OpenAMP Framework for Zynq Devices

Getting Started Guide

UG1186 (v1.0) December 11, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/11/2015	1.0	Initial Public Access Release.

XILINX
ALL PROGRAMMABLE™

Table of Contents

Appendix A: Additional Resources and Legal Notices

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. It this case, the remote is the client processor
- 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).

For more information on the Zynq UltraScale+ MPSoC device programming see the Zynq UltraScale+ MPSoC Software Developers Guide (UG1137) [Ref 3].

For more information on the Zynq-7000 All Programmable (AP) SoC device programming, see the Zynq-7000 All Programmable (AP) SoC Software Developers Guide (UG821) [Ref 4].

For a comprehensive description of the Zynq UltraScale hardware, see the Zynq UltraScale+ MPSoC Technical Reference Manual (UG1087) [Ref 2]

Advantages of OpenAMP FrameWork for Zynq Devices

Some of the advantages that the OpenAMP Framework for Zynq devices provides 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 for you to bootstrap your 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.
- Verification of compatibility with the Zynq UltraScale MPSoC devices (both Cortex-R5 and Cortex-A53) and with the Zynq-7000 AP SoC device, and with PetaLinux 2015.4.
- FreeRTOS support for Cortex-R5 slaves.
- Examples and applications distributed in the Xilinx Software Development Kit (XSDK), with templates to use for echo-tests, matrix multiplications, and RPC.

Software Requirements

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

Prerequisites

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

- Linux, PetaLinux, and Xilinx XSDK
- 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 from the master processor. 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 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.

In the current implementation, RPMsg defines the maximum buffer size to MAX_RPMSG_BUFF_SIZE=512 bytes including the RPMsg header size of 24 bytes.

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 a bare-metal or FreeRTOS application on a client processor) to communicate with a Linux master.

The OpenAMP framework offers 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*, which are client 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, page 8 shows the process interactions.

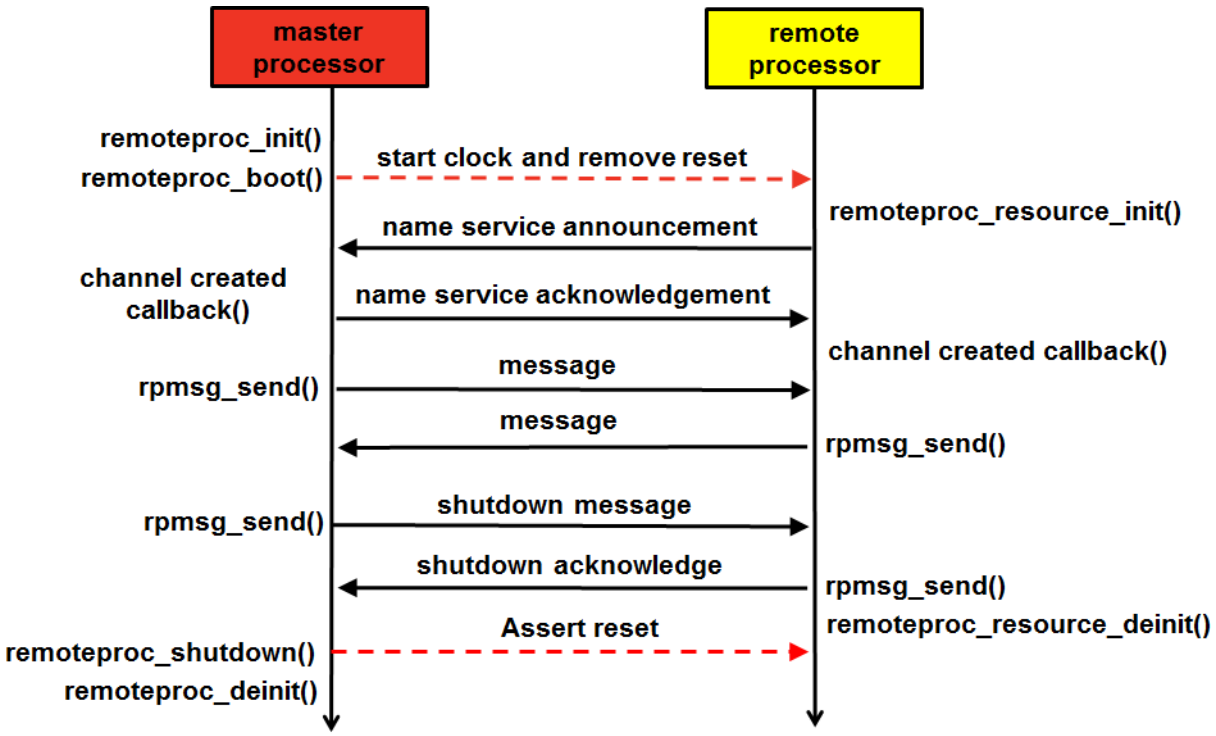

Figure 1-1: **System Sequence Diagram**

For more information, see the specific function descriptions in Chapter 4, remoteproc Development and Chapter 5, RPMsg Development.

Building OpenAMP for Bare-Metal and FreeRTOS Remote Applications

Introduction

The Xilinx® software development kit (XSDK) 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 XSDK and PetaLinux tools.

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

Echo Test in Linux Master and Bare-Metal/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.

IMPORTANT: *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/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 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 remote matrix multiply firmware: `components/apps/mat_mul_demo/data/image_matrix_multiply`

Proxy Application for Linux Masters and Bare-Metal/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 remote matrix proxy firmware: `components/apps/proxy_app/data/image_rpc_demo`

Building Remote Applications in XSDK

You can build remote applications using XSDK by using the following procedures.

Creating the Application with the OpenAMP Library

1. To create a new board support package (BSP), from the XSDK window, select **File > New > Board Support Package**.
   a. Specify a hardware file (`*.hdf`) for the device.
   b. Specify the CPU to run the remote application:
      - For the Zynq UltraScale+ MPSoC device (`zynqMP`), only RPU is supported; select `psu_cortex5_0` or `psu_cortex5_1`.
      - For the Zynq-7000® All Programmable (AP) device (`zynq`), only `ps7_cortexa9` is supported; select `ps7_cortexa9_1`.
   c. Specify the BSP OS platform:
      - `standalone` for a bare-metal application
      - `freertos<version>_xilinx` for a FreeRTOS application.

   The BSP settings open.
2. Navigate to the BSP settings overview: **Settings > Overview**
3. Select the `xilopenamp` library.
4. Navigate to the BSP settings drivers: **Settings > Overview > Drivers > `<selected_processor>`**.
5. Add `-DUSE_AMP=1` to the `extra_compiler_flags`:
   - 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.
   - For the Zynq UltraScale+ MPSoC device (`zynqMP`), when having two R5 OpenAMP slaves in split mode; only one slave will initialize the interrupt controller.
6. Add `-DUSEAMP=1` to the `extra_compiler_flags` to disable the definition of low-level read, write, and open operations. This also disables the vector table location in the tightly-coupled memory (TCM).

Creating an Application Project for OpenAMP

1. From the XSDK 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 (hardware file for the device software created in the previous procedure).
   c. Select the processor.
   d. Select either:
      - **Use Existing** to use the BSP created in the previous procedure,

      or

      - **Create New BSP** to create a new BSP.

IMPORTANT: *If you select Create New BSP, the* `xilopenamp` *library is automatically included, but the compiler flags must be set as described in step 5 and step 6 of the previous procedure.*

   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**.

Including the Remote Application in the PetaLinux Project

After you have developed and built the remote application using XSDK as described in the previous steps, the application must be included in the PetaLinux project so that the OS can load the firmware into a slave.

1. Ensure that you are in the PetaLinux project `root` directory, using the following command:

   ```
   cd <master_root>
   ```

2. 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
```

3. Copy the firmware built with XSDK into this directory, using the following command:

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

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

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

5. Rebuild the PetaLinux project.

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

OpenAMP XSDK Key Source Files

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

- **Platform Info** (`platform_info.c`): This file contains APIs and hard-coded, platform-specific values used to get necessary information for the OpenAMP. There are two core functions, a number of hard-coded values, and a platform-specific array in the file. Documentation is included in the source file.
  - `#define VRING1_IPI_INTR_VECT`: 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 contains two nodes because the same file is intended for use with both master and remote configurations. Only one node definition is required for the master/remote on the Zynq UltraScale+ MPSoC device platform as there are only two cores present in the platform.
  - `platform_get_processor_info`: This copies the target information from the user defined data structures to the HIL proc data structure. For remote contexts, this function is called with the reserved `CPU ID HIL_RSVD_CPU_ID` because is only one master for remotes.
    - **Usage**:

```
int platform_get_processor_info(struct hil_proc *proc , int cpu_id)
```

    - **Arguments**:

      `Proc: HIL` proc to populate

      `cpu_id:` CPU ID

    - **Returns**: Status of execution.

platform_get_processor_for_fw

Not used.

- **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`.
- **Bare-Metal** (`baremetal.c/.h`): The bare-metal provided for inclusion within the final standalone software configuration files contain platform-specific APIs that allow the remote application to communicate with the hardware. In this case it includes functions to initialize and control the GIC, interrupts, memory mappings, platform cache, and the inter-processor interrupt (IPI).
- **Platform** (`platform.c/.h`): These files contain the platform specific implementation of the inter-process communication (IPC) hardware layer interface. It contains platform-specific operations to control the inter-processor interrupt (IPI), CPUs, and interrupts.

Building the OpenAMP Linux Master or Bare-Metal/FreeRTOS Application

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 `<master_root>`:

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

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

   ```
   cd <master_root>
   ```

3. These steps are for the Zynq-7000 (`zynq`) device only:

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

4. 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. Set memory split to 2G/2G (or use 1G/3G user/kernel):

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

f. Enable High Memory support:

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

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 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 overlay contains nodes that OpenAMP requires in the device tree.

- For ZynqMP_R5:

```
reserved-memory {
        #address-cells = <2>;
        #size-cells = <1>;
        ranges;

        };
};
amba {
        test_r50: zynqmp_r5_rproc0@0 {
                compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
                reg = <0x0 0xff340000 0x100>, <0x0
0xff9a0000 0x400>, <0x0 0xff5e0000 0x400>;/* IPI, RPU regiters address */
                        reg-names = "ipi", "rpu_base", ""rpu_base"";
```

```
                        core_conf = "split0";/* R5 operation mode, split0, split1,
                                              /* and lock-step */
                        interrupt-parent = <&gic>;
                        interrupts = <0 29 4>;/* APU IPI interrupt ID */
        } ;
} ;
```

- For Zynq A9:

```
amba {
      remoteproc0: remoteproc@0
        {
        compatible = "xlnx,zynq_remoteproc";
        reg = < 0x00000000 0x10000000 >; /* memory reserved for the remote's
                                         /* firmware and shared memory
                                         /* between the two cores*/
        firmware = "firmware";
        vring0 = <15>; /* the soft interrupt ID for the master core */
        vring1 = <14>; /* the soft interrupt ID for the remote core */
       };
     };
```

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 <master_root>
   ```

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 will be located in the image/linux folder:

```
<master_root>/images/linux
```

Booting the PetaLinux Project

You can boot the PetaLinux project from QEMU or hardware. See the *Zynq UltraScale+ QEMU User Guide* (UG1169) [Ref 1] for more information.

Booting on QEMU

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

1. Navigate to the PetaLinux directory: `cd <master_root>`
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 <master_root>
   ```

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: Echo test, Matrix Multiplication test, and the Proxy application.

Note: In the following examples, the `zynqmp_r5_remoteproc` parameter `firmware=` is for `RPU-0` and is used with DTS `remoteproc` section `core_conf` parameter set either to `split0` or `lockstep`. When using `RPU-1`, change the parameter to `firmware1=`, and set the DTS `core_conf` parameter to use `split1`.

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
```

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_r5_user_dev_driver
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) MPSoC 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.

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,
                              rpmsg_chnl_cb_t channel_created,
                              rpmsg_chnl_cb_t channel_destroyed,
                              rpmsg_rx_cb_t default_cb,
                              struct remote_proc** rproc_handle);
```

Arguments

`rsc_info`	Pointer to resource table info control block.
`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.

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 on 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 on 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 on 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 on 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 on 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 on 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-init

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.

Usage

```
struct rpmsg_channel *rpmsg_create_channel(struct remote_device *rdev, char
*name);
```

Arguments

rdev	Pointer to the RPMsg remote device
name	Channel name

Returns

Pointer to the new RPMsg channel.

rpmsg_delete_channel

Description

Deletes the given RPMsg channel. You mush have first created the RPMsg channel using the pmsg_create_channel API.

Usage

```
void rpmsg_delete_channel(struct rpmsg_channel *rp_chnl);
```

Arguments

rp_chn: Pointer to the RPMsg channel to be deleted.

Returns

None

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.

Xilinx Documentation

1. *Zynq UltraScale+ MPSoC QEMU User Guide* (UG1169)
2. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
3. *Zynq UltraScale+ MPSoC Software Developers Guide* (UG1137)
4. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)

Please Read: Important Legal Notices

Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

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

AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, and PrimeCell are trademarks of ARM in the EU and other countries.