Toyon Research Corporation

Lab 9: QPSK Receiver/Transmitter

Chilipepper Tutorial Projects

Version 1.0
1/28/2014

Table of Contents

Lab 9: QPSK Receiver/Transmitter

Introduction

This lab will extend the previous labs and allow you to combine your receive core designed in lab 8 with the transmitter designed in lab 3. The focus of this combination will be on sending a message to one FPGA via terminal, sending this message to the Chilipepper for transmission, receiving the transmitted QPSK waveform using a second Chilipepper board, and outputting this message directly to another terminal as done in the previous lab. Both the transmit and receive core created in the previous labs will be used in this lab to create a single EDK project, and therefore prior knowledge of the workings of HDL Coder as well as the Xilinx EDK environment is assumed. It is recommended that you complete the previous labs before completing this lab.

This lab is created using:

- MATLAB 2014a
- Xilinx ISE Design Suite 14.7
- Windows 7, 64-bit

Procedure

This lab is organized into a series of steps, each including general instructions and supplementary steps, allowing you to take advantage of the lab according to your experience level.

This lab consists of the following basic steps:

- Generate HDL code from MATLAB functions
- Generate an IP core using MATLAB HDL Coder
- Configure your created PCores and export the design into SDK
- Create software to run your design
- Test and verify your results

Objectives

After completing this lab, you will be able to:

- Send and Receive a QPSK Waveform using the Chilipepper FMC
- Send to and receive from the MicroBlaze Processor Serial Port
- Create a software application to test your design
- Verify your results using a standard terminal and ChipScope

Generate HDL code Step 1

This section will show you how to create your MATLAB function and test bench files which are required to export your design into EDK.

1.1 PCores

This Lab will combine each of the following previously created PCores into the design.

- `ADC Driver`
- `DAC Driver`
- `DC Offset`
- `MCU Driver`
- `QPSK RX Core`
- `QPSK TX Core`
- `RX FIFO`
- `TX FIFO`

Since the cores have already been created, we can copy them into our EDK project without having to create any HDL Coder projects.

1. Make a **new EDK project** using the same method used in the previous labs. Be sure that you **import** the correct system configuration file.
2. Once the project is created, copy the previously created cores into your EDK pcores folder. Then simply select project -> **rescan user repositories** to show your newly added user PCores within your EDK project.

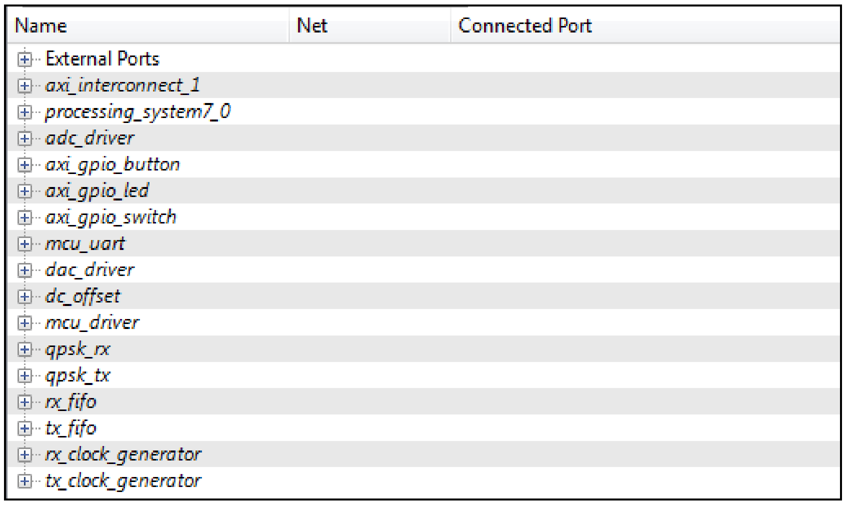
Configure Cores and Export Design Step 2

This section will show you how to integrate your PCores into your FPGA design using EDK. There are several components that must be configured for the design of this project. A quick list of the cores needed is given below. Refer to lab 0 sections 4.3 and 5.1 for information on how to add cores to the design.

2.1 Needed IP Cores

- ADC Driver
- DAC Driver
- MCU Driver
- MCU UART
- DC Offset
- QPSK RX
- QPSK TX
- RX FIFO
- TX FIFO
- Clock Generator (one for RX and one for TX)
- Several GPIO cores
- Processing System
- AXI Interconnect

In addition, several of these cores will require external ports. Be sure that you have access to modifying the external port settings. Refer to Figure 2-1 Below.



