|  |
| --- |
| Toyon Research Corporation |
| Lab 3: Output QPSK |
| Chilipepper Tutorial Projects |

|  |
| --- |
| Version 1.2  1/10/2014 |

Table of Contents

[Introduction 4](#_Toc377131942)

[Procedure 4](#_Toc377131943)

[Objectives 4](#_Toc377131944)

[Generate HDL code 5](#_Toc377131945)

[1.1 DAC and MCU Driver MATLAB Files 5](#_Toc377131946)

[1.2 QPSK TX MATLAB Functions 5](#_Toc377131947)

[1.3 TX\_FIFO MATLAB Function 9](#_Toc377131948)

[1.4 QPSK TX Test Bench Scripts 10](#_Toc377131949)

[1.5 HDL Coder Project 11](#_Toc377131950)

[Configure Cores and Export Design 17](#_Toc377131951)

[2.1 Needed IP Cores 17](#_Toc377131953)

[2.2 Configuring the DAC Driver Port 18](#_Toc377131954)

[2.3 Configuring the MCU Driver Port 18](#_Toc377131955)

[2.4 Configuring the MCU UART 18](#_Toc377131956)

[2.5 Configuring the QPSK TX 18](#_Toc377131957)

[2.6 Configuring the TX FIFO 19](#_Toc377131958)

[2.7 Configuring the LEDs GPIO Port 19](#_Toc377131959)

[2.8 Configuring the Switch GPIO Port 19](#_Toc377131960)

[2.9 Configuring the Button GPIO Port 19](#_Toc377131961)

[2.10 Configuring the Clock Generator IP Core 20](#_Toc377131962)

[2.11 Pin Assignments 21](#_Toc377131963)

[Create software project 23](#_Toc377131964)

[3.1 Creating a new C Project 23](#_Toc377131966)

[3.2 Programming the Board 24](#_Toc377131967)

[3.3 Debugging with SDK 25](#_Toc377131968)

[Testing and Design Verification 26](#_Toc377131969)

[4.1 Verification with another Chilipepper Board 26](#_Toc377131970)

[4.2 Verification with ChipScope 26](#_Toc377131971)

[4.3 MATLAB Analysis 28](#_Toc377131972)

[Appendix A MATLAB byte2sym Function 30](#_Toc377131973)

[Appendix B MATLAB TX FIFO Function 34](#_Toc377131974)

[Appendix C MATLAB Test Bench 36](#_Toc377131975)

[Appendix D SDK source file 39](#_Toc377131976)

Lab 3: Output QPSK

# Introduction

This lab will show you how to transmit a QPSK Waveform on an FPGA Mezzanine Card (FMC) radio board using the Xilinx Zed Board FPGA and the Toyon Chilipepper FMC. The Digital to Analog Conversion (DAC) used to transmit the signal will take place on the Chilipepper board. The Creation and Modulation of the waveform will be done using hardware PCores created in MATLAB HDL Coder. The FMC initialization and microcontroller (MCU) signal control will be handled in software using the Xilinx Software Development Kit (SDK). Finally, the testing of results will be done using either a second Chilipepper board, or ChipScope and MATLAB. This lab assumes prior knowledge of the workings of HDL Coder as well as the Xilinx EDK environment. It is recommended that you complete the previous labs before completing this lab.

This lab is created using:

* MATLAB 2013b
* 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:

* Modulate a signal using hardware on the FPGA
* Create a MATLAB Function to implement a FIFO buffer
* Transmit a QPSK Waveform using the Chilipepper FMC
* Create a software application to test your design
* Verify your results in ChipScope and analyze them using MATLAB

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 DAC and MCU Driver MATLAB Files

Just like the previous lab, we need an MCU driver to handle the control signals to and from the Chilipepper, and a DAC to interleave our signal before transmitting it. Since these PCores have already been created in the previous labs, we can simply use the same PCores for this lab as well. Refer to Lab 1 for information on how to create this PCore if needed.

