Wireless HDL Toolbox™

User's Guide

MATLAB® & SIMULINK®

R2020a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`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

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)

Contents

3 Examples

Model Architecture

Streaming Sample Interface

In this section...

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.

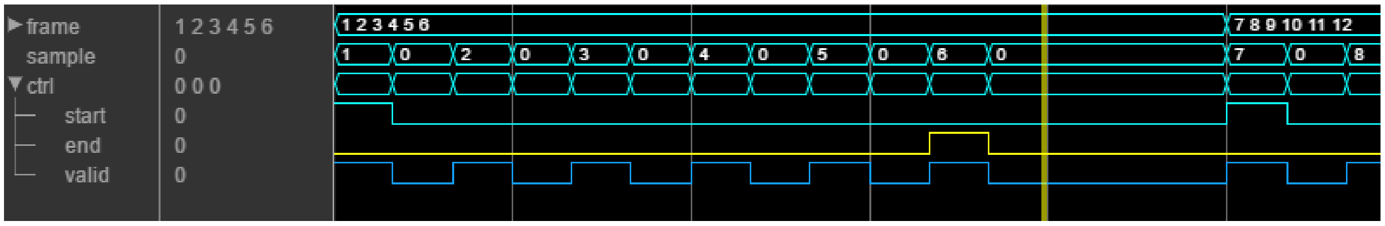
Port	Description	Data Type
`sample`	Scalar integer value that represents one sample. The protocol also allows for a vector of integer values that represent a single sample, such as for turbo-encoded samples.	Supported data types include: • `Boolean` • `uint` or `int` • `ufix` or `sfix` `double` and `single` 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.

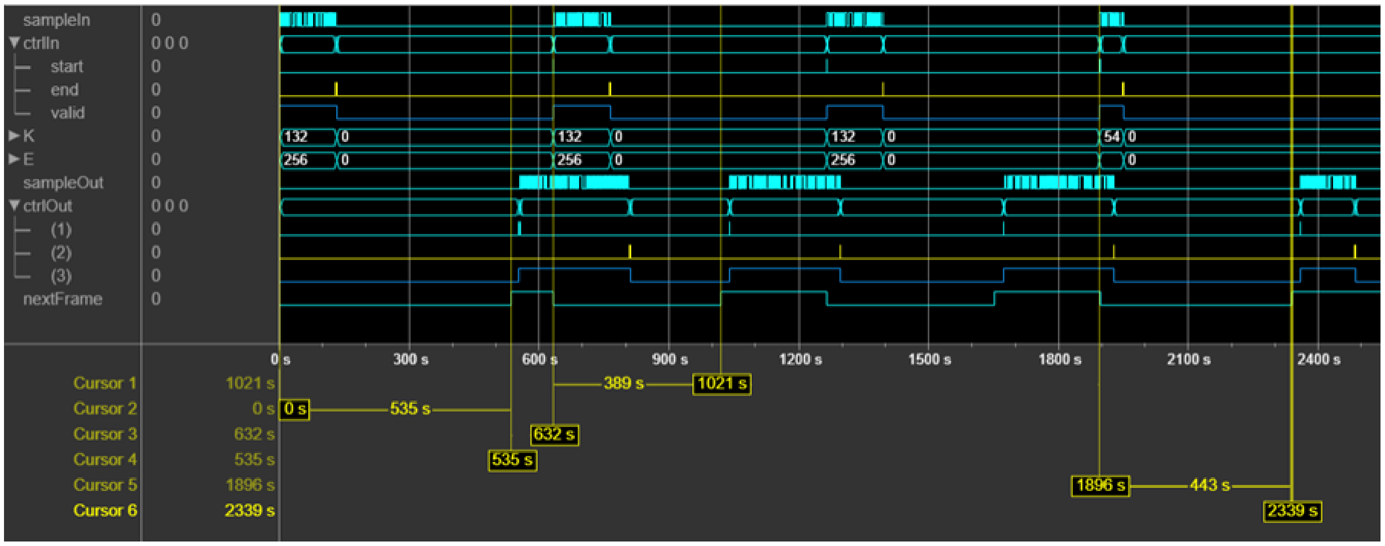
The control signals `start` and `end` are 1 for the first and last valid samples of the frame, respectively. The `valid` signal is 1 for each valid input sample. The `valid` signal is 0 for the idle cycles inserted between the samples and between the frames. The six-sample frame is now represented by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

Using the nextFrame Output Signal

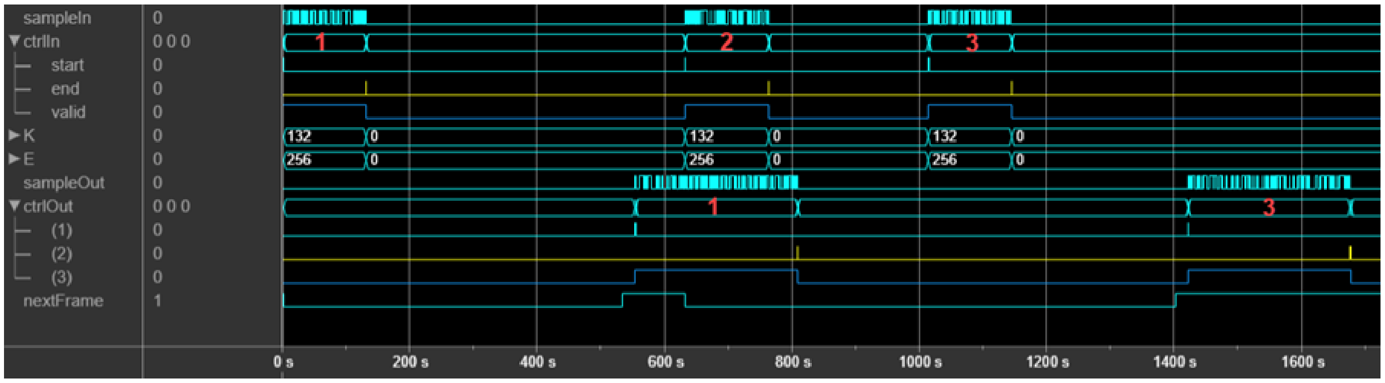
The NR Polar Encoder, NR Polar Decoder, NR LDPC Encoder, NR LDPC Decoder, and RS Decoder blocks each provide an output signal to indicate when the block is ready to receive the start of a new frame. This signal is necessary because these blocks cannot accept a new frame at certain stages of internal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

Port	Description	Data Type
nextFrame	Boolean scalar that indicates when the block can accept the start of a new frame	`Boolean`

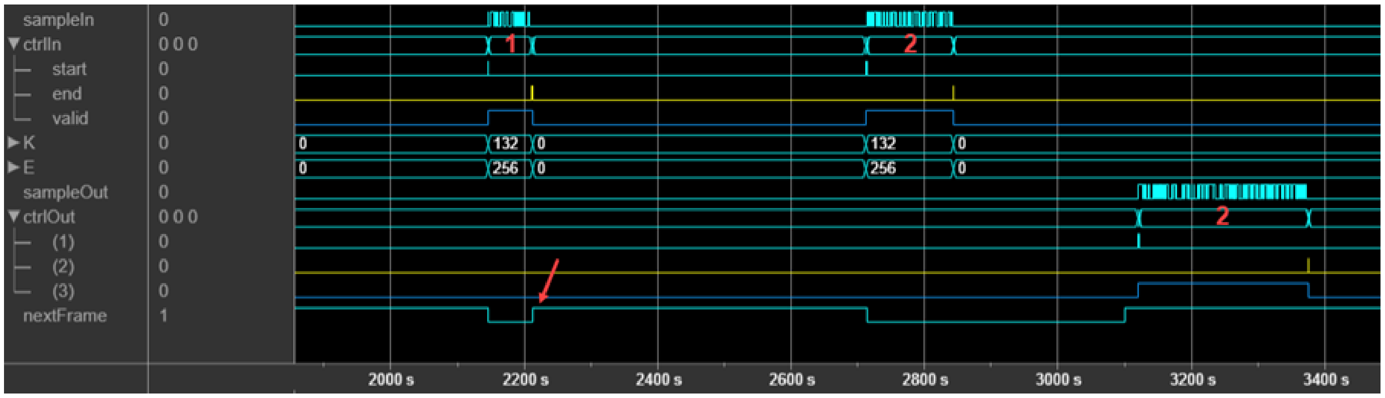
This waveform shows the NR Polar Encoder block processing several frames. The **nextFrame** output signal is `0` when the block is processing data, and `1` when the block is ready to receive the start of a new frame. The cursors show the latency varying with the values of the input **K** and **E** port 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
`start`	`true` for the first sample in the frame	`Boolean`
`end`	`true` for the last sample in the frame	`Boolean`
`valid`	`true` for any valid sample	`Boolean`

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

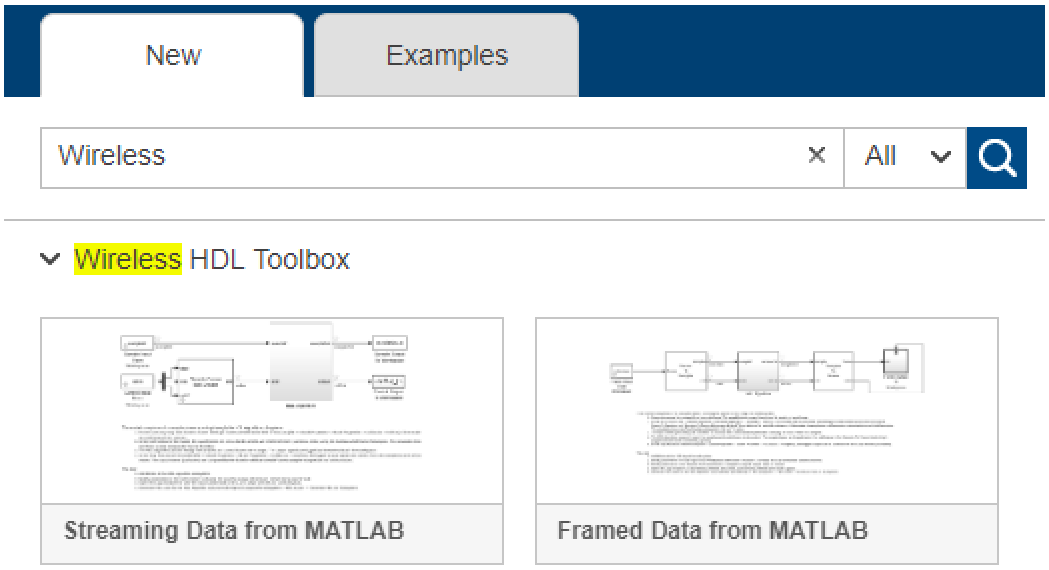
About Simulink Model Templates

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” (Simulink).

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

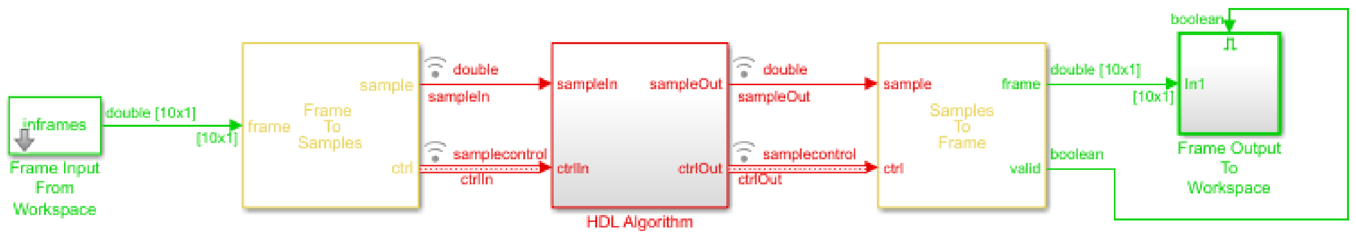
The **Framed Data from MATLAB** template imports framed data from the MATLAB workspace, assuming all frames are the same size. Then, it converts the data to a sample stream by using the Frame To Samples block.

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.



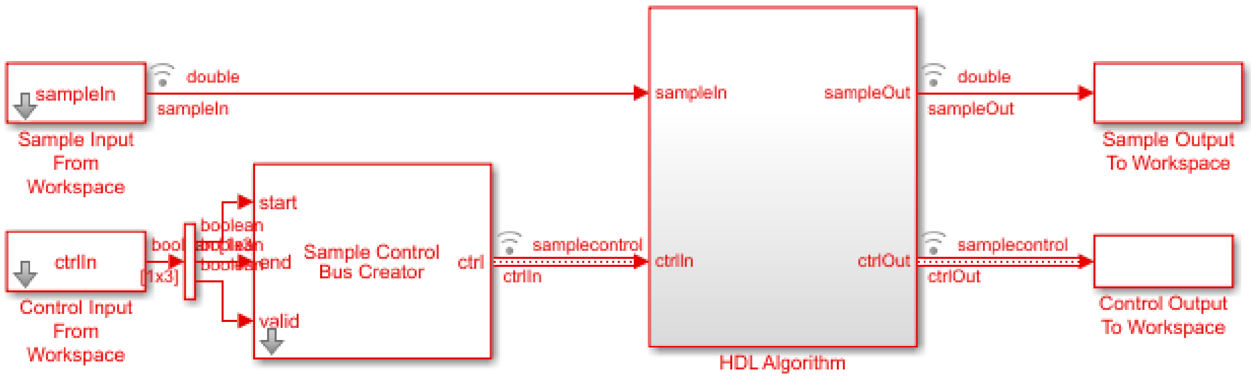
Streaming Data Template

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

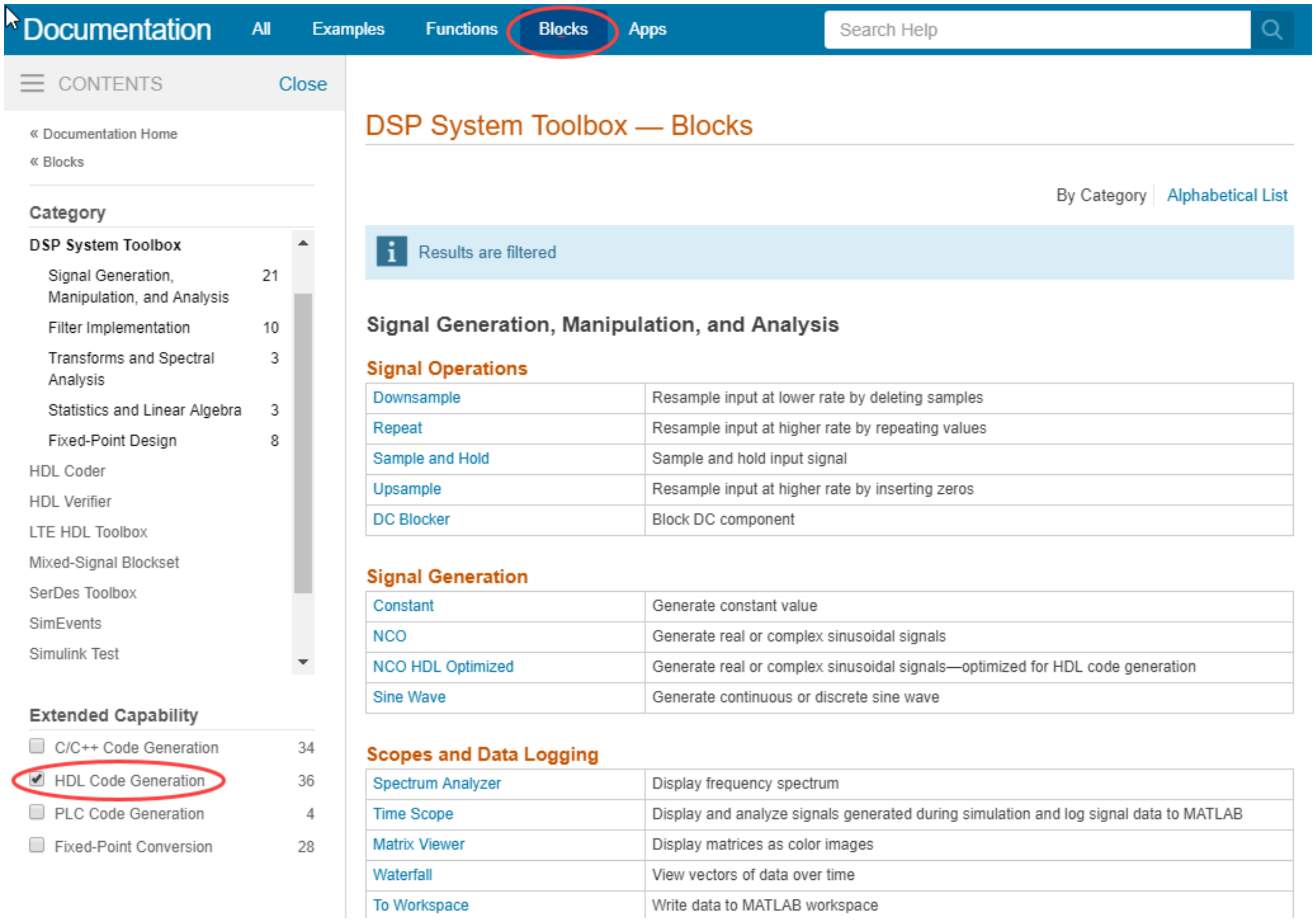
Other MathWorks® products also include blocks supported for HDL code generation that you can use to build up your design.