Name	Net	Connected Port
⊞ External Ports		
⊞ *axi_interconnect_1*		
⊞ *processing_system7_0*		
⊞ *adc_driver*		
⊞ *axi_gpio_button*		
⊞ *axi_gpio_led*		
⊞ *axi_gpio_switch*		
⊞ *mcu_uart*		
⊞ *dac_driver*		
⊞ *dc_offset*		
⊞ *mcu_driver*		
⊞ *qpsk_rx*		
⊞ *qpsk_tx*		
⊞ *rx_fifo*		
⊞ *tx_fifo*		
⊞ *rx_clock_generator*		
⊞ *tx_clock_generator*		

Figure 2-1: EDK project ports list

2.2 Configuring the Pcore Ports

Each of the ports used in this lab can be configured exactly as the previous labs design. Refer to Lab 8 (RX) and lab 3 (TX) for more information on individual port configuration.

2.3 Configuring the GPIO Ports

There are several GPIO cores that are used in this lab to help with program verification. The required cores and their configurations are given below.

1) Add a GPIO core for initiating packet transmission via button press. Configure it as Channel 1 only, with a bit width of just 1 bit. Call it axi_gpio_button or something similar. Expand the port, right click the GPIO_IO_I input pin, and select "Make External". Configure the others as "no connection".

2) This lab will have two general purpose LEDs that can be used for program debugging if necessary. Add another GPIO core and configure it as 1 bit for both Channel one and Channel two. Expand the port, right click the GPIO_IO and GPIO2_IO pins, and select "Make External". Configure the others as "no connection".

3) The last GPIO used for this lab is for changing the mode of the demo from continuous transmit to button transmit to normal operation etc. Configure this core as a single channel with a width of 2 bits. Call it axi_gpio_switch or something similar.

Once all of your cores have been added, your design should look similar to Figure 2-1 shown above.

2.4 Configuring the TX Clock Generator IP Core

The TX Clock Generator is used in this project to distribute the appropriate clock signals to each of the PCores required for Chilipepper initialization, the TX processing chain, and any external hardware which may require a clock signal. For this project, the TX Clock Generator is sourced from the 40 MHz `pll_clk_out` on the Chilipepper radio board (as described in the **Chilipepper user's guide**). This signal is then distributed to 6 other devices; 4 PCores (MCU Driver, qpsk_tx, tx_fifo, dac_driver) and the `TX_CLK` and `RX_CLK` signals. The TX and RX clock signals are used to latch data from the TXD and RXD lines to the DAC and ADC respectively on the radio board. For this lab, the Clock Generator has been named tx_clock_generator.

1. **Double click** the Clock Generator PCore and **configure** the settings as follows

- Input Clock Frequency of **40Mhz**
- CLKOUT0 Required Frequency of **20MHz**, 0 Phase, **PLLE0** group and **Buffered True**
- CLKOUT1 Required Frequency of **40MHz**, 0 Phase, **PLLE0** group and **Buffered True**

- CLKOUT2 Required Frequency of **40Mhz**, 180 Phase, **PLLE0** group and **Buffered True**
- CLKOUT3 Required Frequency of **40Mhz**, 0 Phase, **PLLE0** group and **Buffered True**

Now that the settings are configured you should have several clocks in your clock generator list. Notice that the clock generator locked pin is left unconnected within this design.

2. **Connect** the pins according to the following.

- CLKIN $\longrightarrow$ External Ports
- CLKOUT0 $\longrightarrow$ mcu:: IPCORE_CLK, qpsk_tx::IPCORE_CLK and tx_fifo::IPCORE_CLK
- CLKOUT1 $\longrightarrow$ dac_driver::IPCORE_CLK
- CLKOUT2 $\longrightarrow$ External Ports
- CLKOUT2 $\longrightarrow$ External Ports
- RST $\longrightarrow$ net_gnd
- LOCKED $\longrightarrow$ **No Connection**

2.5 Configuring the RX Clock Generator IP Core

In addition to the TX Clock Generator, another clock generator is required for this design. As mentioned in Lab 2 and the Chilipepper User's Guide, the receiver chain is to be clocked using the RX return clock on the Chilipepper board to ensure data is latched properly from the ADC. In this design, there are four cores which must be clocked using the RX return clock; therefore a new clock generator called rx_clock_generator is used to distribute the clock signal.

1. **Double click** the Clock Generator PCore and **configure** the settings as follows

- Input Clock Frequency of **40Mhz**
- CLKOUT0 Required Frequency of **40MHz**, 180 Phase, **PLLE0** group and **Buffered True**
- CLKOUT1 Required Frequency of **20MHz**, 180 Phase, **PLLE0** group and **Buffered True**

