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

This guide will present how to setup and use Xilinx's KRIA boards, 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 SoC comprising both a general purpose ARM core (Cortex A53), capable of running a Linux distribution, as well as another ARM core, real-time enabled, capable to run a RTOS (Cortex R5F). The figure 1 below shows a schematic view of the overall system we are trying to archive.

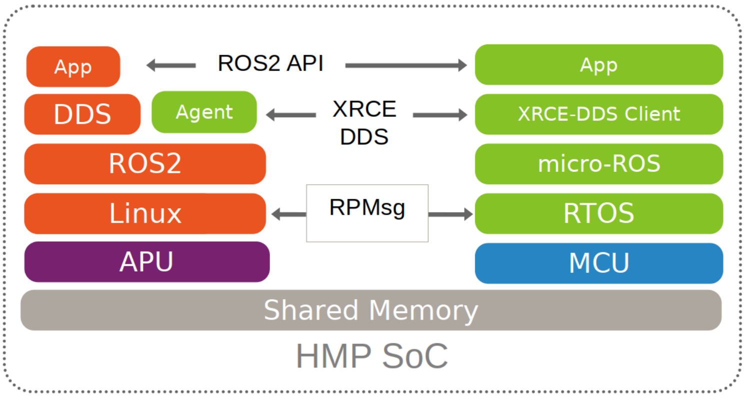

Figure 1: The Linux and ROS2 environment (orange) will communicate with the real-time, FreeRTOS and micro-ROS side (blue) using RPMsg (shared memory).

This document will give a step-by-setep, chapter by chapter indication on how to go from a new board to a system ready to be used for testing internal communication for ROS, between both types of cores using the SoC shared memory.

2 Boot firmware update

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 documentation[1], but I will present it step by step here.

2.1 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 page[2] on the topic, in the table detailing what version is best suited for which board.

In our case, we want to access this[3] page. AMD login will be asked, then download of firmware can be done.

2.2 Reaching the board recovery tool

Now the firmware `.bin` image is available[4], 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.
- 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.

2.3 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.

[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`

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

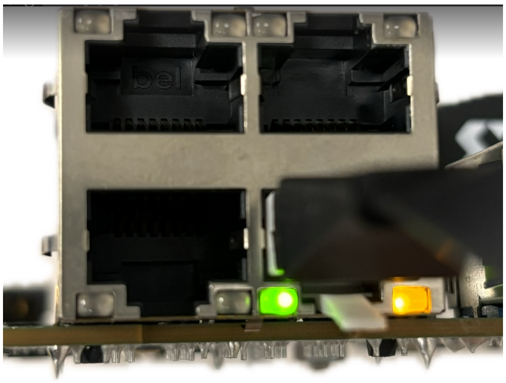

Figure 2: Port to be used for the Ethernet Boot Recovery Tool access on the KR260 board.

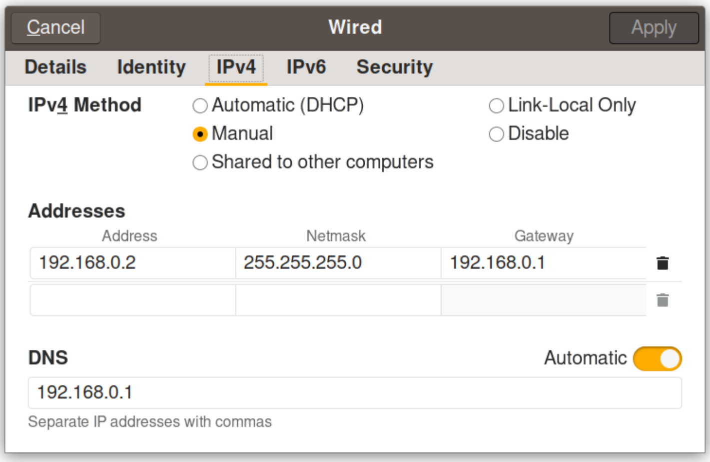

Figure 3: IPv4 settings for accessing the Recovery Tool.

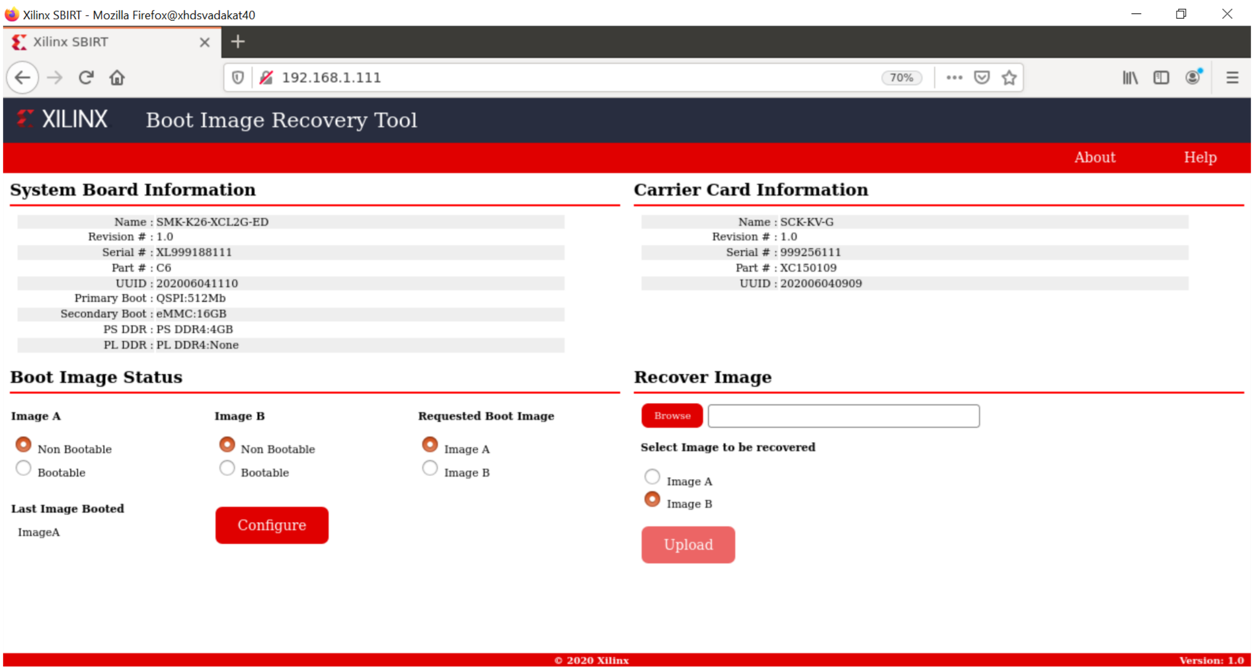

Figure 4: The recovery tool for the board, access from Firefox. We can see board information at the center, and the tools to upload the firmware at the bottom of the page.

3 Installing Ubuntu LTS 22.04

Withe 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 section[5]. The figure 5 below shows where this operating system sits in the general system we are implementing.

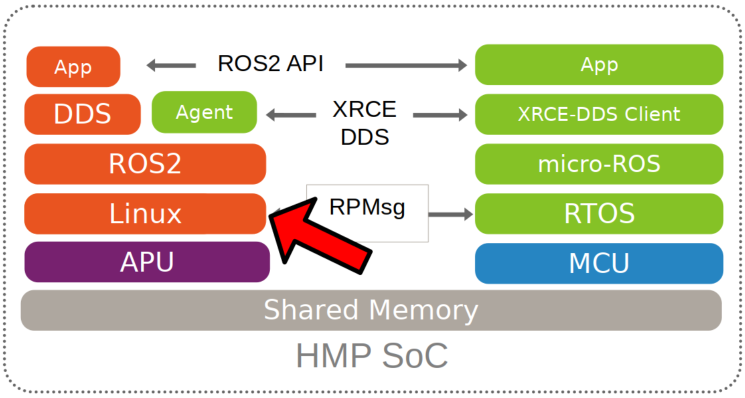

Figure 5: The Linux operating system (red border)runs on the APU (application, general purpose) side of the Kria board CPU. It is the base layer for the ROS2 system.

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 !! <u>If you are unsure of what to do, seek assistance !</u>

With the image available on thy machine and a SD card visible as `/dev/sda` device[6] one can simply run the dd command as follow to write the image to a previously formatted drive (here `/dev/sda`):

```
unxz iot-limerick-kria-classic-desktop-2204-x07-20230302-63.img.xz
sudo dd if=iot-limerick-kria-classic-desktop-2204-x07-20230302-63.img \
    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):

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

[5]The 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).

[6]Again, it is <u>critical</u> to be 100\the correct device!

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

```
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 it's 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):

```
kria# ip addr

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 file[7].

The name of the files might vary:

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

You can then set the wanted IP as follow[8]:

```
network:
  renderer: NetworkManager
  version: 2
  ethernets:
    eth0:
      dhcp4: false
      addresses:
        - 192.168.11.107/24
      routes:
        - to: default
          via: 192.168.11.1
      nameservers:
        addresses: [192.168.11.1]
```

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

```
sudo netplan apply
```

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

[8] For 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`.

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`:

```
export https_proxy="http://<AXIA_username>:\
       <AXIA_psw>@proxy.ic.nanzan-u.ac.jp:8080"

echo "export http_proxy=\""$https_proxy"\"" >> ~/.bashrc
echo "export https_proxy=\""$https_proxy"\"" >> ~/.bashrc
echo "export ftp_proxy=\""$https_proxy"\"" >> ~/.bashrc
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 you 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:

```
sudo visudo

%sudo   ALL=(ALL:ALL) NOPASSWD: ALL
```

Again, this is merely a convenience setup for devices staying at you 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:

```
sudo apt-get update
sudo apt-get dist-upgrade
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.

```
sudo add-apt-repository ppa:ubuntu-xilinx/updates
sudo add-apt-repository ppa:xilinx-apps/ppa
sudo apt update
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 purges 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 order[9], but the system fan runs sensibly slower without all of this stuff:

```
sudo systemctl disable snapd.service
sudo systemctl disable snapd.socket
sudo systemctl disable snapd.seeded.service

sudo snap list #show installed package, remove then all:
sudo snap remove --purge firefox
sudo snap remove --purge gnome-3-38-2004
sudo snap remove --purge gnome-42-2204
sudo snap remove --purge gtk-common-themes
sudo snap remove --purge snapd-desktop-integration
sudo snap remove --purge snap-store
sudo snap remove --purge bare
sudo snap remove --purge core20
sudo snap remove --purge core22
sudo snap remove --purge snapd
sudo snap list # check that everything is uninstalled

sudo rm -rf /var/cache/snapd/
sudo rm -rf ~/snap
sudo apt autoremove --purge snapd

# check once more that there is no more snap on the system
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:

```
sudo apt-get autoremove --purge yaru-theme-icon \
     fonts-noto-cjk yaru-theme-gtk vim-runtime \
     ubuntu-wallpapers-jammy humanity-icon-theme

sudo apt-get autoclean
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 benefits[10].

