Abstract

To be able to implement control strategies rapidly, a powerful control platform
based on a Xilinx Zynq System-on-Chip (SoC), which combines an FPGA and multiple ARM processors in one chip, needs to be developed. The multiple chip architecture allows to have dedicated processor cores for different tasks. The remote processor, an ARM Cortex-R5, is used to run a computational intensive baremetal application while the master processor, an ARM Cortex-A53, operates PetaLinux as embedded Linux Operating System offering simple communication with other systems via the remote channels.

It is recommended to know the basic functions and commands for the Linux shell changing directories, list the contents of a folder, etc. in order to comprehend the paper. A console input is structured into the directory from where the command should be executed, a $ or # signaling normal user or root and the command itself. The ~ sign is used for the home directory and when it doesn't matter from what directory the command runs. The file paths given in this paper always refer to the home directory as starting point.

Table of contents

1 Preparation

1.1 Setting up the Virtual Machine

The toolchain for building an image for the Trenz-Board, which is based on two Cortex-A53 and two Cortex-R5 processors, is set up on a Linux machine. The host machine uses Windows10 as Operating System on a 64-bit architecture.

For the first step the Virtualbox 5.2.20 must be installed on the host machine and the iso-file (ubuntu-16.04.5-desktop-amd64.iso) for the Virtual Machine, in this case Ubuntu16.04, needs to be downloaded. It is furthermore recommended to install the guest additions for Virtual Machines (VBoxGuestAdditions_5.2.20.iso), a set of device drivers and display application for convenient working. It is important to reserve at least 100 GB memory space for the VDI-hard disk since Vivado and SDK consume much memory. There is no minimum for the RAM but the more the better. If the build process of the kernel-image takes too long, it will be aborted. After the successful booting of the VM the Vivado-Software (Vivado HLx 2017.4: WebPACK and Editions - Linux Self Extracting Web Installer) needs to be downloaded from the Xilinx-website. Vivado can be installed by opening a rootshell via entering `~$: sudo su` in the command line and subsequently dragging and dropping the downloaded file or typing the file-path into the rootshell and pressing enter. The installation folder corresponds to `/opt/Xilinx/` and the tools Vivado and SDK for UltraScale+ must be selected.

Next the Petalinux-installer for version 2017.4 (petalinux-v2017.4-final-installer.run) needs to be downloaded and executed with the command `<plnx>$: bash ./petalinux-v2017.4-final-installer.run` from the folder in which the installer is saved, here. During the installation acknowledge the licenses by pressing Q and afterwards Y for each license agreement and install the required libraries respectively packages like gcc, libssl-dev, libncurses5-dev, zlib1g-dev etc. if asked with the command `~$: sudo apt-get install <package-name1> <package-name2> etc.`

1.2 Creating the Petalinux project

First set the environment variables for Petalinux by sourcing the corresponding script. This is done with the command `~$: source ~/<plnx>/settings.sh`.

If the Board Support Package file with the ending bsp is available, the Petalinux-project can be generated with the command `<plnx>$: petalinux-create –t project –s <path_to_bsp-file>`.

If only the Hardware description file with the ending hdf is given, first a Petalinux-project template must be created with the command `<plnx>$: petalinux-create --type project --template zynqMP --name <proj-name>` where zynqMP corresponds to the UltraScale+ MPSoC. Afterwards the hardware configuration is set by importing the hardware description with the command `<plnx-proj-root>$: petalinux-config --get-hw-description=<path-to-folder-containing-the-hdf-file>`. In the following configuration menu the Subsystem AUTO Hardware Settings for the right processor (psu_cortexa53_0), memory settings (elf_ddr_0 according to device-tree) and the USB-access to the Board serial settings (baudrate=115200) must be checked.

2 Building the firmware

2.1 Creating a Bare-Metal application for the OpenAMP-Framework

OpenAMP framework uses the following key components:

- virtIO: the virtIO is a virtualization standard for network and disk device drivers where only the driver on the guest device is aware it is running in a virtual environment, and cooperates with the hypervisor. This concept is used by RPMsg and remoteproc for a processor to communicate to the remote.

- remoteproc: This API controls the life cycle management (LCM) of the remote processors. The Remoteproc API that OpenAMP uses is compliant with the infrastructure present in the Linux Kernel 3.18 and later. The remoteproc uses information published through the remote processor firmware resource table to allocate system resources and to create virtIO devices.

- RPMsg: This API allows inter-process communications (IPC) between software running on independent cores in an AMP system. This is also compliant with the RPMsg bus infrastructure present in the Linux Kernel version 3.18 and later. [1]

It is common for the master processor in an AMP system to bring up software on the remote cores on a demand-driven basis. These cores then communicate using inter process communication (IPC). This allows the master processor to off-load work to the other processors, called remote processors. Such activities are coordinated and managed by the Xilinx OpenAMP software which builds upon the before mentioned key components.

The general OpenAMP flow as it is shown in figure 1:

1.) The Linux master configures the remote processor and shared memory is created.
2.) The master boots the remote processor.
3.) The remote processor calls remoteproc_resource_init(), which creates and initializes the virtIO resources and the RPMsg channels for the master.
4.) The master receives these channels and invokes the callback channel that was created.
5.) The master responds to the remote context, acknowledging the remote processor and application
6.) The remote invokes the RPMsg channel that was registered. The RPMsg channel is now established, and both sides can use the RPMsg calls to communicate. [2]