## 1.2 QPSK TX MATLAB Functions

Your MATLAB functions will eventually become a core that will be synthesized into hardware. The algorithm describes the operations in each clock cycle, and processes data on a sample-by-sample basis. This lab requires several MATLAB functions all used in conjunction to generate and transmit the QPSK waveform. The first function used is shown in Figure 1-1.



Figure 1‑: MATLAB function to Create i and q channel outputs.

While this function does not play a huge role in creating the modulated signal, it calls other functions which do, the first of which is the function qpsk\_tx\_byte2sym.m which can be seen in **Appendix A**. The primary purpose of this function is to properly format binary data into symbols which can be transmitted using QPSK. Reviewing the function shows that there are several important aspects to the algorithm, some of which are mentioned below

1. Create a new **MATLAB function** with the contents of Appendix A.
2. The algorithm uses an internal TX\_FIFO buffer to store bytes waiting to be transmitted.
3. Initially, nothing is transmitted, and the algorithm simply fills its buffer with data needing to be transmitted.
4. Once all data has been received (indicated by the tx\_en\_in) the algorithm sends a string of overhead bits consisting of pad bits and header bits.
5. Lastly the algorithm sends the data in its buffer until the buffer is empty, and then activates the tx\_done output
6. Save this function as qpsk\_tx\_byte2sym.m inside the project directory.

There is a function called within the qpsk\_tx\_byte2sym which is used to select a bit to transmit. This function is called mybitget and as seen from the algorithm, it is called twice; once for the i channel and once for the q channel. These bits are then combined by creating a complex number in which i represents the real portion of the number and q the imaginary portion and eventually transmitted as a single waveform. By sending the i and q channel bits simultaneously in this way, the overall transmission requires only sending four two-bit sequences for any given byte. Each two-bit sequence can create one of four different signals, called symbols. The mybitget function is shown in Figure 1-2.



Figure 1‑2: MATLAB Function to select bits for QPSK Symbol creation.

QPSK takes advantage of these two-bit symbols, by using 1 bit to modulate a sine wave and the other bit to modulate a cosine wave. Before modulation, each zero bit is changed from a 0 to a negative 1 (also called a non-return to zero or NRZ signal). The 2 sinusoids are then combined into a single waveform just before being transmitted. The result of the modulation and combining the waveforms causes the phase of the signal to vary according to the table below. The resultant phase possibilities are shown in the right most table. In addition, a plot of each of the symbols on a complex plane (also called a scatter plot) can be seen in Figure 1-3.

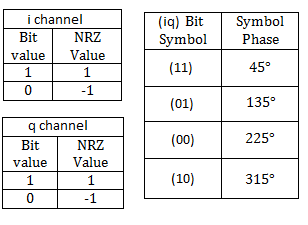


Table -1: The left most tables show NRZ bit values, while the rightmost table shows the symbol phase values.

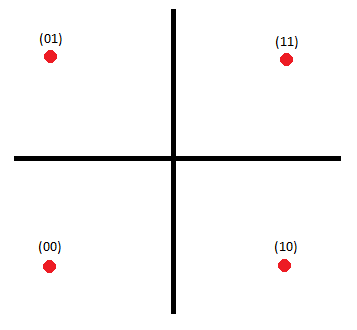


Figure 1‑3: Scatter plot of the QPSK symbols.

1. Create a new **MATLAB function** with the contents of Figure 1-2.
2. **Save** this function as mybitget.m inside the project directory.

The next function used to format the transmit data is called qpsk\_srrc. The purpose of this function is to apply a square-root-raised-cosine filter (SRRC) which is used for pulse shaping. This allows for the transmission of the QPSK waveform with a minimal amount of inter symbol interference. The MATLAB function is shown in Figure 1-4 below, and as you can see it uses a variable called SRRC to filter the buffer containing the symbol data. This variable is a look up table (LUT) that was created using the MATLAB function shown in Figure 1-5.



