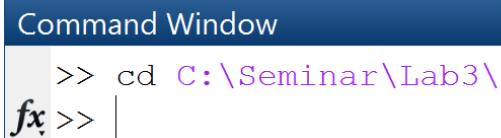


Deployed Design Debugging

In this lab we will examine different ways of debugging our deployed HDL design through MATLAB and Simulink. The methods covered will be: standard IQ interface manipulation, External Mode, and AXI-Memory Mapped interfaces. These techniques are very useful for accessing portions of a design once deployed to hardware for purposes of data inspection, algorithmic tuning, verification,...etc.

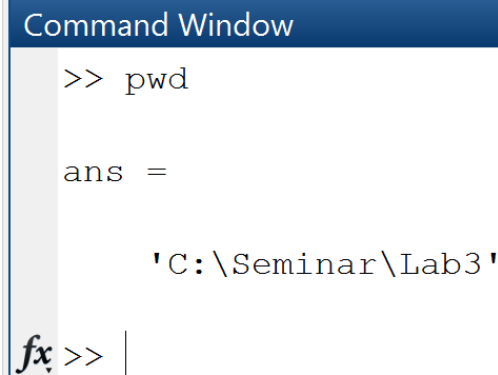
Before we dig into these methods, let us first discuss the models themselves and how they are structured. This discussion will provide motivation for the debugging techniques and show how models can be designed to make them debug-able.

For this lab make sure you are in the lab3 folder which should be in the C:\Seminar\Lab3 folder:



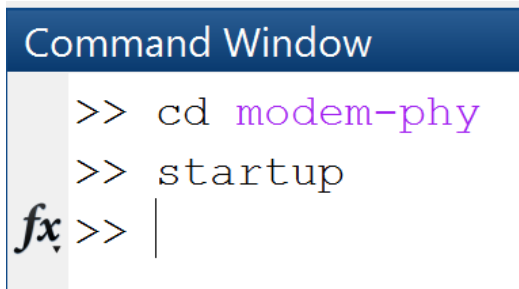
```
Command Window
>> cd C:\Seminar\Lab3\
fx>> |
```

To make sure MATLAB is setup correctly, type “pwd” into the command prompt and hit enter. This should output the following



```
Command Window
>> pwd
ans =
    'C:\Seminar\Lab3'
fx>> |
```

Now go into the “modem-phy” folder and run the startup.m script to initialize the environment:

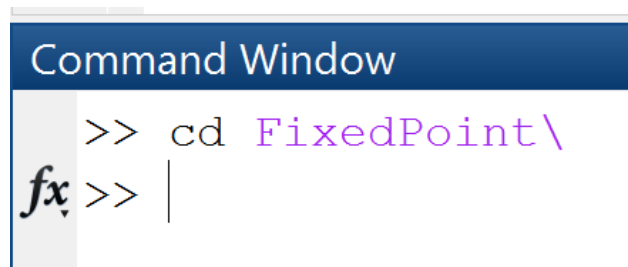
A screenshot of the MATLAB Command Window. The title bar is dark blue with the text "Command Window" in white. The window has a light gray background. On the left side, there is a vertical gray bar with the text "fx" in a stylized font. The command prompt shows two lines of code: ">> cd modem-phy" and ">> startup". The cursor is on the third line, which starts with ">> " followed by a vertical bar "|".

```
Command Window
>> cd modem-phy
>> startup
fx >> |
```

This folder contains the entire modem physical layer design including:

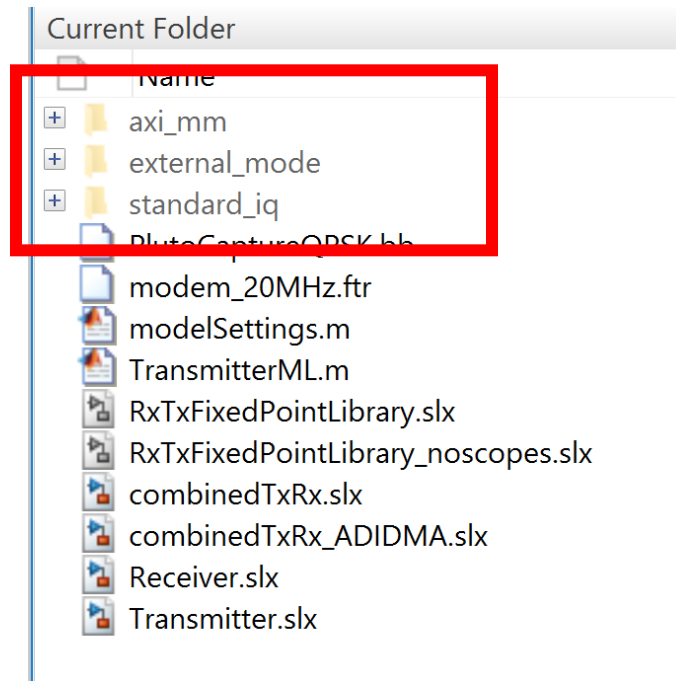
- MATLAB Reference Design
- Simulink Model
- Test infrastructure
- Scripts to control the build process

