See all versions of this document

Libmetal and OpenAMP User Guide

UG1186 (v2025.2) November 26, 2025

Table of Contents

Overview

Introduction

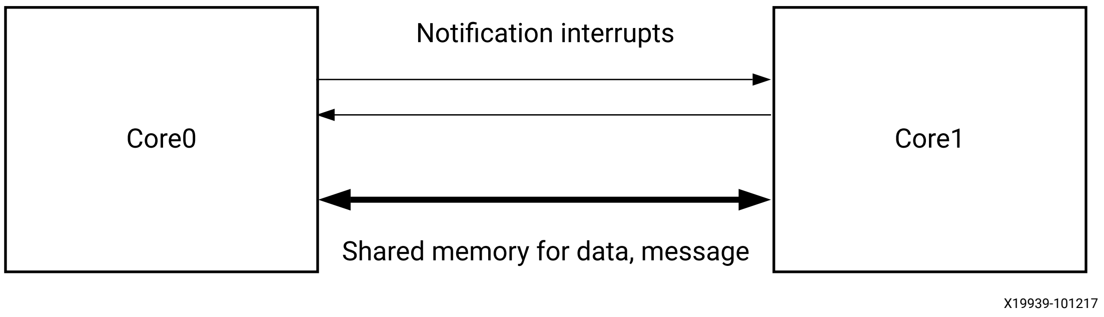
This user guide describes how to develop a methodology to enable communication between multiple processors on AMD Zynq™ 7000 devices, AMD Zynq™ UltraScale+™ MPSoCs, and AMD Versal™ adaptive SoCs.

The libmetal shared memory user API based on the ION interface will be obsoleted and replaced in future releases. This decision is due to the fact that the Linux kernel is 'Destaging ION'. For more information, see Destaging ION Article.

The libmetal AMD Vitis™ tools flow uses its own copy of libmetal from its own code repository. To reduce code management overhead this local repository is replaced with an AMD external repository.

The basic development concept uses the foundational principles of Interrupts and Shared Memory as the method of communicating between elements:

Figure 1: **Inter Processor Communication**



The libmetal library provides common user APIs (Application Programming Interfaces), used to access devices, handle device interrupts, and request memory across different operating environments. You can use libmetal to build your own AMP (Asymmetric Multi-/Processing) solution. AMD uses the OpenAMP (Open Asymmetric Multi-processing) project as the default AMP solution. OpenAMP builds on top of libmetal to provide a framework for remote processor management and inter-processor communication. This document describes the relationship between Libmetal and OpenAMP in the subsequent sections.

Software Tools Requirements

PetaLinux and Vitis tools are required to follow the instructions in this document to build applications. See AMD Documentation for more detailed information.

Prerequisites

To use the OpenAMP Framework effectively, you must have a basic understanding of:

- Linux, PetaLinux, and Vitis tools
- How to boot an AMD board using JTAG boot
- The remoteproc, RPMsg, and virtIO components used in Linux and bare-metal

Introduction to the New Upstream remoteproc Driver

- New zynqmp remoteproc bindings in `kernel v6.12: <kernel src root>/Documentation/devicetree/bindings/remoteproc/xlnx,zynqmpr5fss.yaml`
- Bindings document also provides an example of a device-tree for the zynqmp platform.

  New zynqmp remoteproc driver in kernel v6.12: `/drivers/remoteproc/xlnx_r5_remoteproc.c`
- Enable the following Kconfig option from `drivers/remoteproc/Kconfig`: CONFIG_XLNX_R5_REMOTEPROC

New xlnx remoteproc bindings and a new xlnx remoteproc driver has the following advantages:

- Data-driven design allows TCM representation in the device-tree.
- No hard coding needed in the driver (only TCM hard coding maintained for backward compatibility).
- New platform support can be added easily with new address space.
- Allows the usage of only the remoteproc if RPMsg is used from the user space.
- Available in upstream Linux kernel from v6.12.

Attach-detach Feature in the remoteproc Subsystem

Introduction

- There is a use case where RPU is already running firmware before the Linux kernel boot. In this case, Linux cannot load firmware in the RPU. Instead, Linux connects to the running firmware through a special operation in the remoteproc framework called *attach*. Implementation of attach and detach is platform-specific.

- To perform the attach operation, the remoteproc driver needs to retrieve the resource table runtime from the firmware. This RPU firmware requires a special metadata that holds the resource table address and size.
- This resource table metadata structure is pre-defined and the RPU firmware is expected to provide a specific location. This specific location is the start address of the first element of the memory-regions property list in the Linux remoteproc device-tree node.

Resource Table Metadata Structure

The firmware is expected to provide the resource table metadata through the following data structure:

```
struct remote_resource_table_metadata {
    const int version;  /* version of this data structure */
    const u32 magic_num;  /* 32-bit magic number constant to 78616d70 i.e.
'xamp' characters */
    const u32 comp_magic_num; /* complement of magic number */
    const u32 rsc_tbl_size;  /* size of resource table */
    const uintptr_t rsc_tbl; /* address of resource table */
}__attribute__((packed));
```

The version of this structure is 1 and is mapped to the order of the structure fields. The magic_num field value is constant: 78616d70, and right after that ~(magic_num) constant which is expected to be: 879e928f.

Linux Driver Parsing

- During boot, the Linux driver looks for the magic number and the ~(magic number) at the pre-defined memory location. If found, it's assumed that the RPU firmware is loaded and already running.
- In this case, the attach operation executes and if the resource table was parsed successfully, the Remoteproc state is moved from `OFFLINE` to `ATTACHED`.

Once the RPMsg communication is done from the Linux side, it can either stop the remote processor or detach from the remote processor. The difference is that after the stop operation, the RPU powers down and Linux loads the firmware and starts again to run the RPU. However, the detach operation notifies the RPU that Linux is done with the RPMsg communication, but it can attach again meaning that the RPU firmware is still running.

Chapter 2

Libmetal

Overview

The libmetal library is maintained by the OpenAMP open source community. It provides common user APIs to access devices, handle device interrupts, and request memory across different operating environments.

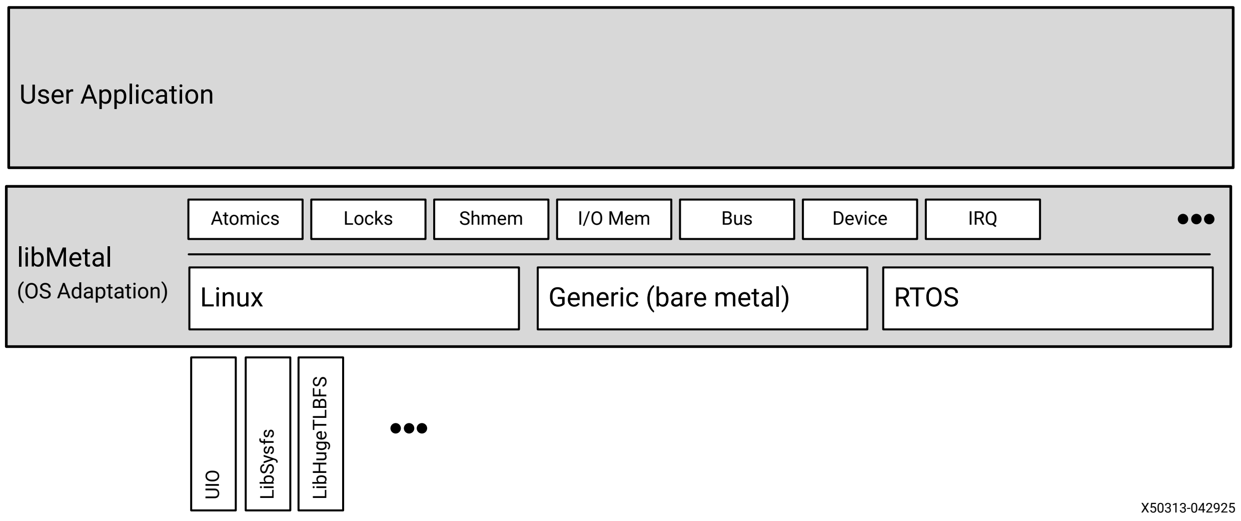
libmetal is available for the following operating systems/software configurations:

- Linux (Linux user space based on uniform I/O (UIO) and virtual function I/O (VFIO) support in the kernel.)
- FreeRTOS
- Bare-metal environments

***Note*:** VFIO support is no longer supported.

The following architecture diagram shows how the application accesses the libmetal library:

Figure 2: **Libmetal Architecture**



See the *AMD Libmetal Source Code* for more details on the libmetal APIs.

***Note*:** For new feature information on Libmetal (newer than 2019), see the AMD Wiki at https://wiki.xilinx.com/openamp.

Access Devices with Libmetal

Libmetal allows you to access devices similarly across varying operating environments.

The flow for using libmetal is as follows:

1. Start libmetal environment.
2. Add devices.
3. Open the devices.
4. Register interrupt if required.
5. Write and read device registers with libmetal API.
6. Close the device.
7. Close the libmetal environment.

The above steps are explained in the following subsections.

Different platforms can have different device abstractions. Following is a table to explain how libmetal manages devices differently:

Table 1: **Libmetal Devices**

Linux	Baremetal FreeRTOS
Devices are described in a device tree.	Because there is no device tree abstraction, devices must be defined statically before attempting to open them.
"platform" bus definition is in Linux kernel. It is used by Linux to present memory mapped devices.	No standard for bus abstraction. Libmetal library defines generic bus structure to manage devices.

