OpenAMP

OpenAMP Project

May 06, 2024

CONTENTS:

CHAPTER

ONE

OPENAMP PROJECT

1.1 Project Overview

1.1.1 OpenAMP Intro

OpenAMP is a community effort that is standardizing and implementing how multiple embedded environments interact with each other using AMP. It provides conventions and standards as well as an open source implementation to facilitate AMP development for embedded systems. Read more about Asymmentric Multiprocessing *here*.

The vision is that regardless of the operating environment/operating system, it should be possible to use identical interfaces to interact with other operating environments in the same system.

Furthermore, these operating environments can interoperate over a standardized protocol, making it possible to mix and match any two or more operating systems in the same device.

Read more about OpenAMP System Considerations *here*.

To accomplish the above, OpenAMP is divided into the following efforts:

- **A standardization group under Linaro Community Projects**
  - **Standardizing the low-level protocol that allows systems to interact (*more info here*)**
    - Built on top of virtio BROKEN LINK
  - **Standardizing on the user level APIs that allow applications to be portable**
    - RPMSG BROKEN LINK
    - *remoteproc*
  - Standardizing on the low-level *OS/HW abstraction layer* that abstracts the open source implementation from the underlying OS and hardware, simplifying the porting to new environments
- **An open source project that implements a clean-room implementation of OpenAMP**
  - Runs in multiple environments, see below
  - BSD License
  - Please join the *OpenAMP open source project*!
  - See https://github.com/OpenAMP/open-amp

1.1.2 Operating Environments

OpenAMP is supported in various operating environments through an a) OpenAMP open source project (OAOS), b) a Linux kernel project (OALK), and c) multiple proprietary implementations (OAPI). The Linux kernel support (OALK) comes through the regular remoteproc/RPMsg/Virtio efforts in the kernel.

The operating environments that OpenAMP supports include:

- Linux user space - OAOS
- Linux kernel - OALK
- Multiple RTOS's - OAOS/OAPI including Nucleus, FreeRTOS, uC/OS, VxWorks and more
- Bare Metal (No OS) - OAOS
- In OS's on top of hypervisors - OAOS/OAPI
- Within hypervisors - OAPI

1.1.3 OpenAMP Capabilities

OpenAMP currently supports the following interactions between operating environments:

- Lifecycle operations - Such as starting and stopping another environment
- Messaging - Sending and receiving messages
- Proxy operations - Remote access to systems services such as file system

Read more about the OpenAMP System Components *here*.

In the future OpenAMP is envisioned to also encompass other areas important in a heterogeneous environment, such as power management and managing the lifecycle of non-CPU devices.

1.1.4 OpenAMP Guidelines

There are a few guiding principles that governs OpenAMP:

- **Provide a clean-room implementation of OpenAMP with business friendly APIs and licensing**
  - Allow for compatible proprietary implementations and products
- **Base as much as possible on existing technologies/open source projects/standards**
  - In particular remoteproc, RPMsg and virtio
- Never standardize on anything unless there is an open source implementation that can prove it
- **Always be backwards compatible (unless there is a really, really good reason to change)**
  - In particular make sure to be compatible with the Linux kernel implementation of remoteproc/RPMsg/virtio

1.2 Samples and Demos

1.2.1 System Reference Samples and Demos on the AMD-Xilinx platform

Demo: echo_test

This demo uses the Linux kernel rpmsg framework to send various size of data buffer to remote processor and validates integrity of received buffer from remote processor. If buffer data does not match, then number of different bytes are reported on console.

Platform: Xilinx Zynq UltraScale+ MPSoC(a.k.a ZynqMP)

Board: ZynqMP Zcu102

Remote Processor firmware (image_echo_test)

- Remote processor firmware for Xilinx ZynqMP cortex-r5 platform based on: rpmsg-echo.c
- Instructions to compile: ZynqMP r5f generic baremetal
- RPU firmware elf file is expected in sdk at path: /lib/firmware/
- Xilinx Vendor specific SDK is required to build RPU firmware: Xilinx Petalinux
- More information is provided here: Xilinx Wiki page for OpenAMP

Run the demo

Assume all the binaries are board specific.

```
# Specify remote processor firmware to be loaded.
echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware

# Load and start target firmware onto remote processor
echo start > /sys/class/remoteproc/remoteproc0/state

# check remote processor state
cat /sys/class/remoteproc/remoteproc0/state

# load rpmsg_char driver
modprobe rpmsg_char

# load rpmsg_ctrl driver
modprobe rpmsg_ctrl

# Run echo_test application on host processor
echo_test

# unload rpmsg_ctrl driver
modprobe -r rpmsg_ctrl

#unload rpmsg_char driver
modprobe -r rpmsg_char
```

(continues on next page)

(continued from previous page)

```
# Stop remote processor
echo stop > /sys/class/remoteproc/remoteproc0/state
```

Demo: matrix multiply

This example demonstrate interprocessor communication using rpmsg framework in the Linux kernelspace. Host (this) application generates two random matrices and send them to remote processor using rpmsg framework in the Linux kernelspace and waits for the response. Remote processor firmware receives both matrices and multiplies them and sends result back to host processor. Host processor prints the result on console after receiveing it. If -n option is passed, then above demo runs times. User can also pass custom endpoint information with -s (source address) and -e (destination address) options as well.

Platform: Xilinx Zynq UltraScale+ MPSoC(a.k.a ZynqMP)

Board: ZynqMP Zcu102

Remote Processor firmware (image_matrix_multiply)

- Remote processor firmware for Xilinx ZynqMP cortex-r5 platform based on: matrix_multiply.c
- Instructions to compile: ZynqMP r5f generic baremetal
- RPU firmware elf file is expected in sdk at path: /lib/firmware/
- Xilinx Vendor specific SDK is required to build RPU firmware: Xilinx Petalinux
- More information is provided here: Xilinx Wiki page for OpenAMP

Run the demo

Assume all the binaries are board specific.

```
# Specify remote processor firmware to be loaded.
echo image_matrix_multiply > /sys/class/remoteproc/remoteproc0/firmware

# Load and start target Firmware onto remote processor
echo start > /sys/class/remoteproc/remoteproc0/state

# check remote processor state
cat /sys/class/remoteproc/remoteproc0/state

# load rpmsg_char driver
modprobe rpmsg_char

# load rpmsg_ctrl driver
modprobe rpmsg_ctrl

# Run Matrix multiplication application on host processor
mat_mul_demo

# unload rpmsg_ctrl driver
```

(continues on next page)

(continued from previous page)

```
modprobe -r rpmsg_ctrl

#unload rpmsg_char driver
modprobe -r rpmsg_char

# Stop remote processor
echo stop > /sys/class/remoteproc/remoteproc0/state
```

Demo: proxy_app

This app demonstrates two functionality

1) Use of host processor's file system by remote processor
2) remote processor's standard IO redirection to host processor's standard IO

case 1: This app allows remote processor to use file system of host processor. Host processor file system acts as proxy of remote file system. Remote processor can use open, read, write, close calls to interact with files on host processor.

File "remote.file" is available after app exits on host side that is created by remote processor that contains string "This is a test string being written to file.." written by remote firmware. This demonstrates remote firmware can create and write files on host side.

case 2: This application also demonstrates redirection of standard IO. Remote processor can use host processor's stdin and stdout via proxy service that is implemented on host side. This is achieved with open-amp proxy service implemented here: rpmsg_retarget.c Remote side firmware uses two types of output functions to print message on console 1) xil_printf i.e. using same UART console as of APU and 2) Standard "printf" function that is re-directed to standard output of Host. Both function uses different ways to output messages, but using same console.

This is interactive demo:

1. When the remote application prompts you to Enter name, enter any string without space.
2. When the remote application prompts you to Enter age , enter any integer.
3. When the remote application prompts you to Enter value for pi, enter any floating point number. After this, remote application will print all the inputs entered by user on console of host processor.

Remote firmware's standard IO are redirected to host processor's standard IO. So, when remote uses "printf" and "scanf" functions actually host processor's console is used for printing output and scanning inputs. Host communicates with remote via rpmsg_char driver and Remote communicates to Host via redirected Standard IO.

Platform: Xilinx Zynq UltraScale+ MPSoC(a.k.a ZynqMP)

Board: ZynqMP Zcu102

Remote Processor firmware (image_rpc_demo)

- Remote processor firmware for Xilinx ZynqMP cortex-r5 platform based on: rpc_demo.c
- Instructions to compile: ZynqMP r5f generic baremetal
- RPU firmware elf file is expected in sdk at path: /lib/firmware/
- Xilinx Vendor specific SDK is required to build RPU firmware: Xilinx Petalinux
- More information is provided here: Xilinx Wiki page for OpenAMP

Run the demo

Assume all the binaries are zcu102 board specific.

```
# Specify remote processor firmware to be loaded.
echo image_rpc_demo > /sys/class/remoteproc/remoteproc0/firmware

# Load and start target Firmware onto remote processor.
echo start > /sys/class/remoteproc/remoteproc0/state

# load rpmsg_char driver
modprobe rpmsg_char

# load rpmsg_ctrl driver
modprobe rpmsg_ctrl

# Run proxy application.
proxy_app

# unload rpmsg_ctrl driver
modprobe -r rpmsg_ctrl

#unload rpmsg_char driver
modprobe -r rpmsg_char

# Stop target firmware
echo stop > /sys/class/remoteproc/remoteproc0/state
```

1.2.2 System Reference Samples and Demos on STM32MP157C/F-DK2 board

Based on a fork of the yocto [meta-st-stm32mp-oss](https://github.com/STMicroelectronics/meta-st-stm32mp-oss) environment, designed to update and test upstream code on STM32MP boards,

Prerequisite

Some specifics package could be needed to build the ST images. For details refer to

STMPU wiki PC prerequisite

Installation

Create the structure of the project

```
mkdir stm32mp15-demo
cd stm32mp15-demo
mkdir stm32mp1_distrib_oss
mkdir zephy_rpmsg_multi_services
```

At this step you should see following folder hierarchy:

```
stm32mp15-demo
    |___ stm32mp1_distrib_oss
    |___ zephy_rpmsg_multi_services
```

Generate the stm32mp15 image

Install stm32mp1_distrib_oss kirkstone

From the stm32mp15-demo directory

```
cd stm32mp1_distrib_oss

mkdir -p layers/meta-st
git clone https://github.com/openembedded/openembedded-core.git layers/openembedded-core
cd layers/openembedded-core
git checkout -b WORKING origin/kirkstone
cd -

git clone https://github.com/openembedded/bitbake.git layers/openembedded-core/bitbake
cd layers/openembedded-core/bitbake
git checkout -b WORKING  origin/2.0
cd -

git clone https://github.com/openembedded/meta-openembedded.git layers/meta-openembedded
cd layers/meta-openembedded
git checkout -b WORKING origin/kirkstone
cd -

git clone https://github.com/STMicroelectronics/meta-st-stm32mp-oss.git layers/meta-st/
↪meta-st-stm32mp-oss
cd layers/meta-st/meta-st-stm32mp-oss
git checkout -b WORKING origin/kirkstone
cd -
```

Initialize the Open Embedded build environment

The OpenEmbedded environment setup script must be run once in each new working terminal in which you use the BitBake or devtool tools (see later) from stm32mp15-demo/stm32mp1_distrib_oss directory

```
source ./layers/openembedded-core/oe-init-build-env build-stm32mp15-disco-oss

bitbake-layers add-layer ../layers/meta-openembedded/meta-oe
bitbake-layers add-layer ../layers/meta-openembedded/meta-perl
bitbake-layers add-layer ../layers/meta-openembedded/meta-python
bitbake-layers add-layer ../layers/meta-st/meta-st-stm32mp-oss

echo "MACHINE = \"stm32mp15-disco-oss\"" >> conf/local.conf
echo "DISTRO = \"nodistro\"" >> conf/local.conf
echo "PACKAGE_CLASSES = \"package_deb\" " >> conf/local.conf
```

Build stm32mp1_distrib_oss image

From stm32mp15-demo/stm32mp1_distrib_oss/build-stm32mp15-disco-oss/ directory

```
bitbake core-image-base
```

Note that

- to build around 30 GB is needed
- building the distribution can take more than 2 hours depending on performance of the PC.

Install stm32mp1_distrib_oss

From 'stm32mp15-demo/stm32mp1_distrib_oss/build-stm32mp15-disco-oss/' directory,populate your microSD card inserted on your HOST PC using command

```
cd tmp-glibc/deploy/images/stm32mp15-disco-oss/
# flash wic image on your sdcar. replace <device> by mmcblk<X> (X = 0,1..) or sd<Y> ( Y␣
↪= b,c,d,..) depending on the connection
dd if=core-image-base-stm32mp15-disco-oss.wic of=/dev/<device> bs=8M conv=fdatasync
```

Generate the Zephyr rpmsg multi service example

Prerequisite

Please refer to the Getting Started Guide zephyr documentation

Initialize the Zephyr environment

```
cd zephy_rpmsg_multi_services
git clone https://github.com/OpenAMP/openamp-system-reference.git
west init
west update
```

Build the Zephyr image

From the zephy_rpmsg_multi_services directory

```
west build -b stm32mp157c_dk2 openamp-system-reference/examples/zephyr/rpmsg_multi_
↪services
```

Install the Zephyr binary on the sdcard

The Zephyr sample binary is available in the sub-folder of build directory stm32mp15-demo/zephy_rpmsg_multi_services/build/zephyr/rpmsg_multi_services.elf. It needs to be installed on the "rootfs" partition of the sdcard

```
sudo cp build/zephyr/rpmsg_multi_services.elf <mountpoint>/rootfs/lib/firmware/
```

Don't forget to properly unmoumt the sdcard partitions.

Demos

Start the demo environment

- power on the stm32mp157C/F-dk2 board, and wait login prompt on your serial terminal

```
stm32mp15-disco-oss login: root
```

There are 2 ways to start the coprocessor:

- During the runtime, by the Linux remoteproc framework

```
root@stm32mp15-disco-oss:~# cat /sys/class/remoteproc/remoteproc0/state
offline
root@stm32mp15-disco-oss:~# echo rpmsg_multi_services.elf > /sys/class/remoteproc/
↪remoteproc0/firmware
root@stm32mp15-disco-oss:~# echo start >/sys/class/remoteproc/remoteproc0/state
root@stm32mp15-disco-oss:~# cat /sys/class/remoteproc/remoteproc0/state
running
```

- In the boot stages, by the U-Boot remoteproc framework
  - Prerequisite Copy the firmware in the bootfs partition

```
root@stm32mp15-disco-oss:~# cp /lib/firmware/rpmsg_multi_services.elf /boot/
root@stm32mp15-disco-oss:~# sync
```

  - Boot the board and go in U-Boot console

```
root@stm32mp15-disco-oss:~# reboot
```

Enter in the U-boot console by interrupting the boot with any keyboard key.

```
STM32MP>
```

  - Load and start the Coprocessor firmware:

```
STM32MP> load mmc 0#bootfs ${kernel_addr_r} rpmsg_multi_services.elf
816776 bytes read in 148 ms (5.3 MiB/s)
STM32MP> rproc init
STM32MP> rproc load 0 ${kernel_addr_r} ${filesize}
Load Remote Processor 0 with data@addr=0xc2000000 816776 bytes: Success!
STM32MP> rproc start 0
STM32MP> run bootcmd
```

To automatically load the firmware by U-Boot, refer to the STMicorelectronics wiki

- Check that the remoteproc state is "detached"

```
root@stm32mp15-disco-oss:~# cat /sys/class/remoteproc/remoteproc0/state
detached
```

- Attach the Linux remoteproc framework to the Zephyr

```
root@stm32mp15-disco-oss:~# echo start >/sys/class/remoteproc/remoteproc0/
↪state
root@stm32mp15-disco-oss:~# cat /sys/class/remoteproc/remoteproc0/state
attached
```

The communication with the Coprocessor is not initilaized, following traces on console are observed:

```
root@stm32mp15-disco-oss:~#
[   54.495343] virtio_rpmsg_bus virtio0: rpmsg host is online
[   54.500044] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x400
[   54.507923] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty addr 0x401
[   54.514795] virtio_rpmsg_bus virtio0: creating channel rpmsg-raw addr 0x402
[   54.548954] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: new channel:␣
↪0x402 -> 0x400!
[   54.557337] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 1␣
↪(src:    0x400)
[   54.565532] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 2␣
↪(src:    0x400)
[   54.581090] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 3␣
↪(src:    0x400)
[   54.588699] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 4␣
↪(src:    0x400)
[   54.599424] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 5␣
↪(src:    0x400)
...
```

This informs that following rpmsg channels devices have been created:

- a rpmsg-client-sample device
- a rpmsg-tty device
- a rpmsg-raw device

Run the multi RPMsg services demo

OpenAMP multi services sample Application using resource table

Overview

This application demonstrates how to use OpenAMP with Zephyr based on a resource table. It is designed to respond to the:

- Linux rpmsg client sample
- Linux rpmsg tty driver

- Linux rpmsg char driver

This sample implementation is compatible with platforms that embed a Linux kernel OS on the main processor and a Zephyr application on the co-processor.

