OpenAMP Framework for Zynq Devices

Getting Started Guide

UG1186 (v2016.3) October 19, 2016

XILINX

ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/19/2016	2016.3	Added Chapter 4, Linux Userspace RPMsg Added Appendix C, Libmetal Introduction and Libmetal Examples Added Appendix D, Linux Userspace RPMsg Application Flow Additional technical changes and enhancements throughout document.
07/05/2016	2016.2	Changed Steps a and b to be Zynq_A9 specific. in Chapter 3, Building and Running a Linux Project with Applications. Removed extraneous 0x0 notations in Chapter 3, Building and Running a Linux Project with Applications and in Appendix A, Configuration Parameters.
05/26/2016	2016.1	Change version to match Vivado release.
05/05/2016	2.0	Changed the title to Chapter 2, Building Linux Applications and Remote Firmware. Added a note to the introduction of Chapter 2, Building Linux Applications and Remote Firmware. Changed Settings for the Device Tree Binary Source in Chapter 3. Added steps to Setting up PetaLinux with OpenAMP in Chapter 3. Modified the procedure for Setting up PetaLinux with OpenAMP in Chapter 3. Modified Running the Proxy Application in Chapter 3. Added Appendix A, Configuration Parameters. Added Appendix B, Exercise. Added document references to Appendix E, Additional Resources and Legal Notices.
11/18/2015	1.0	Initial Public Access Release.

Table of Contents

Overview

Introduction

Xilinx® 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 OpenAMP framework provides the following for both Zynq® UltraScale+™ MPSoC and Zynq-7000™ All Programmable (AP) SoC devices:

- The `remoteproc`, `RPMsg`, and `virtIO` components that are used for a Linux master or a bare-metal remote configuration.
- Proxy infrastructure and demos that showcase the ability of a proxy on a master processor running Linux on the ARM processor unit (APU) to handle `printf`, `scanf`, `open`, `close`, `read`, and `write` calls from a bare-metal OS-based remote contexts running on the remote processor unit (RPU).

Some of the advantages provided by the OpenAMP Framework for Zynq-7000 AP Soc and Zynq UltraScale+ MPSoC devices are, as follows:

- Process overviews for using the OpenAMP Framework components, with descriptions of all included functions.
- Sample implementations of using AMP across a heterogeneous system with `RPMsg`.
- Bare-metal and Linux examples to bootstrap development. Step-by-step procedures for building bare-metal and FreeRTOS applications are provided, as well as pointers to further explanatory information in the code base.
- Demonstration of using `RPMsg` communication channel implementation for a multiprocessor system-on-chip such as the Zynq UltraScale+ MPSoC device.
- FreeRTOS support for Cortex™-A9 and Cortex-R5 slaves.
- Examples and applications distributed in the Xilinx Software Development Kit (SDK), with templates to use for echo-tests, matrix multiplications, and RPC.

Software Requirements

The requirement of the current versions of PetaLinux and SDK requirements must be met.

- Petalinux must be installed
- SDK might need to be installed if you want to rebuild the remote processor firmware.

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

Components in OpenAMP

OpenAMP framework uses the following key components:

- **virtIO**: 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, and cooperates with the hypervisor. This concept is used by `RPMsg` and `remoteproc` for a processor to communicate to the remote.
- **remoteproc**: This API controls the life cycle management (LCM) of the remote processors. The `remoteproc` API that OpenAMP 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.
- **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 main Linux Kernel allows the following:

- Linux applications running on the master processor to control the LCM of a remote processor
- IPC between the master and remotes

The main Linux Kernel *does not* include source code required to support other platforms running on the remote processor (such as bare-metal or FreeRTOS applications) to communicate with a Linux master.

The OpenAMP framework provides this missing functionality by providing the infrastructure required for FreeRTOS and bare-metal environments to communicate with the Linux Kernel in AMP systems. This is possible because the OpenAMP framework builds upon the `remoteproc`, `RPMsg`, and `virtIO` functions included in the Linux Kernel.

Process Overview

It is common for the master processor in an AMP system to bring up software on the remote cores on a demand-driven basis. These cores then communicate using inter process communication (IPC). This allows the master processor to off-load work to the other processors, called *remote processors*. Such activities are coordinated and managed by the Xilinx OpenAMP software which builds upon pre-established capabilities within Linux: such as the `RPMsg`, `remoteproc`, and `virtIO` functions.

The general OpenAMP flow is as follows:

1. The Linux master configures the remote processor and shared memory is created.
2. The master boots the remote processor.
3. The remote processor calls `remoteproc_resource_init()`, which creates and initializes the `virtIO` resources and the `RPMsg` channels for the master.
4. The master receives these channels and invokes the callback channel that was created.
5. The master responds to the remote context, acknowledging the remote processor and application.
6. The remote invokes the `RPMsg` channel that was registered. The `RPMsg` channel is now established, and both sides can use the `RPMsg` calls to communicate.

To shut down the remote processor:

1. The master application sends an application-specific shutdown message to the remote application.
2. The remote application cleans up its resources and sends an acknowledgment to the master.
3. The remote calls the `remoteproc_resource_deinit()` function to free the remoteproc resources on the remote side.
4. The master shuts down the remote processor and frees the `remoteproc` on its side.

Figure 1-1 shows the process interactions.

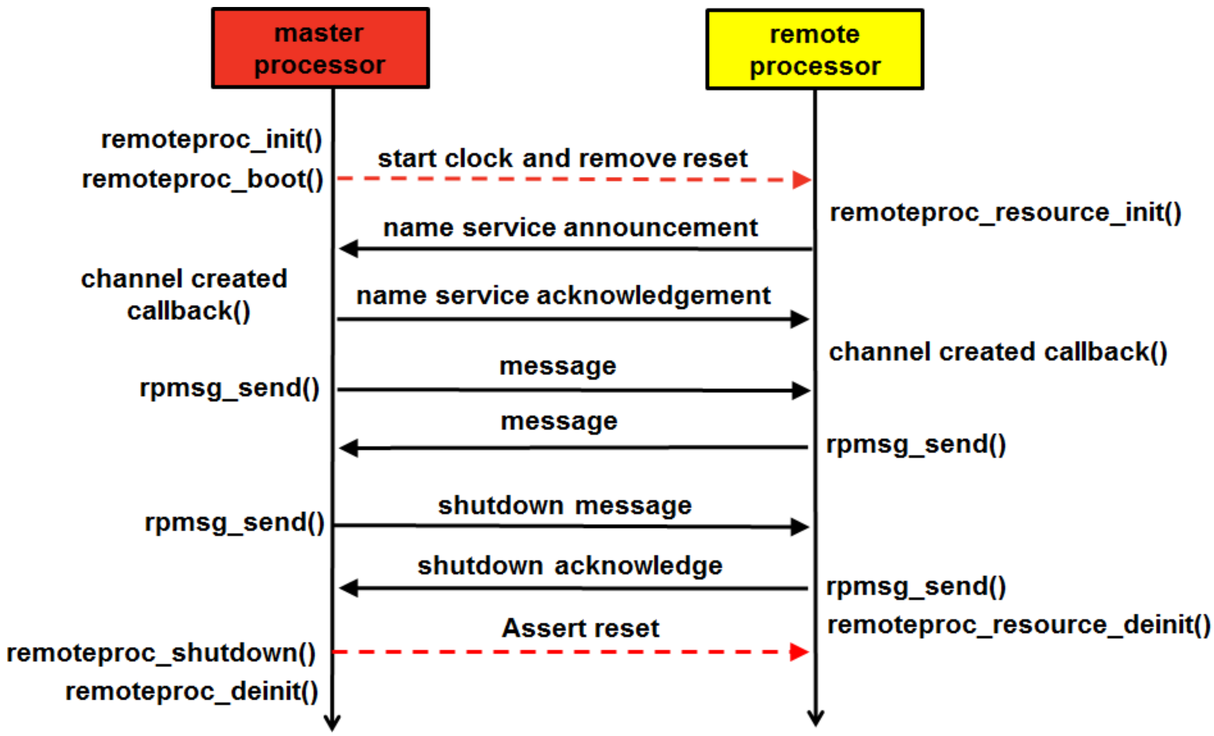

Figure 1-1: **System Sequence Diagram**

For more information, see the specific function descriptions in Chapter 5, Remoteproc Development and Chapter 6, RPMsg Development.

Building Linux Applications and Remote Firmware

Introduction

The Xilinx® software development kit (SDK) contains templates to aid in the development of OpenAMP Linux master applications, and bare-metal/FreeRTOS remote applications.

The following sections describe how to create OpenAMP applications with SDK and PetaLinux tools.

- Use SDK to create the bare-metal or FreeRTOS remote applications
- Use PetaLinux tools to create Linux user applications and Kernel user modules, build the Linux kernel, generate the device tree, and generate the `rootfs`.

Note: It is assumed here that you use the demo Linux applications already included in the PetaLinux BSP, and it is built using Petalinux. You can otherwise build your own Linux applications with SDK documentation. See the *Xilinx Software Developer Kit Help* (UG782) for more information [Ref 3].

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.

For more information on the echo test application, see the relevant source code in the PetaLinux BSP:

- Linux master (Kernel space): `components/modules/rpmsg_echo_test_kern_app/`

- Linux master (user space): `components/apps/echo_test/`
- Bare-metal remote echo test firmware: `components/apps/echo_test/data/image_echo_test`

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`. For more information on the matrix multiplication application, see the relevant source code:

- Linux master (Kernel space): `components/modules/rpmsg_mat_mul_kern_app/`
- Linux master (user space): `components/apps/mat_mul_demo/`
- Bare-metal pre-built, remote matrix multiply firmware: `components/apps/mat_mul_demo/data/image_matrix_multiply`

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. For more information on the proxy application, see the relevant source code:

- Linux master (Kernel space): `components/modules/rpmsg_proxy_dev_driver/`
- Linux master (user space): `components/apps/proxy_app/`
- Bare-metal, prebuilt remote proxy firmware: `components/apps/proxy_app/data/image_rpc_demo`

Building Remote Applications in SDK

You can build remote applications using 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 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), only Cortex-R5 (RPU) is supported.

        Select `psu_cortex5_0` or `psu_cortex5_1`.

      - For the Zynq-7000 All Programmable (AP) SoC device (zynq), only `ps7_cortexa9` 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 SDK project explorer, right-click the BSP and select **Board Support Package Settings.**
5. Navigate to the **BSP Settings** > **Overview** > **OpenAMP**. Set the **WITH_PROXY** parameter as follows:
   - For the OpenAMP RPC Demo, 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.
6. Navigate to the BSP settings drivers: **Settings > Overview > Drivers > <selected_processor>**.
7. Add any necessary parameters to the `extra_compiler_flags`:

   For the Zynq UltraScale+ MPSoC device (zynqMP):

   - When having two Cortex-R5 running concurrently in *split* mode, only one of them needs to set this parameter and it shall be the one that starts the last, add:

     `-DUSE_AMP=1`

     This parameter tells the library not to perform some shared device initialization (for example GIC) as it is already initialized by the processor that started first.

CAUTION! *Do not set this parameter when the two Cortex-R5 are running in lockstep mode, or if only one of the Cortex-R5 is running (as is the case when running in split mode with only one processor up and running).*

   For the Zynq-7000 All Programmable (AP) SoC device (`zynq`):

   - 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. Click the **OK** button.

OpenAMP SDK Key Source Files

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

- **Platform Info** (`platform_info.c` and `platform_info.h`): These files contain hard-coded, platform-specific values used to get necessary information for OpenAMP.
    - `#define VRING1_IPI_INTR_VECT` or `IPI_IRQ_VECT_ID`: This is the inter-processor interrupt (IPI) vector for the remote processor.
    - `struct hil_proc proc_table` (Array): This array provides definition of CPU nodes for master and remote context. It is intended for use with both master and remote configurations.
- **Resource Table** `(rsc_table.c/.h)`: The resource table contains entries that specify the memory and `virtIO` device resources including the firmware ELF start address and size. The `virtIO` device contains device features, `vring` addresses, size, and alignment information. The resource table entries are specified in `rsc_table.c` and the `remote_resource_table` structure is specified in `rsc_table.h`.
- **Helper** `(helper.c/.h` and `sys_init.c)`: They contain platform-specific APIs that allow the remote application to communicate with the hardware. They include functions to initialize and control the GIC.

Chapter 3

Building and Running a Linux Project with Applications

Introduction

This chapter describes how to perform the following:

- Setting up PetaLinux with OpenAMP
- Settings for the Device Tree Binary Source
- Building the Applications and the Linux Project
- Booting the PetaLinux Project
- Running the Example Application

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>`:

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

2. Navigate to the `<plnx_proj>` directory:

   ```
   cd <plnx_proj>
   ```

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 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 SDK for the remote processor into this directory:

```
components/apps/<app_name>/data
```

c. Modify the `..components/apps/<app_name>/Makefile` to install the remote processor firmware in the `RootFS`. for example:

```
install:
$(TARGETINST) -d -p 755 data/<myfirmware> /lib/firmware/<myfirmware>
```

TIP: *If you want to try one of the demonstration applications, you can replace the existing firmware at: <master_root>`components/apps/<echo_test|mat_mul_demo|proxy_app>/data/`.*

---

4. For the Zynq®-7000 AP SoC (`zynq`) device only, do step a and step b:

   a. Set the kernel base address. Because bare-metal and RTOS boot support is from address `0`; consequently, you must set the location for Linux to a higher address:

   - Run `petalinux-config`, and set the kernel base address to `0x10000000`, as follows:

   ```
   Subsystem AUTO Hardware Settings --->
     Memory Settings --->
       (0x10000000) kernel base address
   ```

   b. If you have configured using PetaLinux U-Boot `autoconfig`, set the memory address into which the U-Boot loads the Kernel.

   - Run `petalinux-config`:

   ```
   u-boot Configuration --->
     (0x11000000) netboot offset
   ```

5. 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
```

e. For the Zynq-7000 All Programmable (AP) SoC (`Zynq`) only, set memory split to 2G/2G (or use 1G/3G user/kernel):

```
Kernel Features--->
   Memory split (...)--->
    (x) 2G/2G user/kernel split
```

f. For Zynq-7000 All Programmable (AP) SoC (`Zynq`) only, enable High Memory support:

```
Kernel Features--->
    [*] High Memory Support--->
```

6. 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 are enabled:

```
Apps --->
     [*] echo_test --->
     [*] mat_mul_demo --->
     [*] proxy_app --->
```

c. Ensure the OpenAMP modules are enabled:

```
Modules --->
     [*] rpmsg_proxy_dev_driver --->
     [*] rpmsg_user_dev_driver --->
```

Settings for the Device Tree Binary Source

The PetaLinux reference BSP includes a Device Tree Binary (DTB) for OpenAMP located at:

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

This is built from the Device Tree Source (DTS), in the reference PetaLinux BSP, which is located at:

```
subsystems/linux/configs/device-tree/openamp.dts
```

This file is the same as the standard `system-top.dts`, except it has the following line incorporated:

```
/include/ "openamp-overlay.dtsi"
```

This includes the DTS overlay which is in the PetaLinux BSP, located at:

```
subsystems/linux/configs/device-tree/openamp-overlay.dtsi
```

The `openamp.dtb` and `dts` files are provided for reference only. You need to edit the `system-top.dts` file and include `openamp.dtsi` for your project.

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

- For `ZynqMP` running Linux on Cortex™-A53 and communicating with Cortex-R5:

```
{
        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;
                rproc_0_reserved: rproc@3ed000000 {
                        no-map;
                        reg = <0x0 0x3ed00000 0x0 0x1000000>;
                };
        };

        amba {
                test_r50: zynqmp_r5_rproc@0 {
                        compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
                        reg = <0x0 0xff340000 0x0 0x100>,
                          <0x0 0xff9a0000 0x0 0x400>, <0x0 0xff5e0000 0x0 0x400>;
                        reg-names = "ipi", "rpu_base", "rpu_base";
                        core_conf = "split0";
                        interrupt-parent = <&sic>;
                        interrupts = <0 29 4>;
               } ;
        };

};
```

- For `Zynq_A9`:

```
{
        amba {
                remoteproc0: remoteproc@0 {
                        compatible = "xlnx,zynq_remoteproc";
                        reg = < 0x00000000 0x10000000 >;
                        firmware = "firmware";
                        vring0 = <15>;
                        vring1 = <14>;
                };
        };
};
```

In particular for `ZynqMP`, you might want to 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, Configuration Parameters gives a more detailed explanation of those parameters.

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 `image/linux` folder:
`<plnx_proj>/images/linux`

Booting the PetaLinux Project

You can boot the PetaLinux project from QEMU or hardware.

Booting on QEMU

After a successful build, 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`

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. Program the relevant bitstreams to the board. Ensure that it is using RTL v5.2; this must be done separately from PetaLinux.
3. If the board is connected to a remote system, start the `hw_server` on the remote system.
4. Open a console terminal and connect it to UART on the board.

Downloading the Images

1. Navigate to the PetaLinux directory:

   ```
   cd <plnx_proj>
   ```

2. Run the PetaLinux boot:

   - Using a remote system:

     ```
     petalinux-boot --jtag --kernel --hw_server-url <remote_system>
     ```

   - Using a local system:

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

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

1. Load the Echo test firmware and driver. This loads the remoteproc and `RPMsg` modules:

   - For the Zynq UltraScale+™ MPSoC device (`ZynqMP_R5`):

     ```
     modprobe zynqmp_r5_remoteproc firmware=image_echo_test
     modprobe rpmsg_user_dev_driver
     ```

   - For the Zynq-7000 All Programmable (AP) SoC device (`Zynq_A9`):

     ```
     modprobe zynq_remoteproc firmware=image_echo_test
     modprobe rpmsg_user_dev_driver
     ```

2. Run the test:

   ```
   echo_test
   ```

3. The test starts, follow the on-screen instructions to complete the test.

4. After you have completed the test, unload the application:
   - For the Zynq UltraScale+ MPSoC device (`ZynqMP_R5`):

     ```
     modprobe -r rpmsg_user_dev_driver
     modprobe -r zynqmp_r5_remoteproc
     ```
   - For the Zynq-7000 All Programmable (AP) SoC device (`Zynq_A9`):

     ```
     modprobe -r rpmsg_user_dev_driver
     modprobe -r zynq_remoteproc
     ```

IMPORTANT: *After you have exited the application, you must unload and re-load the module if you want to re-run the test.*

Running the Matrix Multiplication Test

1. Load the Matrix Multiply application. This loads the `remoteproc`, `RPMsg` modules, and applications.
   - For the Zynq UltraScale+ MPSoC device (`ZynqMP_R5`):

     ```
     modprobe zynqmp_r5_remoteproc firmware=image_matrix_multiply
     modprobe rpmsg_user_dev_driver
     ```
   - For the Zynq-7000 All Programmable (AP) MPSoC device (`Zynq_A9`):

     ```
     modprobe zynq_remoteproc firmware=image_matrix_multiply
     modprobe rpmsg_user_dev_driver
     ```
2. Run the test:

   ```
   mat_mul_demo
   ```

   The test starts.
3. Follow the on screen instructions to complete the test.
4. After you have completed the test, unload the application:
   - For the Zynq UltraScale+ MPSoC device (`ZynqMP_R5`):

     ```
     modprobe -r zynqmp_r5_remoteproc
     ```
   - For the Zynq-7000 All Programmable (AP) MPSoC device (`Zynq_A9`):

     ```
     modprobe -r rpmsg_user_dev_driver
     modprobe -r zynq_remoteproc
     ```

IMPORTANT: *After you have exited the application, you must unload and re-load the module if you want to re-run the test.*

Running the Proxy Application

1. Load and run the proxy application in one step. The proxy application automatically loads the required modules:
   - For the Zynq UltraScale+ MPSoC device (`ZynqMP_R5`):

     ```
     proxy_app -m zynqmp_r5_remoteproc
     ```
   - For the Zynq-7000 All Programmable (AP) SoC device (`Zynq_A9`):

     ```
     proxy_app -m zynq_remoteproc
     ```
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.

Linux Userspace RPMsg

Linux Userspace RPMsg Overview

The OpenAMP library depends on libmetal library. With the use of libmetal library, OpenAMP is able to access the IPI device and shared memory from the Linux userspace and, therefore, OpenAMP can enable RPMsg in the Linux userspace.

For more information about the libmetal library, see Appendix C, Libmetal Introduction and Libmetal Examples.

To try the RPMsg in the Linux userspace, follow the example in this chapter.

Linux Userspace RPMsg Example

Note: This RPMsg in Linux userspace example only supports Zynq® UltraScale+™ MPSoC devices.

You can boot RPU independently with the RPMsg in Linux userspace implementation. You can also reuse the OpenAMP RPU applications created in Building Remote Applications in SDK for your RPU firmware.

The following sections provide the steps to build the RPMsg Linux userspace applications.

Build Linux Userspace RPMsg Demo Application 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. Enable the required `rootfs` packages and applications:

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

4. Ensure `open-amp`, `libmetal`, and `sysfs` packages are enabled

```
Filesystem Packages--->
   Base --->
      Sysfsutils--->
         [*] libsysfs2
   Libs --->
      libmetal--->
         [*] libmetal
      open-amp--->
         [*] open-amp
```

5. Ensure the RPMsg demo application is enabled:

```
Apps --->
   [*] echo_test --->
   [*] mat_mul_demo --->
   [*] proxy_app --->
```

Setting Device Tree for the Linux Userspace RPMsg Application Demo

The libmetal Linux demo uses UIO devices for IPI and shared memory. Copy the following to the `subsystems/linux/configs/device-tree/system-top.dts` 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 {
  /* UIO device node for vring device memory */
  vring: vring@0 {
                        compatible = "vring_uio";
                        reg = <0x0 0x3ed40000 0x0 0x40000>;
                };
/* UIO device node for shared memory device memory */
                shm0: shm@0 {
                        compatible = "shm_uio";
                        reg = <0x0 0x3ed80000 0x0 0x80000>;
                };
  /* UIO device node for IPI device */
                ipi0: ipi@0 {
                        compatible = "ipi_uio";
                        reg = <0x0 0xff340000 0x0 0x1000>;
```

```
                    interrupt-parent = <&gic>;
                    interrupts = <0 29 4>;
            };
        };
    };
```

Before you build the application, you can review the source code in the `<plnx_proj>/components/apps` directory if you have created your project from the PetaLinux Zynq UltraScale+ MPSoC board reference BSP. This list shows the directories for the OpenAMP source code:

- `<plnx_proj>/components/apps/echo_test/open-amp`
- `<plnx_proj>/components/apps/mat_mul_demo/open-amp`
- `<plnx_proj>/components/apps/proxy_app/open-amp`

For more information on how to write an PMsg Linux userspace application, see Appendix D, Linux Userspace RPMsg Application Flow.

Build the Linux Demo Application and the Linux Project

1. Go to the PetaLinux 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>/images/linux` directory.

Testing on Hardware

1. Go to your 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 o see the textual output.

4. Boot the RPU firmware built with Xilinx® SDK with `xsdb` command:

```
$ xsdb
xsdb% connect
xsdb% ta 7 # this is the RPU0 target.
    # you can use "ta" to see all the targets and which you have connected to.
xsdb% rst -processor # reset the connected RPU target
xsdb% dow <the RPU OpenAMP demo ELF image built with Xilinx SDK>
xsdb% run # This will start the RPU
```

   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 applications "`echo_test-openamp`", "`mat_mul_demo-openamp`", "`proxy_app-openamp.`". This process produces output similar to the following:

```
# echo_test-openamp
...
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-ZCU102-2016_3:~#

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

# proxy_app-openamp
login[1900]: root login on 'ttyPS0'
root@Xilinx-ZCU102-2016_3:~# proxy_app-openamp
metal: warning:   skipped page size 2097152 - invalid args
metal: info:      metal_uio_dev_open: No IRQ for device 3ed80000.shm.
metal: info:      metal_uio_dev_open: No IRQ for device 3ed40000.vring.
metal: info:      metal_uio_dev_open: No IRQ for device 3ed40000.vring.
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-ZCU102-2016_3:~#
```

Remoteproc Development

Introduction

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

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

remoteproc API Functions

remoteproc_resource_init

Description

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

Usage

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

Arguments

`rsc_info`	Pointer to resource table info control block.
`pdata`	Platform data for remote processor
`channel_created`	Callback function for channel creation
`channel_destroyed`	Callback function for channel deletion.
`rdefault_cb`	Default callback for channel I/O.
`rproc_handle`	Pointer to new remoteproc instance
`rpmsg_role`	- 1 for rpmsg master, or 0 for rpmsg slave

Returns

Status of execution.

remoteproc_resource_deinit

Description

Uninitialized resources for `remoteproc` remote configuration.

Usage

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

Arguments

`rproc` - pointer to `remoteproc` instance.

Returns

Status of execution.

remoteproc_shutdown

Description

This function shutdowns the remote execution context.

Usage

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

Arguments

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

Returns

Status of function execution.

RPMsg Development

Introduction

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

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

RPMsg API Functions

rpmsg_sendto

Description

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

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

Usage

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

Arguments

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

Returns

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

rpmsg_send

Description

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

Usage

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

Arguments

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

Returns

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

rpmsg_send_offchannel

Description

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

Usage

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

Arguments

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

Returns

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

rpmsg_trysend

Description

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

Usage

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

Arguments

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

Returns

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

rpmsg_trysendto

Description

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

Usage

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

Arguments

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

Returns

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

rpmsg_trysend_offchannel

Description

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

Usage

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

Arguments

`rpdev`	The `RPMsg` channel.
`src`	Source address.
`dst`	Destination address.
`data`	Payload of message.
`len`	Length of payload.

Returns

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

rpmsg_init

Description

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

Usage

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

Arguments

`param dev_id`	The `RPMsg` remote device associated with the driver to be initialized.
`@param rdev`	Source address.
`@param channel_created`	Destination address.
`@param channel_destroyed`	Callback function for channel deletion.
`@default_cb`	Payload of message.
`@param role`	Length of payload.

Returns

Status of function execution.

rpmsg_deinit

Description

Releases the `rpmsg` driver resources for a given remote instance.

Usage

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

Arguments

`rdev`: Pointer to device de-initialize.

Returns

None.

rpmsg_get_buffer_size

Description

Returns buffer size available for sending messages.

Usage

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

Arguments

`Channel`: Pointer to the `rpmsg` channel or device.

Returns

Buffer size.

rpmsg_create_channel

Description

Creates `rpmsg` channel with the given name for remote device.

Appendix A

Configuration Parameters

Introduction

This appendix lists the configuration parameters that are verified to work.

- `Zynq-A9`:

  Cortex™-A9 #0 running Linux and Cortex-A9 #1 remote running demo applications on Standalone or FreeRTOS.

- `ZynqMP`:

  Cortex-A53s running Linux and Cortex-R5s as remote(s) running demo applications on Standalone or FreeRTOS in one of the following configurations:

  a. Cortex-R5 in lockstep mode.

  b. Cortex-R5 in split mode with either:

     - Cortex-R5 #0 remote and Cortex-R5 #1 not running
     - Cortex-R5 #1 remote and Cortex-R5 #0 not running
     - Cortex-R5 #0 and Cortex-R5 #1 as remotes running concurrently and independently, each with its own channel to separate applications on A53.

The following parameters are the ones you need to inspect and/or modify for your design.

Check the Wiki: *OpenAMP* [Ref 1] where more detailed information could be provided.

DTS configuration for OpenAMP

File location

```
<petalinux project directory>/subsystems/linux/configs
/device-tree/openamp-overlay.dtsi
```

General Information

General information on DTS file format can be found by searching online for the specification.

For Zynq UltraScale+ MPSoC Device using Cortex-R5

The `reserved-memory` section below defines which part of the memory visible to Cortex-A53 can be reserved for Cortex-R5 firmware use. The current address below points to DDR location.

The `zynqmp_r5_rproc` section defines:

- `reg` and `reg-names`: Provide a map of where the registers for the inter-processor interrupts (IPI), (RPU), and (ABP) blocks are located in the chip. For example, the IPI registers below are located starting at address `0xff340000`. For more information on registers definition and addresses, see the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 2].
- `interrupts`: interrupt number used by OpenAMP.
- `core_conf`: Provides the mode of operation for Cortex-R5. Values are:
    - `split0=cortex-R5 #0`
    - `split1=cortex-R5 #1,`
    - `lockstep`

Code Example

```
{
        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;
                rproc_0_reserved: rproc@3ed000000 {
                        no-map;
                        reg = <0x0 0x3ed00000 0x0 0x1000000>;
                };
        };

        amba {
                test_r50: zynqmp_r5_rproc@0 {
                        compatible = "xlnx,zynqmp-r5-remoteproc-1.0";

                      reg = <0x0 0xff340000 0x0 0x100>, <0x0 0xff9a0000 0x0 0x400>,
<0x00xff5e0000 0x0 0x400>;
                        reg-names = "ipi", "rpu_base", "apb_base";
                        core_conf = "split0";
                        interrupt-parent = <&gic>;
                        interrupts = <0 29 4>;
                } ;
        };
};
```

For Zynq-7000 AP SoC Device using Cortex-A9

- `reg`: memory range and size used by the firmware.
- `vring0` and `vring1`: two separate interrupts used for signaling between the CPU cores.

Code Example

```
{
        amba {
                remoteproc0: remoteproc@0 {
                        compatible = "xlnx,zynq_remoteproc";
                        reg = < 0x00000000 0x10000000 >;
                        firmware = "firmware";
                        vring0 = <15>;
                        vring1 = <14>;
                };
        };
};
```

Linux RPMsg Buffer Size

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

While you might be interested in redefining this, resizing the RPMsg size and its effects has not been verified.

In addition to changing the `rpmsg` kernel module, you would need to change your user driver module (for example: the `rpmsg_user_dev_driver` in the provided examples), as well as the OpenAMP library.

Application Resource Table and Linker Script Files

The demo applications use three files (`rsc_table.c`, `rsc_table.h`, and `lscript.ld`) to define the memory usage for OpenAMP. The *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 2] provides detailed information on the different type of memory accessible.

The `resource_table` contained in the `rsc_table.c` file defines the memory regions shared between the remote processor and the remoteproc driver running on Linux. This one extracts the resource table from the generated ELF file for the remote processor.

You could, for example, add or remove `carveout` sections, in which case you would change the `CARVEOUT_SRC` and `CARVEOUT_SRC_OFFSETS` as well as the `NUM_TABLE_ENTRIES` in the `rsc_table.c` file, and the `remote_resource_table` structure in the `rsc_table.h` file.

Each `CARVEOUT_SRC` entry contains a start address and a length that needs to be defined based your application need.

Note: `Carveout` is defined in the Linux Kernel `remoteproc` documentation as "physically contiguous memory regions.

The `lscript.ld` is for the linker use, and defines the memory usage for the R5 application as for any other applications.

Compilation Flags

The following parameters can be provided to the toolchain via the extra compiler flags.

You can access the **extra_compiler_flag** field in the Xilinx® SDK BSP for your application.

See the *SDK Help* [Ref 3] for more information.

For the Zynq®-7000 All Programmable (AP) SoC device (`zynq`):

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

```
-DUSE_AMP=1
```

b. To allow OpenAMP to redirect `_open()`, `_close()`, `_read()`, and `_write()`, add

```
-DUNDEFINE_FILE_OPS
```

This parameter is used when the OpenAMP library is linked with the `rpmsg_retarget.o` file. This can be enabled or disabled when creating the application BSP in the Xilinx SDK, and setting the **WITH_PROXY** option in the OpenAMP section to either **True** or **False**.

Note: You do not need to set this flag when using Xilinx SDK. It is automatically set when changing the WITH_PROXY parameter.

For the Zynq UltraScale+™ MPSoC device (`zynqMP`):

a. When having two Cortex-R5 running concurrently in split mode, only one of them needs to set this parameter, and it shall be the one that start the last, add:

```
-DUSE_AMP=1
```

This parameters tells the library not to perform some shared device initialization (for example: GIC) as it is already initialized by the processor that started first.

IMPORTANT: *Do not set this parameter when the two Cortex-R5 run in lockstep mode, or if only one of the Cortex-R5 is running (such as in split mode with only one processor up and running).*

b. To allow OpenAMP to redirect `_open()`, `_close()`, `_read()`, and `_write()`, add:

```
-DUNDEFINE_FILE_OPS.
```

This parameter is used when the OpenAMP library is linked with the `rpmsg_retarget.o` file. This can be enabled or disabled when creating the application BSP in the Xilinx SDK and setting the **WITH_PROXY** option in the OpenAMP section to either **True** or **False**.

Note: You do not need to set this flag when using Xilinx SDK. It is automatically set when changing the WITH_PROXY parameter.

c. To force the vector table location in OCM (instead of TCM), add:

```
-DVEC_TABLE_IN_OCM
```

Changing the RPMsg Channel ID

Changing the RPMsg ID might be required if you need to create multiple OpenAMP slaves, because the messages carry an individual identifier associated to each channel.

To change the RPMsg ID:

1. Modify the `rpmsg_user_dev_driver, LKM`, by changing the string `‘.name’` in the structure `rpmsg_user_dev_drv_id_table`, so that it is a unique identifier for this channel.
2. Modify user application `platform_info.c` file by changing the channel name in this file.

Exercise

ZynqMP Two Cortex-R5 Running Concurrently

ZynqMP Cortex™-A53 running one Linux application connected to one Cortex-R5 in split mode and another application connected to the other Cortex-R5. For simplicity, use the pre-existing `echo_test` demo application.

In this example, Cortex R5 #0 boots first, followed by Cortex-R5 #1. This order is important here because Cortex R5 #0 needs to first initialize the interrupt controller shared by both cores.

The following steps are what you need to change:

1. Modify the `rpmsg_user_dev_driver`, `LKM`:
    a. Change directories to the petalinux project:

    ```
    cd <petalinux project directory>
    ```

    b. Make a copy of the driver code and create a new instance (see the *PetaLinux Tools Reference Guide* (UG1144) [Ref 4]).

    ```
    petalinux-create -t modules --name rpmsg_user_dev_driver_r5_1 --enable
    cd <petalinux project directory>/components/modules/rpms_user_dev_driver_r5_1
    cp ../rpmsg_user_dev_driver/rpmsg_user_dev_driver
    . /rpmsg_user_dev_driver_r5_1.c
    ```

    c. Edit `rpmsg_user_dev_driver_r5_1.c` file, and change the `rpmsg_user_dev_drv` structure, so that the string, '`.drv.name`', is unique to this driver.

    d. Change the channel name to be unique. See Appendix A, Configuration Parameters for more information.

    e. Change the device name in `device_create()` to be unique (will show in `/dev/`...)

    Note: The `echo_test` demo application can take the following as a argument: `-d /dev/<your device name>`, that it uses it when calling `open()`.

f. Inside `init()`, change `class_create()` and `alloc_chrdev_region()` string parameter `rpmsg_user_dev` to `rpmsg_user_dev_r5_1`.

g. Build the driver, and add it to `rootfs`.

```
petalinux-build
```

2. Use SDK to create two echo-test remote firmware applications as explained in this document: One to run on Cortex-R5 #0, and one to run on Cortex-R5 #1.

3. Modify the Cortex-R5-1 remote firmware application in SDK:

   a. Edit `platform_info.c` and change the channel name to match the one in the `rpmsg_user_dev_driver` above. Also replace `0xff310000` with `0xff320000` in the two VRING descriptors.

   b. Edit `sys_init.c` to replace `IPI_BASEADDR` value `0xff310000` with `0xff320000`. Also update the IPI_DEV_NAME to `ff320000.ipi`.

   c. Edit `platform_info.h`, and search and replace `IPI_IRQ_VECT_ID` value from `65` to `66`.

   d. Edit the linker script file, `lscript.ld`, to avoid memory conflict with other remote processors. For example, to increase DDR start address.

   e. Edit the `carveout` sections in `rsc_table.c` for both applications to match the linker script so that they do not conflict.

   f. Add to this application BSP the extra compiler flag -DUSE_AMP=1

4. Add the necessary entry to your DTS file for each Cortex-R5 in split mode:

```
amba {
        test_r50: zynqmp_r5_rproc0@0 {
          compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
          reg = <0x0 0xff340000 0x0 0x100>, <0x0 0xff9a0000 0x0 0x400>,
          <0x0 0xff5e0000 0x0 0x400>;
          reg-names = "ipi", "rpu_base", "apb_base";
          core_conf = "split0";
          interrupt-parent = <&gic>;
          interrupts = <0 29 4>;
     } ;
        test_r51: zynqmp_r5_rproc1@1 {
            compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
            reg = <0x0 0xff340000 0x0 0x100>, 0xff9a0000 0x0 0x400>,
            <0x0 0xff5e0000 0x0 0x400>;reg-names = "ipi", "rpu_base", "apb_base";
            core_conf = "split1";
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
                    } ;
      };
```

5. Run the demonstration applications.

   a. Connect to your target using either serial, telnet, or ssh to have two separate terminals with which to run your linux applications concurrently.

b. Load both remote firmware using the following syntax:

```
modprobe zynqmp_r5_remoteproc firmware=<Cortex R5 #0 elf file>
firmware1=<Cortex R5 #1 elf file>
```

c. Load RPMsg user device driver for Cortex R5 #0:

```
modprobe rpmsg_user_dev_driver
```

d. Load rpmsg user device driver for Cortex R5 #1:

```
modprobe rpmsg_user_dev_driver_r5_1
```

e. Start the Cortex-R5 #0 `echo_test` Linux application in one terminal:

```
echo_test
```

f. Start the Cortex-R5 #1 `echo_test` Linux application in another terminal:

```
echo_test -d /dev/<your device name>
```

Note: More details can be found on the Xilinx® Wiki: *OpenAMP* [Ref 1].

Libmetal Introduction and Libmetal Examples

Libmetal Overview

Libmetal is a library 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 currently supports Zynq®-7000 and Zynq UltraScale+™ MPSoC platforms in the following operating systems:

- Linux userspace
- FreeRTOS
- Bare-metal environments

The following architecture diagram shows how OpenAMP uses libmetal:

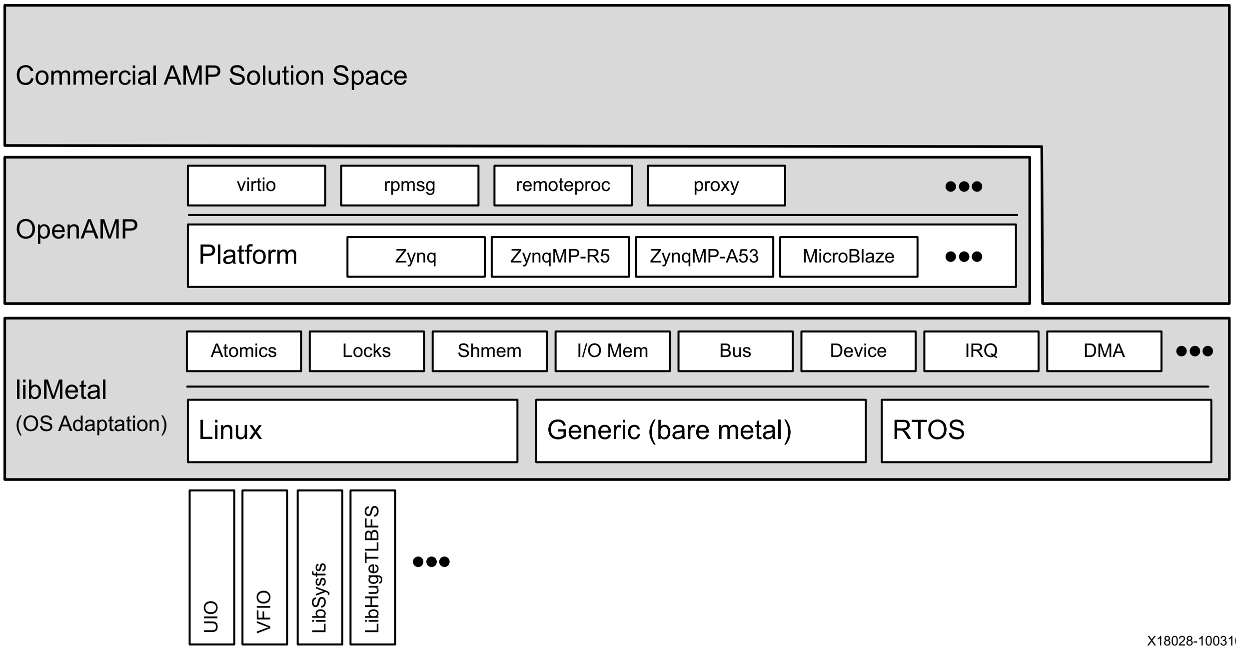

Figure C-1: **OpenAMP libmetal Architecture**

Please refer to the https://github.com/xilinx/libmetal/tree/xlnx-2016.3 for more details on libmetal APIs.

Libmetal Example

This example shows how to use libmetal to build simple inter-processor communication between APU and RPU on a Zynq UltraScale+ MPSoC platform. The example uses the following resources for the inter-processor communication:

- DDR device memory as shared memory
- IPI (inter-processor interconnect) for notification

This chapter describes how to build the libmetal example with Xilinx® SDK and PetaLinux tools.

Note: This example is for Zynq UltraScale+ MPSoC platforms only. It runs on RPU, and it only supports bare metal environments.

Build Libmetal Bare-Metal Firmware with Xilinx SDK

1. From the SDK window, create the application project by selecting **File** > **New** > **Application Projects**.
   a. Specify the BSP OS platform:
      - **standalone** for a bare-metal application.
   b. Specify the hardware platform.
   c. Select the processor:
      - Cortex™-R5 (RPU) is supported. Select **psu_cortex5_0** or **psu_cortex5_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 Echo Demo**.
   g. Click **Finish**.
   h. Before you build the application, review the source code of the generated application from the SDK project explorer. The key source files of the libmetal demo application are as follows:
      - `sys_init.c` - System initialization, such as GIC initialization, and metal device definition for IPI device and shared memory
      - `libmetal_amp_demo.c` – Demo application that illustrates how to use IPI and shared memory with libmetal for inter-processor communication.
2. To build the application project, right-click the created project and select **Build project**. The generated ELF will be in "`<RPU_app_proj>/Debug/`" directory.

Enable Linux Demo Application Using libmetal with PetaLinux Tools

Before using PetaLinux tools, follow these preparatory steps:

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

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

2. Navigate to the directory:

   ```
   $ cd <plnx_proj>
   ```

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

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

4. Ensure libmetal and sysfs packages are enabled:

   ```
   Filesystem Packages--->
      Base --->
         Sysfsutils--->
            [*] libsysfs2
      Libs --->
         libmetal--->
            [*] libmetal
   ```

5. Ensure the libmetal demo application is enabled:

   ```
   Apps --->
      [*] libmetal-demo --->
   ```

Setting Device Tree for the Libmetal Linux Application Demo

The libmetal Linux demo uses UIO devices for IPI and shared memory. Copy the following to your `subsystems/linux/configs/device-tree/system-top.dts` in your 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 descriptor (APU to RPU) */
               shm0_desc: shm_desc@0 {
                        compatible = "shm_desc_uio";
                        reg = <0x0 0x3ed00000 0x0 0x10000>;
                };
  /* Shared memory descriptor (RPU to APU) */
                shm1_desc: shm_desc@1 {
                        compatible = "shm_desc_uio";
                        reg = <0x0 0x3ed10000 0x0 0x10000>;
                };
  /* Shared memory */
                shm0: shm@0 {
                        compatible = "shm_uio";
                        reg = <0x0 0x3ed20000 0x0 0x40000>;
                };
  /* IPI device */
                ipi0: ipi@0 {
                        compatible = "ipi_uio";
                        reg = <0x0 0xff340000 0x0 0x1000>;
                        interrupt-parent = <&gic>;
                        interrupts = <0 29 4>;
                };
        };
};
```

Before you build the application, you can review the source code in your `<plnx_proj>/components/apps/libmeta-demo` directory if you have created your project from the PetaLinux Zynq UltraScale+ MPSoC board reference BSP.

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>/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 the RPU firmware built with Xilinx SDK with the `xsdb` command:

```
$ xsdb
xsdb% connect
xsdb% ta 7 # this is the RPU0 target.
                     # you can use "ta" to see all the targets and which you have
connected to.
xsdb% rst -processor # reset the connected RPU target
xsdb% dow <the RPU libmetal demo ELF image built with Xilinx SDK>
xsdb% run # This will start the RPU
```

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 `libmetal-demo`. This process produces output similar to the following:

```
# libmetal-demo

metal: warning:   skipped page size 2097152 - invalid args
metal: info:      metal_uio_dev_open: No IRQ for device 3ed00000.shm_desc.
metal: SERVER> SENDING message...
SERVER> Wait for echo test to start.
info:      metal_uio_dev_open: No IRQ for device 3ed10000.shm_desc.
metal: info:      metal_uio_dev_open: No IRQ for device 3ed20000.shm.
CLIENT> Start shm atomic testing...
CLIENT> shm atomic testing PASS!
CLIENT> Start echo flood testing....
CLIENT> It sends msgs to the remote.
CLIENT> And then it waits for msgs to echo back and verifiy.
CLIENT> Sending shutdown message...
CLIENT> Total packages: 1024, time_avg = 0s, 7721ns
```

Linux Userspace RPMsg Application Flow

The OpenAMP library provides RPMsg APIs for applications to use in sending messages to and receiving messages from another processor. It also provides a remoteproc driver that triggers the inter-processor interrupt (IPI) to notify another processor if there is a message that needs to be sent and that monitors the IPI for notifications from another processor. The provided demo polls the IPI interrupt status register to see if IPI is triggered by another processor.

The following flow diagram of an RPMsg in Linux Userspace Application. This diagram shows the RPMsg application running on APU and talking to the firmware on RPU.

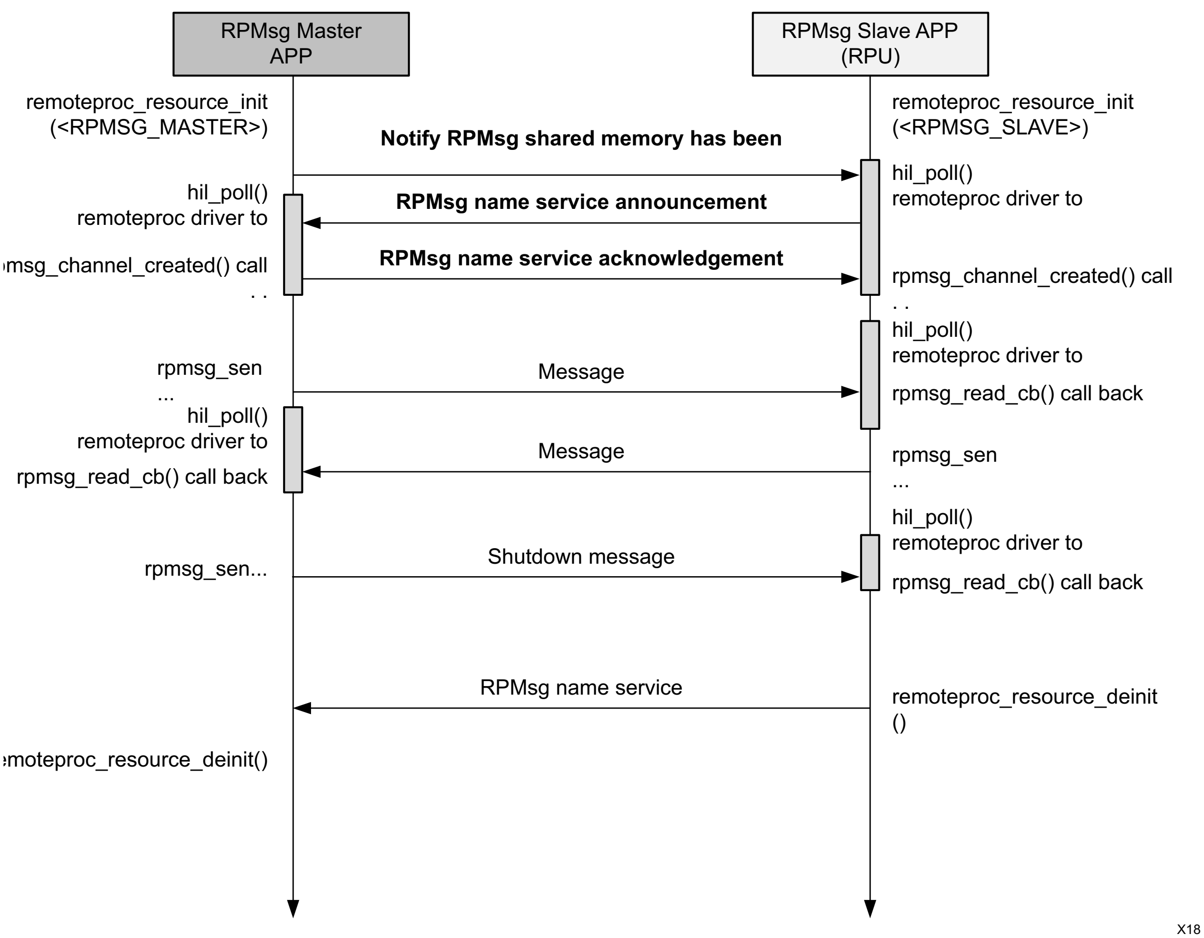

Figure D-1: RPMsg Flow Diagram in Linux Userspace Application

Linux Userspace RPMsg Application Platform Data Definition

The Linux application is required to define the resource table and the platform information data. For examples, you can refer to the `rsc_table.c` and `platform_info.c` files in the demo applications source code directory. :

- Resource table:
`<plnx_proj>/components/apps/echo_test/open-amp/rsc_table.c`
- Platform specific data:
`<plnx_proj>/components/apps/echo_test/open-amp/platform_info.c`

Resource Table

As shown in the following example, you need to define the address vrings, which contains the shared memory descriptors in the resource table.

```
{
   1, /* resource table structure version */
   1, /* number of source entries */
   {0, 0,}, /* resource table reserved fields */
   {
        /* offsets of resource entries */
        offsetof(struct remote_resource_table, rpmsg_vdev),
   },
   /* Virtio device entry */
  /* { RSC_VDEV, rpmsg_dev_id, notify_id, rpmsg_virtio_features(1 for name service),
       config_len, status, num_of_vrings, reserved } */
   {
        RSC_VDEV, 7, 0, 1, 0, 0, 0, 2, {0, 0},
   },
   /* vrings entries */
   /* { vring_phy_addr, vring_alignment, num_buffers, notify_id, reserved } */
   { 0x3ed400000, 0x1000, 256, 1, 0},
   { 0x3ed800000, 0x1000, 256, 2, 0},
};
```

Platform Data

You need to specify the IPI device, vring device, and shared memory device in the platform data definition. Each device should have a device node defined in the device tree. What you need to define depends on the remoteproc driver in the OpenAMP library. The following example is based on the RPMsg remoteproc Linux Userspace driver for Zynq® UltraScale+™ MPSoC platform.

```
struct rproc_info_plat_local proc_table = {
   {
      REMOTE_CPU_ID, /* CPU ID of master */
```

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
2. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
3. *Xilinx Software Developer Kit Help* (UG782)
4. *PetaLinux Tools Reference Guide* (UG1144)
5. Xilinx libmetal source code
6. Xilinx OpenAMP source code

Please Read: Important Legal Notices