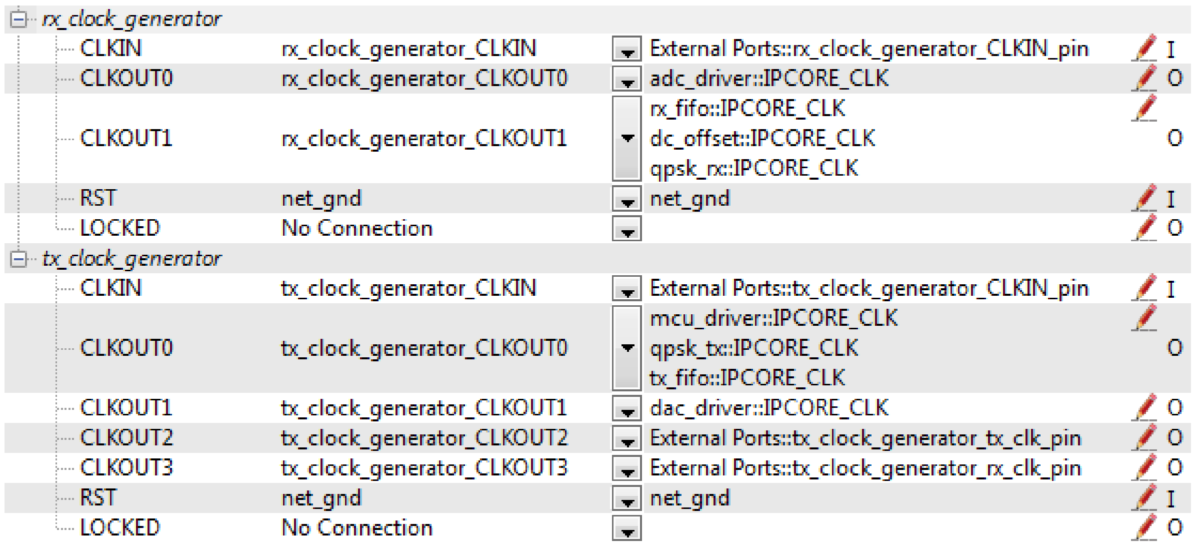
Now that the settings are configured you should have several clocks in your clock generator list.

2. **Connect** the pins according to the following.

- CLKIN $\longrightarrow$ External Ports

- CLKOUT0 ⟶ adc_driver::IPCORE_CLK
- CLKOUT1 ⟶ dc_offset:: IPCORE_CLK, qpsk_rx::IPCORE_CLK and rx_fifo::IPCORE_CLK
- RST ⟶ net_gnd
- LOCKED ⟶ **No Connection**

Your Clock Generator ports should look similar to Figure 2-2 below.



rx_clock_generator			
CLKIN	rx_clock_generator_CLKIN	External Ports::rx_clock_generator_CLKIN_pin	I
CLKOUT0	rx_clock_generator_CLKOUT0	adc_driver::IPCORE_CLK	O
CLKOUT1	rx_clock_generator_CLKOUT1	rx_fifo::IPCORE_CLK dc_offset::IPCORE_CLK qpsk_rx::IPCORE_CLK	O
RST	net_gnd	net_gnd	I
LOCKED	No Connection		O
tx_clock_generator			
CLKIN	tx_clock_generator_CLKIN	External Ports::tx_clock_generator_CLKIN_pin	I
CLKOUT0	tx_clock_generator_CLKOUT0	mcu_driver::IPCORE_CLK qpsk_tx::IPCORE_CLK tx_fifo::IPCORE_CLK	O
CLKOUT1	tx_clock_generator_CLKOUT1	dac_driver::IPCORE_CLK	O
CLKOUT2	tx_clock_generator_CLKOUT2	External Ports::tx_clock_generator_tx_clk_pin	O
CLKOUT3	tx_clock_generator_CLKOUT3	External Ports::tx_clock_generator_rx_clk_pin	O
RST	net_gnd	net_gnd	I
LOCKED	No Connection		O

Figure 2-2: Clock Generator port configurations

Be sure your External Port pins, as well as your PCores match the names shown in the figures above.

2.6 Pin Assignments

Once the clock generator is configured correctly, the `IPCORE_CLK` for the other cores should be set as well. The next step is to setup the **pin assignments** for the external ports.

1. Open the **Project** tab.
2. Double-click on the **UCF File: data\system.ucf** from this panel, to open the constraints file.
3. Fill in the pin out information for your design using Appendix A as a reference.

2.7 Adding ChipScope Peripheral (optional)

Given that the output of this design will be sent directly out to a UART port, there is no need for verification using ChipScope. However, the ChipScope output may provide valuable debugging information if the design is not functioning properly. It is therefore recommended that several key components of the design such as the tx and rx FIFO cores be sent to ChipScope for debugging purposes.

1. Select Debug -> **Debug Configuration** from the top menu.
2. Click the **Add ChipScope Peripheral** button on the bottom left hand side of the screen.
3. Select To **monitor arbitrary system level signals** (middle option) from the list.
4. Add some of the outputs of the design to the ChipScope ports. If you add too many ports, your design may not pass timing, so only add the ones which will be most useful for debugging.
5. Click ok to finish configuration of your ChipScope peripheral.

Once completed, you're ready to generate your bitstream file! Select the Export Design button from the navigator window on the left. Click the Export and Launch SDK button. This process may take awhile.

Create software project Step 3

Once the design is compiled and exported, you'll be greeted with a screen asking you where you would like to store your software project. It is very helpful to create the SDK folder in the same directory as your MATLAB and EDK folders. Doing this will keep all relevant files in the same location.

