Sample openamppdf doc

version

Tammy Leino

October 11, 2022

Contents

Welcome to the OpenAMP Project Documentation **1**

OpenAMP Project 1

Project Overview 1

OpenAMP Intro 1

Operating Environments 1

OpenAMP Capabilities 2

OpenAMP Guidelines 2

Current Work 2

Remoteproc Sub-Group 2

Introduction 2

Communications 2

Mailing list 2

Meetings 2

Documentation 2

Placeholder for Bill to populate 2

OpenAMP Introduction from Linaro Connect HKG18 3

Future work 3

System Device Tree Sub-Group 3

Introduction 3

Communications 3

Mailing list 3

Meetings 3

Documentation 3

System DT intro presentation given at Linaro Connect SAN19 3

Placeholder for Stefano to populate 3

Lopper 3

Future work 3

Application Services Sub-Group 3

Introduction 3

Communications 4

Mailing list 4

Meetings 4

Documentation 4

Placeholder 4

Future work 4

Inclusive Language and Biased Terms Sub-Group 4

replace master with main 4

remoteproc context 4

virto context 4

OpenAMP CI Sub-Group 5

IPC Sub-Group 5

System Reference / End-to-End Example Sub-Group 5

What is OpenAMP System Reference / End-to-end example? 5

Communications 5

Repository 5
Documentation 5
Samples and demos 5
Milestones 5
Future work 5
Efforts 5
High level plan for Xilinx Software Stack 6
Older Topics 6
Formation of OpenAMP project 6
Samples and Demos 6
System Reference Samples and Demos 6
Echo Test 6
ST Echo Test Example 6
Xilinx Echo Test Example 6
Multi RPMSG services demo 6
STM32MP157C/F-DK2 board 6
Prerequisite 6
Installation 6
Generate the stm32mp15 image 7
Install stm32mp1_distrib_oss dunfell 7
Initialize the Open Embedded build environment 7
Build stm32mp1_distrib_oss image 7
Flash stm32mp1_distrib_oss 8
Generate the Zephyr firmware image 8
Install meta-zephyr Honister version 8
Initialize the OpenEmbedded build environment 8
Build the Zephyr image 8
Install the Zephyr binary on the sdcard 9
Demos 9
Start the demo environment 9
Demo 1: rpmsg-client-sample device 9
Demo 2: rpmsg-tty device 10
Demo 3: dynamic creation/release of a rpmsg-tty device 10
Demo 4: rpmsg-char device 11
Demo 5: Multi endpoints demo using rpmsg-ctrl device 12
Maintenance 13
Contributing to the OpenAMP Project 13
Meeting Notes 14
Future Work 14
Remoteproc Sub-Group Future Work 14
HW description (Xilinx, TI, ST, Mentor) 14
System Resource Management – (ST, Xilinx, TI, Cypress) 14
OpenAMP project support for Virtio spec (Xilinx, TI, ST, Mentor) 14
Remoteproc Virtio driver 14
Big data – (Xilinx, TI, ST) 15
Kernel remoteproc CI patches - Loic 15

OpenAMP kernel staging tree - Bill 15
Remoteproc/rpmsg CI & regression test – (Xilinx, TI, ST) 15
Secure coprocessor support – Loic 16
Linux RPMSG only mode - Bill (TI,) 16
RPMSG robustness - Bill (TI,ST) 16
Early coprocessor boot – late attach, detach - Loic (ST,TI) 16
Lifecycle management and Trusted & Embedded Base Boot Requirement (T/E-BBR) – Etsam 17
Lifecycle management with Linux remote – Etsam 17
Rpmsg protocol documentation for remote – Etsam 17
MCU – MCU issues, rpsmg only - Mark 17
64 bits support - Clément 18
Misc I/O over VirtIO/RPMSG - Loic, Bill 18
Improve Coprocessor debug capabilities - Loic 19
Past presentations and TODO lists 19
Higher Level Services Sub-Group Future Work 19
System Device Tree Sub-Group Future Work 19
Standardizing Hypercalls Sub-Group 19
HMM (Heterogeneous Memory Management) 19
Use Cases 19
zone device and memory hotplug (on arm64) 20
hmm address space mirroring w/ rproc 20
Hardware Description 20
Problems 20
Potential solutions 20
Others interested in this problem? 20
Links 20
OpenAMP Protocol Details 21
Asymmetric Multiprocessing Intro 21
Components and Capabilities 21
RPMsg Messaging Protocol 22
Protocol Layers 22
Physical Layer – Shared Memory 23
Media Access Layer - VirtIO 23
Transport Layer - RPMsg 25
RPMsg Header Definition 25
Flags Field 26
RPMsg Channel 26
RPMsg Endpoint 26
RPMsg Protocol Limitations 27
RPMsg Communication Flow 27
Life Cycle Management 29
LCM Overview 29
Creation and Boot of Remote Firmware Using remoteproc 31
Defining the Resource Table and Creating the Remote ELF Image 31
Procedure 31
Making Remote Firmware Accessible to the Master 32

Procedure 32
System Wide Considerations 32
Porting GuideLine 34
Add System/Machine Support in Libmetal 34
Platform Specific Remoteproc Driver 34
Platform Specific Porting to Use Remoteproc to Manage Remote Processor 34
Platform Specific Porting to Use RPMsg 35
Resource Table Evolution 38
Overview 38
Needs 38
Compatibility with Linux kernel 38
Review current resource table fields 38
64 bit addresses 38
Parameter passing 38
Evolutive virtio dev features 38
Pointer to Device Tree 38
vdev buffer management 38
Trace evolution 38
Enhancement 39
Define resource table ownership 39
Include the resource table size 39
New resource to provide processors states 39
Mechanisms 39
Resource table version number 39
Per item version number 39
Per item Priority 39
vdev buffer management 40
Traces 40
Firmware publishes multiple versions of Resource table 40
Using OpenAMP CI 40
Running the Xilinx QEMU in Docker 40
Get Docker on your machine 40
run QEMU inside docker 40
look around the container 41
reattach to qemu serial console 43
kill and cleanup 43
Items for improvement 43
Priority Items 43
Nice to have items 43
Indices and tables **44**

Welcome to the OpenAMP Project Documentation

OpenAMP Project

Project Overview

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

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

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.

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

Current Work

Remoteproc Sub-Group

Introduction

Bill to revise summary about this group The OpenAMP remoteproc sub-group (A.K.A. “OpenAMP Classic”) works on the original parts of OpenAMP, focusing on open-amp, libmetal, remoteproc, rpmsg, virtio.

Communications

Mailing list

We have a Mailman list for OpenAMP Remoteproc sub-group discussions. You can find info about it, reach the link to the archives, and subscribe/unsubscribe here.

Meetings

Check out the meeting notes for the OpenAMP Remoteproc meetings on the OpenAMP Meeting Notes page. The cadence is every 2 weeks on Thursday mornings at 11am Eastern Time. Join the openamp-rp mailing list (see above) for the latest about the upcoming calls.

Documentation

Placeholder for Bill to populate

Placeholder for other documentation here

OpenAMP Introduction from Linaro Connect HKG18

Slides

Future work

Check out the future work list for the OpenAMP Remoteproc sub-group in the Remoteproc Sub-Group Future Work section.

System Device Tree Sub-Group

Introduction

Todays heterogeneous SoCs are very hard to configure. Issues like which cores, memory and devices belongs to which operating systems, hypervisors and firmware is done in an ad-hoc, error prone way. System Device Trees will change all that by extending todays device trees, used by Linux, Xen, uboot, etc. to describe the full system and also include configuration information on what belongs where.

Communications

Mailing list

We have a Mailman list for system-dt discussions. You can find info about it, reach the link to the archives, and subscribe/unsubscribe here

Meetings

Check out the meeting notes for the System Device Tree meetings on the OpenAMP Meeting Notes page.

Documentation

System DT intro presentation given at Linaro Connect SAN19

- Video
- Slides

Placeholder for Stefano to populate

Placeholder for other documentation here

Lopper

Lopper is a device tree manipulation tool that has been created to provide data driven manipulation and transformation of System Device Trees into any number of output formats, with an emphasis on conversion to standard device trees.

The source code and information can be found at: https://github.com/devicetree-org/lopper/

Future work

Check out the future work list for the System Device Tree sub-group in the System Device Tree Sub-Group Future Work section

Application Services Sub-Group

Introduction

This sub-group covers the application space that sits on top of OpenAMP and describes higher level services for

- file sharing
- proxy and/or forwarding of IP ports
- debug proxy
- high level IPC APIs for send-receive-reply / byte streams / message-based connections / pub/sub
- IPC server registration and client binding
- application partitioning using RPCs, C & non-C languages, canonical format
- bare metal APIs (using RPC) for stdio, socket IO, other APIs

Communications

Mailing list

We have a Mailman list for app-services discussions. You can find info about it, reach the link to the archives, and subscribe/unsubscribe here.

Meetings

Check out the meeting notes for the Application Services meetings on the OpenAMP Meeting Notes page.

Documentation

Placeholder

Placeholder for other documentation here

Future work

Check out the future work list for the Application Services sub-group in the Higher Level Services Sub-Group Future Work section.

