Toyon Research Corporation

Lab 4: DC Offset Correction

Chilipepper Tutorial Projects

Version 1.2

1/10/2014

Table of Contents

Lab 4: DC Offset Correction

Introduction

This lab will show you the first step in the process to recover your QPSK Waveform generated in Lab 3 using a second Xilinx Zed Board FPGA and Toyon Chilipepper FMC. The Analog to Digital Conversion (ADC) used to receive the signal will take place on the Chilipepper board. The FMC initialization and microcontroller (MCU) signal control will be handled in software using the Xilinx Software Development Kit (SDK). Finally, verification of the received signal will be done using ChipScope. In future labs, we will increase this verification gradually using 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:

- Implement DC Offset correction for a received Waveform
- Receive 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 ADC 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 an ADC Driver to deinterleave our signal before processing 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 these PCores if needed.

1.2 DC Offset MATLAB Function

The purpose of the DC Offset Correction within this lab is to remove any unwanted DC signal component which may prevent proper demodulation of the waveform. If the DC offset of the received signal is very high (higher than the signal content), then it is possible that the receiver gain was reduced to prevent saturation of the signal. For this reason, it is also the job of the DC Offset Correction core to adjust the receiver gain as needed. Since the data coming from the ADC Driver is held for 2 clock cycles (to deinterleave the rxd data), the DC Offset core should be clocked at one half the rate of the ADC Driver core. The MATLAB function used to create the DC Offset PCore is shown in Appendix A.

1. Create a directory for the project under C:\QPSK_Projects\Lab_4.
2. Create a MATLAB directory within the main project directory.
3. Create a new **MATLAB function** with the contents of Appendix A.
4. Save this function as `dc_offset_correction.m` inside the MATLAB directory.

1.3 MATLAB Test Bench

Now that you have created the code needed to correct the DC offset, we also need to create a test bench script to test the algorithm. This is done by observing the output graph of the result, which in this case is the signal I and q channels with their corrected means, as well as a plot of the RSSI output. To accomplish this, it is necessary to have a "test" signal from the ADC to use for the analysis; Therefore in addition to this script, you will need a signal exported from ChipScope with the ADC output for both the i and q channels. You can either use the ChipScope data obtained from Lab 2, or download the DC.prn file provided on the GitHub Repo[1]. The code for the test bench can be found in Appendix B

[1] Found at https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/Labs/Lab_4/MATLAB

Note	If you use the ChipScope data exported from Lab 2 as your ADC signal input you may need to modify the Test Bench script file to correctly load the variables you exported. Run "help textscan" to get more information on how to change this line to load data from your ChipScope Export.

1. Create a new **MATLAB script** with the contents of Appendix B.
2. Save this function as `dc_offset_correction_tb.m` inside the MATLAB project directory

1.4 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 ADC Driver PCores, thus the only core we need to create for this lab is the dc_offset PCore. Using the same steps outlined in the previous labs, create a new HDL coder project called `dc_offset_pcore`. Add both your `dc_offset_correction.m` file and your `dc_offset_correction_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. Modify your HDL Coder design to match the following Fixed-Point conversions

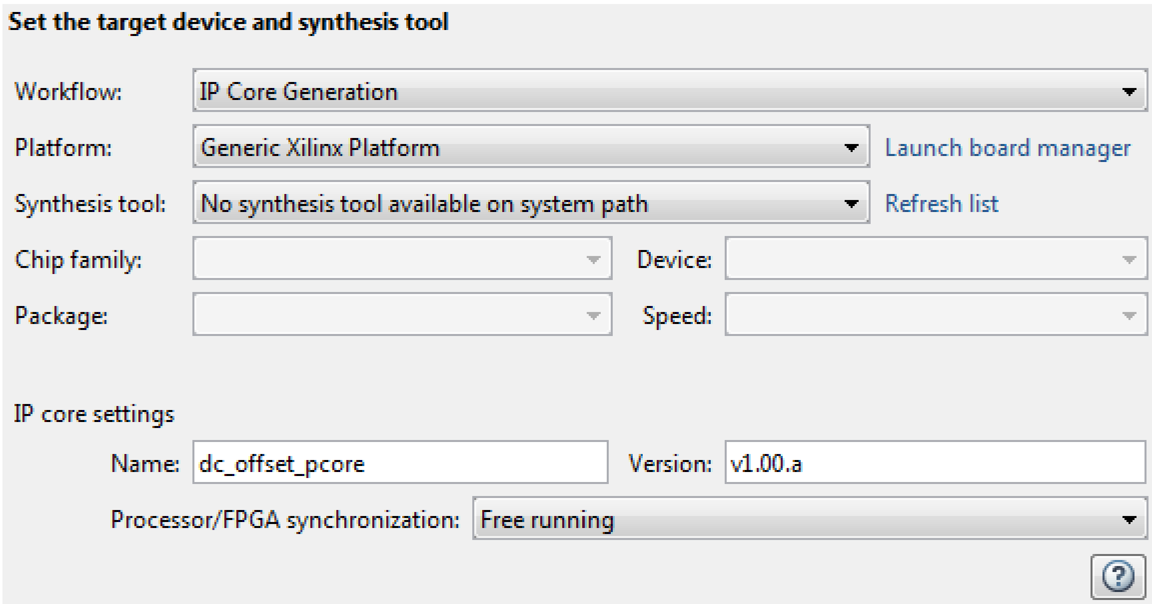
Variables | Function Replacements | Type Validation Output

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
gain_en_in	double	0	1			Yes	numerictype(0, 1, 0)
i_in	double	-274	-224			Yes	**numerictype(1, 12, 0)**
q_in	double	-220	-165			Yes	**numerictype(1, 12, 0)**
rssi_high_goal_in	double	1500	1500			Yes	**numerictype(0, 24, 0)**
rssi_low_goal_in	double	500	500			Yes	**numerictype(0, 24, 0)**
rx_en_in	double	0	1			Yes	numerictype(0, 1, 0)
Output							
blinky	double	0	0			Yes	numerictype(0, 1, 0)
dir_en_out	double	0	1			Yes	numerictype(0, 1, 0)
dir_out	double	0	2			Yes	numerictype(0, 2, 0)
i_out	double	-274	-205.19			No	**numerictype(1, 12, 0)**
q_out	double	-196	-42.65			No	**numerictype(1, 12, 0)**
rssi_en_out	double	0	1			Yes	numerictype(0, 1, 0)
rssi_out	double	0	94674			Yes	**numerictype(0, 24, 0)**
Persistent							
blinky_cnt	double	0	8192			Yes	**numerictype(0, 25, 0)**
counter	double	0	256			Yes	numerictype(0, 9, 0)
dir_state	double	0	1			Yes	numerictype(0, 1, 0)
i_dc	double	-50.01	0			No	**numerictype(1, 24, 12)**
i_mean	double	-184.25	0			No	**numerictype(1, 24, 12)**
noise_dec	double	0	2			Yes	**numerictype(0, 20, 0)**
noise_inc	double	0	11			Yes	**numerictype(0, 20, 0)**
noise_offset	double	0	140			Yes	**numerictype(0, 20, 0)**
q_dc	double	-133.73	0			No	**numerictype(1, 24, 12)**
q_mean	double	-133.73	0			No	**numerictype(1, 24, 12)**
rssiHold	double	0	94674			Yes	**numerictype(0, 24, 0)**
rssi_sum	double	0	24236538.99			No	**numerictype(0, 32, 0)**
Local							
ai	double	224	274			Yes	**numerictype(0, 11, 0)**
alpha	double	0	0			No	**numerictype(0, 12, 12)**
aq	double	165	220			Yes	**numerictype(0, 11, 0)**
rssi_diff	double	0	94305.73			No	**numerictype(0, 24, 0)**
rssi_inst	double	52913.85	104229			No	**numerictype(0, 23, 0)**

Figure 1-1: Variable types for dc_offset_correction function

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

Therefore you can leave the default settings. Ensure your Workflow settings resemble figure 1-4 below



1-2: Settings for Xilinx Zed Board HDL Coder Design

5. Just below the synthesis tool settings, **rename your PCore** to `dc_offset_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.

6. 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.

Ports

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
◢ Inport			
i_in	numerictype(1, 12, 0)	External Port	
q_in	numerictype(1, 12, 0)	External Port	
gain_en_in	numerictype(0, 1, 0)	AXI4-Lite	x"100"
rssi_low_goal_in	numerictype(0, 24, 0)	AXI4-Lite	x"104"
rssi_high_goal_in	numerictype(0, 24, 0)	AXI4-Lite	x"108"
rx_en_in	numerictype(0, 1, 0)	AXI4-Lite	x"10C"
◢ Outport			
i_out	numerictype(1, 12, 0)	External Port	
q_out	numerictype(1, 12, 0)	External Port	
rssi_out	numerictype(0, 24, 0)	AXI4-Lite	x"110"
rssi_en_out	numerictype(0, 1, 0)	AXI4-Lite	x"114"
dir_out	numerictype(0, 2, 0)	AXI4-Lite	x"118"
dir_en_out	numerictype(0, 1, 0)	AXI4-Lite	x"11C"
blinky	numerictype(0, 1, 0)	External Port	

Figure 1-3: Port Interface settings for the dc offset correction HDL Coder project

7. 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>.

