LLAMAComm: Lincoln Laboratory Ad-hoc MIMO Adaptive Communications

Adam R. Margetts, Derek P. Young, Bruce McGuffin, Daniel Bliss, Andrew Worthen, David Romero, and Glenn Fawcett
MIT Lincoln Laboratory

Document Version: 2.17, 2016-06-07

Distribution Statement A (Approved for Public Release, Distribution Unlimited)
This work was sponsored by the Defense Advanced Research Projects Agency under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Contents

1 Introduction **1**

1.1 Software Overview 1

2 Tutorial **4**

2.1 Installing LLAMAComm 4

2.2 Running the Example 5

2.2.1 Node Map 6

2.2.2 Timing Diagram Figure 7

2.2.3 Text Output 8

2.2.4 Simulator Output Files 9

2.3 Creating Nodes, Modules, and Environments 11

2.3.1 The Half-Duplex Radios 11

2.3.2 The Full-Duplex Radios 25

2.3.3 The Interference Sources 30

2.3.4 The LL MIMO Nodes 31

2.3.5 The Example Environment 41

2.3.6 The Start Script 42

2.3.7 The Global Variables File 43

2.4 Debugging Tips 44

2.4.1 Timing Diagram 44

2.4.2 Separating the Received Signals 45

2.4.3 Getting the Time-Varying Channel Impulse Response 46

2.4.4 Information in the Workspace 46

2.4.5 Other Debugging Tips 51

2.5 Example: Building a Receive-only Node 51

2.5.1 Receive Callback Function 51

2.5.2 Controller Callback Function 54

2.5.3 Building the Observer Node 58

2.5.4 Simulating the Observer Node 61

3 Behind the Scenes **63**
3.1 Simulation Execution 63
3.1.1 Run Controllers 63
3.1.2 Run Arbitrator 64
3.1.3 Stepping Through an Example Scenario 66
3.1.4 Ending the Simulation Gently 70
3.2 Environment Modeling and Signal Processing 70
3.2.1 Creating Link Objects 70
3.2.2 Combining Signals 71
3.2.3 Local Oscillator Errors 73
3.2.4 Power Measurements 73
3.2.5 Additive Noise 74

4 Reference **75**
4.1 Global Variables Defined 75
4.2 @node 77
4.2.1 Node Properties 77
4.2.2 Node Methods 78
4.3 @module 80
4.3.1 Module Properties 80
4.3.2 Module Methods 83
4.4 @environment 83
4.4.1 Environment Properties 83
4.4.2 Environment Methods 85
4.5 @link 85
4.5.1 Link Properties 85
4.5.2 Link Methods 86
4.6 Utility Functions 86
4.7 File Input and Output 87
4.7.1 Info Source 87
4.7.2 Bits Files, `.bit` 87
4.7.3 Signal Files, `.sig` 88
4.8 Acknowledgements 90

List of Figures

1.1 Example of how a scenario relates to software structure 2

2.1 The canonical example scenario 5
2.2 Node map and detail plot of antenna positions 6
2.3 Screen capture of timing diagram from example simulation 7
2.4 Default plot for full-duplex UserB node and custom plot for LL MIMO receive node 8
2.5 Simulator output files 10
2.6 Example half-duplex nodes 12
2.7 State diagrams for half-duplex controller functions 20
2.8 Example full-duplex nodes 25
2.9 State diagram for full-duplex controller function 28
2.10 Example interference nodes 31
2.11 LL MIMO nodes 31
2.12 State diagrams for LL MIMO controllers 38

3.1 Flowchart of LLAMAComm's main program loop 64
3.2 Flowchart of the arbitrator 65
3.3 Arbitrator Example: Loop 1 66
3.4 Arbitrator Example: Loop 2 68
3.5 Arbitrator Example: Loop 3 69
3.6 Signal processing overview. 71
3.7 Frequency matching block diagram. 72

List of Tables

4.1 `.bit` file block format 88
4.2 `.sig` file block format 89

Chapter 1

Introduction

The Lincoln Laboratory Ad-hoc MIMO Adaptive Communication (LLAMAComm) simulation tool is a software package written in MATLAB. It is designed to provide a framework for simulating cognitive MIMO communication links in the presence of interference. Cognitive radios pose a special challenge because they scavenge for unused bands in a wide range of frequencies. These frequency bands contain different types of interference sources which also need to be simulated.

Users of this software package write functions that define the physical and link layers of a radio under development. LLAMAComm takes these functions and applies propagation models to simulate the transmission of the radio signal waveform. It also coordinates the simulation so that multiple radios may be simulated simultaneously.

1.1 Software Overview

Many of features are realized by making the structure of the simulator *generalized*. LLAMAComm simulations are built around the notion of a node. Nodes represent radios. The nodes are meant to be self-contained; nodes do not share information with other nodes and can only communicate through the wireless channel.

To run a simulation, nodes are thrown into an environment by defining their position in a configuration file. The simulator then interacts with the nodes to automatically coordinate the execution of the simulation. The propagation model and coordination between various transmitting and receiving nodes are taken care of by LLAMAComm.

Nodes in LLAMAComm are implemented as MATLAB objects. The node objects contain a list of parameters, module objects (which describe the radio interface), and a controller function. Figure 1.1 illustrates how a scenario consisting of two FM transceivers and an interfering FM transmitter would be implemented in the simulator. User A, User B, and the FM interferer (labeled `Interference`) are all node objects.

The user-defined node objects contain two module objects each: FM Tx and FM Rx. Module objects contain functions that implement the physical layer of the radio. The transceivers here might use BPSK with a 1 MHz bandwidth, so functions in FM Tx would contain a BPSK modulator and some simple filters. These modules can be reused between different nodes. In this example, User B uses the same FM modulator and demodulator as User A.

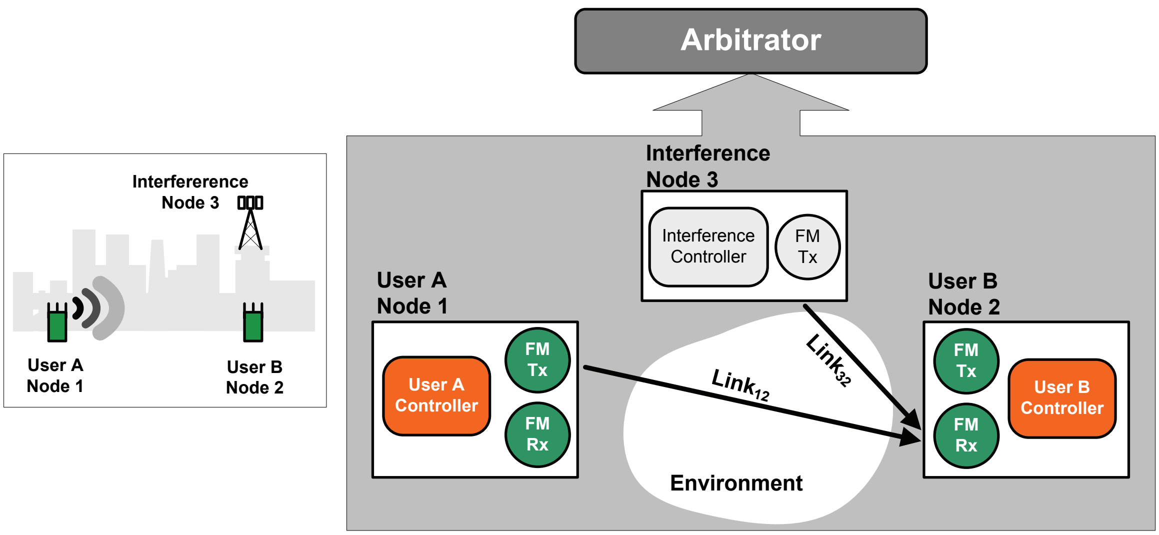


Figure 1.1: Example of how a scenario relates to software structure

The controller functions are state machines written in a specific way so that they can coordinate with the LLAMAComm arbitrator. The coordination is handled through a request/acknowledge system. Nodes that wish to transmit or receive a segment of analog signal must send a request to the arbitrator containing the segment start time and segment duration. The arbitrator coordinates execution order so that any possible interference is included in a segment of received signal. In practice, this coordination is transparent to the user.

The remaining portions of the simulation—including the environment and link objects—are created by the simulator and operate in the background. Basically, the user is responsible for programming the portions of Figure 1.1 that are in color. Since LLAMAComm includes some pre-made interference sources, this example would require the user to write only four MATLAB functions: the FM transmit module, FM receive module, User A controller, and User B controller.

This is a simple example, intended as a quick overview. A more complicated scenario would contain more nodes, and each node may contain more than two modules. There are also other features of the simulator not yet mentioned, such as the special “genie” link. These details, and more, are covered in the following chapters.

Chapter 2

Tutorial

The purpose of this chapter is to demonstrate how to use LLAMAComm through the study of one canonical example. This chapter will first walk through running the example which is provided with the installation archive. It will then delve into the the MATLAB code used to implement the example radios. By the end of this chapter, the reader should be comfortable with the idea of simulating a new radio in LLAMAComm.

2.1 Installing LLAMAComm

LLAMAComm is provided as a single archive file with the extension `.tar.gz`. To install LLAMAComm, simply place the archive file into a working directory. If using UNIX or one of its variants, unpack the archive using the command:
`tar -xzvf llamacommVxxx.tar.gz` (where `xxx` depends on the version of the software). PC users can unpack the archive using WinZip. Successfully unpacking the archive will create the following directory structure:

```
llamacomm/
    docs/
    simulator/
    user_code/
        examples/
            BPSKNodes/
            InterferenceNodes/
            MIMONodes/
```

The directory `docs/` contains LLAMAComm documentation (this file), `simulator/` contains the MATLAB code that implements LLAMAComm, and `user_code/` contains user-defined radios. The installation archive ships with several examples in `user_code/examples/`.

When developing radios for simulation in LLAMAComm, user code should be confined to a folder within `user_code/`. (e.g. `user_code/newradio/`). Code within `simulator/` should never be modified.

2.2 Running the Example

LLAMAComm is set up by default to run the example simulation. Start MATLAB and change the working directory to `installdir/llamacomm/user_code`. Run `StartExample.m`. The example simulation should finish without error in about 30 seconds.

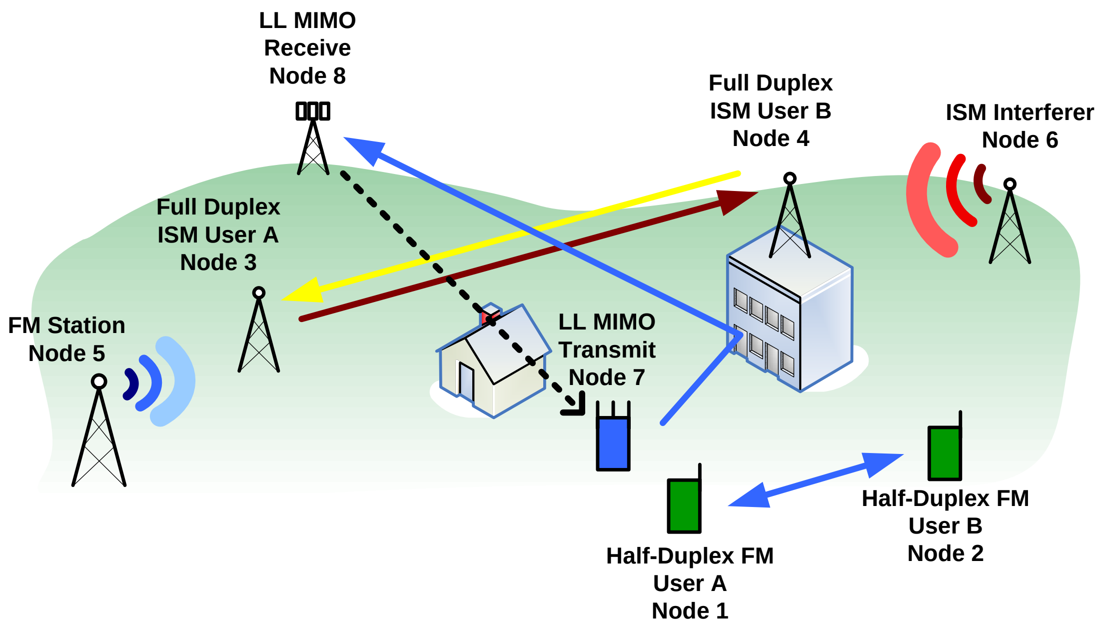
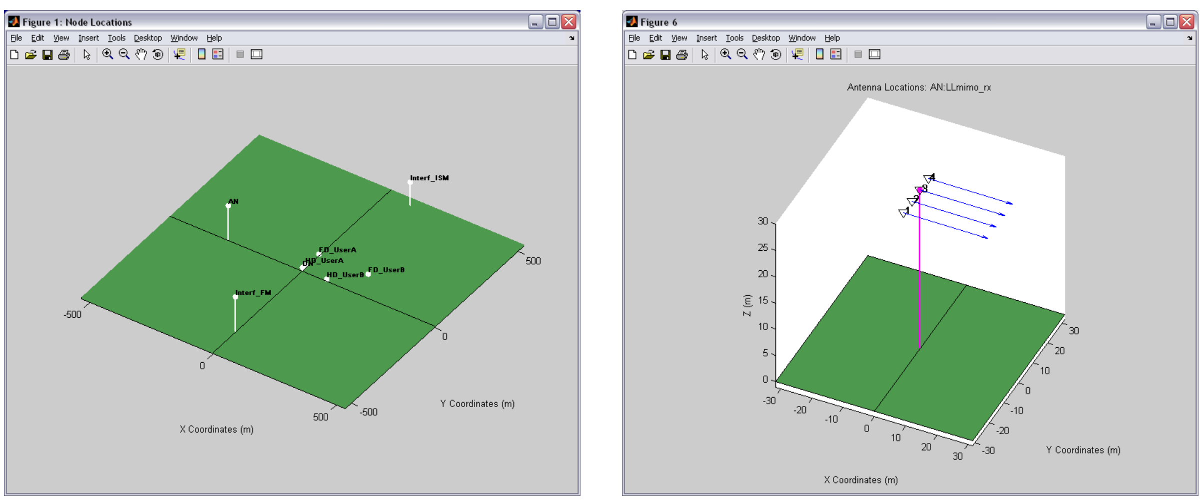


Figure 2.1: The canonical example scenario

Figure 2.1 shows a cartoon of the example scenario. The example is a rural environment containing eight nodes. The color of the arrows corresponds to the frequency of the transmissions. The two half-duplex nodes transmit and receive at the same frequency in the FM radio band. The two full-duplex nodes transmit in two different frequency bands. The yellow arrow corresponds to 1500 MHz, while the dark red arrow corresponds to 2495 MHz (ISM band). Two interferers are present—one in the FM band, and one in the ISM band. The LL MIMO transmitter transmits in the FM band, but uses a special "genie" channel to send channel information from the receiver back to the transmitter. This genie channel is represented by a black dotted line. Note that genie transmit modules can "multi-cast" to multiple genie receive modules.

2.2.1 Node Map

LLAMAComm produces a couple plots by default. When the example scenario is run, a map is produced to show the positions and relative heights of the nodes present in the scenario. Figure 2.2(a) is a screen capture of the node map. The map may be rotated and zoomed using the MATLAB figure toolbar. Clicking with the regular mouse cursor[1] on any of the node names will produce a pop-up plot of the antenna positions and look angles. Figure 2.2(b) shows this detail plot for the node named `AN`.



(a) Node map (b) Antenna map

Figure 2.2: Node map and detail plot of antenna positions

[1]If nothing happens when clicking, check that no buttons (including the arrow) are depressed on the figure toolbar and try again.

2.2.2 Timing Diagram Figure

Signal processing in the simulation is done in a block-wise fashion. These blocks (which we refer to as segments) are drawn in a timing diagram figure as the simulation runs. Figure 2.3 shows a screen capture of the timing diagram.

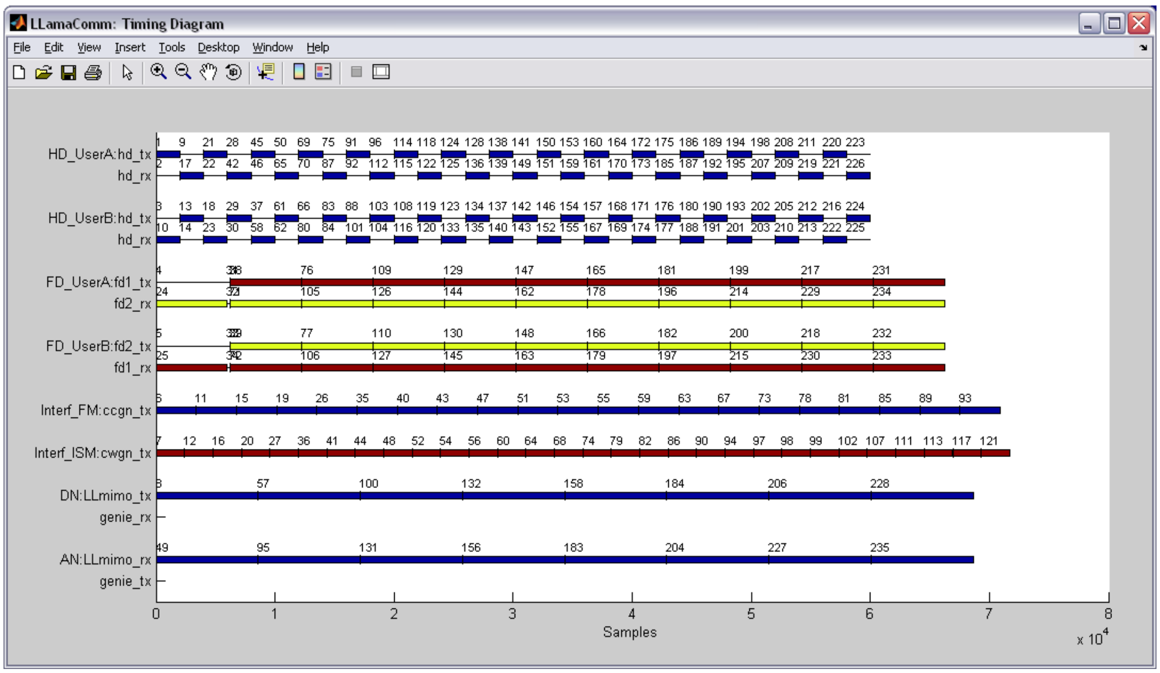
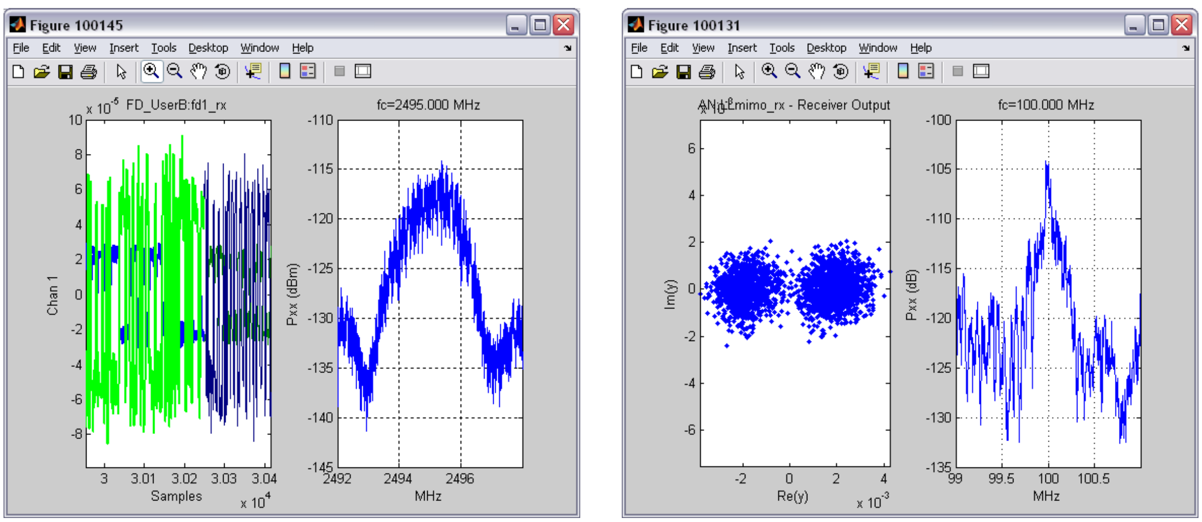


Figure 2.3: Screen capture of timing diagram from example simulation

Each horizontal line in the diagram is associated with a module within a node. For example, the half-duplex radio (labeled `HD_UserA`) contains two modules: `hd_tx` and `hd_rx`. `hd_tx` is the BPSK transmit module, and `hd_rx` is the BPSK receive module. This node alternates between transmitting and receiving because it is operating in half-duplex mode. All the segments are blue because the radios are operating at the same center frequency. In contrast, the full-duplex nodes have a red segments and yellow segments because the modules operate at different frequencies.

