OpenAMP Guide: RPU usage on Zynq Ultrascale+

This document presents a tutorial regarding the use of OpenAMP on a Zynq Ultrascale+ for the communication between APU and RPU. First, the preparation of the environment is shown: all the programs and editor needed to write, compile and load an entire software stack on an MPSoC (like the Zynq Ultrascale+) are deepened. Therefore, we see how to run an OpenAMP pre-built example and how to test it on an emulated architecture using QEMU. Finally, a methodology to write code both for RPU and for APU and to load the software on a real chip is presented.

References

- PetaLinux doc: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1144-petalinux-tools-reference-guide.pdf
- OpenAMP and Libmetal doc: https://www.origin.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1186-zynq-openamp-gsg.pdf
- OpenAMP code on github: https://github.com/OpenAMP/open-amp
- OpenAMP motivation for mixed criticality: https://dornerworks.com/blog/openamp-to-isolate-critical-software/
- Yocto: https://wiki.yoctoproject.org/wiki/FAQ
- ZCU104 Board Setup: https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/2017_4/xtp504-zcu104-setup-c-2017-4.pdf
- ZCU104 Evaluation Board doc: https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf
- How to create a platform from scratch with Vitis: https://github.com/Xilinx/Vitis-Tutorials/tree/2021.1/Vitis_Platform_Creation
- Create a design from zero on Zynq Ultrascale+: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1209-embedded-design-tutorial.pdf

Preparation of the environment

To compile an entire software stack with the aim to run it on a Zynq Ultrascale you need a tool called Petalinux that is described later. The Latter has minimum workstation requirements:

- 8 GB RAM (recommended minimum for Xilinx® tools)
- 2 GHz CPU clock or equivalent (minimum of eight cores)
- 100 GB free HDD space
- Supported OS:
    - Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.1, 8.2 (64-bit)
    - CentOS Workstation/Server 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.1, 8.2 (64-bit)
    - Ubuntu Linux Workstation/Server 16.04.5, 16.04.6, 18.04.1, 18.04.2, 18.04.3, 18.04.4, 18.04.5, 20.04, 20.04.1 (64-bit)

For this tutorial, I decided to use *Ubuntu 20.04.1* on top of a Virtual Machine to make the solution as most easy and portable as possible. If you decide to prepare the environment on a bare-metal machine you can jump the following part.

Here are the main **advantages** to using PetaLinux on a virtual machine:

- Easy to keep working setups for multiple versions of PetaLinux. Although you can install multiple versions of PetaLinux side-by-side on the same machine, in time the dependencies can change or be updated, rendering older versions unstable. With VMs you can keep a working setup for each version of PetaLinux that you need to support.
- Practical if you don't have a spare physical machine to dedicate to the specific requirements of the PetaLinux tools. For one, PetaLinux runs on Linux, and secondly, it has a bunch of dependencies that you may not want to install on your main workstation.

First, to execute a virtual machine inside another virtual machine enable the Virtualization extensions. If you don't know how to do it, follow the following guides:

- For VMware Workstation, and VirtualBox: https://www.tactig.com/enable-intel-vt-x-amd-virtualization-pc-vmware-virtualbox/
- For VMware Fusion: https://techgenix.com/vmware-fusion-5-enable-vt-xept-inside-a-virtual-machine-288/

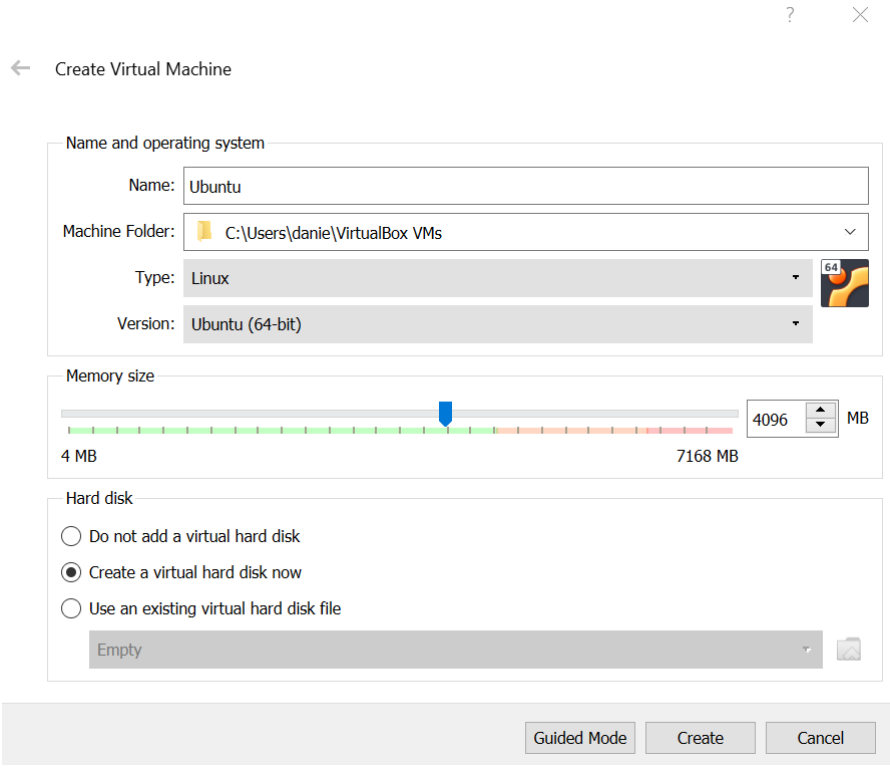
Then, download and install a Virtual Machines Manager like Virtual Box, VMware, or any other (For this tutorial I chose to use VirtualBox).

- VirtualBox download: https://www.virtualbox.org/.
- VMware download: https://www.vmware.com/it/products/workstation-player/workstation-player-evaluation.html

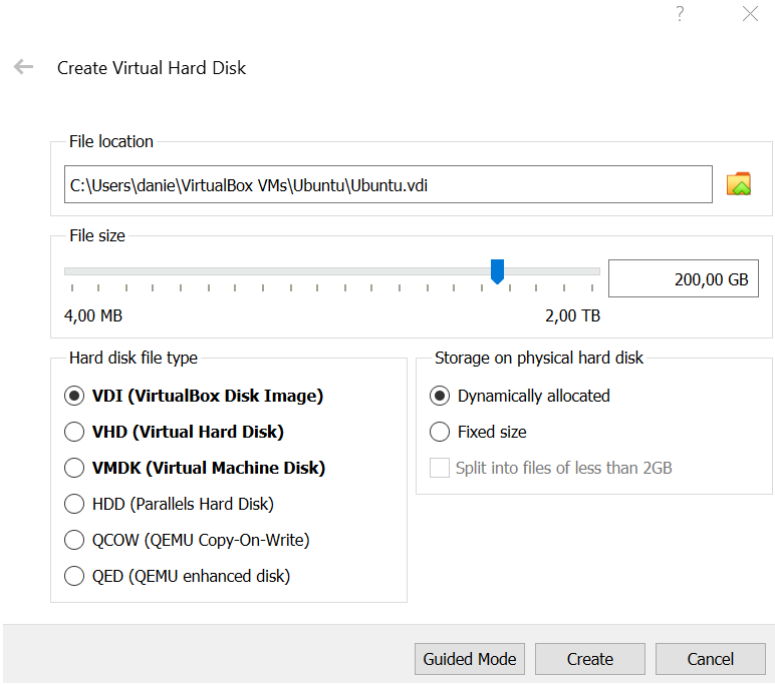
Once the Virtual Machine Manager is installed download the Ubuntu 20.04 ISO image, which you can find here:

- Ubuntu 20.04 ISO: https://releases.ubuntu.com/20.04/

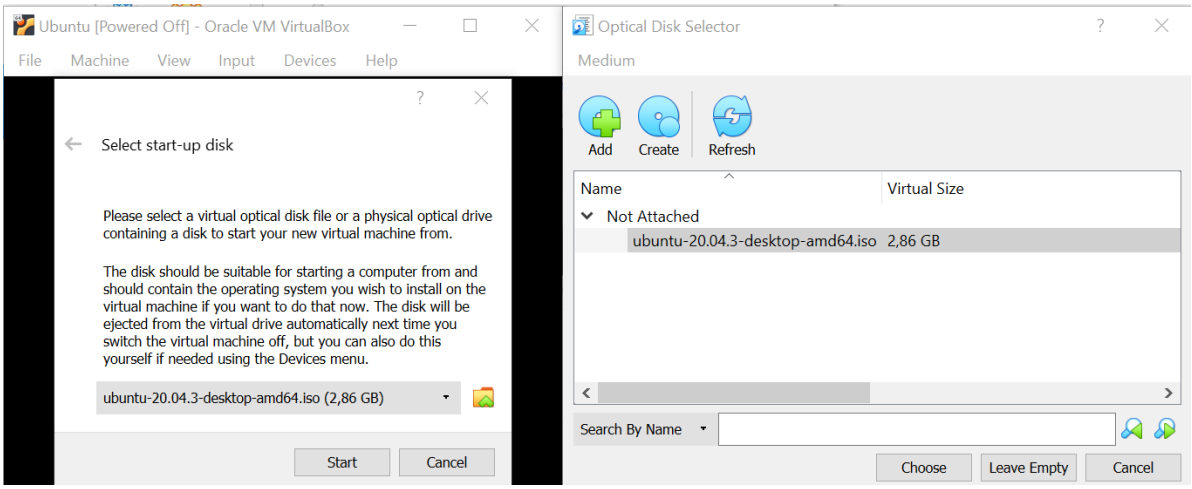
Create a new Ubuntu Virtual Machine using the downloaded ISO (The example is done on VirtualBox, but it is similar on other VM Managers). Open VirtualBox, click on **New**, and compile the widget. Choose **Linux** as type and **Ubuntu(64bit)** as Version. Then assign the quantity of RAM to the virtual machine. I decided to assign 4GB but if you have 8GB is better.



Click Create. Now choose the quantity of disk reserved for the VM, the more the better. Unfortunately, you need to install a lot of software so I suggest you give 200GB (100GB if you don't want to use OpenAMP in User Space).



Click Create again. Finally, **start** the VM. Once started the VM, choose the start-up disk. You can insert the ISO image downloaded before.



Click **Choose** and then **Start**, then complete the Ubuntu installation.

Petalinux

Petalinux is a Software Development Kit (**SDK**) for System on Chip FPGA-Based provided by Xilinx to automate and simplify both the development of a Linux-based embedded environment for a specific board and the loading of the software on the board. To realize it, Petalinux is based on **Yocto** but it simplifies the approach providing only seven commands through which it is possible to take all steps from the creation of a project until the loading on board.

- petalinux-create
- petalinux-config
- petalinux-build
- petalinux-util
- petalinux-package
- petalinux-upgrade
- petalinux-devtool
- petalinux-boot

Dependencies

As with many software development tools, there are a variety of dependencies that you will need in order for PetaLinux to operate. Many of the packages may already be installed on your computer, but some may not. try to download these packages:

```
sudo apt-get -y install iproute2 \
gcc \
g++ \
net-tools \
libncurses5-dev \
zlib1g:i386 \
libssl-dev \
flex \
bison \
libselinux1 \
xterm \
autoconf \
libtool \
texinfo \
zlib1g-dev \
gcc-multilib \
build-essential \
screen \
pax \
gawk \
python3 \
python3-pexpect \
python3-pip \
python3-git \
python3-jinja2 \
xz-utils \
debianutils \
iputils-ping \
libegl1-mesa \
libsdl1.2-dev \
pylint3 \
cpio \
libtinfo5
```

Download and Install

Download Petalinux in your Ubuntu OS from this link:

- https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html

Once the download has been completed, make a directory in which you would like the PetaLinux tools to be installed in. In your terminal, change the directory (cd) into the directory the installer was downloaded into (likely Downloads) and run the installer with a specified path to the directory you just created. Starting from your home directory, open the terminal and enter the following commands:

```
mkdir petalinux/2021.1
cd Downloads
chmod +x petalinux-v2021.1-final-installer.run
./petalinux-v2021.1-final-installer.run -d ../petalinux/2021.1
```

OSS: The PetaLinux tools need to be installed as a non-root user

Change /bin/sh to bash

Furthermore, PetaLinux tools require that your host system */bin/sh* is '**bash**'. If you are using Ubuntu distribution like me, probably your */bin/sh* is '**dash**'. To change it to bash:

1. In the terminal, run this command: `chsh -s /bin/bash`.
2. Reboot the VM.
3. Open a terminal and run these commands to make `/bin/sh` link to `/bin/bash`:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

After the installation, the remaining setup is completed automatically by sourcing the provided settings scripts. In home directory

```
source petalinux/2021.1/settings.sh
```

You can verify that the working environment has been set:

```
echo $PETALINUX
```

(you should see something like: /home/<user>/petalinux/2021.1)

To avoid needing to type the source commands into the shell every time, add a couple of lines to the .bashrc script. To customize this system wide, use a text editor to open your .bashrc file. For Ubuntu, this will be bash.bashrc located in the /etc directory

```
sudo gedit /etc/bash.bashrc
```

Once you have the script open, add the command for sourcing the appropriate file. In my case:

```
    esac
fi

# if the command-not-found package is installed, use it
if [ -x /usr/lib/command-not-found -o -x /usr/share/command-not-found/command-not-f
        function command_not_found_handle {
                # check because c-n-f could've been removed in the meantime
                if [ -x /usr/lib/command-not-found ]; then
                   /usr/lib/command-not-found -- "$1"
                   return $?
                elif [ -x /usr/share/command-not-found/command-not-found ]; then
                   /usr/share/command-not-found/command-not-found -- "$1"
                   return $?
                else
                   printf "%s: command not found\n" "$1" >&2
                   return 127
                fi
        }
fi

source /home/daniele/petalinux/2021.1/settings.sh
```

Install Vitis

If you need an editor to write and compile new code to run on the MPSoC or you want to program the FPGA to configure new devices you need Vitis. Download Vitis from the following link:

- Vitis: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html

Assuming that you downloaded the file in the Download directory, extract the installer and then run it

```
tar xvzf ~/Downloads/Xilinx_Unified_2020.1_0602_1208.tar.gz -C ~/Downloads/.
cd ~/Downloads/Xilinx_Unified_2020.1_0602_1208/
./xsetup
```

Complete the installation. I decided to install Xilinx in the directory "*home/<user>/Xilinx*".

