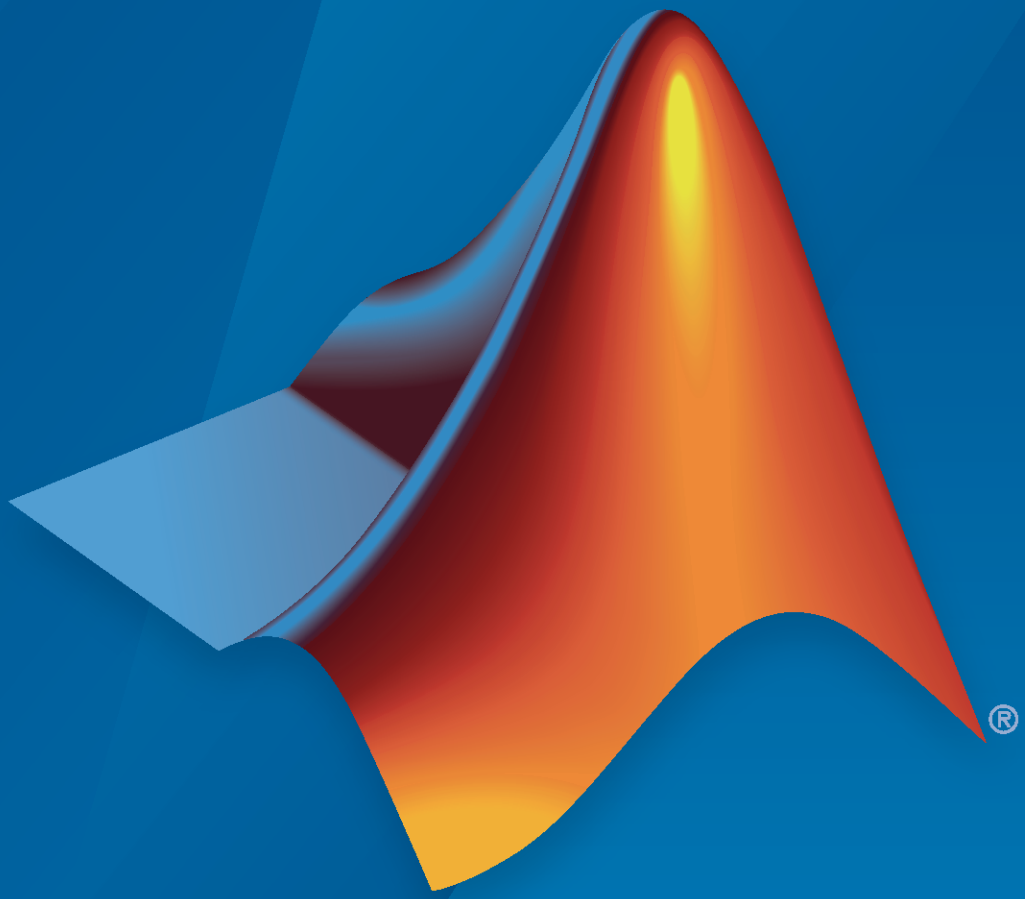
Wireless HDL Toolbox™

Getting Started Guide



MATLAB® & SIMULINK®

R2023a

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™ Getting Started

Revision History

September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 2.0 (Release 2020a)
September 2020	Online only	Revised for Version 2.1 (Release 2020b)
March 2021	Online only	Revised for Version 2.2 (Release 2021a)
September 2021	Online only	Revised for Version 2.3 (Release 2021b)
March 2022	Online only	Revised for Version 2.4 (Release 2022a)
September 2022	Online only	Revised for Version 2.5 (Release 2022b)
March 2023	Online only	Revised for Version 2.6 (Release 2023a)

Contents

Wireless HDL Toolbox Getting Started

Wireless HDL Toolbox Product Description

Design and implement 5G and LTE communications subsystems for FPGAs, ASICs, and SoCs

Wireless HDL Toolbox (formerly LTE HDL Toolbox™) provides pre-verified, hardware-ready Simulink® blocks and subsystems for developing 5G, LTE, WLAN, satellite and custom OFDM-based wireless communication applications. It includes reference applications, IP blocks, and gateways between frame- and sample-based processing.

You can modify the reference applications for integration into your own design. HDL implementations of the toolbox algorithms are optimized for efficient resource usage and performance for prototyping or for production deployment on FPGA, ASIC, and SoC devices.

The toolbox algorithms are designed to generate readable, synthesizable code in VHDL® and Verilog® (with HDL Coder™). For over-the-air testing of 5G, LTE, and custom OFDM-based designs, you can connect transmitter and receiver models to radio devices (with Communications Toolbox™ hardware support packages).

Tutorials

Introduction to 5G NR Signal Detection

This example shows how to implement PSS detection and SSB grid recovery for HDL code generation.

Introduction

This example introduces 5G NR signal detection, and a workflow for developing a design for HDL code generation. It uses a simplified version of the algorithm implemented in the “NR HDL Cell Search” reference application. The NR HDL Cell Search reference application is highly configurable and extensible, resulting in a complex design. This example focusses on the core concepts of the signal detection algorithm by reducing the complexity and flexibility of the design, in order to introduce the workflow used to develop an HDL implementation.

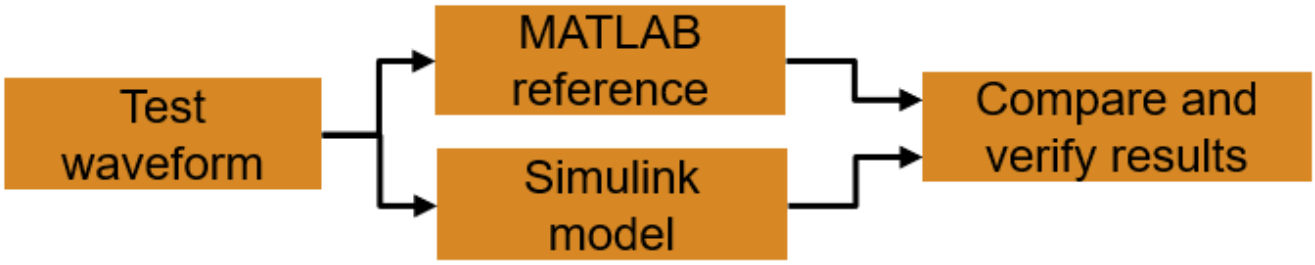
5G downlink signals contain a synchronization signal block (SSB), also referred to as the synchronization signal / physical broadcast channel (SS/PBCH block), which can be used by a receiver to synchronize to the signal, and to decode some parameters of the cell. An SSB consists of the primary and secondary synchronization signals (PSS and SSS), and the physical broadcast channel (PBCH) with its demodulation reference signals (DMRS). The PSS and SSS sequence numbers are used to calculate the cell identification number, and the PBCH carries the master information block (MIB). For frequency range 1 (FR1, carrier frequencies < 6 GHz) the SSB will use a subcarrier spacing (SCS) of either 15 or 30 kHz. More information on the contents of the SSB can be found in the tutorial “Synchronization Signal Blocks and Bursts” (5G Toolbox).

This example uses the PSS for signal detection and timing synchronization and the SSB is OFDM demodulated using the timing information from the PSS. The SSS and PBCH are decoded from the OFDM demodulated data to verify the data recovered by the signal detection and demodulation algorithm.

Example Structure

The signal detection and demodulation algorithm is implemented in both floating point MATLAB code and a fixed-point Simulink model. Developing the first version of the design in MATLAB allows for a focus on the high level algorithm without including the additional complexity of hardware implementation details. Once the MATLAB reference algorithm is proven to work this is used as the basis for a Simulink model that supports HDL code generation. The MATLAB reference can be used to verify the output of the Simulink model.