The small number near each segment indicates the order in which the segment was processed. This is mainly for debugging purposes, but it is useful to note that the segments are not processed sequentially. Segments are processed out of order so that a receive segment contains the contribution of all in-band signals present in its duration. This scheduling is handled by the simulator.

After the simulation has finished running, clicking on any of the rectangles in the timing diagram with the mouse cursor will bring up a plot of the data in that segment. It also is possible to define a custom plot. Figure 2.4 shows a picture of the default plot and a custom plot. Both plots show the power spectral density in the right subplot. In the left subplot, the default plot shows the real (blue) and imaginary (green) parts of the “analog” waveform. The highlighted portion is the segment that was selected. The adjacent segment is also plotted, but in thinner blue and green lines, so that the continuity of the signal can be examined. The left subplot of the custom plot shows a constellation plot of the beamformer output for the LL MIMO receiver.



(a) Default plot

(b) Custom plot

Figure 2.4: Default plot for full-duplex UserB node and custom plot for LL MIMO receive node

By clicking on other segments in the timing diagram, it should be possible to observe the interaction between the various nodes. For example, segments of signal for `FD_UserA:fd2_rx` appear very clean because they contain no interference. In contrast, receive segments for `HD_UserA:hd_rx` and `HD_UserB:hd_rx` are very noisy because they contain interference from `Interf_FM:ccgn_tx` and `DN:LLmimo_tx`.

2.2.3 Text Output

The simulation also produces a great deal of text output. Like typical MATLAB programs, any function may write to the command window. The output from running the example simulation is shown below. The LL MIMO transmitter and receiver dump many of their operating parameters to screen as the simulation runs. The calculated bit error rates are also displayed at the end. As expected, links with interferers present exhibit a higher bit error rate.

```
 ...
LLMimo Transmit Block 7/8
  txPower: 0.006163
LLMimo Receive Block 7/8
 rxDemod: eig(rNoise) (dB) = -36.6221 -54.2714  -74.452...
 rxDemod: frob(hEst)^2 (dB) = 1.7569
 rxDemod: eig(hhd) (dB) =2.29083 -12.2962 -26.7783 -29.1565
 Passed back to Tx:
         vTx: [0.3667+i*0.1181;0.2149+i*0.1234;0.2007+i*...
    domAtten: 1.6947
Simulation finished successfully.
HD_UserA->HD_UserB BER: 0.001133
HD_UserB->HD_UserA BER: 0.003533
FD_UserA->FD_UserB BER: 0.000300
FD_UserB->FD_UserA BER: 0.000000
LL MIMO example, DN->AN BER:0.010254
Elapsed time is 25.098511 seconds.
>>
```

2.2.4 Simulator Output Files

LLAMAComm produces a number of output files which can be used later for analyzing the results. These files include records of all transmitted/received signals, transmitted/received bits, the timing diagram figure, and workspace-level variables (including all objects). By default, the save directory is `llamacom/user_code/save/timestamp`, where `timestamp` is generated based on the clock. Each run of the simulation will produce a new folder in the `save/` subdirectory with a different timestamp.

Figure 2.5 shows the output produced by running the example scenario. These files are located in `llamacomm/user_code/save/20060406T132616` because the simulation was started at 1:26:16 pm on April 6, 2006.

Files with the extension `.sig` are saved automatically by LLAMAComm. These files contain baseband samples of the transmitted/received waveforms at the simulator's universal sample rate. There is one `.sig` file for each module in the scenario. The naming convention is `nodename-modulename.sig`.

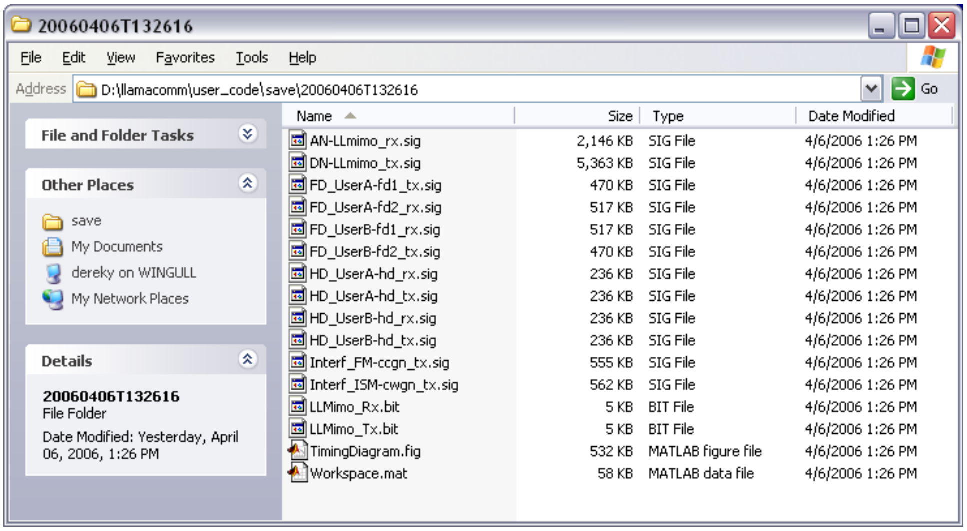


Figure 2.5: Simulator output files

Files with the `.bit` extension keep a record of the binary bits transmitted or received. These files are used to calculate the bit error rate at the end of the simulation. For short simulations, transmitted and received bits can simply be stored within the node objects. For long simulations, however, it is more memory-efficient to save this information to disk. The half-duplex and full-duplex examples in the following sections save the bits within the node object, while the LL MIMO example writes the bits to disk.

`TimingDiagram.fig` is a saved copy of the timing diagram discussed in §2.2.2. Opening the saved figure in MATLAB will bring back the figure window. Callback plots (the plots that appear when the segment rectangles are clicked with the mouse) will function just as before since the data used to generate the plots is stored in the `.sig` files.[2]

`Workspace.mat` contains all the variables accessible from the MATLAB workspace at the completion of a simulation. This includes a copy of all the objects and setup properties. From this information, it is possible to determine all the properties of the finished simulation.

[2]The MATLAB paths may need to be manually configured for the callbacks to work correctly.

2.3 Creating Nodes, Modules, and Environments

The previous section explored the output generated by the example simulation. This section discusses the MATLAB code required to actually implement the example.

LLAMAComm is implemented using MATLAB classes. Instances of a class are known as objects. MATLAB classes are defined by creating a subdirectory with the class name appended to the `@` character. All functions within these class directories have access to object variables—these are referred to as class methods. Functions outside the class directories normally do not have access to object variables.[3]

By using classes, information can be protected from improper sharing or modification. For example, a receiver should not know the true channel matrix, so the channel matrix is hidden within a link object where it cannot be directly accessed by code written for the receiver. For debugging, utilities have been included to display the information to screen.

Using LLAMAComm requires some understanding of the various classes. There are four classes used to implement the simulator: `@node`, `@module`, `@link`, and `@environment`. Users interested in simulating their own designs will need to become familiar with the node and module classes. The environment and link classes are not as important to the user. The environment class is used only once during startup by the user, and the link class is accessed only by internal code.

Because of the divisions enforced by the simulator, the operation of one set of radios does not affect another (other than possibly causing interference, as it would in the real world). As a result, radio pairs can be designed independently. Each example radio in the simulation will be discussed separately in the following sections. We will examine the MATLAB code closely.

2.3.1 The Half-Duplex Radios

The half-duplex radios were the first example created for LLAMAComm. They consist of two nearly identical transceivers that transmit a very simple BPSK signal in the FM frequency band. It is assumed that the radios are already synchronized.

Nodes represent physical radios. Since there are two radios in the example scenario, there are two node objects. Figure 2.6 shows two nodes labeled `HD_UserA` and `HD_UserB`. Conceptually, modules represent the physical layer of a communication system, and controllers represent the link layer as well as any operator actions. Modules are responsible for the modulation/demodulation and coding/decoding of a signal. The

[3]Please refer to the MATLAB documentation on programming with classes for more detail.

controller determines when to transmit/receive and what to do with the data. The controller implements the functionality of the microcontroller in a physical radio.

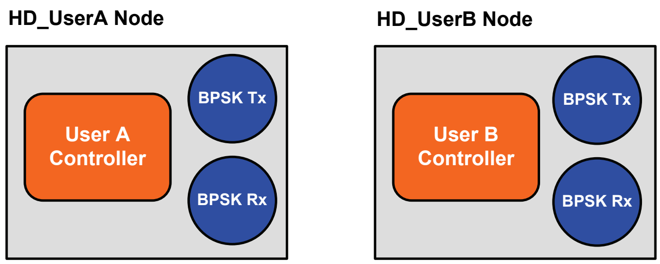


Figure 2.6: Example half-duplex nodes

In LLAMAComm, modules must either be transmitters or receivers. So, for a transceiver, the minimum number of modules is two. Note that for certain communication systems, it may be desirable to have multiple receive modules (a software radio, for example). This can be simulated by adding any number of module objects into the node. It is the controller's responsibility to decide which modules to use at any given time for sending or receiving data.

Each node object in Figure 2.6 is shown to contain a controller function and two module objects labeled `BPSK Tx` and `BPSK Rx`. The module objects are the same in the two radios since they transmit in the same frequency using the same transmission scheme. The two controller functions are very similar; they are different only in that one controller starts in a transmit state while the other starts in a receive state.

There are six files in the `user_code/examples/BPSKNodes` directory that are used to implement the half-duplex radios: `HD_BuildNodes.m`, `BPSK_Receive.m`, `BPSK_Transmit.m`, `HD_UserA_Controller.m`, `HD_UserB_Controller.m`, `HD_CalculateBER.m`.

HD_BuildNodes.m

The basic steps for building any LLAMAComm node are shown below. In all the example radios provided, these steps are incorporated into a "top-level" function with the name `*_BuildNodes.m`.

1. Specify required transmit and receive module properties

2. Provide handle to callback function that does the modulation/demodulation

3. Call module class constructor to create module objects

4. Specify required node properties
5. Provide handle to controller function
6. Include copies of the newly created module objects
7. Call node class constructor and return new node objects

Code to construct the half-duplex nodes is found in `llamacomm/user_code/examples/BPSKNodes/HD_BuildNodes.m`. Let us walk through the file. The block of code listed below is the function header and help comments. The build functions return an array of completed node objects.

```
function hdNodes = HD_BuildNodes


% Function HD_BuildNodes.m:
% Example function/script for building example half-duplex nodes with
% common transmit/receive modules and user parameters.
%
% USAGE: hdNodes = BuildUserNodes
%
% Input arguments:
%  -none-
%
% Output arguments:
%  hdNodes  (1xN Node obj array) Newly-created user nodes
%
```

The next block of code defines two structs that contain all the required properties for building a module. `hd_tx_mod_params` defines the transmit module and `hd_rx_mod_params` defines the receive module.[4]

```
% Define transmit/recieve modules

hd_tx_mod_params.name = 'hd_tx';
hd_tx_mod_params.callbackFcn = @BPSK_Transmit;
hd_tx_mod_params.fc = 98.5e6;
hd_tx_mod_params.type = 'transmitter';
hd_tx_mod_params.loError = 0;
hd_tx_mod_params.antType = {'dipole_halfWavelength'};
hd_tx_mod_params.antPosition = [0 0 0];
```

[4]Future versions of this code may have default values assigned for some variables such as wall material and antenna type.

```
   hd_tx_mod_params.antPolarization = {'v'};
32 hd_tx_mod_params.antAzimuth = [0];
   hd_tx_mod_params.antElevation = [0];
34 hd_tx_mod_params.exteriorWallMaterial = 'none';
   hd_tx_mod_params.distToExteriorWall = [0];
36 hd_tx_mod_params.exteriorBldgAngle = [0];
   hd_tx_mod_params.numInteriorWalls = [0];
38
   hd_rx_mod_params.name = 'hd_rx';
40 hd_rx_mod_params.callbackFcn = @BPSK_Receive;
   hd_rx_mod_params.fc = 98.5e6;
42 hd_rx_mod_params.type = 'receiver';
   hd_rx_mod_params.loError = 0;
44 hd_rx_mod_params.antType = {'dipole_halfWavelength'};
   hd_rx_mod_params.antPosition = [0 0 0];
46 hd_rx_mod_params.antPolarization = {'v'};
   hd_rx_mod_params.antAzimuth = [0];
48 hd_rx_mod_params.antElevation = [0];
   hd_rx_mod_params.exteriorWallMaterial = 'none';
50 hd_rx_mod_params.distToExteriorWall = [0];
   hd_rx_mod_params.exteriorBldgAngle = [0];
52 hd_rx_mod_params.numInteriorWalls = [0];
   hd_rx_mod_params.noiseFigure = 6; % (dB) noise figure of receiver
54
   hd_tx_mod = module(hd_tx_mod_params);
56 hd_rx_mod = module(hd_rx_mod_params);
```

Module names (`hd_tx` and `hd_rx`, in this example) must be unique for a given node. `callbackFcn` is the function handle to the function that performs the modulation/demodulation. In this case, the functions are `BPSK_Receive.m` and `BPSK_Transmit.m`. MATLAB syntax uses the `@` symbol to turn a function name into a function handle. `fc` specifies the center frequency of the modulated signal. `type`, is a string that tells LLAMAComm whether the module is a transmitter or a receiver. Details on the other properties can be found in the reference section of the documentation.

The last two lines of the block of code shown above (lines 49-50) are calls to the module class constructor. The constructor expects a struct with the specific field names provided in the example. It returns a module object. In this example, two modules are produced: `hd_tx_mod` and `hd_rx_mod`.

```
   % Define nodes
60
   userA_node_params.name = 'HD_UserA';
```

```
userA_node_params.location = [0 20 3];
userA_node_params.velocity = [0 0 0];
userA_node_params.controllerFcn = @HD_UserA_Controller;
userA_node_params.modules = [hd_tx_mod hd_rx_mod];

userB_node_params.name = 'HD_UserB';
userB_node_params.location = [100 0 2];
userB_node_params.velocity = [0 0 0];
userB_node_params.controllerFcn = @HD_UserB_Controller;
userB_node_params.modules = [hd_tx_mod hd_rx_mod];

userA_node = node(userA_node_params);
userB_node = node(userB_node_params);
```

The code block above creates two node objects. It is similar in structure to the code used to create the module objects. The first parts assigns properties in a struct. The last two lines call the node class constructor.

The node name must be unique within a scenario. The node location is specified in rectangular coordinates in meters. The node velocity is in meters/second. The property `controllerFcn` is a function handle. In this case, node `HD_UserA` is using the controller function defined by `HD_UserA_Controller.m`.

The `modules` property should be an array of module objects. This example uses the two modules defined earlier in the build function: `hd_tx_mod` and `hd_rx_mod`.

The next block of code in the file sets user-defined parameters in the node objects.

```
% Set user parameters

userParams.power = 10; % (Watts)
userParams.dataLen = 800;
userParams.trainingLen = 200;
userParams.nOversamp = 2;
userParams.bitLen = userParams.dataLen+userParams.trainingLen;
userParams.blockLen = userParams.nOversamp*userParams.bitLen;

userParams.trainingSeq = rand(1,userParams.trainingLen)>.5;

userParams.nBlocksToSim = 15;
userParams.receivedBlocks = 0;
userParams.transmittedBlocks = 0;

userParams.txBits = zeros(userParams.nBlocksToSim,...
                          userParams.bitLen,'uint8');
```

```
userParams.rxBits = zeros(userParams.nBlocksToSim,...
                          userParams.bitLen,'uint8');

userParams.equalizerLen = 21;
userParams.equalizerDelay = 10;

userA_node = SetUserParams(userA_node,userParams);
userB_node = SetUserParams(userB_node,userParams);
```

User parameters are a special part of the node. These parameters, which consist of anything that can be put into a single MATLAB structure, can be written and read at the user's discretion. This area is useful for storing information that a physical radio would have in its EEPROM, such as a training sequence, packet length, or initial transmit power. The user parameters are also useful for passing information between the controller function and the transmit/receive functions. In fact, *this is the only method of communication between the modules and the controller function.*

The user parameters are stored within a node, so they are accessed using class methods: `SetUserParams()` and `GetUserParams()`. This example sets the power, defines the data block structure, and sets aside memory to hold transmitted and received bits. (This works since this is a small example. Larger examples such as LL MIMO store the bits to file.) The last two lines call `SetUserParams()` to store the structure into the node object.

```
% Put user nodes into array
hdNodes = [userA_node userB_node];
```

Finally, the last line of the function returns the two newly created half-duplex nodes in a 1x2 array.

BPSK_Transmit.m and BPSK_Receive.m

As mentioned in the previous sections, `BPSK_Transmit.m` and `BPSK_Receive.m` define the physical layer of the example half-duplex radios. Code from `BPSK_Transmit.m` is listed below.

```
function [nodeobj,sig] = BPSK_Transmit(nodeobj,modname,blockLen)

% Function BPSK_Transmit.m:
% This is an example of a user-written callback function for a module
% object.  It generates or loads data to transmit. The output signal
% should be analog baseband, and can be multichannel.
%
```

```
% This example is a single-channel BPSK transmitter.  2x oversampled.
% There is no filtering or pulse shaping, so the signal is full bandwidth.
%
% USAGE: [nodeobj,sig] = BPSK_Transmit(nodeobj,modname,blockLen)
%
% Input arguments:
%  nodeobj    (node obj) Parent node object
%  modname    (string) The name of the module that has activated this
%              callback function
%  blockLen   (int) The block length of the analog signal expected
%              by the arbitrator
%
% Output arguments:
%  nodeobj    (node obj) Modified copy of the node object
%  sig        (NxblockLen) Analog baseband signal for N channels
%
% See also: BPSK_Receive.m
```

Transmit and receive callback functions must use these exact input and output arguments. This is required to properly interface to the simulator's arbitrator function. `BPSK_Transmit()` takes in a node object, the module name, and the block length of the signal to be transmitted. It returns a modified node object (the user properties will be modified) and a single segment of complex baseband signal at the simulator's universal baseband sample rate.

The function starts by retrieving the user parameters. User parameters are stored in the node object and must be retrieved using `GetUserParams()`. This is shown in the following block of code.

```
% Load user parameters
p = GetUserParams(nodeobj);
```

In the example, data is randomly generated. This is shown in the code fragment below. The code produces random ± 1's and attaches the fixed training sequence. The data is then duplicated using `repmat` to give an effect data rate of `fs/2`. `fs` is the universal simulation sample rate set in `InitGlobals.m`. It is set by default to 12.5 MHz.

```
% Make some random data and attach training sequence
randdata = rand(1,p.dataLen)>.5;
bits = [p.trainingSeq randdata];
sig = -(bits*2-1);

% Upsample by duplicating bits
```

```
sig = repmat(sig,p.nOversamp,1);
sig = sig(:).';
```

It is also possible to read data from file or generate it in the controller function and then pass the data block to the transmit function. In order keep the distinction between physical layer and link layer as sharp as possible, it is preferable to build the data block in the controller function if memory constraints allow. This is done in the LL MIMO example (§2.3.4).

```
% Calculate signal power (Watts)
filePow = var(sig);

% Set the transmitted signal power
sig = sqrt(p.power/(filePow))*sig;
```

The power of the transmitted signal is adjusted. The units of `sig` are volts. The units of power are watts. Power is calculated by assuming a 1Ω load. This convention is followed throughout the simulation.

```
% Check blockLen
if size(sig,2) ~= blockLen
    error('Block length of Tx data is wrong.');
end
```

The value of `blockLen` is set by the controller function. It is passed into the transmit function by the arbitrator. This block of code does a quick check to make sure that the final block length of the transmit signal is correct.

```
% Save txbits
p.transmittedBlocks = p.transmittedBlocks+1;
p.txBits(p.transmittedBlocks,:) = bits;

% Save user params
nodeobj = SetUserParams(nodeobj,p);
```

Finally, `SetUserParams()` must be used to save the transmitted bits into the user parameters. The values are later used to calculate the bit error rate.

