

Toyon Research Corporation

# Lab 3: Output QPSK

Chilipepper Tutorial Projects

Version 0.1

1/9/2013

## Table of Contents

Introduction .....	3
Procedure.....	3
Objectives .....	3
Generate HDL Code .....	4
1.1    MATLAB Functions.....	4
1.2    MATLAB Testbench.....	9
1.3    HDL Coder Project .....	10
Create and Export Simulink Models.....	12
2.1    Modify Tone Simulink Design .....	12
2.1    Create MCU Simulink Design .....	14
2.3    Create DAC driver Simulink Design.....	15
Configure Cores and Export Design .....	17
3.1    Needed IP Cores .....	17
3.2    Configuring the tx Port .....	18
3.3    Configuring the DAC Driver Port.....	18
3.4    Configuring the MCU Port .....	18
3.5    Configuring the Clock Generator IP Core .....	18
2.5    Pin Assignments.....	20
Create software project .....	22
3.1    Creating a new C Project .....	22
3.2    Debugging with SDK, iMPACT and ChipScope Pro.....	23
3.3    Debugging with SDK.....	24
Testing and Design Verification.....	25
4.1    Verification with ChipScope Pro .....	25
4.2    MATLAB Analysis .....	27
Appendix A    MATLAB Core Functions.....	29
Appendix B    MATLAB Test Bench .....	32

# 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 on the FPGA following the “Black Box” approach used in previous labs. 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 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 2012b
- Xilinx ISE Design Suite 14.3 with EDK and System Generator
- 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
- Create and export Simulink models using System Generator
- 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 Simulink model 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 as well as the process for generating the HDL code used in the Simulink model.

### 1.1 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.

```
function [i_out, q_out, re_byte_out, tx_done_out] = ...
    qpsk_tx(data_in, empty_in, clear_fifo_in, tx_en_in)

    [d_b2s, re_byte_b2s, tx_done_b2s] = ...
        qpsk_tx_byte2sym(data_in, empty_in, clear_fifo_in, tx_en_in);

    [d_ssrc] = qpsk_ssrc(d_b2s);

    % make i/q discrete ports and scale to the full 12-bit range of the DAC
    % (one bit is for sign)
    i_out = round(real(d_ssrc)*2^11);
    q_out = round(imag(d_ssrc)*2^11);

    re_byte_out = re_byte_b2s;
    tx_done_out = tx_done_b2s;
end
```

Figure 1-0-1: MATLAB function to Create i and q channel outputs.

This function is primarily used to call other functions which create the waveforms, and format the output before sending it to the DAC.

1. Create a directory for the project under C:\QPSK\_Projects\Project\_3.
2. Create a new **MATLAB function** with the contents of Figure 1-1.
3. **Save** this function as `qpsk_tx.m` inside the project directory.

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

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

4. Create a new **MATLAB function** with the contents of Appendix A.
  - a. The algorithm uses an internal `TX_FIFO` buffer to store bytes waiting to be transmitted.
  - b. Initially, nothing is transmitted, and the algorithm simply fills its buffer with data needing to be transmitted.
  - c. 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.
  - d. Lastly the algorithm sends the data in its buffer until the buffer is empty, and then activates the `tx_done` output
5. Save this function as `qpsk_tx_byte2sym.m` inside the project directory

There is a function 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. 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.

```
function b = mybitget(by, p)

    switch p
        case 1
            u = floor(by/2^0);
        case 2
            u = floor(by/2^1);
        case 3
            u = floor(by/2^2);
        case 4
            u = floor(by/2^3);
        case 5
            u = floor(by/2^4);
        case 6
            u = floor(by/2^5);
        case 7
            u = floor(by/2^6);
        case 8
            u = floor(by/2^7);
        otherwise
            u = 0;
    end
    b = mod(u, 2);
end
```

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

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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 result of the modulation causes the phase of the sine and cosine waves to vary according to the table below. When grouping each modulated sinusoid into a bit symbol, 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.

i channel		(iq) Bit Symbol	Symbol Phase
Bit value	NRZ Value		
1	1	(11)	45°
0	-1	(01)	135°

q channel		(iq) Bit Symbol	Symbol Phase
Bit value	NRZ Value		
1	1	(00)	225°
0	-1	(10)	315°

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

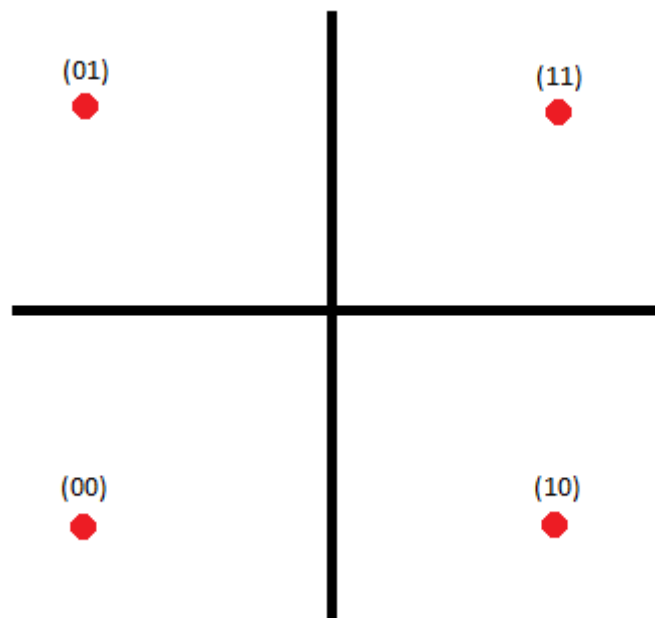


Figure 1-3: Scatter plot of the QPSK symbols.

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

6. Create a new **MATLAB function** with the contents of Figure 1-2.
7. **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 intersymbol 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.

```
function [d_out] = qpsk_srrc(d_in)

    persistent buf

    OS_RATE = 8;
    f = SRRC;

    if isempty(buf)
        buf = complex(zeros(1, OS_RATE*2+1), zeros(1, OS_RATE*2+1));
    end

    buf = [buf(2:end) d_in];

    d_out = buf*f;

end
```

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

```
function make_srrc_lut
    OS_RATE = 8;

    f = firrcos(2*OS_RATE, .25, .25, 1, 'rolloff', 'sqrt');
    f = f/sum(abs(f)); % make sure no matter what we don't go beyond 1

    fid = fopen('SRRC.m', 'w+');
    fprintf(fid, 'function y = SRRC\n');
    fprintf(fid, '%s#codegen\n');
    fprintf(fid, 'y = [\n');
    fprintf(fid, '%13.12f\n', f);
    fprintf(fid, '];\n');
    fclose(fid);

end
```

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

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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.

