MTRION55EVK Ethernet Test

Created on: Sep 18, 2024

Created by: Grit CHristange

© ARIES Embedded GmbH. The information contained in this document is strictly confidential. This document may not be copied, reproduced, translated, changed or distributed without the written approval of ARIES Embedded GmbH

CHAPTER

ONE

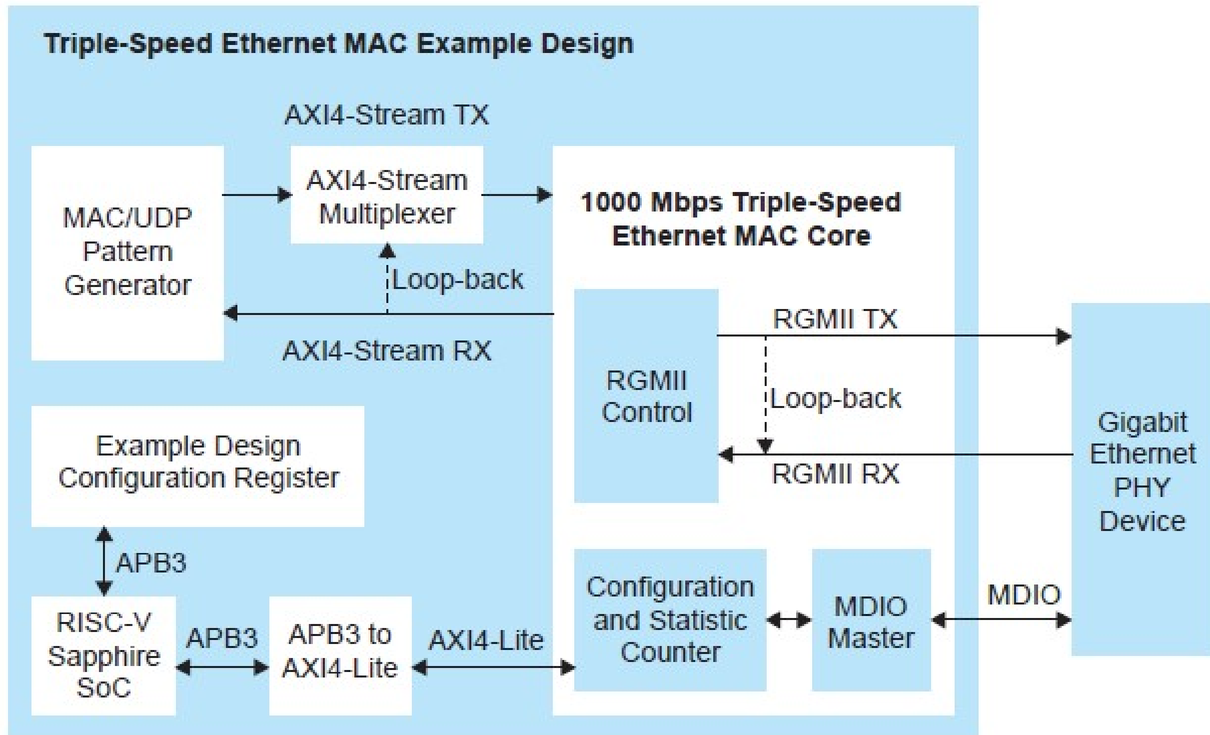
MTRION55EVK ETHERNET TEST REPORT V0

1.1 Intention

In the following we decribe what was done to bring up the **ethernet functionality** of the Aries **MTRION55EVK** Rev.2.0 with MTRION55 FPGA Module Rev.2.0 using Efinix Efinity 2024.1.163 and RISC-V Embedded Software IDE v2024.1.0.1 Software.

1.2 Test Scenario

The test was built upon the example design desribed in the Efinix Triple Speed Ethernet MAC Core User Guide UG-CORE-TSEMAC-v5.0, pg. 20ff.



`Fig1: Example Test Structure`

Means our test structure was like this: FT2232 Mini USB for JTAG Programming (FPGA and SPI Flash via FPGA Loader), Micro USB for RiscV Uart Terminal (115200 Baud) were connected to a Window PC, Ethernet was connected to an Ubuntu (20.04) PC. For sending and receiving of the ethernet frames we used **Wireshark** 3.2.3 and **Packet Sender** 8.6.5 on the latter.

Fig2: Aries Test Execution

1.3 Adapted Efinity Test Design

We started with the RTL sources and RISCV C sources which were found in the \ip\ethmac\T120F324_devkit subdirectory of our Efinity test project where we instantiated the **efx_tsemac 6.0 IP** in the vhdl top entity of our RTL test design. We also adopted the test infra structure as to be found in the example top and added the peripherals in the Interface Designer according to our pinout and clock scenario. We followed UG-CORE-TSEMAC-v5.0, later during HW debugging it turned out that we do not have to invert rgmii_txc.

Besides the tsemac IP we used

- RISCV efx_soc 3.1.0,
- efx_ddr_reset_controller 5.0 and
- efx_apb3_2_axi4_lite 5.0

from the Efinity IP Catalog.

We configured 2 PLLs in the Interface Designer:

The first PLL generates based on a 25MHz input reference

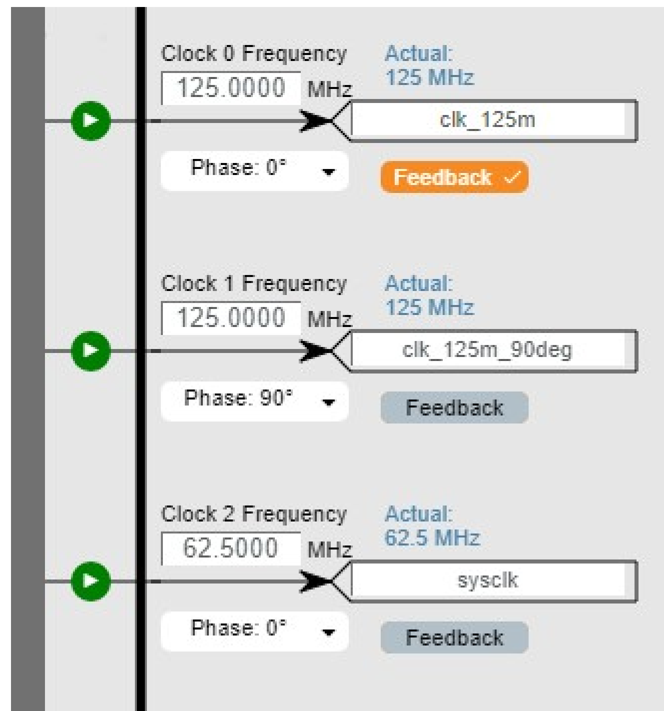
- 2 125 clocks for the ethernet domain, 1 is phase shifted for the RGMII DDR Interface and

- 62.5 MHz for the AXI ethernet streams connected to the MAC IP (e.g. the pattern generation) and RiscV.

To relax the timing of the pattern generation we reduced the speed.

We chose exact half the 125MHz for the control path because of timing violations in the MAC IP crossing the clock domains. (We don't know if false paths would be applicable there.) With a half clock the timing analysis passed. To achieve the timing easier we inserted asynchronous fifos in the AXI ethernet paths to connect the streams with a 125MHz clock to the MAC IP. Means also the clock crossing inside the MAC IP was avoided. For this fifo decoupling we reused the example submodule mac_rx2tx (for the shortcut there), a manually implemented asynchronous fifo, which fitted very well here too.

The 62.5MHz clock had to be considered in the RiscV configuration.



`Fig3: PLL1 Configuration`

The second PLL provides 100 and 400MHz for the DDR memory area using a 48MHz input clock.

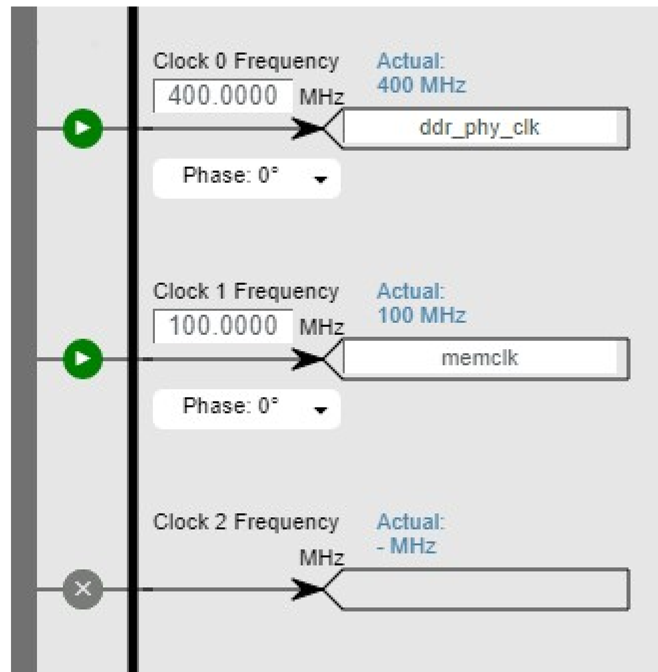


Fig4: PLL2 Configuration

We preserved the register structure of the example design and reused the RiscV Software applying smaller changes: We provided our IP/MAC address in BSP file tseDemo.h in the embedded_sw\riscv\bsp\efinix\EfxSapphireSoc\app subdirectory, which has to match with the host connected (the Ubuntu PC in our case):

```
// 00-24-9b-06-04-6d
#define DST_MAC_H           0x0024
#define DST_MAC_L           0x9b06046d
//#define DST_MAC_H         0xffff
//#define DST_MAC_L         0xffffffff
// ea-e8-5e-00-60-c8
#define SRC_MAC_H           0xeae8
#define SRC_MAC_L           0x5e0060c8
// c0-a8-01-64
// 192-168-1-100
#define SRC_IP              0xc0a80164
// 192-168-0-254
#define DST_IP              0xc0a800fe
// #define DST_IP           0xc0a80165
#define SRC_PORT            0x0521
#define DST_PORT            0x2715
```

Because of another PHY IC at MTRION55EVK (KSZ9131RNXC) we had to change the MDIO configuration in efx_tse_phy.h in embedded_sw\riscv\software\standalone\driver subdirectory. We removed the PhyDlySetRXTX function and avoided other register accesses not applicable for our PHY:

```
// davicom PHY special:
// to unlock extended registers,
//   gc: indirect write x168 @ 8040 addr x40 with wr indication
// Phy_Wr(0x1f, 0x0168);
```

(continues on next page)

(continued from previous page)

```
// Phy_Wr(0x1e, 0x8040);

while(1) {
        // davicom PHY special:
        // read 0x11, a register showing the speed mode
        // Value = Phy_Rd(0x11);
        // Link up and DUPLEX mode
        // if((Value&0x2400) == 0x2400) {
        //      if((Value&0xc000) == 0x8000) {          //1000Mbps
        //        if(PRINTF_EN == 1) {
        //                      bsp_print("Info : Phy Link up on 1000Mbps.");
        //              }
        //              return 0x4;
        //      } else if((Value&0xc000) == 0x4000) {   //100Mbps
        //              if(PRINTF_EN == 1) {
        //                     bsp_print("Info : Phy Link up on 100Mbps.");
        //              }
        //              return 0x2;
        //      } else if((Value&0xc000) == 0x0) {      //10Mbps
        //        if(PRINTF_EN == 1) {
        //                bsp_print("Info : Phy Link up on 10Mbps.");
        //              }
        //              return 0x1;
        //      }
        // }

        bsp_uDelay(100000);
        return 0x04;
}
```

For static timing analysis we followed Efinity Timing Closure User Guide UG-EFN-TIMING-v5.0. For finding a good timing setup we applied:

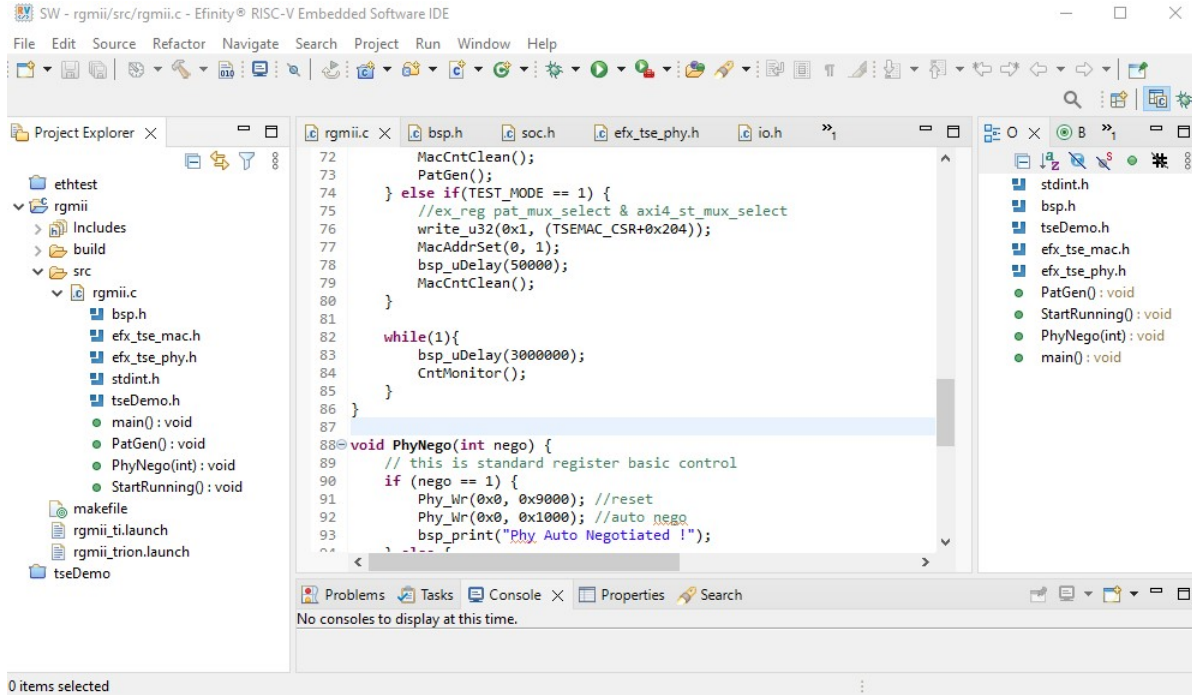
```
c:\Efinity\2024.1\bin>setup.bat
```

and in the project directory:

```
python3 C:\Efinity\2023.2\scripts\efx_run_pnr_sweep.py main.xml
sweep_opt_levels
```

1.4 Debugging and Test

The RiscV SW was imported into the RiscV Embedded Software IDE and built according to the guideline given in the Sapphire RISC-V SoC Hardware and Software User Guide (UG-RISCV-SAPPHIRE-v6.1) g.46ff



`Fig5: SW Project Import`

The resulting .bin File was combined with the FPGA .hex image according the directives found in the Efinity Software User Guide UG-EFN-SOFTWARE USER GUIDE-v13.5 pg.94 (Program Multiple Images (Bitstream and Data)). For Flashing we used Programming Mode “SPI Active using JTAG Bridge(new)” with the bridge image provided in the Efinity installation (Efinity\2024.1\pgm\fli\trion\u00220A79_t55.bit).

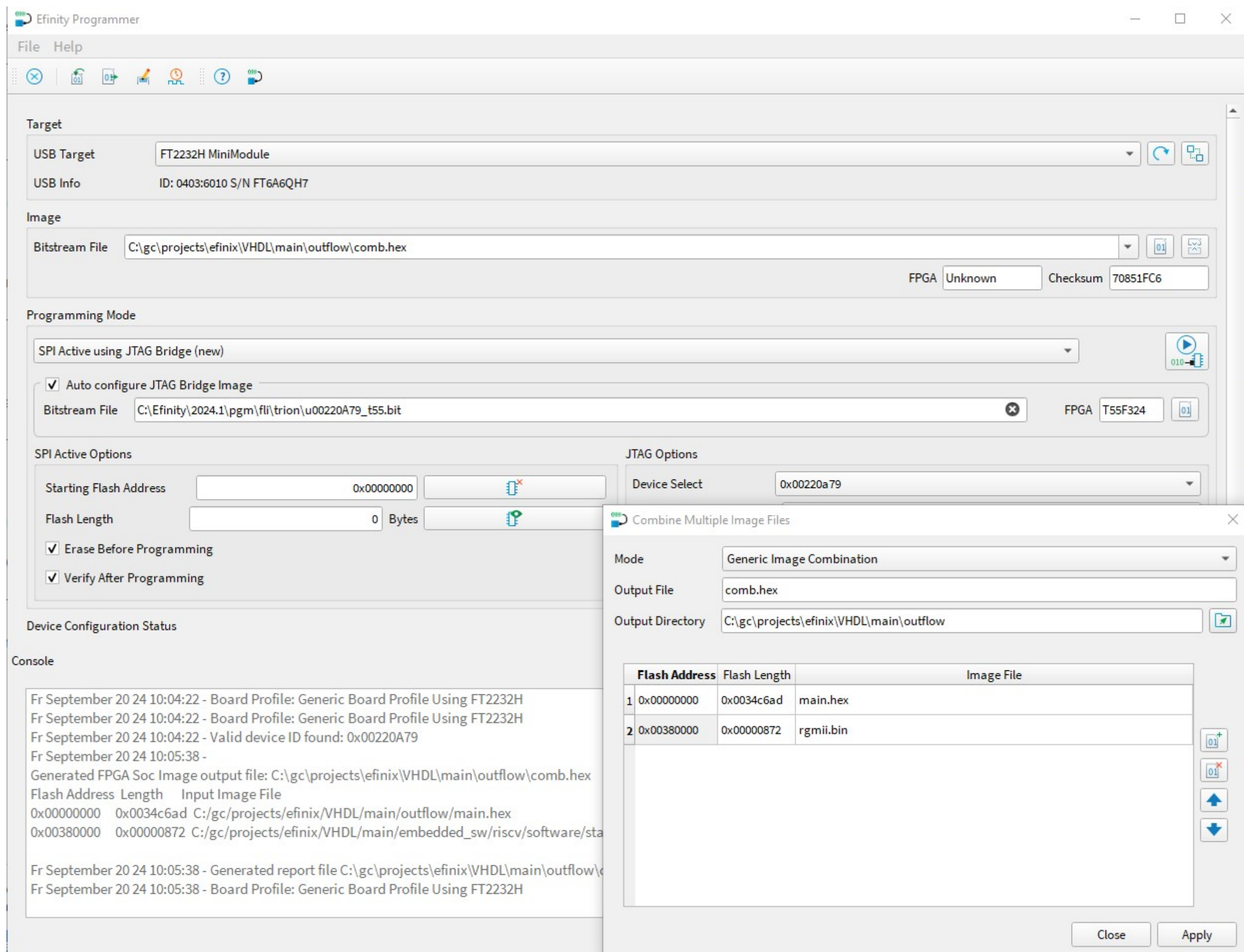


Fig6: Flashing SW and FPGA image

After the very first flashing we were able to debug via JTAG/openocd (according UG-RISCV-SAPPHIRE-v6.1) In ftdi.cfg in embedded_sw\riscv\bsp\efinix\EfxSapphireSoc\openocd, we had to comment: #ftdi device_desc “MTRION55EVK” (which was derived from the Sapphire IP configuration under Debug Custom Target Board and troubled).

For HW debugging we used Efinix Debug Wizard with the logic analyzer core.

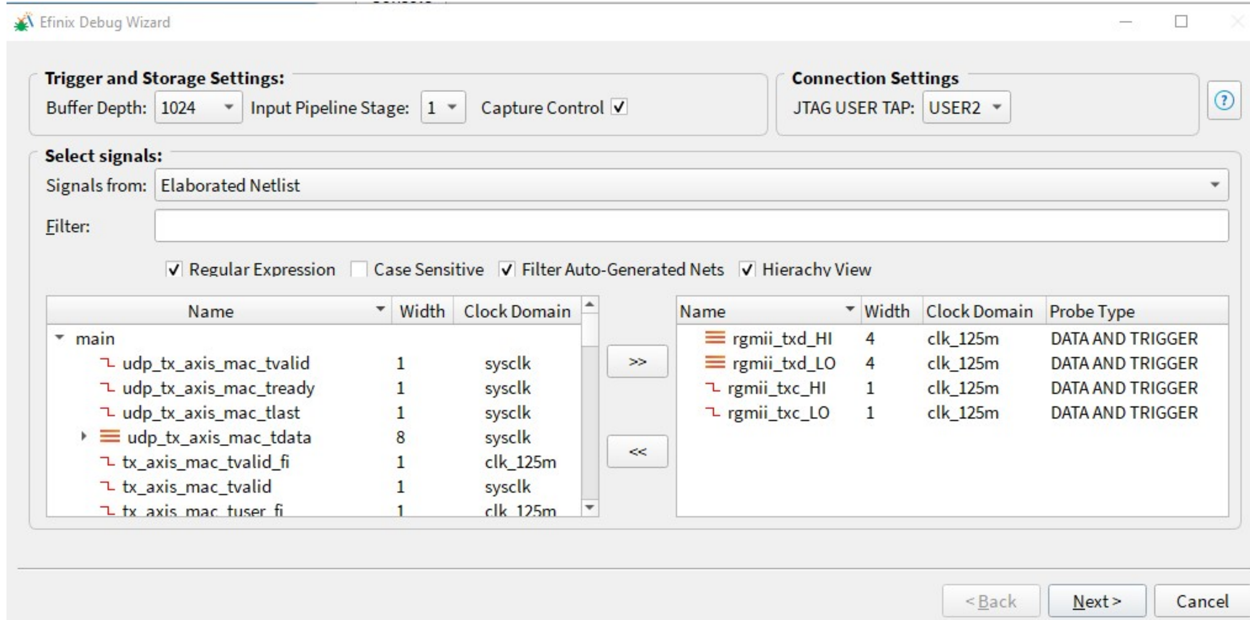


Fig7: Debug Wizard

Like in the example design we tested the transmit direction receiving MAC frames generated in the pattern generator of our RTL test design using wireshark on Ubunutu.