Inclusive Language and Biased Terms Sub-Group

This proposal from Bill Mills was approved at the 10/22/21 OpenAMP TSC:

replace master with main

github has special case logic to make this easier: https://github.com/github/renaming

remoteproc context

- “slave” should be “remote processor”
- “master” should be “remoteproc host”

virto context

virtio spec uses “device” and “driver”. suggest we use “virtio device” and “virtio driver”

Examples of devices:

- vitioblk device
- virtio network interface

For today’s remoteproc rpmsg, Linux is always the driver side.

Some virtio “devices” are not very device like and instead are more like services. Alternative for such cases

- “application service” and “application client”

Examples:

- vsock: service is the one that calls accept on the socket
- p9fs: service is the side that has the filesystem

Note that a remote processor can host a service and be a client at the same time. The terminology is per service.

OpenAMP CI Sub-Group

- CI regression for every push request from OpenAMP repo
- Enabling OpenAMP tests for different users to run on different platforms

IPC Sub-Group

- Shared memory support to use DMA buffer in Linux userspace

System Reference / End-to-End Example Sub-Group

What is OpenAMP System Reference / End-to-end example?

This working group aims to put together end-to-end system reference material showcasing all the different aspects of OpenAMP, on multiple vendor platforms.

Communications

- To sign up for the mailing list and to see the archives, visit this page BROKEN LINK
- **Meetings: Small group who is working on the project is meeting meeting weekly on Wednesday during Aug-Oct 2021 & then will decide on frequency.**
    - Reach out on the Mailing list BROKEN LINK if you need info about this.
    - Project Meeting Notes

Repository

GitHub repository openamp-system-reference

Documentation

WIP documentation folder on Google Drive (Note: Google doc access is currently restricted to working group members. Will publish documents once they are sufficiently ready)

Samples and demos

Please refer to Samples and demos page.

Milestones

Future work

Efforts

- Baremetal-baremetal
- RTOS-RTOS

High level plan for Xilinx Software Stack

- QEMU & initial doc – Mid Sept.
- Userspace & kernel space demos – Sept end
- Hardware demo – End - Oct.
- System-dt flow (Without using Xilinx tools) – End-Nov
- Advanced app – End-Jan
- Completely upstream flow – (Based on When Xilinx driver is merged in upstream kernel)

Older Topics

Formation of OpenAMP project

- Launched as a Linaro Community Project at Linaro Connect SAN19

Samples and Demos

System Reference Samples and Demos

Echo Test

ST Echo Test Example

Instruction to install the yocto environment, build and load an image, run the echo example is available here: https://github.com/arnopo/oe-manifest/blob/OpenAMP/README.md

For more information and help on the stm32mp15 platform and environment, please refer to stm32mpu wiki.

Xilinx Echo Test Example

WIP document (Note Google doc access is currently restricted to working group members. To publish once it is sufficiently ready)

Multi RPMSG services demo

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,
- a fork of the yocto [meta-zephyr](https://git.yoctoproject.org/meta-zephyr/).

Prerequisite

TBC

Installation

Create the structure of the project

```
mkdir stm32mp15-demo
cd stm32mp15-demo
mkdir zephy_distrib
mkdir stm32mp1_distrib_oss
```

At this step you should see following folder hierarchy:

```
stm32mp15-demo
    |___ stm32mp1_distrib_oss
    |___ zephy_distrib
```

Generate the stm32mp15 image

Install stm32mp1_distrib_oss dunfell

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/dunfell
cd -

git clone https://github.com/openembedded/bitbake.git layers/openembedded-core/bitbake
cd layers/openembedded-core/bitbake
git checkout -b WORKING  origin/1.46
cd -

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

git clone https://github.com/arnopo/meta-st-stm32mp-oss.git layers/meta-st/meta-st-stm32mp-o
cd layers/meta-st/meta-st-stm32mp-oss
git checkout -b WORKING origin/dunfell
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.

Flash 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,
dd if=core-image-base-stm32mp15-disco-oss.wic of=/dev/<device> bs=8M conv=fdatasync
```

Generate the Zephyr firmware image

Install meta-zephyr Honister version

From stm32mp15-demo directory

```
cd zephy_distrib

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

git clone https://github.com/openembedded/bitbake.git layers/openembedded-core/bitbake
cd layers/openembedded-core/bitbake
git checkout -b WORKING  origin/1.46
cd -

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

git clone https://github.com/arnopo/meta-zephyr.git layers/meta-zephyr
cd layers/meta-zephyr
git checkout -b WORKING origin/OpenAMP_demo
cd -
```

Initialize the OpenEmbedded 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 the stm32mp15-demo/zephy_distrib directory

```
MACHINE="stm32mp157c-dk2" DISTRO="zephyr" source layers/openembedded-core/oe-init-build-env
bitbake-layers add-layer ../layers/meta-openembedded/meta-oe/
bitbake-layers add-layer ../layers/meta-openembedded/meta-python/
bitbake-layers add-layer ../layers/meta-zephyr/
```

Build the Zephyr image

For instance to build the zephyr-openamp-rsc-table example which answers to the Linux rpmsg sample client example

From the stm32mp15-demo/zephy_distrib/build-zephyr directory

```
MACHINE="stm32mp157c-dk2" DISTRO="zephyr" bitbake zephyr-openamp-rsc-table
```

Note that

- to build around 30 GB is needed,
- building the distribution can take 1 or 2 hours depending on performance of the PC.

Install the Zephyr binary on the sdcard

The Zephyr sample binary is available in the sub-folder of build directory stm32mp15-demo/zephy_distrib/build-zephyr/tmp-newlib/deploy/images/stm32mp157c-dk2/. It needs to be installed on the “rootfs” partition of the sdcard

```
sudo cp tmp-newlib/deploy/images/stm32mp157c-dk2/zephyr-openamp-rsc-table.elf <mount    poin
```

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
```

- Start the Cortex-M4 firmware

```
echo zephyr-openamp-rsc-table.elf > /sys/class/remoteproc/remoteproc0/firmware
echo start >/sys/class/remoteproc/remoteproc0/state
```

You should observe following traces on console

```
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 -
[   54.557337] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 1 (src:
[   54.565532] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 2 (src:
[   54.581090] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 3 (src:
[   54.588699] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 4 (src:
[   54.599424] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 5 (src:
...
```

This inform that following rpmsg channels devices have been created

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

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 -
[   54.557337] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 1 (src:
[   54.565532] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 2 (src:
...
[   55.436401] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 99 (src
[   55.445343] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: incoming msg 100 (sr
[   55.454280] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: goodbye!
[   55.461424] virtio_rpmsg_bus virtio0: destroying channel rpmsg-client-sample addr 0x400
[   55.469707] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1024: rpmsg sample client
```

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@stm32mp15-disco-oss:~# 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@stm32mp15-disco-oss:~# cat /dev/ttyRPMSG0 &
root@stm32mp15-disco-oss:~# echo " hello Zephyr" >/dev/ttyRPMSG0
TTY 0:  hello Zephyr
root@stm32mp15-disco-oss:~# 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@stm32mp15-disco-oss:~# cat /dev/ttyRPMSG0 &
root@stm32mp15-disco-oss:~# echo " hello Zephyr" >/dev/ttyRPMSG0
TTY 0:  hello Zephyr
```

- Download rpmsgexport tools relying on the /dev/rpmsg_ctrl, and compile it in an arm environment unsing 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- create a new TTY channel Create a rpmsg-tty channel with local address set to 257 and undefined remote address -1.

```
root@stm32mp15-disco-oss:~# ./rpmsgexportdev /dev/rpmsg_ctrl0 rpmsg-tty 257 -1
```

The /dev/ttyRPMSG1 is created

```
root@stm32mp15-disco-oss:~# 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@stm32mp15-disco-oss:~# dmesg
[  115.757439] rpmsg_tty virtio0.rpmsg-tty.257.-1: TX From 0x101, To 0x35, Len 40, Flags 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
[  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:
[  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
```

2- Play with /dev/ttyRPMSG0 and /dev/ttyRPMSG1

```
root@stm32mp15-disco-oss:~# cat /dev/ttyRPMSG0 &
root@stm32mp15-disco-oss:~# cat /dev/ttyRPMSG0 &
root@stm32mp15-disco-oss:~# echo hello dev0 >/dev/ttyRPMSG0
TTY 0: hello dev0
root@stm32mp15-disco-oss:~# echo hello dev1 >/dev/ttyRPMSG1
TTY 1: hello dev1
```

3- Destroy RPMSG TTY devices

- Destroy the /dev/ttyRPMSG0

```
root@stm32mp15-disco-oss:~# ./rpmsgexportdev /dev/rpmsg_ctrl0 -d rpmsg-tty 257 -1
```

- Destroy the /dev/ttyRPMSG1

Get the source address

```
root@stm32mp15-disco-oss:~# cat /sys/bus/rpmsg/devices/virtio0.rpmsg-tty.-1.*/src
0x402
```

- Destroy the /dev/ttyRPMSG1 specifying the address 1026 (0x402)

```
root@stm32mp15-disco-oss:~# ./rpmsgexportdev /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 rpmsgexport tools relying on the /dev/rpmsg_ctrl, an compile it in an arm environment unsing 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@stm32mp15-disco-oss:~# 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@stm32mp15-disco-oss:~# ./rpmsgping /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 rpmsgexport tools relying on the /dev/rpmsg_ctrl, an compile it in an arm environment unsing 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@stm32mp15-disco-oss:~# ls /dev/rpmsg*
/dev/rpmsg0       /dev/rpmsg_ctrl0
```