8. Create 2 new **MATLAB function** with the contents of Figure 1-4 and 1-5.
9. **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 is a second LUT which is used to send a header for the QPSK packet. This Header is most commonly used to assist with correctly receiving the transmitted packet. The header used for this lab is called the “kassami sequence” and it is a sequence bit (NRZ in this case) which is proven to have very good cross-correlation values. This means that when using correlation to correctly detect this sequence at the receiver (in the presence of white Gaussian noise) it is highly likely that this sequence can be detected without error. The code to create the LUT is shown in Figure 1-6.

```
function make_train_lut

hks_i = comm.KasamiSequence('SamplesPerFrame', 65, 'Index', 1);
hks_q = comm.KasamiSequence('SamplesPerFrame', 65, 'Index', 3);
x_i = step(hks_i);
x_q = step(hks_q);
t_i = x_i*2-1;
t_q = x_q*2-1;

fid = fopen('TB_i.m', 'w+');
fprintf(fid, 'function y = TB_i\n');
fprintf(fid, '%%#codegen\n');
fprintf(fid, 'y = [\n');
fprintf(fid, '%1d\n', t_i);
fprintf(fid, '];\n');
fclose(fid);
fid = fopen('TB_q.m', 'w+');
fprintf(fid, 'function y = TB_q\n');
fprintf(fid, '%%#codegen\n');
fprintf(fid, 'y = [\n');
fprintf(fid, '%1d\n', t_q);
fprintf(fid, '];\n');
fclose(fid);

end
```

Figure 1-6: MATLAB function to create kassami sequence header for QPSK Packet

This completes the MATLAB functions required for creation of the QPSK waveform.



## 1.2 MATLAB Testbench

The last MATLAB code required for this core is the test bench script. This script is required for HDL generation, but also allows you to test the functionality of the MATLAB algorithm. The code used for this script is shown in Appendix B. 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 and analyze the results. Setting it to 1 simulates the waveform, 0 loads it from a prn file

### Note

If you decide to load data from ChipScope, you must be sure the names of the i and q channel variables in the script, match the names of the i and q channel data ports in Chipscope.

In addition to the `sim` variable, there is also a function which is called from within the script called `CreateAppend16BitCRC`. This function is shown in Figure 1-7. The purpose of this function 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.

```
function msg_out = CreateAppend16BitCRC(msg_no_zeros)

    valueCRC = 65535;
    genPoly = 4129;

    msg_in = [msg_no_zeros 0 0];
    for i1 = 1:length(msg_in)
        for i2 = 1:8
            b = mod(floor(msg_in(i1)/(2^(8-i2))),2);
            valueCRCsh1 = bitsll(valueCRC,1);
            valueCRCadd1 = bitor(valueCRCsh1,b);
            if floor(valueCRCadd1/2^16) == 1
                valueCRC = bitxor(valueCRCadd1,genPoly);
            else
                valueCRC = valueCRCadd1;
            end
            valueCRC = mod(valueCRC,2^16);
        end
    end

    msg_out = [msg_no_zeros mod(floor(valueCRC/2^8),2^8) mod(valueCRC,2^8)];
end
```

Figure 1-7: MATLAB function to append a 16 Bit CRC to the qpsk packet.

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

1. Create a new **MATLAB script** with the contents of Appendix B.
2. **Save** this script as `qpsk_tx_tb.m` inside the project directory
3. Create a new **MATLAB function** with the contents of Figure 1-7.
4. **Save** this function as `CreateAppend16BitCRC.m` inside the project directory
5. **Run** the test bench script in MATLAB (be sure the project directory is in the MATLAB PATH variable) to test the algorithm and the lookup tables. You may have to run the `make_trig_lut.m` and `make_srrc_lut.m` functions first to create the lookup tables.

**Note**

Feel free to change the ASCII message to experiment with different messages and message lengths. In addition it may be helpful to view a scatter plot of the resultant waveform to verify the phase angles of the qpsk symbols.

6. Once you have verified that your algorithm is correct, proceed to the next step of the lab.

### 1.3 HDL Coder Project

Using the same steps outlined in the previous labs, create a new HDL coder project called `transmit_qpsk`. Add your MATLAB function `qpsk_tx.m` and your test bench script `qpsk_tx_tb.m` to the **MATLAB Function** and **MATLAB Test Bench** categories respectively. The other MATLAB files will be automatically added to the project later; just be sure they are located in the same directory as your qpsk files.



Be sure that the `sim` parameter in your test bench is set to 1 before you add your test bench script to the HDL coder project!

1. Once inside the workflow advisor screen, configure the **Clocks & Ports** settings as well as the **Black Box Generator** settings (See lab 0).
2. Right-click **Propose Fixed-Point Types**, and select **Run to Selected Task**. For this Lab, most of the proposed types do not require changes. However in the `qpsk_tx_byte2sym` function, changes your settings to resemble the ones shown in Figure 1-8.

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

	Min	Max	IsInt	Type	RoundMode	OverflowMode
CORE_LATEN...	8.0	8.0	<input checked="" type="checkbox"/>	ufix4	floor	wrap
OS_RATE	8.0	8.0	<input checked="" type="checkbox"/>	ufix4	floor	wrap
PAD_BITS	24.0	24.0	<input checked="" type="checkbox"/>	ufix5	floor	wrap
SYM_PER_BYTE	4.0	4.0	<input checked="" type="checkbox"/>	ufix3	floor	wrap
clear_fifo_in	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
count	0.0	32.0	<input checked="" type="checkbox"/>	ufix10	floor	wrap
d_out	-1.0	1.0	<input checked="" type="checkbox"/>	sfix3	floor	wrap
data	0.0	157.0	<input checked="" type="checkbox"/>	ufix8	floor	wrap
data_in	0.0	157.0	<input checked="" type="checkbox"/>	ufix8	floor	wrap
diLatch	-1.0	1.0	<input checked="" type="checkbox"/>	sfix3	floor	wrap
dqLatch	-1.0	1.0	<input checked="" type="checkbox"/>	sfix3	floor	wrap
empty_in	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
rdCount	0.0	18.0	<input checked="" type="checkbox"/>	ufix10	floor	wrap
rdIndex	1.0	19.0	<input checked="" type="checkbox"/>	ufix10	floor	wrap
reBuf	0.0	8.0	<input checked="" type="checkbox"/>	ufix4	floor	wrap
re_byte_out	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
sentTrain	0.0	90.0	<input checked="" type="checkbox"/>	ufix7	floor	wrap
sym2	0.0	6.0	<input checked="" type="checkbox"/>	ufix3	floor	wrap
symIndex	0.0	4.0	<input checked="" type="checkbox"/>	ufix3	floor	wrap
tbi	-1.0	1.0	<input checked="" type="checkbox"/>	sfix3	floor	wrap
tbq	-1.0	1.0	<input checked="" type="checkbox"/>	sfix3	floor	wrap
txDone	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
tx_done_out	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
tx_en_in	0.0	1.0	<input checked="" type="checkbox"/>	ufix1	floor	wrap
tx_fifo	0.0	157.0	<input checked="" type="checkbox"/>	ufix8	floor	wrap
wrCount	0.0	18.0	<input checked="" type="checkbox"/>	ufix10	floor	wrap
wrIndex	1.0	1024.0	<input checked="" type="checkbox"/>	ufix11	floor	wrap

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

- Once you have corrected the **Type** setting for all your variables, right-click **Code Generation**, and select **Run to Selected Task**. A Simulink model containing your design should appear after the task is completed.

## Create and Export Simulink Models

## Step 2

---

This section will show you how to customize your Simulink Models to generate the qpsk waveform and control the Chilipepper MCU and DAC.

