 rpmsg bus。
3. Linux 端 rpmsg bus 匹配到相应的驱动，触发其 probe 函数。
4. Linux 端驱动 probe 函数完成一些资源的分配以及文件节点的生成。
5. Linux 端驱动的 probe 函数调用完后，rpmsg bus 会回复一个 ACK。
6. RTOS 端收到 ACK 后设置端点的状态，此时使用 is_rpmsg_ept_ready 函数会返回 true。

一个简易的 rpmsg 驱动通信流程如下：

图 17-2: Rpmsg 通信流程



端点除了有专用端点（如 ctrl 端点、heartbeat 端点，命名服务端点），还包含有由 rpmsg_client.c 统一创建的端点，这部分端点是面向用户的端点，这部分端点会向用户空间呈现 /dev/rpmsgX 文件操作接口，在跨核进行自定的数据交互时，应该通过用户端点来进行。用户可以操作 /dev/rpmsgX，读取文件可以获取远端发送过来的数据，也可以写入 /dev/rpmsgX 来向远端发送数据。接下来，通过用户端点的创建流程，来理解通信链路建立流程：

1. 【Linxu 端】用户空间通过操作 ctrl 端点提供的用户空间文件（例如：/dev/rpmsg_ctrl-e907_rproc@1a00000），对这个文件写入 RPMSG_CREATE_EPT_IOCTL 指令进行创建节点
2. 【Linux 端】rpmsg_master.c 响应 RPMSG_CREATE_EPT_IOCTL 指令，然后通过 ctrl 端点发送 RPMSG_CREATE_CLIENT 命令到 RV/DSP，让 RV/DSP 先创建端点。RV/DSP 创建端点时，会通过命名服务间接触发 Linux 端也创建配对的端点
3. 【RV/DSP】rpmsg_master.c 的接收数据线程响应 RPMSG_CREATE_CLIENT 命令，创建 client 端点（该端点用于和 Linux 端的 client 端点进行点对点通信），所有的 client 端点的 name 都是统一的（例如：sunxi,rpmsg_client）。在创建端点时，会统一将端点的 src 和 dts 预设置为 RPMSG_ADDR_ANY，然后在本地分配 src，dst 由 Linux 端分配（dst 就是 Linux 端的端点的 src）

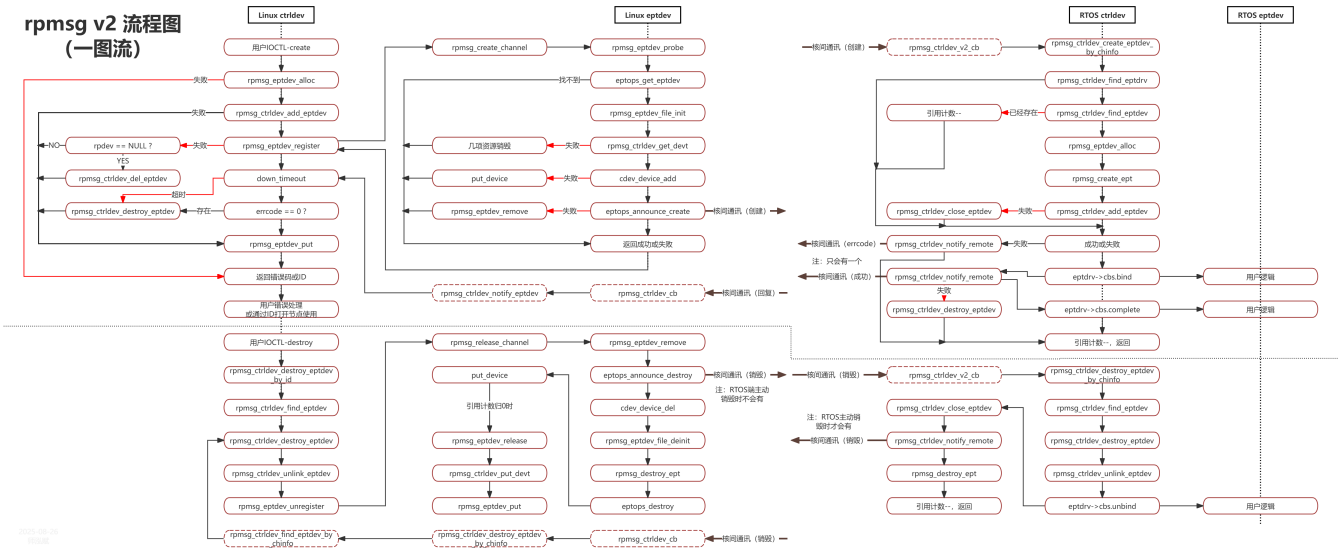
通过命名服务端点，发生命名消息，发送给 Linux 端

5. 【Linux 端】在 rpmsg_ns.c/rpmsg_ns_cb 回调函数中接收到命名消息，调用接口 rpmsg_create_channel 创建通道并最终会调用到 rpmsg_core.c/rpmsg_register_device 注册一个 rpmsg_device（一个 rpmsg_device 对应一个 rpmsg 通信设备），多个 client 端点的 rpmsg_device 的设备名是统一的（例如：sunxi,rpmsg_client）
6. 【Linux 端】rpmsg_bus 总线匹配机制，调用总线的 probe 接口（rpmsg_dev_probe），在总线的 probe 中创建端点（该端点和从核创建的端点时一对通信的端点）。创建的端点接收到远端的 RPMSG_CREATE_CLIENT 消息，则添加 /dev/rpmsX 设备，并且为每个端点的替换正式的接收数据回调函数
7. 【Linux 端】总线的 probe 函数，调用对应驱动的 probe 函数（rpmsg_client.c），这个函数中初始化端点对应的设备文件（/dev/rpmsgX）以及其文件操作合集 fops
8. 【linux 端】用户空间的 RPMSG_CREATE_EPT_IOCTL 至此完成，RV/DSP 和 Linux 端都创建了一个端点，这两个端点可以配对进行数据交互
9. 【linux 端】用户空间对 /dev/rpmsgX 进行 read/write，即可完成对端点的读写操作。写入 /dev/rpmsgX 的数据会发送给 RV/DSP，RV/DSP 发送过来的数据，会缓存在内核空间，用户可以将其读出

sg V2

Rpmsg V1 创建销毁流程复杂，因此设计简化版本。Rpmsg V2 框架重新设计了端点的创建与销毁机制，通过 ctrl 端点直接在 Master 与 Remote 之间交换控制消息（NS_CREATE/NS_DESTROY），省去了 V1 中经由命名服务的间接流程，使得端点生命周期管理更加简洁可控。V2 同时支持动态创建多个同名端点实例，并通过 Kconfig 兼容层实现 V1/V2 API 共存与切换。

图 17-3: Rpmsg V2 整体框图



V2 框架中，Linux 端与 RTOS 端各有一个 ctrl 端点（sunxi,rpmsg_ctrl_v2），负责端点的创建和销毁管理。Linux 端通过 /dev/rpmsg_ctrl-* 字提供 ioctl 接口，RTOS 端通过 rpmsg_client_v2_bind/unbind 等 API 管理端点驱动。两端通过 ctrl 端点互v_msg 消息来协同完成端点的创建与销毁。其端点创建流程如下：

1. 【Linux 端】用户空间通过操作 ctrl 端点文件（例如：/dev/rpmsg_ctrl-e907_rproc@1a00000），调用 ioctl RPMSG_CREATE_EPT_IOCTL 发起创建端点请求，传入端点名称 name
2. 【Linux 端】rpmsg_ctrldev.c 响应 ioctl，分配 eptdev 并通过 idr_alloc 分配端点 ID（范围 512~1023），然后调用 rpmsg_create_channel 创建 rpmsg channel，触发 RTOS 端命名服务
3. 【Linux 端】创建 rpmsg channel 后，Linux 端 rpmsg bus 匹配到 sunxi,rpmsg_ept_v2 驱动，触发 eptdev probe，此时 RTOS 端尚未创建配对端点
4. 【Linux 端】eptdev probe 完成后，Linux 端通过 ctrl 端点向 RTOS 发送 NS_CREATE 消息，携带端点名称、src/dst 地址信息
5. 【RTOS 端】rpmsg_ctrldev_v2_cb 接收 NS_CREATE 消息，查找已注册的 eptdrv（通过 rpmsg_client_v2_bind 注册），检查实例数上限后分配 eptdev，通过 bitmap 分配地址，调用 rpmsg_create_ept 创建端点，依次调用 bind 和 complete 回调

RTOS 端通过 ctrl 端点回复 NS_CREATE 消息，携带分配好的地址信息

7. 【Linux 端】Linux 端收到回复后，up(&eptdev->notify) 唤醒等待，ioctl 返回端点 ID，/dev/rpmsgN 设备节点可被用户态 read/write 使用

端点销毁流程如下：

1. 【Linux 端】用户空间调用 ioctl RPMSG_DESTROY_EPT_IOCTL，传入端点 ID
2. 【Linux 端】通过 ctrl 端点向 RTOS 发送 NS_DESTROY 消息，RTOS 端 rpmsg_ctrldev_v2_cb 接收后查找并销毁对应 eptdev，调用 unbind 回调，释放地址和资源
3. 【Linux 端】销毁完成后，/dev/rpmsgN 设备节点被移除

AMP RPMsg

AMP 场景下，RTOS 或裸机系统中进行处理器之间进行相互通信，需要使用到 OpenAMP 的 RPMsg 框架，通过本节，用户可以了解 OpenAMP RPMsg 通信步骤及其接口。**注意**：用户编写 rpmsg 通信程序时，需要基于全志的 rpmsg 框架及接口来进行编写，具体请参考“rpmsg 应用层通信程序”。

OpenAMP RPMsg 通信的常见步骤如下：

```
#include <openamp/rpmsg.h>
#include <openamp/sunxi_helper/openamp_platform.h>

/* Endpoint callback, called when the endpoint receives data. */
static int rpmsg_ept_callback(struct rpmsg_endpoint *ept, void *data,
        size_t len, uint32_t src, void *priv)
{
    /* ... */
    return RPMSG_SUCCESS;


/* Endpoint seRVice unbind callback, called when remote endpoint is destroyed. */
static void rpmsg_ns_unbind_callback(struct rpmsg_endpoint *ept)
{
    /* ... */
}

int main(int argc, char *argv[])
{
    struct rpmsg_endpoint ept;
    struct rpmsg_device *rpmsg_dev;

    /* 1. Get rpmsg device */
    rpmsg_dev = openamp_platform_get_rpmsg_vdev(RPMSG_DEV_ID);

    /* 2. Create endpoint */
    rpmsg_create_ept(&ept, rpmsg_dev, EPT_NAME, EPT_SRC_ADDR, EPT_DST_ADDR,
pmsg_ept_callback,rpmsg_ns_unbind_callback);

    /* 3. Waiting for endpoint ready */
    while (!is_rpmsg_ept_ready(ept)) {
        /* sleep ... */
    }

    /* 4. Send data to remote */
    rpmsg_send(ept, data, data_len);

    /* 5. Destroy endpoint */
    rpmsg_destroy_ept(ept);

    return 0;
}
```

device，为后面创建 endpoint 做准备。

2. 创建 endpoint。应用程序通过操作 endpoint 与远端处理器进行通信，endpoint 有源地址和目标地址。在本地处理器，endpoint 通过源地址进行区分；在远端处理器 endpoint 通过目的地址进行区分。当某个 endpoint 收到远端处理器发过来的消息时，它的回调函数会被调用到。
3. 在第一次使用某个 endpoint 向远端发送消息前，需要等待它准备好。
4. 使用 `rpmsg_send` 向远端发送信息。
5. 当不再需要使用 endpoint 时，使用 `rpmsg_destroy_ept` 销毁它。

上述步骤中的 OpenAMP RPMsg 接口被全志封装成以下三个接口，具体详见于 `rtos-components/thirdparty/openamp/sunxi_helper/openamp.c`

```
/*打开端点，内部调用了OpenAMP RPMsg的rpmsg_create_ept接口*/
struct rpmsg_endpoint *openamp_ept_open(const char *name, int rpmsg_id,
                                uint32_t src_addr, uint32_t dst_addr,void *priv,
```

```
    rpmsg_ept_cb cb, rpmsg_ns_unbind_cb unbind_cb)

/*关闭端点，内部调用了OpenAMP RPMsg的rpmsg_destroy_ept接口*/
void openamp_ept_close(struct rpmsg_endpoint *ept)

/*发送消息到rpmsg端点，内部调用了OpenAMP RPMsg的rpmsg_send接口*/
int openamp_rpmsg_send(struct rpmsg_endpoint *ept, void *data, uint32_t len)
```

17.3.1 OpenAMP RPMsg 接口说明

17.3.1.1 获取 RPMsg Device

```
/* <openamp/sunxi_helper/openamp_platform.h> */

struct rpmsg_device *openamp_platform_get_rpmsg_vdev(int id);
```

这个是全志自己定义的 API，并非来自 OpenAMP 官方。全志平台的 OpenAMP 在初始化过程中会创建 RPMsg device，该 API 就是用来获取前面已经初始化好的 RPMsg device。

一般来说，一个 RPMsg device 对应一个可进行通信的远端处理器。

具体的 ID 号，在全志平台上其定义如下：

- 对于主核：0 代表 DSP 端，1 代表 RV 端；
- 对于任一从核：0 均代表 ARM 端，1 代表另一个从核。如，对于 RV 端，ID=1 表示远端的 DSP 处理器；对于 DSP 端，ID=1 表示远端的 RV 处理器。

17.3.1.2 创建 endpoint

```
pmsg.h> */


/**
 * rpmsg_create_ept - create rpmsg endpoint and register it to rpmsg device
 *
 * Create a RPMsg endpoint, initialize it with a name, source address,
 * remoteproc address, endpoint callback, and destroy endpoint callback,
 * and register it to the RPMsg device.
 *
 * @ept: pointer to rpmsg endpoint
 * @name: seRVice name associated to the endpoint
 * @src: local address of the endpoint
 * @dest: target address of the endpoint
 * @cb: endpoint callback
 * @ns_unbind_cb: end point service unbind callback, called when remote ept is
 *                destroyed.
 *
 * In essence, an rpmsg endpoint represents a listener on the rpmsg bus, as
 * it binds an rpmsg address with an rx callback handler.
 *
 * Rpmsg client should create an endpoint to discuss with remote. rpmsg client
 * provide at least a channel name, a callback for message notification and by
 * default endpoint source address should be set to RPMSG_ADDR_ANY.
 *
 * As an option Some rpmsg clients can specify an endpoint with a specific
 * source address.
 */

int rpmsg_create_ept(struct rpmsg_endpoint *ept, struct rpmsg_device *rdev,
             const char *name, uint32_t src, uint32_t dest,
             rpmsg_ept_cb cb, rpmsg_ns_unbind_cb ns_unbind_cb);
```

对于同一个 RPMsg device，可以创建多个地址不同的 endpoint，提供给不同模块使用。

cb 函数在远端处理器往该 endpoint 的地址发送消息时会被调用到；

- ns_unbind_cb 函数会在远端对应的 endpoint 销毁时被调用到（因为要实现通信，远端会有一个对应的 endpoint 与本地的 endpoint 关联起来），如果不需要，可设为 NULL。

endpoint 的地址有一些特殊值：

- **0x35**（十进制的 53）：预留出来用作 name service 消息包的收发。
- **≥ 0x400**（十进制的 1024）：大于或等于 0x400 的值会被用作自动分配的地址值。当用户创建 endpoint 时，如果将 src 或 dest 设为 RPMSG_ADDR_ANY，RPMsg 会自动为其分配空闲的地址，这个分配的值就会从 0x400 开始。
- **RPMSG_ADDR_ANY**（十六进制的 0xFFFFFFFF）：在自动分配地址和 name service 通知机制中使用（使用时最好用 RPMsg 提供的宏 RPMSG_ADDR_ANY，以免该值后续发生改变）。

name service 机制，用于通知远端进行 RPMsg device/channel/endpoint 的动态申请。

1. 当创建 endpoint 时，如果将 dest 设为 RPMSG_ADDR_ANY，会往远端发送 name service 消息包。
后调用函数应 RPMsg
3. 远端分配好地址后，发送消息回来。
4. 收到远端回应后，dest 就会变成远端对应 endpoint 的地址，此时就可开始通过本地的 endpoint 与远端的 endpoint 进行通信。

出于代码通用性的考虑，一般来说都会推荐将地址设为 RPMSG_ADDR_ANY，让其自动分配。

17.3.1.3 等待 endpoint 准备好

```
/* <openamp/rpmsg.h> */

/**
 * is_rpmsg_ept_ready - check if the rpmsg endpoint ready to send
 *
 * @ept: pointer to rpmsg endpoint
 *
 * Returns 1 if the rpmsg endpoint has both local addr and destination
 * addr set, 0 otherwise
 */
static inline unsigned int is_rpmsg_ept_ready(struct rpmsg_endpoint *ept)
{
    return ept && ept->rdev && ept->dest_addr != RPMSG_ADDR_ANY;
}
```

在每次数据发送前，都建议使用该接口检查 endpoint 是否就绪。（当使用 name service 机制时，等待实际就是远端是否已经进行了回应，是否已经修改了 endpoint 的 dest_addr）。

17.3.1.4 向远端发送数据

向远端发送数据可使用以下 API（具体的描述可参考头文件 openamp/include/openamp/rpmsg.h ）：

```
pmsg.h> */


static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
                 int len);
static inline int rpmsg_sendto(struct rpmsg_endpoint *ept, const void *data,
                   int len, uint32_t dst);
static inline int rpmsg_send_offchannel(struct rpmsg_endpoint *ept,
                    uint32_t src, uint32_t dst,
                    const void *data, int len);

static inline int rpmsg_trysend(struct rpmsg_endpoint *ept, const void *data,
                int len);
static inline int rpmsg_trysendto(struct rpmsg_endpoint *ept, const void *data,
                  int len, uint32_t dst);
static inline int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept,
                       uint32_t src, uint32_t dst,
```

```
        *data, int len);const void
```

- send 和 trysend 类别函数的区别主要是：当底层用来传输数据的 buffer 都满了时，send 类别的会阻塞住，直到 buffer 可用或 15 秒超时返回；trysend 则是立即返回。
- send、sendto、send_offchannel 这三类函数的区别是：send 函数直接使用 ept 的 src 和 dst；sendto 函数使用 ept 的 src，但 dst 是使用额外传进来的参数；send_offchannel 函数则是 src 和 dst 都与 ept 无关，都是使用额外传进来参数。

由此可见，往另一端发送数据时，其实并不一定要用 endpoint 里的地址。如果之前创建有别的 endpoint，理论上已经可以往远端的任意地址发送数据。

一般为了方便管理，更为推荐使用创建对应地址的 endpoint 来传输数据。

说明

的内部代码实现来看，单次 rpmsg_send 发送的数据量无法超过底层 RPMsg 单个消息包 buffer 的大小，否则会被截断。当前 buffer 的大小固定是 512 bytes，扣除 16 bytes 的头部信息，用户单次可传输的数据量为 496 bytes。

17.3.1.5 销毁 endpoint

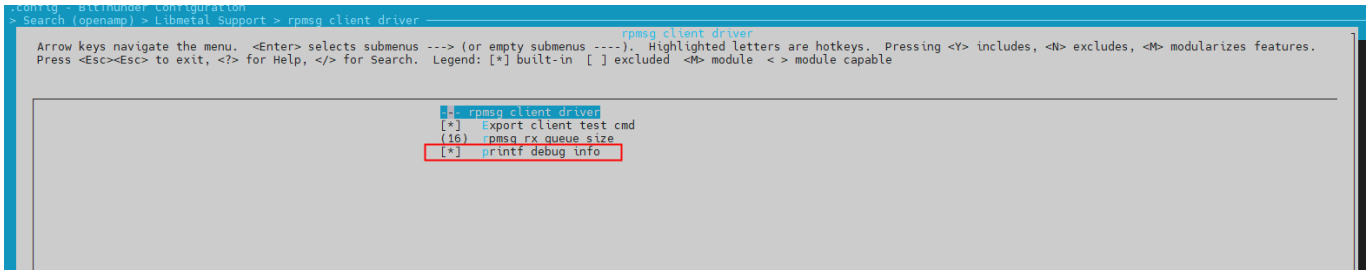
```
/* <openamp/rpmsg.h> */

/**
 * rpmsg_destroy_ept - destroy rpmsg endpoint and unregister it from rpmsg
 *                     device
 *
 * @ept: pointer to the rpmsg endpoint
 *
 * It unregisters the rpmsg endpoint from the rpmsg device and calls the
 * destroy endpoint callback if it is provided.
 */
stindy*_eptt(struct
```

