

# Libmetal and OpenAMP User Guide

UG1186 (v2018.2) June 6, 2018



## Revision History

06/06/2018: Released with Vivado<sup>®</sup> Design Suite 2018.2 without changes from 2018.1.

Section	Revision Summary
<b>05/04/2018 Version 2018.1</b>	
General updates	Editorial updates only. No technical content updates.
<b>04/18/2018 Version 2018.1</b>	
<a href="#">Enable Linux Demo Application Using Libmetal with PetaLinux Tools</a>	Added a note update.

# Table of Contents

## Chapter 1: Overview

Introduction .....	5
--------------------	---

## Chapter 2: Libmetal

Overview .....	7
Access Devices with Libmetal .....	8
Xilinx Libmetal AMP Demo .....	12

## Chapter 3: OpenAMP

Overview .....	20
Components in OpenAMP .....	20
Connection between OpenAMP and Libmetal .....	22
How to Write a Simple OpenAMP Application .....	22
OpenAMP Demos .....	26

## Chapter 4: System Design Consideration

Supported Configuration .....	43
Other Consideration .....	44
Known Limitations .....	44

## Appendix A: Libmetal APIs

Libmetal API Functions .....	46
------------------------------	----

## Appendix B: OpenAMP APIs

Remoteproc APIs .....	66
Remoteproc API Functions .....	66
RPMmsg Development .....	68
RPMmsg API Functions .....	68

## Appendix C: Additional Resources and Legal Notices

Xilinx Resources .....	74
Solution Centers .....	74
Documentation Navigator and Design Hubs .....	74



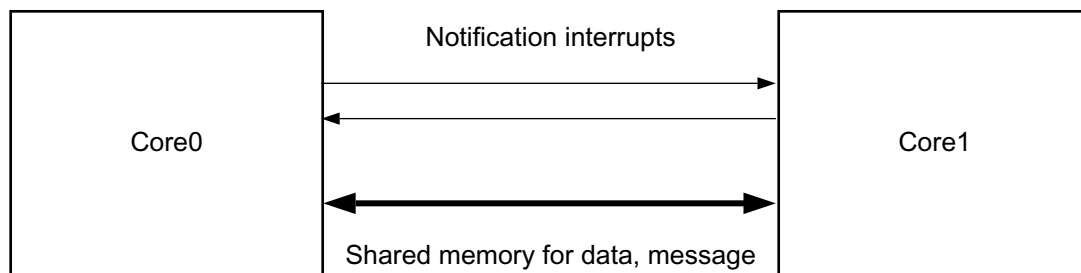
**Xilinx Documentation ..... 75**  
**Please Read: Important Legal Notices ..... 75**

## Overview

### Introduction

This user guide describes how to develop a methodology to enable communication between multiple processors on Xilinx® Zynq® and Zynq UltraScale+™ MPSoC platforms.

The basic development concept is based on the principles of Interrupts and Shared Memory, two foundational principles, that of interrupts and shared memory between the communicating elements:



X19939-101217

*Figure 1-1: Inter Processor Communication*

The libmetal library provides common user APIs, used to access devices, handle device interrupts, and request memory across different operating environments. You can use libmetal to build your own AMP solution. Xilinx uses the OpenAMP 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 Xilinx SDK are required in order to follow the instructions in this document to build applications.

- PetaLinux
- Xilinx SDK

## Prerequisites

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

- Linux, PetaLinux, and Xilinx SDK.
- How to boot a Xilinx board using JTAG boot.
- The remoteproc, RPMsg, and virtIO components used in Linux and bare-metal.

## 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
- FreeRTOS
- Bare-metal Environments

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



**Figure 2-1: Libmetal Architecture**

See the `libmetal` sources [Ref 5] for more details on the `libmetal` APIs.

## 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 may have different device abstractions. Following is a table to explain how libmetal manages devices differently:

**Table 2-1: Libmetal Devices**

Linux	Baremetal and FreeRTOS
1. Devices are described in a device tree.	1. Because there is no device tree abstraction, devices must be defined statically before attempting to open them.
2. "platform" bus definition is in Linux kernel. It is used by Linux to present memory mapped devices.	2. No standard for bus abstraction. Libmetal library defines generic bus structure to manage devices.

### Start Libmetal Environment, Add and Open the Devices

1. Initialize libmetal environment with call to `metal_init()`.

```
struct metal_init_params metal_param = METAL_INIT_DEFAULTS;  
metal_init(&metal_param);
```

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. If don't want to use the default ones, you can define yours. */
        }
    },
    .node = {NULL}, /* will be set by libmetal later. used to keep track of the devices list */
    .irq_num = 1, /* number of interrupts of this device */
    .irq_info = (void *)65, /* interrupt information, here is the irq vector id */
};
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 = <0x0 0xff340000 0x0 0x1000>;
        interrupt-parent = <&gic>;
        interrupts = <0 29 4>;
    };
};
```

### 3. 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 IPI (inter-processor interconnect) and Shared Memory including libmetal as an example.

**Note:** The following section refers to the IPI elements of the ZU+ MPSoC hardware as described in Chapter 13 of the *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#)).

## 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 IPI and Shared Memory with Libmetal

### *Zynq UltraScale+ MPSoC IPI Hardware*

The IPI (Inter Processor Interrupt) interrupt can be used for notification of 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.

For users of libmetal, 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 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 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);

```

## Xilinx Libmetal AMP Demo

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

Xilinx SDK and PetaLinux tools include a libmetal demo to demonstrate how to use the libmetal library to build simple interprocessor communication between APU and RPU on a Zynq UltraScale+ MPSoC platform.

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

- DDR
- IPI (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 Xilinx® SDK and PetaLinux tools.

The Libmetal AMP Demonstration includes:

- Shared memory.
- Shared memory with atomics.
- IPI with shared memory.
- IPI latency measurement.
- Shared memory latency measurement.
- Shared memory throughput measurement.

## Build Libmetal Bare-Metal Firmware with Xilinx SDK

1. From the Xilinx SDK window, create the application project by selecting **File > New > Application Projects**.
  - a. Specify the BSP OS platform:
    - **standalone** for a bare-metal application.
  - b. Specify the hardware platform.
  - c. Select the processor:
    - Cortex™-R5 (RPU) is supported. Select **psu\_cortexr5\_0** or **psu\_cortexr5\_1**.
  - d. Select one of the following BSP options:
    - Use **Existing** if you had previously created an application with a BSP and want to reuse the same BSP. In this case, you need to make sure that the `libmetal` library is selected in the BSP.
    - Use **Create New BSP** to create a new BSP. If you make this selection, the `libmetal` library is automatically included.
  - e. Click **Next** to select an available template. (Do not click Finish.)
  - f. From the available templates, select **libmetal AMP Demo**.
  - g. Click **Finish**.
2. Before you build the application, review the source code of the generated application from the Xilinx SDK project explorer. The key source files of the `libmetal` demonstration application are as follows:
  - `sys_init.c`: System initialization, such as GIC initialization, and metal device definition for IPI device and shared memory.
 

**Note:** If you have selected `psu_cortex_r5_1`, change the following: In `sys_init.c`, change `IPI_BASE_ADDR` to `0xFF320000` and `IPI_IRQ_VECT_ID` to `66`.
  - `libmetal_amp_demo.c`: Demo application that illustrates how to use IPI and shared memory with `libmetal` for inter-processor communication.

- `common.h`: common file with shared resources and functions needed for multiple demos in Xilinx Libmetal AMP Demo as well as function headers for each demo.
  - `ipi_latency_demo.c`: Demo application that measures latency between APU and RPU.
  - `ipi_shmem_demo.c`: Demonstrates how to access shared memory and IPI.
  - `shmem_atomic_demo.c`: Demonstrates how to access shared memory with atomics.
  - `shmem_demo.c`: Demonstrates use of shared memory between APU and RPU.
  - `shmem_latency_demo.c`: Demo application that measures shared memory latency between APU and RPU.
  - `shmem_throughput_demo.c`: Demo application that measures shared memory throughput between APU and RPU.
3. To build the application project, right-click the created project and select **Build project**. The generated ELF is in the `<RPU_app_proj>/Debug/` directory.

## Enable Linux Demo Application Using Libmetal with PetaLinux Tools

Before using PetaLinux tools, follow these preparatory steps:

1. Create the PetaLinux master project in a suitable directory without any spaces. In this guide it is named `<plnx-proj-root>`:

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

**Note:** The petalinux bsp's can be found at <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html>.

2. Navigate to the directory:

```
$ cd <plnx-proj-root>
```

3. Enable the required `rootfs` packages and applications:

```
$ petalinux-config -c rootfs
```

4. Ensure `libmetal` and `sysfs` packages are enabled:

```
Filesystem Packages--->
  misc --->
    sysfsutils --->
      [*] libsysfs
  Libs --->
    libmetal--->
      [*] libmetal
```

5. Ensure the `libmetal` demo application is enabled:

```
Filesystem Packages --->
  libs --->
    libmetal-->
      [*] libmetal-demos
```

## 6. Setting Device Tree for the Libmetal Linux Application Demonstration.

The device tree changes need to be added to system-user.dtsi.

Petalinux system-user.dtsi path:

```
<plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/
files/system-user.dtsi
```

**Note:** Reserved memory node is for shared memory and firmware. This can be moved if you wish to load firmware elsewhere. You need to add device tree nodes manually to the system-user.dtsi file.

```
{
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;
        rproc_0_reserved: rproc@3ed000000 {
            no-map;
            reg = <0x0 0x3ed00000 0x0 0x2000000>;
        };
    };

    amba {
        /* Shared memory */
        shm0: shm@0 {
            compatible = "shm_uio";
            reg = <0x0 0x3ed80000 0x0 0x1000000>;
        };

        /* IPI device */
        ipi_amp: ipi@ff340000 {
            compatible = "ipi_uio";
            reg = <0x0 0xff340000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
        };
    };
};&ttc0 {
    compatible = "ttc0";
    status = "okay";
};
```

If you wish to load firmware via remoteproc, you can also define a remoteproc device node in the device tree.

