

Toyon Research Corporation

Lab 3: Output QPSK

Chilipepper Tutorial Projects

Table of Contents

Introduction	4
Procedure.....	4
Objectives	4
Generate HDL code	5
1.1 DAC and MCU Driver MATLAB Files	5
1.2 QPSK TX MATLAB Functions.....	5
1.3 TX_FIFO MATLAB Function	9
1.4 QPSK TX Test Bench Scripts.....	10
1.5 HDL Coder Project	11
Configure Cores and Export Design	17
2.1 Needed IP Cores	17
2.2 Configuring the DAC Driver Port	18
2.3 Configuring the MCU Driver Port	18
2.4 Configuring the MCU UART.....	18
2.5 Configuring the QPSK TX.....	18
2.6 Configuring the TX FIFO	19
2.7 Configuring the LEDs GPIO Port	19
2.8 Configuring the Switch GPIO Port	19
2.9 Configuring the Button GPIO Port	19
2.10 Configuring the Clock Generator IP Core.....	20
2.11 Pin Assignments.....	21
Create software project	23
3.1 Creating a new C Project.....	23
3.2 Programming the Board.....	24
3.3 Debugging with SDK.....	25
Testing and Design Verification.....	26
4.1 Verification with another Chilipepper Board	26

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

4.2	Verification with ChipScope.....	26
4.3	MATLAB Analysis.....	28
	Appendix A MATLAB byte2sym Function	30
	Appendix B MATLAB TX FIFO Function	33
	Appendix C MATLAB Test Bench script.....	35
	Appendix D SDK source file.....	38

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 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:

- 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 Driver 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.

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

    persistent blinky_cnt

    if isempty(blinky_cnt)
        blinky_cnt = 0;
    end

    [byte_i_out, byte_q_out, request_byte, tx_done] = ...
        qpsk_tx_byte2sym(data_in, empty_in, clear_fifo_in, tx_en_in);

    byte_out = complex(byte_i_out, byte_q_out);

    [d_ssrc] = qpsk_srrc(byte_out);

    % 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);
    tx_done_out = tx_done;

    blinky_cnt = blinky_cnt + 1;
    if blinky_cnt == 20000000
        blinky_cnt = 0;
    end
    blinky = floor(blinky_cnt/10000000);
end
```

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

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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 directory for the project under `C:\QPSK_Projects\Project_3`.
2. Create a MATLAB directory within the main project directory.
3. 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
4. Save this function as `qpsk_tx_byte2sym.m` inside the MATLAB 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.

```

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.

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.

i channel		(iq) Bit Symbol	Symbol Phase
Bit value	NRZ Value		
1	1	(11)	45°
0	-1	(01)	135°
q channel		(00)	225°
Bit value	NRZ Value	(10)	315°
1	1		
0	-1		

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

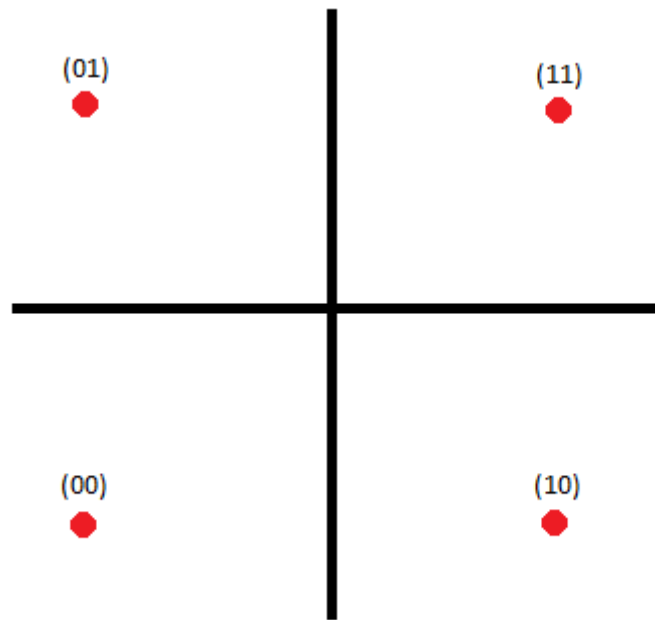


Figure 1-3: Scatter plot of the QPSK symbols.