17.4 通信测试

17.4.1 rpmsg 通信测试

为了便于后面的调试，RTOS 环境下 rpmsg 需要选上调试配置，如下：

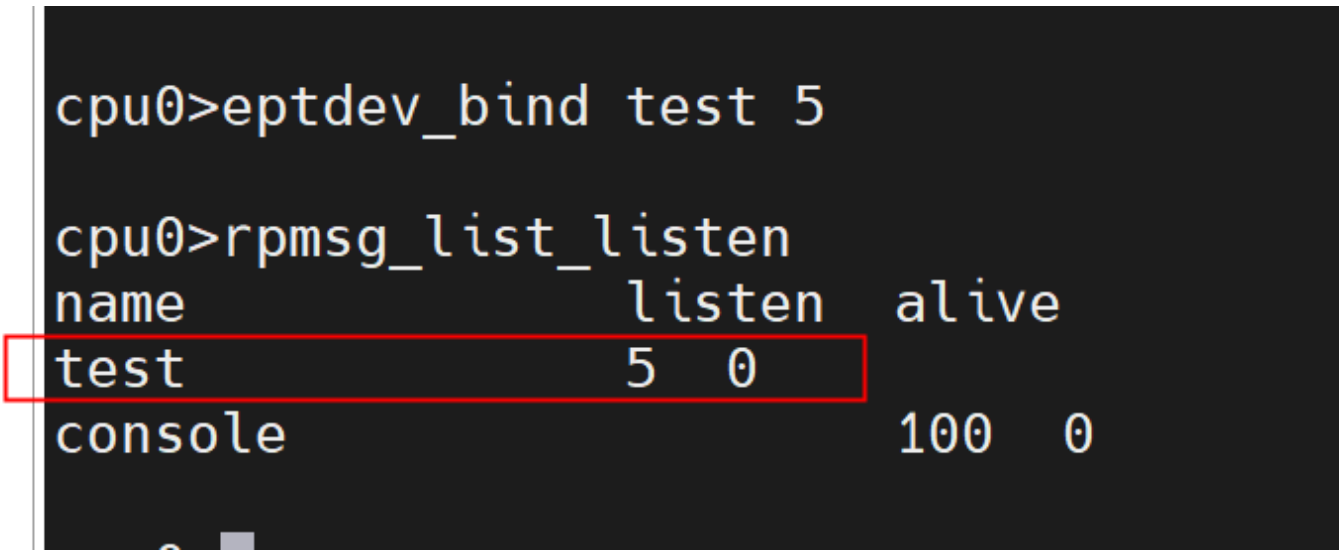
图 17-4: rpmsg 调试配置



17.4.1.1 名字监听

RTOS 控制台输入 eptdev_bind 命令进行监听。

图 17-5: rpmsg test



eptdev_bind 命令说明：

- 参数一：name，表示需要监听的端点名称。
- 参数二：cnt，该名称的端点最多可以创建多少个链接。

rpmsg_list_listen 命令无参数，该命令会列举出所有正在监听的端点的设备名字，数量等。

17.4.1.2 节点创建

msg_demo 命令进行节点创建。

图 17-6: rpmsg test

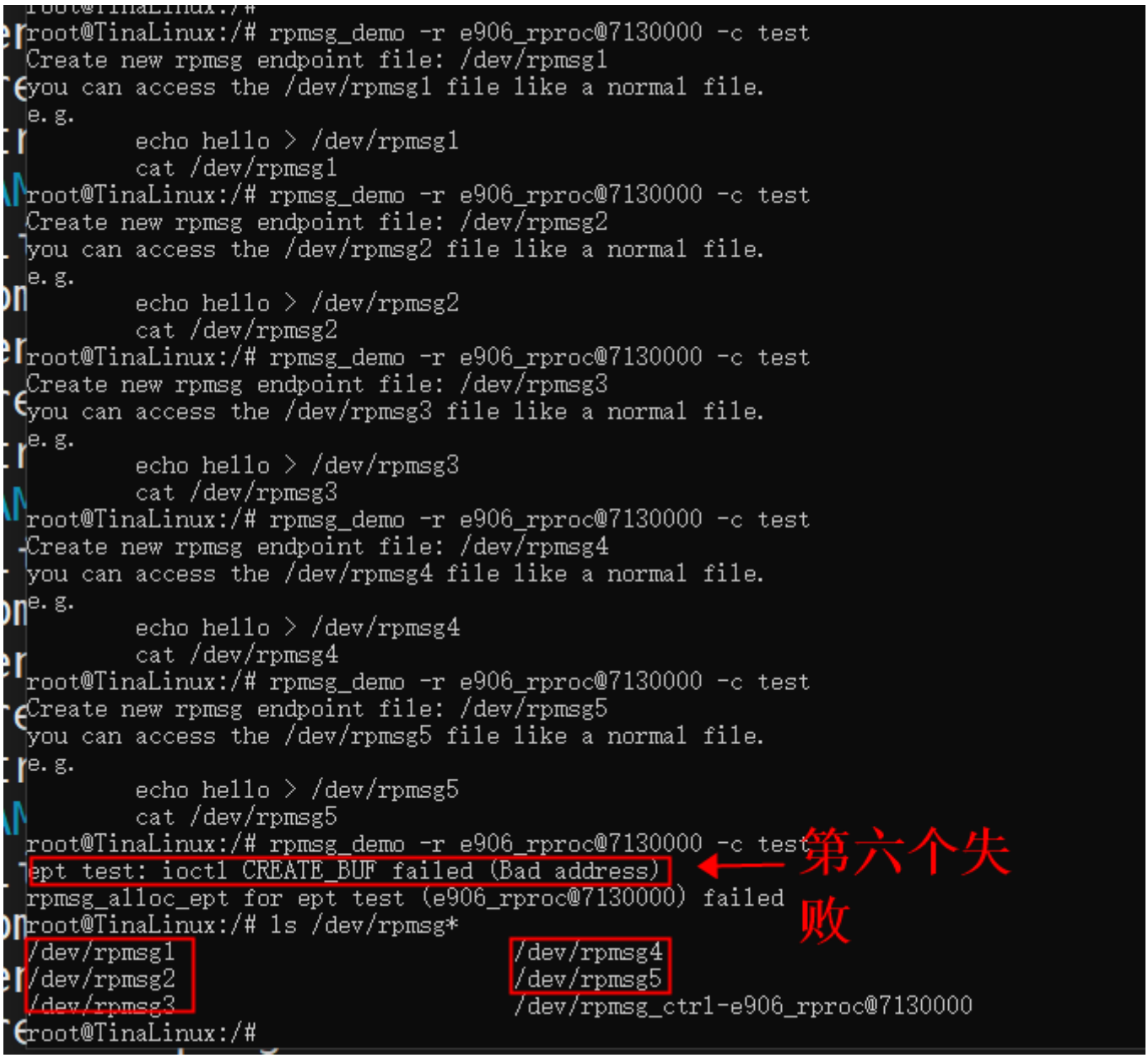


图 17-7: rpmsg test

```
cpu0>ctrldev: Rx 44 Bytes
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_client ept ready
client: Rx 8 Bytes
rpmsg1: binding
send 0x13131411 to rpmsg1
create rpmsg1 client success
ctrldev: Rx 44 Bytes
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_client ept ready
client: Rx 8 Bytes
rpmsg2: binding
send 0x13131411 to rpmsg2
create rpmsg2 client success
ctrldev: Rx 44 Bytes
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_client ept ready
client: Rx 8 Bytes
rpmsg3: binding
send 0x13131411 to rpmsg3
create rpmsg3 client success
ctrldev: Rx 44 Bytes
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_client ept ready
client: Rx 8 Bytes
rpmsg4: binding
send 0x13131411 to rpmsg4
create rpmsg4 client success
ctrldev: Rx 44 Bytes
[AMP_INFO][openamp_ept_open:119]Waiting for sunxi,rpmsg_client ept ready
client: Rx 8 Bytes
rpmsg5: binding
send 0x13131411 to rpmsg5
create rpmsg5 client success
ctrldev: Rx 44 Bytes
listen is full.

send 0x13131416 to rpmsg6
```

根据 log 可以看出，创建了一个 rpmsg1-5 5 个设备，因为 RV 只监听了 5 个，故最多只能创建 5 个。

rpmsg_demo 参数说明：

- -r name：指定远端处理器名字，具体名字由 dts 中的 rproc 节点决定；如在 MR527 平台上，RV 在 dts 的节点名为 e906_rproc@7130000。
- -c name：需要创建的端点的名字，rpmsg 通过名字进行匹配。
- -d name：需要删除的端点的名字。

17.4.1.3 节点通信

标准的文件操作 V 发送数据即可。

图 17-8: rpmsg test

```
root@TinaLinux:/#
root@TinaLinux:/# echo hello,rpmsg1.I am linux. > dev/rpmsg1
root@TinaLinux:/# echo hello,rpmsg2.I am linux. > dev/rpmsg2
root@TinaLinux:/# echo hello,rpmsg3.I am linux. > dev/rpmsg3
root@TinaLinux:/# echo hello,rpmsg4.I am linux. > dev/rpmsg4
root@TinaLinux:/# echo hello,rpmsg5.I am linux. > dev/rpmsg5
root@TinaLinux:/#
```

图 17-9: rpmsg test

```
cpu0>rpmsg1: Rx 25 Bytes
Data:hello,rpmsg1.I am linux.

rpmsg2: Rx 25 Bytes
Data:hello,rpmsg2.I am linux.

rpmsg3: Rx 25 Bytes
Data:hello,rpmsg3.I am linux.

rpmsg4: Rx 25 Bytes
Data:hello,rpmsg4.I am linux.

rpmsg5: Rx 25 Bytes
Data:hello,rpmsg5.I am linux.
```

RV 向 Linux 发送数据：

图 17-10: rpmsg test

```
cpu0>rpmsg_list_epts
name              major           src - dst
rpmsg5            0               0x406 - 0x406
rpmsg4            0               0x405 - 0x405
rpmsg3            0               0x404 - 0x404
rpmsg2            0               0x403 - 0x403
rpmsg1            0               0x402 - 0x402

cpu0>eptdev_send 3 hello,linux
will send hello,linux to rpmsg3

cpu0>
```

图 17-11: rpmsg test

```
root@TinaLinux:/#
root@TinaLinux:/# cat dev/rpmsg3
hello,linux
```

rpmsg_list_epts 命令说明：

- 无参数，列举出当前可用的端点的名称，地址信息等

eptdev_send 命令说明：

- 参数一：id，指定使用哪一个 rpmsgX 端点发送消息，这里的 X 就是参数一指定的 id，可通过 rpmsg_list_epts 查看当前可用的端点。
- 参数二：data，要发送的数据

17.4.1.4 节点关闭

Linux 主动释放节点，比如主动删除 rpmsg2 节点：

图 17-12: rpmsg test

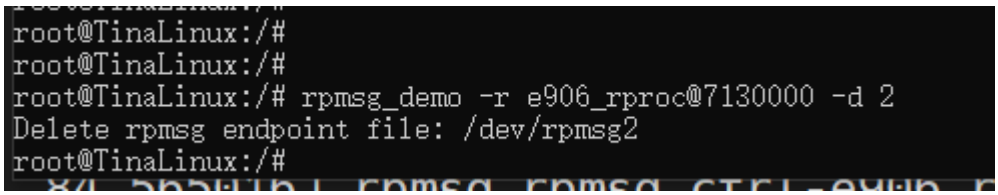


图 17-13: rpmsg test

RV 主动关闭设备：

图 17-14: rpmsg test

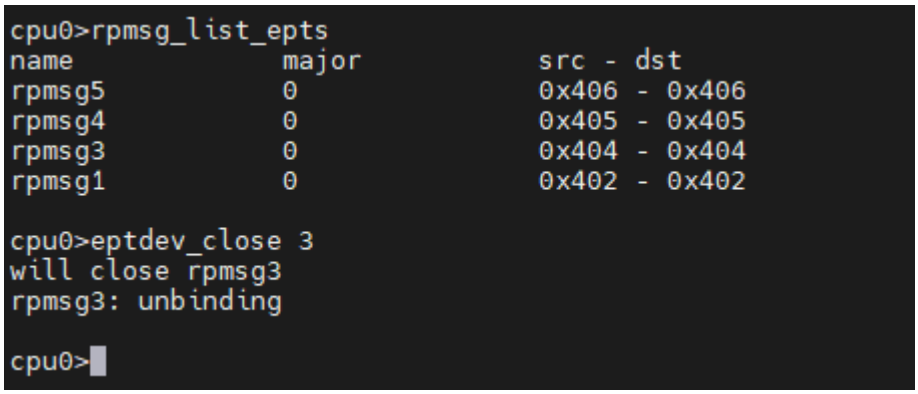


图 17-15: rpmsg test

则会释放所有的链接：

图 17-16: rpmsg test

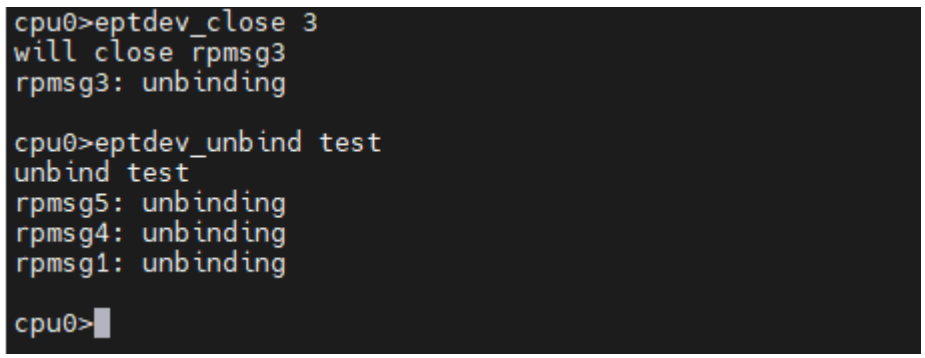
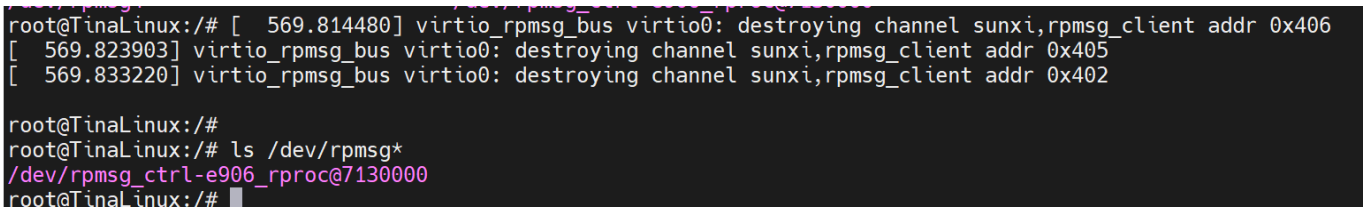


图 17-17: rpmsg test



eptdev_close 命令说明：

- 参数一：id，指定删除哪一个 rpmsgX 端点，这里的 X 就是参数一指定的 id，可通过 rpmsg_list_epts 查看当前可用的端点。

eptdev_unbind 命令说明：

- 第一个参数：name，表示需要取消监听的端点名称；取消监听时，会把所有通过该 name 创建的端点进行释放。

17.4.2 rpmsg 应用层通信程序

17.4.2.1 rpmsg_test

此程序适用于 RV 和 Linux 应用层的通信，**用户可参考该软件包开发自己的通信应用。**

RV 程序：rpmsg_test，源码路径：rtos-components/thirdparty/openamp/rpmsg_demo/rpmsg_ctrl/test

Linux 配合使用的程序：rpmsg_test，源码路径：platform/allwinner/system/rpmsg/test

该软件包功能如下：

自动生成生成随机数据和 MD5 值，发送给另一端

2. 对收到的数据进行 MD5 校验，并与收到的 MD5 进行比较，不相同则打印数据校验失败错误

17.4.2.2 编译

RV 编译时需要在 mrtos_menuconfig 中选上：

```
System components  --->
    thirdparty components --->
        OpenAMP Support  --->
            [*] rpmsg client driver  --->
                [*]   Export client test cmd
```

Linux 编译时需要在 m menuconfig 中选上：

```
Allwinner  --->

        -*- librpmsg..................................................... rpmsg library
        <*> rpmsg_test.................................................. rpmsg test demo
```

17.4.2.3 基本使用

对于 RV 而言，rpmsg_test 的参数说明如下：

- -h：打印帮助信息。
- -c：创建端点。
- -d：删除端点。
- -v：冗余打印，否则只有出错时才打印。
- -M：最大端点创建数量，默认是 10。

端点名称。

- -t tx_delay：指定每次的发送间隔，默认是 500 ms。
- -L tx_len：指定每次发送的数据长度，默认是 32 bytes，注意这个值不能超过 496，一次 Rpmsg 传输最多能传输 496 个字节。

例子：

- rpmsg_test -N “test” -L 256 -t 10 -c：监听端点，建立连接后将会创建一个发送线程，每隔 10ms 发送一次长度为 256 字节的消息
- rpmsg_test -N “test” -d：取消监听

对于 Linux 而言，rpmsg_test 的参数说明如下：

- -r name：指定远端处理器名字。
- -c name：需要创建的端点的名字，rpmsg 通过名字进行匹配。
- -v：冗余打印，否则只有出错时才打印。

指定每次的发送间隔，默认是

指定每次的发送间隔，默认是 [illegible] 能传输

- -R：接收警告阈值，当接收耗时超过该阈值后，会打印警告信息
- -T：发送警告阈值，当发送耗时超过该阈值后，会打印警告信息

例子：

- rpmsg_test -r e906_rproc@7130000 -c test -L 256 -t 10：创建端点，该端点会创建一个发送线程，每隔 10ms 发送一次长度为 256 字节的消息

17.4.2.4 简单例子

先确保 Linux 有以下 RV rpmsg 的控制节点，如果没有，请检查 RV 是否有正常启动。

```
root@TinaLinux:/# ls -l dev/rpmsg*
/dev/rpmsg_ctrl-e906_rproc@7130000
```

以下是一个使用 rpmsg_test 的简单例子：

```
Linux与RV通信：

1、RV输入 rpmsg_test -N test -L 496 -c -t 10 -M 10

2、Linux输入 rpmsg_test -r e906_rproc@7130000 -c test -L 496 -t 5 &
```

