Open Amp Guide: RPU usage on Zynq Ultrascale+

Real-Time Industrial Systems

Marcello Cinque, Daniele Ottaviano

Roadmap

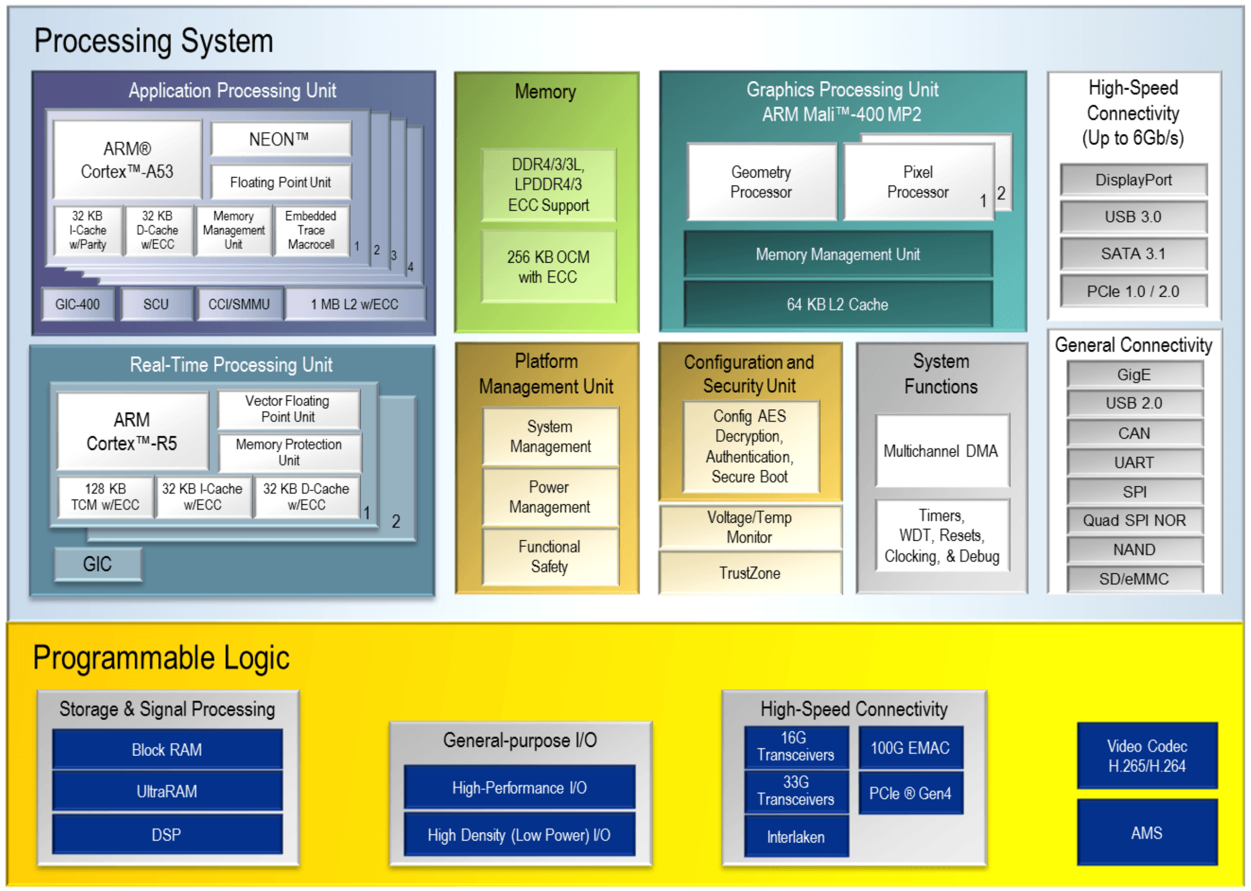
- Introduction
- Environment setup: Petalinux and Vitis/Vivado
- OpenAMP Pre-built application
  - Test on QEMU
- OpenAMP Kernel Space vs User Space
- Building an OpenAMP application (Kernel Space)
  - Building an RPU Firmware
  - Building a Linux Kernel Space Application
  - Configure and Build with Petalinux
  - Test on Board

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

Introduction

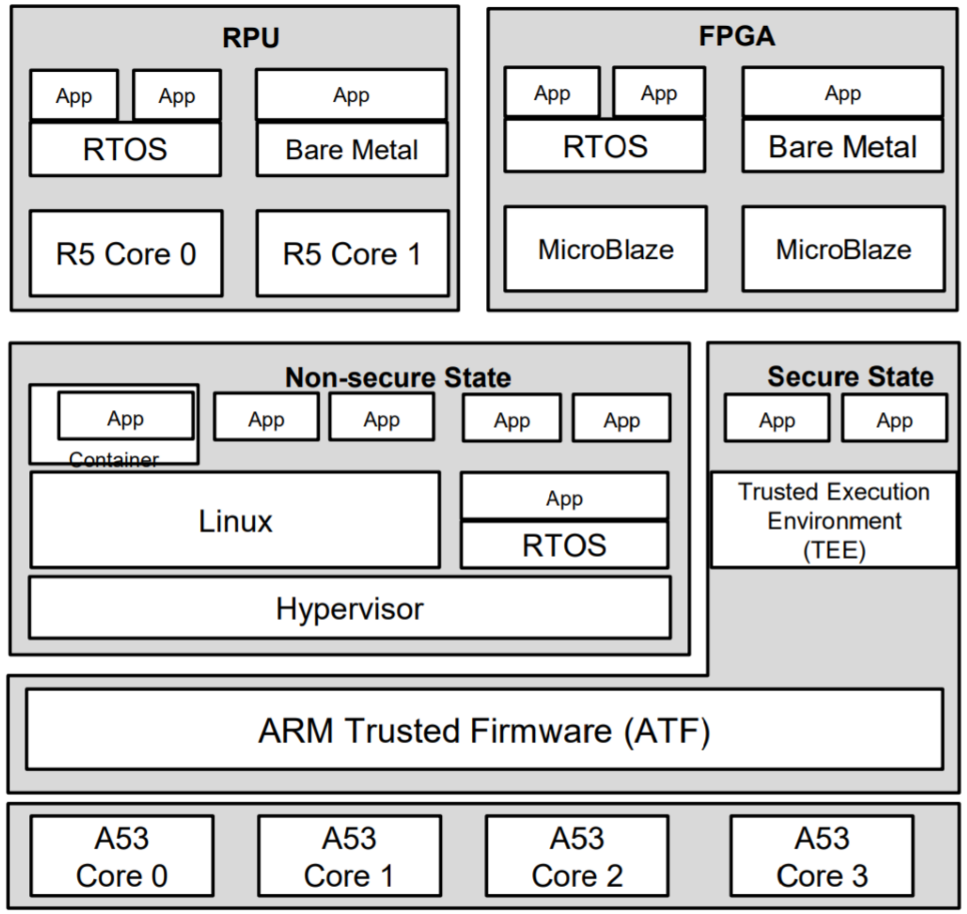
Zynq® UltraScale+™ MPSoC

- Recently, Traditional Programmable Logic Controllers (PLCs) used in industrial environments are inadequate to meet market demand
- Emerging *Multiprocessor System-on-Chip* (MPSoC) platforms, featuring **asymmetric multi-processing** (AMP) guarantee:
  - ✓Flexibility
  - ✓Real-Time features
  - ✓Power Management



Operating Environments on MPSoCs

- The **asymmetrical** architecture of the MPSoCs allows various operating environments running side-by-side on the same chip:
  - APU:
    - ARM TrustZone
    - Hypervisor
    - O.S. (Linux)
    - Bare-Metal
  - RPU:
    - RTOS
    - Bare-Metal
  - FPGA:
    - RTOS
    - Bare-Metal

