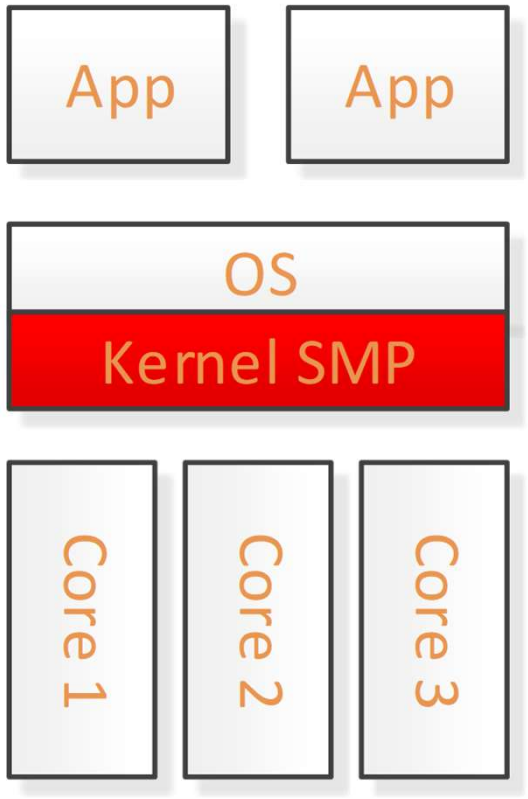
OpenAMP

Xiang Xiao
2020.3.6

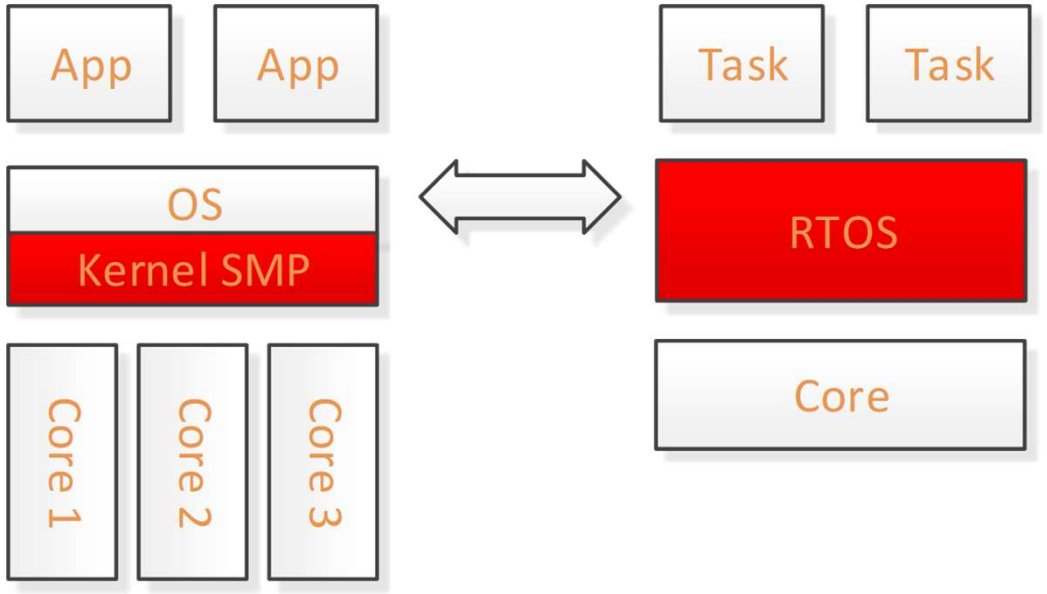
SMP vs AMP(1)

- SMP on homogeneous architectures:
  - Single OS controlling two or more identical cores sharing system resources
  - Dynamic scheduling and load balancing



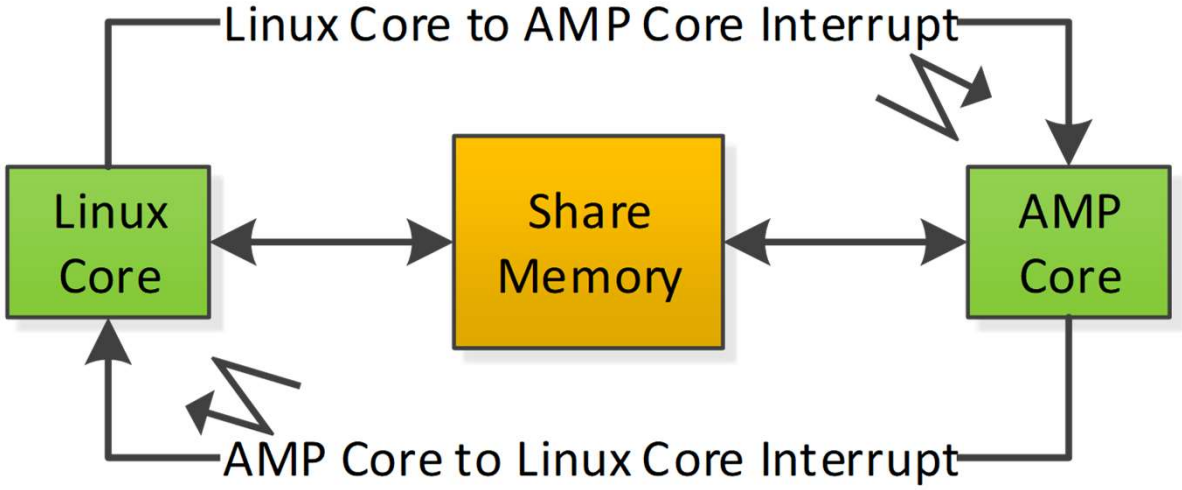
SMP vs AMP(2)

- AMP on heterogeneous architectures:
  - Different OS on each core --> full-featured OS alongside a real-time kernel
  - Inter processor communication protocol
  - Efficient when the application can be statically partitioned across cores - high performance is achieved locally



SMP vs AMP(3)

- How does it work?
  - There is a concept of master and slave
  - Master manages shared memory
  - Master may control slave's life-cycle

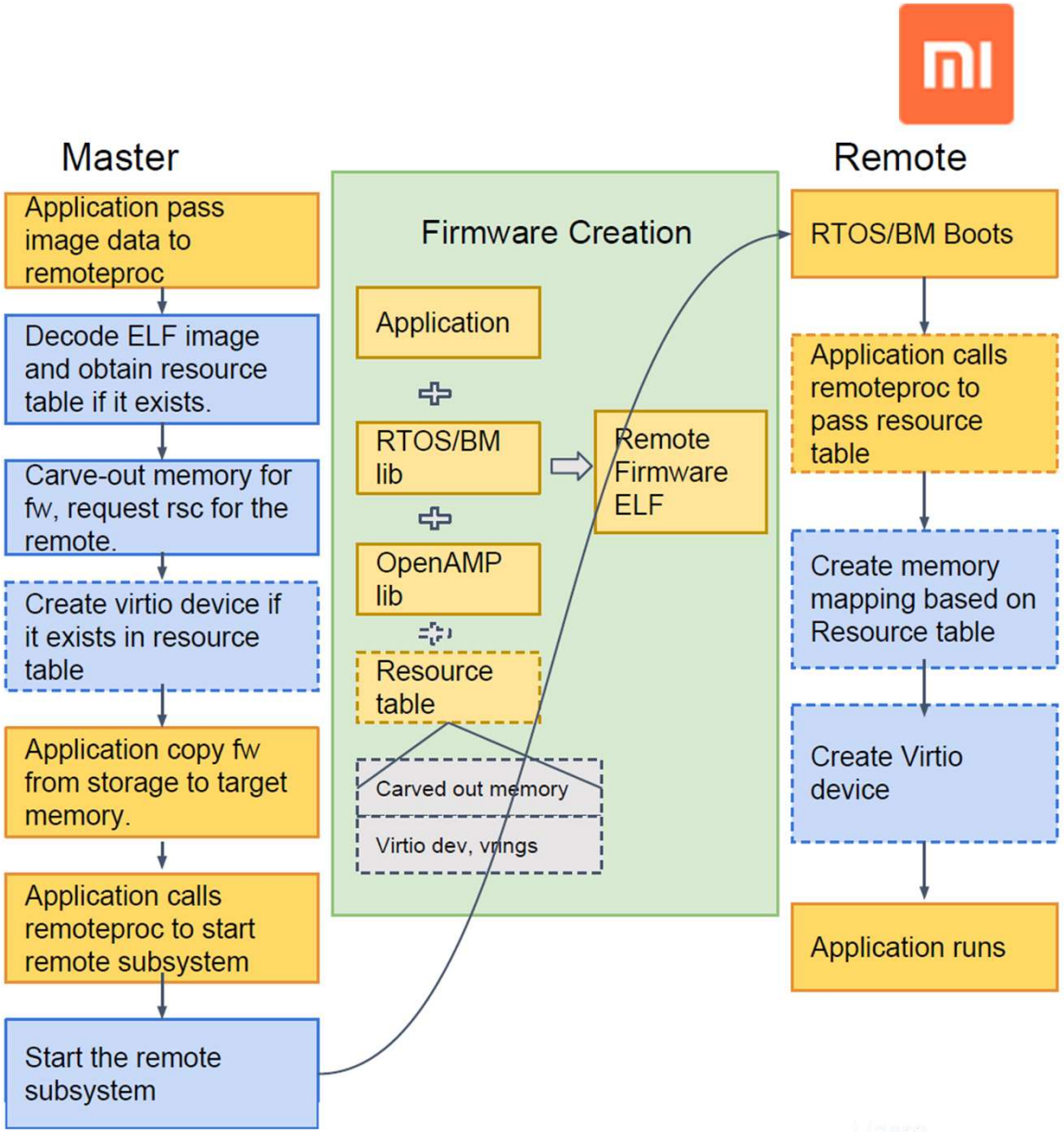


Remoteproc(1)

- Provides user APIs to do life cycle management of the remote system and manage the resources of the remote system.
  - load remote system image
  - setup resources for the remote system
  - start the remote system
  - manage the resource of the remote system
  - suspend the remote system
  - restore the remote system
  - stop the remote system
  - release the resource of the remote system
  - shutdown the remote system and release its source