4- Create 3 new endpoints

```
root@stm32mp15-disco-oss:~# ./rpmsgexport /dev/rpmsg_ctrl0 my_endpoint1 100 1
root@stm32mp15-disco-oss:~# ./rpmsgexport /dev/rpmsg_ctrl0 my_endpoint2 101 1
root@stm32mp15-disco-oss:~# ./rpmsgexport /dev/rpmsg_ctrl0 my_endpoint2 103 1
root@stm32mp15-disco-oss:~# ls /dev/rpmsg?
/dev/rpmsg0  /dev/rpmsg1  /dev/rpmsg2  /dev/rpmsg3
```

5- Test them

```
root@stm32mp15-disco-oss:~# ./rpmsgping  /dev/rpmsg0
message for /dev/rpmsg0: "from ept 0x0402: ping /dev/rpmsg0"
root@stm32mp15-disco-oss:~# ./rpmsgping  /dev/rpmsg1
message for /dev/rpmsg1: "from ept 0x0001: ping /dev/rpmsg1"
root@stm32mp15-disco-oss:~# ./rpmsgping  /dev/rpmsg2
message for /dev/rpmsg2: "from ept 0x0001: ping /dev/rpmsg2"
root@stm32mp15-disco-oss:~# ./rpmsgping  /dev/rpmsg3
message for /dev/rpmsg3: "from ept 0x0001: ping /dev/rpmsg3"
```

6- Destroy them

```
root@stm32mp15-disco-oss:~# ./rpmsgdestroyept /dev/rpmsg1
root@stm32mp15-disco-oss:~# ./rpmsgdestroyept /dev/rpmsg2
root@stm32mp15-disco-oss:~# ./rpmsgdestroyept /dev/rpmsg3
root@stm32mp15-disco-oss:~# ls /dev/rpmsg?
/dev/rpmsg0
```

Maintenance

Contributing to the OpenAMP Project

- **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
- **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
- **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
- **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)
- **Push rights**

- Push rights restricted to Core Team
- Generally exercised by top-level maintainer (Wendy Liang)
- Maintainer manages delegation within core team

Meeting Notes

Meeting_Notes

TODO - Move notes to google doc drive? It may be arduous to post notes to this page

Future Work

Remoteproc Sub-Group Future Work

This list (needs update) covers topics for the “OpenAMP remoteproc” sub-group to discuss and work on. This sub-group covers areas such as remoteproc, rpmessage, virtio, big buffers, etc.

The sections below should include an short abstract and links to prior discussion etc. More detailed information should be added on a new wiki page and a link to that page should be added to the topic here.

HW description (Xilinx, TI, ST, Mentor)

- How to describe HW to System SW – Complete HW vs. OS centric subsystems – DT, IPExact. More info can be found here.
- Q: 2019/11/7, Is this the same as SystemDT and should be moved to that group, or are there rp specific topics?

System Resource Management – (ST, Xilinx, TI, Cypress)

- How to configure and assign resources and peripherals to coprocessors
- Complications include clock and power domain control, system firewall and/or IOMMU configuration
- Some of this gets easier if both Linux and the remote processor are using a central system controller for clock. power, and peripheral access control like SCMI or TISCI

OpenAMP project support for Virtio spec (Xilinx, TI, ST, Mentor)

- Packed vrings, indirect descriptors, …
- Virtio device handshake improvements
- **Use of existing virtio drivers (ex: virtblk, virtnet, virtcon) from remoteprocs**
    - remoteproc could be frontend or backend (ex:could be the block user or block device)
    - Q: 2019/10/7, should this be its own topic and leave this one to be just virtio mechanics?

Remoteproc Virtio driver

Remoteproc framework is covering coprocessor management (init, load, start, stop, crash, dump…), but also provides functions for vring support. That complicates the device management, mainly the parent/child relation and the memory region assignment. Idea is to create a virtrproc like virtmem or virtcon and rely on DT definition for driver probing.

- ST has some patches to share

Big data – (Xilinx, TI, ST)

- Rpmsg buffer management
- Zero copy and big buffer
- Heterogeneous Memory Management. More info can be found here BROKEN LINK
- libmetal shared memory allocation: https://github.com/OpenAMP/libmetal/issues/70
- sub topics include allocation, mapping, IOMMU, remoteproc MMU, cache maintenance, address translation
- DMA-BUF heaps seems to be the obvious choice for allocation API now

Kernel remoteproc CI patches - Loic

The rate of remoteproc/rpmsg patch review/acceptance seems slow, what can we do to accelerate it? This page has a lot of stuff (and there is a lot more waiting as 2nd tier of features). (This is not about maintainer bashing; it is about working better together.)

- **more mailing list discussion**
    - patch review should be done on kernel rpmsg list not openamp-rp list
    - openamp-rp list can be used for broader arch discussion and Linux/RTOS interaction
- more reviewed by/acked by/tested by cooperation
- remoteproc CI loop should help

OpenAMP kernel staging tree - Bill

- Create an OpenAMP github repo for linux to collect all carried and WIP patches in one place
- This is not to create a product, it is to more easily see what problem everyone is working around
- Suggestion is to have a branch per vendor where they collect all the remoteproc related patches as a rebase branch
- **Extra: Some have suggested trying to merge all the vendor work together ahead of getting to mainline**
    - However this step seems controversial, lets not focus on that, at least for now
- **Alternative: just collect a list of links to the vendor trees**
    - This is definitely easier but it can be a lot of work to find the relevant patches

Remoteproc/rpmsg CI & regression test – (Xilinx, TI, ST)

Create a CI environment that can build both RTOS and Linux side and test existing and new operations.

- QEMU based AMP SOC highly desired as target
- Should be usable in cloud CI (travis etc) and on individual developers desktop
- V0.1: Start with QEMU fork and any publicly reproducible build
- V1: Add build flow for easy developer mode
- Expand to board farm as second phase
- Nice to have: use upstream QEMU

There is a Docker container (work in process), that runs an instance of Xilinx QEMU that supports up to 4 A-53 cores and 2 R5 cores. This boots Linux, loads the firmware into the R5 via remoteproc, and can run the OpenAMP rpmsg echo test. https://hub.docker.com/repository/docker/edmooring/qemu

Secure coprocessor support – Loic

Add a generic rproc driver to support secure and isolated coprocessor thanks to some trusted services based on OPTEE or other Trusted OS. Standard flow will be to load coprocessor firmware and its associated signature somewhere in secure/non-secure shared memory and then to request secure world to load, authenticate coprocessor image and then start coprocessor Some tasks:

- Define a standard file format for firmware to authenticate (is ELF still relevant or should we rely on PKCS11 header like to integrate signature ?)
- Define TA API to control secure coprocessor (load, start, stop…)
- How to manage resource table in that case? Should we rely on some secure services or should we consider it as input for communication link (aka rpmsg) configuration between coprocessor and Linux kernel (and in that case could stay non-secure)
- Q: Can we make this generic enough to be used when coproessor is to be owned/trusted by a hypervisor instead of the secure world

Linux RPMSG only mode - Bill (TI,)

RPMSG should be usable in the Linux kernel when Linux will never be the life-cycle manager for that coprocessor.

In many modern systems the Linux kernel is not the most trusted entity in the system. Sometimes a given coprocessor is loaded and managed by another entity that is more secure (ex: secure world), more safe (ex: dual lock-set R5), or more trusted (hypervisor). In these system Linux may never be the one that loads/starts/restarts/crash dumps the coprocessor but Linux still needs to use a rpmsg channel and perhaps other virtio based communication channels.

- Need to tell remoteproc that it is not the controller
- Need to find the resource table in use
- remoteproc still needs to do: IOMMU mapping, PA to DA, etc
- Q: Can we make a generic remoteproc that is usable for this case that can handle some (but not all) SOCs?
- This model brings up many cases for robustness

RPMSG robustness - Bill (TI,ST)

- If coprocessor goes down and comes backup, Linux needs to recognize that and re-establish rpmsg communication
- If Linux goes down (crash/shutdown) and restarts while the coprocessor stays running, coprocessor needs to recognize that and re-establisg rpmsg communication
- **The restart should be advertised to the applications not hidden from them. Applications should take recovery actions themselves.**
    - On Linux this would probably mean an error code from the existing handle and a need to open a new one (or a issue a reset ioctl)

Early coprocessor boot – late attach, detach - Loic (ST,TI)

Late attach is different from rpmsg only mode in that once Linux comes up, it becomes the life cycle controller for the coprocessor.

- Linux should have the option to stay with the firmware already loaded on the coprocessor.
- Linux could later stop or reload the coprocessor with different firmware
- Linux may take ownership of crashdump and debug logs

- Late attach could require that matching firmware file exists in the filesystem, this would make finding the resource table easier
- Or late attach could require to know where firmware resource table has previously been loaded as would be required in RPMSG only mode

Detach means that Linux stops becoming the life cycle owner. This could happen while Linux is running or as part of Linux crash/shutdown.