Figure 1‑4: MATLAB function used to filter the transmit data.



Figure 1‑5: MATLAB function to create square-root-raised-cosine filter.

Observing the function in Figure 1-5 reveals how the filter was created in MATLAB and what its parameters are: such as the order and in this case the transition band.

1. Create 2 new **MATLAB function** with the contents of Figure 1-4 and 1-5.
2. **Save** these functions as qpsk\_srrc.m and make\_srrc\_lut.m respectively inside the project directory.

In addition to the LUT required to filter the transmitted data, there are 2 additional LUTs which are used to send a header for the QPSK packet. This Header is most commonly used to assist with correctly receiving the transmitted packet. The two lookup files named TB\_i and TB\_q can be found on the GitHub website under the section for this Lab[[1]](#footnote-1).

1. Download a copy, or create new files for the TB\_i and TB\_q LUTs, and place them in your MATLAB directory.

## 1.3 TX\_FIFO MATLAB Function

In addition to the FIFO used internally within the qpsk\_tx\_byte2sym.m function, there is another FIFO used in this Lab. The purpose of this FIFO is to assist with the handshaking required when sending data from the SDK project to the HDL Pcore. The code for this FIFO is shown in Appendix B.

1. Create a new **MATLAB function** with the contents of Appendix B.
2. **Save** this function as tx\_fifo.m inside the project directory.

## 1.4 QPSK TX Test Bench Scripts

As in the previous labs, the Test Bench script is required for HDL generation, but also allows you to test the functionality of the MATLAB algorithms. The code used for this script is shown in **Appendix C** and serves as the test bench for both the qpsk\_tx core and the tx\_fifo core. Additionally, there is a variable called sim in the script which allows you to either load QPSK data from ChipScope or simulate a transmitted QPSK signal in MATLAB and analyze the results. Setting it to 1 simulates the waveform, 0 loads it from a ChipScope prn file.

1. Create a new **MATLAB script** with the contents of Appendix C.
2. **Save** this script as qpsk\_tb.m inside the project directory

One other MATLAB file which must be created before executing the test bench is the CreateAppend16BitCRC function. This function is shown in Figure 1-7 and its purpose is to append the 16Bit CRC to the end of the QPSK packet when simulating the packet creation within the test bench. For the actual packet transmission, this process is handled using code run on the MCU.



Figure 1‑6: MATLAB function to append a 16 Bit CRC to the QPSK packet.

1. Create a new **MATLAB function** with the contents of Figure 1-7.
2. **Save** this function as CreateAppend16BitCRC.m inside the project directory

## 1.5 HDL Coder Project

Now that the MATLAB files have been created, we can turn them into PCores. As mentioned earlier, we will reuse the previously created MCU and DAC Driver PCores, thus the only cores we need to create for this lab are the qpsk\_tx and tx\_fifo PCores. Using the same steps outlined in the previous labs, create a new HDL coder project called qpsk\_tx. Add both your qpsk\_tx.m file and your qpsk\_tb.m files to the **MATLAB Function** and **MATLAB Test Bench** categories respectively.

1. Once inside the workflow advisor screen, click on **HDL Code Generation** on the left hand side, and be sure to set the clock to be driven at the **DUT base rate** as in the previous labs.
2. Right-click **Fixed-Point Conversion,** and select **Run to Selected Task.**
3. There two functions within the qpsk\_tx PCore that requires modification of the proposed variables types. For these functions, use the figures below to correct each variable type.

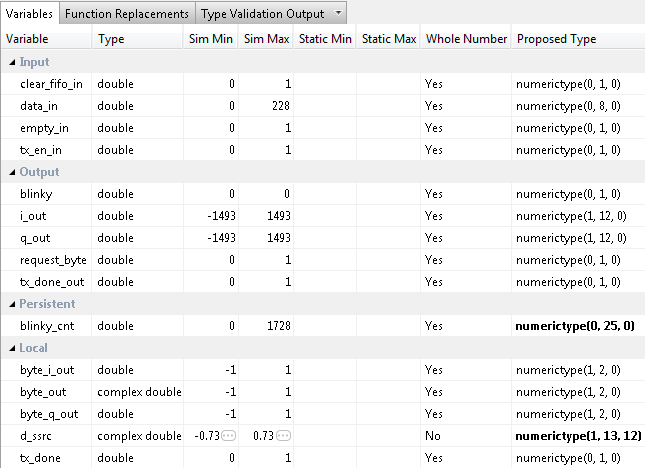


Figure 1‑7: Variable types for qpsk\_tx function

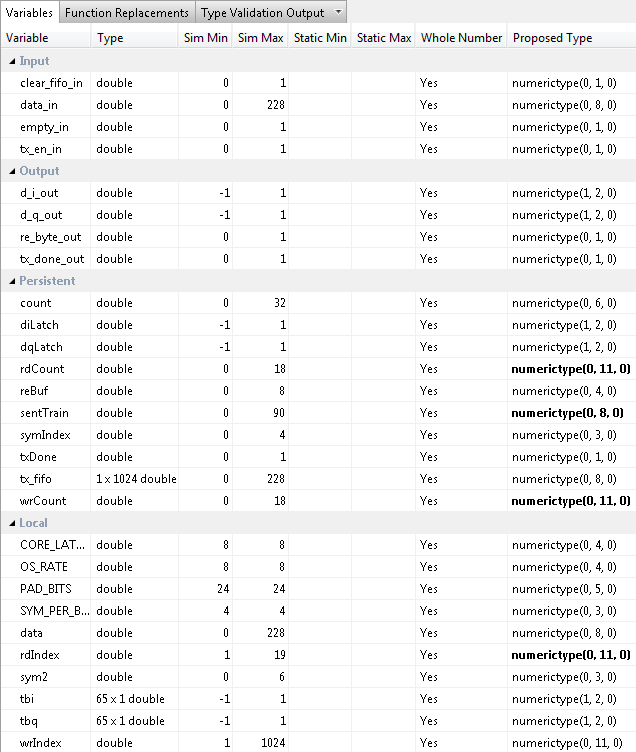
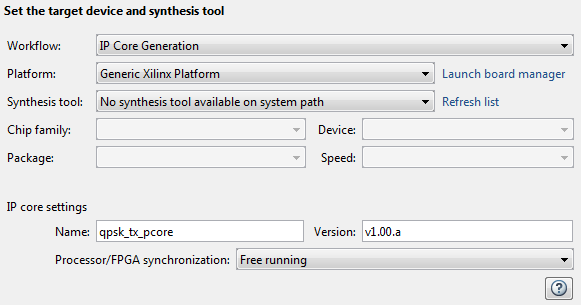


Figure 1-8: Variable types for the qpsk\_tx\_byte2sym function

1. Once you have corrected the **Type** setting for all your variables, click **Select** **Code Generation Target**. Here you can select the FPGA you will use for your design. For this Lab, we will not be using any of the built-in Zynq board functionality within our MATLAB PCores. Therefore you can leave the default settings. Ensure your Workflow settings resemble figure 1-4 below



1‑9: Settings for Xilinx Zed Board HDL Coder Design

1. Just below the synthesis tool settings, **rename your PCore** to qpsk\_tx\_pcore or something similar. This is optional as MATLAB will give its default name for each of your cores, as well as a default version, however it is helpful to rename your core for easier netlist configuration later in the lab.
2. Once the platform and synthesis tool are set, you can click **Set Target Interface** to configure the input and output ports of the design. For this Lab, follow the settings shown in Figure 1-10 below.

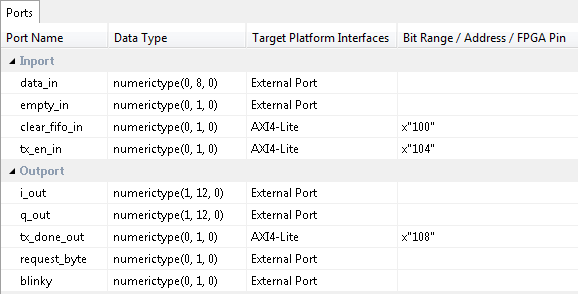


Figure 1-10: Port Interface settings for qpsk\_tx HDL Coder project

1. Once the ports are set, right-click **HDL Code Generation** and select Run This Task. This will create a PCore for your design that can be used directly within Xilinx EDK. By default, the PCore is created in <Project Directory/MATLAB folder/codegen/ipcore>.
2. Repeat this process for the tx\_fifo function. **Name the PCore** tx\_fifo\_pcoreand verify your **Fixed-Point variable** conversions and your **Target interface port** settings using the Figures below. Also don’t forget to set both projects to use the **DUT base** clock rate.

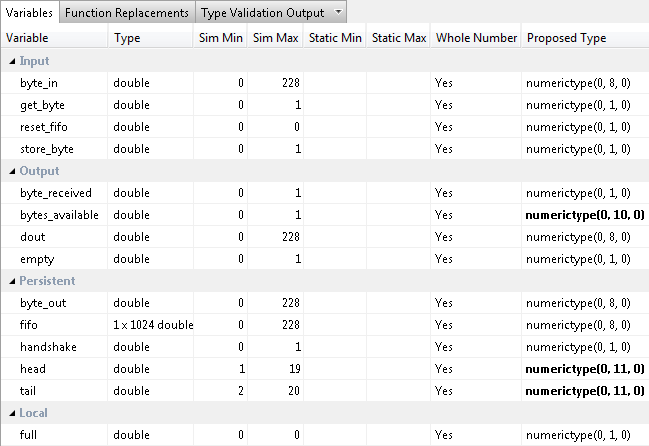


Figure 1-11: Fixed-Point Variables for TX FIFO PCore

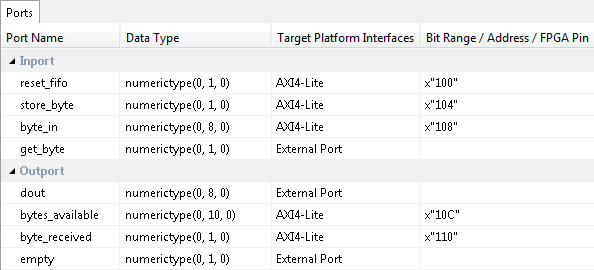


Figure 1-12: Port settings for TX FIFO PCore

1. Once all of the PCores have been created, make a **new EDK project** using the same method used in the previous lab. Be sure that you **import** the correct system configuration file.
2. Once the project is created, **copy each of the PCore folders** from the MATLAB directory into the PCores folder of your **EDK Project**. Don’t forget to also copy any previously created cores you may be reusing as well. 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

* DAC Driver
* MCU Driver
* MCU UART
* QPSK TX PCore
* TX FIFO PCore
* Clock Generator
* GPIO for LEDs, Switches and Buttons
* 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.

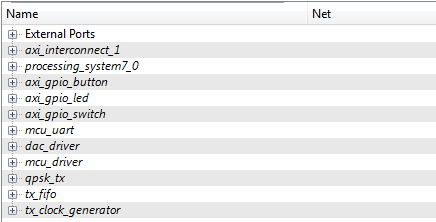


Figure 2‑1: EDK project ports list

## 2.2 Configuring the DAC Driver Port

**Expand** the **DAC Driver** core. There are 7 individual I/O pins which need to be routed in this core.

1. The first two are the tx\_i and tx\_q pins. These are input pins which are used to create the **TXD output** by interleaving an I and Q channel. The signals come directly from the qpsk tx core. **Assign** these two pins to the i\_out and q\_out output pins from the **qpsk\_tx PCore**.
2. Next are the txd, tx\_iq\_sel, and blinky output pins. These pins carry signals which should be routed directly to physical components on the FPGA as **external pins**. The first two are sent to the **FMC** connector port and into the Chilipepper radio board, while the blinky pin connects to an LED on the FPGA. **Assign** all of these pins as **external ports.**
3. Connect the IPCORE\_RESETN port to the processing\_system7 FCLK\_RESET0\_N port.
4. The IPCORE\_CLK pin can be skipped for now and will be connected later in **section 2.9**

## 2.3 Configuring the MCU Driver Port

**Expand** the **MCU Driver** core. There are 9 individual I/O pins which need to be routed on this core.

1. Configuring this core is very simple as **all of the pins** with the exception of the IPCORE\_CLK and the IPCORE\_RESETN are simply **assigned** as **external ports**.
2. Connect the IPCORE\_RESETN port to the processing\_system7 FCLK\_RESET0\_N Port and skip the IPCORE\_CLK for now.

## 2.4 Configuring the MCU UART

1. Under the Communications Low-Speed section, add the AXI UART (Lite) to your design
2. Name the core mcu\_uart as shown in Figure 2-1. Keep all configuration settings as default.
3. This core requires no other customization; just verify the RX and TX pins are set as External ports.

## 2.5 Configuring the QPSK TX

**Expand** the **QPSK TX** core. There are 8 individual I/O pins which need to be routed on this core.

1. If the DAC driver was previously configured correctly, the i\_out and q\_out pins of the qpsk\_tx core should already be set.
2. The data\_in, empty\_in, and request byte ports of the qpsk\_tx core should be connected to the dout, empty, and get byte ports of the tx\_fifo core respectively.
3. Set the blinky pin as an External port.
4. Connect the IPCORE\_RESETN port to the processing\_system7 FCLK\_RESET0\_N Port and skip the IPCORE\_CLK for now.

## 2.6 Configuring the TX FIFO

**Expand** the **TX FIFO** core. There are 5 individual I/O pins which need to be routed on this core.

1. If the qpsk\_tx core was previously configured correctly, the get\_byte, dout and empty pins of the tx\_fifo core should already be set.
2. Connect the IPCORE\_RESETN port to the processing\_system7 FCLK\_RESET0\_N Port and skip the IPCORE\_CLK for now.

## 2.7 Configuring the LEDs GPIO Port

This port will be used later in SDK to verify the functionality of the transmitter.

1. Add an AXI General Purpose IO to the design. Check the box to Enable Channel 2 and give each channel a width of 1 bit. Name the port axi\_gpio\_led or something similar.
2. Expand the IO\_IF section of the GPIO, and assign the GPIO\_IO and GPIO2\_IO pins to external ports. The other pins can be left blank.

## 2.8 Configuring the Switch GPIO Port

This port will be used later in SDK to change the transmit mode.

1. Add an AXI General Purpose IO to the design. Leave both boxes unchecked and give channel 1 a width of 2 bits. Name the port axi\_gpio\_switch or something similar.
2. Expand the IO\_IF section of the GPIO, and assign the GPIO\_IO pin to an external port. The other pins can be left blank.

## 2.9 Configuring the Button GPIO Port

This port will be used later in SDK to transmit packets individually.

1. Add an AXI General Purpose IO to the design Check the box to Support Interrupts. Give channel 1 a width of 1 bit and check the box for Channel 1 to be input only. Name the port axi\_gpio\_button or something similar.
2. Expand the IO\_IF section of the GPIO, and assign the GPIO\_IO\_I pin to an external port. The other pins can be left blank.