3.1 Creating a new C Project

This section will show you how to create a C program to test your QPSK RX project.

1. Select **File → New → Application Project**.
2. Name the project "qpsk_example" or something similar and leave the other settings at their defaults. Click next.
3. On the next screen, be sure to select **Hello World** from the list of Available Templates.
4. Click **Finish**. You should now see your qpsk_example project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the qpsk_example project folder, and into the src folder, you should see a `helloworld.c` file. Feel free to rename this file to main.c or something more appropriate.
6. **Double click** the file to open it and **replace** all of its contents with the code in Appendix B.
7. **Download** the **Chilipepper.c** and **Chilipepper.h** files from the GitHub repository[1] if you don't already have them. Copy them into the source directory with your main.c file.
8. Open the Chilipepper.c file and modify it for this lab. **All** PCores should be defined at the top of the file for this design.

Note	You may be required to add the Math Library to the project to define the pow function used in the Chilipepper.c Library file. If so, follow the optional step 9 listed below.

9. (Optional) Click on **Project →Properties.** Open the **C/C++ Build** arrow and click the settings option. Under **ARM gcc linker**, click the Libraries folder. Click the button, type the letter **m** into the prompt and select ok. **Apply** and hit ok.

[1] Can be found at https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/ChilipepperSupport/Library%20Files

3.2 Programming the Board

Once your program is written and compiled you are ready to test the design! This is done by programming the FPGA with your hardware descriptions defined in the bit file generated in EDK, and running your software on top of this design.

1. Connect the Chilipepper to the FPGA board and verify all cables are connected properly and the jumper settings are correct. Verify this by using the *Chilipepper Getting Started Guide*[2] as a reference. Also See Lab 0 for details on Jumper Configuration.

2. Once the FPGA and radio board are connected correctly, turn on the board.

3. Open iMPACT in the ISE Design tools.

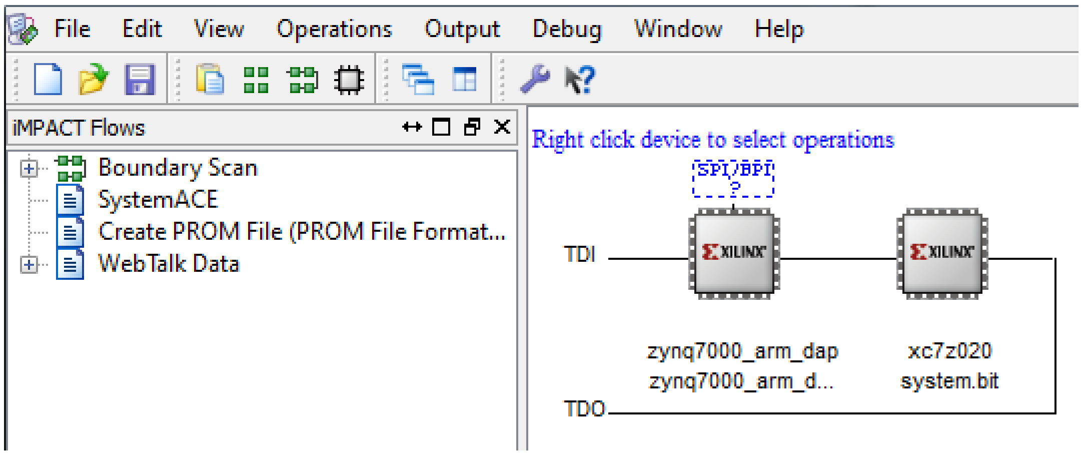
4. Select no if Impact asks you to load the last saved project.

5. Select yes to allow iMPACT to automatically create a new project for you. If you receive any connection errors, verify your USB or JTAG programmer cables are connected properly.

6. Select the Automatic option for the JTAG boundary scan setting and click ok.

7. Hit yes to assign configuration files. Bypass the first file selection, but for the second selection, browse to the location of your system.bit file. It should be inside the "Implementation" folder of your EDK project folder.

8. Select ok on the next screen verifying that the board displayed is your Zynq xc7z020 board. It should look similar to Figure 3-2 below.



3-2: configuration for Zed Board System.bit file

9. Right click on the xc7z020 board icon (should be on the right), select program and hit ok.

[2] Can be found at https://github.com/Toyon/Chilipepper/tree/master/QPSK_Radio/DemoFilesAndDocumentation

Note	If you are running this lab from two separate PCs, you will need to repeat this process for the second board using the Lab 9 system.bit file. Alternatively, you can run Lab 9 directly from the SD card by loading a standard SD card with the Boot.bin file for lab 9, which can be found on the github repo.