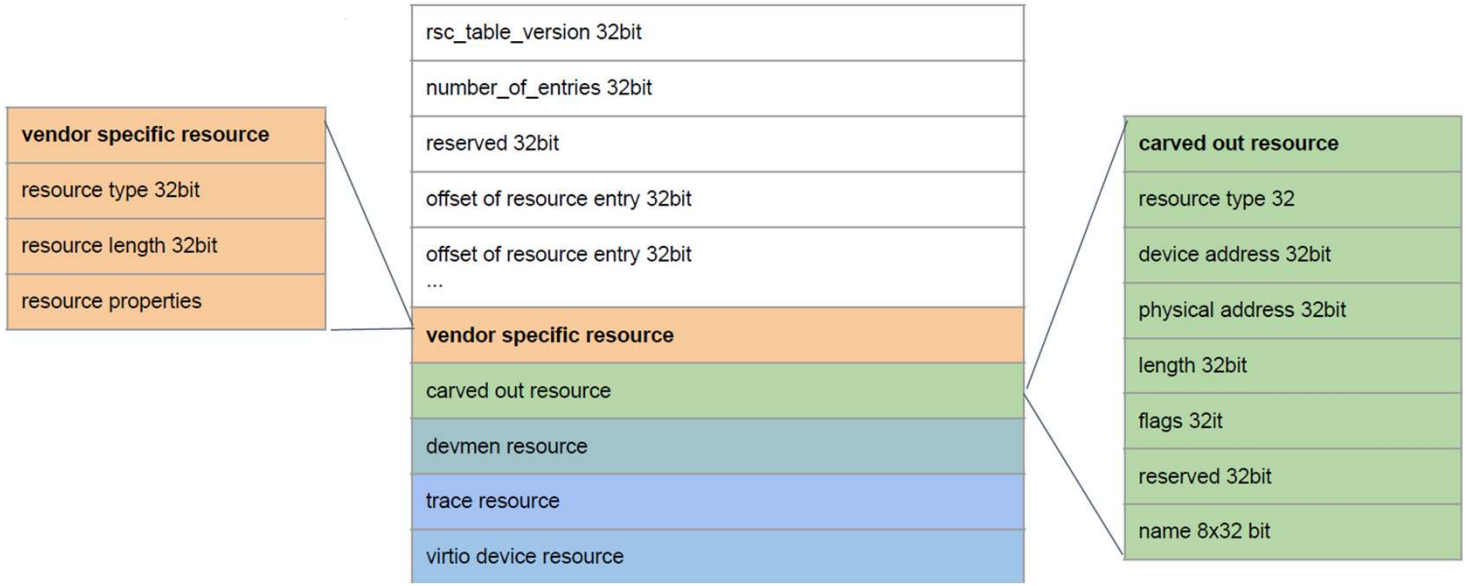
Remoteproc(2)

- Framework API
  - rproc_alloc(…, ops, firmware, …)
  - rproc_add()
  - rproc_del()
  - rproc_put()
- Driver callback
  - start()
  - stop()

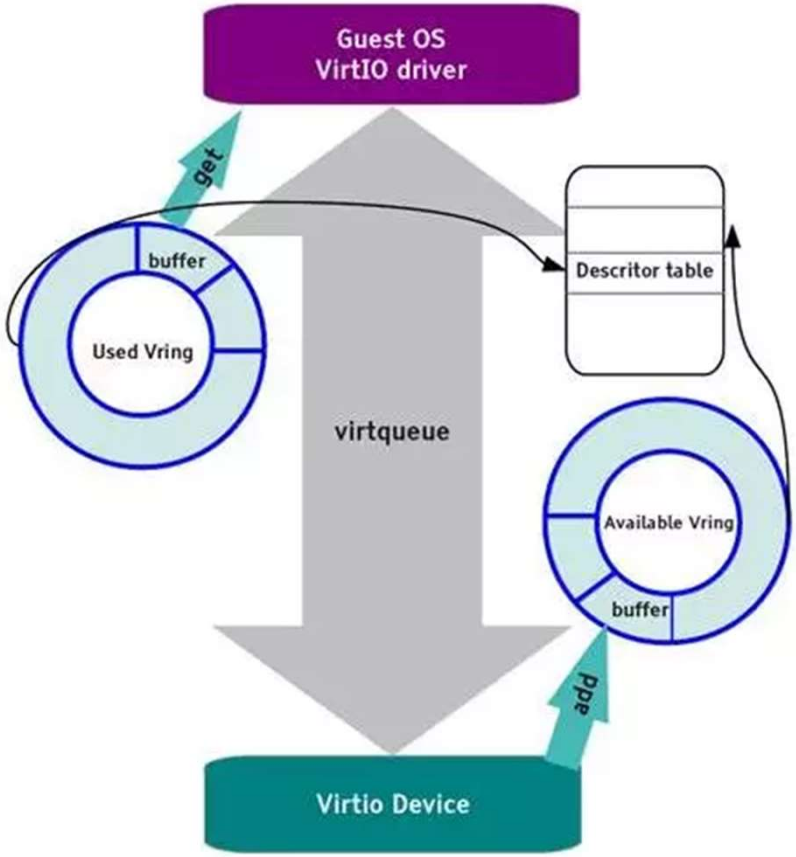


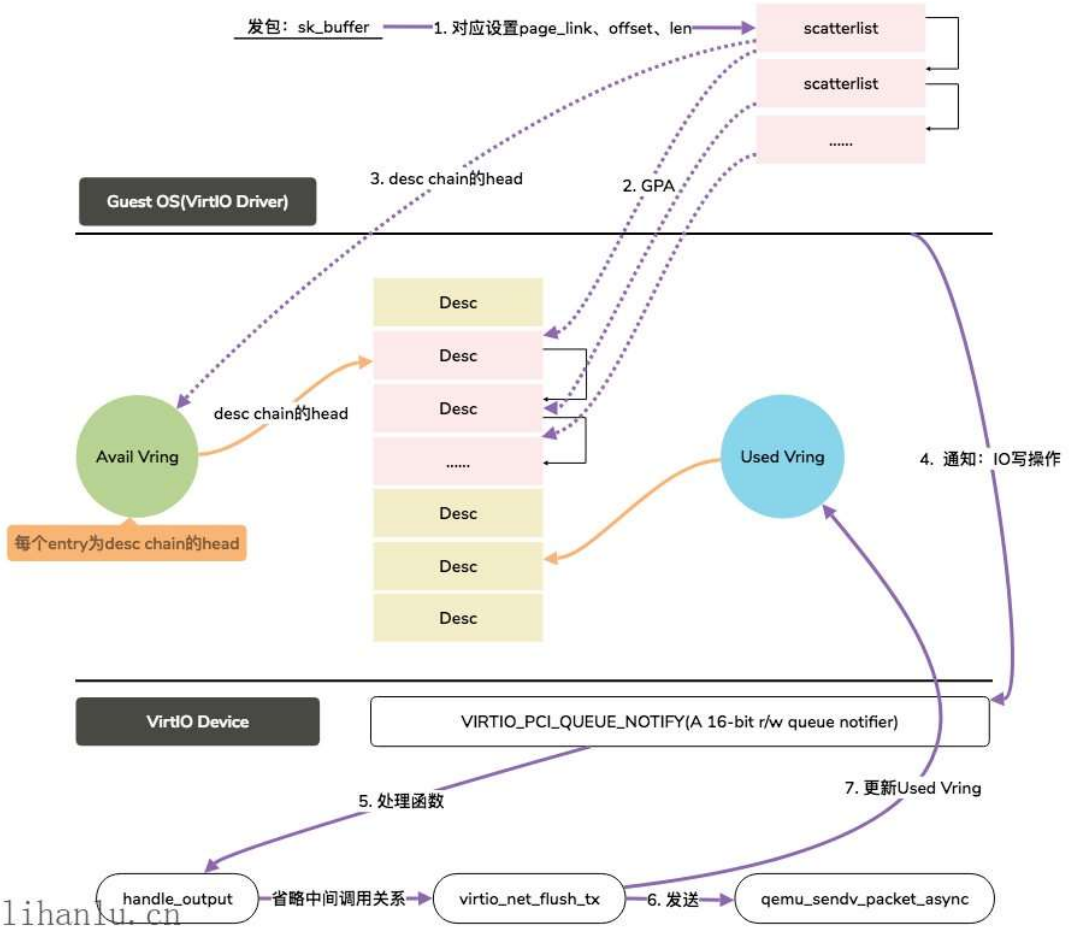
Remoteproc(3)

- A resource table is essentially a list of system resources required by the remote system. It may also include configuration entries. e.g. virtio configuration space. If needed, the remote firmware should contain this table as a dedicated ".resource_table" ELF section.

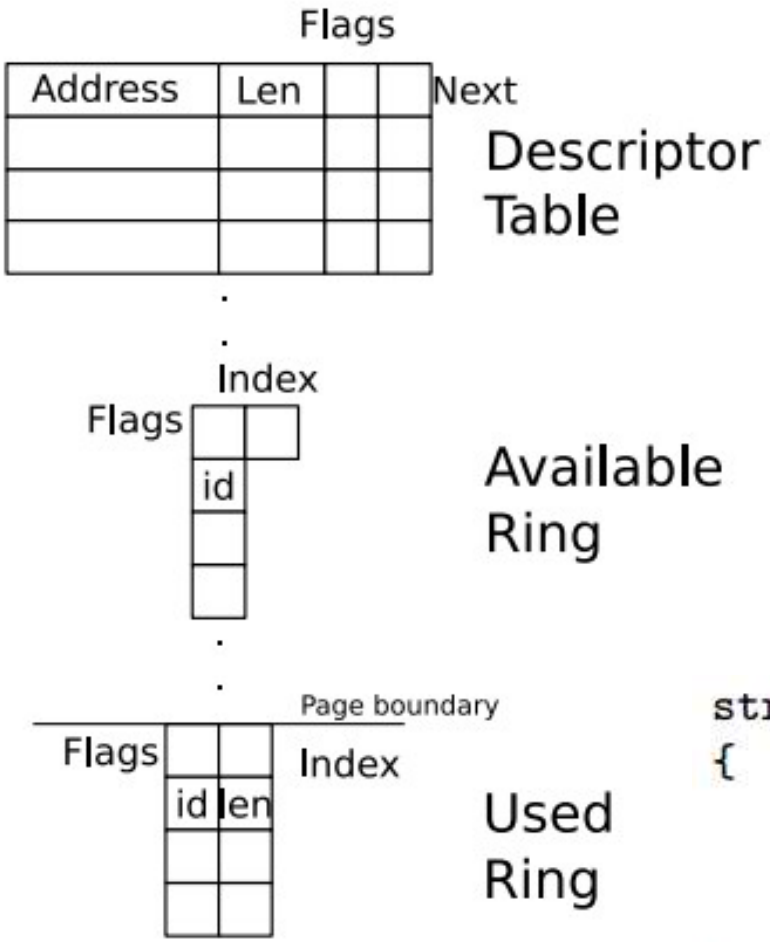


Virtio(1)





Virtio(2)

