OpenAMP项目文档

OpenAMP的整体架构

OpenAMP是用于异构SOC的管理框架，支持处理器的生命周期管理（称为remoteproc）和处理器间的通信（rpmsg）。Linux已经有remoteproc和rpmsg两个机制，为什么还需要OpenAMP呢？因为Linux已有机制只能在Linux侧控制其它处理器核，不能反之。而OpenAMP通过不同的操作系统抽象层支持了Linux，freertos, nutt, zephyr, bare metal，提供了系统的灵活性。

为了便于移植和开发，OpenAMP分为open-amp和libmetal两个组件。设计上libmetal负责操作系统的抽象层（包括内存管理，同步原语，设备管理等），open-amp仅仅需要考虑也业务相关的逻辑。open-amp中实现了和前述Linux类似的rpmsg和remoteproc。前者基于virtio管理share memory，后者直接使用libmetal抽象的memory做远端处理器的生命周期管理。实际实现中，rpmsg的share memory的页表属性，中断管理仍然和具体硬件相关，需要单独为不同硬件平台实现。remoteproc，也就是远端处理器管理，也和硬件相关，需要每个硬件平台单独实现。

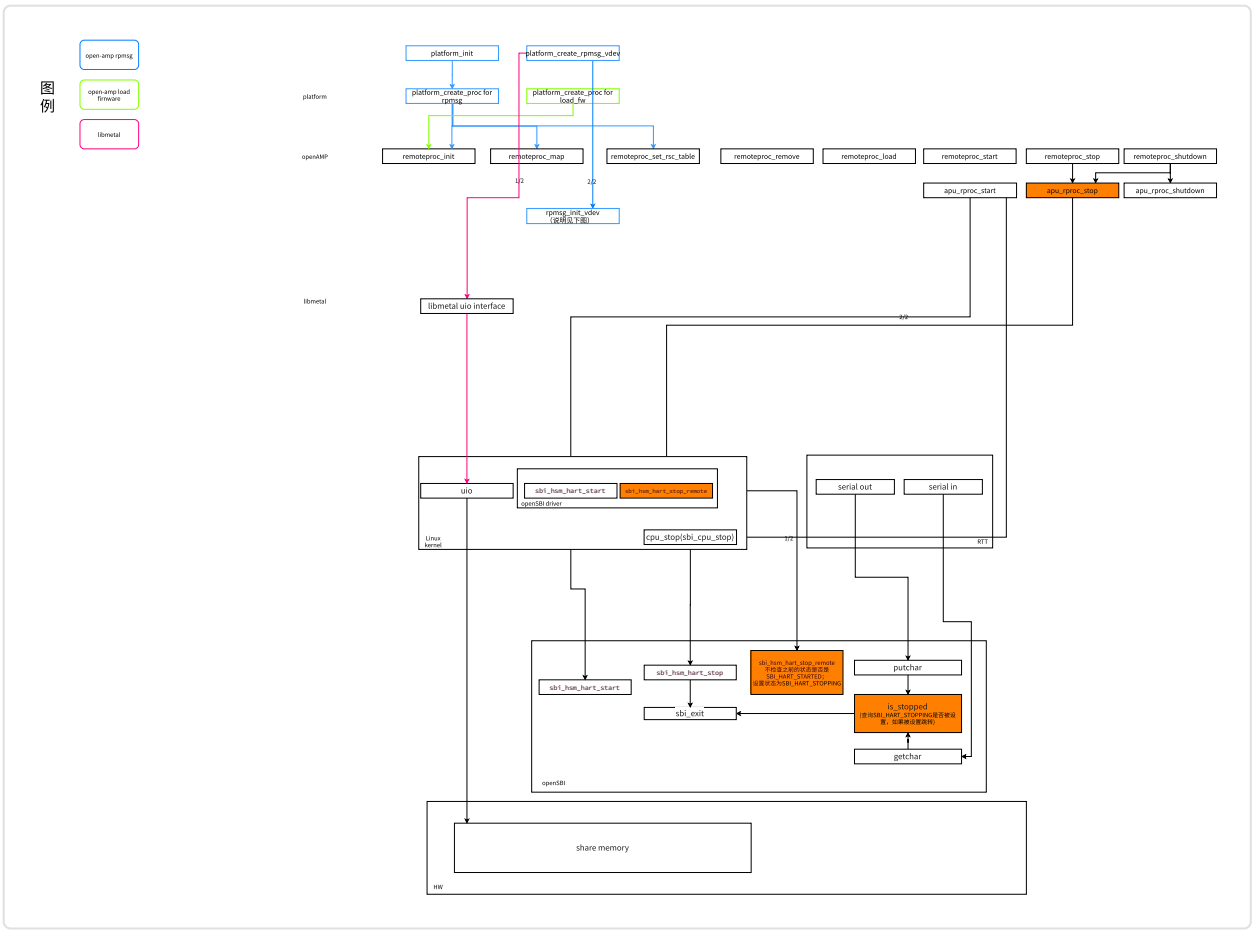
OpenAMP的目录结构

openamp目录结构

	A	B	C	D
1	apps	示例代码，测试代码，rpmsg和remoteroc对具体硬件（对于RTOS和bare metal）和操作系统（对于Linux）相关的移植代码		
2		examples	示例代码	
3			load_fw	通过remoteproc给另一个处理器加载固件的例子
4			echo	rpmsg ping-pong测试，数据原样返回。 rpmsg-ping.c消息发送；rpmsg-echo.c消息回复。 和rpmsg_sample_echo的区别是什么?
5			matrix_multiply	rpmsg矩阵乘法。本地处理器把自己的计算结果与远端处理器比较，如果一致，测试通过。每次通信计算矩阵乘法中的一步（[i,j]位置的结果等于[i,k]乘以[k,j]）
6			rpc_demo	应该是proxy，但是没有看懂。
7		machine		basemetal系统移植层
8		system		操作系统移植层
9		tests		rpmsg测试
10	cmake	项目相关的cmake文件，例如依赖库的处理，平台相关部分。		
