

## AXI-MM

While External Mode is useful for inspecting data values and tuning system parameters, if higher-bandwidth data transfers, or programmatic control from MATLAB is required, then a different method needs to be used. This method is called AXI-Memory Mapped (AXI-MM) access. This is built upon the IIO kernel framework. This method allows direct access to the same registers as before, but enables a higher-performance operation.

Close all existing models from previous parts of the lab and navigate to the “aximm” folder with another updated model for this interface.

```
Command Window
>> cd C:\Seminar\Lab3\modem-phy\FixedPoint\axi_mm\
fx>> |
```

Configure the libraries by running the following commands:

```
Command Window
>> libiioPSP.enableIIO
>> libiioSDRPSP.enableIIORadio
fx>> |
```

These will enable the components necessary to use AXI-MM in our reference design. Open the model “combinedTxRx\_AXIMM.slx”:

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

Figure 15 highlights the differences between this model and the previous one. In this model, we will use the fourth input from the “Configuration Variables”. On the output side we expose two more ports. The new outputs are “selectedError”, which is chosen based on the value of the new fourth input, and “packetLen”, which is connected to the Header Decoder in the Receiver IP. The Header Decoder is responsible for determining the length of the received packet. When a new packet is received, this value will update to show the length of the last packet received. Since the transmitter is unchanged from previous versions, all recovered packets will be of length 200. Note, by default, before any data is recovered, the value 8188 will be returned. Due to the structure of the transmitted data, it can never be of length 8188.

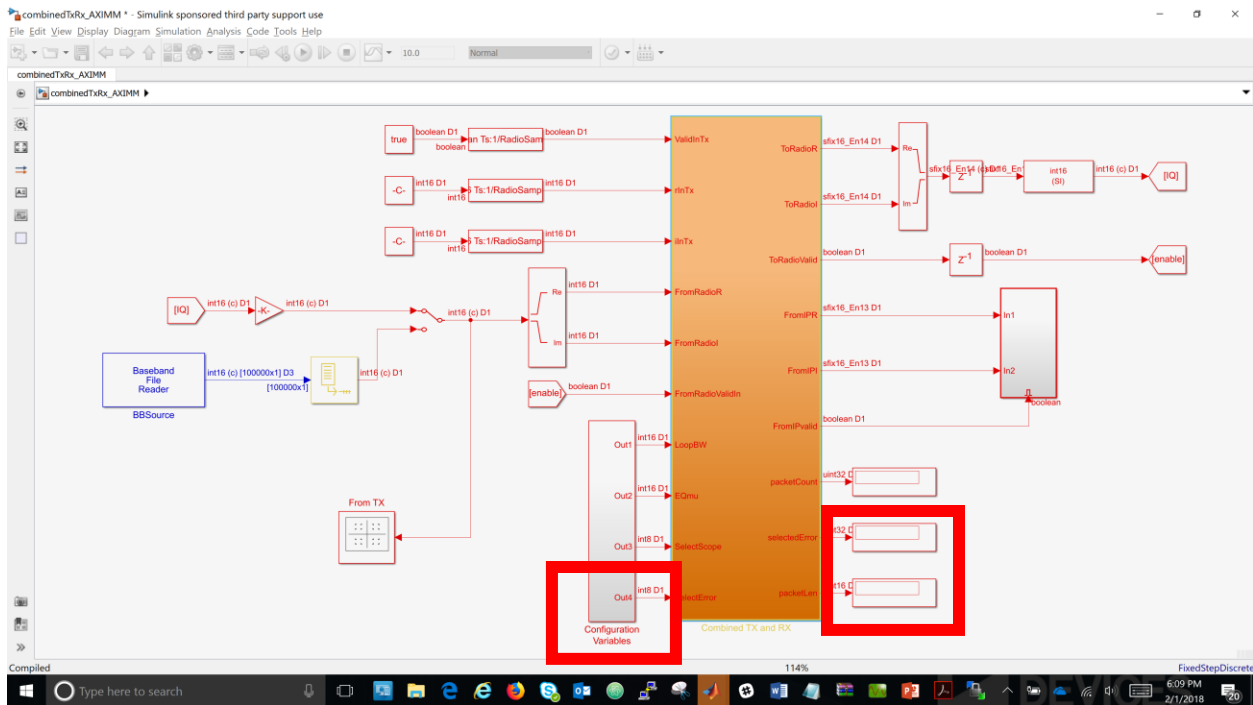
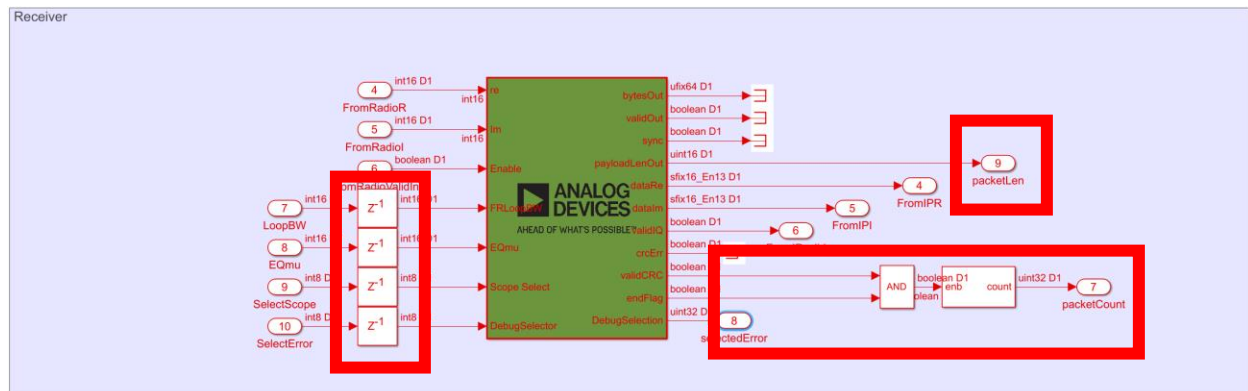