## 2.10 Configuring the Clock Generator IP Core

The Clock Generator is used in this project to distribute the appropriate clock signals to each of the PCores required for transmitting the qpsk signal, as well as 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, tx\_fifo, qpsk\_tx, dac\_driver) and the TX\_CLK and RX\_CLK signals; which latch data from the TXD and RXD lines to the DAC and ADC respectively on the radio board. Although no ADC is used within the design, the clock is required for proper initialization of the Chilipepper FMC. 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**, 0Phase, **PLLE0** group and **Buffered True**
* CLKOUT2 Required Frequency of **40Mhz**, 0Phase, **PLLE0** group and **Buffered True**
* CLKOUT3 Required Frequency of **40Mhz**, 0Phase, **PLLE0** group and **Buffered True**

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

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

* CLKIN External Ports
* CLKOUT0 mcu\_driver::IPCORE\_CLK and tx\_fifo::IPCORE\_CLK and qpsk\_tx:: IPCORE\_CLK
* CLKOUT1 External Ports
* CLKOUT2 External Ports
* RST net\_gnd
* LOCKED External Port

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

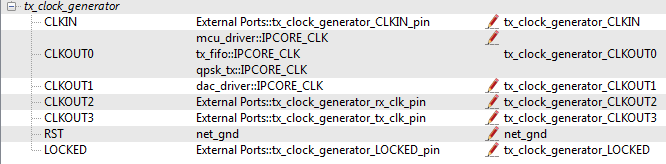


Figure 2‑2: Clock Generator port configuration

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

## 2.11 Pin Assignments

Once the clock generator is configured correctly, the IPCORE\_CLK clock 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 Figure 2-3 below as a reference.



Figure 2‑3: EDK project pin assignments

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 workspace folder in the same directory as your Sysgen 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 transmit project.

1. Select **File 🡪 New** 🡪 **Application Project**.
2. Name the project “qpsk\_transmit” 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 tone project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the tone 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 D.
7. **Download** the **Chilipepper.c** and **Chilipepper.h** files from the GitHub repository[[2]](#footnote-2) 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. The only PCores that should be defined at the top of the file are the MCU\_Driver, DAC\_DRIVER, TX\_PCORE and TX\_FIFO.

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.

**Note**

1. (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.



## 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[[3]](#footnote-3)* 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

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



Figure 3‑3: iMPACT configuration screen

## 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 tone 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 qpsk\_tx, MCU and DAC PCores.

Testing and Design Verification Step 4

## 4.1 Verification with another Chilipepper Board

If you have access to a second Chilipepper board and FPGA, you can verify the qpsk\_tx lab by setting up the second board to receive the transmitted message. To do this, you will also need an SD Card which will allow you to load a design on the second board. The second board can then output any messages it receives directly to a uart terminal.

1. Download the BOOT.bin file for Lab 8 from the GitHub website[[4]](#footnote-4). Copy the file to a blank SD Card.
2. Connect the second FPGA to your PC, and configure it to boot from the SD Card.
3. Once the system has loaded, connect to a terminal to view the boards output. The terminal should be configured as 115200 baud.

Once the second board is configured, send a packet from the Lab 3 board, and you should see “hello world” display on the terminal from the second board.

## 4.2 Verification with ChipScope

If you don’t have access to a second board, you can still verify the qpsk transmit signal by analyzing the signal in MATLAB. To do this, you will need to expand the EDK design to also include a ChipScope Debug configuration.

1. Add a ChipScope Peripheral to your design to monitor the output of the DAC Driver. Refer to Lab 2 section 2.7 for information on how to add ChipScope to the design. Be sure you assign ChipScope to the same clock used for the dac\_driver.
2. Re-export your new design, and run it.
3. Open **ChipScope Pro Analyzer**. Be sure that the JTAG cable is connected to the FPGA board properly.
4. Once the program opens, click the  (open cable) button to open your JTAG connection to the board. If your jumpers are configured correctly, you should see the following devices on the cable.