`BPSK_Receive.m` is very similar to `BPSK_Transmit.m`. The difference is that it receives and demodulates the bits. The code for `BPSK_Receive` is included below for comparison.

```
function nodeobj = BPSK_Receive(nodeobj,modname,sig)

```

```
% Function BPSK_Receive.m:
% This is an example of a user-written callback function for a
% receive module.  This example demodulates an analog, BPSK signal.
%
% This example is designed to work with BPSK_Transmit
%
% USAGE: nodeobj = BPSK_Receive(nodeobj,modname,sig)
%
% Input arguments:
%  nodeobj    (node obj) Parent node object
%  modname    (string) The name of the module that has activated this
%              callback function
%  sig        (NxblockLen) Analog baseband signal for N channels
%
% Output arguments:
%  nodeobj    (node obj) Modified copy of the node object
%
% See also: BPSK_Transmit.m

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (c) 2006-2016 Massachusetts Institute of Technology %
% All rights reserved.   See software license below.            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Load user parameters
p = GetUserParams(nodeobj);

% Downsample
sig = sig(1:p.nOversamp:end);

% Demod
nSamps = length(sig);
nTrain = p.trainingLen;
sTrain = -(p.trainingSeq*2- 1);

% adapt equalizer
eqLen = p.equalizerLen;
eqDelay = p.equalizerDelay;
R = toeplitz([sig(1) zeros(1,eqLen-1)], sig(1:nTrain));
Sd = [zeros(1,eqDelay),sTrain(1:nTrain-eqDelay)];
f = Sd*R'*inv(R*R');

% equalize and extract bit estimates
nBits = p.bitLen;
sHat = conv(sig,f);
```

```
bits = real(sHat(eqDelay+1:nBits+eqDelay))<0;

% Save rx bits
p.receivedBlocks = p.receivedBlocks+1;
p.rxBits(p.receivedBlocks,:) = bits;

% Save user params
nodeobj = SetUserParams(nodeobj,p);
```

HD_UserA_Controller.m and HD_UserB_Controller.m

The structure for any controller function can be described as a finite state machine. All the examples were developed by first drawing the state diagram. Figure 2.7 shows the state diagrams for the `HD_UserA` controller and the `HD_UserB` controller. The outputs are shown in blue, and the inputs are shown in green.

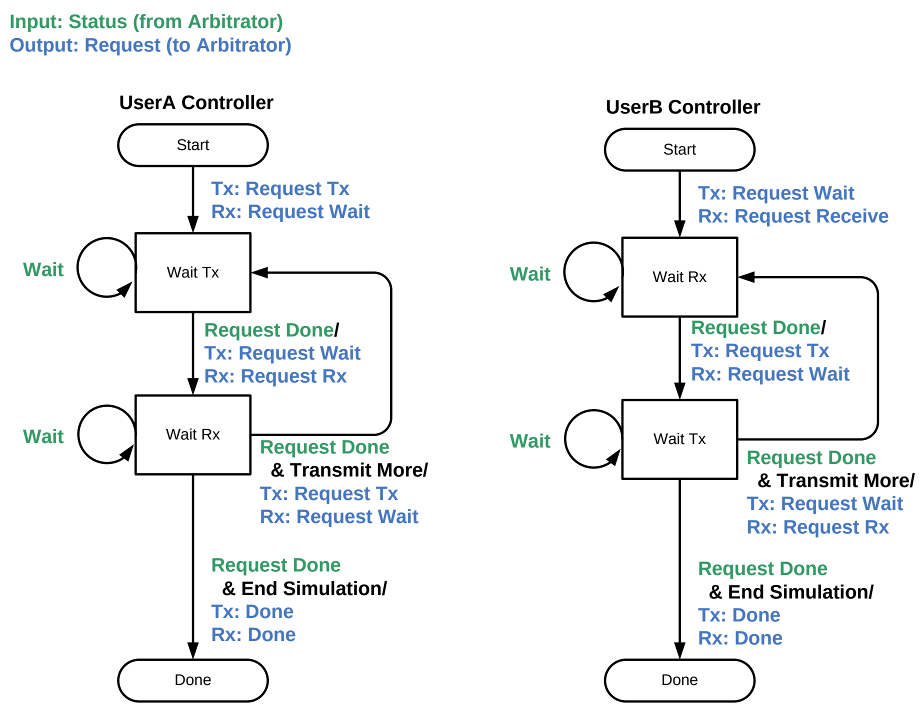


Figure 2.7: State diagrams for half-duplex controller functions

In order to transmit or receive, a request must be made to the arbitrator. During each iteration of the simulation loop, the controller checks the status of the node's modules to see if the requests are complete. Once complete, the controller moves on to the next state. Otherwise, it continues to wait. Note that it is possible to write a controller so that the simulation stalls. The arbitrator is designed to detect such a state and abort the simulation. This will be discussed in more detail in Section 3.1.

Implementing the state machines shown in Figure 2.7 is relatively straight-forward. The code for `HD_UserA_Controller` is examined in the listing below.

```
1  function [nodeobj,status] = HD_UserA_Controller(nodeobj)

3  % Function HD_UserA_Controller.m:
   % Controller state machine for example Half-Duplex radio.
5  % These controllers assume that the user nodes are already synchronized.
   % The UserA and UserB controllers are identical except that UserA starts
7  % by sending a block and UserB starts by receiving a block.
   % The controller switches to the done state after receiving/sending a set
9  % number of blocks.
   %
11 % USAGE: [nodeobj,status] = HD_UserA_Controller(nodeobj)
   %
13 % Input arguments:
   %  nodeobj    (node obj) Parent node object.
15 %
   % Output arguments:
17 %  nodeobj    (node obj) Modified copy of node object
   %  status     (string) Either 'running' or 'done'
19 %

21 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   % Copyright (c) 2006-2016 Massachusetts Institute of Technology %
23 % All rights reserved.   See software license below.            %
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
25
   status = 'running';  % Default status
```

The function headers for all controller functions need to have the same input and outputs. The controller function takes in a node object and returns a modified node object and a status string. The status string is used to detect when a simulation is finished. On line 21, the status is set to `'running'`. When the controller reaches the `done` state, the status is set to `'done'` (line 69).

The user parameters and current state are stored within the node object. They are extracted using the functions `GetUserParams()` and `GetNodeState()`. This is shown below.

```
% Load user parameters
p = GetUserParams(nodeobj);

% Get current node state
currState = GetNodeState(nodeobj);
```

The next code block implements the state machine shown in Figure 2.7. When a module needs to do something, it raises a request flag that the simulation arbitrator sees. Requests are set using the function `SetModuleRequest()`. A request consists of the node object, the module name, a job (`wait`, `transmit`, `receive`, or `done`), and a block length. The block length tells the arbitrator how many samples are in the segment of signal to be processed. The arbitrator uses this information to determine the execution order of the simulation.

```
% State machine next-state and output "logic"
switch currState
    case 'start'
        % Set up module for transmission of 1 block
        nodeobj = SetModuleRequest(nodeobj,'hd_tx','transmit',p.
            blockLen);
        nodeobj = SetModuleRequest(nodeobj,'hd_rx','wait',p.blockLen);
        nextState = 'transmit_wait';

    case 'transmit_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            % Transmission done, receive
            nodeobj = SetModuleRequest(nodeobj,'hd_tx','wait',p.
                blockLen);
            nodeobj = SetModuleRequest(nodeobj,'hd_rx','receive',p.
                blockLen);
            nextState = 'receive_wait';
        else
            % All requests not satisfied, wait
            nextState = 'transmit_wait';
        end

    case 'receive_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            if  p.receivedBlocks>=p.nBlocksToSim
```

```
                % Goto done
                nodeobj = SetModuleRequest(nodeobj,'hd_tx','done');
                nodeobj = SetModuleRequest(nodeobj,'hd_rx','done');
                nextState = 'done';
            else
                % Receive done, go back to transmit
                nodeobj = SetModuleRequest(nodeobj,'hd_tx','transmit',p.blockLen);
                nodeobj = SetModuleRequest(nodeobj,'hd_rx','wait',p.blockLen);
                nextState = 'transmit_wait';
            end
        else
            % Requests pending, wait
            nextState = 'receive_wait';
        end

    case 'done'
        % Done!
        nextState = 'done';
        status = 'done';

    otherwise
        error(sprintf('Unknown state: %s',currState));
end
```

As the simulation runs, the arbitrator lowers request flags that have been satisfied. `CheckRequestFlags()` checks to see if requests for all modules within the node have been satisfied.

For every sample in the simulation, each module must be transmitting, receiving, or waiting. If this rule is not followed, the simulation will stall. The only exception is the `done` job. Once a module is marked as `done`, it is ignored by the arbitrator.

Finally, the state and the user parameters are stored using `SetNodeState()` and `SetUserParams()` in the code block listed below. This saves the state and user parameters so that they can be loaded the next time the controller function is run.

```
% Set next state
nodeobj = SetNodeState(nodeobj,nextState);

% Store possibly modified user params
nodeobj = SetUserParams(nodeobj,p);
```

HD_CalculateBER.m

The bit error rate (BER) is a commonly calculated metric for communication simulations. The following function demonstrates how to calculate the BER in a simple simulation where the sent and received bits are stored within the user parameters.

```
function ber = HD_CalculateBER(nodes)

% Function HD_CalculateBER.m:
% Calculates the Bit-Error Rate for each of the two user nodes.
%
% USAGE:  ber = HD_CalculateBER(nodes)
%
% Input arguments:
%  nodes     (node obj array, 1xN) Node objects
%
% Output arguments:
%  ber       (1x2) HD_UserA->HD_UserB and HD_UserB->HD_UserA BER
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (c) 2006-2016 Massachusetts Institute of Technology %
% All rights reserved.   See software license below.            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Extract user parameters
UA = GetUserParams(FindNode(nodes,'HD_UserA'));
UB = GetUserParams(FindNode(nodes,'HD_UserB'));

% Calculate BER
errAB = find(xor(UA.txBits,UB.rxBits));
errBA = find(xor(UB.txBits,UA.rxBits));

berAB = length(errAB)/prod(size(UA.txBits));
berBA = length(errBA)/prod(size(UB.txBits));

ber(1) = berAB;
ber(2) = berBA;

% Print results to screen
fprintf('HD_UserA->HD_UserB BER: %f\n',berAB);
fprintf('HD_UserB->HD_UserA BER: %f\n',berBA);
```

This function is designed to be run from the top level of the simulation. It has access to all nodes in the simulator. The relevant node is found using `FindNode()`. The transmitted and received bits are extracted using `GetUserParams()`. The bits are compared, and the results are printed to screen.

2.3.2 The Full-Duplex Radios

The full-duplex radio example is very similar to the half-duplex example. The difference is that the transmit and receive modules operate at different frequencies, and the controller function activates both transmit and receive modules simultaneously.

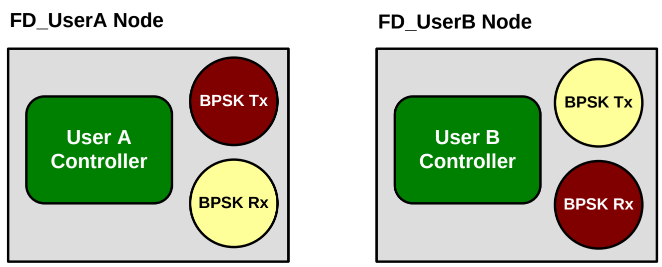


Figure 2.8: Example full-duplex nodes

Figure 2.8 illustrates the object construction that is required to implement this example. As before, we will look at the relevant portions of code that were required to implement this example. This time, however, we will concentrate mainly on the differences between this example and the half-duplex example.

FD_BuildNodes.m

In the half-duplex example from §2.3.1, we built two modules and used copies of the two in each node. In the full-duplex example, however, we need to build four nodes since the transmit and receive frequencies are different.

The following fragments of code from `user_code/examples/BPSKNodes/FD_BuildNodes.m` illustrate the important differences between the half-duplex example and this example. (Some code has been snipped for clarity).

```
function fdNodes = FD_BuildNodes
...

% Define transmit/recieve modules
```

```
fd1_tx_mod_params.name = 'fd1_tx';
fd1_tx_mod_params.callbackFcn = @BPSK_Transmit;
fd1_tx_mod_params.fc = 2495e6;
fd1_tx_mod_params.type = 'transmitter';
...

fd1_rx_mod_params.name = 'fd1_rx';
fd1_rx_mod_params.callbackFcn = @BPSK_Receive;
fd1_rx_mod_params.fc = 2495e6;
fd1_rx_mod_params.type = 'receiver';
...

fd2_tx_mod_params.name = 'fd2_tx';
fd2_tx_mod_params.callbackFcn = @BPSK_Transmit;
fd2_tx_mod_params.fc = 1500e6;
fd2_tx_mod_params.type = 'transmitter';
...

fd2_rx_mod_params.name = 'fd2_rx';
fd2_rx_mod_params.callbackFcn = @BPSK_Receive;
fd2_rx_mod_params.fc = 1500e6;
fd2_rx_mod_params.type = 'receiver';
...

fd1_tx_mod = module(fd1_tx_mod_params);
fd1_rx_mod = module(fd1_rx_mod_params);
fd2_tx_mod = module(fd2_tx_mod_params);
fd2_rx_mod = module(fd2_rx_mod_params);
```

The code block above builds the four modules used in this example. Note that these four modules have unique names. The center frequencies are set at 2495 MHz and 1500 MHz. Also, the BPSK receive and transmit functions were reused for this example. The next block of code in the file constructs the node objects.

```
% Define nodes

FD_userA_node_params.name = 'FD_UserA';
FD_userA_node_params.location = [0 100 2];
FD_userA_node_params.velocity = [0 0 0];
FD_userA_node_params.controllerFcn = @FD_UserA_Controller;
FD_userA_node_params.modules = [fd1_tx_mod fd2_rx_mod];

FD_userB_node_params.name = 'FD_UserB';
FD_userB_node_params.location = [200 100 2];
```

```
FD_userB_node_params.velocity = [0 0 0];
FD_userB_node_params.controllerFcn = @FD_UserB_Controller;
FD_userB_node_params.modules = [fd2_tx_mod fd1_rx_mod];

FD_userA_node = node(FD_userA_node_params);
FD_userB_node = node(FD_userB_node_params);
```

The user parameters (shown below) are similar to the half-duplex example. For fun, `nOversamp` was increased to 3. This property is used within `BPSK_Receive()` and `BPSK_Transmit()`. Increasing `nOversamp` has the effect of increasing the number of samples per bit for the full-duplex transceivers.

```
% Set user parameters

userParams.power = 10; % (Watts)
userParams.dataLen = 1800;
userParams.trainingLen = 200;
userParams.nOversamp = 3;
userParams.bitLen = userParams.dataLen+userParams.trainingLen;
userParams.blockLen = userParams.nOversamp*userParams.bitLen;

userParams.trainingSeq = rand(1,userParams.trainingLen)>.5;

userParams.nBlocksToSim = 10;
userParams.receivedBlocks = 0;
userParams.transmittedBlocks = 0;

userParams.txBits = zeros(userParams.nBlocksToSim,...
                          userParams.bitLen,'uint8');
userParams.rxBits = zeros(userParams.nBlocksToSim+1,...
                          userParams.bitLen,'uint8');

userParams.equalizerLen = 21;
userParams.equalizerDelay = 10;

userA_node = SetUserParams(FD_userA_node,userParams);
userB_node = SetUserParams(FD_userB_node,userParams);


% Put nodes into output array
fdNodes = [userA_node userB_node];
```

At the end of `FD_BuildNodes.m`, the user parameters are saved into the node object, and the nodes are returned in a 1x2 array.

FD_UserA_Controller.m

This section discusses the controller function for the full-duplex node `UserA` (the controller for `UserB` is almost identical). This controller is noticeably different from the controllers for the half-duplex examples. In this example, nodes both transmit and receive at the same time. There are also some additional states added for listening before the start of communication. In a more complicated example, this listening period might be used to listen for a synchronization sequence or for transmissions from possible interference sources. Figure 2.9 shows the state diagram for the controller.

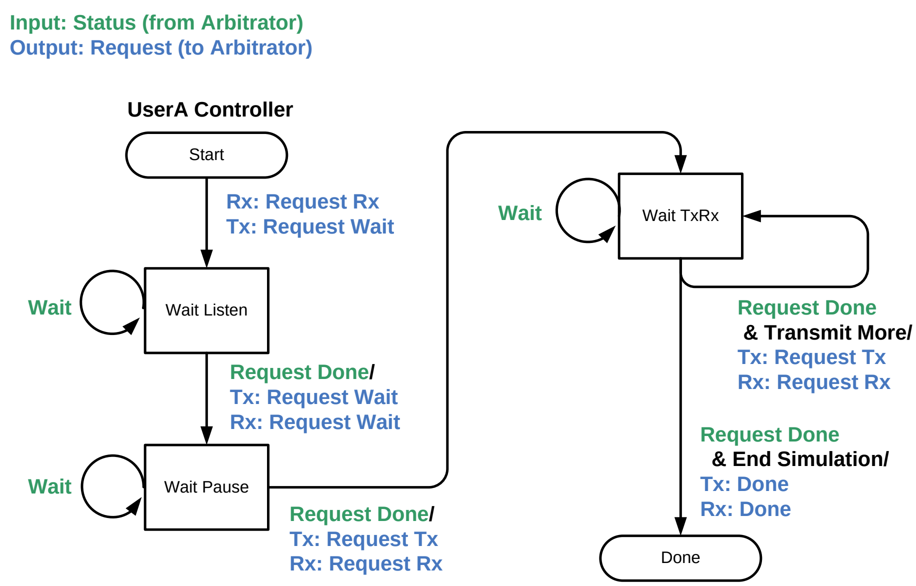


Figure 2.9: State diagram for full-duplex controller function

The corresponding MATLAB code for the state machine is shown in the following listing.

```
function [nodeobj,status] = FD_UserA_Controller(nodeobj)
...

status = 'running';  % Default status

% Load user parameters
p = GetUserParams(nodeobj);

% Get current node state
currState = GetNodeState(nodeobj);
```

```

% State machine next-state and output "logic"
switch currState
    case 'start'
        % Listen for awhile
        nodeobj = SetModuleRequest(nodeobj,'fd2_rx','receive',p.
            blockLen);
        nodeobj = SetModuleRequest(nodeobj,'fd1_tx','wait',p.blockLen);
        nextState = 'listen_wait';

    case 'listen_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            % Pause for awhile
            nodeobj = SetModuleRequest(nodeobj,'fd2_rx','wait',250);
            nodeobj = SetModuleRequest(nodeobj,'fd1_tx','wait',250);

            nextState = 'pause_wait';
        else
            % Requests pending, wait
            nextState = 'listen_wait';
        end

    case 'pause_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            % Start tx/rx
            nodeobj = SetModuleRequest(nodeobj,'fd2_rx',...
                                        'receive',p.blockLen);
            nodeobj = SetModuleRequest(nodeobj,'fd1_tx',...
                                        'transmit',p.blockLen);
            nextState = 'txrx_wait';
        else
            % All requests not satisfied, wait
            nextState = 'pause_wait';
        end

    case 'txrx_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            if p.transmittedBlocks >= p.nBlocksToSim
                % Goto done
                nextState = 'done';
                nodeobj = SetModuleRequest(nodeobj,'fd2_rx','done');
                nodeobj = SetModuleRequest(nodeobj,'fd1_tx','done');
```

```
            else
                % Tx/Rx again
                nodeobj = SetModuleRequest(nodeobj,'fd2_rx',...
                                                'receive',p.blockLen);
                nodeobj = SetModuleRequest(nodeobj,'fd1_tx',...
                                                'transmit',p.blockLen);
                nextState = 'txrx_wait';
            end
        else
            % All requests not satisfied, wait
            nextState = 'txrx_wait';
        end

    case 'done'
        % Done!
        nextState = 'done';
        status = 'done';

    otherwise
        error(sprintf('Unknown state: %s',currState));
end

% Set next state
nodeobj = SetNodeState(nodeobj,nextState);

% Store possibly modified user params
nodeobj = SetUserParams(nodeobj,p);
```

The structure of the code is similar to the previous half-duplex example. The state machine is implemented as a large switch statement. The state and user parameters are loaded at the beginning of the function, and saved at the end. This general structure should be flexible enough to implement any controller function.

