

Setting up and using Xilinx KRIA KV260 and KR260

南山大学

2024-05-02

Vincent Conus - Source available at [GitLab](#) 



Contents

| | | |
|-----------|--|----------------------------|
| 1 | Introduction & motivation | 3 |
| 2 | Boot firmware | 3 |
| 2.0.1 | Getting the new firmware | 3 |
| 2.0.2 | Reaching the board recovery tool | 3 |
| 2.0.3 | Updating the boot firmware | 4 |
| 3 | Installing Ubuntu LTS 22.04 | 5 |
| 3.1 | Preparing and booting a Ubuntu 22.04 media | 6 |
| 3.2 | Network and admin setups | 6 |
| 3.2.1 | Static IP address | 6 |
| 3.2.2 | Proxy and DNS | 7 |
| 3.2.3 | root password | 7 |
| 3.2.4 | Adding Xilinx specific repositories | 8 |
| 3.2.5 | Purging snap | 8 |
| 3.2.6 | Other unused heavy packages | 9 |
| 3.2.7 | Slow boot services to disable | 9 |
| 3.2.8 | Jupyter notebook setup | 9 |
| 3.2.9 | Enabling remoteproc with Device-Tree Overlay patching | 10 |
| 3.2.10 | Installing Docker | 12 |
| 3.2.11 | Adding a swap partition | 13 |
| 3.2.12 | TODO Using a PetaLinux kernel in Ubuntu | 13 |
| 4 | Setting up a Vitis IDE project | 13 |
| 4.0.1 | Setting up the IDE | 13 |
| 4.0.2 | Setting up and building a new project for the Kria board | 15 |
| 4.0.3 | Enabling the Stream Buffer system | 19 |
| 4.0.4 | Importing an existing project | 20 |
| 5 | DONE R5F firmware without an IDE | 20 |
| 6 | RPMMsg echo_test software | 21 |
| 7 | Building micro-ROS as a static library | 21 |
| 8 | Including micro-ROS to the real-time firmware | 23 |
| 9 | micro-ROS adaptation for the firmware | 25 |
| 9.1 | Time functions | 25 |
| 9.2 | Memory allocators | 25 |
| 9.3 | Custom transport layer | 25 |
| 9.4 | Building the updated firmware outside of the IDE | WORK_IN_PROGRESS |
| 9.4 | WORK_IN_PROGRESS | 27 |
| 10 | Loading the firmware | 28 |
| 11 | Loading the firmware using JTAG and Vitis IDE | 29 |
| 11.1 | Testing xsct and running the drivers | 29 |
| 11.1.1 | Drivers | 29 |
| 11.1.2 | dialout access | 29 |
| 11.1.3 | TCL JTAG boot | 29 |
| 11.2 | Vitis IDE classic | 30 |

| | |
|---|----------------------------|
| 12 Running a ROS2 node | 33 |
| 12.1 On the host Linux ("bare-metal") | 34 |
| 12.2 In a container (Docker) | 35 |
| 13 micro-ROS XRCE-DDS Agent | 37 |
| 13.1 Building a XRCE-DDS agent in a Docker | 37 |
| 13.2 Building the Agent version with a modified transport | 37 |
| 13.3 Creating a custom transport agent | WORK_IN_PROGRESS |
| WORK_IN_PROGRESS | 38 |
| 14 Running the ping-pong node | WORK_IN_PROGRESS 39 |
| 15 Monitoring and performances evaluation | WORK_IN_PROGRESS 39 |
| 16 Conclusion & future | WORK_IN_PROGRESS 39 |
| A DTO patch | 40 |
| B Custom tool-chain CMake settings | 43 |
| C Custom Colcon meta settings | 44 |
| D Firmware time functions | 46 |
| D.1 main | 46 |
| D.2 header file | 47 |
| E Firmware memory allocation functions | 48 |
| E.1 main | 48 |
| E.2 header file | 49 |
| F [deprecated] Installing Linux (PetaLinux option) | 49 |
| F.1 General installation | 50 |
| F.2 First login | 52 |
| F.3 Testing openamp "echo-test" | 52 |
| F.4 "echo-test" performance comparison | 53 |
| F.5 Enabling SSH | 54 |
| F.6 Setting up a static IP address for PetaLinux | 54 |
| G References | 55 |

1 Introduction & motivation

This guide will present how to setup and use Xilinx's KRIA board, in particular for running ROS on a host Ubuntu system, as well as for deploying micro-ROS[1] as a firmware on the MCU part of this board's chip.

The use of this device in particular is interesting because of the presence of a CPU comprising both a general purpose ARM core, capable of running a Linux distribution, as well as another ARM core, real-time enabled, capable to run a RTOS. The figure 1 below shows a schematic view of the overall system we are trying to archive.

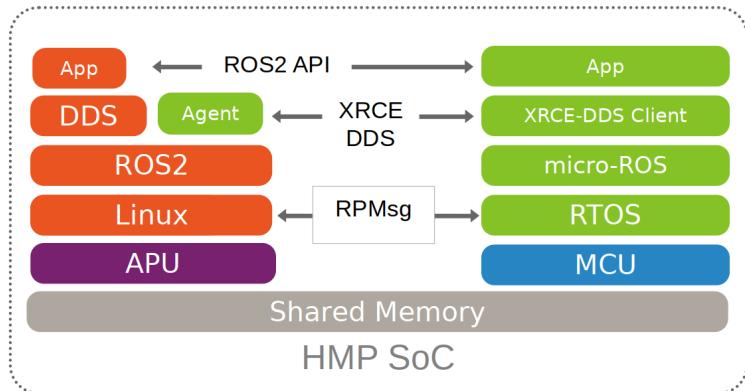


Figure 1: The Linux and ROS2 environment (orange) will communicate with the real-time, FreeRTOS and micro-ROS side (blue) using RPMsg (shared memory).

2 Boot firmware

The goal for the Linux side of the deployment is to have the latest LTS version of Ubuntu up and running.

In order to be able to boot such a newer version of Linux, the boot image of the board must first be updated.

The procedure is available in the official documentation¹, but I will present it step by step here.

2.0.1 Getting the new firmware

A 2022.2 version of the board firmware is best in order to run the latest version of Ubuntu or PetaLinux properly.

The image download link can be obtained at the Atlassian page² on the topic, in the table detailing what version is best suited for which board.

In our case, we want to access this³ page. AMD login will be asked, then download of firmware can be done.

2.0.2 Reaching the board recovery tool

Now the firmware .bin image is available⁴, it is possible to update it using the boards recovery tool. Here are the steps that must be taken in order to reach this tool and update the board:

- Power off the board and hold the firmware update button (FWUEN) when powering back the board.
- Connect the board to your machine via a Ethernet cable. This will obviously cut you internet access, so you should be set for that.
- In the case of the KR260, the bottom right port should be used, as seen in the figure 2 below.

¹<https://docs.xilinx.com/r/en-US/ug1089-kv260-starter-kit/Firmware-Update>

²<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1641152513/Kria+SOMs+Starter+Kits#K26-Boot-Firmware-Updates>

³<https://www.xilinx.com/member/forms/download/design-license-xef.html?filename=BOOT-k26-starter-kit-20230516185703.bin>

⁴Something that looks like BOOT-k26-starter-kit-20230516185703.bin, at the time of writing this section.

- Select the wired network as your connection (must be "forced", since it doesn't have internet access). To do so, you should disable the IPv6, and set the IPv4 as manual, as visible in the figure 3 hereafter.
- Set a fixed IP address for your machine, in the 192.168.0.2/24 range, except the specific 192.168.0.111, which will be used by the board, and 192.168.0.1 which is the DNS and gateway. The netmask and gateway should also be respectively set to 255.255.255.0 and 192.168.0.1.
- Unplug the board power cord again. Hold the firmware update button (FWUEN) when powering back the board.
- Using a web browser on your host machine, access <http://192.168.0.111>. Thou shall now see the interface, as visible on the figure 4 below. If the page struggle to appear, you should try to un-plug and re-plug the Ethernet cable.

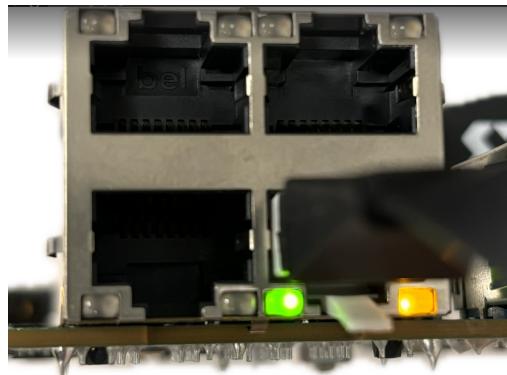


Figure 2: Port to be used for the Ethernet Boot Recovery Tool access on the KR260 board.

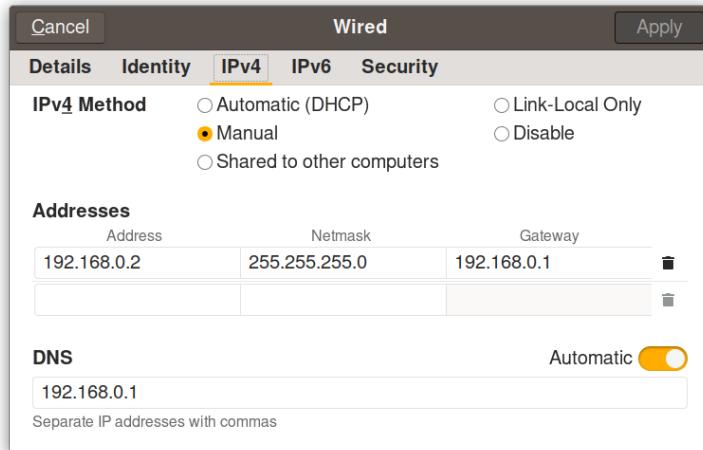


Figure 3: IPv4 settings for accessing the Recovery Tool.

2.0.3 Updating the boot firmware

From this "recovery" page, it is possible to upload the .bin file downloaded previously onto the board using the "Recover Image" section at the bottom right of the page.

The board can be re-booted afterwards.

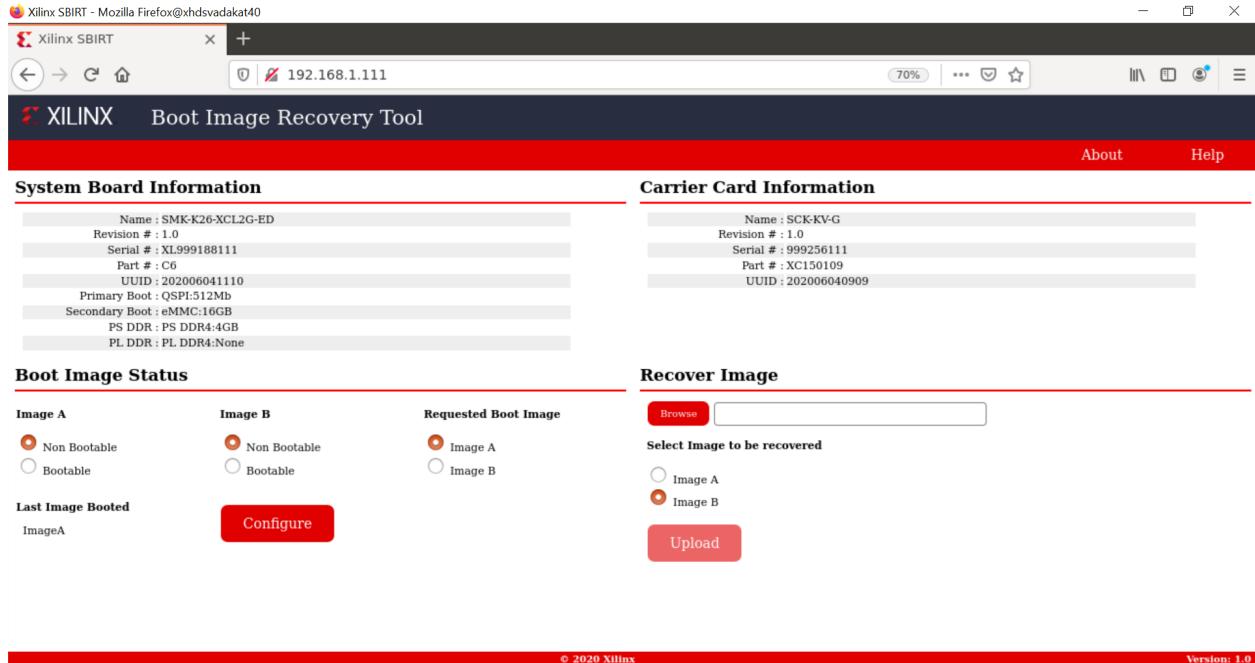


Figure 4: The recovery tool for the board, access from Firefox. We can see board information at the center, and the tools to upload the firmware at the bottom of the page.

3 Installing Ubuntu LTS 22.04

With the boot firmware being up-to-date, we can proceed to install a Linux distribution on our Kria board. The step needed to archive a full installation of Ubuntu LTS 22.04 will be presented in this section⁵. The figure 5 below shows where this operating system sits in the general system we are implementing.

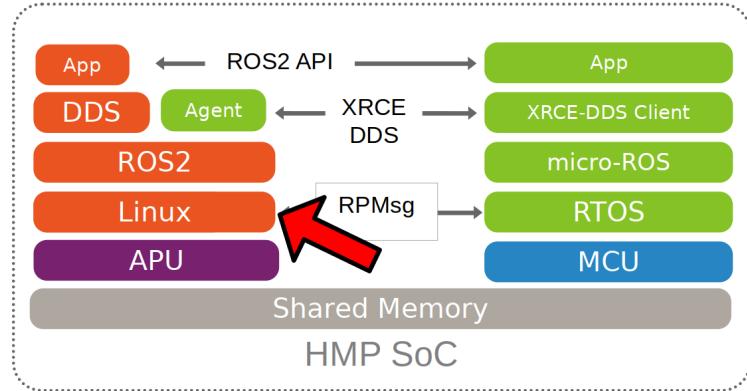


Figure 5: The Linux operating system (red border) runs on the APU (application, general purpose) side of the Kria board CPU. It is the base layer for the ROS2 system.

⁵The same procedure should work for other versions of Ubuntu, as long as they support the Kria board, but for this report and project, only the LTS 22.04 was tested (as of 2023-08-30).

3.1 Preparing and booting a Ubuntu 22.04 media

An official Ubuntu image exists and is provided by Xilinx, allowing the OS installation to be quick and straightforward. Ubuntu is a common and easy to use distribution. Furthermore, it allows to install ROS2 as a package, which is most convenient and will be done later in this guide.

Once the image has been downloaded at Canonical's page we can flash it onto the SD card, with the following instructions.

DANGER: The next part involve the dd command writing on disks!!! As always with the dd command, thou have to be **VERY** careful on what arguments thou give. Selecting the wrong disk will result on the destruction of thy data !! If you are unsure of what to do, seek assistance !

With the image available on thy machine and a SD card visible as /dev/sda device⁶ one can simply run the dd command as follow to write the image to a previously formatted drive (here /dev/sda):

```
1 unxz iot-limerick-kria-classic-desktop-2204-x07-20230302-63.img.xz
2 sudo dd if=iot-limerick-kria-classic-desktop-2204-x07-20230302-63.img \
3     of=/dev/sda status=progress bs=8M && sync
```

Once the SD card is flashed and put back in the board, the micro-USB cable can be connected from the PC to the board. It is then possible to connect to the board in serial with an appropriate tool, for example picocom, as in the following example (the serial port that "appeared" was the /dev/ttyUSB1 in this case, and the 115200 bit-rate is the default value for the board):

```
1 sudo picocom /dev/ttyUSB1 -b 115200
```

In my case, I am using Emacs's serial-term:

```
1 M-x serial-term RET /dev/ttyUSB1 RET 115200 RET
```

The default username / password pair for the very first boot is `ubuntu` and `ubuntu`. You will then be prompted to enter a new password.