Tested on board:

- Lstm32mp157C_dk2
- Lstm32mp157F_dk2

Building the application

Zephyr

```
west build -b <target board> openamp-system-reference/examples/zephyr/rpmsg_multi_
↪services
```

Linux

Enable:

- the SAMPLE_RPMSG_CLIENT configuration to build and install the rpmsg_client_sample.ko module on the target,
- the RPMSG_TTY configuration to build and install the rpmsg_tty.ko module on the target
- the RPMSG_CHAR configuration to build and install the rpmsg_char.ko module on the target
- build and install the rpmsg-utils binaries

Running the sample

Zephyr console

Open a serial terminal (minicom, putty, etc.) and connect the board with the following settings:

- Speed: 115200
- Data: 8 bits
- Parity: None
- Stop bits: 1

Reset the board.

Linux console

Open a Linux shell (minicom, ssh, etc.)

- Insert a module into the Linux Kernel:

```
root@linuxshell: insmod rpmsg_client_sample.ko rpmsg_tty.ko rpmsg_char.ko rpmsg_ctrl.ko
```

- Start the demo environment

First copy the rpmsg_multi_services.elf file on the target rrottfs in /lib/firmware folder. Then start the firmware:

```
root@linuxshell: echo rpmsg_multi_services.elf > /sys/class/remoteproc/remoteproc0/
↪firmware
root@linuxshell: echo start >/sys/class/remoteproc/remoteproc0/state
```

Result on Zephyr console on boot

The following messages will appear on the corresponding Zephyr console

```
[   54.495343] virtio_rpmsg_bus virtio0: rpmsg host is online
[   54.500044] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x400
[   54.507923] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty addr 0x401
[   54.514795] virtio_rpmsg_bus virtio0: creating channel rpmsg-raw addr 0x402
[   54.548954] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: new channel:␣
↪0x402 -> 0x400!
[   54.557337] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 1␣
↪(src: 0x400)
[   54.565532] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 2␣
↪(src: 0x400)
[   54.581090] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 3␣
↪(src: 0x400)
[   54.588699] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 4␣
↪(src: 0x400)
[   54.599424] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 5␣
↪(src: 0x400)
...
```

This inform that following rpmsg channels devices have been created:

- a rpmsg-client-sample device

```
root@linuxshell: dmesg
...
[   54.500044] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr␣
↪0x400
...
```

- a rpmsg-tty device

```
root@linuxshell: ls /dev/ttyRPMSG*
/dev/ttyRPMSG0
```

- a rpmsg-raw device

```
root@linuxshell: ls /dev/rpmsg?
/dev/rpmsg0
```

The following messages will appear on the corresponding Zephyr console or in the remoteproc trace buffer depending on the Hardware.

```
root@linuxshell:  cat /sys/kernel/debug/remoteproc/remoteproc0/trace0
*** Booting Zephyr OS build zephyr-v3.2.0-1-g6b49008b6b83  ***
Starting application threads!

OpenAMP[remote]  linux responder demo started

OpenAMP[remote] Linux sample client responder started

OpenAMP[remote] Linux tty responder started

OpenAMP[remote] Linux raw data responder started

OpenAMP[remote] create a endpoint with address and dest_address set to 0x1
OpenAMP Linux sample client responder ended
```

Demo 1: rpmsg-client-sample device

Principle

This demo is automatically run when the co-processor firmware is started. It confirms that the rpmsg and virtio protocols are working properly. The Zephyr requests the creation of the rpmsg-client-sample channel to the Linux rpmsg framework using the “name service announcement” rpmsg. On message reception the Linux rpmsg bus creates an associated device and probes the rpmsg-client-sample driver. The Linux rpmsg-client-sample driver sent 100 messages to the remote processor, which answers to each message. After answering to each rpmsgs the Zephyr destroys the channel.

Associated traces

```
[   54.548954] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: new␣
↪channel: 0x402 -> 0x400!
[   54.557337] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024:␣
↪incoming msg 1 (src: 0x400)
[   54.565532] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024:␣
↪incoming msg 2 (src: 0x400)

  ...

[   55.436401] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024:␣
↪incoming msg 99 (src: 0x400)
[   55.445343] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024:␣
↪incoming msg 100 (src: 0x400)
[   55.454280] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024:␣
↪goodbye!
[   55.461424] virtio_rpmsg_bus virtio0: destroying channel rpmsg-client-
```

(continues on next page)

(continued from previous page)

```
↪sample addr 0x400
[   55.469707] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: rpmsg␣
↪sample client driver is removed
```

Demo 2: rpmsg-tty device

Principle

This channel allows to create a /dev/ttyRPMSGx for terminal based communication with Zephyr.

Demo

1. Check presence of the /dev/ttyRPMSG0

   By default the Zephyr has created a rpmsg-tty channel

   ```
   [   54.507923] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty addr 0x401
   root@linuxshell: ls /dev/ttyRPMSG*
   /dev/ttyRPMSG0
   ```

2. Send and receive messages on /dev/ttyRPMSG0

   The zephyr is programmed to resent received messages with a prefixed "TTY 0: ", 0 is the instance of the tty link

   ```
   root@linuxshell: cat /dev/ttyRPMSG0 &
   root@linuxshell: echo "Hello Zephyr" >/dev/ttyRPMSG0
   TTY 0: Hello Zephyr
   root@linuxshell: echo "Goodbye Zephyr" >/dev/ttyRPMSG0
   TTY 0: Goodbye Zephyr
   ```

Demo 3: dynamic creation/release of a rpmsg-tty device

Principle

This demo is based on the rpmsg_char restructuring series not yet upstreamed. This series de-correlates the /dev/rpmsg_ctrl from the rpmsg_char device and then introduces 2 new rpmsg IOCtrls:

- RPMSG_CREATE_DEV_IOCTL : to create a local rpmsg device and to send a name service creation announcement to the remote processor
- RPMSG_RELEASE_DEV_IOCTL: release the local rpmsg device and to send a name service destroy announcement to the remote processor

Demo

1. Prerequisite

   Due to a limitation in the rpmsg protocol, the zephyr does not know the existence of the /dev/ttyRPMG0 until the Linux sends it a first message. Creating a new channel before this first one is well establish leads to bad endpoints association. To avoid this, just send a message on /dev/ttyRPMSG0

   ```
   root@linuxshell: cat /dev/ttyRPMSG0 &
   root@linuxshell: echo "Hello Zephyr" >/dev/ttyRPMSG0
   TTY 0: Hello Zephyr
   ```

   Download rpmsg-utils tools relying on the /dev/rpmsg_ctrl, and compile it in an arm environment using make instruction and install it on target.

   optional: enable rpmsg bus trace to observe RPmsg in kernel trace:

   ```
   root@linuxshell: echo -n 'file virtio_rpmsg_bus.c +p' > /sys/kernel/debug/
   ↪dynamic_debug/control
   ```

2. create a new TTY channel

   Create a rpmsg-tty channel from Linux with local address set to 257 and undefined remote address -1.

   Note: Current Linux implementation has a limitation. When it initiates a name service announcement, It is not able to associate the remote endpoint to the created channel. Following patch has to be applied on top waiting a upstreamed solution:

   <https://lore.kernel.org/lkml/20220316153001.662422-1-arnaud.pouliquen@foss.st.com/>

   ```
   root@linuxshell: ./rpmsg_export_dev /dev/rpmsg_ctrl0 rpmsg-tty 257 -1
   ```

   The /dev/ttyRPMSG1 is created

   ```
   root@linuxshell: ls /dev/ttyRPMSG*
   /dev/ttyRPMSG0  /dev/ttyRPMSG1
   ```

   A name service announcement has been sent to Zephyr, which has created a local endpoint (@ 0x400), and sent a "bound" message to the /dev/ttyRPMG1 (@ 257)

   ```
   root@linuxshell: dmesg
   [  115.757439] rpmsg_tty virtio0.rpmsg-tty.257.-1: TX From 0x101, To 0x35, Len␣
   ↪40, Flags 0, Reserved 0
   [  115.757497] rpmsg_virtio TX: 01 01 00 00 35 00 00 00 00 00 00 00 28 00 00␣
   ↪00  ....5.......(...
   [  115.757514] rpmsg_virtio TX: 72 70 6d 73 67 2d 74 74 79 00 00 00 00 00 00␣
   ↪00  rpmsg-tty.......
   [  115.757528] rpmsg_virtio TX: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00␣
   ↪00  ................
   [  115.757540] rpmsg_virtio TX: 01 01 00 00 00 00 00 00                      ␣
   ↪  ........
   [  115.757568] remoteproc remoteproc0: kicking vq index: 1
   [  115.757590] stm32-ipcc 4c001000.mailbox: stm32_ipcc_send_data: chan:1
   [  115.757850] stm32-ipcc 4c001000.mailbox: stm32_ipcc_tx_irq: chan:1 tx
   ```

(continues on next page)

(continued from previous page)

```
[  115.757906] stm32-ipcc 4c001000.mailbox: stm32_ipcc_rx_irq: chan:0 rx
[  115.757969] remoteproc remoteproc0: vq index 0 is interrupted
[  115.757994] virtio_rpmsg_bus virtio0: From: 0x400, To: 0x101, Len: 6,␣
↪Flags: 0, Reserved: 0
[  115.758022] rpmsg_virtio RX: 00 04 00 00 01 01 00 00 00 00 00 00 06 00 00␣
↪00  ................
[  115.758035] rpmsg_virtio RX: 62 6f 75 6e 64 00                              ␣
↪  bound.
[  115.758077] virtio_rpmsg_bus virtio0: Received 1 messages
```

3. Play with /dev/ttyRPMSG0 and /dev/ttyRPMSG1

```
root@linuxshell: cat /dev/ttyRPMSG0 &
root@linuxshell: cat /dev/ttyRPMSG1 &
root@linuxshell: echo hello dev0 >/dev/ttyRPMSG0
TTY 0: hello dev0
root@linuxshell: echo hello dev1 >/dev/ttyRPMSG1
TTY 1: hello dev1
```

4. Destroy RPMSG TTY devices

   Destroy the /dev/ttyRPMSG1

```
root@linuxshell: ./rpmsg_export_dev /dev/rpmsg_ctrl0 -d rpmsg-tty 257 -1
```

   Destroy the /dev/ttyRPMSG0 * Get the source address

```
root@linuxshell: cat /sys/bus/rpmsg/devices/virtio0.rpmsg-tty.-1.*/src
0x402
```

   - Destroy the /dev/ttyRPMSG0 specifying the address 1026 (0x402)

```
root@linuxshell: ./rpmsg_export_dev /dev/rpmsg_ctrl0 -d rpmsg-tty 1026 -1
```

   The /dev/ttyRPMGx devices no more exists

Demo 4: rpmsg-char device

Principle

This channel allows to create a /dev/rpmsgX for character device based communication with Zephyr.

Demo

1. Prerequisite

   Download rpmsg-utils tools relying on the /dev/rpmsg_ctrl, an compile it in an arm environment using make instruction and install it on target

   optional: enable rpmsg bus trace to observe rp messages in kernel trace:

```
echo -n 'file virtio_rpmsg_bus.c +p' > /sys/kernel/debug/dynamic_debug/control
```

2. Check presence of the /dev/rpmsg0

   By default the Zephyr has created a rpmsg-raw channel

```
[   54.514795] virtio_rpmsg_bus virtio0: creating channel rpmsg-raw addr 0x402
```

3. Check device exists

```
root@linuxshell: ls /dev/rpmsg?
/dev/rpmsg0
```

4. Send and receive messages on /dev/rpmsg0

   The zephyr is programmed to resent received message with a prefixed "from ept 0x0402: ", 0x0402 is the zephyr endpoint address

```
root@linuxshell: ./rpmsg_ping /dev/rpmsg0
message for /dev/rpmsg0: "from ept 0x0402: ping /dev/rpmsg0"
```

Demo 5: Multi endpoints demo using rpmsg-ctrl device

Principle

Use the rpmsg_ctrl RPMSG_CREATE_EPT_IOCTL IoCtrl to instantiate endpoints on Linux side. Theses endpoints will not be associated to a channel but will communicate with a predefined remote proc endpoint. For each endpoint created, a /dev/rpmsg sysfs interface will be created On Zephyr side, an endpoint with a prefixed address 0x1 has been created. When it receives a message it re-sends a the message to the Linux sender endpoint, prefixed by "from ept 0x0001:"

Demo

1. Prerequisite

   Download rpmsg-util tools relying on the /dev/rpmsg_ctrl, an compile it in an arm environment using make instruction and install it on target

   optional: enable rpmsg bus trace to observe rp messages in kernel trace:

```
echo -n 'file virtio_rpmsg_bus.c +p' > /sys/kernel/debug/dynamic_debug/control
```

2. Check presence of the /dev/rpmsg0

   By default the Zephyr has created a rpmsg-raw channel

```
[   54.514795] virtio_rpmsg_bus virtio0: creating channel rpmsg-raw addr 0x402
```

3. Check device exists

```
root@linuxshell: ls /dev/rpmsg*
/dev/rpmsg0       /dev/rpmsg_ctrl0
```

4. Create 3 new endpoints

```
root@linuxshell: ./rpmsg_export_ept /dev/rpmsg_ctrl0 my_endpoint1 100 1
root@linuxshell: ./rpmsg_export_ept /dev/rpmsg_ctrl0 my_endpoint2 101 1
root@linuxshell: ./rpmsg_export_ept /dev/rpmsg_ctrl0 my_endpoint2 103 1
root@linuxshell: ls /dev/rpmsg?
/dev/rpmsg0  /dev/rpmsg1  /dev/rpmsg2  /dev/rpmsg3
```

5. Test them

```
root@linuxshell: ./rpmsg_ping  /dev/rpmsg0
message for /dev/rpmsg0: "from ept 0x0402: ping /dev/rpmsg0"
root@linuxshell: ./rpmsg_ping  /dev/rpmsg1
message for /dev/rpmsg1: "from ept 0x0001: ping /dev/rpmsg1"
root@linuxshell: ./rpmsg_ping  /dev/rpmsg2
message for /dev/rpmsg2: "from ept 0x0001: ping /dev/rpmsg2"
root@linuxshell: ./rpmsg_ping  /dev/rpmsg3
message for /dev/rpmsg3: "from ept 0x0001: ping /dev/rpmsg3"
```

6. Destroy them

```
root@linuxshell: ./rpmsg_destroy_ept /dev/rpmsg1
root@linuxshell: ./rpmsg_destroy_ept /dev/rpmsg2
root@linuxshell: ./rpmsg_destroy_ept /dev/rpmsg3
root@linuxshell: ls /dev/rpmsg?
/dev/rpmsg0
```

1.2.3 linux_rpc_demo

This readme is about the OpenAMP linux_rpc_demo.

The linux_rpc_demo is about remote procedure calls between linux host and linux remote using rpmsg to perform

1. File operations such as open, read, write and close
2. I/O operation such as printf, scanf

Compilation

Linux Compilation

- Install libsysfs devel and libhugetlbfs devel packages on your Linux.
- build libmetal library:

```
    $ mkdir -p build-libmetal
    $ cd build-libmetal
    $ cmake <libmetal_source>
    $ make VERBOSE=1 DESTDIR=<libmetal_install> install
```

- build OpenAMP library:

```
    $ mkdir -p build-openamp
    $ cd build-openamp
    $ cmake <openamp_source> -DCMAKE_INCLUDE_PATH=<libmetal_built_include_dir> \
```

(continues on next page)

(continued from previous page)

```
          -DCMAKE_LIBRARY_PATH=<libmetal_built_lib_dir> -DWITH_APPS=ON -DWITH_
↪PROXY=ON
    $ make VERBOSE=1 DESTDIR=$(pwd) install
```

The OpenAMP library will be generated to `build/usr/local/lib` directory, headers will be generated to `build/usr/local/include` directory, and the applications executable will be generated to `build/usr/local/bin` directory.

- cmake option `-DWITH_APPS=ON` is to build the demonstration applications.
- cmake option `-DWITH_PROXY=ON` to build the linux rpc app.