2.3.3 The Interference Sources

The example scenario includes two interference sources: one in the FM radio band (99.5 MHz), and one in the ISM band (2495 MHz). The FM interferer transmits complex colored gaussian noise. The ISM interferer transmits complex white gaussian noise. Interference sources are transmit-only, so contain only a transmit module. Figure 2.10 shows a diagram of the example interference nodes.

There are five files in the `user_code/examples/InterferenceNodes` directory. The interference sources are constructed in `Ex_BuildInterference.m`.

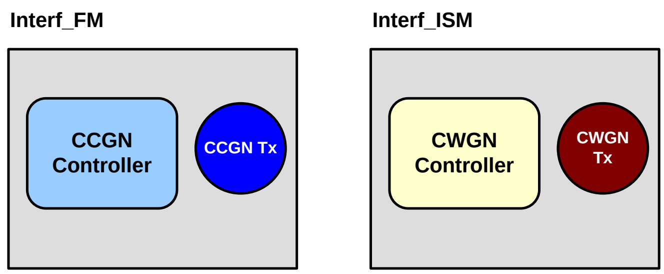


Figure 2.10: Example interference nodes

`CCGN_Controller.m` and `CWGN_Controller.m` define the controller functions. `CWGN_Transmit.m` and `CCGN_Transmit.m` define the transmit functions. It is left to the interested reader to examine the source code on her own.

One thing to note within the controller functions is that the modules are never marked as "done" (by using `SetModuleRequest(...,'done')`). This is so that the simulation will stall if there are not enough interference samples created.

2.3.4 The LL MIMO Nodes

The final nodes included in the canonical example are the Lincoln Laboratory MIMO nodes. These demonstrate the simulation of a very basic MIMO (Multiple Input Multiple Output) communication system Files related to these nodes can be found in the `user_code/examples/MIMONodes` directory. This example also makes use of the special "genie" channel for quickly implementing a reverse link, and defines custom plots for the timing diagram.

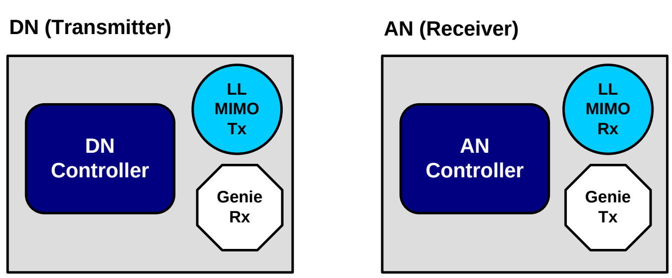


Figure 2.11: LL MIMO nodes

Figure 2.11 shows the node structure for the LL MIMO nodes.`DN` is an acronym for "disadvantaged node", and `AN` stands for "advantaged node". The genie channel is implemented by including genie modules. The genie modules are shown as octagons in the figure.

LLMimo_BuildNodes.m

The LL MIMO nodes are constructed in `LLMimo_BuildNodes.m`. The structure of the file should be very familiar to anyone who has read the previous sections explaining the half-duplex and full-duplex examples. The only differences are the inclusion of the genie modules, and the way the user parameters are defined. Portions of the file are explained below.

```
function LLMimoNodes = LLMimo_BuildNodes(range, envType)

% Function LLMimo_BuildNodes.m:
% Example function/script for building user nodes with common
% transmit/receive modules and user parameters.
%
% USAGE: LLMimoNodes = LLMimo_BuildNodes
%
% Input arguments:
%  range        (string) range is 'short', 'short-medium', 'medium', or 'long'
%  envType      (string) Environment type: 'rural', 'suburban', or 'urban'
%
% Output arguments:
%  LLMimoNodes  (1xN Node obj array) Newly-created user nodes
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (c) 2006-2016 Massachusetts Institute of Technology %
% All rights reserved.   See software license below.            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define transmit/receive modules

tx_mod_params.name = 'LLmimo_tx';
tx_mod_params.callbackFcn = @LLMimo_Transmit;
tx_mod_params.fc = 850e6;
tx_mod_params.type = 'transmitter';
tx_mod_params.loError = 0;
tx_mod_params.antType = {'dipole_halfWavelength'};
tx_mod_params.antPosition = [0 -5 0;
                              0 -4 0;
```

```
                             0 -3 0;
                             0 -2 0;
                             0 -1 0;
                             0 0 0;
                             0 1 0;
                             0 2 0;
                             0 3 0;
                             0 4 0]*3e8/tx_mod_params.fc;
tx_mod_params.antPolarization = {'v'};
tx_mod_params.antAzimuth = [0];
tx_mod_params.antElevation = [0];
tx_mod_params.exteriorWallMaterial = 'none';
tx_mod_params.distToExteriorWall = [0];
tx_mod_params.exteriorBldgAngle = [0];
tx_mod_params.numInteriorWalls = [0];


rx_mod_params.name = 'LLmimo_rx';
rx_mod_params.callbackFcn = @LLMimo_Receive;
rx_mod_params.TDCallbackFcn = @LLMimo_Rx_TDCallback;
rx_mod_params.fc = 850e6;
rx_mod_params.type = 'receiver';
rx_mod_params.loError = 0;
rx_mod_params.noiseFigure = 6; % (dB) noise figure of receiver
rx_mod_params.antType = {'dipole_halfWavelength'};
rx_mod_params.antPosition = [0 -4 0;
                             0 -2 0;
                             0 0 0;
                             0 2 0]*3e8/rx_mod_params.fc;
rx_mod_params.antPolarization = {'v'};
rx_mod_params.antAzimuth = [0];
rx_mod_params.antElevation = [0];
rx_mod_params.exteriorWallMaterial = 'none';
rx_mod_params.distToExteriorWall = [0];
rx_mod_params.exteriorBldgAngle = [0];
rx_mod_params.numInteriorWalls = [0];


tx_mod = module(tx_mod_params);
rx_mod = module(rx_mod_params);
```

Module objects are constructed in the normal fashion. However, since this example involves a MIMO transmitter and receiver, the antenna position, `antPosition` (line 11 and 38), is a 2-dimensional matrix rather than a single vector.[5] Also, `rx_mod_params.TDCallbackFcn` has been defined on line 32. The function that it points to, `LLMIMO_Rx_TDCallback` is a custom plot for the timing diagram callback.

```
   % Create genie modules
74 tx_gen.name = 'genie_tx';
   tx_gen.type = 'transmitter';
76
   rx_gen.name = 'genie_rx';
78 rx_gen.type = 'receiver';

80 gen_tx_mod = module(tx_gen,1);
   gen_rx_mod = module(rx_gen,1);
```

The section above creates the genie receive and transmit modules. Genie modules transfer information without actually sending an analog signal through the environment. Instead, data sent from one genie module "magically" appears at another. Unlike normal modules, no functions need to be written to define the modulation/demodulation schemes. Note that a genie transmit module can multi-cast a message to several genie receive modules.

The genie modules are defined by a name and a type (transmitter or receiver). To build a genie module, simply include an extra argument in the call to the module constructor (lines 60-61). This extra argument is the "genie flag". It instructs the constructor to make a genie module.

```
84 % Define nodes

86 DN_node_params.name = 'DN';
   switch lower(range)
88     case 'short'
           DN_node_params.location = [1.0e3 0 3];
90     case 'short-medium'
           DN_node_params.location = [2.5e3 0 3];
92     case 'medium'
           DN_node_params.location = [5.0e3 0 3];
94     case 'long'
           DN_node_params.location = [1.0e4 0 3];
96     otherwise
           error('Incorrect range case')
98 end
   DN_node_params.velocity = [0 0 0];
```

[5]Please refer to the reference section for more details.

```
DN_node_params.controllerFcn = @LLMimo_DN_Controller;
DN_node_params.modules = [tx_mod gen_rx_mod];

AN_node_params.name = 'AN';
switch lower(envType)
    case 'rural'
        AN_node_params.location = [0 0 100];
    case 'suburban'
        AN_node_params.location = [0 0 30];
    case 'urban'
        AN_node_params.location = [0 0 30];
    otherwise
        error('Incorrect environment type!')
end
AN_node_params.velocity = [0 0 0];
AN_node_params.controllerFcn = @LLMimo_AN_Controller;
AN_node_params.modules = [rx_mod gen_tx_mod];

DN_node = node(DN_node_params);
AN_node = node(AN_node_params);
```

The code block above builds the nodes. The genie module is included in the array of modules just like a normal module. The genie modules are used in the controller function similar to a normal module, but using different function calls. This is discussed in more detail later. The DN location is specified by the function input `range` and the AN location is specified by the function input `envType`.

User parameters are defined by populating a struct with field names and values just as before. In this example, however, one node only transmits, while the other only receives. As a result, some parameters, such as the training sequence, are shared, while other parameters, such as the transmit bits, are not.

```
% Define shared parameters (packet definition)
sharedParams.noiseLen = 100;
sharedParams.hTrainingLen = 100;
sharedParams.hTrainingSeq = rand(GetNumAnts(tx_mod),...
    sharedParams.hTrainingLen)>.5;
sharedParams.trainingLen = 100;
sharedParams.trainingSeq = rand(1,sharedParams.trainingLen)>.5;
sharedParams.infoLen = 512;
sharedParams.spreadRatio = 5;
sharedParams.nOversamp = 3;
sharedParams.blockLen = sharedParams.nOversamp*...
    (sharedParams.noiseLen+sharedParams.hTrainingLen+...
     sharedParams.trainingLen+sharedParams.spreadRatio...
```

```
        *sharedParams.infoLen);
136 sharedParams.nBlocksToSim = 8;
```

The shared parameters, shown above, include all the information that is required to synchronize between the two radios. These parameters define the packet structure and the training sequences. The number of packets to simulate is also defined here (line 92).

```
138 % DN-specific (Transmitter) parameters
    dnParams.transmittedBlocks = 0;
140 dnParams.txPower = 1; % (Watts)
    dnParams.targDomAtten = 2;
142 dnParams.infoSourceFilename = './user_code/examples/MIMONodes/
       testdata_long.dat';
    dnParams.infoSourceFID = [];
144 dnParams.infoBits = [];
    dnParams.txBitsFilename = '';
146 dnParams.txBitsFID = [];
    dnParams.getFromRx = [];
```

The transmitter (`DN`) in this example does not generate data randomly as in previous examples. Instead, it reads binary data from a file. This file is specified as a parameter named `infoSourceFilename` (line 131). When running, the controller function will read a block of data from file and put it into `infoBits`. `LLMimo_Transmit()` will then modulate this data for transmission. The field `infoSourceFID` will be used to hold a pointer to the `testdata_long.dat` once it is opened.

While it's possible to read the entire datafile and put the bits into `infoBits`, this is not suggested. Doing so will use a lot of computer memory and incur a large amount of overhead during function calls. The simulation will run slowly.[6]

`LLMimo_Transmit()` will save all transmitted bits into the file to be specified in the field `txBitsFilename`. This is used later for calculating the BER. In previous examples, bits were stored directly in the node. However, this example is sufficiently large that it is more computationally efficient to write the information to file for the same reasons explained above.

The field `getFromRx` is a struct. It is used to hold data received by the genie receive module. The information is passed from the genie receive module, to the controller, and then to the transmit function `LLMimo_Transmit()` in the transmit

[6]MATLAB passes a *copy* of the node object during function calls. There is no such thing as passing objects by reference.

module LLMimo_Tx. This process can be better understood by seeing it in the controller function in the following section.

```
% AN-specific (Receiver) parameters
anParams.receivedBlocks = 0;
anParams.epsilon = 10^-5;
anParams.lagRange = [-50:5];
anParams.rxBitsFilename = '';
anParams.rxBitsFID = [];
anParams.passToTx = [];
```

Parameters for the receiver are defined in the fragment of code shown above. epsilon and lagRange define parameters that are used by the STAP receiver in LLMimo_Receive(). The demodulated bits are stored into the file specified by rxBitsFilename.

The field passToTx is used to hold information being passed back to the DN. Data stored here by LLMimo_Receive() is copied into the genie transmit module by the AN controller function and sent to the DN genie receive module (or to others if multi-cast).

```
% Save parameters within nodes
dnParams = StructMerge(dnParams,sharedParams);
DN_node = SetUserParams(DN_node,dnParams);

anParams = StructMerge(anParams,sharedParams);
AN_node = SetUserParams(AN_node,anParams);


% Put user nodes into array
LLMimoNodes = [DN_node AN_node];
```

StructMerge() is a utility function that merges the fields of two different structs into one. The merged parameters are stored into the node objects and the node objects are returned as an array.

LLMimo_Transmit.m and LLMIMO_Receive.m

The transmit and receive function for the LL MIMO example implement a simple MIMO link. The transmit function uses a very simple repetition code, and the signal is BPSK encoded and resampled without pulse shaping. The receiver uses a STAP beamformer. In terms of programming, only one line is worth noting.

```
% Save received bits to file for comparison with received bits
[count,fPtr] = WriteBitBlock(p.rxBitsFID,demodBits);
```

The code above from `LLMimo_Receive.m` shows a call to the function `WriteBitBlock()`. This function saves the demodulated bits to file for calculation of the BER. There is a corresponding line of code in `LLMimo_Transmit.m` that writes transmitted bits to file. The files are initialized at startup by the controller function.

LLMimo_AN_Controller.m and LLMimo_DN_Controller.m

The structure of the controller function should be familiar, by now. However, since this is the first example that incorporates the use of the genie modules, it may be useful to examine the details.

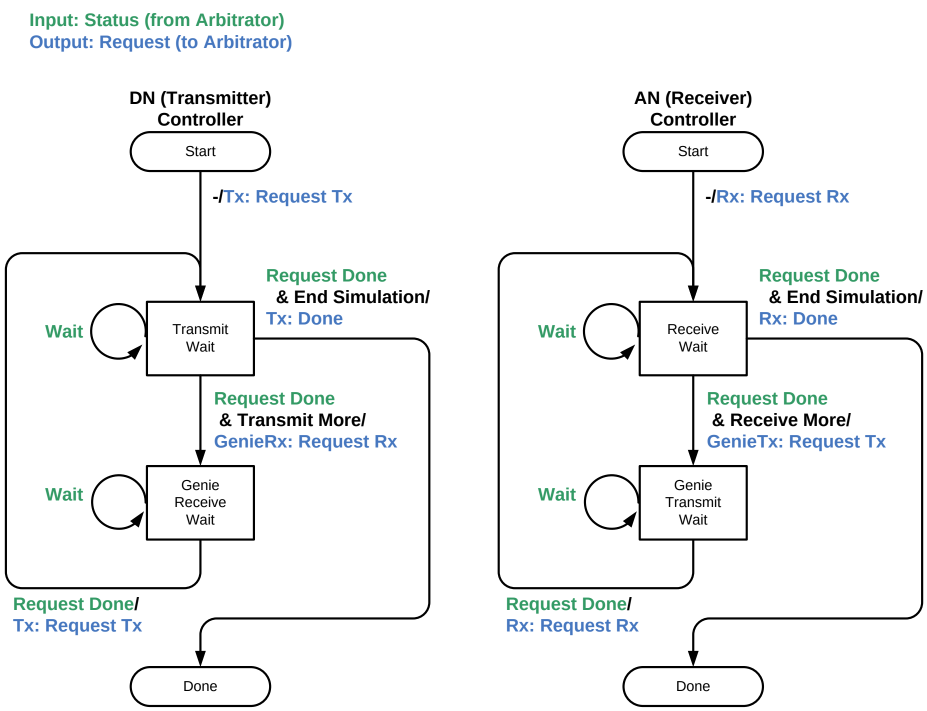


Figure 2.12: State diagrams for LL MIMO controllers

Figure 2.12 shows the state diagrams for the controller functions. Note that neither the LL MIMO transmit module nor the LL MIMO receive module are ever placed into a wait state. This is because the genie module is special. It transmits information in zero time. As a result, a separate state is required in the controller (as in the half-duplex example), but there is no need to instruct the transmit/receive modules to wait.

Requests for genie modules to transmit or receive are made using the functions `SetGenieTxRequest()` and `SetGenieRxRequest()`. Data is loaded into or read from the modules using `ReadGenieInfo()` and `WriteGenieInfo()`. The code fragment below from `LLMimo_AN_Controller.m` illustrates how the functions are used to initiate a genie transmission.

```
% Put send data into module's genie queue
nodeobj = WriteGenieInfo(nodeobj,'genie_tx',p.passToTx);

% Request send through genie channel to DN's genie receive
nodeobj = SetGenieTxRequest(nodeobj,'genie_tx','DN','genie_rx');
...
```

In additional to the node object and module name, `SetGenieTxRequest()` takes the destination node and module name. This is required because it is possible for a genie transmit module to send to any genie receive module in a scenario. To multi-cast to multiple genie modules, the `toNodeName` and `toModName` can be (equal sized) cell arrays. Data is delivered to the genie queue for each specified node/module address.

Similarly, the code snippets below from `LLMimo_DN_Controller.m` show the code required to receive from a genie channel.

```
...
% Transmission done, receive feedback info from genie channel
nodeobj = SetGenieRxRequest(nodeobj,'genie_rx');
```

```
...
requests = CheckRequestFlags(nodeobj);
if requests==0
    % Got feedback, copy received info into user params
    [nodeobj,p.getFromRx] = ReadGenieInfo(nodeobj,'genie_rx');
...
```

This example provided a very brief overview to how genie channels are used. It should be enough information to get started.