11	docs			文档
12	lib	include	openamp	openamp头文件
13		proxy	作为远端处理器proxy，实现open, read, write和close四个接口	
14		remoteproc	remoteproc公共函数和API	
15		rpmsg	rpmsg公共函数和API	
16		virtio	基于virtio的virtqueue的实现	
17	scripts	辅助脚本		

OpenAMP机制简述

整体设计见图一



具体设计可以参见open-amp/docs/remoteproc-design.md和open-amp/docs/rpmsg-design.md。

编译

下载代码

`git clone --recurse-submodules git@github.com:bjzhang/rtthread-openamp-project-sources.git`

准备rt-thread包

目前修改还没有提交，packages部分提交后，仅仅需要保证`~/.env/packages/packages/`是最新的。

`pushd port_rtt/rt-thread-src/bsp/riscv64-virt`

`scons --menuconfig`直接退出，不需要保存

`pushd ~/.env/packages/packages/`

`git remote add bjzhang https://github.com/bjzhang/packages.git`

`git fetch bjzhang`

`git checkout -f bjzhang/rv64-openamp-port`

`popd; popd`

生成补丁

生成buildroot编译时所需的补丁。buildroot允许用户添加自己的补丁，补丁的目录由`BR2_GLOBAL_PATCH_DIR`描述，相详见：`../buildroot/buildroot-frag-config`”。

`./scripts/generating_opensbi.sh`

`./scripts/generating_qemu.sh`

编译

下述过程编译buildroot所有的包，使用buildroot的post image阶段调用脚本编译rt-thread及其libmetal和open-amp两个包。`rt-thread.elf`会直接放到rootfs中，不需要用户拷贝。