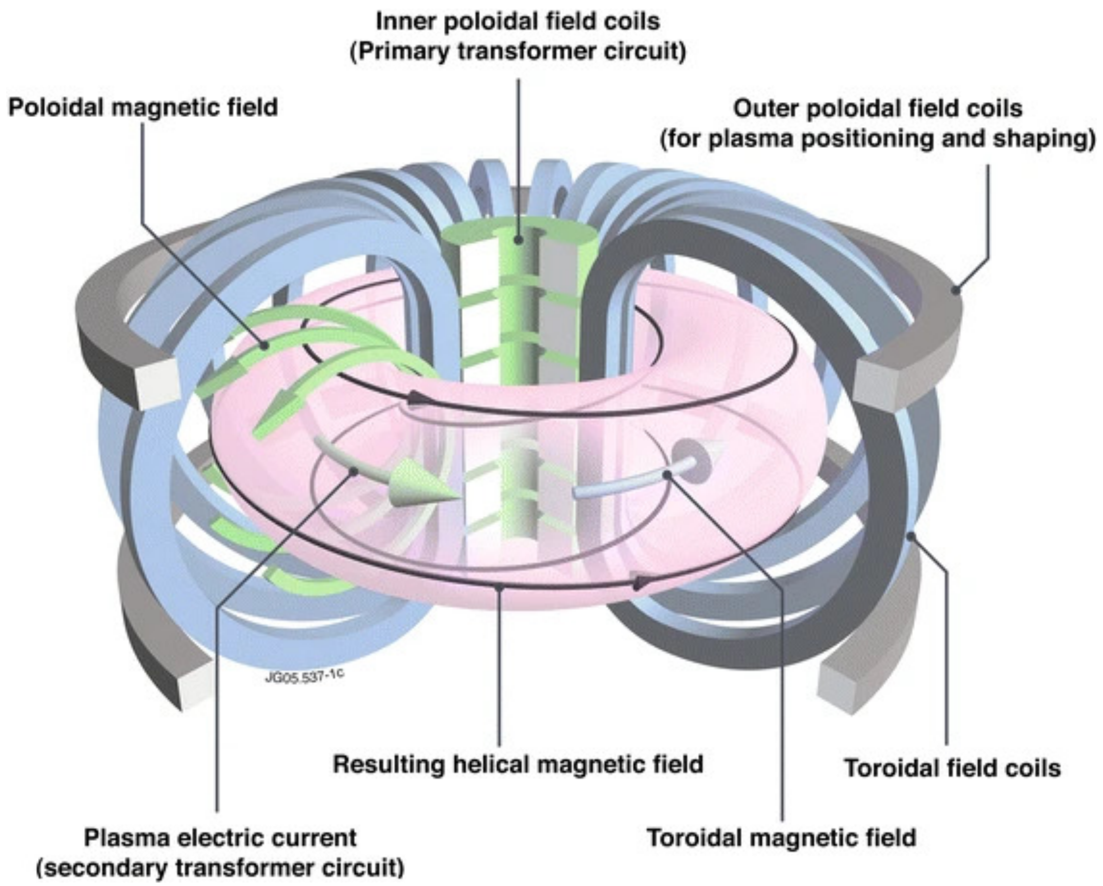


OpenAMP

- OpenAMP (Open Asymmetric Multi-Processing) seeks to **standardize** the interactions between operating environments in a heterogeneous embedded system through open-source solutions for Asymmetric MultiProcessing (AMP).
- OpenAMP is a framework providing software components able to:
  - Provides **Life Cycle Management** and **Inter Processor Communication** capabilities for management of remote compute resources and their associated software contexts.
  - Provides a **standalone library** usable with RTOS and baremetal software environments.
  - Compatibility with upstream Linux remoteproc, rpmsg and VirtIO components.

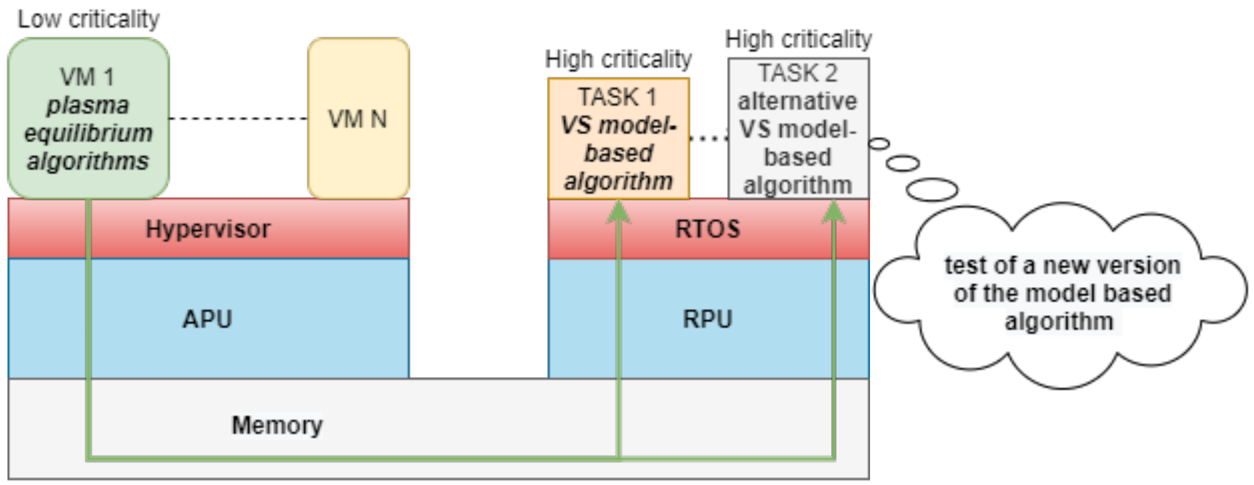
Example: ITER

- **ITER** is an international project aimed at proving the feasibility of energy production by means of nuclear fusion on Earth
- Some requirements must be considered during the design of the **real-time** infrastructure
- These requirement calls for **mixed-criticality systems (MCS)** to:
  - implement integrated control systems with different levels of criticality


Inner poloidal field coils
(Primary transformer circuit)
Poloidal magnetic field
Outer poloidal field coils
(for plasma positioning and shaping)
JG05.537-1c
Resulting helical magnetic field
Toroidal field coils
Plasma electric current
(secondary transformer circuit)
Toroidal magnetic field

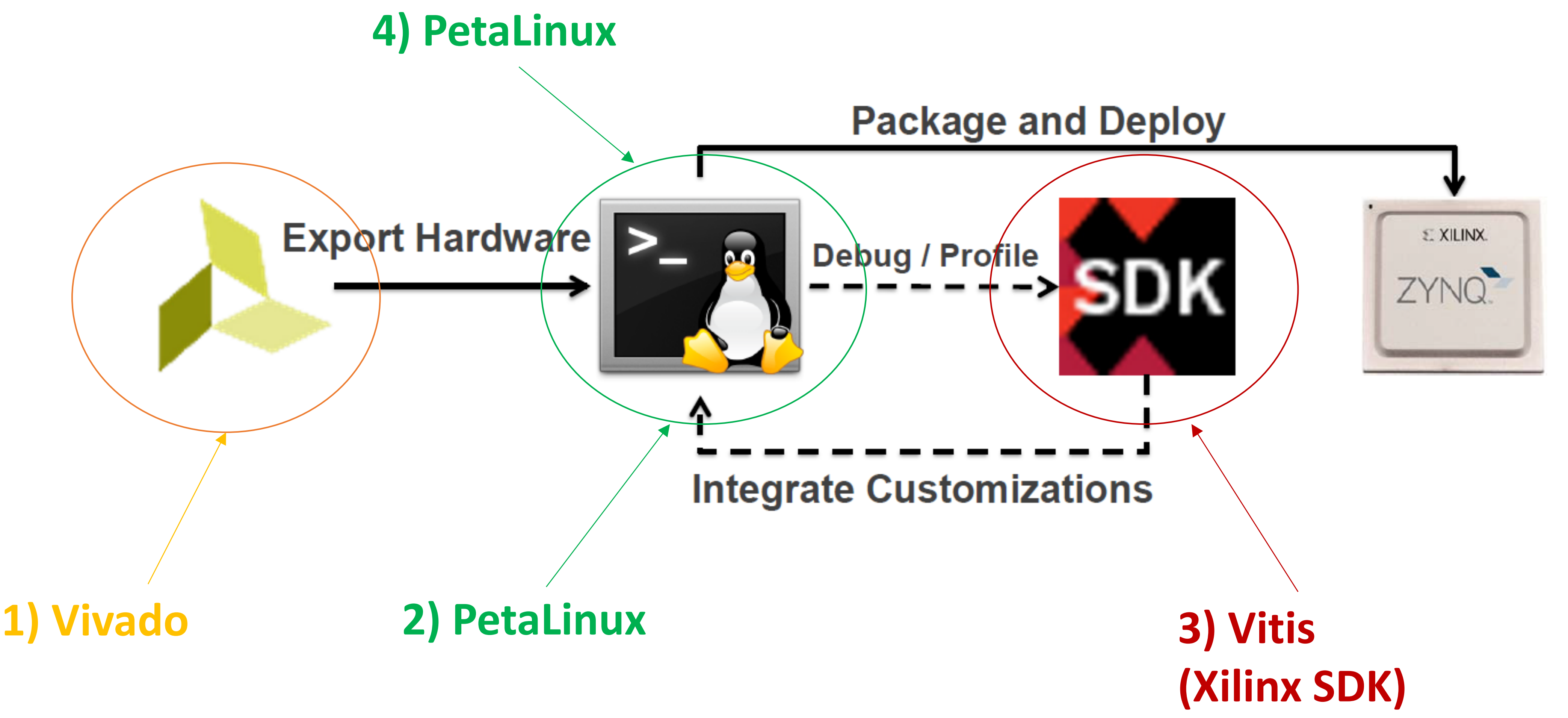
Example: ITER Vertical Stabilization

- **Vertical Stabilization** (VS) is an example of advanced control system that may exploit a **mixed-criticality real-time infrastructure**
- **Model-based** control approaches have been proposed in literature to solve the VS problem.
  - **Performance can be improved by calculating the plasma equilibrium reconstruction**
  - **Tasks can be duplicated to provide fault-tolerance**