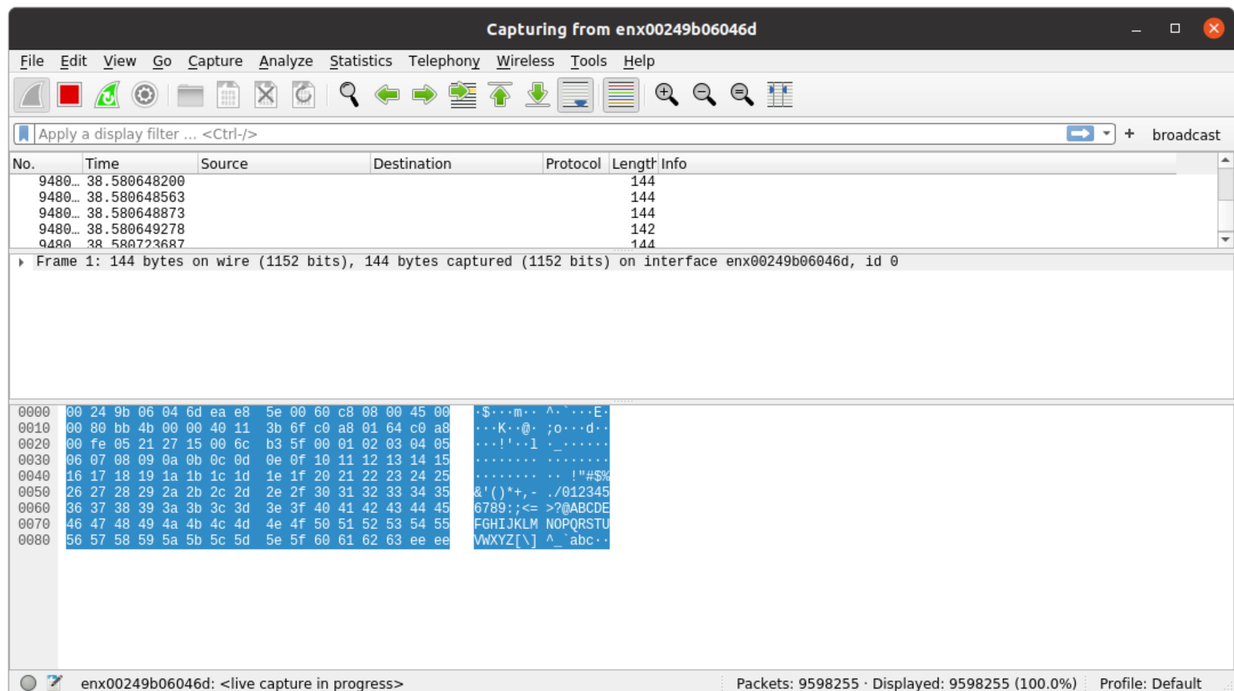


Fig8: Wireshark Test for Transmit Direction

Receive direction was tested switching in the RiscV SW to test mode 1 (tseDemo.h: #define TEST_MODE 1). Sending a broadcast using Packet Sender on Ubuntu we were able to receive the frame which was routed back in our RTL design by Wireshark.

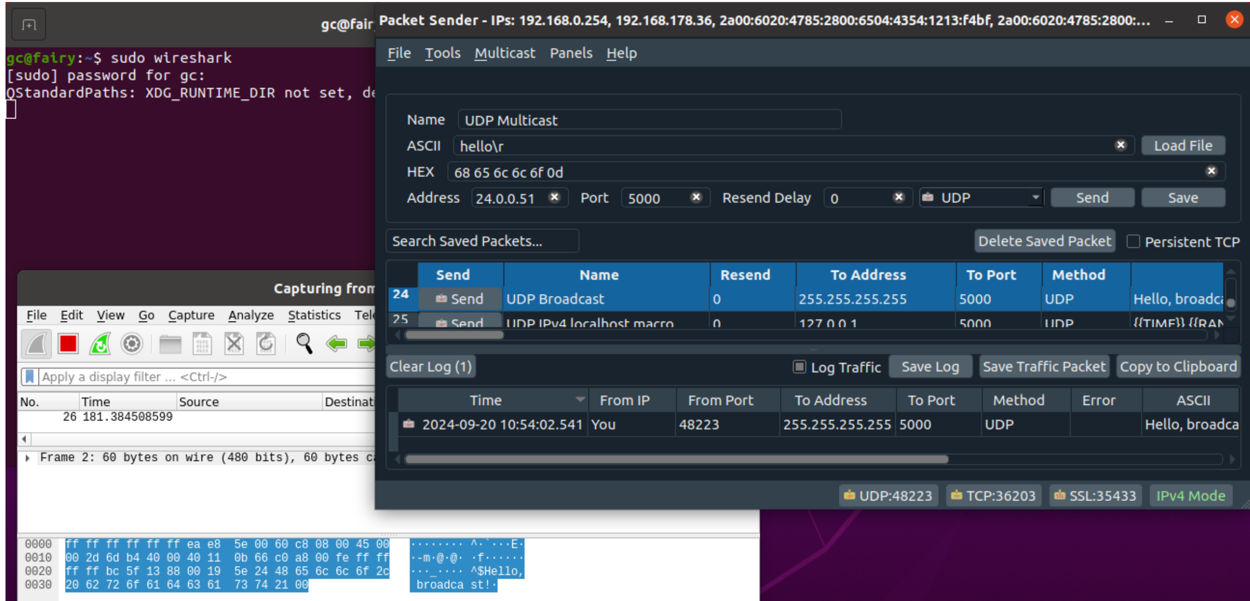


Fig9: Wireshark Test for Receive Direction