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

```
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, '%%#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.

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.

7. Create 2 new **MATLAB function** with the contents of Figure 1-4 and 1-5.
8. **Save** these functions as `qpsk_srrc.m` and `make_srrc_lut.m` respectively inside the MATLAB 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 Lab¹.

9. 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 MATLAB directory.

¹ Can be found at https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/Labs/Lab_3/MATLAB

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 MATLAB 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.

```
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 1-6: MATLAB function to append a 16 Bit CRC to the QPSK packet.

3. Create a new **MATLAB function** with the contents of Figure 1-7.
4. **Save** this function as `CreateAppend16BitCRC.m` inside the MATLAB 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 are 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.

Variables	Function Replacements	Type Validation Output					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
<code>clear_fifo_in</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>
<code>data_in</code>	double	0	228			Yes	<code>numerictype(0, 8, 0)</code>
<code>empty_in</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>
<code>tx_en_in</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>
Output							
<code>blinky</code>	double	0	0			Yes	<code>numerictype(0, 1, 0)</code>
<code>i_out</code>	double	-1493	1493			Yes	<code>numerictype(1, 12, 0)</code>
<code>q_out</code>	double	-1493	1493			Yes	<code>numerictype(1, 12, 0)</code>
<code>request_byte</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>
<code>tx_done_out</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>
Persistent							
<code>blinky_cnt</code>	double	0	1728			Yes	<code>numerictype(0, 25, 0)</code>
Local							
<code>byte_i_out</code>	double	-1	1			Yes	<code>numerictype(1, 2, 0)</code>
<code>byte_out</code>	complex double	-1	1			Yes	<code>numerictype(1, 2, 0)</code>
<code>byte_q_out</code>	double	-1	1			Yes	<code>numerictype(1, 2, 0)</code>
<code>d_src</code>	complex double	-0.73	0.73			No	<code>numerictype(1, 13, 12)</code>
<code>tx_done</code>	double	0	1			Yes	<code>numerictype(0, 1, 0)</code>

Figure 1-7: Variable types for `qpsk_tx` function

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

Variables	Function Replacements	Type Validation Output					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
clear_fifo_in	double	0	1			Yes	numerictype(0, 1, 0)
data_in	double	0	228			Yes	numerictype(0, 8, 0)
empty_in	double	0	1			Yes	numerictype(0, 1, 0)
tx_en_in	double	0	1			Yes	numerictype(0, 1, 0)
▲ Output							
d_i_out	double	-1	1			Yes	numerictype(1, 2, 0)
d_q_out	double	-1	1			Yes	numerictype(1, 2, 0)
re_byte_out	double	0	1			Yes	numerictype(0, 1, 0)
tx_done_out	double	0	1			Yes	numerictype(0, 1, 0)
▲ Persistent							
count	double	0	32			Yes	numerictype(0, 6, 0)
diLatch	double	-1	1			Yes	numerictype(1, 2, 0)
dqLatch	double	-1	1			Yes	numerictype(1, 2, 0)
rdCount	double	0	18			Yes	numerictype(0, 11, 0)
reBuf	double	0	8			Yes	numerictype(0, 4, 0)
sentTrain	double	0	90			Yes	numerictype(0, 8, 0)
symIndex	double	0	4			Yes	numerictype(0, 3, 0)
txDone	double	0	1			Yes	numerictype(0, 1, 0)
tx_fifo	1 x 1024 double	0	228			Yes	numerictype(0, 8, 0)
wrCount	double	0	18			Yes	numerictype(0, 11, 0)
▲ Local							
CORE_LAT...	double	8	8			Yes	numerictype(0, 4, 0)
OS_RATE	double	8	8			Yes	numerictype(0, 4, 0)
PAD_BITS	double	24	24			Yes	numerictype(0, 5, 0)
SYM_PER_B...	double	4	4			Yes	numerictype(0, 3, 0)
data	double	0	228			Yes	numerictype(0, 8, 0)
rdIndex	double	1	19			Yes	numerictype(0, 11, 0)
sym2	double	0	6			Yes	numerictype(0, 3, 0)
tbi	65 x 1 double	-1	1			Yes	numerictype(1, 2, 0)
tbq	65 x 1 double	-1	1			Yes	numerictype(1, 2, 0)
wrIndex	double	1	1024			Yes	numerictype(0, 11, 0)

Figure 1-8: Variable types for the qpsk_tx_byte2sym function

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

- 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

Set the target device and synthesis tool

Workflow:

Platform: [Launch board manager](#)

Synthesis tool: [Refresh list](#)

Chip family: Device:

Package: Speed:

IP core settings

Name: Version:

Processor/FPGA synchronization:

[?](#)

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

- 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.
- 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.