Early booted processor patch sent on ML: https://lore.kernel.org/patchwork/patch/1147726/

Lifecycle management and Trusted & Embedded Base Boot Requirement (T/E-BBR) – Etsam

Lifecycle management with Linux remote – Etsam

The life cycle management of Linux is required in scenarios where it provides the rich execution environment and certified software environment (e.g on low end CPUs such as cortex M or R) is the system master and responsible to start/stop/recover Linux. The intent here is to cover the driver architecture (e.g. remoteproc replacement) and device tree bindings for remote Linux. No plan to cover the Linux bootstrapping and RPMSG remote mode operation. They can be treated as separate topics. Linux will still assumed to be the RPMsg master.

Rpmsg protocol documentation for remote – Etsam

The RPMSG framework master side protocol is well manifested in Linux upstream and new masters (e.g. OpenAMP) can be written using it as a reference. However, there is no standard Doc/Implementation for remote which often leads to problematic protocol scenarios. For instance, consider the two cases below:

- **At what point VDEV Resource is initialized by the Master, specially when the vrings are dynamically allocated?**
    - Conversely, at what point remote should access that vdev resource? Consider the case where vdev resource is accessed by the remote right after boot up, assumption here is that it is initialized by the master before starting the remote. This worked for kernel v3.18 , however, it is broken for v4.9 (may be for others as well) where the vring addresses are populated after remote is booted. For latter, this leads to race condition and sometimes remote ends up accessing uninitialized vdev resource. Apparently, the correct point to access vdev resource is after vdev status field (DRIVER_OK) is updated by the master.
- **When to send the Name service announcement?**
    - In response to first kick from the master: This works if remote is up and running. What if kick is sent and remote is not yet operational? It will miss the kick and consequently NS will not be sent.
    - In response to VDEV status update (DRIVER_OK). Will work if remote comes up after the first kick. The status will still be in the shared memory and can be used to send the NS.

The foolproof approach would be to use both kick and VDEV status to send the NS. This point was found after trial and error.

It is required to document all such scenarios i.e. all master actions and expected response from the remote, to enable seamless operation with different remotes.

MCU – MCU issues, rpsmg only - Mark

- **libopenamp example of rpmsg only**
    - Does it exist, are there any remaining issues?
    - target for CI loop
    - how big is it? Measure as regression test
- **MCU boot first - late-attach**
    - Crash recovery

- **Are there MISRA issues with libopenamp?**
    - Is malloc required? Can this be easily mapped to something more static
- **“Big” Data in context of MCU to MCU (same SOC)**
    - zero-copy?
- **OpenAMP improvement**
    - Libmetal rework to be continued
    - Integrates last feedbacks coming from Nordic benchmark
    - Reduce memory footprint
    - Stack usage
- rpmsg-lite: anything missing from libopenamp?

64 bits support - Clément

- **64 bits support in elf file, see https://patchwork.kernel.org/patch/11175161/**
    - Elf 64 files are needed for 64 bits processors
- **64 bits addresses in resources table**
    - Currently, addresses are only on 32bits, this is really limitating for 64bits
- **64 bits features in vdev declaration**
    - Virtio Features are now using 64 bits. Without this support, we are tied to legacy 32bits features.
    - Need to switch to at least 64 bits and provisioned more bits for future evolution

Misc I/O over VirtIO/RPMSG - Loic, Bill

- Put the “IO” in VirtIO
- **Virtual UART**
    - ST has patches
    - should this be full UART control (baud rate, HW flow control, RI/DCE/DTE/CTS/RTS) or just a communication channel
    - Is Linux the device or the user?
    - example: MCU wants to send console messages to Linux for logging
    - example: MCU owns a physical UART but wants Linux to use it.
- **Virtual I2C**
    - ST has some patches
    - exmaple: MCU owns phy I2C with multiple devices, presents virtual I2C for to Linux so it can use some but not all of the devices
- **Virtual SPI**
    - ST has some patches
- Virtual GPIO
- **Virtual register bank**
    - via: regmap
    - Can be used on its own or as the base level of some of the above (GPIO seems obvious)

- Should these be over RPMSG, direct over virtio, or both?

Reference: ST Presentation at ELC-E 2019, https://elinux.org/images/6/63/ELC_EU19_rpmsg.pdf

Improve Coprocessor debug capabilities - Loic

Today rproc framework offers access to a virtual trace file (circular buffer filed by coprocessor) which limit coprocessor debug capabilities. Tracks to explore:

- How to store coprocessor traces in a log file (syslog like) to improve trace depth?
- How to get same timestamp between Linux and coprocessors to correlate trace
- How to control coprocessor debug infrastructure (coresight?)
- Is it possible to debug coprocessor firmware thanks to GDB/GDB server over rpmsg or mailbox?
- How to avoid clashing with external JTAG debugger (RPMSG only mode may help here)

Past presentations and TODO lists

- **TI presentation on TODO list from 2017**
  - https://github.com/OpenAMP/openamp.github.io/blob/master/docs/linaro-2017/OpenAMP-TI-Roadmap.pdf
- **ST presentation from 2018 on short term TODO list**
  - https://github.com/OpenAMP/openamp.github.io/blob/master/docs/linaro-2018hkg/OpenAMP-short-term-to
- **Xilinx presentation from 2017 on Intro**
  - https://github.com/OpenAMP/openamp.github.io/blob/master/docs/linaro-2017/OpenAMP-Intro-Feb-2017.p
- **TI presentation from 2018 on Big Data and robustness, IPC only and life cycle**
  - https://github.com/OpenAMP/openamp.github.io/blob/master/docs/linaro-2018hkg/OpenAMP-Buffer-Excha
- **TI presentation from 2017 on coprocessor memory types and howto handle**
  - https://github.com/OpenAMP/openamp.github.io/blob/master/docs/linaro-2017/OpenAMP-memory-types.p

Higher Level Services Sub-Group Future Work

2019-10-17 - This list (to be populated) covers topics for the OpenAMP Higher Level Services (sub-group to decide what name they want to use) sub-group to discuss what to work on at their sub-group meetings. This sub-group covers areas such as Proxy, eRPC, WindRiver islet.

System Device Tree Sub-Group Future Work

2019-10-17 - This list (to be populated) covers topics for the OpenAMP System Device Tree sub-group to discuss what to work on at their sub-group meetings. This sub-group welcomes participants from other areas of Device Tree beyond OpenAMP as well, such as DeviceTree.org, Device Tree Evolution projects. OpenAMP Project is hosting this discussion because of its cross-functional nature.

Standardizing Hypercalls Sub-Group

2019-10-17: This list (to be populated) covers topics for the OpenAMP Standardizing Hypercalls (sub-group to decide what name they want to use) sub-group to discuss what to work on at their sub-group meetings.

HMM (Heterogeneous Memory Management)

Use Cases

- pcie/ccix endpoint side memory pools

- embedded soc e.g. dedicated media buffer alloc pool

zone device and memory hotplug (on arm64)

- arch pte devmap bit options (mair bits, pte_none?, pte_special?)
- device pluggable private and public memory

hmm address space mirroring w/ rproc

- tie into vfio-map shmem model w/ remoteproc instances?
- could we somehow do better than dma migration w/ pgtable remapping?
- (opt.) tie into vfio-bind shmem model needs intg. iommu fault handling

Hardware Description

Problems

- **Need to communicate HW details (addresses, topologies, …) to SW subsystems during build**
  - Both at system level (complete HW including all CPUs) and subsystem level
  - Tool to create subsystems with assigned devices out of system level info
- **Need to communicate HW details during runtime**
  - Device trees used by Linux, Xen, etc.
  - How to communicate to coprocessors (remoteproc) what devices it has?
- **Need tools to create static config data (#defines, .h, .c files) from HW description**
  - Zephyr working on this. Need general solution.
- **Need compact format for runtime use cases**
  - DTB is not very compact. Using strings instead of labels, no compression

Potential solutions

- **Come up with standard, humanly readable, HW description format for usage during build**
  - Possible candidates include extended Device Trees, IPExact, …
  - A “System Level Device Tree” would add another level with multiple CPUs and mappings
- **Come up with standard compressed HW description**
  - Potential candidate is CBOR

Others interested in this problem?

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.

OpenAMP Protocol Details

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 typicallyt 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.

Components and Capabilities

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

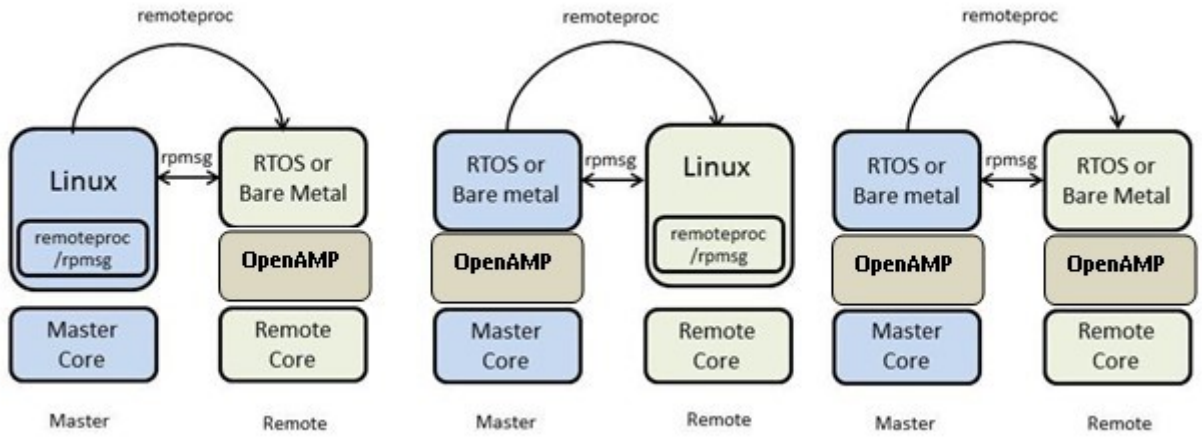
- remoteproc — This component allows for the Life Cycle Management (LCM) of remote processors from software running on a master processor. The remoteproc API provided by the OpenAMP Framework is compliant with the remoteproc infrastructure present in upstream Linux 3.4.x kernel onward. The Linux remoteproc 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.