Then, as for petalinux you can add a new command to your .bashrc file to setup the environment every time a new terminal is opened:

```
sudo gedit /etc/bash.bashrc
```

Once you have the script open, add the command for sourcing the appropriate file. In my case:

```
source /home/daniele/Xilinx/Vitis/2021.1/settings64.sh
```

Now is a good moment to take a "snapshot" of your VM so that you can go back to a clean setup if ever it gets corrupted down the line.

OpenAMP Pre-built application on QEMU

The simplest way to get started with PetaLinux consists in using a **Board Support Package** (BSP) provided by Xilinx. BSPs are completed and working project archives, ready to be utilized and pre-configured for a specific board; in my case, the BSP for **zcu104** (Zync Ultrascale+ 104) has been utilized. Download a BSP from this link (I downloaded **ZCU104 Base 2021.1**):

- BSP download: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms.html.

Once the BSP has been downloaded, create a new petalinux project starting from the BSP with the following command:

```
petalinux-create -t project -s /home/<user>/Downloads/xilinx-zcu104-v2021.1-
final.bsp -n <name_of_the_project>
```

Now you should have a petalinux project directory. enter in the directory (in my case I called it *project_zcu104*, so *cd project_zcu104*)

First, you want to run a pre-built application that make use of OpenAMP to communicate with the RPU. To do it you just need to change the device tree blob file (.dtb) that is in the directory */pre-built/linux/images*. A device tree is a *data structure* describing the hardware components of a particular computer so that the operating system's kernel can use and manage those components, including the CPU or CPUs, RPUs, the memory, the buses and the peripheral. In the downloaded BSP the device tree is already compiled for the pre-built application. Petalinux, at compilation time, uses as default device tree the file called system.dtb. To run the OpenAMP tests you want to use the file called openamp.dtb instead, in which the RPU is considered. Hence, rename the file openamp.dtb (save the old system.dtb before):

```
cp pre-built/linux/images/system.dtb pre-built/linux/images/system_old.dtb
cp pre-built/linux/images/openamp.dtb pre-built/linux/images/system.dtb
```

Then use a petalinux command to run the pre-built Linux on QEMU:

```
petalinux-boot --qemu --prebuilt 3
```

Wait for the system to boot and finally, you can try your pre-built Linux on an emulated ZCU104. Once in the system it is possible to run three example applications that make use of OpenAMP:

1. **ECHO TEST**: 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** in kernel space.
    - The Linux master then transmits payloads to the remote firmware using **RPMsg** in kernel space. The remote firmware echoes back the received data using **RPMsg** in user space through OpenAMP.
    - The Linux master verifies and prints the payload.

    To run the test first load the firmware on RPU with the following command:

    ```
    echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware
    ```

    then start the RPU by sending the start message:

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

    You should see the RPU log on the screen.

    Finally, start the Linux Application that make use of RPMsg to communicate with RPU:

    ```
    echo_test
    ```

    If you want to stop the RPU you can launch the following command:

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

2. **MATRIX MULTIPLICATION TEST**: 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*.

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

3. **PROXY TEST**: This application creates a proxy between the Linux master and the remote core, which allows the remote firmware to use the 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

   ```
   proxy_app
   ```

Pre-built application Test on Board

It is possible to run the same example applications shown in the previous section on the Board in two ways:

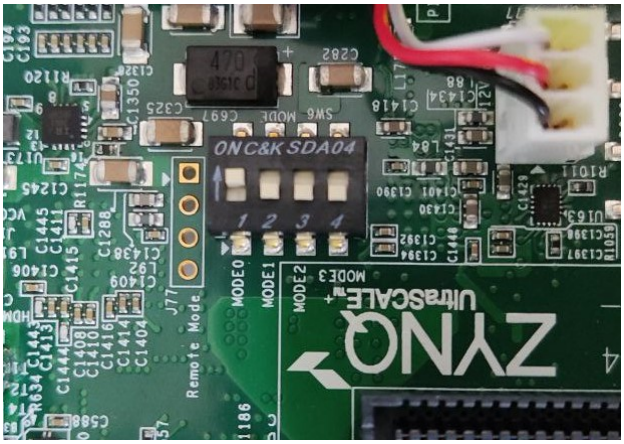
1. **SD card**: Extract files BOOT.BIN, image.ub, and openamp.dtb files from a pre-built PetaLinux BSP tarball to an SD card. Note that the OpenAMP related device nodes are not in the default system.dtb, but are included in the prebuilt openamp.dtb

   ```
   tar xvf xilinx-zcu104-v2021.1-final.bsp --strip-components=4 --wildcards
   */BOOT.BIN */image.ub */openamp.dtb
   cp BOOT.BIN image.ub openamp.dtb <your sd card>
   ```

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

   ```
   cp BOOT.BIN image.ub openamp.dtb <your sd card>
   ```

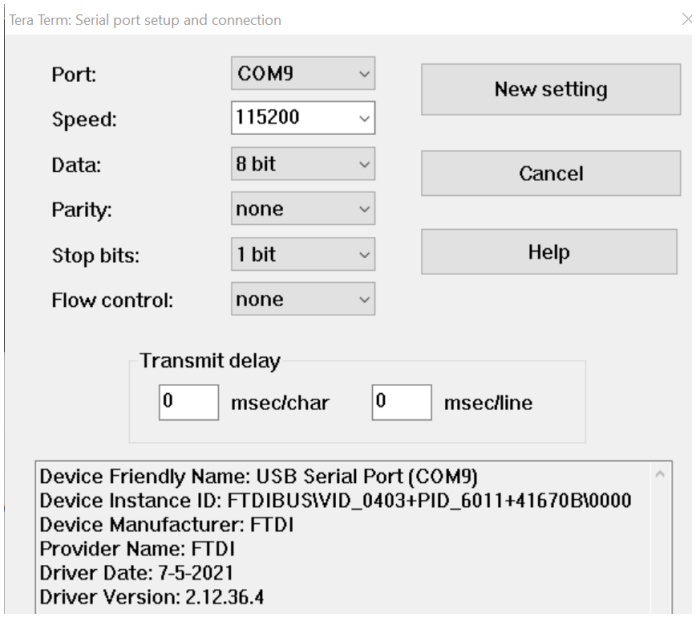
   Prepare the board to load from SD changing the switch configuration as needed (In my case I use the zcu104):



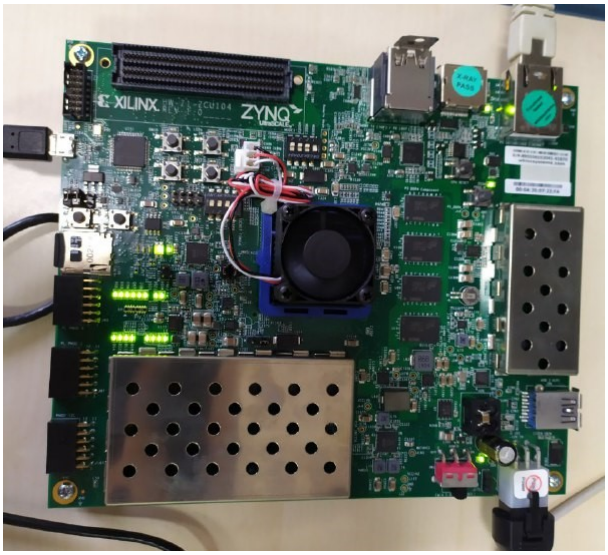
Then install FTDI USB UART Drivers (see these slides page 10) if you want more details on installation and if you don't know how to find the right COM):