17.5 rpmsg 传输性能测试

用户可以使用 rpmsg_test 命令对 rpmsg 进行性能测试，测试发送方向和接收方向各自的耗时以及速率。本章节主要描述这个测试的操作步骤并解读测试结果。

层测试方法

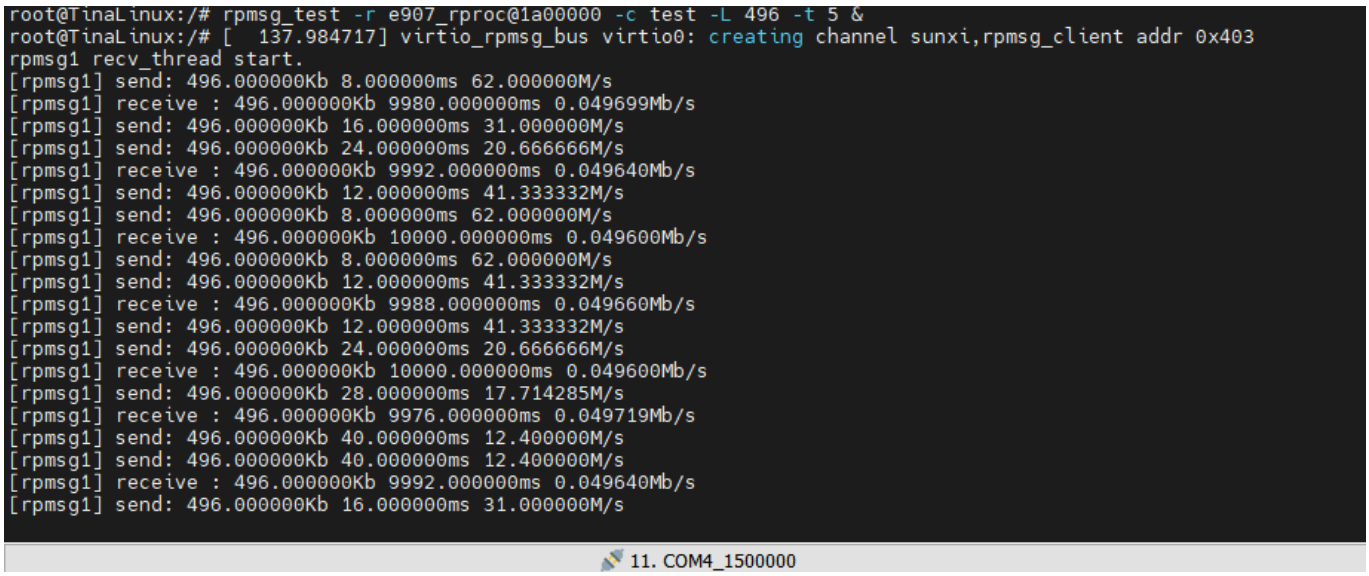
以 MR536 为例，下面的指令是 ARM 主核与 RV 从核之间的传输性能测试指令，指令的参数请查看“rpmsg 应用层通信程序”章节。

测试指令汇总如下，请根据平台，调整远端处理器名字

```
Linux与RV通信：

1、首先，RV输入 rpmsg_test -N test -L 496 -c -t 10 -M 10

2、然后，Linux输入 rpmsg_test -r e907_rproc@1a00000 -c test -L 496 -t 5 &
```

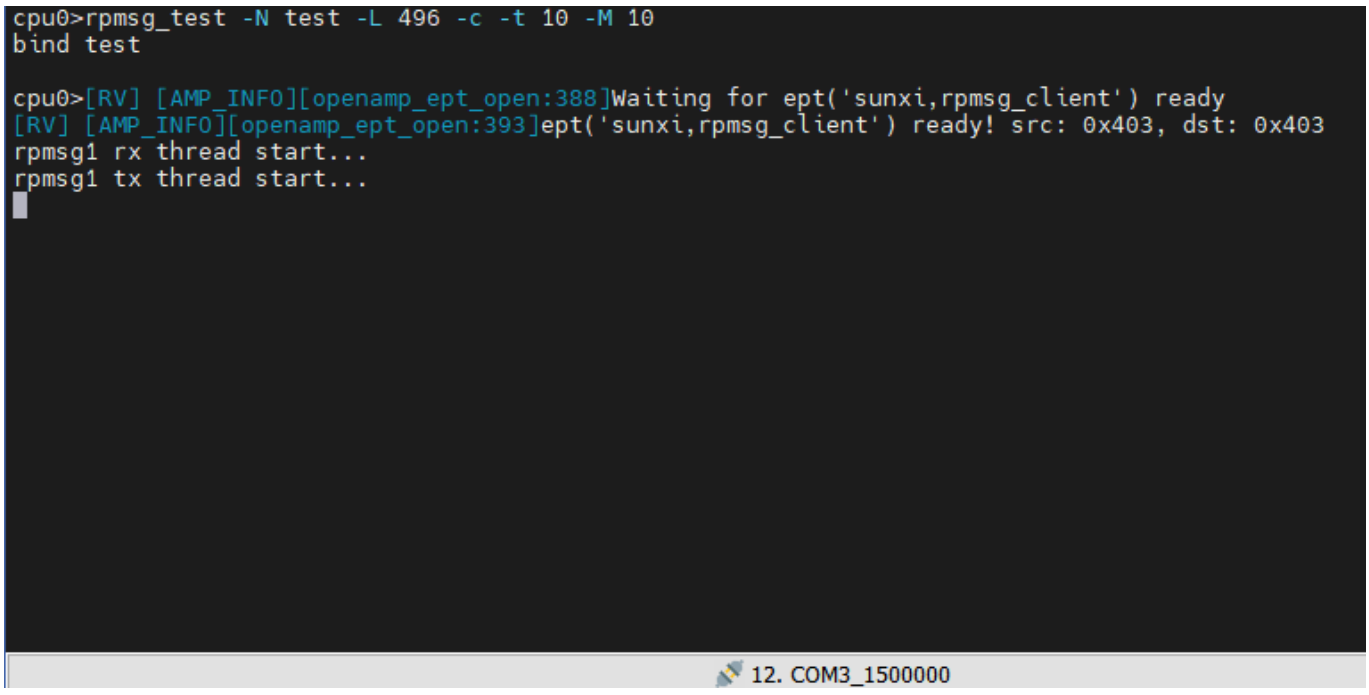
主核测试结果：

图 17-18: 主核传输性能测试结果



从核测试结果：

图 17-19: 从核传输性能测试结果



如图，主核与从核之间是通过 src 和 dst 均为 0x403 的端点进行通信，主核的这个端点的接收和发送被映射成对/dev/rpmsg1 设备文件文件的读操作和写操作。结果解读如下：

输出结果	结果解读
[rpmsg1] send: 496.000000Kb 8.000000ms 62.000000M/s	发送 496KB 数据，耗时 8 毫秒，发送速率为 64MB/s
[rpmsg1] receive : 496.000000Kb 9980.000000ms 0.049699Mb/s	接收 496KB 数据，耗时 9980 毫秒，接收速率为 0.049699MB/s

g 通信耗时测量

rpmsg 框架目前支持通信耗时测量功能，又叫性能跟踪，可通过此功能测量单次 rpmsg 通信过程中发送方和接收方各个阶段的耗时信息。

17.6.1 工作原理

通信双方记录 rpmsg 发送接收流程中某些执行点的时间戳，且通信双方获取到的时间戳具有相同的时间基准。

17.6.2 性能跟踪点和性能跟踪阶段

目前支持如下几个性能跟踪点：

- RPMSG_PERF_POINT_SENDER_FILL：发送方开始填充数据
- RPMSG_PERF_POINT_SENDER_NOTIFY：发送方开始通知对端
- RPMSG_PERF_POINT_SENDER_END：发送方发送结束
- F_P 接收方准备接收 _PREPARE
- RPMSG_PERF_POINT_RECEIVER_EXEC_CB：接收方开始执行回调函数
- RPMSG_PERF_POINT_RECEIVER_END：接收方接收处理结束

其中还有一个隐式的性能跟踪点，为发送方开始发送的时刻，提供给用户的 API 中没有此性能跟踪点，因为用户 API 获取到的数据是经过处理后的更方便查看的数据 (相对时间戳)，故此隐式跟踪点的相对时间戳为 0us。

目前支持如下几个性能跟踪阶段：

- RPMSG_PERF_STAGE_SENDER_PREPARE：发送方准备
- RPMSG_PERF_STAGE_SENDER_FILL：发送方填充数据
- RPMSG_PERF_STAGE_SENDER_NOTIFY：发送方通知对端
- RPMSG_PERF_STAGE_RECEIVER_RECV：发送方通知到接收方准备接收
- RPMSG_PERF_STAGE_RECEIVER_PREPARE：接收方准备
- F_ST 接收方执行回调函数 EC_CB

上述这些性能跟踪点和性能跟踪阶段都有对应的宏，Linux 环境可在 bsp/include/linux/aw_rpmsg.h 文件里找到，RTOS 环境可在 rtos/lichee/rtos-components/thirdparty/openamp/include/openamp/rpmsg.h 文件里找到。

17.6.3 配置信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPMSG_VIRTIO、CONFIG_AW_RPMSG_PERF_TRACE，禁用内核配置 CONFIG_RPMSG_VIRTIO。

从核 RTOS 环境需要启用配置 CONFIG_AW_RPMSG_PERF_TRACE。

由于通信耗时测量功能基于 AMP 时间戳模块，故 AMP 时间戳模块也需要配置，另外 Linux 和 RTOS 环境还需分别配置 CONFIG_RPMSG_AMP_TS_DEV_ID，且 Linux 环境下此配置的值需要和 dts 里 AMP 时间戳设备的 ID 一致，RTOS 环境下暂时只支持设置为 0，AMP 时间戳设备的 dts 配置也可参考 “主核和从核获取相同时基的时间戳”章节。

17.6.4 API 说明

rpmsg 通信涉及到通信双方，涉及到主核 Linux 和从核 RTOS，因此主核 Linux 环境和从核 RTOS 环境都有类似 API。另外由于主核 Linux 环境同时支持在内核态和用户态使用 rpmsg，故其 API 也区分内核态和用户态。目前 SDK 中 Linux 内核态 API 和 RTOS 环境 API 一致。

17.6.4.1 Linux 内核态和 RTOS 环境 API

17.6.4.1.1 rpmsg_perf_data_t 数据结构

rpmsg_perf_data_t 表示某次通信过程中的性能跟踪数据：

```
PERF_POINT_SENDER_FILL0
  #define RPMSG_PERF_POINT_SENDER_NOTIFY 1
  #define RPMSG_PERF_POINT_SENDER_END 2
  #define RPMSG_PERF_POINT_RECEIVER_PREPARE 3
  #define RPMSG_PERF_POINT_RECEIVER_EXEC_CB 4
  #define RPMSG_PERF_POINT_RECEIVER_END 5

  #define RPMSG_PERF_STAGE_SENDER_PREPARE 0
  #define RPMSG_PERF_STAGE_SENDER_FILL 1
  #define RPMSG_PERF_STAGE_SENDER_NOTIFY 2
  #define RPMSG_PERF_STAGE_RECEIVER_RECV 3 /* sender notify to receiver prepare */
  #define RPMSG_PERF_STAGE_RECEIVER_PREPARE 4
  #define RPMSG_PERF_STAGE_RECEIVER_EXEC_CB 5

  #define RPMSG_PERF_STAGE_NUM (RPMSG_PERF_STAGE_RECEIVER_EXEC_CB + 1)
  #define RPMSG_PERF_POINT_NUM RPMSG_PERF_STAGE_NUM

  typedef struct rpmsg_perf_data {
imestamp[RPMSG_PERF_POINT_NUM];
      uint32_t time_span[RPMSG_PERF_STAGE_NUM];
      uint64_t raw_timestamp;
  } rpmsg_perf_data_t;
```

成员说明：

- timestamp：性能跟踪点的相对时间戳数组
- time_span：性能跟踪阶段的耗时信息数组
- raw_timestamp：隐式性能跟踪点的原始时间戳 (绝对时间戳)

其中时间戳信息和耗时信息的单位都是 us。

获取到性能跟踪数据后，可使用 RPMSG_PERF_POINT_XXX 宏以及 RPMSG_PERF_STAGE_XXX 宏来拿到具体性能跟踪点的相对时间戳或性能跟踪阶段的耗时。

17.6.4.1.2 获取性能跟踪数据

```
int rpmsg_get_perf_data(struct rpmsg_endpoint *ept, rpmsg_perf_data_t *perf_data);
```

参数：

- ept：指向 rpmsg 端点的指针
- perf_data：指向获取到的性能跟踪数据的指针

返回值：

- 0：获取成功
- 其他值：获取失败

印性能跟踪数据

```
void rpmsg_dump_perf_data(const rpmsg_perf_data_t *perf_data);
```

参数：

- perf_data：指向性能跟踪数据的指针

录接收方处理结束的时间戳

```
void rpmsg_record_receiver_end_ts(struct rpmsg_endpoint *ept);
```

参数：

- ept：指向 rpmsg 端点的指针

由于在 Linux 内核态或 RTOS 环境下是在 rpmsg 回调函数中接收 rpmsg 数据的，属于异步方式，因此接收方处理结束的时间戳必须在 rpmsg 回调函数内记录，然后直接获取性能跟踪数据，此 API 就是提供给用户在 rpmsg 回调函数的最后面调用的。

17.6.4.2 Linux 用户态 API

用户态 API 和内核态 API 类似，但由于用户态是通过主动读取文件方式来读取内核态驱动接收到的 rpmsg 数据，而不是直接在 rpmsg 回调函数里读取 rpmsg 数据，因此无法在 rpmsg 回调函数里获取到 rpmsg 性能跟踪数据，故在设计时将 rpmsg 性能跟踪数据放在了 rpmsg 数据的前
过读取 rpmsg 端点设备文件来获取性能跟踪数据，且只有设置了内核态向用户态传递性能跟踪数据时才会生效。

17.6.4.2.1 设置传递性能跟踪数据给用户态

```
int rpmsg_set_perf_data_delivery(int ept_dev_fd, int is_delivery);
```

参数：

- ept_dev_fd：rpmsg 端点设备的文件描述符
- is_delivery：是否传递性能跟踪数据到用户态，0：不传递，其它值：传递

返回值：

- 0：设置成功

失败

17.6.4.2.2 获取性能跟踪数据

```
int rpmsg_get_perf_data(const uint8_t *buf, uint32_t len, rpmsg_perf_data_t *perf_data, uint32_t *payload_offset);
```

参数：

- buf：指向每次通过 rpmsg 端点设备文件读取到的数据 buffer
- len：数据 buffer 长度
- perf_data：指向获取到的性能跟踪数据的指针
- payload_offset：指向 rpmsg payload 在数据 buffer 中的偏移的指针

返回值：

- 0：获取成功
- 其他值：获取失败

17.6.4.2.3 打印性能跟踪数据

```
void rpmsg_dump_perf_data(const rpmsg_perf_data_t *perf_data);
```

参数：

- perf_data：指向性能跟踪数据的指针

此 API 打印的格式如下：

```
Sender  >>
Start         : 0 us [492485.952 ms]
Fill buffer   : 2 us
Notify        : 6 us
End           : 13 us
Receiver<<
Prepare       : 28 us
Exec CB       : 39 us
End           : 1130 us
Time span:
Sender  >>
Prepare       : 2 us
Fill buffer   : 4 us
Notify        : 7 us
Receiver<<
Recv          : 22 us
Prepare       : 11 us
```

: 1091 us

其首先打印各性能跟踪点的相对时间戳信息，然后分别打印各性能跟踪阶段的耗时信息，其中第一个性能跟踪点为隐式性能跟踪点，且会打印其原始的绝对时间戳 (单位为 ms)。上面打印的各性能跟踪点以及各性能跟踪阶段的顺序和上面“性能跟踪点和性能跟踪阶段”章节中介绍的顺序一致，各性能跟踪点和性能跟踪阶段的详细信息请参考此章节。

17.6.5 API 使用方法

Linux 内核态以及 RTOS 环境可参考 rtos/lichee/rtos-components/thirdparty/openamp/rpmsg_demo/rpmsg_ctrl/test/test.c 文件里的相关逻辑来获取 rpmsg 性能跟踪数据，具体使用步骤如下：

1. 在想要获取性能跟踪数据的 rpmsg 回调函数的最后面调用 rpmsg_record_receiver_end_ts 记录时间戳
2. 紧跟着调用 rpmsg_get_perf_data 获取性能跟踪数据并保存
3. 根据具体需求来使用获取到的 rpmsg 性能跟踪数据

参考 platform/allwinner/[illegible] 步骤如下：

1. 对想要获取性能跟踪数据的 rpmsg 端点设备文件调用 rpmsg_set_perf_data_delivery 设置传递性能跟踪数据到用户态
2. 通过 rpmsg 端点设备文件读取数据到用户态的某个 buffer
3. 调用 rpmsg_get_perf_data 来从步骤 2 中的 buffer 里获取性能跟踪数据并保存
4. 根据具体需求来使用获取到的 rpmsg 性能跟踪数据

⚠ 注意

上述 API 都是需要在接收方调用的，因为单次 rpmsg 通信的结束点在接收方这端

17.6.6 使用示例

当前 SDK 中 rpmsg_test 程序支持获取性能跟踪数据并打印，增加参数-p 时就会自动获取性能跟踪数据并打印，下面是具体命令：

RTOS 端：

```
rpmsg_test -N test -L 464 -c -v -p
```

Linux 端：

```
rpmsg_test -r e907_rproc@1a00000 -c test -L 464 -v -p
```

测试日志如下：

RTOS 端：

```
st -N test -L 464 -c -v -p


cpu0>[RV] [AMP_INFO][openamp_ept_open:321]Waiting for ept('sunxi,rpmsg_client') ready
[RV] [AMP_INFO][openamp_ept_open:326]ept('sunxi,rpmsg_client') ready! src: 0x403, dst: 0x403
rpmsg1: binding
rpmsg1 rx thread start...
rpmsg1 tx thread start...
[test]data:876a8b34a281ec3603011f4cf8723b52... [md5:3c7c136f2e354c0f20b7536c6bb357cd]

rpmsg1: rx 464 Bytes
'sunxi,rpmsg_client' performance data:
Timestamp:
Sender  >>
Start          : 0 us [492485.952 ms]
Fill buffer    : 2 us
Notify         : 6 us
End            : 13 us

 :28us         Prepare
Exec CB        : 39 us
End            : 1130 us
Time span:
Sender  >>
Prepare        : 2 us
Fill buffer    : 4 us
Notify         : 7 us
Receiver<<
Recv           : 22 us
Prepare        : 11 us
Exec CB        : 1091 us
data:65bfb259ab096627a8565d03fc00b10a... check:b71b5291be282d4490af7766e00c5cad[test] success
```

Linux 端:

```
907_rproc@1a00000 -c test -L 464 -v -p
[  488.746109] aw_virtio_rpmsg_bus virtio0: creating channel sunxi,rpmsg_client addr 0x403
[rpmsg1]data:65bfb259ab09rpmsg1 recv_thread start.
invalid perf data delivery magic(0x348b6a87), shoule be 0x46524550
6627a8565d03fc00b10a... [md5:b71b5291be282d4490af7766e00c5cad]

get rpmsg performance data failed, ret: -5
[rpmsg1] receive : 0.464000Kb 0.000000ms infMb/s
[rpmsg1] send: 0.464000Kb 0.000000ms infM/s
data:876a8b34a281ec3603011f4cf8723b52... check:3c7c136f2e354c0f20b7536c6bb357cd[rpmsg1] success

'rpmsg1' performance data:
Timestamp:
Sender  >>
Start          : 0 us [492986.915 ms]
Fill buffer    : 11 us
Notify         : 26 us
End            : 44 us

Prepare        : 348 us
Exec CB        : 358 us
End            : 372 us
Time span:
Sender  >>
Prepare        : 11 us
Fill buffer    : 15 us
Notify         : 18 us
Receiver<<
Recv           : 322 us
Prepare        : 10 us
[rpmsg1]data:826caa07b4Exec CB       : 14 us
[rpmsg1] receive : 0.464000Kb 504.000000ms 0.000921Mb/s
234c1cdata:22b45edd0c441a4c2b5e6a0a85cc33... [md5:5cb21f7a6a2357dcad88dc65be6820b9684d... check:0
     b560873a1b6f4062613dcedeb77834951d29f]
```

```
g1]success
```