It is possible to disable them as follow:

```
sudo systemctl disable systemd-networkd-wait-online.service
sudo systemctl disable NetworkManager-wait-online.service
sudo systemctl disable cups.service
sudo systemctl disable docker.service
```

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

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

```
sudo systemctl disable containerd.service
sudo systemctl disable cloud-init-local.service
```

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

```
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 itself[11]:

```
sudo apt-get update && sudo apt-get install python3 python3-pip python3-venv python3-virtualenv

virtualenv myjupyter
source ./myjupyter/bin/activate
python3 -m pip install jupyter pandas numpy matplotlib scipy

sudo reboot now
```

Then in a terminal on your host machine (not on the KRIA board), you can run the following command[12] to bind local ports:

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

Then on the opened SSH shell to the KRIA board:

```
source ./myjupyter/bin/activate
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.

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 guides[13 , 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.

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

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

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

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

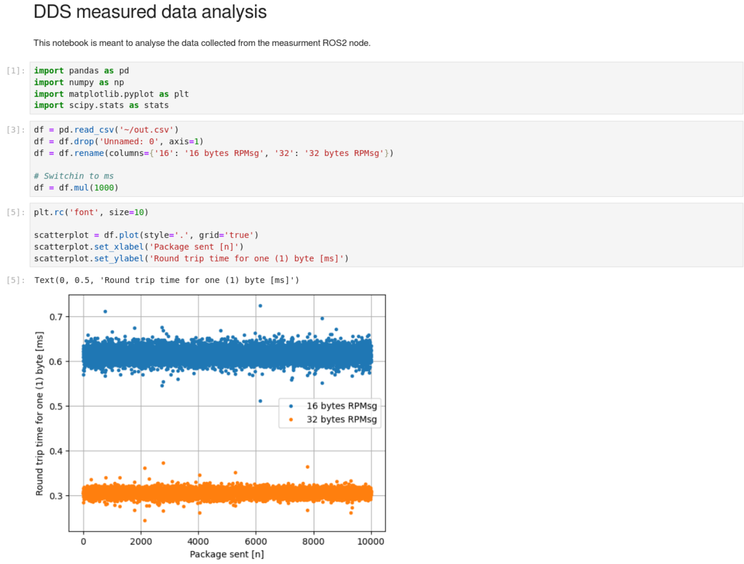

Figure 6: A test Jupyter Notebook for CSV data analysis.

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):

```
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.

```
wget
↪ https://gitlab.com/sunoc/xilinx-kria-kv260-documentation/-/raw/7a8f7c4e66e09b9d66aba8d2e08fc446ff485ca8/src/system.patch

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 needed[15]:

```
sudo nano /etc/default/flash-kernel

LINUX_KERNEL_CMDLINE="quiet splash cma=512M cpuidle.off=1"
LINUX_KERNEL_CMDLINE_DEFAULTS=""
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:

[15]The 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.

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

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

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

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

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:

```
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo \
    "deb [arch="$(dpkg --print-architecture)" \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && \
         echo "$VERSION_CODENAME")" stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli \
     containerd.io docker-buildx-plugin docker-compose-plugin
sudo usermod -aG docker $USER
newgrp docker

docker run hello-world
```

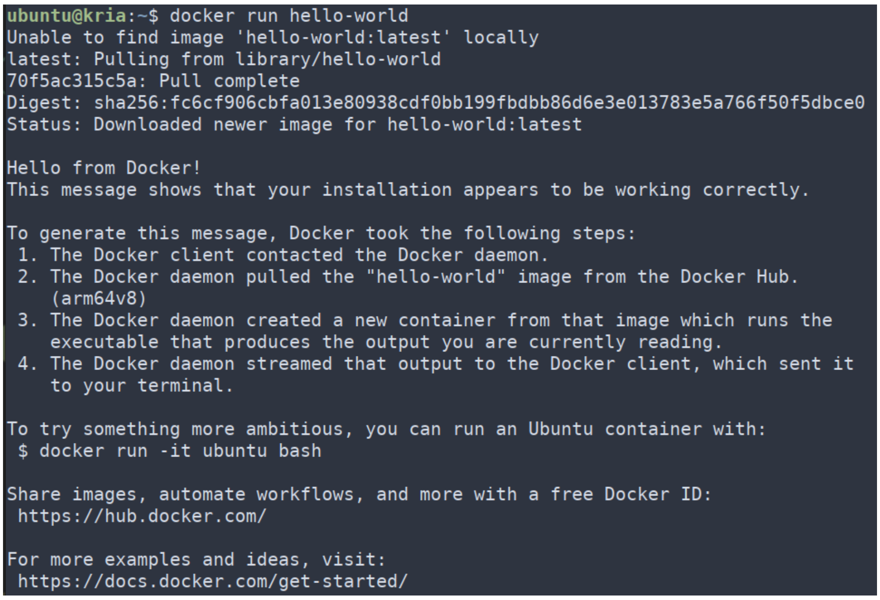

Figure 7: The return of a successful run of the `hello world` test Docker container.

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.

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

3.2.12 TODO Using a PetaLinux kernel in Ubuntu

4 RPMsg standalone evaluation

4.1 TODO RPMsg Cortex R5F demonstration firmware

4.2 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 !

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

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

5 Building micro-ROS as a static library

This section will present the way to build manually the micro-ROS system as static library that can be used to port it to a different platform.

5.1 Initial setup

In this section, the goal is to build the micro-ROS library in order to be able to integrate it's 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 Docker[16] 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:

```
pushd /home/$USER/Downloads
wget
↪  https://developer.arm.com/-/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.
tar -xvf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
popd

toolchain="/home/$USER/Downloads/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/"


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:

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

5.2 Building the static library

Now we are in the ROS2 container, we can build the micro-ROS firmware as presented in the dedicated micro-ROS guide: #+BEGIN$_{\text{SRC}}$ sh echo 127.0.0.1 $HOSTNAME ¿¿ /etc/hosts sudo apt update sudo apt-get -y install python3-pip \ wget \ nano

. /opt/ros/ROS_{\text{DISTRO}}$/setup.bash

mkdir microros$_{\text{ws}}$ cd microros$_{\text{ws}}$ git clone -b ROS_{\text{DISTRO}}$ \ `https://github.com/micro-ROS/micro_ros_setup.git` \

[16] If 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.

6 Setting up a Vitis IDE project

As visible in the official documentation[17], 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.

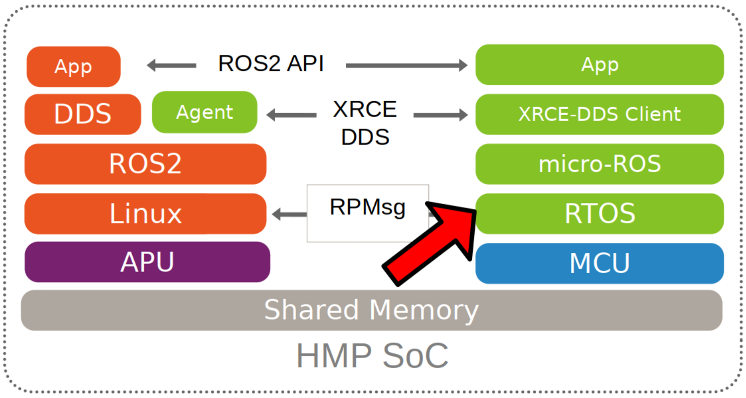

Figure 8: The FreeRTOS firmware and it's application (red border) are running on the real-time capable side of the Kria CPU. A micro-ROS application is shown here, but any real-time firmware will be deployed in the same way.

6.1 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 page[18]. You will need to get a file named something like `Xilinx_Unified_2022.2_1014_8888_Lin64.bin`[19].

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.

6.1.1 Dependencies & installation

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

```
sudo apt-get -y update

sudo apt-get -y install libncurses-dev \
    ncurses-term \
    ncurses-base \
```

[17] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1837006921/OpenAMP+Base+Hardware+Configurations/#Build-RPU-firmware

[18] https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html

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

```
    ncurses-bin \
    libncurses5 \
    libtinfo5 \
    libncurses5-dev \
    libncursesw5-dev
```

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

```
./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:

```
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 is 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.

6.1.2 Platform configuration file

With a Xilinx account, a `.bsp` archive can be downloaded for the taget platform[20].

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

```
tar xvfz xilinx-kr260-starterkit-v2022.2-10141622.bsp
ls xilinx-kr260-starterkit-2022.2/hardware/xilinx-kr260-starterkit-2022.2/
```

6.1.3 [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 Vitis[21].

A Xilinx dedicated repository[22] 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 forum[23], 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:

```
sudo apt-get update
sudo apt-get install cmake tcl libidn11-dev \
    libidn-dev libidn12 idn
sudo ln -s /usr/lib/x86_64-linux-gnu/libidn.so.12 \
```

[20] https://www.xilinx.com/member/forms/download/xef.html?filename=xilinx-kr260-starterkit-v2022.2-10141622.bsp

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

[22] https://github.com/Xilinx/kria-vitis-platforms

[23] https://support.xilinx.com/s/question/0D52E00006jrzsYSAQ/platform-project-cannot-be-created-on-vitis?language=en_US

```
    /usr/lib/x86_64-linux-gnu/libidn.so.11

cd ~/Xilinx
git clone --recursive \
    https://github.com/Xilinx/kria-vitis-platforms.git
cd kria-vitis-platforms/k26/platforms
export XILINX_VIVADO=/home/$USER/Xilinx/Vivado/2022.2/
export XILINX_VITIS=/home/$USER/Xilinx/Vitis/2022.2/
make platform PLATFORM=k26_base_starter_kit
```

6.2 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.

