

# OpenAMP Framework for Zynq Devices

## *Getting Started Guide*

UG1186 (v2016.1) May 26 , 2016

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/26/2016	2016.1	Change version to match Vivado release.
05/05/2016	2.0	Changed the title to <a href="#">Chapter 2, Building Linux Applications and Remote Firmware</a> . Added a note to the introduction of <a href="#">Chapter 2, Building Linux Applications and Remote Firmware</a> . Changed <a href="#">Settings for the Device Tree Binary Source</a> in Chapter 3. Added steps to <a href="#">Setting up PetaLinux with OpenAMP</a> in Chapter 3. Modified the procedure for <a href="#">Setting up PetaLinux with OpenAMP</a> in Chapter 3. Modified <a href="#">Running the Proxy Application</a> in Chapter 3. Added <a href="#">Appendix A, Configuration Parameters</a> . Added <a href="#">Appendix B, Exercise</a> . Added document references to <a href="#">Appendix C, Additional Resources and Legal Notices</a> .
11/18/2015	1.0	Initial Public Access Release.

# Table of Contents

## Chapter 1: Overview

Introduction .....	5
Components in OpenAMP .....	6
Process Overview .....	7

## Chapter 2: Building Linux Applications and Remote Firmware

Introduction .....	9
Echo Test in Linux Master and Bare-Metal or FreeRTOS Remotes .....	9
Matrix Multiplication for Linux Master and Bare-Metal or FreeRTOS Remotes .....	10
Proxy Application for Linux Masters and Bare-Metal or FreeRTOS Remotes .....	10
Building Remote Applications in XSDK .....	11
OpenAMP XSDK Key Source Files .....	13

## Chapter 3: Building and Running a Linux Project with Applications

Introduction .....	14
Setting up Petalinux with OpenAMP .....	14
Settings for the Device Tree Binary Source .....	17
Building the Applications and the Linux Project .....	18
Booting the Petalinux Project .....	18
Running the Example Applications .....	20

## Chapter 4: Remoteproc Development

Introduction .....	22
remoteproc API Functions .....	22

## Chapter 5: RPMsg Development

Introduction .....	24
RPMsg API Functions .....	24

## Appendix A: Configuration Parameters

Introduction .....	31
--------------------	----

**Appendix B: Exercise**

ZynqMP Two Cortex-R5 Running Concurrently ..... 36

**Appendix C: Additional Resources and Legal Notices**

Xilinx Resources ..... 39  
Solution Centers ..... 39  
Xilinx Documentation ..... 39  
Please Read: Important Legal Notices ..... 40

# 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-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.

- Petalinux must be installed
- XSDK 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 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. 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.

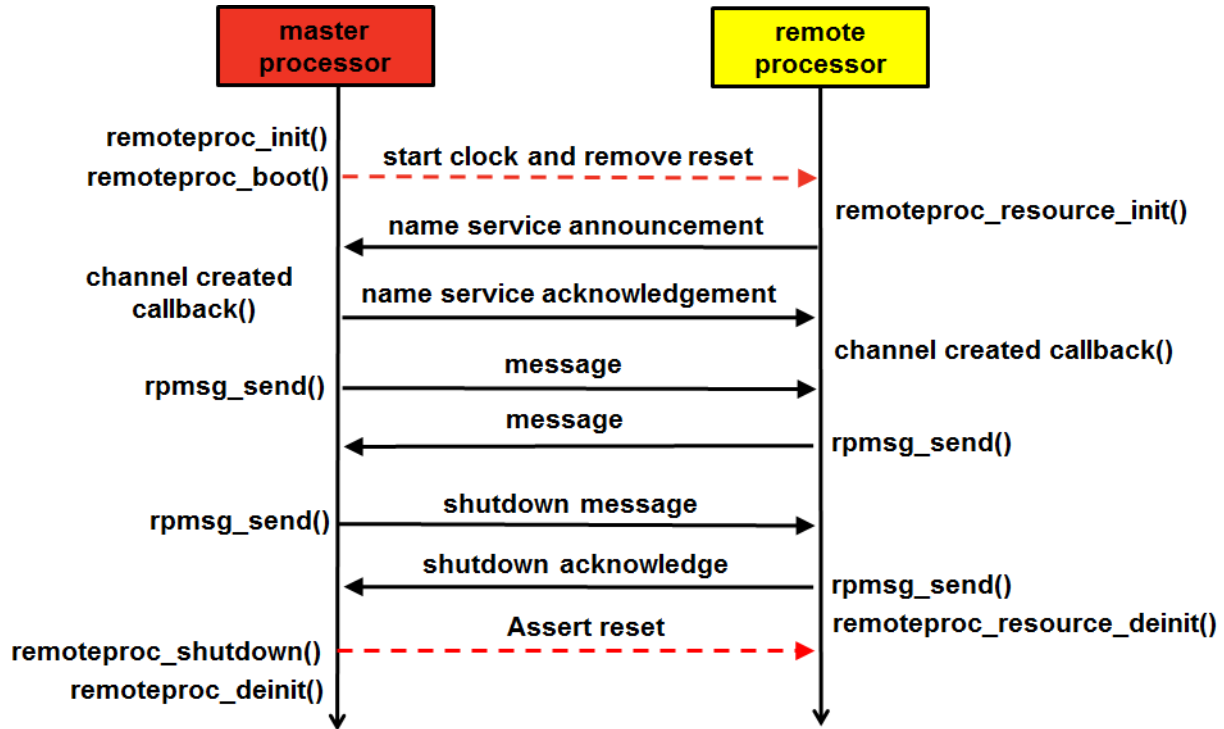


Figure 1-1: System Sequence Diagram

For more information, see the specific function descriptions in [Chapter 4, Remoteproc Development](#) and [Chapter 5, RPMsg Development](#).