Run the Demo

linux_rpc_demo:

- Start rpc demo server on one console

```
$ sudo LD_LIBRARY_PATH=<openamp_built>/usr/local/lib:<libmetal_built>/usr/local/lib␣
↪\
   build-openamp/usr/local/bin/linux_rpc_demod-shared
```

- Run rpc demo client on another console to perform file and I/O operations on the server

```
$ sudo LD_LIBRARY_PATH=<openamp_built>/usr/local/lib:<libmetal_built>/usr/local/lib␣
↪\
   build-openamp/usr/local/bin/linux_rpc_demo-shared 1
```

Enter the inputs on the host side the same gets printed on the remote side. You will see communication between the host and remote process using rpmsg calls.

Note:

`sudo` is required to run the OpenAMP demos between Linux processes, as it doesn't work on some systems if you are normal users.

1.2.4 OpenAMP Demo Docker images

The OpenAMP project maintains the following docker images to demonstrate the project. In the future docker images for use with CI will also be provided. At this time the following images are provided:

Table 1: Docker images

Name	Description
openamp/demo-lite	Just enough to run the openamp QEMU demos and lopper CLI demo
openamp/demo	Placeholder for image *to build* and run the above demos

Docker setup

You will need docker on your machine. A docker install from your Linux distribution or from the official docker project should work fine.

Warning: Arm64 host machines (like a MacBook with Apple Silicon instead of x86) are not tested at this time. Binary translation may or may not work. An aarch64 docker image will be provided in the future.

Some quick start information is given below but also checkout the docker cheat-sheet at the end of this document.

Docker quick start for Ubuntu

Example for Ubuntu 20.04 or 22.04:

```
$ sudo apt update; sudo apt install docker.io; sudo adduser $USER docker
```

Then **logout** and **log back in** in order to get the new group. You can check your groups with the command:

```
$ groups
```

Warning: If you cannot add yourself to the docker group, you can run docker with `sudo` but doing so with *any* docker image is not recommended based on general security best practice. There is nothing in the openamp images that should make them more dangerous than other images.

Your life will be easier if you are not behind a corporate firewall. However if you can pull the docker image you should be able to run the demos as they are self contained. Some of the other activities described like installing new packages etc may not work without additional effort. If needed, please checkout this tutorial

Docker for other host systems

There are a ton of tutorials for installing and using docker on the web. Some good ones include:

- Official docker documentation
- Digital Ocean tutorial

Start the container

```
you@your-machine:~$ docker run -it openamp/demo-lite
```

This will pull the openamp/demo-lite image to your machine if it is not already there. If it is there it will be used without checking if it is the latest.

It will then create and start a container based on this image and attach to it.

You will now be at a command prompt inside the container. As part of logging you in, some guidance will be printed on how to run the demos.

```
Welcome to the OpenAMP demo (lite version)
You can run the demos by using
    $ qemu-zcu102 demo1
Where demo1 is any of demo1 demo2 demo3 or demo4
You can also run the lopper demo like
    $ ./demo5/demo5

Enjoy!
dev@openamp-demo:~$
```

Run the QEMU based demos

To run demo1, use the following command:

```
dev@openamp-demo:~$ qemu-zcu102 demo1
```

This will:

- **Build a custom cpio file for the tftp/zcu102 directory**
    - This cpio will contain the contents of the base cpio file plus the contents of the my-extra-stuff directory
    - This is done every boot so changes to the my-extra-stuff directory will be used on the next boot
- **Start tmux and create multiple panes**
    - The main QEMU pane with the main UART
    - A "host" pane for container level commands
    - Two additional UART panes
- **QEMU will:**
    - Emulate the four A53 CPUs and the two R5 CPUs
    - In a separate QEMU process, emulate the microblaze based PDU
- **The A53s (in main QEMU pane) will:**
    - Run TrustedFirmware-A, and U-Boot
    - U-boot will autoboot from TFTP (provided by by QEMU from the tftp directory)
    - Load and run the kernel, dtb, and cpio based initramfs
    - present a login prompt
- The container shell pane will present a container prompt
- **The 2nd and 3rd UART panes will**
    - wait for QEMU to start
    - connect to the other UARTs of the emulated SOC
    - The 2nd UART is not used by demo 1 & 2 but is used by demo 3 & 4
    - The 3rd UART is not currently used

Let the SOC autoboot (don't stop at the U-boot count down) and then login as directed (user is root with no password). If you don't see the login prompt hit enter to get a fresh prompt. At SOC login, instructions will be printed for running the current demo.

```
Poky (Yocto Project Reference Distro) 4.0 generic-arm64 /dev/ttyPS0

(Login as root with no password)
generic-arm64 login: root
This is demo1, rpmsg examples on R5 lockstep
There are 3 sub-demos here: demo1A demo1B and demo1C
Look at them
$ cat demo1A
or just run them
$ ./demo1A

root@generic-arm64:~#
```

Demo1 contains 3 sub-demos, demo1A, demo1B and demo1C. You should look at each before running it:

```
root@generic-arm64:~# cat ./demo1A
#!/bin/sh

R5_0=/sys/class/remoteproc/remoteproc0

echo "Make sure the R5 is not running"
echo stop >$R5_0/state 2>/dev/null

echo "Set the firmware to use"
echo image_echo_test_zcu102 >$R5_0/firmware

echo "Start the R5"
echo start >$R5_0/state

echo "Now run the echo test Linux application"
echo_test
```

and then run it:

```
root@generic-arm64:~# ./demo1A
Make sure the R5 is not running
Set the firmware to use
Start the R5
[  809.815718] remoteproc remoteproc0: powering up ff9a0000.rf5ss:r5f_0
[  809.818340] remoteproc remoteproc0: Booting fw image image_echo_test_zcu102, size␣
↪610856
main():98[  op 8enamp 109.833571ib v]  remotersion: eproc0#v1.dev0buffe1.0 (r:␣
↪registered virtio0 (type 7)
main():99 Major: 1, main():100 Minor: 1, main():101 Patch: 0)
[  809.833965] remoteproc remmain()ote:103 libmetal libpro version: c0: 1.1.remot0 (e␣
↪processor ff9maina0000.rf5s():104 Major: 1, s:r5f_0 mais innow up
():105 Minor: 1, main():106 Patch: 0)
main():108 Starting application...
0 L7 registered generic bus
```

[snip]

```
sending payload number 470 of size 487
```

(continues on next page)

(continued from previous page)

```
echo test: sent : 487
received payload number 470 of size 487

sending payload number 471 of size 488
echo test: sent : 488
received payload number 471 of size 488

**************************************

Echo Test Round 0 Test Results: Error count = 0

**************************************
18 L6 rpmsg_endpoint_cb():36 shutdown message is received.
19 L7 app():82 done
[  814.610677] virtio_rpmsg_bus virtio0: 20 L6 main():129 Stopdestroyiping ang channelpp␣
↪rlication.pm..
sg-openamp-demo-channel addr 0x400
21 L7 unregistered generic bus
```

Do the same for `demo1B` and `demo1C`.

To exit QEMU do either one of these:

- In QEMU pane, hit **Ctrl-A** and then **x**
- Click the "host" shell pane and type the `exit` command

Now do the same for `demo2`, `demo3`, and `demo4`. These demos do not have sub-demos so contain a single demo script.

Run the Lopper CLI demo

The Lopper demo is fairly standalone but the container already has the needed requirements and the and the git repository has already been cloned with the correct branch. Additionally, scripts have been written to cut down the typing or cut-and-paste required.

To run this demo use:

```
dev@openamp-demo:~$ ./demo5/demo5
```

The script will first give the URL of the README file. You should open this URL in a browser and follow along.

The script will then step you through the commands in the README and let you view the various files. At the end you can look at all the files in the ~/demo5/lopper/demos/openamp directory.

Exit and clean-up the docker container

When at the docker container prompt, the exit command will stop the container and return you to your machine's prompt.

```
dev@openamp-demo:~$ exit
you@your-machine:~$
```

Now the container is not running but still exists. To check and delete it do:

```
you@your-machine:~$ docker ps -a
CONTAINER ID   IMAGE               COMMAND                  CREATED       STATUS        ␣
↪              PORTS     NAMES
nnnnnnnnnnnn   openamp/demo-lite   "/bin/sh -c 'su -l d..."   2 hours ago   Exited (0)␣
↪36 seconds ago             random_name
nnnn openamp/demo-lite "bash"  Exited (0) 2 minutes ago random_name
dev@openamp-demo:~$ docker rm random_name
```

Note: You can use tab completion to fill in the random name assigned to the container

The reusable docker image still exists on your machine. To see the images and delete the the openamp ones, you can do:

```
you@your-machine:~$ docker image list
openamp/demo-lite   latest        6ee85d920453   24 hours ago   837MB
you@your-machine:~$ docker image rm openamp/demo-lite
```

qemu-zcu102 tips and tricks

Some help is available with `qemu-zcu102 help` but it is not yet complete.

tmux mouse mode is turned on. You can:

- click in a pane to give it focus
- hold the right mouse button to show a menu (zoom and un-zoom are useful)
- the mouse scroll wheel will scroll the pane, use `q` to exit this mode
- if you don't need the 2nd or 3rd UART pane, you can kill them with the right button menu
- you can drag the pane borders to resize the panes
- you can kill the container host pane w/o stopping QEMU

The container host pane can be used with `ssh` to connect with the emulated SOC or with `scp` to transfer files. SSH configuration is already setup for the name `qemu-zcu102`.

From the container host pane:

```
dev@openamp$ ssh qemu-zcu102
root@generic-arm64:~# exit
dev@openamp$ date >date.txt; scp date.txt qemu-zcu102:
dev@openamp$ ssh qemu-zcu102 cat date.txt
```

You can manually send output to the 2nd UART like so:

```
root@generic-arm64:~# echo “Hello there” >/dev/ttyPS1
```

Docker cheat-sheet

First some tips specific to the openamp demo containers

The container is based on the standard Ubuntu 20.04 docker image. Like the Ubuntu standard images it is minimized (no man pages etc). However bash completion has been added.

There is no init system running (no systemd, no sysvinit) so no daemons are running. You cannot ssh into the container nor use scp between your host and the container. You can use `docker cp` and `docker attach` in a fashion *similar* to `scp` and `ssh` respectively.

You have no password required sudo access as the `dev` user. You can update and install packages if you wish.

All of the below are standard docker usage but may be helpful to people less familiar with docker.

You can add `--rm` to the `docker run` command to automatically delete the container when you exit. You cannot change your mind while running the container so do this only if you are sure you do not want to reuse the changes you made in the container. This will not delete the image, just the container.

To restart and reattach to a container that is stopped, do this (tab completion will help with the random name):

```
you@your-machine:~$ docker start random_name
you@your-machine:~$ docker attach random_name
```

To detach from a container without stopping it, you can use `Ctrl-p Ctrl-q`. To reattach use the attach command as show above.

`docker ps` will show all running containers and `docker ps -a` will show all containers running or stopped

1.2.5 Hypervisorless virtio binary demo (openamp/demo-lite)

A binary-only version of the hypervisorless virtio setup can be used in a containerized deployment based on the openamp/demo-lite image from Docker Hub.

```
you@your-machine:~$ docker run -it openamp/demo-lite
dev@openamp-demo:~$ qemu-zcu102 ./demo4
```

Let U-boot autoboot, don't stop it. U-boot will tftp load uEnv.txt which will tftp load the kernel, dtb, and cpio. Use root as user to login to the A53 terminal.

You can run `./demo4` or `cat demo4` and follow the commands.

When the Physical Machine Monitor starts, you will see an output similar to

```
/hvl/lkvm run --debug --vxworks --rsld --pmm --debug-nohostfs --transport mmio --shmem-
↪addr 0x37000000 --shmem-size 0x1000000 --cpus 1 --mem 128 --no-dtb --debug --rng --
↪network mode=tap,tapif=tap0,trans=mmio --vproxy
  Info: (virtio/mmio.c) virtio_mmio_init:620: virtio-mmio.devices=0x200@0x37000000␣
↪[0x4d564b4c:0x4]
[   48.008155] IPv6: ADDRCONF(NETDEV_CHANGE): tap0: link becomes ready
  Info: (virtio/mmio.c) virtio_mmio_init:620: virtio-mmio.devices=0x200@0x37000200␣
↪[0x4d564b4c:0x1]
```

You can see the results of the entropy test in the Zephyr console in the 2nd UART and interact with the system using shell commands.

E.g. Show the network interface configuration on Zephyr and ping the back-end (PetaLinux) runtime.

```
uart:~$ device list
devices:
- sys_clock (READY)
- UART_1 (READY)
- rpu_0_ipi (READY)
- virtio1 (READY)
- virtio0 (READY)
- virt-rng (READY)
  requires: virtio0
- virt-net (READY)
  requires: virtio1
uart:~$ net iface

Interface 0x78109a70 (Ethernet) [1]
===================================
Link addr : 00:00:00:00:00:00
MTU       : 1500
Flags     : AUTO_START,IPv4
Ethernet capabilities supported:
IPv4 unicast addresses (max 1):
        192.168.200.2 manual preferred infinite
IPv4 multicast addresses (max 1):
        <none>
IPv4 gateway : 0.0.0.0
IPv4 netmask : 255.255.255.0
DHCPv4 lease time : 0
DHCPv4 renew time : 0
DHCPv4 server     : 0.0.0.0
DHCPv4 requested  : 0.0.0.0
DHCPv4 state      : disabled
DHCPv4 attempts   : 0

uart:~$ net ping -c 10 192.168.200.254
PING 192.168.200.254
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=0 ttl=64 time=291 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=1 ttl=64 time=149 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=2 ttl=64 time=180 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=3 ttl=64 time=243 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=4 ttl=64 time=241 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=5 ttl=64 time=240 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=6 ttl=64 time=167 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=7 ttl=64 time=168 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=8 ttl=64 time=272 ms
28 bytes from 192.168.200.254 to 192.168.200.2: icmp_seq=9 ttl=64 time=157 ms
```

When you are ready to stop, from the QEMU pane input Ctrl-A x and the QEMU instance will terminate.

Refer to https://github.com/danmilea/hypervisorless_virtio_zcu102/blob/main/README.md for information on creating a hypervisorless virtio build environment.

1.2.6 Lopper Demonstration

1) Clone lopper, using the systemdt-linaro-demo branch

The hash is only required as our input system device tree has not been updated to the latest bus naming used in the openamp assists.

```
% git clone https://github.com/devicetree-org/lopper.git -b systemdt-linaro-demo
% cd lopper
```

Ensure that the support requirements are installed.

```
% cat Pipfile

[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
flask = "*"
flask-restful = "*"
pandas = "*"
"ruamel.yaml" = "*"
anytree = "*"
humanfriendly = "*"
```

2) Change into the lopper demo directory

```
% cd demos/openamp
```

3) Execute Lopper with openamp assists and lops

```
% export LOPPER_DIR="<path to your lopper clone>"
% $LOPPER_DIR/lopper.py -f -O scratch --enhanced --permissive \
                        -a openamp.py -a openamp_xlnx.py -a openamp-xlnx-zynq.py \
                        -i ./inputs/openamp-overlay-zynqmp.yaml \
                        -i $LOPPER_DIR/lopper/lops/lop-xlate-yaml.dts \
                        -i $LOPPER_DIR/lopper/lops/lop-a53-imux.dts -i $LOPPER_DIR/
↪lopper/lops/lop-domain-linux-a53.dts \
                        -i $LOPPER_DIR/lopper/lops/lop-openamp-versal.dts -i $LOPPER_
↪DIR/lopper/lops/lop-domain-linux-a53-prune.dts \
                        inputs/system-dt/system-top.dts linux-boot.dts
```

The outputs from this run are: linux-boot.dts and openamp-channel-info.txt

3a) linux-boot.dts

Note that this linux device tree has been created by modifying and transforming the input system device tree (system-top.dts), based on the description and values in a yaml domain file (openamp-overlay-zynqmp.yaml), transformed by assists (openamp, openampy_xlnx, openamp-xlnx-zynq) and lop files. The lop files provide unit transformations and control the overall flow of the modifications, while the assists provide more complex and context aware changes to the device tree.