A sample remoteproc device node using memory in both TCM and DDR could look like the following:

The shm0 device tree node is used by the Libmetal application for shared memory starting at the address 0x3ed80000.

**Note:** Firmware memory needs to correspond to the firmware's linker script. An example linker script for this application can be found at:

[https://github.com/OpenAMP/libmetal/blob/master/examples/system/generic/zynqmp\\_r5/zynqmp\\_amp\\_demo/lscrip.ld](https://github.com/OpenAMP/libmetal/blob/master/examples/system/generic/zynqmp_r5/zynqmp_amp_demo/lscrip.ld).

```
/{
power-domains {
    pd_r5_0: pd_r5_0 {
        #power-domain-cells = <0x0>;
        pd-id = <0x7>;
    };
    pd_tcm_0_a: pd_tcm_0_a {
        #power-domain-cells = <0x0>;
        pd-id = <0xf>;
    };
    pd_tcm_0_b: pd_tcm_0_b {
        #power-domain-cells = <0x0>;
        pd-id = <0x10>;
    };
};

amba {

    /* firmware memory nodes */
    r5_0_tcm_a: tcm@ffe00000 {
        compatible = "mmio-sram";
        reg = <0x0 0xFFE00000 0x0 0x10000>;
        pd-handle = <&pd_tcm_0_a>;
    };
    r5_0_tcm_b: tcm@ffe20000 {
        compatible = "mmio-sram";
        reg = <0x0 0xFFE20000 0x0 0x10000>;
        pd-handle = <&pd_tcm_0_b>;
    };

    elf_ddr_0: ddr@3ed00000 {
        compatible = "mmio-sram";
        reg = <0x0 0x3ed00000 0x0 0x100000>;
    };
    test_r5_0: zynqmp_r5_rproc@0 {
        compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
        reg = <0x0 0xff9a0100 0x0 0x100>,
            <0x0 0xff9a0000 0x0 0x100>;
        reg-names = "rpu_base", "rpu_glbl_base";
        dma-ranges;
        core_conf = "split0";
        srams = <&r5_0_tcm_a &r5_0_tcm_b &elf_ddr_0>;
        pd-handle = <&pd_r5_0>;
    };
};
};
```

The source code of the libmetal example on the Linux side can be found on the following web site:

[https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp\\_amp\\_demo](https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp_amp_demo)



- common.h
- ipi\_latency\_demo.c
- ipi\_shmem\_demo.c
- libmetal\_amp\_demo.c
- shmem\_atomic\_demo.c
- shmem\_demo.c
- shmem\_latency\_demo.c
- shmem\_throughput\_demo.c
- sys\_init.c
- sys\_init.h

## Build Libmetal Linux Demo in Xilinx SDK

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 Xilinx SDK (XSDK).

Following are the steps in Xilinx SDK to generate the application.

1. Run XSDK.
2. Select **create a new Application project**.

```
OS: Linux
Processor: psu_cortexa53
Linux sysroot: the sysroot you built from your PetaLinux project:
    "<plnx-proj-root>/build/tmp/sysroots/plnx_aarch64"
Click Next
```

3. Select **Linux Hello World** and then click **Finish**.
4. Right-click **project** and select **properties**.

```
C/C++ Build • Settings
  Tool Setting Tab Libraries
    Libraries (-l) add "metal"
  Miscellaneous
    Add "sysroot" setting to "Linker Flags":
    "--sysroot=<plnx-proj-root>/build/tmp/sysroots/plnx_aarch64"
    click OK
```

5. Copy files located at  
([https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp\\_amp\\_demo](https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp_amp_demo)) to the application's src directory.
  - **common.h**
  - **ipi\_latency\_demo.c**

- **ipi\_shmem\_demo.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 IPI mask value in common.h to 0x200, which is the default RPU1 IPI mask.

6. Install the Linux application executable built from XSDK and firmware into the `rootfs` built with PetaLinux tools using a Yocto Recipe created by:

```
$ petalinux-create -t apps --template install --name libmetal-linux-app -install
--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"

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> "
```