Start Libmetal Environment, Add, and Open the Devices

```
struct metal_init_params metal_param = METAL_INIT_DEFAULTS;

metal_init(&metal_param);
```

1. Initialize libmetal environment with call to `metal_init()`.
2. Add devices:
   a. This step is only needed for Baremetal or FreeRTOS as there is no standard such as device tree used in Baremetal to describe devices.

b. Statically define the libmetal device and register it to the appropriate bus.

c. The following code snippet shows how to statically define the Triple Timer Counter device for Baremetal or FreeRTOS.

d. When initializing the `metal_device` struct provide the following: a name string, a bus for the device, the number of regions, table of each region in the device, a node to keep track of the device for the appropriate bus, the number of IRQs per device and an IRQ ID if necessary.

```
const metal_phys_addr ipi_phy_addr = 0xff310000;

static struct metal_device static_dev = {
    .name = "ff310000.ipi",
    .bus = NULL, // will be set later in metal_device_open()
    .num_regions = 1, // number of I/O regions
    .regions = {
        {
            .virt = (void *)0xff310000, // virtual address
            .physmap = &ipi_phy_addr,   // pointer to base physical address
of the I/O region
            .size = 0x1000,             // size of the region
            .page_shift = (-1UL),       // page shift; in baremetal/
FreeRTOS, memory is flat, no pages
            .page_mask = (-1UL),        // page mask
            .mem_flags = DEVICE_NONSHARED | PRIV_RW_USER_RW, // memory
attributes
            .ops = { NULL },            // no user-specific I/O region
operations
        }
    },
    .node = { NULL }, // will be set by libmetal later
    .irq_num = 1,     // number of interrupts of this device
    .irq_info = (void *)65 // interrupt information; here it's the IRQ
vector ID
};

// Register the device
metal_register_generic_device(&static_dev);
```

For libmetal in Linux userspace, devices need to be placed in the device tree. Here is an example:

```
amba {

       ipi_amp: ipi@ff340000 {

             compatible = "ipi_uio"; /* used just as a label as libmetal
will bind this device as UIO device */

             reg = <00x 0xff340000 0x0 0x1000>;

             interrupt-parent = <&gic>;

             interrupts = <0 29 4>;

        };

};
```

Open Devices.

Next, open the device to access the memory mapped device I/O regions and retrieve interrupts if applicable.

```
struct metal_device *dev;

… // instantiate device here

metal_device_open(BUS_NAME, DEVICE_NAME, &dev);
```

Register the Interrupt, Write, and Read Device Registers

This section assumes that you have already initialized the libmetal environment, register devices if necessary, and open these devices.

In Baremetal or FreeRTOS, you have to explicitly initialize the Generic Interrupt Controller (GIC) using the Inter-Processor Interrupt (IPI) and Shared Memory including libmetal as an example.

***Note*:** The following section refers to the IP integrator elements of the Zynq UltraScale+ MPSoC hardware as described in Chapter 13 of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) and Chapter 52 of the Inter-Processor Interrupts" of *Versal Adaptive SoC Technical Reference Manual* (AM011).

Close Device and Close Libmetal Environment

After using the libmetal APIs to talk to the devices, close the device and libmetal environment as follows:

```
/* Close the opened device */

metal_device_close(device);

/* Close the libmetal environment */

metal_finish();
```

Access IP Integrator and Shared Memory with Libmetal

Zynq UltraScale+ MPSoC IP Integrator Hardware

The inter processor interrupt (IPI) can be used for notification messages between processors. The following example does not use the IPI shared buffer. Libmetal does not provide IPI drivers. It only provides a way to interact with IPI as a device. You need to manage the IPI.

The libmetal library is used to access IPI as a generic device. You need to define how to access IPI in your application. Using a standalone IPI driver, the driver defines the method used to send and receive messages between IPI blocks.

***Note*:** Libmetal in Linux user space does not allow use of IPI buffer. Because the IPI buffer is only used for the interaction with PMU firmware and it can only be accessed from Arm® trusted firmware (ATF).

You can interact with the IPI registers via `metal_io_read32()` and `metal_io_write32()`, and handle IPI interrupt with libmetal IRQ APIs.

Following, is an example of how to access Zynq UltraScale+ MPSoC IPI registers, and handle IPI interrupts.

This is an example of an IPI libmetal device static definition for baremetal/FreeRTOS:

```
static struct metal_device ipi_dev = { /* IPI device */
    .name = "ff310000.ipi", /* device name */
    .bus = NULL, /* bus. This field is NULL as it does not need to be set.
It will be set in metal_device_open() */
    .num_regions = 1, /* number of I/O regions in device */
    .regions = { /* define structure of each I/O region in device */
        {
            .virt = (void*)0xFF3100, /* virtual address */
            .physmap = 0xFF3100, /* physical address */
            .size = 0x1000, /* size of region */
            .page_shift = (sizeof(metal_phys_addr_t) << 3), /* page shift */
            .page_mask = (unsigned long)(-1), /* page mask */
            .mem_flags = DEVICE_NONSHARED | PRIV_RW_USER_RW, /* memory
flags */
            .ops = {NULL}, /* user-defined memory operations. We do not
have any custom operations so leave this as NULL. */
        }
    },
    .node = {NULL}, /* node to point to device in list of nodes on bus.
This will be set in metal_device_open, so leave as NULL */
    .irq_num = 1, /* The number of IRQs per device. In this case we are
using only one interrupt. */
    .irq_info = (void*)65, /* IRQ ID */
};

/* Open the IPI device, use the IPI device as follows: */

/* open the IPI device */
metal_device_open("generic", "ff310000.ipi", &ipi);

/* Get the IPI device libmetal I/O region */
io_region = metal_device_io_region(ipi, 0);

/* disable IPI interrupt */
metal_io_write32(ipi_io_region, IPI_IDR_OFFSET, IPI_MASK);

/* clear old IPI interrupt */
metal_io_write32(ipi_io_region, IPI_ISR_OFFSET, IPI_MASK);

/* Register IPI irq handler */
metal_irq_register(ipi_irq, ipi_irq_handler, ipi_dev, private_data);

/* Enable IPI interrupt */
metal_io_write32(ipi_io_region, IPI_IER_OFFSET, IPI_MASK);
```

Shared Memory

Libmetal provides a way to access and interact with a memory device. However the memory type is user-defined.

In the Linux userspace, libmetal uses the UIO (Userspace I/O) driver so interaction is limited to treating the memory as device memory.

Libmetal provides I/O region abstraction that gives access to memory mapped I/O and shared memory regions. This includes primitives to read, and write memory with ordering constraints and the ability to translate between physical and virtual addressing on systems that support virtual memory.

Following is an example to statically define, open, read, and write from a shared memory device. This example shows a shared memory libmetal device with static definition for baremetal/ FreeRTOS:

```
static struct metal_device shm_dev = { /* shared memory device */
    .name = "3ed80000.shm", /* device name */
    .bus = NULL, /* device bus */
    .num_regions = 1, /* number of regions on device */
    {
        {
            .virt = (void*)0x3ED80000, /* virtual address */
            .physmap = 0x3ED80000, /* physical address */
            .size = 0x800000, /* size of region */
            .page_shift = (sizeof(metal_phys_addr_t) << 3), /* page shift */
            .page_mask = (unsigned long)(-1), /* page mask */
            .mem_flags = NORM_SHARED_NCACHE | PRIV_RW_USER_RW, /* memory
flags */
            .ops = {NULL}, /* user defined memory operations */
        }
    },
    .node = {NULL}, /* node to point to device in list of nodes on bus */
    .irq_num = 0, /* Number of IRQs per device. This is 0 because there are
no interrupts we want to use for this device.*/
    .irq_info = NULL, /* IRQ info. This is NULL because we are not using
this device for interrupts. */
};

/* Open the shared memory device, use the shared memory device as follows:
*/

/* Open the shared memory device */
ret = metal_device_open("platform", "3ed80000.shm", &dev); /* the first
argument, bus name, is 'platform' for generic platform. */

/* get shared memory device IO region */
io = metal_device_io_region(device, 0);

/* read data from the shared memory */
metal_io_block_read(io, READ_OFFSET, destination, data_length);

/* write data to the shared memory */
ret = metal_io_block_write(io, WRITE_OFFSET, source, data_length);
```

AMD Libmetal AMP Demo

The Libmetal AMP Demonstration Application describes how to open and access devices, namely shared memory and interrupts.

AMD Vitis™ unified software platform and PetaLinux tools include a libmetal demo to demonstrate how to use the libmetal library to build simple interprocessor communication between APU (Application Processing Unit) and RPU (Real-Time Processor) on a Zynq UltraScale+ MPSoC platform.

The example uses the following resources for the inter-processor communication:

- DDR memory.
- IP integrator (Inter Processor Interrupts) for notification.
- Triple Timer Counter for measurement of latency and throughput demonstrations.

The next section describes how to build the libmetal example with AMD Vitis™ Unified IDE™ unified software platform and Yocto project tools.

The Libmetal AMP demonstration includes:

- Shared memory
- Shared memory with atomics
- IP integrator with shared memory
- IP integrator latency measurement
- Shared memory latency measurement
- Shared memory throughput measurement

Build Libmetal Baremetal Firmware with AMD Vitis Unified IDE

1. Start the AMD Vitis™ unified software platform.
2. Create the BSP: **File → New Component → Platform**
    a. Select the name and location
    b. Select the design
    c. Select **Standalone** for the operating system
    d. Select **Processor**
    e. Select **Finish**

3. Add the Libmetal libraries.
   a. In the Vitis components view select the created platform > processor > domain > Board Support Package.
      i. Check Libmetal libraries.

   ***Note*:** To build an OpenAMP RPC demo so that the demo uses the OpenAMP provided proxy information, do the following:

4. Build the BSP: **View → Flow**then click **Build** in the flow view.
5. Create the application: Select **File → New Example → Libmetal AMP Demo**.
6. Select **Example Application → Select Platform → Next → Select Domain → Finish**.
7. Build the application: **View → Flow** the select the application project name and click **Build** in flow view.