- http://www.ftdichip.com/Drivers/VCP.htm
- "WHQL Certified. Includes VCP and D2XX. Available as a setup executable" http://www.ftdichip.com/Drivers/CDM/CDM21228_Setup.zip

Download a program (like *Tera Term* or any other) on your host to receive data from a serial port with a baud rate speed equal to 115200 (In my case the port is COM9 but it probably changes for you):



Finally connect the board to your PC via USB:



Turn on the board, and you should see the first stage boot loader that loads u-boot. After that, insert the following command to boot Linux from SD using openamp.dtb and image.ub:

```
ZynqMP> mmcinfo && fatload mmc 0 0x10000000 image.ub && fatload mmc 0
0x14000000 openamp.dtb
```

you should see:

```
Device: mmc@ff170000
Manufacturer ID: 27
OEM: 5048
```

```
Name: SD8GB
Bus Speed: 50000000
Mode: SD High Speed (50MHz)
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 7.4 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
14527568 bytes read in 961 ms (14.4 MiB/s)
36103 bytes read in 19 ms (1.8 MiB/s)
```

then run *bootm* command to boot the application image from memory:

```
ZynqMP> bootm 0x10000000 0x10000000 0x14000000
```

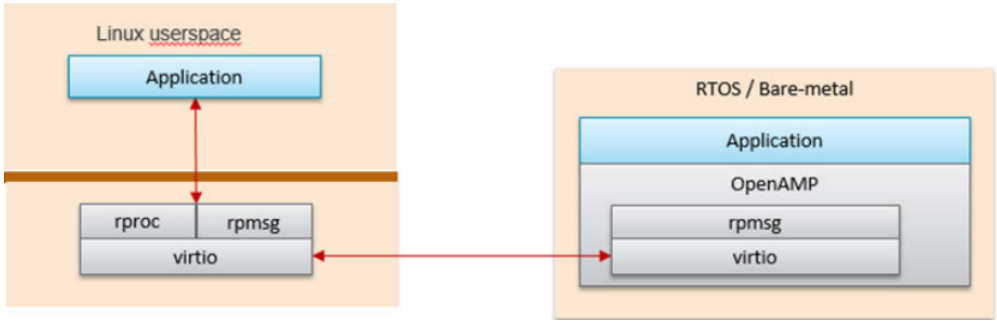
2. **JTAG**: You need to have connected a JTAG cable, installed JTAG drivers, and created a PetaLinux project using a provided BSP. To do this, go in the */pre-built/linux/images* directory and replace the *system.dtb* file by *openamp.dtb*, then type

```
petalinux-boot --jtag --prebuilt 3.
```

OpenAMP Kernel Space vs User Space

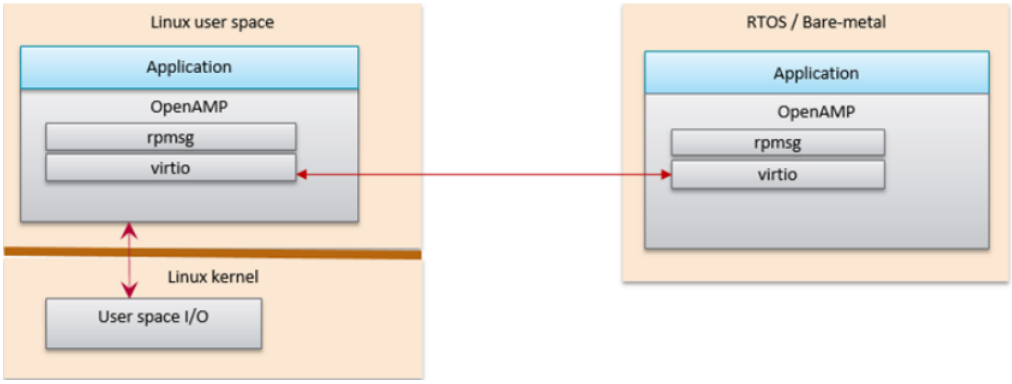
There are two ways to use OpenAMP in Xilinx Zynq Ultrascale+. The first is called "Kernel Space" in the OpenAMP documentation and consists in running *RPMsg* and *remoteproc* in Linux Kernel on APU as Master and OpenAMP on RPU as a slave. The second one is called "User Space" and consists in running OpenAMP in userspace both on APU and on RPU.

- **Kernel Space**:



Linux kernel space provides RPMsg and Remoteproc, but **the RPU application requires Linux to load it** in order to talk to the RPMsg counterpart in the Linux kernel. This is the Linux kernel RPMsg and Remoteproc implementation limitation.

- **User Space**:



OpenAMP library can also be used in Linux userspace. In this configuration, **the remote processor can run independently** to the Linux host processor.

Building an OpenAMP application (Kernel Space)

Everything we saw up to know is based on pre-built code. In this section we see how to create a user application for our board (ZCU104) that uses OpenAMP to communicate with the RPU. To do it you need to:

- Build an OpenAMP application on top of a firmware for the RPU.
- Build a Linux OpenAMP Kernel Space Application with Petalinux
- Configure and Compile the Linux Kernel with Petalinux

In this section I show you how to build and run from scratch the code for the *echo test* seen in the previous demo. In addition, we delve into the code to understand how OpenAMP is used in this simple example.

Build an OpenAMP application on top of a firmware for the RPU

To build the RPU firmware Vitis is necessary: The Xilinx® Vitis contains templates to aid in the development of OpenAMP bare-metal/FreeRTOS remote applications.

First, if it is not already downloaded together with Vitis (see */home//Xilinx/Vitis/2021.1/data/embeddedsw/lib/fixed_hwplatforms*), download the platform file (.xsa) of your board here (In my case "*ZCU104 Base 2021.1*"):

- https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms.html

Extract the files and copy the file "*xilinx_zcu104_base_202110_1.xsa*" (see xilinx_zcu104_base_202110_1/hw/) in the following path: */home/<user>/Xilinx/Vitis/2021.1/data/embeddedsw/lib/fixed_hwplatforms*)

OSS: it is also possible to create from zero the ".xsa" file exploiting the Vivado application from Xilinx. If you are interested in create the platform from scratch read this document: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1209-embedded-design-tutorial.pdf

1. From the Xilinx Vitis window, create the application project by selecting **File > New > Application Projects** or select **Create Application Project**.
2. The wizard will guide you through the steps of creating new application projects.
    - Go in **create a new platform project from hardware (XSA)**. Click **Browse...** and select the .xsa file previously downloaded or the one related to your chip.

    - In Boot Components select **psu_cortexr5_0** and click **Next**.
    - Give a Name to the application project. I called it "*firmware-rpu*", and select as target processor "*psu_cortexr5_0*" (If you want to use "*psu_cortexr5_1*" see the official documentation for the changes), click **Next**.
    - In Domain details you have to choose to run bare-metal software or freeRTOS. In this tutorial I chose ""**freertos10_xilinx**"" as Operating System. Click **Next**.
    - Finally select **OpenAMP echo-test** as template e and click **Finish**.
3. In the Xilinx Vitis project Explorer, click the platform (in my case xilinx_zcu104_base_202110_1) and click **platform.spr**. Below **freertos10_xilinx_psu_cortexr5_0** click **Board Support Package** and then click **Modify BSP Settings...**. Check if OpenAMP and Libmetal are checked. If not, check it and then click **OK**.