技巧

上述日志中打印的性能跟踪数据的格式可参考“打印性能跟踪数据”章节

核间通信uf

18.1 RPBuf 介绍

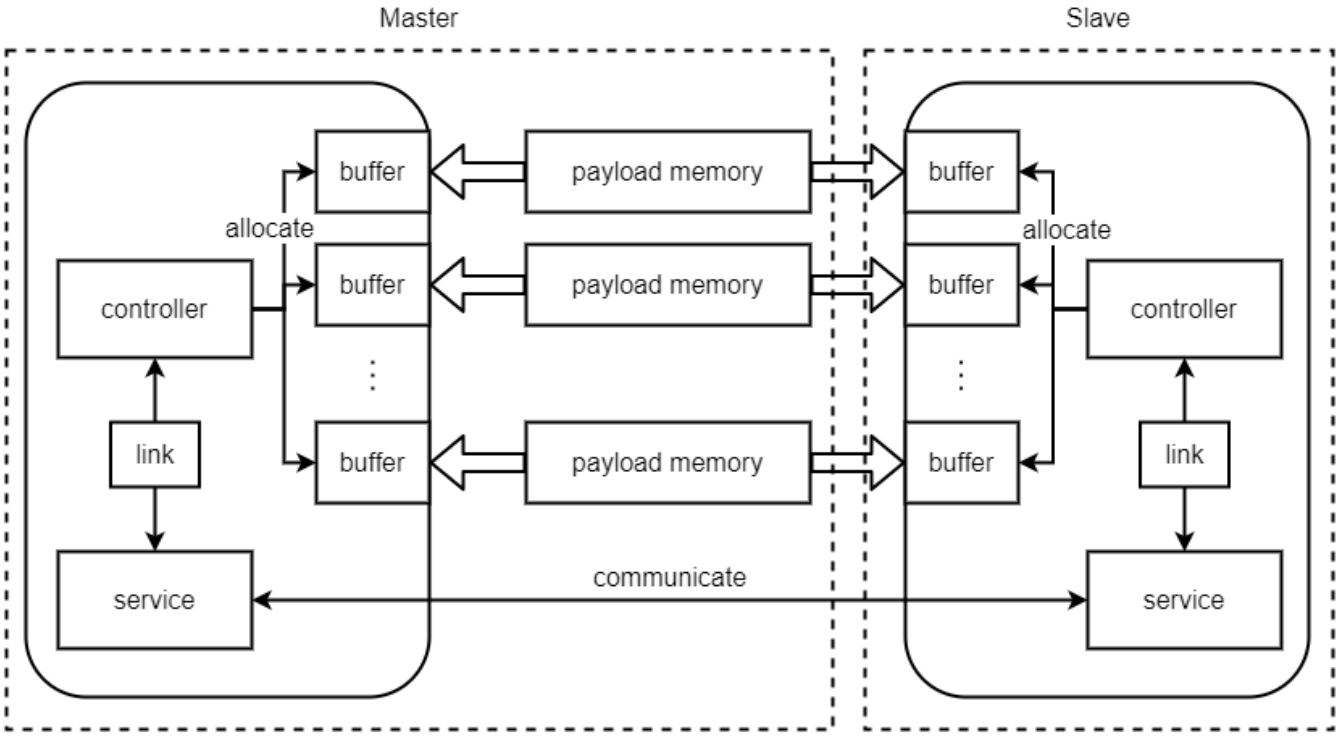
RPBuf（Remote Processor Buffering）是为了能在异构的多处理器之间高效传输大量数据（单次大于 496byte）而设计出来。Linux 原生的 RPMsg 框架已经可以支持在异构核间进行传输数据，但 RPMsg 单次传输的长度最大为 512 bytes（在 Linux 内核代码中用宏来定义），除去头部的 16 bytes，有效数据最大为 496 bytes，除此之外，被传输的数据本身需要被拷贝到共享内存，远端处理器收到通知后再从共享内存中将数据拷贝出来，整个传输过程不可避免地存在 “写入” 和 “读出” 两次拷贝。因此若要传输大量数据，会存在多次拷贝和频繁的核间中断，并不高效。在全志平台中，针对单次传输数据量大于 496bytes 的情况，我们额外设计了 RPBuf 框架。

RPBuf 同样依赖于共享内存来传输数据，但两端的处理器都能直接通过地址访问共享内存，当把数据拷贝到共享内存后，会将数据所在位置的地址偏移和长度使用 rpmsg 告知另一端，让另一端自行从该地址读取数据，从而避免额外的拷贝，实现单次也能传输大量数据。

f 相关概念与流程

18.2.1 RPBuf 框架基本组件

图 18-1: RPBuf Components



- **controller**：供用户操作用来申请 buffer 的接口，一个 controller 对应一个远端处理器。
- **service**：在 RPBuf 内部负责与远端处理器通信的组件，使用的是 RPmsg 机制。rpbuf 在进行跨核数据交互时不进行数据的拷贝操作，而是传输数据在共享内存中的 offset 以及 len 给远端处理器，远端处理器从相同的位置取出相同长度的数据，这需要将数据发送方将 offset 以及 len 封装成 CMD 命令，然后借助 rpmsg 传送到远端处理器。
- **link**：负责将 controller 和 service 连接起来。每一个 controller 都必须与一个 service 连接起来，否则它无法与远端通信。
- **buffer**：表示申请到的共享内存。用户通过操作 buffer 对象，可直接访问对应的共享内存。对于同一片共享内存，在通信的两端都各有一个对应的 buffer 对象，它们的名字和长度相同。
- **payload memory**：用来存放实际传输数据的共享内存，因此称为 payload（有效负载）。它统一由 master 端分配，在两端都各有名字相同的 buffer 对象与其对应。

使用 RPBuf 通信的两端处理器必然有一端作为 master，另一端作为 slave。master 和 slave 对于 buffer 均可进行读写，并没有方向上的限制。两者唯一的区别在于 payload memory 是统一由 master 分配，slave 只负责使用。

为 controller 和 service 两大组件，如此设计的目的在于将 “buffer 管理” 和 “通信” 两个功能解耦。对于内存管理方式不同的平台，可以定义不同的 controller；对于通信方式不同的平台，也可以定义不同的 service，从而做到将不同的 controller 实现和不同的 service 实现进行自由组合。

link 通过名为 **token** 的结构判断哪个 controller 跟哪个 service 连接。每一个 controller 和 service 在注册时都需要指定一个 token，只有 token 相同的才能连接起来。

18.2.2 申请/释放 buffer

用户使用 RPBuf 申请共享 buffer 时，需要在 master 端和 slave 端都指定相同的 “名字” 和 “长度”，否则两端无法匹配上，并且只有在两端都申请成功后，buffer 才可被使用（该状态称为 “**available**”）。

 说明

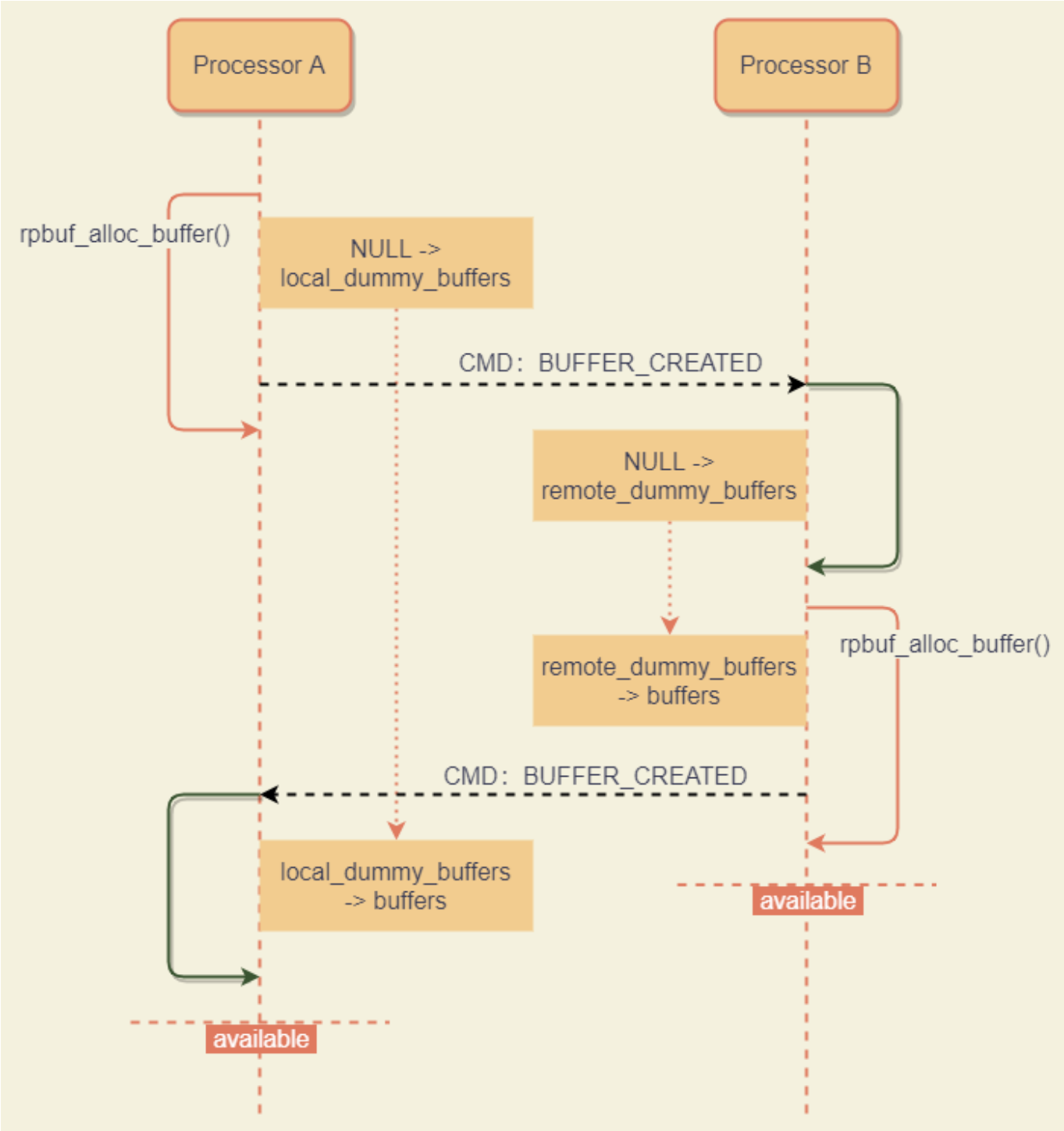
buffer 的 payload memory 是只由 master “分配”，但 “申请 buffer” 这一操作，master 和 slave 都要自己主动进行。

对于 master 和 slave 孰先孰后申请 buffer，并没有什么要求，只要能确保两端最后都申请了就行。在 controller 内部会维护以下三个结构，用来区分 buffer 当前处于什么状态：

- remote_dummy_buffers：用来记录远端已申请、本地未申请的 buffer。
- local_dummy_buffers：用来记录本地已申请、远端未申请的 buffer。
- buffers：用来记录本地和远端均已申请的 buffer。只有在这里的 buffer 才会被认为是 “available”。

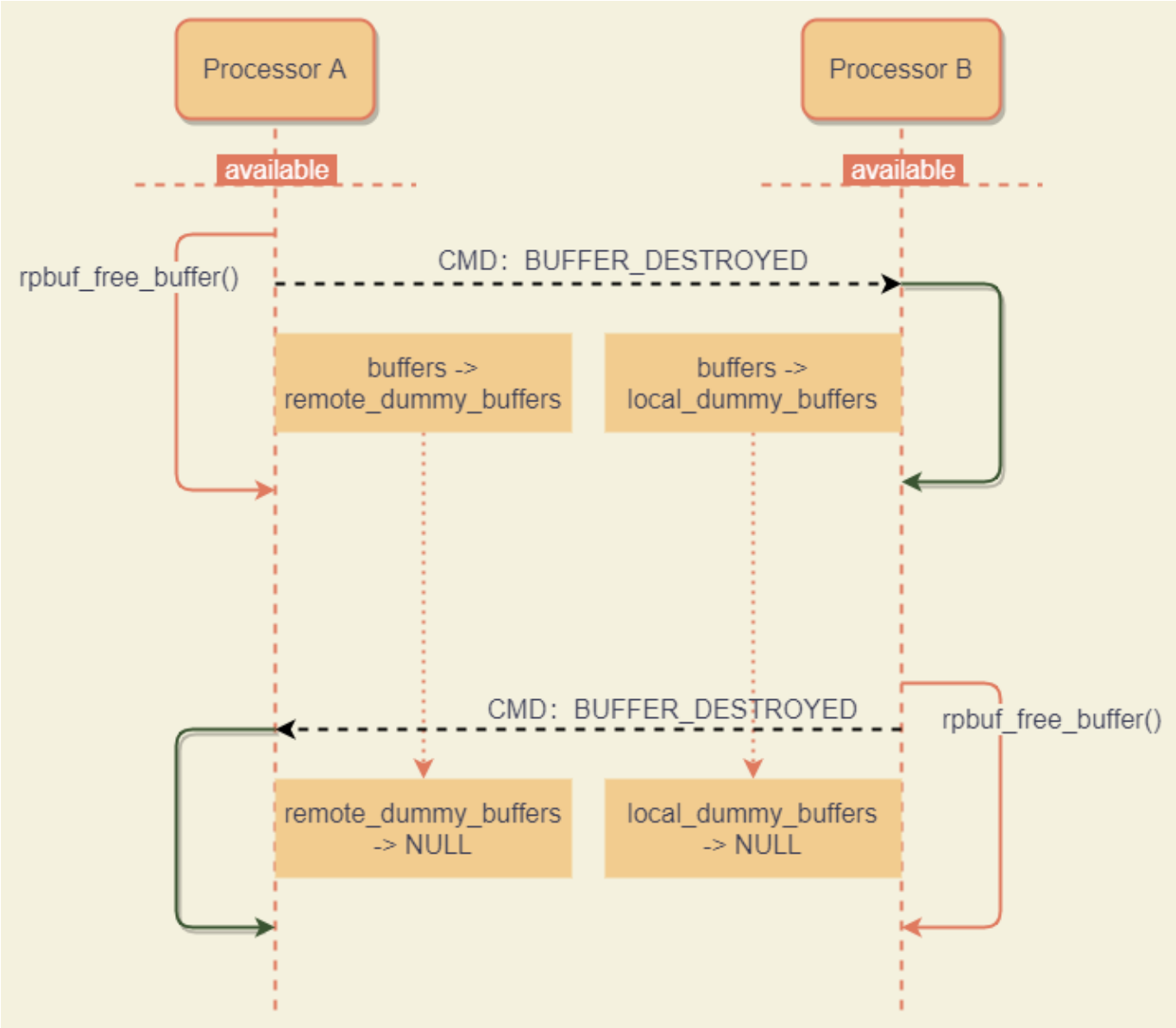
申请 buffer 时其状态切换如下所示：

图 18-2: RPBuf Allocate Buffer Flowchart



释放 buffer 时也类似，无需区分先后顺序：

图 18-3: RPBuf Free Buffer Flowchart

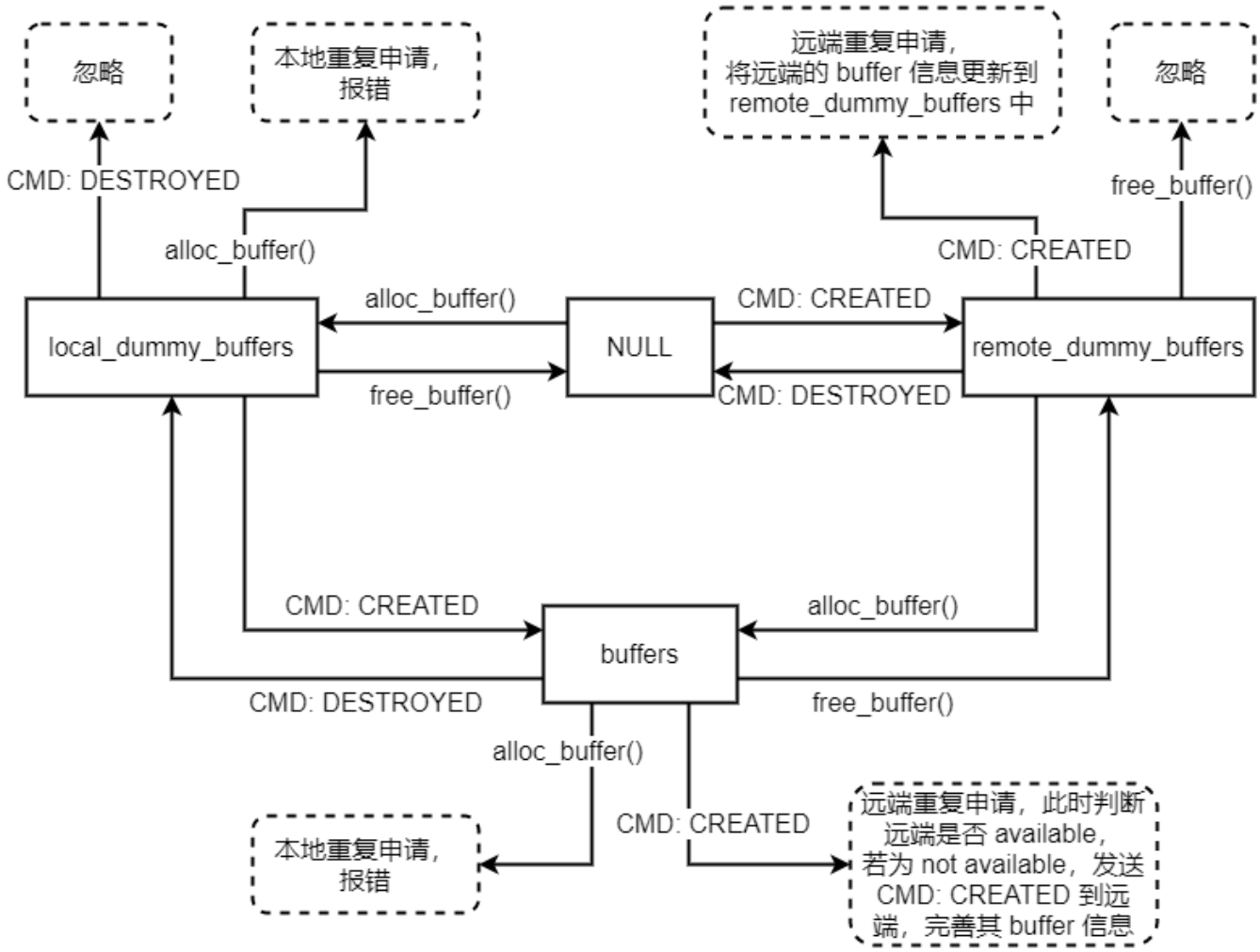


⚠ 注意

在任意一端释放了 buffer 后，实际对两端来说该 buffer 都已经处于 “not available” 的状态，此时都不应该对 buffer 进行任何读写操作。但因为 buffer 的地址是直接暴露给用户，RPBuf 框架无法限制用户对该地址的直接访问，因此用户需自行确保不会在 not available 时读写 buffer。

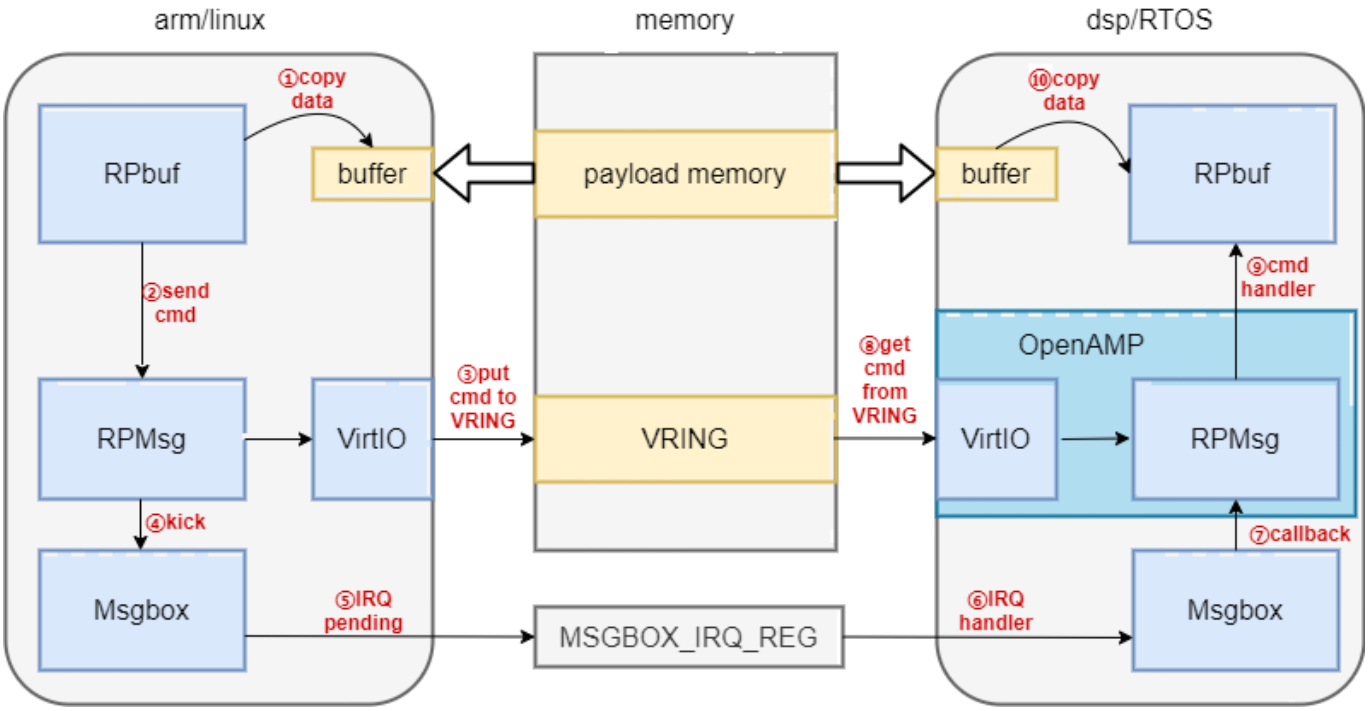
上面流程图描述的是正常情况下 buffer 的状态切换，而在异常情况下（如通信失败、远端处理器崩溃后重新运行等），需要进行一些特殊处理。下图描述的是从单个处理器自身的角度来看，当它的 buffer 分别处于 `NULL`、`local_dummy_buffers`、`remote_dummy_buffers`、`buffers` 时，在本地执行 `rpbuf_alloc_buffer()`/`rpbuf_free_buffer()` 或收到远端传过来的 `RPBUF_SERVICE_CMD_BUFFER_CREATED`/`RPBUF_SERVICE_CMD_BUFFER_DESTROYED` 后，会如何处理：