Build Libmetal Firmware and Linux Application with Yocto Tools

SoC	YAML_NAME	MACHINE_NAME	Artifactory Link
KRIA SOM	../sources/meta-openamp/vendor/xilinx/meta-xilinx-standalone-sdt/recipes-openamp/libmetal/overlays/libmetal-overlay-zynqmp.yaml	k26-smk-sdt	Index of /sswreleases
VEK280	../sources/meta-openamp/vendor/xilinx/meta-xilinx-standalone-sdt/recipes-openamp/libmetal/overlays/libmetal-overlay-versal.yaml	versal-vek280-sdt-seg-ospi	Index of /sswreleases

1. Generate the Yocto project:

```
mkdir  yocto
curl https://storage.googleapis.com/git-repo-downloads/repo > repo
chmod a+x repo
 ./repo init -u https://github.com/Xilinx/yocto-manifests.git -b 2025.1
-m default-edf.xml
./repo sync
source edf-init-build-env
```

2. Generate conf/dts with gen machineconf run using the attached YAML and the SDT for the SoC:

```
<yocto project>/build$ # Current directory after previous step

export ARTIFACTORY_LINK=...
export MACHINE_NAME=...
export YAML_NAME=...

gen-machineconf parse-sdt $ARTIFACTORY_LINK \
     -c conf -l conf/sdt-autogen.conf \
     -g full --machine-name $MACHINE_NAME \
     --add-config CONFIG_YOCTO_BBMC_CORTEXR5_0_FREERTOS=y \
     --add-config CONFIG_YOCTO_BBMC_CORTEXR5_1_BAREMETAL=y \
     --domain-file $YAML_NAME;
```

3. Add the required packages to the Yocto build. Enable OpenAMP-related packages by appending the following to the conf/local.conf:

```
IMAGE_INSTALL:append = " packagegroup-openamp libmetal-fw-freertos
kernel-module-uio-pdrv-genirq kernel-module-uio-dmem-genirq "
```

4. Build the Yocto project:

```
MACHINE=<MACHINE_NAME> bitbake core-image-minimal
```

Build Libmetal Linux Demo in AMD Vitis (Optional)

PetaLinux uses meta-openamp to build libmetal library and the libmetal Linux demo application. If you want to create your own libmetal application, you can do it with the AMD Vitis™ unified software platform.

Following are the steps in the AMD Vitis™ unified software platform to generate the application.

1. Build and package sysroots.

```
$ petalinux-build -s

$ petalinux-package --sysroot
```

2. Run Vitis.
3. Create a new platform project for the Linux application select File > New Platform Project and specify a name for your platform project.

```
OS: Linux
{
    Processor: psu_cortexa53
    {
        Linux sysroot: the sysroot you built from your PetaLinux project:
        {
            "--sysroot=/<plnx-proj-root>/images/linux/sdk/sysroots/
aarch64-xilinx-linux"
        }
        Click Next
    }
}
```

4. Create a new platform from the hardware (XSA). Click **Browse** and import your .xsa file.
   - Select Operating System: Linux
   - Processor: psu_cortexa53
   - Deselect Generate boot components.
   - Using the boot components generated by the PetaLinux project and click **Finish**.

5. Create a Linux libmetal application project File > New Application Project > Next.
    - Select a platform from repository, select the Linux platform project that was created in the previous steps.
    - Specify the application project name, processor psu_cortexa53 SMP.
    - Specify the sysroots path in your PetaLinux project, browse to the directory below:
    - images/linux/sdk/sysroots/aarch64-xilinx-linux
    - Optional:
    - RootFS - rootfs.tar.gz
    - Kernel Image - image.ub
    - Available template - Linux empty Application
6.
    ```
    C/C++ Build • Settings
    Tool Setting Tab Libraries
    Libraries (-l) add "metal"
    ```
7. Copy files located at (https://github.com/OpenAMP/libmetal/tree/main/examples/system/linux/xlnx/zynqmp_amp_demo) to the application's `src` directory.
    - common.h
    - ipi_latency_demo.c
    - ipi_shmem_demo.c
    - ipi-uio.c
    - shmem_atomic_demo.c
    - shmem_demo.c
    - shmem_latency_demo.c
    - shmem_throughput_demo.c
    - sys_init.c
    - sys_init.h
    - libmetal_amp_demo.c

    ***Note*:** The demo talks to RPU 0 by default, if you want to change the demo to talk to RPU 1, change the IP integrator mask value in `common.h` to 0x200, which is the default RPU1 IP integrator mask.
8. Install the Linux application executable built from Vitis and firmware into the rootfs built with PetaLinux tools using a Yocto Recipe created by:

```
petalinux-create apps -n <app_name> --enable
```

Modify the `project-spec/meta-user/recipes-apps/<app_name>/<application name>.bb` to install the remote processor firmware in the RootFS as follows:

```
SUMMARY = "Simple test application"

SECTION = "PETALINUX/apps"

LICENSE = "MIT"

LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://<linux-app> \
file://<firmware> \
"

S = "${WORKDIR}"

INSANE_SKIP:${PN} = "arch"

RDEPENDS:${PN} += " \
    libmetal-xlnx \
    open-amp \
"

do_install() {
    # Install firmware into /lib/firmware on target
    install -d ${D}/lib/firmware
    install -m 0644 ${S}/<firmware> ${D}/lib/firmware/<firmware>

    # Install linux application into /usr/bin on target
    install -d ${D}/usr/bin
    install -m 0755 ${S}/<linux-app> ${D}/usr/bin/<linux-app>
}

FILES:${PN} = "/lib/firmware/<firmware> /usr/bin/<linux-app> "
```

Note: For libmetal linux side demo on Zynq UltraScale+ MPSoC to talk to RPU1,
modify the following: Changehttps://github.com/OpenAMP/libmetal/blob/main/examples/system/linux/xlnx/zynqmp_amp_demo/common.h from 0x100 to 0x200.

Build the Linux Demo Application and the Linux Project

1. Go to the PetaLinux tools project:

   ```
   $ cd <plnx_proj>
   ```

2. Build the PetaLinux project:

   ```
   $ petalinux-build
   ```

The kernel images and the device tree binary are located in the `<plnx-proj-root>/images/linux` directory.

Testing on Hardware

1. Go to the PetaLinux project:

   ```
   $ cd <plnx_proj>
   ```

2. Build the PetaLinux project:

   ```
   $ petalinux-build
   ```

3. Run PetaLinux boot:

   ```
   $ petalinux-boot jtag --kernel
   ```

   If you encounter any issues, append `-v` to these commands to see the textual output.

4. Boot RPU firmware with remoteproc sysfs.

   The firmware should be placed in the `/lib/firmware` directory.

   ```
   $ echo <firmware_name> > /sys/class/remoteproc/remoteproc0/firmware

   $ echo start > /sys/class/remoteproc/remoteproc0/state
   ```

   You can also use other methods to boot Linux on APU and the firmware on RPU, such as SD boot. This example only documents JTAG boot.

5. On the APU Linux target console, run the demo application on the Linux application you built with XVitis or use the prebuilt "libmetal_amp_demo" provided with PetaLinux BSP. This process produces output similar to the following:

   ```
   # <linux libmetal application

   metal: warning:   skipped page size 2097152 - invalid args

   CLIENT> ****** libmetal demo: shared memory ******
   metal: info:      meta

   SERVER> Demo has started.
   SERVER> Shared memory test finished

   SERVER> ====== libmetal demo: atomic operation over shared memory ======
   SERVER> Starting atomic add on shared memory demo.

   l_uio_dev_open: No IRQ for device 3ed80000.shm.

   CLIENT> Setting up shared memory demo.
   CLIENT> Starting shared memory demo.
   CLIENT> Sending message: Hello World - libmetal shared memory demo
   CLIENT> Message Received: Hello World - libmetal shared memory demo
   CLIENT> Shared memory demo: Passed.

   CLIENT> ****** libmetal demo: atomic operation over shared memory ******
   ```

***Note*:** One method with which the application can be debugged is XSDB. See the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) for more information on the use of XSDB.

Straight Line Speculation Vulnerability

In 2020 Arm announced CVE-2020-13844 (https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability) which allows malevolent code to obtain secret data via a similar mechanism to Spectre. This vulnerability does not affect the Cortex R5 used in the AMD UltraScale™ and AMD Versal™ device RPUs, and also does not affect the Cortex-A53 used in UltraScale+ devices.

The mitigation for this vulnerability is to add the compiler option "-mharden-sls=all" to the build flags for the open-amp and libmetal libraries, and to the build flags for any application which is linked to them. This results in 2-5% larger executable code and probably a slight performance decrease.

***Note*:** This compiler option is only available in 2022.2 and after.

Chapter 3

OpenAMP

Overview

Open Asymmetric Multi-processing (OpenAMP) is a framework providing the software components needed to enable the development of software applications for asymmetric multi-processing (AMP) systems. The framework provides the following key capabilities.

- Provides Life Cycle Management, and Inter Processor Communication capabilities for management of remote compute resources and their associated software contexts.
- Provides a standalone library usable with RTOS and Bare-metal software environments.
- Compatibility with upstream Linux remoteproc, rpmsg, and VirtIO components.