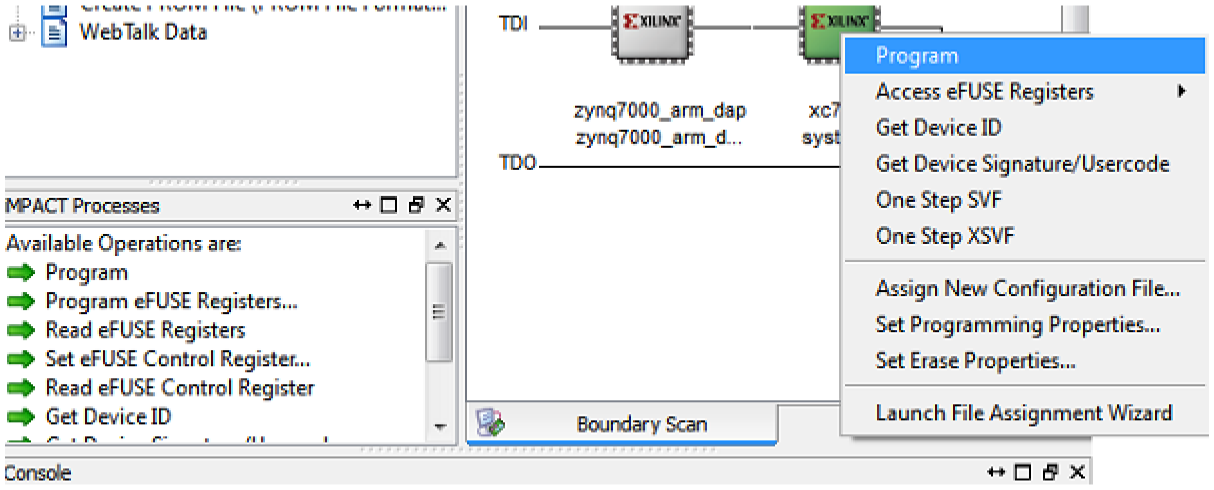

WebTalk Data
TDI
TDO
zynq7000_arm_dap
zynq7000_arm_d...
xc7
syst
MPACT Processes
Available Operations are:
Program
Program eFUSE Registers...
Read eFUSE Registers
Set eFUSE Control Register...
Read eFUSE Control Register
Get Device ID
Program
Access eFUSE Registers
Get Device ID
Get Device Signature/Usercode
One Step SVF
One Step XSVF
Assign New Configuration File...
Set Programming Properties...
Set Erase Properties...
Launch File Assignment Wizard
Boundary Scan
Console

Figure 3-3: iMPACT configuration screen

To load Lab 9 via SD card:

1. Place the file on the SD card, and place the card inside the SD slot of the FPGA.
2. Configure the jumpers on the FPGA as shown in Figure 4-3.
3. Turn on the board, and the program should load after about 15 seconds. Check for the blue light, indicated the load was successful.

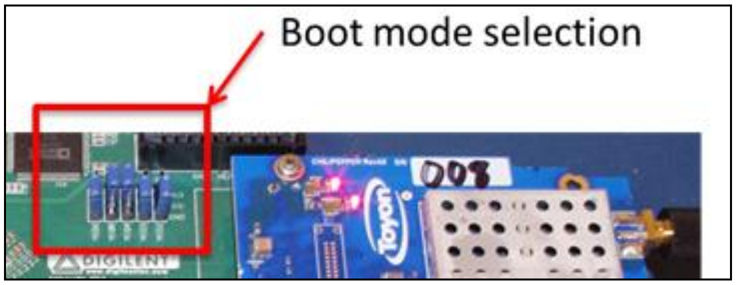


Figure 4-3: Jumper configuration needed to load a project via SD card

3.3 Debugging with SDK

If the hardware design is correct, you should see a blue light on the ZED Board indicating the program was successful. You can now return to the SDK project screen to test your software.

1. Test it by **right clicking** the `qpsk_rx` project folder and selecting **Debug As → Launch on Hardware (GDB)**.

2. You should now be taken to a screen which shows the `init_platform()` function as highlighted. You can now start the software program by clicking the **play** button in the top menu.

If the software initialization worked, you should see a green light on the Chilipepper, as well as the Blinking LEDs on the FPGA from the PCore blinky pins.

Testing and Design Verification Step 4

4.1 Verification with Terminal

Once you have the lab running successfully on both boards, the next step is to verify functionality by connecting both FPGAs to a terminal to view the received QPSK packets.

1. Connect both FPGAs to the PC using a micro USB cable. The cable should be plugged into the UART port on the FPGA, shown in Figure 4-1 below.