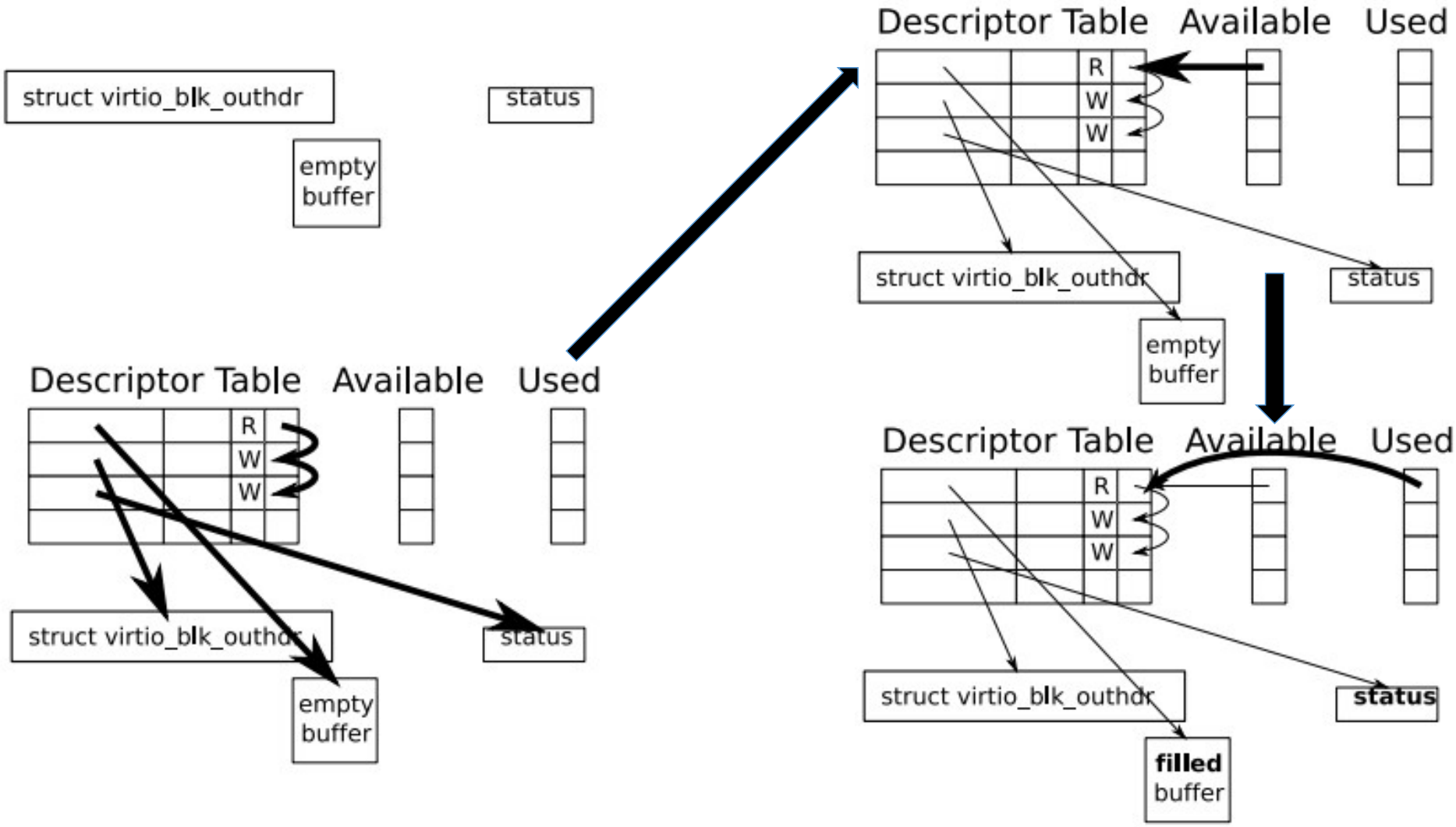



```
struct vring_desc
{
        __u64 addr;
        __u32 len;
        __u16 flags;
        __u16 next;
};
struct vring_avail
{
        __u16 flags;
        __u16 idx;
        __u16 ring[NUM];
};
```

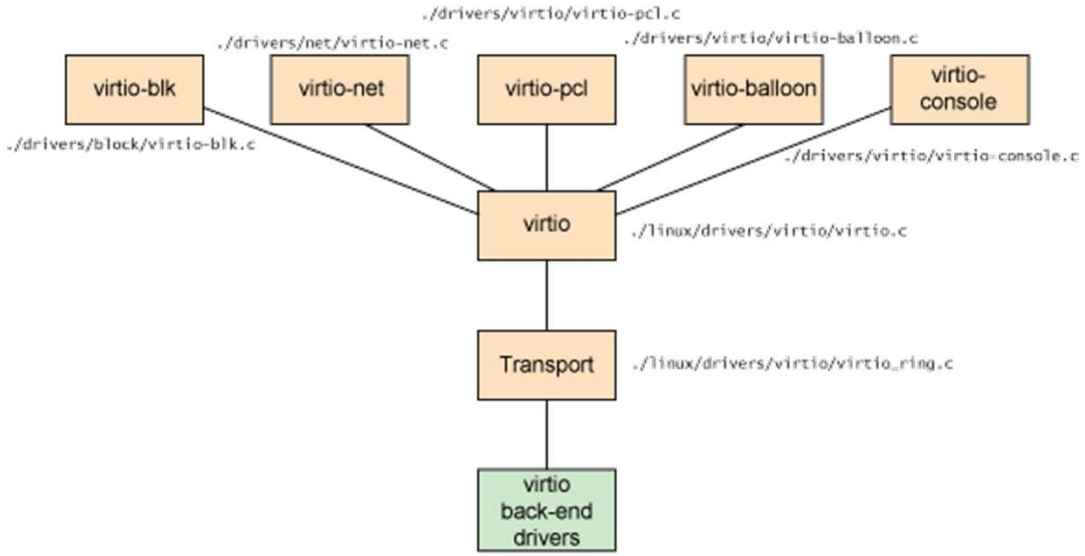
```
struct vring_used
{
        __u16 flags;
        __u16 idx;
        struct vring_used_elem ring[];
};
```

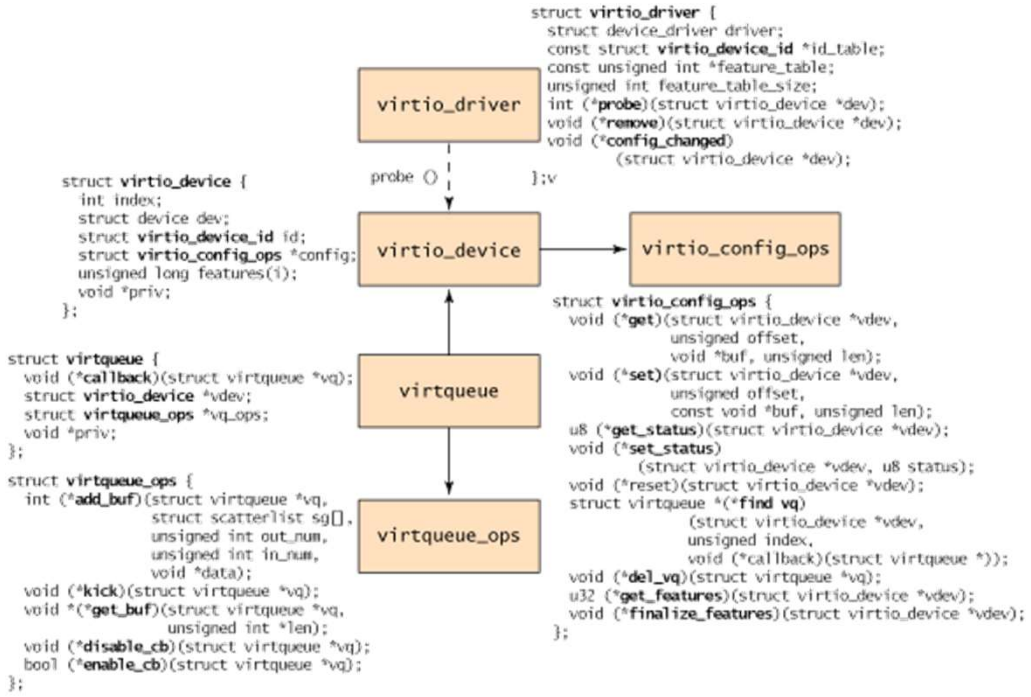
```
struct vring_used_elem
{
        __u32 id;
        __u32 len;
};
```

Virtio(3)

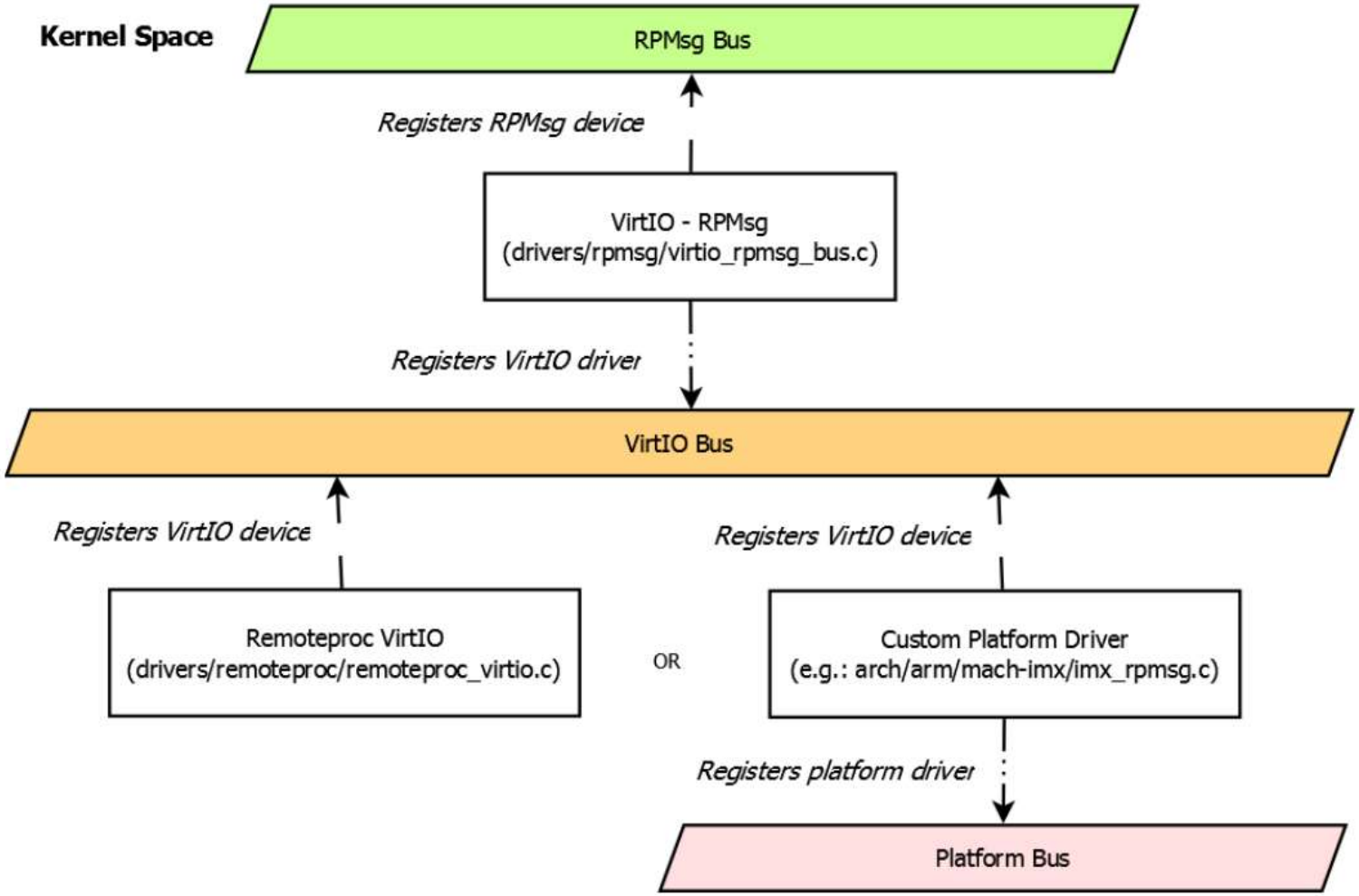

struct virtio_blk_outhdr
status
empty buffer
Descriptor Table
Available
Used
R
W
W
struct virtio_blk_outhdr
status
empty buffer
Descriptor Table
Available
Used
R
W
W
struct virtio_blk_outhdr
status
empty buffer
Descriptor Table
Available
Used
R
W
W
struct virtio_blk_outhdr
status
filled buffer