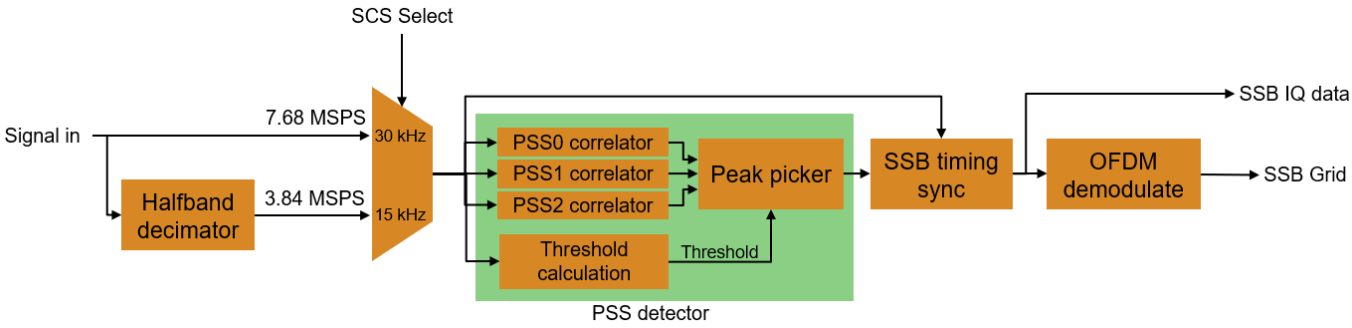
The Simulink model supports HDL code generation and includes hardware implementation details such as fixed-point data types, control signals, buffering, delay balancing and serialization for reducing resource use. The output from the Simulink model matches the output from the MATLAB reference code, with some small quantization error on the data signals due to the fixed-point data types used in the Simulink model.



When you run this example, the script loads a test waveform from a file and defines variables used by the signal detection and demodulation algorithm. The MATLAB reference code performs signal detection and SSB demodulation, and then the script simulates the Simulink model to perform the same operations on the test waveform. The outputs from the MATLAB reference and the Simulink model are compared to verify the behavior. The MIB message is decoded from the SSB grid recovered by the Simulink model to confirm the SSB grid data is correct.

Signal Detection and Demodulation Algorithm Overview

The PSS in the received signal is detected using three correlators, one for each of the three possible PSS sequences defined by the 5G standard. A strong correlation between the received signal and one of the expected PSS sequences indicates that a signal has been detected. The signal power is measured over the correlation length, and is used to calculate a threshold for the correlation output. A signal is detected when one of the correlator outputs exceeds the threshold. The scaling factor is used to set the sensitivity of the detector and help to avoid false positive detections. A minimum threshold value is also defined to prevent noise triggering the detector where no signal is present.



The PSS detector selects the first occasion that the output of any of the three PSS correlators exceeds the threshold. The index of the PSS correlator that exceeded the threshold gives nCellID2 for the cell, which is part of the physical layer cell identity. The location of the correlation peak is used to determine the timing of the start of the SSB. The SSB timing information is used to OFDM demodulate the SSB resource grid. This resource grid contains the PSS, SSS, PBCH and BCH-DMRS. The information contained in the grid can be verified by decoding the MIB information using 5G Toolbox functions.

The detector requires an input sampling rate of 7.68 MSPS. It supports FR1 SSB configurations, and OFDM demodulates the SSB from the first PSS that exceeds the threshold. For 30 kHz SSB subcarrier spacing the PSS correlator uses the 7.68 MSPS input signal. For the 15 kHz SSB subcarrier spacing a

halfband decimator is used to downsample the signal to 3.84 MSPS. The recovered SSB grid consists of 4 OFDM symbols, each with 240 subcarriers.

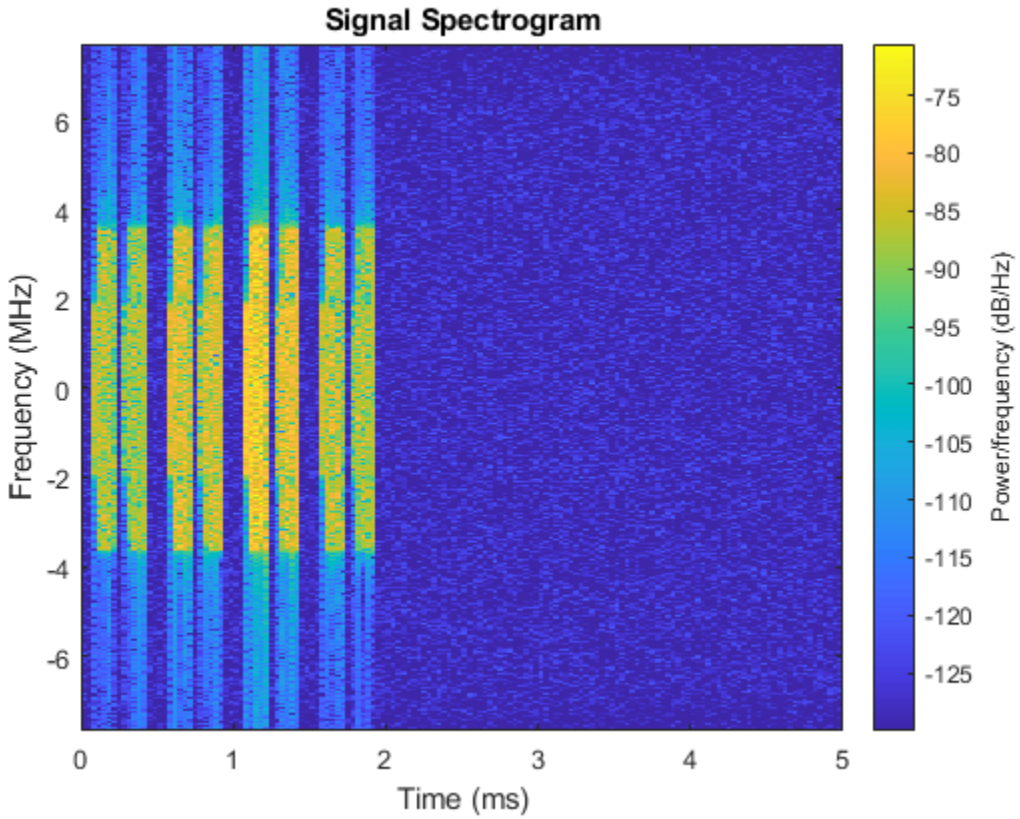
Setup Test Waveform and Workspace Variables

Load the test waveform from `waveformIntroSignalDetection.mat`. The waveform was generated using 5G Toolbox. For more information on generating 5G waveforms refer to the “5G NR Downlink Vector Waveform Generation” (5G Toolbox) example. The waveform is a FR1, 5 ms in duration, with SSB Pattern Case C and a subcarrier spacing of 30 kHz. The signal is normalized to have a maximum absolute value of 0.95.

```
rx = load('waveformIntroSignalDetection.mat');
rxWaveform = rx.waveform;
Fs_rxWaveform = rx.sampleRate;
ssbPattern = rx.ssbBlockPattern;
Lmax = rx.L_max;
```

Plot spectrogram of rxWaveform.

```
figure;
spectrogram(rxWaveform(:,1),ones(512,1),0,512,'centered',Fs_rxWaveform,'yaxis','MinThreshold',-1
title('Signal Spectrogram');
```



Set SSB subcarrier spacing (`scsSSB`) based on the SSB pattern (`ssbPattern`). SSB pattern Case A uses 15 kHz subcarrier spacing. SSB patterns Case B and Case C use 30 kHz subcarrier spacing.

```
if strcmp(ssbPattern,"Case A")
    scsSSB = 15;
```

```
else
    scsSSB = 30;
end
```

Resample the signal to 7.68e6, the sampling rate required at the input to the detector.

```
Fs_in = 7.68e6;
signalIn = resample(rxWaveform,Fs_in,Fs_rxWaveform);
```

Define constants for detector implementation.

```
Fs_30 = 7.68e6;  % Sampling frequency for 30 kHz subcarrier spacing (=Fs_in)
Fs_15 = 3.84e6;  % Sampling frequency for 15 kHz subcarrier spacing
Nfft = 256;      % FFT size (SSB occupies 240 subcarriers)
```

Get correlation filter coefficients for the 3 PSS sequences.

```
pssCorrelationCoeffs = getCorrelationCoeffs(Nfft);
```

The detector uses a halfband filter to reduce sample rate from the input sampling rate of 7.68 MHz to 3.84 MHz. The detector uses the sample rate 7.68 MHz for subcarrier spacing 30 kHz, and 3.84 MHz for subcarrier spacing 15 kHz.

```
Fc_15 = 120 * 15e3;
scs15halfband = dsp.FIRHalfbandDecimator(...
    'Specification',       'Transition width and stopband attenuation', ...
    'TransitionWidth',     Fs_15 - 2*Fc_15, ...
    'StopbandAttenuation', 60, ...
    'SampleRate',          Fs_in);
halfband_grpDelay = grpdelay(scs15halfband,2);
halfband_transient = ceil(halfband_grpDelay(2)/2);
```