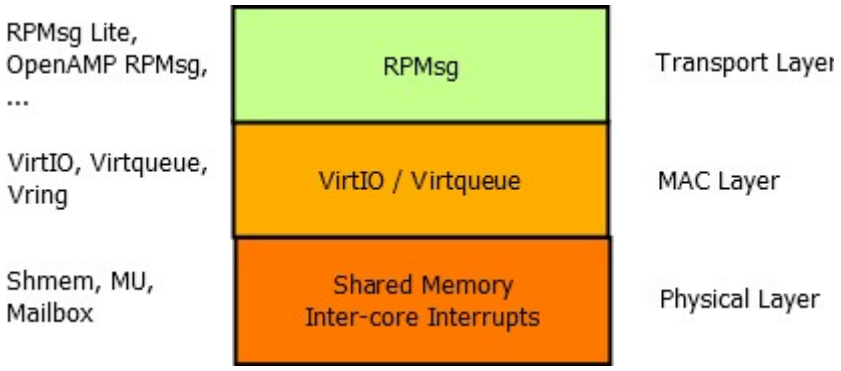
In addition to providing a software framework/API for LCM and IPC, the OpenAMP Framework supplies a proxy infrastructure that provides a transparent interface to remote contexts from Linux user space applications running on the master processor. The proxy application hides all the logistics involved 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.

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).

Protocol Layers

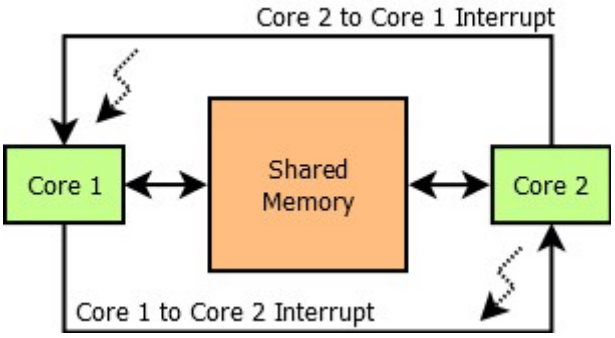
The whole communication implementation can be separated in three different ISO/OSI layers - Transport, Media Access Control and Physical layer. Each of them can be implemented separately and for example multiple implementations of the Transport Layer can share the same implementation of the MAC Layer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



Physical Layer – Shared Memory

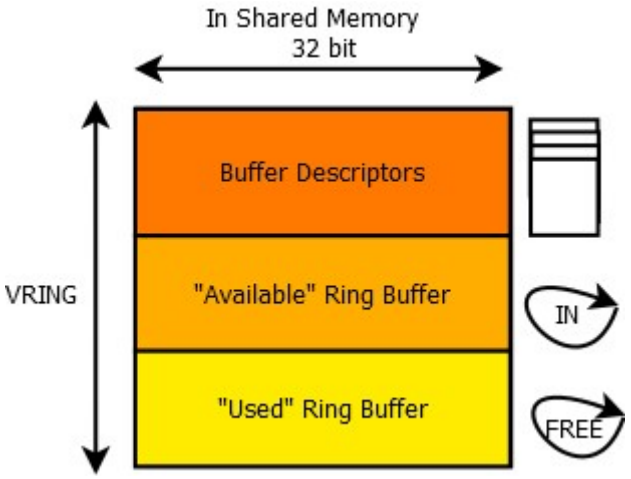
The solution proposed in this document requires only two basic hardware components - shared memory (accessible by both communicating sides) and inter-core interrupts (in a specific configuration optional). The minimum configuration requires one interrupt line per communicating core meaning two interrupts 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

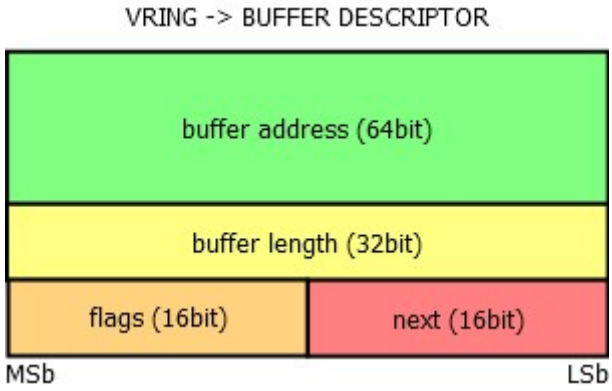
This layer is the key part of the whole solution - thanks to this layer, there is no need for inter-core synchronization. This is achieved by a technique called single-writer single-reader circular buffering, which is a data structure enabling multiple asynchronous contexts to interchange data.



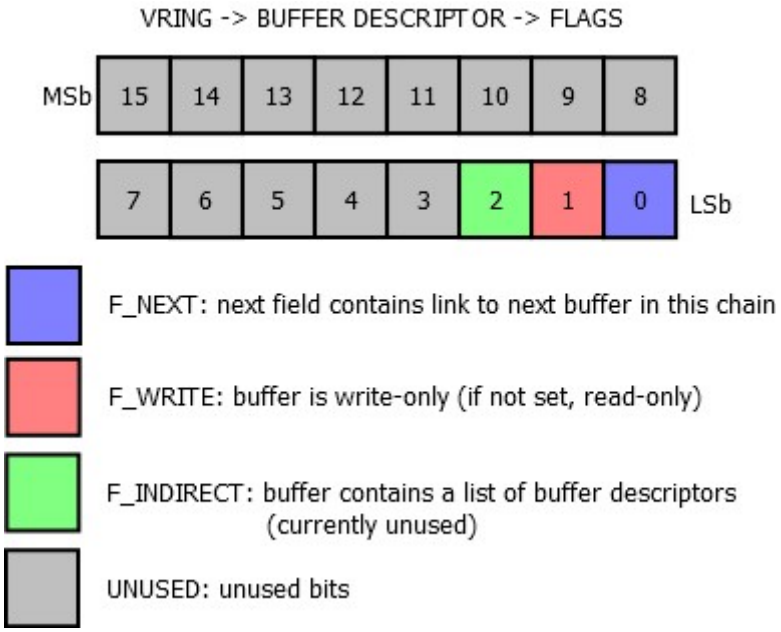
This technique is however applicable only in core-to-core configuration, not in core-to-multicore configuration, since in such a case, there would be multiple writers to the “IN” ring buffer. This would require a synchronization element, [such as a semaphore?], which is not desirable.

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.

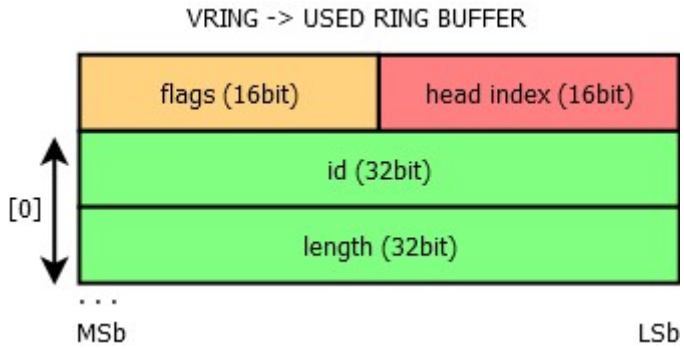
Each buffer descriptor contains a 64-bit buffer address, which holds an address to a buffer stored in the shared memory (as seen physically by the “receiver” or host of this vring), its length as a 32-bit variable, 16-bit flags field and 16-bit link to the next buffer descriptor. The link is used to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



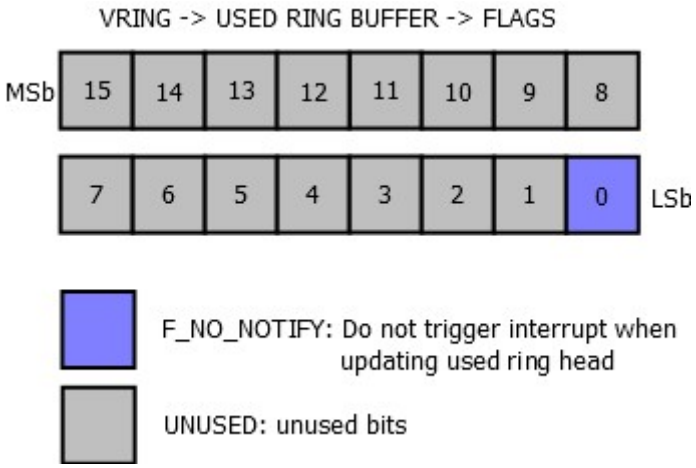
The input ring buffer contains its own flags field, where only the 0th bit is used - if it is set, the “writer” side should not be notified, when the “reader” side consumes a buffer from the input or “avail” ring buffer. By default the bit is not set, so after the reader consumes a buffer, the writer 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.