Virtio(4)

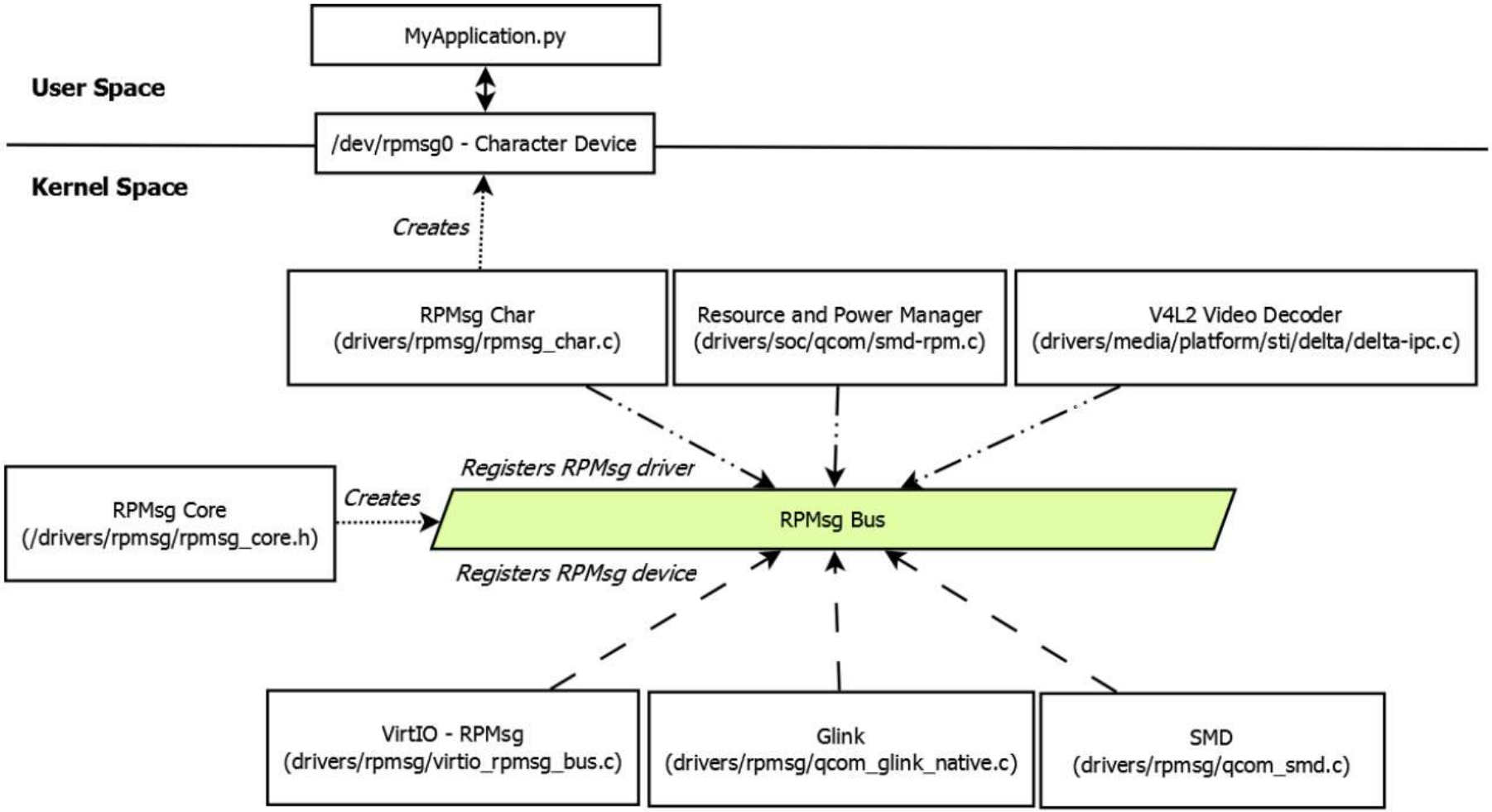

./drivers/virtio/virtio-pcl.c
./drivers/net/virtio-net.c
./drivers/virtio/virtio-balloon.c
virtio-blk
virtio-net
virtio-pcl
virtio-balloon
virtio-console
./drivers/block/virtio-blk.c
./drivers/virtio/virtio-console.c
virtio
./linux/drivers/virtio/virtio.c
Transport
./linux/drivers/virtio/virtio_ring.c
virtio back-end drivers


virtio_driver
probe ()
virtio_device
virtio_config_ops
virtqueue
virtqueue_ops
struct virtio_driver {
struct device_driver driver;
const struct virtio_device_id *id_table;
const unsigned int *feature_table;
unsigned int feature_table_size;
int (*probe)(struct virtio_device *dev);
void (*remove)(struct virtio_device *dev);
void (*config_changed)
(struct virtio_device *dev);
};
struct virtio_device {
int index;
struct device dev;
struct virtio_device_id id;
struct virtio_config_ops *config;
unsigned long features(i);
void *priv;
};
struct virtqueue {
void (*callback)(struct virtqueue *vq);
struct virtio_device *vdev;
struct virtqueue_ops *vq_ops;
void *priv;
};
struct virtqueue_ops {
int (*add_buf)(struct virtqueue *vq,
struct scatterlist sg[],
unsigned int out_num,
unsigned int in_num,
void *data);
void (*kick)(struct virtqueue *vq);
void *(*get_buf)(struct virtqueue *vq,
unsigned int *len);
void (*disable_cb)(struct virtqueue *vq);
bool (*enable_cb)(struct virtqueue *vq);
};
struct virtio_config_ops {
void (*get)(struct virtio_device *vdev,
unsigned offset,
void *buf, unsigned len);
void (*set)(struct virtio_device *vdev,
unsigned offset,
const void *buf, unsigned len);
u8 (*get_status)(struct virtio_device *vdev);
void (*set_status)
(struct virtio_device *vdev, u8 status);
void (*reset)(struct virtio_device *vdev);
struct virtqueue *(*find_vq)
(struct virtio_device *vdev,
unsigned index,
void (*callback)(struct virtqueue *));
void (*del_vq)(struct virtqueue *vq);
u32 (*get_features)(struct virtio_device *vdev);
void (*finalize_features)(struct virtio_device *vdev);
};

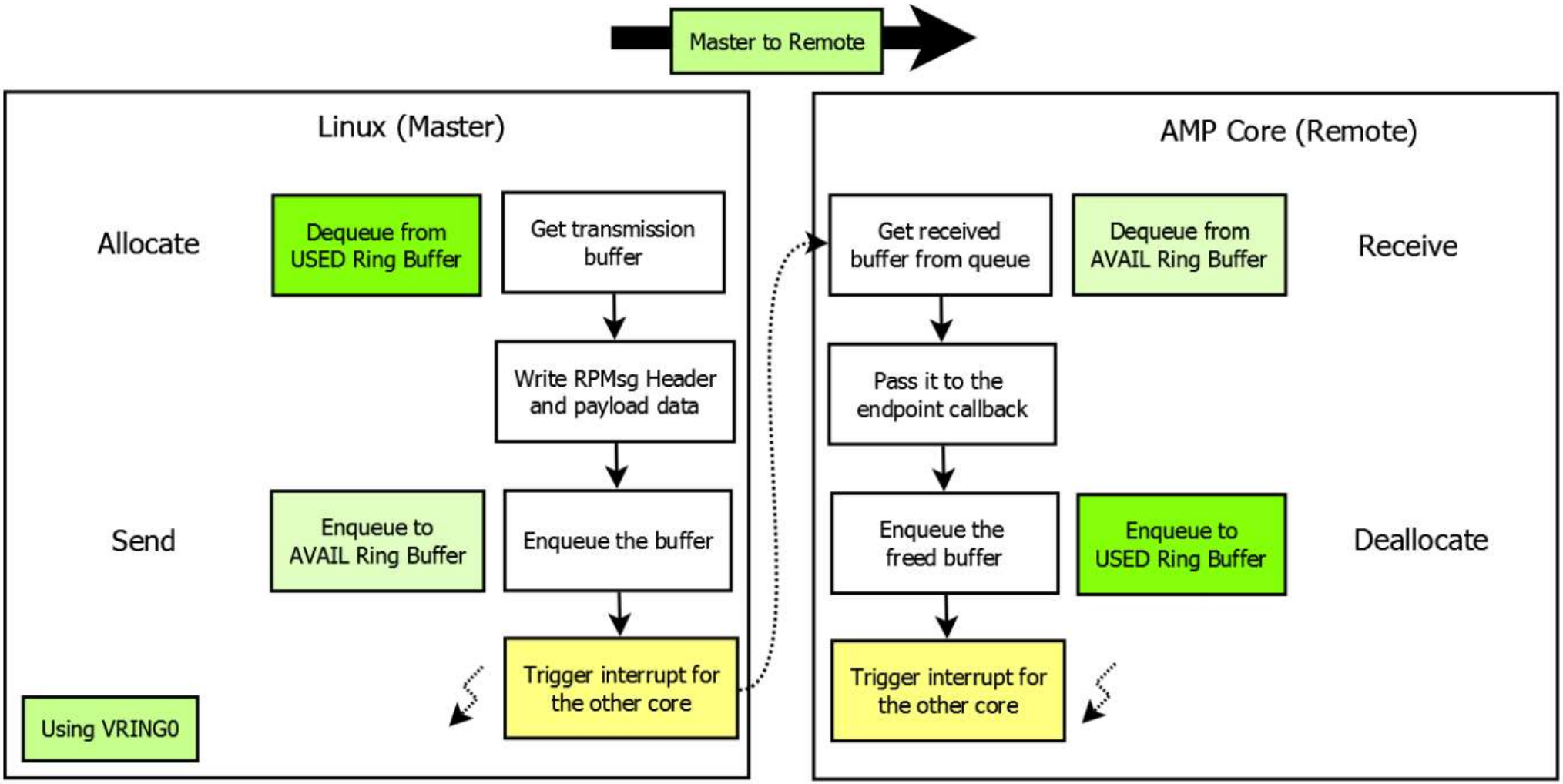
Rpmsg(1)