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

Ports			
Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
▲ Inport			
data_in	numerictype(0, 8, 0)	External Port	
empty_in	numerictype(0, 1, 0)	External Port	
clear_fifo_in	numerictype(0, 1, 0)	AXI4-Lite	x"100"
tx_en_in	numerictype(0, 1, 0)	AXI4-Lite	x"104"
▲ Outport			
i_out	numerictype(1, 12, 0)	External Port	
q_out	numerictype(1, 12, 0)	External Port	
tx_done_out	numerictype(0, 1, 0)	AXI4-Lite	x"108"
request_byte	numerictype(0, 1, 0)	External Port	
blinky	numerictype(0, 1, 0)	External Port	

Figure 1-10: Port Interface settings for qpsk_tx HDL Coder project

- 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>.
- Repeat this process for the tx_fifo function. **Name the PCore** tx_fifo_pcore and 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.

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

Variables	Function Replacements	Type Validation Output					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
byte_in	double	0	228			Yes	numerictype(0, 8, 0)
get_byte	double	0	1			Yes	numerictype(0, 1, 0)
reset_fifo	double	0	0			Yes	numerictype(0, 1, 0)
store_byte	double	0	1			Yes	numerictype(0, 1, 0)
▲ Output							
byte_received	double	0	1			Yes	numerictype(0, 1, 0)
bytes_available	double	0	1			Yes	numerictype(0, 10, 0)
dout	double	0	228			Yes	numerictype(0, 8, 0)
empty	double	0	1			Yes	numerictype(0, 1, 0)
▲ Persistent							
byte_out	double	0	228			Yes	numerictype(0, 8, 0)
fifo	1 x 1024 double	0	228			Yes	numerictype(0, 8, 0)
handshake	double	0	1			Yes	numerictype(0, 1, 0)
head	double	1	19			Yes	numerictype(0, 11, 0)
tail	double	2	20			Yes	numerictype(0, 11, 0)
▲ Local							
full	double	0	0			Yes	numerictype(0, 1, 0)

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

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
▲ Inport			
reset_fifo	numerictype(0, 1, 0)	AXI4-Lite	x"100"
store_byte	numerictype(0, 1, 0)	AXI4-Lite	x"104"
byte_in	numerictype(0, 8, 0)	AXI4-Lite	x"108"
get_byte	numerictype(0, 1, 0)	External Port	
▲ Outport			
dout	numerictype(0, 8, 0)	External Port	
bytes_available	numerictype(0, 10, 0)	AXI4-Lite	x"10C"
byte_received	numerictype(0, 1, 0)	AXI4-Lite	x"110"
empty	numerictype(0, 1, 0)	External Port	

Figure 1-12: Port settings for TX FIFO PCore

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

9. 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.
10. 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.

Name	Net
⊕ External Ports	
⊕ axi_interconnect_1	
⊕ processing_system7_0	
⊕ axi_gpio_button	
⊕ axi_gpio_led	
⊕ axi_gpio_switch	
⊕ mcu_uart	
⊕ dac_driver	
⊕ mcu_driver	
⊕ qpsk_tx	
⊕ tx_fifo	
⊕ tx_clock_generator	

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.

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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.

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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**, 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.

2. **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

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

- RST → net_gnd
- LOCKED → External Port

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

tx_clock_generator		
CLKIN	External Ports::tx_clock_generator_CLKIN_pin	tx_clock_generator_CLKIN
CLKOUT0	mcu_driver::IPCORE_CLK	tx_clock_generator_CLKOUT0
	tx_fifo::IPCORE_CLK	
	qpsk_tx::IPCORE_CLK	
CLKOUT1	dac_driver::IPCORE_CLK	tx_clock_generator_CLKOUT1
CLKOUT2	External Ports::tx_clock_generator_rx_clk_pin	tx_clock_generator_CLKOUT2
CLKOUT3	External Ports::tx_clock_generator_tx_clk_pin	tx_clock_generator_CLKOUT3
RST	net_gnd	net_gnd
LOCKED	External Ports::tx_clock_generator_LOCKED_pin	tx_clock_generator_LOCKED

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` 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.