For the purpose of this lab we will be working in the “FixedPoint” folder, which contains all of the fixed-point designs. Navigate into this folder by typing “cd FixedPoint”.

A screenshot of the MATLAB Command Window. The title bar is dark blue with the text "Command Window" in white. The window has a light gray background. On the left side, there is a vertical gray bar with the text "fx" in a stylized font. The command prompt shows two lines of code: ">> cd FixedPoint\" and ">> |". The cursor is on the third line, which starts with ">> " followed by a vertical bar "|".

```
Command Window
>> cd FixedPoint\
fx >> |
```

The image bellows shows the files available in this folder. The three subfolders contain the different models for the different deployed debugging strategies we will implement.



Let us discuss how these different models are built. We will start by entering the `standard_iq` folder:

```
Command Window
>> cd standard_iq
fx >> |
```

Next open the main model here:

```
Command Window
fx >> open combinedTxRx_StandardIQ.slx|
```

Within this model we first see a large orange subsystem called “Combined TX and RX”, which holds the Transmitter and Receiver IP subsystems. The remaining blocks are for control, visualization, and management of data sources. Now, let us look inside the “Combined TX and RX” subsystem by double-clicking on the orange block itself.

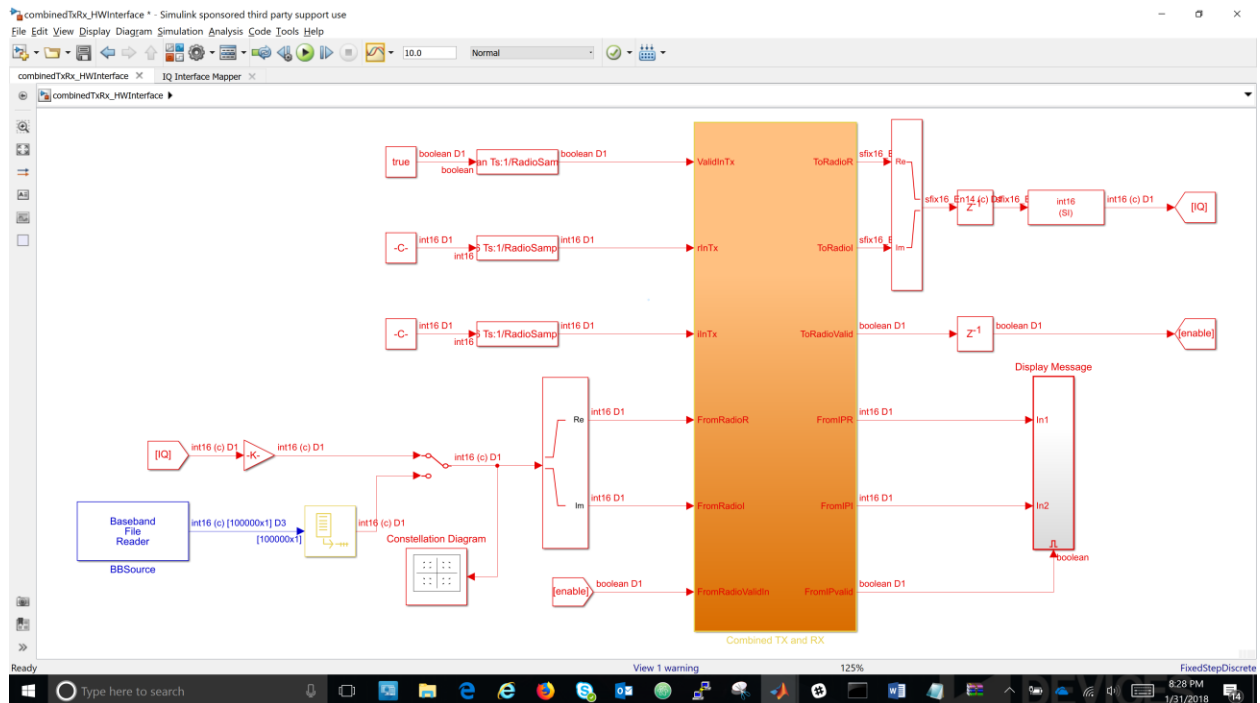


Figure 1

Inside this block, we can see two large green blocks, “HDL Receiver” and “HDL Transmitter”, as shown in Figure 2. These are our main processing blocks and contain a number of inputs and outputs. These are used for both configuration and data streaming into and out of these subsystems. While we won’t be examining these blocks in too much detail, it is important to illustrate how these subsystems are connected to the parent subsystem. In this model you will notice that only IQ and valid signals are mapped to the parent subsystem. This will change depending on the tooling and debugging method used. What we call the “Standard IQ” debugging method is by far the simplest and only uses the IQ data ports to pass data into and out of the subsystems.

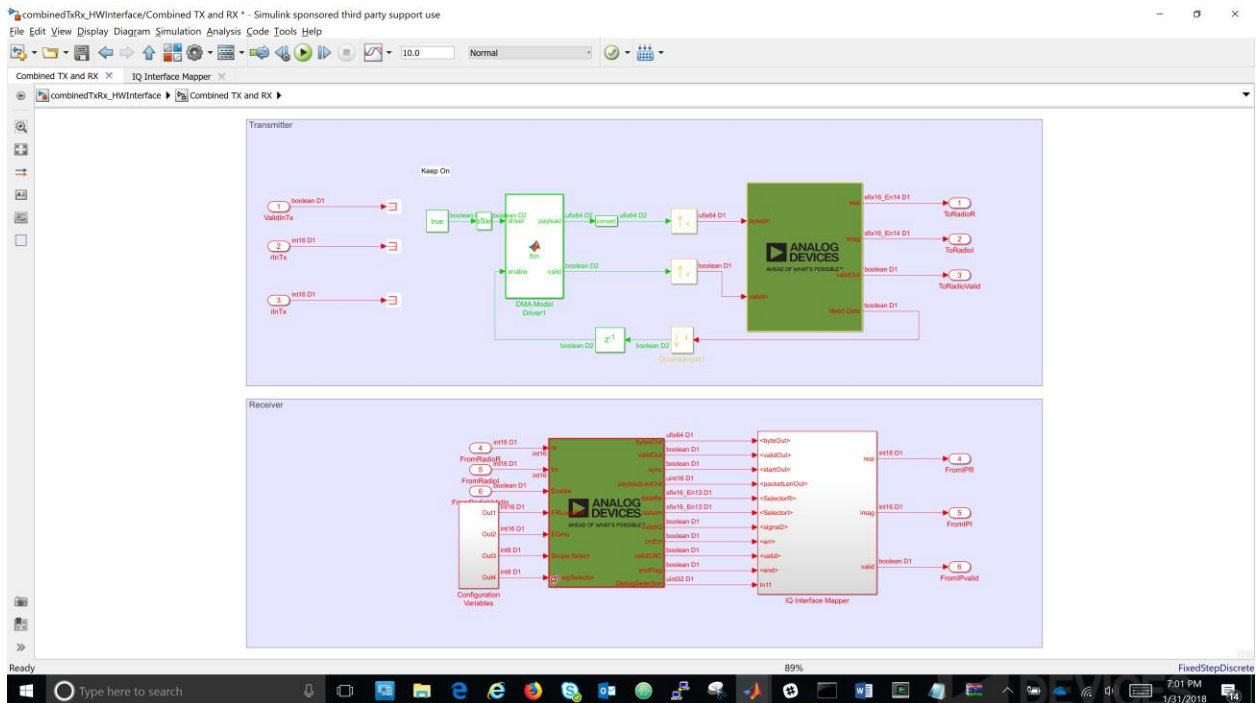
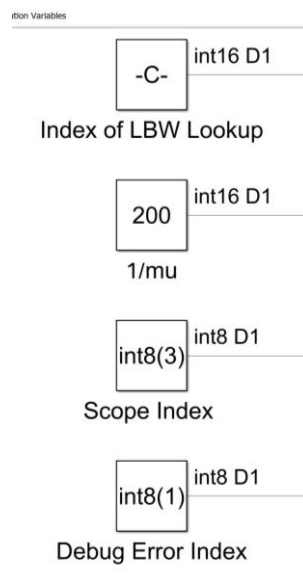