We can see that nodes such as reserved-memory have been created from the vring descriptions in the yaml file.

yaml:

```
    definitions:
        OpenAMP:
             openamp_channel0_access_srams: &openamp_channel0_access_srams # used for␣
↪access in each domain
                   - dev: psu_r5_0_atcm_global
                     flags: 0
                   - dev: psu_r5_0_btcm_global
                     flags: 0

             rpu0vdev0vring0: &rpu0vdev0vring0
                 - start: 0x3ed40000
                   size: 0x2000
                   no-map: 1

             rproc0: &rproc0
                 - start: 0x3ed00000
                   size: 0x40000
                   no-map: 1


             rpu0vdev0vring1: &rpu0vdev0vring1
                 - start: 0x3ed44000
                   size: 0x4000
                   no-map: 1

             rpu0vdev0buffer: &rpu0vdev0buffer
                 - start: 0x3ed48000
                   size: 0x100000
                   no-map: 1
```

dts:

```
        reserved-memory {
                #address-cells = <0x2>;
                #size-cells = <0x2>;
                ranges;

                rproc0 {
                        no-map;
                        reg = <0x0 0x3ed00000 0x0 0x40000>;
                        phandle = <0xd0>;
                };
```

(continues on next page)

(continued from previous page)

```
                rpu0vdev0vring0 {
                        no-map;
                        reg = <0x0 0x3ed40000 0x0 0x2000>;
                        phandle = <0xd1>;
                };

                rpu0vdev0vring1 {
                        no-map;
                        reg = <0x0 0x3ed44000 0x0 0x4000>;
                        phandle = <0xd2>;
                };

                rpu0vdev0buffer {
                        no-map;
                        reg = <0x0 0x3ed48000 0x0 0x100000>;
                        compatible = "shared-dma-pool";
                        phandle = <0xd3>;
                };
        };
```

3b) openamp-channel-info.txt

This file is an export of significant values in the yaml, which were used to created nodes and properties in the dts file. They are consumed by things such as baremetal builds, or other build systems. This ensures that the dts and applications are kept in sync and agree on critical values.

```
CHANNEL0VRING0BASE="0x3ed40000"
CHANNEL0VRING0SIZE="0x2000"
CHANNEL0VRING1BASE="0x3ed44000"
CHANNEL0VRING1SIZE="0x4000"
CHANNEL0VDEV0BUFFERBASE="0x3ed48000"
CHANNEL0VDEV0BUFFERSIZE="0x100000"
CHANNEL0VDEV0BUFFERRX="FW_RSC_U32_ADDR_ANY"
CHANNEL0VDEV0BUFFERTX="FW_RSC_U32_ADDR_ANY"
CHANNEL0ELFBASE="0x3ed00000"
CHANNEL0ELFSIZE="0x40000"
CHANNEL0TO_HOST="0xff340000"
CHANNEL0TO_HOST-BITMASK="0x1000000"
CHANNEL0TO_HOST-IPIIRQVECTID="0x3f"
CHANNEL0TO_REMOTE="0xff310000"
CHANNEL0TO_REMOTE-BITMASK="0x100"
CHANNEL0TO_REMOTE-IPIIRQVECTID="0x41"
```

3c) Modify values in the yaml

We change:

- vring base and size
- access to new devices
- memory for the domain

```
% diff -u openamp-overlay-zynqmp.yaml openamp-overlay-zynqmp-dev-mem.yaml
--- openamp-overlay-zynqmp.yaml 2022-11-25 03:55:42.912355236 +0000
+++ openamp-overlay-zynqmp-dev-mem.yaml 2022-11-25 03:57:16.404274348 +0000
@@ -7,8 +7,8 @@
                flags: 0

          rpu0vdev0vring0: &rpu0vdev0vring0
-             - start: 0x3ed40000
-               size: 0x2000
+             - start: 0x00c0ffee
+               size: 0xFEEE
               no-map: 1

          rproc0: &rproc0
@@ -43,6 +43,10 @@
            # if we want to have a list merge, it should be in a list
            - dev: ipi@ff340000  # used for Open AMP RPMsg IPC
              flags: 0
+           - dev: ethernet@ff0e0000
+             flags: 0
+           - dev: ethernet@ff0d0000
+             flags: 0
            - <<+: *openamp_channel0_access_srams

        reserved-memory:
@@ -50,6 +54,12 @@
            # if we want an object / node merge, it should be like this (a map)
            <<+: [ *rpu0vdev0vring1, *rpu0vdev0vring0, *rpu0vdev0buffer, *rproc0 ]

+       memory:
+           os,type: linux
+           memory:
+             - start: 0x4000beef
+               size:  0x7c00beef
+
        domain-to-domain:
            compatible: openamp,domain-to-domain-v1
            remoteproc-relation:
```

3d) run the lopper with the new inputs

```
    % $LOPPER_DIR/lopper.py -f -O scratch --enhanced --permissive \
                            -a openamp.py -a openamp_xlnx.py -a openamp-xlnx-zynq.py \
                            -i ./inputs/openamp-overlay-zynqmp-dev-mem.yaml \
                            -i $LOPPER_DIR/lopper/lops/lop-xlate-yaml.dts \
                            -i $LOPPER_DIR/lopper/lops/lop-a53-imux.dts -i $LOPPER_DIR/
↪lopper/lops/lop-domain-linux-a53.dts \
                            -i $LOPPER_DIR/lopper/lops/lop-openamp-versal.dts -i $LOPPER_
↪DIR/lopper/lops/lop-domain-linux-a53-prune.dts \
                                    inputs/system-dt/system-top.dts linux-boot2.dts
```

We can see that:

```
% diff -u linux-boot.dts linux-boot2.dts
```

a) A new ethernet device has been made available

```
--- linux-boot.dts        2022-11-25 03:29:00.661642062 +0000
+++ linux-boot2.dts        2022-11-25 03:59:59.544134215 +0000
@@ -1209,6 +1209,25 @@
                         phandle = <0x33>;
                 };

+                gem2: ethernet@ff0d0000 {
+                        compatible = "cdns,zynqmp-gem", "cdns,gem";
+                        status = "disabled";
+                        interrupt-parent = <&gic_a53>;
+                        interrupts = <0x0 0x3d 0x4 0x0 0x3d 0x4>;
+                        reg = <0x0 0xff0d0000 0x0 0x1000>;
+                        clock-names = "pclk", "hclk", "tx_clk", "rx_clk";
+                        #address-cells = <0x1>;
+                        #size-cells = <0x0>;
+                        #stream-id-cells = <0x1>;
+                        iommus = <&smmu 0x876>;
+                        power-domains = <0x78 0x1f>;
+                        resets = <0x4 0x1f>;
+                        clocks = <&zynqmp_clk 0x1f>,
+                         <&zynqmp_clk 0x6a>,
+                         <&zynqmp_clk 0x2f>,
+                         <&zynqmp_clk 0x33>;
+                };
+
                 gem3: ethernet@ff0e0000 {
```

b) the vring base and size addresses have been adjusted

```
--- linux-boot.dts        2022-11-25 03:29:00.661642062 +0000
+++ linux-boot2.dts        2022-11-25 03:59:59.544134215 +0000

                rpu0vdev0vring0 {
                        no-map;
-                       reg = <0x0 0x3ed40000 0x0 0x2000>;
-                       phandle = <0xd1>;
+                       reg = <0x0 0xc0ffee 0x0 0xfeee>;
+                       phandle = <0xd2>;
                };
```

c) the memory node has been modified

```
--- linux-boot.dts        2022-11-25 03:29:00.661642062 +0000
+++ linux-boot2.dts        2022-11-25 03:59:59.544134215 +0000

@@ -3146,7 +3165,7 @@
        psu_ddr_0_memory: memory@0 {
                compatible = "xlnx,psu-ddr-1.0";
                device_type = "memory";
-               reg = <0x0 0x0 0x0 0x7ff00000 0x0 0x7ff00000 0x0 0x100000>;
+               reg = <0x0 0x4000beef 0x0 0x7c00beef>;
                phandle = <0x9>;
        };

d) that phandles have been adjusted to allow for new devices

                rproc0 {
                        no-map;
                        reg = <0x0 0x3ed00000 0x0 0x40000>;
-                       phandle = <0xd0>;
+                       phandle = <0xd1>;
                };
```

4) Xen extraction demo

```
% $LOPPER_DIR/lopper.py --permissive -f inputs/dt/host-device-tree.dts system-device-
↪tree-out.dts  -- \
      extract -t /bus@f1000000/serial@ff010000 -i zynqmp-firmware -x pinctrl-0 -x␣
↪pinctrl-names -x power-domains -x current-speed -x resets -x 'interrupt-controller.*' -
↪- \
      extract-xen -t serial@ff010000 -o serial@ff010000.dts
[INFO]: cb: extract( /, <lopper.LopperSDT object at 0x7f15355d7310>, 0, ['-t', '/
↪bus@f1000000/serial@ff010000', '-i', 'zynqmp-firmware', '-x', 'pinctrl-0', '-x',
↪'pinctrl-names', '-x', 'power-domains', '-x', 'current-speed', '-x', 'resets', '-x',
↪'interrupt-controller.*'] )
[INFO]: dropping masked property pinctrl-0
```

(continues on next page)

(continued from previous page)

```
[INFO]: dropping masked property power-domains
[INFO]: dropping masked property pinctrl-names
[INFO][extract-xen]: updating sdt with passthrough property
```

4a) serial@ff010000.dts is the extracted device tree

```
% cat serial@ff010000.dts

    /dts-v1/;

    / {
            #address-cells = <0x2>;
            #size-cells = <0x2>;

            passthrough {
                    compatible = "xlnx,zynqmp-zcu102-rev1.0", "xlnx,zynqmp-zcu102",
↪"xlnx,zynqmp", "simple-bus";
                    ranges;
                    #address-cells = <0x2>;
                    #size-cells = <0x2>;

                    serial@ff010000 {
                            port-number = <0x1>;
                            device_type = "serial";
                            cts-override;
                            clocks = <&clock_controller 0x39>,
                             <&clock_controller 0x1f>;
                            clock-names = "uart_clk", "pclk";
                            reg = <0x0 0xff010000 0x0 0x1000>;
                            interrupts = <0x0 0x16 0x4>;
                            interrupt-parent = <0xfde8>;
                            status = "okay";
                            compatible = "cdns,uart-r1p12", "xlnx,xuartps";
                            u-boot,dm-pre-reloc;
                            xen,path = "/axi/serial@ff010000";
                            xen,force-assign-without-iommu = <0x1>;
                            xen,reg = <0x0 0xff010000 0x0 0x1000 0x0 0xff010000>;
                    };

                    zynqmp-firmware {
                            phandle = <0xc>;
                            #power-domain-cells = <0x1>;
                            method = "smc";
                            u-boot,dm-pre-reloc;
                            compatible = "xlnx,zynqmp-firmware";
                            extracted,path = "/firmware/zynqmp-firmware/clock-controller
↪";

                            clock_controller: clock-controller {
                                    phandle = <0x3>;
```

(continues on next page)

(continued from previous page)

```
                                clock-names = "pss_ref_clk", "video_clk", "pss_alt_
↪ref_clk", "aux_ref_clk", "gt_crx_ref_clk";
                                clocks = <&pss_ref_clk>,
                                 <&video_clk>,
                                 <&pss_alt_ref_clk>,
                                 <&aux_ref_clk>,
                                 <&gt_crx_ref_clk>;
                                compatible = "xlnx,zynqmp-clk";
                                #clock-cells = <0x1>;
                                u-boot,dm-pre-reloc;
                        };
                };

                pss_ref_clk: pss_ref_clk {
                        phandle = <0x6>;
                        clock-frequency = <0x1fc9350>;
                        #clock-cells = <0x0>;
                        compatible = "fixed-clock";
                        u-boot,dm-pre-reloc;
                        extracted,path = "/pss_ref_clk";
                };

                video_clk: video_clk {
                        phandle = <0x7>;
                        clock-frequency = <0x1fc9f08>;
                        #clock-cells = <0x0>;
                        compatible = "fixed-clock";
                        u-boot,dm-pre-reloc;
                        extracted,path = "/video_clk";
                };

                pss_alt_ref_clk: pss_alt_ref_clk {
                        phandle = <0x8>;
                        clock-frequency = <0x0>;
                        #clock-cells = <0x0>;
                        compatible = "fixed-clock";
                        u-boot,dm-pre-reloc;
                        extracted,path = "/pss_alt_ref_clk";
                };

                aux_ref_clk: aux_ref_clk {
                        phandle = <0x9>;
                        clock-frequency = <0x19bfcc0>;
                        #clock-cells = <0x0>;
                        compatible = "fixed-clock";
                        u-boot,dm-pre-reloc;
                        extracted,path = "/aux_ref_clk";
                };

                gt_crx_ref_clk: gt_crx_ref_clk {
                        phandle = <0xa>;
                        clock-frequency = <0x66ff300>;
```

(continues on next page)

(continued from previous page)

```
                                #clock-cells = <0x0>;
                                compatible = "fixed-clock";
                                u-boot,dm-pre-reloc;
                                extracted,path = "/gt_crx_ref_clk";
                        };
                };
        };
```

4b) system-device-tree-out.dts for the modified system device tree with passthrough option

```
% grep -C4 xen,passthrough system-device-tree-out.dts

                        pinctrl-0 = <0x3c>;
                        cts-override;
                        device_type = "serial";
                        port-number = <0x1>;
                        xen,passthrough;
                };

                usb0@ff9d0000 {
                        #address-cells = <0x2>;
```

4c) extract an ethernet device

We use the output system device tree from the previous run, as the input for this run.

```
% $LOPPER_DIR/lopper.py --permissive -f system-device-tree-out.dts system-device-tree-
↪out-final.dts  -- \
                          extract -o extracted_tree.dts -p -t ethernet@ff0e0000 -i␣
↪zynqmp-firmware -x 'interrupt-controller.*' -x power-domains -x current-speed -- \
                          extract-xen -v -t ethernet@ff0e0000 -o xen-passthrough-eth.dts

[INFO]: cb: extract( /, <lopper.LopperSDT object at 0x7efd688df340>, 0, ['-o',
↪'extracted_tree.dts', '-p', '-t', 'ethernet@ff0e0000', '-i', 'zynqmp-firmware', '-x',
↪'interrupt-controller.*', '-x', 'power-domains', '-x', 'current-speed'] )
[INFO]: dropping masked property power-domains
[INFO][extract-xen]: ethernet@ff0e0000 interrupt parent found, updating
[INFO][extract-xen]: smmu@fd800000 interrupt parent found, updating
[INFO][extract-xen]: updating sdt with passthrough property
[INFO][extract-xen]: reg found: reg = <0x0 0xff0e0000 0x0 0x1000>; copying and extending␣
↪to xen,reg
[INFO][extract-xen]: deleting node (referencing node was removed): /extracted/
↪smmu@fd800000

% grep -C4 xen,passthrough system-device-tree-out-final.dts

                        phy-mode = "rgmii-id";
                        xlnx,ptp-enet-clock = <0x0>;
                        local-mac-address = [FF FF FF FF FF FF];
```

(continues on next page)

(continued from previous page)

```
                        phandle = <0x22>;
                        xen,passthrough;

                        ethernet-phy@c {
                                reg = <0xc>;
                                ti,rx-internal-delay = <0x8>;
--
                        pinctrl-0 = <0x3c>;
                        cts-override;
                        device_type = "serial";
                        port-number = <0x1>;
                        xen,passthrough;
                };

                usb0@ff9d0000 {
                        #address-cells = <0x2>;
```

1.3 Contributing to the OpenAMP Project

1.3.1 Release Cycle

- 6 month release cycle aligned with Ubuntu (xx.04 and xx.10)
- (feature freeze) release branch cut 1 month before release target
- Maintainence releases are left open-ended for now

1.3.2 Roadmap discussion and publication

- Feature freeze period of a release used for roadmap discussions for next release
- Contributers propose features posted and discussed on mailing list
- Maintainer collects accepted proposals
- Maintainer posts list of development tasks, owners, at open of release cycle

1.3.3 Patch process

- Patches posted on the mailing list for review
- Pull request on github once review cycles are complete
- Maintainer ensures a minimum of 1 week review window prior to merge

1.3.4 Platform maintainers

- Platform code refers to sections of code that apply to specific vendor's hardware or operating system platform
- Platform maintainers represent OS or hardware platform's interests in the community
- Every supported OS or hardware platform must have a platform maintainer (via addition to MAINTAINERS file in code base), or patches may not be merged.
- Support for an OS or hardware platform may be removed from the code base if the platform maintainer is non-responsive for more than 2 release cycles
- Responsible for verification and providing feedback on posted patches
- Responsible to ACK platform support for releases (No ACK => platform not supported in the release)

1.3.5 Push rights

- Push rights restricted to the Core Team
- Generally exercised by the maintainers for each repository
- Maintainers manage delegation between themselves

1.4 Links

- The OpenAMP project home page: https://www.openampproject.org/
- **OpenAMP mailing lists: https://lists.openampproject.org/mailman/listinfo**

  Note: Before getting the mailing lists, we used this Google Group. The Google Group is only listed here for reference to older content. Please use the mailing lists.