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```
##### 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;
#####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;
#####Tx – FMC interface at 2.5V #####
NET dac_driver_rx_iq_sel_pin              LOC = B16 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[0]                  LOC = A18 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[1]                  LOC = A19 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[2]                  LOC = E20 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[3]                  LOC = G21 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[4]                  LOC = F19 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[5]                  LOC = G15 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[6]                  LOC = E19 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[7]                  LOC = G16 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[8]                  LOC = G19 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[9]                  LOC = A16 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_pin[10]                 LOC = A17 | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_rxd_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 tx_clock_generator_LOCKED_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 qpsk_tx_blinky_pin                      LOC = W22 | IOSTANDARD = LVCMOS33; # "LD5"
##### Switches#####
NET axi_gpio_switch_GPIO_IO_I_pin           LOC = F22 | IOSTANDARD = LVCMOS33; # "SW0"
##### Buttons#####
NET axi_gpio_button_GPIO_IO_I_pin           LOC = P16 | IOSTANDARD = LVCMOS33; # "BTCenter"
```

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 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 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 qpsk_transmit project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the qpsk_transmit 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 repository² 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 `MCU_DRIVER`, `DAC_DRIVER`, `TX_PCORE` and `TX_FIFO`.

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.

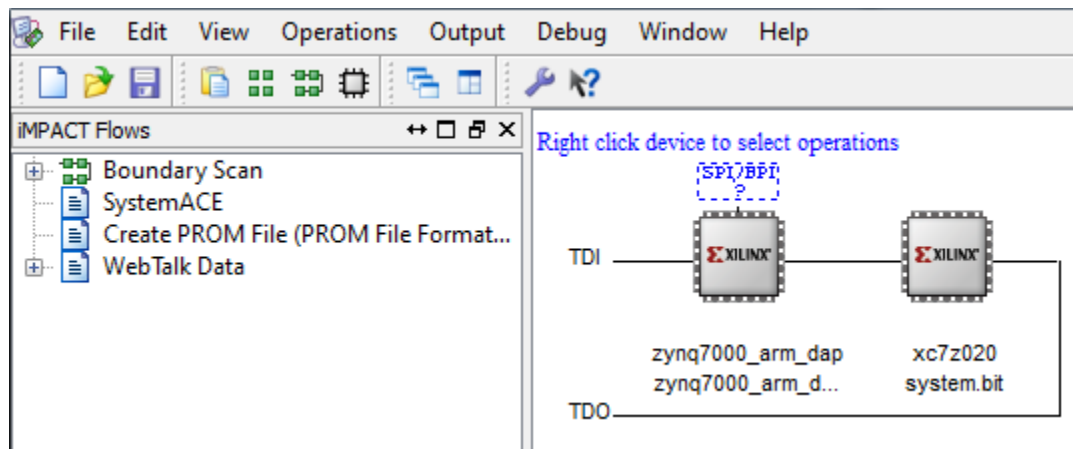
9. (Optional) Click on **Project → Properties**. Open the **C/C++ Build** arrow and click the settings option. Under **ARM gcc linker**, click the Libraries folder. Click the  button, type the letter **m** into the prompt and select ok. **Apply** and hit ok.

² 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*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 3-1 below.



3-1: 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.

³ Can be found at https://github.com/Toyon/Chilipepper/tree/master/QPSK_Radio/DemoFilesAndDocumentation

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

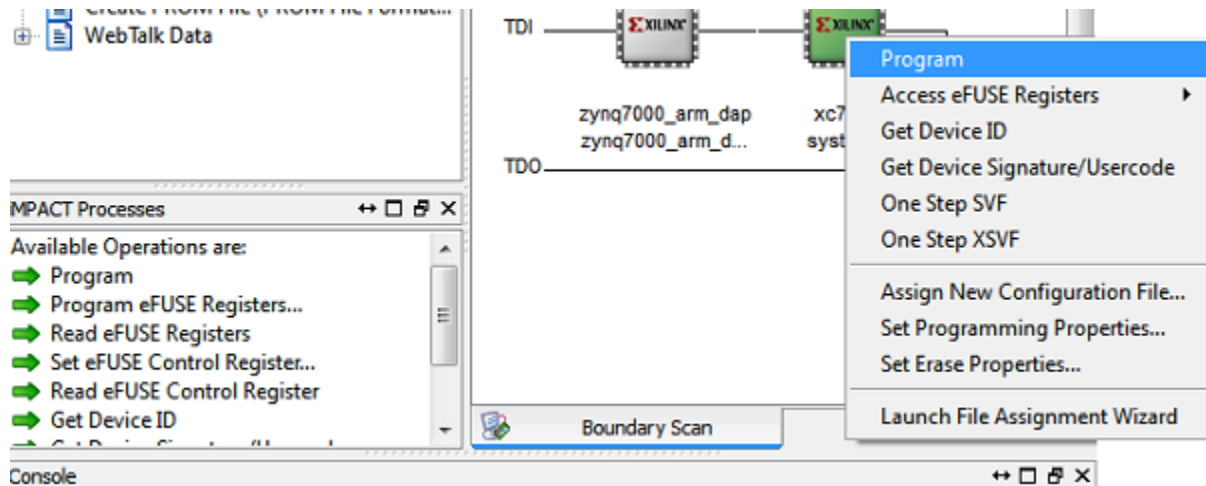


Figure 3-2: iMPACT configuration screen

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 `qpsk_transmit` 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 website⁴. 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.

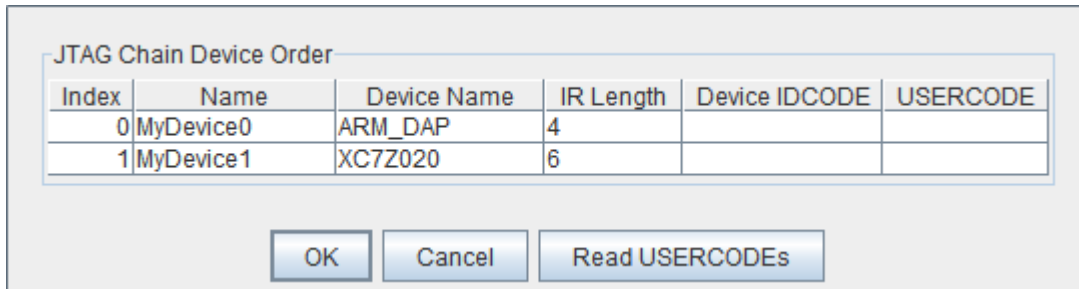
4.2 Verification with ChipScope

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.

⁴ https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/Labs/Lab_8/DemoFilesAndDocumentation/BOOT

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper



Index	Name	Device Name	IR Length	Device IDCODE	USERCODE
0	MyDevice0	ARM_DAP	4		
1	MyDevice1	XC7Z020	6		

OK Cancel Read USERCODEs

Note

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.

5. **(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.
6. Select ok to get to the Analyzer main screen. Open the file menu and select **Import**.
7. 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.
8. 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.
9. 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

10. Open the file menu and select **Export**.
11. Click the ASCII radio box and change the Signal to Export to the Bus Plot. Then click export.
12. 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**.

Note

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.

4.3 MATLAB Analysis

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