Low criticality
VM 1 *plasma equilibrium algorithms*
VM N
Hypervisor
APU
Memory
High criticality
TASK 1 *VS model-based algorithm*
High criticality
TASK 2 alternative VS model-based algorithm
RTOS
RPU
test of a new version of the model based algorithm

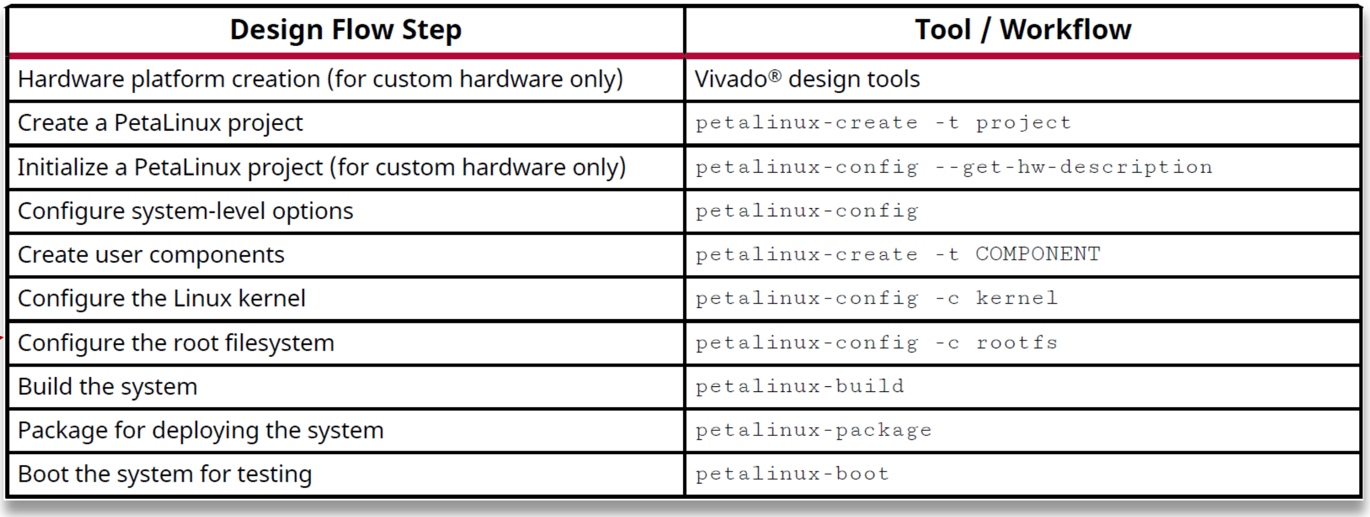
Environment Setup

Environment setup



Environment setup: Petalinux

- **Petalinux** is a Software Development Kit (**SDK**) for System on Chip FPGA-Based provided by Xilinx to automate and simplify:
  - the development of a **Linux-based embedded environment** for a specific board
  - the loading of the software on the board.
- Usually, Petalinux uses a sequential workflow model as shown in the table:



Design Flow Step	Tool / Workflow
Hardware platform creation (for custom hardware only)	Vivado® design tools
Create a PetaLinux project	`petalinux-create -t project`
Initialize a PetaLinux project (for custom hardware only)	`petalinux-config --get-hw-description`
Configure system-level options	`petalinux-config`
Create user components	`petalinux-create -t COMPONENT`
Configure the Linux kernel	`petalinux-config -c kernel`
Configure the root filesystem	`petalinux-config -c rootfs`
Build the system	`petalinux-build`
Package for deploying the system	`petalinux-package`
Boot the system for testing	`petalinux-boot`

OSS: It is possible to add some custom user programs in the root filesystem!
(VITIS/ Xilinx SDK)

Environment setup: Petalinux

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

- OSS: Choose the versions carefully!

Environment setup: Petalinux

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

Environment setup: Petalinux

- As with many software development tools, there are a variety of **dependencies** that you will need in order for PetaLinux to operate.
  - ➢ ***sudo apt-get -y install iproute2 gcc g++ net-tools libncurses5-dev zlib1g:i386 libssl-dev flex bison libselinux1 xterm autoconf libtool texinfo zlib1g-dev gcc-multilib build-essential screen pax gawk python3 python3-pexpect python3-pip python3-git python3-jinja2 xz-utils debianutils iputils-ping libegl1-mesa libsdl1.2-dev pylint3 cpio libtinfo5***
- Download Petalinux from this link:
  - https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html

Environment setup: Petalinux

- Once the download has been completed, make a directory in which you would like the PetaLinux tools to be installed in and run the installer :
  - ***mkdir petalinux/2021.1***
  - ***cd Downloads***
  - ***chmod +x petalinux-v2021.1-final-installer.run***
  - ***./petalinux-v2021.1-final-installer.run -d ../petalinux/2021.1***

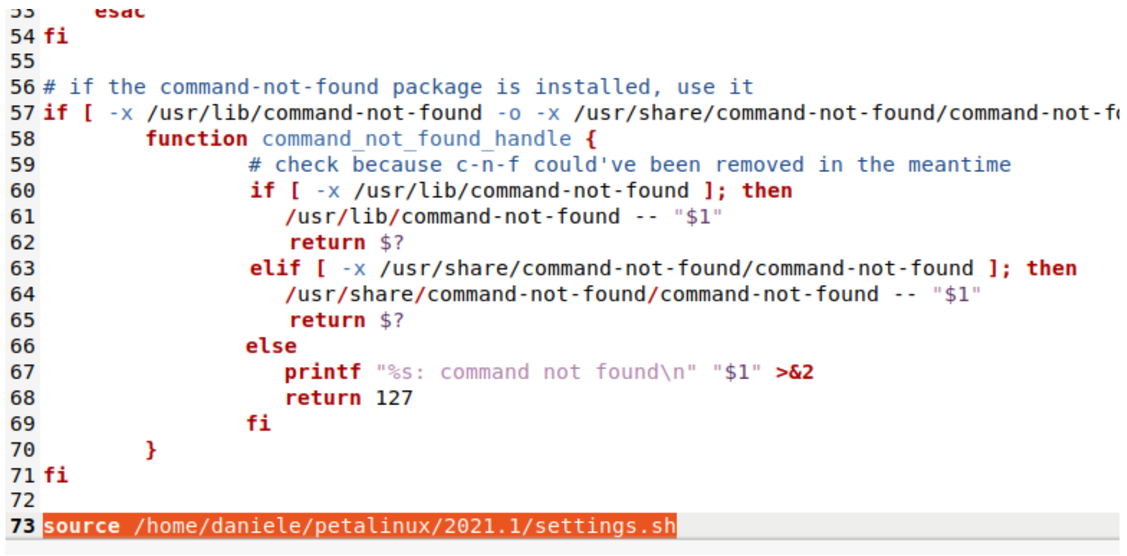
- *OSS : The PetaLinux tools need to be installed as a* ***non-root user***

Petalinux: change /bin/sh to bash

- Petalinux tools require that your host system */bin/sh* is '**bash**'. If you are using Ubuntu distribution like me, probably your */bin/sh* is '**dash**'.
- To change it to bash (This is not the only way you can do it):
  1. In the terminal, run this command: `***chsh -s /bin/bash***`.
  2. Reboot the VM.
  3. Open a terminal and run these commands to make `/bin/sh` link to `/bin/bash`:
     - ***sudo rm /bin/sh***
     - ***sudo ln -s /bin/bash /bin/sh***