`cd buildroot-src`

`make BR2_EXTERNAL=../ qemu_riscv64_virt_defconfig`

使用Linux kernel的merge_config.sh合并config

`../scripts/merge_config.sh .config ../buildroot/buildroot-frag-config`

`make -j<nrcpu>`

编译结束，会自动把二进制复制到demo_bin。

测试

启动Linux，回到top dir，执行`./scripts/boot.sh`启动Linux。在另一个终端，执行`ssh -p 2222 root@localhost`登陆。

buildroot的root密码是Linux，已经设置ssh允许root登陆。

插入openamp驱动。

`insmod /lib/modules/5.10.7/extra/riscv_openamp.ko`

启动rt-thread

`load_fw-shared rtthread.elf 1`

Plain Text

```
# load_fw-shared rtthread.elf 1
Loading Exectuable Demo
metal: debug:     added page size 4096 @/tmp
metal: debug:     registered platform bus
```

```
apu_rproc_init: node id: 1
metal: debug:     opened sysfs device platform:90200000.rproc
metal: debug:     opened platform:90200000.rproc as /dev/uio0
metal: info:      metal_uio_dev_open: No IRQ for device 90200000.rproc.
Successfully open rproc device.
Successfully added rproc shared memory
Start to load executable with remoteproc_load()
metal: debug:     remoteproc_load: check remoteproc status
metal: debug:     remoteproc_load: open executable image
file size 1721872
1721872 read
metal: debug:     remoteproc_load: check loader
metal: debug:     remoteproc_load: loading headers
metal: debug:     Loading ELF headering
metal: debug:     Loading ELF program header.
metal: debug:     Loading ELF section header.
metal: debug:     Loading ELF section header complete.
metal: debug:     Loading ELF shstrtab.
metal: debug:     remoteproc_load, load header 0x0, 0x1a4610, next 0x3fee9b6460, 0x0
mem_image_load: offset=0x48c00, size=0x100
metal: debug:     remoteproc_load: load executable data
metal: debug:     segment: 1, total segs 1
metal: debug:     load data: da 0x90200000, offset 0x1000, len = 0x47d00, memsize = 0x4dac8, state 0x20801
mem_image_load: offset=0x1000, size=0x47d00
Copy to 0x3fde23f000 (pa: 90200000) with size 0x47d00
metal: debug:     load data: da 0xffffffffffffffff, offset 0x0, len = 0x0, memsize = 0x0, state 0x40801
metal: debug:     remoteproc_load, update resource table
metal: debug:     remoteproc_load: successfully load firmware
ERROR: write cpu hotplug attribute /sys/devices/system/cpu/cpu1/online as 0 failed, Device or resource busy
Set hartid to 1.
Set paddr to 0x90200000.
Set state to start.
successfully started the processor
waiting for remote proc shutdown request
```

测试libmetal share memory

再打开一个终端登录（`ssh -p 2222 root@localhost`）

rt-thread侧运行：`msh />shmem_demod`。Linux侧执行`libmetal_amp_demo-share`。

Plain Text

```
msh />shmem_demod

CLIENT> Configuration share memory

CLIENT> Wait for shared memory demo to start.

CLIENT> Demo has started.

CLIENT> Shared memory test finished
```

Plain Text

```
# libmetal_amp_demo-share
CLIENT> ****** libmetal demo: shared memory ******
metal: info:      metal_uio_dev_open: No IRQ for device 90000000.shm.
CLIENT> Setting up shared memory demo.
CLIENT> Starting shared memory demo.
CLIENT> Sending message: Hello World - libmetal shared memory demo
CLIENT> Message Received: Hello World - libmetal shared memory demo
CLIENT> Shared memory demo: Passed.
```