```
fid = fopen('TX.prn');
M = textscan(fid, '%d %d %d %d', 'Headerlines', 1);
fclose(fid);
i_out = double(M{3})';
q_out = double(M{4})';
plot(1:1048, i_out, 1:2048, q_out)
```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

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 “Transmitted message correctly” when you run qpsk_tb.

Appendix A MATLAB byte2sym Function

MATLAB function `qpsk_tx_byte2sym.m`

```

%#codegen
% this core runs at an oversampling rate of 8
function [d_i_out, d_q_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);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% if want to transmit a new packet reset things
if clear_fifo_in == 1
    wrCount = 0;
    txDone = 0;
    reBuf = 0;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% we are ready to transmit some data
rdIndex = wrCount-rdCount+1;
if rdIndex <= 0
    rdIndex = 1024;
end
data = tx_fifo(rdIndex);

```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

d_i_out = 0;
d_q_out = 0;
% 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 got
% written to the fifo.
PAD_BITS = 24;
if empty_in == 1 && tx_en_in == 1 && txDone == 0
    if sentTrain <= PAD_BITS
        if count == 0
            diLatch = mod(sentTrain,2)*2-1;
            dqLatch = mod(sentTrain,2)*2-1;
        end
        count = count + 1;
        if count >= OS_RATE
            count = 0;
            sentTrain = sentTrain + 1;
        end
        d_i_out = diLatch;
        d_q_out = dqLatch;
    elseif sentTrain <= 65+PAD_BITS
        if count == 0
            diLatch = tbi(sentTrain-PAD_BITS);
            dqLatch = tbq(sentTrain-PAD_BITS);
        end
        count = count + 1;
        if count >= OS_RATE
            count = 0;
            sentTrain = sentTrain + 1;
        end
        d_i_out = diLatch;
        d_q_out = dqLatch;
    else
        if mod(count,OS_RATE) == 0
            sym2 = symIndex*2;
            diLatch = mybitget(data,sym2+1)*2-1;
            dqLatch = mybitget(data,sym2+2)*2-1;
            symIndex = symIndex + 1;
        end
        d_i_out = diLatch;
        d_q_out = dqLatch;

        count = count + 1;
        if count >= OS_RATE*SYM_PER_BYTE
            count = 0;
            symIndex = 0;
            rdCount = rdCount - 1;
        end
        if rdCount == 0
            txDone = 1;
        end
    end
end
end

```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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. So, I'm doing that we reBuf
% counter. There are definitely more efficient ways to do this but I'm
% gonna leave that for another day.
wrIndex = 1024;
re_byte_out = 0;
if empty_in == 0 && reBuf == 0
    reBuf = CORE_LATENCY;
    txDone = 0;
    re_byte_out = 1;
end
if reBuf > 0
    reBuf = reBuf - 1;
end
if reBuf == 1
    wrCount = wrCount + 1; %total number of bytes to send out
    wrIndex = wrCount;
    rdCount = wrCount;
    reBuf = 0;
    count = 0;
    sentTrain = 1;
end
tx_fifo(wrIndex) = data_in;

tx_done_out = txDone;
```


Appendix B MATLAB TX FIFO Function

MATLAB function `tx_fifo.m`

```
function [dout, bytes_available, byte_received, empty] = ...
    tx_fifo(reset_fifo, store_byte, byte_in, get_byte)
%
% First In First Out (FIFO) structure.
% This FIFO stores integers.
% The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;

% handshaking logic
byte_received = 0;
if store_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_received = 1;
end

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

%%%%%%%%%%%%get%%%%%%%%%%%%%%%%%%%%%%%%
if (get_byte && ~empty)
    head = head + 1;
    if head == 1025
        head = 1;
    end
    byte_out = fifo(head);
end
```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```
#####put#####
if (store_byte && ~full && handshake == 0)
    fifo(tail) = byte_in;
    tail = tail + 1;
    if tail == 1025
        tail = 1;
    end
    byte_received = 1;
    handshake = 1;
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
```

Appendix C MATLAB Test Bench script

MATLAB script qpsk_tx_tb.m

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Model/simulation parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
OS_RATE = 8;
SNR = 100;
fc = 10e3/20e6; % sample rate is 20 MHz, top is 10 kHz offset
muFOC = floor(.01*2^12)/2^12;
muTOC = floor(.01*8*2^12)/2^12;
sim = 1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize LUTs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
make_srrc_lut;
make_trig_lut;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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 ID
    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;
store_byte = 0;
numBytesFromFifo = 0;
num_samp = m1*8*2*2*3;
x = zeros(1,num_samp);

```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

CORE_LATENCY = 4;
data_buf = zeros(1,CORE_LATENCY);
store_byte_buf = zeros(1,CORE_LATENCY);
clear_buf = zeros(1,CORE_LATENCY);
tx_en_buf = zeros(1,CORE_LATENCY);
re_byte_out(1) = 0;
reset_fifo = 0;
byte_request = 0;
for il = 1:num_samp
    % first thing the processor does is clear the internal tx fifo
    if il == 1
        clear_fifo_in = 1;
    else
        clear_fifo_in = 0;
    end

    data_buf = [data_buf(2:end) data_in];
    store_byte_buf = [store_byte_buf(2:end) store_byte];
    clear_buf = [clear_buf(2:end) clear_fifo_in];
    tx_en_buf = [tx_en_buf(2:end) tx_en_in];

    [new_data_in, bytes_available, byte_recieved, empty_in] = ...
    tx_fifo(reset_fifo, store_byte_buf(1), data_buf(1), byte_request);

    [i_out, q_out, tx_done_out, request_byte] = ...
    qpsk_tx(new_data_in,empty_in,clear_buf(1),tx_en_buf(1));
    x_out = complex(i_out,q_out)/2^11;
    x(il) = x_out;
    byte_request = request_byte;

    %% Emulate write to FIFO interface
    if mod(il,8) == 1 && numBytesFromFifo < length(messageWithCRC)
        data_in = messageWithCRC(numBytesFromFifo+1);
        numBytesFromFifo = numBytesFromFifo + 1;
    end
    %% Software lags a bit on the handshaking signals %%
    if (0 < mod(il,8) && mod(il,8) < 5) && tx_en_in == 0
        store_byte = 1;
    else
        store_byte = 0;
    end
    % processor loaded all bytes into FIFO so begin transmitting
    if (numBytesFromFifo == length(messageWithCRC) && mod(il,8) > 5)
        empty_in = 1;
        tx_en_in = 1;
    end
end
if ~sim % load data that was transmitted and captured from chipscope
    if 1
        fid = fopen('tx.prn');
        M = textscan(fid,'%d %d %d %d %d %d %d %d %d %d','Headerlines',1);
        fclose(fid);
        iFile = double(M{3})'/2^11;
        qFile = double(M{4})'/2^11;
    end
end