Figure 15

Inside the “Combined TX and RX” subsystem is the “Configuration Variables” are connected to the main Receiver subsystem, along with receiver output ports connected to the receiver subsystem. Delays were added to provide default values into the system at start time.



## Updating the SD Card

To take advantage of the AXI-MM features, we need to update the SD Card in our RF SOM hardware. First power down the board and remove the SD Card. Insert the SD Card into your PC and run the script “updateSDCard.m”:

## Command Window

```
>> updateSDCard  
## Copying to SD card.  
fx >> |
```

Remove the card from your PC, insert back into the board, and power up the board.

## Send New Design to Board

Load the new bitstream onto the board:

## Command Window

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

## Running the Deployed Design

Now that the design is deployed we can explore some details about this synthesized version. As in the last design, the extra ports also have been mapped to AXI-lite. For this design, we have four input control registers and three read registers. However, unlike in the last model where we used External Mode to connect to these registers, the IQ streaming interfaces we will use both MATLAB and Simulink. Specifically, we will be using Simulink to visualize the IQ constellations and use MATLAB to read and write from the AXI-lite registers we have exposed.

Open the Simulink model “interface\_model.slx”:

## Command Window

```
fx >> open interface_model.slx|
```

This model, shown in Figure 16, contains our interface blocks to the transceiver. The Receiver block is connected to a constellation block, whose source will be control by our AXI-lite registers as in the previous exercise.