Environment setup: Petalinux

- After the installation, the remaining setup is completed automatically by sourcing the provided settings scripts.
  - ➢ ***source petalinux/2021.1/settings.sh***
- You can verify that the working environment has been set:
  - ➢ ***echo $PETALINUX***
- To avoid needing to type the source commands into the shell every time, add a couple of lines to the .bashrc script.
  - To customize this system wide, use a text editor to open your .bashrc file. For Ubuntu, this will be bash.bashrc located in the /etc directory



```
    esac
fi

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

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

Environment setup: Vitis/Vivado

- If you need an editor to **write and compile new code** to run on the MPSoC or you want to **program the FPGA** to configure new devices you need respectively **Vitis** and **Vivado**. Download Vitis from the following link:
  - https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html
- Assuming that you downloaded the file in the Download directory, extract the installer and then run it

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

Environment setup: Vitis/Vivado

- Complete the installation. I decided to install Xilinx in the directory "home/<user>/Xilinx".
- Then, as for petalinux you can **add a new command to your .bashrc** file to setup the environment every time a new terminal is opened:
  - ***sudo gedit /etc/bash.bashrc***
- Once you have the script open, add the command for sourcing the appropriate file. In my case:
  - ***source /home/daniele/Xilinx/Vitis/2021.1/settings64.sh***

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

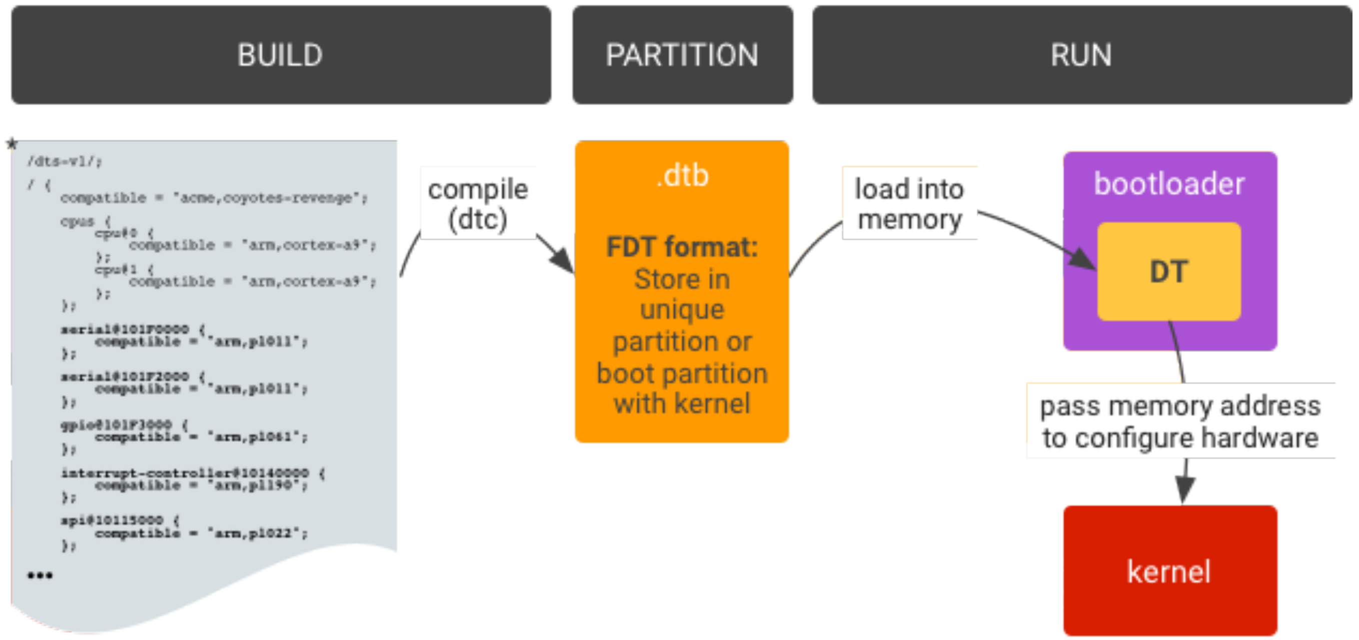
OpenAMP Pre-built application

Pre-built application: BSP

- The simplest way to get started with PetaLinux consists in using a **Board Support Package** (BSP):
  - BSPs are completed and working project archives, ready to be utilized and pre-configured for a specific board
  - BSP download link (I downloaded **ZCU104 Base 2021.1**): https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms.html.
- Create a new *petalinux project* starting from the BSP is easy. You can use the following petalinux command:
  - ***petalinux-create -t project -s /home/<user>/Downloads/xilinx-zcu104-v2021.1-final.bsp -n <name_of_the_project>***
- Now you should have a petalinux project directory.

Pre-built application: Device Tree

- To run an application that make use of OpenAMP to comunicate with the RPU we need to change the **Device Tree Blob** file (.dtb) read by Linux.
- A device tree is a *data structure* describing the hardware components of a particular computer (or Board) so that the operating system's kernel can use and manage those components, including the CPU or CPUs, **RPUs**, the memory, the buses and the peripheral.


BUILD
PARTITION
RUN
compile (dtc)
.dtb
FDT format: Store in unique partition or boot partition with kernel
load into memory
bootloader
DT
pass memory address to configure hardware
kernel

```
/dts-v1/;

/ {
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };

    serial@101F0000 {
        compatible = "arm,pl011";
    };

    serial@101F2000 {
        compatible = "arm,pl011";
    };

    gpio@101F3000 {
        compatible = "arm,pl061";
    };

    interrupt-controller@10140000 {
        compatible = "arm,pl190";
    };

    spi@10115000 {
        compatible = "arm,pl022";
    };

...
```

Pre-built application: Device Tree

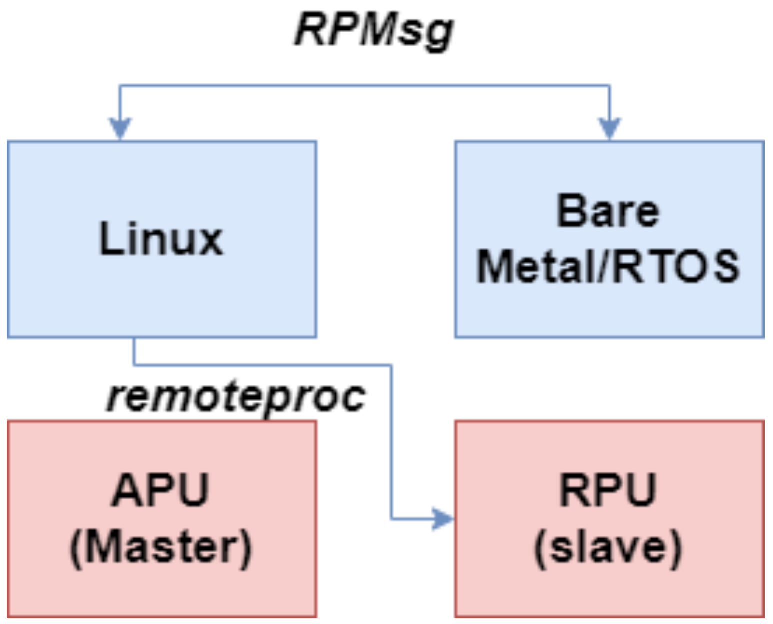
- Petalinux, at compilation time, uses as default device tree blob the file called **system.dtb**. To run the OpenAMP tests you want to use the file called **openamp.dtb** instead, in which the RPU is considered. (Both are in the "*<petalinux_project> /linux /images*" directory)
- Hence, rename the file openamp.dtb (save the old system.dtb before):
  - ***cp <petalinux_project>/pre-built/linux/images/system.dtb <petalinux_project>/pre-built/linux/images/system_old.dtb***
  - ***cp <petalinux_project>/pre-built/linux/images/openamp.dtb <petalinux_project>/pre-built/linux/images/system.dtb***

Pre-built application: Test on QEMU

- To run the pre-built application run this petalinux command in the project directory:
  - ➢ ***petalinux-boot --qemu --prebuilt 3***
- Wait for the system to boot and finally, you can try your pre-built Linux on an emulated ZCU104. Once in the system it is possible to run three example applications that make use of OpenAMP:
  - Echo test
  - Matrix multiplication test
  - Proxy test

Echo Test

- This test application sends a number of payloads from the master to the remote and tests the integrity of the transmitted data:
  1. The echo test application uses the Linux master to boot the remote bare-metal firmware using **remoteproc** in kernel space.
  2. The Linux master then transmits payloads to the remote firmware using **RPMsg** in kernel space. The remote firmware echoes back the received data using **RPMsg** in user space through OpenAMP.
  3. The Linux master verifies and prints the payload.



Echo Test

- To run the test first load the firmware on RPU with the following command:
    - ***echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware***
- then start the RPU by sending the start message:
    - ***echo start > /sys/class/remoteproc/remoteproc0/state***
- Finally, start the Linux Application that make use of RPMsg to communicate with RPU:
    - ***echo_test***
- To stop the RPU:
    - ***echo stop > /sys/class/remoteproc/remoteproc0/state***

Matrix Multiplication Test

- The matrix multiplication application provides a more complex test that generates two matrices on the master. These matrices are then sent to the remote, which is used to multiply the matrices. The remote then sends the result back to the master, which displays the result.
- Run:

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

Proxy Test

- This application creates a proxy between the Linux master and the remote core, which allows the remote firmware to run **Remote Procedure Calls (RPC):**
  - Read and write form the master console
  - Read and write I/O files own by the master.