Kernel Space
RPMsg Bus
Registers RPMsg device
VirtIO - RPMsg
(drivers/rpmsg/virtio_rpmsg_bus.c)
Registers VirtIO driver
VirtIO Bus
Registers VirtIO device
Registers VirtIO device
Remoteproc VirtIO
(drivers/remoteproc/remoteproc_virtio.c)
OR
Custom Platform Driver
(e.g.: arch/arm/mach-imx/imx_rpmsg.c)
Registers platform driver
Platform Bus

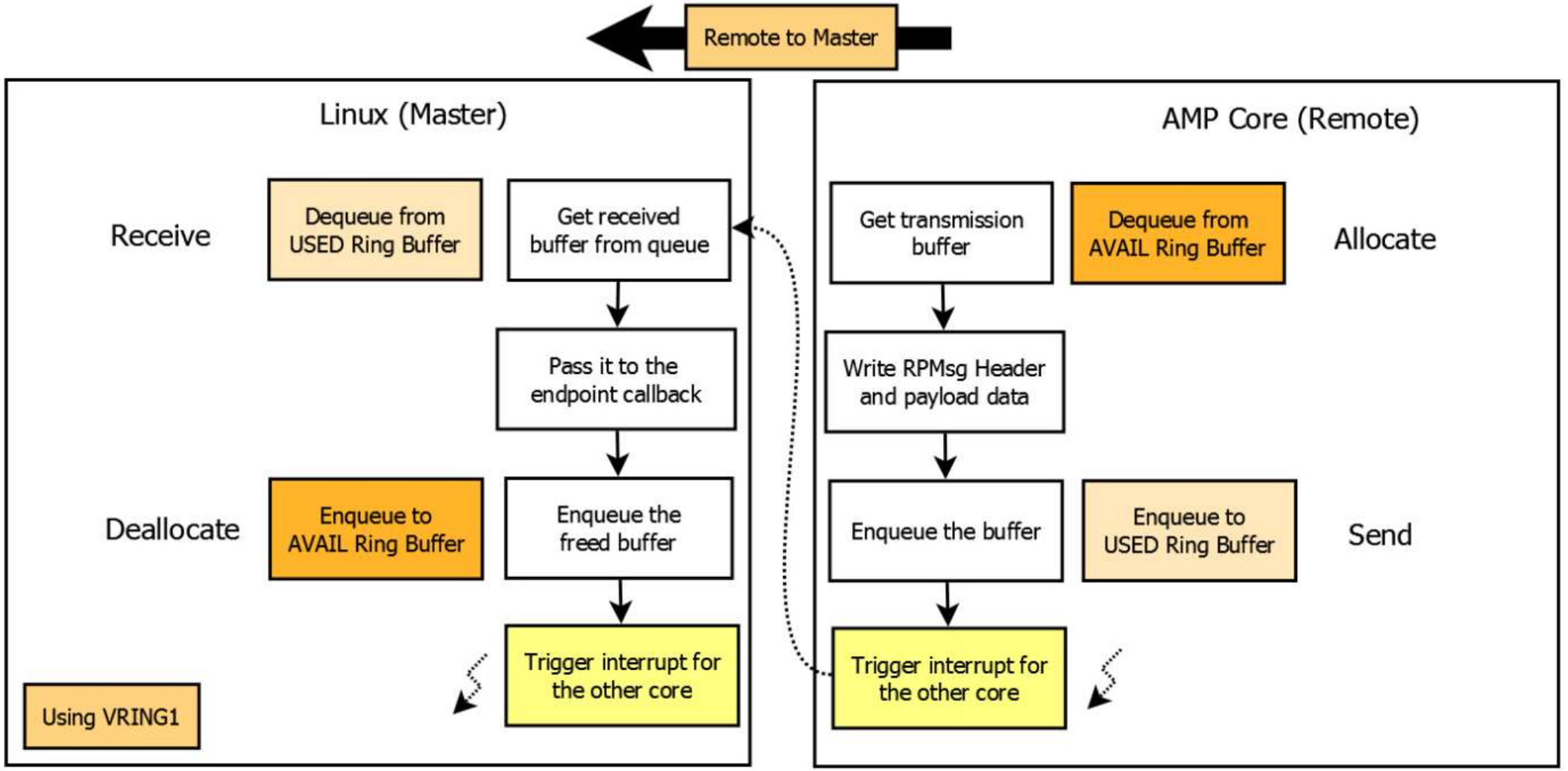
Rpmsg(2)


MyApplication.py
User Space
/dev/rpmsg0 - Character Device
Kernel Space
Creates
RPMsg Char
(drivers/rpmsg/rpmsg_char.c)
Resource and Power Manager
(drivers/soc/qcom/smd-rpm.c)
V4L2 Video Decoder
(drivers/media/platform/sti/delta/delta-ipc.c)
Registers RPMsg driver
RPMsg Core
(/drivers/rpmsg/rpmsg_core.h)
Creates
RPMsg Bus
Registers RPMsg device
VirtIO - RPMsg
(drivers/rpmsg/virtio_rpmsg_bus.c)
Glink
(drivers/rpmsg/qcom_glink_native.c)
SMD
(drivers/rpmsg/qcom_smd.c)

Rpmsg(3)

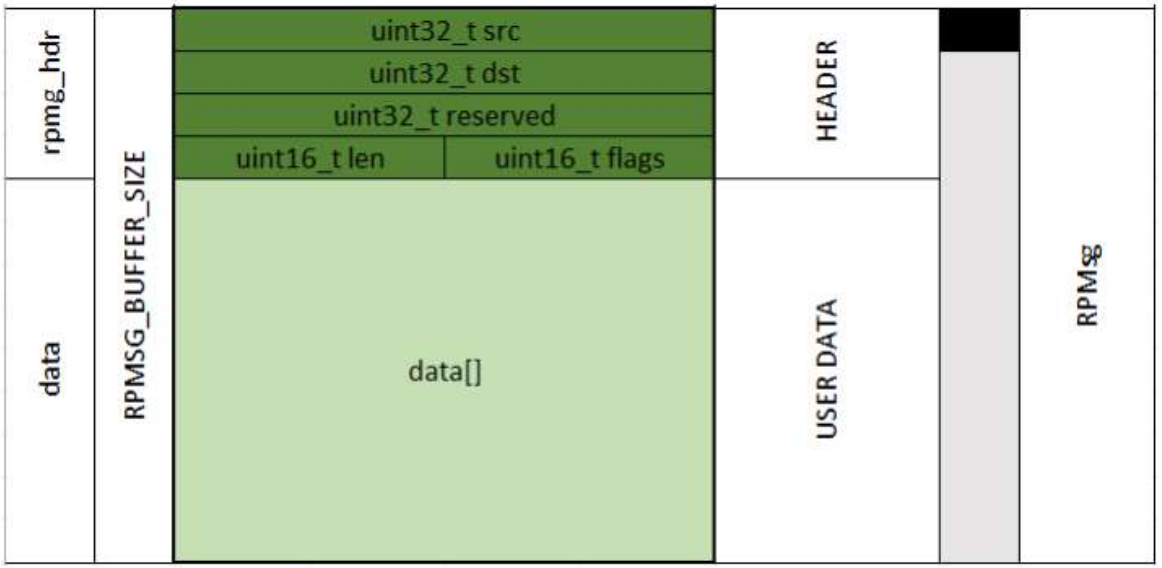

Master to Remote
Linux (Master)
AMP Core (Remote)
Allocate
Dequeue from USED Ring Buffer
Get transmission buffer
Write RPMsg Header and payload data
Send
Enqueue to AVAIL Ring Buffer
Enqueue the buffer
Trigger interrupt for the other core
Using VRING0
Get received buffer from queue
Dequeue from AVAIL Ring Buffer
Receive
Pass it to the endpoint callback
Enqueue the freed buffer
Enqueue to USED Ring Buffer
Deallocate
Trigger interrupt for the other core

Rpmsg(4)


Remote to Master
Linux (Master)
AMP Core (Remote)
Receive
Dequeue from USED Ring Buffer
Get received buffer from queue
Pass it to the endpoint callback
Deallocate
Enqueue to AVAIL Ring Buffer
Enqueue the freed buffer
Trigger interrupt for the other core
Using VRING1
Get transmission buffer
Dequeue from AVAIL Ring Buffer
Allocate
Write RPMsg Header and payload data
Enqueue the buffer
Enqueue to USED Ring Buffer
Send
Trigger interrupt for the other core

Rpmsg(5)

- Naming service
  - Convert the name to port number
  - Two phase handshake
  - Fix port number(53)




```
struct rpmsg_ns_msg {
        char name[RPMSG_NAME_SIZE];
        u32 addr;
        u32 flags;
} __packed;
```

```
enum rpmsg_ns_flags {
        RPMSG_NS_CREATE         = 0,
        RPMSG_NS_DESTROY        = 1,
        RPMSG_NS_BIND           = 2,
        RPMSG_NS_UNBIND         = 3,
};
```

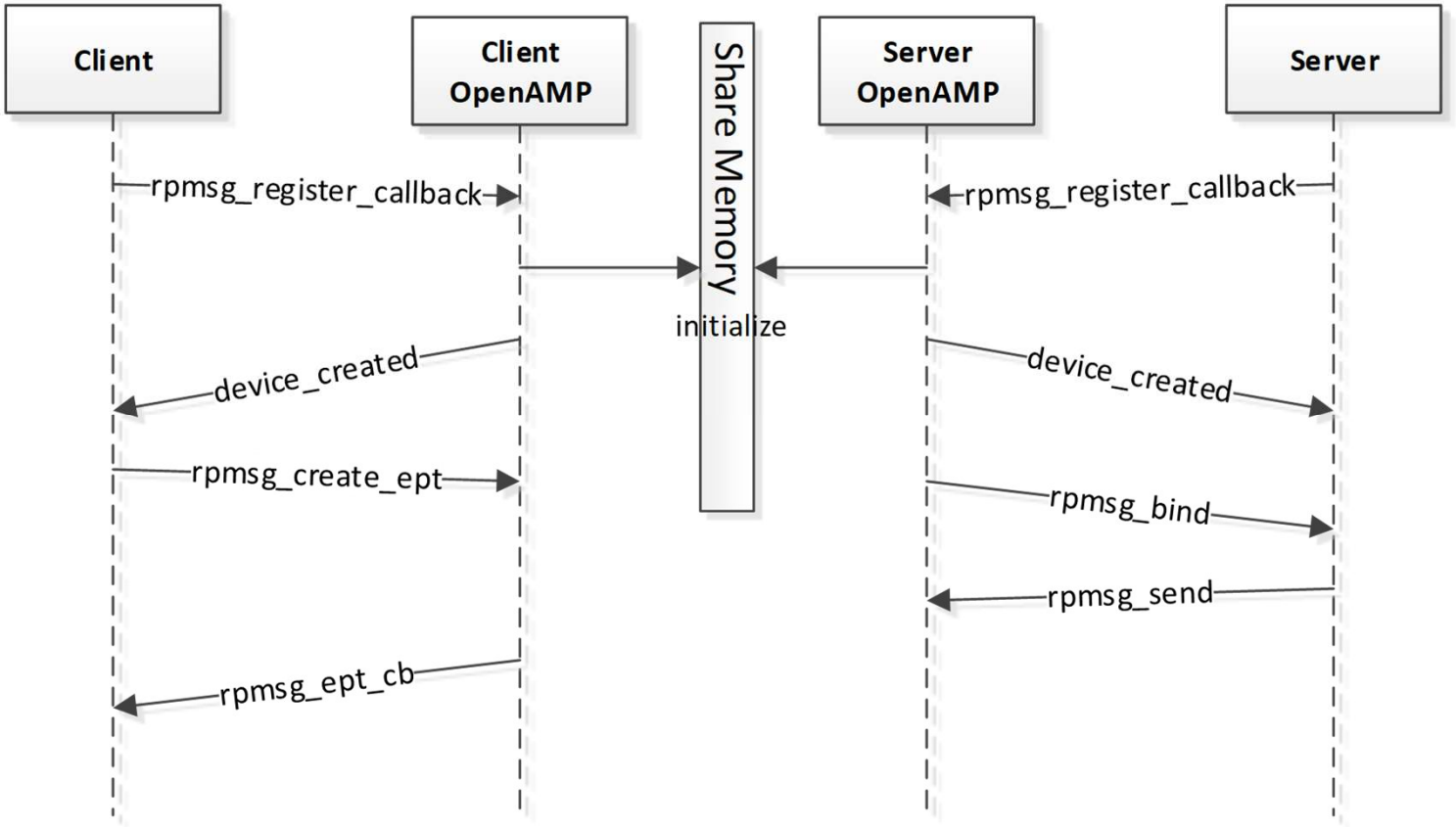
API(1)