MATLAB Reference Implementation of Signal Detection and Demodulation

Run the MATLAB implementation of the signal detection and demodulation algorithm and display the locations of the PSS peaks located in the waveform.

```
fprintf('Running MATLAB signal detection and demodulation code\n');
```

```
Running MATLAB signal detection and demodulation code
```

If the subcarrier spacing of the SSB is 15 kHz the input signal is downsampled by 2 using a halfband filter to reduce the sampling rate from 7.68 MSPS (`Fs_30`) to 3.84 MSPS (`Fs_15`). For 30 kHz subcarrier spacing no sampling rate change is required as the sample rate of the input signal (`Fs_in`) matches the sample rate required to process the signal (`Fs_30`). This functionality is implemented by the `nrhdlSignalDetection/SSB Detect and Demod/scsSelection` subsystem in the Simulink model.

```
if scsSSB == 15
    Fs_correlator = Fs_15;
    halfbandOut = scs15halfband(signalIn);
    correlatorIn = halfbandOut(halfband_transient:end);
else
    Fs_correlator = Fs_30;      % Fs_30 == Fs_in
    correlatorIn = signalIn;
end
```

Correlate signal with all 3 PSS sequences. Each of the PSS sequences is Nfft coefficients in length. This functionality is implemented by the blocks `PSS Correlator 0`, `PSS Correlator 1` and `PSS`

`Correlator 2` in the `nrhdlSignalDetection/SSB Detect and Demod/PSS Detector` subsystem.

```
psscorr      = zeros(length(correlatorIn),3);
psscorr(:,1) = filter(pssCorrelationCoeffs(:,1),1,correlatorIn);
psscorr(:,2) = filter(pssCorrelationCoeffs(:,2),1,correlatorIn);
psscorr(:,3) = filter(pssCorrelationCoeffs(:,3),1,correlatorIn);
```

Calculate the magnitude squared of each of the PSS correlator outputs. This functionality is implemented by the blocks `xCorrMag0`, `xCorrMag1` and `xCorrMag2` in the `nrhdlSignalDetection/SSB Detect and Demod/PSS Detector` subsystem.

```
psscorr_mag_sq = abs(psscorr).^2;
```

Calculate signal energy at the PSS correlators using a moving average filter. PSS correlations are calculated over Nfft samples, so the signal energy is measured over the same window. This calculation is implemented by the blocks `signalMag` and `signalMagAverage` in the `nrhdlSignalDetection/SSB Detect and Demod/PSS Detector` subsystem.

```
energyFilt  = ones(Nfft,1);
sig_mag_sq  = abs(correlatorIn).^2;
sigEnergy   = filter(energyFilt,1,sig_mag_sq);

% Calculate threshold for correlation values from signal energy.
minimumThreshold  = Nfft*(2^-12)^2;          % Set minimumThreshold to avoid triggering on noise
PSSThreshold_dB  = -6;                       % Strength threshold in dBs (0 is least sensitive).
thresholdScaling = 10^(PSSThreshold_dB/10);  % Threshold scaling factor
threshold = sigEnergy.*thresholdScaling;     % Calculate the threshold using the signal power and
threshold(threshold<minimumThreshold) = minimumThreshold; % Apply minimum threshold value where c
```

Find correlation values that exceed the threshold, and pick maximum value within `window_length` samples of each peak to ensure local maximum is selected. This is implemented in the `nrhdlSignalDetection/SSB Detect and Demod/PSS Detector/Find Peak` subsystem in the Simulink model.

```
window_length=11;
[locations_1] = peakPicker(psscorr_mag_sq(:,1),threshold,window_length);
[locations_2] = peakPicker(psscorr_mag_sq(:,2),threshold,window_length);
[locations_3] = peakPicker(psscorr_mag_sq(:,3),threshold,window_length);
```

Combine all of the locations into a single vector and check that some peaks were found.

```
ml_locations = [locations_1 locations_2 locations_3];
if isempty(ml_locations)
    fprintf('No correlation peaks found \n');
    return;
end
```

Select the first correlation peak found. The PSS sequence number of the filter producing the peak gives nCellID2 for the signal. This function is performed by `nrhdlSignalDetection/SSB Detect and Demod/PSS Detector/Find Peak/register PSS ID` in the Simulink model.

```
selectedPeak = min(ml_locations);
[~,pssCorrNumber] = max(psscorr(selectedPeak,:));
ml_nCellID2 = pssCorrNumber-1;
```

Calculate sample offset from PSS peak to start of SSB. The correlation peak occurs at the end of the OFDM symbol containing the PSS. To find the start of the OFDM symbol carrying the PSS the FFT

length is added to the cyclic prefix length to give the sample offset. The cyclic prefix length for the SSB OFDM symbols is 18 samples. This functionality is implemented by the `delay to symbol start` and `OFDM Demod Controller` blocks in the `nrhdlSignalDetection/SSB Detect and Demod/OFDM Demodulator` subsystem.

```
pssSymbolStart = selectedPeak-(Nfft+18-1); % Jump back to start of OFDM symbol containing PSS (FI
nrDemodSymbolEnd = pssSymbolStart + 4*(Nfft+18); % SSB is 4 OFDM symbols long
```

The nrOFDMDemodulate function requires input data starting at a slot boundary, which is one OFDM symbol before the start of the SSB. The first OFDM symbol in the slot has a different cyclic prefix length of either 20 for SSB subcarrier spacing of 15 kHz, or 22 for SSB subcarrier spacing of 30 kHz. The starting point of demodulation is adjusted to account for this. This additional offset is not required in the Simulink implementation as OFDM Demodulator block does not have the requirement to start on a slot boundary.

```
if scsSSB == 15
    slotFirstCyclicPrefix = 20;
else
    slotFirstCyclicPrefix = 22;
end
nrDemodSymbolStart = pssSymbolStart-(Nfft+slotFirstCyclicPrefix);
```

OFDM Demodulate the SSB. In the Simulink model this functionality is performed by the `OFDM Demodulator` block in the subsystem `nrhdlSignalDetection/SSB Detect and Demod/OFDM Demodulator`.

```
carrier = nrCarrierConfig('SubcarrierSpacing',scsSSB,'NSizeGrid',20);
nrDemodSignalIn = correlatorIn(nrDemodSymbolStart:nrDemodSymbolEnd-1);
grid = nrOFDMDemodulate(carrier,nrDemodSignalIn,'SampleRate',Fs_correlator,'Nfft',256);
```

Discard first OFDM symbol to get SSB from grid. The `ml_SSBGrid` signal is equivalent to output from the OFDM Demodulator in the Simulink model, `sl_SSBGrid`.

```
ml_SSBGrid = grid(:,2:5);
```

Get the IQ samples that cover the SSB region, prior to OFDM demodulation. This is equivalent to the input to the OFDM Demodulator in the Simulink model, `sl_SSBRawIQ`.

```
ml_SSBRawIQ = nrDemodSignalIn(Nfft+slotFirstCyclicPrefix+1:end);

fprintf('MATLAB signal detection and demodulation complete\n');
```

```
MATLAB signal detection and demodulation complete
```

Display MATLAB peak locations and nCellID2.

```
fprintf('MATLAB Peak Location(s) \n');
fprintf('    %i \n',ml_locations);

fprintf('MATLAB nCellID2 = %i\n', ml_nCellID2);
```

```
MATLAB Peak Location(s)
    826
    2470
    4666
    6310
    8506
    10150
```

```
    12346
    13990
MATLAB nCellID2 = 0
```