- The Linux master boots the firmware using the **proxy_app**. The remote firmware executes **I/O operations on the Linux file system** (FS), which is on the master processor. The remote firmware also uses the master console to receive input and display output
- Run:
  - ***proxy_app***

OpenAMP
Kernel Space vs User Space

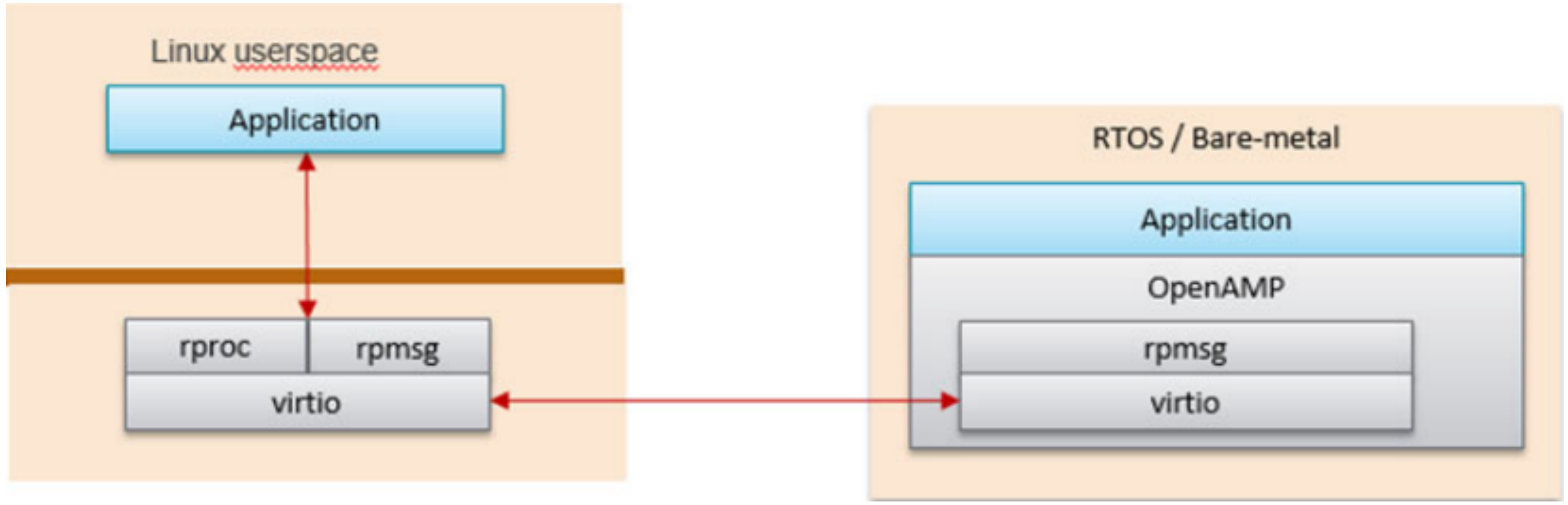
OpenAMP Kernel Space vs User Space

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

- We will see how to configure an OpenAMP application in Kernel Space but it is important to know the differences between these mode of execution.

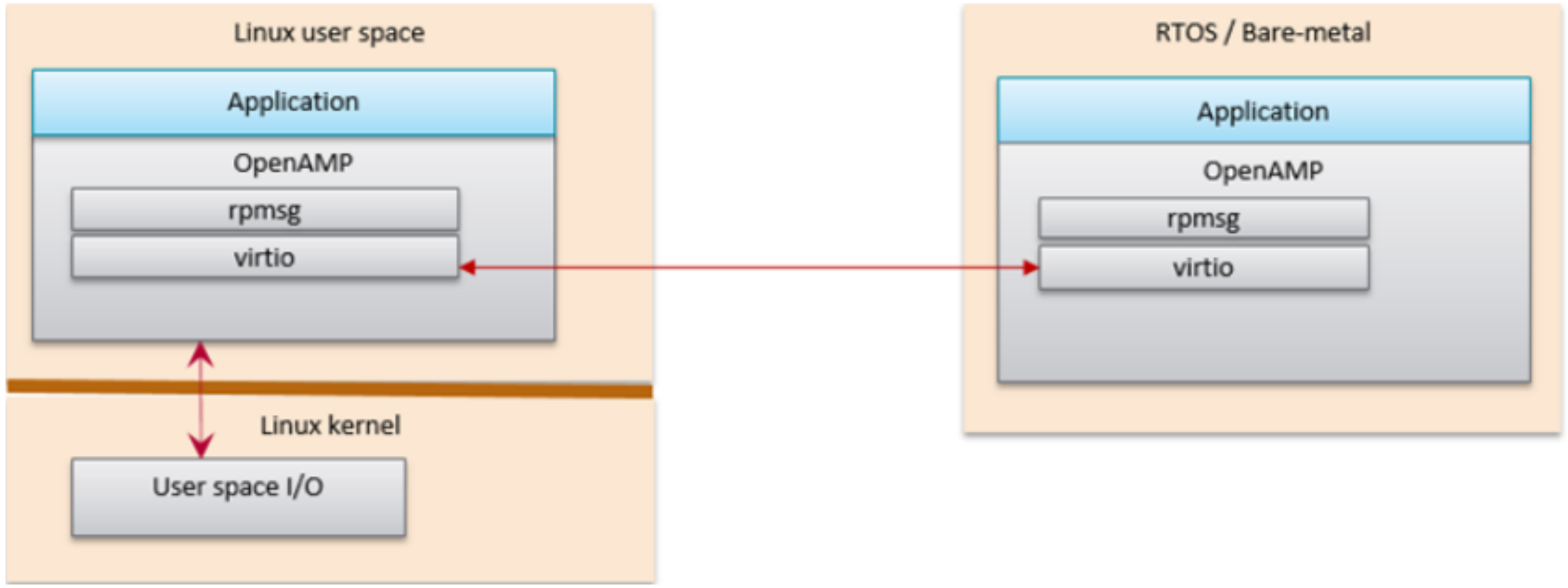
Kernel Space

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



User Space

- OpenAMP library can also be used in Linux userspace. In this configuration, **the remote processor can run independently** to the Linux host processor.
- Limitation: It is not possible to start/stop the remote processor from Linux userspace



Building an OpenAMP application (Kernel Space)

Building an OpenAMP application

- How to create a user application for our board (in this example for zcu104) that uses OpenAMP to communicate with the RPU?
- To do it you need to:
    1. Build an RPU firmware.
    2. Build a Linux OpenAMP Kernel Space Application with Petalinux.
    3. Configure and Compile the Linux Kernel with Petalinux.

Hardware Definition File

- To build an RPU firmware we are going to use **Vitis**.
- First, we need the **Hardware Definition File** (file “***.xsa***” that stand for Xilinx Support Archive) targeting our board. There are three possibility:
    1. During the Vitis installation there are some .xsa files installed by **default** in the path: *Xilinx/Vitis/2021.1/data/embeddedsw/lib/fixed_hwplatforms*
    2. If your board is not in the Vitis directory you can **download** it from the Xilinx website (In my case "*ZCU104 Base 2021.1*"): https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms.html
    3. If the board isn’t a standard board for developer you probably will not find it on the website. Hence, you have to **create from scratch** the “.xsa” file exploiting the Vivado tool. This is out of the scope of this presentation

Build an RPU firmware

Build an RPU firmware

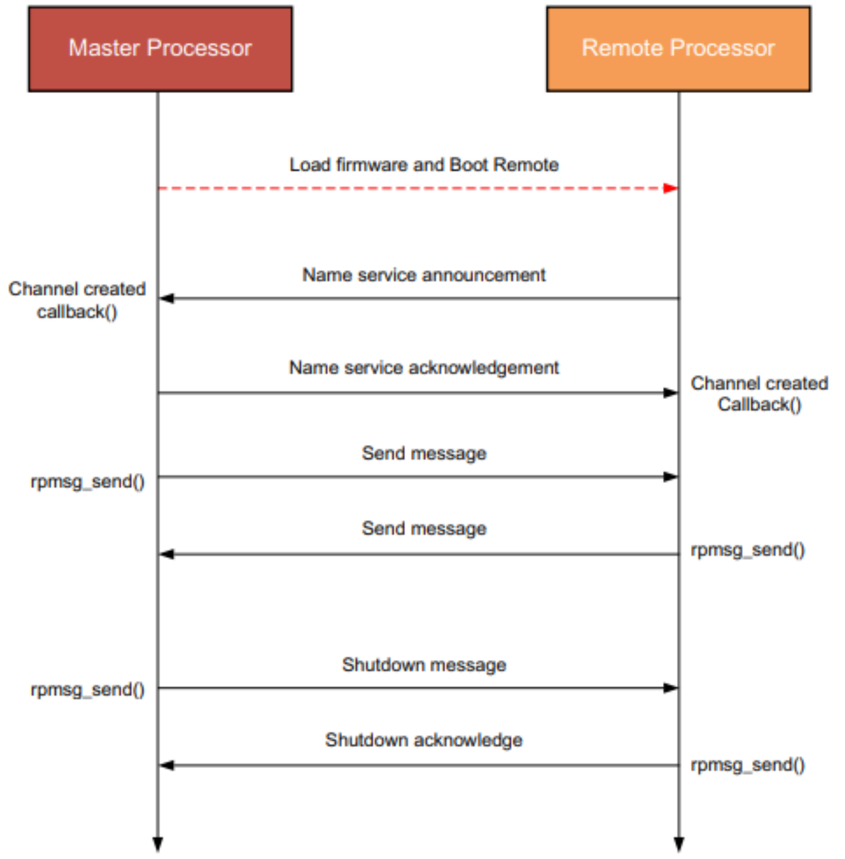
1. From the Xilinx Vitis window, create the application project by selecting **File > New > Application Projects** or select **Create Application Project**.
2. The wizard will guide you through the steps of creating new application projects.
   i. Go in **create a new platform project from hardware (XSA)**. Click **Browse...** and select the .xsa file previously downloaded or the one related to your chip.
   ii. In Boot Components select **psu_cortexr5_0** and click **Next**.
   iii. Give a Name to the application project. I called it "*firmware-rpu*", and select as target processor "*psu_cortexr5_0*" (If you want to use "psu_cortexr5_1" see the official documentation for the changes), click **Next**.
   iv. In Domain details you have to choose to run bare-metal software or freeRTOS. In this tutorial I chose "**freertos10_xilinx**" as Operating System. Click **Next**.
   v. Finally select **OpenAMP echo-test** as template e and click **Finish**.
3. In the Xilinx Vitis project Explorer, click the platform (in my case xilinx_zcu104_base_202110_1) and click **platform.spr**. Below **freertos10_xilinx_psu_cortexr5_0** click **Board Support Package** and then click **Modify BSP Settings....** Check if OpenAMP and Libmetal are checked. If not, check it and then click **OK**.

Build an RPU firmware: Source Files

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

Communication flow

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



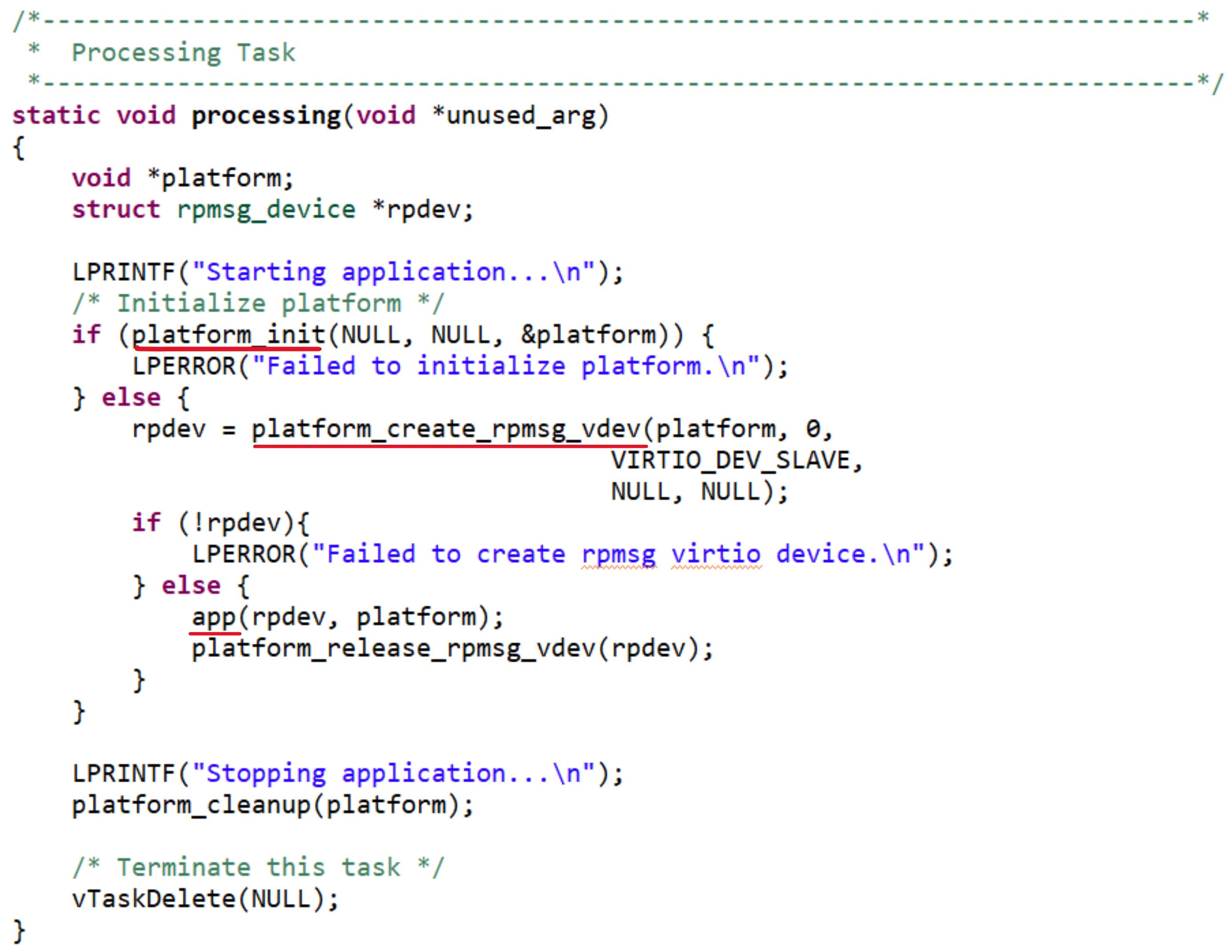
Echo Test RPU: Code

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

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

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

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

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

Echo Test RPU: Code

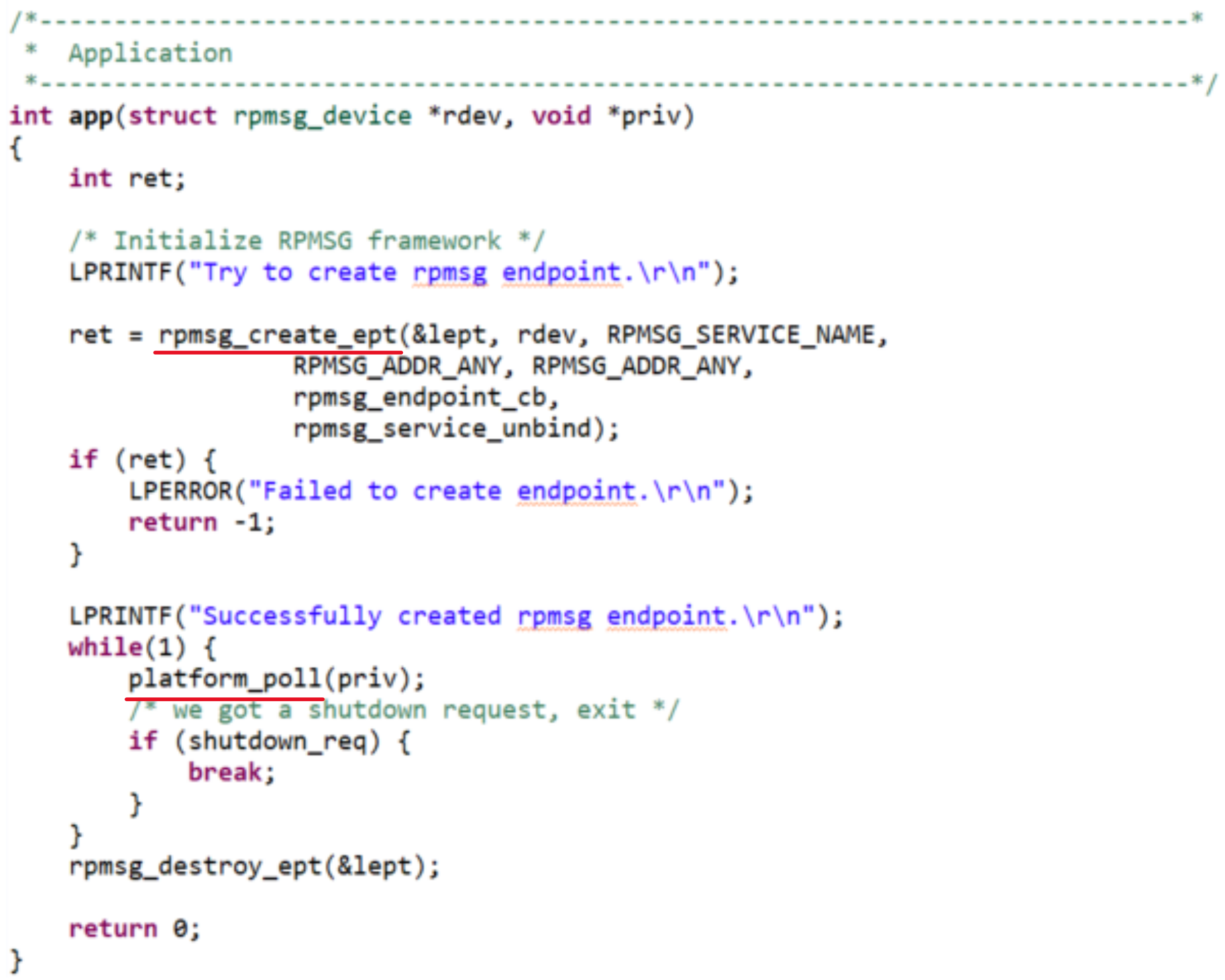
- **platform_init()**:
  - initialize the hardware platform
  - create **remoteproc** struct from the resource table
  - Parse the resource table
- **platform_create_rpmsg_vdev():**
  - Reserve memory for the communication channel
  - Create a VirtIO device
  - Create an RPMsg virtio device
  - return an RPMsg device (or RPMsg channel)
- Then if nothing went wrong it start the **app()** function.



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

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

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

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

Echo Test RPU: Code

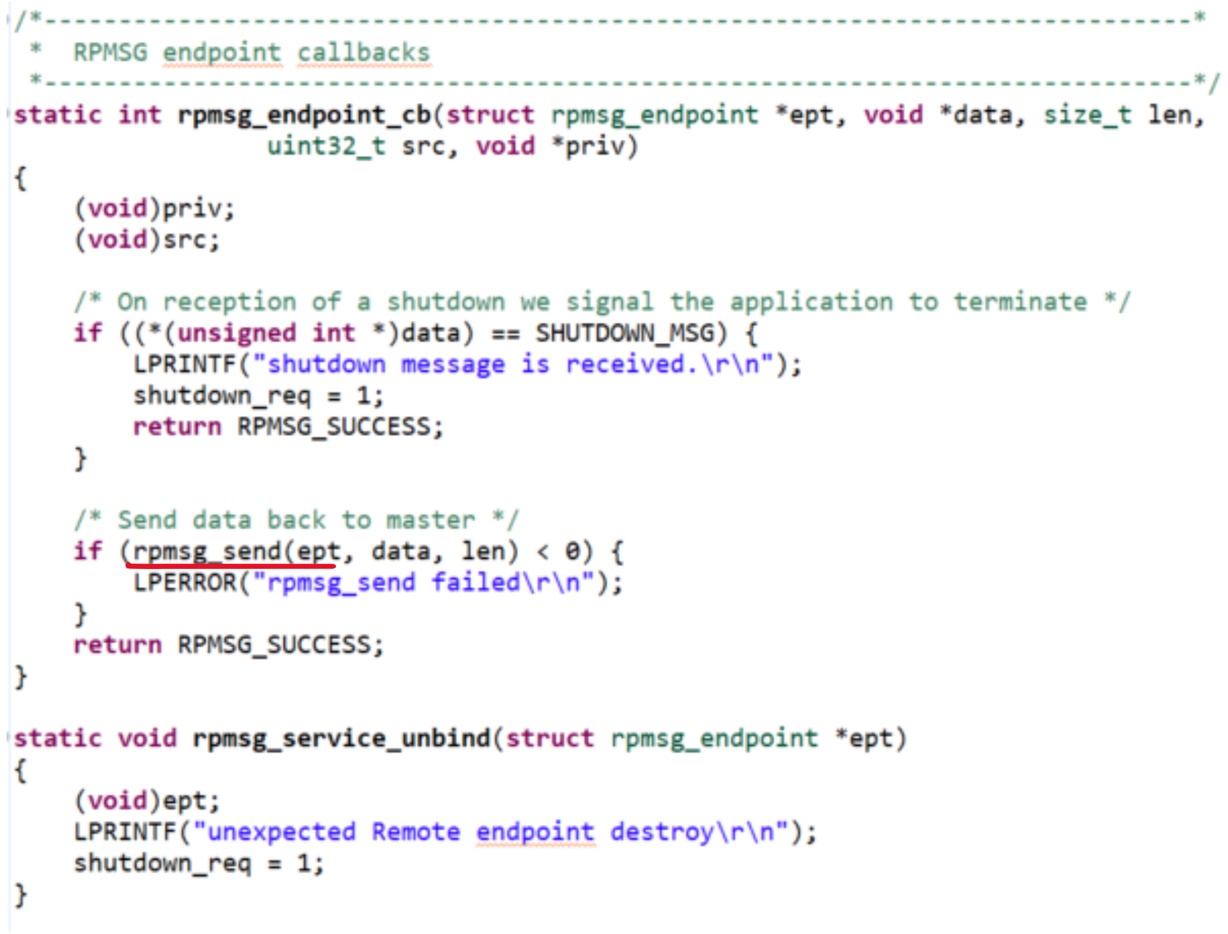
- **rpmsg_create_ept():**
  - Create an endpoint
  - Bind the endpoint to the RPMsg device previously created
  - Bind the endpoint to a callback
  - Bind the endpoint to a service unbind callback, called when the remote endpoint is destroyed
- Therefore, using the function **platform_poll()** the application waits for an interrupt coming from the remote processor



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

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

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

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

    return 0;
}
```