## 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.

Note that 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 XSDK 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 *****
```

# 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:

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

---

## Components in OpenAMP

RPMsg, VirtIO and remoteproc are implemented in upstream Linux kernel. The OpenAMP library provides the implementation for these components for the following environments: baremetal, FreeRTOS, 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 a 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 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 Xilinx® Zynq® and Zynq UltraScale+™ MPSoC platforms:

1. Linux kernel master and RPU OpenAMP slave.

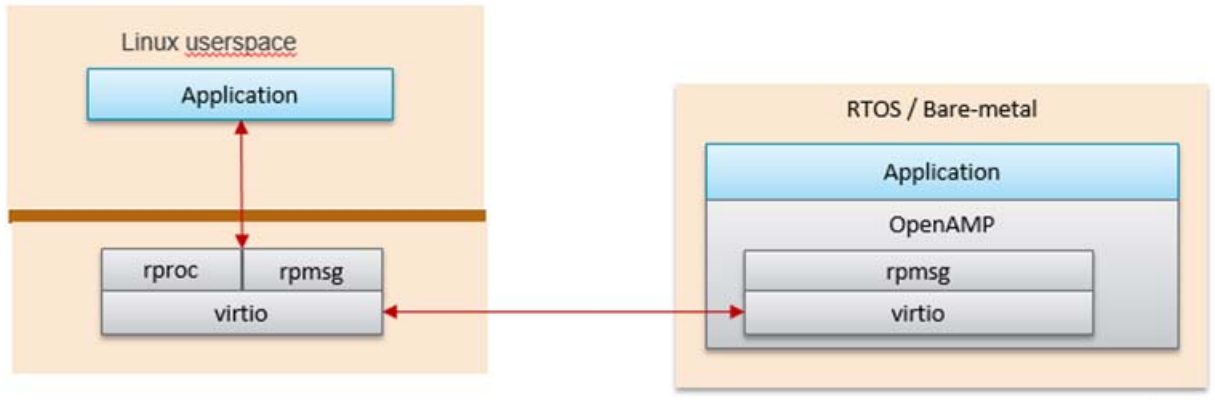


Figure 3-1: RPMsg Implementation in Kernel Space

Linux kernel space provides RPMsg and Remoteproc, but the RPU application requires Linux to load it in order 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.

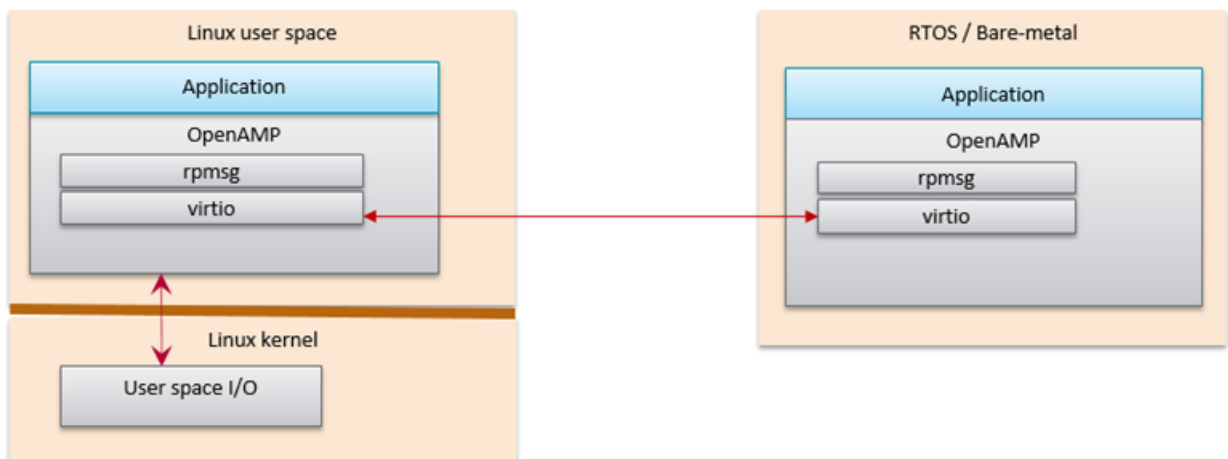


Figure 3-2: OpenAMP RPMsg Implementation in Linux Userspace

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

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 IPI and shared memory. OpenAMP leverages standards for shared memory management, lifecycle management and communication. The following diagram shows the relationship between libmetal and OpenAMP:

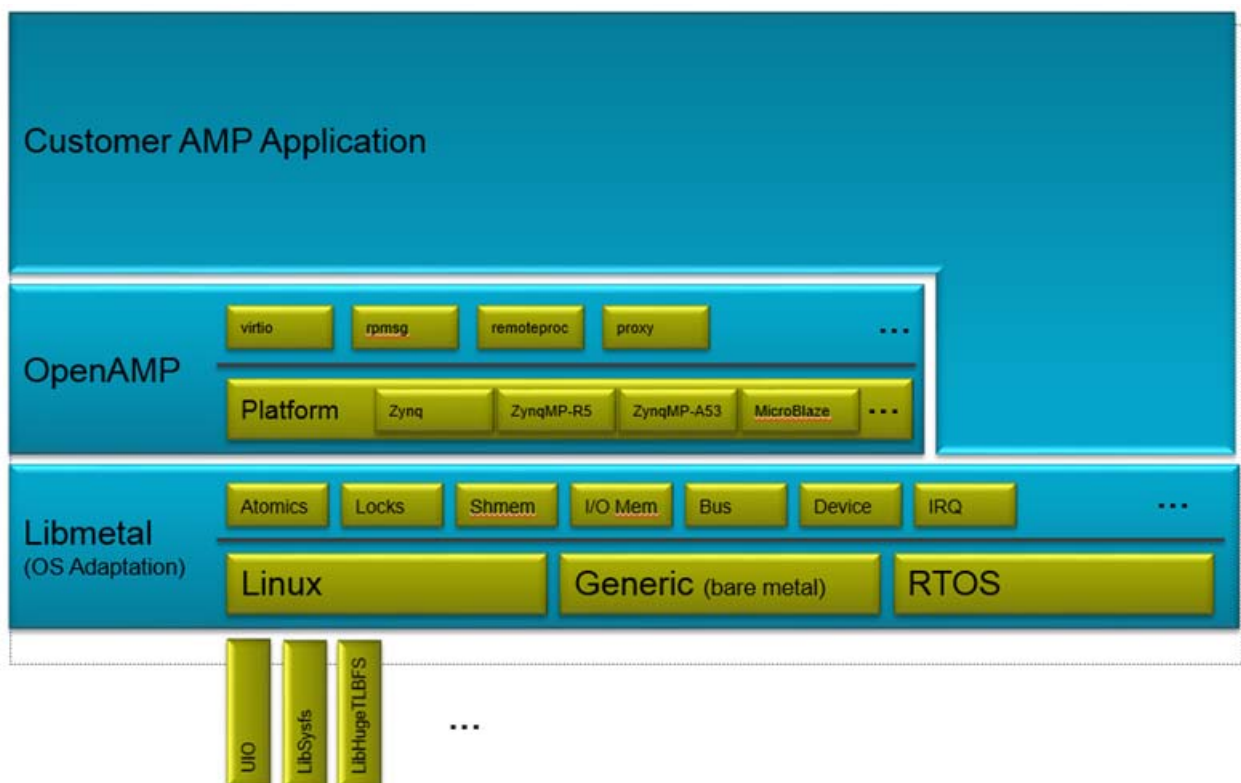


Figure 3-3: Libmetal and OpenAMP Connection

## How to Write a Simple OpenAMP Application

To write an OpenAMP application there are a few necessary pieces as follows:

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 `remote_proc`.

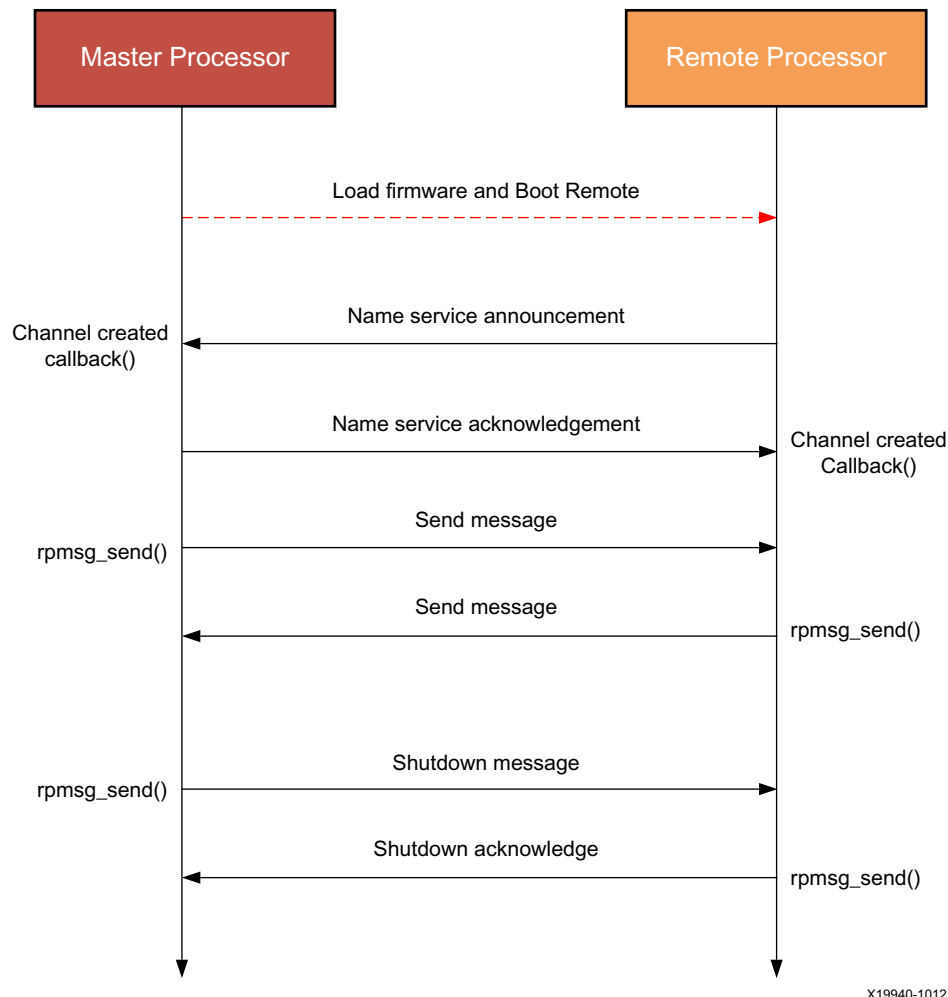
2. Create a `hil_proc` instance.

`hil_proc` represents a remote processor and encapsulates shared memory and notification information required for IPC. `hil_proc` is required to initialize the RPMMsg framework. To create a `hil_proc` instance, provide the `hil_proc` platform operations, the remote CPU ID (for example, the CPU ID of the remote processor in its cluster), and private data.

3. Define RPMMsg callback functions for channel creation, deletion and receive.
4. Call `remoteproc_resource_init()` to configure the RPMMsg framework.

`remoteproc_resource_init()` initializes the RPMMsg framework in an OpenAMP application using these the firmware resource table and `hil_proc` structures in addition to the RPMMsg callback functions as arguments.

5. Use `rpmsg_send()` to send message across to the remote processor
6. After initializing the framework with `remote_proc_resource_init()`, the flow of an OpenAMP application consists of the RPMMsg channel communicating between the master and remote processor via the RPMMsg `send()` and I/O callback functions. The following diagram displays the flow:



X19940-101217

Figure 3-4: Flow Diagram

Following is a sample OpenAMP set up and flow with a resource table, hil\_proc 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 set the shared memory address range. It is required if
you want to specified shared memory statically. If you want the host to allocate
shared memory, you don't need to define this resource entry. Please note that dynamic
shared memory allocation is only supported when the Linux host uses RPMsg kernel
space implementation. */
{
RSC_RPROC_MEM, 0x3ed40000, 0x3ed40000, 0x100000, 0},
}
}

```



```

/* virtio device header */
{
    RSC_VDEV, VIRTIO_ID_RPMMSG_, 0, RPMMSG_IPU_C0_FEATURES, 0, 0, 0,
    NUM_VRINGS, {0, 0},
}
};

struct hil_proc * hil_proc_instance = hil_create_proc(
/* Processor operations. Used to define the notification operation. and
 * remote processor management. For the current release, we only use the
 * notification definition feature.
 */
    &zynqmp_r5_a53_proc_ops,
    APU_CPU_ID, /* CPU ID */
/* Private data for user-specific operations. We set this to NULL
 * because it is unused.
 */
    NULL
);

static struct rpmsg_channel *my_channel;
static struct rpmsg_endpoint *my_endpoint; /* local side of RPMsg channel */

/* Used to receive data */
static void rpmsg_callback(struct rpmsg_channel *channel, void *data, int len,
                          void *priv, u32 src)
{
    // User define how to receive RPMsg
}

static void channel_created_callback(struct rpmsg_channel * channel)
{
    // create endpoint for communication
    my_endpoint = rpmsg_create_ept(channel, rpmsg_read_cb, RPMMSG_NULL,
                                   RPMMSG_ADDR_ANY);
}

static void channel_deleted_callback(struct rpmsg_channel *channel)
{
    // destroy endpoint
    rpmsg_destroy_ept(rp_ept);
}

int main(){
/* Initialize RPMsg framework */
    remoteproc_resource_init(
        &table, /* resource table */
        hil_proc_instance, /* hil_proc instance */
        channel_created_callback, /* rpmsg channel created callback */
        channel_deleted_callback, /* rpmsg channel deleted callback */
        rpmsg_callback, /* rpmsg channel IO callback */

        &remoteproc_ptr, /* pointer to new remoteproc instance */
        1 /* denote that this configuration is for RPMsg master */
    );

    void * data; /* data to send across channel */

```

```
int len; /* length of data sent across channel */
/* set up data and len here */
/* send data across to remote processor */
rpsmsg_send( my_channel, data, len);
return 0;
}
```

## OpenAMP Demos

Following are descriptions for each of the OpenAMP demonstration applications.

### Echo Test in Linux Master and Bare-Metal 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 bare-metal firmware using `remoteproc`.
- The Linux master then 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 Bare-Metal or FreeRTOS Remotes

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

The Linux master boots the bare-metal remote firmware using `remoteproc`. It then transmits two randomly-generated matrices using `RPMsg`.

The bare-metal firmware multiplies the two matrices and transmits the result back to the master using `RPMsg`.

### Proxy Application for Linux Masters and Bare-Metal 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.

## Petalinux Images Quick Try

Use the following basic steps to boot Linux and run an OpenAMP application using pre-built images. The following steps apply to the ZCU102 board.

The echo-test application sends packets from Linux running on quad-core Cortex-A53 to a single Cortex-R5 running FreeRTOS, which sends them back.

1. Extract files `BOOT.BIN`, `image.ub`, and `openamp.dtb` files from a pre-built PetaLinux BSP tarball to an SD card. Note that the OpenAMP related device nodes are not in the default `system.dtb`, but are included in the prebuilt `openamp.dtb`.

```
host shell$ tar xvf xilinx-zcu102-v2017.3-final.bsp --strip-components=4 --wildcards
*/BOOT.BIN */image.ub */openamp.dtb
host shell$ cp BOOT.BIN image.ub openamp.dtb <your sd card>
```

**Note:** Alternatively, if you already created a PetaLinux project with a provided BSP for your board, you can find pre-built images in the `<your project>/pre-built/linux/images/` directory.

2. Go to u-boot prompt and boot Linux from the SD card:

```
...
Hit any key to stop autoboot: 0
ZynqMP> mmcinfo && fatload mmc 0 ${netstart} ${kernel_img} && fatload mmc 0
0x14000000 openamp.dtb
Device: sdhci@ff170000
...
reading image.ub
31514140 bytes read in 2063 ms (14.6 MiB/s)
reading openamp.dtb
38320 bytes read in 18 ms (2 MiB/s)
ZynqMP> bootm $netstart - $netstart 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`, then type `petalinux-boot --jtag --prebuilt 3`.

3. At the Linux login prompt, type `root` for user and `root` for password, and then run the echo-test demo.