CHAPTER

TWO

OPENAMP PROTOCOL DETAILS

2.1 Asymmetric Multiprocessing Intro

An embedded AMP system is characterized by multiple homogeneous and/or heterogeneous processing cores integrated into one System-on-a-Chip (SoC). Examples include:

- The Xilinx MPSoC that has four ARM Cortex-A53, two ARM Cortex-R5, and potentially a number of MicroBlaze cores.
- The NXP i.MX6SoloX/i.MX7d SoCs that utilizes ARM Cortex-A9 and ARM Cortex-M4F cores
- The Texas Instruments TI AM57x SoCs that have dual ARM Cortex A15, dual ARM Cortex M4, and C66x DSP cores.

These cores typically run independent instances of homogeneous and/or heterogeneous software environments, such as Linux, RTOS, and Bare Metal that work together to achieve the design goals of the end application. While Symmetric Multiprocessing (SMP) operating systems allow load balancing of application workload across homogeneous processors present in such AMP SoCs, asymmetric multiprocessing design paradigms are required to leverage parallelism from the heterogeneous cores present in the system.

Increasingly, today's multicore applications require heterogeneous processing power. Heterogeneous multicore SoCs often have one or more general purpose CPUs (for example, dual ARM Cortex A9 cores on Xilinx Zynq) with DSPs and/or smaller CPUs and/or soft IP (on SoCs such as Xilinx Zynq MPSOC). These specialized CPUs, as compared to the general purpose CPUs, are typically dedicated for demand-driven offload of specialized application functionality to achieve maximum system performance. Systems developed using these types of SoCs, characterized by heterogeneity in both hardware and software, are generally termed as AMP systems.

Other reasons to run heterogeneous software environments (e.g. multi-OS) include:

- **Needs for multiple environments with different characteristics**
  - Real-time (RTOS) and general purpose (i.e. Linux)
  - Safe/Secure environment and regular environment
  - GPL and non-GPL environments
- **Integration of code written for multiple environments**
  - Legacy OS and new OS

In AMP systems, it is typical for software running on a master to bring up software/firmware contexts on a remote on a demand-driven basis and communicate with them using IPC mechanisms to offload work during run time. The participating master and remote processors may be homogeneous or heterogeneous in nature.

A master is defined as the CPU/software that is booted first and is responsible for managing other CPUs and their software contexts present in an AMP system. A remote is defined as the CPU/software context managed by the master software context present.

2.2 Components and Capabilities

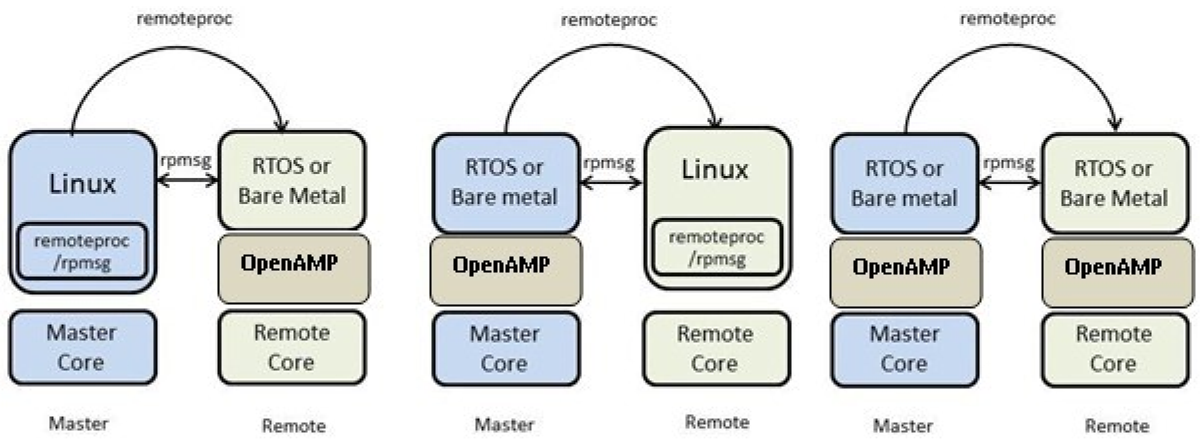
The key components and capabilities provided by the OpenAMP Framework include:

- remoteproc — This component allows for the Life Cycle Management (LCM) of remote processors from software running on a master processor. The remoteproc API provided by the OpenAMP Framework is compliant with the remoteproc infrastructure present in upstream Linux 3.4.x kernel onward. The Linux remoteproc infrastructure and API was first implemented by Texas Instruments.
- RPMsg – The RPMsg API enables Inter Processor Communications (IPC) between independent software contexts running on homogeneous or heterogenous cores present in an AMP system. This API is compliant with the RPMsg bus infrastructure present in upstream Linux 3.4.x kernel onward. The Linux RPMsg bus and API infrastructure was first implemented by Texas Instruments.

Texas Instruments' remoteproc and RPMsg infrastructure available in the upstream Linux kernel today enable the Linux applications running on a master processor to manage the life cycle of remote processor/firmware and perform IPC with them. However, there is no open- source API/software available that provides similar functionality and interfaces for other possible software contexts (RTOS- or bare metal-based applications) running on the remote processor to communicate with the Linux master. Also, AMP applications may require RTOS- or bare metal-based applications to run on the master processor and be able to manage and communicate with various software environments (RTOS, bare metal, or even Linux) on the remote processor.

The OpenAMP Framework fills these gaps. It provides the required LCM and IPC infrastructure from the RTOS and bare metal environments with the API conformity and functional symmetry available in the upstream Linux kernel. As in upstream Linux, the OpenAMP Framework's remoteproc and RPMsg infrastructure uses virtio as the transport layer/abstraction.

The following figure shows the various software environments/configurations supported by the OpenAMP Framework. As shown in this illustration, the OpenAMP Framework can be used with RTOS or bare metal contexts on a remote processor to communicate with Linux applications (in kernel space or user space) or other RTOS/bare metal-based applications running on the master processor through the remoteproc and RPMsg components. Managing Remote Processes with the OpenAMP framework



The OpenAMP Framework also serves as a stand-alone library that enables RTOS and bare metal applications on a master processor to manage the life cycle of remote processor/firmware and communicate with them using RPMsg.

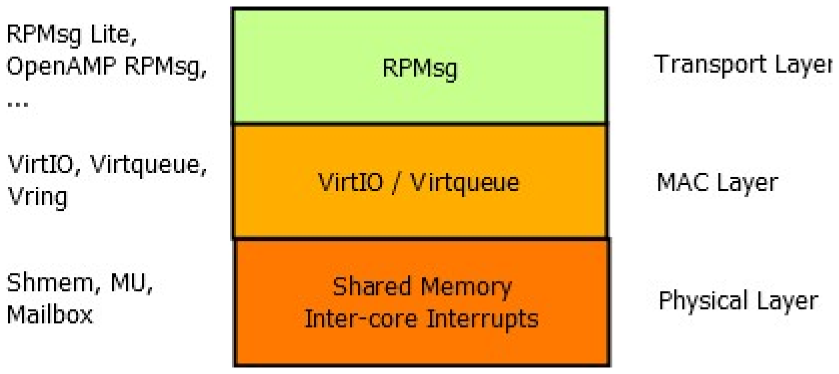
In addition to providing a software framework/API for LCM and IPC, the OpenAMP Framework supplies a proxy infrastructure that provides a transparent interface to remote contexts from Linux user space applications running on the master processor. The proxy application hides all the logistics involved in bringing-up the remote software context and its shutdown sequence. In addition, it supports RPMsg-based Remote Procedure Calls (RPCs) from remote context. A retargeting API available from the remote context allows C library system calls such as "_open", "_close", "_read", and "_write" to be forwarded to the proxy application on the master for service. For more information on this infrastructure and its capabilities, see Figure 5-1 on page 60. In addition to the core capabilities, the OpenAMP Framework contains abstraction layers (porting layer) for migration to different software environments and new target processors/platforms.

2.3 RPMsg Messaging Protocol

In asymmetric multiprocessor systems, the most common way for different cores to cooperate is to use a shared memory-based communication. There are many custom implementations, which means that the considered systems cannot be directly interconnected. Therefore, this document's aim is to offer a standardization of this communication based on existing components (RPMsg, VirtIO).

2.3.1 Protocol Layers

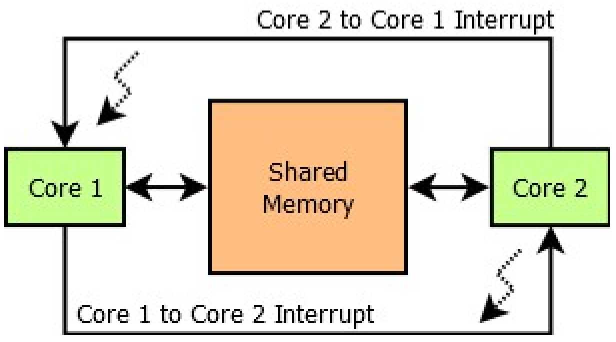
The whole communication implementation can be separated in three different ISO/OSI layers - Transport, Media Access Control and Physical layer. Each of them can be implemented separately and for example multiple implementations of the Transport Layer can share the same implementation of the MAC Layer (VirtIO) and the Physical Layer. Each layer is described in following sections.



Physical Layer – Shared Memory

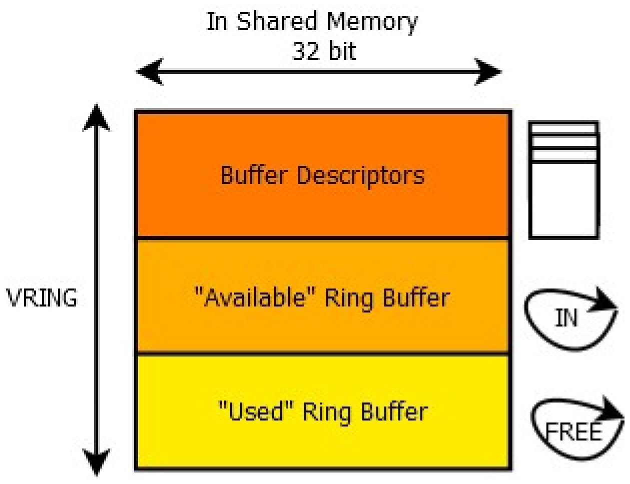
The solution proposed in this document requires only two basic hardware components - shared memory (accessible by both communicating sides) and inter-core interrupts (in a specific configuration optional). The minimum configuration requires one interrupt line per communicating core meaning two interrupts in total. This configuration is briefly presented in figure at the beginning of this section. It is to be noticed that no inter-core synchronization hardware element such as inter-core semaphore, inter-core queue or inter-core mutex is needed! This is thanks to the nature of the virtqueue, which uses single-writer-single-reader circular buffering. (As defined in next subsection)

In case the "used" and "avail" ring buffers have a bit set in their configuration flags field, the generation of interrupts can be completely suppressed - in such a configuration, the interrupts are not necessary. However both cores need to poll the "ring" and "used" ring buffers for new incoming messages, which may not be optimal.



Media Access Layer - VirtIO

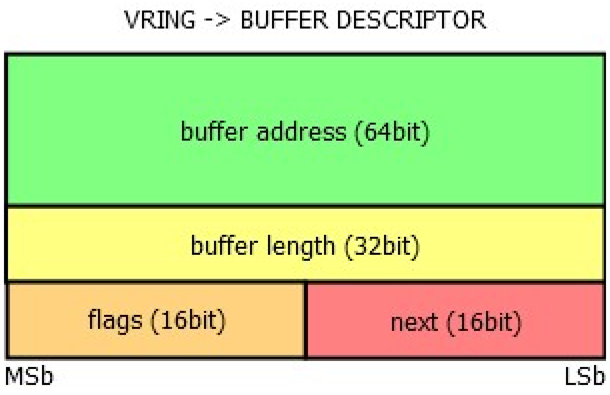
This layer is the key part of the whole solution - thanks to this layer, there is no need for inter-core synchronization. This is achieved by a technique called single-writer single-reader circular buffering, which is a data structure enabling multiple asynchronous contexts to interchange data.



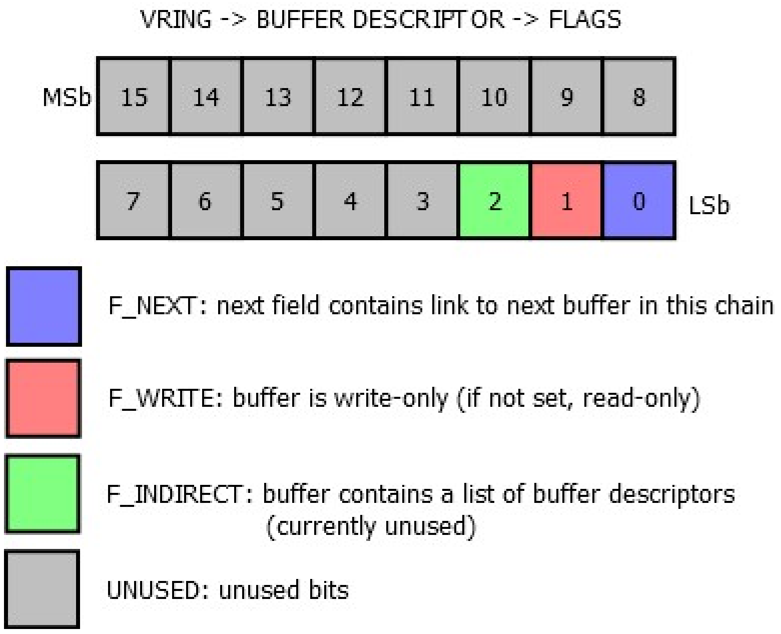
This technique is however applicable only in core-to-core configuration, not in core-to-multicore configuration, since in such a case, there would be multiple writers to the “IN” ring buffer. This would require a synchronization element, [such as a semaphore?], which is not desirable.

The above shown picture describes the vring component. Vring is composed of three elementary parts - buffer descriptor pool, the “available” ring buffer (or input ring buffer) and the “used” ring buffer (or free ring buffer). All three elements are physically stored in the shared memory.

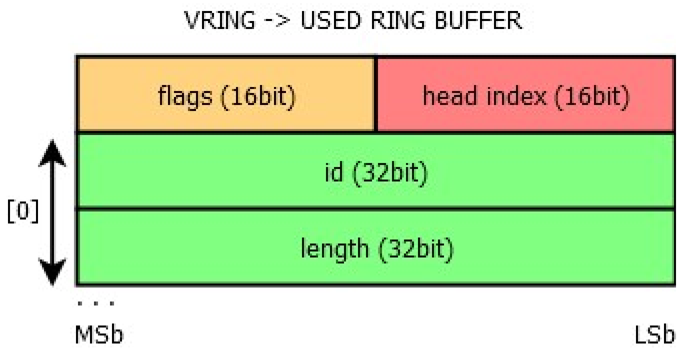
Each buffer descriptor contains a 64-bit buffer address, which holds an address to a buffer stored in the shared memory (as seen physically by the “receiver” or host of this vring), its length as a 32-bit variable, 16-bit flags field and 16-bit link to the next buffer descriptor. The link is used to chain unused buffer descriptors and to chain descriptors, which have the F_NEXT bit set in the flags field to the next descriptor in the chain.



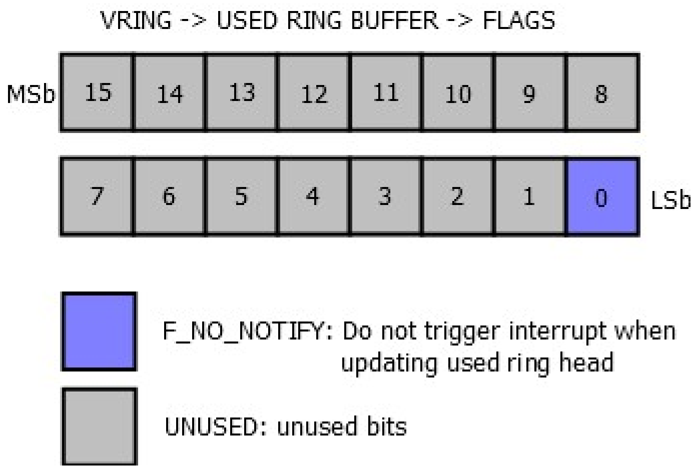
The input ring buffer contains its own flags field, where only the 0th bit is used - if it is set, the "writer" side should not be notified, when the "reader" side consumes a buffer from the input or "avail" ring buffer. By default the bit is not set, so after the reader consumes a buffer, the writer should be notified by triggering an interrupt. The next field of the input ring buffer is the index of the head, which is updated by the writer, after a buffer index containing a new message is written in the ring[x] field.