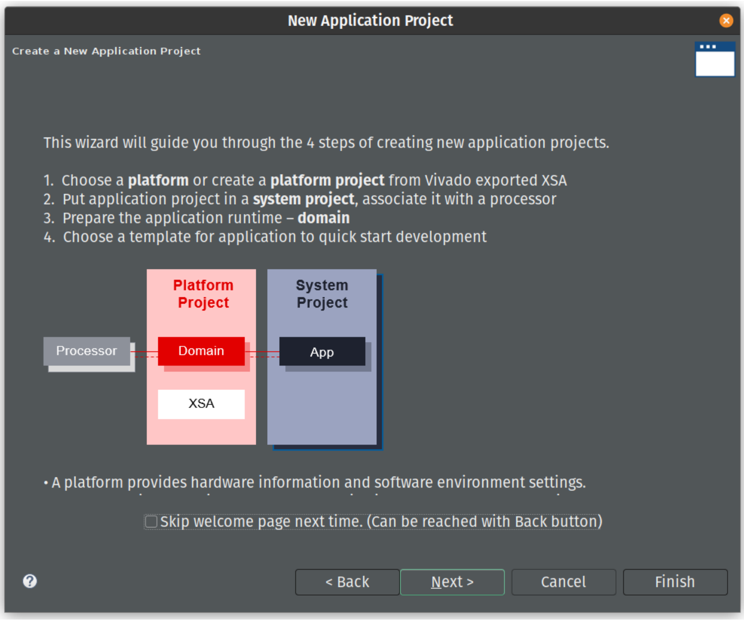

Figure 9: We are starting with creating a "New Application Project" You should be greeted with this wizard window. Next.

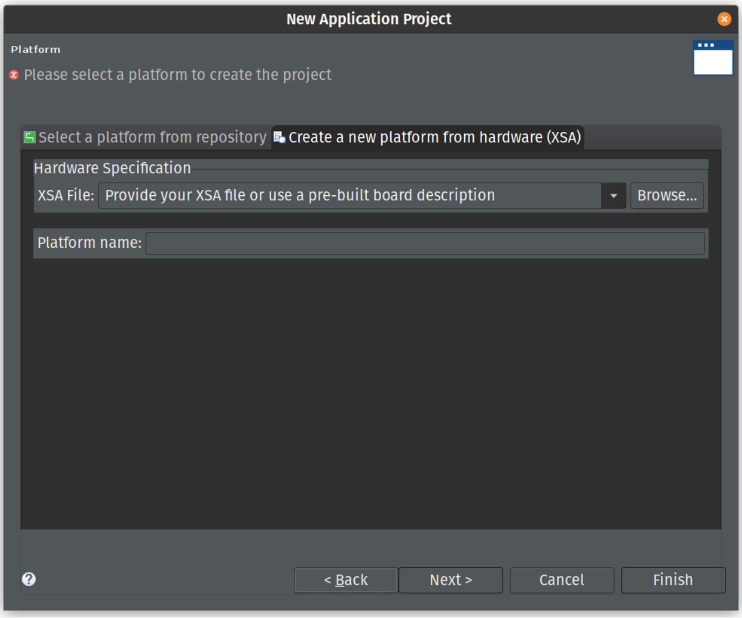

Figure 10: For the platform, we need to get our build Kria configuration. In the "Create a new platform" tab, click the "Browse..." button.

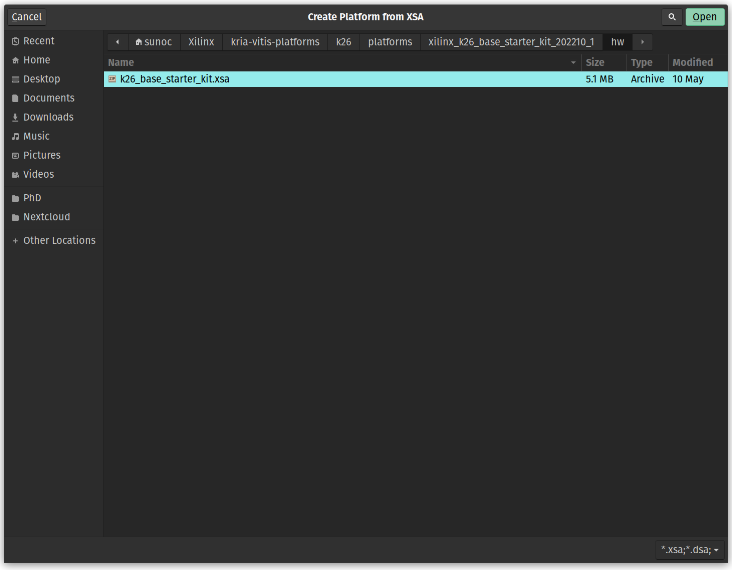

Figure 11: In the file explorer, we should navigate in the "k26" directory, where the configuration file was build. From here we are looking for a ".xsa" file, located in a "hw" directory, as visible.

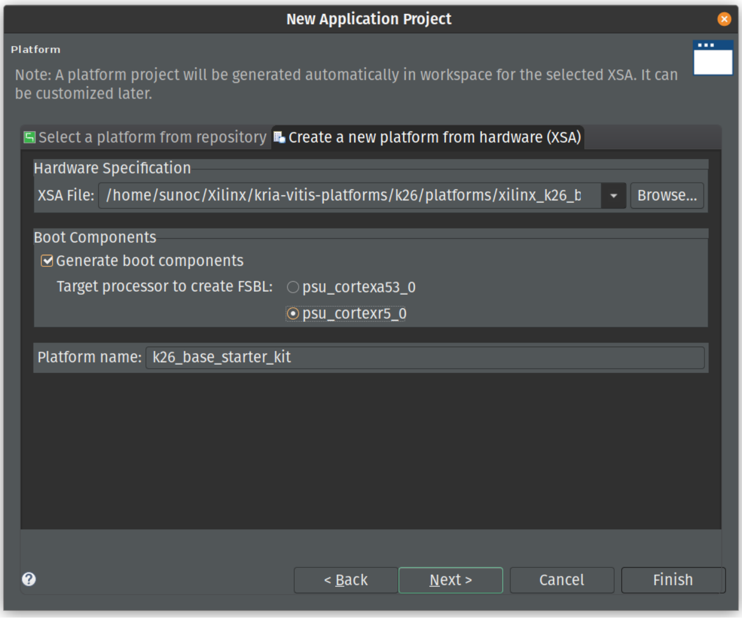

Figure 12: With the configuration file loaded, we can now select a name for our platform, but most importantly, we have to select the "psu Cortex5 0" core as a target. The other, Cortex 53 is the APU running Linux.

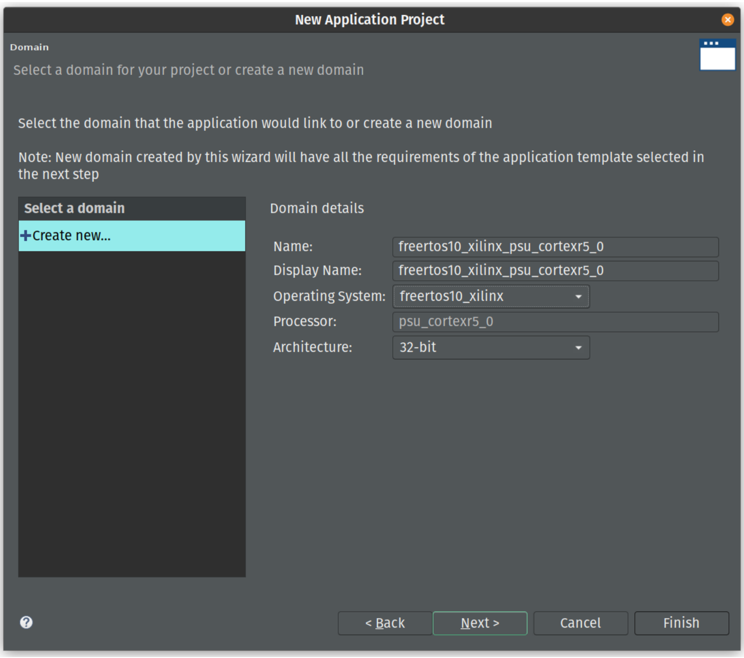

Figure 13: Here, we want to select "freertos10 xilinx" as our Operating System. The rest can remain unchanged.

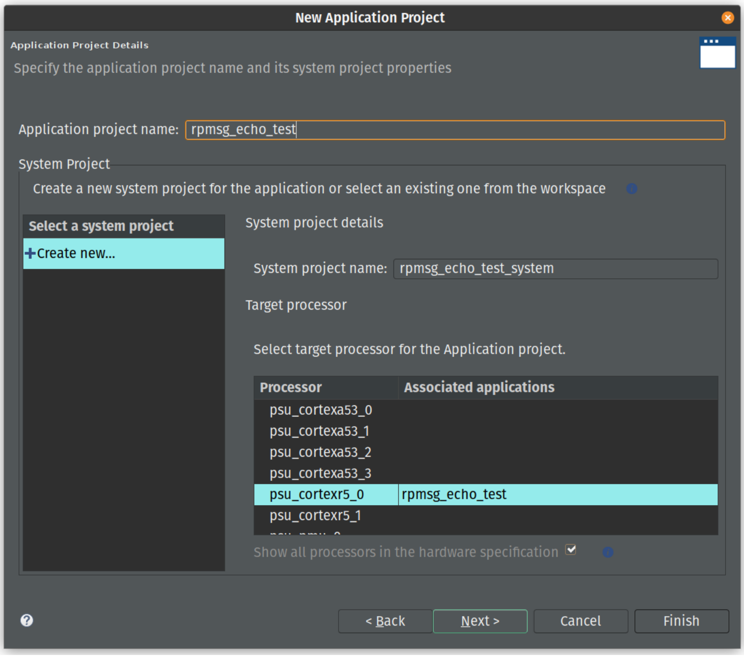

Figure 14: In this next window, we can give a name to our firmware project. It is also critical here to select the core we want to build for. Once again, we want to use the "psu cortex5 0".

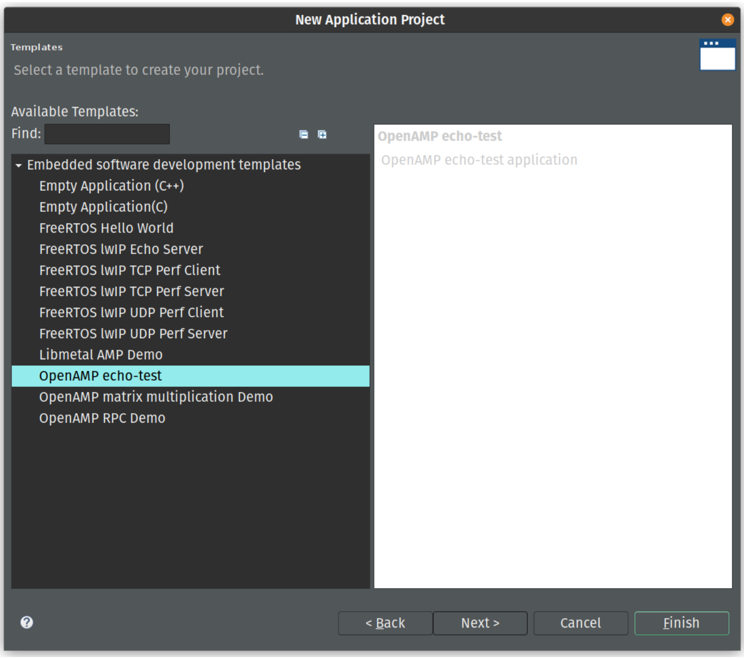

Figure 15: Finally, we can select the demonstration template we are going to use; here we go with "OpenAMP echo-test" since we want to have some simple try of the RPMsg system. Finish.

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: lscript.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.

6.3 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:

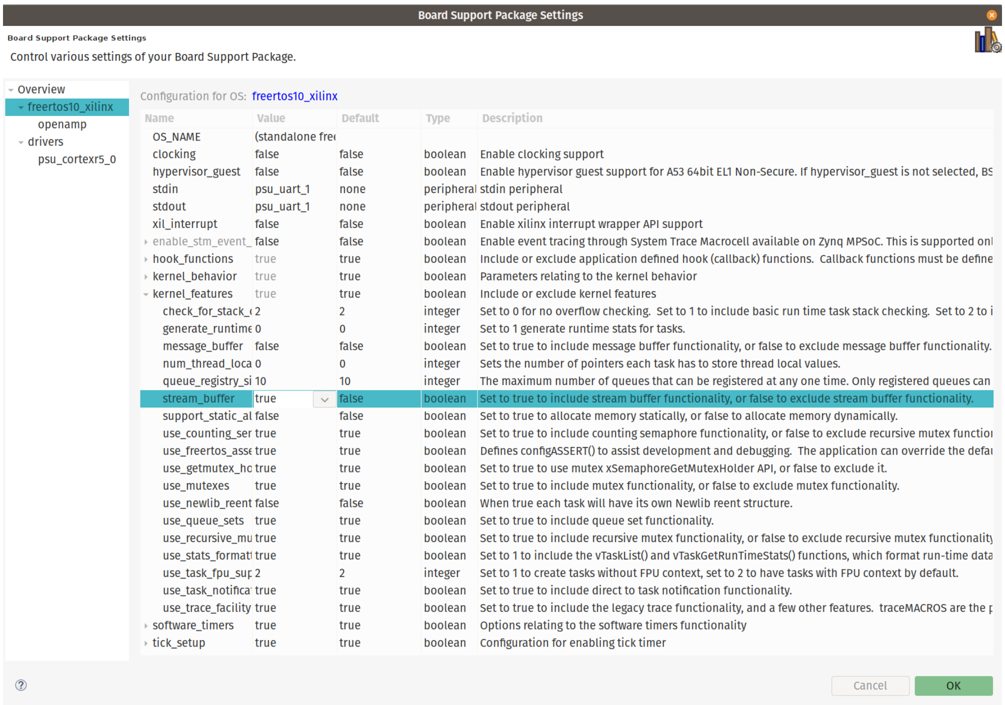

Figure 17: Enabling Stream Buffer in the Vitis IDE setting: this is a setting that can be found in the "platform.spr" element of your project (the platform, not the firmware project itself). From that file, you can access the settings with the button "Modify BSP Settings", and then as visible, in the 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 you header file (in our project, this will happen in the `microros.h` header file), before the `#include "FreeRTOS.h"` in order to override the setting from this include.

6.4 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 you home directory, at `~/workspace`, and that the Docker container is named `ros_build`:

```
mkdir /home/$USER/workspace/microros_lib

docker cp ros_build:/microros_ws/firmware/build/\
      libmicroros.a /home/$USER/workspace/microros_lib/

docker cp ros_build:/microros_ws/firmware/build/include \
      /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.

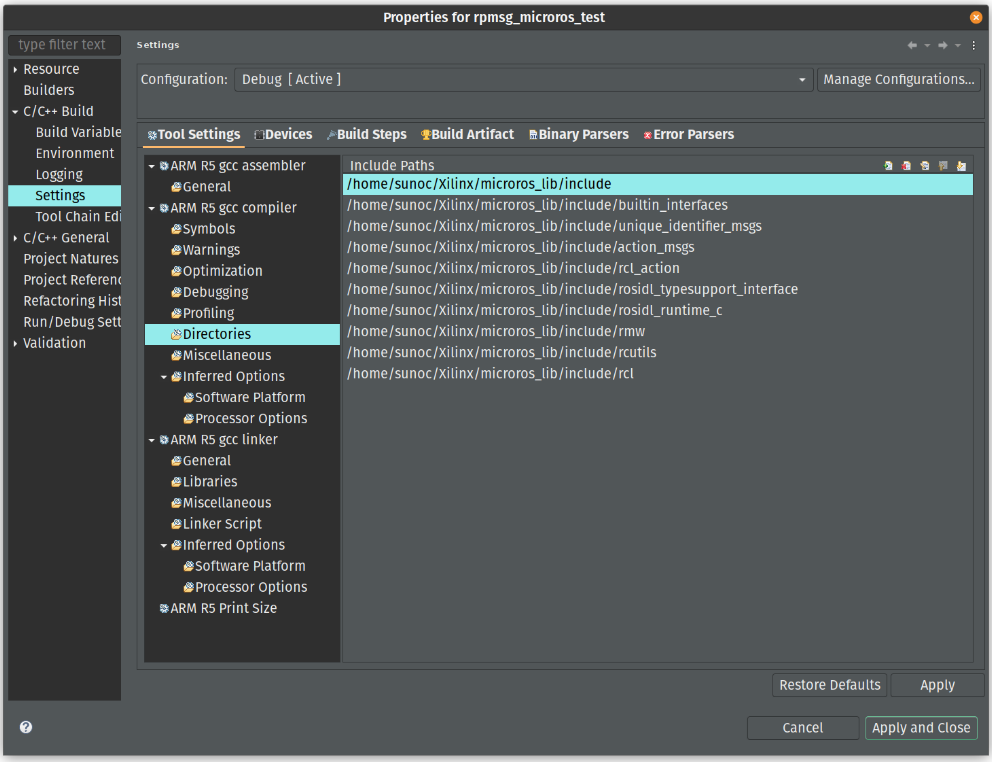

Figure 18: Firstly, in the "C/C++ Build" settings of your firmware project, under the "Settings" menu, you should find the gcc compiler "Directories". In here you should add the "include" directory of your library. Be careful however, if your include files are in a second layer of directory (as it is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

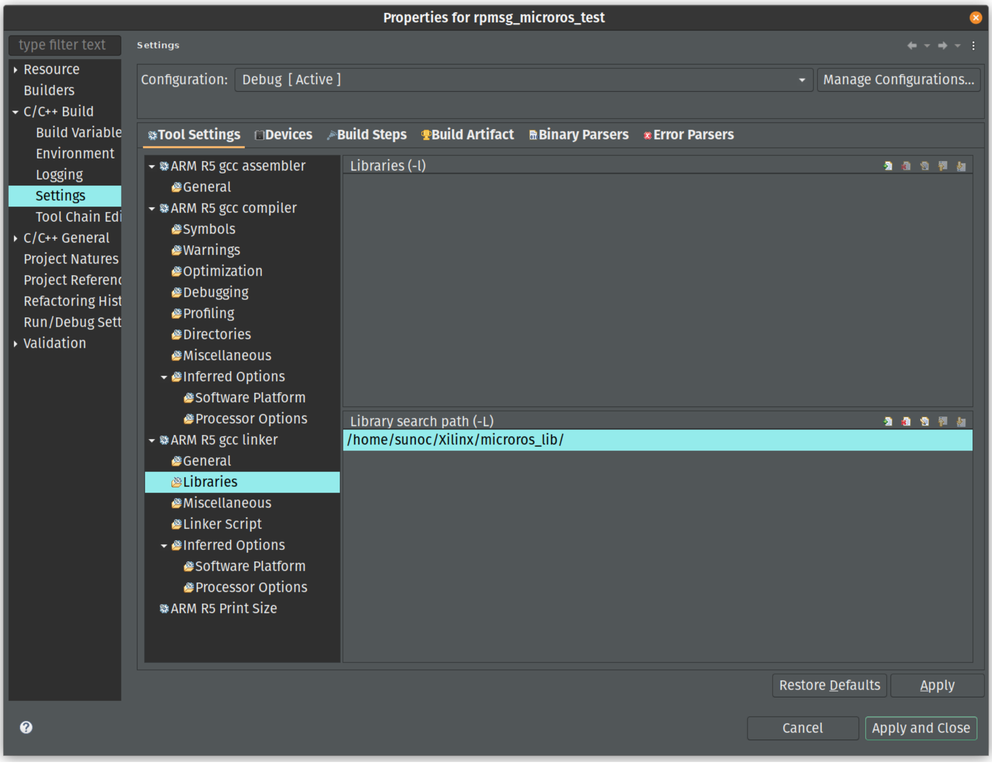

Figure 19: Secondly, in the gcc linker "Libraries", you can add the top level directory of your library. In our case, it is the directory that contains both the "include" directory added earlier, and also the "libmicroros.a" file.

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:

```
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#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.

7 ROS2 host system setup

This is focused on the ROS2 system being used on the Kria board as a base to run and / or build micro-ROS components.

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 28 below shows what part of the overall system we are talking about here.

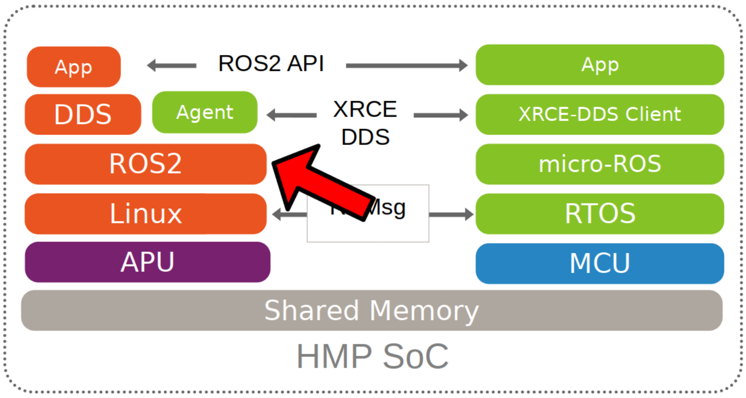

Figure 20: The ROS2 middle (red border) runs on top of the Linux, on the general-purpose core of the Kria board.

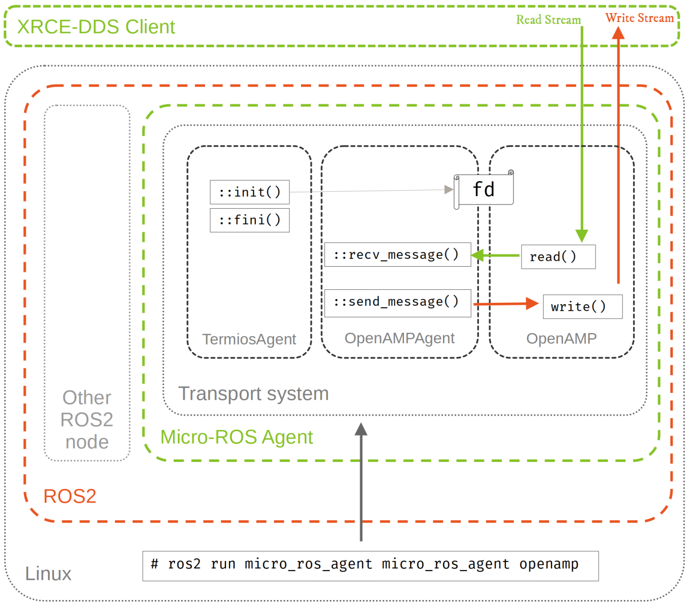

Figure 21: Methods architecture for the modified Agent node.

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

Since an Ubuntu distribution is installed on the board, the installation of ROS2 can be done[24] in a standard way, using the repository.

[24] As 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.

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 here[25].

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:

```
locale  # check for UTF-8
sudo apt update && sudo apt install -y locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale  # verify settings

sudo apt install -y software-properties-common
sudo add-apt-repository universe
sudo apt update && sudo apt install -y curl wget

wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
sudo mv ros.key /usr/share/keyrings/ros-archive-keyring.gpg
```

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

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

It is then possible to install ROS2[26] as follow:

```
sudo apt update
sudo apt upgrade -y
sudo apt install -y ros-$ROS_DISTRO-desktop \
     ros-$ROS_DISTRO-ros-base \
     python3-argcomplete \
     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:

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

And then:

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

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

[25] The `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`.

[26] This 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.

7.2 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 you 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 installer 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 container[27]. With the second command, we can execute `bash` as a way to open a terminal to the "inside" the container:

```
docker run -d --name ros_agent -it --net=host -v \
       /dev:/dev --privileged ros:iron
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 7.1 above:

```
source /opt/ros/$ROS_DISTRO/setup.bash

# Create a workspace and download the micro-ROS tools
mkdir microros_ws
cd microros_ws
git clone -b $ROS_DISTRO https://github.com/micro-ROS/\
    micro_ros_setup.git src/micro_ros_setup

# Update dependencies using rosdep
sudo apt update && rosdep update
rosdep install --from-paths src --ignore-src -y

# Install pip
sudo apt-get install python3-pip

# Build micro-ROS tools and source them
colcon build

# Download micro-ROS-Agent packages
source install/local_setup.bash
ros2 run micro_ros_setup create_agent_ws.sh

# Build step
ros2 run micro_ros_setup build_agent.sh

# Run a micro-ROS agent
ros2 run micro_ros_agent micro_ros_agent serial \
     --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 shell[28]:

[27] This 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.

[28] You 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.

```
source /opt/ros/$ROS_DISTRO/setup.bash

# Subscribe to micro-ROS ping topic
ros2 topic echo /microROS/ping
```

8 micro-ROS Client

8.1 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.

8.1.1 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.

8.1.2 [DEPRECATED] 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.

8.1.3 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 trigger by an incoming message.

This situation meant that we count not simple translate the communication layers from one to another: a non-blocking polling and buffer system needed to be put into places. The proposed solution that was implemented and that is currently being tested is showed and detailed in the figure 22 below.

The next figure 23 show a more visual representation of the two tasks and the functions used in them.

A version of this firmware is available at my gitlab repository[29]. This firmware was built using Xilinx's IDE, which setup was presented in the section 6.1, 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 22, 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 it's `rclc` system run extensive initialization tasks to:
  - Initialize the support system

[29] micro-ROS ping pong test firmware repository: `https://gitlab.com/sunoc/libmicroros_kv260`

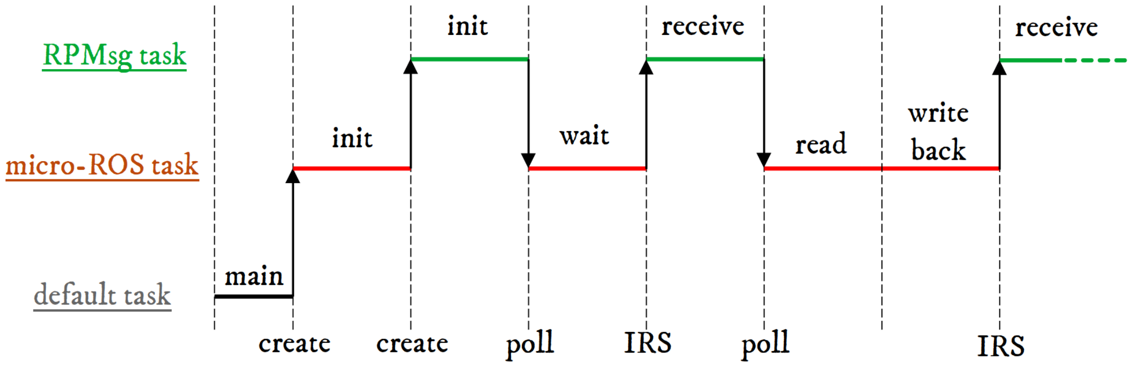

Figure 22: Two tasks are being run concurrently in order to manage the communication situation, with binary semaphore-based lock-unlock system.
The role of the micro-ROS task (red) is to make the four functions ("open", "read", "write" and "close") available and running the actual software function. 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.

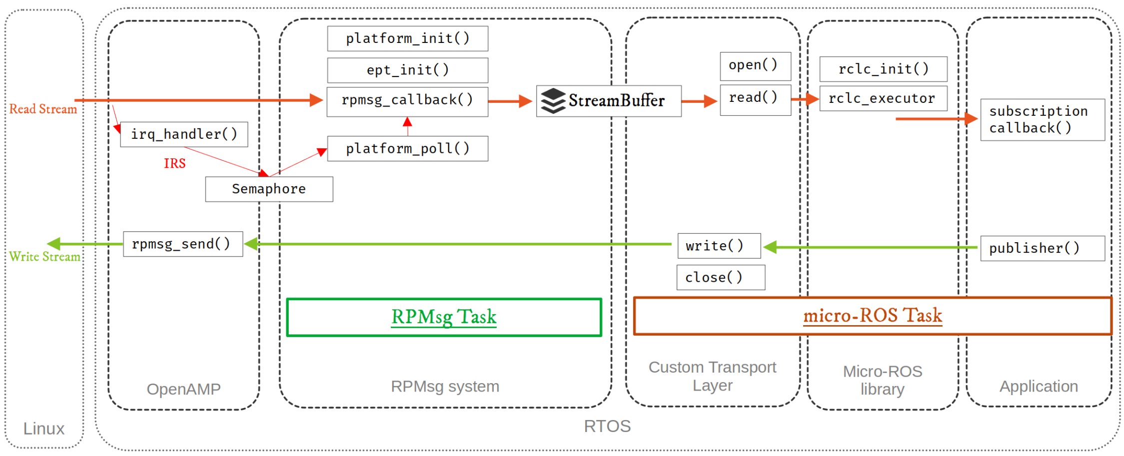

Figure 23: Functions architecture for the Client firmware.

  - Initialize the support node
  - 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.

8.2 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:

```
scp /home/sunoc/workspace/rpmsg_pingpong_microros_lib/\
    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-space[30], to test a basic RPMsg setup[31]:

```
sudo -s
mv image_echo_test /lib/firmware
echo image_echo_test  > /sys/class/remoteproc/\
     remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
echo_test
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.

[30] In 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`.

[31] It 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 4.2.

8.2.1 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 advantaged 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 debugger in a easier way.

8.3 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.

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

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

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 follow, as root:

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

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

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`:

```
proc boot_jtag { } {
############################
# Switch to JTAG boot mode #
############################
targets -set -filter {name =~ "PSU"}
# update multiboot to ZERO
mwr 0xffca0010 0x0
# change boot mode to JTAG
mwr 0xff5e0200 0x0100
# reset
rst -system
}


# Jtag connection
connect
puts stdout "Connect Jtag"

# Switching to JtagBoot
```

```
puts -nonewline stdout "Start transition to jtag boot mode..."
boot_jtag
puts stdout "done."

# Jtag disconnection
puts -nonewline stdout "Disconnect Jtag..."
disconnect
puts stdout "done."
```

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

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

8.3.1 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.

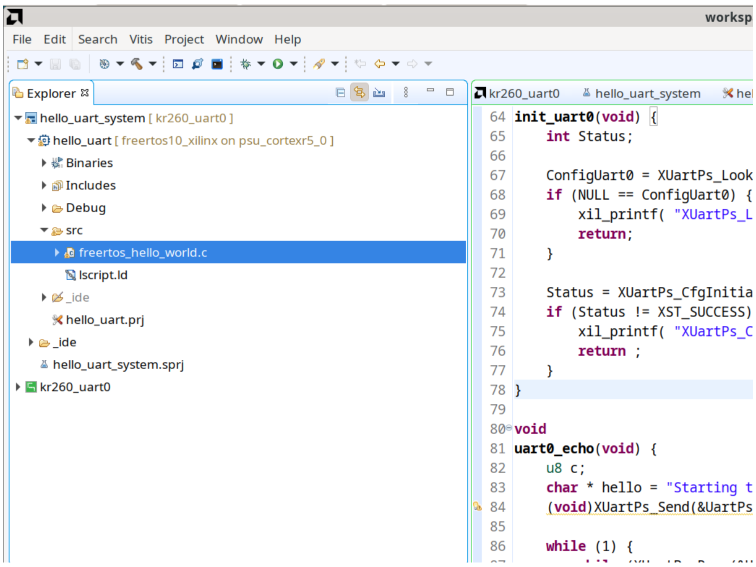

Figure 24: With a project meant to be build and run on the Cortex R5F, you must first produce the `.elf` binary file.

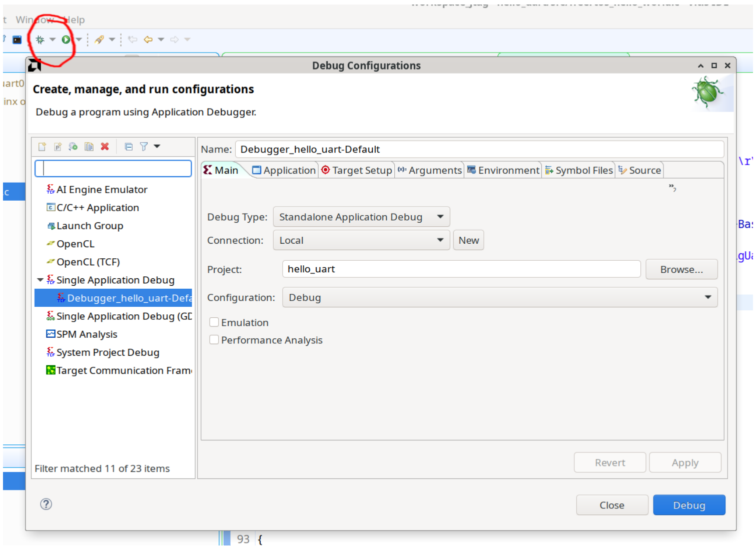

Figure 25: Then you can open the Debug Configuration window.

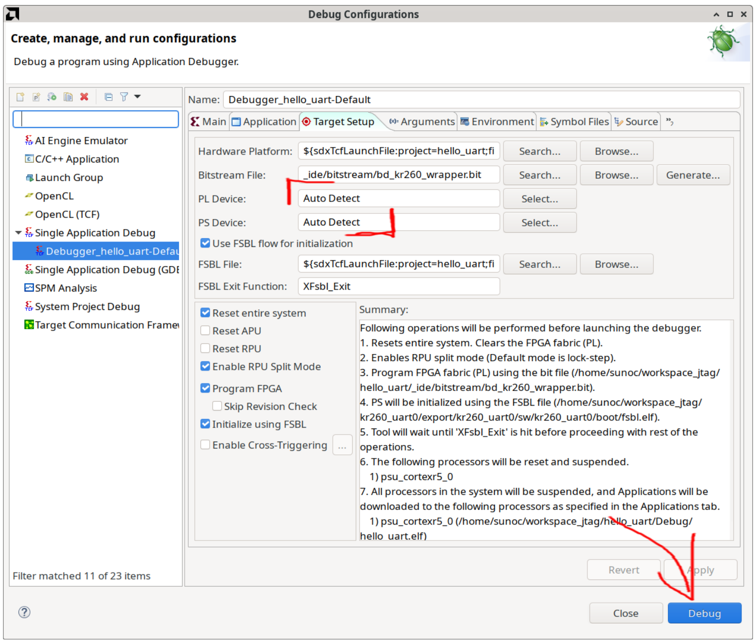

Figure 26: All parameters should stay the same. You can then run "Debug".

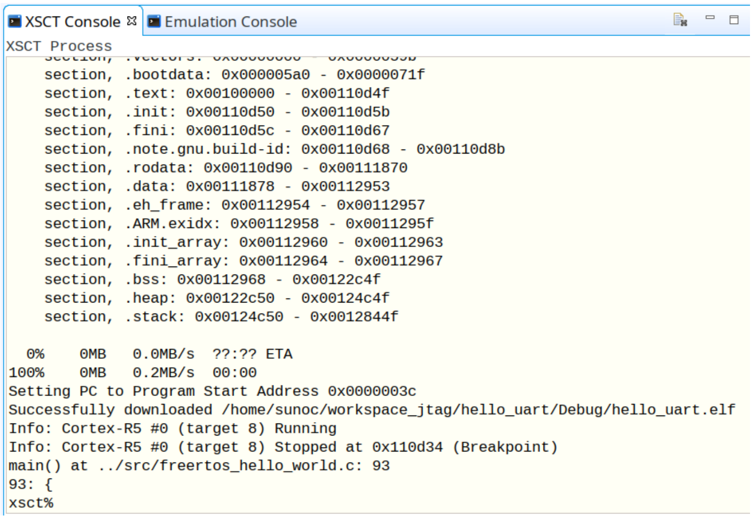

Figure 27: The XSCT console will confirm the successful load of the firmware. If not, you will get error messages and should reboot the board.

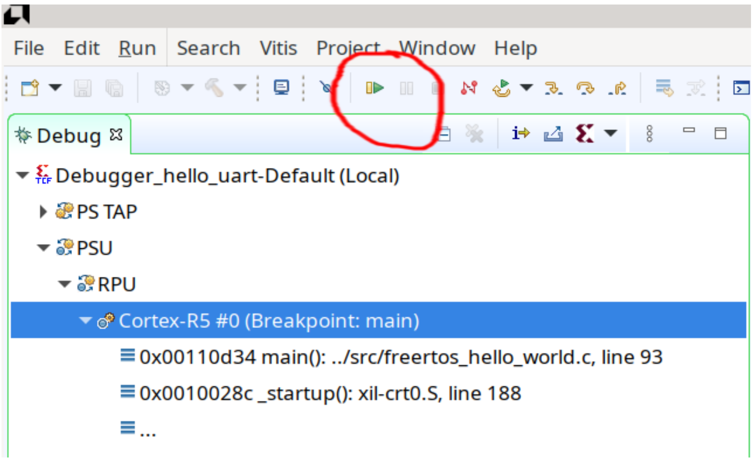

Figure 28: Selecting the Cortex R5, core #0, you can then run the application.

9 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 ROS2[32]. 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.

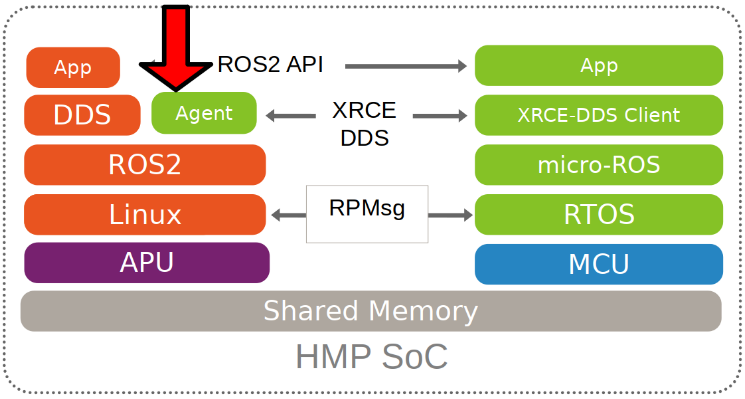

Figure 29: The agent (red border) allows for a micro-ROS instance to communicate with a ROS2 system. It is deployed on the Linux side, as a ROS2 node.

9.1 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.

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

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

9.2 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 it the same as usual, just pulling from my fork of eProsima repository and using the `develop` branch:

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

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

```
cmake ..
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 follow[33] , [34]:

```
source /opt/ros/$ROS_DISTRO/setup.bash

./MicroXRCEAgent serial --dev /dev/null --verbose 6
```

[33]The `--dev /dev/null` option is only a placeholder as the serial requires it but not RPMsg.

[34]The `--verbose 6` allows to see the maximum amount of information about the data being transmitted.

10 micro-ROS Talker

11 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 8.2.
- On the same Linux you plan to run the node, you will need to run and keep the Agent, as presented in the section 9.
- 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 about ROS2.

12 Monitoring and performances evaluation WORK_IN_PROGRESS

13 Shared memory data transfer system WORK_IN_PROGRESS

14 Conclusion & future WORK_IN_PROGRESS

A DTO patch

This file is available in this repository: system.patch

```
--- system.dts.orig        2024-04-15 01:36:19.698949885 +0000
+++ system.dts       2024-09-05 08:57:00.690107364 +0000
@@ -196,7 +196,7 @@

         firmware {

-                zynqmp-firmware {
+                zynqmp_firmware: zynqmp-firmware {
                         compatible = "xlnx,zynqmp-firmware";
                         #power-domain-cells = <0x01>;
                         method = "smc";
@@ -708,7 +708,7 @@
                         phandle = <0x41>;
                 };

-                interrupt-controller@f9010000 {
+                gic: interrupt-controller@f9010000 {
                         compatible = "arm,gic-400";
                         #interrupt-cells = <0x03>;
                         reg = <0x00 0xf9010000 0x00 0x10000 0x00 0xf9020000 0x00 0x20000 0x00 0xf9040000 0x00 0x20000 0x00 0xf9060000 0x00 0x20000>;
@@ -1567,7 +1567,7 @@
                         pinctrl-names = "default";
                         u-boot,dm-pre-reloc;
                         compatible = "xlnx,zynqmp-uart\0cdns,uart-r1p12";
-                        status = "okay";
+                        status = "disabled";
                         interrupt-parent = <0x05>;
                         interrupts = <0x00 0x16 0x04>;
                         reg = <0x00 0xff010000 0x00 0x1000>;
@@ -1912,18 +1912,6 @@
                 reg = <0x00 0x00 0x00 0x80000000 0x08 0x00 0x00 0x80000000>;
         };

-        reserved-memory {
-                #address-cells = <0x02>;
-                #size-cells = <0x02>;
-                ranges;
-
-                pmu@7ff00000 {
-                        reg = <0x00 0x7ff00000 0x00 0x100000>;
-                        no-map;
-                        phandle = <0x7a>;
-                };
-        };
-
         gpio-keys {
                 compatible = "gpio-keys";
                 autorepeat;
@@ -1964,6 +1952,174 @@
```

```
                  pwms = <0x1f 0x02 0x9c40 0x00>;
          };

+        reserved-memory {
+            #address-cells = <2>;
+            #size-cells = <2>;
+            ranges;
+            rpu0vdev0vring0: rpu0vdev0vring0@3ed80000 {
+                no-map;
+                reg = <0x0 0x3ed80000 0x0 0x4000>;
+            };
+            rpu0vdev0vring1: rpu0vdev0vring1@3ed84000 {
+                no-map;
+                reg = <0x0 0x3ed84000 0x0 0x4000>;
+            };
+            rpu0vdev0buffer: rpu0vdev0buffer@3ed88000 {
+                no-map;
+                reg = <0x0 0x3ed88000 0x0 0x100000>;
+            };
+            rproc_0_reserved: rproc_0_reserved@3ec00000 {
+                no-map;
+                reg = <0x0 0x3ec00000 0x0 0x180000>;
+            };
+
+            rpu1vdev0vring0: rpu1vdev0vring0@3ef80000 {
+                no-map;
+                reg = <0x0 0x3ef80000 0x0 0x4000>;
+            };
+            rpu1vdev0vring1: rpu1vdev0vring1@3ef84000 {
+                no-map;
+                reg = <0x0 0x3ef84000 0x0 0x4000>;
+            };
+            rpu1vdev0buffer: rpu1vdev0buffer@3ef88000 {
+                no-map;
+                reg = <0x0 0x3ef88000 0x0 0x100000>;
+            };
+            rproc_1_reserved: rproc_1_reserved@3ef00000 {
+                no-map;
+                reg = <0x0 0x3ef00000 0x0 0x80000>;
+            };
+            uros_buf0: uros_buf@40000000 {
+                         compatible = "shared-dma-pool";
+                         reusable;
+                         reg = <0x8 0x40000000 0x0 0x00400000>;
+                         label = "uros_buf0";
+            };
+
+        };
+        tcm_0a: tcm_0a@ffe00000 {
+            no-map;
+            reg = <0x0 0xffe00000 0x0 0x10000>;
+            status = "okay";
+            compatible = "mmio-sram";
+            power-domain = <&zynqmp_firmware 15>;
```

```
+   };
+   tcm_0b: tcm_0b@ffe20000 {
+     no-map;
+     reg = <0x0 0xffe20000 0x0 0x10000>;
+     status = "okay";
+     compatible = "mmio-sram";
+     power-domain = <&zynqmp_firmware 16>;
+   };
+   tcm_1a: tcm_1a@ffe90000 {
+     no-map;
+     reg = <0x0 0xffe90000 0x0 0x10000>;
+     status = "okay";
+     compatible = "mmio-sram";
+     power-domain = <&zynqmp_firmware 17>;
+   };
+   tcm_1b: tcm_1b@ffeb0000 {
+     no-map;
+     reg = <0x0 0xffeb0000 0x0 0x10000>;
+     status = "okay";
+     compatible = "mmio-sram";
+     power-domain = <&zynqmp_firmware 18>;
+    };
+
+   rf5ss@ff9a0000 {
+     compatible = "xlnx,zynqmp-r5-remoteproc";
+     xlnx,cluster-mode = <1>;
+     ranges;
+     reg = <0x0 0xFF9A0000 0x0 0x15000>;
+     #address-cells = <0x2>;
+     #size-cells = <0x2>;
+     r5f_0 {
+       compatible = "xilinx,r5f";
+       #address-cells = <2>;
+       #size-cells = <2>;
+       ranges;
+       sram = <&tcm_0a &tcm_0b>;
+       memory-region = <&rproc_0_reserved>, <&rpu0vdev0buffer>, <&rpu0vdev0vring0>, <&rpu0vdev0vring1>;
+       power-domain = <&zynqmp_firmware 7>;
+       mboxes = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;
+       mbox-names = "tx", "rx";
+        };
+     r5f_1 {
+       compatible = "xilinx,r5f";
+       #address-cells = <2>;
+       #size-cells = <2>;
+       ranges;
+       sram = <&tcm_1a &tcm_1b>;
+       memory-region = <&rproc_1_reserved>, <&rpu1vdev0buffer>, <&rpu1vdev0vring0>, <&rpu1vdev0vring1>;
+       power-domain = <&zynqmp_firmware 8>;
+       mboxes = <&ipi_mailbox_rpu1 0>, <&ipi_mailbox_rpu1 1>;
+       mbox-names = "tx", "rx";
+     };
```

```
+         };
+
+        zynqmp_ipi1 {
+            compatible = "xlnx,zynqmp-ipi-mailbox";
+            interrupt-parent = <&gic>;
+            interrupts = <0 29 4>;
+            xlnx,ipi-id = <7>;
+            #address-cells = <1>;
+            #size-cells = <1>;
+            ranges;
+            /* APU<->RPU0 IPI mailbox controller */
+            ipi_mailbox_rpu0: mailbox@ff990000 {
+                reg = <0xff990600 0x20>,
+                      <0xff990620 0x20>,
+                      <0xff9900c0 0x20>,
+                      <0xff9900e0 0x20>;
+                reg-names = "local_request_region",
+                        "local_response_region",
+                        "remote_request_region",
+                        "remote_response_region";
+                #mbox-cells = <1>;
+                xlnx,ipi-id = <1>;
+            };
+         };
+        zynqmp_ipi2 {
+            compatible = "xlnx,zynqmp-ipi-mailbox";
+            interrupt-parent = <&gic>;
+            interrupts = <0 30 4>;
+            xlnx,ipi-id = <8>;
+            #address-cells = <1>;
+            #size-cells = <1>;
+            ranges;
+            /* APU<->RPU0 IPI mailbox controller */
+            ipi_mailbox_rpu1: mailbox@ff3f0b00 {
+                reg = <0xff3f0b00 0x20>,
+                      <0xff3f0b20 0x20>,
+                      <0xff3f0940 0x20>,
+                      <0xff3f0960 0x20>;
+                reg-names = "local_request_region",
+                        "local_response_region",
+                        "remote_request_region",
+                        "remote_response_region";
+                #mbox-cells = <1>;
+                xlnx,ipi-id = <2>;
+            };
+         };
+         amba_pl: amba_pl@0 {
+                 #address-cells = <2>;
+                 #size-cells = <2>;
+                 compatible = "simple-bus";
+                 ranges ;
+                 udmabuf@0x00 {
+                         compatible  = "ikwzm,u-dma-buf";
+                         device-name = "udmabuf0";
```

```
+				size = <0x0 0x00200000>;
+				memory-region = <&uros_buf0>;
+			};
+			udmabuf@0x01 {
+				compatible  = "ikwzm,u-dma-buf";
+				device-name = "udmabuf1";
+				size = <0x0 0x00200000>;
+				memory-region = <&uros_buf0>;
+			};
+		};
+
		__symbols__ {
			pinctrl_usb1_default = "/firmware/zynqmp-firmware/pinctrl/usb1-default";
			pinctrl_usb0_default = "/firmware/zynqmp-firmware/pinctrl/usb0-default";
```

B Custom tool-chain CMake settings

This file is available in this repository: custom r5f toolchain.cmake

```
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_CROSSCOMPILING 1)
set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)


set(ARCH_CPU_FLAGS "-O2 -ffunction-sections -fdata-sections -fno-exceptions -mcpu=cortex-r5 -mfpu=vfpv3-d16 -mfloat-abi=hard -Wl,--gc-sections -nostdlib --param max-inline-insns-single=500 -DF_CPU=84000000L -D'RCUTILS_LOG_MIN_SEVERITY=RCUTILS_LOG_MIN_SEVERITY_NONE'")
set(ARCH_OPT_FLAGS "")

set(CMAKE_C_COMPILER arm-none-eabi-gcc)
set(CMAKE_CXX_COMPILER arm-none-eabi-g++)

set(CMAKE_C_FLAGS_INIT "-std=c11 ${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS} -DCLOCK_MONOTONIC=0" CACHE STRING "" FORCE)
set(CMAKE_CXX_FLAGS_INIT "-std=c++14 ${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS} -DCLOCK_MONOTONIC=0" CACHE STRING "" FORCE)



set(__BIG_ENDIAN__ 0)
```

C Custom Colcon meta settings

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

```
{
    "names": {
        "tracetools": {
            "cmake-args": [
                "-DTRACETOOLS_DISABLED=ON",
                "-DTRACETOOLS_STATUS_CHECKING_TOOL=OFF"
            ]
        },
        "rosidl_typesupport": {
            "cmake-args": [
               "-DROSIDL_TYPESUPPORT_SINGLE_TYPESUPPORT=ON"
            ]
        },
        "rcl": {
            "cmake-args": [
                "-DBUILD_TESTING=OFF",
                "-DRCL_COMMAND_LINE_ENABLED=OFF",
                "-DRCL_LOGGING_ENABLED=OFF"
            ]
        },
        "rcutils": {
            "cmake-args": [
                "-DENABLE_TESTING=OFF",
                "-DRCUTILS_NO_FILESYSTEM=ON",
                "-DRCUTILS_NO_THREAD_SUPPORT=ON",
                "-DRCUTILS_NO_64_ATOMIC=ON",
                "-DRCUTILS_AVOID_DYNAMIC_ALLOCATION=ON"
            ]
        },
        "microxrcedds_client": {
            "cmake-args": [
                "-DUCLIENT_PIC=OFF",
                "-DUCLIENT_PROFILE_UDP=OFF",
                "-DUCLIENT_PROFILE_TCP=OFF",
                "-DUCLIENT_PROFILE_DISCOVERY=OFF",
                "-DUCLIENT_PROFILE_SERIAL=OFF",
                "-UCLIENT_PROFILE_STREAM_FRAMING=OFF",
                "-DUCLIENT_PROFILE_CUSTOM_TRANSPORT=ON",
                "-DUCLIENT_CUSTOM_TRANSPORT_MTU=8232"
            ]
        },
        "rmw_microxrcedds": {
            "cmake-args": [
                "-DRMW_UXRCE_MAX_NODES=1",
                "-DRMW_UXRCE_MAX_PUBLISHERS=5",
                "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=5",
                "-DRMW_UXRCE_MAX_SERVICES=1",
                "-DRMW_UXRCE_MAX_CLIENTS=1",
                "-DRMW_UXRCE_MAX_HISTORY=4",
                "-DRMW_UXRCE_TRANSPORT=custom"
            ]
```

```
        }
    }
}
```

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

```
#include "microros.h"


int _gettimeofday( struct timeval *tv, void *tzvp )
{
        XTime t = 0;
    XTime_GetTime(&t);  //get uptime in nanoseconds
    tv->tv_sec = t / 1000000000;  // convert to seconds
    tv->tv_usec = ( t % 1000000000 ) / 1000;  // get remaining microseconds
    return 0;  // return non-zero for error
} // end _gettimeofday()


void UTILS_NanosecondsToTimespec( int64_t llSource,
                                  struct timespec * const pxDestination )
{
    long lCarrySec = 0;

    /* Convert to timespec. */
    pxDestination->tv_sec = ( time_t ) ( llSource / NANOSECONDS_PER_SECOND );
    pxDestination->tv_nsec = ( long ) ( llSource % NANOSECONDS_PER_SECOND );

    /* Subtract from tv_sec if tv_nsec < 0. */
    if( pxDestination->tv_nsec < 0L )
    {
        /* Compute the number of seconds to carry. */
        lCarrySec = ( pxDestination->tv_nsec / ( long ) NANOSECONDS_PER_SECOND ) + 1L;

        pxDestination->tv_sec -= ( time_t ) ( lCarrySec );
        pxDestination->tv_nsec += lCarrySec * ( long ) NANOSECONDS_PER_SECOND;
    }
}

int clock_gettime( clockid_t clock_id,
                   struct timespec * tp )
{
        TimeOut_t xCurrentTime = { 0 };

    /* Intermediate variable used to convert TimeOut_t to struct timespec.
     * Also used to detect overflow issues. It must be unsigned because the
     * behavior of signed integer overflow is undefined. */
    uint64_t ullTickCount = 0ULL;

    /* Silence warnings about unused parameters. */
    ( void ) clock_id;

    /* Get the current tick count and overflow count. vTaskSetTimeOutState()
     * is used to get these values because they are both static in tasks.c. */
```

```c
    vTaskSetTimeOutState( &xCurrentTime );

    /* Adjust the tick count for the number of times a TickType_t has overflowed.
     * portMAX_DELAY should be the maximum value of a TickType_t. */
    ullTickCount = ( uint64_t ) ( xCurrentTime.xOverflowCount ) << ( sizeof( TickType_t
    ↪  ) * 8 );

    /* Add the current tick count. */
    ullTickCount += xCurrentTime.xTimeOnEntering;

    /* Convert ullTickCount to timespec. */
    UTILS_NanosecondsToTimespec( ( int64_t ) ullTickCount * NANOSECONDS_PER_TICK, tp );

    return 0;
}
```