Echo Test RPU: Code

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



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

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

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

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

Petalinux application create

- Once the preparation of the files is finished, click **Debug** to compile the files and the platform. In output you should have a *".elf"* file
- This file must be loaded in the petalinux project to install the RPU firmware in the linux rootfs. Go in the project directory and run:
  - ***petalinux-create -t apps --template install -n <app_name> --enable***
- Then copy the "*.elf*" file in the app directory:
  - ***cp firmware-rpu.elf <petalinux_project>/project-spec/meta-user/recipes-apps/<app_name>/files/***

Petalinux application create

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

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

Device Tree Configuration

- Last but not least adjust the device-tree in the directory:
  - *project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi*

- To make Linux see the RPU as a device, simply append the content of *openamp.dtsi,* present in the same directory, to the *system-user.dtsi* file.

Build a Linux OpenAMP application

Build a Linux OpenAMP application

- Download the code (LICENSE, Makefile, echo_test.c) for the Linux application:
  - https://github.com/OpenAMP/meta-openamp/tree/master/recipes-openamp/rpmsg-examples/rpmsg-echo-test
- Create a new application on Petalinux project:
  - ➢ ***petalinux-create -t apps --template install -n <app_name> --enable***
- Now copy the three downloaded files (LICENSE, Makefile and echo_test.c) in the following directory:
  - ➢ ***cp LICENSE Makefile echo_test.c project-spec/meta-user/recipes-apps/echo-test-kernel/files***

Build a Linux OpenAMP application

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

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

# Add any other object files to this list below


all: $(APP)

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

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

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

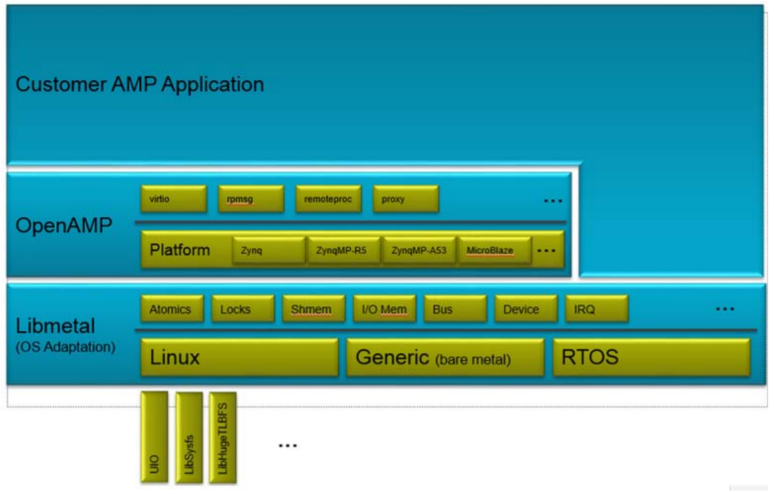
Build a Linux OpenAMP application

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

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

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

S = "${WORKDIR}"

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

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

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

Configure and Compile the Linux Kernel with Petalinux

- Before to build the kernel you need to configure the rootfs and you can do it using Petalinux:
  - ➢ ***petalinux-config -c rootfs***
- Using the interface check the configuration options in the image →



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

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

Petalinux configuration: SD Card

- To run the code on Board using the SD Card we need to change the configuration.
  - Open the terminal in the project directory and type:
    - ➢ ***petalinux-config***
- Check the following configuration option:
  - Image Packaging Configuration ---> Root filesystem type ---> EXT4 (SD/eMMC/SATA/USB)
- exit and save the configuration.

- Finally build the kernel (It might take some time)
  - ➢ ***petalinux-build***
- Create the BOOT.BIN file:
  - ➢ ***petalinux-package --boot --u-boot --format BIN --force***

Test on Board

Test on Board (SD card)

- First format the SD card. To do it use a program like **gparted** (or do it directly from terminal):
  - ➢ ***sudo apt-get install gparted***
  - ➢ ***sudo gparted***
- Create two partitions:
  - **BOOT**: The first partition is called "BOOT" and is 500Mb (leave the first 4Mb free to speed up the memory accesses). The format is "*Fat32*". This partition is used to store the boot loader (u-boot), the device tree and the kernel image.
  - **RootFS**: The second partition in called "RootFS" and can take all the remaining space. The format is "*ext4*". This partition is used to store the root file system.

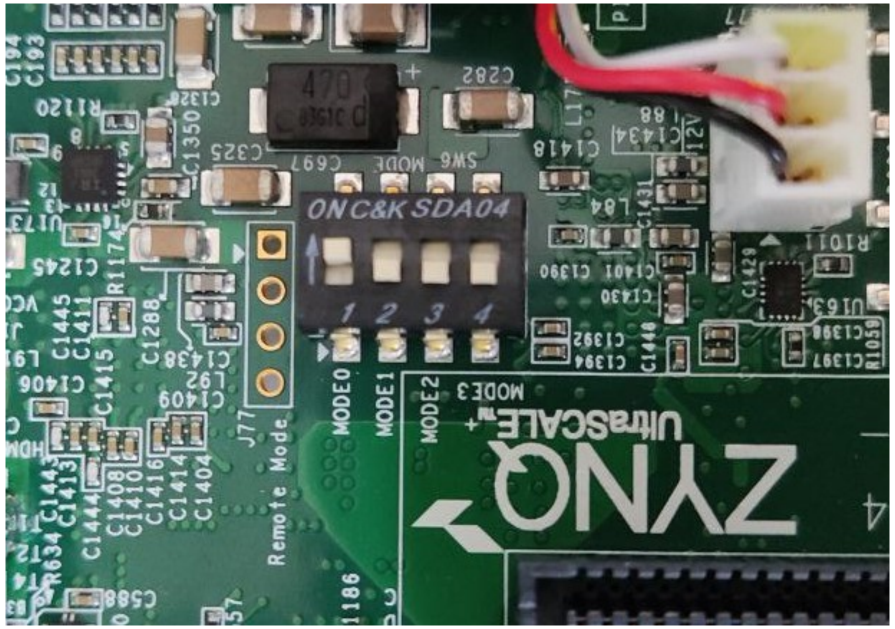
Test on Board (SD card)

- Copy the following three files in the BOOT partition. Go in the project directory and type (I assume that the path for your SD card is */media/<user>/*):
  - ***cp images/linux/BOOT.BIN /media/<user>/BOOT/***
  - ***cp images/linux/image.ub /media/<user>/BOOT/***
  - ***cp images/linux/boot.scr /media/<user>/BOOT/***

- Moreover, you must extract the roofs in the RootFS partition (same assumption on the SD card path):
  - ***sudo tar xvf images/linux/rootfs.tar.gz -C /media/<user>/RootFS***

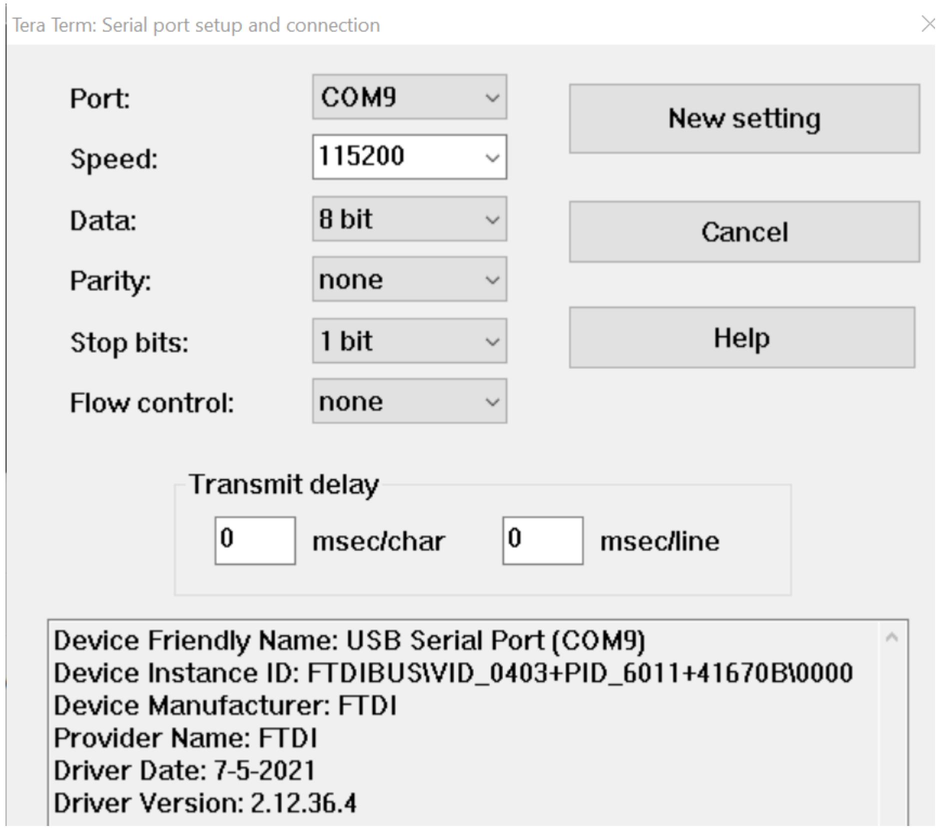
Test on Board: switch configuration

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



Test on Board: UART Drivers

- Then install **FTDI USB UART Drivers**:
  - http://www.ftdichip.com/Drivers/VCP.htm
  - "WHQL Certified. Includes VCP and D2XX. Available as a setup executable": http://www.ftdichip.com/Drivers/CDM/CDM21228_Setup.zip
- Download a program (like **Tera Term** or any other) on your host to receive data from a serial port with a baud rate speed equal to 115200 (In my case the port is COM9 but it probably changes for you)



Port:	COM9
Speed:	115200
Data:	8 bit
Parity:	none
Stop bits:	1 bit
Flow control:	none

Transmit delay
0 msec/char 0 msec/line

```
Device Friendly Name: USB Serial Port (COM9)
Device Instance ID: FTDIBUS\VID_0403+PID_6011+41670B\0000
Device Manufacturer: FTDI
Provider Name: FTDI
Driver Date: 7-5-2021
Driver Version: 2.12.36.4
```

Test on Board

- Finally connect the board to your PC via USB
- Turn on the board:
  - You should see **the first stage boot loader (FSBL)** that loads **u-boot**.
  - Then u-boot load the Linux O.S.

- To test the application run:
  - ***cd /lib/firmware***
  - ***echo firmware-rpu > /sys/class/remoteproc/remoteproc0/firmware***
  - ***echo start > /sys/class/remoteproc/remoteproc0/state***
  - ***echo-test-kernel***