Once logged in, it is typically easier and more convenient to connect the board using SSH. When the board is connected to the network, it is possible to know its IP address with the `ip` command; then it is possible to connect to the board with `ssh`, as follow (example, with the first command to be run on the board and the second one on the host PC, both without the first placeholder hostnames):

```
1 kria# ip addr
2
3 host# ssh ubuntu@192.168.4.11
```

3.2 Network and admin setups

This section presents a variety of extra convenience configurations that can be used when setting-up the Kria board.

3.2.1 Static IP address

A static IP can be set by writing the following configuration into your `netplan` configuration file⁷.

The name of the files might vary:

⁶Again, it is critical to be 100\the correct device!

⁷The `chmod` command is used to update the permissions and silence some warnings

```

1 sudo chmod 0600 /etc/netplan/50-cloud-init.yaml
2 sudo nano /etc/netplan/50-cloud-init.yaml

```

You can then set the wanted IP as follow⁸:

```

1 network:
2   renderer: NetworkManager
3   version: 2
4   ethernets:
5     eth0:
6       dhcp4: false
7       addresses:
8         - 192.168.11.107/24
9       routes:
10      - to: default
11        via: 192.168.11.1
12       nameservers:
13         addresses: [192.168.11.1]

```

Finally, the change in settings can be applied as follow:

```

1 sudo netplan apply

```

3.2.2 Proxy and DNS

An issue that can occur when connecting the board to the internet is the conflicting situation with the university proxy. Indeed, as the network at Nanzan University requires to go through a proxy, some DNS errors appeared.

In that case, it might become needed to setup the proxy for the school.

This can be done as follow, by exporting a https base proxy configuration containing you AXIA credentials (this is specific to Nanzan University IT system), then by consolidating the configuration for other types of connections in the bashrc:

```

1 export https_proxy="http://<AXIA_username>:\n<AXIA_psw>@proxy.ic.nanzan-u.ac.jp:8080"
2
3
4 echo "export http_proxy=\"$https_proxy\" >> ~/.bashrc
5 echo "export https_proxy=\"$https_proxy\" >> ~/.bashrc
6 echo "export ftp_proxy=\"$https_proxy\" >> ~/.bashrc
7 echo "export no_proxy=\"localhost, 127.0.0.1,::1\" >> ~/.bashrc

```

Eventually the board can be rebooted in order for the setup to get applied cleanly.

3.2.3 root password

WARNING: Depending on your use-case, the setup presented in this subsection can be a critical security breach as it remove the need for a root password to access the admin functions of the board's Linux.
When in doubt, do not apply this configuration!!

If your board does not hold important data and is available to you only, for test or development, it might be convenient for the sudo tool to not ask for the password all the time. This change can be done by editing the sudoers file, and adding the parameter NOPASSWD at the sudo line:

⁸For the routing part, it is key to have the to with a '-' in front of it; and then the via without, but aligned with the t.

```

1 sudo visudo
2
3 %sudo    ALL=(ALL:ALL) NOPASSWD: ALL

```

Again, this is merely a convenience setup for devices staying at your desk. If the board is meant to be used in any kind of production setup, a password should be set for making administration tasks.

With all of these settings, you should be able to update the software of your board without any issues:

```

1 sudo apt-get update
2 sudo apt-get dist-upgrade
3 sudo reboot now

```

3.2.4 Adding Xilinx specific repositories

The following commands will add PPA repositories that are specific for Xilinx boards using Ubuntu. It is then possible to update the package list and eventually upgrade to some new packages.

```

1 sudo add-apt-repository ppa:ubuntu-xilinx/updates
2 sudo add-apt-repository ppa:xilinx-apps/ppa
3 sudo apt update
4 sudo apt upgrade

```

3.2.5 Purging snap

As the desktop-specific software are not used at all in the case of our project, there are some packages that can be purged in order for the system to become more lightweight.

In particular, the main issue with Ubuntu systems is the forced integration of Snap packages. Here are the command to use in order to remove all of that. These steps take a lot of time and need to be executed in that specific order⁹, but the system fan runs sensibly slower without all of this stuff:

```

1 sudo systemctl disable snapd.service
2 sudo systemctl disable snapd.socket
3 sudo systemctl disable snapd.seeded.service
4
5 sudo snap list #show installed package, remove then all:
6 sudo snap remove --purge firefox
7 sudo snap remove --purge gnome-3-38-2004
8 sudo snap remove --purge gnome-42-2204
9 sudo snap remove --purge gtk-common-themes
10 sudo snap remove --purge snapd-desktop-integration
11 sudo snap remove --purge snap-store
12 sudo snap remove --purge bare
13 sudo snap remove --purge core20
14 sudo snap remove --purge core22
15 sudo snap remove --purge snapd
16 sudo snap list # check that everything is uninstalled
17
18 sudo rm -rf /var/cache/snapd/
19 sudo rm -rf ~/snap
20 sudo apt autoremove --purge snapd

```

⁹The snap packages depends on each others. Dependencies cannot be removed before the package(s) that depends on them, thus the specific delete order.

```

21
22 # check once more that there is no more snap on the system
23 systemctl list-units | grep snapd

```

3.2.6 Other unused heavy packages

Some other pieces of software can safely be removed since the desktop is not to be used:

```

1 sudo apt-get autoremove --purge yaru-theme-icon \
2   fonts-noto-cjk yaru-theme-gtk vim-runtime \
3   ubuntu-wallpapers-jammy humanity-icon-theme
4
5 sudo apt-get autoclean
6 sudo reboot now

```

3.2.7 Slow boot services to disable

These packages (in particular the first one) are taking up a LOT of time at boot while providing no benefits¹⁰. It is possible to disable them as follow:

```

1 sudo systemctl disable systemd-networkd-wait-online.service
2 sudo systemctl disable NetworkManager-wait-online.service
3 sudo systemctl disable cups.service
4 sudo systemctl disable docker.service
5 sudo systemctl disable containerd.service
6 sudo systemctl disable cloud-init-local.service

```

Additional, potentially unused services can be found using the very handy command:

```

1 sudo systemd-analyze blame

```

3.2.8 Jupyter notebook setup

Here are some instruction on how to install and setup Jupyter on a KRIA board, accessing it remotely and using it for making data analysis.

The following commands will set the required packages and install Jupyter itself¹¹:

```

1 sudo apt-get update && sudo apt-get install python3 python3-pip python3-venv python3-virtualenv
2
3 virtualenv myjupyter
4 source ./myjupyter/bin/activate
5 python3 -m pip install jupyter pandas numpy matplotlib scipy
6
7 sudo reboot now

```

Then in a terminal on your host machine (not on the KRIA board), you can run the following command¹² to bind local ports:

```

1 ssh -L 8888:localhost:8888 ubuntu@192.168.11.107

```

¹⁰The CUPS and Docker services will be activated when used instead of during boot time.

¹¹Alongside other packages useful for data analysis, such as pandas or numpy.

¹²In this example, the full username@IP is used, but a .ssh/config is also usable.

Then on the opened SSH shell to the KRIA board:

```
1 source ./myjupyter/bin/activate
2 jupyter notebook
```

From there, it is possible to use the displayed URL (something that looks like `http://localhost:8888/tree?token`) to access the remote Notebook system from a local web browser. It is possible to do so with `localhost` since we have the `ssh` port map connection going on.

Eventually creating Notebooks and stuff, it is possible to obtain a situation like shown in the figure 6 below.

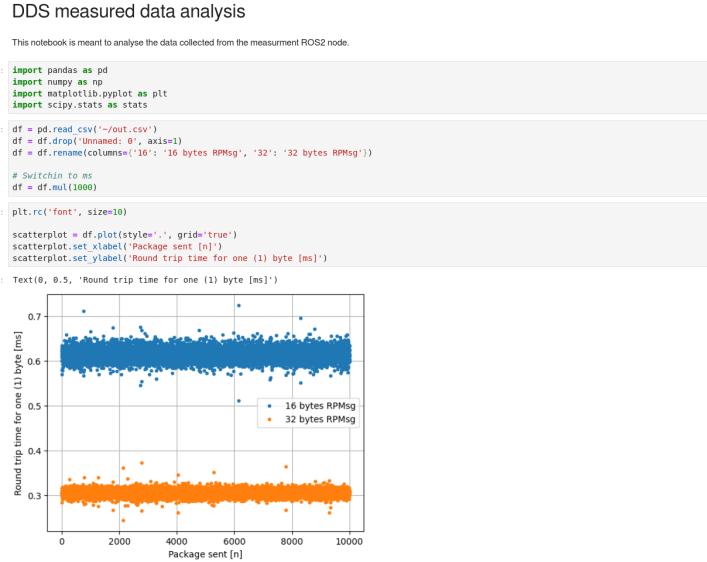


Figure 6: A test Jupyter Notebook for CSV data analysis.

3.2.9 Enabling remoteproc with Device-Tree Overlay patching

One of the advantage of this Kria board, as cited previously, is the presence of multiple types of core (APU, MCU, FPGA) on the same chip.

The part in focus in this guide is the usage of both the APU, running a Linux distribution and ROS2; and the MCU, running FreeRTOS and micro-ROS. Online available guides¹³ , ¹⁴ also provide information on how to deploy these types of systems and enabling `remoteproc` for the Kria board, but this guide will show a step-by-step, tried process to have a heterogeneous system up and running.

The communication between both side is meant to be done using shared memory, but some extra setup is required in order to be running the real-time firmware, in particular for deploying micro-ROS on it.

As a first step in that direction, this section of the report will present how to setup and use as an example firmware that utilizes the `remoteproc` device in Linux in order to access shared memory and communicate with the real-time firmware using the RPMsg system.

The communication system and interaction from the Linux side towards the real-time capable core is not enabled by default within the Ubuntu image provided by Xilinx.

In that regard, some modification of the device tree overlay (DTO) is required in order to have the `remoteproc` system starting.

Firstly, we need to get the original firmware device tree, converted into a readable format (DTS):

¹³A slideshow (JP) from Fixstar employees presents how to use the device tree to enable the communication between the cores.

¹⁴A blog post (JP) shows all major steps on how to enable the `remoteproc`.

```
1 sudo dtc /sys/firmware/fdt 2> /dev/null > system.dts
```

Then, a custom-made patch file can be downloaded and applied. This file is available at the URL visible in the command below but also in this report appendix A.

```
1 wget
→ https://gitlab.com/sunoc/xilinx-kria-kv260-documentation/-/raw/7a8f7c4e66e09b9d66aba8d2e08fc446ff485ca8/src/system.patch
2
3 patch system.dts < system.patch
```

As for the board to be able to reserve the correct amount of memory with the new settings, some cma kernel configuration is needed¹⁵:

```
1 sudo nano /etc/default/flash-kernel
2
3 LINUX_KERNEL_CMDLINE="quiet splash cma=512M cpuidle.off=1"
4 LINUX_KERNEL_CMDLINE_DEFAULTS=""
5 sudo flash-kernel
```

Now the DTS file has been modified, one can regenerate the binary and place it on the /boot partition and reboot the board:

```
1 dtc -I dts -O dtb system.dts -o user-override.dtb
2 sudo mv user-override.dtb /boot/firmware/
3 sudo reboot now
```

After rebooting, you can check the content of the remoteproc system directory, and a remoteproc0 device should be visible, as follow:

```
1 ls /sys/class/remoteproc/
2 # remoteproc0
```

If it is the case, it means that the patch was successful and that the remote processor is ready to be used!

¹⁵The overlapping memory will not prevent the board to boot, but it disables the PWM for the CPU fan, which will then run at full speed, making noise.

3.2.10 Installing Docker

It is possible to have a version of Docker installed simply by using the available repository, but since we are on Ubuntu, a PPA is available from Docker in order to have the most up-to-date version.

Following the official documentation, the following steps can be taken to install the latest version of Docker on a Ubuntu system. The last command is meant to test the install. If everything went smoothly, you should see something similar to what is presented in the figure 7 below, after the commands:

```
1 sudo apt-get update
2 sudo apt-get install ca-certificates curl
3 sudo install -m 0755 -d /etc/apt/keyrings
4 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
5     sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
6
7 sudo chmod a+r /etc/apt/keyrings/docker.gpg
8
9 echo \
10   "deb [arch="$(dpkg --print-architecture)" \
11     signed-by=/etc/apt/keyrings/docker.gpg] \
12       https://download.docker.com/linux/ubuntu \
13     '$(. /etc/os-release && \
14       echo "$VERSION_CODENAME")' stable" | \
15   sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
16
17 sudo apt-get update
18 sudo apt-get install docker-ce docker-ce-cli \
19   containerd.io docker-buildx-plugin docker-compose-plugin
20 sudo usermod -aG docker $USER
21 newgrp docker
22
23 docker run hello-world
```

```
ubuntu@kria:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
70f5ac315c5a: Pull complete
Digest: sha256:fc6cf906cbfa013e80938cdf0bb199fbdbb86d6e3e013783e5a766f50f5dbce0
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 7: The return of a successful run of the hello world test Docker container.

3.2.11 Adding a swap partition

This part is very optional, in particular as it might slow down a bit the boot time of the board (~2s), however it might become handy to have swap memory available to avoid system failure under heavy use.

This whole procedure must be done externally, with the board system SD card mounted on a host PC as an external volume. As it is highly platform dependant, I will not give a detailed explanation on how to do it, yet here are the key points that should be done:

- Shutdown the Kria board, take out the SD card and put it in a host machine.
- Make sure the disk is visible.
- Make sure all volumes are **unmounted**.
- Resize the main root partition (**not** the boot) so a space the size of the wanted swap is free **after** the partition. You'd want something around 1GB.
- In the empty space, create a new partition, which type is "linux swap".
- Find and take note of the UUID of the new partition. This is useful hereafter.
- sync
- Un-mount everything, eject SD card.
- Put the SD card back in the Kria.
- Boot back to Ubuntu.

Going back on the Kria board Ubuntu after boot, the `/etc/fstab` file can be updated as follow, modulo your actual UUID for the newly created partition, to enable swap at boot time.