VRING -> BUFFER DESCRIPTOR -> FLAGS
MSb
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
LSb
F_NEXT: next field contains link to next buffer in this chain
F_WRITE: buffer is write-only (if not set, read-only)
F_INDIRECT: buffer contains a list of buffer descriptors (currently unused)
UNUSED: unused bits

The last part of the vring is the "used" ring buffer. It contains also a flags field and only the 0th bit is used - if set, the writer side will not be notified when the reader updates the head index of this free ring buffer. The following picture shows the ring buffer structure. The used ring buffer differs from the avail ring buffer. For each entry, the length of the buffer is stored as well.



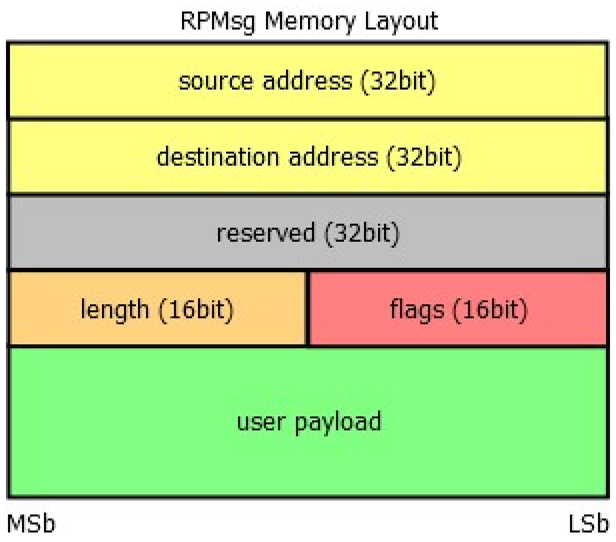
Both “used” and “avail” ring buffers have a flags field. Its purpose is mainly to tell the writer whether he should interrupt the other core when updating the head of the ring. The same bit is used for this purpose in both “used” and “avail” ring buffers:


VRING -> USED RING BUFFER -> FLAGS
MSb
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
LSb
F_NO_NOTIFY: Do not trigger interrupt when updating used ring head
UNUSED: unused bits

Transport Layer - RPMsg

RPMsg Header Definition

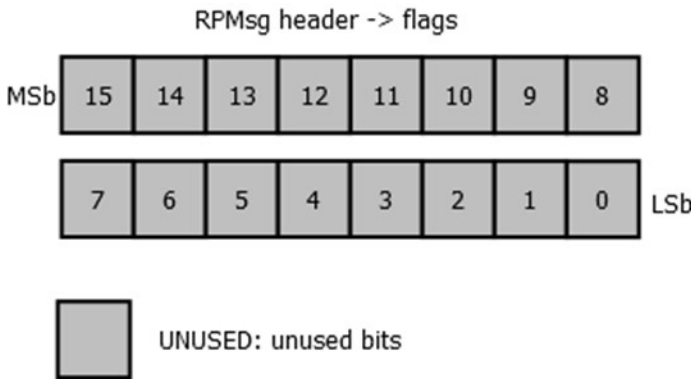
Each RPMsg message is contained in a buffer, which is present in the shared memory. This buffer is pointed to by the address field of a buffer descriptor from vring’s buffer descriptor pool. The first 16 bytes of this buffer are used internally by the transport layer (RPMsg layer). The first word (32bits) is used as an address of the sender or source endpoint, next word is the address of the receiver or destination endpoint. There is a reserved field for alignment reasons (RPMsg header is thus 16 bytes aligned). Last two fields of the header are the length of the payload (16bit) and a 16-bit flags field. The reserved field is not used to transmit data between cores and can be used internally in the RPMsg implementation. The user payload follows the RPMsg header.



Special consideration should be taken if an alignment greater than 16 bytes is required; however, this is not typical for a shared memory, which should be fast and is therefore often not cached (alignment greater than 8 bytes is not needed at all).

Flags Field

The flags field of the RPMsg header is currently unused by RPMsg and is reserved. Any propositions for what this field could be used for is welcome. It could be released for application use, but this can be considered as inconsistent - RPMsg header would not be aligned and the reserved field would be therefore useless.



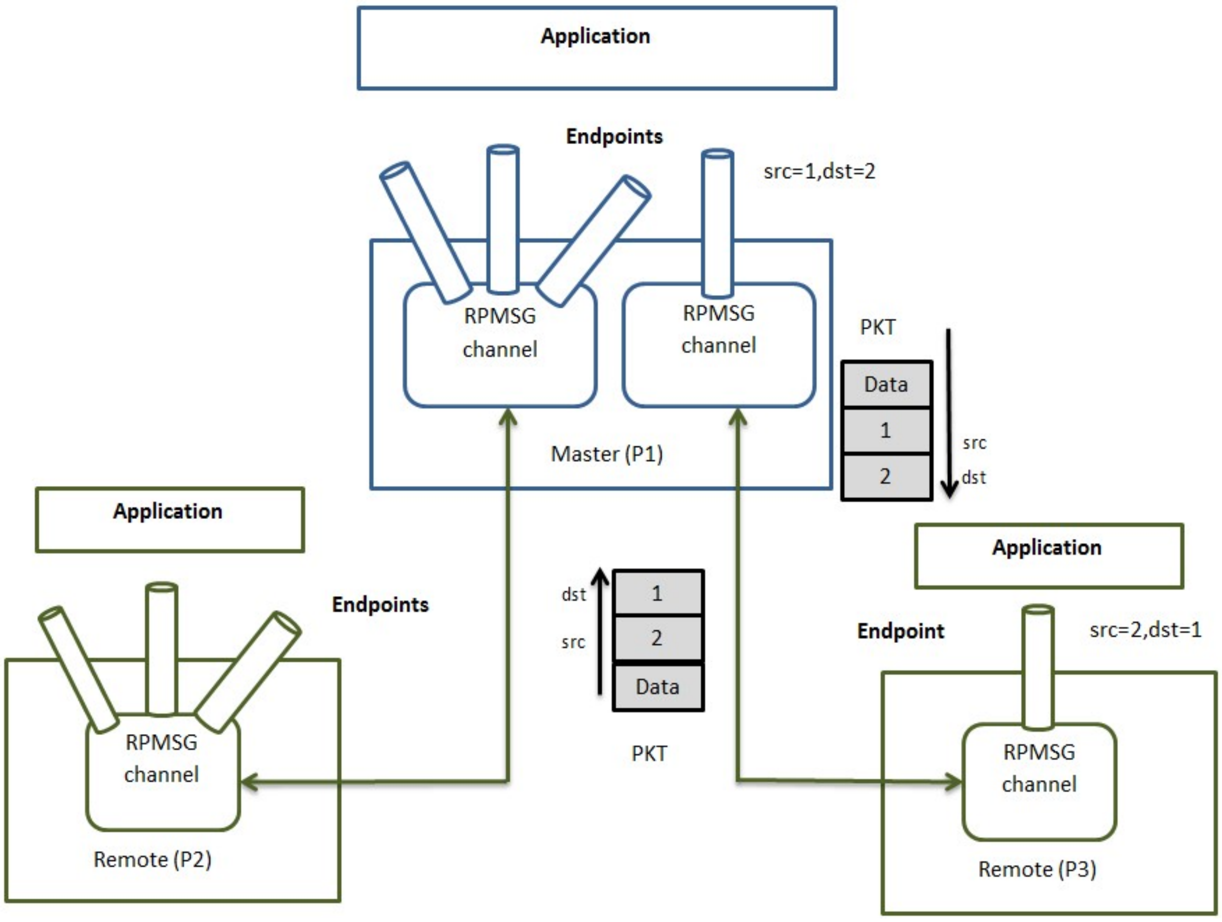
RPMsg Channel

Every remote core in RPMsg component is represented by RPMsg device that provides a communication channel between master and remote, hence RPMsg devices are also known as channels RPMsg channel is identified by the textual name and local (source) and destination address. The RPMsg framework keeps track of channels using their names.

RPMsg Endpoint

RPMsg endpoints provide logical connections on top of RPMsg channel. It allows the user to bind multiple rx callbacks on the same channel.

Every RPMsg endpoint has a unique src address and associated call back function. When an application creates an endpoint with the local address, all the further inbound messages with the destination address equal to local address of endpoint are routed to that callback function. Every channel has a default endpoint which enables applications to communicate without even creating new endpoints.

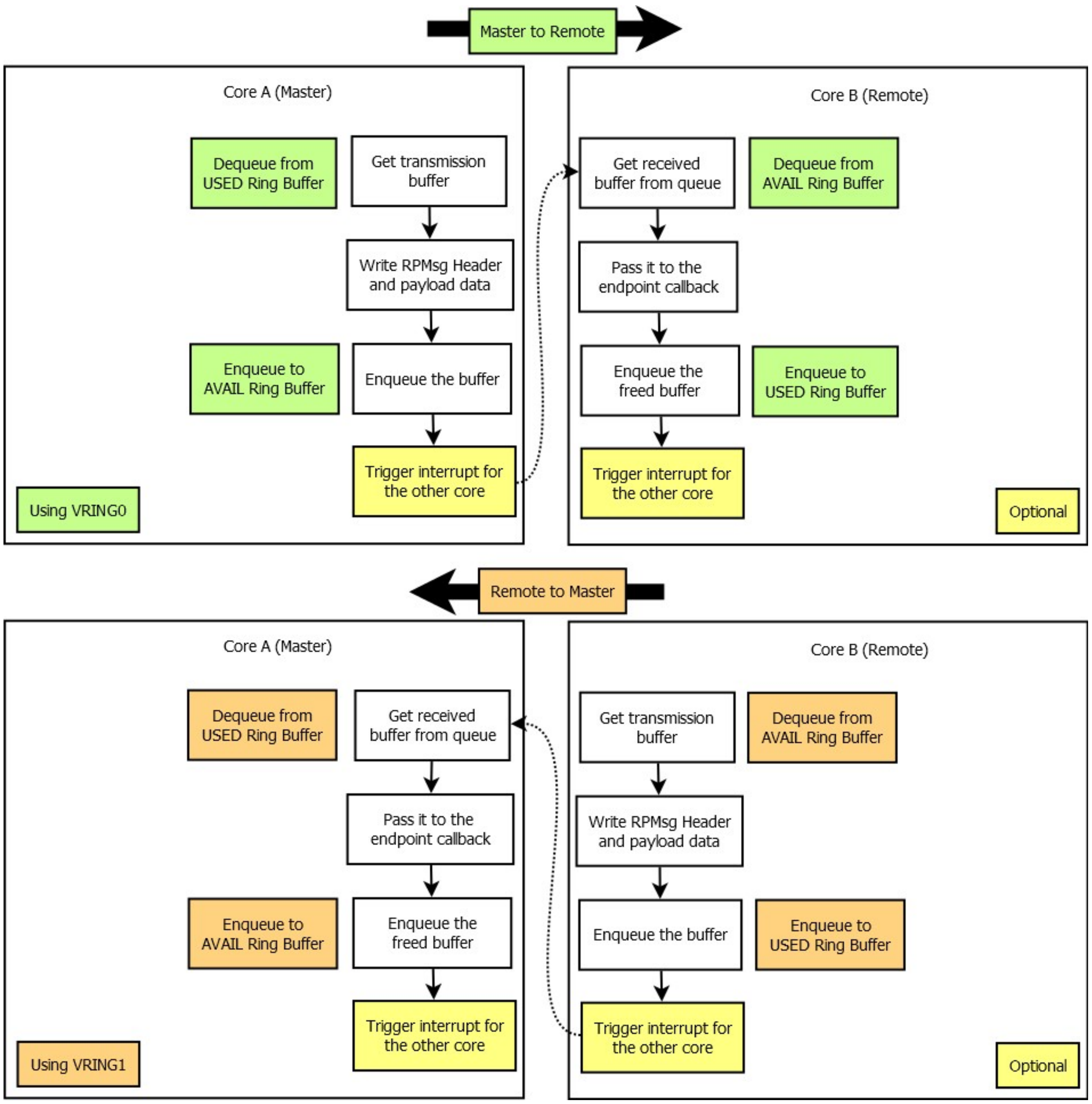


2.3.2 RPMsg Protocol Limitations

The RPMSG document has the concept of the static channel but it is not implemented in upstream Linux and OpenAMP. Please see https://www.kernel.org/doc/Documentation/rpmsg.txt. The protocol must define connection sequence when channel is created statically. No synchronization point is defined by the RPMsg after which both sides can communicate reliably with each other. In the current protocol, at startup, the master sends notification to remote to let it know that it can receive name service announcement. However, master does not consider the fact that if the remote is ready to handle notification at this point in time.

2.4 RPMsg Communication Flow

The following figure describes the sequence used for transaction of a RPMsg message from one core to the other. The sequence differs according to the roles of core A and core B. In the figure above, core A is the Master and core B is the Remote. The Master core allocates buffers used for the transmission from the "used" ring buffer of a vring, writes RPMsg Header and application payload to it and then enqueues it to the "avail" ring buffer.


Master to Remote
Core A (Master)
Dequeue from USED Ring Buffer
Get transmission buffer
Write RPMsg Header and payload data
Enqueue to AVAIL Ring Buffer
Enqueue the buffer
Trigger interrupt for the other core
Using VRING0
Core B (Remote)
Get received buffer from queue
Dequeue from AVAIL Ring Buffer
Pass it to the endpoint callback
Enqueue the freed buffer
Enqueue to USED Ring Buffer
Trigger interrupt for the other core
Optional
Remote to Master
Core A (Master)
Dequeue from USED Ring Buffer
Get received buffer from queue
Pass it to the endpoint callback
Enqueue to AVAIL Ring Buffer
Enqueue the freed buffer
Trigger interrupt for the other core
Using VRING1
Core B (Remote)
Get transmission buffer
Dequeue from AVAIL Ring Buffer
Write RPMsg Header and payload data
Enqueue the buffer
Enqueue to USED Ring Buffer
Trigger interrupt for the other core
Optional

The Remote core gets the received RPMsg buffer from the “avail” ring buffer, processes it and then returns it back to the “used” ring buffer. When the Remote core is sending a message to the Master core, “avail” and “used” ring buffers role are swapped.

The reason for swapping the roles of the ring buffers comes from the fact, that the Master core works as a Buffer Provider. The Buffer Provider has a complete control of memory management and shared memory allocation. Obviously, when the Master core, or Buffer Provider, does not fill the “avail” ring buffer of VRING1 (Orange), the Remote core is unable to send a message to the master. This can be used to throttle the communication generated by the Remote core. It is to be noticed, that the master always dequeues from the “used” ring buffer and enqueues to the “avail” ring buffer. For the remote, the situation is inverse. The triggering of interrupts is optional. It is governed by the flags in “used” and “avail” ring buffers.

2.5 Life Cycle Management

The LCM(Life Cycle Management) component of OpenAMP is known as remoteproc. Remoteproc APIs provided by the OpenAMP Framework allow software applications running on the master processor to manage the life cycle of a remote processor and its software context. A complete description of the remoteproc workflow and APIs are provided.