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

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

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

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.



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

Results are filtered

Signal Generation, Manipulation, and Analysis

Signal Operations

Downsample	Resample input at lower rate by deleting samples
Repeat	Resample input at higher rate by repeating values
Sample and Hold	Sample and hold input signal
Upsample	Resample input at higher rate by inserting zeros
DC Blocker	Block DC component

Signal Generation

Constant	Generate constant value
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals—optimized for HDL code generation
Sine Wave	Generate continuous or discrete sine wave

Scopes and Data Logging

Spectrum Analyzer	Display frequency spectrum
Time Scope	Display and analyze signals generated during simulation and log signal data to MATLAB
Matrix Viewer	Display matrices as color images
Waterfall	View vectors of data over time
To Workspace	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 Using Simulink HDL Workflow Advisor” (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.

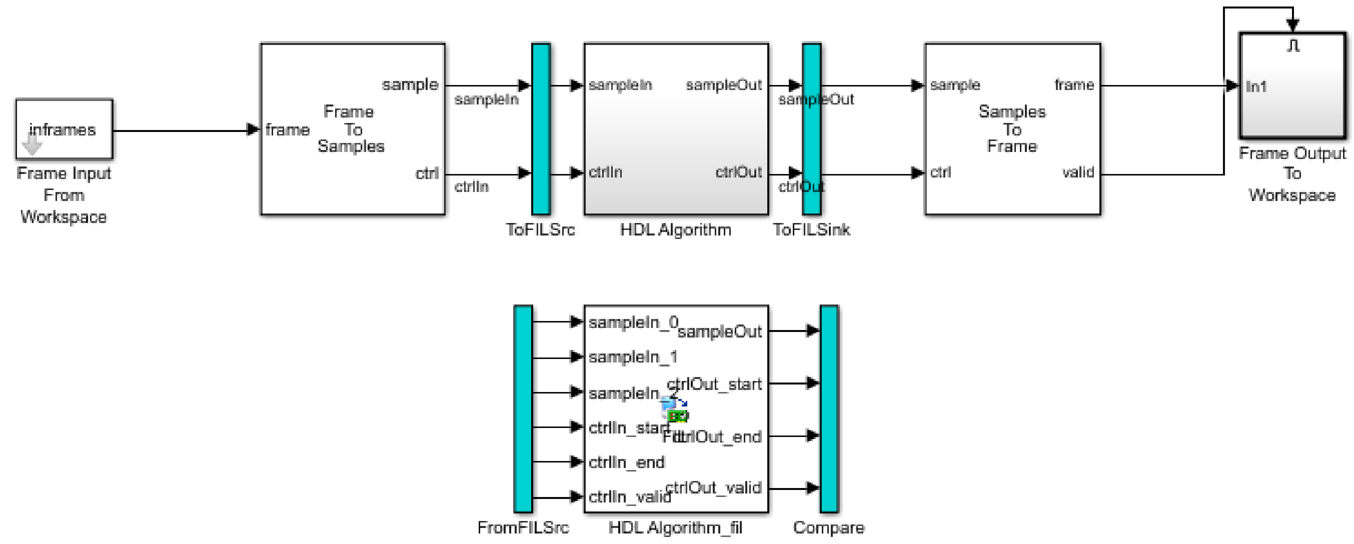
When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Wireless HDL Toolbox designs, the FIL block in that model replicates the sample-streaming interface and sends one sample at a time to the FPGA. Both 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

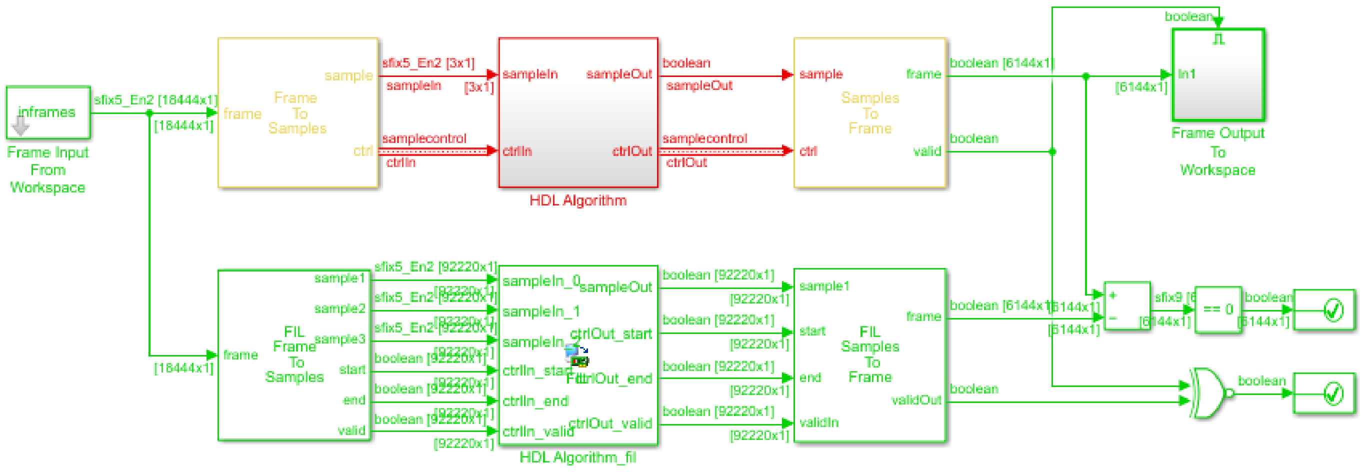
The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in Simulink. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.



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

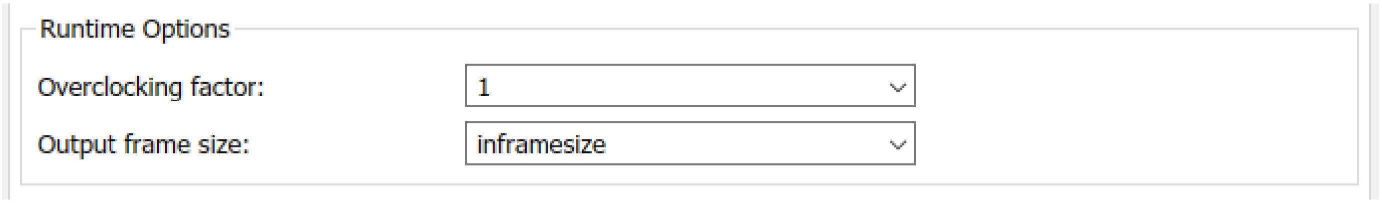
To improve the communication bandwidth with the FPGA board, modify the autogenerated model. The modified model uses the FIL Frame To Samples and FIL Samples To Frame blocks to send one frame at a time.



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: 1

Output frame size: inframesize

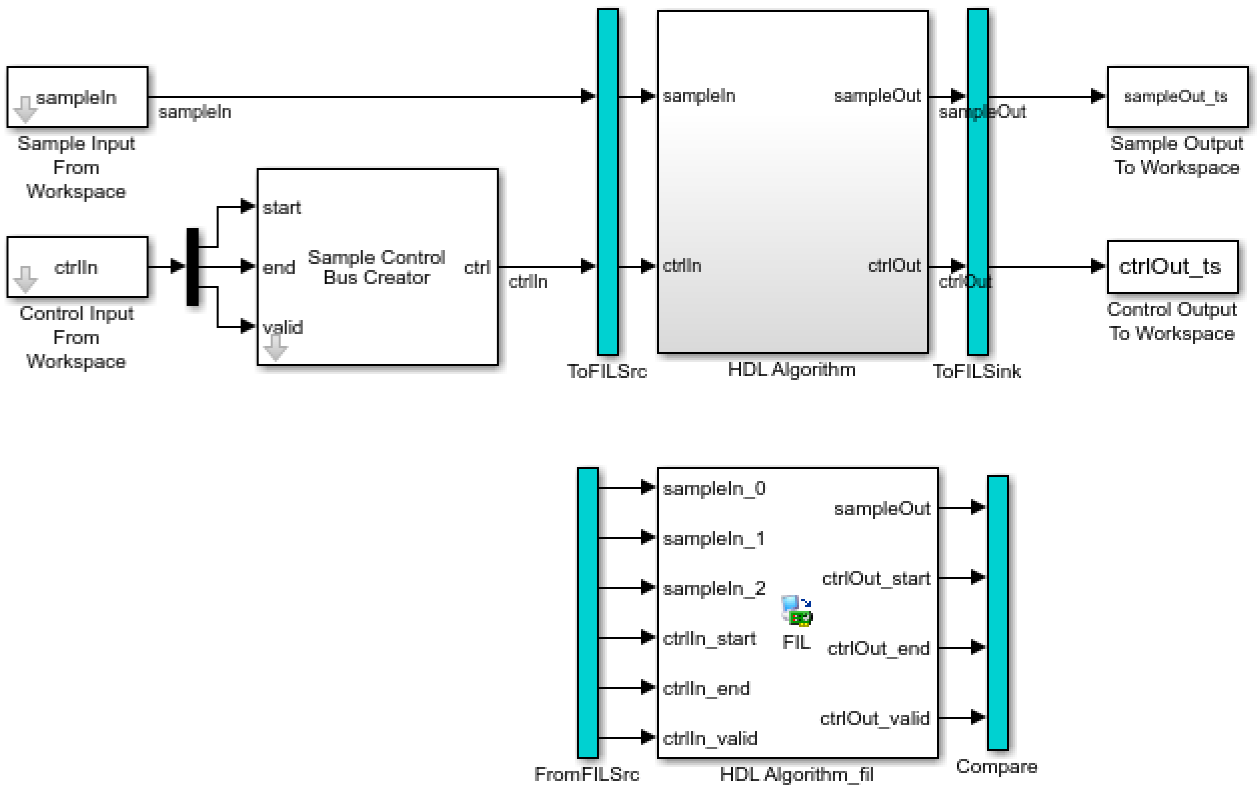
4 In the FIL Frame To Samples and FIL Samples To Frame blocks, set the parameters to match the settings of the Frame To Samples and Samples To Frame blocks.
5 Branch the frame output of the Samples To Frame block for comparison. You can compare the entire frame at once with a Diff block. Compare the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

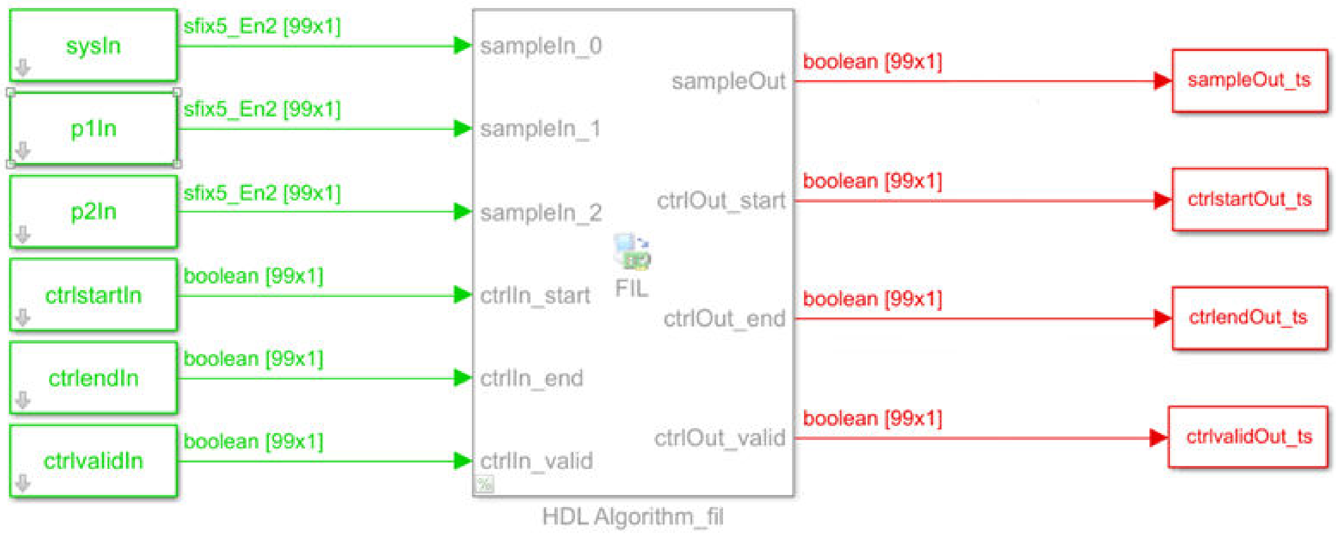
The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in MATLAB. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.



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);
```

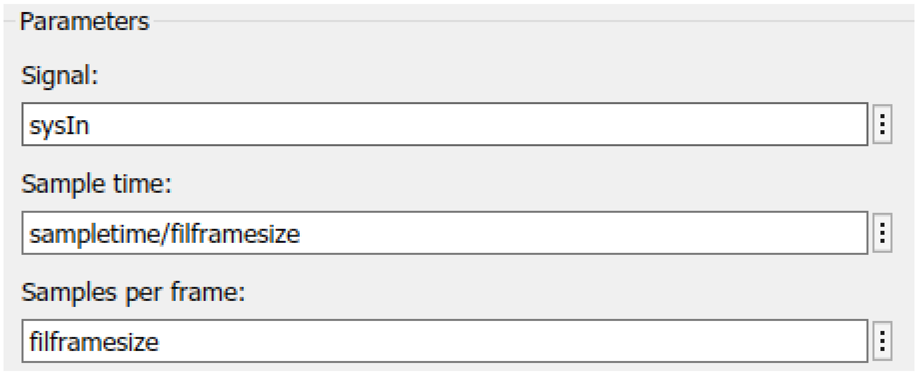
5 Reshape the output variables for input to the `whdlSamplesToFrames` function. Recreate an *N*-by-3 control signal matrix and a vector of sample data. In this example, the output sample is a single value. If the output sample is multiple values, build an *N*-by-*SampleSize* sample matrix.

```
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.