```
plnx_aarch64 login: root
Password:
root@plnx_aarch64:~# echo image_echo_test >
/sys/class/remoteproc/remoteproc0/firmware
root@plnx_aarch64:~# echo start > /sys/class/remoteproc/remoteproc0/state
[ 177.375451] remoteproc remoteproc0: powering up ff9a0100.zynqmp_r5_rproc
[ 177.384705] remoteproc remoteproc0: Booting fw image image_echo_test, size 644144
[ 177.396832] remoteproc remoteproc0: registered virtio0 (type 7)
[ 177.399108] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 177.412370] zynqmp_r5_remoteproc ff9a0100.zynqmp_r5_rproc: RPU boot from TCM.
[ 17Starting application...
Try to init remoteproc resource
```

```
Init remoteproc resource succeeded
Waiting for events...
7.422089] remoteproc remoteproc0: remote processor ff9a0100.zynqmp_r5_rproc is now
up
[ 177.442121] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel
addr 0x1
root@plnx_aarch64:~# modprobe rpmsg_user_dev_driver
[ 188.089835] rpmsg_user_dev_driver virtio0:rpmsg-openamp-demo-channel:
rpmsg_user_dev_rpmsg_drv_probe
[ 188.101250] rpmsg_user_dev_driver virtio0:rpmsg-openamp-demo-channel: new
channel: 0x400 -> 0x1!
root@plnx_aarch64:~# echo_test
Echo test start
Open rpmsg dev!
[ 190.364739] rpmsg_user_dev_driver virtio0:rpmsg-openamp-demo-channel: Sent
init_msg to target 0x1.
*****

Echo Test Round 0

*****
```

**Note:** 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](#))).

## Building OpenAMP application for RPU Firmware

### Introduction

The Xilinx® software development kit (Xilinx SDK) contains templates to aid in the development of OpenAMP bare-metal/FreeRTOS remote applications. The following sections describe how to create OpenAMP applications with Xilinx SDK and PetaLinux tools.

- Use Xilinx SDK to create the bare-metal or FreeRTOS remote applications.

## Building Remote Applications in Xilinx SDK

You can build remote applications using Xilinx SDK by using the following procedures. The PetaLinux BSP already include pre-built firmware for a remote processor (Zynq® Cortex™-A9 #1 and Zynq UltraScale+™ MPSoC Cortex-R5 #0);The following steps are necessary only if you plan to re-build the demo applications running on the remote processor.

### Creating an Application Project for OpenAMP

1. From the Xilinx SDK window, create the application project by selecting **File > New > Application Projects**.
  - a. Specify the BSP OS platform:
    - standalone for a bare-metal application.

- `freertos<version>_xilinx` for a FreeRTOS application.
- b. Specify the hardware platform.
- c. Select the processor:
  - For the Zynq UltraScale+ MPSoC device (ZynqMP), Cortex-R5 (RPU) is supported.
  - Select `psu_cortexr5_0` or `psu_cortexr5_1`.
  - For the Zynq-7000 SoC device (zynq), only Cortex-A9 is supported.
  - Select `ps7_cortexa9_1`.
- d. Select one of the following:
  - **Use Existing** if you had previously created an application with a BSP and want to re-use the same BSP.
  - **Create New BSP** to create a new BSP.




---

**IMPORTANT:** If you select **Create New BSP**, the `openamp` library is automatically included, but the compiler flags must be set as indicated in the upcoming steps.

---

- e. Click **Next** to select an available template (do *not* click **Finish**).
- 2. Select one of the three application templates available for OpenAMP remote bare-metal from the available templates:
  - OpenAMP echo-test
  - OpenAMP matrix multiplication Demo
  - OpenAMP RPC Demo
- 3. Click **Finish**.
- 4. In the Xilinx SDK project explorer, right-click the BSP and select **Board Support Package Settings**.
- 5. Navigate to the **BSP Settings > Overview > OpenAMP**.
- 6. Set the **WITH\_PROXY** parameter as follows:
  - For the OpenAMP RPC demonstration, set the parameter to `true` (default).
  - For other demo applications, set the parameter to `false`.

**Note:** Having `WITH_PROXY=true` is needed for OpenAMP to redirect `_open()`, `_close()`, `_read()`, and `_write()` to the master processor and instruct the makefile to compile extra code that is not needed or desired for other applications.
- 7. Navigate to the BSP settings drivers: **Settings > Overview > Drivers > <selected\_processor>**.

For the Zynq-7000 SoC device (zynq) only:

- To disable initialization of shared resources when the master processor is handling shared resources initialization, add:

```
-DUSE_AMP=1
```

In the following examples, `ps7_cortexa9_0` runs Linux while the OpenAMP slave runs on `ps7_cortexa9_1`, therefore you need to set this parameter.

8. Add any necessary parameters to the `extra_compiler_flags`.
9. Click **OK**.

## OpenAMP Xilinx SDK Key Source Files

The following key source files are available in the Xilinx SDK application

- **Platform Info** (`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 IPI agent used for interprocessor communication.
  - `#define IPI_BASE_ADDR`: The base address of IPI agent used for interprocessor communication.
  - `#define RPMSG_CHAN_NAME`: The name used to identify a communication channel between two processors.
  - `#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.  
[https://www.xilinx.com/html\\_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html#\\_overview.html](https://www.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html#_overview.html)
- **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`.

For the `RSC_RPROC_MEM` resource, the Linux kernel `remoteproc` allocates shared memory for `vrings` and `RPMsg` buffers from the memory specified in this resource. If you do not specify this resource in the resource table, the Linux side allocates the memory from its system memory. If you specify it in the resource table, it must be inside the range defined by the DTS reserved-memory section for `rproc`. It should not overlap its address with the memory nodes in the device tree, which are used to load the firmware.

- **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 XSDK, the specific application is located (echo\_test.c/matrix\_multiply.c/rpc\_demo.c)

## Building Linux Application that uses RPMMsg in kernel space

### Setting up PetaLinux with OpenAMP

PetaLinux requires the following preparation before use:

1. Create the PetaLinux master project in a suitable directory without any spaces. In this guide it is named <plnx-proj-root>:

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

2. Navigate to the <plnx-proj-root> directory:

```
$ cd <plnx-proj-root>
```

3. Include a remote application in the PetaLinux project.

This step is needed if you are not using one of the pre-built remote firmware already included with the PetaLinux BSP. After you have developed and built a remote application (for example, with Xilinx SDK) it must be included in the PetaLinux project so that it is available from the Linux filesystem for remoteproc.

- a. Create a PetaLinux application inside the components/apps/<app\_name> directory, using the following command:

```
$ petalinux-create -t apps --template install -n <app_name> --enable
```

- b. Copy the firmware (the .elf file) built with Xilinx SDK for the remote processor into this directory:

```
project-spec/meta-user/recipes-apps/<app-name>/files/
```

- c. Modify the project-spec/meta-user/recipes-apps/<app\_name>/<app\_name>.bb to install the remote processor firmware in the RootFS.

For example:

```
SUMMARY = "Simple test application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://<myfirmware>"
S = "${WORKDIR}"
INSANE_SKIP_${PN} = "arch"

do_install() {
    install -d ${D}/lib/firmware
    install -m 0644 ${S}/<myfirmware> ${D}/lib/firmware/<myfirmware>
}
```

```
FILES_${PN} = "/lib/firmware/<myfirmware>"
```

#### 4. For all devices, configure the kernel options to work with OpenAMP:

##### a. Start the PetaLinux Kernel configuration tool:

```
petalinux-config -c kernel
```

##### b. Enable loadable module support:

```
[*] Enable loadable module support --->
```

##### c. Enable user space firmware loading support:

```
Device Drivers --->
  Generic Driver Options --->
    <*> Userspace firmware loading support
```

##### d. Enable the remoteproc driver support: Note that the commands differ, based on which Zynq device you are using:

```
Device Drivers --->
  Remoteproc drivers --->
    # for R5:
    <M> ZynqMP_r5 remoteproc support
    # for Zynq A9
    <M> Support ZYNQ remoteproc
```

#### 5. Enable all of the modules and applications in the RootFS:




---

**IMPORTANT:** *These options are only available in the PetaLinux reference BSP. The applications in this procedure are examples you can use.*

---

##### a. Open the RootFS configuration menu:

```
petalinux-config -c rootfs
```

##### b. Ensure the OpenAMP applications and rpmsg modules are enabled:

```
Filesystem Packages --->
  misc --->
    packagegroup-petalinux-openamp --->
      [*] packagegroup-petalinux-openamp
```

**Note:** packagegroup-petalinux-openamp enables many openamp related sub-components. If you need more fine-grained control, do not set this packagegroup. Instead, enable the following individual components as needed:

```
rpmsg-echo-test, rpmsg-mat-mul, rpmsg-proxy-app, kernel-module-rpmsg-proxy,
kernel-module-rpmsg-user
```

Links to each of the packages' source code for the above components can be found in the following:

- rpmsg-echo-test:  
<https://gitenterprise.xilinx.com/Yocto/meta-openamp/tree/rel-v2018.1/recipes-openamp/rpmsg-examples/rpmsg-echo-test>



- rpmsg-mat-mul:  
<https://gitenterprise.xilinx.com/Yocto/meta-openamp/tree/rel-v2018.1/recipes-openamp/rpmsg-examples/rpmsg-mat-mul>
  - rpmsg-proxy-app:  
<https://gitenterprise.xilinx.com/Yocto/meta-openamp/tree/rel-v2018.1/recipes-openamp/rpmsg-examples/rpmsg-proxy-app>
  - rpmsg-proxy-module:  
<https://gitenterprise.xilinx.com/Yocto/meta-openamp/tree/rel-v2018.1/recipes-kernel/rpmsg-proxy-module>
  - rpmsg-user-module:  
<https://gitenterprise.xilinx.com/Yocto/meta-openamp/tree/rel-v2018.1/recipes-kernel/rpmsg-proxy-module>
- c. If needed, enable inclusion of default remote processor firmware images:

```
Filesystem Packages --->
misc --->
  openamp-fw-echo-testd --->
    [*] openamp-fw-echo-testd
  openamp-fw-mat-muld --->
    [*] openamp-fw-mat-muld
  openamp-fw-rpc-demo --->
    [*] openamp-fw-rpc-demo
```

**Note:** This includes the same remote processor firmwares provided by pre-built images as found in the rootfs /lib/firmware directory. It is not needed if you build new images with the Xilinx SDK.

## Settings for the Device Tree Binary Source

The PetaLinux reference BSP includes a device tree binary (DTB) for OpenAMP located at:

```
pre-built/linux/images/openamp.dtb
```

The device tree setting for the shared memory and the kernel remoteproc is demonstrated in:

```
project-spec/meta-user/recipes-bsp/device-tree/files/openamp-overlay.dtsi
```

The openamp.dtb and openamp-overlay.dtsi files are provided for reference only. You need to edit the system-user.dtsi file to include the content from openamp-overlay.dtsi for your project.

The overlay contains nodes that OpenAMP requires in the device tree.

- The device tree example is for ZynqMP:

```
/ {
  reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
```

```

        ranges;
/* Reserved DDR memory for RPU firmware and shared memory between APU and RPU */
rproc_0_reserved: rproc@3ed00000 {
    no-map;
    reg = <0x0 0x3ed00000 0x0 0x1000000>;
};

power-domains {
    pd_r5_0: pd_r5_0 {
        #power-domain-cells = <0x0>;
        pd-id = <0x7>;
    };
    pd_tcm_0_a: pd_tcm_0_a {
        #power-domain-cells = <0x0>;
        pd-id = <0xf>;
    };
    pd_tcm_0_b: pd_tcm_0_b {
        #power-domain-cells = <0x0>;
        pd-id = <0x10>;
    };
};

amba {
    r5_0_tcm_a: tcm@ffe00000 {
        compatible = "mmio-sram";
        reg = <0x0 0xFFE00000 0x0 0x10000>;
        pd-handle = <&pd_tcm_0_a>;
    };
    r5_0_tcm_b: tcm@ffe20000 {
        compatible = "mmio-sram";
        reg = <0x0 0xFFE20000 0x0 0x10000>;
        pd-handle = <&pd_tcm_0_b>;
    };

    elf_dds_0: ddr@3ed00000 {
        compatible = "mmio-sram";
        reg = <0x0 0x3ed00000 0x0 0x40000>;
    };

test_r5_0: zynqmp_r5_rproc@0 {
    compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
    reg = <0x0 0xff9a0100 0x0 0x100>,
        <0x0 0xff340000 0x0 0x100>,
        <0x0 0xff9a0000 0x0 0x100>;
    reg-names = "rpu_base", "ipi", "rpu_glbl_base";
    dma-ranges;
    core_conf = "split0";
    srams = <&r5_0_tcm_a &r5_0_tcm_b &elf_dds_0>;
    pd-handle = <&pd_r5_0>;
    interrupt-parent = <&gic>;
    interrupts = <0 29 4>;
};
};
};

```

**Note:** OpenAMP running on Linux does not support use of the default IPI. IPI configuration for OpenAMP running on Linux is configured in the device tree. IPI informatoin can be found in the IPI

module of UG1087:

[https://www.xilinx.com/html\\_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html](https://www.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html)

In the above device tree demo, the OpenAMP in APU uses the PL0 IPI instead of the default APU IPI for inter-processor notification because the default APU IPI has been dedicated to the communication with PMU FW.

For ZynqMP, you can configure how the Cortex-R5 is operating by setting the `core_conf` parameter. The current settings works with the demo applications referenced in this document. [Appendix A, Libmetal APIs](#) gives a more detailed explanation of those parameters.

- For Zynq\_A9:

```
/ {
    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        rproc_0_reserved: rproc@3e000000 {
            no-map;
            reg = <0x3e000000 0x01000000>;
        };
    };

    amba {
        elf_dds_0: ddr@0 {
            compatible = "mmio-sram";
            reg = <0x3e000000 0x400000>;
        };
    };

    remoteproc0: remoteproc@0 {
        compatible = "xlnx,zynq_remoteproc";
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        srams = <&elf_dds_0>;
    };
};
```

## Building the Applications and the Linux Project

To build the applications and Linux project:

1. Ensure that you are in the PetaLinux project root directory:

```
cd <plnx_proj>
```

2. Build PetaLinux: `petalinux-build`



**TIP:** If you encounter any issues append `-v` to `petalinux-build` to see the respective textual output.

If the build is successful, the images are in the `images/linux` folder:

```
<plnx_proj>/images/linux
```

## Booting the PetaLinux Project

You can boot the PetaLinux project from QEMU or hardware.

### Booting on QEMU

After a successful build, run the PetaLinux project on QEMU:

1. Navigate to the PetaLinux directory: `cd <plnx_proj>`
2. Run PetaLinux boot: `petalinux-boot --qemu --kernel`

**Note:** Booting OpenAMP on QEMU is only valid for Zynq architecture.

### Booting on Hardware

After a successful build, you can run the PetaLinux project on hardware. Follow these procedures to boot OpenAMP on a board.

#### Setting Up the Board

1. Connect the board to your computer, and ensure that it is powered on.
2. If the board is connected to a remote system, start the `hw_server` on the remote system.
3. Open a console terminal and connect it to UART on the board.

#### Downloading the Images

1. Navigate to the PetaLinux directory:  
`cd <plnx_proj>`
2. Run the PetaLinux boot:
  - Using a remote system:  
`petalinux-boot --jtag --kernel --hw_server-url <remote_system>`
  - Using a local system:  
`petalinux-boot --jtag --kernel -bitstream <bitstream>`




---

**TIP:** If you encounter any issues append `-v` to the above commands to see the textual output.

---

## 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](#)

**Note:** Some important things to note are:

- After booting the Linux Kernel the remoteproc driver is already loaded. If not, check it has been enabled in the kernel config and check your device tree.
- If you have unloaded the remoteproc driver, you can load it as follows:
  - For the Zynq UltraScale+ MPSoC device:
 

```
modprobe zynqmp_r5_remoteproc
```
  - For the Zynq 7000 SoC device:
 

```
modprobe zynq_remoteproc
```

## 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
modprobe rpmsg_user_dev_driver
```

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:

```
modprobe -r rpmsg_user_dev_driver
echo stop > /sys/class/remoteproc/remoteproc0/state
```

If you want to simply reload and run the RPU firmware, you can keep `rpmsg_user_dev_driver` LKM loaded and simply re-issue a `start`.

## 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
modprobe rpmsg_user_dev_driver
```

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:

```
modprobe -r rpmsg_user_dev_driver
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 then prompts you to **re-run** the test.
6. After you exit the application, the module unloads automatically.

## Building Linux Applications Using OpenAMP RPMsg in Linux Userspace

### *Build Linux Userspace RPMsg Demo Applications Using PetaLinux Tools*

Before using PetaLinux tools, follow these preparatory steps:

1. Create the PetaLinux master project in a suitable directory without any spaces. In this guide it is named <plnx\_proj>:

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

2. Navigate to the directory:

```
$ cd <plnx_proj>
```

### 3. Start the rootfs configuration utility:

```
$ petalinux-config -c rootfs
```

### 4. Enable the required rootfs packages for this demo:

```
Filesystem Packages --->
  misc --->
    packagegroup-petalinux-openamp --->
      [*] packagegroup-petalinux-openamp
```

**Note:** packagegroup-petalinux-openamp enables many openamp related sub-components. If you want to enable only the components needed here, do not set this packagegroup. Instead, enable the following individual components:  
open-amp, open-amp-demos, libmetal

### 5. Setting Device Tree for the Linux Userspace RPMsg Application Demo

The libmetal Linux demo uses Userspace I/O (UIO) devices for IPI and shared memory. Copy the following to

<plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi in the PetaLinux project and modify as needed.

```
/ {
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;
        /* Reserved DDR memory for RPU firmware and shared memory between APU and RPU */
        rproc_0_reserved: rproc@3ed00000 {
            no-map;
            reg = <0x0 0x3ed00000 0x0 0x1000000>;
        };
    };
    amba {
        /* UIO device node for vring device memory */
        vring: vring@0 {
            compatible = "vring_uio";
            reg = <0x0 0x3ed40000 0x0 0x40000>;
        };
        /* UIO device node for shared memory device memory */
        shm0: shm@0 {
            compatible = "shm_uio";
            reg = <0x0 0x3ed80000 0x0 0x80000>;
        };
        /* UIO device node for IPI device */
        ipi0: ipi@0 {
            compatible = "ipi_uio";
            reg = <0x0 0xff340000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
        };
    };
};
```

**Note:** As the default APU IPI has been dedicated to PMU FW communication, OpenAMP picked another IPI (PL0 IPI) for communication notification.

You can find the source code of the Linux userspace RPMsg applications demos in the following locations:

- For the common code across the three applications:
- `platform_info.c` and `platform_info.h` define platform specific data and implement API's to set platform specific information for OpenAMP.
  - [https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/platform\\_info.c](https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/platform_info.c)
  - [https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/platform\\_info.h](https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/platform_info.h)
- `rsc_table.c` and `rsc_table.h` populate the resource table for the remote core for use by the Linux master.
  - [https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/rsc\\_table.c](https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/rsc_table.c)
  - [https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/rsc\\_table.h](https://github.com/xilinx/open-amp/blob/master/apps/machine/zynqmp/rsc_table.h)
- Application specific code:
  - [https://github.com/xilinx/open-amp/blob/master/apps/echo\\_test/echo\\_test.c](https://github.com/xilinx/open-amp/blob/master/apps/echo_test/echo_test.c)
  - [https://github.com/xilinx/open-amp/blob/master/apps/matrix\\_multiply/matrix\\_multiply.c](https://github.com/xilinx/open-amp/blob/master/apps/matrix_multiply/matrix_multiply.c)
  - [https://github.com/OpenAMP/open-amp/blob/master/apps/rpc\\_demo/rpc\\_demo.c](https://github.com/OpenAMP/open-amp/blob/master/apps/rpc_demo/rpc_demo.c)

6. Build the PetaLinux project with `petalinux-build`:

```
$ petalinux-build
```

The kernel images and the device tree binary are located in the `<plnx_proj>/images/linux` directory.