```
1 sudo -s
2 echo "UUID=8b13ed05-a91d-4x50-a44a-e654a0c67a2c none    swap      sw      0      0" >> /etc/fstab
3 reboot now
```

3.2.12 TODO Using a PetaLinux kernel in Ubuntu

4 Setting up a Vitis IDE project

As visible in the official documentation¹⁶, this section will present the required steps for building a new firmware for the R5F core of our Kria board.

The goal here is to have a demonstration firmware running, able to use the RPMsg system to communicate with the Linux APU. The figure 8 below shows where the real-time firmware is positioned in the global project.

4.0.1 Setting up the IDE

Xilinx's Vitis IDE is the recommended tool used to build software for the Xilinx boards. It also include the tools to interact with the FPGA part, making the whole software very large (around 200GB of disk usage).

However, this large tool-set allows for a convenient development environment, in particular in our case where some FreeRTOS system, with many dependencies is to be build.

The installer can be found on Xilinx download page¹⁷. You will need to get a file named something like `Xilinx_Unified_2022.2_1014_8888_Lin64.bin`¹⁸.

¹⁶<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1837006921/OpenAMP+Base+Hardware+Configurations/#Build-RPU-firmware>

¹⁷<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html>

¹⁸The name of the installer binary file might change as a new version of the IDE is release every year or so.

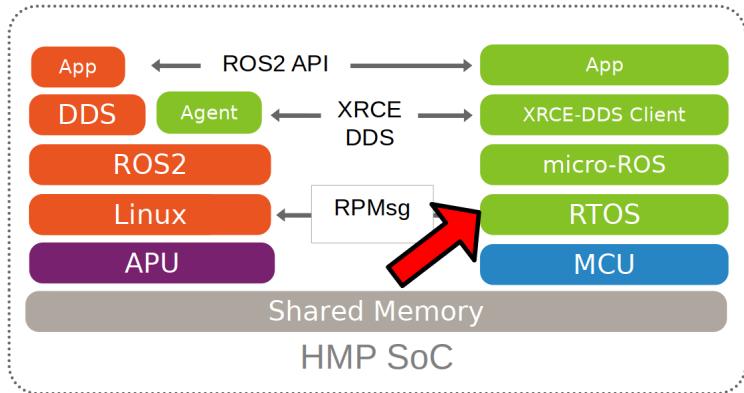


Figure 8: The FreeRTOS firmware and it's application (red border) are running on the real-time capable side of the Kria CPU. A micro-ROS application is shown here, but any real-time firmware will be deployed in the same way.

Vitis IDE installer is compatible with versions of Ubuntu, among other distributions, but not officially yet for the 22.04 version. Furthermore, the current install was tested on Pop OS, a distribution derived from Ubuntu. However, even with this more unstable status, no major problems were encountered with this tool during the development stages.

This guide will present a setup procedure that supposedly works for all distributions based on the newest LTS from Ubuntu. For other Linux distributions or operating system, please refer to the official documentation.

1. Dependencies & installation Some packages are required to be installed on the host system in order for the installation process to happen successfully:

```

1 sudo apt-get -y update
2
3 sudo apt-get -y install libncurses-dev \
4     ncurses-term \
5     ncurses-base \
6     ncurses-bin \
7     libncurses5 \
8     libtinfo5 \
9     libncurses5-dev \
10    libncursesw5-dev

```

Once this is done, the previously downloaded binary installer can be executed:

```
1 ./Xilinx_Unified_2022.2_1014_8888_Lin64.bin
```

If it is not possible to run the previous command, make the file executable with the `chmod` command:

```
1 sudo chmod +x ./Xilinx_Unified_2022.2_1014_8888_Lin64.bin
```

From there you can follow the step-by-step graphical installer. The directory chosen for the rest of this guide for the Xilinx directory is directly the `$HOME`, but the installation can be set elsewhere if needed.

WARNING: This whole procedure can take up to multiple hours to complete and is prone to failures (regarding missing dependencies, typically), so your schedule should be arranged accordingly.

2. Platform configuration file With a Xilinx account, a .bsp archive can be downloaded for the target platform¹⁹.

Once the file is downloaded, the following commands allows to "un-tar" it, making the needed .xsa file accessible via a file explorer.

```

1 tar xvfvz xilinx-kr260-starterkit-v2022.2-10141622.bsp
2 ls xilinx-kr260-starterkit-2022.2/hardware/xilinx-kr260-starterkit-2022.2/

```

3. [DEPRECATED] Platform configuration file generation In order to have the libraries and configurations in the IDE ready to be used for our board, we need to obtain some configuration files that are specific for the Kria KV260, as presented in the Xilinx guide for Kria and Vitis²⁰.

A Xilinx dedicated repository²¹ is available for us to download such configurations, but they required to be built.

As for the dependencies, Cmake, tcl and idn will become needed in order to build the firmware. Regarding idn, some version issue can happen, but as discussed in a thread on Xilinx's forum²², if libidn11 is specifically required but not available (it is the case for Ubuntu 22.04), creating a symbolic link from the current, 12 version works as a workaround.

Here are the steps for installing the dependencies and building this configuration file:

```

1 sudo apt-get update
2 sudo apt-get install cmake tcl libidn11-dev \
3     libidn-dev libidn12 idn
4 sudo ln -s /usr/lib/x86_64-linux-gnu/libidn.so.12 \
5     /usr/lib/x86_64-linux-gnu/libidn.so.11
6
7 cd ~/Xilinx
8 git clone --recursive \
9     https://github.com/Xilinx/kria-vitis-platforms.git
10 cd kria-vitis-platforms/k26/platforms
11 export XILINX_VIVADO=/home/$USER/Xilinx/Vivado/2022.2/
12 export XILINX_VITIS=/home/$USER/Xilinx/Vitis/2022.2/
13 make platform PLATFORM=k26_base_starter_kit

```

4.0.2 Setting up and building a new project for the Kria board

With the platform configuration files available, we can now use the IDE to generate a new project for our board. The whole process will be described with screen captures and captions.

¹⁹<https://www.xilinx.com/member/forms/download/xfef.html?filename=xilinx-kr260-starterkit-v2022.2-10141622.bsp>

²⁰https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/build_vitis_platform.html?highlight=xsa

²¹<https://github.com/Xilinx/kria-vitis-platforms>

²²https://support.xilinx.com/s/question/0D52E00006jrzsySAQ/platform-project-cannot-be-created-on-vitis?language=en_US

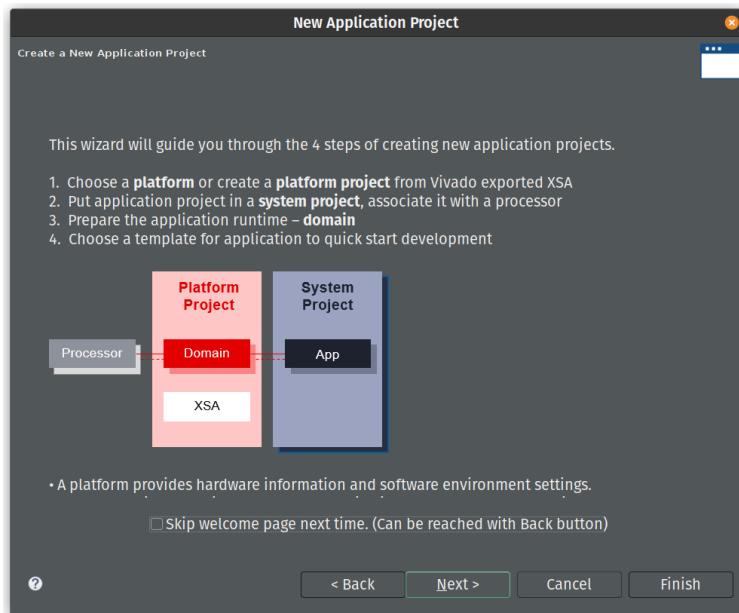


Figure 9: We are starting with creating a "New Application Project". You should be greeted with this wizard window. Next.

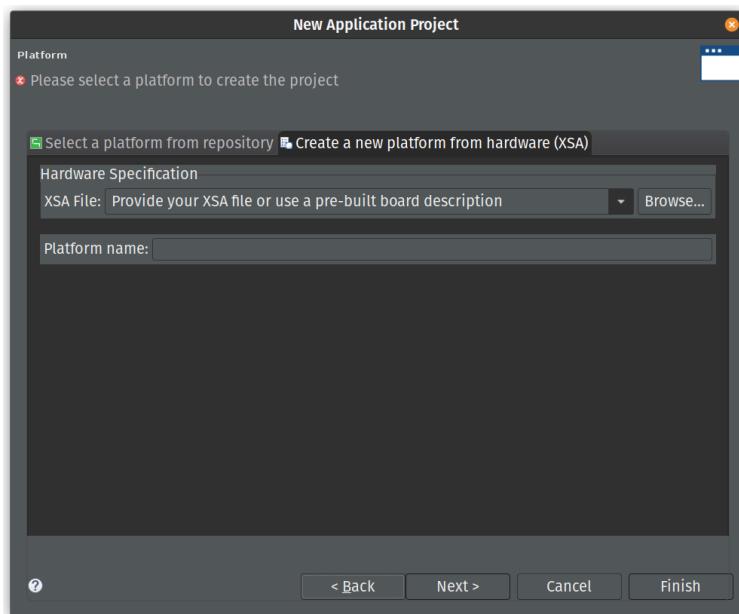


Figure 10: For the platform, we need to get our build Kria configuration. In the "Create a new platform" tab, click the "Browse..." button.

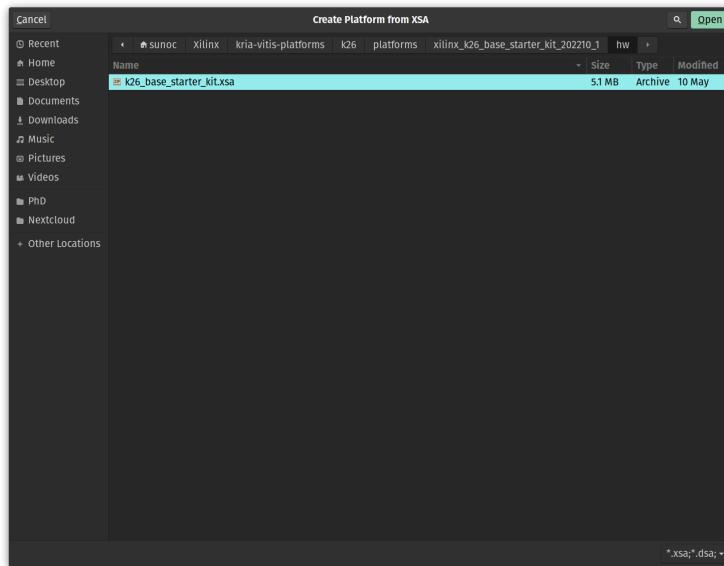


Figure 11: In the file explorer, we should navigate in the "k26" directory, where the configuration file was build. From here we are looking for a ".xsa" file, located in a "hw" directory, as visible.

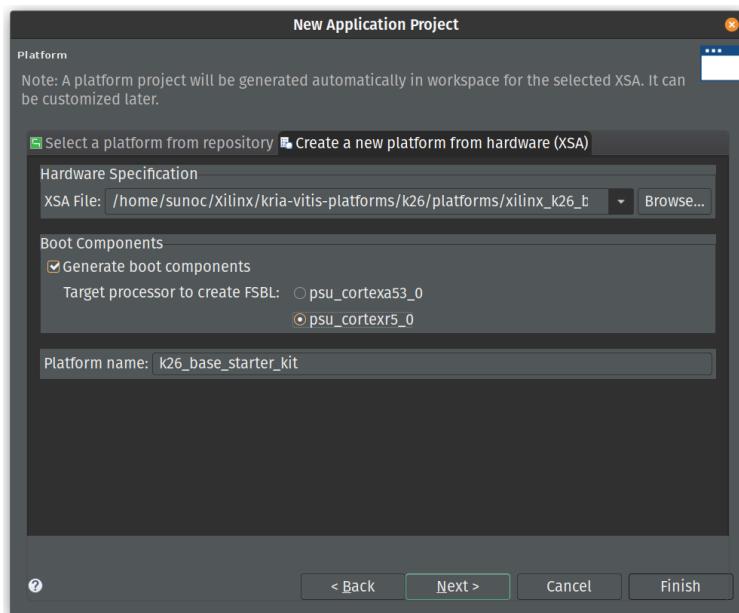


Figure 12: With the configuration file loaded, we can now select a name for our platform, but most importantly, we have to select the "psu Cortex5 0" core as a target. The other, Cortex 53 is the APU running Linux.

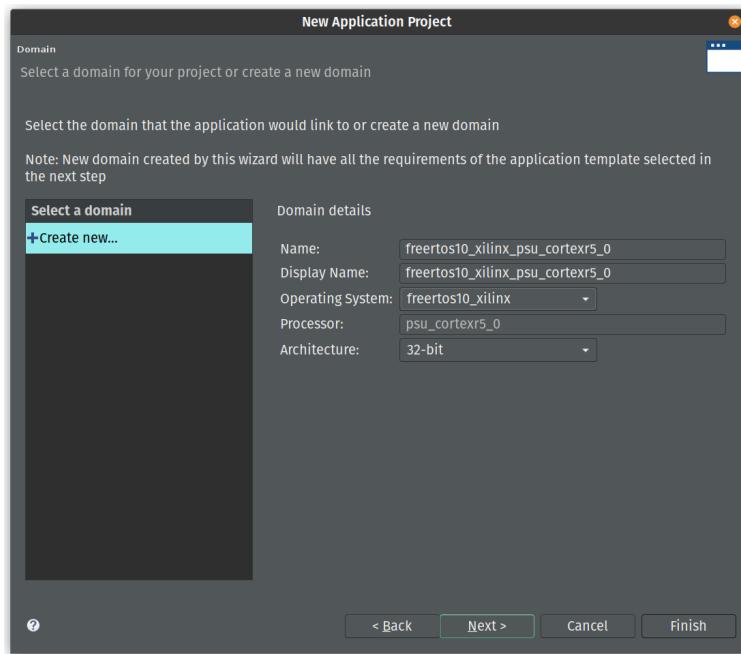


Figure 13: Here, we want to select "freertos10 xilinx" as our Operating System. The rest can remain unchanged.

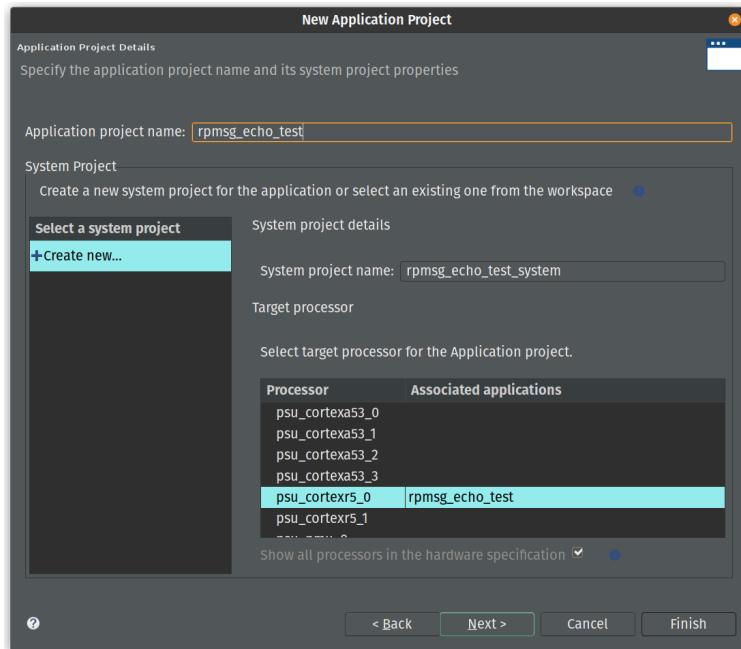


Figure 14: In this next window, we can give a name to our firmware project. It is also critical here to select the core we want to build for. Once again, we want to use the "psu cortexr5 0".

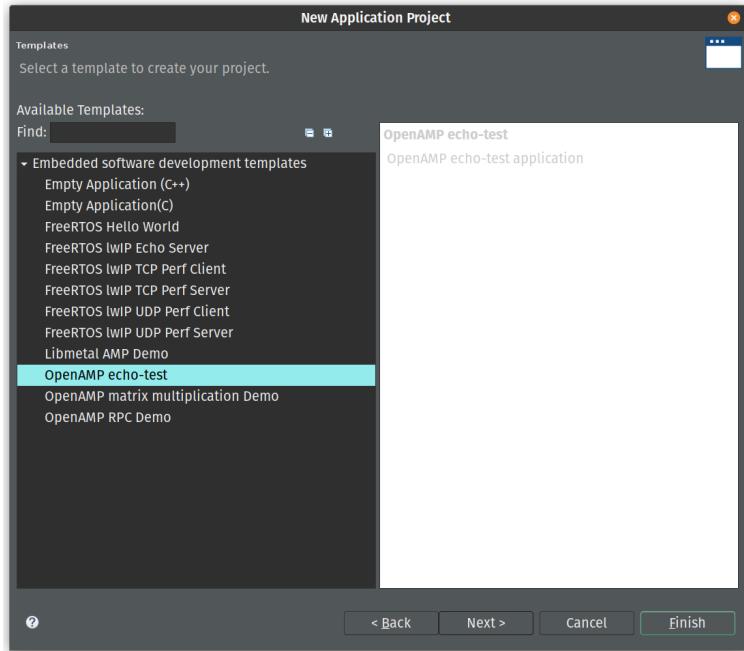


Figure 15: Finally, we can select the demonstration template we are going to use; here we go with "OpenAMP echo-test" since we want to have some simple try of the RPMsg system. Finish.

In the Xilinx documentation, it is made mention of the addresses setting that should be checked in the `script.ld` file. The values in the figure 16 below look different from what could be set in the DTO for the Linux side, but they appear to work for the example we are running, including the new DTO patch without overlapping memory:

| Available Memory Regions | | |
|---------------------------------|--------------|------------|
| Name | Base Address | Size |
| psu_ddr_S_AXI_BASEADDR | 0x3ED00000 | 0x00140000 |
| psu_ocm_ram_1_S_AXI_BASEADDR | 0xFFFF0000 | 0x00010000 |
| psu_r5_tcm_ram_0_S_AXI_BASEADDR | 0x00000000 | 0x00015000 |
| psu_r5_tcm_ram_1_S_AXI_BASEADDR | 0x00020000 | 0x00015000 |

Figure 16: `script.ld` memory configuration for the firmware memory setup. The same file is available as a whole in this repository's `src` directory.

Once your example project is built and you have a `.elf` file available, you can jump directly in further sections to see how to deploy and use your firmware.

The section in between will present setup specifically needed for micro-ROS.

4.0.3 Enabling the Stream Buffer system

This is a subpart in the general configuration in the project related to some specific functions for FreeRTOS threads messaging system, however, this point in particular created so much pain I needed to include in early in this guide for not to forget about it and keeping a clear track on how to enable this setting.

Indeed, two settings need to be enabled in order to be able to call functions such as `xMessageBufferCreate`, useful when working with tasks in FreeRTOS, as visible in the figure 17 below:

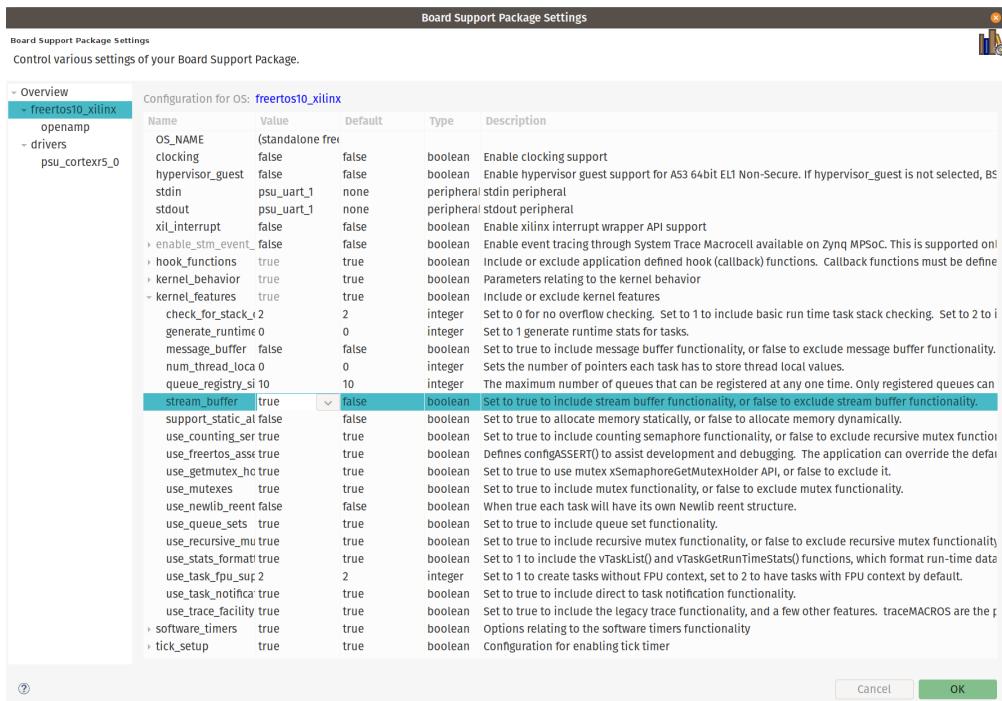


Figure 17: Enabling Stream Buffer in the Vitis IDE setting: this is a setting that can be found in the "platform.spr" element of your project (the platform, not the firmware project itself). From that file, you can access the settings with the button "Modify BSP Settings", and then as visible, in the tab freertos10_xilinx, it is needed to toggle here the `stream_buffer` setting in the `kernel_features`, from the default "false" to "true".

The second setting is useful in the case when a buffer callback function is used, such as `xMessageBufferCreateWithCallback`. In that case, you must include `#define configUSE_SB_COMPLETED_CALLBACK 1` on the top of your header file (in our project, this will happen in the `microros.h` header file), before `#include "FreeRTOS.h"` in order to override the setting from this include.

4.0.4 Importing an existing project

5 DONE R5F firmware without an IDE

- State "DONE" from "TODO" [2024-02-13 Tue 15:28]

The Section 4 was merely a presentation on how bootstrap a brand new development project using Xilinx provided templates in their crap IDE.

Once the project exist, one can simply compile it again using the generated `Makefile`.

The information on how to handle the project are available directly in the repository of said project²³ and it's various branches for ongoing feature development.

²³https://gitlab.com/sunoc/libmicroros_kv260

6 RPMsg echo_test software

In order to test the deployment of the firmware on the R5F side, and in particular to test the RPMsg function, we need some program on the Linux side of the Kria board to "talk" with the real-time side.

Some source is provided by Xilinx to build a demonstration software that does this purpose: specifically interact with the demonstration firmware.

Here are the steps required to obtain the sources, and build the program.

As a reminder, this is meant to be done on the Linux running on the Kria board, NOT on your host machine !