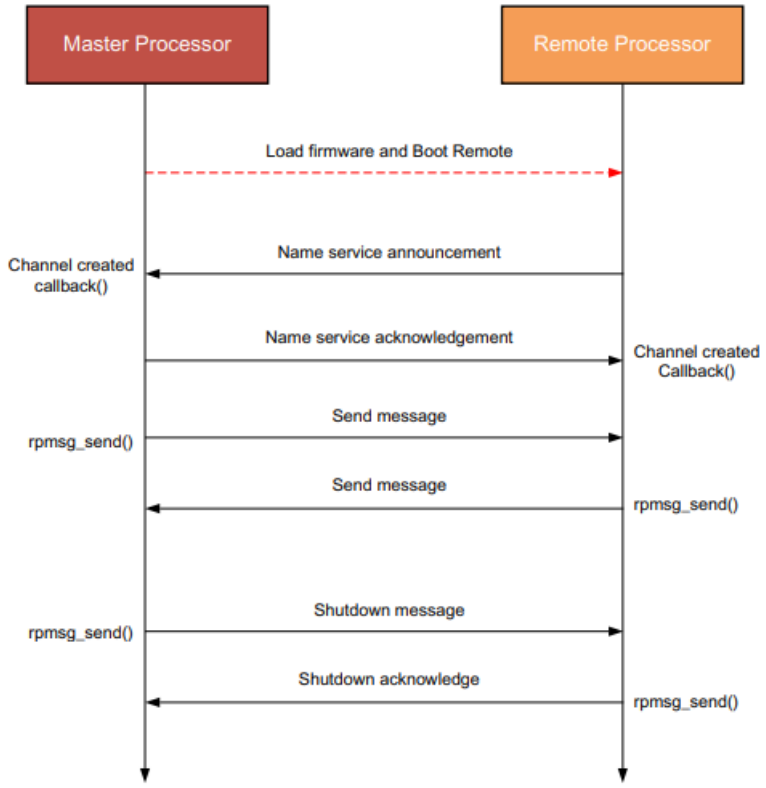
In the src directory of your project you can see some files. Let's understand the meaning of these files:

- **Platform Info** (platform_info.c/.h): These files contain hard-coded, platform-specific values used to get necessary information for OpenAMP.
- **Resource Table** (rsc_table.c/.h): The resource table contains entries that specify the memory and virtIO device resources. The virtIO device contains device features, vring addresses, size, and alignment information. The resource table entries are specified in rsc_table.c and the remote_resource_table structure is specified in rsc_table.h.
- **Helper** (helper.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.
- **Zynq specific** (zynqmp_r5_a53_rproc.c): This file define Xilinx ZynqMP R5 to A53 platform specific remoteproc implementation.
- **Application code** (rpmsg_echo.c/rpmsg_echo.h): The application code create a task that waits for a message from the APU. When the message is received it wakes up and sends an echo back.

Code

Let's see how does OpenAMP works. First, you should have in mind the communication flow:

1. The Master load the firmware and load the remote processor
2. The remote processor uses the *resource table*, which is inside the firmware, to define a list of system resources needed together with their addresses in memory. Hence, it setup the hardware.
3. From the resource table, the remote processor creates a *remoteproc* struct based on real hardware.
4. Define the RPMsg *callback functions* and create the *virtIO device*.
5. Create an RPMsg *endpoint* and associate the RPMsg device with the callback functions
6. After initializing the framework, the communication can start through the *rpmsg_send()* function and *I/O callback* functions using the RPMsg channel created.



Once the flow is clear you can see the application code running on RPU starting from the *main* function.

```
/*-----------------------------------------------------------------------------*
 *  Application entry point
 *-----------------------------------------------------------------------------*/
int main(void)
{
    BaseType_t stat;

    /* Create the tasks */
    stat = xTaskCreate(processing, ( const char * ) "HW2",
                1024, NULL, 2, &comm_task);
    if (stat != pdPASS) {
        LPERROR("cannot create task\n");
    } else {
        /* Start running FreeRTOS tasks */
        vTaskStartScheduler();
    }

    /* Will not get here, unless a call is made to vTaskEndScheduler() */
    while (1) ;

    /* suppress compilation warnings*/
    return 0;
}
```

It simply creates a freeRTOS task and starts the scheduler. The task runs the *processing()* function

```
/*-----------------------------------------------------------------------------*
 *  Processing Task
 *-----------------------------------------------------------------------------*/
static void processing(void *unused_arg)
{
    void *platform;
    struct rpmsg_device *rpdev;

    LPRINTF("Starting application...\n");
    /* Initialize platform */
    if (platform_init(NULL, NULL, &platform)) {
        LPERROR("Failed to initialize platform.\n");
    } else {
        rpdev = platform_create_rpmsg_vdev(platform, 0,
                                           VIRTIO_DEV_SLAVE,
                                           NULL, NULL);
        if (!rpdev){
            LPERROR("Failed to create rpmsg virtio device.\n");
        } else {
            app(rpdev, platform);
            platform_release_rpmsg_vdev(rpdev);
        }
    }

    LPRINTF("Stopping application...\n");
    platform_cleanup(platform);

    /* Terminate this task */
    vTaskDelete(NULL);
}
```

The function *platform_init()*:

- initialize the hardware platform
- create *remoteproc* struct from the resource table
- Parse the resource table

Then the function *platform_create_rpmsg_vdev()*:

- Reserve memory for the communication channel
- Create a VirtIO device
- Create an RPMsg virtio device
- return an RPMsg device (or RPMsg channel)

Then if nothing went wrong it start the *app()* function.

```
/*-----------------------------------------------------------------------------*
 *  Application
 *-----------------------------------------------------------------------------*/
int app(struct rpmsg_device *rdev, void *priv)
{
    int ret;

    /* Initialize RPMSG framework */
    LPRINTF("Try to create rpmsg endpoint.\r\n");

    ret = rpmsg_create_ept(&lept, rdev, RPMSG_SERVICE_NAME,
                  RPMSG_ADDR_ANY, RPMSG_ADDR_ANY,
                  rpmsg_endpoint_cb,
                  rpmsg_service_unbind);
    if (ret) {
        LPERROR("Failed to create endpoint.\r\n");
        return -1;
    }

    LPRINTF("Successfully created rpmsg endpoint.\r\n");
    while(1) {
        platform_poll(priv);
        /* we got a shutdown request, exit */
        if (shutdown_req) {
            break;
        }
    }
    rpmsg_destroy_ept(&lept);

    return 0;
}
```

the function *rpmsg_create_ept()*:

- Create an *endpoint*
- Bind the endpoint to the RPMsg device previously created

- Bind the endpoint to a callback
- Bind the endpoint to a service unbind callback, called when the remote endpoint is destroyed

Therefore, using the function *platform_poll()* the application waits for an interrupt coming from the remote processor.