### 2.1 Modify Tone Simulink Design

1. As in the previous labs, you first need to copy and paste the contents of the design into a **new Simulink Model**. Call this new model `tx.slx`, and **save** it into the **Sysgen folder**. Be sure to change the `cfg` file as well to find the files in your new directory structure.
2. Once you have created your new model, add blocks from the Xilinx blockset library to create a model similar to the one shown in Figure 2-1. Configure the blocks using the following.
  - a. The assert block should be configured to assert the rate explicitly, with sample rate 1.
  - b. Convert 1 should have a Boolean output type.
  - c. Configure the **From FIFO** block as follows: **Shared memory name** `tx_fifo`; **Ownership** Owned elsewhere; **Memory type** Block RAM; **Performance Options** Standard FIFO; **Depth** 16; **BITS of precision to use for % full port** 1.
  - d. Convert 2 should have a Fixed-point output type, unsigned with 1 bit and no binary points. Set the Quantization and Overflow to Truncate and Wrap respectively.
  - e. The constant is a Fixed-point unsigned 0 constant with 1 bit and no binary points. It is a sampled constant with a period of 1.
  - f. The two From Registers should have the names changed to those shown in the figure. They are owned and initialized elsewhere with a sample period of 1. Additionally both registers are Fixed-point and unsigned with 1 bit and no binary points.
  - g. The counter is a Free running up counter with an initial value of 0 and a step of 1. Set it as an unsigned value with single bit precision, no binary points, and explicit period one.
  - h. Convert 3 converts the value of this counter to Boolean with 0 Latency.
  - i. The To Register should also have its named changed to the one shown in the figure. It is locally owned and initialized with a value of 0. Specify the output type as unsigned, Fixed-point, and 1 bit with 0 binary points.
  - j. The blinky subsystem is the same one used from the previous labs
  - k. Lastly, the Chipscope block should have 8 trigger ports, 0 display settings, 1 match unit with a basic match type, and can use the trigger ports as data. Set the Depth of the capture buffer to 8192 and select both checkboxes under Implementation.

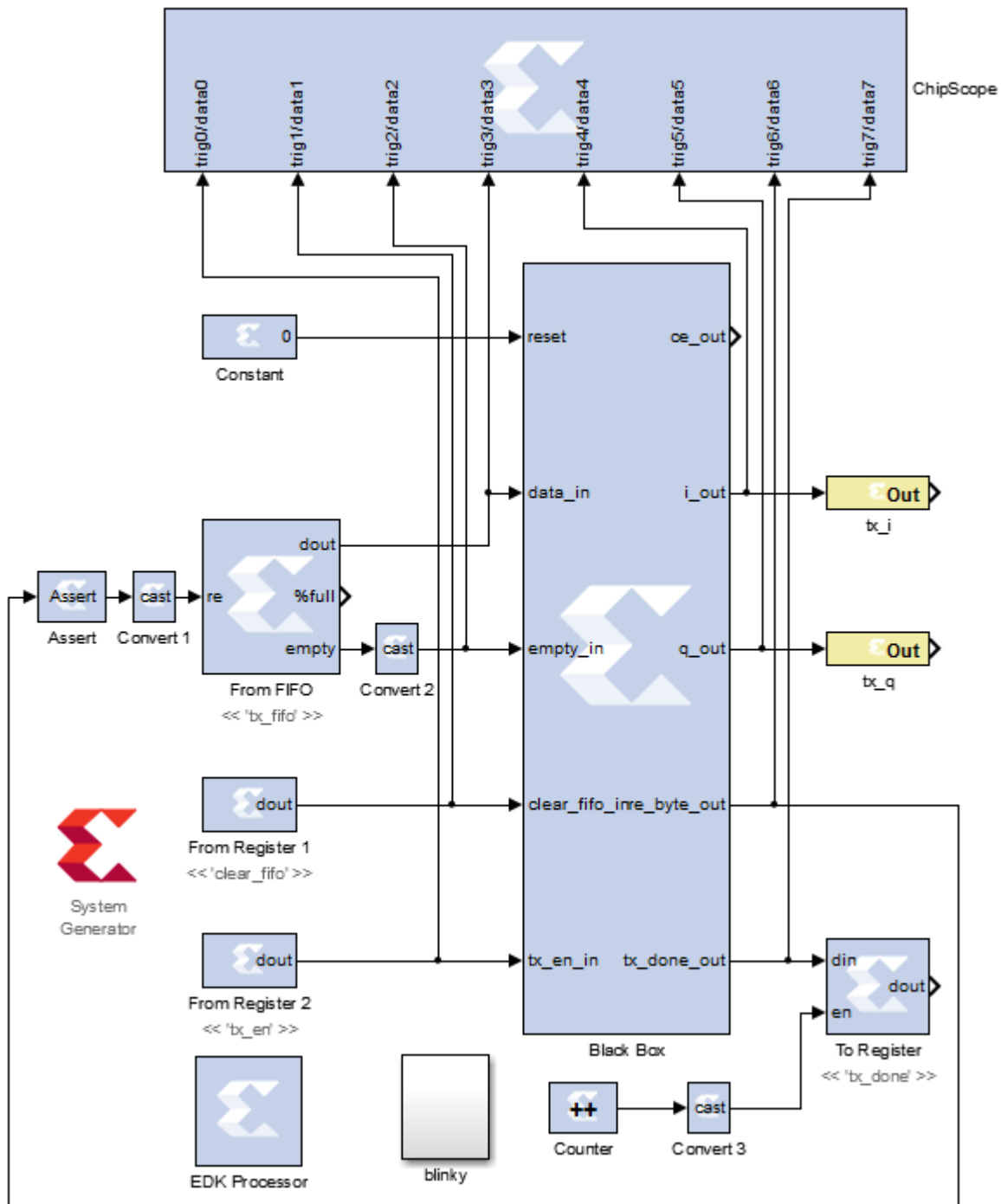


Figure 2-1: Simulink model for tx.slx

- Now configure the **System Generator** using the same method used in the previous labs. Save the design, and be sure to change the cfg file as well to find the files in your new directory structure.

**Note**

Be sure to first sync the EDK Processor to the memory mapped registers using the method used in the previous labs

## 2.1 Create MCU Simulink Design

The **Simulink model**<sup>1</sup> in Figure 2-1 will be used for the control signals to and from the **MCU**. This is the same model used in the previous labs.