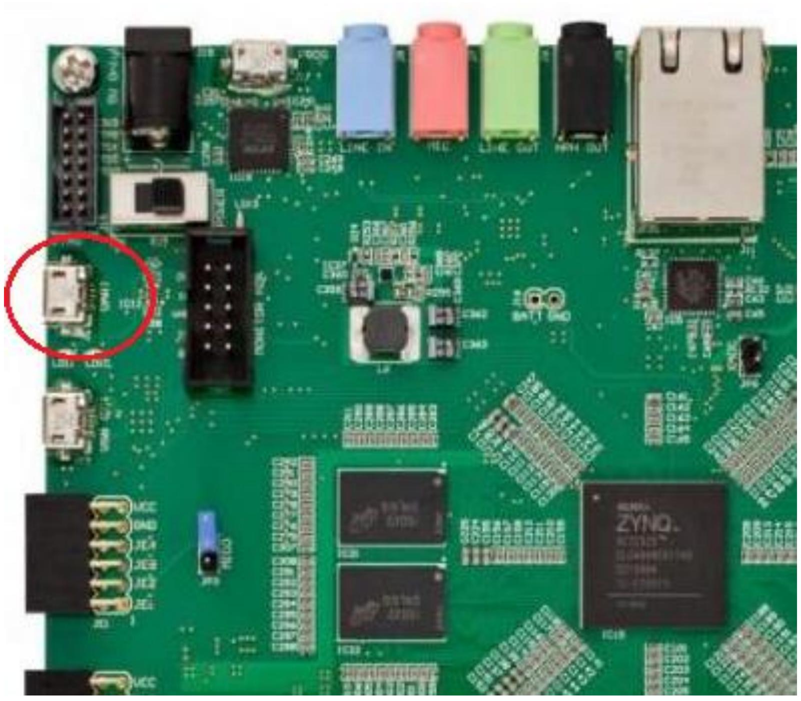


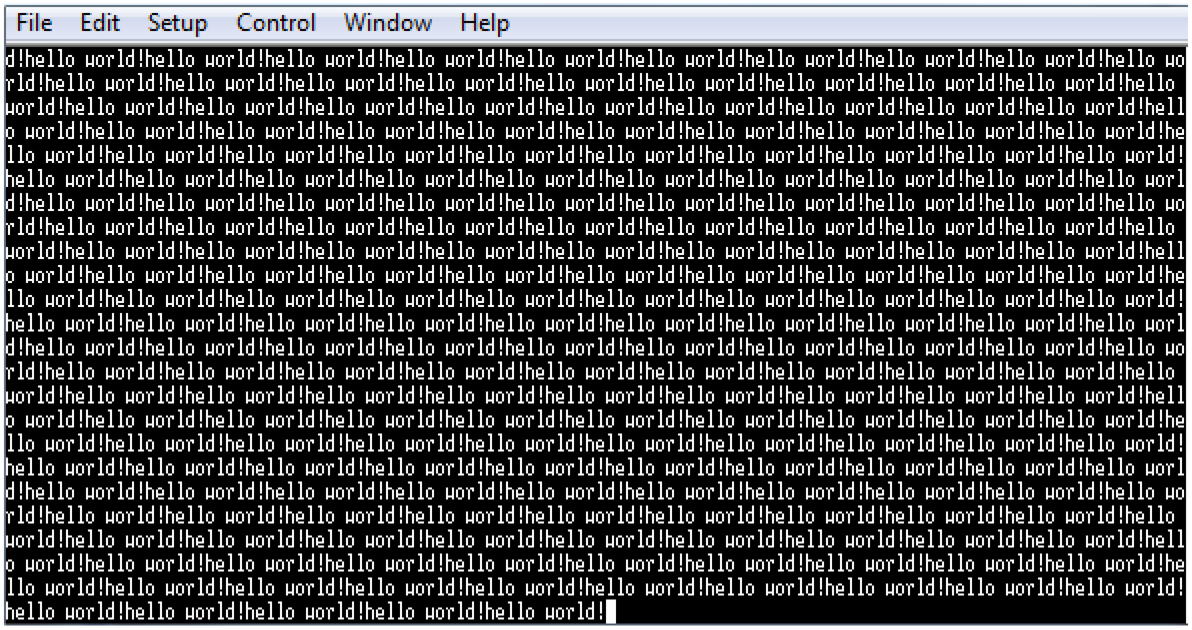
Figure 4-1: Circled in this figure is the UART port of the Xilinx Zed Board FPGA.

2. With the boards powered on, open up two hyper terminal windows usinga program such as Tera Term, and configure them with the following settings

   Baud: 115200; Data: 8 bit; No Parity;1 Stop bit; No Flow Control.

   Notice that the baud rate used is configured in the `SetupPeripherals` function in our main.c function created earlier. The other settings are all defaults for the XUartPs port.

3. Once the terminals are configured, you should be able to view your sent packets by typing directly into the terminal of the other board. Be sure that both boards have their switches configured for normal operation (mode 0).



```
d!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello wo
rld!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello 
world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hell
o world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!he
llo world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!
hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello worl
d!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello wo
rld!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello 
world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hell
o world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!he
llo world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!
hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello worl
d!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello wo
rld!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello 
world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hell
o world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!he
llo world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!
hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello worl
d!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello wo
rld!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello 
world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hell
o world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!he
llo world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello world!
hello world!hello world!hello world!hello world!hello world!
```