LLMimo_CalcBER.m

The LL MIMO example saves the transmitted and received bits to file rather than within the node object. This was done to conserve memory (at the expense of disk space). As a result, calculating the BER requires reading data from file. The code block below from `LLMimo_CalcBER.m` illustrates this process using the functions provided with LLAMAComm. `ReadBitBlock()` reads bits from files that were written by `WriteBitBlock()` (which is called from within `LLMimo_Transmit()` and `LLMimo_Receive()`).

```
function ber = LLMimo_CalcBER(nodes)

% Extract user parameters
pAN = GetUserParams(FindNode(nodes,'AN'));
pDN = GetUserParams(FindNode(nodes,'DN'));

% Open saved files for reading
ANfid = OpenBitFile(pAN.rxBitsFilename);
DNfid = OpenBitFile(pDN.txBitsFilename);

% Loop through blocks
ANfPtr = 0;
DNfPtr = 0;
errCount = 0;
bitCount = 0;
while(1)

    [ANbits,ANfPtr] = ReadBitBlock(ANfid,ANfPtr);
    [DNbits,DNfPtr] = ReadBitBlock(DNfid,DNfPtr);

    if isempty(ANbits) && isempty(DNbits)
        % Done with files
        break;

    elseif ~isempty(ANbits) && ~isempty(DNbits)
        % Compare data
        errs = find(xor(ANbits,DNbits));

        % Count errors and bits
        errCount = errCount+length(errs);
        bitCount = bitCount+size(ANbits,2);

    else
        error('Bit files have unequal length!');
    end
end

% Calculate BER
ber = errCount/bitCount;

% Print results to screen
```

```
fprintf('LL MIMO example, DN->AN BER: %f\n',ber);

% Close opened files
fclose(ANfid);
fclose(DNfid);
```

2.3.5 The Example Environment

The example simulation is run using an example environment created in `user_code/examples/Ex_BuildEnvironment.m`. The code sets some properties and builds an environment object. Details about the environment properties can be found in Chapter 4.

```
function env = Ex_BuildEnvironment

% Function Ex_BuildEnvironment.m:
% Builds an example rural environment object.
%
% USAGE: env = Ex_BuildEnvironment
%
% Input args:
%  -none-
%
% Output arguments:
%  env    (environment object) New environment object
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (c) 2006-2016 Massachusetts Institute of Technology %
% All rights reserved.   See software license below.            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

test_env_params.envType = 'rural';
test_env_params.propParams.delaySpread = .2e-6;  % sec
test_env_params.propParams.velocitySpread = 0.1;  % m/s
test_env_params.propParams.alpha = 0.5;
test_env_params.propParams.longestCoherBlock = 1;
test_env_params.propParams.stfcsChannelOversamp = 3;
test_env_params.propParams.wssusChannelTapSpacing = []; % samples
test_env_params.propParams.los_dist = 10;             % m
test_env_params.building.avgRoofHeight = 4;
env = environment(test_env_params);
```

2.3.6 The Start Script

The start script is responsible for setting up the required paths, calling the proper build files, and starting the simulation by calling the `Main()` function. The contents of the `StartExample.m` script are shown below.

```
% Script StartExample.m:
% The start script sets up the MATLAB path, calls user functions to set up
% the simulation universe, and starts the simulation.  When the simulation
% is complete, user-defined functions can be called to analyze the
% results.
%
% This script runs the example described in the documentation:
%    Easy rural environment
%    2 Half-duplex nodes transmitting BPSK in the FM band
%    2 Full-duplex nodes transmitting BPSK in GlobalStar/ISM
%    1 Complex white gaussian noise interferer in the ISM band
%    1 complex colored gaussian noise interferer in the FM band
%    2 LL Mimo nodes (forward link with genie reverse link)
%
```

```
% Add required directories containing simulator functions
% to the MATLAB search path
SetupPaths;

% Paths to user-defined functions
addpath ./examples
addpath ./examples/BPSKNodes
addpath ./examples/MIMONodes
addpath ./examples/InterferenceNodes

% Clear old variables
%clear all;

% Initialize global variables
InitGlobals;

% Start timer to measure simulation time
tic

% Build example environment
env = Ex_BuildEnvironment;
```

```
42
   % Populate simulation universe with nodes
44 nodes = HD_BuildNodes;
   nodes = [nodes FD_BuildNodes];
46 nodes = [nodes Ex_BuildInterference];
   nodes = [nodes LLMimo_BuildNodes('short','rural')];
48
   % Make a map of the nodes
50 MakeNodeMap(nodes,1);

52 % Start simulation
   [nodes,env,success] = Main(nodes,env);
54
   % Analyze results
56 if success
       HD_BER = HD_CalculateBER(nodes);
58     FD_BER = FD_CalculateBER(nodes);
       LL_BER = LLMimo_CalcBER(nodes);
60 end

62 % Stop timer
   toc
64
   % Save workspace variables and timing diagram figure
66 save(fullfile(saveDir,'Workspace'));
   if timingDiagramFig
68     saveas(timingDiagramFig,fullfile(saveDir,'TimingDiagram'),'fig');
   end
```

The start script is also responsible for saving all the workspace variables and the timing diagram figure. We suggest that this file be used as a template. Make a copy of this file and modify lines 36-40 and 50-52 as needed.

2.3.7 The Global Variables File

The last programming file that will be discussed as part of the examples is `InitGlobals.m`. This file holds parameters that are shared across the entire simulation. This file should not be modified. Instead, you should copy it and name it something like `InitGlobals_myname.m`. **User parameters should not be added to this file! Variables defined here should never be written to while the simulation is running.** For a listing and description of the global variables, see Section 4.1.

2.4 Debugging Tips

The previous sections covered the implementation of the radios in the example scenario. Most of these examples were developed after the core of the simulator had been completed. During development, it was useful to be able to view the properties and variables within the objects that are normally hidden. As a result, some useful utilities were developed to aid in debugging.

2.4.1 Timing Diagram

The timing diagram was the main tool developed for debugging the controller functions and program execution order. If the simulation stalls, looking at where the timing diagram ends should give clues into which module has misbehaved. Signal information can be viewed by clicking on the segment rectangles on the diagram.

A powerful way to use the timing diagram for debugging the state machine code in the controller function is to color the blocks according to the current state of the state machine. LLAMAComm checks to see if the field `faceColor` is present in the user parameters with subfields corresponding to the module names and if so, colors the block accordingly. To gain insight into this feature, add the following example code to the end of the full-duplex controller function `HD_UserA_Controller.m` before the user parameters are updated:

```
% Update the timing diagram color for half-duplex user A
switch nextState
    case 'start'
        colorRGB = [0 0 1]; % Blue
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'transmit_wait'
        colorRGB = [1 0 0]; % Red
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'receive_wait'
        colorRGB = [0 1 0]; % Green
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'done'
        colorRGB = [1 0 1]; % Magenta
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
end
```

```
% Store possibly modified user params
nodeobj = SetUserParams(nodeobj,p);
```

Also add the following code the end of the full-duplex controller function `HD_UserB_Controller.m` before the user parameters are updated:

```
% Update the timing diagram color for half-duplex user B
switch nextState
    case 'start'
        colorRGB = [0 0 1]; % Blue
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'transmit_wait'
        colorRGB = [0 1 0]; % Green
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'receive_wait'
        colorRGB = [1 0 0]; % Red
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
    case 'done'
        colorRGB = [1 0 1]; % Magenta
        p.faceColor.hd_tx = colorRGB;
        p.faceColor.hd_rx = colorRGB;
end

% Store possibly modified user params
nodeobj = SetUserParams(nodeobj,p);
```

You should now see the timing diagram blocks for the half-duplex users colored according to the next state of the state machine.

2.4.2 Separating the Received Signals

A useful aid for testing algorithms is to have access to individual noise-free received signals from all in-band transmit modules, rather than the superposition of all the received signals. For example, a receiver could be operated with and without an interference signal present in order to test an interference rejection algorithm. LLAMAComm provides this capability in the following way. During node construction, if a field in the user-parameters struct of a given node is named `separateTheReceivedSignals`, and it is set to 1, then during execution of each module's receiver callback function in that node, a struct called `sigSep` containing the separated signals and noise will be placed as a field in the user-parameters struct. Each separated signal is labeled by the node and module that produced it; the noise is labeled by `additiveNoise`.

2.4.3 Getting the Time-Varying Channel Impulse Response

There are two ways one may obtain the time-varying impulse response generated by LLAMAComm. The first way allows the user to obtain the channel impulse response during run time. This is described in the following paragraph.

If the user creates a field in the user parameters called `getChannelResponse` and sets it to 1, LLAMAComm will create a user parameter field called `chanResp` and populate it with the channel impulse in the middle of the block for each active link in the block. The channel information is also included and each response is labeled by the node and module that produced it. The channel impulse response is then available for evaluation when the module's receive callback function is called. The response has dimensions $[n_R, n_T, n_L]$, defined as the number of receivers, transmitter, and lags, respectively. If more samples of the impulse response are desired, the user can create a field in the user parameters called "channelResponseTimes" and set it to an array with elements in the range $[0, 1]$. The impulse response will be calculated at:

```
  sampTimes = blockStartSamp + round(channelResponseTimes*blockLen)
```

An error is thrown if any element of channelResponseTimes is not in the set $[0, 1]$.

After the simulation has finished, one may also generate the time-varying impulse response of a specified channel by executing the following code:

```
>> sampRate = 12.5e6; % (Hz) Simulation sample rate
>> linkNum = 1; % Choose one of the links to examine
>> % Sample the channel every millisecond for .1 seconds
>> time = (0:.001:.1); % (sec)
>> hTime = GetChannelResponse(env,linkNum,time,sampRate);
>> % Plot the 1st tap of the channel between the 1st Tx and the 1st Rx
>> plot(time, squeeze(abs(hTime(1,1,1,:))))
```

2.4.4 Information in the Workspace

After the simulation has finished running, there is also a wealth of information in the workspace. Typing `env` will list the environment properties along with a numbered list of the link objects created during the simulation.

```
>> env
```

```
env =

            envType: rural
          propParams.
           delaySpread: 2e-07 s
        velocitySpread: 0.1 m/s
                 alpha: 0.5
     longestCoherBlock: 1 s
  stfcsChannelOversamp: 3
wssusChannelTapSpacing:  samples
              los_dist: 10 m
            building.
         avgRoofHeight: 4 m

           shadow.
     nodeArray: [1x8 struct]
    linkMatrix: [8x8 logical]
          Krho: [25x25 double]
      corrLoss: [25x1 double]
     linkNames: {1x25 cell}

Link #         From (node:module) -> To (node:module:fc)

   1.              HD_UserA:hd_tx -> HD_UserB:hd_rx:98.500 MHz
   2.           Interf_FM:ccgn_tx -> HD_UserB:hd_rx:98.500 MHz
   3.              HD_UserB:hd_tx -> HD_UserA:hd_rx:98.500 MHz
   4.           Interf_FM:ccgn_tx -> HD_UserA:hd_rx:98.500 MHz
   5.          Interf_ISM:cwgn_tx -> FD_UserB:fd1_rx:2495.000 MHz
   6.                DN:LLmimo_tx -> AN:LLmimo_rx:850.000 MHz
   7.             FD_UserB:fd2_tx -> FD_UserA:fd2_rx:1500.000 MHz
   8.             FD_UserA:fd1_tx -> FD_UserB:fd1_rx:2495.000 MHz
```

One can view information about link number 1 by typing the following:

```
>> DisplayLinkParams(env,1)

   'HD_UserA:hd_tx' -> 'HD_UserB:hd_rx:98.50 Mhz'

        channel.
                    chan: [1x3 double]
              chanTensor: [4-D double]
                fakeHpow: 0.9349
              nDelaySamp: 3
          nPropDelaySamp: 0
```

```
      longestCoherBlock: 1
        dopplerSpreadHz: 0.0329
           nDopplerSamp: 1
              hOverSamp: 3
   stfcsChannelOversamp: 3
           ricePhaseRad: 3.6380
               freqOffs: 0
                phiOffs: 3.4765
               chanType: 'stfcs'


      pathLoss.
    shadowLinkIndex: 1
          rangeLoss: 67.5098
     shadowCorrLoss: -0.0883
          shadowStd: 5.3631
         shadowLoss: -0.4738
      externalNoise: 11.9818
        noiseFigure: 6
          antGainTx: 0.6503
          antGainRx: 0.6503
      totalPathLoss: 65.7355
           riceKdB: 0.9450
         riceMedKdB: 4.7728
   distBetweenNodes: 101.9853


    propParams.
              delaySpread: 2.0000e-07
           velocitySpread: 0.1000
                    alpha: 0.5000
        longestCoherBlock: 1
     stfcsChannelOversamp: 3
   wssusChannelTapSpacing: []
                 los_dist: 10
            linkParamFile: []
                 chanType: 'stfcs'
```

Similarly, information about nodes and modules can be accessed by using some provided functions. The array of all nodes in the simulation is available from the MATLAB workspace. Basic information about a single node can be displayed by using the node object's built-in display function.

```
>> nodes(1)
```

```
ans =

          name: HD_UserA
      location: [0 20 3]
      velocity: [0 0 0]
 controllerFcn: @HD_UserA_Controller
         state: 'done'
       modules: 'hd_tx' 'hd_rx'

 User Parameters:
                power: 10
              dataLen: 800
          trainingLen: 200
            nOversamp: 2
               bitLen: 1000
             blockLen: 2000
          trainingSeq: [1x200 logical]
         nBlocksToSim: 15
       receivedBlocks: 15
    transmittedBlocks: 15
               txBits: [15x1000 uint8]
               rxBits: [15x1000 uint8]
         equalizerLen: 21
       equalizerDelay: 10
```

FindNode() can be used to find a node by name, and DisplayModule() can be used to bring up detailed information about the module. The example below shows the contents of the module object after the example simulation was run. The history section is a record of every state the module has been in. It is useful for debugging the controller function.

```
>> hd = FindNode(nodes,'HD_UserA');
>> DisplayModule(hd,'hd_tx')

modobj =

            name: hd_tx
              fc: 98500000 Hz
              fs: 12500000 Hz
            type: transmitter
     callbackFcn: @BPSK_Transmit
         loError: 0 parts
    loCorrection: 0 parts
         antType: dipole_halfWavelength
     antPosition:     0.00     0.00     0.00
```

```
antPolarization: v
     antAzimuth:     0.00
   antElevation:     0.00
                     0.00
   exteriorWallMaterial: none
   distToExteriorWall:      0.00
   exteriorBldgAngle:     0.00

               Request
            job: 'done'
    requestFlag: 0
    blockLength:
     blockStart: 60001

               History
   Start    Length         Job      fc (MHz)      fs (MHz)       fPtr
       1      2000    transmit     98.500000     12.500000          0
    2001      2000        wait     98.500000     12.500000         -1
    4001      2000    transmit     98.500000     12.500000      16044
    6001      2000        wait     98.500000     12.500000         -1
    8001      2000    transmit     98.500000     12.500000      32088
   10001      2000        wait     98.500000     12.500000         -1
   12001      2000    transmit     98.500000     12.500000      48132
   14001      2000        wait     98.500000     12.500000         -1
   16001      2000    transmit     98.500000     12.500000      64176
   18001      2000        wait     98.500000     12.500000         -1
   20001      2000    transmit     98.500000     12.500000      80220
   22001      2000        wait     98.500000     12.500000         -1
   24001      2000    transmit     98.500000     12.500000      96264
   26001      2000        wait     98.500000     12.500000         -1
   28001      2000    transmit     98.500000     12.500000     112308
   30001      2000        wait     98.500000     12.500000         -1
   32001      2000    transmit     98.500000     12.500000     128352
   34001      2000        wait     98.500000     12.500000         -1
   36001      2000    transmit     98.500000     12.500000     144396
   38001      2000        wait     98.500000     12.500000         -1
   40001      2000    transmit     98.500000     12.500000     160440
   42001      2000        wait     98.500000     12.500000         -1
   44001      2000    transmit     98.500000     12.500000     176484
   46001      2000        wait     98.500000     12.500000         -1
   48001      2000    transmit     98.500000     12.500000     192528
   50001      2000        wait     98.500000     12.500000         -1
   52001      2000    transmit     98.500000     12.500000     208572
   54001      2000        wait     98.500000     12.500000         -1
   56001      2000    transmit     98.500000     12.500000     224616
```

```
    58001      2000        wait      98.500000      12.500000          -1


                Save file info
        filename: './save/20120323T164527/HD_UserA-hd_tx.sig'
            fid:


>>
```

2.4.5 Other Debugging Tips

Finally, conventional debugging techniques such as setting breakpoints and printing debugging output are often very effective.

2.5 Example: Building a Receive-only Node

After looking over the example LLAMAComm code, you might not know where to begin building your own code. In this section, we walk you through the process of modifying the example code to build a simple receiver-only node. This experience will help you design more complicated nodes and simulations. Please make the changes exactly as given so the line numbers can be referred to correctly. In many cases you can copy and paste directly from the electronic version of this document.

To begin, create the directory `user_code\observer` and copy in the following files from `user_code\examples\InterferenceNodes`: `CWGN_Controller.m`, and `Ex_BuildInterference.m`. In addition copy in the file `BPSK_Receive.m` from `user_code\examples\BPSKNodes` directory.

In the `observer` subdirectory, change the file names from `BPSK_Receive.m`, `CWGN_Controller.m`, and `Ex_BuildInterference.m` to `Observer_Receive.m`, `Observer_Controller.m`, and `Build_Observer.m`, respectively.

2.5.1 Receive Callback Function

The receiver you are building is very simple. It measures the power in the received signal `sig` and passes this measurement, via the user-parameters struct, to the controller callback function. The waveform amplitude unit in LLAMAComm is Volts, so the average power (across a 1-Ohm resistor) is the mean square of a waveform (see Section 3.2.4 for more information on measuring power in LLAMAComm). The controller callback function is the state machine that determines the functionality of the node. It will be modified in Section 2.5.2. For now, we'll concentrate on modifying the receiver callback function code. Open `Observer_Receive.m` in the MATLAB editor. Do the following:

1. Search on `BPSK_Receive` and replace with `Observer_Receive` (you should make three replacements).

2. Change the first six lines of the comments section from

```
% Function Observer_Receive.m:
% This is an example of a user-written callback function for a
% receive module.  This example demodulates an analog, BPSK signal.
%
% This example is designed to work with BPSK_Transmit
%
```

to the following:

```
% Function Observer_Receive.m:
% This is an example of a user-written callback function for a
% receive module.
%
```

Now that you've updated the header comments, you can start modifying the function's code.

The user-parameters struct is used to pass information between the controller callback function and receiver or transmitter callback functions. Notice on line 26 that the user-parameters struct is obtained; on line 48, the user-parameters struct is updated—the user parameters are initialized when the node is built (see Section 2.5.3).

To implement the receiver functionality, do the following:

3. Replace lines 28-50 with the following:

```
28 % Compute the block received power (Watts)
29 recPow = sum(abs(sig(:)).^2)/length(sig(:));
30 
31 % Get the current block number (updated in the controller function)
32 n = p.observedBlocks;
33 
34 % Update the received power array in the user parameters
35 p.recPow(n) = recPow;
36 
```

```
% Make a time plot of the received power (dB Watts) in each block
figure(2), plot(10*log10(p.recPow),'b*-')
xlabel('Observed Block Number')
ylabel('Observed Power (dBW)')
grid on
```

After you've completed the modifications, `Observer_Receive.m` should look like the following:

```
function nodeobj = Observer_Receive(nodeobj,modname,sig)

% Function Observer_Receive.m:
% This is an example of a user - written callback function for a
% receive module .
%
% USAGE: nodeobj = Observer_Receive(nodeobj,modname,sig)
%
% Input arguments:
%  nodeobj    (node obj) Parent node object
%  modname    (string) The name of the module that has activated this
%              callback function
%  sig        (NxblockLen) Analog baseband signal for N channels
%
% Output arguments:
%  nodeobj    (node obj) Modified copy of the node object
%
% See also: BPSK_Transmit.m

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (c) 2006-2016 Massachusetts Institute of Technology %
% All rights reserved.   See software license below.            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Load user parameters
p = GetUserParams(nodeobj);

% Compute the block received power ( Watts )
recPow = sum (abs (sig (:)).^2)/ length (sig (:));

% Get the current block number ( updated in the controller function )
n = p. observedBlocks ;

% Update the received power array in the user parameters
p. recPow (n) = recPow ;

```

```
   % Make a time plot of the received power (dB Watts ) in each block
38 figure (2) , plot (10* log10 (p. recPow ),'b*-')
   xlabel ('Observed Block Number')
40 ylabel ('Observed Power (dBW)')
   grid on
42
   % Save user params
44 nodeobj = SetUserParams(nodeobj ,p);
```

Notice on line 32 that we've assumed there is a field in the user-parameters struct called `observedBlocks`. It counts the number of observed blocks and is incremented by the controller callback function (see Section 2.5.2).

You're now done with `Observer_Receive.m`. Please save the file and congratulate yourself. Next you will modify the controller callback function.

2.5.2 Controller Callback Function

The controller callback function is a state machine and in general can be made as complex as you want. Depending on the node's current state and the status of the node's modules, the controller callback function sends transmit or receive requests for the node's modules to the LLAMAComm arbitrator. The LLAMAComm arbitrator calls the transmit or receive callback functions[7] to In our case, we just want to receive blocks until a specified number has been reached.

Open `Observer_Controller.m` (remember that you created this file according to the instructions at the beginning of Section 2.5). Make the following replacements:

1. Search on `CWGN_Controller` and replace with `Observer_Controller` (you should make three replacements).

2. Change the first four lines of the comments section from

```
% Function user_code/Observer_Controller.m:
% Controller state machine for an example interference source that
% transmits complex white gaussian noise in a specified band.
%
```

to the following:

```
% Function user_code/Observer_Controller.m:
% Controller state machine for a receiver that computes and
```

[7]The callback function name is specified as a `module` property (c.f. Section 4.3.1).

```
% plots the received power in a specified band.
%
```

You're now ready to begin modifying the code.

As you saw in the receiver callback function, the user-parameters struct is obtained at the beginning (on line 26) and updated at the end (on line 67). This is how the controller interacts with transmit or receive module callback functions.