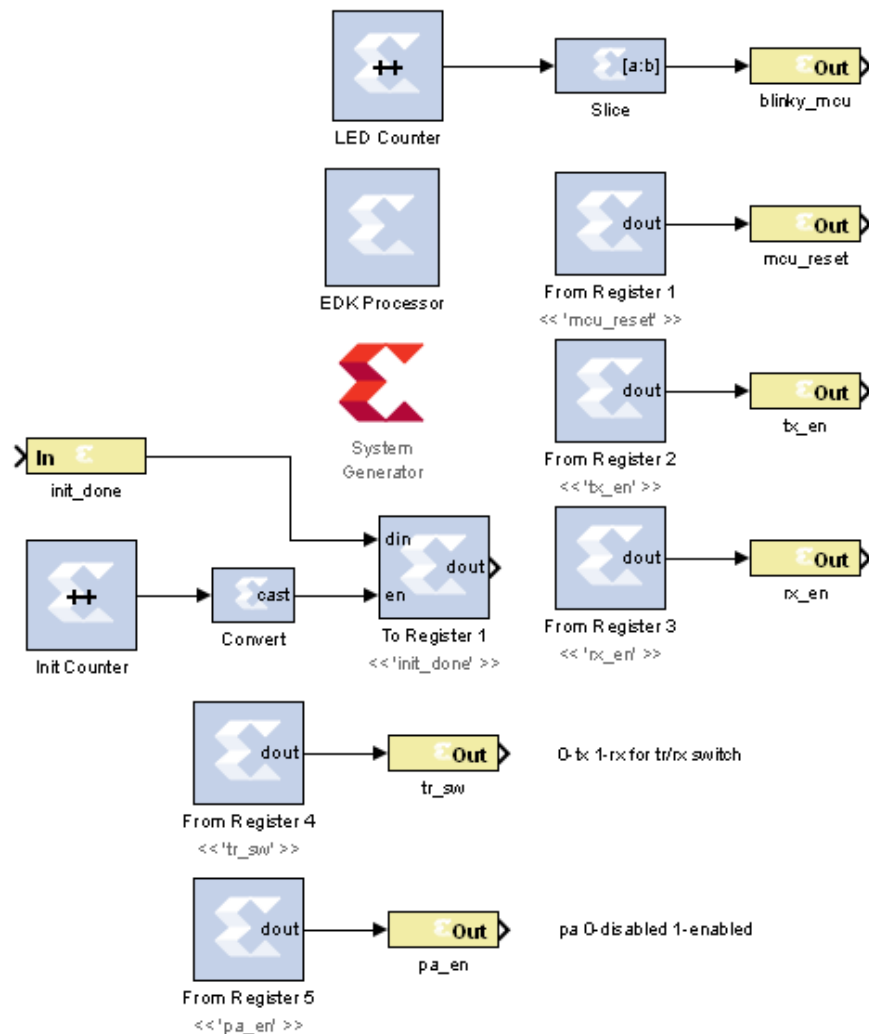


Figure 2-1: Simulink model for MCU control

<sup>1</sup> This model can be downloaded from <https://github.com/rcagley/Chilipepper/tree/master/Labs/Lab%202/sysgen>

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

1. **Configure** this model and the system generator the same as in previous labs, and **save** the design. Name the file **mcu.slx** or something similar using the appropriate directory structures.

### 2.3 Create DAC driver Simulink Design

The **Simulink model**<sup>2</sup> In Figure 1-2 will be used for creating the signals which interface to the DAC. This is the same model used in Lab 2.

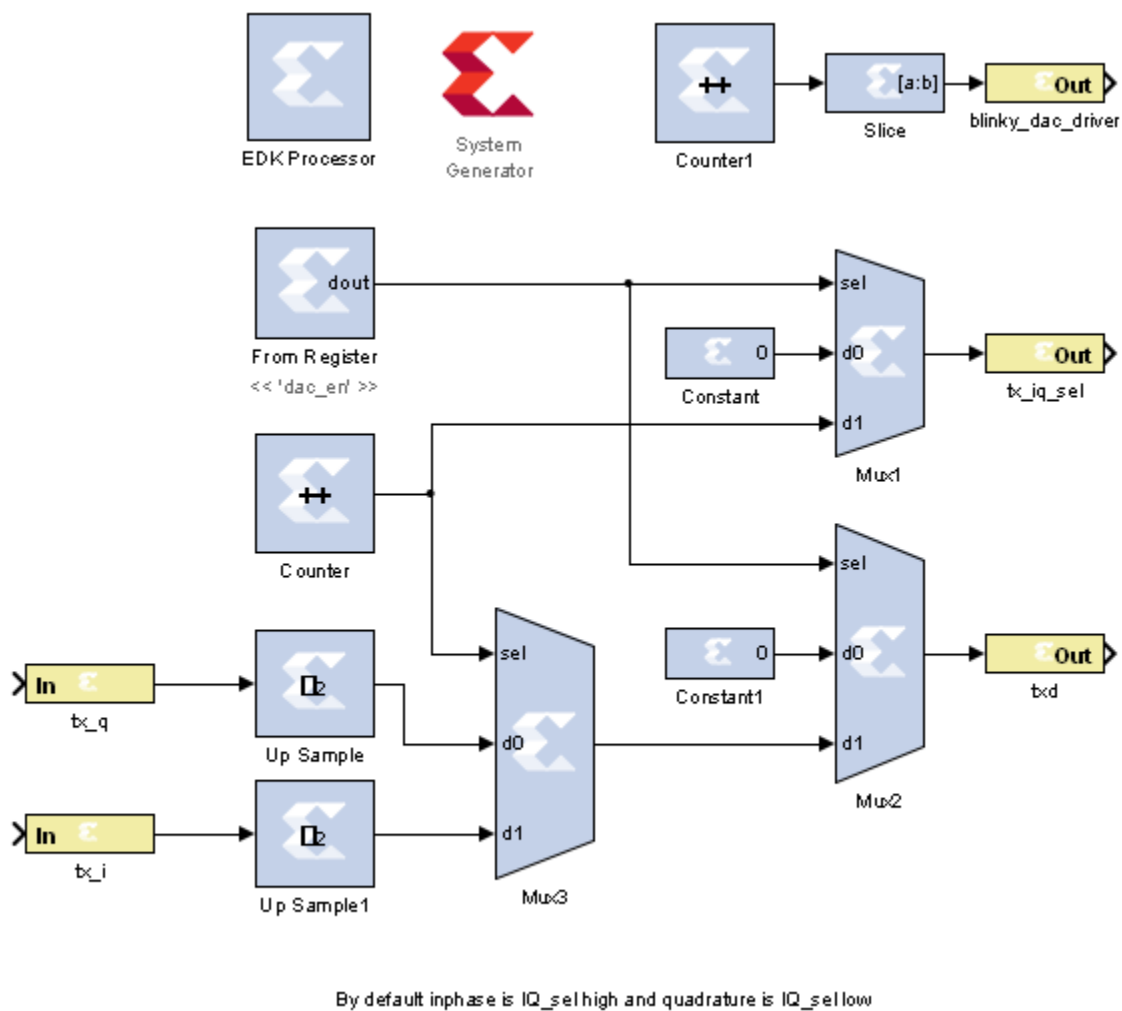


Figure 2-2: Simulink model for ADC control

<sup>2</sup> This model can be downloaded from <https://github.com/rcagley/Chilipepper/tree/master/Labs/Lab%202/sysgen>

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

1. **Configure** this model and the system generator the same as in previous labs, and **save** the design. Name the file **dac\_driver.slx** or something similar using the appropriate directory structures.

Refer to the previous labs to **Create a New Blank EDK Project**. Be sure to follow the directory structure used. Once your project is created, **export** each model 1 by 1 into the newly created EDK project. Be sure your **Compilation Settings** are correct and refer to previous labs as a reference if needed. Once each Simulink model has been exported successfully, you're ready to configure your FPGA design.