# Building Linux Applications and Remote Firmware

---

## 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 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 XSDK 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 XSDK

You can build remote applications using XSDK 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 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.
    - 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 XSDK project explorer, right-click the BSP and select **Board Support Package Settings**.
5. Navigate to the BSP settings drivers: **Settings > Overview > Drivers > <selected\_processor>**.
6. Add any necessary parameters to the `extra_compiler_flags`:

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

```
-DUNDEFINE_FILE_OPS
```

For the Proxy Application demo in the next chapters, you need to set this option, and also make sure the **WITH\_PROXY** option in **Settings > Overview > OpenAMP** is set to `True` (default).

By setting these two options you are instructing the OpenAMP framework to redirect those function calls to the master processor. It is normally not necessary to set those for other applications.

For the Zynq UltraScale+ MPSoC device (zynqMP):

- To force the vector table location in OCM (instead of TCM) add:

```
-DVEC_TABLE_IN_OCM.
```

All application examples in the following chapters have been setup to use OCM for their vector table, so you need to set this parameter. OCM was chosen here to demonstrate its use; however, it is recommended to use TCM if possible for the vector table to get better latency and less jitter.

- 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.

7. Click the **OK** button.

---

## OpenAMP XSDK Key Source Files

The following key source files are available in the Xilinx XSDK 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: 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.
- **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): It contains platform-specific APIs that allow the remote application to communicate with the hardware. It includes functions to initialize and control the GIC.

# 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 `<master_root>`:

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

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

```
cd <master_root>
```

3. Include a remote application in the PetaLinux project.

This step is needed if you are not using one of the pre-built remote firmware already included with the PetaLinux BSP. After you have developed and built a remote application (for example, with XSDK) 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 built with XSDK into this directory:

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

- c. 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>
```




---

**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. These steps are for the Zynq-7000 AP SoC (zynq) device only:

- 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. 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 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 <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 are in the `image/linux` folder:  
`<master_root>/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 <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:

### 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.

# 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*

<code>rsc_info</code>	Pointer to resource table info control block.
<code>channel_created</code>	Callback function for channel creation.

<code>channel_destroyed</code>	Callback function for channel deletion.
<code>rdefault_cb</code>	Default callback for channel I/O.
<code>rproc_handle</code>	Pointer to new remoteproc instance.

**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*

<code>rpdev</code>	The RPMsg channel
<code>data</code>	Payload of message



<code>len</code>	Length of payload
<code>dst</code>	Destination address

### **Returns**

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

## **rpmsg\_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**

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

### **Returns**

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

## **rpmsg\_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

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

## Returns

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

## rpmsg\_trysend

### Description

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

### Usage

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

### Arguments

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

### Returns

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

## rpmsg\_trysendto

### Description

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

### Usage

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

### Arguments

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

### Returns

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

## rpmsg\_trysend\_offchannel

### Description

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

### Usage

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

## Arguments

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

## Returns

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

## rpmsg\_init

### Description

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

### Usage

```
int rpmsg_init( 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

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

## Returns

Status of function execution.

## **rpmsg\_deinit**

### **Description**

Releases the rpmsg driver resources for a given remote instance.

### **Usage**

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

### **Arguments**

rdev: Pointer to device de-initialize.

### **Returns**

None.

## **rpmsg\_get\_buffer\_size**

### **Description**

Returns buffer size available for sending messages.

### **Usage**

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

### **Arguments**

Channel: Pointer to the rpmsg channel or device.

### **Returns**

Buffer size.

## **rpmsg\_create\_channel**

### **Description**

Creates rpmsg channel with the given name for remote device.

### **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 must have first created the RPMsg channel using the rpmsg\_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

# 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@3ed00000 {
            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", "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 RPMMsg 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 RPMMsg 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 **PROXY** option in the OpenAMP section to either **True** or **False**.

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 **PROXY** option in the OpenAMP section to either **True** or **False**.

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

```
-DVEC_TABLE_IN_OCM
```




---

**IMPORTANT:** All application example have been set up to use OCM for their vector table, so you need to set this parameter.

---

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

```
petalinux-build
```

2. Use XSDK 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 XSDK:
  - a. Edit `platform_info.c` and change the channel name to match the one in the `rpmsg_user_dev_driver` above.
  - b. Search and replace `IPI_BASEADDR` value from `0xff310000` to `0xff320000`.
  - c. Search and replace `REMOTE_CPU_ID` value from `1` to `2`.
  - d. Edit `platform_info.h`, and search and replace `VRING1_IPI_INTR_VECT` value from `65` to `66`.
  - e. Edit the linker script file, `lscript.ld`, to avoid memory conflict with other remote processors. For example, to increase DDR start address and use TCM for the vector table.
  - f. Edit the `carveout` sections in `rsc_table.c` for both applications to match the linker script so that they do not conflict.
  - g. 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>, <0x0 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 \[Ref 1\]](#).

# 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. [OpenAMP Wiki](#)
2. *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#))
3. *Xilinx Software Developer Kit Help* ([UG782](#))
4. *PetaLinux Tools Reference Guide* ([UG1144](#))

---

## Please Read: Important Legal Notices

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

© Copyright 2015-2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.