```
1 git clone https://github.com/Xilinx/meta-openamp.git
2 cd meta-openamp
3 git checkout xlnx-rel-v2022.2
4 cd ./recipes-openamp/rpmmsg-examples/rpmmsg-echo-test
5 make
6 sudo ln -s $(pwd)/echo_test /usr/bin/
```

Once this is done, it is possible to run the test program from the Kria board's Ubuntu by running the `echo_test` command.

7 Building micro-ROS as a static library

In this section, the goal is to build the micro-ROS library in order to be able to integrate its functions into our Cortex R5F firmware.

All of this should be done via cross-compiling on a host machine, however it is most common in the guides about micro-ROS to build the firmware and libraries within a Docker, so we can have access of the ROS environment without installing it permanently.

One can simply run this command to summon a ROS2 Docker²⁴ with the wanted version, but first we also need to check the cross-compilation tools.

We are downloading the latest `arm-none-eabi` gcc compiler directly from the ARM website.

The cross-compilation tool can then be extracted, set as our `toolchain` variable, then passed as a parameter when creating the Docker container:

```
1 pushd /home/$USER/Downloads
2 wget
3   → https://developer.arm.com/-/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
4 tar -xvf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
5 popd
6
7
8
9 toolchain="/home/$USER/Downloads/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/"
10
11
12 docker run -d --name ros_build -it --net=host \
    --hostname ros_build \
    -v $toolchain:/armr5-toolchain \
    --privileged ros:iron
```

Now the container named `ros_build` was created, it is possible to "enter" it, and having access to the tools in it by running the following command that will open a bash shell in said container:

```
1 docker exec -it ros_build bash
```

²⁴If Docker is not set up on your machine, you can follow the guide on the official website. When you can successfully run the "hello-world" container, you are good to go.

Now we are in the ROS2 container, we can build the micro-ROS firmware as presented in the dedicated micro-ROS guide:

```

1 echo 127.0.0.1 $HOSTNAME >> /etc/hosts
2 sudo apt update
3 sudo apt-get -y install python3-pip \
4     wget \
5     nano
6
7 . /opt/ros/$ROS_DISTRO/setup.bash
8
9 mkdir microros_ws
10 cd microros_ws
11 git clone -b $ROS_DISTRO \
12     https://github.com/micro-ROS/micro_ros_setup.git \
13     src/micro_ros_setup
14
15 sudo rosdep fix-permissions && \
16     rosdep update && \
17     rosdep install --from-paths src --ignore-src -y
18
19 colcon build
20 . ./install/local_setup.bash
21
22 ros2 run micro_ros_setup create_firmware_ws.sh generate_lib

```

From that point, we will need some extra configuration files for our Cortex R5F.

Both configuration files²⁵ will be downloaded from my repository; we also are going to copy the cross-compiler into the microros workspace, then we can build the library with the following ros2 command:

```

1 wget https://gitlab.com/sunoc/xilinx-kria-kv260-documentation/-/raw/main/src/custom_r5f_toolchain.cmake
2
3 wget https://gitlab.com/sunoc/xilinx-kria-kv260-documentation/-/raw/main/src/custom_r5f_colcon.meta
4
5 cp -r /armr5-toolchain/ $(pwd)/firmware/toolchain && \
6     export PATH=$PATH:$(pwd)/firmware/toolchain/bin
7
8 ros2 run micro_ros_setup build_firmware.sh \
9     $(pwd)/custom_r5f_toolchain.cmake \
10    $(pwd)/custom_r5f_colcon.meta

```

²⁵Both files are also visible in the appendixes B and C the end of this report.}

8 Including micro-ROS to the real-time firmware

Now we have a Vitis demonstration project available and the `libmicroros` static library available, we can combine both by including this library into our Kria project.

On the host machine running the IDE, we can download the static library and the include files from the Docker builder. Here, we assume your Vitis IDE workspace sits in your home directory, at `~/workspace`, and that the Docker container is named `ros_build`:

```
1 mkdir /home/$USER/workspace/microros_lib
2
3 docker cp ros_build:/microros_ws/firmware/build/ \
4     libmicroros.a /home/$USER/workspace/microros_lib/
5
6 docker cp ros_build:/microros_ws/firmware/build/include \
7     /home/$USER/workspace/microros_lib/
```

Many parameters are available to be set up in the IDE for the compilation tool-chain, but the figures 18 and 19 below will show you a setup that worked to have the IDE to recognize the include files and to be able to use them for compiling the firmware.

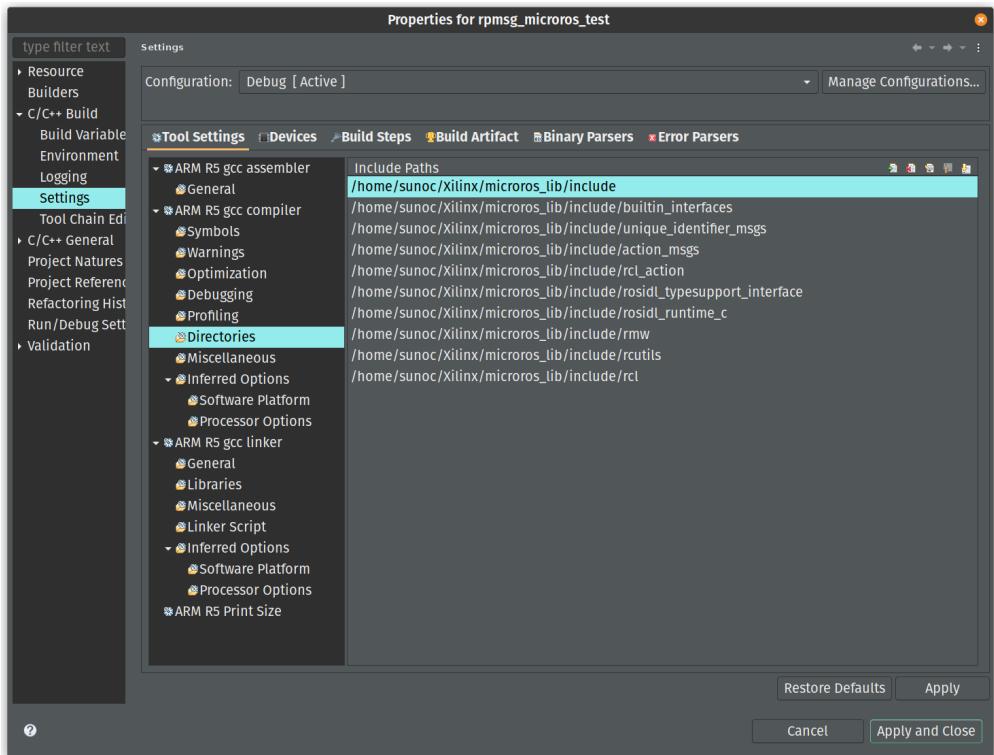


Figure 18: Firstly, in the "C/C++ Build" settings of your firmware project, under the "Settings" menu, you should find the gcc compiler "Directories". In here you should add the "include" directory of your library. Be careful however, if your include files are in a second layer of directory (as it is the case for `libmicroros`) you will need to include each sub-directory individually, as visible in this figure.

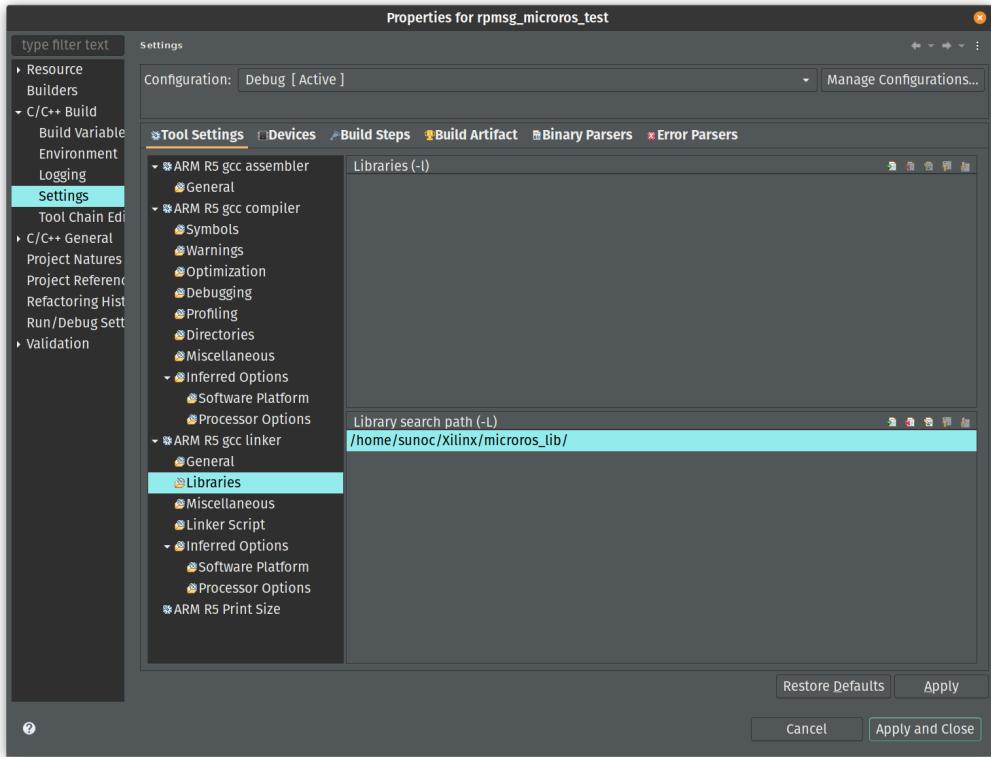


Figure 19: Secondly, in the gcc linker "Libraries", you can add the top level directory of your library. In our case, it is the directory that contains both the "include" directory added earlier, and also the "libmicroros.a" file.

With both of these setup in your project and as a minimal test to see if the setup was made correctly, you should be able to include the following micro-ROS libraries into your project:

```

1 #include <rcl/rcl.h>
2 #include <rcl/error_handling.h>
3 #include <rclc/rclc.h>
4 #include <rclc/executor.h>
```

The details for the inclusions and the use-case of the library will depend on the implementation of the firmware itself.

But in general, as the firmware is successfully built and an .elf file is available and can be uploaded as a firmware to the Kria board (or any remote server accessible through SSH, for that matter) with the following command²⁶:

```

1 scp /home/sunoc/workspace/rpmsg_pingpong_microros_lib/\
2   Debug/rpmsg_pingpong_microros_lib.elf  ubuntu@192.168.1.10:/home/ubuntu/
```

²⁶Note that in that case, we are retrieving the binary for a project named rpmsg_pingpong_microros_lib. Your own path will vary depending on how your project was named in the first place. Obviously, the SSH connection parameters will also be specific to your case. The ubuntu@192.168.1.10 set here are merely an example.

9 micro-ROS adaptation for the firmware

Beyond the inclusion of the library itself, actually using the micro-ROS system within an external project require more than just importing the needed functions.

Indeed, if you would be just adding the various function for sending messages to the general ROS2 network, you would face issues with four key aspects. These are presented in the following dedicated sub-sections.

9.1 Time functions

As micro-ROS can be used on a variety of board, it does not understand by itself what time functions are meant to be used.

In that regard, some API-style function are being used in the library and it is then needed for the person using a new board to implement these function inner working using the board own time-related function calls.

In particular for this part, the `clock_gettime` function is key, and could simply be implemented with some FreeRTOS time functions.

The end result for these implementation are visible in the appendix D, and can be reused as-if for the Kria board setups.

9.2 Memory allocators

Similarly to the time function, it is required to re-implement some form of memory allocating functions in order for the library to be able to work with such functions in a formalized way.

As for now, the current version of the allocator function can be seen in the appendix E, but the current setup is not completely "clean", some further formatting, test and modification will be needed.

9.3 Custom transport layer

This part is the key translation layer that needs to happen in order for the DDS system from the micro-ROS library to be using the communication channel we want it to.

A problem that had to be figured out lives in the fact that the operation of micro-ROS DDS and the board's RPMsg communication system does not operate in the same fashion.

The former expects to have four functions ("open", "read", "write" and "close") that can be called and used by the main system, while the latter relies on FreeRTOS callback system, waiting on the service interrupt routine to be triggered by an incoming message.

This situation meant that we cannot simply translate the communication layers from one to another: a non-blocking polling and buffer system needed to be put into place. The proposed solution that was implemented and that is currently being tested is shown and detailed in the figure 20 below.

The next figure 21 shows a more visual representation of the two tasks and the functions used in them.

A version of this firmware is available at my gitlab repository²⁷. This firmware was built using Xilinx's IDE, which setup was presented in the section 4.0.1, however it was tested and given the provided Makefile system, it is possible to modify and rebuild the firmware without this specific tools, as long as the compiler is installed correctly.

Beyond the general two-tasks behavior of the figure 20, here are the main steps of the execution of the firmware:

- During the RPMsg init phase (RPMsg task), a "hello" message is exchanged with the Linux side to confirm the OpenAMP system is also ready there.
- Memory allocations and custom transport function are set in the micro-ROS task.
- The micro-ROS and its `rclc` system run extensive initialization tasks to:
 - Initialize the support system
 - Initialize the support node

²⁷micro-ROS ping pong test firmware repository: https://gitlab.com/sunoc/libmicroros_kv260

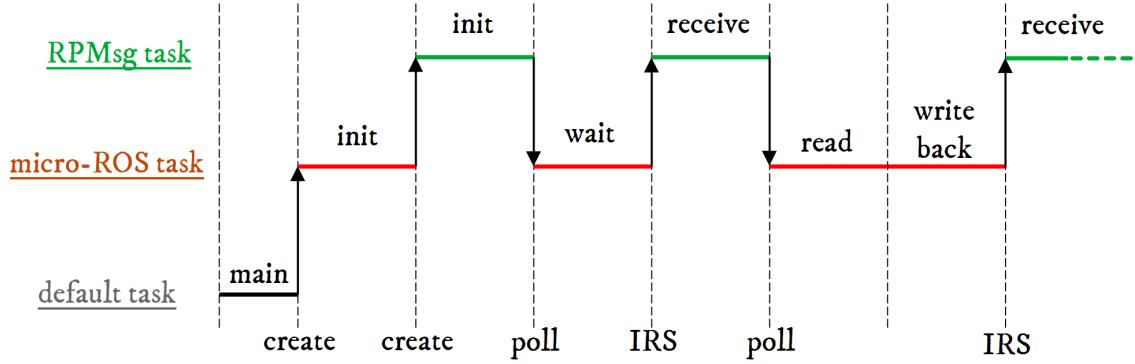


Figure 20: Two tasks are being run concurrently in order to manage the communication situation, with binary semaphore-based lock-unlock system.

The role of the micro-ROS task (red) is to make the four functions ("open", "read", "write" and "close") available and running the actual software function. In this example, polling the read function and writing back when something is receive (ping-pong function). The use of the `rpmsg_send()` function is done directly from the micro-ROS task, bypassing the RPMsg task in this situation.

In the libmicroros implementation currently being developed, the micro-ROS task holds all the DDS and micro-ROS system, including the mentioned allocators function.

The RPMsg task (green) is used to firstly set the RPMsg communication with the Linux system, then it stays locked until the ISR (interrupt service routine) is triggered by an incoming message. The message is then passed to the micro-ROS task using a buffer.

When a shutdown signal is received from the Linux, both functions will gracefully close and are getting killed.

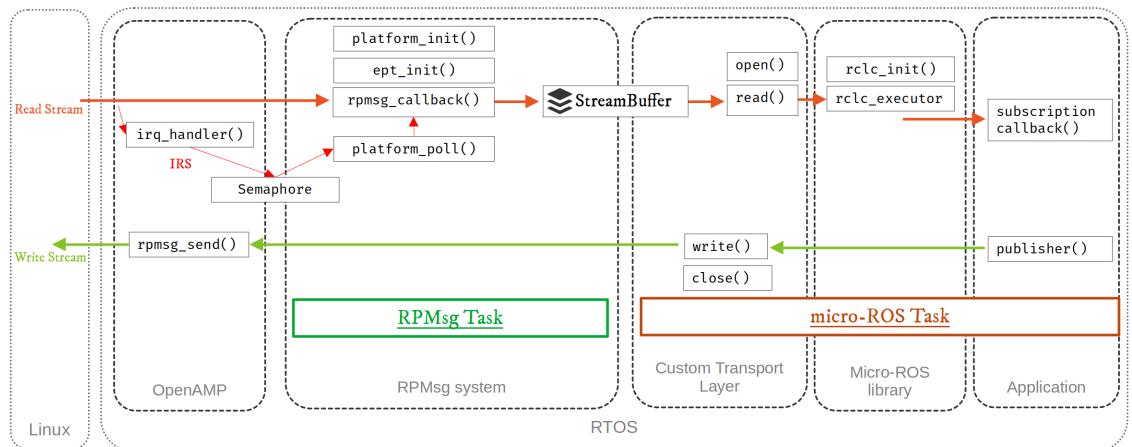


Figure 21: Functions architecture for the Client firmware.

- Initialize the publishers
- Initialize the subscribers
- Finally, the ping-pong function become effective with the micro-ROS task polling to receive some message and sending it back to the sender.

9.4 Building the updated firmware outside of the IDE

WORK_IN_PROGRESS

While the use of the IDE presented in the previous sections is much needed in order to first create a RPMsg-ready base firmware, it might be inconvenient having to carry this extremely heavy tool for any modification or rebuild of the firmware.

Fortunately, the Vitis IDE projects have a very handy function: they general a pure Makefile, with all the needed dependencies and path for library set. This make system can be used externally, however, and as explained previously, two conditions must be fulfilled:

- The cross-compiler needs to be in the \$PATH.
- Standard building tools must be installer, in particular git in order to obtain the repository, and make, obviously.

```

1 cd /tmp
2 git clone https://gitlab.com/sunoc/libmicroros_kv260.git
3 cd libmicroros_kv260/Debug
4 make

```

From there, you will have the binary `rpmsg_pingpong_microros.lib.elf` file created and ready to be deployed on the KRIA board.

10 Loading the firmware

Having a version of our .elf firmware (with or without the included micro-ROS library) built and loaded onto our Kria's Linux, we want to load and run it on the Cortex micro-controller side.

As a reminder, the firmware can be loaded from the host machine IDE workspace to the Kria board through SSH using the following command:

```
1 scp /home/sunoc/workspace/rpmsg_pingpong_microros_lib/\
2   Debug/rpmsg_pingpong_microros_lib.elf  ubuntu@192.168.1.10:/home/ubuntu/
```

The following instructions will show how to use this binary file, and in particular how to upload and start the firmware on the R5F real time core from the Linux user-space²⁸, to test a basic RPMsg setup²⁹:

```
1 sudo -s
2 mv image_echo_test /lib/firmware
3 echo image_echo_test > /sys/class/remoteproc/\
4   remoteproc0/firmware
5 echo start > /sys/class/remoteproc/remoteproc0/state
6 echo_test
7 echo stop > /sys/class/remoteproc/remoteproc0/state
```

In this setup you need to be careful for the name of the .elf binary to be exactly used in the first mv and echo command. In this example, the binary would be named image_echo_test.elf, and moved from \$HOME to /lib/firmware.

The debug of the firmware itself is done by reading the "printf" visible from the serial return of the board (typically a /dev/ttyUSB1), but two things are to be noted:

- If the echo start command fails, either the previous firmware run was not stopped, or the new binary itself is impossible to run.
- In general, if the echo_test runs, it means that everything is okay and that the RPMsg system worked successfully.

²⁸In this sequence, we are entering a root shell with sudo -s, but this can also be archived by putting the commands in a script to be executed with sudo.

²⁹It is also important to note that the echo_test part is specific for the RPMsg base demonstration firmware. It is not to be used for other firmware. The instruction to build and use this particular program on the Kria Linux is visible in the section 6.

11 Loading the firmware using JTAG and Vitis IDE

This is more of an alternative way, but it is possible to load the Client firmware onto the Cortex R5F onto the board from a host PC IDE, without booting or using the Linux on the KRIA board itself.

This section will present the way to do so, and most critically, the steps that must be taken beforehand on the host machine for that system to work.

There are two key advantages to using this method:

- The load time for the firmware is real fast. In particular, re-loading the firmware doesn't require to reboot the entire Linux system.
- IDE debugging available. Running the JTAG on the board allows for the firmware to be debugged in a easier way.

11.1 Testing xsct and running the drivers

Firstly, this part of the guide assumes that a version of the Vitis Classic IDE was installed on the host system. Some of the needed tools to test the working of the JTAG system are located in the installation directory of the IDE.

In my case, I'll have to go to `~/tools/Xilinx/Vitis/2023.2`, but your mileage might vary depending on the settings you choose during the IDE installation and also the version of the IDE itself.

11.1.1 Drivers

Firstly, we need to load some drivers in order for the JTAG USB system to work:

```
1 sudo ./data/xicom/cable_drivers/lin64/install_script/install_drivers/install_drivers
```

11.1.2 dialout access

It is also much needed for the user that runs the IDE to have read/write access to the USB devices. This can be done as follows, as root:

```
1 usermod -aG dialout ${USER}
```

You'll need to log out and log back in for the changes to apply though.

11.1.3 TCL JTAG boot

A short TCL program was provided by Honda-sensei and should be loaded in order for the board to boot in JTAG mode. The following code block shows the TCL, and it should be saved somewhere, for example in `~/Downloads/boot_jtag.tcl`:

```
1 proc boot_jtag { } {
2 #####
3 # Switch to JTAG boot mode #
4 #####
5 targets -set -filter {name =~ "PSU"}
6 # update multiboot to ZERO
7 mwr 0xffca0010 0x0
8 # change boot mode to JTAG
9 mwr 0xff5e0200 0x0100
10 # reset
11 rst -system
12 }
```

```

13
14
15 # Jtag connection
16 connect
17 puts stdout "Connect Jtag"
18
19 # Switching to JtagBoot
20 puts -newline stdout "Start transition to jtag boot mode..."
21 boot_jtag
22 puts stdout "done."
23
24 # Jtag disconnection
25 puts -newline stdout "Disconnect Jtag..."
26 disconnect
27 puts stdout "done."

```

Eventually it is possible to load the JTAG boot script as follow:

```

1 cd ~/tools/Xilinx/Vitis/2023.2
2 source settings64.sh
3 xsct -norlwrap ~/Downloads/boot_jtag.tcl

```

11.2 Vitis IDE classic

Note that the steps presented here must be taken after running the `xsct` command; without rebooting the board in the mean time.

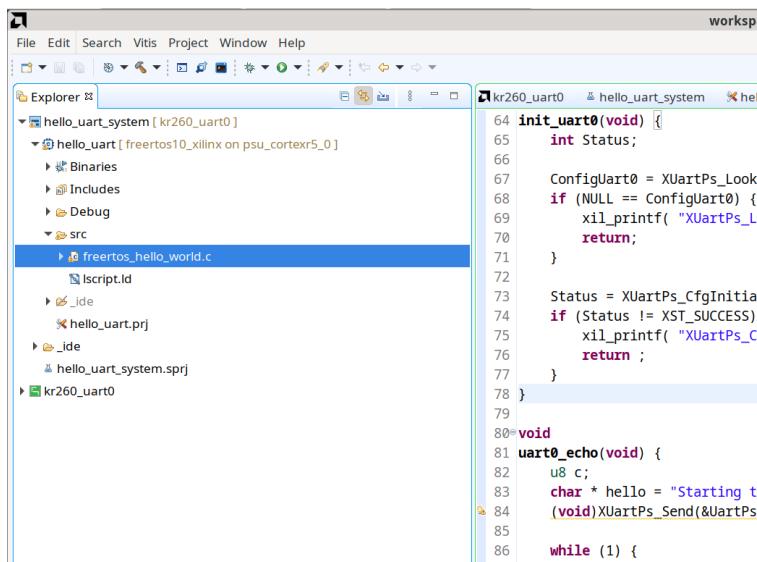


Figure 22: With a project meant to be build and run on the Cortex R5F, you must first produce the .elf binary file.

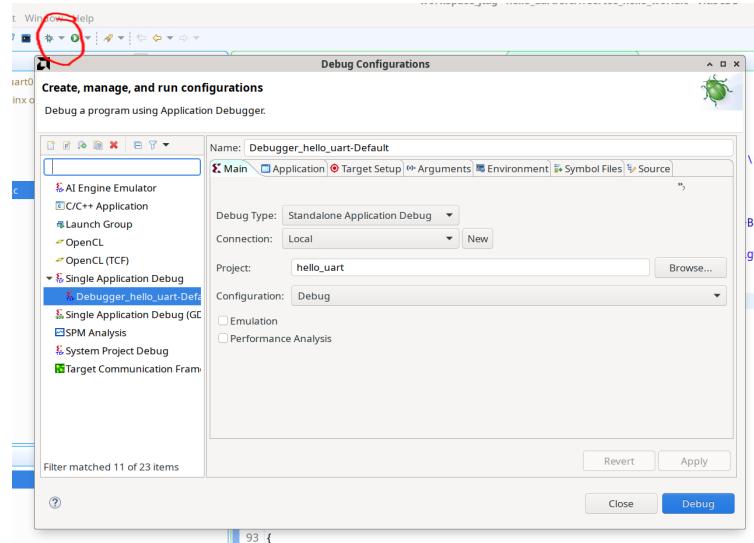


Figure 23: Then you can open the Debug Configuration window.

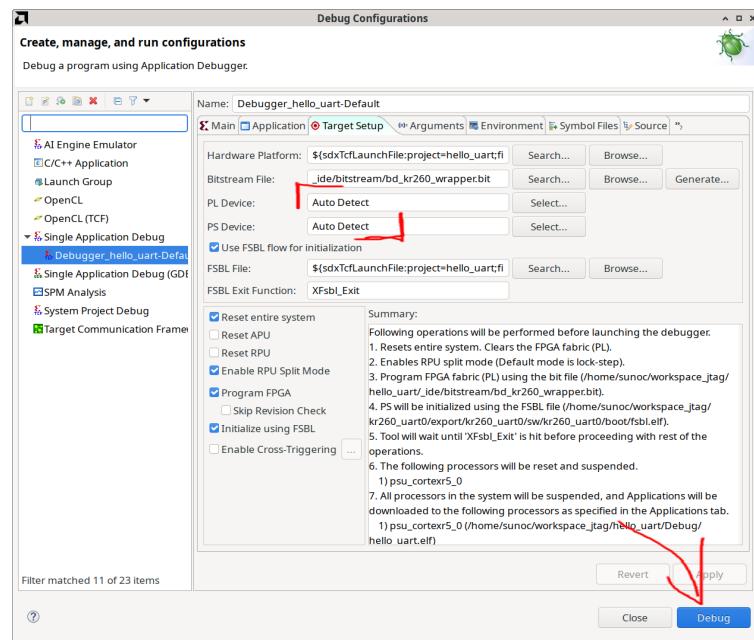


Figure 24: All parameters should stay the same. You can then run "Debug".