8. Once the PCore has 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.

9. 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

- ADC Driver
- MCU Driver
- MCU UART
- DC Offset
- Clock Generator (one for RX and one for TX)
- Processing System
- AXI Interconnect

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

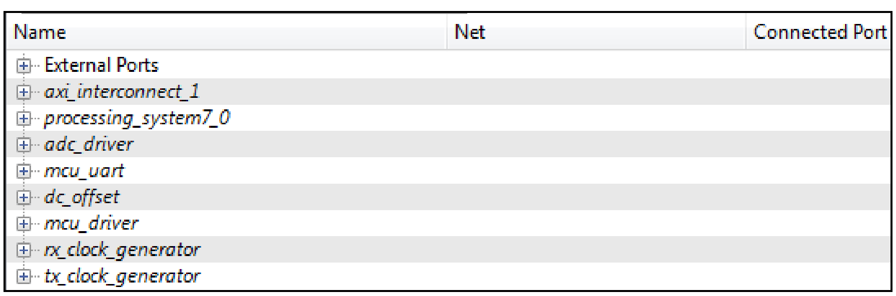


Figure 2-1: EDK project ports list

2.2 Configuring the ADC Driver Port

Expand the **ADC Driver** port. There are 6 individual I/O pins which need to be routed on this port.

1. First we will configure the `rx_iq_sel`, the `rxd` and the `blinky` pins. Each of these pins can be assigned as **External ports**.
2. Next are the `rx_i` and the `rx_q` output pins. Connect these pins to the `i_in` and `q_in` pins of the dc_offset PCore.
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 2.5**

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 DC Offset

Expand the **DC Offset** core. There are 7 individual I/O pins which need to be routed on this core.

1. If the ADC driver was previously configured correctly, the i_in and q_in pins of the dc_offset core should already be set.
2. The i_out and q_out pins will be connected to ChipScope for MATLAB Analysis. They can be left unconnected for now.
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 Clock Generator IP Core

The TX Clock Generator is used in this project to distribute the appropriate clock signals to each of the PCores required for Chilipepper initialization, 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 3 other devices; 1 PCore (MCU 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 DAC 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**, 180 Phase, **PLLE0** group and **Buffered True**
   - CLKOUT2 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:: IPCORE_CLK
   - CLKOUT1 ⟶ External Ports
   - CLKOUT2 ⟶ External Ports
   - RST ⟶ net_gnd
   - LOCKED ⟶ External Port

2.7 Configuring the RX Clock Generator IP Core

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

1. **Double click** the Clock Generator PCore and **configure** the settings as follows
   - Input Clock Frequency of **40Mhz**
   - CLKOUT0 Required Frequency of **40MHz**, 180 Phase, **PLLE0** group and **Buffered True**
   - CLKOUT1 Required Frequency of **20MHz**, 180 Phase, **PLLE0** group and **Buffered True**

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

2. **Connect** the pins according to the following.
   - CLKIN ⟶ External Ports
   - CLKOUT0 ⟶ adc_driver::IPCORE_CLK
   - CLKOUT1 ⟶ dc_offset:: IPCORE_CLK
   - RST ⟶ net_gnd
   - LOCKED ⟶ External Port

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

rx_clock_generator			
CLKIN	External Ports::rx_clock_generator_CLKIN_pin	rx_clock_generator_CLKIN	I
CLKOUT0	adc_driver::IPCORE_CLK	rx_clock_generator_CLKOUT0	O
CLKOUT1	dc_offset::IPCORE_CLK	rx_clock_generator_CLKOUT1	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::rx_clock_generator_LOCKED_pin	rx_clock_generator_LOCKED	O
tx_clock_generator			
CLKIN	External Ports::tx_clock_generator_CLKIN_pin	tx_clock_generator_CLKIN	I
CLKOUT0	mcu_driver::IPCORE_CLK	tx_clock_generator_CLKOUT0	O
CLKOUT1	External Ports::tx_clock_generator_tx_clk_pin	tx_clock_generator_CLKOUT1	O
CLKOUT2	External Ports::tx_clock_generator_rx_clk_pin	tx_clock_generator_CLKOUT2	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::tx_clock_generator_LOCKED_pin	tx_clock_generator_LOCKED	O

Figure 2-2: Clock Generator port configurations

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

2.8 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.

```
############################################### PL clocks and reset ##########################################
NET tx_clock_generator_CLKIN_pin          LOC = D18   | IOSTANDARD = LVCMOS25;
NET tx_clock_generator_CLKIN_pin          TNM_NET = tx_clock_generator_CLKIN;
TIMESPEC TS_tx_clock_generator_CLKIN = PERIOD tx_clock_generator_CLKIN 40.000 MHz;
#####################################################################################################################
NET rx_clock_generator_CLKIN_pin          LOC = L18   | IOSTANDARD = LVCMOS25;
NET rx_clock_generator_CLKIN_pin          TNM_NET = rx_clock_generator_CLKIN;
TIMESPEC TS_rx_clock_generator_CLKIN = PERIOD rx_clock_generator_CLKIN 40.000 MHz;
##########################################Chilipepper Rx and Tx clock lines####################################
NET tx_clock_generator_tx_clk_pin         LOC = C17        | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET tx_clock_generator_rx_clk_pin         LOC = J18        | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
#############################################Rx – FMC interface at 2.5V ########################################
NET adc_driver_rx_iq_sel_pin              LOC = N19        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[0]                 LOC =M21         | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[1]                 LOC = J21        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[2]                 LOC = M22        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[3]                 LOC = J22        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[4]                 LOC = T16        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[5]                 LOC = P20        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[6]                 LOC = T17        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[7]                 LOC = N17        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[8]                 LOC = J20        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[9]                 LOC = P21        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[10]                LOC = N18        | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[11]                LOC = J16        | IOSTANDARD = LVCMOS25;
############################################ 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 tx_clock_generator_LOCKED_pin         LOC = T22        | IOSTANDARD =LVCMOS33;  # "LD0"
NET rx_clock_generator_LOCKED_pin         LOC = T21        | IOSTANDARD =LVCMOS33;  # "LD1"
NET adc_driver_blinky_pin                 LOC = U22        | IOSTANDARD =LVCMOS33;  # "LD2"
NET mcu_driver_blinky_pin                 LOC = U21        | IOSTANDARD =LVCMOS33;  # "LD3"
NET dc_offset_blinky_pin                  LOC = V22        | IOSTANDARD =LVCMOS33;  # "LD4"
```

Figure 2-3: EDK project pin assignments

2.9 Adding ChipScope Peripheral

The last step is to setup the ChipScope peripheral to verify the functionality of the dc_offset_correction.

1. Select Debug -> **Debug Configuration** from the top menu
2. Click the **Add ChipScope Peripheral** button on the bottom left hand side of the screen
3. Select To **monitor arbitrary system level signals** (middle option) from the list.
4. Add the i_out and q_out pins from the dc_offset Port. Additionally, you should set the clock to the same clock used for the core, which for this design is rx_clock_generator_clockout_1.
5. (optional) you can also add the rx_i and rx_q signals from the ADC Driver to see the before and after affect of the dc correction.
6. Click ok to finish configuration of your ChipScope peripheral. Your new port list should look similar to Figure 2-4 below. Be sure your Clock and dc_offset ports have the ChipScope peripherals in the correct locations.

dc_offset			
IPCORE_CLK	rx_clock_generator::CLKOUT1	rx_clock_generator_CLKOUT1	I
IPCORE_RESETN	processing_system7_0::FCLK_RESET0_N	processing_system7_0_FCLK...	I
i_in	adc_driver::rx_i	dc_offset_i_in	I
q_in	adc_driver::rx_q	dc_offset_q_in	I
i_out	chipscope_ila_0::TRIG0	dc_offset_i_out_to_chipscop...	O
q_out	chipscope_ila_0::TRIG0	dc_offset_q_out_to_chipsco...	O
blinky	External Ports::dc_offset_blinky_pin	dc_offset_blinky	O
(BUS_IF) S_AXI	Connected to BUS axi_interconnect_1	Connected to BUS axi_interc...	
mcu_driver			
chipscope_icon_0			
chipscope_ila_0			
rx_clock_generator			
CLKIN	External Ports::rx_clock_generator_CLKIN_pin	rx_clock_generator_CLKIN	I
CLKOUT0	adc_driver::IPCORE_CLK	rx_clock_generator_CLKOUT0	O
CLKOUT1	dc_offset::IPCORE_CLK chipscope_ila_0::CLK	rx_clock_generator_CLKOUT1	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::rx_clock_generator_LOCKED_pin	rx_clock_generator_LOCKED	O
tx_clock_generator			
CLKIN	External Ports::tx_clock_generator_CLKIN_pin	tx_clock_generator_CLKIN	I
CLKOUT0	mcu_driver::IPCORE_CLK	tx_clock_generator_CLKOUT0	O
CLKOUT1	External Ports::tx_clock_generator_tx_clk_pin	tx_clock_generator_CLKOUT1	O
CLKOUT2	External Ports::tx_clock_generator_rx_clk_pin	tx_clock_generator_CLKOUT2	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::tx_clock_generator_LOCKED_pin	tx_clock_generator_LOCKED	O

Figure 2-4: Ports list after adding ChipScope peripheral to monitor ADC signals

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 dc offset correction project.

1. Select **File → New → Application Project**.
2. Name the project "qpsk_rx" 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_rx project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the qpsk_rx 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 Figure 3-1.
7. **Download** the **Chilipepper.c** and **Chilipepper.h** files from the GitHub repository[2] if you don't already have them. Copy them into the source directory with your main.c file.
8. Open the Chilipepper.c file and modify it for this lab. The only PCores that should be defined at the top of the file are MCU_DRIVER, DC_OFFSET, and MCU_UART.

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

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

XUartPs uartPs;
XUartPs_Config *pUartPsConfig;

int main()
{

	init_platform();

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

	Chilipepper_SetPA(0);
	Chilipepper_SetTxRxSw(1);   // 0- transmit, 1-receive
	Chilipepper_SetDCOC(1);     // enable dc offset correction

	while (1)
	{
		Chilipepper_ControlAgc(); //update the Chilipepper AGC
	}
	cleanup_platform();
	return 0;
}
```

Figure 3-1: main.c file for DC Offset Correction SDK Project

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.

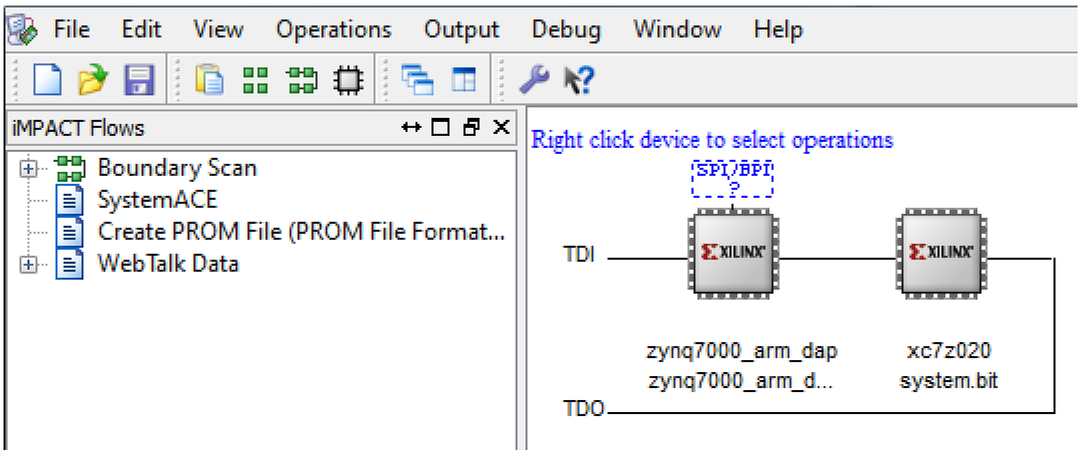
5. Select yes to allow iMPACT to automatically create a new project for you. If you receive any connection errors, verify your USB or JTAG programmer cables are connected properly.

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

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

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

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



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

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

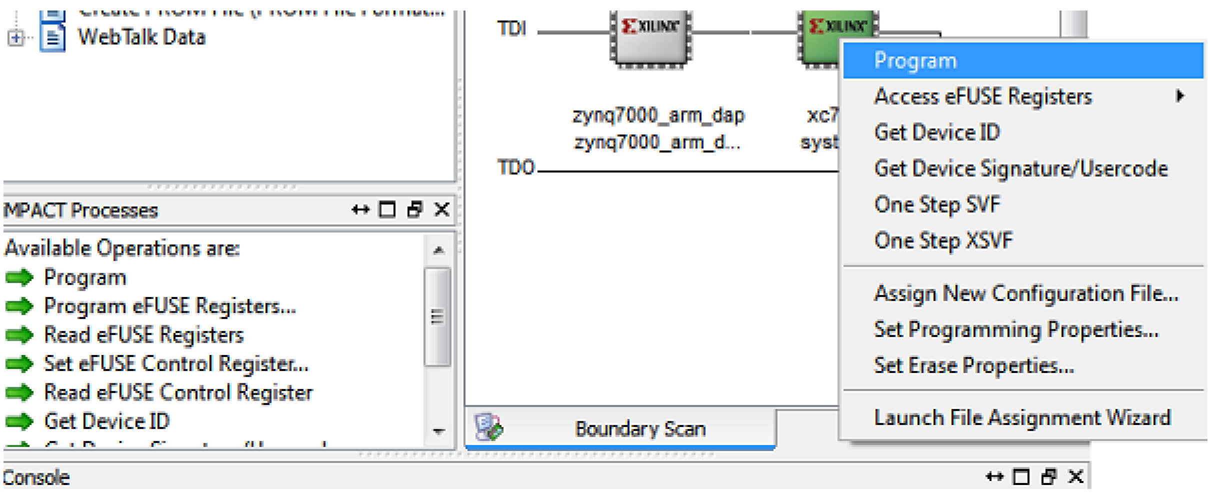


Figure 3-3: iMPACT configuration screen

3.3 Debugging with SDK

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

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

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

If the software initialization worked, you should see a green light on the Chilipepper, as well as the Blinking LEDs on the FPGA from the dc_offset, MCU and ADC PCores.

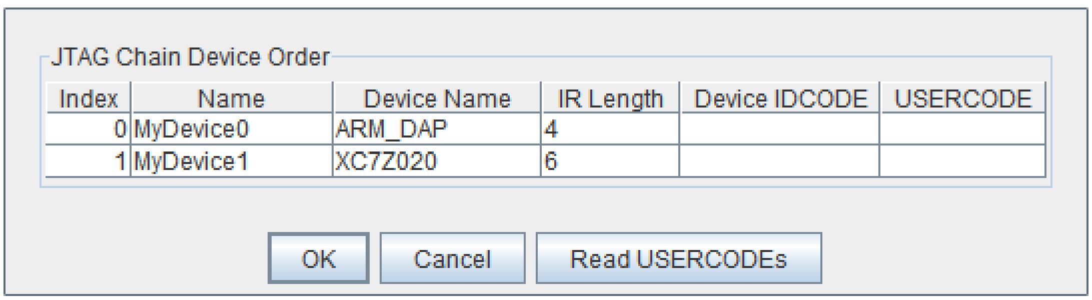
Testing and Design Verification Step 4

4.1 Verification with ChipScope Pro

There are several methods available for verifying the MATLAB functions. For verification of the DC Offset, ChipScope is recommended as it provides the most useful view of the dc output, especially when compared to the output of the ADC.

1. To verify the dc offset signals, you will need to open **ChipScope Pro Analyzer**. Be sure that the JTAG cable is connected to the FPGA board properly.

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.



Note	If you receive an error from ChipScope stating that you either cannot detect or cannot open the cable, try using the optional Step 3 to configure your cable setup correctly.

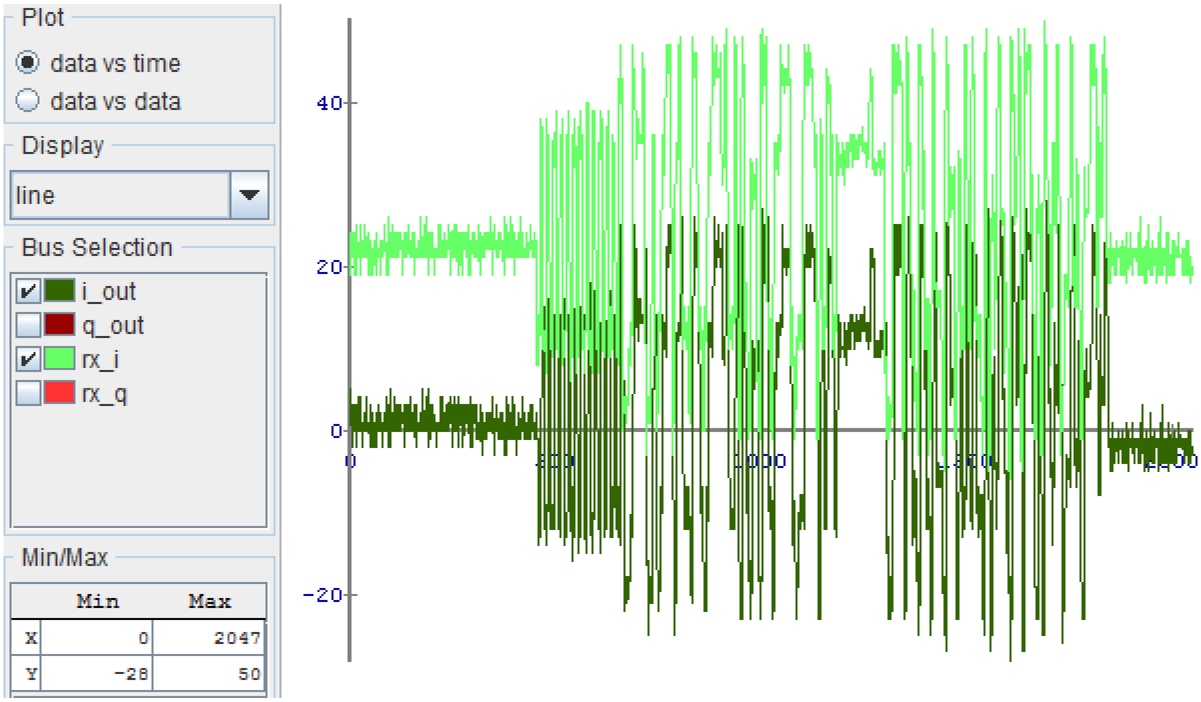
3. (**Optional**)Click JTAG Chain in the top menu selection. Select the option for **Open Plug-in**... You will be greeted with a Plug-in Parameters screen. Enter the following in the box, and hit ok. "**xilinx_tcf URL=tcp::3121**". Then click the open cable button and proceed as usual.

4. Select ok to get to the Analyzer main screen. Open the **file menu** and select **Import**.

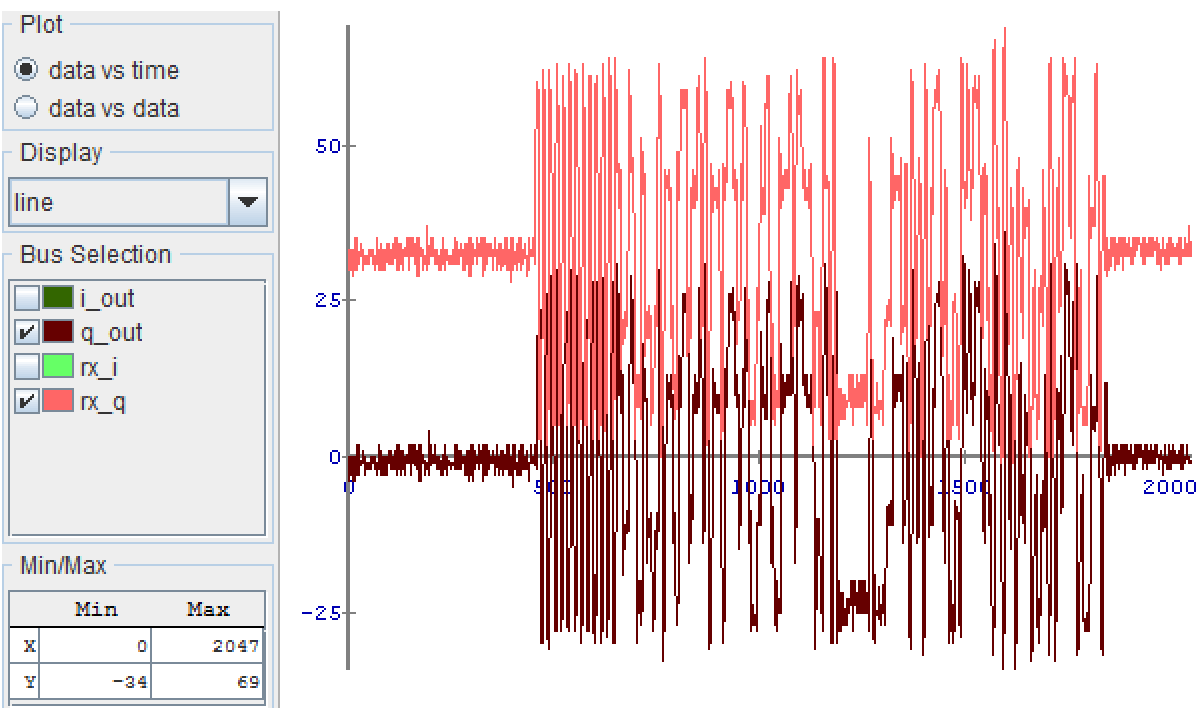
5. Click **Select New File**, and browse to the location of your ChipScope **CDC file**, which is located in the <EDK/implementation/chipscope_ila_0_wrapper> folder of your project directory. This file was created for you when you generated your bit file in EDK, assuming you added the ChipScope peripheral appropriately. It tells the ChipScope program how to interpret the data it is receiving from the JTAG port.

6. On the Bus Plot screen, you can view the I and Q channel signals 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, both signals should be set to the signed decimal bus radix.

7. 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 Figures 4-1 and 4-2.



4-1: ADC I channel output vs. dc_offset I channel output



4-2: ADC q channel output vs. dc_offset q channel output

From the above plots you can clearly see the core is shifting the waveform back towards dc as expected. These signals were created using Lab 3 to create the QPSK signal, however even without the use of the second Chilipepper board, you can clearly see the noise sent out of the ADC is also corrected back to 0 DC.

Appendix A MATLAB DC Offset Correction Function

MATLAB function `dc_offset_corerction.m`

```
function [i_out, q_out, rssi_out, rssi_en_out, dir_out, dir_en_out,...
    blinky] = dc_offset_correction(i_in, q_in, gain_en_in,...
        rssi_low_goal_in, rssi_high_goal_in, rx_en_in)

persistent i_dc q_dc i_mean q_mean
persistent counter rssi_sum
persistent dir_state
persistent rssiHold
persistent noise_offset noise_inc noise_dec
persistent blinky_cnt

%alpha = alpha_in/2^12;
alpha = 1/2^12;

if isempty(i_dc)
    i_dc = 0;
    q_dc = 0;
    i_mean = 0;
    q_mean = 0;
    counter = 0;
    noise_inc = 0;
    noise_dec = 0;
    noise_offset = 0;
    rssi_sum = 0;
    dir_state = 0;
    rssiHold = 0;
    blinky_cnt = 0;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DC Correction section
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if rx_en_in == 1
    i_mean = (1-alpha)*i_mean + alpha*i_in;
    q_mean = (1-alpha)*q_mean + alpha*q_in;

    i_dc = (1-alpha)*i_dc + alpha*i_in; %update the i dc offset
    q_dc = (1-alpha)*q_dc + alpha*q_in; %update the q dc offset

    if abs(i_mean) > (50 + noise_offset)  % too much noise, raise cieling.
        noise_inc = noise_inc + 1;
        i_dc = 0;
    else
        noise_dec = noise_dec + 1;
    end
```