Before diving headlong into changing the example code, let's make sure we understand what is already there. Below is a listing of lines 23-67 of the example code before you modify it:

```
   status = 'running';  % Default status
24
   % Load user parameters
26 p = GetUserParams(nodeobj);

28 % Get current node state
   currState = GetNodeState(nodeobj);
30
   % State machine next-state and output "logic"
32 switch currState
       case 'start'
34         % Set up module for transmission of 1 block
           nodeobj = SetModuleRequest(nodeobj,'cwgn_tx','transmit',p.
               blockLen);
36         nextState = 'transmit_wait';

38     case 'transmit_wait'
           requests = CheckRequestFlags(nodeobj);
40         if requests==0
               if p.transmittedBlocks>=p.nBlocksToTx;
42                 % Goto done
                   nextState = 'done';
44             else
                   % Transmission done, transmit again
46                 nodeobj = SetModuleRequest(nodeobj,'cwgn_tx','transmit'
                       ,p.blockLen);
                   nextState = 'transmit_wait';
48             end
           else
50             % All requests not satisfied, wait
               nextState = 'transmit_wait';
52         end
```

```
54        case 'done'
              % Done!
56            nextState = 'done';
              status = 'done';
58
          otherwise
60            error(sprintf('Unknown state: %s',currState));
      end
62
      % Set next state
64    nodeobj = SetNodeState(nodeobj,nextState);

66    % Store possibly modified user params
      nodeobj = SetUserParams(nodeobj,p);
```

The node’s current state is returned by `GetNodeState()` on line 29 and used in the switch statement on line 32. Notice there are three states: ‘`start`’, ‘`transmit_wait`’, and ‘`done`.’

In the ‘`start`’ state[8], the controller requests the module `cwgn_tx` to transmit a block of length `p.blockLen`. The node’s next state is set to ‘`transmit_wait`’ and on line 64, the node state is updated. The next time the controller callback function is called, the state machine enters the ‘`transmit_wait`’ state.

In the ‘`transmit_wait`’ state, the controller callback function checks if the `cwgn_tx` module’s `transmit` job has completed by seeing if the request flag has cleared. The module request flag[9] is obtained by calling `CheckRequestFlags()`. If the transmit job has completed, i.e., `if requests == 0`, then we check if the specified number of blocks have been transmitted, i.e., `if p.transmittedBlocks>=p.nBlocksToTx`—if the specified number is reached, the controller puts the node in the ‘`done`’ state; if not, the controller requests the module `cwgn_tx` to transmit another block of length `p.blockLen` and keeps the node in the ‘`transmit_wait`’ state. If the module request flag has not cleared, i.e., the statement `if requests == 0` is false, then the controller jumps to the `else` statement on line 44 and keeps the node in the ‘`transmit_wait`’ state (this is the “wait” part of the ‘`transmit_wait`’ state).

In the ‘`done`’ state, the controller callback function perpetually keeps the node in the ‘`done`’ state and sets the status variable to `done`. Typically, the module would be given a `done` job to signal to the LLAMAComm arbitrator that the module has no more

[8] The ‘`start`’ state is returned by `GetNodeState()` when the controller callback function is called for the first time.

[9] Note that in general, `CheckRequestFlags()` would return an array of request flags—one for each module in the node.

samples to transmit (see Section 2.3.3 for more details on why this is not done here).

By now, you should anticipate the following obvious changes to the example code:

3. Search on ‘`transmit_wait`’ and replace with ‘`receive_wait`’ (there should be four replacements).

4. Change the comments in lines 34 and 45 to reflect the new functionality (e.g., change *transmission* to *reception* and change *transmit* to *receive*).

5. Change the name of the module from `cwgn_tx` to `observer_rx` in lines 35 and 46.

6. Change the `'transmit'` job request to a `'receive'` job request in lines 35 and 46.

7. In line 41, change `p.transmittedBlocks` to `p.observedBlocks` and change `p.nBlocksToTx` to `p.nBlocksToObserve`.

To finish up, you have to add a few lines of code. You must increment the `observedBlocks` counter and send a `done` job to the receive module. Please add the following:

8. After the `SetModuleRequest()` function calls (there are two of them), increment the `observedBlocks` counter by inserting the following line:

   ```
   p.observedBlocks = p.observedBlocks + 1;
   ```

9. After the IF statement: if `p.observedBlocks>=p.nBlocksToObserve`, send a `done` job to the module by inserting the following line:

   ```
   nodeobj = SetModuleRequest(nodeobj,'observer_rx','done');
   ```

After making all the changes, lines 31 through 64 of `Observer_Controller.m` should read:

```
% State machine next-state and output "logic"
32 switch currState
    case 'start'
34      % Set up module for reception of 1 block
        nodeobj = SetModuleRequest(nodeobj,'observer_rx','receive',p.
            blockLen);
36      p.observedBlocks = p.observedBlocks + 1;
        nextState = 'receive_wait';
```

```

    case 'receive_wait'
        requests = CheckRequestFlags(nodeobj);
        if requests==0
            if p.observedBlocks>=p.nBlocksToObserve;
                nodeobj = SetModuleRequest(nodeobj,'observer_rx','done');
                % Goto done
                nextState = 'done';
            else
                % Reception done, receive again
                nodeobj = SetModuleRequest(nodeobj,'observer_rx','receive',p.blockLen);
                p.observedBlocks = p.observedBlocks + 1;
                nextState = 'receive_wait';
            end
        else
            % All requests not satisfied, wait
            nextState = 'receive_wait';
        end

    case 'done'
        % Done!
        nextState = 'done';
        status = 'done';

    otherwise
        error(sprintf('Unknown state: %s',currState));
end
```

Please save the changes you have made.

Now that you have a node controller callback function and a receive module callback function you are ready to build the observer node.

2.5.3 Building the Observer Node

The user-defined build-node function returns one or more nodes to be simulated in the LLAMAComm environment.

Open `Build_Observer.m` (remember that you created this file according to the instructions at the beginning of Section 2.5) and begin with the following

1. Search on `Ex_BuildInterference` and replace with `Build_Observer` (you should make three replacements).

2. Search on `interfNodes` and replace with `observerNode` (you should make five replacements).

3. Change the 14 lines of the header comments from

```
% Function Build_Observer.m:
% Example function for building interference nodes.
% This function returns 2 transmit-only nodes:
%  1. Complex colored gaussian noise in the FM band
%  2. Complex white gaussian noise in the ISM band
%
% USAGE: interfNodes = Build_Observer
%
% Input arguments:
%  -none-
%
% Output arguments:
%  interfNodes  (1xN Node obj array) Newly-created interference
   nodes
%
```

to the following 11 lines:

```
% Function Build_Observer.m:
% This function returns a receive-only observer node
%
% USAGE: observerNode = Build_Observer
%
% Input arguments:
%  -none-
%
% Output arguments:
%  observerNode  (1xN Node obj array) Newly-created observer node
%
```

You're now ready to begin modifying the rest of the code. The changes are numbered below:

4. You are building only one node, so delete all the lines from 56 to the end of the function.

5. Search on `fm_` and replace with `observer_` (you should make thirty replacements).

6. Change the comment on line 21 to read: `% Build observer node`.

7. Change the name of the module on line 23 from `'ccgn_tx'` to `'observer_rx'` (in general, you can make the name whatever you want as long as no other module in the node shares the same name).

8. Change the name of the module callback function on line 24 from `@CCGN_Transmit` to `@Observer_Receive`.

9. Change the module type on line 26 from `'transmitter'` to `'receiver'`.

10. Change the name of the node on line 40 from `'Interf_FM'` to `Observer_1` (in general, a node can have any unique name).

11. Change the node location on line 41 from `[0 -400 30]` to `[10 -350 50]`. Nodes cannot share the same location.

12. Change the name of the controller callback function on line 43 from `@CCGN_Controller` to `@Observer_Controller`.

13. Change the struct field name on line 48 from `.transmittedBlocks` to `.observedBlocks`.

14. Change the struct field name on line 49 from `nBlocksToTx` to `nBlocksToObserve`.

15. On line 49, reduce the number of blocks to observe from 21 to 15.

16. After line 36, add the following receiver module property:

```
observer_mod_params.noiseFigure = 4; % (dB)
```

Lines 21 to the end should be as follows:

```
   % Build observer node
22
   observer_mod_params.name = 'observer_rx';
24 observer_mod_params.callbackFcn = @Observer_Receive;
   observer_mod_params.fc = 99.5e6;
26 observer_mod_params.type = 'receiver';
   observer_mod_params.loError = 0;
28 observer_mod_params.antType = {'dipole_halfWavelength'};
   observer_mod_params.antPosition = [0 0 0];
30 observer_mod_params.antPolarization = {'v'};
   observer_mod_params.antAzimuth = [0];
32 observer_mod_params.antElevation = [0];
```

```
observer_mod_params.exteriorWallMaterial = 'none';
observer_mod_params.distToExteriorWall = [0];
observer_mod_params.exteriorBldgAngle = [0];
observer_mod_params.numInteriorWalls = [0];
observer_mod_params.noiseFigure = 4; % (dB)

observer_interf_mod = module(observer_mod_params);

observer_node_params.name = 'Observer_1';
observer_node_params.location = [10 -350 50];
observer_node_params.velocity = [0 0 0];
observer_node_params.controllerFcn = @Observer_Controller;
observer_node_params.modules = [observer_interf_mod];

observer_interf_node = node(observer_node_params);

observer_user_params.observedBlocks = 0;
observer_user_params.nBlocksToObserve = 15;
observer_user_params.blockLen = 3377;
observer_user_params.power = 1; % (Watts) transmit power

observerNode = SetUserParams(observer_interf_node,observer_user_params);
```

After making the above modifications and saving your changes, you are ready for the last step: including the observer node in the example simulation.

2.5.4 Simulating the Observer Node

New nodes are easily incorporated into existing LLAMAComm simulations. Open `StartExample.m` in the MATLAB editor and make the following modifications:

1. Insert the path to the observer node code by adding the following after line 29:

   ```
   addpath ./observer
   ```

2. Build the observer node and incorporate it into the node array by adding the following after line 48:

   ```
   nodes = [nodes Build_Observer];
   ```

Make sure you save the changes when you are finished.

Now it is time to see how well you followed the above instructions! Change the current MATLAB directory to `/llamacomm/user_code/examples` and run `StartExample.m`. If you don't have any bugs, you should see the received power displayed in a MATLAB figure and the observer node should appear at the bottom of the timing diagram. The observer node will also be present in the node map.

Chapter 3

Behind the Scenes

The tutorial walked through a number of examples that illustrate the structure of code written for LLAMAComm. This chapter fills in some of the details concerning how LLAMAComm works in the background. We will look at how the simulation is moderated by the arbitrator and the signal processing involved in combining signals from multiple sources. These details are pertinent for developing more advanced simulations and understanding the results.

3.1 Simulation Execution

The main program loop of LLAMAComm is shown in Figure 3.1. This loop represents the core of the simulator. There are two primary blocks: `Run Controllers` and `Run Arbitrator`. The first calls the node controller functions, while the second controls the simulation's execution order and calls the module transmit and receive functions. It is this second block—the arbitrator—that is truly responsible for coordinating the execution of the simulation.

3.1.1 Run Controllers

In the `Run Controllers` block, the controller function in each node is executed once. For digital designers, this is analogous to providing one clock edge to a finite state machine implemented in digital logic. The state is stored within the node object. When the controller function is *clocked*, it is given the opportunity to analyze its inputs, set some outputs, and change its state. Controller functions should be written so that the order in which they are called relative to other controllers should not matter. The

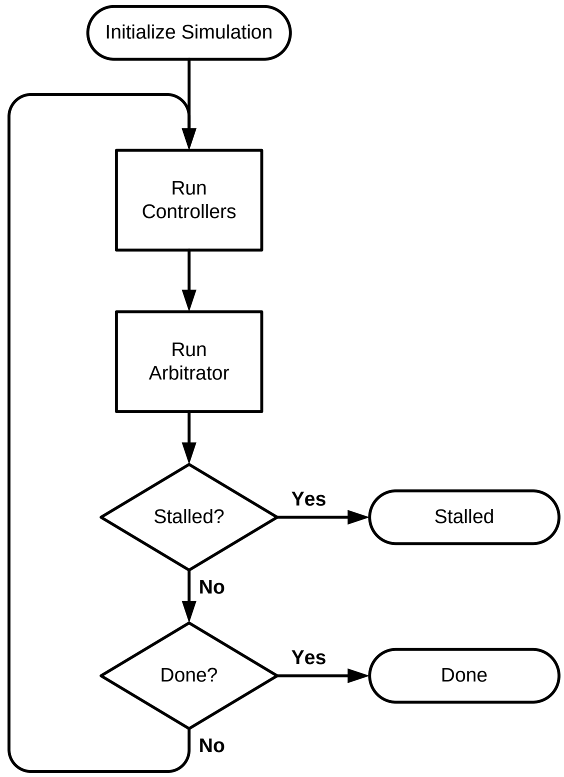


Figure 3.1: Flowchart of LLAMAComm's main program loop

tutorial in Chapter 2 covered the implementation of controller functions in great detail.

3.1.2 Run Arbitrator

In the `Run Arbitrator` block, the arbitrator looks at all the nodes in the scenario, and satisfies as many outstanding requests as possible. A simplified version of the arbitrator's flowchart is shown in Figure 3.2. The arbitrator sees only the modules in a scenario. In fact, it only looks at the module properties dealing with requests to transmit and receive.

The arbitrator flowchart (Figure 3.2) contains a block labeled `Query Modules` that requires special attention. What this block does is ask every other module in the scenario, "Do you have transmit data for samples x to y?" Modules can respond with one of three messages: *data available*, *not transmitting in-band*, or *not ready*. *Data available* means that the transmit data has been calculated and is ready to go. *Not*

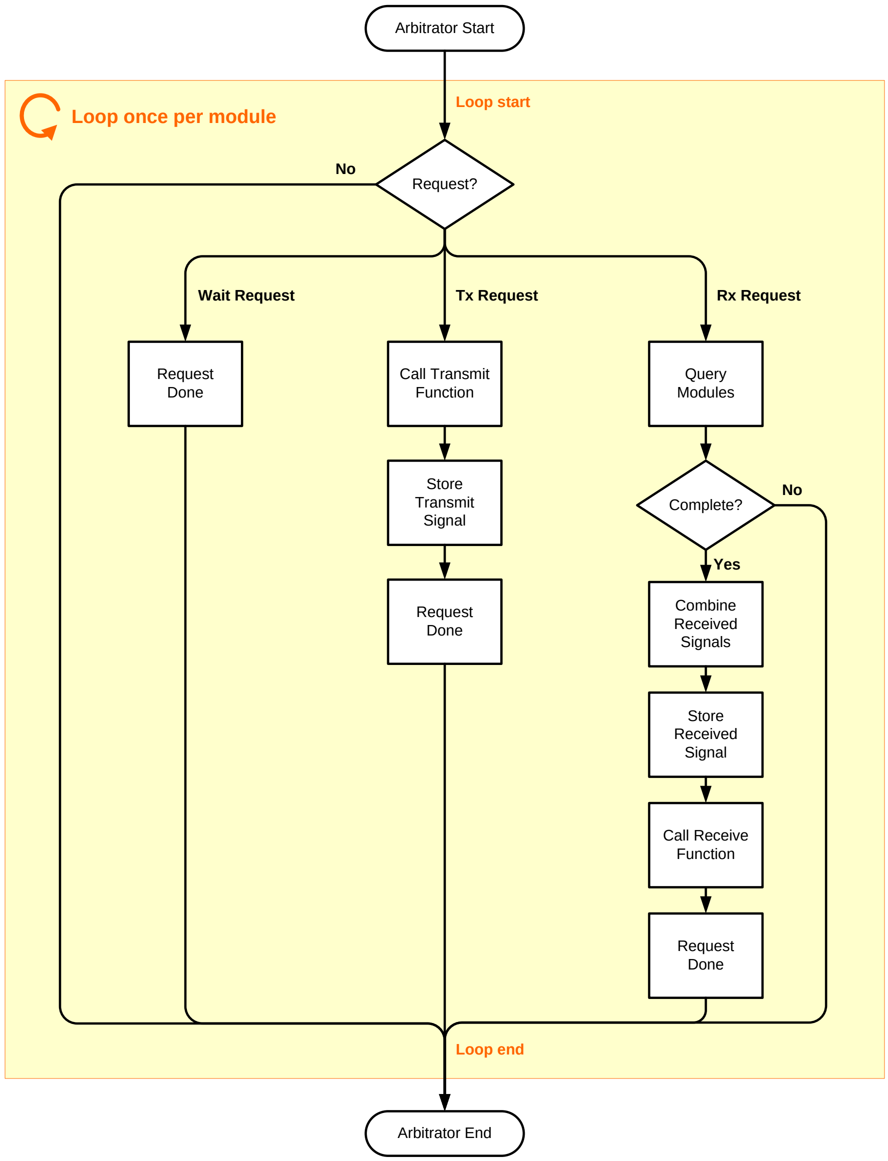


Figure 3.2: Flowchart of the arbitrator

transmitting in-band means that either the module is not a transmitter (never transmits), or is transmitting out of band during the requested segment. *Not ready* means that the module has not yet calculated the data for that segment and cannot reply with certainty. A query is considered complete if no module responds *not ready*.

This mechanism ensures that a receiver does not attempt to receive a segment of data until all transmitters in the requested time period have already transmitted.

This is also a good time to note that changing the center frequency of a module is allowed on a segment-by-segment basis. As a result, the arbitrator must also check for frequency overlap in each segment when querying the modules.

3.1.3 Stepping Through an Example Scenario

A good way to understand the interaction between the controllers and the arbitrator is by studying an example from the perspective of the arbitrator. Figures 3.3 to 3.5 illustrate this process by stepping through a very short scenario.

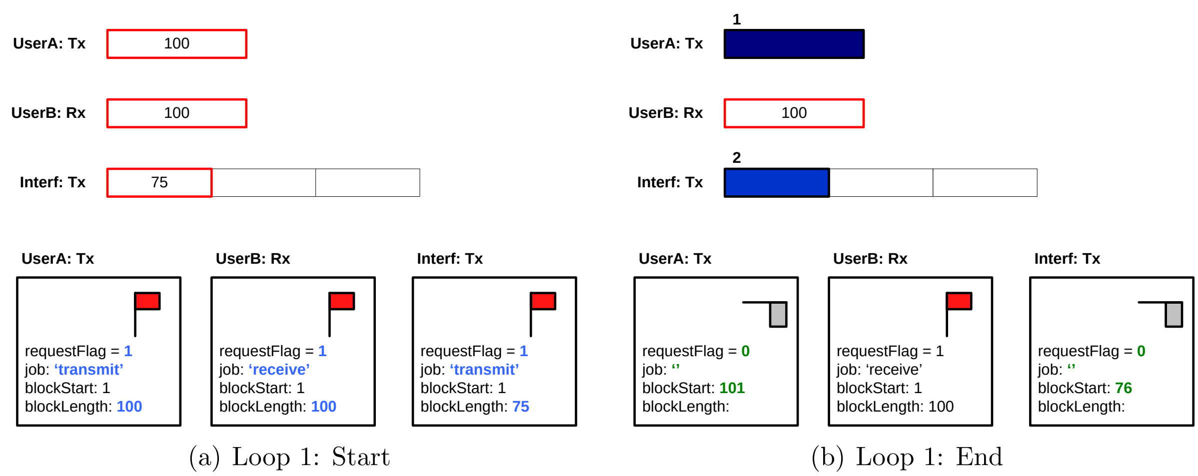


Figure 3.3: Example simulation at the start and end of the first pass of `RunArbitrator()`

Figure 3.3 shows an example of the arbitrator's view of the simulation before and after the first pass of `RunArbitrator()`. This example consists of three modules: `UserA:Tx`, `UserB:Rx`, and `Interf:Tx`. These modules are represented as boxes containing a request flag. The properties shown in each box are the module properties relevant to a request. The two user modules have controller functions that are written to transmit and receive a single 100-sample segment of signal. The interference module is configured to transmit three, 75-sample segments.