## Update RPU Firmware Resource Table with Xilinx SDK

In this demo, both the Linux and RPU are using static vdev and vrings from the resource table, and using the Linux kernel remoteproc driver to set the virtio device status is not supported. Edit the RPU firmware `rsc_table.c` file to set the virtio device status to `VIRTIO_CONFIG_STATUS_DRIVER_OK` of the `RSC_VDEV` entry in the resources structure. Following is the example:

```
{ RSC_VDEV, VIRTIO_ID_RPMSG, 0, RPMSG_IPU_C0_FEATURES, 0, 0,
  VIRTIO_CONFIG_STATUS_DRIVER_OK, NUM_VRINGS, {0, 0}, }
```

Without this change, the RPU firmware waits forever for this virtio device status bit to be set.



## 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 Xilinx SDK with 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/download.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
}
```

4. On the APU Linux target console, run the demo applications `echo_test-openamp`, `mat_mul_demo-openamp`, and `proxy_app-openamp`. This process produces output similar to the following:

```
# echo_test-openamp
metal: warning:   skipped page size 2097152 - invalid args
metal: info:      metal_uio_dev_open: No IRQ for device 3ed40000.vring.
metal: info:      metal_uio_dev_open: No IRQ for device 3ed40000.vring.
metal: info:      metal_uio_dev_open: No IRQ for device 3ed80000.shm.
echo test: sent : 488
received payload number 471 of size 488
*****
Test Results: Error count = 0
*****
Quitting application .. Echo test end
rpsmsg_channel_deleted
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2016_3:~#

# mat_mul-openamp
...
CLIENT> Matrix multiply: sent : 296
CLIENT> Quitting application .. Matrix multiplication end
CLIENT> *****
CLIENT> Test Results: Error count = 0
CLIENT> *****
CLIENT> rpsmsg_channel_deleted
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2016_3:~#
```

```
# proxy_app-openamp
login[1900]: root login on 'ttyPS0'
root@Xilinx-ZCU102-2016_3:~# proxy_app-openamp
...
Master> Remote proc resource initialized.
Master> RMSG channel has created.
Remote>FreeRTOS Remote Procedure Call (RPC) Demonstration
Remote>*****
Remote>Rmsg 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 rmsg-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
root@Xilinx-ZCU102-2016_3:~#
```

## System Design Consideration

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

### Supported Configuration

Note that **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 4-1: 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 <a href="#">Petalinux Images Quick Try</a>	Yes See <a href="#">Building Linux Applications Using OpenAMP RPMsg in Linux Userspace</a>	Yes See <a href="#">Xilinx Libmetal AMP Demo</a>
Supports warm restart: Auto APU/RPU reconnect after APU restart	Yes See <a href="http://www.wiki.xilinx.com/OpenAMP">http://www.wiki.xilinx.com/OpenAMP</a>	No	User defined
Supports pre-defined shared memory range	Yes See <a href="#">How to Write a Simple OpenAMP Application</a>	Yes See <a href="#">Building Linux Applications Using OpenAMP RPMsg in Linux Userspace</a>	Yes See <a href="#">Shared Memory and Enable Linux Demo Application Using Libmetal with PetaLinux Tools</a>
Linux can dynamically allocate shared memory range	Yes See <a href="#">How to Write a Simple OpenAMP Application</a>	No	No

Table 4-1: 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
Supports Multiple communication channels (e.g. both RPUs)	Yes See <a href="#">OpenAMP Demos</a>	Yes See <a href="#">OpenAMP Demos</a>	Yes See <a href="#">OpenAMP Demos</a>
Works with FSBL RPU boot	No	Yes	Yes See <a href="http://www.wiki.xilinx.com/OpenAMP">http://www.wiki.xilinx.com/OpenAMP</a>
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 Consideration

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.

## Known Limitations

The following are the known limitations in OpenAMP:

- Running OpenAMP demo for Zynq® devices with QEMU is not supported.  
Only OpenAMP demos for Zynq UltraScale+™ MPSoC devices are supported with QEMU.
- Shared memory cannot be used as normal memory in Linux Userspace. It must be used as device memory, since 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.

## Linux RPMsg Buffer Size

The OpenAMP message size is limited by the buffer size defined in the `rpmsg` kernel module. For the Linux 4.9 kernel, this is currently defined as 512 bytes with 16 bytes for the message header and 496 bytes of payload.



---

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

---

# Libmetal APIs

---

## Libmetal API Functions

The libmetal APIs described as follows are for libmetal users. If you are a libmetal developer who is changing the libmetal library to enable libmetal for their platform/OS, please refer to the libmetal doxygen for internal libmetal APIs.

### Top Level Interfaces

#### *metal\_init*

##### Description

Initialize libmetal library.

##### Arguments

params: Initialization params.

##### Returns

Returns 0 on success, or -errno on failure.

##### Usage

```
int metal_init(const struct metal_init_params params);
```

#### *metal\_finish*

##### Description

Shutdown libmetal library and release all reserved resources.

##### Usage

```
void metal_finish(void);
```

## Interrupt Handling Interfaces

### *metal\_irq\_handler*

#### Description

Type of interrupt handler.

#### Arguments

- irq: Interrupt id
- priv: Private data

#### Returns

Returns irq handled status.

#### Usage

```
typedef int (*metal_irq_handler) (int irq, void *priv);
```

### *metal\_irq\_register*

#### Description

- Register interrupt or register interrupt handling of a specific interrupt.
- If the interrupt handler parameter (irq\_handler) is NULL, deregister the interrupt handler.
- If the interrupt handler, device (dev), and driver ID (drv\_id) are NULL, deregister all handlers corresponding to the interrupt.
- If the interrupt handler is NULL, but either the device or the driver ID is not NULL, only deregister the interrupt handler Which has been registered with the same device and driver ID.

#### Arguments

irq	Interrupt id.
irq_handler	Interrupt handler.
dev	Metal device this irq belongs to.
drv_id	Driver id. It can be used for driver data.

#### Returns

Returns 0 on success, non-zero on failure.

### Usage

```
int metal_irq_register(int irq, metal_irq_handler irq_handler, struct metal_device
*dev, void *drv_id)
```

### ***metal\_irq\_save\_disable***

#### Description

Disable interrupts.

#### Returns

Interrupts state.

### Usage

```
unsigned int metal_irq_save_disable(void);
```

### ***metal\_irq\_restore\_enable***

#### Description

Restores interrupts to their previous state.

#### Arguments

Flags previous interrupts state.

### Usage

```
void metal_irq_restore_enable(unsigned int flags);
```

### ***metal\_irq\_enable***

#### Description

Enables the given interrupt.

#### Arguments

- Vector
- Interrupt vector number

### Usage

```
void metal_irq_enable(unsigned int vector);
```



## *metal\_irq\_disable*

### Description

Disables the given interrupt.

### Arguments

- Vector
- Interrupt vector number

### Usage

```
void metal_irq_disable(unsigned int vector);
```

## Shared Memory Interfaces

### *metal\_shmem\_open*

#### Description

Open a libmetal shared memory segment.

#### Arguments

<code>name</code>	Name of segment to open.
<code>size</code>	Size of segment.
<code>io</code>	I/O region handle, if successful.

#### Returns

Returns 0 on success, or -errno on failure.

#### Usage

```
extern int metal_shmem_open(const char *name, size_t size, struct metal_io_region
**io);
```

### *metal\_shmem\_register\_generic*

#### Description

- Statically register a generic shared memory region.
- Shared memory regions may be statically registered at application initialization, or may be dynamically opened.
- This interface is used for static registration of regions.

- Subsequent calls to `metal_shmem_open()` look up in this list of pre-registered regions.

### Arguments

shmem: Generic shmem structure.

### Returns

Returns 0 on success, or `-errno` on failure.

### Usage

```
extern int metal_shmem_register_generic(struct metal_generic_shmem *shmem);
```

## Spinlock Interfaces

### *metal\_spinlock\_init*

#### Description

Initialize a libmetal spinlock.

#### Arguments

slock: Spinlock to initialize.

#### Usage

```
static inline void metal_spinlock_init(struct metal_spinlock *slock)
```

### *metal\_spinlock\_acquire*

#### Description

Acquire a spinlock.

#### Arguments

slock: Spinlock to acquire.

#### Usage

```
static inline void metal_spinlock_acquire(struct metal_spinlock *slock)
```

## ***metal\_spinlock\_release***

### **Description**

Release a previously acquired spinlock.

### **Arguments**

slock: Spinlock to release.

### **Usage**

```
static inline void metal_spinlock_release(struct metal_spinlock *slock)
```

## **Sleep Interfaces**

### ***metal\_sleep\_usec***

### **Description**

Delay the next execution in the calling thread for usec microseconds.

### **Arguments**

usec: Microsecond intervals

### **Returns**

Returns 0 on success, non-zero for failures.

### **Usage**

```
int metal_sleep_usec(unsigned int usec);
```

## **Mutex Interfaces**

### ***metal\_mutex\_init***

### **Description**

Initialize a libmetal mutex.

### **Arguments**

mutex Mutex to initialize.

### **Usage**

```
static inline void metal_mutex_init(metal_mutex_t *mutex); metal_mutex_deinit
```

### Description

Deinitialize a libmetal mutex.

### Arguments

mutex: Mutex to deinitialize.

### Usage