## Configure Cores and Export Design

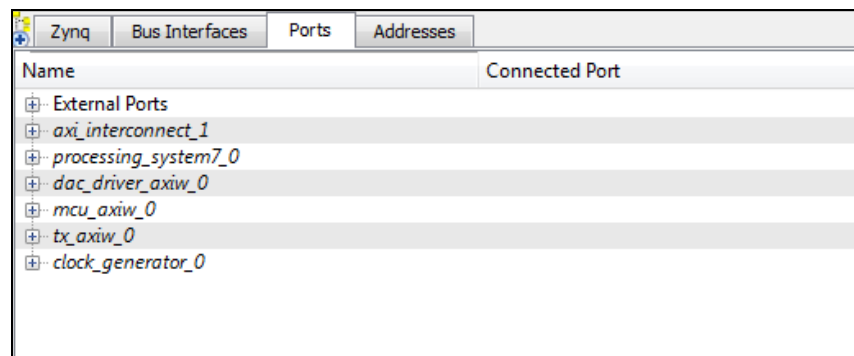
## Step 3

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 previous labs for information on how to add cores to the design.

### 3.1 Needed IP Cores

- Tx PCore created in Simulink
- DAC Driver PCore created in Simulink
- MCU PCore created in Simulink
- Clock Generator IP Core
- Processing System IP Core
- AXI Interconnect IP Core

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 3-1 Below.



Zynq Bus Interfaces Ports Addresses	
Name	Connected Port
+ External Ports	
+ axi_interconnect_1	
+ processing_system7_0	
+ dac_driver_axiw_0	
+ mcu_axiw_0	
+ tx_axiw_0	
+ clock_generator_0	

Figure 3-1: EDK project ports list

### 3.2 Configuring the tx Port

Configure the port as the following.

1. The blinky\_tx pin should be sent to the external ports.
2. The tx\_i and tx\_q lines should connect to the tx\_i and tx\_q lines on the dac driver.

### 3.3 Configuring the DAC Driver Port

The DAC Driver port in this lab should be configured identically to the DAC Driver port in lab 1.

### 3.4 Configuring the MCU Port

The MCU port in this lab should be configured identically to the MCU port in the previous labs.

### 3.5 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, as well as any external hardware which may require a clock signal. For this project, the 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 4 other devices; 3 PCores (tx, MCU and DAC) and the `TX_CLK` signal which latches data from the FPGA to the DAC on the radio board.

1. **Double click** the Clock Generator PCore and **configure** the settings as follows
  - Input Clock Frequency of **40Mhz**
  - CLKFBIN Required Frequency of **40Mhz** with **no Clock Deskew**
  - CLKFBOUT Required Frequency of **40Mhz**, Required Group **PLLE0**, and **Buffered True**
  - CLKOUT0 Required Frequency of **40MHz**, 0Phase, **PLLE0** group and **Buffered true**
  - CLKOUT1 Required frequency of **20Mhz**, 0Phase, **PLLE0** group and **Buffered true**
  - CLKOUT2 Required Frequency of **40MHz**, **180 Phase**, **PLLE0** group and **Buffered true**

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

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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

- CLKIN → External Ports
- CLKOUT0 → dac\_driver::sysgen\_clk
- CLKOUT1 → mcu::sysgen\_clk **and** tx::sysgen\_clk
- CLKOUT2 → External Ports
- CLKFBIN → CLKFBOUT
- RST → net\_gnd
- LOCKED → External Port

**Note**

The CLKOUT2 pin has a 180 degree phase shift as mentioned in the **Chilipepper user's guide**. This line will be used as the TX\_CLK signal from the FPGA to the radio board.

Your Clock Generator port should look similar to Figure 3-2 below.

Zynq Bus Interfaces Ports Addresses		
Name	Connected Port	Direction
External Ports		
axi_interconnect_1		
processing_system7_0		
dac_driver_axiw_0		
mcu_axiw_0		
tx_axiw_0		
clock_generator_0		
CLKIN	External Ports::clock_generator_0_pll_pin	I
CLKOUT0	dac_driver_axiw_0::sysgen_clk	O
CLKOUT1	mcu_axiw_0::sysgen_clk tx_axiw_0::sysgen_clk	O
CLKOUT2	External Ports::clock_generator_0_tx_clk_pin	O
CLKFBIN	clock_generator_0::CLKFBOUT	I
CLKFBOUT	clock_generator_0::CLKFBIN	O
RST	net_gnd	I
LOCKED	External Ports::clock_generator_0_LOCKED_pin	O

Figure 3-2: Clock Generator port configuration

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

## 2.5 Pin Assignments

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

1. **Rename** the pins of the external ports so they are easily identifiable. Figure 3-3 shows the names used in this demo, however you don't have to use the same naming convention.
2. Fill in the pin out information for your design using Figure 2-3 below as a reference.

```
##### PL clocks and reset #####
NET clock_generator_0_pll_pin      LOC = D18 | IOSTANDARD = LVCMOS25;
NET clock_generator_0_pll_pin      TNM_NET = clock_generator_0_pll;
TIMESPEC TS_clock_generator_0_pll = PERIOD clock_generator_0_pll 40.000 MHz;
##### Rx – FMC interface at 2.5V #####
NET clock_generator_0_tx_clk_pin    LOC = C17      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_tx_iq_sel_pin LOC = B16      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[0]    LOC = A18      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[1]    LOC = A19      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[2]    LOC = E20      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[3]    LOC = G21      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[4]    LOC = F19      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[5]    LOC = G15      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[6]    LOC = E19      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[7]    LOC = G16      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[8]    LOC = G19      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[9]    LOC = A16      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[10]   LOC = A17      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_txd_pin[11]   LOC = C18      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
##### MCU Interface #####
NET mcu_axiw_0_mcu_reset_pin        LOC = K20      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_tx_en_pin            LOC = D22      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_tr_sw_pin            LOC = D20      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_rx_en_pin            LOC = C22      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_pa_en_pin            LOC = E21      | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_init_done_pin        LOC = K19      | IOSTANDARD = LVCMOS25;
##### LEDs #####
NET clock_generator_0_LOCKED_pin     LOC = T22 | IOSTANDARD=LVCMOS33; # "LD0"
NET mcu_axiw_0_blinky_mcu_pin       LOC = T21 | IOSTANDARD=LVCMOS33; # "LD1"
NET tx_axiw_0_blinky_tx_pin          LOC = U22 | IOSTANDARD=LVCMOS33; # "LD2"
NET dac_driver_axiw_0_blinky_dac_driver_pin LOC = U21 | IOSTANDARD=LVCMOS33; # "LD3"
```

Figure 2-3: EDK project pin assignments



Be sure that the **orientation** of the TXD pins is set correctly. If you follow the pin list in the figure above, you must **reverse** the TXD pins in the external ports assignment section. This is done using the same method used in Lab 0 Section 5.2 for the LEDs.

---

At the time of this tutorial, Xilinx had a [documented issue](#)<sup>3</sup> with AXI-bus generation for Simulink PCores targeting the Zynq FPGA. Refer to this issue for more information. As in Lab 0 section 5.2, this bug must be corrected for our project. The steps to perform are identical to those in the previous labs; however they must be performed for **both** of the PCores used in this lab.

Once the fix is applied, 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.

---

<sup>3</sup> Issue can be found at <http://www.xilinx.com/support/answers/51739.htm>

## 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 project. There are several source files<sup>4</sup> which are required to support your software application. Some of the files are created when you create your new C project and do not need to be modified. A list of the files which are not created automatically and/or need to be modified is listed below with a link to download them.

1. [helloworld.c](#) – This file is created when you create your Hello World template project. (see previous labs). It should be modified to look similar to the file found on the website.
2. [Chilipepper.c](#) – This file needs to be created. There are three main functions associated with this file. The first function is `Chilipepper_Initialize`, which is handled similarly to the previous labs. The next is `Chilipepper_AppendCrc` which handles adding the CRC to the end of the packet. The last important function within this file is `Chilipepper_WriteTestPacket` which handles creating a packet to transmit based on a user input string.
3. [Chilipepper.h](#) – This file holds the function prototypes for the Chilipepper.c functions.
4. [xbasictypes.c](#) – This file defines some assert functions which can be used to assist in troubleshooting your software application .
5. [Xbasictypes.h](#) - This file is needed to define some of the variable types used throughout the code.
6. In addition to the files given, you also need to include a Math library which contains the `pow` function that is used when creating the CRC.
  - a. Right click on your project, and select C/C++ Build Settings.
  - b. Under ARM gcc linker, select Libraries.
  - c. Click the  button in the Libraries (-l) section, and type 'm' then hit ok. This adds the math library needed to use this function. Hit apply and Ok to return to your project.

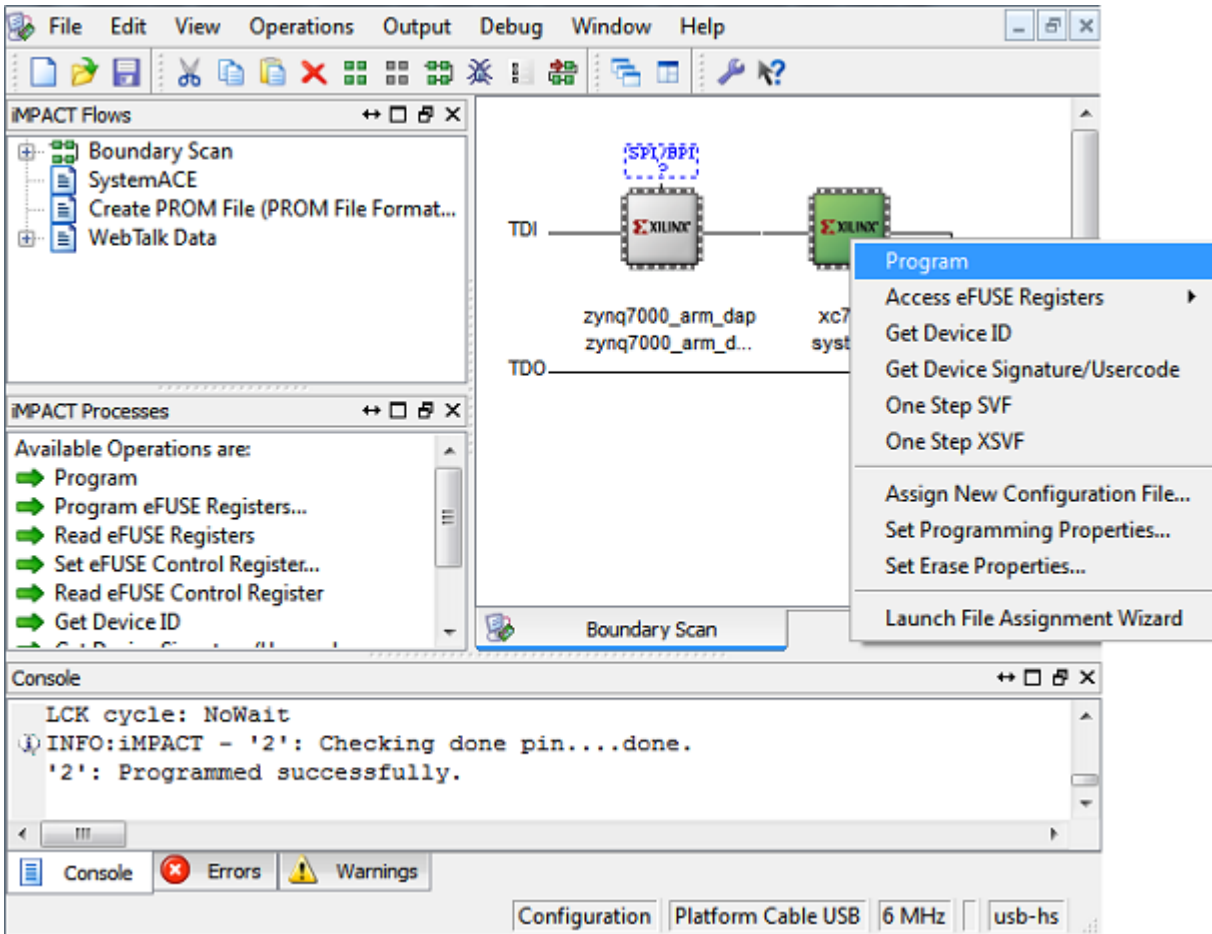
---

<sup>4</sup> The source files can be found at [https://github.com/rcagley/Chilipepper/blob/master/Labs/Lab%203/workspace/hello\\_world/src/helloworld.c](https://github.com/rcagley/Chilipepper/blob/master/Labs/Lab%203/workspace/hello_world/src/helloworld.c)

### 3.2 Debugging with SDK, iMPACT and ChipScope Pro

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 user guide* and the *ZED Board Hardware users guide* 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.
9. Right click on the xc7z020 board icon (should be on the right), select program and hit ok.



3-2: iMPACT configuration screen

### 3.3 Debugging with SDK

If the hardware design is correct, you should see the LEDs start blinking on the board, as well as a blue light 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 `hello_world` project folder and selecting **Debug As → Launch on Hardware**.
2. You should now be taken to a screen which shows the first `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.




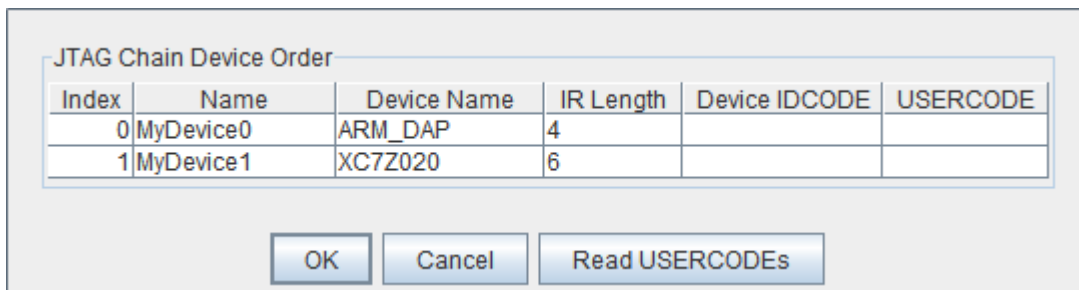
## Testing and Design Verification

## Step 4

### 4.1 Verification with ChipScope Pro

There are several methods available for verifying the qpsk transmission. This lab focuses on verification using ChipScope Pro, as well as exporting to MATLAB for further analysis. In addition, if you have multiple boards available, you can also verify the transmission by receiving the qpsk packet on your second board (more is covered on this topic in future labs).

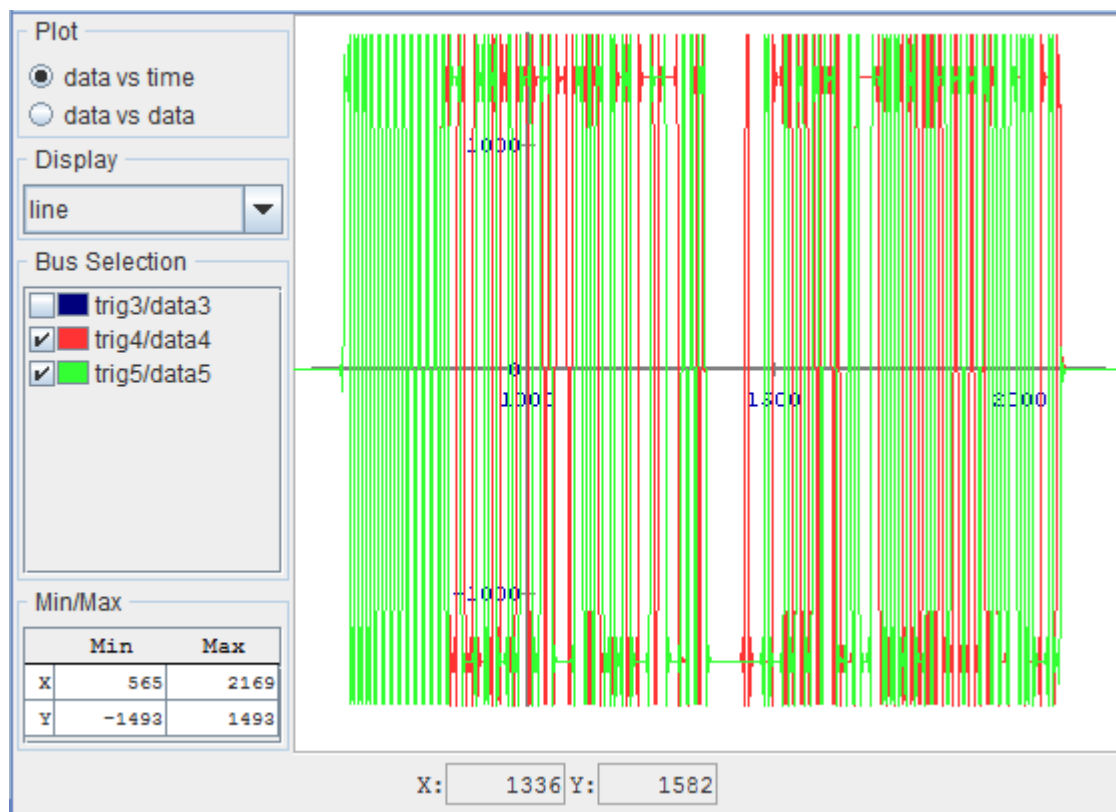
1. To verify the received signal, you will need to open **ChipScope Pro Analyzer**. Be sure that the JTAG cable is connected to the FPGA board properly (or the 6 pin output of the Chilipepper).
2. Once the program opens, click the  (open cable) button to open your JTAG connection to the board. If your jumpers are configured correctly, you should see the following devices on the cable.



3. Select ok to get to the Analyzer main screen. Open the file menu and select **Import**.
4. Click **Select New File**, and browse to the location of your ChipScope **CDC file**, which is located in the Sysgen/netlist folder of your project directory. This file was created for you when you generated your PCores from your Simulink Model design. It tells the ChipScope program how to interpret the data it is receiving from the JTAG port.
5. Next double click on the **bus plot** option in the New Project menu in the top left hand side of the screen. This will open a window which allows you to view a signal **value vs. time** plot of your waveforms.
6. Under Data Port in the Signals Dev menu on the left side of the screen, right click on the trig4/data4 and trig5/data5 ports, and change their **bus radix** to **signed decimal**. Click OK to accept the default decimal values.

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

7. On the Bus Plot screen, you can change the color of each of the signals to get a better view of each individual signal. Click the **check box** next to any of the signals you wish to see on the plot.
8. Click the **play button** in the top menu bar to display the signal. Additionally you can set up triggering options for periodic or continuous playback of the received signal. Your received signal should look similar to Figure 4-1 below.



4-1: both i and q channels of the qpsk waveform in ChipScope Pro

If you changed the message within your C program, your waveform may look slightly different than the one shown here, however you should have the same header padding which can be seen at the front portion of the waveform.

## 4.2 MATLAB Analysis

Now that you have verified the transmitted signal, you can get a pretty good idea of what your qpsk waveform looks like in the time domain. However, ChipScope allows you to export the data received directly into MATLAB for further analysis.

1. Before we export the data, we need to format it as best we can using the triggering options within ChipScope.
  - a. To start, under Trigger Setup, set the depth of the capture to 4096. Your depth may be longer if you created a long test message, however this should be fine for the Hello World example presented here.
  - b. For this example, we will set our trigger to occur when trig2/data2 is not equal to zero. This occurs anytime the FIFO buffer is not empty, and indicates that we are sending data to the buffer to be transmitted.
  - c. In the Match section of Trigger Setup, change the triggering function for Match Unit M2 to `<>` and the Value to one.
  - d. In the Trig section, make sure your condition is active, and set the Condition equation to M2.
2. Now when you hit play, your qpsk data should be near the beginning of your capture.

### Note

The MATLAB test bench function will search for any qpsk signal in the variables exported. Be sure that only one broadcast of your ASCII string is within your capture of ChipScope to prevent possible data interference in MATLAB.

3. It will be helpful later in your MATLAB code if you rename your **Data Port variables**. Right click on the trig4/data4 and trig5/data5 Ports, and **change the names** to something more descriptive, such as `tx_i` and `tx_q` respectively. If needed, you can use the Simulink model to find which signal each port has.
4. Open the file menu and select **Export**.
5. Click the ASCII radio box and select Bus Plot Buses under Signals to export, then click export.
6. It is recommended that you save this file into the project directory with your MATLAB files. Call it something descriptive such as **TX.prn**.
7. To test the data in MATLAB, run your test bench script by first navigating to your project directory (this is necessary to load the prn file correctly).

---

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

---

8. Run the test bench with the sim parameter set to 0. You should see your message output with a plot in MATLAB of the qpsk waveform.



Experiment with other ASCII strings in your C code to verify the timing of the buffers. If you notice that your message is incomplete, it's possible you could be overloading your FIFO buffer before you have a chance to load your message to transmit.

