Setting up and using Xilinx KRIA KV260

南山大学

2023-8-30

Vincent Conus - Source available at GitLab 🦊

Contents

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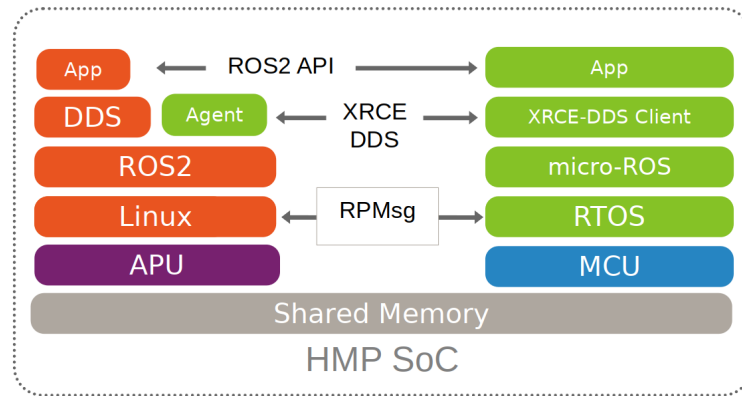
Figure 1: The Linux and ROS2 environment (orange) will communicate with the real-time, FreeRTOS and micro-ROS side (blue) using RPMsg (shared memory).

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 documentation, but I will present it step by step here.

2.1   Getting the new firmware

A 2022 version of the board firmware is required in order to run the latest version of Ubuntu properly.

The image can be downloaded at the atlassian page on the topic, or even directly with the following command:

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

2.2   Reaching the board recovery tool

Now the firmware `.bin` image is available, 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:

- Connect the board to your machine via a Ethernet cable. This will obviously cut you internet access, so you should be set for that.

- Select the wired network as your connection (must be "forced", since it doesn't have internet access).

- Set a fixed IP address for your machine, in the `192.168.0.1/24` range, except the specific `192.168.0.111`, which will be used by the board. The netmark and gateway should also be respectively set to `255.255.255.0` and `192.168.0.1`.

- 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 2 below. If the page struggle to appear,

you should try to un-plug and re-plug the Ethernet cable.

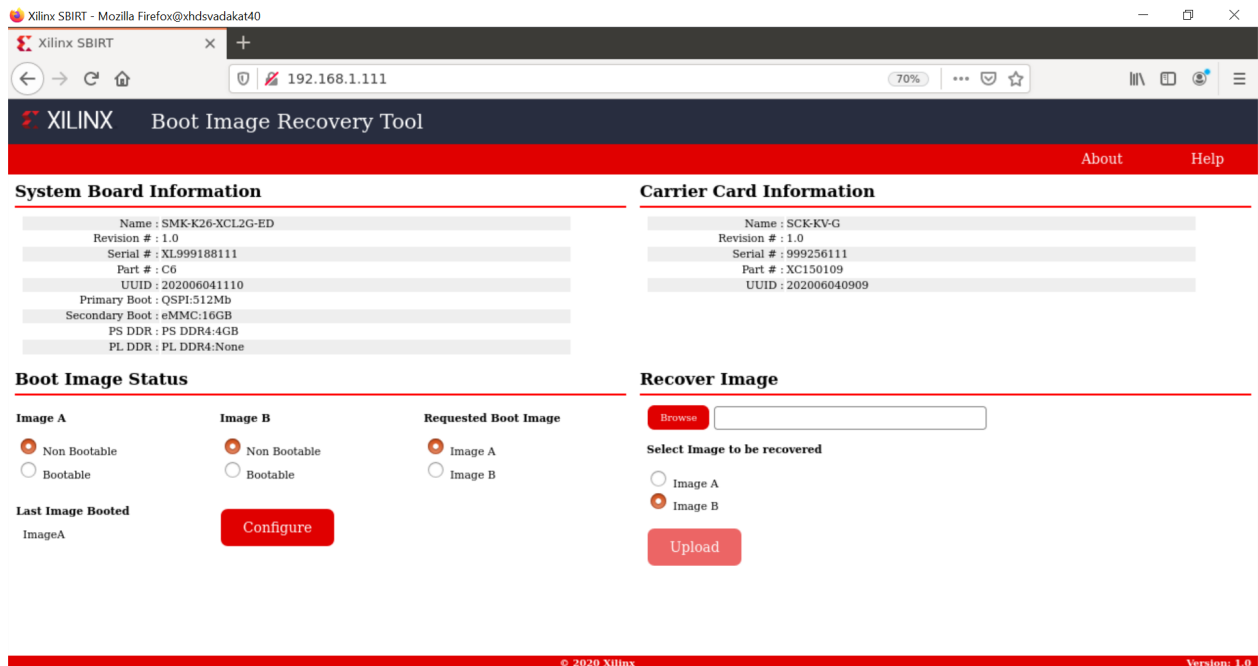Figure 2: The recovery tool for the board, access from Firefox. We can see board information at the center, and the tools to upload the firmware at the bottom of the page.

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.

3   Installing Linux

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[1]. The figure 3 below shows where this operating system sits in the general system we are implementing.
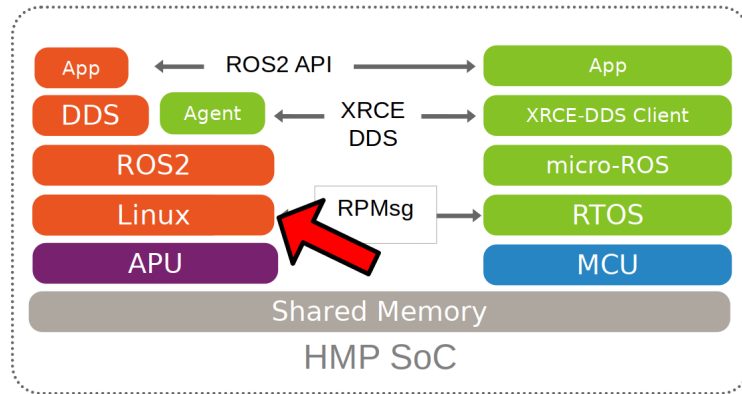

Figure 3: The Linux operating system (red border)runs on the APU (application, general purpose) side of the Kria board CPU. It is the base layer for the ROS2 system.

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 straight-forward. 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` device[2] 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 bitrate is the default value for the board):

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

---

[1]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).

[2]Again, it is critical to be 100\the correct device!

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):

```
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   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.

Firstly, it is possible to set a DNS IP address in `/etc/resolv.conf` by editing it and adding your favorite DNS, for example `nameserver 1.1.1.1` next to the other `nameserver` entry. The resolver can then be restarted.

```
1   sudo nano /etc/resolv.conf
2
3   sudo systemctl restart systemd-resolved
```

Secondly, 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>:\
2          <AXIA_psw>@proxy.ic.nanzan-u.ac.jp:8080"
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\"" \
8       >> ~/.bashrc
```

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

3.2.2   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:

```
1  sudo visudo
2
3  %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:

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

3.2.3   Static IP address

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

The name of the files might vary:

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

You can then set the wanted IP as follow. Note that a custom DNS was also set in that case.

```
1  network:
2    renderer: NetworkManager
3    version: 2
4    ethernets:
5      eth0:
6        addresses:
7          - 192.168.11.103/24
8        routes:
9          - to: default
10           via: 192.168.11.1
11        nameservers:
12          addresses:
13            - 8.8.8.8
14            - 1.1.1.1
```

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

```
1  sudo netplan apply
```

3.2.4   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[3], but the system fan runs sensibly slower without all of this stuff:

---

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

```
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
21
22  systemctl list-units | grep snapd
```

3.2.5 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
```

4 Enabling `remoteproc`

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[4] , [5] 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.

---

[4] A slideshow (JP) from Fixstar employees presents how to use the device tree to enable the communication between the cores.
[5] A blog post (JP) shows all major steps on how to enable the `remoteproc`.

4.1 Device-Tree Overlay patching

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):

```
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/-/\
2       blob/b7300116e153f4b5a1542f8804e4646db8030033/src/system.patch
3
4   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[6]:

```
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!

---

[6]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.

5 Building an example RPMsg real-time firmware

As visible on the official Xilinx documentation about building a demo firmware, 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 4 below shows where the real-time firmware is positioned in the global project.
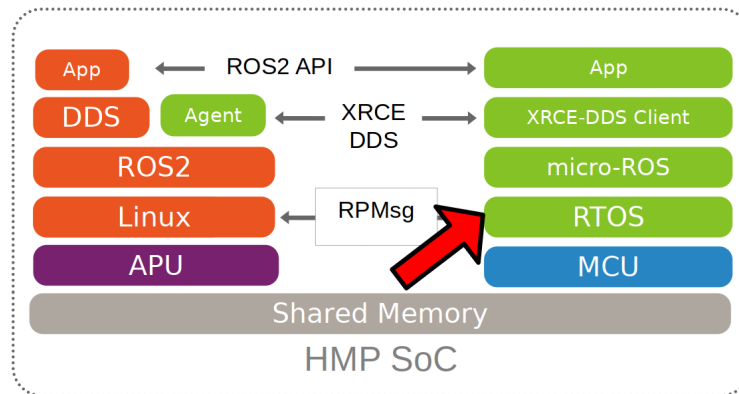

Figure 4: The FreeRTOS firmware and it's application (red border) are running on the real-time capable side of the Kria CPU. A micro-ROS application is shown here, but any real-time firmware will be deployed in the same way.

5.1 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. You will need to get a file named something like `Xilinx_Unified_2022.2_1014_8888_Lin64.bin`[7].

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.

5.1.1 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 \
```

---

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

```
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 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.

5.1.2 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.

A Xilinx dedicated repository 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, 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
```

5.2 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.
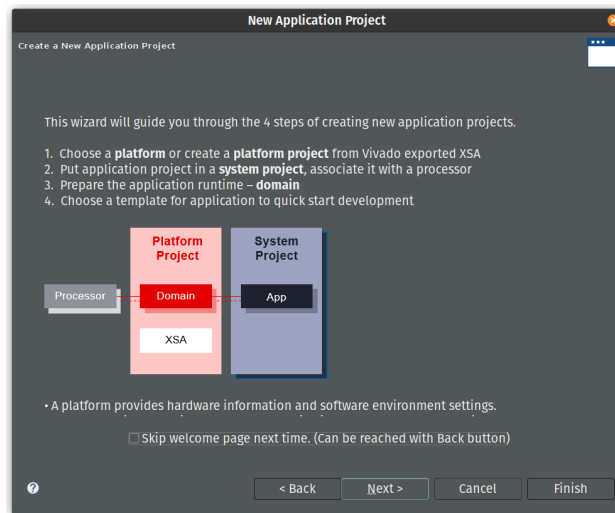
Figure 5: We are starting with creating a "New Application Project" You should be greeted with this wizard window. Next.
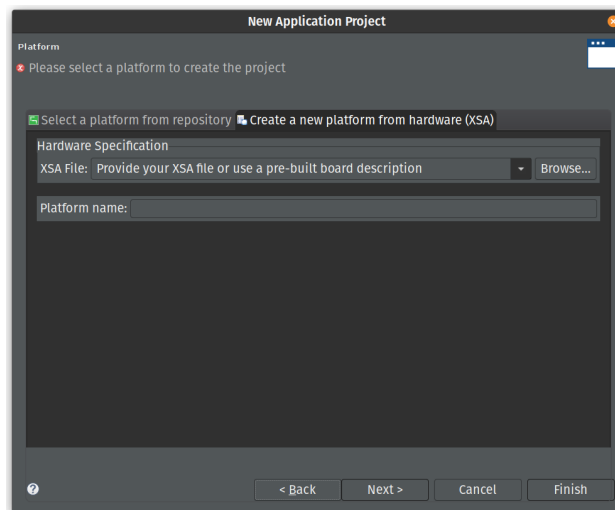
Figure 6: For the platform, we need to get our build Kria configuration. In the "Create a new platform" tab, click the "Browse. . . " button.
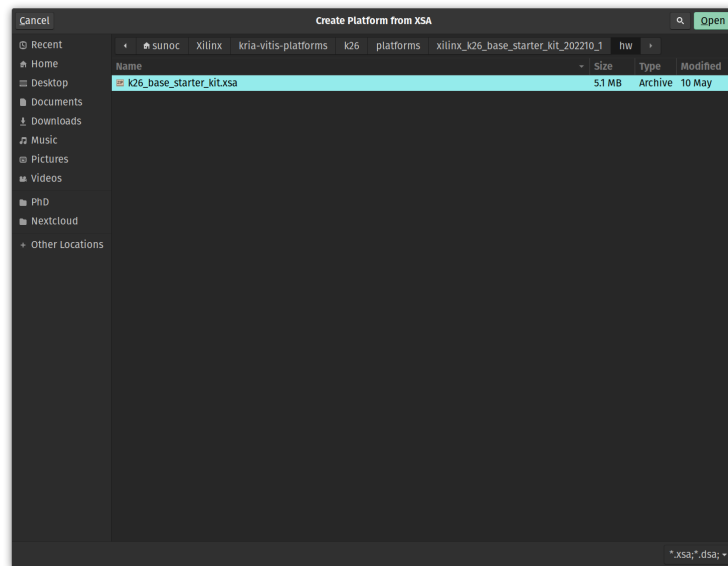
Figure 7: In the file explorer, we should navigate in the "k26" directory, where the configuration file was build. From here we are looking for a ".xsa" file, located in a "hw" directory, as visible.
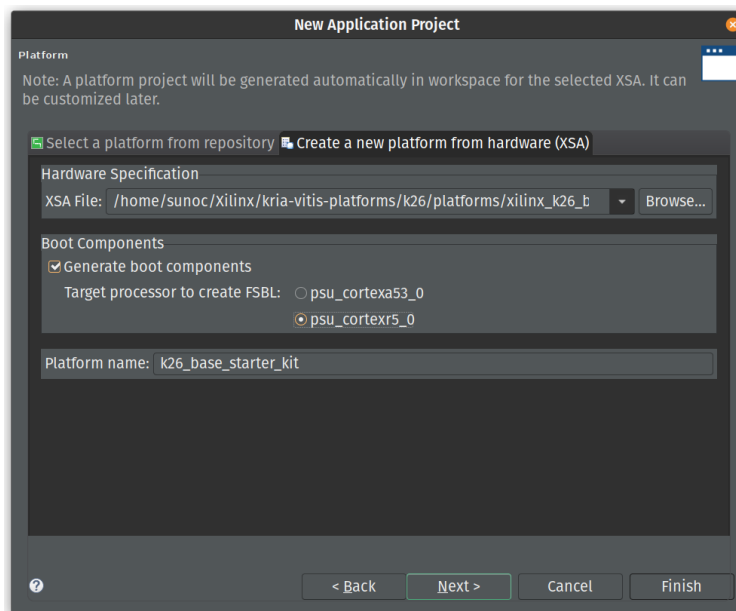
Figure 8: With the configuration file loaded, we can now select a name for our platform, but most importantly, we have to select the "psu Cortex5 0" core as a target. The other, Cortex 53 is the APU running Linux.
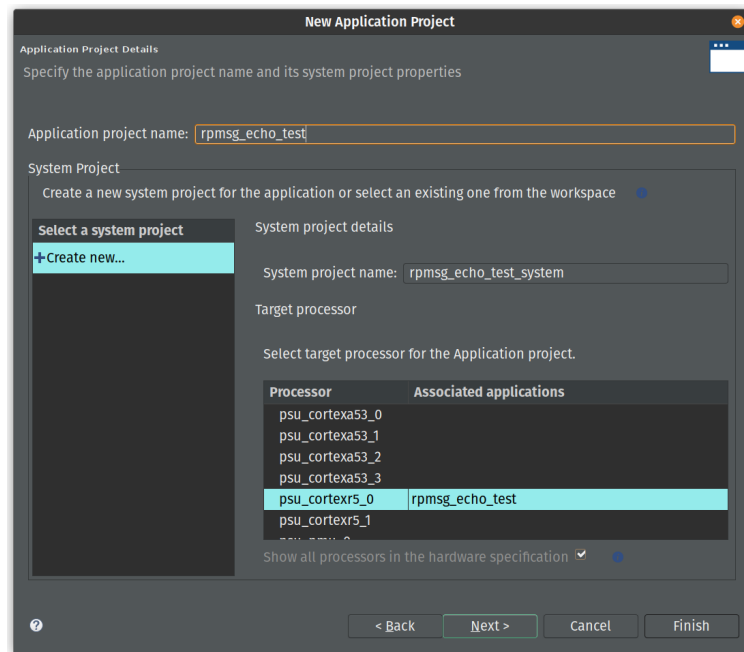
Figure 9: In this next window, we can give a name to our firmware project. It is also critical here to select the core we want to build for. Once again, we want to use the "psu cortex5 0".
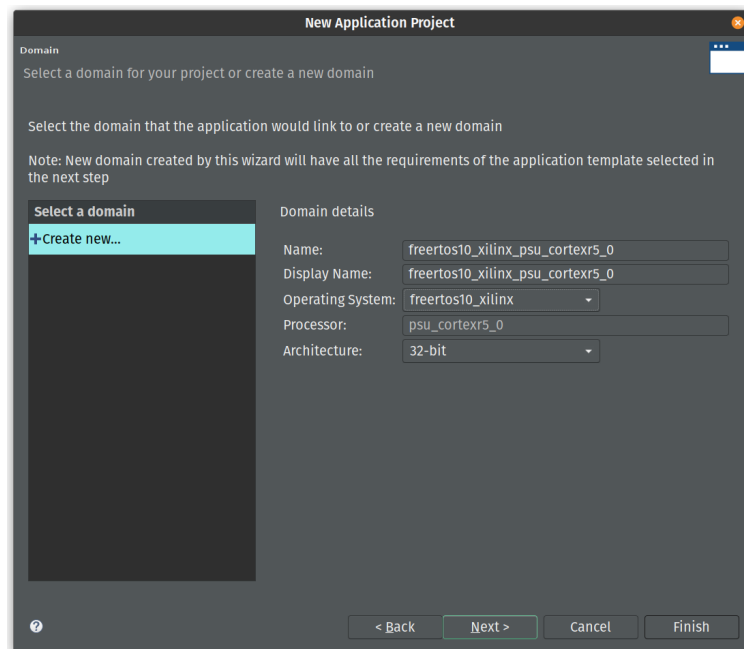
Figure 10: Here, we want to select "freertos10 xilinx" as our Operating System. The rest can remain unchanged.
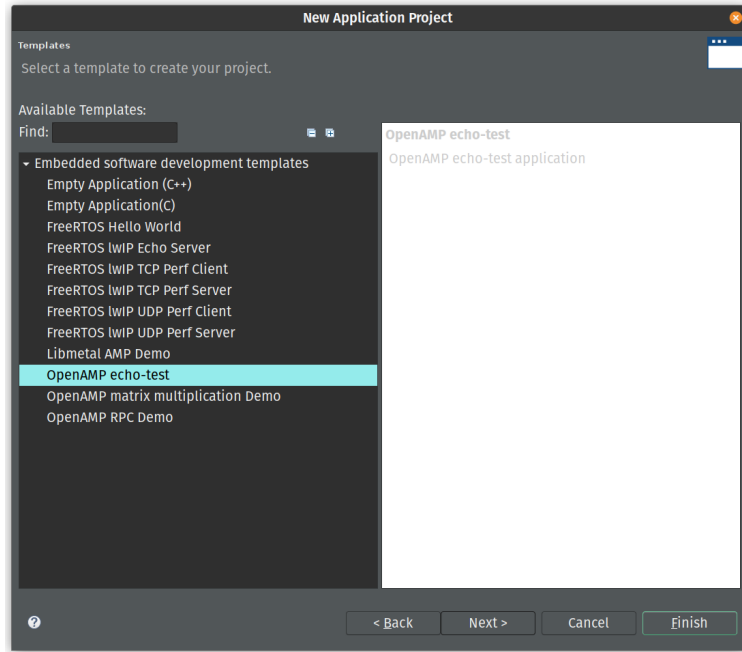
Figure 11: Finally, we can select the demonstration template we are going to use; here we go with "OpenAMP echo-test" since we want to have some simple try of the RPMsg system. Finish.

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 12 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 12: 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 to the 10 section to see how to deploy and use your firmware.

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

5.3   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 13 below:
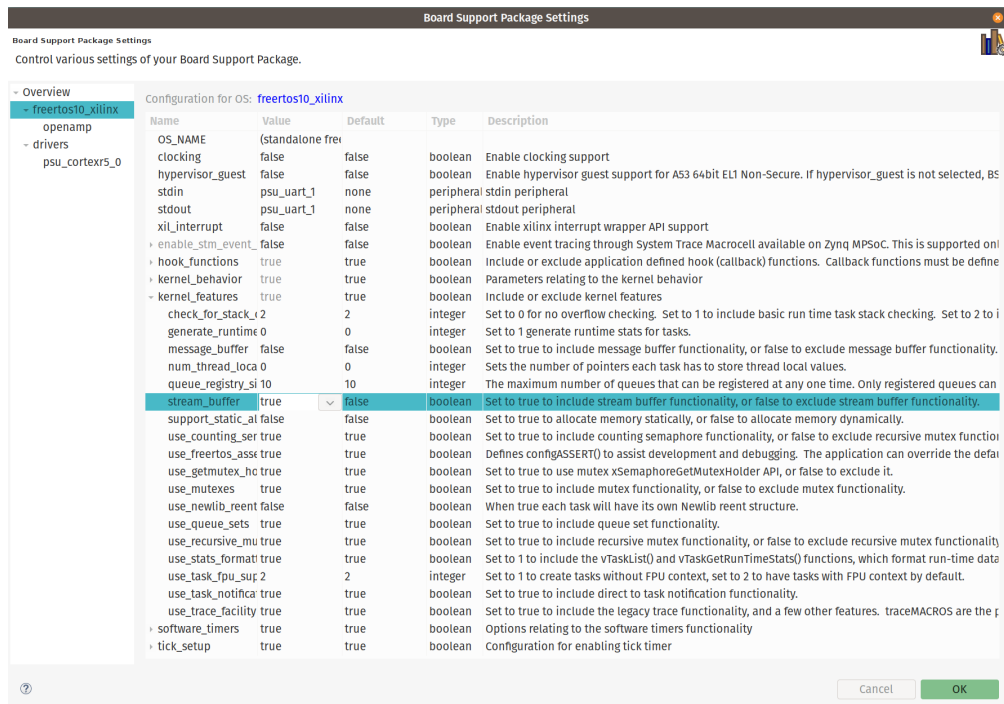
Figure 13: Enabling Stream Buffer in the Vitis IDE setting: this is a setting that can be found in the "platform.spr" element of your project (the platform, not the firmware project itself). From that file, you can access the settings with the button "Modify BSP Settings", and then as visible, in the 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 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/rpmsg-examples/rpmsg-echo-test
5   make
6   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.

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 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 firmwares 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[8] 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/$USER/Downloads
2   wget https://developer.arm.com/-/media/Files/downloads/\
3   gnu/12.2.mpacbti-rel1/binrel/arm-gnu-toolchain-12.2\
4   .mpacbti-rel1-x86_64-arm-none-eabi.tar.xz
5   tar -xvf arm-gnu-toolchain-12.2.mpacbti-rel1-x86_64-\
6   arm-none-eabi.tar.xz
7   popd
8
9   toolchain="/home/$USER/Downloads/arm-gnu-toolchain-\
10  12.2.mpacbti-rel1-x86_64-arm-none-eabi/"
11
12
13  docker run -d --name ros_build -it --net=host \
14  --hostname ros_build \
15  -v /dev:/dev \
16  -v $toolchain:/armr5-toolchain \
17  --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:

---

[8]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.

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

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

```
1   sudo apt update
2   sudo apt-get -y install python3-pip \
3       wget \
4       nano
5
6   . /opt/ros/\$ROS_DISTRO/setup.bash
7
8   mkdir microros_ws
9   cd microros_ws
10  git clone -b \$ROS_DISTRO \
11      https://github.com/micro-ROS/micro_ros_setup.git \
12      src/micro_ros_setup
13
14  sudo rosdep fix-permissions &&\
15      rosdep update &&\
16      rosdep install --from-paths src --ignore-src -y
17
18  colcon build
19  . ./install/local_setup.bash
20
21  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 files[9] 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-\
2   documentation/-/raw/main/src/custom_r5f_toolchain.cmake
3
4   wget https://gitlab.com/sunoc/xilinx-kria-kv260-\
5   documentation/-/raw/main/src/custom_r5f_colcon.meta
6
7   cp -r /armr5-toolchain/ \$(pwd)/firmware/toolchain && \
8   export PATH=\$PATH:\$(pwd)/firmware/toolchain/bin
9
10  ros2 run micro_ros_setup build_firmware.sh \
11  \$(pwd)/custom_r5f_toolchain.cmake \$(pwd)/\
12  custom_r5f_colcon.meta
```

---

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

18

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 you 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 toolchain, but the figures 14 and 15 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.
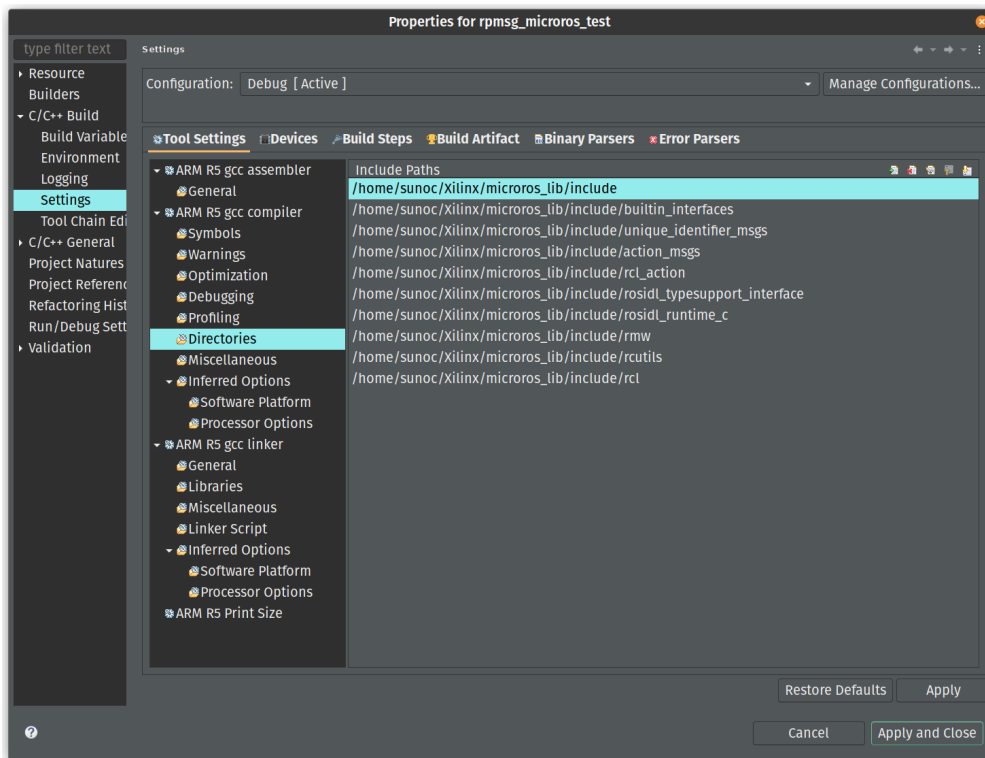
Figure 14: Firstly, in the "C/C++ Build" settings of your firmware project, under the "Settings" menu, you should find the gcc compiler "Directories". In here you should add the "include" directory of your library. Be careful however, if your include files are in a second layer of directory (as it is the case for libmicroros) you will need to include each sub-directory individually, as visible in this figure.

19
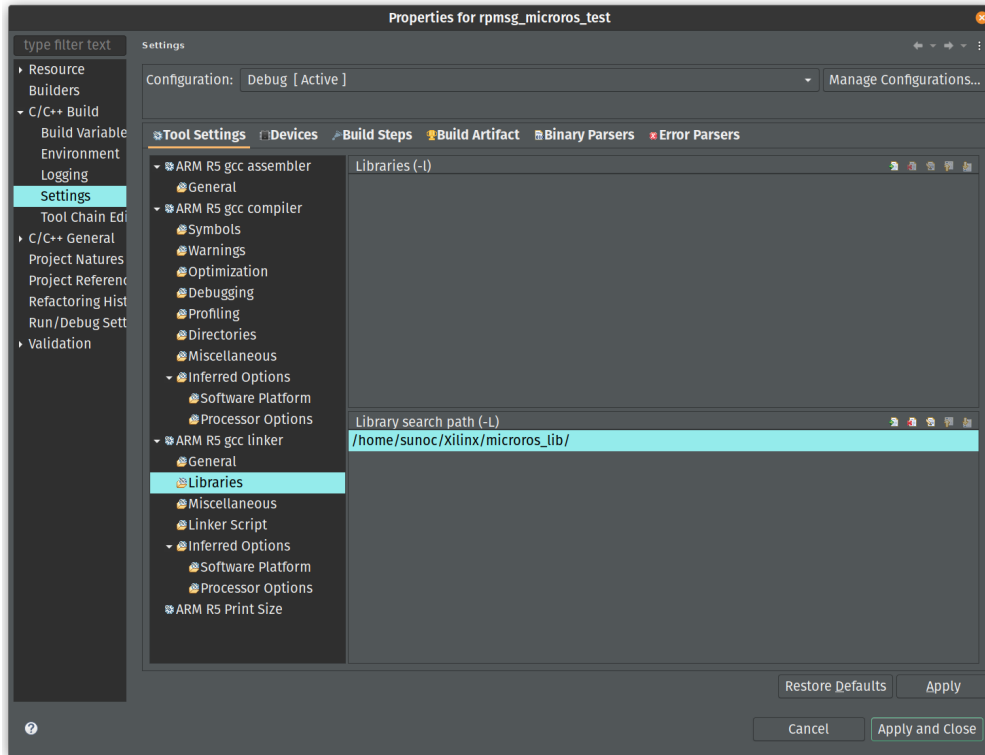
Figure 15: Secondly, in the gcc linkers "Libraries", you can add the top level directory of your library. In our case, it is the directory that contains both the "include" directory added earlier, and also the "libmicroros.a" file.

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 command[10]:

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

---

[10]Note that in that case, we are retrieving the binary for a project named `rpmsg_pingpong_microros_lib`. You 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 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 16 below.

As this part of the firmware was developed and tested on it's own, the overall implementation within the "real" ping-pong application is still a work in progress, and further modifications are expected when the system will be more thoroughly tested.

A version of this firmware is available at my gitlab repository[11]. This firmware was built using Xilinx's IDE, which setup was presented in the section 5.1, 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 16, 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:

---

[11]micro-ROS ping pong test firmware repository: `https://gitlab.com/sunoc/libmicroros_kv260`
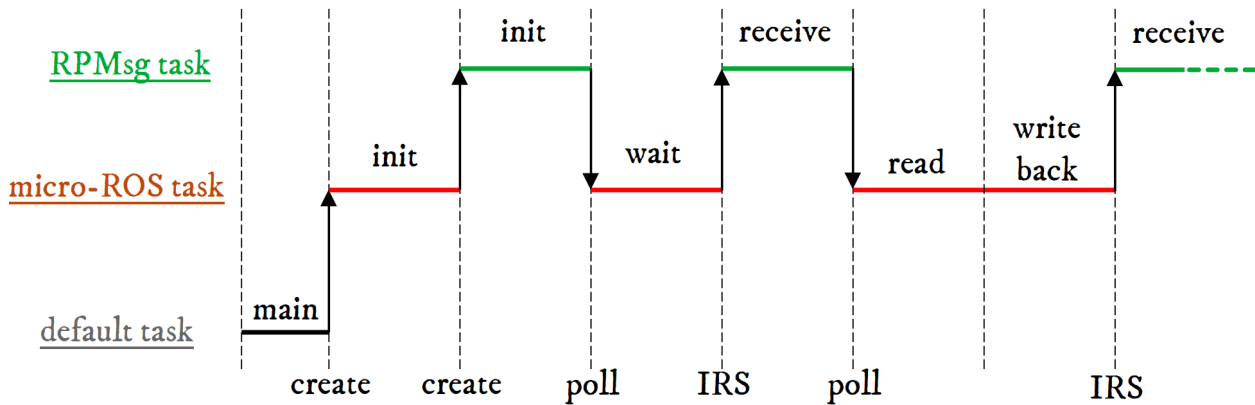
Figure 16: Two tasks are being run concurrently in order to manage the communication situation, with binary semaphore-based lock-unlock system.

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.

- – Initialize the support system
- – 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.

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-space[12], to test a basic RPMsg setup[13]:

```
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.

---

[12]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`.

[13]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 6.

11 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 17 below shows what part of the overall system we are talking about here.
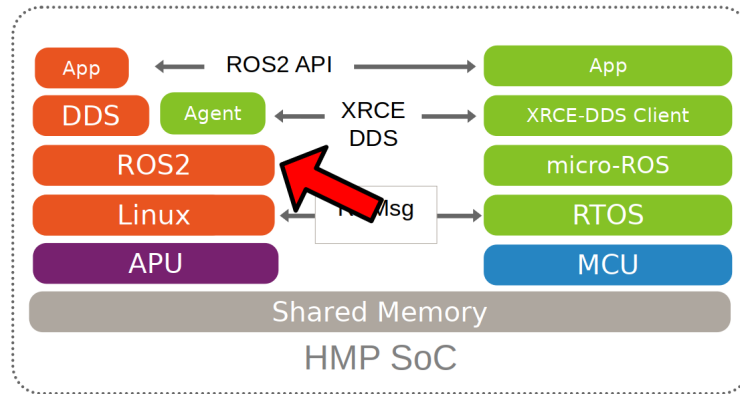
Figure 17: The ROS2 middle (red border) runs on top of the Linux, on the general-purpose core of the Kria board.

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

Since an Ubuntu distribution is installed on the board, the installation of ROS2 can be done[14] 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 here[15].

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:

---

[14]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.

[15]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.

```
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 ROS2[16] as follow:

```
1    sudo apt update
2    sudo apt upgrade -y
3    sudo apt install -y ros-humble-desktop \
4        ros-humble-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/humble/setup.bash
2    ros2 run demo_nodes_cpp talker
```

And then:

```
1    source /opt/ros/humble/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 then.

11.2   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.

11.2.1   Installing Docker on Ubuntu

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 18 below, after the commands:

```
1    sudo apt-get update
2    sudo apt-get install ca-certificates curl sudo
3    gnupg 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
```

---

[16]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   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 18: The return of a successful run of the `hello world` test Docker container.

11.2.2 Running a ROS2 container

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[17]. 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
```

---

[17]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.

From there, it becomes possible to simply use ROS2 as you would for a bare-metal install, and as presented in the section 11.1 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[18]:

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

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

---

[18]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.

12 micro-ROS 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 19 below. In particular, it will be useful to modify this agent in order to archive the full RPMsg communication for ROS2[19]. 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.
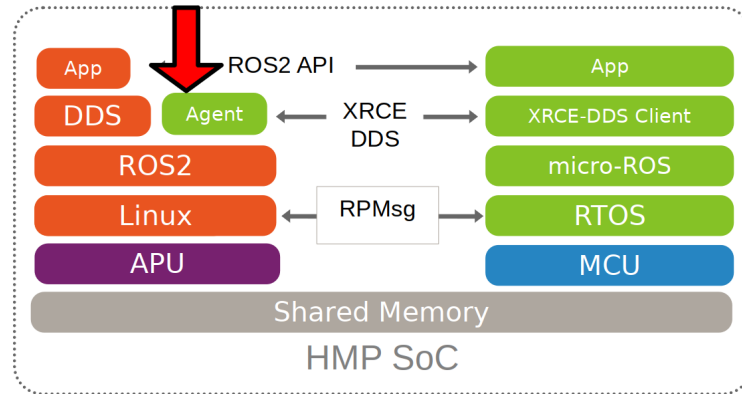
Figure 19: The agent (red border) allows for a micro-ROS instance to communicate with a ROS2 system. It is deployed on the Linux side, as a ROS2 node.

12.1 Building a eProsima bare-metal example agent

A key aspect to understand about modifying the agent (as this will be needed later on to support our new communication system), is that the default system and instruction provided by micro-ROS does not allow such modification[20].

In order to avoid that, one can get and run a ROS2 node design by eProsima, that can be deployed on it's own (without other ROS2 application and nodes) and eventually modified.

```
1    sudo nano /microros_ws/build/micro_ros_agent/agent/src/xrceagent/src/cpp/transport/custom/CustomAgent.cpp
```

The same file is available online as a reference, on eProsima XRCE-DDS-Agent GitHub repository.

```
1    git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
2    cd Micro-XRCE-DDS-Agent/
3
4    rm -rf examples/
5    git clone https://gitlab.com/sunoc/rpmsg-micro-ros-agent.git ./examples/custom_agent
6
7    mkdir build && cd build
8    cmake -DUAGENT_BUILD_USAGE_EXAMPLES=ON ..
9    make
```

The custom agent is then available to run from the ./src/examples/custom_agent/CustomXRCEAgent.

All the diff when the custom agent system was added: https://github.com/eProsima/Micro-XRCE-DDS-Agent/pull/205/files

---

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

[20]This information was eventually found on a discussion in a GitHub Issue thread.

For rebuilding the custom agent with a simple `make clean && make all` will conveniently only rebuild the part that was modify. Which means that once all the libraries have been compiled once, the subsequent re-compilation of the agent itself can be done quickly !

Major details on how to implement a custom agent specified here: `https://github.com/eProsima/Micro-XRCE-DDS-Agent/issues/195#issuecomment-721002153`

12.2 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 \
2       /dev:/dev --privileged ros:iron
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
```

12.3 Building the Agent version with a modified transport

The first working implementation of the DDS over RPMsg happend 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:

```
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
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 follow[21],[22]:

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

12.4 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  cd /Micro-XRCE-DDS-Agent/build
2  make clean && make -j4
3  ./MicroXRCEAgent serial --dev /dev/null --verbose 6
```

---

[21]The `--dev /dev/null` option is only a placeholder as the serial requires it but not RPMsg.
[22]The `--verbose 6` allows to see the maximum amount of information about the data being transmitted.

13 Running the ping-pong node WORK_IN_PROGRESS

A custom ping-pong node for ROS2 was developped 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 12.

- 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 dependancies issues, as presented in the section 11.

14 Conclusion & future WORK_IN_PROGRESS

.1 TODO Update the figures of the report

I should use the one used in the SASIMI2024 publication!

A DTO patch

This file is available in this repository: system.patch

```
diff -u --label /ssh\:kria\:/home/ubuntu/system_original.dts --label
   /ssh\:kria\:/home/ubuntu/system.dts /tmp/tramp.KJbEbz.dts /tmp/tramp.zHBG7v.dts
--- /ssh:kria:/home/ubuntu/system_original.dts
+++ /ssh:kria:/home/ubuntu/system.dts
@@ -138,7 +138,7 @@

        firmware {

-               zynqmp-firmware {
+               zynqmp_firmware: zynqmp-firmware {
                        compatible = "xlnx,zynqmp-firmware";
                        #power-domain-cells = <0x01>;
                        method = "smc";
@@ -719,7 +719,7 @@
                        phandle = <0x44>;
                };

-               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>;
@@ -1536,7 +1536,7 @@
                        pinctrl-names = "default";
                        u-boot,dm-pre-reloc;
                        compatible = "xlnx,zynqmp-uart\0cdns,uart-r1p12";
-                       status = "okay";
+                       status = "disabled";
                        interrupt-parent = <0x04>;
                        interrupts = <0x00 0x16 0x04>;
                        reg = <0x00 0xff010000 0x00 0x1000>;
@@ -1909,6 +1909,84 @@
                        pwms = <0x1b 0x02 0x9c40 0x00>;
                };

+       reserved-memory {
+               #address-cells = <2>;
+               #size-cells = <2>;
+               ranges;
+               rpu0vdev0vring0: rpu0vdev0vring0@3ed40000 {
+                       no-map;
+                       reg = <0x0 0x3ed40000 0x0 0x4000>;
+               };
+               rpu0vdev0vring1: rpu0vdev0vring1@3ed44000 {
+                       no-map;
```

31

```
45  +                    reg = <0x0 0x3ed44000 0x0 0x4000>;
46  +                };
47  +                rpu0vdev0buffer: rpu0vdev0buffer@3ed48000 {
48  +                    no-map;
49  +                    reg = <0x0 0x3ed48000 0x0 0x100000>;
50  +                };
51  +                rproc_0_reserved: rproc_0_reserved@3ec00000 {
52  +                    no-map;
53  +                    reg = <0x0 0x3ec00000 0x0 0x140000>;
54  +                };
55  +            };
56  +            tcm_0a: tcm_0a@ffe00000 {
57  +                no-map;
58  +                reg = <0x0 0xffe00000 0x0 0x15000>;
59  +                status = "okay";
60  +                compatible = "mmio-sram";
61  +                power-domain = <&zynqmp_firmware 15>;
62  +            };
63  +            tcm_0b: tcm_0b@ffe20000 {
64  +                no-map;
65  +                reg = <0x0 0xffe20000 0x0 0x15000>;
66  +                status = "okay";
67  +                compatible = "mmio-sram";
68  +                power-domain = <&zynqmp_firmware 16>;
69  +            };
70  +            rf5ss@ff9a0000 {
71  +                compatible = "xlnx,zynqmp-r5-remoteproc";
72  +                xlnx,cluster-mode = <1>;
73  +                ranges;
74  +                reg = <0x0 0xFF9A0000 0x0 0x15000>;
75  +                #address-cells = <0x2>;
76  +                #size-cells = <0x2>;
77  +                r5f_0 {
78  +                    compatible = "xilinx,r5f";
79  +                    #address-cells = <2>;
80  +                    #size-cells = <2>;
81  +                    ranges;
82  +                    sram = <&tcm_0a &tcm_0b>;
83  +                    memory-region = <&rproc_0_reserved>, <&rpu0vdev0buffer>,
    ↪ <&rpu0vdev0vring0>, <&rpu0vdev0vring1>;
84  +                    power-domain = <&zynqmp_firmware 7>;
85  +                    mboxes = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;
86  +                    mbox-names = "tx", "rx";
87  +                };
88  +            };
89  +            zynqmp_ipi1 {
90  +                compatible = "xlnx,zynqmp-ipi-mailbox";
91  +                interrupt-parent = <&gic>;
92  +                interrupts = <0 29 4>;
93  +                xlnx,ipi-id = <7>;
94  +                #address-cells = <1>;
95  +                #size-cells = <1>;
96  +                ranges;
97  +                /* APU<->RPU0 IPI mailbox controller */
```

```
98    +              ipi_mailbox_rpu0: mailbox@ff990600 {
99    +                  reg = <0xff990600 0x20>,
100   +                        <0xff990620 0x20>,
101   +                        <0xff9900c0 0x20>,
102   +                        <0xff9900e0 0x20>;
103   +                  reg-names = "local_request_region",
104   +                        "local_response_region",
105   +                        "remote_request_region",
106   +                        "remote_response_region";
107   +                  #mbox-cells = <1>;
108   +                  xlnx,ipi-id = <1>;
109   +              };
110   +          };
111   +
112   +
113          __symbols__ {
114              cpu0 = "/cpus/cpu@0";
115              cpu1 = "/cpus/cpu@1";
116
117   Diff finished.  Wed May 24 10:15:49 2023
```

B Custom toolchain 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  set(CMAKE_INSTALL_LIBDIR /usr/)
5  set(PLATFORM_NAME "LwIP")
6
7
8  set(ARCH_CPU_FLAGS "-mcpu=cortex-r5 -mthumb -mfpu=vfpv3-d16 -mfloat-abi=hard -DARMR5 -O0
   ↪  -Wall -fdata-sections -ffunction-sections -fno-tree-loop-distribute-patterns
   ↪  -Wno-unused-parameter -Wno-unused-value -Wno-unused-variable -Wno-unused-function
   ↪  -Wno-unused-but-set-variable" CACHE STRING "" FORCE)
9  set(ARCH_OPT_FLAGS "")
10
11 set(CMAKE_C_COMPILER arm-none-eabi-gcc)
12 set(CMAKE_CXX_COMPILER arm-none-eabi-g++)
13
14 set(CMAKE_C_FLAGS_INIT "-std=c11 ${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS}
   ↪  -DCLOCK_MONOTONIC=0" CACHE STRING "" FORCE)
15 set(CMAKE_CXX_FLAGS_INIT "-std=c++14 ${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS}
   ↪  -DCLOCK_MONOTONIC=0" CACHE STRING "" FORCE)
16
17
18
19 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_PIC=OFF",
33                  "-DUCLIENT_PROFILE_UDP=OFF",
34                  "-DUCLIENT_PROFILE_TCP=OFF",
35                  "-DUCLIENT_PROFILE_DISCOVERY=OFF",
36                  "-DUCLIENT_PROFILE_SERIAL=OFF",
37                  "-UCLIENT_PROFILE_STREAM_FRAMING=ON",
38                  "-DUCLIENT_PROFILE_CUSTOM_TRANSPORT=ON"
39              ]
40          },
41          "rmw_microxrcedds": {
42              "cmake-args": [
43                  "-DRMW_UXRCE_MAX_NODES=1",
44                  "-DRMW_UXRCE_MAX_PUBLISHERS=5",
45                  "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=5",
46                  "-DRMW_UXRCE_MAX_SERVICES=1",
47                  "-DRMW_UXRCE_MAX_CLIENTS=1",
48                  "-DRMW_UXRCE_MAX_HISTORY=4",
49                  "-DRMW_UXRCE_TRANSPORT=custom"
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

```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. */
```

```
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
↪         ) * 8 );
54
55        /* Add the current tick count. */
56        ullTickCount += xCurrentTime.xTimeOnEntering;
57
58        /* Convert ullTickCount to timespec. */
59        UTILS_NanosecondsToTimespec( ( int64_t ) ullTickCount * NANOSECONDS_PER_TICK, tp );
60
61        return 0;
62    }
```

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

```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);
```

```
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
   ↪  * state){
55    (void) state;
56    LPRINTF("-- Calloc %d x %d = %d -> (prev: %d
   ↪  B)\r\n",number_of_elements,size_of_element, number_of_elements*size_of_element,
   ↪  xPortGetFreeHeapSize());
57 //  absoluteUsedMemory += number_of_elements*size_of_element;
58 //  usedMemory += number_of_elements*size_of_element;
59
60    return pvPortCalloc(number_of_elements,size_of_element);
61 }
```

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_
```