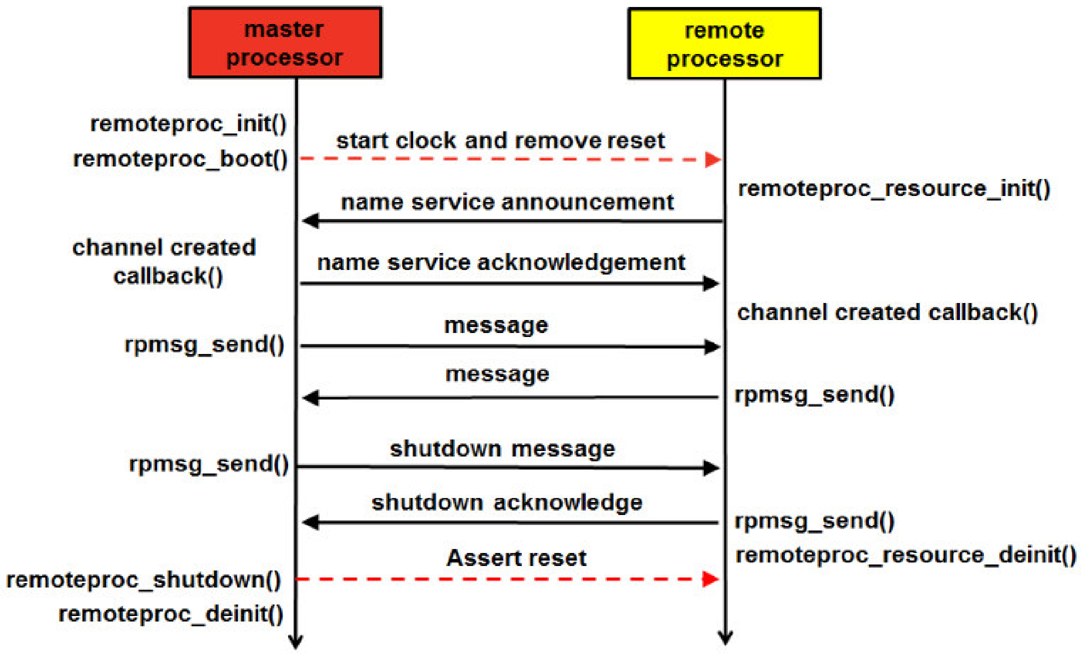


figure 1: General OpenAMP flow

The Bare-Metal application, in this case a square-calculator, will run on the remote processor Cortex-R5 and square numbers that the calculator receives from the Linux master running on the processor Cortex-A53.

First set the environment variables for SDK with the command `~$: source /opt/Xilinx/SDK/2017.4./settings.sh` and start SDK `~$: xsdk &`. The &-sign lets SDK run in background. Secondly define the path to the workspace-folder, by default `~/workspace`, and create a new Application project. For the target hardware the corresponding hdf-file of the target-board must be chosen and the Cortex-R5 processor psu_cortexr5_0 must be selected. Thirdly click "next" and use OpenAMP echo test as template. Eventually check whether the compiler-flags are set properly, `compiler=armr5-none-eabi-gcc` and `compiler_flags= -O2 –c –mcpu=cortex-r5`

The following key source files are available in the newly created Xilinx SDK application:

- Platform Info (platform_info.c/.h):
  - These files contain hard-coded, platform-specific values used to get necessary information for OpenAMP.
    - #define VRING1_IPI_INTR_VECT or IPI_IRQ_VECT_ID: This is the inter-processor interrupt (IPI) vector for the remote processor.
    - #define IPI_BASE_ADDR: The IPI base address for the remote processor.
    - #define RPMSG_CHAN_NAME: The name used to identify a communication channel between two processors

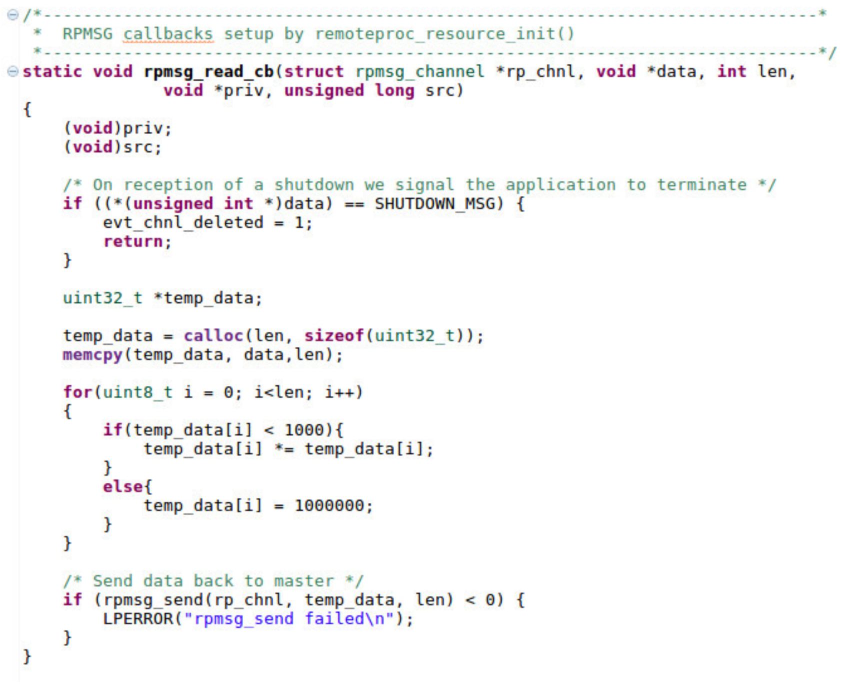
- Resource Table (rsc_table.c/.h):
    - The resource table contains entries that specify the memory and virtIO device resources. The virtIO device contains device features, vring addresses, size, and alignment information. The resource table entries are specified in rsc_table.c and the remote_resource_table structure is specified in rsc_table.h. For the RSC_RPROC_MEM resource, the Linux kernel remoteproc allocates shared memory for vrings and RPMsg buffers from the memory specified in this resource. It should not overlap its address with the memory nodes in the device tree which are used to load the firmware.

- Helper (helper.c/.h):
    - They contain platform-specific APIs that allow the remote application to communicate with the hardware. They include functions to initialize and control the GIC. [3]

The function that transmits data to the master processor is called rpmsg_read_cb and resides inside rpmsg-echo.c. An excerpt is shown in figure 2.