```

XSCT Console Emulation Console
XSCT Process
  section, .vectors: 0x00000000 - 0x00000000
  section, .bootdata: 0x0000005a0 - 0x00000071f
  section, .text: 0x00100000 - 0x00110d4f
  section, .init: 0x00110d50 - 0x00110d5b
  section, .fini: 0x00110d5c - 0x00110d67
  section, .note.gnu.build-id: 0x00110d68 - 0x00110d8b
  section, .rodata: 0x00110d90 - 0x00111870
  section, .data: 0x00111878 - 0x00112953
  section, .eh_frame: 0x00112954 - 0x00112957
  section, .ARM.exidx: 0x00112958 - 0x0011295f
  section, .init_array: 0x00112960 - 0x00112963
  section, .fini_array: 0x00112964 - 0x00112967
  section, .bss: 0x00112968 - 0x00122c4f
  section, .heap: 0x00122c50 - 0x00124c4f
  section, .stack: 0x00124c50 - 0x0012844f

  0%   0MB   0.0MB/s  ???:? ETA
100%   0MB   0.2MB/s  00:00
Setting PC to Program Start Address 0x0000003c
Successfully downloaded /home/sunoc/workspace_jtag/hello_uart/Debug/hello_uart.elf
Info: Cortex-R5 #0 (target 8) Running
Info: Cortex-R5 #0 (target 8) Stopped at 0x110d34 (Breakpoint)
main() at ../src/freertos_hello_world.c: 93
93: {
xsct%

```

Figure 25: The XSCT console will confirm the successful load of the firmware. If not, you will get error messages and should reboot the board.

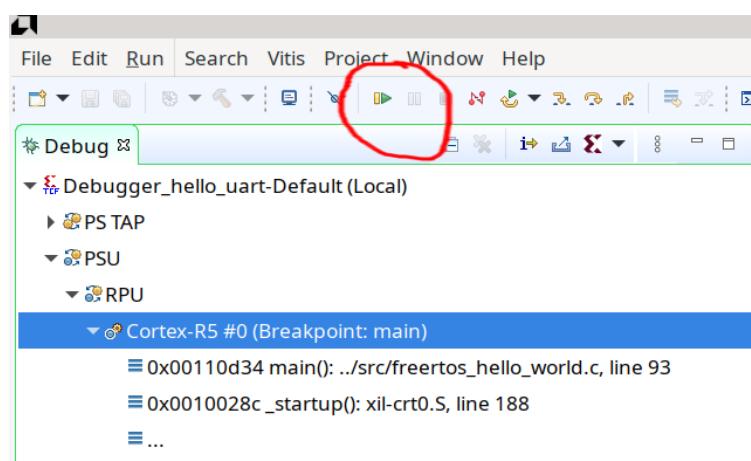


Figure 26: Selecting the Cortex R5, core #0, you can then run the application.

12 Running a ROS2 node

This section as well as the next one are rather "separated" from the rest of the report, as they are focused on the ROS2 system being used on the Kria board.

In this first section, the installation of ROS2 as a system will be presented, with two different ways of approaching the problem.

As for the previous section, the figure 27 below shows what part of the overall system we are talking about here.

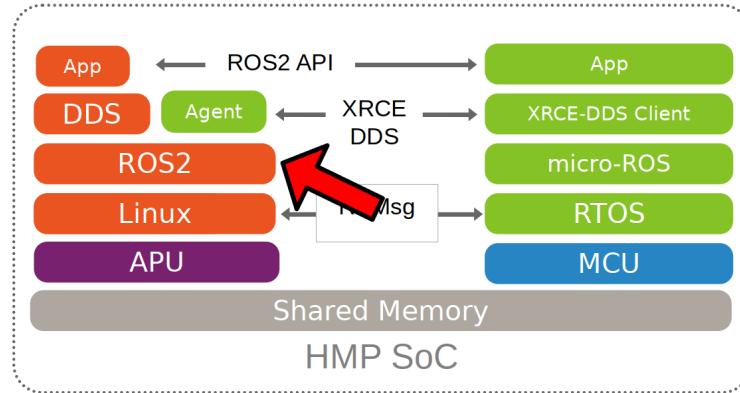


Figure 27: The ROS2 middle (red border) runs on top of the Linux, on the general-purpose core of the Kria board.

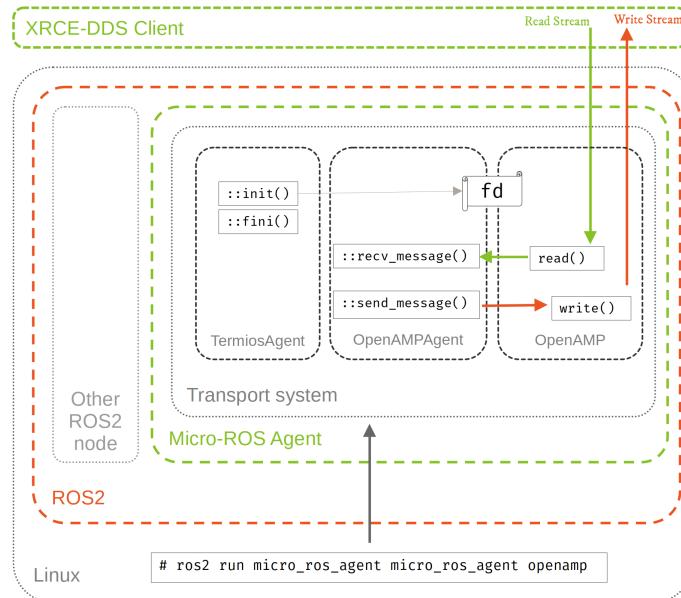


Figure 28: Methods architecture for the modified Agent node.

12.1 On the host Linux ("bare-metal")

Since an Ubuntu distribution is installed on the board, the installation of ROS2 can be done³⁰ in a standard way, using the repository.

An official documentation is provided with ROS2 themselves with a step-by-step guide on how to install ROS2 on a Ubuntu system{}). We will be following this guide here³¹.

Firstly, we need to update the locals, enable the universe Ubuntu repository, get the key and add the repository for ROS2. This can be done as follow:

```
1 locale # check for UTF-8
2 sudo apt update && sudo apt install -y locales
3 sudo locale-gen en_US en_US.UTF-8
4 sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
5 export LANG=en_US.UTF-8
6 locale # verify settings
7
8 sudo apt install -y software-properties-common
9 sudo add-apt-repository universe
10 sudo apt update && sudo apt install -y curl wget
11
12 wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
13 sudo mv ros.key /usr/share/keyrings/ros-archive-keyring.gpg
```

Then, a thicc one-liner is available to add the ROS2 repository to our system:

```
1 echo "deb [arch=$(dpkg --print-architecture) \
2 signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
3 http://packages.ros.org/ros2/ubuntu $(. \
4 /etc/os-release && echo $UBUNTU_CODENAME) main" | \
5 sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

It is then possible to install ROS2³² as follow:

```
1 sudo apt update
2 sudo apt upgrade -y
3 sudo apt install -y ros-$ROS_DISTRO-desktop \
4     ros-$ROS_DISTRO-ros-base \
5     python3-argcomplete \
6     ros-dev-tools
```

Once installed, it is possible to test the system with a provided example. You need to open two terminals and log wish SSH onto the board, then running respectively:

```
1 source /opt/ros/$ROS_DISTRO/setup.bash
2 ros2 run demo_nodes_cpp talker
```

And then:

³⁰As always, this configuration was tested solely on Ubuntu LTS 22.04, with the ROS2 versions Humble and then Iron being deployed. Other combination of versions should work as well, but they are not tested for this guide. In case of doubt or problem, please refer to the official documentation.

³¹The curl command from the guide does not work through the school proxy, but the command wget used instead does work. The key is then moved to the correct spot with mv.

³²This command installs a complete "desktop" version of ROS2, containing many useful package for our project. If space is a constraint, different, less complete packages can be install. Please refer to the official documentation about it.

```

1 source /opt/ros/$ROS_DISTRO/setup.bash
2 ros2 run demo_nodes_py listener

```

You should be able to see the first terminal sending "Hello world" messages, and the second one receiving them.

12.2 In a container (Docker)

As containers are used to test and build micro-ROS configurations, running ROS2 in a Docker is a great way to have a reproducible configuration of your system.

This part of the guide will present how to install Docker on the Kria board and then how to use it to deploy the latest version of ROS2. This section of the report assumes that Docker was installed on the target system as presented in section 3.2.10.

The following commands will pull a ROS container, version iron, and name it `ros_build`.

A key part for having access to the interfaces (serial) is the mapping of the whole `/dev` range of devices from the host machine to the internal `/dev` of the container³³. With the second command, we can execute bash as a way to open a terminal to the "inside" the container:

```

1 docker run -d --name ros_agent -it --net=host -v \
2   /dev:/dev --privileged ros:iron
3 docker exec -it ros_agent bash

```

From there, it becomes possible to simply use ROS2 as you would for a bare-metal install, and as presented in the section 12.1 above:

```

1 source /opt/ros/$ROS_DISTRO/setup.bash
2
3 # Create a workspace and download the micro-ROS tools
4 mkdir microros_ws
5 cd microros_ws
6 git clone -b $ROS_DISTRO https://github.com/micro-ROS/\
7   micro_ros_setup.git src/micro_ros_setup
8
9 # Update dependencies using rosdep
10 sudo apt update && rosdep update
11 rosdep install --from-paths src --ignore-src -y
12
13 # Install pip
14 sudo apt-get install python3-pip
15
16 # Build micro-ROS tools and source them
17 colcon build
18
19 # Download micro-ROS-Agent packages
20 source install/local_setup.bash
21 ros2 run micro_ros_setup create_agent_ws.sh
22
23 # Build step
24 ros2 run micro_ros_setup build_agent.sh
25
26 # Run a micro-ROS agent

```

³³This is an example and this situation can become a security issue. It would be a better practice in a production environment to map only the devices that are actually in use.

```
27 ros2 run micro_ros_agent micro_ros_agent serial \
28   --dev /dev/ttyUSB1
```

Then once again in a similar way to the bare-metal deployment, it is possible to run a demonstration the ping-pong topic communication from a different shell³⁴:

```
1 source /opt/ros/$ROS_DISTRO/setup.bash
2
3 # Subscribe to micro-ROS ping topic
4 ros2 topic echo /microROS/ping
```

³⁴You need to be careful to have you shell in the "correct" space: these command need to be run inside the container in which the previous setup were install, not on the host running the container system. The hostname should help you to figure out where you are.

13 micro-ROS XRCE-DDS Agent

The micro-ROS Agent on the ROS2 side is the last piece of the puzzle needed to allow our DDS environment to use RPMsg as a mean of communication, as visible on the schematic of the figure 29 below. In particular, it will be useful to modify this agent in order to archive the full RPMsg communication for ROS2³⁵. An official documentation exists, but it gives little to no detail on how to deploy such modified, custom transport setup. This part of the guide will focus on it.

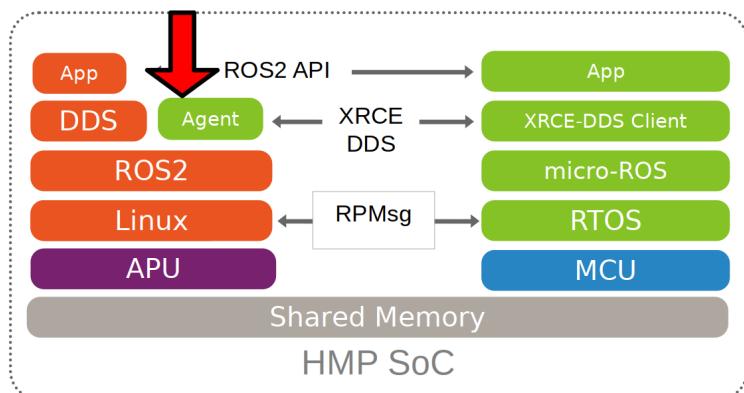


Figure 29: The agent (red border) allows for a micro-ROS instance to communicate with a ROS2 system. It is deployed on the Linux side, as a ROS2 node.

13.1 Building a XRCE-DDS agent in a Docker

The same command presented above for running a custom agent "bare-metal" can be run inside a Docker.

```
1 docker run -d --name XRCE_DDS_Agent -it --net=host -v \ /dev:/dev --privileged ros:iron
2
3 docker exec -it XRCE_DDS_Agent bash
```

```
1 git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
2 cd Micro-XRCE-DDS-Agent
3 docker build -t xrce-dds-agent .
4 docker run -it --privileged -v /dev:/dev xrce-dds-agent serial \
5 --dev /dev/ttyACM0
```

13.2 Building the Agent version with a modified transport

The first working implementation of the DDS over RPMsg happened over a XRCE-DDS agent whose serial transport layer has been modified in order to use OpenAMP's RPMsg instead.

The build process is the same as usual, just pulling from my fork of eProsima repository and using the develop branch:

```
1 git clone https://github.com/sunoc/Micro-XRCE-DDS-Agent.git -b develop
2 cd Micro-XRCE-DDS-Agent
3 mkdir build
4 cd build
```

³⁵This will be done with ROS2 agents "custom transport" system, which has little documentation. Some discussions about it exist though.

```
5 cmake ..
6 make
```

Giving that your are working from an environment in which the ROS2 tools are available, it is then possible to run the Agent as follow³⁶ , ³⁷:

```
1 source /opt/ros/$ROS_DISTRO/setup.bash
2
3 ./MicroXRCEAgent serial --dev /dev/null --verbose 6
```

13.3 Creating a custom transport agent

WORK_IN_PROGRESS

These command allows to go build the agent and run it with maximum verbosity (for debugging purpose).

```
1 source /opt/ros/$ROS_DISTRO/setup.bash
2 cd /Micro-XRCE-DDS-Agent/build
3 make clean && make -j4
4 ./MicroXRCEAgent serial --dev /dev/null --verbose 6
```

³⁶The --dev /dev/null option is only a placeholder as the serial requires it but not RPMsg.