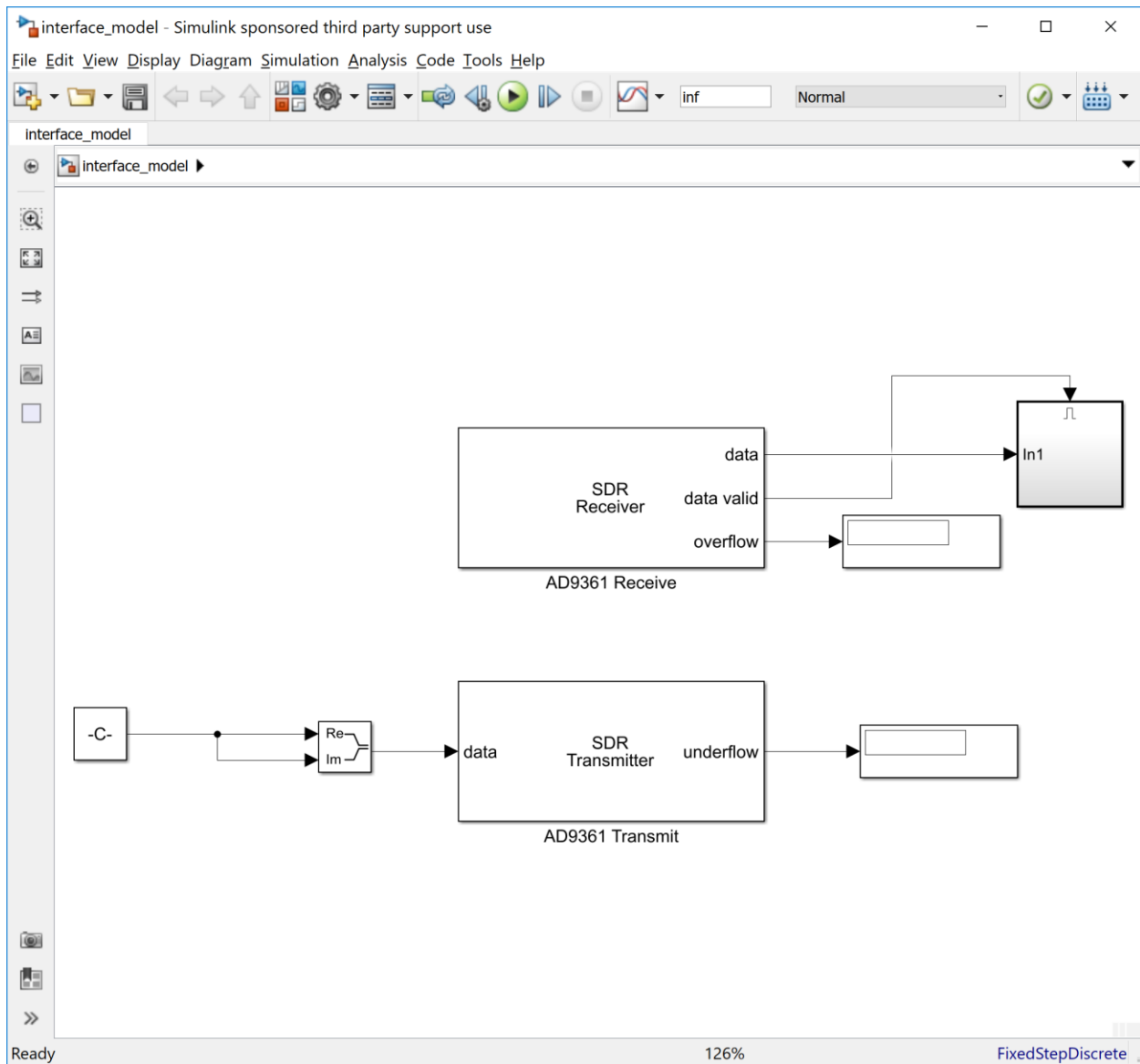

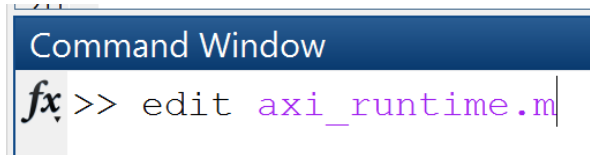


Figure 16

We can start this model up by hitting the Run button . Once running, a constellation diagram should appear. The default constellation selection will be index 1, which is right after SRRC. For now, we can ignore this and leave things running in the background.

Next, we will change our attention to controlling the AXI-lite registers. For this example, we will be using the script "axi\_runtime.m". Open this script with the following command:



In the top half of this script, we declare a set of System objects for each input and output register we want to interact with. Each System object needs three parameters: a URI address, an offset address for the register, and the input data type. The URI is just the IP address of our radio, and the address offset is provided by HDL Coder. You can view these in the “hdlworkflow.m” script if curious.

The hardware data type is the port input type we used on the model. Unfortunately, we cannot automatically create these objects and must manually translate them from our model.

```
% Frequency Recovery Loop Bandwidth
w1 = matlabshared.libiio.aximm.write('uri',radioIP);
w1.AddressOffset = hex2dec('100');
w1.HardwareDataType='int16';
```

After the object declaration we can use these objects to directly write to registers and read from registers in MATLAB. In the section of code outlined in Figure 17, we update the register values as before.

```
%% Writes
LoopBW = int16(40); %[1 128] Valid
w1(LoopBW);
EQmu = int16(300); % Value Inverted internally
w2(EQmu);
Scope = int8(1); % [1 4] Valid
w3(Scope);
```

Figure 17

In the final section of code, which is shown in Figure 18, we are reading from the output registers and displaying their values. We are taking advantage of the fourth input configuration parameter we added previously. This, like the “SelectScope” input, is also connected to a mux. This mux has several debug counters, or statuses, connected to it. If we provide a specific index, we can check different information internally on the receiver. This way we do not have to expose a large number of registers on the output. In the loop shown in Figure 18 we index through the different debug signal and read out their relevant values.

