Libmetal and OpenAMP User Guide

UG1186 (v2019.1) May 22, 2019

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
General Updates.	General Updates.

Table of Contents

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:

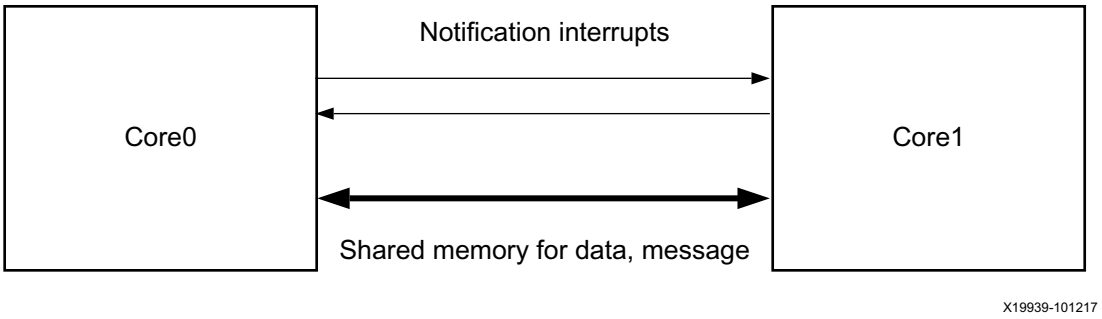

Figure 1-1: **Inter Processor Communication**

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

Software Tools Requirements

PetaLinux and Xilinx SDK (Software Development Kit) are required in order to follow the instructions in this document to build applications. See Xilinx Documentation for more detailed information.

- 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 = <00x 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 GIC (Generic Interrupt Controller) using the IPI (Inter-Processor Interrupt) 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 (Userspace I/O) driver so interaction is limited to treating the memory as device memory.

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

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

```
static struct metal_device shm_dev = { /* shared memory device */
    .name = "3ed80000.shm", /* device name */
    .bus = NULL, /* device bus */
```

```
    .num_regions = 1, /* number of regions on device */
    {
      {
        .virt = (void*) 0x3ED80000, /* virtual address */
        .physmap = 0x3ED80000, /* physical address */
        .size = 0x800000, /* size of region */
        .page_shift = (sizeof(metal_phys_addr_t) << 3), /* page shift */
        .page_mask = (unsigned long)(-1), /* page mask */
        .mem_flags = NORM_SHARED_NCACHE | PRIV_RW_USER_RW, /* memory flags */
        .ops = {NULL}, /* user defined memory operations */
      }
    },
    .node = {NULL}, /* node to point to device in list of nodes on bus */
    .irq_num = 0, /* Number of IRQs per device. This is 0 because there are no
interrupts we want to use for this device.*/
    .irq_info = NULL, /* IRQ info. This is NULL because we are not using this device
for interrupts. */
  }
* Open the shared memory device, use the shared memory device as follows:
/* Open the shared memory device */
  ret = metal_device_open("platform", "3ed80000.shm", &dev); /* the first argument,
bus name, is 'platform' for generic platform. */
  /* get shared memory device IO region */
  io = metal_device_io_region(device, 0);
/* read data from the shared memory*/
  metal_io_block_read(io, READ_OFFSET, destination, data_length);
/* write data to the shared memory*/
  ret = metal_io_block_write(io, WRITE_OFFSET, source, data_length);
```

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 (Application Processing Unit) and RPU (Real-Time Processor) on a Zynq UltraScale+ MPSoC platform.

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

- DDR (Double Data Rate)
- 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 (Board Support Package) OS platform:
      - **standalone** for a bare-metal application.
   b. Select one of the predefined hardware platform: **ZCU102_hw_platform**.
   c. Select the processor:
      - One of the Cortex™-R5 (RPUs) 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_demod.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_demod.c`: Demo application that measures latency between APU and RPU.
- `ipi_shmem_demod.c`: Demonstrates how to access shared memory and IPI.
- `shmem_atomic_demod.c`: Demonstrates how to access shared memory with atomics.
- `shmem_demod.c`: Demonstrates use of shared memory between APU and RPU.
- `shmem_latency_demod.c`: Demo application that measures shared memory latency between APU and RPU.
- `shmem_throughput_demod.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

Use PetaLinux Tools to complete the following 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";
};
```

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

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:

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/lscript.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

- 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. Building and package sysroots.

```
$ petalinux-build -s
$ petalinux-package --sysroot
```

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

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

4. Select **Linux Hello World** and then click **Finish**.
5. 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>/images/linux/sdk/sysroots/aarch64-xilinx-linux"
   click OK
```

6. Copy files located at (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.

7. 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/firware 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 ******
```

Note: One method with which the application can be debugged is XSDB. See the *Embedded System Tools Reference Manual* (UG1043) for more information on the use of XSDB.

OpenAMP

Overview

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

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

Components in OpenAMP

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

virtIO: OpenAMP library implements virtIO standard for shared memory management. The `virtIO` is a virtualization standard for network and disk device drivers where only the driver on the guest device is aware it is running in a virtual environment, with the hypervisor.

remoteproc: Remoteproc provides capability for life cycle management (LCM) of the remote processors. The `remoteproc` API that OpenAMP library uses is compliant with the infrastructure present in the Linux Kernel 3.18 and later. The remoteproc uses information published through the remote processor firmware resource table to allocate system resources and to create `virtIO` devices. The remoteproc can be used to load arbitrary firmware; it is not limited to OpenAMP firmware.

RPMsg: This API allows inter-process communications (IPC) between software running on independent cores in an AMP system. This is also compliant with the `RPMsg` bus infrastructure present in the Linux Kernel version 3.18 and later.

The following diagrams show how OpenAMP is used in Xilinx Zynq and Zynq UltraScale+ MPSoC platforms:

1. Linux kernel master and RPU OpenAMP slave.

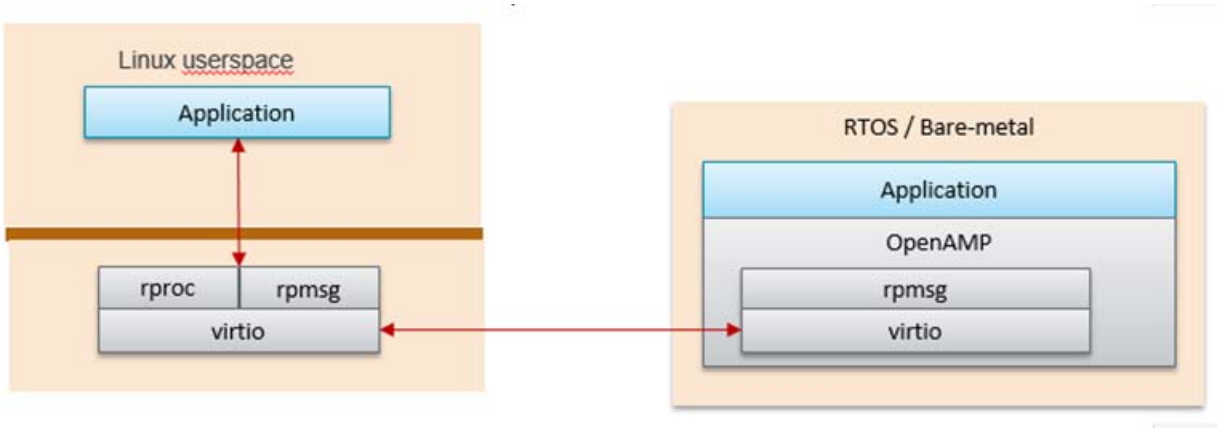

Figure 3-1: **RPMsg Implementation in Kernel Space**

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.

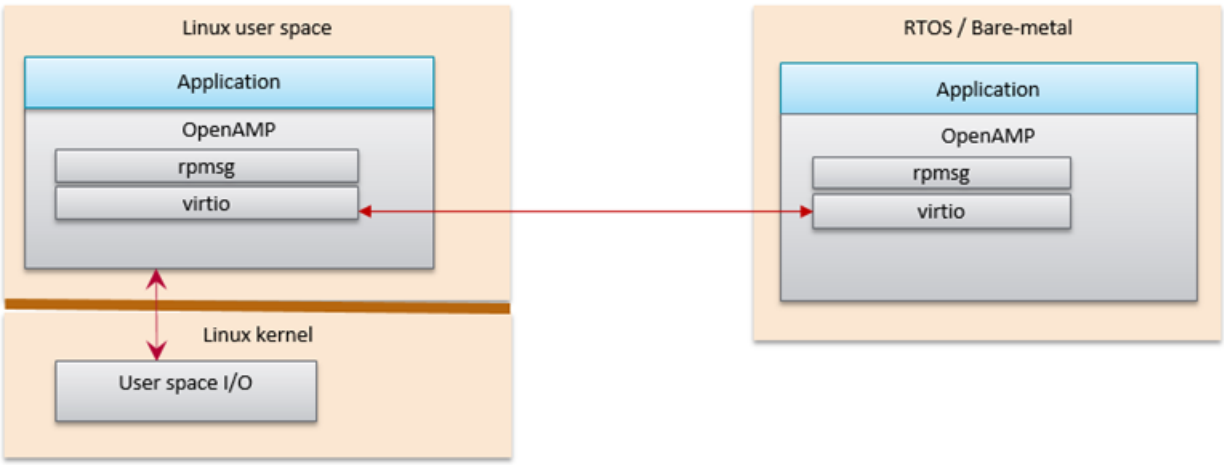

Figure 3-2: **OpenAMP RPMsg Implementation in Linux Userspace**

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 (Inter-Processor Interrupt) and shared memory. OpenAMP leverages standards for shared memory management, lifecycle management and communication. A diagram to show the connection between libmetal and OpenAMP is as follows:

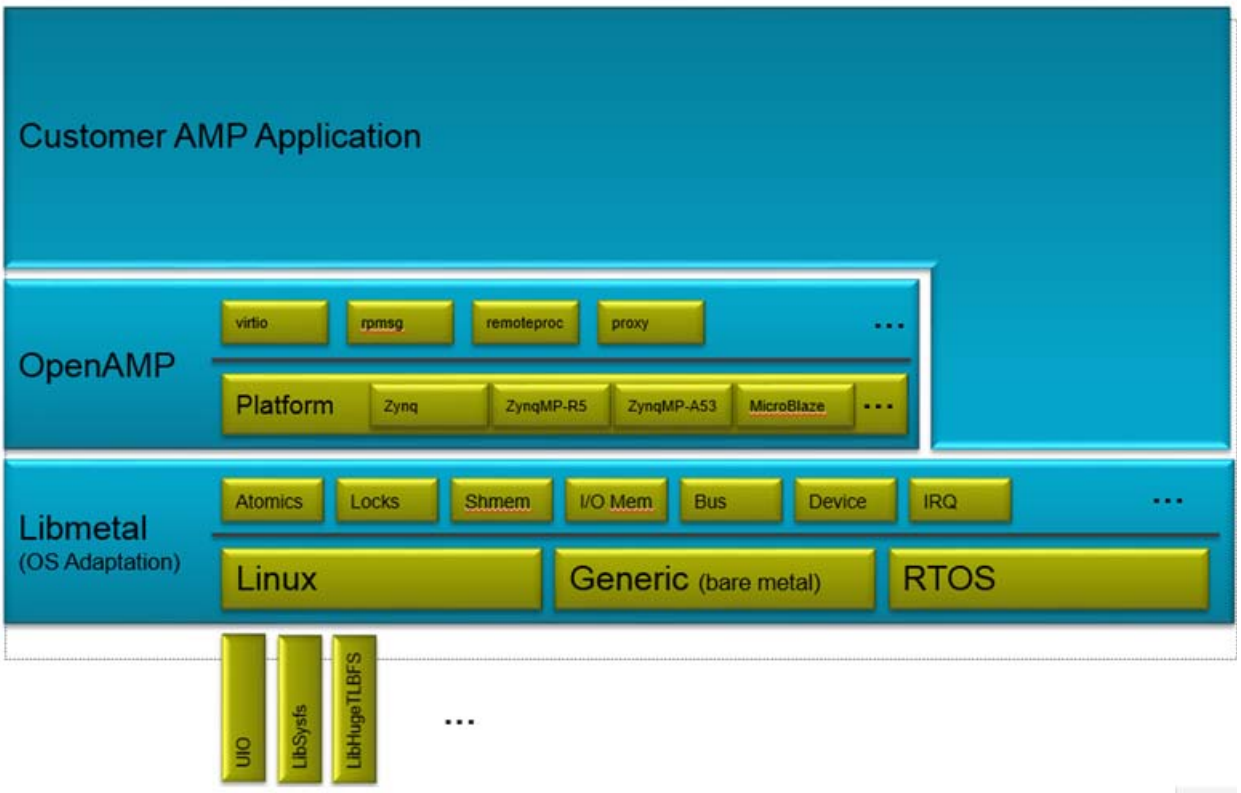

Figure 3-3: **Libmetal and OpenAMP Connection**

How to Write a Simple OpenAMP Application

To write an OpenAMP application there 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_proc.

2. Create remoteproc struct using resource table.
3. Define RPMsg callback functions.
4. Create RPMsg virtio device.
5. Create an RPMsg endpint and associate the RPMsg device with the callback functions.
6. Use rpmsg_send() to send message across to the remote processor.
7. After initializing the framework, the flow of an OpenAMP application consists of the PRMsg channel acting as communication between the master and remote processor via the RPMsg send() and I/O callback functions. The following is a flow diagram to show this.

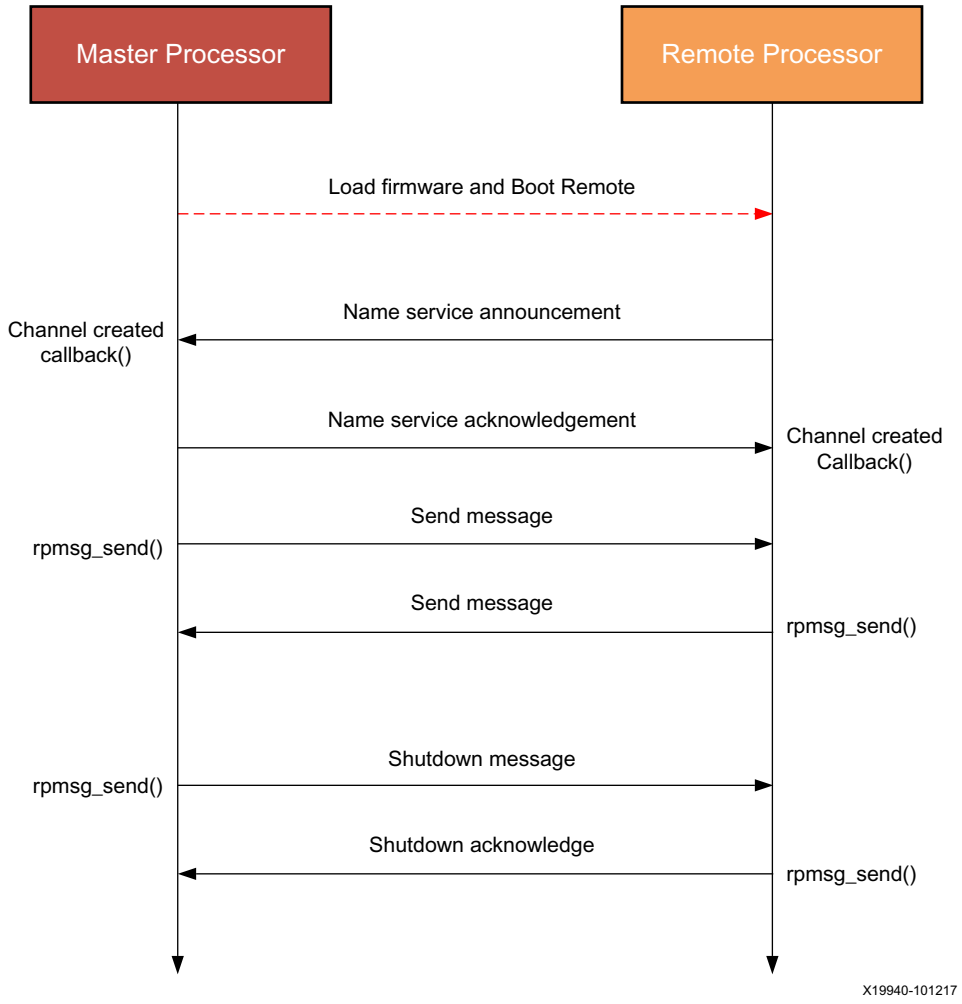

Figure 3-4: **Flow Diagram**

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

```
struct resource_table table = {
/* Version number. If the structure changes in the future, this acts as
```

```
 * reference to what the structure is.
 */
  .ver = 1,
* Number of resources; Matches number of offsets in array */
  .num = 2,
/* reserved (must be zero) */
  .reserved = 0,
  { /* array of offsets pointing at various resource entries   */
/* This RSC_RPROC_MEM entry set the shared memory address range. It is required to
tell the Linux kernel range of the shared memory the remote can access. */
 */
    {RSC_RPROC_MEM, 0x3ed40000, 0x3ed40000, 0x100000, 0},
/* virtio device header */
    {
   RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0, 0,
   NUM_VRINGS, {0, 0},
   }
  }
};
#include <openamp/remoteproc.h>
#include <openamp/rpmsg.h>
#include <openamp/rpmsg_virtio.h>

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

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

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

/* RPMsg device */
struct rpmsg_device *rpmsg_dev;

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

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

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

```
struct metal_io_region *shm_io;

/* VirtIO device */
struct virtio_device *vdev;

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

/* Shared buffers */
void *shbuf;

/* RPMsg endpoint */
struct rpmsg_endpoint ept;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  rpmsg_deinit_vdev(&rpmsg_vdev);

  remoteproc_remove_virtio(&rproc, vdev);

  remoteproc_remove(&rproc);
}
```

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-v2018.3-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
 [  265.772355] remoteproc remoteproc0: powering up ff9a0100.zynqmp_r5_rproc
[  265.779900] remoteproc remoteproc0: Booting fw image
echotest_standalone_r5_0.elf, size 719860
[  265.790005] zynqmp_r5_remoteproc ff9a0100.zynqmp_r5_rproc: RPU boot from TCM.
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.
[  265.797738] remoteproc remoteproc0: registered virtio0 (type 7)
[  265.800388] virtio_rpmsg_bus virtio0: rpmsg host is online
[  265.830254] remoteproc remoteproc0: remote processor ff9a0100.zynqmp_r5_rproc is
now up
[  265.838381] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel
addr 0x0
root@xilinx-zcu102-2019_1:/lib/firmware# echo_test -d /dev/rpmsg0

 Echo test start

 Open rpmsg dev /dev/rpmsg0!

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

  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 the **OK** button.

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 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

- **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 will allocate 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 RPMsg 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 (that is, 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 --->
-> Petalinux Package Groups
 -> 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
```

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

- rpmsg-echo-test: https://github.com/Xilinx/meta-openamp/tree/rel-v2019.1/recipes-openamp/rpmsg-examples/rpmsg-echo-test
- rpmsg-mat-mul: https://github.com/Xilinx/meta-openamp/tree/rel-v2019.1/recipes-openamp/rpmsg-examples/rpmsg-mat-mul
- rpmsg-proxy-app: https://github.com/Xilinx/meta-openamp/blob/master/recipes-openamp/rpmsg-examples/rpmsg-proxy-app/proxy_app.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 DTB (Device Tree Binary) 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.dtsi
```

The `openamp.dtb` and `openamp.dtsi` files are provided for reference only. You need to edit the `system-user.dtsi` file to include the content from `openamp.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;
    rproc_0_dma: rproc@3ed400000 {
      no-map;
      compatible = "shared-dma-pool";
      reg = <0x0 0x3ed40000 0x0 0x100000>;
    };
    rproc_0_reserved: rproc@3ed000000 {
      no-map;
      reg = <0x0 0x3ed00000 0x0 0x40000>;
    };
  };

  zynqmp-rpu {
    compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;
    core_conf = "split";
    r5_0: r5@0 {
      #address-cells = <2>;
      #size-cells = <2>;
      ranges;
      memory-region = <&rproc_0_reserved>, <&rproc_0_dma>;
      pnode-id = <0x7>;
      mboxes = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;
      mbox-names = "tx", "rx";
      tcm_0_a: tcm_0@0 {
        reg = <0x0 0xFFE00000 0x0 0x10000>;
        pnode-id = <0xf>;
      };
      tcm_0_b: tcm_0@1 {
        reg = <0x0 0xFFE20000 0x0 0x10000>;
        pnode-id = <0x10>;
      };
    };
  };


  zynqmp_ipi1 {
    compatible = "xlnx,zynqmp-ipi-mailbox";
    interrupt-parent = <&gic>;
    interrupts = <0 29 4>;
    xlnx,ipi-id = <7>;
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    /* APU<->RPU0 IPI mailbox controller */
    ipi_mailbox_rpu0: mailbox@ff90000 {
      reg = <0xff990600 0x20>,
```

```
                <0xff990620 0x20>,
                <0xff9900c0 0x20>,
                <0xff9900e0 0x20>;
        reg-names = "local_request_region",
                "local_response_region",
                "remote_request_region",
                "remote_response_region";
        #mbox-cells = <1>;
        xlnx,ipi-id = <1>;
      };
    };
  };
```

For reference to device tree documentation on mailbox usage for device tree please see: https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/devicetree/bindings/mailbox/xlnx%2czynqmp-ipi-mailbox.txt

For reference to device tree documentation on ZynqMP usage of remoteproc driver see: https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/devicetree/bindings/remoteproc/xilinx%2czynqmp-r5-remoteproc.txt

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

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_ddr_0: ddr@0 {
        compatible = "mmio-sram";
        reg = <0x3e000000 0x400000>;
      };
    };
    remoteproc0: remoteproc@0 {
```

```
        compatible = "xlnx,zynq_remoteproc";
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        srams = <&elf_ddr_0>;
      };
};
```

Building the Applications and the Linux Project

To build the applications and Linux project, do the following:

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 (Quick Emulator) or hardware.

Booting on QEMU

After a successful build, you can run the PetaLinux project on QEMU as follows.

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 ZynqMP.

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 (Universal Asynchronous Receiver/Transmitter) 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
```

2. Run the test:

```
echo_test
```

The test starts.

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

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

Debugging an OpenAMP Application

Debugging RPU Firmware

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

```
xsdb% bpadd -addr 0x3ed011c8
0
xsdb% Info: Breakpoint 0 status:
   target 7: {Address: 0x3ed011c8 Type: Hardware}
xsdb% dow ~/test.elf
Downloading Program -- ~/test.elf
  section, .vectors: 0x00000000 - 0x0000051f
  section, .text: 0x3ed00000 - 0x3ed0d73f
  section, .init: 0x3ed0d740 - 0x3ed0d74b
  section, .fini: 0x3ed0d74c - 0x3ed0d757
  section, .rodata: 0x3ed0d758 - 0x3ed0ee8f
  section, .data: 0x00000520 - 0x00001623
  section, .resource_table: 0x00001700 - 0x000017ff
  section, .eh_frame: 0x3ed0ee90 - 0x3ed0ee93
  section, .ARM.exidx: 0x3ed0ee94 - 0x3ed0ee9b
  section, .init_array: 0x3ed0ee9c - 0x3ed0eea3
  section, .fini_array: 0x3ed0eea4 - 0x3ed0eea7
  section, .bss: 0x3ed0eea8 - 0x3ed0f157
  section, .heap: 0x00001800 - 0x000057ff
  section, .stack: 0x00005800 - 0x00008fff
100%    0MB   0.3MB/s  00:00
Setting PC to Program Start Address 0x00000000
Successfully downloaded ~/test.elf
xsdb% con
xsdb% Info: Cortex-R5 #0 (target 7) Stopped at 0x3ed011c8 (Breakpoint)
platform_init() at ../src/platform_info.c: 295
295: {
xsdb% locals
argc     : 0
argv     : 0
platform  : 0
proc_id   : 0
rsc_id    : 1053874736
rproc     : 1053824852
xsdb% con
```

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

Debugging Linux OpenAMP Application

To generate a OpenAMP Linux application with debugging symbols in Petalinux do the following:

1. Enable open-amp demos and open-amp with debug symbols. These can be enabled via petalinux-config -c rootfs

   ```
   -> Filesystem Packages
   --> libs
   ---> open-amp
   [*]open-amp
   [*]open-amp-dbg
   [*]open-amp-demos
   ```

2. Enable the gdb package. The gdb package can be enabled as follows:

   ```
   petalinux-config -c rootfs
   -> Filesystem Packages
   --> misc
   ---> gdb
   ```

3. Build with petalinux-build.

   Please refer to gdb documentation for how to debug a linux application using gdb at: https://www.gnu.org/software/gdb/documentation/

Running the Matrix Multiplication Test

1. Load the Matrix Multiply firmware and RPMsg module:

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

2. Run the test:

   ```
   mat_mul_demo
   ```

   The test starts.

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

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

Running the Proxy Application

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

   ```
   proxy_app
   ```

2. When the application prompts you to *Enter name,* enter any string.
3. When the application prompts you to *Enter age*, enter any integer.
4. When the application prompts you to *Enter value for pi,* enter any floating point number.
5. The application 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;
        rproc_0_reserved: rproc@3ed000000 {
            no-map;
            reg = <0x0 0x3ed00000 0x0 0x1000000>;
        };
    };

    amba {
        /* Shared memory (APU to RPU) */
        shm0: shm@0 {
            compatible = "shm";
            reg = <0x0 0x3ed20000 0x0 0x0100000>;
            /* reg = <0x0 0x3ed04000 0x0 0x0100000>; */
        };
                /* IPI device */
        ipi0: ipi@0 {
            compatible = "ipi_uio";
            reg = <0x0 0xff340000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
        };
    };
};

&uart1 {
    status = "disabled";
};
```

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 specifc data and implement API's to set platform specific information for OpenAMP.
  - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/platform_info.c
  - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/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/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.c
  - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.h
- Application specific code:
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/echo/rpmsg-echo.c
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/matrix_multiply/matrix_multiply.c
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/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.

Building RPU Firmware

1. Add the following section to the linker and amend the LENGTH of the psu_ddr_S_AXI_BASEADDR to: 0x00040000:

```
.resource_table 0x3ed20000 : {
  . = ALIGN(4);
  *(.resource_table)
} > psu_ddr_S_AXI_BASEADDR
```

Note: The resource table section has be added to specify that it is now placed in reserved memory.

2. In rsc_table.c change RING_TX from FW_RSC_U32_ADDR_ANY to 0x3ed40000 and RING_RX from FW_RSC_U32_ADDR_ANY to 0x3ed44000 as OpenAMP's implementation of RPMsg in userspace only allows static vring entries.

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/system.bit
            [destination_cpu=pmu] <plnx_proj>/images/linux/pmufw.elf
            [destination_cpu=r5-0] <RPU firmware>
            [destination_cpu=a53-0, exception_level=el-3,
   trustzone] <plnx_proj>/images/linux/arm/bl31.elf
            [destination_cpu=a53-0,
   exception_level=el-2] <plnx_proj>/images/linux/u-boot.elf
   }
   ```

4. 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:

   ```
   root@xilinx-zcu102-2019_1:~# rpmsg-echo-ping-shared
   metal: info:      metal_uio_dev_open: No IRQ for device 3ed20000.shm.
   Successfulinitializing rpmsg vdev
   Try to create rpmsg endpoint.
   Successfully created rpmsg endpoint.
   ly open shm device.
   Successfully added shared memory
   Successfully probed IPI device
   Successfully initialized Linux r5 remoteproc.
   Successfully initialized remoteproc
   Calling mmap resource table.
   Successfully mmap resource table.
   Successfully set resource table to remoteproc.
   Creating virtio...
   Successfully created virtio device.
   initializing rpmsg vdev
   echo test: sent : 488
    received payload number 471 of size 488
   **********************************
    Test Results: Error count = 0
   **********************************
   Quitting application .. Echo test end
   rpmsg_channel_deleted
   WARNING rx_vq: freeing non-empty virtqueue
   WARNING tx_vq: freeing non-empty virtqueue
   root@Xilinx-ZCU102-2019_1:~#
   ```

```
# matrix_multiply-shared
...
CLIENT> Matrix multiply: sent : 296
CLIENT> Quitting application .. Matrix multiplication end
CLIENT> **********************************
CLIENT>  Test Results: Error count = 0
CLIENT> **********************************
CLIENT> rpmsg_channel_deleted
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2019_1:~#

# rpc_demod-shared
login[1900]: root login on 'ttyPS0'
root@Xilinx-ZCU102-2019_1:~# proxy_app-openamp
...
Master> Remote proc resource initialized.
Master> RPMSG channel has created.
Remote>FreeRTOS Remote Procedure Call (RPC) Demonstration
Remote>************************************************
Remote>Rpmsg based retargetting to proxy initialized..
Remote>FileIO demo ..

Remote>Creating a file on master and writing to it..
… …
Remote>Repeat demo ? (enter yes or no)
no
Remote>RPC retargetting quitting ...
Remote> Firmware's rpmsg-openamp-demo-channel going down!
Master>
RPC service exiting !!
Master> sending shutdown signal.
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2019_1:~#
```

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 Petalinux Images Quick Try	Yes See Building Linux Applications Using OpenAMP RPMsg in Linux Userspace	Yes See Xilinx Libmetal AMP Demo
Supports warm restart: Auto APU/RPU reconnect after APU restart	Yes See http://www.wiki.xilinx.com/OpenAMP	No	User defined
Supports pre-defined shared memory range	Yes See How to Write a Simple OpenAMP Application	Yes See Building Linux Applications Using OpenAMP RPMsg in Linux Userspace	Yes See Shared Memory and Enable Linux Demo Application Using Libmetal with PetaLinux Tools
Linux can dynamically allocate shared memory range	Yes See How to Write a Simple OpenAMP Application	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 OpenAMP Demos	Yes See OpenAMP Demos	Yes See OpenAMP Demos
Works with FSBL RPU boot	No	Yes	Yes See http://www.wiki.xilinx.com/OpenAMP
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 4.14 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

`name`	Name of segment to open.
`size`	Size of segment.
`io`	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.

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

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 Abstraction

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: Returnsed 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: Returnsed 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);
```

Libary 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_init

Description

Initialize `remoteproc` instance.

Usage

```
struct remoteproc *remoteproc_init(struct remoteproc *rproc,
          struct remoteproc_ops *ops, void *priv);
```

Arguments

`rproc`	Pointer to remoteproc instance
`ops`	Pointer to remoteproc operations
`priv`	Pointer to private data

Returns

Created `remoteproc` pointer.

remoteproc_remove

Description

Remove `remoteproc` instance.

Usage

```
int remoteproc_resource_remove(struct remoteproc *rproc);
```

Arguments

`rproc` - pointer to `remoteproc` instance.

Returns

No return.

remoteproc_get_io_with_name

Description

This function gets remoteproc memory I/O region with name.

Usage

```
struct metal_io_region *
remoteproc_get_io_with_name(struct remoteproc *rproc,
         const char *name);
```

Arguments

`rproc` - Pointer to the remote processor.

`name`- Name of the shared memory.

Returns

Metal I/O region pointer, NULL for failure.

remoteproc_get_io_with_pa

Description

This function gets `remoteproc` memory I/O region with physical address.

Usage

```
struct metal_io_region *
remoteproc_get_io_with_pa(struct remoteproc *rproc,
         metal_phys_addr_t pa);
```

Arguments

`rproc` - Pointer to the remote processor.

`pa`- Physical address.

Returns

Metal I/O region pointer, NULL for failure.

remoteproc_get_io_with_da

Description

This function gets `remoteproc` memory I/O region with device address.

Usage

```
struct metal_io_region *
remoteproc_get_io_with_da(struct remoteproc *rproc,
         metal_phys_addr_t da,
         unsigned long *offset);
```

Arguments

`rproc`	Pointer to the remote processor
`da`	Physical address
`offset`	I/O region offset of the device address

Returns

Metal I/O region pointer, NULL for failure.

remoteproc_get_io_with_va

Description

This function gets `remoteproc` memory I/O region with virtual address.

Usage

```
struct metal_io_region *
remoteproc_get_io_with_va(struct remoteproc *rproc,
         void *va);
```

Arguments

`rproc` - Pointer to the remote processor.

`va`- Virtual address.

Returns

Metal I/O region pointer, NULL for failure.

remoteproc_mmap

Description

This function asks `remoteproc` to mmap memory.

Usage

```
void *remoteproc_mmap(struct remoteproc *rproc,
          metal_phys_addr_t *pa, metal_phys_addr_t *da,
          size_t size, unsigned int attribute,
          struct metal_io_region **io);
```

Arguments

`rproc`	Pointer to the remote processor
`pa`	Physical address pointer
`da`	Device address pointer
`size`	Size of the memory
`attribute`	Memory attribute
`io`	Pointer to the I/O region

Returns

Returns pointer to the memory.

remoteproc_parse_rsc_table

Description

This function parses resource table of `remoteproc`.

Usage

```
int remoteproc_parse_rsc_table(struct remoteproc *rproc,
            struct resource_table *rsc_table,
            size_t rsc_size);
```

Arguments

`rproc`	Pointer to the remote instance
`rsc_table`	Pointer to resource table
`rsc_size`	Resource table size

Returns

Returns 0 for success and negative value for errors.

remoteproc_set_rsc_table

Description

This function parses and sets resource table of `remoteproc`.

Usage

```
int remoteproc_set_rsc_table(struct remoteproc *rproc,
          struct resource_table *rsc_table,
          size_t rsc_size);
```

Arguments

`rproc`	Pointer to the remote instance
`rsc_table`	Pointer to resource table
`rsc_size`	Resource table size

Returns

Returns 0 for success and negative value for errors.

remoteproc_create_virtio

Description

This function creates virtio device, it returns pointer to the created virtio device.

Usage

```
struct virtio_device *
remoteproc_create_virtio(struct remoteproc *rproc,
      int vdev_id, unsigned int role,
      void (*rst_cb)(struct virtio_device *vdev));
```

Arguments

`rproc`	Pointer to the remoteproc instance
`vdev_id`	Virtio device ID
`role`	Virtio device role
`rst_cb`	Virtio device reset callback

Returns

Return pointer to the created virtio device, NULL for failure.

remoteproc_remove_virtio

Description

This function removes virtio device.

Usage

```
void remoteproc_remove_virtio(struct remoteproc *rproc,
           struct virtio_device *vdev);
```

Arguments

`rproc` - Pointer to the remote instance.

`vdev` - Pointer to the virtio device.

Returns

No return.

remoteproc_get_notification

Description

This function notifies remoteproc and will check its subdevices for the notification.

Usage

```
int remoteproc_get_notification(struct remoteproc *rproc,
       uint32_t notifyid);
```

Arguments

`rproc` - Pointer to the remote instance.

`notifyid` - Notification id.

Returns

Return 0 for succeed, negative value for failure

RPMsg Development

Introduction

The RPMsg 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` and `rpmsg_virtio.h` header files.

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

RPMsg API Functions

rpmsg_send_offchannel_raw()

Description

Sends a message across to the remote processor specifying source and destination address. This function sends data of length `len` to the remote dst address from the source src address. The message will be sent to the remote processor which the channel belongs to.

In case there are no `TX` buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, `-ERESTARTSYS` is returned.

Usage

```
int rpmsg_send_offchannel_raw(struct rpmsg_endpoint *ept, uint32_t src,
           uint32_t dst, const void *data, int size,
           int wait)
```

Arguments

`ept`	The `RPMsg` endpoint
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or negative error value on failure.

rpmsg_send()

Description

Send a message across to the remote processor. This function sends data of length `len` based on the ept. The message will be sent to the remote processor which the channel belongs to, using ept's source and destination addresses.

In case there are no `TX` buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, `-ERESTARTSYS` is returned.

Usage

```
static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
          int len)
```

Arguments

`ept`	The `RPMsg` endpoint
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or negative error value on failure.

rpmsg_sendto()

Description

Send a message across to the remote processor. This function sends data of length `len` based on the ept. The message will be sent to the remote processor which the channel belongs to, using ept's source and destination addresses.

In case there are no `TX` buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, `-ERESTARTSYS` is returned.

Usage

```
static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
          int len)
```

Arguments

`ept`	The `RPMsg` endpoint
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or negative error value on failure.

rpmsg_send_offchannel()

Description

Send a message using explicit src/dst addresses. This function sends data of length `len` to the remote dst address, and uses src as the source address. The message will be sent to the remote processor which the ept channel belongs to.

In case there are no `TX` buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, `-ERESTARTSYS` is returned.

Usage

```
static inline int rpmsg_send_offchannel(struct rpmsg_endpoint *ept,
        uint32_t src, uint32_t dst,
        const void *data, int len)
```

Arguments

`ept`	The `RPMsg` endpoint
`src`	Source address
`dst`	Destination address
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or negative error value on failure.

rpmsg_trysend()

Description

Send a message across to the remote processor. This function sends data of length `len` on the ept channel. The message will be sent to the remote processor which the ept channel belongs to, using ept's source and destination addresses.

In case there are no `TX` buffers available, the function will immediately return `-ENOMEM` without waiting until one becomes available.

Usage

```
static inline int rpmsg_trysend(struct rpmsg_endpoint *ept, const void *data,
       int len)
```

Arguments

`ept`	The `RPMsg` endpoint
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or negative error value on failure.

rpmsg_trysendto()

Description

Send a message across to the remote processor. This function sends data of length `len` to the remote dst address. The message will be sent to the remote processor which the eptchannel belongs to, using ept's source address.

In case there are no `TX` buffers available, the function will immediately return `-ENOMEM` without waiting until one becomes available.

Usage

```
static inline int rpmsg_trysendto(struct rpmsg_endpoint *ept, const void *data,
        int len, uint32_t dst)
```

Arguments

`ept`	The `RPMsg` endpoint
`data`	Payload of message
`len`	Length of payload
`dst`	Destination address

Returns

Returns number of bytes it has sent or the negative error value on failure.

rpmsg_trysend_offchannel()

Description

Send a message using explicit src/dst addresses. This function sends data of length `len` to the remote dst address, and uses src as the source address. The message will be sent to the remote processor which the ept channel belongs to.

In case there are no `TX` buffers available, the function will immediately return `-ENOMEM` without waiting until one becomes available.

Usage

```
static inline int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept,
           uint32_t src, uint32_t dst,
           const void *data, int len)
```

Arguments

`ept`	The `RPMsg` endpoint
`src`	Source address
`dst`	Destination address
`data`	Payload of message
`len`	Length of payload

Returns

Returns number of bytes it has sent or the negative error value on failure.

rpmsg_init_ept

Description

Initialize RPMsg endpoint. Initialize an RPMsg endpoint with a name, source address, remoteproc address, endpoitn callback, and destroy endpoint callback.

Usage

```
static inline void rpmsg_init_ept(struct rpmsg_endpoint *ept,
         const char *name,
         uint32_t src, uint32_t dest,
         rpmsg_ept_cb cb,
         rpmsg_ns_unbind_cb ns_unbind_cb)
```

Arguments

`ept`	Pointer to `RPMsg` endpoint
`name`	Service name associated to the endpoint
`src`	Local address of the endpoint
`dest`	Target address of the endpoint
`cb`	Endpoint callback
`ns_unbind_cb`	End point service unbind callback, called when remote ept is destroyed.

rpmsg_create_ept

Description

Create RPMsg endpoint and register it to RPMsg device. Create a RPMsg endpoint, initialize it with a name, source address, remoteproc address, endpoitn callback, and destroy

endpoint callback, and register it to the RPMsg device. In essence, an RPMsg endpoint represents a listener on the RPMsg bus, as it binds an RPMsg address with an rx callback handler.

RPMsg client should create an endpoint to discuss with remote. RPMsg client provides at least a channel name, a callback for message notification and by default endpoint source address should be set to `RPMSG_ADDR_ANY`.

As an option Some RPMsg clients can specify an endpoint with a specific source address.

Usage

```
int rpmsg_create_ept(struct rpmsg_endpoint *ept, struct rpmsg_device *rdev,
        const char *name, uint32_t src, uint32_t dest,
        rpmsg_ept_cb cb, rpmsg_ns_unbind_cb ns_unbind_cb)
```

Arguments

`ept`	Pointer to `RPMsg` endpoint
`name`	Service name associated to the endpoint
`src`	Local address of the endpoint
`dest`	Target address of the endpoint
`cb`	Endpoint callback
`ns_unbind_cb`	End point service unbind callback, called when remote ept is destroyed.

rpmsg_destroy_ept

Description

Destroy RPMsg endpoint and unregister it from the RPMsg device. It unregisters the RPMsg endpoint from the RPMsg device and calls the destroy endpoint callback if it is provided.

Usage

```
void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);
```

Arguments

`ept` - Pointer to the RPMsg endpoint.

is_rpmsg_ept_ready

Description

Check if the RPMsg endpoint ready to send.

Usage

```
static inline unsigned int is_rpmsg_ept_ready(struct rpmsg_endpoint *ept)
```

Arguments

`ept` - Pointer to the RPMsg endpoint.

Returns

1 if the RPMsg endpoint has both local addr and destination addr set, 0 otherwise.

rpmsg_virtio_get_buffer_size

Description

Get RPMsg virtio buffer size.

Usage

```
int rpmsg_virtio_get_buffer_size(struct rpmsg_device *rdev);
```

Arguments

`rdev` - Pointer to the RPMsg device

Returns

Next available buffer size for text, negative value for failure.

rpmsg_init_vdev

Description

Initialize RPMsg virtio device.

Master side: Initialize RPMsg virtio queues and shared buffers, the address of shm can be ANY. In this case, function will get shared memory from system shared memory pools. If the vdev has RPMsg name service feature, this API will create an name service endpoint.

Slave side: This API will not return until the driver ready is set by the master side.

Usage

```
int rpmsg_init_vdev(struct rpmsg_virtio_device *rvdev,
        struct virtio_device *vdev,
        rpmsg_ns_bind_cb ns_bind_cb,
        struct metal_io_region *shm_io,
        struct rpmsg_virtio_shm_pool *shpool);
```

Arguments

`rvdev`	Pointer to `RPMsg` virtio endpoint
`vdev`	Pointer to the virtio device
`ns_bind_cb`	Callback handler for name service announcement without local endpoints waiting to bind.
`shm_io`	Pointer to the share memory I/O region.
`shpool`	Pointer to shared memory pool. RPMsg_virtio_init_shm_pool has to be called first to fill this structure.

Returns

Status of function selection.

rpmsg_deinit_vdev

Description

Deinitialize RPMsg virtio device.

Usage

```
void rpmsg_deinit_vdev(struct rpmsg_virtio_device *rvdev);
```

Arguments

`rdev` - Pointer to the RPMsg virtio device

rpmsg_virtio_init_shm_pool

Description

Initialize default shared buffers pool RPMsg virtio has default shared buffers pool implementation. The memory assigned to this pool will be dedicated to the RPMsg virtio. This function has to be called before calling rpmsg_init_vdev, to initialize the rpmsg_virtio_shm_pool structure.

Usage

```
void rpmsg_virtio_init_shm_pool(struct rpmsg_virtio_shm_pool *shpool,
      void *shbuf, size_t size);
```

Arguments

`shpool`	Pointer to the shared buffers pool structure
`shbuf`	Pointer to the beginning of shared buffers
`size`	Shared buffers total size

rpmsg_virtio_get_rpmsg_device

Description

This function gets the RPMsg device from RPMsg virtio device.

Usage

```
static inline struct rpmsg_device *
rpmsg_virtio_get_rpmsg_device(struct rpmsg_virtio_device *rvdev)
```

Arguments

`rdev` - Pointer to the RPMsg virtio device

Returns

RPMsg device pointed by RPMsg virtio device.

rpmsg_virtio_shm_pool_get_buffer

Description

This function gets the buffer in the shared memory pool.

RPMsg virtio has default shared buffers pool implementation. The memory assigned to this pool will be dedicated to the RPMsg virtio. If you prefer to have other shared buffers allocation, you can implement your rpmsg_virtio_shm_pool_get_buffer function.

Usage

```
metal_weak void *
rpmsg_virtio_shm_pool_get_buffer(struct rpmsg_virtio_shm_pool *shpool,
        size_t size);
```

Arguments

`shpool`	Pointer to the shared buffers pool
`size`	Shared buffers total size

Returns

Buffer pointer if free buffer is available, NULL otherwise.

XILINX

Appendix C

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® 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