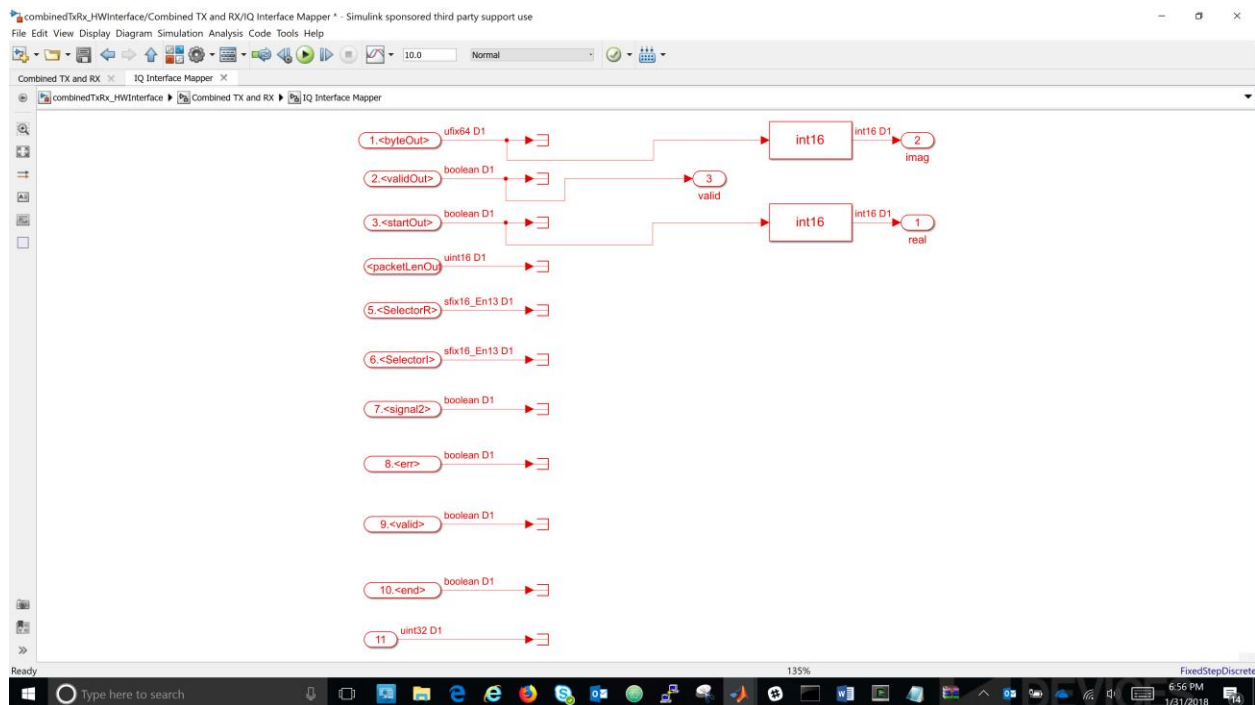
Figure 2


Due to limitations from the ADI reference design and MathWorks standard workflow, the standard HDL reference design requires the IQ ports (including valid) to be mapped. Even if they are not actually used. This is due to how the DMA engines and connected cores are configured. However, in custom reference designs you can remove this limitation, which is a common design path. The standard designs are considered just starting points for users.

Next, if we consider the Receiver section we can observe a subsystem called “Configuration Variable”. Looking inside this subsystem by double-clicking on its mask we notice a set of “Constant” blocks. These are used as a static configuration for the receiver.



Connected to the output of the “HDL Receiver” subsystem is another subsystem called “IQ Interface Mapper”, which is where we do a static selection of what will be connected out of the entire design. Double-click on this block and you will observe the “bytesOut” port is connected to the imaginary port and “startOut” is connected to the real port. The “validOut” port is related to the “byteOut” port pulsing high when we have valid data. We connect this to the “valid” output port. We are essentially reusing the IQ data lines to transmit binary data out with some control signals.