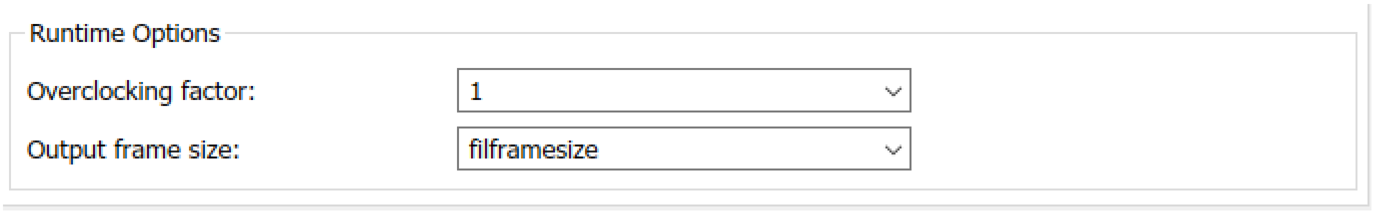

Parameters

Signal: sysIn

Sample time: sampletime/filframesize

Samples per frame: filframesize

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


Runtime Options

Overclocking factor: 1

Output frame size: filframesize

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” 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"
- "LTE HDL SIB1 Recovery"

3

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`.

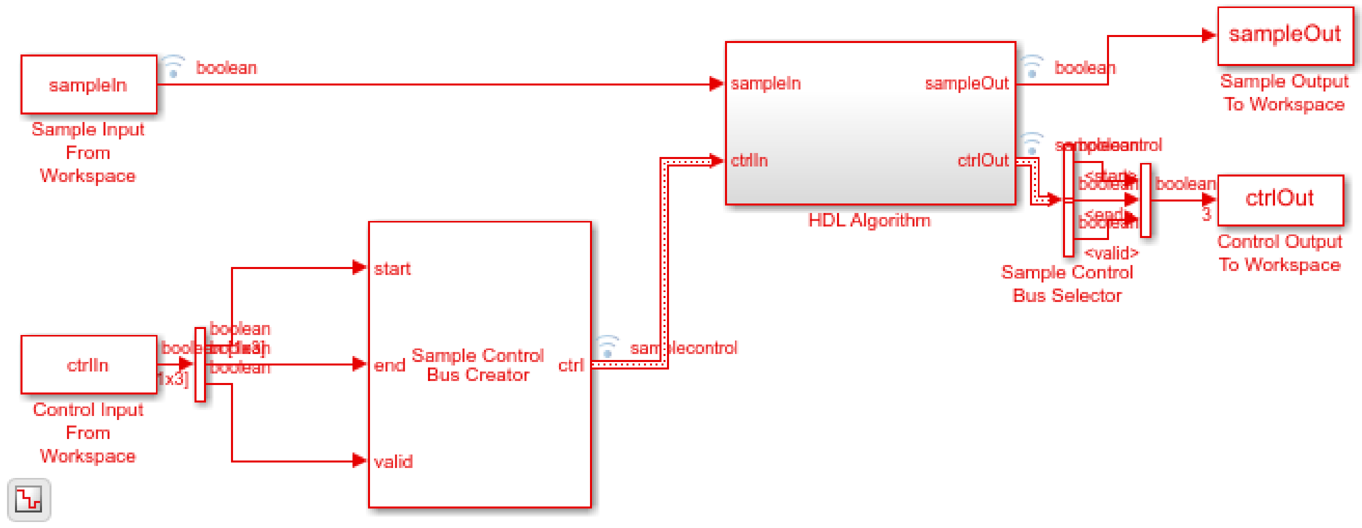
```
frameLength = 256;
numframes   = 2;
rng(0);

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`.

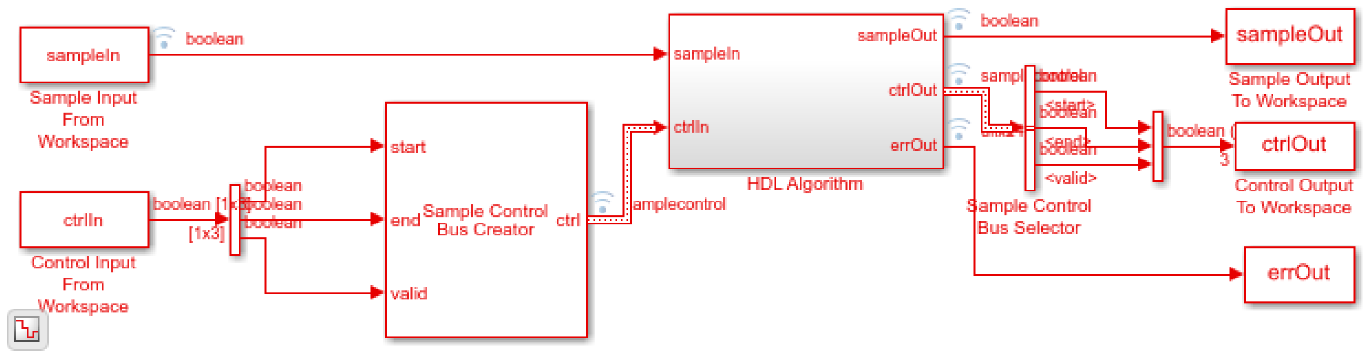
```
frameLength = 256;
numframes   = 2;
rng(0);

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`.

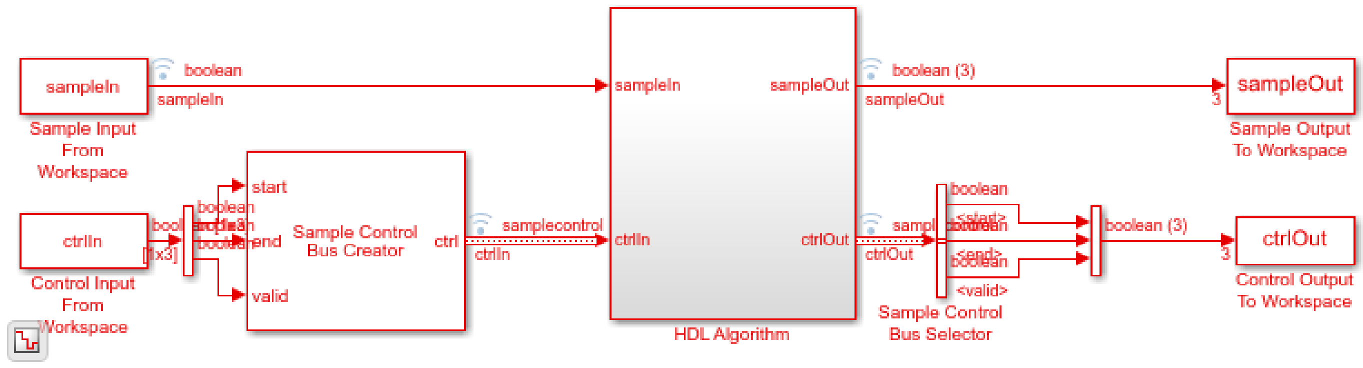
```
rng(0);
turboframesize = 40;
numframes = 2;

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` + 17 cycles to complete encoding of a frame.

```
inframes = txBits;

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

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = inframesize+20;

[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 * `numTurboIterations` * *HalfIterationLatency* + (`inframesize` / `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`

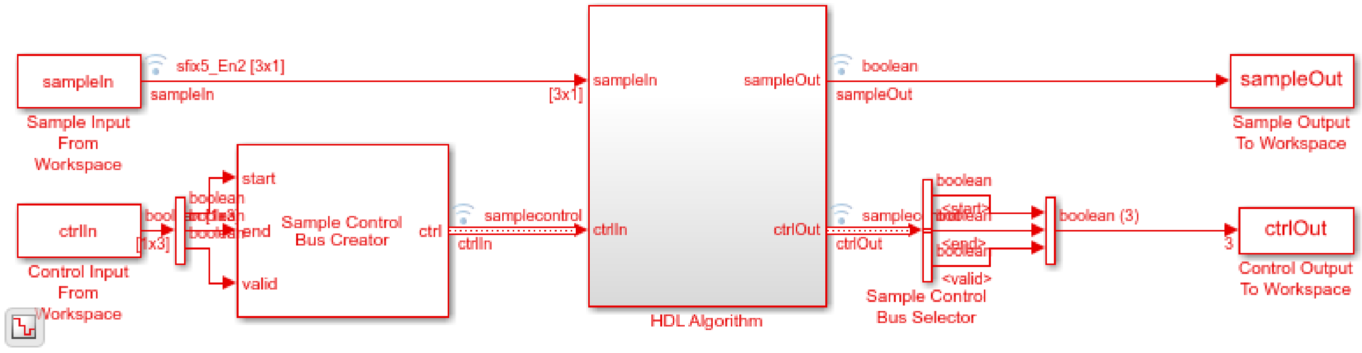
Reorder the samples using the interleave option of the `whdlFramesToSamples` function.

```
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`.

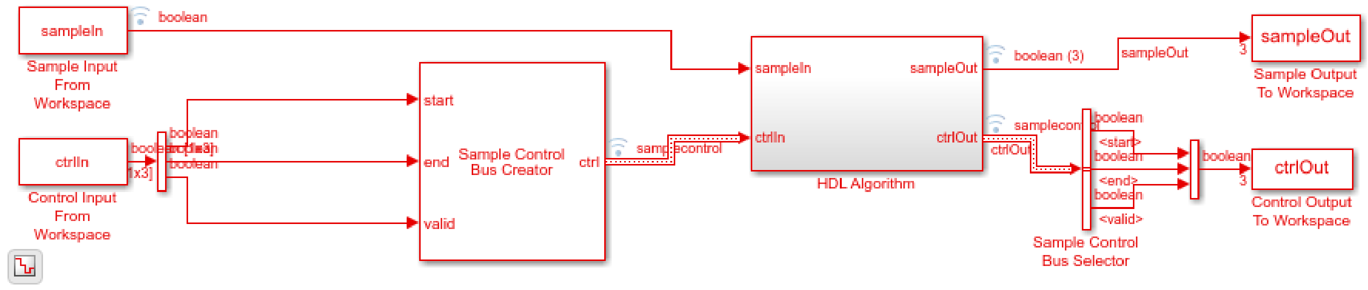
```
rng(0);
frameLength = 256;
numframes   = 2;

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: `G0_1 G1_1 G2_1 G0_2 G1_2 G2_2 ... Gn G1_n G2_n`
- LTE Toolbox output: `G0_1 G0_2 ... G0_n G1_1 G1_2 ... G1_n G2_1 G2_2 ... G2_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. `EbNo` is the ratio of energy per uncoded bit to noise spectral density, in dB. `EcNo` 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, `No`.

```
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.

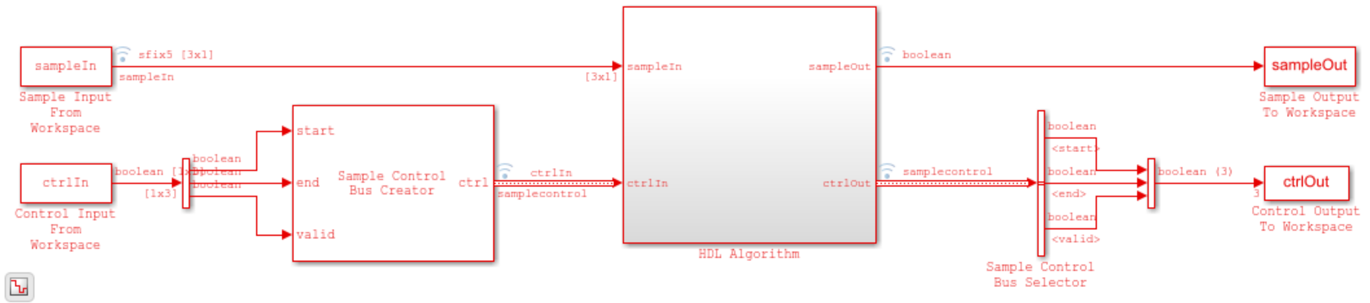
The LTE Convolutional Decoder block expects the input data to contain the three encoded bits interleaved.

- 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

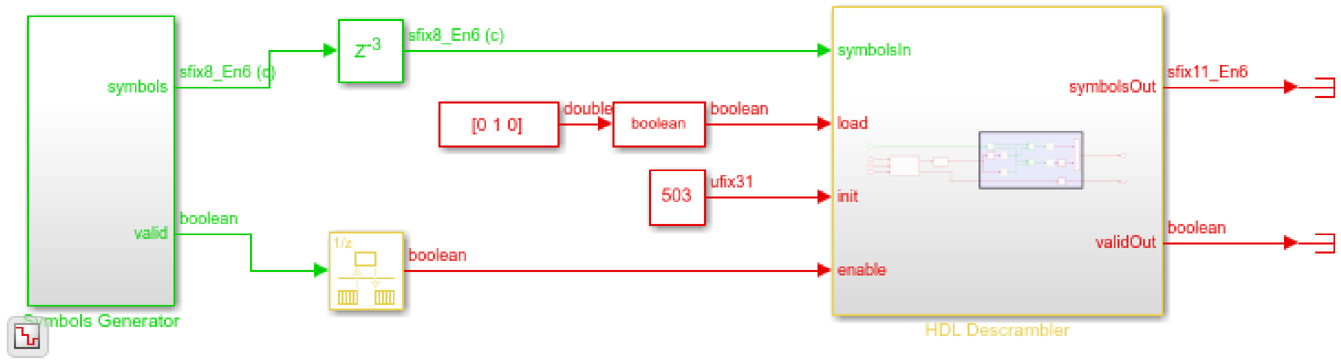
- "Convolutional Encode of Streaming Samples" on page 3-11

Descrambling with Gold Sequence Generator

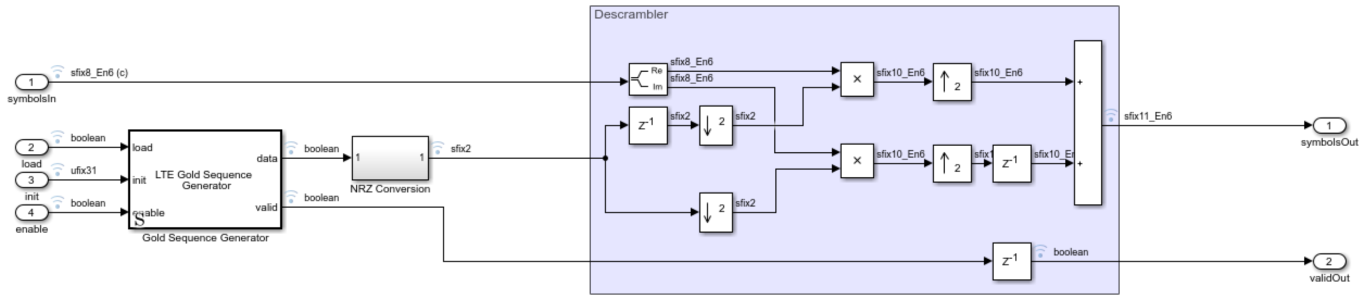
This example shows how to use the LTE Gold Sequence Generator block to implement an LTE descrambler.

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.

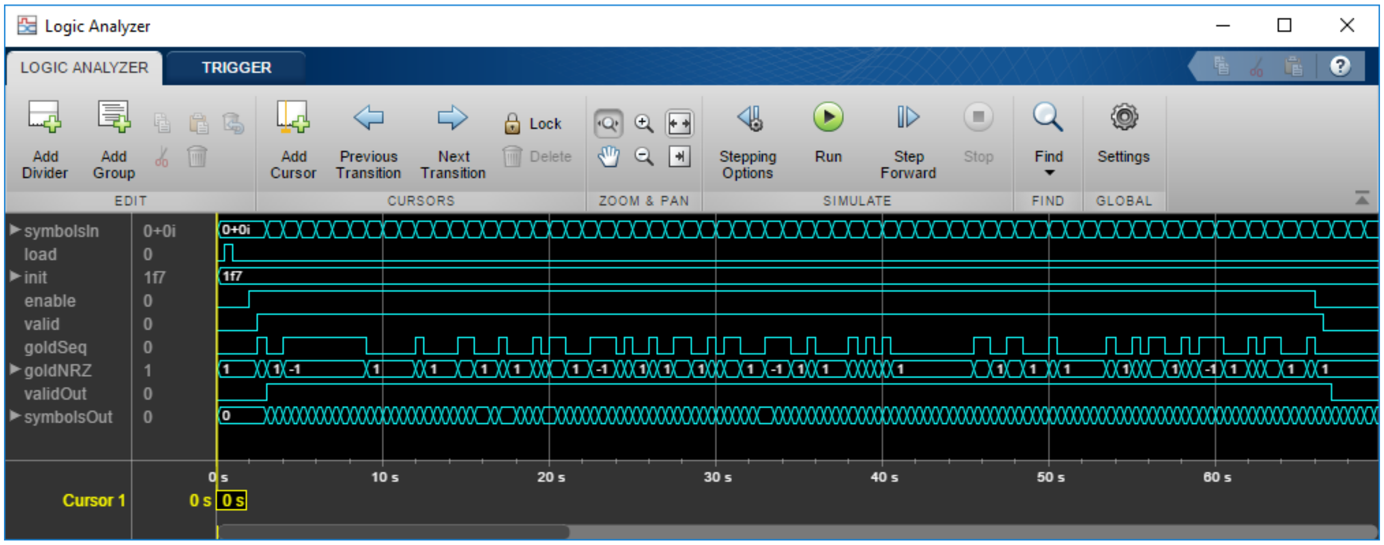
You can generate HDL from the HDL Descrambler subsystem.



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

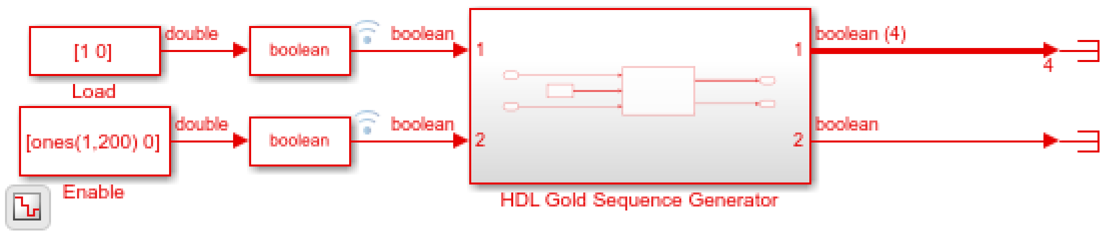
Blocks
LTE Gold Sequence Generator

Parallel Gold Sequence Generation

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

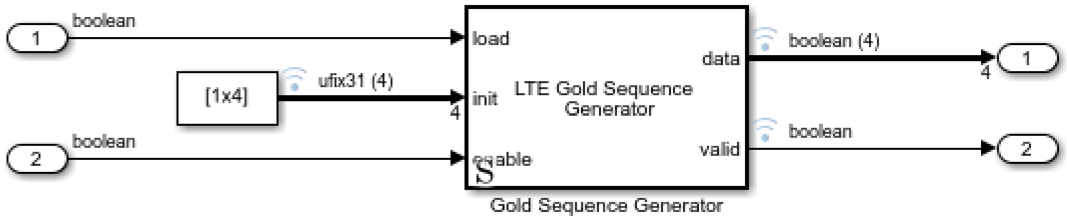
The example model initializes the Gold Sequence Generator block with a vector that represents the **init** values for each of four channels. The block returns four independent Gold sequences.

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

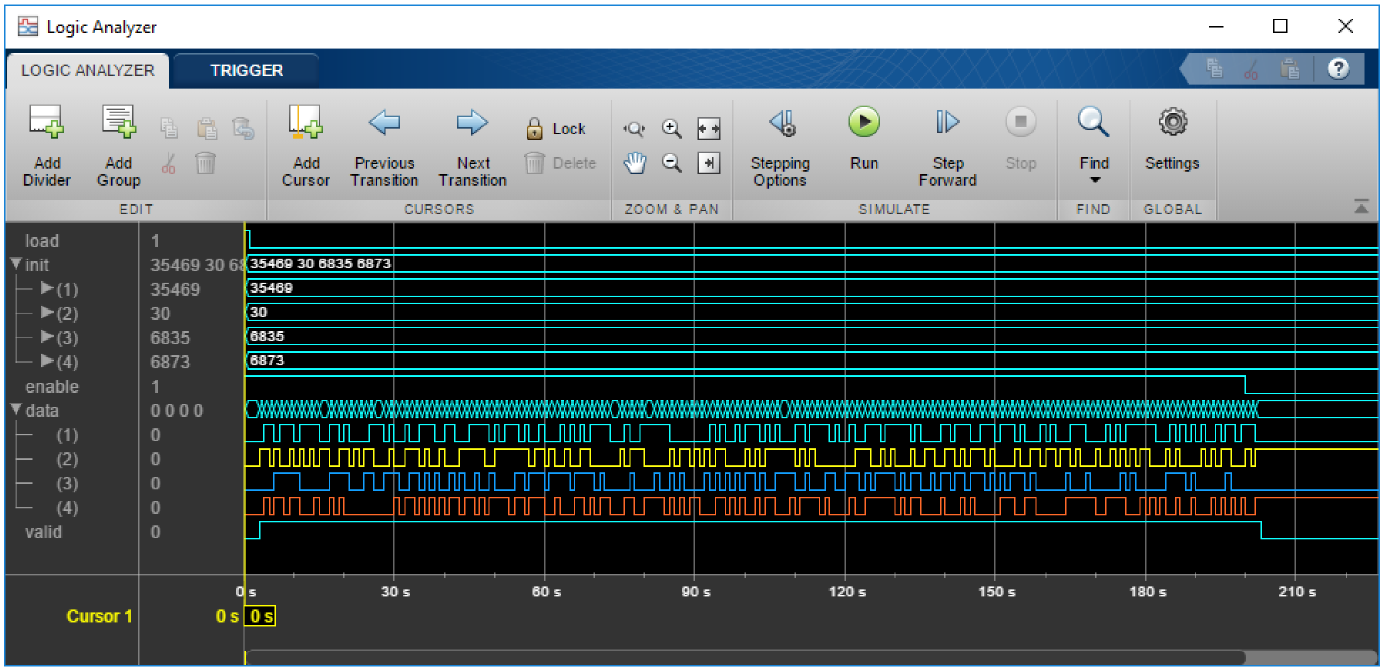


The Gold Sequence Generator block has no block parameters. It is configured to match the polynomial and shift length required by LTE standard TS 36.212. You must initialize the sequence with a 31-bit value on the **init** port, and load the value into the block by setting the **load** signal to 1 for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

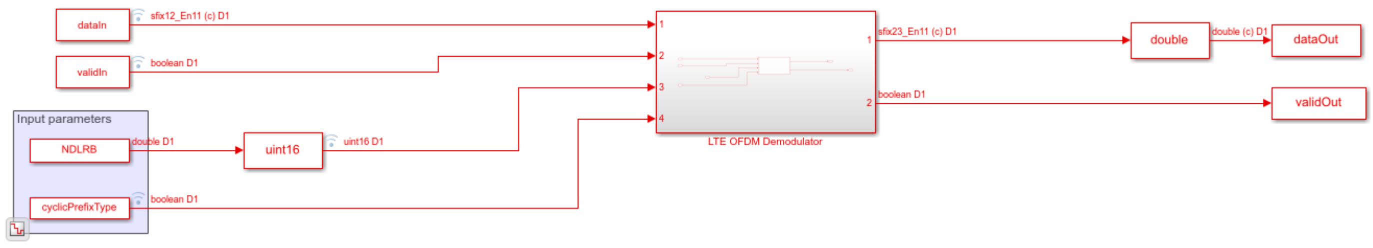
```
NDLRB = info.NDLRB;
if strcmp(info.CyclicPrefix,'Normal')
    cyclicPrefixType = false;
else
    cyclicPrefixType = true;
end

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; % NDRLB 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)
```

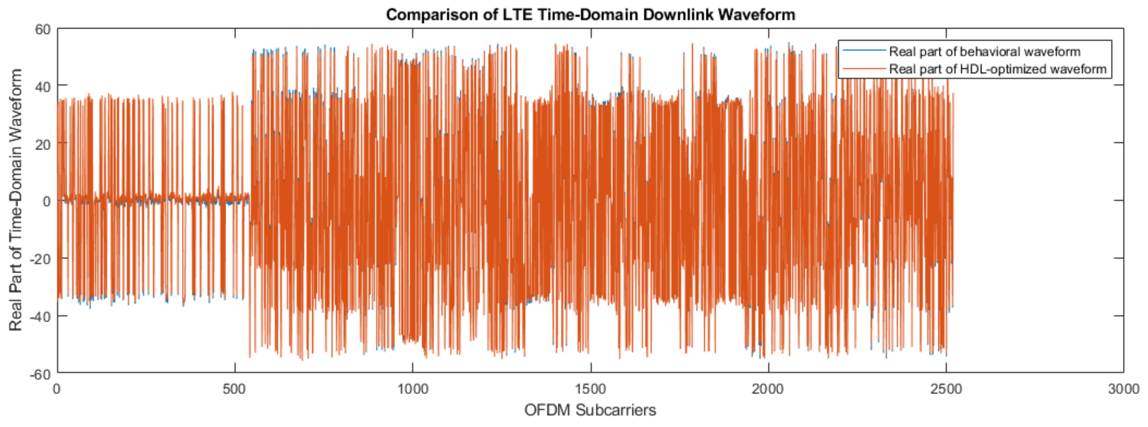


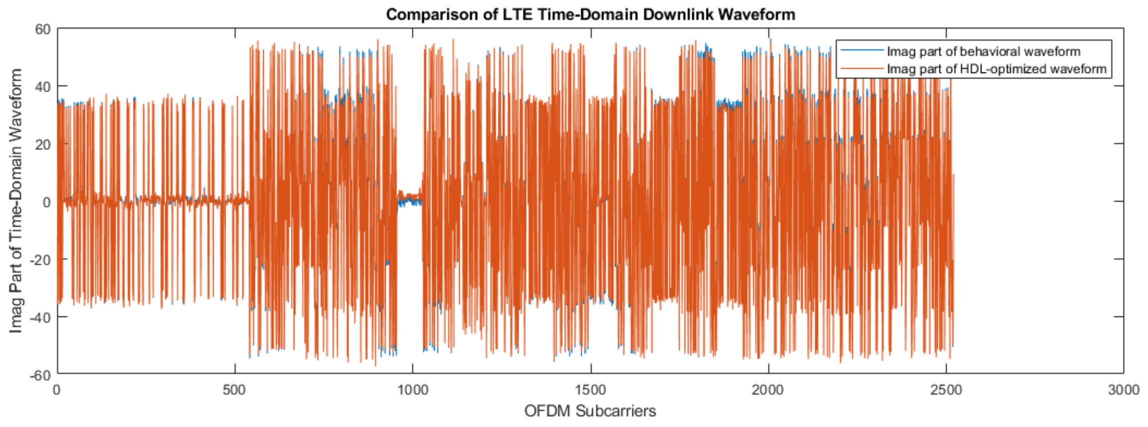
Compare the output of the Simulink model against the behavioral results, and calculate the SQNR of the HDL-optimized LTE OFDM Demodulator block.

```
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')

sqnrRealdB = 10*log10(var(real(rxgridSimulink))/abs(var(real(rxgridSimulink))-var(real(refGrid(:
sqnrImagdB = 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',sqnrRealdB)
fprintf('\n SQNR of imaginary part is %.2f dB\n',sqnrImagdB)
```

```
 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`.

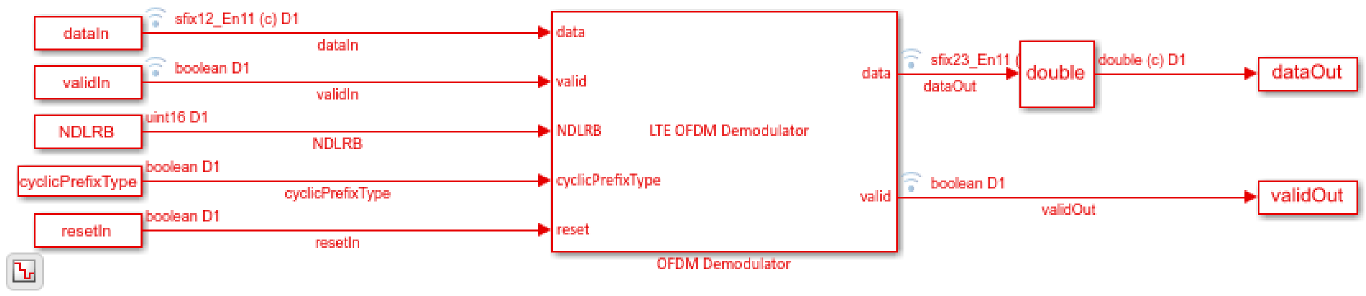
```
dataIn = fi(dataIn_fp,1,12,11);

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; % NDRLB 100

sampling_time = 1/FsRx;
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 = '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);
```

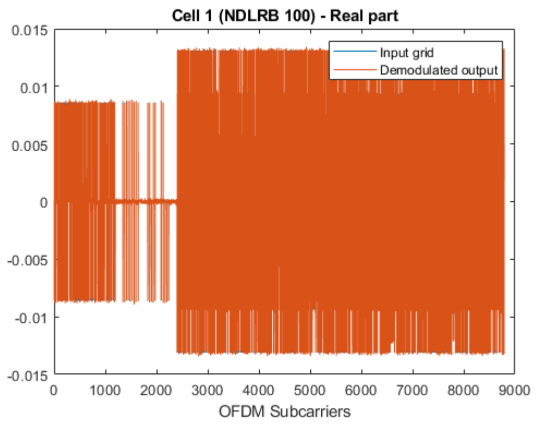
Generate reference data by flattening and normalizing the unmodulated resource grid data. Truncate the first cell in the same way as the modulated input data. Apply complex scaling to each demodulated sequence so that it can be compared to its corresponding reference data.

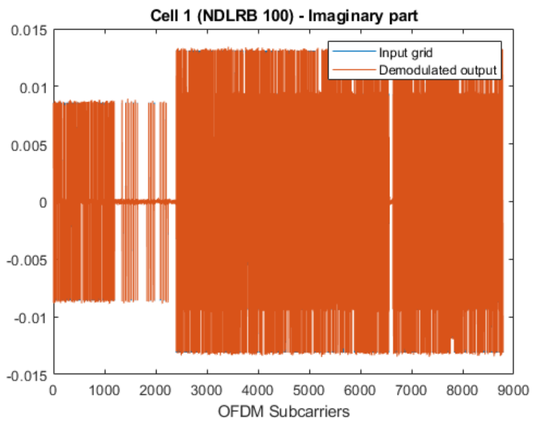
```
refData1 = grid1(:);
refData1 = refData1(1:length(demodData1));
refData2 = grid2(:);

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

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

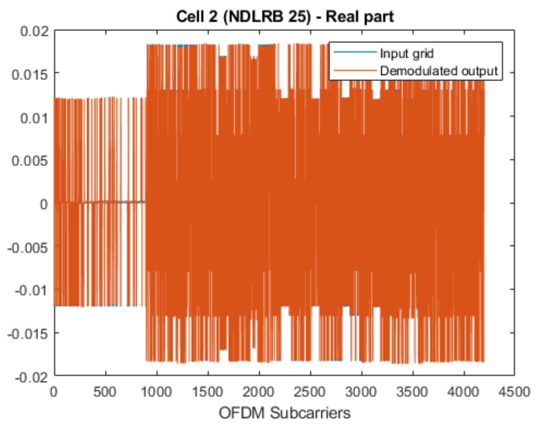
Compare the output of the Simulink model against the truncated input grid, and display the results.

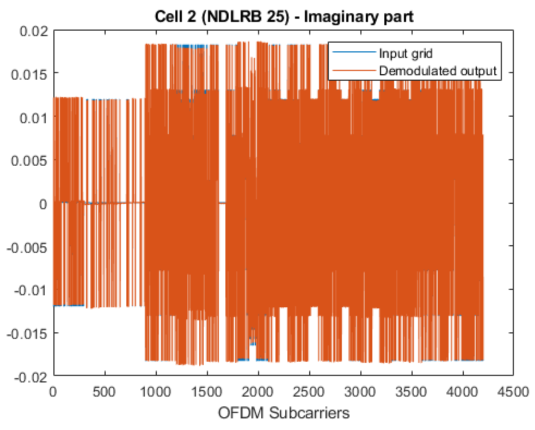
```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,2,1)
plot(real(refData1(:)))
hold on
plot(squeeze(real(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDLRB %d) - Real part', info1.NDLRB))
xlabel('OFDM Subcarriers')

subplot(2,2,2)
plot(imag(refData1(:)))
hold on
plot(squeeze(imag(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDLRB %d) - Imaginary part', info1.NDLRB))
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 (NDLRB %d) - Real part', info2.NDLRB))
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 (NDLRB %d) - Imaginary part', info2.NDLRB))
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™.

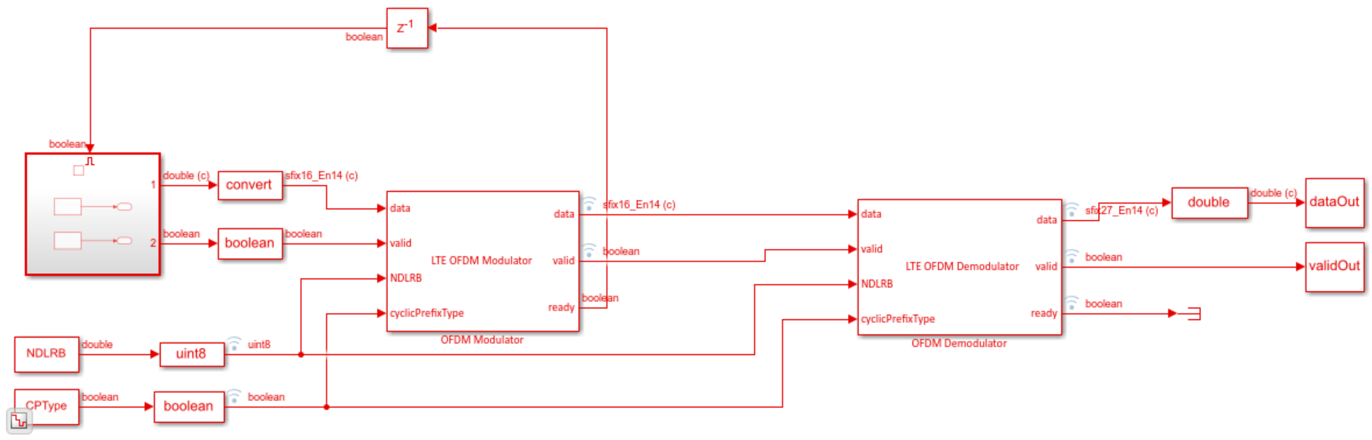
```
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;

% -------------------------------------------------------------
%      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));
```

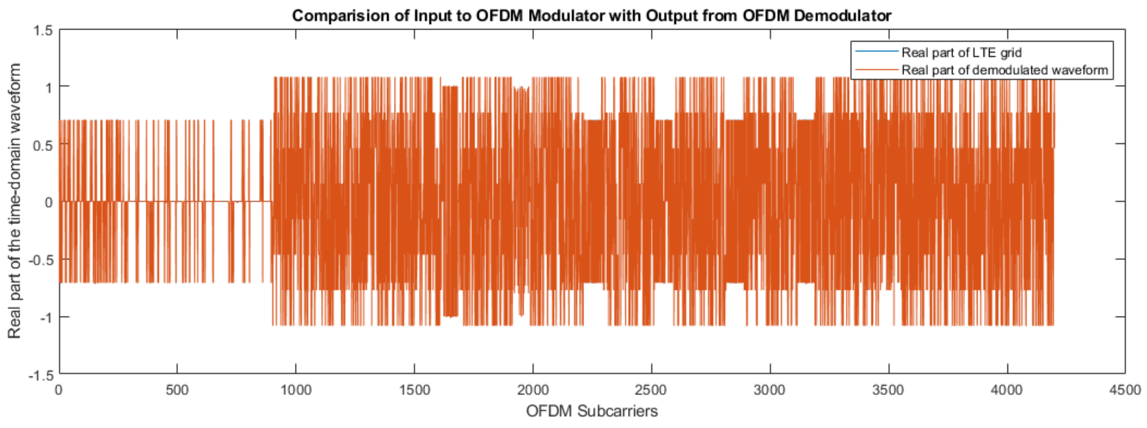
Run the Simulink model to modulate and demodulate the samples, and save the output samples to a workspace variable.

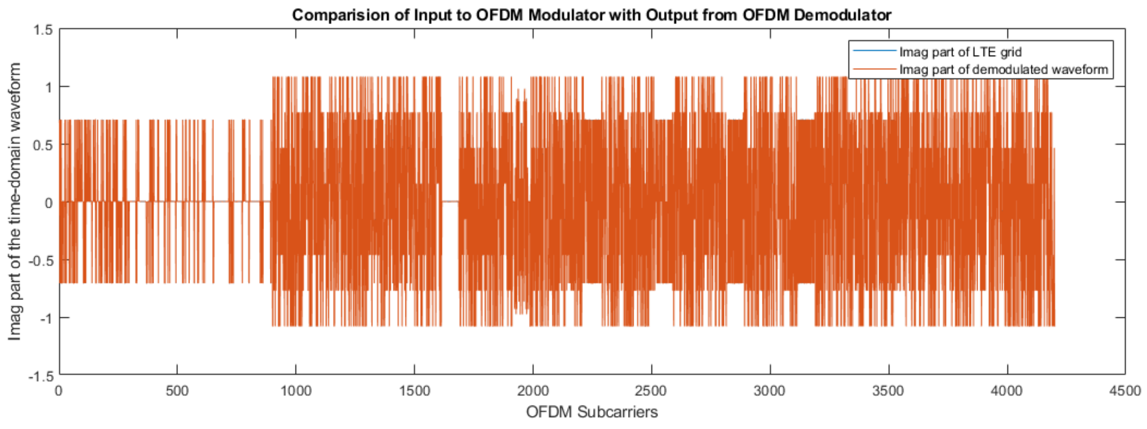
```
open_system('LTEHDLOFDMModDemodExample')
sim('LTEHDLOFDMModDemodExample');

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('Comparision 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('Comparision 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™.

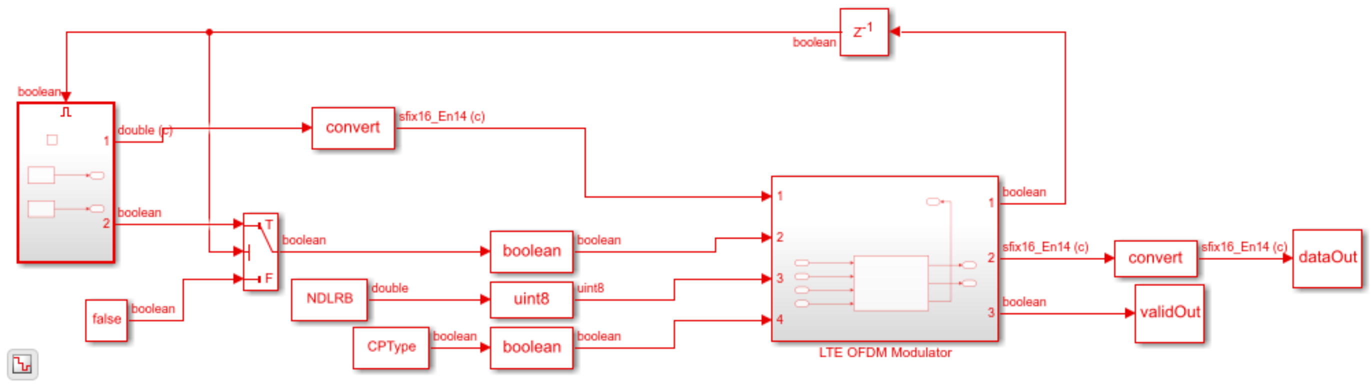
```
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;
% ---------------------------------------------------------------
%      NDLRB              |   Sampling Rate (MHz)
% ---------------------------------------------------------------
%       6                  |   R.4
%       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));
```

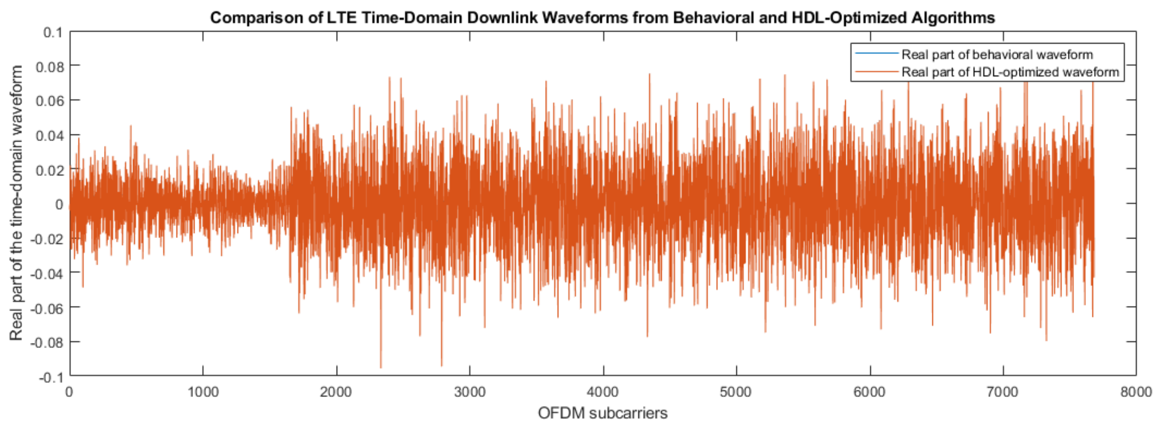
Run the Simulink model.

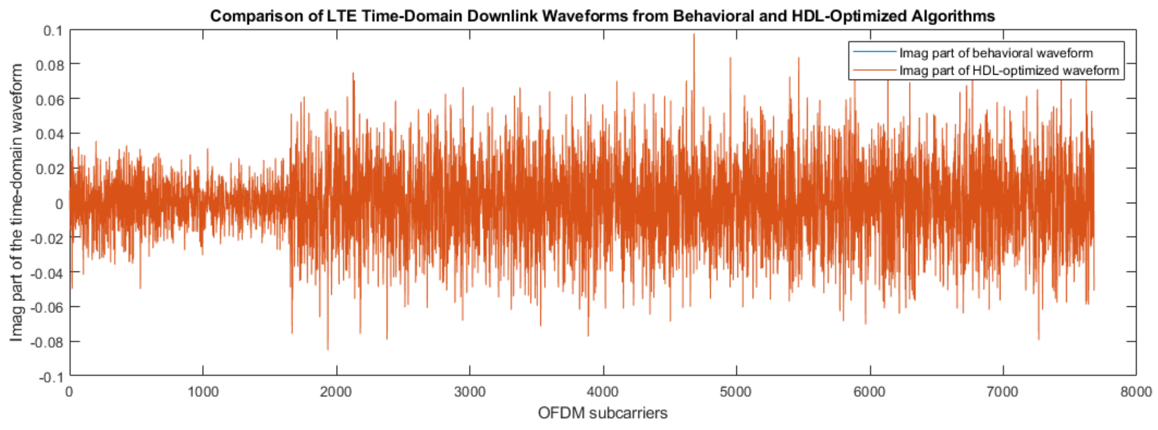
```
modelname = 'OFDMModulatorModelExample';
open_system(modelname);
sim(modelname);
```



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 Algori
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 Algori
xlabel('OFDM subcarriers');
ylabel('Imag part of the time-domain waveform');
```





See Also

Blocks

OFDM Modulator

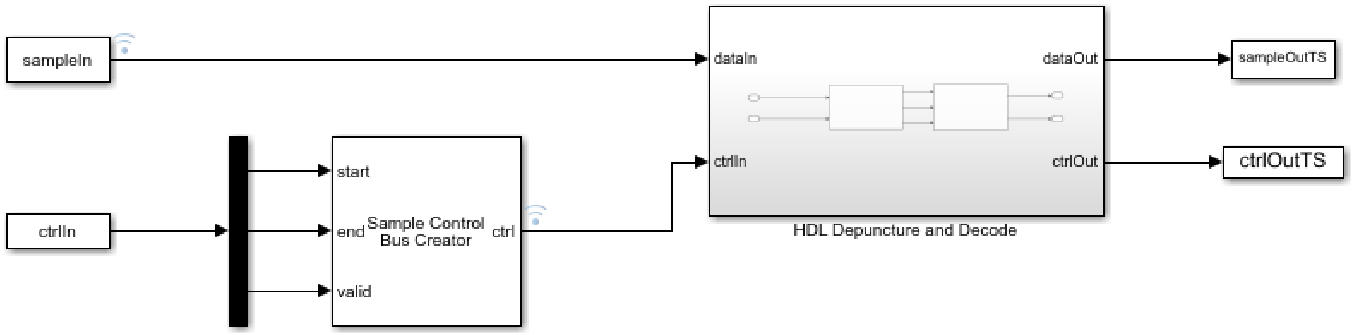
Depuncture and Decode Streaming Samples

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);
    % 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;
```

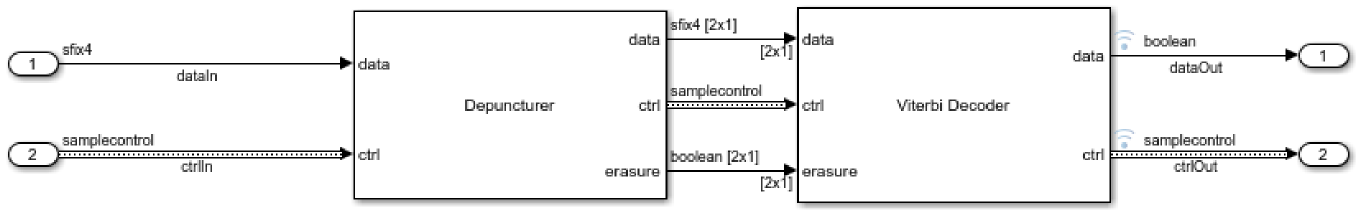
Run the Simulink Model

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.

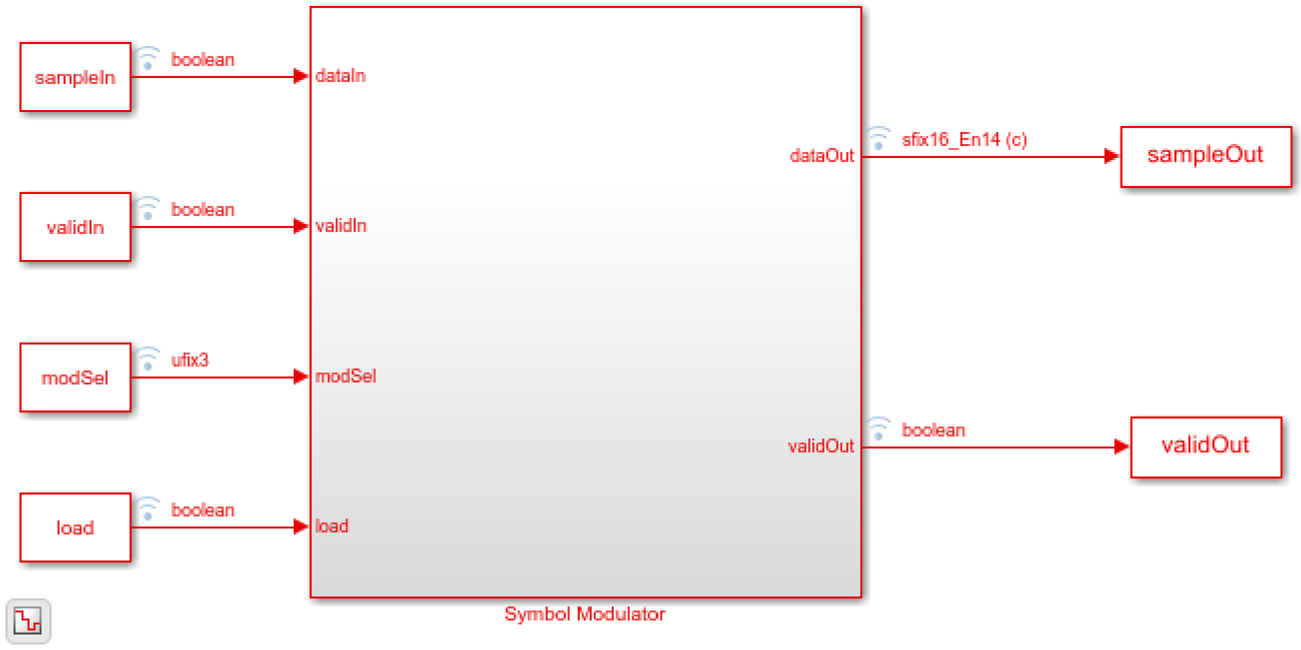
```
for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);

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 outp
```

```
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.

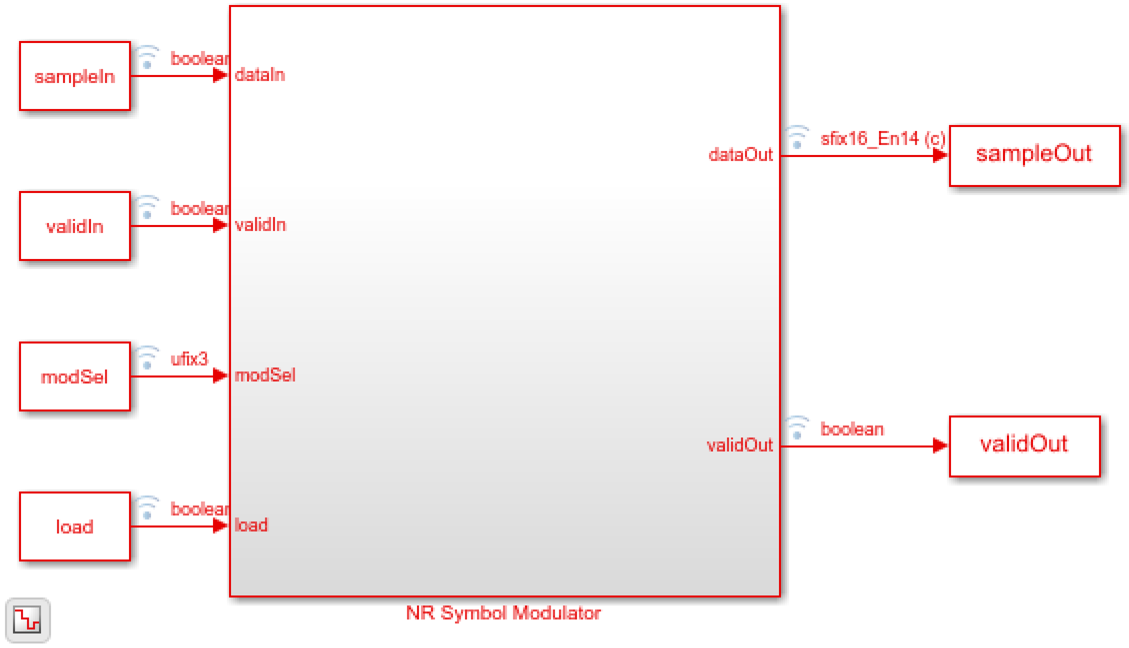
```
for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);

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 `lteSymbolDemodulator` 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
```

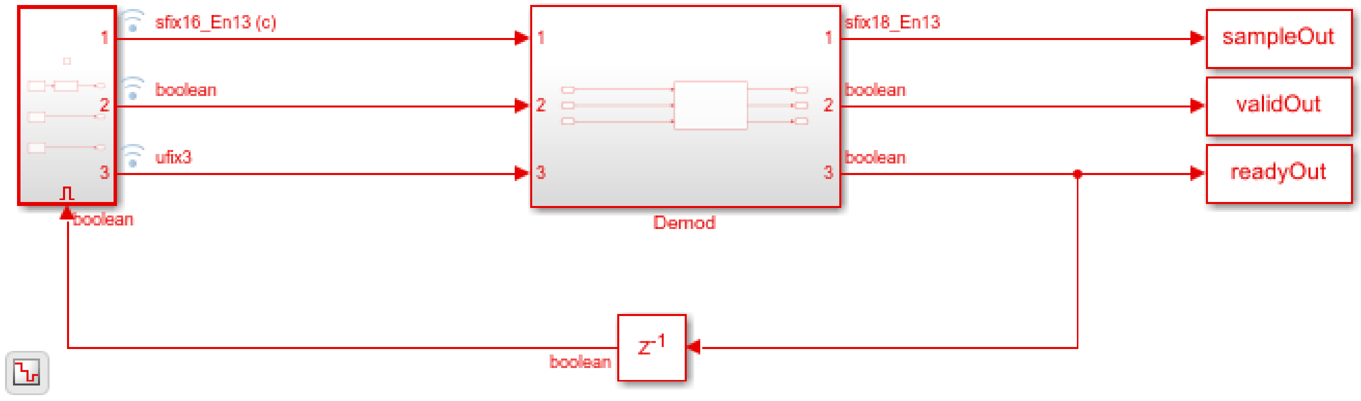
Convert the framed input data to a stream of samples and input the stream to the LTE Symbol Demodulator Simulink block.

```
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.

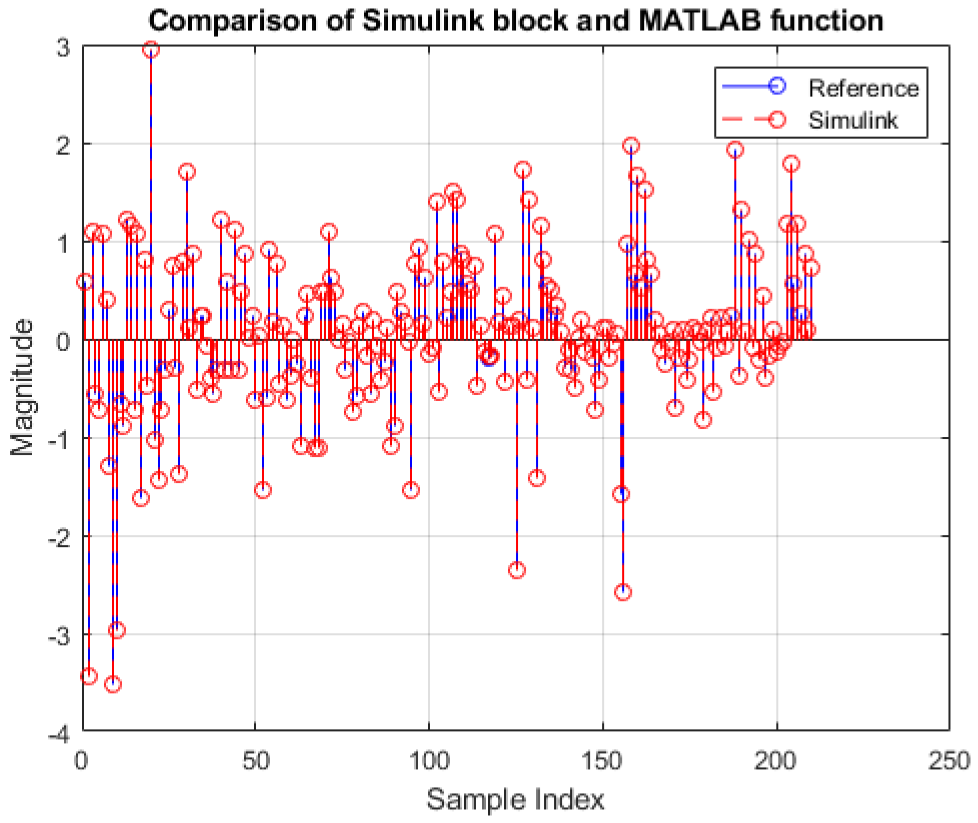
```
lteHDLOutput = sampleOut(validOut).';
```

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 `nrSymbolDemodulator` 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.

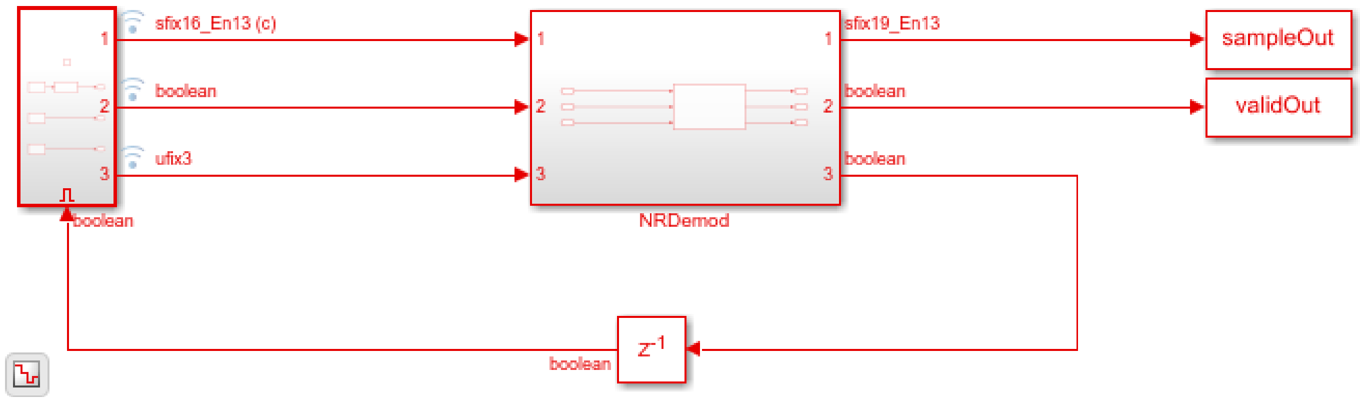
```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

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.

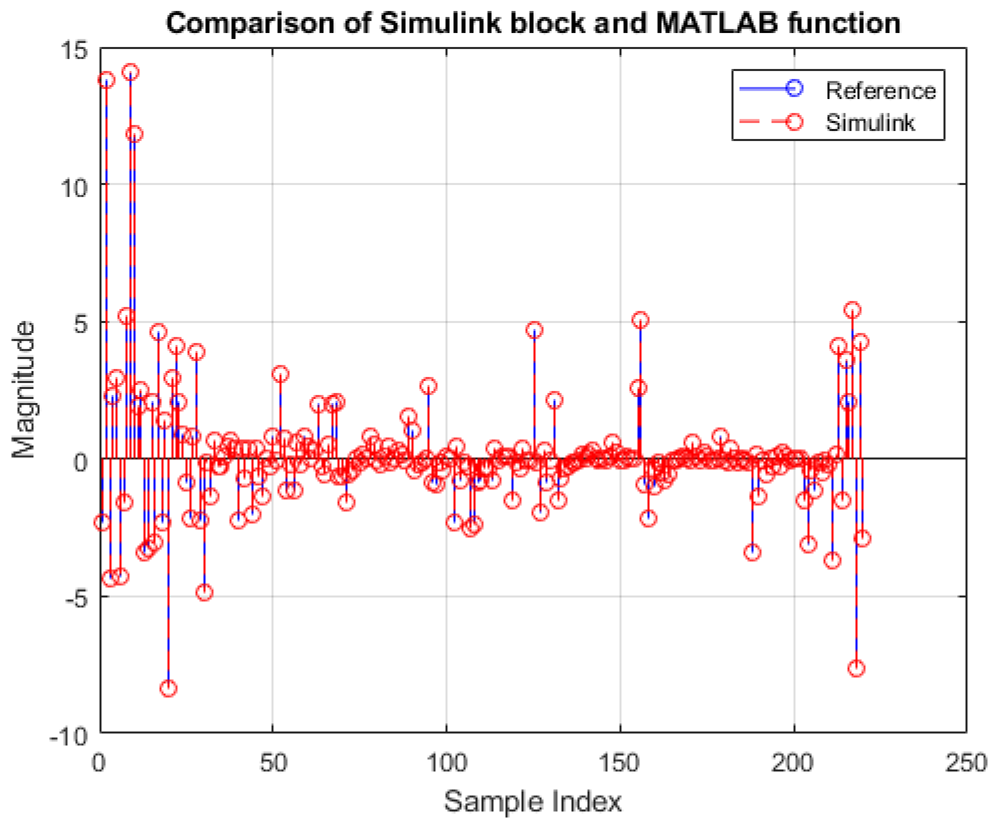
```
nrHDLOutput = sampleOut(validOut).';
```

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 $\frac{3}{4} = \frac{23.04e6}{30.72e6}$

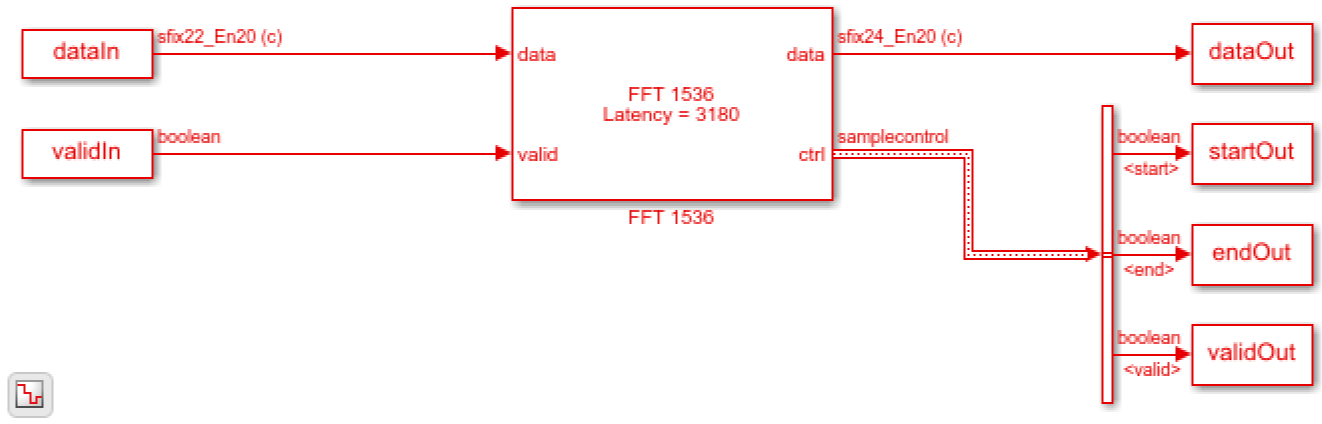
```
rx = resample(tx,3,4); % rate conversion from 30.72Msps to 23.04Msps
```

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;
```



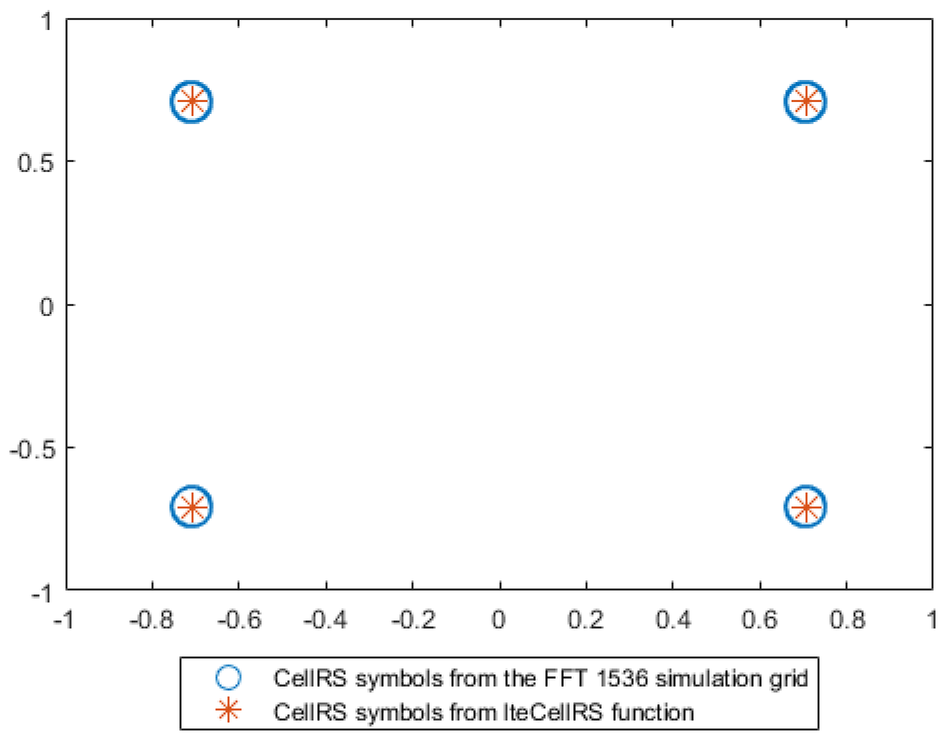
```
simOut = dataOut(validOut);
simOut = double(simOut(:)*1536);
```

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

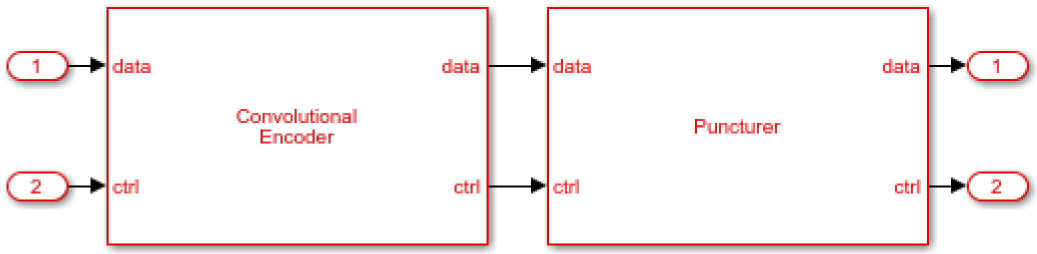
Convolutional Encode and Puncture Streaming Samples

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

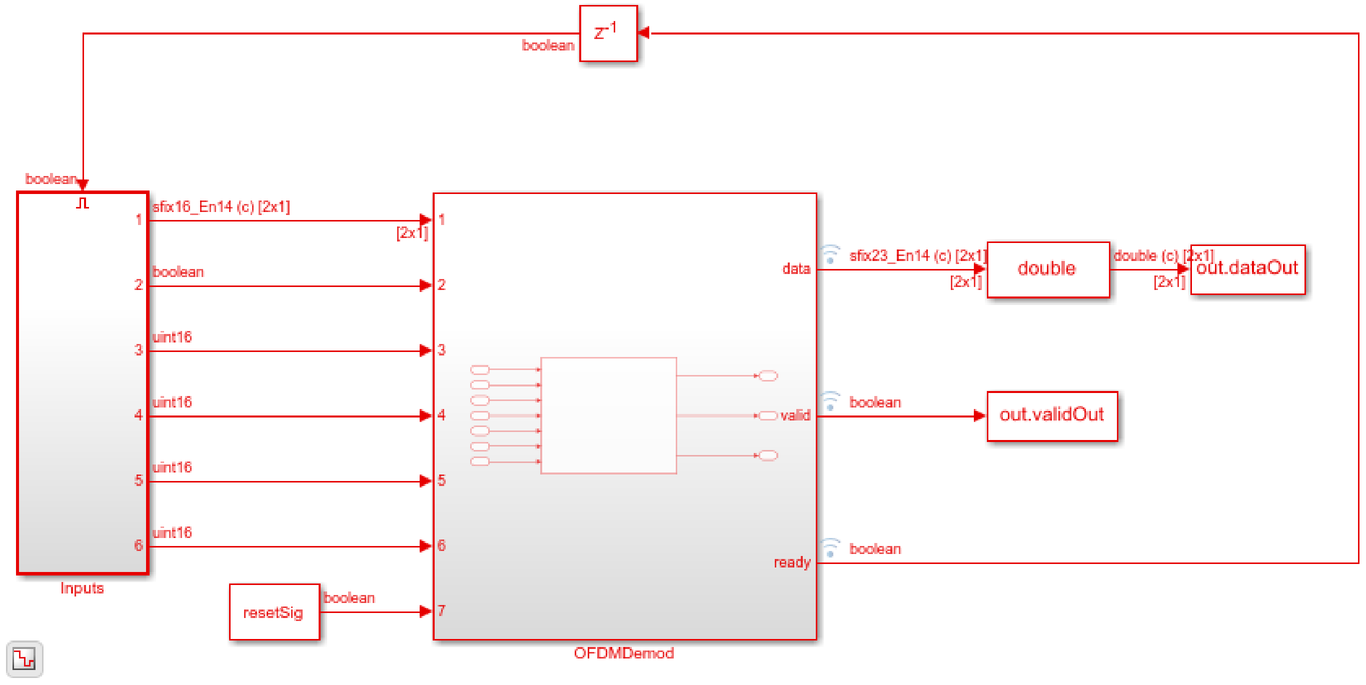
```
data = randn(fftLen,numOFDMSym)+1i*randn(fftLen,numOFDMSym);
dataIn = ofdmmod(data,fftLen,cpLen);
```

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 `ofdmdemod_baseline` function

```
[dataOut1] = ofdmdemod_baseline(dataIn,fftLen,cpLen,symbOffset,NullInd.',[],Normalize,RoundingMe
matOut = dataOut1(:);
```

Compare the output of the Simulink model against the output of `ofdmdemod_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(:);
```

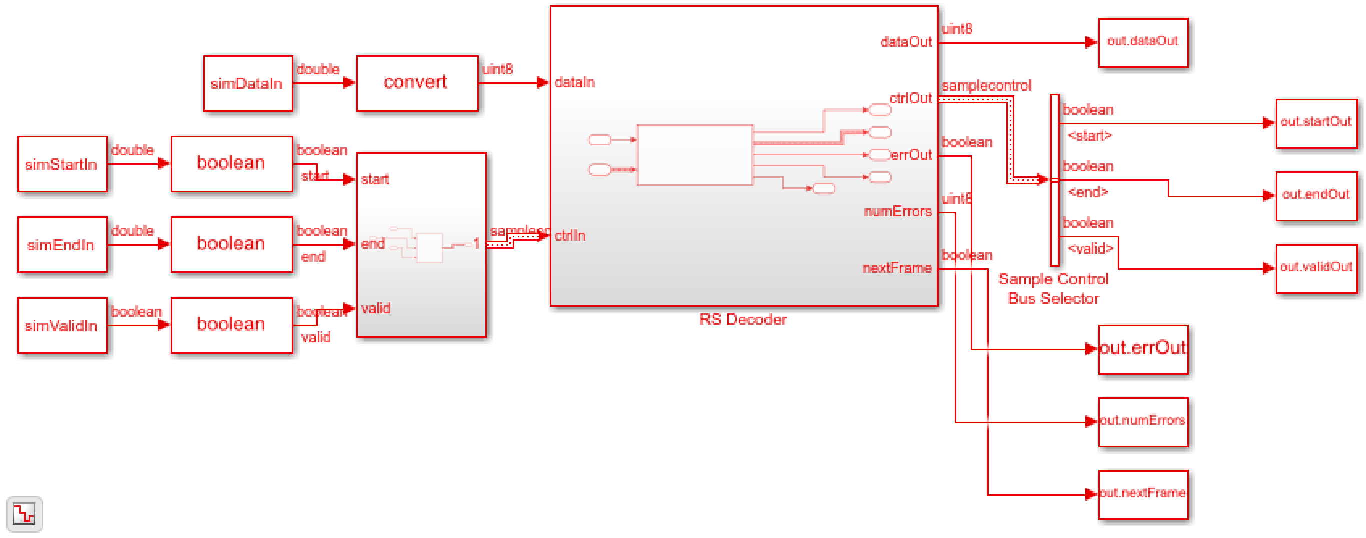
Import the encoded random input samples to the Simulink® model

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 ou
```

```
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 Encoder Decoder blocks. In this example, the hardware-optimized results of these blocks are compared with respective functions in the 5G Toolbox™.

Generate Input Data for Encoder

Choose a series of input values for bgn and liftingSize. These values must be supported by the 5G NR standard. Generate the corresponding input vectors of bgn and liftingSize values over time. Generate random frames of input data and convert them to boolean vectors and control signals that indicate the frame boundaries. The example model imports the workspace variables `encSampleIn`, `encStartIn`, `encEndIn`, `encValidIn`, `encbgnIn`, `encliftingSizeIn`, `sampleTime`, and `simTime`.

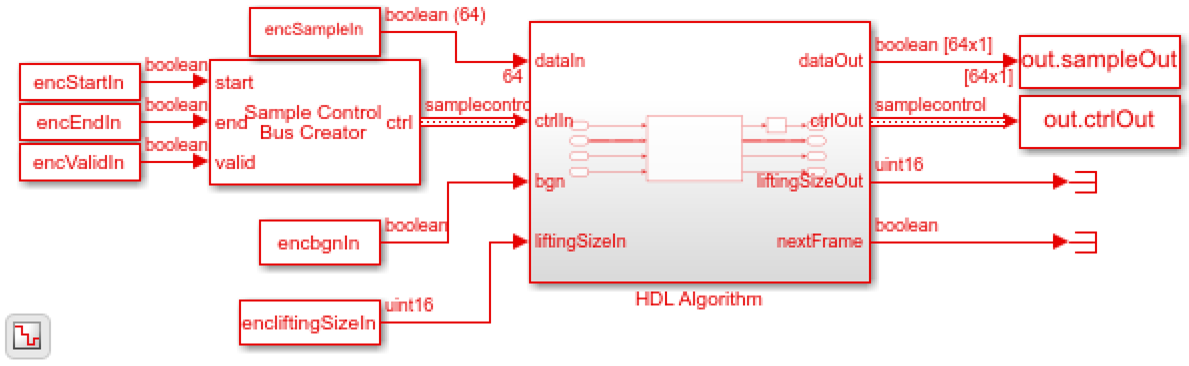
`encFrameGap` accomodates the latency of the NR LDPC Encoder block for bgn and liftingSize values. Use the **nextFrame** signal to determine when the block is ready to accept the start of the next input frame.

```
bgn         = [0; 1; 1; 0];
liftingSize = [4; 384; 144; 208];
numFrames = 4;

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);
    len = K(ii) * ceil(liftingSize(ii)/64);

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

    encSampleIn   = logical([encSampleIn encIn zeros(64,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,encFrameGa
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 from the workspace and returns LDPC encoded output vectors along with the control signals that indicate the frame boundaries. The model exports `sampleOut` and `ctrlOut` to the MATLAB workspace.

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



Verify Encoder Results

Convert the streaming data back to frames for comparision with the results of the `nrLDPCEncode` function from the 5G Toolbox™.

```
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(i
    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 to generate input log-likelihood ratios (LLR) for the NR LDPC Decoder block. Use channel, modulator, and demodulator objects to add some noise to the signal.

Again, create vectors of bgn and liftingSize and convert the frames of data to input LLR vectors with control signals. The example model imports the workspace variables `decSampleIn`, `decStartIn`, `decEndIn`, `decValidIn`, `decbgnIn`, `decliftingSizeIn`, `numIter`, `sampleTime`, and `simTime`.

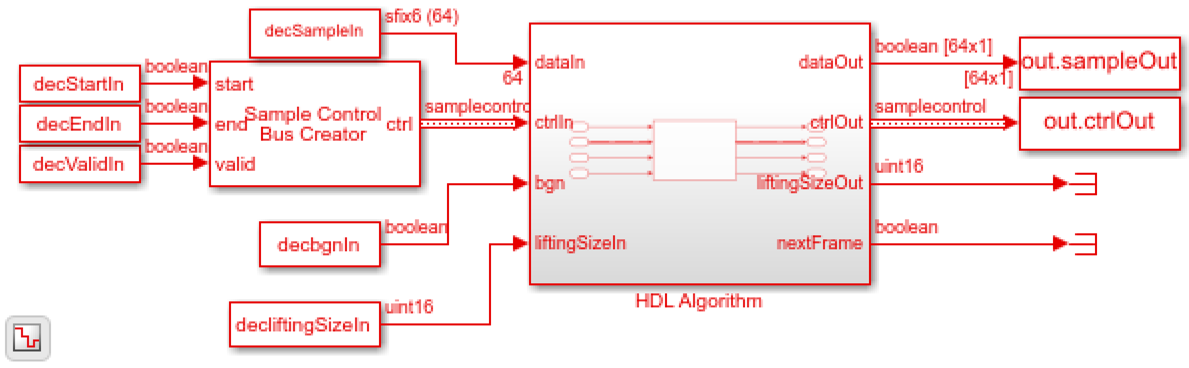
`decFrameGap` accomodates the latency of the NR LDPC Decoder block for bgn, liftingSize, and number of iterations. Use the **nextFrame** signal to determine when the block is ready to accept the start of the next input frame.

```
nVar = 2.5;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);

numIter = 8;
decFrameGap = numIter *1200;
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);

    len = N(ii)* ceil(liftingSize(ii)/64);

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

    decSampleIn   = [decSampleIn decIn zeros(64,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,decFrameGa
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 from the workspace and returns a stream of decoded output samples along with control signals that indicate the frame boundaries. The model exports `sampleOut` and `ctrlOut` to the MATLAB workspace.

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



Verify Decoder Results

Convert the streaming data returned from the Simulink model into frames for comparision with the results of the `nrLDPCDecode` function from the 5G Toolbox™.

```
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(i
    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 | `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 and reference subcarriers. In this example model, averaging and interpolation features are enabled. The example model supports HDL code generation for the HDL Algorithm subsystem.

Set up input data parameters

```
rng('default');
numOFDMSym = 28;
numOFDMSymToBeAvg = 14;
interpolFac = 3;
maxNumScPerSym = 72;
numScPerSym = 72;
```

Generate data subcarriers of random input samples

Using `numScPerSym` and `numOFDMSym` values, generate the data subcarriers of random input samples.

```
dataIn = rand(1,numOFDMSym*numScPerSym) + 1i*rand(1,numOFDMSym*numScPerSym);
validIn = true(1,length(dataIn));
```

Generate reference data subcarriers of random input samples

Using `numScPerSym` and `numOFDMSym` values, generate the reference data subcarriers of random input samples.

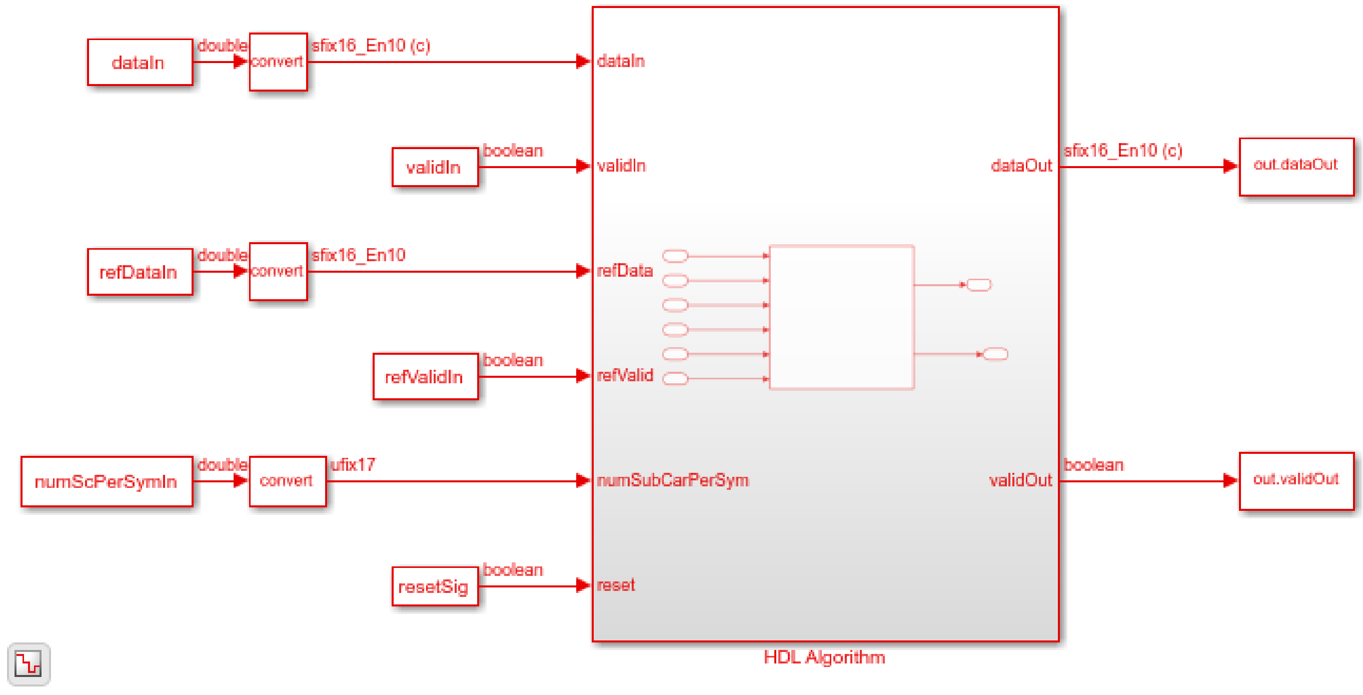
```
refDataIn = randsrc(size(dataIn,1),size(dataIn,2), [-1 1]);
refValidIn = boolean(zeros(1,numOFDMSym*numScPerSym));
startRefValidIndex = randi(interpolFac,1,1);
for numOFDMSymCount = 1:numOFDMSym
    refValidIn(startRefValidIndex+(numOFDMSymCount-1)*numScPerSym:interpolFac:numScPerSym*numOFDM
end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

Generate the number of subcarriers per symbol samples

```
numScPerSymIn = numScPerSym*true(1,length(dataIn));
resetSig = false(1,length(dataIn));
```

Run the Simulink® model

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



Export the stream of channel estimates from Simulink to the MATLAB® workspace

```
simOut = out.dataOut.Data(out.validOut.Data);
```

Estimate the channel with the generated random input samples

You can use the output of the function `channelEstReference` as a reference data for comparison.

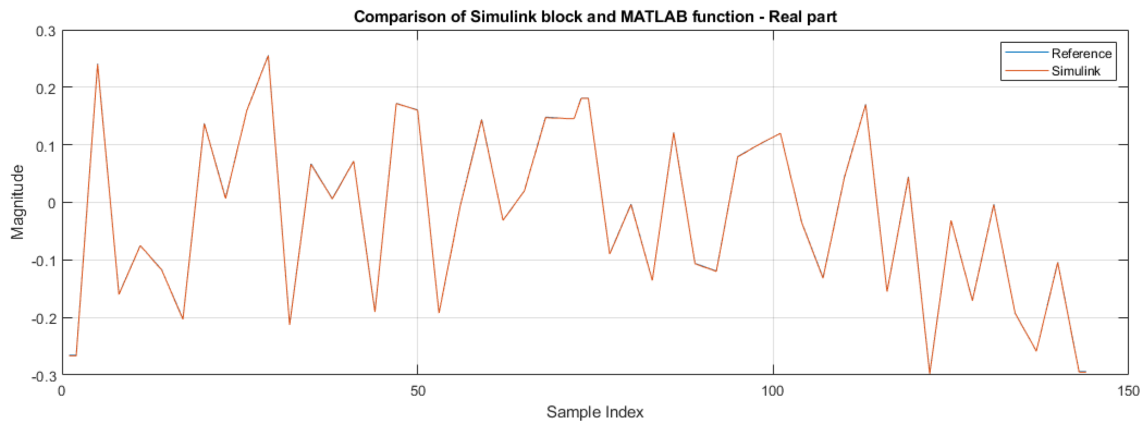
```
dataOut1 = channelEstReference(...
    numOFDMSymToBeAvg,interpolFac,numScPerSym,numOFDMSym,...
    dataIn,validIn,refDataIn,refValidIn,numScPerSymIn);

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
```

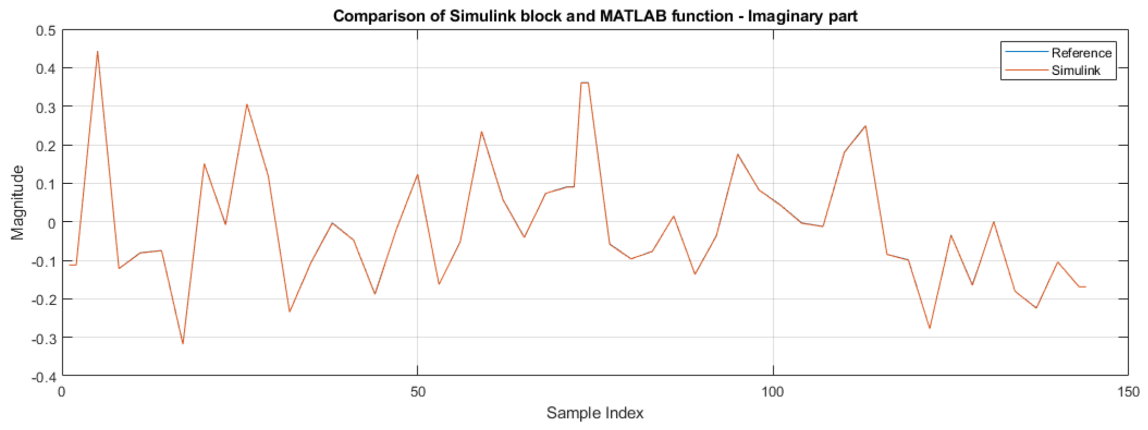
Compare Simulink block output data with the reference MATLAB function output

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

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

```
 OFDM Channel Estimator:
 SQNR of real part is 31.82 dB
 SQNR of imaginary part is 29.41 dB
```





See Also

Blocks

OFDM Channel Estimator

Modulate and Demodulate OFDM Streaming Samples

This example shows how to simulate the OFDM Modulator and Demodulator blocks. In this example model, the OFDM Modulator and OFDM Demodulator HDL subsystems are connected to eachother and the output of the OFDM Demodulator block is compared with the input of the OFDM Modulator block. You can generate HDL code for these blocks.

Set input data parameters

The example model imports the workspace variables `dataIn`, `validIn`, `fftLen`, `maxFFTLen`, `cpLen`, `numLG`, `numRG`, `numSymb`, and `DCNull`.

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

Generate input data frames

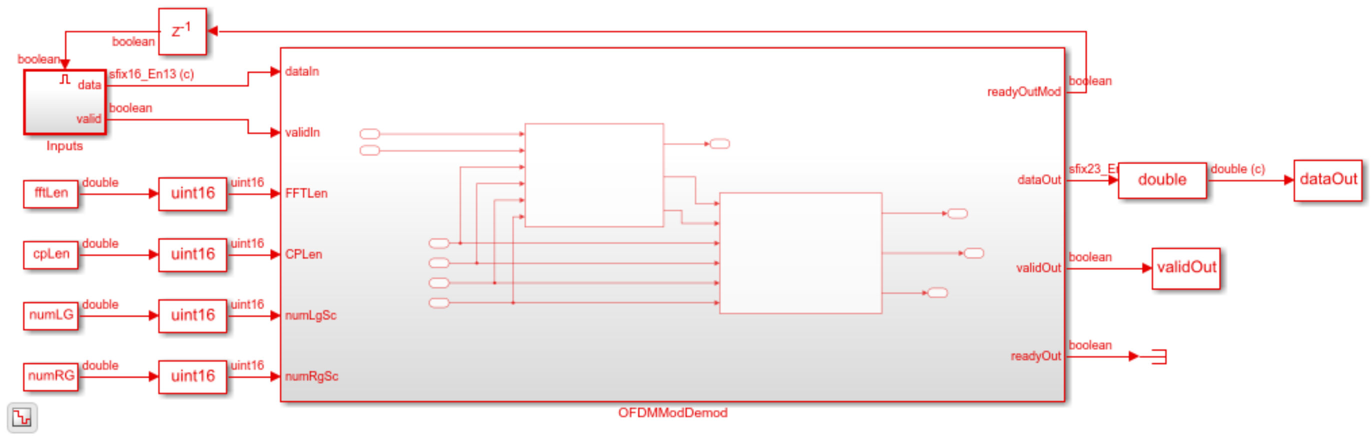
Generate random frames of complex input data and a control signal that indicates the frame boundaries.

```
rng default;
dataIn = complex(randn(numActData,numSymb),randn(numActData,numSymb));
data = dataIn(:);
valid = true(1,length(data));

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

Run the Simulink model

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

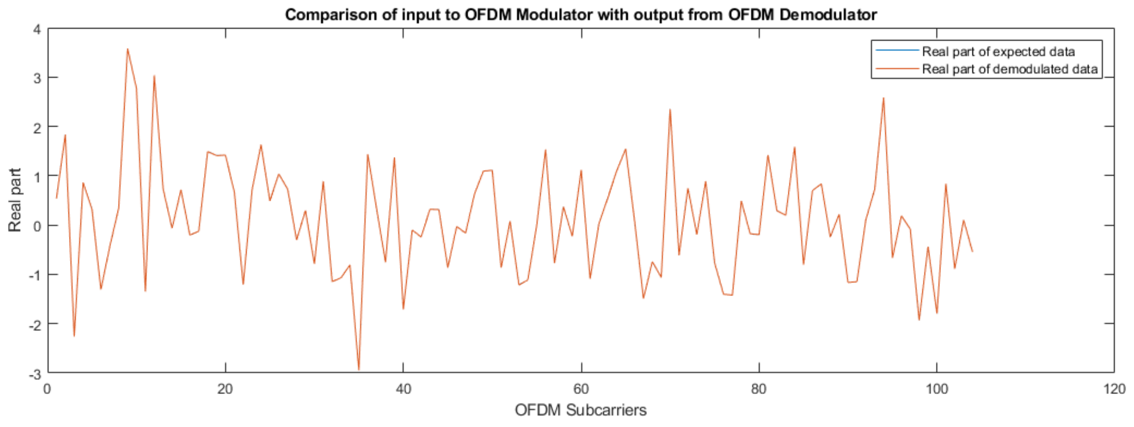
```
open_system('genhdlOFDMModDemodExample')
sim('genhdlOFDMModDemodExample');

simOut = dataOut(validOut);
```

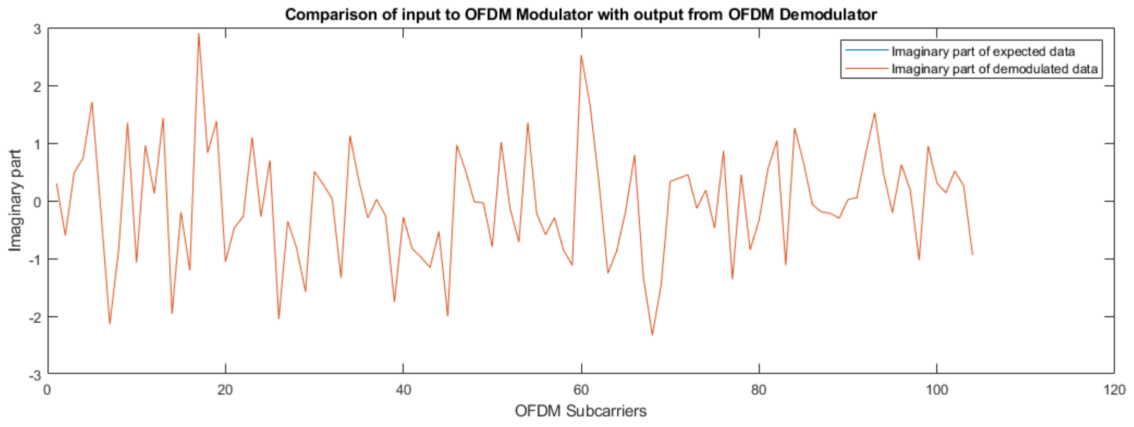


Compare OFDM Modulator and OFDM Demodulator blocks

The inputs provided to the OFDM Modulator block are compared with the outputs generated from the OFDM Demodulator block using `plot` function.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(data(:)));
hold on
plot(squeeze(real(simOut)));
legend('Real part of expected data','Real part of demodulated data');
title('Comparison of input to OFDM Modulator with output from OFDM Demodulator');
xlabel('OFDM Subcarriers');
ylabel('Real part');

subplot(2,1,2)
plot(imag(data(:)))
hold on
plot(squeeze(imag(simOut)))
legend('Imaginary part of expected data','Imaginary part of demodulated data');
title('Comparison of input to OFDM Modulator with output from OFDM Demodulator');
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 NR Polar 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 preprending 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`.

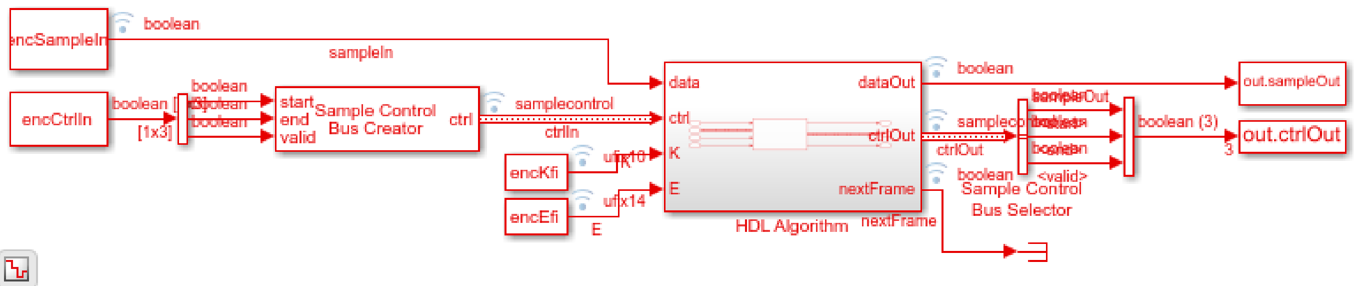
For this example, the number of invalid cycles between frames is empirically chosen to accomodate the latency of the NR Polar Encoder block for the specified **K** and **E** values. When the values of **K** and **E** are larger than in this example, the number of invalid cycles between frames must be 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 comparision 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 u
    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 accomodate 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.

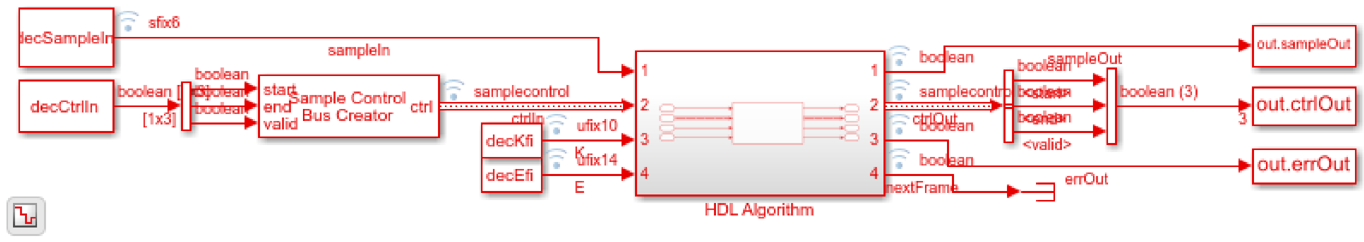
```
nVar = 1.5;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);
idleCyclesBetweenFrames = 2500;
btwFrames = false(idleCyclesBetweenFrames,1);
```

```
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. 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 comparision 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 = 2;
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 downlinl
    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
```

```
Maximum frame size computed to be 121 samples.
Decoded Frame 1: Behavioral and HDL simulation differ by 54 bits
The decoded output message from the HDL simulation differs from the input message by 69 bits
HDL and behavioral simulations both reported a decode error.
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`