```
    if abs(q_mean) > (50 + noise_offset)  % too much noise, raise cieling.
        noise_inc = noise_inc + 1;
        q_dc = 0;
    else
        noise_dec = noise_dec +1;
    end
    if (noise_inc > 10)
        %there is a high dc_offset value that needs to be corrected
        noise_offset = noise_offset + 10;
        noise_inc = 0;
    end
    if (noise_dec > 100000)
        %dc offset threshold is higher than needed
        noise_offset = noise_offset - 10;
        noise_dec = 0;
    end
end
i_out = i_in - i_dc;
q_out = q_in - q_dc;
%correct false positive/nagatives
if (abs(i_mean) < 50)
    noise_inc = 0;
end
if (abs(i_mean) > noise_offset - 10)
    noise_dec = 0;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RSSI Estimation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rssi_inst = i_out*i_out + q_out*q_out;
rssi_en_out = 0;
rssi_out = 0;

if rx_en_in == 1
    if counter == 0 && rssi_inst > 2*50*50
        counter = 1;
        rssi_sum = 0;
    end
    if counter ~= 0
        if rssi_inst < 2*50*50
            counter = 0;
        else
            counter = counter + 1;
            rssi_sum = rssi_sum + rssi_inst;

            if counter >= 2^8
                counter = 0;
                rssi_out = round(rssi_sum/2^8);
                rssiHold = rssi_out;
                rssi_en_out = 1;
            end
        end
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gain Correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dir_out = 0;
dir_en_out = 0;

% dir_out = 0 - do nothing 1 - increase 2 - decrease
ai = abs(i_in);
aq = abs(q_in);
% only increase power if the rssi is away from the mean
rssi_diff = abs(rssiHold-(i_mean*i_mean+q_mean*q_mean));
if rx_en_in == 1
    switch dir_state
        case 0 % wait for some action and the processor is done
            if gain_en_in == 0
                if rssi_en_out == 1
                    if rssi_diff < rssi_low_goal_in %too low - increase
                        dir_out = 1;
                        dir_en_out = 1;
                        dir_state = 1;
                    end
                    if rssi_diff > rssi_high_goal_in %too high - decrease
                        dir_out = 2;
                        dir_en_out = 1;
                        dir_state = 1;
                    end
                end
                % we're saturating the ADC so decrease gain
                % this overrides anything else
                if (ai > 1500) || (aq > 1500)
                    dir_out = 2; % decrease
                    dir_en_out = 1;
                    dir_state = 1;
                end
            end
        case 1 % see if the MCU has done something and if so reset
            if gain_en_in == 1
                dir_out = 0;
                dir_en_out = 1;
                dir_state = 0;
            end
        otherwise
            dir_state = 0;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    blinky_cnt = blinky_cnt + 1;
    if blinky_cnt == 20000000
        blinky_cnt = 0;
    end
    blinky = floor(blinky_cnt/10000000);
end
```

Appendix B MATLAB DC Offset Correction Test Bench Script

MATLAB script `dc_offset_correction_tb.m`

```
clear all

fid = fopen('dc.prn');
M = textscan(fid,'%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d
%d','Headerlines',1);
%M = textscan(fid,'%d %d %d %d','Headerlines',1);
fclose(fid);
is = double(M{3});
qs = double(M{4});

l = zeros(1,length(is));

figure(1)
clf
subplot(2,1,1)
plot(is)
hold on
plot(l,'g');
title('Original: Inphase');
subplot(2,1,2)
plot(qs)
hold on
plot(l,'g');
title('Original: Quadrature');

disp(['Original Mean I: ',num2str(mean(is)),' Mean Q: ',num2str(mean(qs))]);


% L is the recovery time of the filter, i.e., it takes about that many
% samples for it to recover.
% Becuase an averaging COMB filter is essentially a high pass filter it is
% best to have an offset in frequency above DC in order to be able to keep
% the filter length as short as possible
Ns = length(is);
io = zeros(1,Ns);
qo = zeros(1,Ns);
rssi = zeros(1,Ns);
ff_in = 2^3/2^12*2^12;
rssiH = 0;
for i1 = 1:length(is)
    i_in = is(i1);
    q_in = qs(i1);

    [i_out, q_out, rssi_out, rssi_en_out, dir_out, dir_en_out] = ...
        dc_offset_correction(i_in, q_in, mod(i1,2), ...
            500, 1500, +(i1>3000));
```

```
    io(i1) = i_out;
    qo(i1) = q_out;
    if rssi_en_out
        rssiH = rssi_out;
    end
    rssi(i1) = rssiH;
end

figure(2)
clf
subplot(2,1,1)
plot(io)
hold on
plot(l,'g');
title('Corrected: Inphase');
subplot(2,1,2)
plot(qo)
hold on
plot(l,'g');
title('Corrected: Quadrature');

disp(['Corrected Mean I: ',num2str(mean(io)),' Mean Q: ',num2str(mean(qo))]);

figure(3)
clf
s = complex(io,qo);
plot(s.*conj(s),'r')
hold on
plot(rssi,'b');
title('rssi');
```