If you receive an error from ChipScope stating that you either cannot detect or cannot open the cable, try using the optional Step 5 to configure your cable setup correctly.

**Note**

1. (**Optional**)Click JTAG Chain in the top menu selection. Select the option for **Open Plug-in**... You will be greeted with a Plug-in Parameters screen. Enter the following in the box, and hit ok. “**xilinx\_tcf URL=tcp::3121**”. Then click the open cable button and proceed as usual.
2. Select ok to get to the Analyzer main screen. Open the file menu and select **Import**.
3. Click **Select New File**, and browse to the location of your ChipScope **CDC file**, which is located in the <EDK/implementation/ chipscope\_ila\_0\_wrapper> folder of your project directory. This file was created for you when you generated your bit file in EDK, assuming you added the ChipScope peripheral appropriately. It tells the ChipScope program how to interpret the data it is receiving from the JTAG port.
4. On the Bus Plot screen, you can view the txd signal that you connected to your ChipScope peripheral previously. Right click on a signal to change its features such as bus radix, name or color. For this Lab, the txd signal should be set to the signed decimal bus radix.
5. Click the **play button** in the top menu bar to display the signal. For this lab, it is helpful to set triggering options such that you can see the output just as a packet is being transmitted. Your transmitted waveform should look similar to either Figure 4-1.

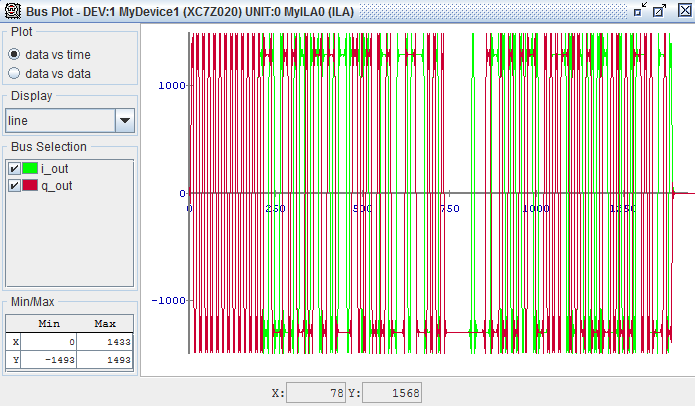


Figure 4-1: QPSK transmit waveform

1. Open the file menu and select **Export**.
2. Click the Ascii radio box and change the Signal to Export to the Bus Plot. Then click export.
3. Save the file in a convenient location. It is recommended you save this file in the same location as your MATLAB files. Call it something descriptive such as **TX.prn**.

This file contains all the information used to display the Bus Plots in ChipScope, and can be opened directly in a MATLAB script to display, as well as analyze using built in MATLAB spectral analysis functions.

**Note**

## MATLAB Analysis

1. To display the data in MATLAB, simply run the code shown below.



This plots the waveforms in MATLAB. To verify the qpsk waveform, simply change the sim variable in the qpsk\_tb file to 0, and load the TX.prn file data directly. If your design is working, you should see a “Tranmitted message correctly” when you run qpsk\_tb.

1. MATLAB byte2sym Function

MATLAB function qpsk\_tx\_byte2sym.m







1. MATLAB TX FIFO Function

MATLAB script tx\_fifo.m





1. MATLAB Test Bench

MATLAB function qpsk\_tx\_tb.m







1. SDK source file

MATLAB script main.c





1. Can be found at <https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/Labs/Lab_3/MATLAB> [↑](#footnote-ref-1)
2. Can be found at <https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/ChilipepperSupport/Library%20Files> [↑](#footnote-ref-2)
3. Can be found at <https://github.com/Toyon/Chilipepper/tree/master/QPSK_Radio/DemoFilesAndDocumentation> [↑](#footnote-ref-3)
4. <https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/Labs/Lab_8/DemoFilesAndDocumentation/BOOT> [↑](#footnote-ref-4)