If we go back up to the “Combined TX and RX” subsystem by pressing the “Up to parent” button , we will change focus to the transmitter side. In the highlighted box in Figure 3 is a controller that was inserted into the design to continuously adding packets with a cycling message. You will also notice that we ignore all input signal. We do this to ignore any possibility of underflow control to the Transmitter, which simplifies debugging.

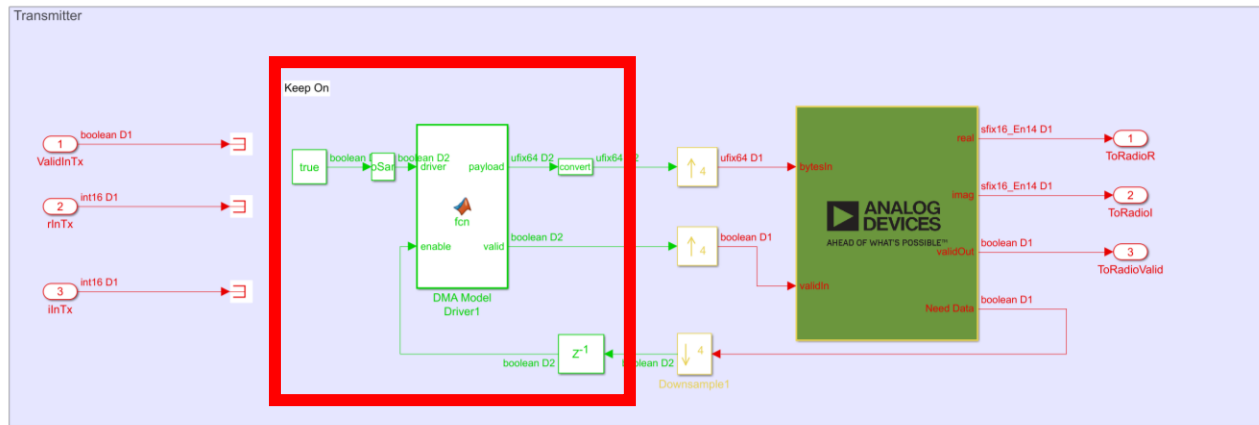


Figure 3

Open the transmitter controller by double-clicking on the “DMA Model Driver” block, which will open a MATLAB script. On line 9 of this script we define the default message to be transmitted. Each character is mapped to a 64-bit word and the remaining frame is padded with alternating 1’s and 0’s. The transmitter will increment the last character for each frame cycling from 0 – 9 and then repeating the pattern.

```

1 function [payload,valid] = fcn(driver, enable)
2 %#codegen
3
4 persistent indx mode pSize counter msg payloadIndx
5
6 mSize = fi(length('Hello World 0')*8,0,64,0); % each char will be mapped to a 64 bit word
7
8 if isempty(indx)
9     msg = fi(int8('Hello World 0'),0,64,0);
10    pSize = fi(8*200,0,64,0); % Bytes in message (must be multiple of 64 bits)
11    mode = int8(0);
12    counter = fi(0,0,64,0);
13    payloadIndx = uint8(0);

```

The last piece we need to look at in the model is the output decoder. This is connected to the main “Combined TX and RX” subsystem and is responsible for printing the received messages to the console. The MATLAB function block is located within the block highlighted in Figure 4. This block simply casts the output words into characters for display, given the appropriate control signaling embedded in the IQ data signals.

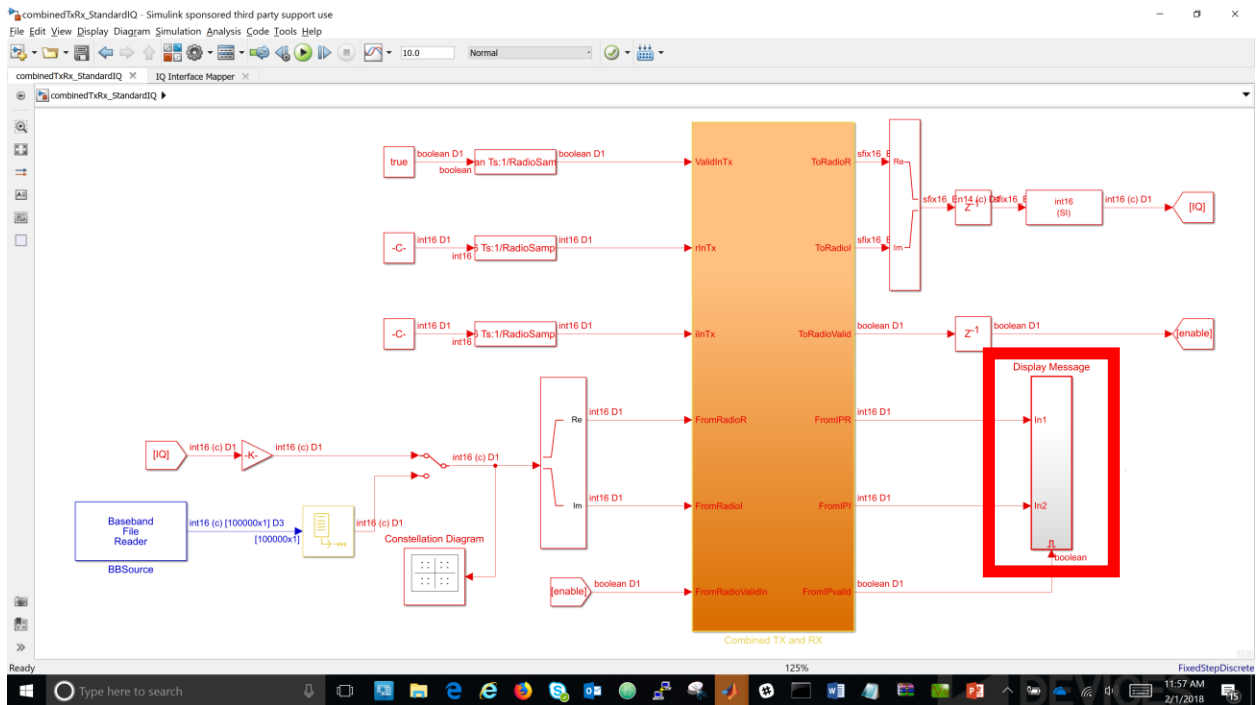


Figure 4

Before we move the design to the board, let us look at the expected outputs. Press the run button



and you should see a number of scopes appear which show the activity of the signal chain. We can examine the output of the “Display Message” block in the Diagnostic Viewer. To open the viewer, go to View->Diagnostic Viewer as in Figure 5.

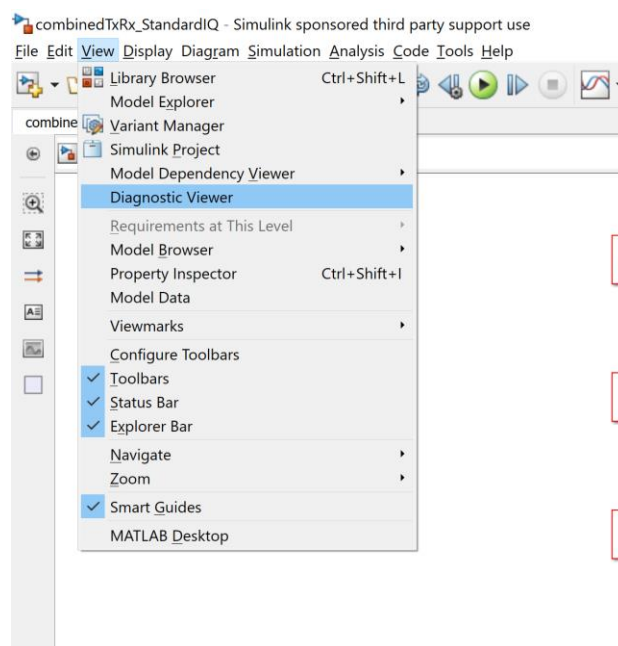


Figure 5

This will pop-up an additional window which will look similar to Figure 6, where our decoded messages are printed.

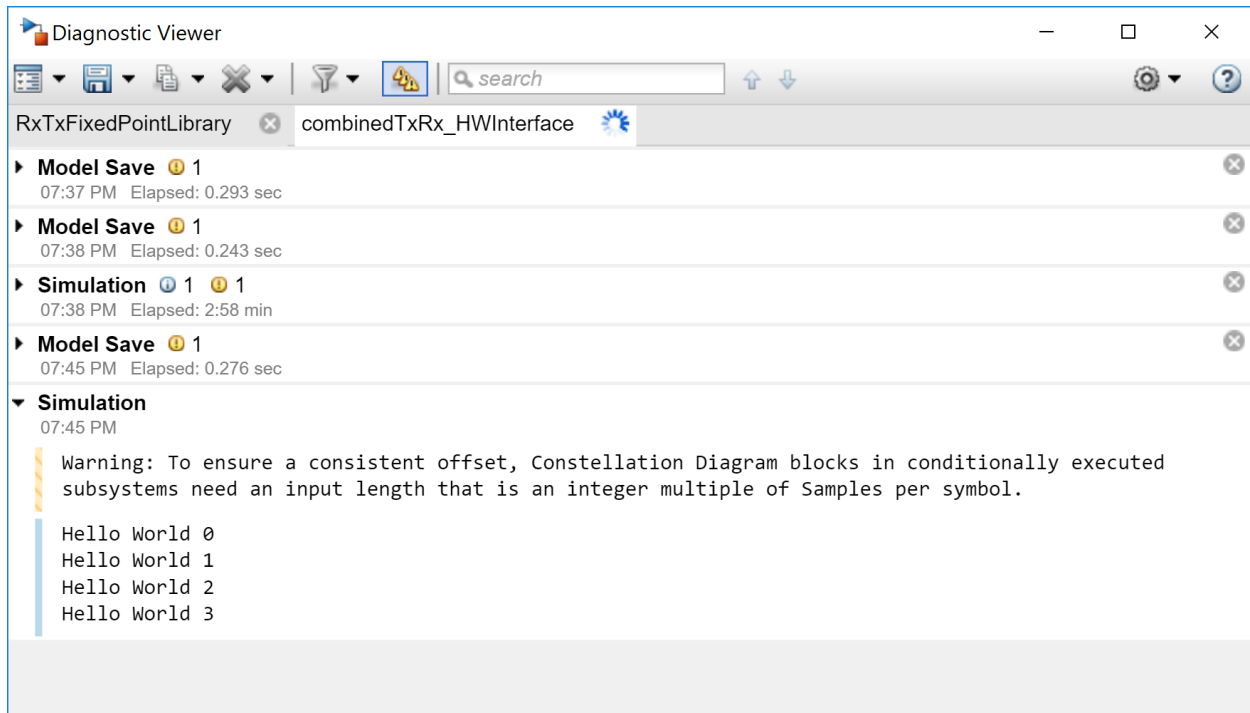
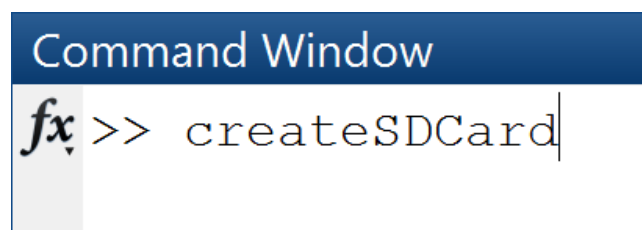


Figure 6

Creating an SD Card For Development Board

Each development board must have a bootable SD card which will both flash our bitfile onto the FPGA and load our operating system onto the ARM for access and control from MATLAB and Simulink. MathWorks provides a standard image derived from ADI reference designs for this purpose. We will create the base SD card here and load additional images as we progress through our deployed designs. ***First take the SD card from the RFSOM board and insert it into your local machine. It should appear as drive D: (If not please let the instructor know). Next run the following command:***



Once complete eject SD card and insert into RFSOM FMC carrier board.

Running the Design in Hardware

The current model has been validated to support HDL code generation. For the sake of time, we have already synthesized the design. If you want to look at the generated Vivado project, look in the folder "hdl_prj" and you can examine the design files. At this point we can deploy the design to hardware. To

do this we, need to load the bitstream onto the device, which can be done from MATLAB. . To load the bitstream, make sure:

1. The Ethernet cable is plugged into the RFSOM board and your PC
2. The USB cable is connected to the USB-UART connection on the RFSOM board and connected to your PC
3. The first SD card is inserted into the board
4. The board is powered up