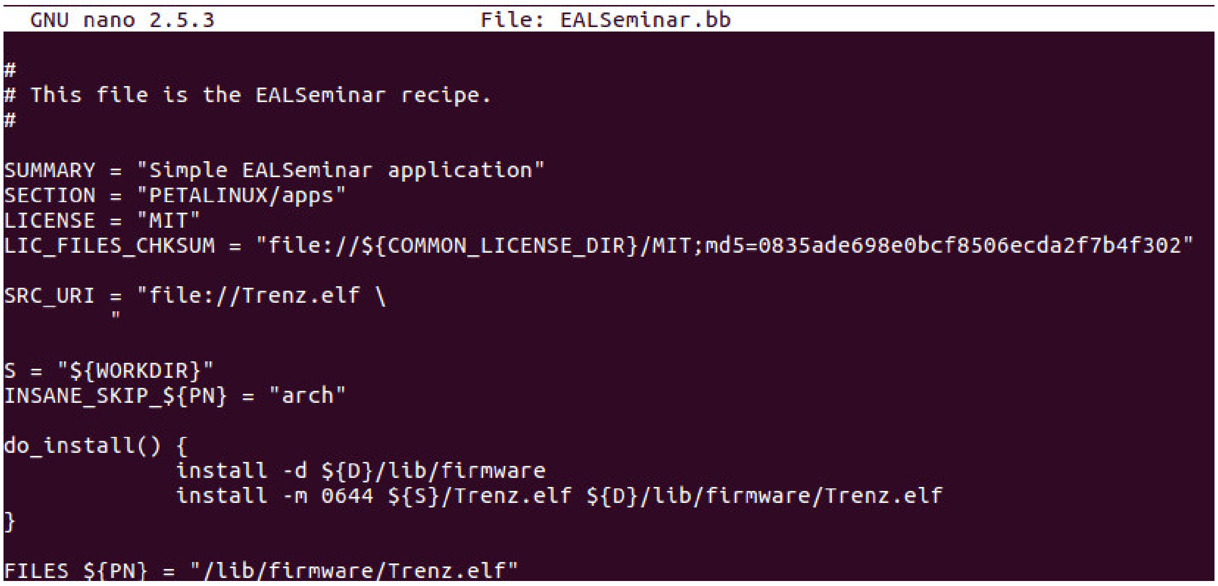
The 32-bit pointer temp_data reads in the data, at maximum one thousand 32-bit integers what corresponds to 4kB, squares the values and sends them back to the master processor as second argument of rpmsg_send. After building the project the SDK-console outputs whether building was successful or not. The generated elf-file resides in `workspace/<SDK-proj-name>/Debug/`.

In order to include the bare-metal application in the Kernel image, create a new application with the command `<plnx-proj-root>$: petalinux-create –t apps –template install –n <BM-app-name> --enable`. Copy the created elf-file into the folder `<plnx-proj-root>/project-spec/meta-user/recipes-apps/<BM-app-name>/files/` and modify the corresponding recipe `<plnx-proj-root>/project-spec/meta-user/recipes-apps/<BM-app-name>/<BM-app-name>.bb` according to figure 3.



```c
/*-----------------------------------------------------------------------------*
 *  RPMSG callbacks setup by remoteproc_resource_init()
 *-----------------------------------------------------------------------------*/
static void rpmsg_read_cb(struct rpmsg_channel *rp_chnl, void *data, int len,
                void *priv, unsigned long src)
{
    (void)priv;
    (void)src;

    /* On reception of a shutdown we signal the application to terminate */
    if ((*(unsigned int *)data) == SHUTDOWN_MSG) {
        evt_chnl_deleted = 1;
        return;
    }

    uint32_t *temp_data;

    temp_data = calloc(len, sizeof(uint32_t));
    memcpy(temp_data, data,len);

    for(uint8_t i = 0; i<len; i++)
    {
        if(temp_data[i] < 1000){
            temp_data[i] *= temp_data[i];
        }
        else{
            temp_data[i] = 1000000;
        }
    }

    /* Send data back to master */
    if (rpmsg_send(rp_chnl, temp_data, len) < 0) {
        LPERROR("rpmsg_send failed\n");
    }
}
```

figure 2: rpmsg-echo.c excerpt



```
  GNU nano 2.5.3                    File: EALSeminar.bb

#
# This file is the EALSeminar recipe.
#

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

SRC_URI = "file://Trenz.elf \
        "

S = "${WORKDIR}"
INSANE_SKIP_${PN} = "arch"

do_install() {
             install -d ${D}/lib/firmware
             install -m 0644 ${S}/Trenz.elf ${D}/lib/firmware/Trenz.elf
}

FILES ${PN} = "/lib/firmware/Trenz.elf"
```