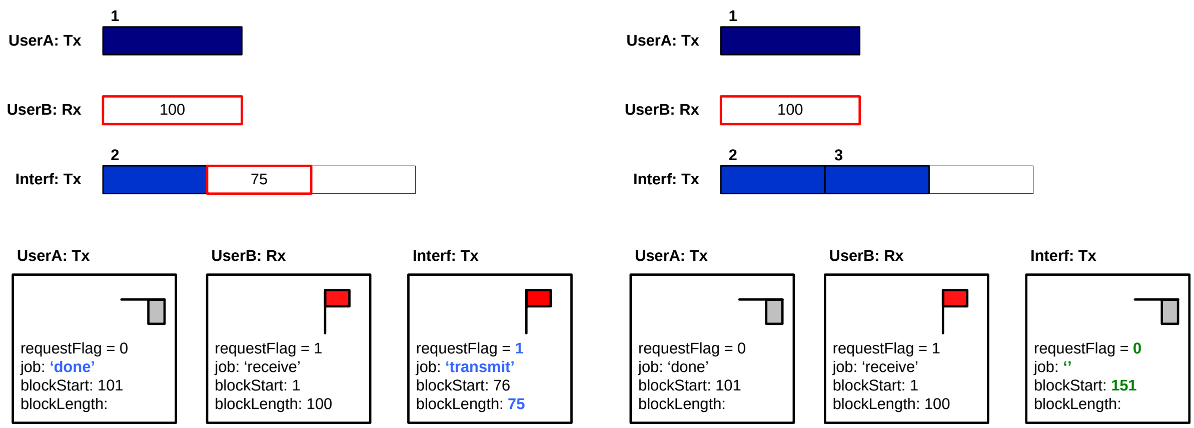
In Figure 3.3(a), the controller functions have set the properties shown in blue. The timing diagram above the picture of the modules, shows the progress of the simulation so far. At the start, all modules have an outstanding request to the arbitrator. The arbitrator attempts to process each of these modules in turn. The steps are listed below as a mock transcript of the arbitrator execution.

```
Process UserA:Tx
    Request outstanding
    Call transmit function
    Save transmit waveform to disk
    Request done
Process UserB:Rx
    Request outstanding
    Query results = not complete
    Skipping request
Process Interf:Tx
    Request outstanding
    Call transmit function
    Save transmit waveform to disk
    Request done
```

The arbitrator is operating left to right in this example. First, the request to transmit by `UserA:Tx` is processed. After calling the transmit function and writing the resulting baseband data to disk, the arbitrator lowers the request flag, deletes the outstanding job name, and increments the start index for the next block. The request to receive by `UserB:Rx` is then skipped because the required data is not yet available. `UserB:Rx` needs data from sample 1 to 100. During the first pass `UserA:Tx` has data ready, but `Interf:Tx` responds *not ready* to the query. After `UserB:Rx` is skipped, the request from `Interf:Tx` is satisfied.

Figure 3.4 shows the example at the beginning and end of the second pass of the arbitrator. `UserA:Tx`'s controller has tagged the module as "done", so from now on, `UserA:Tx` will be ignored by the arbitrator. The request from `UserB:Rx` is still outstanding and has not changed. The controller function for the interference module has issued another transmit request. The mock transcript for this iteration of the arbitrator follows.

```
Process UserA:Tx
    Skip (module marked as Done)
Process UserB:Rx
    Request outstanding
    Query results = not complete
```



(a) Loop 2: Start

(b) Loop 2: End

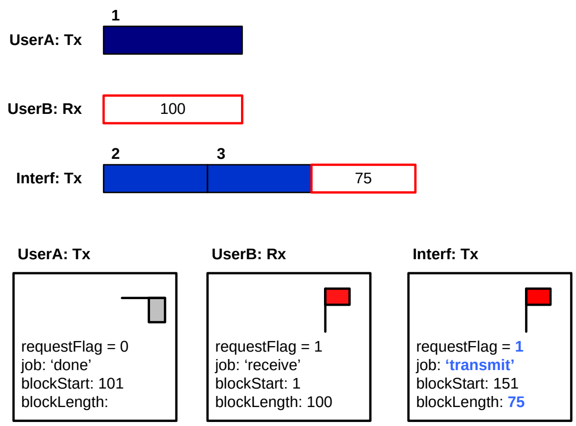
Figure 3.4: Example simulation at the start and end of the second pass of `RunArbitrator()`

```
        Skipping request
    Process Interf:Tx
        Request outstanding
        Call transmit function
        Save transmit waveform to disk
        Request done
```

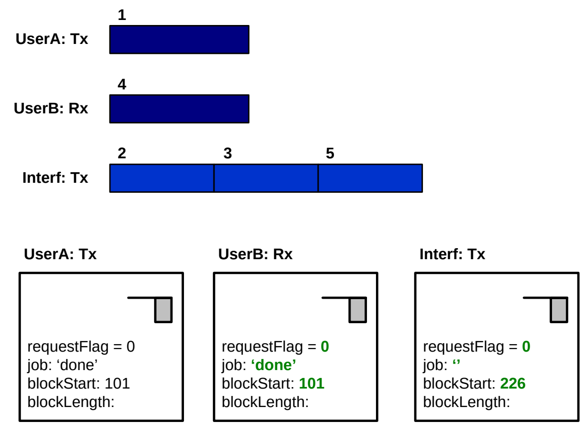
In this pass, `UserB:Rx` is again skipped because `Interf:Tx` has not yet transmitted and therefore responds *not ready*. `Interf:Tx` is able to transmit at the end of the iteration, and its waveform is saved to disk.

Figure 3.5 shows the third and final pass of the arbitrator. At the start of the iteration, the request from `UserB:Rx` is still outstanding. The interference controller has requested to transmit another segment. The transcript follows.

```
    Process UserA:Tx
        Skip (module marked as Done)
    Process UserB:Rx
        Request outstanding
        Query results = complete
        Combining received signals
        Save received waveform to disk
        Request done
```



(a) Loop 3: Start



(b) Loop 3: End

Figure 3.5: Example simulation at the start and end of the third pass of `RunArbitrator()`

```
Process Interf:Tx
    Request outstanding
    Call transmit function
    Save transmit waveform to disk
    Request done
```

In this pass, `UserB:Rx` can finally receive. The queries to `UserA:Tx` and `Interf:Tx` both respond *data ready* since data for samples 1 to 100 are available in all transmit modules. The relevant portions of data are loaded from disk and combined as described in Section 3.2.2. Signal combining adds the contributions from all segments that overlap the requested receive segment in time and frequency. The multichannel propagation models are called from within the signal combining block to generate the simulated baseband received signal. This baseband signal is passed into the user-defined receive function and demodulated. The receive request for `UserB:Rx` is marked as complete. Following this, the interference module request is satisfied as usual.

From the previous example, it is easy to see that requests to transmit are never stalled. It is only receive requests that force the simulation to execute out of order.

The operation of the genie channels was not mentioned in this example. In practice, genie module requests work much like transmit requests. Genie transmit requests are processed on the very first pass. Genie receive module will have to wait no more than one iteration if the transmitter and receiver are synchronized properly.

3.1.4 Ending the Simulation Gently

Other than crashing or forcibly quitting MATLAB, there are two ways a simulation may end: all the controller functions must reach a done state, or the arbitrator decides that the simulation has stalled.

The arbitrator decides that *the simulation has stalled if for any single iteration of the main program loop there are outstanding requests, and none could be satisfied.* This may occur, for example, if a receive block requires data samples 10000-12000, but these samples are never available because all the other radios in the simulation stopped transmitting after 5000 samples.

It is for this reason that modules can be marked as *done*. This is accomplished by calling `SetModuleRequest()` with the `job` argument set to `'done'`. Modules marked as *done* are assumed to be not transmitting for all samples beyond what is contained in its history. So, in the example mentioned above, the receive module requesting samples 10000-12000 would proceed by receiving samples with zero signal.

Marking the module as *done* should not be confused with the controller function's `done` state. Controller functions are responsible for returning a status message to the main program loop that is either `'running'` or `'done'`. All controllers begin by returning `'running'`. When a controller reaches the `done` or `finish` state, it returns `'done'` for the status. When all controllers return `'done'` for the status, the main program loop knows that the simulation has successfully finished.

The half-duplex, full-duplex, and LL MIMO controllers in the example section (§2.3) are careful to mark modules *done* when finished. The interference source controllers, in contrast, reach the `done` state but do not mark the modules as *done*. This is so that LLAMAComm will stall if not enough interference samples are produced. (We did not want the example radios to accidentally operate without interference.)

3.2 Environment Modeling and Signal Processing

The environment modeling and signal processing are explained in the following subsections.

3.2.1 Creating Link Objects

The `link` object is created by the function `/@envirnoment/BuildLink.m`, which uses sophisticated environment modeling to build the channel parameters used by the signal processing engine. For debugging purposes, the `link` object also contains the path-loss, the shadow-loss, and other channel properties (see Section 4.5.1).Each `link` is created as needed. For example, if a transmit module and receive module have non-overlapping bands, then no `link` is created between them.

It may occur that the transmit and receive `module` are contained within the same node. In the current version of LLAMAComm, a dummy `link` is created such that there is no self-interference. Future versions of LLAMAComm will address the issue of self-interference.

3.2.2 Combining Signals

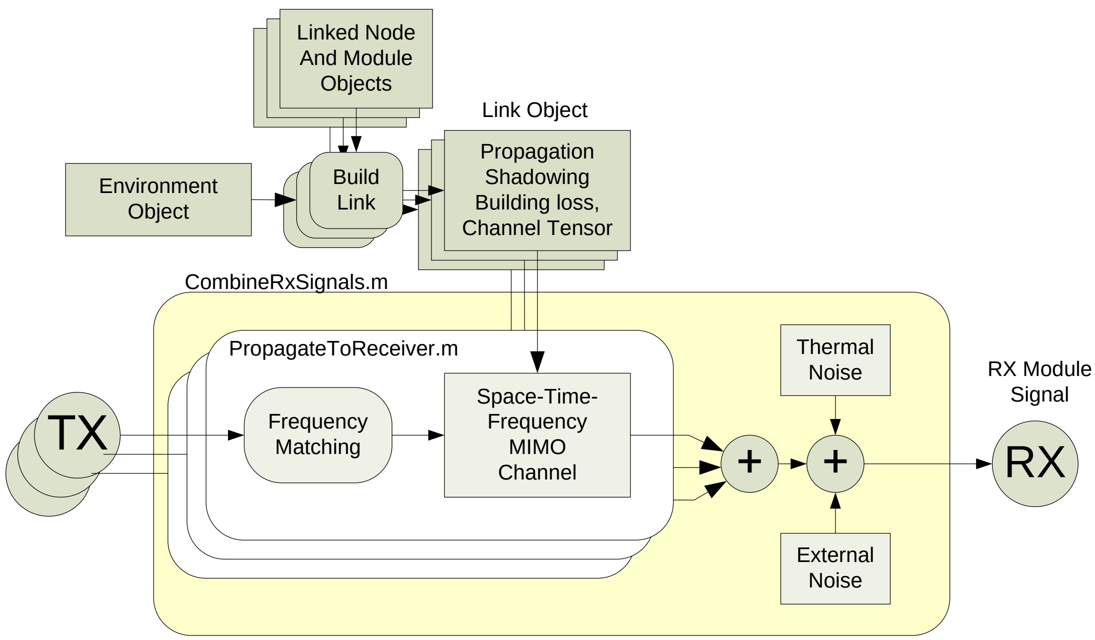


Figure 3.6: Signal processing overview.

The signal processing chassis of LLAMAComm, called by `@node/RunArbitrator.m`, is the function `@node/CombineRxSignals.m`. A block diagram overview of the signal processing is shown in Fig. 3.6. `CombineRxSignals.m` takes as input the analog signals produced by the relevant transmit modules, runs these signals through their respective channels, combines the outputs, and adds white Gaussian noise. The relevant modules are those whose transmit band overlap with the band of the receive module.

A chassis needs an engine to operate, and in the case of LLAMAComm, the signal processing engine is the function `@link/PropagateToReceiver.m`. This function

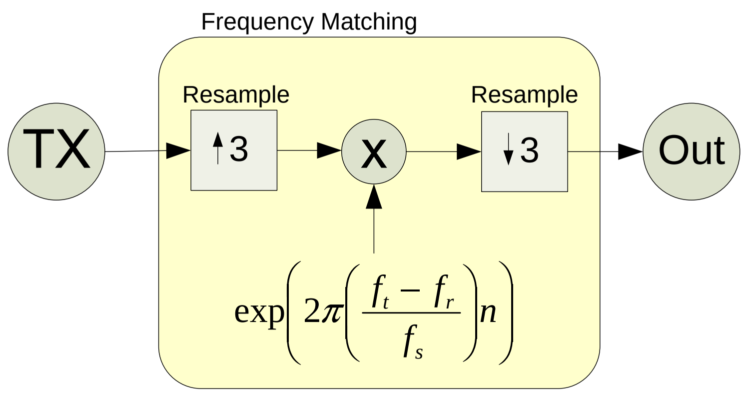


Figure 3.7: Frequency matching block diagram.

reads the specified transmit waveform from file, applies frequency matching and propagation delays when needed, and finally applies the appropriate physical layer channel model specified in the pertinent `link` object. The result is the noiseless received signal seen at the receive `module`channel output.

Frequency matching is needed when the transmit and receive modules have different center frequencies. Figure 3.7 shows the frequency-matching block diagram. The interpolation and decimation is done on a block-by-block basis using the MATLAB built-in function `resample.m`. The function `resample.m` assumes the signal is zero outside given the window of samples; hence, the edges will be mangled. This can be avoided by setting all the transmit and receive modules in the same band to the same center frequency, thereby eliminating the need for frequency matching. In the future, LLAMAComm will have the option to perform non-causal processing to eliminate the edge mangling.

As an example of a case where frequency-matching is necessary, suppose the simulation sample rate is $f_s = 12.5$ MHz and suppose we have transmit and receive modules with center frequencies of $f_t = 98$ MHz and $f_r = 100$ MHz, respectively. The band occupied by the transmit module is $[91.75, 104.25]$ MHz, while the band observed by the receive module is $[93.75, 106.25]$ MHz. The receiver "sees" only the following portion of the transmit signal: $[93.75, 104.25]$ MHz. We can create the proper received signal by interpolating, appropriately modulating, and decimating the transmit signal.

3.2.3 Local Oscillator Errors

This section discusses how LLAMAComm simulates fine local oscillator errors (as opposed to the coarse center frequency mismatches discussed in the previous section). Each module has two properties regarding the local oscillator: `.loError` and `.loCorrection`. Both of these module properties are given in units of parts. For example, to simulate a local oscillator error of 0.95 parts per million, let `loError = 0.95e-6`, resulting in a frequency offset of, e.g., 950 Hz at a center frequency of 1 GHz. The `.loError` module property can only be modified during node construction at the beginning of the simulation.

During simulation, the user can adjust the local oscillator through the `loCorrection` module property by calling the node method `SetLoCorrection.m`. For example, to correct the previous local oscillator error of 0.95 parts per million, set `loCorrection = -0.95e-6`.

The fine frequency offset of the link will be calculated relative to the receiver local oscillator error and the receiver center frequency. For example, given a nominal receive center frequency of `fr` Hz, the link frequency offset will be:
`freqOffset = (loErrorTx + loCorrectionTx - loErrorRx - loCorrectionRx)*fr; % (Hz)`.

The fine frequency offset is the last processing applied to the received signal in `/simulation/@link/PropagateToReceiver.m` as follows:

```
freqOffset = (mTx.loError + mTx.loCorrection ...
           - mRx.loError - mRx.loCorrection)*fr;

if freqOffset ~= 0
    modulation = exp(1i*2*pi*(freqOffset/fs)...
                    *((0:size(rxsig,2)-1) + startRx));
    rxsig = repmat(modulation,size(rxsig,1),1).*rxsig;
end
```

Because there is no filtering after the fine frequency offsetting, an error is thrown if the fine frequency offset is greater than 1% of the simulation sample rate.

3.2.4 Power Measurements

Understanding transmit power measurements is central to connecting the simulation results to the real world. The amplitudes of LLAMAComm simulation signals have SI-units of Volts, and for simplicity, power measurements are taken assuming a 1-Ohm load. For example, the power of a voltage signal $x(t)$ can be obtained from the RMS measurement

$$V_{\text{RMS}}^2 = \frac{1}{T}\int_0^T |x(t)|^2 dt, \tag{3.1}$$

where T is the measurement duration. One can approximate the integral by summing the area of N rectangles each with width $T_s := \frac{T}{N}$ centered at times $\{nT_s\}_{n=0}^{N-1}$, and with height $|x(nT_s)|^2$. Mathematically, this approximation is written

$$V_{\text{RMS}}^2 \approx \frac{1}{T}\sum_{n=0}^{N-1} |x(nT_s)|^2 T_s, \tag{3.2}$$

$$= \frac{1}{N}\sum_{n=0}^{N-1} |x(nT_s)|^2. \tag{3.3}$$

Therefore, the power in Watts (across a 1-Ohm load) of a simulated waveform is simply the sample autocorrelation at lag zero, i.e., the average squared modulus of the signal.

3.2.5 Additive Noise

Adding white Gaussian noise to the received signal is a fundamental part of any detection or estimation simulation. The additive noise in LLAMAComm is generated by the function `@environment/GetAdditiveNoise(env,modRx)`. The formula for calculating the variance of the zero-mean i.i.d. complex-valued noise samples is as follows:

$$\sigma_w^2 = KTf_s(F_{\text{ext}} + F_{\text{int}}) \tag{3.4}$$

where $K := 1.38 \times 10^{-23}$ is Boltzmann's constant, $T := 300$ is the temperature in Kelvin, f_s is the simulation sample rate, F_{ext} is the external noise factor, and F_{int} is the internal noise factor (i.e., in dB the noise figure).

Chapter 4

Reference

The reference chapter defines the global variables and the various MATLAB objects implemented specifically for LLAMAComm. Descriptions of important properties and method functions are provided.

4.1 Global Variables Defined

The file `InitGlobals.m` holds parameters that are shared across the entire simulation. This file should not be modified. Instead, you should copy it and name it something like `InitGlobals_myname.m`. **User code should not be added to this file! Variables defined here should never be written to while the simulation is running.** The global variables are listed and described below:

`simulationSampleRate` The universal baseband sample rate is defined by this global variable. This value may be modified by the user. However, this value affects all the the signal processing in the simulation, so the value should be changed with care. Although it is possible to read this value within user-defined functions, this is not recommended. We instead recommend using the function `GetModFs()`, to return the sample rate which is stored within every module object in the scenario.

`chanType` This global variable determines the channel propagation model used by LLAMAComm. Options are '`wssus`', '`stfcs`', '`los_awgn`', '`wideband_awgn`', or '`env_awgn`'. The '`wssus`' channel model generates channel realizations based on a wide-sense stationary uncorrelated-scattering (WSSUS) model with exponential power profile where the space-lag taps are i.i.d. and vary according to Jake's model. There are no spatial correlations; thus, the '`wssus`' channel model is more appropriate for single antenna simulations. The ‘`stfcs`’ channel model creates space-time-frequency-correlated channel taps; however, the power profile is not exponential. The model choice ‘`los_awgn`’ only applies line-of-sight propagation loss and should be used only for debugging. The channel model ‘`env_awgn`’ is the same as ‘`los_awgn`’ but uses the given environment’s median path loss and antenna gains to compute the total path loss. It removes any effects of shadow loss and Rayleigh fading and ignores delay spread and Doppler settings. The ’`wideband_awgn`’ model implements fractional sample time delays between transmit and receive antennas with line of site loss. The channel models are described more detail in the tutorial powerpoint presentation provided with the LLAMAComm distribution.

`includePropagationDelay` Set this global flag to include propagation delays between transmit and receive modules. The delay is proportional to the distance between the transmitter and receiver and is rounded to the nearest sample.

`includeFractionalDelay` Set this global flag to include fractional-sample delays. This is done by applying a non-causal, length-63, fraction-delay filter to the signals.

`randomDelaySpread` Setting this global variable to 1 makes the delay spread for each link a log-normal random variable paramaterized by the link distance and correlated with the link shadow loss. (See `/@node/GetDelaySpread.m` for more details.) If this global variable is set to 0, then the delay spread for all links in the simulation will be equal to the environment property `env.propParams.delaySpread`. The random delay spread model is based on environmental parameters and is taken from [Greenstein, IEEE Trans. Veh. Technology, May 1997].

`saveRootDir` The save directory for saving simulation results.

`savePrecision` The precision of the simulation save files.

`timingDiagramFig` The figure number associated with the timing diagram. If set to zero, the timing diagram is not created.