图 18-4: RPBuf Buffer States



传输流程

图 18-5: RPBuf Wrok



- **RPBuf**：基于共享内存和 RPMsg 消息通知，实现传输大数据传输的框架。
- **RPMsg**：基于 VirtIO 管理的共享内存，实现数据传输的框架。

是一套用在虚拟环境中传输数据的框架，这里用作共享内存

- **OpenAMP**：OpenAMP 框架为 RTOS、裸机和 Linux 用户空间提供了 RPMsg、VirtIO、remoteproc（未列出）的实现，并且与 Linux 内核兼容。
- **Msgbox**：是全志平台提供的一套消息中断机制，已通过 linux 内核中原生的 mailbox 框架作适配。
- **MSGBOX_IRQ_REG**：Msgbox 的中断相关寄存器。
- **buffer**：表示申请到的共享内存。用户通过操作 buffer 对象，可直接访问对应的共享内存。
- **payload memory**：用来存放实际传输数据的共享内存，因此称为 payload（有效负载）。
- **VRING**：由 VirtIO 管理的一个环形共享内存。

上图展示了一次 RPBuf 数据传输的通信过程，下面详细说明：

1. arm 端把数据拷贝到 buffer 中，在初始化时已经将 buffer 和 payload memory 地址绑定，因此数据拷贝后相当于存放到了 payload memory 中。

令后加上数据在 payload memory 中的起始地址和长度，组成数据包，调用 RPMsg 接口发送，在驱动中我们定义了以下几个命令。

cmd	功能
RPBUF_SERVICE_CMD_BUFFER_CREATED	创建命令，通知另一端处理器创建 RPBuf。
RPBUF_SERVICE_CMD_BUFFER_DESTROYED	释放命令，通知另一端处理器释放 RPBuf。
RPBUF_SERVICE_CMD_BUFFER_TRANSMITTED	消息传输命令，传输数据的起始地址和长度给另一端处理器。
RPBUF_SERVICE_CMD_BUFFER_ACK	消息同步命令，用于收发消息的同步。

3. RPMsg 通过调用 VirtIO 的接口，把数据包存放到 VRING 中。
4. RPMsg 通过 kick 的机制，使 Msgbox 产生一个中断。

断标记位置 1。

6. dsp 端的 Mgxbox 接收到中断后，触发其中断处理函数，并把数据放到队列中。
7. Mgxbox 驱动会有一个 task 不断等待队列的数据，当有数据则取出并调用 RPMsg 初始化时注册的 callback。
8. 在 callback 中会调用 VirtIO 的接口，把数据包从 VRING 中取出。
9. 调用 RPBuf 的消息处理函数，解析数据包，根据不同的 cmd 执行不同的 handler，这里收到的 cmd 是 RPBUF_SERVICE_CMD_BUFFER_TRANSMITTED。
10. 根据接收到的地址和长度从 payload memory 中拷贝出数据。

说明

在每次数据拷贝之后，都需要使用 RPMsg 发送消息传输命令，对于单次传输字节数小于 496bytes 时，使用 RPBuf 反而会降低性能，因为不仅需要用 RPMsg 发消息，还多了两次从 payload memory 中拷贝数据的操作，因此建议数据量较小时使用 RPMsg。

同步

当本地处理器将数据拷贝至 payload memory 并发送消息传输命令到远端处理器后，并不知道远端处理器在何时会取走消息，假如远端处理器没有及时取走数据，那么本地处理器可能会再次拷贝造成数据覆盖，旧数据丢失的问题，因此 RPBbuf 在驱动中增加一个数据同步的机制。同步机制的原理：数据发送端在发送数据时，添加一个同步标志，数据接收端的接收数据处理函数在接收数据时，判断发送端是否设置了同步标志，如果设置了同步标志，则回传一个应答，告知数据发送端，数据已经被接收。

18.2.4.1 在 RTOS 中设置同步

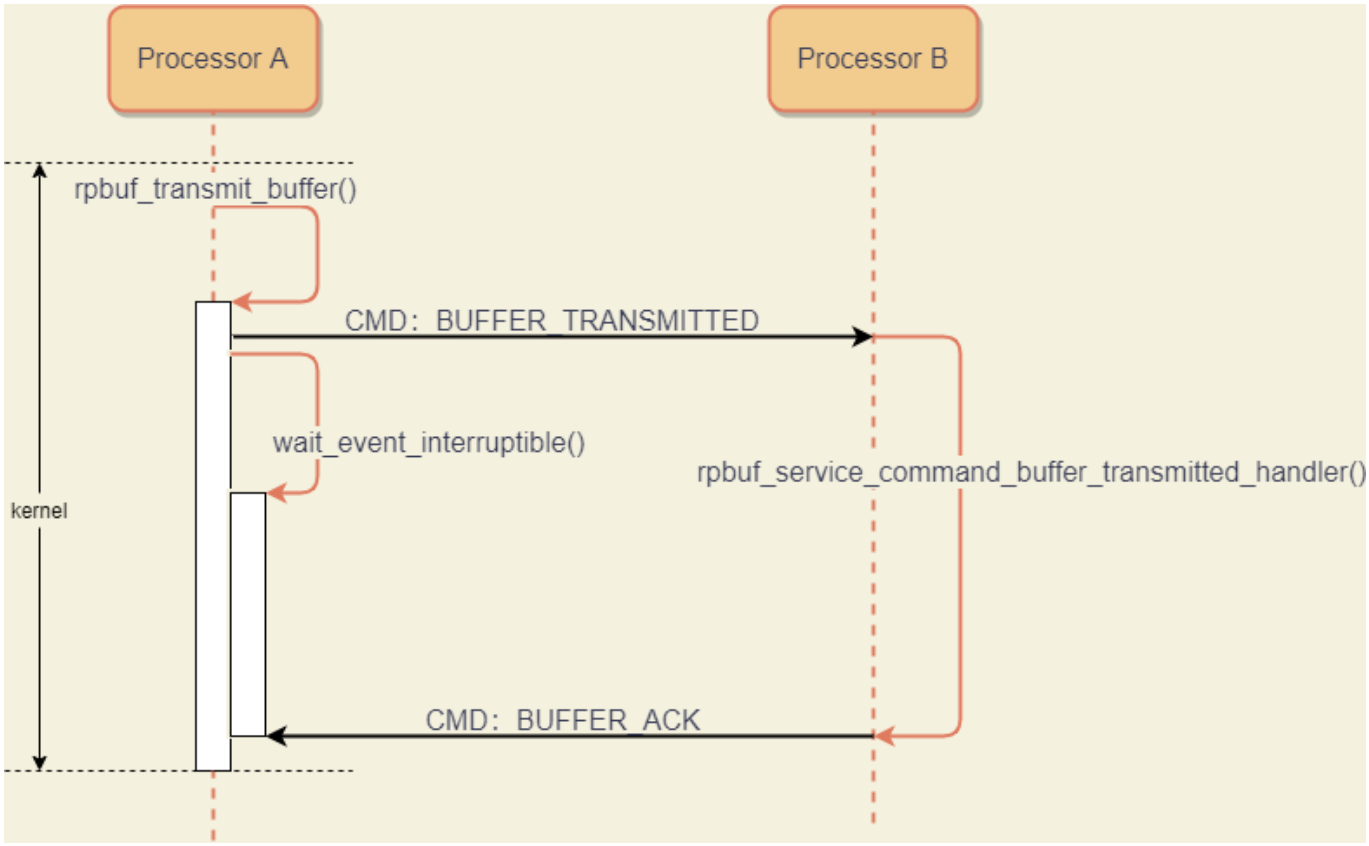
```
int rpbuf_buffer_set_sync(struct rpbuf_buffer *buffer, bool sync)
```

18.2.4.2 在 linux 用户空间中设置同步

```
int rpbuf_set_sync_buffer(rpbuf_buffer_t *buffer, uint32_t is_sync)
```

据同步流程

图 18-6: RPBuf Ack Flowchart



linux 应用层先把数据拷贝到共享内存中，然后调用 rpbuf 的传输接口，这会从应用层切换到内核中，调用`rpbuf_transmit_buffer()`，这会发送一个消息传输命令`RPBUF_SERVICE_CMD_BUFFER_TRANSMITTED`，然后在`wait_event_interruptible()`中阻塞，DSP 收到消息后会根据消息的 CMD 调用不同的处理函数，当解析到消息传输命令会调用`rpbuf_service_command_buffer_transmitted_handler()`，在里面会根据收到的偏移地址和长度刷 cache，然后调用应用中注册的 rx_cb，最后发送一个消息同步命令`RPBUF_SERVICE_CMD_BUFFER_ACK`，linux 收到消息同步命令之后会调用`rpbuf_service_command_buffer_ack_handler()`，在里面会唤醒等待事件，然后内核的阻塞线程被唤醒，内核切换到应用层中继续执行往下的逻辑。

18.2.5 异步通知

在应用层中调用`rpbuf_install_rx_signal()`，可以让内核使用异步通知，当内核收到消息传输命令后，会发送一个`SIGIO`信号给应用层，然后应用层捕获该信号后会调用其处理函数，处理函数通过第二个参数`sighandler_t handler`进行安装，具体用法可以参考rpbuf_demo 程序章节里 linux

说明

目前 linux 每个进程只支持一个 rpbuf 使用异步通知机制，因为 rpbuf 接收消息后内核发出的信号是 SIGIO，进程捕获信号后只会执行对应处理函数，并不会关注该信号从何处来的，也不知道到底是哪一个 rpbuf 收到的消息发出了信号，假如有多个 rpbuf 都调用了`rpbuf_install_rx_signal()`，那么只会触发最后安装的处理函数。

18.3 代码分析

18.3.1 Linux

Linux 内核的 RPBuf 代码如下：

```
├── drivers
│   └── rpbuf
│       ├── rpbuf_controller_sunxi.c    // 全志平台的 rpbuf controller 实现
│       ├── rpbuf_core.c                // rpbuf 核心代码
│       ├── rpbuf_dev.c                 // 在 /dev 下提供节点供用户空间操作
│       ├── rpbuf_internal.h            // rpbuf 内部头文件
│       ├── rpbuf_sample_sunxi.c        // 在内核使用 rpbuf 的一个例子，也提供一些 sysfs 节点供简单调试使用
│       └── rpbuf_service_rpmsg.c       // 通过 rpmsg 实现的 rpbuf service
└── include
    └── linux
        └── rpbuf.h                     // 提供给其他驱动使用的 rpbuf 头文件
```

Tina 系统的 package 中提供了在用户空间使用 rpbuf 的库和示例 demo：

```
platform/allwinner/system/rpbuf/
├── demo          // 用户空间使用 rpbuf 的示例 demo，依赖于 librpbuf
/将与内核rpbuf_dev交互的操作封装成接口供其他应用程序使用
  //用户空间压测rpbuf的示例demo，依赖于librpbuf
```

18.3.1.1 在内核空间使用 RPBuf

供其他内核模块使用的 RPBuf 接口都声明在头文件 `include/linux/rpbuf.h` 中，在使用上大致分为以下步骤：

1. 创建并使能 RPBuf controller（后面以 `rpbuf_controller_sunxi` 为例来说明）。
2. 创建并使能 RPBuf service（后面以 `rpbuf_service_rpmsg` 为例来说明）。
3. 获取到 controller 的指针（一个 controller 对应一个远端处理器）。
4. 通过 controller 申请 buffer（`rpbuf_alloc_buffer()`），待 buffer available 后可对其地址直接进行读写。
5. 使用 `rpbuf_transmit_buffer()` 将数据传输到远端；若远端有数据传输过来，该 buffer 的 `rx_cb()` 会被触发。
6. 使用结束后，释放 buffer（`rpbuf_free_buffer()`）。

18.3.1.1.1 创建并使能 rpbuf_controller_sunxi

除了要使能编译 `rpbuf_controller_sunxi.c` 之外，我们还需要在 DTS 中定义对应的 rpbuf_controller device，以下是一个例子：

```
dsp0_rpbuf_controller: rpbuf_controller@0 {
    compatible = "allwinner,rpbuf-controller";
    remoteproc = <&dsp0_rproc>;
    ctrl_id = <0>;    /* index of /dev/rpbuf_ctrl */
    memory-region = <&dsp0_rpbuf_reserved>;  /* optional */
    status = "okay";
};
```

- `remoteproc` 表示它对应哪个远端处理器，它会以该 remoteproc device 作为 token。
- `ctrl_id` 表示它生成的供用户空间操作的节点 `/dev/rpbuf_ctrlX` 的编号（即这里的 `X` 值）。
- `memory-region`（可选）用于将一片预留内存赋给 controller，让其从中申请 payload memory。例如若我们定义了如下的 reserved-memory `dsp0_rpbuf_reserved`，则 `dsp0_rpbuf_controller` 默认都会从首地址为 0x42244000、长度为 0x2000 这片内存区域来申请 payload memory。

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

    dsp0_rpbuf_reserved:dsp0_rpbuf@42244000 {
        compatible = "shared-dma-pool";
        no-map;
        reg = <0x0 0x42244000 0x0 0x2000>;
```

当 rpbuf_controller_sunxi 驱动加载时，会调用 rpbuf_create_controller() 和 rpbuf_register_controller() 来创建和注册 controller，此时需要指定 enum rpbuf_role，表示该 RPBuf 是作为 master 还是 slave。

```
/* "rpbuf_internal.h" */

struct rpbuf_controller *rpbuf_create_controller(struct device *dev,
                        const struct rpbuf_controller_ops *ops,
                        void *priv);
void rpbuf_destroy_controller(struct rpbuf_controller *controller);

int rpbuf_register_controller(struct rpbuf_controller *controller,
                 void *token, enum rpbuf_role role);
void rpbuf_unregister_controller(struct rpbuf_controller *controller);
```

在调用 rpbuf_create_controller() 时，我们还可以自定义一些 ops，来覆盖 RPBuf 默认的行为。详见章节自定义 ops。

18.3.1.1.2 创建并使能 rpbuf_service_rpmsg

rpbuf_service_rpmsg 是依赖 RPMsg 框架实现，它本质上是一个 rpmsg_driver，因此它的初始化是靠 RPMsg 的 name service 机制实现：只有远端的 rpbuf_service_rpmsg 在初始化过程中给 Linux 发送 name service 消息包后，才会触发 linux 加载这端的 rpbuf_service_rpmsg。这整个过程对 RPBuf 用户来说是隐式的，无法直接感知，但如果 RPBuf service 未初始化完成，调用相关 API 操作 buffer 时会报错，因为 RPBuf 是当且仅当 controller 和 service 都初始化完成后才能正常使用。

当 rpbuf_service_rpmsg 驱动加载时，类似于 controller，也会调用 rpbuf_create_service() 和 rpbuf_register_service() 创建和注册 service。

```
/* "rpbuf_internal.h" */

struct rpbuf_service *rpbuf_create_service(struct device *dev,
                      const struct rpbuf_service_ops *ops,
                      void *priv);
void rpbuf_destroy_service(struct rpbuf_service *service);

int rpbuf_register_service(struct rpbuf_service *service, void *token);
void rpbuf_unregister_service(struct rpbuf_service *service);
```

Linux 的 rpbuf_service_rpmsg 驱动会从它的父设备中寻找实际使用的 remoteproc device，以它作为 token。因为不同内核版本的 remoteproc/RPMsg 代码有异，寻找 remoteproc device 的方式也会有所差异：

Linux-4.9 是往上第 3 个父设备：

```
    /*
     * We should find the underlying remoteproc device.

     * Normally, the rpmsg device is created from:
     *     remoteproc device    <--- defined by vendors, usually in device tree
     *     |--- rproc     <--- rproc_alloc()
     *          |--- virtio    <--- rproc_handle_vdev() & register_virtio_device()
     *               |--- rpmsg    <--- rpmsg_register_device()
     *
     * Therefore the 3rd parent of rpmsg device is the underlying remoteproc
     * device.
     */
    dev_tmp = dev;
    for (i = 0; i < 3; i++) {
        dev_tmp = dev_tmp->parent;
        if (!dev_tmp) {
            dev_err(dev, "cannot get rpmsg device parent %d\n", i);
            ret = -ENOENT;
```

```
oto err_destroy_service;
            }
            dev_dbg(dev, "rpmsg device parent %d: %s\n", i, dev_name(dev_tmp));
        }

        /* We use the underlying remoteproc device as rpbuf link token */
        ret = rpbuf_register_service(service, dev_tmp);
        if (ret < 0) {
            dev_err(dev, "rpbuf_register_service failed\n");
            goto err_destroy_service;
        }
```

Linux-5.4 是往上第 4 个父设备：

```
        /*
         * We should find the underlying remoteproc device.
         *
ay, the rpmsg device
         *     remoteproc device    <--- defined by vendors, usually in device tree
         *     |--- rproc     <--- rproc_alloc()
         *          |--- rproc_vdev    <---- rproc_handle_vdev()
         *               |--- virtio    <--- register_virtio_device()
         *                    |--- rpmsg    <--- rpmsg_register_device()
         *
         * Therefore the 4th parent of rpmsg device is the underlying remoteproc
         * device.
         */
        dev_tmp = dev;
        for (i = 0; i < 4; i++) {
            dev_tmp = dev_tmp->parent;
            if (!dev_tmp) {
                dev_err(dev, "cannot get rpmsg device parent %d\n", i);
                ret = -ENOENT;
                goto err_destroy_service;
            }
bg(dev,%s\n",i,dev_name(dev_tmp));
        }

        /* We use the underlying remoteproc device as rpbuf link token */
        ret = rpbuf_register_service(service, dev_tmp);
        if (ret < 0) {
            dev_err(dev, "rpbuf_register_service failed\n");
            goto err_destroy_service;
        }
```

18.3.1.1.3 获取 controller 的指针

若某一个模块的驱动想要使用 RPBuf，它需要先获取到对应 controller 的指针，方法如下：

首先在该驱动模块的 DTS 中增加一个 `rpbuf` 节点，并将对应的 controller device 节点传给它，以 `rpbuf_sample_sunxi.c`：

```
    rpbuf_sample: rpbuf_sample@0 {
        compatible = "allwinner,rpbuf-sample";
        rpbuf = <&dsp0_rpbuf_controller>;
        status = "okay";
    };
```

如果需要跟多个远端通信，则传递多个 controller：

```
    rpbuf_sample: rpbuf_sample@0 {
        compatible = "allwinner,rpbuf-sample";
        rpbuf = <&dsp0_rpbuf_controller>, <&dsp1_rpbuf_controller>;
        status = "okay";
    };
```

然后通过 `rpbuf_get_controller_by_of_node()` 即可获取到对应的 controller 指针：