rpmsg: rpmsg echo

rt-thread执行`rpmsg_echo`，Linux执行`rpmsg-sample-ping-shared`

Plain Text

```
msh />rpmsg_echo
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.
echo message number 1: hello world!
echo message number 2: hello world!
echo message number 3: hello world!
echo message number 4: hello world!
echo message number 5: hello world!
```

```
echo message number 6: hello world!
echo message number 7: hello world!
echo message number 8: hello world!
echo message number 9: hello world!
echo message number 10: hello world!
echo message number 11: hello world!
echo message number 12: hello world!
echo message number 13: hello world!
echo message number 14: hello world!
echo message number 15: hello world!
echo message number 16: hello world!
echo message number 17: hello world!
echo message number 18: hello world!
echo message number 19: hello world!
echo message number 20: hello world!
echo message number 21: hello world!
echo message number 22: hello world!
echo message number 23: hello world!
echo message number 24: hello world!
echo message number 25: hello world!
echo message number 26: hello world!
echo message number 27: hello world!
echo message number 28: hello world!
echo message number 29: hello world!
echo message number 30: hello world!
echo message number 31: hello world!
echo message number 32: hello world!
echo message number 33: hello world!
echo message number 34: hello world!
echo message number 35: hello world!
echo message number 36: hello world!
echo message number 37: hello world!
echo message number 38: hello world!
echo message number 39: hello world!
echo message number 40: hello world!
echo message number 41: hello world!
echo message number 42: hello world!
echo message number 43: hello world!
echo message number 44: hello world!
echo message number 45: hello world!
echo message number 46: hello world!
echo message number 47: hello world!
echo message number 48: hello world!
echo message number 49: hello world!
echo message number 50: hello world!
```

```
echo message number 51: hello world!
echo message number 52: hello world!
echo message number 53: hello world!
echo message number 54: hello world!
echo message number 55: hello world!
echo message number 56: hello world!
echo message number 57: hello world!
echo message number 58: hello world!
echo message number 59: hello world!
echo message number 60: hello world!
echo message number 61: hello world!
echo message number 62: hello world!
echo message number 63: hello world!
echo message number 64: hello world!
echo message number 65: hello world!
echo message number 66: hello world!
echo message number 67: hello world!
echo message number 68: hello world!
echo message number 69: hello world!
echo message number 70: hello world!
echo message number 71: hello world!
echo message number 72: hello world!
echo message number 73: hello world!
echo message number 74: hello world!
echo message number 75: hello world!
echo message number 76: hello world!
echo message number 77: hello world!
echo message number 78: hello world!
echo message number 79: hello world!
echo message number 80: hello world!
echo message number 81: hello world!
echo message number 82: hello world!
echo message number 83: hello world!
echo message number 84: hello world!
echo message number 85: hello world!
echo message number 86: hello world!
echo message number 87: hello world!
echo message number 88: hello world!
echo message number 89: hello world!
echo message number 90: hello world!
echo message number 91: hello world!
echo message number 92: hello world!
echo message number 93: hello world!
echo message number 94: hello world!
```

```
104 echo message number 95: hello world!
105 echo message number 96: hello world!
106 echo message number 97: hello world!
107 echo message number 98: hello world!
108 echo message number 99: hello world!
109 echo message number 100: goodbye!
110 reach message limit, exit
111 Stopping application...
```

Plain Text

```
1  ...
2  received message 98: hello world! of size 12
3   seed hello world!:
4   rnum 97:
5  rpmsg sample test: message 99 sent
6  received message 99: hello world! of size 12
7   seed hello world!:
8   rnum 98:
9  rpmsg sample test: message 100 sent
10 received message 100: goodbye! of size 8
11  seed goodbye!:
12  rnum 99:
13 echo test: service is destroyed
14 **********************************
15  Test Results: Error count = 0
16 **********************************
17 Quitting application .. rpmsg sample test end
18 Stopping application...
```

rpmsg: msg ping

rt-thread执行`rpmsg_update`，Linux执行`msg-test-rpmsg-ping-shared`

Plain Text

```
msh />rpmsg_update
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.
unexpected Remote endpoint destroy
Stopping application...
```

Plain Text

```
...
echo test: sent : 495
 received payload number 478 of size 495
sending payload number 479 of size 496
echo test: sent : 496
 received payload number 479 of size 496
**********************************
 Test Results: Error count = 0
**********************************
Quitting application .. Echo test end
Stopping application...
```

rpmsg: matrix_multiply

rt-thread执行`matrix_multiplyd`，Linux执行`matrix_multiply-shared`

Erlang

```
msh />matrix_multiplyd
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Waiting for events...
ERROR: Endpoint is destroyed
Stopping application...
msh />
```

Plain Text

```
...
CLIENT> Input matrix 1

 1  8  5  1  9  4
 0  8  3  2  4  2
 2  2  4  8  8  1
 3  6  2  4  8  2
 0  8  5  7  8  8
 3  0  6  0  3  7
CLIENT> Matrix multiply: sent : 296
CLIENT> **********************************
CLIENT>  Test Results: Error count = 0
CLIENT> **********************************
CLIENT> Quitting application .. Matrix multiplication end
CLIENT> Stopping application...
```