Components in OpenAMP

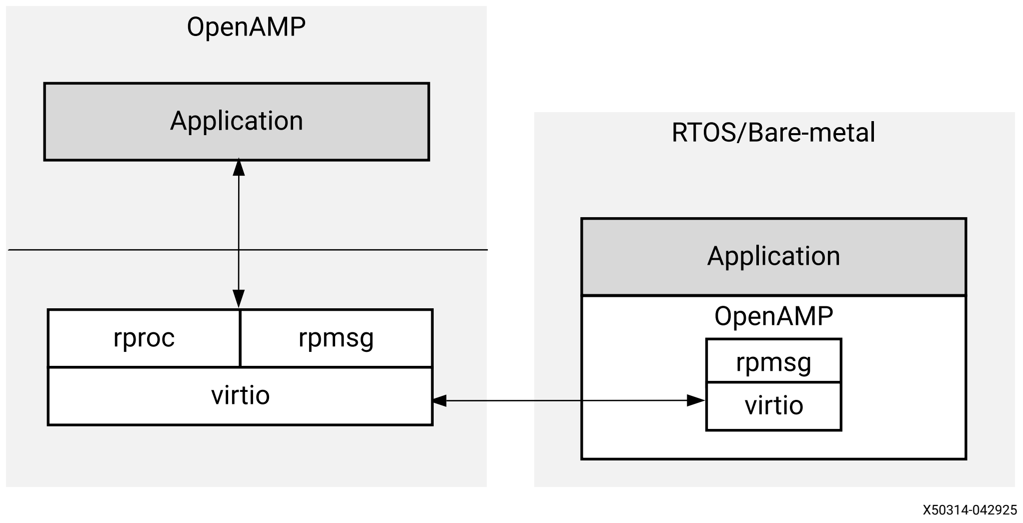
RPMsg (Remote Processor Messaging), VirtIO (Virtualization Module), and remoteproc are implemented in upstream Linux kernel. OpenAMP library provides the implementation for these components for the following environments: Baremetal, FreeRTOS (Real-Time Operating System), and Linux userspace.

- **virtIO:** OpenAMP library implements virtIO standard for shared memory management. The virtIO is a virtualization standard for network and disk device drivers where only the driver on the guest device is aware it is running in a virtual environment, with the hypervisor.
- **remoteproc:** Remoteproc provides capability for life cycle management (LCM) of the remote processors. The remoteproc API that OpenAMP library uses is compliant with the infrastructure present in the Linux Kernel 3.18 and later. The remoteproc uses information published through the remote processor firmware resource table to allocate system resources and to create virtIO devices. The remoteproc can be used to load arbitrary firmware; it is not limited to OpenAMP firmware.
- **RPMsg:** This API allows inter-process communications (IPC) between software running on independent cores in an AMP system. This is also compliant with the RPMsg bus infrastructure present in the Linux Kernel version 3.18 and later.

The following diagrams show how OpenAMP is used in MPSoC platforms:

1. Linux kernel master and RPU OpenAMP slave.

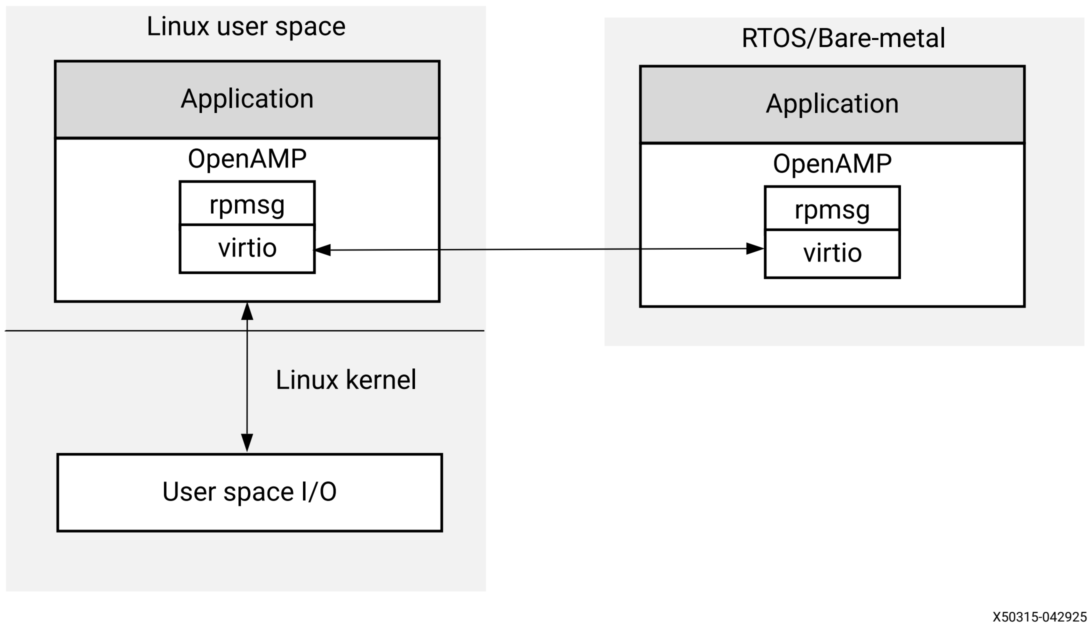
Figure 3: **RPMsg Implementation in Kernel Space**



Linux kernel space provides RPMsg and Remoteproc, but the RPU application requires Linux to load it to talk to the RPMsg counterpart in the Linux kernel. This is the Linux kernel RPMsg and Remoteproc implementation limitation.

2. Linux userspace OpenAMP application and RPU OpenAMP application.

Figure 4: **OpenAMP RPMsg Implementation in Linux Userspace**

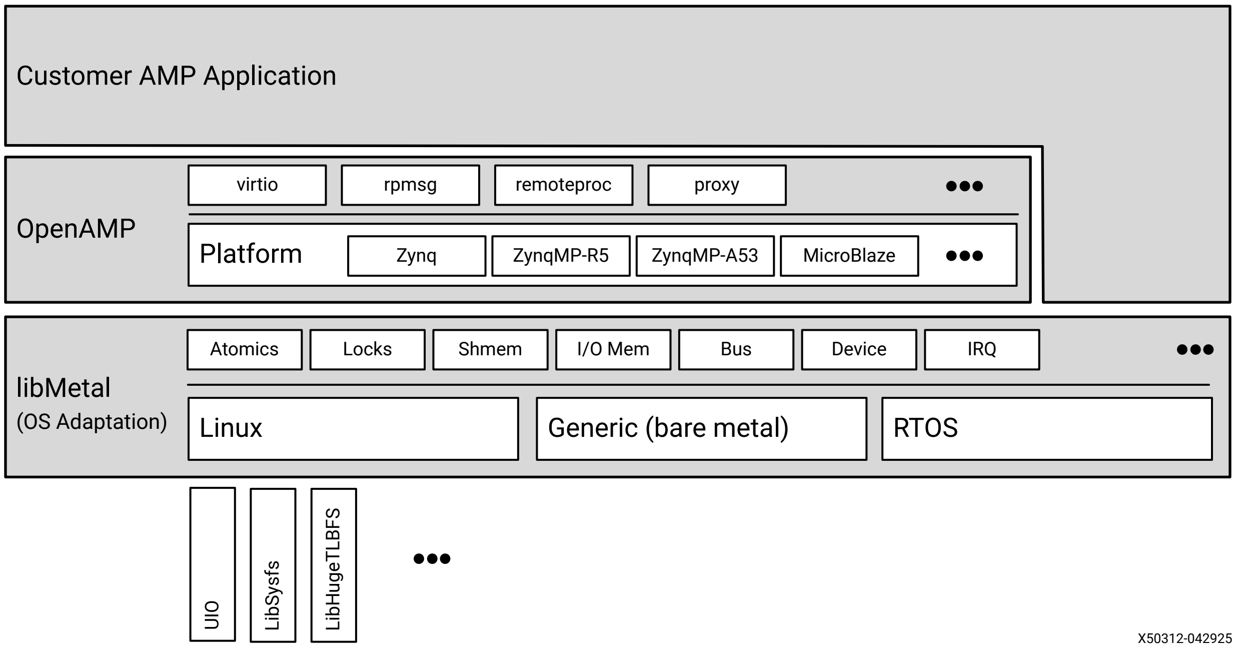


OpenAMP library can also be used in Linux userspace. In this configuration, the remote processor can run independently to the Linux host processor.

Connection between OpenAMP and Libmetal

OpenAMP uses Libmetal as an abstraction layer to access devices, handle interrupts and shared memory. Libmetal is used because it provides a uniform interface for accessing devices and memory. OpenAMP uses libmetal to access Inter-Processor Interrupt (IPI) and shared memory. OpenAMP leverages standards for shared memory management, lifecycle management and communication. A diagram to show the connection between libmetal and OpenAMP is as follows:

Figure 5: **Libmetal and OpenAMP Connection**



Writing a Simple OpenAMP Application

To write an OpenAMP application, follow these steps:

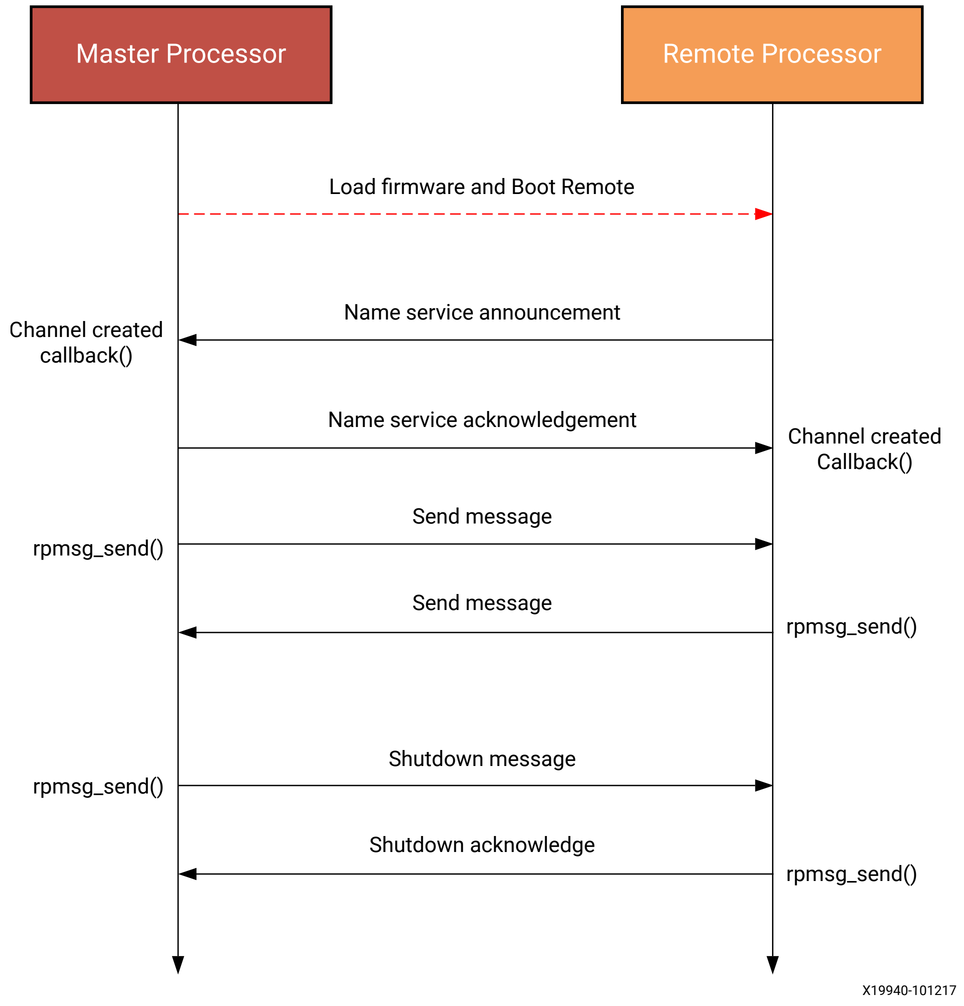
1. A firmware resource table.

   The resource table defines the necessary firmware entries for the OpenAMP application. It is a list of system resources required by the remote_proc.

2. Create remoteproc struct using resource table.
3. Define RPMsg callback functions.
4. Create RPMsg virtio device.
5. Create an RPMsg endpint and associate the RPMsg device with the callback functions.
6. Use `rpmsg_send()` to send message across to the remote processor.

After initializing the framework, the flow of an OpenAMP application consists of the RPMsg channel acting as communication between the master and remote processor via the RPMsg `send()` and I/O callback functions. The following is a flow diagram to show this.

Figure 6: **Flow Diagram**



Following is a sample OpenAMP set up and flow with a resource table, Remoteproc instance and RPMsg callback functions:

```
struct resource_table table = {
    /* Version number. If the structure changes in the future, this acts as
     * reference to what the structure is.
     */
    .ver = 1,

    /* Number of resources; Matches number of offsets in array */
    .num = 2,

    /* reserved (must be zero) */
    .reserved = 0,

    { /* array of offsets pointing at various resource entries */
        /* This RSC_RPROC_MEM entry sets the shared memory address range.
It tells the Linux kernel the shared memory range the remote can access. */
        {RSC_RPROC_MEM, 0x3ed40000, 0x3ed40000, 0x100000, 0},

        /* virtio device header */
        {
            RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0, 0,
            NUM_VRINGS, {0, 0},
        }
    }
};

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

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

/* Remoteproc instance */
struct remoteproc rproc;

/* RPMsg VirtIO device instance */
struct rpmsg_virtio_device rpmsg_vdev;

/* RPMsg device */
struct rpmsg_device *rpmsg_dev;

/* Resource Table */
void *rsc_table = &resource_table;

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

/* Shared memory metal I/O region */
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 */
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 */
    vdev = remoteproc_create_virtio(&rproc, 0, VIRTIO_DEV_MASTER, NULL);

    /* Initialize the shared buffers pool (only if VirtIO master) */
    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 */
    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);

    /* Wait until endpoint is ready */
    while (!is_rpmsg_ept_read(&ept)) {
        rproc_virtio_notified(vdev, RSC_NOTIFY_ID_ANY);
    }

    /* Send RPMsg */
    rpmsg_send(&ept, data, size);

    do {
        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 and clean up */
    rpmsg_destroy_ept(&ept);
    rpmsg_deinit_vdev(&rpmsg_vdev);
    remoteproc_remove_virtio(&rproc, vdev);
    remoteproc_remove(&rproc);
}
```

OpenAMP Demos

Following are descriptions for each of the OpenAMP demonstration applications.

Echo Test in Linux Master and Baremetal or FreeRTOS Remotes

This test application sends a number of payloads from the master to the remote and tests the integrity of the transmitted data.

- The echo test application uses the Linux master to boot the remote Baremetal firmware using remoteproc.
- The Linux master transmits payloads to the remote firmware using RPMsg. The remote firmware echoes back the received data using RPMsg.
- The Linux master verifies and prints the payload.

Matrix Multiplication for Linux Master and Baremetal or FreeRTOS Remotes

The matrix multiplication application provides a more complex test that generates two matrices on the master. These matrices are sent to the remote, which is used to multiply the matrices. The remote sends the result back to the master, which displays the result.

The Linux master boots the Baremetal remote firmware using remoteproc. It transmits two randomly-generated matrices using RPMsg.

The Baremetal firmware multiplies the two matrices and transmits the result back to the master using RPMsg.

Proxy Application for Linux Masters and Baremetal or FreeRTOS Remotes

This application creates a proxy between the Linux master and the remote core, which allows the remote firmware to use console and execute file I/O on the master.

The Linux master boots the firmware using the proxy_app. The remote firmware executes file I/O on the Linux file system (FS), which is on the master processor. The remote firmware also uses the master console to receive input and display output.

OpenAMP Quick Try

You can find domain YAMLs for ZynqMP, AMD Versal™ Gen 1, and AMD Versal™ 2VE/2VM devices in the meta-xilinx layer on GitHub, at: https://github.com/Xilinx/meta-xilinx/tree/rel-v2025.2/meta-xilinx-standalone-sdt/conf/domainyaml

This location is part of the Yocto meta-xilinx repository.

Building and Booting OpenAMP Images

This page provides build and boot instructions for Yocto EDF images: Getting Started - Walk through examples. Use the following table to identify which SoCs and corresponding images to select for OpenAMP demos.

Table 2: **Supported Machines**

SoC	Build RootFS and Kernel	Build Boot BIN
KRIA KV260 and KRIA KR260 SOM	MACHINE=kria-zynqmp-generic bitbake kria-image-full-cmdline	MACHINE=k26-smk-sdt bitbake xilinx-bootbin
VEK280	MACHINE=amd-cortexa72-common bitbake edf-linux-disk-image	MACHINE=versal-vek280-sdt-seg bitbake xilinx-bootbin
Versal-2ve-2vm	MACHINE=amd-cortexa78-mali-common bitbake edf-linux-disk-image	MACHINE=versal-2ve-2vm-vek385-sdt-seg bitbake xilinx-bootbin

If you want to generate your own image, append the following lines to your Yocto project's `local.conf`:

Table 3: **Configuration Entries for local.conf to Enable Custom Image Generation**

SoC	MACHINE NAME	Local.conf Contents to Append
KRIA KV260 and KRIA KR260 SOM	kria-zynqmp-generic	`IMAGE_INSTALL:append = " \ vek385-openamp-fw-examples \ rpmsg-utils \ packagegroup-openamp "`
VEK280	amd-cortexa72-common	`IMAGE_INSTALL:append = " \ vek280-openamp-fw-examples \ packagegroup-openamp "`
Versal-2ve-2vm	amd-cortexa78-mali-common	`IMAGE_INSTALL:append = " \ openamp-zephyr-demo \ rpmsg-utils \ packagegroup-openamp "`

Linux Boot from U-Boot and OpenAMP Application Run

The following steps describe how to boot Linux on an AMD Zynq™ UltraScale+™ MPSoC board using U-Boot from an SD card (or JTAG as an alternative) and then run the OpenAMP echo-test demo to verify communication between the APU and RPU.

1. Go to the U-Boot prompt and boot Linux from the SD card:

   ```
   ZynqMP> mmcinfo && fatload mmc 0 0x10000000 image.ub && fatload mmc 0
   0x14000000 openamp.dtb

   Device: mmc@ff170000

   Manufacturer ID: 3

   OEM: 5344

   Name: SL16G

   Bus Speed: 100000000

   Mode: UHS SDR50 (100MHz)

   Rd Block Len: 512

   SD version 3.0

   High Capacity: Yes

   Capacity: 14.8 GiB

   Bus Width: 4-bit

   Erase Group Size: 512 Bytes

   57212600 bytes read in 3718 ms (14.7 MiB/s)

   40961 bytes read in 26 ms (1.5 MiB/s)

   ZynqMP> bootm 0x10000000 0x10000000 0x14000000
   ```

   Note: As an alternative to all steps above to SD boot, you can JTAG boot the board. For this you need to have connected a JTAG cable, installed JTAG drivers, and created a PetaLinux project using a provided BSP.

   To do this, you must go in the `<your project>/pre-built/linux/images` directory and replace the `system.dtb` file by `openamp.dtb`. Next, type `petalinux-boot --jtag --prebuilt 3`.