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.



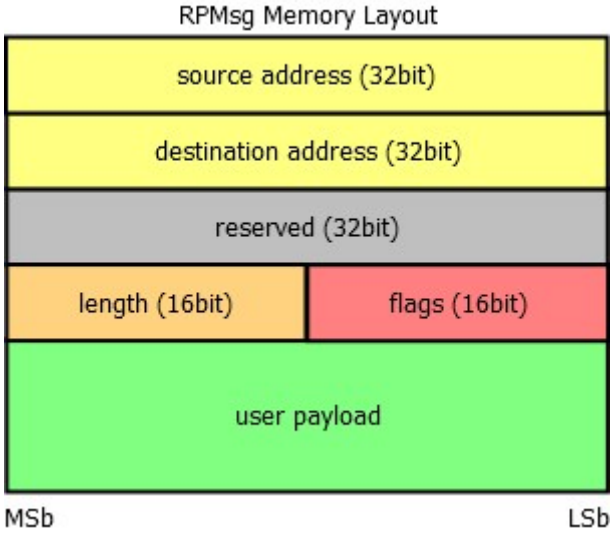
Both “used” and “avail” ring buffers have a flags field. Its purpose is mainly to tell the writer whether he should interrupt the other core when updating the head of the ring. The same bit is used for this purpose in both “used” and “avail” ring buffers:



Transport Layer - RPMsg

RPMsg Header Definition

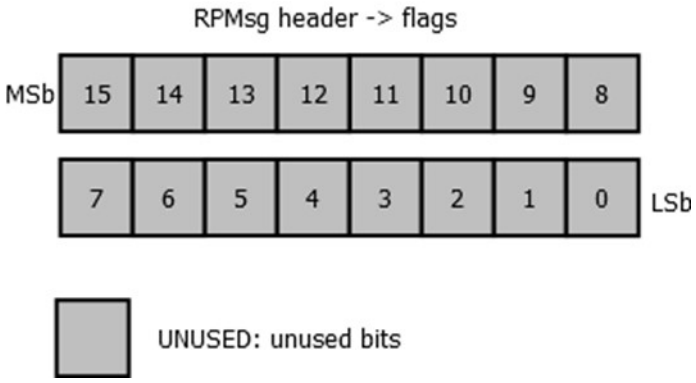
Each RPMsg message is contained in a buffer, which is present in the shared memory. This buffer is pointed to by the address field of a buffer descriptor from vring’s buffer descriptor pool. The first 16 bytes of this buffer are used internally by the transport layer (RPMsg layer). The first word (32bits) 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.



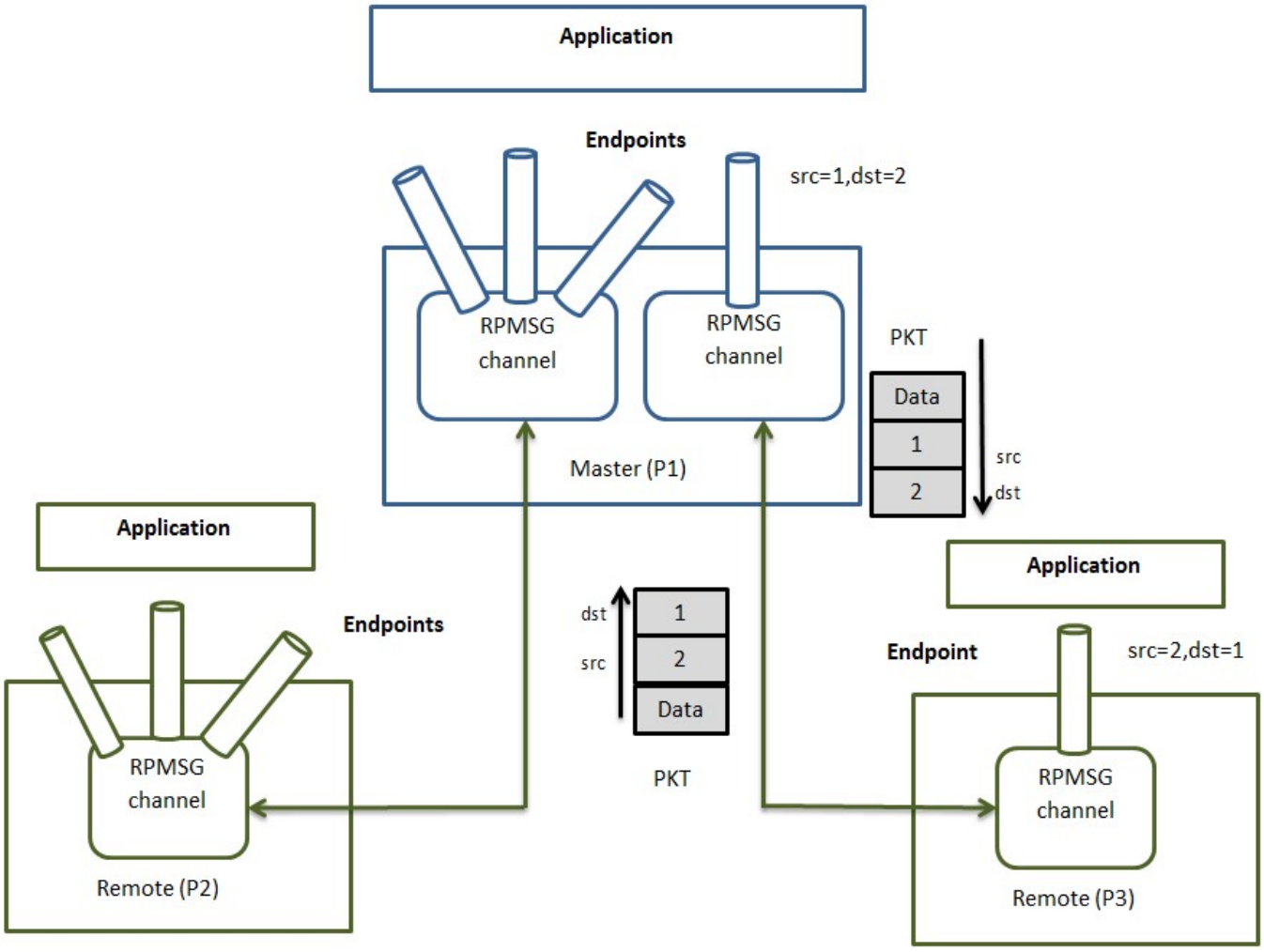
RPMsg Channel

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.

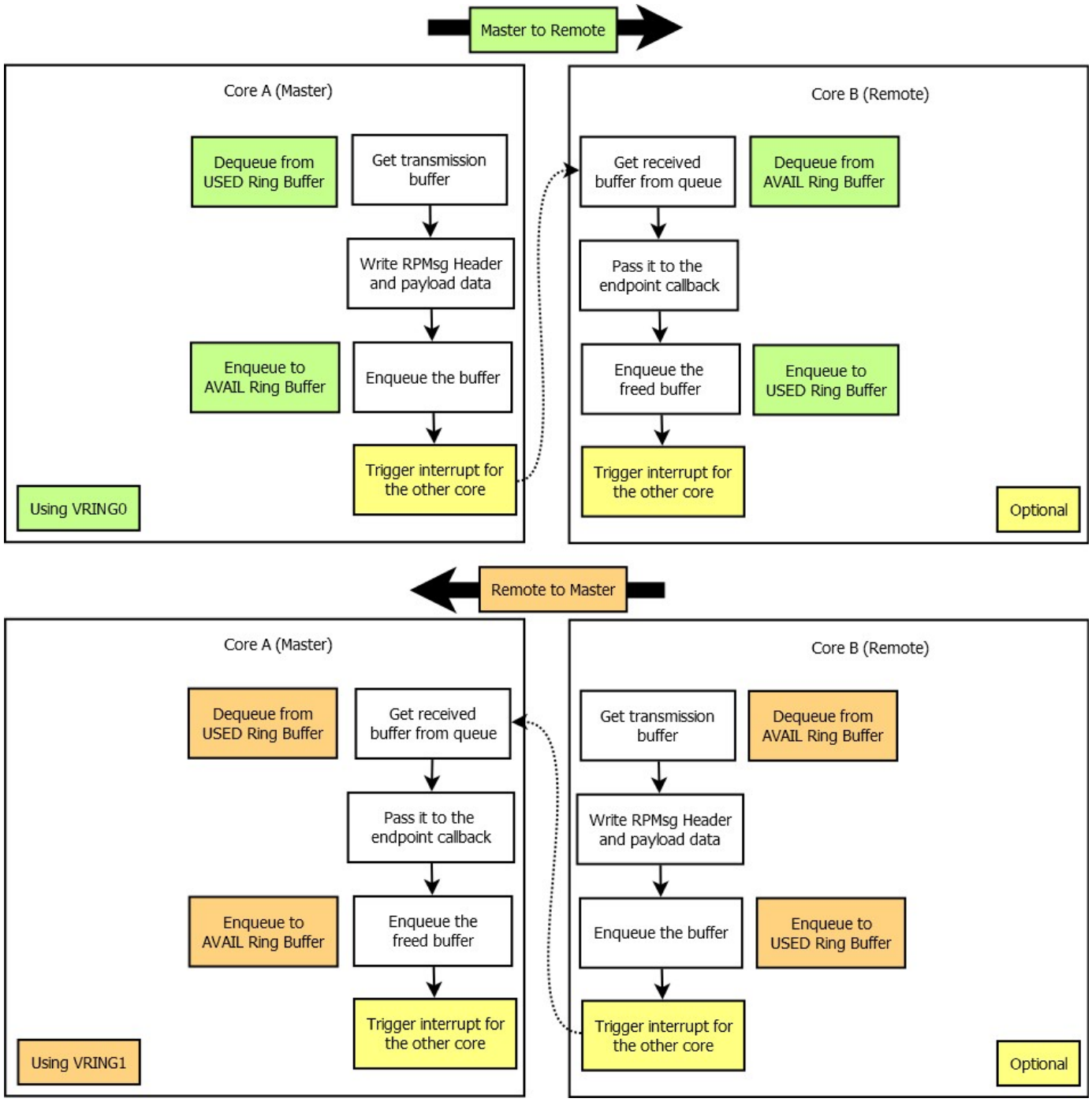


RPMsg Protocol Limitations

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.

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.



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.

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.

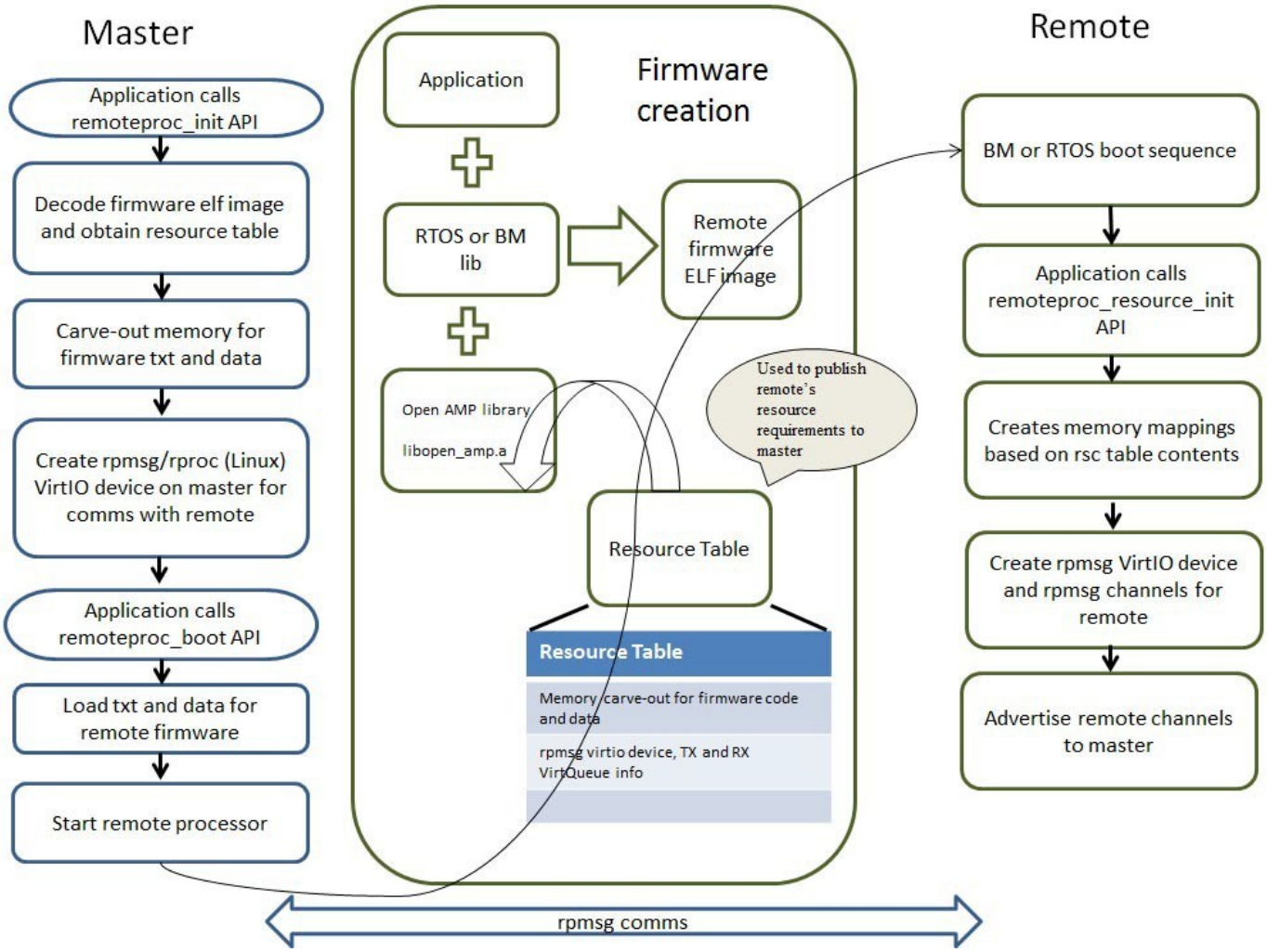
LCM Overview

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.



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.

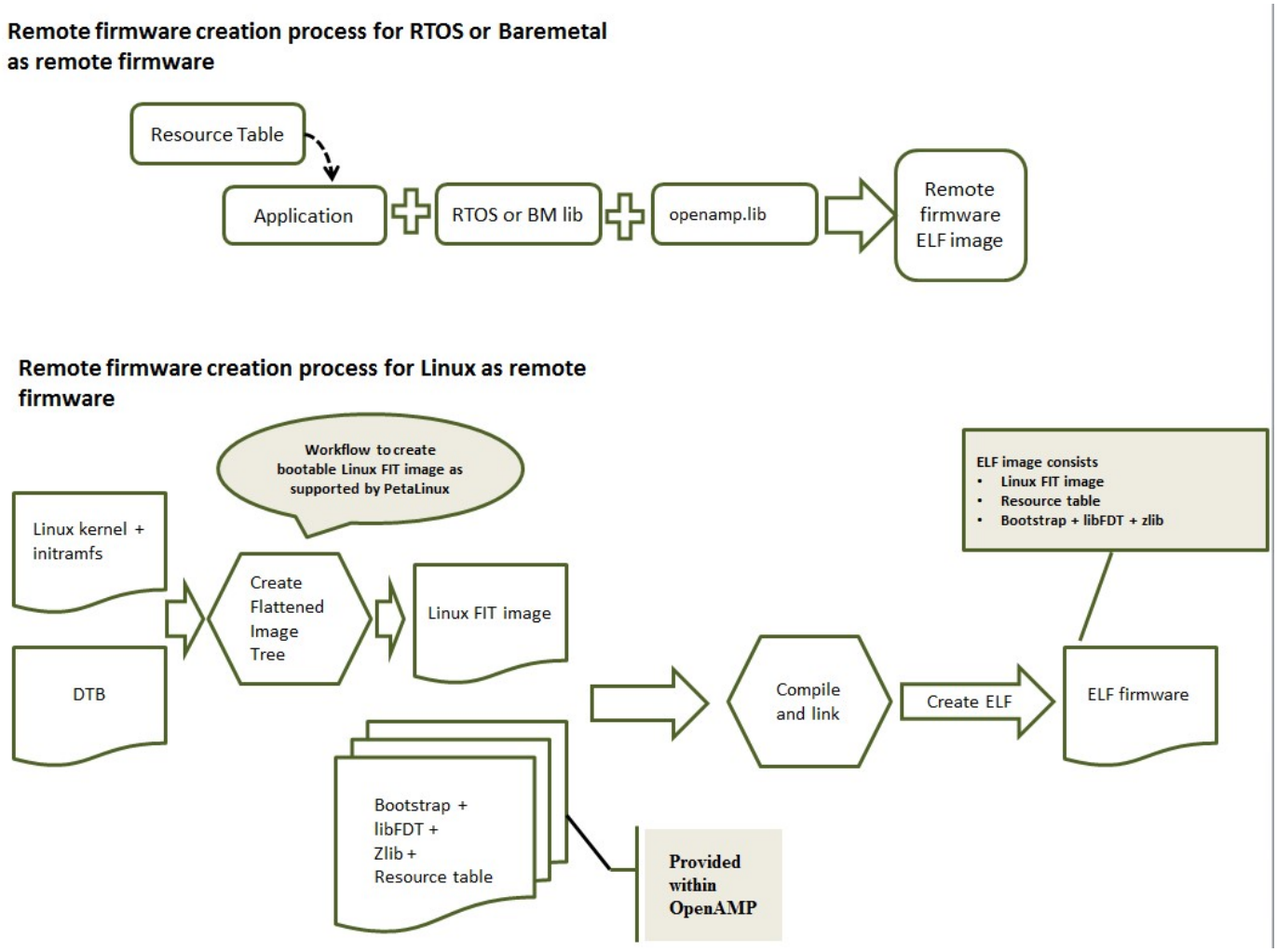
For gracefully bringing down the remote context, the following steps can be performed:

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.