```
static inline void metal_mutex_deinit(metal_mutex_t *mutex);
```

## ***metal\_mutex\_deinit***

### Description

Deinitialize a metal mutex.

### Arguments

mutex: Mutex to check.

### Usage

```
static inline void metal_mutex_deinit(metal_mutex_t *mutex);
```

## ***metal\_mutex\_try\_acquire***

### Description

Try to acquire a mutex.

### Arguments

mutex: Mutex to mutex.

### Returns

0 on failure to acquire, non-zero on success.

### Usage

```
static inline int metal_mutex_try_acquire(metal_mutex_t *mutex);
```

## ***metal\_mutex\_acquire***

### Description

Acquire a mutex.

### Arguments

mutex: Mutex to mutex.

### Usage

```
static inline void metal_mutex_acquire(metal_mutex_t *mutex);
```

## ***metal\_mutex\_release***

### Description

Release a previously acquired mutex.

### Arguments

mutex: Mutex to mutex.

### Usage

```
static inline void metal_mutex_release(metal_mutex_t *mutex);
```

## ***metal\_mutex\_is\_acquired***

### Description

Checked if a mutex has been acquired.

### Arguments

mutex: Mutex to check.

### Usage

```
static inline int metal_mutex_is_acquired(metal_mutex_t *mutex);
```

## I/O Interfaces

### ***metal\_io\_init***

### Description

Open a libmetal I/O region.

### Arguments

io	I/O region handle.
virt	Virtual address of region.

<code>physmap</code>	Array of physical addresses per page.
<code>size</code>	Size of region.
<code>page_shift</code>	Log2 of page size (-1 for single page).
<code>mem_flags</code>	Memory flags.
<code>ops</code>	<code>ops</code>

### Usage

```
static inline void metal_io_init(struct metal_io_region *io, void *virt, const
metal_phys_addr_t *physmap, size_t size, unsigned page_shift, unsigned int
mem_flags, const struct metal_io_ops *ops)
```

## ***metal\_io\_finish***

### Description

Close a libmetal shared memory segment.

### Arguments

`io`: I/O region handle

### Usage

```
static inline void metal_io_finish(struct metal_io_region *io)
```

## ***metal\_io\_region\_size***

### Description

Get size of I/O region.

### Arguments

`io`: I/O region handle

### Returns

Size of I/O region.

### Usage

```
static inline size_t metal_io_region_size(struct metal_io_region *io)
```

## ***metal\_io\_virt***

### **Description**

Get virtual address for a given offset into the I/O region.

### **Arguments**

- io: I/O region handle.
- offset: Offset into shared memory segment.

### **Returns**

NULL if offset is out of range, or pointer to offset.

### **Usage**

```
static inline void metal_io_virt(struct metal_io_region *io, unsigned long offset)
```

## ***metal\_io\_virt\_to\_offset***

### **Description**

Convert a virtual address to offset within I/O region.

### **Arguments**

- io: I/O region handle.
- virt: Virtual address within segment..

### **Returns**

METAL\_BAD\_OFFSET if out of range, or offset.

### **Usage**

```
static inline unsigned long metal_io_virt_to_offset(struct metal_io_region *io, void *virt)
```

## ***metal\_io\_phys***

### **Description**

Get physical address for a given offset into the I/O region.

### **Arguments**

- io: I/O region handle.

- offset: Offset into shared memory segment.

### Returns

METAL\_BAD\_PHYS if offset is out of range, or physical address of offset.

### Usage

```
static inline metal_phys_addr_t metal_io_phys(struct metal_io_region *io, unsigned long offset)
```

## *metal\_io\_phys\_to\_offset*

### Description

Convert a physical address to offset within I/O region.

### Arguments

- io: I/O region handle.
- phys: Physical address within segment.

### Returns

METAL\_BAD\_OFFSET if out of range, or offset.

### Usage

```
static inline unsigned long metal_io_phys_to_offset(struct metal_io_region *io, metal_phys_addr_t phys)
```

## *metal\_io\_phys\_to\_virt*

### Description

Convert a physical address to virtual address.

### Arguments

- io: Shared memory segment handle.
- phys: Physical address within segment.

### Returns

NULL if out of range, or corresponding virtual address.

### Usage

```
static inline void metal_io_phys_to_virt(struct metal_io_region *io, metal_phys_addr_t phys)
```



## ***metal\_io\_virt\_to\_phys***

### **Description**

Convert a virtual address to physical address.

### **Arguments**

- io: Shared memory segment handle.
- virt: Virtual address within segment.

### **Returns**

METAL\_BAD\_PHYS if out of range, or corresponding physical address.

### **Usage**

```
static inline metal_phys_addr_t metal_io_virt_to_phys(struct metal_io_region *io,
void *virt)
```

## ***metal\_io\_read***

### **Description**

Read a value from an I/O region.

### **Arguments**

- io: I/O region handle.
- offset: Offset into I/O region.
- order: Memory ordering.
- width: Width in bytes of datatype to read. This must be 1, 2, 4, or 8, and a compile time constant for this function to inline cleanly.

### **Returns**

Value.

### **Usage**

```
static inline uint64_t metal_io_read(struct metal_io_region *io, unsigned long
offset, memory_order order, int width)
```

## ***metal\_io\_write***

### **Description**

Write a value into an I/O region.

### Arguments

- io: I/O region handle.
- offset: Offset into I/O region.
- value: Value to write.
- order: Memory ordering.
- width: Width in bytes of datatype to read. This must be 1, 2, 4, or 8, and a compile time constant for this function to inline cleanly.

### Usage

```
static inline void metal_io_write(struct metal_io_region *io, unsigned long offset,
uint64_t value, memory_order order, int width)
```

## ***metal\_io\_block\_read***

### Description

Read a block from an I/O region.

### Arguments

- io: I/O region handle.
- offset: Offset into I/O region.
- dst: destination to store the read data.
- len: length in bytes to read.

### Returns

On success, number of bytes read. On failure, negative value.

### Usage

```
int metal_io_block_read(struct metal_io_region *io, unsigned long offset, void
*restrict dst, int len);
```

## ***metal\_io\_block\_write***

### Description

Write a block into an I/O region.

### Arguments

- io: I/O region handle.

- offset: Offset into I/O region.
- src: Source to write.
- len: Length in bytes to write.

### Returns

On success, number of bytes written. On failure, negative value.

### Usage

```
int metal_io_block_write(struct metal_io_region *io, unsigned long offset, const
void *restrict src, int len);
```

## *metal\_io\_block\_set*

### Description

Fill a block of an I/O region.

### Arguments

- io: I/O region handle.
- offset: Offset into I/O region.
- value: Value to fill into the block
- len: Length in bytes to fill.

### Returns

On success, number of bytes filled. On failure, negative value.

### Usage

```
int metal_io_block_set(struct metal_io_region *io, unsigned long offset, unsigned
char value, int len);
```

## Bus Abstratction

## *metal\_bus\_register*

### Description

Register a libmetal bus.

### Arguments

bus: Pre-initialized bus structure.

### Returns

0 on success, or -errno on failure.

### Usage

```
extern int metal_bus_register(struct metal_bus *bus);
```

## ***metal\_bus\_unregister***

### Description

Unregister a libmetal bus.

### Arguments

bus: Pre-registered bus structure.

### Returns

0 on success, or -errno on failure.

### Usage

```
extern int metal_bus_unregister(struct metal_bus *bus);
```

## ***metal\_bus\_find***

### Description

Find a libmetal bus by name.

### Arguments

- name: Bus name.
- bus: Returned bus handle.

### Returns

0 on success, or -errno on failure.

### Usage

```
extern int metal_bus_find(const char *name, struct metal_bus **bus);
```

## metal\_register\_generic\_device

### Description

Statically register a generic libmetal device. Devices may be statically registered at application initialization, or may be dynamically opened via sysfs or libfdt based enumeration at runtime. This interface is used for static registration of devices. Subsequent calls to metal\_device\_open() look up in this list of pre-registered devices on the "generic" bus.

### Arguments

device: Generic device.

### Returns

0 on success, or -errno on failure.

### Usage

```
extern int metal_register_generic_device(struct metal_device *device);
```

## metal\_device\_open

### Description

Open a libmetal device by name.

### Arguments

- bus\_name: Bus name.
- dev\_name: Device name.
- device: Returns device handle.

### Returns

0 on success, or -errno on failure.

### Usage

```
extern int metal_device_open(const char *bus_name, const char *dev_name, struct metal_device **device);
```

## metal\_device\_close

### Description

Close a libmetal device.

### Arguments

device: Device handle.

### Usage

```
extern void metal_device_close(struct metal_device *device);
```

## ***metal\_device\_io\_region***

### Description

Get an I/O region accessor for a device region.

### Arguments

- device: Device handle.
- index: Region index.

### Returns

I/O accessor handle, or NULL on failure.

### Usage

```
static inline struct metal_io_region metal_device_io_region(struct metal_device
*device, unsigned index)
```

## Condition Variable Interfaces

### ***metal\_condition\_init***

### Description

Initialize a libmetal condition variable.

### Arguments

cv: Condition variable to initialize.

### Usage

```
static inline void metal_condition_init(struct metal_condition *cv);
```

### ***metal\_condition\_signal***

### Description

Notify one waiter before calling this function, the caller should have acquired the mutex.

### Arguments

cv: Condition variable

### Returns

Zero on no errors, non-zero on errors.

### Usage

```
static inline int metal_condition_signal(struct metal_condition *cv);
```

## ***metal\_condition\_broadcast***

### Description

Notify all waiters before calling this function, the caller should have acquired the mutex.

### Arguments

cv: Condition variable

### Returns

Zero on no errors, non-zero on errors.

### Usage

```
static inline int metal_condition_broadcast(struct metal_condition *cv);
```

## ***metal\_condition\_wait***

### Description

Block until the condition variable is notified. Before calling this function, the caller should have acquired the mutex.

### Arguments

- cv: Condition variable
- m: Mutex

### Returns

0 on success, non-zero on failure.

### Usage