- Initialize
  - rpmsg_register_callback
- The peer callback
  - device_created
  - device_destroyed
- Create endpoint
  - rpmsg_create_ept
- The endpoint callback
  - rpmsg_bind

API(2)

- Send data
  - rpmsg_send
  - rpmsg_get_tx_payload_buffer
  - rpmsg_send_nocopy
- Receive data through callback
  - rpmsg_ept_cb
  - rpmsg_hold_rx_buffer
  - rpmsg_release_rx_buffer

API(3)


Client
Client OpenAMP
Share Memory
Server OpenAMP
Server
rpmsg_register_callback
rpmsg_register_callback
initialize
device_created
device_created
rpmsg_create_ept
rpmsg_bind
rpmsg_send
rpmsg_ept_cb

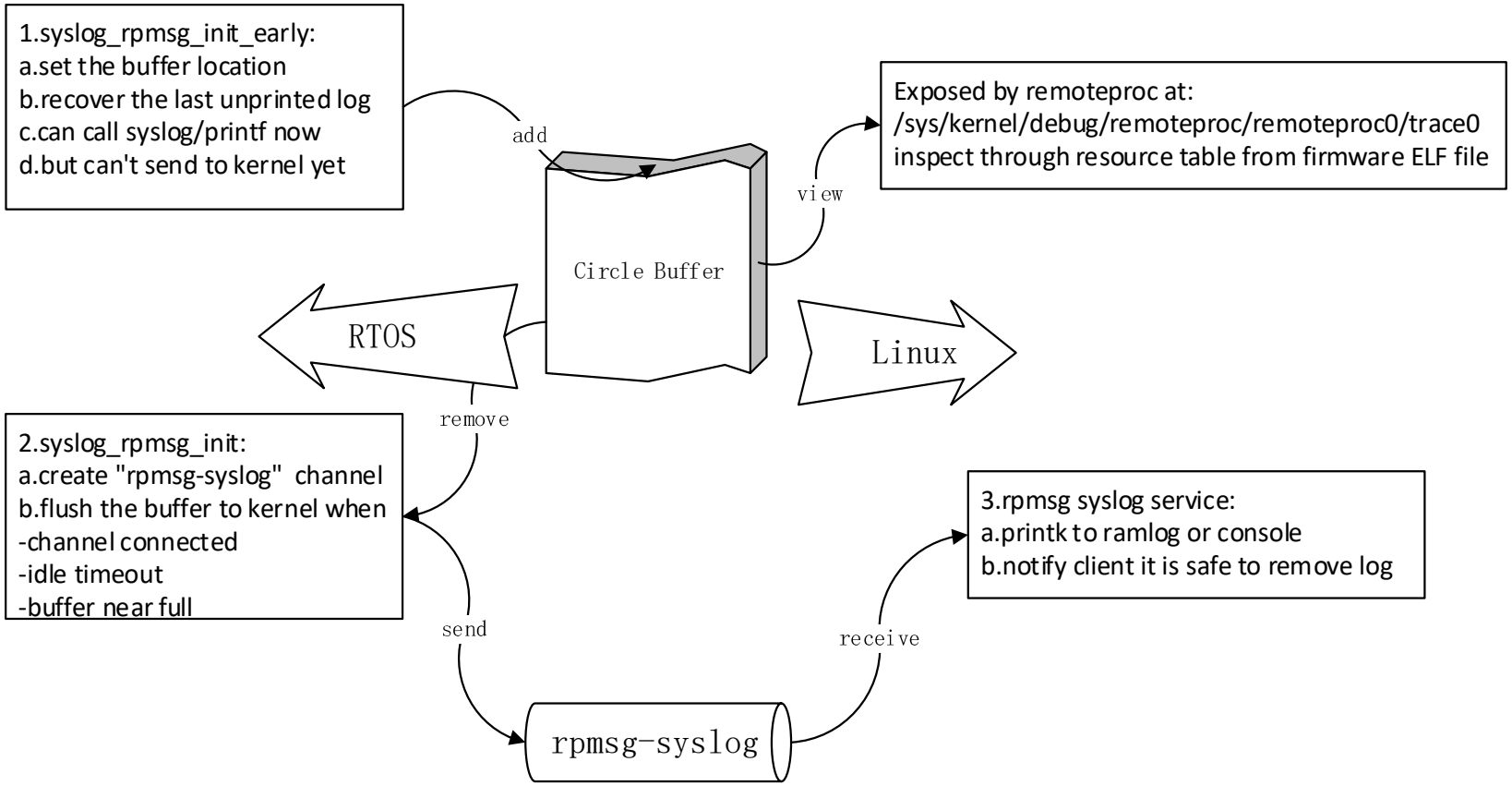
API(4)

- Support the multiple remote pair
- Support the multiple channel
- The channel is identified by unique name
- Convert to the unique id for space/speed
- The channel is bidirectional
- The buffer size and number is configurable
- Don’t support command/response

Rpmsg Syslog(1)

- Redirect log to master core
  - Linux kernel, NuttX…
- Work as early as possible
  - Two phase initialization
- Never lost the log
  - Hang during boot or runtime
  - Full system crash(panic, watchdog…)

Rpmsg Syslog(2)

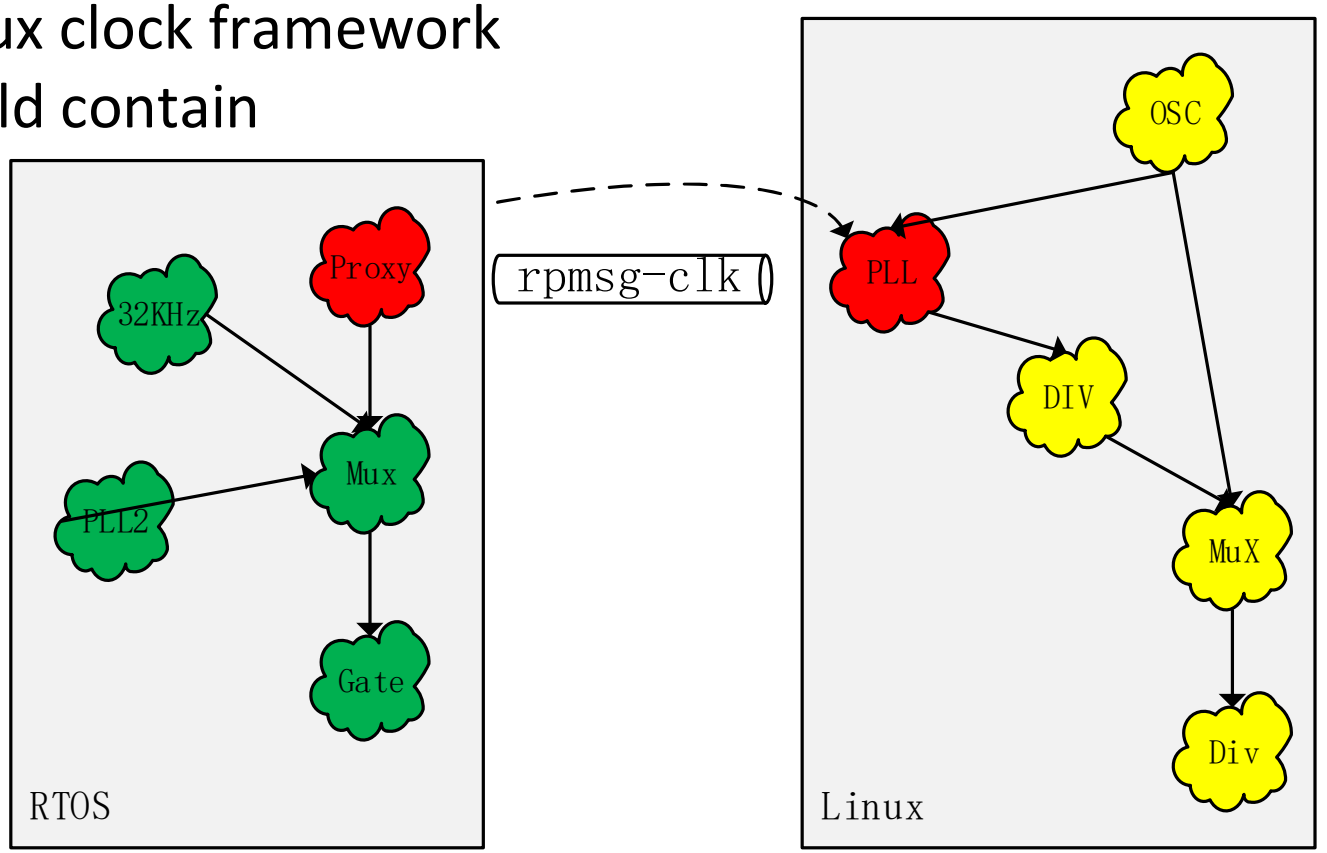


Rpmsg TTY

- Like pseudo terminal but between two CPU
- No different from real tty(open/read/write/close)
- Full duplex communication
- Support multiple channels as need
  - Connect RTOS shell
  - Make integrated GPS like external(NMEA)
  - Make integrated modem like external(ATCMD)

Rpmsg CLK

- Make RTOS access Linux clock framework
- Clock tree in RTOS could contain
  - The real clock
  - The rpmsg clock
  - Mix both