---

## Appendix A MATLAB Core Functions

MATLAB function `qpsk_tx_byte2sym.m`

```

%#codegen
% this core runs at an oversampling rate of 8
function [d_out, re_byte_out, tx_done_out] = ...
    qpsk_tx_byte2sym(data_in, empty_in, clear_fifo_in, tx_en_in)

    OS_RATE = 8;
    SYM_PER_BYTE = 4; % number of symbols per byte (QPSK 4)
    tbi = TB_i;
    tbq = TB_q;
    CORE_LATENCY = 8;

    persistent count
    persistent symIndex
    persistent diLatch dqLatch
    persistent tx_fifo
    persistent wrCount rdCount
    persistent txDone
    persistent sentTrain
    persistent reBuf

    if isempty(count)
        count = 0;
        symIndex = 0;
        diLatch = 0; dqLatch = 0;
        wrCount = 0; rdCount = 0;
        txDone = 0;
        sentTrain = 0;
        reBuf = 0;
    end
    if isempty(tx_fifo)
        tx_fifo = zeros(1,1024); % internal tx buffer (1024 bytes)
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % if want to transmit a new packet reset variables
    if clear_fifo_in == 1
        wrCount = 0;
        txDone = 0;
        reBuf = 0;
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % we are ready to transmit some data
    rdIndex = wrCount-rdCount+1; % as rd decrements, index increments
    if rdIndex <= 0
        rdIndex = 1024;
    end
    data = tx_fifo(rdIndex); % get next byte of data
    d_out = 0; % initialize output

```

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

% fifo should be empty and the processor says go ahead and transmit
% we stop when we've written all the data out that we wrote to the fifo.
% This core doesn't care about packet length, just about how many bytes
% were written to the fifo.
PAD_BITS = 24;
if empty_in == 1 && tx_en_in == 1 && txDone == 0
    if sentTrain <= 65+PAD_BITS % Overhead bits
        if count == 0 && sentTrain <= PAD_BITS % sending pad bits
            diLatch = mod(sentTrain,2);
            diLatch = diLatch*2-1;
            dqLatch = diLatch;
        elseif count == 0 % sending header bits
            diLatch = tbi(sentTrain-PAD_BITS);
            dqLatch = tbq(sentTrain-PAD_BITS);
        end
        count = count + 1; % increment latency counter
        if count >= OS_RATE
            count = 0; % reset latency counter
            sentTrain = sentTrain + 1; % next bit
        end
    else % sending data!
        if mod(count,OS_RATE) == 0 % Latency check
            sym2 = symIndex*2;
            diLatch = mybitget(data,sym2+1)*2-1; % get next i bit
            dqLatch = mybitget(data,sym2+2)*2-1; % get next j bit
            symIndex = symIndex + 1; % next symbol index
        end
        count = count + 1; % increment latency counter
        if count >= OS_RATE*SYM_PER_BYTE
            count = 0; % reset latency counter
            symIndex = 0; % reset symbol index
            rdCount = rdCount - 1; % get next byte
        end
        if rdCount == 0 % when transmitted all bytes
            txDone = 1; % done transmitting
        end
    end
    d_out = complex(diLatch, dqLatch); % output i and q bit
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transfer data from processor to internal buffer. Because the core has a
% non-zero throughput we need to stale a bit for the requested data to make
% it to our input.
wrIndex = 1024;
re_byte_out = 0;
if empty_in == 0 && reBuf == 0
    reBuf = CORE_LATENCY; % initialize tx_fifo latency counter
    txDone = 0; % initialize done sending variable
    re_byte_out = 1; % read tx_fifo buffer line (active high)
end
if reBuf > 0
    reBuf = reBuf - 1; % decrement tx_fifo latency counter
end

```

## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```
if reBuf == 1
    wrCount = wrCount + 1; % total number of bytes written
    wrIndex = wrCount;    % offset used to write to end of tx_fifo
    rdCount = wrCount;    % track bytes (read) when transmitting
    reBuf = 0;            % maintain latency for reading from tx_fifo
    count = 0;            % maintain latency for tx packet
    sentTrain = 1;        % current byte location of packet overhead
end
tx_fifo(wrIndex) = data_in;
tx_done_out = txDone;
end
```

## Appendix B MATLAB Test Bench

MATLAB script `qpsk_tx_tb.m`

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialization and Model/simulation parameters %%%%%%%%%%
sim = 1;      % simulation param (1 is sim qpsk, 0 is loaded from chipscope)
OS_RATE = 8;
make_srrc_lut;
make_train_lut;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% load packet that was transmitted and captured from chipscope %%%%
if (~sim)
    xloadChipscopeData( 'tx.prn' );
    iFile = tx_i( 1:end )/2^11;
    qFile = tx_q( 1:end )/2^11;
    x = complex( iFile, qFile );
else
    % Emulate microprocessor packet creation %%%%%%%%%%
    % data payload creation
    messageASCII = 'Hello World!';
    message = double( unicode2native( messageASCII ) );

    % add on length of message to the front with four bytes
    msgLength = length( message );
    messageWithNumBytes = [ mod( msgLength, 2^8 ), ...
        mod( floor( msgLength/2^8 ), 2^8 ), ...
        mod( floor( msgLength/2^16 ), 2^8 ), 1, message ];

    % add two bytes at the end, which is a CRC
    messageWithCRC = CreateAppend16BitCRC( messageWithNumBytes );
    m1 = length( messageWithCRC );
    % FPGA radio transmit core%%%%%%%%%
    data_in = 0;
    empty_in = 1;
    tx_en_in = 0;
    numBytesFromFifo = 0;
    num_samp = m1*8*2*2*3;
    x = zeros( 1, num_samp );
    CORE_LATENCY = 4;
    data_buf = zeros( 1, CORE_LATENCY );
    empty_buf = ones( 1, CORE_LATENCY );
    clear_buf = zeros( 1, CORE_LATENCY );
    tx_en_buf = zeros( 1, CORE_LATENCY );
    for i1 = 1:num_samp
        % first thing the processor does is clear the internal tx fifo
        if i1==1
            clear_fifo_in = 1;
        else
            clear_fifo_in = 0;
        end
    end
end

```



## Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

        if il==5 % wait a little bit then begin to load the fifo
            empty_in = 0;
            numBytesFromFifo = 0;
        end
        data_buf = [ data_buf( 2:end ), data_in ];
        empty_buf = [ empty_buf( 2:end ), empty_in ];
        clear_buf = [ clear_buf( 2:end ), clear_fifo_in ];
        tx_en_buf = [ tx_en_buf( 2:end ), tx_en_in ];
        [i_out,q_out,re_byte_out,tx_done_out] = qpsk_tx( data_buf( 1 ),...
            empty_buf( 1 ), clear_buf( 1 ), tx_en_buf( 1 ) );
        x_out = complex( i_out, q_out )/2^11;
        x( il ) = x_out;
        %% Emulate read FIFO AXI interface
        if re_byte_out==1 && numBytesFromFifo<length( messageWithCRC )
            data_in = messageWithCRC( numBytesFromFifo + 1 );
            numBytesFromFifo = numBytesFromFifo + 1;
        end
        % processor loaded all bytes into FIFO so begin transmitting
        if numBytesFromFifo==length( messageWithCRC )
            empty_in = 1;
            tx_en_in = 1;
        end
    end
end
index = find( abs( x )>sum( SRRRC ) );
% constant is pad bits
offset = index( 1 ) + (24*OS_RATE) + 6 + length( TB_i )*OS_RATE;
if (~sim)
    idx = offset:OS_RATE:index(end)-(OS_RATE*4*3);
else
    idx = offset:OS_RATE:index(end);
end
% seperate the channels
y = x( idx );
sc = zeros( 1, 2*length(y));
sc( 1:2:end ) = real( y );
sc( 2:2:end ) = imag( y );
sh = sign( sc );
sb = (sh + 1)/2;
d = zeros( 1, length(y)/4 );
% convert the data to decimal numbers
for il = 1:length(y)/4
    si = sb( 1 + (il - 1)*8:i1*8 );
    d( il ) = bi2de( round(si) );
end
% plot i and q channels and display the recovered messeage
figure( 1 )
clf
plot( real( x ), 'red' )
hold on
plot( imag( x ), 'green' )
title( 'Transmit samples' );
disp('Your message was');
disp(native2unicode(d(5:end-2)));

```