测试问题解决

buildroot可以`make xxx-rebuild`（针对guest里面的包，例如opensbi，linux kernel等）或`make host-xxx-rebuild`（针对host上直接运行的包，例如qemu） 重新编译。如果是修改了补丁或需要重新configure或其它原因，可以用dirclean，例如`make xxx-dirclean`或`make host-xxx-dirclean`。

OpenAMP移植

说明和本移植强相关的内容。

libmetal移植

metal_init

metal_init会初始化log handler和level；初始化总线，共享内存和设备的链表；最后调用操作系统的`metal_sys_init`。对于rt-thread，后者是注册总线，其中包括如何管理该总线的设备。例如打开设备最后会调用到rt-thread实现的`metal_generic_dev_sys_open`。对于Linux有通用的实现，核心是围绕uio设备的初始化，uio支持platform总线和pci总线。

Mutex

`libmetal/lib/system/rtthread/mutex.h`

使用了rt-thread的mutex接口，rtt的mutex需要唯一名字，使用指针地址作为名字避免重名。`rt_mutex_init`实际不会失败，所以不在`metal_mutex_init`中检查起返回值。

测试用例

原有linux下面只有两个平台有share memory测试，笔者复制到了`libmetal/examples/rv64_virt/generic`，修改了share memor地址（`SHM_DEV_NAME`），删除了不需要的用例（主要是由于暂时不支持IPI），并添加了相应了`CMakelists.txt`，具体的用例shmem_demo.c没有修改。

openamp移植

内存分配

系统内存划分

share memory

区域划分：open-amp/apps/system/linux/machine/rv64_virt/platform_info.h

分为两部分：两侧的share memory；非IPI下，用于查询的share memory。

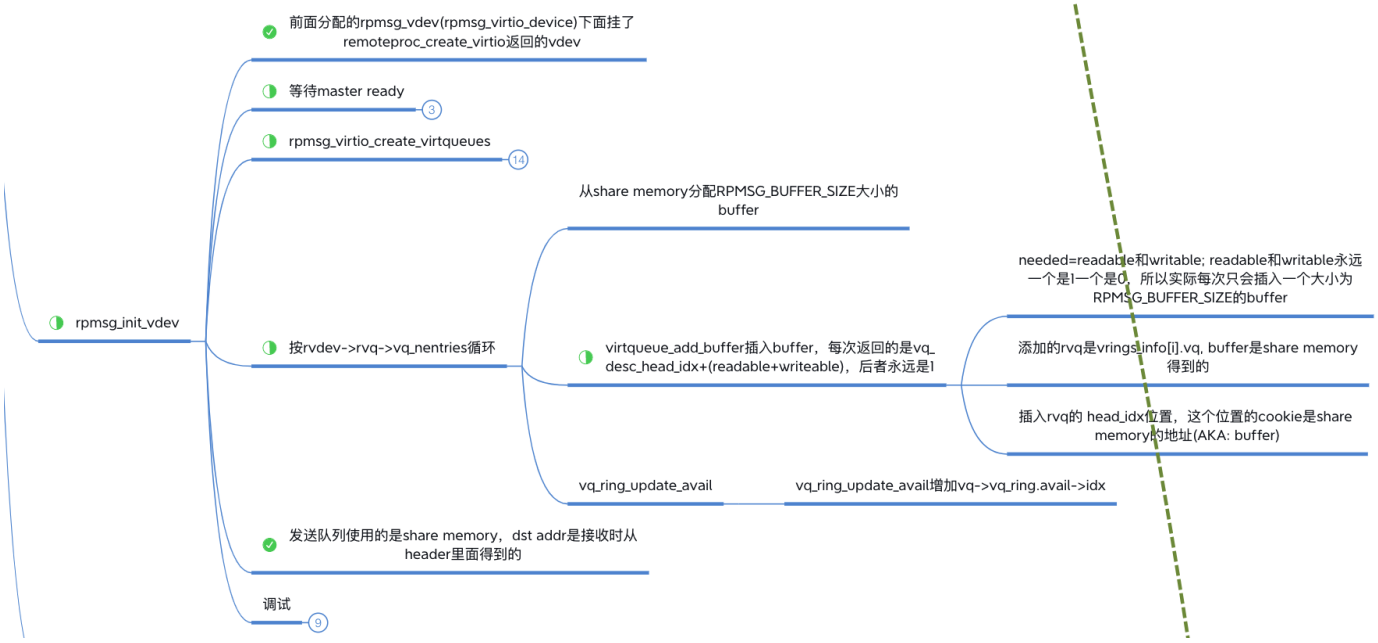
宏定义	含义	相关函数	作用
SHM_BASE SHM_DEV_NAME	Share memory起始地址 share memory对应的uio设备名称	remoteproc_init	共享内存的起始地址。在Linux侧远端处理器初始化时传入，用于打开uio设备。在rtt侧，由于确定buffer可用，不需要专门打开共享内存。后续直接mmap share memory中具体的区域