```
int metal_condition_wait(struct metal_condition *cv, metal_mutex_t *m);
```

## Allocation Interfaces

### *metal\_allocate\_memory*

#### Description

Allocate requested memory size. Returns a pointer to the allocated memory.

#### Arguments

size: Size in byte of requested memory.

#### Returns

Memory pointer, or 0 if it failed to allocate.

#### Usage

```
static inline void *metal_allocate_memory(unsigned int size);
```

### *metal\_free\_memory*

#### Description

Free the memory previously allocated.

#### Arguments

ptr: Pointer to memory.

#### Usage

```
static inline void metal_free_memory(void *ptr);
```

## Library Version Interfaces

### *metal\_ver\_major*

#### Description

Library major version number. Returns the major version number. This is required to match the value of METAL\_VER\_MAJOR, which is the major version of the library that the application was compiled against.

#### Returns

Major version number of the library linked into the application.



### Usage

```
extern int metal_ver_major(void);
```

### ***metal\_ver\_minor***

### Description

Library minor version number. This could differ from the value of METAL\_VER\_MINOR, which is the minor version of the library that the application was compiled against.

### Returns

Minor version number of the library linked into the application.

### Usage

```
extern int metal_ver_minor(void);
```

### ***metal\_ver\_patch***

### Description

Library patch level. This could differ from the value of METAL\_VER\_PATCH, which is the patch level of the library that the application was compiled against.

### Returns

Patch level of the library linked into the application.

### Usage

```
extern int metal_ver_patch(void);
```

### ***metal\_ver***

### Description

Library version string. This could differ from the value of METAL\_VER, which is the version string of the library that the application was compiled against.

### Returns

Version string of the library linked into the application.

### Usage

```
extern const char *metal_ver(void);
```

# OpenAMP APIs

---

## Remoteproc APIs

### Introduction

The `remoteproc` APIs provided by the OpenAMP framework allows software applications on the master to manage the remote processor and its relevant software.

This chapter introduces the `remoteproc` implementation in the OpenAMP library, and provides a brief overview of the `remoteproc` APIs and workflow.

---

## Remoteproc API Functions

### `remoteproc_resource_init`

#### *Description*

Initializes resources for `remoteproc` remote configuration. Only `remoteproc` remote applications are allowed to call this function. This API is called when the remote application is running on the remote processor to create the `virtIO/RPMsg` devices which are used for IPC. This API causes `remoteproc` to use the `RPMsg` name service to announce the `RPMsg` channels served by the remote application.

#### *Usage*

```
int remoteproc_resource_init( struct rsc_table_info *rsc_info,
                             struct hil_proc *proc,
                             rpmsg_chnl_cb_t channel_created,
                             rpmsg_chnl_cb_t channel_destroyed,
                             rpmsg_rx_cb_t default_cb,
                             struct remote_proc **rproc_handle,
                             int rpmsg_role);
```

## Arguments

<code>rsc_info</code>	Pointer to resource table info control block.
<code>proc</code>	Pointer to the <code>hil_proc</code> .
<code>channel_created</code>	Callback function for channel creation
<code>channel_destroyed</code>	Callback function for channel deletion.
<code>rdefault_cb</code>	Default callback for channel I/O.
<code>rproc_handle</code>	Pointer to new remoteproc instance
<code>rpmsg_role</code>	- 1 for rpmsg master, or 0 for rpmsg slave

## Returns

Status of execution.

# remoteproc\_resource\_deinit

## Description

Uninitialized resources for `remoteproc` remote configuration.

## Usage

```
int remoteproc_resource_deinit(struct remote_proc *rproc);
```

## Arguments

`rproc` - pointer to `remoteproc` instance.

## Returns

Status of execution.

## remoteproc\_shutdown

### Description

This function shutdowns the remote execution context.

### Usage

```
int remoteproc_shutdown(struct remote_proc *rproc);
```

### Arguments

`rproc` - pointer to `remoteproc` instance to shutdown.

### Returns

Status of function execution.

---

## RPMMsg Development

### Introduction

The RPMMsg APIs provided by the OpenAMP framework allow bare-metal or RTOS applications to perform inter-process interrupts (IPI) in an AMP configuration, running on either a master or remote processor. This information is based on the documentation available in the `rpmsg.h` header file.

This chapter introduces the RPMMsg implementation in the OpenAMP library, and provides a brief overview of the RPMMsg APIs and workflow.

---

## RPMMsg API Functions

### rpmsg\_sendto

#### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the `rpdev` channel using the source address of the `rpdev`.

If there are no TX buffers available, the function remains blocked until one becomes available, or a time-out of 15 seconds elapses. When the latter occurs, ERESTARTSYS is returned. This API can be called from process context only.

## Usage

```
static inline int rpmsg_sendto ( struct rpmsg_channel *rpdev,
                                void *data, int len, unsigned long dst)
```

## Arguments

rpdev	The RPMsg channel
data	Payload of message
len	Length of payload
dst	Destination address

## Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_send

### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the rpdev channel using the source and destination address of the rpdev. If there are no Tx buffers available, the function remains blocked until one becomes available, or a time-out of 15 seconds elapses. When the latter occurs, ERESTARTSYS is returned. Presently, this API can be called from process context only.

## Usage

```
static inline int rpmsg_send(struct rpmsg_channel *rpdev, void *data, int len)
```

## Arguments

rpdev	The rpmsg channel
data	Payload of message
len	Length of payload

## Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_send\_offchannel

### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the `rpdev` channel using `src` as the source address. If there are no TX buffers available, the function remains blocked until one becomes available, or a time-out of 15 seconds elapses. When the latter occurs, `ERESTARTSYS` is returned. This API can be called from process context only.

### Usage

```
static inline int rpmsg_send_offchannel(struct rpmsg_channel *rpdev,
                                       uint32_t, uint32_t dst,
                                       void *data, int len)
```

### Arguments

<code>rpdev</code>	The <code>rpmsg</code> channel.
<code>src</code>	Source address.
<code>dst</code>	Destination address.
<code>data</code>	Payload of message.
<code>len</code>	Length of payload.

### Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_trysend

### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the `rpdev` channel using the source of the `rpdev` and destination addresses. If there are no TX buffers available, the function immediately returns `ENOMEM` without waiting until one becomes available. This API can be called from process context only.

### Usage

```
static inline int rpmsg_trysend(struct rpmsg_channel *rpdev, void *data, int len)
```

### Arguments

<code>rpdev</code>	The rpmsg channel
<code>data</code>	Payload of message
<code>len</code>	Length of payload

### Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_trysendto

### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the `rpdev` channel using the source addresses of the `rpdev`. If there are no TX buffers available, the function immediately returns `ENOMEM` without waiting until one becomes available. This API can be called from the process context only.

### Usage

```
static inline int rpmsg_trysendto(struct rpmsg_channel *rpdev,
                                void *data, int len, uint32_t dst)
```

### Arguments

<code>rpdev</code>	The rpmsg channel
<code>data</code>	Payload of message
<code>len</code>	Length of payload
<code>dst</code>	Destination address

### Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_trysend\_offchannel

### Description

Sends a message containing data and payload length to the destination address of the remote processor respective to the `rpdev` channel using `src` as the source address. If there are no TX buffers available, the function immediately returns `ENOMEM` without waiting until one becomes available. This API can be called from process context only.

## Usage

```
static inline int rpmsg_trysend_offchannel (struct rpmsg_channel *rpdev,
                                           uint32_t src,
                                           uint32_t long dst,
                                           void *data, int len)
```

## Arguments

<code>rpdev</code>	The RPMsg channel.
<code>src</code>	Source address.
<code>dst</code>	Destination address.
<code>data</code>	Payload of message.
<code>len</code>	Length of payload.

## Returns

Returns 0 on success, and an appropriate error value upon failure.

## rpmsg\_init

### Description

Allocates and initializes the `rpmsg` driver resources for a given device ID (`cpu_id`). The successful return from this function enables the IPC link.

## Usage

```
int rpmsg_init( struct hil_proc *proc, struct remote_device **rdev,
                rpmsg_chnl_cb_t channel_created,
                rpmsg_chnl_cb_t channel_destroyed,
                rpmsg_rx_cb_t default_cb, int role);
```

## Arguments

<code>param proc</code>	The pointer to <code>hil_proc</code> .
<code>@param rdev</code>	Source address.
<code>@param channel_created</code>	Destination address.
<code>@param channel_destroyed</code>	Callback function for channel deletion.
<code>@default_cb</code>	Payload of message.
<code>@param role</code>	Length of payload.



### Returns

Status of function execution.

## rpmsg\_deinit

### Description

Releases the rpmsg driver resources for a given remote instance.

### Usage

```
void rpmsg_deinit(struct remote_device *rdev);
```

### Arguments

rdev: Pointer to device de-initialize.

### Returns

None.

## rpmsg\_get\_buffer\_size

### Description

Returns buffer size available for sending messages.

### Usage

```
int rpmsg_get_buffer_size(struct rpmsg_channel *rp_chnl)
```

### Arguments

Channel: Pointer to the rpmsg channel or device.

### Returns

Buffer size.

## rpmsg\_create\_channel

### Description

Creates rpmsg channel with the given name for remote device.

# Additional Resources and Legal Notices

---

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

---

## Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

---

## Documentation Navigator and Design Hubs

Xilinx<sup>®</sup> Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx 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:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

**Note:** For more information on Documentation Navigator, see the [Documentation Navigator](#) page on the Xilinx website.

## Xilinx Documentation

1. OpenAMP Wiki: <http://www.wiki.xilinx.com/OpenAMP>
2. Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085)
3. Xilinx Software Developer Kit Help (UG782)
4. PetaLinux Tools Documentation: Reference Guide (UG1144)
5. Xilinx libmetal source code: <https://github.com/Xilinx/libmetal>
6. Xilinx OpenAMP source code: <https://github.com/Xilinx/open-amp>

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

### AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

This document contains preliminary information and is subject to change without notice. Information provided herein relates to products and/or services not yet available for sale, and provided solely for information purposes and are not intended, or to be construed, as an offer for sale or an attempted commercialization of the products and/or services referred to herein.

© Copyright 2015-2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries.