

# Wireless HDL Toolbox™

User's Guide



# MATLAB® & SIMULINK®

R2020b



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

## *Wireless HDL Toolbox™ User's Guide*

© COPYRIGHT 2017 - 2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## **Revision History**

September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 2.0 (Release 2020a)
September 2020	Online only	Revised for Version 2.1 (Release 2020b)

## Model Architecture

### 1

<b>Streaming Sample Interface</b> .....	<b>1-2</b>
What Is a Streaming Sample Interface? .....	1-2
How Does a Streaming Sample Interface Work? .....	1-2
Why Use a Streaming Sample Interface? .....	1-2
Sample Stream Conversion .....	1-3
Timing Diagram of Serial Sample Interface .....	1-3
Using the nextFrame Output Signal .....	1-4
<b>Sample Control Bus</b> .....	<b>1-7</b>
Troubleshooting: .....	1-7
<b>Configure the Simulink Environment for Hardware Design</b> .....	<b>1-8</b>
About Simulink Model Templates .....	1-8
Create Model Using Wireless HDL Toolbox Model Template .....	1-8
Wireless HDL Toolbox Model Templates .....	1-9

## HDL Code Generation and Verification

### 2

<b>HDL Code Generation Support</b> .....	<b>2-2</b>
HDL Code Generation Support in Wireless HDL Toolbox .....	2-2
Other Blocks Supporting HDL Code Generation .....	2-2
Streaming Sample Interface in HDL .....	2-3
<b>Generate HDL Code</b> .....	<b>2-5</b>
Prepare Model .....	2-5
Generate HDL Code .....	2-5
Generate HDL Test Bench .....	2-5
<b>FPGA-in-the-Loop</b> .....	<b>2-6</b>
FIL Workflow: Framed Data from MATLAB .....	2-6
FIL Workflow: Streaming Data from MATLAB .....	2-8
<b>Prototype LTE Algorithms on Hardware</b> .....	<b>2-12</b>
How to Install Support Packages .....	2-12
Design Requirements .....	2-13
Design for Debugging .....	2-13

<b>Append CRC Checksum to Streaming Data</b> .....	<b>3-2</b>
<b>Check for CRC Errors in Streaming Samples</b> .....	<b>3-4</b>
<b>Turbo Encode Streaming Samples</b> .....	<b>3-6</b>
<b>Turbo Decode Streaming Samples</b> .....	<b>3-8</b>
<b>Convolutional Encode of Streaming Samples</b> .....	<b>3-11</b>
<b>Convolutional Decode of Streaming Samples</b> .....	<b>3-13</b>
<b>Descrambling with Gold Sequence Generator</b> .....	<b>3-16</b>
<b>Parallel Gold Sequence Generation</b> .....	<b>3-18</b>
<b>LTE OFDM Demodulation of Streaming Samples</b> .....	<b>3-20</b>
<b>Reset and Restart LTE OFDM Demodulation</b> .....	<b>3-24</b>
<b>Modulate and Demodulate LTE Resource Grid</b> .....	<b>3-28</b>
<b>OFDM Modulation of LTE Resource Grid Samples</b> .....	<b>3-31</b>
<b>Depuncture and Decode Streaming Samples</b> .....	<b>3-34</b>
<b>LTE Symbol Modulation of Data Bits</b> .....	<b>3-38</b>
<b>NR Symbol Modulation of Data Bits</b> .....	<b>3-40</b>
<b>LTE Symbol Demodulation of Complex Data Symbols</b> .....	<b>3-42</b>
<b>NR Symbol Demodulation of Complex Data Symbols</b> .....	<b>3-45</b>
<b>Application of FFT 1536 block in LTE OFDM Demodulation</b> .....	<b>3-48</b>
<b>Convolutional Encode and Puncture Streaming Samples</b> .....	<b>3-51</b>
<b>OFDM Demodulation of Streaming Samples</b> .....	<b>3-54</b>
<b>Decode and recover message from RS codeword</b> .....	<b>3-57</b>
<b>LDPC Encode and Decode of Streaming Data</b> .....	<b>3-59</b>
<b>Estimate Channel Using Input Data and Reference Subcarriers</b> .....	<b>3-63</b>
<b>Modulate and Demodulate OFDM Streaming Samples</b> .....	<b>3-71</b>
<b>Polar Encode and Decode of Streaming Samples</b> .....	<b>3-74</b>

4

Sample Rate Conversion for an LTE Receiver .....	4-2
HDL Code Generation for Filtered OFDM (F-OFDM) Transmitter .....	4-15
HDL Implementation of a Variable-Size FFT .....	4-25
Accelerate BER Measurement for Wireless HDL LTE Turbo Decoder ...	4-35
Encode message to RS codeword .....	4-41
HDL Implementation of AWGN Generator .....	4-44
HDL Implementation of Digital Predistorter .....	4-55
Encode Streaming Data Using General CRC Generator HDL Optimized Block for 5G NR Standard .....	4-60

Reference Applications

5

NR HDL MIB Recovery .....	5-2
NR HDL Cell Search and MIB Recovery MATLAB Reference .....	5-14
NR HDL Cell Search .....	5-30
LTE HDL Cell Search .....	5-46
LTE HDL SIB1 Recovery .....	5-63
LTE HDL MIB Recovery .....	5-80
LTE HDL PBCH Transmitter .....	5-91
HDL OFDM MATLAB References .....	5-107
HDL OFDM Transmitter .....	5-121
HDL OFDM Receiver .....	5-136



# Model Architecture

---

## Streaming Sample Interface

In this section...
“What Is a Streaming Sample Interface?” on page 1-2
“How Does a Streaming Sample Interface Work?” on page 1-2
“Why Use a Streaming Sample Interface?” on page 1-2
“Sample Stream Conversion” on page 1-3
“Timing Diagram of Serial Sample Interface” on page 1-3
“Using the nextFrame Output Signal” on page 1-4

### What Is a Streaming Sample Interface?

In hardware, processing an entire frame of data at one time has a high cost in memory and area. To save resources, serial processing is preferable in HDL designs. Wireless HDL Toolbox blocks operate on one sample at a time rather than a frame. The blocks accept and return data as a serial stream of samples and control signals. The control signals indicate the frame boundaries. The protocol mimics the characteristics of a real-world system, including inactive intervals between samples and frames.

### How Does a Streaming Sample Interface Work?

The control protocol uses start and end signals to demark each frame, and a valid signal to indicate which samples to process. The Wireless HDL Toolbox streaming sample protocol allows you to configure the number of idle cycles between samples and between frames. Idle cycles model the bursty character of real-world systems.

This protocol allows for frames of different sizes, such as if runt or partial frames enter the system due to synchronization changes.

### Why Use a Streaming Sample Interface?

#### Format Independence

The blocks that use this interface do not need a configuration option for an exact frame size or inactive intervals. In addition, if you change the input data timing for your design, you do not need to update each block. Instead, update the stream configuration once at the serialization step. Some blocks still require a maximum frame size parameter to allocate memory resources.

#### Error Tolerance

By using a streaming sample interface with control signals, each Wireless HDL Toolbox block starts computation on a fresh set of samples at the start-of-frame signal. Computations on the new frame occur whether or not the block receives the end signal for the previous frame.

The protocol tolerates minor timing errors. If the number of valid and invalid cycles between start and end signals varies, the blocks continue to operate correctly. This protocol makes the system resilient to runt frames and synchronization changes.

The Wireless HDL Toolbox encoder blocks require minimum between-frame spacing to accommodate insertion of codewords. The turbo and convolutional decoder blocks require that the previous frame



is decoded (has asserted the frame end signal) before the next frame arrives. The polar, LPDC, and RS encoder and decoder blocks provide a signal to indicate when the block is ready to receive the start of a new frame.

## Sample Stream Conversion

Use the Frame To Samples block to convert framed data to a stream of samples and control signals that conform to this protocol. The control signals are grouped in a bus data type called `samplecontrol`.

The Frame To Samples block can serialize fixed-size frames. If your frames vary in size, use the `whdlFramesToSamples` function to convert framed data to vectors of samples and control signals in MATLAB®. Then import the vectors to Simulink®. Use the Sample Control Bus Creator block to create a `samplecontrol` bus in your model.

If your data is already in a serial format, design your own logic to generate these control signals from your existing serial control scheme.

### Supported Sample Data Types

Wireless HDL Toolbox blocks have an input and output port, `sample`, for the streaming sample data. The blocks capture one sample at a time from the input, and produce one sample at a time for output. The samples can be one of these supported data types.

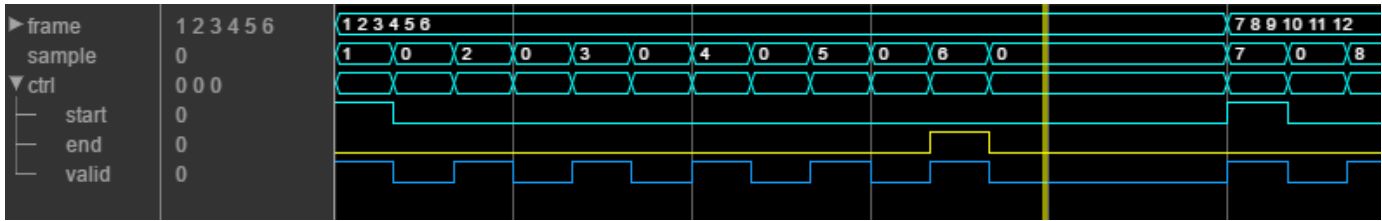
Port	Description	Data Type
<code>sample</code>	Scalar integer value that represents one sample.  The protocol also allows for a vector of integer values that represent a single sample, such as for turbo-encoded samples.	Supported data types include: <ul style="list-style-type: none"> <li>• Boolean</li> <li>• <code>uint</code> or <code>int</code></li> <li>• <code>ufix</code> or <code>sfix</code></li> </ul> <code>double</code> and <code>single</code> are supported for simulation but not for HDL code generation.

### Streaming Sample Control Signals

Wireless HDL Toolbox blocks have an input and output port, `ctrl`, for the frame control signals relating to each sample. These three control signals indicate the validity of a sample and the boundaries of the frame. The control signal port is a nonvirtual bus data type called `samplecontrol`. For details of the bus data type, see “Sample Control Bus” on page 1-7.

## Timing Diagram of Serial Sample Interface

The timing diagram illustrates the streaming sample protocol. It shows a six-sample input frame and the equivalent sequence of control and data signals.



The input frame is  $([1\ 2\ 3\ 4\ 5\ 6])'$ , and the serializer is configured to insert idle cycles around the valid samples:

- One idle cycle between samples
- Three idle cycles between frames
- One value representing each sample (default output size)

You can specify these parameters by using either the `Frame To Samples` block or the `whdlFramesToSamples` function.

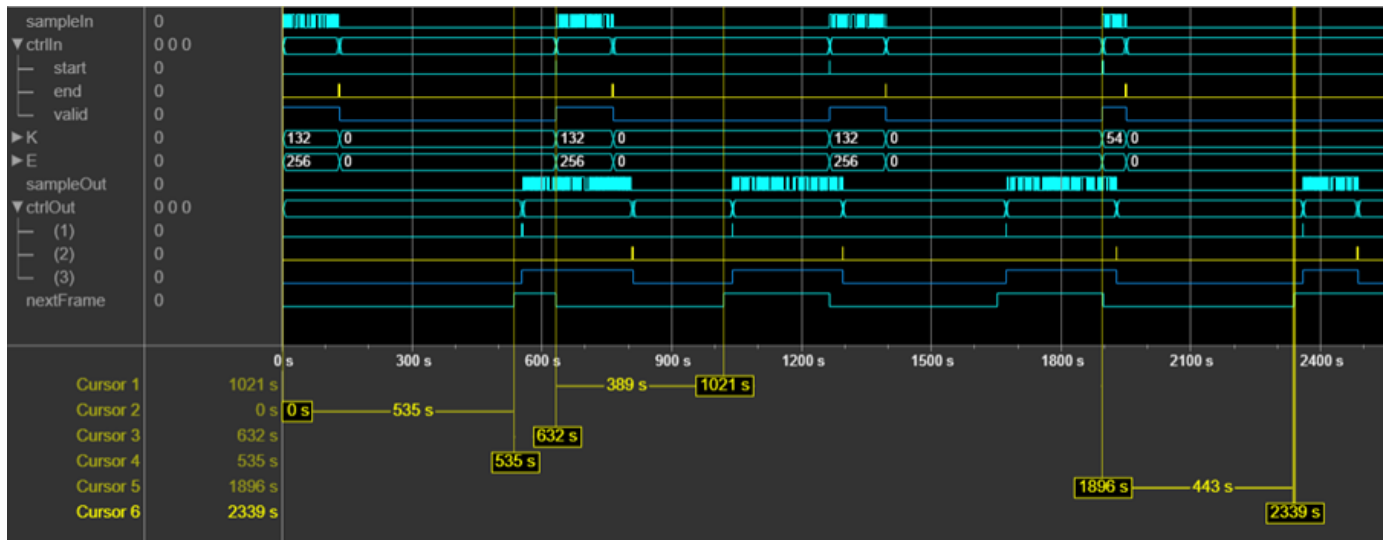
The control signals `start` and `end` are 1 for the first and last valid samples of the frame, respectively. The `valid` signal is 1 for each valid input sample. The `valid` signal is 0 for the idle cycles inserted between the samples and between the frames. The six-sample frame is now represented by streaming data over 15 cycles.

### Using the nextFrame Output Signal

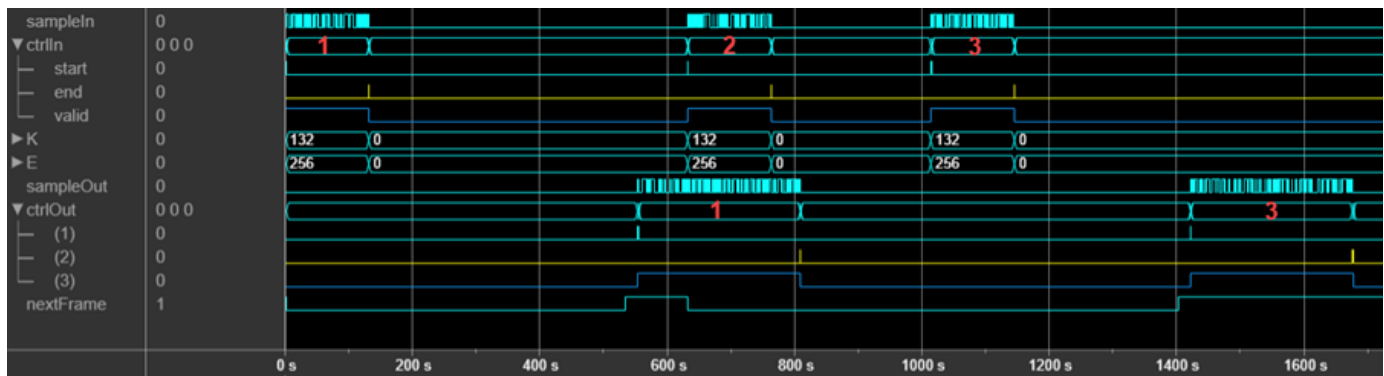
The NR Polar Encoder, NR Polar Decoder, NR LDPC Encoder, NR LDPC Decoder, and RS Decoder blocks each provide an output signal to indicate when the block is ready to receive the start of a new frame. This signal is necessary because these blocks cannot accept a new frame at certain stages of internal computations, and the latency of those stages can vary with the values of input ports.

Port	Description	Data Type
<code>nextFrame</code>	Boolean scalar that indicates when the block can accept the start of a new frame	Boolean

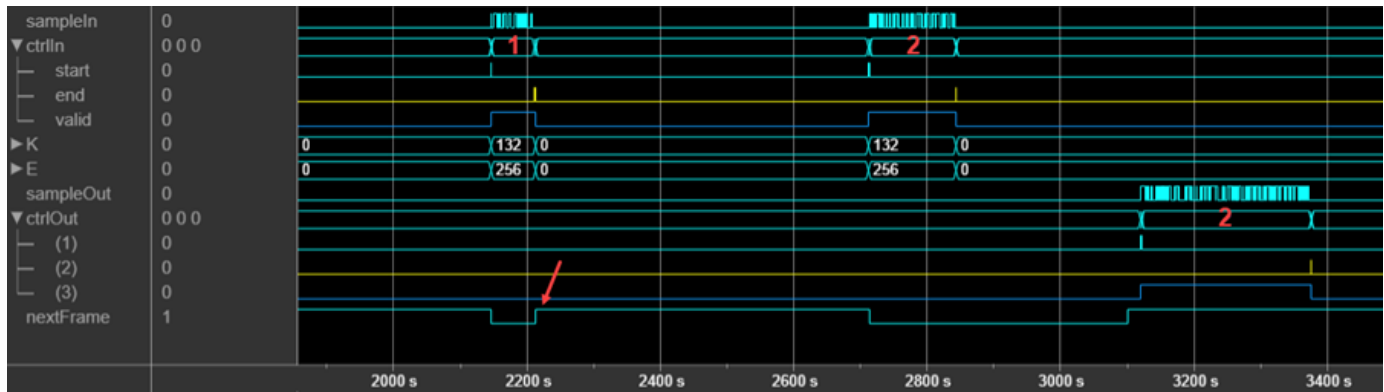
This waveform shows the NR Polar Encoder block processing several frames. The `nextFrame` output signal is 0 when the block is processing data, and 1 when the block is ready to receive the start of a new frame. The cursors show the latency varying with the values of the input `K` and `E` port values. For the first frame with given `K` and `E` values, the block must determine the message length and information bit mapping for those values. This configuration stage means the block needs some time before it is ready to accept the next input frame. For subsequent frames with the same values for `K` and `E`, the block is ready sooner because it does not need to recompute the configuration.



If the block receives an input **start** signal while **nextFrame** is 0, the block discards the frame in progress and begins processing the new data. This waveform shows an NR Polar Encoder input frame (3) applied when **nextFrame** is 0. The block discards the frame in progress (2) and processes the new frame (3) as normal.



If the block receives an invalid input frame, for example, if the frame size is not within the supported range, then the block sets **nextFrame** to 1 one cycle after the input **end** signal. This behavior indicates that the input frame is discarded. This waveform shows an NR Polar Encoder input frame (1) that does not have the correct number of samples expected for the accompanying **K** and **E** values. The waveform shows the **nextFrame** signal set to 1 immediately after the input **end** signal from frame 1. The block discards the frame in progress (1) and processes the new frame (2) as normal.



## See Also

### Blocks

Frame To Samples | Samples To Frame

### Functions

whdlFramesToSamples | whdlSamplesToFrames

## Related Examples

- “Verify Turbo Decoder with Streaming Data from MATLAB”
- “Verify Turbo Decoder with Framed Data from MATLAB”

## Sample Control Bus

Wireless HDL Toolbox blocks use a nonvirtual bus data type, `samplecontrol`, for control signals associated with serial data. The bus contains three `boolean` signals indicating the validity of a sample and the boundaries of the frame. You can easily connect one block to another, because all Wireless HDL Toolbox blocks use this bus for input and output. To convert frames into a sample stream and a `samplecontrol` bus, use the `Frame To Samples` block. This block serializes fixed-size frames. If your frames vary in size, use the `whdlFramesToSamples` function to convert the frames to a data vector in MATLAB, and then import the data into Simulink.

Signal	Description	Data Type
<code>start</code>	<code>true</code> for the first sample in the frame	<code>Boolean</code>
<code>end</code>	<code>true</code> for the last sample in the frame	<code>Boolean</code>
<code>valid</code>	<code>true</code> for any valid sample	<code>Boolean</code>

### Troubleshooting:

When you generate HDL code from a Simulink model that uses this bus, you may need to declare an instance of `samplecontrol` bus in the base workspace. If you encounter the error `Cannot resolve variable 'samplecontrol'` when you generate HDL code in Simulink, use the `samplecontrolbus` function to create an instance of the bus type. Then try generating HDL code again.

To avoid this issue, the Wireless HDL Toolbox model template includes this line in the `InitFcn` callback.

```
evalin('base','samplecontrolbus')
```

You can also call this command from the MATLAB command line.

### See Also

#### Blocks

[Frame To Samples](#) | [Samples To Frame](#)

### More About

- “Streaming Sample Interface” on page 1-2


## Configure the Simulink Environment for Hardware Design

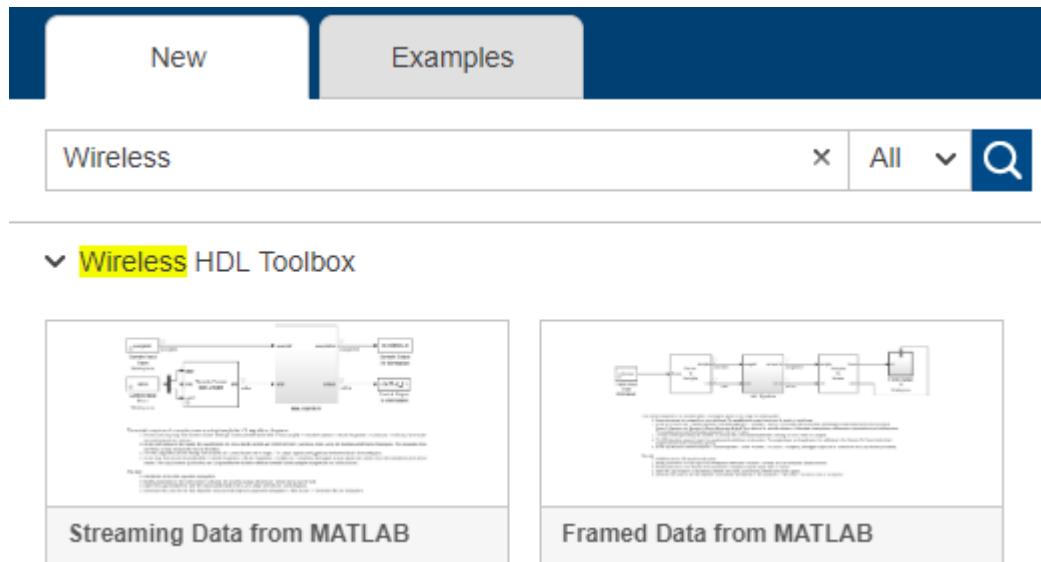
### About Simulink Model Templates

Simulink model templates provide common configuration settings and best practices for new models. Instead of using the default canvas of a new model, select a template model to help you get started.

For more information on Simulink model templates, see “Build and Edit a Model Interactively”.

### Create Model Using Wireless HDL Toolbox Model Template

- 1 Click the Simulink button, , or type `simulink` at the MATLAB command prompt.
- 2 On the Simulink start page, find the Wireless HDL Toolbox section, and click the **Streaming Data from MATLAB** or **Framed Data from MATLAB** template.



A new model, with the template contents and settings, opens in the Simulink Editor. Select **Save** to save the model.

Alternatively, you can create a new model from the template on the command line. For example:

```
new_system my_whdl_Fmodel FromTemplate whdl_framed_data.sltx
open_system my_whdl_Fmodel
```

Or:

```
new_system my_whdl_Smodel FromTemplate whdl_streaming_data.sltx
open_system my_whdl_Smodel
```

## Wireless HDL Toolbox Model Templates

Both Wireless HDL Toolbox model templates include an empty subsystem, HDL Algorithm. This subsystem accepts and returns streaming data and accompanying control signals using the `samplecontrolbus`. You can design an HDL-targeted algorithm within this subsystem.

The templates also configure the model for HDL code generation. Both templates:

- Configure solver settings equivalent to calling `hdlsetup`
- Display data rates and data types in the Model Editor
- Create an instance of `samplecontrolbus` in the workspace (in `InitFcn`)
- Enable `fileIO` mode when generating an HDL test bench

The simulation time, input data, and block parameters are defined in the callback function, `InitFcn`. To view or edit this function, on the **Modeling** tab, expand **Model Settings** and click **Model Properties**, and then on the **Callbacks** tab, click `InitFcn*`.

### Framed Data Template

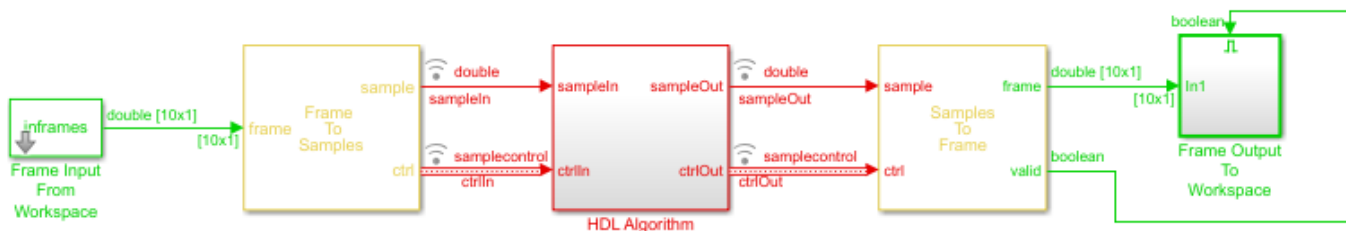
The **Framed Data from MATLAB** template imports framed data from the MATLAB workspace, assuming all frames are the same size. Then, it converts the data to a sample stream by using the `Frame To Samples` block.

The output of the HDL Algorithm subsystem is connected to a `Samples To Frame` block. This block converts the output back to framed data for export to the MATLAB workspace.

The `InitFcn` defines placeholder input frames and settings for the `Frame Input From Workspace`, `Frame To Samples`, and `Samples To Frame` blocks.

The `StopFcn` applies the `valid` signal to the output data and creates a single variable in the workspace.

The model has one data rate for the framed data and a faster data rate for the sample stream. You can display these rates as different colors in the Simulink model.



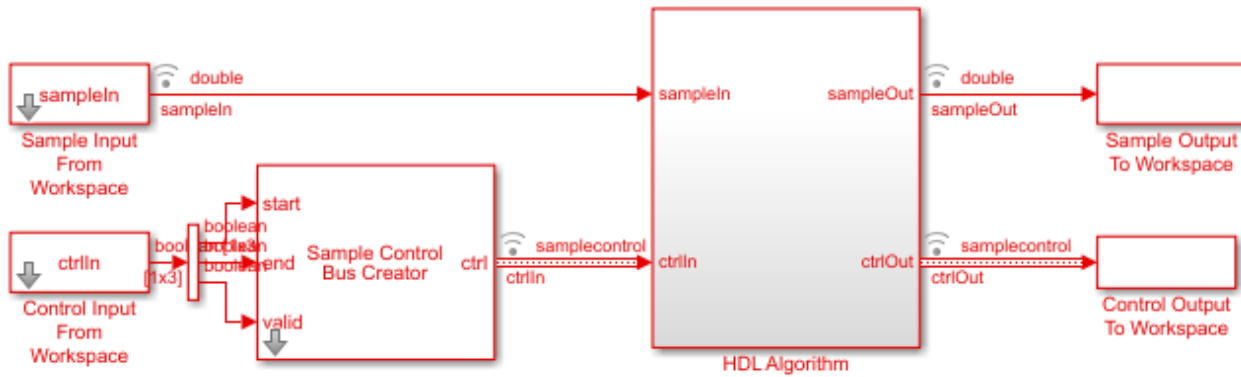
### Streaming Data Template

Use the **Streaming Data from MATLAB** template when your data stream has different-sized frames. The `InitFcn` defines placeholder input frames and uses the `whdlFramesToSamples` function to convert framed data to vectors of data and control signals. The `From Workspace` block imports these variables to the model.

To connect to the HDL Algorithm subsystem and any Wireless HDL Toolbox blocks that you add inside it, the model converts the control signals to the `samplecontrolbus` type, using the Sample Control Bus Creator block.

The model exports the streaming data and control signals back to the MATLAB workspace. The `StopFcn` uses the `whdLSamplesToFrames` function to convert them back to framed data.

The model has a single data rate because all signals in the model represent streaming samples.



## See Also

### Blocks

Frame To Samples | Sample Control Bus Creator | Samples To Frame

### Functions

whdLFramesToSamples | whdLSamplesToFrames

## More About

- “Streaming Sample Interface” on page 1-2



# HDL Code Generation and Verification

---

## HDL Code Generation Support

You can use Simulink for rapid prototyping of hardware designs. Wireless HDL Toolbox blocks, when used with HDL Coder™, support HDL code generation. HDL Coder tools generate target-independent synthesizable Verilog® and VHDL® code for FPGA programming or ASIC prototyping and design.

### HDL Code Generation Support in Wireless HDL Toolbox

Most blocks in Wireless HDL Toolbox support HDL code generation.

The following blocks are for simulation only and are not supported for HDL code generation:

- Frame To Samples
- Samples To Frame
- FIL Frame To Samples
- FIL Samples To Frame

### Other Blocks Supporting HDL Code Generation

Other MathWorks® products also include blocks supported for HDL code generation that you can use to build up your design.

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the **HDL Coder**, **Communications Toolbox HDL Support**, **DSP System Toolbox HDL Support** block libraries, and others.

To create a library of HDL-supported blocks from all your installed products, enter `hdl lib` at the MATLAB command line. This command requires an HDL Coder license.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click **Blocks** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.

Documentation All Examples Functions **Blocks** Apps Search Help

CONTENTS Close

« Documentation Home  
« Blocks

**Category**

**DSP System Toolbox**

- Signal Generation, Manipulation, and Analysis 21
- Filter Implementation 10
- Transforms and Spectral Analysis 3
- Statistics and Linear Algebra 3
- Fixed-Point Design 8

HDL Coder  
HDL Verifier  
LTE HDL Toolbox  
Mixed-Signal Blockset  
SerDes Toolbox  
SimEvents  
Simulink Test

**Extended Capability**

- C/C++ Code Generation 34
- HDL Code Generation** 36
- PLC Code Generation 4
- Fixed-Point Conversion 28

**DSP System Toolbox — Blocks**

By Category | [Alphabetical List](#)

**i** Results are filtered

**Signal Generation, Manipulation, and Analysis**

**Signal Operations**

<a href="#">Downsample</a>	Resample input at lower rate by deleting samples
<a href="#">Repeat</a>	Resample input at higher rate by repeating values
<a href="#">Sample and Hold</a>	Sample and hold input signal
<a href="#">Upsample</a>	Resample input at higher rate by inserting zeros
<a href="#">DC Blocker</a>	Block DC component

**Signal Generation**

<a href="#">Constant</a>	Generate constant value
<a href="#">NCO</a>	Generate real or complex sinusoidal signals
<a href="#">NCO HDL Optimized</a>	Generate real or complex sinusoidal signals—optimized for HDL code generation
<a href="#">Sine Wave</a>	Generate continuous or discrete sine wave

**Scopes and Data Logging**

<a href="#">Spectrum Analyzer</a>	Display frequency spectrum
<a href="#">Time Scope</a>	Display and analyze signals generated during simulation and log signal data to MATLAB
<a href="#">Matrix Viewer</a>	Display matrices as color images
<a href="#">Waterfall</a>	View vectors of data over time
<a href="#">To Workspace</a>	Write data to MATLAB workspace

## Streaming Sample Interface in HDL

The streaming sample control bus data type used by Wireless HDL Toolbox blocks is flattened into separate signals in HDL.

In VHDL, the interface is declared as:

```

PORT( clk           : IN   std_logic;
      reset         : IN   std_logic;
      enb           : IN   std_logic;
      in0           : IN   std_logic_vector(7 DOWNTO 0); -- uint8
      in1_start     : IN   std_logic;
      in1_end       : IN   std_logic;
      in1_valid     : IN   std_logic;
      out0          : OUT  std_logic_vector(7 DOWNTO 0); -- uint8
      out1_start    : OUT  std_logic;
      out1_end      : OUT  std_logic;
      out1_valid    : OUT  std_logic
    );

```

In Verilog, the interface is declared as:

```
input  clk;
input  reset;
input  enb;
input  [7:0] in0; // uint8
input  in1_start;
input  in1_end;
input  in1_valid;
output [7:0] out0; // uint8
output out1_start;
output out1_end;
output out1_valid;
```

### See Also

### More About

- “Streaming Sample Interface” on page 1-2
- “Generate HDL Code” on page 2-5

## Generate HDL Code

You can generate HDL code from subsystems that include blocks supported for HDL code generation, such as the model in “Verify Turbo Decoder with Streaming Data from MATLAB”. In that example, you can generate HDL code from the HDL Algorithm subsystem.

To generate HDL code, you must have an HDL Coder license.

### Prepare Model

Run `hdlsetup` to configure the model for HDL code generation. If you started your design using the Wireless HDL Toolbox Simulink model template, your model is already configured for HDL code generation.

### Generate HDL Code

Right-click the HDL Algorithm subsystem, and select **HDL Code > Generate HDL for Subsystem** to generate HDL using the default settings. The output log of this operation is shown in the MATLAB Command Window, along with the location of the generated files.

To change code generation options, use the **HDL Code Generation** panes of the Simulink Configuration Parameters dialog box. For guidance through the HDL code generation process, or to select a target device or synthesis tool, right-click the HDL Algorithm subsystem, and select **HDL Code > HDL Workflow Advisor**.

Alternatively, from the MATLAB Command Window, you can call:

```
makehdl([modelName '/HDL Algorithm'])
```

### Generate HDL Test Bench

You can select options to generate a test bench in the Simulink Configuration Parameters dialog box or in the HDL Workflow Advisor.

Alternatively, to generate an HDL test bench from the command line, call:

```
makehdltb([modelName '/HDL Algorithm'])
```

### See Also

#### Functions

`makehdl` | `makehdltb`

### Related Examples

- “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder)
- “Choose a Test Bench for Generated HDL Code” (HDL Coder)

## FPGA-in-the-Loop

FPGA-in-the-loop (FIL) enables you to run a Simulink simulation that is synchronized with an HDL design running on an Intel® or Xilinx® FPGA board. This link between the simulator and the board enables you to verify HDL implementations directly against Simulink or MATLAB algorithms. You can apply real-world data and test scenarios from these algorithms to the HDL design on the FPGA.

When simulating Wireless HDL Toolbox blocks, you must use a streaming sample interface. Streaming sample data, while required for hardware implementations of communications systems, is time-consuming at the FPGA-in-the-loop interface with Simulink.

You can convert from frames to samples and samples to frames either in Simulink or in MATLAB. Depending on your workflow, you can optimize your FPGA-in-the-loop simulation in one of two ways.

One workflow is a Simulink model that imports framed data from MATLAB. This type of model then uses the Frame To Samples and Samples To Frame blocks to convert the data format. For FPGA-in-the-loop, replace these conversion blocks with FIL Frame To Samples and FIL Samples To Frame blocks.

The other workflow is a Simulink model that imports streaming data from MATLAB. This type of model goes with a MATLAB script that uses the `ltehdlFrameToSamples` and `ltehdlSamplesToFrames` functions. For FPGA-in-the-loop, modify your script and Simulink model so that they pass vectors of data to the FPGA-in-the-loop interface.

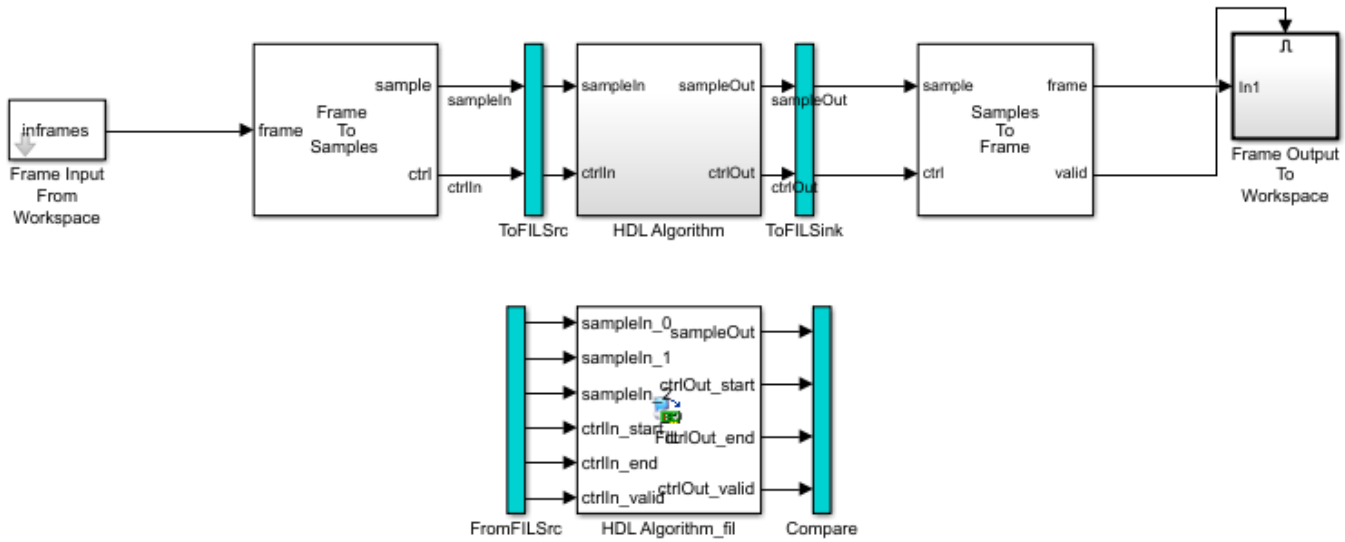
When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Wireless HDL Toolbox designs, the FIL block in that model replicates the sample-streaming interface and sends one sample at a time to the FPGA. Both these modifications construct vectors that make more efficient use of the interface between the Simulink model and the FPGA board.

The instructions that follow show how to modify FPGA-in-the-loop models for the “Verify Turbo Decoder with Streaming Data from MATLAB” and “Verify Turbo Decoder with Framed Data from MATLAB” workflow examples.

### FIL Workflow: Framed Data from MATLAB

#### Autogenerated FIL Model

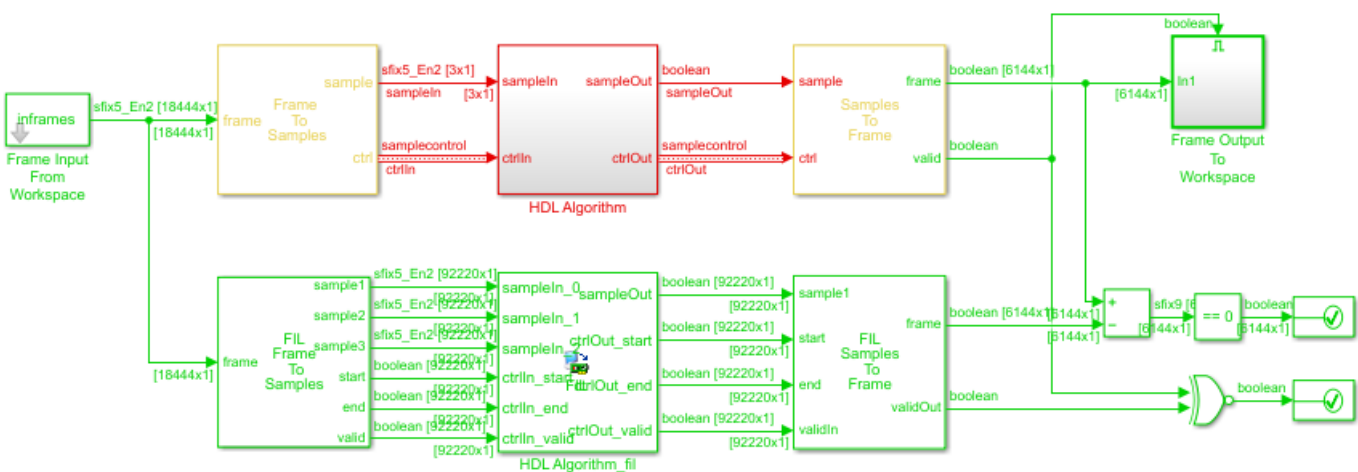
The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in Simulink. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.



The blue ToFILSrc subsystem branches the sample-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The blue ToFILSink subsystem branches the sample-stream output of the HDL Algorithm block into the Compare subsystem, where it is compared with the output of the HDL Algorithm\_fil block. This setup is slow because the model sends only a single sample, and its associated control signals, in each packet to and from the FPGA board.

### Modified FIL Model

To improve the communication bandwidth with the FPGA board, modify the autogenerated model. The modified model uses the FIL Frame To Samples and FIL Samples To Frame blocks to send one frame at a time.



To create this modified FIL model:

- 1 Remove the blue subsystems, and create a branch at the **frame** input port of the Frame To Samples block.

- 2 Insert the FIL Frame To Samples block before the HDL Algorithm\_fil block. Insert the FIL Samples To Frame block after the HDL Algorithm\_fil block.
- 3 Set the **Output frame size** on the FIL block to the input frame size.

### Runtime Options

Overclocking factor:	<input type="text" value="1"/>
Output frame size:	<input type="text" value="inframesize"/>

- 4 In the FIL Frame To Samples and FIL Samples To Frame blocks, set the parameters to match the settings of the Frame To Samples and Samples To Frame blocks.
- 5 Branch the frame output of the Samples To Frame block for comparison. You can compare the entire frame at once with a Diff block. Compare the `validOut` signals using an XOR block.

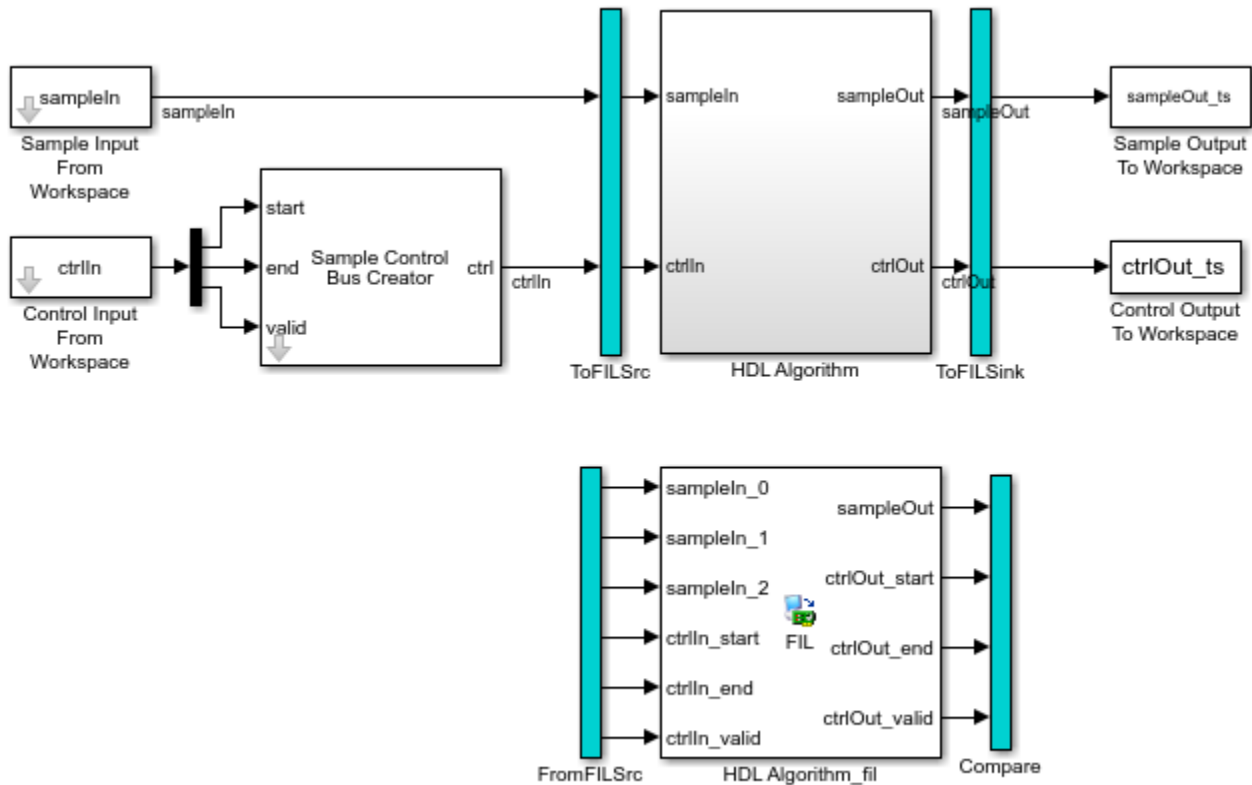
The input size at the FIL block is the frame size from the input data frames. The vector size of the FIL block ports does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board. This modified model sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

## FIL Workflow: Streaming Data from MATLAB

### Autogenerated FIL Model

The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in MATLAB. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.

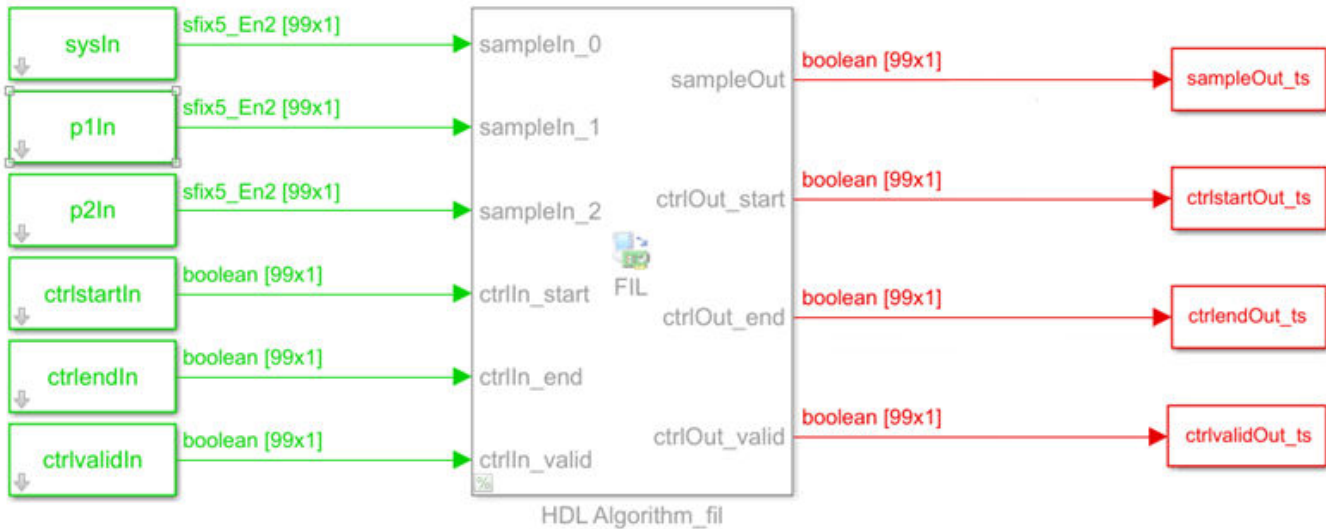




The blue ToFILSrc subsystem branches the sample-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The blue ToFILSink subsystem branches the sample-stream output of the HDL Algorithm block into the Compare subsystem, where it is compared with the output of the HDL Algorithm\_fil block. This setup is slow because the model sends only a single sample, and its associated control signals, in each packet to and from the FPGA board.

### Modified FIL Model

To improve the communication bandwidth with the FPGA board, use the generated FIL block in a different model. The alternate model imports and exports vectors of flattened data. The accompanying MATLAB script reshapes the input and output data, and verifies the FIL output against a behavioral model. Reshaping the data in MATLAB is easier and the simulation is faster than reshaping in Simulink.



First, modify the accompanying MATLAB script:

- 1 Pick a frame size for the FIL simulation. This size does not have to match the actual frame sizes in the generated data. It can contain your entire data set. The FIL block divides the data into maximum size packets for communication with the FPGA board.

```
filframesize = 99;
```

- 2 Combine the cell array of input frames into one matrix.

```
allframes = [inframes{:}];
```

- 3 Flatten the samples and control signals so there is one vector for each input port on the FIL block. This model includes the LTE Turbo Decoder block, so the input samples consist of three values.

```
sysIn = allframes(1:3:end);
p1In = allframes(2:3:end);
p2In = allframes(3:3:end);
```

```
ctrlstartIn = ctrlIn(1:3:end);
ctrlendIn = ctrlIn(2:3:end);
ctrlvalidIn = ctrlIn(3:3:end);
```

- 4 Call the FIL model.

```
simTime = size(allframes,1);
modelname = 'TurboDecoderStreamingFILVectortoSL';
open_system(modelname);
sim(modelname);
```

- 5 Reshape the output variables for input to the whdlSamplesToFrames function. Recreate an  $N$ -by-3 control signal matrix and a vector of sample data. In this example, the output sample is a single value. If the output sample is multiple values, build an  $N$ -by-SampleSize sample matrix.

```
sampleOut = squeeze(sampleOut_ts.Data);
ctrlOut = [squeeze(ctrlstartOut_ts.Data) ...
```

```
squeeze(ctrlendOut_ts.Data) ...
squeeze(ctrlvalidOut_ts.Data)];
```

Then, create a Simulink model:

- 1 Copy the generated FIL block into a new model.
- 2 Configure and connect a Signal From Workspace block for each input port on the FIL block. Use the variables from your MATLAB script as the parameter values.

Parameters

Signal:

Sample time:

Samples per frame:

- 3 Set the **Output frame size** on the FIL block to the desired FIL frame size.

Runtime Options

Overclocking factor:

Output frame size:

- 4 Configure and connect a To Workspace block for each output port of the FIL block.

The input size at the FIL block is the frame size you specify on the Signal To Workspace blocks. The vector size of the FIL block ports does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board. This modified model sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

## See Also

## More About

- “Verify Turbo Decoder with Streaming Data from MATLAB”
- “Verify Turbo Decoder with Framed Data from MATLAB”

## Prototype LTE Algorithms on Hardware

The Communications Toolbox™ Support Package for Xilinx Zynq-Based Radio enables you to design, prototype, and verify practical wireless communications systems on Xilinx Zynq-based radio hardware.

- Use the Xilinx Zynq-based radio as an I/O peripheral to transmit and receive real-time arbitrary waveforms using MATLAB System objects or Simulink blocks.
- Transmit and receive RF signals out of the box, enabling quick testing of SDR designs under real-world conditions.
- Transmit and receive data on one or two channels.
- Configure RF radio settings easily.
- Acquire high-bandwidth signals by using burst mode.
- In Simulink, customize and prototype SDR algorithms. Target only the FPGA fabric of the device, or deploy partitioned hardware-software co-design implementations across the ARM® processor and the FPGA fabric of the device (Windows® operating system only).
- Run application examples to get started.

The support package provides two workflows:

- FPGA-only targeting - This workflow uses generated HDL code from HDL Coder and HDL Coder Support Package for Xilinx Zynq Platform.
- Hardware-software co-design - This workflow also uses HDL Coder and HDL Coder Support Package for Xilinx Zynq Platform. It additionally requires Simulink Coder™, Embedded Coder®, and Embedded Coder Support Package for Xilinx Zynq Platform.

The “LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) support package example shows how to use the hardware-software co-design workflow to deploy the design from “LTE HDL MIB Recovery” on page 5-80 to a hardware board with a radio daughter card. The “LTE Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) support package example shows how to capture live LTE data for use in testing your designs.

## How to Install Support Packages

A support package is an add-on that enables you to use a MathWorks product with specific third-party hardware and software. Support packages use the license of the base product. For instance, Communications Toolbox Support Package for Xilinx Zynq-Based Radio requires a license for Communications Toolbox.

Install support packages using the MATLAB **Add-Ons** menu. You can also use the **Add-Ons** menu to update installed support package software or update the firmware on third-party hardware.

To install support packages, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. You can filter this list by selecting categories (such as hardware vendor or application area), or by performing a keyword search.

Search the **Add-Ons** list for Zynq, and install these support packages:

- Communications Toolbox Support Package for Xilinx Zynq-Based Radio

- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform (only needed for hardware-software co-design)

When the support package installation is complete, you must set up the host computer and radio hardware. For Windows systems, the installer provides guided setup steps. For Linux® systems, the installer links to manual setup instructions.

## Design Requirements

The Communications Toolbox Support Package for Xilinx Zynq-Based Radio provides a reference design that you can use to create an IP core that integrates into the radio hardware. Use the HDL Workflow Advisor to guide you through generating a shareable and reusable IP core module using the reference design.

To work with the reference design, your FPGA targeted design must use a streaming data interface with a control signal that indicates the validity of each sample. Wireless HDL Toolbox blocks provide this interface. Use the Sample Control Bus Selector block to separate the valid control signal from the bus.

To deploy a design using the support package, your design must meet these preconditions.

- Each data input or output must be 16 bits. The HDL subsystem that fits into the reference design does not support complex signals at the ports. To handle complex inputs and outputs, model separate I and Q ports at the subsystem boundaries.
- Model all the ports for a given reference design, even when the ports are not used.
- In Simulink, the input and output data and valid signals must be driven at the same sample rate. Therefore, the input and output clock rates of the subsystem must be equal.
- Clock the data and valid signals at the fastest rate of the HDL subsystem.
- For the FPGA-only targeting workflow:
  - Duplex operation is not supported. Use either the transmit or the receive operation, but not both.
- For the hardware-software co-design workflow:
  - Duplex operation is supported. You can use both the Transmitter and Receiver blocks in the same design.
  - AXI4-Lite register ports can be clocked at arbitrary rates.
  - In single-channel mode, you can transmit or receive data frames containing an even number of samples only. If you use an odd number of samples, the software inserts a zero sample at the end of each frame.

The real-time design encounters a larger volume of data and a larger set of state progressions than you can simulate in Simulink. Make sure to model and generate control logic to handle the restart between subframes. Consider adding extra subsystem ports for debug visibility of these extended states once the design is deployed to the board.

## Design for Debugging

Once the design is deployed to the board, you have much less visibility of the internal signals in your design. To improve visibility, you can add temporary output ports to your subsystem before you

generate your IP core. Signals that can help with debugging are design state, mux select signals or other control parameters, and data values at intermediate stages of the data path. You can also add input ports and muxes to give the option for external control of parameters such as mux select signals and gain values.

When you simulate the design on the board in External mode, you can drive and view these ports from Simulink. The Xilinx Zynq AXI Interface block from the generated software model provides a Simulink interface to the input and output ports of your design while it is running on the board.

Once you are confident that your design is behaving as intended, you can remove these ports and regenerate the IP core.

Another debugging strategy is to include a known input signal stored in memory on the FPGA. This memory can be part of the generated HDL code from your Simulink model. The “LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) support package example shows an input port `externalDataSel` that provides a switch between a stored data set and the live data from the radio.

## See Also

### More About

- “Communications Toolbox Support Package for Xilinx Zynq-Based Radio”
- “FPGA Targeting Workflow” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)
- “Hardware-Software Co-Design Workflow” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)
- “LTE HDL MIB Recovery” on page 5-80
- “LTE HDL SIB1 Recovery” on page 5-63

# Reference Page Examples

---

## Append CRC Checksum to Streaming Data

This example shows how to use the LTE CRC Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

- 1 Generate frames of random input samples in MATLAB.
- 2 Generate and append a CRC checksum using the LTE Toolbox function `lteCRCEncode`.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 4 To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE CRC Encoder.
- 5 Export the stream of bits, which now has an appended CRC checksum, to the MATLAB® workspace.
- 6 Convert the sample stream back to framed data, and compare the frames with the reference frames and checksum.

Generate input data frames. Generate reference output data using `lteCRCEncode`.

```
frameLength = 256;
numframes   = 2;
rng(0);

txframes    = cell(1,numframes);
txcodeword  = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for ii = 1:numframes
    txframes{ii} = randi([0 1],frameLength,1)>0.5;

    CRCType = '24B';
    CRCMask = 50;
    txcodeword{ii} = lteCRCEncode(txframes{ii},CRCType,CRCMask);
end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully encoded before the next one starts. For CRC 24 encoding, the checksum adds 24 parity bits at the end of the frame. The hardware-friendly algorithm also adds `CRCLength + 3` cycles of latency.

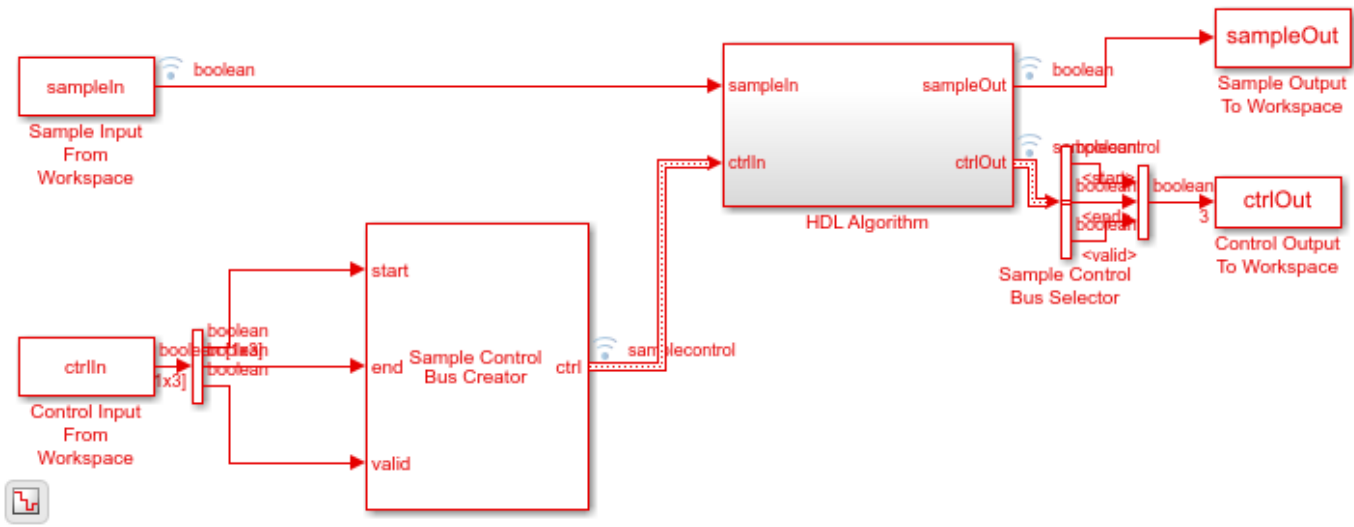
```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = 24+27;
outputSize                = 1;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txframes,idleCyclesBetweenSamples,idleCyclesBetweenFrames,outputSize);
```

Run the Simulink model.

```
sampletime = 1;
simTime = length(ctrlIn);
modelName = 'ltehdlCRCEncoderModel';
open(modelName);
sim(modelName);
```





The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the reference data.

```
txhdlframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE CRC Encoder\n');
for ii = 1:numframes
    numBitsDiff = sum(double(txcodeword{ii})-double(txhdlframes{ii}));
    fprintf([' Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'], ii, numBitsDiff);
end
```

Maximum frame size computed to be 280 samples.

```
LTE CRC Encoder
Frame 1: Behavioral and HDL simulation differ by 0 bits
Frame 2: Behavioral and HDL simulation differ by 0 bits
```

## See Also

### Blocks

LTE CRC Encoder

### Functions

lteCRCEncode

## More About

- “Check for CRC Errors in Streaming Samples” on page 3-4

## Check for CRC Errors in Streaming Samples

This example shows how to use the LTE CRC Decoder block to check encoded data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

- 1 Generate frames of random input samples in MATLAB.
- 2 Generate and append the CRC checksum using the LTE Toolbox function `lteCRCEncode`.
- 3 Convert framed input data and checksum to a stream of samples and import it to Simulink®.
- 4 To check the samples against the checksum using a hardware-friendly architecture, run the Simulink model. The model contains the Wireless HDL Toolbox™ block LTE CRC Decoder.
- 5 Export the stream of samples back to the MATLAB® workspace.
- 6 Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames, then generate the CRC checksum using `lteCRCEncode`.

```
frameLength = 256;
numframes   = 2;
rng(0);

txframes    = cell(1,numframes);
txcodeword  = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for ii = 1:numframes

    txframes{ii} = randi([0 1],frameLength,1)>0.5;

    CRCType = '24B';
    CRCMask = 50;
    txcodeword{ii} = boolean(lteCRCEncode(txframes{ii},CRCType,CRCMask));

end
```

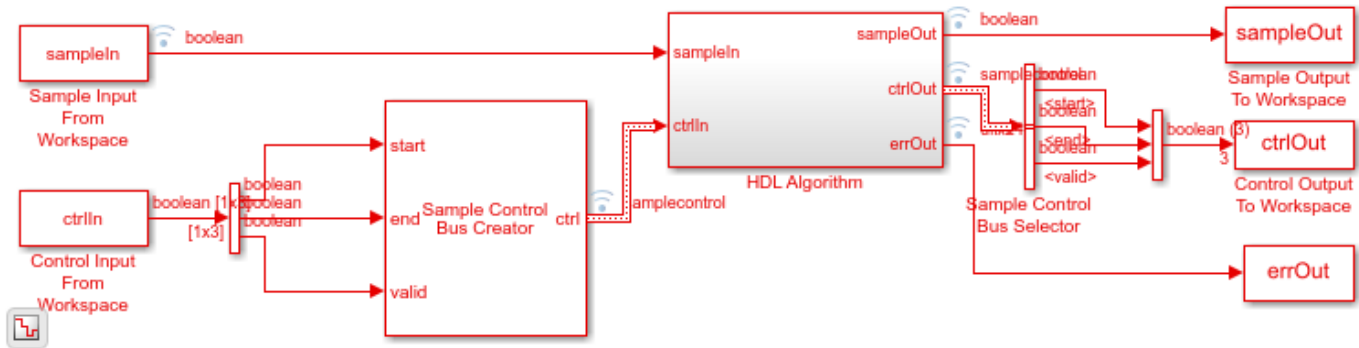
Serialize input data for the Simulink model. The LTE CRC Decoder block does not require any space between frames, but the hardware-friendly algorithm adds latency of  $(3 * CRCLength / SampleSize) + 5$  cycles. This example uses scalar input samples, so the latency is  $(3 * CRCLength) + 5$ .

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = 77;
samplesizeIn             = 1;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txcodeword,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesizeIn);
```

Run the Simulink model.

```
sampletime = 1;
simTime = length(ctrlIn);
modelName = 'ltehdlCRCDecoderModel';
open_system(modelName);
sim(modelName);
```



The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the input frames.

```
txhdlframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE CRC Decoder\n');
for ii = 1:numframes
    numBitsDiff = sum(double(txframes{ii})-double(txhdlframes{ii}));
    fprintf([' Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'], ii, numBitsDiff);
end
```

Maximum frame size computed to be 256 samples.

LTE CRC Decoder

```
Frame 1: Behavioral and HDL simulation differ by 0 bits
Frame 2: Behavioral and HDL simulation differ by 0 bits
```

## See Also

### Blocks

LTE CRC Decoder

### Functions

`lteCRCDecode`

## More About

- “Append CRC Checksum to Streaming Data” on page 3-2

## Turbo Encode Streaming Samples

This example shows how to use the LTE Turbo Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

- 1 Generate frames of random input samples in MATLAB®.
- 2 Encode the data using the LTE Toolbox function `lteTurboEncode`.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 4 To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE Turbo Encoder.
- 5 Export the stream of encoded samples to the MATLAB workspace.
- 6 Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames. Generate reference encoded data using `lteTurboEncode`.

```
rng(0);
turboframesize = 40;
numframes = 2;

txBits = cell(1,numframes);
codedData = cell(1,numframes);

for ii = 1:numframes
    txBits{ii} = logical(randi([0 1],turboframesize,1));
    codedData{ii} = lteTurboEncode(txBits{ii});
end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully encoded before the next one starts. The LTE Turbo Encoder block takes `inframesize + 16` cycles to complete encoding of a frame.

```
inframes = txBits;

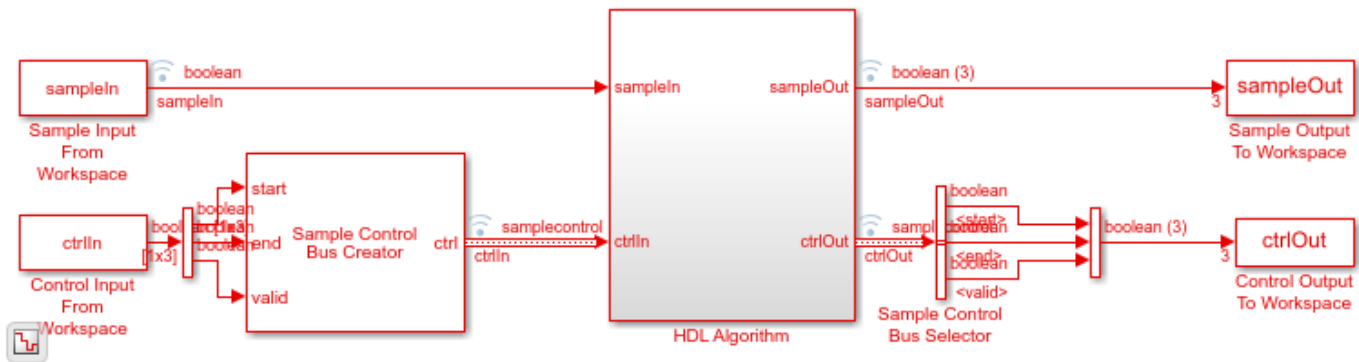
inframesize = size(inframes{1},1);

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = inframesize+16;

[sampleIn,ctrlIn] = ...
    whdlFramesToSamples(inframes, ...
                        idlecyclesbetweensamples, ...
                        idlecyclesbetweenframes);
```

Run the Simulink model. The simulation time equals the number of input samples. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete encoding of both frames.

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrlIn,1);
modelname = 'ltehdlTurboEncoderModel';
open_system(modelname);
sim(modelname);
```



The Simulink model exports `sampleOut_ts` and `ctrlOut_ts` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the reference encoded frames.

The output samples of the LTE Turbo Encoder block are interleaved with the parity bits.

Hardware-friendly output: `S_1 P1_1 P2_1 S2 P1_2 P2_2 ... Sn P1_n P2_n`

LTE Toolbox output: `S_1 S_2 ... S_n P1_1 P1_2 ... P1_n P2_1 P2_2 ... P2_n`

Reorder the samples using the `interleave` option of the `whdlSamplesToFrames` function. Compare the reordered output frames with the reference encoded frames.

```
sampleOut = sampleOut';
interleaveSamples = true;
outframes = whdlSamplesToFrames(sampleOut(:),ctrlOut,[],interleaveSamples);
```

```
fprintf('\nLTE Turbo Encoder\n');
for ii = 1:numframes
    numBitsDiff = sum(outframes{ii} ~= codedData{ii});
    fprintf([' Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'],ii,numBitsDiff);
end
```

Maximum frame size computed to be 132 samples.

```
LTE Turbo Encoder
Frame 1: Behavioral and HDL simulation differ by 0 bits
Frame 2: Behavioral and HDL simulation differ by 0 bits
```

## See Also

### Blocks

LTE Turbo Encoder

### Functions

`lteTurboEncode`

## More About

- “Turbo Decode Streaming Samples” on page 3-8

## Turbo Decode Streaming Samples

This example shows how to use the **LTE Turbo Decoder** block to decode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™.

- 1 Generate frames of random input samples in MATLAB®. Encode the samples and add noise to the data.
- 2 Decode the data using the LTE Toolbox function, `lteTurboDecode`.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 4 To decode the samples using a hardware-friendly architecture, execute the Simulink model, which contains the **LTE Turbo Decoder** block.
- 5 Export the stream of decoded bits to the MATLAB workspace.
- 6 Convert the sample stream back to framed data, and compare the frames with the decoded frames from Step 2.

Generate input data frames. Turbo encode the data, modulate the message, and add noise to the resulting constellation. Demodulate the noisy constellation and generate soft bit values. Generate reference decoded data using `lteTurboDecode`. For the hardware-friendly model, convert the soft bits into a fixed-point data type.

```
rng(0);
numframes = 2;

txBits = cell(1,numframes);
softBits = cell(1,numframes);
rxBits = cell(1,numframes);
inframes = cell(1,numframes);

for ii = 1:numframes
    txBits{ii} = randi([0 1],6144,1);
    codedData = lteTurboEncode(txBits{ii});
    txSymbols = lteSymbolModulate(codedData,'QPSK');
    noise = 0.5*complex(randn(size(txSymbols)),randn(size(txSymbols)));
    rxSymbols = txSymbols + noise;
    softBits{ii} = lteSymbolDemodulate(rxSymbols,'QPSK','Soft');
    rxBits{ii} = lteTurboDecode(softBits{ii});
    inframes{ii} = fi(softBits{ii},1,5,2);
end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully decoded before the next one starts. The **LTE Turbo Decoder** block takes  $2 * \text{numTurboIterations} * \text{HalfIterationLatency} + (\text{inframesize} / \text{samplesizeIn})$  cycles to complete decoding of a frame. For details of the *HalfIterationLatency* calculation see the Turbo Decoder block reference page.

The **LTE Turbo Decoder** block expects input samples are interleaved with the parity bits.

Hardware-friendly input: S\_1 P1\_1 P2\_1 S2 P1\_2 P2\_2 ... Sn P1\_n P2\_n

LTE Toolbox input: S\_1 S\_2 ... S\_n P1\_1 P1\_2 ... P1\_n P2\_1 P2\_2 ... P2\_n

Reorder the samples using the `interleave` option of the `whdlFramesToSamples` function.

```
inframesize = size(inframes{1},1); %includes 4 tail bit samples
encoderrate = 3; % rate 1/3 Turbo code
```

```

samplesizeIn = encoderrate; % 3 samples in at a time

idlecyclesbetweensamples = 0;
outframesize = size(txBits{1},1);
numTurboIterations = 6;
halfIterationLatency = (ceil(outframesize/32)+3)*32; % window size=32
algframedelay = 2*numTurboIterations*halfIterationLatency+(inframesize/samplesizeIn);
idlecyclesbetweenframes = algframedelay;

interleaveSamples = true;
[sampleIn,ctrlIn] = ...
    whdlFramesToSamples(inframes, ...
        idlecyclesbetweensamples, ...
        idlecyclesbetweenframes, ...
        samplesizeIn, ...
        interleaveSamples);

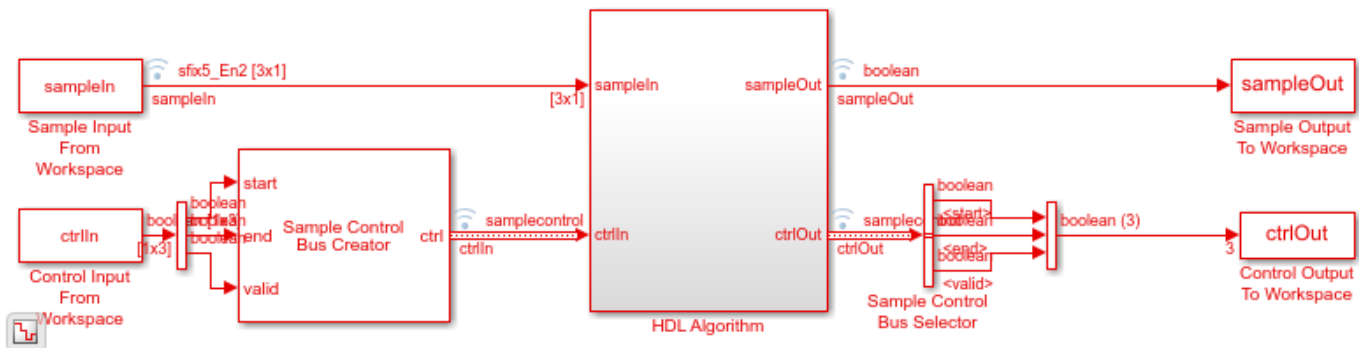
```

Run the Simulink model. The simulation time equals the number of input samples. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete decoding of both frames.

```

sampletime = 1;
simTime = size(ctrlIn, 1);
modelname = 'ltehdlTurboDecoderModel';
open_system(modelname);
sim(modelname);

```



The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. De-serialize the output samples, and compare to the decoded frame.

```

outframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE Turbo Decoder\n');
for ii = 1:numframes
    numBitsDiff = sum(outframes{ii} ~= rxBits{ii});
    fprintf([' Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,numBitsDiff);
end

```

Maximum frame size computed to be 6144 samples.

LTE Turbo Decoder

```
Frame 1: Behavioral and HDL simulation differ by 0 bits  
Frame 2: Behavioral and HDL simulation differ by 0 bits
```

### See Also

#### Blocks

LTE Turbo Decoder

#### Functions

lteTurboDecode

### More About

- “Turbo Encode Streaming Samples” on page 3-6



## Convolutional Encode of Streaming Samples

This example shows how to use the LTE Convolutional Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

- 1 Generate frames of random input samples in MATLAB®.
- 2 Encode the data using the LTE Toolbox function `lteConvolutionalEncode`.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 4 To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE Convolutional Encoder.
- 5 Export the stream of encoded bits to the MATLAB workspace.
- 6 Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames. Generate reference encoded data using `lteConvolutionalEncode`.

```
rng(0);
frameLength = 256;
numframes = 2;

txframes = cell(1,numframes);
txcodeword = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for k = 1:numframes
    txframes{k} = randi([0 1],frameLength,1)>0.5;
    txcodeword{k} = lteConvolutionalEncode(txframes{k});
end
```

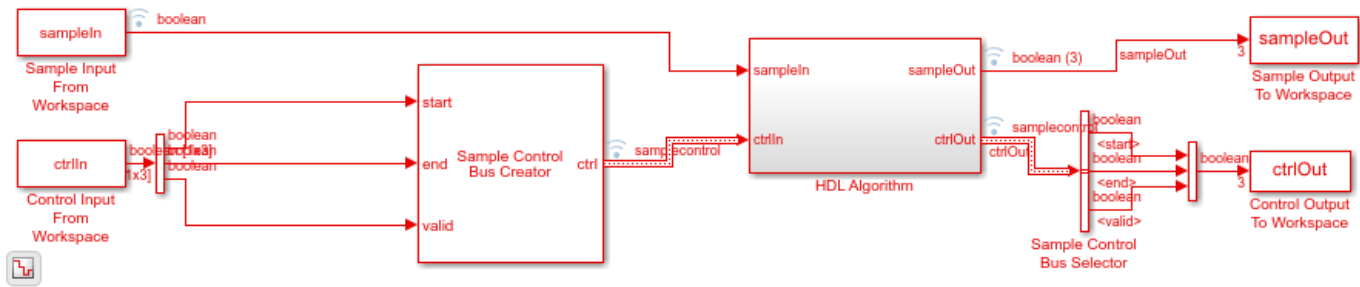
Serialize input data for the Simulink model. Leave enough time between frames so that each frame is fully encoded before the next one starts. The block takes `frameLength + 5` cycles to encode the frame.

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames = frameLength+5;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txframes,idleCyclesBetweenSamples,idleCyclesBetweenFrames);
```

Run the Simulink model. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete encoding of both frames.

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrlIn,1);
modelName = 'ltehdlConvolutionalEncoderModel';
open_system(modelName);
sim(modelName);
```



The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare them to the encoded frame.

The output samples of the LTE Convolutional Encoder block are the interleaved results of the three polynomials.

- Hardware-friendly output:  $G_{0\_1} G_{1\_1} G_{2\_1} G_{0\_2} G_{1\_2} G_{2\_2} \dots G_n G_{1\_n} G_{2\_n}$
- LTE Toolbox output:  $G_{0\_1} G_{0\_2} \dots G_{0\_n} G_{1\_1} G_{1\_2} \dots G_{1\_n} G_{2\_1} G_{2\_2} \dots G_{2\_n}$

The `whdlSamplesToFrames` function provides an option to reorder the samples. Compare the reordered output frames with the reference encoded frames.

```
interleaveSamples = true;
sampleOut = sampleOut';
txhdlframes = whdlSamplesToFrames(sampleOut(:),ctrlOut,[],interleaveSamples);
```

```
fprintf('\nLTE Convolutional Encoder\n');
for k = 1:numframes
    numBitsDiff = sum(double(txcodeword{k})-double(txhdlframes{k}));
    fprintf([' Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'],k,numBitsDiff);
end
```

Maximum frame size computed to be 768 samples.

```
LTE Convolutional Encoder
Frame 1: Behavioral and HDL simulation differ by 0 bits
Frame 2: Behavioral and HDL simulation differ by 0 bits
```

## See Also

### Blocks

LTE Convolutional Encoder

### Functions

`lteConvolutionalEncode`

## More About

- “Convolutional Decode of Streaming Samples” on page 3-13

## Convolutional Decode of Streaming Samples

This example shows how to use the LTE Convolutional Decoder block to decode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

- 1 Generate LTE convolutionally encoded messages in MATLAB®, using LTE Toolbox.
- 2 Call Communications Toolbox™ functions to perform BPSK modulation, transmission through an AWGN channel, and BPSK demodulation. The result is soft-bit values that represent log-likelihood ratios (LLRs).
- 3 Quantize the soft bits according to the signal-to-noise ration (SNR).
- 4 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 5 To decode the samples using a hardware-friendly architecture, execute the Simulink model, which contains the LTE Convolutional Decoder block.
- 6 Export the stream of decoded bits to the MATLAB workspace.
- 7 Convert the sample stream back to framed data, and compare the frames with the original input frames.

Calculate the channel SNR and create the modulator, channel, and demodulator System objects.  $E_bN_0$  is the ratio of energy per uncoded bit to noise spectral density, in dB.  $E_cN_0$  is the ratio of energy per channel bit to noise spectral density, in dB. The code rate of the convolutional encoder is 1/3. Therefore each transmitted bit contains 1/3 of a bit of information.

```
EbNo = 10;
EcNo = EbNo - 10*log10(3);

modulator = comm.BPSKModulator;
channel = comm.AWGNChannel('EbNo',EcNo);
demodulator = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio');
```

Generate input data frames. Encode the data, modulate the message, and add channel effects to the resulting constellation. Demodulate the transmitted constellation and generate soft-bit values. For the hardware-friendly model, convert the soft bits into a fixed-point data type. The optimal soft-bit quantization step size is a function of the noise spectral density,  $N_0$ .

```
rng(0);
messageLength = 100;
numframes = 2;
numSoftBits = 5;

txMessages = cell(1,numframes);
rxSoftMessages = cell(1,numframes);

No = 10^((-EcNo)/10);
quantStepSize = sqrt(No/2^numSoftBits);

for k = 1:numframes

    txMessages{k} = randi([0 1],messageLength,1,'int8');
    txCodeword = lteConvolutionalEncode(txMessages{k});

    modOut = modulator.step(txCodeword);
    chanOut = channel.step(modOut);
```

```

demodOut = -demodulator.step(chanOut)/4;

rxSoftMessagesDouble = demodOut./quantStepSize;
rxSoftMessages{k} = fi(rxSoftMessagesDouble,1,numSoftBits,0);

```

end

Serialize input data for the Simulink model. Leave enough time between frames so that each frame is fully decoded before the next one starts. The LTE Convolutional Decoder block takes (2 \* messageLength) + 140 cycles to complete decoding of a frame.

The LTE Convolutional Decoder block expects the input data to contain the three encoded bits interleaved.

- Hardware-friendly input: G0\_1 G1\_1 G2\_1 G0\_2 G1\_2 G2\_2 ... G0\_n G1\_n G2\_n
- LTE Toolbox input: G0\_1 G0\_2 ... G0\_n G1\_1 G1\_2 ... G1\_n G2\_1 G2\_2 ... G2\_n

```

idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames = 2 * messageLength + 140;
samplesizeIn = 3;
interleaveSamples = true;

```

```

[sampleIn,ctrlIn] = whdlFramesToSamples(rxSoftMessages,...
    idleCyclesBetweenSamples,...
    idleCyclesBetweenFrames,...
    samplesizeIn,...
    interleaveSamples);

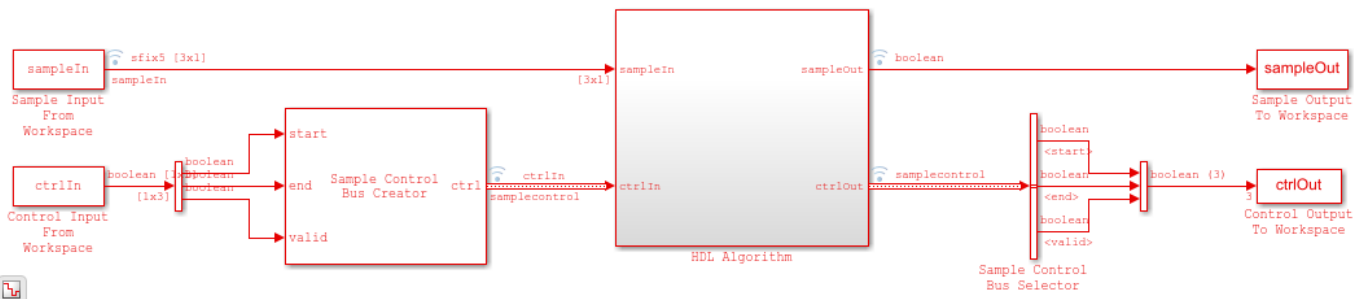
```

Run the Simulink model. Because of the added idle cycles between frames, the streaming input variables include enough cycles for the model to complete decoding of both frames.

```

sampletime= 1;
simTime = size(ctrlIn,1);
modelname = 'ltehdlConvolutionalDecoderModel';
open(modelname);
sim(modelname);

```



The Simulink model exports sampleOut and ctrlOut back to the MATLAB workspace. Deserialize the output samples, and compare to the decoded frame.

```

rxMessages = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE Convolutional Decoder\n');
for k = 1:numframes
    numBitsDiff = sum(double(txMessages{k})-double(rxMessages{k}));

```

```
    fprintf([' Frame %d: Behavioral and ' ...  
            'HDL simulation differ by %d bits\n'], k, numBitsDiff);  
end
```

Maximum frame size computed to be 100 samples.

LTE Convolutional Decoder

Frame 1: Behavioral and HDL simulation differ by 0 bits

Frame 2: Behavioral and HDL simulation differ by 0 bits

## See Also

### Blocks

LTE Convolutional Decoder

### Functions

lteConvolutionalDecode

## More About

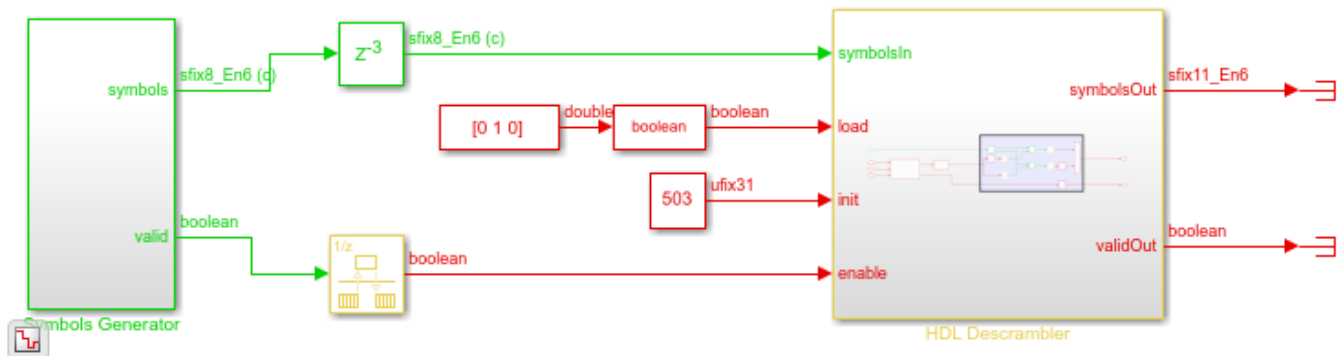
- “Convolutional Encode of Streaming Samples” on page 3-11

## Descrambling with Gold Sequence Generator

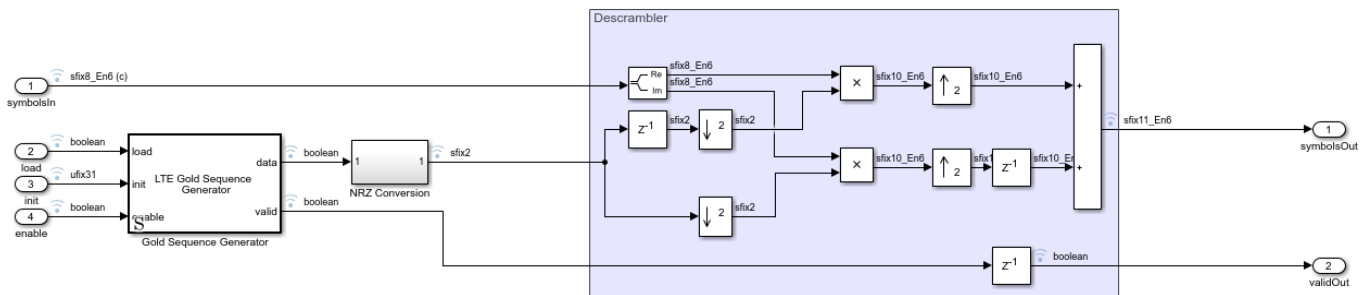
This example shows how to use the LTE Gold Sequence Generator block to implement an LTE descrambler.

The example model generates random I-Q pairs, multiplies the I and Q components with a generated Gold sequence, and interleaves the I and Q into a single data stream.

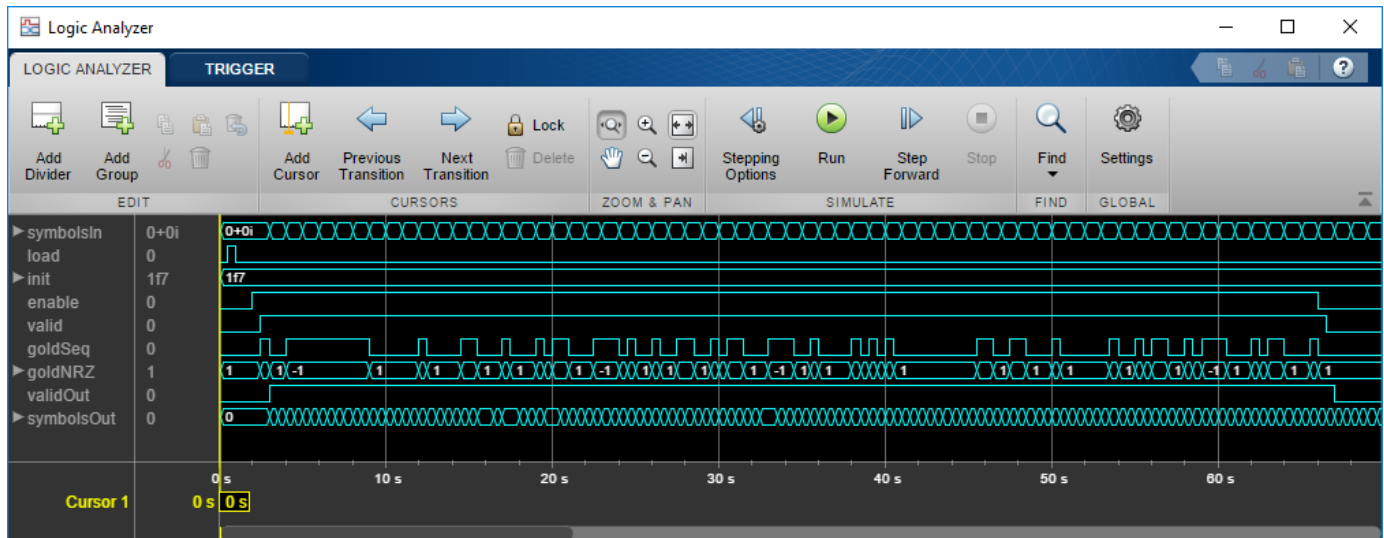
You can generate HDL from the HDL Descrambler subsystem.



The LTE Gold Sequence Generator block has no block parameters. It is configured to match the polynomial and shift length required by LTE standard TS 36.212. You must initialize the sequence with a 31-bit value on the **init** port, and load the value into the block by setting the **load** signal to 1 for one cycle. The **enable** signal generates the Gold sequence values. The output **valid** signal indicates when the output is available.



You can add data logging on the signals and use the Logic Analyzer to view the waveforms.



To generate and check the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('ltehdlGoldDescramblerModel/HDL_Descrambler')
```

To generate a test bench, use the following command:

```
makehdltb('ltehdlGoldDescramblerModel/HDL_Descrambler')
```

## See Also

### Blocks

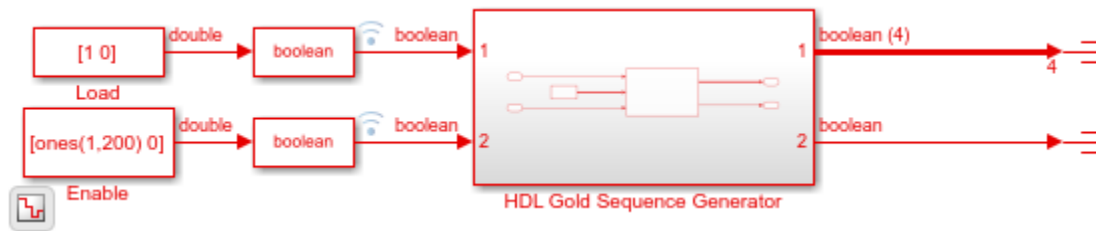
LTE Gold Sequence Generator

## Parallel Gold Sequence Generation

This example shows how to use the LTE Gold Sequence Generator block to generate multiple sequences in parallel for use in channel estimation.

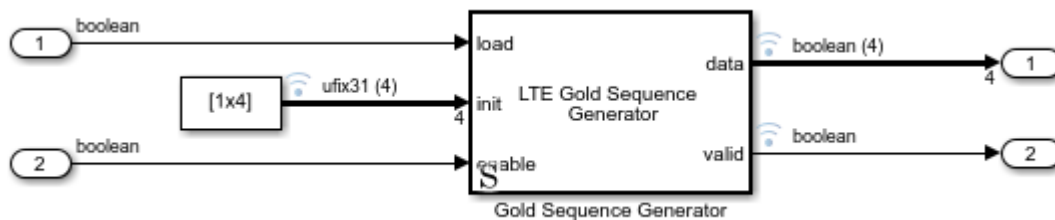
The example model initializes the LTE Gold Sequence Generator block with a vector that represents the **init** values for each of four channels. The block returns four independent Gold sequences.

You can generate HDL from the HDL Gold Sequence Generator subsystem.



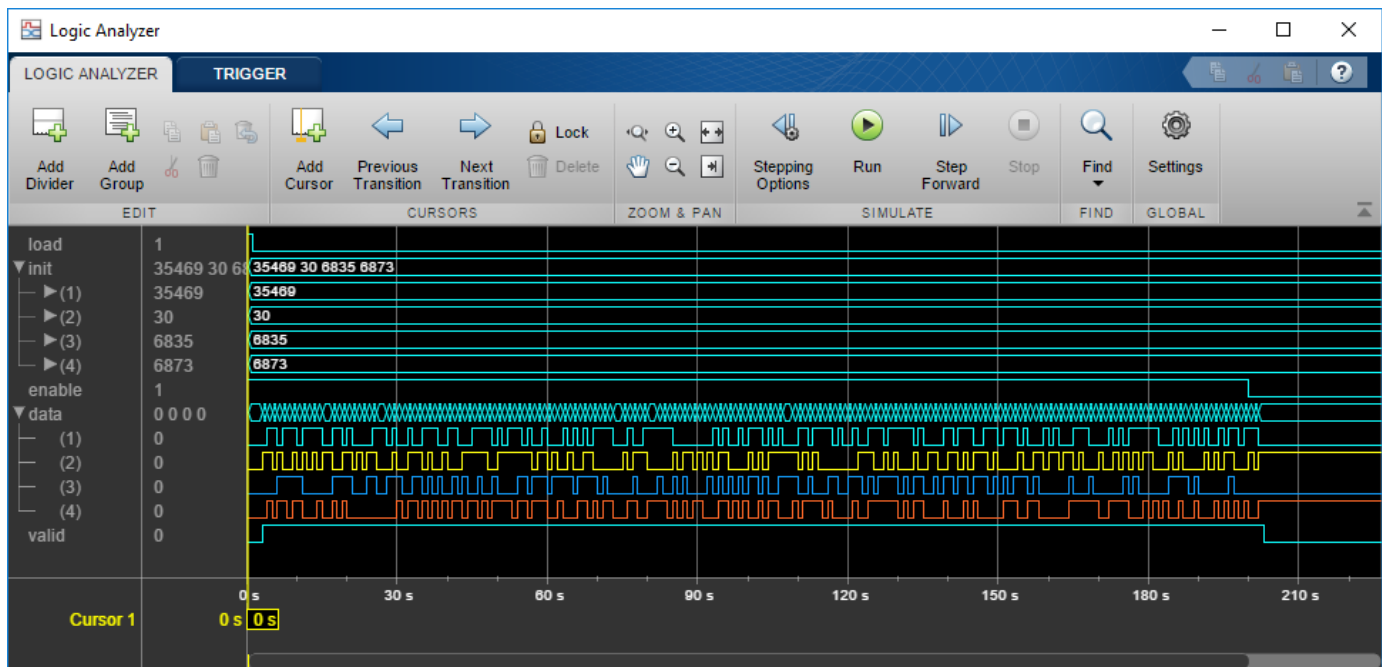
The LTE Gold Sequence Generator block has no block parameters. It is configured to match the polynomial and shift length required by LTE standard TS 36.212. You must initialize the sequence with a 31-bit value on the **init** port, and load the value into the block by setting the **load** signal to 1 for one cycle. This model has four **init** values, representing four channels.

The **enable** signal generates the Gold sequence values. The output is a vector of four values. The output **valid** signal indicates when the output data is available.



You can add data logging on the signals and use the Logic Analyzer to view the waveforms.





To generate and check the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('ltehdlGoldVectorModel/HDL Gold Sequence Generator')
```

To generate a test bench, use the following command:

```
makehdltb('ltehdlGoldVectorModel/HDL Gold Sequence Generator')
```

## See Also

### Blocks

LTE Gold Sequence Generator

## LTE OFDM Demodulation of Streaming Samples

This example shows how to use the LTE OFDM Demodulator block to return the LTE resource grid from streaming samples. You can generate HDL code from this block.

Generate input LTE OFDM symbols using LTE Toolbox™. Select a reference channel based on NDLRB, and specify the type of cyclic prefix.

```

enb = lteRMCDL('R.5');
enb.TotSubframes = 1;
enb.CyclicPrefix = 'Normal'; % or 'Extended'
% -----
%      NDLRB | Reference Channel
% -----
%      6      | R.4
%     15      | R.5
%     25      | R.6
%     50      | R.7
%     75      | R.8
%    100      | R.9
% -----

[waveform,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);
%%In this example, the Input data sample rate parameter is set to |Use
% maximum input data sample rate|. Hence, the LTE OFDM Demodulator block
% expects input samples at 30.72 MHz sample rate to correspond to the
% size of the FFT. The sample rate of |waveform| depends on NDLRB,
% so the generated waveform might be at a lower rate. To generate
% a test waveform, upsample the signal to 30.72 MHz, normalize the power,
% and add noise. Scale the signal magnitude to be in the range -1 to 1 for
% easy conversion to fixed-point types.

FsRx = 30.72e6;
FsTx = info.SamplingRate;
% -----
%      NDLRB          | Sampling Rate (MHz)
% -----
% 1) 6                | 1.92
% 2) 15               | 3.84
% 3) 25               | 7.68
% 4) 50               | 15.36
% 5) 75               | 30.72
% 6) 100              | 30.72
% -----

tx = resample(waveform,FsRx,FsTx);
avgTxPower = (tx' * tx) / length(tx);
tx = tx / sqrt(avgTxPower);
n = 0.1 * complex(randn(length(tx),1),randn(length(tx),1));
rx = tx + n;
rx = 0.99 * rx / max(abs(rx));

```

Use an LTE Toolbox function as a behavioral reference for the OFDM demodulation. Downsample the test waveform to the actual sample rate for the selected NDLRB. Then, compensate for the scale factor that results from the difference in FFT sizes.

```
refInput = resample(rx,FsTx,FsRx);
refGrid = lteOFDMDemodulate(info,refInput);
refGrid = refGrid * FsRx/FsTx;
```

Set up the Simulink™ model input data. Convert the test waveform to a fixed-point data type to model the result from a 12-bit ADC. The Simulink sample time is 30.72 MHz.

The Simulink model imports the sample stream `dataIn` and `validIn`, the input parameters `NDLRB` and `cyclicPrefixType`, and the variable `stopTime`.

```
NDLRB = info.NDLRB;
if strcmp(info.CyclicPrefix,'Normal')
    cyclicPrefixType = false;
else
    cyclicPrefixType = true;
end
```

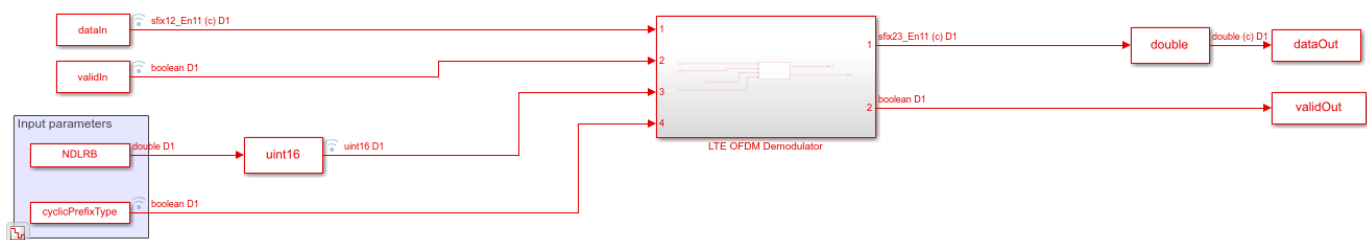
```
sampling_time = 1/FsRx;
dataIn = fi(rx,1,12,11);
validIn = true(length(dataIn),1);
```

Calculate the Simulink simulation time, accounting for the latency of the LTE OFDM Demodulator block. The latency of the FFT is fixed because the block uses a 2048-point FFT. Assume the maximum possible latency of the cyclic prefix removal and subcarrier selection operations. The simulation must run long enough to apply the input data, plus the latency of the final input symbol.

```
FFTlatency = 4137;
CPRemove_max = 512; % extended CP
carrierSelect_max = 424; % NDLRB 100
stopTime = sampling_time*(length(dataIn)+CPRemove_max+FFTlatency+carrierSelect_max);
```

Run the Simulink model. The model imports the `dataIn` and `validIn` structures and returns `dataOut` and `validOut`.

```
modelName = 'LTEOFDMDemodulatorExample';
open(modelName)
set_param(modelName,'SampleTimeColors','on');
set_param(modelName,'SimulationCommand','Update');
sim(modelName)
```



Compare the output of the Simulink model against the behavioral results, and calculate the SQNR of the HDL-optimized LTE OFDM Demodulator block.

```
rxgridSimulink = dataOut(validOut);

figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(refGrid(:)))
```

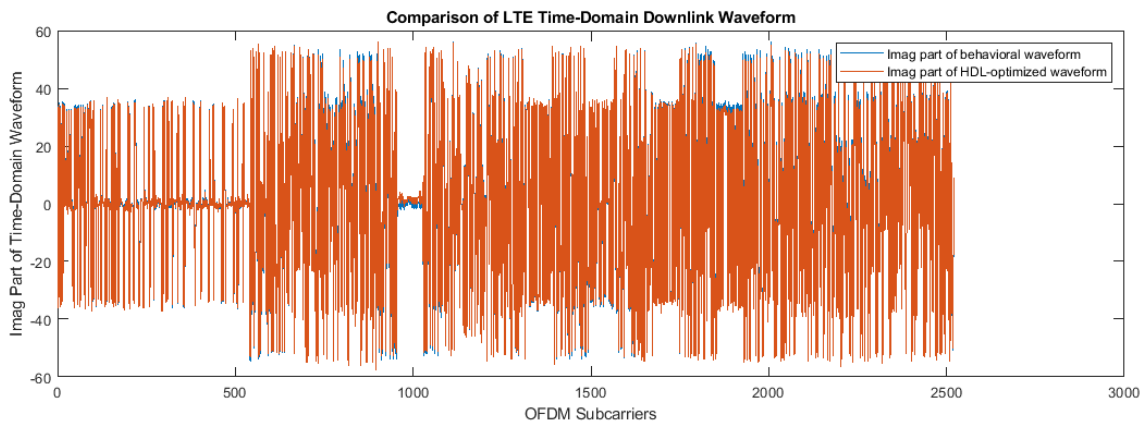
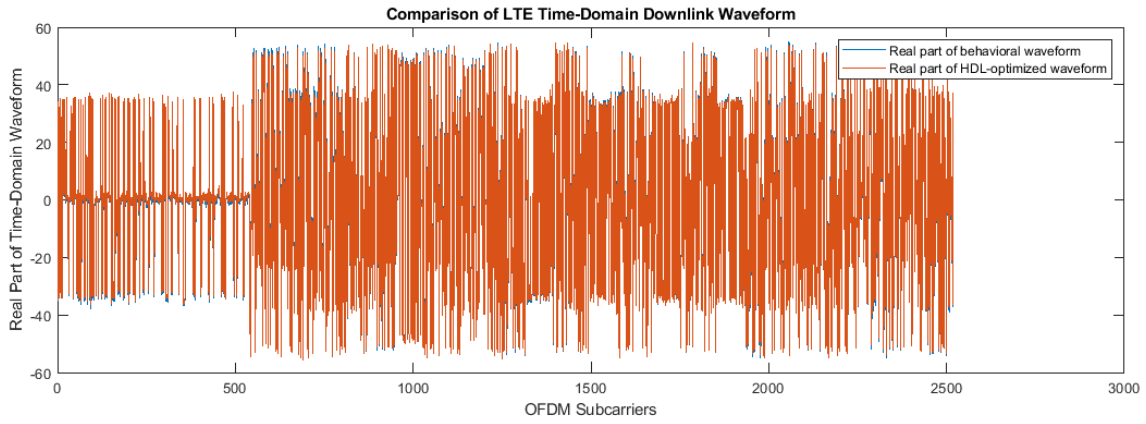
```
hold on
plot(squeeze(real(rxgridSimulink)))
legend('Real part of behavioral waveform','Real part of HDL-optimized waveform')
title('Comparison of LTE Time-Domain Downlink Waveform')
xlabel('OFDM Subcarriers')
ylabel('Real Part of Time-Domain Waveform')

subplot(2,1,2)
plot(imag(refGrid(:)))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of behavioral waveform','Imag part of HDL-optimized waveform')
title('Comparison of LTE Time-Domain Downlink Waveform')
xlabel('OFDM Subcarriers')
ylabel('Imag Part of Time-Domain Waveform')

sqrRealdB = 10*log10(var(real(rxgridSimulink))/abs(var(real(rxgridSimulink))-var(real(refGrid(:))))
sqrImagdB = 10*log10(var(imag(rxgridSimulink))/abs(var(imag(rxgridSimulink))-var(imag(refGrid(:))))

fprintf('\n LTE OFDM Demodulator: \n SQNR of real part is %.2f dB',sqrRealdB)
fprintf('\n SQNR of imaginary part is %.2f dB\n',sqrImagdB)
```

```
LTE OFDM Demodulator:
SQNR of real part is 25.98 dB
SQNR of imaginary part is 23.23 dB
```



## See Also

### Blocks

LTE OFDM Demodulator

## Reset and Restart LTE OFDM Demodulation

This example shows how to recover the LTE OFDM Demodulator block from an unfinished LTE cell. The input data is truncated to simulate the loss of a signal or a reset from the upstream parts of the receiver. The example model uses the reset signal to clear the internal state counters of the LTE OFDM Demodulator block and then restart calculations on the next cell. In this example, the Input data sample rate parameter of LTE OFDM Demodulator is set to Use maximum input data sample rate. So, the base sampling rate of the block is 30.72 MHz.

Generate two input LTE OFDM cells that use different NDLRBs or different types of cyclic prefix. Upsample both waveforms to the base sampling rate of 30.72 MHz.

```
% -----
%      NDLRB  |  Reference Channel
% -----
%      6      |  R.4
%     15      |  R.5
%     25      |  R.6
%     50      |  R.7
%     75      |  R.8
%    100      |  R.9
% -----
```

```
enb1 = lteRMCDL('R.9');
enb1.TotSubframes = 1;
enb1.CyclicPrefix = 'Normal'; % or 'Extended'
[waveform1,grid1,info1] = lteRMCDLTool(enb1,[1;0;0;1]);
```

```
enb2 = lteRMCDL('R.6');
enb2.TotSubframes = 1;
enb2.CyclicPrefix = 'Normal'; % or 'Extended'
[waveform2,grid2,info2] = lteRMCDLTool(enb2,[1;0;0;1]);
```

```
FsRx = 30.72e6;
tx1 = resample(waveform1,FsRx,info1.SamplingRate);
tx2 = resample(waveform2,FsRx,info2.SamplingRate);
```

Truncate the first waveform two-thirds through the cell. Concatenate the shortened cell with the second generated cell, leaving some invalid samples in between. Add noise, and scale the signal magnitude to be in the range [-1, 1] for easy conversion to fixed point.

```
tx1 = tx1(1:2*length(tx1)/3);

Lgap1 = 3000;
Lgap2 = 10000;
rx = [zeros(Lgap1,1); tx1; zeros(Lgap2,1); tx2];

L = length(rx);
rx = rx + 2e-4*complex(randn(L,1),randn(L,1));

dataIn_fp = 0.99*rx/max(abs(rx));
```

The LTE OFDM Demodulator block maintains internal counters of subframes within each cell. The block requires a reset after an incomplete cell to clear the counters before it can correctly demodulate subsequent cells. Create a reset pulse signal at the end of the first waveform.

```

resetIndex = Lgap1 + length(tx1);
resetIn = false(length(rx),1);
resetIn(resetIndex) = true;

```

Set up the Simulink™ model input data. Convert the test waveform to a fixed-point data type to model the result from a 12-bit ADC. The Simulink sample time is 30.72 MHz.

The Simulink model imports the sample stream `dataIn` and `validIn`, the input parameters `NDLRB` and `cyclicPrefixType`, the reset signal `resetIn`, and the simulation length `stopTime`.

```

dataIn = fi(dataIn_fp,1,12,11);

```

```

validIn = [false(Lgap1,1); true(length(tx1),1); false(Lgap2,1); true(length(tx2),1)];
validIn(resetIndex+1:Lgap1+length(tx1)) = false;

```

```

NDLRB = uint16([info1.NDLRB*ones(Lgap1 + length(tx1),1); info2.NDLRB*ones(Lgap2 + length(tx2),1)

```

```

cpType1 = strcmp(info1.CyclicPrefix,'Extended');
cpType2 = strcmp(info2.CyclicPrefix,'Extended');
cyclicPrefixType = [repmat(cpType1,Lgap1 + length(tx1),1); repmat(cpType2,Lgap2 + length(tx2),1)

```

Calculate the Simulink simulation time, accounting for the latency of the LTE OFDM Demodulator block. The latency of the FFT is fixed because the block uses a 2048-point FFT. Assume the maximum possible latency of the cyclic prefix removal and the subcarrier selection operations.

```

FFTlatency = 4137;
CPRemove_max = 512; % extended CP
carrierSelect_max = 424; % NDLRB 100

```

```

sampling_time = 1/FsRx;
stopTime = sampling_time*(length(dataIn) + CPMove_max + FFTlatency + carrierSelect_max);

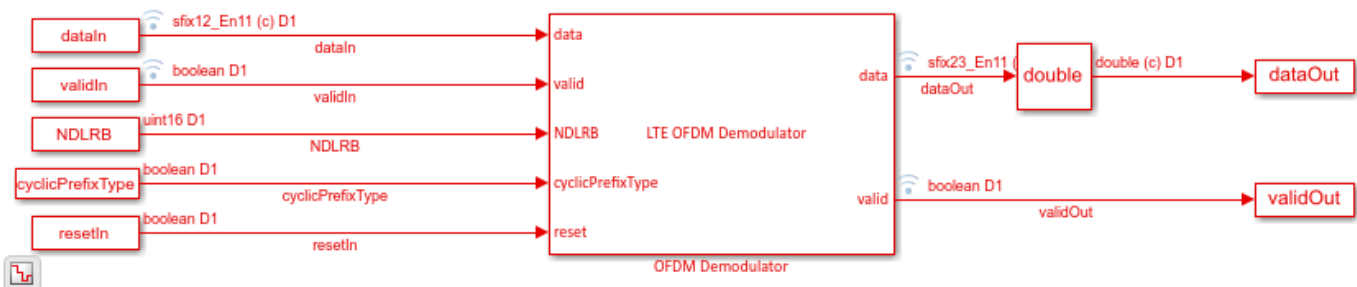
```

Run the Simulink model. The model imports the `dataIn` and `validIn` structures and returns `dataOut` and `validOut`.

```

modelName = 'LTEOFDMDemodResetExample';
open(modelName)
set_param(modelName,'SampleTimeColors','on');
set_param(modelName,'SimulationCommand','Update');
sim(modelName)

```



Split `dataOut` and `validOut` into two parts as divided by the reset pulse. The block applies the reset to the output data one cycle after the reset is applied on the input. Use the `validOut` signal to collect the valid output samples.

```

dataOut1 = dataOut(1:resetIndex);
dataOut2 = dataOut(resetIndex+1:end);

```

```

validOut1 = validOut(1:resetIndex);
validOut2 = validOut(resetIndex+1:end);

demodData1 = dataOut1(validOut1);
demodData2 = dataOut2(validOut2);

```

Generate reference data by flattening and normalizing the unmodulated resource grid data. Truncate the first cell in the same way as the modulated input data. Apply complex scaling to each demodulated sequence so that it can be compared to its corresponding reference data.

```

refData1 = grid1(:);
refData1 = refData1(1:length(demodData1));
refData2 = grid2(:);

refData1 = refData1/norm(refData1);
refData2 = refData2/norm(refData2);

demodData1 = demodData1/(refData1'*demodData1);
demodData2 = demodData2/(refData2'*demodData2);

```

Compare the output of the Simulink model against the truncated input grid, and display the results.

```

figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,2,1)
plot(real(refData1(:)))
hold on
plot(squeeze(real(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDRB %d) - Real part', info1.NDRB))
xlabel('OFDM Subcarriers')

subplot(2,2,2)
plot(imag(refData1(:)))
hold on
plot(squeeze(imag(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDRB %d) - Imaginary part', info1.NDRB))
xlabel('OFDM Subcarriers')

subplot(2,2,3)
plot(real(refData2(:)))
hold on
plot(squeeze(real(demodData2)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 2 (NDRB %d) - Real part', info2.NDRB))
xlabel('OFDM Subcarriers')

subplot(2,2,4)
plot(imag(refData2(:)))
hold on
plot(squeeze(imag(demodData2)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 2 (NDRB %d) - Imaginary part', info2.NDRB))
xlabel('OFDM Subcarriers')

sqnrRealdB1 = 10*log10(var(real(demodData1))/abs(var(real(demodData1)) - var(real(refData1(:)))));
sqnrImagdB1 = 10*log10(var(imag(demodData1))/abs(var(imag(demodData1)) - var(imag(refData1(:)))));

```



```

fprintf('\n Cell 1: SQNR of real part is %.2f dB',sqnrRealdB1)
fprintf('\n Cell 1: SQNR of imaginary part is %.2f dB\n',sqnrImagdB1)

sqnrRealdB2 = 10*log10(var(real(demodData2))/abs(var(real(demodData2)) - var(real(refData2(:))))
sqnrImagdB2 = 10*log10(var(imag(demodData2))/abs(var(imag(demodData2)) - var(imag(refData2(:)))))

fprintf('\n Cell 2: SQNR of real part is %.2f dB',sqnrRealdB2)
fprintf('\n Cell 2: SQNR of imaginary part is %.2f dB\n',sqnrImagdB2)

```

```

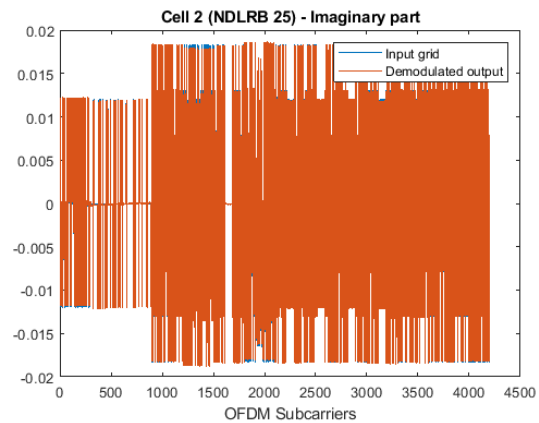
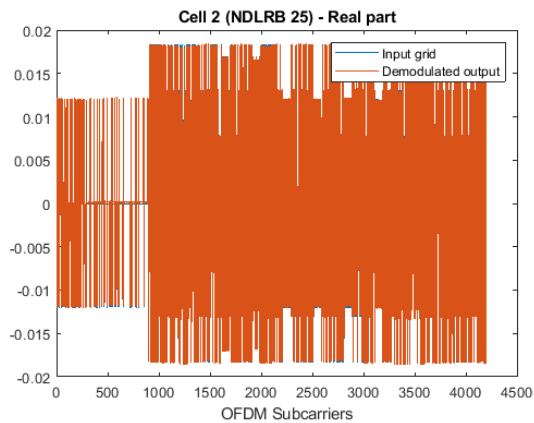
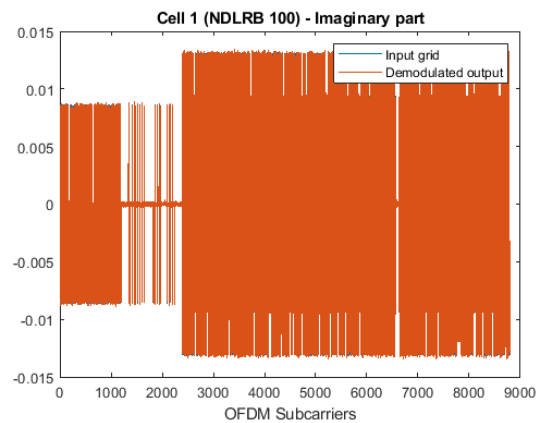
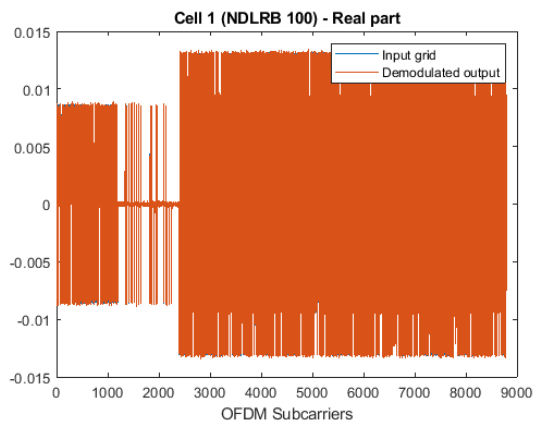
Cell 1: SQNR of real part is 33.71 dB
Cell 1: SQNR of imaginary part is 52.26 dB

```

```

Cell 2: SQNR of real part is 32.41 dB
Cell 2: SQNR of imaginary part is 36.72 dB

```



## See Also

### Blocks

LTE OFDM Demodulator

## Modulate and Demodulate LTE Resource Grid

This example shows how to modulate and then demodulate LTE resource grid samples. The model connects the LTE OFDM Modulator block to the LTE OFDM Demodulator block. To verify the algorithms of both blocks, this example compares the output of the demodulator with the input of the modulator. You can generate HDL code from either block.

Generate the input resource grid using LTE Toolbox™.

```
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;
```

```
% -----
%      NDLRB          |      Sampling Rate (MHz)
% -----
%          6          |      R.4
%          15         |      R.5
%          25         |      R.6
%          50         |      R.7
%          75         |      R.8
%          100        |      R.9
% -----
```

```
[~,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);
```

```
NDLRB=info.NDLRB;
if strcmp(enb.CyclicPrefix,'Normal')
    CPTYPE=false;
else
    CPTYPE=true;
end
```

```
sampling_time=1/30.72e6;
modulatorLatency=4137+2048*2;
demodulatorLatency=4137+2048*2;
stoptime=enb.TotSubframes*(30720+modulatorLatency+demodulatorLatency)*sampling_time;
```

Convert the LTEGrid sample frames to a stream of samples with control signals for input to the Simulink® model.

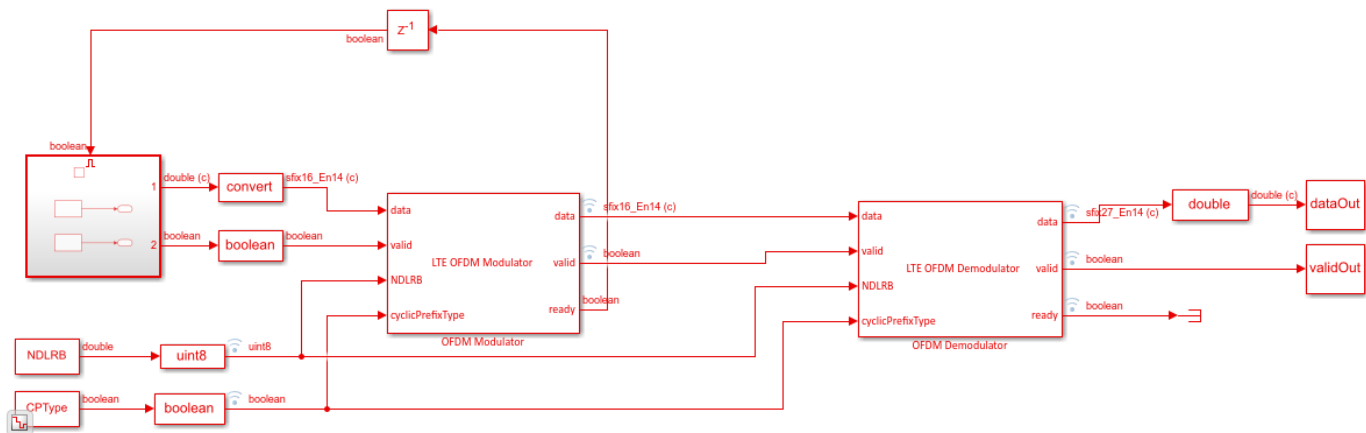
```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;

[dataIn,ctrl] = whdlFramesToSamples(mat2cell(LTEGrid(:),numel(LTEGrid),1),...
    idlecyclesbetweensamples,idlecyclesbetweenframes);
validIn = logical(ctrl(:,3));
```

Run the Simulink model to modulate and demodulate the samples, and save the output samples to a workspace variable.

```
open_system('LTEHDLofDMModDemodExample')
sim('LTEHDLofDMModDemodExample');

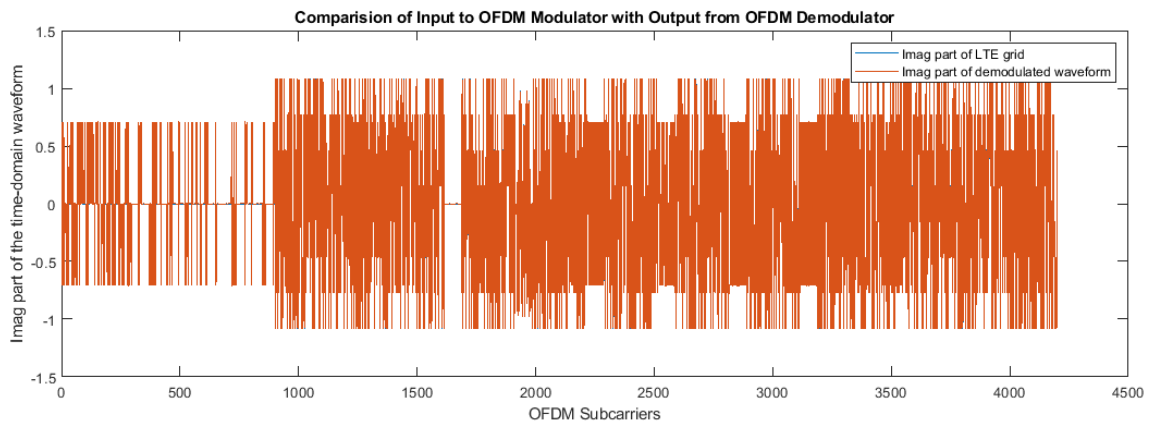
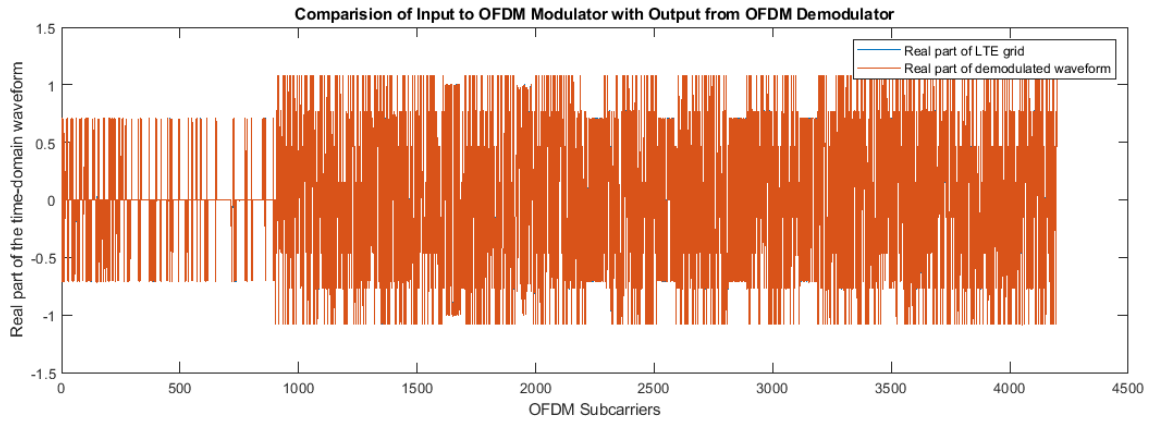
rxgridSimulink = dataOut(validOut);
```



Compare the input of the modulator, generated from the `lteRMCDLTool` function, and the output of the demodulator from the model.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(LTEGrid(:)));
hold on
plot(squeeze(real(rxgridSimulink)));
legend('Real part of LTE grid','Real part of demodulated waveform');
title('Comparison of Input to OFDM Modulator with Output from OFDM Demodulator');
xlabel('OFDM Subcarriers');
ylabel('Real part of the time-domain waveform');

subplot(2,1,2)
plot(imag(LTEGrid(:)))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of LTE grid','Imag part of demodulated waveform');
title('Comparison of Input to OFDM Modulator with Output from OFDM Demodulator');
xlabel('OFDM Subcarriers');
ylabel('Imag part of the time-domain waveform');
```



## See Also

### Blocks

LTE OFDM Demodulator | LTE OFDM Modulator

## OFDM Modulation of LTE Resource Grid Samples

This example shows how to use the LTE OFDM Modulator block to modulate LTE resource grid samples to an equivalent time-domain signal output. You can generate HDL code from this block.

Generate the input resource grid using LTE Toolbox™.

```
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;
% -----
%      NDLRB          |      Sampling Rate (MHz)
% -----
%      6              |      R.4
%      15             |      R.5
%      25             |      R.6
%      50             |      R.7
%      75             |      R.8
%      100            |      R.9
% -----

[~,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);
[eNodeBOutput,~] = lteOFDMModulate(enb,LTEGrid);
```

Convert the LTEGrid sample frames to a stream of samples with control signals for input to the Simulink® model.

```
NDLRB=info.NDLRB;
if strcmp(enb.CyclicPrefix,'Normal')
    CPTYPE=false;
else
    CPTYPE=true;
end

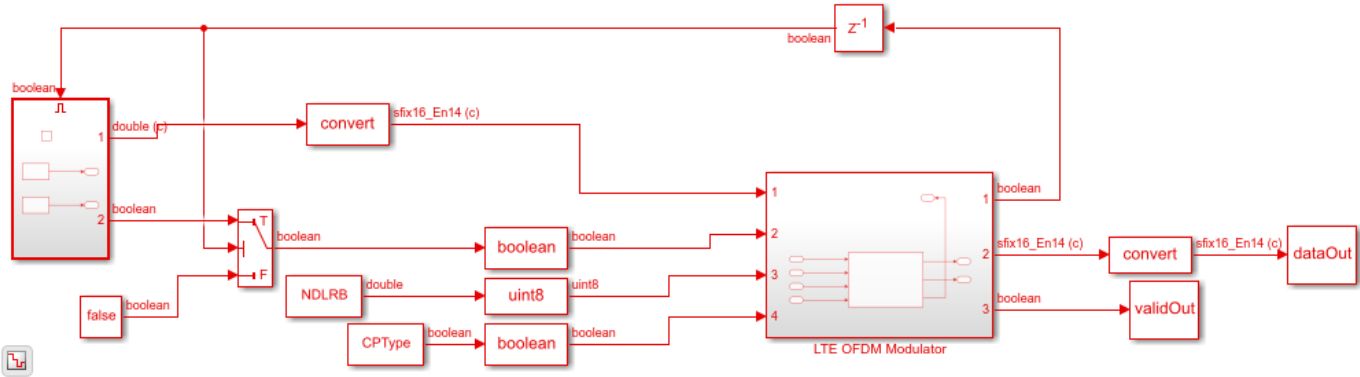
sampling_time=1/30.72e6;
stoptime=enb.TotSubframes*(30720+4137+2048*2)*sampling_time;

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;

[dataIn,ctrl] = whdlFramesToSamples(mat2cell(LTEGrid(:),numel(LTEGrid),1),...
    idlecyclesbetweensamples,idlecyclesbetweenframes);
validIn = logical(ctrl(:,3));
```

Run the Simulink model.

```
modelname = 'OFDMModulatorModelExample';
open_system(modelname);
sim(modelname);
```

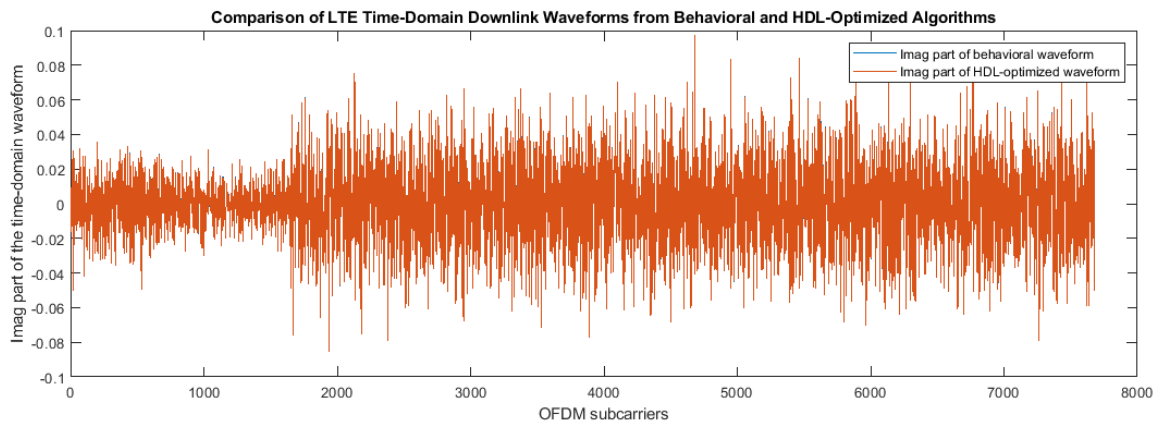
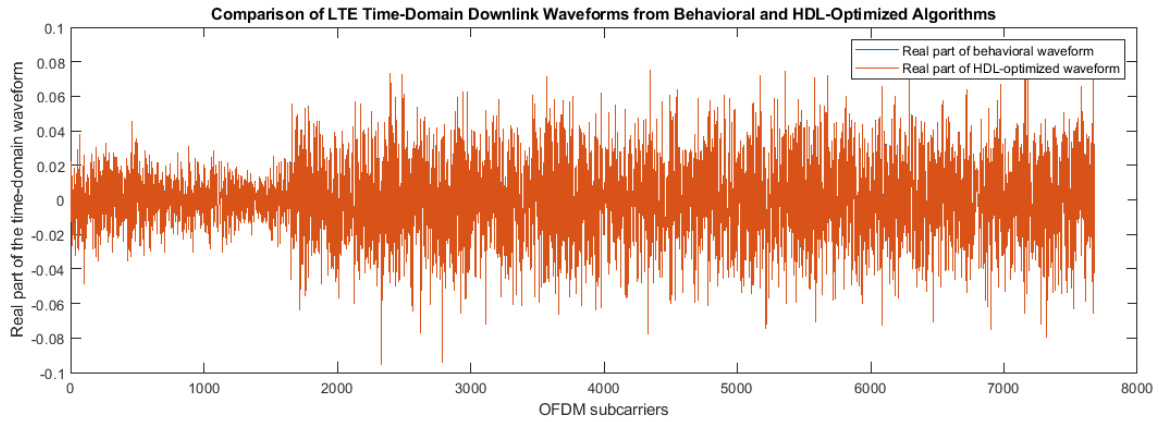


Save the output of the Simulink model and then compare the output of the model against the output of the `lteOFDMModulate` function.

```
rxgridSimulink=dataOut(validOut);
```

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(eNodeBOutput));
hold on
plot(squeeze(real(rxgridSimulink)));
legend('Real part of behavioral waveform','Real part of HDL-optimized waveform');
title('Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algorithms');
xlabel('OFDM subcarriers');
ylabel('Real part of the time-domain waveform');

subplot(2,1,2)
plot(imag(eNodeBOutput))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of behavioral waveform','Imag part of HDL-optimized waveform');
title('Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algorithms');
xlabel('OFDM subcarriers');
ylabel('Imag part of the time-domain waveform');
```



## See Also

### Blocks

LTE OFDM Modulator

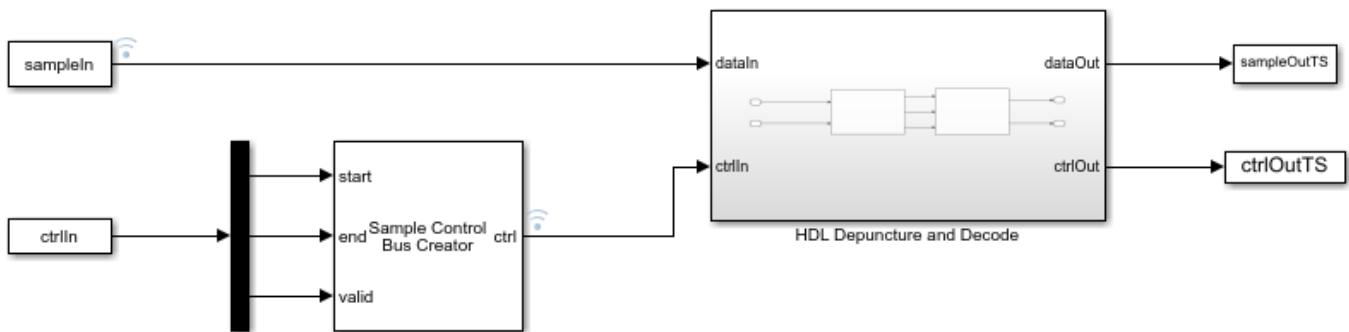
## Depuncture and Decode Streaming Samples

This example shows how to use the hardware-friendly Depuncturer block and Viterbi Decoder block to decode samples encoded at WLAN code rates.

Generate input samples in MATLAB® by encoding random data, BPSK-modulating the samples, applying a channel model, demodulating the samples, and creating received soft-decision bits. Then, import the soft-decision bits into a Simulink® model to depuncture and decode the samples. Export the result of the Simulink simulation back to MATLAB and compare it against the original input samples.

The example model supports HDL code generation for the HDL Depuncture and Decode subsystem.

```
modelName = 'ltehdlViterbiDecoderModel';
open_system(modelName);
```



### Set Up Code Rate Parameters

Set up workspace variables that describe the code rate. The Viterbi Decoder block supports constraint lengths in the range [3,9] and polynomial lengths in the range [2,7].

Choose a traceback depth in the range [3,128]. For non-punctured samples, the recommended depth is 5 times the *constraintLength*. For punctured samples, the recommended depth is 10 times the *constraintLength*.

Starting from a code rate of 1/2, IEEE 802.11 WLAN specifies three puncturing patterns to generate three additional code rates. Choose one of these code rates, and then set the frame size and puncturing pattern based on that rate. You can also choose the unpunctured code rate of 1/2.

IEEE 802.11 WLAN specifies different modulation types for different code rates and uses 'Terminated' mode. This example uses BPSK modulation for all rates and can run with 'Terminated' or 'Truncated' operation mode. The blocks also support 'Continuous' mode, but it is not included in this example.

```
constraintLength = 7;
codeGenerator = [133 171];
opMode = 'Terminated';
tracebackDepth = 10*constraintLength;

trellis = poly2trellis(constraintLength,...
    codeGenerator);
```



```

% IEEE 802.11n-2009 WLAN 1/2 (7, [133 171])
% Rate   Puncture Pattern   Maximum Frame Size
% 1/2    [1;1;1;1]               2592
% 2/3    [1;1;1;0]               1728
% 3/4    [1;1;1;0;0;1]      1944
% 5/6    [1;1;1;0;0;1;1;0;0;1] 2160
codeRate = 3/4;

if (codeRate == 2/3)
    puncVector = logical([1;1;1;0]);
    frameSize = 1728;
elseif (codeRate == 3/4)
    puncVector = logical([1;1;1;0;0;1]);
    frameSize = 1944;
elseif (codeRate == 5/6)
    puncVector = logical([1;1;1;0;0;1;1;0;0;1]);
    frameSize = 2160;
else % codeRate == 1/2
    puncVector = logical([1;1;1;1]);
    frameSize = 2592;
end

if strcmpi(opMode,'Terminated')
    % Terminate the state at the end of the frame
    tailLen = constraintLength-1;
else
    % Truncated mode
    tailLen = 0;
end

```

### Generate Samples for Decoding

Use Communications Toolbox™ functions and System objects to generate encoded samples and apply channel noise. Demodulate the received samples, and create soft-decision values for each sample.

```

EbNo = 10;
EcNo = EbNo - 10*log10(numel(codeGenerator));

numFrames = 5;
numSoftBits = 4;

txMessages = cell(1,numFrames);
rxSoftMessages = cell(1,numFrames);

No = 10^((-EcNo)/10);
quantStepSize = sqrt(No/2^numSoftBits);

modulator = comm.BPSKModulator;
channel = comm.AWGNChannel('EbNo',EcNo);
demodulator = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio');

for ii = 1:numFrames
    txMessages{ii} = [randn(frameSize - tailLen,1)
        zeros(tailLen,1)]>0;
    % Convolutional encoding and puncturing
    txCodeword = convenc(txMessages{ii},trellis,puncVector);
    % Modulation
    modOut = modulator.step(txCodeword);
end

```

```

% Channel
chanOut = channel.step(modOut);
% Demodulation
demodOut = -demodulator.step(chanOut)/4;
% Convert to soft-decision values
rxSoftMessagesDouble = demodOut./quantStepSize;
rxSoftMessages{ii} = fi(rxSoftMessagesDouble,1,numSoftBits,0);
end

```

### Set Up Variables for Simulink Simulation

The Simulink model requires streaming samples with accompanying control signals. Use the `whdlFramesToSamples` function to convert the framed `rxSoftMessages` to streaming samples and generate the matching control signals.

Calculate the required simulation time from the latency of the depuncture and decoder blocks.

```

samplesizeIn = 1;
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
if strcmpi(opMode,'Truncated')
    % Truncated mode requires a gap between frames of at least constraintLength-1
    idlecyclesbetweenframes = constraintLength - 1;
end

```

```

[sampleIn,ctrlIn] = whdlFramesToSamples(rxSoftMessages, ...
    idlecyclesbetweensamples,idlecyclesbetweenframes,samplesizeIn);

```

```

depunLatency = 6;
vitLatency = 4*tracebackDepth + constraintLength + 13;
latency = vitLatency + depunLatency;

```

```

simTime = size(ctrlIn,1) + latency;
sampletime = 1;

```

### Run the Simulink Model

Call the Simulink model to depuncture and decode the samples. The model exports the decoded samples to the MATLAB workspace. The Depuncture and Viterbi Decoder block parameters are configured using workspace variables. Because **Operation mode** is a list parameter, use `set_param` to assign the workspace value.

Convert the streaming samples back to framed data for comparison.

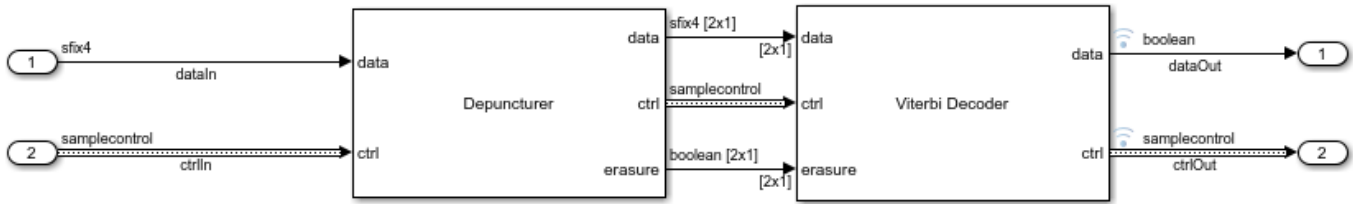
```

set_param([modelName '/HDL Depuncture and Decode'],'Open','on');
set_param([modelName '/HDL Depuncture and Decode/Viterbi Decoder'],...
    'TerminationMethod',opMode);
sim(modelname);

sampleOut = squeeze(sampleOutTS.Data);
ctrlOut = [squeeze(ctrlOutTS.start.Data) ...
    squeeze(ctrlOutTS.end.Data) ...
    squeeze(ctrlOutTS.valid.Data)];
rxMessages = whdlSamplesToFrames(sampleOut,ctrlOut);

```

Maximum frame size computed to be 1944 samples.



## Verify Results

Compare the output samples against the generated input samples.

```
fprintf('\nDecoded Samples\n');
for ii = 1:numFrames
    numBitsErr = sum(xor(txMessages{ii},rxMessages{ii}));
    fprintf('Frame #%d: %d bits mismatch \n',ii,numBitsErr);
end
```

Decoded Samples

```
Frame #1: 0 bits mismatch
Frame #2: 0 bits mismatch
Frame #3: 0 bits mismatch
Frame #4: 0 bits mismatch
Frame #5: 0 bits mismatch
```

## See Also

### Blocks

Depuncturer | Viterbi Decoder

## LTE Symbol Modulation of Data Bits

This example shows how to use the LTE Symbol Modulator block to modulate data bits to complex data symbols. You can generate HDL code from this block.

Set up input data parameters. Choose a data length for each modulation type. The data length must be an integer multiple of number of bits per symbol.

```
rng(0);
framesize = 240;

% Map modulation names to values
% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% others - QPSK

% For LTE Symbol Modulator Simulink block
modSelVal = [0;1;2;3;4];

% For |lteSymbolModulate| function
modSelStr = {'BPSK', 'QPSK', '16QAM', '64QAM', '256QAM'};

outWordLength = 16;
numframes = length(modSelVal);
dataBits = cell(1,numframes);
modSelTmp = cell(1,numframes);
lteFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

Convert the framed input data to a stream of samples and input the stream to the LTE Symbol Modulator Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataBits, idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp, idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
load = logical(ctrl(:,1)');
validIn = logical(ctrl(:,3)');

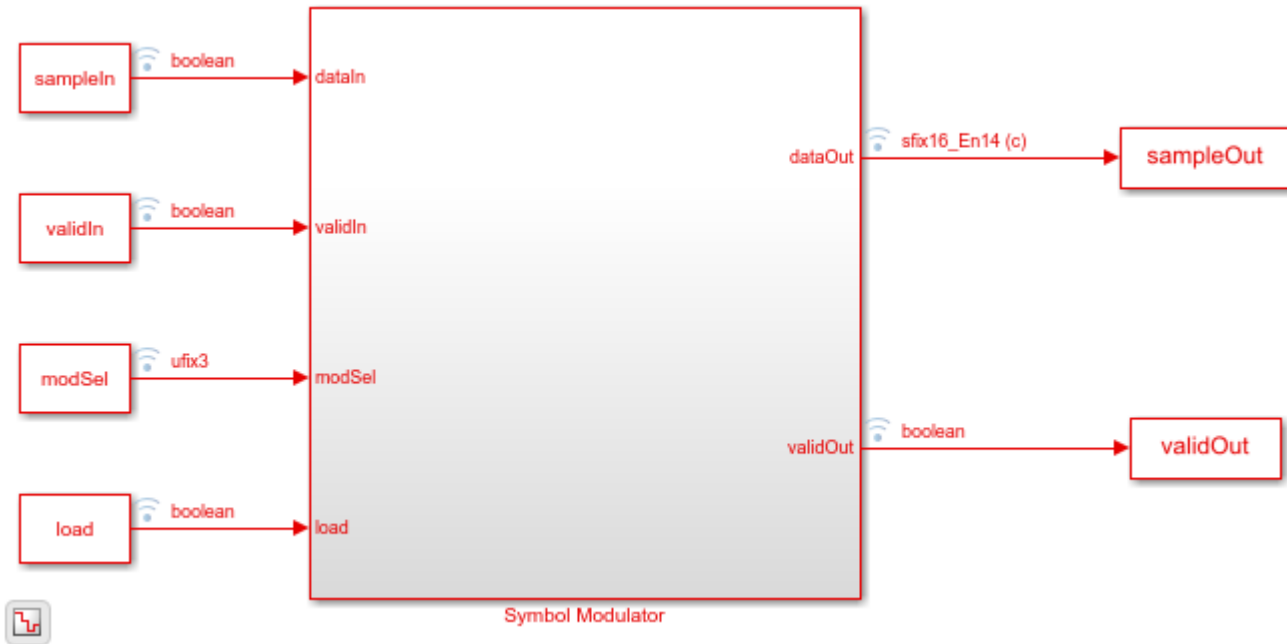
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1);
```

Run the Simulink model.

```

modelName = 'ltehdlSymbolModulatorModel';
open_system(modelname);
sim(modelname);

```



Export the stream of modulated samples from Simulink to the MATLAB workspace.

```

sampleOut = squeeze(sampleOut).';
lteHDLOutput = sampleOut(squeeze(validOut));

```

Modulate data bits with `lteSymbolModulate` function and use its output as a reference data.

```

for ii = 1:numframes
    lteFcnOutput{ii} = lteSymbolModulate(dataBits{ii},modSelStr{ii}).';
end

```

Compare the output of the Simulink model against the output of `lteSymbolModulate` function.

```

fprintf('\nLTE Symbol Modulator\n');
lteFcnOutput = fi(cell2mat(lteFcnOutput),1,outWordLength,outWordLength-2);
difference = sum(abs(lteHDLOutput-lteFcnOutput(1:length(lteHDLOutput))));
fprintf('\nTotal number of samples differed between Simulink block output and Reference data output: %d\n',difference);

```

LTE Symbol Modulator

Total number of samples differed between Simulink block output and Reference data output: 0

## See Also

### Blocks

LTE Symbol Modulator

## NR Symbol Modulation of Data Bits

This example shows how to use the NR Symbol Modulator block to modulate data bits to complex data symbols. You can generate HDL code from this block.

Set up input data parameters. Choose a data length for each modulation type. The data length must be an integer multiple of number of bits per symbol.

```

rng(0);
framesize = 240;

% Map modulation names to values
% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% 5 - pi/2-BPSK
% others - QPSK

% for NR Symbol Modulator Simulink block
modSelVal = [0;1;2;3;4;5];

% for nrSymbolModulate function
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM','pi/2-BPSK'};

outWordLength = 16;
numframes = length(modSelVal);
dataBits = cell(1,numframes);
modSelTmp = cell(1,numframes);
nrFcnOutput = cell(1,numframes);

```

Generate frames of random input samples.

```

for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end

```

Convert the framed input data to a stream of samples and input the stream to the Simulink block.

```

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataBits,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
load = logical(ctrl(:,1)');
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1);

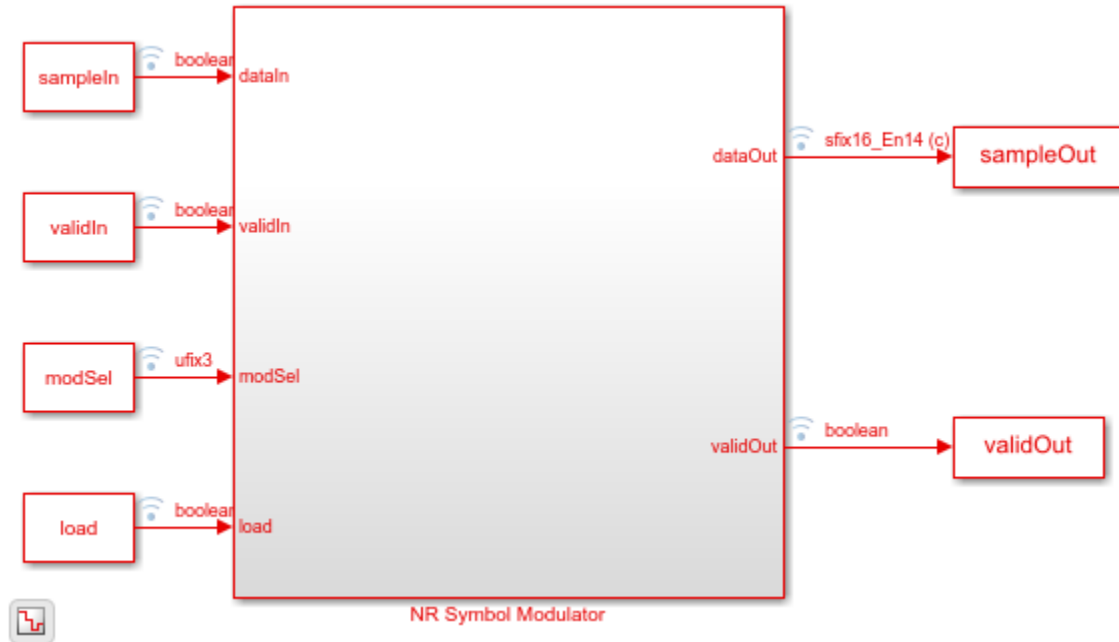
```

Run the Simulink model.

```

modelName = 'nrhdlSymbolModulatorModel';
open_system(modelName);
sim(modelName);

```



Export the stream of modulated samples from Simulink to the MATLAB workspace.

```

sampleOut = squeeze(sampleOut).';
nrHDLOutput = sampleOut(squeeze(validOut));

```

Modulate frame data bits with nrSymbolModulate function and use the output of this function as a reference data.

```

for ii = 1:numframes
    nrFcnOutput{ii} = nrSymbolModulate(dataBits{ii},modSelStr{ii}).';
end

```

Compare the output of the Simulink model against the output of nrSymbolModulate function.

```

fprintf('\nNR Symbol Modulator\n');
nrFcnOutput = fi(cell2mat(nrFcnOutput),1,outWordLength,outWordLength-2);
error = sum(abs(nrHDLOutput-nrFcnOutput(1:length(nrHDLOutput))));
fprintf('\nTotal number of samples differed between Behavioral and HDL simulation: %d \n',error)

```

```
NR Symbol Modulator
```

```
Total number of samples differed between Behavioral and HDL simulation: 0
```

## See Also

### Blocks

NR Symbol Modulator

## LTE Symbol Demodulation of Complex Data Symbols

This example shows how to use the LTE Symbol Demodulator block to demodulate complex LTE data symbols to data bits or LLR values. The workflow follows these steps:

- 1 Set up input data parameters.
- 2 Generate frames of random input samples.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink®.
- 4 Run the Simulink® model, which contains the LTE Symbol Demodulator block.
- 5 Export the stream of demodulated samples from Simulink to the MATLAB® workspace.
- 6 Demodulate data symbols with `lteSymbolDemodulate` function to use its output as a reference data.
- 7 Compare Simulink block output data with the reference MATLAB function output.

Set up input data parameters.

Map modulation names to values. The numerical values are used to set up the LTE Symbol Demodulator block. The strings are used to configure the `lteSymbolDemodulate` function.

```
rng(0);
framesize = 10;

% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% others - QPSK
modSelVal = [0;1;2;3;4];
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM'};

decType = 'Soft';

numframes = length(modSelVal);
dataSymbols = cell(1,numframes);
modSelTmp = cell(1,numframes);
lteFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

Convert the framed input data to a stream of samples and input the stream to the LTE Symbol Demodulator Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
```

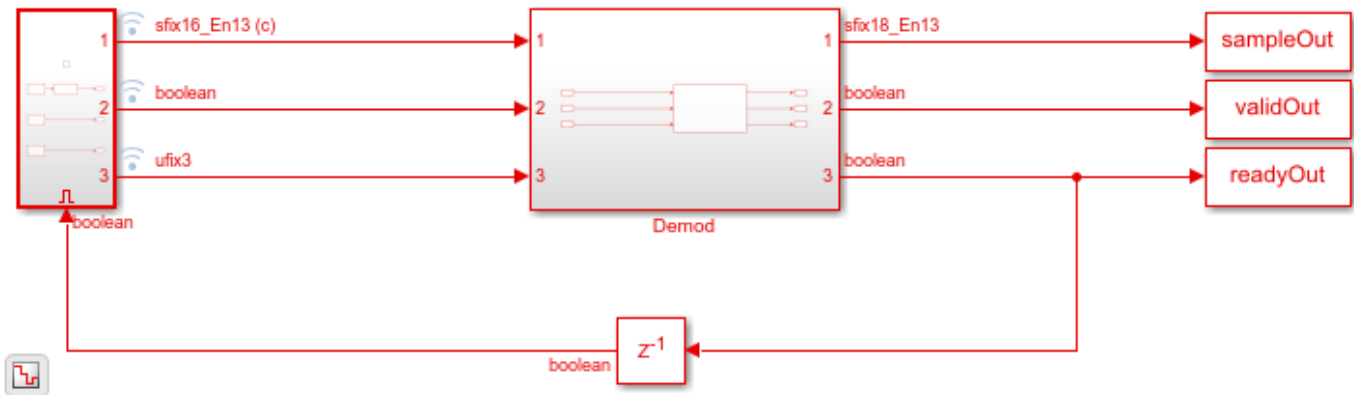


```
validIn = logical(ctrl(:,3)');
```

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*8;
```

Run the Simulink model.

```
modelname = 'ltehdlSymbolDemodulatorModel';
open_system(modelname);
set_param([modelname '/Demod/LTE Symbol Demodulator'], 'DecisionType', decType)
sim(modelname);
```



Export the stream of demodulated samples from Simulink to the MATLAB workspace.

```
lteHDLOutput = sampleOut(validOut).';
```

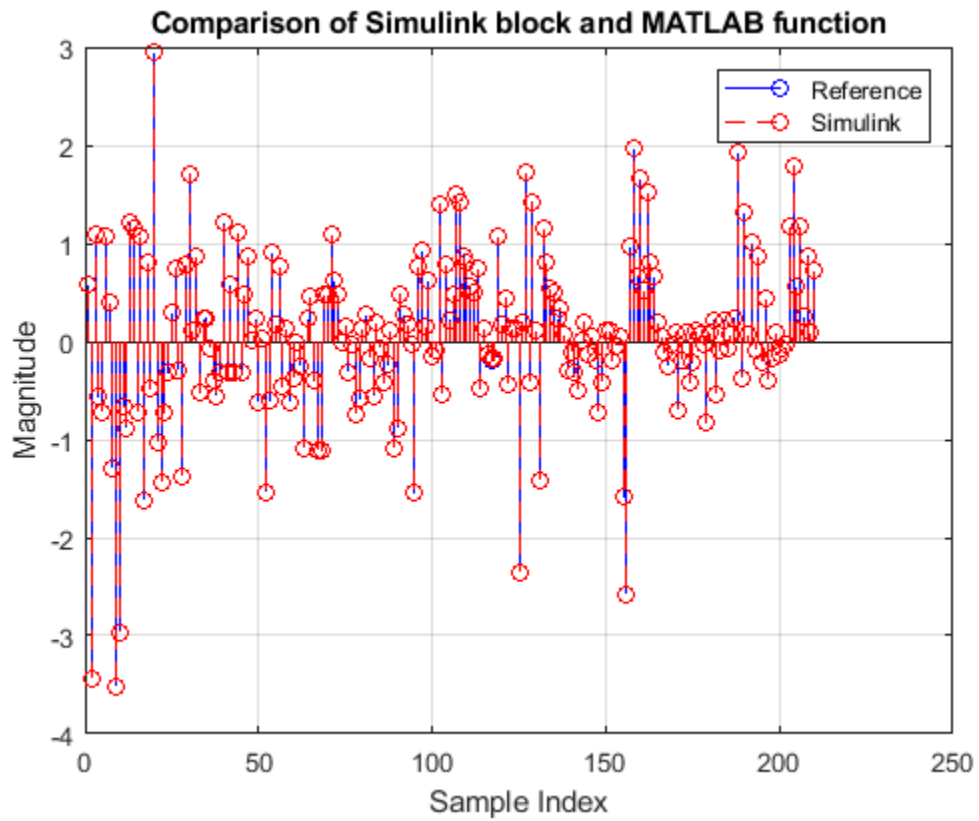
Demodulate data symbols with `lteSymbolDemodulate` function and use its output as a reference data.

```
for ii = 1:numframes
    lteFcnOutput{ii} = lteSymbolDemodulate(dataSymbols{ii}, modSelStr{ii}, decType).';
end
```

Compare the output of the Simulink model against the output of `lteSymbolDemodulate` function.

```
lteFcnOutput = double(cell2mat(lteFcnOutput));
```

```
figure(1)
stem(lteHDLOutput, 'b')
hold on
stem(lteFcnOutput, '--r')
grid on
legend('Reference', 'Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function')
```



## See Also

### Blocks

LTE Symbol Demodulator

## NR Symbol Demodulation of Complex Data Symbols

This example shows how to use the NR Symbol Demodulator block to demodulate complex NR data symbols to data bits or LLR values. The workflow follows these steps:

- 1 Set up input data parameters.
- 2 Generate frames of random input samples.
- 3 Convert framed input data to a stream of samples and import the stream into Simulink.
- 4 Run the Simulink® model, which contains the NR Symbol Demodulator block.
- 5 Export the stream of demodulated samples from Simulink to the MATLAB® workspace.
- 6 Demodulate data symbols with `nrSymbolDemodulate` function to use its output as a reference data.
- 7 Compare Simulink block output data with the reference MATLAB function output.

Set up input data parameters.

Map modulation names to values. The numerical values are used to set up the NR Symbol Demodulator block. The strings are used to configure the `nrSymbolDemodulate` function.

```
rng(0);
framesize = 10;

% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% 5 - pi/2-BPSK
% others - QPSK
modSelVal = [0;1;2;3;4;5];
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM','pi/2-BPSK'};

decType = 'Soft';

numframes = length(modSelVal);
dataSymbols = cell(1,numframes);
modSelTmp = cell(1,numframes);
nrFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

Convert the framed input data to a stream of samples and input the stream to the NR Symbol Demodulator Simulink block.

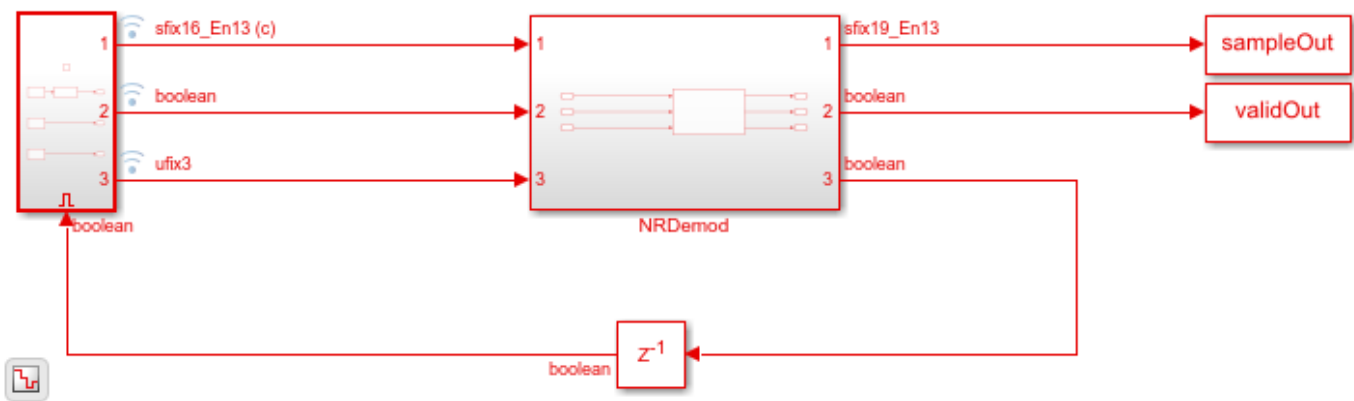
```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
```

```
idlecyclesbetweenframes);
validIn = logical(ctrl(:,3)');
```

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*8;
```

Run the Simulink model.

```
modelname = 'nrhdlSymbolDemodulatorModel';
open_system(modelname);
set_param([modelname '/NRDemod/NR Symbol Demodulator'],'DecisionType',decType)
sim(modelname);
```



Export the stream of demodulated samples from Simulink to the MATLAB workspace.

```
nrHDLOutput = sampleOut(validOut).';
```

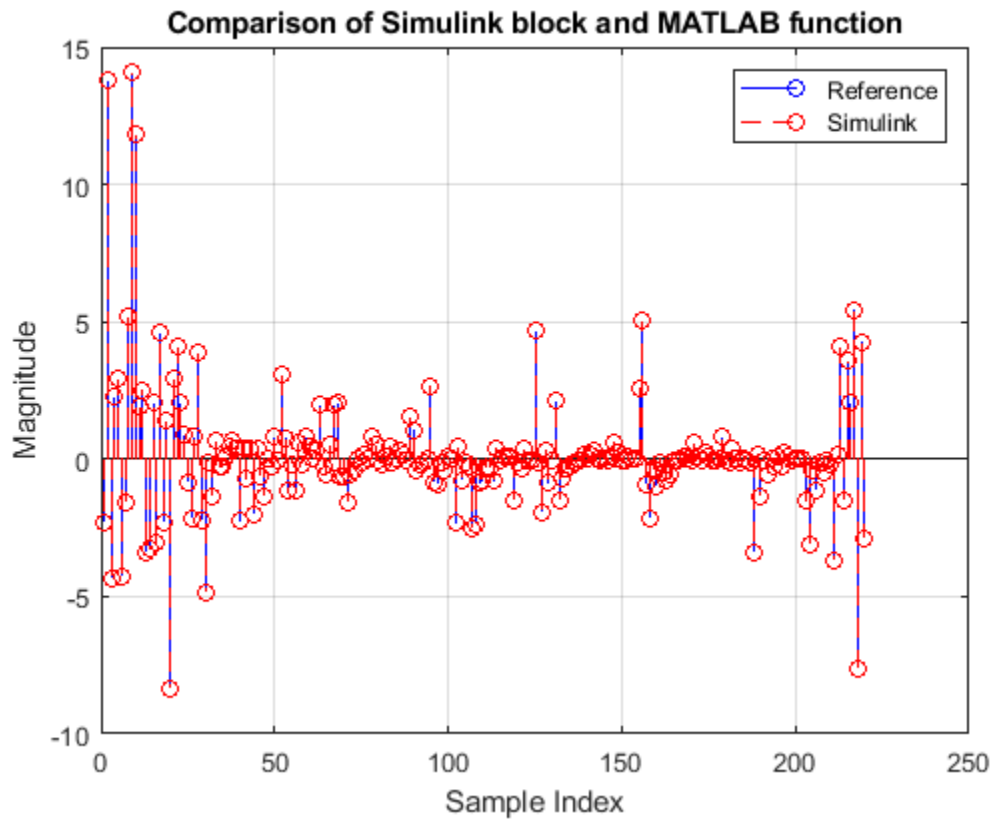
Demodulate data symbols with nrSymbolDemodulate function and use its output as a reference data.

```
for ii = 1:numframes
nrFcnOutput{ii} = nrSymbolDemodulate(dataSymbols{ii},modSelStr{ii},'DecisionType',decType,1).';
end
```

Compare the output of the Simulink model against the output of nrSymbolDemodulate function.

```
nrFcnOutput = double(cell2mat(nrFcnOutput));

figure(1)
stem(nrHDLOutput,'b')
hold on
stem(nrFcnOutput,'--r')
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function')
```



## See Also

### Blocks

NR Symbol Demodulator

## Application of FFT 1536 block in LTE OFDM Demodulation

This example shows how to use the FFT 1536 block in LTE OFDM demodulation.

- 1 Generate transmitter waveform.
- 2 Remove cyclic prefix.
- 3 Prepare inputs for FFT 1536 simulation.
- 4 Form resource grid.
- 5 Compare the CellRS symbols from the grid with that of `lteCellRS` function.
- 6 Generate HDL code.

Generate transmitter waveform.

```
cfg = lteTestModel('1.1', '15MHz');
cfg.TotSubframes = 1;
tx = lteTestModelTool(cfg);
```

The above transmitter waveform generation uses a 2048-point FFT, which results in a scaling factor of  $\frac{1}{2048}$  in OFDM modulation. If a 1536-point FFT were used, the waveform would have a scaling factor of  $\frac{1}{1536}$ . This example multiplies the waveform by a factor of  $\frac{2048}{1536}$  to achieve the correct scaling.

```
tx = tx*(2048/1536);
```

To achieve a 23.04 Msps sampling rate, resample the `tx` samples by  $\frac{3}{4} = \frac{23.04e6}{30.72e6}$

```
rx = resample(tx,3,4); % rate conversion from 30.72Msps to 23.04Msps
```

Remove cyclic prefix. The first symbol of each slot has 12 additional CP samples.

```
rx(11520+1:11520+12) = []; % discard 12 CP samples in slot 2
rx(1:12) = []; % discard 12 CP samples in slot 1
rx = reshape(rx,108+1536,14); % reshape to form 14 OFDM symbols
rx(1:108,:) = []; % discard remaining 108 CP samples from all symbols
```

Prepare inputs for FFT 1536 simulation.

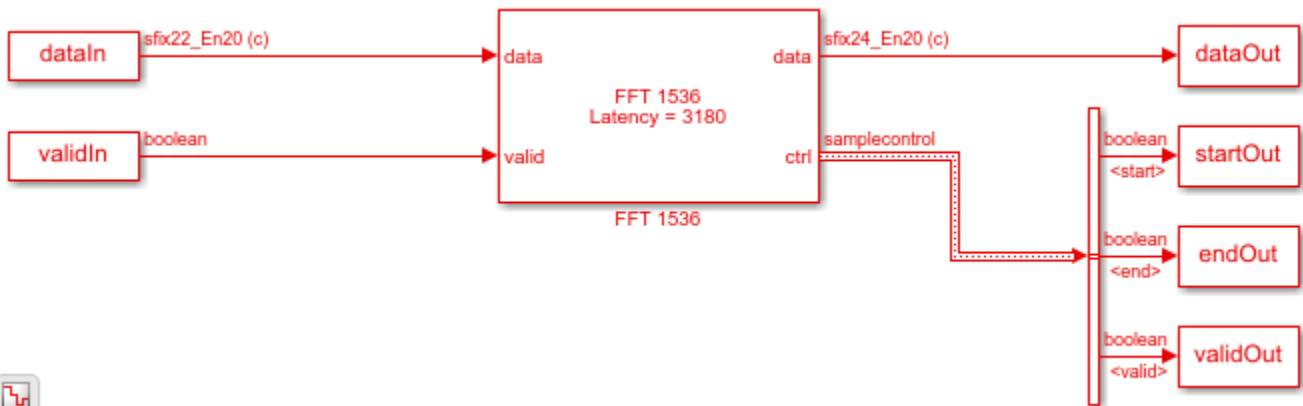
```
SampleTime = 4.3e-8; % 1/23.04e6;
data = rx(:);
valid = true(1536*14,1);
data = fi(data,1,22,20);
```

```
dataIn = timeseries(data,(0:length(data)-1).*SampleTime);
validIn = timeseries(valid,(0:length(valid)-1).*SampleTime);
```

```
FFT1536Latency = 3180;
```

```
NofClks = FFT1536Latency+length(data); % number of simulation clock cycles
StopTime = (NofClks)*SampleTime;
```

```
open_system HDLFFT1536model;
sim HDLFFT1536model;
```



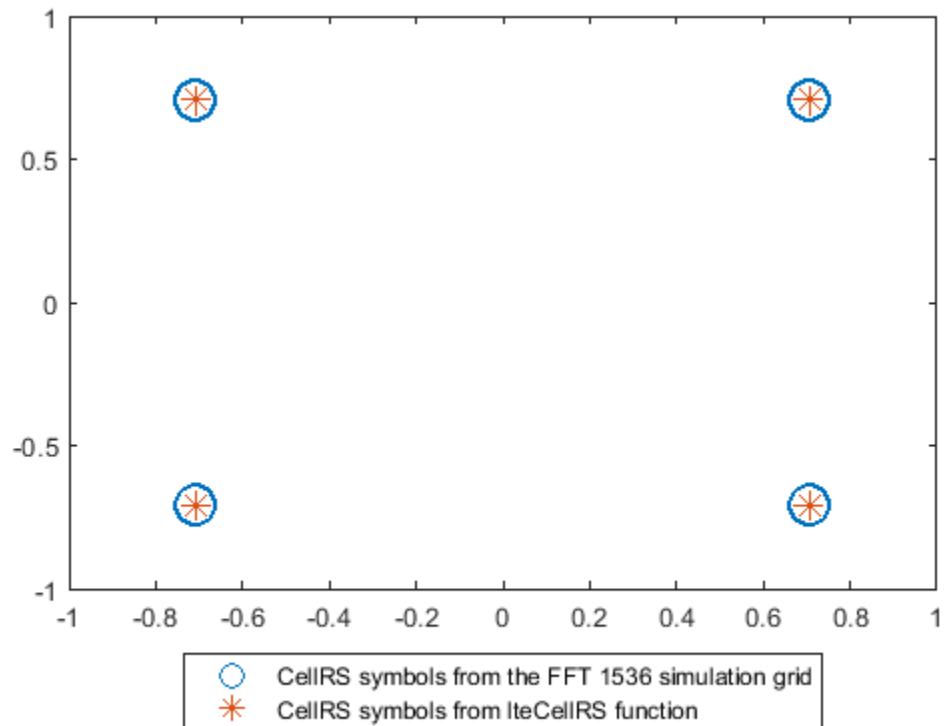
```
simOut = dataOut(validOut);
simOut = double(simOut(:)*1536);
```

Form the resource grid and remove the DC subcarrier.

```
fftOut = fftshift(reshape(simOut,1536,14));
resourceGrid = fftOut(318+1:318+1+900,:);
resourceGrid(900/2+1,:) = [];
```

Compare the CellRS symbols from the grid with the symbols returned from the `lteCellRS` function.

```
cellRS = lteCellRS(cfg);
cellRSIndices = lteCellRSIndices(cfg);
simCellRS = resourceGrid(cellRSIndices);
figure;
plot(real(simCellRS),imag(simCellRS),'o','MarkerSize',15);
hold on;
plot(real(cellRS),imag(cellRS),'*','MarkerSize',10)
legend('CellRS symbols from the FFT 1536 simulation grid'...
,'CellRS symbols from lteCellRS function','Location','southoutside')
axis([-1 1 -1 1]);
```



To generate HDL code for the FFT 1536 block, you must have an HDL Coder™ license. To generate HDL code from the FFT 1536 block in this model, right-click the block and select Create Subsystem from Selection. Then right-click the subsystem and select HDL Code > Generate HDL Code for Subsystem.

## See Also

### Blocks

FFT 1536



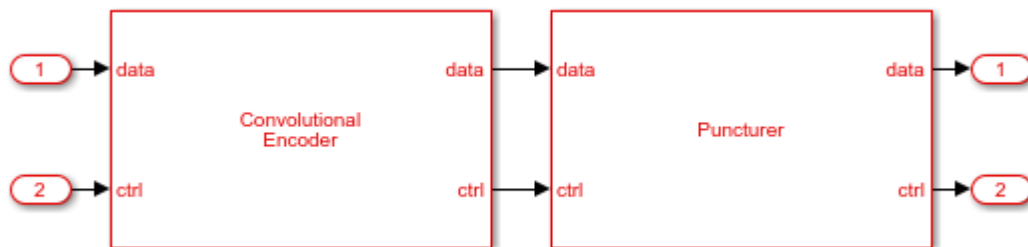
## Convolutional Encode and Puncture Streaming Samples

This example shows how to use the hardware-friendly Convolutional Encoder and Puncturer blocks to encode samples at WLAN code rates.

- 1 Generate random input frame samples with frame control signals by using the `whdlFramesToSamples` function in MATLAB®.
- 2 Import these samples into a Simulink® model and run the model to encode and puncture the samples.
- 3 Export the result of the Simulink simulation back to MATLAB.
- 4 Generate reference samples using the `convenc` MATLAB function with puncturing enabled.
- 5 Compare the Simulink results with the reference samples.

The example model supports HDL code generation for the `EncodeAndPuncture` subsystem, that contains the Convolutional Encoder and Puncturer blocks.

```
modelName = 'GenConvEncPuncturerModel';
open_system(modelName);
```



Set up workspace variables that describe the code rate. The Convolutional Encoder block supports constraint lengths in the range [3,9] and polynomial lengths in the range [2,7].

Starting from a code rate of 1/2, IEEE 802.11 WLAN specifies three puncturing patterns to generate three additional code rates. Choose one of these code rates, and then set the frame size and puncturing pattern based on that rate. You can also choose the unpunctured code rate of 1/2.

IEEE 802.11 WLAN specifies different code rates and uses 'Terminated' mode. The blocks also support 'Continuous' mode and 'Truncated' modes, but they are not included in this example.

```
constraintLength = 7;
codeGenerator = [133 171];

trellis = poly2trellis(constraintLength,...
    codeGenerator);

% IEEE 802.11n-2009 WLAN 1/2 (7, [133 171])
% Rate   Puncture Pattern   Maximum Frame Size
% 1/2    [1;1;1;1]                2592
% 2/3    [1;1;1;0]                1728
% 3/4    [1;1;1;0;0;1]           1944
% 5/6    [1;1;1;0;0;1;1;0;0;1]   2160
codeRate = 3/4;
if (codeRate == 2/3)
    puncVector = logical([1;1;1;0]);
```

```
    frameSize = 1728;
elseif (codeRate == 3/4)
    puncVector = logical([1;1;1;0;0;1]);
    frameSize = 1944;
elseif (codeRate == 5/6)
    puncVector = logical([1;1;1;0;0;1;1;0;0;1]);
    frameSize = 2160;
else % codeRate == 1/2
    puncVector = logical([1;1;1;1]);
    frameSize = 2592;
end
```

Generate input frame samples for encoding and puncturing by using Communications Toolbox™ System objects to generate encoded samples.

```
numFrames = 5;

txMessages = cell(1,numFrames);
txCodeword = cell(1,numFrames);

for ii = 1:numFrames
    txMessages{ii} = logical(randn(frameSize-constraintLength+1,1));
end
```

Set up variables for Simulink simulation. The Simulink model requires streaming samples with accompanying control signals. Calculate the required simulation time from the latency of the Convolutional Encoder and Puncturer blocks.

```
sampleSizeIn = 1;
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames = constraintLength-1;
[sampleIn,ctrlIn] = whdlFramesToSamples(txMessages, ...
    idleCyclesBetweenSamples,idleCyclesBetweenFrames,sampleSizeIn);

startIn = ctrlIn(:,1);
endIn = ctrlIn(:,2);
validIn = ctrlIn(:,3);

simTime = size(ctrlIn,1)+6;
sampleTime = 1;
```

Run the Simulink model.

```
set_param([modelName '/EncodeAndPuncture'], 'Open', 'on');
sim(modelName);
```

Convert the streaming samples from the Simulink block output to framed data for comparison.

```
sampleOut = squeeze(sampleOut);
startOut = ctrlOut(:,1);
endOut = ctrlOut(:,2);
validOut = ctrlOut(:,3);

idxStart = find(startOut.*validOut);
idxEnd = find(endOut.*validOut);
```

Generate reference samples using `convenc` MATLAB function.

```

for ii = 1:numFrames
    txCodeword{ii} = convenc([txMessages{ii};false(constraintLength-1,1)],...
        trellis,puncVector);
end

```

Compare the output samples against the generated input samples.

```

fprintf('\nEncoded Samples\n');
for ii = 1:numFrames
    idx = idxStart(ii):idxEnd(ii);
    idxValid = (validOut(idx));
    dataOut = sampleOut(:,idx);
    hdlTxCoded = dataOut(:,idxValid);
    numBitsErr = sum(xor(txCodeword{ii},hdlTxCoded(:)));
    fprintf('Number of samples mismatched in the frame #%d: %d bits\n',ii,numBitsErr);
end

```

Encoded Samples

```

Number of samples mismatched in the frame #1: 0 bits
Number of samples mismatched in the frame #2: 0 bits
Number of samples mismatched in the frame #3: 0 bits
Number of samples mismatched in the frame #4: 0 bits
Number of samples mismatched in the frame #5: 0 bits

```

## See Also

### Blocks

Convolutional Encoder | Puncturer

## OFDM Demodulation of Streaming Samples

This example shows how to use the OFDM Demodulator block to demodulate complex time-domain OFDM samples to subcarriers for a vector input. This example model supports HDL code generation for the OFDMDemod subsystem.

### Set up input data parameters

```
rng('default');
numOFDMSym = 2;
maxFFTLen = 128;
DCRem = true;
RoundingMethod = 'floor';
Normalize = false;
cpFraction = 1;
fftLen = 64;
cpLen = 16;
numLG = 6;
numRG = 5;
if DCRem
    NullInd = [1:numLG fftLen/2+1 fftLen-numRG+1:fftLen];
else
    NullInd = [1:numLG fftLen-numRG+1:fftLen]; %#ok<UNRCH>
end
symbOffset = floor(cpFraction*cpLen);
vecLen = 2;
```

### Generate frames of random input samples

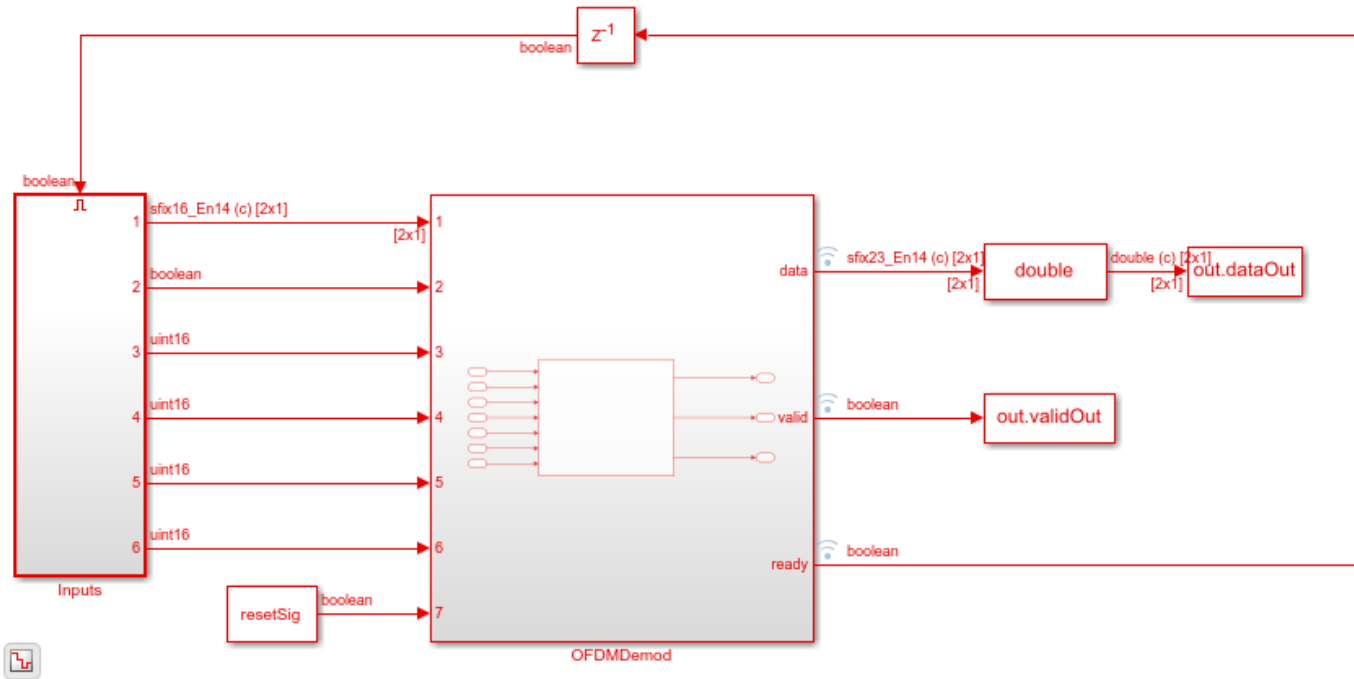
```
data = randn(fftLen,numOFDMSym)+1i*randn(fftLen,numOFDMSym);
dataIn = ofdmmod(data,fftLen,cpLen);
```

### Convert the framed input data to a stream of samples and import the input stream to Simulink®

```
data = dataIn(:);
valid = true(length(dataIn)/vecLen,1);
fftSig = fftLen*ones(length(dataIn),1);
CPSig = cpLen*ones(length(dataIn),1);
LGSig = numLG*ones(length(dataIn),1);
RGSig = numRG*ones(length(dataIn),1);
resetSig = false(length(data),1);
sampleTime = 1/vecLen;
stopTime = (maxFFTLen*3*numOFDMSym)/vecLen;
```

### Run the Simulink model

```
modelName = 'genhdlOFDMDemodulatorModel';
open_system(modelName);
out = sim(modelName);
```



### Export the stream of demodulated samples of the Simulink block to the MATLAB® workspace

```
simOut = squeeze(out.dataOut(:,1,out.validOut==1));
```

### Demodulate random input samples using ofdm demod\_baseline function

```
[dataOut1] = ofdm demod_baseline(dataIn,fftLen,cpLen,symbOffset,NullInd.',[],Normalize,RoundingMethod);
matOut = dataOut1(:);
```

### Compare the output of the Simulink model against the output of ofdm demod\_baseline function

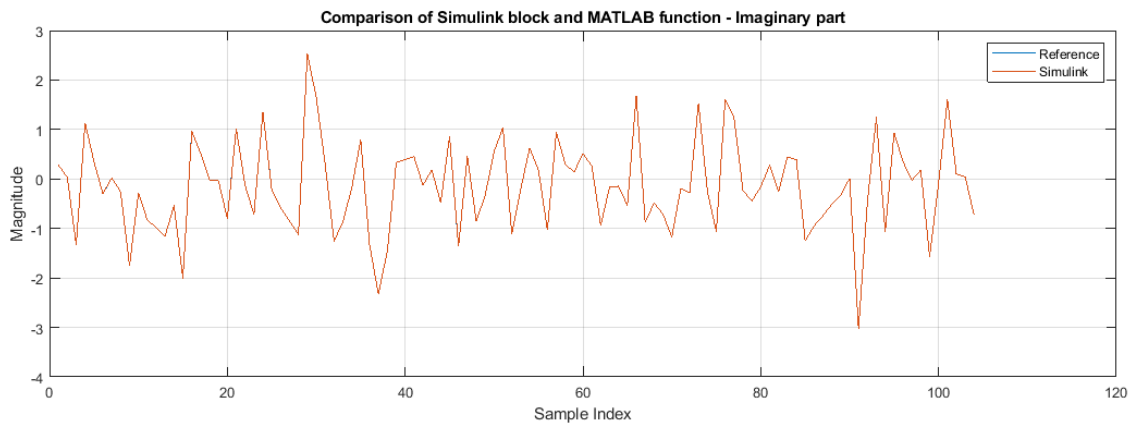
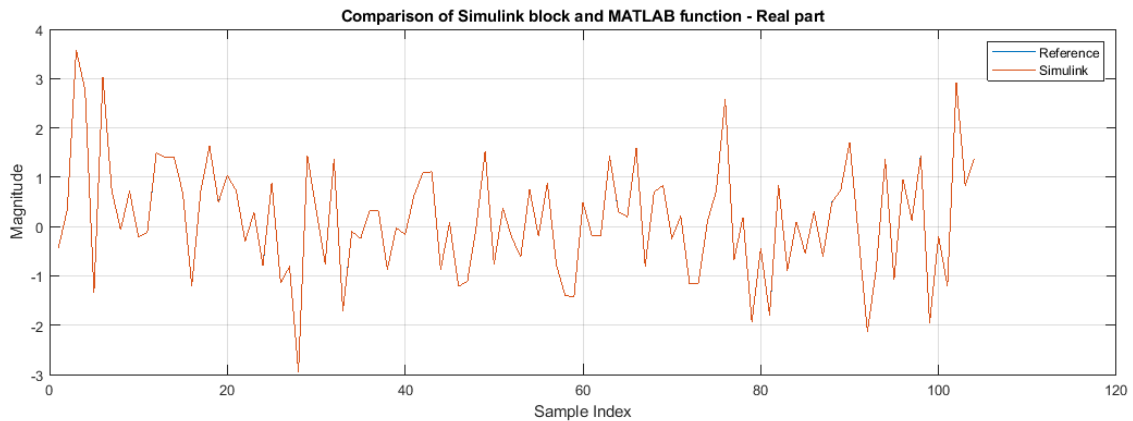
```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(matOut(:)));
hold on;
plot(real(simOut(:)));
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function - Real part')
```

```
subplot(2,1,2)
plot(imag(matOut(:)));
hold on;
plot(imag(simOut(:)));
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function - Imaginary part')
```

```
sqnrRealdB=10*log10(var(real(simOut(:)))/abs(var(real(simOut(:)))-var(real(matOut(:)))));
sqnrImagdB=10*log10(var(imag(simOut(:)))/abs(var(imag(simOut(:)))-var(imag(matOut(:)))));
```

```
fprintf('\n OFDM Demodulator: \n SQNR of real part is %.2f dB',sqnrRealdB);
fprintf('\n SQNR of imaginary part is %.2f dB\n',sqnrImagdB);
```

```
OFDM Demodulator:
SQNR of real part is 47.77 dB
SQNR of imaginary part is 42.69 dB
```



## See Also

### Blocks

OFDM Demodulator

## Decode and recover message from RS codeword

This example shows how to use RS Decoder block to decode and recover a message from a Reed-Solomon (RS) codeword. In this example, a set of random inputs are generated and provided to the `comm.RSEncoder` function and its output is provided to the RS Decoder block. The output of the RS Decoder block is compared with the input of the `comm.RSEncoder` function to check whether any errors are encountered. The example model supports HDL code generation for the RS Decoder subsystem.

### Set up input data parameters

```
n = 255;
k = 239;
primPoly = [1 0 0 0 1 1 1 0 1];
B = 1;
nMessages = 4;
data = zeros(k,nMessages);
inputMsg = (zeros(n,nMessages));
startSig = [];
endSig = [];
```

### Generate random input samples

Generate random samples based on `n`, `k`, and `m` values and provide them as input to the `comm.RSEncoder` function. Here, `n` is the codeword length, `k` is the message length, and `m` is the gap between the frames.

```
hRSEnc = comm.RSEncoder;
hRSEnc.CodewordLength = n;
hRSEnc.MessageLength = k;
m=0;

for ii = 1:nMessages
    data(:,ii) = randi([0 n],k,1);
    [inputMsg(1:n,ii)] = hRSEnc(data(:,ii));
    inputMsg1(1:n,ii) = inputMsg(1:n,ii);
    [inputMsg(n+1:n+m,ii)] = zeros(m,1);
    validIn(1:n,ii) = true;
    validIn(n+1:n+m) = false;

endSig = [endSig [false(n-1,1); true;false(m,1)];];
startSig = [startSig [true;false(n+m-1,1)]];
```

```
end
```

```
refOutput = data(:);
```

### Import the encoded random input samples to the Simulink® model

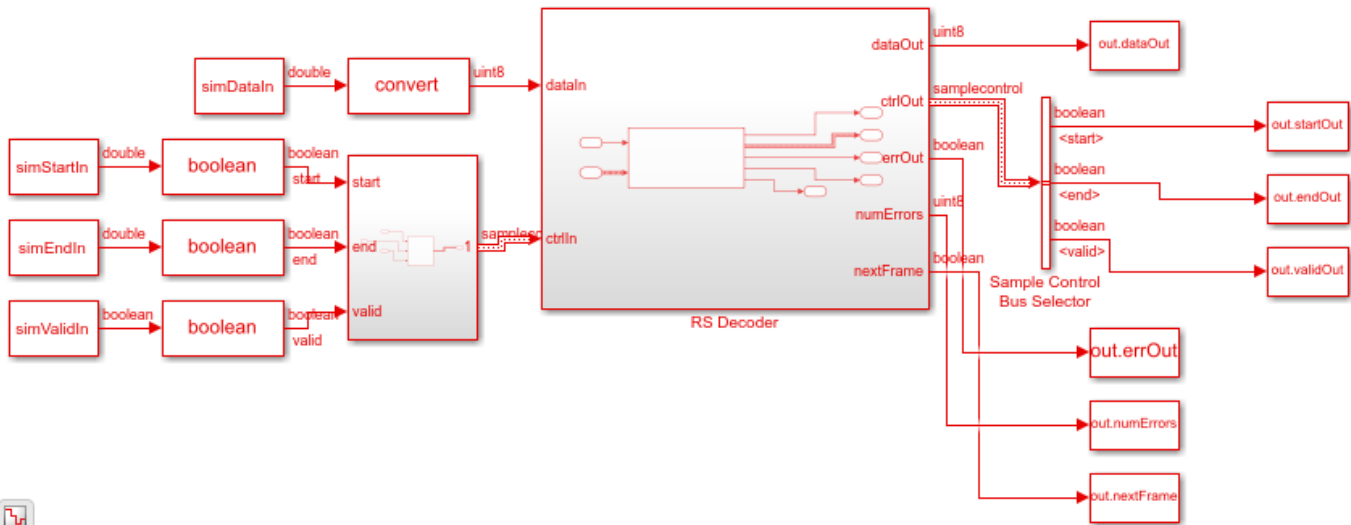
The output of the `comm.RSEncoder` function is provided as input to the Simulink block.

```
simDataIn = inputMsg(:);
simStartIn = startSig(:);
simEndIn = endSig(:);
simValidIn = validIn(:);
```

### Run the Simulink model

```

modelName = 'RSDecoder';
open_system(modelname);
out = sim(modelname);
    
```



### Export the decodes samples of the Simulink block to the MATLAB® workspace.

```
simOutput = out.dataOut(out.validOut);
```

### Compare the output of the Simulink block with the inputs provided to the comm.RSEncoder function

```

fprintf('\nHDL RS Decoder\n');
difference = double(simOutput) - double(refOutput);
fprintf('\nTotal number of samples differed between Simulink block output and MATLAB function output is: %d\n', difference);
    
```

HDL RS Decoder

Total number of samples differed between Simulink block output and MATLAB function output is: 0

### See Also

#### Blocks

RS Decoder



## LDPC Encode and Decode of Streaming Data

This example shows how to simulate the NR LDPC Encoder and NR LDPC Decoder Simulink® blocks and compare the hardware-optimized results with the results from the 5G Toolbox™ functions. These blocks support scalar and vector inputs. The NR LDPC Decoder block enables you to select either Min-sum or Normalized min-sum algorithm for decoding operation.

### Generate Input Data for Encoder

Choose a series of input values for `bgn` and `liftingSize` according to the 5G NR standard. Generate the corresponding input vectors for the selected base graph number (`bgn`) and `liftingSize` values. Generate random frames of input data and convert them to Boolean data and control signal that indicates the frame boundaries. `encFrameGap` accommodates the latency of the NR LDPC Encoder block for `bgn` and `liftingSize` values. Use the `nextFrame` signal to determine when the block is ready to accept the start of the next input frame.

```
bgn          = [0; 1; 1; 0];
liftingSize  = [4; 384; 144; 208];
numFrames   = 4;
serial      = false; % {false,true};

encbgnIn    = [];encliftingSizeIn = [];
msg         = {numFrames};
K           = [];N = [];
encSampleIn = [];encStartIn = [];encEndIn = [];encValidIn = [];
encFrameGap = 2500;
for ii = 1:numFrames
    if bgn(ii) == 0
        K(ii) = 22;
        N(ii) = 66;
    else
        K(ii) = 10;
        N(ii) = 50;
    end
    frameLen = liftingSize(ii) * K(ii);
    msg{ii} = randi([0 1],1,frameLen);
    if serial
        len = K(ii) * liftingSize(ii);
        encFrameGap = liftingSize(ii) * N(ii) + 2500;
    else
        len = K(ii) * ceil(liftingSize(ii)/64); %#ok<*UNRCH>
        encFrameGap = 2500;
    end

    encIn = ldpc_dataFormation(msg{ii},liftingSize(ii),K(ii),serial);

    encSampleIn = logical([encSampleIn encIn zeros(size(encIn,1),encFrameGap)]); %#ok<*AGROW>
    encStartIn  = logical([encStartIn  1 zeros(1,len-1) zeros(1,encFrameGap)]);
    encEndIn    = logical([encEndIn    zeros(1,len-1) 1 zeros(1,encFrameGap)]);
    encValidIn  = logical([encValidIn  ones(1,len) zeros(1,encFrameGap)]);
    encbgnIn    = logical([encbgnIn    repmat(bgn(ii),1,len) zeros(1,encFrameGap)]);
    encliftingSizeIn = uint16([encliftingSizeIn repmat(liftingSize(ii),1,len) zeros(1,encFrameGap)]);
end

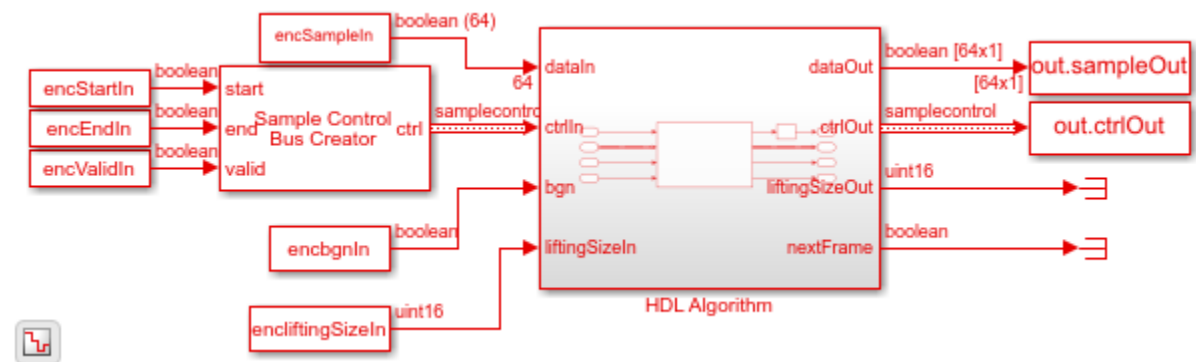
encSampleIn = timeseries(logical(encSampleIn'));
```

```
sampleTime = 1;
simTime = length(encValidIn); %#ok<NASGU>
```

### Run Encoder Model

The HDL Algorithm subsystem contains the NR LDPC Encoder block. Running the model imports the input signal variables `encSampleIn`, `encStartIn`, `encEndIn`, `encValidIn`, `encbgnIn`, `encliftingSizeIn`, `sampleTime`, and `simTime` and exports `sampleOut` and `ctrlOut` variables to the MATLAB® workspace.

```
open_system('NRLDPCEncoderHDL');
encOut = sim('NRLDPCEncoderHDL');
```



### Verify Encoder Results

Convert the streaming data output of the block to frames and then compare them with the output of the `nrLDPCEncode` function.

```
startIdx = find(encOut.ctrlOut.start.Data);
endIdx = find(encOut.ctrlOut.end.Data);

for ii = 1:numFrames
    encHDL{ii} = ldpc_dataExtraction(encOut.sampleOut.Data, liftingSize(ii), startIdx(ii), endIdx(ii));
    encRef = nrLDPCEncode(msg{ii}', bgn(ii)+1);
    error = sum(abs(encRef - encHDL{ii}));
    fprintf(['Encoded Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'], ii, error);
end
```

```
Encoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

### Generate Input Data for Decoder

Use the encoded data from the NR LDPC Encoder block to generate input log-likelihood ratio (LLR) values for the NR LDPC Decoder block. Use channel, modulator, and demodulator system objects to add some noise to the signal. Again, create vectors of `bgn` and `liftingSize` and convert the frames of data to LLRs with a control signal that indicates the frame boundaries. `decFrameGap` accommodates the latency of the NR LDPC Decoder block for `bgn`, `liftingSize`, and number of iterations. Use the **nextFrame** signal to determine when the block is ready to accept the start of the next input frame.

```

nVar = 1.2;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);

algo = 'Normalized min-sum'; % { Min-sum, 'Normalized min-sum' };
if strcmpi(algo,'Min-sum')
    alpha = 1;
else
    alpha = 0.75;
end

numIter = 8;
decbgnIn = []; decliftingSizeIn = [];
rxLLR = {numFrames};
decSampleIn = []; decStartIn = []; decEndIn = []; decValidIn = [];

for ii=1:numFrames
    mod = bpskMod(double(encHDL{ii}));
    rSig = chan(mod);
    rxLLR{ii} = fi(bpskDemod(rSig),1,6,0);

    if serial
        len = N(ii)* liftingSize(ii);
        decFrameGap = numIter *7000 + liftingSize(ii) * K(ii);
    else
        len = N(ii)* ceil(liftingSize(ii)/64);
        decFrameGap = numIter *1200;
    end

    decIn = ldpc_dataFormation(rxLLR{ii}',liftingSize(ii),N(ii),serial);

    decSampleIn = [decSampleIn decIn zeros(size(decIn,1),decFrameGap)]; %#ok<*AGROW>
    decStartIn = logical([decStartIn 1 zeros(1,len-1) zeros(1,decFrameGap)]);
    decEndIn = logical([decEndIn zeros(1,len-1) 1 zeros(1,decFrameGap)]);
    decValidIn = logical([decValidIn ones(1,len) zeros(1,decFrameGap)]);
    decbgnIn = logical([decbgnIn repmat(bgn(ii),1,len) zeros(1,decFrameGap)]);
    decliftingSizeIn = uint16([decliftingSizeIn repmat(liftingSize(ii),1,len) zeros(1,decFrameGap)]);
end

decSampleIn = timeseries(fi(decSampleIn',1,6,0));

simTime = length(decValidIn);

```

### Run Decoder Model

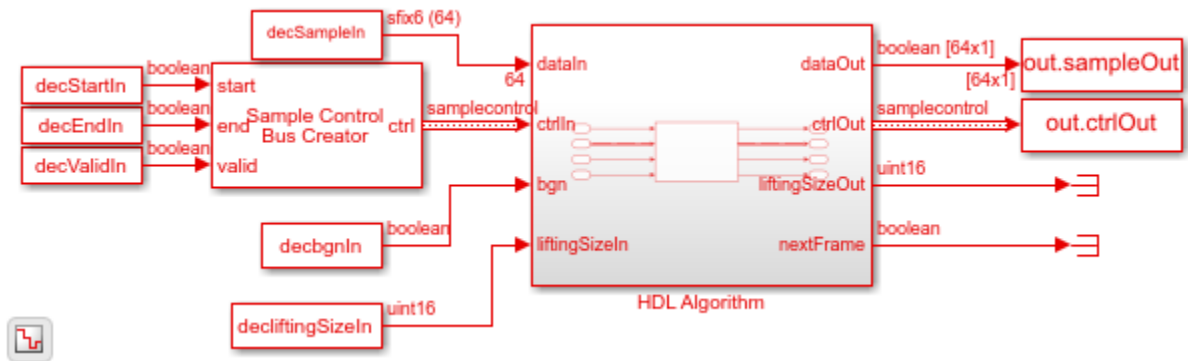
The HDL Algorithm subsystem contains the NR LDPC Decoder block. Running the model imports the input signal variables `decSampleIn`, `decStartIn`, `decEndIn`, `decValidIn`, `decbgnIn`, `decliftingSizeIn`, `numIter`, `sampleTime`, and `simTime` and exports a stream of decoded output samples `sampleOut` along with control signal `ctrlOut` to the MATLAB workspace.

```

open_system('NRLDPCDecoderHDL');
if alpha ~= 1
    set_param('NRLDPCDecoderHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Normalized min-sum')
else
    set_param('NRLDPCDecoderHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Min-sum');

```

```
end
decOut = sim('NRLDPCDecoderHDL');
```



### Verify Decoder Results

Convert the streaming data output of the block to frames and then compare them with the output of the `nrLDPCDecode` function.

```
startIdx = find(decOut.ctrlOut.start.Data);
endIdx = find(decOut.ctrlOut.end.Data);

for ii = 1:numFrames
    decHDL{ii} = ldpc_dataExtraction(decOut.sampleOut.Data, liftingSize(ii), startIdx(ii), endIdx(ii));
    decRef = nrLDPCDecode(double(rxLLR{ii}), bgn(ii)+1, numIter, 'Algorithm', 'Normalized min-sum', 'Termination', 'max');
    error = sum(abs(double(decRef) - decHDL{ii}));
    fprintf(['Decoded Frame %d: Behavioral and HDL simulation differ by %d bits\n'], ii, error);
end
```

```
Decoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

### See Also

#### Blocks

NR LDPC Decoder | NR LDPC Encoder

#### Functions

`nrLDPCDecode` | `nrLDPCEncode`

## Estimate Channel Using Input Data and Reference Subcarriers

This example shows how to use the OFDM Channel Estimator block to estimate a channel using input data and reference subcarriers. In this example model, the averaging and interpolation features are enabled. The HDL Algorithm subsystem in this example model supports HDL code generation.

### Set Input Data Parameters

Set up these workspace variables for the model to use. You can modify these values according to your requirement.

```
rng('default');
numOFDMSym = 980;
numOFDMSymToBeAvg = 14;
interpolFac = 3;
maxNumScPerSym = 72;
numOFDMSymPerFrame = 140;
numScPerSym = 72;
```

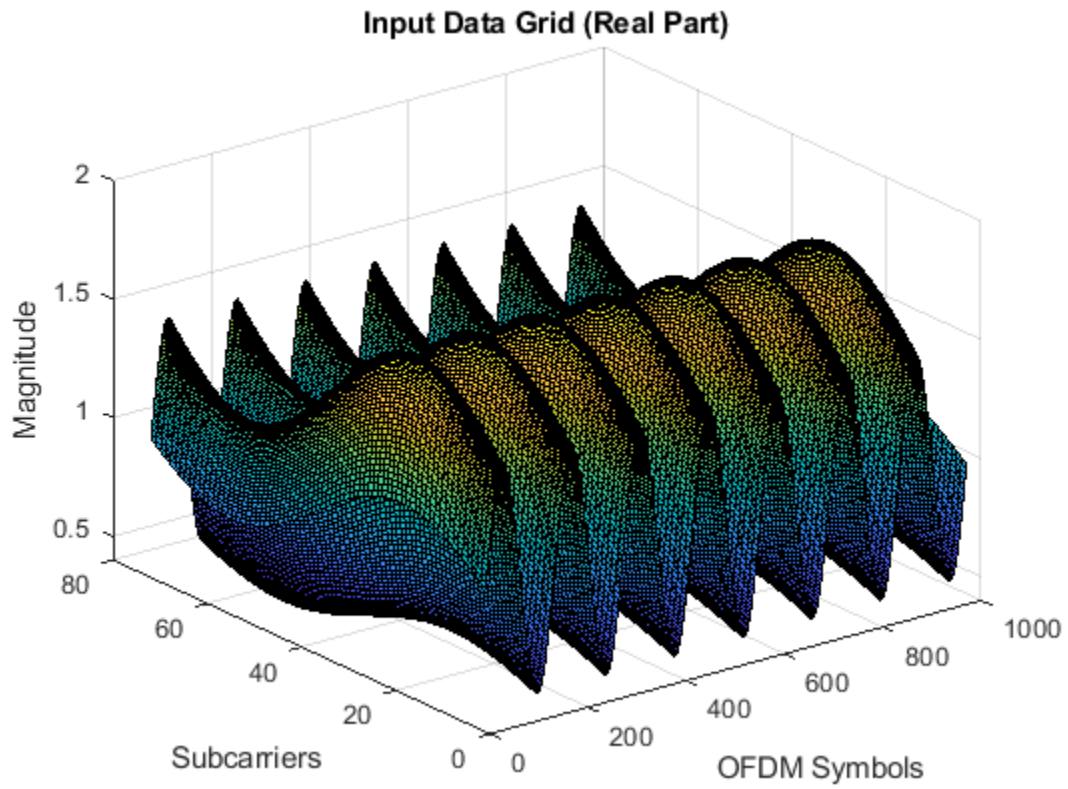
### Generate Sinusoidal Input Data Subcarriers

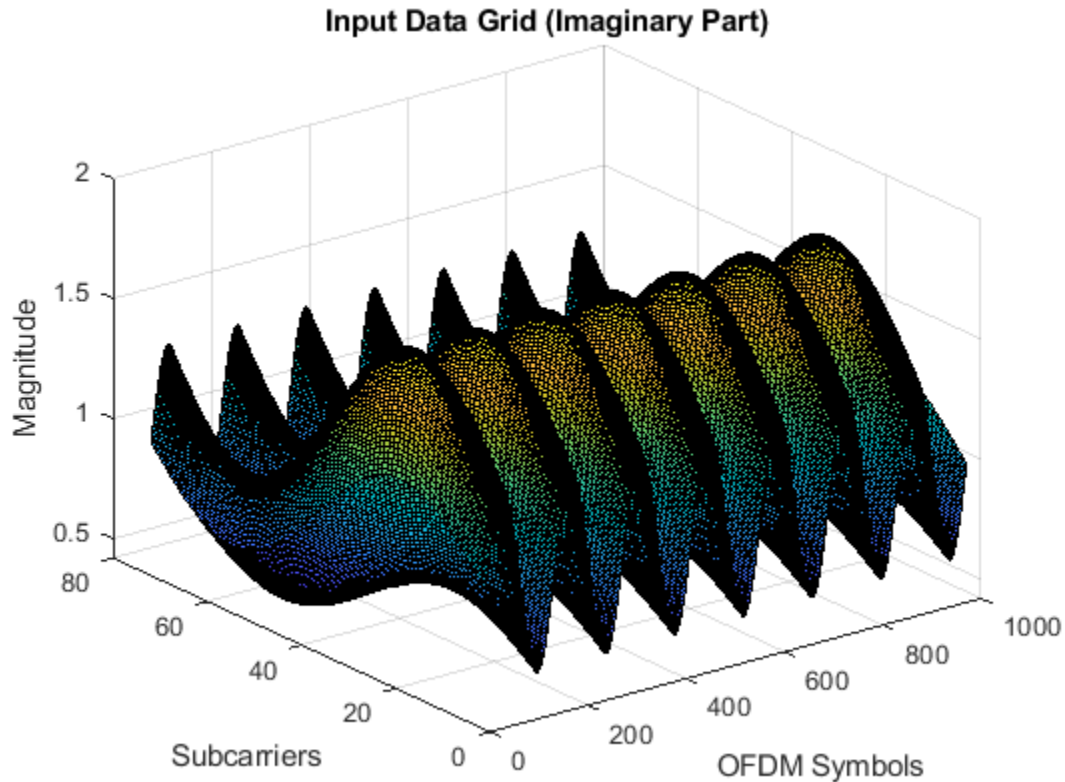
Use the numScPerSym and numOFDMSym variables to generate complex sinusoidal input data subcarriers with their real and imaginary parts generated separately.

```
dataInGrid = zeros(numScPerSym,numOFDMSym);
for numScPerSymCount = 0:numScPerSym - 1
    for numOFDMSymCount = 0:numOFDMSym - 1
        realXgain = 1 + .2*sin(2*pi*numScPerSymCount/numScPerSym);
        realYgain = 1 + .5*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        imagXgain = 1 + .3*sin(2*pi*numScPerSymCount/numScPerSym);
        imagYgain = 1 + .4*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        dataInGrid(numScPerSymCount+1,numOFDMSymCount+1) = realXgain*realYgain + 1i*(imagXgain*imagYgain);
    end
end
validIn = true(1,length(dataInGrid(:)));

figure(1);
surf(real(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Real Part)')

figure(2);
surf(imag(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Imaginary Part)')
```





### Generate Reference Data Subcarriers

Generate reference data subcarriers.

```
refDataIn = randsrc(size(dataInGrid(:,1)),size(dataInGrid(:,2)),[1 1]);
refValidIn = boolean(zeros(1,numOFDMSym*numScPerSym));
startRefValidIndex = randi(interpolFac,1,1);
for numOFDMSymCount = 1:numOFDMSym
    refValidIn(startRefValidIndex+(numOFDMSymCount-1)*numScPerSym:interpolFac:numScPerSym*numOFDMSymCount-1) = true;
end
```

### Generate Signal with Number of Subcarriers per Symbol

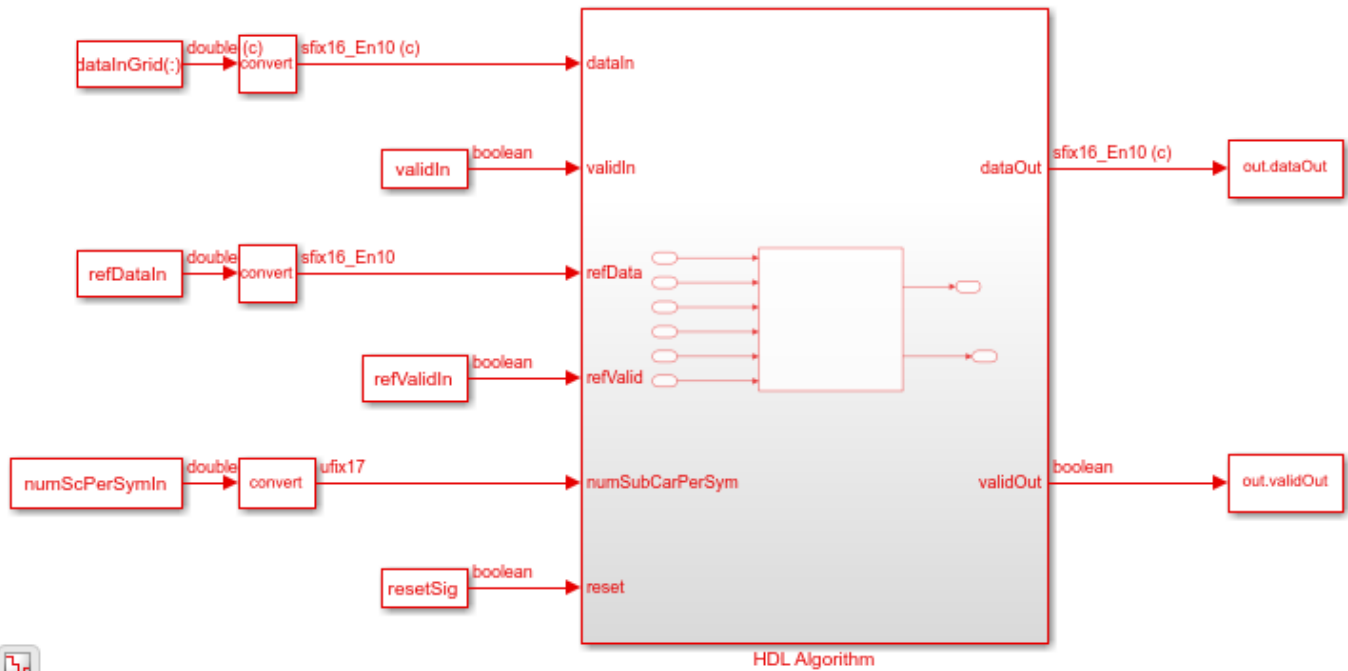
Generate a signal with the number of subcarriers per symbol.

```
numScPerSymIn = numScPerSym>true(1,length(dataInGrid(:)));
resetSig = false(1,length(dataInGrid(:)));
```

### Run Simulink® Model

Run the model. Running the model imports the input signal variables from the MATLAB workspace to the OFDM Channel Estimator block in the model.

```
modelName = 'genhdlOFDMChannelEstimatorModel';
open_system(modelName);
out = sim(modelName);
```



### Export Stream of Channel Estimates from Simulink to MATLAB® Workspace

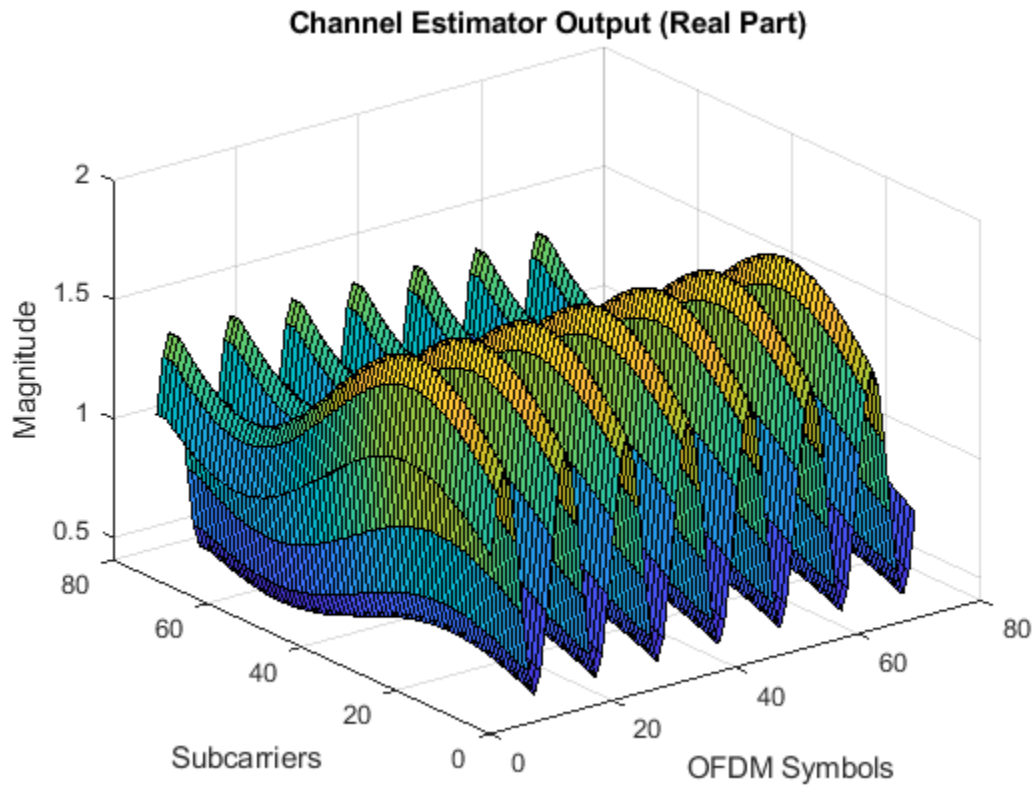
Export the output of the OFDM Channel Estimator block to the MATLAB® workspace. Plot the real part and imaginary part of the exported block output.

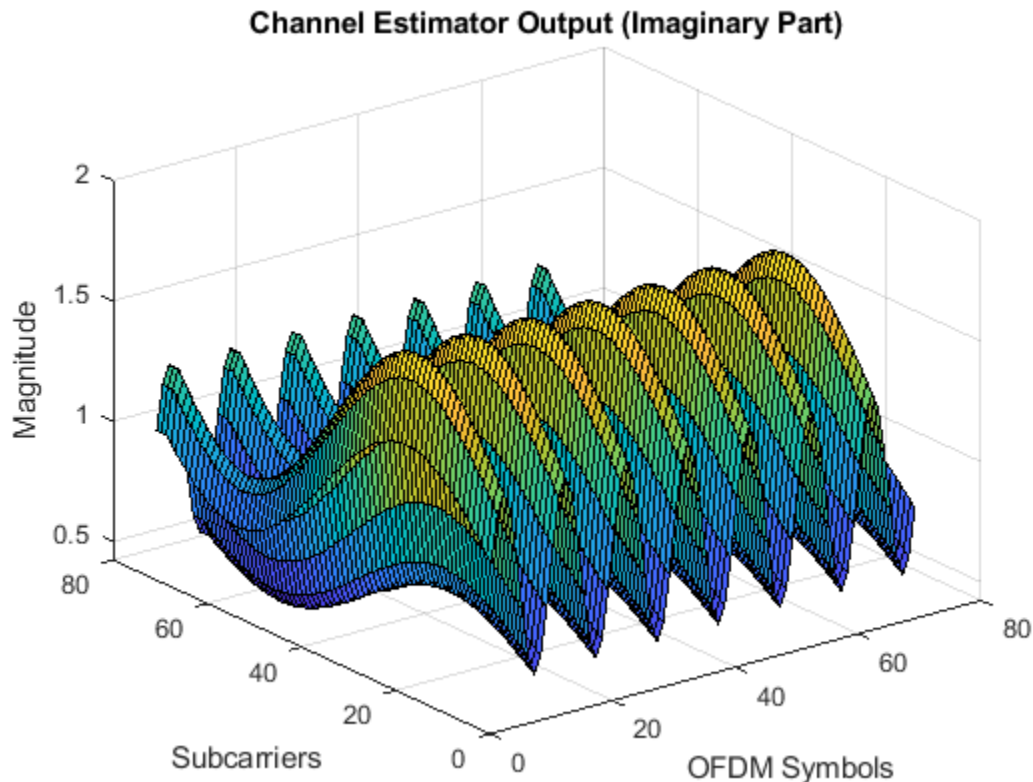
```
simOut = out.dataOut.Data(out.validOut.Data);
N = length(simOut) - mod(length(simOut), numScPerSym);
temp = simOut(1:N);
channelEstimateSimOut = reshape(temp, numScPerSym, length(temp)/numScPerSym);
```

```
figure(3);
surf(real(channelEstimateSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Channel Estimator Output (Real Part)')
```

```
figure(4);
surf(imag(channelEstimateSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Channel Estimator Output (Imaginary Part)')
```







### Estimate Channel Using MATLAB® Function

Estimate the channel by using the `channelEstReference` function with the sinusoidal input data subcarriers.

```
dataOut1 = channelEstReference(...
    numOFDMSymToBeAvg,interpolFac,numScPerSym,numOFDMSym, ...
    dataInGrid(:),validIn,refDataIn,refValidIn,numScPerSymIn);
matlabOut = dataOut1(:);
matOut = zeros(numel(matlabOut)*numScPerSym,1);
for ii= 1:numel(matlabOut)
loadArray = [matlabOut(ii).dataOut; zeros((numel(matlabOut)-1)*numScPerSym,1)];
shiftArray = circshift(loadArray,(ii-1)*numScPerSym);
matOut = matOut + shiftArray;
end
```

### Compare Simulink Block Output with MATLAB® Function Output

Compare the OFDM Channel Estimator block output with `channelEstReference` function output. Plot the output comparison as a real part and an imaginary part using separate plots.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(matOut(:)));
hold on;
plot(real(simOut(:)));
grid on
legend('MATLAB Reference Output','Simulink Block Output')
```

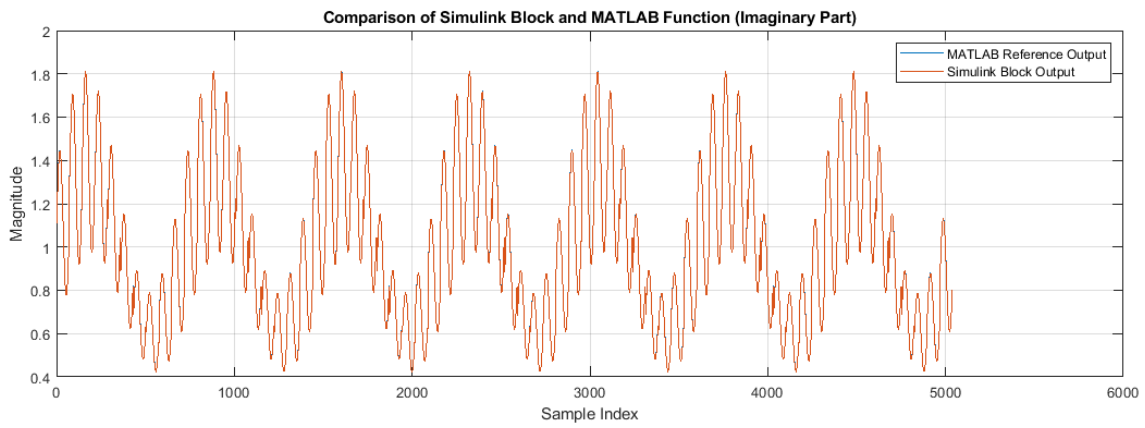
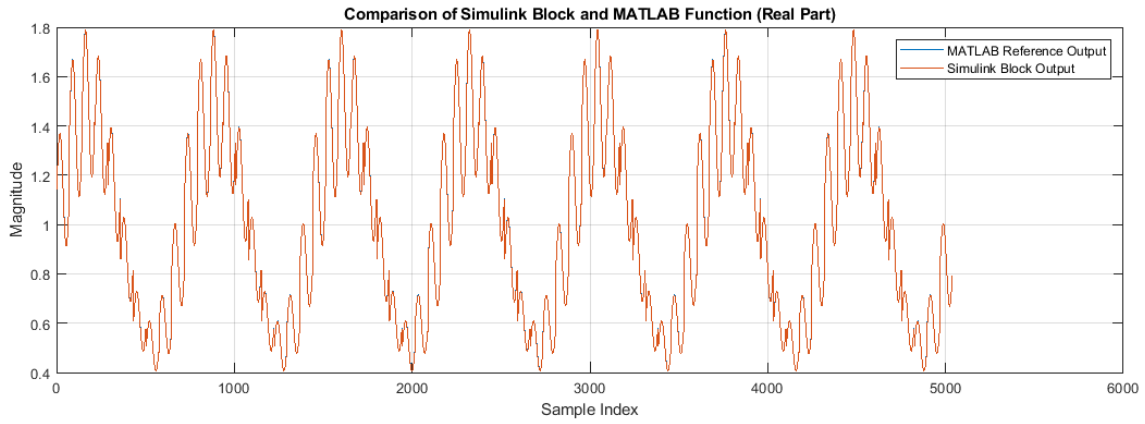
```
xlabel('Sample Index')
ylabel('Magnititude')
title('Comparison of Simulink Block and MATLAB Function (Real Part)')

subplot(2,1,2)
plot(imag(matOut(:)));
hold on;
plot(imag(simOut(:)));
grid on
legend('MATLAB Reference Output','Simulink Block Output')
xlabel('Sample Index')
ylabel('Magnititude')
title('Comparison of Simulink Block and MATLAB Function (Imaginary Part)')

sqrRealdB = 10*log10(double(var(real(simOut(:)))/abs(var(real(simOut(:)))-var(real(matOut(:))))
sqrImagdB = 10*log10(double(var(imag(simOut(:)))/abs(var(imag(simOut(:)))-var(imag(matOut(:))))

fprintf('\n OFDM Channel Estimator \n SQNR of real part: %.2f dB',sqrRealdB);
fprintf('\n SQNR of imaginary part: %.2f dB\n',sqrImagdB);

OFDM Channel Estimator
SQNR of real part: 38.54 dB
SQNR of imaginary part: 37.77 dB
```



## See Also

### Blocks

OFDM Channel Estimator

## Modulate and Demodulate OFDM Streaming Samples

This example model shows how to simulate OFDM Modulator block and Demodulator blocks. In this model, an OFDM Modulator and an OFDM Demodulator block are connected back-to-back. The **OFDM parameters source** parameter in these blocks is set to `Input port`, enabling you to dynamically change the input values of these blocks. You can change these values using the script in this example. These blocks support scalar and vector inputs. To verify the functionality of these blocks, the input provided to the OFDM Modulator block is compared with the output of the OFDM Demodulator block. The OFDMModDemod HDL subsystem in this example model supports HDL code generation.

### Set Input Data Parameters

Set up these workspace variables for the model to use. You can modify these values according to your requirement. The example model uses these workspace variables `dataIn`, `validIn`, `fftLen`, `maxFFTLen`, `cpLen`, `numLG`, `numRG`, `numSymb`, and `DCNull` to configure the OFDM Modulator and OFDM Demodulator blocks.

```
fftLen = 64;
maxFFTLen = 128;
cpLen = 16;
numLG = 6;
numRG = 5;
numSymb = 2;
DCNull = 1; % 1 or 0
vecLen = 8; % 1, 2, 4, 8, 16, 32, or 64
if DCNull==1
    numActData = fftLen - (numLG+numRG+1);
else
    numActData = fftLen - (numLG+numRG);
end
```

### Generate Input Data Frames

Generate random frames of complex input data and a control signal that indicates the frame boundaries.

```
rng default;
dataIn = complex(randn(numActData*numSymb,1),randn(numActData*numSymb,1));
dataVec = []; % Store data arranged in vector form
presentSymbDataStartIndex = 0;
for ii = 1:numSymb
    counter = 0;
    for jj = 1:ceil(numActData/vecLen)
        if jj == ceil(numActData/vecLen)
            numZerosToBeAppended = vecLen - (numActData-counter);
            dataVec = [dataVec [dataIn(presentSymbDataStartIndex+counter+(1:vecLen-numZerosToBeAppended))]];
        else
            dataVec = [dataVec dataIn(presentSymbDataStartIndex+counter+(1:vecLen))];
        end
        counter = counter + vecLen;
    end
    presentSymbDataStartIndex = presentSymbDataStartIndex + numActData;
end
data = dataVec.';
```

```
valid = boolean(ones(size(data,1),1)); % Valid signal generation
```

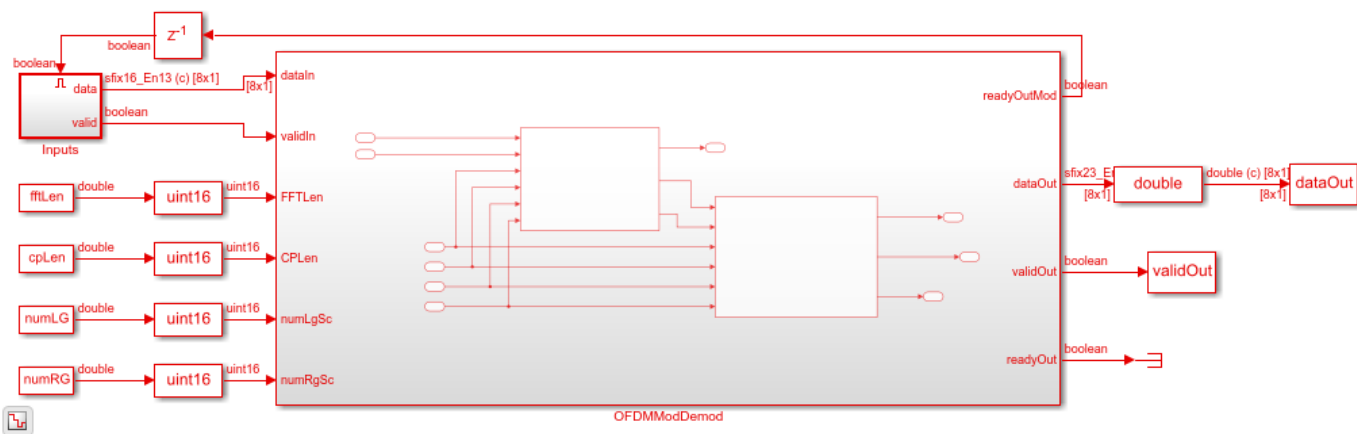
```
sampling_time = 1;
stoptime = maxFFTLen*6*numSymb;
```

### Run the Simulink® Model

Run the model to import the input signal variables `dataIn`, `validIn`, `fftLen`, `maxFFTLen`, `cpLen`, `numLG`, `numRG`, `numSymb`, and `DCNull` from the workspace to the OFDM Modulator block. The OFDM Modulator block returns OFDM-modulated output samples and a control signal. These OFDM-modulated samples are fed to the OFDM Demodulator block, which returns OFDM demodulated samples.

```
open_system('genhdlOFDMModDemodExample')
sim('genhdlOFDMModDemodExample');
```

```
% Store valid data from Simulink model
dataOut1 = dataOut.data;
simOut = dataOut1(:,:,validOut);
simOut = simOut(:);
```



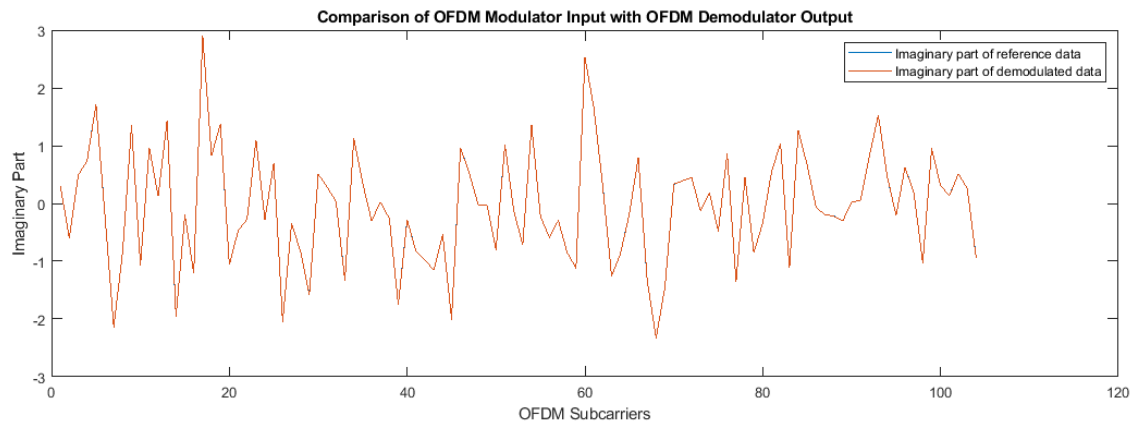
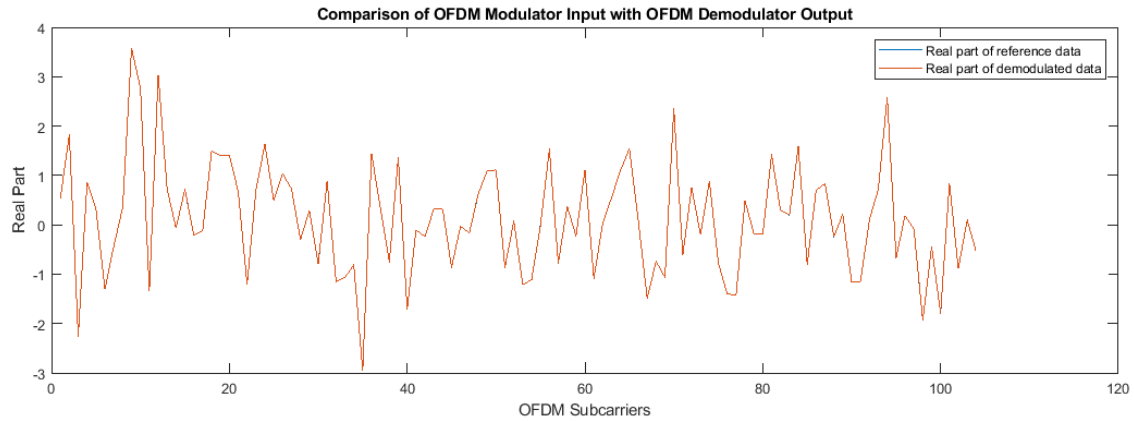
### Compare OFDM Modulator Input with OFDM Demodulator Output

Compare the input data provided to the OFDM Modulator block with the output data generated from the OFDM Demodulator block.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(dataIn(1:size(simOut))));
hold on
plot(squeeze(real(simOut)));
legend('Real part of reference data','Real part of demodulated data');
title('Comparison of OFDM Modulator Input with OFDM Demodulator Output');
xlabel('OFDM Subcarriers');
ylabel('Real Part');

subplot(2,1,2)
plot(imag(dataIn(1:size(simOut))));
hold on
plot(squeeze(imag(simOut)))
legend('Imaginary part of reference data','Imaginary part of demodulated data');
```

```
title('Comparison of OFDM Modulator Input with OFDM Demodulator Output');  
xlabel('OFDM Subcarriers');  
ylabel('Imaginary Part');
```



## See Also

### Blocks

OFDM Modulator | OFDM Demodulator

## Polar Encode and Decode of Streaming Samples

This example shows how to simulate the NR Polar Encode and Decode blocks and compare the hardware-optimized results with the results from 5G Toolbox™ functions.

### Generate Input Data for Encoder

Choose a series of input values for **K** and **E**. These values must be valid pairs supported by the 5G NR standard. Generate random frames of input data and add a CRC codeword. This example uses uplink mode, so each message has 11 CRC bits. Downlink messages have 24 CRC bits, and downlink DCI messages require prepending 1s to the frame.

Convert the message frames to streams of Boolean samples and control signals that indicate the frame boundaries. Generate input vectors of **K** and **E** values over time. The example model imports the workspace variables `encSampleIn`, `encCtrlIn`, `encKfi`, `encEfi`, `sampleTime`, and `simTime`.

For this example, the number of invalid cycles between frames is empirically chosen to accommodate the latency of the NR Polar Encoder block for the specified **K** and **E** values. When the values of **K** and **E** are larger than in this example, the number of invalid cycles between frames must be longer. Use the `nextFrame` output signal of the block to determine when the block is ready to accept the start of the next input frame.

```
K = [132; 132; 132; 54];
E = [256; 256; 256; 124];
numFrames = 4;
numCRCBits = 11;
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames = 500;
samplesPerCycle = 1;
btwSamples = false(idleCyclesBetweenSamples,1);
btwFrames = false(idleCyclesBetweenFrames,1);

encKfi = [];
encEfi = [];
dataIn = {numFrames};
for ii = 1:numFrames
    msg = randi([0 1],K(ii)-numCRCBits,1);
    msg = nrcrcEncode(msg,'11'); % CRC poly is '11' for uplink and '24C' for downlink
    encKfi = [encKfi;repmat([fi(K(ii),0,10,0);btwSamples],length(msg),1);btwFrames];
    encEfi = [encEfi;repmat([fi(E(ii),0,14,0);btwSamples],length(msg),1);btwFrames];
    dataIn{1,ii} = logical(msg);
end

[encSampleIn,encCtrlIn] = whdlFramesToSamples(...
    dataIn,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesPerCycle);

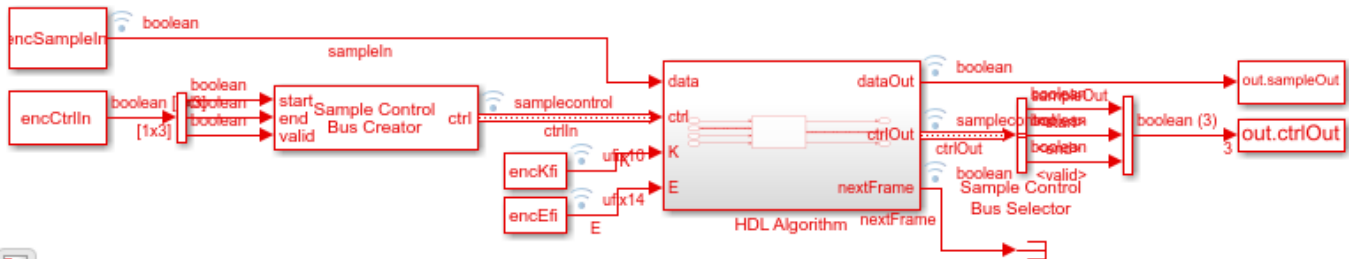
sampleTime = 1;
simTime = length(encCtrlIn) + K(numFrames)*2; %#ok<NASGU>
```

### Run Encoder Model

The HDL Algorithm subsystem contains the NR Polar Encoder block. Running the model imports the input signal variables from the workspace and returns a stream of polar-encoded output samples and control signals that indicate the frame boundaries. The model exports variables `sampleOut` and `ctrlOut` to the MATLAB workspace.



```
open_system('NRPolarencodeHDL');
encOut = sim('NRPolarencodeHDL');
```



### Verify Encoder Results

Convert the streaming data back to frames for comparison with the results of the 5G Toolbox™ `nrPolarEncode` function.

```
encHDL = whdlSamplesToFrames(encOut.sampleOut,encOut.ctrlOut);
```

```
for ii=1:numFrames
    encRef = nrPolarEncode(double(dataIn{ii}),E(ii),10,false); % last two arguments needed for up
    error = sum(abs(encRef - encHDL{ii}));
    fprintf(['Encoded Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'],ii,error);
end
```

Maximum frame size computed to be 256 samples.

```
Encoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

### Generate Input Data for Decoder

Use the encoded data to generate input log-likelihood ratios (LLRs) for the NR Polar Decoder block. Use channel, modulator, and demodulator System objects to add noise to the signal.

Again, create vectors of **K** and **E** values, and convert the frames of data to streaming samples with control signals. The example model imports the workspace variables `decSampleIn`, `decCtrlIn`, `decKfi`, `decEfi`, `sampleTime`, and `simTime`.

For this example, the number of invalid cycles between frames is empirically chosen to accommodate the latency of the NR Polar Decoder block for the specified **K** and **E** values. When the values of **K** and **E** are larger than in this example, the number of invalid cycles between frames must be longer. Use the **nextFrame** output signal of the block to determine when the block is ready to accept the start of the next input frame.

```
nVar = 0.7;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);
% more idle cycles greater list lengths. max 5251 for list 4.
% 1st pkt LL=8 just over 5000, not sure what is max?
```

```

% should i make this a more simulink-y example to show how to use the fifo
% with the nextframe signal?
idleCyclesBetweenFrames = 6000;
btwFrames = false(idleCyclesBetweenFrames,1);
decKfi = [];
decEfi = [];
rxLLR = {numFrames};
rxLLRfi = {numFrames};
for ii=1:numFrames
    mod = bpskMod(double(enchDL{ii}));
    rSig = chan(mod);
    rxLLR{1,ii} = bpskDemod(rSig);
    rxLLRfi{1,ii} = fi(rxLLR{1,ii},1,6,0);
    decKfi = [decKfi; repmat([fi(K(ii),0,10,0);btwSamples],length(rSig),1);btwFrames];
    decEfi = [decEfi; repmat([fi(E(ii),0,14,0);btwSamples],length(rSig),1);btwFrames];
end

[decSampleIn,decCtrlIn] = whdlFramesToSamples(...
    rxLLRfi,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesPerCycle);

simTime = length(decCtrlIn) + K(numFrames)*2;

```

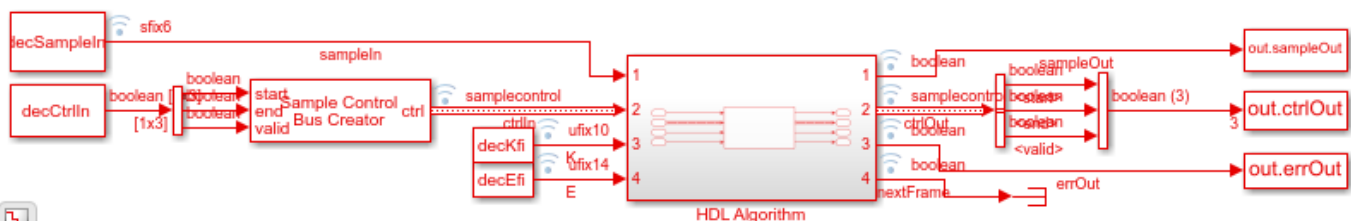
### Run Decoder Model

The HDL Algorithm subsystem contains the NR Polar Decoder block configured to use a list length of eight. Running the model imports the input signal variables from the workspace and returns a stream of decoded output samples and control signals that indicate the frame boundaries. The model exports variables `sampleOut`, `ctrlOut`, and `errOut` to the MATLAB workspace. Select the valid values of the `errOut` signal by using the `ctrlOut.valid` signal.

```

open_system('NRPolArDecodeHDL');
decOut = sim('NRPolArDecodeHDL');

```



### Verify Decoder Results

Convert the streaming samples returned from the Simulink model into frames for comparison with the results of the 5G Toolbox™ `nrPolArDecode` function.

The `nrPolArDecode` function returns the decoded message, including 24 recalculated CRC bits. The NR Polar Decoder block returns the decoded message without the CRC bits, and returns the CRC status separately on the `err` port.

The block and function output bits can differ for frames that report a decoding error. The block can return a decoding error in cases when the function successfully decodes the message. The overall decoding performance of the block is very close to that of the function.

```

decHDL = whdlSamplesToFrames(decOut.sampleOut,decOut.ctrlOut);
errHDL = decOut.errOut(decOut.ctrlOut(:,2));

```

```

L = 8;
for ii = 1:numFrames
    decRef = nrPolarDecode(rxLLR{1,ii},K(ii),E(ii),L,10,false,11); % last three arguments needed
    [decRef,errRef] = nrCRCDecode(decRef,'11'); % CRC poly is '11' for uplink, '24C' for downlink
    error = sum(abs(decRef - decHDL{1,ii}));
    fprintf(['Decoded Frame %d: Behavioral and ' ...
            'HDL simulation differ by %d bits\n'],ii,error);
    msg = dataIn{1,ii}(1:(length(dataIn{ii})-numCRCBits));
    loopErr = sum(abs(msg - decHDL{1,ii}));
    fprintf(['The decoded output message from the HDL simulation',...
            ' differs from the input message by %d bits \n'],loopErr);
    errRef = any(errRef);
    if ~errHDL(ii) && ~errRef
        fprintf('HDL and behavioral simulations successfully decoded the message. \n');
    elseif errHDL(ii) && ~errRef
        fprintf(['Behavioral simulation successfully decoded the message,',...
                ' but HDL sim reported a decode error\n']);
    elseif ~errHDL(ii) && errRef
        fprintf(['HDL simulation successfully decoded the message',...
                ' but behavioral simulation reported a decode error\n']);
    else
        fprintf('HDL and behavioral simulations both reported a decode error. \n');
    end
end
end

```

Maximum frame size computed to be 121 samples.

```

Decoded Frame 1: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 2: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 3: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 4: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.

```

## See Also

NR Polar Decoder | NR Polar Encoder | nrPolarDecode | nrPolarEncode



# Featured Examples

---

## Sample Rate Conversion for an LTE Receiver

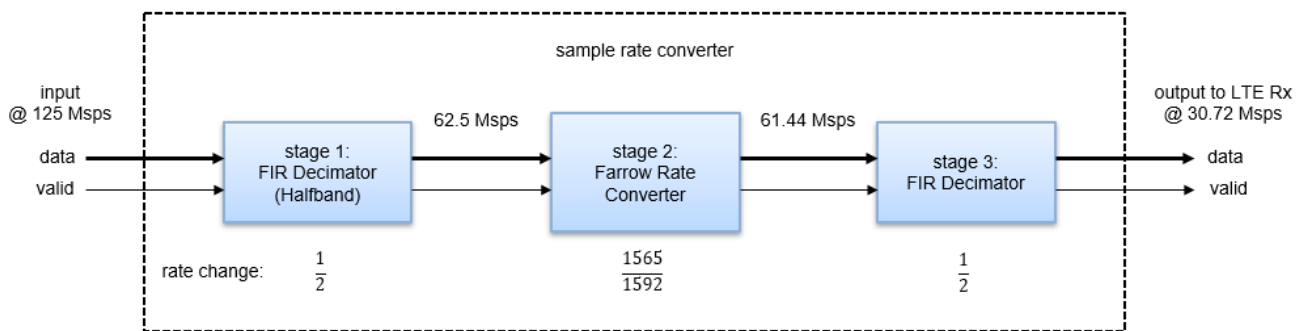
This example shows how to design and implement sample rate conversion for an LTE receiver front end. The model is compatible with the Wireless HDL Toolbox™ receiver reference applications, and supports HDL code generation with HDL Coder™.

### Introduction

The “LTE HDL Cell Search” on page 5-46, “LTE HDL MIB Recovery” on page 5-80, and “LTE HDL SIB1 Recovery” on page 5-63 reference applications require an input sampling rate of 30.72 Msps. In practice, the sampling rate presented to hardware may differ from this, for example due to choice of components or system design decisions. Therefore, sample rate conversion may be required to integrate these reference applications into a larger system. The model shown in this example converts from 125 Msps to 30.72 Msps using two FIR Decimation filters and a Farrow rate converter. The rate change from 125 Msps to 30.72 Msps was deliberately chosen because it is not trivial to implement yet represents an example of the type of rate change often required in a radio receiver.

### Sample Rate Converter Design Overview

The conversion from 125 Msps to 30.72 Msps corresponds to a rate change factor of 0.24576. This is implemented with the filter chain shown. First, the input signal is decimated by two (i.e. a rate change of 1/2) using a halfband filter. Next, a Farrow rate converter is used to make a fine adjustment to the sample rate by a factor of  $1565/1592 = 0.983$ . Last, a decimating FIR filter implements the final decimate-by-two stage.



The reasons behind this choice of filters is as follows:

- 1 The first filter stage can be done efficiently with a halfband filter. The subsequent filter then has two cycles available per input sample to implement resource sharing.
- 2 A Farrow rate converter was chosen to implement the fine adjustment stage due to the high rate change resolution achievable with this approach. This leads to a flexible design which can be readily modified to implement other rate changes.
- 3 Farrow rate converters are expensive in terms of multipliers. This block was placed second in the filter chain, as this option resulted in the least resource utilization while still meeting the specification. If the filter was first in the chain, the width of its transition band could have been relaxed, leading to a shorter filter, however no resource sharing would have been possible. If the

filter was last in the chain, it would have required a narrower transition band leading to a longer filter, however more resource sharing would have been possible.

- 4 It then follows that the last stage is a decimating FIR filter, which can use resource sharing by a factor of two.

In this example, the clock rate is 125 MHz and the input sampling rate is 125 Msps, therefore no resource sharing is implemented in the first filter stage. Stages two and three have a minimum of two cycles per sample available, therefore resource sharing by a factor of two is implemented in parts of the Farrow Rate Converter, and in the final FIR decimation stage. This approximately halves the number of multipliers required to implement these stages compared to a fully parallel implementation.

All of the filter stages have valid input and output signals. These signals are used to represent different sampling rates throughout the filter chain. It's essential for the Farrow rate converter to have a valid output signal because it implements a non-integer rate change. However providing a valid input signal at the first stage means that it is not necessary to pass new data into the sample rate converter on every cycle. This is relevant in scenarios where the hardware clock rate is greater than the input sampling rate.

### Top Level Parameters

Configure the top level parameters of the sample rate converter.  $F_{sADC}$  is the input rate, while  $F_{sLTERx}$  is the output rate; that is, the input to the LTE receiver.  $F_{pass}$  is the passband cut-off frequency and is set to 10 MHz to accommodate the maximum possible LTE bandwidth of 20 MHz.  $F_{stop}$  is set to the Nyquist rate, however can be adjusted if more out-of-band signal rejection is required.  $A_{st}$  is the stopband attenuation in dBs, and  $A_p$  is the desired amount of passband ripple.

```
FsADC    = 125e6;
FsLTERx  = 30.72e6;
Fpass    = 10e6;
Fstop    = FsLTERx/2;
Ast      = 60;
Ap       = 0.1;
```

### Farrow Rate Converter

The Farrow rate converter consists of (i) a fractional delay filter implemented using a Farrow structure and (ii) control logic to determine when to generate output samples, and with which sampling phase. In this example, the Farrow fractional delay filter approximates the impulse response of a custom *prototype* filter using a set of 3rd order polynomials. The prototype filter is designed taking the signal bandwidth and output sampling rate into account, allowing the filter length to be minimized while avoiding aliasing within the signal of interest. The Farrow filter structure is the same as that used in the `dsp.VariableIntegerDelay` (DSP System Toolbox) and `dsp.FarrowRateConverter` (DSP System Toolbox) System objects. Note that the System objects were not used here as they don't support HDL code generation from Simulink.

Define the key parameters of the Farrow rate converter. `numTaps` is the number of taps in each fixed-coefficient FIR of the farrow structure. It is also the number of polynomials used in the approximation. `FsIn` and `FsOut` are the input and output rates respectively. `Fsig` is the bandwidth of the signal of interest. The filter is designed to avoid aliasing within this region. `sps` is the number of samples per section (also known as the oversampling factor) used while designing the prototype filter.

```
farrow.numTaps = 6;
farrow.FsIn    = FsADC/2;
farrow.FsOut   = 2*FsLTERx;
```

```
farrow.Fsig    = Fpass;
farrow.sps    = 16;
```

Design the prototype filter, and approximate it with a set of polynomials. A helper class called `FarrowDesignUtils` contains a set of methods which are used to design and analyze the fractional delay filter. These methods will not be discussed in detail. Refer to the source code for more information.

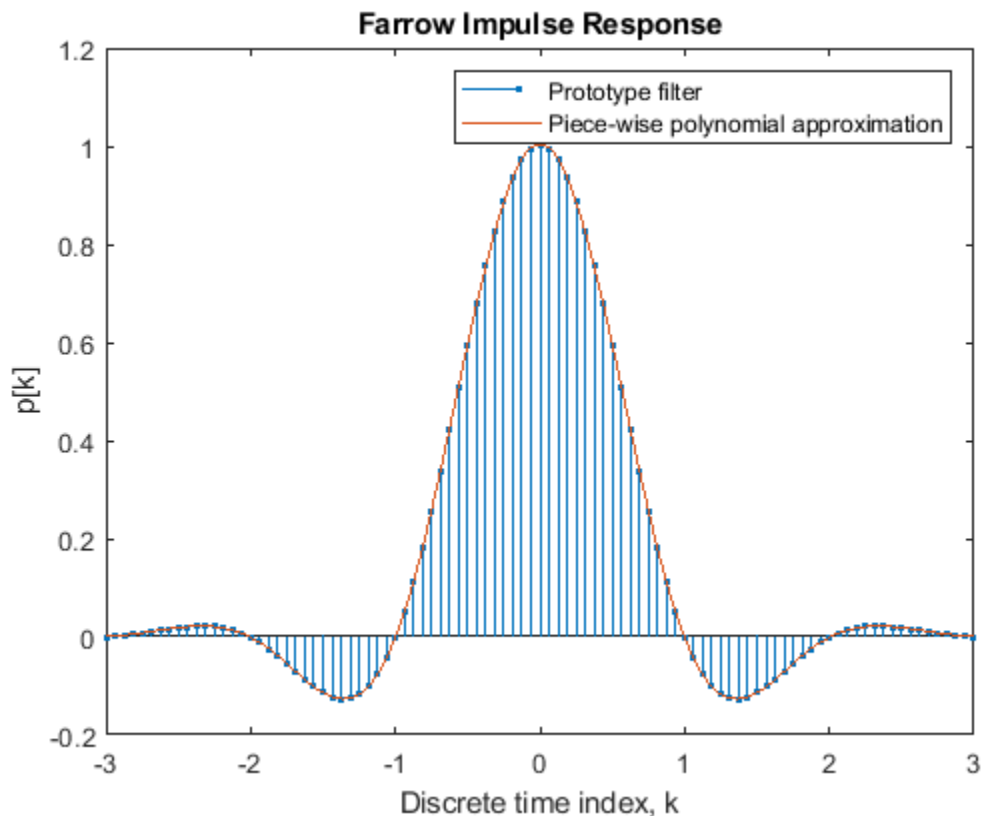
```
farrow.prototype = FarrowDesignUtils.designFilterPrototype(farrow);
farrow.polynomials = FarrowDesignUtils.generatePolynomialCoefficients(farrow);
```

Evaluate the impulse response of the approximation, and compare it to the prototype filter. For visualization purposes, the reconstruction is performed with 100 samples per section in contrast to the prototype filter, which only contains 16 samples per section.

```
[protoInterp,ta] = FarrowDesignUtils.evaluateApproximation(farrow.polynomials,100);

srcPlots.FarrowIR = figure;
tp = ((0:length(farrow.prototype)-1) - floor(length(farrow.prototype)/2))/farrow.sps;
stem(tp,farrow.prototype, '.'); hold on;
plot(ta,protoInterp);

SRCTestUtils.setPlotNameAndTitle('Farrow Impulse Response');
ylabel('p[k]');
xlabel('Discrete time index, k');
legend('Prototype filter', 'Piece-wise polynomial approximation');
```





Compare the approximation to the prototype filter in the frequency domain. The reconstruction is performed with 16 samples per section to match the sampling rate of the prototype filter and facilitate the comparison. The plot also highlights the spectral components which will alias on top of the signal of interest once it has been converted to the output rate. This shows that no significant aliasing will occur.

```

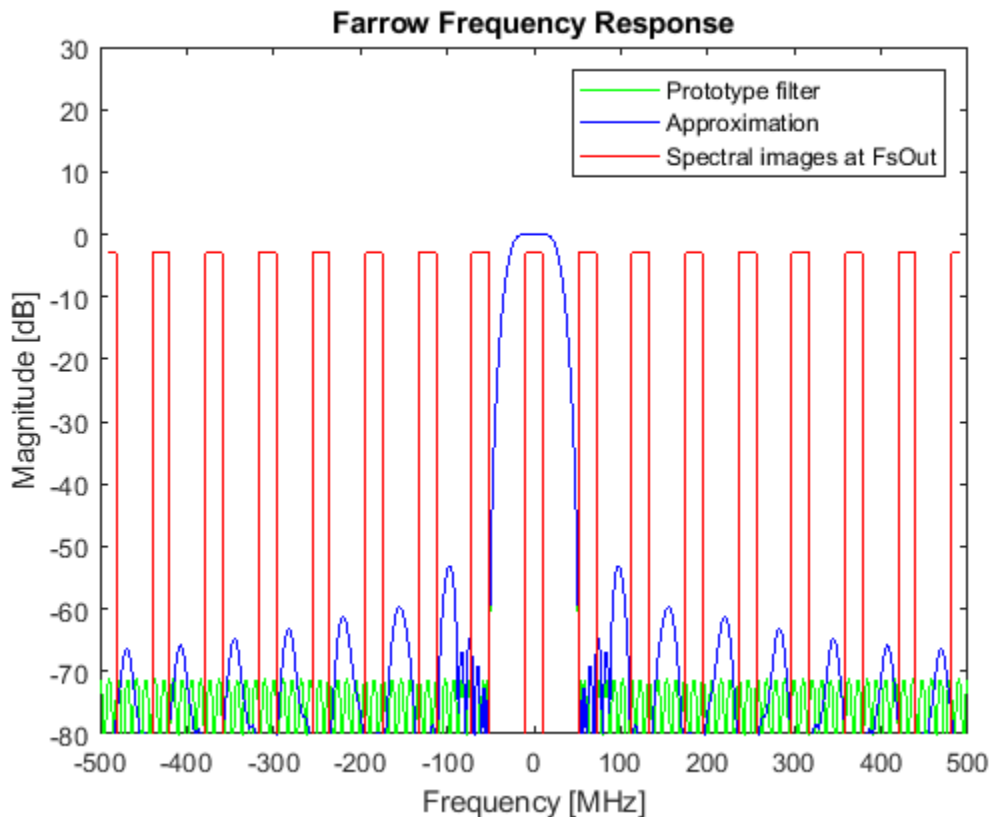
protoApprox = FarrowDesignUtils.evaluateApproximation(farrow.polynomials, farrow.sps);

srcPlots.FarrowFreq = figure; clf;

Fsover = farrow.sps * farrow.FsIn;
Nfft = 2048;
f = Fsover*(-Nfft/2:Nfft/2-1)/Nfft;
plot(f/1e6, 20*log10(abs(fftshift(fft(farrow.prototype/farrow.sps, Nfft))))) , 'g'); hold on;
plot(f/1e6, 20*log10(abs(fftshift(fft(protoApprox/farrow.sps, Nfft))))) , 'b');
ax = axis;
axis([ax(1) ax(2) -80 30]);
FarrowDesignUtils.plotSignalImages(farrow.FsOut);

SRCTestUtils.setPlotNameAndTitle('Farrow Frequency Response');
xlabel('Frequency [MHz]');
ylabel('Magnitude [dB]');
legend('Prototype filter', 'Approximation', 'Spectral images at FsOut');

```



## Decimating FIR Filters

Design the first and last FIR filter stages. Both filters use 16-bit coefficients. For convenience, the coefficients data type is defined.

```
FIRCoeffsDT = numerictype(1,16,15);
```

### Halfband Decimator

Design a halfband filter to efficiently decimate the input by 2.

```
hbParams.FsIn          = FsADC;
hbParams.FsOut         = FsADC/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fpass;
hbParams.StopbandAttenuation = Ast + 10;

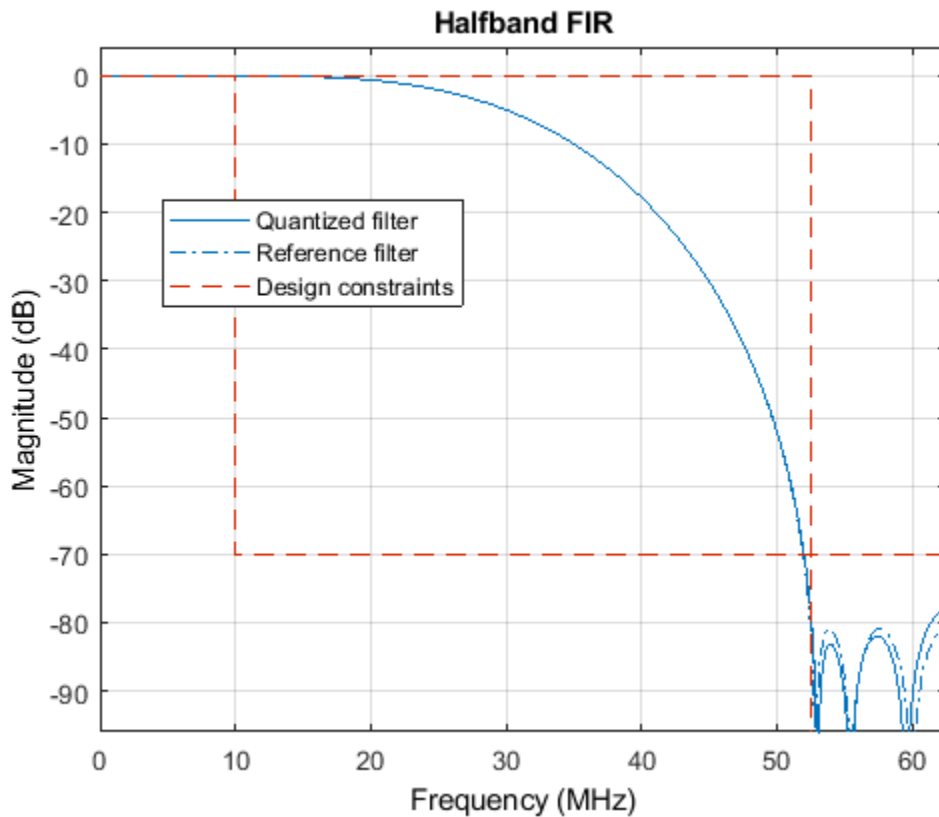
hbSpec = fdesign.decimator(2,'halfband',...
    'Tw,Ast',...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation,...
    hbParams.FsIn);

halfband = design(hbSpec,'SystemObject',true);

halfband.FullPrecisionOverride = false;
halfband.CoefficientsDataType = 'Custom';
halfband.CustomCoefficientsDataType = numerictype([],...
    FIRCoeffsDT.WordLength,FIRCoeffsDT.FractionLength);

Plot the frequency response of the filter, including the quantized response.

srcPlots.halfband = fvtool(halfband,'arithmetic','fixed');
SRCTestUtils.setPlotNameAndTitle('Halfband FIR');
legend('Quantized filter','Reference filter','Design constraints');
```



### Final FIR Decimator

Design the final decimate-by-2 FIR filtering stage.

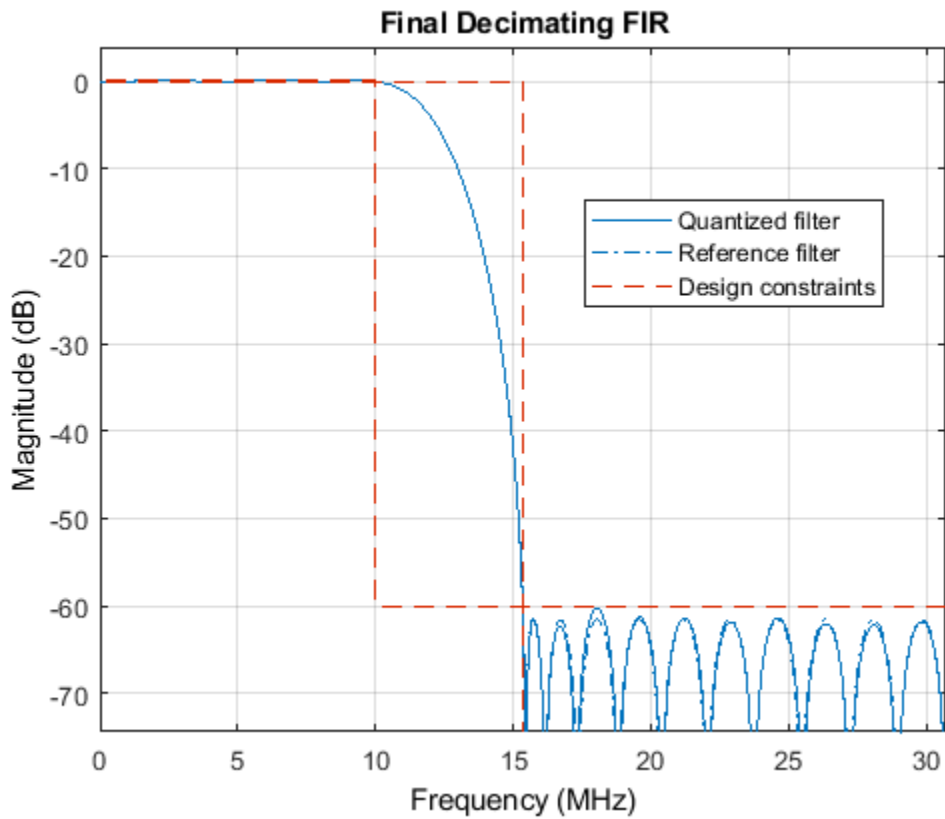
```
finalSpec = fdesign.decimator(2, 'lowpass', ...
    'Fp,Fst,Ap,Ast', Fpass, Fstop, Ap, Ast, farrow.FsOut);

finalFilt = design(finalSpec, 'equiripple', 'SystemObject', true);

finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([], ...
    FIRCoeffsDT.WordLength, FIRCoeffsDT.FractionLength);
```

Plot the frequency response of the filter, including the quantized response.

```
srcPlots.finalFilt = fvtool(finalFilt, 'arithmetic', 'fixed');
SRCTestUtils.setPlotNameAndTitle('Final Decimating FIR');
legend('Quantized filter', 'Reference filter', 'Design constraints');
```



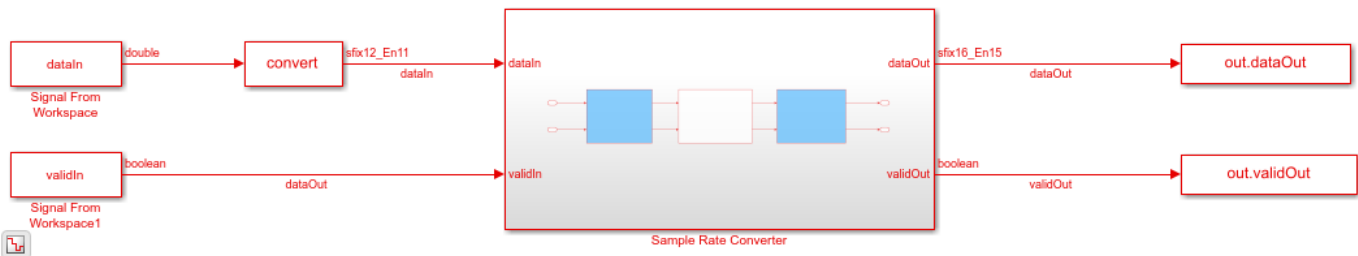
### Simulink HDL Implementation

Open the model and update the diagram. The top level of the model is shown. HDL code can be generated for the **Sample Rate Converter** subsystem.

```

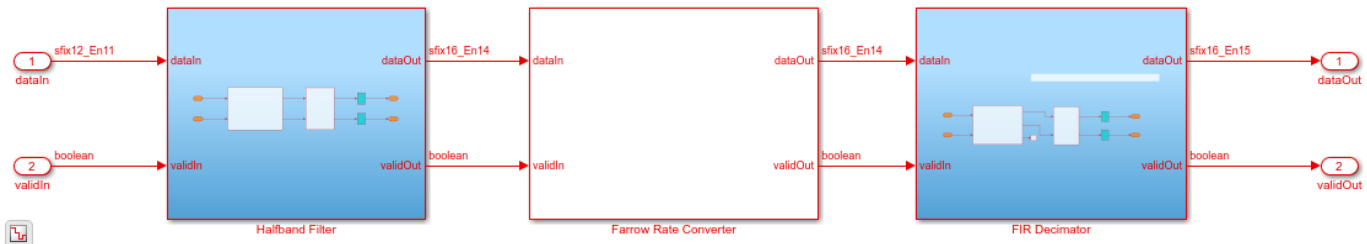
stopTime = 0;
dataIn = 0;
validIn = false;
modelName = 'SampleRateConversionHDL';
open_system(modelName);
set_param(modelName, 'SimulationCommand', 'Update');
set_param(modelName, 'Open', 'on');
    
```

**Sample Rate Conversion for an LTE Receiver**



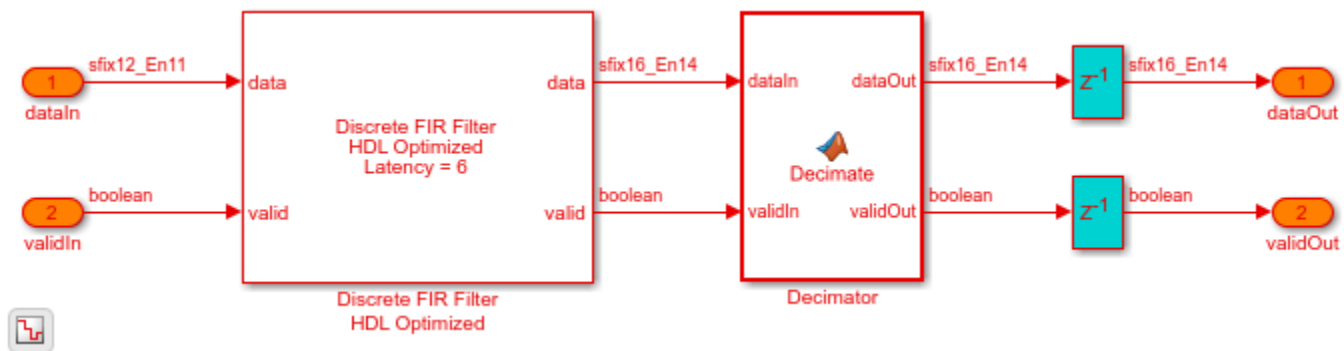
As discussed, the sample rate converter contains a halfband filter, a Farrow rate converter and a final FIR decimation stage.

```
set_param([modelName '/Sample Rate Converter'],'Open','on');
```



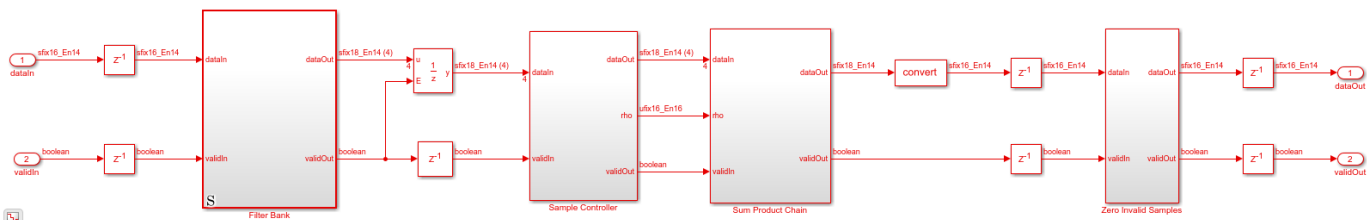
The halfband FIR is implemented using the Discrete FIR Filter HDL Optimized block, and a MATLAB function block to implement decimation by 2. The FIR block uses a transposed filter structure, which optimizes for symmetry and zero coefficients.

```
set_param([modelName '/Sample Rate Converter/Halfband Filter'],'Open','on');
```



The Farrow rate converter comprises a **Filter Bank** of fixed-coefficient FIRs, a **Sample Controller** for generating the output timing, and a **Sum Product Chain** to compute the final output samples. The **Sample Controller** uses a validOut signal to tell the **Sum Product Chain** when to generate a new output sample. It also passes the new sampling phase as a fraction,  $\rho$ , where  $0 \leq \rho < 1$ .

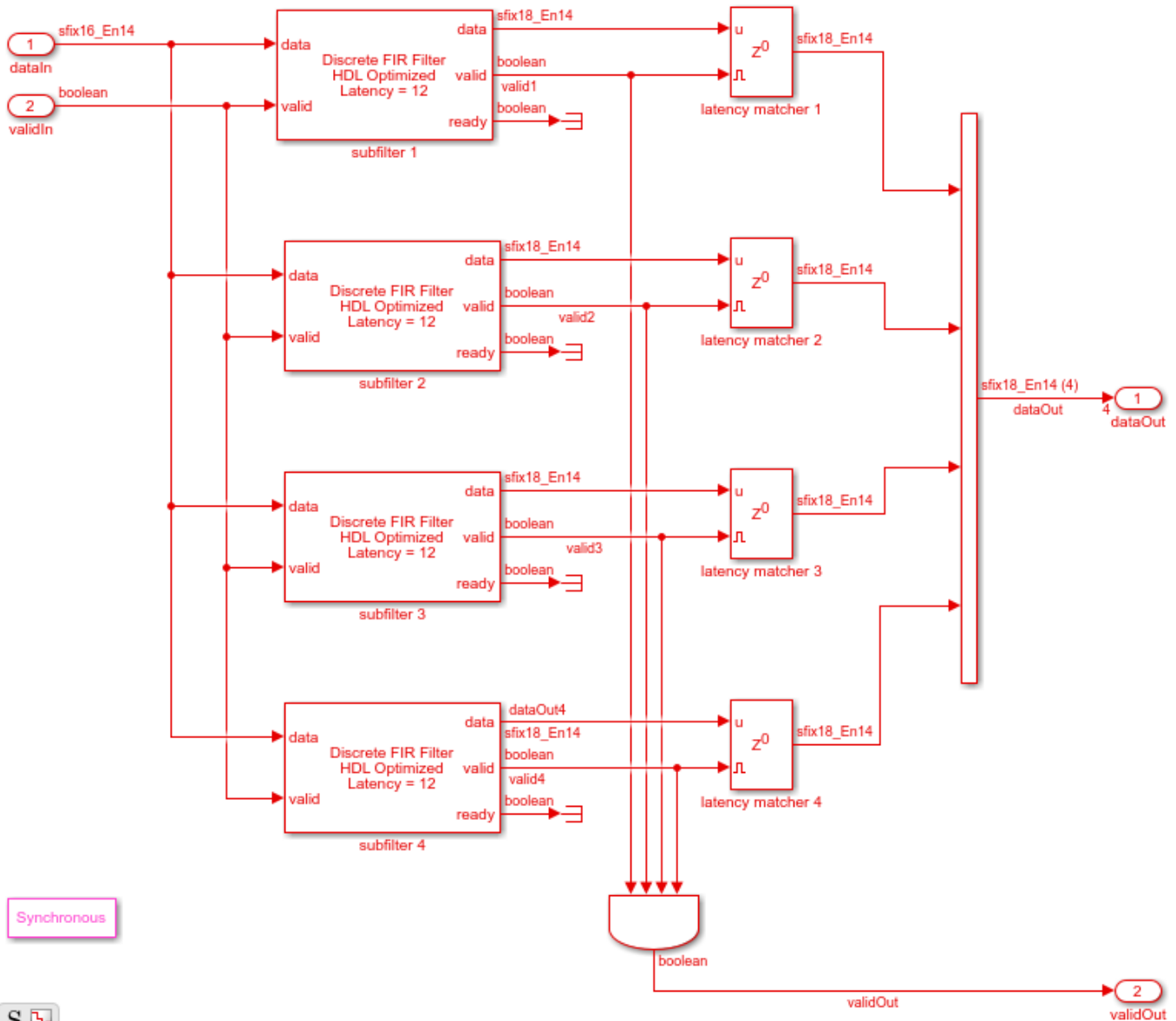
```
set_param([modelName '/Sample Rate Converter/Farrow Rate Converter'],'Open','on');
```



The **Filter Bank** subsystem is implemented with four Discrete FIR Filter HDL Optimized blocks. Each block is configured to share resources according to the **Min cycles between valid input samples** parameter of the **Farrow Rate Converter** subsystem, which is set to 2 in this case. The latency of the FIRs may differ from one another due to symmetry and zero coefficient optimization, therefore each filter also has an associated **latency matcher** (delay) block to compensate for any differences. The additional delay needed to compensate for the latency of each filter is calculated by

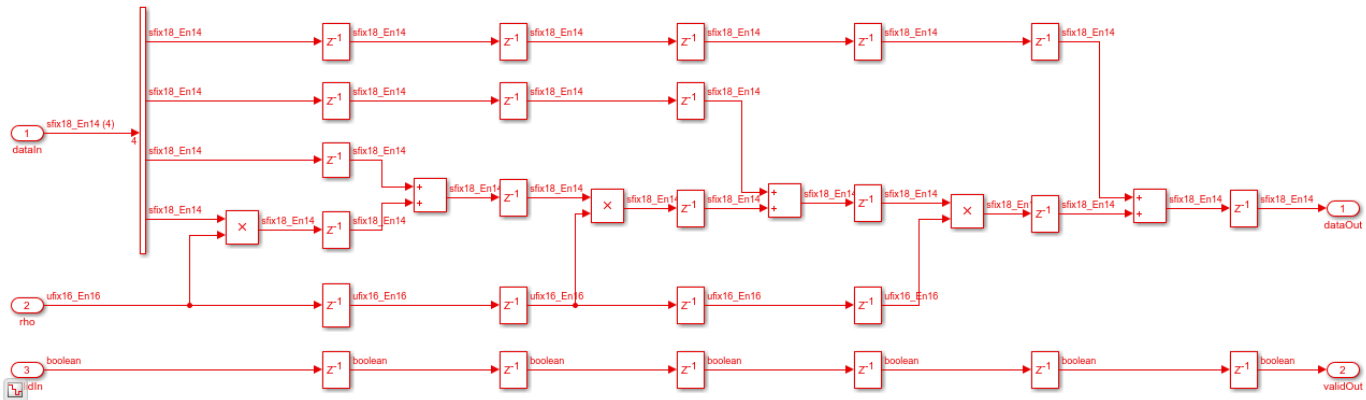
the `getSubFilterMatchingLatencies` function. `getSubFilterMatchingLatencies` is called during model initialization and assigned to a variable called `matchLatencies`. To see this, edit the **Farrow Rate Converter** subsystem mask and go to the Initialization tab. In this example, all of the filters have equal latency, therefore all of the **latency matcher** have a delay of zero. If the Farrow coefficients are changed via the block mask, the FIR latencies may change and the latency matcher blocks will automatically compensate for any differences. Finally all four filter outputs are passed out in a vector.

```
set_param([modelName '/Sample Rate Converter/Farrow Rate Converter/Filter Bank'], 'Open', 'on');
```



The **Sum Product Chain** combines the four FIR outputs with `rho` to generate output samples according to the Farrow structure.

```
set_param([modelName '/Sample Rate Converter/Farrow Rate Converter/Sum Product Chain'], 'Open', 'on');
```



### Validation and Verification

An LTE test signal is generated at 125 Msps and passed through the rate converter. An Error Vector Magnitude (EVM) measurement is then performed, confirming that the resampler is suitable for use in an LTE receiver. For reference, three different methods are used to resample the signal to 30.72 Msps and their EVM results compared. The three methods are:

- 1 The MATLAB resample function.
- 2 A MATLAB model of the rate converter.
- 3 The Simulink HDL model of the rate converter.

In addition, to confirm correct operation of the HDL implementation, the root-mean-square error between the outputs of the MATLAB and Simulink rate converter models is computed.

Generate a 20 MHz LTE test signal sampled at 125 Msps.

```
rng(0);
enb          = lteRMCDL('R.9');
enb.TotSubframes = 2;
[tx, ~, sigInfo] = lteRMCDLTool(enb,randi([0 1],1000,1));
dataIn = resample(tx,FsADC,sigInfo.SamplingRate);
dataIn = 0.95 * dataIn / max(abs(dataIn));
validIn = true(size(dataIn));
```

Use the `resample` function to resample the received signal from the ADC rate to 30.72 Msps. This provides a good quality reference to compare to the rate converter.

```
resampleOut = resample(dataIn,FsLTERx,FsADC);
```

Pass the signal through a MATLAB model of the rate converter.

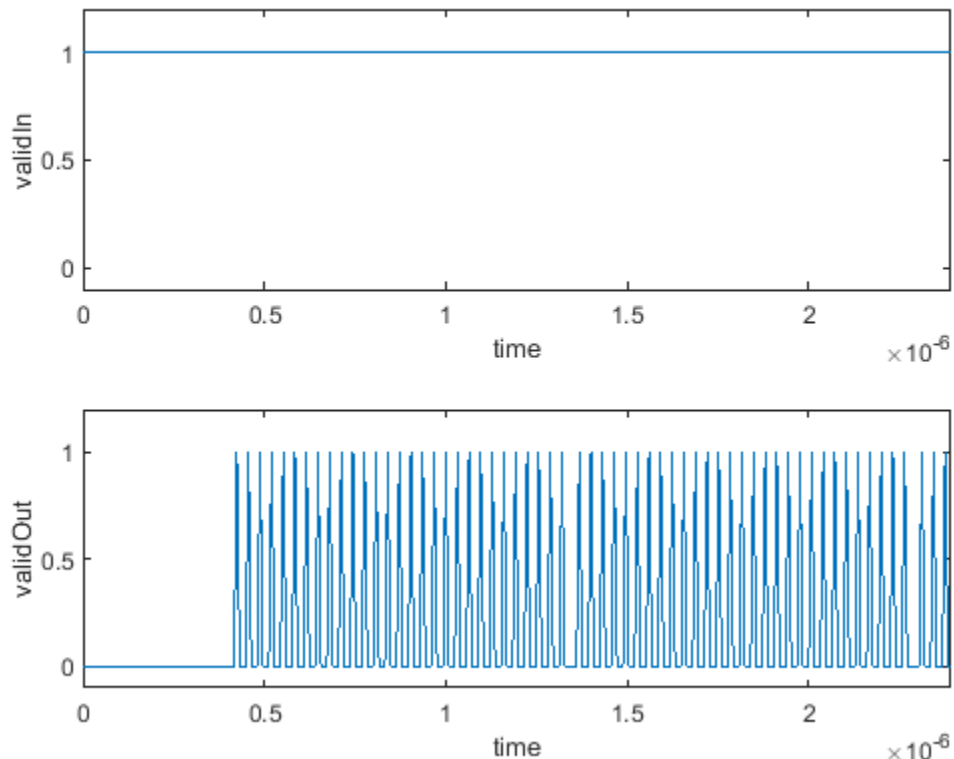
```
halfbandOut      = halfband(dataIn);
farrowOut        = FarrowDesignUtils.convertSampleRate(farrow,halfbandOut);
farrowOut        = farrowOut(1:length(farrowOut)-mod(length(farrowOut),2));
floatResamplerOut = finalFilt(farrowOut);
```

Pass the signal through the fixed-point Simulink HDL implementation model.

```
stopTime        = (length(dataIn)+1000)/FsADC;
simOut          = sim(modelName);
fiResamplerOut  = simOut.dataOut(simOut.validOut);
fiResamplerOut  = fiResamplerOut(1:length(floatResamplerOut));
```

Plot `validIn` and `validOut` to show the overall rate change of the sample rate converter. `validIn` is always HIGH, whereas `validOut` is HIGH about a quarter (0.24576%) of the time.

```
srcPlots.validSignals = figure;
Ns = 300;
validInSlice = validIn(1:Ns);
validOutSlice = simOut.validOut(1:Ns);
subplot(2,1,1);
plot((0:Ns-1)/FsADC,validInSlice);
axis([0 (Ns-1)/FsADC -0.1 1.2]);
ylabel('validIn');
xlabel('time');
subplot(2,1,2);
plot((0:Ns-1)/FsADC,validOutSlice);
axis([0 (Ns-1)/FsADC -0.1 1.2]);
ylabel('validOut');
xlabel('time');
```



Compute the root mean square error between the outputs of the MATLAB and Simulink models of the rate converter

```
e = floatResamplerOut-fiResamplerOut;
rootMeanSquareError = sqrt((e' * e)/length(e));
disp(['Root-mean-square error: ' num2str(rootMeanSquareError)]);
```

Root-mean-square error: 9.4529e-05

Measure the EVM of all three resampling methods.



```

results.resampleEVM           = SRCTestUtils.MeasureEVM(sigInfo, resampleOut, FsLTERx);
results.floatPointSRCEVM     = SRCTestUtils.MeasureEVM(sigInfo, floatResamplerOut, FsLTERx);
[results.fixedPointSRCEVM, fiEqSymbols] = SRCTestUtils.MeasureEVM(sigInfo, fiResamplerOut, FsLTERx);

disp('LTE Error Vector Magnitude (EVM) Measurements');
disp(['    resample function RMS EVM: ' num2str(results.resampleEVM.RMS*100,3) ' %']);
disp(['    resample function Peak EVM: ' num2str(results.resampleEVM.Peak*100,3) ' %']);
disp(['    floating point SRC RMS EVM: ' num2str(results.floatPointSRCEVM.RMS*100,3) ' %']);
disp(['    floating point SRC Peak EVM: ' num2str(results.floatPointSRCEVM.Peak*100,3) ' %']);
disp(['    fixed point HDL SRC RMS EVM: ' num2str(results.fixedPointSRCEVM.RMS*100,3) ' %']);
disp(['    fixed point HDL SRC Peak EVM: ' num2str(results.fixedPointSRCEVM.Peak*100,3) ' %']);

```

```

LTE Error Vector Magnitude (EVM) Measurements
    resample function RMS EVM: 0.0138 %
    resample function Peak EVM: 0.0248 %
    floating point SRC RMS EVM: 0.0439 %
    floating point SRC Peak EVM: 0.158 %
    fixed point HDL SRC RMS EVM: 0.0515 %
    fixed point HDL SRC Peak EVM: 0.176 %

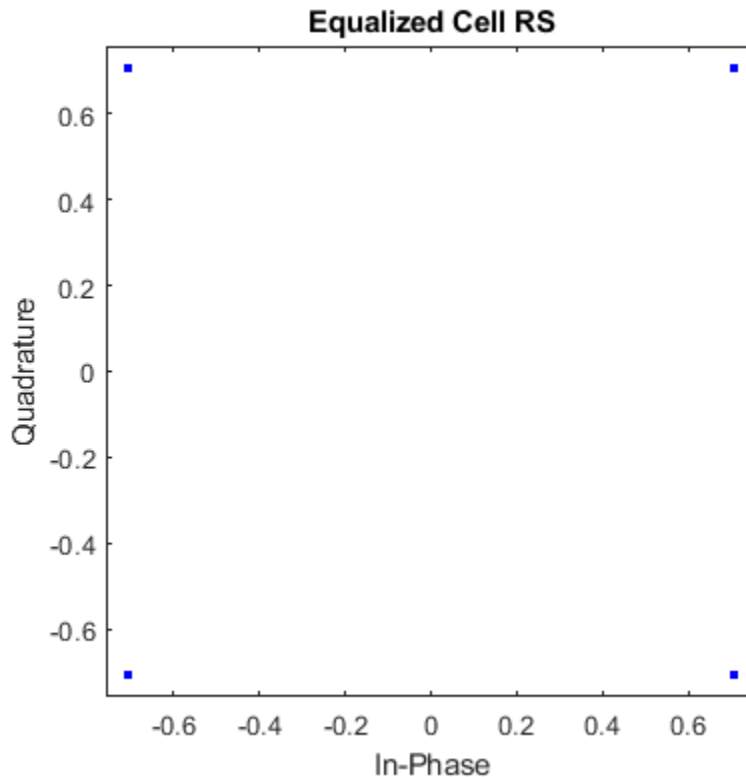
```

Confirm that the signal quality is high by plotting the equalized pilot symbols from the EVM measurement of the HDL implementation. Note that almost no blurring of the constellation points is visible.

```

srcPlots.scatterPlot = scatterplot(fiEqSymbols);
SRCTestUtils.setPlotNameAndTitle('Equalized Cell RS');

```



### HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL testbench for the **Sample Rate Converter** subsystem. The resulting HDL code was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table. The design met timing with a clock frequency of 200 MHz.

```
disp(table(...  
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...  
    categorical({'1553'; '46'; '5629'; '0'; '60'}),...  
    'VariableNames',{'Resource','Usage'}));
```

Resource	Usage
LUT	1553
LUTRAM	46
FF	5629
BRAM	0
DSP	60

## HDL Code Generation for Filtered OFDM (F-OFDM) Transmitter

Filtered OFDM (F-OFDM) applies a filter to the symbols after the IFFT in the transmitter to improve bandwidth while maintaining the orthogonality of the complex symbols. This example implements a transmitter F-OFDM for HDL code generation. The example shows how to go from a MATLAB® reference model to an HDL-optimized Simulink® model. It includes converting from double to fixed-point types, and minimizing the resource use of the design on an FPGA.

Refer to “F-OFDM vs. OFDM Modulation” for comparison between OFDM and F-OFDM waveforms.

### System Parameters

Set the desired F-OFDM properties.

```
NDLRB          = 108;
WaveformType   = 'F-OFDM';
SubcarrierSpacing = 60*1e3; %Hz
CellRefP       = 1;
CyclicPrefix   = 'Normal';
FilterLength    = 513;
ToneOffset     = 2.5000;
CyclicExtension = 'off';
```

Call the `h5gOFDMInfo` function to calculate F-OFDM parameters. The method calculates FFT length, cyclic prefix lengths and number of subcarriers.

```
genb = struct('NDLRB', NDLRB,...
             'WaveformType', WaveformType,...
             'SubcarrierSpacing', SubcarrierSpacing*1e-3,...
             'FilterLength', FilterLength,...
             'ToneOffset', ToneOffset,...
             'CellRefP', CellRefP,...
             'CyclicPrefix', CyclicPrefix,...
             'CyclicExtension', CyclicExtension);
info = h5gOFDMInfo(genb);
```

### Generate a Grid of Input Data

```
QAMModulation = '64QAM';
TotSubframes  = 5;
[txgrid, bitsIn] = generateOFDMGrid(genb,info,QAMModulation,TotSubframes);
```

### Reference MATLAB Model

The reference model runs a floating-point F-OFDM system and plots the spectrum. Use the reference model to compare against the fixed-point model that supports HDL code generation.

```
[txSig_ref,txinfo] = h5gOFDMModulate(genb,txgrid);
```

Model the channel by adding noise to the signal.

```
snrdB = 18;
S = RandStream('mt19937ar','Seed',1);
rxSig_ref = awgn(double(txSig_ref),snrdB,'measured',S);
```

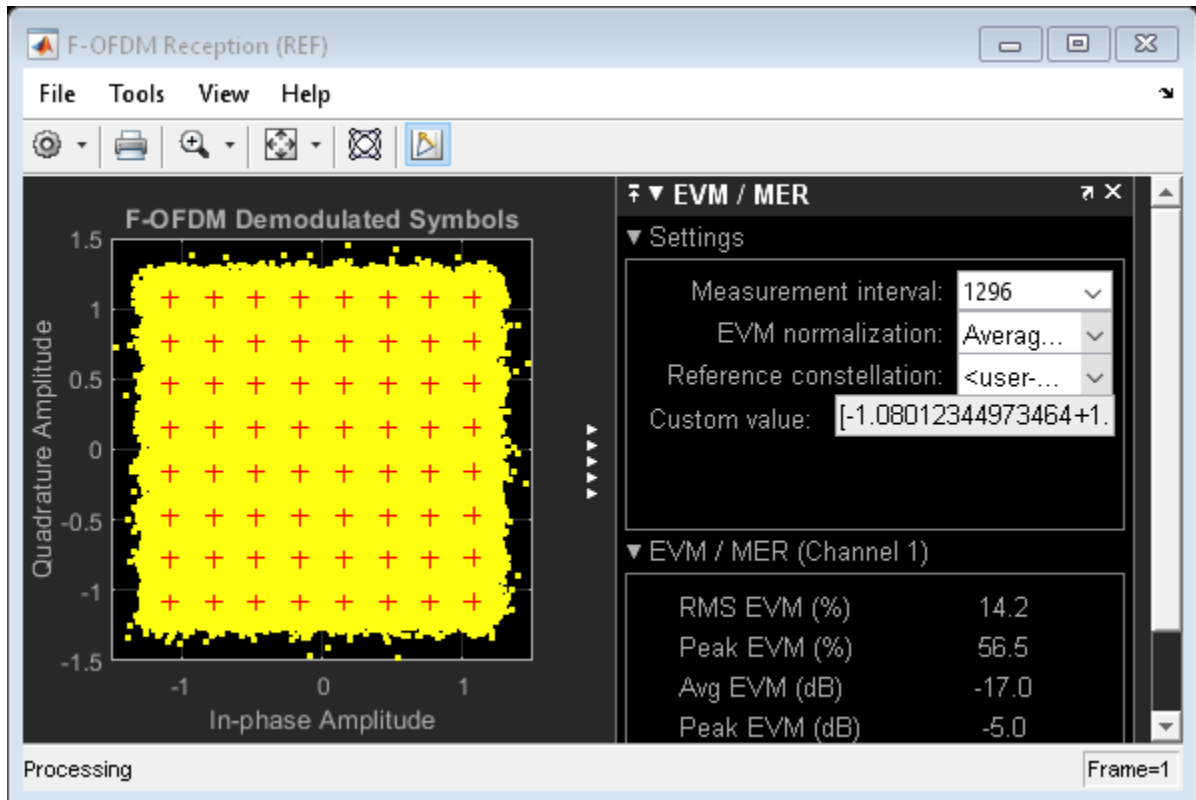
The received signal must be synchronized and aligned. In real situations, the receiver includes symbol synchronization. In this example, the receiver corrects for the shift of the frame by the transmitter filter by  $\frac{\text{FilterLength}}{2}$ .

```
rxSig_ref_sync = circshift(rxSig_ref, -floor(FilterLength/2));
```

Recover data, calculate BER, and display constellation.

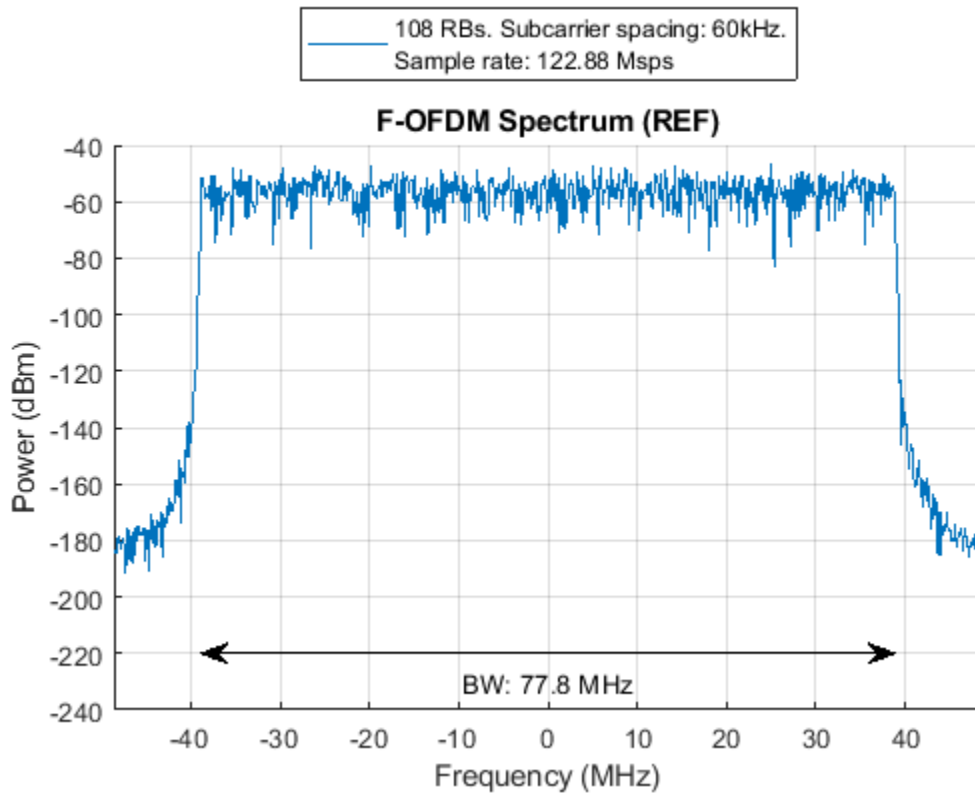
```
[constDiagRx, ber, rxgrid_ref] = FOFDM_Receiver(rxSig_ref_sync, bitsIn, genb,...
        QAMModulation, 'F-OFDM Reception (REF)');
disp(['F-OFDM Reception (REF)', ' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snrdB) ' dB']);
constDiagRx(rxgrid_ref(:));
```

F-OFDM Reception (REF) BER = 0.0094568 at SNR = 18 dB



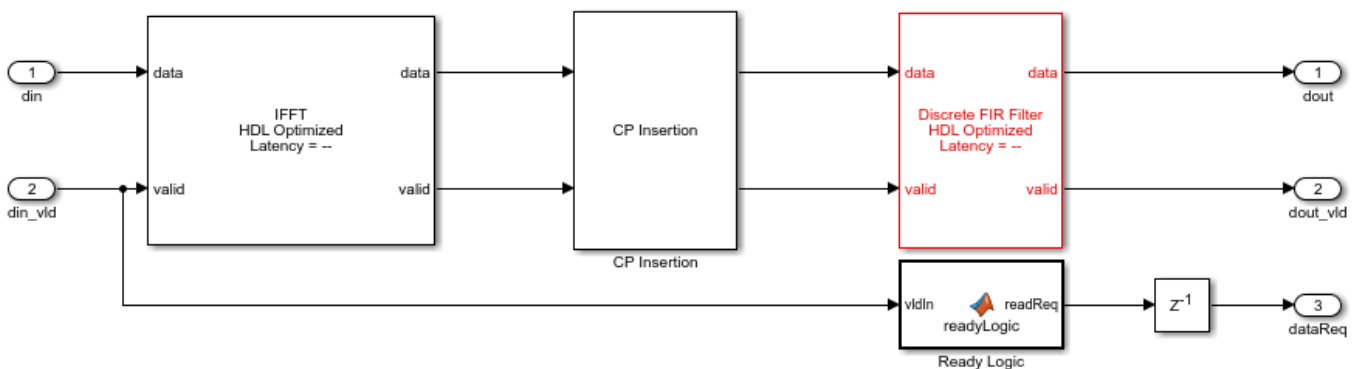
The spectrum shows clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLspectrum(txSig_ref, txinfo, genb, 'F-OFDM Spectrum (REF)');
```



### Simulink Fixed-point Model

```
model = 'F0FDMTransmitterHDLExample_FixPt';
load_system(model);
open_system([model, '/F-OFDM']);
```



To generate HDL from the model, fixed-point data type must be used instead of double. For 64-point QAM, at least 6 bits + 1 sign bit is needed. However, to achieve reasonable BER, the input word length must be increased, considering the FPGA's limitation. Multipliers in FPGAs have limited input word length. For example, Xilinx's DSP48 has 18\*25-bit multiplier. For an optimal design, a wordlength is chosen so that all multipliers in the FFT and the filter are smaller than 18\*25-bit

multipliers. In this example, the FFT HDL Optimized block uses the "Divide butterfly outputs by two" option. The input word length is 16 bits.

You can run the Simulink model with floating point data by setting WORDLENGTH=-1. However, this mode is not supported for HDL code generation.

```
WORDLENGTH = 16;
```

Set the number of fractional bits to WORDLENGTH - 2 bits to cover  $-1 \leq \text{Symbol} \leq 1$ .

```
FRACTIONLENGTH = WORDLENGTH - 2;
```

### Generating OFDM Symbols

The input data to the IFFT is assumed to be a proper OFDM symbol and resides in a memory (OFDM Symbol subsystem in the model) that can be read by F-OFDM Subsystem. Therefore, the transmitter's sample rate depends on the data availability in the memory and FPGA clock frequency. If the data is available all the time, then the sample rate is limited to

$$\text{clock\_frequency} * \frac{\text{FFTLength}}{(\text{FFTLength} + \text{Max}(\text{CyclicPrefixLength}))}.$$

On the other hand, the required sample rate is calculated by  $\text{SubcarrierSpacing} * \text{FFTLength}$  and it is equal to 122.88 Msps for this example. To achieve 122.88 Msps the clock frequency should be at least 135.36 MHz.

```
ifftin = generateOFDMSymbol(txgrid,info,genb);
```

### Filter Design

The appropriate filter should have a flat passband over the subcarriers and sharp transition to minimize guard bands. It also needs sufficient stopband attenuation. A prototype filter  $w = w_1 * w_2$  is used, where  $w_1$  is a SINC function and

$$w_2 = 0.5 * (1 + \cos(\frac{2 * \pi * n}{N-1})).$$

```
fnum = generateFilterCoef(genb,info);
```

### Simulation

Set up the model and run. Note that due to the system latency, the model needs to be simulated longer to collect enough data.

```
Nfft = info.Nfft;
CyclicPrefixLengths = info.CyclicPrefixLengths;
SymbolsPerSubframe = info.SymbolsPerSubframe;

STOPTIME = 4 * TotSubframes * info.SamplesPerSubframe;

sim(model);
txSig_fixpt = TX_WAVEFORM(1: size(txSig_ref));
```

Model the channel by adding some noise to the signal. Note that the same noise is used as in the reference MATLAB model.

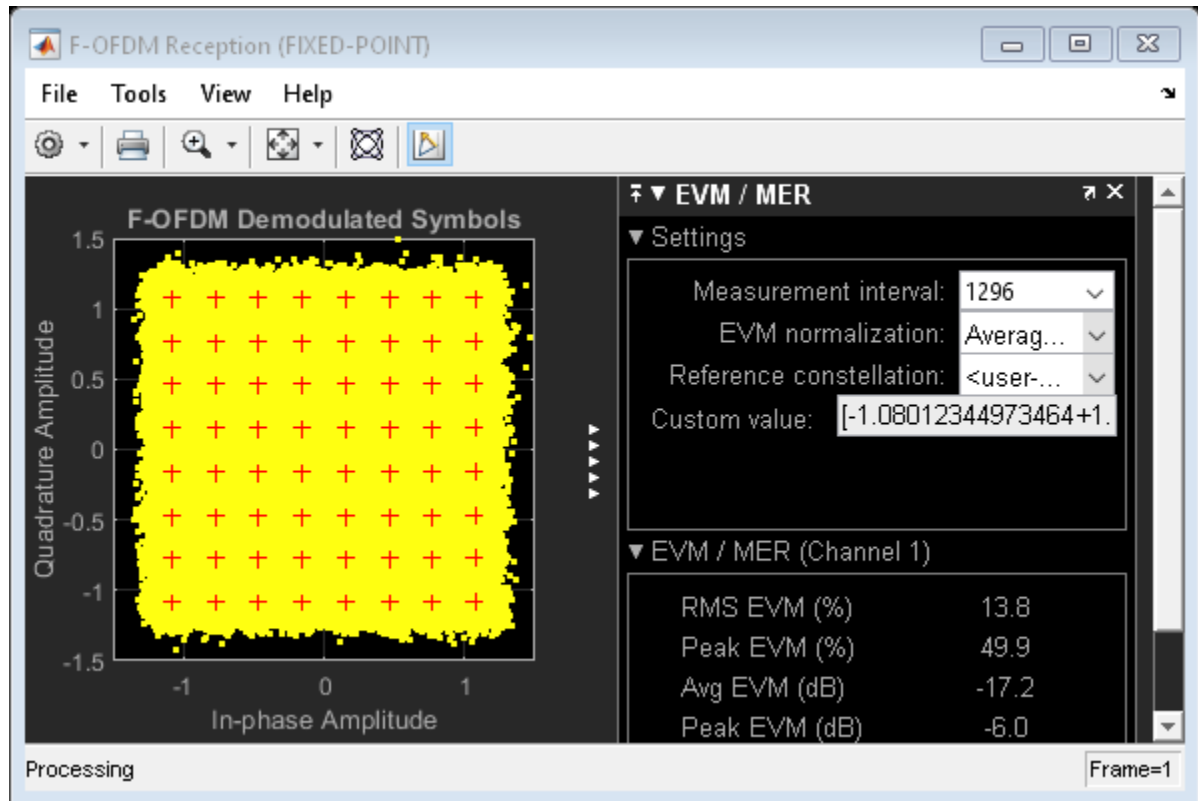
```
S = RandStream('mt19937ar','Seed',1);
rxSig_fixpt = awgn(double(txSig_fixpt),snrdB,'measured',S);
```

Perform symbol synchronization, recover data, calculate BER, and display constellation.

```
rxSig_fixpt_sync = circshift(rxSig_fixpt, -floor(genb.FilterLength/2));
```

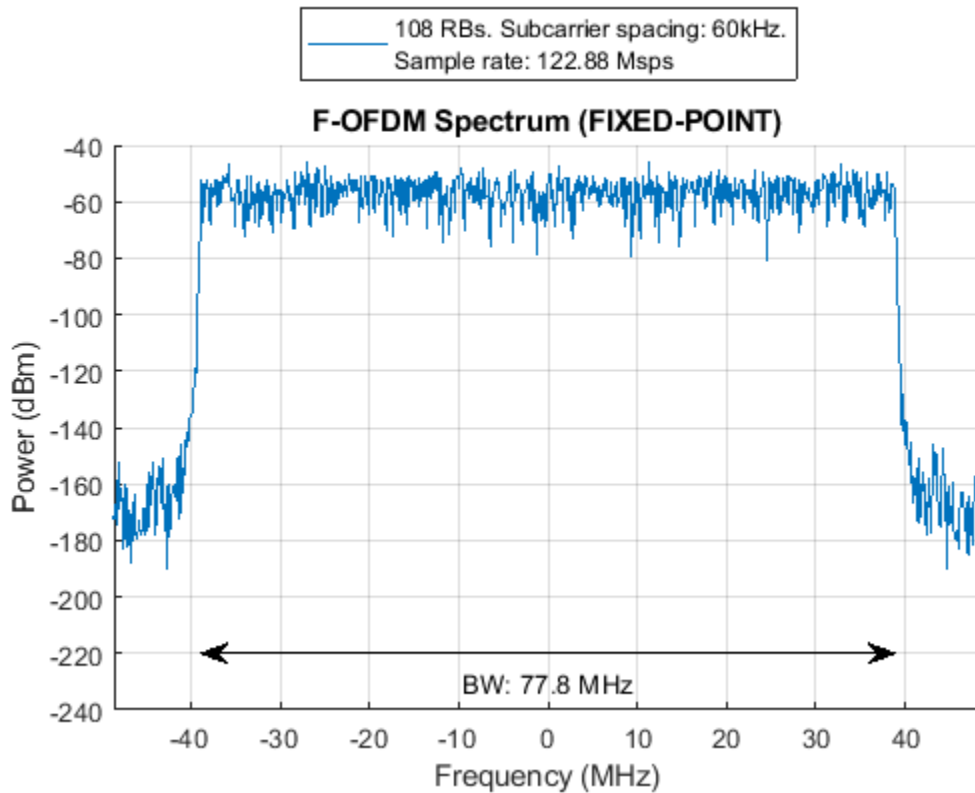
```
[constDiagRx,ber,rxgrid_fixpt] = FOFDM_Receiver(rxSig_fixpt_sync,bitsIn, ...
                                               genb, QAMModulation,'F-OFDM Reception (FIXED-POINT)');
disp(['F-OFDM Reception (FIXED-POINT)', ' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snrdB) ' dB
constDiagRx(rxgrid_fixpt(:));
```

F-OFDM Reception (FIXED-POINT) BER = 0.0094453 at SNR = 18 dB



The spectrum shows even for fixed point a clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLspectrum(txSig_fixpt,txinfo,genb,'F-OFDM Spectrum (FIXED-POINT)');
```



### Simulink HDL Optimized Model

The fixed point model uses a 513-tap filter in the time domain. This filter requires  $2 \times 513$  multipliers since the output of IFFT is complex. Even when implemented using a symmetric filter it needs 513 multipliers which is too many multipliers for a normal size FPGA. To reduce the number of multipliers in the filter, the HDL Optimized model filters in the frequency domain. A frequency domain FIR filter requires FFT of the input multiplied by FFT of the coefficients and then IFFT the result. The number of complex multipliers in this case is

$$2 * \text{ceil}\left(\frac{\log_2(\text{FFTLength})}{2}\right) - 1.$$

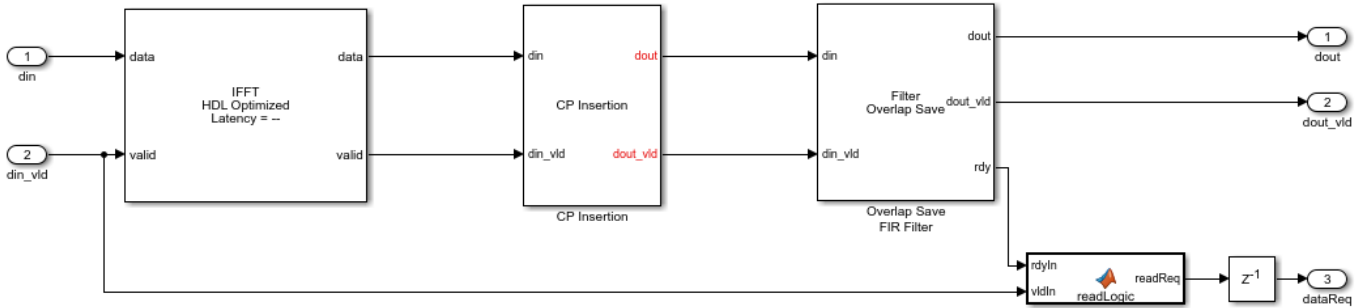
The frequency domain filter in this example uses 11 complex multipliers. Note that the actual number of real multipliers depends on FFT and IFFT block setting (Complex multiplication option) and word length. In the HDL Optimized model, the time domain FIR filter is replaced by a frequency domain FIR filter implemented with an overlap-save architecture. Due to overlapping characteristic of the overlap-save architecture, the sample-rate is limited to

$$\text{clock\_frequency} * \frac{\text{FFTLength}}{(\text{FFTLength} + \text{Max}(\text{CyclicPrefixLength}) + 2 * (\text{FilterLength} - 1))}.$$

Therefore, to achieve 122.88 Mps sample-rate for this example, the clock frequency must be at least 196.8 MHz.

```
model = 'FOFDMTransmitterHDLExample_HDLOpt';
load_system(model);
open_system([model, '/F-OFDM']);
```





Set the length of the FFT for the filter. The length must be at least  $2 * \text{FilterLength}$  for frequency domain filtering. However, because it must process the whole OFDM symbol at once use  $N_{\text{fft}}$  for FFT length inside the filter. Then, calculate the FFT of the coefficients. Bit-reverse the result since the output of the FFT for the filter is bit-reversed.

```
filterFFTLen = Nfft;
fftFnum = bitrevorder(fft(fnum,filterFFTLen).');
```

For fixed-point input data, the output of the FFT inside the filter has a bit-growth =  $\log_2(N_{\text{fft}}) = 11$  bits. To map most of the multipliers into DSP block in FPGA, limit the input word length. For example if DSP has a  $25 * 18$ -bit multiplier, the WORDLENGTH must be 14 bits to achieve 25-bits output of the FFT inside the filter. Also, use 18-bit coefficients.

```
WORDLENGTH = 14;
FRACTIONLENGTH = WORDLENGTH - 2;
if WORDLENGTH > 0 %for fixed point data
    COEF_WL = 18;
    COEF_FR = COEF_WL - 2;
    fftFnum = fi(fftFnum, 1, COEF_WL, COEF_FR, 'RoundingMethod', 'Nearest', ...
        'OverflowAction', 'Wrap');
```

```
end
STOPTIME = 4 * TotSubframes * info.SamplesPerSubframe;
```

```
sim(model);
txSig_HDL0pt = TX_WAVEFORM_HDL0pt(1: size(txSig_ref));
```

Model the channel by adding some noise to the signal. Note that the same noise is used as in the reference MATLAB model.

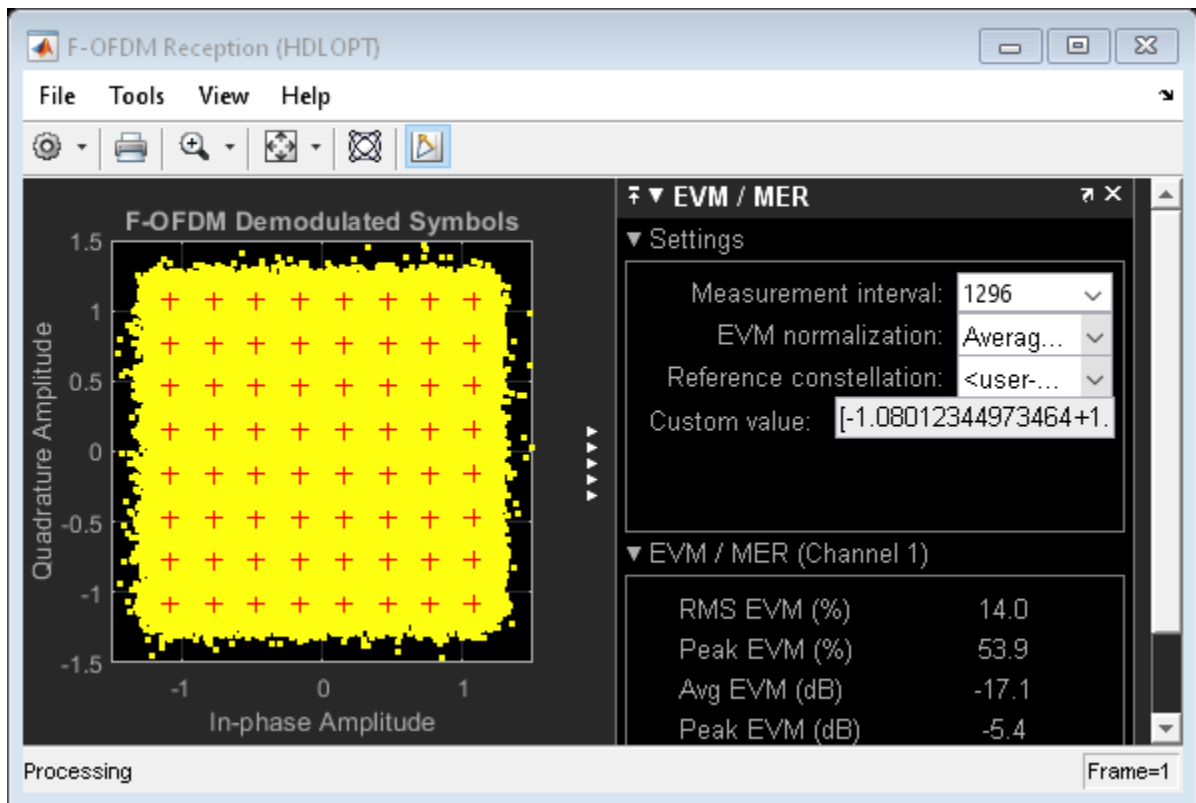
```
S = RandStream('mt19937ar', 'Seed', 1);
rxSig_HDL0pt = awgn(double(txSig_HDL0pt), snrdB, 'measured', S);
```

Perform symbol synchronization, recover data, calculate BER, and display constellation.

```
rxSig_HDL0pt_sync = circshift(rxSig_HDL0pt, -floor(genb.FilterLength/2));
```

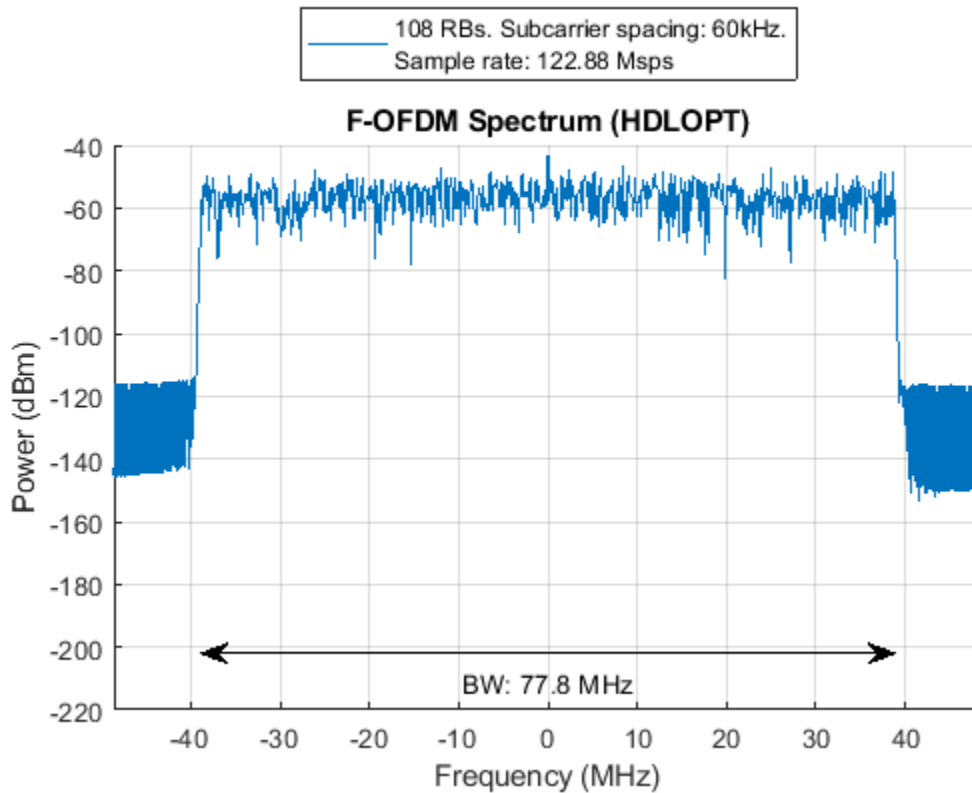
```
[constDiagRx, ber, rxgrid_HDL0pt] = FOFDM_Receiver(rxSig_HDL0pt_sync, bitsIn, ...
    genb, QAMModulation, 'F-OFDM Reception (HDL0PT)');
disp(['F-OFDM Reception (HDL0PT)', ' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snr dB) ' dB']);
constDiagRx(rxgrid_HDL0pt(:));
```

```
F-OFDM Reception (HDL0PT) BER = 0.010038 at SNR = 18 dB
```



The spectrum shows even for fixed point a clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLspectrum(txSig_HDL0pt,txinfo,genb,'F-OFDM Spectrum (HDLOPT)');
```



### Generate HDL Code and Test Bench

Use a temporary directory for the generated files:

```
systemname = 'F0FDMTransmitterHDLExample_HDL0pt/F-OFDM';
workingdir = tempname;
```

You can run the following command to check the F-OFDM subsystem for HDL code generation compatibility:

```
checkhdl(systemname, 'TargetDirectory', workingdir);
```

Run the following command to generate HDL code:

```
makehdl(systemname, 'TargetDirectory', workingdir);
```

Run the following command to generate the test bench:

```
makehdltb(systemname, 'TargetDirectory', workingdir);
```

### Synthesis Result

The design was synthesized for Xilinx Zynq-7000 (xc7z045-ffg900, speed grade 2) using Vivado. This FPGA has 900 DSP48 slices and therefore, the fixed-point version of the design doesn't fit in this device. The HDL Optimized version of the design fits in this chip and achieves a clock frequency of 205.8 MHz which meets the required clock frequency of 196.8 MHz. The design uses 94 DSP48 (10%) and 24 block RAMs (4%).

### **Conclusion**

In this example a Simulink fixed-point model was developed and optimized for hardware. The model minimized resource usage by optimizing use of DSP on the FPGA. Comparing the results of the floating-point model with the fixed-point model shows that 16-bit data has a similar bit error rate to the floating-point data.

### **See Also**

### **Related Examples**

- “F-OFDM vs. OFDM Modulation”

## HDL Implementation of a Variable-Size FFT

This example shows how to implement a variable-size FFT using a single FFT core.

This example includes two models *VariableSizeFFTHDLExample* and *VariableSizeFFTArbitraryValidPatternHDLExample* that show variable-size FFT implementations for different input valid patterns.

Many popular standards like WLAN, WiMax, digital video broadcast (DVB), digital audio broadcast (DAB), and long term evolution (LTE) provide multiple bandwidth options. The required FFT length for OFDM modulation and demodulation for these standards varies with bandwidth option. For example, LTE supports different channel bandwidth options from 1.4 MHz to 20 MHz, which require FFT lengths of 128 to 2048 respectively. The FFT HDL Optimized (DSP System Toolbox) block generates HDL code for a specific FFT length. This example demonstrates how to use the FFT HDL Optimized block to implement a variable-size FFT.

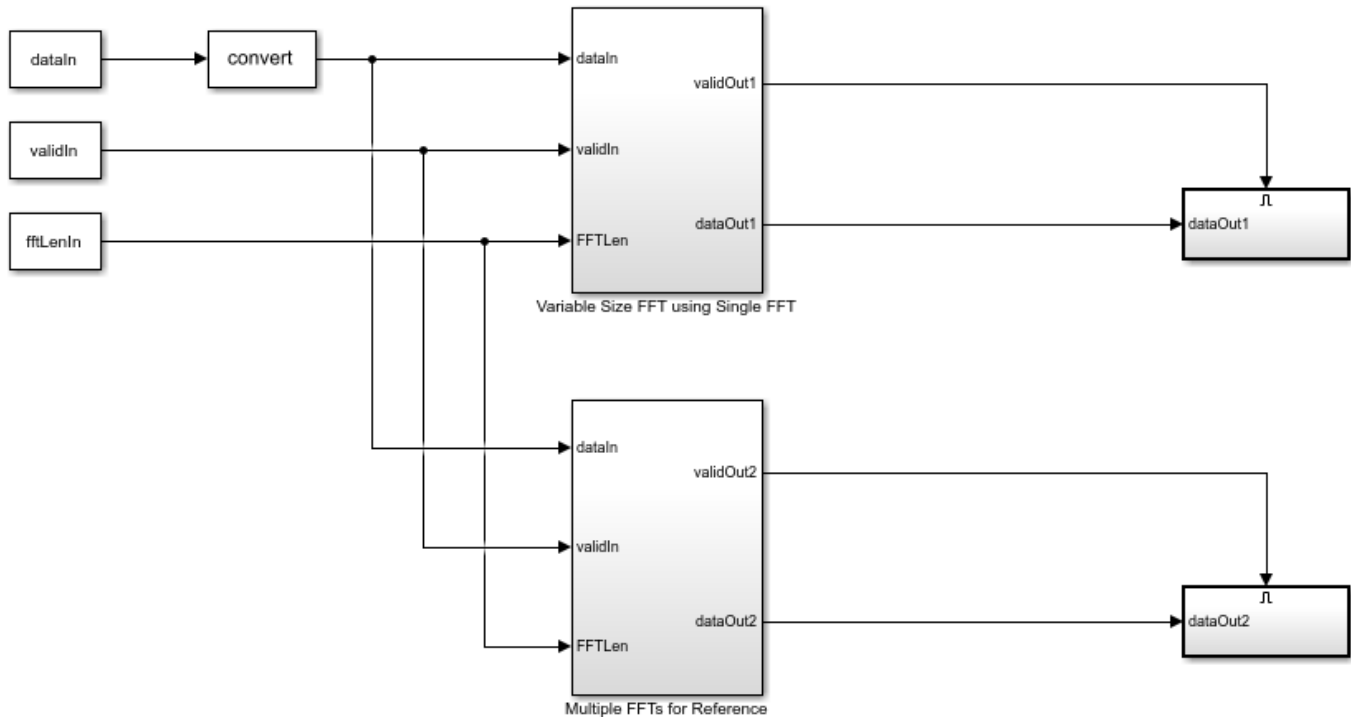
This example generates input data in MATLAB® and imports it to Simulink® for the simulation. The imported data is fed to the implementations of variable-size FFT using a single FFT and multiple FFTs. To demonstrate that the single-FFT implementation matches the results of using multiple FFTs of various sizes, both the output streams from the Simulink simulation are exported to MATLAB and compared.

### Model Architecture

The top-level subsystem in both the models implement a variable-sized FFT. The top subsystem uses a single FFT block and the bottom subsystem provides reference data by using multiple FFT blocks of various sizes.

The model *VariableSizeFFTHDLExample* can process data with a gap between valid samples, provided the gap depends on FFT length.

```
modelName = 'VariableSizeFFTHDLExample';  
open_system(modelName);
```



### Configuration of FFT Lengths

The FFT lengths are specified through a variable `fftLenVecMulFFTs`. The largest of these lengths is stored in a variable `fftLenSinFFT` and used as the FFT length for the FFT block in the 'Variable Size FFT using Single FFT' subsystem.

The input `fftLenIn` is generated by using the vector of FFT lengths specified in `fftLenVecMulFFTs`.

```
fftLenVecMulFFTs = [128;256;512;1024;2048];
% Single FFT length used by variable size FFT.
fftLenSinFFT = max(fftLenVecMulFFTs);
% Generate |fftLenIn| by repeating each element of |fftLenVecMulFFTs| by
% |fftLenSinFFT| times and arranging in a single column.
fflen = repmat(fftLenVecMulFFTs.',fftLenSinFFT,1);
fftLenIn = uint16(fflen(:));
```

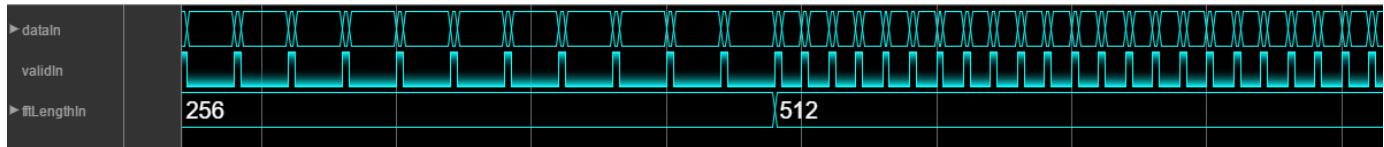
### Input Generation

`dataIn`, `validIn`, and `fftLenIn` inputs are generated in MATLAB and imported to the Simulink model. Random complex input data `randInputData` is generated for each of the FFT lengths specified in `fftLenVecMulFFTs`. Different FFT lengths correspond to different bandwidths and different sampling rates. For instance, in LTE, the FFT lengths of 128, 256, 512, 1024, and 2048 correspond to the sampling rates 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, and 30.72 MHz respectively. The symbol time for any FFT length is  $66.67\mu\text{s}$ . The example operates at the highest rate among the FFT lengths specified.

The `dataIn` signal is generated by padding zeros in between the `randInputData` samples. The figure below shows the input data and valid patterns for `fftLenVecMulFFTs` of 256 and 512 and

fftLenSinFFT being 2048. For the FFT length of 256, the example inserts 7 invalid samples for every valid sample and for the FFT length of 512, the code inserts 3 invalid samples for every valid sample.

The model *VariableSizeFFTHDLExample* requires the input valid pattern to have a gap between valid samples as shown in the figure below.



```

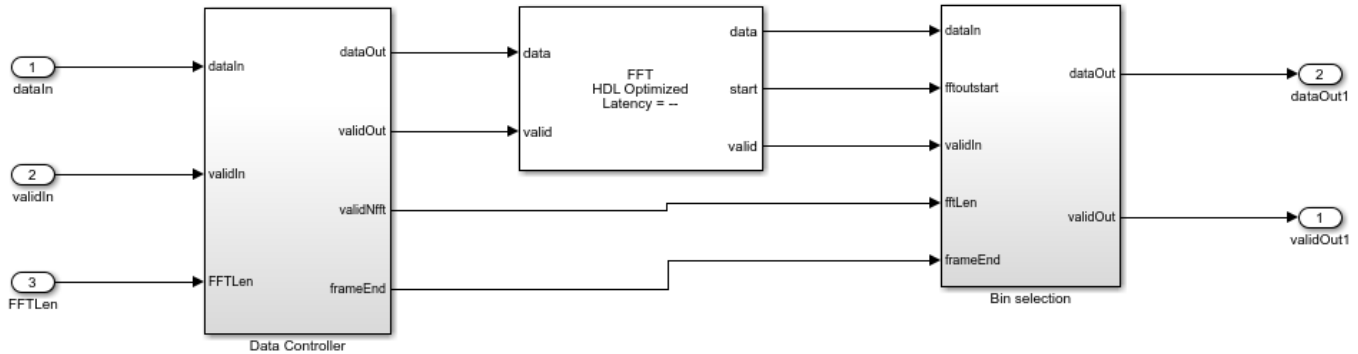
rng('default');
dataIn = zeros(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
validIn = false(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
% Loop over the FFT lengths
for ind = 1:length(fftLenVecMulFFTs)
    % Generate data of FFT length samples
    randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)));
    % Zero padding in between input data samples
    upSamplingFac = fftLenSinFFT/fftLenVecMulFFTs(ind);
    dataIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(randInputData,upSamplingFac);
    % Valid corresponding to the generated data
    tempValid = true(1,fftLenVecMulFFTs(ind));
    validIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(tempValid,upSamplingFac);
end
inputDataType = 'fixdt(1,16,14)'; % Input data type can be modified here
set_param('VariableSizeFFTHDLExample/Data Type Conversion','OutDataTypeStr', inputDataType);
% Get FFT latency
fftObj = dsp.HDLFFT('FFTLength',fftLenSinFFT,...
    'Architecture','Streaming Radix 2^2',...
    'ComplexMultiplication','Use 3 multipliers and 5 adders',...
    'BitReversedOutput',false,...
    'BitReversedInput',false,...
    'Normalize',false);
latency=getLatency(fftObj); % Default latency is 4137 for 2048 point FFT.
additionPipelineDelay = 6; % Number of additional pipeline delays
% Simulink simulation end time Total Latency = Latency of FFT + Latency of
% data controller (5 clock cycles).
% Total simulation running time = Total
% number of input samples + Total Latency + Pipeline delay.
simTime = fftLenSinFFT*(length(fftLenVecMulFFTs) + 1) + latency + additionPipelineDelay ;

```

### Variable-Size FFT using Single FFT

The 'Variable-Size FFT using Single FFT' design includes a Data Controller, an FFT HDL Optimized block, and a Bin selection subsystem.

```
open_system([modelName '/Variable Size FFT using Single FFT']);
```



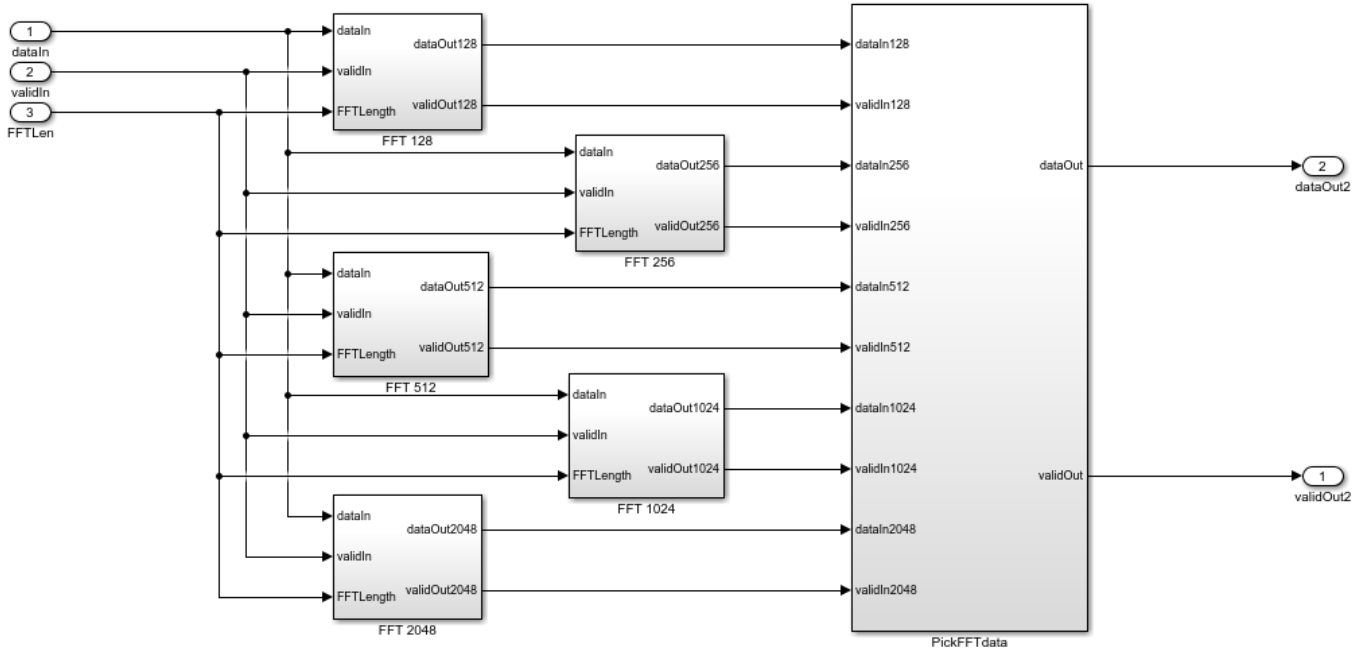
The **Data Controller** subsystem controls the input data so that the input to the **FFT HDL Optimized** block has data samples with zeros padded in between them. The **FFT HDL Optimized** block is configured for an FFT length of 2048, the largest FFT length required by the LTE standard. To simplify selection of the output bins, the FFT block is configured to output the samples in bit-natural order. The FFT length is specified through input port and is sampled at the start of the frame. The requested FFT length must be delayed to match the FFT latency. The FFT length is registered using the start output signal of the FFT and the generated end of the frame signal. This method avoids implementing a large delay-matching memory. Since the input data has zeros in between samples, the output of the large FFT contains repeated copies of the FFT length samples. To get the required FFT output, the first FFT length samples are collected from the FFT output. This operation is performed by modifying the output valid signal of the FFT using the **Bin selection** subsystem.

### Multiple FFTs for Reference

This subsystem is used as a reference to compare against the output of Variable Size FFT using Single FFT. The subsystem includes five different FFT blocks (FFT 128, FFT 256, FFT 512, FFT 1024, and FFT 2048) and one MATLAB Function block. The input data will be fed to all five FFTs. Depending on the requested FFT length, one of the five FFT blocks is activated and FFT operation is performed. The MATLAB function block `pickFFTData` selects the output from the appropriate FFT block. The output is saved to MATLAB for comparison with the output of the Variable-Size FFT using Single FFT.

```
open_system([modelName '/Multiple FFTs for Reference']);
```





### Run Simulink Model

The MATLAB script configures desired vector of FFT lengths, the size of the single FFT, and generates input data with a valid signal. It then runs the model, and compares the output of the two subsystems in MATLAB.

Run the model using the `sim` command on the MATLAB command line.

```
sim(modelname);
```

### Verification

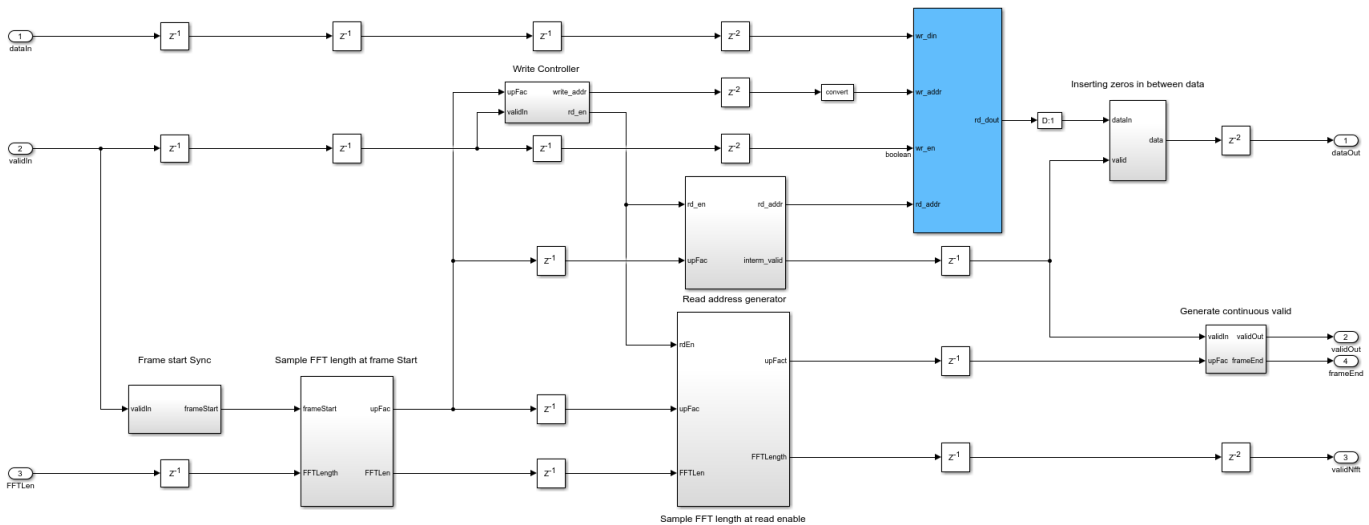
The output from both subsystems is sent to the MATLAB workspace and the difference is plotted. In this case, the output of the two subsystems are identical and the error between the two sets of values is 0.

```
dataOut1 = out1(:);
dataOut2 = out2(:);
figVSF = figure('Visible', 'off');
plot(abs(dataOut1-dataOut2));
title('Difference between the two outputs for fixed valid pattern')
xlabel('Sample Index');
ylabel('Error');
figVSF.Visible = 'on';
bdclose(modelname);
```



data is read from the other half of the memory. As a result, the total latency is increased by `fftLenSinFFT`.

```
modelName = 'VariableSizeFFTArbitraryValidPatternHDLExample';
load_system(modelName);
open_system([modelName '/Variable Size FFT using Single FFT/Data Controller']);
```



## Arbitrary Input Data and Valid Generation

For generating arbitrary data and valid inputs, users can select any of these three options: zero padding of fixed size in between data samples, zero padding at the end of data samples, and zero padding of random size in between data samples. The input data and valid generation for these three different zero padding patterns are shown below. The *VariableSizeFFTArbitraryValidPatternHDLExample* model uses the generated data and valid for simulation and verification.

```
% Initialization of input data and valid
dataIn = zeros(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
validIn = false(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
zeroPaddingPattern = 'InBetween'; % 'AtEnd', 'Random'
switch zeroPaddingPattern
case 'InBetween'
    % Zero padding in between input data samples
    for ind = 1:length(fftLenVecMulFFTs)
        % Generate data of FFT length samples
        randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)));
        % Zero padding in between input data samples
        upSamplingFac = fftLenSinFFT/fftLenVecMulFFTs(ind);
        dataIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(randInputData,upSamplingFac);
        % Valid corresponding to the generated data
        validIn((ind-1)*fftLenSinFFT+1:upSamplingFac:fftLenSinFFT*ind) = true;
    end
case 'AtEnd'
    % Zero padding at the end of input data samples
    for ind = 1:length(fftLenVecMulFFTs)
        % Generate data of FFT length samples
        randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)));
        % Zero padded data
```

```

        dataIn(((ind-1)*fftLenSinFFT+1):((ind-1)*fftLenSinFFT+fftLenVecMulFFTs(ind))) = randn(1,fftLenVecMulFFTs(ind));
        % Valid corresponding to data generated
        validIn(((ind-1)*fftLenSinFFT+1):((ind-1)*fftLenSinFFT+fftLenVecMulFFTs(ind))) = true;
    end
    otherwise % Random
        for ind =1:length(fftLenVecMulFFTs)
            % Zero padding at random
            randIndices = randperm(fftLenSinFFT);
            % Generate data of FFT length samples
            randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)));
            indices = randIndices(1:fftLenVecMulFFTs(ind));
            % If the random indices does not have the first sample
            if(sum(indices==1)==0)
                indices(1) = 1;
            end
            % Zero padded data
            dataIn(indices+(ind-1)*fftLenSinFFT) = randInputData;
            % Valid corresponding to data generated
            validIn(indices+(ind-1)*fftLenSinFFT) = true;
        end
    end
end

```

### Run the Simulink model

Before running the model, make sure that `dataIn`, `validIn`, `fftLenIn`, and the necessary variables are initialized.

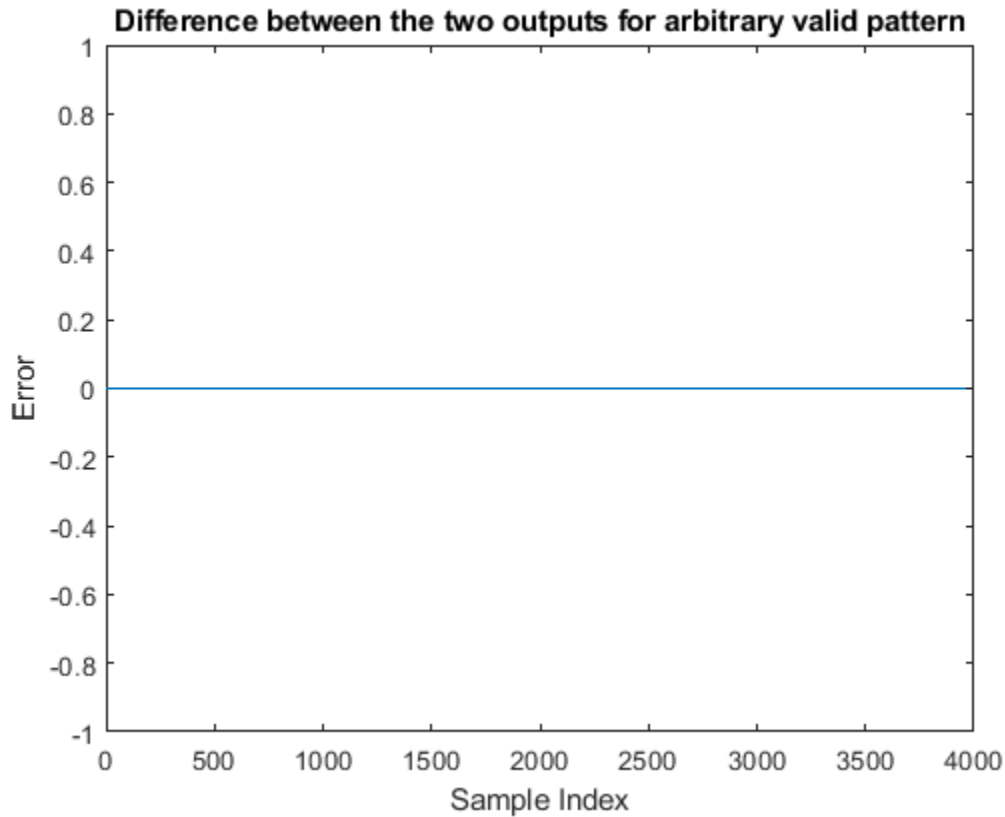
```
sim(modelname);
```

### Verification

```

dataOut1 = out1(:);
dataOut2 = out2(:);
figVSFAIV = figure('Visible', 'off');
plot(abs(dataOut1-dataOut2));
title('Difference between the two outputs for arbitrary valid pattern');
xlabel('Sample Index');
ylabel('Error');
figVSFAIV.Visible = 'on';
bdclose(modelname);

```



### HDL Code Generation and Verification

To generate the HDL code referenced in this example, an HDL Coder™ license is needed.

You can use the commands `makehdl` and `makehdltb` to generate the HDL code and the testbench for the subsystems.

HDL code generated for the Variable Size FFT subsystems were synthesized for the Xilinx® Zynq®-7000 ZC706 board. The synthesis results are shown in the following table.

<b>Hardware Type</b>	<b>Single FFT</b>	<b>Single FFT supporting arbitrary input valid pattern</b>	<b>Multiple FFTs</b>
Slice LUT	5580	5732	19736
Slice Registers	8035	8082	29303
RAMB36E1	2	6	4
RAMB18E1	18	18	48
DSP48E1	16	16	58
Max Freq (in MHz) Post P&R	265	237.6	243.1

The table above shows that implementing a variable-size FFT using a single FFT uses fewer hardware resources than using a multiple FFT solution. To support an arbitrary input valid pattern, the hardware implementation uses more RAM.

### **See Also**

FFT HDL Optimized

# Accelerate BER Measurement for Wireless HDL LTE Turbo Decoder

This example shows the workflow to measure the BER of the Wireless HDL Toolbox™ Turbo Decoder using `parsim` to parallelize the simulations across `EbNo` points. This approach can be used to accelerate other Monte Carlo simulations.

## Introduction

HDL implementations of reference applications are often complex and take a lot of time to simulate. As a result, figuring out the bit error rate (BER) performance by running multiple simulations at different SNR points can be very time consuming. One way to optimize this is to parallelize simulations using the `parsim` command. The `parsim` command runs multiple simulations in parallel when called with a Parallel Computing Toolbox™ license available. This example measures the BER of the Wireless HDL LTE Turbo Decoder. To achieve sufficient statistical accuracy, around 100 errors must be obtained at the decoder for each `EbNo` value. This translates to  $1e8$  bits at a BER of  $10e-6$ . This type of Monte Carlo simulation is a suitable candidate to parallelize using `parsim`, where the BER for every `EbNo` point is performed on workers in parallel.

For every parallel simulation, this example sets up the input data as follows:

- 1 Generate input data frames;
- 2 Turbo encode;
- 3 QPSK modulate;
- 4 Add AWGN based on the `EbNo` value;
- 5 Demodulate the noisy symbols;
- 6 Generate soft decisions.

The soft decisions become the input to the Wireless HDL LTE Turbo Decoder in Simulink®. The turbo decoded bits are compared to the transmitted bits to calculate the BER. Each parallel simulation sends the results back to the main host.

## Configure Parameters and Simulation Objects

The total number of information bits for each `EbNo` point, `bitsPerEbNo`, is divided over multiple simulations, defined by `parsimPerEbNo`. In this way, every simulation runs `bitsPerParsim` bits for a single `EbNo` point. The total number of simulations is `length(EbNo)*parsimPerEbNo`. This example is configured to run only a small number of bits for demonstration purposes. In a real scenario, you must run a sufficient number of samples through the decoder for an accurate measure of the BER at the higher `EbNo` points. When choosing these parameters, consider the memory resources available on the host. A large input data set per simulation or large number of workers could result in slow down or memory exhaustion. The structure `simParam` contains the parameters required for each simulation. This structure is sent to the simulations at a later stage.

```

EbNo = 0:0.1:1.1;
bitsPerEbNo = 1e5; %1e8;
parsimPerEbNo = 2; %10;
bitsPerParsim = ceil(bitsPerEbNo/parsimPerEbNo);

simParam.blkSize = 6144;
simParam.turboIterations = 6;
simParam.numFrames = ceil(bitsPerParsim/simParam.blkSize); % frames per simulation

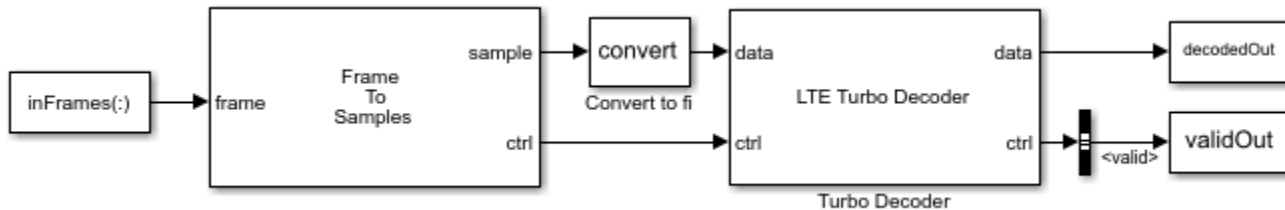
```

```

simParam.modScheme = 'QPSK';
simParam.bps = 2; % bits per symbol
tailBits = 4; % encoder property
simParam.encoderRate = simParam.blkSize/(3*(simParam.blkSize+tailBits)); % rate 1/3 Turbo code
simParam.samplesizeIn = floor(1/simParam.encoderRate); % 3 samples in at a time
simParam.inframeSize = simParam.samplesizeIn*(simParam.blkSize+tailBits);

model = 'LTEHDLTurboDecoderBERExample';
open_system(model);

```



Start a local parallel pool with minimum of 1 and maximum of `maxNumWorkers`. If a Parallel Computing Toolbox™ licence is not available, the simulations will be serialized. The actual size of the pool depends on the number of available cores. Each parallel worker gets assigned one core on which an independent MATLAB® session is launched.

```

maxNumWorkers = 3;
pool = parpool('local', [1 maxNumWorkers]);

```

```

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 3).

```

Preallocate a `parsim` object to hold the data required for each simulation. The object can also include handles to functions, which the model calls before or after a simulation. The MATLAB® session on which `parsim` is executed acts as the main host. The main host is responsible for launching the simulations on the workers, sending the required data to every worker, and receiving the results.

```

parsimIn(1:length(EbNo)*parsimPerEbNo) = Simulink.SimulationInput(model);

```

Replicate `EbNo` points to set up `parsimPerEbNo` simulations.

```

repEbNo = repmat(EbNo,parsimPerEbNo,1);
repEbNo = repEbNo(:);

```

Minimizing data transmission to the workers improves the performance and stability of the main host. Therefore, this example generates the input data in-model, rather than passing the large input data set to each worker. Input data is generated using the pre-simulation function, `presimGenInput` and the BER calculation is also performed in the post-simulation function, `postsimOutput`. These function handles are assigned to each `SimulationInput` object. The post-simulation function is assigned inside the pre-simulation function as shown in the section Pre-Simulation and Post-Simulation Functions.

```

for noiseRatio = 1:length(repEbNo)
    % Calculate the noise variance.
    EsNo = repEbNo(noiseRatio) + 10*log10(simParam.bps);
    snrdB = EsNo + 10*log10(simParam.encoderRate);
    noiseVar = 1./(10.^(snrdB/10));

```



```

% Use random but reproducible data.
seed = noiseRatio;

% For Rapid Accelerator mode, set the simulation
% stop time before compilation.
parsimIn(noiseRatio) = parsimIn(noiseRatio).setModelParameter('StopTime', num2str(simParam.numRuns));

% Set pre-simulation function.
parsimIn(noiseRatio) = parsimIn(noiseRatio).setPreSimFcn(@(simIn) presimGenInput(simIn, noiseRatio));
end

```

Run and show progress of the simulations in the command window. At the end of the simulations, the results are sent back to the main host in an array of structures, `parsimOut`, with one entry created per simulation. Once simulations are complete, shut down the parallel pool.

```

parsimOut = parsim(parsimIn, 'ShowProgress', 'on', 'StopOnError', 'on');
delete(pool);

```

```

[21-Nov-2019 19:28:49] Checking for availability of parallel pool...
[21-Nov-2019 19:28:50] Starting Simulink on parallel workers...
[21-Nov-2019 19:29:07] Configuring simulation cache folder on parallel workers...
[21-Nov-2019 19:29:08] Loading model on parallel workers...
[21-Nov-2019 19:29:15] Running simulations...
Analyzing and transferring files to the workers ...done.
[21-Nov-2019 19:31:06] Completed 1 of 24 simulation runs
[21-Nov-2019 19:31:06] Completed 2 of 24 simulation runs
[21-Nov-2019 19:31:06] Completed 3 of 24 simulation runs
[21-Nov-2019 19:31:14] Completed 4 of 24 simulation runs
[21-Nov-2019 19:31:14] Completed 5 of 24 simulation runs
[21-Nov-2019 19:31:14] Completed 6 of 24 simulation runs
[21-Nov-2019 19:31:21] Completed 7 of 24 simulation runs
[21-Nov-2019 19:31:21] Completed 8 of 24 simulation runs
[21-Nov-2019 19:31:21] Completed 9 of 24 simulation runs
[21-Nov-2019 19:31:27] Completed 10 of 24 simulation runs
[21-Nov-2019 19:31:27] Completed 11 of 24 simulation runs
[21-Nov-2019 19:31:27] Completed 12 of 24 simulation runs
[21-Nov-2019 19:31:34] Completed 13 of 24 simulation runs
[21-Nov-2019 19:31:34] Completed 14 of 24 simulation runs
[21-Nov-2019 19:31:34] Completed 15 of 24 simulation runs
[21-Nov-2019 19:31:41] Completed 16 of 24 simulation runs
[21-Nov-2019 19:31:41] Completed 17 of 24 simulation runs
[21-Nov-2019 19:31:41] Completed 18 of 24 simulation runs
[21-Nov-2019 19:31:47] Completed 19 of 24 simulation runs
[21-Nov-2019 19:31:47] Completed 20 of 24 simulation runs
[21-Nov-2019 19:31:47] Completed 21 of 24 simulation runs
[21-Nov-2019 19:31:54] Completed 22 of 24 simulation runs
[21-Nov-2019 19:31:54] Completed 23 of 24 simulation runs
[21-Nov-2019 19:31:54] Completed 24 of 24 simulation runs
[21-Nov-2019 19:31:54] Cleaning up parallel workers...
Parallel pool using the 'local' profile is shutting down.

```

### Plot BER

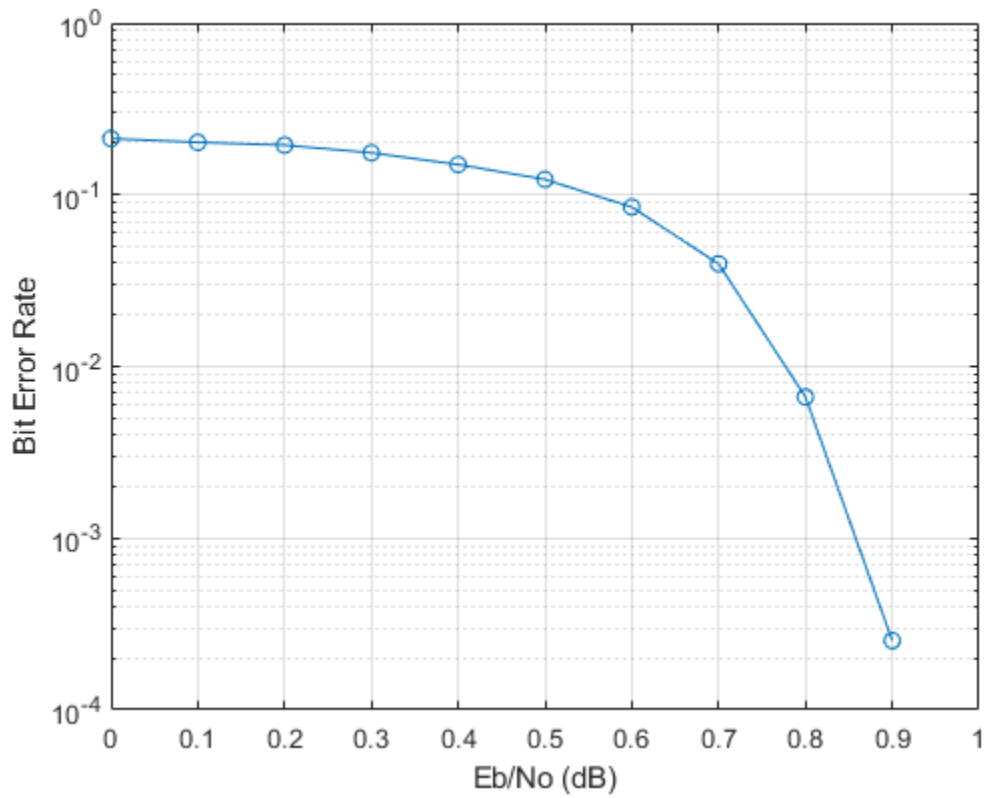
Extract the BER values from the array of structures. Combine the BER results for each `EbNo` point and find the average BER per `EbNo` point.

```

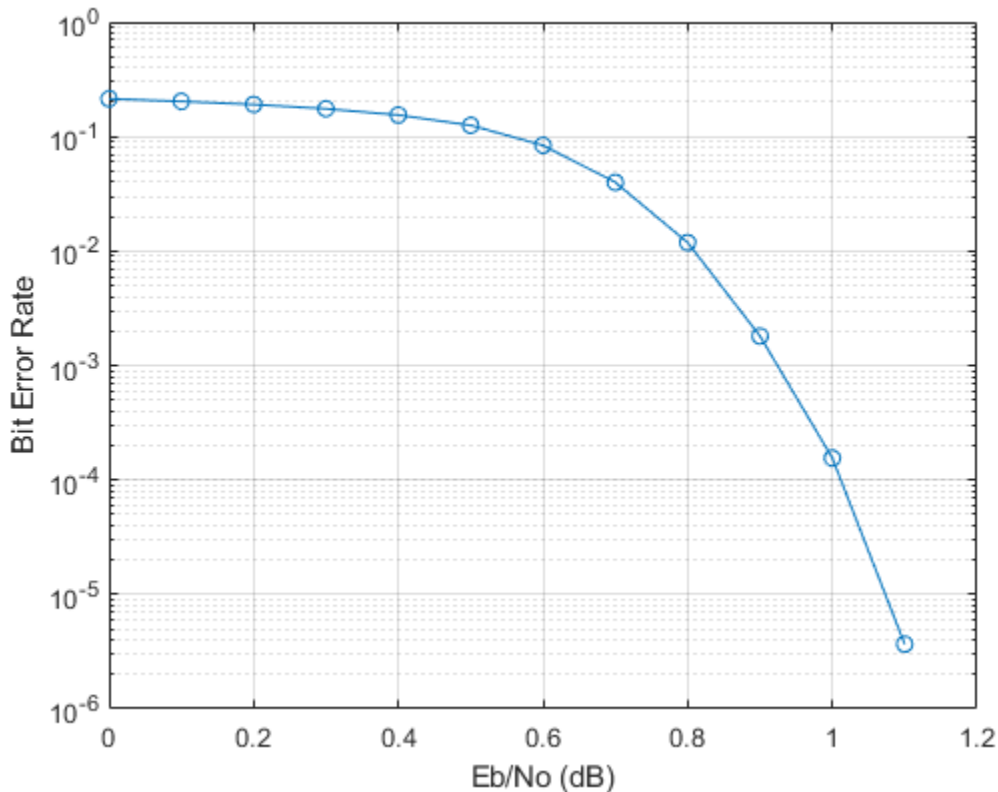
BER = [parsimOut(:).BER];
BER = transpose(reshape(BER, parsimPerEbNo, length(BER)/parsimPerEbNo));

```

```
avgBER = mean(BER,2);  
semilogy(EbNo,avgBER,'-o');  
grid;  
xlabel('Eb/No (dB)');  
ylabel('Bit Error Rate');
```



The plot below shows the results of the BER measurement with  $\text{bitsPerEbNo} = 1e8$ .



### Pre-Simulation and Post-Simulation Functions

These functions independently generate input data and process output data for each simulation, which eliminates the need for the main host to store the data in memory for all simulations. The `presimGenInput` function generates input bits, then encodes, modulates and converts them to soft decisions. To make the input frames and parameters available to the model, they are assigned as variables in the global workspace using the `setVariable` function.

```
function simIn = presimGenInput(simIn,noiseVar,seed,simParam)

    rng(seed);

    % Preallocate arrays for speed.
    txBits = zeros(simParam.blkSize,simParam.numFrames,'int8');
    inFrames = zeros(simParam.inframeSize,simParam.numFrames,'single');

    % Generate input frames, turbo encode, modulate and add noise
    % based on noise variance.
    for currentFrame = 1:simParam.numFrames
        txBits(:,currentFrame) = randi([0 1],simParam.blkSize,1);
        codedData = lteTurboEncode(txBits(:,currentFrame));
        txSymbols = lteSymbolModulate(codedData,simParam.modScheme);
        noise = (sqrt(noiseVar/2))*complex(randn(size(txSymbols)),randn(size(txSymbols)));
        rxSymbols = txSymbols + noise;
        inFrames(:,currentFrame) = lteSymbolDemodulate(rxSymbols,simParam.modScheme,'Soft');
    end
end
```

```
% Set up parameters for Frame to Samples block to serialize data.
% Leave sufficient gap between frames.
simParam.idleCyclesBetweenSamples = 0;
halfIterationLatency = (ceil(simParam.blkSize/32)+3)*32; % window size = 32
algFrameDelay = 2*simParam.turboIterations*halfIterationLatency+(simParam.inframeSize/simPara
simParam.idleCyclesBetweenFrames = algFrameDelay;

% Assign variables to global workspace.
simIn = simIn.setVariable('inFrames',inFrames);
simIn = simIn.setVariable('simParam',simParam);

% Set post-simulation function and send required data.
simIn = simIn.setPostSimFcn(@(simOut) postsimOutput(simOut,txBits,simParam));
```

end

The post-simulation function receives the outputs of the simulation and computes the BER. The results are stored in a structure `results` which `parsim` returns as `parsimOut`.

```
function results = postsimOutput(out, txBits, simParam)
    decodedOutValid = out.decodedOut(out.validOut);

    results.numErrors = sum(xor(txBits(:),decodedOutValid));
    results.BER = results.numErrors/(simParam.numFrames*simParam.blkSize);
```

end

## Conclusion

This example showed how to efficiently measure the BER curve for the Wireless HDL LTE Turbo Decoder using `parsim`. If a parallel pool is not used, the linear time to complete the simulations would be approximately 16 hours. As a result of parallelization, the time to run all simulations came down to 5.4 hours, using 3 workers. This was achieved by running the simulations in Rapid Accelerator mode. This workflow can be applied to complex reference applications which require Monte Carlo or other simulations.

## Encode message to RS codeword

This example shows how to use the RS Encoder block to encode a message to a Reed-Solomon (RS) codeword. In this example, a set of random inputs frames are generated and provided to the `comm.RSEncoder` function. Using the `whd1FramesToSamples` function, these frames are converted into samples and provided as input to the RS Encoder block. The output of the RS Encoder block is then compared with the output of the `comm.RSEncoder` function to check whether the encoded output codeword for the given input message is same. By default, the puncturing option is disabled in this example. To enable puncturing, set the puncturing value to `true`. This example model supports HDL code generation for the RS Encoder subsystem.

### Set Up Input Data Parameters

Set up these workspace variable for the models to use. These variables configure the RS Encoder block inside the model.

```
nMessages = 3;
n = 255; % Specify codeword length
k = 239; % Specify message length
m = n-k; % Parity length
inDataType = fixdt(0,ceil(log2(n)),0);
puncturing = false; % true for puncturing
puncturePattern = randsrc(m,1,[0 1]); % Considered, when punturing is true
shortMsg = false; % true for shortened message
k1 = k-1; % Considered when shortMsg is true
```

### Generate Random Input Samples

Generate random samples using `n`, `k`, and `m` variables and provide those generated samples as input to the `comm.RSEncoder` function.

```
hRSEnc = comm.RSEncoder;
hRSEnc.CodewordLength = n;
hRSEnc.MessageLength = k;

if isequal(shortMsg,true)
    hRSEnc.ShortMessageLength = k1;
else
    k1 = k;
end

if isequal(puncturing,true)
    hRSEnc.PuncturePatternSource = "Property";
    hRSEnc.PuncturePattern = puncturePattern;
    puncLen = n-k-sum(hRSEnc.PuncturePattern);
else
    puncLen = 0;
end

data = cell(1,nMessages);
refData = (zeros(k1+m-puncLen,nMessages));

for ii = 1:nMessages
    data{ii} = randi([0 n],k1,1);
    refData(:,ii) = hRSEnc(data{ii});
end
```

```
refOutput = refData(:);
```

### Generate Input Control Samples for the Simulink® Model

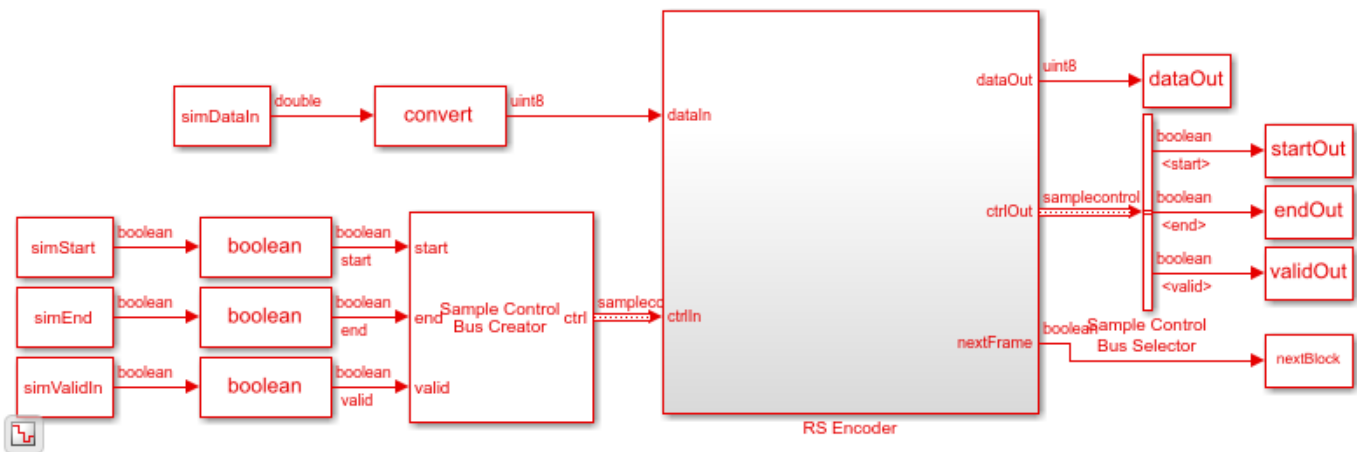
```
gapBetweenFrames = n-k;
gapBetweenSamples = 0;
```

```
[simDataIn, ctrlIn] = whdlFramesToSamples(data,gapBetweenSamples,gapBetweenFrames);
simStart = ctrlIn(:,1);
simEnd = ctrlIn(:,2);
simValidIn = ctrlIn(:,3);
stopTime = length(simValidIn);
```

### Run Simulink Model

Run the Simulink model. The block imports the workspace variables and generates the output.

```
modelName = 'HDLRSEncoder';
open_system(modelname);
if isequal(puncturing,true)
    set_param([modelName '/RS Encoder/RS Encoder'],'PuncturePatternSource','on');
    set_param([modelName '/RS Encoder/RS Encoder'],'PuncturePattern',[' ' num2str(puncturePattern)
end
out = sim(modelname);
```



### Export the Simulink Block Output to the MATLAB® Workspace

The encoded samples from the RS Encoder block are exported to the MATLAB® workspace.

```
simOutput = dataOut(validOut);
```

### Compare the Simulink Block Output with the MATLAB Function Output

Capture the output of the RS Encoder block. Compare that output with the output of the comm.RSEncoder function.

```
fprintf('\nHDL RS Encoder\n');
difference = double(simOutput) - double(refOutput);
fprintf('\nTotal Number of samples differed between Simulink block output and MATLAB function output\n');
```

HDL RS Encoder

Total Number of samples differed between Simulink block output and MATLAB function output is: 0

## **See Also**

### **Blocks**

RS Encoder

## HDL Implementation of AWGN Generator

This example shows the implementation of an additive white Gaussian noise (AWGN) generator that is optimized for HDL code generation and hardware implementation. The hardware implementation of AWGN accelerates the performance evaluation of wireless communication systems using an AWGN channel. In this example, the Simulink® model accepts signal-to-noise ratio (SNR) values as inputs and generates Gaussian random noise along with valid signal. The example supports SNR input ranges from -20 to 31 dB in steps of 0.1 dB.

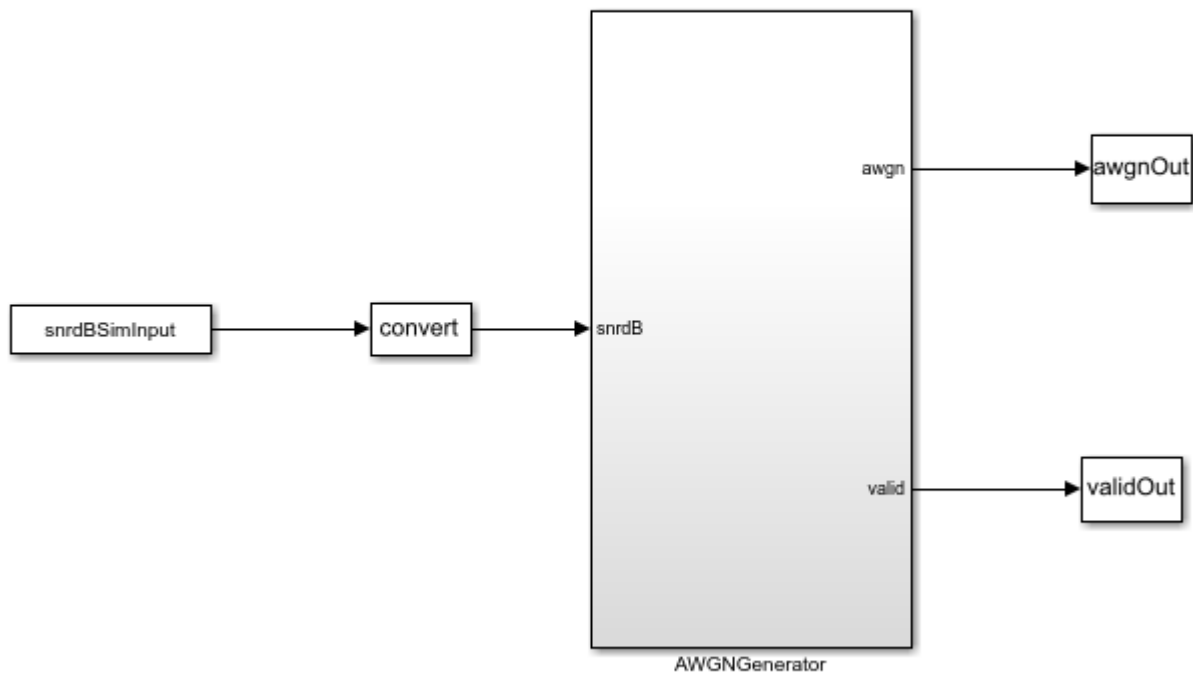
Modern wireless communication systems includes many different simulation parameters, such as channel bandwidth, modulation type, and code rate. The performance evaluation of these systems with these simulation parameters is a bottleneck. Hardware capabilities of FPGAs can speed up simulations.

### Model Architecture

`% Run this command to open the HDLAWGNGenerator model.`

```
modelName = 'HDLAWGNGenerator';
open_system(modelName);
```

## HDL AWGN Generator



Copyright 2020 The MathWorks, Inc.

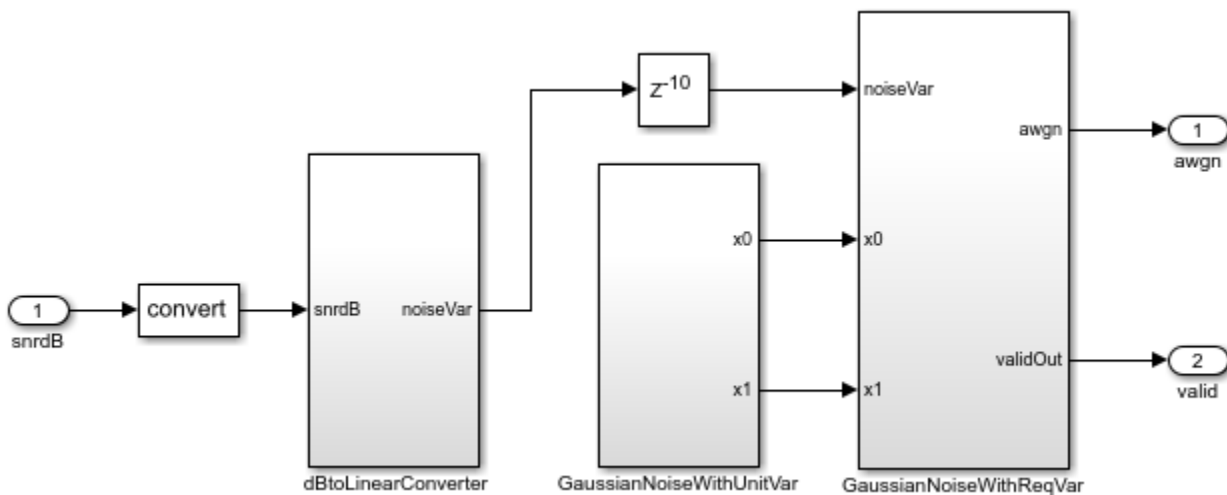


This example demonstrates the implementation of an AWGN generator based on the Box-Muller method. The Box-Muller method is widely adopted for Gaussian noise generation because of its hardware-friendly architecture and constant output rate. The top-level structure of the model includes these three subsystems.

- SNR dB to Linear Scale Converter
- Gaussian Noise Generator with Unit Variance
- Gaussian Noise Generator with Required Variance

`% Run this command to open the subsystems inside AWGNGenerator model.`

```
open_system([modelName '/AWGNGenerator']);
```

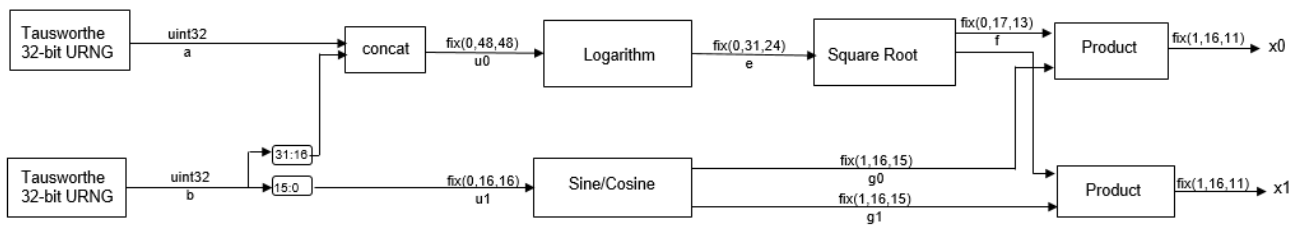


### SNR dB to Linear Scale Converter

The dBtoLinearConverter subsystem takes an SNR value in dB as input and converts it into noise variance in a linear scale. This noise power is used to multiply the output of the Gaussian noise with unit variance. This lookup table approach is used for converting an SNR value in dB to a noise power value in a linear scale. During the conversion, the signal power is assumed to be 1. This subsystem has a latency of 1 clock cycle.

### Gaussian Noise Generator with Unit Variance

The GaussianNoiseWithUnitVar subsystem generates Gaussian noise with unit variance by using the Box-Muller method. The Box-Muller method uses two uniformly distributed random variables to generate two normally distributed random variables through a series of logarithmic, square root, sine, and cosine operations as shown in this figure. Those two uniformly distributed random variables are generated using the Tausworthe algorithm.



### Implementation of HDL Tausworthe Uniform Random Number

The Tausworthe Uniform Random Number Generator module is used to generate two 32-bit uniform random integers. Each 32-bit uniform random number with improved statistical properties is obtained by combining three linear feedback shift register (LFSR) based uniform random number generators (URNGs). This implementation requires these two seeds: TausURNG1 and TausURNG2. The `whdlexamples.hdlawgnGen_init.m` script file initializes these seeds.

The ConcatandExtract subsystem accepts 32-bit uniform random integers,  $a$  and  $b$ , to generate two uniform random numbers,  $u0$  and  $u1$ , in the range  $[0, 1)$  with bit-widths 48 and 16, respectively.  $u0$  is generated by concatenating the 32-bit value of  $a$  and higher 16 bits of  $b$ . Uniform random number  $u1$  is generated by extracting the lower 16 bits of  $b$ .

```
open_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen']);
close_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen']);
open_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen/TausURNG1']);
close_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen/TausURNG1']);
```

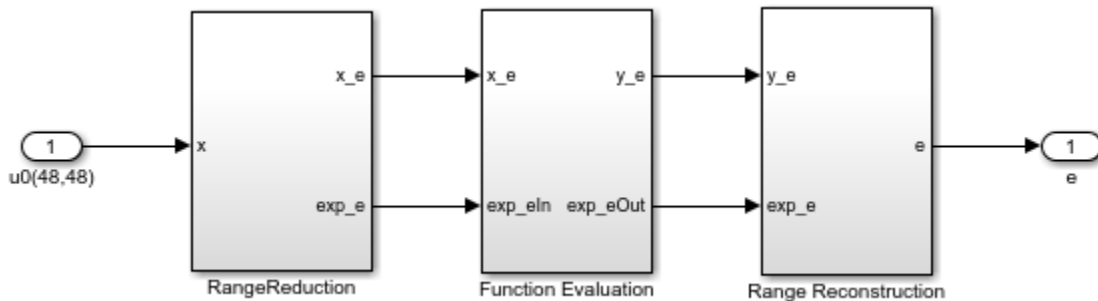
### Implementation of HDL Logarithm

HDL logarithm subsystem evaluates the approximate logarithm based on the piecewise linear polynomial method. This module has latency of 3 clock cycles. Implementation of the HDL logarithm involves these three steps.

- 1 Range Reduction - In this step, the original range of the input, which is  $[0, 1-2^{(-48)})$  is reduced to a more convenient smaller range of  $[1, 2)$ . The log function is approximated on the reduced range in the next step.
- 2 Function Evaluation - The log function is approximated over 256 equally spaced segments in the range  $[1, 2)$  by using a second-degree polynomial. Coefficients of the second-degree polynomial are obtained using the `polyfit` function. These coefficients are stored in a lookup table, which is indexed using the first 8 bits of input to the function evaluation block.
- 3 Range Reconstruction - The result of the function evaluation is expanded back to the original range. A bit left shift operation is used for range reconstruction and to implement the  $-2*\log$  function.

Run this command to open HDL logarithm subsystem.

```
open_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/logImplementation/log']);
```

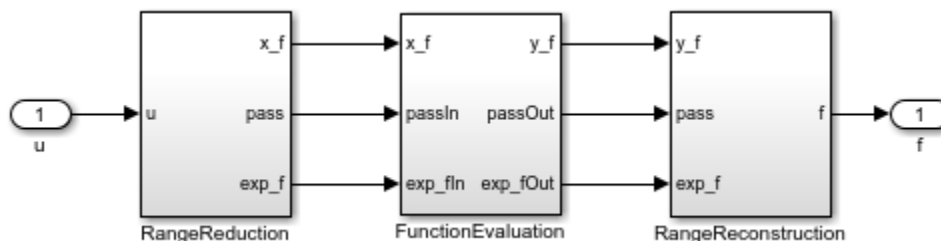


### Implementation of HDL Square Root

The HDL Square root subsystem evaluates approximate square root based on the piecewise linear polynomial method. This module has a latency of 2. The implementation of the HDL square root involves these three steps.

- 1 Range Reduction - The input data type to the module is  $fi(0, 31, 24)$ . This range is reduced to a smaller range of  $[1, 4)$ . The square root function is approximated on the reduced range in the next step.
- 2 Function Evaluation - The square root function is approximated over 64 equally spaced segments in the range  $[1, 2)$  and  $[2, 4)$  by using a first-degree polynomial. Coefficients of the first-degree polynomial are stored in a lookup table, which is indexed using the first 6 bits of input to the function evaluation block.
- 3 Range Reconstruction - The result of the function evaluation is expanded back to the original range using a left shift operation.

```
close_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/logImplementation/log']);
open_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/SqrtImplementation/SqrtEval']);
```



### Implementation of HDL Sine and Cosine

The HDL optimized implementation of a sine or cosine function uses a lookup table approach. **Sin** and **Cos** are implemented using the existing Sine HDL Optimized (HDL Coder) and Cosine HDL Optimized (HDL Coder) blocks in the HDL Coder / Lookup Tables library.

```
close_system([modelName '/AWGNGenerator/GaussianNoiseWithUnitVar/SqrtImplementation/SqrtEval']);
```

### Gaussian Noise Generator with Required Variance

The GaussianNoiseWithReqVar subsystem converts Gaussian noise with unit variance to Gaussian noise with required variance. This subsystem takes inputs from dBToLinearConvertor and GaussianNoiseWithUnitVar subsystems. The linear noise variance obtained from

dBToLinearConvertor is multiplied with normally distributed random variables obtained from GaussianNoiseWithUnitVar.

### Results and Plots

The whdlexamples.hdlawgnGen\_init.m script file is used to specify the SNR range, generate the required number of noise samples, initialize the seeds for TausURNG1 and TausURNG2 subsystem and to generate coefficients for the function evaluation of the HDL log and square root.

The whdlexamples.hdlawgnGen\_init.m script file is the initialization function of HDLAWGNGenerator model. This function generates the input data and initializes the seeds for tausURNG and coefficients for the function evaluation. Simulate HDLAWGNGenerator.slx to generate  $10^6$  valid AWGN samples for each SNR of 5 dB and 15 dB. The implementation is pipelined to maximize the synthesis frequency, generating AWGN with an initial latency of 11. Plot the probability density function (PDF) of the AWGN output.

```
latency = 11;
NumOfSamples = 10^6;

% Simulate the model
open_system('HDLAWGNGenerator');
set_param(gcs, 'SimulationMode', 'Accel');
fprintf('\n Simulating HDL AWGN Generator...\n');
outSimulink = sim('HDLAWGNGenerator', 'ReturnWorkspaceOutputs', 'on');
fprintf('\n Simulation complete.\n');
awgnSimulink = outSimulink.awgnOut;

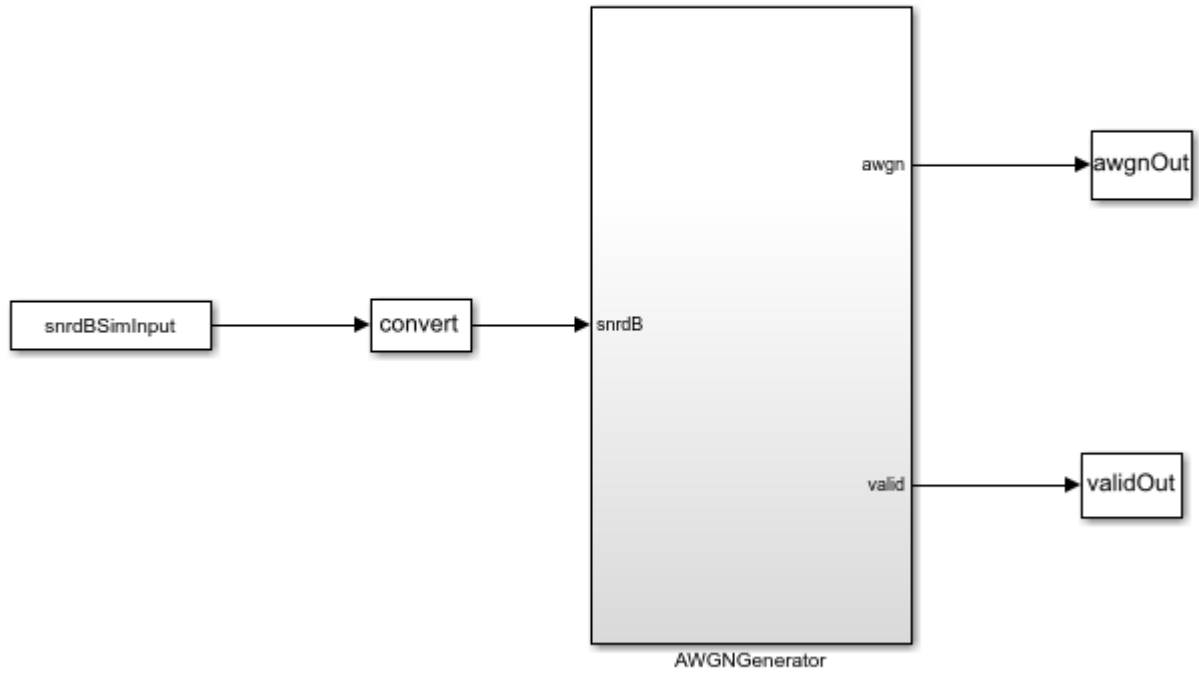
% Plot PDF
figure;
title('PDF for Real Part of AWGN');
hold on
histogram(real(awgnSimulink(latency+1:NumOfSamples+latency)),500, ...
    'Normalization','pdf','BinLimits',[-2 2],'FaceColor','blue', ...
    'EdgeColor','none');
histogram(real(awgnSimulink(NumOfSamples+latency+1:end)),500,...
    'Normalization','pdf','BinLimits',[-2 2],'FaceColor','yellow', ...
    'EdgeColor','none');
legend('5 dB SNR','15 dB SNR');

figure;
title('PDF for Imaginary Part of AWGN');
hold on
histogram(imag(awgnSimulink(latency+1:NumOfSamples+latency)),500, ...
    'Normalization','pdf','BinLimits',[-2 2],'FaceColor','blue', ...
    'EdgeColor','none');
histogram(imag(awgnSimulink(NumOfSamples+latency+1:end)),500, ...
    'Normalization','pdf','BinLimits',[-2 2],'FaceColor','yellow', ...
    'EdgeColor','none');
legend('5 dB SNR','15 dB SNR');
```

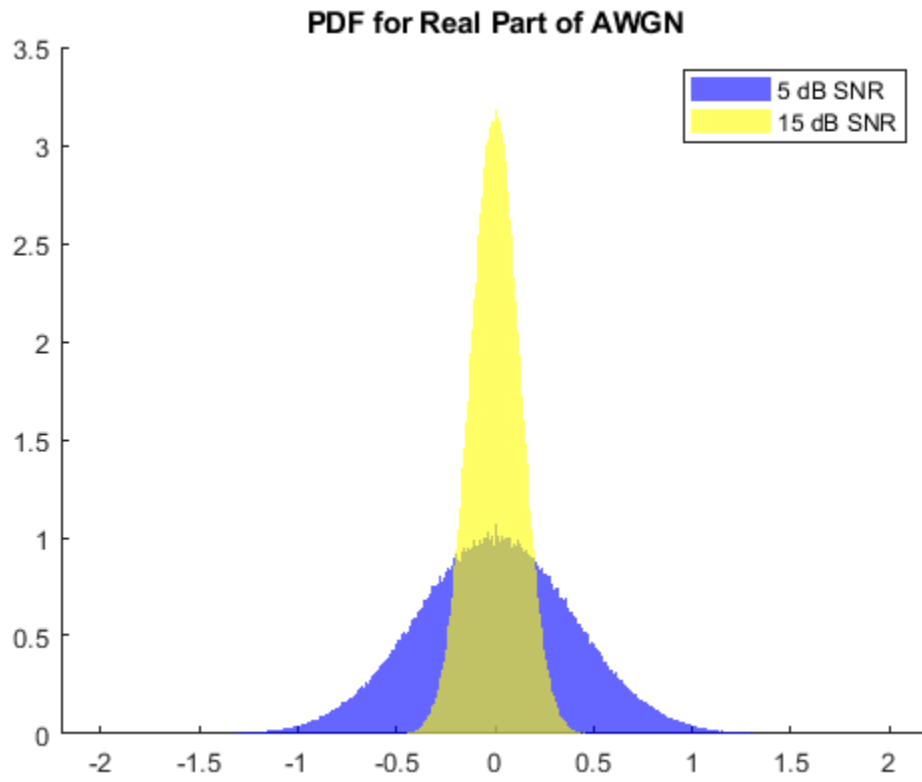
Simulating HDL AWGN Generator...

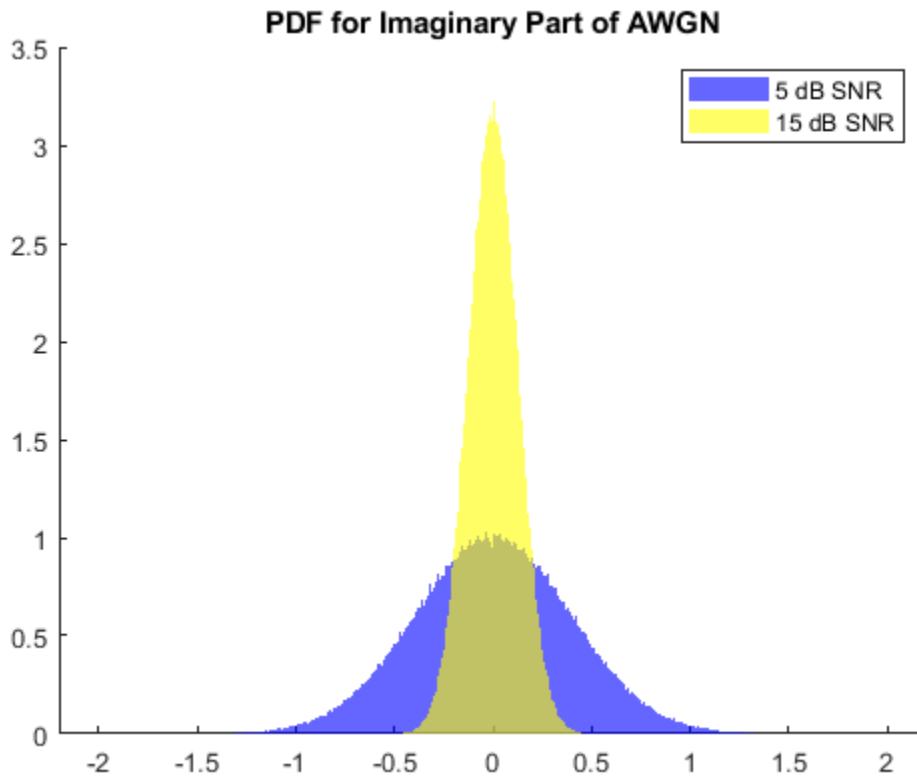
Simulation complete.

# HDL AWGN Generator



Copyright 2020 The MathWorks, Inc.





## Verification

Compare the output of the AWGN Simulink model with the output of the HDL equivalent AWGN MATLAB® function.

```

NumOfSamples = 1000;
% MATLAB output
fprintf('\n Simulating MATLAB HDL AWGN Generator for comparison...\n');
awgnMatlab=whdlexamples.hdlawgn(snrdBSimInput(1:NumOfSamples),seedsURNG1,seedsURNG2);
fprintf('\n Simulation complete. \n')

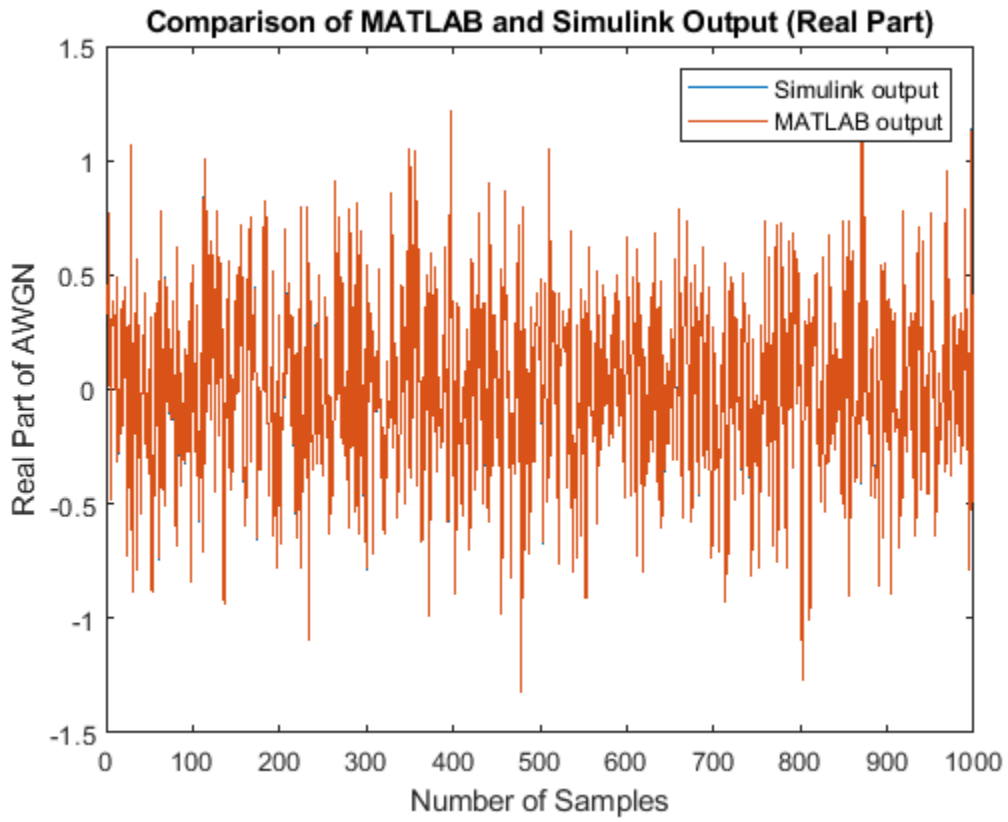
% Compare MATLAB and Simulink outputs
figure;
ax=axes('FontSize', 20);
plot(1:1000,real([awgnSimulink(latency+1:NumOfSamples+latency) awgnMatlab]));
xlabel(ax,'Number of Samples');
ylabel(ax,'Real Part of AWGN');
title(ax,'Comparison of MATLAB and Simulink Output (Real Part)');
legend('Simulink output','MATLAB output');

figure;
ax=axes('FontSize', 20);
plot(1:1000,imag([awgnSimulink(latency+1:NumOfSamples+latency) awgnMatlab]));
xlabel(ax,'Number of Samples');
ylabel(ax,'Imaginary Part of AWGN');
title(ax,'Comparison of MATLAB and Simulink Output (Imaginary Part)');
legend('Simulink output','MATLAB output');

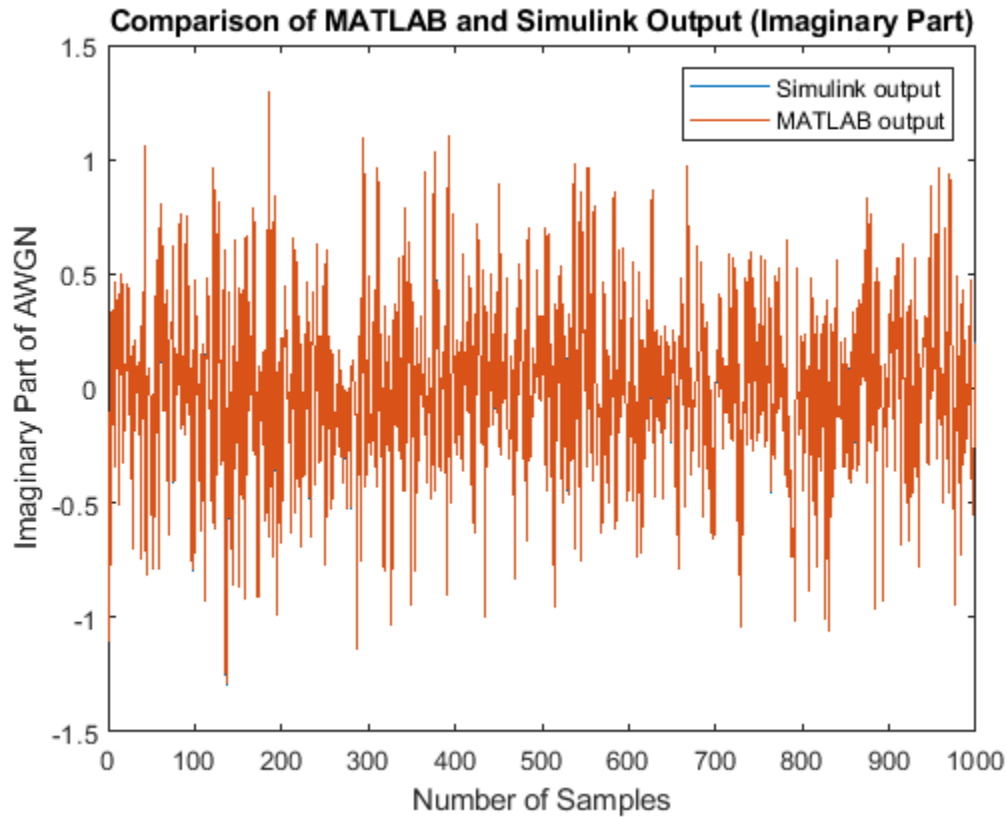
```

Simulating MATLAB HDL AWGN Generator for comparison...

Simulation complete.







### HDL Code Generation

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, enter this command at the MATLAB command prompt.

```
makehdl('HDLAWGNGenerator/AWGNGenerator')
```

To generate a test bench, enter this command at the MATLAB command prompt.

```
makehdltb('HDLAWGNGenerator/AWGNGenerator')
```

In this example, HDL code generated for the AWGNGenerator module is implemented for the Xilinx® Zynq®-7000 ZC706 board. The implementation results are shown in this table.

Hardware Type	Usage
Slice LUT	6171
Slice Registers	1668
RAMB18E1	9
DSP48E1	16
Max Freq (MHz)	250

**References**

1. J.D. Lee, J.D. Villasenor, W. Luk, and P.H.W. Leong. "A Hardware Gaussian Noise Generator Using the Box-Muller Method and Its Error Analysis," 659-71. IEEE, 2006. <https://doi.org/10.1109/TC.2006.81>.

## HDL Implementation of Digital Predistorter

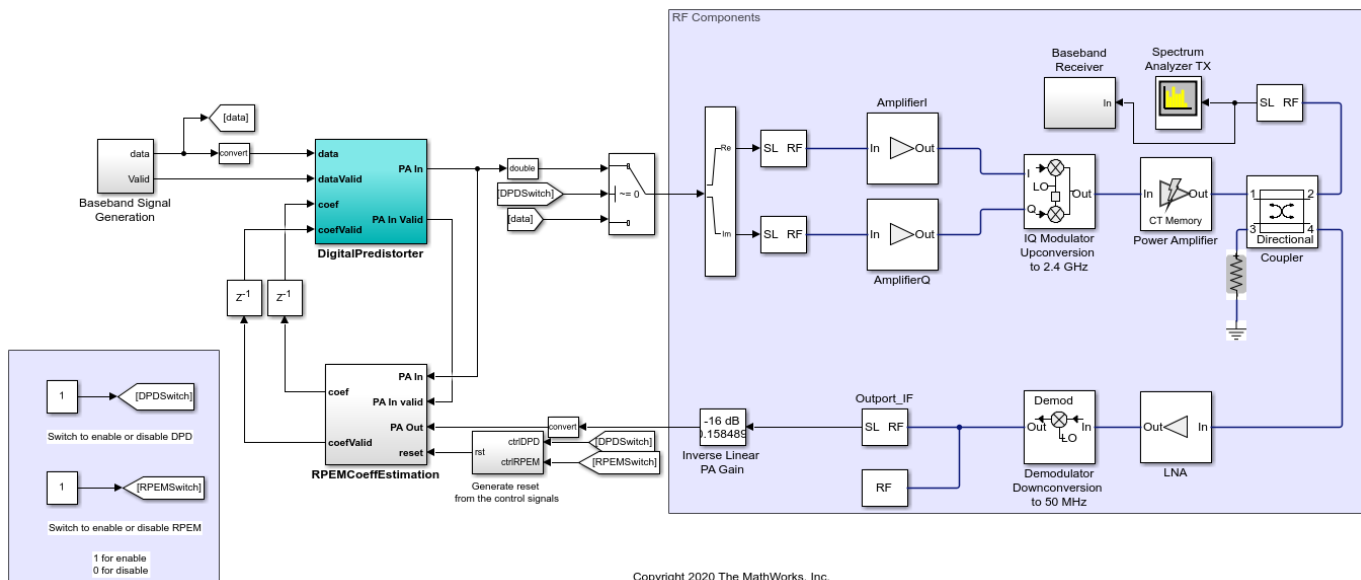
This example shows the implementation of a digital predistorter (DPD) model that is optimized for HDL code generation and hardware implementation. The predistortion mechanism is executed in two stages. In the first stage, a set of DPD coefficients are estimated based on the input and output data of the power amplifier (PA). In the second stage, the input data of the PA is predistorted based on the estimated DPD coefficients and provided as new input to the PA. This example demonstrates a system-level simulation in which the DigitalPredistorter subsystem generates HDL code, while the DPD coefficient estimation generates C/C++ code. This example model supports only Normal and Accelerator simulation modes.

### Digital Predistortion

Digital predistortion is a baseband signal processing technique that is used for correcting impairments in radio frequency (RF) power amplifiers. These impairments cause out-of-band emissions or spectral regrowth and in-band distortion, which results in an increased bit error rate (BER) and a decreased throughput of the system. Power amplifiers cause unwanted effects in the system due to their nonlinear behavior. Communication systems using orthogonal frequency division multiplexing (OFDM), such as a wireless local area network (WLAN), worldwide interoperability for microwave access (WiMax), long term evolution (LTE), and 5th generation mobile network (5G), are vulnerable to these unwanted effects. A precorrection is applied on the signal so that the cascade of the DPD and PA is close to an ideal, linear, and memoryless system. This linearization can improve PA power efficiency and can be more spectrum efficient. This figure shows the top-level structure of the example.

Run this command to open the example.

```
modelName = 'DPDHDLExample';
open_system(modelName);
```



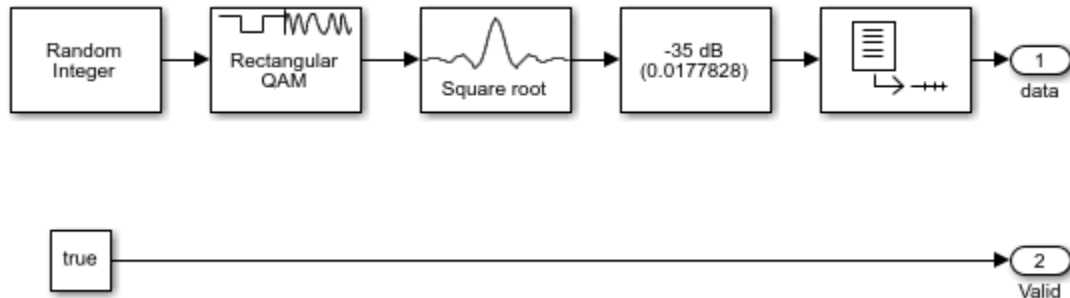
Copyright 2020 The MathWorks, Inc.

### Baseband Signal Generation

The Baseband Signal Generation subsystem generates a baseband signal that is provided as input data to the digital predistorter. Random input data is generated using a random integer signal

generator and is mapped to 16-QAM modulation using a rectangular QAM modulator. The modulated symbols are passed through a root raised cosine filter with a roll-off factor of 0.2. After applying an appropriate amplitude gain on the filtered data, the amplified data is provided as input to the DPD subsystem. You can also replace this Baseband Signal Generation subsystem with any custom transmitter to provide data to the DPD subsystem. This figure shows the baseband input signal generation for this example. Run this command to open the Baseband Signal Generation subsystem.

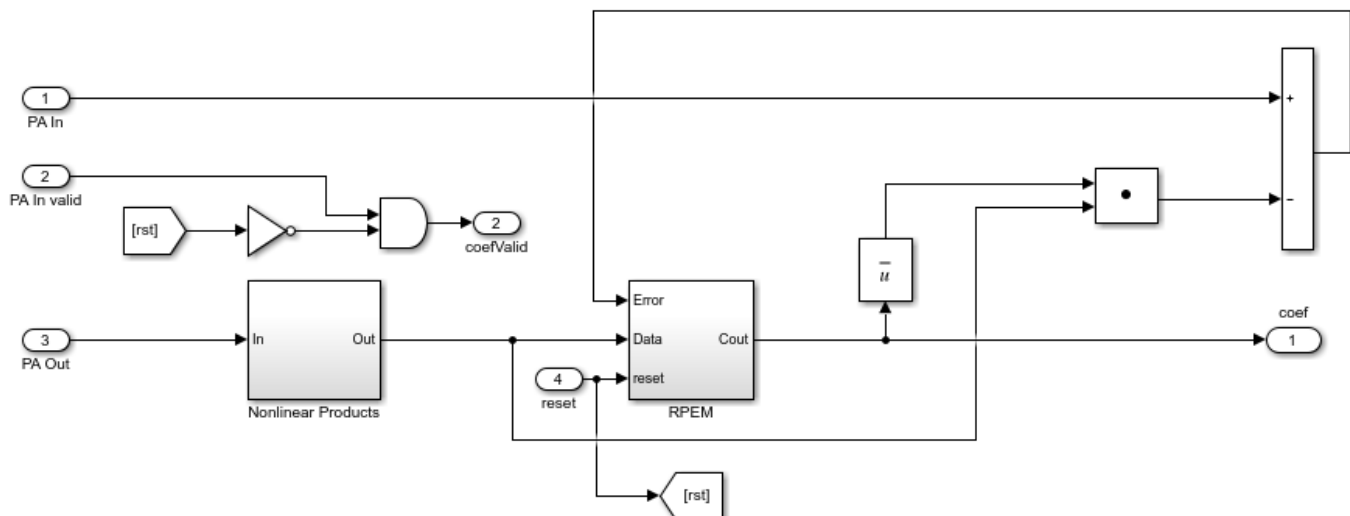
```
load_system(modelname);
open_system([modelname '/Baseband Signal Generation']);
```



### Coefficients Estimation

The RPEMCoeffEstimation subsystem estimates a set of coefficients by collecting data from the input and output of the PA. These coefficients are used to distort the signal before the power amplifier. PA characteristics vary over time and operating conditions, so an adaptive recursive prediction error method (RPEM) algorithm is used to estimate the DPD coefficients. The number of coefficients to be estimated depends on the memory depth and polynomial degree of the PA. In this example, because the total number of coefficients that need to be estimated is 25, the memory depth and polynomial degree of the PA are set to 5. For more information about the RPEM, see [ 1 ]. To generate C/C++ code for RPEMCoeffEstimation subsystem, use the `rtwbuild` command. Run this command to open the RPEMCoeffEstimation subsystem.

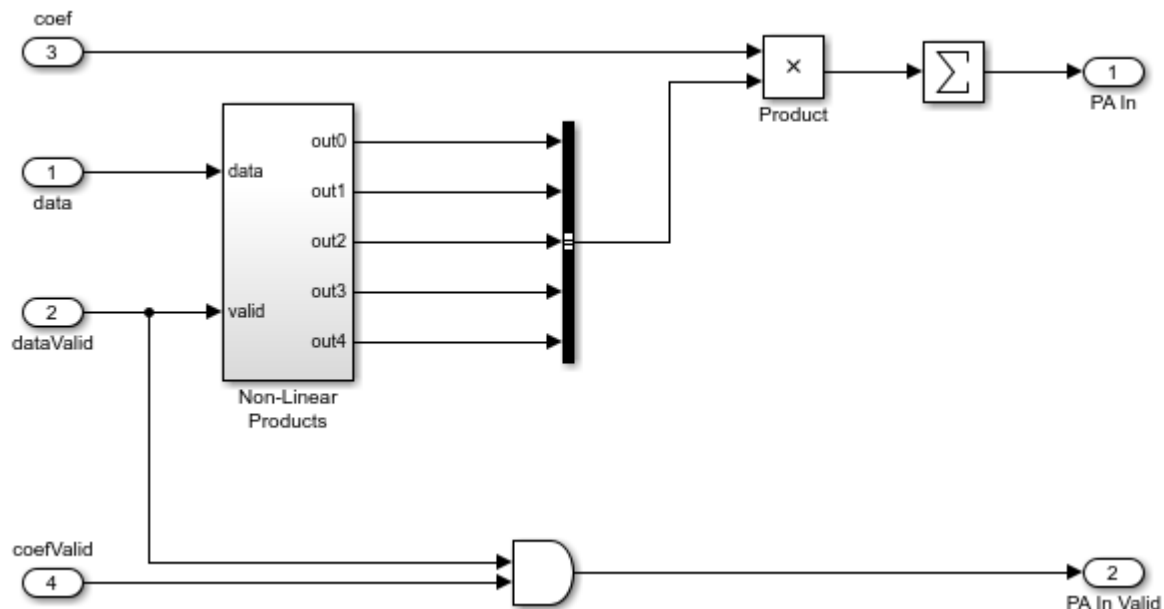
```
load_system(modelname);
open_system([modelname '/RPEMCoeffEstimation']);
```



## Digital Predistorter

The DigitalPredistorter subsystem distorts the input data using the coefficients estimated by RPEMCoeffEstimation subsystem. The DPD design in this example is based on a memory polynomial, which corrects the nonlinearities and memory effects in the PA. The estimated coefficients and the generated input data are provided as input to the DPD for applying predistortion. The input data is first placed in a shift register based on the memory depth. Second, this vector is concatenated with the nonlinear products of the data depending on the polynomial degree. This concatenation forms a vector of 25 (memory depth times degree) elements. The dot product of the obtained vector and estimated coefficients provides the predistorted input that is fed as input to PA after upsampling. Run this command to open the DigitalPredistorter subsystem.

```
load_system(modelname);
open_system([modelname '/DigitalPredistorter']);
```



## RF Blocks Configuration

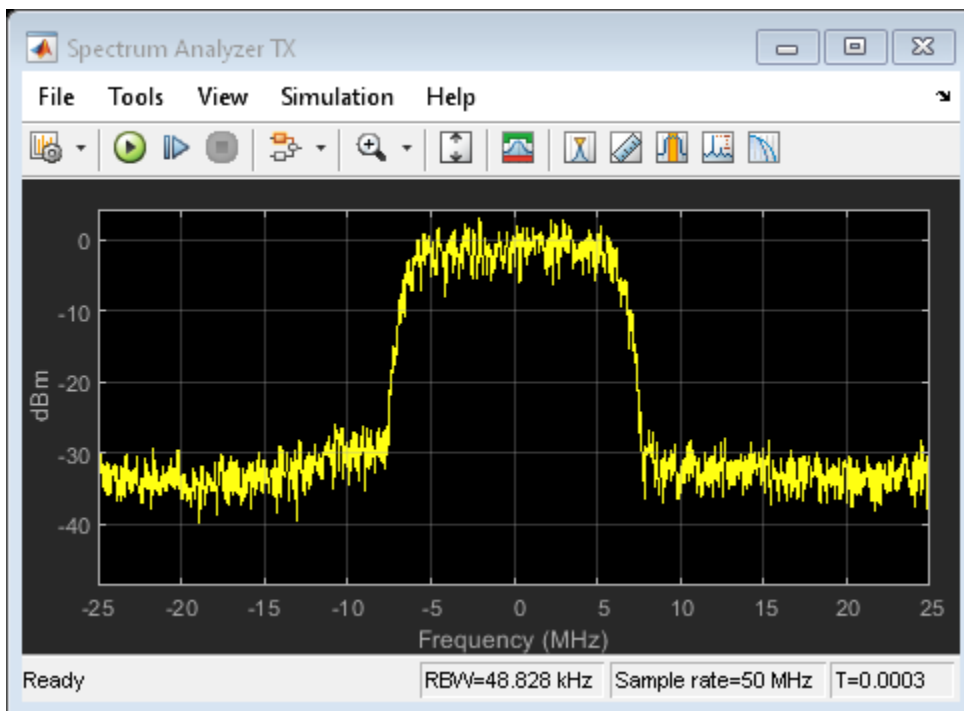
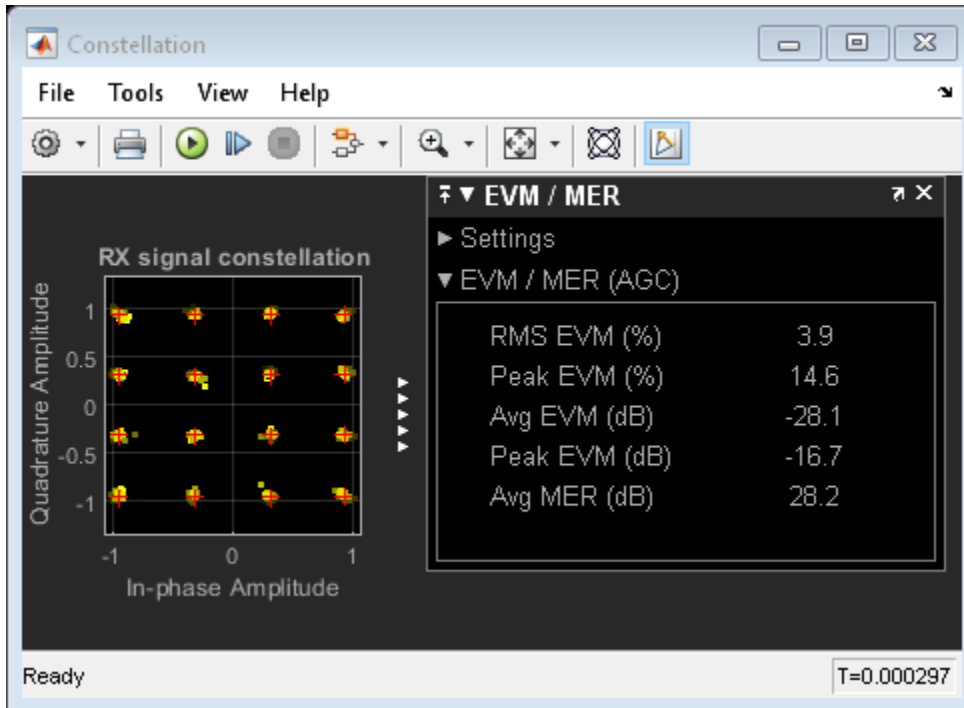
This example has a control switch to enable or disable predistortion and coefficient estimation. If you enable the switch, the example provides the output data from DigitalPredistorter subsystem as input to RF blocks. Otherwise, the example provides the output data from Baseband Signal Generation subsystem as input to RF blocks as in-phase (I), quadrature-phase (Q) samples. These I/Q samples are upsampled to 2.4 GHz and provided as input to the PA. The coefficient matrix required by the PA is preloaded based on the standard-compliant LTE signal with a sample rate of 15.36 MHz. These coefficients are stored in a MAT file, and the values are loaded while initializing the example. In the other path, the data is passed through a low noise amplifier (LNA) and is down-converted before providing it to the RPEMCoeffEstimation subsystem.

## Verification and Results

Run the model. By default, the DPD and RPEMCoeffEstimation are enabled. If you disable the DPD, the error vector magnitude (EVM) increases, and the spectral regrowth in adjacent channels

increases. The constellation and spectrum analyzer diagrams show the results of running the model with the DPD enabled.

```
sim(modelname);
```



## HDL Code Generation and Implementation Results

To check and generate HDL for this example, you must have HDL Coder™. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the **DigitalPredistorter** subsystem.

The DigitalPredistorter subsystem is synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The frequency obtained after place and route is about 220 MHz. Create a table that displays the post place and route resource utilization results for a 16-bit complex input.

```
F = table(...
    categorical({'Slice Registers'; 'Slice LUT'; 'DSP'}), ...
    categorical({'4429'; '8196'; '160'}), ...
    categorical({'218600'; '437200'; '900'}), ...
    categorical({'2.03'; '1.87'; '17.78'}), ...
    'VariableNames', ...
    {'Resources', 'Utilized', 'Available', 'Utilization (%)'});
disp(F);
```

Resources	Utilized	Available	Utilization (%)
Slice Registers	4429	218600	2.03
Slice LUT	8196	437200	1.87
DSP	160	900	17.78

## References

1. Gan, Li, and Emad Abd-Elrady. "Digital Predistortion of Memory Polynomial Systems Using Direct and Indirect Learning Architectures." In *Proceedings of the Eleventh IASTED International Conference on Signal and Image Processing (SIP)* (F. Cruz-Roldan and N. B. Smith, eds.), No. 654-802. Calgary, AB: ACTA Press, 2009.

## Encode Streaming Data Using General CRC Generator HDL Optimized Block for 5G NR Standard

This example shows how to use the General CRC Generator HDL Optimized block for encoding streaming data according to the 5G NR standard.

In this example, the output of this block is compared with the function `nrCRCEncode` (5G Toolbox). A cyclic redundancy check (CRC) is an error-detection code designed to detect errors in streaming data. A CRC generator calculates a short fixed-length binary sequence checksum and appends it with the data. A CRC detector performs a CRC on the data and compares the resulting checksum with the appended checksum. If the two checksums do not match, an error is detected. The CRC generator and detector are used in the 5G NR system to detect any errors in the transport blocks of control and uplink and downlink data channels. The 5G NR standard specifies six different cyclic generator polynomials: CRC6, CRC11, CRC16, CRC24A, CRC24B, and CRC24C. For more information about these polynomials, see TS 38.212 Section 5.1 [ 1 ].

### Generate Input Data for NR CRC Generator

Select a CRC polynomial specified in the 5G NR standard. Generate random input data of length `frameLen` and control signals that indicate the frame boundaries. The example model imports the MATLAB® workspace variables `dataIn`, `startIn`, `endIn`, `validIn`, `sampleTime`, and `simTime`.

```
CRCType = 'CRC24A'; % Specify the CRCType as 'CRC6','CRC11','CRC16','CRC24A','CRC24B' or 'CRC24C'
frameLen = 100;
msg = randi([0 1],frameLen,1);

[dataIn,ctrlIn] = whdlFramesToSamples(msg);

dataIn = timeseries(logical(dataIn));
startIn = timeseries(logical(ctrlIn(:,1)));
endIn = timeseries(logical(ctrlIn(:,2)));
validIn = timeseries(logical(ctrlIn(:,3)));

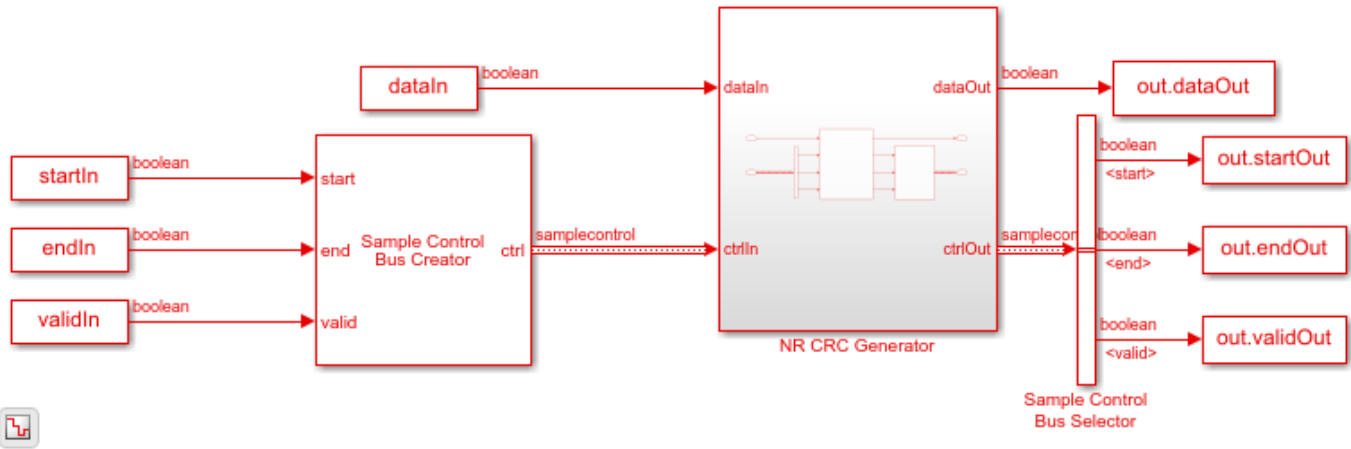
sampleTime = 1;
simTime = length(ctrlIn(:,3)) + 100;
```

### Run NR CRC Generator Model

The `nrCRCGeneratorExampleInit.m` script configures the General CRC Generator HDL Optimized block by setting the parameters of the block based on the specified CRC generator polynomial, `CRCType`. This script also provides input to the reference function `nrCRCEncode` (5G Toolbox). The NR CRC Generator subsystem contains the General CRC Generator HDL Optimized block. Running the model imports the input signal variables from the workspace and returns the CRC-encoded output and control signals that indicate the frame boundaries. The model exports variables `encOut` and `ctrlOut` to the MATLAB® workspace.

```
[poly,crcPolynomial,initState,finalXORValue] = nrCRCGeneratorExampleInit(CRCType);
open_system('NRCRCGeneratorHDL');
encOut = sim('NRCRCGeneratorHDL');
```





## Verify NR CRC Generator Results

Convert the streaming data output of the NR CRC Generator subsystem to frames. Compare those frames with the output of the nrCRCEncode function.

```
startIdx = find(encOut.startOut);
endIdx = find(encOut.endOut);
dataOut = encOut.dataOut;

dataRef = nrCRCEncode(msg,poly);
bitErr = sum(abs(dataRef - dataOut(startIdx:endIdx)));
fprintf('CRC-encoded frame: Behavioral and HDL simulation differ by %d bits\n',bitErr);

close_system('NRCRCGeneratorHDL');

CRC-encoded frame: Behavioral and HDL simulation differ by 0 bits
```

## References

- 1 3GPP TS 38.212. NR ; Multiplexing and Channel Coding. 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.

## See Also

### Blocks

General CRC Generator HDL Optimized

### Functions

nrCRCEncode



# Reference Applications

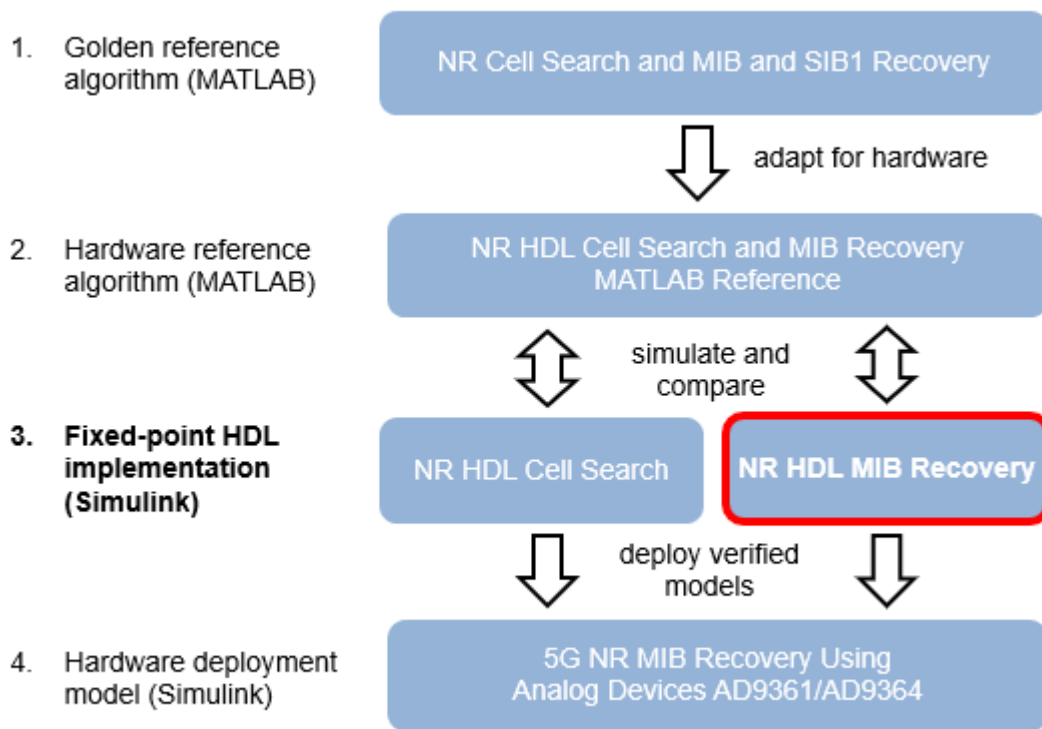
---

## NR HDL MIB Recovery

This example shows the design of a 5G NR synchronization signal block (SSB) decoding and master information block (MIB) recovery model optimized for HDL code generation and hardware implementation.

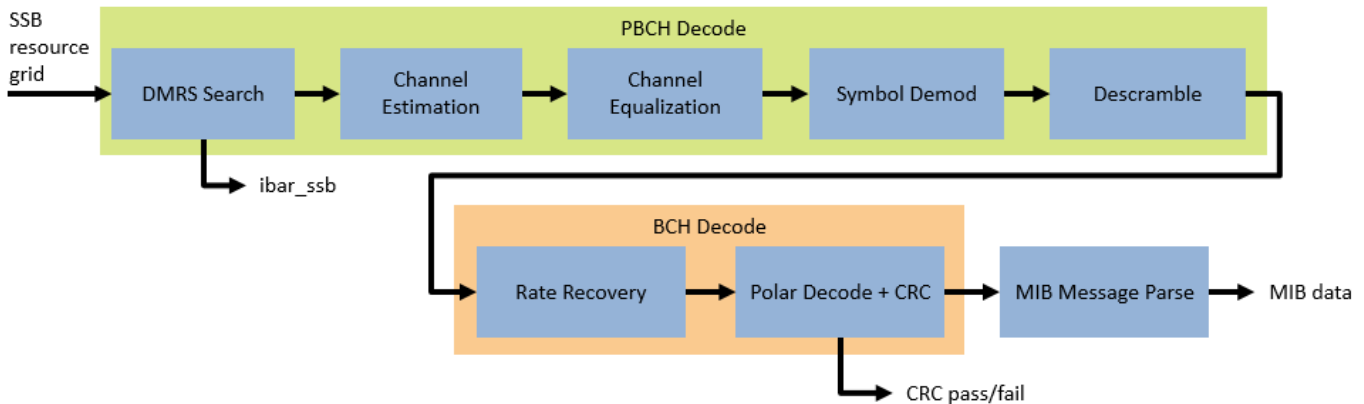
### Introduction

The Simulink models described in this example are HDL optimized implementations of SSB decoding and MIB recovery for 5G. This example is one of a related set which show the workflow for designing and deploying a 5G NR cell search and MIB recovery algorithm to hardware. The figure shows the complete set of examples and the current example within the workflow. For more details on the overall algorithm and workflow, see the “NR HDL Cell Search and MIB Recovery MATLAB Reference” on page 5-14 example. The “5G NR MIB Recovery Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example shows how to deploy the algorithm to an SoC. For a general description of how MATLAB and Simulink can be used together to develop deployable models, see “Wireless Communications Design for FPGAs and ASICs”.



MIB recovery requires SSB detection, demodulation and SSB decoding. This example focuses on the SSB decoding aspect of MIB recovery, and SSB detection and decoding are described in “NR HDL Cell Search” on page 5-30. The SSB decoding described in this example is designed to be used with either MATLAB or Simulink SSB detection implementations. This example introduces the SSB decoding model, using MATLAB reference to generate the input to the SSB decoding model and to verify the behavior of the model. The example then shows a Simulink model integrating the SSB detection and SSB decoding parts of the receiver to recover MIB from a baseband waveform.

After an SSB has been detected and demodulated, the SSB needs to be decoded to extract the MIB contents. SSB decoding requires demodulation reference signal (DMRS) search, channel estimation and phase equalization, and broadcast channel (BCH) decoding steps as shown in the figure below.



## File Structure

This example uses these files.

### Simulink models

- `nrhd\SSBDecoding`: This Simulink model uses the `nrhd\SSBDecodingCore` model reference to simulate the behaviour of the SSB decoding part of the MIB recovery process.
- `nrhd\MIBRecovery`: This Simulink model combines the processing of the SSB detector and the SSB decoder into an integrated model illustrating the complete MIB recovery process. This model uses the `nrhd\SSBDetectionCore` and `nrhd\SSBDecodingCore` model references.
- `nrhd\SSBDecodingCore`: This model reference implements the SSB decoding algorithm.
- `nrhd\SSBDetectionCore`: This model reference implements the SSB detection algorithm.

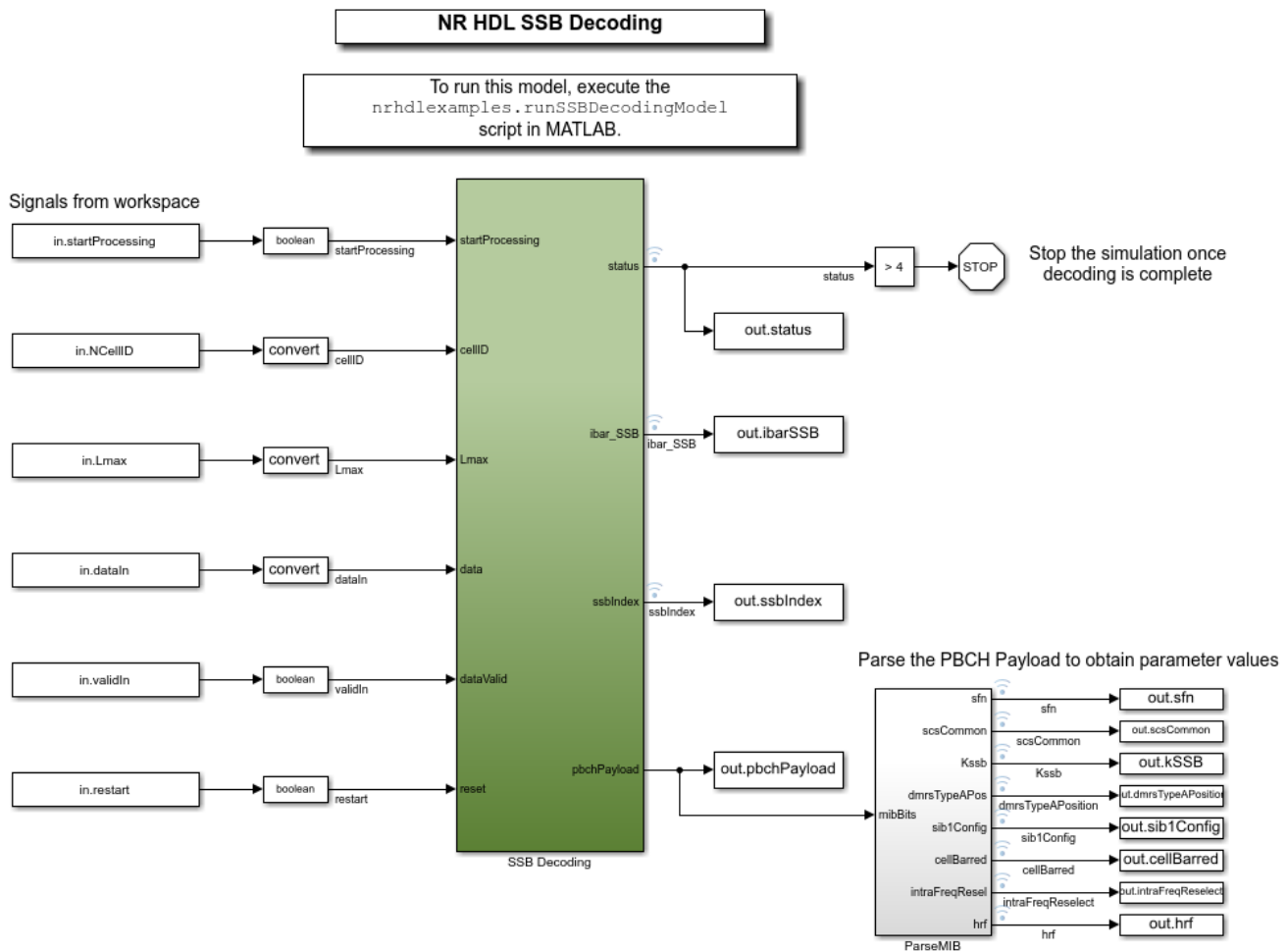
### MATLAB functions and scripts

- `nrhd\examples.runSSBDecodingModel`: This script uses the MATLAB reference to implement the cell search and SSB detection algorithms, then runs the `nrhd\SSBDecoding` Simulink model by calling the `nrhd\examples.ssbDecodeSimulink` function. The model is verified using 5G Toolbox functions and the MATLAB reference.
- `nrhd\examples.runMIBRecoveryModel`: This script used the MATLAB reference to implement the cell search algorithm, then runs the `nrhd\MIBRecovery` Simulink model. The script verifies the operation of the model using 5G toolbox and the MATLAB reference code.
- `nrhd\examples.ssbDecodeSimulink`: This function runs the `nrhd\SSBDecoding` Simulink model to decode the SSB. It has the same input and output arguments as the `nrhd\examples.ssbDecode` function from the MATLAB reference.

This example also uses a number of helper functions from the `nrhd\examples` package. The Simulink models and the `nrhd\examples` package are on the MATLAB path. To open one of the models, enter its name at the MATLAB command prompt. To open a function or script from the `nrhd\examples` package, use the `edit` command.

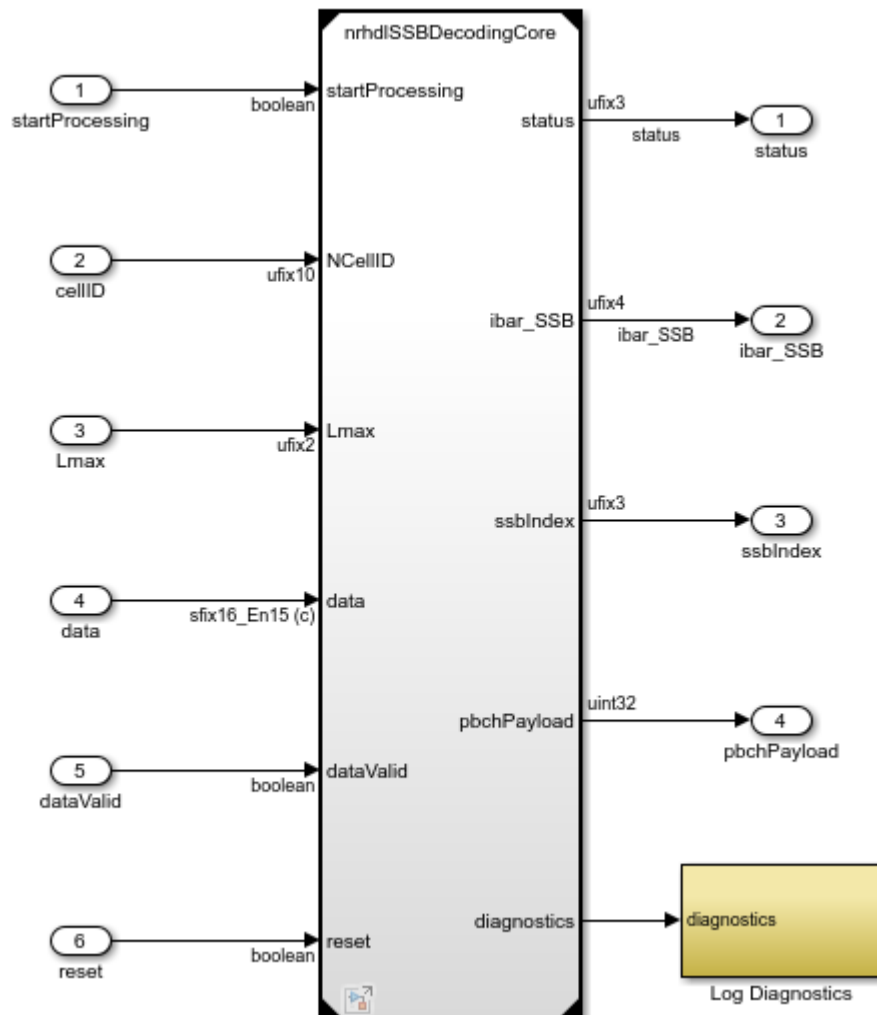
## NR HDL SSB Decoding

This figure shows the `nrhdlSSBDecoding` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the SSB Decoding subsystem, and writes the outputs back to the workspace. The ParseMIB subsystem takes the `pbchPayload` and interprets the bit fields to produce the MIB parameter outputs. To run the model, call the `nrhdlexamples.runSSBDecodingModel` script in MATLAB.



## SSB Decoding Interface

The SSB Decoding subsystem contains an instance of the `nrhdlSSBDecodingCore` model reference. This section describes the inputs and outputs of that model.



### Inputs

- *data*: 16-bit signed complex-valued signal carrying the 4 OFDM symbols of the SSB.
- *dataValid*: 1-bit control signal to validate data.
- *NCellID*: 10-bit unsigned number which provides cell ID number for the detected SSB.
- *startProcessing*: 1-bit control signal which indicates when all data has been written and that cellID and Lmax are valid.
- *Lmax*: 2-bit unsigned number which indicates the maximum number of SSBs in a burst. A value of 0 indicates 4 SSBs and a value of 1 indicates 8 SSBs.
- *Reset*: 1-bit control signal to reset the processing.

### Outputs

- *Status*: 3-bit unsigned value indicating the progress of the current operation. See below for more information on the possible values of this signal.
- *Ibar\_SSB*: 3-bit unsigned value that is the index calculated by the DMRS search process.

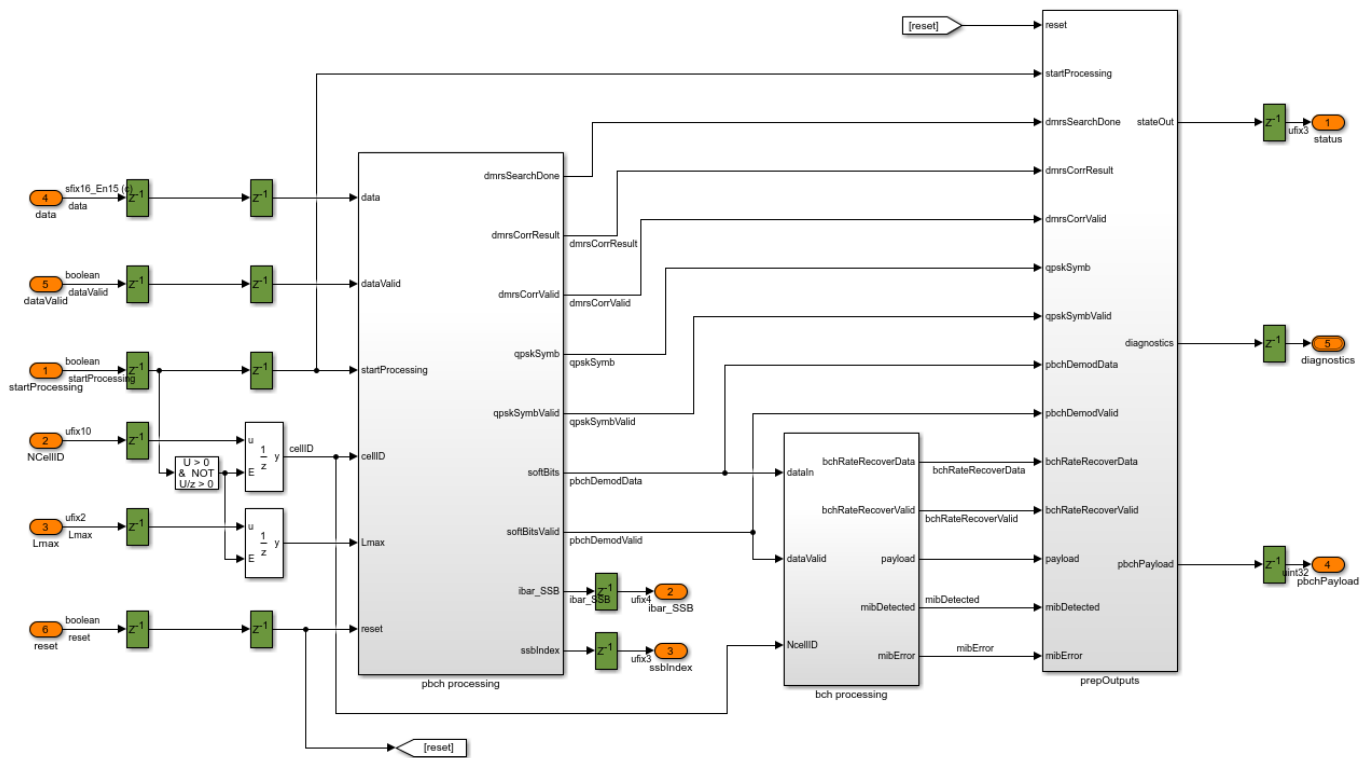
- *ssbIndex*: 3-bit unsigned value that is the index of the SSB, calculated using *ibar\_ssb* and *Lmax*.
- *pbchPayload*: 32-bit unsigned value that contains the MIB and additional PBCH timing data.
- *Diagnostics*: Bus containing diagnostic signals.

Status Signal States

- 0: idle
- 1: performing DMRS search
- 2: performing PBCH decoding
- 3: performing rate recovery
- 4: performing polar decoding
- 5: CRC error (end state)
- 6: CRC pass, MIB detected (end state)

SSB Decode Model Reference Structure

This diagram shows the top level of the `nrhdLSSBDecodingCore` model. The input data is 4 OFDM symbols for the synchronization signal block (SSB), with the data scaled within the range +/-1. The model starts processing when all of the SSB data has been input to the model and `startProcessing` is asserted. The `startProcessing` signal indicates when all of the SSB data has been written, the `NCellID` and `Lmax` inputs are valid, and processing of the SSB can begin.



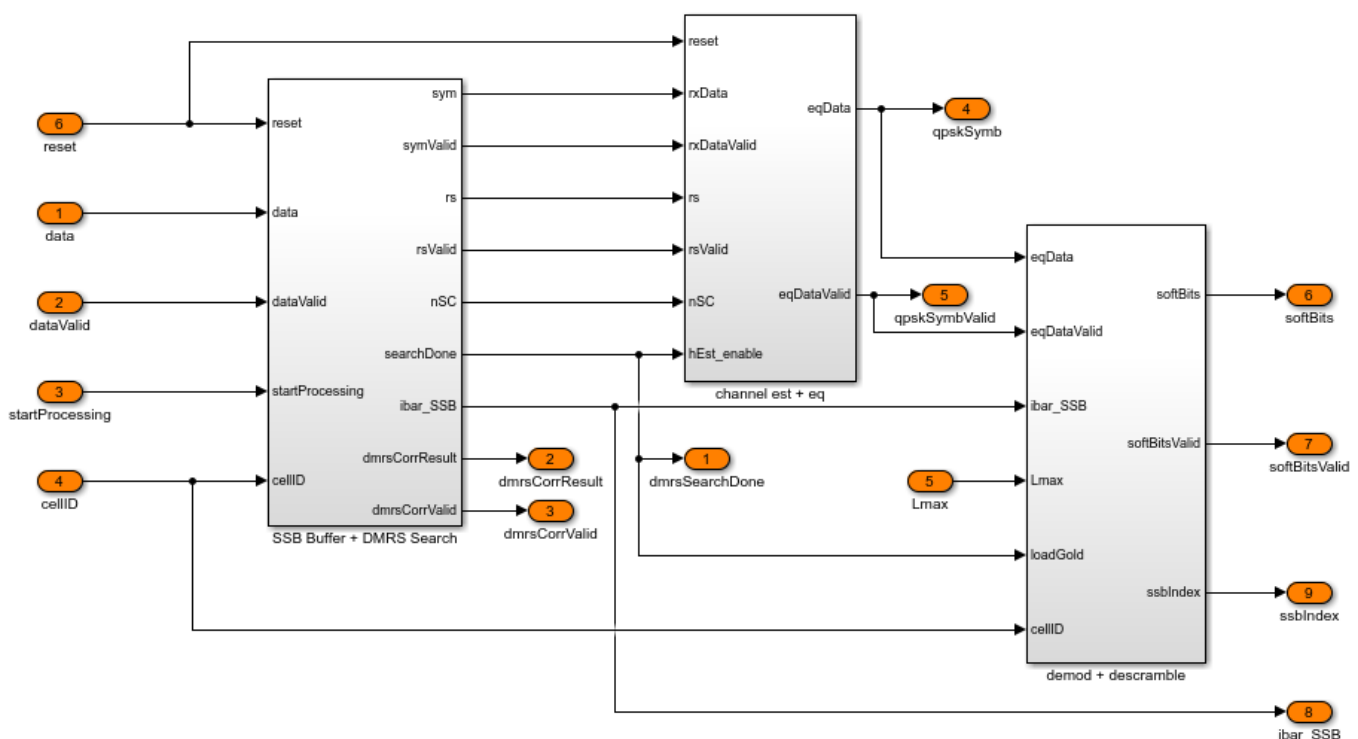
The `pbch processing` subsystem performs DMRS search, channel estimation and equalization, QPSK symbol demodulation, and descrambling. The output from the `pbch processing` subsystem is passed to the `bch processing` subsystem which performs rate recovery, polar decoding, and CRC decoding. The



prepOutputs subsystem uses the control signal to produce the status output, and creates the diagnostics bus using signals from intermediate points in the processing.

### PBCH Processing Subsystem

The pbch processing subsystem performs DMRS search, channel estimation and equalization, and QPSK demodulation and descrambling. Incoming data is stored in a RAM buffer where it is held until startProcessing is asserted, indicating that all required information is available to start the DMRS search process. The DMRS search reads the DMRS symbols from the RAM and correlates with the 8 possible DMRS sequences, selecting the strongest correlation value to determine ibar\_SSB. Once the DMRS search has been completed ibar\_SSB is used to generate the reference DMRS required for channel estimation. The reference DMRS passed to the channel est + eq subsystem along with the received PBCH symbols and associated DMRS.

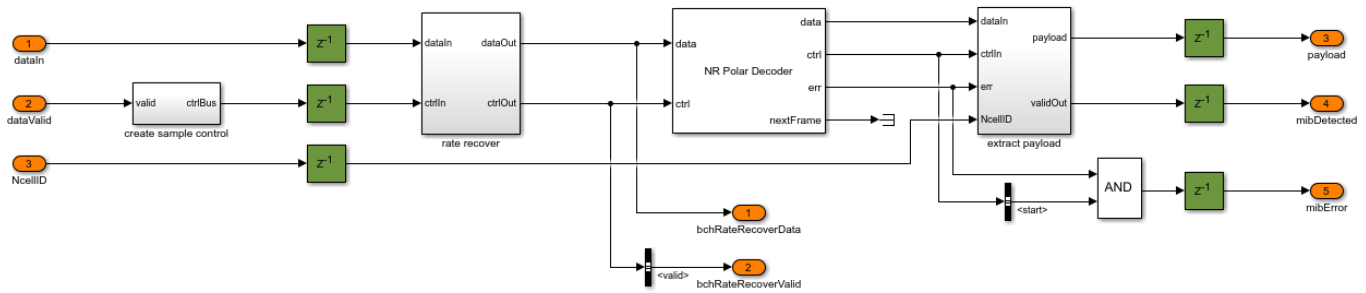


The channel est + eq subsystem performs channel estimation using the received data and the reference DMRS. The channel estimate performs linear interpolation between DMRS locations within an OFDM symbol, but does not perform averaging between symbols in case of any residual carrier frequency offset. Phase equalization of the PBCH symbols is then performed, before QPSK demodulation and descrambling are performed, using ibar\_SSB and Lmax to calculate the descrambling sequence.

### BCH Processing Subsystem

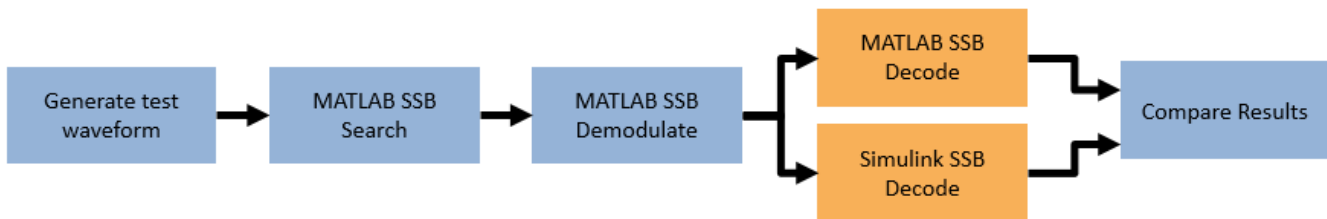
The BCH processing performs rate recovery, polar decoding and CRC decoding of the BCH. The rate recovery subsystem includes signal scaling and wordlength reduction to prepare the data for polar decoding. The scaled, rate recovered, soft bits are then passed to the NR Polar Decoder block, which also performs CRC decoding. The err output port from the NR Polar Decoder block indicates if

decoding was successful or encountered any errors. The extract payload subsystem performs descrambling and deinterleaving of the payload bits.



### SSB Decode Simulation Setup

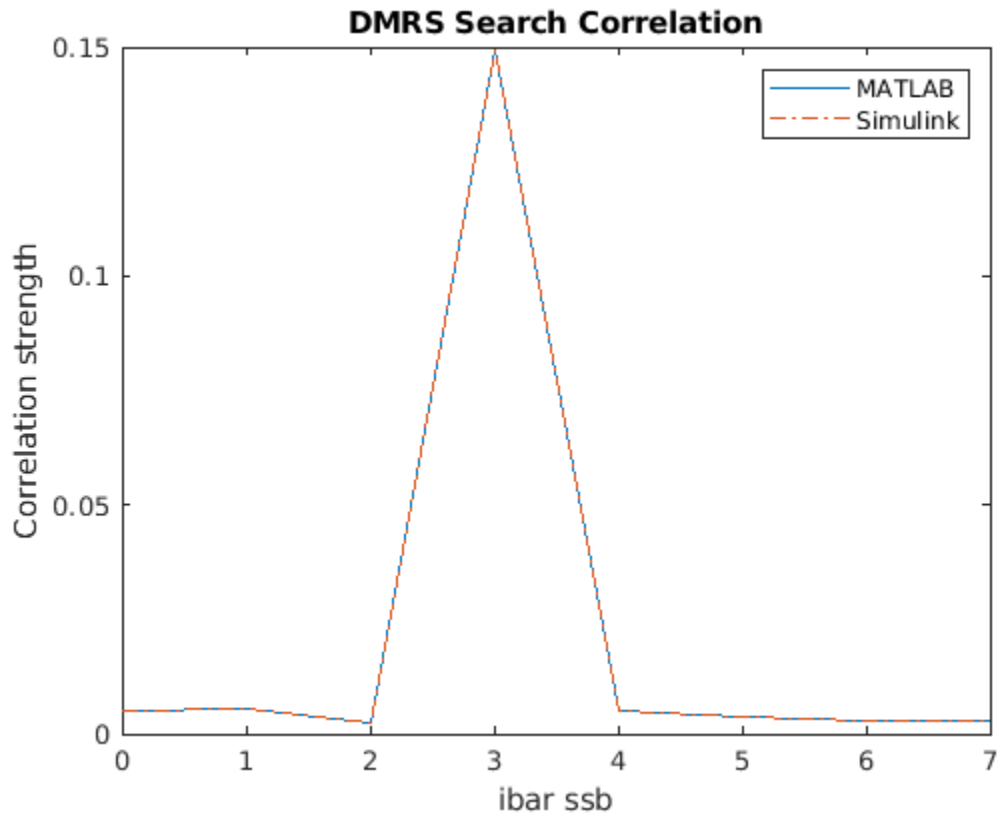
This block diagram shows the simulation setup implemented by the `nrdhlexamples.runSSBDecodingModel` script. The script uses 5G Toolbox functions to generate a test waveform. The test waveform is then processed using the MATLAB reference code for the SSB detector to search for, then demodulate, the strongest SSB in the waveform. This provides the SSB data input for the SSB decoding stage. The SSB data is passed to both MATLAB and Simulink implementations, and the outputs are compared to verify the operation of the Simulink model.



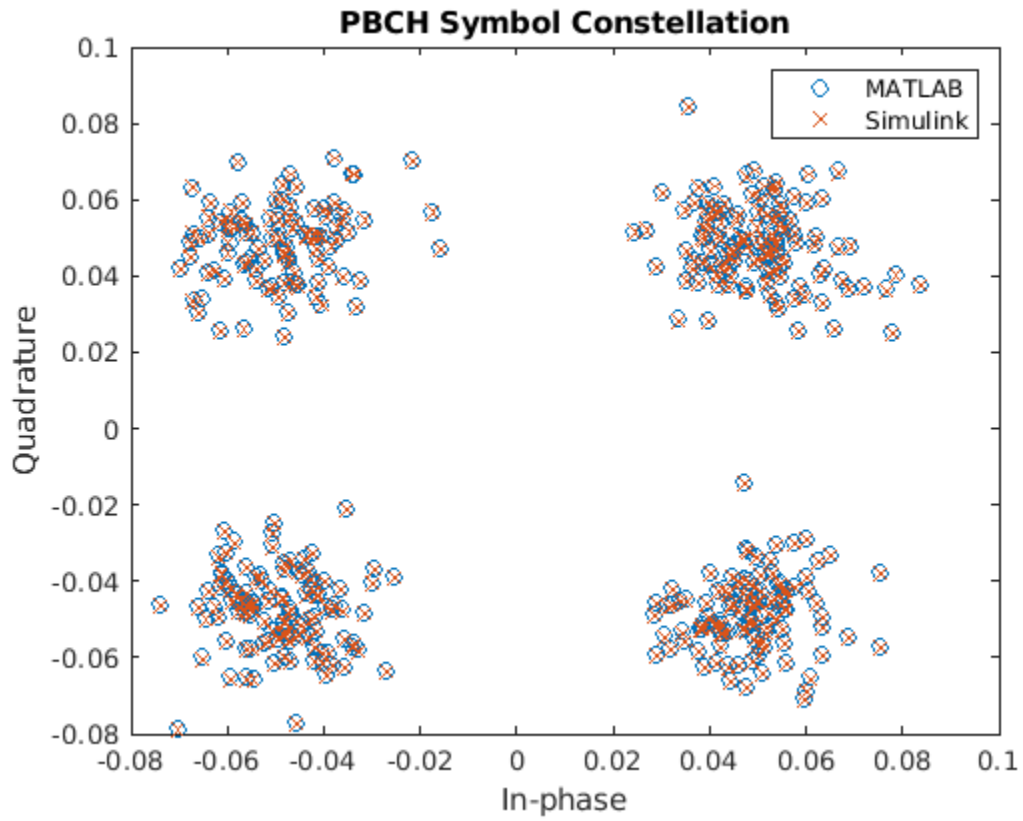
### SSB Decode Simulation Results

Call the `nrdhlexamples.runSSBDecodingModel` function to perform the SSB decode simulation setup as described above. This function calls the MATLAB reference code followed by the Simulink model using the functions `nrdhlexamples.SSBDecoding` and `nrdhlexamples.SSBDecodingSimulink`.

The signals from the diagnostics bus can be used to compare and verify intermediate signals from the Simulink simulation with the MATLAB equivalent. The plot of the correlation strengths from the DMRS search process is shown below, with the MATLAB and Simulink signals producing the same result.



The equalized QPSK symbols representing the PBCH are also plotted by the script, and the output is shown below. This plot shows that the Simulink model matches the MATLAB reference.



The script also displays the final result of the decoding process in the command prompt, with both the simulation and MATLAB reference results shown for comparison.

```
MATLAB decoded information
  pbchPayload: 218103952
    ssbIndex: 3
      hrf: 0
      err: 0
      mib: [1x1 struct]
```

```
Simulink decoded information
  pbchPayload: 218103952
    ssbIndex: 3
      hrf: 0
      err: 0
      mib: [1x1 struct]
```

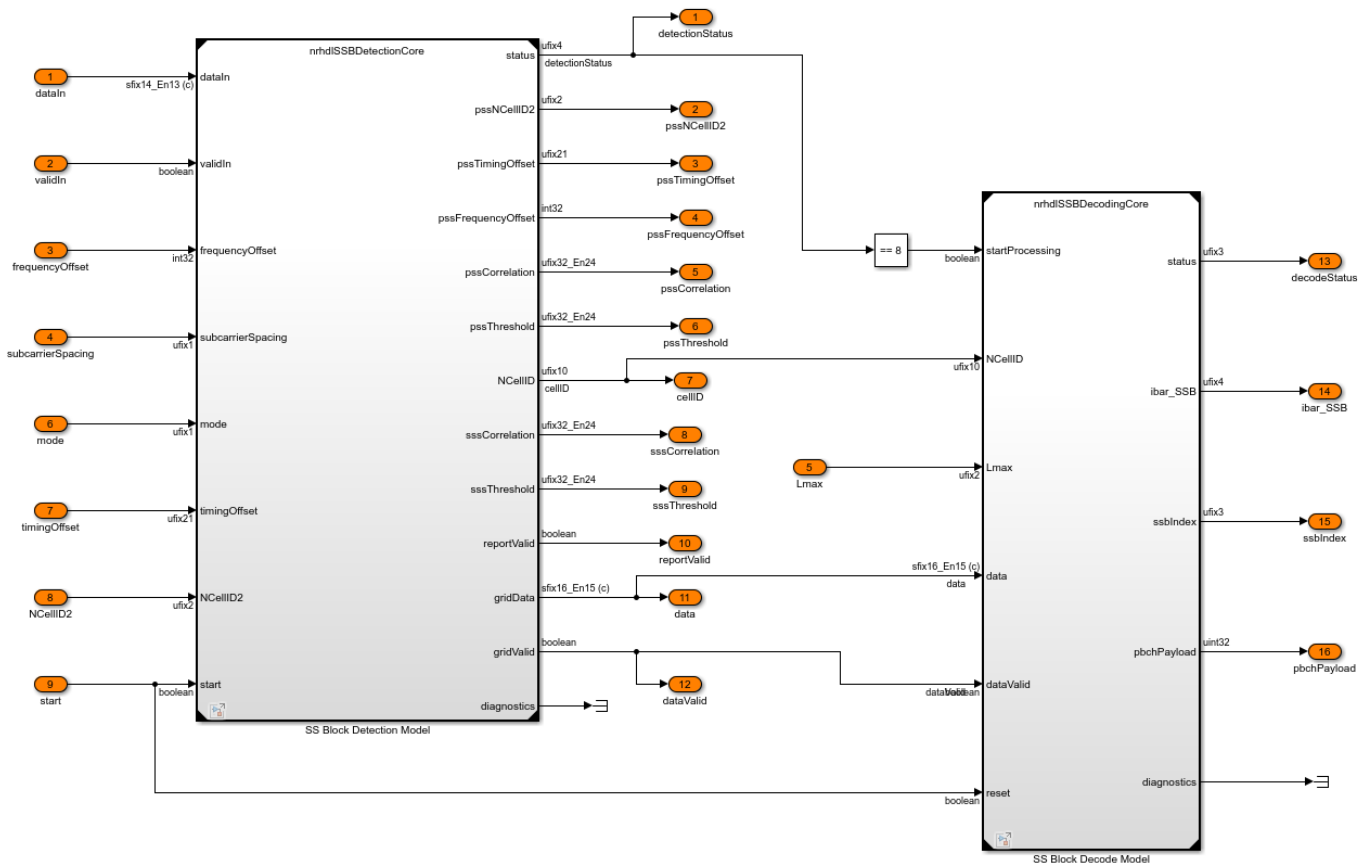
```
MATLAB decoded MIB parameters
      NFrame: 105
  SubcarrierSpacingCommon: 30
      k_SSB: 0
  DMRSTypeAPosition: 2
  PDCCHConfigSIB1: 0
      CellBarred: 0
  IntraFreqReselection: 0
```

```
Simulink decoded MIB parameters
      NFrame: 105
  SubcarrierSpacingCommon: 30
      k_SSB: 0
  DMRSTypeAPosition: 2
  PDCCHConfigSIB1: 0
      CellBarred: 0
  IntraFreqReselection: 0
```

## MIB Recovery Model

The `nrhdlMIBRecovery` model connects the two reference models for SSB Decoding and SSB detection (`nrhdlSSBDecodingCore` and `nrhdlSSBDetectionCore`) to create a complete MIB recovery implementation. This model can be used to recover MIB from baseband 5G waveforms. The script `nrhdlexamples.runMIBRecoveryModel` can be used to run this model and compare against the MATLAB reference. To reduce the processing time required the cell search part of the algorithm is performed in MATLAB then, once the strongest SSB has been determined, the Simulink model is used to re-acquire, demodulate, and decode the SSB.

The status signal from the detector is used to start the SSB decoder when it has reached state 8, indicating that demodulation is complete, SSS has been found and the demodulated grid has been output. When the SSB decoder has the demodulated grid and received the `startProcessing` signal it will decode the SSB, outputting the PBCH payload which is then parsed to extract the MIB data.



### HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for `nrhdSSBDecoding/SSB Decoding` or `nrhdMIBRecovery/MIB Recovery` subsystems. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 150 MHz.

Resource utilization for `nrhdSSBDecoding` model:

Resource	Usage
Slice Registers	8896
Slice LUTs	11229
RAMB18	4
RAMB36	5
DSP48	38

Resource utilization for `nrhdMIBRecovery` model:

Resource	Usage
----------	-------

Slice Registers	87760
Slice LUTs	47747
RAMB18	14
RAMB36	5
DSP48	247

## See Also

## Related Examples

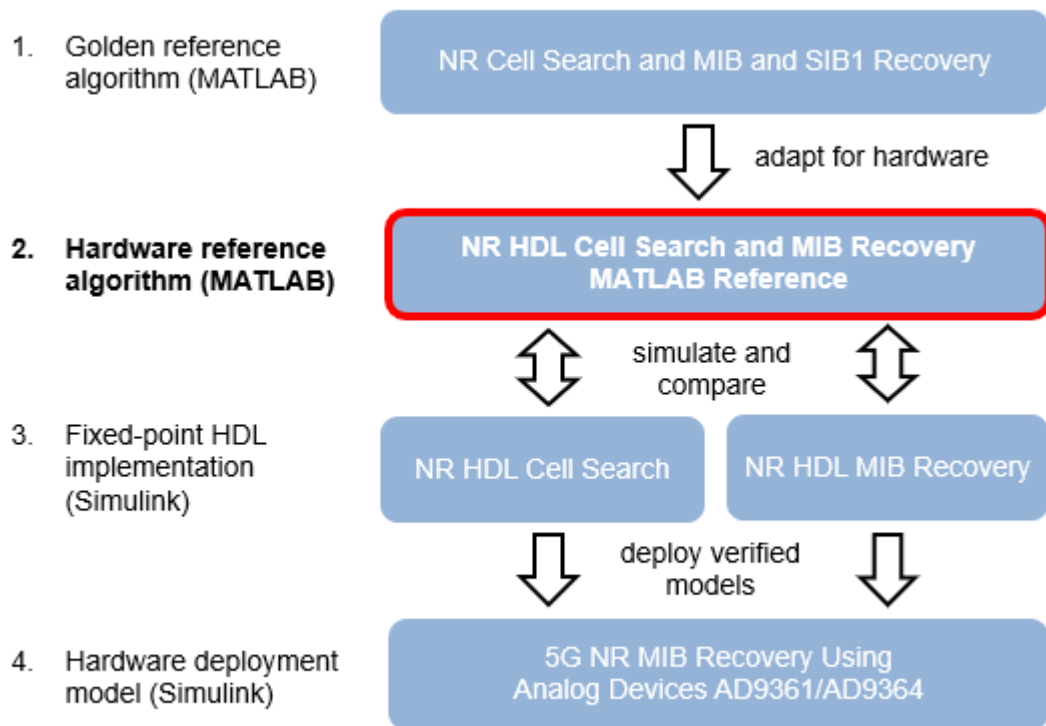
- “NR HDL Cell Search” on page 5-30

## NR HDL Cell Search and MIB Recovery MATLAB Reference

This example shows how to model a 5G NR cell search and MIB recovery hardware algorithm in MATLAB as a step towards developing a Simulink HDL implementation. Use this MATLAB reference to verify the Simulink models in the “NR HDL Cell Search” on page 5-30 and “NR HDL MIB Recovery” on page 5-2 examples.

### Introduction

The NR HDL Cell Search and MIB Recovery MATLAB Reference example bridges the gap between a mathematical algorithm and its hardware implementation by providing a MATLAB model of the algorithms that are implemented in hardware. The MATLAB reference is created to evaluate hardware-friendly algorithms and generate test vectors for verifying the Simulink HDL implementation. These related examples show the workflow for designing and deploying a 5G cell search and MIB recovery algorithm to hardware.



- “NR Cell Search and MIB and SIB1 Recovery” (5G Toolbox): MATLAB golden reference of the floating-point algorithm.
- NR HDL Cell Search and MIB Recovery MATLAB Reference (this example): MATLAB hardware reference that models hardware-friendly algorithms and generates test waveforms. This MATLAB code operates on vectors and matrices of floating-point data samples and does not support HDL code generation.
- “NR HDL Cell Search” on page 5-30: Simulink model of the 5G cell search subsystem that uses the same algorithm as the MATLAB reference. This model operates on scalar fixed-point data and is optimized for HDL code generation.

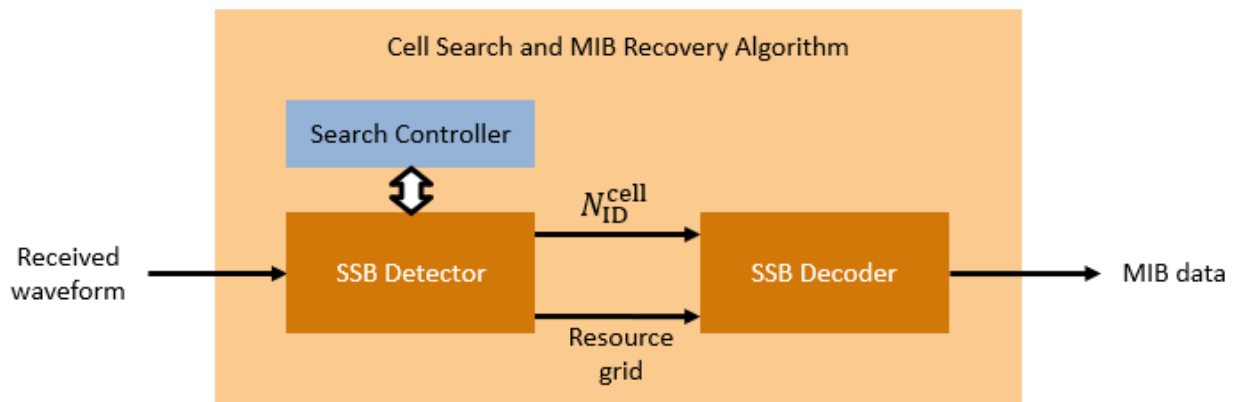


- “NR HDL MIB Recovery” on page 5-2: Adds MIB recovery to the cell search model using the same algorithm as the MATLAB reference. This model operates on scalar fixed-point data and is optimized for HDL code generation.
- “5G NR MIB Recovery Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Simulink model that is configured for deployment to an SoC.

For a general description of how MATLAB and Simulink can be used together to develop deployable models, see “Wireless Communications Design for FPGAs and ASICs”.

### Cell Search and MIB Recovery Overview

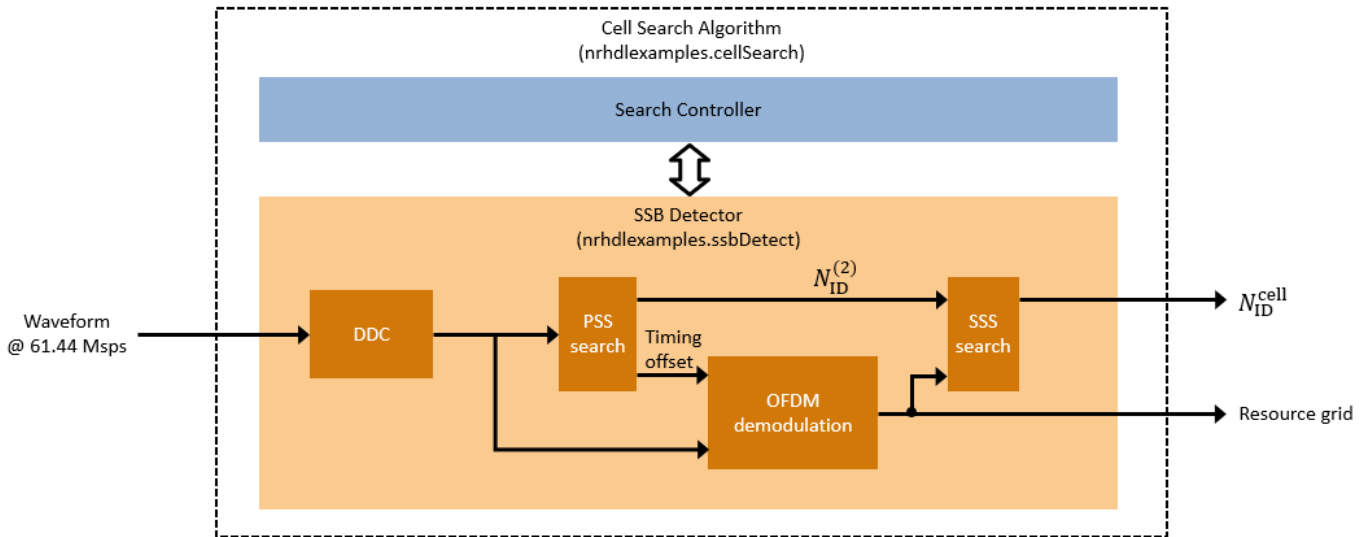
A block diagram of the cell search and MIB recovery algorithm is shown. The algorithm detects, demodulates and decodes 5G NR synchronization signal blocks (SSBs) and is a hardware-friendly version of the corresponding steps in the “NR Cell Search and MIB and SIB1 Recovery” (5G Toolbox) example. At the top level, the algorithm consists of a Search Controller, an SSB Detector and an SSB Decoder. This example explains each of these blocks in more detail and demonstrates the corresponding MATLAB reference functions, which are used to explore algorithms for hardware implementation and to verify the streaming fixed-point Simulink models.



### Cell Search

Cell search consists of carrier frequency recovery, Primary Synchronization Signal (PSS) search, OFDM demodulation and Secondary Synchronization Signal (SSS) search. The Search Controller and the SSB Detector work together to perform these processing steps. The SSB Detector performs all of the high-speed signal processing tasks, making it well suited for FPGA or ASIC implementation. The Search Controller coordinates the search and operates at a low rate, making it well suited for software implementation on an embedded processor.

The algorithm starts by using the PSS to search for SSBs with subcarrier spacings of 15 kHz and 30 kHz across a range of coarse frequency offsets. The subcarrier spacing and coarse frequency offset search ranges are configurable. If SSBs are detected, the receiver OFDM demodulates the resource grid of the SSB with the strongest PSS and determines its cell ID using the SSS. The residual fine frequency offset is corrected during the OFDM demodulation phase.



- *SSB Detector*: Searches for and OFDM-demodulates SSBs at a given carrier frequency offset and subcarrier spacing and measures the residual fine carrier frequency offset.
- *Digital Down Converter (DDC)*: Performs frequency translation to correct frequency offsets in the received waveform and then decimates the signal from 61.44 Msps to 7.68 Msps.
- *PSS search*: Searches for PSS symbols within the waveform.
- *OFDM demodulation*: OFDM-demodulates an SSB resource grid.
- *SSS search*: Searches for SSS and determines the overall cell ID.
- *Search Controller*: Coordinates the cell search by directing the SSB Detector to search for PSS symbols at different coarse frequency offsets and subcarrier spacings and to demodulate the SSB with the strongest PSS.

In the MATLAB reference, the `nrhdlexamples.cellSearch` function implements the cell search algorithm. This function implements the Search Controller shown in the diagram, and calls the `nrhdlexamples.ssbDetect` function, which implements the SSB Detector. The “NR HDL Cell Search” on page 5-30 example shows the streaming fixed-point Simulink HDL implementation of the SSB Detector. In the “5G NR MIB Recovery Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example, the SSB Detector is implemented in programmable logic while the Search Controller is implemented in software on the integrated processing system.

### Search Controller

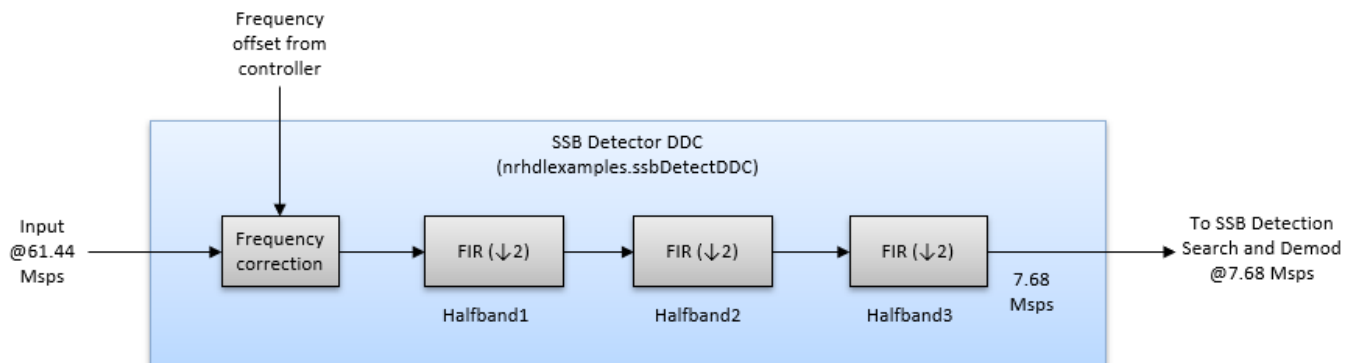
The Search Controller is responsible for coordinating the overall search. The algorithm follows these steps.

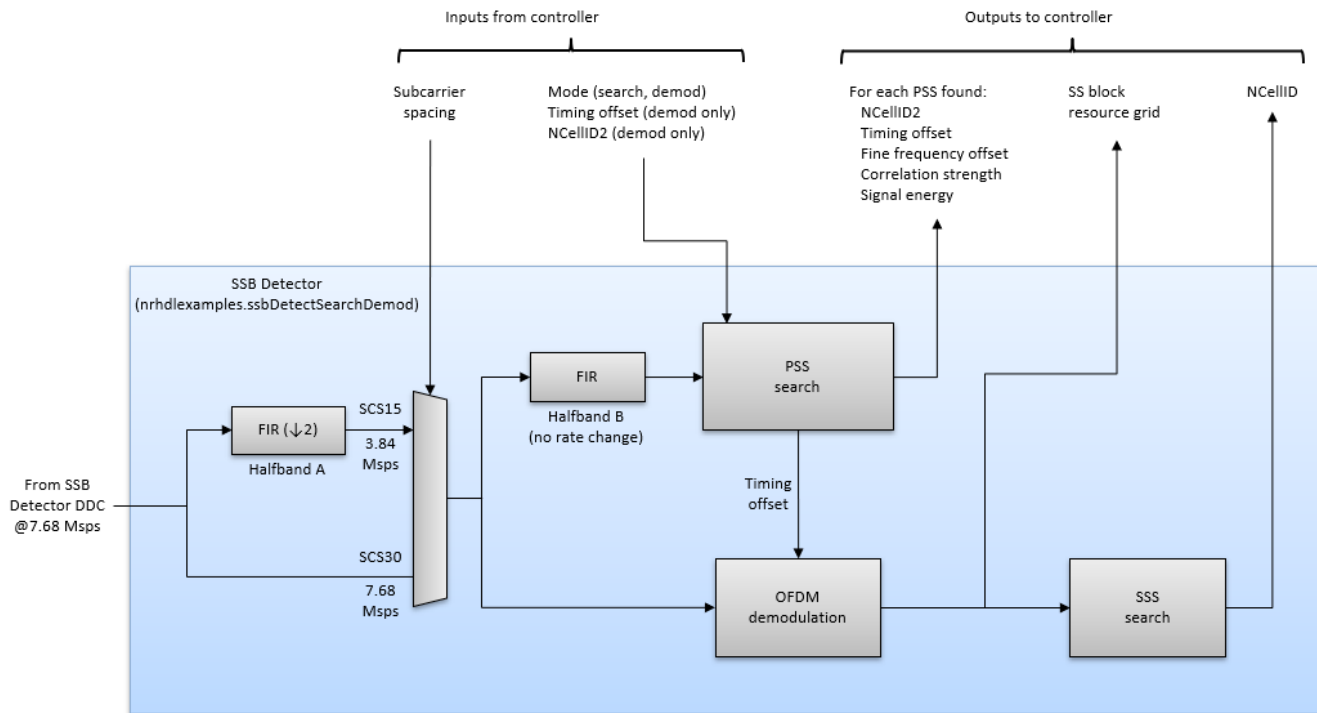
- 1 For each subcarrier spacing, step through each coarse frequency offset and use the SSB Detector to search for SSBs until one or more is detected. The coarse frequency offset step size is half the subcarrier spacing. When SSBs are detected at a given frequency, record the residual fine carrier frequency offset of the strongest SSB that is returned.

- 2 Move to the next coarse frequency step and search for SSBs again. If the search detects SSBs, choose the coarse frequency offset that resulted in the smallest fine frequency offset measurement. Otherwise, pick the last coarse frequency offset.
- 3 Compute the total frequency offset by adding the coarse and fine frequency offsets together.
- 4 Use the SSB Detector to correct the frequency offset and perform one more search for SSBs.
- 5 Pick the SSB with the strongest PSS correlation. Use the SSB Detector in demodulation mode to find and demodulate the SSB and determine its cell ID.

### SSB Detector

These diagrams show the SSB Detector structure and the parameters and data passed to and from the Search Controller. The SSB Detector is subdivided into two functions: SSB Detector DDC (`nrhdlexamples.ssbDetectDDC`) and SSB Detection Search and Demod (`nrhdlexamples.ssbDetectSearchDemod`). The DDC accepts samples at 61.44 Msps and performs a frequency shift followed by decimation by a factor of 8 using halfband filters. The frequency offset, in Hz, is provided by the search controller and is used by the algorithm to compensate for both coarse and fine frequency offsets.





SSB Detection Search and Demod accepts samples at 7.68 Msps. For 30 kHz subcarrier spacing, it uses the samples at this rate. For 15 kHz subcarrier spacing, it decimates the input by a factor of two, operating at 3.84 Msps. SSB Detection Search and Demod has two modes of operation: search and demodulation.

In search mode, the function searches for SSBs at the specified subcarrier spacing using the PSS, and returns a list of those detected. For each SSB that is found, the function returns these parameters:

- *NCellID2*: Indicates which of the three possible PSS sequences (0,1, or 2) was detected.
- *timing offset*: The timing offset from the start of the waveform to the start of the SSB.
- *fine frequency offset*: The residual fine frequency offset in Hz measured by using the cyclic prefixes of all four OFDM symbols in the SSB.
- *correlation strength*: The measured PSS correlation level.
- *signal energy*: The total energy in the samples in which the PSS was detected.

In demodulation mode, the function attempts to find a specific SSB by using its timing offset and NCellID2. If the function finds the specified PSS, the receiver OFDM demodulates the SSB resource grid and attempts to detect its SSS. In demodulation mode, the function returns these results.

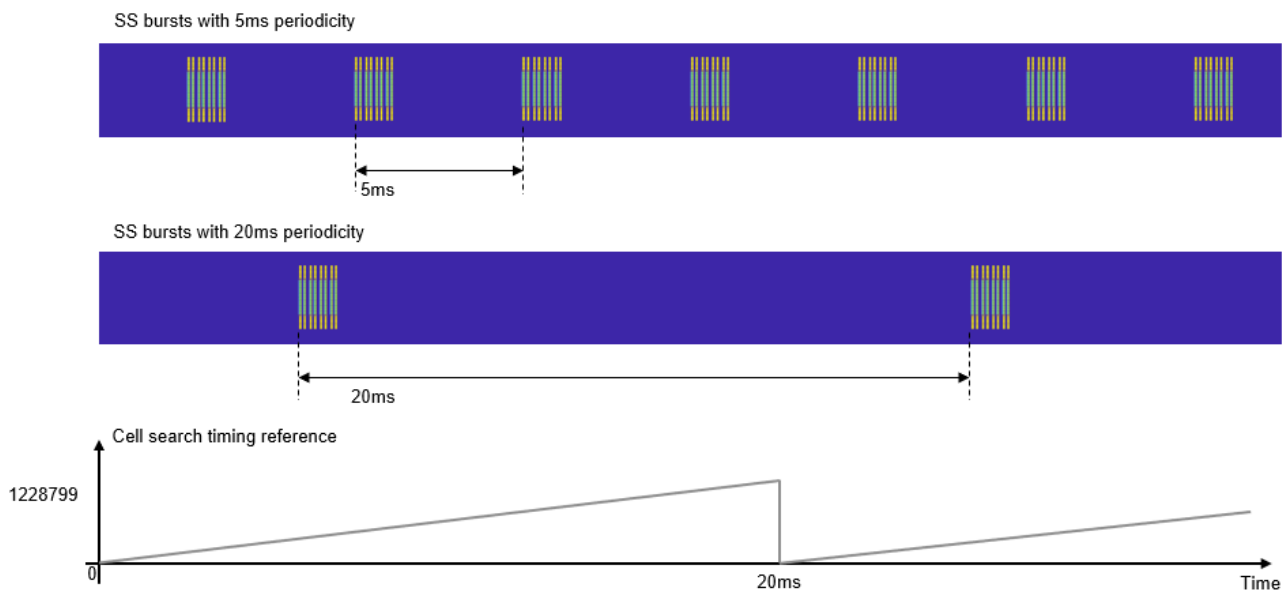
- Updated parameters for only the specified SSB if the PSS is found.
- The demodulated SSB resource grid if the PSS is found.
- The cell ID if the SSS is found.

The OFDM demodulator uses a 256-point FFT to demodulate the SSB resource grid, which contains 240 active subcarriers.

## Timing Offsets

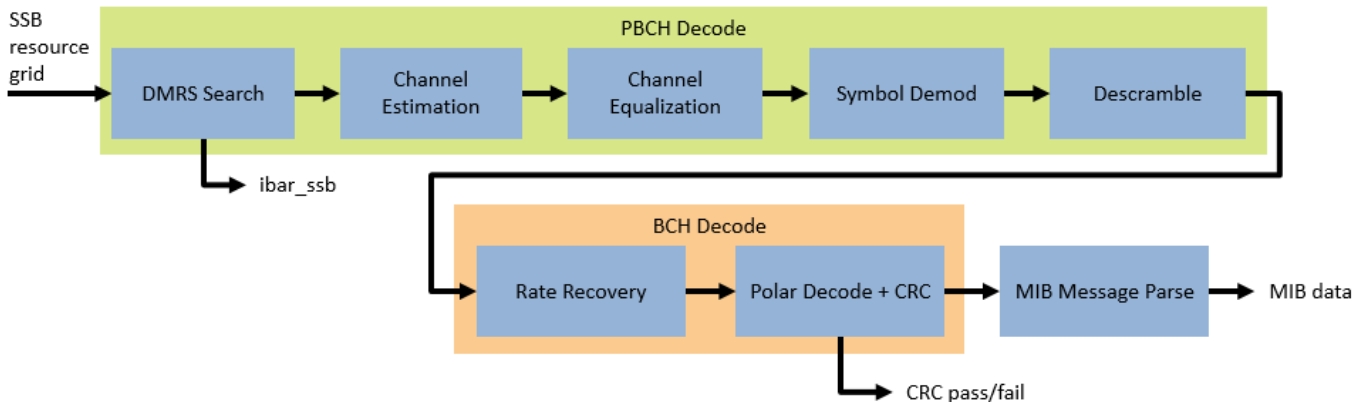
The cell search algorithm uses timing offsets to identify positions within the received waveform and intermediate signals. A timing offset is the number of samples from the start of the waveform to a given position, such as the start of an SSB. Timing offsets are given in samples at 61.44 Msps and wrap around every 20 ms, or 1228800 samples. In 5G NR, UEs can assume that the SS burst periodicity is 20 ms or less for cell search purposes, hence the reason for this choice of timing reference periodicity.

The figure shows two 5G waveforms with different SS burst periodicities (5 ms and 20 ms) and the receiver timing reference. The MATLAB reference can detect SSBs at any position within the received waveform. However, if the waveform is longer than 20 ms, ambiguity in the returned timing offsets exists because the timing reference wraps around every 20 ms. Additionally, the receiver can demodulate only SSBs that begin within the first 20 ms of the waveform.



## SSB Decoding

The diagram shows the structure of the SSB decoder, which is implemented by the `nrhdlexamples.ssbDecode` function. The algorithm takes the SSB resource grid from the OFDM demodulation phase of the SSB detector, processes it through PBCH and BCH decoding, and outputs MIB parameters and PBCH timing information.



PBCH decoding takes the demodulated OFDM symbols of the resource grid and processes using these steps:

- *DMRS Search*: Searches for the index used for demodulation reference symbol (DMRS) generation
- *Channel Estimation*: Calculates an estimate of the channel using the DMRS
- *Channel Equalization*: Equalizes the received data using the channel estimate
- *Symbol Demod*: Performs QPSK demodulation to get the PBCH soft bits
- *Descramble*: Descrambles the soft bits

BCH Decode then processes the descrambled soft bits to recover the MIB data using these steps:

- *Rate Recovery*: Combines repeated soft bits then performs scaling and quantization
- *Polar Decode + CRC*: Performs polar decoding to get the message bits and CRC decoding to check for errors
- *MIB Message Parse*: Interprets the decoded message bits to produce the MIB parameter outputs

### Generate a Test Waveform

This section shows how to use the MATLAB reference functions to search for SSBs in a waveform.

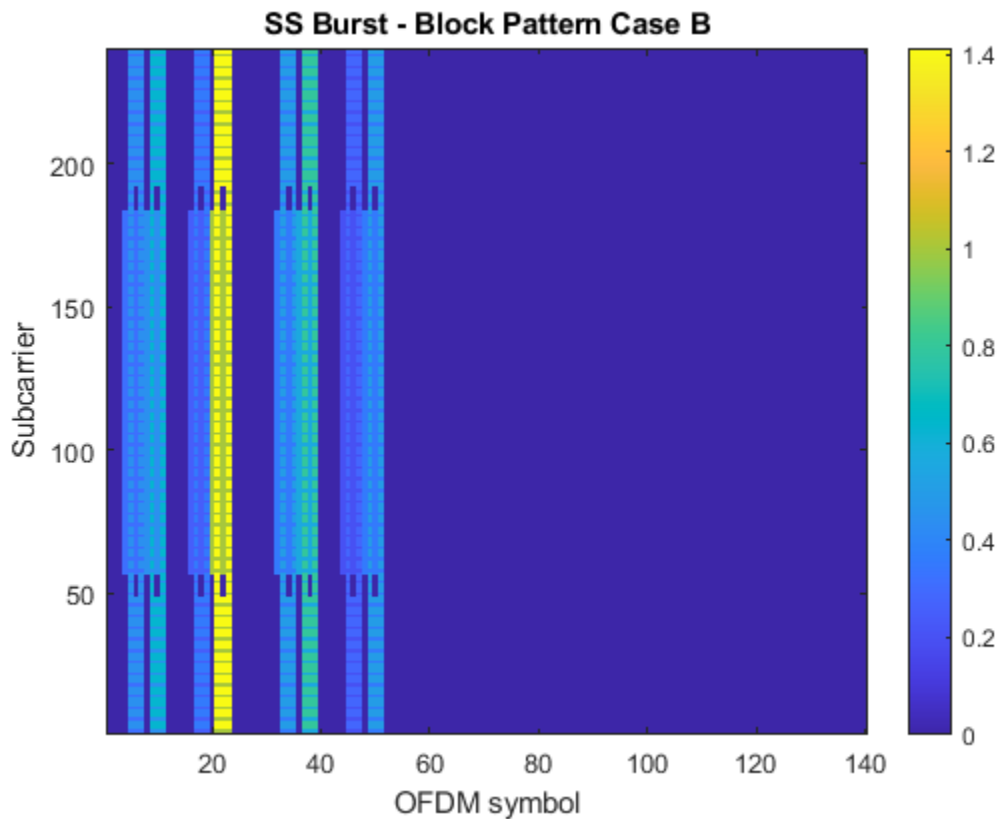
Use the `nrhdlexamples.generateSSBurstWaveform` function to generate an SS burst waveform. This function is based on the “Synchronization Signal Blocks and Bursts” (5G Toolbox) example. The burst has these parameters.

- SSB pattern is case B
- Subcarrier spacing is 30 kHz
- NCellID is 249
- Active SSBs within the burst is 8

```
rng('default');
[rxWaveform,txGrid,txMIB] = nrhdlexamples.generateSSBurstWaveform();
```

Plot the resource grid of the burst waveform. The amplitude of each resource element is indicated by its color. The plot shows eight SSBs. The SSBs are generated with different power levels to model what a UE typically receives.

```
figure(1); clf;
imagesc(abs(txGrid));
colorbar;
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS Burst - Block Pattern Case B');
```



### Detect SSBs

Use the `nrhdlexamples.ssbDetect` function to find SSBs in the waveform by searching for PSS symbols. This example calls the function with a coarse carrier frequency offset estimate of zero and a subcarrier spacing of 30. The function corrects the coarse frequency offset and measures the residual fine frequency offset of each SSB. Frequency offset input and output are give in Hz. The function returns a list of detected PSS symbols as a structure array. Display the structure array contents by converting it to a table.

```
FoCoarse = 0;
scs = 30;
[pssList,diagnostics] = nrhdlexamples.ssbDetect(rxWaveform,FoCoarse,scs);

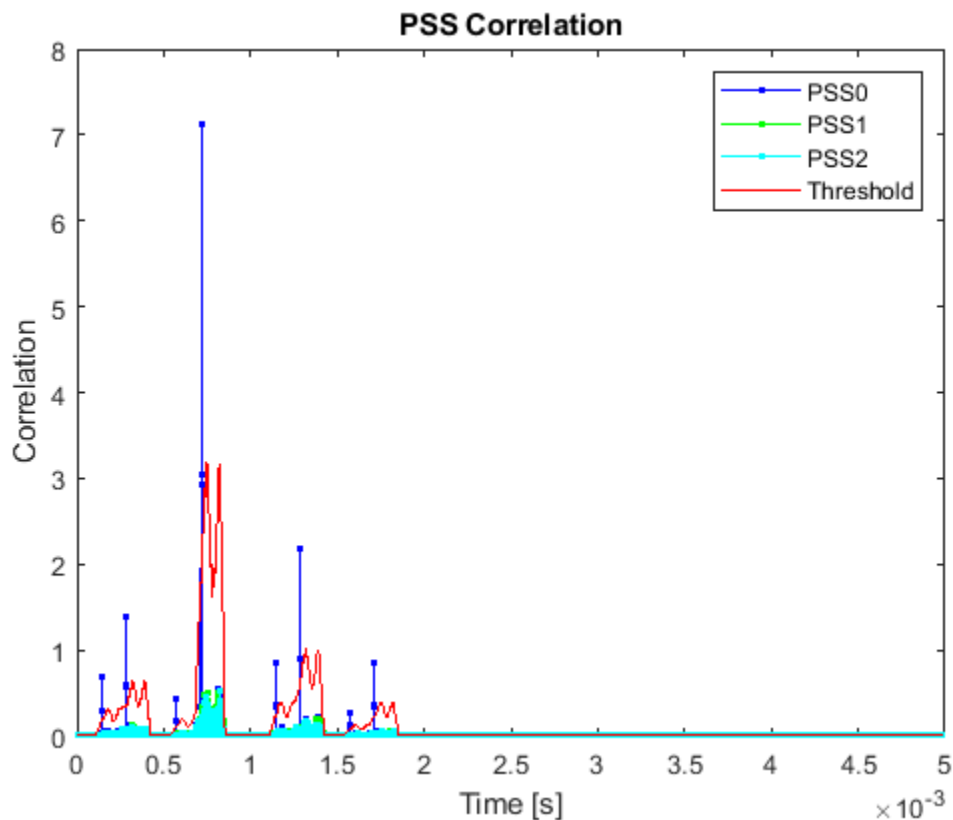
disp(struct2table(pssList));
```

NCellID2	timingOffset	pssCorrelation	pssEnergy	frequencyOffset
0	6608	0.69435	0.70703	7

0	15376	1.3933	1.4119	-51
0	32944	0.43994	0.44712	-207
0	41712	7.1226	7.2182	-154
0	68048	0.84535	0.88463	204
0	76816	2.1805	2.245	140
0	94384	0.2794	0.28375	488
0	1.0315e+05	0.8552	0.89668	132

The `nrhdlexamples.ssbDetect` function also returns a structure containing diagnostic signals. Use this output to plot the PSS correlation results. Each peak in the correlator output shown corresponds to an entry in the PSS list.

```
figure(2); clf;
nrhdlexamples.plotUtils.PSSCorrelation(diagnostics, 'PSS Correlation');
```



Use the `nrhdlexamples.ssbDetect` function to OFDM-demodulate one of the SSBs and attempt SSS detection. For this operation, call the function with an optional 4th argument that specifies the timing offset and `NCellID2` of the desired SSB. This example chooses the PSS with the highest correlation metric, however you can choose any of the detected SSBs. Correct the frequency offset by passing in the sum of the coarse and fine frequency offset estimates.

```
[~,maxCorrIdx] = max(vertcat(pssList.pssCorrelation));
chosenPSS = pssList(maxCorrIdx);
FoFine = chosenPSS.frequencyOffset;
FoEst = FoCoarse + FoFine;
```



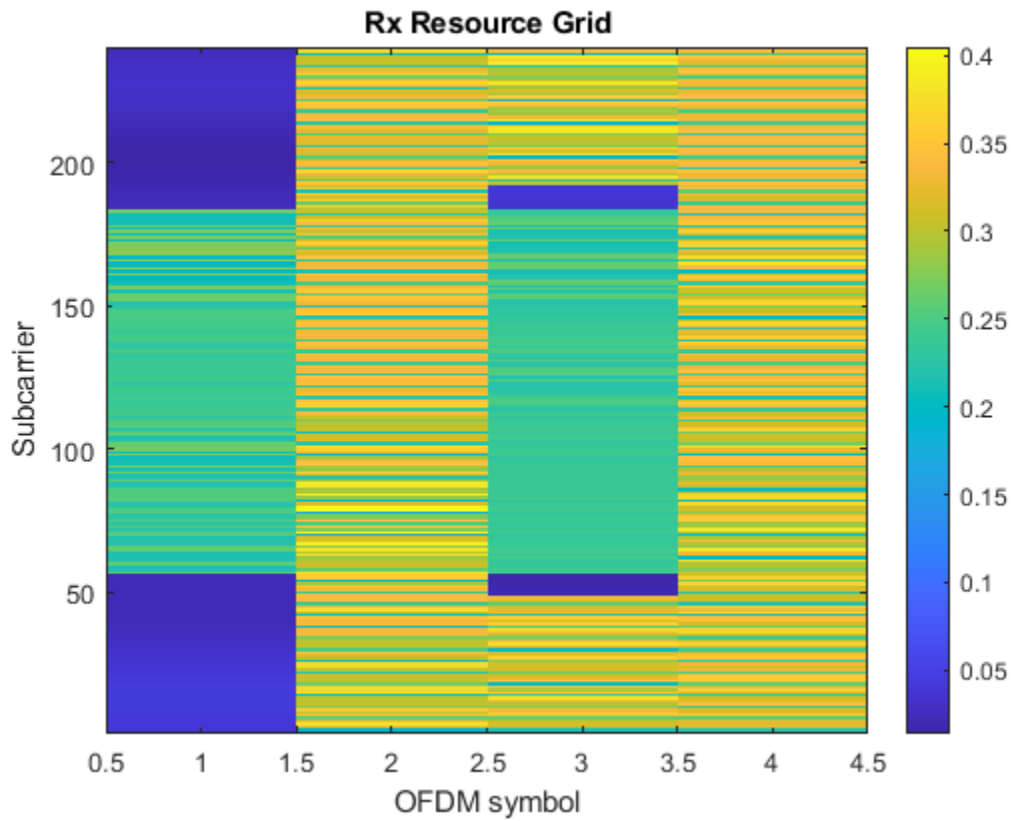
```
[ssBlockInfo,ssGrid,diagnostics] = nrhdlexamples.ssbDetect(rxWaveform,FoEst,scs,chosenPSS);
```

In demodulation mode, the function returns three outputs instead of two. The `ssBlockInfo` structure contains further details of the SSB, such as the SSS correlation strength and the overall cell ID. The `ssGrid` output is a matrix containing the demodulated OFDM symbols. Display the SSB info to confirm that the cell ID is correctly decoded.

```
disp(ssBlockInfo);  
  
      NCellID2: 0  
      timingOffset: 41712  
      pssCorrelation: 7.1219  
      pssEnergy: 7.2185  
      NCellID1: 83  
      sssCorrelation: 7.1383  
      sssEnergy: 7.1743  
      NCellID: 249  
      frequencyOffset: 0
```

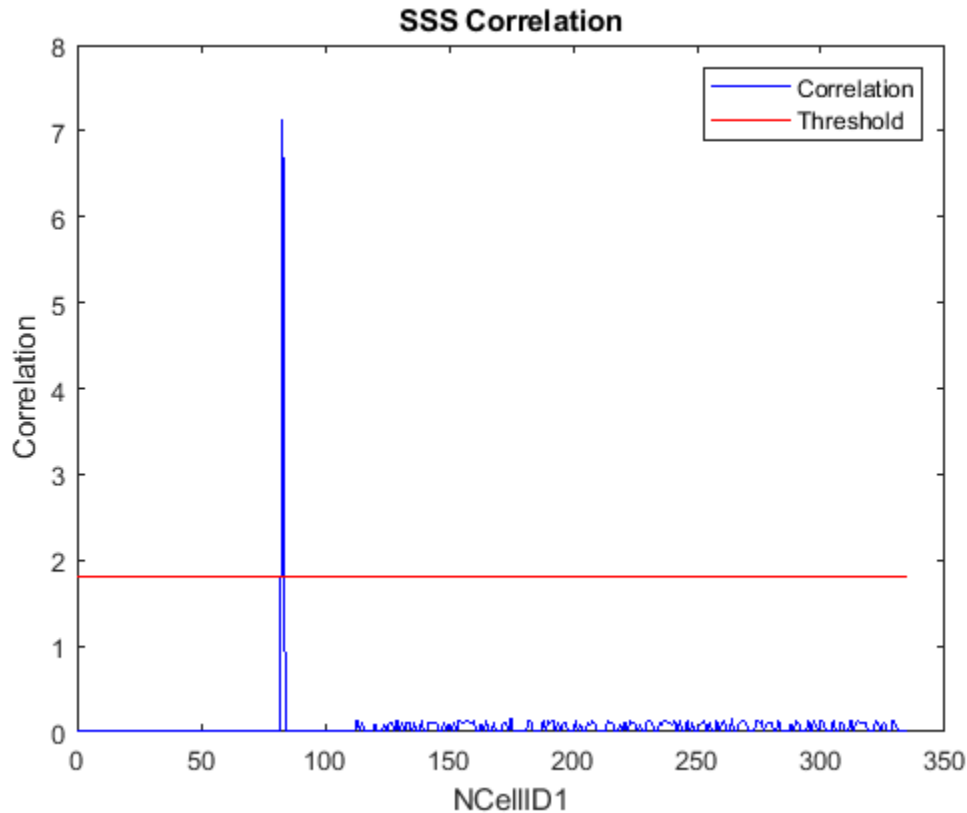
Display the resulting resource grid.

```
figure(3); clf;  
imagesc(abs(ssGrid));  
colorbar;  
axis xy;  
xlabel('OFDM symbol');  
ylabel('Subcarrier');  
title('Rx Resource Grid');
```



The diagnostics output includes SSS correlation results for all 336 possible sequences. Plot the SSS correlation results.

```
figure(4); clf;  
nrhdlexamples.plotUtils.SSSCorrelation(diagnostics, 'SSS Correlation')
```



### Search for Cells

This section shows how to use the `nrhdlexamples.cellSearch` function to search for and demodulate SSBs when the frequency offset and subcarrier spacing are not known. As described previously, the `nrhdlexamples.cellSearch` function builds on the `nrhdlexamples.ssbDetect` function by adding a search controller that looks for SSBs at different subcarrier spacings and frequency offsets.

Apply a frequency offset to test the coarse and fine frequency recovery functionality.

```
Fo          = 10000;
t           = (0:length(rxWaveform)-1) ./ 61.44e6;
rxWaveformFo = rxWaveform .* exp(1i*2*pi*Fo*t);
```

Define the frequency range endpoints and subcarrier spacing search space and call the `nrhdlexamples.cellSearch` function. The function displays information on the search progress as it runs. The frequency range endpoints must be multiples of half the maximum subcarrier spacing.

```
frequencyRange = [-30 30];
subcarrierSpacings = [15 30];
```

```
[ssBlockInfo,ssGrid] = nrhdlexamples.cellSearch(rxWaveformFo,frequencyRange,subcarrierSpacings,s
    'DisplayPlots',false,...
    'DisplayCommandWindowOutput',true));
```

```
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -30 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -22.5 kHz)
```

```

Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -15 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -7.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 0 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 7.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 15 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 22.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 30 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: -30 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: -15 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: 0 kHz) ... PSS detected.
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: 15 kHz) ... PSS detected.
Found PSS with (subcarrierSpacing: 30 kHz, frequencyOffsetEstimate: 9846 Hz)
Correcting frequency offset and searching for PSS again.
Found the following PSS symbols:

```

NCellID2	timingOffset	pssCorrelation	pssEnergy	frequencyOffset
0	6608	0.69422	0.70705	161
0	15376	1.3933	1.412	103
0	32944	0.43981	0.44714	-53
0	41712	7.1219	7.2185	0
0	68048	0.84567	0.88466	358
0	76816	2.1812	2.2451	294
0	94384	0.2793	0.28376	642
0	1.0315e+05	0.85524	0.89673	286

Strongest PSS:

```

NCellID2: 0
timingOffset: 41712
pssCorrelation: 7.1219
pssEnergy: 7.2185
frequencyOffset: 0

```

Attempting to reacquire strongest PSS and demodulate the corresponding SS block.

```

NCellID2: 0
timingOffset: 41712
pssCorrelation: 7.1219
pssEnergy: 7.2185
NCellID1: 83
sssCorrelation: 7.1383
sssEnergy: 7.1743
NCellID: 249
frequencyOffset: 9846
subcarrierSpacing: 30

```

Cell search summary:

```

Subcarrier spacing: 30 kHz
Frequency offset: 9846 Hz
Timing offset: 41712
NCellID: 249

```

As shown in the summary, the receiver returned the correct subcarrier spacing of 30 kHz, a cell ID of 249, and the measured frequency offset is close to the expected value of 10 kHz.

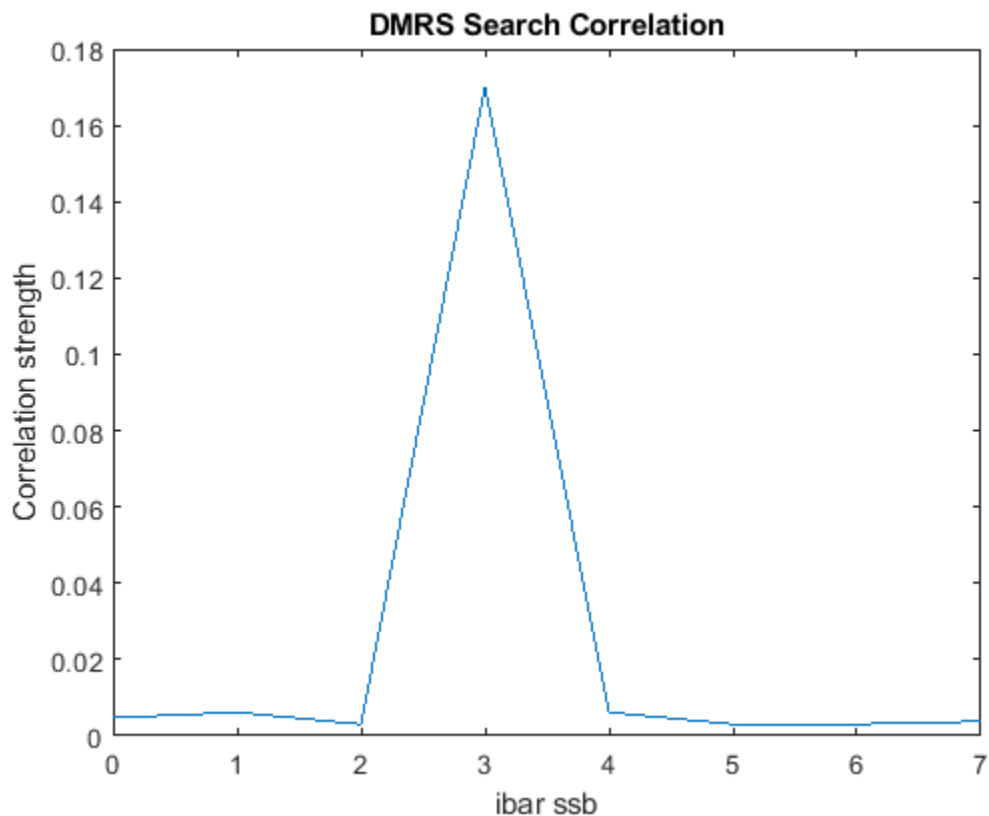
## Decode SSB

Use the `nrhdlexamples.ssbDecode` function to decode the resource grid and recover the MIB. The `nrhdlexamples.ssbDecode` function is based on the BCH decoding stages of the “NR Cell Search and MIB and SIB1 Recovery” (5G Toolbox) example.

```
[mibInfo,decodeDiags] = nrhdlexamples.ssbDecode(ssGrid,ssBlockInfo.NCellID,8);
```

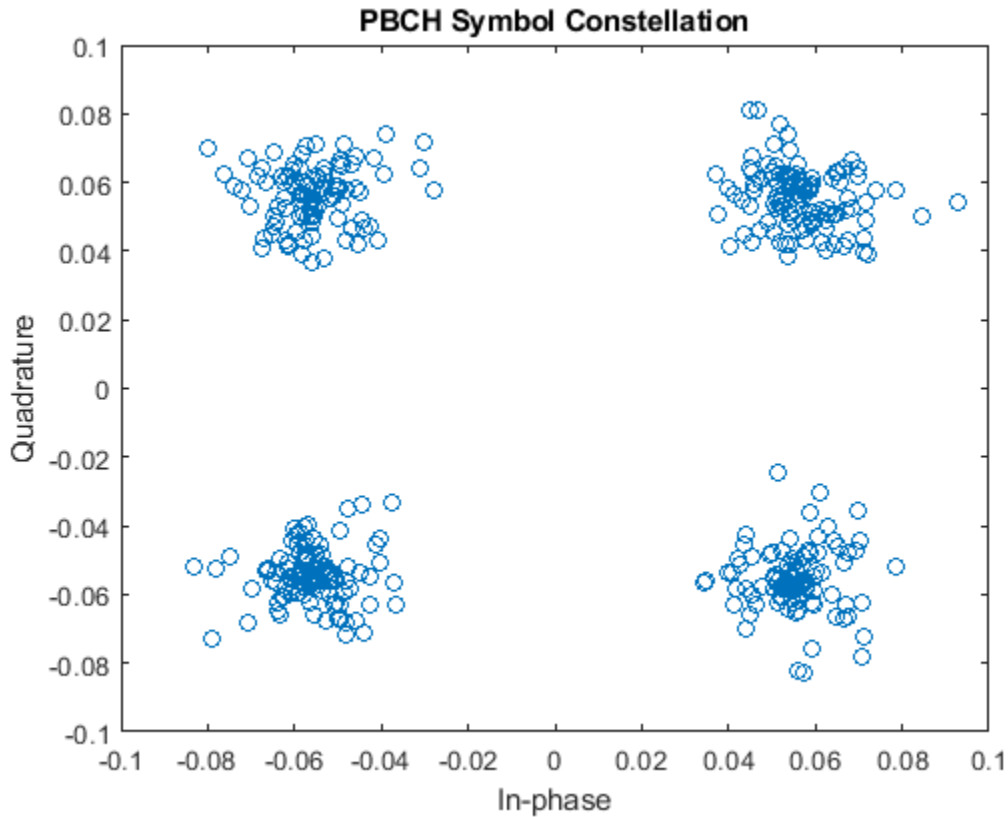
Plot the correlation peaks for the DMRS search. DMRS search is performed to determine `ibar_ssb` and the SSB index.

```
figure(5); clf;
plot(0:7,decodeDiags.dmrsCorr);
title('DMRS Search Correlation');
xlabel('ibar_ssb');
ylabel('Correlation strength');
```



Plot the PBCH QPSK constellation after phase equalization.

```
figure(6); clf;
plot(decodeDiags.qpskSymb,'o');
title('PBCH Symbol Constellation');
xlabel('In-phase');
ylabel('Quadrature');
```



Display the decoded information and compare the transmitted and received MIB structures. These result show that the information was successfully decoded.

```
disp(['BCH CRC: ' num2str(mibInfo.err) newline]);
```

```
disp('Decoded information');
disp(mibInfo);
```

```
disp('Decoded MIB');
disp(mibInfo.mib);
```

```
disp('Expected MIB');
disp(txMIB);
```

```
BCH CRC: 0
```

```
Decoded information
  pbchPayload: 218103952
    ssbIndex: 3
      hrf: 0
      err: 0
      mib: [1x1 struct]
```

```
Decoded MIB
          NFrame: 105
SubcarrierSpacingCommon: 30
              k_SSB: 0
DMRSTypeAPosition: 2
```

```
PDCCHConfigSIB1: 0  
  CellBarred: 0  
IntraFreqReselection: 0
```

Expected MIB

```
  NFrame: 105  
SubcarrierSpacingCommon: 30  
  k_SSB: 0  
DMRSTypeAPosition: 2  
  PDCCHConfigSIB1: 0  
    CellBarred: 0  
IntraFreqReselection: 0
```

## See Also

### Related Examples

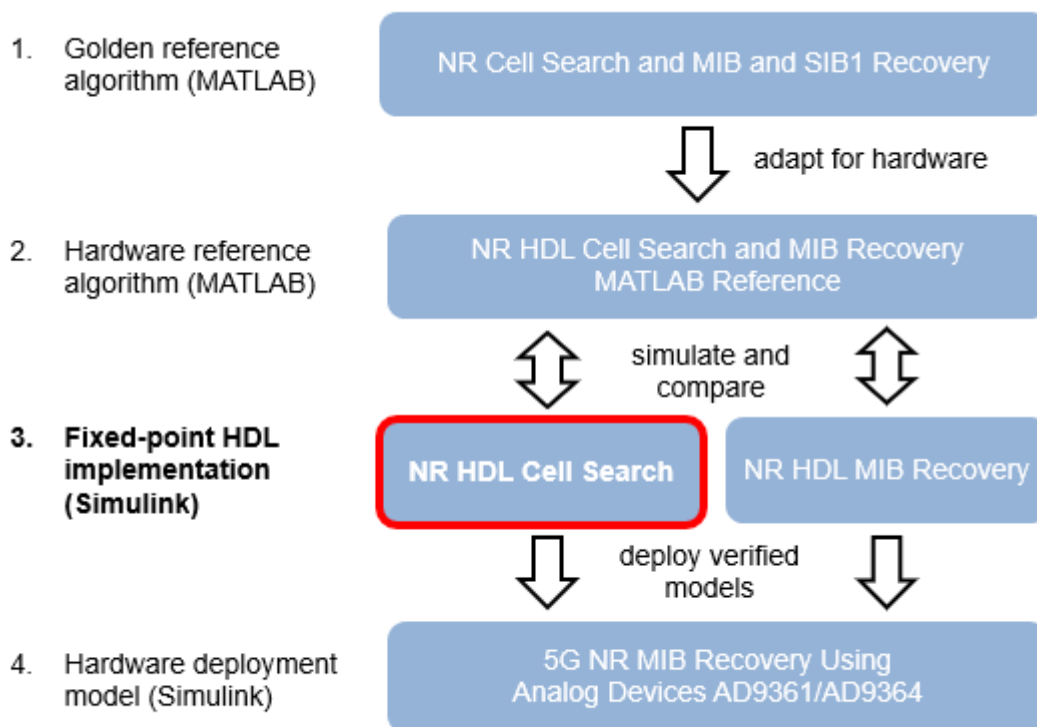
- “NR HDL Cell Search” on page 5-30
- “NR HDL MIB Recovery” on page 5-2

## NR HDL Cell Search

This example shows the design of a 5G NR cell search subsystem optimized for HDL code generation and hardware implementation.

### Introduction

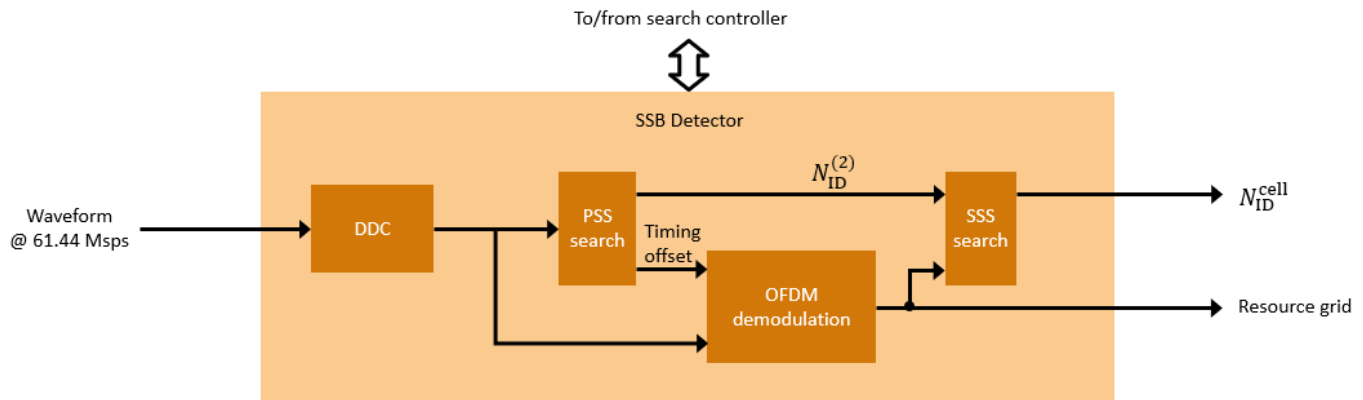
The Simulink model described in this example is an HDL optimized implementation of a synchronization signal block (SSB) detector for 5G NR cell search. This example is one of a related set which show the workflow for designing and deploying a 5G NR cell search and MIB recovery algorithm to hardware. The figure shows the complete set of examples and the current example within the workflow. For more details on the overall algorithm and workflow, see the “NR HDL Cell Search and MIB Recovery MATLAB Reference” on page 5-14 example. The “5G NR MIB Recovery Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example shows how to deploy the algorithm to an SoC. For a general description of how MATLAB and Simulink can be used together to develop deployable models, see “Wireless Communications Design for FPGAs and ASICs”.



A block diagram of the SSB detector is shown in the figure. This performs all of the high speed signal processing tasks associated with the cell search algorithm therefore is well suited for FPGA or ASIC implementation. The SSB detector searches for SSBs in time at a given frequency offset and subcarrier spacing. It is designed to be used as part of a larger system that implements carrier frequency offset recovery and subcarrier spacing detection. A controller must be used co-ordinate the overall cell search as shown in the “5G NR MIB Recovery Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.



The SSB detector performs primary synchronization sequence (PSS) search, orthogonal frequency division multiplexing (OFDM) demodulation, and secondary synchronization sequence (SSS) search. It also includes a digital down converter (DDC) for correcting frequency offsets in the received signal.



## File Structure

This example uses these files.

- `nrdlexamples.runSSBDetectionModel`: This script runs the `nrdLSSBDetection` model by calling the `nrdlexamples.ssbDetectSimulink` function and verifies the model by using 5G Toolbox functions and the “NR HDL Cell Search and MIB Recovery MATLAB Reference” on page 5-14.
- `nrdlexamples.ssbDetectSimulink`: This function runs the Simulink model and has the same input and output arguments as the `nrdlexamples.ssbDetect` function from the MATLAB reference.
- `nrdLSSBDetection`: This top-level Simulink model is a lightweight wrapper that instantiates the `nrdLSSBDetectionCore` model reference.
- `nrdLSSBDetectionCore`: This model reference implements the SSB detection algorithm.

This example also uses helper functions from the `nrdlexamples` package. The Simulink models and the `nrdlexamples` package are on the MATLAB path. To open one of the models, enter its name at the MATLAB command prompt.

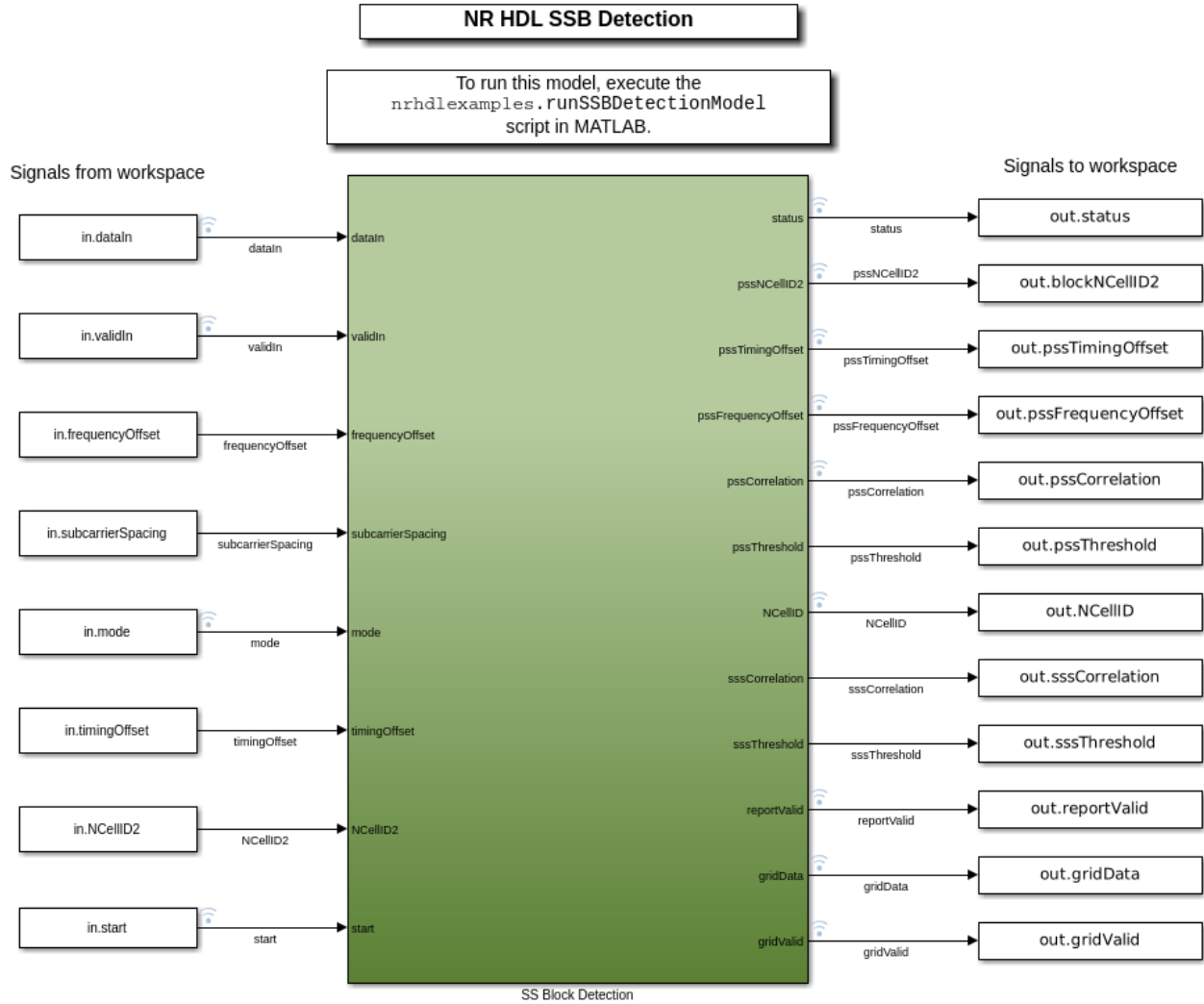
```
nrdLSSBDetection
```

To open a function or script from the `nrdlexamples` package, use the `edit` command.

```
edit nrdlexamples.runSSBDetectionModel
```

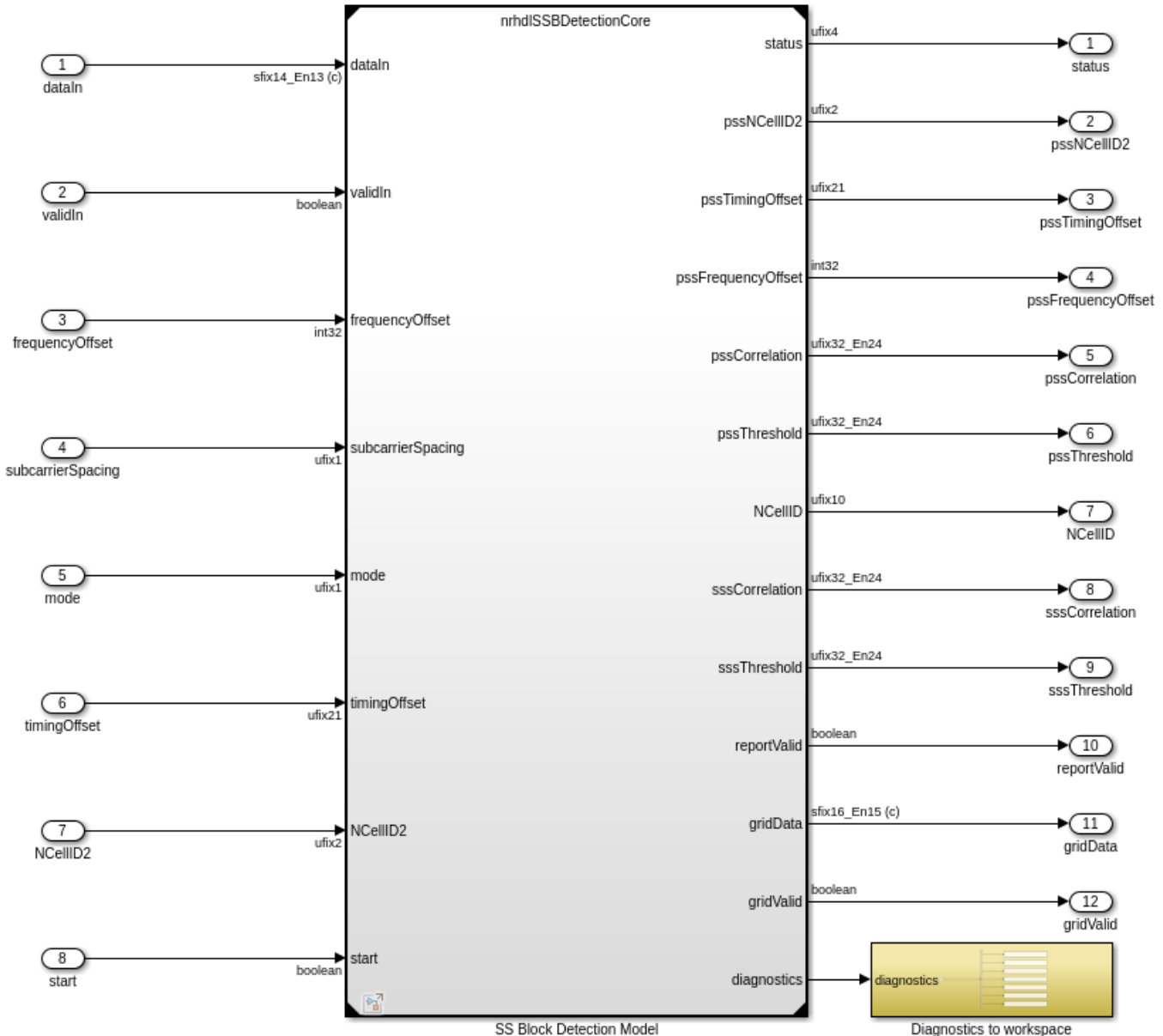
## NR HDL Cell Search Model

This figure shows the `nrdLSSBDetection` model. The top level of the model reads signals from the MATLAB base workspace, passes them to the SS Block Detection subsystem, and writes the outputs back to the workspace. To run the model, call the `nrdlexamples.runSSBDetectionModel` script in MATLAB.



### SS Block Detection Interface

The SS Block Detection subsystem contains a Model block that references the `nrhdlSSBDetectionCore` model. This section describes the inputs and outputs of that model.



## Inputs

- *dataIn*: 14-bit signed complex-valued signal, sampled at 61.44 Msps.
- *validIn*: 1-bit control signal to validate *dataIn*.
- *frequencyOffset*: 32-bit signed value specifying the frequency offset to be corrected. This signal is connected to an NCO with a 32-bit accumulator. Use this equation to convert the value to Hz:  

$$\text{frequencyOffset\_Hz} = \text{frequencyOffset} * 61.44\text{e}6 / 2^{32};$$
- *subcarrierSpacing*: 1-bit unsigned value specifying the subcarrier spacing. Set this signal to 0 to select 15kHz, or 1 to select 30kHz.
- *mode*: 1-bit unsigned value specifying the operation mode. Set this signal to 0 for search mode, or 1 for demod mode.

- *timingOffset*: 21-bit unsigned value specifying the timing offset of the start of the SSB to be demodulated. Specify the timing offset in samples at 61.44 Msps, from 0 to 1228799. This parameter applies only for demod mode.
- *NCellID2*: 2-bit unsigned value specifying the PSS (0, 1, or 2) of the SSB to be demodulated. This parameter applies only for demod mode.
- *start*: 1-bit control signal used to start a search or demodulation operation. To start an operation, set *frequencyOffset*, *subcarrierSpacing*, *mode*, *timingOffset*, and *NCellID2* to the desired values and set *start* to 1 (*true*) for one or more cycles. If an operation is already in progress, that operation is cancelled when *start* is set to 1 (*true*). The new operation begins when *start* is returned to 0 (*false*).

### Outputs

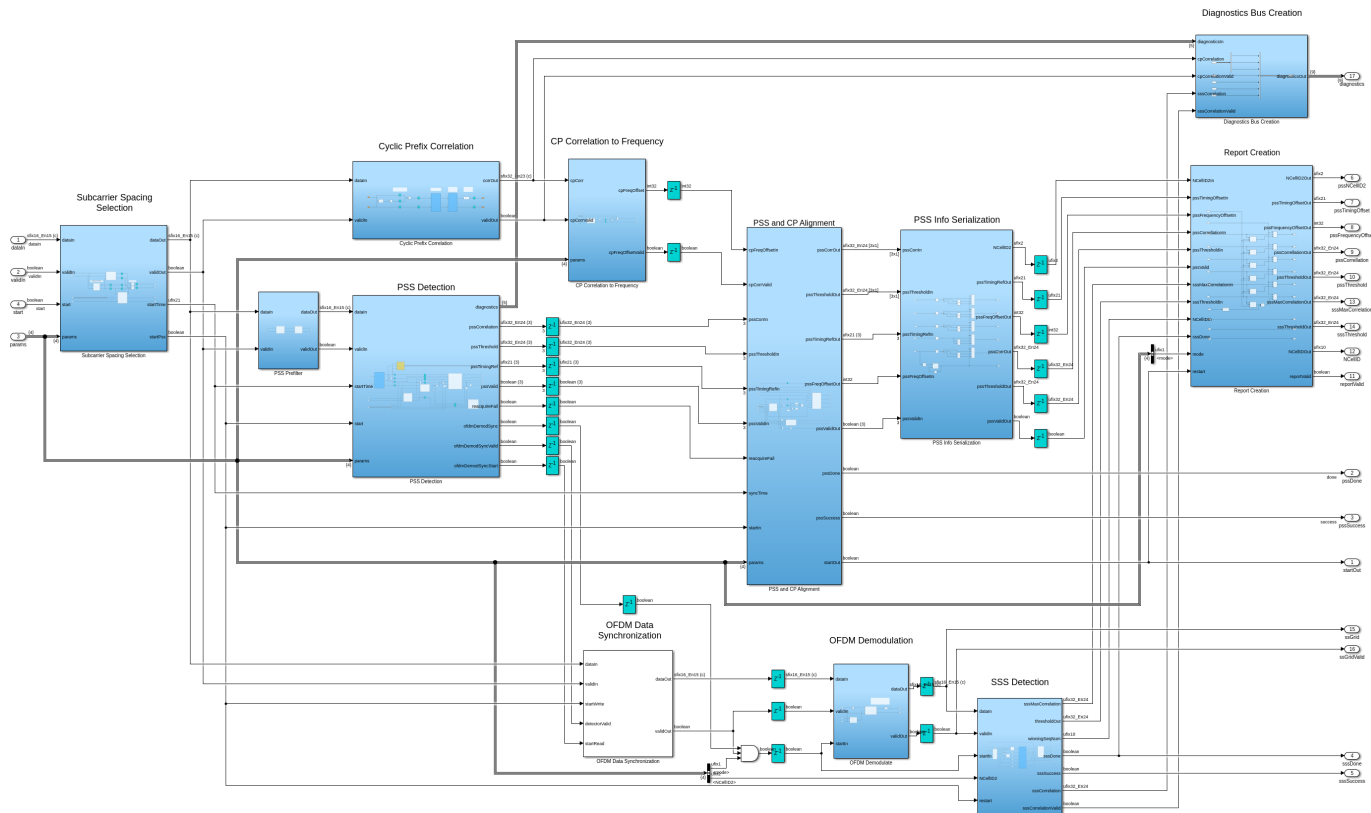
- *status*: 4-bit unsigned value that indicates the progress of the current operation. See the next section for the possible values of this signal.
- *pssNCellID2*: 2-bit unsigned value that is the PSS (0, 1 or 2) of the detected SSB.
- *pssTimingOffset*: 21-bit unsigned value that is the timing offset of the detected SSB. The timing offset is in samples at 61.44 Msp from 0 to 1228799.
- *pssFrequencyOffset*: 32-bit signed value that is the frequency offset of the detected SSB. This signal has the same units as the *frequencyOffset* input.
- *pssCorrelation*: 32-bit unsigned value that is the strength of the PSS correlation.
- *pssThreshold*: 32-bit unsigned value that is the threshold value when PSS was detected.
- *NCellID*: 10-bit unsigned value that is the cell ID of the demodulated SSB. This value is returned only in demod mode.
- *sssCorrelation*: 32-bit unsigned value that is the SSS correlation strength. This signal is returned only in demod mode.
- *sssThreshold*: 32-bit unsigned value that is the SSS threshold. This value is returned only in demod mode.
- *reportValid*: 1-bit control signal. In search mode, this signal validates *pssNCellID2*, *pssTimingOffset*, *pssFrequencyOffset*, *pssCorrelation*, and *pssThreshold* for each PSS that is detected. In demod mode, this signal also validates *NCellID*, *sssCorrelation*, and *sssThreshold*. In demod mode, *sssCorrelation* and *sssThreshold* are only valid if the specified SSB was found using its PSS, and *NCellID* is only valid if the SSS was detected.
- *gridData*: 16-bit signed complex-values that are the resource grid data. The receiver returns all four symbols of the SSB resource grid. Values are returned one resource element at a time. The resource grid is only returned in demod mode.
- *gridValid*: 1-bit control signal that validates the *gridData* output. Data is only returned if the specified SSB was found using its PSS. This signal is returned only in demod mode.
- *diagnostics*: Bus containing diagnostic signals.

### Status Signal States

- 0 Idle.
- 1 Search mode -- Searching for PSS.
- 2 Search mode -- Operation complete, no PSS found.
- 3 Search mode -- Operation complete, found one or more PSSs.
- 4 Demod mode -- Waiting for specified PSS timing offset.



time to the other timing references in the model. This signal tells the other timing references when a new subcarrier spacing and corresponding sampling rate applies. The other timing references wait until the start time before changing their increment. This design is possible only because hardware latency means the other timing references lag behind the Start Controller. This architecture enables the receiver to keep track of time consistently, even when a sampling rate change occurs.



The SS Block Core subsystem contains these main subsystems.

- *Subcarrier Spacing Selection*: Converts the input to two synchronized sample streams, one at 7.68 Msps and one at 3.84 Msps, and selects which stream to pass to subsequent processing stages according to the subcarrier spacing.
- *PSS Detection*: Searches for PSS symbols in the received signal. The next section describes this subsystem in more detail.
- *Cyclic Prefix Correlation*: Computes cyclic prefix (CP) correlation values. Each result is averaged across the last four OFDM symbols.
- *CP Correlation to Frequency*: Converts CP correlation values to fine frequency offset estimates.
- *PSS and CP Alignment*: Matches a CP-based frequency estimate with each PSS symbol detection instance. This alignment is necessary because the frequency estimate for a given PSS detection instance is available only at the end of the corresponding SSB.
- *PSS Info Serialization*: If PSS is detected on more than one PSS correlator output at the same timing offset, this block serializes the results so that they are returned from the detector one at a time.
- *OFDM Data Synchronization*: Synchronizes the OFDM demodulator input with the output of the PSS detector. This synchronization enables the PSS detector to trigger the OFDM demodulation

process at the correct time. The synchronized data is one OFDM symbol behind the PSS correlator as the peak detection occurs at the end of the first OFDM symbol to be demodulated.

- *OFDM Demodulation*: OFDM-demodulates `all four symbols of the specified SSB.
- *SSS Detection*: Extracts the SSS resource elements from the OFDM demodulator output and correlates them with all 336 possible sequences to determine the cell ID.
- *Create report*: Aligns all of the parameters corresponding to one SSB detection, so that they are all valid at the same time.

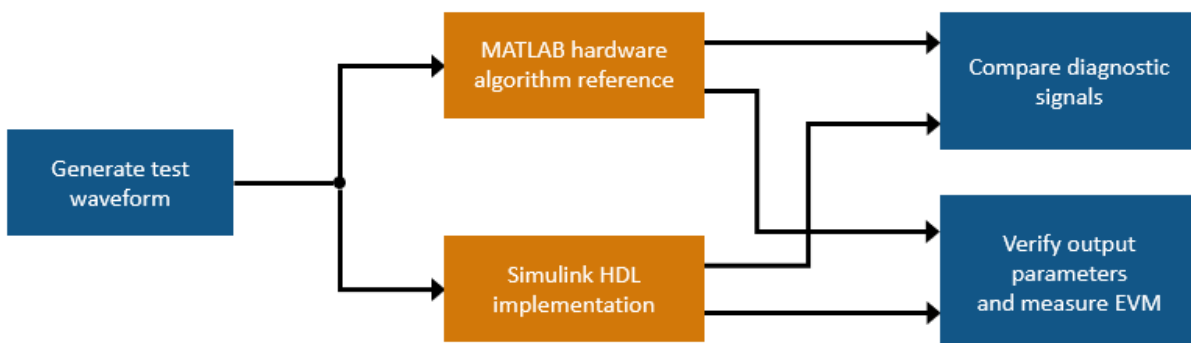
## Simulation Setup

This block diagram shows the simulation setup implemented by the `nrhdlexamples.runSSBDetectionModel` function. The function uses 5G Toolbox functions to generate a test waveform that has these parameters.

- SSB pattern is case B
- Subcarrier spacing is 30
- NCellID is 249
- Active SSBs within the burst is 8
- Frequency offset is 10 kHz
- Channel is modeled with AWGN

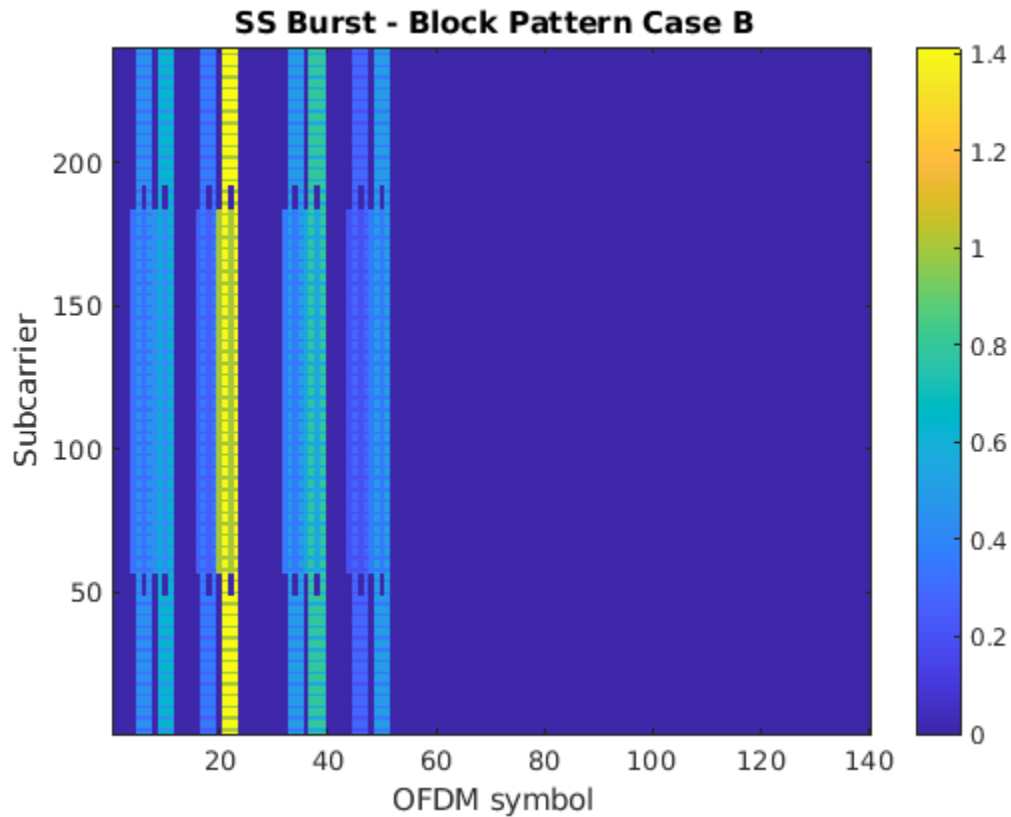
The function passes the test waveform to the MATLAB and Simulink implementations of the SSB detector in search mode and then in demodulation mode. Key diagnostic signals from each detector are compared in terms of their relative mean-squared error (MSE) and the final outputs are compared. The function calls 5G Toolbox functions to decode the BCH, recover the MIB, and measure the EVM of the resource grid returned by the Simulink implementation.

To run the simulation, call the `nrhdlexamples.runSSBDetectionModel` function. This diagram shows the actions performed by the script.



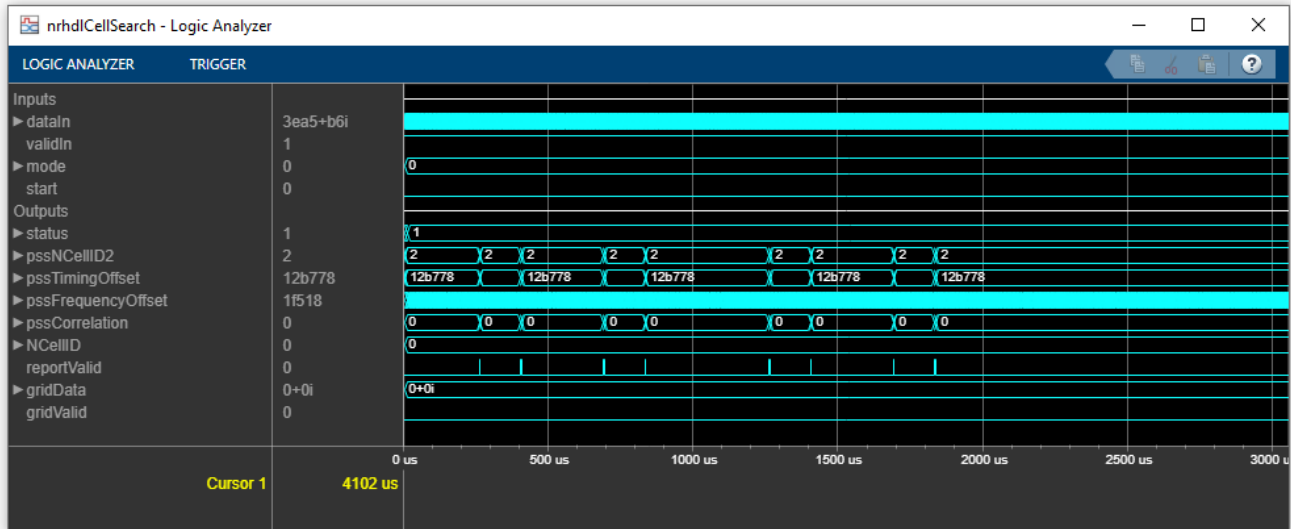
## Search Mode Simulation

The simulation starts by generating a test waveform as previously described. The plot shows the combined resource grid of all eight SSBs in the transmitted waveform.

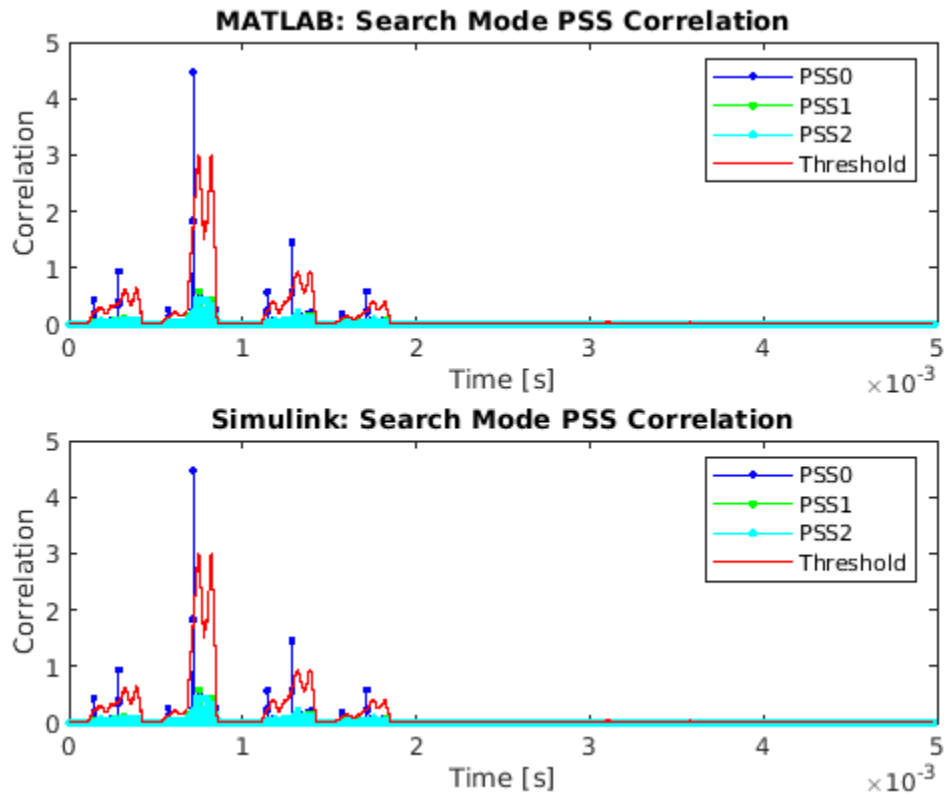


In search mode, the detector looks for PSS symbols within a 20 ms time window, which begins after a pulse on the *start* input triggers the search operation. If no PSS symbols are found after 20 ms, the detector sets the *status* output to 2 - indicating the search has failed. In this example, the detector finds all eight SSBs. The *status* output is set to 1 during the search, and a *report* is returned for each SSB by asserting the *reportValid* signal. View these signals by opening the Simulink Logic Analyzer tool after the search mode simulation is complete. The simulation runs for 5 ms. If the simulation ran for more than 20 ms, then the *status* output is eventually set to 3 - indicating the search has succeeded.





The script generates a plot that shows the PSS correlation outputs and the PSS detection threshold. Tables are displayed at the command line showing the parameters of each SSB. The final table shows the MSE between the MATLAB and Simulink implementations for each correlator output and for the detection threshold. These results show that the MATLAB and Simulink implementations match very closely. The small differences between the two implementations are due to quantization errors. These errors occur because the MATLAB reference uses floating-point data types, and the Simulink model uses fixed-point data types.



```
>> nrhdlexamples.runSSBDetectionModel
Searching for SSBs using the MATLAB reference.
Searching for SSBs using the Simulink model.
Running nrhdlSSBDetection.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlSSBDetectionCore is up to date.
```

## Build Summary

```
0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 2.4403s
.....
```

SSBs found by MATLAB reference:

<u>NCellID2</u>	<u>timingOffset</u>	<u>pssCorrelation</u>	<u>pssEnergy</u>	<u>frequencyOffset</u>
0	6608	0.44464	0.77583	9813
0	15376	0.95079	1.4485	9951
0	32944	0.26498	0.50451	9899
0	41712	4.4733	6.7781	9959
0	68048	0.57662	0.94766	10104
0	76816	1.4418	2.2051	9965
0	94384	0.19222	0.42516	9941
0	1.0315e+05	0.6041	0.9353	9985

SSBs found by Simulink model:

<u>NCellID2</u>	<u>timingOffset</u>	<u>pssCorrelation</u>	<u>pssEnergy</u>	<u>frequencyOffset</u>
0	6608	0.44453	0.77583	9813
0	15376	0.95087	1.4484	9951
0	32944	0.26509	0.50451	9899
0	41712	4.4724	6.7775	9959
0	68048	0.57638	0.94769	10104
0	76816	1.4419	2.2051	9965
0	94384	0.19233	0.42519	9941
0	1.0315e+05	0.60386	0.93529	9986

Relative mean-squared error between MATLAB and Simulink in search mode:

<u>name</u>	<u>relativeMSEdB</u>
{'PSS correlation 0'}	-66.829
{'PSS correlation 1'}	-61.967
{'PSS correlation 2'}	-61.62
{'PSS threshold' }	-79.185

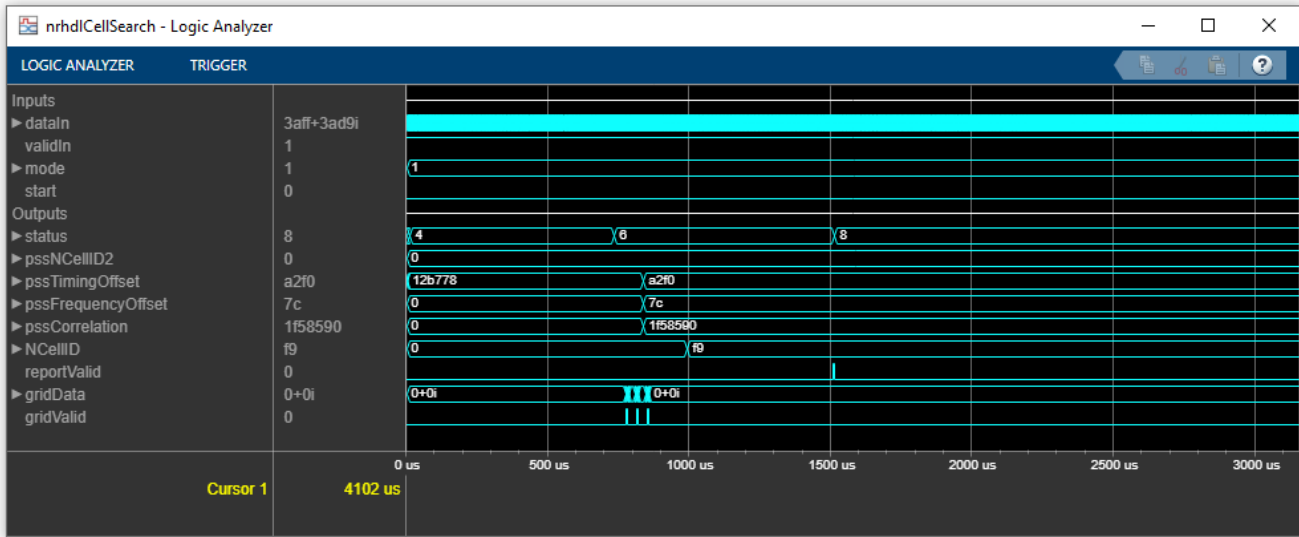
## Demodulation Mode Simulation

In demodulation mode, the detector recovers the specified SSB by searching for its PSS, OFDM-demodulating the resource grid, and searching for the SSS within the appropriate resource elements. If the PSS is not found at the specified timing offset, the detector sets the *status* output to 5 and stops searching.

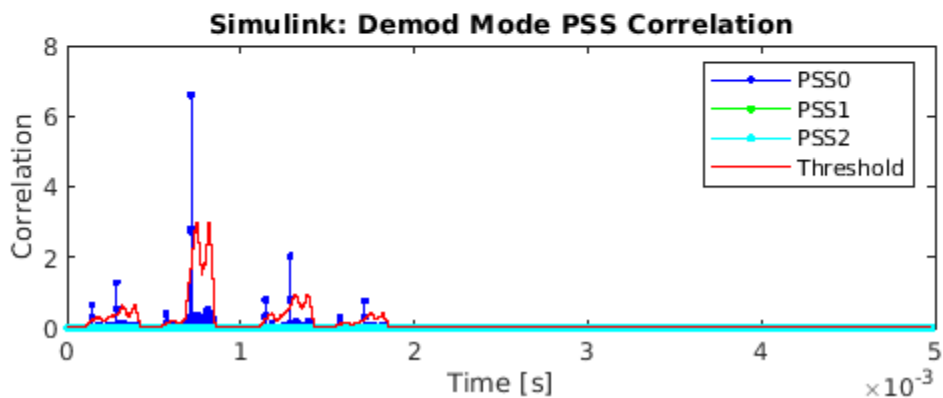
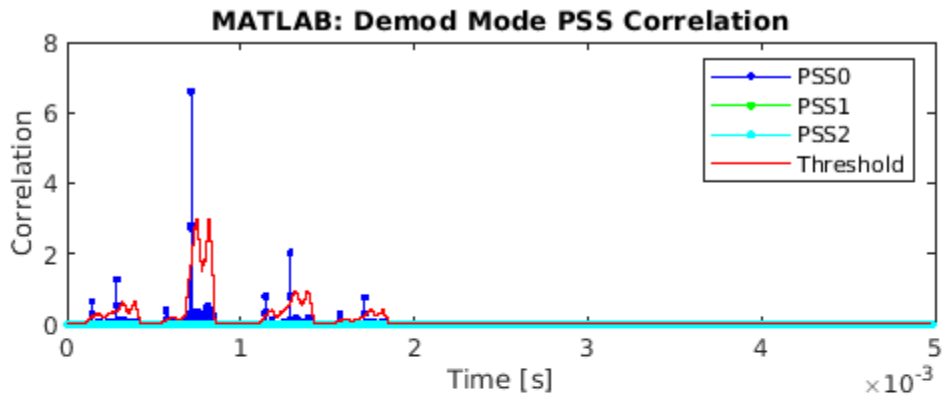
In this example, the detector recovers the specified SSB. The test bench in the `nrhdlexamples.runSSBDetectionModel` function chooses the SSB with the strongest PSS from the initial search. The frequency offset measurement from this SSB is used as a frequency offset estimate and passed back into the detector. The Simulink Logic Analyzer tool shows the detector output as it progresses through these steps.

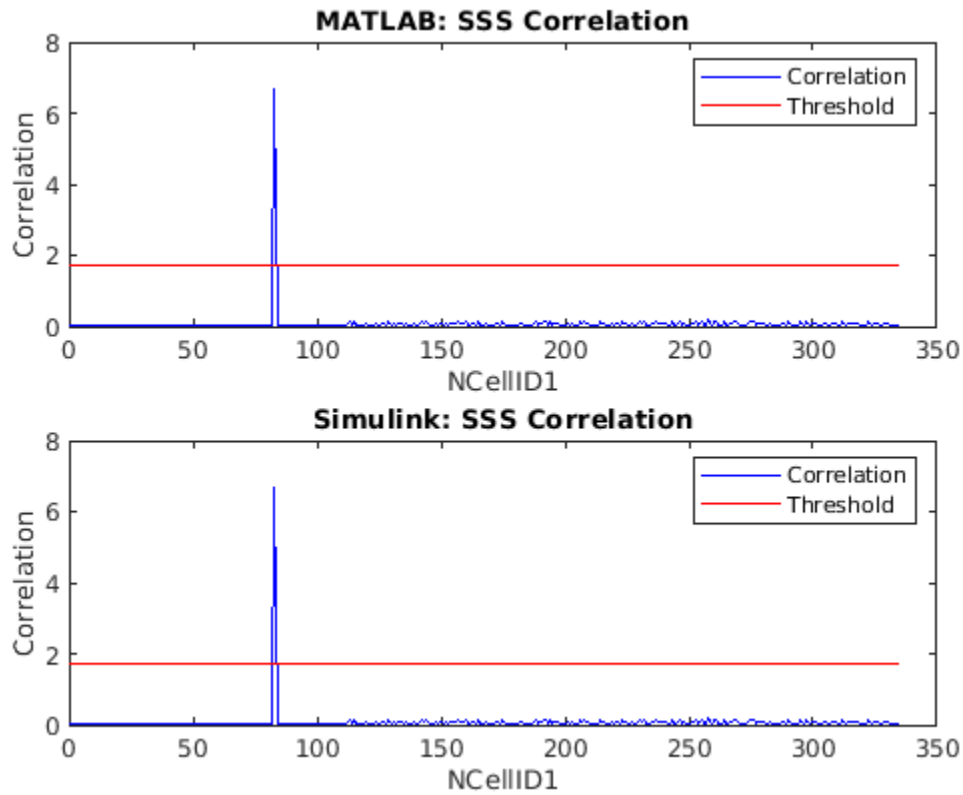
- 1 The detector sets *status* output to 4 while it waits for the specified timing offset and searches for the specified PSS.
- 2 PSS is found. The detector sets the *status* output to 6 - the detector is now searching for the SSS within the resource grid. The four demodulated OFDM symbols are output, indicated by asserting *gridValid*.
- 3 After the SSS is determined, the detector asserts *reportValid* to indicate that all of the PSS and SSS parameters, including *NCellID*, are valid. The *status* output changes to 8, to indicate that the operation is complete and SSS and cell ID are ready.

If the detector is unable to determine the SSS, then it sets the *status* output to 7.



These plots show the PSS and SSS correlation signals for both detector implementations. The PSS correlation levels are stronger in demod mode than in search mode because the frequency offset is corrected. The final reports are displayed at the command line, showing that both detectors returned similar parameters and determined the cell ID is 249. The table shows the relative MSE for the PSS and SSS correlation signals and the resource grid.





```
Demodulating the strongest SSBs using the MATLAB reference.
Demodulating the strongest SSBs using the Simulink model.
Running nrhdLSSBDetection.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdLSSBDetectionCore is up to date.
```

Build Summary

```
0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 0.8955s
.....
```

SS block report from MATLAB

```
NCellID2: 0
timingOffset: 41712
pssCorrelation: 6.6169
pssEnergy: 6.7620
NCellID1: 83
sssCorrelation: 6.7558
sssEnergy: 6.8791
NCellID: 249
frequencyOffset: 0
```

SS block report from Simulink

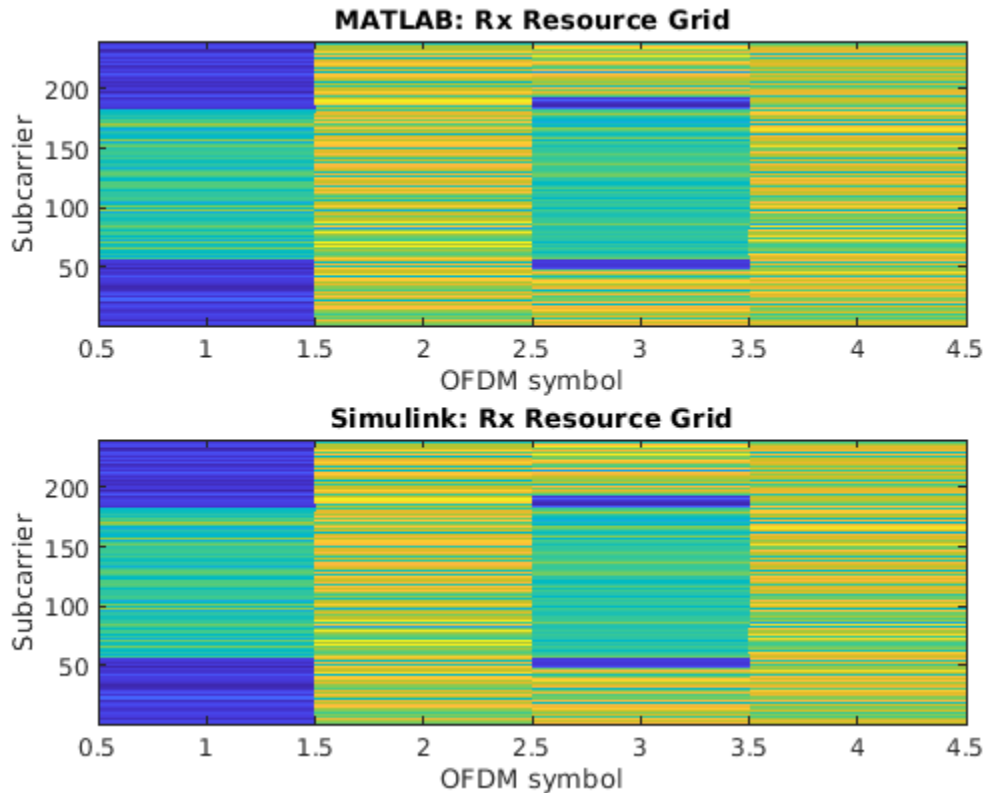
```
NCellID2: 0
timingOffset: 41712
pssCorrelation: 6.6175
pssEnergy: 6.7630
NCellID1: 83
sssCorrelation: 6.7552
sssEnergy: 6.8802
NCellID: 249
frequencyOffset: 0
```

Relative mean-squared error between MATLAB and Simulink in demod mode:

name	relativeMSEdB
{'PSS correlation 0'}	-69.231
{'PSS threshold' }	-78.437
{'SSS correlation' }	-70.056
{'Rx resource grid' }	-70.584

## Resource Grid Verification

The plot shows the demodulated resource grids returned by MATLAB and Simulink.



As a final verification step, the script decodes the BCH from the Simulink output by calling `nrdhlexamples.ssbDecode`. The command line output shows that the CRC check passed, and confirms that the MIB content matches that of the transmitted waveform.

Decoding BCH from Simulink resource grid output:

```
BCH CRC: 0

Decoded (Rx) MIB
  NFrame: 105
  SubcarrierSpacingCommon: 30
  k_SSB: 0
  DMRSTypeAPosition: 2
  PDCCHConfigSIB1: 0
  CellBarred: 0
  IntraFreqReselection: 0

Expected (Tx) MIB
  NFrame: 105
  SubcarrierSpacingCommon: 30
  k_SSB: 0
  DMRSTypeAPosition: 2
  PDCCHConfigSIB1: 0
  CellBarred: 0
  IntraFreqReselection: 0
```

## HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the

nrhdlSSBDetection/SS Block Detection subsystem. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 230 MHz.

Resource	Usage
Slice Registers	79357
Slice LUTs	37659
RAMB18	7
RAMB36	1
DSP48	208

## See Also

## Related Examples

- “NR HDL Cell Search and MIB Recovery MATLAB Reference” on page 5-14

## LTE HDL Cell Search

This example shows the design of a LTE cell search and selection system optimized for HDL code generation and hardware implementation.

### Introduction

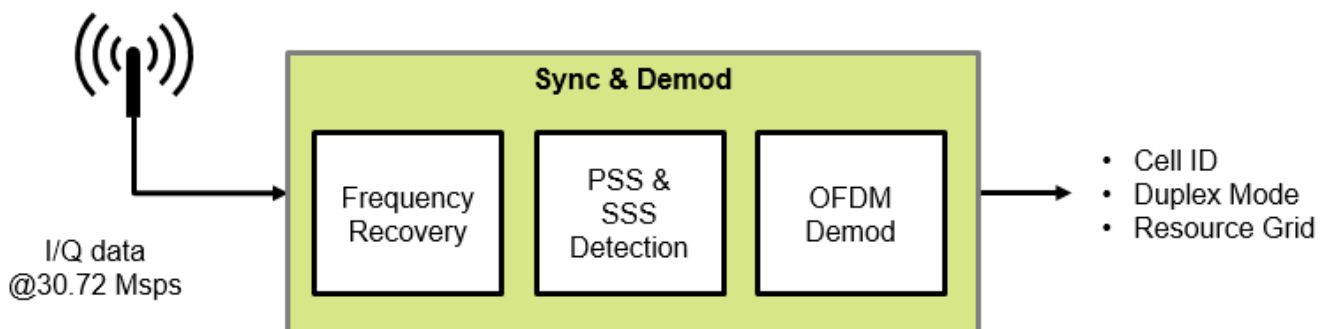
Cell search and selection is the first step taken by User Equipment (UE) in attempting to gain access to an LTE network. The cell search and selection procedure involves detecting candidate eNodeB signals and then selecting one to synchronize to. This includes determining the chosen eNodeB's physical layer cell identity (cell ID) and duplex mode. Additionally, the UE acquires frequency and timing synchronization during this process. Once this procedure has been completed, the UE can demodulate the OFDM signal transmitted by the cell and recover its Master Information Block (MIB). A MIB Recovery model with HDL code generation capability, which reuses the cell search and selection functionality shown here, is presented in the "LTE HDL MIB Recovery" on page 5-80.

The functionality in the present example is based on the cell search functionality of the LTE Toolbox "Cell Search, MIB and SIB1 Recovery" (LTE Toolbox). However, the algorithms have been optimized for HDL code generation. LTE Toolbox was used extensively in the development of the present example. The HDL model described here performs the following functions:

- Frequency recovery
- Primary and secondary synchronization signal detection
- OFDM demodulation

The frequency recovery algorithm within the HDL model can only correct offsets fewer than  $\pm 7.5$  kHz. Large frequency offset recovery greater than  $\pm 7.5$  kHz is possible by driving the input and monitoring the outputs with an external controller. A demonstration of large frequency offset correction can be found in the "LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.

Once the model has completed the cell search and selection procedure, it outputs the cell ID, duplex mode and unequalized resource grid of the cell. This functionality is shown below. The model supports downlink signals with 15 kHz subcarrier spacing and normal cyclic prefix length. Frequency Division Duplex (FDD) and Time Division Duplex (TDD) modes are both supported. The duplex mode is automatically detected.





The LTE standard provides two *physical signals* to aid the cell search process. These are the Primary Synchronization Signal (PSS) and the Secondary Synchronization Signal (SSS). Refer to Appendix A for more information on LTE downlink synchronization signals.

### Example Structure

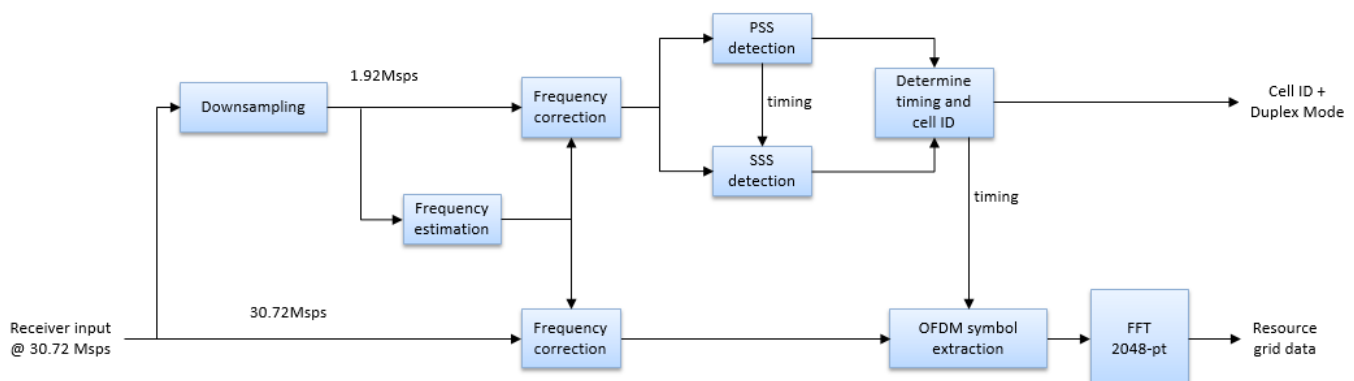
The model consists of 5 files:

- `ltehdlCellSearch.slx`: This is the top level of the model, and acts as a test bench for `ltehdlDownlinkSyncDemod.slx`.
- `ltehdlDownlinkSyncDemod.slx`: Model reference which implements the cell search, synchronization, and OFDM demodulation functionality.
- `ltehdlCellSearch_init.m`: MATLAB® script for generating stimulus.
- `ltehdlCellSearch_analyze.m`: MATLAB script for analyzing output and displaying plots at the end of the simulation.
- `ltehdlCellSearchTools.m`: MATLAB class containing helper methods for analyzing and plotting results.

**Note:** `ltehdlDownlinkSyncDemod.slx` does not appear in the example working folder as it is shared with other examples. The file is on the MATLAB path and can be opened by entering `ltehdlDownlinkSyncDemod` at the MATLAB command line.

### Model Architecture

The structure of the cell search and selection subsystem is shown below. The input is complex 16-bit data sampled at 30.72 Msps. The signal is passed to two signal processing data paths; one at 1.92 Msps and one at 30.72 Msps. Frequency recovery and PSS detection are performed on the 1.92 Msps data path. This sampling rate is used for two reasons. First, the cell bandwidth is not known at this stage therefore the smallest LTE bandwidth of 1.4 MHz is assumed for frequency recovery. This approach works irrespective of the actual cell bandwidth. Second, the PSS and SSS only occupy the six central resource blocks (1.4 MHz). Therefore, detection can be performed effectively at 1.92 Msps and resource sharing techniques can be used to optimize the hardware implementation.



The following steps describe the receiver operation.

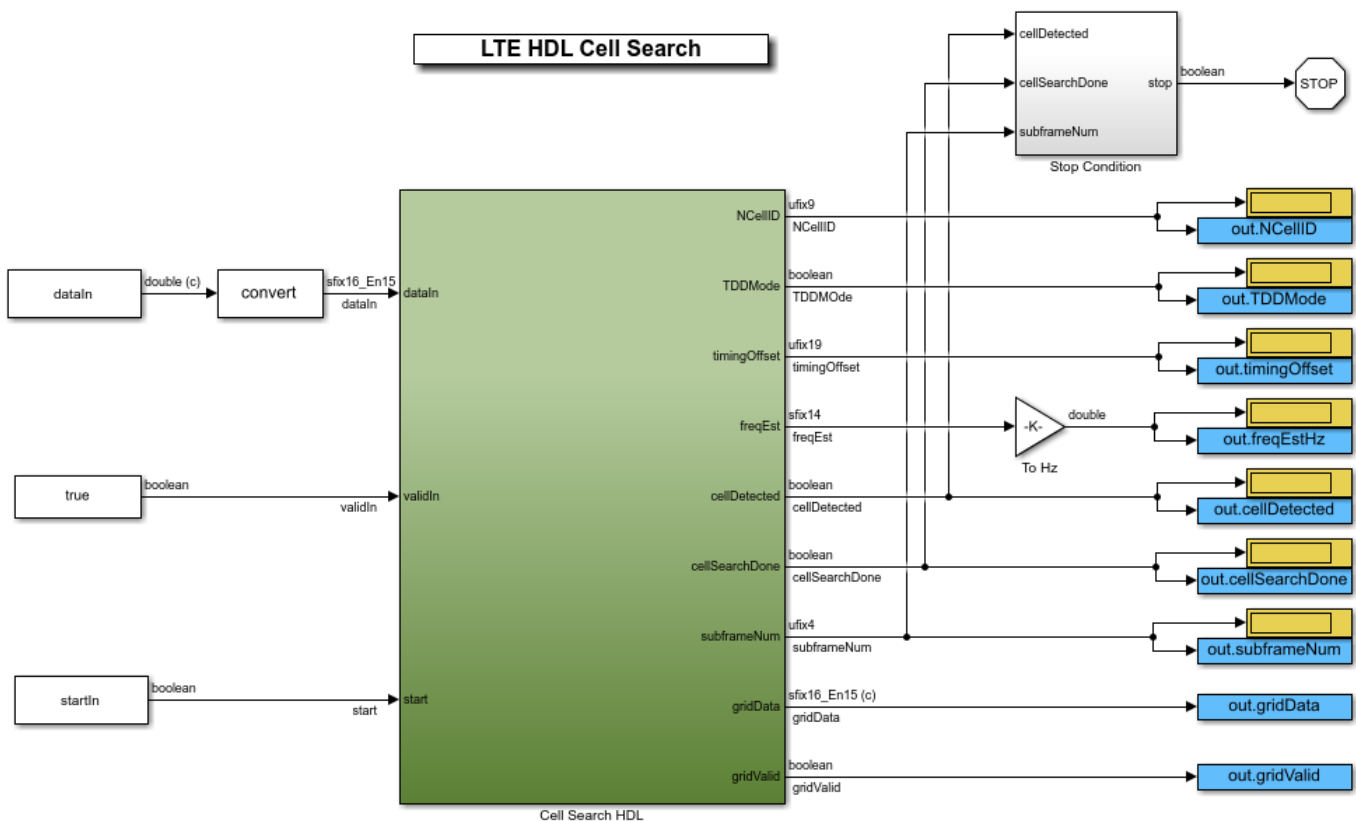
- 1 The frequency estimation block estimates the frequency offset over a 10 ms period.

- 2 The frequency correction blocks are then activated on both the 1.92 Msp/s and 30.72 Msp/s sample streams.
- 3 PSS detection begins immediately after the frequency estimation stage has been completed.
- 4 SSS detection begins when PSS detection detects a valid PSS signal. If a valid SSS is found, this means that a valid cell has been detected and the duplex mode is now known.
- 5 The cell ID and frame start position are computed.
- 6 On the next frame boundary, the receiver starts to extract OFDM symbols from the 30.72 Msp/s sample stream. Each symbol is passed through a 2048-point FFT to perform OFDM demodulation.

Appendix B provides more details of the cell search and selection algorithm used in this example.

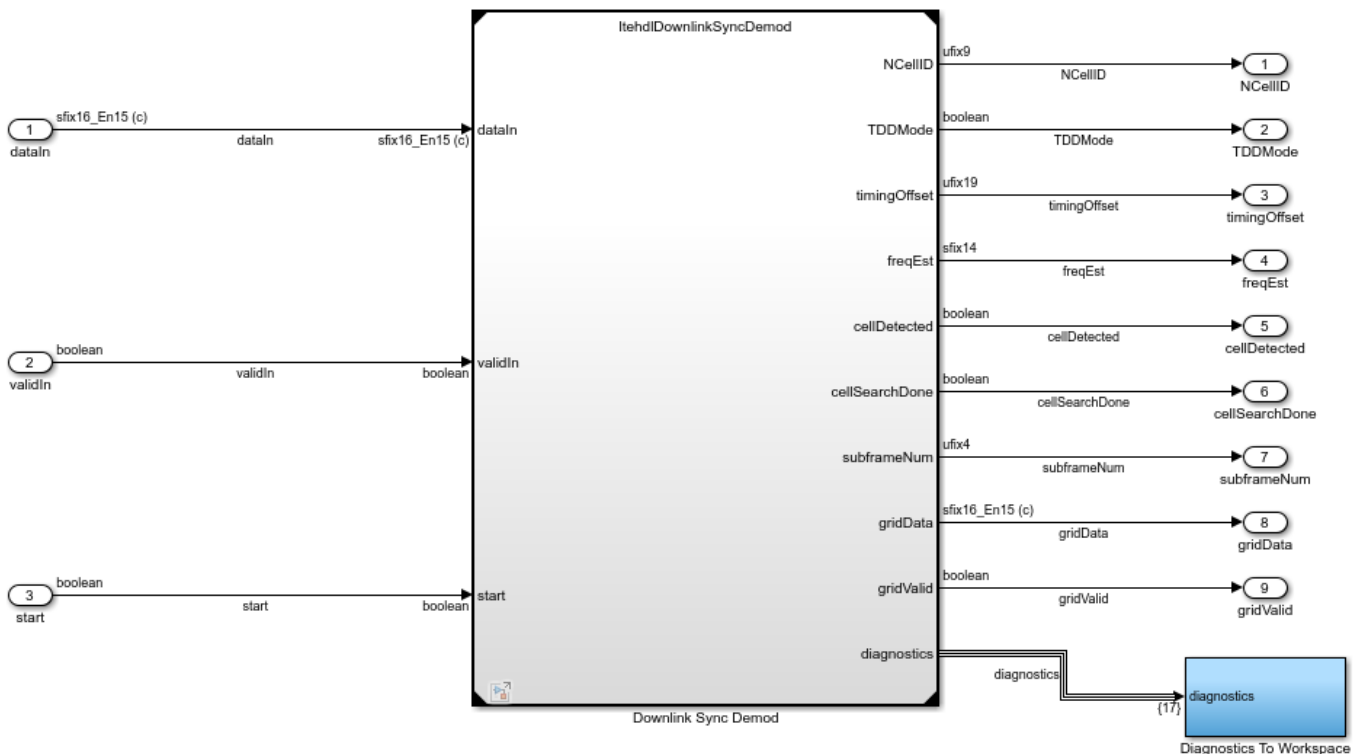
### Cell Search Simulink Model

The top level of `ltehdlCellSearch.slx` is shown below. This model references `ltehdlDownlinkSyncDemod.slx`. `ltehdlCellSearch_init.m` is called by the `InitFcn` callback and `ltehdlCellSearch_analyze.m` is called by the `StopFcn` callback. The model uses a **Stop** sink to terminate the simulation when either (i) the `subframeNum` output is 5 or (ii) `cellSearchDone` is asserted `true` and no cell is detected. HDL code can be generated for the **Cell Search HDL** subsystem.



The **Cell Search HDL** subsystem is primarily a wrapper for the **ltehdlDownlinkSyncDemod** model. It contains a **Model** block (**Downlink Sync Demod**) which references `ltehdlDownlinkSyncDemod.slx`, and a **Diagnostics To Workspace** subsystem, which logs all of

the diagnostic outputs. The diagnostic outputs are used by `ltehdlCellSearch_analyze.m` to generate plots showing the internal operation.



## Downlink Synchronization and Demodulation Model Reference

The `ltehdlDownlinkSyncDemod` model reference implements all of the cell search, synchronization and OFDM demodulation functionality. Appendix B details the cell search and selection algorithm implemented by this model. The top level of `ltehdlDownlinkSyncDemod` closely matches the architecture which was presented earlier.

Model Inputs:

- *dataIn*: Complex signed 16-bit data carrying the baseband input signal.
- *validIn*: Boolean, indicating whether *dataIn* is valid.
- *start*: Boolean. Assert this input `true` for one cycle at any time to initiate a cell search. This is referred to as a start command.

Model Outputs:

- *NCellID*: 9-bit cell ID of the detected eNodeB.
- *TDDMode*: Boolean, indicating the duplex mode of the detected cell: `false` for FDD, `true` for TDD.
- *timingOffset*: 19-bit timing offset. Indicates the number of samples from the first sample to enter the receiver to the first sample of the first full frame, from 0 to 307199.
- *freqEst*: 14-bit signed frequency offset estimate. Multiply this output by  $15e3 / 2^{14}$  in order to convert to Hz as shown in the `LTEHDLCellSearch` model.

- *cellDetected*: Boolean, indicating that a cell has been found.
- *cellSearchDone*: Boolean, indicating that the cell search has completed. If a cell is found, *cellDetected* and *cellSearchDone* will be asserted true at the same time. If no cell is found, *cellDetected* will remain false and *cellSearchDone* will be asserted true within 100 ms of the start command being issued. The time taken for *cellSearchDone* to be asserted depends on how many attempts are taken to detect PSS and SSS. See Appendix B for more details.
- *subframeNum*: 4-bit unsigned integer. Indicates which subframe is currently being passed out of the *gridData* port, from 0 to 9.
- *gridData*: 16-bit data carrying the demodulated resource grid.
- *gridValid*: Boolean, indicating whether *gridData* is valid.
- *diagnostics*: Bus signal, carrying various diagnostic outputs.

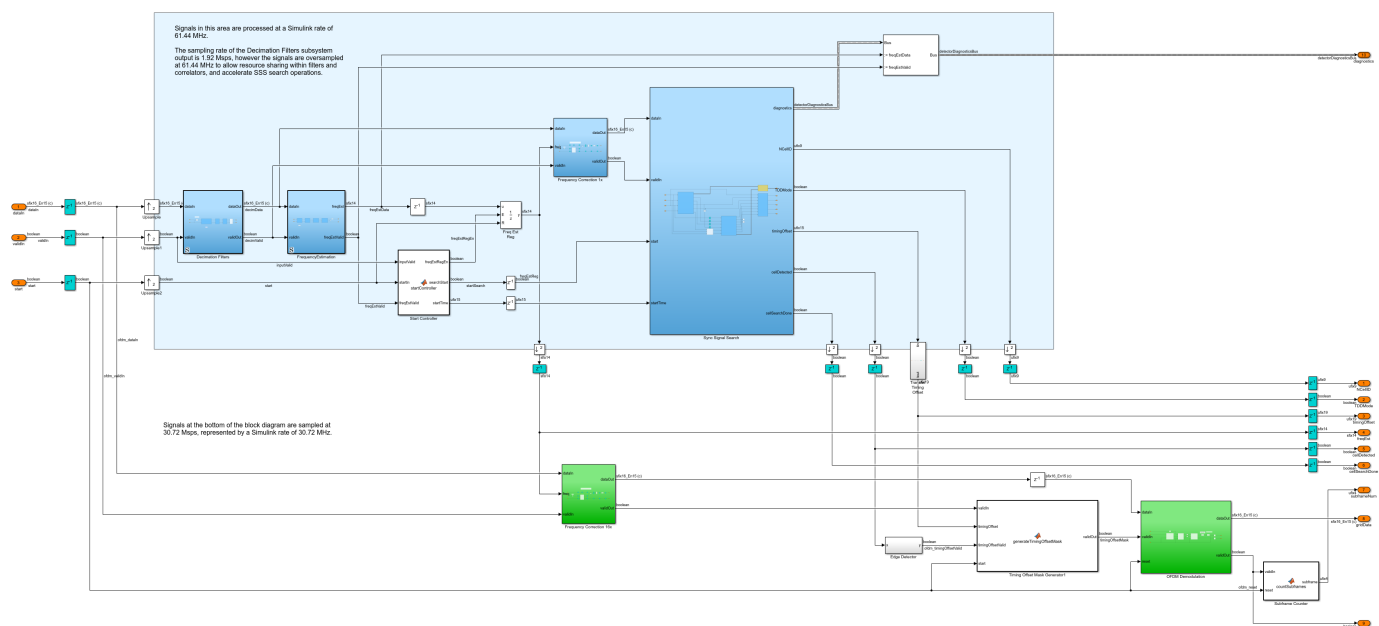
**ltehdldownlinkSyncDemod** uses two Wireless HDL Toolbox™ example functions during initialization: `ltehdldefineReceiverBuses` and `ltehdldownlinkSyncDemodConstants`. `ltehdldefineReceiverBuses` is shared with other Wireless HDL Toolbox examples, and defines a set of Simulink buses. This function is called in the `InitFcn` of **ltehdldownlinkSyncDemod**. Only the `detectorDiagnosticsBus` output of the function is used here. The bus object is stored in the Base Workspace, making it available to both the **ltehdldownlinkSyncDemod** and **ltehdlCellSearch** models.

```
[~,~,~,~,detectorDiagnosticsBus] = ltehdldefineReceiverBuses();
```

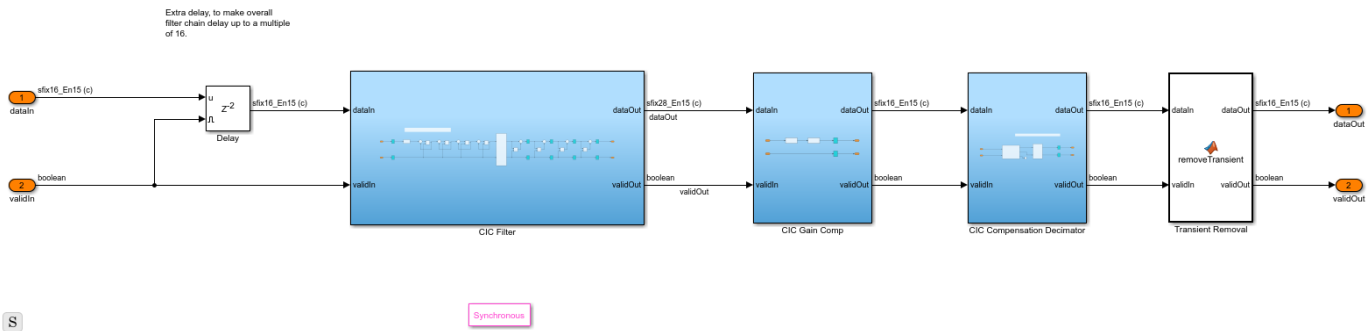
The model relies on precomputed constants and lookup tables stored in a structure called `cellDetectorConfig`. This structure is generated by the `ltehdldownlinkSyncDemodConstants` function and is only used inside the **ltehdldownlinkSyncDemod** model reference. Therefore, it is defined in the Model Workspace rather than the Base Workspace. Use the Model Explorer to view the Model Workspace, which contains the following initialization code.

```
cellDetectorConfig = ltehdldownlinkSyncDemodConstants(30.72e6);
```

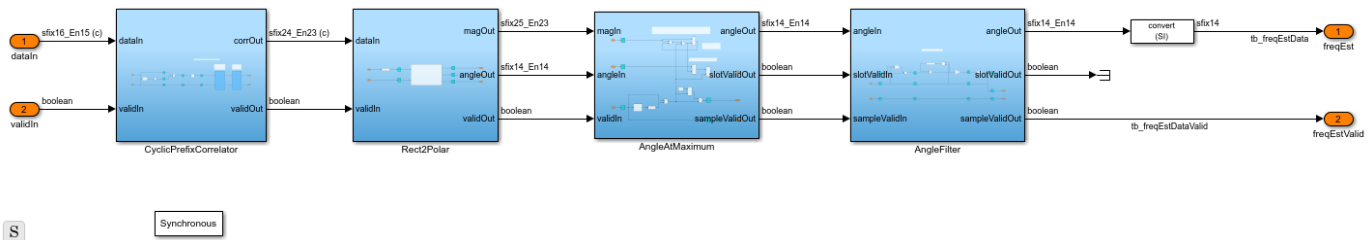
The internal structure of **ltehdldownlinkSyncDemod** is shown.



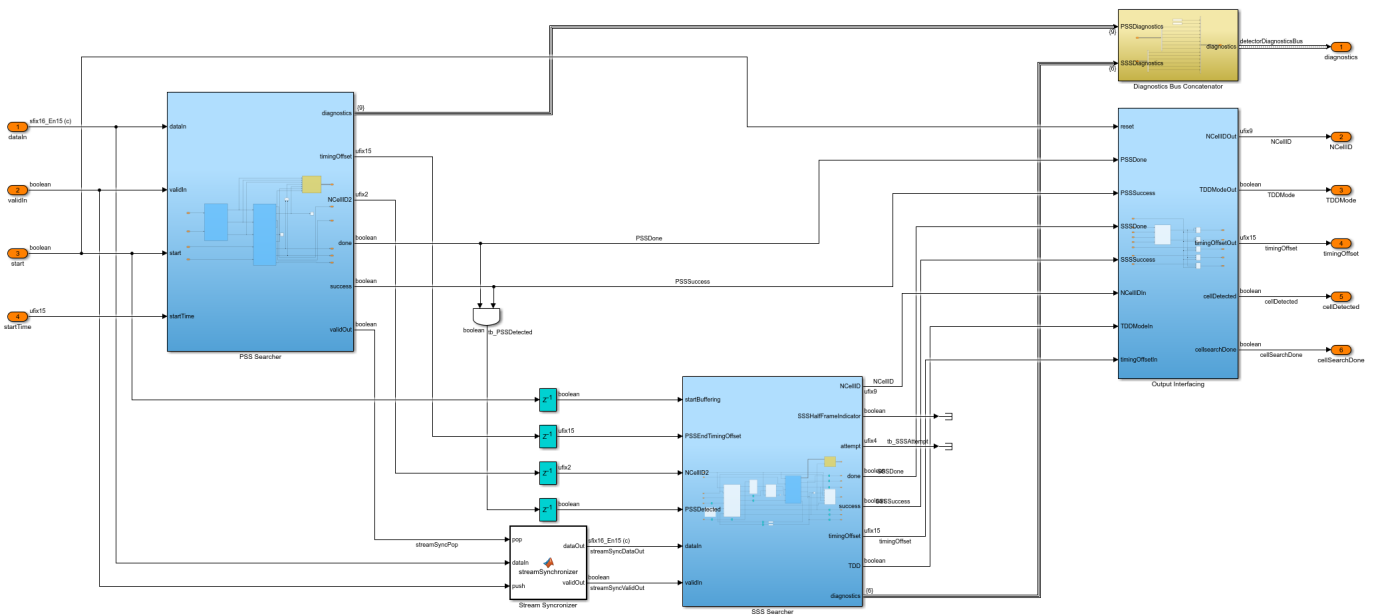
The **Decimation Filters** subsystem resamples the input data from 30.72 Msps to 1.92 Msps. It consists of CIC decimation, CIC gain compensation, CIC droop compensation, and transient removal. The filter chain is designed to have a group delay which is equal to an integer number of samples at 1.92 Msps. The **Transient Removal** block removes the initial transient due to this group delay from the sample stream. This is important because the frame timing offset is measured on the 1.92 Msps stream and then used to recover timing on the 30.72 Msps stream. Removing the initial transient from the decimation filter chain simplifies the logic which transfers the timing information.



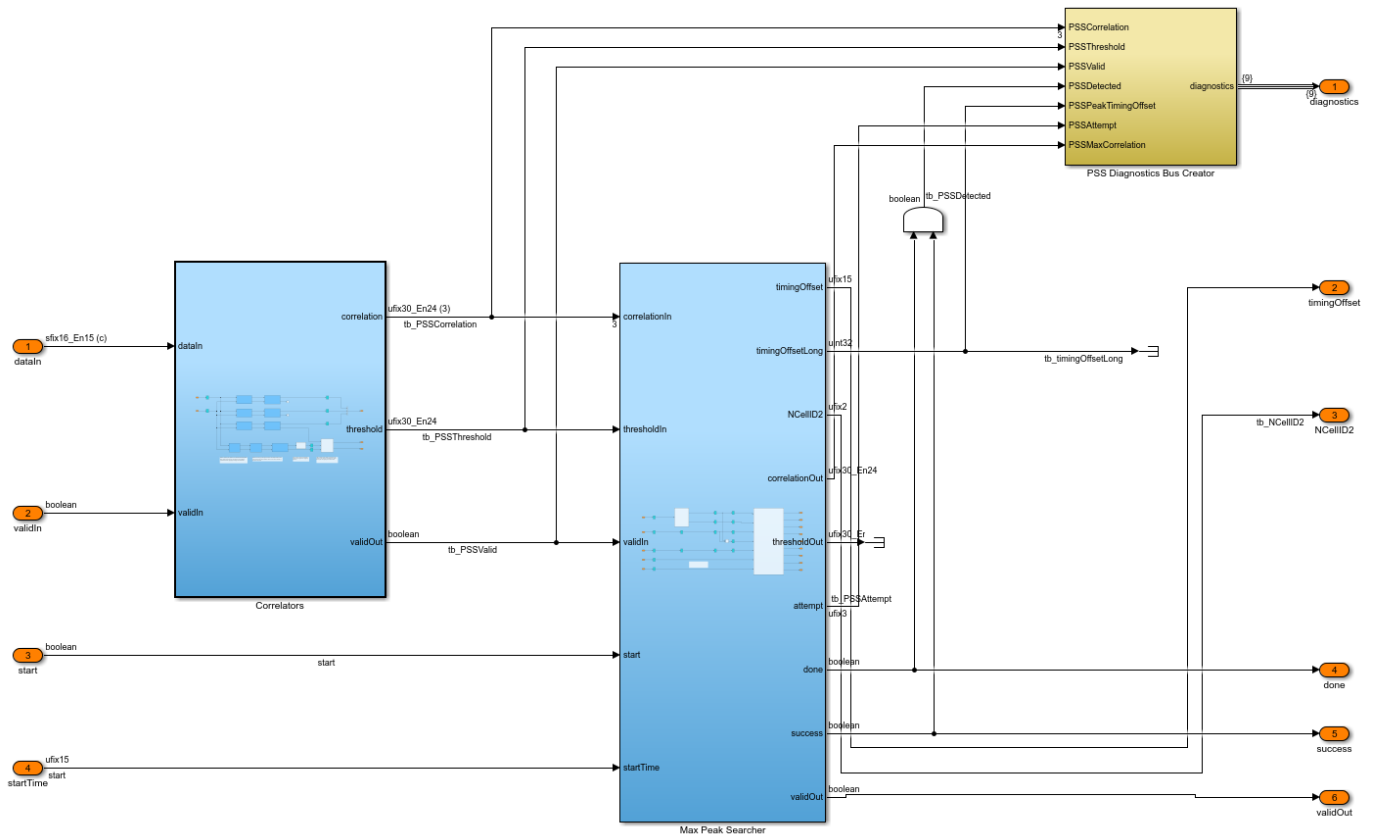
The **FrequencyEstimation** subsystem uses the cyclic prefix to estimate the frequency offset of the incoming signal. Every 960 samples, the **AngleAtMaximum** subsystem selects the strongest correlation peak and records its phase angle. The **AngleFilter** subsystem implements an averaging filter with a window duration of 10 ms. The resulting phase angle serves as a frequency estimate. Appendix B provides more information on how the cyclic prefix can be used to estimate the frequency offset.



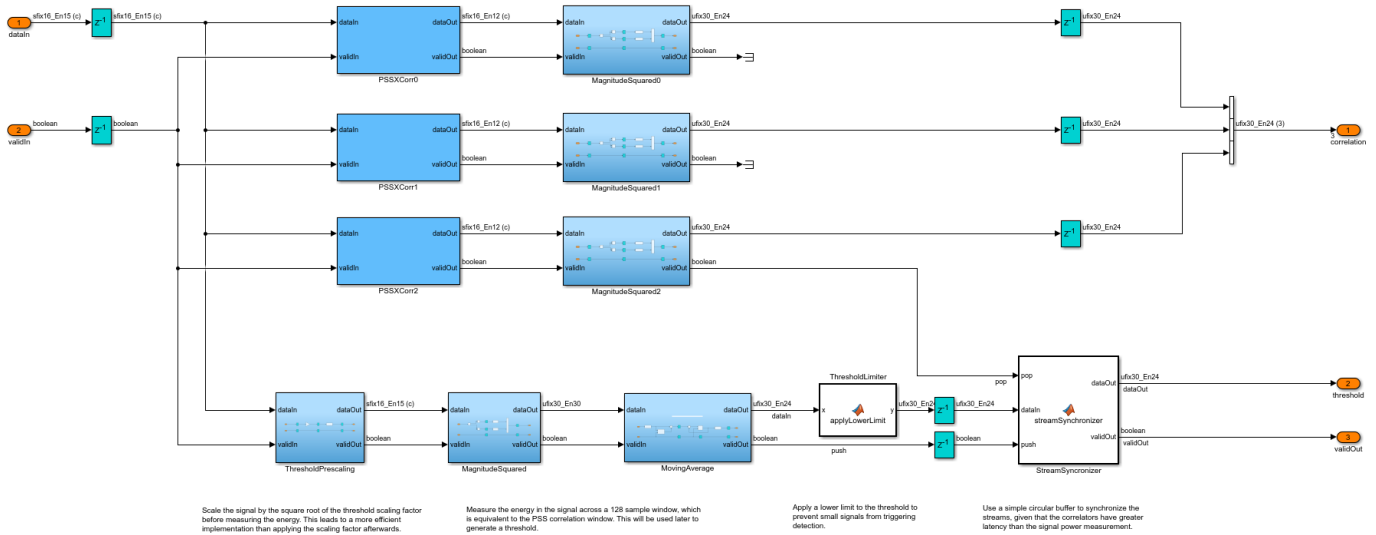
The **Sync Signal Search** subsystem implements PSS and SSS detection. Timing is crucial in this part of the design, because the **SSS Searcher** uses the frame timing information from the **PSS Searcher** to identify SSS search locations. The **PSS Searcher** provides a **validOut** signal which is used by the **Stream Synchronizer** block to delay the input stream and compensate for the **PSS Searcher** pipeline latency. Synchronizing the input stream to the **PSS Searcher** outputs simplifies the design of the **SSS Searcher**.



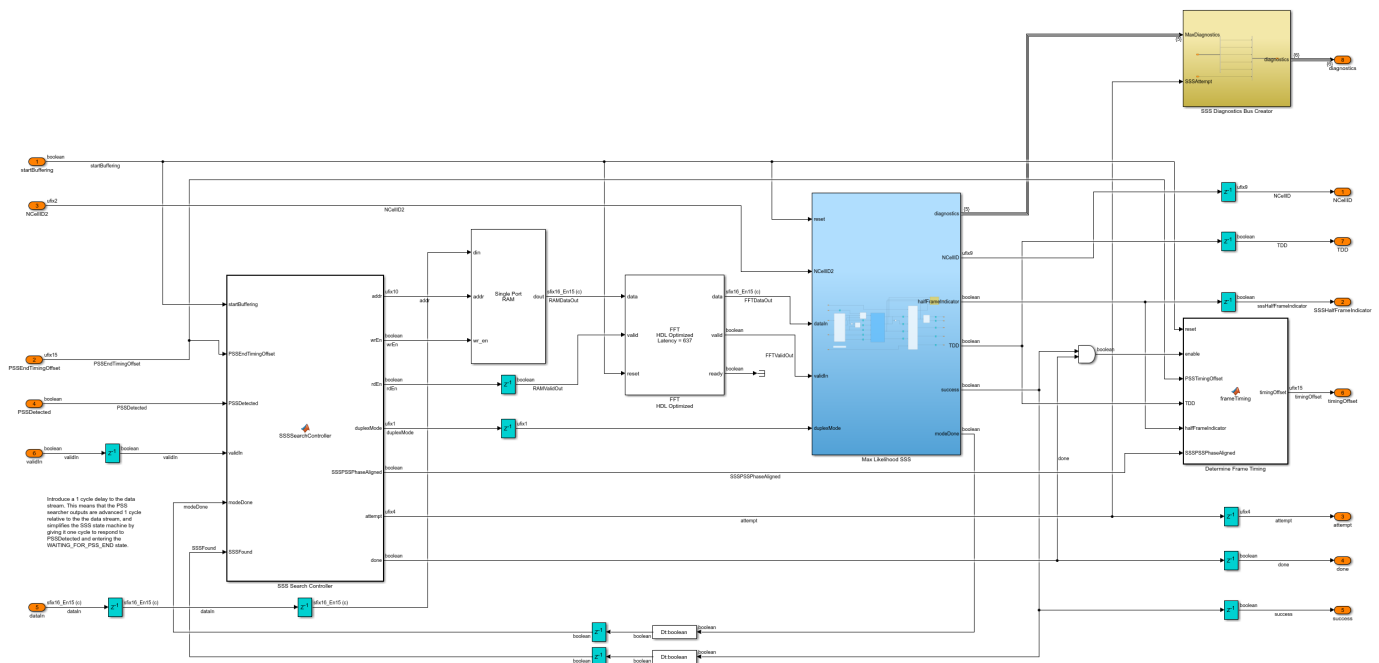
The **PSS Searcher** is made up of two subsystems: the **Correlators** and the **Max Peak Searcher**. Together, these subsystems implement the PSS search algorithm described in Appendix B.



The **Correlators** subsystem contains a matched filter for each of the three PSS sequences, and a set of subsystems for determining the threshold. A lower limit is applied to the threshold to prevent small signals triggering false alarms. The PSS correlators and the threshold generation logic have different pipeline delays, therefore, a stream synchronizer is used to re-align their outputs.



Once a cell search is underway, the **SSS Searcher** continually stores samples in a circular buffer. Once PSS is detected, it continues to load samples into the buffer until the SSS search location has been reached and stored. The SSS search location is computed from the PSS timing information provided by the **PSSEndTimingOffset** signal. Next, the FDD location samples are read from the buffer, passed through a 128-point FFT, and the **Max Likelihood SSS** subsystem computes the correlation metrics and threshold. The same operation is then applied to the TDD location samples. The **Max Likelihood SSS** subsystem chooses the maximum correlation metric which exceeded the threshold and determines the duplex mode and frame timing. Finally, the frame timing offset is computed.



## Initialization and Analysis Scripts

### Initialization Script

ltehdlCellSearch\_init.m is called in the InitFcn callback of ltehdlCellSearch.slx. Stimulus can either be loaded from a file containing a captured off-the-air waveform, or generated with LTE Toolbox.

```
% ltehdlCellSearch model initialization script
% Generates workspace variables needed by the ltehdlCellSearch model.
```

```
SamplingRate = 30.72e6;
simParams.Ts = 1/SamplingRate;
```

```
% Choose to load a captured off-the-air waveform from a file,
% or generate a test waveform with LTE Toolbox.
loadfromfile = true;
```

```
if loadfromfile
    % Load captured off-the-air waveform.
    load('eNodeBWaveform.mat');
    dataIn = resample(rxWaveform,SamplingRate,fs);
else
    % Generate a test waveform with LTE Toolbox.
    dataIn = hGeneratedDLRXWaveform();
end
```

```
% Scale signal level to be in the range -1 to +1.
dataIn = 0.95 * dataIn / max(abs(dataIn));
```

```
% Start 1 subframe into the waveform (chosen arbitrarily).
startIn = false(length(dataIn),1);
```



```

startIn(1e-3*SamplingRate) = true;

% Configure PSS and SSS attempts
PSSAttempts = 2;
SSSAtempts = 4;

% Determine stop time.
simParams.stopTime = (length(dataIn)-1)/SamplingRate;

```

### Analysis Script

`ltehdlCellSearch_analyze.m` is called in the `StopFcn` callback of `ltehdlCellSearch.slx`. This script relies heavily on `ltehdlCellSearchTools.m` to analyze the model output and display the plots.

```

% ltehdlCellSearch model analysis script
% Post-processes model outputs and generates plots.

% Check if any simulation output exists to analyze.
if exist('out','var') && ~isempty(out.PSSDetected)

    % Post-process the model output to extract key cell parameters,
    % diagnostics and signals.

    [signals, report] = ltehdlCellSearchTools.processOutput(dataIn,startIn,out);

    % Plot results

    ltehdlCellSearchTools.figure('Input waveform and search stages'); clf;
    ltehdlCellSearchTools.plotSearchStates(signals,report);

    ltehdlCellSearchTools.figure('Frequency estimation'); clf;
    ltehdlCellSearchTools.plotFrequencyEstimate(signals,report);

    ltehdlCellSearchTools.figure('PSS search'); clf;
    ltehdlCellSearchTools.plotPSSCorrelation(signals,report);

    ltehdlCellSearchTools.figure('SSS search');
    ltehdlCellSearchTools.plotSSSCorrelation(signals,report);

end

```

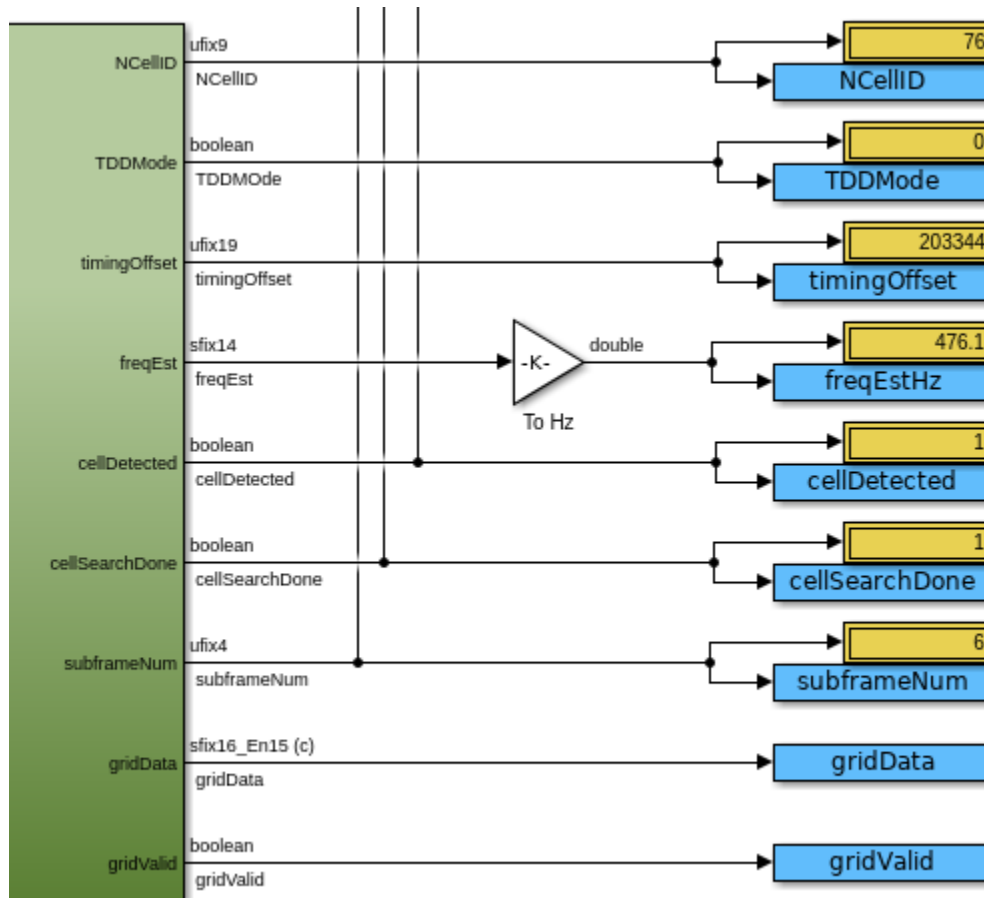
### Analysis Tools Class

This class contains helper functions for analyzing and plotting model output. Refer to `ltehdlCellSearchTools.m` for more information.

### Simulation Output and Analysis

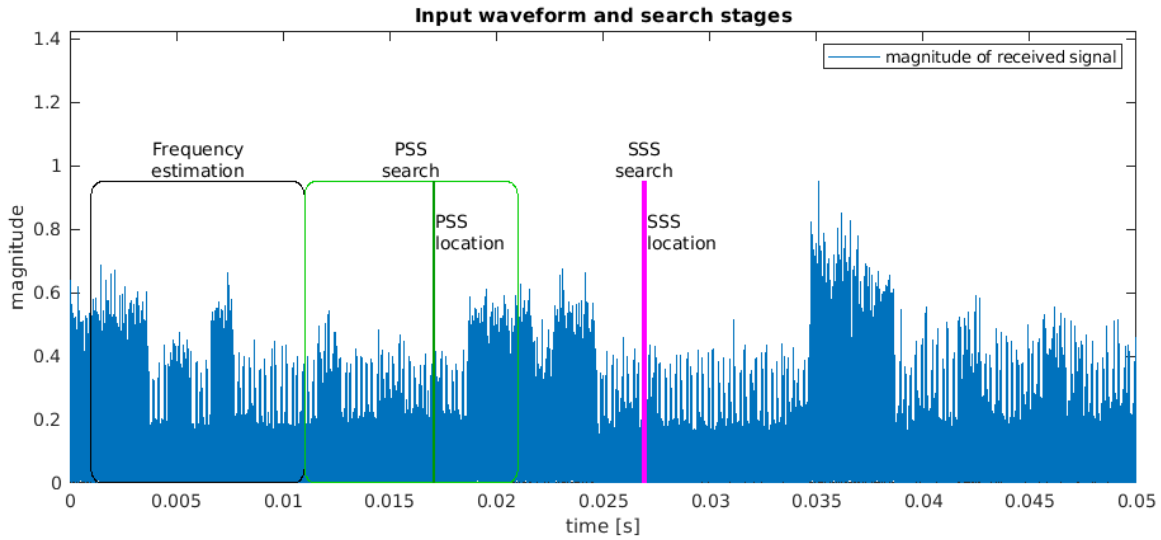
To execute the simulation, use the run button in the **ltehdlCellSearch** model. Simulink will automatically call **ltehdlCellSearch\_init** and **ltehdlCellSearch\_analyze** via the **InitFcn** and **StopFcn** callbacks respectively. Note that it will take a while to build the **ltehdlDownlinkSyncDemod** model reference on the first run. The simulation generates two main types of output: (i) **Display** blocks at the top level of the **ltehdlCellSearch** block diagram show key detection parameters, and (ii) four plots are generated at the end of the simulation.

The *NCellID*, *TDDMode*, *timingOffset*, *freqEst*, *cellDetected*, and *cellSearchDone* outputs all have associated **Display** blocks. Their values are shown below at the end of a simulation which used the captured off-the-air waveform (eNodeBWaveform.mat) as stimulus.

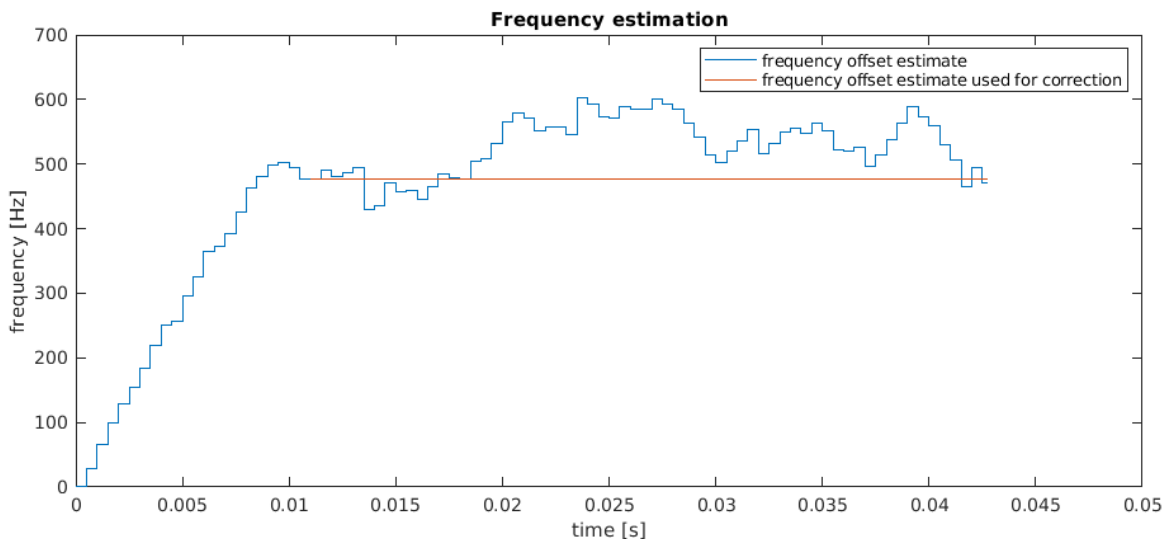


The **Input waveform and search stages** plot shows:

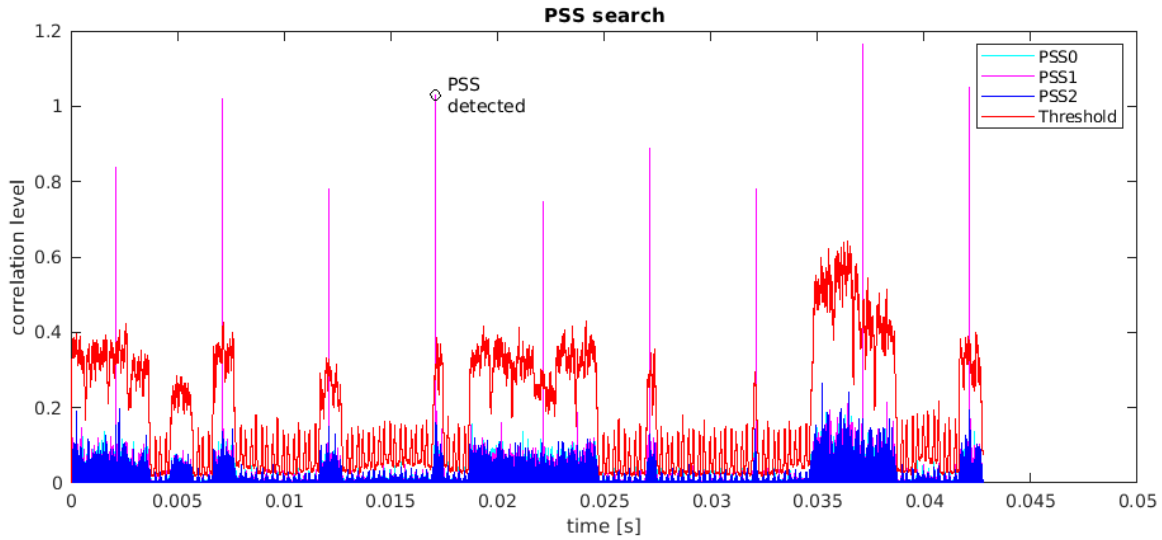
- The magnitude of the input waveform vs time.
- The time window during which frequency estimation occurs.
- The PSS search window for each attempt (one in this case) and the location of the detected PSS.
- The SSS search windows for TDD and FDD for each attempt (one in this case), and the location of the detected SSS.



The **Frequency estimation** plot shows the output of the frequency estimator vs time. At the end of the 10 ms frequency estimation time window, the frequency estimate is loaded into a register and used to correct the frequency offset. This value is also shown on the plot. In this case the frequency offset is just below 500 Hz, which is well within the -7.5 kHz to +7.5 kHz operating range of the frequency recovery algorithm.



The cell ID is made up of two components, NCellID1 and NCellID2, where NCellID1 is the SSS sequence number, and NCellID2 is the PSS sequence number (See Appendix A). The **PSS search** plot shows all three PSS correlator outputs, and the PSS threshold. PSS was detected approximately 17 ms into the waveform on PSS #1, therefore NCellID2 = 1.

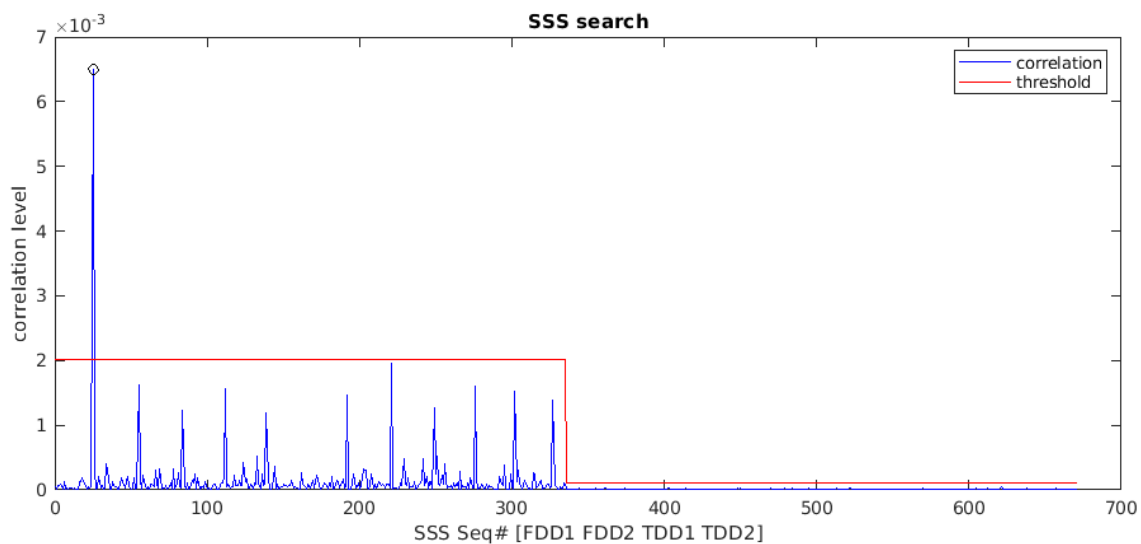


The **SSS search** plot shows the correlation metrics for the successful SSS detection attempt, and the SSS threshold. As previously discussed, the SSS detection algorithm determines the duplex mode and half frame position, as well as the cell ID. As a result,  $4 \times 168 = 672$  correlation metrics are computed during each attempt. The correlation metrics are shown in the following order along the x-axis:

- FDD1: metrics at the FDD location for SSS sequences corresponding to 1st half frame
- FDD2: metrics at the FDD location for SSS sequences corresponding to 2st half frame
- TDD1: metrics at the TDD location for SSS sequences corresponding to 1st half frame
- TDD2: metrics at the TDD location for SSS sequences corresponding to 2st half frame

SSS was detected in the FDD location for SSS sequence corresponding to the 1st half frame. The SSS sequence number is 25 therefore  $N_{CellID1} = 25$ . The final cell ID is therefore:

$$N_{CellID} = 3 \times N_{CellID1} + N_{CellID2} = 76.$$



## HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL testbench for the **Cell Search HDL** subsystem. Note that testbench generation can take a while due to the length of the tests vectors that are generated.

The **Cell Search HDL** subsystem was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below. The design met timing with a clock frequency of 200 MHz.

Resource	Usage
Slice Registers	44658
Slice LUTs	20271
RAMB18	25
RAMB36	11
DSP48	110

## Appendix A - LTE Downlink Synchronization Signals

LTE provides two *physical signals* to aid the cell search and synchronization process. These are the Primary Synchronization Signal (PSS) and the Secondary Synchronization Signal (SSS).

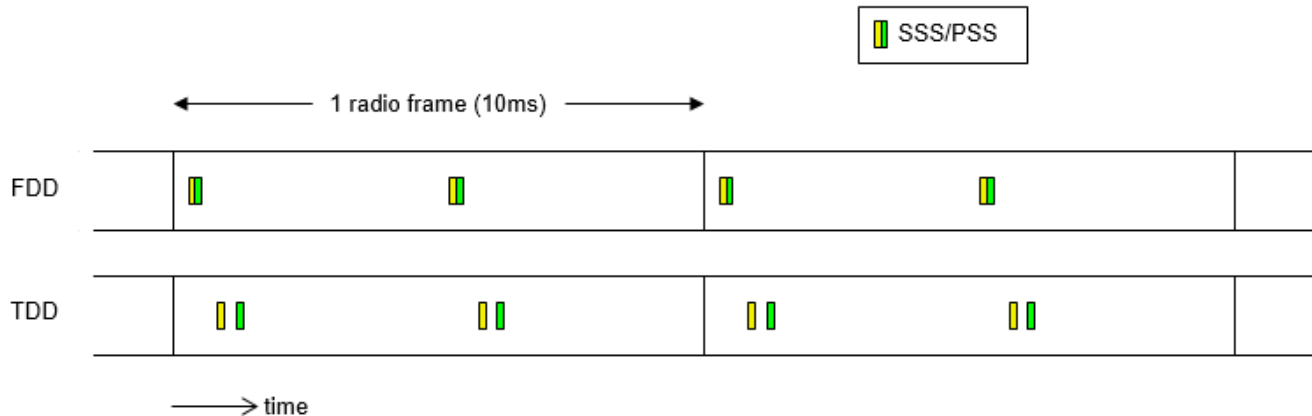
The cell ID of the eNodeB is encoded in the PSS and SSS. The duplex mode, cyclic prefix length, and frame timing can be determined from their positions within the received signal. The PSS and SSS are transmitted twice every frame. There are 3 possible PSS sequences, and the eNodeB transmits the same PSS every half frame. For each PSS, there are 168 possible SSS sequences in the first half of the frame and 168 different possible SSS sequences in the second half of the frame. This means that once a SSS has been detected, the receiver knows if it is in the first or second half of a frame. The PSS and SSS sequences depend on the cell ID, therefore, there are  $3 * 168 = 504$  possible cell IDs. The cell ID is

$$N_{\text{CellID}} = 3 * N_{\text{CellID1}} + N_{\text{CellID2}}$$

where  $N_{\text{CellID2}}$  is the PSS sequence number from 0 to 2, and  $N_{\text{CellID1}}$  is the SSS sequence number from 0 to 167. Each instance of the PSS occupies the central 62 subcarriers of one OFDM symbol, as does each instance of the SSS. For normal cyclic prefix mode the locations of the PSS and SSS signals are follows:

- FDD Mode: PSS is in symbol 6 of subframe 0, SSS is in symbol 5 of subframe 0
- TDD Mode: PSS is in symbol 2 of subframe 1, SSS is in symbol 13 of subframe 0

There are 14 symbols in each subframe, numbered from 0 to 13. Therefore, in FDD mode, the PSS is transmitted one OFDM symbol after the SSS, whereas in TDD mode the PSS is transmitted three OFDM symbols after the SSS. This difference in relative timing allows the receiver to discriminate between the two duplex modes. The positions of PSS and SSS within radio frames in FDD and TDD mode are illustrated below.



For more details see “Synchronization Signals (PSS and SSS)” (LTE Toolbox).

### Appendix B - Cell Search and Selection Algorithm

This section describes the algorithm used by the model to detect eNodeB signals. The algorithm is designed to cope with real world conditions such as frequency offsets, noise and interference, and variation in the SNR of the PSS and SSS over time. To detect eNodeB in the presence of such conditions, the example uses three techniques:

- 1 Frequency recovery is applied prior to PSS and SSS detection.
- 2 Dynamic thresholds are used to validate the PSS and SSS correlation metrics and minimize the probability of false alarm.
- 3 Multiple attempts are made to detect the PSS and SSS; for example, if none of the correlation metrics for a specific instance of the SSS exceed the threshold, the detector will wait half a frame and try again, up to a predefined number of attempts.

### Frequency Recovery

Frequency recovery is performed by utilizing the time domain structure of the received signal. In LTE (as with other OFDM based systems), each symbol consists of a *useful part* and a *Cyclic Prefix* (CP). The CP is generated by copying a small slice from the end of the symbol and prepending it to the start of the symbol. This can be exploited in a receiver by multiplying the received signal with the complex conjugate of a delayed version of itself, and then integrating across the CP duration, where the delay is the duration of the useful part. In effect, the received signal is cross-correlated with a delayed version of itself. The magnitude of the integrator output has peaks at symbol boundaries. The phase angle of the signal at these peaks is related to the frequency offset. This approach is used in the present example, and combined with additional averaging, to estimate the frequency offset. The algorithm can detect frequency offsets from -7.5 kHz to +7.5 kHz.

### PSS Detection

PSS detection is performed by continuously cross-correlating the received signal with all three possible PSS sequences in the time domain. In addition, the energy of the signal within the span of the correlators is computed on each time step and then scaled to generate a threshold. The PSS detection algorithm aims to pick the strongest cell by picking the maximum PSS correlation metric within a 10 ms time window. The following pseudocode describes the search algorithm:

```

initialize position of first 10 ms search window

for k = 1 to 4 (number of PSS attempts)

    find correlation levels which exceed the threshold
    if any correlation levels exceed the threshold
        find the max correlation level of those which exceed the threshold
        PSS detected: break loop and start SSS search
    else
        PSS not detected: move search window to next 10ms period
    end
end
end

```

### SSS Detection

Once PSS is located, the detector can narrow down the position of the SSS to two possible locations; one for FDD and one for TDD. The SSS correlation metrics are computed in the frequency domain, by evaluating the dot product of the sequence. The following algorithm is used to search for and select an SSS sequence.

```

initialize SSS search window

for k = 1 to 8 (number of SSS attempts)

    for each duplex mode in [FDD, TDD]
        extract 128 point search window for current duplex mode
        compute FFT and extract SSS subcarriers
        compute correlation metrics for SSS sequences corresponding to 1st half frame
        compute correlation metrics for SSS sequences corresponding to 2nd half frame
        compute signal energy-based threshold
    end

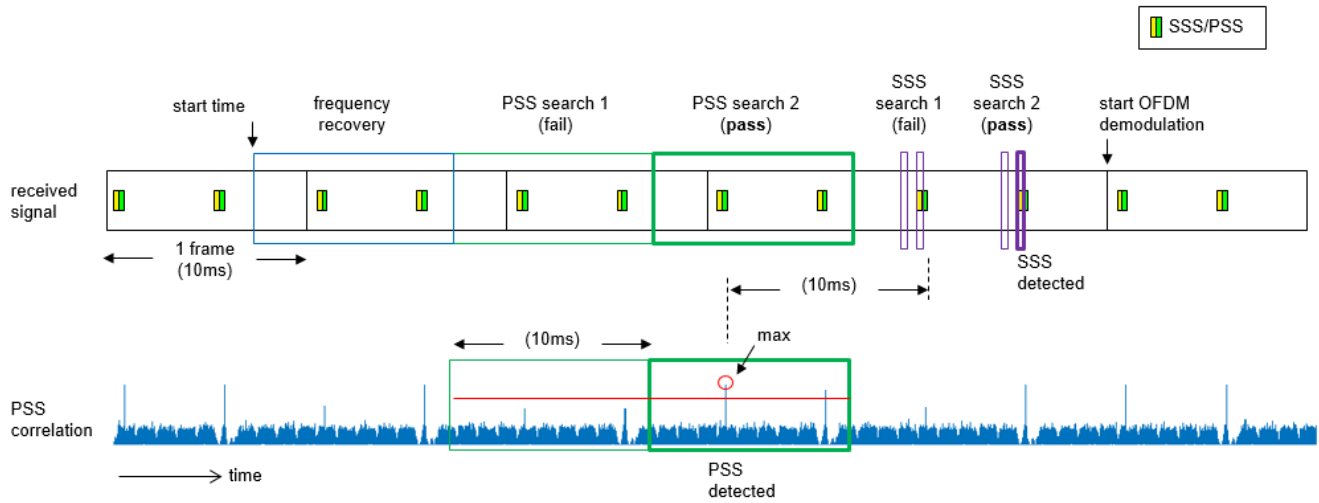
    discard correlation metrics which do not exceed the threshold
    if any metrics exceeded the threshold
        pick maximum correlation metric from surviving metrics
        SSS detected: break loop and proceed to next processing stage
    else
        SSS not detected: move SSS search window later by half a frame
    end
end

end

```

### Cell Search Illustration

The cell search algorithm is shown below for a scenario where PSS and SSS each take 2 attempts to detect valid signals. The figure also shows the frequency recovery stage. Initially, the receiver has no knowledge of the received signal frame timing. In the Simulink model (and on hardware), a *start* input is used to trigger the detection process. The receiver begins by measuring the frequency offset, which takes 10 ms. Next, the first 10 ms PSS search takes place. In this case, no PSS is detected, therefore a second PSS search is initiated. This time PSS is detected. The first SSS search takes place just short of 10 ms after the location of the detected PSS, avoiding the need to buffer significant amounts of data, and making the algorithm hardware friendly. As shown, SSS also takes two attempts in this case. From the location of the detected SSS, the receiver knows the duplex model (FDD in this case) and the frame timing.



## References

1. 3GPP TS 36.214 "Physical layer"

## See Also

### Related Examples

- "LTE HDL MIB Recovery" on page 5-80
- "LTE HDL SIB1 Recovery" on page 5-63
- "LTE HDL PBCH Transmitter" on page 5-91



## LTE HDL SIB1 Recovery

This example shows the design of an HDL optimized receiver that can recover the first System Information Block (SIB1) from an LTE downlink signal.

### Introduction

This design builds on the “LTE HDL MIB Recovery” on page 5-80, adding the processing required to decode SIB1. It is based on the LTE Toolbox™ “Cell Search, MIB and SIB1 Recovery” (LTE Toolbox).

In order to decode the SIB1 message, additional steps are required after the MIB (Master Information Block) has been decoded. This design adds functionality to locate and decode the PCFICH (Physical Control Format Indicator Channel), the PDCCH (Physical Downlink Control Channel), and the PDSCH (Physical Downlink Shared Channel). The extensible architecture used in the “LTE HDL MIB Recovery” on page 5-80 allows the design to be expanded, while reusing the core functionality of the MIB recovery implementation.

### Summary of SIB1 Processing Stages

The initial stages of SIB1 recovery are the same as for the “LTE HDL MIB Recovery” on page 5-80, composed of the cell search, PSS/SSS detection, OFDM demodulation, and MIB decoding. LTE signal detection, timing and frequency synchronization, and OFDM demodulation are performed on the received data, providing information on the subframe number, duplex mode, and cell ID of the received waveform. The received data is buffered into the grid subframe memory buffer and, once a complete subframe has been stored in the memory, the channel estimate is calculated. The channel estimate can then be used to equalize the grid as data is read out from the buffer. When subframe 0 has been stored in the buffer, and the channel estimate calculated, the Physical Broadcast Channel (PBCH) can then be retrieved from the grid, equalized, and decoded, recovering the MIB message.

The MIB message contains a number of parameters which are required to decode the subsequent channels. One of these parameters is the System Frame Number (SFN). The SFN is required to determine the location of the SIB1 message, since the SIB1 message is only sent in even numbered frames ( $\text{mod}(\text{SFN}, 2) = 0$ ). Hence, if the MIB message was decoded within an odd frame, the receiver must wait until the next even frame before attempting to decode the SIB1. When the receiver has decoded the MIB message, and has received subframe 5 of an even frame, an attempt at decoding the SIB1 can be made.

The MIB message also provides the NDLRB system parameter, indicating the Number of Downlink Resource Blocks used by the transmitter. For different NDLRB values (different bandwidths) the number of active subcarriers is different. Hence the NDLRB affects the indexing of the resource grid memory for each of the channels processed after the PBCH.

NDLRB is first used to calculate the Resource Elements (REs) allocated to the Physical Control Format Indicator Channel (PCFICH), and the corresponding symbols can be retrieved from the resource grid. The PCFICH Decoder then attempts to decode the CFI data using the symbols retrieved from the resource grid.

The CFI indicates the number of OFDM symbols allocated to the Physical Downlink Control Channel (PDCCH). The CFI, in conjunction with the MIB parameters NDLRB, PHICH Duration, and  $N_g$ , is used to calculate which Resource Elements (REs) are allocated to the PDCCH. These REs are requested from the grid, and passed to the PDCCH decoder. If the signal being decoded is using Time Division Duplexing (TDD) the PDCCH allocation varies based on the TDD configuration used. Because the TDD configuration is not known at this point, each of the TDD configurations that affect the PDCCH allocation are tried, until successfully decoding.

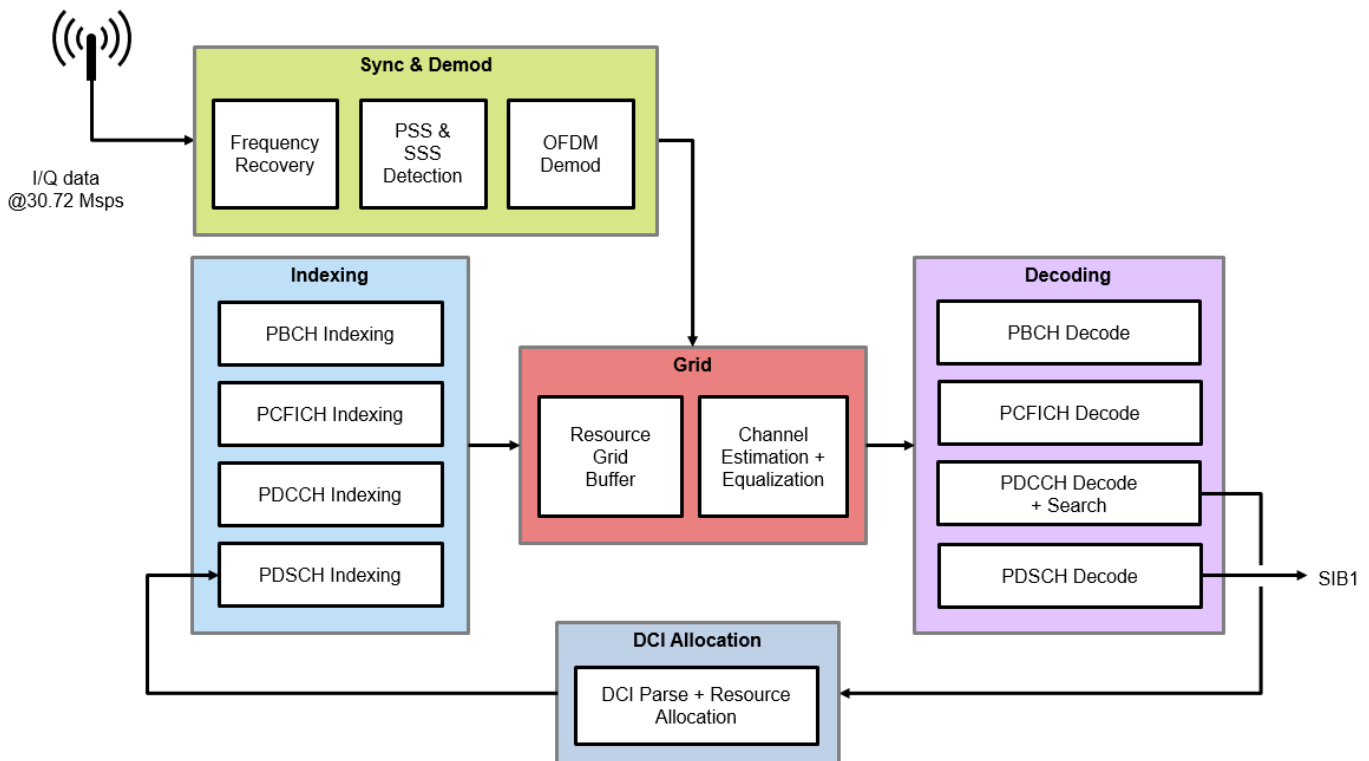
Once the PDCCH has been decoded, a blind search of the PDCCH common search space is conducted to find the DCI (Downlink Control Information) message for the SIB1. This DCI message has a CRC scrambled with the SI-RNTI (System Information Radio Network Temporary Identifier) and carries information about the allocation and encoding of the SIB1 message within the PDSCH. The search operation blindly attempts to decode DCI messages with a number of possible formats, from a number of candidates. If the signal being decoded is using TDD and a DCI message is not found during the search, then PDCCH decoding will be re-attempted for any untried TDD configurations.

Once located, the DCI message is parsed, giving the DCI allocation type, RIV, and Gap parameters required for the PDSCH resource allocation calculation. The Physical Resource Blocks (PRBs) allocated to the SIB1 message within the PDSCH can then be calculated. Parsing the DCI message also provides information on the transport block length and redundancy versions required to decode the PDSCH.

Using the PRB allocation information the REs allocated to the SIB1 message within the PDSCH can be calculated. The PDSCH decoding then processes the data retrieved from the resource grid. If decoding is error free the SIB1 message bits are returned.

### Architecture and Configuration

The architecture is designed to be extensible, allowing channel processing subsystems to be added, removed, or exchanged for alternative implementations. This extensibility is illustrated by the additions made to the MIB design to produce the SIB1 design. The core functionality is the same, with additional processing and control added for the three extra channels required to decode the SIB1.



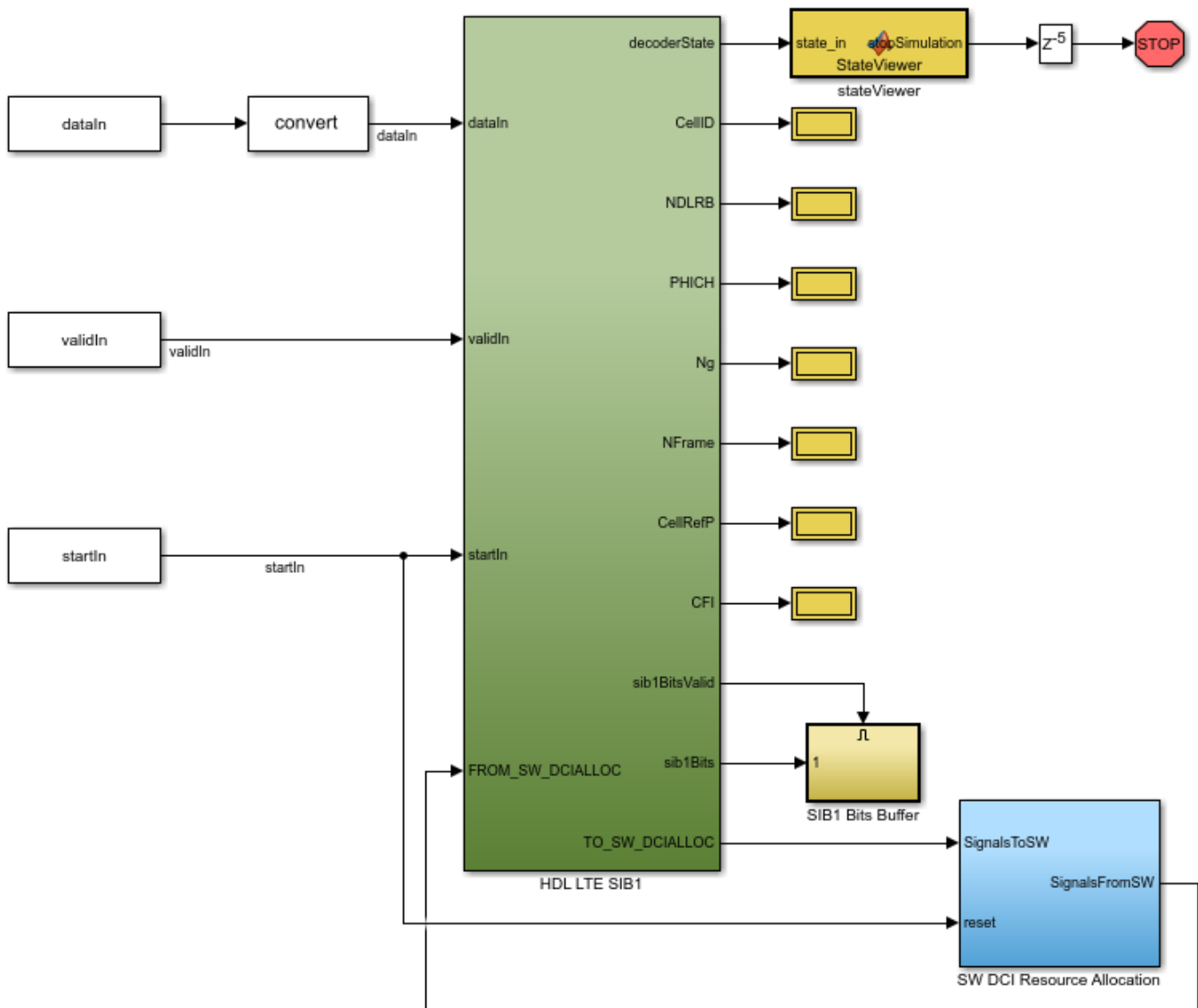
To allow reuse and sharing of the main subsystems of the model, the example uses “Model References”. Model referencing allows for unit testing of each of the subsystems, and for the models to be instantiated in multiple different examples. The LTE HDL Cell Search, LTE HDL MIB Recovery and LTE HDL SIB1 recovery all share reference models.

- Cell search, synchronization and OFDM demodulation perform initial stages of detecting a downlink signal and synchronization. Unequalized grid data is streamed out to be buffered in the grid memory for further processing.
- The central resources of the grid memory, channel estimation, and channel equalization are grouped together, with an interface such that data can be requested by providing an address to the grid, and equalized symbols are output for processing by the decoding stages.
- The indexing subsystems request data from the grid by providing a subcarrier number, an OFDM symbol number, and a read enable flag. These signals are grouped together in a bus for easier routing in the Simulink model. Only one indexing subsystem can access the grid at a time. A controller is used to avoid contention and enable the indexing subsystems at the correct time. Each of the indexing subsystems has a corresponding decoding subsystem, which attempts to decode the data requested from the grid by the indexing subsystem.
- The decoding subsystems receive equalized complex symbols from the grid, with a signal indicating when the incoming data is valid. The decoding subsystems must be enabled before they will start to process valid samples at the input, and it is expected that only one of the decoding subsystems will be enabled at any point in time. A central controller for the SIB1 decoder enables the decoding subsystems at the appropriate time.
- The control subsystem tracks the state of the decoder and enables the decoding and indexing subsystems in the correct sequence using the done, valid, detected, and error signals (as appropriate) for the various processing stages.
- The DCI resource allocation function (`ltehdDCIResourceAllocation`) was selected for implementation on software, as part of a hardware/software co-design implementation. This function was selected due to the low frequency of calculation, and the complex loop behaviour making it inefficient to implement in hardware.

### Structure of Example Model

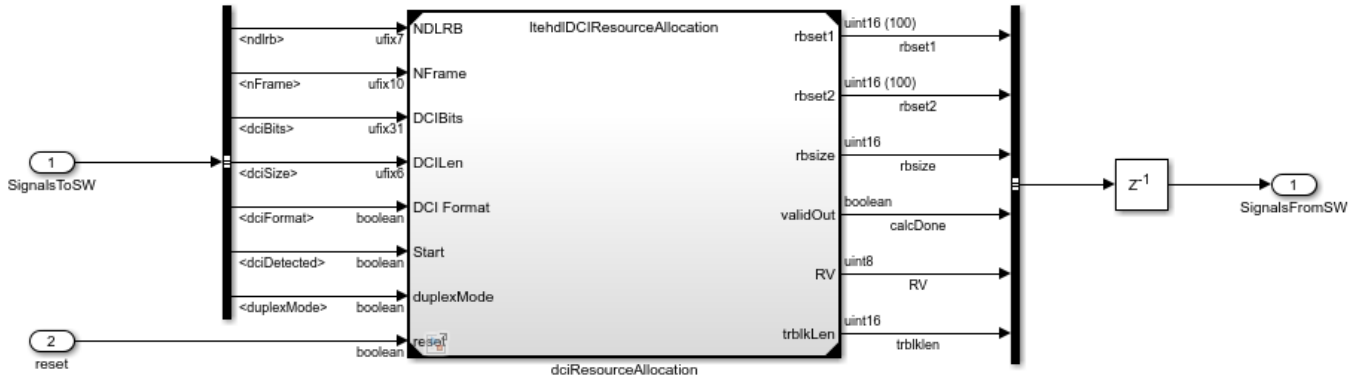
The top-level of the **ltehdSIB1Recovery** model is shown in the figure below. The **HDL LTE SIB1** subsystem supports HDL code generation. The **SW DCI Resource Allocation** subsystem represents the software portion of a design partitioned for hardware/software co-design implementation. The **stateViewer** MATLAB Function block generates text information messages based on the *decoderState* signal from the **HDL LTE SIB1**, and prints this information to both the Simulink Diagnostic Viewer and to a MATLAB figure window. The **stateViewer** also produces the *stopSimulation* signal, which stops the simulation when the decoder reaches a terminal state, as indicated by the text information messages.

## LTE HDL SIB1 Recovery Example



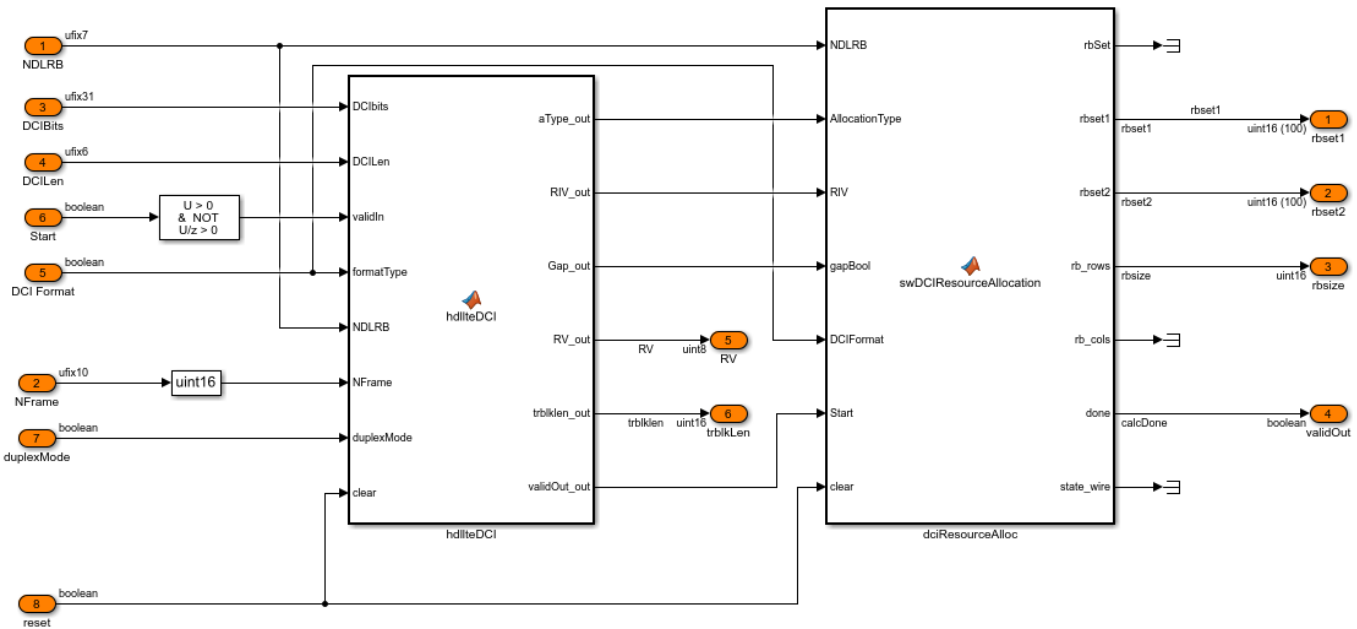
### SW DCI Resource Allocation

The **SW DCI Resource Allocation** subsystem contains an instance of the **ltehdIDCIResourceAllocation** model. Buses are used here to facilitate signal routing to and from this subsystem.



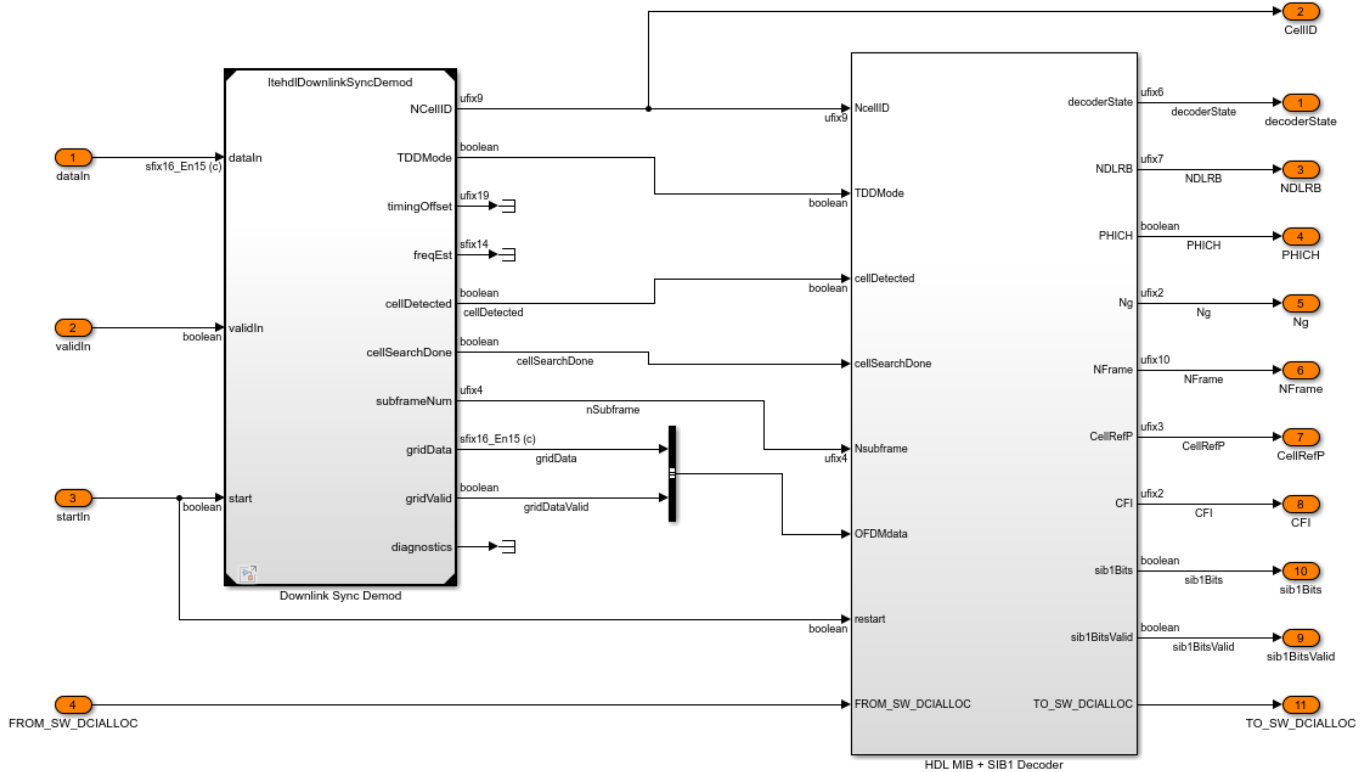
### dcResourceAllocation

The **ltehdlDCIResourceAllocation** model reference performs parsing of the DCI message bits, generates the DCI parameters, then uses the DCI parameters to perform the DCI Physical Resource Block (PRB) allocation calculation. These operations are equivalent to the LTE Toolbox functions `lteDCI` and `lteDCIResourceAllocation`. Due to the complexity of the PRB allocation calculation, this part of the design was selected for implementation in software, as an HDL implementation would require a large amount of hardware resources.



### HDL LTE SIB1

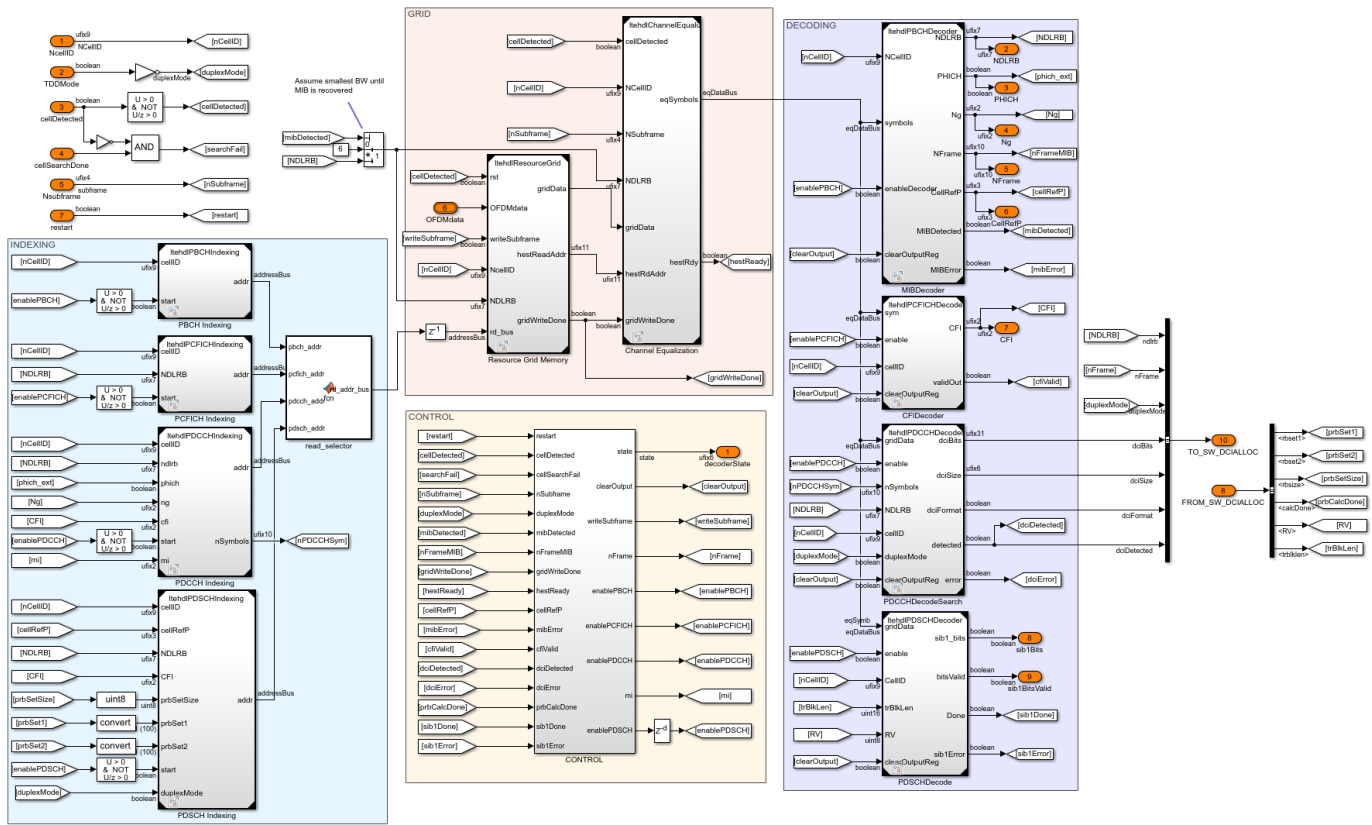
The **HDL LTE SIB1** subsystem contains 2 subsystems. The **Downlink Sync Demod** subsystem is an instance of the **ltehdlDownlinkSyncDemod** model, which is described in the “LTE HDL Cell Search” on page 5-46 example. It performs the cell search, timing and frequency synchronization, and OFDM demodulation. The **HDL MIB + SIB1 Decoder** subsystem performs the channel decoding operations required to decode the MIB and SIB1 messages, as described below.



### HDL MIB + SIB1 Decoder

The **HDL MIB + SIB1 Decoder** structure can be seen below. It receives OFDM demodulated grid data from the **Downlink Sync Demod** subsystem, and stores the data in a subframe buffer, **Resource Grid Memory**. It then calculates the channel estimate for the received data in the **Channel Estimation** subsystem and uses this to equalize data as it is read out of the **Resource Grid Memory**. A series of channel decoding steps are then performed in order to decode the SIB1 message. In total there are 10 referenced models at this level of hierarchy: 4 channel decoders, 4 channel index generation subsystems, and 2 subsystems performing resource grid buffering, channel estimation, and equalization.

The **PBCH Indexing**, **Resource Grid Memory**, **Channel Equalization** and **MIB Decoder** all instantiate the same referenced models used in the MIB example. For more detailed information about the operation of these referenced models, refer to “LTE HDL MIB Recovery” on page 5-80.

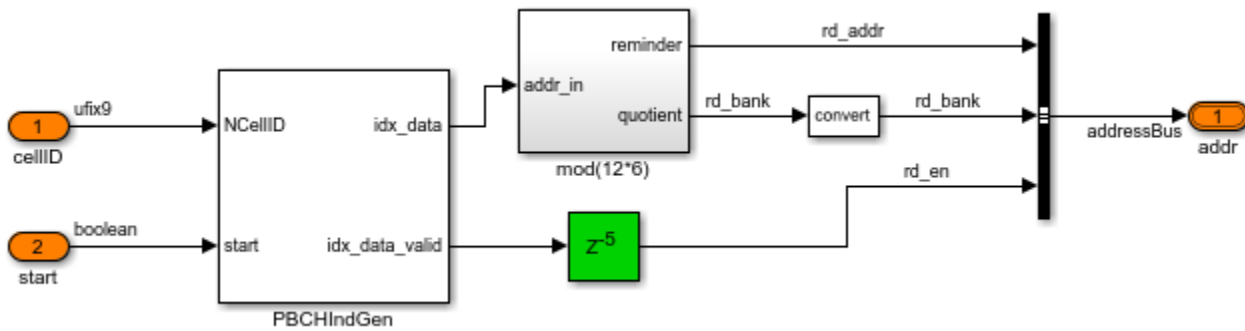


## Indexing Subsystems

There are 4 indexing subsystems, corresponding to the 4 channels that need to be decoded in order to receive a SIB1 message: PBCH, PCFICH, PDCCH, and PDSCH. Each of the indexing subsystems has a corresponding decoding subsystem. The indexing subsystems use an address bus, consisting of a read address corresponding to the subcarrier number, a read bank corresponding to an OFDM symbol, and a read enable signal to control access to the grid. The **read\_selector** MATLAB Function block selects between the outputs of the 4 indexing subsystems based on the read enable signal. It is assumed that only one indexing subsystem will attempt to read from the grid at any point in time, with the **CONTROL** subsystem in charge of enabling the indexing subsystems at the appropriate time.

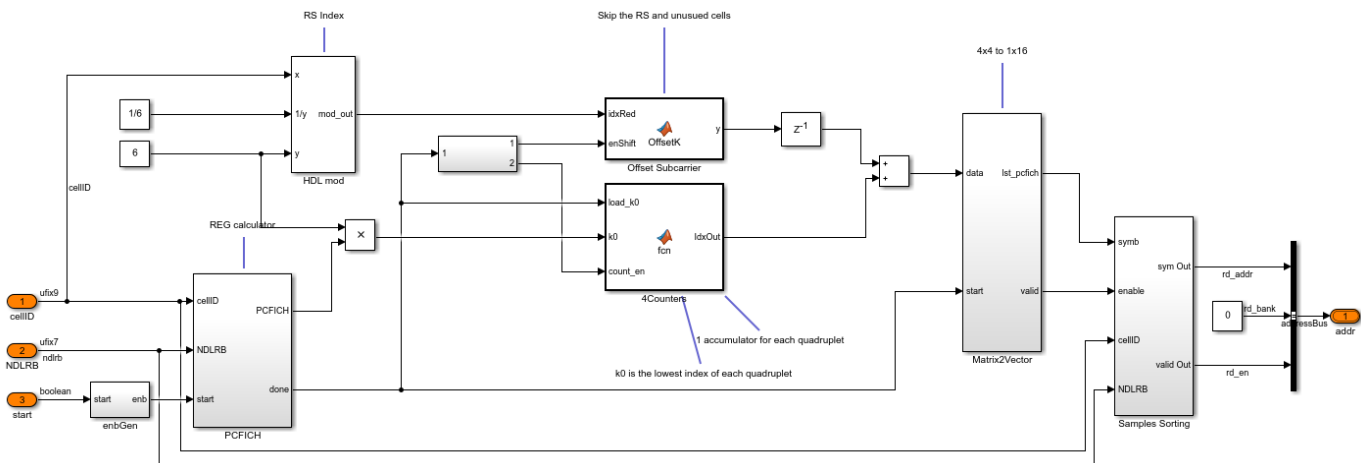
## PBCH Indexing

The **PBCH Indexing** block references the **ltehdIPBCHIndexing** model. It performs the index generation for the PBCH and is equivalent to the LTE Toolbox function `ltePBCHIndices`.



### PCFICH Indexing

The **PCFICH Indexing** block references the **ltehdlPCFICHIndexing** model. It generates the indices required to read the PCFICH symbols from the grid memory and is equivalent to the LTE Toolbox function `ltePCFICHIndices`. The PCFICH is always in the first OFDM symbol (the first memory bank of the grid buffer) and is 16 symbols in length, in 4 groups of 4 symbols. The 4 groups of symbols are distributed at quarters of the occupied bandwidth, with an offset dependent on the Cell ID.



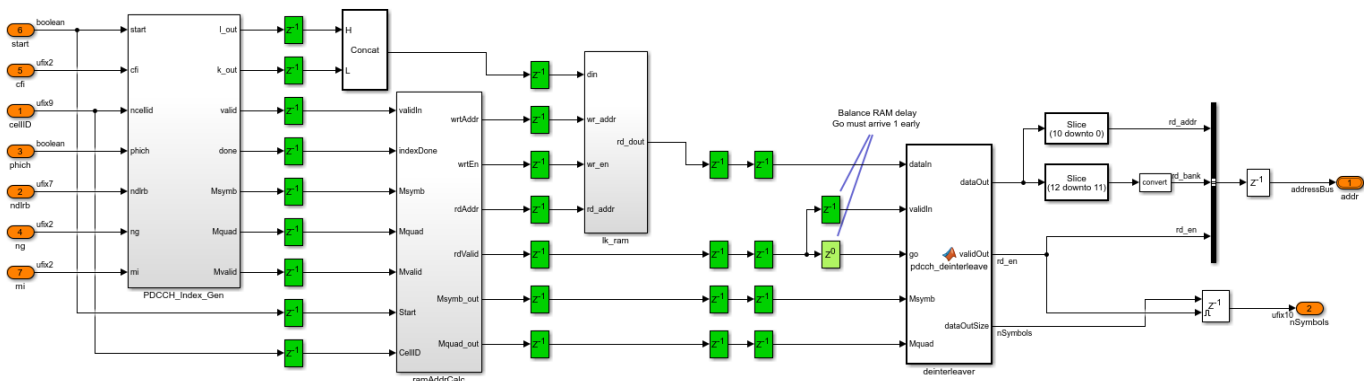
### PDCCH Indexing

The **PDCCH Indexing** subsystem generates the indices required to read the PDCCH symbols from the grid memory. It references the **ltehdlPDCCHIndexing** model and is equivalent to the LTE Toolbox functions `ltePDCCHIndices` and `ltePDCCHDeinterleave`. The PDCCH spans between 1 and 4 OFDM symbols, as defined by the value decoded from the PCFICH. The number of subcarriers spanned by the PDCCH depends on NDLRB. As a result, the number of symbols read from the grid varies, which is indicated by the *nSymbols* output. The PDCCH occupies all of the OFDM symbols indicated by the CFI, but must exclude any locations which have already been allocated to other channels, such as the PCFICH and PHICH. The main indexing calculation is performed by the **PDCCH\_Index\_Gen** subsystem. It calculates the locations of the PCFICH and PHICH then excludes these locations from the range of indices occupied by the PDCCH. In TDD mode number of symbols occupied by the PHICH varies based on the TDD configuration. For different TDD configurations there are three possible values of *mi* (0, 1, and 2), as specified in section 6.9 of [ 1 ], which is a multiplier to the size of the region allocated to the PHICH. When in the duplexing mode is FDD, *mi* is



always 1. The size of the PDCCH in terms of both quadruplets (groups of 4 symbols) and symbols is given by the *Mquad* and *M symb* outputs.

The **ramAddrCalc** and **lk\_ram** subsystems are used to perform a cyclic shift on the quadruplets using the *cellID*. Because the DCI message for SIB1 is always transmitted in the common search space of the PDCCH, it is possible to reduce the number of symbols that are read from the grid memory by retrieving only the symbols from the common search space. In order to do this the PDCCH deinterleaving operation is performed, and the first 576 symbols are requested from the grid. If there are less than 576 symbols in the PDCCH then all of the symbols will be requested. In LTE Toolbox, the PDCCH deinterleaving operation is performed as part of the `ltePDCCHDecode` function. However, as this function simply re-orders the data and does not change the data content, it is possible to move this processing stage to an earlier point in the receiver. By moving the deinterleaver to act on the indices, rather than the data, and reducing to the common search space after deinterleaving, the memory requirements for the deinterleaver and the PDCCH decoder are reduced.

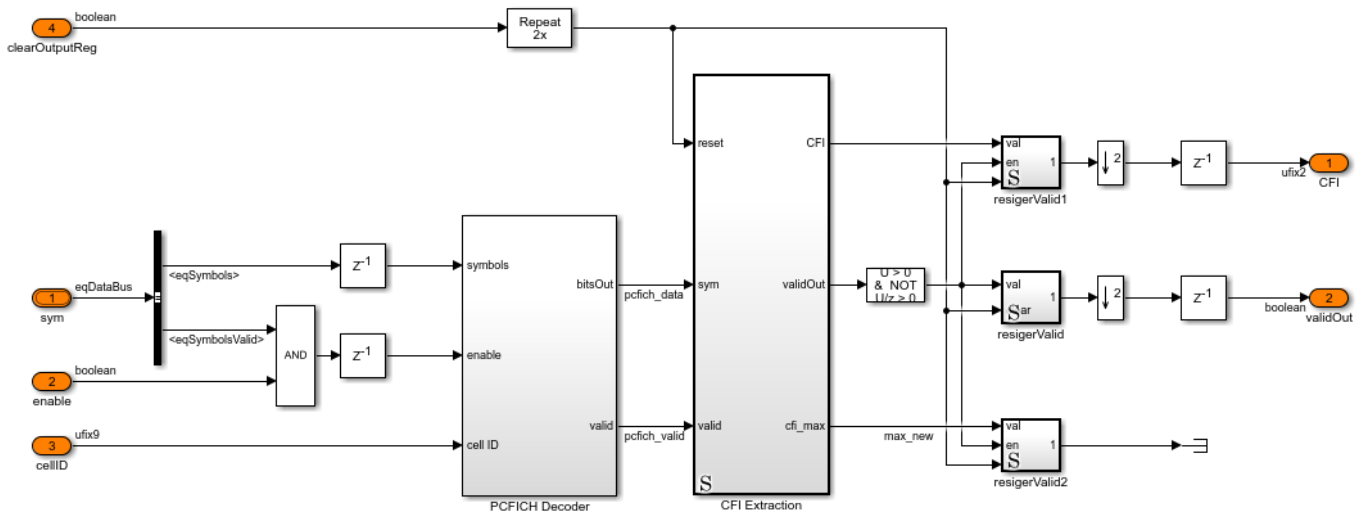


## PDSCH Indexing

The **PDSCH Indexing** calculates the locations of the PDSCH in the grid memory based on the Physical Resource Block (PRB) set, which is passed to this block from the DCI resource allocation calculation in the **SW DCI Resource Allocation** subsystem. The **PDSCH Indexing** is an instance of the **ltehdlPDSCHIndexing** model and is equivalent to the LTE Toolbox function `ltePDSCHIndices`. The PDSCH occupies all of the symbols in the PRB set which have not previously been allocated to another channel. Hence the PDSCH indexing function must exclude any locations which are allocated to the PSS and SSS, and all of the control channel region (i.e. the OFDM symbols indicated by the PCFICH). As the SIB1 message always occurs in subframe 5 of an even frame, there is no need to exclude the PBCH locations, as these only occur in subframe 0.



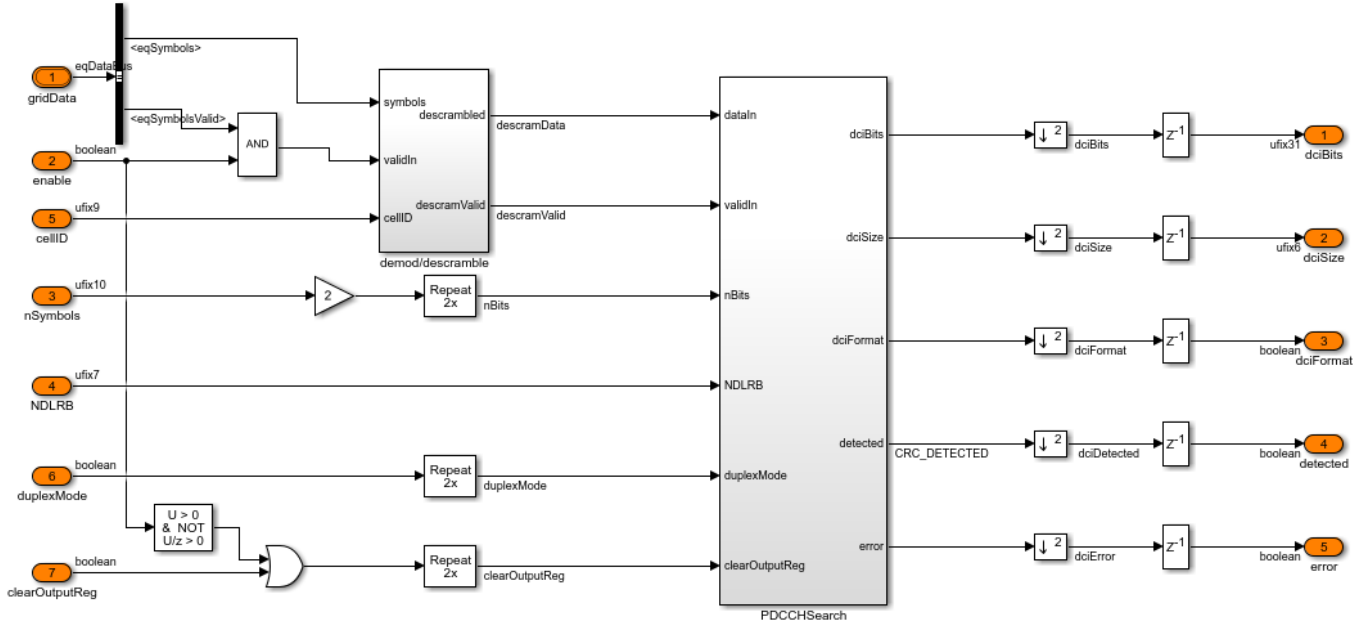
The **CFI Decoder** uses the **ltehdlPCFICHDecoder** referenced model. It performs the PCFICH and CFI decode operations equivalent to the `ltePCFICHDecode` and `lteCFIDecode` functions in LTE Toolbox. The input from the **Channel Equalization** is the 16 symbols requested by the **PCFICH Indexing**. The **PCFICH Decoder** subsystem performs descrambling and QPSK demodulation on the 16 PCFICH symbols to produce 32 soft bits. The **CFI Extraction** subsystem then correlates the soft bits with the three CFI codewords. The codeword with the strongest correlation gives the CFI value of 1, 2, or 3. The CFI value indicates the number of OFDM symbols occupied by the PCFICH. If NDLRB is greater than ten, the number of OFDM symbols is equal to the CFI value (1, 2, or 3). If NDLRB is less than or equal to ten, the number of OFDM symbols is one larger than the CFI value (2, 3, or 4). This information is used by the **PDCCH Indexing** and **PDSCH Indexing** subsystems.



### PDCCHDecodeSearch

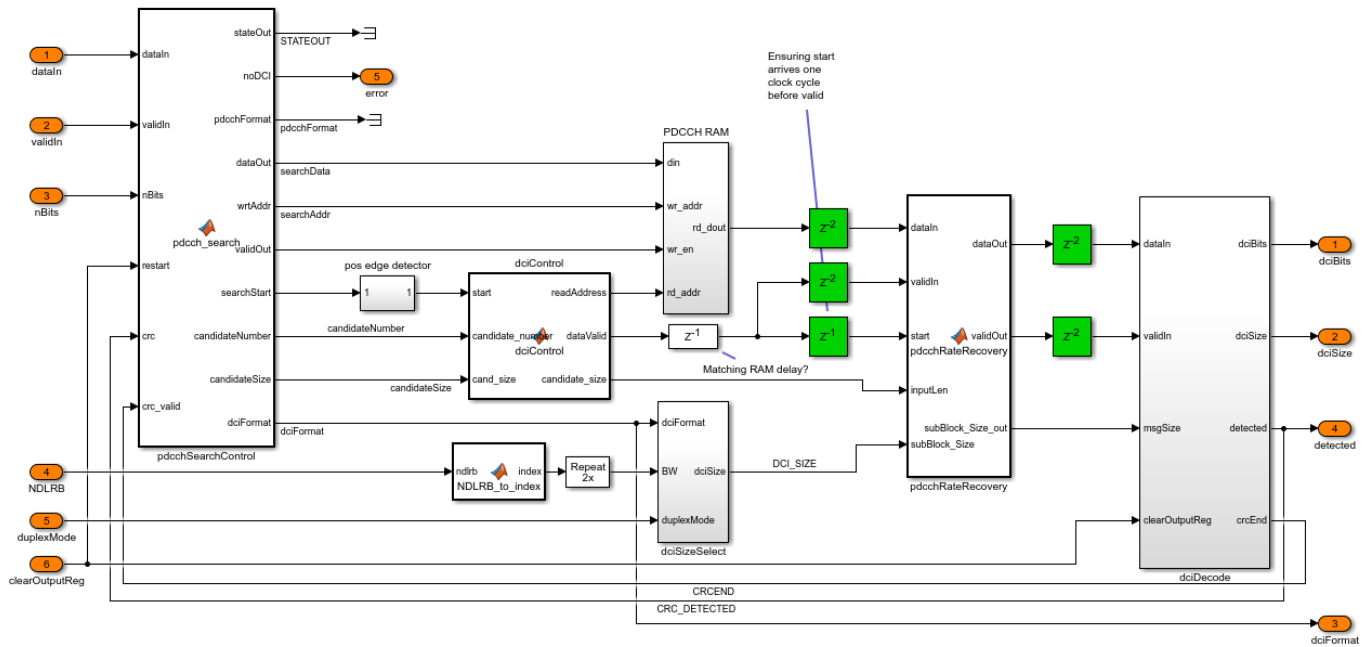
The **PDCCHDecodeSearch** subsystem uses the **ltehdlPDCCHDecode** referenced model. It performs the PDCCH decode, blind PDCCH search, and DCI decode operations required to locate and decode the SIB1 DCI message within the PDCCH. This is roughly equivalent to the LTE Toolbox functions `ltePDCCHDecode`, `ltePDCCHSearch`, and `lteDCI` (which is used within `ltePDCCHSearch`) with a few modifications. As the SIB1 DCI message is always within the common search space of the PDCCH, only these symbols are retrieved from the grid buffer, as described above for **PDCCH Indexing**. The SIB1 DCI message is always DCI format 1A or 1C. It is found in the PDCCH common search space using PDCCH aggregation levels 4 or 8, and the CRC for the DCI message is scrambled with the System Information Radio Network Temporary Identifier (SI-RNTI). Using this information the search can be simplified compared to the LTE Toolbox `ltePDCCHSearch` implementation. For more information on the LTE Toolbox PDCCH search process, see the “PDCCH Blind Search and DCI Decoding” (LTE Toolbox) example. The **PDCCHSearch** subsystem blindly attempts to decode DCI messages from all of the possible candidates and combinations within the common search space until a DCI message with the correct CRC mask is decoded, indicating that the SIB1 DCI message has been found, or all candidates have been attempted, and no SIB1 DCI message has been found. When a SIB1 DCI message has been found, the search stops, and the information from the decoded DCI message is returned from the block. This information is then passed to the **SW DCI Resource Allocation** subsystem to parse the DCI message, and determine which resources in the PDSCH have been allocated to the SIB1 message.

The **demod/descramble** subsystem performs descrambling and QPSK demodulation, while the **PDCCHSearch** subsystem performs the search process described in more detail below.



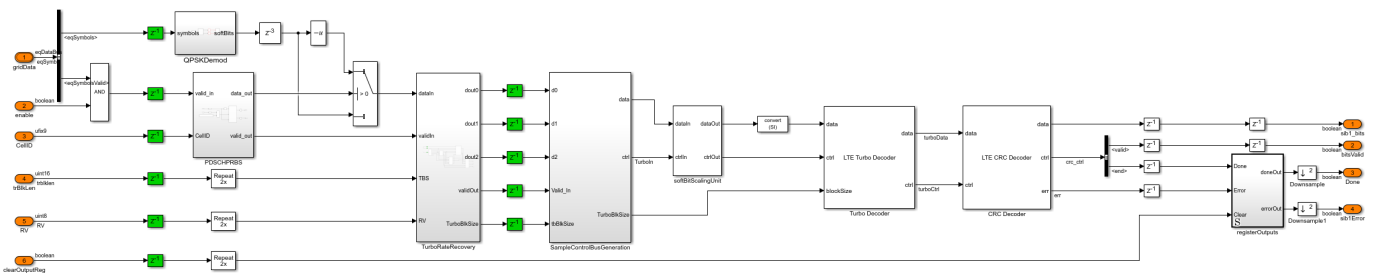
## PDCCHSearch

Within the **PDCCHSearch** subsystem there are a number of processing stages which combine to perform the PDCCH search operation. The **pdchSearchControl** MATLAB Function block writes the incoming data to the **PDCCH RAM**, then controls the search process, iterating through the different combinations of DCI format, PDCCH format, and PDCCH candidates. The **dciControl** MATLAB Function block generates the read addresses for the **PDCCH RAM** given the PDCCH candidate number and size. The **pdchRateRecovery** MATLAB Function block is equivalent to the LTE Toolbox function `lteRateRecoverConvolutional`, performing the deinterleaving and rate recovery for the convolutional decoder. The **dciDecode** subsystem performs the convolutional decoding of the rate recovered bits, then checks the message CRC with the SI-RNTI to determine if a SIB1 DCI message has been found. If successfully decoded, the DCI message bits are buffered and output, and the search process is stopped. The PDCCH search process will also stop if all of the possible candidates have been checked, but no DCI message for SIB1 has been found, with the *error* output being asserted.



### PDSCHDecode

The **PDSCHDecode** subsystem uses the **ltehdlPDSCHDecode** referenced model. It is equivalent to the **ltePDSCHDecode** and **lteDLSCHDecode** functions in LTE Toolbox. The **QPSKDemod** and **PDSCHPRBS** demodulate the incoming signals and generate the descrambling sequence. The descrambled bits are then passed to **TurboRateRecovery** which performs deinterleaving and rate recovery of the incoming bits. The **SampleControlBusGeneration** subsystem generates the control signals required to interface with the **LTE Turbo Decoder** and **LTE CRC Decoder**, which decode the signal. The **LTE CRC Decoder** indicates the status of the CRC decode, asserting the *err* signal, along with the *end* signal in the *ctrl* bus output, if errors have been detected. If the CRC does not detect any errors then the SIB1 message has been successfully decoded, and the *sib1\_bits* are streamed out from the block, with *bitsValid* indicating when *sib1\_bits* are valid. Once the SIB1 message has been detected, and the bits output from **PDSCHDecode**, the simulation stops. No attempt is made to combine the different Redundancy Versions (RVs) of the DLSCH.

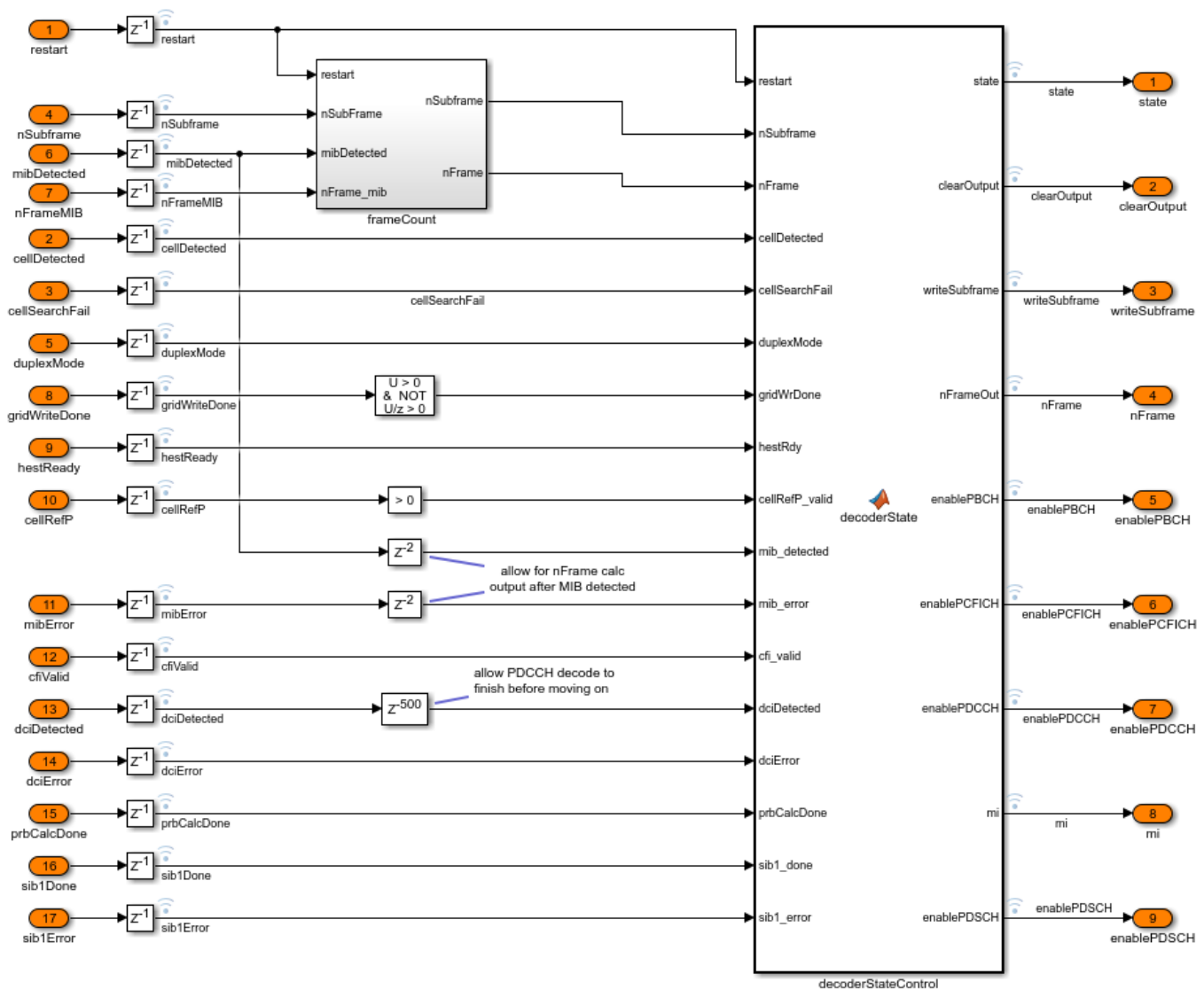


### CONTROL Subsystem

The **CONTROL** subsystem tracks the state of the decoder through the different channel processing stages, enabling each of the indexing and decoding subsystems in turn. The subframe number and frame number are taken as inputs, allowing the **frameCount** function to track the System Frame Number (SFN). The subframe and frame numbers are used to determine when channels will be

available for decode (e.g. SIB1 is only transmitted on subframe 5 of even numbered frames). The **decoderState** MATLAB Function block implements a simple state machine that keeps track of which processing stages have been completed, and which stage to enable next. The state of the decoder is output from the controller, and is parsed by the **stateViewer** MATLAB Function block at the top level of the model to produce human readable messages.

When the received signal is in TDD mode the **CONTROL** subsystem manages the blind search of each of the TDD configurations, running the **PDCCH Indexing** and **PDCCH Decoding** subsystems for each of the three possible  $mi$  values. The different  $mi$  values  $\{0,1,2\}$  result in different PHICH allocations, hence different PDCCH allocations. The PDCCH allocations are calculated, and the PDCCH decode attempted for each  $mi$  value, until a SIB1 DCI message is found, or all of the possibilities are exhausted.



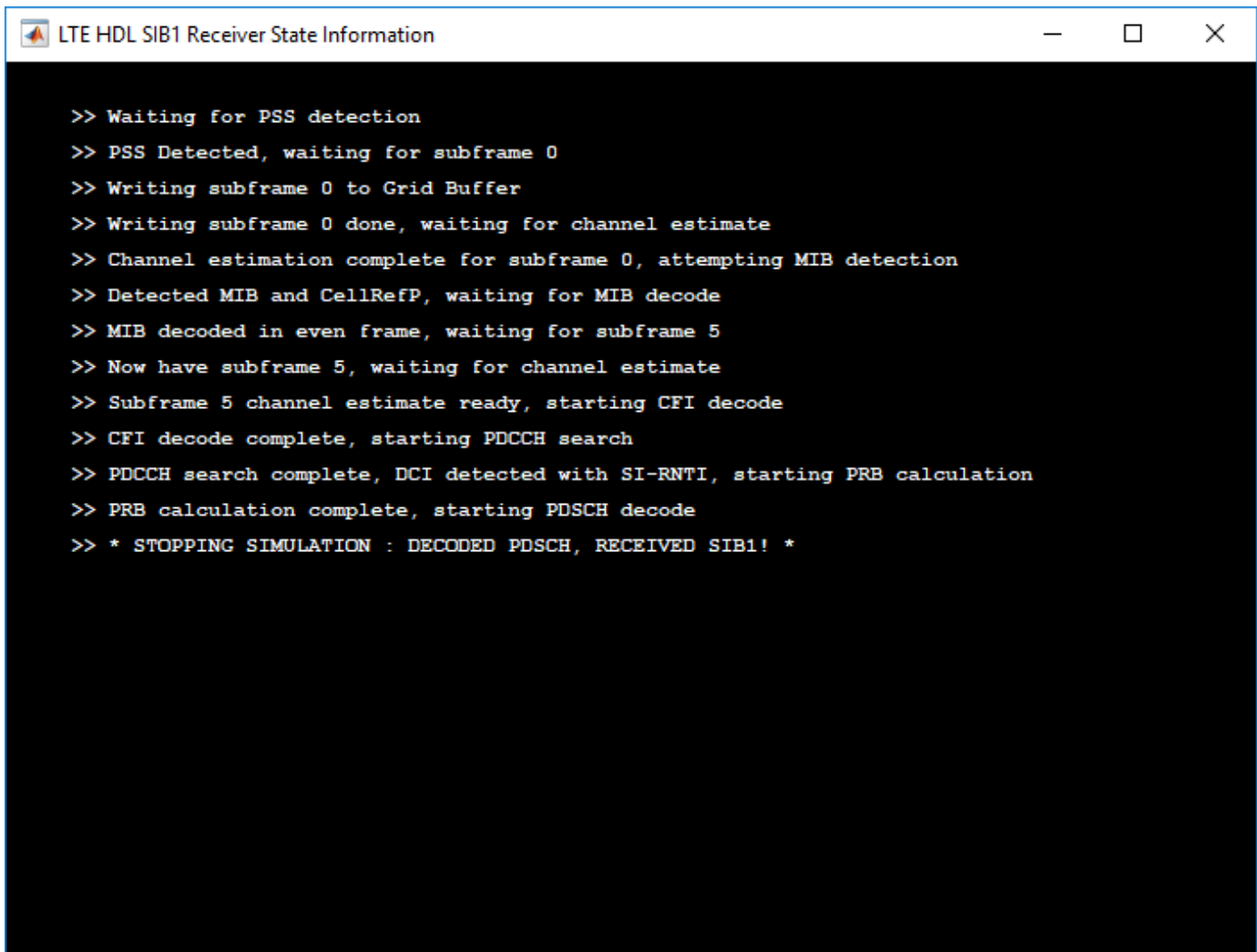
## Results and Display

The simulation model is configured to stop the simulation under a number of conditions:

- If the cell search does not find any cells.
- If the MIB detection has an error.
- If a SI-RNTI DCI message is not detected during the PDCCH search.
- At the end of the PDSCH decoding attempt.

If the SIB1 message is successfully decoded, it is output from the *sib1Bits* port, with the *sib1BitsValid* port indicating when the output is valid. The data is buffered and sent to the MATLAB workspace.

The LTE HDL SIB1 Receiver State Information figure window displays text messages indicating the current state of the decoder. The state of the system is tracked by the **CONTROL** subsystem, with the *decoderState* signal passed up to the top level of the model where the **statePrint** MATLAB Function block generates the text info messages.

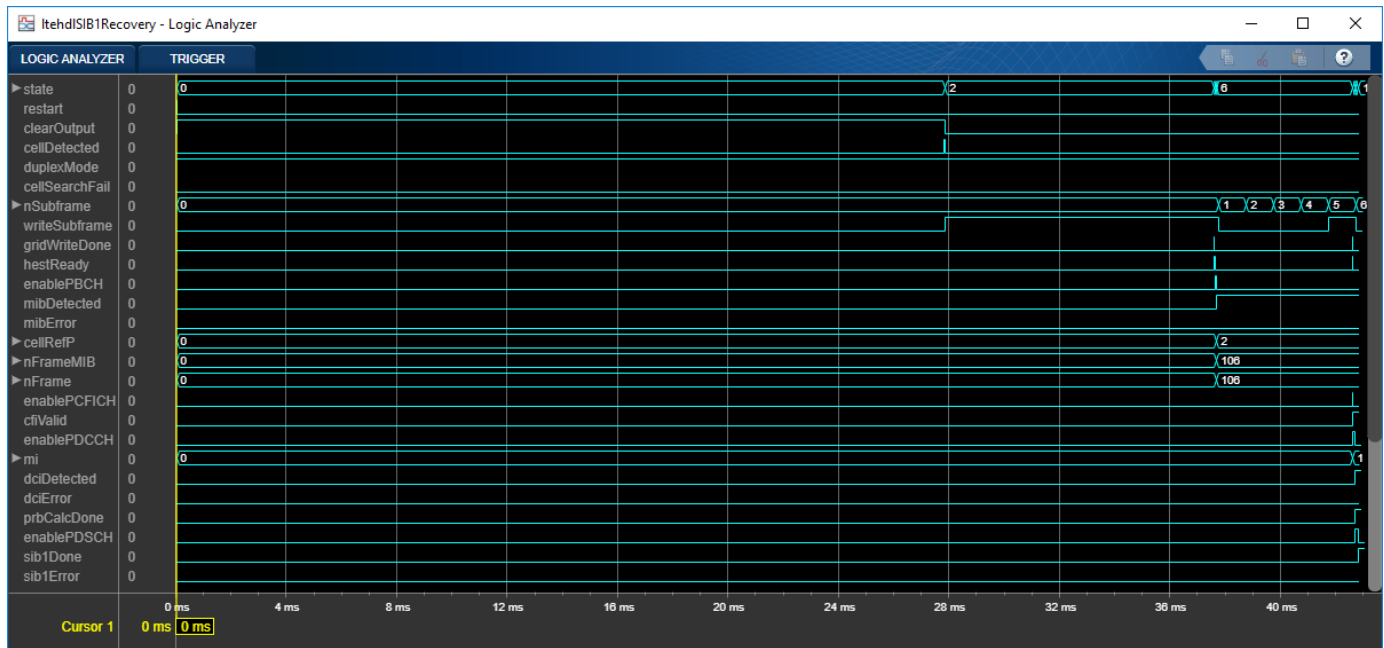


```

>> Waiting for PSS detection
>> PSS Detected, waiting for subframe 0
>> Writing subframe 0 to Grid Buffer
>> Writing subframe 0 done, waiting for channel estimate
>> Channel estimation complete for subframe 0, attempting MIB detection
>> Detected MIB and CellRefP, waiting for MIB decode
>> MIB decoded in even frame, waiting for subframe 5
>> Now have subframe 5, waiting for channel estimate
>> Subframe 5 channel estimate ready, starting CFI decode
>> CFI decode complete, starting PDCCH search
>> PDCCH search complete, DCI detected with SI-RNTI, starting PRB calculation
>> PRB calculation complete, starting PDSCH decode
>> * STOPPING SIMULATION : DECODED PDSCH, RECEIVED SIB1! *

```

The display blocks in the top level of the model show some of the key parameters decoded by each of the channel processing stages. A number of the key control signals, from within the **CONTROL** subsystem, are logged for viewing with the logic analyzer.



### HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Note that test bench generation for this example takes a long time due to the length of the simulation required to create the test vectors.

HDL code for the **HDL LTE SIB1** subsystem was generated using the HDL Workflow Advisor IP Core Generation workflow for a Xilinx® Zynq®-7000 ZC706 evaluation board, and then synthesized. The post place and route resource utilization results are shown below. The design met timing with a target clock frequency of 150MHz. Using the workflow advisor IP core generation workflow allows the input and output ports to be mapped to AXI4-Lite registers, reducing the number of FPGA IO pins required, and allows the design to be split between hardware and software.

Resource	Usage
Slice Registers	128726
Slice LUTs	70032
RAMB18	52
RAMB36	193
DSP48	156

For more information see “Prototype LTE Algorithms on Hardware” on page 2-12.

### Simulation Limitations

The **stateViewer** MATLAB function block is not supported for simulation in rapid accelerator mode. This block can be removed or commented out if rapid accelerator simulation is required.



**References**

1. 3GPP TS 36.211, "Physical Channels and Modulation"

**See Also****Related Examples**

- "LTE HDL Cell Search" on page 5-46
- "LTE HDL MIB Recovery" on page 5-80

## LTE HDL MIB Recovery

This example shows the design of a LTE MIB recovery system optimized for HDL code generation and hardware implementation.

### Introduction

The model presented in this example can be used to locate and decode the MIB from LTE downlink signals. It builds upon the “LTE HDL Cell Search” on page 5-46 example, adding processing stages to decode the MIB. The Master Information Block (MIB) message is transmitted in the Physical Broadcast Channel (PBCH), and carries essential system information:

- Number of Downlink Resource Blocks (NDRB), indicating the system bandwidth
- System Frame Number (SFN)
- PHICH (Physical HARQ Indicator Channel) Configuration

The design is optimized for HDL code generation and the architecture is extensible, allowing additional processing stages to be added, such as indexing and decoding for the PCFICH, PDCCH and PDSCH (see “LTE HDL SIB1 Recovery” on page 5-63).

### MIB Processing Stages

In order to decode the MIB message this example performs these operations:

- Cell search and OFDM demodulation
- Buffering grid data
- Channel estimation and equalization
- PBCH Indexing - locating PBCH within the grid
- PBCH Decoding - decoding PBCH, BCH, and MIB

### Cell Search and OFDM Demodulation

LTE signal detection, timing and frequency synchronization, and OFDM demodulation are performed on the received data. This produces the grid data and provides information on the subframe number and cell ID of the received waveform. The MIB message is always carried in subframe 0, and the cellID is used to determine the location of the cell-specific reference signals (CRS) for channel estimation, as well as being used to initialize the descrambling sequence for PBCH Decoder.

### Buffering Grid Data

As the MIB message is always carried in subframe 0 of the downlink signal, subframe 0 is buffered in a memory bank. At the same time as the subframe is being written to the memory bank, the location of the CRS are calculated using the cellID, and CRS are sent to the channel estimator.

### Channel Estimation

The CRS from the received grid are then compared to the expected values, and the phase offset calculated. The channel estimates for each CRS are averaged across time, and linear interpolation is used to estimate the channel for subcarriers which do not contain CRS. The channel estimate for the subframe is used to equalize data when it is read from the grid memory.

### PBCH Indexing

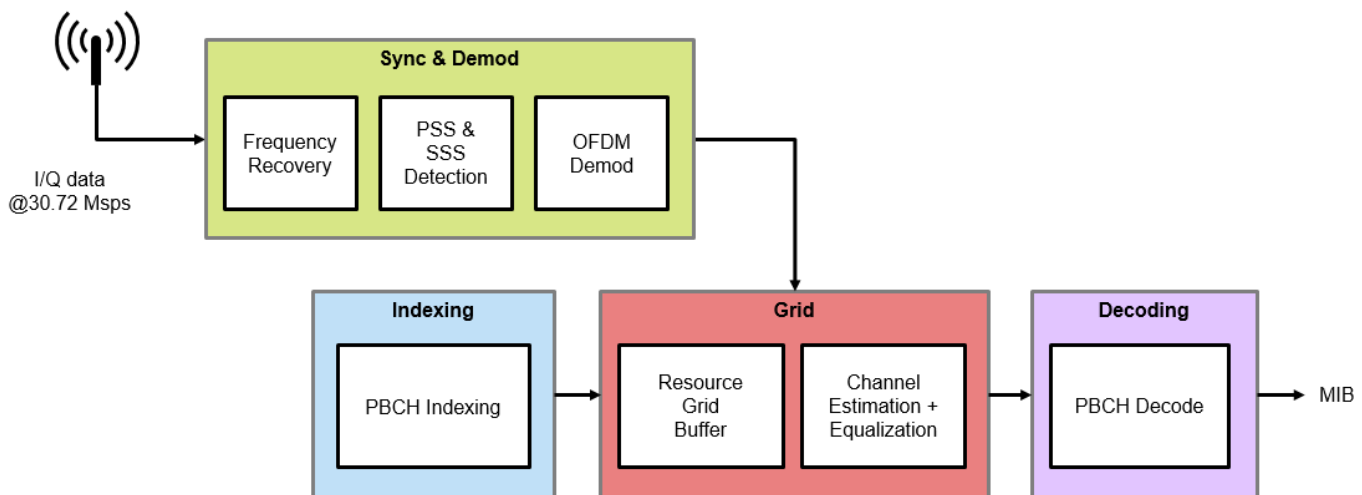
The PBCH is always allocated to the central 6 Resource Blocks (RBs) of subframe 0, within the first 4 OFDM symbols of the 2nd slot. It occupies all of the Resource Elements (REs) within this region, excluding the locations allocated to CRS. The locations of the CRS are calculated using the cellID, then the addresses of the REs occupied by the PBCH can be calculated (240 locations in total), and the data retrieved from the grid memory bank.

### PBCH Decoding

As the PBCH data is read from the grid memory bank it is equalized using the channel estimate. The 240 equalized PBCH symbols are buffered, and PBCH and BCH decoding are attempted for each of the 4 possible versions of the MIB within a PBCH transport block. Each of these versions requires a different descrambling sequence, so descrambling, demodulation, rate recovery, convolutional decoding, and CRC check must be attempted for each. If successfully decoded, the CRC value gives the cellRefP value - the number of transmit antennas, and the MIB bits can be parsed to give the system parameters.

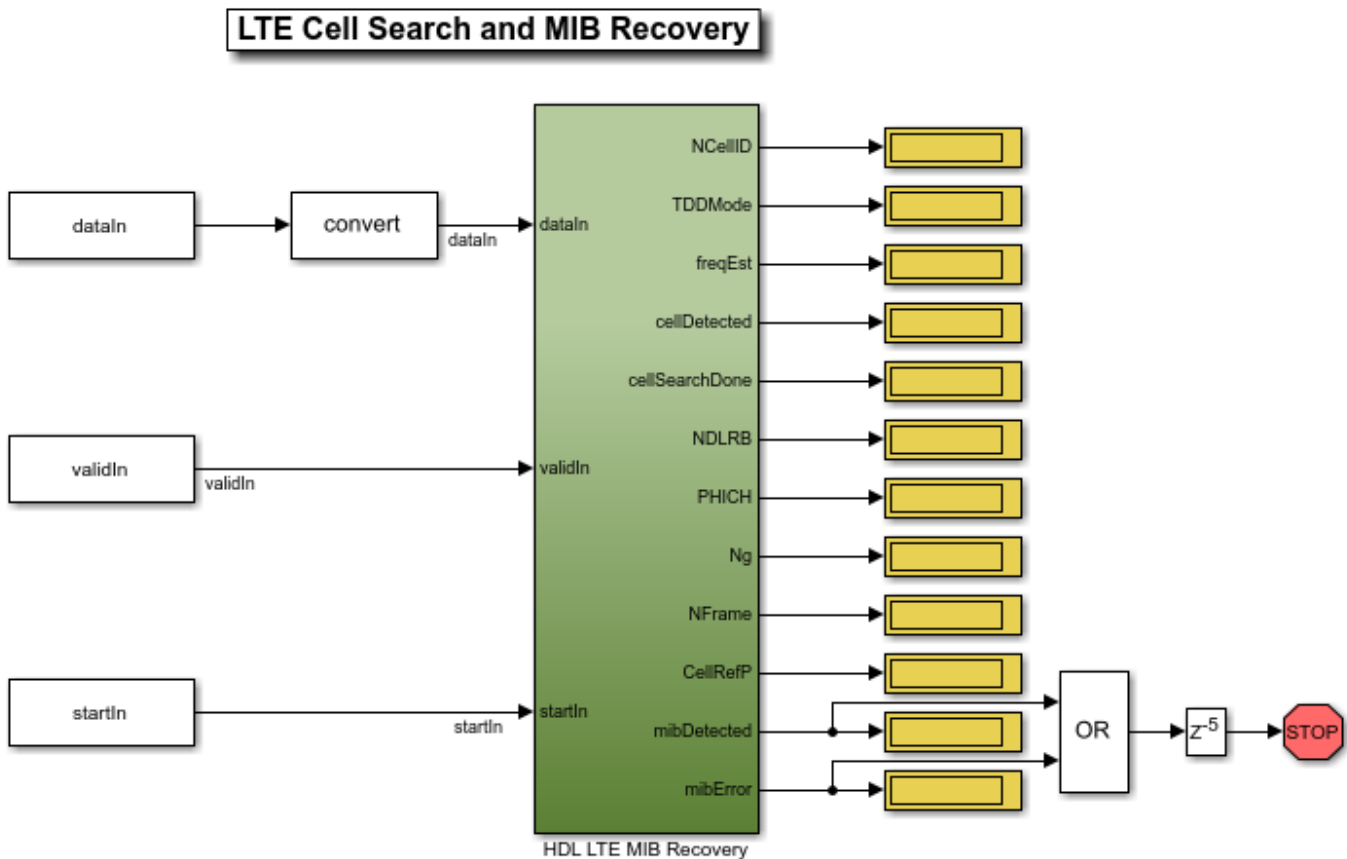
### Model Architecture

The architecture of the LTE HDL Cell Search and MIB Recovery implementation is shown in the diagram below.



The input to the receiver is baseband I/Q data, sampled at 30.72 Msps. A 2048-point FFT is used for OFDM demodulation, and is sufficient to decode all of the supported LTE bandwidths. The resource grid buffer is capable of storing one subframe of LTE data. Once the receiver has synchronized to a cell, data from the OFDM demodulator is written into the grid buffer. The PBCH indexing block then generates the indices of the resource elements which carry the PBCH. Those resource elements are read out of the grid buffer and equalized, before being passed through the PBCH decoder. This architecture is designed to be extensible and scalable so that additional channel indexing and decoding functions can be inserted as needed. For example it can be extended to perform SIB1 recovery as shown in the “LTE HDL SIB1 Recovery” on page 5-63 example.

The top level of the **ltehdlMIBRecovery** model is shown below. HDL code can be generated for the **HDL LTE MIB Recovery** subsystem.



The `ltehdlMIBRecovery_init.m` script is executed automatically by the model's `InitFcn` callback. This script generates the `dataIn` and `startIn` stimulus signals as well as any of the constants needed to initialize the model. Input data can be loaded from a file which, for this example, is "LTE Receiver Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio). Alternatively, an LTE waveform can be synthesized using LTE Toolbox functions. To select an input source, change the `loadfromfile` parameter in `ltehdlMIBRecovery_init.m`.

```

SamplingRate = 30.72e6;
simParams.Ts = 1/SamplingRate;

loadfromfile = true;

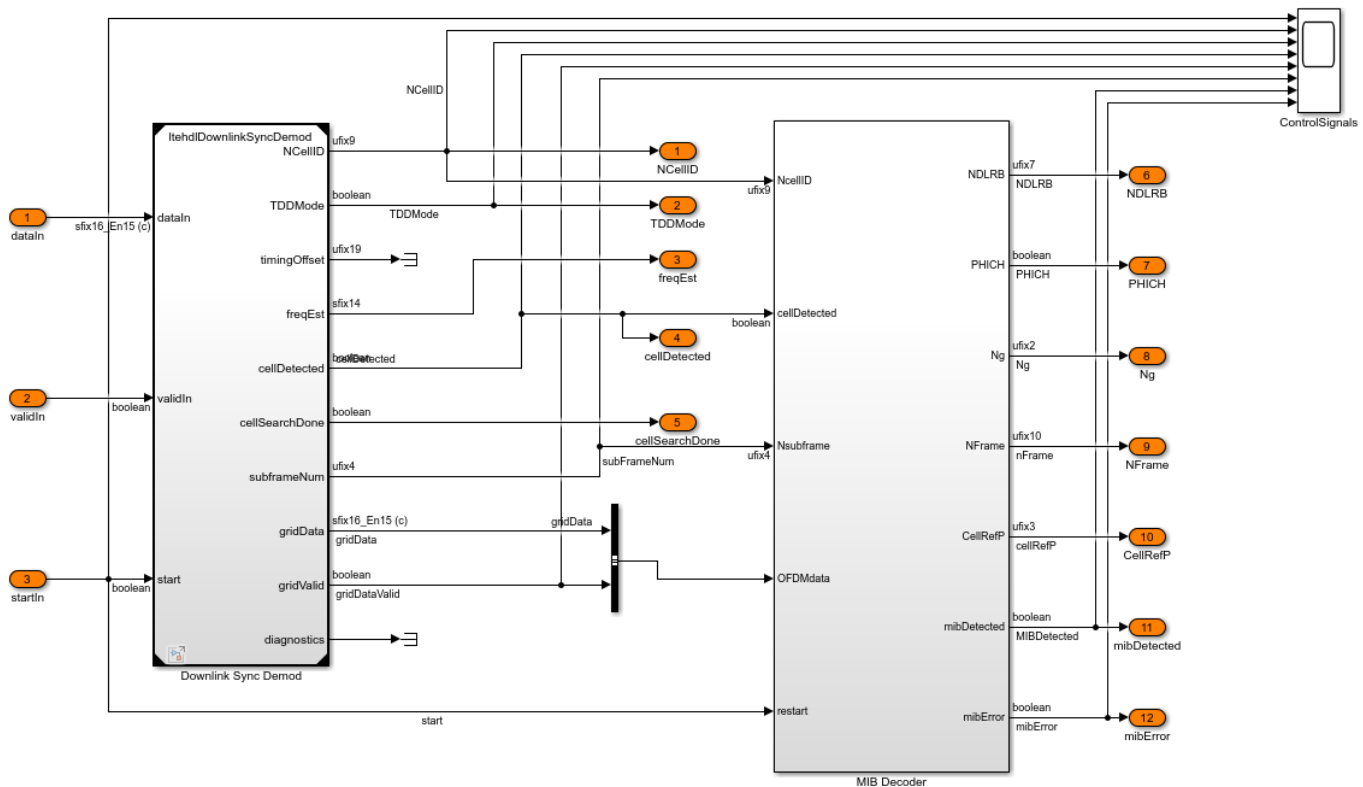
if loadfromfile
    load('eNodeBWaveform.mat');
    dataIn = resample(rxWaveform,SamplingRate,fs);
else
    dataIn = hGeneratedDLRXWaveform();
end

```

### HDL Optimized LTE MIB Recovery

The structure of the **HDL LTE MIB Recovery** subsystem is shown below. The **Downlink Sync Demod** block performs frequency and time synchronization, PSS/SSS signal detection, and OFDM

demodulation. The **MIB Decoder** subsystem buffers subframe 0 of the incoming data, performs channel estimation, and attempts to decode the PBCH to recover the MIB information.



## Downlink Synchronization and Demodulation

The **Downlink Sync Demod** subsystem takes in I/Q data at 30.72 Msp/s, and outputs the unequalized downlink resource grid data. It is an instance of the **ltehdlDownlinkSyncDemod** model reference, which implements the following functions:

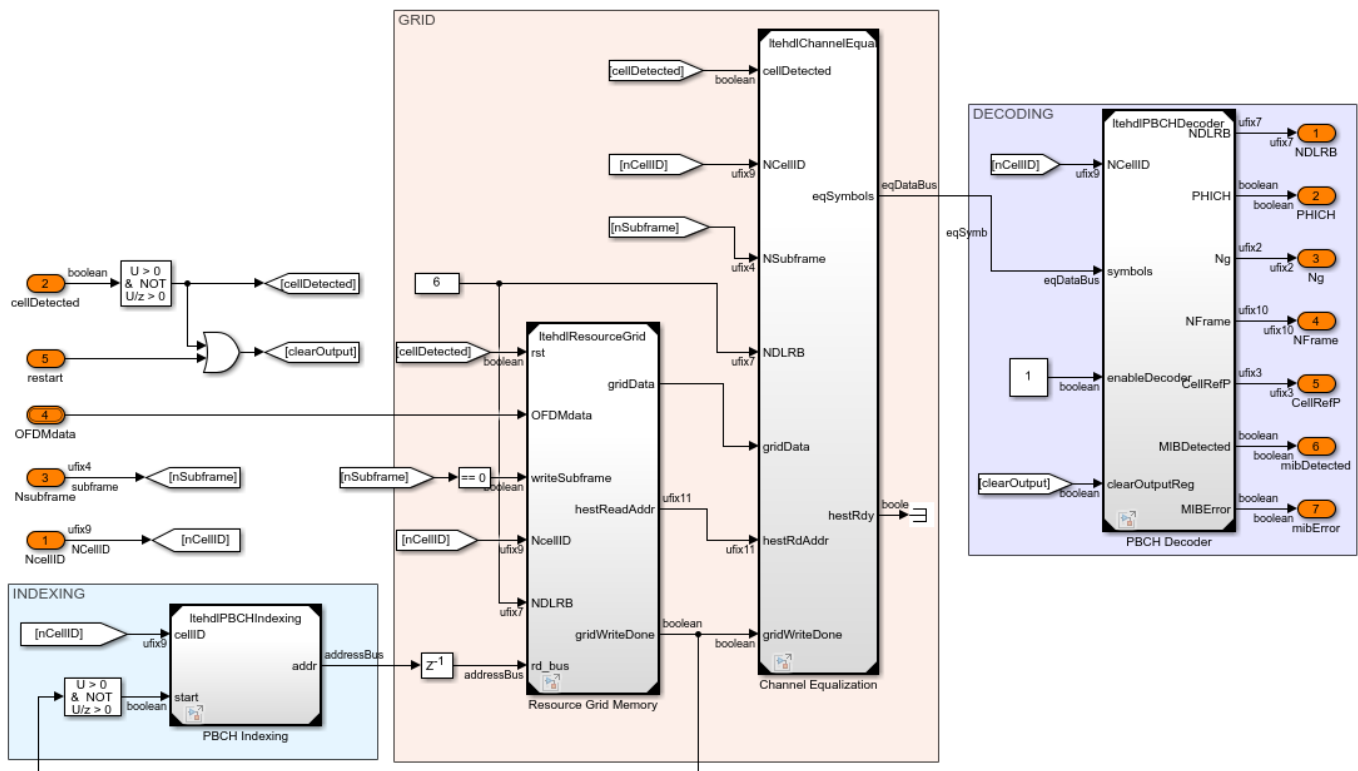
- Frequency recovery
- Primary Synchronization Signal (PSS) detection
- Secondary Synchronization Signal (SSS) detection
- Timing recovery, based on the PSS and SSS signals
- OFDM demodulation (using a 2048 point FFT)
- Cell ID calculation, based on PSS and SSS detection results

The operation of the **ltehdlDownlinkSyncDemod** is described in more detail in the “LTE HDL Cell Search” on page 5-46 example.

## MIB Decoder

The **MIB Decoder** subsystem is shown below. It consists of four subsystems: **PBCH Indexing**, **Resource Grid Memory**, **Channel Equalization**, and **PBCH Decoder**. The order of operations is as follows:

- 1 The *cellDetected* input is asserted, preparing the subsystem to receive and process data.
- 2 OFDM data is streamed into the **MIB Decoder** subsystem, and subframe 0 is stored in the **Resource Grid Memory**.
- 3 The **Channel Equalization** subsystem calculates a channel estimate for subframe 0
- 4 The **PBCH Indexing** block starts generating PBCH resource element indices.
- 5 Those resource elements are then read out of the **Resource Grid Memory** and equalized by the **Channel Equalization** block.
- 6 Finally the equalized PBCH data is passed through the **PBCH Decoder** block and the MIB is extracted.

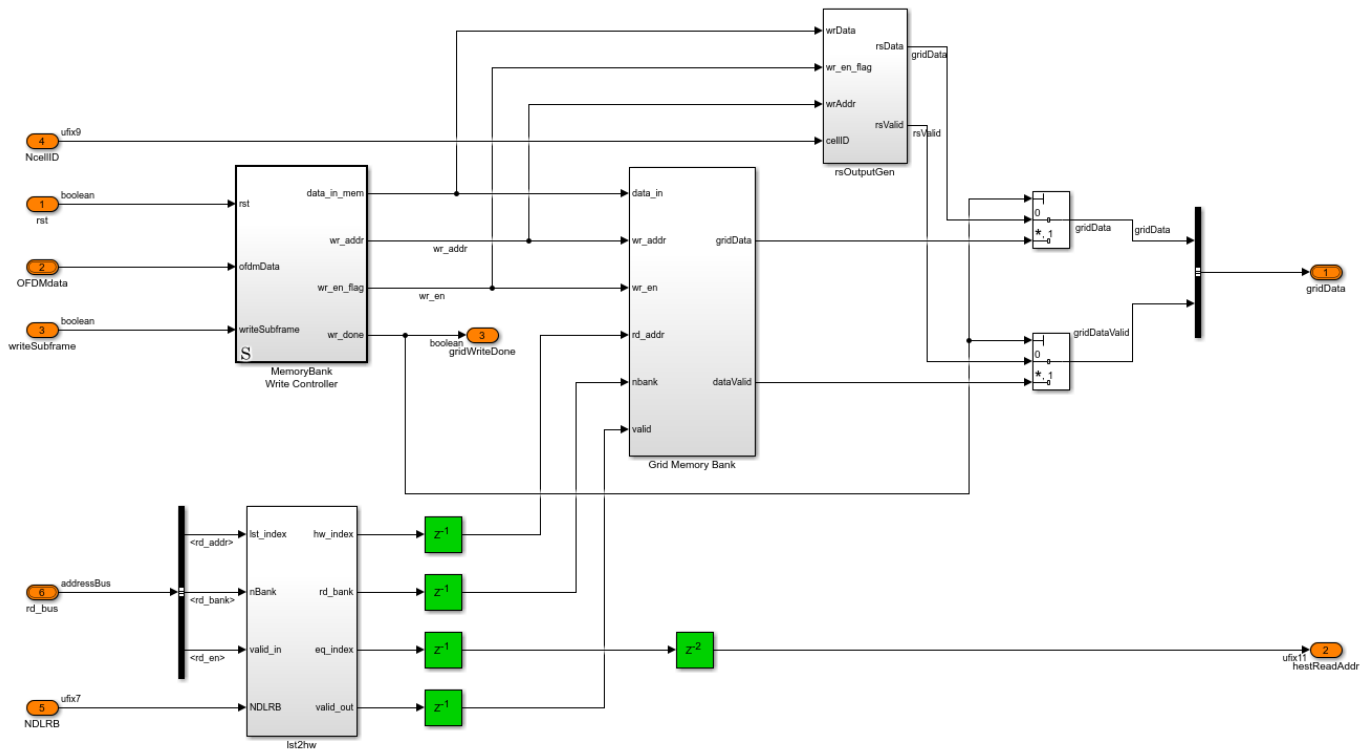


### Resource Grid Memory

The **Resource Grid Memory** block contains a memory bank, logic to control reading and writing of the grid memory bank, and logic to locate and output the CRS. The memory bank capacity is one subframe of demodulated OFDM data at the largest supported LTE bandwidth (20MHz).

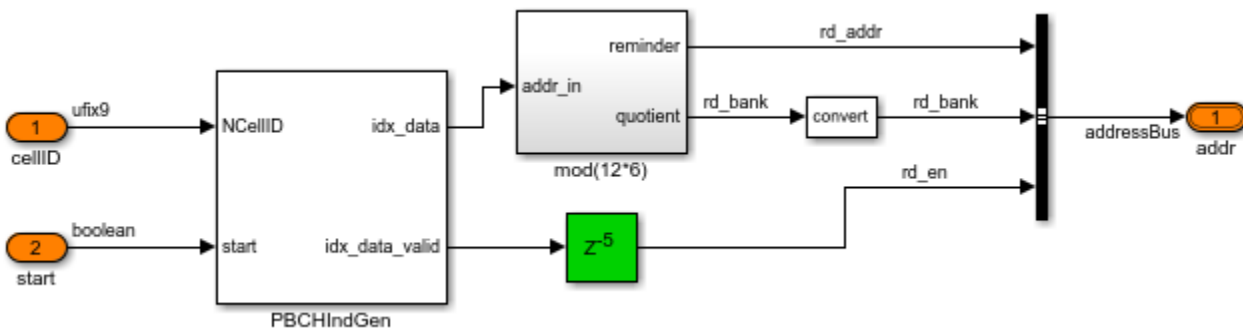
The **MemoryBank Write Controller** is responsible for writing subframes of data to the memory bank. The *writeSubframe* input enables the write controller for the appropriate subframes; subframe 0 in the case of the present example. The **LTE Memory Bank** contains RAM of dimensions 14 x 2048 x 16 bit complex values; that is 14 ODFM symbols, each containing 2048 complex values. The **rsOutputGen** subsystem calculates the locations of the cell reference symbols, extracts these from the data as it is written to the grid memory, and outputs these via the *gridData* output signal.

The *gridData* output port carries the CRS signals, from **rsOutputGen**, when data is being written to the grid memory (*gridWriteDone* output port is low) and carries data from the **LTE Memory Bank** when the write to the grid memory is complete (*gridWriteDone* output port is high).



### PBCH Indexing

The **PBCH Indexing** block computes the memory addresses required to retrieve the PBCH from the grid memory buffer. This is equivalent to the LTE Toolbox `ltePBCHIndices` function. The data retrieved from the grid memory is then equalized and passed to the **PBCH Decoder** for processing. The PBCH Indexing subsystem becomes active after the data for subframe 0 has been written to the grid memory, as indicated by the *gridWriteDone* output of the **Resource Grid Memory** subsystem. The PBCH is always 240 symbols in length, centered in the middle subcarriers, in the first 4 symbols within the 2nd slot of subframe 0.



### Channel Estimation and Equalization

The **Channel Equalization** block contains three main subsystems. **cellRefGen** generates the cell-specific reference signal (CRS) symbols using a Gold Sequence generator. **chEst** performs channel estimation assuming two transmit antennas by using a simple, hardware-friendly channel estimation

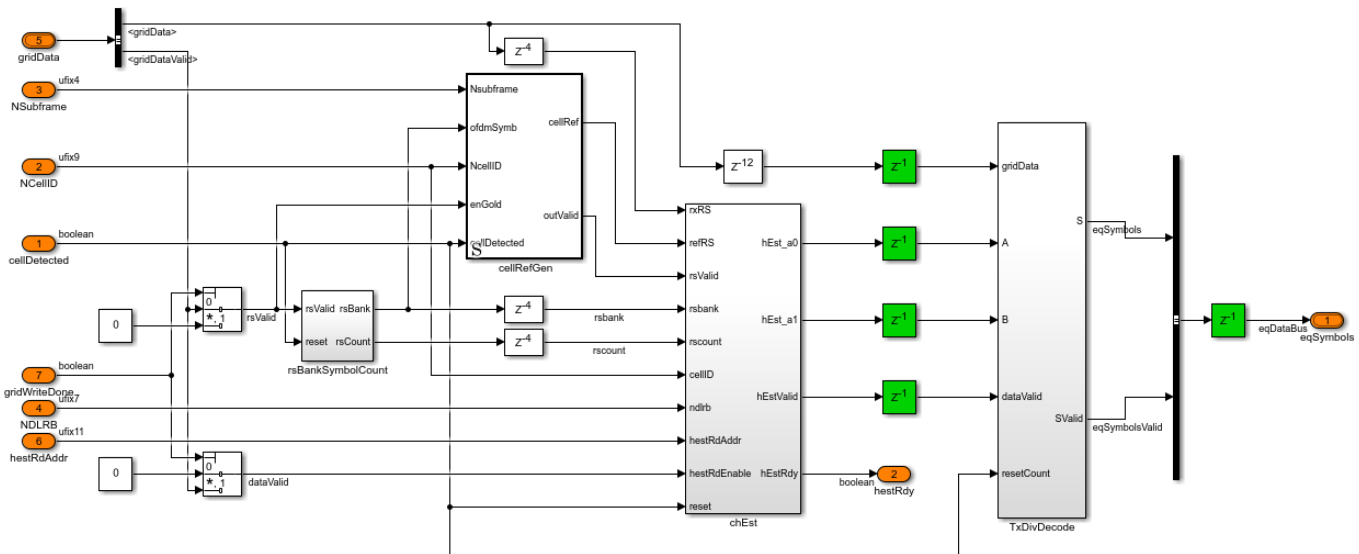
algorithm. **TxDivDecode** performs transmit diversity decoding to equalize the phase of the received data, using the channel estimates.

The channel estimator assumes the transmitter is using two antennas, generating a channel estimate for each antenna. For each antenna the channel estimator generates a single complex-valued channel estimate for each subcarrier of the subframe using the following algorithm:

- 1 Estimate the channel at each CRS resource element by comparing the received value to the expected symbol value (generated by **cellRefGen**).
- 2 Average these channel estimates across time (for the duration of the subframe) to generate a single complex-valued channel estimate for each subcarrier that contains CRS symbols.
- 3 Use linear interpolation to estimate the channel for subcarriers which do not contain CRS symbols.

The simple time average algorithm used for the channel estimation assumes low channel mobility. Therefore, the channel estimate may not be of sufficient quality to decode waveforms that were transmitted through fast fading channels. The algorithm also avoids using a division operation when calculating the channel estimate at each CRS. This means that the amplitude of the received signal will not be corrected, which is suitable for QPSK applications, but will not work for QAM, where accurate amplitude correction is required for reliable decoding.

Once the channel estimates are calculated for each of the transmit antennas they are used to equalize the *gridData* as it is read out from the **Resource Grid Memory**. **TxDivDecode** performs the inverse of the precoding for transmit diversity (as described in of TS 36.211 Section 6.3.4.3 [ 1 ]) and produces an equalized output signal, which is then passed to the **PBCH Decoder**.

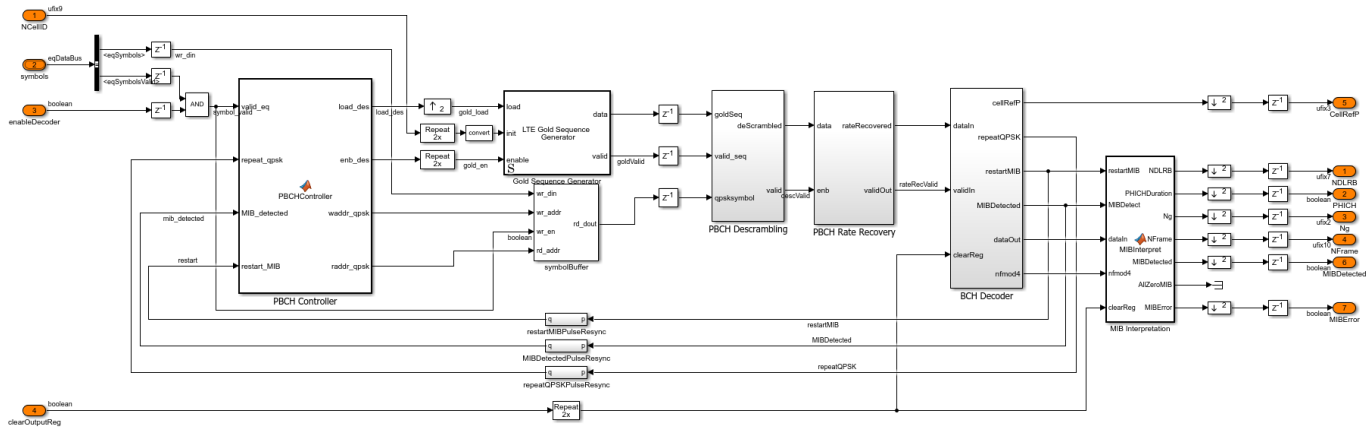


### PBCH Decoder

The **PBCH Decoder** performs QPSK demodulation, descrambling, rate recovery, and BCH decoding. It then extracts the MIB output parameters using the **MIB Interpretation** function block. These operations are equivalent to the `ltePBCHDecode` and `lteMIB` functions in the LTE Toolbox.



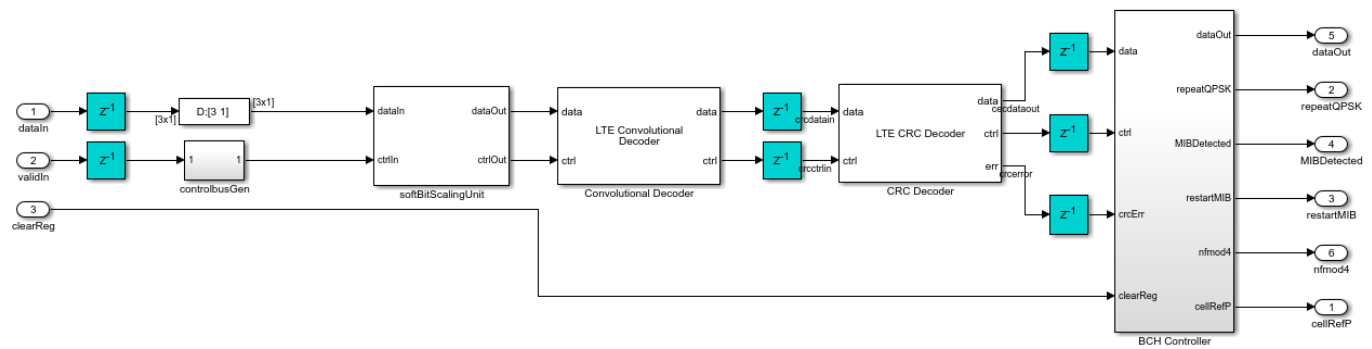
The PBCH Controller stores the equalized data in memory for iterative convolutional decoding attempts. The 4 attempts made at decoding the MIB correspond to the 4 repetitions of the MIB data per PBCH transport block.



### BCH Decoder

The **BCH Decoder** quantizes the soft decisions and then decodes the data using the LTE Convolutional Decoder and LTE CRC Decoder blocks. The recommended wordlength of soft decisions at the input to the convolutional decoder is 4 bits. However, the **BCH Decoder** block receives 20-bit soft decisions as input. Therefore the **softBitScalingUnit** block dynamically scales the data so that it utilizes the full dynamic range of the 4 bit soft decisions. The CRC decoder block is configured to return the full checksum mismatch value. The CRC mask, once checked against the allowed values, provides *cellRefP*; the number of cell-specific reference signal antenna ports at the transmitter. If the CRC checksum does not match one of the accepted values then MIB has not been successfully decoded and the PBCH Controller decides whether or not to initiate another decoding attempt.

When a MIB has been successfully decoded, the **MIB Interpretation** subsystem extracts and outputs the fields of the message.



### Performance Analysis

Quality of the input waveform is an important factor that impacts the decoding performance. Common factors that affect signal quality are multi-path propagation conditions, channel attenuation and interference from other cells. The quality of the input waveform can be measured using the *cellQualitySearch* function. This function detects LTE cells in the input waveform and returns a structure per LTE cell containing the following fields:

- `FrequencyOffset`: Frequency offset obtained by `lteFrequencyOffsets` function
- `NCellID`: Physical layer cell identity
- `TimingOffset`: Timing offset of the first frame in the input waveform
- `RSRQdB`: Reference Signal Received Quality (RSRQ) value in dB per TS 36.214 Section 5.1.3 [ 2 ]
- `ReportedRSRQ`: RSRQ measurement report (integer between 0 and 34) per TS 36.133 Section 9.1.7 [ 3 ]

Applying the `cellQualitySearch` function to the captured waveform `eNodeBWaveform.mat` used in `ltehdlMIBRecovery_init.m` returns the following report:

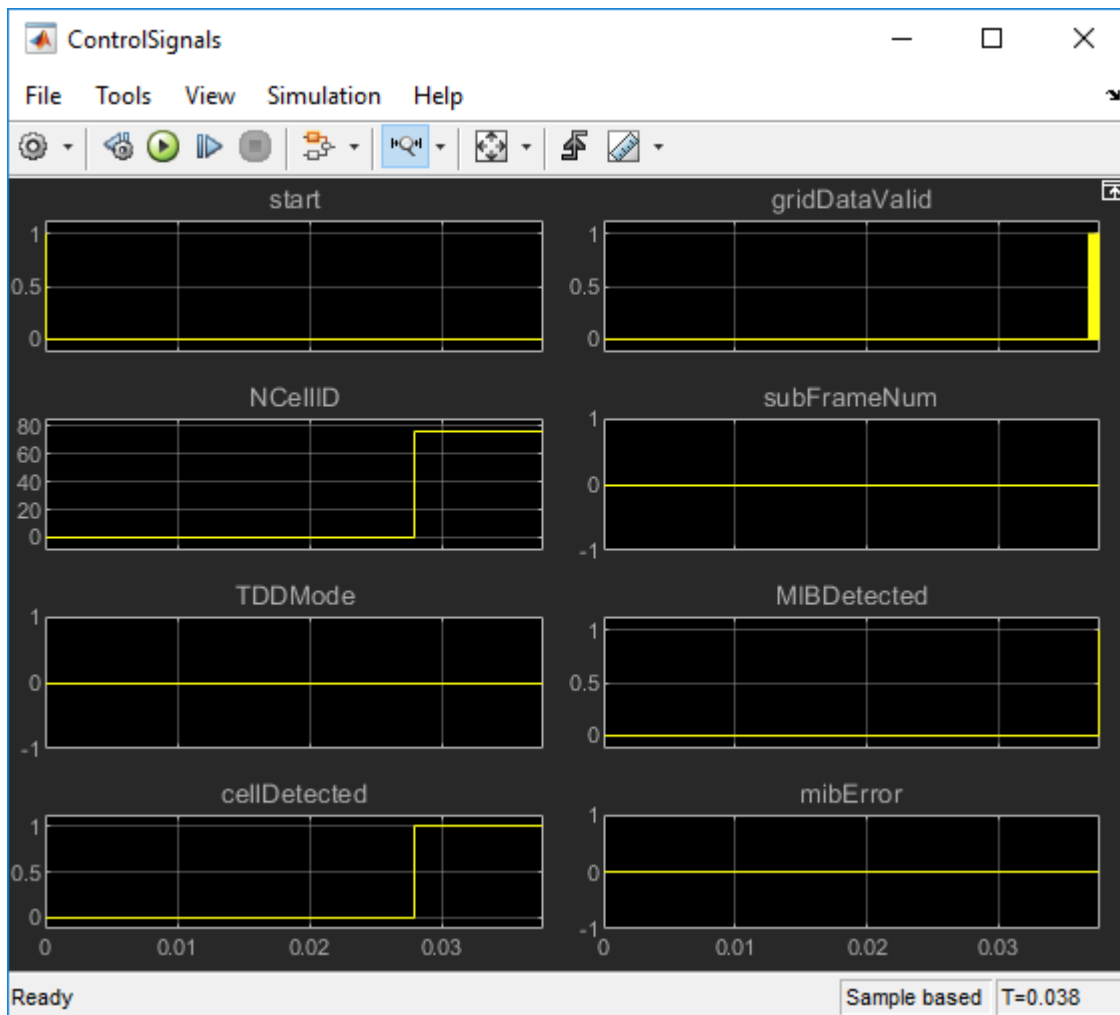
```
FrequencyOffset: 536.8614
NCellID: 76
TimingOffset: 12709
RSRQdB: -5.3654
ReportedRSRQ: 29
```

```
FrequencyOffset: 536.8614
NCellID: 160
TimingOffset: 3108
RSRQdB: -18.1206
ReportedRSRQ: 3
```

There are two cells in the captured waveform, one with cell ID 76 and one with cell ID 160. The cell with `NCellID = 76` has a much higher `ReportedRSRQ`, indicating that it is a stronger signal. In this example the Simulink model decodes the MIB for `NCellID = 76`.

### Results and Display

The scope below shows the key control signals for this example. After a pulse is asserted on the *start* signal the cell search process is started. Successful detection of a cell is indicated by the *cellDetected* signal. When the *cellDetected* signal is asserted the *NCellID* and *TDDMode* signal become active, indicating the cell ID number and whether the cell is using TDD (1) or FDD (0). After the cell has been detected the OFDM demodulator waits until subframe 0 of the next frame to start outputting the grid data, hence there is a gap between *cellDetected* going high, and grid data being output as indicated by the *gridDataValid* signal. When *gridDataValid* is first asserted *subFrameNum* will be zero, and will increment for subsequent subframes. The simulation stops on the *MIBDetected* or *mibError* signals being asserted.



Once MIB has been detected the *NDLRB*, *PHICH*, *Ng*, *nFrame*, and *CellRefP* signals all become active, indicating the key parameters of the cell. These parameters are displayed in the model, as they are static values when the simulation is stopped.

The following MIB information is decoded when decoding the captured waveform:

```
NCellID (Cell ID): 76
TDDMode (0 = FDD, 1 = TDD) : 0
NDLRB (Number of downlink resource blocks): 25
PHICH (PHICH duration) index: 0
Ng (HICH group multiplier): 2
NFrame (Frame number): 262
CellRefP (Cell-specific reference signals): 2
```

This indicates that the duplex mode used by the cell is FDD, the MIB was decoded in frame number 262, the PHICH duration is 'Normal' and the HICH group multiplier value is 'One'.

### HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the **HDL LTE MIB**

**Recovery** subsystem. Because the input waveform in this example contains at least 40 subframes to complete the cell search and MIB recovery, test bench generation takes a long time.

The **HDL LTE MIB Recovery** subsystem was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below. The design met timing with a clock frequency of 140 MHz.

Resource	Usage
Slice Registers	51582
Slice LUTs	29859
RAMB18	38
RAMB36	39
DSP48	134

For more information see “Prototype LTE Algorithms on Hardware” on page 2-12.

### References

- 1 3GPP TS 26.211 "Physical Channels and Modulation"
- 2 3GPP TS 36.214 "Physical layer"
- 3 3GPP TS 36.133 "Requirements for support of radio resource management"

### See Also

### Related Examples

- “LTE HDL Cell Search” on page 5-46
- “LTE HDL SIB1 Recovery” on page 5-63
- “LTE HDL PBCH Transmitter” on page 5-91

## LTE HDL PBCH Transmitter

This example shows how to implement an LTE transmitter Multiple Input Multiple Output (MIMO) design, including PSS, SSS, CRS, and MIB, optimized for HDL code generation.

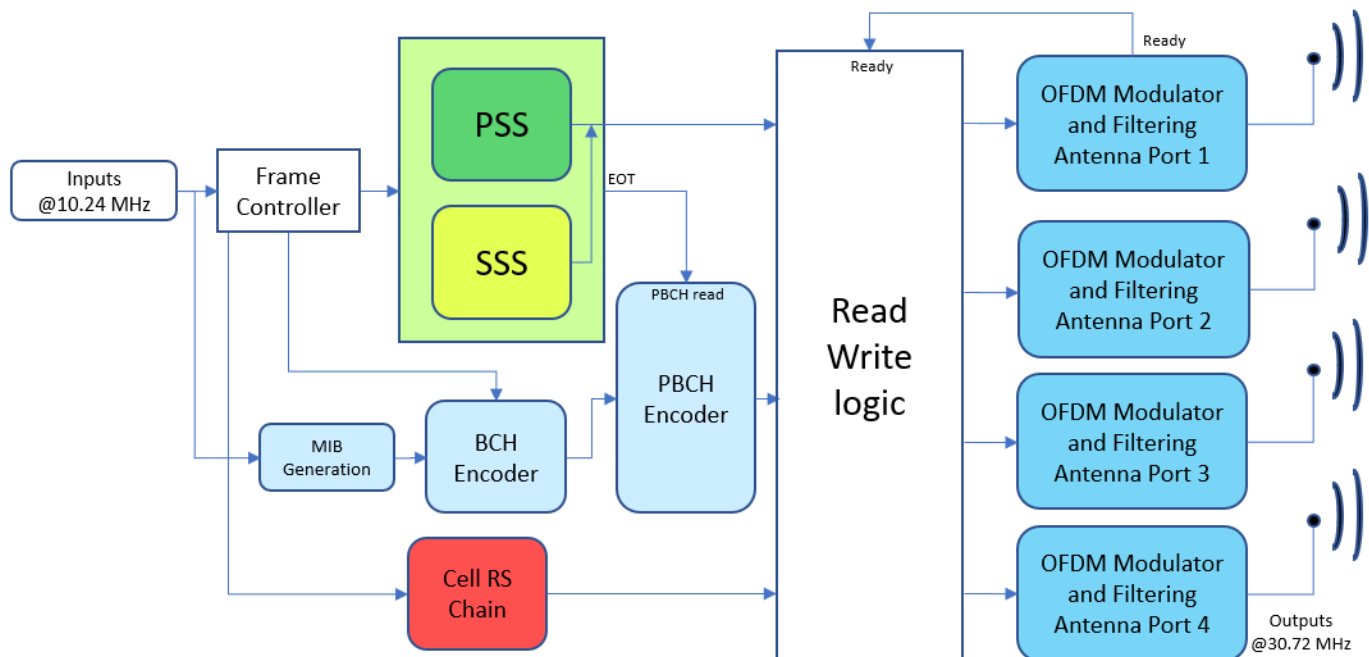
### Introduction

The model in this example generates a baseband waveform specified by 3GPP TS 36.211. The waveform includes the primary synchronization signal (PSS), secondary synchronization signal (SSS), cell-specific reference signals (CRS), and the master information block (MIB) for transmission through the physical broadcast channel (PBCH) for multiple antennas. The model supports dynamic change of NCellID and NDLRB. The MIMO transmitter design is optimized for HDL code generation and when implemented on an FPGA, it can be used to transmit MIMO signals in real time over the air. The MIMO design aids the decoding process in the presence of LTE fading channel. This example supports 1, 2, or 4 antennas and uses transmit diversity as specified in the [ 1 ].

The architecture presented in this example is extensible and allows for integration of additional physical transmission channels such as physical downlink control channel (PDCCH), physical downlink shared channel (PDSCH), physical control format indicator channel (PCFICH), and physical HARQ indicator channel (PHICH).

### Architecture and Configuration

This figure shows the LTE HDL Transmitter architecture with PSS, SSS, CRS, and PBCH transmission chains.



The input sampling rate is assumed to be at 10.24 MHz. PSS, SSS, PBCH, and CRS signals are generated in parallel, based on the input configuration. A single stream of PSS and SSS signals is used for all the antennas. Multiple streams of PBCH data are generated for multiple antennas through the layer mapping and precoding stages. Each antenna is associated with a corresponding LTE memory bank, which is sized to store one subframe of LTE data samples. These generated data

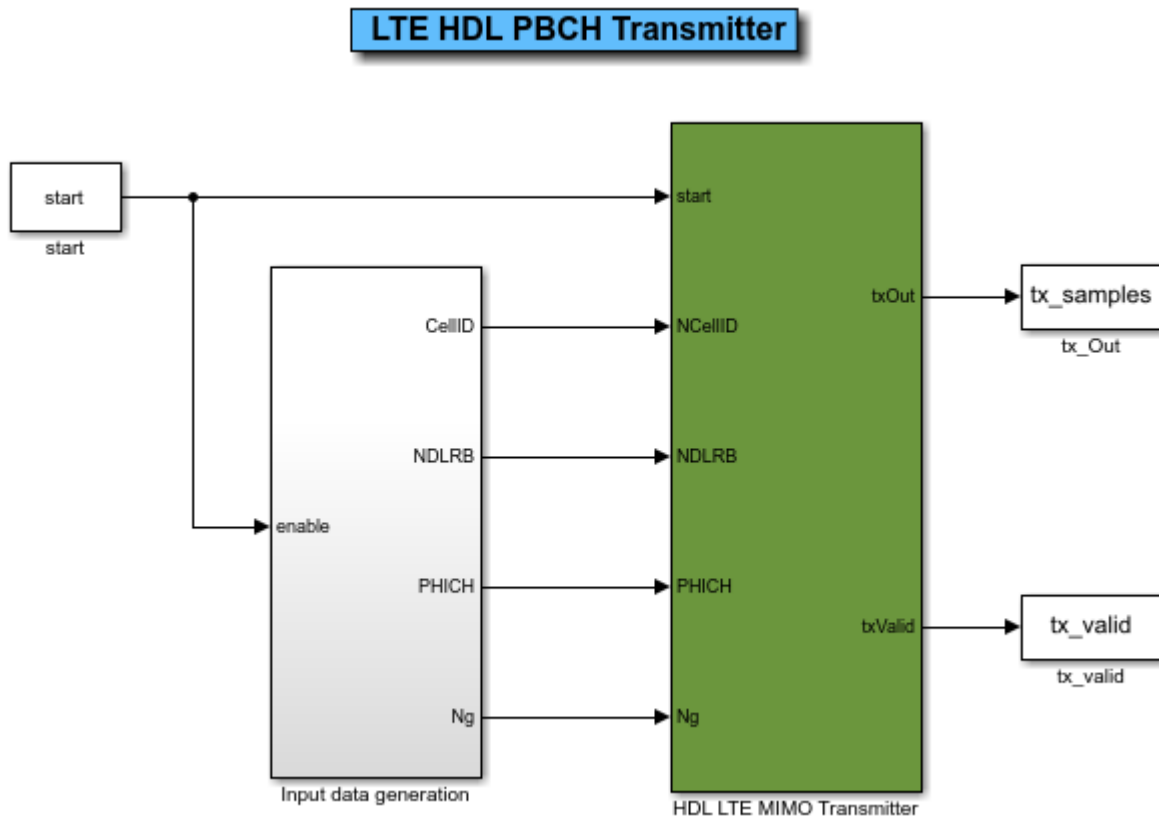
streams are written into LTE memory bank corresponding to indices generated, based on the output *ready* signal of LTE OFDM Modulator. Then, the data is read out of all LTE memory bank in parallel, modulated and transmitted on the antennas simultaneously. The LTE OFDM Modulator block uses a 2048-point FFT to support all NDLRBs.

In this example, the transmitter transmits LTE MIMO signals for the following configurations:

Property	Value
Duplex mode	FDD
CellRefP	1/2/4
Bandwidth	1.4 - 20 MHz
Cyclic prefix	Normal/Extended
Initial subframe	0
Initial frame	0
Ng	Sixth/Half/One/Two
PHICH duration	Normal/Extended

### Structure of Example Model

The top level structure of the **ltehdlTransmitter** model is shown below. You can generate HDL code for the **HDL LTE MIMO Transmitter** subsystem.



Input *start* is a pulse signal to trigger the transmission. You can configure other parameters, including *NDLRB*, *NCellID*, *Cyclic prefix*, *Ng*, *PHICH duration* and *CellRefP* in the workspace after

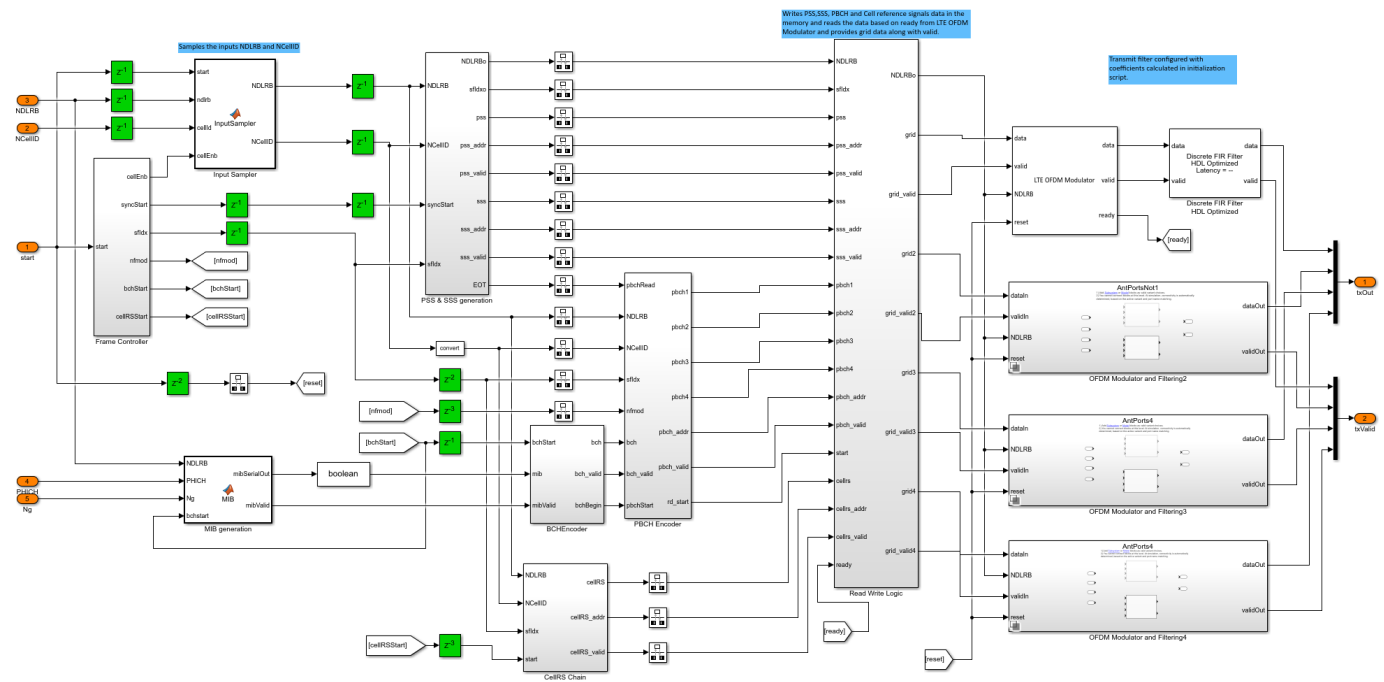
loading or opening the `ltehdlTransmitter.slx` model. The `ltehdlTransmitter_init.m` script is executed automatically by the model's `InitFcn` callback. This script configures the individual blocks in the **HDL LTE MIMO Transmitter** subsystem. The default transmitter configuration used by the `ltehdlTransmitter_init.m` script is:

```
enb.NDLRB = 6; % {6,15,25,50,75,100}
enb.CyclicPrefix = 'Normal'; % {'Normal','Extended'}
enb.Ng = 'Sixth'; % {'Sixth','Half','One','Two'}
enb.PHICHDuration = 'Normal'; % {'Normal','Extended'}
enb.CellRefP = 4; % {1,2,4}
tx_cellids = [390 89 501 231 500]; % {0 to 503}
outRate = 1; % {1,2}
TotalSubframes = 45; % {positive integer}
```

This default configuration can be changed to use other possible values for each variable, as noted in the comment on each line.

### HDL LTE MIMO Transmitter

The structure of the **HDL LTE MIMO Transmitter** subsystem is shown below. The **Frame Controller** controls the subframe and frame indices. The **Input Sampler** samples the inputs *NDLRB* and *NCellID* and then propagates the values to the subsequent blocks. The **PSS & SSS generation** generates PSS, SSS, and the corresponding memory address based on *NDLRB* and subframe index. The **MIB generation** block generates the serial MIB data. The **BCH Encoder** and **PBCH Encoder** generate information for PBCH channel and memory addresses for all the antennas. The **CellRS Chain** generates cell-specific reference signals and corresponding addresses for each antenna. The **Read Write Logic** writes and reads the grid data from each **LTE Memory Bank** and provides the data to the corresponding **LTE OFDM Modulator**. The **Discrete FIR Filter HDL Optimized** filters the modulated data using coefficients that are calculated based on the input configuration.

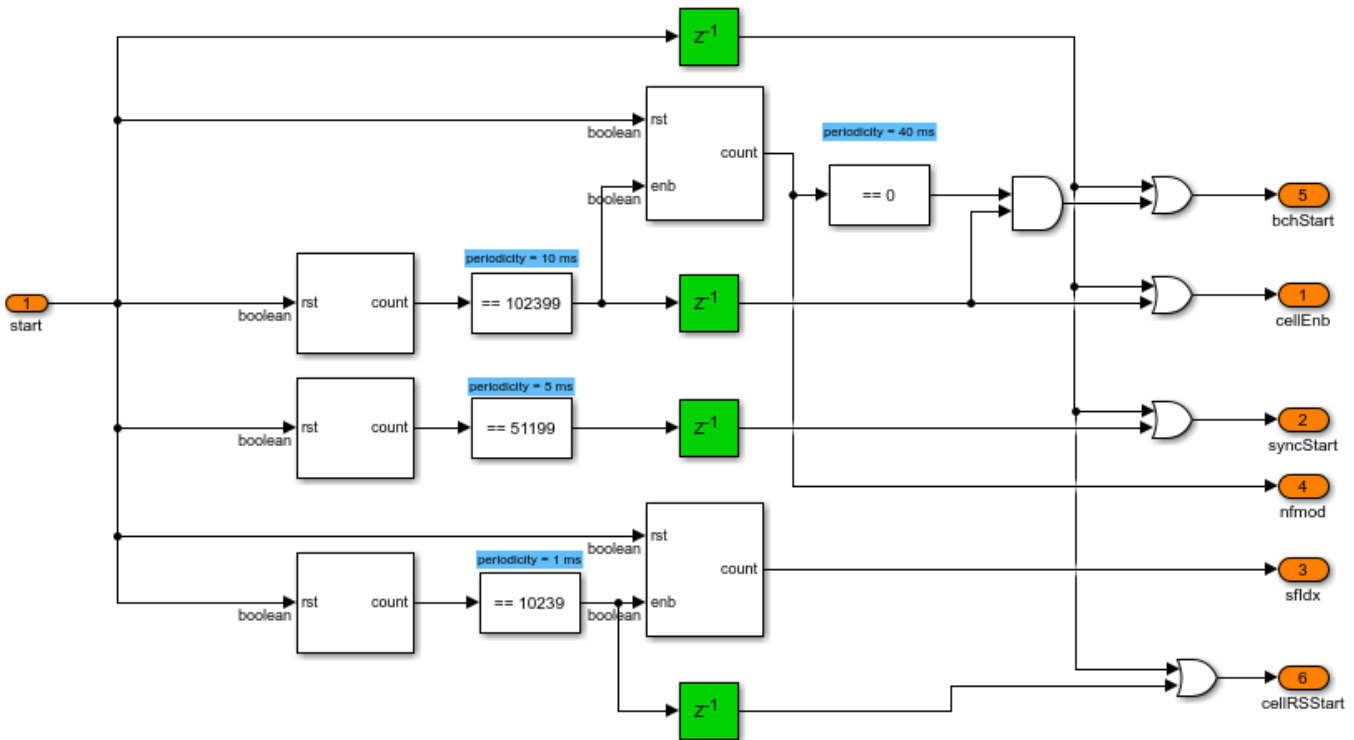


### Frame Controller

This subsystem assumes an input sampling rate of 10.24 MHz. It controls the subframe and radio frame boundaries by providing *cellEnb* signal to sample *NCellID*. It returns radio frame and subframe indices. It also provides *syncStart*, *bchStart*, and *cellRSStart* trigger signals to control the downstream blocks.



Assumes input @ 10.24 MHz. Provides 'syncStart', 'bchStart' and 'cellRSStart' trigger signals for PSS & SSS Chain, BCH Encoder and CellRS Chain subsystems. Also provides cellEnb to Input sampler to sample the input NCellID. Controls subframe and radio frame (rf mod 4) boundaries by providing its indices

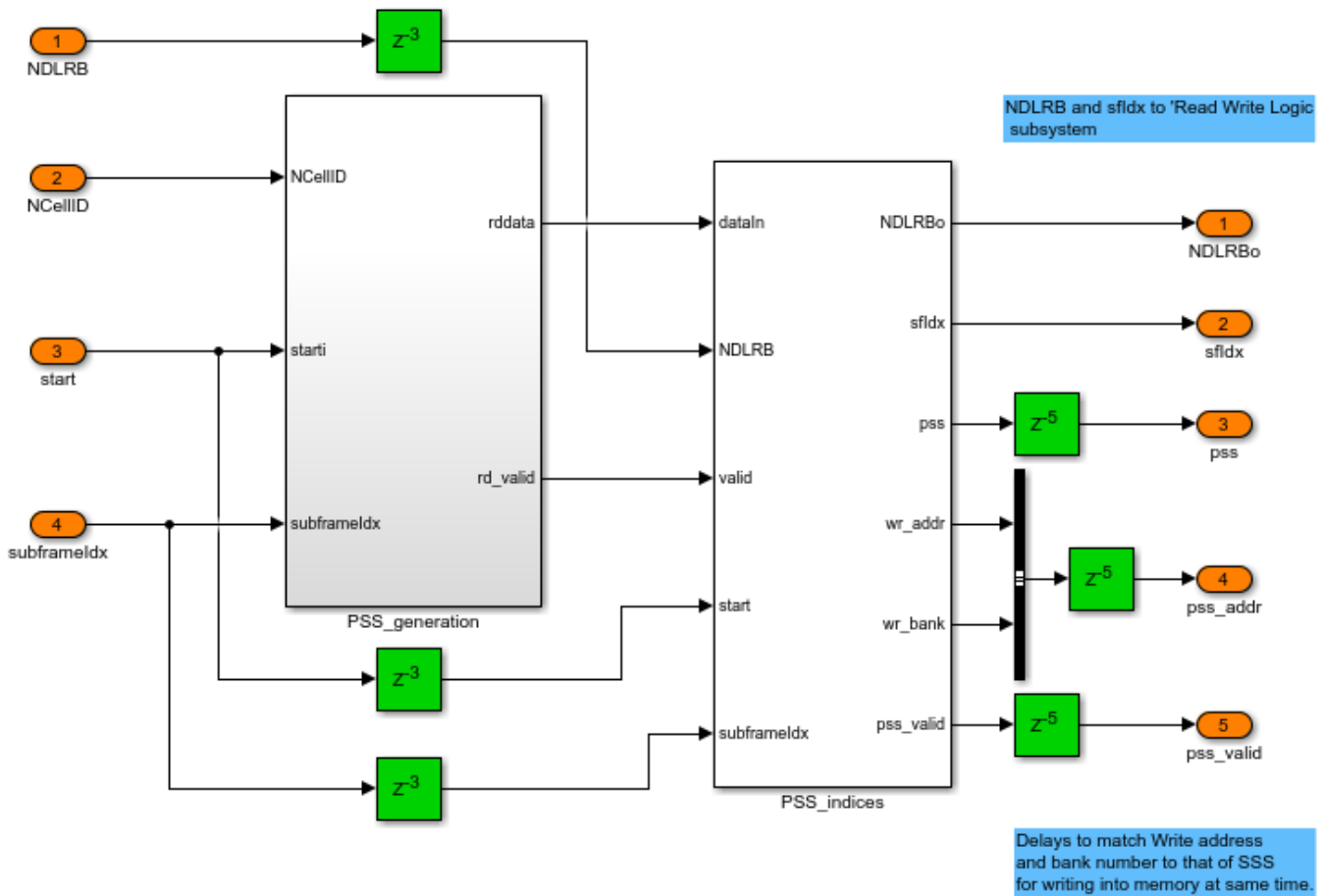


## PSS & SSS Generation

This subsystem generates the primary synchronization signal (PSS), secondary synchronization signal (SSS), and respective write addresses for LTE Memory Bank based on inputs *NDLRB* and *NCellID*. *syncStart* triggers the generation of PSS and SSS. The PSS and SSS occupy the same central 62 subcarriers of two OFDM symbols in a resource grid [ 1 ]. This subsystem generates both the signals and their corresponding addresses at the same time, so that a single stream of both PSS and SSS can be written to all the LTE Memory Banks corresponding to each antenna simultaneously.

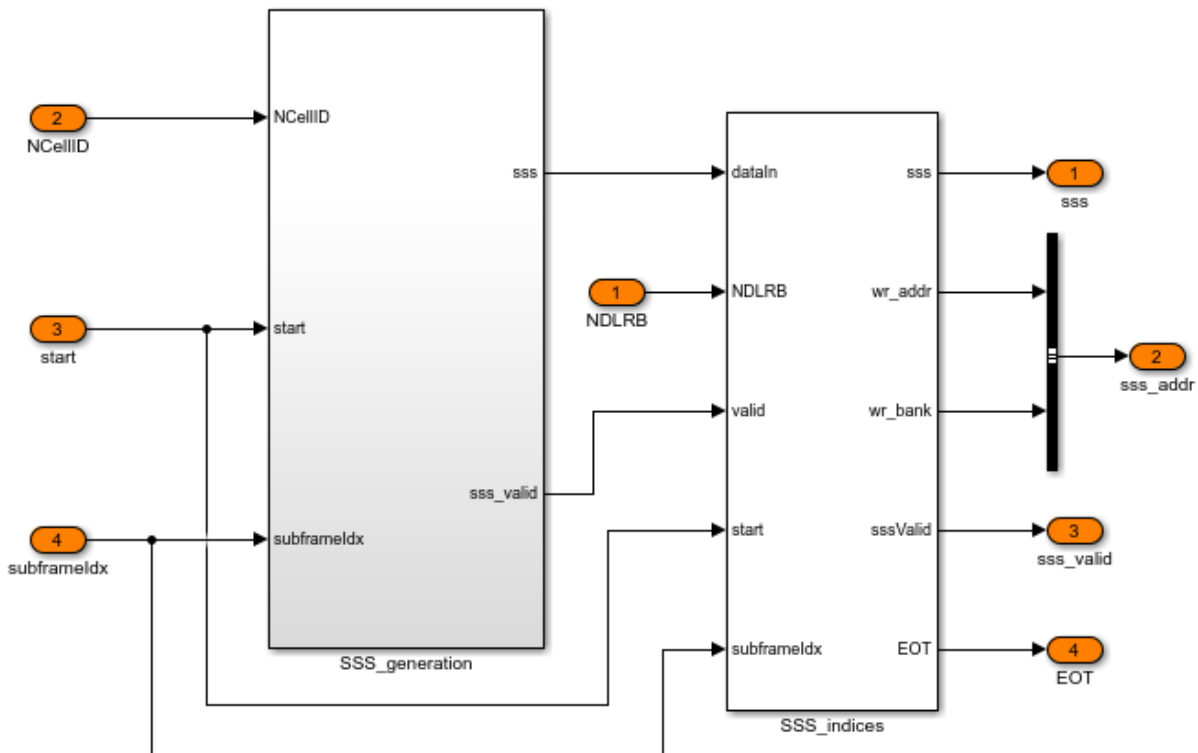
The PSS sequence is generated from a frequency-domain Zadoff-Chu sequence [ 1 ]. The Zadoff-Chu root sequence index depends on *NCellID2*, which is derived from *NCellID*. There are three possible *NCellID2* values, so all possible PSS sequences are precalculated and stored in *PSS\_LUT*.

- **PSS\_generation:** Determines *NCellID2* and reads the corresponding PSS sequence out of *PSS\_LUT* sequentially.
- **PSS\_indices:** Computes the memory addresses required to write PSS data into LTE Memory Bank. This subsystem is equivalent to the LTE Toolbox™ function *ltePSSIndices*.



The SSS sequence is an interleaved concatenation of two 31-bit length binary sequences. The concatenated sequence is scrambled with a scrambling sequence given by PSS. The combination of these sequences differs between subframe 0 and subframe 5 [ 1 ]. The indices  $m_0$  and  $m_1$  are derived from the physical-layer cell identity group,  $N_{CellID1}$  [ 1 ]. These indices and the sequences  $s(n)$ ,  $c(n)$ , and  $z(n)$  are calculated and stored in  $m_0\_LUT$ ,  $m_1\_LUT$ ,  $S\_LUT$ ,  $C\_LUT$ , and  $Z\_LUT$  respectively.

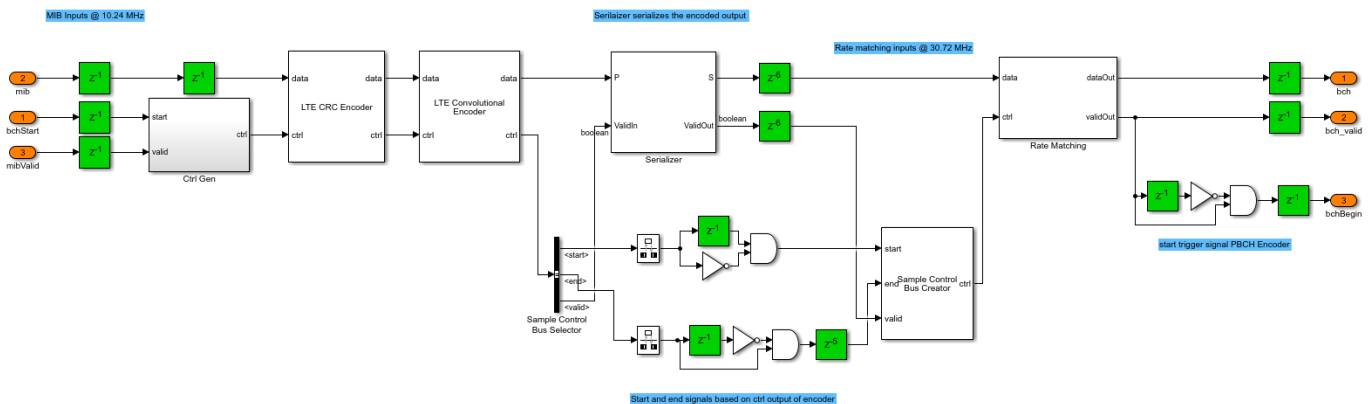
- **SSS\_generation:** Computes  $m_0$  and  $m_1$  based on the  $N_{CellID}$  and calculates indices required for sequences  $s(n)$ ,  $c(n)$ , and  $z(n)$  based on the subframe index. Generates SSS sequence as specified in [ 1 ].
- **SSS\_indices:** Computes memory addresses required to write SSS data into LTE Memory Bank. This subsystem is equivalent to the LTE Toolbox™ function `lteSSSIndices`.



'pbchread' enable reading of PBCH data during nfm0d = 0 (radio frame index mod 4)

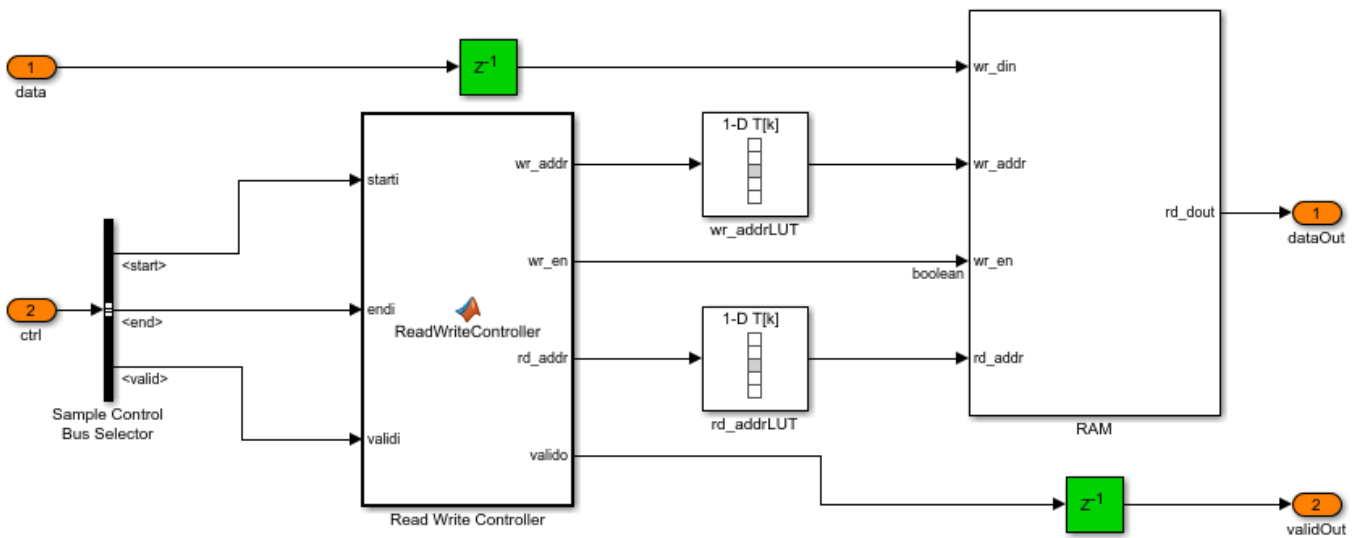
### BCH Encoder

Broadcast Channel (BCH) processes the MIB information arriving to the block in the form of a maximum of one transport block for every transmission time interval (TTI) of 40 ms. The block implements the following coding steps.



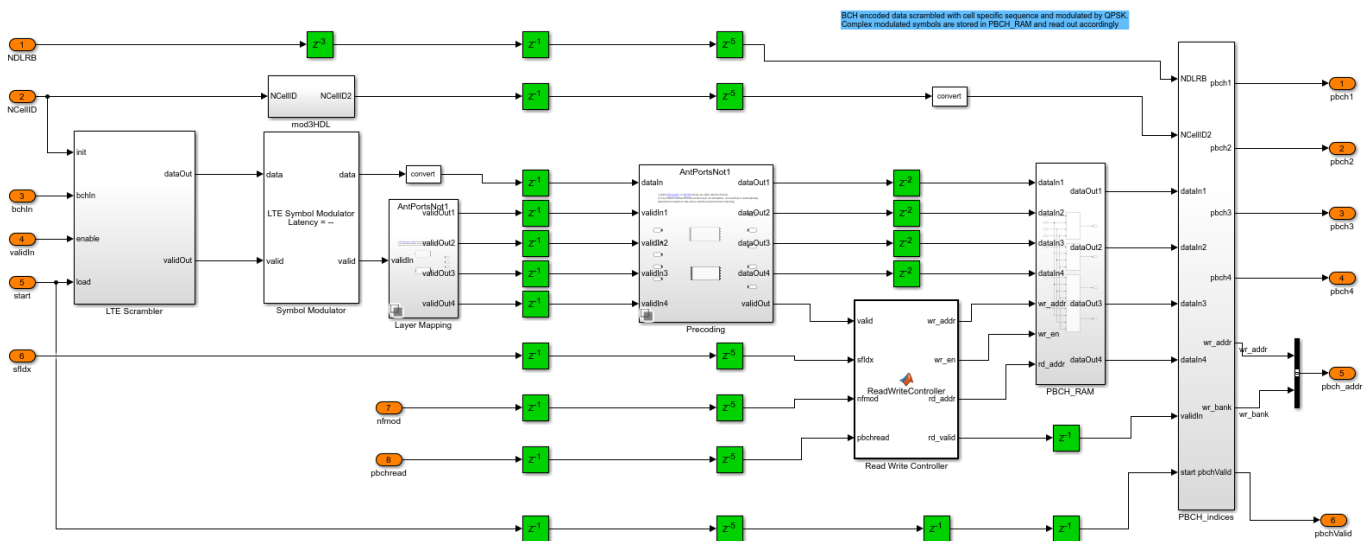
- CRC Encoding:** The entire transport block is used to calculate the CRC parity bits for a polynomial specified in [ 2 ]. The parity bits are then appended to the transport block. After appending, CRC bits are scrambled according to the transmit configuration. The LTE CRC Encoder block uses the CRC mask set by the `ltehdlTransmitter_init.m` script based on the input configuration.

- Channel Coding:** The LTE Convolutional Encoder block encodes the information bits using tail-biting convolutional code with constraint length 7, and polynomial  $G_0 = 133, G_1 = 171, G_2 = 165$  in octal. Because the coding rate of the encoder is 1/3, the coded bits are then serialized using a Serializer1D (HDL Coder) block and control signals are resampled to 30.72 MHz ( $3 * 10.24$  MHz).
- Rate Matching:** The coded bits are interleaved, followed by selection of bits for a particular length using an interleaved address [ 2 ]. For broadcast channel, because the length of the MIB is constant, interleaved write and read addresses are precalculated and stored in `wr_addrLUT` and `rd_addrLUT` respectively. Once all serialized coded bits have been written into interleaved addresses of RAM, the bits are read back using interleaved read addresses.



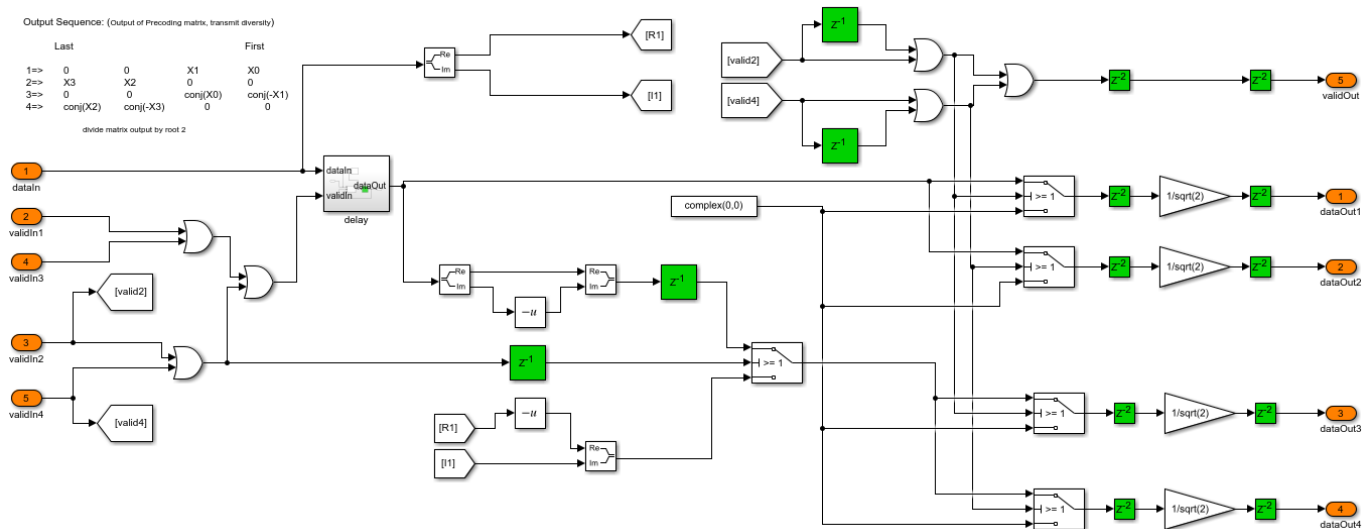
### PBCH Encoder

The physical broadcast channel processes the coded bits in the following steps.

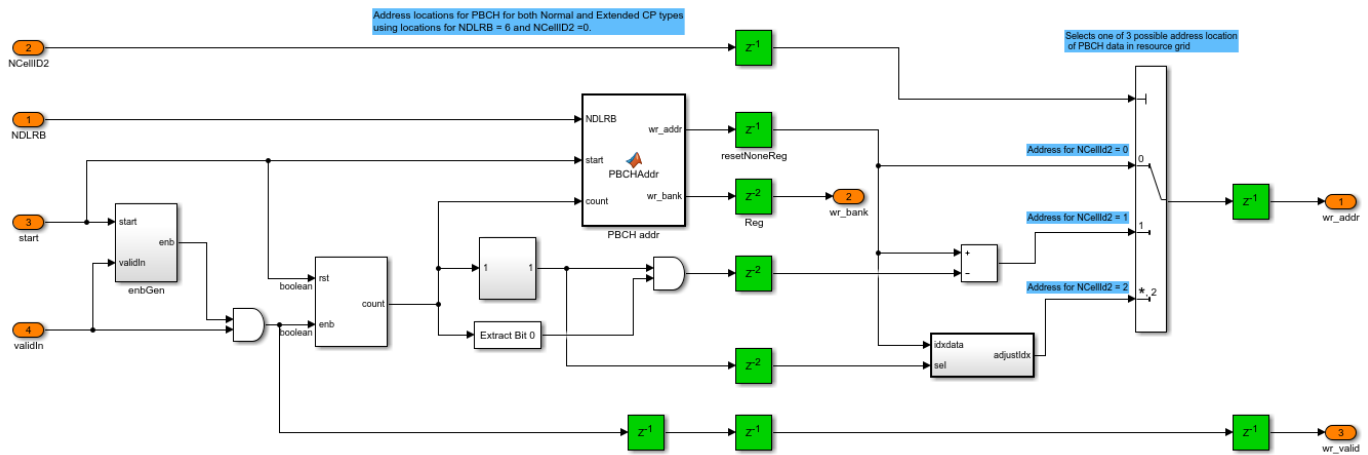


- **Scrambling:** Coded bits from **BCH Encoder** are scrambled with a cell-specific sequence using a LTE Gold Sequence Generator block. The sequence is initialized with NCellID in each radio frame( $n_f$ ) fulfilling  $n_f \bmod 4 = 0$ . The generated cell-specific sequence is scrambled with the input coded bits.
- **QPSK Mapping:** The modulation scheme specified for PBCH channel is QPSK [ 1 ]. The LTE Symbol Modulator block generates complex-valued QPSK modulation symbols.
- **Layer Mapping:** Three subsystems are defined for the layer mapping. These subsystems are placed inside a variant subsystem. Based on the number of antennas used in the input configuration `enb.CellRefP`, the `ltehdlTransmitter_init.m` script selects one of the three subsystems in the variant subsystem. This **Layer Mapping** block separates the input streaming samples into 1, 2, or 4 sequences based on the number of antennas used. The input is streamed out without any processing for a single antenna. For multiple antennas, this block generates a valid signal for each antenna. Only one of the valid signals will be high for each input sample.
- **Precoding:** This block also uses variant subsystem to process input samples differently based on the number of antennas in the transmitter configuration. For `enb.CellRefP` set to 1 the input is streamed out without any processing. For `enb.CellRefP` set to 4 (or 2), every four (or two) consecutive samples  $X_0, X_1, X_2, X_3$  (or  $X_0, X_1$ ) are processed to generate four (or two) streams of 4 (or 2) samples each in four (or two) time instants.

The subsystem shown generates the output sequence for 4 antennas as specified in [ 1 ].



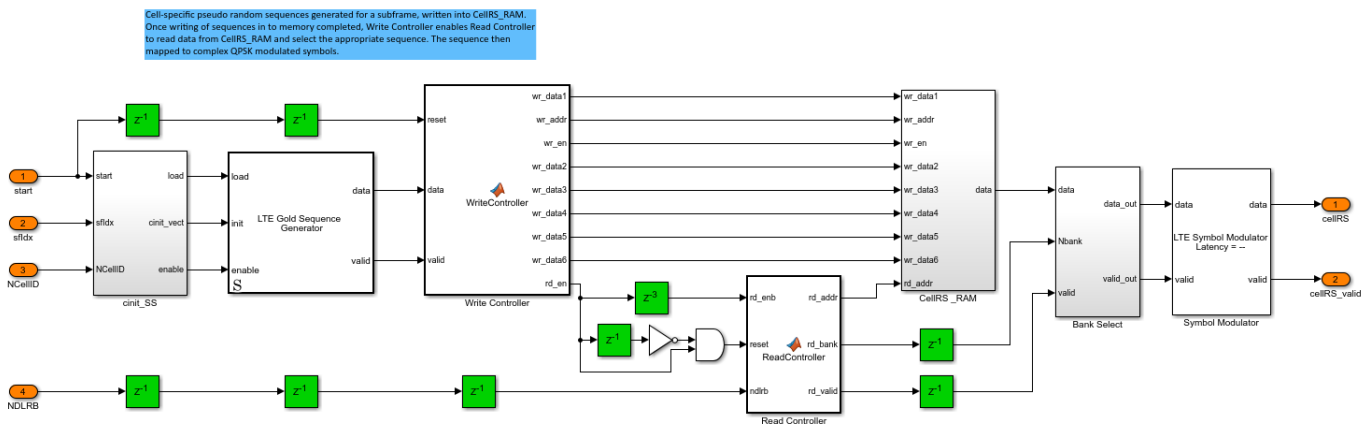
- **Memory:** Complex modulated symbols corresponding to the physical broadcast channel for the initial radio frame are stored in `PBCH_RAM`. For four consecutive radio frames, the number of bits to be transmitted on the physical broadcast channel is 1920 for normal cyclic prefix and 1728 for extended cyclic prefix. The Read Write Controller controls read and write addresses based on  $n_f \bmod 4$ , since the periodicity of the broadcast channel (BCH) is 40 ms.
- **PBCH Indexing:** Computes the memory addresses required to write PBCH data into LTE Memory Bank. The `PBCH_indices` subsystem is equivalent to the LTE Toolbox™ function `ltePBCHIndices`.



### CellRS Chain

The cell-specific reference sequence is complex modulated values of a pseudo-random sequence as defined in [ 1 ]. The pseudo-random sequence generator is initialized with *Cinit* at the start of each OFDM symbol, as specified in [ 1 ].

- CellRS\_generation:** Input *cellRSStart* triggers the generation of CRS signals. Since the CRS is available in six OFDM symbols (four OFDM symbols in antenna port 0 and port 1, and two OFDM symbols in antenna port 2 and port 3) of a single subframe, this subsystem calculates a 6-element *Cinit* vector for every subframe. The LTE Gold Sequence Generator block is initialized with vector *Cinit* to represent multiple channels and provides six different cell-specific pseudo-random sequences. The Write Controller controls writing of these sequences into six memory banks in *CellRS\_RAM*. It also returns *rd\_en*, which enables reading data out of *CellRS\_RAM*. The Read Controller controls reading of CRS data. It reads six OFDM symbols if four antennas are used, and reads only 4 OFDM symbols if one or two antennas are used. It returns *rd\_bank* and *rd\_valid* signals to select an appropriate symbol for the six/four OFDM symbols. The sequence is then mapped to complex QPSK modulated symbols.
- CellRS\_indices:** This subsystem computes the addresses for each **LTE Memory Bank** required to write CRS data. It is equivalent to the LTE Toolbox™ function `lteCellRSIndices`.



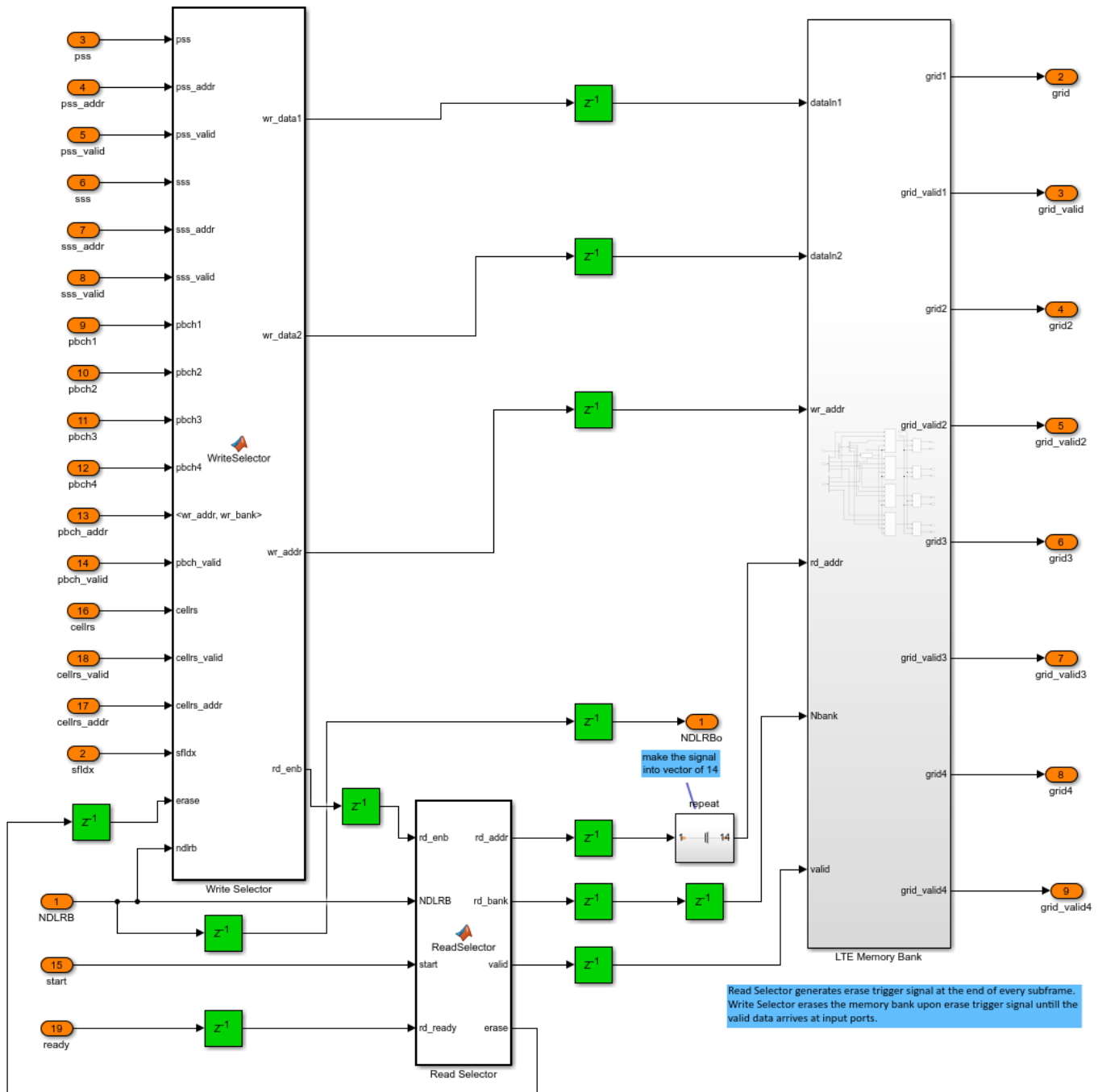
### Read Write Logic

The **Read Write Logic** subsystem contains a Write Selector, Read Selector, four LTE Memory Banks with a Grid Bank Select associated with each of the LTE Memory Bank. The LTE Memory Bank storage capacity is one subframe of complex modulated symbols at the largest supported LTE bandwidth (20 MHz). Each LTE Memory Bank can store 14 x 2048 x 16-bit complex values, that is, 14 OFDM symbols, each containing 2048 complex values.

The Write Selector writes subframes of data into the memory banks. The PSS and SSS occupy central subcarriers. A single stream of PSS and SSS data is used for all the antennas. The PBCH data consists of multiple streams corresponding to each antenna port. The CRS data generated is mapped to the grid based on the four addresses generated for each **LTE Memory bank** in **CellRS\_indices** block. The Write Selector first writes PSS and SSS simultaneously into corresponding locations in all LTE Memory Banks. Then, it writes PBCH data and CRS data into the corresponding LTE Memory Banks and returns *rd\_enb* to indicate that the write is complete.

The Read Selector reads the samples from each **LTE Memory Bank** based on *rd\_enb* and *ready* from the LTE OFDM Modulator block. Each LTE Memory Bank returns a 14 element vector corresponding to a single subcarrier. The **Grid Bank Select** selects the appropriate sample from the 14 element vector to form the resource grid output for each antenna.

Since the scope of this example is limited to PSS, SSS, CRS, and PBCH transmission, all the LTE Memory Banks are erased at the start of every subframe, before writing new data into the memory.



### OFDM Modulation and Filtering

Grid data from LTE Memory Bank is OFDM-modulated using the LTE OFDM Modulator block with 'Output data sample rate' parameter set to 'Match output data sample rate to NDLRB'. The modulated data is filtered using a Discrete FIR Filter HDL Optimized (DSP System Toolbox) block with coefficients generated at a sampling rate corresponding to the NDLRB. Variant subsystems control the number of OFDM modulators and FIR filters used based on the number of antennas, which reduces the resource utilization when a single antenna is used.



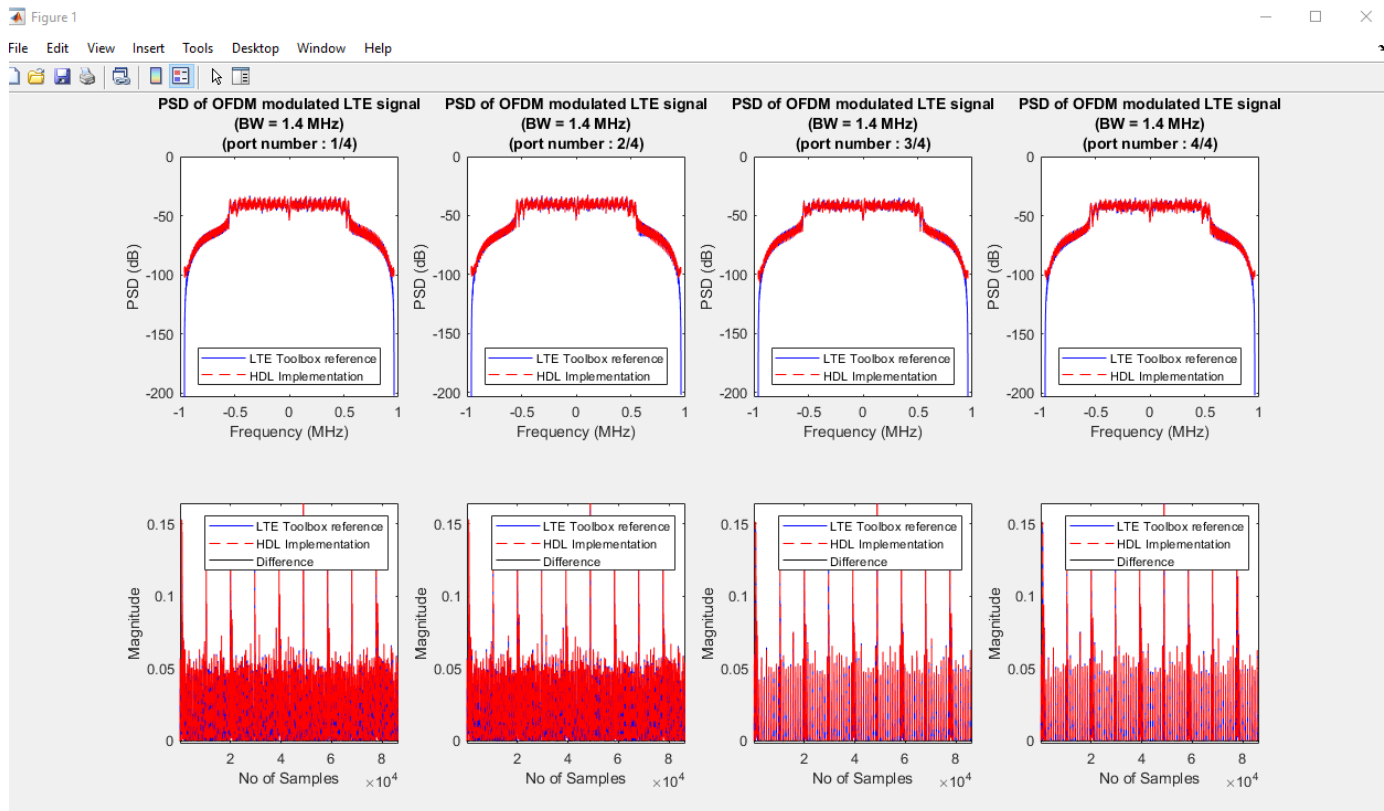
## Verification and Results

After running the simulation, the `ltehdlTransmitter_PostSim.m` script is executed automatically by the `StopFcn` callback of the model. In this example, the transmitter output is verified by the following methods:

### Verification of model's transmitted signal:

The transmitter output signal in this model is cross-verified with a reference transmitter signal that is generated using LTE Toolbox™ functions by the following two subplots for each antenna.

- 1 The first subplot shows the Power Spectral Density (PSD) output of the filtered data. The result is compared with the PSD of the reference output signal generated using LTE Toolbox™. This comparison shows the equivalence of the two signals. The figure shows a transmission bandwidth of  $BW = 1.4\text{MHz}$ .
- 2 The second subplot shows the absolute-value of the transmitted waveform. The result is plotted on top of the absolute-value of the reference transmitter signal generated using LTE Toolbox™. The plot also shows the difference between the samples obtained through HDL implementation and the reference signal. This comparison shows the minimal error between the two transmitter signals.



### Cell Search & MIB Decoding Results:

The valid samples of the transmitter output signal are stored to the workspace variable `txSamples`. These samples are passed through an LTE fading channel to create the receiver input signal, `rxSamples`. The `lteFadingChannel` (LTE Toolbox) function models the LTE fading channel.

This example uses the following channel configuration:

```

chcfg.NRxAnts = 1;
chcfg.MIMOCorrelation = 'Medium';
chcfg.NormalizeTxAnts = 'On';
chcfg.DelayProfile = 'EPA'; % {'off','EPA'}
% The below model configuration exist only if Delay profile is not set
% to 'off'.
chcfg.DopplerFreq = 5;
chcfg.SamplingRate = 30.72e6;
chcfg.InitTime = 0;
chcfg.NTerms = 16;
chcfg.ModelType = 'GMEDS';
chcfg.NormalizePathGains = 'On';
chcfg.InitPhase = 'Random';
chcfg.Seed = 1;

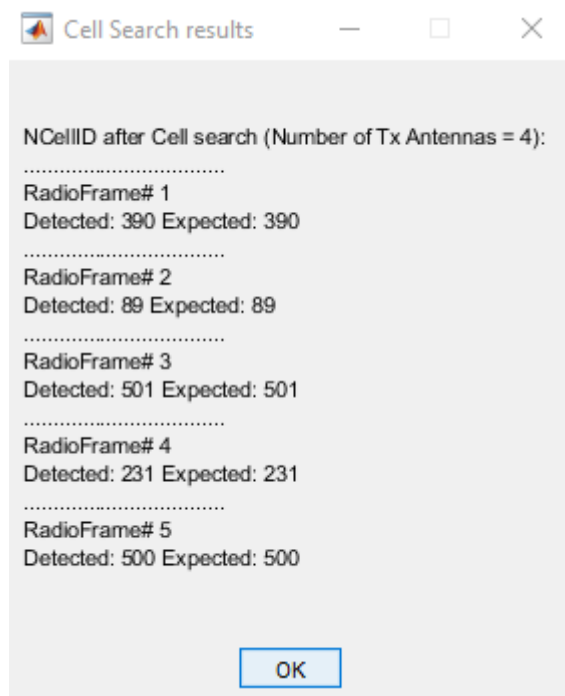
```

To create a fading-free channel, set the `chcfg.DelayProfile` to 'off' in the `ltehdlTransmitter_PostSim.m` script.

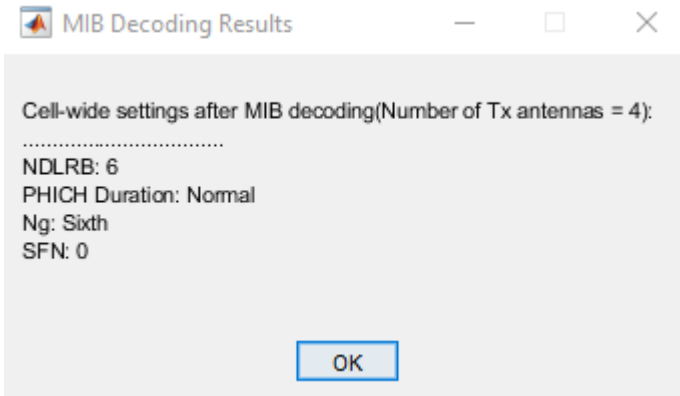
This channel configuration works with the default `enb` structure, and supports changes only in the `enb.PHICHDuration` and `enb.Ng` fields.

The following figures show the results of the cell search and MIB decoding of the channel output, `rxSamples`, using LTE toolbox™ functions. These figures verify the transmitter performance and compare the HDL transmitter implementation against the input configuration defined in `tx_cellids` and `enb`.

- NCellID after Cell Search: Displays the LTE cell search results performed on the fading channel output.



- Cell-wide settings after MIB decoding: Displays the fields of MIB after MIB decoding - NDLRB, Ng, PHICH duration, and System Frame Number (SFN) performed on the fading channel output.

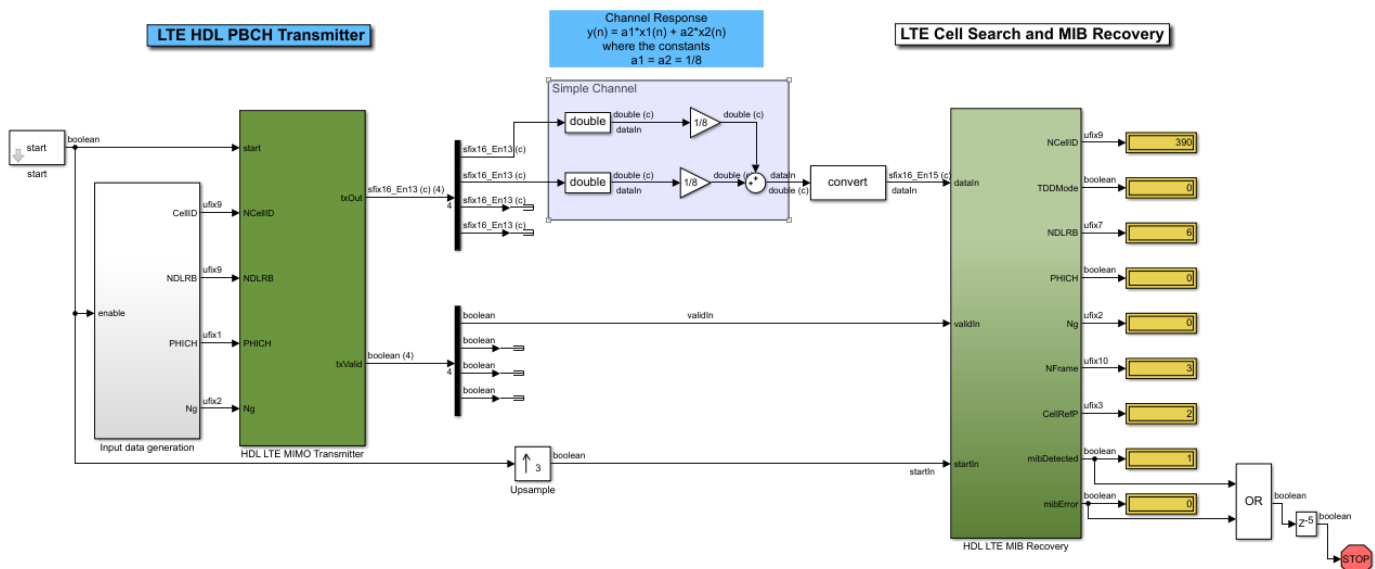


The example model does not support simulation in rapid accelerator mode.

### Validation with Cell Search and MIB Recovery Example

You can verify the **LTE HDL PBCH Transmitter** example by connecting it to the “LTE HDL MIB Recovery” on page 5-80 example model and checking that the output of the transmitter is decoded correctly. To make the transmitter model compatible with the receiver model, make these changes to the transmitter:

- Set the outRate = 2 (default value 1) before running the model. This will set the output rate of each **LTE OFDM Modulator** and generate the fir filter coefficients associated with each antennas.
- Set the enb.CellRefP = 2 (default value 4) before running the model.
- Use the same NCellID for all radio frames in the transmission. i.e. set tx\_cellids to a scalar value in the range 0-503.



The figure shows the **HDL LTE MIMO Transmitter** and **HDL LTE MIB Recovery** subsystems connected together. It also shows the result of simulating the model. The display blocks show the CellID and MIB fields (NDLRB, Ng, PHICH duration and System Frame Number (SFN)) that the receiver decoded from the output of the **HDL LTE MIMO Transmitter** subsystem.

You can also verify the design without using a channel by terminating the output from the second antenna and bypassing the channel system with the output from the first antenna.

### HDL Code Generation

To check and generate HDL for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the **HDL LTE MIMO Transmitter** subsystem. Because the `stopTime` in this example depends on `TotalSubframes`, the test bench generation time depends on the `TotalSubframes`.

The **HDL LTE MIMO Transmitter** subsystem is synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below.

Resources	No. of antennas used = 1	No. of antennas used = 2	No. of antennas used = 4
Slice Registers	12788	23839	45788
Slice LUT	11984	22220	42880
RAMB36	41	82	164
RAMB18	11	21	41
DSP	49	93	177
Max. Frequency (MHz)	210.08	206.39	204.00

### References

- 1 3GPP TS 36.211 "Physical channels and modulation".
- 2 3GPP TS 36.212 "Multiplexing and channel coding".

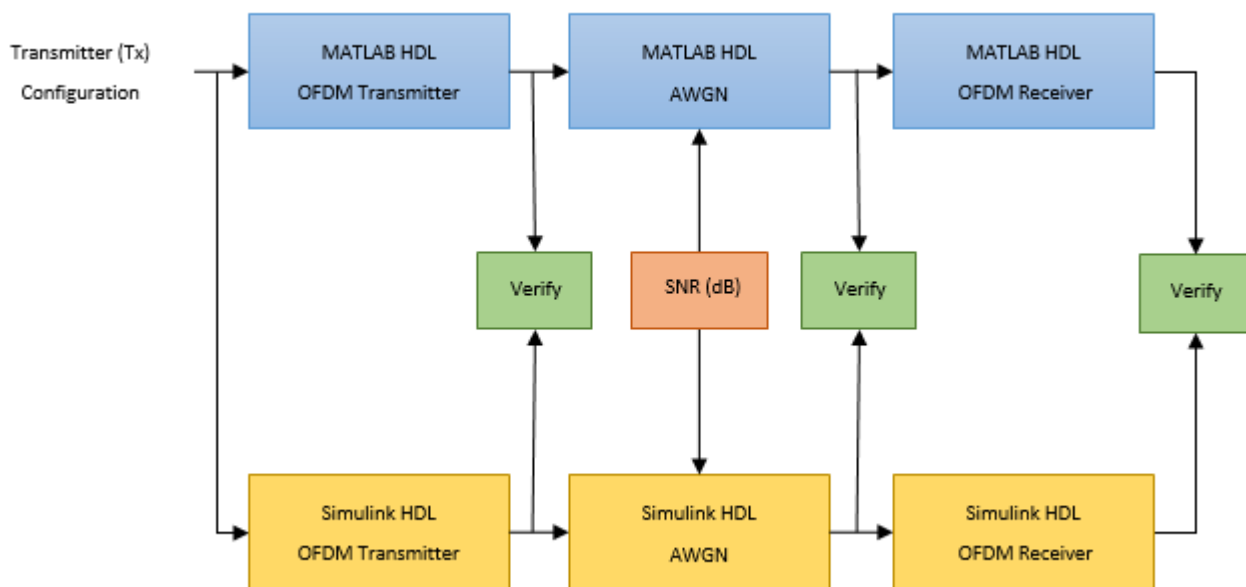
### See Also

### Related Examples

- "LTE HDL Cell Search" on page 5-46
- "LTE HDL MIB Recovery" on page 5-80
- "LTE HDL SIB1 Recovery" on page 5-63

## HDL OFDM MATLAB References

This example shows how to model OFDM transmitter, additive white Gaussian noise (AWGN), and OFDM receiver hardware algorithms in MATLAB as a step towards developing a Simulink® HDL implementation. The HDL OFDM MATLAB® References example bridges the gap between a mathematical algorithm and its hardware implementation. This example provides MATLAB references of the HDL OFDM Transmitter, HDL AWGN, and HDL OFDM Receiver algorithms that you can implement on hardware. You can use these MATLAB references to generate test vectors for verifying the HDL implementation of these Simulink models, “HDL OFDM Transmitter” on page 5-121, “HDL Implementation of AWGN Generator” on page 4-44 and “HDL OFDM Receiver” on page 5-136.



### HDL OFDM Transmitter MATLAB Reference

This section describes the MATLAB reference of HDL OFDM Transmitter.

This MATLAB reference accepts a modulation order, code rate index, number of frames, and data bits to be transmitted as a `txParam` structure or array of structures. `txParam` has these fields.

- `modOrder` — Specify 2, 4, 16, or 64 for 'BPSK', 'QPSK', '16QAM', and '64QAM', respectively. The default value is 4 ('QPSK').
- `codeRateIndex` — Specify 0, 1, 2, or 3 for the rates '1/2', '2/3', '3/4', and '5/6' respectively. The default value is 0 ('1/2').
- `numFrames` — Specify a positive integer. The default value is 5.
- `txDataBits` — Specify binary values in a row or column vector of length `trBlkSize * txParam.numFrames`. The default is a column vector containing randomly generated binary values of length `trBlkSize * txParam.numFrames`.

Calculate the transport block size (`trBlkSize`) by using these parameters.

- numSubCar — Number of subcarriers per symbol
- pilotsPerSym — Number of pilots per symbol
- numDataOFDMSymbols — Number of data OFDM symbols
- bitsPerModSym — Number of bits per modulated symbol
- codeRate — Punctured code rate
- dataConvK — Constraint length of the convolutional encoder
- dataCRCLen — CRC length

```
trBlkSize =
((numSubCar-pilotsPerSym)*numDataOFDMSymbols*bitsPerModSym*codeRate)
-(dataConvK-1)-dataCRCLen
```

For example, to generate a time-domain OFDM transmitter waveform of 5 frames with a modulation scheme of 16QAM and code rate of 1/2 using random data bits in the transport block, format the inputs as structure.

```
txParam.modOrder = 16; % Modulation order corresponding to 16-QAM
txParam.codeRateIndex = 0; % Code rate index corresponding to 1/2
txParam.numFrames = 5; % Number of frames to be generated

% Calculate transport block size (trBlkSize) using parameters
numSubCar = 72; % Number of subcarriers per symbol
pilotsPerSym = 12; % Number of pilots per symbol
numDataOFDMSymbols = 32; % Number of data OFDM symbols
bitsPerModSym = log2(txParam.modOrder); % Bits per modulated symbol
codeRate = 1/2; % Punctured code rate
dataConvK = 7; % Constraint length of convolutional code polynomial
dataCRCLen = 32; % Data CRC length
trBlkSize = ((numSubCar-pilotsPerSym)*numDataOFDMSymbols* ...
    bitsPerModSym*codeRate) - (dataConvK-1) - dataCRCLen;
txParam.txDataBits = randi([0 1],txParam.numFrames*trBlkSize,1);

% Generate complex baseband transmitter waveform
fprintf('\n-----\n');
fprintf('\n Transmitting %d frames ... \n',sum(txParam.numFrames));
[txWaveform,txGrid,txDiagnostics] = whdlexamples.OFDMTx(txParam);
fprintf('\n Transmission successful\n');
fprintf('\n-----\n');
```

```
-----
Transmitting 5 frames ...
Transmission successful
-----
```

whdlexamples.OFDMTx function returns txWaveform, txGrid, and txDiagnostics are described below.

- txWaveform is the generated time-domain waveform returned as a column vector of length  $((\text{fftLen} + \text{cpLen}) \times \text{txParam.numFrames} \times \text{numSymPerFrame}) + (\text{txFilterLen} - 1)$ , where
- fftLen is the FFT length.

- `cpLen` is the cyclic prefix length.
- `numSymPerFrame` is the number of OFDM symbols per frame.
- `txFilterLen` is the transmitter filter length.

If `txParam` is an array of structures, then `txParam.numFrames` is replaced with the sum of all `numFrames` attributes present in the array. The frame structure of the generated time-domain waveform `txWaveform` is similar to the Simulink HDL OFDM Transmitter output waveform. For the detailed explanation of the frame structure, see “HDL OFDM Transmitter” on page 5-121.

2. `txGrid` is the transmitter grid output and is returned as a matrix of dimension `numSubCar`-by-`(txParam.numFrames x numSymPerFrame)`.

3. `txDiagnostics` is a structure or array of structures and consists of these three fields.

- `headerBits` — This field represents the header bits as a column vector of size 22, which includes 3 bits for the FFT length index, 2 bits for the symbol modulation type, 2 bits for the code rate index, and 15 spare bits.
- `dataBits` — This field represents actual data bits transmitted in the given number of frames (`txParam.numFrames`). This field is a binary row or column vector of length `(txParam.numFrames x trbBlkSize)`. The row or column vector depends on the dimension of `txparam.dataBits`. The default size is a column vector of length `txParam.numFrames x trbBlkSize`.
- `ofdmModOut` — This field represents the OFDM modulator output as a column vector of length `(fftLen + cpLen) x txParam.numFrames x numSymPerFrame`.

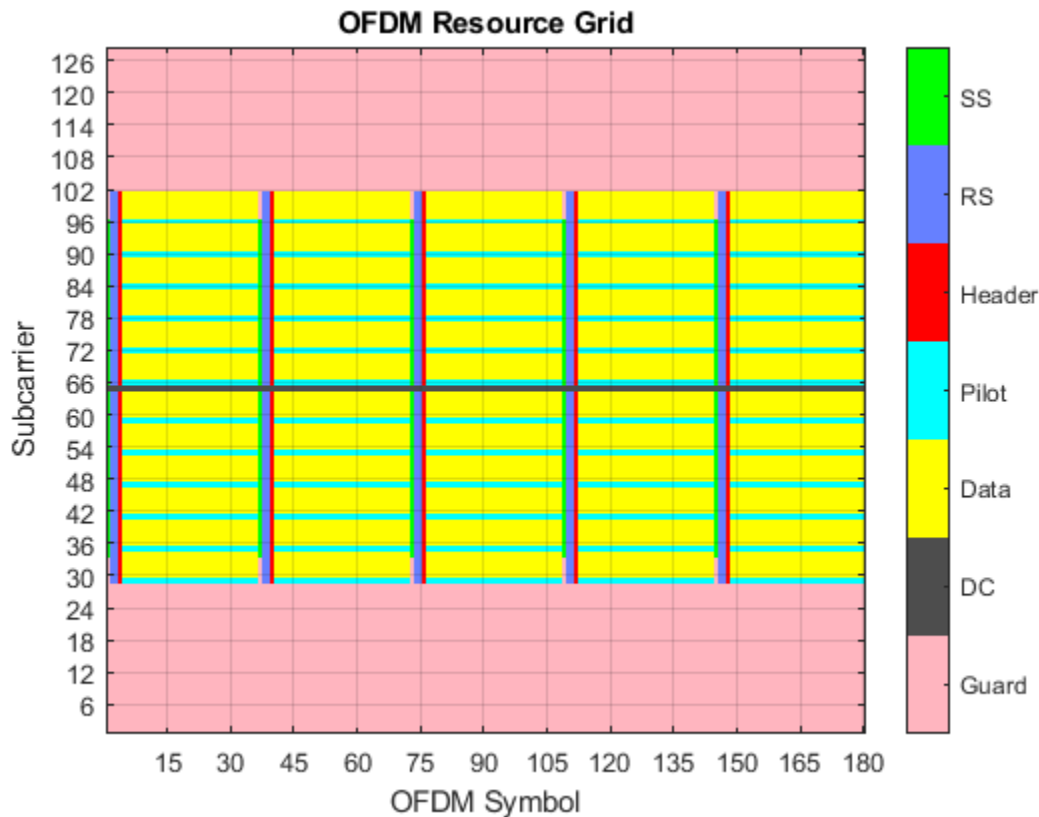
## OFDMTx

`whdlexamples.ofdmTx` function is used to generate OFDM transmitter waveform with synchronization, reference, header, pilots, and data signals. This function returns `txWaveform`, `txGrid`, and `diagnostics` using transmitter parameters `txParam`. This function internally calls these individual functions.

- `generateOFDMSyncSignal` — This function generates the synchronization signal `SyncSignal`. This function uses Zadoff-Chu sequence with a root index of 25 and length of 62.
- `generateOFDMRefSignal` — This function generates the reference signal `refSignal` for the given FFT length `fftLen`. This function uses a BPSK-modulated pseudo random binary sequence.
- `generateOFDMPilotSignal` — This function generates the pilot signal `pilot`. This function uses a BPSK-modulated pseudo random binary sequence.
- `OFDMSymbolModulate` — This function modulates input bits to complex modulation symbols based on the specified modulation scheme BPSK, QPSK, 16QAM, and 64QAM.

Plot the resource grid of the transmitter waveform. The plot indicates the magnitude variations of each resource grid elements.

```
plotResourceGrid(txGrid);
```



### HDL AWGN MATLAB Reference

This section describes the MATLAB reference of HDL AWGN.

This MATLAB reference is used for performance evaluation of the HDL OFDM Transmitter and Receiver algorithms. The HDL AWGN MATLAB reference generates AWGN by accepting the signal-to-noise ratio (SNR) in decibel (dB) and sets of seeds. For more details, see “HDL Implementation of AWGN Generator” on page 4-44. The generated AWGN is added to the HDL OFDM Transmitter output.

```
FFTLen = 128;
CPLen = 32;
usedSubCarr = 72; % Out of 128 subcarriers, 72 subcarriers are loaded with data
```

```
SNRdB = 30;
SNRdBsimInput = SNRdB*ones(length(txWaveform)+633,1);
seedsURNG1 = [121 719 511]; % Seeds for TausURNG1
seedsURNG2 = [2343 323 833]; % Seeds for TausURNG2
txScaleFactor = FFTLen/sqrt(usedSubCarr);
```

```
awgnNoise = whdlexamples.hdlawgn(SNRdBsimInput,seedsURNG1,seedsURNG2);
```

```
rxWaveform = txWaveform + (1/txScaleFactor)*awgnNoise(634:end);
fprintf('\n Applying the AWGN channel at %d dB\n', SNRdB);
```

Applying the AWGN channel at 30 dB



## HDL OFDM Receiver MATLAB Reference

This section describes MATLAB reference of HDL OFDM Receiver.

This MATLAB reference includes time synchronization, CFO estimation and correction, OFDM demodulation, header recovery, CPE estimation and correction, and data recovery.

The `whdlexamples.OFDMRx` function accepts `rxWaveform`, a transmitted waveform passed through an AWGN channel.

The `whdlexamples.OFDMRx` function returns decoded bits `rxBits` and an array of structures, `rxDiagnostics`, consisting of these eight fields.

- `estCFO` — Estimated carrier frequency offset
- `rxConstellationHeader` — Demodulated header constellation symbols
- `rxConstellationData` — Demodulated data constellation symbols
- `softLLR` — Demodulated soft LLR bits
- `decodedCodeRateIndex` — Decoded code rate index from header
- `decodedModOrder` — Decoded modulation order from header
- `headerCRCErrorFlag` — Status of header CRC
- `dataCRCErrorFlag` — Status of data CRC

### OFDMRx

The `whdlexamples.OFDMRx` function is used to demodulate and decode the received `rxWaveform`. This function internally calls these individual functions.

- `OFDMFrequencyOffset` — This function estimates the carrier frequency offset based on cyclic prefix (CP) technique. The cyclic prefix portion of the received time-domain waveform is correlated to estimate frequency offset.
- `OFDMFrequencyCorrect` — This function corrects the carrier frequency offset on the received waveform using the estimated frequency offset.
- `OFDMFrameSync` — This function synchronizes the received waveform by performing correlation using the reference signal. This step reduces the intersymbol interference while demodulating the received waveform.
- `OFDMDemodulation` — This function converts the time-domain waveform to frequency-domain waveform for further decoding. The object `dsp.HDLFFT` is used for HDL implementation of the receiver.
- `OFDMChannelEstimation` — This function performs the estimation of the channel using two reference signals. It uses least squares (LS) estimation technique. LS estimates are averaged to improve channel estimation accuracy.
- `OFDMChannelEqualization` — This function performs zero forcing (ZF) equalization using estimated channel. Then the received waveform that is free of the channel is used for header recovery and data recovery.
- `OFDMHeaderRecovery` — This function recovers header information by performing symbol demodulation, descrambling, and decoding. The success or failure of header information recovery is indicated by the CRC status. This header recovery CRC status is given as an output to the

receiver to indicate frame loss or recovery. When the CRC check fails, the header CRC status is 1. Otherwise, it is 0.

- `OFDMDataRecovery` — This function performs symbol demodulation, Viterbi decoding, depuncturing, and descrambling. The data is processed only when the header CRC check passes. After descrambling, CRC check on the recovered data bits to indicate whether the packet is errored. When the CRC check fails, the header CRC status is 1. Otherwise, it is 0.

```
fprintf('\n Receiving process started...\n');  
[rxDataBits,rxDiagnostics] = whdlexamples.OFDMRx(rxWaveform);  
fprintf('\n Reception completed\n\n');
```

```
% Plot constellation of header and data  
scatterplot(rxDiagnostics.rxConstellationHeader(:))  
title('Header Constellation')
```

```
scatterplot(rxDiagnostics.rxConstellationData(:))  
title('Data Constellation');
```

```
Receiving process started...
```

```
Estimating carrier frequency offset ...
```

```
First four frames are used for carrier frequency offset estimation.
```

```
Estimated carrier frequency offset is -2.218976e-04 KHz.
```

```
Detected and processing frame 5  
-----
```

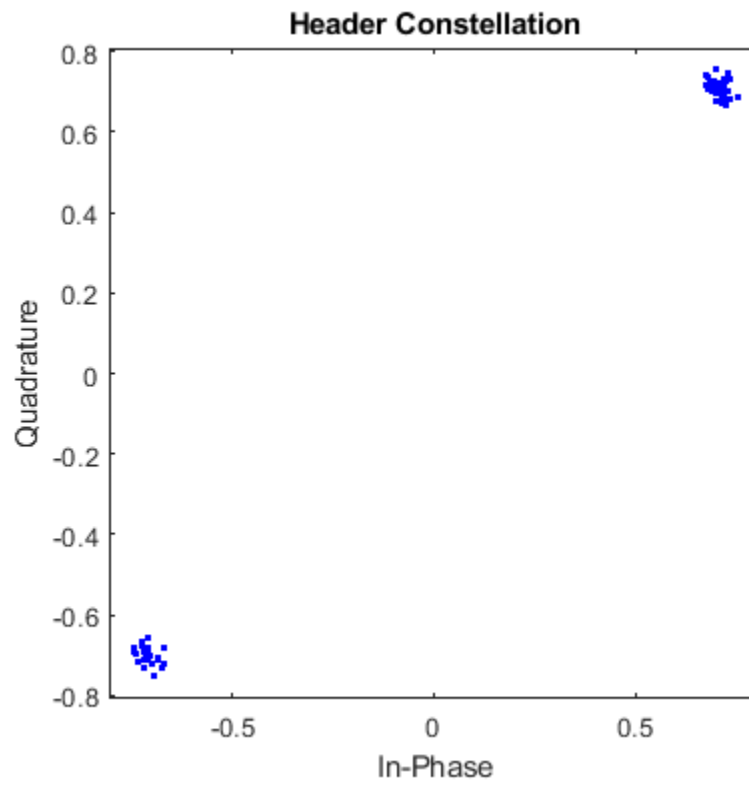
```
Header CRC passed
```

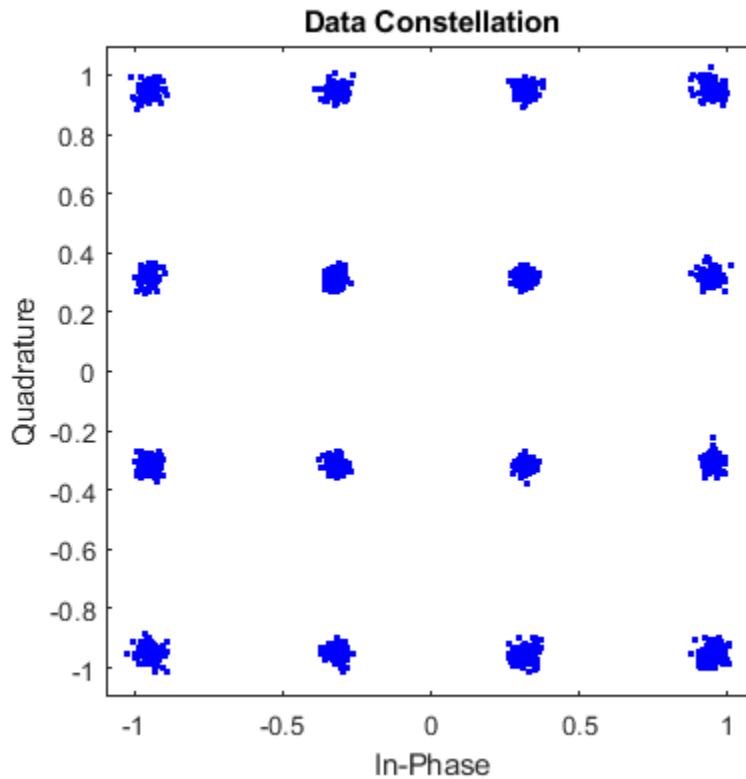
```
Modulation: 16QAM, codeRate=1/2 and FFT Length=128
```

```
Data CRC passed
```

```
Data decoding completed  
-----
```

```
Reception completed
```





### Verify Simulink Model with MATLAB Reference

In this section, Simulink HDL OFDM Transmitter, AWGN generator, and Simulink HDL OFDM Receiver algorithms implemented in fixed point are compared with the equivalent MATLAB HDL reference models implemented in floating point.

The Simulink model consists of an OFDM Transmitter that generates time-domain waveform for a user-defined modulation order and code rate. The waveform is then passed through the AWGN channel that introduces AWGN noise of the desired SNR in dB. Then, the OFDM Receiver algorithm is used to demodulate and decode information bits. The outputs of the Simulink model are verified with the MATLAB reference at each stage.

```
open HDLOFDMTxRx;
sim HDLOFDMTxRx;
```

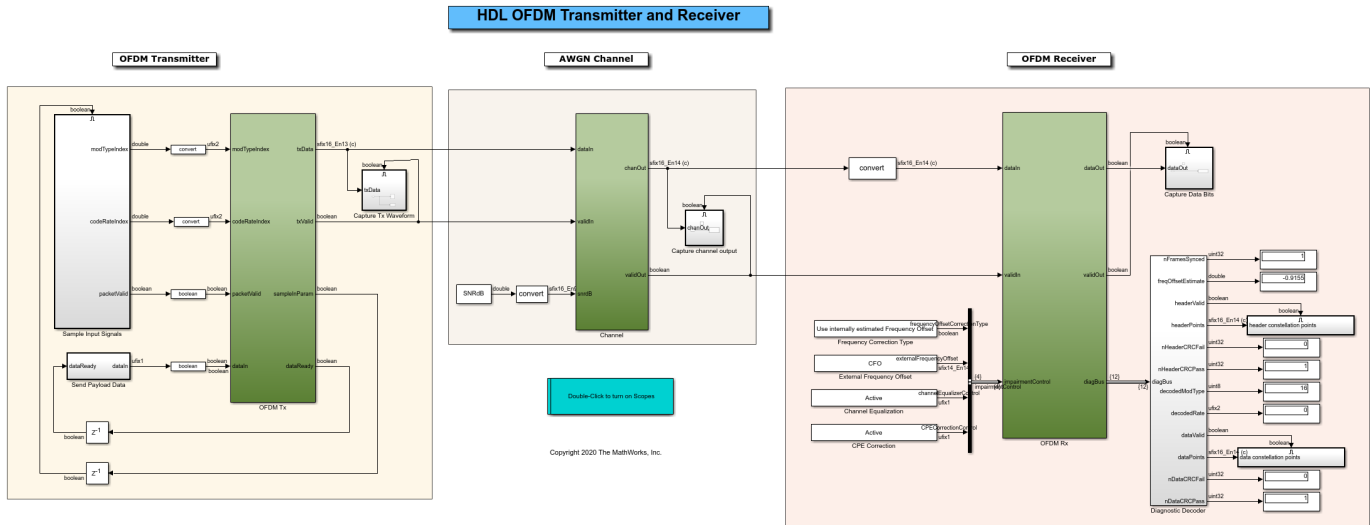
```
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: whdloFDMRx
### Successfully updated the model reference simulation target for: whdloFDMTx
```

Build Summary

Simulation targets built:

Model	Action	Rebuild Reason
whdloFDMRx	Code generated and compiled	whdloFDMRx_msf.mexw64 does not exist.
whdloFDMTx	Code generated and compiled	whdloFDMTx_msf.mexw64 does not exist.

2 of 2 models built (0 models already up to date)  
Build duration: 0h 16m 45.656s



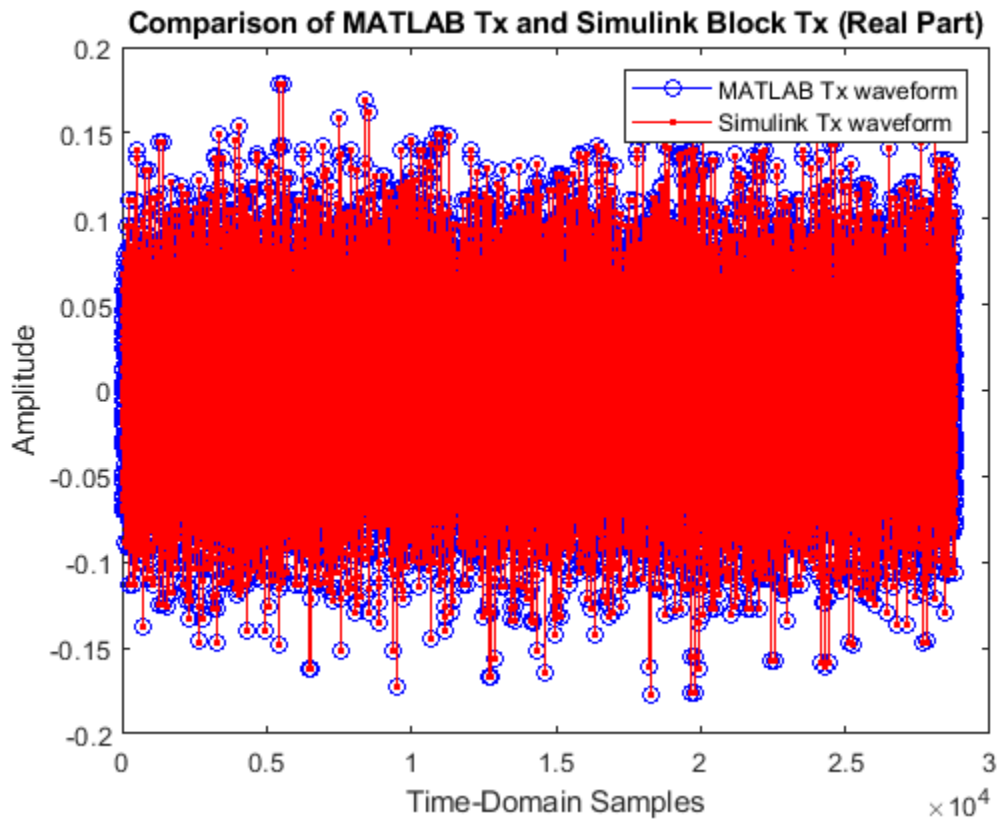
## Verify Simulink HDL OFDM Transmitter with MATLAB HDL OFDM Transmitter

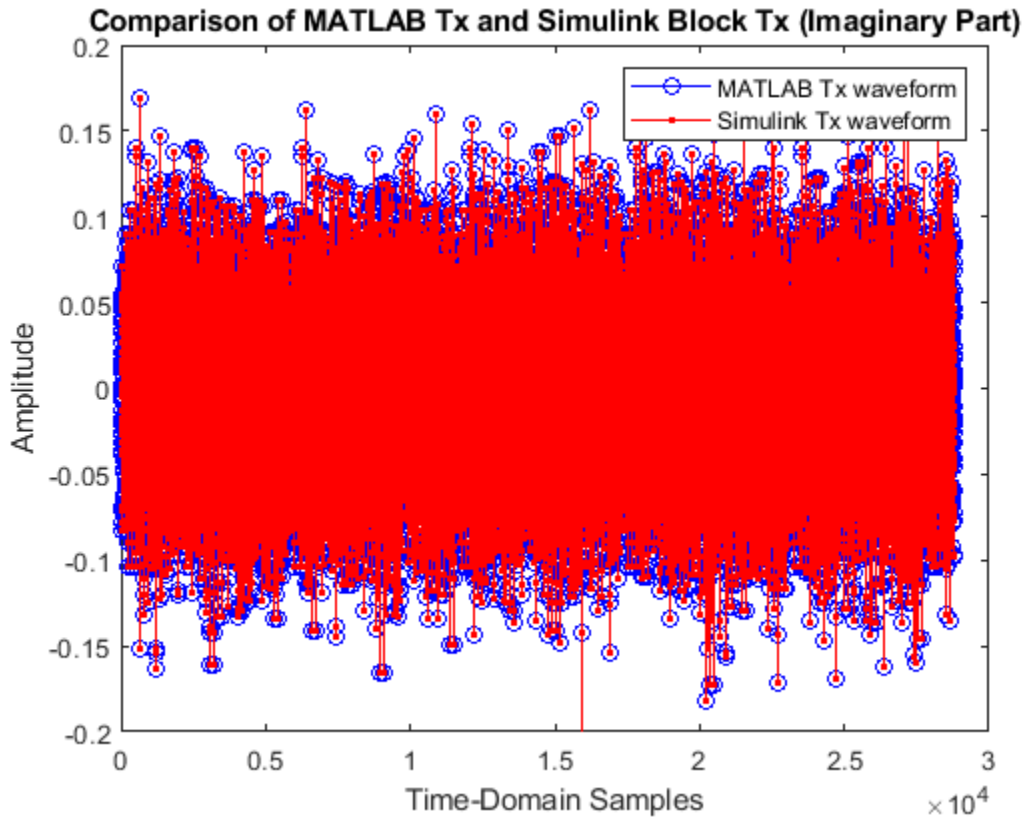
In this section, plot the real and imaginary parts of the HDL OFDM Transmitter MATLAB reference function output `txWaveform` as compared with the output of the “HDL OFDM Transmitter” on page 5-121 block.

```
matlabTxWaveform = txWaveform;
simulinkTxWaveform = simTxOut;
```

```
figure;
plot(real(matlabTxWaveform), '-bo')
hold on
plot(real(simulinkTxWaveform(1:length(matlabTxWaveform))), '-r.')
legend('MATLAB Tx waveform', 'Simulink Tx waveform');
title('Comparison of MATLAB Tx and Simulink Block Tx (Real Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');
```

```
figure;
plot(imag(matlabTxWaveform), '-bo')
hold on
plot(imag(simulinkTxWaveform(1:length(matlabTxWaveform))), '-r.')
legend('MATLAB Tx waveform', 'Simulink Tx waveform');
title('Comparison of MATLAB Tx and Simulink Block Tx (Imaginary Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');
```





### Verify Simulink HDL AWGN Generator with MATLAB HDL AWGN

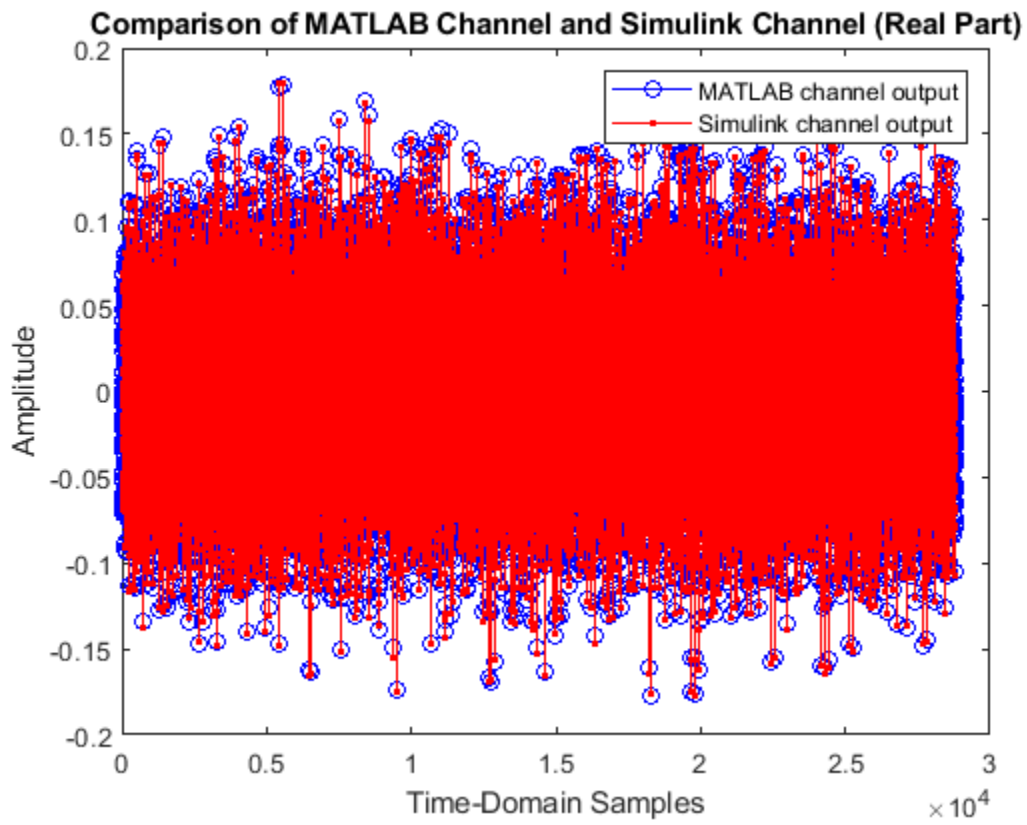
In this section, plot the real and imaginary parts of the MATLAB HDL AWGN is compared with the output of the Simulink AWGN Generator block.

```
matlabChannelOut= rxWaveform;
simulinkChannelOut = simChannelOut;
```

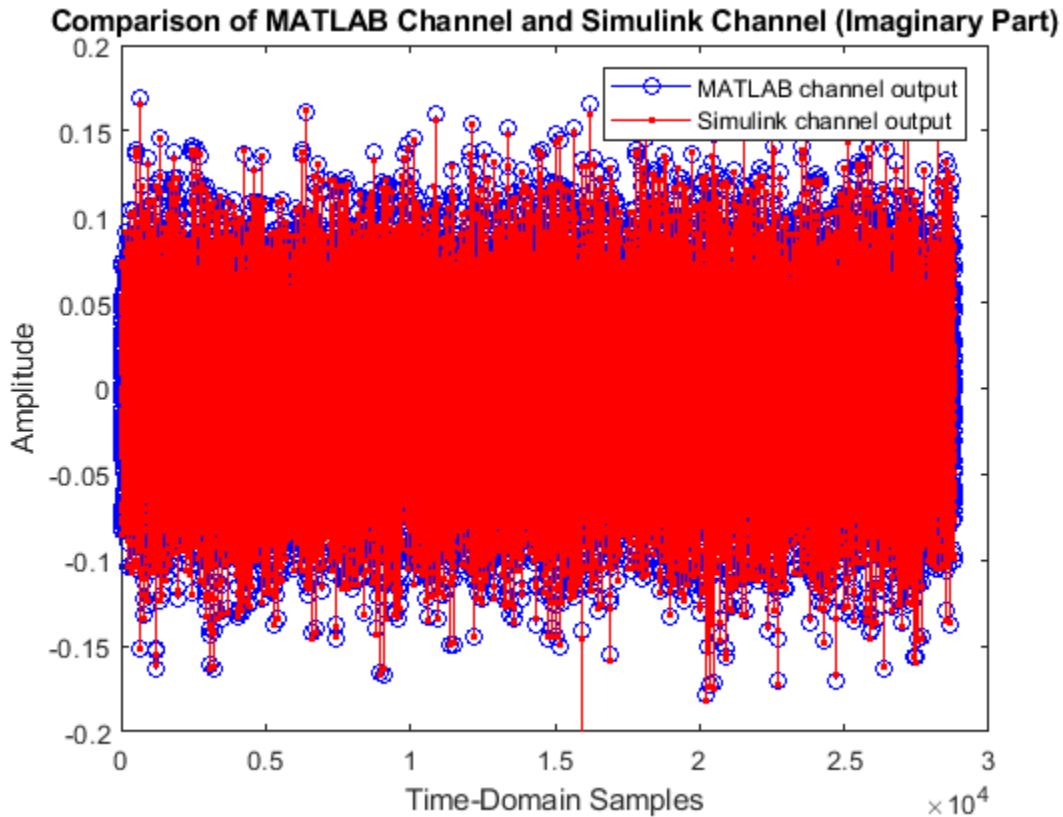
```
figure;
plot(real(matlabChannelOut),'-bo');
hold on;
plot(real(simulinkChannelOut(1:length(matlabChannelOut))),'-r. ');
legend('MATLAB channel output','Simulink channel output');
title('Comparison of MATLAB Channel and Simulink Channel (Real Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');
```

```
figure;
plot(imag(matlabChannelOut),'-bo');
hold on;
plot(imag(simulinkChannelOut(1:length(matlabChannelOut))),'-r. ');
legend('MATLAB channel output','Simulink channel output');
title('Comparison of MATLAB Channel and Simulink Channel (Imaginary Part)');
ylim([-0.2 0.2]);
```

```
xlabel('Time-Domain Samples');  
ylabel('Amplitude');
```





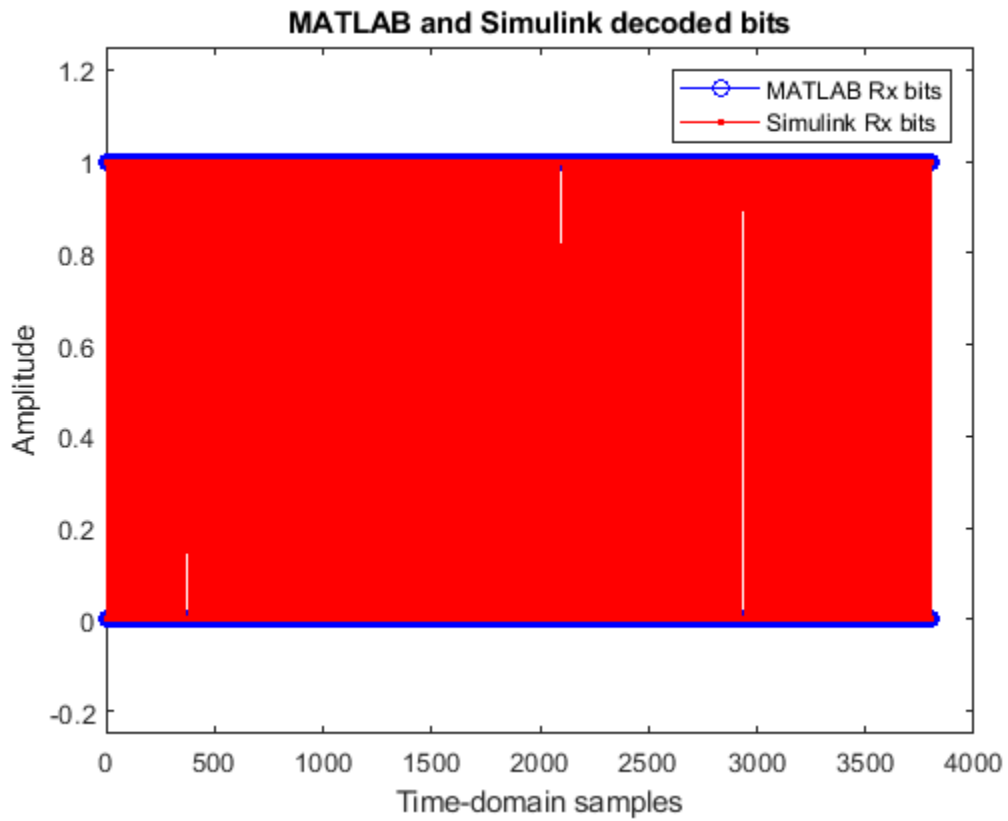


### Verify Simulink HDL OFDM Receiver with MATLAB HDL OFDM Receiver

In this section, plot the decoded bits of the MATLAB receiver as compared with the decoded bits of the Simulink receiver.

```
matlabRxOut= rxDataBits;
simulinkRxOut = simRxDataBits;

figure;
plot(rxDataBits,'-bo');
hold on;
plot(simulinkRxOut(1:length(rxDataBits)),'-r. ');
legend('MATLAB Rx bits','Simulink Rx bits');
title('MATLAB and Simulink decoded bits');
ylim([-0.25 1.25]);
xlabel('Time-domain samples');
ylabel('Amplitude');
```



## See Also

### Related Examples

- "HDL OFDM Receiver" on page 5-136
- "HDL OFDM Transmitter" on page 5-121
- "HDL Implementation of AWGN Generator" on page 4-44

## HDL OFDM Transmitter

This example shows an OFDM-based wireless transmitter implemented in Simulink® that is optimized for HDL code generation and hardware implementation.

This example shows the custom design of an orthogonal frequency-division multiplexing (OFDM) based transmitter. This transmitter model accepts payload data through the input port. It allows you to choose the modulation type and the punctured convolutional code rate of the data from a set of values. These two parameters control the effective data rate of transmission and they can be provided through the input ports of transmitter. The maximum data rate supported by the transmitter is 3 Mbps. The transmitter also accepts an input packetValid signal to control the transmission.

The transmitter in this example works in conjunction with the receiver in the “HDL OFDM Receiver” on page 5-136 example. The transmitter has a MATLAB® floating point equivalent function described in the “HDL OFDM MATLAB References” on page 5-107 example.

### Transmitter Specification

This section explains the specifications of the transmitter related to the OFDM frame configuration and structure, bandwidth, and sample rate.

The transmitter model accepts two parameters, modTypeIndex and codeRateIndex, which allow you to specify the modulation type and punctured convolutional code rate, respectively, of the data. These two parameters are explained in the following tables:

#### modTypeIndex

Value	Represents modulation type
0	BPSK
1	QPSK
2	16QAM
3	64QAM

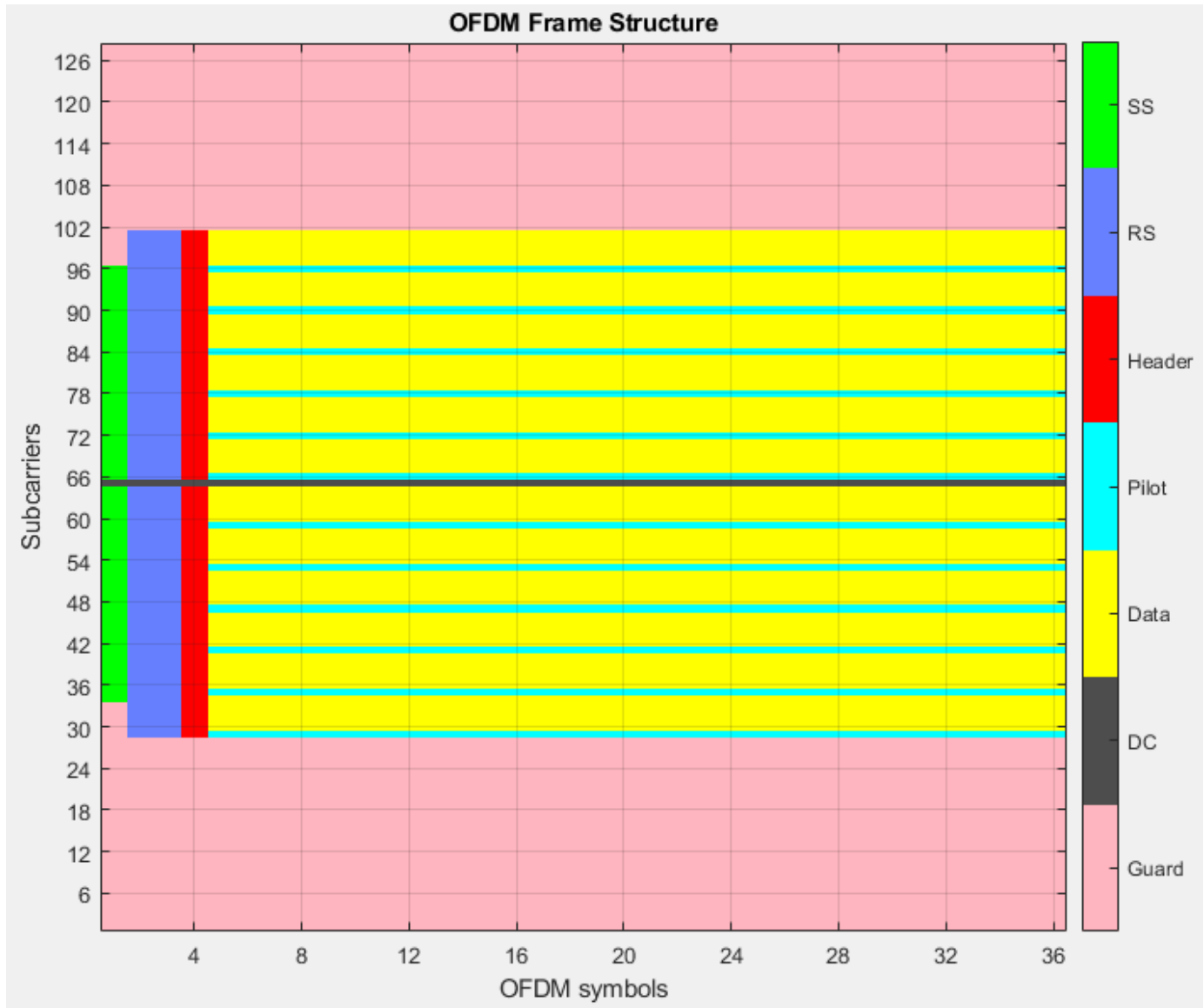
#### codeRateIndex

Value	Represents code rate
0	1/2
1	2/3
2	3/4
3	5/6

### OFDM Frame Structure

Every OFDM system has a frame structure that shows the distribution of samples in the frequency domain across all its subcarriers. The frame structure is as shown in the figure. Each OFDM symbol is comprised of 72 subcarriers, and each OFDM frame consists of 36 OFDM symbols. The frame duration is 3 ms. The first OFDM symbol is formed by synchronization sequence (SS), second and third symbols are formed by reference signals (RS), and the fourth symbol is formed by Header. Data is filled from the fifth symbol to the last (36th) symbol. Pilots are inserted between data such that

there is one pilot for every five data subcarriers as shown below. These pilots help detect and correct phase errors at the receiver.



The OFDM parameters used in the model are given below:

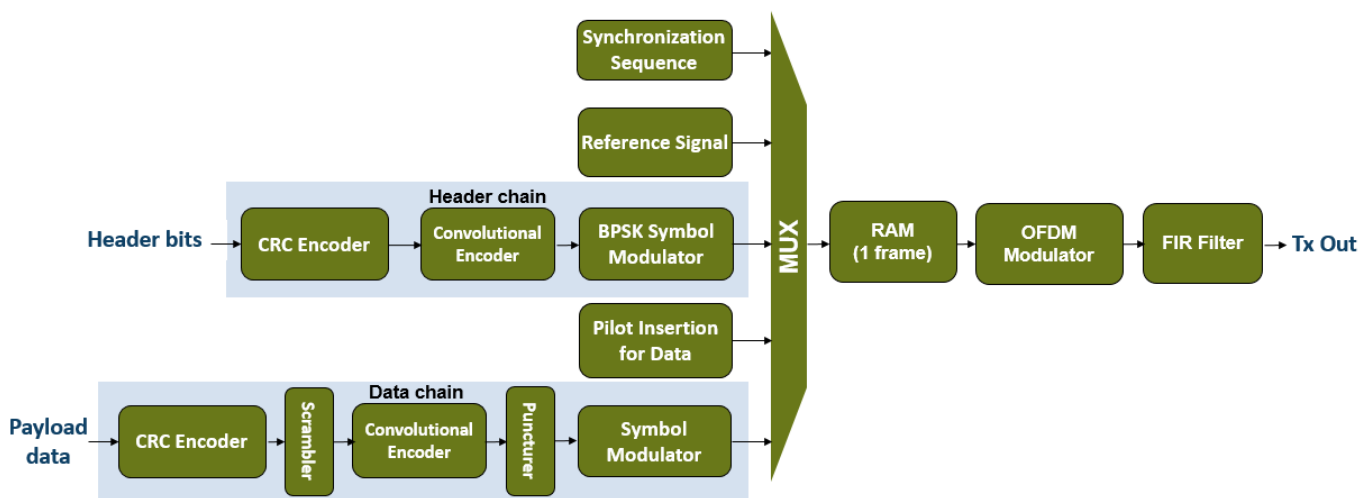
Parameter	Value
Sample rate	1.92 Msps
Subcarrier spacing	15 KHz
FFT Length	128
Bandwidth of OFDM signal	1.4 MHz
Active Subcarriers	72
Left guard subcarriers	28
Right guard subcarriers	27
Cyclic Prefix length	32

Data symbols per frame	32
Pilots per data symbol	12

## Model Architecture

The following figure shows the high-level architecture of the OFDM transmitter. There are five different signals that form the OFDM frame: SS, RS, Header, Pilots, and Data. SS, RS, and Pilots are the same for every frame. They are stored in separate look up tables (LUT) and accessed whenever required. Header and Data vary based on the inputs given to the transmitter. Header bits are formed based on the modulation type and code rate input values. These header bits are processed through the Header chain as shown in the figure. Payload data is provided as an input to the transmitter. This data is processed through multiple stages in the Data chain. Individual stages in the Header and Data chains are explained in further sections.

These five signals are multiplexed based on their valid signals and stored in a RAM. The RAM holds these signals for a duration of one frame. Data stored in the RAM is read out and modulated by the OFDM Modulator block. The OFDM modulated signal is filtered with a passband frequency of 1.4 MHz and sent out as transmitter output.



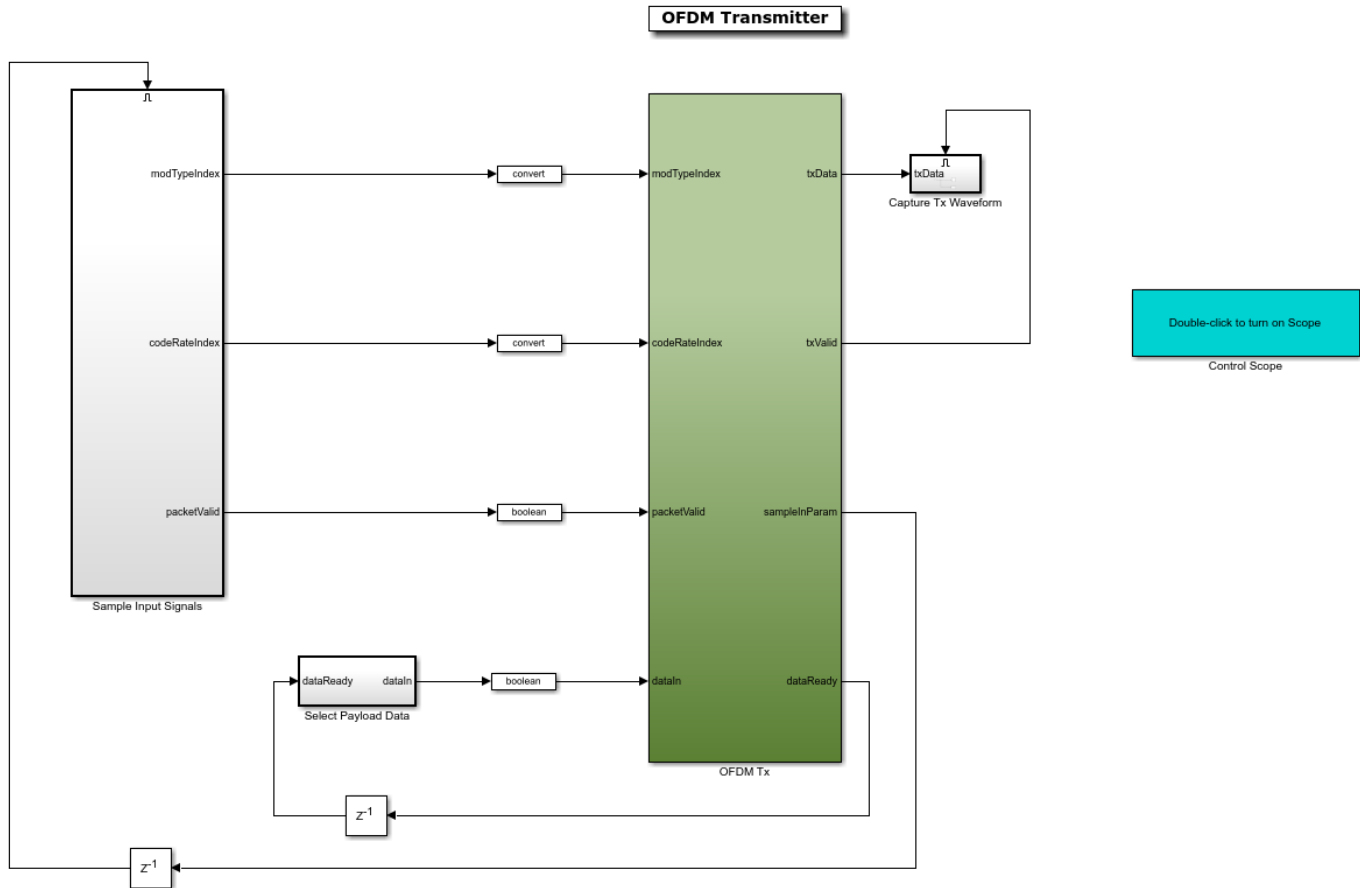
## File Structure

This example uses two Simulink models, an initialization script, and a MATLAB function:

- `whd\OFDMTransmitter.slx` — This is the top-level model in this example. It has an OFDM Tx subsystem that refers to the `whd\OFDMTx.slx` model. There is external interface circuit for the OFDM Tx subsystem, which provides inputs and collects outputs from the subsystem. Running this model runs the remaining three files.
- `whd\examples\OFDMTransmitterInit` — This script initializes the `whd\OFDMTransmitter.slx` model. The script is called in the `InitFcn` callback of the model.
- `whd\OFDMTx.slx` — This model implements the transmitter with total configurability. The `whd\OFDMTransmitter.slx` model refers to this model as explained above.
- `whd\examples\OFDMTxParameters` — A function that generates parameters required for the `whd\OFDMTx.slx` model. This function is called in the Model Workspace of the model.

## Transmitter Interface

The top level `whd1OFDMTransmitter.slx` model shows the OFDM Tx subsystem and its interface.



Copyright 2020 The MathWorks, Inc.

### Model Inputs:

- *modTypeIndex* — Selects the type of symbol modulation to be applied to payload data, specified as a `ufix2` scalar. This port accepts values 0, 1, 2, and 3 which correspond to modulation types BPSK, QPSK, 16QAM, and 64QAM.
- *codeRateIndex* — Selects the code rate of punctured convolutional code to be applied to payload data, specified as a `ufix2` scalar. This port accepts values 0, 1, 2, and 3 which correspond to code rates 1/2, 2/3, 3/4, and 5/6.
- *packetValid* — Controls transmission of frames, specified as a Boolean scalar. If *packetValid* is 1, the transmitter outputs a valid OFDM frame for given input payload data, *modTypeIndex* and *codeRateIndex* values. If *packetValid* is 0, the transmitter outputs a dummy frame without the given input configuration and data.
- *dataIn* — Input payload data, specified as a Boolean scalar.

All input ports run at a sample rate of 30.72 Msps to support different configurations.

### Model Outputs:

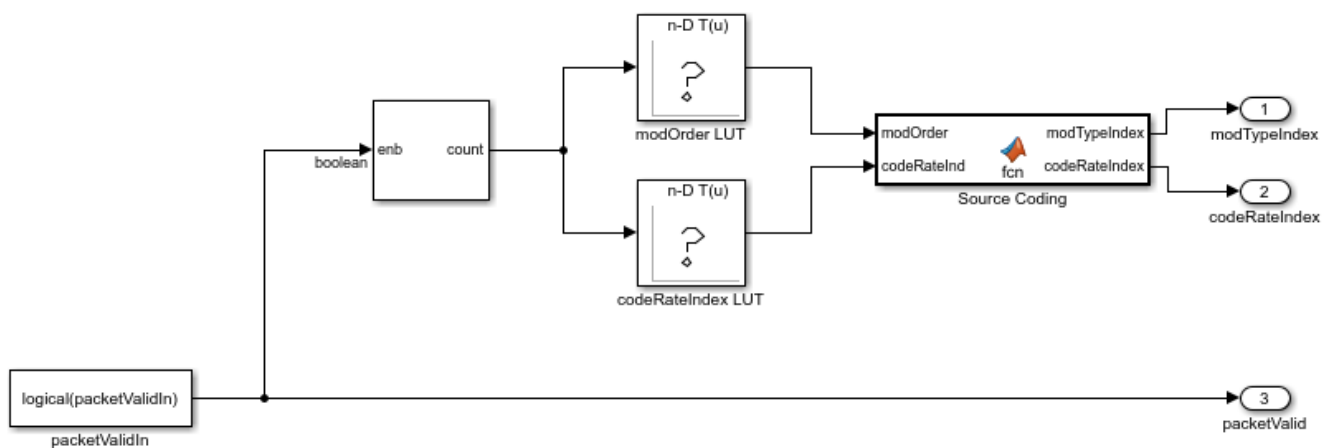
- *txData* — Transmitter output, returned as a complex scalar with `fixdt(1,16,13)` datatype sampled at 1.92 Msps.
- *txValid* — Control signal that validates *txData*, returned as a Boolean scalar sampled at 1.92 Msps.
- *sampleInParam* — Control signal used to sample *modTypeIndex*, *codeRateIndex* and *packetValid* inputs, specified as a Boolean scalar sampled at 30.72 Msps.
- *dataReady* — Control signal that samples input payload data, *dataIn*, specified as a Boolean scalar sampled at 30.72 Msps.

### Sample Input Signals

The Sample Input Signals subsystem samples *modTypeIndex*, *codeRateIndex* and *packetValid* signals based on the *sampleInParam* signal. This subsystem provides outputs only when there is an active *sampleInParam* signal and retains the previous outputs if the *sampleInParam* signal is inactive. The subsystem stores given input *modOrder* and *codeRateIndex* values in two LUTs as shown in the figure. If *packetValid* is 1, corresponding *modOrder* and *codeRateIndex* values are selected from LUTs. The Source Coding function maps *modOrder* values 2, 4, 16, and 64 to the corresponding *modTypeIndex* values 0, 1, 2, and 3 and sets 1 (QPSK) as the default value. It also takes *codeRateIndex* values 0, 1, 2, and 3 from the LUT and sets 0 (1/2 code rate) as the default value. If *packetValid* is 0, the LUTs are ignored.



This subsystem samples parameters at the start of every frame using *sampleInParam* signal



'packetValid' controls the transmission of a packet.

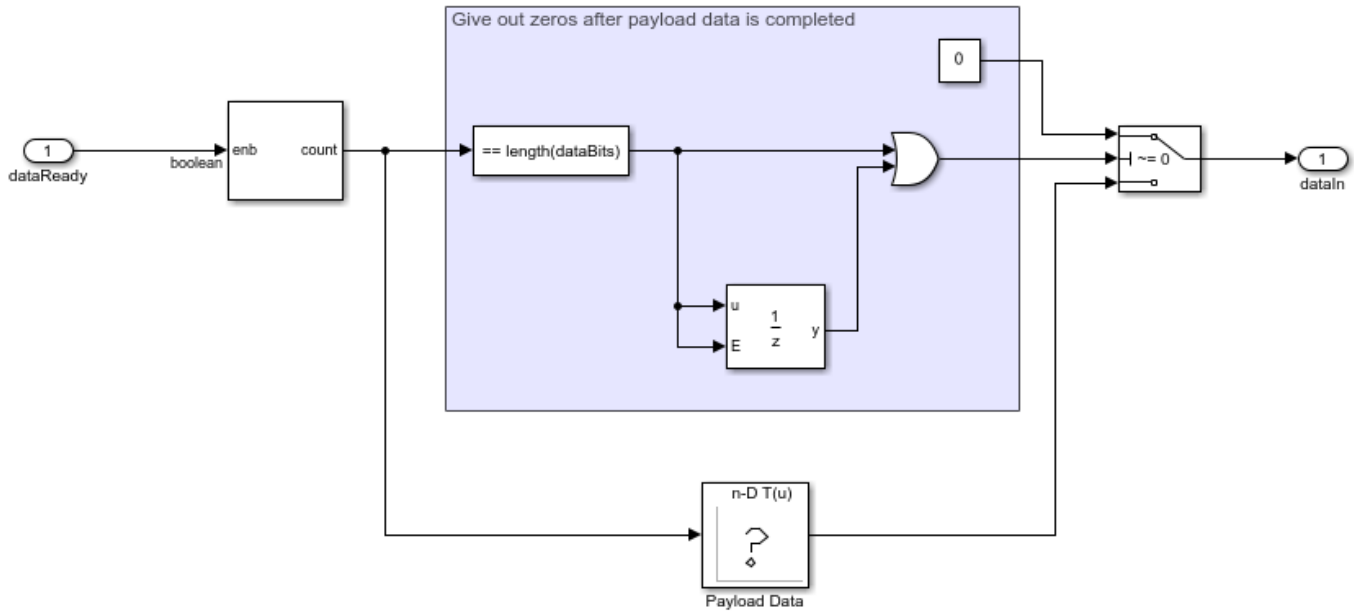
If 'packetValid' is 1, the packet is transmitted with correct synchronization sequence (SS) and the 'modOrder' and 'codeRateIndex' values are considered.

If 'packetValid' is 0, the packet is transmitted with a faulty synchronization sequence (SS) and the corresponding 'modOrder' and 'codeRateIndex' values are ignored (dummy packet).

### Select Payload Data

The Select Payload Data subsystem selects input payload data based on the *dataReady* signal. It has a counter that increments when the *dataReady* signal turns active. There is an LUT that stores the

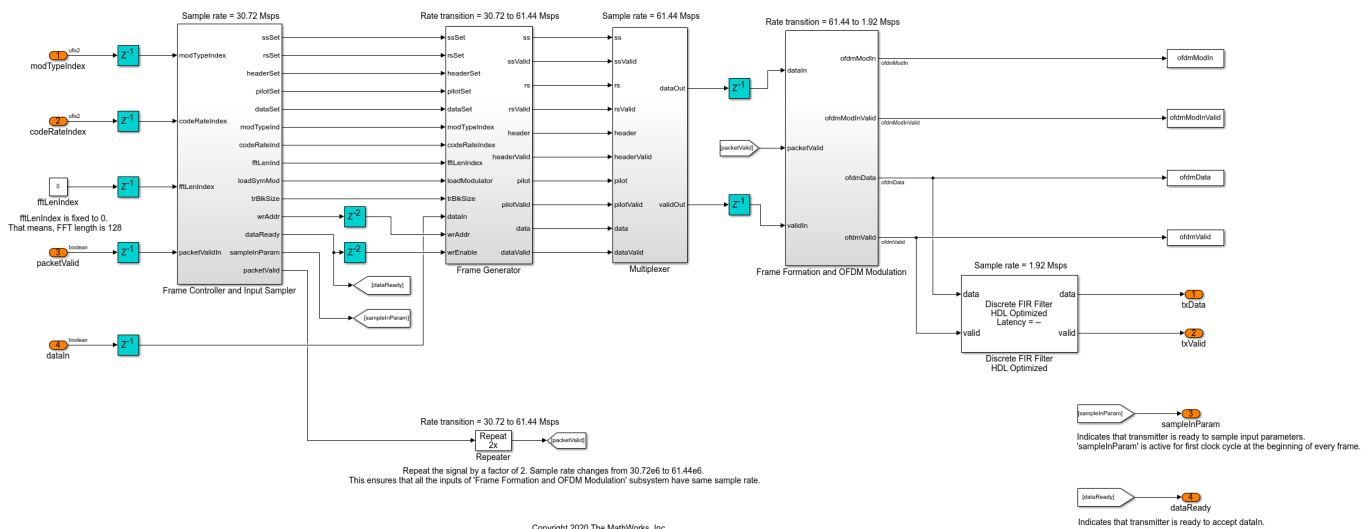
given input payload data. Data from this LUT is selected based on the counter value. There is an additional circuit that returns zeros as data after the counter reaches maximum count.



### Structure of the Transmitter

The `whd1OFDMTx.slx` model is called within the OFDM Tx subsystem. It generates an OFDM transmitter waveform by processing input signals in multiple stages as shown below.

### whd1OFDMTx

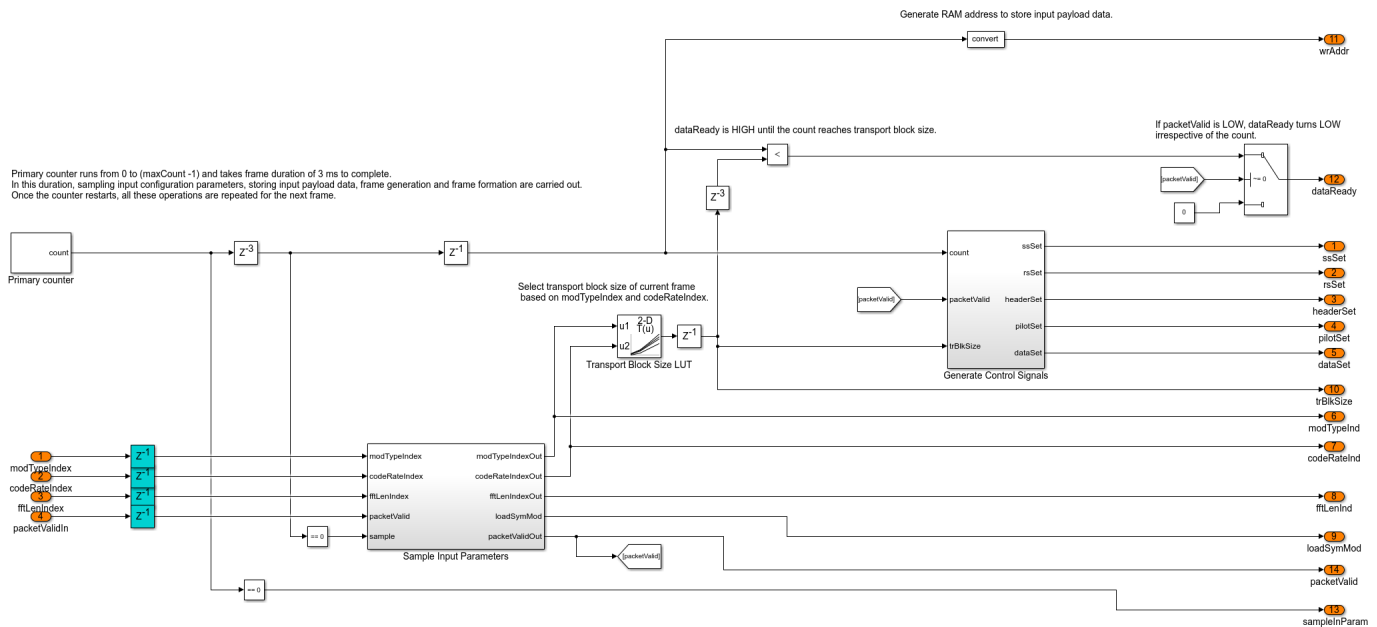


### Frame Controller and Input Sampler

The Frame Controller and Input Sampler subsystem has a counter that increments for a frame duration of 3 ms and then restarts. The counter restarts from 0 at the beginning of every frame and



generates the *sampleInParam* signal to sample input signals *modTypeIndex*, *codeRateIndex*, and *packetValid*. Based on the sampled *modTypeIndex* and *codeRateIndex* values, corresponding transport block size for the frame is selected from the Transport Block Size LUT. Then, the *dataReady* signal becomes active to accept the input data, *dataIn*, of size equal to transport block length. If sampled *packetValid* signal is false, *dataReady* remains inactive indicating generation of a dummy frame that does not require any input data. This subsystem also generates control signals for SS, RS, Header, Pilot, and Data generation that is carried out in further stages.

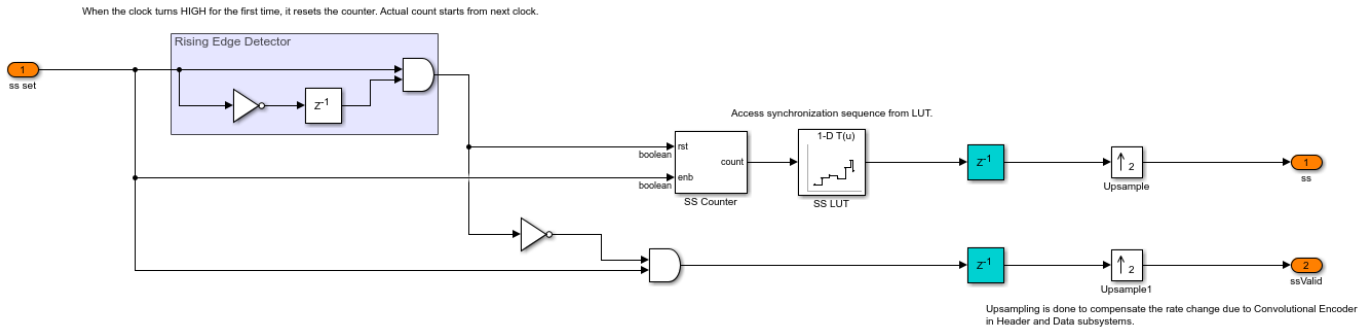


## Frame Generator

The Frame Generator subsystem consists of five subsystems that generate SS, RS, Header, Pilot, and Data signals, which are later OFDM-modulated.

### Frame Generator/SS

The SS (Synchronization Sequence) subsystem accepts *ssSet* control signal, generated from the Frame Controller and Input Sampler subsystem. It is generated considering the length of SS sequence. This signal initially resets the counter. Then, the counter keeps incrementing and simultaneously returns SS from an LUT. Once *ssSet* becomes inactive, the counter stops. Output from LUT is upsampled by a factor of 2 to maintain the same sample time as that of the Header and Data subsystems. RS and Pilot subsystems operate in a similar way by storing the sequences in LUTs and accessing them whenever required.

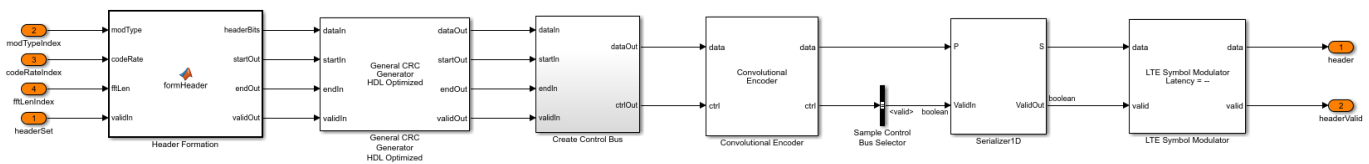


### Frame Generator/Header

The Header subsystem accepts *modTypeIndex*, *codeRateIndex* and *fftLenIndex* as inputs. A *headerSet* signal starts the header formation. The Header Formation function converts the *modTypeIndex* and *codeRateIndex* values into their binary equivalents. For example, a *modTypeIndex* value of 1 is converted into two bits 01. Similarly, *codeRateIndex* values are converted into two equivalent bits. To learn more about these indices, refer to **Transmitter Specification**. *fftLenIndex* is not configurable and its value is fixed to 0. It is converted to 000, which represents an FFT length of 128. *fftLenIndex*, *modTypeIndex*, and *codeRateIndex* are represented using 3, 2, and 2 bits, forming a total of 7 bits. Additionally, 15 spare bits are added, all currently set to 0, forming a total of 22 Header bits.

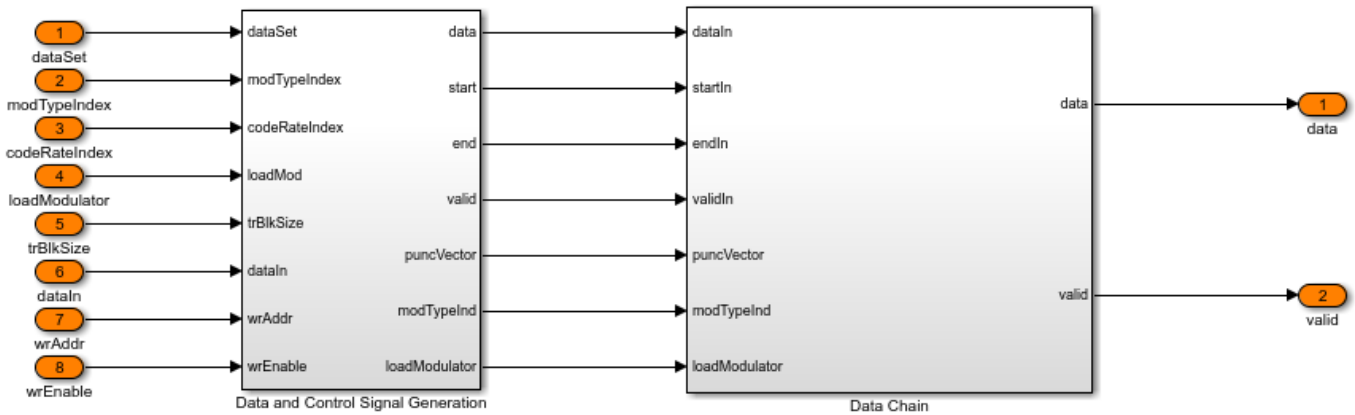
These 22 bits are processed as shown below. For proper error detection, 8 CRC bits are padded using the General CRC Generator HDL Optimized block with [8 7 4 3 1 0] as the CRC polynomial. These 30 bits, that is (22 + 8), are encoded using the Convolutional Encoder block with [171 133] as the polynomial and a constraint length of 7. The encoding is done in terminated mode which adds 6 null bits, that is (7 - 1), to the CRC padded data. After encoding, these 36 bits result in 72 bits due to the 1/2 rate encoding. The output of the Convolutional Encoder block is a two-element vector that is serialized using the Serializer1D (HDL Coder) block leading to rate transition by a factor 2. These 72 bits are BPSK-modulated using the LTE Symbol Modulator block to form Header symbol.

Header bits are formed based on input modTypeIndex, codeRateIndex, and fftLenIndex values. Header bits are then processed through Header chain as shown below.



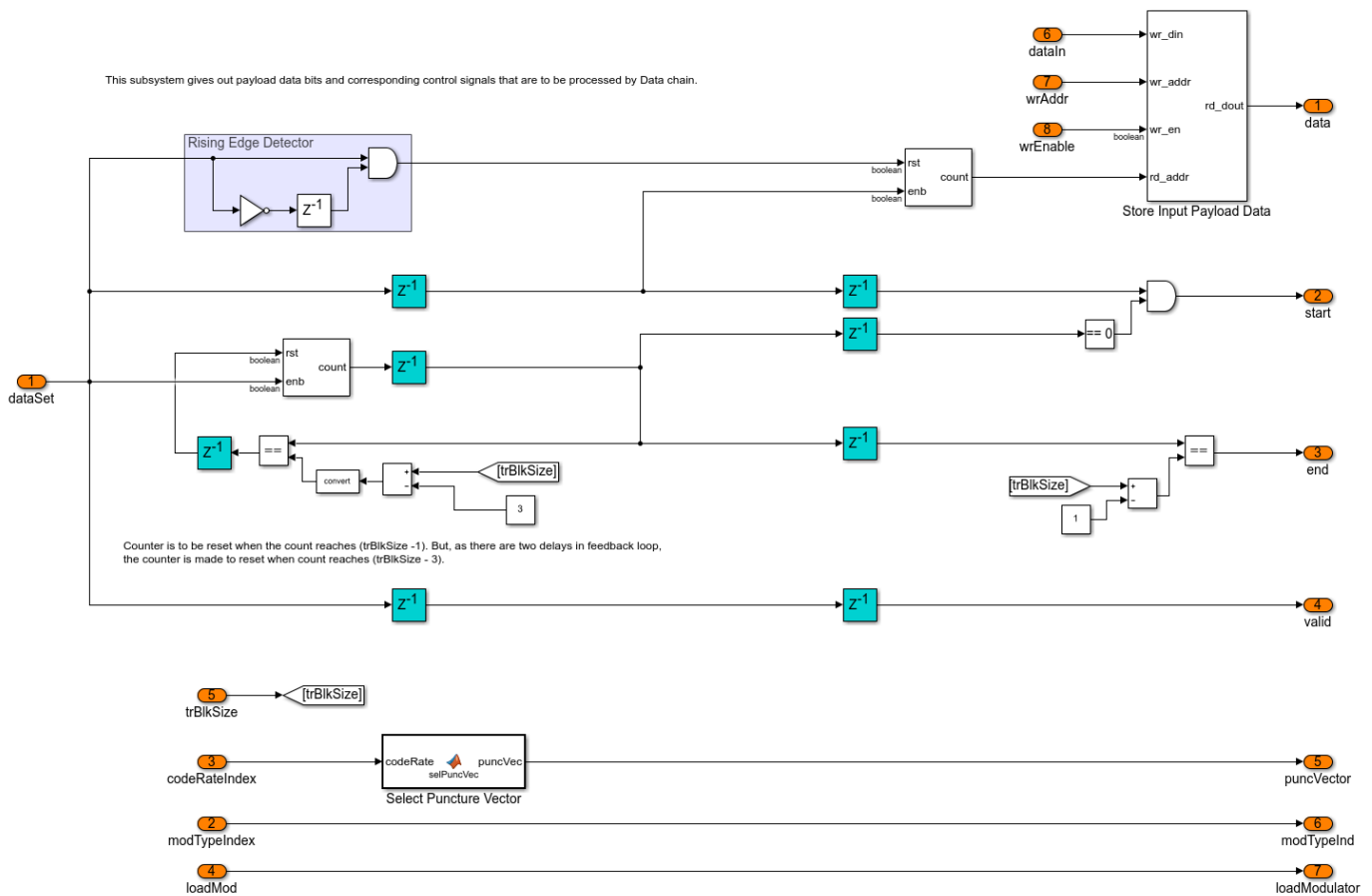
### Frame Generator/Data

The Data subsystem stores input payload data, *dataIn*, and processes it through the Data chain.



### Frame Generator/Data/Data and Control Signal Generation

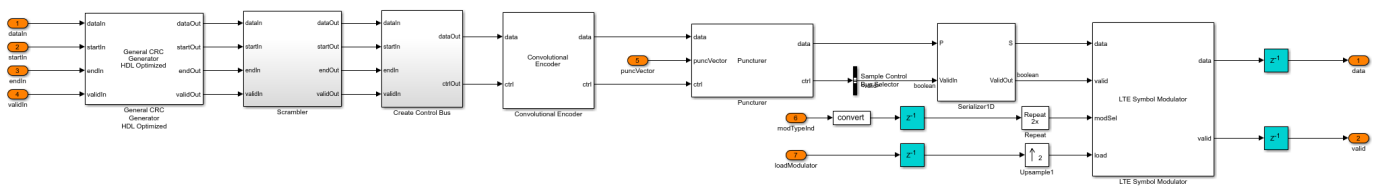
The Data and Control Signal Generation subsystem consists of a RAM where input payload data, *dataIn*, is stored. A *dataSet* signal reads data from this RAM. This subsystem generates *start*, *end*, and *valid* control signals for the RAM data. It even selects the puncture vector based on the *codeRateIndex*.



### Frame Generator/Data/Data Chain

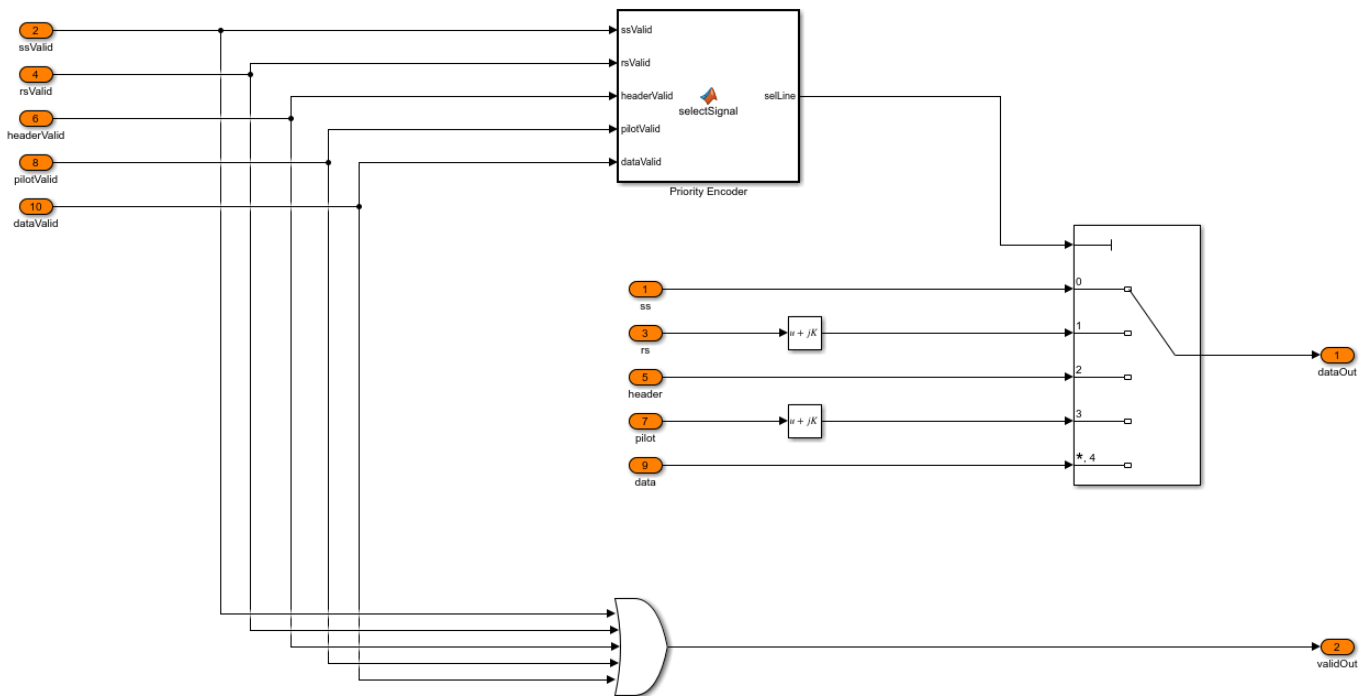
Payload data from RAM and the control signals are given as inputs to the General CRC Generator HDL Optimized block, which appends a 32-bit CRC to the data with [32 26 23 22 16 12 11 10 8 7 5 4 2 1 0] as the CRC polynomial. This CRC-padded data is scrambled with  $x^7 + x^4 + 1$  as the polynomial and [1 0 1 1 1 0 1] as the initial state. The scrambled data is encoded using the Convolutional Encoder block in terminated mode using [171 133] as the polynomial and a constraint length of 7. This output is punctured using the Puncturer block with the puncture vector selected in the Data and Control Signal Generation subsystem. The output of the Puncturer block is a two-element vector and is serialized using Serializer1D (HDL Coder) block. The resultant data is modulated using the LTE Symbol Modulator block using the modulation pattern selected based on *modTypeIndex*.

Input payload data is processed through Data chain as shown below. Puncturer and Symbol Modulator blocks control the effective data rate.



### Multiplexer

The Multiplexer subsystem multiplexes the SS, RS, Header, Pilot, and Data signals based on the corresponding valid signals that are generated by the Frame Generator subsystem.



### Frame Formation and OFDM Modulation

The Frame Formation and OFDM Modulation subsystem takes in the multiplexed *dataOut* and *validOut* signals and writes them into a Dual Rate Dual Port RAM (HDL Coder). This RAM reads and writes data at different rates. Data is written into the RAM at 61.44 Msps. An LUT contains addresses

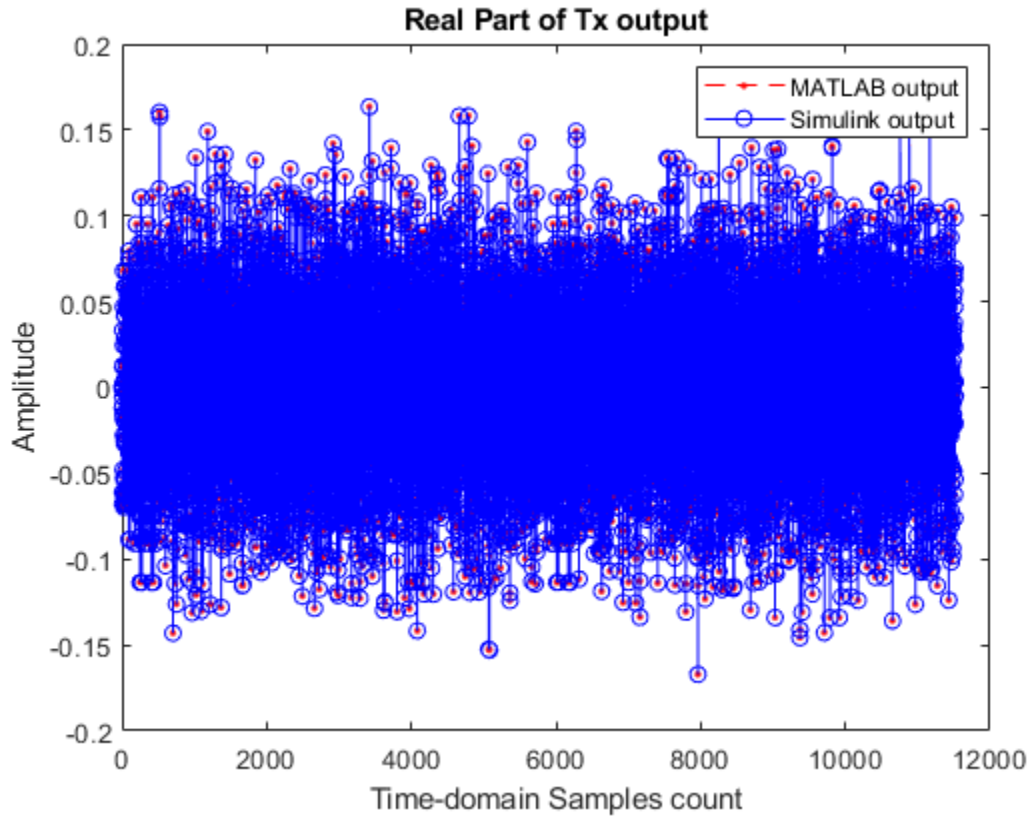


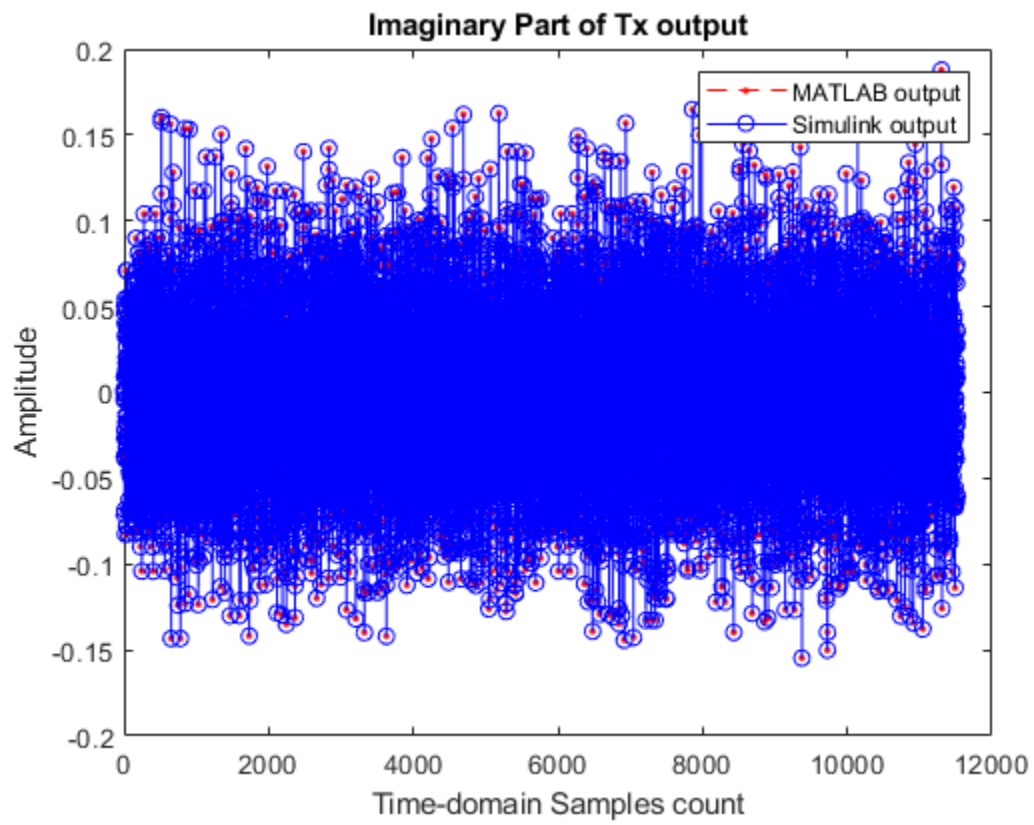
```
>> OFDMTxVerification
```

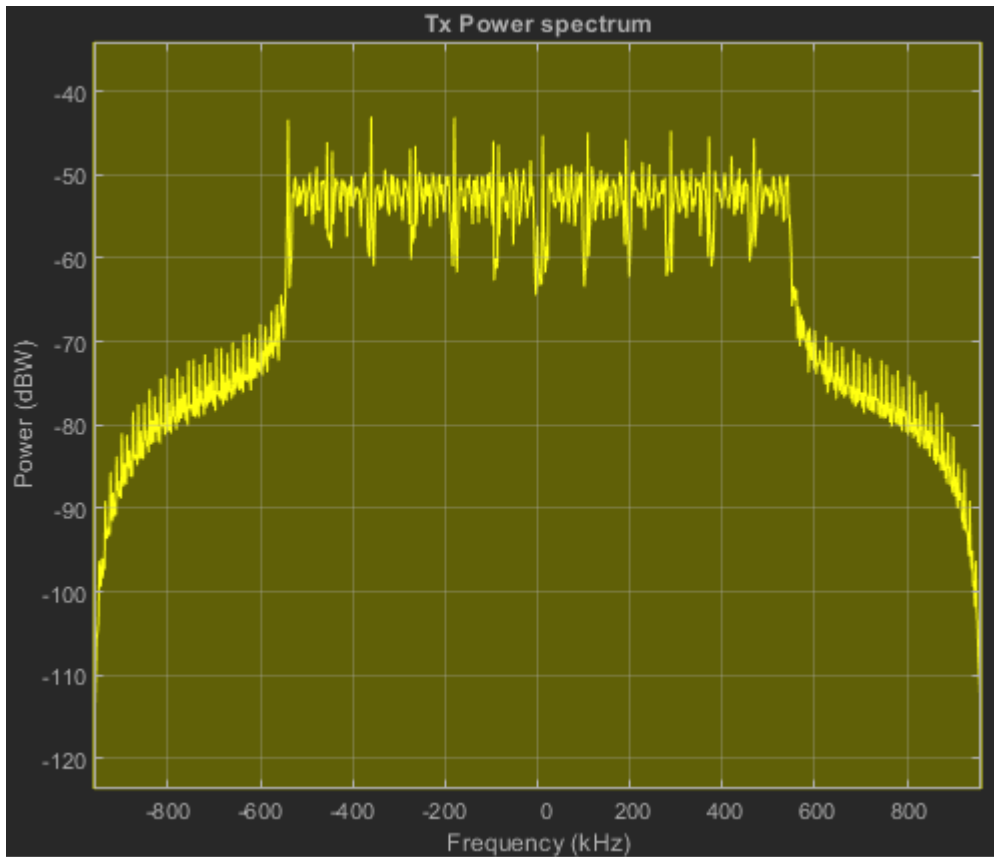
```
### Starting serial model reference simulation build  
### Model reference simulation target for whd1OFDMTx is up to date.
```

```
Build Summary
```

```
0 of 1 models built (1 models already up to date)  
Build duration: 0h 1m 36.745s
```







### HDL Code Generation

To generate HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the OFDMTx subsystem. Testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for the Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization is shown in the table below. Maximum frequency of operation is 227 MHz.

Resources	Usage
Slice Registers	5819
Slice LUT	3631
RAMB36	5
RAMB18	12
DSP48	24

### See Also

#### Blocks

Convolutional Encoder | Discrete FIR Filter HDL Optimized | General CRC Generator HDL Optimized | LTE Symbol Modulator | OFDM Modulator | Puncturer | Serializer1D



**Related Examples**

- “HDL OFDM Receiver” on page 5-136
- “HDL OFDM MATLAB References” on page 5-107

## HDL OFDM Receiver

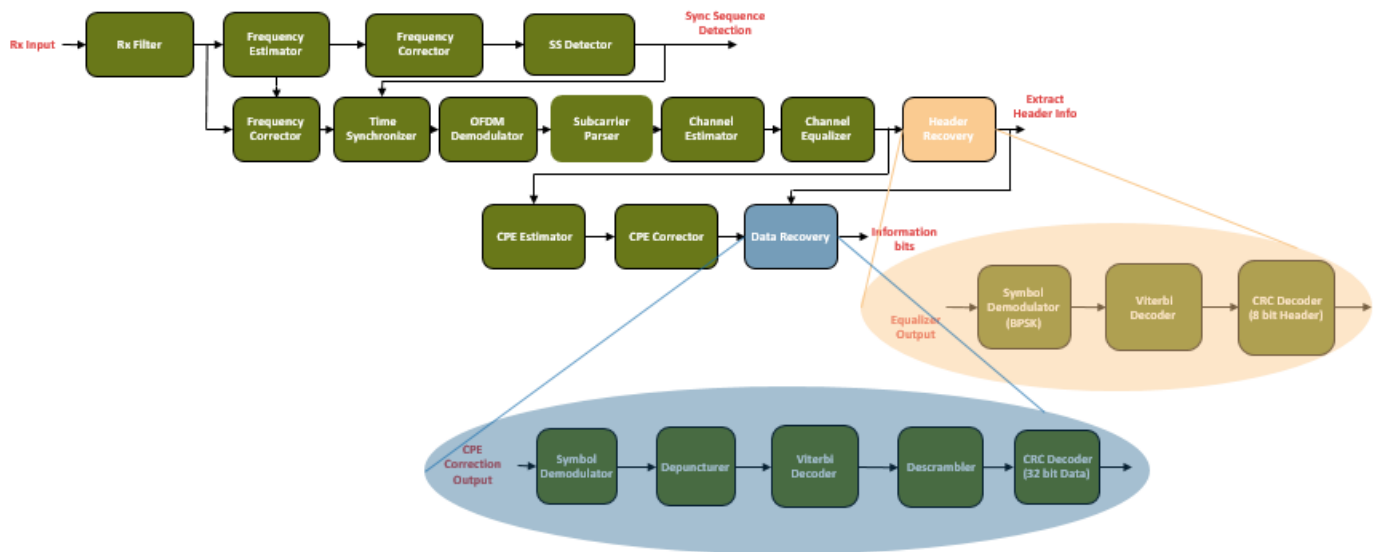
This example shows an OFDM-based wireless receiver implemented using Simulink® blocks optimized for HDL code generation and hardware implementation.

The model shown in this example receives data and decodes it based on the orthogonal frequency division multiplexing (OFDM). The main purpose of this example is to model a custom HDL OFDM wireless communication receiver that can recover information in a real-time scenario and supports data rates up to 3 Mbps. This model enables you to configure parameters: symbol modulation types such as BSPK, QPSK, 16-QAM, and 64-QAM and code rates 1/2, 2/3, 3/4 and 5/6 through punctured convolution encoding. This model enables you to control impairments such as carrier frequency offset (CFO), carrier phase offset (CPO), and rayleigh fading channel, which significantly affect OFDM-based communication system.

The receiver in this example works in conjunction with the transmitter in the **HDL OFDM Transmitter** example. For more information on the transmitter and the transmitted frame format, see the “HDL OFDM Transmitter” on page 5-121 example. The receiver in this example has a MATLAB® floating point equivalent function described in the “HDL OFDM MATLAB References” on page 5-107 example.

### Model Architecture

The following figure shows the architecture of an OFDM Receiver. The receiver samples the input at 1.92 Msps. These samples stream into the Rx Filter. The output from the Rx Filter stream into the Frequency Estimator and the Frequency Corrector. The Frequency Estimator and the Frequency Corrector estimate and correct CFO respectively and the samples stream into the Synchronizing Sequence (SS) Detector. The output of the SS Detector is used for the time synchronization. The time synchronized samples stream into the OFDM Demodulator, which demodulates the input and generates the frequency-domain subcarriers. The Subcarrier Parser parses the channel reference subcarriers, header subcarriers, and data subcarriers. The channel reference subcarriers stream into the Channel Estimator, which estimates the channel frequency response. The Channel Equalizer uses these estimates to equalize the header and data subcarriers in the frequency domain. The Header Recovery recovers the header information using the channel-equalized header subcarriers. The CPE Estimator estimates the common phase error (CPE) in the data sub carriers that get corrected by CPE Corrector. The Data Recovery uses the header information and the CPE-corrected data subcarriers to decode the data bits.



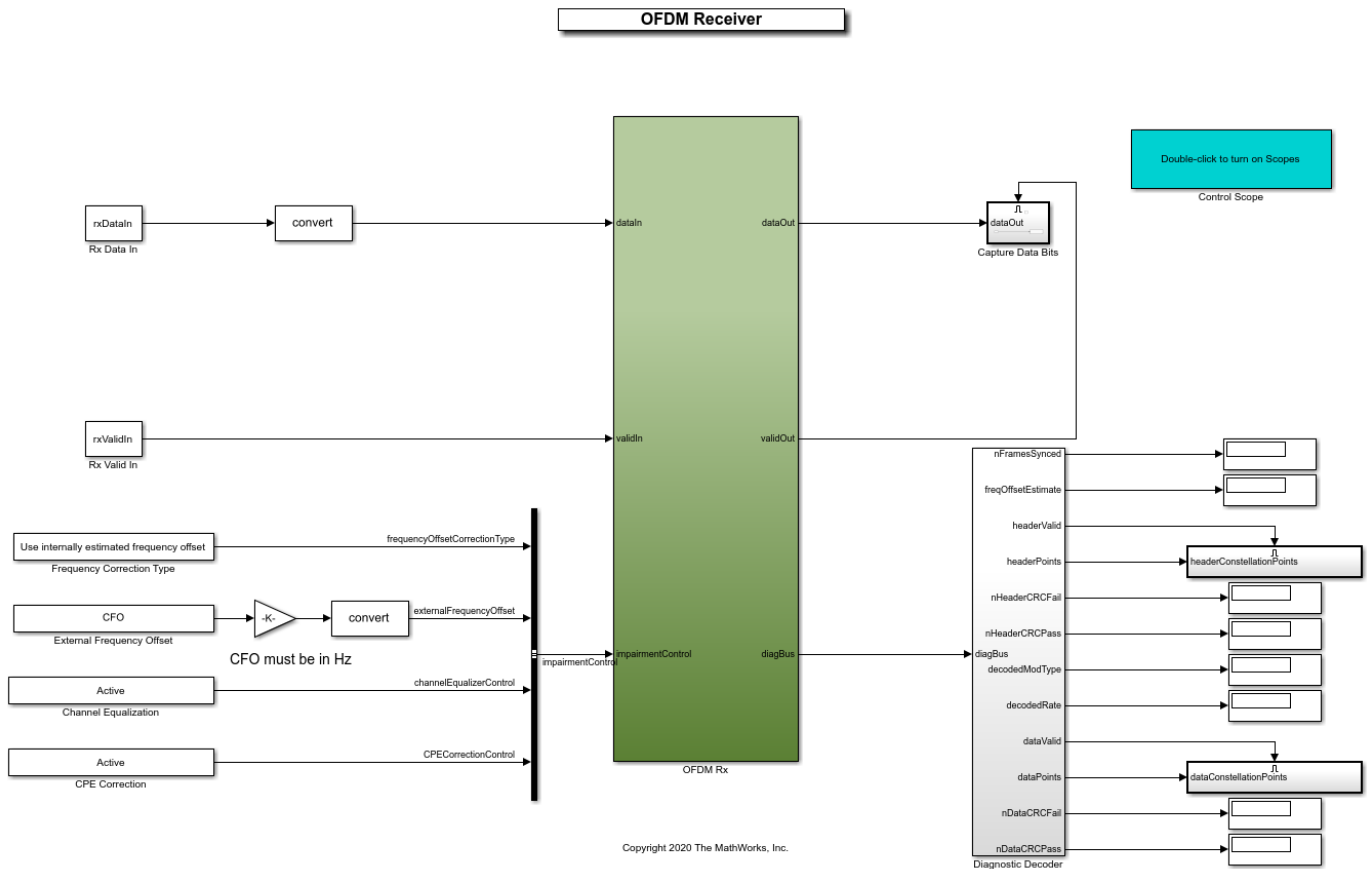
## File Structure

Two Simulink models and three MATLAB files are used to construct this example.

- `whdLOFDMReceiver.slx` — Top level OFDM receiver Simulink model
- `whdLOFDMRx.slx` — Reference model used by the `whdLOFDMReceiver.slx`
- `whdlexamples.OFDMReceiverInit.m` — Initialization script for `whdLOFDMReceiver.slx` initialized in the model's `InitFcn` callback.
- `whdlexamples.OFDMRxParameters.m` — Initialization function for `whdLOFDMRx.slx` initialized in the Model Workspace and model's `InitFcn` callback
- `whdlexamples.OFDMTx.m` — MATLAB floating point equivalent transmitter function to generate Tx waveform in `whdlexamples.OFDMReceiverInit.m`

## Receiver Interface

This figure shows the top-level model in this example.



**Model Inputs:**

- *dataIn* — Input data, specified as a complex signed 16-bit signal sampled at 1.92 Msps.
- *validIn* — Control signal to validate the *dataIn*, specified as a Boolean scalar.
- *impairmentControl* — Bus signal to control the channel impairments.

The *impairmentControl* bus comprises following signals:

- *frequencyOffsetCorrectionType* — Control signal to indicate whether to use internally estimated frequency offset or use externally provided frequency offset for offset correction, specified as Boolean scalar.
- *externalFrequencyOffset* — Real signed 14-bit CFO with range from -7400 Hz to 7400 Hz provided externally for CFO correction.
- *channelEqualizerControl* — Control signal to indicate whether to enable or disable channel equalization, specified as a Boolean scalar.
- *CPECorrectionControl* — Control signal to indicate whether to enable or disable CPE correction, specified as a Boolean scalar.

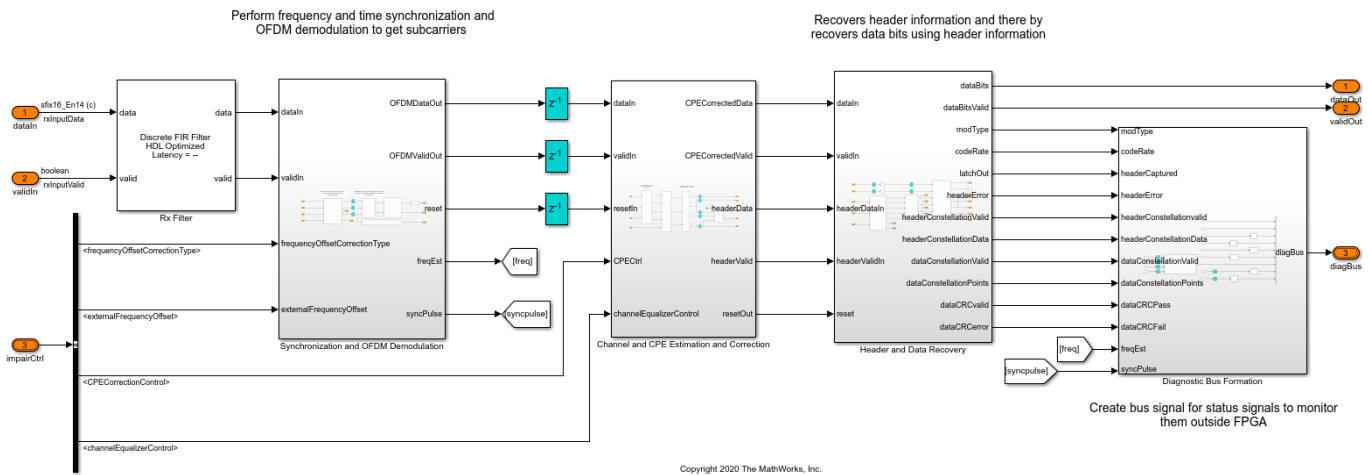
**Model Outputs:**

- *dataOut* — Decoded output data bits, returned as a Boolean scalar.
- *validOut* — Control signal to validate the *dataOut*, returned as a Boolean scalar.

- *diagBus* — Status signal with diagnostic outputs, returned as a Bus signal.

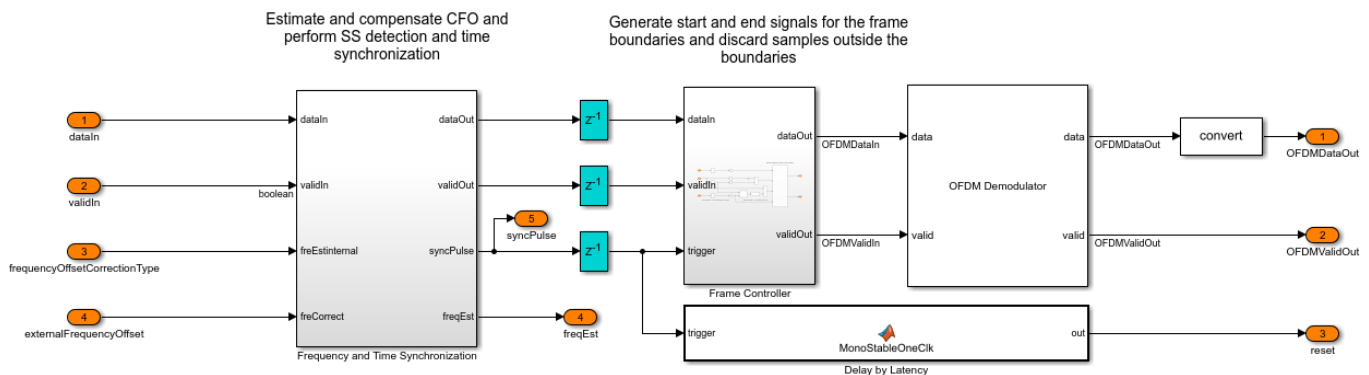
### Structure of the Receiver

The OFDM Rx subsystem performs a set of operations in a sequence. This subsystem uses the `whd\OFDMRx.slx` reference model. This reference model is initialized in its Model Workspace and model's `InitFcn` callback using the `whd\examples.OFDMRxParameters` function. The following figure shows the top-level subsystems in the reference model.

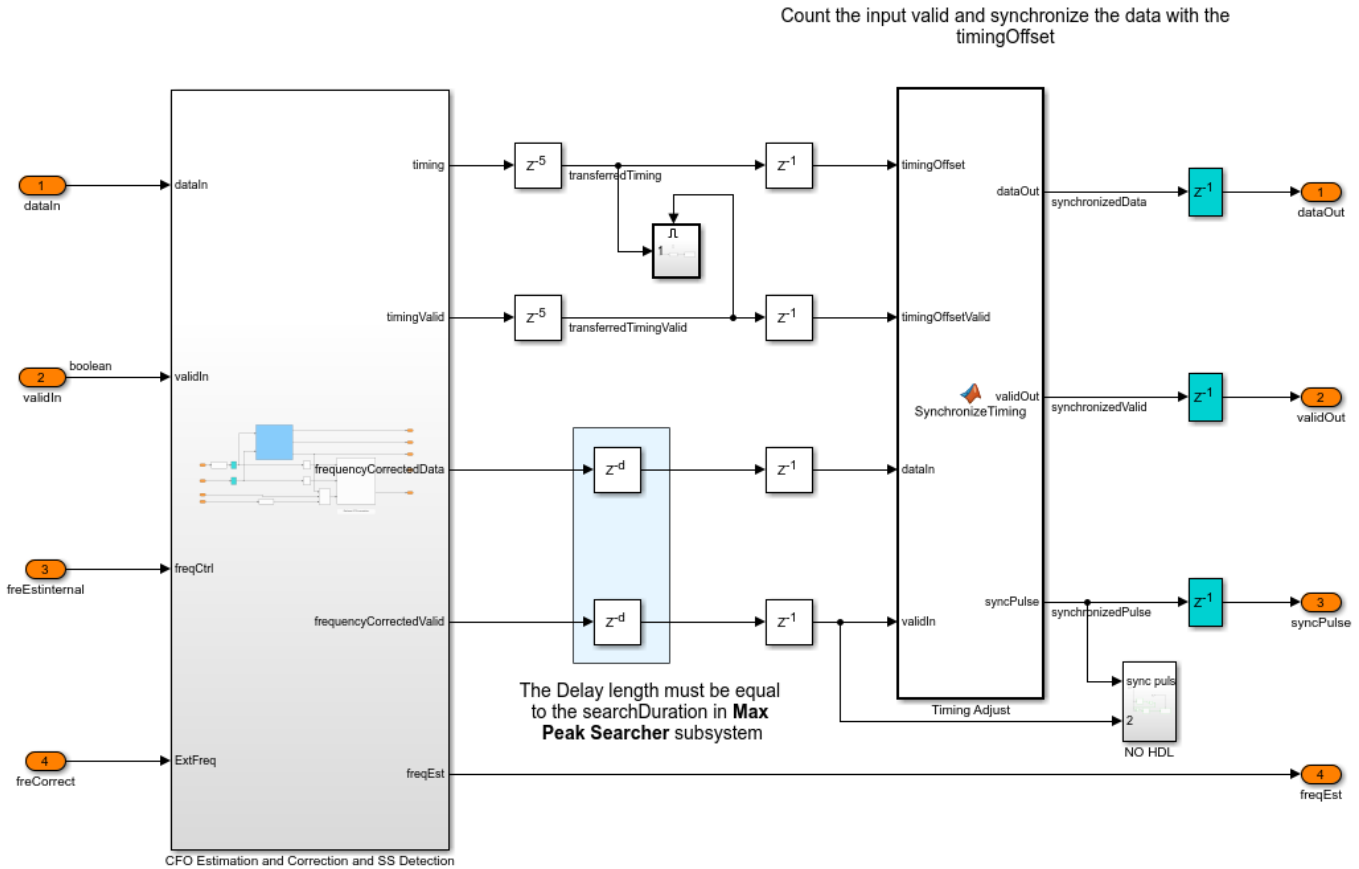


### Synchronization and OFDM Demodulation

The Synchronization and OFDM Demodulation subsystem performs frequency and time synchronizations and OFDM demodulation.



The Frequency and Time Synchronization subsystem comprises Timing Adjust MATLAB function block and CFO Estimation and Correction and SS Detection subsystem.

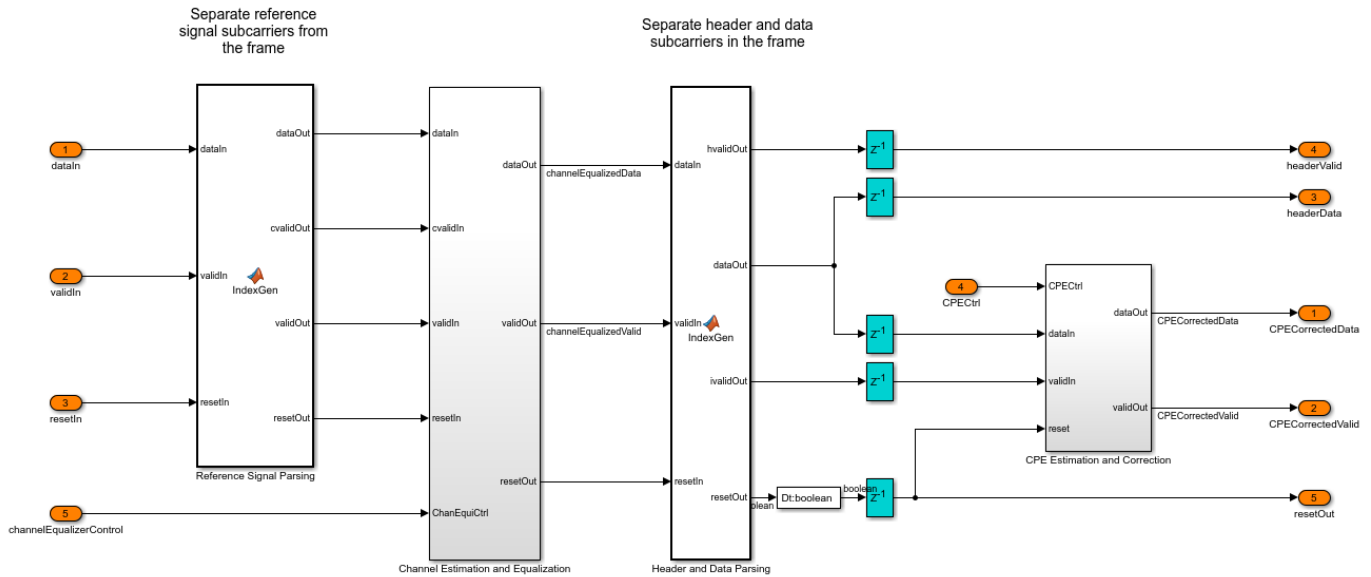


The CFO Estimation and Correction and SS Detection subsystem comprises CFO Estimation and SS Detection subsystem and Frequency Correction Nx subsystem, which perform frequency correction for the input signal. The estimate from the CFO Estimation and SS Detection subsystem is used for frequency correction if the *frequencyOffsetCorrectionType* signal on the top-level model interface is set to use internally estimated frequency offset. The *externalFrequencyOffset* is used for frequency correction if the *frequencyOffsetCorrectionType* signal is set to use externally provided frequency offset.



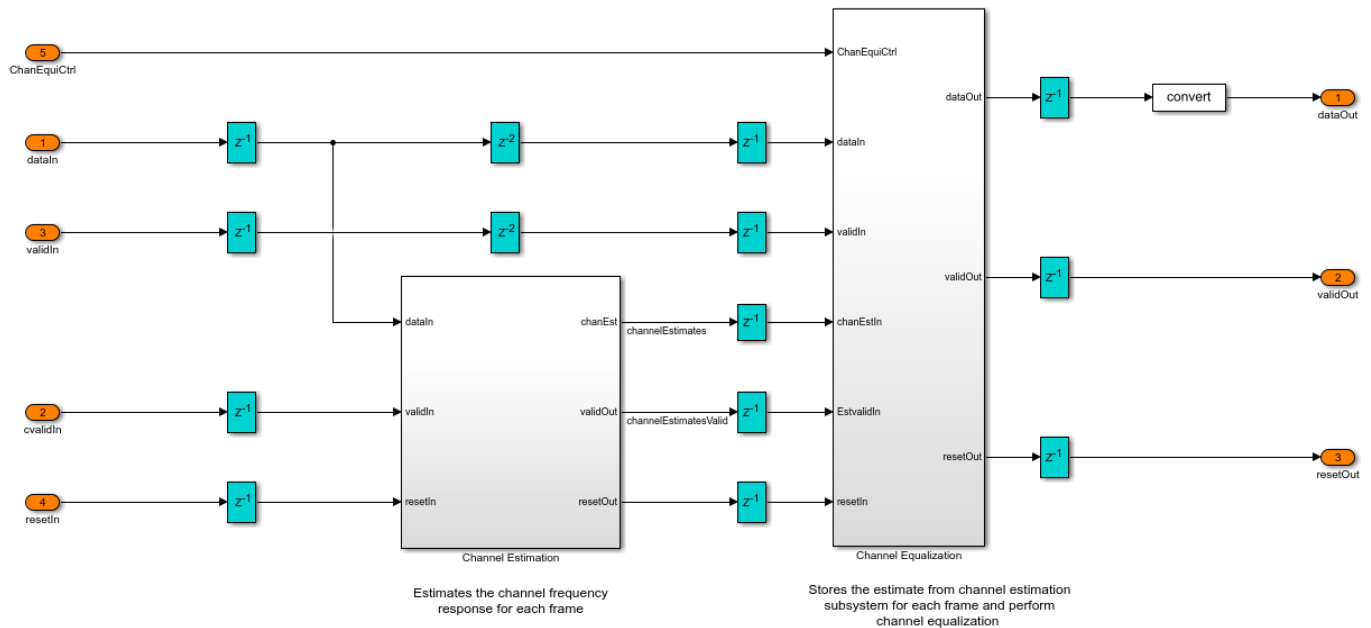






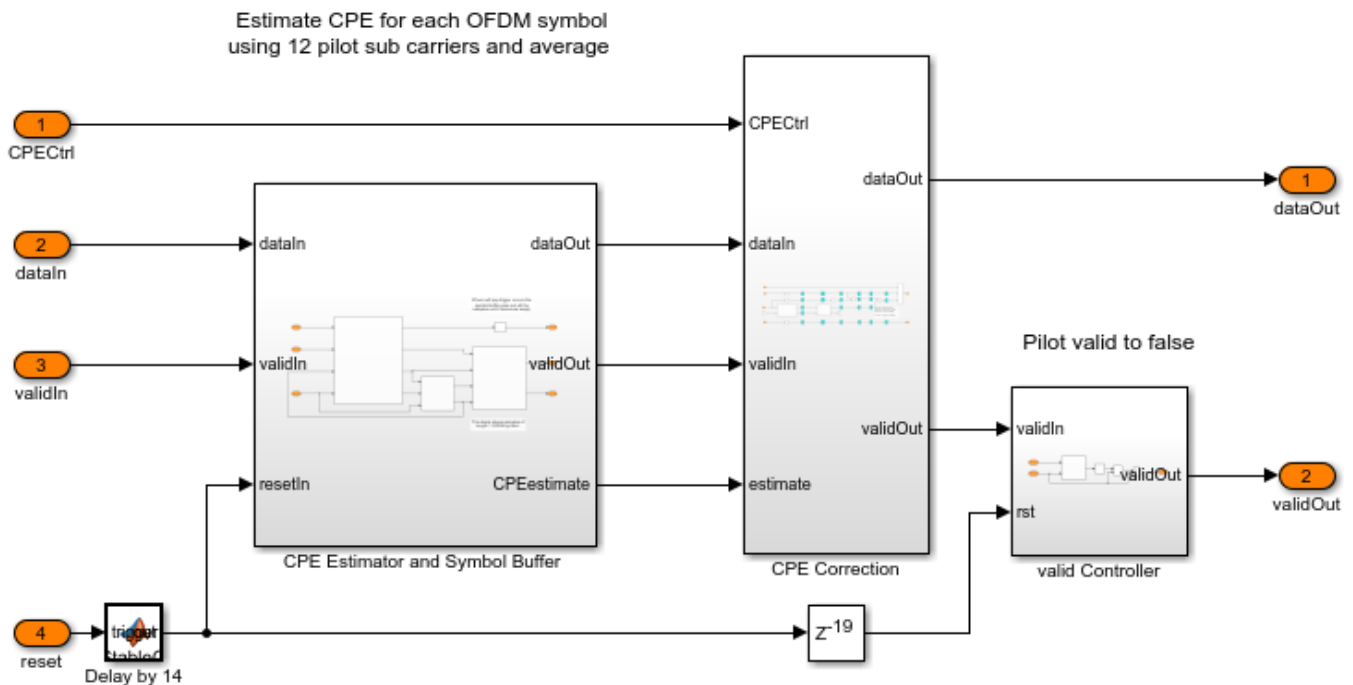
The Reference Signal Parsing MATLAB function block separates the OFDM symbols reserved for computing channel estimates.

The OFDM symbols reserved for computing channel estimates are streamed through Channel Estimation subsystem. The OFDM Channel Estimator block averages the estimates from the two symbols and outputs the final channel estimates. The estimates are streamed into the Channel Equalization subsystem, which stores the estimates in a RAM and performs frequency-domain channel equalization for all the remaining OFDM symbols in the frame.



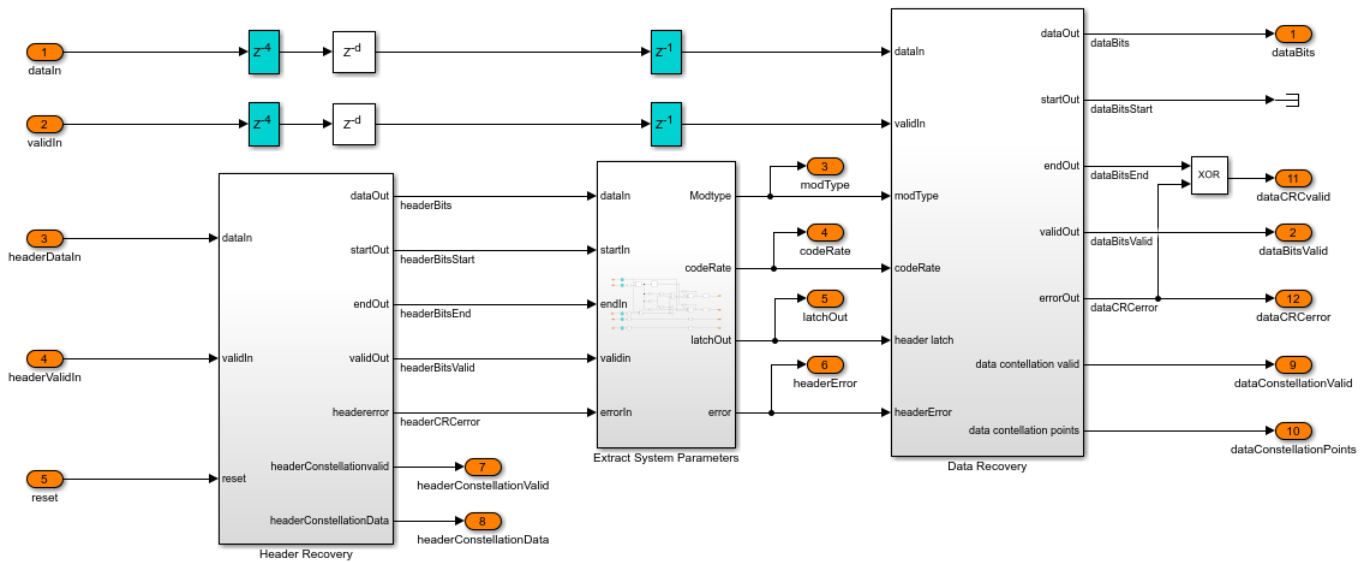
The Header and Data Parsing MATLAB function block separates the OFDM symbols corresponding to header and data symbols.

The frequency domain channel-equalized data subcarriers stream through the Common Phase Error Estimation and Correction subsystem. In the frequency estimation process, there is always a small estimation error due to the channel impairments. This estimation error results in a residual frequency offset in the channel-equalized subcarriers. This results a CPE in all the subcarriers in an OFDM symbol and changes from symbol to symbol. The CPE Estimation subsystem estimates the CPE on each OFDM symbol using the 12 pilot subcarriers. The pilots are the known subcarriers and any phase rotation in the received symbols is estimated by using the pilots. The estimates drawn from the same symbol are averaged to get the final estimate. The symbol is stored in the Symbol Buffer MATLAB function block during estimation. Once the estimate is ready, the symbol is read from this buffer block and the CPE Correction subsystem corrects the CPE in the data subcarriers with that estimate.

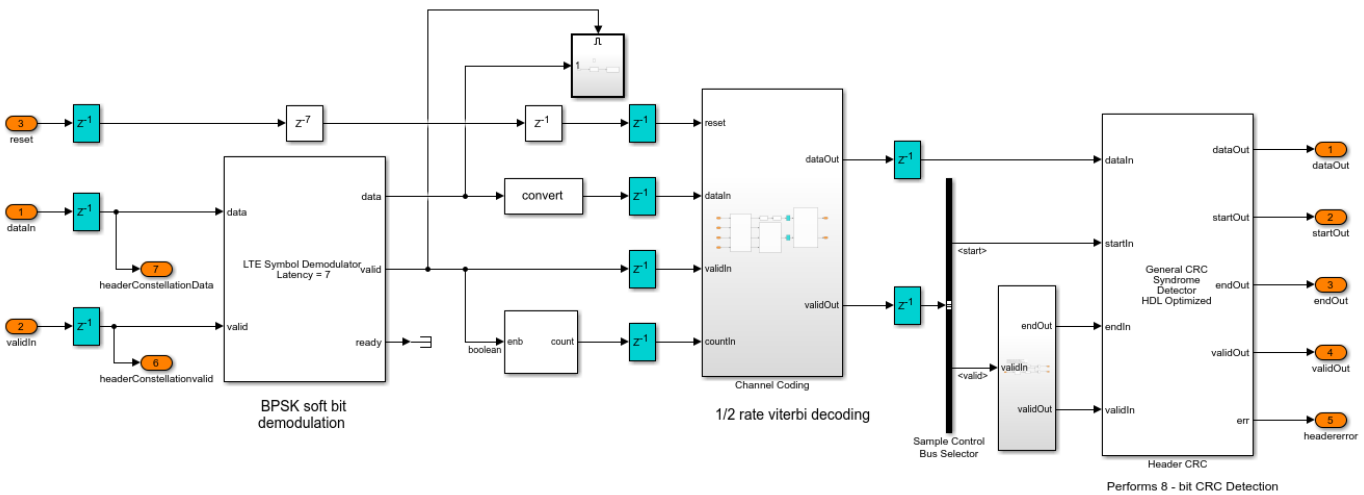


### Header and Data Recovery

The Header and Data Recovery subsystem recovers the header information and the data bits.

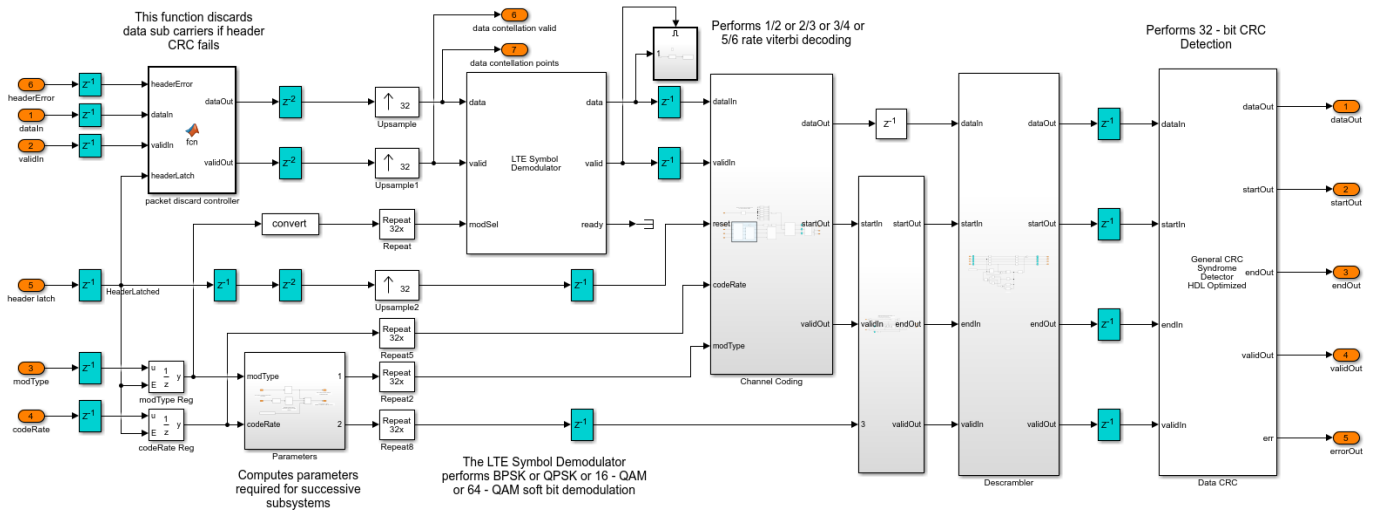


The Header Recovery subsystem recovers the header information to decode data bits. The frequency domain channel-equalized header subcarriers stream into the Header Recovery subsystem. The LTE Symbol Demodulator block performs BPSK soft symbol demodulation. The Channel Coding subsystem is equipped with Viterbi Decoder block, which performs 1/2 rate viterbi decoding. The General CRC Syndrome Detector HDL Optimized block performs 8-bit CRC checksum and validates the decoded bits from the Viterbi Decoder block. The CRC syndrome detector block generates an error signal, if the CRC checksum fails.



The Data Recovery subsystem uses the header information to decode the data bits. The header information is stored in the registers. These registers are used to access the header information. The LTE Symbol Demodulator block performs soft bit BPSK, QPSK, 16-QAM or 64-QAM symbol demodulation associated with the modulation type retrieved from the header information. The Channel Coding subsystem is equipped with Depuncturer and Viterbi Decoder blocks. Each code rate is assigned a predefined punctured vector pattern. Based on the code rate retrieved from the header information, the Channel Coding subsystem performs depuncturing followed by viterbi decoding. The

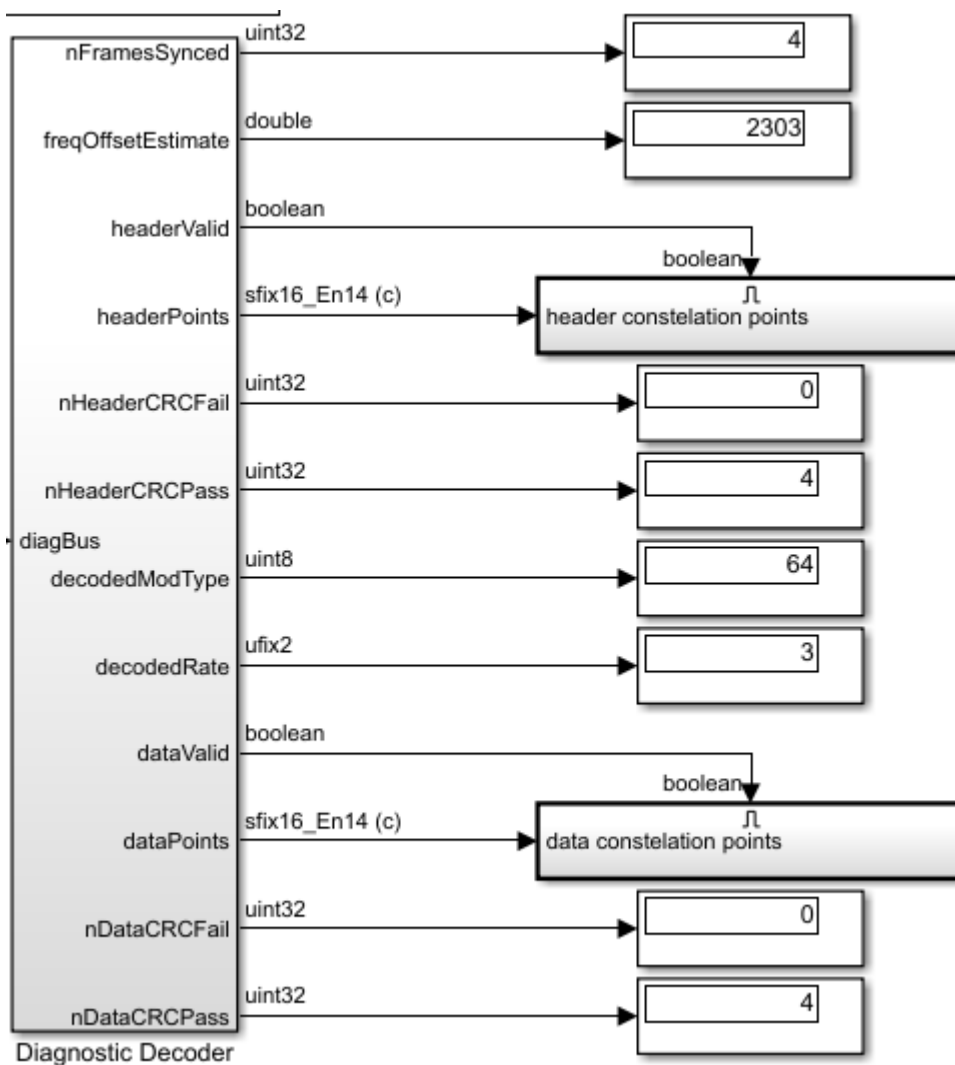
decoded bits are streamed through the Descrambler subsystem. The General CRC Syndrome Detector HDL Optimized block performs 32-bit CRC checksum and validates the descrambled bits. The CRC syndrome detector block generates an error signal, if the CRC checksum fails.



## Diagnostic Bus Formation

The Diagnostic Bus Formation subsystem creates a bus signal for some status signals of the receiver. This bus can be used to analyze the receiver when deployed onto the hardware.

The data bits are decoded in the Data Recovery subsystem and bits stream out of the receiver and are stored to workspace in the Capture Data Bits subsystem in the top-level receiver model. The Diagnostics Decoder subsystem decodes the source-coded header information and counts the number of synchronized frames, number of header CRC passes and failures, and the number of data CRC passes and failures in the bus signal formed in the Diagnostic Bus Formation subsystem. The Simulink display blocks display the Diagnostics Decoder information.



## Run the Receiver

Connect the receiver back-to-back with the transmitter in the “HDL OFDM Transmitter” on page 5-121 example and run the Simulink model. For more information on how to connect the transmitter and the receiver Simulink models back-to-back see the “HDL OFDM MATLAB References” on page 5-107 example.

The following files describe a procedure to initialize, generate inputs, run, and verify the `whdlofdmReceiver.slx` model using the `whdlexamples.OFDMReceiverInit.m` initialization script. You can choose a custom Tx waveform and a channel impairment of your choice from the Custom Frame Configuration section in these files.

- `OFDMRxRealTimeSimulationDisplay.m` - This script mimics a channel in a real-time scenario. You can choose any available channel impairment and run the script. The script displays the outputs and generates plots of estimated frequency offset and SS correlation.
- `OFDMRxFadingChannelResponseDisplay.m` - This script mimics only the fading channel. You can choose only the fading channel impairment and run the script. The script displays the outputs

and generates the plots of channel impulse response and the comparison of estimated frequency response with the frequency response, derived from the impulse response.

**NOTE:** These files are not available on the MATLAB search path. To copy these files locally to the user path, you must open this example.

### Verification and Results

The `whdlexamples.OFDMRx.m` script is a MATLAB floating point equivalent of the reference model `whdLOFDMRx.slx`. The Simulink model and MATLAB floating point equivalent script are compared in the “HDL OFDM MATLAB References” on page 5-107 example.

Run the `OFDMRxRealTimeSimulationDisplay.m` script to run the receiver.

```
>> OFDMRxRealTimeSimulationDisplay
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: whdLOFDMRx

Build Summary

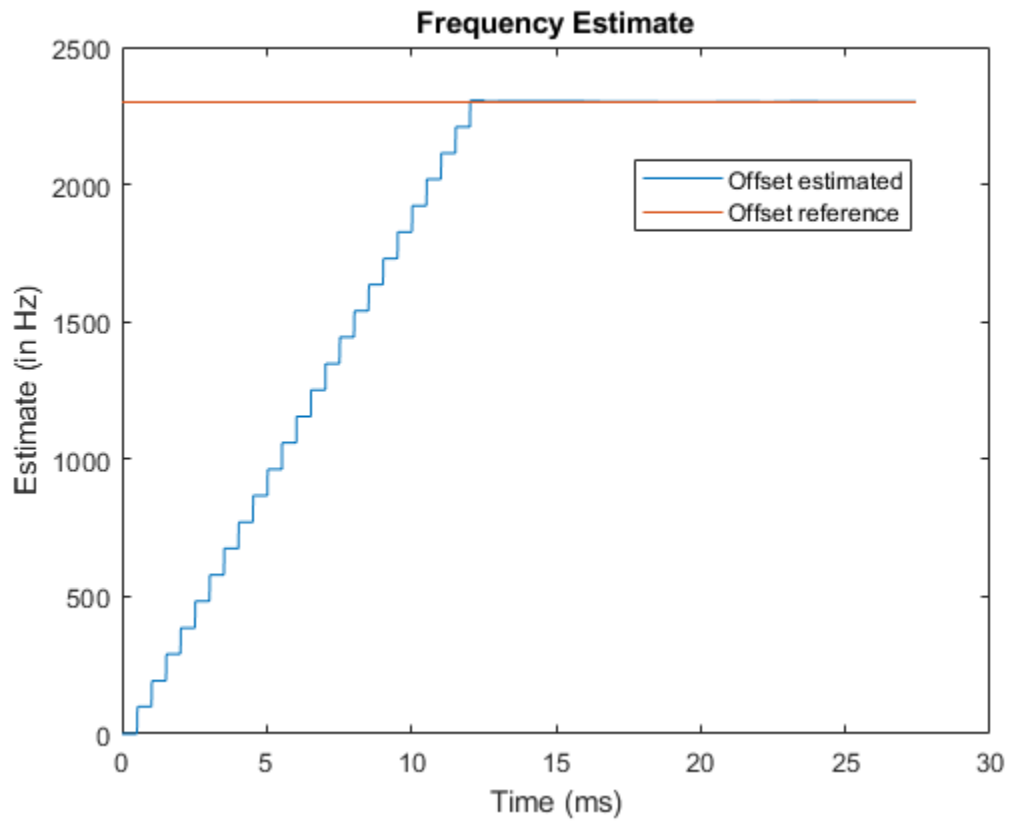
Simulation targets built:

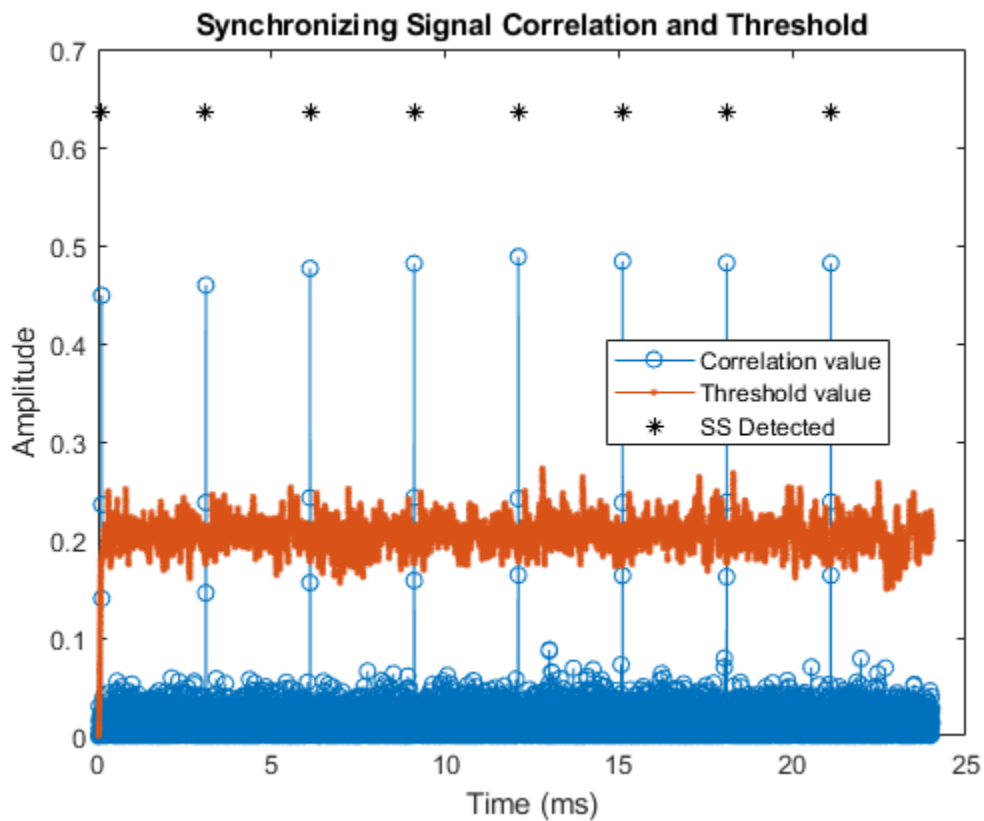
Model          Action                                     Rebuild Reason
=====
whdLOFDMRx     Code generated and compiled whdLOFDMRx_msf.mexw64 does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 7m 34.747s

Number of header CRC failed = 0 per 4

Number of bit errors = 0 per 15208
```





Run the `OFDMRxFadingChannelResponseDisplay.m` script to run the receiver.

```
>> OFDMRxFadingChannelResponseDisplay
### Starting serial model reference simulation build

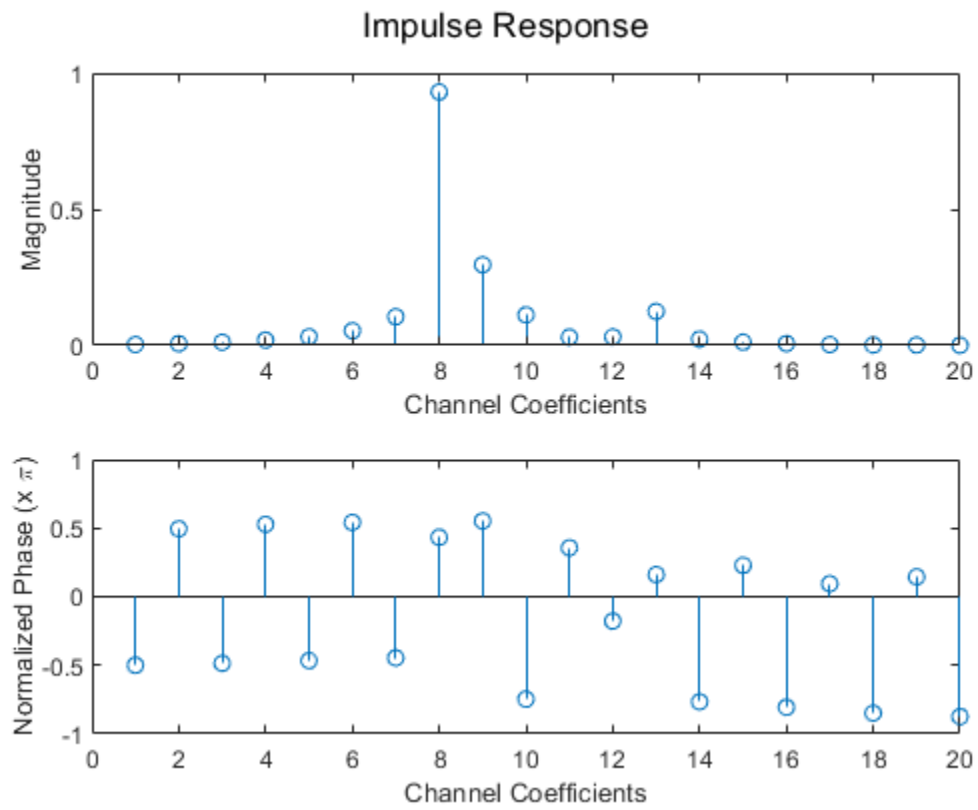
Build Summary

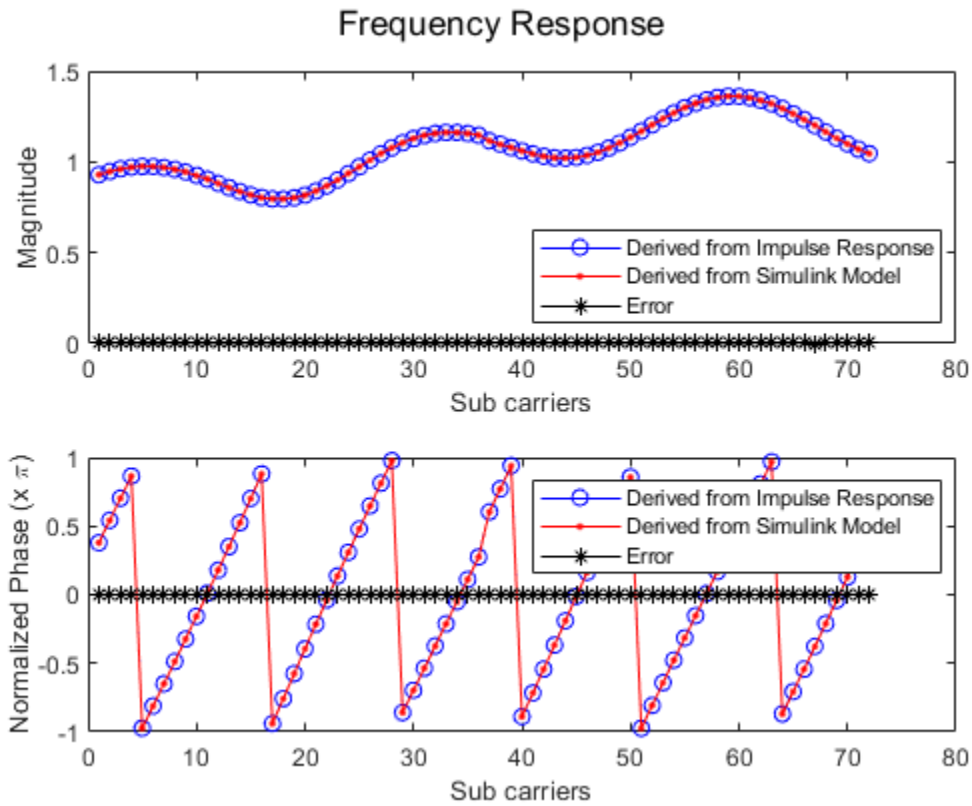
0 of 1 models built (1 models already up to date)
Build duration: 0h 2m 44.917s

Number of header CRC failed = 0 per 1

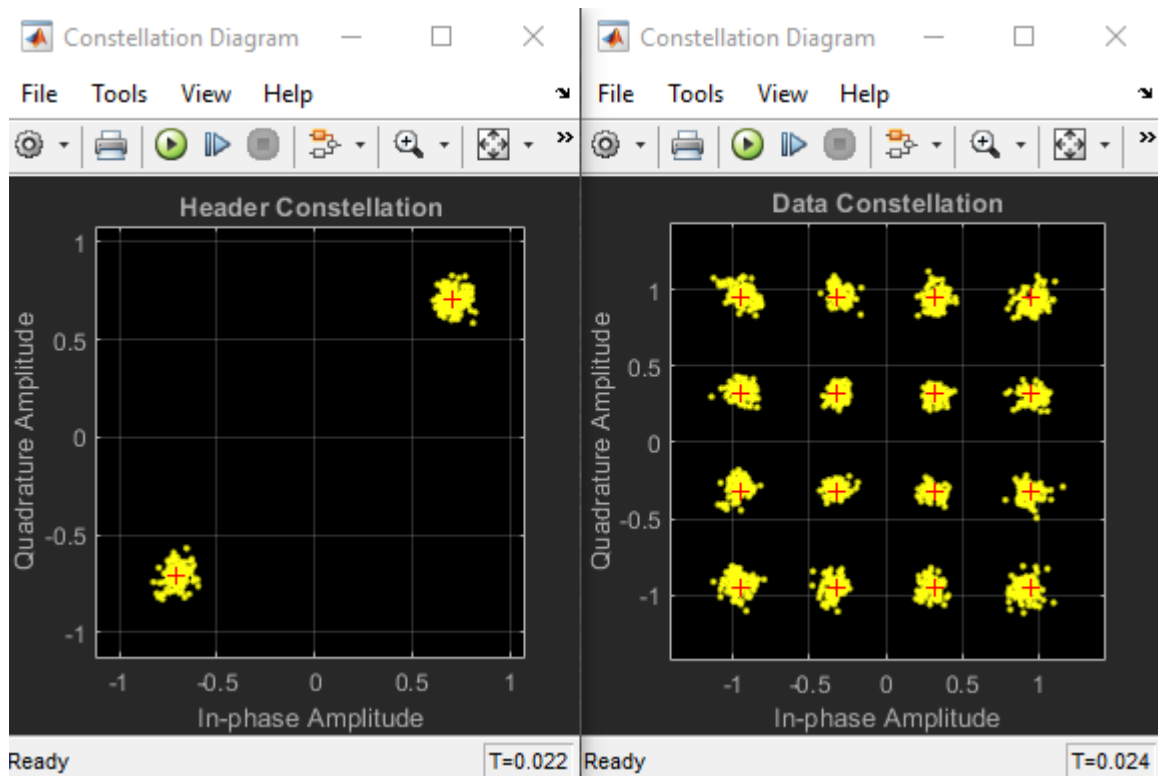
Number of bit errors = 0 per 3162
```







You can see the constellation plot on the constellation scope. The scopes can be activated by using the Control Scope button in the `whd1OFDMReceiver.slx` model.



## HDL Code Generation

To generate the HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdl tb` commands to generate HDL code and HDL testbench for the OFDM Rx subsystem. The testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization and are shown in the table below. The maximum frequency of operation is 192 MHz.

Resources	Usage
Slice Registers	45122
Slice LUT	36246
RAMB36	8
RAMB18	9
DSP48	95

## See Also

### Blocks

Depuncturer | General CRC Syndrome Detector HDL Optimized | LTE Symbol Demodulator | OFDM Channel Estimator | OFDM Demodulator | Viterbi Decoder

### **Related Examples**

- “HDL OFDM Transmitter” on page 5-121
- “HDL OFDM MATLAB References” on page 5-107