```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

else
    M = load('dac.prn');
    if M(1,end-1) == 0
        iFile = M(1:2:end,end)'/2^11;
        qFile = M(2:2:end,end)'/2^11;
    else
        qFile = M(1:2:end,end)'/2^11;
        iFile = M(2:2:end,end)'/2^11;
    end
end
x = complex(iFile,qFile);
end

index = find(abs(x) > sum(SRRC))+24*8; % constant is pad bits
offset = index(1)+6+length(TB_i)*OS_RATE;
idx = offset:8:(offset+8*ml*4-1);
y = x(idx); % four symbos per byte of data
sc = zeros(1,18*8);
sc(1:2:end) = real(y);
sc(2:2:end) = imag(y);
sh = sign(sc);
sb = (sh+1)/2;
d = zeros(1,ml);
for i1 = 1:ml
    si = sb(1+(i1-1)*8:i1*8);
    s1 = [];
    for i2 = 1:length(si)
        s1 = [s1 num2str(round(si(i2)))];
    end
    d(i1) = bin2dec(fliplr(s1));
end
figure(1)
clf
plot(real(x))
hold on
plot(idx,real(y),'ro')
title('Transmit samples');

error_tx = sum(abs(d-messageWithCRC));
if error_tx == 0
    disp('Transmitted message correctly');
else
    disp('Transmitted message incorrectly');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Appendix D SDK source file

Xilinx SDK file `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;

// function declarations
int DebouncButton( void );
int SetupPeripherals( void );

int main()
{
    int i1, sw;
    int aliveLed = 0, statusLed = 0, blinkCounter = 0;

    init_platform();

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

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

    // by default we are in transmit
    Chilipepper_SetPA( 1 );
    Chilipepper_SetTxRxSw( 0 ); // 0- transmit, 1-receive

    while (1)
    {
        // flip the LED1 so the user knows the processor is alive
        blinkCounter += 1;
        if (blinkCounter > 200000)
        {
            aliveLed = ~aliveLed;
            blinkCounter = 1;
            XGpio_DiscreteWrite(&gpio_blinky, 2, aliveLed);
        }
        sw = XGpio_DiscreteRead(&gpio_sw_test, 1);
        switch (sw)
        {
```

Creating Wireless Transceivers Using MATLAB to HDL translation and Toyon Chilipepper

```

    case 0: // 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 );
            }
        }
        break;
    case 1: // initiate packet transmission with a button press
        if (DebouncButton() == 0)
            break;

        statusLed = ~statusLed;
        XGpio_DiscreteWrite(&gpio_blinky, 1, statusLed);
        Chilipepper_WriteTestPacket( 1 );
        break;
    default:
        break;
}
}
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;

    // setup LEDs
    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);

    // setup Switch
    XGpio_Initialize(&gpio_sw_test, XPAR_AXI_GPIO_SWITCH_DEVICE_ID);
    XGpio_SetDataDirection(&gpio_sw_test, 1, 1);

    //setup Button
    XGpio_Initialize(&gpio_btn, XPAR_AXI_GPIO_BUTTON_DEVICE_ID);
    XGpio_SetDataDirection(&gpio_btn, 1, 1);

    return XST_SUCCESS;

    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;
}
```