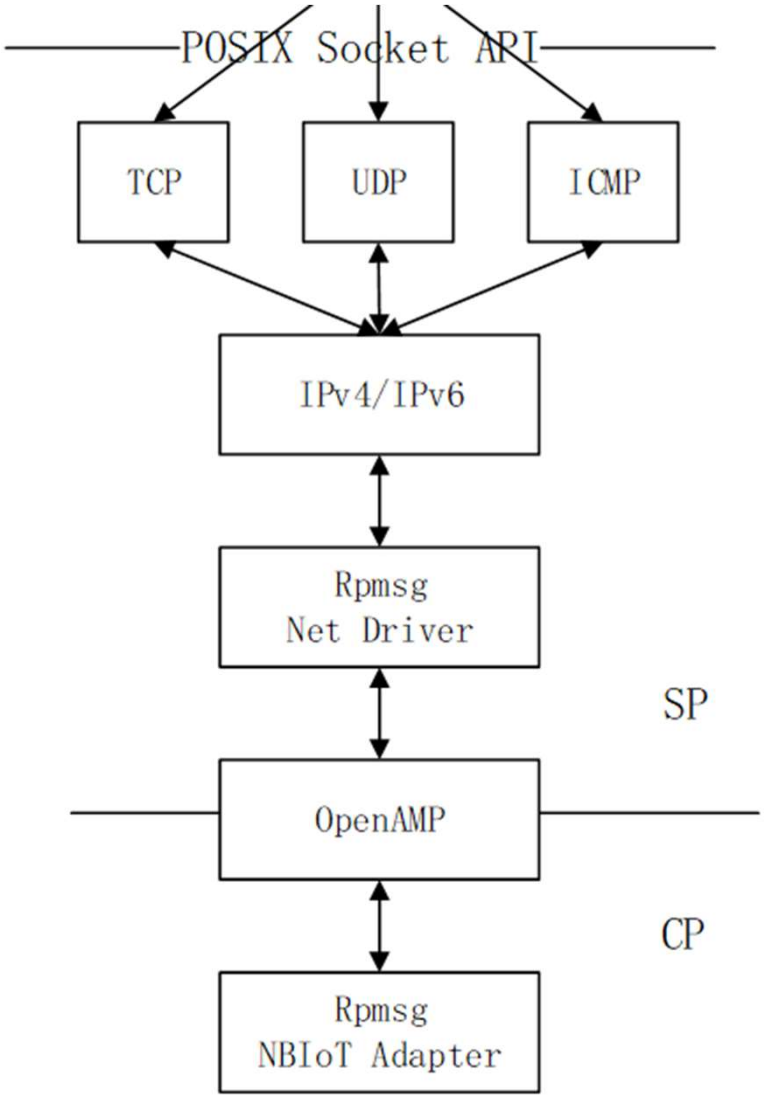


Rpmsg HostFS

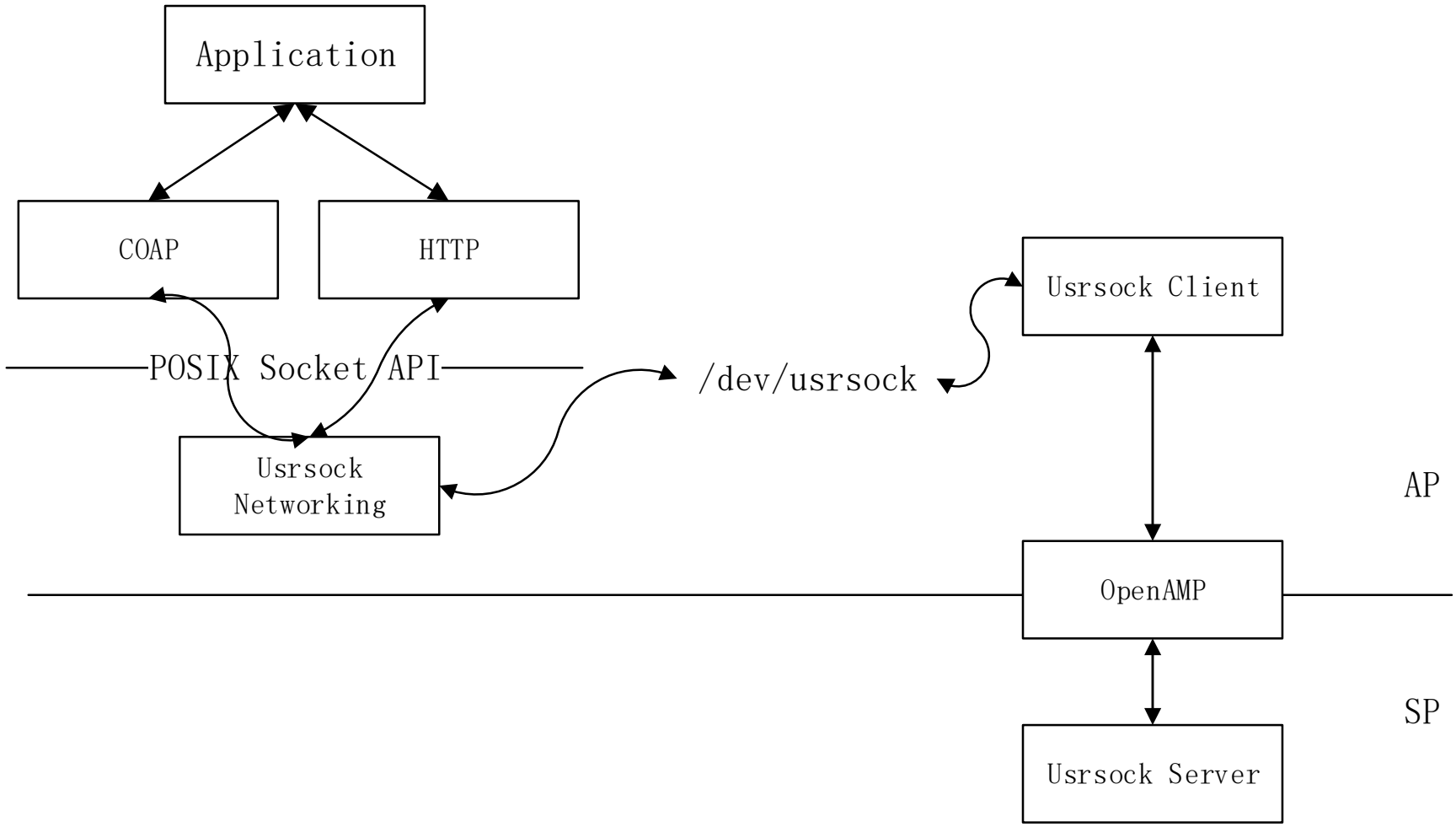
- Like NFS but between two CPU
- Fully access Host(Linux/NuttX) File system
  - Save the tuning parameter during manufacture
  - Load the tuning parameter file in production
  - Save audio dump to file for tuning/debugging
  - Dynamic loading module from host

Rpmsg Net(1)

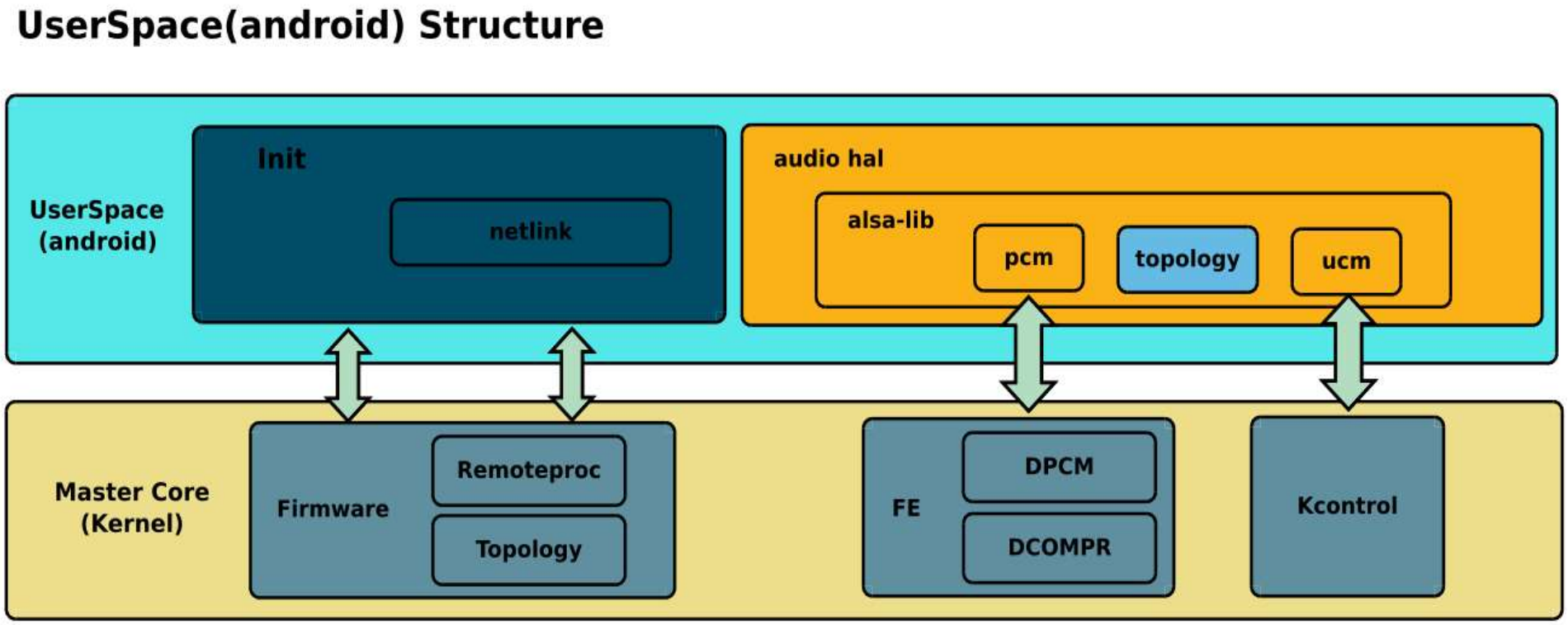
- Rpmsg Usrsock Client(AP)
- Rpmsg Usrsock Server(SP)
- Rpmsg Net Driver(SP)
- Rpmsg NBIoT Adapter(CP)



Rpmsg Net(2)