When the interrupt occurs, if it is addressed to the right endpoint, the callback is executed:

```
/*-----------------------------------------------------------------------------*
 *  RPMSG endpoint callbacks
 *-----------------------------------------------------------------------------*/
static int rpmsg_endpoint_cb(struct rpmsg_endpoint *ept, void *data, size_t len,
                             uint32_t src, void *priv)
{
    (void)priv;
    (void)src;

    /* On reception of a shutdown we signal the application to terminate */
    if ((*(unsigned int *)data) == SHUTDOWN_MSG) {
        LPRINTF("shutdown message is received.\r\n");
        shutdown_req = 1;
        return RPMSG_SUCCESS;
    }

    /* Send data back to master */
    if (rpmsg_send(ept, data, len) < 0) {
        LPERROR("rpmsg_send failed\r\n");
    }
    return RPMSG_SUCCESS;
}

static void rpmsg_service_unbind(struct rpmsg_endpoint *ept)
{
    (void)ept;
    LPRINTF("unexpected Remote endpoint destroy\r\n");
    shutdown_req = 1;
}
```

The callback does a check on the data to verify if it is a Shutdown message. If not, the function sends back the data to the master through a *rpmsg_send()* function. The latter takes as input the endpoint of the master, the message (data) and the length of the message, and uses the RPMsg device to send the message through shared memory.

N.B. There is a Bug in the example. If more tasks are running on top of freeRTOS they may not be scheduled. This is because both the FreeRTOS port and the example itself are configuring the GIC. Thus, the second write of the interrupt vector overwrites the first one.

In order to fix this error modify the file *helper.c* commenting the initialization of the interrupt controller driver:

```
/* Interrupt Controller setup */
static int app_gic_initialize(void)
{
    uint32_t status;
    //XScuGic_Config *int_ctrl_config; /* interrupt controller configuration params */
    uint32_t int_id;
    uint32_t mask_cpu_id = ((u32)0x1 << XPAR_CPU_ID);
    uint32_t target_cpu;

    mask_cpu_id |= mask_cpu_id << 8U;
    mask_cpu_id |= mask_cpu_id << 16U;

    Xil_ExceptionDisable();

    /*
     * Initialize the interrupt controller driver
     */
//  int_ctrl_config = XScuGic_LookupConfig(INTC_DEVICE_ID);
//  if (NULL == int_ctrl_config) {
//      return XST_FAILURE;
//  }

//  status = XScuGic_CfgInitialize(&xInterruptController, int_ctrl_config,
//                    int_ctrl_config->CpuBaseAddress);
//  if (status != XST_SUCCESS) {
//      return XST_FAILURE;
//  }

    /* Only associate interrupt needed to this CPU */
//  for (int_id = 32U; int_id<XSCUGIC_MAX_NUM_INTR_INPUTS;int_id=int_id+4U) {
//      target_cpu = XScuGic_DistReadReg(&xInterruptController,
//                      XSCUGIC_SPI_TARGET_OFFSET_CALC(int_id));
        /* Remove current CPU from interrupt target register */
//      target_cpu &= ~mask_cpu_id;
//      XScuGic_DistWriteReg(&xInterruptController,
//                  XSCUGIC_SPI_TARGET_OFFSET_CALC(int_id), target_cpu);
//  }
    XScuGic_InterruptMaptoCpu(&xInterruptController, XPAR_CPU_ID, IPI_IRQ_VECT_ID);
```

Finished the preparation of the files click **Debug** to compile the files and the platform. In output you should have a .elf file (in my case it is called *firmware-rpu.elf*). This file must be loaded in the petalinux project to install the RPU firmware in the linux rootfs.

To do it open a terminal, go in the project directory, and run:

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

(my app_name is "*firmware-rpu*")

OSS: if you see "ERROR: Failed to get yocto SDK environment file" run *petalinux-config* then simply exit end retry.

Then copy the .elf file in the directory: *project-spec/meta-user/recipes-apps/<app_name>/files/*

```
cp firmware-rpu.elf  project-spec/meta-user/recipes-apps/<app_name>/files/
```

Moreover, adapt the <app_name>.bb file in */project-spec/meta-user/recipes-apps/<app_name>/<app_name>.bb* as below:

```
SUMMARY = "Simple firmware-rpu application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://firmware-rpu.elf"
S = "${WORKDIR}"
INSANE_SKIP_${PN} = "arch"
do_install() {
         install -d ${D}/lib/firmware
         install -m 0644 ${S}/firmware-rpu.elf ${D}/lib/firmware/firmware-rpu
}
FILES_${PN} = "/lib/firmware/firmware-rpu"
```

Last but not least adjust the device-tree in *project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi* to make Linux see the RPU as a device:

```
/include/ "system-conf.dtsi"
/ {
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;
        rpu0vdev0vring0: rpu0vdev0vring0@3ed40000 {
            no-map;
            reg = <0x0 0x3ed40000 0x0 0x4000>;
        };
        rpu0vdev0vring1: rpu0vdev0vring1@3ed44000 {
            no-map;
            reg = <0x0 0x3ed44000 0x0 0x4000>;
        };
        rpu0vdev0buffer: rpu0vdev0buffer@3ed48000 {
            no-map;
            reg = <0x0 0x3ed48000 0x0 0x100000>;
        };
        rproc_0_reserved: rproc@3ed00000 {
            no-map;
            reg = <0x0 0x3ed00000 0x0 0x40000>;
        };
    };


    tcm_0a@ffe00000 {
        no-map;
        reg = <0x0 0xffe00000 0x0 0x10000>;
        phandle = <0x40>;
        status = "okay";
        compatible = "mmio-sram";
    };

    tcm_0b@ffe20000 {
        no-map;
        reg = <0x0 0xffe20000 0x0 0x10000>;
        phandle = <0x41>;
        status = "okay";
        compatible = "mmio-sram";
```

```
    };


    rf5ss@ff9a0000 {
        compatible = "xlnx,zynqmp-r5-remoteproc";
        xlnx,cluster-mode = <1>;
        ranges;
        reg = <0x0 0xFF9A0000 0x0 0x10000>;
        #address-cells = <0x2>;
        #size-cells = <0x2>;

        r5f_0 {
            compatible = "xilinx,r5f";
            #address-cells = <2>;
            #size-cells = <2>;
            ranges;
            sram = <0x40 0x41>;
            memory-region = <&rproc_0_reserved>, <&rpu0vdev0buffer>,
<&rpu0vdev0vring0>, <&rpu0vdev0vring1>;
            power-domain = <0x7>;
            mboxes = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;
            mbox-names = "tx", "rx";
        };
    };


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

        /* APU<->RPU0 IPI mailbox controller */
        ipi_mailbox_rpu0: mailbox@ff990600 {
            reg = <0xff990600 0x20>,
                  <0xff990620 0x20>,
                  <0xff9900c0 0x20>,
                  <0xff9900e0 0x20>;
            reg-names = "local_request_region",
                    "local_response_region",
                    "remote_request_region",
                    "remote_response_region";
            #mbox-cells = <1>;
            xlnx,ipi-id = <1>;
        };
    };
};
```

Build a Linux OpenAMP Kernel Space Application with Petalinux

To build the linux application download the code here:

- https://github.com/OpenAMP/meta-openamp/tree/master/recipes-openamp/rpmsg-examples/rpmsg-echo-test

Create a new application as before. Open the terminal, go in the project directory and run:

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

(my app_name is "*echo-test-kernel*")

Now copy the three downloaded files (LICENSE, Makefile and echo_test.c) in the following directory:

```
cp LICENSE Makefile echo_test.c project-spec/meta-user/recipes-apps/echo-test-kernel/files
```

then change the name of *echo_test.c* (because it already exist in the Linux applications) in something like *echo-test-kernel.c* and change the *Makefile* accordingly:

```
APP = echo-test-kernel
APP_OBJS = echo-test-kernel.o

# Add any other object files to this list below


all: $(APP)

$(APP): $(APP_OBJS)
    $(CC) $(LDFLAGS) -o $@ $(APP_OBJS) $(LDLIBS)

clean:
    rm -rf $(APP) *.o

%.o: %.c
    $(CC) -c $(CFLAGS) -o $@ $<
```

As before adapt the <app_name>.bb file as below (in my case the path is */project-spec/meta-user/recipes-apps/echo-test-kernel/echo-test-kernel.bb*):

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

SRC_URI = "\
    file://LICENSE \
    file://Makefile \
    file://echo-test-kernel.c \
    "
```

```
S = "${WORKDIR}"

RRECOMMENDS_${PN} = "kernel-module-rpmsg-char"

FILES_${PN} = "\
    /usr/bin/echo-test-kernel\
"

do_install() {
         install -d ${D}/usr/bin
         install -m 0755 ${S}/echo-test-kernel ${D}/usr/bin/echo-test-kernel
}
```

Configure and Compile the Linux Kernel with Petalinux

Before to build the kernel you need to configure the rootfs and you can do it using Petalinux. Open the terminal in the project directory and run:

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

using the interface check those configuration options:

```
Filesystem Packages
    --> libs
        --> libmetal
        --> [ * ] libmetal
        --> openamp
        --> [ * ] open-amp
    --> misc
        --> openamp-fw-echo-testd
        --> [ * ] openamp-fw-echo-testd
        --> openamp-fw-mat-muld
        --> [ * ] openamp-fw-mat-muld
        --> openamp-fw-rpc-demod
        --> [ * ] openamp-fw-rpc-demod
        --> rpmsg-echo-test
        --> [ * ] rpmsg-echo-test
        --> rpmsg-mat-mul
        --> [ * ] rpmsg-mat-mul
        --> rpmsg-proxy-app
        --> [ * ] rpmsg-proxy-app

Petalinux Package Groups
    packagegroup-petalinux-openamp
     ---> [*] packagegroup-petalinux-openamp
```

Finally build the kernel (It might take some time):

```
petalinux-build
```

Test on QEMU

Once the build is finished you can test your code through QEMU. To do it launch:

```
petalinux-boot --qemu --kernel
```

And as before load the firmware, start the remote processor and start the Linux application:

```
cd /lib/firmware
echo firmware-rpu > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
echo-test-kernel
```

If you want to stop the remote core launch:

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

Test on Board (SD card)

First format the SD card. To do it use a program like **gparted** (or do it directly from terminal):

```
sudo apt-get install gparted
sudo gparted
```

create two partitions:

- **BOOT**: The first partition is called "BOOT" and is 500Mb (leave the first 4Mb free to speed up the memory accesses). The format is "Fat32". This partition is used to store the boot loader (u-boot), the device tree and the kernel image.
- **RootFS**: The second partition in called "RootFS" and can take all the remaining space. The format is "ext4". This partition is used to store the root file system.

We need to change the configuration. Open the terminal in the project directory and type:

```
petalinux-config
```

Check the following configuration option:

```
Image Packaging Configuration  ---> Root filesystem type ---> EXT4
(SD/eMMC/SATA/USB)
```

OSS: Choose this setting to configure your PetaLinux build for EXT root. By default, it adds the SD/eMMC device name in bootargs. For other devices (SATA/USB), you must change the ext4 device name, as shown in the following examples:

- eMMC or SD root = /dev/mmcblkYpX
- SATA or USB root= /dev/sdX

exit and save the configuration. Then build the kernel :

```
petalinux-build
```

Then, create the BOOT.BIN file:

```
petalinux-package --boot --u-boot --format BIN --force
```

Now copy the following three files in the BOOT partition. Go in the project directory and type (I assume that the path for your SD card is */media/<user>/*):

```
cp images/linux/BOOT.BIN /media/<user>/BOOT/
cp images/linux/image.ub /media/<user>/BOOT/
cp images/linux/boot.scr /media/<user>/BOOT/
```

Moreover, you must extract the roofs in the RootFS partition (same assumption on the SD card path):

```
sudo tar xvf images/linux/rootfs.tar.gz -C /media/<user>/RootFS
```

Finally insert the SD cart in the Board, connect the USB to your PC, start the serial program (like Tera Term) and start the system as seen before in the "Pre-built application Test on Board " section. This time it should boot without writing the address intervals. As before to test the application type:

```
cd /lib/firmware
echo firmware-rpu > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
echo-test-kernel
```

If you want to stop the remote core launch:

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

Building an OpenAMP application (User Space)

As before, to realize an OpenAMP application in User Space we need to build a firmware (file ".elf") to run on RPU and a Linux application (another file ".elf") to run on APU. Hence, we'll see:

- Build an OpenAMP User Space application on top of a firmware for the RPU
- Build a Linux OpenAMP User Space Application with Petalinux
- Configure and Compile the Linux Kernel with Petalinux (User Space)

Build an OpenAMP User Space application on top of a firmware for the RPU

The procedure is the same that has been described in "*Build an OpenAMP application on top of a firmware for the RPU*", but there are some differences:

1. In the Xilinx Vitis project Explorer, click the platform (in my case *xilinx_zcu104_base_202110_1*) and click **platform.spr**. Below **freertos10_xilinx_psu_cortexr5_0** click **Board Support Package** and then click **Modify BSP Settings.** Click **openamp** and in the entry named *WITH_RPMSG_USERSPACE* select value **true**. Click ok and rebuild the project.

2. Open the file called "*rsc_table.c*" in the firmware source files on Vitis. Modify the following flag:

```
/* Virtio device entry */
{
 RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0,
VIRTIO_CONFIG_STATUS_DRIVER_OK,
 NUM_VRINGS, {0, 0},
 },
```

3. In the same file ("*rsc_table.c*") change also the following addresses as shown:

```
#define RING_TX                         0x3ed40000
#define RING_RX                         0x3ed44000
```

Build a Linux OpenAMP User Space Application with Petalinux

In this case we are going to use Vitis to build the application. You can find the source code of the Linux userspace RPMsg applications demos in the following locations:

- For the common code across the three applications:
  - platform_info.c and platform_info.h define platform specifc data and implement API's to set platform specific information for OpenAMP.
    - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp/platform_info.c
    - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp/platform_info.ch
  - rsc_table.c and rsc_table.h populate the resource table for the remote core for use by the Linux master.
    - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.c
    - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.h
  - helper.c provide api to initialize and clean up the system
    - https://github.com/OpenAMP/open-amp/blob/master/apps/system/linux/machine/zynqmp/helper.c
  - zynqmp_linux_r5_proc.c define Xilinx ZynqMP R5 to A53 platform specific remoteproc implementation.
    - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp/zynqmp_linux_r5_proc.c
- Application specific code:
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/echo/rpmsg-ping.c (I'll use the echo example)
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/matrix_multiply/matrix_multiply.c
  - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/rpc_demo/rpc_demo.c

Open Vitis and create an empty application for Linux and for A53 . Click **File**->**New**->**Application project..**

1. The wizard will guide you through the steps of creating new application projects.
    - Go in **create a new platform project from hardware (XSA)**. Click **Browse...** and select the .xsa file previously downloaded or the one related to your chip.
    - In Target processor select **psu_cortexa53 SMP** and give a Name to the application project. I called it "*userspace-test*", click **Next**.
    - In Domain details you have to choose **linux_psu_cortexa53** and **linux** as OS. While in Application settings click **Browse..**, go in the petalinux project and select:
        - Sysroot path: <plnx-proj-root>/images/linux/sdk/sysroots/cortexa72-cortexa53-xilinx-linux
        - Root FS : rootfs.tar.gz
        - Kernel image: image.ub
    - Finally select **empty application (C++)** as template e and click **Finish**.
2. Once project is built, select properties:
    - C/C++ Build --> Settings
        - Tool Settings Tab Libraries
            - Libaries (-l) add "metal" and "open_amp"
            - Miscellaneous
                - in Linker Flags, add "--sysroot=<plnx-proj-root>/images/linux/sdk/sysroots/cortexa72-cortexa53-xilinx-linux"
3. Now insert as source files, the files previously downloaded:
    - helper.c
    - platform_info.c
    - platform_info.h
    - zynqmp_linux_r5_proc.c
    - rpmsg-echo.h
    - rpmsg-ping.c
4. When building the application note a few configuration parameters:
    - If building Linux application to communicate with RPU 1:
        - replace the following in platform_info.c: change IPI_MASK to 0x200
    - The RSC_RPROC_MEM entries must be within the corresponding vring device tree node.
        - Update the following inside of the Linux application's platform_info.c to reflect possible changes to the device tree nodes:
            - IPI_DEV_NAME
            - VRING_DEV_NAME
            - SHM_DEV_NAME
    - The above means that if the new vring entry in the device tree entry is at 0x3ef00000, then the string for the VRING_DEV_NAME should now be "3ef00000.vring" as shown in sysfs on target.
    - Update RING_TX and RING_RX to reflect the vring entry in the device tree.
5. Build the application to generate the file "*userspace-test.elf*"

Create a new application as before: Open the terminal, go in the project directory and run:

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

(my app_name is "*userspace-test*")

Copy the "*userspace-test.elf*" file in the directory just created:

```
cp userspace-test.elf /home/<user>/<project>/project-spec/meta-user/recipes-
apps/userspace-test/files
```

As before adapt the <app_name>.bb file as below (in my case the path is */project-spec/meta-user/recipes-apps/userspace-test/userspace-test.bb*):

```
#
# This file is the userspace-test recipe.
#

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

SRC_URI = "file://userspace-test.elf \
    "

S = "${WORKDIR}"
INSANE_SKIP_${PN} = "arch"
do_install() {
             install -d ${D}/lib/firmware
             install -m 0644 ${S}/userspace-test.elf ${D}/lib/firmware/userspace-
test
}

FILES_${PN} = "/lib/firmware/userspace-test"
```

Configure and Compile the Linux Kernel with Petalinux (User Space)

First check the configuration option:

```
petalinux-config -c rootfs

Petalinux Package Groups
    packagegroup-petalinux-openamp
     ---> [*] packagegroup-petalinux-openamp
```

If you want to use the SD card for testing run:

```
petalinux-config
```

Check the following configuration option:

```
Image Packaging Configuration  ---> Root filesystem type ---> EXT4
(SD/eMMC/SATA/USB)
```

Therefore, add the following device tree content to *<petalinux project>/project-spec/meta-user/recipes-bsp/device-tree/file/system-user.dtsi* (The libmetal Linux demo uses Userspace I/O (UIO) devices for IPI and shared memory.)

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

        amba {
        vring: vring@0 {
            compatible = "vring_uio";
            reg = <0x0 0x3ed40000 0x0 0x40000>;
        };
        shm0: shm@0 {
            compatible = "shm_uio";
            reg = <0x0 0x3ed20000 0x0 0x0100000>;
        };
        ipi0: ipi@0 {
            compatible = "ipi_uio";
            reg = <0x0 0xff340000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
        };
    };
};
```

build the kernel (It might take some time):

```
petalinux-build
```

Test on Board User Space (SD card)

This time to generate the BOOT.BIN file we will use a file called **bootgen.bif**. The latter is important to specify which files have to be packaged into the BOOT.BIN file. In particular, we need it to boot the RPU firmware at the start-up; this is because in User Space it is not possible to use remoteproc to power-on the RPU as we did in kernel space. The **bootgen.bif** file can be written by hand or can be generated using the Xilinx SDK.

Copy the following code in the bootgen.bif file paying attention to change the path according to the position of your files:

```
the_ROM_image:
{
[bootloader, destination_cpu=a53-0] <plnx-proj-root>/images/linux/zynqmp_fsbl.elf

[destination_device=pl] <plnx-proj-root>/project-spec/hw-
description/project_1.bit
[pmufw_image] <plnx-proj-root>/images/linux/pmufw.elf
[destination_cpu=r5-0] <plnx-proj-root>/images/linux/firmware-rpu.elf
[destination_cpu=a53-0, exception_level=el-3, trustzone] <plnx-proj-
root>/images/linux/bl31.elf
[destination_cpu=a53-0, load=0x00100000] <plnx-proj-
root>/images/linux/system.dtb
[destination_cpu=a53-0, exception_level=el-2] <plnx-proj-root>/images/linux/u-
boot.elf
}
```

OSS: I copied the rpu firmware ("*firmware-rpu.elf*") previously generated in the following directory with the other files: <plnx-proj-root>/images/linux/firmware-rpu.elf

Finally it is possible to create the BOOT.BIN file. Go in the project directory and run:

```
petalinux-package --boot --u-boot --format BIN --bif bootgen.bif --force
```

Now insert the SD card in your PC and copy the following three files in the BOOT partition. Go in the project directory and type (I assume that the path for your SD card is */media/<user>/*):

```
cp images/linux/BOOT.BIN /media/<user>/BOOT/
cp images/linux/image.ub /media/<user>/BOOT/
cp images/linux/boot.scr /media/<user>/BOOT/
```

Moreover, you must extract the roofs in the RootFS partition (same assumption on the SD card path):

```
sudo tar xvf images/linux/rootfs.tar.gz -C /media/<user>/RootFS
```

Finally insert the SD cart in the Board, connect the USB to your PC, start the serial program (like Tera Term) and start the system as seen before in the "Pre-built application Test on Board " section. This time it should boot without writing the address intervals.

You should see this code at the start (meaning the RPU is working fine):

```
Xilinx Zynq MP First Stage Boot Loader
Release 2021.1   Jun  6 2021  -  07:07:32
NOTI
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.

CE:  ATF running on XCZU7EV/silicon v4/RTL5.1 at 0xfffea000
NOTICE:  BL31: v2.4(release):xlnx_rebase_v2.4_2021.1
```

```
NOTICE:  BL31: Built : 08:27:07, Apr 28 2021


U-Boot 2021.01 (Jun 01 2021 - 11:54:06 +0000)

....
```

On the APU Linux target console, run the demo applications we created:

```
cd /lib/firmware
```

```
./userspace-test
```