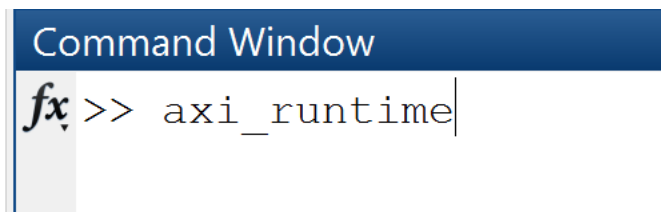
```

%% Continuously read registers
clc;
for k=1:3
    % Check Timing PLL lock
    Error = int8(1); % Select Error index of interest
    w4(Error);
    TimingLocked = r2();
    % Check peaks found by detector
    Error = int8(2); % Select Error index of interest
    w4(Error);
    PeaksFound = r2();
    % Check Frequency PLL lock
    Error = int8(3); % Select Error index of interest
    w4(Error);
    FreqLoopLock = r2();
    % Check direct registers
    numPacketsReceived = r1();
    payloadLen = r3();
    % Display
    table(numPacketsReceived,payloadLen,TimingLocked,FreqLoopLock,...
          PeaksFound)
    pause(1);
end

```

Figure 18

Now Let us see this in action. First, make sure the “interface\_model.slx” model is still running. Next, run the “axi\_runtime.m” script.



```

Command Window
fx>> axi_runtime

```

When this script runs it will output a table of data for each loop iteration. This will be similar to Figure 19 but with different values. Some of the fields are obvious, but the TimingLocked and FreqLoopLoked are boolean status indicators of the PLL lock condition of the timing and frequency recovery loops. PeaksFound counts the number of peaks detected by the receiver correlator. numPacketsReceived will always be greater than PeaksFound.

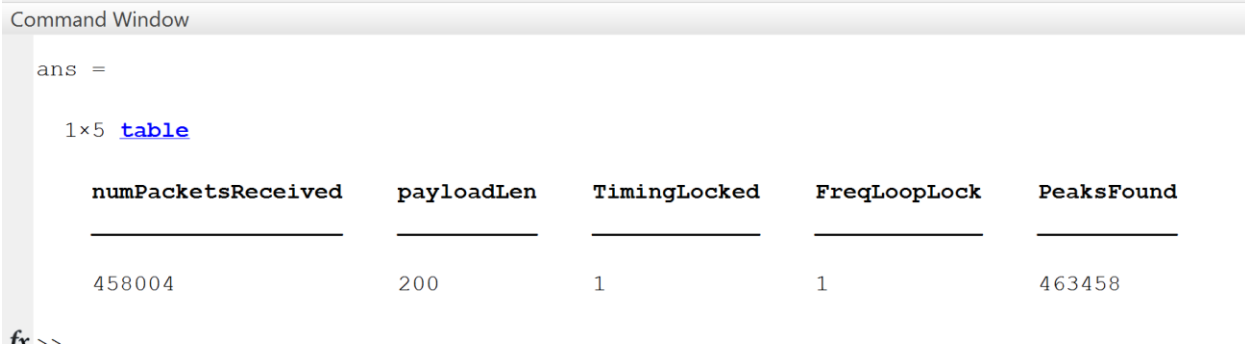
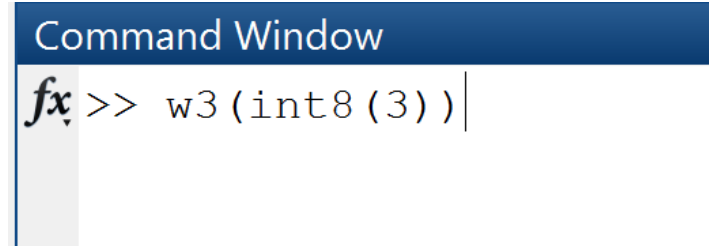


Figure 19

Once this script completes, look at the constellation diagram running in the Simulink model. The constellation should now be different since the “axi\_runtime.m” script updated the “Scope Index” value. To view this change again edit line 37 of the “axi\_runtime.m” script to a different value (between 1 and 4). Alternatively, we can change this through the command line as:



This will select the third Scope Index, which is after Frequency Recovery.

Try different values in the script shown in Figure 20, and watch the constellation diagram and displayed table values in the console.

```
%% Writes
LoopBW = int16(40); %[1 128] Valid
w1(LoopBW);
EQmu = int16(300); % Value Inverted internally
w2(EQmu);
Scope = int8(1); % [1 4] Valid
w3(Scope);
```

Figure 20