2. Run the echo-test demo.

   ```
   root@plnx_aarch64:~# echo image_echo_test > /sys/class/remoteproc/
   remoteproc0/firmware

   root@plnx_aarch64:~# echo start > /sys/class/remoteproc/remoteproc0/state

   [  265.772355] remoteproc remoteproc0: powering up
   ff9a0100.zynqmp_r5_rproc
   ```

```
[  265.779900] remoteproc remoteproc0: Booting fw image
echotest_standalone_r5_0.elf, size 719860

[  265.790005] zynqmp_r5_remoteproc ff9a0100.zynqmp_r5_rproc: RPU boot
from TCM.

Starting application...

Initialize remoteproc successfully.

creating remoteproc virtio

initializing rpmsg shared buffer pool

initializing rpmsg vdev

initializing rpmsg vdev

Try to create rpmsg endpoint.

Successfully created rpmsg endpoint.

[  265.797738] remoteproc remoteproc0: registered virtio0 (type 7)

[  265.800388] virtio_rpmsg_bus virtio0: rpmsg host is online

[  265.830254] remoteproc remoteproc0: remote processor
ff9a0100.zynqmp_r5_rproc is now up

[  265.838381] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-
demo-channel addr 0x0

root@xilinx-zcu102-2019_1:/lib/firmware# echo_test -d virtio0.rpmsg-
openamp-demo-channel.-1.0

Echo test start

Open rpmsg dev /dev/rpmsg0!

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

Echo Test Round 0

**************************************
```

***Note*:** This rpmsg device driver is an out-of-tree Linux kernel module. It can be loaded at boot time if you write a start-up init script (See examples in *PetaLinux Tools Documentation: Reference Guide* (UG1144).

Running the Demos

For demos on VEK280 and KRIA SOM, see Building OpenAMP Application for RPU Firmware, for instructions on running the legacy OpenAMP RPU firmware applications.

Demo to Run Zephyr on R52-0 for Versal-2ve-2vm

The following steps enable you to run the OpenAMP demo on Versal 2VE/2VM:

1. Identify the correct `remoteproc` node. After booting, run the following:

   ```
   dmesg | grep remoteproc
   ```

   Example output:

   ```
   [    2.794038] remoteproc remoteproc0: ebb80000.r52f is available
   [    2.800037] remoteproc remoteproc1: ebbc0000.r52f is available
   [    2.806278] remoteproc remoteproc2: ebc00000.r52f is available
   [    2.812242] remoteproc remoteproc3: ebc40000.r52f is available
   [    2.818393] remoteproc remoteproc4: eba00000.r52f is available  #
   correct node
   ```

   In this example, `remoteproc` corresponds to `remoteproc4` in `/sys/class/remoteproc/remoteproc4`, but depending on the kernel probe, this can change. Notice the ending `EBA00000`.

2. Set the firmware path to the location of the firmware ELF file:

   ```
   /lib/firmware/xilinx/vek385/rpu/vek385-r5-0-zephyr-openamp-rpmsg-multi-
   srv/
       vek385-r5-0-zephyr-openamp-rpmsg-multi-srv_slot0/
       vek385-r5-0-zephyr-openamp-rpmsg-multi-srv.elf
   ```

   Save the path without `/lib/firmware/`:

   ```
   export FW=xilinx/vek385/rpu/vek385-r5-0-zephyr-
   openamp-rpmsg-multi-srv/vek385-r5-0-zephyr-openamp-rpmsg-multi-srv_slot0/
   vek385-r5-0-zephyr-openamp-rpmsg-multi-srv.elf
   ```

3. Start the remote processor by running the following as root:

   ```
   sudo -s
   echo $FW > /sys/class/remoteproc/remoteproc4/firmware
   echo start > /sys/class/remoteproc/remoteproc4/state
   ```

Use the following command to communicate through RPMsg:

```
cat /dev/ttyRPMSG0 &
echo "Hello Zephyr" > /dev/ttyRPMSG0
# Output: TTY 0: Hello Zephyr
echo "Goodbye Zephyr" > /dev/ttyRPMSG0
# Output: TTY 0: Goodbye Zephyr
```

Building OpenAMP Application for RPU Firmware

Introduction

The AMD Vitis™ unified software platform contains templates to aid in the development of OpenAMP Bare-metal/FreeRTOS remote applications. The following sections describe how to create OpenAMP applications with AMD Vitis™ unified software platform and PetaLinux tools.

Use the AMD Vitis™ unified software platform to create the Bare-metal or FreeRTOS remote applications.

Building Remote Applications in AMD Vitis

You can build remote applications using the AMD Vitis™ unified software platform by using the following procedures. The PetaLinux BSP already includes pre-built firmware for a remote processor (AMD Zynq™ UltraScale+™ MPSoCCortex-R5F#0); The following steps are necessary only if you plan to re-build the demo applications running on the remote processor.

Create and Build OpenAMP Applications in Vitis (Unified IDE)

1. Launch the AMD Vitis™ unified software platform.
2. In Vitis, click **File→New Component →Platform**
   a. Enter the name and location for the BSP.
   b. Select the design.
   c. For the operating system, select **Standalone**.
   d. Select the processor.
   e. Click **Finish**.
3. Add the Libmetal and OpenAMP libraries.
   a. In the Vitis components view, select the created platform: **Processor→Domain→Board Support Package**.
   b. Check the boxes for the Libmetal and OpenAMP libraries.

   ***Note*:** You can optionally build an OpenAMP RPC demo that uses the OpenAMP provided proxy information. To do this, follow these instructions:

   1. Navigate to the Vitis components view.
   2. Select the created platform by clicking **processor→domain→Board Support Package→openamp**.
   3. Set the `OPENAMP_WITH_PROXY` flag to true.

   ***Note*:** To use an external repository for OpenAMP or Libmetal libraries, use the following steps:

   1. Create a new directory with the following structure:

   ```
   'libmetal' or 'openamp'
   - data
   - src
   ```

   2. Inside the `data` folder, copy the YAML from the openamp.yaml or libmetal.yaml GitHub repositories.
   3. Inside the `src` folder, copy your modified version of the OpenAMP or Libmetal repository and the `sdt` directory from the libmetal/src/sdt GitHub repository.

4. Point Vitis to your local copy: **Vitis → Embedded SW Repositories → Local Repositories**. Click **+** to add the top level new directory with `data` and `src` inside and then click **OK**.
5. Click **Rescan Repositories**.
6. When adding the library, select the local copy from the drop down menu.

***Note*:** Instead of adding external libraries to Vitis, you should clone the upstream Libmetal and OpenAMP libraries and build them with native CMake.

4. Build the BSP in the flow view by clicking **Build** in the **View → Flow** view.
5. Create the application by selecting **File → New Example → Libmetal AMP Demos**. Select one of the following new components supported demos:
   - OpenAMP RPC Demo
   - OpenAMP matrix multiplication Demo
   - OpenAMP echo-test
   - Libmetal AMP Demo
6. Select **Example Application → Select Platform → Next → Select Domain → Finish**.
7. Build the application by selecting the application project mode in the **View → Flow** menu. Click **Build** in the flow view.

OpenAMP AMD Vitis Key Source Files

The following key source files are available in the AMD Vitis™ unified software platform application.

- Platform Information (platform_info.c/.h): These files contain hard-coded, platform-specific values used to get necessary information for OpenAMP.
  - `#define IPI_IRQ_VECT_ID`: The Inter-Processor Interrupt (IPI) vector of IP integrator agent used for interprocessor communication.
  - `#define IPI_BASE_ADDR`: The base address of IP integrator agent used for interprocessor communication.
  - `#define IPI_CHN_BITMASK`: The IPI bit mask for remote processor. This is necessary because the bit mask identifies which remote processor to communicate with. Bit mask information can be found in the TRM.
- Resource Table (rsc_table.c/.h): The resource table contains entries that specify the memory and virtIO device resources. The virtIO device contains device features, vring addresses, size, and alignment information. The resource table entries are specified in rsc_table.c and the remote_resource_table structure is specified in rsc_table.h.
- Helper (helper.c/.h): They contain platform-specific APIs that allow the remote application to communicate with the hardware. They include functions to initialize and control the GIC.

- Application code (src/<application>.c): In the src directory of the application in the XVitisunified software platform, the specific application is located (rpmsg-echo.c/ matrix_multiply.c/rpc_demo.c).

Running the Example Applications

After the system is up and running, log in with the username and password root. After logging in, the following example applications are available:

- Running the Echo Test
- Running the Matrix Multiplication Test
- Running the Proxy Application

IMPORTANT! *After booting the Linux kernel, the remoteproc driver is already loaded. If not, check that it has been enabled in the kernel configuration and check your device tree.*

Running the Echo Test

1. Load the Echo test firmware and `RPMsg` module:

   ```
   echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware

   echo start > /sys/class/remoteproc/remoteproc0/state
   ```

2. Run the test:

   ```
   echo_test
   ```

   The test starts.

3. Follow the on-screen instructions to complete the test.
4. After you have completed the test, unload the application:

   ```
   echo stop > /sys/class/remoteproc/remoteproc0/state
   ```

Debugging an OpenAMP Application

Debugging RPU Firmware

The Cortex-R5 firmware build with Vitis/Vivado saves its runtime messages into a remoteproc trace buffer. The firmware's resource table communicates the buffer address and size to Linux kernel. The remoteproc exports this buffer via debugFS and it can be viewed by reading /sys/kernel/debug/remoteproc/remoteproc0/trace0, for example: cat /sys/kernel/ debug/remoteproc/remoteproc0/trace0.