figure 3: bare-metal application recipe

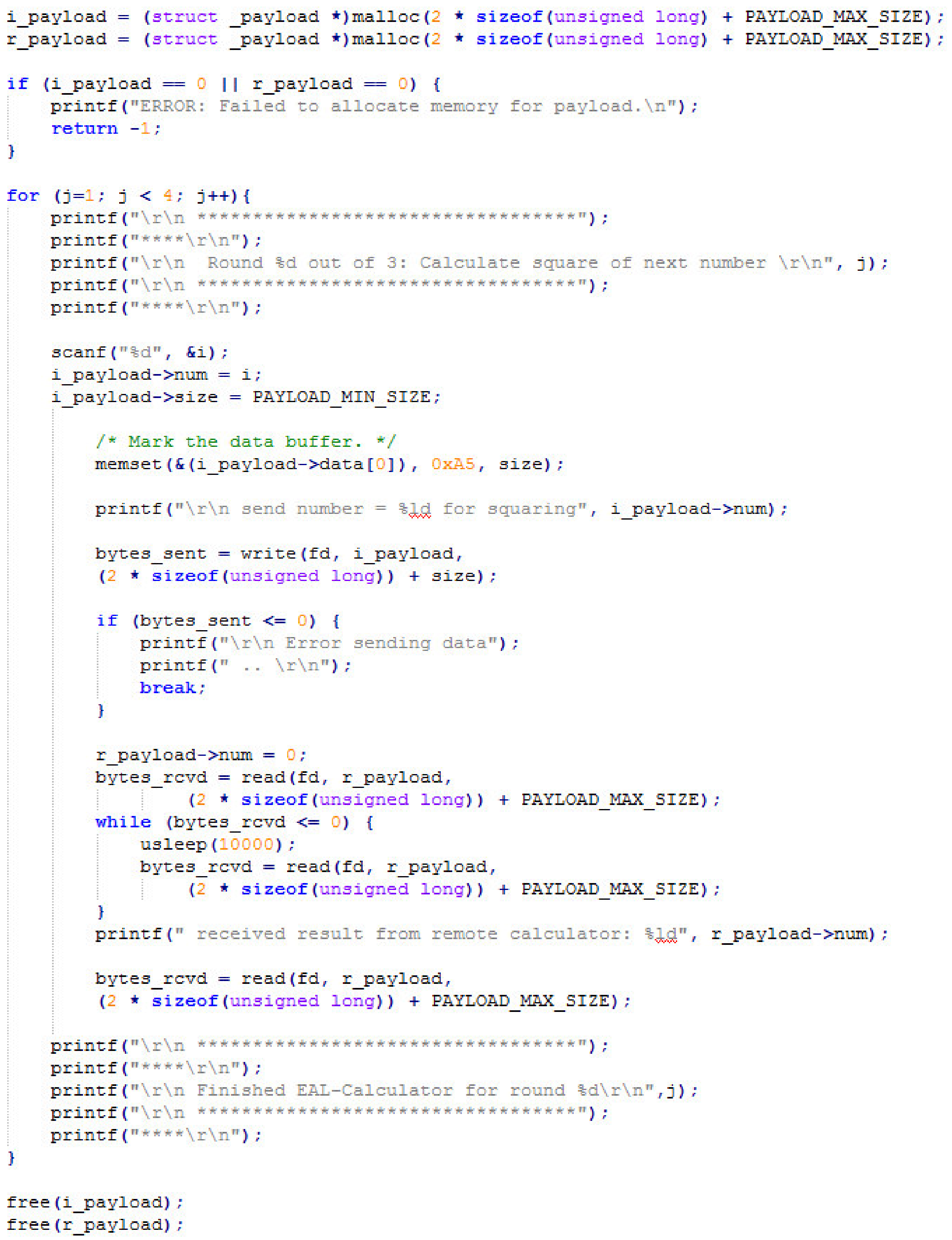
2.2 Creation of the application running on the master processor

First to create an application based on the programming language C type the command `<plnx-proj-root>$: petalinux-create -t apps --template c --name <app-name> --enable` and afterwards navigate to the newly created application with `<plnx-proj-root>$: cd <plnx-proj-root>/project-spec/meta-user/recipes-apps/<app-name>/files`.

The source code must be written into the file `<app-name>.c`

The control application on the master processor parses the user input from the console and forwards the data to the remote processor where they get calculated. The echo_test-software from the git-repository `https://github.com/Xilinx/meta-openamp/blob/master/recipes-openamp/rpmsg-examples/rpmsg-echo-test` serves as template. The source file `echo_test.c` is adapted as shown in extracts in figure 4.

First the two struct-pointers i_payload and r_payload, both equipped with members defining a number for identification, the size and data-array itself are allocated. The struct i_payload transports the user input and r_payload saves the received answers from the remote application. The functions write and read are called with the argument i_payload respectively r_payload to transmit respectively receive data to/from the remote processor. The source file must be saved as `<app-name>.c` under the above given path. At last in the recipe-file `<plnx-proj-root>/project-spec/meta-user/recipes-apps/<app-name>/<app-name>.bb` add the name of the Makefile and source file(s) to SRC_URI if necessary but leave the rest.

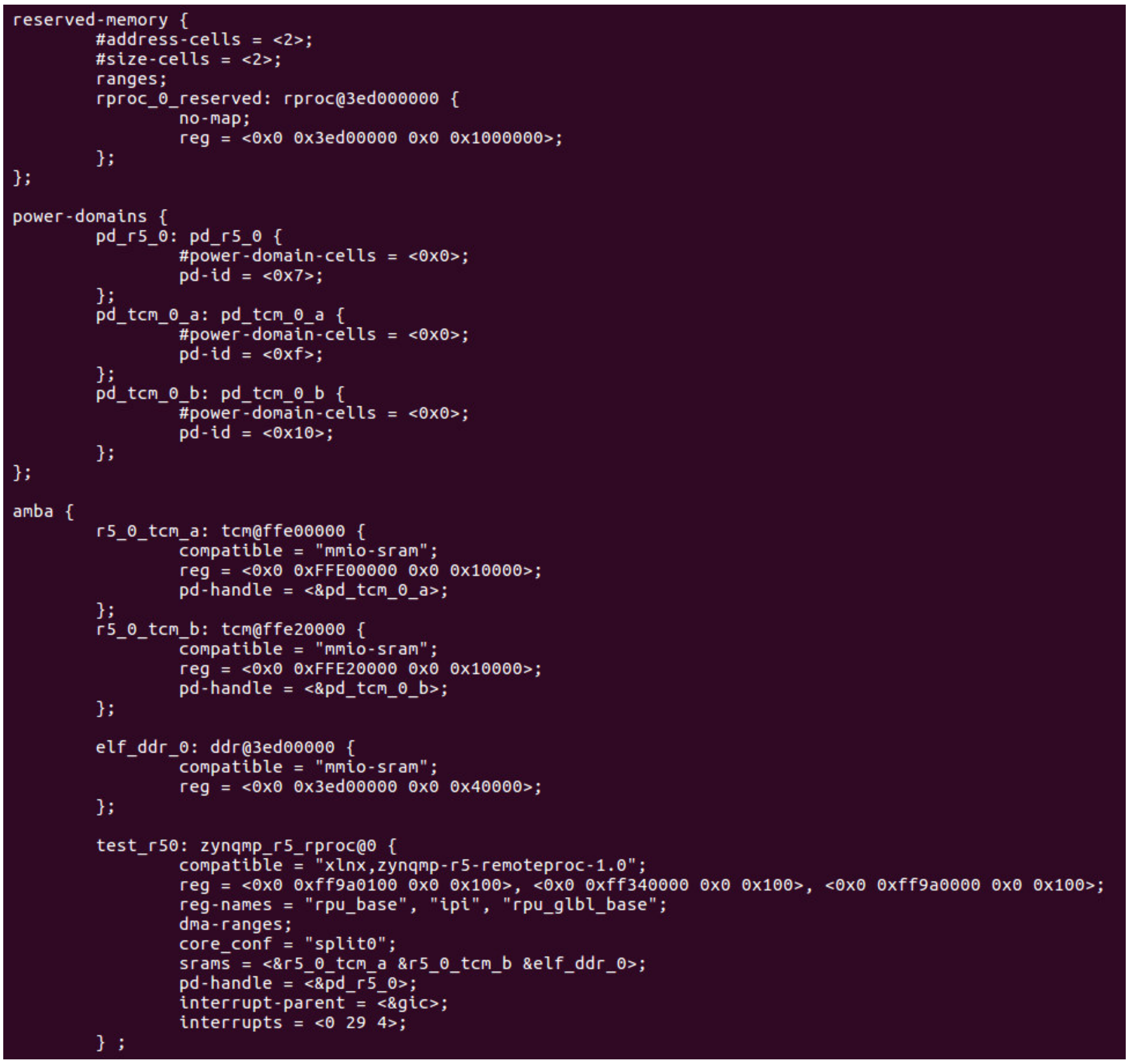


```c
i_payload = (struct _payload *)malloc(2 * sizeof(unsigned long) + PAYLOAD_MAX_SIZE);
r_payload = (struct _payload *)malloc(2 * sizeof(unsigned long) + PAYLOAD_MAX_SIZE);

if (i_payload == 0 || r_payload == 0) {
    printf("ERROR: Failed to allocate memory for payload.\n");
    return -1;
}

for (j=1; j < 4; j++){
    printf("\r\n ***********************************");
    printf("****\r\n");
    printf("\r\n  Round %d out of 3: Calculate square of next number \r\n", j);
    printf("\r\n ***********************************");
    printf("****\r\n");

    scanf("%d", &i);
    i_payload->num = i;
    i_payload->size = PAYLOAD_MIN_SIZE;

        /* Mark the data buffer. */
        memset(&(i_payload->data[0]), 0xA5, size);

        printf("\r\n send number = %ld for squaring", i_payload->num);

        bytes_sent = write(fd, i_payload,
        (2 * sizeof(unsigned long)) + size);

        if (bytes_sent <= 0) {
            printf("\r\n Error sending data");
            printf(" .. \r\n");
            break;
        }

        r_payload->num = 0;
        bytes_rcvd = read(fd, r_payload,
                (2 * sizeof(unsigned long)) + PAYLOAD_MAX_SIZE);
        while (bytes_rcvd <= 0) {
            usleep(10000);
            bytes_rcvd = read(fd, r_payload,
                (2 * sizeof(unsigned long)) + PAYLOAD_MAX_SIZE);
        }
        printf(" received result from remote calculator: %ld", r_payload->num);

        bytes_rcvd = read(fd, r_payload,
        (2 * sizeof(unsigned long)) + PAYLOAD_MAX_SIZE);

    printf("\r\n ***********************************");
    printf("****\r\n");
    printf("\r\n Finished EAL-Calculator for round %d\r\n",j);
    printf("\r\n ***********************************");
    printf("****\r\n");
}

free(i_payload);
free(r_payload);
```

figure 4: control application on master processor

2.3 Modification of the device tree

In order to enable the remote processor and hence AMP, the device-tree must be adapted. The device-tree can be found under `<plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/system-user.dtsi` and should include following text (cf. figure 5).



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

power-domains {
        pd_r5_0: pd_r5_0 {
                #power-domain-cells = <0x0>;
                pd-id = <0x7>;
        };
        pd_tcm_0_a: pd_tcm_0_a {
                #power-domain-cells = <0x0>;
                pd-id = <0xf>;
        };
        pd_tcm_0_b: pd_tcm_0_b {
                #power-domain-cells = <0x0>;
                pd-id = <0x10>;
        };
};

amba {
        r5_0_tcm_a: tcm@ffe00000 {
                compatible = "mmio-sram";
                reg = <0x0 0xFFE00000 0x0 0x10000>;
                pd-handle = <&pd_tcm_0_a>;
        };
        r5_0_tcm_b: tcm@ffe20000 {
                compatible = "mmio-sram";
                reg = <0x0 0xFFE20000 0x0 0x10000>;
                pd-handle = <&pd_tcm_0_b>;
        };

        elf_ddr_0: ddr@3ed00000 {
                compatible = "mmio-sram";
                reg = <0x0 0x3ed00000 0x0 0x40000>;
        };

        test_r50: zynqmp_r5_rproc@0 {
                compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
                reg = <0x0 0xff9a0100 0x0 0x100>, <0x0 0xff340000 0x0 0x100>, <0x0 0xff9a0000 0x0 0x100>;
                reg-names = "rpu_base", "ipi", "rpu_glbl_base";
                dma-ranges;
                core_conf = "split0";
                srams = <&r5_0_tcm_a &r5_0_tcm_b &elf_ddr_0>;
                pd-handle = <&pd_r5_0>;
                interrupt-parent = <&gic>;
                interrupts = <0 29 4>;
        } ;
```

figure 5: modification of the device-tree

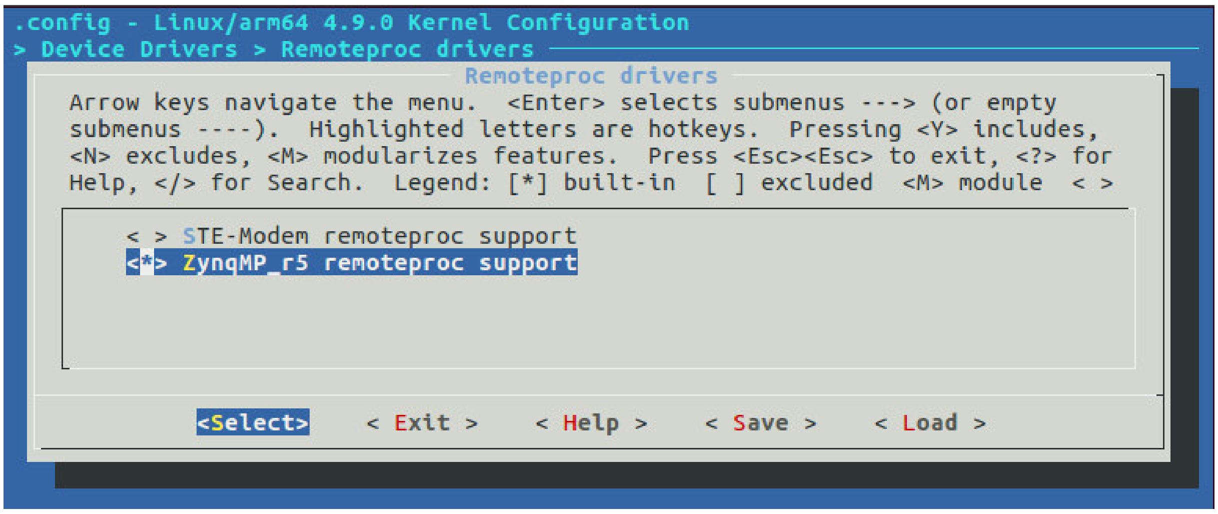
If the target-boards ZynqUltraScale+ ZCU104 or trenz TE0808 are utilized the remote processor is abbreviated with rproc and the executable memory region elf_ddr_0 starts at 0x3ed000000, as one can read in the linkerscript-file `lscript.ld`, that exists inside `workspace/<SDK-proj-name>/src`. The labels r5_0_tcm_a and r5_0_tcm_b define the memory space that is used by the remote processor firmware (= bare-metal application) and the option core_conf = “split0” determines the usage of core 0 of the Cortex-R5.

If the trenz-board TE0808 is being used, some minor changes for the FSBL have to be done. The FSBL-files can be found under `<plnx-proj-root>/components/plnx_workspace/fsbl/fsbl/src`. The files `Si5345-Registers.h`, `si5345.c` and `si534x.h` must be added to the src-folder and `xfsbl_board.c`, `xfsbl_board.h`, `xfsbl_main.c` must be adapted. The diff-command in Linux displays the little differences between the TE0808 and ZCU104 boards.

3 Implementation

3.1 Creation of the Kernel- and Bootimage for OpenAMP

First of all the kernel must be configured with the command `<plnx-proj-root>$: petalinux-config –c kernel`. After a while the configuration menu shows up. Here the settings „Enable loadable module support“, Device Drivers → Generic Driver Options → Userspace firmware loading support must be enabled. Furthermore it is convenient to enable the setting ZynqMP_r5 remoteproc support in the same submenu, so that the remote processor needn't to be manually loaded every time after booting (cf. figure 6)



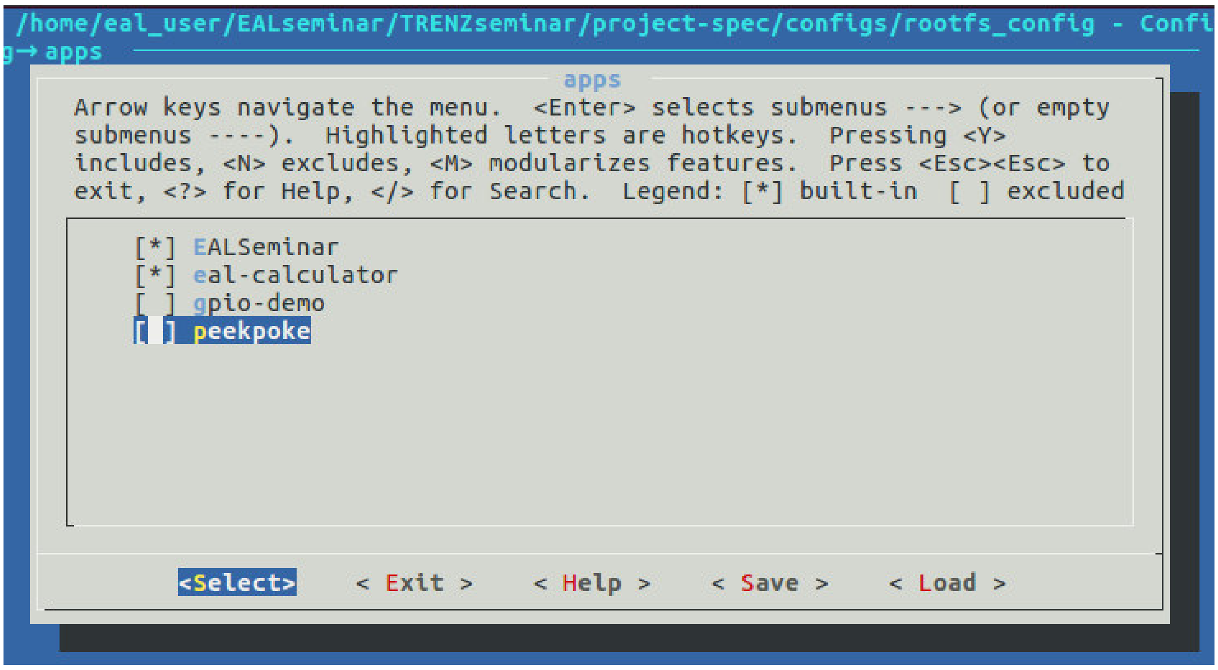
```
.config - Linux/arm64 4.9.0 Kernel Configuration
> Device Drivers > Remoteproc drivers
                         Remoteproc drivers
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for
  Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      < > STE-Modem remoteproc support
      <*> ZynqMP_r5 remoteproc support

          <Select>    < Exit >    < Help >    < Save >    < Load >
```

figure 6: remoteproc kernel configuration:

Secondly the root file system is being configured. The configuration menu is accessed with the command `<plnx-proj-root>$: petalinux-config –c rootfs`. The apps generated in chapter 2.1. and chapter 2.2 must be included (cf. figure 7) and the password can be changed. (in this case the login is **user: root - password: EAL**)

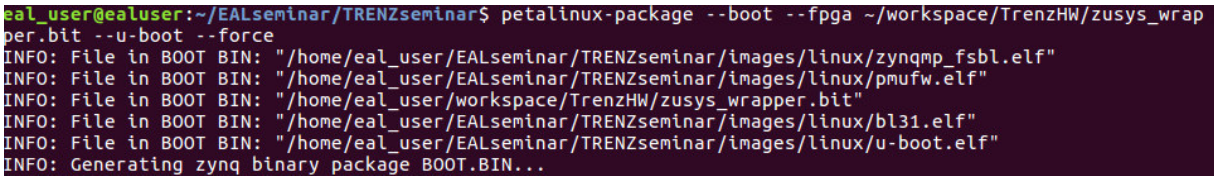
After configuring the kernel image is finally built with the command `<plnx-proj-root>$: petalinux-build`. The kernel image is named `image.ub` and among other build-files can be found under the path `<plnx-proj-root>/images/linux`.

The bootloader, needed for coordinating the boot procedure, is generated with the command `<plnx-proj-root>$: petalinux-package –boot –fpga <fpga-bitstream> –u-boot –force` and consists of `zynqmp_fsbl.elf` (the first stage bootloader), `pmufw.elf` (Power Management Unit firmware), `zusys_wrapper.bit` (fpga-bitstream for the TE0808-board), `bl31.elf` (ARM Trusted firmware) and `u-boot.elf` (second stage bootloader).



```
/home/eal_user/EALseminar/TRENZseminar/project-spec/configs/rootfs_config - Confi
g→ apps
                                   apps
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded

      [*] EALSeminar
      [*] eal-calculator
      [ ] gpio-demo
      [ ] peekpoke

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

figure 7: apps rootfs configuration



```
eal_user@ealuser:~/EALseminar/TRENZseminar$ petalinux-package --boot --fpga ~/workspace/TrenzHW/zusys_wrap
per.bit --u-boot --force
INFO: File in BOOT BIN: "/home/eal_user/EALseminar/TRENZseminar/images/linux/zynqmp_fsbl.elf"
INFO: File in BOOT BIN: "/home/eal_user/EALseminar/TRENZseminar/images/linux/pmufw.elf"
INFO: File in BOOT BIN: "/home/eal_user/workspace/TrenzHW/zusys_wrapper.bit"
INFO: File in BOOT BIN: "/home/eal_user/EALseminar/TRENZseminar/images/linux/bl31.elf"
INFO: File in BOOT BIN: "/home/eal_user/EALseminar/TRENZseminar/images/linux/u-boot.elf"
INFO: Generating zynq binary package BOOT.BIN...
```

figure 8: generation of the bootloader

The generated `BOOT.BIN` can also be found under `<plnx-proj-root>/images/linux`. Finally the `BOOT.bin` and `image.ub` can be transferred into the first FAT32-partition of the SD-Card which must have at least a memory size of 40MB. If the TE0808-Board is used, the SD-card must contain at least two partititions. Check with the command `~$: dmesg` the name of the SD-card, then enter the command `~$: sudo fdisk /dev/<sd-card>`. Within the fdisk-CLI first delete old partitions. Make a new partition by typing 'n'. Make it primary by selecting 'p' then use the default partition number and first sector. Set aside 1G for this partition by typing +1G and set bootable flag for this partition by typing 'a'. Afterwards make the root partition by again typing 'n', selecting primary partition and leaving the first and last sector default. Verify the partition table with ‘p’ where the two new created partitions should be shown and afterwards type 'w' to save and exit the fdisk-CLI, sometimes the SD-card must be reinserted to see changes.

The next step is to format the first partition using the FAT-format with the command `~$: sudo mkfs.vfat -F 32 -n boot /dev/<sd-card>1` and for the second partition the ext4-format with `~$: mkfs.ext4 -L root /dev/<sd-card>2`.

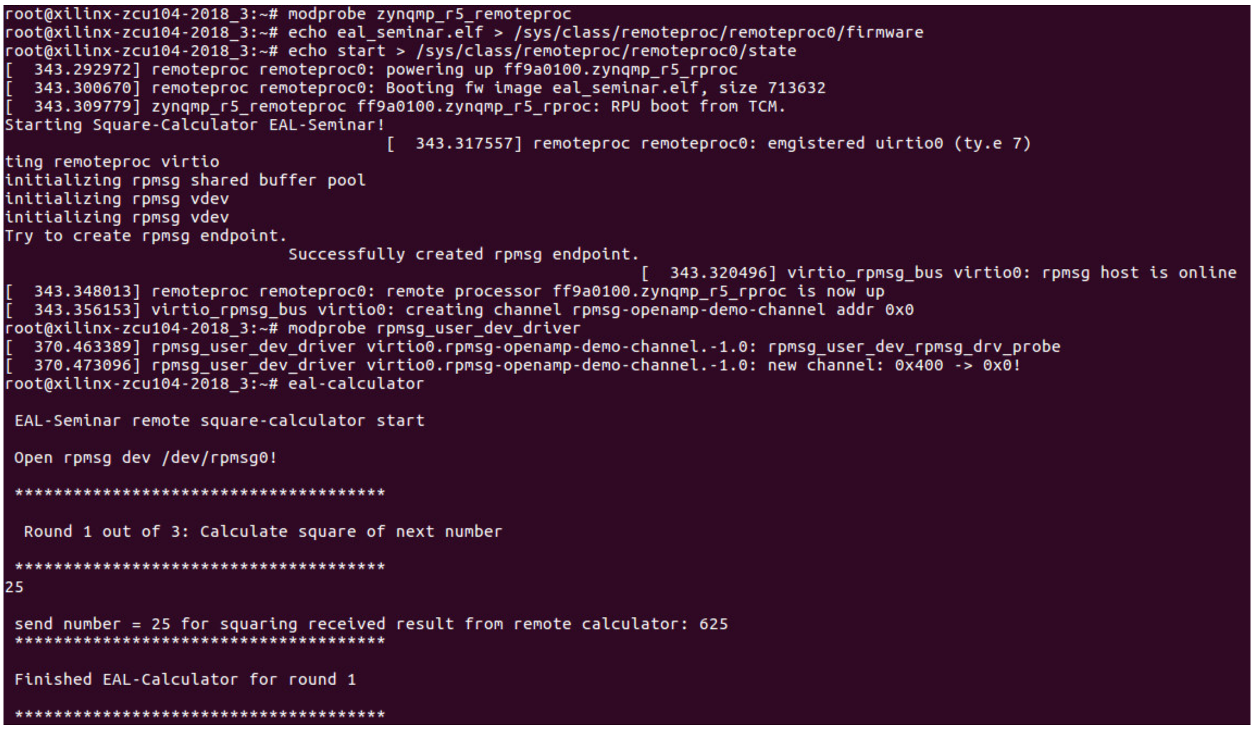
Subsequently mount the two partitions, for this purpose generate two folders `~$: mkdir sdc_boot` and `~$: mkdir sdc_root` and mount them with `~$: sudo mount /dev/<sd-card>1 ~/sdc_boot` respectively `~$: sudo mount /dev/<sd-card>2 ~/sdc_root`.