D.2 header file

```c
/**< Microseconds per second. */
#define MICROSECONDS_PER_SECOND    ( 1000000LL )
/**< Nanoseconds per second. */
#define NANOSECONDS_PER_SECOND     ( 1000000000LL )
/**< Nanoseconds per FreeRTOS tick. */
#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

```
#include "allocators.h"

//int absoluteUsedMemory = 0;
//int usedMemory = 0;

void * __freertos_allocate(size_t size, void * state){
  (void) state;
  LPRINTF("-- Alloc %d (prev: %d B)\r\n",size, xPortGetFreeHeapSize());
//  absoluteUsedMemory += size;
//  usedMemory += size;

  LPRINTF("Return for the allocate function w parameter size = %d\r\n", size);

  return pvPortMalloc(size);
}

void __freertos_deallocate(void * pointer, void * state){
  (void) state;
  LPRINTF("-- Free 0x%x (prev: %d B)\r\n", pointer, xPortGetFreeHeapSize());
  if (NULL != pointer)
  {
//          LPRINTF("Pointer is not null.\r\n");
//          usedMemory -= getBlockSize(pointer);
//          LPRINTF("usedMemory var updated: %d\r\n", usedMemory);
          vPortFree(pointer);
  }
  else
  {
          LPERROR("Trying to deallocate a null pointed. Doing nothing.\r\n");
  }
}

void * __freertos_reallocate(void * pointer, size_t size, void * state){
  (void) state;
  LPRINTF("-- Realloc 0x%x -> %d (prev: %d B)\r\n", pointer, size,
    xPortGetFreeHeapSize());
//  absoluteUsedMemory += size;
//  usedMemory += size;
  if (NULL == pointer)
  {
    return __freertos_allocate(size, state);
  }
  else
  {
//    usedMemory -= getBlockSize(pointer);
//
//    return pvPortRealloc(pointer,size);

```

```
        __freertos_deallocate(pointer, state);
        return __freertos_allocate(size, state);

  }
}

void * __freertos_zero_allocate(size_t number_of_elements, size_t size_of_element, void
↪  * state){
  (void) state;
  LPRINTF("-- Calloc %d x %d = %d -> (prev: %d
  ↪  B)\r\n",number_of_elements,size_of_element, number_of_elements*size_of_element,
  ↪  xPortGetFreeHeapSize());
//  absoluteUsedMemory += number_of_elements*size_of_element;
//  usedMemory += number_of_elements*size_of_element;

  return pvPortCalloc(number_of_elements,size_of_element);
}
```

E.2 header file

```
#ifndef _ALLOCATORS_H_
#define _ALLOCATORS_H_

#include "microros.h"

extern int absoluteUsedMemory;
extern int usedMemory;


void * __freertos_allocate(size_t size, void * state);
void __freertos_deallocate(void * pointer, void * state);
void * __freertos_reallocate(void * pointer, size_t size, void * state);
void * __freertos_zero_allocate(size_t number_of_elements,
                                size_t size_of_element, void * state);

#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 system[35] 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.

[35] It 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 documentation[36], 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 Boot firmware 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 page[37]. 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 name[38].

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:

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

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

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

Installing the petalinux tools can be done with the `.run` script[40], as follow:

```
sudo chmod +x ./petalinux-v2023.2-10121855-installer.run
./petalinux-v2023.2-10121855-installer.run
bash
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:

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

[37] https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2022-2.html

[38] As of the time of writing this section, it was recommended[39]to be using the boot firmware in version 2022.2. Thus the software pack for the board will also be using this version.

[39] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1641152513/Kria+SOMs+Starter+Kits#K26-Boot-Firmware-Updates

[40] It 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`.

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

The more general option allows to enable stuff related to hardware configuration[41]. For this step, it is possible to select the openAMP setting the the DTG Settings, as visible in the Figure 30 below:

```
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:

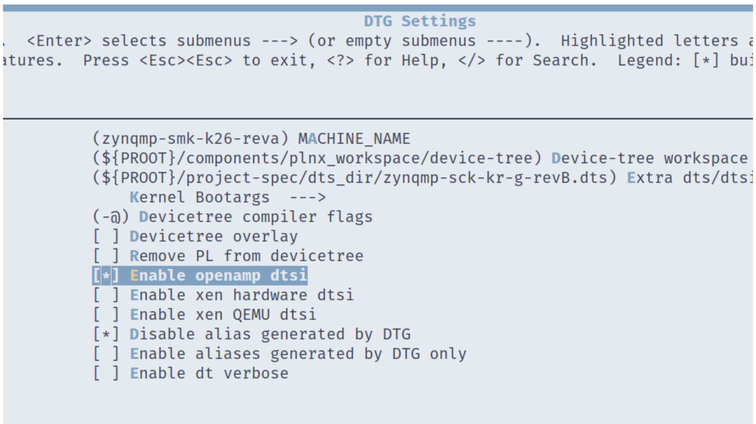

Figure 30: In the hardware config, it is possible to select the openamp device tree source.

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:

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

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

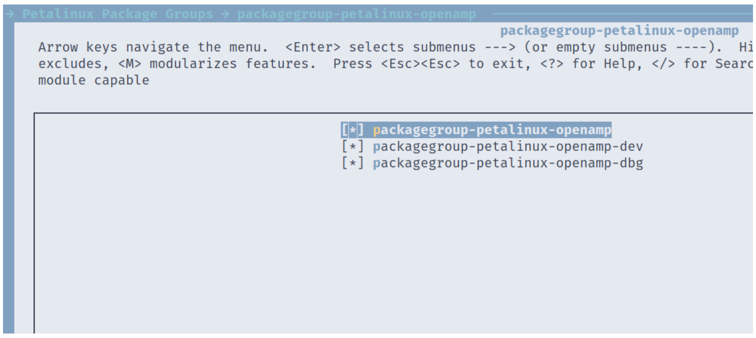

Figure 31: In the rootfs config, it is possible to select the openamp package.

It is then possible to build[42] and package the image. Most critically compared to the example, the `openamp.dtb` overlay must be used in order for OpenAMP and subsequently RPMsg to become usable.

[41] If 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.

[42] Beware, the `petalinux-build` command execution takes a lot of time.

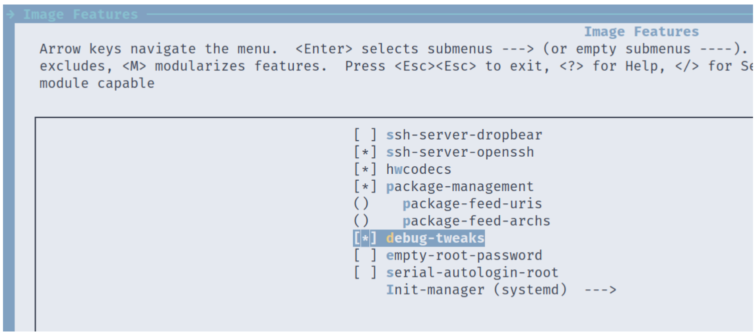

Figure 32: OpenSSH can be configure in the same rootfs config.

```
petalinux-build
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:

```
sudo dd if=./images/linux/petalinux-sdimage.wic \
    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 prompter to enter a login. The default user it `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 documentation[43], 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:

```
sudo -s
echo image_echo_test  > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
echo_test
echo stop > /sys/class/remoteproc/remoteproc0/state
```

[43] https://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

```
sudo dnf install git make gcc binutils packagegroup-core-buildessential
sudo ln -s /usr/bin/aarch64-xilinx-linux-gcc /bin/cc
git clone https://github.com/OpenAMP/openamp-system-reference.git
cd openamp-system-reference/examples/linux/rpmsg-echo-test/
wget https://gitlab.com/sunoc/xilinx-kria-kv260-documentation/-/raw/main/src/echo_test.patch
patch -u -b echo_test.c -i echo_test.patch
make
sudo -s
echo image_echo_test  > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
./echo_test
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.

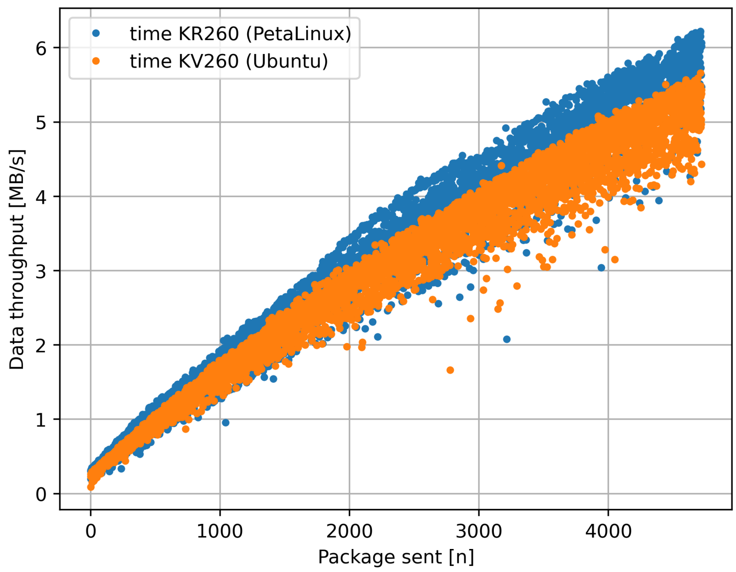

Figure 33: A graph of the data transfer rate for individual packages to be transmitted and received back from the Linux.

This fast delay for data transmission was later confirmed by measurement done using an external logic analyser[44].

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`:

```
# Wired or wireless interfaces
auto eth0
iface eth0 inet static
address 192.168.11.107
netmask 255.255.255.0
network 192.168.11.0
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`:

```
#!/bin/sh -e
ifdown eth0
ifup eth0
exit 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:

```
[Unit]
Description=Restart once more the eth0 network to enable the static IP

[Service]
Type=idle
ExecStart=/bin/sh /home/root/net.sh

[Install]
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:

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

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

[44] 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*. Cham: Springer International Publishing, 2023.