```
uf.h> */


/**
 * rpbuf_get_controller_by_of_node - get rpbuf controller by device of_node
 * @np: of_node of the device
 * @index: index to select which of_node to use
 *
 * Return pointer to the rpbuf_controller on success, or NULL on failure.
 */
struct rpbuf_controller *rpbuf_get_controller_by_of_node(const struct device_node *np, int index);
```

18.3.1.1.4 申请/使用/释放 buffer

驱动操作 RPBuf buffer 的 API 位于头文件 include/linux/rpbuf.h 中，主要有如下：

```
(*rpbuf_available_cb_t)(struct rpbuf_buffer*buffer, void *priv);
typedef int (*rpbuf_rx_cb_t)(struct rpbuf_buffer *buffer, void *data, int data_len, void *priv);
typedef int (*rpbuf_destroyed_cb_t)(struct rpbuf_buffer *buffer, void *priv);

struct rpbuf_buffer_cbs {
    rpbuf_available_cb_t available_cb;
    rpbuf_rx_cb_t rx_cb;
    rpbuf_destroyed_cb_t destroyed_cb;
};

struct rpbuf_buffer *rpbuf_alloc_buffer(struct rpbuf_controller *controller,
                    const char *name, int len,
                    const struct rpbuf_buffer_ops *ops,
                    const struct rpbuf_buffer_cbs *cbs,
                    void *priv);
int rpbuf_free_buffer(struct rpbuf_buffer *buffer);

int rpbuf_buffer_is_available(struct rpbuf_buffer *buffer);

int rpbuf_transmit_buffer(struct rpbuf_buffer *buffer,
              unsigned int offset, unsigned int data_len);

const char *rpbuf_buffer_name(struct rpbuf_buffer *buffer);
int rpbuf_buffer_id(struct rpbuf_buffer *buffer);
void *rpbuf_buffer_va(struct rpbuf_buffer *buffer);
phys_addr_t rpbuf_buffer_pa(struct rpbuf_buffer *buffer);
u64 rpbuf_buffer_da(struct rpbuf_buffer *buffer);
int rpbuf_buffer_len(struct rpbuf_buffer *buffer);
void *rpbuf_buffer_priv(struct rpbuf_buffer *buffer);
```

- rpbuf_alloc_buffer() 用于通过 controller 申请 buffer。对同一个 buffer，本地与远端的 name 和 len 必须相同。此时注册的 rpbuf_buffer_cbs 回调函数，会在 buffer 的不同阶段被调用到：
- available_cb：当 buffer 变为 available 时会被触发。
- 端有数据传输过来后会被触发，可获取到数据的首地址和长度。
- destroyed_cb：当远端 buffer 被释放后会被触发。
- rpbuf_free_buffer() 用于释放 buffer（会触发远端的 destroyed_cb()）。
- rpbuf_is_available() 用于判断当前 buffer 是否为 available。一般情况下如果依靠 available_cb 和 destroyed_cb 来监视 buffer 的 available 状态了，就不需要再用 rpbuf_is_available() 来检查了。
- rpbuf_transmit_buffer() 用于将数据传输到远端，会触发远端的 rx_cb()。其中 offset 为数据所在的首地址相对于 buffer 首地址的偏移，data_len 是数据的长度。这允许每次传输时只传 buffer 的一部分，不必将整个 buffer 都传过去。
- 其他 API 用于获取 buffer 的一些信息（因为 struct rpbuf_buffer 的定义不会直接暴露给用户）。其中 rpbuf_buffer_va() 返回的是可供用户驱动代码直接访问的 buffer 首地址的虚拟地址（Virtual Address）。

1. rpbuf_transmit_buffer() 时传入的 offset 是 “数据首地址相对于 buffer 首地址的偏移”，而远端 rx_cb() 被触发后，其 data 指针指向的是 “数据的首地址”，而非偏移，即：data = buffer 首地址 + offset，需注意区分。
2. 因为 RPbuf 是在异构处理器之间传输数据，期间会涉及刷 cache 的操作，使用 rpbuf_transmit_buffer() 时需注意传输的数据是否按 cacheline 对齐（cacheline 大小视具体的处理器平台而定），否则可能导致无关区域的数据受影响。

18.3.1.1.5 自定义 ops

在创建 controller 或申请 buffer 时，可通过一些自定义的 ops 来覆盖 RPBuf 默认的行为，其优先级为 “buffer > controller > 默认”。

```
/* <linux/rpbuf.h> */

/**
 * rpbuf_buffer_ops - buffer operations
 * @alloc_payload_memory: allocate memory for payload
 * @free_payload_memory: free memory for payload
 * @addr_remap: remap pa or da to va
 * @buf_dev_mmap: mmap file operation for buf_dev
 */
struct rpbuf_buffer_ops {
    int (*alloc_payload_memory)(struct rpbuf_buffer *buffer, void *priv);
    void (*free_payload_memory)(struct rpbuf_buffer *buffer, void *priv);
    void *(*addr_remap)(struct rpbuf_buffer *buffer,
                phys_addr_t pa, u64 da, int len, void *priv);
    int (*buf_dev_mmap)(struct rpbuf_buffer *buffer, void *priv, struct vm_area_struct *vma);
};
```

```
/* "rpbuf_internal.h" */

struct rpbuf_controller_ops {
    int (*alloc_payload_memory)(struct rpbuf_controller *controller,
                    struct rpbuf_buffer *buffer, void *priv);
    void (*free_payload_memory)(struct rpbuf_controller *controller,
                    struct rpbuf_buffer *buffer, void *priv);
    void *(*addr_remap)(struct rpbuf_controller *controller,
                phys_addr_t pa, u64 da, int len, void *priv);
    int (*buf_dev_mmap)(struct rpbuf_controller *controller,
                struct rpbuf_buffer *buffer, void *priv,
                struct vm_area_struct *vma);
};
```

- alloc_payload_memory 与 free_payload_memory 只在 master 端使用，用于申请/释放 payload memory。
- addr_remap 只在 slave 端使用，用于将 master 端在申请 payload memory 后传过来的 pa（Physical Address）或 da（Device Address）转换为可供代码直接访问的 va（Virtual Address）。
- 供在 mmap() 将 buffer 的地址映射到用户空间。

18.3.1.2 在用户空间使用 RPBuf

内核代码 rpbuf_dev.c 会生成 /dev/rpbuf_ctrlX 节点供用户空间程序通过 ioctl() 来使用 RPBuf 框架（每一个 /dev/rpbuf_ctrlX 分别对应每一个 controller）。为方便使用，我们基于此再封装成 librpbuf 库，其 API 位于头文件 librpbuf.h 中：

```
typedef struct __rpbuf_buffer rpbuf_buffer_t;

rpbuf_buffer_t *rpbuf_alloc_buffer(int ctrl_id, const char *name, size_t len);
int rpbuf_free_buffer(rpbuf_buffer_t *buffer);

void *rpbuf_buffer_addr(rpbuf_buffer_t *buffer);
```

```
pbuf_buffer_name(rpbuf_buffer_t *buffer);
uffer_len(rpbuf_buffer_t *buffer);

int rpbuf_buffer_is_available(rpbuf_buffer_t *buffer);

int rpbuf_transmit_buffer(rpbuf_buffer_t *buffer, unsigned int offset, unsigned int data_len);
int rpbuf_receive_buffer(rpbuf_buffer_t *buffer, unsigned int *offset,
        unsigned int *data_len, int timeout_ms);
int rpbuf_install_rx_signal(rpbuf_buffer_t *buffer, sighandler_t handler);
int rpbuf_remove_rx_signal(rpbuf_buffer_t *buffer);
int rpbuf_set_sync_buffer(rpbuf_buffer_t *buffer, uint32_t is_sync);
```

这些 API 与内核头文件 include/linux/rpbuf.h 中的类似，详细用法可参考 librpbuf.h 中的注释和示例 demo。

它们与内核 API 的用法区别，主要在于：

- rpbuf_alloc_buffer() 会阻塞等待 buffer 变为 available，或超时退出（超时的等待时间在 rpbuf_dev.c 中用宏 义为 WAIT_TIMEOUT_MSEC
- rpbuf_buffer_addr() 返回的是执行 mmap() 后将 buffer 地址映射上来的地址，可供用户空间程序直接使用。
- rpbuf_buffer_is_available() 同样是用于检查 buffer 是否为 available，但因为涉及用户态和内核态的切换，并不适合太过频繁地使用。
- 将数据传输到远端同样是使用 rpbuf_transmit_buffer，用法也类似；但因为没有 rx_cb，接收数据则需要应用程序主动调用 rpbuf_receive_buffer() 来判断远端是否传输数据过来（阻塞与否通过 timeout_ms 指定）。
- rpbuf_install_rx_signal() 可以让内核使用异步通知，当内核收到消息传输命令后，会发送一个 SIGIO 信号给应用层，然后应用层捕获该信号后会调用其处理函数，处理函数通过第二个参数 sighandler_t handler 进行安装，具体用法可以参考rpbuf_demo 程序章节里 linux 的源码。
- rpbuf_remove_rx_signal() 是让应用捕获到 SIGIO 信号后执行系统默认操作，不再调用自定义安装的处理函数，与 rpbuf_install_rx_signal() 成对使用。
- rpbuf_set_sync_buffer() 的作用是设置为同步传输，每次调用完 rpbuf_transmit_buffer() 都会阻塞，直到收到一个消息同步命令之后退出。

在应用程序两次调用 rpbuf_receive_buffer() 之间，如果远端传输了两次或以上的数据过来，rpbuf_receive_buffer() 只能拿到最近一次的 offset 和 data_len，因为底层没有缓冲区来保存历史值。这种情况在系统繁忙时比较容易出现，需要应用程序调用 rpbuf_set_sync_buffer() 设置同步方式来避免覆盖旧数据。

18.3.2 RTOS

RTOS 下的 RPBuf 代码结构与 Linux 的类似，但因为不区分内核空间和用户空间，所以不需要 rpbuf_dev 和 librpbuf 这两部分功能的代码。

```
rtos-components/aw/rpbuf/
├── include
│   └── rpbuf.h
├── Kconfig
├── Makefile
├── rpbuf_common.h
├── rpbuf_controller.c
├── rpbuf_core.c
├── rpbuf_demo
│   ├── Makefile
│   ├── modules.order
│   ├── rpbuf_demo.c
│   └── rpbuf_test.c
├── rpbuf_internal.h
├── rpbuf_perf.c
├── rpbuf_reserved_mem.c
└── rpbuf_service_rpmsg.c

rtos-components/thirdparty/openamp/rpbuf_demo/
├── Makefile
```

RPBuf 对外的 API 都位于头文件 include/rpbuf.h 中，大部分的用法与 Linux 的内核 API 类似。

区别较大的主要为用于初始化的 API。RTOS 没有 Linux 那样的设备驱动模型，所以是直接调用以下 API 进行初始化，这些步骤一般放在 rtos/lichee/rtos-components/thirdparty/openamp/rpbuf_demo/rpbuf.c 中进行。

```
/* <rpbuf.h> */

/**
 * rpbuf_init_service - Init service
 * @id: service id
 * @token: used to link with controller
 * @attached_op: attached operation for rpbuf service. What it does depends on
 *               the rpbuf_init_service() implementation.
 * @attached_priv: private data used for attached_op().
 *
 * What @id and @token means is defined by the backend implementation of
 * rpbuf_init_service.
 *
 * Return pointer to rpbuf service on success, or NULL on failure.
 */
struct rpbuf_service *rpbuf_init_service(int id, void *token,
                    rpbuf_service_attached_op_t attached_op,
                    void *attached_priv);

/**
 * rpbuf_deinit_service - The opposite operation of rpbuf_init_service
 * @service: rpbuf service
 */
void rpbuf_deinit_service(struct rpbuf_service *service);

/**
 * rpbuf_init_controller - Init
 * @id: controller id
 * @token: used to link with service
 * @role: MASTER or SLAVE
 * @ops: user-specific rpbuf controller operations
 * @priv: user-specific private data
 *
 * What @id and @token means is defined by the backend implementation of
 * rpbuf_init_controller
 *
 * Return pointer to rpbuf controller on success, or NULL on failure.
 */
struct rpbuf_controller *rpbuf_init_controller(int id, void *token, enum rpbuf_role role,
                           const struct rpbuf_controller_ops *ops,
                           void *priv);

/*
 * rpbuf_deinit_controller - The opposite operation of rpbuf_init_controller
 * @r: rpbuf controller
 */
void rpbuf_deinit_controller(struct rpbuf_controller* controller);
```

18.4 rpbuf_demo 程序

此应用主要是用来展示 rpbuf 的创建，释放，接收和发送等基本功能的应用，每次执行-s 命令发送一条消息，linux 接收部分用的是异步通知机制。

格径

18.4.1.1 Linux

```
platform/allwinner/system/rpbuf/demo/main.c
```

18.4.1.2 RTOS

```
rtos/lichee/rtos-components/aw/rpbuf/rpbuf_demo/rpbuf_demo.c
```

18.4.2 编译

linux 需要在 make menuconfig 中选上：

```
Allwinner --->
    System  --->
        <*> rpbuf_demo          # CONFIG_PACKAGE_rpbuf_demo
```

DSP 需要在 make menuconfig 中选上：

```
System components  --->
    aw components  --->
        RPBuf framework  --->
            [*] RPBuf demo          # CONFIG_COMPONENTS_RPBUF_DEMO
```

重新编译烧录固件后，在控制台中执行 help 命令可看到增加了 rpbuf_demo 命令。

18.4.3 基本使用

f_demo 的 help 信息如下：

```
USAGE:
  rpbuf_demo [OPTIONS]

OPTIONS:
  -s MESSAGE  : send MESSAGE
  -d time     : delay free buffer (default: 100 ms)
  -r          : receive messages
  -t time     : specifies the time of receive messagess, unit:ms
  -a          : sync transmit
  -I ID       : specify rpbuf ctrl ID (default: 0)
  -N NAME     : specify buffer name (default: "rpbuf_demo")
  -L LENGTH   : specify buffer length (default: 32 bytes)
  -O OFFSET   : specify offset in buffer to store data (default: 0)

  e.g.
s "hello" : send string "hello"
  rpbuf_demo -r           : receive data forever
  rpbuf_demo -r -t 1000   : receive data 1000 msecond
```

dsp 下的 rpbuf_demo 的 help 信息如下：

```
USAGE:
  rpbuf_demo [OPTIONS]
OPTIONS:
  -h          : print help message
  -c          : create buffer
  -d          : destory buffer
  -s STRING   : copy STRING to buffer and send to remote
  -l          : list created buffers
  -I ID       : specify controller ID (default: 0)
```

```
cify buffer name (default: "rpbuf_demo")
   -L LENGTH   : specify buffer length (default: 32 bytes)
   -O OFFSET   : specify offset in buffer to store data (default: 0)

e.g.
   First, create a buffer (its name and length should match that of remote rpbuf buffer):
       rpbuf_buffer -N "xxx" -L LENGTH -c
   Then if remote sends data to it, the buffer callback will be called.

   We can send data (e.g. "hello") to remote:
       rpbuf_demo -N "xxx" -s "hello"

   If this buffer is no longer in use, destroy it:
       rpbuf_demo -N "xxx" -d
```

首先，DSP 端使用 `-c` 选项创建 rpbuf（可使用 `-N` 指定名字，使用 `-I` 指定远端通信核的 ID、使用 `-L` 指定长度）。通过 `-I` 选项可列出之前创建过的 rpbuf。

中各 ID 分别对应哪个远端核：

本地核	ID：0	ID：1
CPU	DSP0	DSP1
DSP0	CPU	DSP1
DSP1	CPU	DSP0

DSP 端创建了 rpbuf 之后，如果远端有消息发送过来，其回调函数会被调用到，将收到的数据打印出来。

使用 `-s` 选项可将某个字符串发送到远端。

如果不再使用该 rpbuf，使用 `-d` 参数进行销毁。

demo 最简单的使用方式：

```
linux与DSP0通信：

（ID默认是0，NAME默认rpbuf_demo）

1、DSP0输入rpbuf_demo -c

2、linux输入rpbuf_demo -s "hello"，DSP0打印输出

3、linux输入rpbuf_demo -r

4、DSP0输入rpbuf_demo -s "123"，linux打印输出
```

```
linux与DSP1通信：

AME默认rpbuf_demo）

1、DSP1输入rpbuf_demo -c

2、linux输入rpbuf_demo -I 1 -s "hello"，DSP1打印输出

3、linux输入rpbuf_demo  -I 1 -r

4、DSP1输入rpbuf_demo -s "123"，linux打印输出
```

```
DSP0与DSP1通信：

（因为不能创建同名buffer，假如前面已经创建了名字为rpbuf_demo的buffer，则需要-N指定名字）

1、DSP0输入rpbuf_demo -I 1 -N "rpbuf_dsp_demo" -c
```

```
uf_demo -I 1 -N "rpbuf_dsp_demo" -c

3、DSP0输入rpbuf_demo -I 1 -N "rpbuf_dsp_demo" -s "hellp"，DSP1打印输出

4、DSP1输入rpbuf_demo -I 1 -N "rpbuf_dsp_demo" -s "hellp"，DSP0打印输出
```

技巧

两个核申请创建 rpbuf 时，必须指定名字和长度一致，这样才会访问到同一块内存区域。

18.4.4 rpbuf_test 程序

18.4.4.1 代码路径

此应用是在 rpbuf_demo 的基础上修改而来，可以自定义发送消息的数量和时间间隔，linux 接收部分用的是轮询机制。

18.4.4.1.1 linux

```
platform/allwinner/system/rpbuf/test/main.c
```

18.4.4.1.2 RTOS

```
rtos/lichee/rtos-components/aw/rpbuf/rpbuf_demo/rpbuf_test.c
```

18.4.4.2 编译

linux 需要在 make menuconfig 中选上：

```
Allwinner --->
    System  --->
        <*> rpbuf_test          # CONFIG_PACKAGE_rpbuf_test
```

DSP 需要在 make menuconfig 中选上：

```
System components  --->
    aw components  --->
        RPBuf framework  --->
            [*] RPBuf demo          # CONFIG_COMPONENTS_RPBUF_DEMO
```

重新编译烧录固件后，在控制台中执行 help 命令可看到增加了 rpbuf_test 命令。

本使用

linux 下的 rpbuf_test 的 help 信息如下：

```
USAGE:
  rpbuf_test [OPTIONS]

OPTIONS:
  -d time     : set data sending interval (default: 100 ms)
  -s          : send test messages
  -c          : send count (default: 10)
  -r          : receive messages
  -t time     : specifies the time of receive messagess, unit:ms
  -a          : sync transmit
  -v          : verbosely print check result
```

```
: specify rpbuf ctrl ID (default: 0)
    -N NAME     : specify buffer name (default: "rpbuf_test")
    -L LENGTH   : specify buffer length (default: 32 bytes)

  e.g.
    rpbuf_test -L 0x1000 -c 10 -s -a : send 10 test data, size=0x1000
    rpbuf_test -L 0x1000 -c -1 -s -a : send test data all the time, size=0x1000
    rpbuf_test -L 0x1000 -r          : receive test data forever, size=0x1000
    rpbuf_test -L 0x1000 -r -t 1000  : receive test data 1 second, size=0x1000
```

dsp 下的 rpbuf_test 的 help 信息如下：

```
  USAGE:
    rpbuf_test [OPTIONS]
  OPTIONS:
    -h          : print help message
    -c          : create buffer
: destory buffer
    -s          : send test messagese
    -l          : list created buffers
    -a          : sync transmit
    -v          : verbosely print check result
    -C          : send count (default: 1,always send: -1)
    -I ID       : specify controller ID (default: 0)
    -N NAME     : specify buffer name (default: "rpbuf_test")
    -D TIME     : set data sending interval (default: 100 ms)
    -L LENGTH   : specify buffer length (default: 32 bytes)

  e.g.
    First, create a buffer (its name and length should match that of remote rpbuf buffer):
        rpbuf_test -N "xxx" -L LENGTH -c -a
    Then if remote sends data to it, the buffer callback will be called.

    We can send 100 test data to remote:
        rpbuf_test -N "xxx" -C 100 -s

    We can always send test data to remote, interval time 5ms:
        rpbuf_test -N "xxx" -C -1 -D 5 -s

    If this buffer is no longer in use, destroy it:
        rpbuf_test -N "xxx" -d
```

首先，DSP 端使用 `-c` 选项创建 rpbuf（可使用 `-N` 指定名字，使用 `-I` 指定远端通信核的 ID、使用 `-L` 指定长度）。通过 `-l` 选项可列出之前创建过的 rpbuf。

下表列出本地核中各 ID 分别对应哪个远端核：

本地核	ID：0	ID：1
CPU	DSP0	DSP1
DSP0	CPU	DSP1
CPU	DSP0	

DSP 端创建了 rpbuf 之后，如果远端有消息发送过来，DSP 定义的回调函数会被调用到，将收到的数据打印出来。

使用 `-s` 选项可创建发送线程，发送随机生成的数据并在接收端中校验。

如果不再使用该 rpbuf，使用 `-d` 参数进行销毁。

下面演示 rpbuf_test 最简单的使用方式：

```
linux与DSP0通信：

（ID默认是0，NAME默认rpbuf_test）
```

```
linux发dsp0收：
1、dsp0执行rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -c
2、linux执行rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -s -c -1 -a &

dsp0发linux收：
1、dsp0执行rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -c -a
2、linux执行rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -r &
3、dsp0执行rpbuf_test -N "dsp0_to_cpu" -C -1 -s
```

```
linux与DSP1通信：

（ID默认是0，NAME默认rpbuf_test）

linux发dsp1收：
1、dsp1执行rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp1" -c
2、linux执行rpbuf_test -I 1 -L 1024 -N "cpu_to_dsp1" -s -c -1 -a&

dsp1发linux收：
1、dsp1执行rpbuf_test -I 0 -L 1024 -N "dsp1_to_cpu" -c -a
2、linux执行rpbuf_test -I 1 -L 1024 -N "dsp1_to_cpu" -r &
3、dsp1执行rpbuf_test -I 0 -N "dsp1_to_cpu" -C -1 -s
```

```
DSP0与DSP1通信：

（ID默认是0，NAME默认rpbuf_test）

dsp0发dsp1收：
1、dsp1执行rpbuf_test -I 1 -L 1024 -N "dsp0_to_dsp1" -c
2、dsp0执行rpbuf_test -I 1 -L 1024 -N "dsp0_to_dsp1" -c -a
3、dsp0执行rpbuf_test -N "dsp0_to_dsp1" -C -1 -s

dsp1发dsp0收：
1、dsp0执行rpbuf_test -I 1 -L 1024 -N "dsp1_to_dsp0" -c
uf_test -I 1 -L 1024 -N "dsp1_to_dsp0" -c -a
uf_test -N "dsp1_to_dsp0" -C -1 -s
```

技巧

两个核申请创建 rpbuf 时，必须指定名字和长度一致，这样才会访问到同一块内存区域。

18.5 rpbuf 传输性能测试

用户可以使用 rpbuf_test 命令对 rpbuf 进行性能测试，测试发送方向和接收方向各自的耗时以及速率。本章节主要描述这个测试的操作步骤并解读测试结果。

18.5.1 应用层测试方法

下面的指令是主核与从核之间的传输性能测试指令，指令的参数请查看“rpbuf_test 程序”章节。测试方法以及测试结果汇总如下：

1. 测试一：linux 发 dsp0 收

测试方法：

```
首先，dsp0执行：rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -c
然后，linux执行：rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -s -c -1 -a &   #注意：-a启用了数据同步，这可以避免接收端没有及时提取接收到的
      数据，导致数据可能被发送端的新数据覆盖
```

从核测试结果：

图 18-7: linux 发 dsp0 收的从核测试结果

```
cpu0>rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -c

cpu0>[RPBUF_INFO][rpbuf_addr_remap_default:206]reamp pa:0x42380000 -> va:0x42380000
[RPBUF_INFO][rpbuf_service_command_buffer_created_handler:827]buffer "cpu_to_dsp0" (id:0): local_dummy_buffers ->
buffers
buffer "cpu_to_dsp0" is available
buffer[0] "cpu_to_dsp0" received data (addr: 42380000, offset: 0, len: 1024):
[0]data:a45c750b031b4b3021223f7f81108a4b... check:2eddc2e0e988bcc77b8113ae24977e68 success

buffer[1000] "cpu_to_dsp0" received data (addr: 42380000, offset: 0, len: 1024):
[1000]data:39e15e25d2425a4d86fdc04888d2d376... check:2bde6b98b771ddbcee525d0bd6cec684 success
```

主核测试结果：

图 18-8: linux 发 dsp0 收的主核测试结果

```
root@TinaLinux:/# rpbuf_test -I 0 -L 1024 -N "cpu_to_dsp0" -s -c -1 -a &
root@TinaLinux:/# [  107.663616] sunxi-rpbuf-controller rpbuf_controller@0: "cpu_to_dsp0" allocate payload memory
: va 0x0000000078fbe985, pa 0x0000000042380000, len 1024
[  107.665255] sunxi-rpbuf-controller rpbuf_controller@0: buffer "cpu_to_dsp0" (id:0): remote_dummy_buffers -> bu
ffers
data:a45c750b031b4b3021223f7f81108a4b... [md5:2eddc2e0e988bcc77b8113ae24977e68]

ping: 3.795000ms
bandwidth: N/AMbps
data:a45c750b031b4b3021223f7f81108a4b... [md5:2eddc2e0e988bcc77b8113ae24977e68]

ping: 0.276000ms
bandwidth: N/AMbps
data:a45c750b031b4b3021223f7f81108a4b... [md5:2eddc2e0e988bcc77b8113ae24977e68]
```

测试结果说明：

【【【从核测试结果说明】】】

1. reamp pa:0x42380000 -> va:0x42380000 #说明：rpbuf的payload内存的物理起始地址以及其映射后的虚拟起始地址
2. buffer "cpu_to_dsp0" (id:0): local_dummy_buffers -> buffers， #说明rpbuf的名称为cpu_to_dsp0，控制器的ID为0，local dummy_buffers 转换成 buffers
3. buffer "cpu_to_dsp0" is available， #说明：名称为cpu_to_dsp0的rpbuf为可用状态，可以进行数据的收发
4. buffer[0] "cpu_to_dsp0" received data (addr: 42380000, offset: 0, len: 1024) #说明：dsp端从名cpu_to_dsp0的rpbuf中接收到数据，rpbuf的起始地址为0x42380000，rpbuf接收到的数据长度为1024个字节，数据在rpbuf的偏移为0。
5. [0]data:a45c750b031b4b3021223f7f81108a4b... check:2eddc2e0e988bcc77b8113ae24977e68 success ： #说明：“[0]data”：中的“[0]”是指接收数据的次数不足一千次，如果超过1千次不足2千次，则会是“[1000]data”，中括号中的数值是接收次数对1000求商的结果。“[0]data”后的则是接收到的数据，“check:”后的则是接收到的数据的校验和。

【【【主核测试结果说明】】】

1. sunxi-rpbuf-controller rpbuf_controller@0: "cpu_to_dsp0" allocate payload memory: va 0x0000000078fbe985, pa 0x0000000042380000 4 #说明：rpbuf_controller分配了名为cpu_to_dsp0的rpbuf有效负载内存，这段内存的物理起始地址为：0x42380000，映射到虚拟起始地址为0x78fbe985，rpbuf的长度为1024字节
2. sunxi-rpbuf-controller rpbuf_controller@0: buffer "cpu_to_dsp0" (id:0): remote_dummy_buffers -> buffers #说明：rpbuf controller0的rpbuf名为cpu_to_dsp，远端的dummy_buffers转换成buffers
3. data:a45c750b031b4b3021223f7f81108a4b... [md5:2eddc2e0e988bcc77b8113ae24977e68] #说明：发送的数据以及其md5校验和
4. ping: 3.795000ms #说明：linux数据传输到dsp，dsp提取了数据，dsp发送应答给linux，linux响应了应答，同步数据传输过程的耗时
5. bandwidth: N/AMbps #说明：linux端发送数据，不显示带宽，N/A是 "Not Applicable" 的缩写，意为不适用

2. 测试二：dsp0 发 linux 收

测试方法：

首先，dsp0执行：rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -c -a #注意：-a表示启用数据同步，数据同步可以避免接收端没有及时提取接收到的数据，旧数据被新数据覆盖
然后，linux执行：rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -r &

```
:rpbuf_test -N "dsp0_to_cpu" -C -1 -s
```

从核测试结果：

图 18-9: dsp0 发 linux 收的从核测试结果

```
cpu0>rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -c -a

cpu0>[RPBUF_INFO][rpbuf_addr_remap_default:206]reamp pa:0x42380000 -> va:0x42380000
[RPBUF_INFO][rpbuf_service_command_buffer_created_handler:827]buffer "dsp0_to_cpu" (id:0): local_dummy_buffers ->
buffers
buffer "dsp0_to_cpu" is available


cpu0>rpbuf_test -N "dsp0_to_cpu" -C -1 -s
[0]data:8335de23565a7c474fe35860bbbaaa67... [md5:d471c87434e19dd147368f74b80f18b0]
```

主核测试结果：

图 18-10: dsp0 发 linux 收的主核测试结果

```
root@TinaLinux:/# [  191.219058] sunxi-rpbuf-controller rpbuf_controller@0: buffer "dsp0_to_cpu": NULL -> remote_
dummy_buffers

root@TinaLinux:/# rpbuf_test -I 0 -L 1024 -N "dsp0_to_cpu" -r &
root@TinaLinux:/# [  199.571700] sunxi-rpbuf-controller rpbuf_controller@0: "dsp0_to_cpu" allocate payload memory
: va 0x00000000cfacdadb, pa 0x0000000042380000, len 1024
[  199.573239] sunxi-rpbuf-controller rpbuf_controller@0: buffer "dsp0_to_cpu" (id:0): remote_dummy_buffers -> bu
ffers
ping: 12286.769531ms
bandwidth: 0.000667Mbps
data:8335de23565a7c474fe35860bbbaaa67... check:d471c87434e19dd147368f74b80f18b0 success
```

测试结果说明：

```
【【【从核测试结果说明】】】
1. reamp pa:0x42380000 -> va:0x42380000   #说明：rpbuf的payload在内存中的物理起始地址和映射后的虚拟起始地址
2. buffer "dsp0_to_cpu" (id:0): local_dummy_buffers -> buffers  #说明：rpbuf的名称为cpu_to_dsp0，控制器的ID为0，local dummy_buffers
     转换成 buffers
3. buffer "dsp0_to_cpu" is available  #说明：名称为cpu_to_dsp0的rpbuf已经是可用状态，可以进行数据的收发
4. [0]data:8335de23565a7c474fe35860bbbaaa67... [md5:d471c87434e19dd147368f74b80f18b0]  #说明：dsp发送的数据以及数据的校验和数据的校
     验和，“[0]”中括号中的数值是发送次数对1000求商的结果

【【【主核测试结果说明】】】
1. sunxi-rpbuf-controller rpbuf_controller@0: "dsp0_to_cpu" allocate payload memory: va 0x00000000cfacdadb, pa 0
     x0000000042380000, len 1024    #说明：rpbuf_controller分配了名为cpu_to_dsp0的rpbuf有效负载内存，这段内存的物理起始地址为：
     0x42380000，映射后的虚拟起始地址为0xcfacdadb，rpbuf的长度为1024字节
f-controller rpbuf_controller@0: buffer "cpu_to_dsp0" (id:0): remote_dummy_buffers -> buffers #说明：rpbuf
     controller0的rpbuf名为cpu_to_dsp，远端的dummy_buffers转换成buffers
3. ping: 12286.769531ms     #说明：获得远端发送过来的数据的offset、len的时长
4. bandwidth: 0.000667Mbps  #说明：接收数据的速率，单位：兆位每秒
5. data:8335de23565a7c474fe35860bbbaaa67... check:d471c87434e19dd147368f74b80f18b0 success     #说明：接收到的数据以及数据的MD5校验
     和，sucess表示接收成功
```

18.6 rpbuf 通信耗时测量

rpbuf 框架目前支持通信耗时测量功能，又叫性能跟踪，可通过此功能测量单次 rpbuf 通信过程中发送方和接收方各个阶段的耗时信息。

18.6.1 工作原理

通信双方记录 rpbuf 发送接收流程中某些执行点的时间戳，且通信双方获取到的时间戳具有相同的时间基准。

跟踪点和性能跟踪阶段

目前支持如下几个性能跟踪点：

- RPBUF_PERF_POINT_LOCAL_FLUSH：发送方开始刷 cache
- RPBUF_PERF_POINT_LOCAL_NOTIFY：发送方开始通知对端
- RPBUF_PERF_POINT_LOCAL_WAIT_ACK：发送方等待 ACK
- RPBUF_PERF_POINT_REMOTE_RECV：接收方准备接收
- RPBUF_PERF_POINT_REMOTE_EXEC_CB：接收方开始执行回调函数
- RPBUF_PERF_POINT_REMOTE_SEND_ACK：接收方开始发送 ACK
- RPBUF_PERF_POINT_REMOTE_FINISH：接收方结束
- RPBUF_PERF_POINT_LOCAL_FINISH：发送方结束

其中还有一个隐式的性能跟踪点，为发送方开始发送的时刻，提供给用户的 API 中没有此性能跟踪点，因为用户 API 获取到的数据是经过处理后数据 (相对时间戳)，故此隐式跟踪点的相对时间戳为0。

目前支持如下几个性能跟踪阶段：

- RPBUF_PERF_STAGE_LOCAL_PREPARE：发送方准备
- RPBUF_PERF_STAGE_LOCAL_FLUSH：发送方刷 cache
- RPBUF_PERF_STAGE_LOCAL_NOTIFY：发送方通知对端
- RPBUF_PERF_STAGE_REMOTE_RECV：发送方通知到接收方准备接收
- RPBUF_PERF_STAGE_REMOTE_PREPARE：接收方准备
- RPBUF_PERF_STAGE_REMOTE_EXEC_CB：接收方执行回调函数
- RPBUF_PERF_STAGE_REMOTE_SEND_ACK：接收方发送 ACK
- RPBUF_PERF_STAGE_LOCAL_FINISH：发送方结束

上述这些性能跟踪点和性能跟踪阶段都有对应的宏，Linux 环境可在 bsp/include/linux/rpbuf.h 文件里找到，RTOS 环境可在 rtos/lichee/rtos-xx/include/rpbuf.h 文件里找到。

注意

rpbuf 处于非同步模式下上述性能跟踪点和性能跟踪阶段宏名中带有 REMOTE 的为无效项 (不会记录时间戳信息)

18.6.3 配置信息

主核 Linux 环境需要启用内核配置 CONFIG_AW_RPBUF_PERF_TRACE。

从核 RTOS 环境需要启用配置 CONFIG_AW_RPBUF_PERF_TRACE。

由于通信耗时测量功能基于 AMP 时间戳模块，故 AMP 时间戳模块也需要配置，另外 Linux 和 RTOS 环境还需分别配置所使用的 AMP 时间戳设备的 ID，其中 Linux 是在 dts 里配置，RTOS 是配置 CONFIG_RPBUF_AMP_TS_DEV_ID，且 Linux 环境下此配置的值需要和 dts 里 AMP 时间戳，RTOS 配置的值需要和 dts 里 AMP 时间戳设备的 ID 相同才能获取相同时基的时间戳

18.6.4 API 说明

rpbuf 通信涉及到通信双方，涉及到主核 Linux 和从核 RTOS，因此主核 Linux 环境和从核 RTOS 环境都有类似 API。另外由于主核 Linux 环境同时支持在内核态和用户态使用 rpbuf，故其 API 也区分内核态和用户态。不过目前 SDK 中 Linux 内核态 API、Linux 用户态 API 以及 RTOS 环境 API 都是一致的。

18.6.4.1 rpbuf_perf_data_t 数据结构

rpbuf_perf_data_t 表示某次通信过程中的性能跟踪数据：

```
PERF_POINT_LOCAL_FLUSH0
  #define RPBUF_PERF_POINT_LOCAL_NOTIFY 1
  #define RPBUF_PERF_POINT_LOCAL_WAIT_ACK 2
  #define RPBUF_PERF_POINT_REMOTE_RECV 3
  #define RPBUF_PERF_POINT_REMOTE_EXEC_CB 4
  #define RPBUF_PERF_POINT_REMOTE_SEND_ACK 5
  #define RPBUF_PERF_POINT_REMOTE_FINISH 6
  #define RPBUF_PERF_POINT_LOCAL_FINISH 7

  #define RPBUF_PERF_STAGE_LOCAL_PREPARE 0
  #define RPBUF_PERF_STAGE_LOCAL_FLUSH 1
  #define RPBUF_PERF_STAGE_LOCAL_NOTIFY 2
  #define RPBUF_PERF_STAGE_REMOTE_RECV 3
  #define RPBUF_PERF_STAGE_REMOTE_PREPARE 4
  #define RPBUF_PERF_STAGE_REMOTE_EXEC_CB 5
  #define RPBUF_PERF_STAGE_REMOTE_SEND_ACK 6
  #define RPBUF_PERF_STAGE_LOCAL_FINISH 7

PERF_STAGE_NUM(RPBUF_PERF_STAGE_LOCAL_FINISH+1)
PERF_POINT_NUMRPBUF_PERF_STAGE_NUM

  typedef struct rpbuf_perf_data {
      int is_sync;
      __u32 timestamp[RPBUF_PERF_POINT_NUM];
      __u32 time_span[RPBUF_PERF_STAGE_NUM];
      __u64 raw_timestamp;
  } rpbuf_perf_data_t;
```

成员说明：

- is_sync：此 rpbuf 是否是同步模式
- timestamp：性能跟踪点的相对时间戳数组
- time_span：性能跟踪阶段的耗时信息数组
- raw_timestamp：隐式性能跟踪点的原始时间戳 (绝对时间戳)

其中时间戳信息和耗时信息的单位都是 us。

获取到性能跟踪数据后，可使用 RPBUF_PERF_POINT_XXX 宏以及 RPBUF_PERF_STAGE_XXX 宏来拿到具体性能跟踪点的相对时间戳或性能跟踪阶段的耗时。

18.6.4.2 获取性能跟踪数据

```
int rpbuf_get_perf_data(struct rpbuf_buffer *buffer, rpbuf_perf_data_t *perf_data);
```

参数：

- buffer：指向 rpbuf 的指针
- perf_data：指向获取到的性能跟踪数据的指针

- 0：获取成功
- 其他值：获取失败

18.6.4.3 打印性能跟踪数据

```
void rpbuf_dump_perf_data(const rpbuf_perf_data_t *perf_data);
```

参数：

- perf_data：指向性能跟踪数据的指针

式如下 (以同步模式为例

```
Mode: Sync
Timestamp:
>Start      : 0 us [1598572.584 ms]
>Flush cache : 6 us
>Notify      : 13 us
>Wait ACK    : 68 us
  <Recv      : 158 us
  <Exec CB   : 164 us
  <Send ack  : 176 us
  <Finish    : 178 us
>End         : 417 us
Time span:
>Prepare     : 6 us
>Flush cache : 7 us
>Notify     : 55 us
  <Recv        : 90 us

  <Exec CB     : 12 us
  <Send ACK    : 2 us
>Local finish: 239 us
```

us

其首先打印各性能跟踪点的相对时间戳信息，然后分别打印各性能跟踪阶段的耗时信息，其中第一个性能跟踪点为隐式性能跟踪点，且会打印其原始的绝对时间戳 (单位为 ms)。上面打印的各性能跟踪点以及各性能跟踪阶段的顺序和上面“性能跟踪点和性能跟踪阶段”章节中介绍的顺序一致，各性能跟踪点和性能跟踪阶段的详细信息请参考此章节。

⚠ 注意

rpbuf 处于非同步模式下上述 API 会在开头打印 “Mode: Async”，且不会打印以 “ <” 开头的性能跟踪点和性能跟踪阶段的信息，因此这些性能跟踪点和性能跟踪阶段此时是无效的。

使用方法

可参考 rtos/lichee/rtos-components/aw/rpbuf/rpbuf_demo/rpbuf_test.c 文件或 platform/allwinner/system/rpbuf/test/main.c 文件里的相关逻辑来获取 rpbuf 性能跟踪数据，具体使用步骤如下：

1. 调用 rpbuf_get_perf_data 获取某个 rpbuf 某次通信的性能跟踪数据并保存
2. 根据具体需求来使用获取到的 rpbuf 性能跟踪数据

⚠ 注意

上述 API 都是需要在发送方调用的，因为 rpbuf 处于同步模式时单次 rpbuf 通信的结束点在发送方这端，虽然非同步模式时单次 rpbuf 通信的结束点在接收方这端，但考虑到一致性且非同步模式使用的比较少，故设计成和同步模式时一样在发送方获取了。

18.6.6 使用示例

uf_test 程序支持获取性能跟踪数据并打印，下面是具体命令：

Linux 发 RTOS 收：

```
Linux：rpbuf_test -s -c 10 -I 0 -N test -L 0x1000 -a -p
RTOS：rpbuf_test -c -I 0 -N test -L 0x1000 -a
```

RTOS 发 Linux 收：

```
Linux：rpbuf_test -r -I 0 -N test -L 0x1000 -a
RTOS：rpbuf_test -s -C 10 -I 0 -N test -p
```

测试日志如下:

Linux 端：

```
x:/#rpbuf_test -s -c 10 -I 0 -N test -L 0x1000 -a -p
[ 1499.609737] sunxi-rpbuf-controller rpbuf_controller@0: "test" allocate payload memory: va 0x000000005a806874, pa 0
     x0000000042384000, len 4224
[ 1499.623991] sunxi-rpbuf-controller rpbuf_controller@0: buffer "test" (id:0): remote_dummy_buffers -> buffers
data:27ca300320bce9766c6bdc002f4d9c5b... [md5:121537c4296e79bf2420881c14a47207]

ping: 29.291000ms
bandwidth: N/AMbps
Mode: Sync
Timestamp:
>Start       : 0 us [1503347.735 ms]
>Flush cache : 2 us
>Notify      : 3 us
>Wait ACK    : 18 us
  <Recv      : 14564 us
  <Exec CB   : 14575 us
  <Send ack  : 28931 us
  <Finish    : 28980 us
: 29276 us

Time span:
>Prepare     : 2 us
>Flush cache : 1 us
>Notify     : 15 us
  <Recv        : 14546 us
  <Preprae     : 11 us
  <Exec CB     : 14356 us
  <Send ACK    : 49 us
>Local finish: 296 us
data:27ca300320bce9766c6bdc002f4d9c5b... [md5:121537c4296e79bf2420881c14a47207]

ping: 0.543000ms
bandwidth: N/AMbps
Mode: Sync
Timestamp:
>Start       : 0 us [1503480.788 ms]
>Flush cache : 3 us
: 4 us

>Wait ACK    : 26 us
  <Recv      : 48 us
  <Exec CB   : 58 us
  <Send ack  : 201 us
  <Finish    : 247 us
>End         : 525 us
Time span:
>Prepare     : 3 us
>Flush cache : 1 us
>Notify     : 22 us
  <Recv        : 22 us
  <Preprae     : 10 us
  <Exec CB     : 143 us
  <Send ACK    : 46 us
>Local finish: 278 us
data:27ca300320bce9766c6bdc002f4d9c5b... [md5:121537c4296e79bf2420881c14a47207]
...
sunxi-rpbuf-controller rpbuf_controller@0: buffer "test" (id:0): buffers -> remote_dummy_buffers
root@TinaLinux:/#
root@TinaLinux:/#
root@TinaLinux:/# rpbuf_test -r -I 0 -N test -L 0x1000 -a
[ 1581.067145] sunxi-rpbuf-controller rpbuf_controller@0: buffer "test" (id:0): remote_dummy_buffers -> buffers
ping: 6762.024902ms
bandwidth: 0.004846Mbps
data:ffa85e5c1ec15049a5c20f18ee41c726... check:52e4d5cd03193300cde15564a56a24c1 success

...

ping: 0.030000ms
bandwidth: 318.135925Mbps
data:ffa85e5c1ec15049a5c20f18ee41c726... check:52e4d5cd03193300cde15564a56a24c1 success
```

RTOS 端：

```
st -c -I 0 -N test -L 0x1000 -a

  cpu0>[RPBUF_INFO][rpbuf_addr_remap_default:206]reamp pa:0x42384000 -> va:0x42384000
  [RPBUF_INFO][rpbuf_service_command_buffer_created_handler:827]buffer "test" (id:0): local_dummy_buffers -> buffers
  buffer "test" is available
  buffer[0] "test" received data (addr: 42384000, offset: 0, len: 4096):
  [0]data:27ca300320bce9766c6bdc002f4d9c5b... check:121537c4296e79bf2420881c14a47207 success

  [RPBUF_INFO][rpbuf_service_command_buffer_destroyed_handler:965]buffer "test" (id:0): buffers -> local_dummy_buffers
  buffer "test": remote buffer destroyed
  [RPBUF_INFO][rpbuf_service_command_buffer_destroyed_handler:985]wakeup buffer test, state 2
  [RPBUF_INFO][rpbuf_addr_remap_default:206]reamp pa:0x42384000 -> va:0x42384000
  [RPBUF_INFO][rpbuf_service_command_buffer_created_handler:827]buffer "test" (id:0): local_dummy_buffers -> buffers
  buffer "test" is available

  cpu0>rpbuf_test -s -C 10 -I 0 -N test -p
  Mode: Sync
  Timestamp:
0us[1598572.584ms]
  >Flush cache : 6 us
  >Notify      : 13 us
  >Wait ACK    : 68 us
    <Recv      : 158 us
    <Exec CB   : 164 us
    <Send ack  : 176 us
    <Finish    : 178 us
  >End         : 417 us
  Time span:
  >Prepare     : 6 us
  >Flush cache : 7 us
  >Notify     : 55 us
    <Recv        : 90 us
    <Preprae     : 6 us
    <Exec CB     : 12 us
    <Send ACK    : 2 us
  >Local finish: 239 us

  cpu0>[RPBUF_INFO][rpbuf_service_command_buffer_destroyed_handler:965]buffer "test" (id:0): buffers -> local_dummy_buffers
  buffer "test": remote buffer destroyed
  [RPBUF_INFO][rpbuf_service_command_buffer_destroyed_handler:985]wakeup buffer test, state 2
```

技巧

上述日志中打印的性能跟踪数据的格式可参考“打印性能跟踪数据”章节

其他

19.1 rpmsg 需知

1. 端点是 rpmsg 通信的基础；每个端点都有自己的 src 和 dst 地址，范围（1 - 1023，除了 0x35）。
2. **rpmsg 每次发送数据最大为 512 -16 字节；**（数据块大小为 512，头部占用 16 字节）。
3. rpmsg 使用 name server 机制，当从核创建的端点名，和 linux 注册的 rpmsg 驱动名一样的时候，rpmsg bus 总线会调用其 probe 接口。所以如果需要 Linux 端主动发起创建端点并通知从核，则需要借助上面提到的 rpmsg_ctrl 驱动。
4. linux 端，rpmsg 是串行调用回调的，故建议 rpmsg_driver 的回调中不要调用耗时长的函数，避免影响其他 rpmsg 驱动的运行。
5. rpmsg 的回调函数是在线程环境下执行的，并不是中断环境。

FAQ

20.1 riscv 中怎么从 kernel 中获得一块 reserved 内存，此块内存 linux kernel 不会修改，只被 riscv 修改

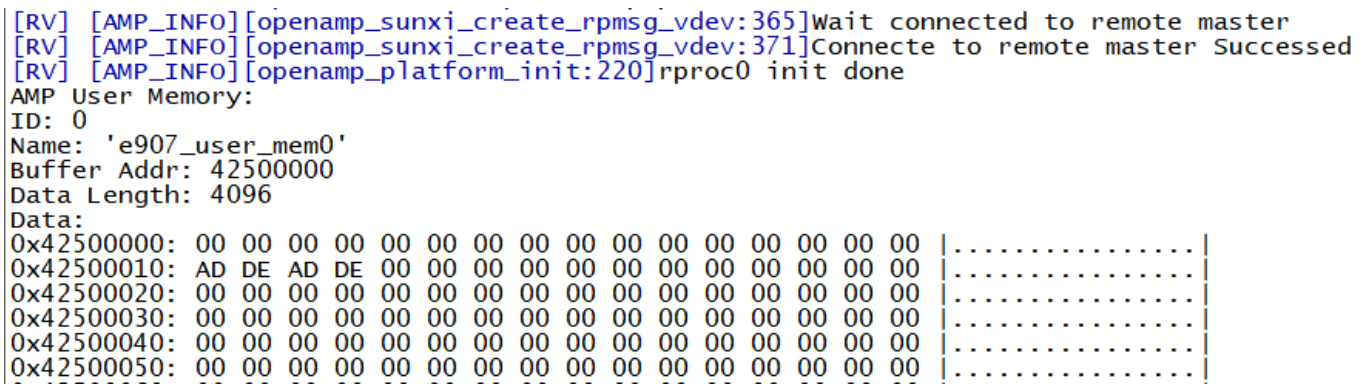
Linux 端，dts 里增加保留内存信息，例如 e907_user_mem0@42500000，然后在 e907_rproc@1a00000 节点的 memory-region 属性里增加这些保留内存的 phandle 引用：

```
diff --git a/configs/evb1/linux-5.15-origin/board.dts b/configs/evb1/linux-5.15-origin/board.dts
index 34134b7..6aef291 100644
--- a/configs/evb1/linux-5.15-origin/board.dts
+++ b/configs/evb1/linux-5.15-origin/board.dts
@@ -103,6 +103,25 @@
                        no-map;
};


+               e907_user_mem0: e907_user_mem0@42500000 {
+                       reg = <0x0 0x42500000 0x0 0x1000>;
+                       no-map;
+               };
+
+               e907_user_mem1: e907_user_mem1@42501000 {
+                       reg = <0x0 0x42501000 0x0 0x2000>;
+                       no-map;
+               };
+
+               e907_user_mem2: e907_user_mem2@42503000 {
+                       reg = <0x0 0x42503000 0x0 0x4000>;
+                       no-map;
+               };
+
+               e907_user_mem3: e907_user_mem3@42507000 {
reg=<0x00x425070000x00x8000>;
+                       no-map;
+               };
        };

        rpbuf_controller0: rpbuf_controller@0 {
@@ -794,7 +813,8 @@
        e907_rproc: e907_rproc@1a00000 {
                mboxes = <&msgbox 4>, <&msgbox 6>;
                mbox-names = "arm-kick", "arm-standby";
-               memory-region = <&rv_vdev0buffer>, <&rv_vdev0vring0>, <&rv_vdev0vring1>, <&e907_dram_reserved>, <&
     e907_share_irq_table>;
+               memory-region = <&rv_vdev0buffer>, <&rv_vdev0vring0>, <&rv_vdev0vring1>, <&e907_dram_reserved>, <&
     e907_share_irq_table>,
+                                                  <&e907_user_mem0>, <&e907_user_mem1>, <&e907_user_mem2>, <&e907_user_mem3>;
                auto-boot;
                fw-region = <&e907_mem_fw>;
                fw-partitions = "riscv0", "riscv0-r";
```

RTOS 端：启用配置 CONFIG_COMPONENTS_AMP_USER_MEMORY、CONFIG_AMP_USER_MEMORY_0。另外根据需要的保留内存数量决定是否开启配置 CONFIG_AMP_USER_MEMORY_1、CONFIG_AMP_USER_MEMORY_2 以及 CONFIG_AMP_USER_MEMORY_3。

若需要 RTOS 系统启动后自动打印 AMP user memory 的信息，可启用配置 CONFIG_DUMP_ALL_AMP_USER_MEMORY_INFO，自动打印效果如下：

图 20-1: amp 用户内存信息



RTOS 端 API：

```
typedef struct amp_user_memory
{
    uint32_t id;
    const char *name;
    uint32_t len;
    void *addr;
} amp_user_memory_t;

int get_amp_user_memory(uint32_t id, amp_user_memory_t *user_mem);
void show_amp_user_memory(uint32_t id);
void show_all_amp_user_memory(void);

API使用方法：
1、调用get_amp_user_memory获取AMP user memory；
2、拿到amp_user_memory_t的addr成员，即为保留内存的起始地址。
```

注意：

1. RTOS 端配置 CONFIG_AMP_USER_MEMORY_DT_NODE_NAME_PREFIX 的值需要和 Linux 端保留内存的节点名去掉 id 后相同，目前此配置的值默认为 e907_user_mem。
2. 若使用多个 AMP user memory，建议这些 AMP user memory 的内存区域相邻 (上面的 dts 配置就是相邻的)，以节省 RISC-V 的 PMP 配置数目。
3. 在主核 Linux 环境重启从核后，AMP user memory 里的数据仍会保留，并不会清空，若需要清空，可在从核 RTOS 环境里自己调用 API 清空。

20.2 如何增加 rv 内存

请查阅“专用内存区域”章节，修改设备树中的预留内存，并调整从核的链接器脚本文件。

rpmsg 端点有哪些方式

释放 rpmsg 端点有三种方式，用户可以通过对 `/dev/rpmsg_ctrl-xxx` 设备文件进行操作 ioctl 操作来释放端点：

- RPMSG_DESTROY_EPT_IOCTL：释放端点 group 中的一个端点
- RPMSG_REST_EPT_GRP_IOCTL：释放端点 group 中的所有端点，保留端点 group
- RPMSG_DESTROY_ALL_EPT_IOCTL：释放所有的端点，删除端点 group

用户在设计 app 时，应考虑 app 异常处理逻辑，在 app 异常时通过 ioctl 释放端点。如果 app 异常退出没有释放端点，下次分配端点时，在端点分组还有可用端点的情况下，会继续从端点分组中分配新的端点来进行 rpmsg 通信，但异常前分配的端点并不会释放。此时内核有类似如下的异常打印：

```
pmsg_eptdev_official_cb: 1490 callbacks suppressed
[  634.408391] rpmsg rpmsg1: rx queue is full.
[  634.419966] rpmsg rpmsg2: rx queue is full.
[  634.424742] rpmsg rpmsg3: rx queue is full.
异常打印解读：有三个端点并没有正常释放，并且每个没有释放的端点的接收队列都满了。
```

为避免这种情况，app 在异常后重启的第一时间，需使用 RPMSG_REST_EPT_GRP_IOCTL 来释放异常前分配的端点。当使用 RPMSG_REST_EPT_GRP_IOCTL 释放异常前分配的端点时，内核会有类似如下的打印：

```
[ 5338.270562] virtio_rpmsg_bus virtio0: destroying channel sunxi,rpmsg_client addr 0x405   说明：释放rpmsg3端点
[ 5338.298587] virtio_rpmsg_bus virtio0: destroying channel sunxi,rpmsg_client addr 0x404   说明：释放rpmsg2端点
[ 5338.315583] virtio_rpmsg_bus virtio0: destroying channel sunxi,rpmsg_client addr 0x403   说明：释放rpmsg1端点
```

20.4 uboot 启动从核时的处理逻辑是怎样的

，首先是要依赖 bootcmd=run boot_riscv 以及 `boot_riscv=bootrv${rv_fw_load_addr}${rv_fw_load_size}0riscv0ris`
cv0-r 这两个 boot 命令，当执行 bootrv 命令时，则会先将 riscv0 分区中的从核固件加载到装载内存中（装载内存是一段临时缓存 ELF 固件的内存，并不是从核的运行内存），装载内存地址为 ${rv_fw_load_addr}。然后会调用 sunxi_riscv_init 来初始化从核，sunxi_riscv_init 中的处理流程如下：

1. 从加载了 riscv0 分区的装载内存中，获取从核的运行内存地址
2. load_elf_fw 接口会把 ELF 固件从装载内存中，拷贝到运行内存
3. 通过 remoteproc 框架的 rproc_load 接口调用到全志自定义的 sunxi_rproc_load 接口。这个接口会根据是否在设备树中定义了 auto_boot 来决定是否处理资源。如果定义了 auto_boot，则会在 uboot 阶段查找资源表（ELF 中的.resource_table 段）并对其进行处理
4. 通过 remoteproc 框架的 rproc_start 接口调用到全志自定义的 sunxi_rproc_start 接口
5. 最后则配置从核的时钟和复位，把从核固件的运行地址写到 RISCV_STA_ADD_REG 中，此时，从核就可以运行，但此时还不能进行 rpmsg 异核通信。从核运行起来后，会阻塞等待主核更新资源表。

20.5 uboot 启动从核之后，kernel 阶段的从核镜像怎么来的

当从核提前被 uboot 启动起来后，[illegible] 调用，在
中会通过判断从核的时钟是否已经被 uboot 打开，如果时钟打开，则说明从核已经被提前启动，此时内核驱动会将从核状态标记为 RPROC_DETACHED，表明此时从核等待被主核 attach。

接着，sunxi_rproc_probe 接口调用 rproc_add 接口向 remoteproc 核心层注册 remoteproc 设备（远端处理器），rproc_add 最终会调用到 rproc_boot 接口来启动远程处理器，rproc_boot 接口调用 rproc_attach 后，主核附加到远程处理器，附加后从核状态标记为 RPROC_ATTACHED，rproc_attach 则会通过 rproc_prepare_device 来执行启动远程处理器所需的所有操作，其中一个操作就是要把从核固件加载到内存（sunxi_request_firmware）。由于 uboot 已经启动从核，因此在内存中有一份从核固件的，所以可以直接使用这份内存中的从核固件，所以在这种场景下，内核会打印提示： “load fw from memory”

20.6 从 kernel 直接启动从核的流程是怎样的

内核直接启动从核时，不会走 DETACHED 转 ATTACHED 的流程，在 sunxi_request_firmware 接口中，优先从 riscv0 分区中加载从核固件，如果 riscv0 中没有则会在 riscv0-r 分区中加载从核固件。如果能从分区中加载固件，则内核会打印提示： “load fw from partition riscv0” 或
[illegible] 加载固件，加载成功时有类似打印提示：
lesystem, filename: amp_rv0.bin”

20.7 /lib/firmware 目录下的 RV 镜像是否还使用，是否可以删除

不建议删除。正常启动过程中，当 riscv0 分区以及 riscv0-r 分区分区镜像不可用时，系统会使用/lib/firmware 下的固件。从核 panic 时，主核 recover 从核时，需要读取/lib/firmware 下的固件文件，以及在主核 linux 命令行对从核进行 stop&start 操作时，也是需要读取/lib/firmware 下的固件文件。这里的 “读取” 只是把固件文件读取到内存中。如果用户的存储空间确实有限，可以在将/lib/firmware 的固件文件内容清空后，往固件文件中写入多于 1 个字节的任意数据，即保留一个相同文件名且占用空间不多的非空文件，这种应用场景下，分区下的固件一定要是完好无损的，否则系统快启启动从核会失败，stop&start 也会失败。 “将/lib/firmware 的固件文件替换成非空且同名的文件” 的操作，不会影响 attached 以及 stop&start 流程，因为这两个流程会通过 rproc_prepare_device –sunxi_rproc_prepare – sunxi_request_firmware 这个调用过程选择从优先内存、分区中加载固件。如果用户的存储空间充裕，建议保留/lib/firmware 下的从核固件，这样即使分区中的固件出现损坏，还有一份备份固件可以正常使用。

从核时该如何操作

riscv0 分区、riscv0-r 分区、/lib/firmware 下的从核固件，这三者都是同一个固件，加载顺序是优先加载 riscv0 分区中的从核固件，其次是 riscv0-r 分区中的从核固件，最后是/lib/firmware 目录下的从核固件。目前，全志方案是只烧录了 riscv0 分区下从核固件以及打包从核固件到/lib/firmware 文件目录。用户 OTA 时，也同时更新 riscv0 分区以及/lib/firmware 下的从核固件即可。

用户调试过程更新固件可以参考“固件更新”章节。

文档问题反馈: aw-document@allwinnertech.com

商标声明

（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。