RSC_MEM_PA	resource table物理地址	platform_create_proc	从et_resource_table获得resource table，并保存到RSC_MEM_PA对应的虚拟地址。看起来Linux和rtt两侧都有resource table，但是rtt侧的resource table的RING_TX和RING_RX是FW_RSC_U32_ADDR_ANY(32位-1)，所以实际上resource table中可配置的地址部分，host传给device的。
SHARED_BUF_PA	共享内存地址	platform_create_rpmsg_vdev	Host基于这个地址初始化通信共享内存；
RING_TX	发送队列	定义在resource table中	
RING_RX	接收队列	定义在resource table中	

OS的内存分配

“libmetal/lib/system/linux/alloc.h”

rpmsg_init_vdev的详细含义见图二



Platform层

platform层利用remoteproc，rpmsg和libmetal API抽象平台相关的行为，由测试用例调用。如果不需要跑open-amp的测试用例，可以不做这一层。业务直接调用上述三者的代码，只是这样业务代码可能会比较冗余，不推荐。

platform_init：包括libmetal的初始化，硬件系统（目前是中断；由于qemu的限制rv64移植暂时不支持中断）和remoteproc的初始化，前者直接调用libmetal的初始化，后者包块resource table和share memory的初始化。

platform_create_rpmsg_vdev：初始化rpmsg virtio device，把share memory buffer填入virtio ring buffer的vring available和used队列。

platform_poll：查询对端消息。

platform_release_rpmsg_vdev：释放rpmsg virtio device。

platform_cleanup：`platform_init`的反函数，移除remoteproc，libmetal和硬件系统（目前是中断）的退出。

remoteproc移植

Load firmware添加了使用ctrl+c停止远端处理器的功能，并且支持从文件中加载rt-thread的镜像。比较大的修改是添加了rv64 virt的远端处理器管理。代码见`apps/examples/load_fw/virt_rv64_rproc_example.c`。处理器代码中与其它平台不同的是，添加了openamp驱动，通过sysfs访问驱动并调用opensbi api启动和停止远端处理器。启动远端处理器使用了opensbi已有的API。停止远端处理器没有现成的API，本次移植中手工添加了API，后续会与社区讨论，能否支持这个新的API。该API编号是`#define SBI_EXT_HSM_HART_STOP_REMOTE 0x1001`，和之前标准的编号有很大距离，基本可以避免冲突。这部分的框图见图一的橙色部分。

remoteproc_mmap

根据`remoteproc_get_mem`获得已有映射，如果没有映射使用远端处理器的mmap函数映射。前者从远端处理器`rproc`的

mems链表查找有无符合条件的memory区域，传入的物理地址和设备地址通常只有一个有效。例如`platform_create_proc`中仅仅传入物理地址；而`remoteproc_load`族函数仅仅插入设备地址。

对于需要映射的情况， 对于Linux，memory区域实际由device tree传入的uio决定，所以仅仅做检查，返回虚拟地址；>对于rt-thread，不需要做专门的管理，所以注册该区域，并返回虚拟地址即可。

已知问题

- 由于qemu rv64 virt不支持IPI，所以open-amp不支持中断方式。
- qemu中固定传给Linux 256M memory，把后面内存留给rt-thread。