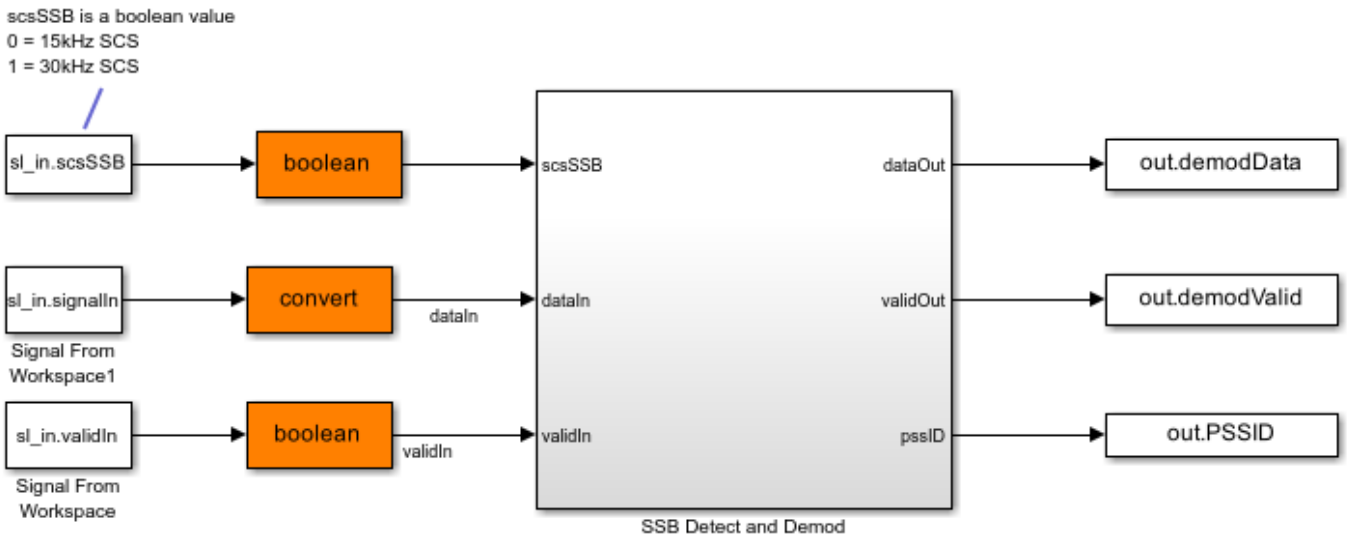
Simulink Model of Signal Detection and Demodulation

The Simulink model implements the signal detection and demodulation algorithm using a hardware modelling style and uses blocks that support HDL code generation. It uses fixed-point arithmetic and includes control signals to control the flow of data through the model. The Simulink model also uses a clock rate that is higher than the sample rate of the input signal, which makes it possible to share resources within the correlation filters to reduce the resource requirements of the design. The variable `sl_in.cyclesPerSample` is used to set the number of clock cycles per input sample. With this variable set to 8 the clock rate used in the model is 61.44 MHz, which is 8x larger than the input sampling rate Fs_in (7.68 MHz). To implement this in the Simulink model the input signal, `sl_in.signalIn`, has seven zeros inserted between each data sample and an input valid signal, `sl_in.validIn`, is created to indicate which of the input signal samples contains a valid data sample.

The input data signal is converted to a 16 bit signed fixed-point value, with 15 fractional bits. A boolean value is used to specify the subcarrier spacing for the SSB, with the value 0 representing 15 kHz, and 1 representing 30 kHz. The demodulated SSB grid output is a 24 bit signed fixed-point value with 15 fractional bits. The PSS ID output is an unsigned 2 bit integer number.

```
fprintf('Opening Simulink model\n');
open_system('nrhdlSignalDetection');
```

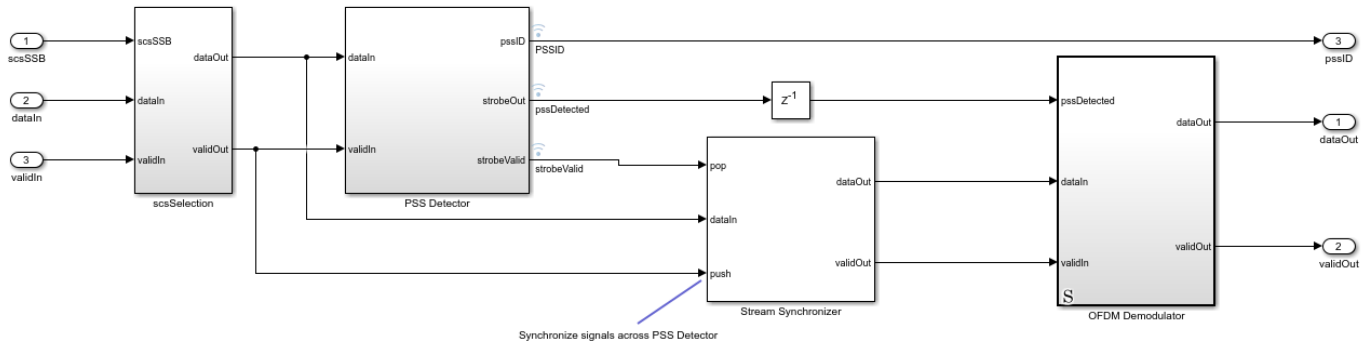
```
Opening Simulink model
```



There are 4 subsystems inside the `SSB Detect and Demod` subsystem, as shown in the diagram. The `scsSelection` subsystem performs a halfband decimation on the input signal and uses the `scsSSB` input to select between the 7.68 MSPS signal at the input for 30 kHz SCS and the 3.84 MSPS signal from the halfband decimator for 15 kHz SCS. The output from the `scsSelection` subsystem is equivalent to the `correlatorIn` signal in the MATLAB code. The `PSS Detector` subsystem implements the PSS correlation filters and threshold calculation, generating a strobe, `strobeOut`, a valid signal, `strobeValid`, and outputting the `pssID` when a PSS signal exceeds the threshold. The

`Stream Synchronizer` subsystem uses a RAM FIFO to balance the delays introduced by the `PSS Detector`. The `OFDM Demodulator` subsystem generates the control signals for the OFDM demodulation from the first `pssDetected` strobe and performs OFDM demodulation.

```
open_system('nrhdlSignalDetection/SSB Detect and Demod','window');
```




```
close_system('nrhdlSignalDetection/SSB Detect and Demod');
```

Initialize constants required by Simulink model.

```
sl_in.scsSSB = isequal(scsSSB,30); % Convert scsSSB to boolean 0=15kHz, 1=30kHz
sl_in.Nfft = Nfft;
sl_in.thresholdScaling = fi(thresholdScaling,0,16,16);
sl_in.minimumThreshold = fi(minimumThreshold,0,16,20);
sl_in.halfbandCoeffs = [0 scs15halfband.coeffs.Numerator];
sl_in.halfbandTransient = halfband_transient-1;
sl_in.pssCorrelationCoeffs = pssCorrelationCoeffs;
sl_in.windowLength = window_length;
```

Set the simulation duration to 4 ms. The spectrogram of the input waveform shows all of the SSB data in the first 4 ms of the signal.

```
sl_in.stopTime = 4e-3;
```

Set the number of clock cycles per input data sample.

```
sl_in.cyclesPerSample = 8;
```

Set the Simulink sample time based on `Fs_in` and `sl_in.cyclesPerSample` to allow for resource sharing.

```
sl_in.sampleTime = 1/(Fs_in*sl_in.cyclesPerSample);
```

Set input data valid high for 1 in every `sl_in.cyclesPerSample`, and pad input data signal with zeros.

```
sl_in.signalIn = upsample(signalIn,sl_in.cyclesPerSample);
sl_in.validIn = upsample(ones(1,length(signalIn)),sl_in.cyclesPerSample);
```

Run Simulink model.

```
fprintf('Starting Simulink simulation\n');
sl_out = sim('nrhdlSignalDetection');
fprintf('Simulink simulation finished\n');
```

```
Starting Simulink simulation
Simulink simulation finished
```

Process Simulink output signals.

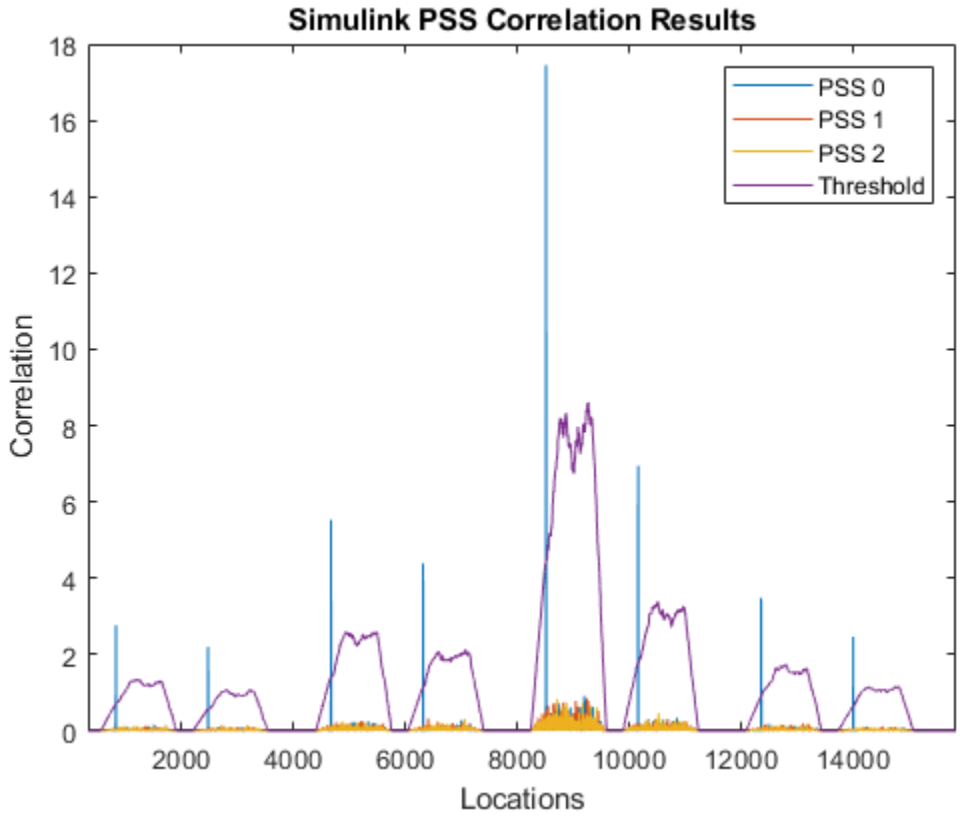
```
sl_correlatorIn = sl_out.correlatorIn(sl_out.correlatorInValid);
sl_demod = sl_out.demodData(sl_out.demodValid);
sl_SSBGrid = reshape(sl_demod(1:240*4),240,[]);
sl_locationPeaks = sl_out.locationPeaks(sl_out.locationPeaksValid);
sl_locations = find(sl_locationPeaks);
sl_correlationOut = sl_out.correlationOutput(sl_out.correlationValid,:);
sl_thresholdOut = sl_out.threshold(sl_out.correlationValid,:);
sl_SSBRawIQ = sl_out.ofdmInputData(sl_out.ofdmInputValid);
sl_nCellID2 = sl_out.PSSID(end);
```

Display and compare results

Display simulation results when both MATLAB and Simulink simulations are complete.

Plot PSS correlation peaks and PSS threshold from Simulink output signals.

```
figure;
plot(sl_correlationOut);
hold on;
plot(sl_thresholdOut);

% Zoom in on first PSS peak and surrounding area.
xlim([min(sl_locations)-500 min(sl_locations)+15000]);
legend('PSS 0','PSS 1','PSS 2','Threshold');
title('Simulink PSS Correlation Results')
ylabel('Correlation');
xlabel('Locations');
```



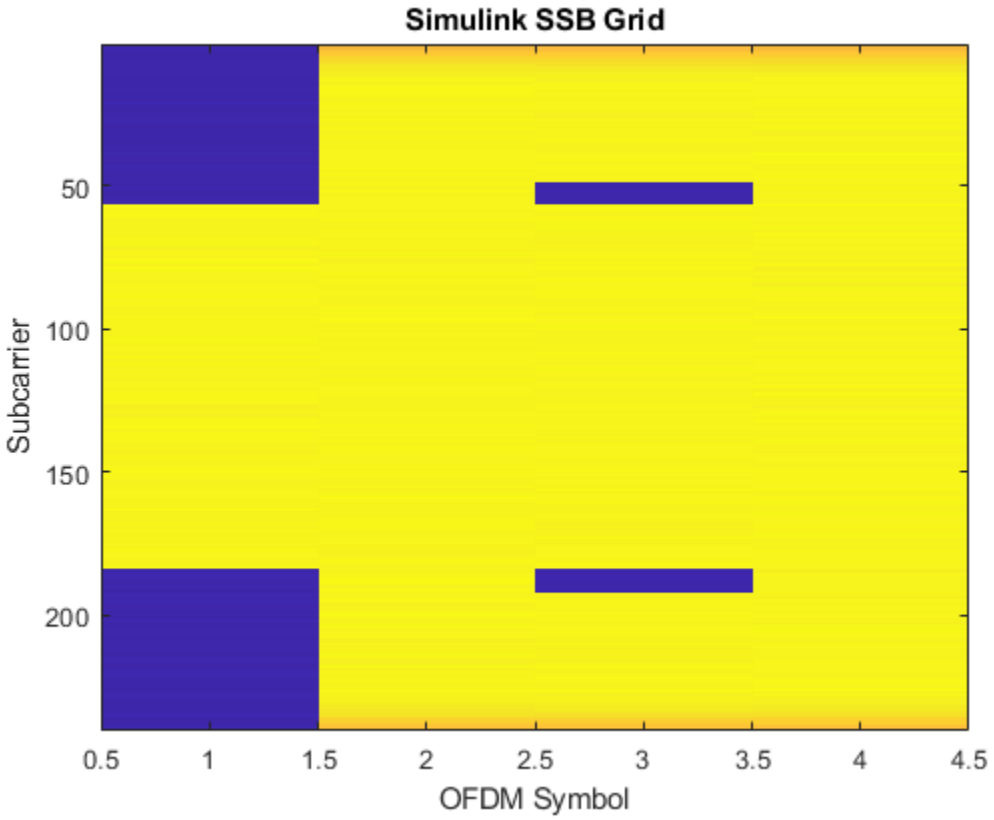
Display Simulink peak locations and nCellID2.

```
fprintf('Simulink Peak Location(s) \n');
fprintf('    %i \n',sl_locations);

fprintf('Simulink nCellID2 = %i \n', sl_nCellID2);
```

```
Simulink Peak Location(s)
    826
    2470
    4666
    6310
    8506
    10150
    12346
    13990
Simulink nCellID2 = 0
```

Plot demodulated SSB grid from Simulink output signals.

```
figure;
imagesc(abs(sl_SSBGrid));
title('Simulink SSB Grid')
ylabel('Subcarrier');
xlabel('OFDM Symbol');
```



Calculate relative mean-squared error between MATLAB and Simulink IQ samples for the SSB grid before OFDM demodulation.

```
SSBIQ_MSE = 10*log10(sum(abs(ml_SSBRawIQ-sl_SSBRawIQ).^2)/sum(abs(ml_SSBRawIQ).^2));
fprintf('Relative mean-squared error between MATLAB and Simulink IQ samples for SSB grid at input
```

```
Relative mean-squared error between MATLAB and Simulink IQ samples for SSB grid at input to OFDM
   -80.5 dB
```

Calculate relative mean-squared error between MATLAB and Simulink SSB grids.

```
SSBGrid_MSE = 10*log10(sum(abs(ml_SSBGrid(:)-sl_SSBGrid(:)).^2)/sum(abs(ml_SSBGrid(:)).^2));
fprintf('Relative mean-squared error between MATLAB and Simulink SSB grids:\n   %.1f dB\n',SSBGri
```

```
Relative mean-squared error between MATLAB and Simulink SSB grids:
   -79.3 dB
```

Decode the MIB from the Simulink SSB grid to further verify data recovered from the signal. If the bchCRC returned is 0 this indicates no errors were found.

```
fprintf('Decoding MIB from Simulink SSB grid \n');
[mibStruct, nCellID, ssbIndex, bchCRC] = mibDecode(sl_SSBGrid, sl_nCellID2, Lmax);

if bchCRC == 0
    fprintf('MIB decode successful\n');
    disp([' Cell identity: ' num2str(nCellID)]);
else
```

```
    fprintf('MIB decode failed, bchCRC was non-zero\n')
end
```

```
Decoding MIB from Simulink SSB grid
MIB decode successful
 Cell identity: 249
```

HDL Code Generation

You can generate HDL code and an HDL testbench for the `nrhdlSignalDetection/SSB Detect and Demod` subsystem in the Simulink model using the makehdl and makehdltb commands. An HDL Coder™ license is required for HDL code generation. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 100 MHz.

```
T = table(...
    categorical({'Slice Registers'; 'Slice LUTs'; 'RAMB18'; 'RAMB36'; 'DSP48'}),...
    [29217; 14222; 9; 1; 313],...
    'VariableNames',{'Resource','Usage'});

disp(T);
```

```
       Resource        Usage
    _______________    _____

    Slice Registers    29217
    Slice LUTs         14222
    RAMB18                 9
    RAMB36                 1
    DSP48                313
```

Further Exploration

1. You can use 5G Toolbox to generate different signals to test the detector. The “Synchronization Signal Blocks and Bursts” (5G Toolbox) example and the “5G NR Downlink Vector Waveform Generation” (5G Toolbox) example.
2. You can try adding impairments to the signal, such as noise or a frequency offset.

See Also

More About

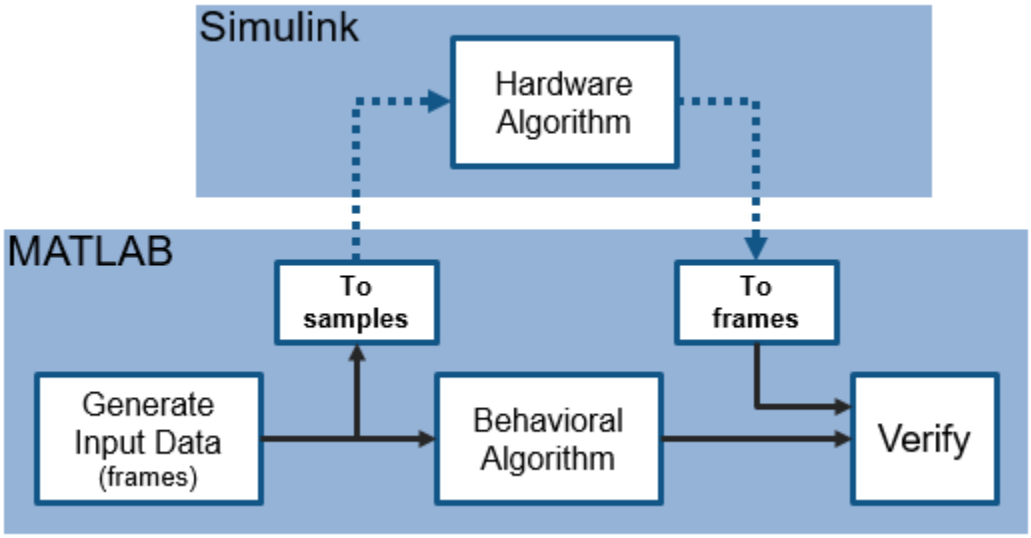
- “NR HDL Reference Applications Overview”

Verify Turbo Decoder with Streaming Data from MATLAB

This example shows how to verify a hardware-targeted Turbo Decoder design using streaming data from MATLAB®.

To run this example, use the VerifyLTEHDLTurboDecoderStreamingData.m script.

LTE Toolbox™ and 5G Toolbox™ functions model operations on framed, floating-point and integer data and provide excellent behavioral references. Hardware designs must use streaming Boolean or fixed-point data. This example converts frames to samples in MATLAB and imports the sample stream to Simulink® for hardware algorithm design. The same data is applied to both the hardware algorithm in Simulink, and the behavioral algorithm in MATLAB. The output sample stream from the Simulink simulation is exported to MATLAB and then converted back to framed data for comparison.



Hardware Targeting in Simulink

The key features of a model for hardware targeting in Simulink® are:

- **Streaming Sample Interface**: LTE Toolbox and 5G Toolbox functions process frames while blocks in Wireless HDL Toolbox use a streaming sample interface. Serial processing is efficient for hardware designs. For further information, see “Streaming Sample Interface”. You can convert frames to samples in Simulink using the Frame To Samples block or in MATLAB using the `whdlFramesToSamples` function. In this example, we convert the frames to samples in MATLAB using the `whdlFramesToSamples` function.
- **Subsystem Targeted for HDL Code Generation**: Design a hardware-friendly sample-streaming model by selecting blocks from the “Blocks”. The part of the design targeted for HDL code generation must be in a separate subsystem.
- **Conversion of Sample-Based Output to Frames**: For verification, you can export the result of your hardware-compatible design to the MATLAB® workspace. You can then compare this result with the output of a MATLAB behavioral design. In this example, we convert the samples to frames in MATLAB using the `whdlSamplesToFrames` function.

You can use the VerifyLTEHDLTurboDecoderStreamingData.m MATLAB script to run the MATLAB behavioral code, set up, import data and run the Simulink™ model, export the data, and compare the behavioral and Simulink output.

The MATLAB script contains six parts:

1. **Behavioral Simulation of Turbo Decoder**
2. **Conversion of Input Frames to Samples**
3. **Set Up the Simulink Model for Hardware Design**
4. **Run Simulink Model**
5. **Conversion of Output Samples to Frames**
6. **Verify Output of Simulink Model**

Behavioral Simulation of Turbo Decoder

For a behavioral simulation of the design, use the `lteTurboDecode` function from LTE Toolbox. The input to this function, **softBits**, is also the input for the HDL targeted design. The behavioral output of the `lteTurboDecode` function, **rxBits** can be used for comparison to the output of the HDL targeted design. Both **softBits** and **rxBits** are frames of floating-point data.

```
% Turbo decoding of soft bits obtained from a noisy constellation
turboFrameSize = 6144;
txBits = randi([0 1],turboFrameSize,1);
codedData = lteTurboEncode(txBits);
txSymbols = lteSymbolModulate(codedData,'QPSK');
noise = 0.5*complex(randn(size(txSymbols)),randn(size(txSymbols)));
rxSymbols = txSymbols + noise;
scatter(real(rxSymbols),imag(rxSymbols),'co'); hold on;
scatter(real(txSymbols),imag(txSymbols),'rx')
legend('Rx constellation','Tx constellation')
softBits = lteSymbolDemodulate(rxSymbols,'QPSK','Soft');
rxBits = lteTurboDecode(softBits);
```

Conversion of Input Frames to Samples

First, convert the input frame to fixed-point.

```
inframes = fi(softBits, 1, 5, 2);
```

Next, convert the framed data to a stream of samples and control signals using the `whdlFramesToSamples` function. This function also adds invalid samples to the sample stream to mimic streaming hardware data. The output of this function is the input to the Simulink model.

In this example, the `whdlFramesToSamples` function is configured to provide the inputs to the LTE Turbo Decoder block in the Simulink model. No invalid samples are inserted between valid samples.

The LTE Turbo Decoder block takes in a frame of data, one sample at a time, runs through the specified number of iterations and can then accept another input frame. To allow the block processing time to run the 6 iterations, we include invalid samples between frames. In this example, we set the number of invalid samples between frames to 7 iterations (or 14 half-iterations, each of which corresponds to a frame of delay). To calculate the exact cycles needed to process a frame, look at the LTE Turbo Decoder.

The turbo encoder function returns the systematic bits first, followed by the first set of parity bits and finally, the second set of parity bits. The LTE Turbo Decoder however requires each set of systematic

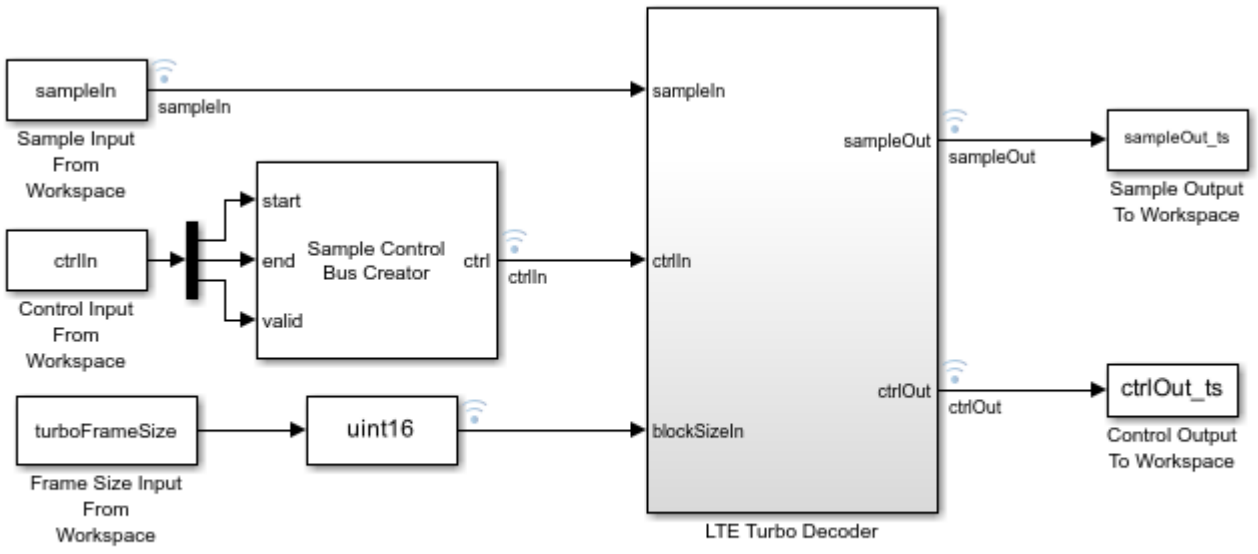
and 2 parity bits to be sent in together. To reshape the input data to the Turbo Decoder block accordingly, we choose the option to compose the output samples from interleaved input samples.

```
inframesize = 18444; % size of softBits

% No invalid cycles between samples
idlecyclesbetweensamples = 0;

% Additional delay in frames for Turbo Decoder output
numTurboIterations = 6;
% latency of Turbo Decoder block per half-iteration
tdlatency = (ceil(turboFrameSize/32)+4)*32;
% delay in frames for Turbo Decoder output
algframedelay = (2*numTurboIterations + 2)*(ceil(tdlatency/turboFrameSize));
idlecyclesbetweenframes = (inframesize/insamplesize)*algframedelay;

% Output is composed of interleaved input samples
% input: S_1 S_2 ... S_n P1_1 P1_2 ... P1_n P2_1 P2_2 ... P2_n
% output: S_1 P1_1 P2_1 S2 P1_2 P2_2 ... Sn P1_n P2_n

interleaveSamples = true;

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

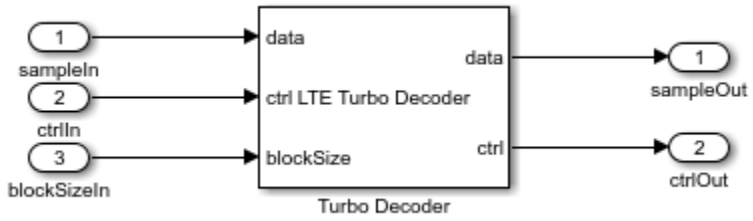
Set Up the Simulink Model for Hardware Design

The model imports sample-based data and control and the frame size from the MATLAB workspace.

```
modelname = 'TurboDecoderStreamingDataHDLExample';
```



The **LTE Turbo Decoder** subsystem consists of the LTE Turbo Decoder block. This block is set to run 6 iterations of decoding.



The MATLAB code defines the settings for the input size, sample time for the Simulink model, and computes the desired simulation time.

```
% Settings for Signal From Workspace block
samplesizeIn = 3; % encode rate is 1/3

sampletime = 1;

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

Run Simulink Model

You can run the model by clicking the Play button or calling the sim command on the MATLAB command line.

```
sim(modelname);
```

Conversion of Output Samples to Frames

Running the model results in streaming samples and control signals logged from the model in the variable **sampleOut_ts** and **ctrlOut_ts**. The sample and control data are converted to frames using the `whdlSamplesToFrames` function.

```
% Reformat the logged data to form the sample and control output
sampleOut = squeeze(sampleOut_ts.Data);
ctrlOut = [squeeze(ctrlOut_ts.start.Data) ...
           squeeze(ctrlOut_ts.end.Data) ...
           squeeze(ctrlOut_ts.valid.Data)];
% Form frames from output sample and control data
outframes = whdlSamplesToFrames(sampleOut, ctrlOut);
% Gather all the bits - expecting only one frame
rxBits_hdl = outframes{:};
```

Verify Output of Simulink Model

Compare the output of the HDL simulation to the output of the behavioral simulation to find the number of bit mismatches between the behavioral code and hardware-targeted model.

This simulation results in no bit errors. Increasing the amount of noise added to the samples or reducing the number of iterations may result in bit errors.

```
% Check number of bit mismatches
numBitsDiff = sum(rxBits_hdl ~= rxBits);
fprintf(['\nLTE Turbo Decoder: Behavioral and ' ...
   'HDL simulation differ by %d bits\n\n'], numBitsDiff);
```

```
>> VerifyLTEHDLTurboDecoderStreamingData

Maximum frame size computed to be 6144 samples.

LTE Turbo Decoder: Behavioral and HDL simulation differ by 0 bits
```

Generate HDL Code and Verify its Behavior

Once your design is working in simulation, you can use HDL Coder™ to “Generate HDL Code” for the **LTE Turbo Decoder** subsystem. Use HDL Verifier™ to generate a “SystemVerilog DPI Test Bench” (HDL Coder) or run “FPGA-in-the-Loop”.

```
makehdl([modelname '/LTE Turbo Decoder'])   % Generate HDL code
makehdltb([modelname '/LTE Turbo Decoder']) % Generate HDL Test bench
```

See Also

Blocks
LTE Turbo Decoder

Functions
`lteTurboDecode` | `whdlFramesToSamples` | `whdlSamplesToFrames`

Related Examples

- “Verify Turbo Decoder with Framed Data from MATLAB” on page 2-19

More About

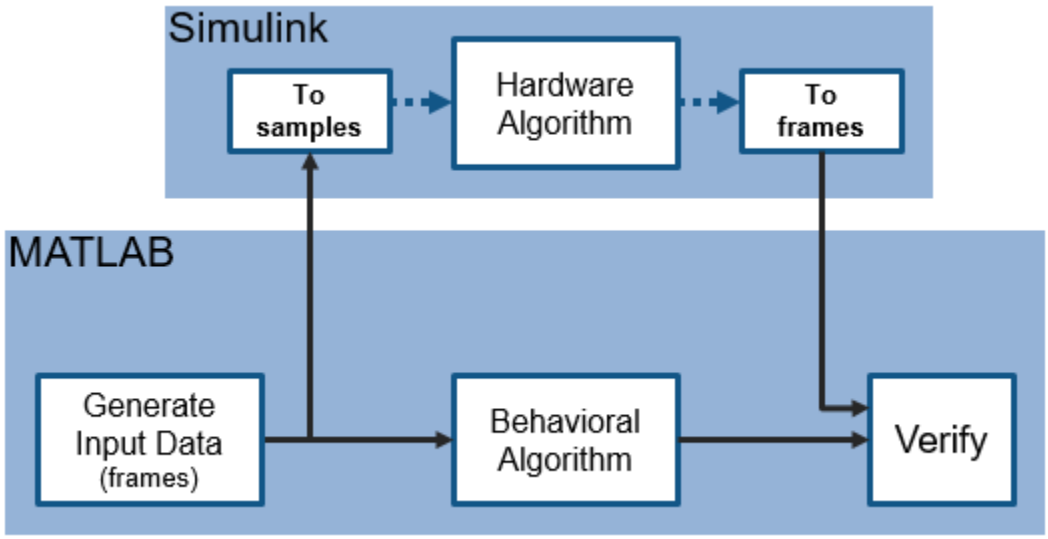
- “Turbo Encode Streaming Samples”

Verify Turbo Decoder with Framed Data from MATLAB

This example shows how to verify a hardware-targeted LTE Turbo Decoder design using frames of data from MATLAB®.

To run this example, use the VerifyLTEHDLTurboDecoderFramedData.m script.

LTE Toolbox™ and 5G Toobox™ functions model operations on framed, floating-point and integer data and provide excellent behavioral references. Hardware designs must use streaming Boolean or fixed-point data. This example imports framed data to Simulink® and then converts to a sample stream for hardware algorithm design. The same data is applied to both the hardware algorithm in Simulink, and the behavioral algorithm in MATLAB. The Simulink model converts the output sample stream to frames and exports the frames to MATLAB for comparison.



Hardware Targeting in Simulink

The key features of a model for hardware targeting in Simulink® are:

- **Streaming Sample Interface**: LTE Toolbox and 5G Toolbox functions process frames while blocks in Wireless HDL Toolbox use a streaming sample interface. Serial processing is efficient for hardware designs. For further information, see “Streaming Sample Interface”. You can convert frames to samples in Simulink using the Frame To Samples block or in MATLAB using the `whdlFramesToSamples` function. In this example, we convert the frames to a stream of samples in Simulink using the Frame To Samples block.
- **Subsystem Targeted for HDL Code Generation**: Design a hardware-friendly sample-streaming model by selecting blocks from the “Blocks”. The part of the design targeted for HDL code generation must be in a separate subsystem.
- **Conversion of Sample-based Output To Frames**: For verification, you can export the result of your hardware-compatible design to the MATLAB® workspace. You can then compare this result with the output of a MATLAB behavioral design. In this example, we convert the samples to frames in Simulink using the Samples To Frame block.

You can use the VerifyLTEHDLTurboDecoderFramedData.m MATLAB script to run the MATLAB behavioral code, set up, import data and run the Simulink™ model, export the data, and compare the behavioral and Simulink output.

The MATLAB script contains four parts:

1. **Behavioral Simulation of Turbo Decoder**
2. **Set Up the Simulink Model for Hardware Design**
3. **Run Simulink Model**
4. **Verify Output of Simulink Model**

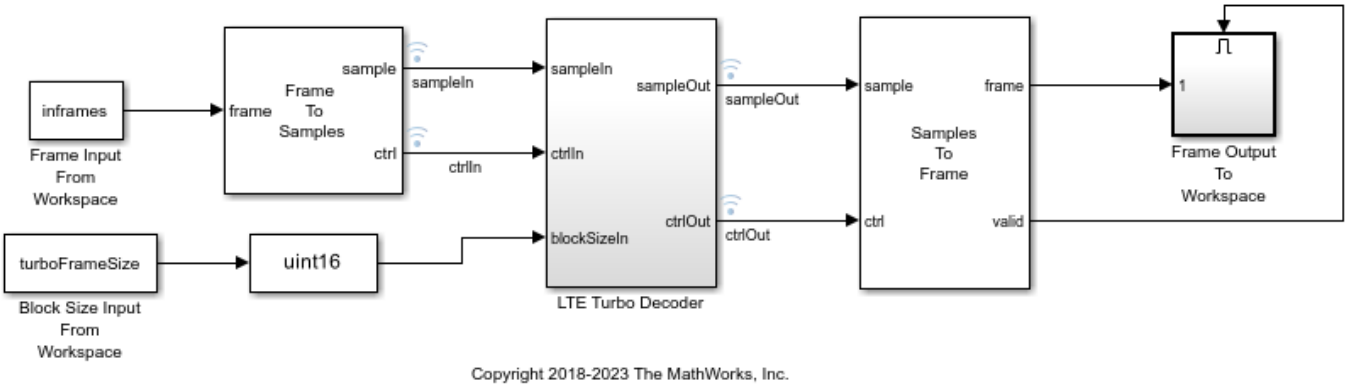
Behavioral Simulation of Turbo Decoder

For a behavioral simulation of the design, use the `lteTurboDecode` function from LTE Toolbox. This code generates input data, **softBits** that we can use as input for the HDL targeted design. The behavioral output of the `lteTurboDecode` function, **rxBits** can be used for comparison to the output of the HDL targeted design. Both **softBits** and **rxBits** are frames of floating-point data.

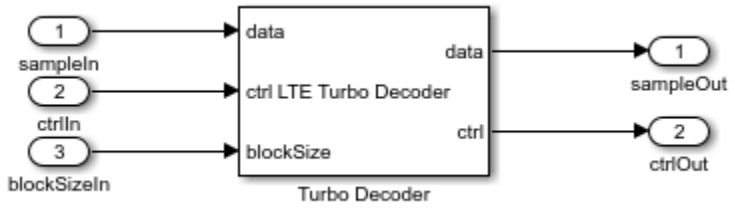
```
% Turbo decoding of soft bits obtained from a noisy constellation
turboFrameSize = 6144;
txBits = randi([0 1],turboFrameSize,1);
codedData = lteTurboEncode(txBits);
txSymbols = lteSymbolModulate(codedData,'QPSK');
noise = 0.5*complex(randn(size(txSymbols)),randn(size(txSymbols)));
rxSymbols = txSymbols + noise;
scatter(real(rxSymbols),imag(rxSymbols),'co'); hold on;
scatter(real(txSymbols),imag(txSymbols),'rx')
legend('Rx constellation','Tx constellation')
softBits = lteSymbolDemodulate(rxSymbols,'QPSK','Soft');
rxBits = lteTurboDecode(softBits);
```

Set Up the Simulink Model for Hardware Design

The model imports frame-based data from the MATLAB workspace. In this case, the variable *inframes* and *blockSizeIn* are the inputs to the model.



The **LTE Turbo Decoder** subsystem consists of the LTE Turbo Decoder block. This block is set to run for a frame size of 6144 systematic bits and 6 iterations of decoding.



The Simulink model imports frames of data from MATLAB and converts them into a stream of samples in the Simulink model.

The Frame To Samples block converts the framed data to a stream of samples and control signals. This block also adds invalid samples to the sample stream to mimic streaming hardware data. This block provides the input to the subsystem targeted for HDL code generation, but does not itself support HDL code generation.

In this example, the Frame To Samples block is configured to provide the input to the LTE Turbo Decoder block. No invalid samples are inserted between valid samples.

The Simulink model also imports the frame size parameter from the workspace and passes it to the blockSize port of the LTE Turbo Decoder block. This example uses one block size and so this port remains at a constant value specified by the *turboFrameSize* parameter.

The LTE Turbo Decoder block takes in one frame of data, runs the specified number of iterations, and can then accept another input frame. In order to allow the block processing time to run the 6 iterations, we send invalid samples between frames. In this example, we set the number of invalid samples between frames to 7 iterations (or 14 half-iterations, each of which corresponds to a frame of delay). To calculate the exact cycles needed to process a frame, look at the LTE Turbo Decoder.

The turbo encoder function sends all systematic bits out first, followed by the first set of parity bits and finally, the second set of parity bits. The LTE Turbo Decoder however requires each set of systematic and 2 parity bits to be sent in together. To reshape the input data to the LTE Turbo Decoder block accordingly, we choose the option to compose the output samples from interleaved input samples.

```
% Open the model
modelname = 'TurboDecoderFramedDataHDLExample';
open_system(modelname);

% Settings for Frame Input From Workspace block
inframesize = length(softBits);

% Setting for LTE Turbo Decoder block
numTurboIterations = 6;

% Settings for Frame To Samples block
idlecyclesbetweensamples = 0;
% approximate latency from LTE Turbo Decoder block per half-iteration
tdlatency = (ceil(turboFrameSize/32)+4)*32;
% approximate delay in frames for LTE Turbo Decoder output
algframedelay = (2*numTurboIterations + 2)*(ceil(tdlatency/turboFrameSize));
% Rate 1/3 code: Systematic + 2 Parity
insamplesize = 3;
idlecyclesbetweenframes = (inframesize/insamplesize)*algframedelay;
% Select the option to compose output from interleaved input samples
```

```
set_param([modelname '/Frame To Samples'],...
    'InterleaveSamples','on');

% Settings for Samples To Frame block
% Exact setting, includes idle cycles introduced
totalframesize = ((inframesize/insamplesize)*...
    (idlecyclesbetweensamples+1))+idlecyclesbetweenframes;
% Alternative setting, same size as LTE Turbo Decoder output frame
% totalframesize = outframesize;
outsamplesize = 1;
```

The MATLAB code also converts the input data to fixed-point for use in the hardware-targeted model, and computes the desired simulation time.

```
% Simulation time
numFrames = 1; % number of frames to run
simTime = numFrames;

% Input frame data for model
% use same data as lteTurboDecode, converted to fixed point
inframes = fi(softBits, 1, 5, 2);
```

Run Simulink Model

You can run the model by clicking the Play button or calling the sim command on the MATLAB command line.

```
sim(modelname);
```

Verify Output of Simulink Model

Running the model results in frames of data logged from the Frame Output To Workspace block in the variable **outframes_logdata**. Compare the data to the output of the behavioral simulation to find the number of bit mismatches between the behavioral code and hardware-targeted model.

This simulation results in no bit errors. Increasing the amount of noise added to the samples or reducing the number of iterations may result in bit errors.

```
% Process output of HDL model and compare to behavioral simulation output
% Gather all the logged data into one array
rxBits_hdl = outframes_logdata(:);

% Check number of bit mismatches
numBitsDiff = sum(rxBits_hdl ~= rxBits);
fprintf(['\nLTE Turbo Decoder: Behavioral and ' ...
   'HDL simulation differ by %d bits\n\n'], numBitsDiff);
```

```
>> VerifyLTEHDLTurboDecoderFramedData

LTE Turbo Decoder: Behavioral and HDL simulation differ by 0 bits
```

Generate HDL Code and Verify Its Behavior

Once your design is working in simulation, you can use HDL Coder™ to “Generate HDL Code” for the **LTE Turbo Decoder** subsystem. Use HDL Verifier™ to generate a “SystemVerilog DPI Test Bench” (HDL Coder) or run “FPGA-in-the-Loop”.

```
makehdl([modelname '/LTE Turbo Decoder'])   % Generate HDL code
makehdltb([modelname '/LTE Turbo Decoder']) % Generate HDL Test bench
```

See Also

Blocks
LTE Turbo Decoder | Frame To Samples | Samples To Frame

Functions
`lteTurboDecode`

Related Examples

- “Verify Turbo Decoder with Streaming Data from MATLAB” on page 2-14

More About

- “Turbo Encode Streaming Samples”