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.



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.

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.

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.

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.

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.

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);

      /* 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);
}
```

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. */
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);

	/* 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);

	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);
}
```

Resource Table Evolution

Overview

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

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.

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.

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

Using OpenAMP CI

The OpenAMP CI solution is just in it’s infancy at the moment and this doc is a work-in-progress. Please keep it up to date and don’t keep historical information (that is what wiki history is for).

Running the Xilinx QEMU in Docker

Get Docker on your machine

You will need docker on your machine. Google for howto setup on your machine.

Example for Ubuntu 18.04:

```
$ sudo apt update; sudo apt install docker.io docker-doc; sudo adduser myuser docker
```

Then logout and log back in.

Your life will be easier if you are not behind a corporate firewall. If you are checkout: https://www.serverlab.ca/tutorials/containers/docker/how-to-set-the-proxy-for-docker-on-ubuntu/

run QEMU inside docker

```
$ docker run -it --name oaci edmooring/qemu:xilinx-qemu
```

This will:

- pull the image from docker hub
- start the container
- attach your interactive terminal to the container
- run a script that will launch the QEMU session
- your terminal will become the simulated SOC’s serial console
- you will see the kernel boot messages
- you will see the remoteproc driver for the zyncmp R5 core probe
- you will get a login prompt
- hit enter once to get a new login prompt (why?)
- login with user of root and password also of root
- you will have a minimal but pretty standard image based on PetaLinux 2019.02
- you can install more software with: (if not behind a firewall)

```
# dnf install tar less mc
```

Explore the system

```
$ uname -a
$ zcat /proc/config.gz
```

To run the rpmsg echo test do the following:

```
$ echo image_echo_test >/sys/class/remoteproc/remoteproc0/firmware
$ echo start >/sys/class/remoteproc/remoteproc0/state
$ echo_test
```

Detach from the container with the key sequence ^P ^Q

look around the container

```
$ docker ps
$ docker exec -it oaci /bin/bash
```

You will now be root in the container. The container is a very minimal Debian 9 system but it does have dpkg and apt. If you are not behind a firewall you can:

```
# apt update; apt install less tar procps mc
```

In any case you can checkout the important parts of the container

```
# find /opt/arm
/opt/arm
/opt/arm/libexec
/opt/arm/libexec/qemu-bridge-helper
/opt/arm/bin
/opt/arm/bin/qemu-system-aarch64
/opt/arm/bin/qemu-pr-helper
/opt/arm/bin/qemu-aarch64
/opt/arm/bin/qemu-img
/opt/arm/bin/ivshmem-client
/opt/arm/bin/qemu-system-arm
/opt/arm/bin/qemu-ga
/opt/arm/bin/ivshmem-server
/opt/arm/bin/qemu-microblazeel
/opt/arm/bin/qemu-io
/opt/arm/bin/qemu-system-microblazeel
/opt/arm/bin/qemu-arm
/opt/arm/bin/qemu-nbd
/opt/arm/share
/opt/arm/share/qemu
/opt/arm/share/qemu/pxe-e1000.rom
/opt/arm/share/qemu/spapr-rtas.bin
/opt/arm/share/qemu/qemu_logo_no_text.svg
/opt/arm/share/qemu/efi-vmxnet3.rom
/opt/arm/share/qemu/pxe-ne2k_pci.rom
/opt/arm/share/qemu/pxe-eepro100.rom
/opt/arm/share/qemu/vgabios-qxl.bin
/opt/arm/share/qemu/s390-netboot.img
/opt/arm/share/qemu/efi-e1000.rom
/opt/arm/share/qemu/vgabios-vmware.bin
/opt/arm/share/qemu/slof.bin
/opt/arm/share/qemu/u-boot.e500
/opt/arm/share/qemu/s390-ccw.img
/opt/arm/share/qemu/efi-pcnet.rom
/opt/arm/share/qemu/QEMU,tcx.bin
/opt/arm/share/qemu/vgabios.bin
```

```
/opt/arm/share/qemu/pxe-pcnet.rom
/opt/arm/share/qemu/openbios-sparc32
/opt/arm/share/qemu/bios.bin
/opt/arm/share/qemu/keymaps
/opt/arm/share/qemu/keymaps/pt-br
/opt/arm/share/qemu/keymaps/nl-be
/opt/arm/share/qemu/keymaps/es
...
# find /usr/local -type f
/usr/local/bin/start-qemu
# cat /usr/local/bin/start-qemu
mkdir -p /tmp/qemu-tmp
/opt/arm/bin/qemu-system-microblazeel -M microblaze-fdt -display none \
   -serial mon:stdio -serial /dev/null \
   -hw-dtb /var/lib/qemu/images/zynqmp-qemu-multiarch-pmu.dtb \
   -kernel /var/lib/qemu/images/pmu_rom_qemu_sha3.elf \
   -device loader,file=/var/lib/qemu/images/pmufw.elf \
   -machine-path /tmp/qemu-tmp \
   -device loader,addr=0xfd1a0074,data=0x1011003,data-len=4 -device loader,addr=0xfd1a007C,d
/opt/arm/bin/qemu-system-aarch64 -M arm-generic-fdt -serial mon:stdio -serial /dev/null -dis
  -device loader,file=/var/lib/qemu/images/bl31.elf,cpu-num=0 \
  -device loader,file=/var/lib/qemu/images/Image,addr=0x00080000 \
  -device loader,file=/var/lib/qemu/images/openamp.dtb,addr=0x1407f000 \
  -device loader,file=/var/lib/qemu/images/linux-boot.elf \
  -gdb tcp::9002 \
  -dtb /var/lib/qemu/images/openamp.dtb \
  -net nic \
  -net nic \
  -net nic \
  -net nic,vlan=1 \
  -net user,vlan=1,tftp=/tftpboot \
  -hw-dtb /var/lib/qemu/images/zynqmp-qemu-multiarch-arm.dtb \
  -machine-path /tmp/qemu-tmp \
  -global xlnx,zynqmp-boot.cpu-num=0 \
  -global xlnx,zynqmp-boot.use-pmufw=true \
  -m 4G
# find /var/lib/qemu
/var/lib/qemu
/var/lib/qemu/images
/var/lib/qemu/images/system.dtb
/var/lib/qemu/images/bl31.elf
/var/lib/qemu/images/zynqmp-qemu-multiarch-pmu.dtb
/var/lib/qemu/images/system.bit
/var/lib/qemu/images/u-boot.elf
/var/lib/qemu/images/bl31.bin
/var/lib/qemu/images/pxelinux.cfg
/var/lib/qemu/images/pxelinux.cfg/default
/var/lib/qemu/images/linux-boot.elf
/var/lib/qemu/images/boot.scr
/var/lib/qemu/images/Image
/var/lib/qemu/images/u-boot.bin
/var/lib/qemu/images/zynqmp-qemu-multiarch-arm.dtb
/var/lib/qemu/images/System.map.linux
/var/lib/qemu/images/zynqmp-qemu-arm.dtb
/var/lib/qemu/images/zynqmp_fsbl.elf
/var/lib/qemu/images/openamp.dtb
/var/lib/qemu/images/pmu_rom_qemu_sha3.elf
/var/lib/qemu/images/pmufw.elf
```

reattach to qemu serial console

```
# echo I am still root in the container
# exit
$ echo I am now back to the host
$ docker attach oaci
# echo now I am in QEMU system
^P ^Q
$ echo back at host prompt
```

kill and cleanup

In order to reuse the same name, we need to stop the container and remove it. You can use ‘stop’ instead of ‘kill’ for a graceful stop

```
$ docker kill oaci || true
$ docker rm oaci || true
```

Items for improvement

Priority Items

- **The SOC rootfs is an initramfs built into the kernel image, this is not convenient**
    - load initramfs.cpio.gz as a separate file if the qemu loader can handle it
    - Else, make an sdcard image or an mtd image that qemu can handle
- **Need to be able to rebuild kernel image and inject it into docker**
    - Should be pretty easy, config and tag and gcc version are visible in running image
    - use -v option in docker command line to get files from host into container
    - in qemu start up, look for Image from host and use it if present
- **Need to be able to rebuild R5 images**
    - need info on how to rebuild
    - inject & use as with kernel image above
- **Need test scripts**
    - Look for test scripts from host, use some defaults if not
    - SOC rootfs should run test scripts at startup
- Should publish the dockerfile that builds this docker image

Nice to have items

- **The SOC rootfs runs dropbear but use of user mode networking at QEMU level leaves it inaccessible**
    - This makes it hard to get files into or out of the system
    - Use bridge networking in QEMU
    - Inside Docker, forward SOC port 22 to host port 2222 and expose
- **A corporate firewall makes it harder to use**
    - Can we push or inherit proxy info from host -> container and container -> SOC rootfs

- **Can we use 2nd serial port direct from R5?**
    - screen or tmux could show both serial ports at the same time
    - could be an option if we don't want it by default
- **Can we run both R5s?**
    - would be good to run MCU to MCU use cases
- **Document how to rebuild other items**
    - PMU firmware, AT-F bl1, linux-loader, sw dtbs
    - **QEMU**
        - install path should be /opt/xilinx right??

Indices and tables

- `genindex`
- `modindex`
- `search`