³⁷The --verbose 6 allows to see the maximum amount of information about the data being transmitted.

14 Running the ping-pong node

WORK_IN_PROGRESS

A custom ping-pong node for ROS2 was developed in order to test the data transfer of the newly created Agent / Client system.

In order to have this node up and running, you need to set basically all the previously presented points:

- The micro-ROS firmware is running on the R5F core, as presented in the section 10.
- On the same Linux you plan to run the node, you will need to run and keep the Agent, as presented in the section 13.
- This ping-pong application is released as a ROS2 Python node. It is best to pull and run it in a ROS2 environment to avoid having dependencies issues, as presented in the section 12.

15 Monitoring and performances evaluation

WORK_IN_PROGRESS

16 Conclusion & future

WORK_IN_PROGRESS

A DTO patch

This file is available in this repository: system.patch

```
1 --- system.dts.orig.2      2024-04-25 05:56:38.473218639 +0000
2 +++ system.dts      2024-04-25 06:03:14.724714961 +0000
3 @@ -196,7 +196,7 @@
4
5     firmware {
6
7 -         zynqmp-firmware {
8 +         zynqmp_firmware: zynqmp-firmware {
9             compatible = "xlnx,zynqmp-firmware";
10            #power-domain-cells = <0x01>;
11            method = "smc";
12 @@ -708,7 +708,7 @@
13             phandle = <0x41>;
14         };
15
16 -         interrupt-controller@f9010000 {
17 +         gic: interrupt-controller@f9010000 {
18             compatible = "arm,gic-400";
19             #interrupt-cells = <0x03>;
20             reg = <0x00 0xf9010000 0x00 0x10000 0x00 0xf9020000 0x00
21             ↳ 0x20000 0x00 0xf9040000 0x00 0x20000 0x00 0xf9060000 0x00
22             ↳ 0x20000>;
23
24 -         pinctrl-names = "default";
25 -         u-boot,dm-pre-reloc;
26 -         compatible = "xlnx,zynqmp-uart\0cdns,uart-rip12";
27 -         status = "okay";
28 +         status = "disabled";
29             interrupt-parent = <0x05>;
30             interrupts = <0x00 0x16 0x04>;
31             reg = <0x00 0xff010000 0x00 0x1000>;
32
33 @@ -1912,18 +1912,6 @@
34             reg = <0x00 0x00 0x00 0x80000000 0x08 0x00 0x00 0x80000000>;
35         };
36
37 -         reserved-memory {
38 -             #address-cells = <0x02>;
39 -             #size-cells = <0x02>;
40 -             ranges;
41
42 -             pmu@7ff00000 {
43 -                 reg = <0x00 0x7ff00000 0x00 0x100000>;
44 -                 no-map;
45 -                 phandle = <0x7a>;
46 -             };
47 -         };
48
49 -         gpio-keys {
50             compatible = "gpio-keys";
51             autorepeat;
52
53 @@ -1964,6 +1952,83 @@
54
```

```

50          pwm_s = <0x1f 0x02 0x9c40 0x00>;
51      };
52
53 +     reserved-memory {
54 +         #address-cells = <2>;
55 +         #size-cells = <2>;
56 +         ranges;
57 +         rpu0vdev0ring0: rpu0vdev0ring0@3ed80000 {
58 +             no-map;
59 +             reg = <0x0 0x3ed80000 0x0 0x4000>;
60 +         };
61 +         rpu0vdev0ring1: rpu0vdev0ring1@3ed84000 {
62 +             no-map;
63 +             reg = <0x0 0x3ed84000 0x0 0x4000>;
64 +         };
65 +         rpu0vdev0buffer: rpu0vdev0buffer@3ed88000 {
66 +             no-map;
67 +             reg = <0x0 0x3ed88000 0x0 0x100000>;
68 +         };
69 +         rproc_0_reserved: rproc_0_reserved@3ed00000 {
70 +             no-map;
71 +             reg = <0x0 0x3ed00000 0x0 0x80000>;
72 +         };
73 +     };
74 +     tcm_0a: tcm_0a@ffe00000 {
75 +         no-map;
76 +         reg = <0x0 0xffe00000 0x0 0x10000>;
77 +         status = "okay";
78 +         compatible = "mmio-sram";
79 +         power-domain = <&zynqmp_firmware 15>;
80 +     };
81 +     tcm_0b: tcm_0b@ffe20000 {
82 +         no-map;
83 +         reg = <0x0 0xffe20000 0x0 0x10000>;
84 +         status = "okay";
85 +         compatible = "mmio-sram";
86 +         power-domain = <&zynqmp_firmware 16>;
87 +     };
88 +     rf5ss@ff9a0000 {
89 +         compatible = "xlnx,zynqmp-r5-remotePROC";
90 +         xlnx,cluster-mode = <1>;
91 +         ranges;
92 +         reg = <0x0 0xFF9A0000 0x0 0x15000>;
93 +         #address-cells = <0x2>;
94 +         #size-cells = <0x2>;
95 +         r5f_0 {
96 +             compatible = "xilinx,r5f";
97 +             #address-cells = <2>;
98 +             #size-cells = <2>;
99 +             ranges;
100 +             sram = <&tcm_0a &tcm_0b>;
101 +             memory-region = <&rproc_0_reserved>, <&rpu0vdev0buffer>,
102 +             ↳ <&rpu0vdev0ring0>, <&rpu0vdev0ring1>;
102 +             power-domain = <&zynqmp_firmware 7>;

```

```

103 +         mboxes = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;
104 +         mbox-names = "tx", "rx";
105 +     };
106 +
107 +     zynqmp_ipi1 {
108 +         compatible = "xlnx,zynqmp-ipi-mailbox";
109 +         interrupt-parent = <&gic>;
110 +         interrupts = <0 29 4>;
111 +         xlnx,ipi-id = <7>;
112 +         #address-cells = <1>;
113 +         #size-cells = <1>;
114 +         ranges;
115 +         /* APU<->RPU0 IPI mailbox controller */
116 +         ipi_mailbox_rpu0: mailbox@ff990000 {
117 +             reg = <0xff990600 0x20>,
118 +                   <0xff990620 0x20>,
119 +                   <0xff9900c0 0x20>,
120 +                   <0xff9900e0 0x20>;
121 +             reg-names = "local_request_region",
122 +                         "local_response_region",
123 +                         "remote_request_region",
124 +                         "remote_response_region";
125 +             #mbox-cells = <1>;
126 +             xlnx,ipi-id = <1>;
127 +         };
128 +
129 +
130     __symbols__ {
131         pinctrl_usb1_default =
132             ↳ "/firmware/zynqmp-firmware/pinctrl/usb1-default";
133         pinctrl_usb0_default =
134             ↳ "/firmware/zynqmp-firmware/pinctrl/usb0-default";

```

B Custom tool-chain CMake settings

This file is available in this repository: custom_r5f_toolchain.cmake

```
1 set(CMAKE_SYSTEM_NAME Generic)
2 set(CMAKE_CROSSCOMPILING 1)
3 set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
4
5
6 set(ARCH_CPU_FLAGS "-O2 -ffunction-sections -fdata-sections -fno-exceptions
7   ↳ -mcpu=cortex-r5 -mfpu=vfpv3-d16 -mfloat-abi=hard -Wl,--gc-sections -nostdlib --param
8     ↳ max-inline-insns-single=500 -DF_CPU=84000000L
9     ↳ -D'RCUTILS_LOG_MIN_SEVERITY=RCUTILS_LOG_MIN_SEVERITY_NONE')
10 set(ARCH_OPT_FLAGS "")
11
12 set(CMAKE_C_COMPILER arm-none-eabi-gcc)
13 set(CMAKE_CXX_COMPILER arm-none-eabi-g++)
14
15
16
17 set(__BIG_ENDIAN__ 0)
```

C Custom Colcon meta settings

This file is available in this repository: custom r5f colcon.meta

```
1  {
2      "names": {
3          "tracetools": {
4              "cmake-args": [
5                  "-DTRACETOOLS_DISABLED=ON",
6                  "-DTRACETOOLS_STATUS_CHECKING_TOOL=OFF"
7              ]
8          },
9          "rosidl_typesupport": {
10             "cmake-args": [
11                 "-DROSIDL_TYPESUPPORT_SINGLE_TYPESUPPORT=ON"
12             ]
13         },
14         "rcl": {
15             "cmake-args": [
16                 "-DBUILD_TESTING=OFF",
17                 "-DRCL_COMMAND_LINE_ENABLED=OFF",
18                 "-DRCL_LOGGING_ENABLED=OFF"
19             ]
20         },
21         "rcutils": {
22             "cmake-args": [
23                 "-DENABLE_TESTING=OFF",
24                 "-DRCUTILS_NO_FILESYSTEM=ON",
25                 "-DRCUTILS_NO_THREAD_SUPPORT=ON",
26                 "-DRCUTILS_NO_64_ATOMIC=ON",
27                 "-DRCUTILS_AVOID_DYNAMIC_ALLOCATION=ON"
28             ]
29         },
30         "microxrcedds_client": {
31             "cmake-args": [
32                 "-DUCLIENT_PROFILE_UDP=OFF",
33                 "-DUCLIENT_PROFILE_TCP=OFF",
34                 "-DUCLIENT_PROFILE_DISCOVERY=OFF",
35                 "-DUCLIENT_PROFILE_SERIAL=OFF",
36                 "-UCLIENT_PROFILE_STREAM_FRAMING=ON",
37                 "-DUCLIENT_PROFILE_CUSTOM_TRANSPORT=ON"
38             ]
39         },
40         "rmw_microxrcedds": {
41             "cmake-args": [
42                 "-DRMW_UXRCE_MAX_NODES=1",
43                 "-DRMW_UXRCE_MAX_PUBLISHERS=5",
44                 "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=5",
45                 "-DRMW_UXRCE_MAX_SERVICES=1",
46                 "-DRMW_UXRCE_MAX_CLIENTS=1",
47                 "-DRMW_UXRCE_MAX_HISTORY=4",
48                 "-DRMW_UXRCE_TRANSPORT=custom"
49             ]
50         }
51     }
```

52 }

53 }

D Firmware time functions

D.1 main

This file is available in this repository: `clock.c` but a potentially more up-to-date version is visible directly at the `libmicroros_kv260` repository: `clock.c`

```
1 #include "microros.h"
2
3
4 int _gettimeofday( struct timeval *tv, void *tzvp )
5 {
6     XTime t = 0;
7     XTime_GetTime(&t); //get uptime in nanoseconds
8     tv->tv_sec = t / 1000000000; // convert to seconds
9     tv->tv_usec = ( t % 1000000000 ) / 1000; // get remaining microseconds
10    return 0; // return non-zero for error
11 } // end _gettimeofday()
12
13
14 void UTILS_NanosecondsToTimespec( int64_t llSource,
15                                     struct timespec * const pxDestination )
16 {
17     long lCarrySec = 0;
18
19     /* Convert to timespec. */
20     pxDestination->tv_sec = ( time_t ) ( llSource / NANOSECONDS_PER_SECOND );
21     pxDestination->tv_nsec = ( long ) ( llSource % NANOSECONDS_PER_SECOND );
22
23     /* Subtract from tv_sec if tv_nsec < 0. */
24     if( pxDestination->tv_nsec < 0L )
25     {
26         /* Compute the number of seconds to carry. */
27         lCarrySec = ( pxDestination->tv_nsec / ( long ) NANOSECONDS_PER_SECOND ) + 1L;
28
29         pxDestination->tv_sec -= ( time_t ) ( lCarrySec );
30         pxDestination->tv_nsec += lCarrySec * ( long ) NANOSECONDS_PER_SECOND;
31     }
32 }
33
34 int clock_gettime( clockid_t clock_id,
35                     struct timespec * tp )
36 {
37     TimeOut_t xCurrentTime = { 0 };
38
39     /* Intermediate variable used to convert TimeOut_t to struct timespec.
40      * Also used to detect overflow issues. It must be unsigned because the
41      * behavior of signed integer overflow is undefined. */
42     uint64_t ullTickCount = OULL;
43
44     /* Silence warnings about unused parameters. */
45     ( void ) clock_id;
46
47     /* Get the current tick count and overflow count. vTaskSetTimeOutState()
48      * is used to get these values because they are both static in tasks.c. */
```

```

49     vTaskSetTimeOutState( &xCurrentTime );
50
51     /* Adjust the tick count for the number of times a TickType_t has overflowed.
52      * portMAX_DELAY should be the maximum value of a TickType_t. */
53     ullTickCount = ( uint64_t ) ( xCurrentTime.xOverflowCount ) << ( sizeof( TickType_t
54     → ) * 8 );
55
56     /* Add the current tick count. */
57     ullTickCount += xCurrentTime.xTimeOnEntering;
58
59     /* Convert ullTickCount to timespec. */
60     UTILS_NanosecondsToTimespec( ( int64_t ) ullTickCount * NANOSECONDS_PER_TICK, tp );
61
62     return 0;
63 }
```

D.2 header file

```

1  /**< Microseconds per second. */
2  #define MICROSECONDS_PER_SECOND    ( 1000000LL )
3  /**< Nanoseconds per second. */
4  #define NANOSECONDS_PER_SECOND     ( 1000000000LL )
5  /**< Nanoseconds per FreeRTOS tick. */
6  #define NANOSECONDS_PER_TICK       ( NANOSECONDS_PER_SECOND / configTICK_RATE_HZ )
```

E Firmware memory allocation functions

E.1 main

This file is available in this repository: allocators.c but a potentially more up-to-date version is visible directly at the libmicroros_kv260 repository: allocators.c

```
1 #include "allocators.h"
2
3 //int absoluteUsedMemory = 0;
4 //int usedMemory = 0;
5
6 void * __freertos_allocate(size_t size, void * state){
7     (void) state;
8     LPRINTF("## Alloc %d (prev: %d B)\r\n",size, xPortGetFreeHeapSize());
9     // absoluteUsedMemory += size;
10    // usedMemory += size;
11
12    LPRINTF("Return for the allocate function w parameter size = %d\r\n", size);
13
14    return pvPortMalloc(size);
15}
16
17 void __freertos_deallocate(void * pointer, void * state){
18     (void) state;
19     LPRINTF("## Free 0x%x (prev: %d B)\r\n", pointer, xPortGetFreeHeapSize());
20     if (NULL != pointer)
21     {
22         LPRINTF("Pointer is not null.\r\n");
23         usedMemory -= getBlockSize(pointer);
24         LPRINTF("usedMemory var updated: %d\r\n", usedMemory);
25         vPortFree(pointer);
26     }
27     else
28     {
29         LPERROR("Trying to deallocate a null pointed. Doing nothing.\r\n");
30     }
31 }
32
33 void * __freertos_reallocate(void * pointer, size_t size, void * state){
34     (void) state;
35     LPRINTF("## Realloc 0x%x -> %d (prev: %d B)\r\n", pointer, size,
36     ↵ xPortGetFreeHeapSize());
37     // absoluteUsedMemory += size;
38     // usedMemory += size;
39     if (NULL == pointer)
40     {
41         return __freertos_allocate(size, state);
42     }
43     else
44     {
45         usedMemory -= getBlockSize(pointer);
46         // return pvPortRealloc(pointer,size);
47 }
```

```

48     __freertos_deallocate(pointer, state);
49     return __freertos_allocate(size, state);
50 }
51 }
52 }
53
54 void * __freertos_zero_allocate(size_t number_of_elements, size_t size_of_element, void
55     * state){
56     (void) state;
57     LPRINTF("<!-- Calloc %d x %d = %d -&gt; (prev: %d
58         B)\r\n", number_of_elements, size_of_element, number_of_elements * size_of_element,
59         xPortGetFreeHeapSize());
60 //    absoluteUsedMemory += number_of_elements * size_of_element;
61 //    usedMemory += number_of_elements * size_of_element;
62
63     return pvPortCalloc(number_of_elements, size_of_element);
64 }</pre>

```

E.2 header file

```

1 #ifndef _ALLOCATORS_H_
2 #define _ALLOCATORS_H_
3
4 #include "microros.h"
5
6 extern int absoluteUsedMemory;
7 extern int usedMemory;
8
9
10 void * __freertos_allocate(size_t size, void * state);
11 void __freertos_deallocate(void * pointer, void * state);
12 void * __freertos_reallocate(void * pointer, size_t size, void * state);
13 void * __freertos_zero_allocate(size_t number_of_elements,
14                               size_t size_of_element, void * state);
15
16 #endif // _ALLOCATORS_H_
```

F [deprecated] Installing Linux (PetaLinux option)

This part is an alternative to the previous section 3, where the installation of PetaLinux as a host system³⁸ will be presented, as being an alternative to the Ubuntu LTS. Here are some reasoning why you'd prefer to use PetaLinux instead of Ubuntu:

- Direct support from Xilinx. While Ubuntu is also an official port, it is provided through Canonical.
- If you plan to use ROS2 in containers.