Below is an example to debug the echo test example running on RPU 0 with AMD System Debugger (XSDB). In this example, the function platform_init is found in platform_info.c at line 295 and is compiled to be at the address 0x3ed011c8. The below example shows how to set and run up to a breakpoint and print the value of local variables in the scope stopped at the breakpoint.

```
xsdb% bpadd -addr 0x3ed011c8

0

xsdb% Info: Breakpoint 0 status:
{
    target 7: { Address: 0x3ed011c8 Type: Hardware }
}

xsdb% dow ~/test.elf

Downloading Program -- ~/test.elf
{
    section, .vectors: 0x00000000 - 0x0000051f
    section, .text: 0x3ed00000 - 0x3ed0d73f
    section, .init: 0x3ed0d740 - 0x3ed0d74b
    section, .fini: 0x3ed0d74c - 0x3ed0d757
    section, .rodata: 0x3ed0d758 - 0x3ed0ee8f
    section, .data: 0x00000520 - 0x00001623
    section, .resource_table: 0x00001700 - 0x000017ff
    section, .eh_frame: 0x3ed0ee90 - 0x3ed0ee93
    section, .ARM.exidx: 0x3ed0ee94 - 0x3ed0ee9b
    section, .init_array: 0x3ed0ee9c - 0x3ed0eea3
    section, .fini_array: 0x3ed0eea4 - 0x3ed0eea7
    section, .bss: 0x3ed0eea8 - 0x3ed0f157
    section, .heap: 0x00001800 - 0x000057ff
    section, .stack: 0x00005800 - 0x00008fff
}

100%    0MB   0.3MB/s  00:00

Setting PC to Program Start Address 0x00000000

Successfully downloaded ~/test.elf

xsdb% con

xsdb% Info: Cortex-R5 #0 (target 7) Stopped at 0x3ed011c8 (Breakpoint)

platform_init() at ../src/platform_info.c: 295
{
295: {
}

xsdb% locals
{
    argc      : 0
    argv      : 0
    platform  : 0
    proc_id   : 0
    rsc_id    : 1053874736
    rproc     : 1053824852
}
```

```
xsdb% con

Info: Cortex-R5 #0 (target 7) Running

xsdb%
```

Running the Matrix Multiplication Test

1. Load the Matrix Multiply firmware and RPMsg module:

   ```
   echo image_matrix_multiply > /sys/class/remoteproc/remoteproc0/firmware

   echo start > /sys/class/remoteproc/remoteproc0/state
   ```

2. Run the test:

   ```
   mat_mul_demo
   ```

   The test starts.

3. Follow the on screen instructions to complete the test.
4. After you have completed the test, unload the application:

   ```
   echo stop > /sys/class/remoteproc/remoteproc0/state
   ```

Running the Proxy Application

1. Load and run the proxy application in one step. The proxy application automatically loads the required modules:

   ```
   proxy_app
   ```

2. When the application prompts you to Enter name, enter any string.
3. When the application prompts you to Enter age, enter any integer.
4. When the application prompts you to Enter value for pi, enter any floating point number.
5. The application prompts you to re-run the test.
6. After you exit the application, the module unloads automatically.

Creating a Boot.bin File

To create a Boot.bin file run the following command by passing the bif file as shown in the example. (Bootgen.bif is the name of the bif file).

```
$ petalinux-package --boot --bif

bootgen.bif --force
```

Testing on Hardware

1. Go to your PetaLinux project:

   ```
   $ cd <plnx_proj>
   ```

2. Build the PetaLinux project:

   ```
   $ petalinux-build
   ```

3. Boot the RPU firmware built with AMD Vitis™ tools with the SD boot. Following is a BIF file example:

   ```
   the_ROM_image:

   {

   [fsbl_config] a53_x64

   [bootloader] <plnx_proj>/images/linux/zynqmp_fsbl.elf

   [destination_device=pl] <plnx_proj>/images/linux/system.bit

   [destination_cpu=pmu] <plnx_proj>/images/linux/pmufw.elf

   [destination_cpu=r5-0] <RPU firmware>

   [destination_cpu=a53-0, exception_level=el-3, trustzone] <plnx_proj>/
   images/linux/arm/bl31.elf

   [destination_cpu=a53-0, exception_level=el-2] <plnx_proj>/images/linux/u-
   boot.elf

   }
   ```

4. To ensure that the user-space demo kernel drivers for rpmsg are not loaded, see the following two examples that demonstrate the ways it can be accomplished:

   - Before loading the RPU firmware remove the kernel modules using the following command: `rm /lib/modules/6.1.30-xilinx-v2025.1/kernel/drivers/rpmsg/*`
   - If the firmware is loaded, check `lsmod` to list down the modules loaded and use `rmmod` to remove the respective kernel loadable modules, perform the previous step to remove kernel modules and reload the firmware.

   ```
   xilinx-zcu102-20251: lsmod
   {
       Module                Size        Used by
       uio_dmem_genirq       16384        0
       rpmsg_ctrl            16384        0
       rpmsg_char            16384        1 rpmsg_ctrl
       virtio_rpmsg_bus      20480        0
       rpmsg_ns              16384        1 virtio_rpmsg_bus
       rpmsg_core            16384        4
   virtio_rpmsg_bus,rpmsg_char,rpmsg_ctrl,rpmsg_ns
       zynqmp_r5_remoteproc  20480        0
       cfg80211              360448       0
       uio_pdrv_genirq       16384        0
   ```

```
    zocl                  196608       0
}

xilinx-zcu102-20251:/lib/firmware# rmmod rpmsg_ctrl
xilinx-zcu102-20251:/lib/firmware# rmmod rpmsg_char
xilinx-zcu102-20251:/lib/firmware# rmmod virtio_rpmsg_bus
xilinx-zcu102-20251:/lib/firmware# rmmod rpmsg_ns
xilinx-zcu102-20251:/lib/firmware# rmmod rpmsg_core
```

5. On the APU Linux target console, run the demo applications rpmsg-echo-ping-shared, matrix_multiply-shared, and rpc_demod-shared. This process produces output similar to the following:

```
root@xilinx-zcu102-2025_1:~# rpmsg-echo-ping-shared
{
    metal: info:      metal_uio_dev_open: No IRQ for device 3ed20000.shm.

    Successful initializing rpmsg vdev
    Try to create rpmsg endpoint.
    Successfully created rpmsg endpoint.
    ly open shm device.
    Successfully added shared memory
    Successfully probed IPI device
    Successfully initialized Linux r5 remoteproc.
    Successfully initialized remoteproc
    Calling mmap resource table.
    Successfully mmap resource table.
    Successfully set resource table to remoteproc.
    Creating virtio...
    Successfully created virtio device.
    initializing rpmsg vdev

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

    **********************************
    Test Results: Error count = 0
    **********************************

    Quitting application .. Echo test end
    rpmsg_channel_deleted

    WARNING rx_vq: freeing non-empty virtqueue
    WARNING tx_vq: freeing non-empty virtqueue
}

root@Xilinx-ZCU102-2025_1:~#

# matrix_multiply-shared
{
    ...
    CLIENT> Matrix multiply: sent : 296
    CLIENT> Quitting application .. Matrix multiplication end

    CLIENT> **********************************
    CLIENT> Test Results: Error count = 0
    CLIENT> **********************************

    CLIENT> rpmsg_channel_deleted

    WARNING rx_vq: freeing non-empty virtqueue
    WARNING tx_vq: freeing non-empty virtqueue
}

root@Xilinx-ZCU102-2025_1:~#

# rpc_demod-shared
{
    login[1900]: root login on 'ttyPS0'

    root@Xilinx-ZCU102-2025_1:~# proxy_app-openamp
    {
        ...
        Master> Remote proc resource initialized.
        Master> RPMSG channel has created.

        Remote> FreeRTOS Remote Procedure Call (RPC) Demonstration
        Remote> ***************************************************
        Remote> Rpmsg based retargetting to proxy initialized..
        Remote> FileIO demo ..
        Remote> Creating a file on master and writing to it..
        … …

        Remote> Repeat demo ? (enter yes or no)
        no

        Remote> RPC retargetting quitting ...
        Remote> Firmware's rpmsg-openamp-demo-channel going down!

        Master> RPC service exiting !!
        Master> sending shutdown signal.

        WARNING rx_vq: freeing non-empty virtqueue
        WARNING tx_vq: freeing non-empty virtqueue
    }
}
```

Deprecated Platforms for OpenAMP

Zynq 7000 Support for OpenAMP is deprecated. AMD has retired OpenAMP support for AMD Zynq™ 7000 SoCs in the Linux kernel. The final officially supported version of OpenAMP on AMD Zynq™ 7000 platforms was in the 2023.2 release. For customers that are interested in porting or maintaining this code in their own projects, AMD has created the table of important links below. OpenAMP remains officially supported on all Zynq UltraScale+ MPSoC and Versal adaptive SoCs.

Table 4: **Zynq 7000 SoC Last Supported OpenAMP Release and Reference Code**

GitHub Repository	Version / Branch	Reference Implementation
openamp	Xilinx/linux-xlnx at xlnx_rebase_v6.1_LTS_2023.1_update (github.com)	open-amp/cmake/platforms at xlnx_rel_v2023.2 · Xilinx/open-amp (github.com)

Table 4: **Zynq 7000 SoC Last Supported OpenAMP Release and Reference Code** *(cont'd)*

GitHub Repository	Version / Branch	Reference Implementation
libmetal	Xilinx/libmetal at xlnx_rel_v2023.2 (github.com)	libmetal/cmake/platforms at xlnx_rel_v2023.2 · Xilinx/libmetal (github.com) libmetal/lib/system/generic/zynq7 at xlnx_rel_v2023.2 · Xilinx/libmetal (github.com) libmetal/lib/system/freertos/zynq7 at xlnx_rel_v2023.2 · Xilinx/libmetal (github.com)
linux-xlnx	Xilinx/linux-xlnx at xlnx_rebase_v6.1_LTS_2023.1_update (github.com)	linux-xlnx/drivers/remoteproc/zynq_remoteproc.c at xlnx_rebase_v6.1_LTS_2023.1_update · Xilinx/linux-xlnx (github.com) **Required additional patches which are required** irqchip: gic: Add changes to handle SGI for Zynq Remoteproc driver · Xilinx/linux-xlnx@b6bbbdb (github.com) irqchip: gic: Add remoteproc changes to the driver · Xilinx/linux-xlnx@220d249 (github.com)
meta openamp	Xilinx/meta-openamp at rel-v2023.2 (github.com)	meta-openamp/classes/yocto-cmake-translation.bbclass at rel-v2023.2 · Xilinx/meta-openamp (github.com)
EmbeddedSW	Xilinx/embeddedsw at xlnx_rel_v2023.2 (github.com)	**Sample reference for openamp echo test application** embeddedsw/lib/sw_apps/openamp_echo_test/src/machine/zynq7 at xlnx_rel_v2023.2 · Xilinx/embeddedsw (github.com) embeddedsw/lib/sw_apps/openamp_echo_test/src/system/generic/machine/zynq7 at xlnx_rel_v2023.2 · Xilinx/embeddedsw (github.com) **Reference related to libmetal** embeddedsw/ThirdParty/sw_services/libmetal/src/libmetal/cmake/platforms at xlnx_rel_v2023.2 · Xilinx/embeddedsw (github.com)
meta-xilinx	Xilinx/meta-xilinx at rel-v2023.2 (github.com)	meta-xilinx/meta-xilinx-core/dynamic-layers/openamp-layer/recipes-bsp/device-tree/files at rel-v2023.2 · Xilinx/meta-xilinx (github.com)

System Design Consideration

This chapter provides information on what various aspects of OpenAMP and Libmetal provide.