Appendix A UCF file

EDK UCF file `system.ucf`

```
################################################ PL clocks and reset ##########################################
NET tx_clock_generator_CLKIN_pin              LOC = D18   | IOSTANDARD = LVCMOS25;
NET tx_clock_generator_CLKIN_pin              TNM_NET = tx_clock_generator_CLKIN;
TIMESPEC TS_tx_clock_generator_CLKIN = PERIOD tx_clock_generator_CLKIN 40.000 MHz;
##############################################################################################################
NET rx_clock_generator_CLKIN_pin              LOC = L18   | IOSTANDARD = LVCMOS25;
NET rx_clock_generator_CLKIN_pin              TNM_NET = rx_clock_generator_CLKIN;
TIMESPEC TS_rx_clock_generator_CLKIN = PERIOD rx_clock_generator_CLKIN 40.000 MHz;
############################################Chilipepper Rx and Tx clock lines####################################
NET tx_clock_generator_tx_clk_pin             LOC = C17         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET tx_clock_generator_rx_clk_pin             LOC = J18         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
##############################################Rx – FMC interface at 2.5V #######################################
NET adc_driver_rx_iq_sel_pin                  LOC = N19         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[0]                     LOC =M21          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[1]                     LOC = J21         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[2]                     LOC = M22         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[3]                     LOC = J22         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[4]                     LOC = T16         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[5]                     LOC = P20         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[6]                     LOC = T17         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[7]                     LOC = N17         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[8]                     LOC = J20         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[9]                     LOC = P21         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[10]                    LOC = N18         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[11]                    LOC = J16         | IOSTANDARD = LVCMOS25;
##############################################Tx – FMC interface at 2.5V #######################################
NET dac_driver_tx_iq_sel_pin                  LOC = B16         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[0]                     LOC = A18         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[1]                     LOC = A19         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[2]                     LOC = E20         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[3]                     LOC = G21         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[4]                     LOC = F19         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[5]                     LOC = G15         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[6]                     LOC = E19         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[7]                     LOC = G16         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[8]                     LOC = G19         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[9]                     LOC = A16         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[10]                    LOC = A17         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_txd_pin[11]                    LOC = C18         | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
```

```
#################################################### MCU Interface ######################################################
NET mcu_uart_RX_pin                        LOC = R19       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_uart_TX_pin                        LOC = L21       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_mcu_reset_out_pin           LOC = K20       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_tx_en_pin                   LOC = D22       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_tr_sw_pin                   LOC = D20       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_rx_en_pin                   LOC = C22       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_pa_en_pin                   LOC = E21       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_init_done_pin               LOC = K19       | IOSTANDARD = LVCMOS25;
######################################################### LEDs ##########################################################
NET axi_gpio_led_GPIO_IO_pin               LOC = T22       | IOSTANDARD =LVCMOS33; # "LD0"
NET axi_gpio_led_GPIO2_IO_pin              LOC = T21       | IOSTANDARD =LVCMOS33; # "LD1"
NET adc_driver_blinky_pin                  LOC = U22       | IOSTANDARD =LVCMOS33; # "LD2"
NET dac_driver_blinky_pin                  LOC = U21       | IOSTANDARD =LVCMOS33; # "LD3"
NET mcu_driver_blinky_pin                  LOC = V22       | IOSTANDARD =LVCMOS33; # "LD4"
NET dc_offset_blinky_pin                   LOC = W22       | IOSTANDARD =LVCMOS33; # "LD5"
NET qpsk_rx_blinky_pin                     LOC = U19       | IOSTANDARD =LVCMOS33; # "LD6"
NET qpsk_tx_blinky_pin                     LOC = U14       | IOSTANDARD =LVCMOS33; # "LD7"
#######################################################Switches ########################################################
NET axi_gpio_switch_GPIO_IO_I_pin[0]       LOC = F22       | IOSTANDARD =LVCMOS25;  # "SW0"
NET axi_gpio_switch_GPIO_IO_I_pin[1]       LOC = G22       | IOSTANDARD =LVCMOS25;  # "SW1"
#########################################################Buttons ########################################################
NET axi_gpio_button_GPIO_IO_I_pin          LOC = P16       | IOSTANDARD =LVCMOS25; # "BTCenter"
```