It is also to be noted that the building process of a PetaLinux image is not as trivial as simply flashing a downloaded ISO file. It takes some time to configure it and even more time to be built, especially if you are working on a lower-end machine.

³⁸It is to be noted that for this part, the "other" KRIA board (KR260) was used instead of the KV260 that was being used for everything tried on top of Ubuntu. No significant difference should be noted.

F.1 General installation

The indication on how to install PetaLinux on a KRIA board is detailed in the official documentation³⁹, but for the KV260 board with an older release of the Linux. This guide will try to keep an up-to-date version of the guide, in particular for the KR260 KRIA board and, at the time of writing, the PetaLinux in version 2023.2 for the installer. The 2022.2 version is recommended for this board's latest updated boot firmware, as presented in the Section 2 above; however I happened to have issues with the building process from a system based on Ubuntu 22.04. Thus the choice of the 2023.2 that should be compatible.

Two tools should be downloaded upfront in order to be ready and create the PetaLinux image for our target board. Both can be obtained at the Xilinx official download page⁴⁰. Login with an AMD account is required in order to download the files we need, namely, we'll have to get:

- The installer tool: `petalinux-v2023.2-10121855-installer.run` or similar name.
- The SOM board support package: `xilinx-kr260-starterkit-v2023.2-10140544.bsp` or similar name⁴¹.

The specific name of the utils you'll get to download will evolve over time.

For the following command, I will assume that both files were downloaded into the same directory and that you have a shell open in said directory.

Firstly, a crap ton of dependencies are needed. They are detailed in the release notes of the version of the tool you are downloading and it depends on your distribution, but here is a one-liner that works for the PetaLinux 2023.2 to be built on Ubuntu(-based) 22.04 LTS:

```
1 sudo apt-get install iproute2 gawk python3 build-essential gcc git make \
2     net-tools libncurses5-dev tftpd zlib1g-dev libssl-dev flex bison libselinux1 \
3     gnupg wget git-core diffstat chrpath socat xterm autoconf libtool tar unzip \
4     texinfo zlib1g-dev gcc-multilib automake zlib1g:i386 screen pax gzip cpio \
5     python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git \
6     python3-jinja2 libegl1-mesa libsdl1.2-dev pylint asciidoc
```

Testing, if building on Debian 12 the following variante of the same command has been confirmed to work:

```
1 sudo apt-get install iproute2 gawk python3 build-essential gcc git make \
2     net-tools libncurses5-dev tftpd-hpa zlib1g-dev libssl-dev flex bison libselinux1 \
3     gnupg wget git-core diffstat chrpath socat xterm autoconf libtool tar unzip \
4     texinfo zlib1g-dev gcc-multilib automake zlib1g screen pax gzip cpio python3-pip \
5     python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 \
6     libegl1-mesa libsdl1.2-dev pylint asciidoc
```

Installing the petalinux tools can be done with the `.run` script⁴³, as follow:

```
1 sudo chmod +x ./petalinux-v2023.2-10121855-installer.run
2 ./petalinux-v2023.2-10121855-installer.run
3 bash
4 source settings.sh
```

A project can then be created, using the `.bsp` file. From there it is possible to build the image file meant to be booted by the board:

³⁹https://xilinx.github.io/kria-apps-docs/kv260/2021.1/build/html/docs/build_petalinux.html

⁴⁰<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2022-2.html>

⁴¹As of the time of writing this section, it was recommended⁴²to be using the boot firmware in version 2022.2. Thus the software pack for the board will also be using this version.

⁴²<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1641152513/Kria+SOMs+Starter+Kits#K26-Boot-Firmware-Updates>

⁴³It is to be noted that the `source` command will run best on bash. Trying to run it on zsh for ex. will cause issues. This can be enforced with the command `sudo dpkg-reconfigure bash`.

```

1 petalinux-create -t project -s xilinx-kr260-starterkit-v2023.2-10140544.bsp
2 cd xilinx-kr260-starterkit-2023.2/

```

The more general option allows to enable stuff related to hardware configuration⁴⁴. For this step, it is possible to select the openamp setting the the DTG Settings, as visible in the Figure 30 below:

```

1 petalinux-config --get-hw-description ./hardware/xilinx-kr260-starterkit-2023.2/kr260_starter_kit.xsa

```

Then, here are the options that we want to enable:

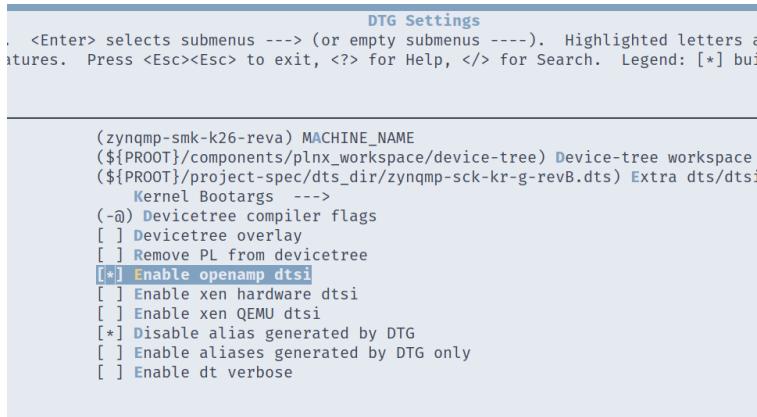


Figure 30: In the hardware config, it is possible to select the openamp device tree source.

Some kernel and module configuration are needed in order for the petalinux system to work properly. These setup can be set using the following command:

```

1 petalinux-config -c rootfs

```

Then, here are the options that we want to enable:

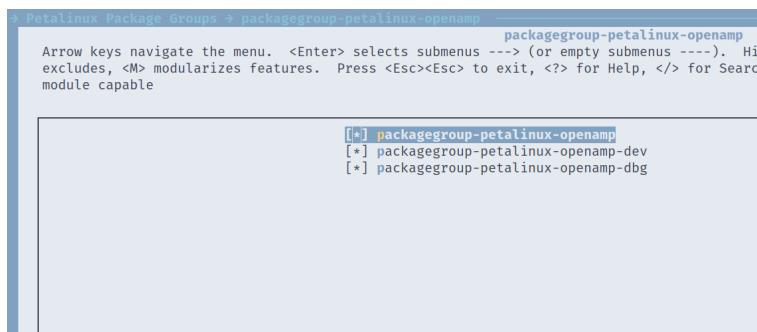


Figure 31: In the rootfs config, it is possible to select the openamp package.

It is then possible to build⁴⁵ and package the image. Most critically compared to the example, the openamp.dtb overlay must be used in order for OpenAMP and subsequently RPMMsg to become usable.

⁴⁴If you search in the work directory, you might notice that three differently named .xsa files exist. After a quick diff check, they appear to be exactly identical.

⁴⁵Beware, the petalinux-build command execution takes a lot of time.

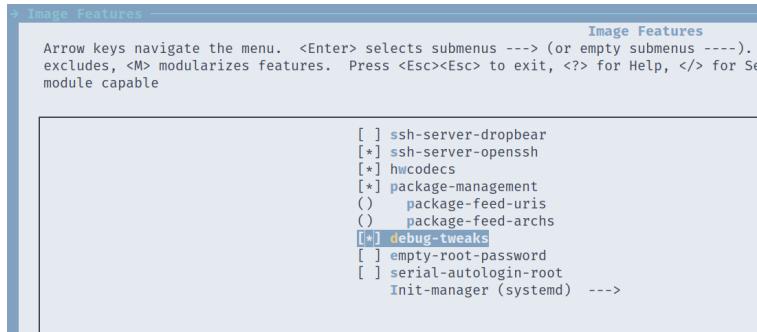


Figure 32: OpenSSH can be configured in the same rootfs config.

```

1 petalinux-build
2 petalinux-package --wic --bootfiles "ramdisk.cpio.gz.u-boot boot.scr Image system.dtb"

```

DANGER: The next part involve the dd command writing on disks!!! As always with the dd command, thou have to be **VERY** careful on what arguments thou give. Selecting the wrong disk will result on the destruction of thy data !! If you are unsure of what to do, seek assistance !

Finally, and now an image has been generated, it can be burnt to a micro-SD card the usual way:

```

1 sudo dd if=./images/linux/petalinux-sdimage.wic \
2   of=/dev/sda status=progress bs=8M && sync

```

From that point, the media can be ejected, put in the board and the boot process should happen successfully.

F.2 First login

On the first start, if you have a serial terminal open to the KR260 board, you'll be prompted to enter a login. The default user is petalinux. You'll then be prompted to enter a password.

F.3 Testing openamp "echo-test"

A good and out-of-the-box way to try and confirm that the OpenAMP setup was done properly is to run the so-called "echo-test". This program will basically send series of packages to the R5F core and expect them to be returned.

As presented in the official documentation⁴⁶, this can be simply done with the following commands. If the return of the echo_test commands states that zero error occurs, the system should be good to go:

```

1 sudo -s
2 echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware
3 echo start > /sys/class/remoteproc/remoteproc0/state
4 echo_test
5 echo stop > /sys/class/remoteproc/remoteproc0/state

```

⁴⁶https://xilinx.github.io/kria-apps-docs/openamp/build/html/openamp_landing.html#openamp-demos

F.4 "echo-test" performance comparison

Now we have a working demonstration for RPMsg, a goal was to compare this system between boards.

Here is the complete sequence in order to archive a timely measured echo_test run, namely:

- Installation of the build dependencies
- Clone of the Xilinx repository
- Patching of the main source to add the time measurement
- Building of the test software
- Communication with the R5F, with reload of the firmware

```

1 sudo dnf install git make gcc binutils packagegroup-core-buildessential
2 sudo ln -s /usr/bin/aarch64-xilinx-linux-gcc /bin/cc
3 git clone https://github.com/OpenAMP/openamp-system-reference.git
4 cd openamp-system-reference/examples/linux/rpmsg-echo-test/
5 wget https://gitlab.com/sunoc/xilinx-krria-kv260-documentation/-/raw/main/src/echo_test.patch
6 patch -u -b echo_test.c -i echo_test.patch
7 make
8 sudo -s
9 echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware
10 echo start > /sys/class/remoteproc/remoteproc0/state
11 ./echo_test
12 echo stop > /sys/class/remoteproc/remoteproc0/state

```

The figure 33 below shows a comparison for the data throughput that is reachable for a packages round trip of individual RPMsg packages for both PetaLinux running on the KR260 board and Ubuntu running on the KV260.

A difference exist but it is not as significant as to explain the rather slow transfer rate when used in combination with ROS2 DDS.

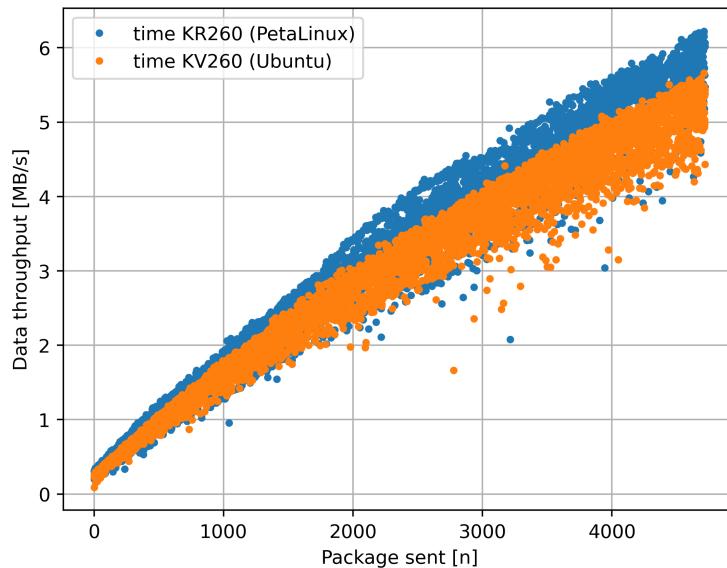


Figure 33: A graph of the data transfer rate for individual packages to be transmitted and received back from the Linux.

This fast delay for data transmission was later confirmed by measurement done using an external logic analyser⁴⁷.

F.5 Enabling SSH

F.6 Setting up a static IP address for PetaLinux

This is not a trivial task in petalinux.

As it appears, the classic /etc/network/interfaces must be modified, however and for some unknown reason the DHCP keep being used even when disabled, and the actual static interface is not used until restarted.

Multiple steps are required in order to mitigate all of this.

First, we can make our standard interfaces configuration, in the /etc/network/interfaces:

```
1 # Wired or wireless interfaces
2 auto eth0
3 iface eth0 inet static
4   address 192.168.11.107
5   netmask 255.255.255.0
6   network 192.168.11.0
7   gateway 192.168.11.1
```

Then we need to have systemd to run a script after boot time to restart the eth0 interface.

A very basic script, as follow need to be placed somewhere on the system, for example here, in the /home/root/net.sh:

```
1#!/bin/sh -e
2ifdown eth0
3ifup eth0
4exit 0
```

Then, we want to create a systemd service that will run this script as idle, i.e. after all the rest. Said file, in our case named updown.service must be placed in the /etc/systemd/system/ directory:

```
1 [Unit]
2 Description=Restart once more the eth0 network to enable the static IP
3
4 [Service]
5 Type=idle
6 ExecStart=/bin/sh /home/root/net.sh
7
8 [Install]
9 WantedBy=multi-user.target
```

Finally, as a root we need to set the correct right for these files and finally enabling the service before reboot:

```
1 sudo -s
2 chmod +x /home/root/net.sh
3 chmod 644 /etc/systemd/system/updown.service
4 systemctl enable updown.service
5 reboot now
```

With all of this, on the next reboot of the board, the static IP should be available alongside the DHCP IP.

⁴⁷<https://gitlab.com/sunoc/saleae-logic-8-documentation>

G References

References

- [1] K. Belsare, A. C. Rodriguez, P. G. Sanchez, J. Hierro, T. Kolcon, R. Lange, I. Lutkebohle, A. Malki, J. M. Losa, F. Melendez, M. M. Rodriguez, A. Nordmann, J. Staschulat, and J. von Mendel, *Micro-ROS*, pp. 3–55. Cham: Springer International Publishing, 2023.