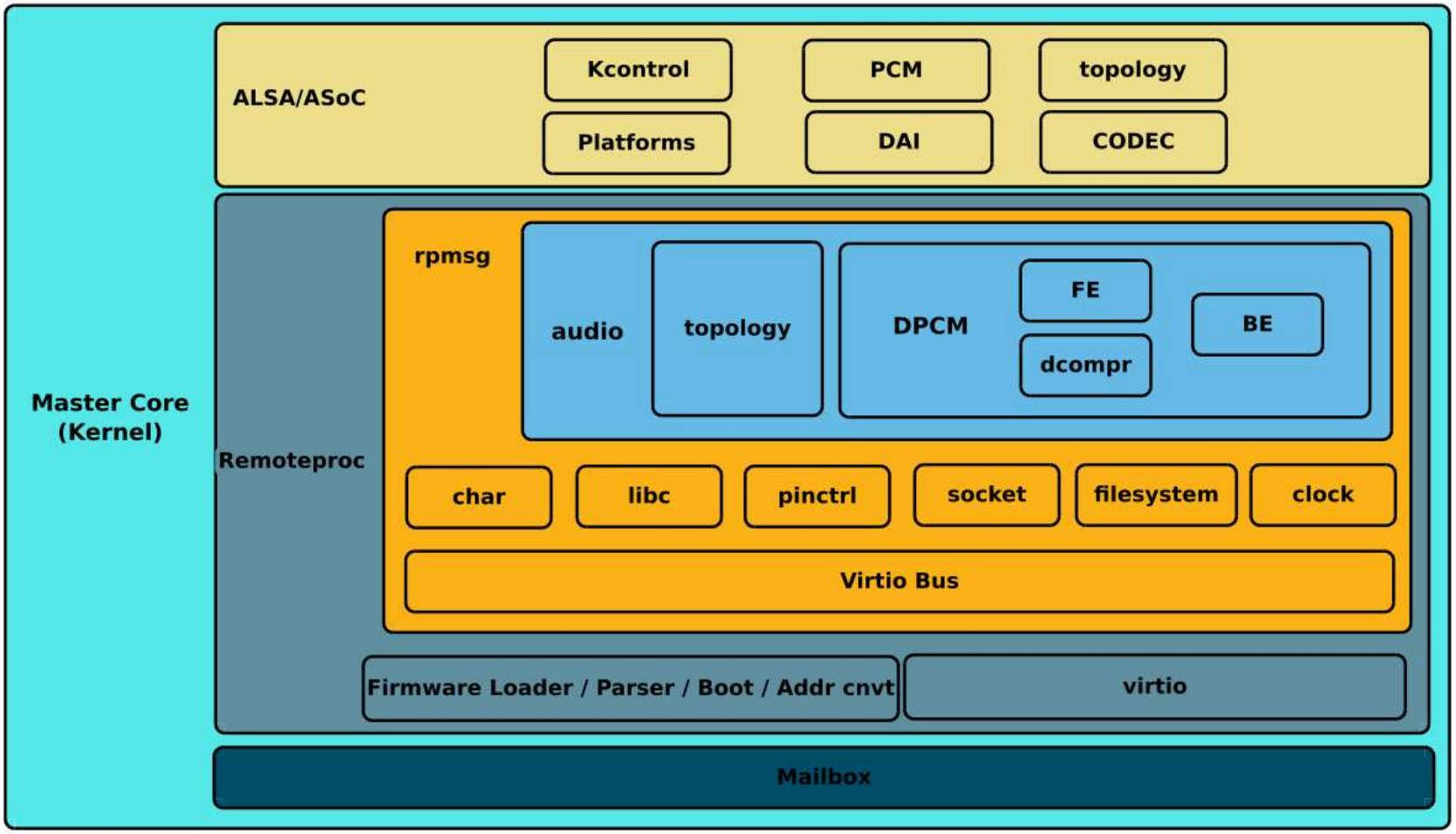

Application
COAP
HTTP
POSIX Socket API
Usrsock Networking
/dev/usrsock
Usrsock Client
AP
OpenAMP
SP
Usrsock Server

Audio Topology(1)



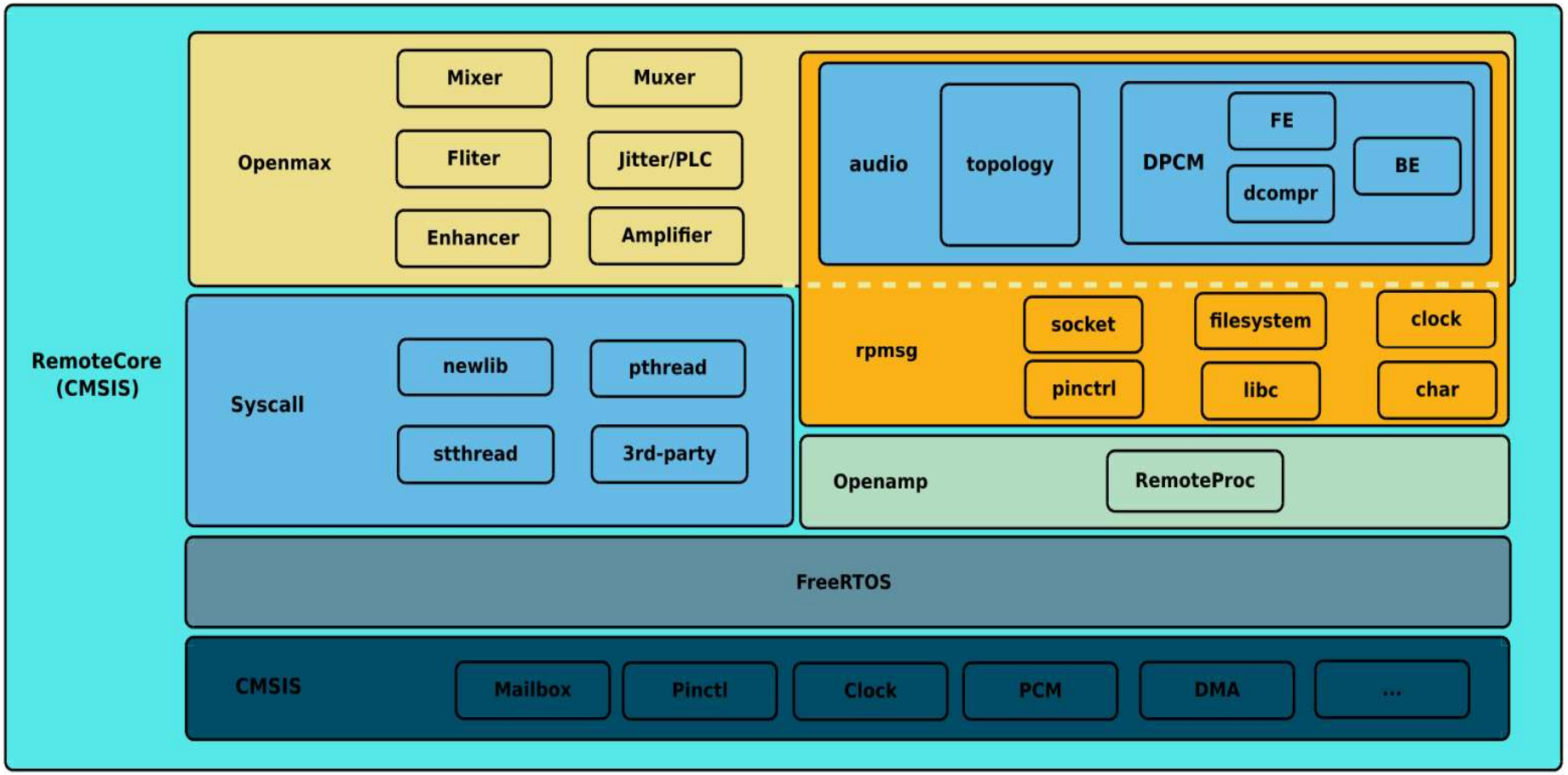
Audio Topology(2)

Master Core(Kernel) Structure



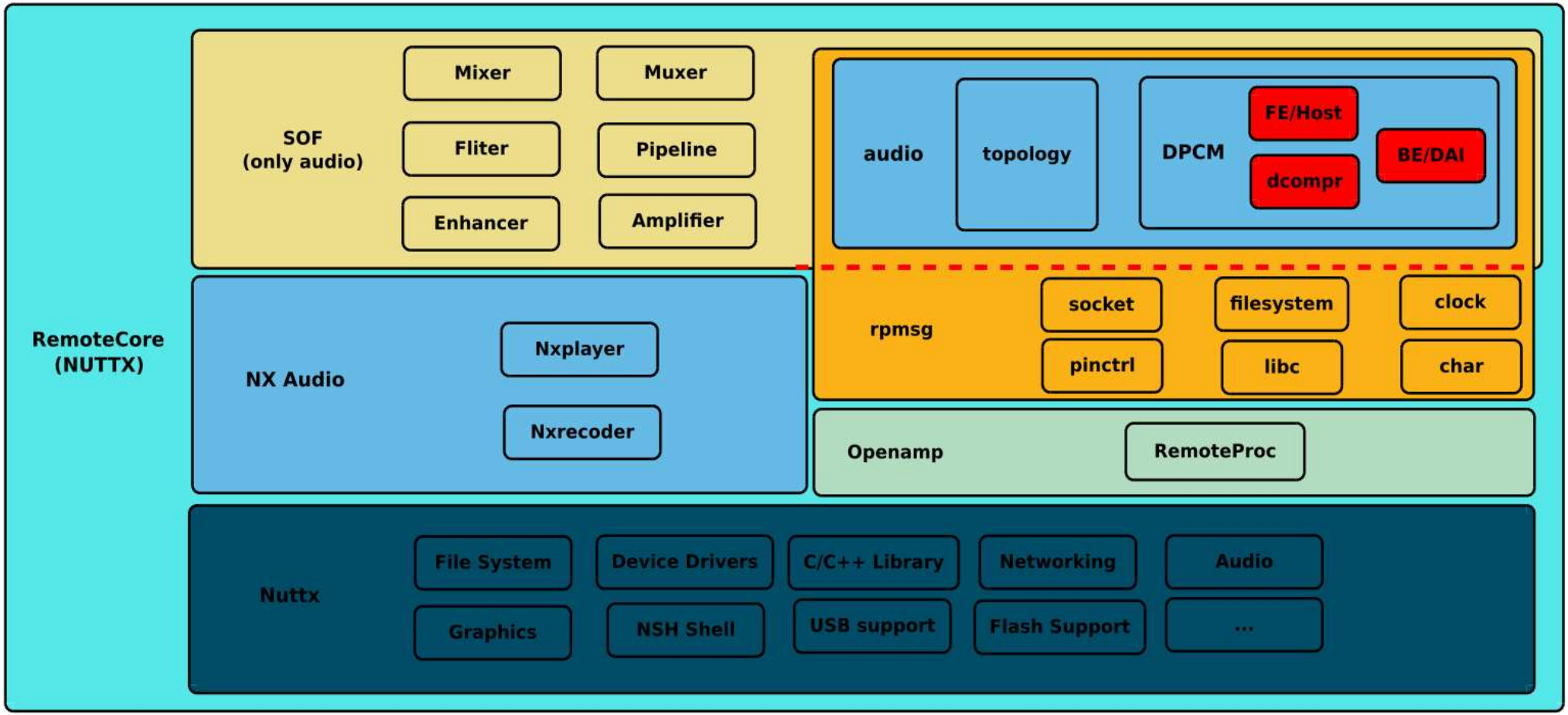
Audio Topology(3)

RemoteCore(cmsis) Structure, Openmax implement



Audio Topology(4)

RemoteCore(NUTTX) Structure, SOF implement



Conclusion

- It is very important and flexible to separate
  - Transport layer and Application layer
- Don’t expose rpmsg channel directly
  - Encapsulate into driver/fs/net subsystem
  - No difference from real device for caller
- Enhancement
  - More proxy for regulator/pinctrl…
  - New PF_RPMSG family for SunRPC/Protobuf…