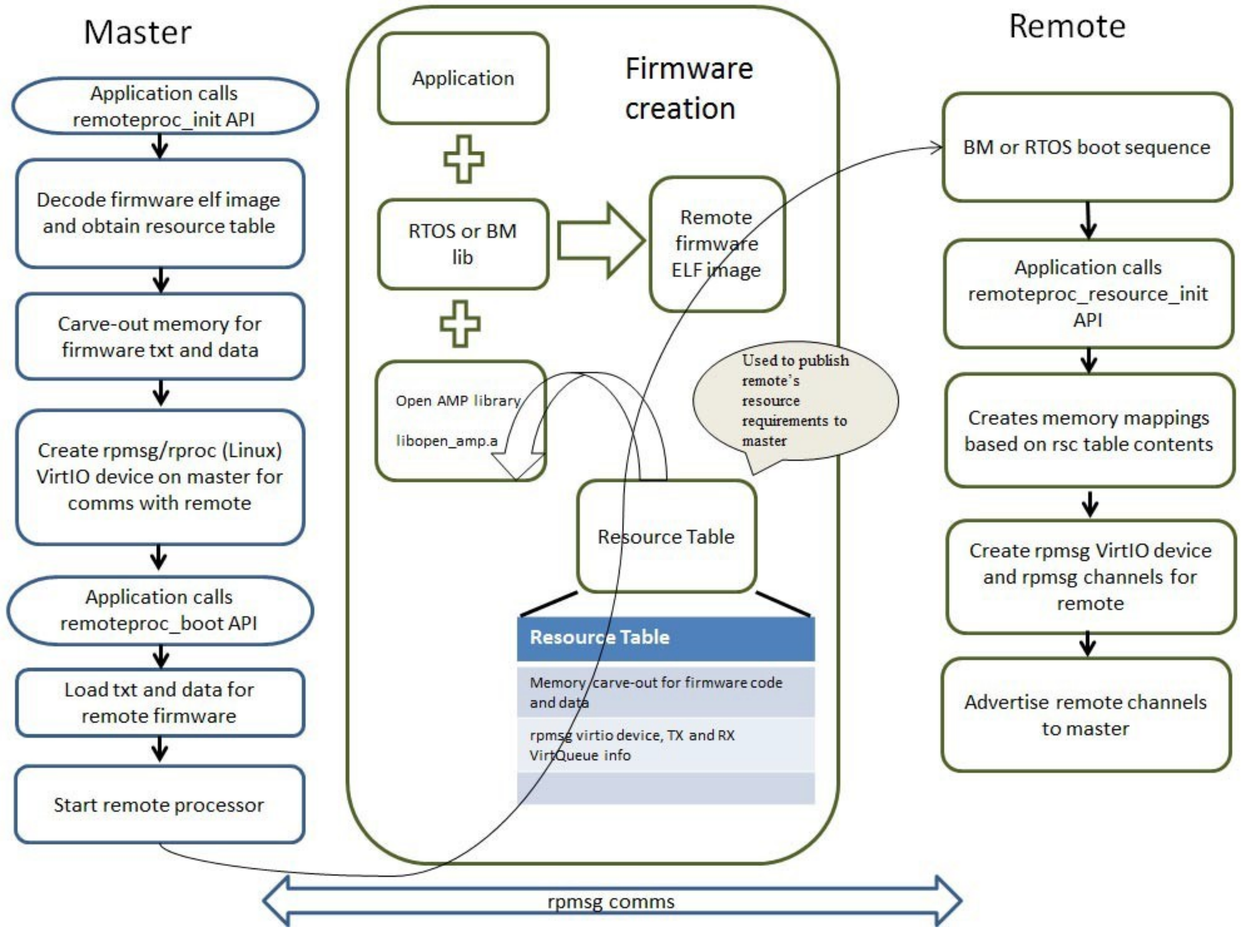
2.5.1 LCM Overview

The remoteproc APIs provide life cycle management of remote processors by performing five essential functions.

- Allow the master software applications to load the code and data sections of the remote firmware image to appropriate locations in memory for in-place execution
- Release the remote processor from reset to start execution of the remote firmware
- Establish RPMsg communication channels for run-time communications with the remote context
- Shut down the remote software context and processor when its services are not needed
- Provide an API for use in the remote application context that allows the remote applications to seamlessly initialize the remoteproc system on the remote side and establish communication channels with the master context

The remoteproc component currently supports Executable and Linkable Format (ELF) for the remote firmware; however, the framework can be easily extended to support other image formats. The remote firmware image publishes the system resources it requires to remoteproc on the master using a statically linked resource table data structure. The resource table data structure contains entries that define the system resources required by the remote firmware (for example, contiguous memory carve-outs required by remote firmware's code and data sections), and features/functionality supported by the remote firmware (like virtio devices and their configuration information required for RPMsg-based IPC).

The remoteproc APIs on the master processor use the information published through the firmware resource table to allocate appropriate system resources and to create virtio devices for IPC with the remote software context. The following figure illustrates the resource table usage.


Master
Application calls remoteproc_init API
Decode firmware elf image and obtain resource table
Carve-out memory for firmware txt and data
Create rpmsg/rproc (Linux) VirtIO device on master for comms with remote
Application calls remoteproc_boot API
Load txt and data for remote firmware
Start remote processor
Application
Firmware creation
RTOS or BM lib
Remote firmware ELF image
Open AMP library
libopen_amp.a
Used to publish remote's resource requirements to master
Resource Table
Resource Table
Memory carve-out for firmware code and data
rpmsg virtio device, TX and RX VirtQueue info
Remote
BM or RTOS boot sequence
Application calls remoteproc_resource_init API
Creates memory mappings based on rsc table contents
Create rpmsg VirtIO device and rpmsg channels for remote
Advertise remote channels to master
rpmsg comms

When the application on the master calls to the remoteproc_init API, it performs the following:

- Causes remoteproc to fetch the firmware ELF image and decode it
- Obtains the resource table and parses it to handle entries
- Carves out memory for remote firmware before creating virtio devices for communications with remote context

The master application then performs the following actions:

1. Calls the remoteproc_boot API to boot the remote context
2. Locates the code and data sections of the remote firmware image
3. Releases the remote processor to start execution of the remote firmware.

After the remote application is running on the remote processor, the remote application calls the remoteproc_resource_init API to create the virtio/RPMsg devices required for IPC with the master context. Invocation of this API causes remoteproc on the remote context to use the rpmsg name service announcement feature to advertise the rpmsg channels served by the remote application.

The master receives the advertisement messages and performs the following tasks:

1. Invokes the channel created callback registered by the master application
2. Responds to remote context with a name service acknowledgement message

After the acknowledgement is received from master, remoteproc on the remote side invokes the RPMsg channel-created callback registered by the remote application. The RPMsg channel is established at this point. All RPMsg APIs can be used subsequently on both sides for run time communications between the master and remote software contexts.

To shut down the remote processor/firmware, the remoteproc_shutdown API is to be used from the master context. Invoking this API with the desired remoteproc instance handle asynchronously shuts down the remote processor. Using this API directly does not allow for graceful shutdown of remote context.

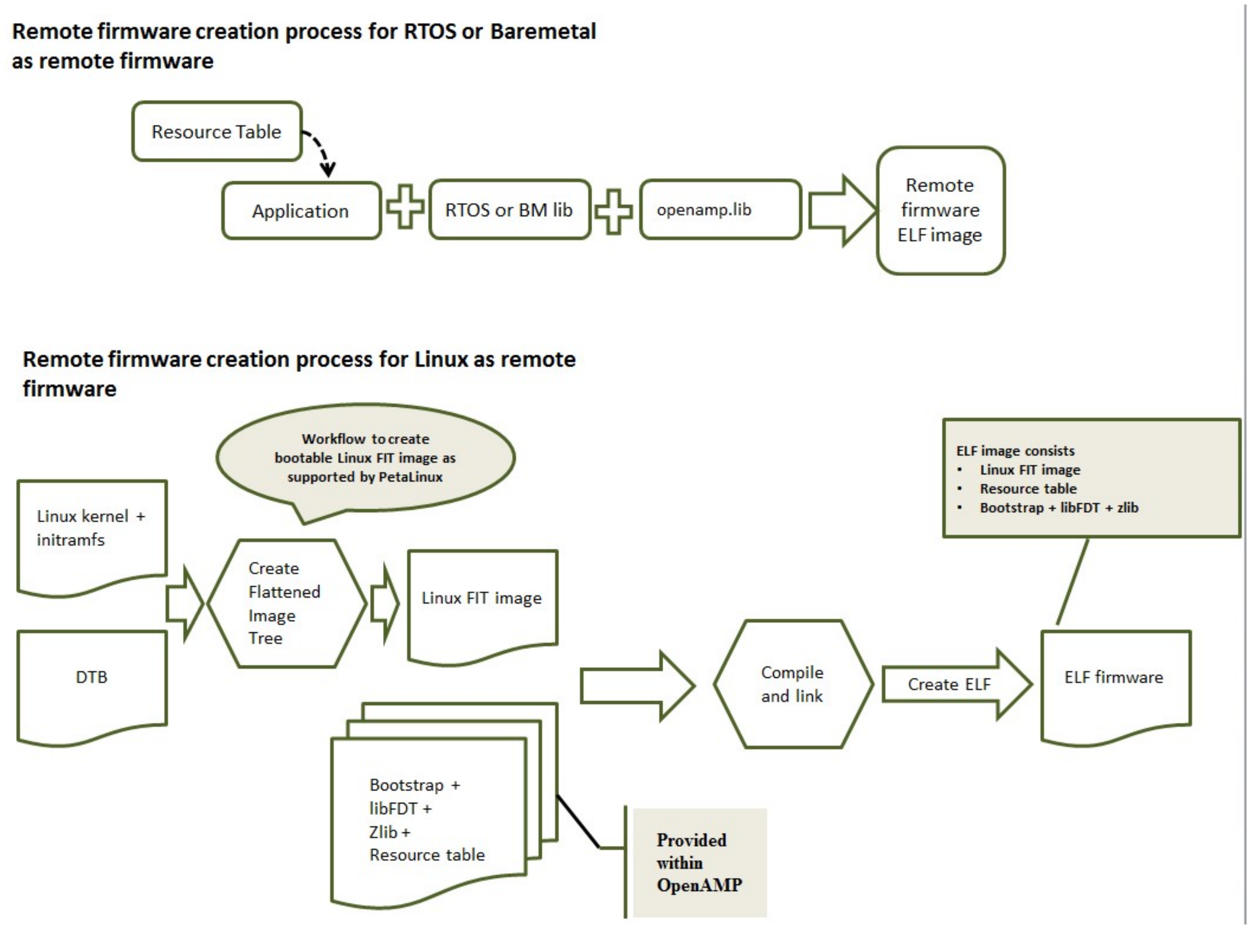
For gracefully bringing down the remote context, the following steps can be performed:

1. The master application sends an application-specific shutdown message to the remote context
2. The remote application cleans up application resources, sends a shutdown acknowledge to master, and invokes remoteproc_resource_deinit API to deinitialize remoteproc on the remote side.
3. On receiving the shutdown acknowledge message, the master application invokes the remoteproc_shutdown API to shut down the remote processor and de-initialize remoteproc using remoteproc_deinit on its side.

2.5.2 Creation and Boot of Remote Firmware Using remoteproc

You can create and boot remote firmware for Linux, RTOS, and bare metal-based remote applications using remoteproc. The following procedure provides general steps for creating and executing remote firmware on a supported platform.

The following figure illustrates the remote firmware creation process.


Remote firmware creation process for RTOS or Baremetal as remote firmware
Resource Table
Application
RTOS or BM lib
openamp.lib
Remote firmware ELF image
Remote firmware creation process for Linux as remote firmware
Workflow to create bootable Linux FIT image as supported by PetaLinux
Linux kernel + initramfs
DTB
Create Flattened Image Tree
Linux FIT image
Bootstrap + libFDT + Zlib + Resource table
Provided within OpenAMP
Compile and link
Create ELF
ELF firmware
ELF image consists
Linux FIT image
Resource table
Bootstrap + libFDT + zlib

Defining the Resource Table and Creating the Remote ELF Image

Creating a remote image through remoteproc begins by defining the resource table and creating the remote ELF image.

Procedure

1. Define the resource table structure in the application. The resource table must minimally contain carve-out and VirtIO device information for IPC.

As an example, please refer to the resource table defined in the bare metal remote echo test application at <open_amp>/apps/machine/zynq/rsc_table.c. The resource table contains entries for memory carve-out and virtio device resources. The memory carve-out entry contains info like firmware ELF image start address and size. The virtio device resource contains virtio device features, vring addresses, size, and alignment information. The resource table data structure is placed in the resource table section of remote firmware ELF image using compiler directives.

2. After defining the resource table and creating the OpenAMP Framework library, link the remote application with the RTOS or bare metal library and the OpenAMP Framework library to create a remote firmware ELF image capable of in-place execution from its pre-determined memory region. (The pre-determined memory region is determined according to guidelines provided by section.)

3. For remote Linux, step 1 describes modifications to be made to the resource table. The previous flow figures shows the high level steps involved in creation of the remote Linux firmware image. The flow shows to create a Linux FIT image that encapsulates the Linux kernel image, Device Tree Blob (DTB), and initramfs.

The user applications and kernel drivers required on the remote Linux context could be built into the initramfs or moved to the remote root file system as needed after boot. The FIT image is linked along with a boot strap package provided within the OpenAMP Framework. The bootstrap implements the functionality required to decode the FIT image (using libfdt), uncompress the Linux kernel image (using zlib) and locate the kernel image, initramfs, and DTB in RAM. It can also set up the ARM general purpose registers with arguments to boot Linux, and transfer control to the Linux entry point.

Making Remote Firmware Accessible to the Master

After creating the remote firmware's ELF image, you need to make it accessible to remoteproc in the master context.

Procedure

1. If the RTOS- or bare metal-based master software context has a file system, place this firmware ELF image in the file system.
2. Implement the get_firmware API in firmware.c (in the <open_amp>/lib/common/ directory) to fetch the remote firmware image by name from the file system.
3. For AMP use cases with Linux as master, place the firmware application in the root file system for use by Linux remoteproc platform drivers.

In the OpenAMP Framework reference port to Zynq ZC702EVK, the bare metal library used by the master software applications do not include a file system. Therefore, the remote image is packaged along with the master ELF image. The remote ELF image is converted to an object file using "objcpy" available in the "GCC bin-utils". This object file is further linked with the master ELF image.

The remoteproc component on the master uses the start and end symbols from the remote object files to get the remote ELF image base and size. Since the logistics used by the master to obtain a remote firmware image is deployment specific, the config_get_firmware API in firmware.c in the <open_amp>/lib/common/ directory implements all the logistics described in this procedure to enable the OpenAMP Framework remoteproc on the master to obtain the remote firmware image.

You can now use the remoteproc APIs.

2.6 System Wide Considerations

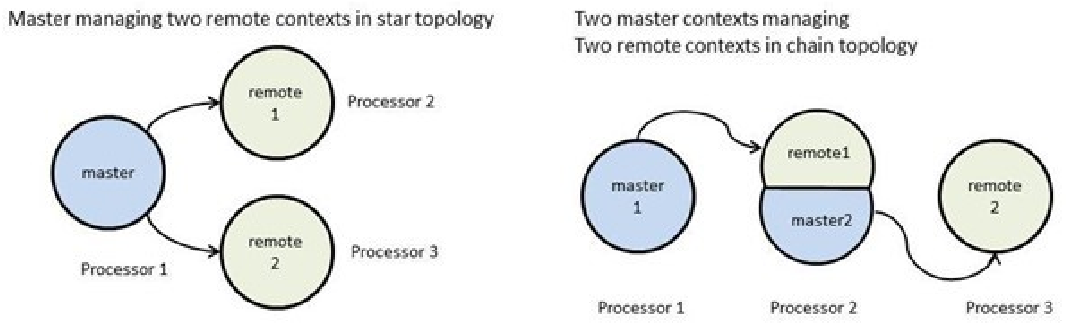
AMP systems could either be supervised (using a hypervisor to enforce isolation and resource virtualization) or unsupervised (modifying each participating software context to ensure best- effort isolation and cooperative usage of shared resources). With unsupervised AMP systems, there is no strict isolation or supervision of shared resource usage.

Take the following system-wide considerations into account to develop unsupervised AMP systems using the OpenAMP framework.

- Determine system architecture/topology

The OpenAMP framework implicitly assumes master-slave (remote) system architecture. The topology for the master-slave (remote) architecture should be determined; either star, chain, or a combination. The following figure shows some simple use cases.

- Case 1 — A single master software context on processor 1 controlling life cycle and communicating with two independent remote software contexts on processors 2 and 3, in star topology,
- Case 2 — Master software context 1 on processor 1 brings up remote software context 1 on processor 2. This context acts as master software context 2 for remote software context 2 on processor 3, in chain topology.



- Determine system and IO resource partitioning

Various OSs, RTOSs, and bare metal environments have their own preferred mechanisms for discovering platform-specific information such as available RAM memory, available peripheral IO resources (their memory-mapped IO region), clocks, interrupt resources, and so forth.

For example, the Linux kernel uses device trees and bare metal environment typically define platform-specific device information in headers or dedicated data structures that would be compiled into the application.