Appendix B main.c

SDK function `main.c`

```
#include <stdio.h>
#include "platform.h"
#include "xgpio.h"
#include "chilipepper.h"
#include "xuartps.h"

XGpio gpio_blinky, gpio_sw_test, gpio_btn;
XUartPs uartPs;
XUartPs_Config *pUartPsConfig;

int DebouncButton( void );
int SetupPeripherals( void );

int main()
{
	int sentCount;
	int aliveLed = 0, statusLed = 0;;
	int numBytes;
	int sw, i1;
	static int BlinkCount;
	int txCount = 0, txTryCount = 0;
	unsigned char numUartRead, curValue, id;
	unsigned char rxBuf[256], txBuf[256];

	init_platform();

	if(SetupPeripherals() != XST_SUCCESS)
		return -1;

	if ( Chilipepper_Initialize() != 0 )
		return -1;

	// by default we are in receive
	Chilipepper_SetPA( 1 );
	Chilipepper_SetTxRxSw( 1 ); // 0- transmit, 1-receive
	Chilipepper_SetDCOC(1);     // enable dc offset correction

	// enable the Chilipepper LED to indicate we are operational
	Chilipepper_SetLed( 1 );

	xil_printf("\r\n\r\nWelcome to Toyon's Chilipepper QPSK demo. This demo was 
written in MATLAB using Mathworks HDL Coder.\r\n\r\n");
	while (1)
	{
		Chilipepper_ControlAgc();
		sw = XGpio_DiscreteRead(&gpio_sw_test, 1);
		switch (sw)
		{
        case 0: // normal operation
        // main priority is to parse OTA packets
                numBytes = Chilipepper_ReadPacket( rxBuf, &id );

                // This is a normal receive situation.
                // We get a packet, write it to UART.
                if (numBytes > 0)
                {
                        sentCount = 0;
                        while (sentCount < numBytes)
                        {
                                curValue = rxBuf[sentCount+4];
                                sentCount += XUartPs_Send(&uartPs, &curValue, 1);
                        }
                        statusLed = ~statusLed;
                        XGpio_DiscreteWrite(&gpio_blinky, 1, statusLed);
                }

                do
                {
                        numUartRead = XUartPs_Recv(&uartPs, &txBuf[txCount+4], 1);
                        if (numUartRead == 1)
                                txCount++;
                } while(numUartRead == 1);

                // only attempt to send something if we have something to send
                if (txCount > 0)
                {
                        if (txCount >= 10 || txTryCount > 100000)
                        {
                                Chilipepper_WritePacket( txBuf, txCount, 1 );
                                txCount = 0;
                                txTryCount = 0;
                        }
                }
                txTryCount++;
                break;
        case 1: // continuously send out packets
                // do it once and then stall for a bit
                for (i1=0; i1<5000; i1++)
                {
                        if (i1 == 0)
                        {
                                Chilipepper_WriteTestPacket( 1 );
                                statusLed = ~statusLed;
                                XGpio_DiscreteWrite(&gpio_blinky, 1, statusLed);
                        }
                }
                break;
```

```
            case 2: // initiate packet transmission with a button press
                if (DebouncButton() == 0)
                    break;

                Chilipepper_WriteTestPacket( 1 );
                statusLed = ~statusLed;
                XGpio_DiscreteWrite(&gpio_blinky, 1, statusLed);
                break;
            default:
                break;
            }
            // flip the LED1 so the user knows the processor is alive
            BlinkCount += 1;
            if (BlinkCount > 200000)
            {
                aliveLed = ~aliveLed;
                BlinkCount = 1;
                XGpio_DiscreteWrite(&gpio_blinky, 2, aliveLed);
            }
        }
        cleanup_platform();
        return 0;
}

int DebouncButton( void )
{
        int btn;
        static int hitZero=0;
        static int btnIntegrator=0;

        btn = XGpio_DiscreteRead(&gpio_btn, 1);

        // decrement and keep track if we've touched zero
        if ( btn==0 )
        {
            if (btnIntegrator > 0)
                btnIntegrator -= 1;
            if (btnIntegrator == 0)
                hitZero = 1;
            return 0;
        }
        if (btnIntegrator < 1000)
            btnIntegrator += 1;

        if (btnIntegrator < 1000)
            return 0;

        if (hitZero == 0)
            return 0;

        // we've hit 1000 so now we know we need to hit zero again
        hitZero = 0;

        return 1;
}
```

```
int SetupPeripherals( void )
{
	int status;

	XGpio_Initialize(&gpio_blinky, XPAR_AXI_GPIO_LED_DEVICE_ID);
	XGpio_SetDataDirection(&gpio_blinky, 2, 0);
	XGpio_SetDataDirection(&gpio_blinky, 1, 0);
	XGpio_DiscreteWrite(&gpio_blinky, 1, 0);
	XGpio_DiscreteWrite(&gpio_blinky, 2, 0);

	XGpio_Initialize(&gpio_sw_test, XPAR_AXI_GPIO_SWITCH_DEVICE_ID);
	XGpio_SetDataDirection(&gpio_sw_test, 1, 3);

	XGpio_Initialize(&gpio_btn, XPAR_AXI_GPIO_BUTTON_DEVICE_ID);
	XGpio_SetDataDirection(&gpio_btn, 1, 1);

	pUartPsConfig = XUartPs_LookupConfig(XPAR_PS7_UART_1_DEVICE_ID);
	if (NULL == pUartPsConfig) {
		return XST_FAILURE;
	}
	status = XUartPs_CfgInitialize(&uartPs, pUartPsConfig, pUartPsConfig-
>BaseAddress);
	if (status != XST_SUCCESS) {
		return XST_FAILURE;
	}
	XUartPs_SetBaudRate(&uartPs, 115200);

	return XST_SUCCESS;
}
```