`timingDiagramForceRefresh` Forces the timing diagram to be redrawn after each block. This allows you to see the blocks displayed as they are calculated, but slows down the simulation significantly.

`timingDiagramShowExecOrder` (true or false) Controls whether or not the execution order is displayed on the blocks in the timing diagram.

`addGaussianNoiseFlag` This flag turns the additive Gaussian noise on/off—used for debugging.

`DisplayLLAMACommWarnings` LLAMAComm warnings are printed to the command window if this flag is set.

4.2 @node

The `node` object is the container for the link-layer radio properties and method functions, which are defined in the following.

4.2.1 Node Properties

The `node` properties describe the top-level aspects of a radio.

`.name` (string) Name of the node. Must be unique in the simulation.

`.location` (1x3 double array) $[x, y, z]$ (m) Location of the `node`'s local antenna coordinate system. The local coordinate system axes are a translation of the global coordinate system, i.e., the x, y, and z axes in the two systems are parallel.

`.velocity` (1x3 double array) $[x, y, z]$ (m/s) Velocity vector of the node relative to the local coordinate system.

`.controllerFcn` (string) Controller callback function handle. For example: `node.controllerFcn = @node_controller`.

`.state` (string) Current state of the `node`'s controller function. States names are user-defined. The first state *must* be named `start`. We recommend that the final state be named `done` or `finished`.

`.modules` (1xN module obj array) Array of `module` objects associated with the node.

`.isCritical` (bool) indicates if the node is critical to the simulation. Note that nodes are critical by default. If only non-critical nodes are running, the arbitrator forces them into the done state so that the simulation may terminate. This is useful for interference nodes or other passive nodes that only need to run when other critical nodes are still operating; otherwise, the user must coordinate between the nodes to gracefully terminate the simulation without stalling.

4.2.2 Node Methods

User functions only have access to the node object. Module objects are stored within node objects and not directly accessible. As a result, the functions listed here are the only ones that should be used to interact with node objects and their modules from within the user-defined files.

Please refer to the the help information[1] or see the actual MATLAB code in `llamacomm/simulator/@node` for a complete description of the input and output arguments. Note that in the function arguments, `nodeobj` is an actual node object, while `modname` is a string containing the module of interests' name. Methods listed below have been divided by functionality.

Extracting Basic Properties

These functions are used to recover basic information about the node or module.

`GetNodeName(nodeobj)` Returns the name of the specified `node` object.

`FindNode(nodearray,nodename)` Given an array of node objects, finds and returns the node object with the specified name.

`GetNumModAnts(nodeobj,modname)` Returns the number of antennas of the named `module` in the specified `node`.

`GetModFc(nodeobj,modname)` Returns the named `module`'s current center frequency.

`GetModFs(nodeobj,modname)` Returns the global simulation sample rate. Note that the simulation sample rate is stored in every module.

`GetLoCorrection(nodeobj,modname)` Returns the current local oscillator correction factor (parts) of the named module.

Setting Basic Properties

Only the center frequency and local oscillator correction factor (parts) of the module can be modified during a simulation run.

`SetModFc(nodobj,modname,newFc)` Sets the center frequency of the named `module` in the specified `node` to the new value `newFc` (Hz).

[1] Help is available from within MATLAB by typing `help functionname`.

`SetLoCorrection(nodobj,modname,loCorrection)` Sets the local oscillator correction factor of the named `module` in the specified `node` to the new value `loCorrection` (parts). For example, to adjust by -0.95 parts per million, let `loCorrection = -0.95e-6`.

Controller Function

These functions deal with controller states and arbitrator requests. They are called from within the controller function.

`GetNodeState(nodeobj)` Returns the current `node` state. Used at the beginning of the controller function.

`SetNodeState(nodeobj,state)` Sets the `node` state to the specified state. Used at the end of the controller function.

`CheckRequestFlags(nodeobj)` Queries all modules within a node to see if there are any outstanding requests to the arbitrator.

`SetModuleRequest(nodobj,modname,job,blockLen)` Sets request flag high for the named `module` so that its requested function will be executed by the simulation arbitrator. The argument `blockLen` (number of time samples) is required when the module is not a genie module. (See §2.3 and §3.1 for more information on how this function is used.)

`SetGenieRxRequest(nodeobj,modname)` Similar to `SetModuleRequest()`, but used for genie modules to receive.

`SetGenieTxRequest(nodeobj,modname,toNodeName,toModName)` Similar to `SetModuleRequest()`, but used for genie modules to send. To multi-cast to multiple genie modules, the `toNodeName` and `toModName` can be (equal sized) cell arrays. Data is delivered to the genie queue for each specified node/module address.

`ReadGenieInfo(nodeobj,modname)` Reads `info` structure from genie receive queue for the named `module`.

`WriteGenieInfo(nodeobj,modname,info)` Adds `info` structure into the genie transmit queue for the named `module`.

User Parameters

The user-defined parameters are used to store information such as the training sequence, packet counters, and anything else that might be used within the firmware of a radio.

`GetUserParams(nodeobj)` Returns the user-defined parameter structure contained in the `node` object.

`SetUserParams(nodobj,p)` Saves the user-defined parameter structure, `p`, into the node object.

Debugging Functions

These functions produce no output to the workspace, but are useful for debugging.

`MakeNodeMap(nodes,mapFig)` Displays a map of the locations of the `node` objects in the array `nodes`. The optional variable `mapFig` indicates the figure number in which the map will be displayed. The default is to create a new figure. This function is called from within the startup file in the examples provided.

`DisplayModule(nodeobj,modname)` Prints the contents of the named `module` to screen. This is used for debugging only.

4.3 @module

The `module` object simulates the physical layer of the communication system. Below, we list the `module` properties and member method functions.

4.3.1 Module Properties

The `module` properties are defined as follows:

Properties Defined During Object Construction

`.name` (string) Module name. Each `module` in a node must have a different name.

`.fc` (double) (Hz) Center frequency of the module.

`.fs` (double) (Hz) The simulation sample rate is read-only.

`.type` (string) Module type: '`transmitter`' or '`receiver`'.

`.callbackFcn` (string) Module call back function handle name. For example, `mod.callbackFcn = @mod_transmit`.

`.loError` (double) (parts) Local oscillator error. For example, if the local oscillator error is -1 part per million, then `loError = -1e-6`.

`.loCorrection` (double) (parts) User defined local oscillator frequency correction factor. This property can be changed dynamically to correct for the local oscillator errors. During simulation of a link, the overall frequency offset will be calculated relative to the receiver local oscillator error. For example, given a nominal receive center frequency of `fr`, the link frequency offset will be: `freqOffset = (loErrorTx + loCorrectionTx - loErrorRx - loCorrectionRx)*fr;`.

`.noiseFigure` (double) (dB) The noise figure is only used if the module is a receiver.

`.antType` (string cell array) The antenna type must have a corresponding function in `llamacomm/simulator/pathloss/antennas` with the same name. Currently, LLAMAComm requires all antennas to be the same type.

`.antPosition` (Nx3 double) $[x, y, z]$ (m) Position of the `module` antennas in the `node`'s local coordinate system. N is the number of antennas.

`.antPolarization` (string) Antenna polarization: '`h`', '`v`', '`rhcp`', or '`lhcp`'.

`.antAzimuth` (1x1 or 1xN double) (radians) Antenna azimuth look angle in the range $(-\pi, \pi]$. Measured in the x-y plane in the counter-clockwise direction from the positive x axis. Currently, LLAMAComm requires all antennas to have the same look angle.

`.antElevation` (1x1 or 1xN double) (radians) Antenna elevation look angle in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Measured from the x-y plane with positive angles in the positive z-axis direction.

`.exteriorWallMaterial` (string) Material of the building's exterior wall. Currently, the supported materials are '`concrete`' or '`none`'.

`.distToExteriorWall` (double) (m) Distance of antennas to exterior wall.

`.numInteriorWalls` (int) Number of walls separating the `module` from the exterior wall.

`.exteriorBldgAngle` (double) (deg) Angle from the face of the building. Currently, LLAMAComm only uses this parameter when the nodes are in close proximity.

`.TDCallbackFcn` (string) Optional user-defined callback function for producing output when the timing diagram is clicked on for this module. Example: `mod.TDCallbackFcn = @mod_TDCallbackFcn;`. For an example of a user-defined timing diagram callback function see `/user/LL/LLMimo_Rx_TDCallback.m`.

Properties Related to Arbitrator Requests

`.job` (string) Specifies what job the module is performing. This parameter is one of the following: `wait`, `receive`, `transmit`, or `done`. Once a module is marked as `done`, it must always remain done. Attempting to change the job will result in an error.

`.requestFlag` (bool) The request flag is set to 1 when there is an outstanding request. It is 0 when there is no request. `SetRequest()` sets this flag high. `RequestDone()` sets it low.

`.blockStart` (int) Sample index of the start of the "current" segment. This number starts at 1 at the beginning of the simulation, and is incremented by the function `RequestDone()` as the simulation runs.

`.history` (struct array) Structure that maintains a history of all signal segments processed for the module. The history is what is queried by the arbitrator to determine if the required data is ready during a receive (see §3.1.2). The fields within each structure of this array are:

- `.start` (int) Sample index for the start of signal segment
- `.blockLen` (int) Number of samples in signal segment
- `.job` (string) The job performed within this segment
- `.fc` (float) Center frequency (Hz)
- `.fs` (float) Sample Rate (Hz) This is always the baseband sample rate defined by the global simulation sample rate
- `.fPtr` (int) Index into the signal file that points to the data for this segment of signal.

Fields for Storing Data to Disk

`.filename` (string) Full path and filename of the `.sig` file used to store baseband data for a transmit or receive module.

`.fid` (file ID) MATLAB file identifier used to read data from the file. This is the argument that is returned by using MATLAB's fopen function.

Special Fields for Genie Modules

`.isGenie` (bool) Flag set within a module object to mark a module as a genie module. This flag is set during construction of the module object.

`.genieToNodeName` (string) Genie transmit modules are allowed to send to any genie receive module. This is the name of the node containing the genie module that the info is being sent to. This field is filled by SetGenieTxRequest() as part of a request to transmit.

`.genieToModName` (string) The genie module name to send to. This field is also filled by SetGenieTxRequest().

`.genieQueue` (struct array) Array of structs for queuing received genie messages until they are read by ReadGenieInfo().

4.3.2 Module Methods

The user does not have access to the `module` object; hence, the `module` method functions cannot directly be called. The user manipulates the `module` properties by calling the appropriate `node` member function (see Section 4.2.2).

4.4 @environment

The environment object contains the properties that are applicable to every link in the simulation. It is also the container of the link object array and the shadowloss struct.

4.4.1 Environment Properties

The user deals very little with the environment object. Only the class constructor is used to set up the environment properties at the beginning of the simulation.

Required User-Define Properties

These properties are set using the class constructor `environment()` when setting up the simulation. (See §2.3.5 for an example.)

`.envType` (string) The environment type can be one of the following: '`rural`', '`suburban`', or '`urban`'.

`.propParams.delaySpread` (double) (s) Maximum time-difference of arrival of reflected signals. This property is ignored if the `randomDelaySpread` global variable flag is set (see Section 4.1).

`.propParams.velocitySpread` (double) (m/s) Maximum Doppler spread induced by the ambient environment.

`.propParams.alpha` (double) Unitless property between 0 and 1 quantifying the amount of spatial correlation in the MIMO channels. Setting $\alpha = 0$ implies line of sight; setting $\alpha = 1$ implies uncorrelated spatial fading. This property is only used if the '`STFCS`' channel model option has been specified in the `chanType` global variable.

`.propParams.longestCoherBlock` (double) (s) Property used to calculate the channel tensor in `/simulator/channel/GetStfcsChannel()`. This property is only used if the '`STFCS`' channel model option has been specified in the `chanType` global variable.

`.propParams.stfcsChannelOversamp` (int) Property used to calculate the channel tensor in `@node/GetChannelTensor()`. This property is only used if the '`STFCS`' channel model option has been specified in the `chanType` global variable.

`.propParams.wssusChannelTapSpacing` (int N) Places $N-1$ zeros between the channel taps to decrease the processing complexity. This is especially useful when the bandwidths of the simulated signals are much less than the simulation sample rate. A reasonable formula to determine N is as follows:

$$N = \frac{f_s}{8B_{\max}}, \tag{4.1}$$

where $B_{\max}$ is the largest signal bandwidth and f_s is the simulation sample rate. Thus, N corresponds to the channel taps spaced at $1/8^{th}$ the symbol rate. Most of the time, however, one should set $N = 1$. This property is only used if the '`wssus`' channel model option has been specified in the `chanType` global variable.

`.los_dist` (double) (m) Link distance below which the link is considered to be line of sight for propagation loss calculation.

`.building.avgRoofHeight` (double) (m) Average building roof height in the simulation.

Internal Simulator Properties

Most of the properties in the environment object are manipulated by the simulator internally.

`.links` (1xN link obj) Array of `link` objects created by `@link/BuildLink.m`.

`.shadow` (struct) Structure of shadowloss properties created by `@environment/SetupShadowloss.m`.

4.4.2 Environment Methods

The user does not have access to the `environment` object during runtime; hence, the `environment` method functions cannot directly be called. After execution, the following function may be called to investigate the link properties:

`DisplayLinkParams(env,linkIndex)` Displays the properties of the `link` object identified by its link index `linkIndex`. The link index can be obtained by typing `env` at the command prompt (see Section 2.4).

4.5 @link

The `link` object is the container for the link properties and the channel tensor. Each `link` is created as needed. For example, if a transmit module and receive module have non-overlapping bands, then no `link` is created between them. All `link` properties are derived through LLAMAComm's sophisticated environment modeling code from the `node`, `module`, and `environment` object properties.

4.5.1 Link Properties

The link properties are as follows:

`.channel` (struct) Structure containing the channel tensor and other channel properties. Created by `@node/GetChannelStruct.m`.

`.pathLoss` (struct) Structure containing the pathloss properties. Created by `@node/GetPathlossStruct.m`.

`.propParams` (struct) Structure containing the propagation properties. Created by `@node/GetPropParamsStruct.m`.

`.antialiasTaps` (1xN double array) Filter used to perform anti-alias filtering when performing band-matching in `tools/ProcessTransmitBlock.m`.

`.fromID` (cell array) Transmitting module identifier: `{'nodeName','moduleName'}`.

`.toID` (cell array) Receiving module identifier: `{'nodeName','moduleName', fc}`, where `fc` is the receiving module's center frequency.

4.5.2 Link Methods

The user does not have access to the `link` object during runtime; hence, the `link` method functions cannot directly be called. After execution, the `link` object properties can be displayed to the screen by calling the function `@environment/DisplayLinkParams()` (see Section 4.4.2).

4.6 Utility Functions

Utility functions are bits of code that we found useful when developing and testing LLAMAComm. They are included in `llamcomm/simulator/tools`. A description of some of these functions follows.

`StructMerge(s,f)` Merges the contents of struct `f` into struct `s`. If a particular field already exists in `s`, it is overwritten. Useful for managing user parameters.

`FieldCopy(s,f)` Copies the values from fields in `f` into the corresponding field in `s`. If the field does not exist in `s`, an error is generated.

`db10(),db20(),undb10(),undb20()` Converts values to and from dB.

4.7 File Input and Output

To be memory efficient, much of the simulation data is cached to file as the simulation runs. A number of functions have been written to facilitate the process of reading and writing data to file. All of these files are located in `llamacomm/simulator/fileio/`. Please refer to the function help for input and output arguments.

4.7.1 Info Source

The info source functions open a binary data file for reading and use the bits as data to be transmitted during a simulation run. (An alternative is to use randomly generated data.) The LL MIMO example in §2.3.4 uses these functions to load data bits for transmission.

`InitInfoSource()` Opens a binary data file for reading.

`ReadInfoBits()` Reads N bits (multiple of 8) from file and returns the bits in a 1xN array.

`InfoBitsRemaining()` Returns the number of bits remaining in the data file that has been opened for reading.

4.7.2 Bits Files, `.bit`

The "data bit" functions are intended to be used for storing transmitted bits and demodulated bits for calculating the bit-error rate. Each bit is stored as a separate byte. This is inefficient, but more convenient for reading/writing an arbitrary number of bits. Bit blocks may contain multichannel data.

File Format

Data bit files have the extension `.bit`. It is a custom block-based file format. The structure of each block is displayed in Table 4.1. The field named `blocksize` contains the block size in bytes. This information is used to navigate up and down the file one block at a time.

File Functions

The functions used to read/write to the `.bit` files are shown below. These functions are also used by the LL MIMO example in §2.3.4.

Table 4.1: `.bit` file block format

Start Index	Field Name	Format	# Bytes
0	[blocksize]	uint32	4
4	nSamps	uint32	4
8	nChannels	uint32	4
9	samples	uint8	$1 \times nSamps$
$9 + nSamps$	[blocksize]	uint32	4

`InitBitFile()` Starts a new `.bit` file for writing.

`OpenBitFile()` Opens an existing `.bit` file for reading.

`ReadBitBlock()` Reads a block of bits from an open file. Requires a file offset that points to the start of a block.

`WriteBitBlock()` Writes a block of bits to file. The block of bits may be multichannel (M channels x N samples)

`NextBitBlock()` Returns a file offset pointing to the start of the next block.

`PrevBitBlock()` Returns a file offset pointing to the start of the previous block.

4.7.3 Signal Files, `.sig`

Binary files with the extension `.sig` are used to store the baseband samples for all transmit and receive modules. These files are created automatically by the arbitrator. When the simulation is running, only signal segments being processed are kept in memory. Signal segments not in use are stored in the `.sig` files. (See §sec:simOutputFiles for file naming conventions.) These files remain when the simulation finishes. The data within is used to generate the timing diagram callback plots and can also be used for post-analysis.

File Format

`.sig` files are block-based. The format of each block is shown in Table 4.2. By default, LLAMAComm stores data in `.sig` files as single-precision floating point. This can be changed to double-precision by modifying a parameter in the `InitGlobals` file (see §2.3.7).

Table 4.2: `.sig` file block format

Start Index	Field Name	Format	# Bytes
0	[blocksize]	uint32	4
4	startIdx	uint32	4
8	nSamps	uint32	4
12	nChannels	uint32	4
16	precision	uint32	4
20	bytes/sample	uint32	4
24	fc	float64	8
32	fs	float64	8
40	samples	float32/64	N
$40 + N$	[blocksize]	uint32	4

The size of the field named `samples` varies depending on the number of samples and the precision. Single-precision uses 4 bytes per sample, whereas double-precision uses 8 bytes per sample.

The number of bytes required to hold the samples, N, is calculated as shown in Equation 4.2. The factor of 2 is required because the data is complex (real and imaginary).

$$N = 2 \times nChan \times nSamps \times bytes/sample \tag{4.2}$$

File Functions

The functions used to read/write to the `.sig` files are shown below. These are used internally by the arbitrator and by the timing diagram callback function. It is also possible to use these functions to manually read the `.sig` files for analysis of the baseband samples using custom algorithms.

`OpenSigFile()` Opens a `.sig` file for reading.

`ReadSigBlock(fid,fPtr)` Reads a signal block from file. Requires a file offset (or pointer) that points to the start of a valid block.

`WriteSigBlock()` Appends a signal block to the end of a `.sig` file. This function should not be used by the user. It is included here for completeness.

`NextSigBlock()` Returns the file offset to the next valid signal block.

`PrevSigBlock()` Returns the file offset to the previous valid signal block.

`ReadSigBlockAdj()` Returns the requested block of data (specified using a file offset) along with the adjacent blocks of data before and after. This function is used by the default timing diagram callback function to generate the time domain plot.

The functions `NextSigBlock()` and `PrevSigBlock()` are provided to allow simple navigation of the `.sig` files. However, the simulator itself uses the module history records to quickly access a block of signal. Each history entry corresponds to a particular segment and contains a file pointer to the corresponding block in storage (see §4.3.1). The history records are used by the arbitrator to determine which segments to use. The relevant segments are then read into memory by the signal processing functions using the file pointers stored in the history record. It is this caching method that allows LLAMAComm run long simulations without using much memory.

4.8 Acknowledgements

The authors wish to acknowledge Derek P. Young for his contributions to the arbitrator design and the writing of this document.