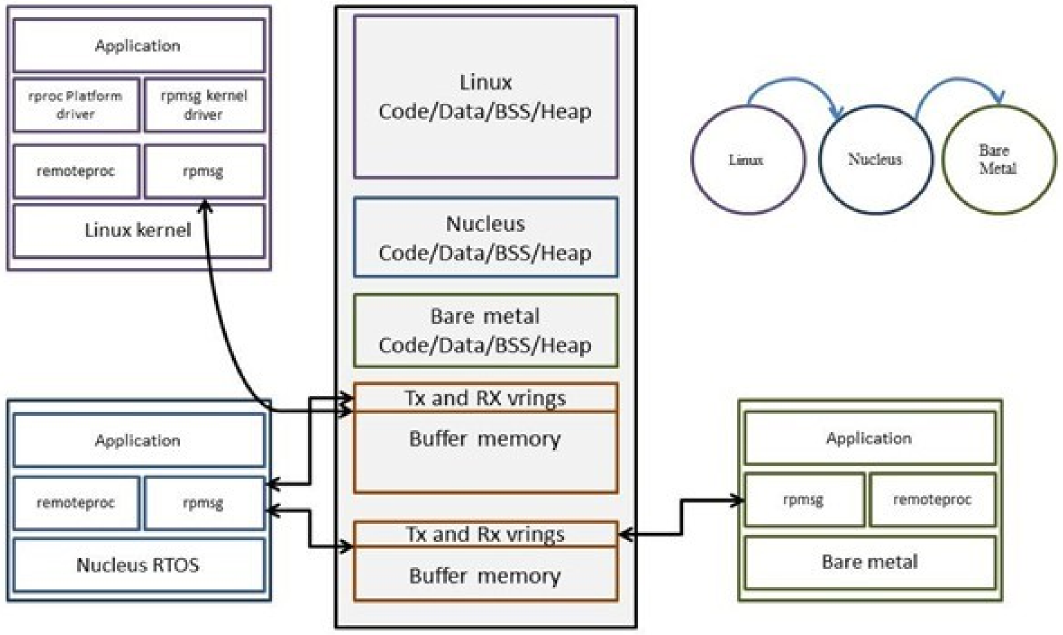
To ensure mutually-exclusive usage of unshared system (memory) and IO resources (peripherals) between the participating software environments in an AMP system, you are required to partition the resources so that each software environment is only aware of the resources that are available to it. This would involve, for example, removing unused resource nodes and modifying the available memory definitions from the device tree sources, platform definition files, headers, and so forth, to ensure best-effort partitioning of system resources.

- Determine memory layout

For the purpose of this description, assume you are using the Zynq SOC used in AMP system architecture with SMP Linux running on the dual Cortex A9 cores, and a RTOS on one instance of Microblaze soft core, and bare metal on another instance of Microblaze soft core in the fabric.

To develop an AMP system using the OpenAMP Framework, it is important to determine the memory regions that would be owned and shared between each of the participating software environments in the AMP system. For example, in a configuration such as this, the memory address ranges owned (for code/data/bss/heap) by each participating OS or bare metal context, and the shared memory regions to be used by IPC mechanisms (virtio rings and memory for data buffers) needs to be determined. Memory alignment requirements should be taken into consideration while making this determination.

The following image illustrates the memory layout for Linux master/RTOS-based remote application, and RTOS-based master/bare metal-based remote application in chain configuration. Determinint the Memory Layout in an AMP System



- Ensure cooperative usage of shared resources between software environments in the AMP system

For the purpose of this discussion, assume you are using a Linux master/bare metal- based remote system configuration.

The interrupt controller is typically a shared resource in multicore SoCs. It is general practice for OSs to reset and initialize (clear and disable all interrupts) the interrupt controller during their boot sequence given the general assumption that the OS would own the entire system. This will not work in AMP systems; if an OS in remote software context resets and initializes the interrupt controller, it would catastrophically break the master software contexts run time since the master context could already be using the interrupt controller to manage its interrupt resources. Therefore, remote software environments should be patched such that they cooperatively use the interrupt controller (for example, do not reset/clear/disable all interrupts blindly but initialize only the interrupts that belong to the remote context). Ensure the timer peripheral used by the

remote OS/RTOS context is different from the one used by the master software context so the individual run-times do not interfere with each other.

2.7 Resource Table Evolution

2.7.1 Overview

This page is to collect all ideas for the evolution of the remoteproc resource table.

2.7.2 Needs

Compatibility with Linux kernel

```
The evolution should be done in cooperation with Linux remoteproc community.
```

Review current resource table fields

Review the current version resource table to confirm fields relevance (e.g name fields). The aim is to keep resource table as small as possible…

64 bit addresses

Parameter passing

In order to pass parameters to the remote application, it could be useful to have a new resource type for that.

Evolutive virtio dev features

Currently, field for virtio dev features is 32 bits in rsc table. However, some virtio dev now have bits > 32 bit.

Pointer to Device Tree

vdev buffer management

- Add possibility to provide DA to fix the vdev buffers memory region and size (carveout?)
- Allow to specify the size of the buffers, depending on the direction.

Trace evolution

Improve the trace mechanism to increase depth.

2.7.3 Enhancement

To confirm a need…

Define resource table ownership

The resource table has to be managed by only one core, the master. To break the master slave relationship, a field that define the ownership of the resource table, could be added

Include the resource table size

Add size information in header to avoid to parse the whole resource table to retrieve the length (memory allocation/mapping)

New resource to provide processors states

Add resource that defines a structure to share the processor states. This can be used by some systems to manage low power and crash mechanisms.

2.7.4 Mechanisms

Discussion of how the add the new capabilities to the resource table while providing back and forward compatibility.

Resource table version number

The existing resource table definition has a version number. This number could be incremented for a new format.

The issue with the method is that it is all or nothing and does not allow new firmware to be used with an old kernel.

clementleger: IMHO, this is not a real "issue", if new firmware requires new capabilities in resource table, then it will use the new format and expect a master that support such features. If the firmware does not need them, then it will stick with old format. Having a backward compatibility seems mandatory however a forward compatibility seems really limitating.

Multiple resource tables of different versions could be included but this is rather bulky and awkward and a new method would need to be defined for marking and locating the various versions.

Per item version number

Resource table item IDs are currently 32 bits. It has been suggested that this is a very large range for this purpose. One idea would be to subdivide the 32 bits into fields and designate some bits to be an item type specific version number.

clementleger: This is a bit clunky. If we are modifying the resource table, it would be better to add a new field to handled such cases. Actually, all resources have reserved fields. IMHO, these fields should be used for versionning if using a new version (they were currently 0) so they are well suited to be used as a versionning field. But this probably be discussed on the mailing list. the vdev_vring resources are tied to rsc_vdev so it will mosty probably be used in conjunction with them and the versionning of vdev_rsc will apply to them.

This would allow multiple versions of the same type to be included in the resourse table and the kernel could look for the greatest version that it understands. The bit fields makes this logic easier than if unrelated 32 bit values were used.

Per item Priority

Item ID bit fields could also/instead be used to define what a consumer should do it if does not understand the Item ID. A priority of "optional" means that the consumer can ignore the Item if it does not understand it and a priority of "required" means that the consumer should refuse to load firmware that contains this item ID. Other priorities might be defined but at least these two would make sense.

vdev buffer management

Should we add a new resource tied to the vdev one to define a DA and buffer size? Do we need to define independent memory region for both direction (P2P)? Buffer size and number of buffers should depend on the direction.

Traces

The existing tracing mechanism is relying on a circular buffer. The depth of the trace is the size of the buffer as no mechanism exists to extract the traces before overflow. Proposal is to implement trace extraction based on a flip flop buffer with notifications (mailbox notifyID). This would allow the main processor to extract the traces in time to fill its own logs buffer/file. The Impact in the resource table would be a new resource structure or the addition of a "trace method" field in existing resource structure.

Firmware publishes multiple versions of Resource table

TODO

2.8 OpenAMP Design Docs

CHAPTER

THREE

OPENAMP LIBRARIES USER GUIDE

3.1 Data Structures

3.2 Porting GuideLine

The OpenAMP Framework uses libmetal to provide abstractions that allow for porting of the OpenAMP Framework to various software environments (operating systems and bare metal environments) and machines (processors/platforms). To port OpenAMP for your platform, you will need to:

- add your system environment support to libmetal,
- implement your platform specific remoteproc driver.
- define your shared memory layout and specify it in a resource table.

3.2.1 Add System/Machine Support in Libmetal

User will need to add system/machine support to lib/system/<SYS>/ directory in libmetal repository. OpenAMP requires the following libmetal primitives:

- alloc, for memory allocation and memory free
- io, for memory mapping. OpenAMP required memory mapping in order to access vrings and carved out memory.
- mutex
- sleep, at the moment, OpenAMP only requires microseconds sleep as when OpenAMP fails to get a buffer to send messages, it will call this function to sleep and then try again.
- init, for libmetal initialization.

Please refer to lib/system/generic/ when adding RTOS support to libmetal.

libmetal uses C11/C++11 stdatomics interface for atomic operations, if you use a different compiler to GNU gcc, you may need to implement the atomic operations defined in lib/compiler/gcc/atomic.h.

3.2.2 Platform Specific Remoteproc Driver

User will need to implement platform specific remoteproc driver to use remoteproc life cycle management APIs. The remoteproc driver platform specific functions are defined in this file: lib/include/openamp/remoteproc.h. Here are the remoteproc functions needs platform specific implementation.

- init(), instantiate the remoteproc instance with platform specific config parameters.
- remove(), destroy the remoteproc instance and its resource.
- mmap(), map the memory speficified with physical address or remote device address so that it can be used by the application.
- handle_rsc(), handler to the platform specific resource which is specified in the resource table.
- config(), configure the remote processor to get it ready to load application.
- start(), start the remote processor to run the application.
- stop(), stop the remote processor from running but not power it down.
- shutdown(), shutdown the remote processor and you can power it down.
- notify(), notify the remote processor.

3.2.3 Platform Specific Porting to Use Remoteproc to Manage Remote Processor

User will need to implement the above platform specific remoteproc driver functions. After that, user can use remoteproc APIs to run application on a remote processor. E.g.:

```
#include <openamp/remoteproc.h>

/* User defined remoteproc operations */
extern struct remoteproc_ops rproc_ops;

/* User defined image store operations, such as open the image file, read
 * image from storage, and close the image file.
 */

extern struct image_store_ops img_store_ops;
/* Pointer to keep the image store information. It will be passed to user
 * defined image store operations by the remoteproc loading application
 * function. Its structure is defined by user.
 */
void *img_store_info;

struct remoteproc rproc;

void main(void)
{
      /* Instantiate the remoteproc instance */
      remoteproc_init(&rproc, &rproc_ops, &private_data);

      /* Optional, required, if user needs to configure the remote before
       * loading applications.
       */
      remoteproc_config(&rproc, &platform_config);
```

(continues on next page)

(continued from previous page)

```
      /* Load Application. It only supports ELF for now. */
      remoteproc_load(&rproc, img_path, img_store_info, &img_store_ops, NULL);

      /* Start the processor to run the application. */
      remoteproc_start(&rproc);

      /* ... */

      /* Optional. Stop the processor, but the processor is not powered
       * down.
       */
      remoteproc_stop(&rproc);

      /* Shutdown the processor. The processor is supposed to be powered
       * down.
       */
      remoteproc_shutdown(&rproc);

      /* Destroy the remoteproc instance */
      remoteproc_remove(&rproc);
}
```

3.2.4 Platform Specific Porting to Use RPMsg

RPMsg in OpenAMP implementation uses VirtIO to manage the shared buffers. OpenAMP library provides remoteproc VirtIO backend implementation. You don't have to use remoteproc backend. You can implement your VirtIO backend with the VirtIO and RPMsg implementation in OpenAMP. If you want to implement your own VirtIO backend, you can refer to the [remoteproc VirtIO backend implementation]: https://github.com/OpenAMP/open-amp/blob/master/lib/remoteproc/remoteproc_virtio.c

Here are the steps to use OpenAMP for RPMsg communication:

```
#include <openamp/remoteproc.h>
#include <openamp/rpmsg.h>
#include <openamp/rpmsg_virtio.h>

/* User defined remoteproc operations for communication */
sturct remoteproc rproc_ops = {
      .init = local_rproc_init;
      .mmap = local_rproc_mmap;
      .notify = local_rproc_notify;
      .remove = local_rproc_remove;
};

/* Remoteproc instance. If you don't use Remoteproc VirtIO backend,
 * you don't need to define the remoteproc instance.
 */
struct remoteproc rproc;

/* RPMsg VirtIO device instance. */
```

(continues on next page)

(continued from previous page)

```
struct rpmsg_virtio_device rpmsg_vdev;

/* RPMsg device */
struct rpmsg_device *rpmsg_dev;

/* Resource Table. Resource table is used by remoteproc to describe
 * the shared resources such as vdev(VirtIO device) and other shared memory.
 * Resource table resources definition is in the remoteproc.h.
 * Examples of the resource table can be found in the OpenAMP repo:
 *  - apps/machine/zynqmp/rsc_table.c
 *  - apps/machine/zynqmp_r5/rsc_table.c
 *  - apps/machine/zynq7/rsc_table.c
 */
void *rsc_table = &resource_table;

/* Size of the resource table */
int rsc_size = sizeof(resource_table);

/* Shared memory metal I/O region. It will be used by OpenAMP library
 * to access the memory. You can have more than one shared memory regions
 * in your application.
 */
struct metal_io_region *shm_io;

/* VirtIO device */
struct virtio_device *vdev;

/* RPMsg shared buffers pool */
struct rpmsg_virtio_shm_pool shpool;

/* Shared buffers */
void *shbuf;

/* RPMsg endpoint */
struct rpmsg_endpoint ept;

/* User defined RPMsg name service callback. This callback is called
 * when there is no registered RPMsg endpoint is found for this name
 * service. User can create RPMsg endpoint in this callback. */
void ns_bind_cb(struct rpmsg_device *rdev, const char *name, uint32_t dest);

/* User defined RPMsg endpoint received message callback */
void rpmsg_ept_cb(struct rpmsg_endpoint *ept, void *data, size_t len,
                uint32_t src, void *priv);

/* User defined RPMsg name service unbind request callback */
void ns_unbind_cb(struct rpmsg_device *rdev, const char *name, uint32_t dest);

void main(void)
{
      /* Instantiate remoteproc instance */
      remoteproc_init(&rproc, &rproc_ops);
```

(continues on next page)

(continued from previous page)

```
    /* Mmap shared memories so that they can be used */
    remoteproc_mmap(&rproc, &physical_address, NULL, size,
                    <memory_attributes>, &shm_io);

    /* Parse resource table to remoteproc */
    remoteproc_set_rsc_table(&rproc, rsc_table, rsc_size);

    /* Create VirtIO device from remoteproc.
     * VirtIO device master will initiate the VirtIO rings, and assign
     * shared buffers. If you running the application as VirtIO slave, you
     * set the role as VIRTIO_DEV_SLAVE.
     * If you don't use remoteproc, you will need to define your own VirtIO
     * device.
     */
    vdev = remoteproc_create_virtio(&rproc, 0, VIRTIO_DEV_MASTER, NULL);

    /* This step is only required if you are VirtIO device master.
     * Initialize the shared buffers pool.
     */
    shbuf = metal_io_phys_to_virt(shm_io, SHARED_BUF_PA);
    rpmsg_virtio_init_shm_pool(&shpool, shbuf, SHARED_BUFF_SIZE);

    /* Initialize RPMsg VirtIO device with the VirtIO device */
    /* If it is VirtIO device slave, it will not return until the master
     * side set the VirtIO device DRIVER OK status bit.
     */
    rpmsg_init_vdev(&rpmsg_vdev, vdev, ns_bind_cb, io, shm_io, &shpool);

    /* Get RPMsg device from RPMsg VirtIO device */
    rpmsg_dev = rpmsg_virtio_get_rpmsg_device(&rpmsg_vdev);

    /* Create RPMsg endpoint. */
    rpmsg_create_ept(&ept, rdev, RPMSG_SERVICE_NAME, RPMSG_ADDR_ANY,
                     rpmsg_ept_cb, ns_unbind_cb);

    /* If it is VirtIO device master, it sends the first message */
    while (!is_rpmsg_ept_read(&ept)) {
            /* check if the endpoint has binded.
             * If not, wait for notification. If local endpoint hasn't
             * been bound with the remote endpoint, it will fail to
             * send the message to the remote.
             */
            /* If you prefer to use interrupt, you can wait for
             * interrupt here, and call the VirtIO notified function
             * in the interrupt handling task.
             */
            rproc_virtio_notified(vdev, RSC_NOTIFY_ID_ANY);
    }
    /* Send RPMsg */
    rpmsg_send(&ept, data, size);
```

(continues on next page)

(continued from previous page)

```
	do {
		/* If you prefer to use interrupt, you can wait for
		 * interrupt here, and call the VirtIO notified function
		 * in the interrupt handling task.
		 * If vdev is notified, the endpoint callback will be
		 * called.
		 */
		rproc_virtio_notified(vdev, RSC_NOTIFY_ID_ANY);
	} while(!ns_unbind_cb_is_called && !user_decided_to_end_communication);

	/* End of communication, destroy the endpoint */
	rpmsg_destroy_ept(&ept);

	rpmsg_deinit_vdev(&rpmsg_vdev);

	remoteproc_remove_virtio(&rproc, vdev);

	remoteproc_remove(&rproc);
}
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search