Finally copy BOOT.BIN and image.ub to the folder sdc_boot and execute `~$: sudo tar zxf  <plnx-proj-root>/images/linux/rootfs.tar.gz -C ~/sdc_root`. At the end unmount both folders with `~$: sudo umount sdc_boot/ sdc_root/.`

3.2 Start-up of the target board

For Xilinx devices the bash-script `install_drivers` under `/opt/Xilinx/Vivado/2018.3/data/xicom/cable_drivers/lin64/install_script/install_drivers` must be executed to source the USB-drivers, afterwards the ouput is displayed on a new terminal with the command `~$: sudo picocom -b 115200 /dev/ttyUSB<number>` .The option b corresponds to the baudrate that has been set in chapter 1.2 and number can be obtained from a dmesg-output, type `~$: dmesg`. After the login with the in chapter 3.1 mentioned username and password, the calculator demonstration application can be launched after following some steps beforehand (cf. figure 9). The bare-metal application, `eal_seminar.elf`, resides inside `/lib/firmware/` and its counterpart, `eal-calculator`, inside `/usr/bin/`.

For starting the TE0808-Board the autoboot must be interrupted by pressing any key, then type `ZynqMP: load mmc 1 0x10000000 image.ub` and afterwards `ZynqMP: bootm`



```
root@xilinx-zcu104-2018_3:~# modprobe zynqmp_r5_remoteproc
root@xilinx-zcu104-2018_3:~# echo eal_seminar.elf > /sys/class/remoteproc/remoteproc0/firmware
root@xilinx-zcu104-2018_3:~# echo start > /sys/class/remoteproc/remoteproc0/state
[  343.292972] remoteproc remoteproc0: powering up ff9a0100.zynqmp_r5_rproc
[  343.300670] remoteproc remoteproc0: Booting fw image eal_seminar.elf, size 713632
[  343.309779] zynqmp_r5_remoteproc ff9a0100.zynqmp_r5_rproc: RPU boot from TCM.
Starting Square-Calculator EAL-Seminar!
                                       [  343.317557] remoteproc remoteproc0: emgistered uirtio0 (ty.e 7)
ting remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
                             Successfully created rpmsg endpoint.
                                                                 [  343.320496] virtio_rpmsg_bus virtio0: rpmsg host is online
[  343.348013] remoteproc remoteproc0: remote processor ff9a0100.zynqmp_r5_rproc is now up
[  343.356153] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel addr 0x0
root@xilinx-zcu104-2018_3:~# modprobe rpmsg_user_dev_driver
[  370.463389] rpmsg_user_dev_driver virtio0.rpmsg-openamp-demo-channel.-1.0: rpmsg_user_dev_rpmsg_drv_probe
[  370.473096] rpmsg_user_dev_driver virtio0.rpmsg-openamp-demo-channel.-1.0: new channel: 0x400 -> 0x0!
root@xilinx-zcu104-2018_3:~# eal-calculator

 EAL-Seminar remote square-calculator start

 Open rpmsg dev /dev/rpmsg0!

 **************************************

  Round 1 out of 3: Calculate square of next number

 **************************************
25

 send number = 25 for squaring received result from remote calculator: 625
 **************************************

 Finished EAL-Calculator for round 1

 **************************************
```

figure 9: demonstration

First the remote processor module is manually loaded with `root@<board_name>#: modprobe zynqmp_r5_remoteproc`. This step is redundant if the kernel has been configured to load this module automatically. Secondly the remote processor is told what firmware to boot from TCM,

`root@<board_name>#: echo <bare-metal application>.elf > /sys/class/remoteproc/remoteproc0/firmware`, and started with `root@<board_name>#: echo start > /sys/class/remoteproc/remoteproc0/state` .

After successful boot the virtIO-device containing the communication channel between the two processors and the rpmsg-endpoint are created. The application on the master processor can ultimately be launched by simply typing its name into the CLI, `root@<board_name>#: <master-processor-application>`. The master application is now parsing the user input and forwards it to the remote processor where the calculation is done. Afterwards the remote processor sends the results back and the master application prints them on the console.

List of figures

Bibliography

[1] Components in AMP, Xilinx OpenAMP Framework UG1186, p. 6

[2] Process overview, Xilinx OpenAMP Framework UG1186, p. 7

[3] OpenAMP SDK key source files, Xilinx OpenAMP Framework UG1186, p. 15