Now check that MATLAB can talk to the board with the following commands:

```
Command Window
>> dev = sdrdev('ADI RF SOM');
>> dev.info
## Establishing connection to hardware. This process can take several seconds.

ans =

    struct with fields:

        Status: 'Full information'
        ProtocolVersion: '8.0.0'
        FirmwareVersion: '8.0.0 for Zynq, build Oct 26 2017 19:25:12'
        HardwareVersion: '8.0.0 for Zynq, build Jan 31 2018 00:12:06'
        HardwareRxCapabilities: 'Device does not have targeted Rx DUT'
        HardwareTxCapabilities: 'Device has targeted Tx: DUT ChannelMapping = 1'
        RFBoardVersion: 'RF Chip: AD9361, PCORE: version 9.0.98'
        RFBoardRxCapabilities: 'BasebandSampleRate: [520.9kHz,61.44MHz]; CenterFrequency: [70MHz,6GHz]; NumChannels=2'
        RFBoardTxCapabilities: 'BasebandSampleRate: [520.9kHz,61.44MHz]; CenterFrequency: [70MHz,6GHz]; NumChannels=2'

fx>> |
```

Figure 7

You should receive an output similar to that of Figure 7. Next, we can load the bitstream onto the board. For convenience, there is a script in this directory which will do this:

```
Command Window


>> sendToBoard
## Loading bitstream file
## Rebooting board
## Reboot complete
fx>> |
```

Once the board has rebooted, the FPGA will contain our design, which was represented by the large orange subsystem in Figure 1. To control the newly deployed logic we have two additional pieces, a Simulink model to control the transmitter and a MATLAB script to parse the transmitted message.

The model “interface_model_tx.slx” is located in the same folder and can be opened with the command:

```
Command Window

fx>> open interface_model_tx
```

This model, shown in Figure 8, is responsible for configuring the transmitter and enabling the transmit path. Start this model with the play button .

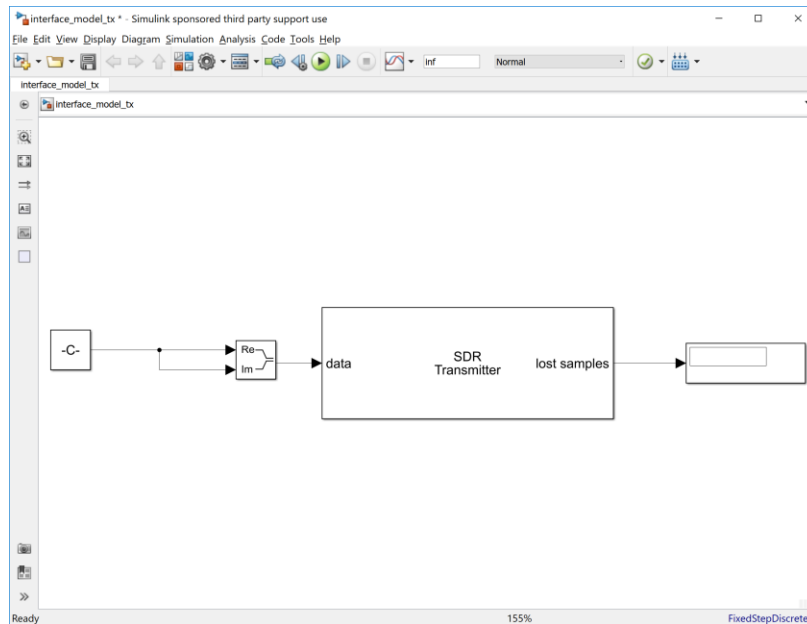


Figure 8

While the transmitter model is running we can check the receiver. This is implemented in MATLAB, using the same System objects we used during testing.

Open the script called “interface_rx.m”:

```
Command Window
>> edit interface_rx.m
fx >> |
```

This script calls the receiver System Object for the RFSOM and parses the IQ data, using the same function as the Simulink model, which was transplanted into this script.

Run this script to see the decoded output, as shown below:

```
Command Window
>> interface_rx
Warning: The BypassUserLogic property is not relevant in this configuration of the System object.
> In interface_rx (line 3)
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Hello World 0
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
```

The recovered messages will start at a random number (0-9) since we are recovering data at a random point in time, but each packet should be in order.

This example demonstrated a loop-back system, with transmit data created by generated IP, sent out through the transceiver, looped-back into the receiver, and decoded by generated IP.

|