Supported Configuration

RPMsg kernel space refers to the kernel drivers implementing VirtIO, RPMsg, and Remoteproc and that *RPMsg user space* refers to the OpenAMP implementation of VirtIO, RPMsg, and Remoteproc.

Table 5: **Features**

	Linux Kernel RPMsg/ Remoteproc on APU + OpenAMP Library Used on RPU	OpenAMP Library Used on Linux Userspace + OpenAMP Library Used on RPU	Libmetal Library Used on Both APU and RPU
Linux boots RPU (RPU is a coprocessor to Linux APU host)	Yes See OpenAMP Quick Try	Yes	Yes See AMD Libmetal AMP Demo
Supports warm restart: Auto APU/RPU reconnect after APU restart	Yes See https://www.wiki.xilinx.com/OpenAMP	No	User defined
Supports pre-defined shared memory range	Yes See Writing a Simple OpenAMP Application	Yes	Yes See Shared Memory and Build Libmetal Firmware and Linux Application with Yocto Tools
Linux can dynamically allocate shared memory range	Yes See Writing a Simple OpenAMP Application	No	No
Supports Multiple communication channels (For example: both RPUs)	Yes See OpenAMP Demos	Yes See OpenAMP Demos	Yes See OpenAMP Demos
Works with FSBL RPU boot	No	Yes	Yes See https://www.wiki.xilinx.com/OpenAMP

Table 5: **Features** *(cont'd)*

	Linux Kernel RPMsg/ Remoteproc on APU + OpenAMP Library Used on RPU	OpenAMP Library Used on Linux Userspace + OpenAMP Library Used on RPU	Libmetal Library Used on Both APU and RPU
Data Transfer Overhead	Memory copy between user application and Linux kernel, and Linux kernel space to shared memory	Memory copy between user application and shared memory	

Other Considerations

- OpenAMP provides the source implementation on Remoteproc, VirtIO, and RPMsg for inter-processor communication. If you already have your communication solution or prefer a lighter solution, you can develop your own solution on top of libmetal library.
- Recently, new device-tree bindings and a device driver were introduced in the upstream Linux kernel. The current fork of the Linux kernel does not use this device driver. The existing remoteproc device driver and bindings in this kernel are scheduled for deprecation in future releases, with the upstream remoteproc device driver and bindings replacing them. You can find the upstream remoteproc device-tree bindings and device driver here:

  1. Upstream remoteproc dt bindings for AMD platforms: https://git.kernel.org/pub/scm/linux/kernel/git/remoteproc/linux.git/tree/Documentation/devicetree/bindings/remoteproc/xlnx,zynqmp-r5fss.yaml?h=rproc-next
  2. Upstream remoteproc device driver for AMD platforms: https://git.kernel.org/pub/scm/linux/kernel/git/remoteproc/linux.git/tree/drivers/remoteproc/xlnx_r5_remoteproc.c?h=rproc-next

  With this change you can expect different device-tree representation then what is currently mentioned in this document for Zynq UltraScale+ MPSoC and later platforms.

Known Limitations

The following are the known limitations in OpenAMP:

- Shared memory cannot be used as normal memory in Linux Userspace. It must be used as device memory, because libmetal in linux userspace uses UIO.
- The default IPIs defined for the APU are used by Linux for power management functions. OpenAMP uses one of the IPIs identified for use by the PL.
- The RPMsg buffer size is limited to 512 bytes, but 496 bytes are used for the payload.

- The behavior of allocated resources on program termination is dependent on the underlying runtime environment. In the AMD Baremetal environment, it is the responsibility of the programmer to do any cleanup, while on Linux, the operating system cleans up any allocated resources on program termination.

Linux RPMsg Buffer Size

The OpenAMP message size is limited by the buffer size defined in the `rpmsg` kernel module. For the Linux 4.19 kernel, this is currently defined as 512 bytes with 16 bytes for the message header and 496 bytes of payload.

***Note*:** Do not redefine the RPMsg buffer size.

Known Issues

After a Software Reboot, Remoteproc RPU Start/Stop Fails

This issue can only occur if resource table metadata is introduced in the firmware (for example, baremetal and freertos).

The sequence of events leading to the issue are as follows:

1. Boot Linux: After you boot Linux the system starts normally, and the `remoteproc` framework initializes.
2. Start RPU: After you start the RPU through the `remoteproc` driver, the RPU firmware runs and creates a valid resource table in DDR memory.
3. Stop RPU: After you stop the RPU using the `remoteproc` stop API, the RPU halts execution but the resource table remains in DDR.
4. Reboot System (Without Full DDR Power Drain): After you reboot the system without removing DDR power, the old RPU resource table still resides in DDR even though the RPU is powered down and no longer running firmware.
5. Boot Linux Again: After you reboot, Linux initializes and probes the `remoteproc` driver.
6. Linux detects cached resource table and attempts attach.

Temporary Workaround for 2025.2

If no `rpmsg` channel is created after the next start, do the following steps:

1. Manually stop the RPU through `sysfs`
2. Load the firmware

3. Restart the RPU

This is only a temporary workaround; the workaround is not needed after RPU state detection is added to the Linux Xilinx `remoteproc` driver.

The issue occurs because the driver finds the stale resource table still present in DDR and assumes the RPU is active. Linux performs an attach instead of a start. The attach operation fails because the RPU is powered down and not executing firmware.

OpenAMP Custom Subsystem Use Case: Start/Stop Failure and False Attach Occurrence

The sequence of events leading to the issue are as follows:

1. Boot to U-Boot
    - After you power on the system and boot to U-Boot
2. RPU starts running
    - The RPU firmware is loaded and begins execution
3. Linux boots
    - As Linux boots, it detects a valid resource table for the RPU and assumes the remote processor is already running
    - The Linux `remoteproc` driver performs an attach rather than a start
4. (Normal operation proceeds)
5. RPU subsystem powers down independently
    - The RPU shuts down outside of Linux control, for example, due to a power management event or a manual reset
    - The resource table remains in memory but is now stale
6. Linux reboots
7. Linux detects cached resource table and attempts attach

On reboot, Linux detects the stale resource table in memory and assumes the RPU is still active. The `remoteproc` driver attempts to attach and incorrectly succeeds by using the cached copy of the resource table, even though the RPU is powered off.

Chapter 5

Libmetal APIs

Libmetal API Functions

OpenAMP project now has separate documentation project that takes care of generating doxygen formatted documentation. Such documentation for Libmetal library can be found here: https://openamp.readthedocs.io/en/latest/doxygen/libmetal/index.html.

Chapter 6

OpenAMP APIs

OpenAMP library APIs

Latest OpenAMP library documentation can be found here: https://openamp.readthedocs.io/en/latest/doxygen/openamp/index.html.

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav, do the following:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **AMDDesignTools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs, do the following:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

1. *Versal Adaptive SoC Technical Reference Manual* (AM011)
2. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
3. *Zynq UltraScale+ Devices Register Reference* (UG1087)
4. *SDK Online Help* (UG782)
5. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
6. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
7. OpenAMP Wiki: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841718/OpenAMP
8. Xilinx libmetal source code: https://github.com/Xilinx/libmetal
9. Xilinx OpenAMP source code: https://github.com/Xilinx/open-amp
10. GitHub - Xilinx/meta-xilinx: Collection of Yocto Project layers to enable AMD Xilinx products
11. meta-xilinx/README.building.md at master · Xilinx/meta-xilinx

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/26/2025 Version 2025.2	
Build Libmetal Firmware and Linux Application with Yocto Tools	Changed the machine name for KRIA SOM
OpenAMP Quick Try	Updated the section.
Building and Booting OpenAMP Images	Created the section.
Running the Demos	Created the section.
Create and Build OpenAMP Applications in Vitis (Unified IDE)	Added a note to step 3.
Known Issues	Created new section.

Section	Revision Summary
05/29/2025 Version 2025.1	
Introduction to the New Upstream remoteproc Driver	Created new section.
Attach-detach Feature in the remoteproc Subsystem	Created new section.
AMD Libmetal AMP Demo	Changed text from "AMD Vitis™ to AMD Vitis Unified IDE™ and PetaLinux to Yocto Project tools."
Build Libmetal Baremetal Firmware with AMD Vitis Unified IDE	Updated the section.
Build Libmetal Firmware and Linux Application with Yocto Tools	Updated the section.
OpenAMP Quick Try	Updated the section.
Running the Example Applications	Removed bullets.
• Settings for the Device Tree Binary Source • Creating an Application Project for OpenAMP • Building Linux Application that uses RPMsg in Kernel Space • Building the Applications and the Linux Project • Booting the PetaLinux Project • Booting on QEMU • Booting on Hardware • Debugging Linux OpenAMP Application • Building Linux Applications Using OpenAMP RPMsg in Linux Userspace • Building RPU Firmware	Removed topics.

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright