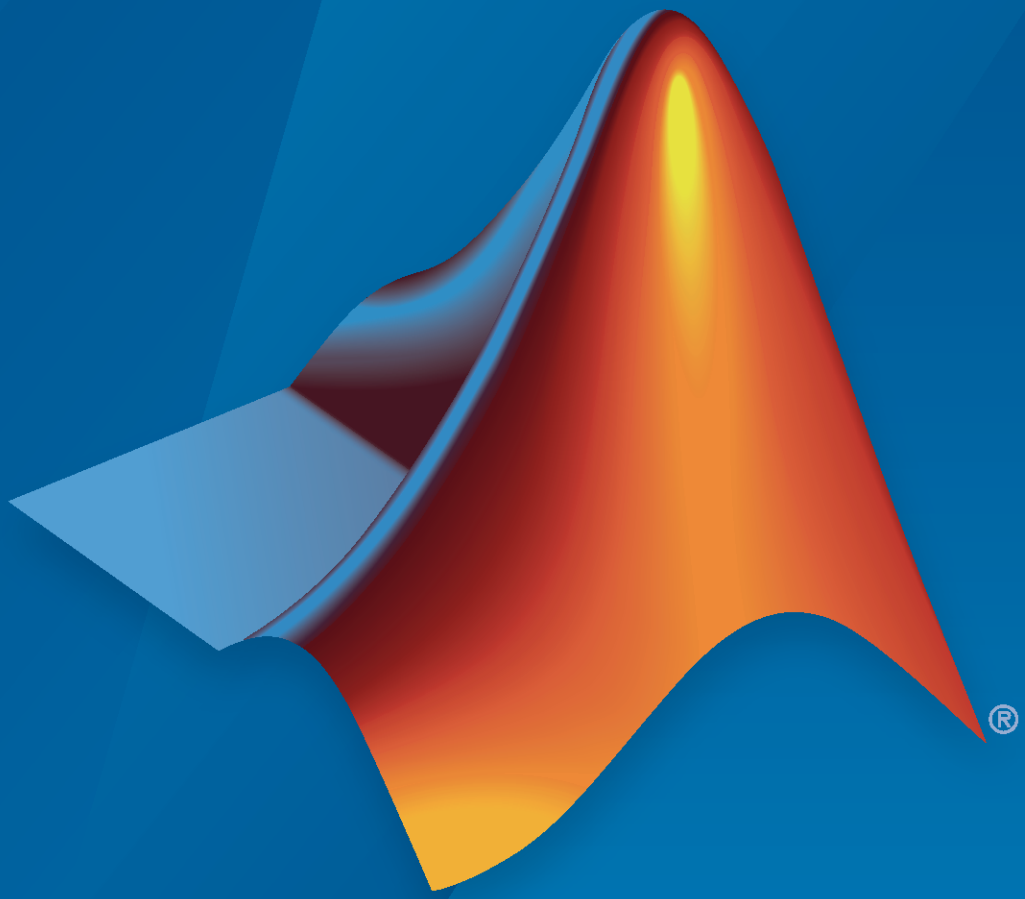
Mixed-Signal Blockset™

User's Guide



MATLAB® & SIMULINK®

R2022b

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

Mixed-Signal Blockset™ User's Guide

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

March 2019	Online only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)
March 2020	Online only	Revised for Version 1.2 (Release 2020a)
September 2020	Online only	Revised for Version 1.3 (Release 2020b)
March 2021	Online only	Revised for Version 2.0 (Release 2021a)
September 2021	Online only	Revised for Version 2.1 (Release 2021b)
March 2022	Online only	Revised for Version 2.2 (Release 2022a)
September 2022	Online only	Revised for Version 2.3 (Release 2022b)

Contents

1 PLL Featured Examples

2 DC Featured Examples

3 Mixing Analog and Digital Signals Featured Examples

4 PLL Block Level Examples

5 Data Converter Block Level Examples

6 Utilities Block Level Examples

7 Mix Analog and Digital Signals

PLL Featured Examples

Phase Noise at PLL Output

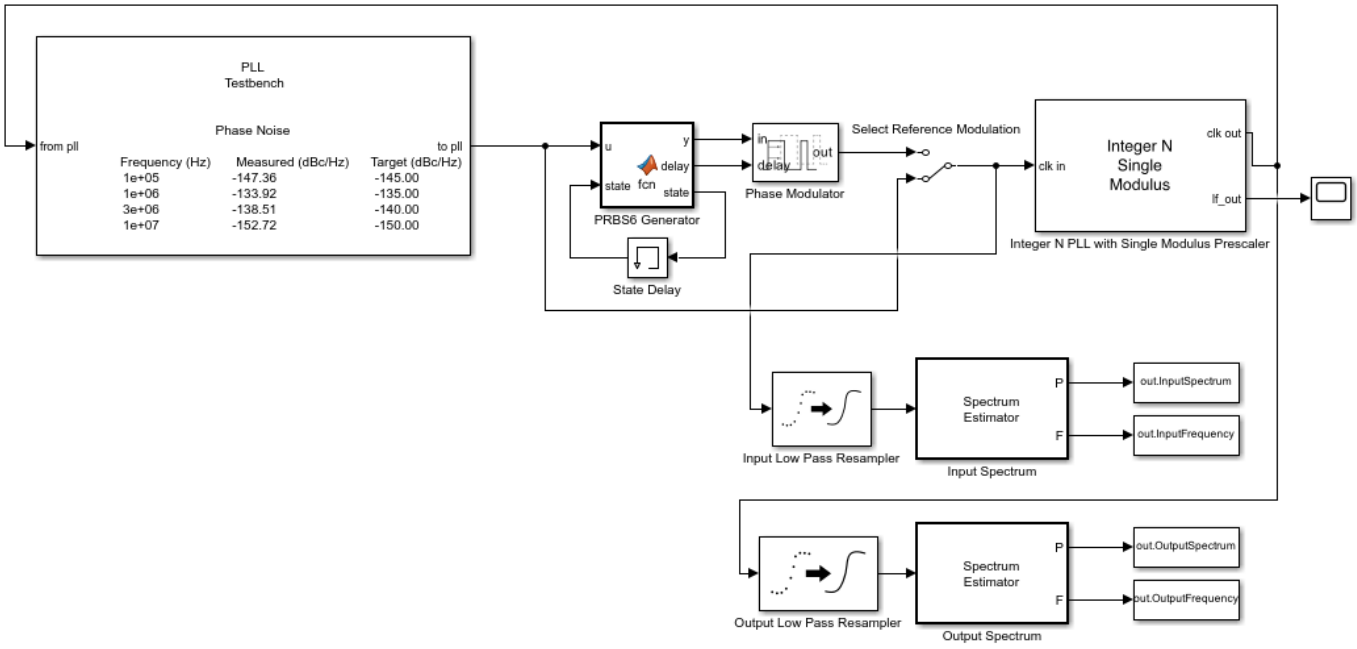
This example shows how to predict the phase noise at the output of a phase-locked loop (PLL), simulate the PLL using the PLL Testbench, and compare the simulation results to theoretical predictions.

This example demonstrates three phase noise effects, individually or combined, depending on the configuration you choose:

1. Reference modulation or phase noise
2. VCO phase noise
3. VCO phase noise subsampled by the feedback prescaler

Open the model `PllPhaseNoiseExample.slx`.

```
open_system('PllPhaseNoiseExample.slx');
```



This example uses an Integer N PLL with Single Modulus Prescaler from the Mixed Signal Blockset™.

The PLL Testbench generates the reference input signal for the PLL and measures the phase spectral density at the output of the PLL.

The optional PRBS6 reference phase modulation in this model is used to contrast the response to reference phase variation with the response to VCO phase noise.

The low pass resamplers at the inputs to the spectrum estimators are anti-aliasing filters. These filters convert the variable step discrete signals to the fixed step discrete signals required by the spectrum estimators.

The spectra of the PLL input and output signals are estimated and logged in the base workspace so that you can compare the simulation results to the results of the theoretical calculations.

The oscilloscope in the example model provides a progress indicator for the simulation.

To evaluate the behavior of the PLL under a variety of conditions, use a separate workspace file for each different set of conditions. This example provides five such supporting .mat files. All of them produce the same loop transient response.

1. `Baseline.mat` - As close to ideal as possible. Use to evaluate numerical noise introduced by the model and simulator.
2. `ReferenceModulation.mat` - Introduce reference phase modulation. Use to evaluate the phase transfer function of the PLL.
3. `VCOPhaseNoise.mat` - Introduce VCO phase noise and a prescaler ratio equal to one. Use to evaluate the PLL control loop's error function.
4. `InbandPhaseNoise.mat` - Introduce VCO phase noise and a prescaler ratio greater than one. Use to evaluate the effect of the prescaler ratio on the PLL control loop's error function.
5. `TotalPhaseNoise.mat` - All phase noise sources are enabled, and the prescaler ratio is greater than one.

Theoretical Calculations

This section shows how to calculate the expected response of the PLL.

The target loop bandwidth of the PLL is 2 MHz with a 45 degree phase margin. The loop filter components are scaled to practical level and the charge pump output current is scaled by the same factor to maintain the same loop dynamics.

The function `getPllLoopResponse` calculates the loop gain as a function of frequency and then calculates the expected response to signals from outside or inside the PLL. Define the input parameters such as charge pump output current, VCO sensitivity, prescaler ratio and passive loop filter component values for the `getPllLoopResponse` function to use.

```
PllKphi = 5e-3;     % Charge Pump output current
PllKvco = 100e6;    % VCO sensitivity
PllN    = 70;       % Prescaler ratio
PllR2   = 1.33e3;   % Loop filter resistance for second order response (ohms)
PllR3   = 1.7e4;    % Loop filter resistance for third order response (ohms)
PllR4   = 0;        % Loop filter resistance for fourth order response (ohms)
PllC1   = 1.31e-11; % Loop filter direct capacitance (F)
PllC2   = 1.44e-10; % Loop filter capacitance for second order response (F)
PllC3   = 9.41e-13; % Loop filter capacitance for third order response (F)
PllC4   = 0;        % Loop filter capacitance for fourth order response (F)
```

The script `prepareExpectedSpectra` computes the spectral density of the reference phase modulation and the VCO output phase noise. The script then also combines the result with the PLL loop response to obtain the phase noise spectral density at the output of the PLL. The reference phase modulation is a deterministic process for which the amplitude of the spectral components is expressed as a fraction of the carrier amplitude (dBc). In contrast, the VCO phase noise is a stochastic process for which the spectral density is expressed in dBc/Hz.

Define the input parameters for the `prepareExpectedSpectra` script such as reference input frequency, number of reference cycles per symbol of PRBS6 phase modulation data pattern, amplitude of a single spectral component of the PRBS6 reference phase modulation, resolution

bandwidth to evaluate spectra with phase noise, frequency offset vector, and phase noise spectral densities at specified frequency offset.

```
PllAddPhaseNoise = 'on'; % Enable VCO phase noise
PllFoffset    = [30e3 100e3 1e6 3e6 10e6]; % VCO offset frequencies (Hz)
PllPhaseNoise = [-56 -106 -132 -143 -152]; % VCO output phase noise (dBc/Hz)
CfgSelectRefMod = '0';   % Enable reference phase noise modulation
CfgRef  = 30e6;          % Reference frequency
CfgCyclesPerSymbol = 2;  % Reference cycles per PRBS6 modulation symbol
CfgModLevel   = -110;    % Reference phase modulation level (dBc/Hz)
CfgResBandwidth = 100e3; % PLL Testbench resolution bandwidth
CfgTargetSpectrum = [100e3 -145;1e6 -135;3e6 -140;10e6 -150];
                         % PLL Testbench target phase noise (dBc/Hz)
```

If the variable `WorkspaceFile` exists and points to a file that can be loaded, load that file into the workspace. To use a configuration supplied with this example, set the value of `WorkspaceFile` to the name of the file for that configuration.

```
if exist('WorkspaceFile','var') && exist(WorkspaceFile,'file')
    load(WorkspaceFile);
end
```

Analyze the PLL control loop using the `getPllLoopResponse` function. The outputs of this function are:

- `LoopFrequency` - The frequency points at which the expected responses is calculated.
- `LoopZofs` - The transfer impedance of the loop filter as a function of the frequency.
- `LoopGofs` - The loop gain from the output of the prescaler to the input of the VCO, including the VCO voltage sensitivity. Note that the prescaler ratio is not included in this output, but is included in the closed loop transfer functions.
- `LoopHofs` - The closed loop phase transfer function fromt the PLL reference input to the PLL output.
- `LoopEofs` - The closed loop phase error transfer function with respect to the VCO output.

```
[LoopFrequency, LoopZofs, LoopGofs, LoopHofs, LoopEofs, LoopPhStep] = ...
    getPllLoopResponse([0,PllR2,PllR3,PllR4],[PllC1,PllC2,PllC3,PllC4], ...
                       PllKphi,PllKvco,PllN);
```

Organize the computation of expected spectral density to place spectral components directly into the frequency bins.

The workspace variables for the expected spectral density are:

- `ExpInputFrequency` - A vector of frequencies for which the expected input spectrum is calculated (Hz).
- `ExpInputSpectrum` - A vector of expected spectrum values at the reference input to the PLL (dBm into a one ohm load at a resolution bandwidth of `CfgResBandwidth`).
- `ExpOutputFrequency` - A vector of frequencies for which the expected output spectrum is calculated (Hz).
- `ExpOutputSpectrum` - A vector of expected spectrum values at the PLL output (dBm into a one ohm load at a resolution bandwidth of `CfgResBandwidth`).

```
prepareExpectedSpectra;
```

Examine Expected Results

In this section, examine the expected PLL behavior.

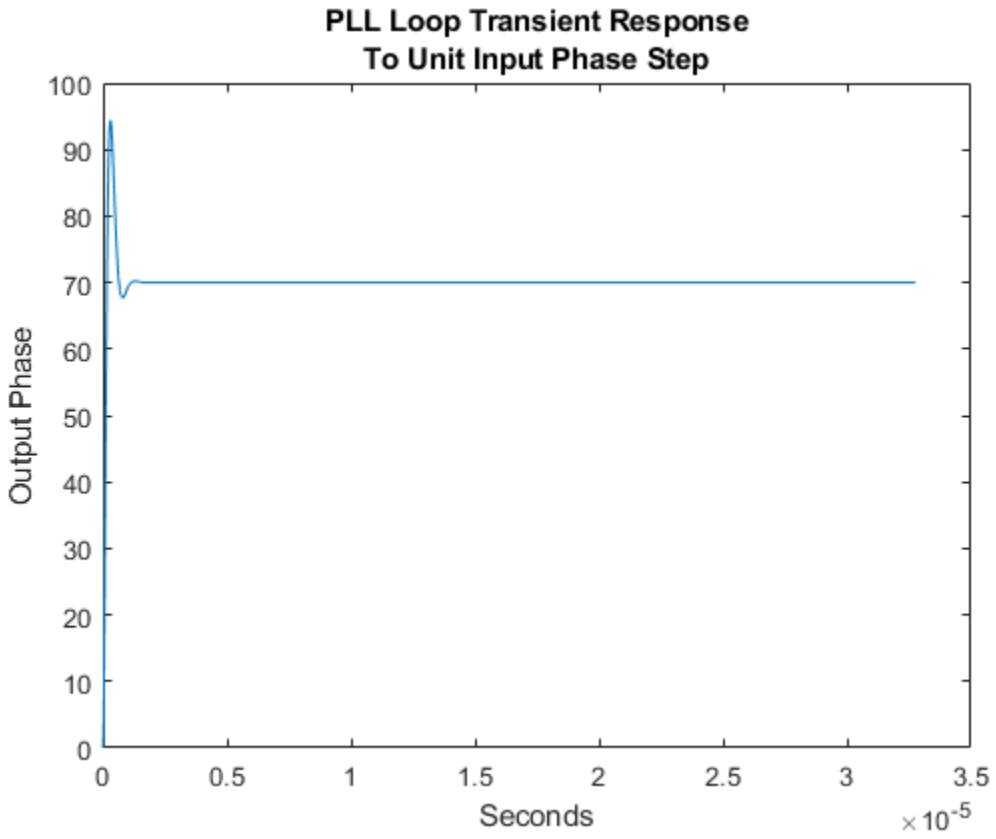
Start by looking at the closed loop response of the PLL. Consider three fundamental responses: transient response, transfer function, and error function.

Transient response

In most PLL applications, the loop transient response is most important when the loop is initially acquiring phase lock. For very small initial frequency offsets or for relatively high closed loop bandwidth, the transient response predicts the loop acquisition time reasonably accurately. However, in many cases, the loop transient response only represents the response during phase acquisition, after frequency acquisition has already occurred. This example only addresses the phase acquisition time.

Plot the loop transient response of the PLL.

```
figure(1);
plot(LoopPhStep.Time,LoopPhStep.Data);
title({'PLL Loop Transient Response';'To Unit Input Phase Step'});
xlabel('Seconds');
ylabel('Output Phase');
```

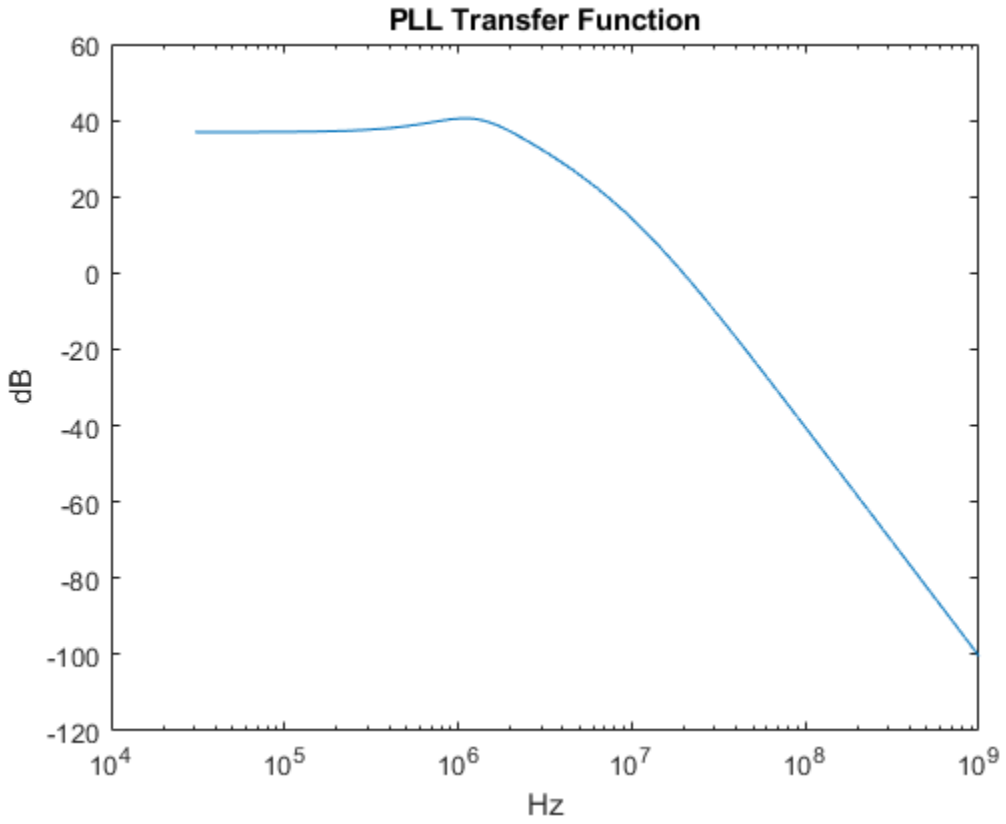


Transfer function

The transfer function from the reference input to the PLL output can be important either when the PLL is expected to track a modulated input very accurately or when it is expected to filter out noise from a noisy input.

Create a log-log plot of the PLL transfer function.

```
figure(2);
semilogx(LoopFrequency,20*log10(abs(LoopHofs)));
title('PLL Transfer Function');
xlabel('Hz');
ylabel('dB');
```

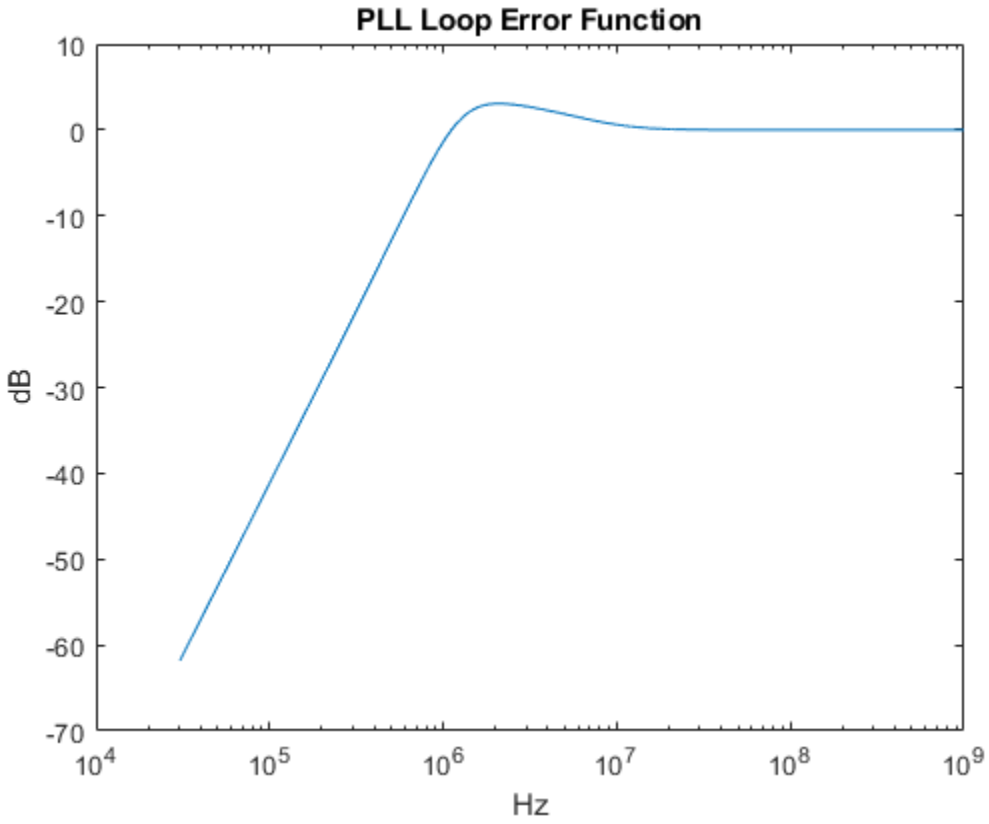


Error function

The PLL control loop is expected to reject phase noise generated inside the PLL itself, such as VCO phase noise. The transfer function between the VCO phase noise and the PLL output is sometimes called the loop error function. Examining this transfer function helps compensate the loop error before it is applied to another noise source.

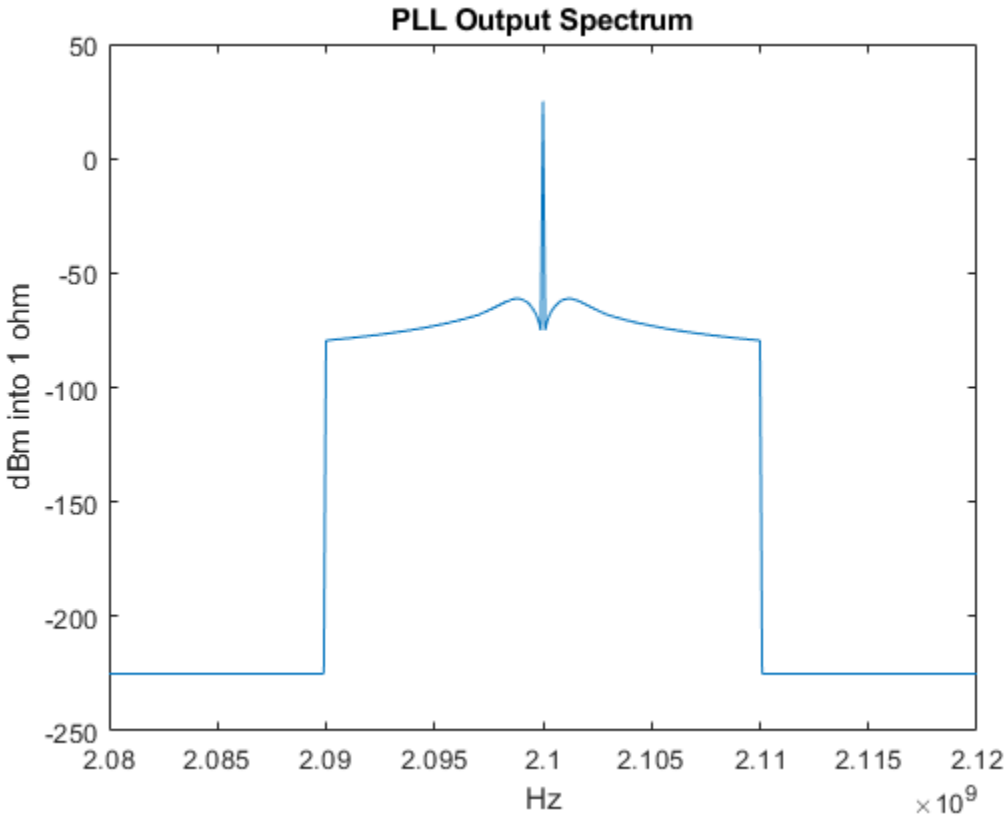
Create a log-log plot of the PLL loop error function.

```
figure(3);
semilogx(LoopFrequency,20*log10(abs(LoopEofs)));
title('PLL Loop Error Function');
xlabel('Hz');
ylabel('dB');
```



Plot the total expected phase noise at the PLL output.

```
figure(4);
plot(ExpOutputFrequency,ExpOutputSpectrum);
title('PLL Output Spectrum');
xlabel('Hz');
ylabel('dBm into 1 ohm');
xlim([PllN*CfgRef-2*PllFoffset(end), PllN*CfgRef+2*PllFoffset(end)]);
```

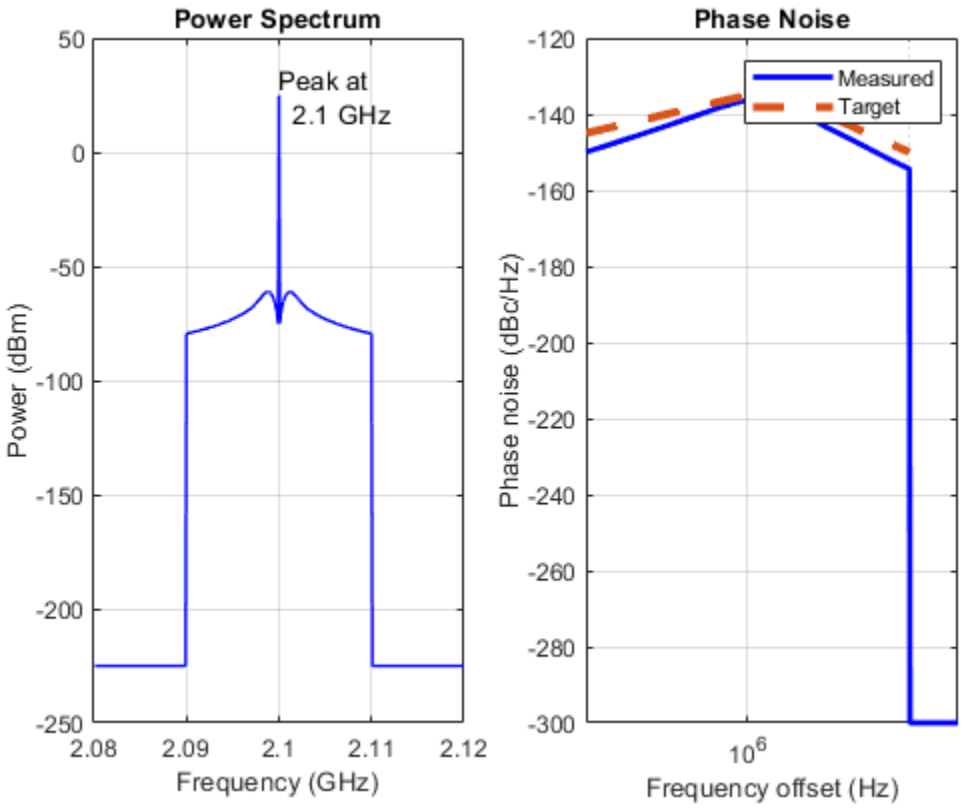


The `phaseNoiseMeasure` function is a callback function used by the PLL Testbench. It displays the target output phase noise spectral density along with the simulated or expected phase noise spectral density.

To compare the expected output phase noise to a specific design target, define the workspace variable `CfgTargetSpectrum`. `CfgTargetSpectrum` consists of two column arrays that specify the target phase noise at the output of the PLL. The first column specifies the frequency offsets in Hz and the second column specifies the corresponding phase noise spectral density if dBc/Hz. The PLL Testbench uses the callback function `phaseNoiseMeasure` to display the expected and simulated phase noise spectral density.

View the expected PLL output phase noise in units of dBc/Hz.

```
[~] = phaseNoiseMeasure(ExpOutputFrequency,ExpOutputSpectrum,...
    CfgResBandwidth, CfgTargetSpectrum(:,1).','on','5', ...
    CfgTargetSpectrum(:,2).');
```

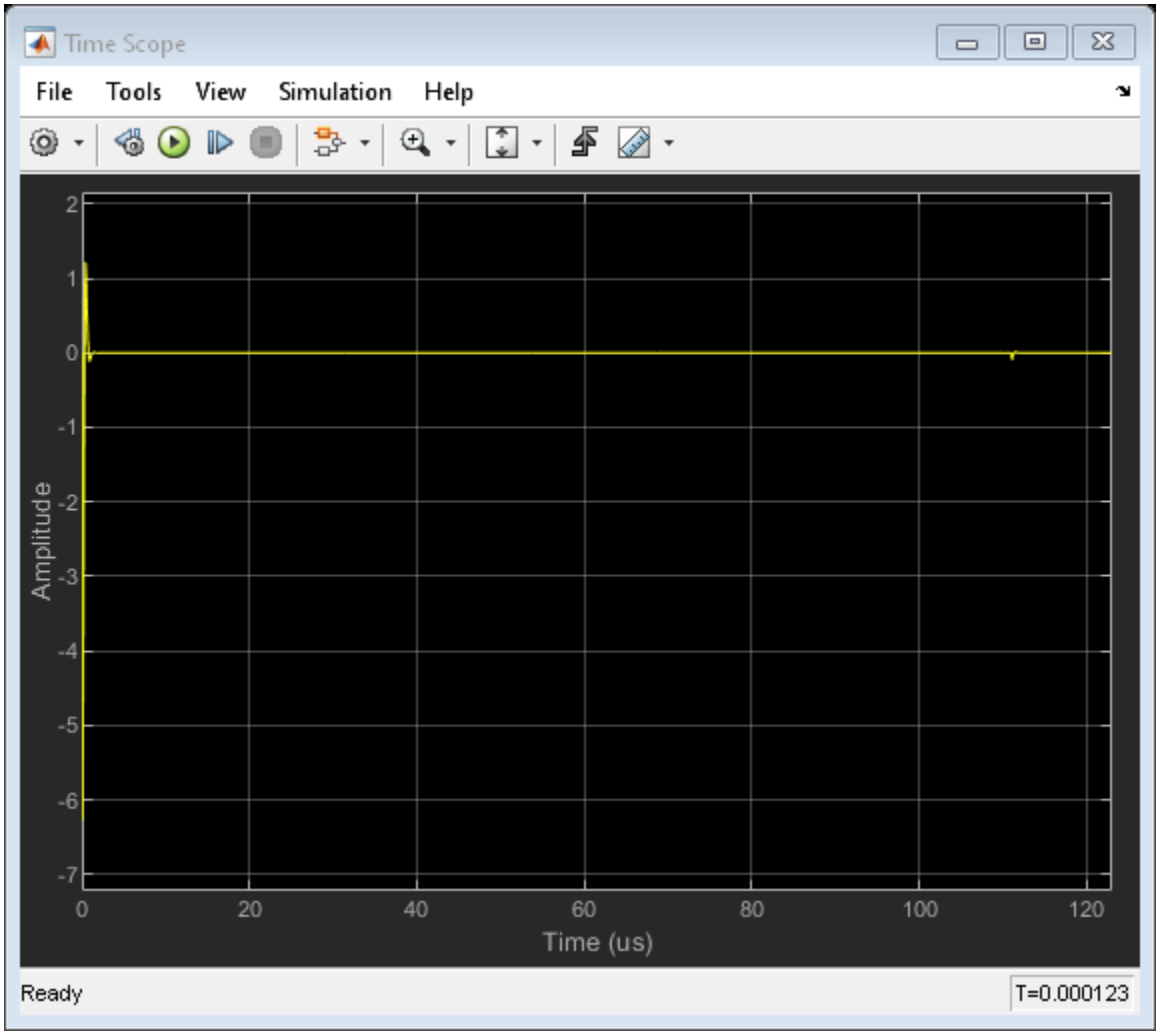


If the loop response is not satisfactory, you might look at more detailed results such as the loop filter transimpedance (`LoopZofs`) to get additional insights that could help you improve the PLL design.

Run the Simulation

While the configuration process is straightforward, there are many parameters to be configured. Use the `configureExamplePLL` script to configure the PLL Simulink® model, then execute the model.

```
configureExamplePll;
SimOut = sim('PllPhaseNoiseExample');
```



To make the plotting easier, transform the results into the following four workspace variables:

- `SimInputFrequency` - A vector of frequencies for which the input spectrum of the simulation is stored (Hz).
- `SimInputSpectrum` - A vector of spectrum values at the reference input to the PLL (dBm into a one ohm load at a resolution bandwidth of `CfgResBandwidth`).
- `SimOutputFrequency` - A vector of frequencies for which the output spectrum of the simulation is calculated (Hz).
- `SimOutputSpectrum` - A vector of simulated spectrum values at the PLL output (dBm into a one ohm load at a resolution bandwidth of `CfgResBandwidth`).

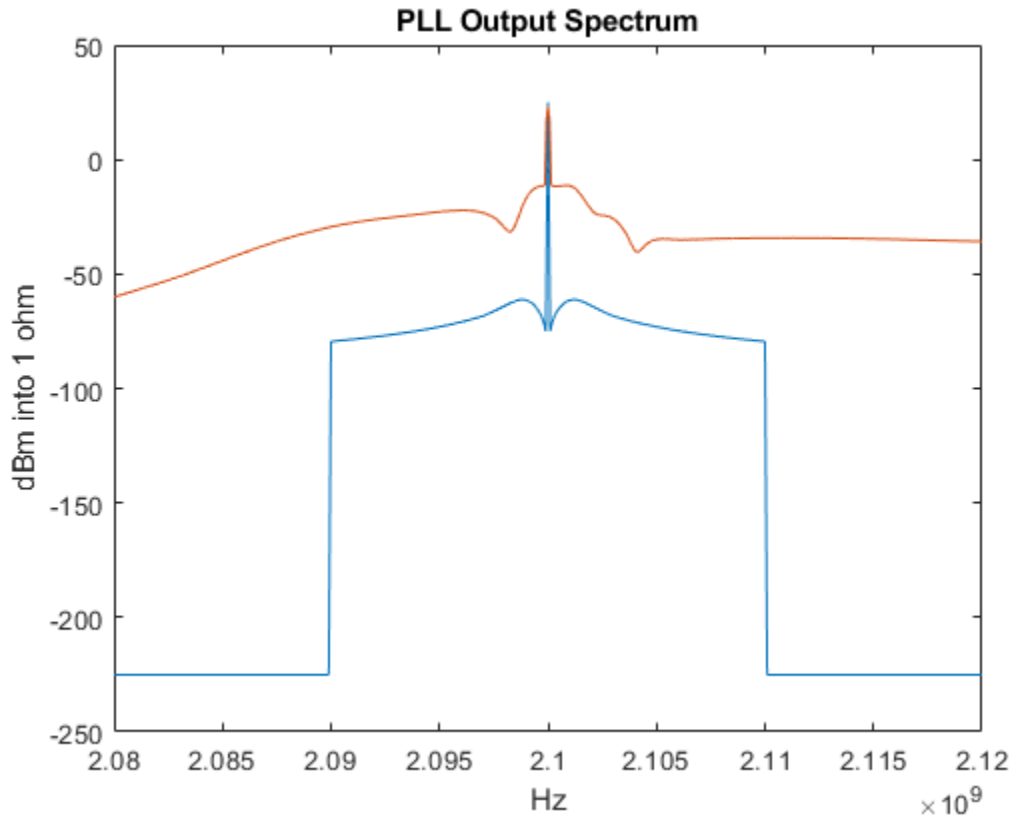
Make the results easier to plot from the workspace.

```
SimInputFrequency = reshape(SimOut.InputFrequency.Data(:,end),1,[]);
SimInputSpectrum = reshape(SimOut.InputSpectrum.Data(:,end),1,[]);
SimOutputFrequency = reshape(SimOut.OutputFrequency.Data(:,end),1,[]);
SimOutputSpectrum = reshape(SimOut.OutputSpectrum.Data(:,end),1,[]);
```

Compare Expected and Simulated Results

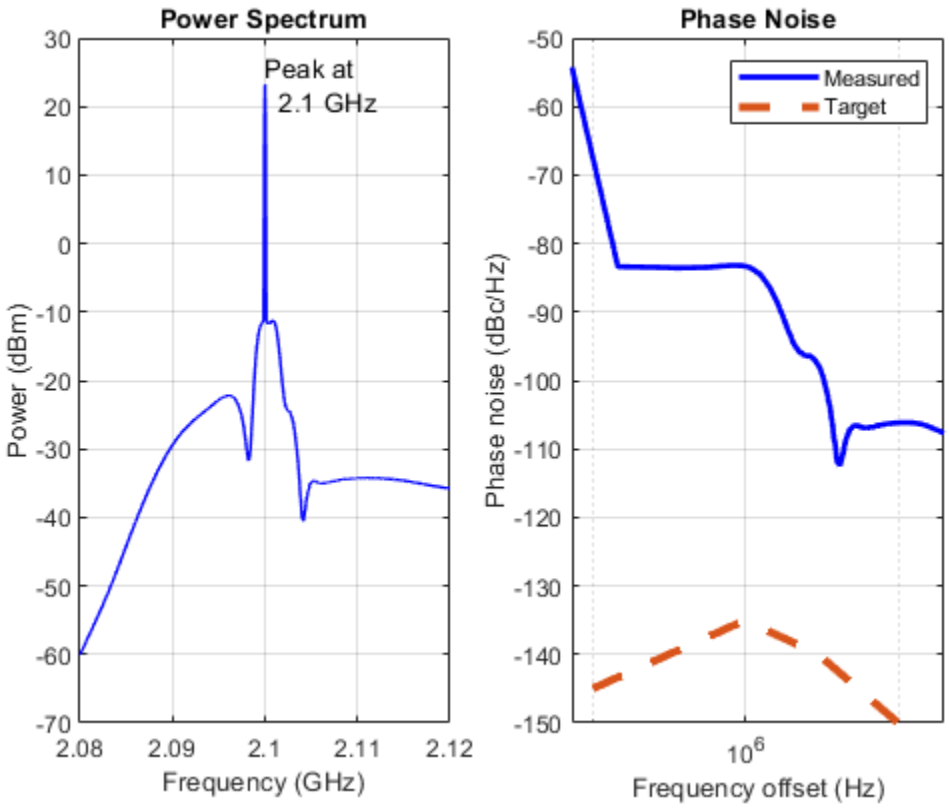
Compare expected and simulated phase noise at the PLL output.

```
figure(4);
plot(ExpOutputFrequency,ExpOutputSpectrum);
title('PLL Output Spectrum');
xlabel('Hz');
ylabel('dBm into 1 ohm');
xlim([PllN*CfgRef-2*PllFoffset(end), PllN*CfgRef+2*PllFoffset(end)]);
hold on;
plot(SimOutputFrequency,SimOutputSpectrum);
hold off;
```



View the expected PLL output phase noise in units of dBc/Hz.

```
[~] = phaseNoiseMeasure(SimOutputFrequency,SimOutputSpectrum,...
    SimOutputFrequency(2)-SimOutputFrequency(1),...
    CfgTargetSpectrum(:,1).','on','5',CfgTargetSpectrum(:,2).');
```



Save the Configuration and Results

Save the entire workspace, including the system configuration in its current state and the results, to a file. If the `WorkspaceFile` variable already exists, you can resave the current state to that file. To save to a new file, change the value of `WorkspaceFile`.

```
if exist('WorkspaceFile','var')
    save(WorkspaceFile);
end
```

See Also

Integer N PLL with Single Modulus Prescaler | PLL Testbench | Variable Pulse Delay

More About

- “PLL Design and Verification Using Data Sheet Specifications” on page 1-13

PLL Design and Verification Using Data Sheet Specifications

This example shows how to use Mixed-Signal Blockset™ to model a commercial off-the-shelf integer-N phase-locked loop (PLL) with dual modulus prescaler operating around 4 GHz. You can verify the PLL performance, including phase noise.

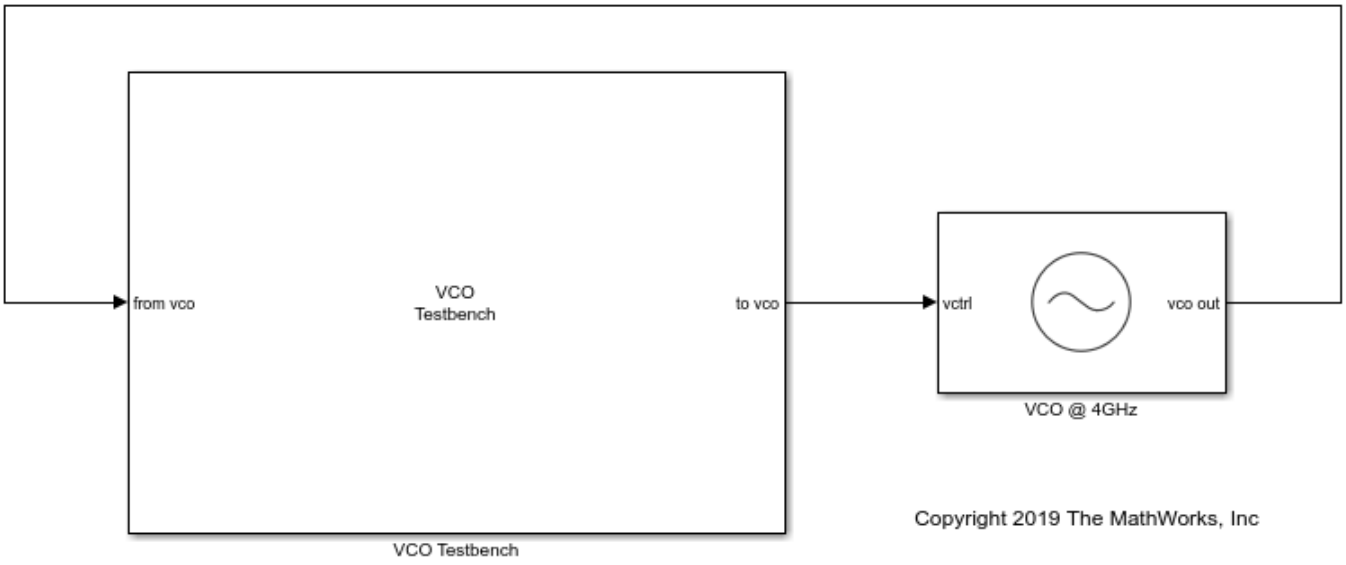
You can use PLL models to explore and design different loop filters, simulate different operating frequencies, determine different divider ratios, or assess the frequency synthesizer performance once embedded in a larger system. For example, Mixed-Signal Blockset PLL models can be helpful if you need to integrate a PLL into a System on Chip or a System on Board.

For this example, use the data sheet of Skyworks SKY73134-11. This is a wideband PLL Frequency Synthesizer operating between 350 MHz and 6.2 GHz.

Build VCO Model

Open the model `VCO_model`, which has VCO and VCO Testbench blocks from Mixed-Signal Blockset connected together.

```
model = 'VCO_model';
open_system(model);
```



Inspect the data sheet to identify the characteristics of the voltage controlled oscillator (VCO). VCO is the main component contributing to the phase noise of the PLL. Table 6 of the data sheet lists the typical VCO sensitivity (*Kvco*) as 20 MHz/V. Set the **Voltage Sensitivity** of the VCO block to `20e6` Hz/V. The data sheet does not provide the free running frequency (*Fo*) of the VCO, so you can set it to an arbitrary value close to the operting frequency. In this case, set **Free running frequency** to `3.9e9` Hz.

For this example, simulate the PLL behavior when the PLL locks around 4 GHz. Table 7 of the data sheet lists the VCO phase noise profile as: around 4 GHz is,

- −105 dBc/Hz at 100 kHz
- −131 dBc/Hz at 1 MHz
- −142 dBc/Hz at 3 MHz
- −152 dBc/Hz at 10 MHz

Use the phase noise profile to add phase noise impairment to the VCO. To speed up simulation, ignore the phase noise data points at lower frequency offsets. Simulation takes longer to capture the phase noise profile close to the carrier.

Measure VCO Phase Noise

The testbench provides the control voltage (*Vctrl*) stimulus to the VCO and measures the VCO phase noise. The output frequency (*F*) of the VCO is :

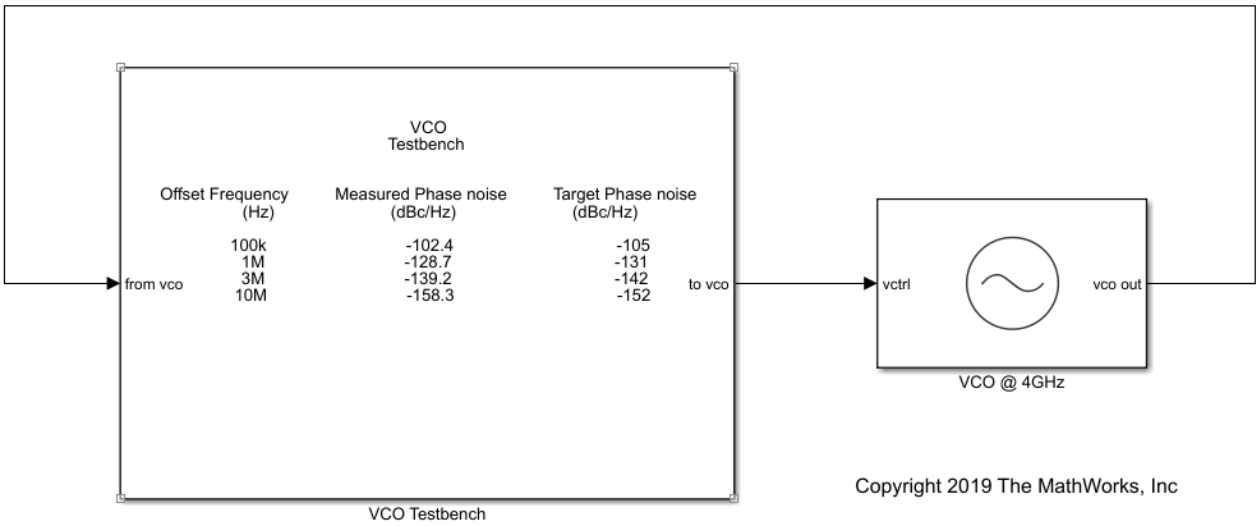
$$F = Kvco * Vctrl + Fo$$

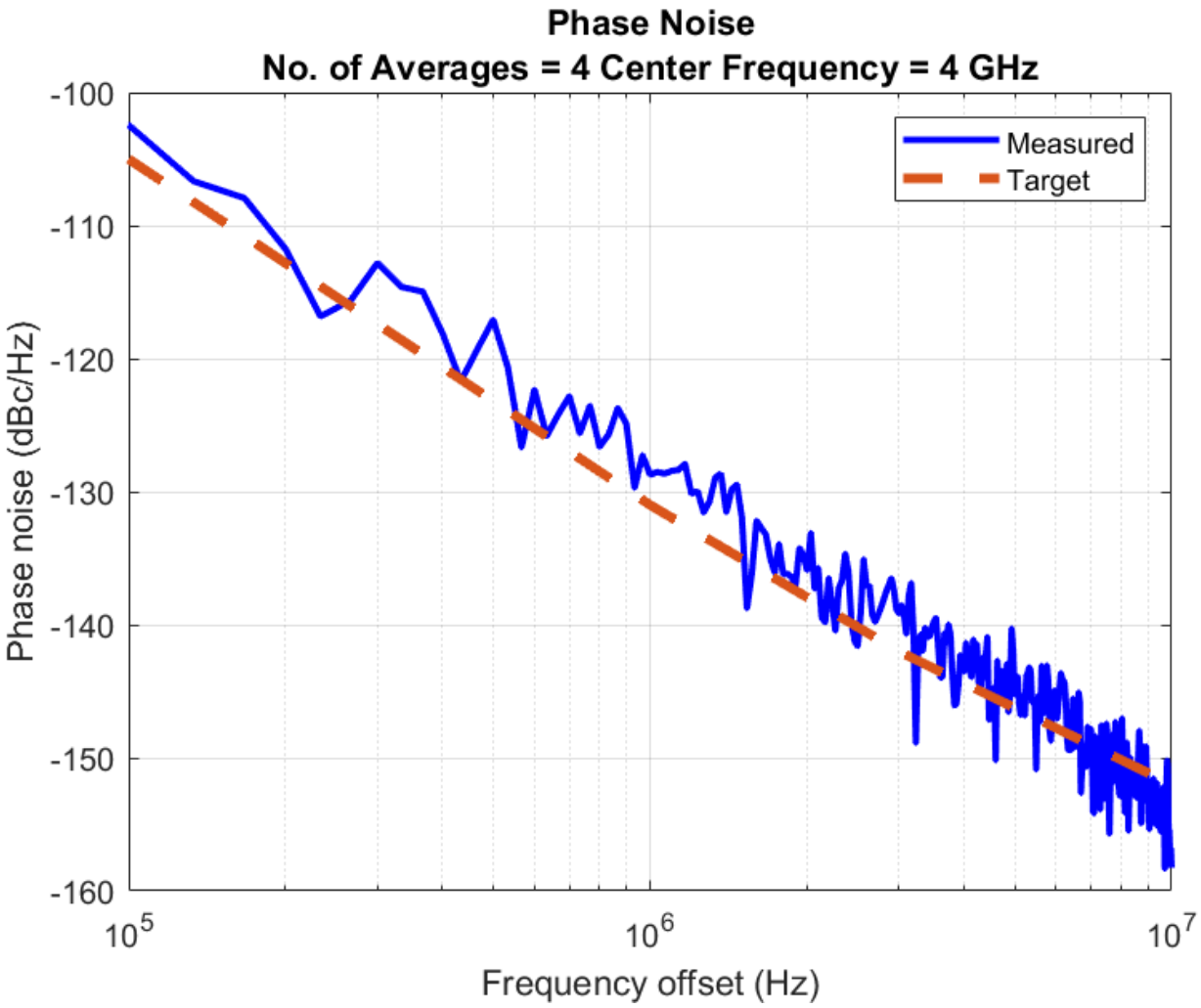
Substituting the values for the output frequency, the free running frequency, and the voltage sensitivity, the equation gives a control voltage of be 5 V.

Click on **Autofill setup parameters** and **Autofill target metric** buttons to automatically populate the measurement parameters for both setup and target metric. The target metric provides the anticipated phase noise profile for comparison with the simulation results. The testbench setup properties define the signal sampling frequency and the measurement resolution bandwidth.

To speed up simulation time, reduce **No. of spectral averages** to 4.

Run the simulation. Verify that the VCO model reproduces the phase noise profile specified in the target metric.



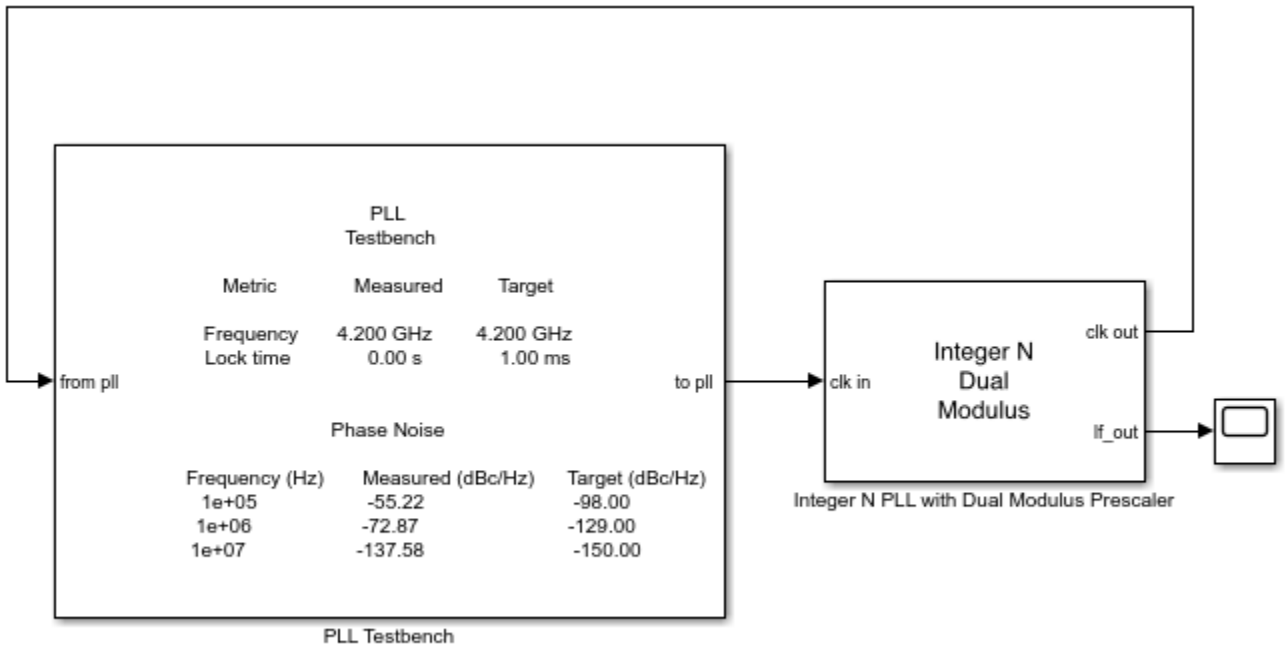


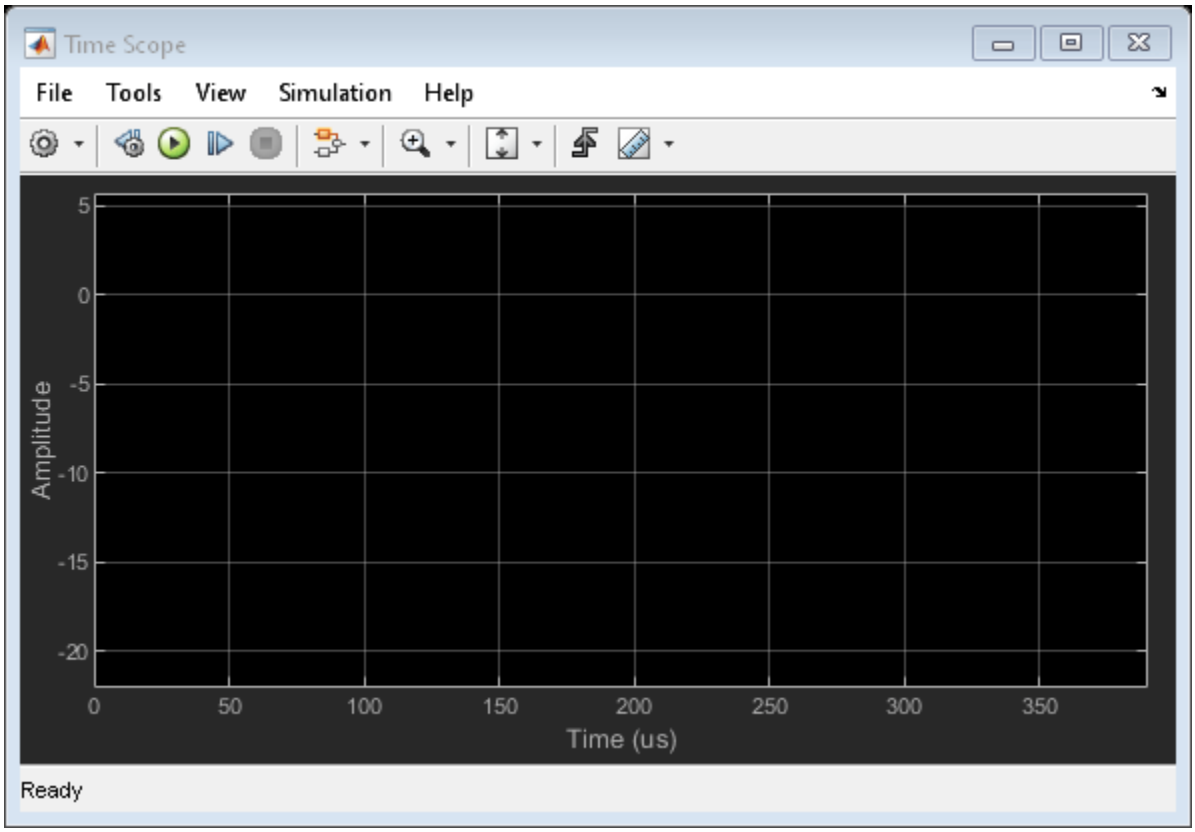
You can now proceed with modeling the complete PLL system and verify its phase noise performance.

Build PLL Model

From the **Features** section of the data sheet, see that this PLL is of type integer-N and it uses a dual modulus prescaler. The data sheet also provides the settings for the prescaler. Hence, use the **Integer-N PLL with Dual Modulus Prescaler** from Mixed-Signal Blockset.

```
bdclose(model);
model = 'PLL_model';
open_system(model);
```





Open the mask of the PLL block and provide the specs for each of the subsystems:

- **Phase Frequency Detector** — For the phase frequency detector (PFD), the data sheet does not provide any detail. Leave the deadband compensation to its default value.
- **Charge Pump** — For the charge pump, the data sheet provides the typical value for the current of 2.7 mA (data sheet, Table 5). Specify the same value in the PLL model.
- **VCO** — For the VCO, use the same specifications used in the **Build VCO Model** section. The sensitivity is equal to `20` MHz/V. Specify a free running frequency that is close to the final locking frequency, for example `4.18` GHz. Include phase noise with the same noise profile that you just simulated.
- **Divider** — For the PLL to lock at 4.2 GHz when using a reference oscillator operating at 1.6 MHz (data sheet, Figure 17), the divider ratio is equal to:

$$\frac{4.2\text{e}9}{1.6\text{e}6} = 2625$$

To achieve the effective divider ratio of 2625, set the **Prescaler divider value** to `16`, **Swallow counter value** to `17`, and **Program counter value** to `163`.

- **Loop Filter** — The data sheet provides the recommended loop filter component values in Figure 22. Manually input these values to implement the filter. Set Loop filter type to 3rd order passive. Also set the loop filter component values:

C1 = `100e-12` F

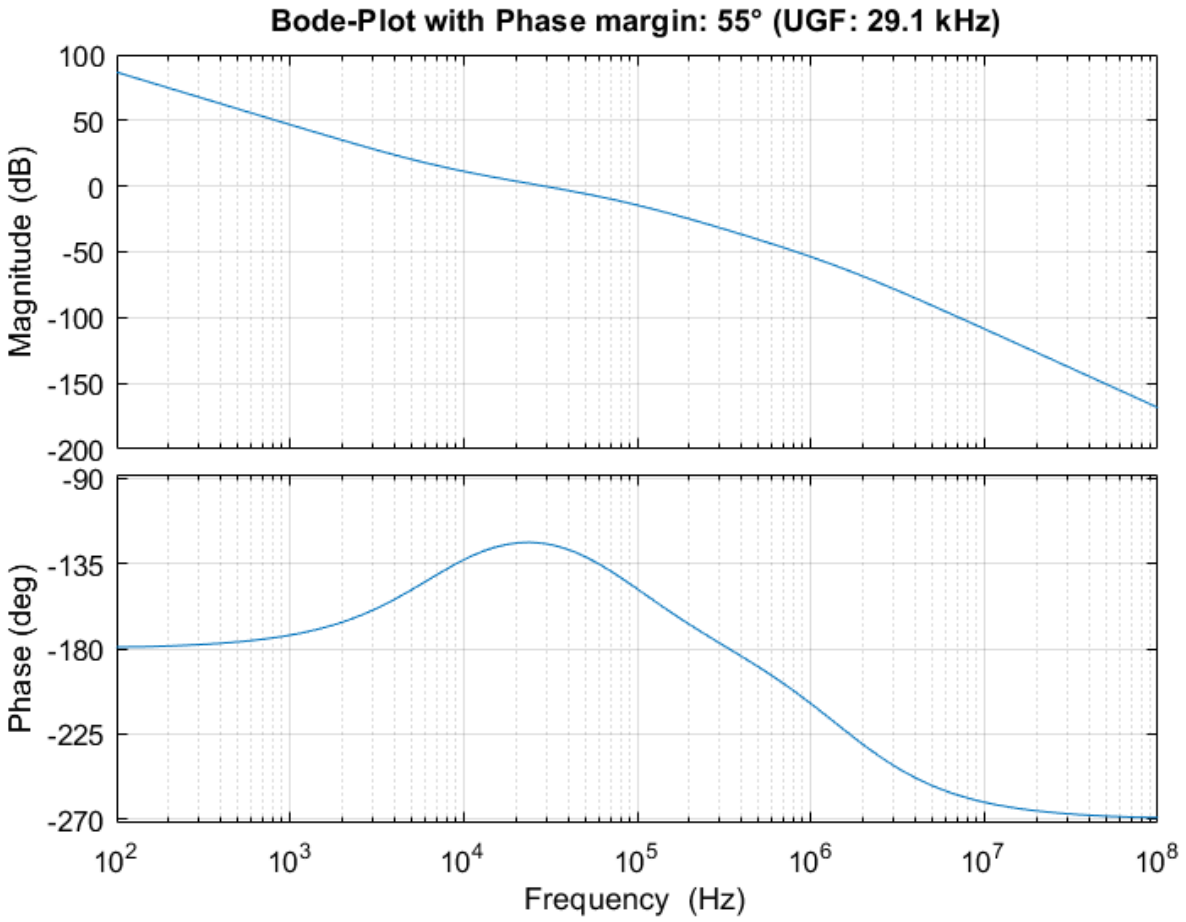
C2 = `2.2e-9` F

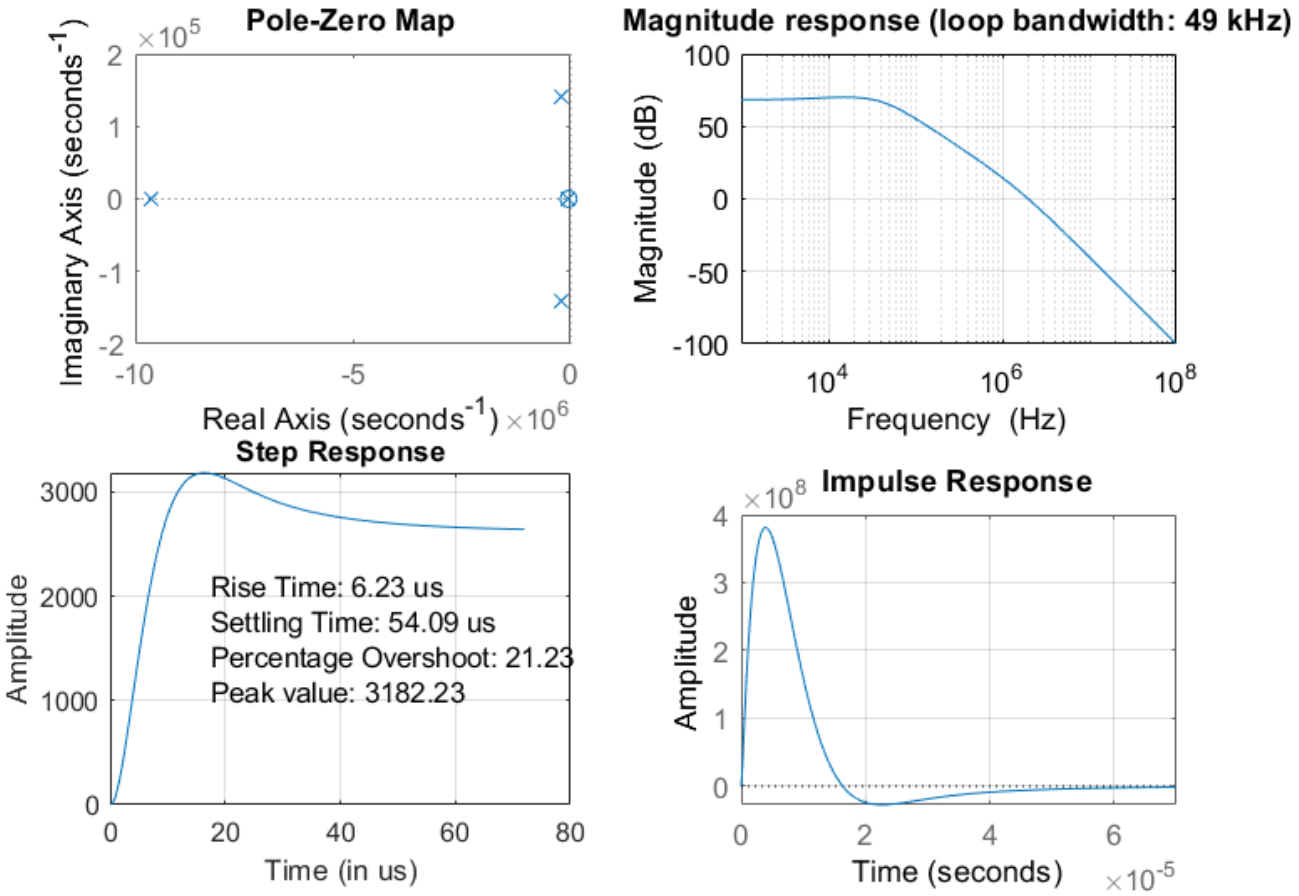
C3 = `100e-12` F

R2 = `10e3` Ω

R3 = `2.2e3` Ω

Verify the closed and open loop performance of the PLL with this implementation of the loop filter. The resulting phase margin is 55 degrees. The phase margin along with the pole zero locations confirm the stability of the PLL.


Bode-Plot with Phase margin: 55° (UGF: 29.1 kHz)
Magnitude (dB)
100
50
0
-50
-100
-150
-200
Phase (deg)
-90
-135
-180
-225
-270
10^2
10^3
10^4
10^5
10^6
10^7
10^8
Frequency (Hz)



Use this setup to design different loop filters and verify your design. For example, use this PLL over a different operating frequency.

Finally, to verify the PLL locking behavior in the time domain, probe and plot the output signal of the loop filter.

Measure PLL Performance

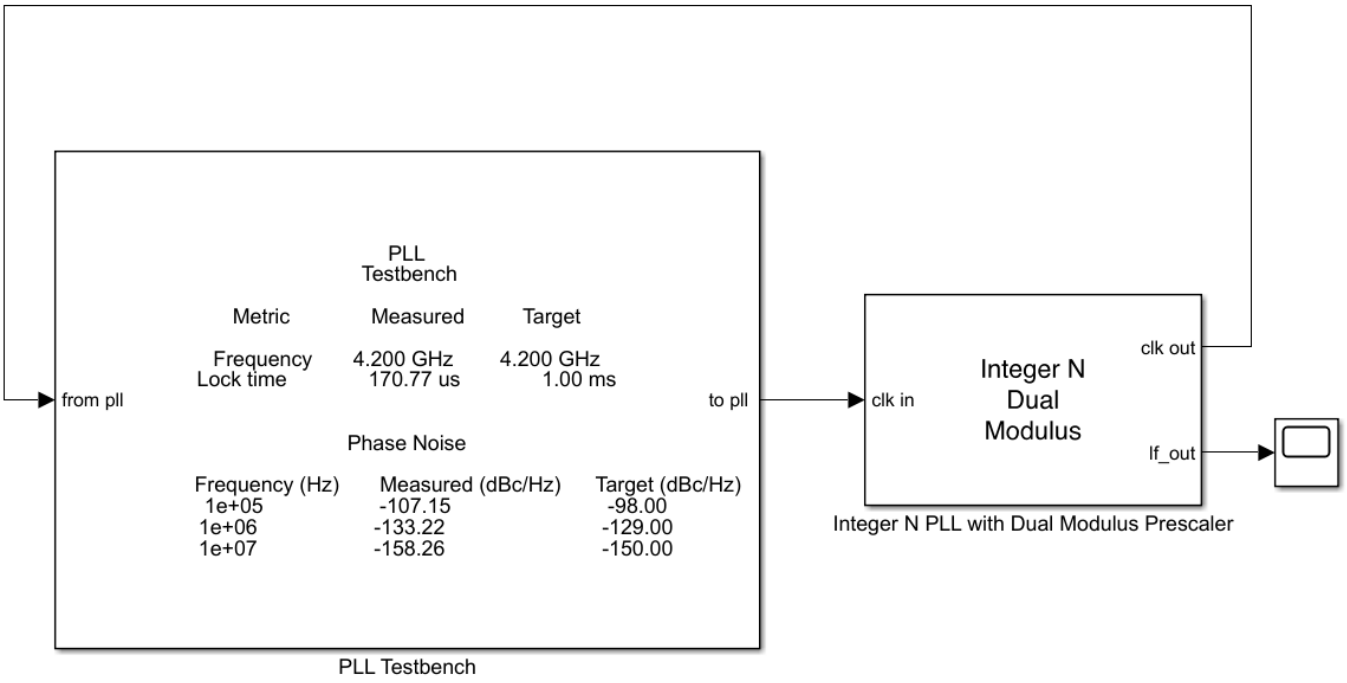
Connect the PLL to a Mixed-Signal Blockset PLL Testbench to validate its performance. The testbench defines the stimuli used to test the PLL. In this case, use a square waveform with 1.6 MHz frequency.

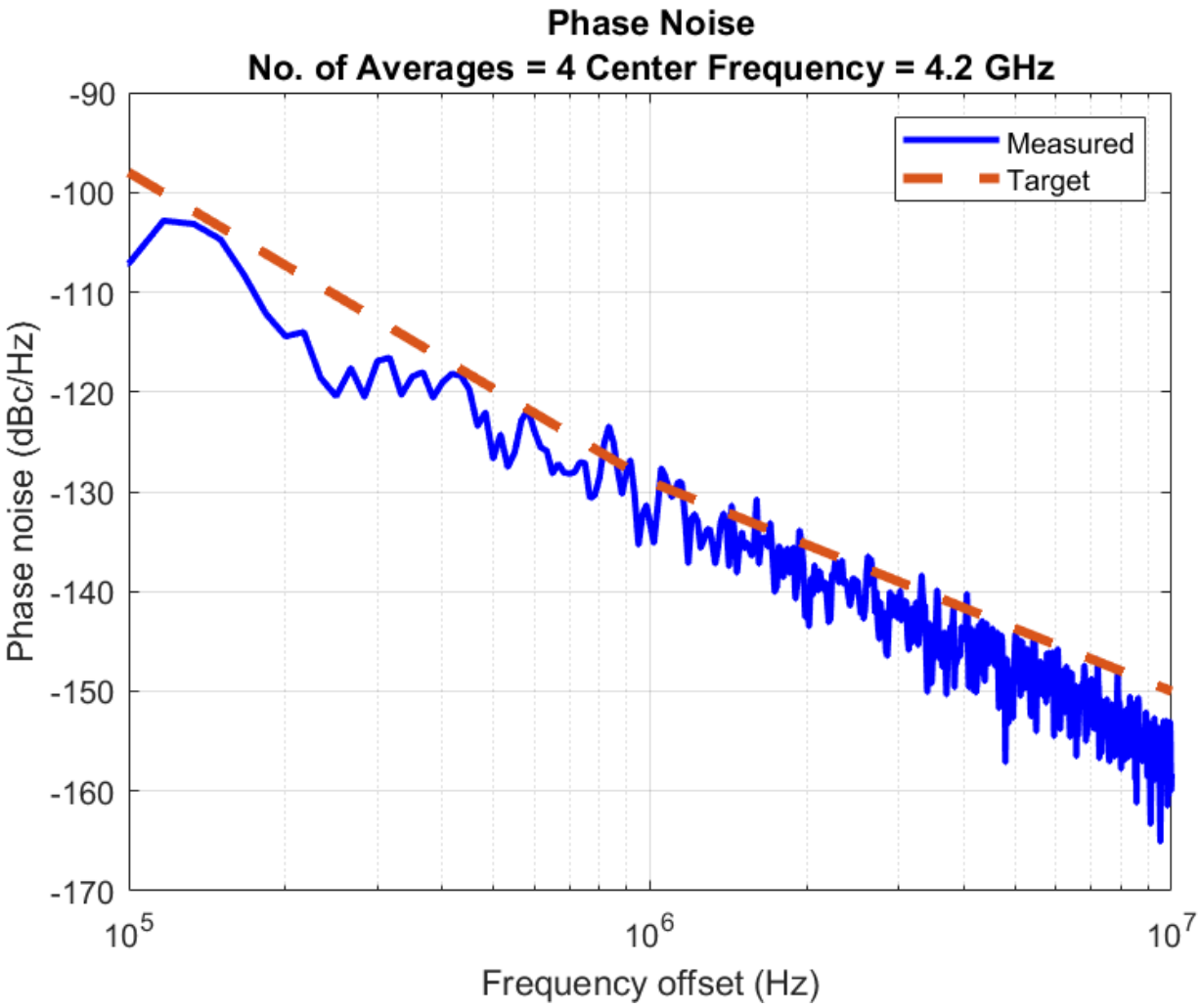
Set up the testbench to measure operating frequency, lock time, and phase noise. The data sheet specifies a 1 ms lock time with 1 ppm frequency error, that is to say 4.2 kHz (Table 5). Provide the same error tolerance. For measuring the phase noise, use a configuration similar to the one used for the VCO, but reduce the resolution bandwidth for higher accuracy.

For the target metric, refer to the data sheet for both the lock time (Table 5) and the closed loop phase noise measurements performed on the evaluation board (Figure 17). The phase noise profile measured at 4.2 GHz is:

- −98 dBc/Hz at 100 kHz
- −129 dBc/Hz at 1 MHz
- −150 dBc/Hz at 10 MHz

With this setup, run the PLL simulation and verify the performance.





In this case, the simulation takes a few minutes. To speed up simulation time, you can relax the settings for measuring the phase noise. For example, use a larger resolution bandwidth or reduce the number of spectral averages.

The simulation results are in excellent agreement with the phase noise measurements reported in the data sheet.

Reference

Skyworks SKY73134-11

See Also

VCO | VCO Testbench | Integer N PLL with Dual Modulus Prescaler | PLL Testbench

More About

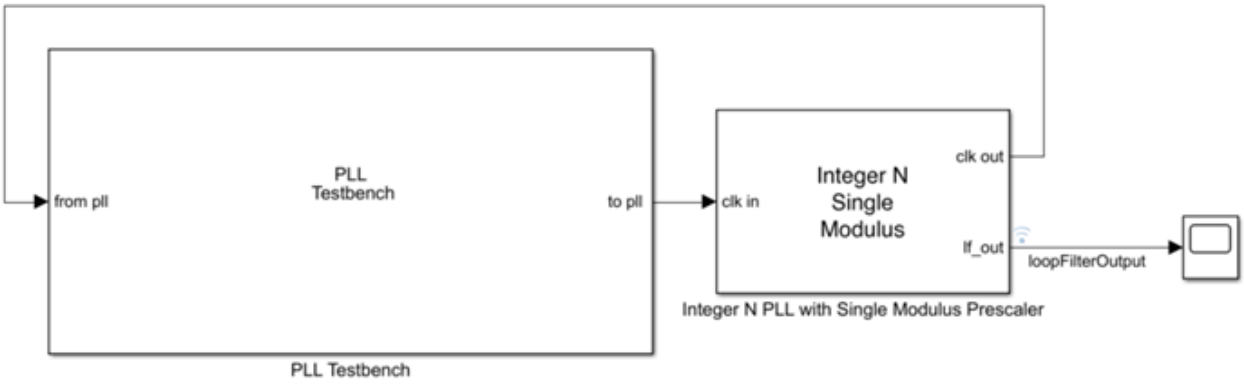
- “Design and Evaluate Simple PLL Model”
- “Phase Noise at PLL Output” on page 1-2

Tune Phase-Locked Loop Using Loop-Shaping Design

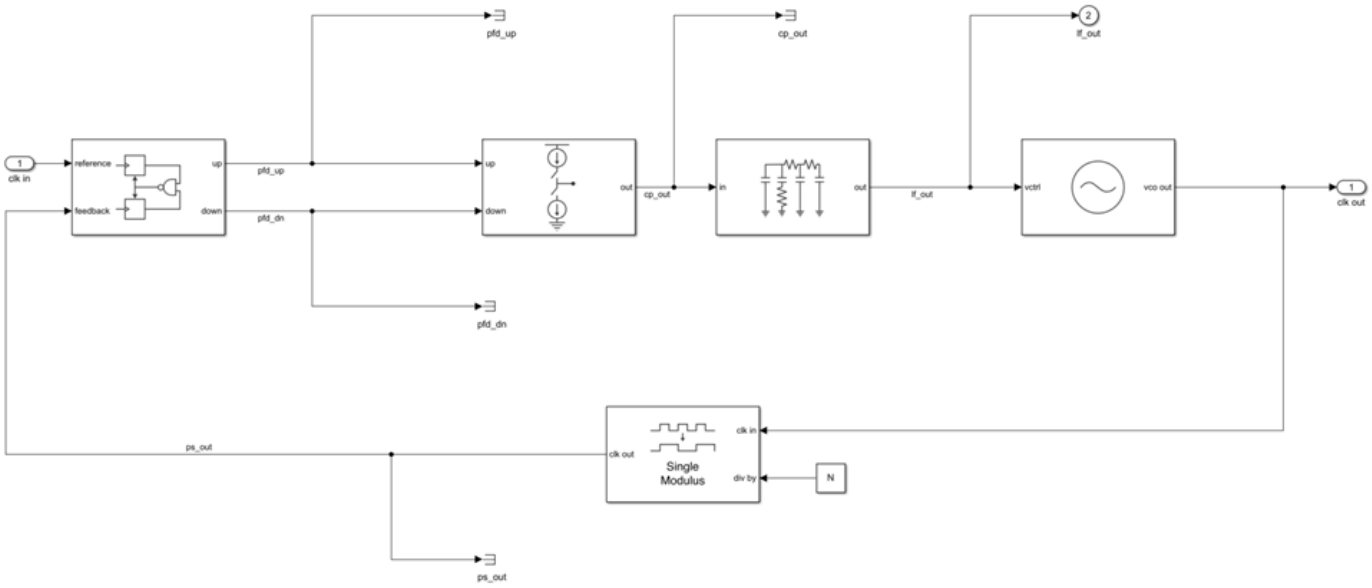
This example shows how to tune the components of a passive loop filter to improve the loop bandwidth of a phase-locked loop (PLL) system. To obtain a desired loop frequency response, this example computes the loop filter parameters using the fixed-structure tuning methods provided in the Control System Toolbox™ software. The PLL system is modeled using a reference architecture block from the Mixed-Signal Blockset™ library.

Introduction

A PLL is a closed-loop system that produces an output signal whose phase depends on the phase of its input signal. The following diagram shows a simple model with a PLL reference architecture block (Integer N PLL with Single Modulus Prescaler) and a PLL Testbench block.



The closed-loop architecture inside the PLL block consists of a phase-frequency detector (PFD), charge pump, loop filter, voltage controlled oscillator (VCO), and prescaler.



The Mixed-Signal Blockset library provides multiple reference architecture blocks to design and simulate PLL systems in Simulink®. You can tune the components of the Loop Filter block, which is a passive filter, to get the desired open-loop bandwidth and phase margin.

Using the Control System Toolbox software, you can specify the shape of the desired loop response and tune the parameters of a fixed-structure controller to approximate that loop shape. For more information on specifying a desired loop shape, see “Loop Shape and Stability Margin Specifications” (Control System Toolbox). In the preceding PLL architecture model, the loop filter is defined as a fixed-order, fixed-structure controller. To achieve the target loop shape, the values of the resistances and capacitances of the loop filter are tuned. Doing so improves the open-loop bandwidth of the system and, as a result, reduces the measured lock time.

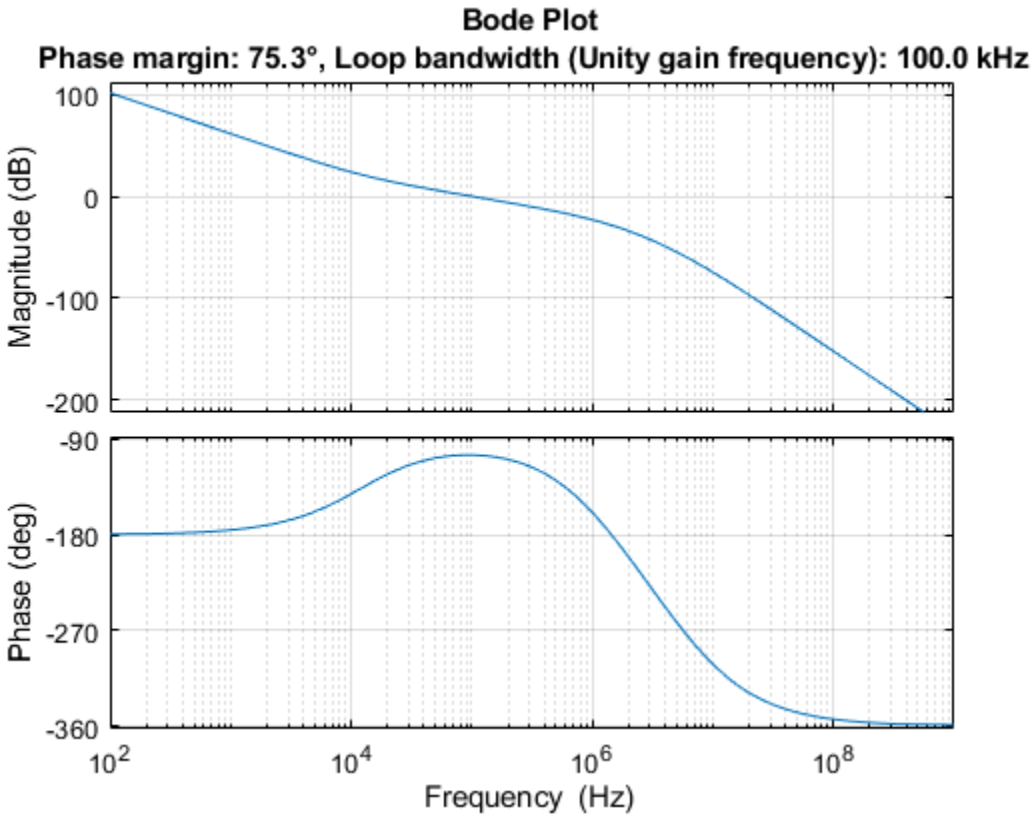
Set Up Phase-Locked Loop Model

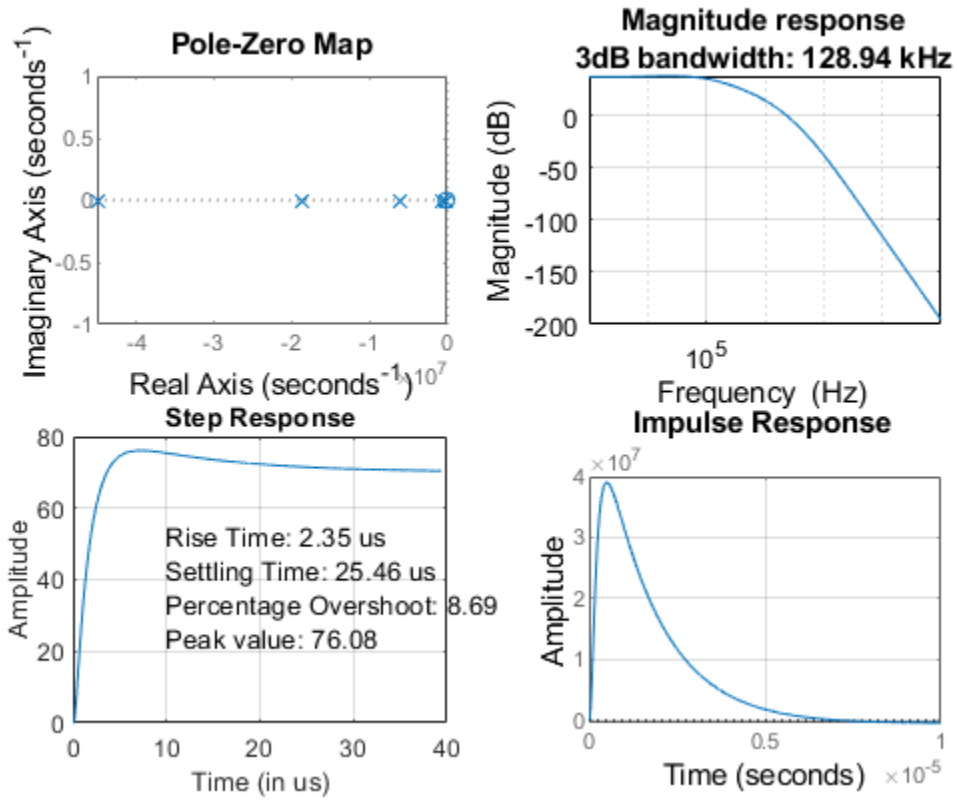
Open the model.

```
model = 'PLL_TuneLoopFilter';
open_system(model)
```

The PLL block uses the configuration specified in “Design and Evaluate Simple PLL Model” for the **PFD**, **Charge pump**, **VCO**, and **Prescalar** tabs in the block parameters. The **Loop Filter** tab specifies the type as a fourth-order filter, and sets the loop bandwidth to 100 kHz and phase margin to 60 degrees. The values for the resistances and capacitances are automatically computed.

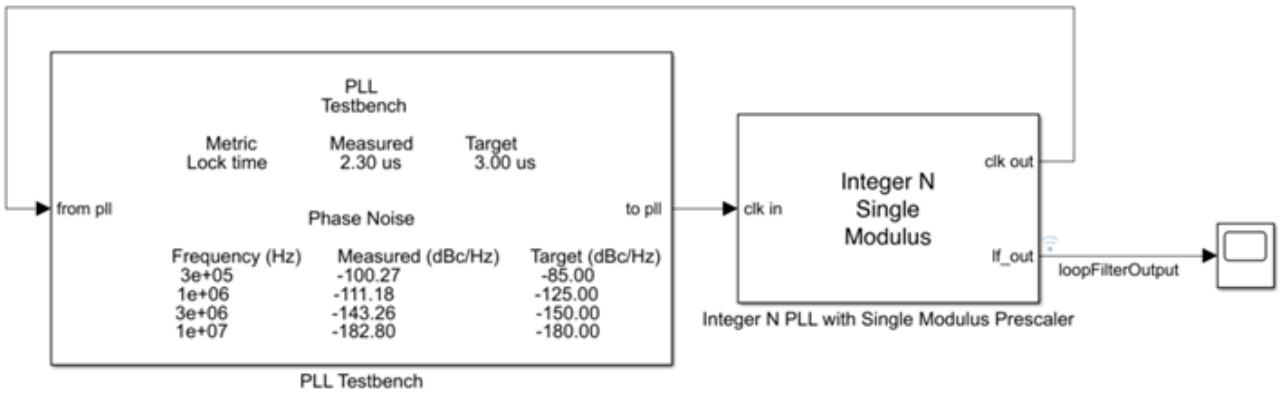
To observe the current loop dynamics of the PLL, in the block parameters, on the **Analysis** tab, select **Open Loop Analysis** and **Closed Loop Analysis**. The unity gain frequency is 100 kHz. The closed-loop system is stable and the 3-dB bandwidth is 128.94 kHz.

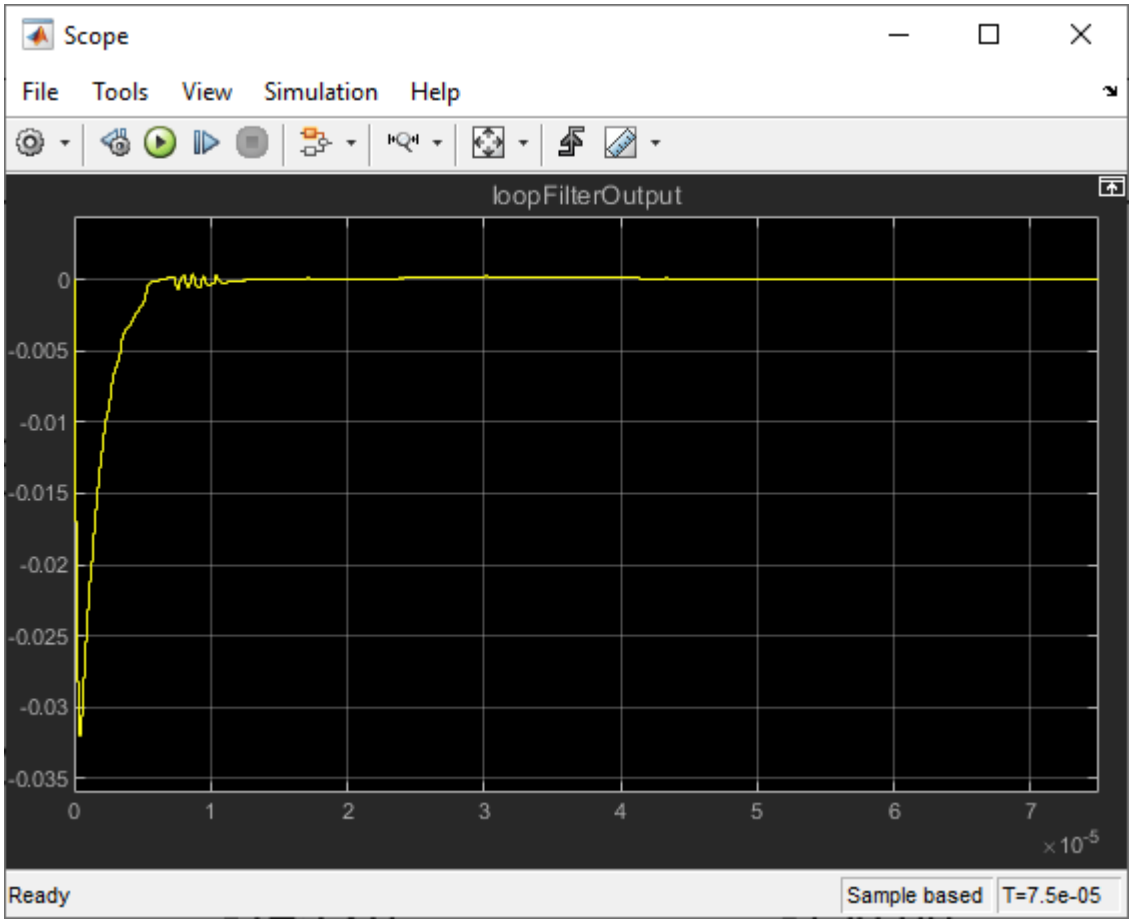

Bode Plot
Phase margin: 75.3°, Loop bandwidth (Unity gain frequency): 100.0 kHz
Magnitude (dB)
100
0
-100
-200
Phase (deg)
-90
-180
-270
-360
10^2
10^4
10^6
10^8
Frequency (Hz)

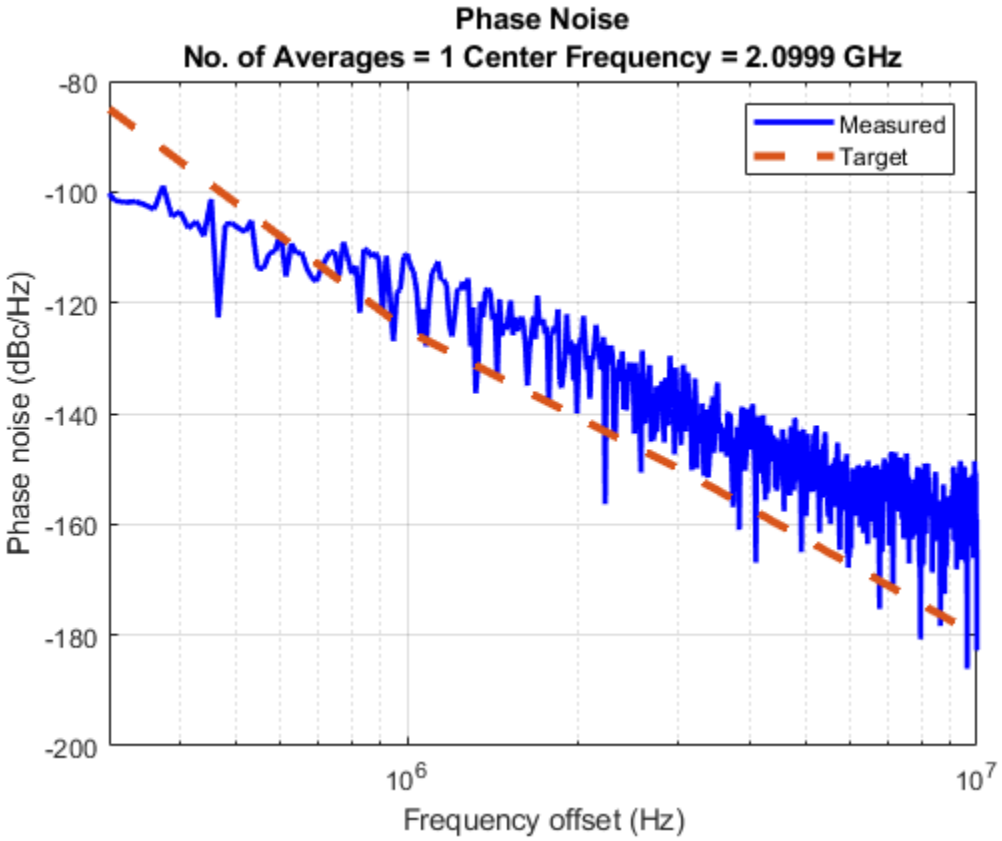


Simulate the model. The PLL Testbench block displays the PLL lock time and phase noise metrics. To plot and analyze the phase noise profile, in the PLL Testbench block parameters, on the **Stimulus** tab, select **Plot phase noise**. The measured lock time is 2.30 microseconds.

```
open_system([model,'/Scope'])
sim(model);
```




Scope
File
Tools
View
Simulation
Help
loopFilterOutput
0
-0.005
-0.01
-0.015
-0.02
-0.025
-0.03
-0.035
0
1
2
3
4
5
6
7
×10^-5
Ready
Sample based
T=7.5e-05



Define the PLL parameters needed to build the closed-loop system in MATLAB®.

```
PllKphi = 5e-3;     % Charge pump output current
PllKvco = 1e8;      % VCO sensitivity
PllN    = 70;       % Prescaler ratio

PllR2   = 88.3;     % Loop filter resistance for second-order response (ohms)
PllR3   = 253;      % Loop filter resistance for third-order response (ohms)
PllR4   = 642;      % Loop filter resistance for fourth-order response (ohms)
PllC1   = 8.13e-10; % Loop filter direct capacitance (F)
PllC2   = 1.48e-7;  % Loop filter capacitance for second-order response (F)
PllC3   = 1.59e-10; % Loop filter capacitance for third-order response (F)
PllC4   = 9.21e-11; % Loop filter capacitance for fourth-order response (F)
```

Build Custom Tunable System

To model the loop filter as a tunable element, first create tunable scalar real parameters (see `realp` (Control System Toolbox)) to represent each filter component. For each parameter, define the initial value and bounds. Also, specify whether the parameter is free to be tuned.

Use the current loop filter resistance and capacitance values as the initial numeric value of the tunable parameters.

```
% Resistances
R2 = realp('R2',PllR2);
R2.Minimum = 50;
R2.Maximum = 2000;
```

```
R2.Free = true;

R3 = realp('R3',PllR3);
R3.Minimum = 50;
R3.Maximum = 2000;
R3.Free = true;

R4 = realp('R4',PllR4);
R4.Minimum = 50;
R4.Maximum = 2000;
R4.Free = true;

% Capacitances
C1 = realp('C1',PllC1);
C1.Minimum = 1e-12;
C1.Maximum = 1e-7;
C1.Free = true;

C2 = realp('C2',PllC2);
C2.Minimum = 1e-12;
C2.Maximum = 1e-7;
C2.Free = true;

C3 = realp('C3',PllC3);
C3.Minimum = 1e-12;
C3.Maximum = 1e-7;
C3.Free = true;

C4 = realp('C4',PllC4);
C4.Minimum = 1e-12;
C4.Maximum = 1e-7;
C4.Free = true;
```

Using these tunable parameters, create a custom tunable model based on the loop filter transfer function equation specified in the More About section of the Loop Filter block reference page. `loopFilterSys` is a `genss` (Control System Toolbox) model parameterized by `R2`, `R3`, `R4`, `C1`, `C2`, `C3`, and `C4`.

$$Z(s) = \frac{R_2C_2s + 1}{s(A_4s^3 + A_3s^2 + A_2s + A_1)}$$

$$A_4 = C_1C_2C_3C_4R_2R_3R_4$$

$$A_3 = C_1C_2R_2R_3(C_3 + C_4) + C_4R_4(C_2C_3R_3 + C_1C_3R_3 + C_1C_2R_2 + C_2C_3R_2)$$

$$A_2 = C_2R_2(C_1 + C_3 + C_4) + R_3(C_1 + C_2)(C_3 + C_4) + C_4R_4(C_1 + C_2 + C_3)$$

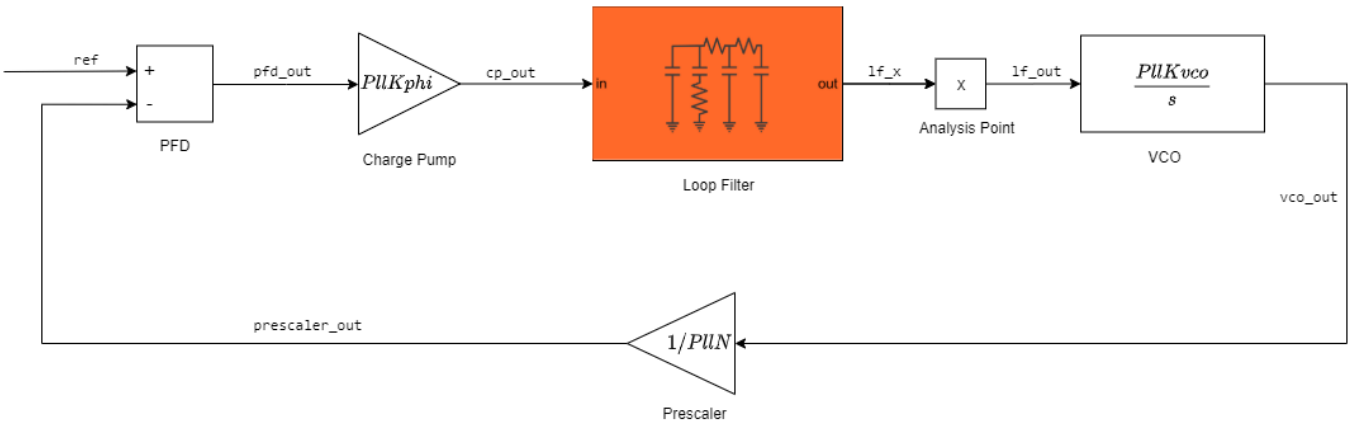
$$A_1 = C_1 + C_2 + C_3 + C_4$$

```
A4 = C1*C2*C3*C4*R2*R3*R4;
A3 = C1*C2*R2*R3*(C3+C4)+C4*R4*(C2*C3*R3+C1*C3*R3+C1*C2*R2+C2*C3*R2);
A2 = C2*R2*(C1+C3+C4)+R3*(C1+C2)*(C3+C4)+C4*R4*(C1+C2+C3);
A1 = C1+C2+C3+C4;

loopFilterSys = tf([R2*C2, 1],[A4, A3, A2, A1, 0]);
```

Use the transfer function representations to define the fixed blocks in the architecture (charge pump, VCO, and prescaler), based on their respective frequency response characteristics [1].

```
chargePumpSys    = tf(PllKphi,1);       % Linearized as a static gain
vcoSys           = tf(PllKvco,[1 0]);   % Linearized as an integrator
prescalerSys     = tf(1/PllN,1);        % Linearized as a static gain
```

Define input and output names for each block. Connect the elements based on signal names (see `connect` (Control System Toolbox)) to create a tunable closed-loop system (see `genss`) representing the PLL architecture as shown.




```
chargePumpSys.InputName = 'pfd_out';                % Charge pump (fixed block)
chargePumpSys.OutputName = 'cp_out';

loopFilterSys.InputName = 'cp_out';                 % Loop filter (tunable block)
loopFilterSys.OutputName = 'lf_x';

AP = AnalysisPoint('X');                            % Analysis point does not change the architec
AP.InputName = 'lf_x';
AP.OutputName = 'lf_out';

vcoSys.InputName = 'lf_out';                        % VCO (fixed block)
vcoSys.OutputName = 'vco_out';

prescalerSys.InputName = 'vco_out';                 % Prescaler (fixed block)
prescalerSys.OutputName = 'prescaler_out';

pfd = sumblk('pfd_out = ref - prescaler_out');      % Phase-frequency detector (sum block)

% Create a genss model for the closed-loop architecture
CL0 = connect(chargePumpSys,loopFilterSys,AP,vcoSys,prescalerSys,pfd,'ref','vco_out');
```

Loop-Shaping Design

Define the loop gain as a frequency-response data model by providing target gains for at least two decades below and two decades above the desired open-loop bandwidth. The desired roll-off is typically higher, which results in a higher attenuation of phase noise.

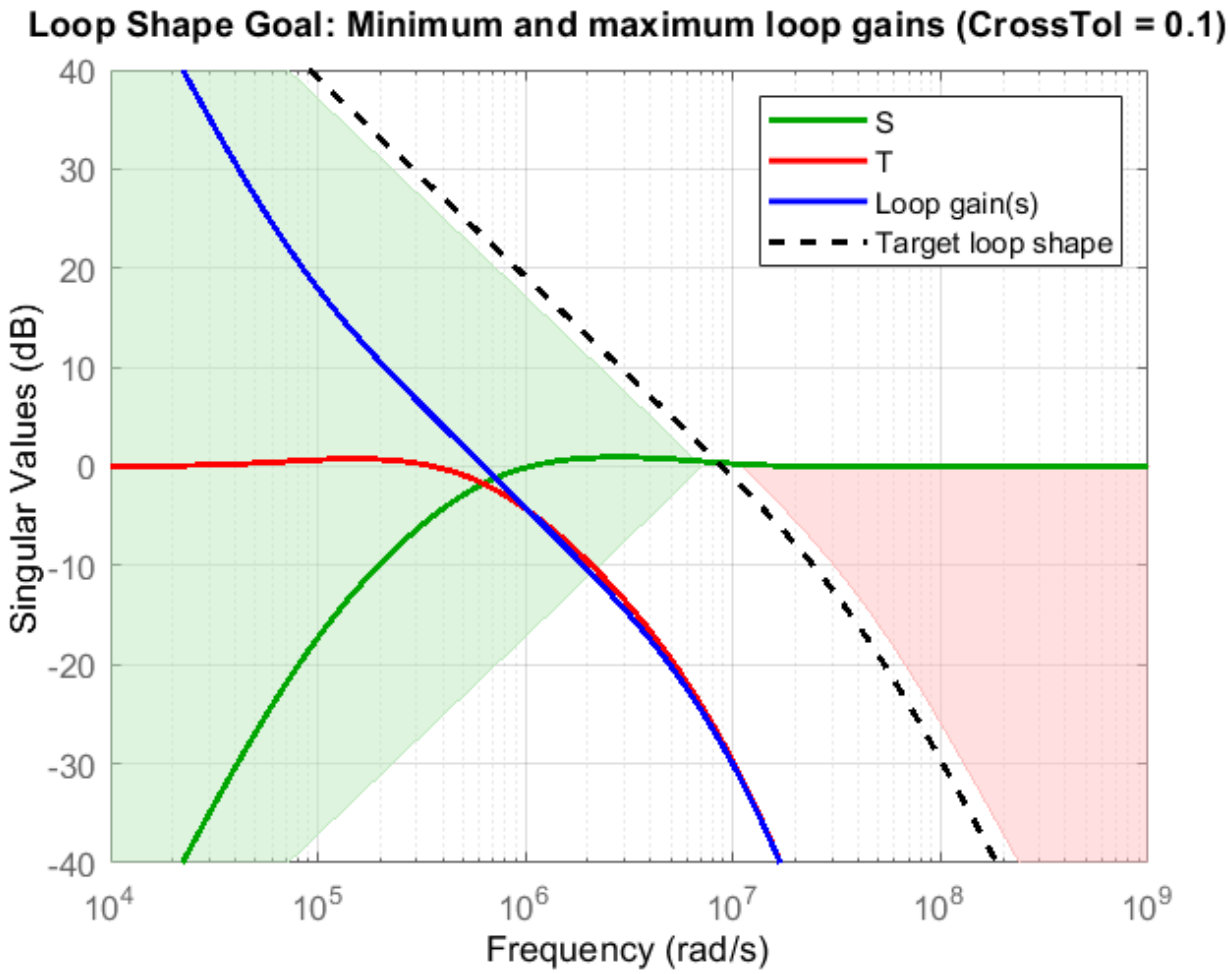
Specifying the appropriate target loop shape is the critical aspect of this design. The tunable compensator is a fourth-order system with a single integrator and a single zero, and the plant represents an integrator. The loop gains must be a feasible target for the open-loop structure.

For tuning the loop filter, create a tuning goal based on a target loop shape specifying the integral action, a 3 MHz crossover, and a roll-off requirement of 40 dB/decade. The goal is enforced for three decades below and above the desired open-loop bandwidth.

```
LoopGain = frd([100,10,1,1e-2,1e-4],2*pi*[1e4,1e5,3e6,3e7,3e8]);    % frd uses response data and
LoopShapeGoal = TuningGoal.LoopShape('X',LoopGain);                  % Use AnalysisPoint as locati
LoopShapeGoal.Focus = 2*pi*[1e3, 1e9];                               % Enforce goal in frequency r
LoopShapeGoal.Name = 'Loop Shape Goal';                              % Tuning goal name

MarginsGoal = TuningGoal.Margins('X',7.6,60);
MarginsGoal.Focus = [0 Inf];
MarginsGoal.Openings = {'X'};
MarginsGoal.Name = 'Margins Goal';
```

Observe the current open-loop shape of the PLL system with reference to the target loop shape. **S** represents the inverse sensitivity function and **T** represents the complementary sensitivity function. By default, Control System Toolbox plots use rad/s as the frequency unit. For more information on how to change the frequency unit to Hz, see “Toolbox Preferences Editor” (Control System Toolbox).

```
figure
viewGoal(LoopShapeGoal,CL0)
```



Use `systune` to tune the fixed-structure feedback loop. Doing so computes the resistance and capacitance values to meet the soft design goal based on the target loop shape. Run the tuning algorithm with five different initial value sets in addition to the initial values defined during the creation of the tunable scalar real parameters.

```
Options = systuneOptions();
Options.SoftTol = 1e-5;      % Relative tolerance for termination
Options.MinDecay = 1e-12;    % Minimum decay rate for closed-loop poles
Options.MaxRadius = 1e12;    % Maximum spectral radius for stabilized dynamics
Options.RandomStart = 5;     % Number of different random starting points

[CL,fSoft,gHard,Info] = systune(CL0,[LoopShapeGoal; MarginsGoal],[],Options);
```

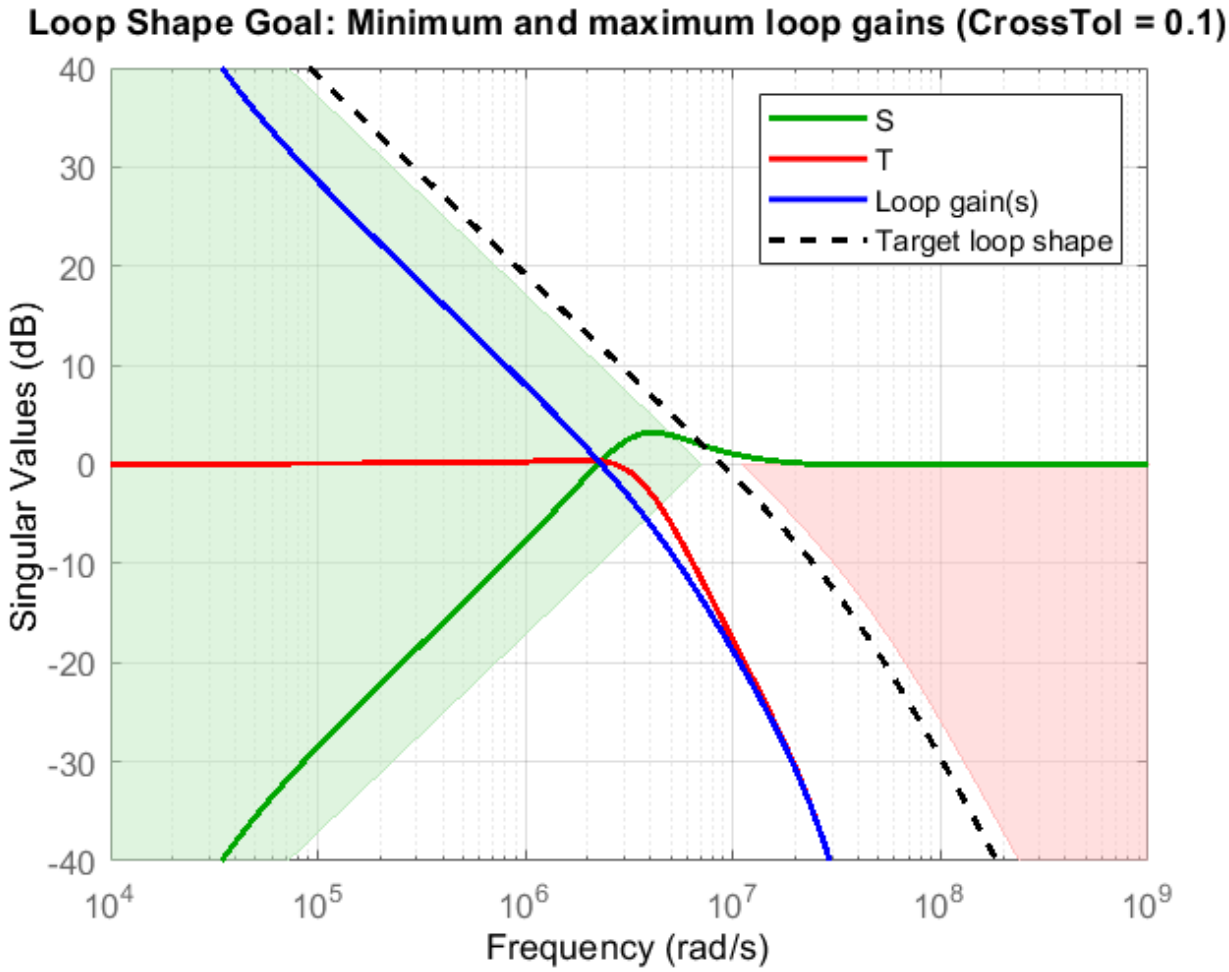
```
Final: Soft = 3.16, Hard = -Inf, Iterations = 79
Final: Failed to enforce closed-loop stability (max Re(s) = 3.1e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 6.7e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 6.3e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 7.5e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 4.9e+04)
```

`systune` returns the tuned closed-loop system `CL` in generalized state-space form.

The algorithm fails to converge for the random initial values, and provides a feasible solution only when the current loop filter component values are chosen as the initial conditions. For tuning problems that are less complex, such as a third-order loop filter, the algorithm is less sensitive to initial conditions and randomized starts are an effective technique for exploring the parameter space and converging to a feasible solution.

Examine the tuned open-loop shape with reference to the target loop shape. Observe that while the tuned loop shape does not meet the target, the open-loop bandwidth increases while the loop keeps the same high-frequency attenuation.

```
figure
viewGoal(LoopShapeGoal,CL)
```



Export Results to Simulink Model

Extract the tuned loop filter component values.

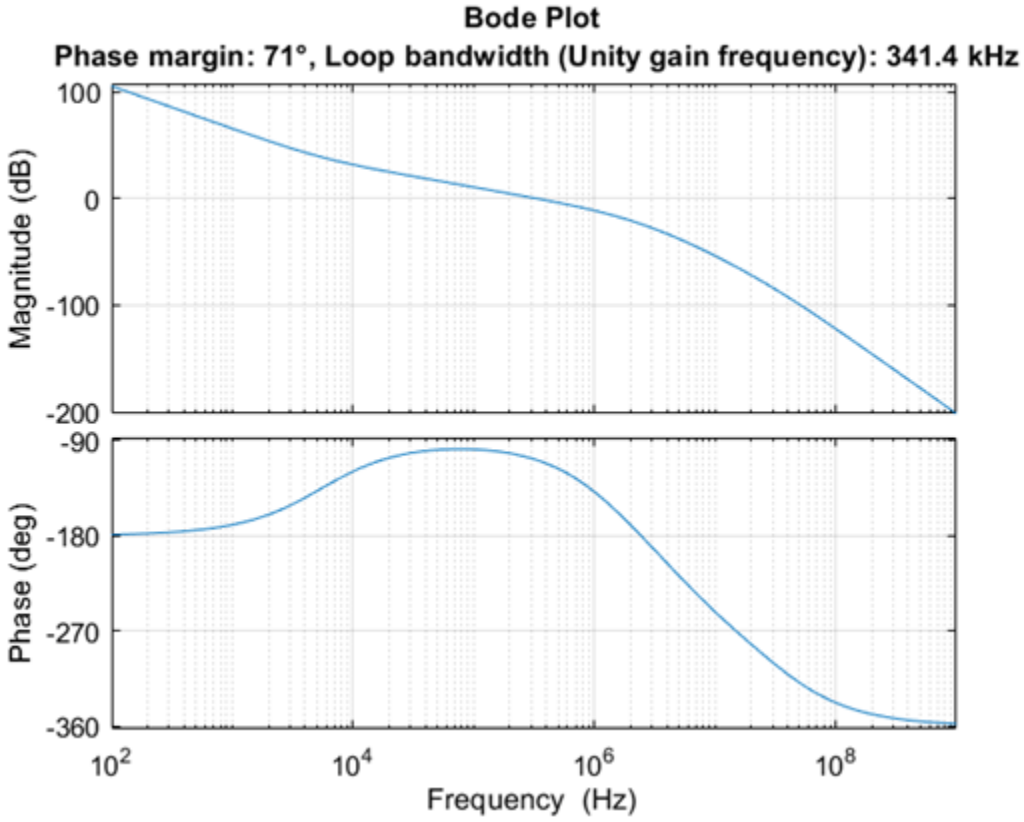
```
Rtuned = [getBlockValue(CL,'R2'),...
          getBlockValue(CL,'R3'),...
          getBlockValue(CL,'R4')];

Ctuned = [getBlockValue(CL,'C1'),...
          getBlockValue(CL,'C2'),...
          getBlockValue(CL,'C3'),...
          getBlockValue(CL,'C4')];
```

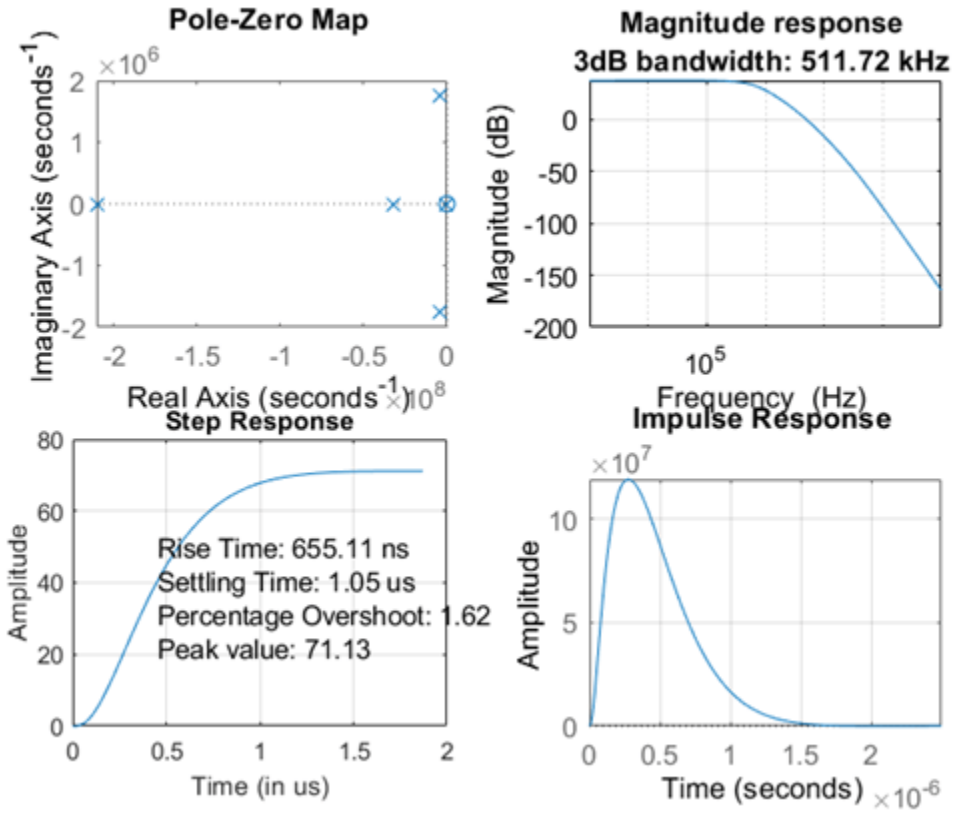
Write the tuned loop filter component values to the PLL block using the `setLoopFilterValue` helper function provided with the example.

```
blk = [model,'/Integer N PLL with Single Modulus Prescaler'];
setLoopFilterValue(blk,Rtuned,Ctuned);
```

Observe the Open Loop Analysis and Closed Loop Analysis plots from the **Analysis** tab in the Integer N PLL with Single Modulus Prescaler block parameters. The unity gain frequency and the 3-dB

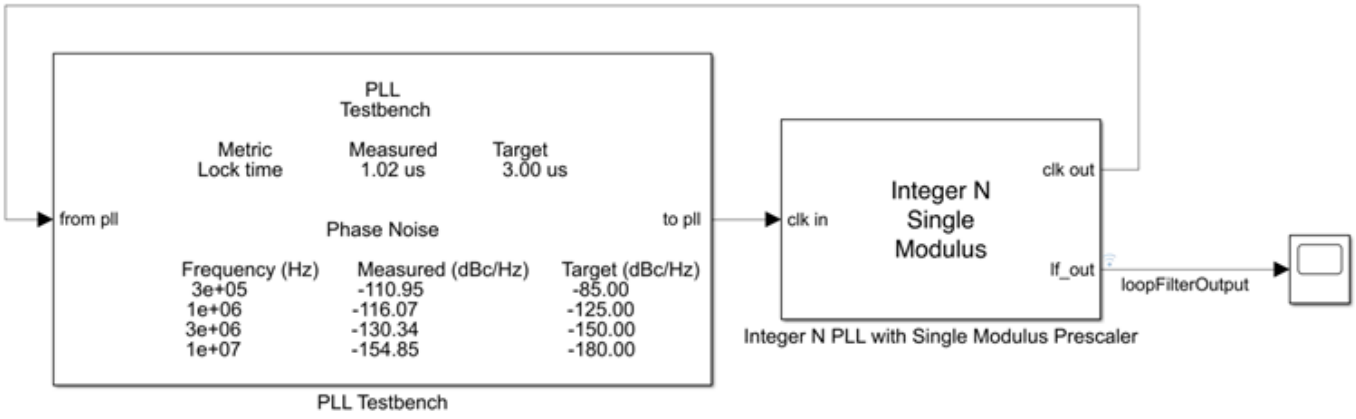
bandwidth show improvement and are now 341.4 kHz and 511.72 kHz, respectively, while the loop keeps the same phase noise profile.

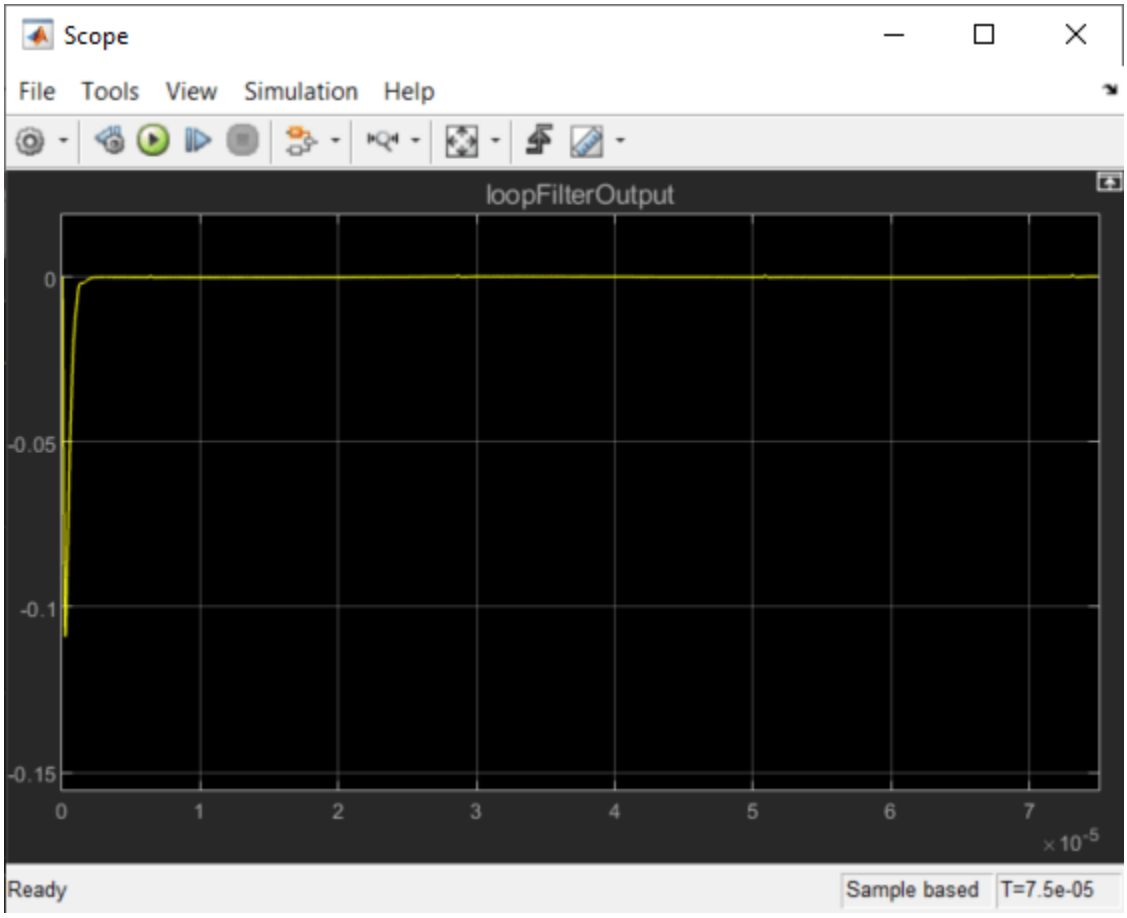


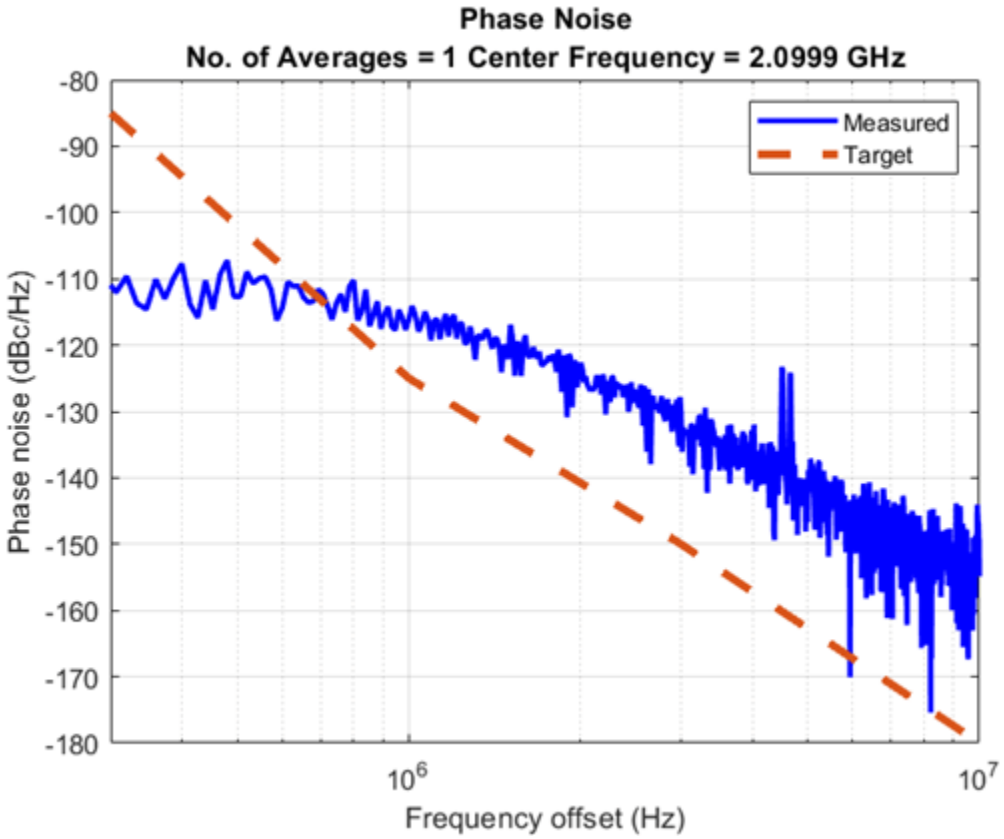


Simulate the model and get the PLL Testbench measurements and loop filter output with the tuned components.

```
sim(model);
```




Scope
File Tools View Simulation Help
loopFilterOutput
0
-0.05
-0.1
-0.15
0 1 2 3 4 5 6 7
×10^-5
Ready
Sample based
T=7.5e-05



References

[1] Banerjee, Dean. *PLL Performance, Simulation and Design*. Indianapolis, IN: Dog Ear Publishing, 2006.

Digital Phase Locked Loop

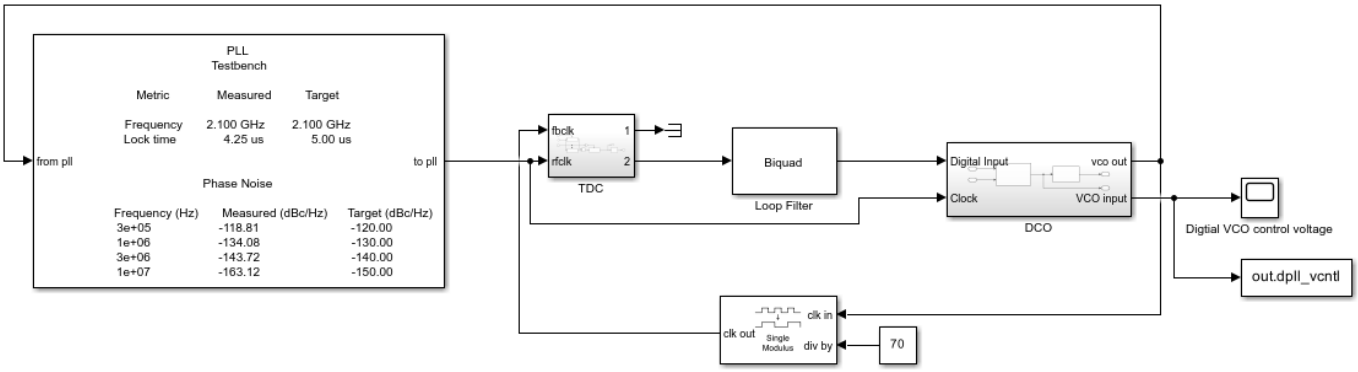
This example shows how to model a digital phase locked loop using the Mixed-Signal Blockset™. In a digital phase locked loop, phase detection is performed by a time to digital converter (TDC), loop filtering is performed by a digital filter, and the oscillator is a digitally controlled oscillator (DCO). This example demonstrates a behavioral model of a TDC, using the BiquadFilter from the DSP System Toolbox™ as the digital loop filter, and using VCOs and DACs from the Mixed-Signal Blockset to model the DCO.

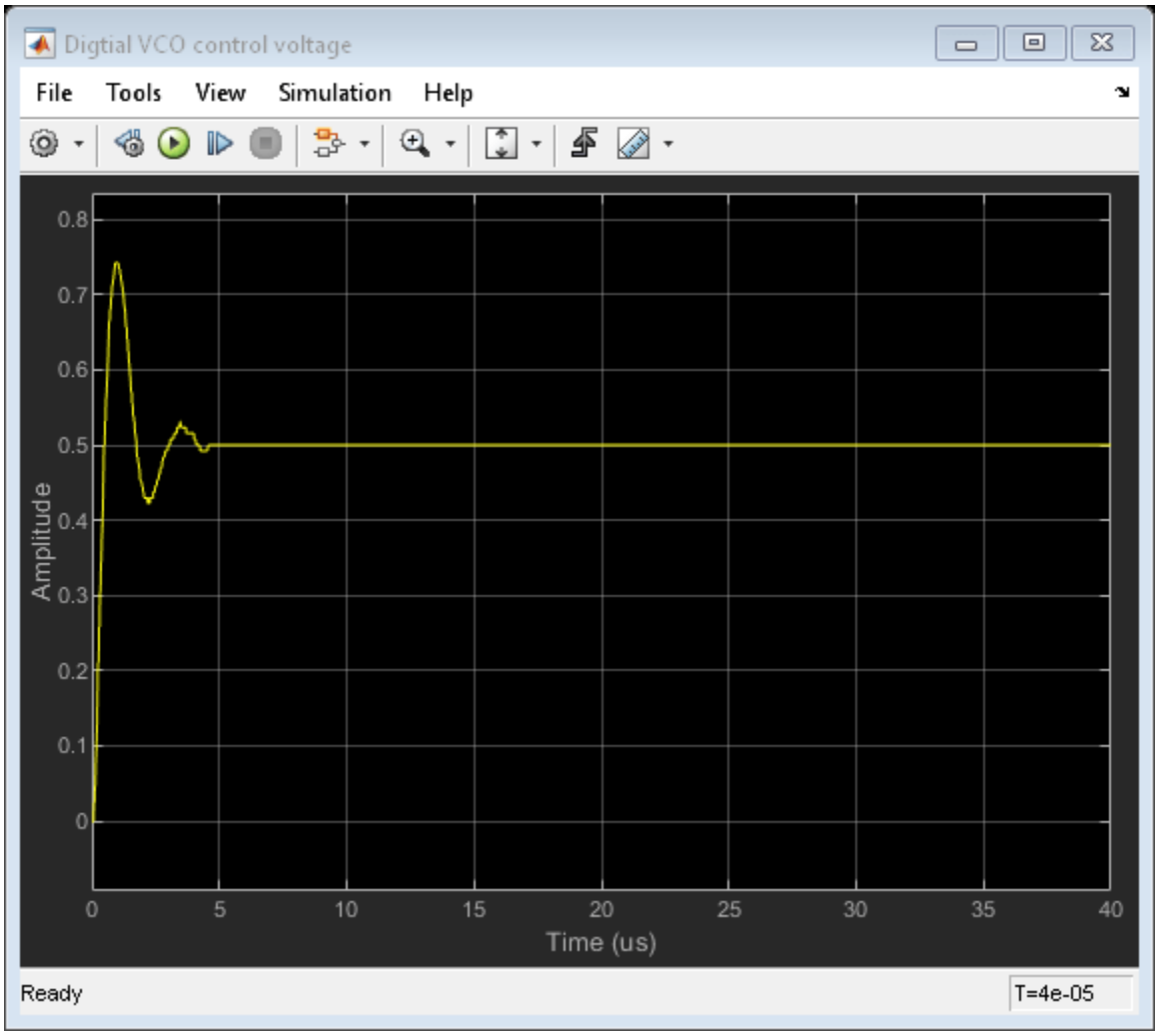
This example also demonstrates one method for designing the control loop of a digital PLL by using the PLL architectures from the Mixed-Signal Blockset to design an analog PLL and then translate the analog design to its digital equivalent.

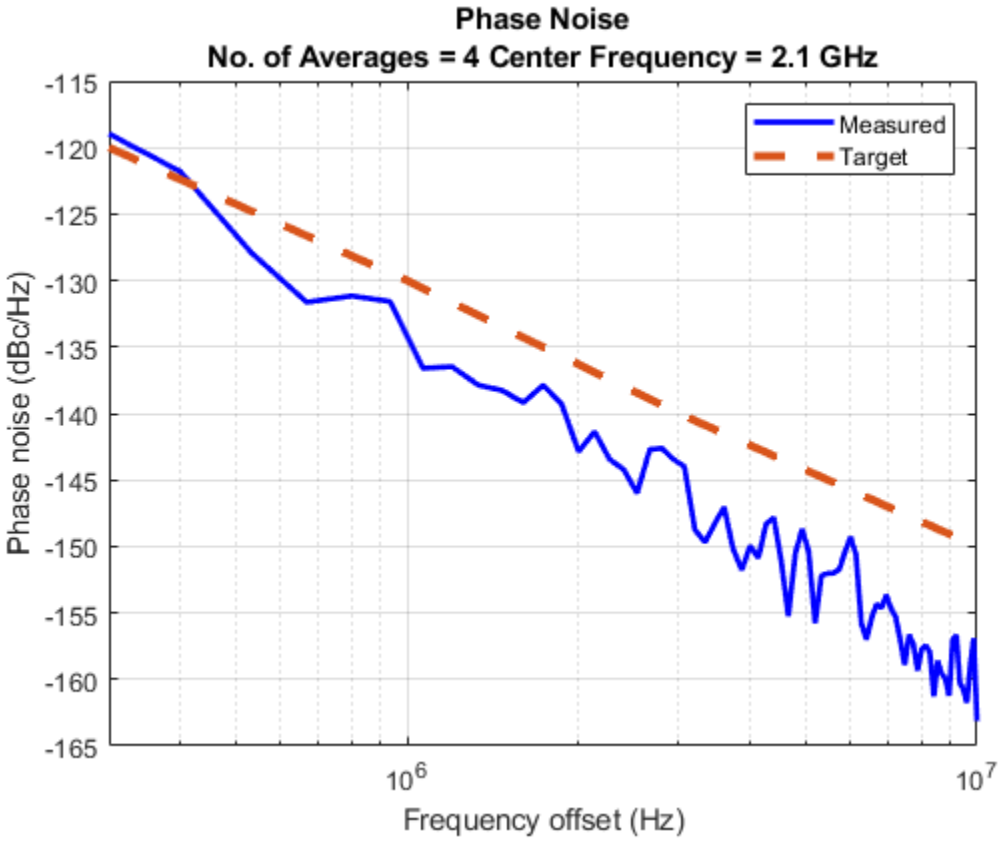
Digital PLL

Open and run the example model of a digital PLL. Then select the PLL Testbench in the model and click the Plot phase noise button (or call the button's callback function). Save loop filter waveform and phase noise data for later comparison to analog PLL results.

```
open('DigitalPLL.slx');
simout = sim(bdroot);
msblks.PLL.pllTBPlotPhaseNoiseButton([bdroot '/PLL Testbench']);
dpll_vcntl = simout.dpll_vcntl;
getDigitalPllPhaseNoise;
```



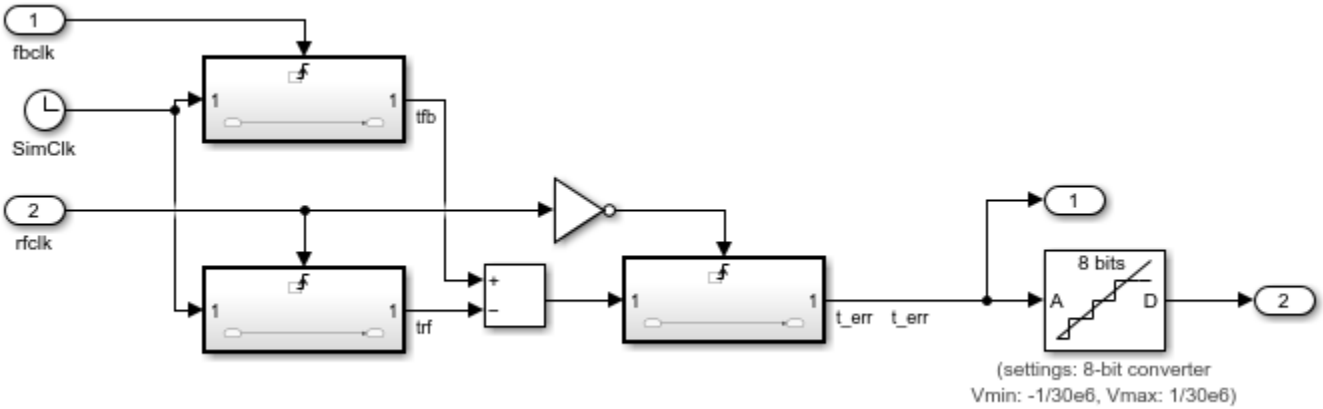

Digtial VCO control voltage
File
Tools
View
Simulation
Help
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Amplitude
0
5
10
15
20
25
30
35
40
Time (us)
Ready
T=4e-05



Time to Digital Converter

The behavioral model of the TDC uses triggered subsystems to sample the clock time at the rising edges of the reference clock and the PLL's fractional divider. It then calculates the difference between these two sampled clock times and converts to an eight bit integer output value. To model a digital feedback loop clocked by the reference, the output port sample rate is set to the reference frequency.

```
open_system('DigitalPLL/TDC','force');
```



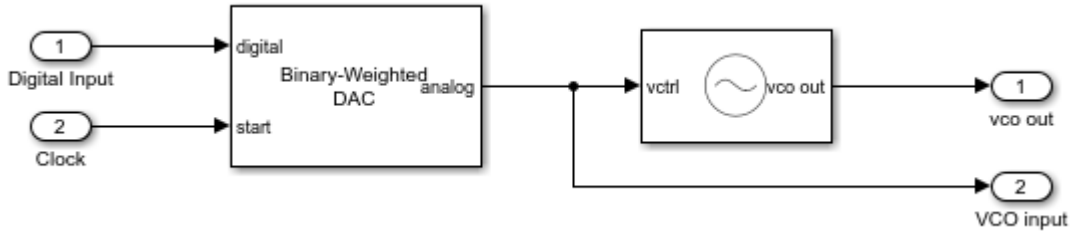
Digital Controlled Oscillator

The DCO model represents a circuit in which the control for a VCO is set by a digital to analog converter (DAC). The conversion in the DAC is triggered by the reference clock.

The phase noise impairment in the VCO is enabled. The DAC model can also model impairments. However the DAC impairments are not enabled in the example model.

You could alternatively model a DCO by directly setting a Mixed-Signal Blockset VCO's control voltage array to an array of consecutive integers that spans the range of digital input values. Then you can set the output frequency array to supply the output frequency for each possible input value.

```
open_system('DigitalPLL/DCO','force');
```



Digital Loop Filter

The loop filter is a BiquadFilter from the DSP System Toolbox. It is clocked at the reference clock frequency by the output port of the TDC.

The digital loop filter is designed using the automated design feature of the Integer N PLL with Single Modulus Prescaler model from the Mixed-Signal Blockset PLL Architectures library. An analog loop filter is designed to achieve a specified loop bandwidth and phase margin, then the circuit values and sample interval are translated to digital filter coefficients. To obtain the correct overall loop gain, a scale factor is applied to the biquad filter model of the loop filter.

The derivation of the loop gain scale factor parallels that of A Design Procedure for All-Digital Phase-Locked Loops Based on a Charge-Pump Phase-Locked-Loop Analogy.

The variables used in the derivation are

- I_{CP} Charge pump maximum current amplitude
- $Z(s)$ Analog loop filter transfer impedance
- K_{VCO} VCO sensitivity for both analog and digital PLLs
- T_{REF} Reference clock period
- Δ_{TDC} TDC time resolution
- $H(s)$ Digital loop filter transfer function
- K_{DAC} DAC gain

Note that for this example, the total range of the TDC is two reference clock periods.

The equation to make the loop gain of the analog and digital PLLs equal is:

$$\frac{I_{CP}}{2\pi} Z(s) \frac{K_{VCO}}{s} = \frac{2T_{REF}}{2_\pi} \frac{1}{\Delta_{TDC}} H(s) K_{DAC} \frac{K_{VCO}}{s}$$

Since the TDC and the DAC in this example have been configured to have the same number of bits, this equation reduces to:

$$H(s) = \frac{I_{CP}}{2} Z(s)$$

Since the function `getSOSfromAnalogPLL` included in this example produces filter coefficients that result in a filter gain $Z(s)$, the scale factor for the loop filter is $\frac{I_{CP}}{2}$.

While the loop filter coefficients, input signal and output signal in this example are all double precision floating point, it would also be possible to use Fixed-Point Designer™ to model the entire digital feedback path using exactly the same digital precision as in the circuit design, thus obtaining even more precise evaluation of impairments such as quantization noise.

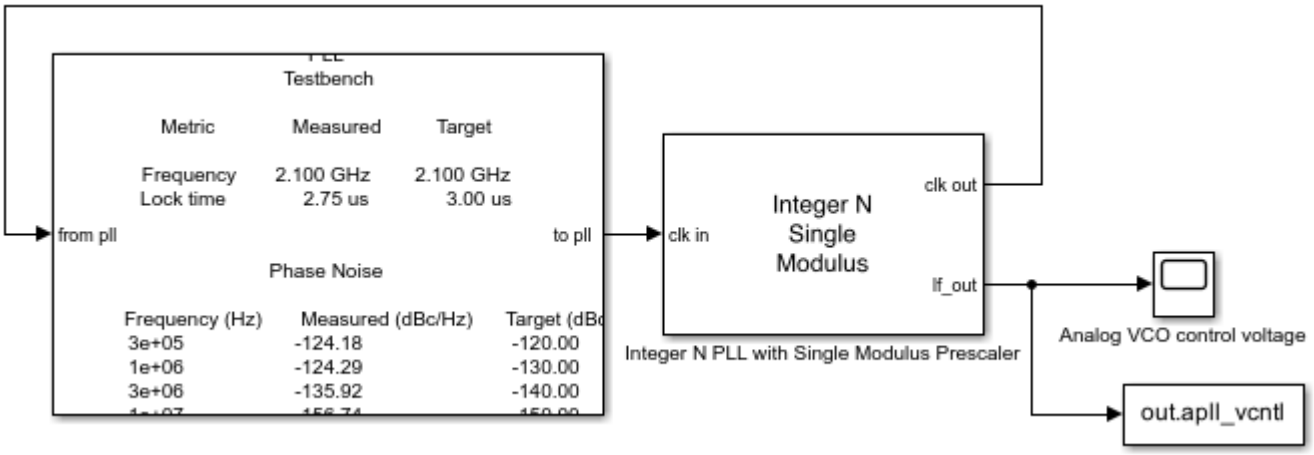
You can design and configure the digital loop filter by running the script:

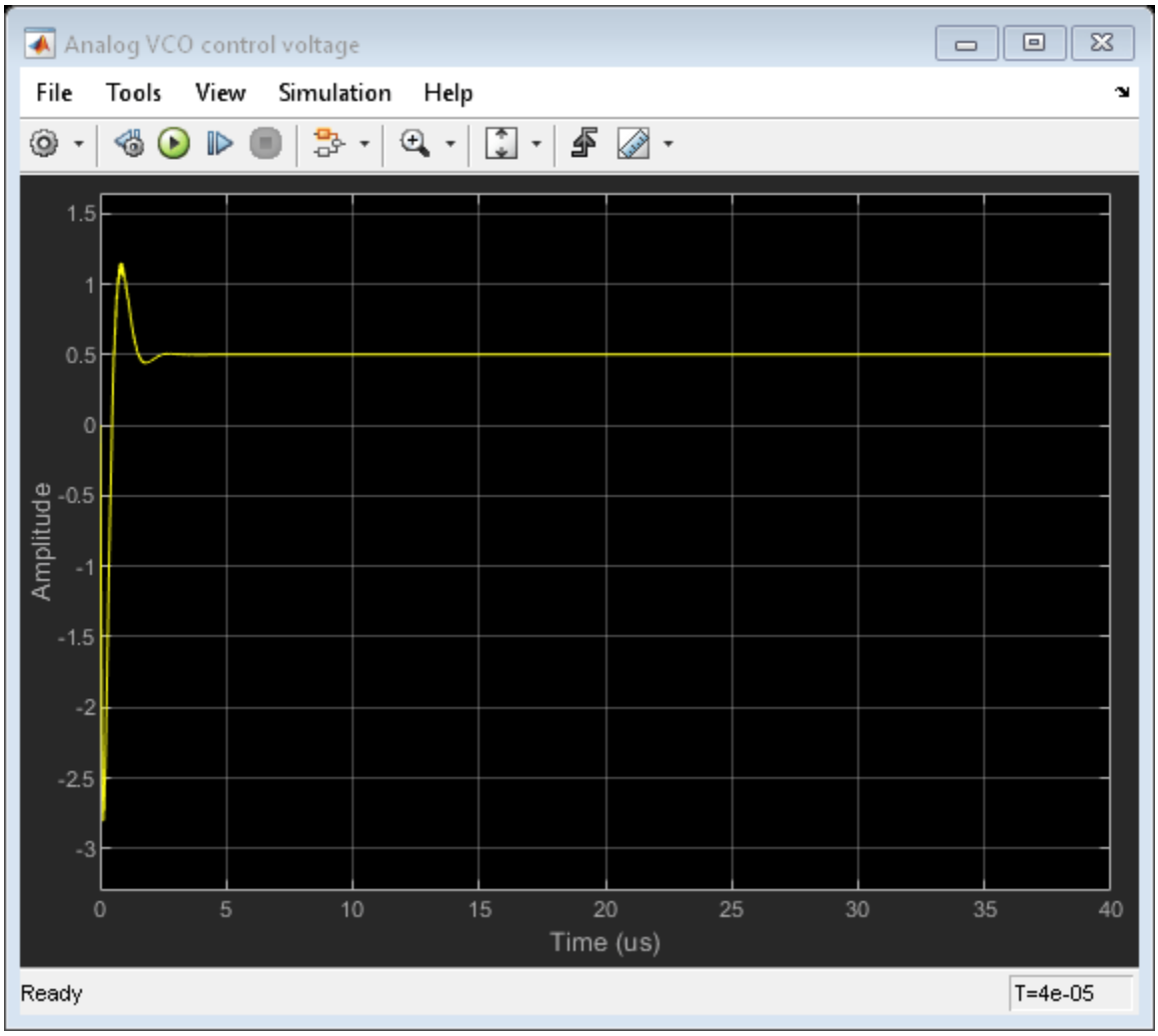
```
%%%designDigitalLoopFilter;
```

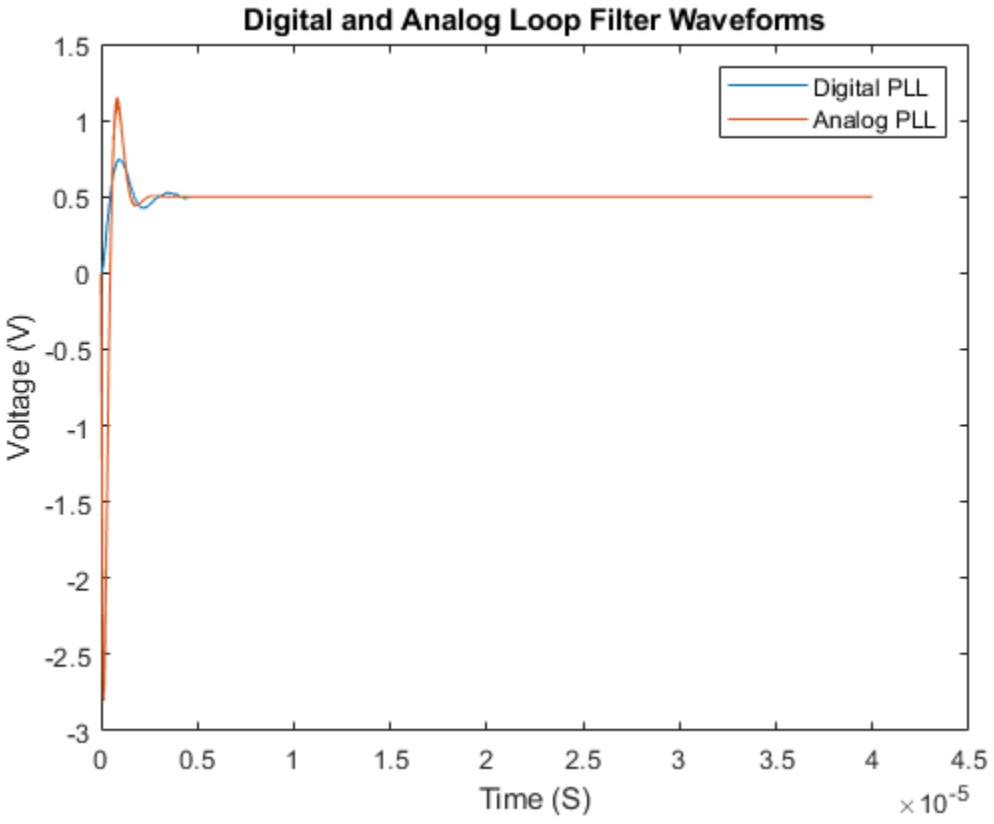
Compare to Analog PLL

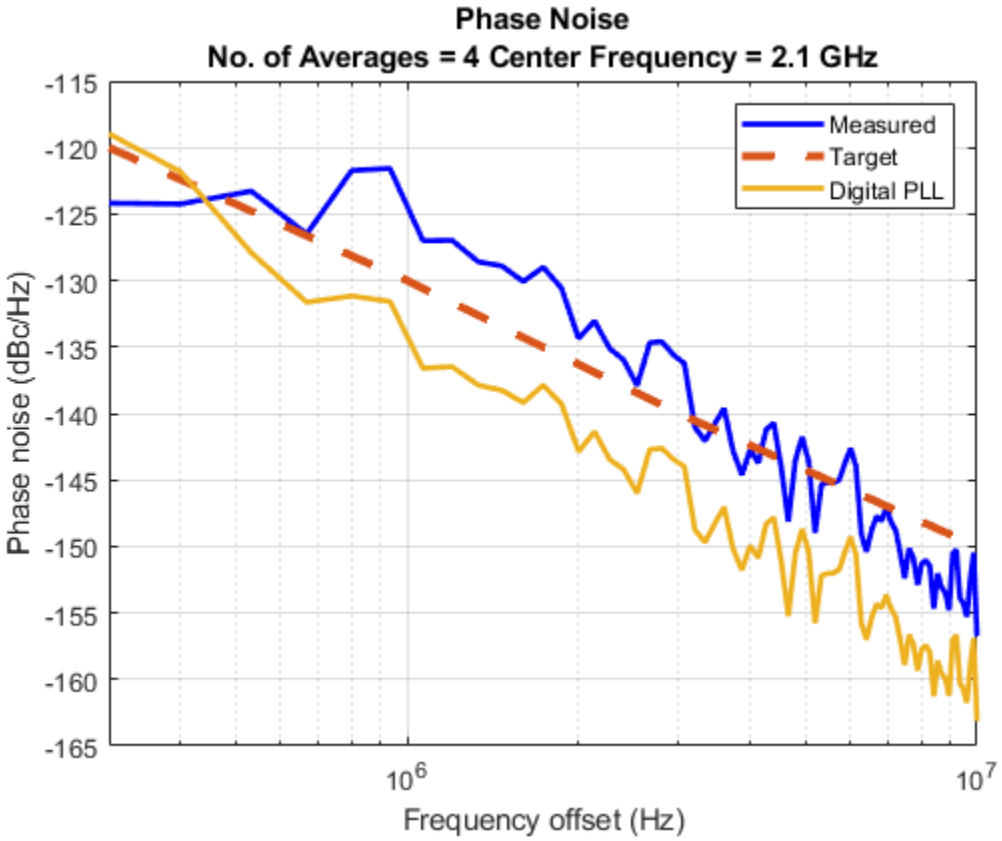
You can compare the loop acquisition of the analog PLL to that of the digital PLL. The additional high frequency ripple in the analog loop filter response is due to the charge pump waveform.

```
open('AnalogPLL.slx');
simout = sim(bdroot);
apll_vcntl = simout.apll_vcntl;
plotLoopFilterWaveforms;
msblks.PLL.pllTBPlotPhaseNoiseButton([bdroot '/PLL Testbench']);
plot(xdpll,ydpll,'LineWidth',2,'DisplayName','Digital PLL');
```




Analog VCO control voltage
File
Tools
View
Simulation
Help
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
Amplitude
0
5
10
15
20
25
30
35
40
Time (us)
Ready
T=4e-05


Digital and Analog Loop Filter Waveforms
Digital PLL
Analog PLL
Voltage (V)
Time (S)
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
×10-5


Phase Noise
No. of Averages = 4 Center Frequency = 2.1 GHz
Measured
Target
Digital PLL
Phase noise (dBc/Hz)
-115
-120
-125
-130
-135
-140
-145
-150
-155
-160
-165
10^6
10^7
Frequency offset (Hz)

Model PLLs in the Phase Domain

This example shows how to model a phase-locked loop (PLL) in the phase domain, compare the analytic results to simulation results in the time domain, and identify the advantages and disadvantages of each approach. Most PLL analysis is performed in the phase domain, and this example shows you how to produce a complete analysis using a minimum of time and effort.

The phase domain analysis calculates the PLL transfer function, loop error function, lock time and noise transfer impedances using the Control System Toolbox™. (Automated loop tuning is described in the "Tune Phase-Locked Loop Using Loop-Shaping Design" on page 1-23 example.)

The PLL design and the time domain simulation model were derived from the "Phase Noise at PLL Output" on page 1-2 example.

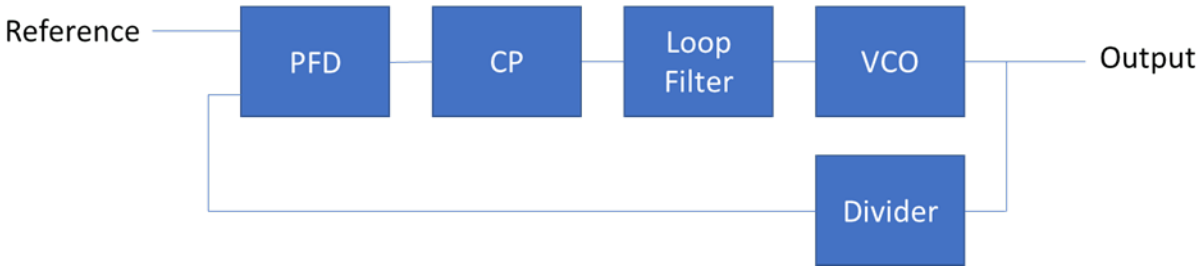
The general loop filter modeling procedure uses the Mixed-Signal Blockset™ Linear Circuit Wizard and was derived from the "Circuit Design Details Affect PLL Performance" on page 3-16 example.

Phase Domain Block Models

This section shows how to create the phase domain models for the individual blocks of an analog phase/frequency locked loop.

The term phase domain refers to the representation of a periodic signal in terms of its phase with respect to an ideal reference with the same period, instead of in terms of its voltage or current as a function of time. Most PLL analyses (in contrast with simulations) treat the PLL as a linear control system in the frequency domain, with the reference signal and the output of the VCO/divider represented in the phase domain.

The Control System Toolbox supports both transfer function (tf) and state space (ss) models. While the tf objects tend to be easier to use, the ss objects are often more accurate at higher frequencies.



To enable comparison of analysis results to time domain simulation results, this example uses the same design values as the Phase noise at PLL output example.

```
PllKphi = 5e-3;     % Charge Pump output current
PllKvco = 100e6;    % VCO sensitivity
PllN    = 70;       % Prescaler ratio
PllR2   = 1.33e3;   % Loop filter resistance for second order response (ohms)
PllR3   = 1.7e4;    % Loop filter resistance for third order response (ohms)
PllR4   = 0;        % Loop filter resistance for fourth order response (ohms)
PllC1   = 1.31e-11; % Loop filter direct capacitance (F)
PllC2   = 1.44e-10; % Loop filter capacitance for second order response (F)
PllC3   = 9.41e-13; % Loop filter capacitance for third order response (F)
PllC4   = 0;        % Loop filter capacitance for fourth order response (F)
```

Reference

The reference signal is assumed to have the same frequency as the ideal reference used to define the phase domain. However, there can be a time-varying phase offset between the reference signal and the ideal reference. This phase offset $\theta_{ref}(t)$, or $\theta_{ref}(s)$ in the Laplace domain, is an input to the linear control system.

VCO and Clock Divider

The VCO output phase is the integral of the VCO control voltage.

$$\theta_{VCO}(t) = \int 2\pi K_{VCO} V_{CNTL}(t) dt$$

Or, in the Laplace domain,

$$\theta_{VCO}(s) = \frac{2\pi K_{VCO}}{s} V_{CNTL}(s)$$

From this equation, the statement to construct an ss model of the VCO is

```
vco = ss(0,2*pi*PllKvco,1,0);
```

Given a divider ratio PllN (not necessarily integer), the ss object for the divider is

```
divN = ss(0,0,0,1/PllN);
```

The VCO also has phase noise at its output, characterized as a phase noise density $n_{VCO}(f)$, filtered by the closed loop gain of the PLL.

Phase/Frequency Detector (PFD) and Charge Pump (CP)

Given phase domain inputs $\theta_{ref}(t)$ and $\theta_{VCO}(t)$, and maximum output current I_{CP}, the average output current of the PFD/CP is:

$$i_{CP}(t) = (\theta_{ref}(t) - \theta_{VCO}(t)) \frac{I_{CP}}{2\pi}$$

Or in the Laplace domain:

$$i_{CP}(s) = (\theta_{ref}(s) - \theta_{VCO}(s)) \frac{I_{CP}}{2\pi}$$

The CP also has phase noise at its output, characterized as a phase noise density $n_{CP}(f)$.

Loop Filter

The loop filter presents a transfer impedance $Z_{LF}(s)$ to the CP output, thus converting the CP output current to a control voltage applied to the VCO. [1] describes a family of widely used second, third and fourth order loop filters. More convenient modeling of this and other loop filter circuit designs using the Linear Circuit Wizard is described in the General Loop Filters section below.

The second order filter is described by the equation

$$Z_2(s) = \frac{1 + sR_2C_2}{s(C_1 + C_2)(1 + sR_2\frac{C_1C_2}{C_1+C_2}}$$

The higher order filters are formed by adding a series R/shunt C section. Given an k-1 order filter design for which Z_(k-1) (s)= (b(s))/(a_(k-1) (s)) , followed by series resistor R_k and shunt capacitor C_k, the transfer impedance of the higher order filter is

$$Z_k(s) = \frac{b(s)}{a_{k-1}(s)(1 + sR_kC_k) + sC_kb(s)}$$

One way to transform these equations into an ss object is to calculate the transfer function coefficients and then use the `tf2ss` function. Given that the filter has nsections sections, with corresponding element values, the procedure is

```
nsections = 3;
b = zeros(1,nsections+1);
a            = [PllR2*PllC1*PllC2, PllC1+PllC2, 0];
b(end-1:end) = [PllR2*PllC2, 1];
if nsections > 2
    tmp = conv(b, [PllC3 0]);
    a = conv(a, [PllR3*PllC3, 1]) + tmp(end-3:end);
end
if nsections > 3
    tmp = conv(b, [PllC4 0]);
    a = conv(a, [PllR4*PllC4, 1]) + tmp(end-4:end);
end
[A,B,C,D] = tf2ss(b,a);
zlf = ss(A,B,C,D);
```

The resistors in the loop filter also produce Johnson noise. This noise can be modeled as a noise current that is then applied to a transfer impedance to produce a noise voltage at the output of the loop filter. Given a resistance R, the current spectral density is $4kT/R$ Watts/Hz, where k is Boltzmann's constant and T is the absolute temperature.

The noise transfer impedance calculation follows a very similar process as that for the overall loop filter transfer impedance. However it must be calculated separately for each resistor in the loop filter. The function `getNoiseTransferImpedance(R,C,n)` included with this example produces an ss model for the device noise transfer impedance of the resistor R_n.

Control Loop Analysis

This section shows how to perform numerous open and closed loop analyses using the phase domain models of the PLL blocks. The process is to first create a closed loop system model and then invoke functions that perform the desired analysis using that model.

Most of the analyses only require a single input, single output model of the PLL, and so you will define that model first. The noise filtering analysis requires a multiple input model, and you will define that in the noise filtering section.

In PLL control loop modeling the divider ratio N is usually kept separate from the gain G due to the PFD/CP, loop filter, and VCO.

$$G(s) = \frac{I_{CP}}{2\pi}Z(s)\frac{2\pi K_{VCO}}{s} = \frac{I_{CP}Z(s)K_{VCO}}{s}$$

The loop transfer function (from reference input to VCO output) is

$$H(s) = \frac{G(s)}{(1 + G(s)/N}$$

The corresponding MATLAB code is

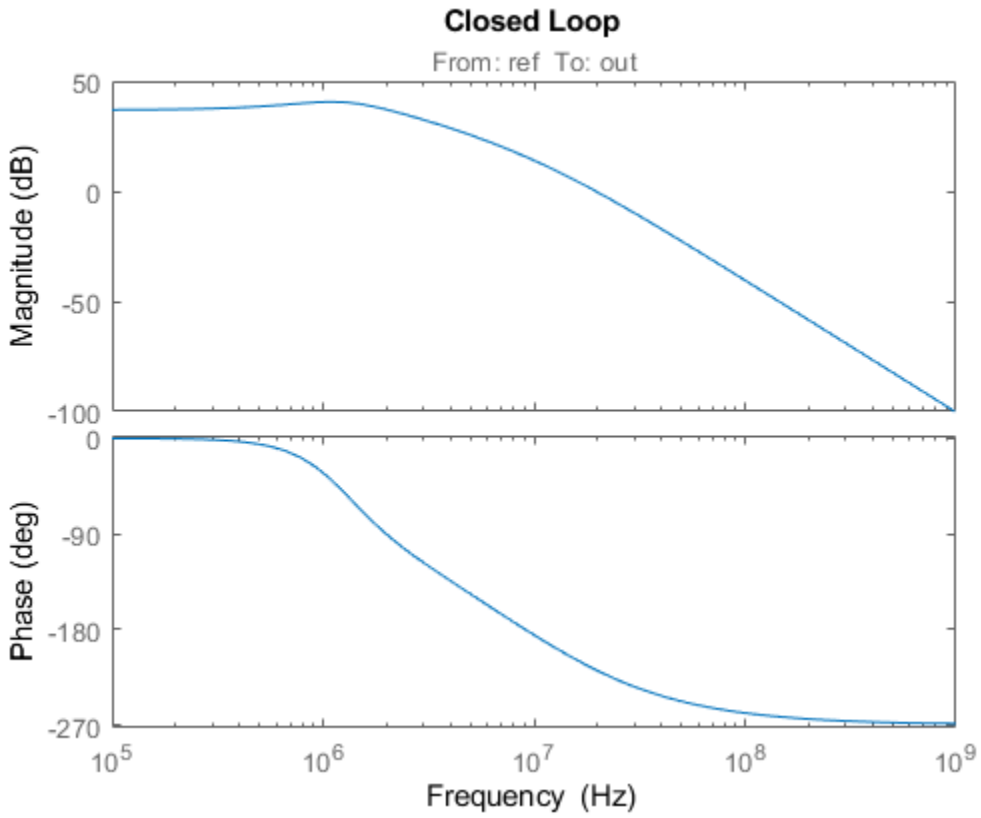
```
G = PllKphi/2/pi*zlf*vco;
H = feedback(G,divN);
H.InputName  = 'ref';
H.OutputName = 'out';
```

Closed Loop Transfer Function

Use the bode function to plot the magnitude and phase of the closed loop transfer function.

NOTE: Configure the bode function to scale the frequency in Hz and not rad/s.

```
figure(1);
opts = bodeoptions('cstprefs');
opts.FreqUnits = 'Hz';
bodeplot(H,opts);
title('Closed Loop');
```

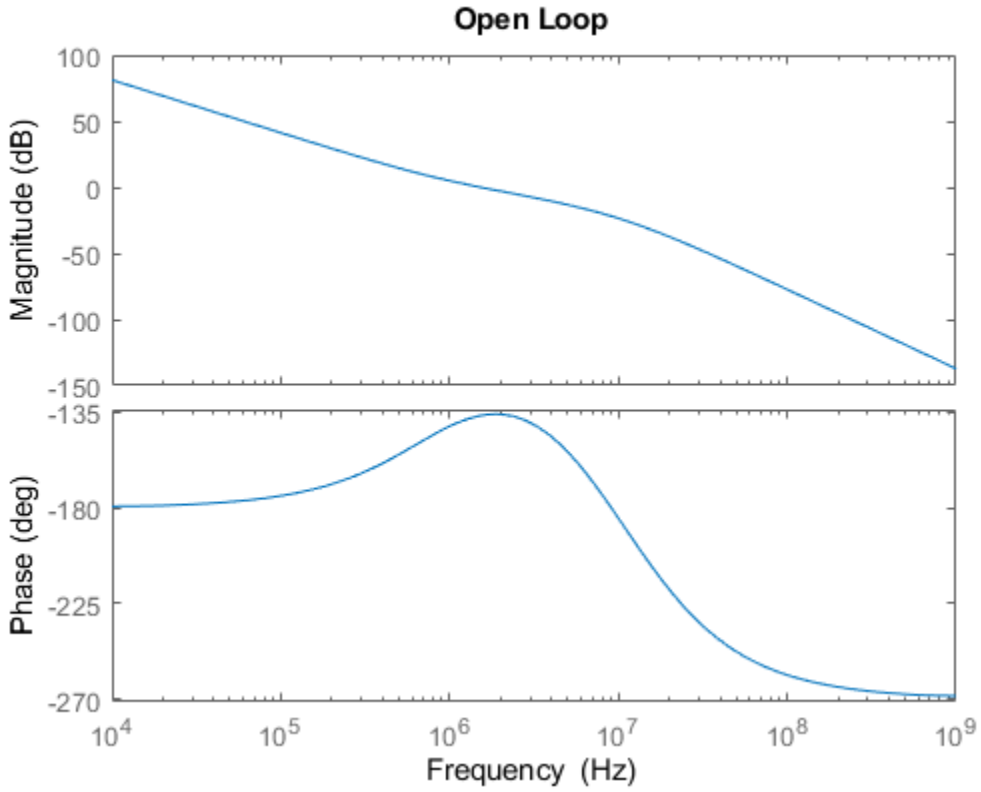


You can also use other variants of the bode function to output results to the workspace for use in other analyses.

Loop Stability

Use the bode function again to plot the magnitude and phase of the loop gain, only this time use the series function instead of the feedback function

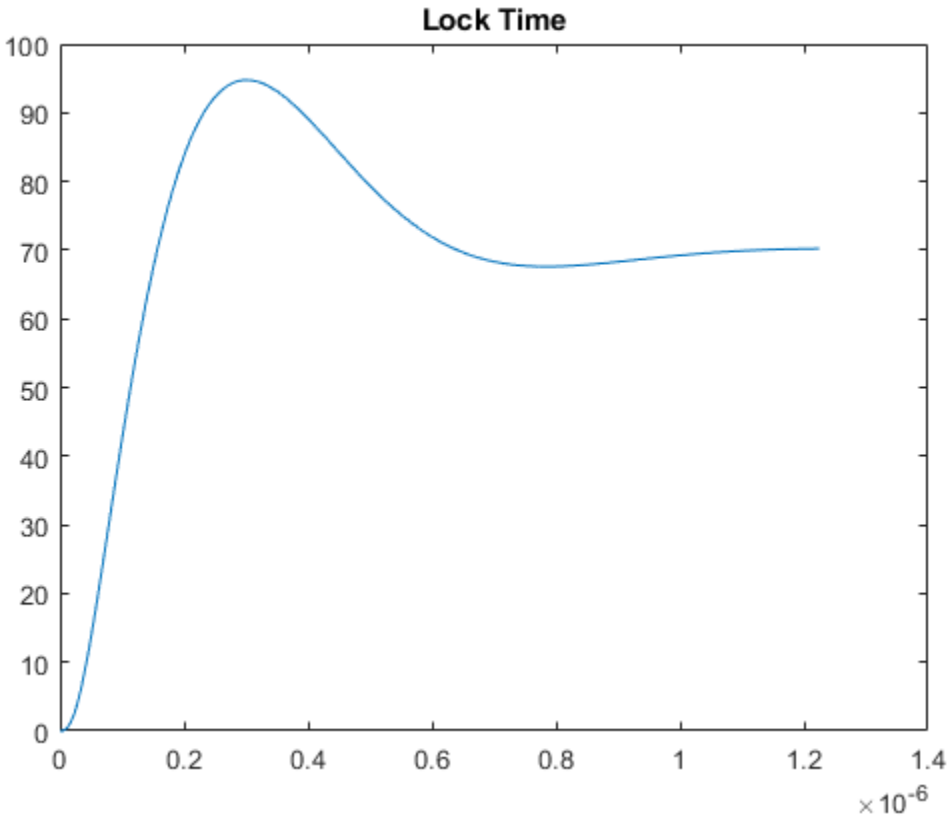
```
GoverN = series(G,divN);
figure(2);
bodeplot(GoverN,opts);
title('Open Loop');
```



Lock Time

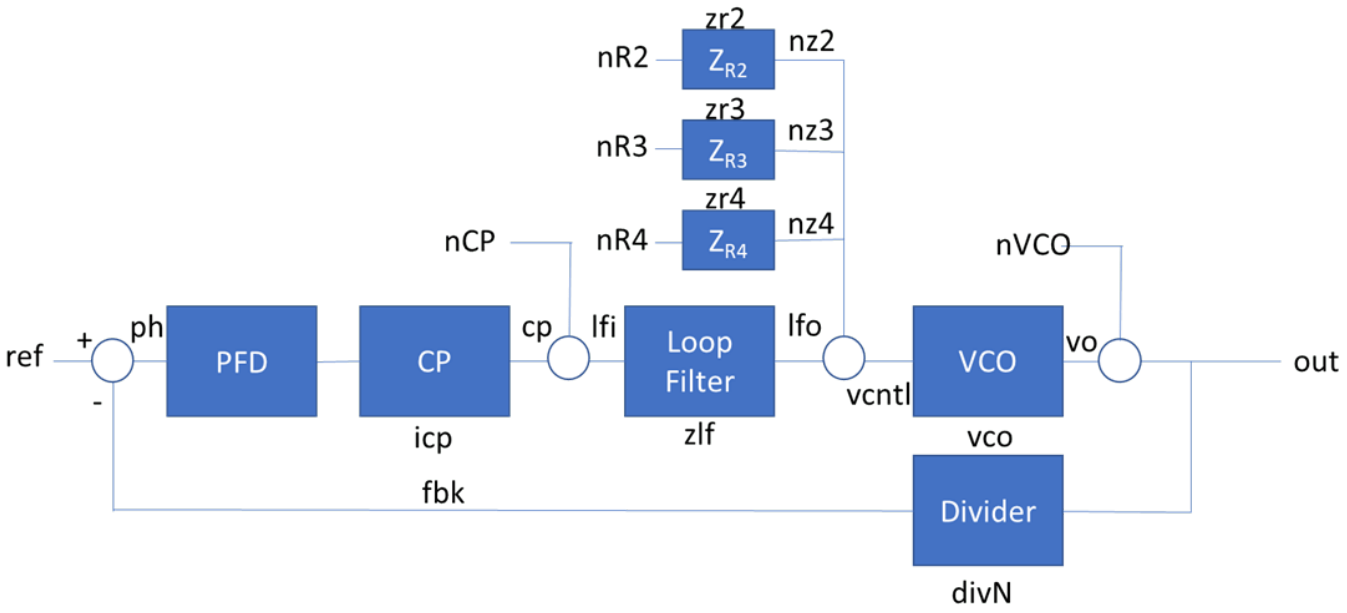
Obtain the loop lock time from the loop step response.

```
figure(3);
[ystep,tstep] = step(H);
plot(tstep,ystep);
title('Lock Time');
```



Noise Filtering

The analysis of noise filtering usually requires multiple inputs- one for each noise source. While such a model could be used for all the analyses presented earlier in this section, that would complicate the coding without providing additional insight. The required block diagram is shown below.



The procedure is to use the connect function to obtain a transfer function from each noise source to the PLL output, apply each of these transfer functions to their respective noise source, and then plot the resulting output noise spectral densities.

The zlf, vco, and divN blocks are already defined.

Define blocks required for noise filtering analysis.

```
R = [PllR2, PllR3, PllR4];
C = [PllC1, PllC2, PllC3, PllC4];
zr2 = getNoiseTransferImpedance(R,C,2);
zr3 = getNoiseTransferImpedance(R,C,3);
```

The example loop filter is third order. No need for zr4.

```
icp = ss(0,0,0, PllKphi/2/pi);
sum1 = sumblk('ph = ref - fbk');
sum2 = sumblk('lfi = cp + nCP');
sum3 = sumblk('vcntl = lfo + nz2 + nz3');
sum4 = sumblk('out = vo + nVCO');
%
zlf.InputName = 'lfi';    zlf.OutputName = 'lfo';
vco.InputName = 'vcntl';    vco.OutputName = 'vo';
divN.InputName = 'out';    divN.OutputName = 'fbk';
icp.InputName = 'ph';    icp.OutputName = 'cp';
zr2.InputName = 'nR2';    zr2.OutputName = 'nz2';
zr3.InputName = 'nR3';    zr3.OutputName = 'nz3';
```

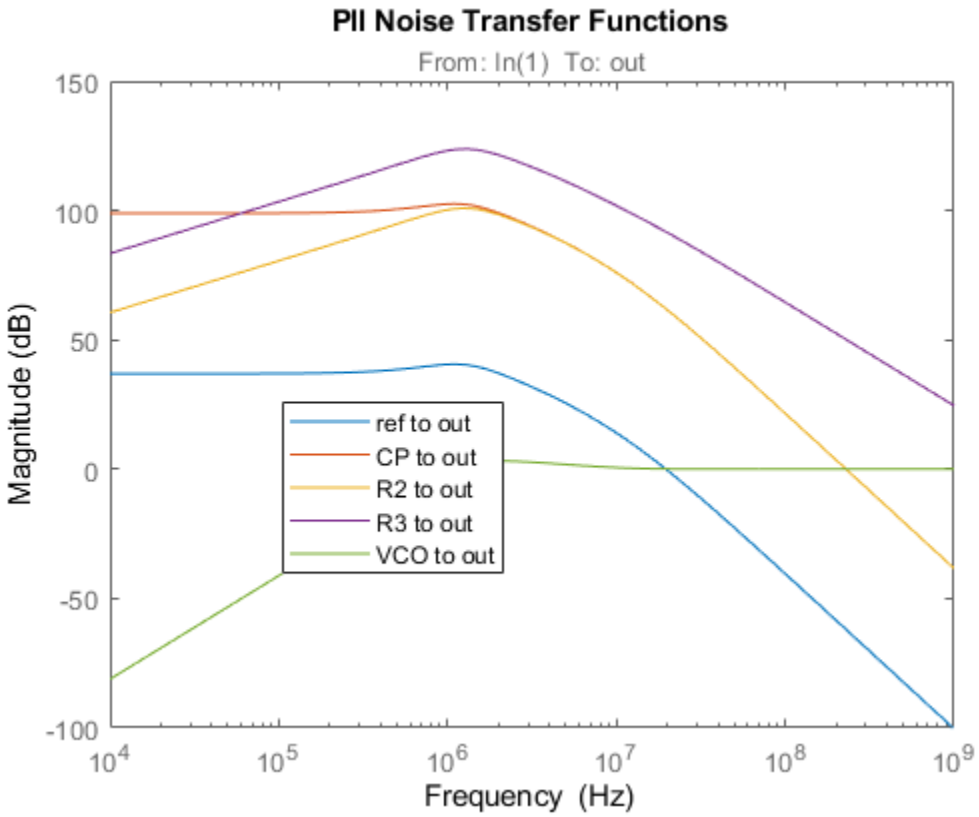
Connect the PLL as a control system.

```
Hnoise = connect(icp,zlf,vco,divN,zr2,zr3,sum1,sum2,sum3,sum4,...
                 {'ref','nCP','nR2','nR3','nVCO'},'out');
Href = Hnoise(1,1);
Hncp = Hnoise(1,2);
```

```
Hnr2 = Hnoise(1,3);
Hnr3 = Hnoise(1,4);
Hnvco = Hnoise(1,5);
```

Plot the transfer functions

```
figure(4);
opts.PhaseVisible = 'off';
bodeplot(Href,Hncp,Hnr2,Hnr3,Hnvco,opts);
title('Pll Noise Transfer Functions');
legend('ref to out','CP to out','R2 to out','R3 to out','VCO to out');
```



Output Noise Spectral Density

Each of the PLL's internal noise sources can have a flicker noise component (1/f) as well as an additive white Gaussian noise (AWGN) component that is flat with frequency. In this section you will model the combination of these two components as an AWGN process driving a single filter with a gain that is unity at high frequencies but increases as 1/f below a predefined corner frequency. The `getFlickerFilter` function supplied with this example provides a useful approximation.

The model for the VCO phase noise n_VCO can be obtained from the measured or specified phase noise of the VCO using the process defined for the MSB Ring Oscillator model. This process fits the measured data to a physical model of the VCO (including flicker noise) while avoiding a couple of inaccuracies present in most oscillator phase noise measurements. While the Ring Oscillator model process produces a period offset standard deviation, the application to phase domain modeling requires that the period offset be scaled to a phase offset.

You can derive the n_VCO model for the time domain simulation used in this example using the following steps.

```
PllFoffset    = [30e3 100e3 1e6 3e6 10e6]; % VCO offset frequencies (Hz)
PllPhaseNoise = [-56 -106 -132 -143 -152]; % VCO output phase noise (dBc/Hz)
CfgRef  = 30e6;          % Reference frequency

f0 = PllN*CfgRef;
[periodjitter,fcorner] = ...
    msblks.VCO.estimatePhaseNoiseCore(f0,PllFoffset,PllPhaseNoise);
phasejitter = 2*pi*f0*periodjitter; % Convert from period jitter to phase jitter
% Scale for the fact that the phase jitter occurs in one VCO output cycle.
n_VCO = ss(0,sqrt(f0)*phasejitter,1,0) * getFlickerFilter(fcorner);
```

You can model the Johnson current noise in the loop filter resistors by multiplying the broadband current noise density times a flicker filter. Due to the high quality of modern manufacturing, the typical corner frequency for a resistor is somewhere between 1kHz and 10kHz.

```
kB = 1.3805e-23; % Boltzmann's constant
temp = 273 + 25; % Temperature, in degrees Kelvin for an ambient of 25C
fc_resistors = 3e3; % Typical resistor flicker corner frequency
n_r2 = sqrt(4*kB*temp/PllR2)*getFlickerFilter(fc_resistors);
n_r3 = sqrt(4*kB*temp/PllR3)*getFlickerFilter(fc_resistors);
```

The CP is an active circuit, and the noise it generates is determined by details of the circuit design and device parameters. Its noise density is best obtained from device data sheets. However at least one version of such data data sheet parameters [1] characterizes the noise at the output of the PLL and not the output of the charge pump.

For the sake of having some sort of model, suppose that

1. In steady state operation, on average one or the other CP output transistor is operating with a duty cycle d. 0.01 < d < 0.1

2. When one of the CP output transistors is on, its $V_{DS} = 0$ channel conductivity g_{ds} is $g_{ds} = \frac{I_{CP}}{A * V_{DD}}$ where $1 <= V_{DD} <= 5$ and the saturation factor $2 < A < 10$.

3. The channel current integration constant is $2/3 <= \gamma < 1.5$

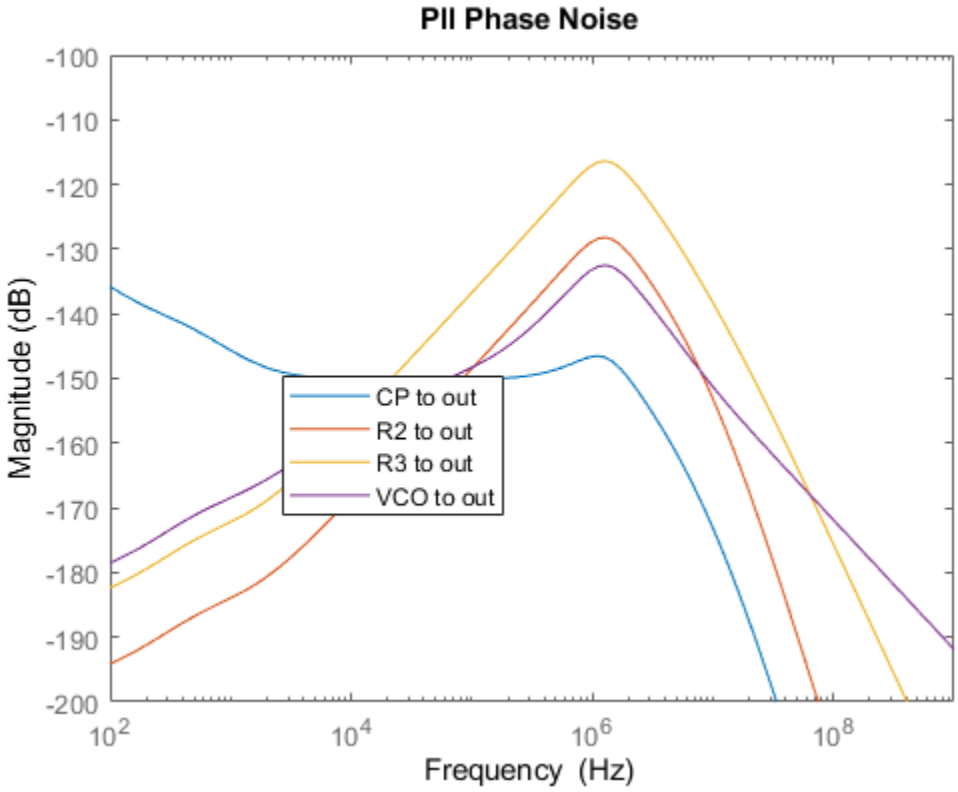
Then the CP current noise spectral density is $\sqrt{4kTd\gamma \frac{I_{CP}}{A * V_{DD}}}$

```
CP_d = 0.03;
CP_A = 4;
CP_VDD = 5;
CP_gamma = 1;
n_CP = sqrt(4*kB*temp*CP_d*CP_gamma*PllKphi/CP_A/CP_VDD)*...
    getFlickerFilter(fc_resistors);
```

Plot the phase noise contributions for all four noise sources.

```
figure(5);
opts.PhaseVisible = 'off';
opts.MagLowerLimMode = 'manual';
opts.MagLowerLim = -200;
opts.XLimMode = 'manual';
```

```
opts.XLim = [100,1e9];
nout_CP = n_CP*Hncp;
nout_R2 = n_r2*Hnr2;
nout_R3 = n_r3*Hnr3;
nout_VCO = n_VCO*Hnvco;
bodeplot(nout_CP,nout_R2,nout_R3,nout_VCO,opts);
title('Pll Phase Noise');
legend('CP to out','R2 to out','R3 to out','VCO to out');
opts.PhaseVisible = 'on';
opts.MagLowerLimMode = 'auto';
opts.XLimMode = 'auto';
```



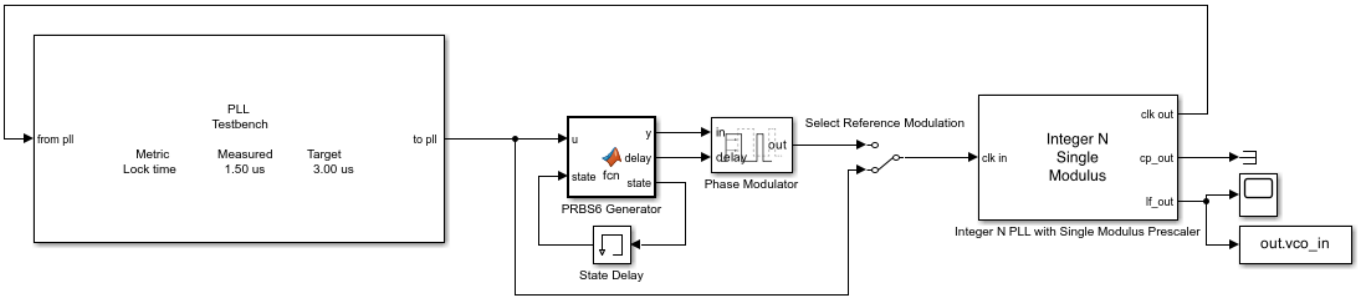
Compare to time domain results

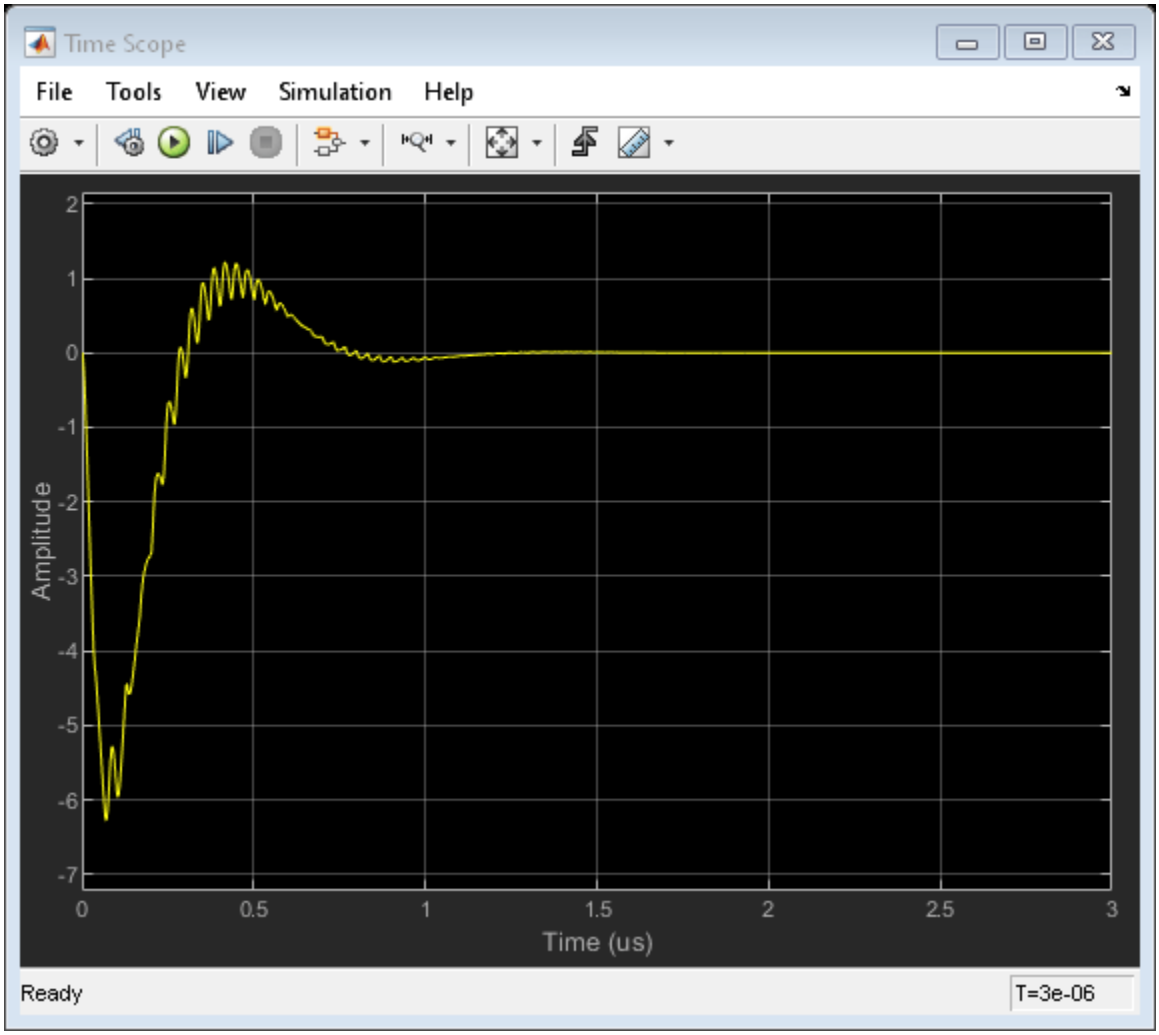
This section compares the phase domain analysis results to the results of a time domain simulation using the model presented in the “Phase Noise at PLL Output” on page 1-2 example. The clearest and most direct approach is to compare the step response predicted by the analysis with the phase lock response produced by the simulation.

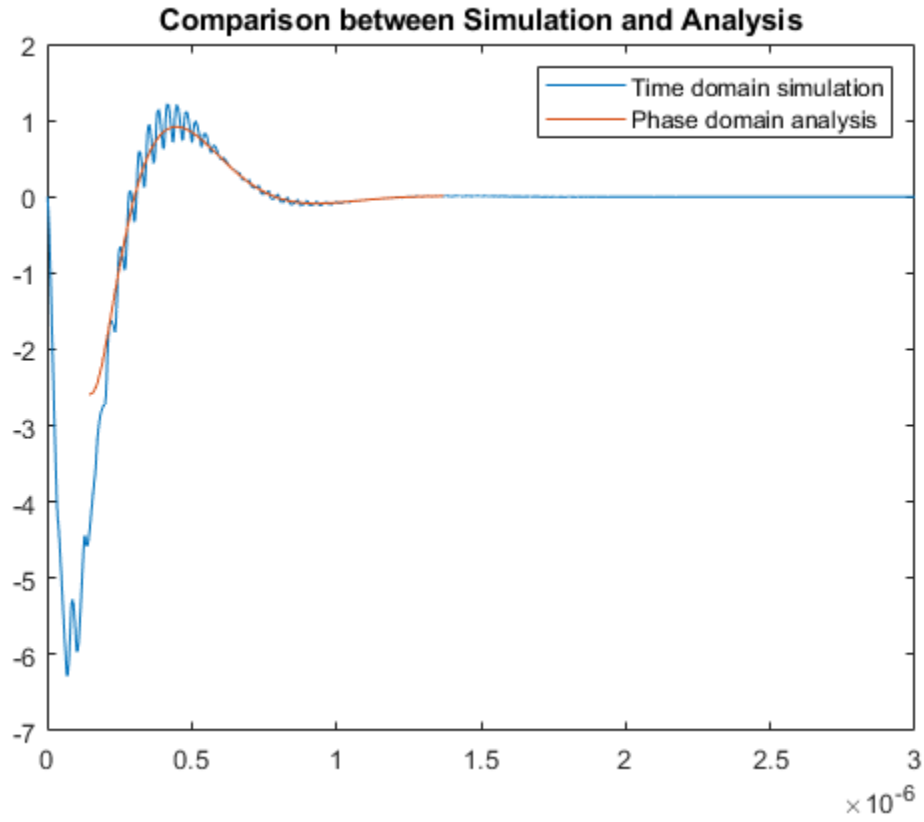
The PllPhaseNoiseExample.slx model included with this example is statically configured for the purposes of this section. The workspace output and a CP output monitor port give you access to additional data when performing follow-on investigations such as are suggested in this section and the Compare Z domain with Laplace domain subsection.

Run the time domain simulation and compare the loop lock response to a manually adjusted copy of the phase domain step response.

```
open_system('PllLoopDynamicsExample.slx');
out = sim('PllLoopDynamicsExample.slx');
figure(6);
plot(out.vco_in.Time,squeeze(out.vco_in.Data));
hold on;
plot(tstep+1.46e-7,0.913/24.7*(ystep-70));
title('Comparison between Simulation and Analysis')
legend('Time domain simulation','Phase domain analysis')
```







The time domain loop lock response agrees quite well with the phase domain analysis result. It is interesting that there is a portion of the time domain response that precedes the phase domain step response. Using features explained in the following paragraph, you can demonstrate that this phenomenon occurs even for very small phase steps, and is therefore a linear response phenomenon.

As an aid to further study, the reference phase modulator in the model has been configured to produce a PRBS6 modulation with a long enough period to capture the entire step response, and a low enough amplitude to assure operation within the linear range. You can explore the resulting behaviors by toggling the switch at the output of the reference phase modulator, and you may also find it helpful to examine the CP output to verify operation in the linear range.

Because of the very high loop bandwidth, some leakage of the high frequency portions of the CP output are clearly visible in the loop filter output. These high frequency perturbations modulate the VCO frequency and cause spurious responses at the output of the PLL. Phase domain analysis does not address these spurious responses.

General Loop Filters

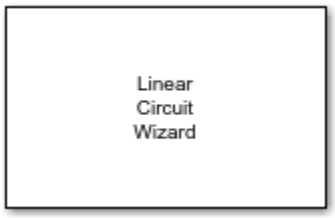
This section demonstrates how to use the Linear Circuit Wizard to obtain ss objects for detailed loop filter designs, including their device noise transfer impedances.

The Linear Circuit Wizard reads a SPICE netlist and then sets up and solves Kirchhoff%s equations algorithmically. Several different outputs are available including a report of poles, zeros, and gains. You will transform these poles, zeros, and gains into an ss object.

While this example supplies a SPICE netlist for the one loop filter that is used throughout the example, you can apply the following procedure to the SPICE netlist for any of a wide variety of circuits.

1. Open the model PllLoopDynamicsLCW.slx.

```
open_system('PllLoopDynamicsLCW.slx');
```



2. Open the mask for the Linear Circuit Wizard.

3. Select the Browse function and then select the SPICE file LoopFilter.sp.

4. Select the Parse netlist and redefine ports function.

5. Go to the Port Definition tab and make sure that the desired input and output ports were configured by the netlist. If they were not configured by the netlist, you can use the Ports tab to add, delete, or modify ports.

6. Go to the Device noise tab and enable the device noise for any potential device noise sources you want to have modeled. In this example, enable the device noise generator for the resistors R2 and R3.

7. Select the Output poles and zeros function. A structure LinearCircuitWizardPoleZero is added to your workspace.

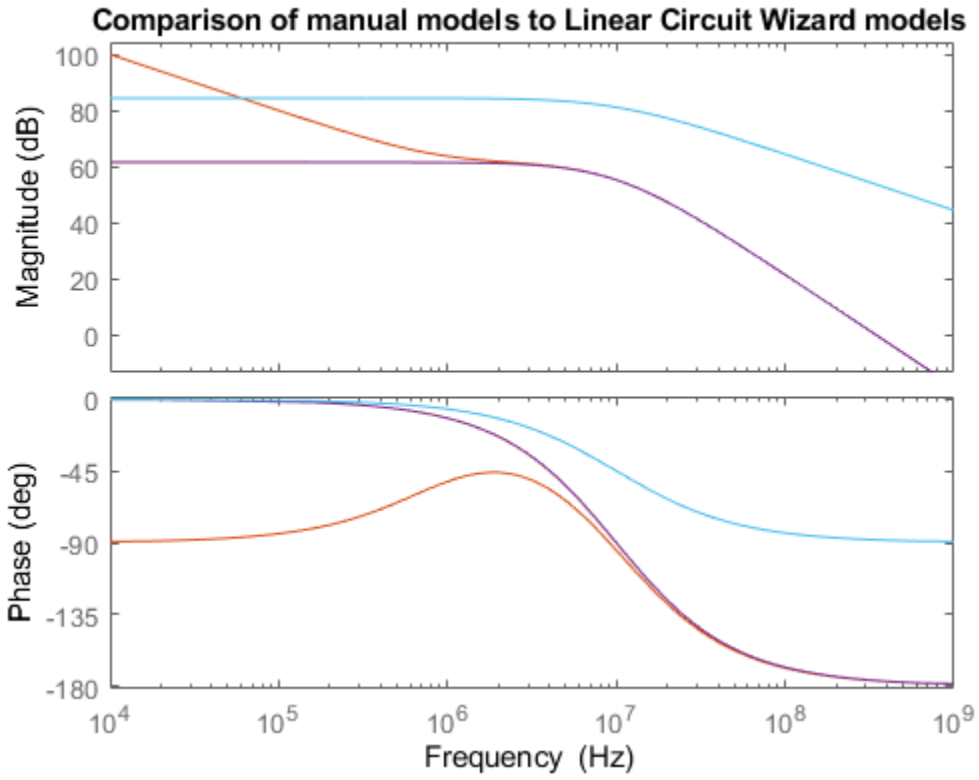
For use in the following transformations, a copy of LinearCircuitWizardPoleZero has also been saved to the file PllLCWPoleZero.mat included with this example.

The following transforms a set of poles, zeros and gains into a separate zpk model and then ss model for each transfer impedance. Note that the Gains are low frequency gains, not including poles or zeros at the origin, while the k factor is based on a different formulation of the transfer function. The `gain2k` function is supplied with this example to aid you in performing the gain to k transformation.

```
load('PllLCWPoleZero.mat');
p = LinearCircuitWizardPoleZero.Poles;
z = LinearCircuitWizardPoleZero.Zeros;
g = LinearCircuitWizardPoleZero.Gains;
k = gain2k(p,z,g);
[A,B,C,D] = zp2ss(z{1},p,k(1)); % Loop filter transfer impedance
zlf_lcw = ss(A,B,C,D);
[A,B,C,D] = zp2ss(z{2},p,k(2)); % R2 transfer impedance
zr2_lcw = ss(A,B,C,D);
[A,B,C,D] = zp2ss(z{3},p,k(3)); %  R3 transfer impedance
zr3_lcw = ss(A,B,C,D);
```

Use bode plots to demonstrate that the transfer impedance models produced through the Linear Circuit Wizard are identical to those produced through direct symbolic expressions.

```
figure(7)
bodeplot(zlf,zlf_lcw,zr2,zr2_lcw,zr3,zr3_lcw,opts);
title('Comparison of manual models to Linear Circuit Wizard models');
```



Z Domain Modeling

This section presents modeling in the uniform sampled time domain (Z domain) as an alternative to modeling in the Laplace domain.

The PFD/CP only samples the phase difference between reference and output once every reference cycle [2, 3]. Thus the dynamics of the PLL are inherently tied to that uniform sample rate, and the mathematics that rigorously applies to a linear control loop analysis is Z domain mathematics. There are important behaviors such as spurious responses that occur at higher frequencies. However linear mathematics does not apply to those behaviors.

Given a Z domain expression $Z_{LF}(z)$ of the loop filter response, the Z domain expression for the loop gain [3] is

$$G(s) = \frac{\tau_{ref} I_{CP} K_{VCO}}{1 - z^{-1}} Z_{LF}(z)$$

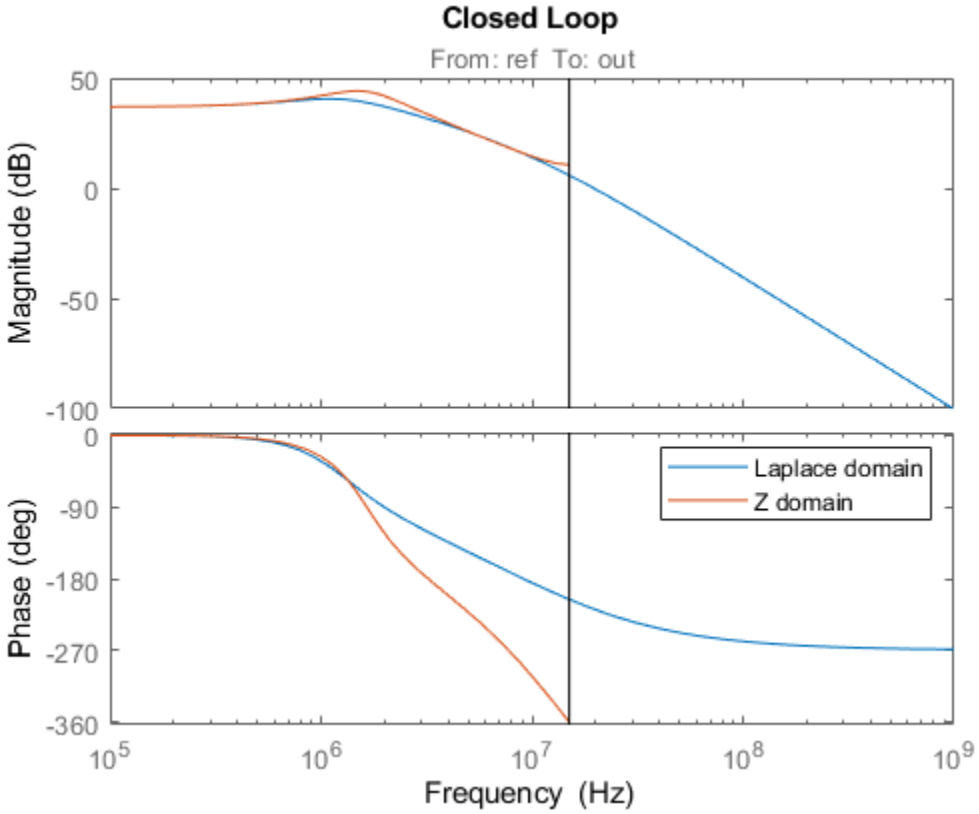
The remaining task is to convert the $Z_{LF}(s)$ function from the Laplace domain analysis to the Z domain. There are many ways to perform this task, and several of them have been implemented in the MATLAB `c2d` function. The appropriate option in this application is zero-order hold, the default for the `c2d` function. Note that the Z domain modeling does require an explicit sample time value equal to the period of the reference signal.

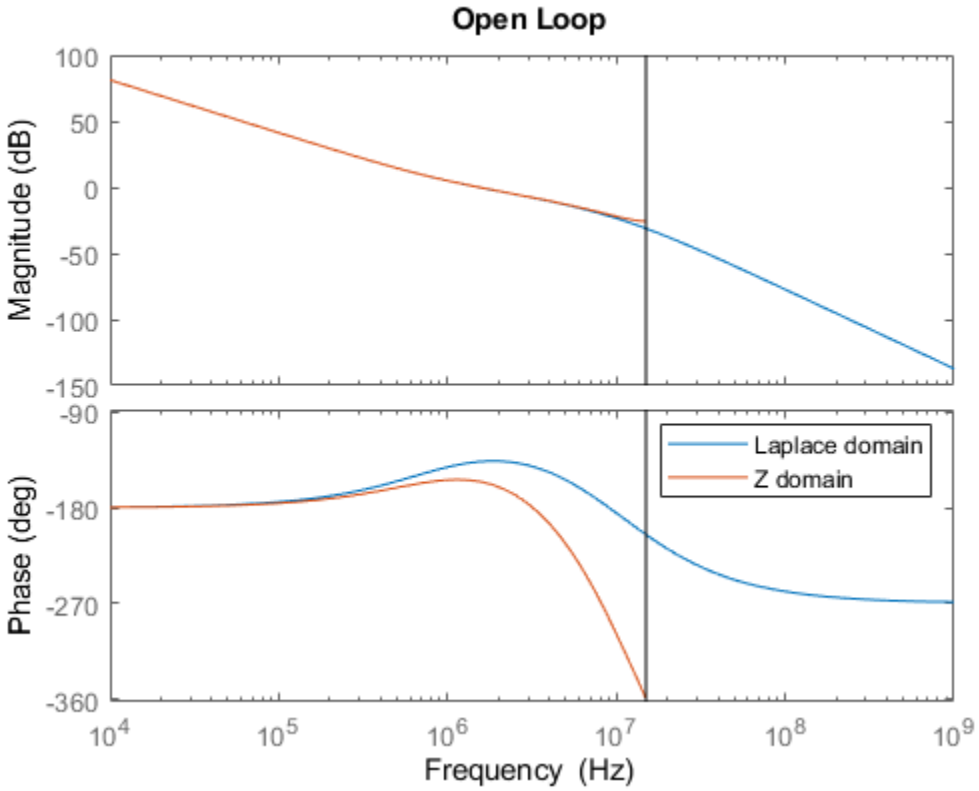
```
fref = 30e6;
ts = 1/fref;
vcoz = ss(1,ts*PllKphi*PllKvco,1,0,ts);
divNz = ss(0,0,0,1/PllN,ts);
zlfz = c2d(zlf,ts);
Gz = vcoz*zlfz;

Hz = feedback(Gz,divNz);
Hz.InputName  = 'ref';
Hz.OutputName = 'out';
GzoverN = Gz*divNz;
```

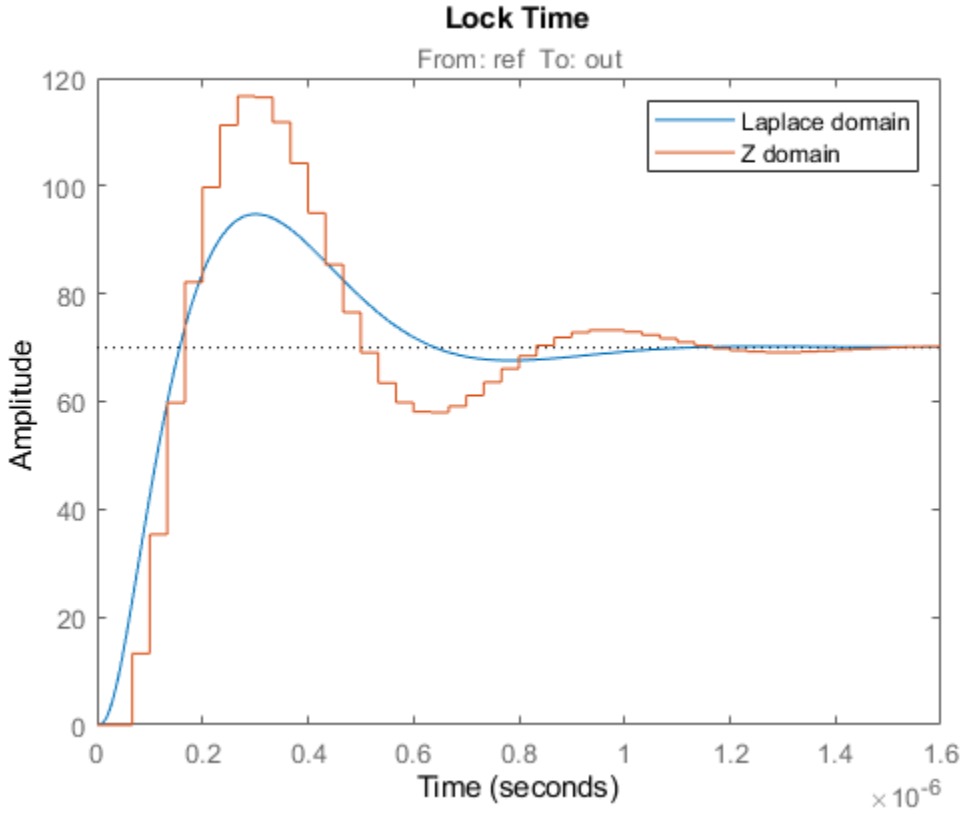
Compare Z domain with Laplace domain

It is informative to compare the Z domain predictions of the closed loop, open loop, and step responses with the corresponding results from the Laplace domain analysis.

```
figure(1);
bodeplot(H,Hz,opts);
title('Closed Loop');
legend('Laplace domain','Z domain');

figure(2);
bodeplot(GoverN,GzoverN,opts);
title('Open Loop');
legend('Laplace domain','Z domain');

figure(3);
step(H,Hz);
title('Lock Time');
legend('Laplace domain','Z domain');
```


Closed Loop
From: ref To: out
Magnitude (dB)
50
0
-50
-100
Phase (deg)
0
-90
-180
-270
-360
10^5
10^6
10^7
10^8
10^9
Frequency (Hz)
Laplace domain
Z domain


Open Loop
Magnitude (dB)
100
50
0
-50
-100
-150
Phase (deg)
-90
-180
-270
-360
Laplace domain
Z domain
10^4
10^5
10^6
10^7
10^8
10^9
Frequency (Hz)



In this comparison, note that while the frequency response and step response from the Z domain analysis are very similar to those obtained from the Laplace domain analysis, the phase margin predicted by the Z domain analysis is far less than the design value chosen as part of the design in the Laplace domain. This is a critical observation, as the loop phase margin is a fundamental requirement of the control loop design.

As stated in [2], this is a typical result for PLLs whose loop bandwidth equals or exceeds one tenth of the reference frequency. For much smaller loop bandwidths the difference between Z domain and Laplace domain is much smaller.

Note, however, that it is the Laplace domain analysis result that closely matches the time domain simulation. You might find this to be a suitable topic for further study.

Advantages and Disadvantages of Phase Domain Modeling

+ Both open and closed loop linear control analyses are easy, fast, reliable and accurate.

+ You can readily include detailed phase noise modeling in your analysis, with the effect of each phase noise source indicated clearly and accurately.

+ Using the Linear Circuit Wizard, you can easily and accurately include detailed linear circuit designs, especially for the loop filter.

- Phase domain modeling is limited to one sample per reference cycle, and therefore a Nyquist frequency limit of one half the reference frequency. For example, it is not applicable to the analysis of spurious output responses past this Nyquist limit.

- While phase domain modeling can be applied to nonlinear behaviors such as frequency acquisition, such applications lose the inherent advantages of linear control analysis associated with phase domain modeling.

- Phase domain modeling is usually not applicable to models that combine the PLL with other circuits that use the output of the PLL.

References

1. Dean Banerjee, PLL Performance, Simulation and Design, 5th edition, Texas Instruments, SNAA106C, May 2017.

2. P. K. Hanumolu, M. Brownlee, K. Mayaram and Un-Ku Moon, "Analysis of charge-pump phase-locked loops," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 51, no. 9, pp. 1665-1674, Sept. 2004, doi: 10.1109/TCSI.2004.834516.

3. Mike Steinberger, "Discrete Time Modeling of PLLs", MathWorks document, May 31, 2022.

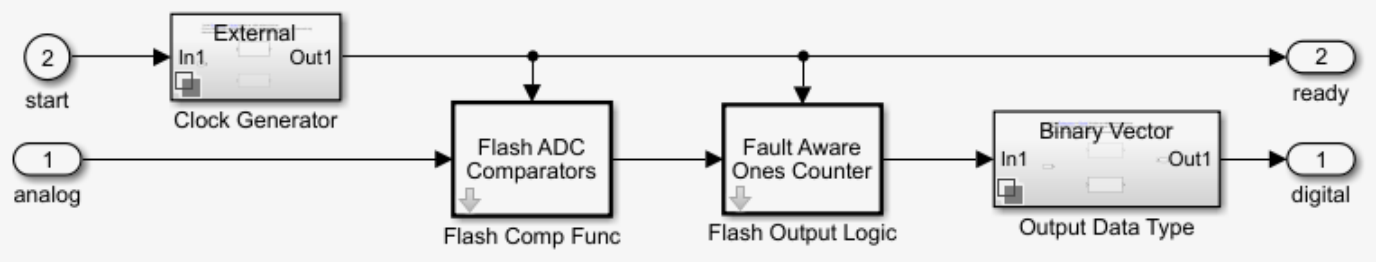
DC Featured Examples

Effect of Metastability Impairment in Flash ADC

This example shows how to customize a flash Analog to Digital Converter (ADC) by adding the metastability probability as an impairment. You can measure the metastability probability impairment to validate your implementation. The example also shows the effect of metastability on the dynamic performance of the flash ADC. When the digital output from a comparator is ambiguous (neither zero nor one), the output is defined as metastable. The ambiguous output is expressed as `NaN`. This example model uses a MATLAB function block to add the metastability impairment to a flash ADC architecture. Another subsystem reports the metastability probability on the fly.

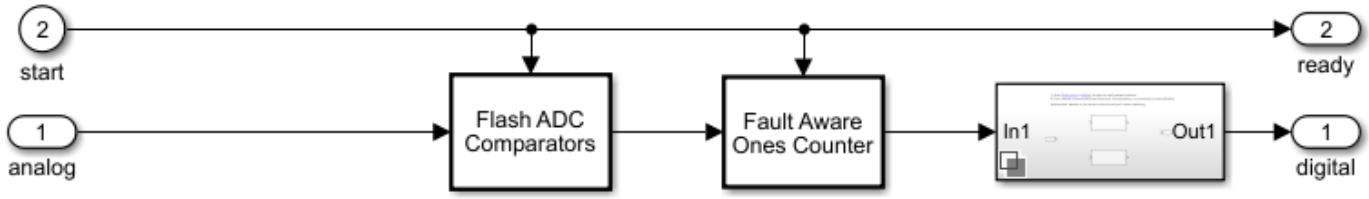
Customize Flash ADC

Extract the inner structure of the flash ADC to add customized impairment. Add a Flash ADC block from the Mixed-Signal Blockset™ library to a Simulink® canvas. Look under the mask to find the flat structure of the ADC. Copy and paste the complete structure to another new blank canvas.



Delete the Clock Generator block because it is not used to provide the start conversion clock. An external Stimuli subsystem is used for that purpose. The flash ADC now consists of three major components:

- Flash ADC Comparators
- Fault Aware Ones Counter
- Output Data Type



Flash ADC Comparators

An *N*-bit flash ADC uses 2^{Nbits} comparators in parallel. The Flash ADC Comparators subsystem itself is based on MATLAB® code. Before the simulation starts, the comparators calculate the individual reference voltages and store them in a vector. On every specified edge, the input is compared to the references using MATLAB's ability to compare vectors. This generates thermometer code similar to the real flash ADC, without the lag from *N* individual comparator blocks in the model.

To create a 10- bit ADC, set **Number of bits (nbits)** to `10`, **Input Range** to `[-1 1]`, and **INL Vector** to `0`. **Trigger type** is kept at its default value `Rising edge`.

Fault Aware Ones Counter

The Fault Aware Ones Counter subsystem implements the impairments in the flash ADC architecture. Real ADCs handle conversion from thermometer to binary through logic circuits. This subsystem takes the sum-of-elements of the vector stored by the comparators and applies that sum to a lookup table to simulate missing codes, otherwise known as bubbles.

Set the Fault Aware Ones Counter parameters: **Number of Bits (nbits)** to `10`, **Input Range** to `[-1 1]`, and **Bubble Codes** to `[]`. **Trigger type** is kept at its default value `Rising edge`.

Output Data Type

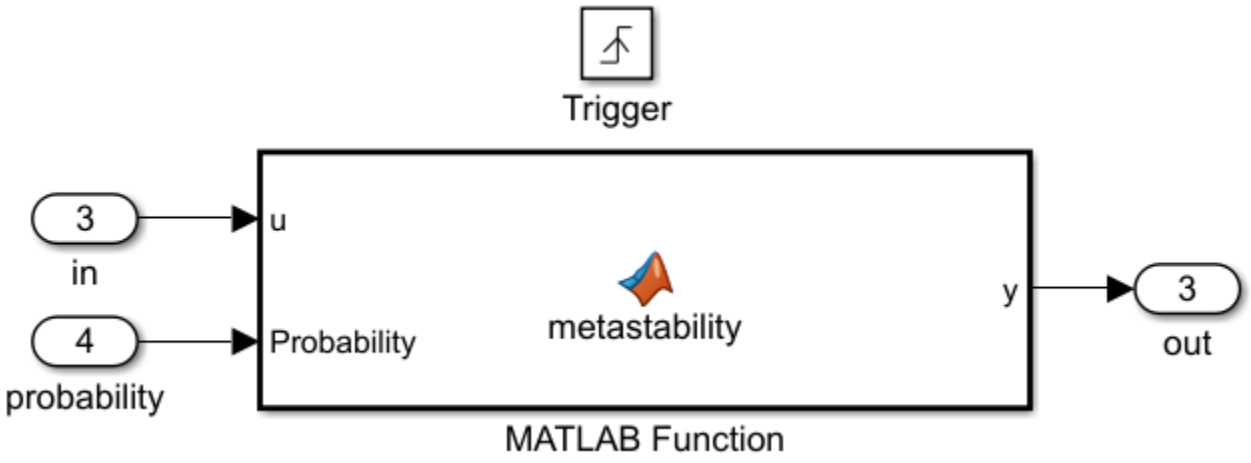
The Output Data Type subsystem handles conversion from the data type at the output of the Fault Aware Ones Counter to the data type specified on the mask of the Flash ADC.

Break the library link between the Output Data Type block and its reference library. Set **Input dynamic range** to `[-1 1]` and **Bipolar data type** to `double`.

Implement Metastability Probability as an Impairment to Flash ADC

To add metastability impairment, place a triggered subsystem with a MATLAB function block after the Flash ADC Comparators subsystem. The MATLAB function block sets thermometer code signals to `NaNs` with a probability from a uniform random number generator. The block resets the signals on the next relevant edge which is why a triggered subsystem is used. Use this code to implement the Metastability Impairment subsystem.

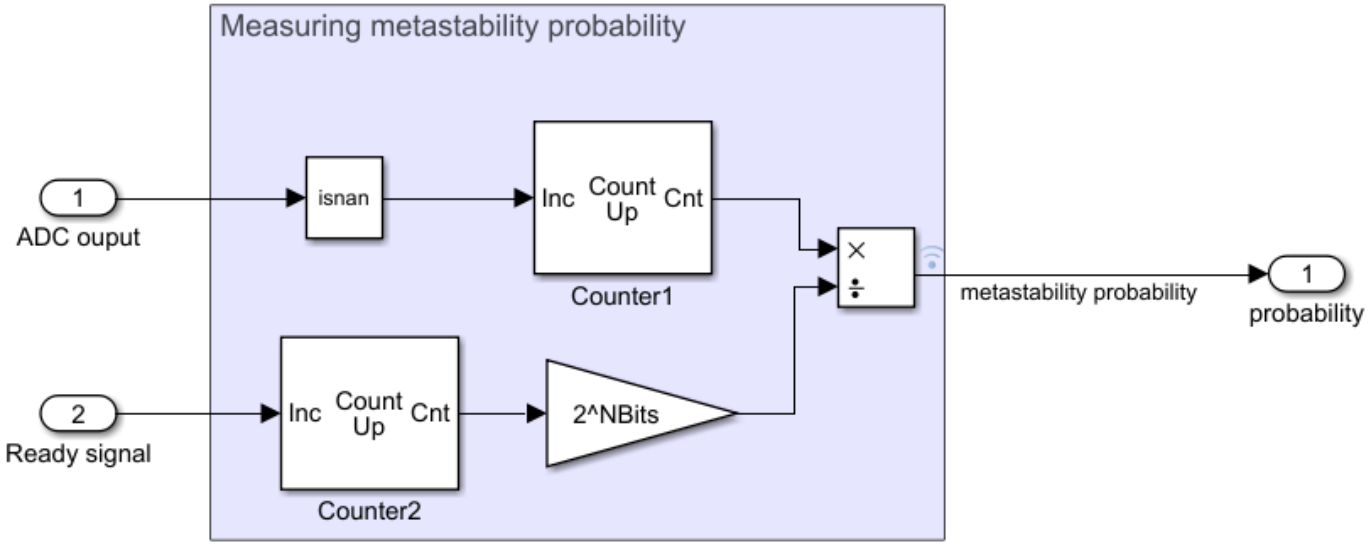
```
%  function y = metastability(u, Probability)
%   mult = ones(size(u));
%   mult(rand(size(u)) < Probability(1)) = NaN; % metastability = NaN
%   y = u .* mult;
%  end
```



Provide the metastability probability that you want to implement through a constant block connected to the **Probability** port.

Implement Measuring Metastability Probability

To measure metastability impairment, count the number of `NaNs` encountered and divide that by the number of total comparator outputs generated during the complete simulation. A simple Simulink implementation of metastability probability measurement is:



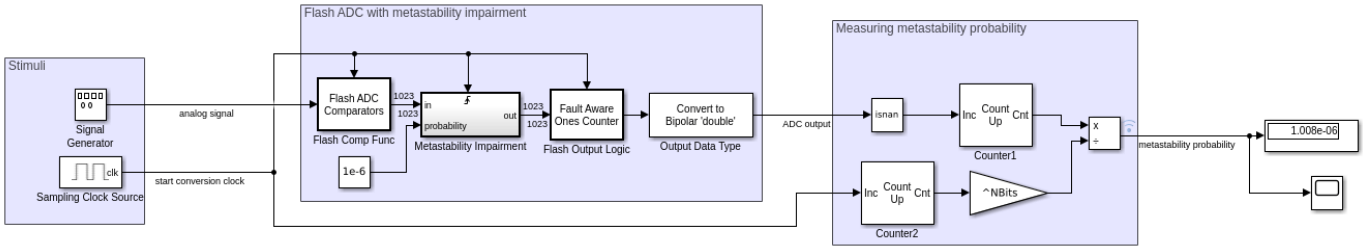
The Inports are:

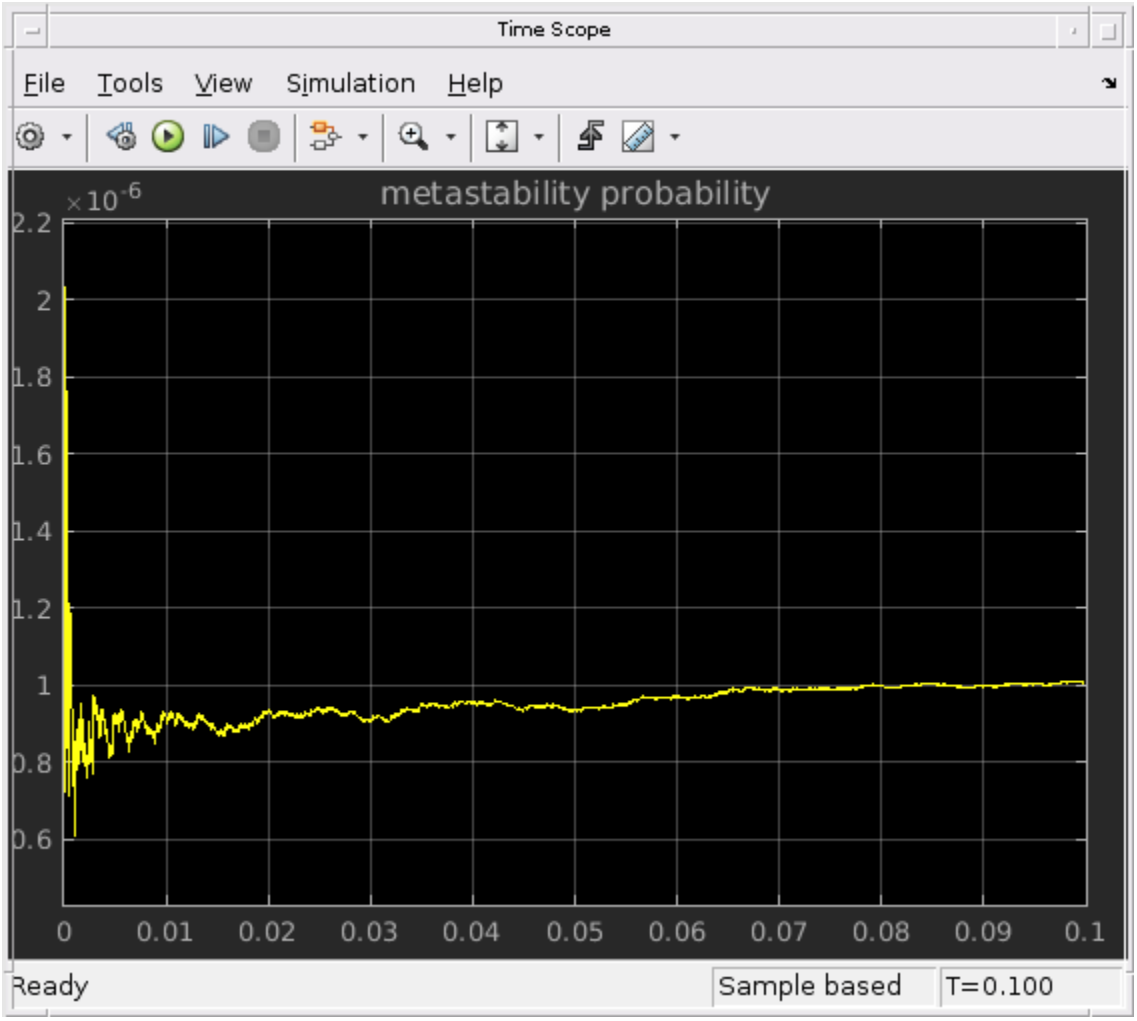
- ADC output- Receives the output digital code generated by the flash ADC.
- Ready signal- Receives the ready signal which represents the rate at which the digital conversion is taking place. The digital code gets generated at each rising edge of the signals received by 'Ready signal' port.

Simulation for Metastability Measurement

The model below combines the customized flash ADC with its output connected to the metastability probability measurement system. In the model, you have a 10-bit flash ADC with metastability probability of 1e-6 added. The Stimuli subsystem generates an analog signal of 100 Hz and a start conversion clock with a frequency of 100 MHz. The ADC operates at the rate defined by the start conversion clock frequency. A dashboard scope provides the behavior of the probability number over time. A display block shows the current probability being measured by the subsystem. You must run the simulation for a long enough period to see the probability number settled at the desired value, in this case 1e-6.

```
NBits=10;
model1='flashAdc_metastability.slx';
open_system(model1);
open_system([bdroot,'/Time Scope']);
sim(model1);
```

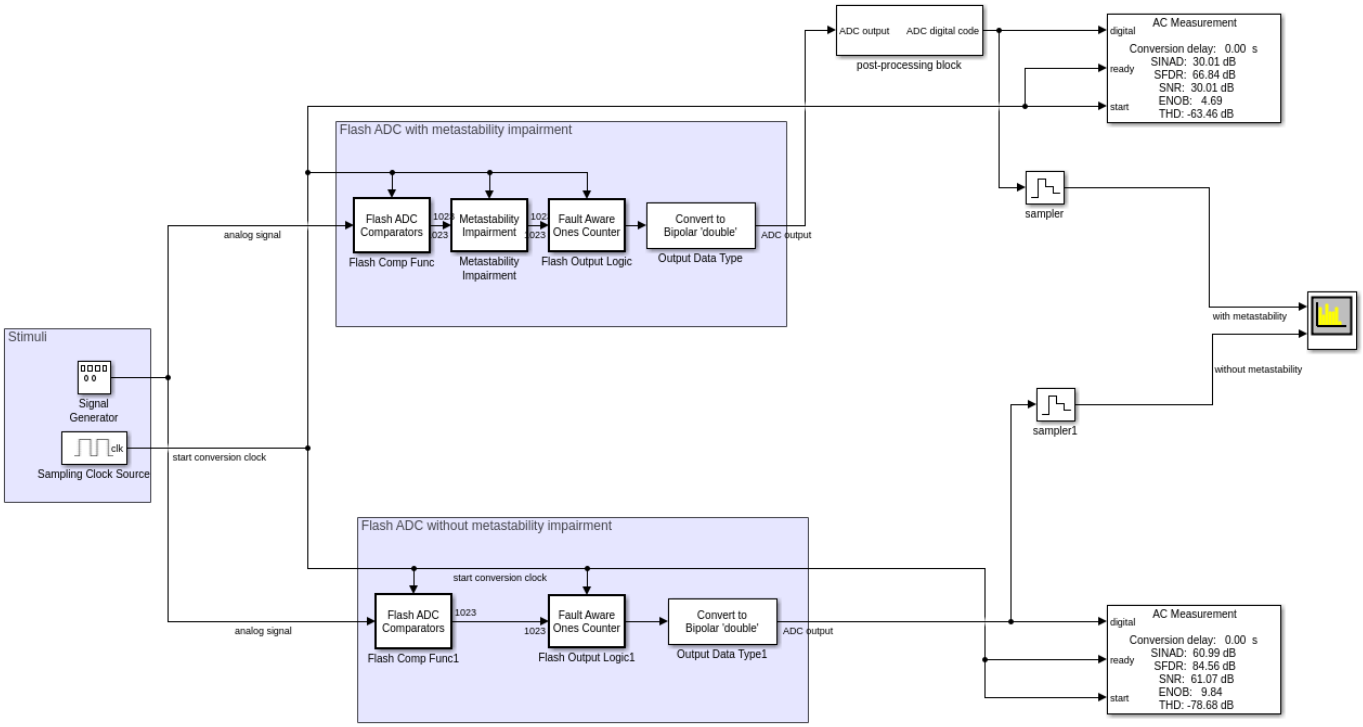


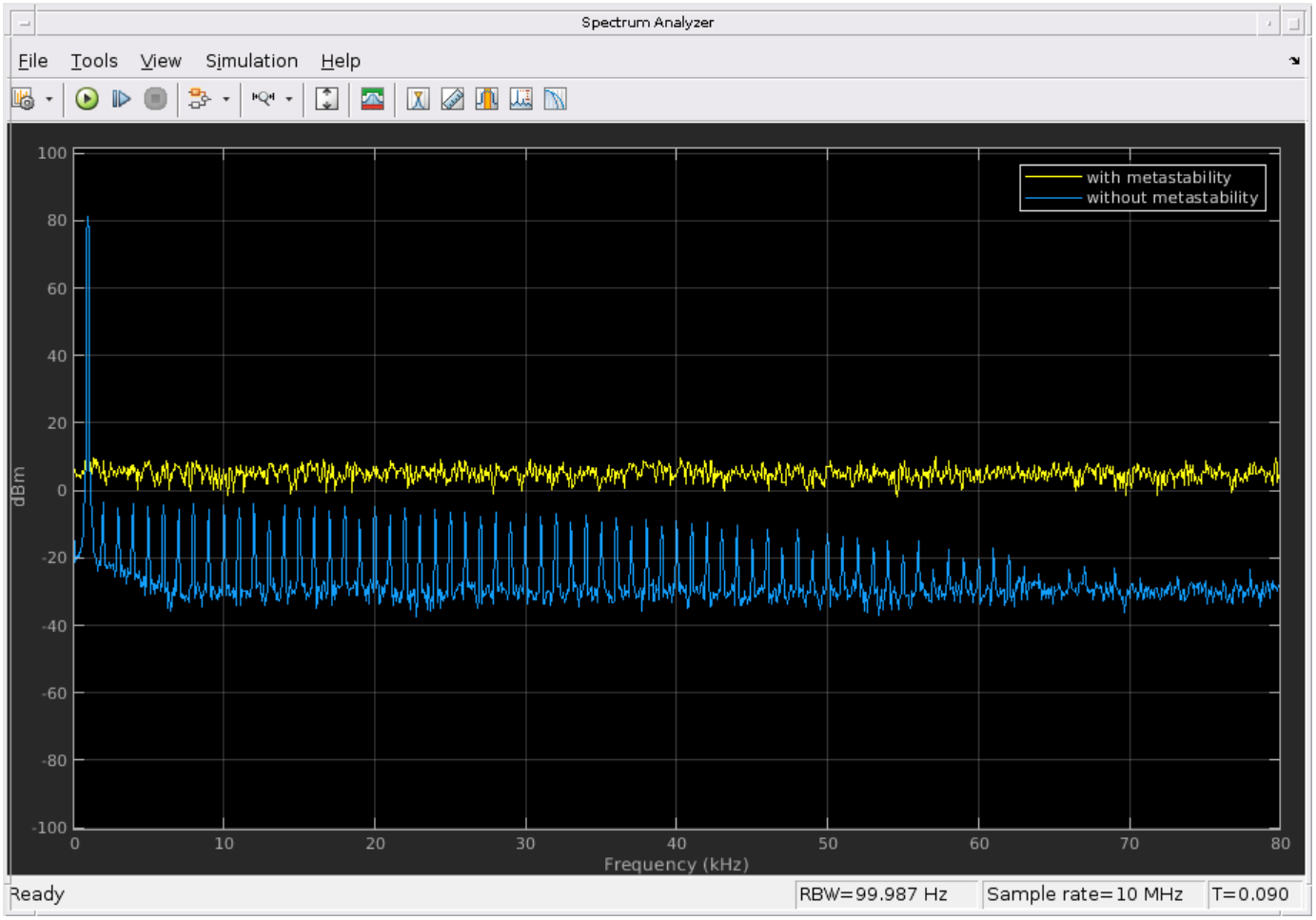


Effect of Metastability on Dynamic Performance of ADC

You can observe the effect of metastability on the dynamic performance of ADCs. The model shows two setup of flash ADC systems: one with metastability and the other without. A postprocessing block that takes in the impaired digital output and converts the `NaNs` to zeros. This is because the digital output with `NaNs` cannot be recognized by a spectrum analyzer as valid signal for spectral analysis. Attach an ADC AC measurement block to observe various performance metrics like SNR, ENOB, noise floor and so on. The simulation results show the AC analysis causes a significant drop in performance for ADC with metastability, as shown by the lower ENOB and higher noise floor.

```
model2='flashAdc_metastability_Effect.slx';
open_system(model2);
sim(model2);
```





See Also

Flash ADC | Sampling Clock Source | ADC AC Measurement

More About

- “Analyzing Simple ADC with Impairments” on page 2-27
- “Compare SAR ADC to Ideal ADC” on page 2-48

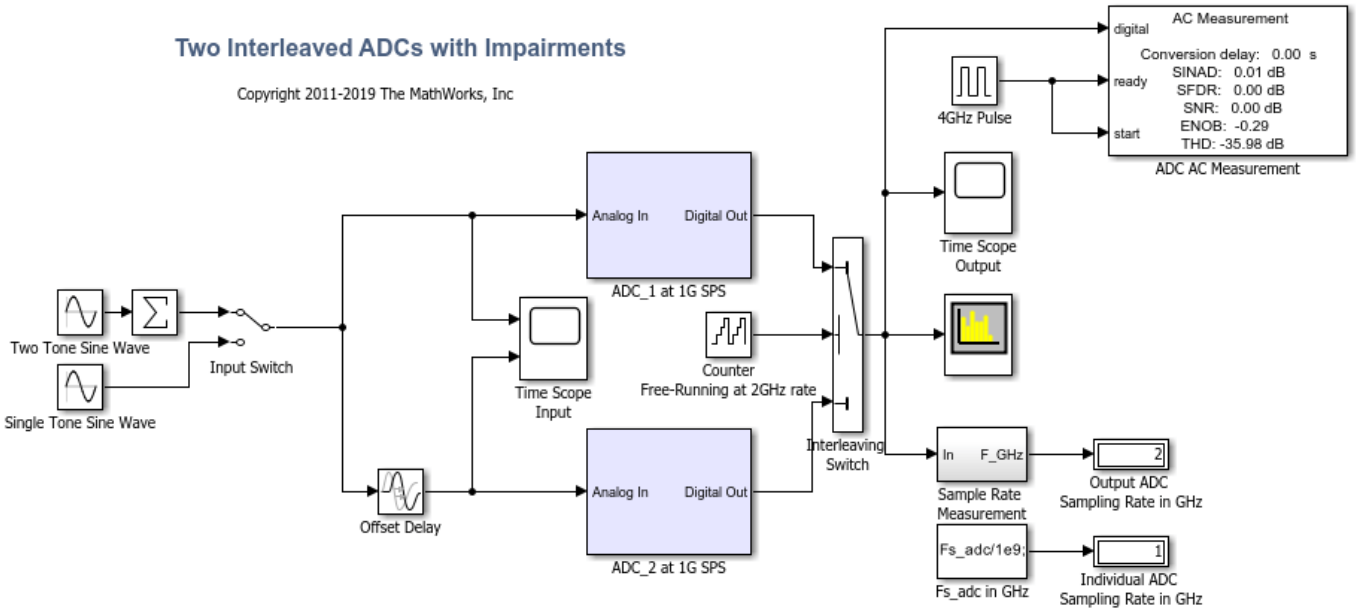
Design and Evaluate Interleaved ADC

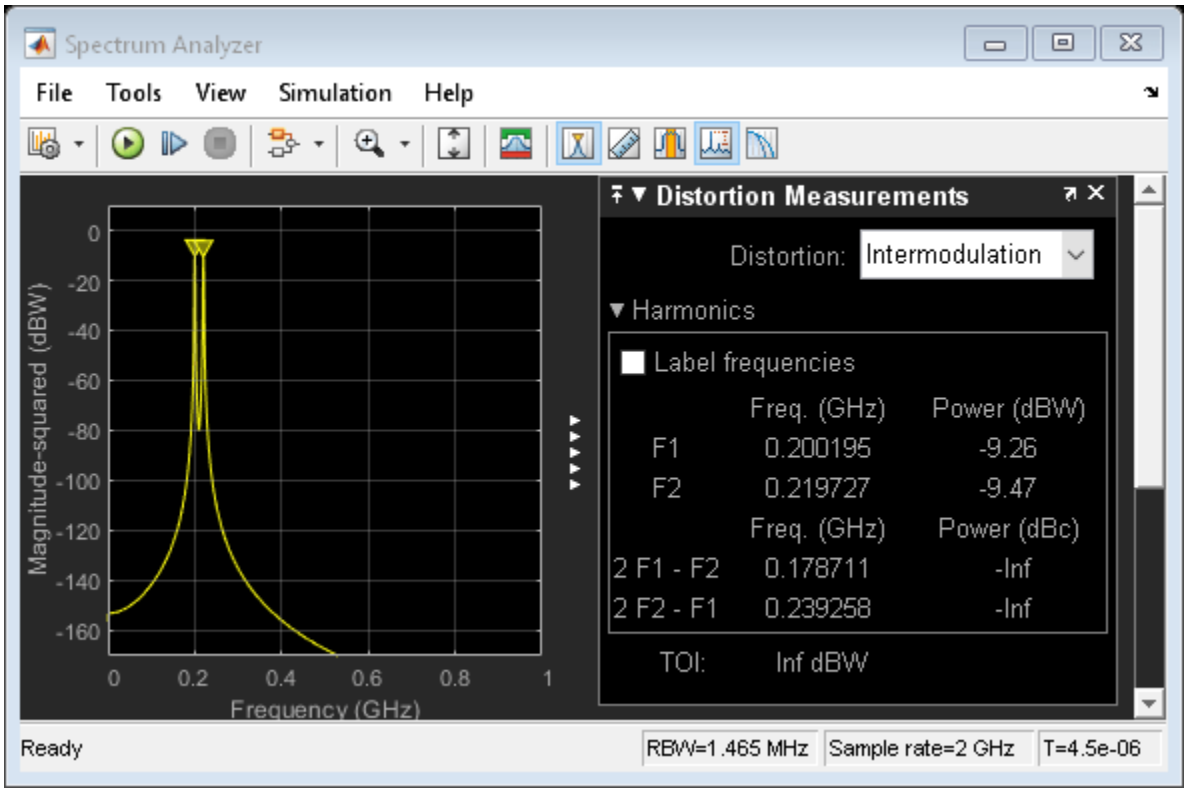
This interleaved ADC model highlights some of the typical impairments introduced by data converters and their effects on a larger system.

Model

In this example, interleave two simple ADCs based on the model “Analyzing Simple ADC with Impairments” on page 2-27 to create the equivalent of one ADC operating at 2X the individual ADC sampling rate. Use a two-tone test signal at `200` MHz and `220` MHz as the input to verify the distortion introduced by the ADC operation.

```
model = 'interleaved_adc';
open_system(model)
set_param([model '/ADC_1 at 1G SPS'],'jitter','off');
set_param([model '/ADC_1 at 1G SPS'],'nonlinearity','off');
set_param([model '/ADC_1 at 1G SPS'],'quantization','off');

set_param([model '/ADC_2 at 1G SPS'],'jitter','off');
set_param([model '/ADC_2 at 1G SPS'],'nonlinearity','off');
set_param([model '/ADC_2 at 1G SPS'],'quantization','off');

set_param([model '/Offset Delay'],'DelayTime','.5/Fs_adc');
set_param([model '/Two Tone Sine Wave'],'Amplitude','.5');
set_param([model, '/Input Switch'], 'sw', '1');

sim(model)
```





To bypass the impairments, use appropriate switch positions inside the ADC blocks. The ADC behavior is purely ideal. The two ADCs in the top-level model are identical with the exception that the noise generators in each ADC have different seeds to make the noise uncorrelated.

Each ADC operates at 1 GHz rate, set by the MATLAB® variable *Fs_adc* defined in the initialization callback of this model. The operating rate of the ADCs is indicated by the green signals and blocks in the diagram. The input signal of the second ADC is delayed by an amount equal to half a period of the ADC sampling frequency.

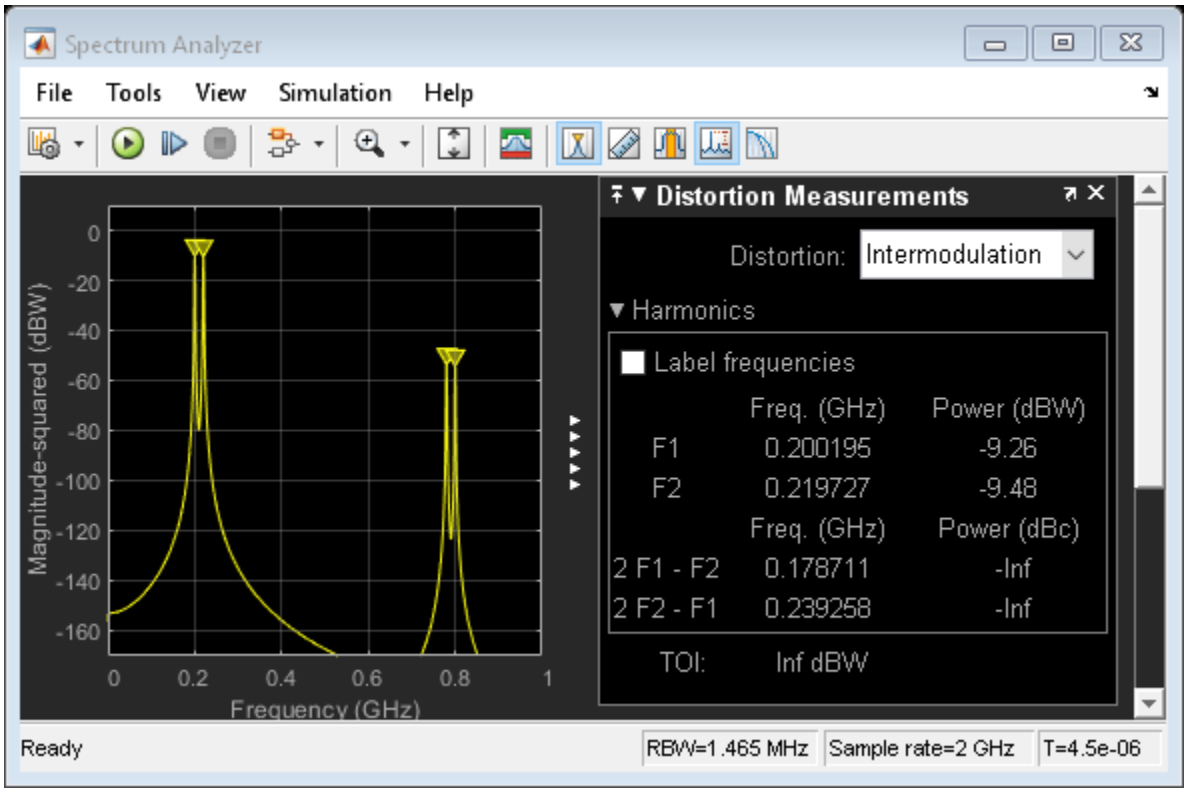
Timing Imperfection

The precision of the timing between the individual ADCs is critical. To see the effect of a timing mismatch, open the Offset Delay block and simply add `10` ps to the **delay** value.

```
set_param([model '/Offset Delay'],'DelayTime','.5/Fs_adc + 10e-12');
```

The `10` ps error causes a significant degradation of the ADC performance, even though both ADCs are perfectly ideal. To compensate for the performance degradation, some form of drift compensation is necessary. For more information, see Time-Interleaved ADC Error Correction.

```
sim(model)
```



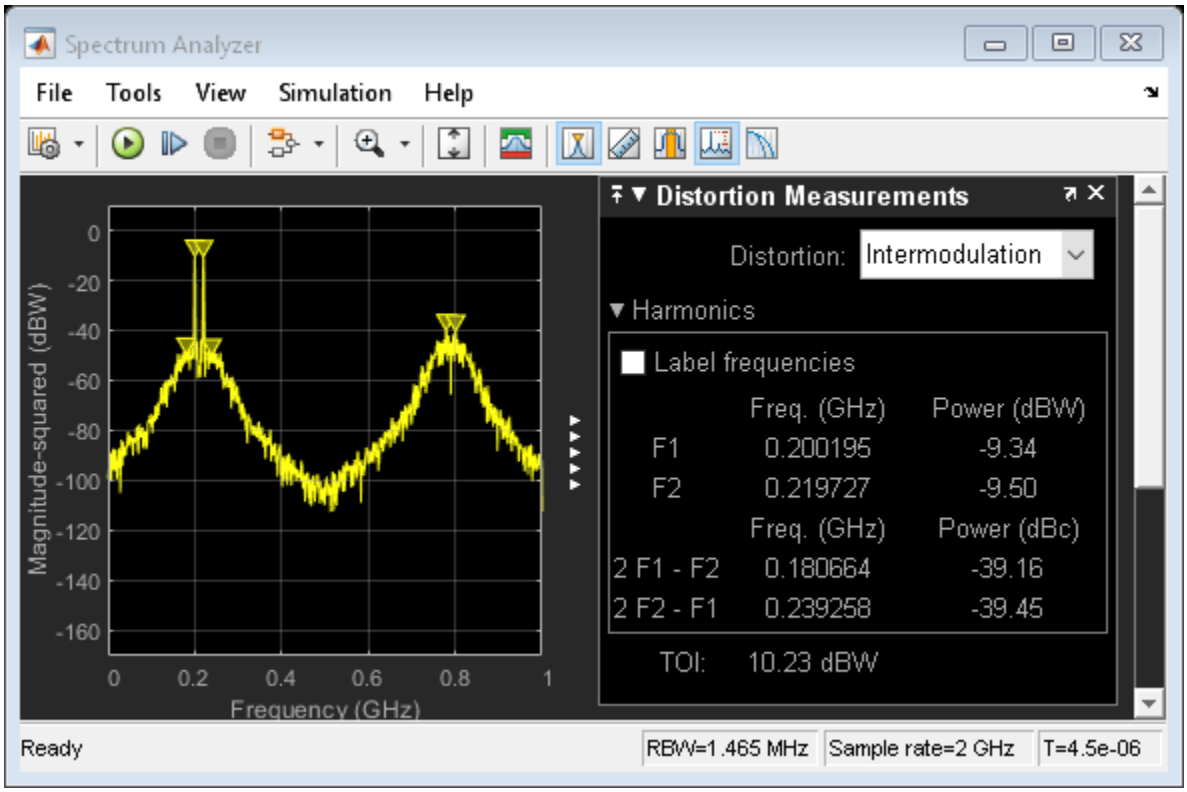
Effect of Aperture Jitter

Remove the fixed offset of `10` ps and enable the aperture jitter impairment in each of the ADC subsystems.

```
set_param([model '/Offset Delay'],'DelayTime','.5/Fs_adc');
set_param([model '/ADC_1 at 1G SPS'],'jitter','on');
set_param([model '/ADC_2 at 1G SPS'],'jitter','on');
```

The noise around the two-tone test signal at `200` MHz is expected, as a direct result of the ADC jitter. The additional noise around `800` MHz is the result of interleaving two uncorrelated noise sources.

```
sim(model)
```



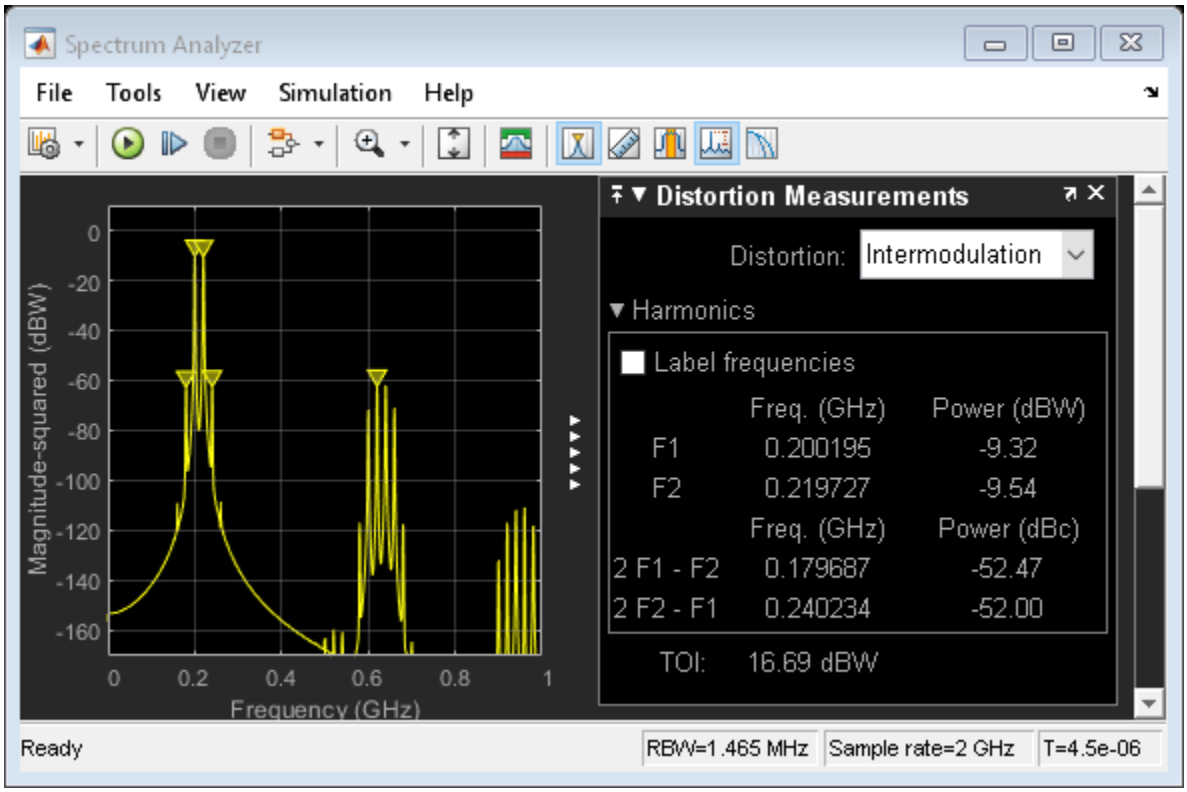
Effect of Nonlinearity

Remove the jitter impairment and activate the nonlinearity impairment in both ADCs.

```
set_param([model '/ADC_1 at 1G SPS'],'jitter','off');
set_param([model '/ADC_2 at 1G SPS'],'jitter','off');
set_param([model '/ADC_1 at 1G SPS'],'nonlinearity','on');
set_param([model '/ADC_2 at 1G SPS'],'nonlinearity','on');
```

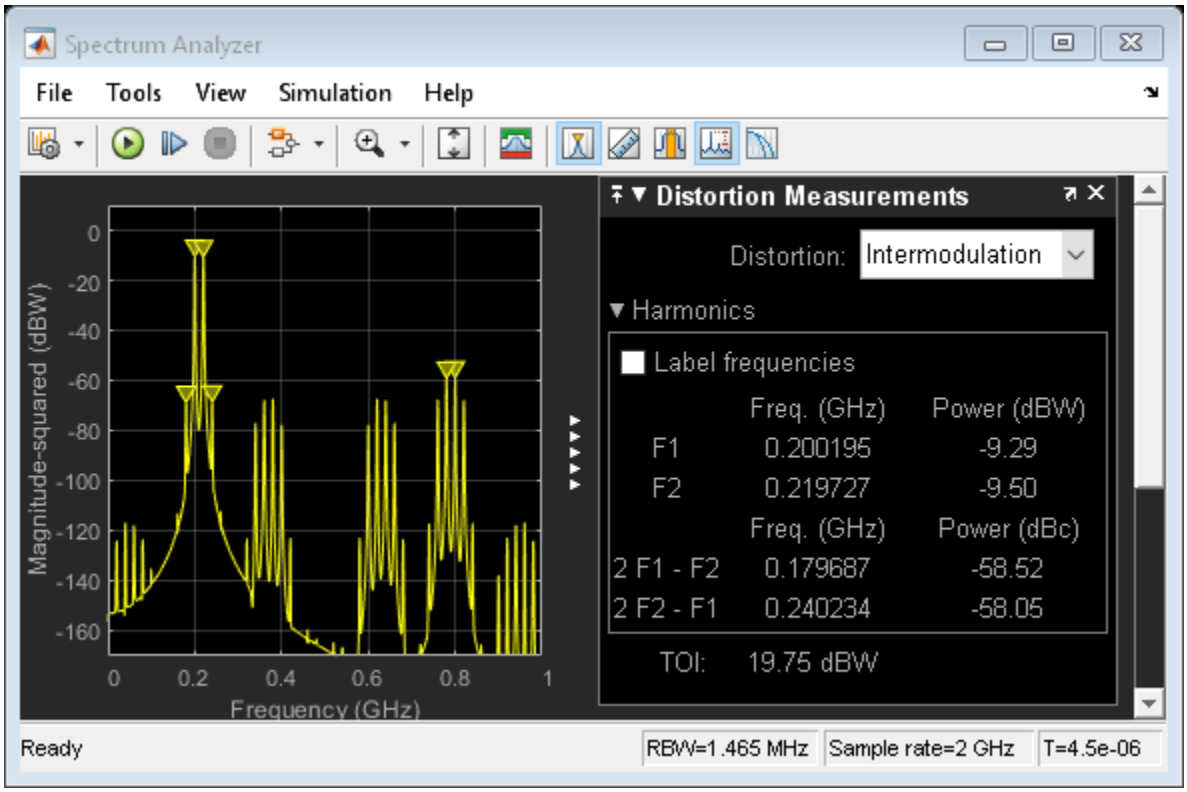
The spectrum now shows 3rd order IMD products around two tones and harmonically related spurs around the `600` MHz region.

```
sim(model)
```



Even though the ADC nonlinear effects are identical and create exactly the same odd order components, there is actually some cancellation of terms. If just one nonlinearity is enabled, the resulting spectrum is worse than when both ADCs are nonlinear.

```
set_param([model '/ADC_2 at 1G SPS'],'nonlinearity','off');
sim(model)
```



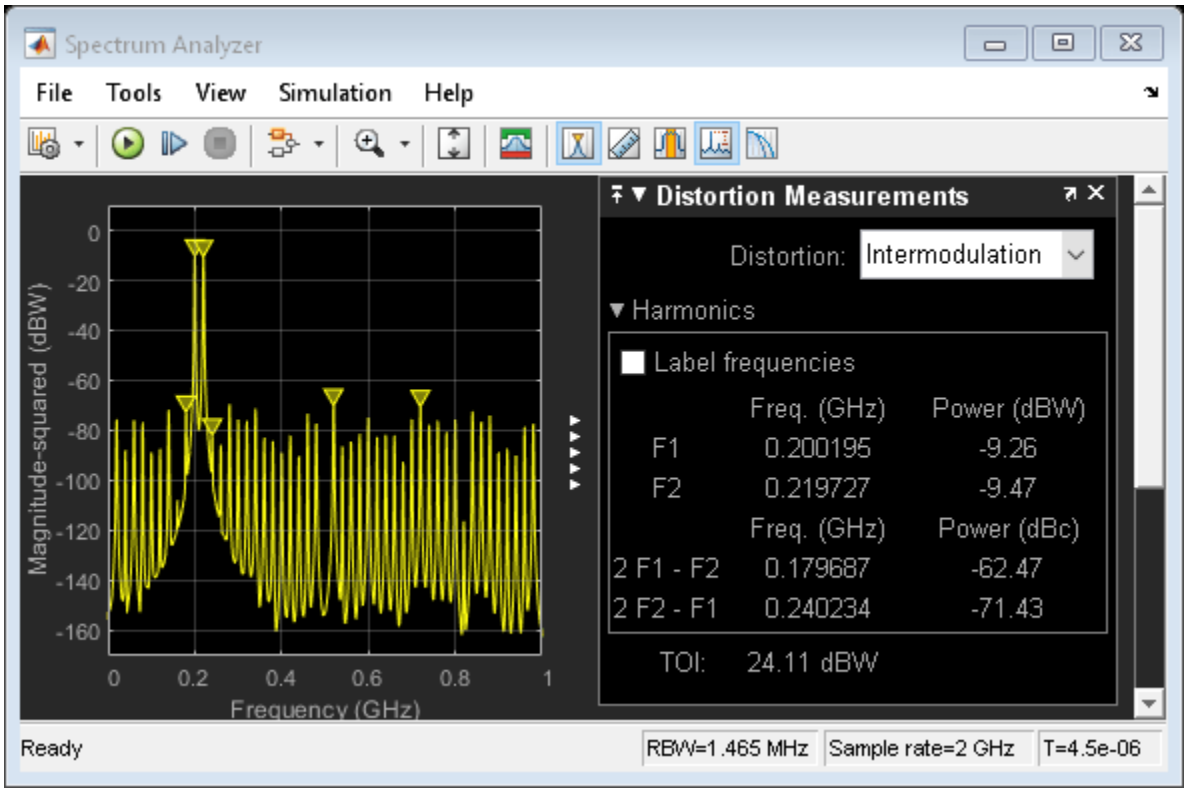
Effect of Quantization and Saturation

Remove the linearity impairment and activate the quantization. The quantizer is set to 9 bits, and the signal level is close to the full scale of +/-1, which can be seen in the input Time Scope.

```
set_param([model '/ADC_1 at 1G SPS'],'nonlinearity','off');
set_param([model '/ADC_2 at 1G SPS'],'nonlinearity','off');
set_param([model '/ADC_1 at 1G SPS'],'quantization','on');
set_param([model '/ADC_2 at 1G SPS'],'quantization','on');
```

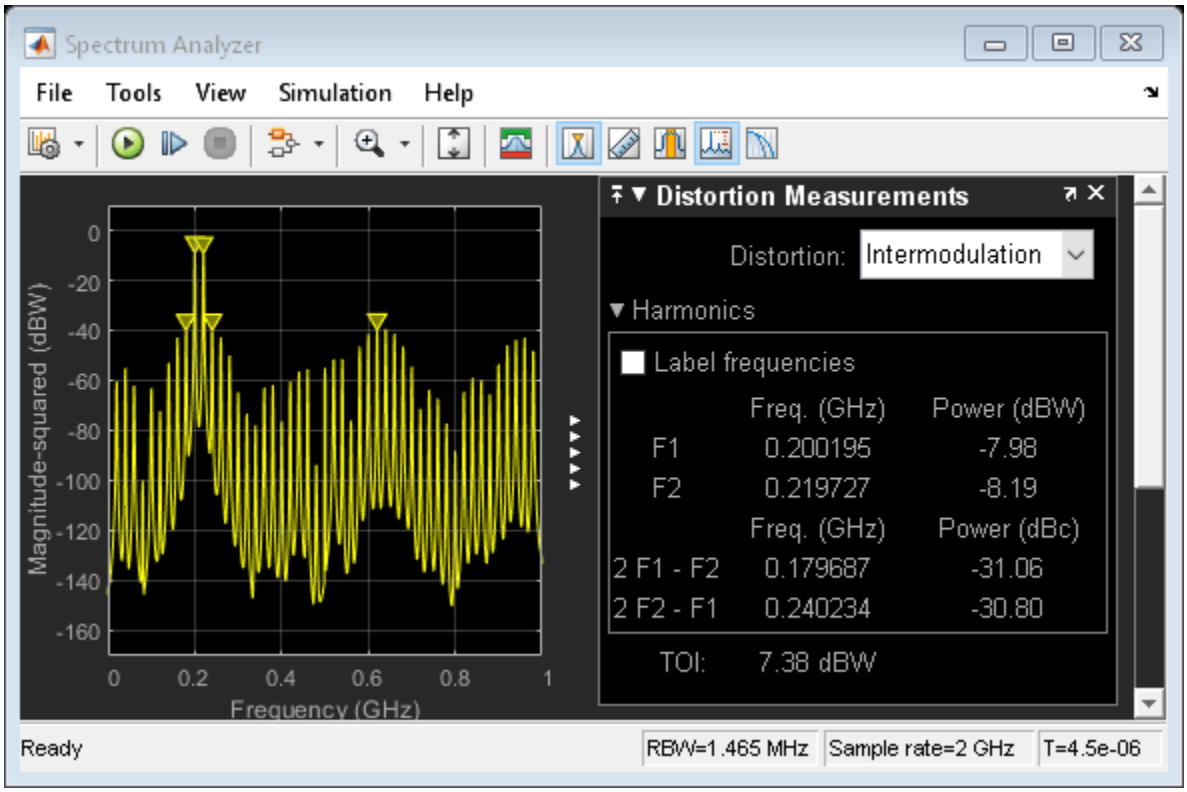
The spectrum shows the noise floor increasing as an effect of quantization.

```
sim(model)
```



Multiply the two tone test signal by a factor of 1.2. The increased amplitude saturates each ADC, producing a clipped waveform and a dirty spectrum.

```
set_param([model '/Two Tone Sine Wave'],'Amplitude','.5*1.2');
sim(model)
```



ENOB, SFDR, and Other Single Tone Measurements

ADCs are often characterized by their Effective Number of Bits (ENoB), Spurrious-Free Dynamic Range (SFDR), and other similar measurements.

These quantities are derived from a single tone test. To change the ADC's input from the Two Tone Sine Wave source to the Single Tone Sine Wave source and back, double click on the Input Switch. This test uses a single sine wave with a frequency of `200` MHz.

```
set_param([model, '/Input Switch'], 'sw', '0');
```

The ADC AC Measurement block from the Mixed-Signal Blockset™ measures conversion delay, SINAD (the ratio of signal to noise and distortion), SFDR, SNR (Signal to Noise Ratio), ENOB and the ADC's output noise floor.

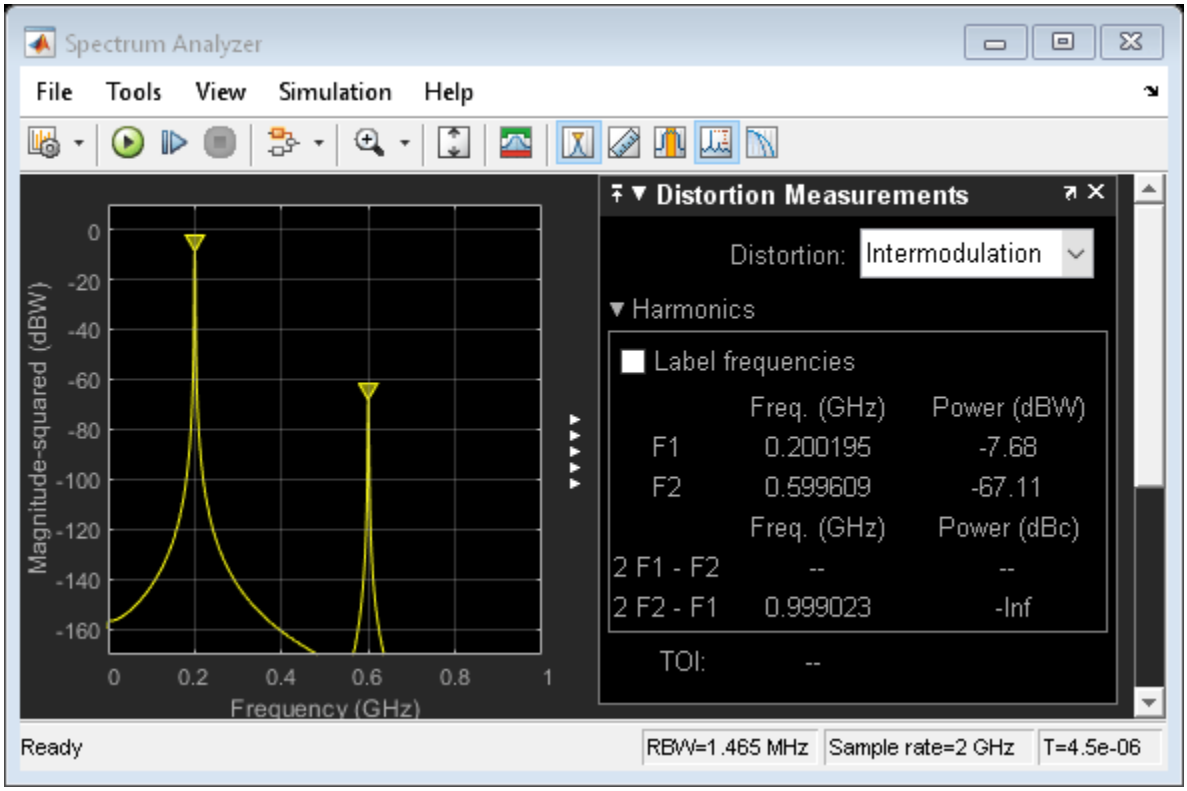
```
set_param([model, '/ADC AC Measurement'], 'Commented', 'off');
```

This block requires a rising edge on its start and ready ports for every conversion that the ADC makes. In this model, these are provided by a 4 GHz pulse generator. To use the ADC AC Measurement block in this model, uncomment the block by right clicking on it and selecting "Uncomment" from the menu. The expected ENOB from a dynamic range of 2 and a least significant bit value (quantization interval) of `2^-8` is 9 bits.

```
sim(model);
disp(interleaved_adc_output)
```

```
           SNR: 44.7989
          SFDR: 58.3746
         SINAD: 45.2291
```

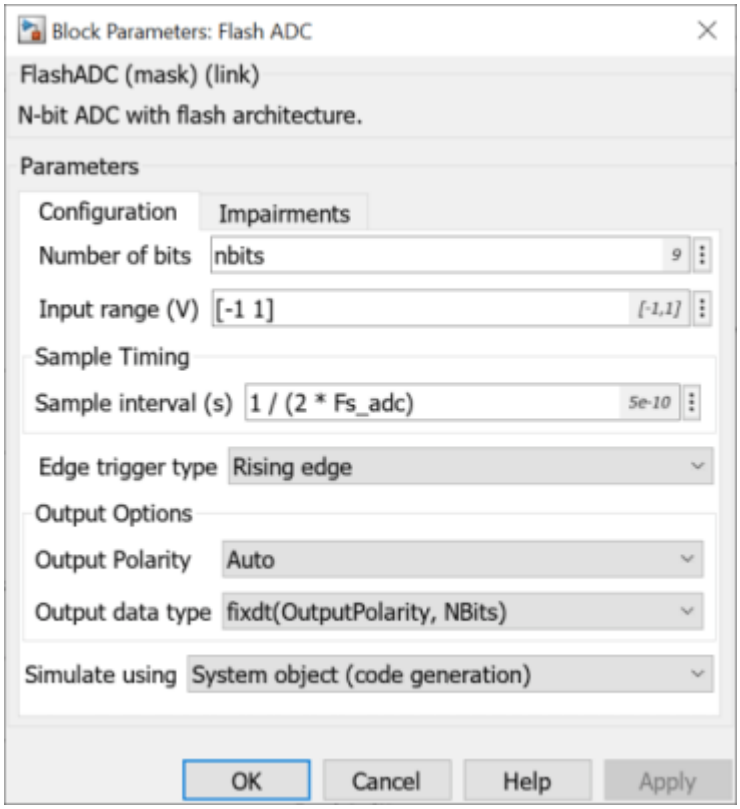
```
      ENOB: 7.2208
NoiseFloor: -50.3029
  MaxDelay: 0
 MeanDelay: 0
  MinDelay: 0
```



Substitution of ADC Block with System object™

Open the mask dialog of both Flash ADCs in the model 'interelaved_flash_adc.slx' and set **Simulate using** to `System object (code generation)`.

```
bdclose(model);
model = 'interleaved_flash_adc';
load_system(model);
```

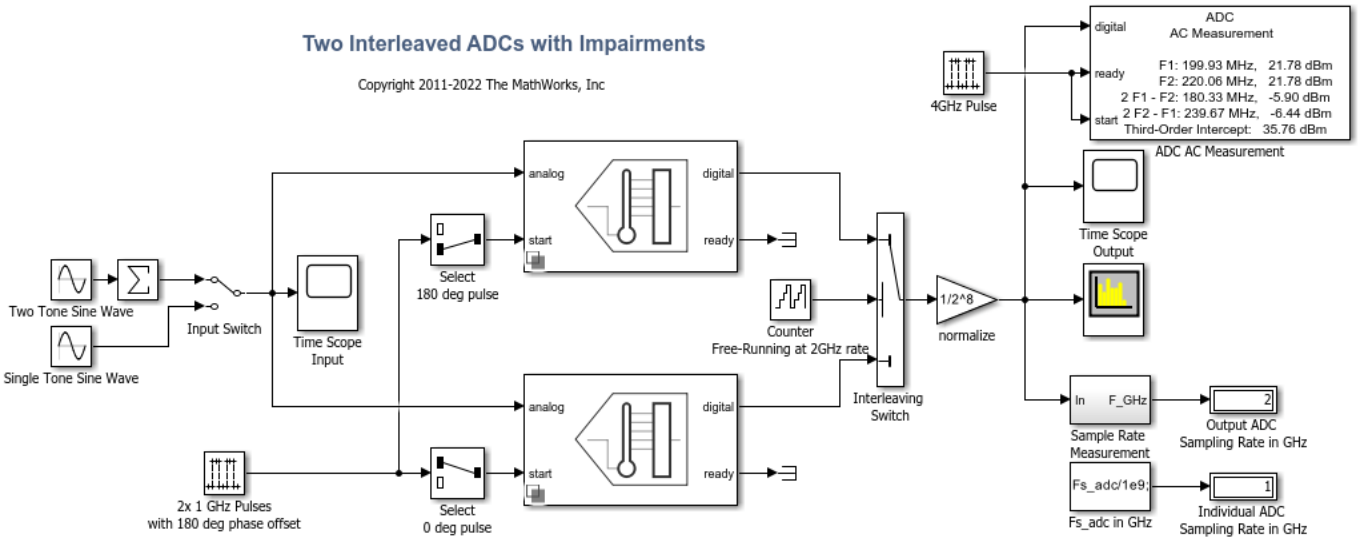


The system-object Flash ADC model is almost identical to the Simulink block Flash ADC model. But the system object model does not support aperture jitter impairment or internal conversion start clock. The system object model is also limited to fixed-step sample times. Generally, the system object model will require more compile time but run faster than the Simulink block model, making it useful for long or complex simulations. To configure the fixed-step sample rates of the ADCs, set their **Sample interval (s)** to `1 / (2 * Fs_adc)`. This is the rate at which the ADC model checks whether or not its inputs have changed. Due to rising edge detection on the conversion start signal, this rate should be at least twice the ADC's sample frequency. Setting `Fs_adc` to `1e9` and `nbits` to `9` in the base MATLAB workspace gives this model the same performance as the previous model.

The conversion start signal, specified in the 2x 1 GHz Pulses with 180 deg phase offset block, has the same **Sample time** of `1 / (2 * Fs_adc)`. This is the Nyquist rate for that signal. With this **Sample time**, set the **Period** to 2 samples, and the **Pulse width** to 1 sample. Set **Phase delay** to `[0 1]` samples to output a vector containing two antiphase conversion start signals. Use the two selector blocks to route the delayed clock signal to the top ADC and the un-delayed clock signal to the the Interleaving Switch's initial output comes from its bottom input, input 0, and the second output comes from the top input, input 1.

Set up the ADC AC Measurement block's parameters according to the parameters of the ADC and of the stimulus signal. Set the **Resolution bandwidth (Hz)** to `2,000,000` because the default value `20,000,000` is not enough resolution to differentiate the fundamentals.

```
sim(model);
open_system(model);
```



See Also

ADC AC Measurement

More About

- “Analyzing Simple ADC with Impairments” on page 2-27

Design and Evaluate Successive Approximation ADC Using Stateflow

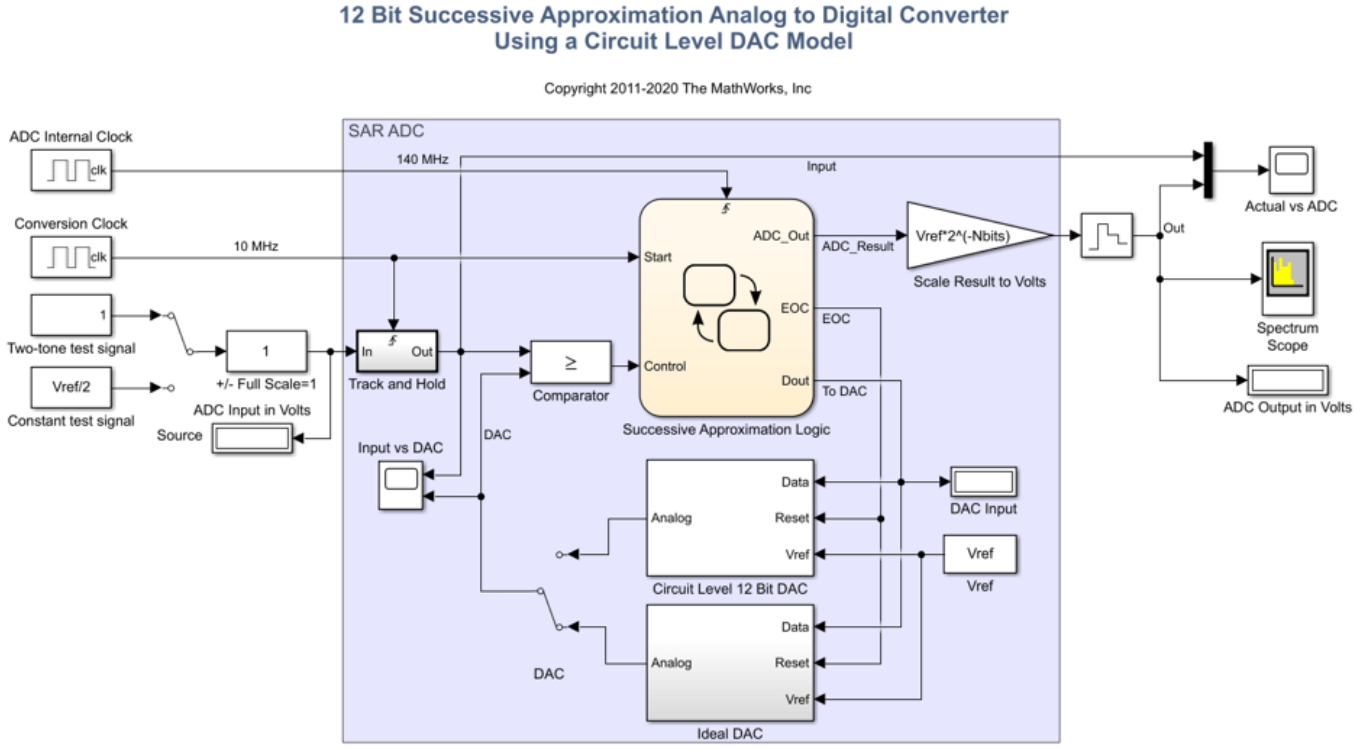
This example shows a 12 bit Successive Approximation Register (SAR) ADC with a circuit-level DAC model.

Successive Approximation ADCs typically have 12 to 16 bit resolution, and their sampling rates range from 10 kSamples/sec to 10 MSamples/sec. They tend to cost less and draw less power than subranging ADCs.

Model

Open the system `MSADCSuccessiveApproximation`.

```
model = 'MSADCSuccessiveApproximation';
open_system(model)
```



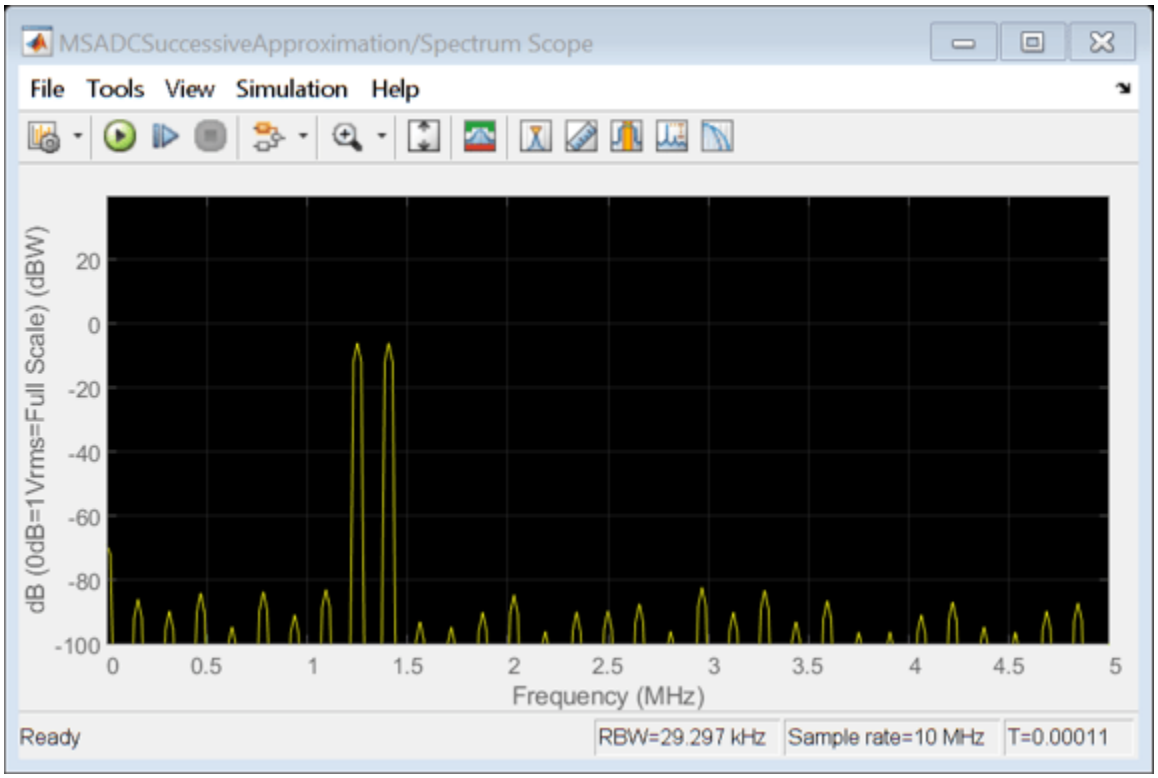
Set the switches to their default positions, selecting the two-tone source and the ideal DAC model.

```
set_param([model '/Source'],'sw','1');
set_param([model '/DAC'],'sw','0');
```

The top-level model consists ofa testbench and the device under test. The testbench includes the test signal generators and the time domain scopes and spectrum analyzer for measurement purposes. The device under test, highlighted in blue in the model, contains a Track and Hold:, a Comparator, control logic, and charge scaled DAC.

The test signal is either a two-tone sine wave or a constant DC level input. This test signal is sampled and held at the ADC's output word rate of 10 MHz. The output of the sampler serves as one input to a comparator. The second comparator input is the DAC output which is an incrementally stepped reference level. If the output of the sampler is greater than or equal to the DAC output, then the comparator outputs a logical 1. When this happens, the corresponding bit of the output is set to logical 1. Otherwise, the comparator outputs a logical 0 which does not increment the ADC output word. This single comparator is only place in the successive approximation converter where analog is converted to digital.

```
sim(model);
```



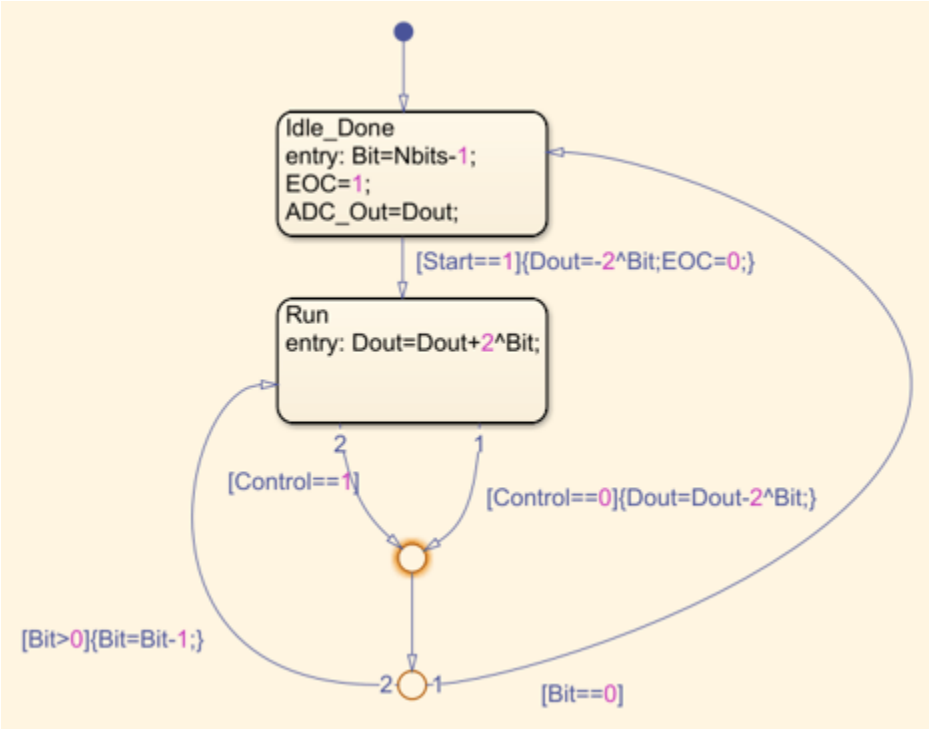
Define the number of bits (*NBits*) and ADC conversion rate (*Fs*) in the MATLAB® workspace. The ADC operating clock rate is determined from *Nbits* and *Fs* .

```
Nbits = 12;
Fs = 1e7;
ADC_clock = Fs*(Nbits+2);
```

Successive Approximation Control Logic

This model uses Stateflow to model the successive approximation control logic. The state-machine serves as a sequencer that starts by outputting a count corresponding to midscale which in this case is 0 volts. The state-machine then performs a binary search of one bit position at a time to find the count corresponding to the closest approximation to the sampled input signal within 12 bits of resolution.

```
open_system([model '/Successive Approximation Logic'],'force')
```



On a particular bit, if the comparator outputs a 1, then that bit is set. Otherwise that bit position is cleared. Because there are 12 bits, it takes 12 clock cycles at the bit rate clock to complete the conversion for a given input sample.

In this model, the bit rate clock denoted by block labeled ADC Internal Clock runs at `140` MHz. This clock is 14 times faster than the sample rate clock denoted by the block labeled Conversion Clock in the upper left corner of the model. After the control logic sequences from bit 11 down to bit 0 the end-of-conversion (EOC) line goes high, telling the DAC circuitry to reset.

DAC Circuit-Level Implementation

The circuit-level DAC uses a multi-stage charge-scaled array of capacitors in a split-array format. This architecture provides several advantages including reduced area or parts count, a built-in sample and hold, low power dissipation, and a relatively small range of capacitance values as would be required without a split-array.

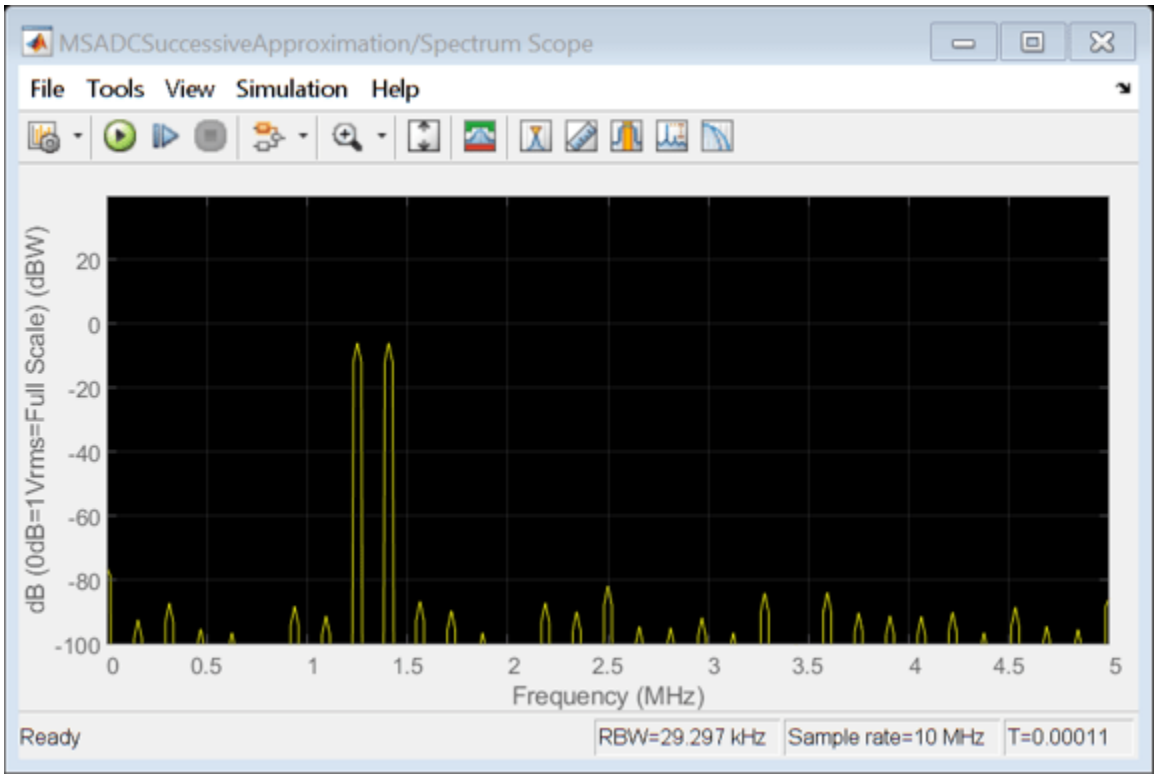
There are two versions of a digital to analog converter (DAC) in this model, one at the circuit-level and the second representing ideal DAC behavior. The ideal DAC block takes the input count and multiplies it by

$$\frac{V_{ref}}{2^{N_{bits}}} = \frac{2\sqrt{2}}{2^{12}} \left[\frac{Volts}{Count} \right]$$

to generate the output comparison voltage [1].

Set the switch to enable the circuit-level DAC model. Run the model.

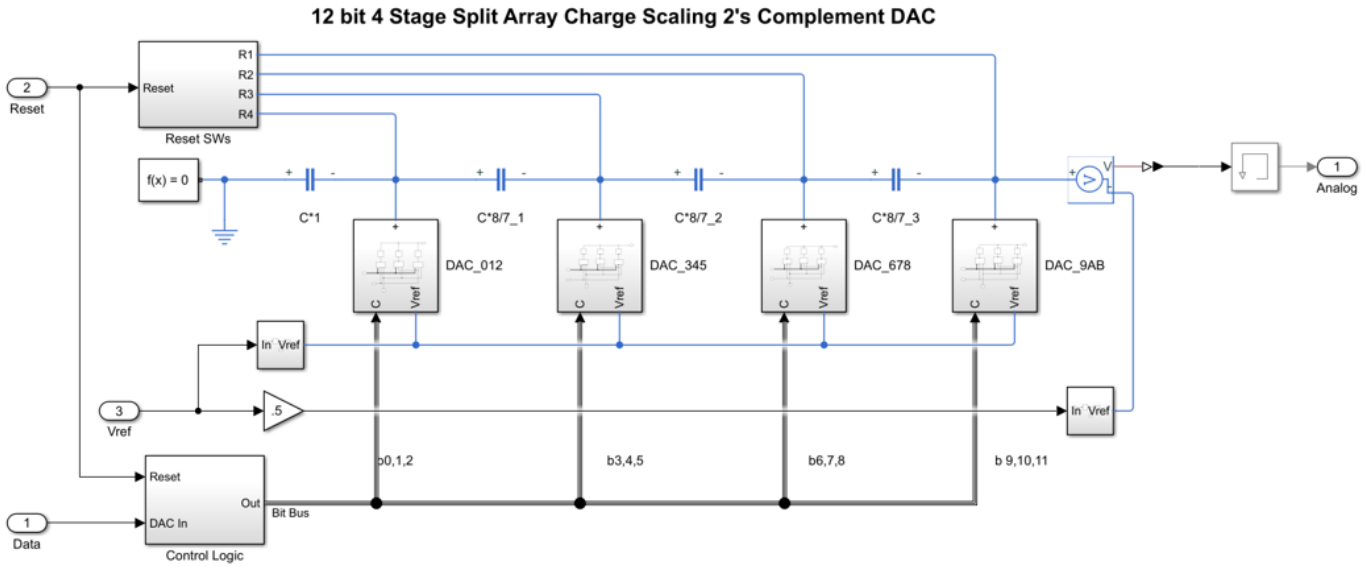
```
set_param([model '/DAC'],'sw','1');
sim(model);
```



This particular charge-scaled array uses $N_{caps} = 3$ binary-weighted capacitors per stage with a total of $K = 4$ stages providing a total of $N_{caps} * K = 12$ bits of DAC resolution. The binary-weighted capacitors per stage have a value of $1C, 2C$, and $4C$. The larger the capacitance corresponds to a higher bit position within a particular stage. For example, setting the low side of the $4C$ capacitor high has 4 times the output voltage impact relative to setting the low side of the $1C$ capacitor high.

If you change the value of the variable *Nbits*, the physical number of bits of the converter, you need to modify the circuit level implementation of the DAC. The ideal DAC implementation and the control logic are parametrized with respect to the number of bits.

```
open_system([model '/Circuit Level 12 Bit DAC'],'force')
```



Each stage is separated by a scaling capacitor with value $\frac{8}{7}C$. The scaling capacitor serves the purpose of attenuating the output voltage of each stage's output voltage. The further the stage is from the DAC output node, the more it is attenuated. The attenuation is 8x per scaling capacitor which corresponds to $2^{N_{caps}}$.

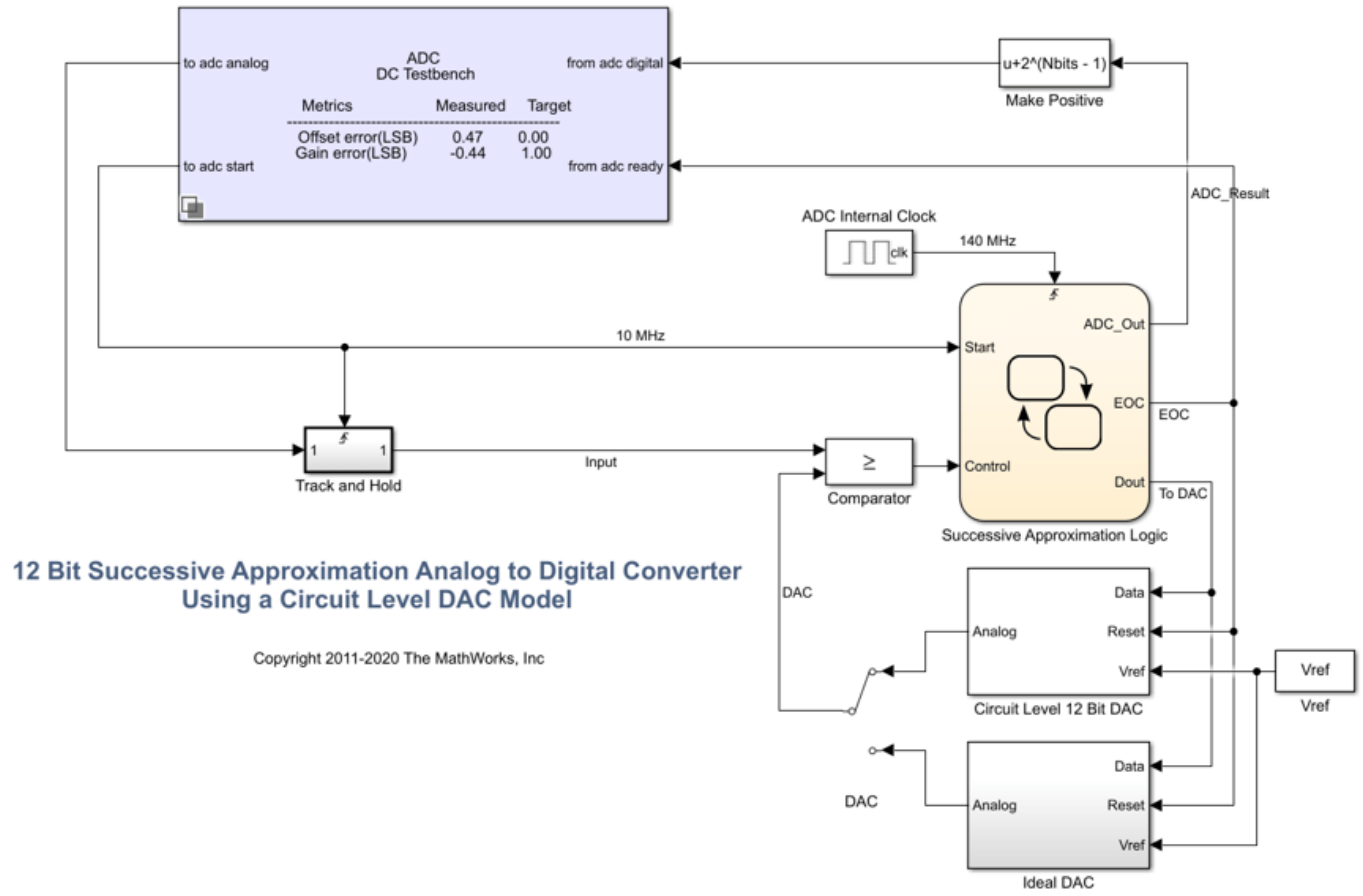
The three MSBs are closest to the output, bits 0, 1, and 2 while there LSBs, bits 10, 11, and 12 are furthest away. At any given time, the DAC is in one of two modes. It is either generating an output voltage based on a particular input count or it is being reset when the EOC line goes high. When EOC goes high, the low side of each capacitor in the DAC is switched to ground rather than data, thus draining the capacitors of charge in preparation for the next approximation. This effectively drains the capacitive network of charge preparing it for the next input sample.

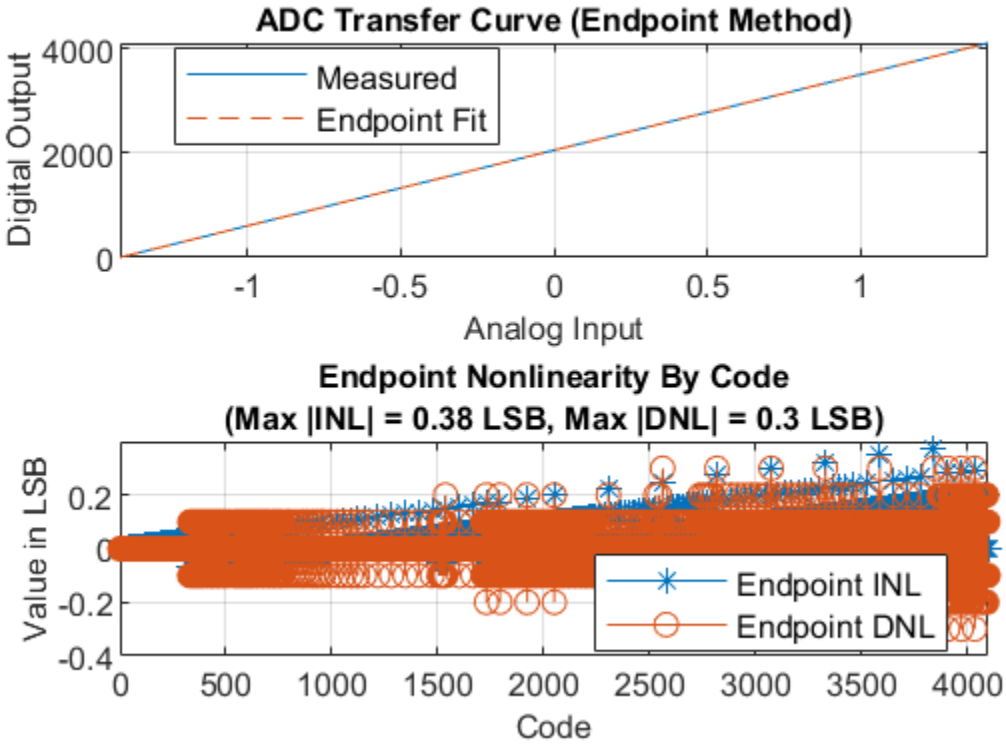
Measurement Testbench

The ADC Testbench block from the Mixed-Signal Blockset™ can provide a performance analysis of the ADC.

In DC mode, the ADC Testbench tests the linearity of the ADC. The test result is used to generate offset and gain error measurements which are displayed on the block mask. The full test results are available for export or visualization via the buttons on the ADC Testbench block mask.

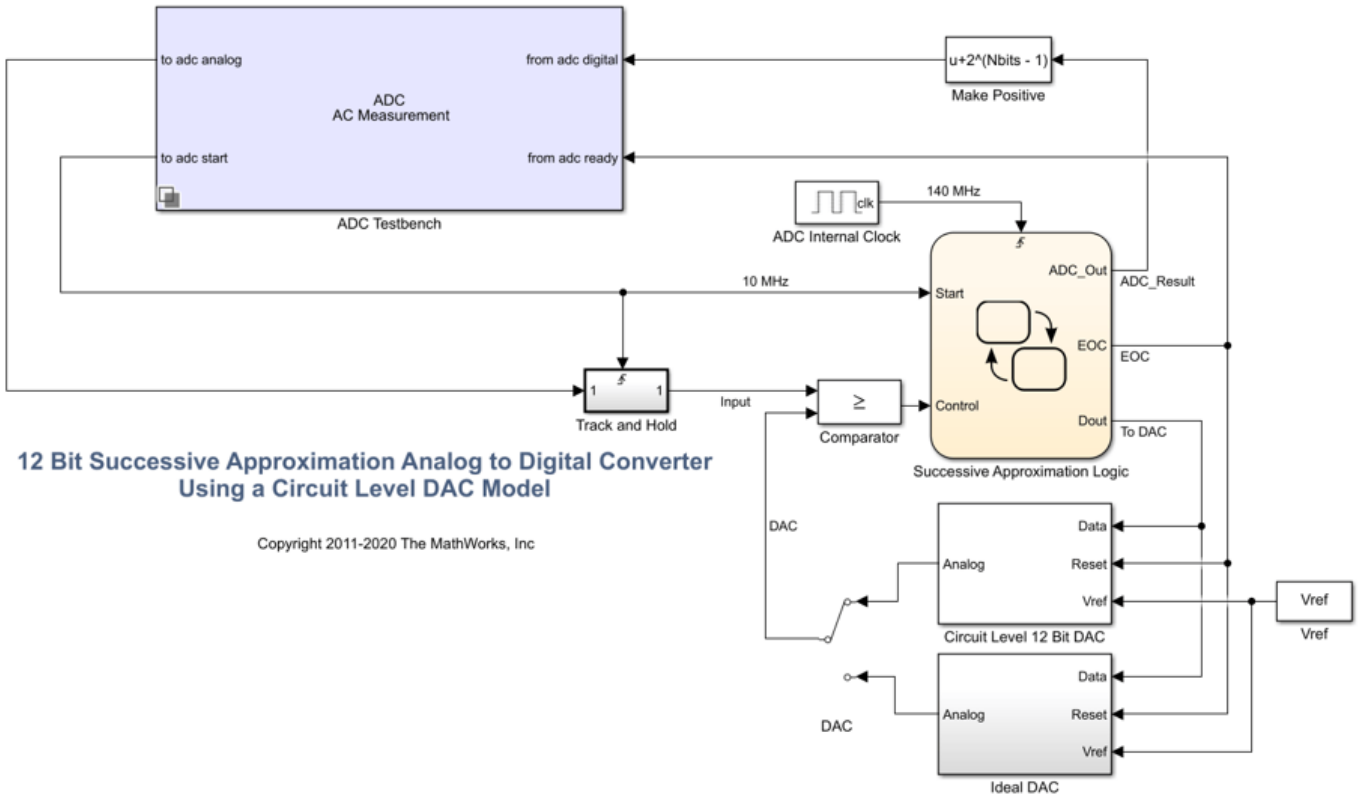
```
bdclose(model);
model = 'MSADCSAR_DC';
open_system(model);
sim(model);
```





The AC mode of the ADC Testbench provides insight into the frequency performance of the ADC, including measurements like the ENoB (Effective Number of Bits), the maximum measured conversion delay, and the noise floor of the converter. These measurements are displayed on the block icon after simulation and are available for export via a button on the block mask.

```
model = 'MSADCSAR_AC';
open_system(model);
set_param([model '/DAC'],'sw','1');
```



References

1 Haideh Khorramabadi UC Berkeley, Department of Electrical Engineering and Computer Sciences, Lecture 15, page 38 <http://inst.eecs.berkeley.edu/~ee247/fa06/lectures/L15_f06.pdf>

See Also

Sampling Clock Source | ADC DC Measurement | ADC AC Measurement

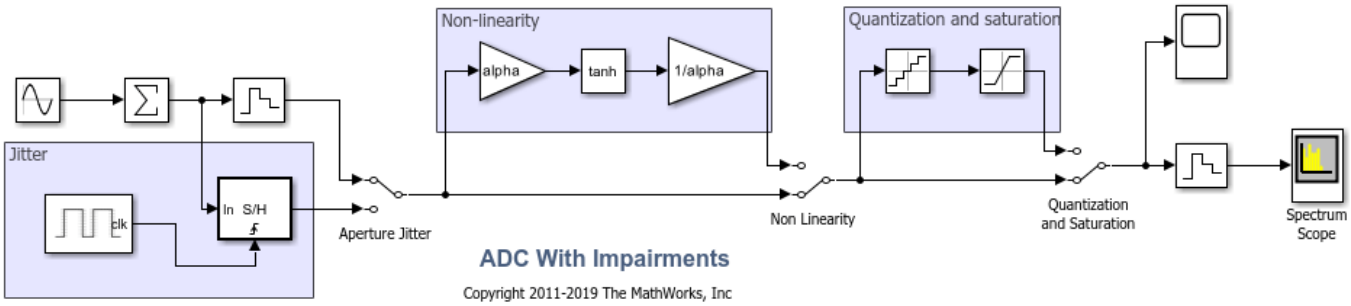
More About

- “Compare SAR ADC to Ideal ADC” on page 2-48

Analyzing Simple ADC with Impairments

This example shows how to implement a basic ADC using a Zero-Order Hold block as a sampler. This simple ADC highlights some of the typical impairments introduced in an analog-to-digital converters such as aperture jitter, nonlinearity, quantization, and saturation. This example shows how to measure the effects of such impairments using a Spectrum Analyzer block and the ADC AC Measurement block from the Mixed-Signal Blockset™. To better approximate real-world performance, you can individually enable the impairments in the model.

```
model = 'MSADCImpairments';
open_system(model)
```

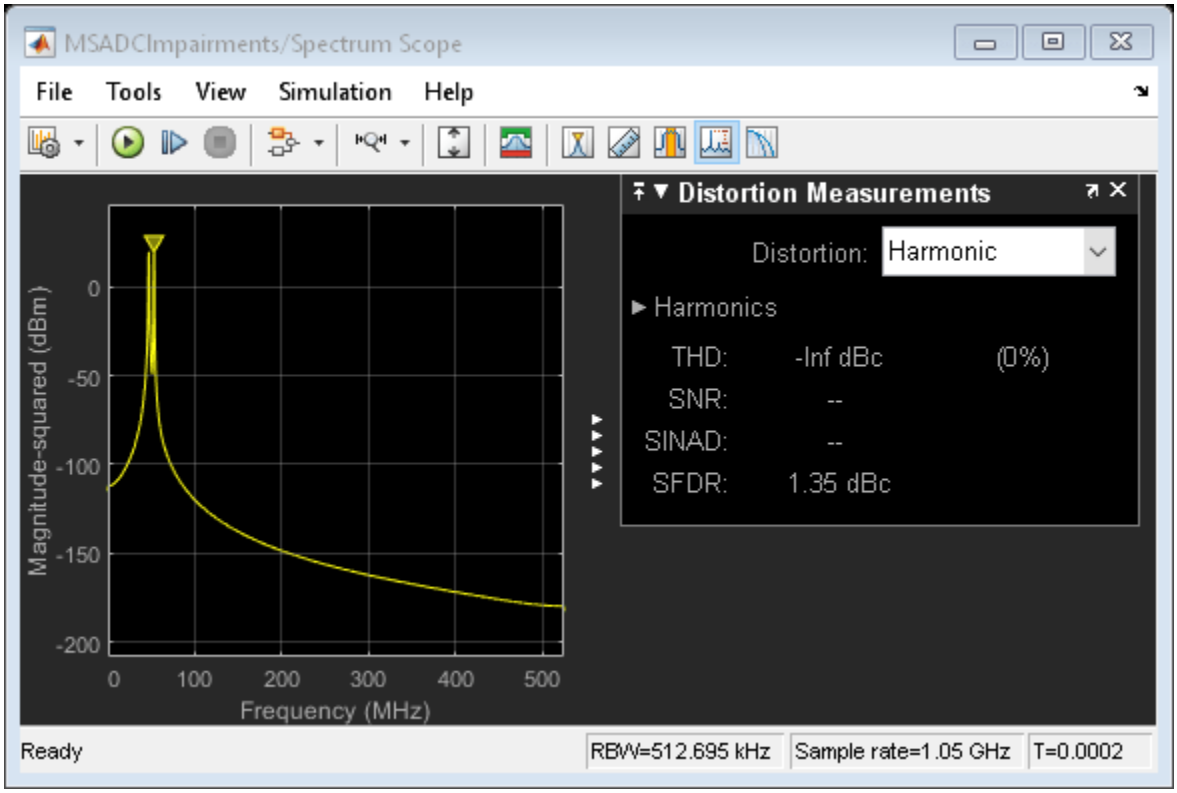


To observe the behavior of an ideal ADC, bypass the impairments using the switches. Set the Sine Wave source to generate two tones as an input signal.

```
set_param([model '/Aperture Jitter'],'sw','1');
set_param([model '/Non Linearity'],'sw','0');
set_param([model '/Quantization and Saturation'],'sw','0');
set_param([model '/Sine Wave'],'Frequency', '2*pi*[47 53]*1e6');
```

Simulate the model and observe the expected clean output spectrum of the ADC.

```
sim(model);
```



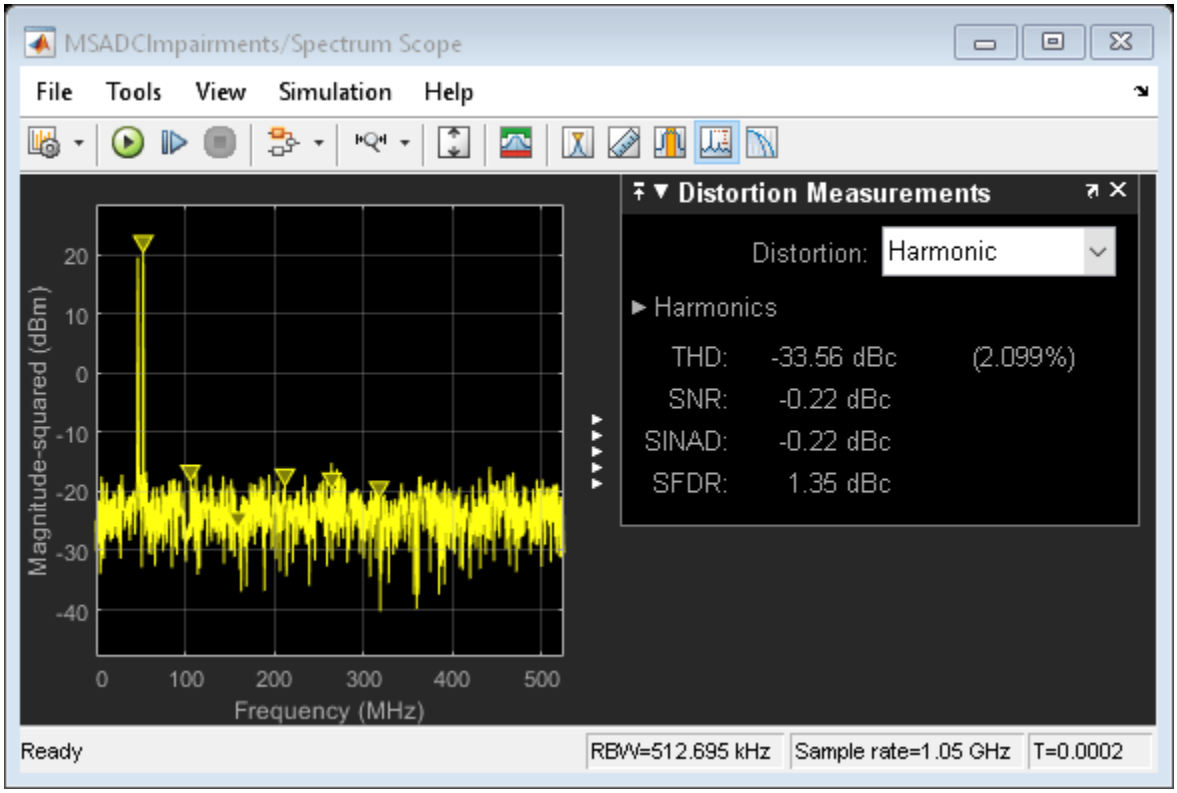
Effect of Aperture Jitter

Set the first switch to the down position. The Variable Delay block delays the signal sample-by-sample by the amount on its *td* input. The Noise Source block generates a uniform random variable, which is low-pass filtered by the Shape the jitter noise spectrum block before it arrives at the *td* input to the Variable Delay. Use a shaped uniform noise distribution to represent the jitter. Notice that in this model, the clock of the ADC is specified in the ideal zero-order hold block, and it is equal to 1/Fs, where Fs is a MATLAB® variable defined in the model initialization callback and equal to `1.024` GHz.

```
set_param([model '/Aperture Jitter'],'sw','0');
```

As expected, the spectrum degrades because of the presence of the jitter.

```
sim(model);
```



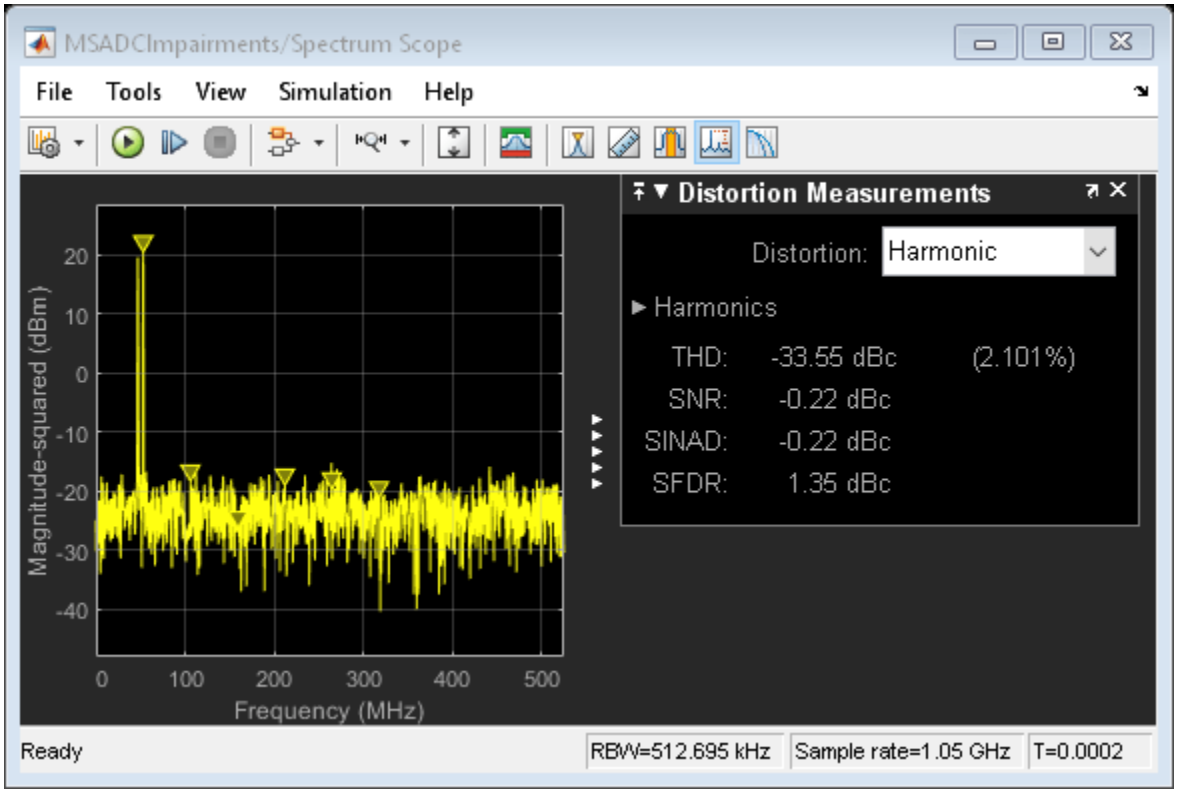
Effect of Nonlinearity

Set the second switch to the up position. This enables the ADC nonlinearity. A scaled hyperbolic tangent function provides nonlinearity. Its scale factor, *alpha*, determines the amount of nonlinearity the tanh applies to the signal. By default, *alpha* is `0.01`.

```
set_param([model '/Non Linearity'],'sw','1');
```

The spectrum degrades because of the nonlinearity as higher order harmonics get generated.

```
sim(model);
```



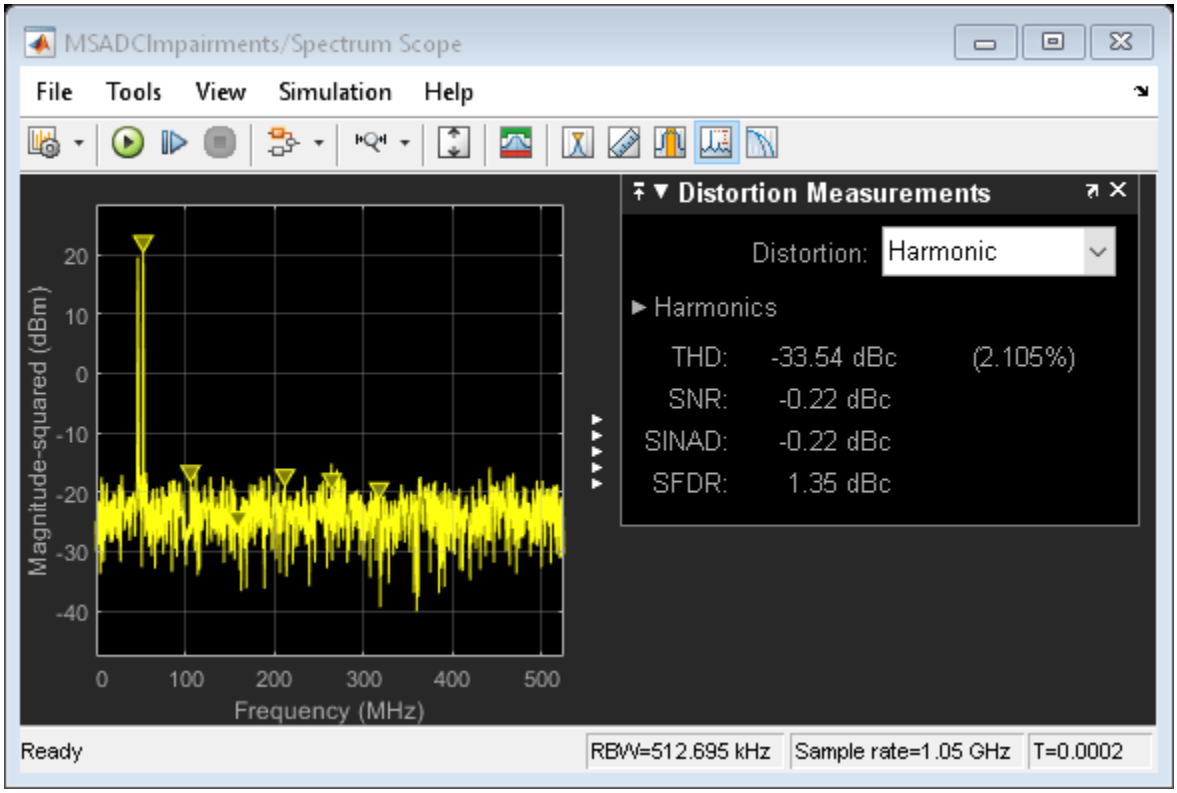
Effect of Quantization and Saturation

Set the third switch to the up position enabling the ADC quantization and hard saturation.

```
set_param([model '/Quantization and Saturation'],'sw','1');
```

The spectrum degrades because of the quantization effects. The noise floor raises as seen in the spectrum.

```
sim(model);
```

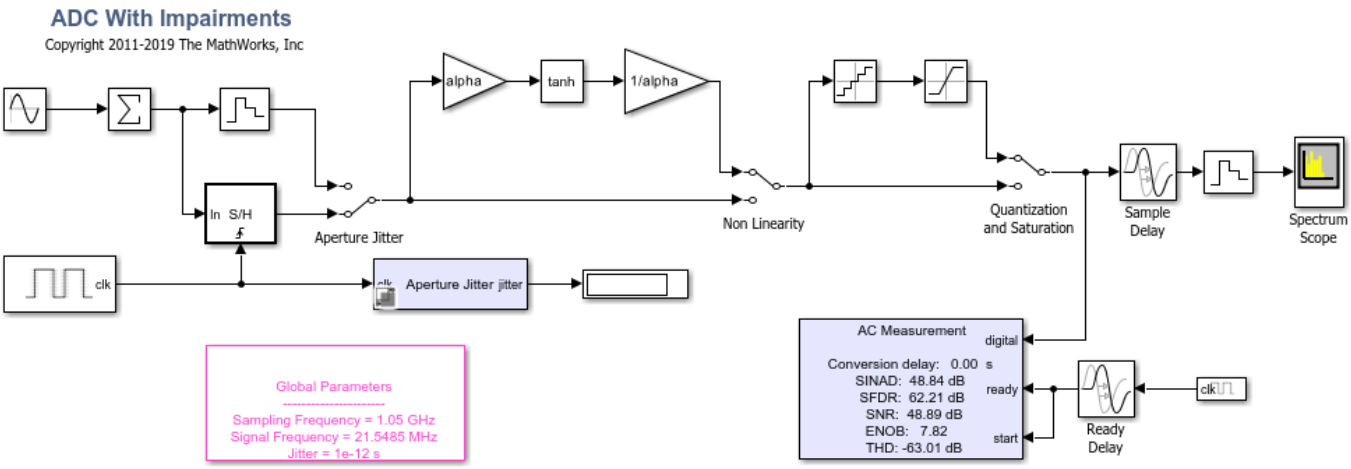


ADC AC Measurements

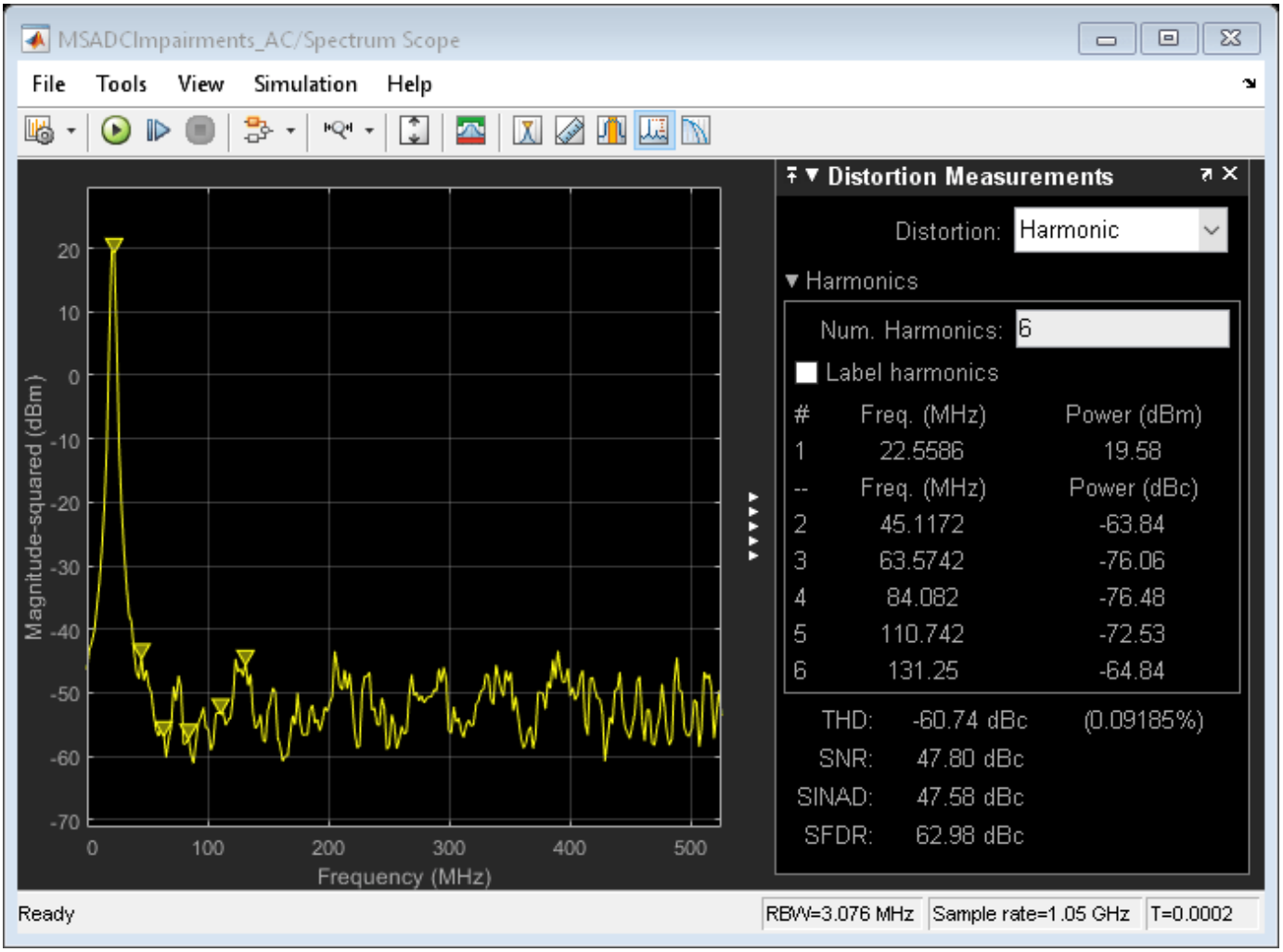
Use the ADC AC Measurement block in the Mixed-Signal Blockset™ to measure the noise performance of the ADC and compute the effective number of bits (ENOB).

Use single sinusoidal tone as input to the ADC to measure other metrics.

```
bdclose(model);
model = 'MSADCImpairments_AC';
open_system(model);
```



```
Ftest = 33/round(2*pi*2^8)*Fs;
set_param([model '/Sine Wave'],'Frequency', '2*pi*Ftest');

scopecfg = get_param([model '/Spectrum Scope'], 'ScopeConfiguration');
scopecfg.DistortionMeasurements.Algorithm = 'Harmonic';
scopecfg.FFTLength = '512';
scopecfg.WindowLength = '512';

sim(model);
```



The Aperture Jitter Measurement block from Mixed-Signal Blockset™ measures the average jitter introduced on the signal to be approximately equal to 1 ps. Delay is added

Additionally, use the spectrum analyzer to measure:

- Output Third Order Intercept Point (OIP3)
- Signal to Noise Ratio (SNR)
- Total Harmonic Distortion (THD)

Increase the factor *alpha* to increase the nonlinearity of the ADC and make the effects of nonlinearity more evident on top of the noise floor. This is just for demonstration purposes.

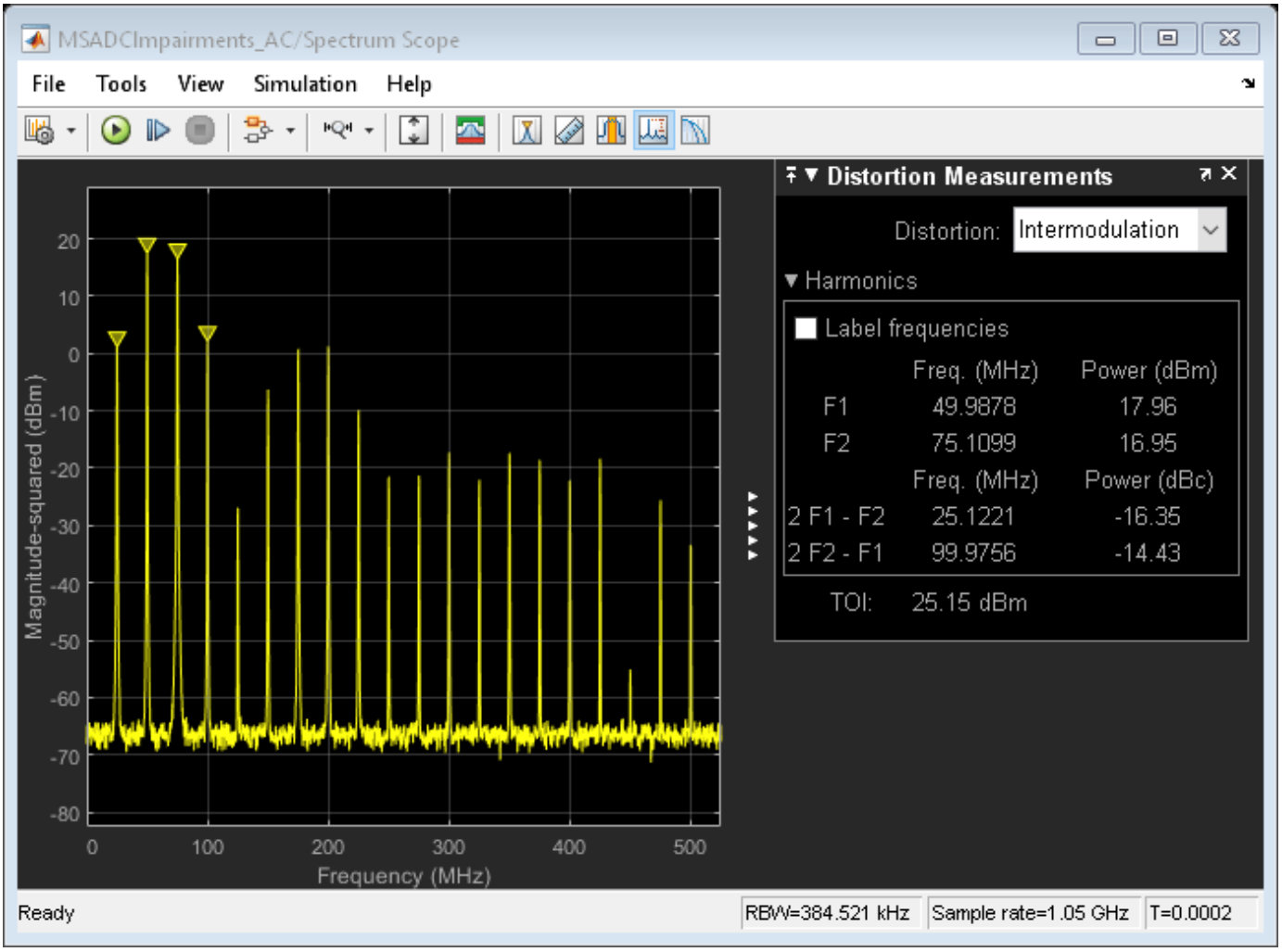
```
alpha = 0.8;
```

Use a two tone test signal as input to the ADC for the intermodulation measurements.

```
set_param([model '/Sine Wave'],'Frequency', '2*pi*[50e6 75e6]');
```

To enable distortion measurements in the spectrum analyzer, click on **Distortion Measurement** as in the figure below and select `Intermodulation` as **Distortion** type.

```
scopecfg.DistortionMeasurements.Algorithm = 'Intermodulation';
scopecfg.FFTLength = '4096';
scopecfg.WindowLength = '4096';

sim(model);
```



The scope allows for the measurement of the third order products adjacent to the input signals, and determines the output referred third order intercept point.

See Also

Sampling Clock Source | Aperture Jitter Measurement | ADC AC Measurement

More About

- “Compare SAR ADC to Ideal ADC” on page 2-48
- “Design and Evaluate Interleaved ADC” on page 2-8
- “Subranging ADC” on page 2-35

Subranging ADC

This example shows how to model a 6-bit Subranging ADC with pipelining and an error correcting second stage.

Subranging ADCs are typically faster than sigma delta and successive approximation ADCs but provide less resolution. Typical sample rates are in the 10M sample/sec range, and typical resolution is 8 to 16 bits. Additional stages and/or bits per stage yield higher resolutions.

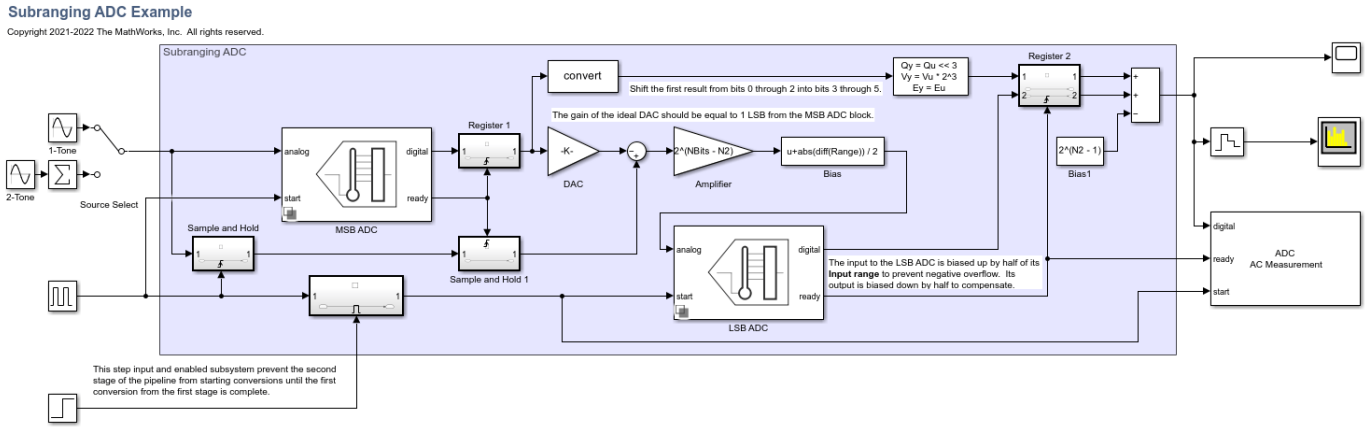
Model

The subranging ADC is a two stage data converter. The first stage converter is a 3-bit Flash ADC. It drives a 3-bit ideal DAC, which in turn drives the second stage. The second stage converter is 4-bit flash ADC. The extra bit corrects the errors in the first ADC and improves conversion accuracy. The sampling rate of the ADC is `100` MHz, and is defined in the model initialization callback by MATLAB® variable Fs. Additional MATLAB® variables N1, N2, and NBits set the number of bits for the first stage, second stage and overall converter respectively.

The model is based on the following Analog Devices tutorial: ADC Architectures V: Pipelined Subranging ADCs.

The first ADC serves as a coarse 3 bit converter. The quantization error due to the first ADC is itself quantized by the second ADC. To generate this error signal, a 3-bit DAC converts the coarse ADC output to an analog signal which is subtracted against the original analog input. The difference is the residue signal. The residue signal is amplified and converted back to a digital signal by the second ADC. The 3 bit ADC output (MSBs) and the 4-bit ADC output (LSBs) are concatenated to form an overall 6 bit unsigned ADC output.

```
model = 'subranging_adc_ac';
load_system(model);
sc = get_param([model '/Spectrum Analyzer'], 'ScopeConfiguration');
sc.DistortionMeasurements.Algorithm = 'Harmonic';
open_system(model);
```

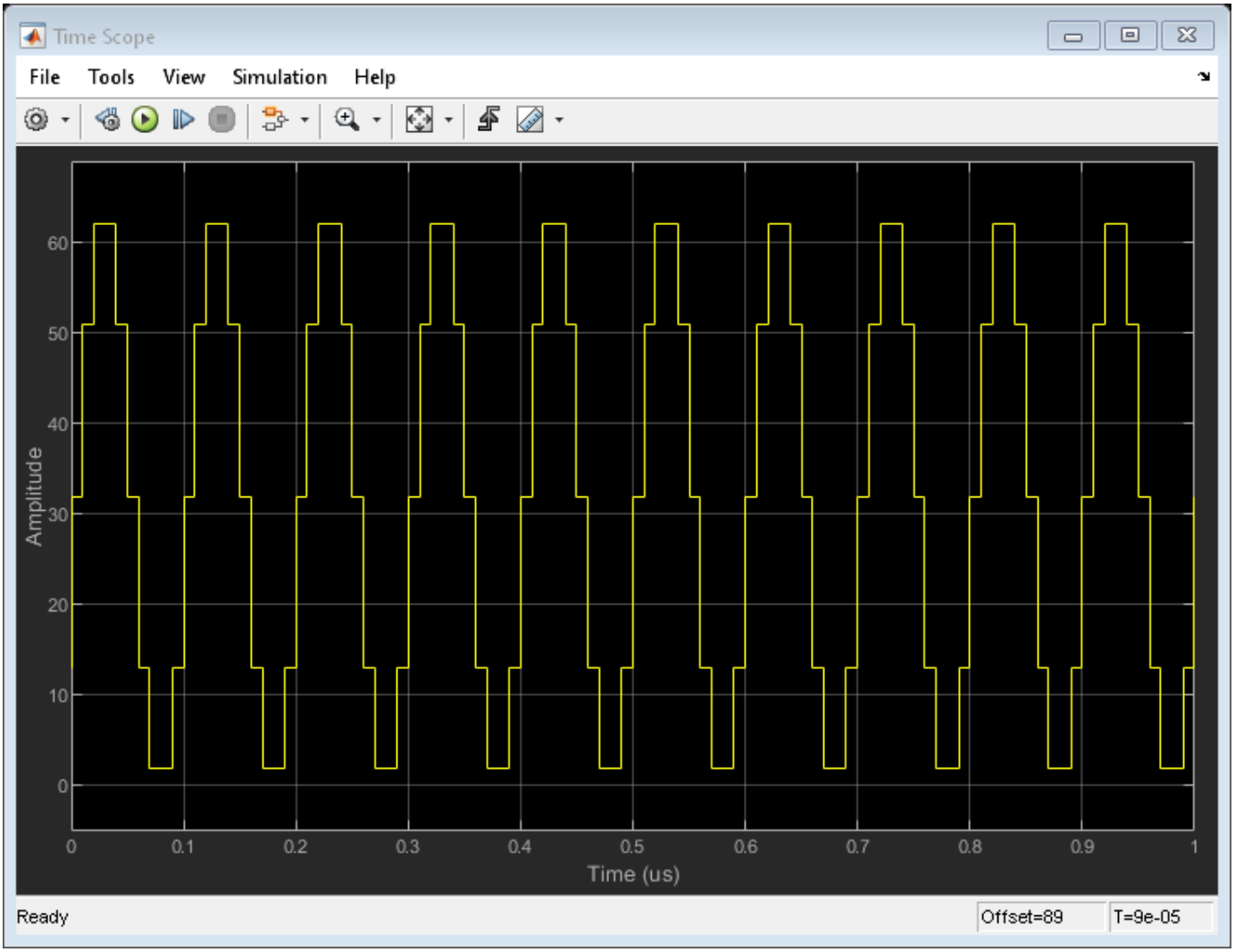


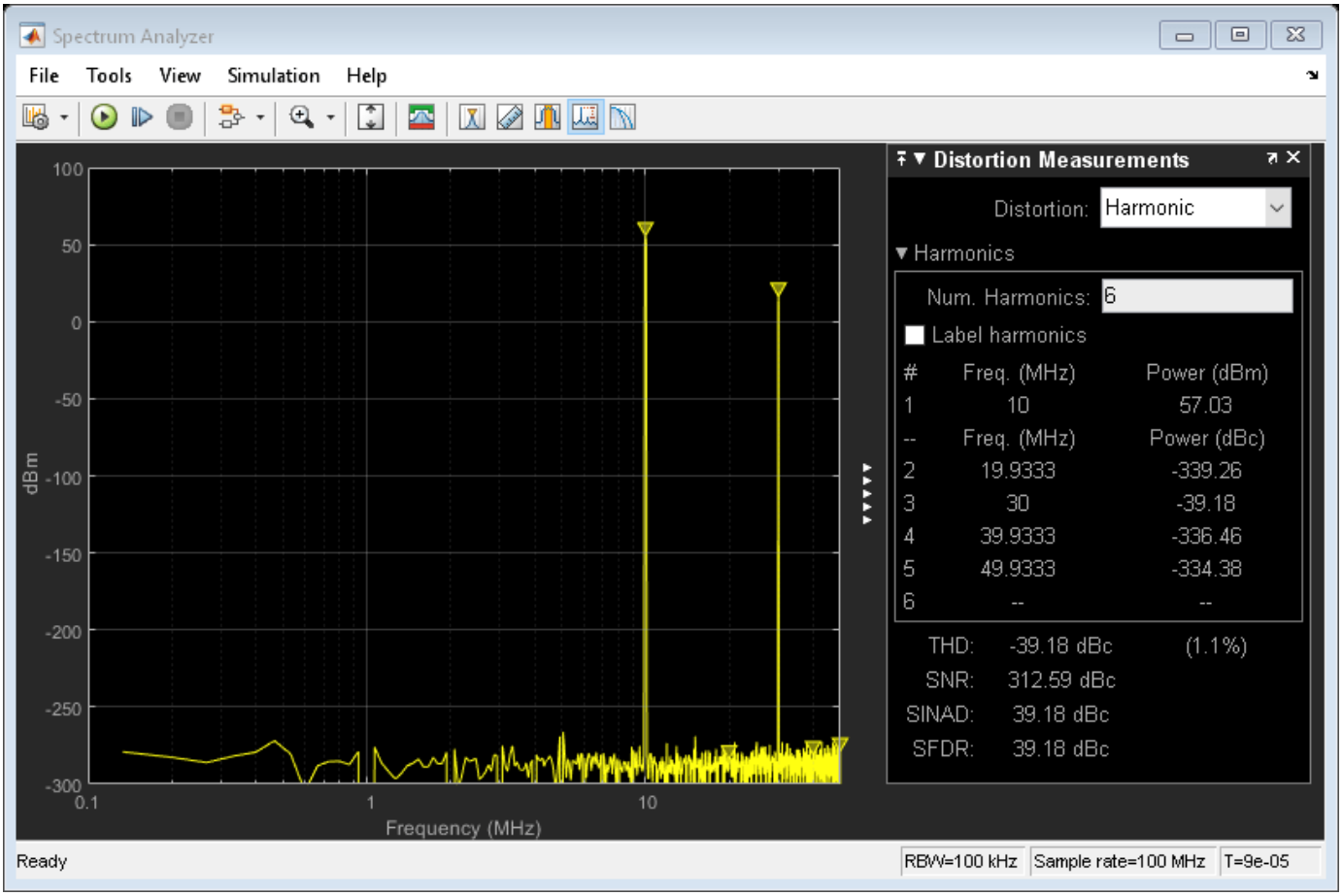
Dynamic Testing

This model uses one of two test sources. Use a sine wave for dynamic testing, e.g. ENoB, SNR, and SFDR. Use a ramp signal for static testing, e.g. missing codes and nonlinearity quantifiers.

To determine SNR, ENOB and other dynamic characteristics of the subranging ADC, use the ADC AC Measurement block from the Mixed-Signal Blockset™. Select the correct switch position to use the Sine Wave as input source. Set the simulation stop time using the button on the mask of the ADC AC Measurement block and set its **Distortion measurement type** is set to `Harmonic`.

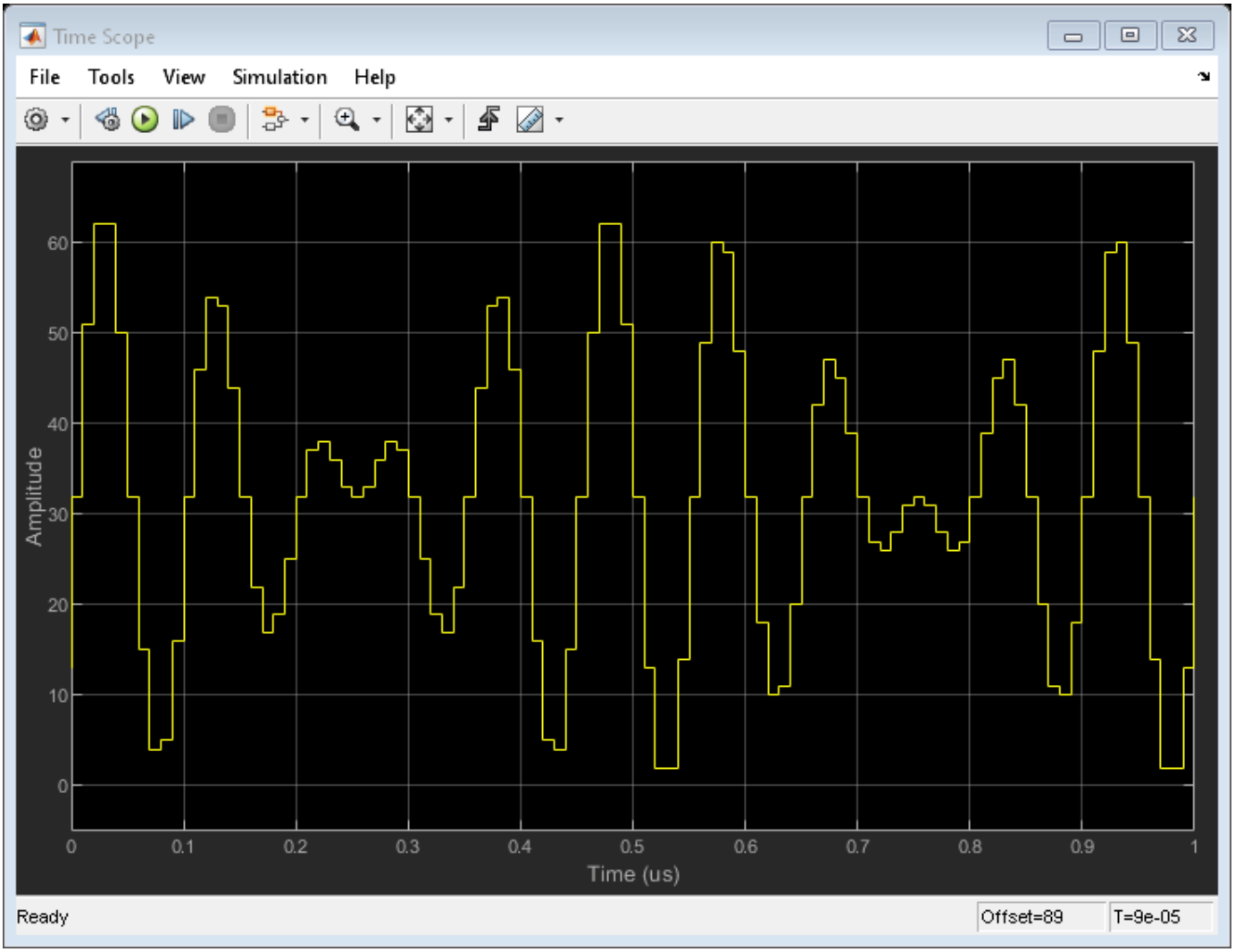
```
set_param([model '/Source Select'],'sw','1');
set_param([model '/ADC AC Measurement'], 'InputFrequency', '10e6');
open_system([model '/Time Scope']);
open_system([model '/Spectrum Analyzer']);
sim(model);
```

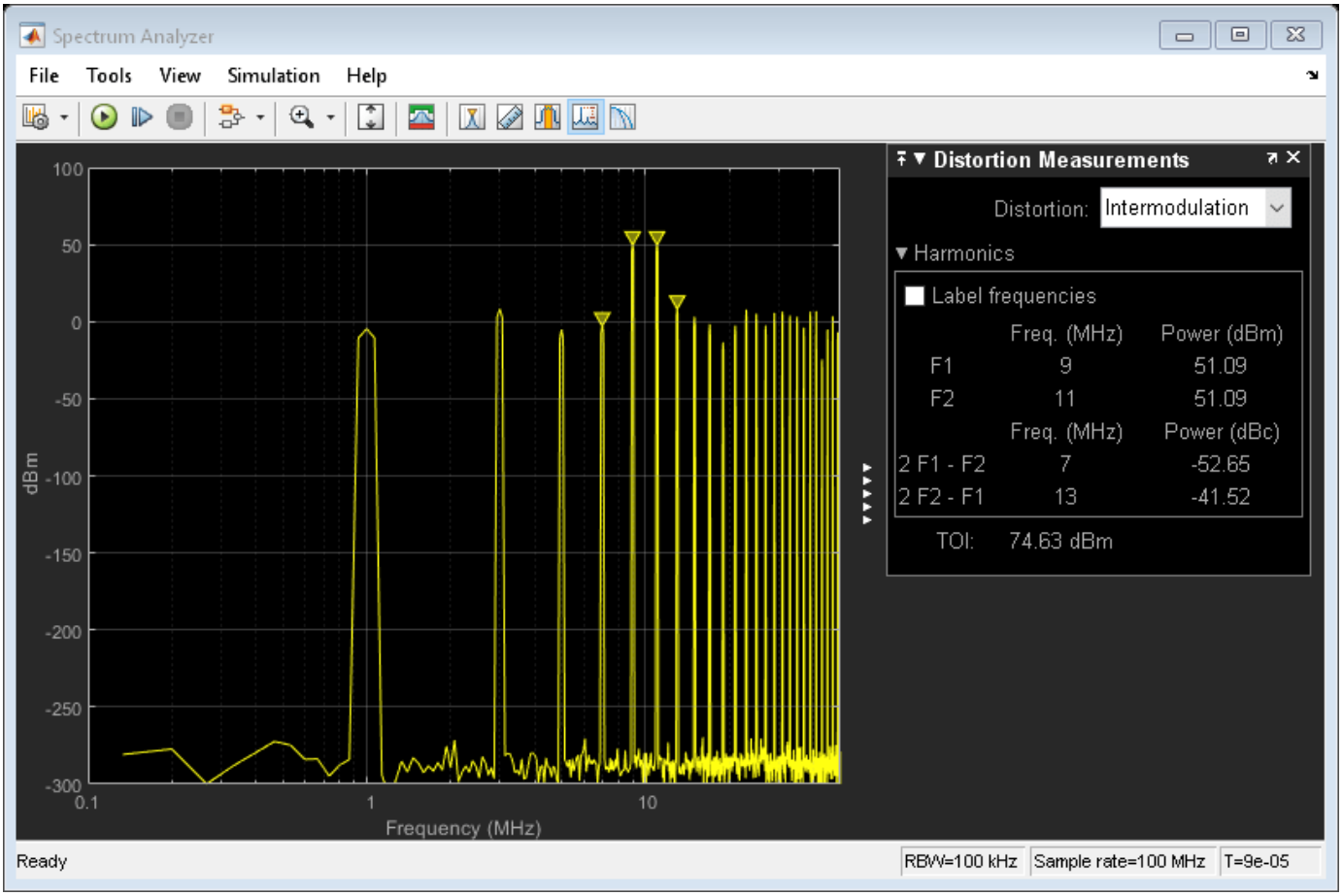




Add another sine wave to the input to test intermodulation distortion. Observe and measure the results using the ADC AC Measurement block with its **Distortion measurement type** set to `Intermodulation`.

```
set_param([model '/Source Select'],'sw','0');
set_param([model '/ADC AC Measurement'], 'DistortionMeasurement', 'Intermodulation');
sc.DistortionMeasurements.Algorithm = 'Intermodulation';
set_param([model '/ADC AC Measurement'], 'InputFrequency', '[9, 11] * 1e6');
sim(model);
```

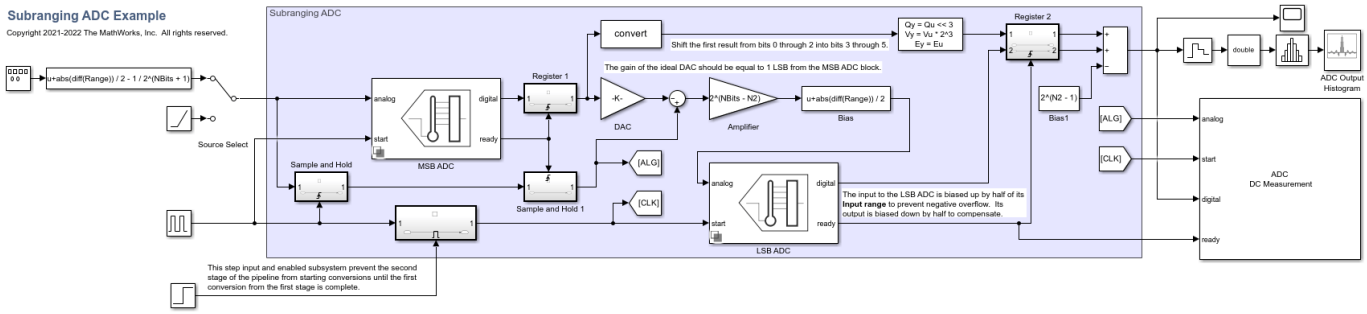

Time Scope
File Tools View Simulation Help
Amplitude
60
50
40
30
20
10
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Time (us)
Ready
Offset=89
T=9e-05



Missing Code Analysis

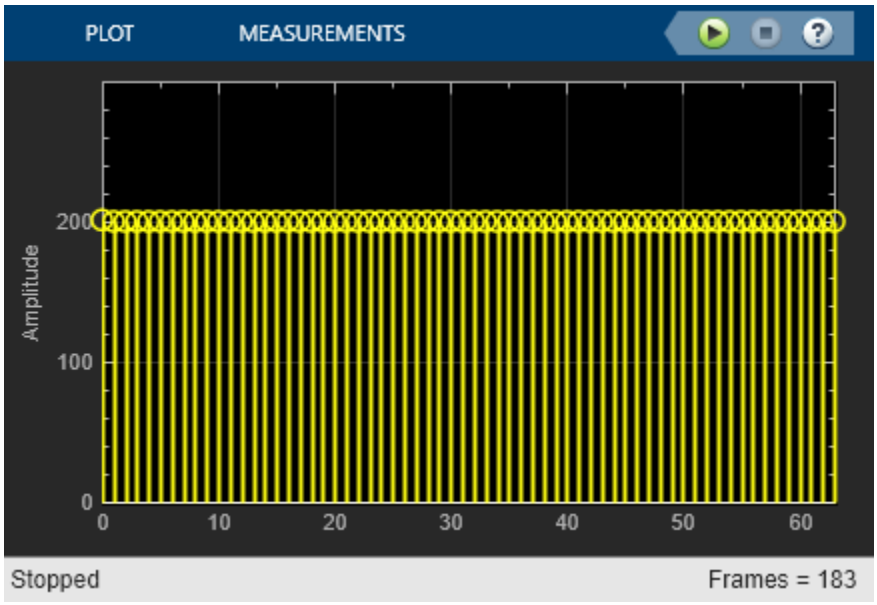
This example uses a histogram block to plot the frequency of occurrence of each ADC output code. Use the Sawtooth Generator block as the input source by setting the input switch to the proper position.

```
close_system(model, 0);
model = 'subranging_adc_dc';
load_system(model);
set_param([model '/MSB ADC'],'OffsetError','0');
set_param([model '/MSB ADC'],'GainError','0');
set_param([model '/Source Select'],'sw','1');
open_system(model);
```



Set the simulation time to `1.28e-4` s, which is 10x longer than the recomended minimum simulation stop time from the ADC DC Measurement block. Given our sawtooth stimulus, this will cause our entire transfer curve to be swept 10 times, increasing the accuracy of the histogram by reducing the influence of random impairments such as aperture jitter.

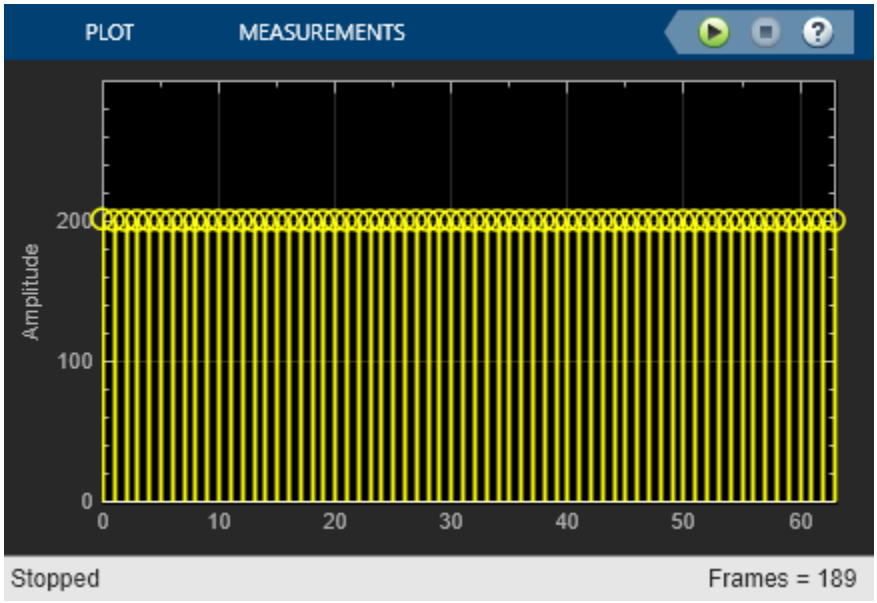
```
set_param(model, 'StopTime', '1.28e-4');
open_system([model '/ADC Output Histogram']);
sim(model);
```



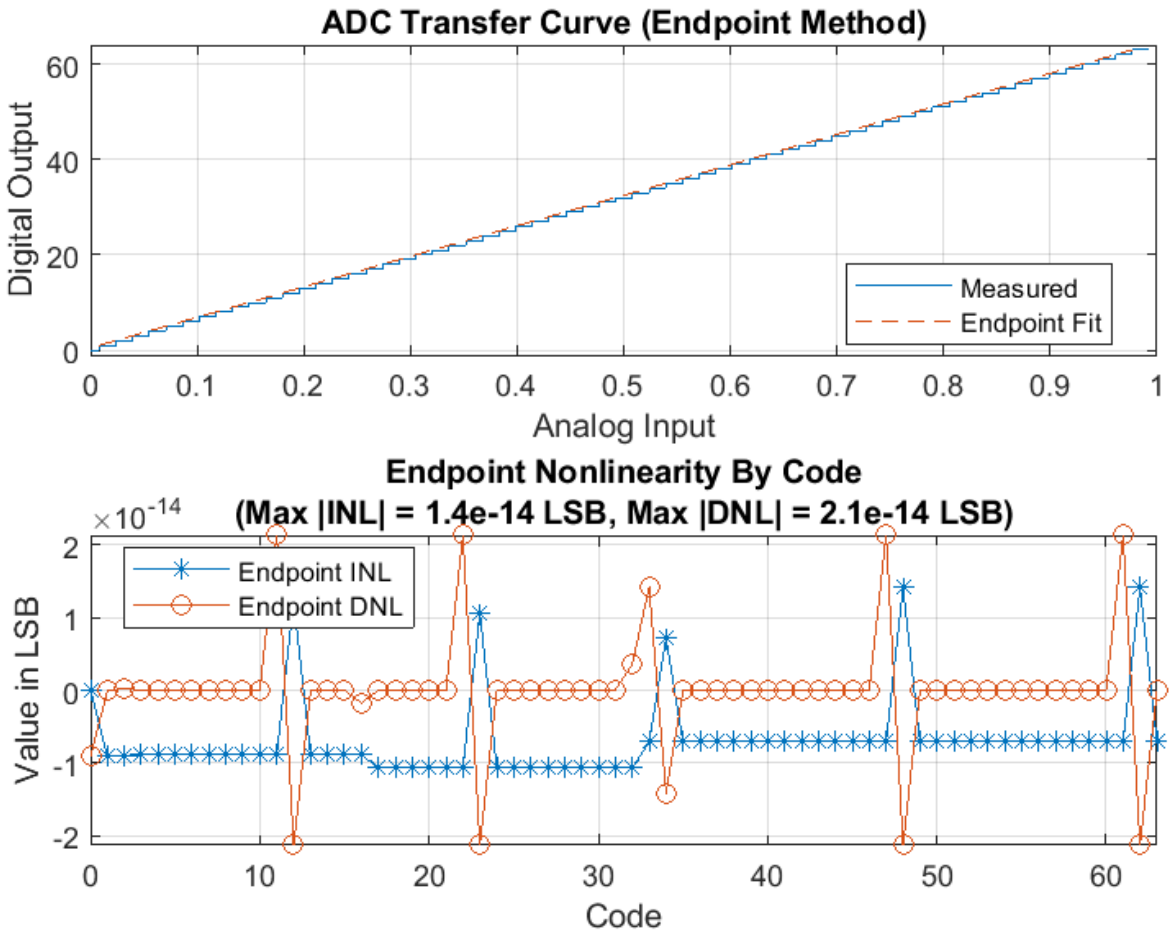
You can observe how the second stage corrects errors from the first by adding **Offset error** and **Gain error** in the impairments tab of the MSB ADC. Any fraction of an LSB introduced as a form of gain or offset error results in an imbalance or missing code in the ADC output histogram. Anything other than a flat histogram with a sawtooth input signifies some amount of non-ideal ADC behavior in the form of integral non-linearity, differential non-linearity, or missing codes.

Total errors `max(abs(`**`Offset error + Gain error, Offset error`**`))` of up to 0.5 LSB in the MSB ADC are corrected by the extra bit of the LSB ADC. Larger errors in the MSB ADC and any errors in the LSB ADC influence the output.

```
set_param([model '/MSB ADC'], 'OffsetError', '0.2');
set_param([model '/MSB ADC'], 'GainError', '0.2');
sim(model);
```



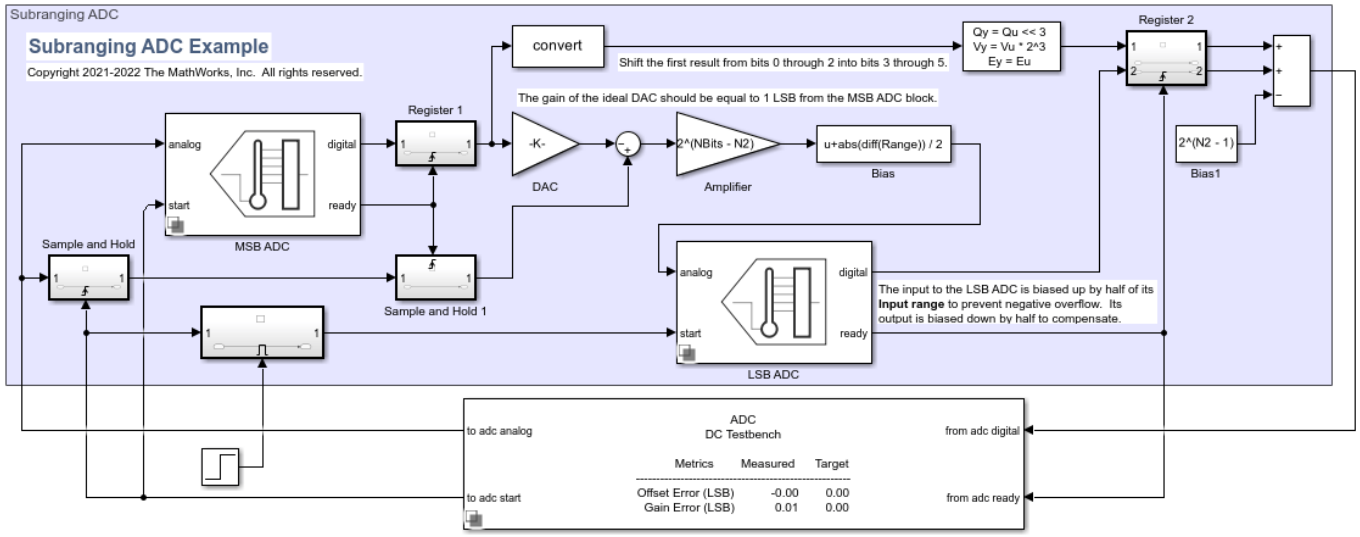
View the results of the ADC DC Measurement block by clicking the Plot measurement results button in the block mask.



ADC Testbench

Verify that the results of your tests are due only to properties of the ADC rather than of the input signals or output processing with the ADC Testbench. Connect the inputs and outputs of the converter to the inputs and outputs of the ADC Testbench and run the simulation. The results of the test will show up on the block mask once you run the simulation.

```
close_system(model, 0);
model = 'subranging_adc_testbench';
load_system(model);
sim(model);
open_system(model);
```



See Also

Flash ADC | ADC Testbench | ADC AC Measurement | ADC DC Measurement

More About

- “Analyzing Simple ADC with Impairments” on page 2-27
- “Effect of Metastability Impairment in Flash ADC” on page 2-2

Measuring Offset and Gain Errors in ADC

This example shows how offset error and gain error are calculated and how each affects the transfer curve of an ADC. Offset error and gain error characterize part of the linearity error in an ADC.

Offset error is the difference between the center of the least significant code and the center of the same code on an ideal ADC with the same number of bits. Offset error is usually reported in units of the least significant bit (LSB) of the converter. One LSB is equivalent to the converter's quantization interval.

Gain error in LSB is the difference between the center of the most significant code after offset error correction and the center of the same code on an ideal ADC with the same number of bits.

Due to the difficulty in determining the precise location of the center of a code with a non finite boundary, these values are most commonly measured with respect to the first and last threshold of the converter, respectively.

Units for Offset Error and Gain Error

The unit LSB, which is used in this example, is defined as:

$$1\,[LSB] = \frac{FSR}{2^{N_{bits}}}$$

Thus, an error in V (volts) translates into LSB as follows:

$$E\,[LSB] = 2^{N_{bits}} \frac{E\,[V]}{FSR\,[V]}$$

where FSR is the full scale range of the ADC and N_{bits} is the number of bits of the ADC.

Linearity errors are also commonly reported in normalized units with respect to the full scale range:

$$E_{Offset} = \frac{E_{Offset}\,[V]}{FSR\,[V]}$$

Another unit sometimes used for linearity errors used is percent full scale. Percent full scale is defined as normalized units multiplied by 100:

$$E_{Gain}\,[\%] = 100 \frac{E_{Gain}\,[V]}{FSR\,[V]}$$

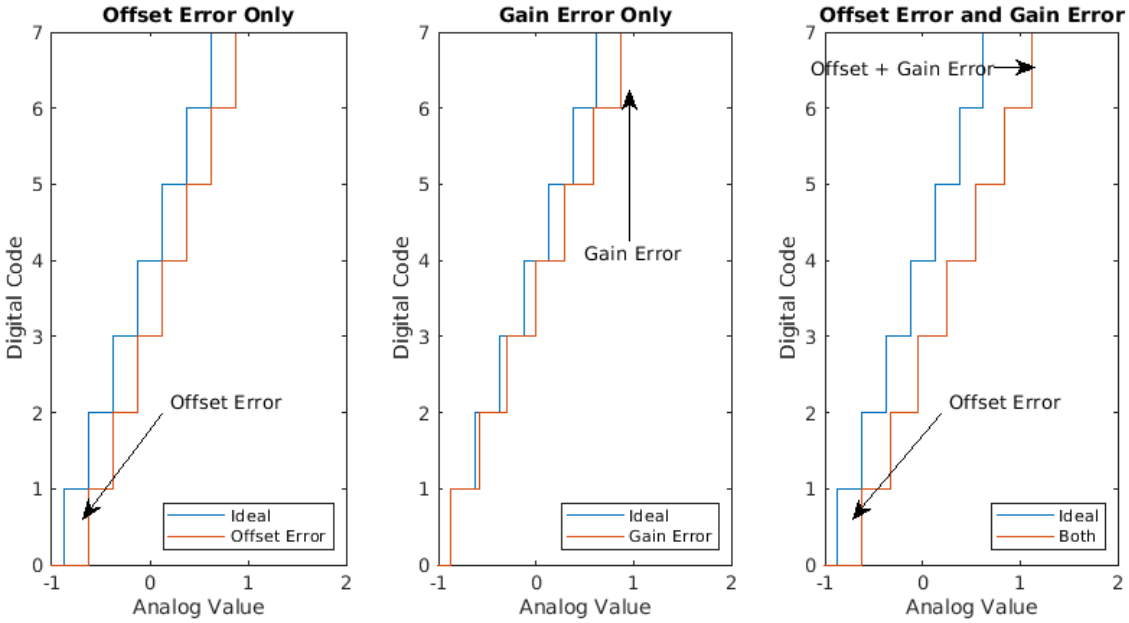
Calculate Offset Error and Gain Error

Use a 3-bit ADC with dynamic range [-1 1]. Define an offset error of 1 LSB and a gain error of 1 LSB.

```
Nbits = 3;
Range = [-1 1]; % ADC Full Scale Range
OffsetError = 1;
GainError = 1;
LSB = (max(Range) - min(Range)) / (2^Nbits - 1);
```

The digital codes from the ideal ADC are identical to those from the experimental ADCs.

```
[TC0Analog, TC0Digital] = og2tc(Nbits, Range, 0, 0); % Ideal ADC Transfer Curve
TC1Analog = og2tc(Nbits, Range, OffsetError, 0); % Offset Error Only
TC2Analog = og2tc(Nbits, Range, 0, GainError); % Gain Error Only
TC3Analog = og2tc(Nbits, Range, OffsetError, GainError); % Both Offset Error and Gain Error
plotAdcTcForExample(TC0Digital, TC0Analog, TC1Analog, TC2Analog, TC3Analog, Nbits);
```

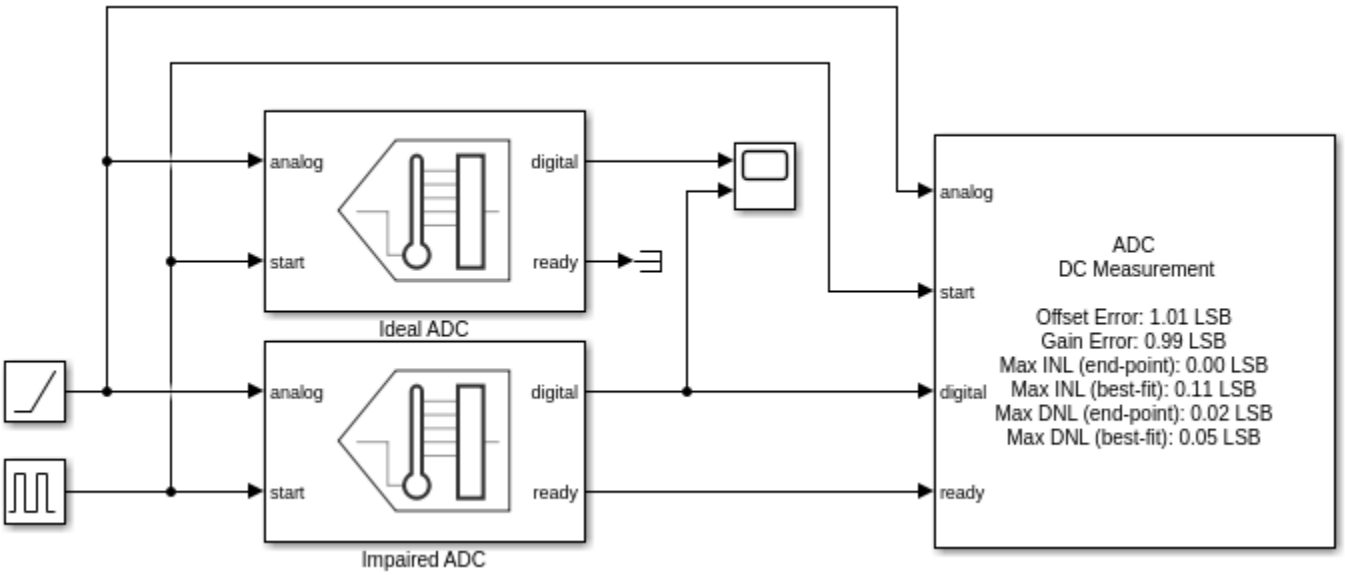


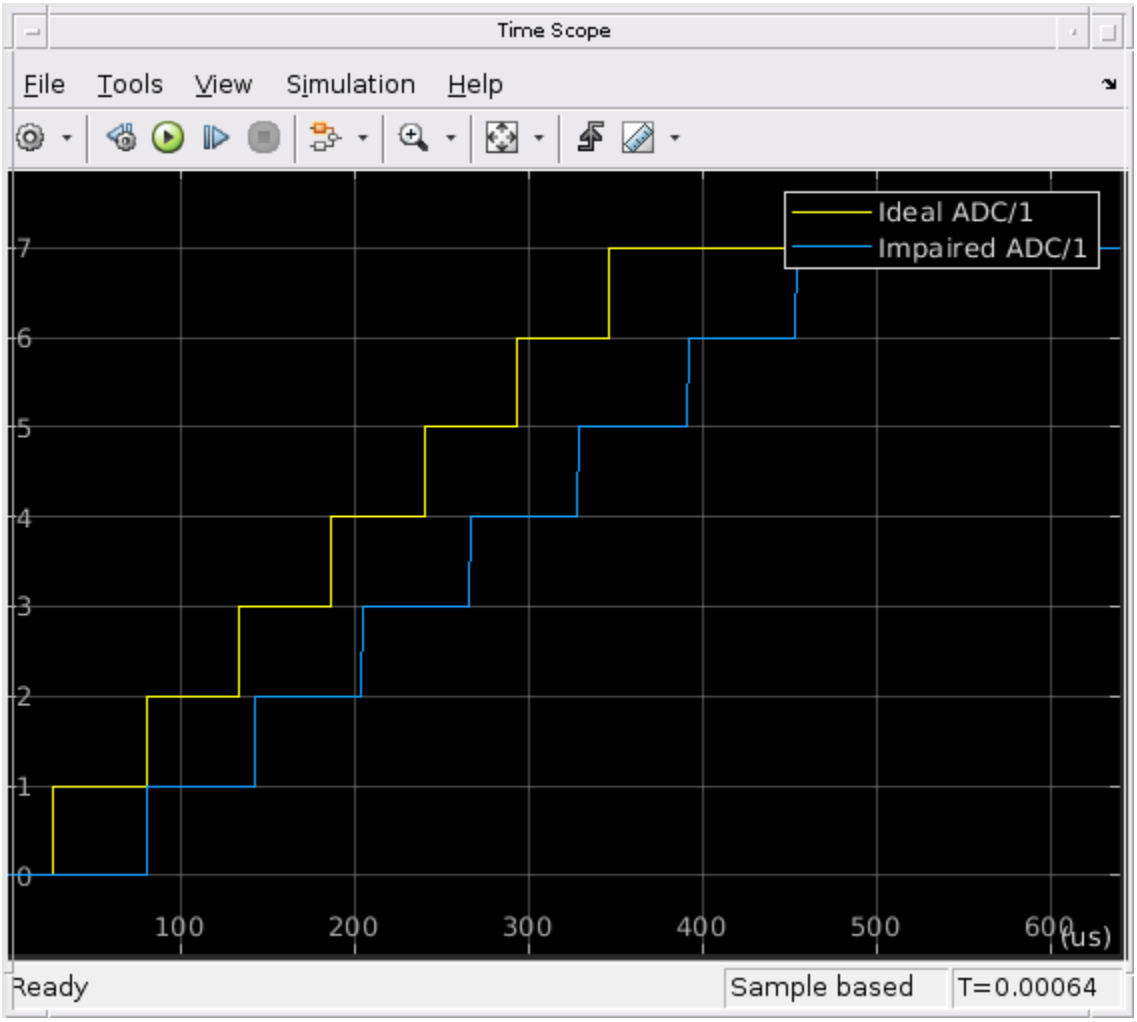
The sum of offset error and gain error is known as full-scale error. In monopolar converters zero error, essentially offset error defined at analog level `0`, is identical to regular offset error. In bipolar converters such as those above, offset error and zero error are different quantities.

Use of Linearity Errors as Impairments

Compare a flash ADC with offset and gain error to one with no impairments.

```
model = 'OffsetGainExample';
open_system(model);
open_system([model '/Time Scope']);
sim(model);
```


analog
start
digital
ready
Ideal ADC
analog
start
digital
ready
Impaired ADC
analog
start
digital
ready
ADC
DC Measurement
Offset Error: 1.01 LSB
Gain Error: 0.99 LSB
Max INL (end-point): 0.00 LSB
Max INL (best-fit): 0.11 LSB
Max DNL (end-point): 0.02 LSB
Max DNL (best-fit): 0.05 LSB



The ADC DC Measurement block confirms the values of the impairments to within a margin of error determined by the sample rate of the system. The offset error and gain error were both entered as 1 LSB. Offset error was measured as 1.08 LSB and gain error was measured as 0.97 LSB. Errors in these measurements are due to the converter not sampling exactly at its threshold values. The maximum expected error in LSB is:

$$E_{max} = \frac{m}{f_s}$$

where m is the slope of the input ramp/sawtooth signal in LSB/s, and f_s is the ADC's conversion start frequency.

See Also

Flash ADC | ADC DC Measurement

Compare SAR ADC to Ideal ADC

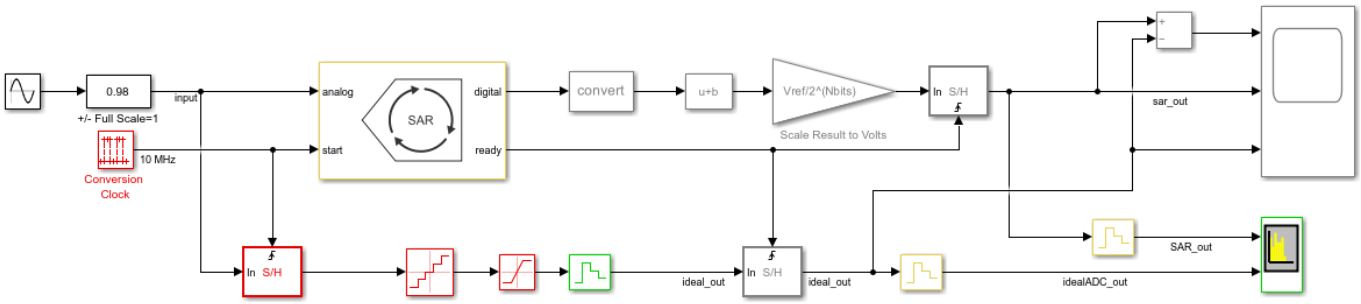
This example shows a comparison of the SAR ADC from the Mixed-Signal Blockset™ to the ideal ADC model with impairments presented in “Analyzing Simple ADC with Impairments” on page 2-27.

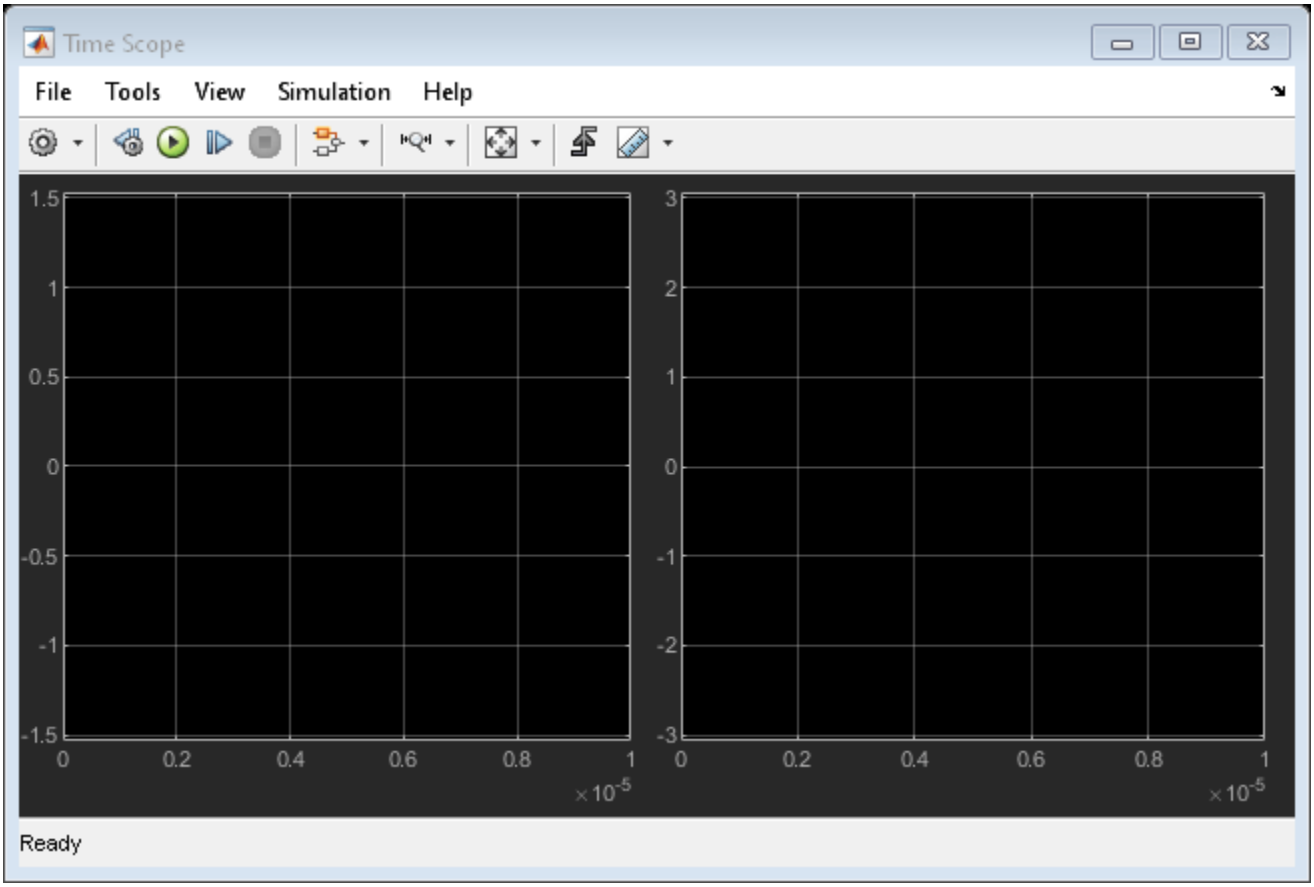
This model compares SAR ADC and an equivalent ideal ADC model. To scale the integer output of the SAR ADC to match the floating point output of the ideal ADC, use an output gain and offset. For the comparison to be valid, both ADCs must sample their inputs and then provide their outputs concurrently.

The ideal ADC model is composed of a Quantizer block, a saturation block and a Zero-Order Hold block. The quantization interval is set to be 1 LSB of the SAR ADC. The limits of the saturation block match the SAR ADC's **Input range** parameter, `[-Vref Vref] / 2`. The Zero-Order Hold and both Sample and Hold blocks sample at *Fs*, though they do so at differnt times.

The SAR ADC outputs an unsigned, 16-bit integer. The minimum value of the converter's output is 0 and maximum value is 2^ **NBits** - 1. This is less than the maximum value of the integer as long as the integer has more bits than the converter. The Bias and Scale Results to Volts blocks scale the former integer onto the same range as the converter input and the output of the ideal ADC, by default -*Vref* `/ 2` to *Vref* `/ 2`.

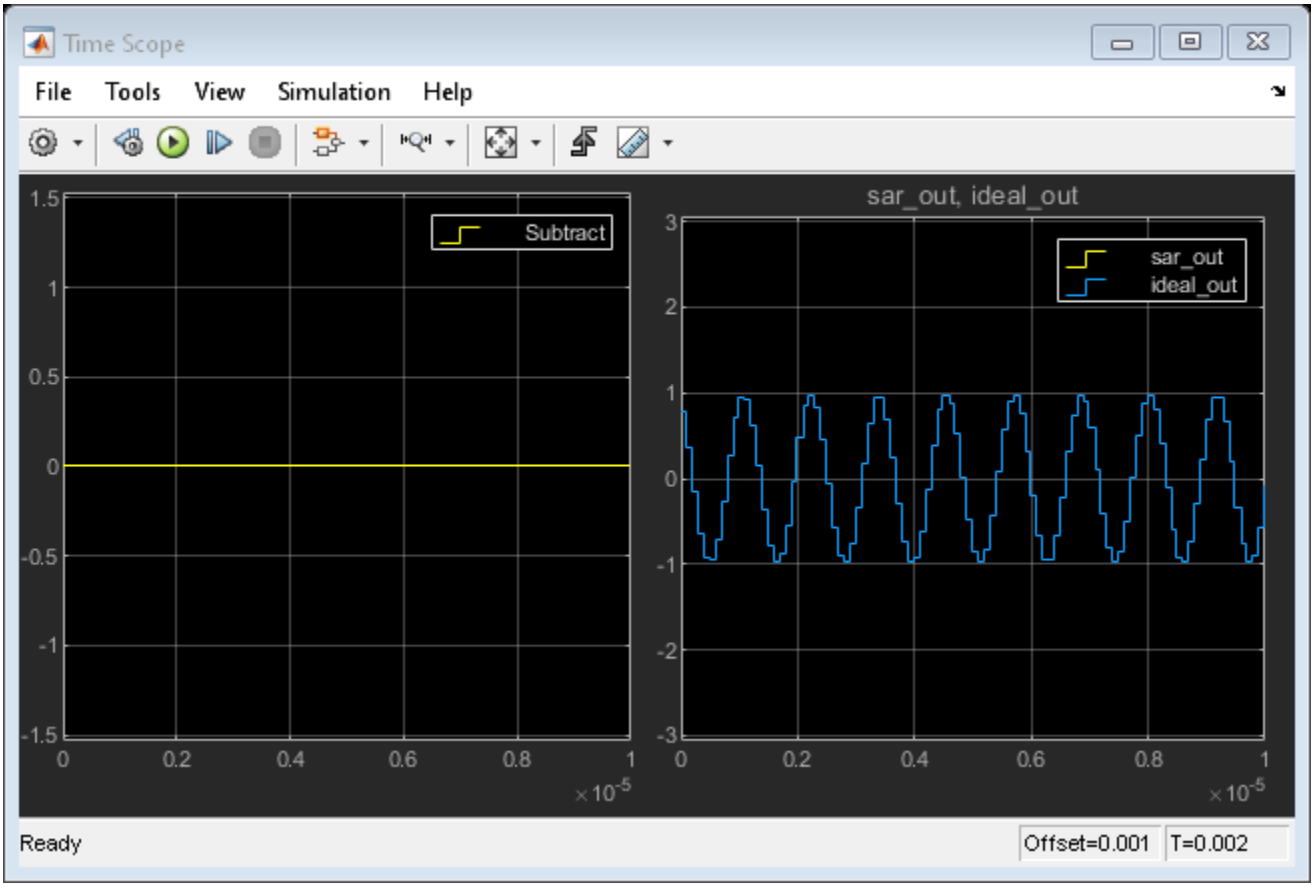
```
model = 'SARIdealCompare';
open_system(model);
```





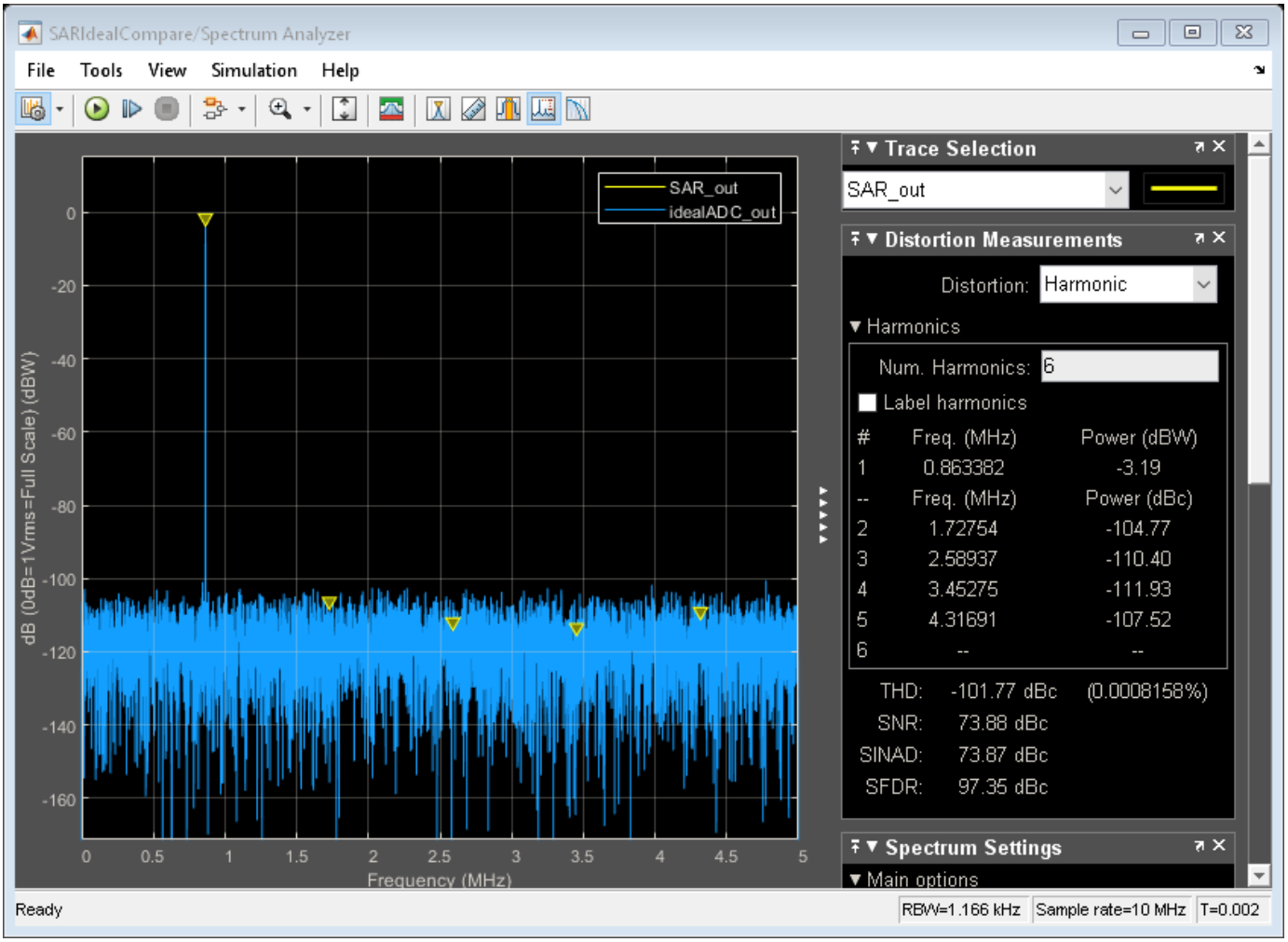
The Time Scope block shows the difference between the two outputs on the left and the two outputs themselves on the right.

```
open_system([model '/Time Scope']);
sim(model);
```



The Spectrum Analyzer block shows the spectra of the two outputs.

```
close_system([model '/Time Scope']);
open_system([model '/Spectrum Analyzer']);
sim(model);
```



You can substitute the Flash ADC for the SAR ADC in this model to measure its performance.

See Also

SAR ADC | Flash ADC

More About

- “Design and Evaluate Successive Approximation ADC Using Stateflow” on page 2-19

ADC Linearity Measurement Using Histogram

Statistical measurement of ADC Linearity is popular in physical systems due to its noise tolerance and relative simplicity of implementation in a physical environment. This example covers problems inherent to statistical linearity measurements and some possible solutions for implementing these measurements in Simulink®.

The Model

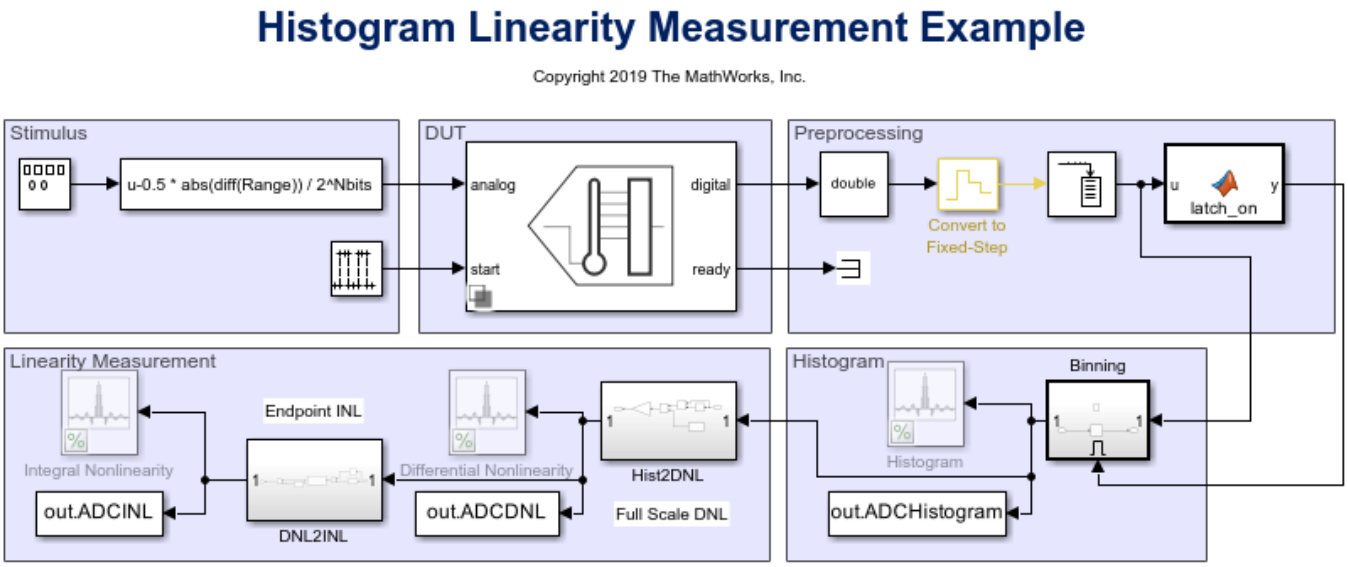
A histogram-based linearity measurement can be broken into five sections:

1. Stimulus
2. Device Under Test (DUT)
3. Preprocessing
4. Histogram Computation
5. Linearity Measurement

The problems inherent to the statistical method occur primarily in the Stimulus and Preprocessing sections. The stimulus signal needs to be uniform over the ADC's input to give a flat histogram for an unimpaired converter, while the preprocessing needs to ensure that exactly one period of the stimulus is binned by the histogram at a time.

Alternatively, the stimulus frequency can be nonuniform but the histogram must be manipulated using the PDF of the nonuniform input so as to remove the influence of the nonuniformity on the result.

```
s = warning('off', 'Simulink:blocks:DivideByZero');
model = 'histogramLinearityExample';
open_system(model);
set_param([model '/Flash ADC'], 'EnableImp', 'off');
```



The sawtooth stimulus frequency automatically adapts to different combinations of simulation runtime and converter number of bits.

$$F_{sawtooth} = \max\left\{\frac{|V_{max} - V_{min}|}{T_{stop}}, \frac{F_{sample}}{2^{(2N_{bits})}}\right\}$$

V_{max} and V_{min} are defined in *Range*, T_{stop} is defined in the model parameter **StopTime**, F_{sample} is defined by *Fs* and N_{bits} is defined by *Nbits*. $F_{sample}/2^{2N_{bits}}$ serves to prevent a frequency of 0 in the case of a non-finite simulation time. Otherwise, the sawtooth period takes the entire simulation time.

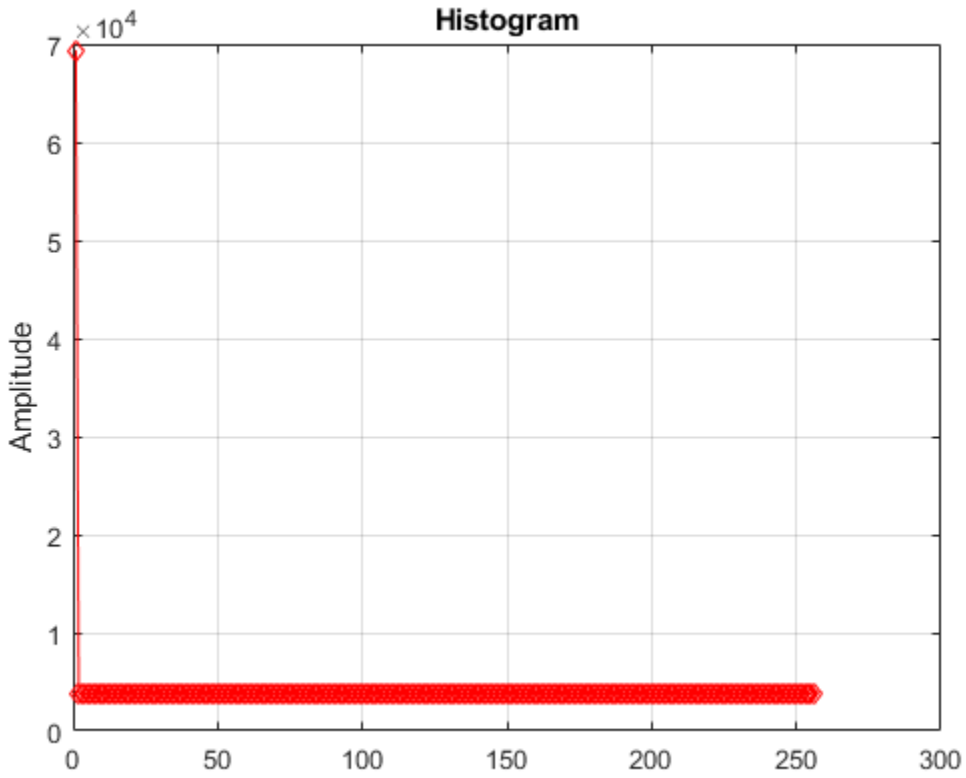
Buffering

In this model, the samples are buffered separately from the histogram computation for two reasons. First, it is important that the histogram updates at an integer multiple of the stimulus period. This ensures that the histogram has the correct shape. Second, the number of samples per period of the sawtooth stimulus is dependant on its frequency and the frequency of the conversion start signal, which is the sampling frequency. Therefore, the number of samples to buffer is the sample frequency divided by the sawtooth frequency.

$$N = \frac{F_{sample}}{F_{sawtooth}}$$

The buffer block causes its own problem while ensuring histogram uniformity. The initial output of the buffer is passed to the histogram at the start of simulation, effectively a buffer full of zeros all determined by an initial condition.

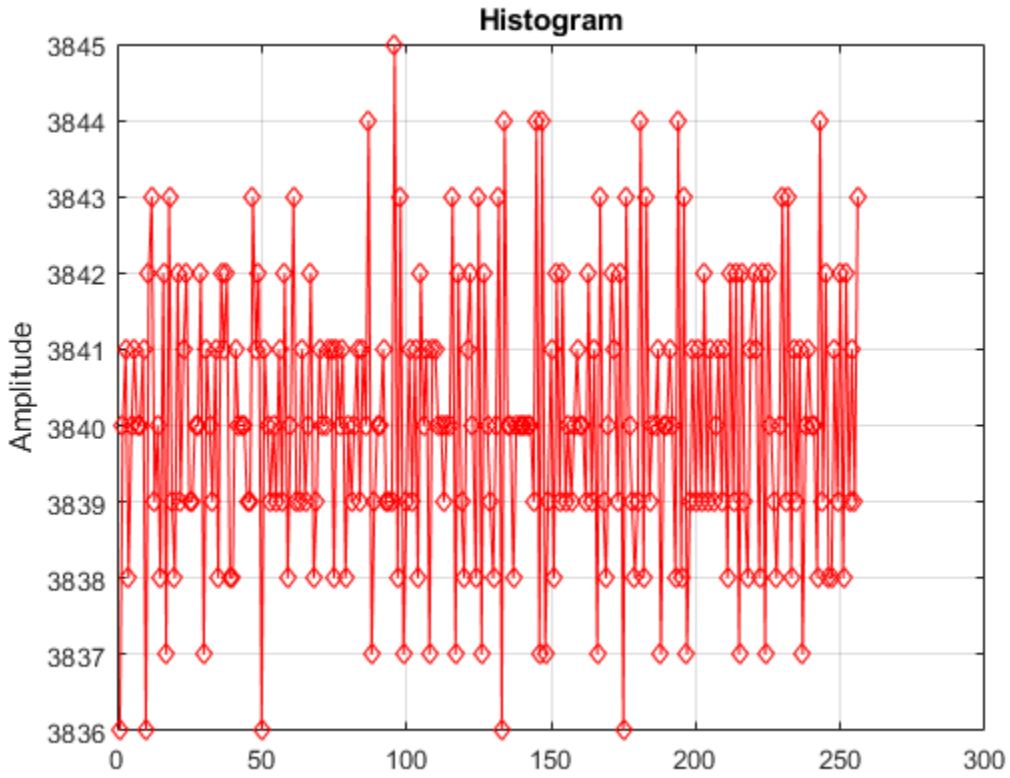
```
model3 = 'histogramInitialCondition';
load_system(model3);
%open_system([model '/Histogram']);
out=sim(model3);
plot(out.ADCHistogram,'-rd');
grid on;
title('Histogram');
ylabel('Amplitude');
```



The resulting histogram is corrupted by the massive spike at 0, due to the propagation of the initial condition.

The MATLAB Function block in the Histogram section keeps the histogram disabled until the next buffer, the first one filled with simulation data, arrives at the histogram. Therefore, this prevents the final histogram from being dependent on initial conditions.

```
open_system(model);
set_param([model '/Flash ADC'], 'EnableImp', 'off');
out=sim(model);
plot(out.ADCHistogram,'-rd');
grid on;
title('Histogram');
ylabel('Amplitude');
```



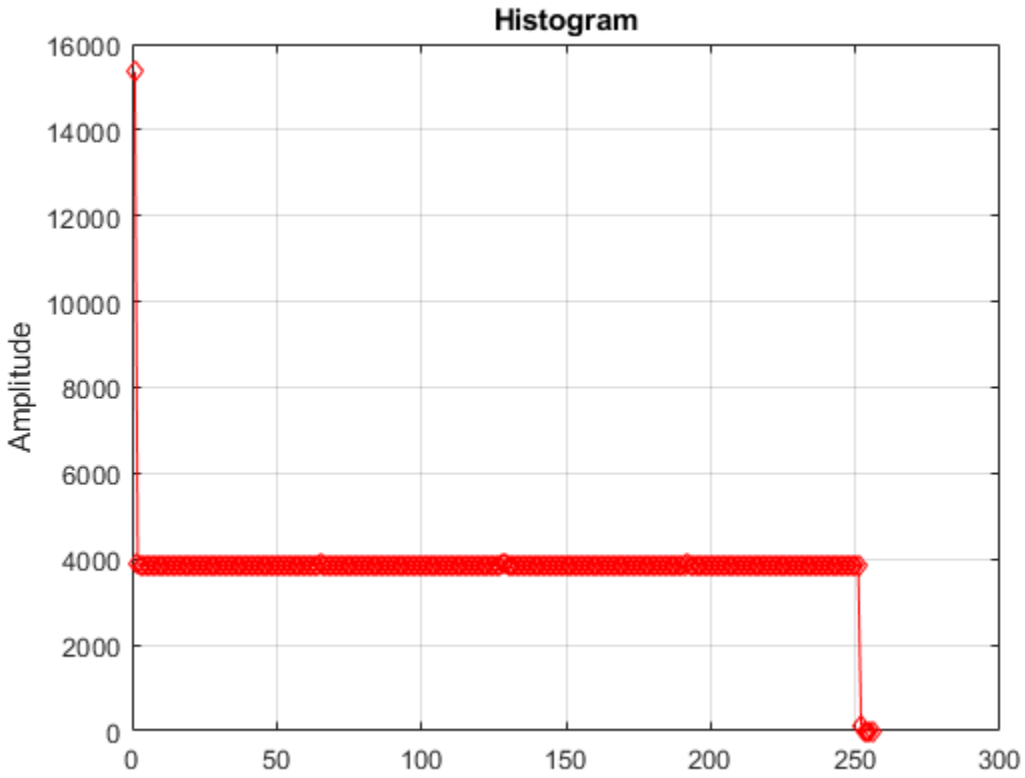
Input Uniformity

A ramp or sawtooth input provides a linear sweep of all of the values over the ADC's dynamic range.

However, due to threshold placement in ADCs, an ideal ADC's transfer curve is not centered in its dynamic range; the ideal transfer curve is left shifted by 0.5 LSB. Therefore, the first threshold is at 0.5 LSB and the last threshold is 1.5 LSB lower than the maximum dynamic range. Compensate for this effect by adding a -0.5 LSB bias to the stimulus.

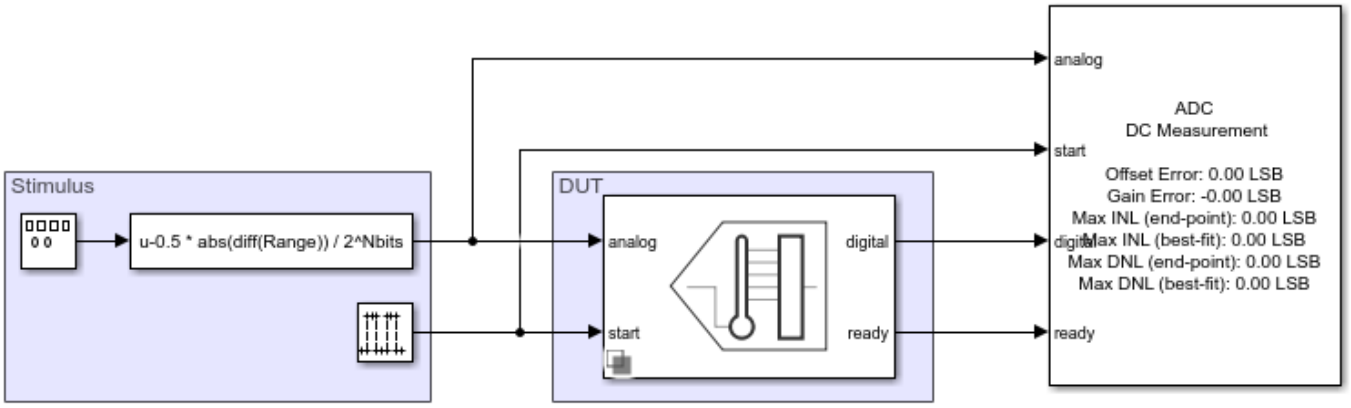
More severe problems occur with converters that have large offset and/or gain errors. Specifically, if the offset error is lower than -1 LSB or the full scale error (the sum of offset and gain error) exceeds 1 LSB, some codes at the bottom or top of the transfer curve (respectively) will be measured as missing, regardless of their actual functionality. Knowing this, statistical linearity measurement techniques should only be applied after offset and gain errors have been corrected.

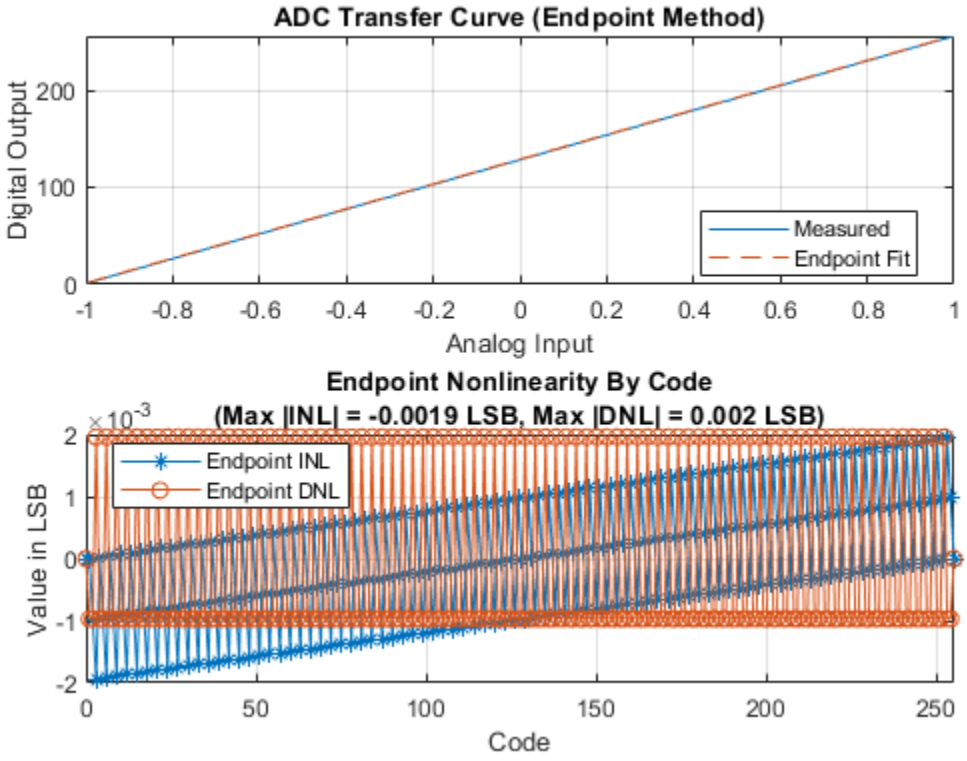
```
set_param([model '/Flash ADC'], 'EnableImp', 'on');
set_param([model '/Flash ADC'], 'OffsetError', '3');
set_param([model '/Flash ADC'], 'GainError', '2');
%open_system([model '/Histogram']);
out=sim(model);
plot(out.ADCHistogram,'-rd');
grid on;
title('Histogram');
ylabel('Amplitude');
```

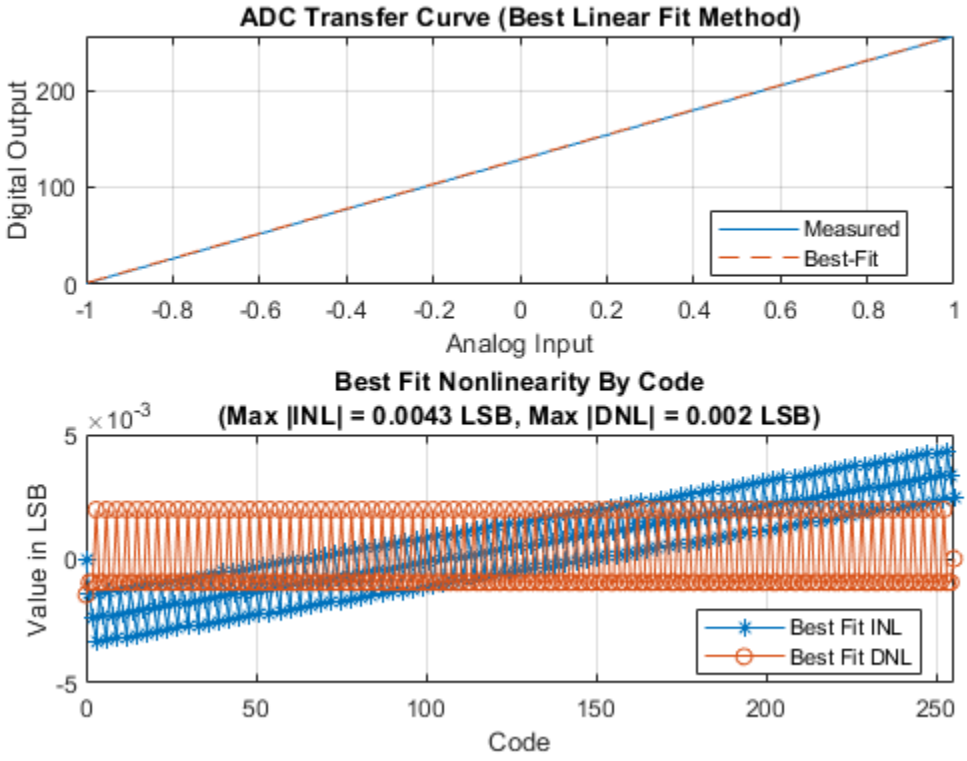


To measure the full transfer curve of an ADC with large offset and/or full scale errors in Simulink®, use the ADC DC Measurement block or the ADC Testbench. Configuration of the stimulus to cover the whole ADC transfer curve will not effect these blocks' results the way it will effect the histogram.

```
model2 = 'flashWithTestbench';
open_system(model2);
set_param([model2 '/Flash ADC'], 'EnableImp', 'off');
set_param([model2 '/Flash ADC'], 'OffsetError', '3');
set_param([model2 '/Flash ADC'], 'GainError', '2');
sim(model2);
msblks.ADC.adcDcMeasurementPlot([model2 '/ADC DC Measurement']);
```



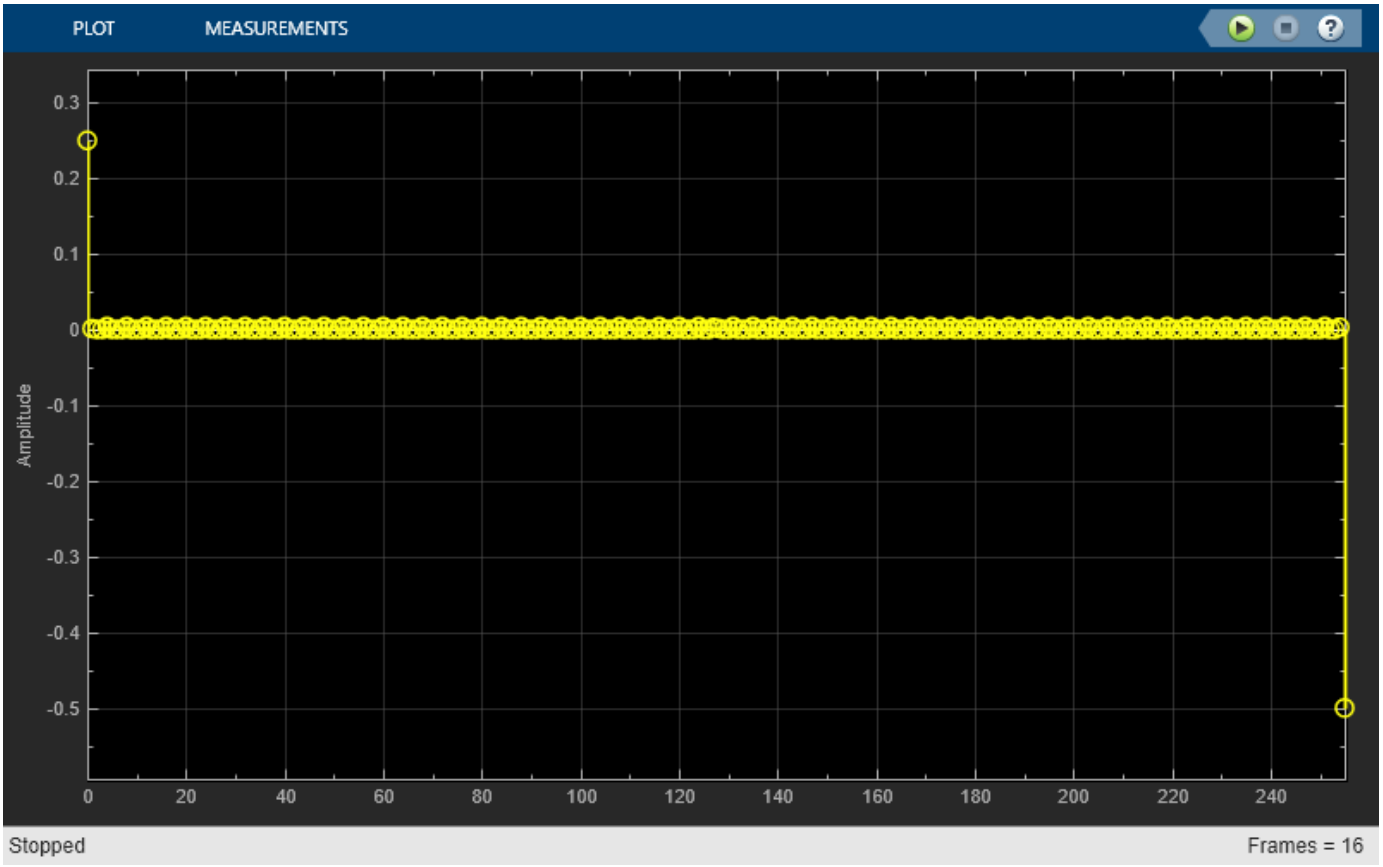

ADC Transfer Curve (Endpoint Method)
Digital Output
Analog Input
Measured
Endpoint Fit
Endpoint Nonlinearity By Code
(Max |INL| = -0.0019 LSB, Max |DNL| = 0.002 LSB)
Endpoint INL
Endpoint DNL
Value in LSB
Code

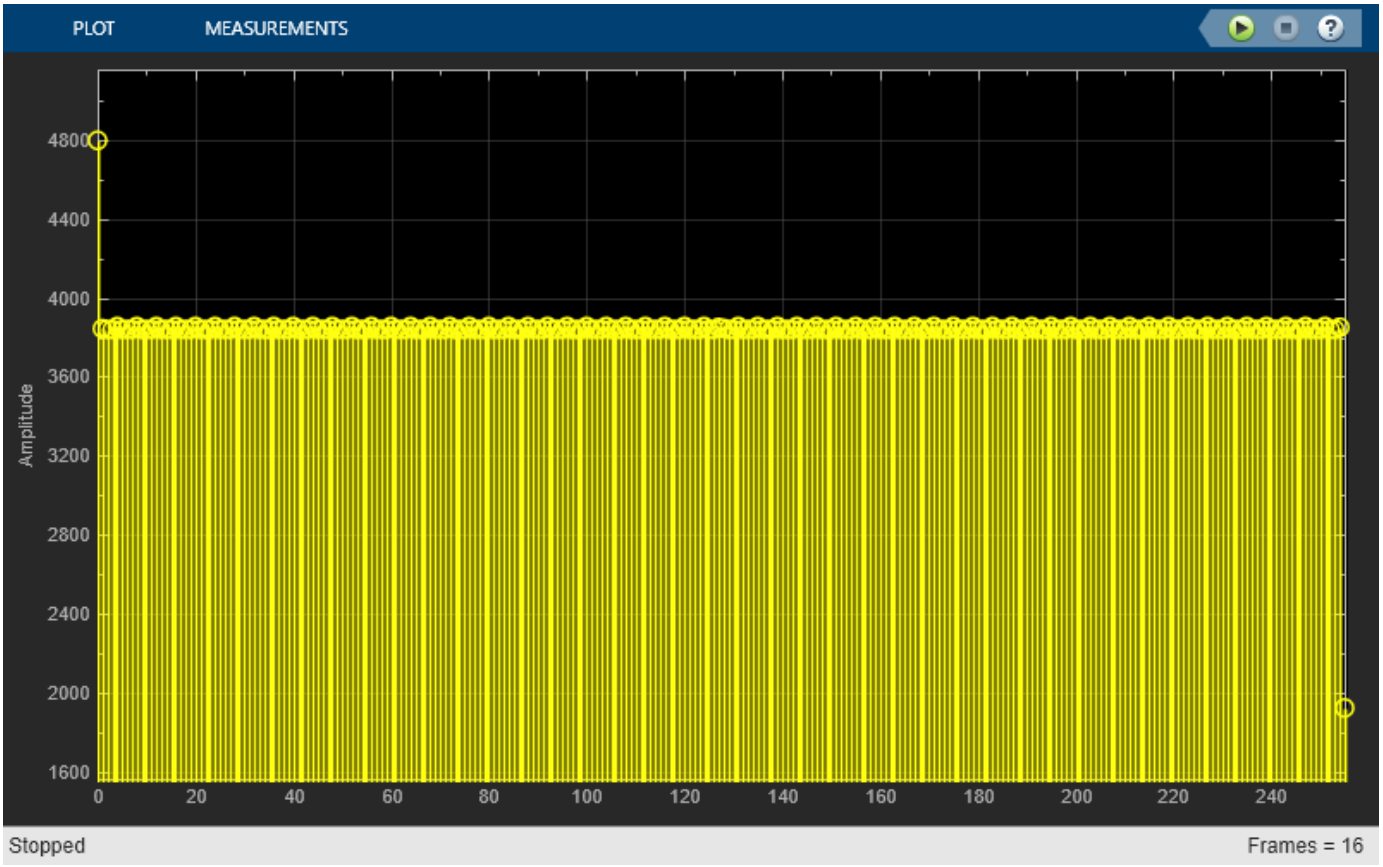


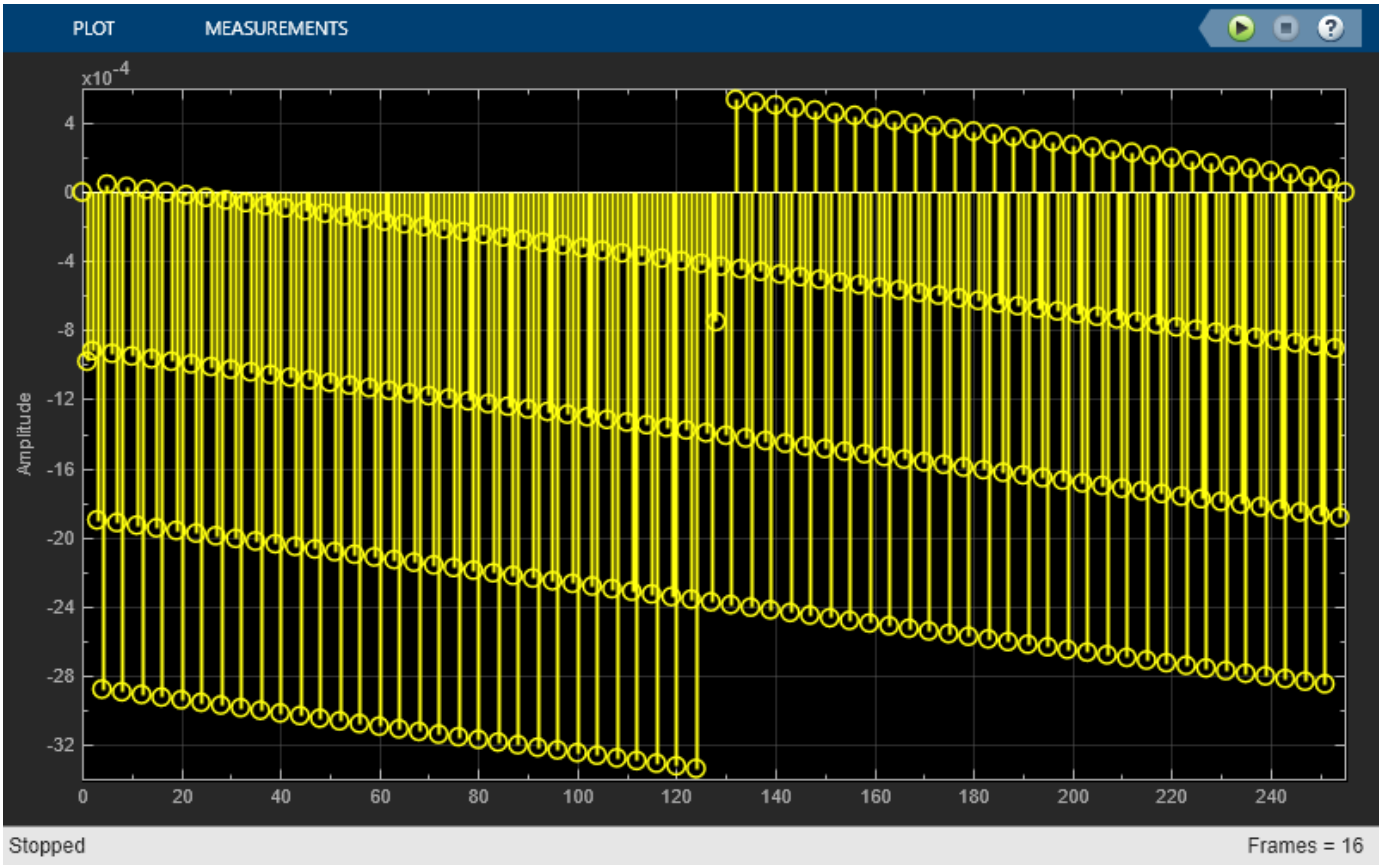
Linearity Measurement

With the input and preprocessing conditions met, the histogram provides an excellent visual representation of the linearity of the data converter. A set of mathematical operations applied to this histogram yield measurements for both DNL and INL.

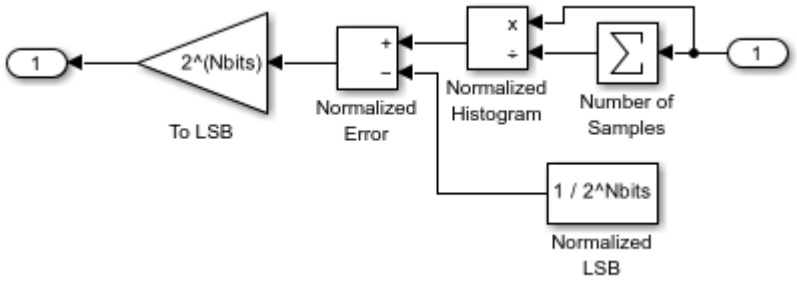
```
set_param([model '/Flash ADC'], 'EnableImp', 'on');
set_param([model '/Flash ADC'], 'OffsetError', '0.25');
set_param([model '/Flash ADC'], 'GainError', '0.25');

set_param([model '/Differential Nonlinearity'],'Commented','off');
set_param([model '/Integral Nonlinearity'],'Commented','off');
set_param([model '/Histogram'],'Commented','off');

open_system([model '/Differential Nonlinearity']);
open_system([model '/Integral Nonlinearity']);
open_system([model '/Histogram']);
sim(model);
```


PLOT
MEASUREMENTS
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
Amplitude
0
20
40
60
80
100
120
140
160
180
200
220
240
Stopped
Frames = 16


PLOT
MEASUREMENTS
4800
4400
4000
3600
3200
2800
2400
2000
1600
Amplitude
0
20
40
60
80
100
120
140
160
180
200
220
240
Stopped
Frames = 16

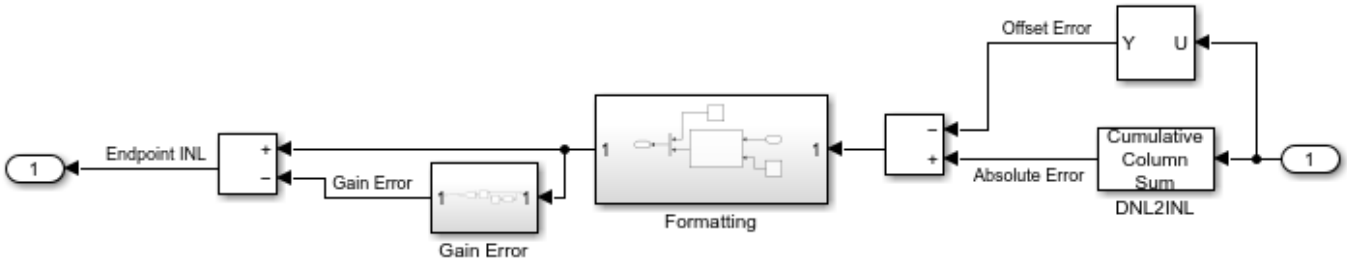



```
open_system([model '/Hist2DNL']);
```



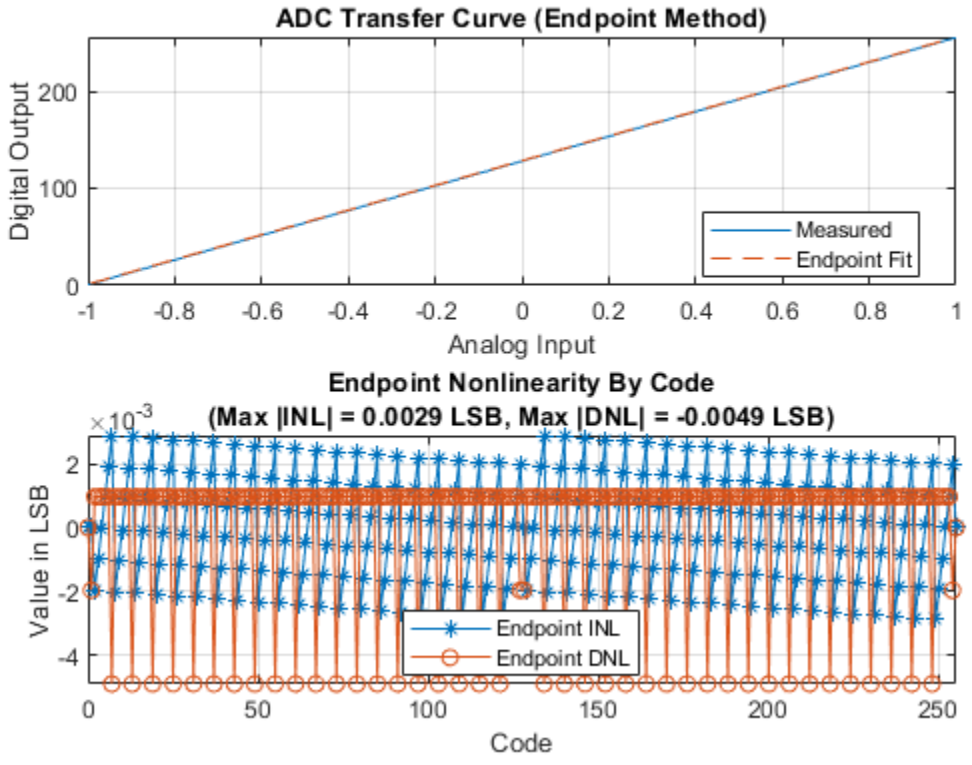
INL is subsequently computed from DNL.

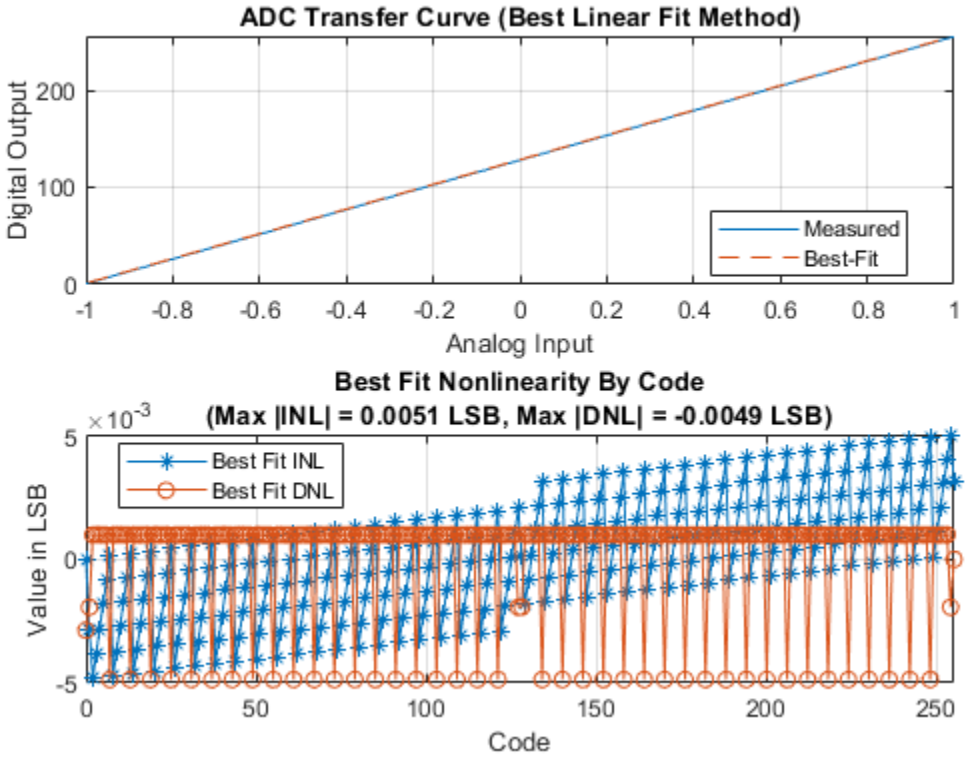
```
open_system([model '/DNL2INL']);
```



Endpoint INL computed in this manner is equivalent to the endpoint INL computed by the ADC DC Measurement block, though the ADC DC Measurement block does not report full scale DNL.

```
open_system(model2);
set_param([model2 '/Flash ADC'], 'EnableImp', 'on');
set_param([model2 '/Flash ADC'], 'OffsetError', '0.25');
set_param([model2 '/Flash ADC'], 'GainError', '0.25');
sim(model2);
msblks.ADC.adcDcMeasurementPlot([model2 '/ADC DC Measurement']);
```






```
warning(s.state, 'Simulink:blocks:DivideByZero');
```

Oversampling Interpolating DAC

This example shows how to model a 12-bit Oversampling Interpolating DAC.

Oversampling Interpolating DACs use interpolation to achieve a higher resolution output than provided on their input. This allows less complicated output filtering for a DSP system operating close to its Nyquist rate.

Model

The oversampling interpolating DAC has three basic functional blocks. The first block is an FIR Interpolation filter block from the DSP System Toolbox™ to increase the sample rate from the input sample rate of `Fs` to the DAC's sample rate of `M * Fs`. The second block is a MATLAB Function block that handles unbuffering the output of the FIR Interpolation block. The MATLAB Function block has a clock-triggered nature which allows it's output to have jitter. This gives it advantage over the Unbuffer block from the DSP System Toolbox™. The third block is a Mixed-Signal Blockset™ DAC operating at `M * Fs`.

The parameters of the DAC are from the AD9773 datasheet.

The workspace variable parameters:

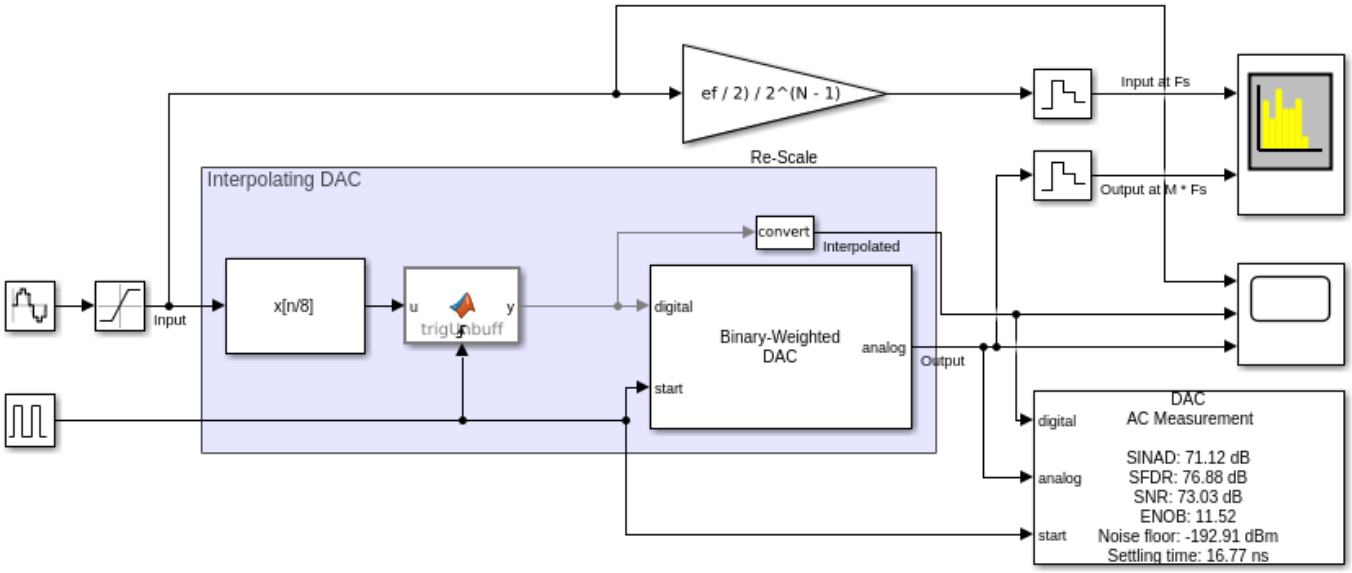
- `Fin = 66.176` kHz is the input signal frequency.
- `Fs = 1.125` MHz is the input sample frequency.
- `M = 8` is the oversample factor/ratio.
- `N = 12` is the number of bits of the DAC.
- `Ref = 1.2` is the reference (dynamic output range) of the DAC.

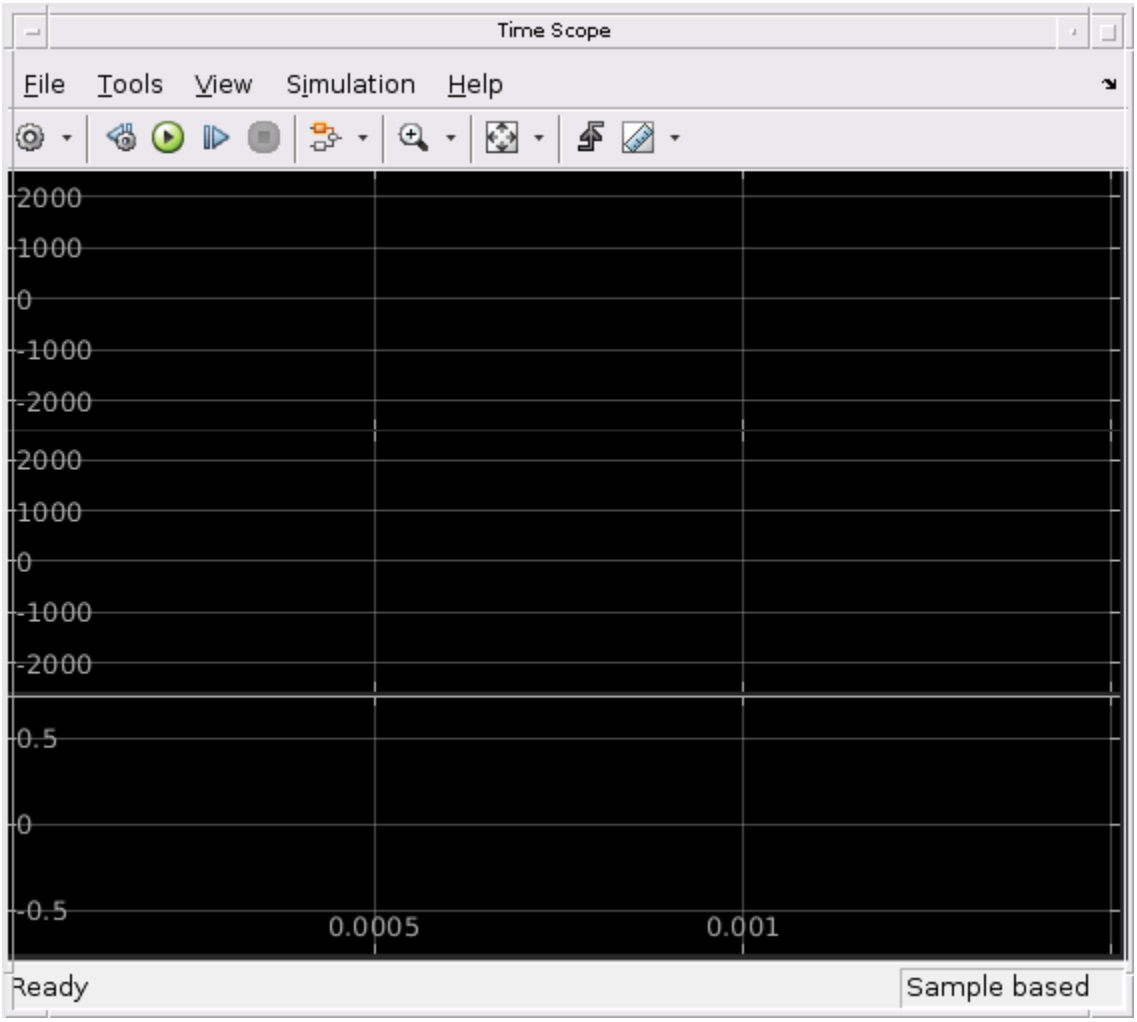
The DAC sample frequency is determined by the oversample factor and the input sample frequency:

$$F_{s_{DAC}} = M \cdot F_s = 9MHz$$

For simplicity, **Offset error** and **Gain error** have been left at `0 %FS` in this model.

```
model = 'InterpolatingDAC';
open_system(model);
```



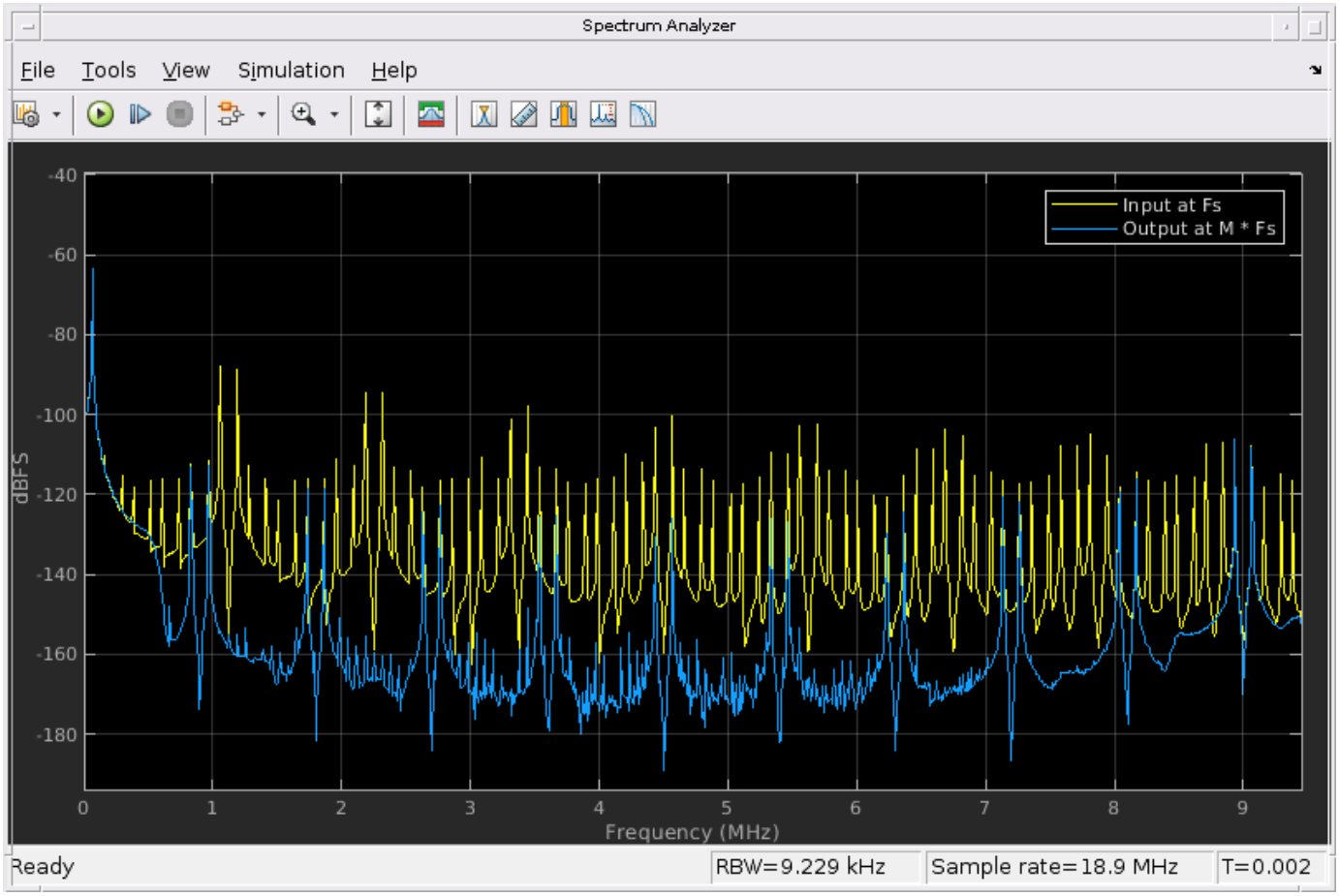


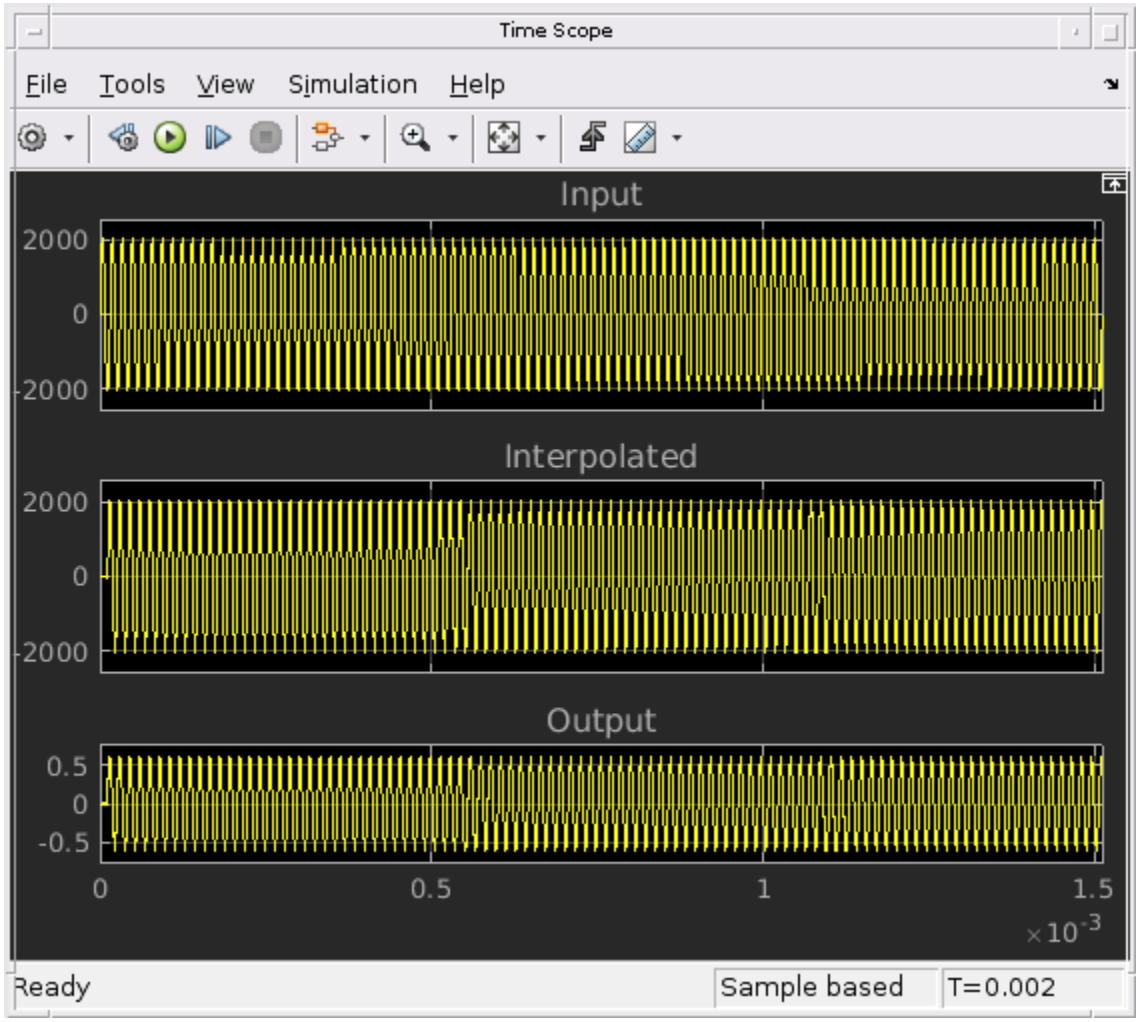
Dynamic Testing

To determine SNR, ENOB and other dynamic characteristics of the interpolating DAC, use the DAC AC Measurement block from the Mixed-Signal Blockset™.

Use the Spectrum Analyzer to compare the low-sample rate input to the output of the oversampled interpolating DAC. The gain block next to the Spectrum Analyzer matches the input wave to the amplitude of the DAC's output for side-by-side comparison.

```
open_system([model '/Spectrum Analyzer']);
sim(model);
```

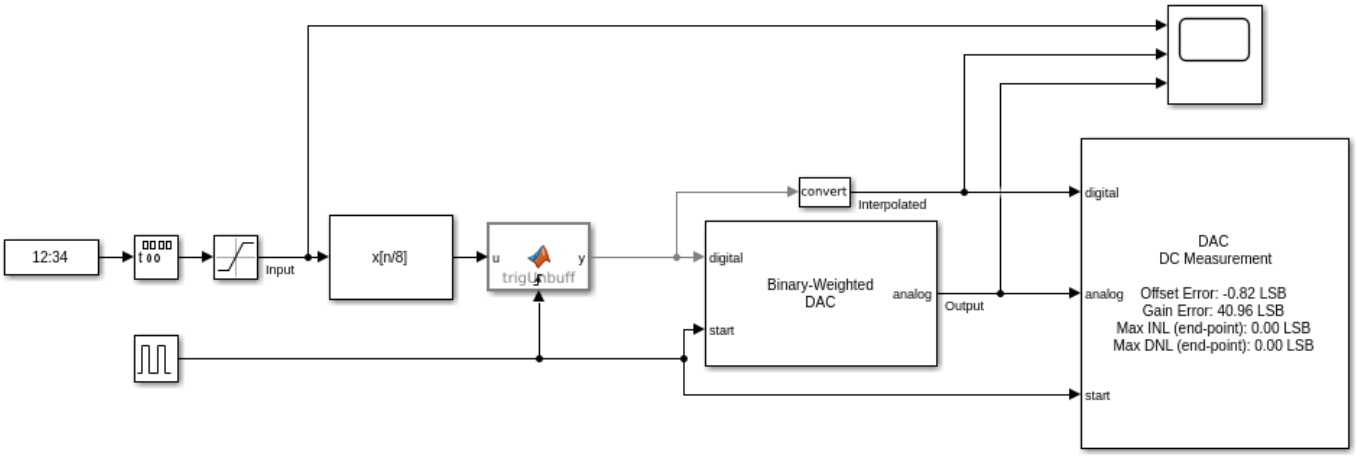

Spectrum Analyzer
File Tools View Simulation Help
Input at Fs
Output at M * Fs
dBFS
-40
-60
-80
-100
-120
-140
-160
-180
0 1 2 3 4 5 6 7 8 9
Frequency (MHz)
Ready
RBW=9.229 kHz
Sample rate=18.9 MHz
T=0.002

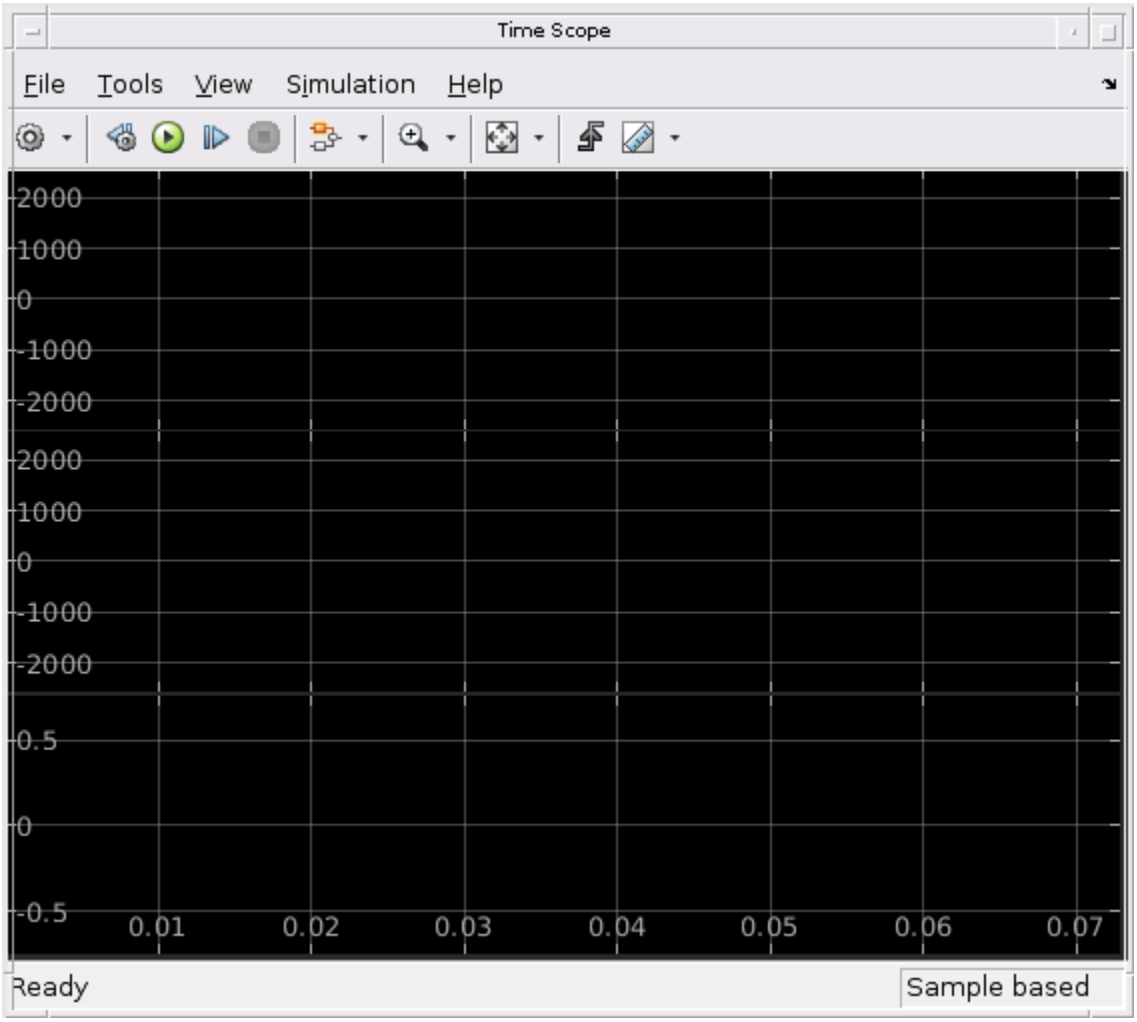


Static Testing

To determine offset error, gain error, INL and DNL use the DAC DC Measurement block. Set the **Offset error** of the Binary Weighted DAC to `-0.02 %FS` (-0.8192 LSB) and set its the **Gain error** to `1.0 %FS` (40.96 LSB).

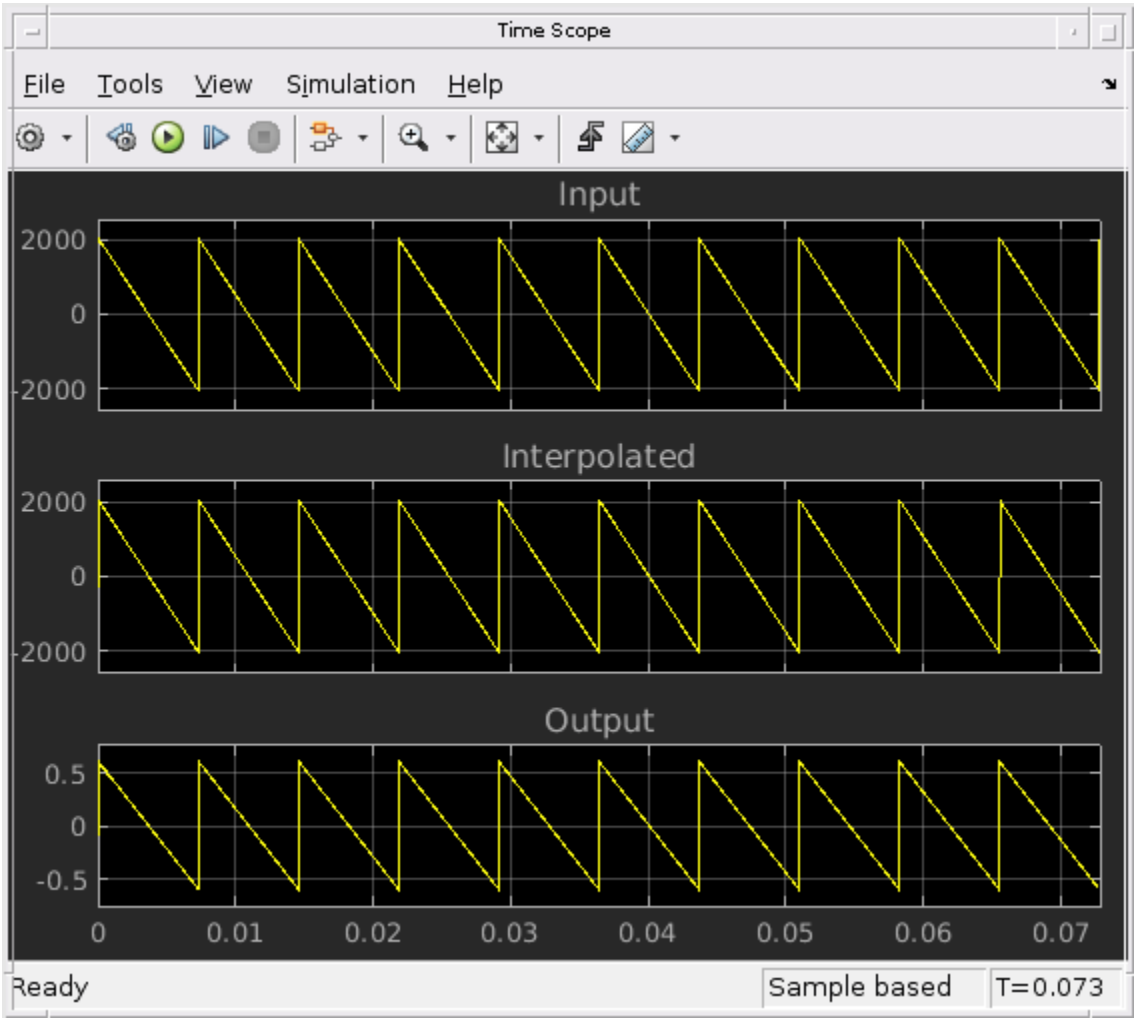
```
bdclose(model);
model = 'InterpolatingDACDC';
open_system(model);
```

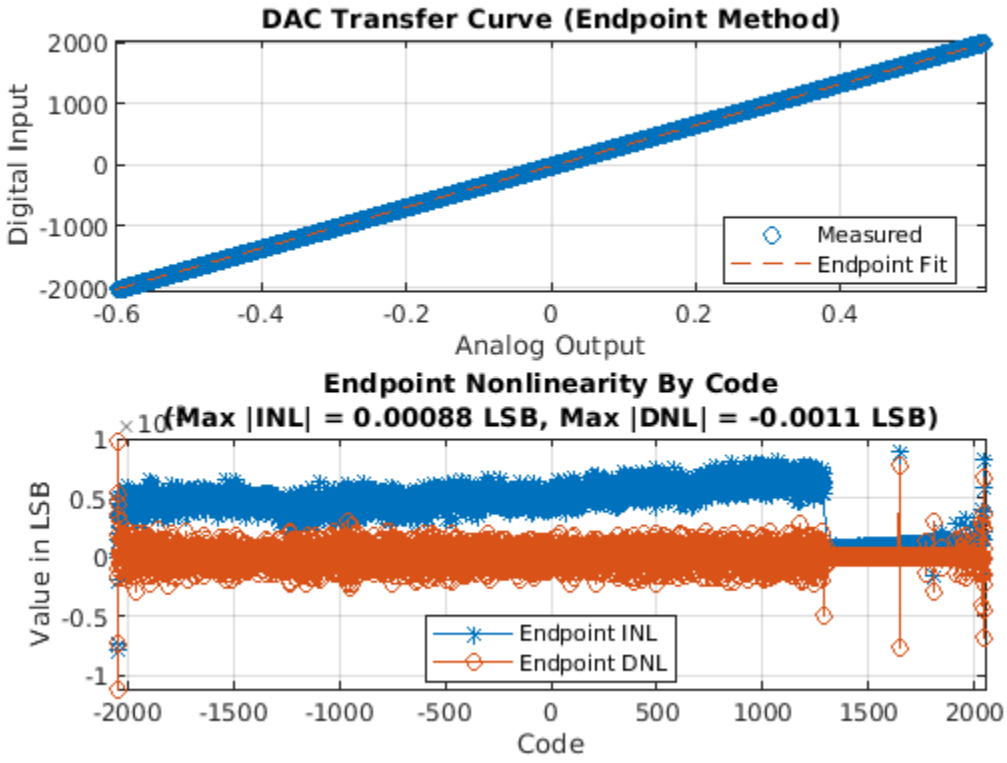




A summary of the measurements is reported on the block icon. Open the block mask and press the Plot button to view the full INL and DNL plots.

```
sim(model);
mask = Simulink.Mask.get([model '/DAC DC Measurement']);
button = mask.getDialogControl('PlotBtn');
eval(regexprep(button.Callback, 'gcb',['"' model '/DAC DC Measurement"']));
```


Time Scope
File Tools View Simulation Help
Input
2000
0
-2000
Interpolated
2000
0
-2000
Output
0.5
0
-0.5
0 0.01 0.02 0.03 0.04 0.05 0.06 0.07
Ready
Sample based
T=0.073


DAC Transfer Curve (Endpoint Method)
Digital Input
Analog Output
Measured
Endpoint Fit
Endpoint Nonlinearity By Code
(Max |INL| = 0.00088 LSB, Max |DNL| = -0.0011 LSB)
Value in LSB
Code
Endpoint INL
Endpoint DNL

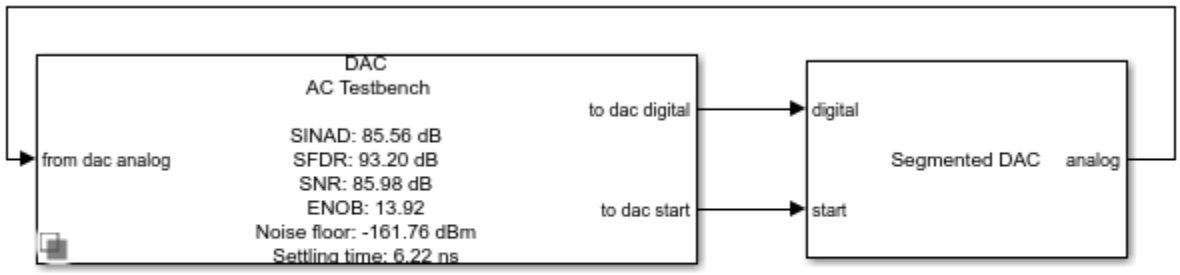
Design and Evaluate Segmented DAC

This example shows how to design and evaluate a segmented DAC using reference architecture and validate the DAC using the DAC Testbench. For this example, use the datasheet of AD9775. This is a commercial, off-the-shelf 14-bit DAC from Analog Devices.

Set Up Segmented DAC Testbench Model

Open the model SegmentedDACTestbench attached to this example as a supporting file. The model consists of a Segmented DAC block and a DAC Testbench.

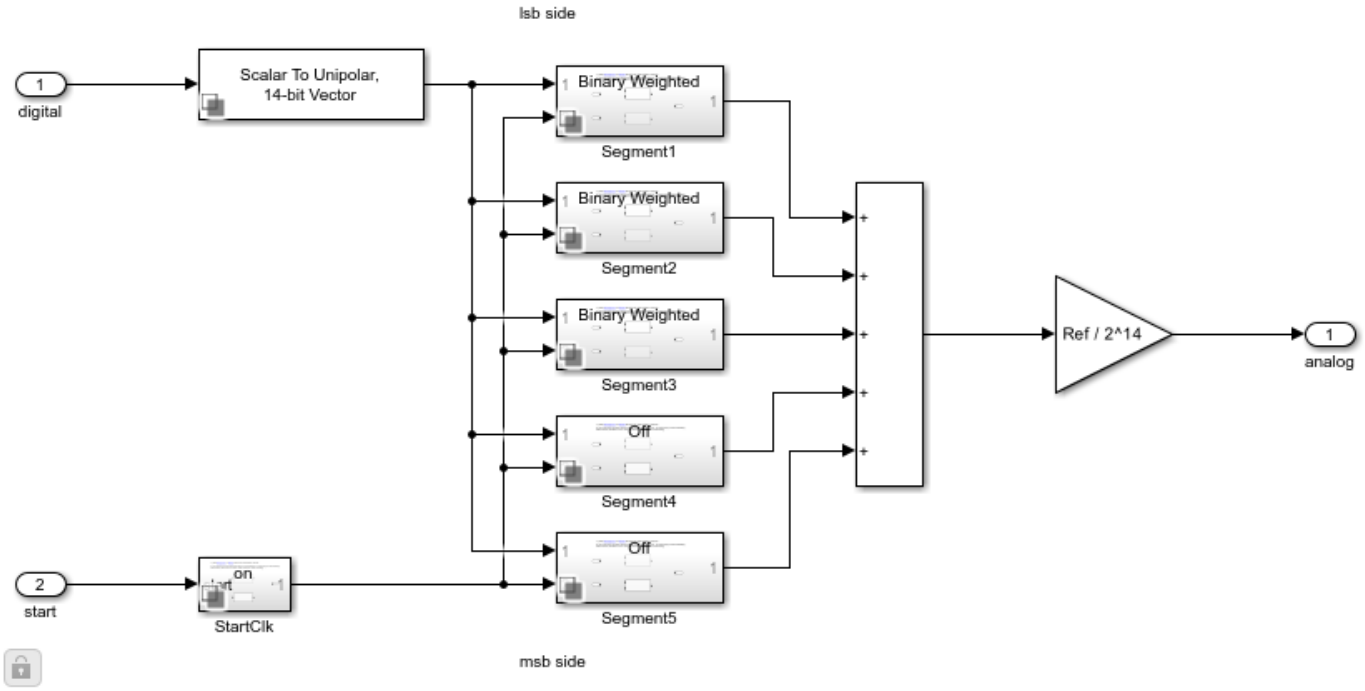
```
model = 'SegmentedDacTestbench';
open_system(model);
```



Segmented DAC Block

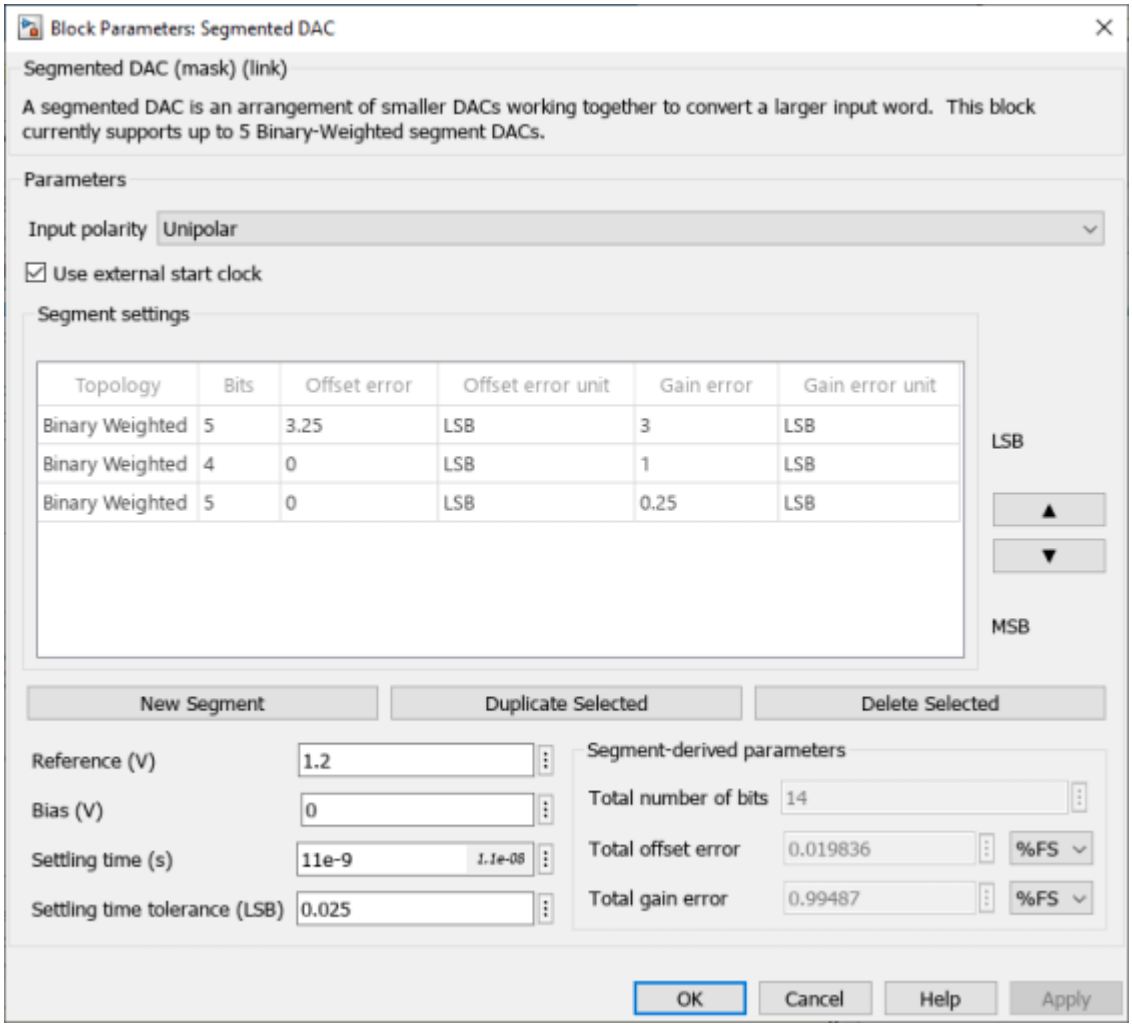
The Segmented DAC block is composed of a number of segments, wired in parallel. These segments are each fed by a subset of the input data word. The results of the conversions from each segment DAC are summed to get the output for the full input word.

```
open_system([model '/Segmented DAC'], 'force');
```



The Logical Vector Conversion block splits the input word into its component bits. Inside each segment a Selector block picks the subset of the input word for that segment and another Logical Vector Conversion block converts it back to a scalar value for the segment DAC to convert. The individual segment DACs are Binary Weighted DAC blocks. Their parameter settings are set during model initialization by the Segmented DAC block. Finally, the segments' outputs are added and scaled to the reference of the Segmented DAC block.

Double click the Segmented DAC block to open the Block Parameters dialog box. Use the table in the center to set parameters of individual segments of the DAC. The effects of the table parameters on the overall operation of the DAC are summarized in the bottom right corner.



The **Input polarity** is set `Unipolar`, the **Reference (V)** is set to `1.20` V , the **Bias (V)** is set to `0` V, the **Settling time (s)** is set to `11e-9` s, and the **Settling time tolerance (LSB)** is set to `0.025` LSB based on the datasheet.

In the **Segment settings** table insert a row to have three rows to define a three-segment DAC architecture. Set the **Bits** of the top and bottom rows to `5` and the **Bits** of the middle row to `4`. This sets the LSB segment (specified by the top row) and the MSB segment (specified by the bottom row) to each be a 5-bit DAC while the middle segment is a 4-bit DAC. Set the **Offset error** and **Gain Error** fields for all segments to `0`. Click anywhere in a row or Shift+Click on multiple rows in the table to select them. When a row is selected, use the buttons on the right side of the table to move it up or down in the table changing where its segment is in relation to the input word. The top row's segment always converts the LSB and the bottom segment always converts the MSB. Use the buttons below the table to add or remove segments.

Measure DC Performance Metrics Using Endpoint Method

Double click the DAC Testbench block to open the Block Parameters dialog box. The **Measurement** option is selected as `DC`. Set the **Start conversion frequency (Hz)** to `65e6` Hz. In the **Setup** tab, click the **Autofill setup parameters** button to automatically propagate the DAC parameters to the testbench. Set the **Settling time (s)** to `11e-9` s to ensure the DC measurement results are not based

on the transitions between codes. In the **Target Metric** tab, click the **Autofill target metric** button to automatically propagate the DAC target metrics to the testbench. Apply the changes. Set the **Recommended min simulation stop time (s)** as model stop time by clicking the **Set as model stop time** button.

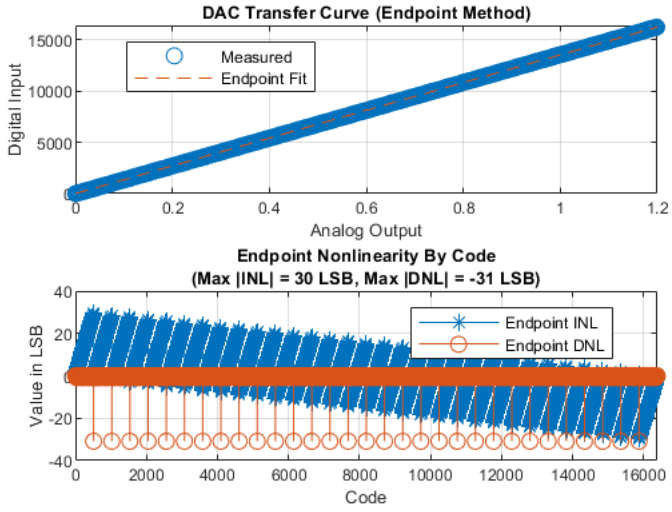
Run the simulation for `5.042` ms.

```
bdclose(model);
sim(model);
```

The measured offset and gain errors displayed on the icon of the DAC Testbench are within the **Error tolerance (LSB)** of their expected values.

$$E_{\%FS} = \frac{100 \cdot E_{LSB}}{2^{N_{bits}}} = \frac{10}{2^{14}} = 0.00061\%FS$$

Double click the DAC Testbench block to open the Block Parameters dialog box. Click the **Plot DC analysis results** button to view the DAC transfer curve, endpoint nonlinearity and best fit nonlinearity.



Measure AC Performance Metrics Using a Single Tone

Double click the DAC Testbench block to open the Block Parameters dialog box. Set the **Measurement option** as `AC` . In the **Stimulus** tab, **Start conversion frequency (Hz)** is set to `65e6` Hz to allow the DAC's output to settle between conversions. In the **Setup** tab, click the **Autofill setup parameters** button to automatically propagate the DAC parameters to the testbench. Apply

the changes. Set the **Recommended min simulation stop time (s)** as model stop time by clicking the **Set as model stop time button**.

```
set_param([model '/DAC Testbench'], 'MeasureOption', 'AC');
```

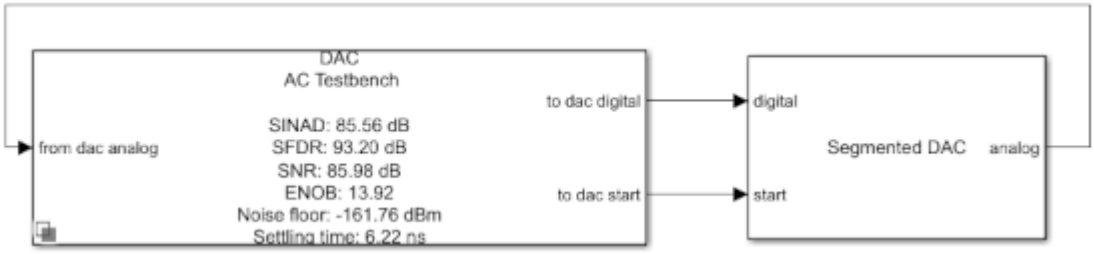
Open the Segmented DAC block parameters dialog and set the **Offset error** and **Gain error** entries in the **Segment settings** table to `0` for all segments. This prevents the linearity impairments from affecting harmonic performance.

```
tablestr = "{'Binary Weighted', '5', '0', 'LSB', '0', 'LSB'; " + ...
            "'Binary Weighted', '4', '0', 'LSB', '0', 'LSB'; " + ...
            "'Binary Weighted', '5', '0', 'LSB', '0', 'LSB'}";
set_param([model '/Segmented DAC'], 'SegmentSettings', tablestr);
```

Run the simulation for `9.5` us.

```
set_param(model, 'StopTime', '9.5e-05');
sim(model);
```

The harmonic distortion measurements are displayed on the icon of the DAC Testbench.



Compare Binary Weighted DAC to Ideal DAC

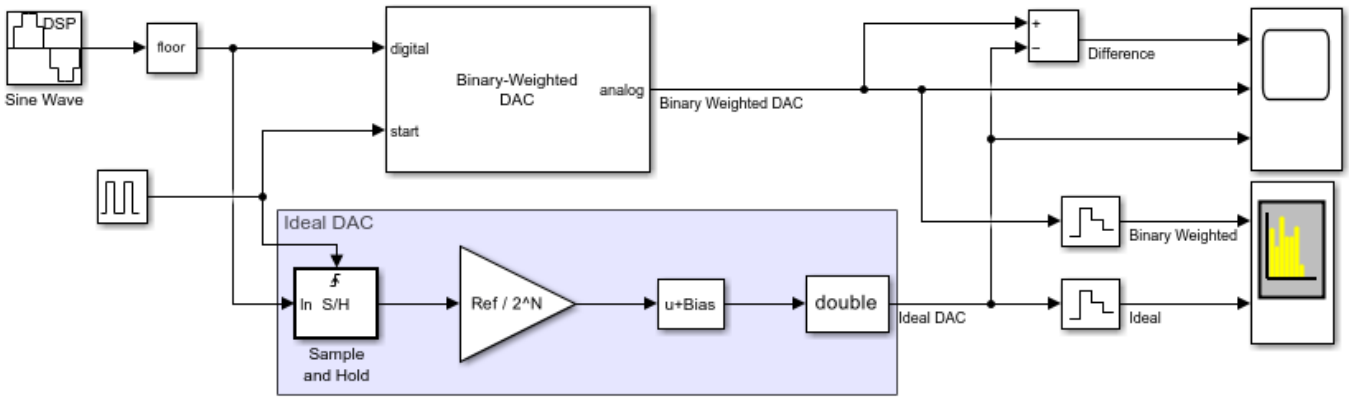
This example shows a comparison of the Binary Weighted DAC from the Mixed-Signal Blockset™ to an ideal DAC model.

This model compares the Binary Weighted DAC and an equivalent ideal DAC model. For the comparison to be valid, both DACs must sample their inputs concurrently. The parameters (reference, bias, sampling frequency, number of bits) of both the Binary Weighted DAC and the ideal DAC are derived from the TLC5615 datasheet.

The ideal DAC model is composed of a Sample and Hold block, Gain block, Bias block and Data Type Conversion block. The Sample and Hold ensures input data enters the DAC at the correct rate (`Fs`) even if the digital input signal was sampled at some other rate. The Gain block scales the digital word to an analog scale and the Bias block applies a bias to match the output of the Binary Weighted DAC. The Binary Weighted DAC uses its reference parameter as its full scale output range, thereby making its throughput gain equivalent to `Ref / (2^NBits - 1)`. The Data Type Conversion block ensures the output data type matches the Binary Weighted DAC.

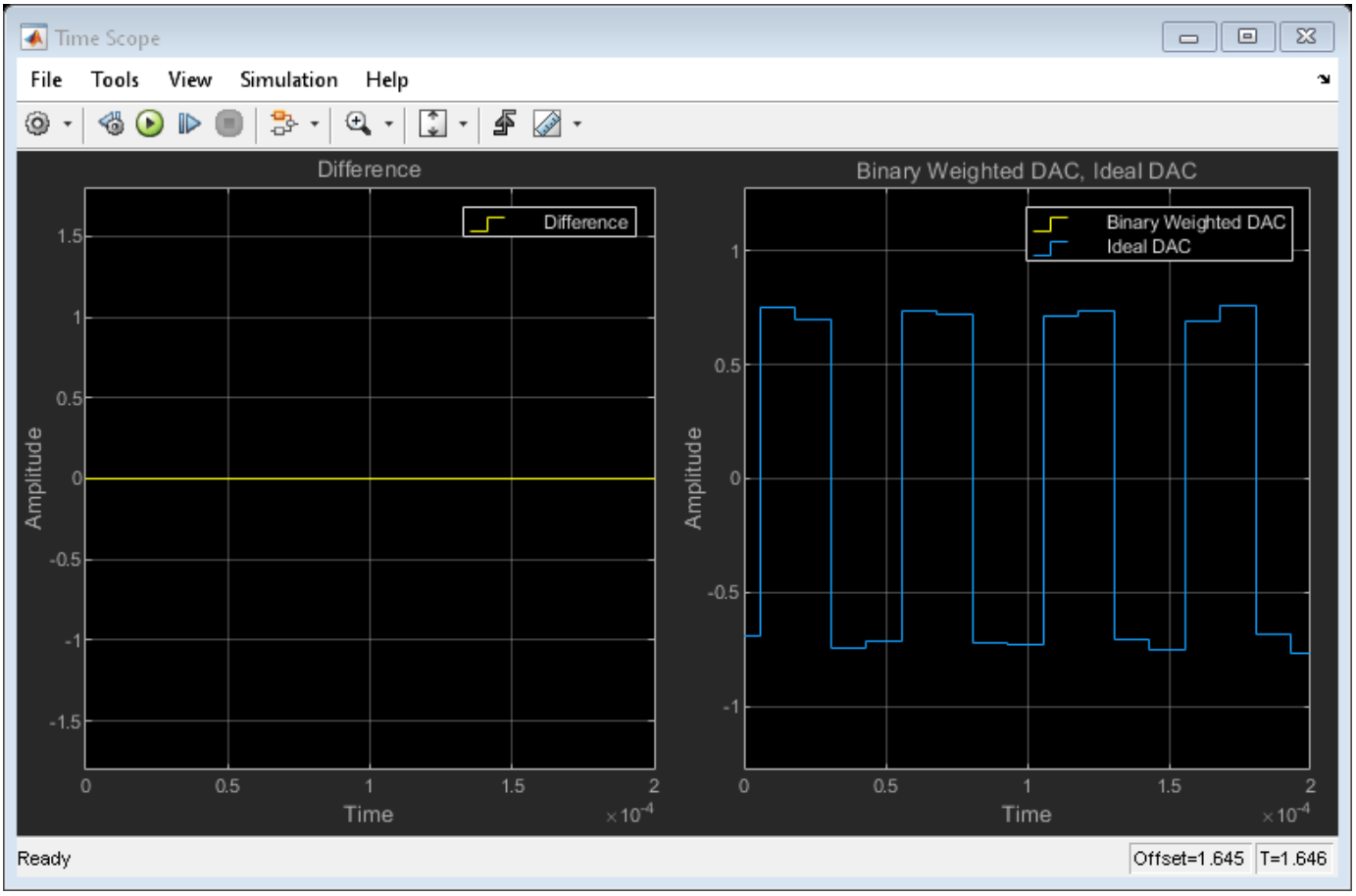
The Binary Weighted DAC outputs a scalar double. The outputs of both the Binary Weighted DAC and the ideal DAC produce values on the interval `[-Ref, Ref - 1 LSB] / 2`.

```
model = 'BinDacIdealCompare';
open_system(model);
```



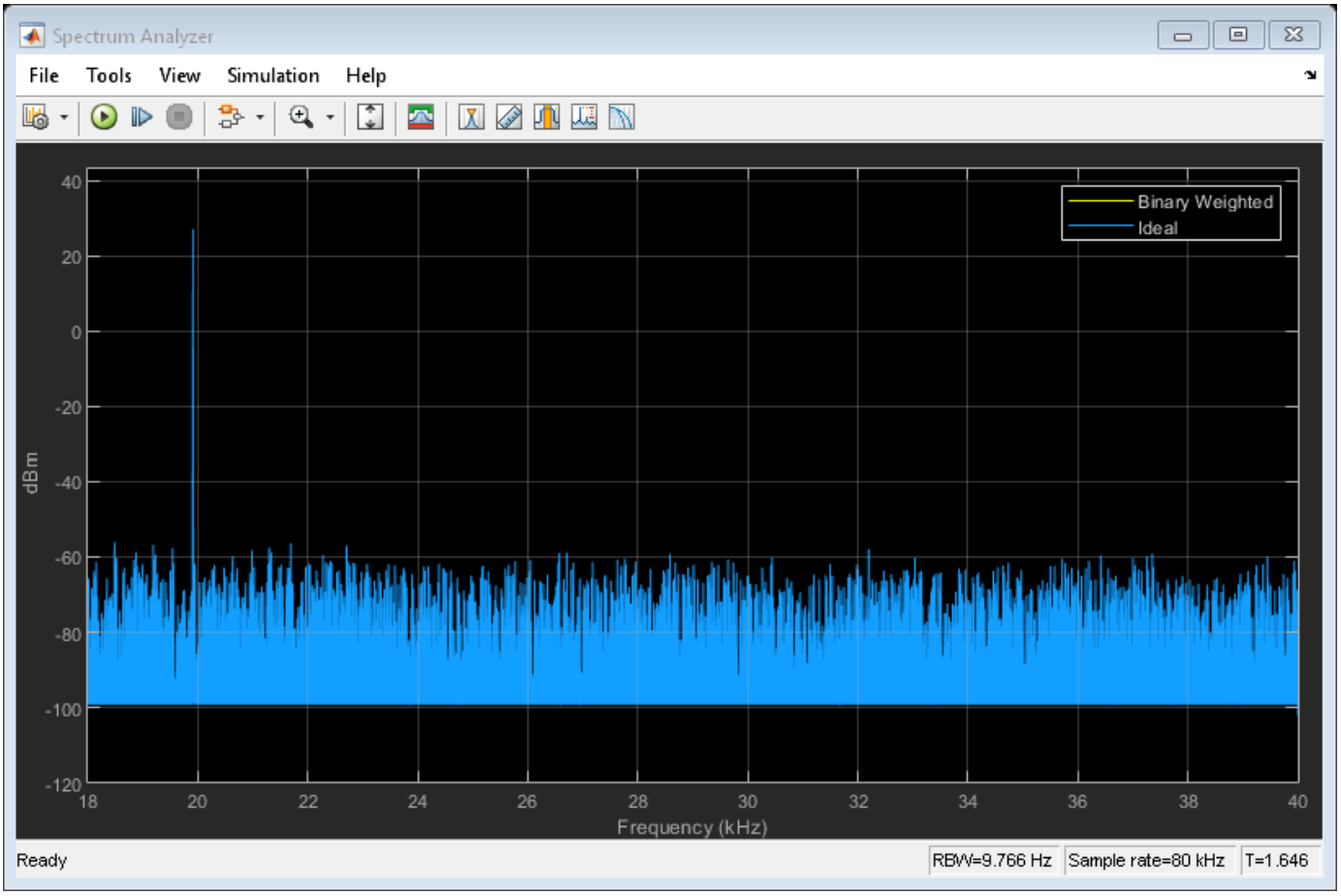
The Scope block shows the difference between the two outputs on the left and the two outputs themselves on the right.

```
open_system([model '/Time Scope']);
sim(model);
```



The Spectrum Analyzer shows the spectra of the two outputs. The Spectrum Analyzer requires fixed-step sampled inputs, which the Zero-Order Hold blocks perform at the same rate that the DAC performs conversions, `Fs`.

```
close_system([model '/Time Scope']);
open_system([model '/Spectrum Analyzer']);
sim(model);
```


Spectrum Analyzer
File
Tools
View
Simulation
Help
Binary Weighted
Ideal
dBm
40
20
0
-20
-40
-60
-80
-100
-120
18
20
22
24
26
28
30
32
34
36
38
40
Frequency (kHz)
Ready
RBW=9.766 Hz
Sample rate=80 kHz
T=1.646

Delta Sigma Modulator Data Converter with Half-Band Filter for Decimation

This example shows how to use the Delta Sigma Modulator (DSM) data converter building block with a downstream decimation filtering scheme to be used for an Analog-to-Digital converter (ADC) application. DSMs use oversampling technique which results in the phenomenon of noise shaping by which the in-band quantization noise is strongly attenuated. In general, the noise transfer function (NTF) is given by

$$NTF = \frac{(1-z^{-1})^N}{D(z)}$$

where N is the order of the DSM and D(z) is the denominator polynomial to be designed. The procedure to systematically design a realizable NTF is explained in [1].

The sampling rate Fs in a DSM is typically much greater than the Nyquist rate. The ratio of Fs to Nyquist rate is called Oversampling Ratio (OSR) given by:

$$OSR = \frac{Fs}{2Fb} >> 1$$

where Fb is the highest frequency content (bandwidth) of the input signal.

DSM output is at the high rate of Fs. Decimation filters are used to reduce the data rate to the Nyquist rate (i.e. 2xFb) and filter the data stream. DSM and the decimation filter together comprise the DSM ADC. This example goes through the case of a DSM using 1-bit quantizer, but it is not uncommon to design DSMs with multi-bit quantizers. In [1], SQNR for a second order single bit modulator is shown empirically to be close to 75dB, resulting in an Effective Number Of Bits (ENOB) of 11 bits as explained in the next section. Theoretical SNR calculation of a DSM has been explained in [2].

Design Flow

A typical design flow for modeling a DSM for a specific application is as follows:

1) Design an NTF to achieve a given system specification. The order N depends on the required SQNR and the denominator polynomial D(z) is designed to increase the modulator stability at high frequencies by reducing the gain (low-pass behavior) at high frequencies. (In this example we try to achieve SQNR of 70dB by choosing a 2nd order DSM with OSR = 64. ENOB for this case is 11.3 bits using the expression given below.

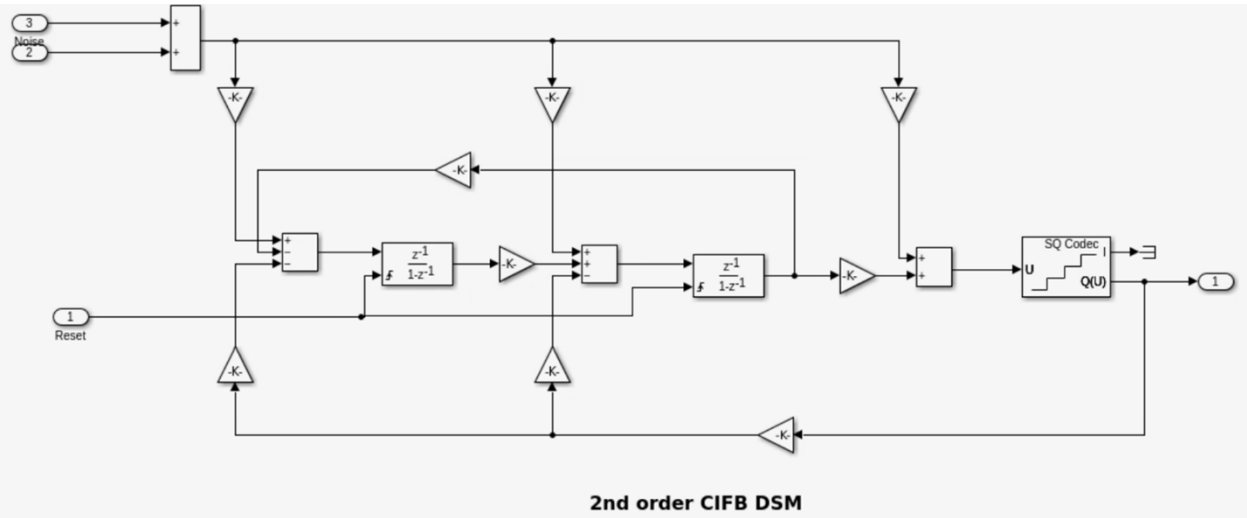
$$ENOB = \frac{(SQNR - 1.76dB)}{6.02dB}$$

2) Choose a DSM structure that provides the best trade-off for the given specifications e.g., quantizer saturation vs. input signal range. A CIFB structure is chosen in this example.

3) Design the decimation filter to be used in conjunction with the DSM to build a DSM ADC. The decimation filter bit-width is chosen to be an integer greater than or equal to ENOB.

4) Run time-domain simulation of the model and check its performance (DSM output spectrum, SNR, etc.)

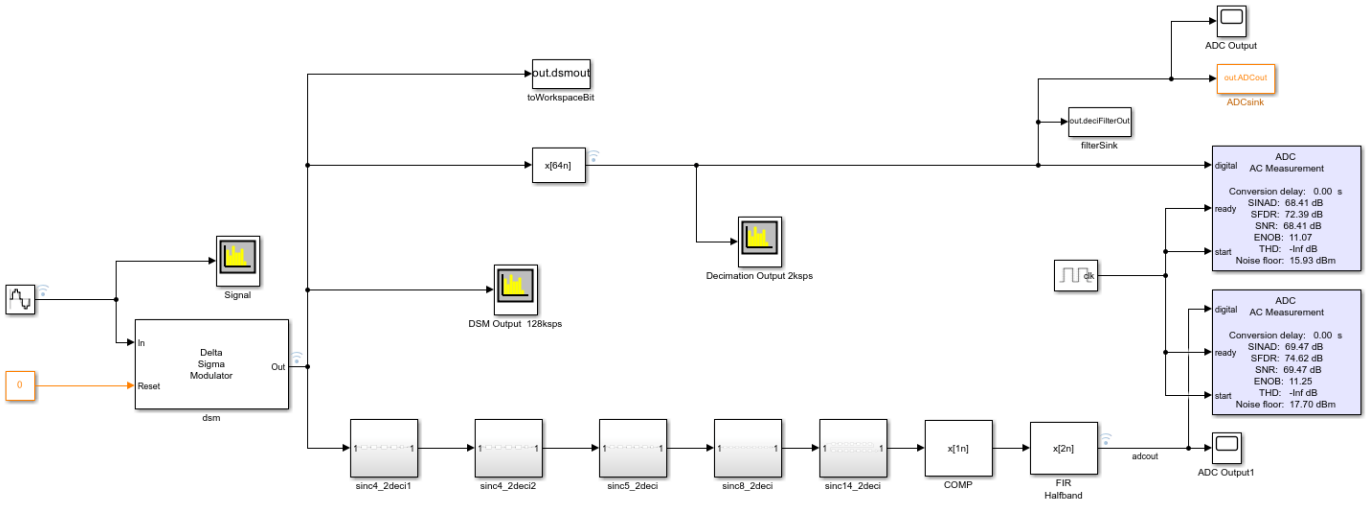
5) Add impairments to see if specifications are still met or if the architecture needs to be modified. Few key impairments for DSM are switched-capacitor kTC noise, switched-capacitor clock jitter, amplifier thermal noise, etc.



Simulate DSM ADC

To simulate the DSM ADC example, open the model "DSM_Decimation_filter" attached to this example:

```
open_system('DSM_Decimation_Filter.slx');
```



An analog input (sine wave of 990Hz) stimulus is over-sampled by the DSM block and is converted into a digital bit-stream. Specification for the DSM block is shown below:

1) Sampling Frequency (Fs) = 128KHz 2) Bandwidth (Fb) = 1KHz 3) Over Sampling Ratio (OSR) = 64

Delta Sigma Modulator

DSM block from the Mixed-Signal Blockset is a masked subsystem containing variant subsystems to define different DSM architectures and orders. Using the block you can select out of 4 different

architectures with orders ranging from 2 to 6. Hence, a total of 20 different architectures can be modeled. DSM architectures can be classified as:

1) CIFB: Cascade of Integrators, Feedback

2) CRFB: Cascade of Resonators, Feedback

3) CIFF: Cascade of Integrators, Feedforward

4) CRFF: Cascade of Resonators, Feedforward

The coefficients 'a', 'g', 'b' and 'c' for the generalized DSM structures can be determined using the procedure explained in chapter 4 in [1]. The gain coefficients in this example were obtained using the functions provided in the Delta Sigma Toolbox by Richard Schreier [3].

Decimation Filter

A decimation filter serves two purposes: 1) Filter the out-of-band noise 2) "Decimate" the DSM output data rate from Fs to the Nyquist rate 2xFb = (Fs / OSR).

For efficiency of implementation, a decimation filter is implemented as a cascade of digital filters instead of a single stage. In this example, the decimation filter is designed to have an alias attenuation of > 100 dB and < 0.1 dB of passband variation. Since the OSR of DSM is 64, a decimation factor of 64 is implemented. The filter comprises a cascade of 5 sinc filters each with a decimation factor of 2, a compensation FIR filter to compensate for droop in the sinc filters, followed by a half-band filter. The order and gain of the sinc filters are mentioned in Table 14.2 in [1]. See chapter 14 in [1] for a detailed discussion of Decimation Filter design.

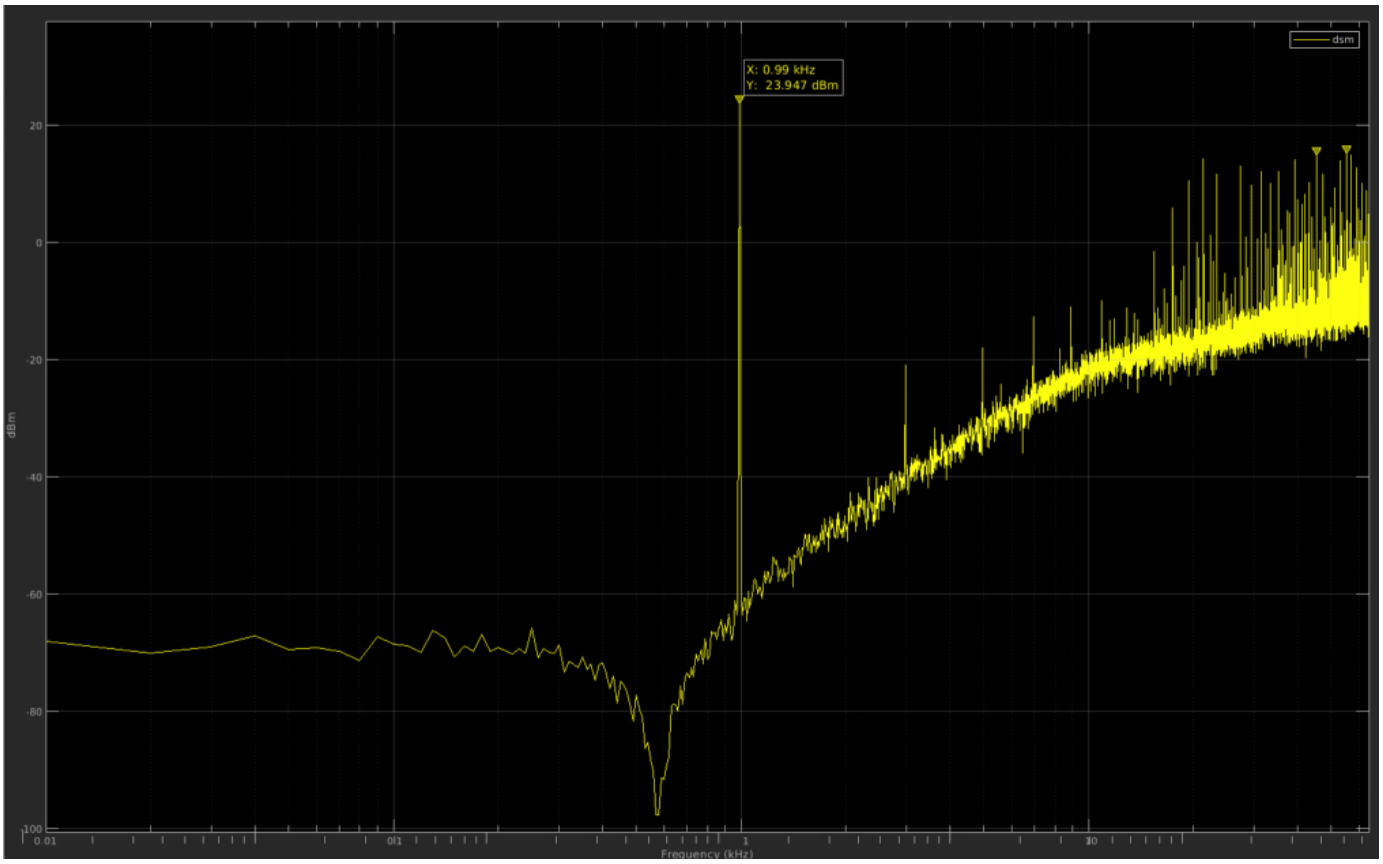
Half-band filter is a low pass filter which reduces the maximum bandwidth of sampled data by one octave. In this example, half-band filter has been implemented using FIR Decimation block with the filter coefficients obtained by using the DSP System Toolbox function 'firhalfband' as shown below:

firhalfband('minorder', 0.45, 1E-05);

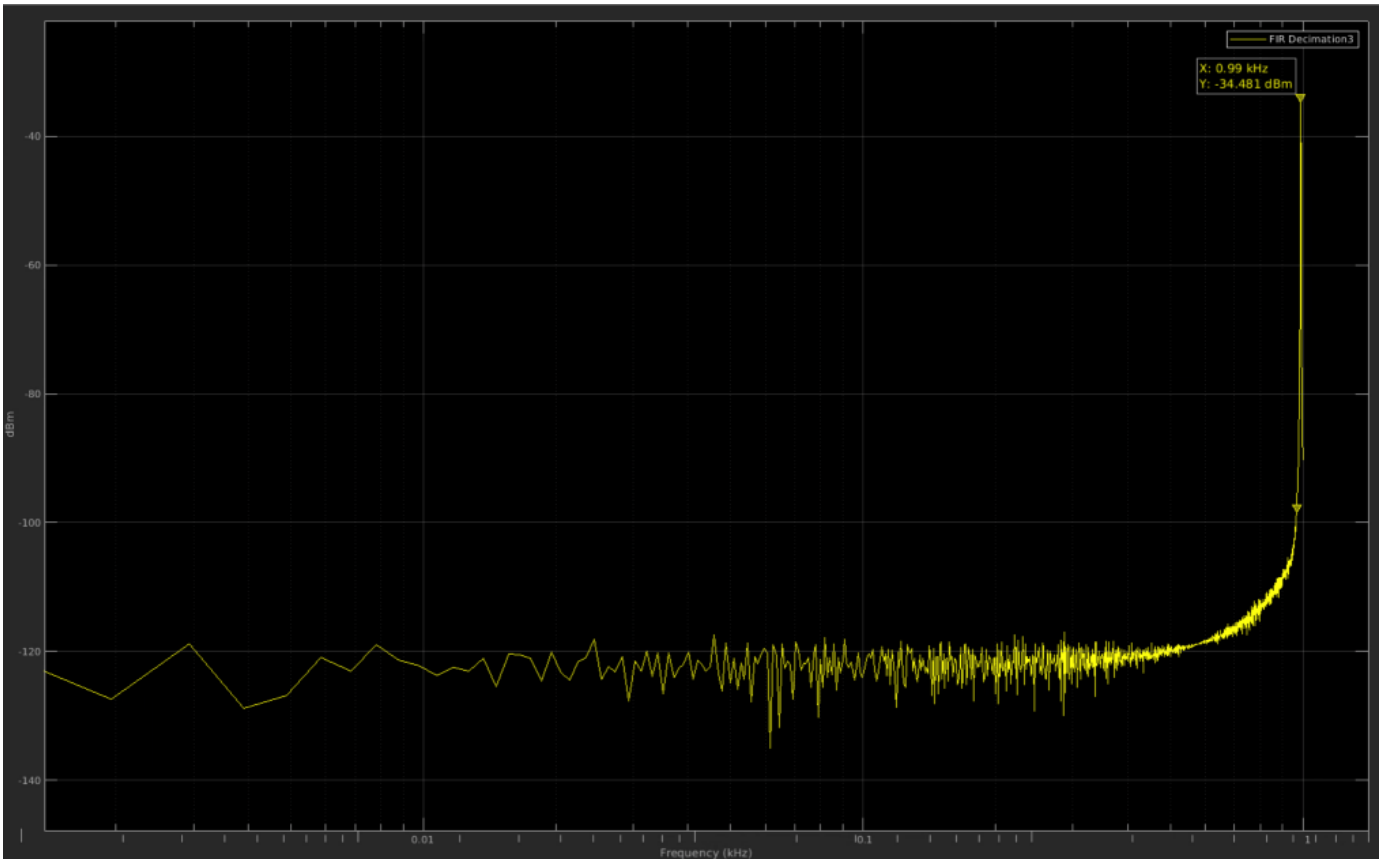
Alternatively, decimation filter can also be implemented using an FIR Decimation block (from DSP System Toolbox) with the decimation option set to 64 in the mask. More information on FIR Decimation block can be found on the documentation page for `dsp.FIRDecimator`.

Simulation Results

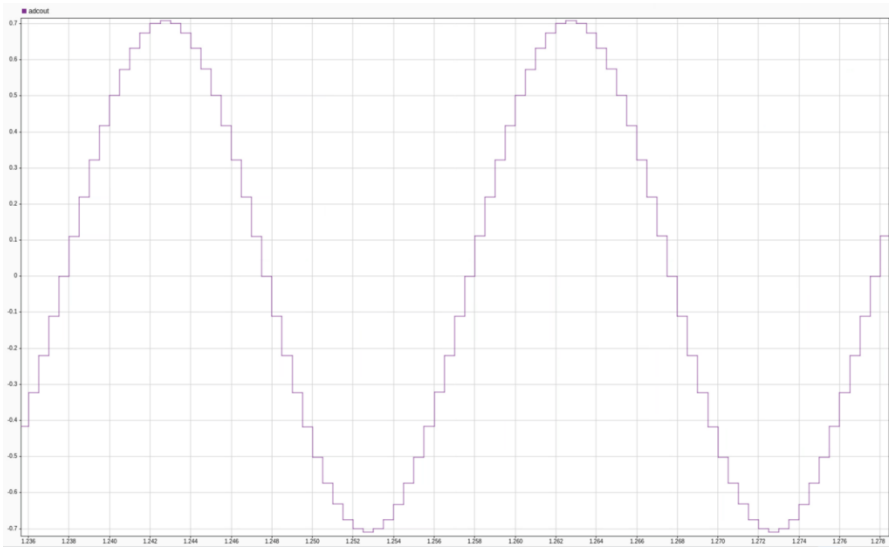
The ADC model with 2nd order CIFB DSM block was simulated with a sinusoidal signal at 990Hz (very close to Fb). DSM output (128Ksps) is fed to a Spectrum analyzer to analyze power spectrum of the ADC output and the spectrum shows a spike at 990Hz as expected.



The input to the decimation filter gets decimated by a factor of 64. The data rate is equal to 2Ksps, since OSR = 64. Spectrum of the decimation filter output (2ksps) is shown below with a spike at 990Hz.



Additional simulation was run with an input frequency of 50Hz and the DSM ADC output for this sinusoidal signal is captured.



The ADC output was fed to an "ADC AC measurement" block to check the performance of the DSM ADC. Both filter outputs ("FIR Decimation" and "Cascade of sinc filters") show an ENOB = 11 bits as expected.

References

[1] Shanthi Pavan; Richard Schreier; Gabor C. Temes, Understanding Delta-Sigma Data Converters, second edition, IEEE Press, copyright 2017.

[2] Valeri Mladenov; Panagiotis Karampelas; Georgi Tsenov and Vassiliki Vita, Approximation Formula for Easy Calculation of Signal-to-Noise Ratio of Sigma-Delta Modulators

[3] Richard Schreier (2022). Delta Sigma Toolbox (https://www.mathworks.com/matlabcentral/fileexchange/19-delta-sigma-toolbox), MATLAB Central File Exchange. Retrieved June 14, 2022.

Mixing Analog and Digital Signals Featured Examples

Digital Timing Using Solutions to Ordinary Differential Equations

This example shows how to model a three stage ring oscillator using models defined by ordinary differential equations (ODE).

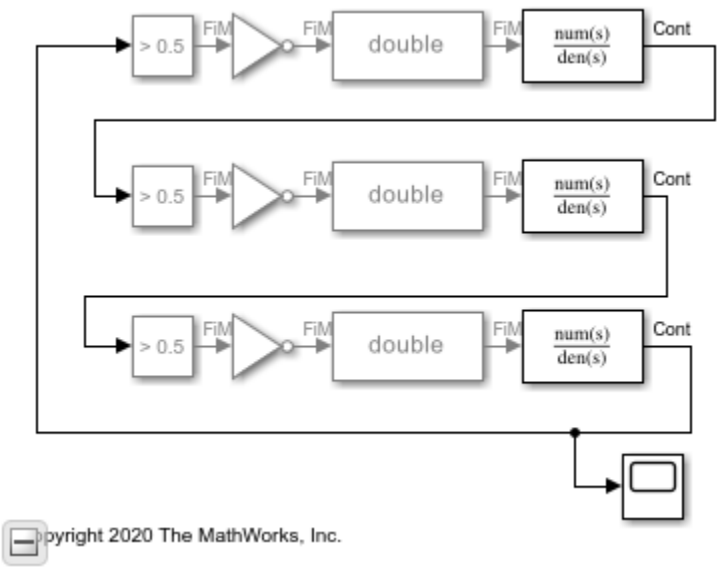
This example is the third of three examples that use a three stage ring oscillator model to explore the range of options for simulating the analog applications of digital circuits. The delays in each stage determine the ring oscillator's output frequency, making the accurate modeling of these delays essential to the simulation of the circuit.

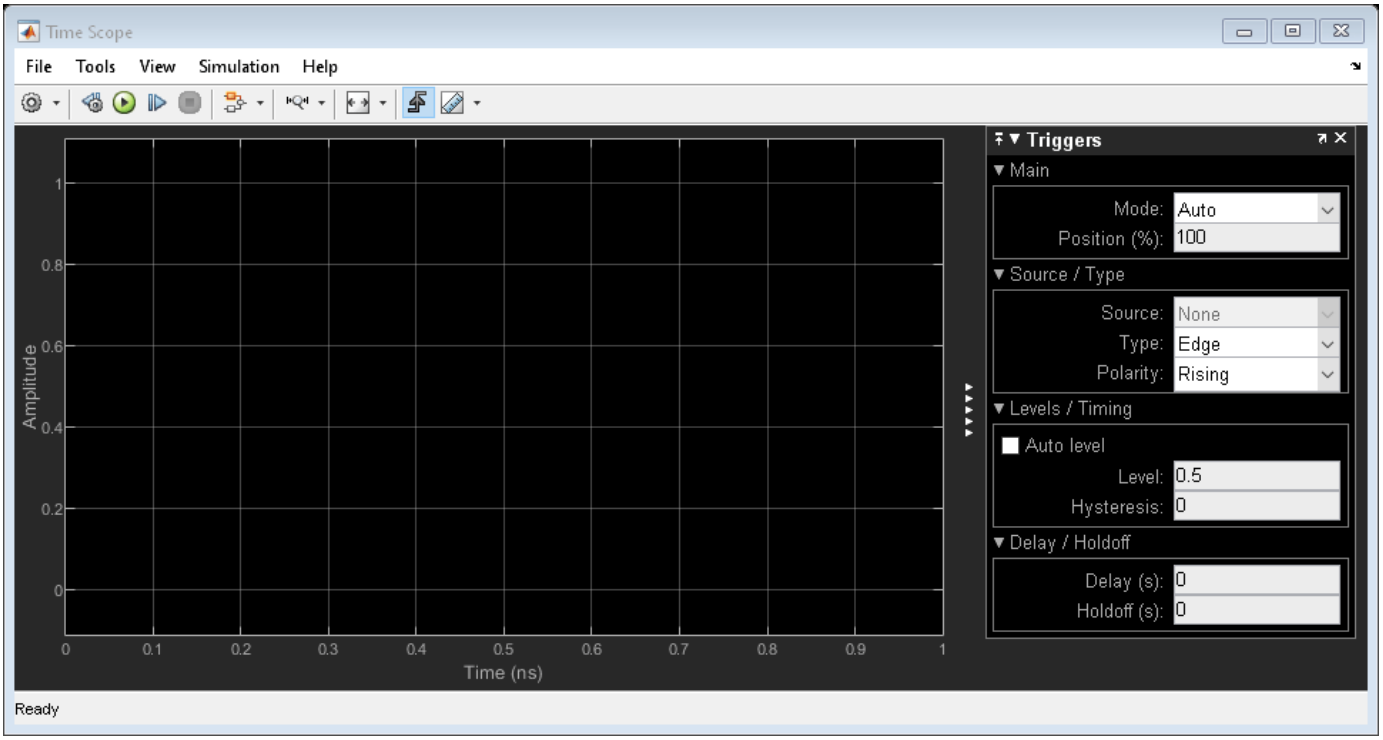
The first two examples in the sequence, “Logic Timing Simulation” on page 3-12 and “Digital Timing Using Fixed Step Sampling” on page 3-7, contain background information about this example. Read them first, in sequence, if you have not already done so.

This model uses blocks defined by ODEs, and depends on the services of an ODE solver. For each stage, the zero crossing detection capabilities of a Compare To Constant block are used to produce a saturated input to the inverter. The inverter output is converted from Boolean to double to drive a Transfer Function block. The Transfer Function block defines the shape of the inverter output transitions.

Load the ODE-based model and update the model to display sample times.

```
open_system('OdeWaveform');
set_param(gcs,'SimulationCommand','update');
```





Continuous Time Model Using Exponential Decay

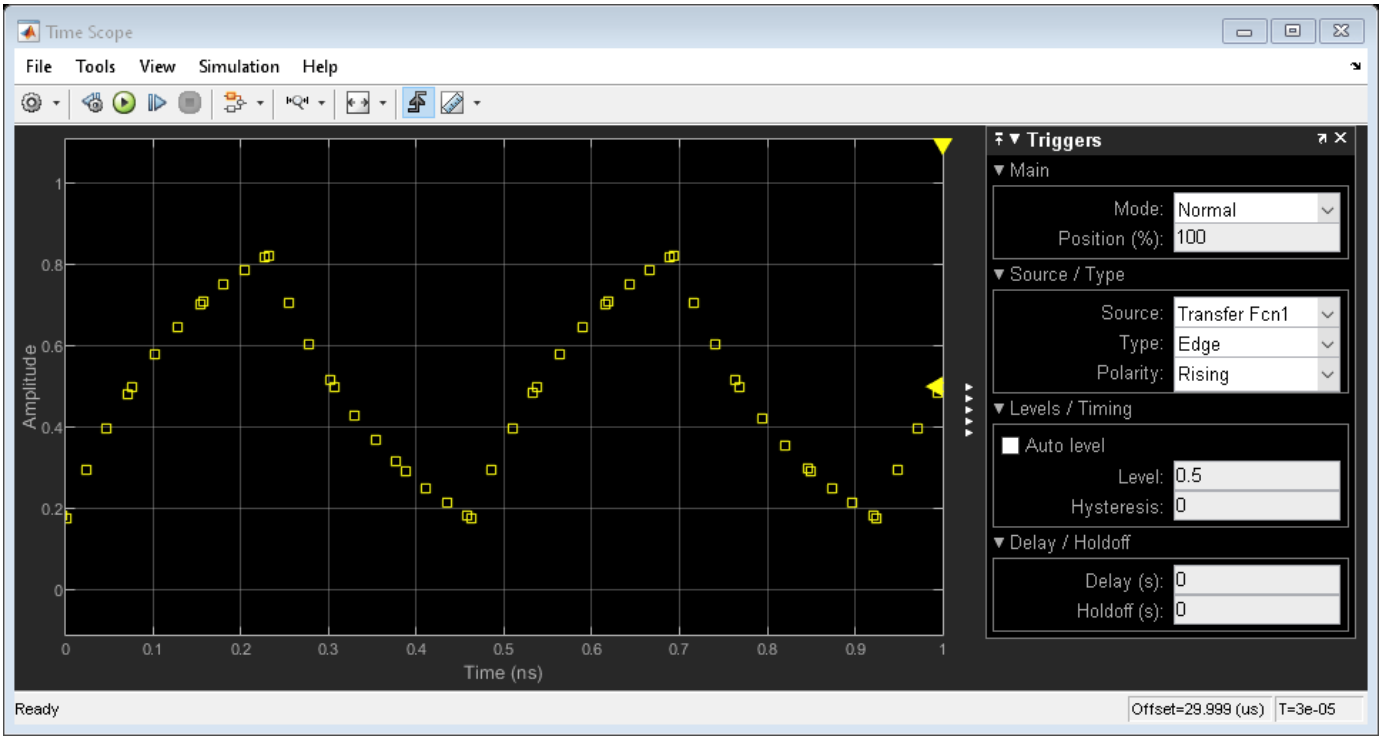
In this section, use a single pole response to evaluate the ring oscillator output when the inverter output is modeled as the response of an RC circuit.

For this section, the Transfer Function blocks are configured for a single pole response. The pole for one of the logic stages is set to a slightly different value than for the other two stages so that the model will enter the correct mode of oscillation.

The solver selection is set to `auto`, with a **Relative Tolerance** of `1e-9`.

Run the ODE-based model with the single pole rise/fall response.

```
sim('OdeWaveform');
```



Continuous Time Model with Nearly Constant Slew Rate

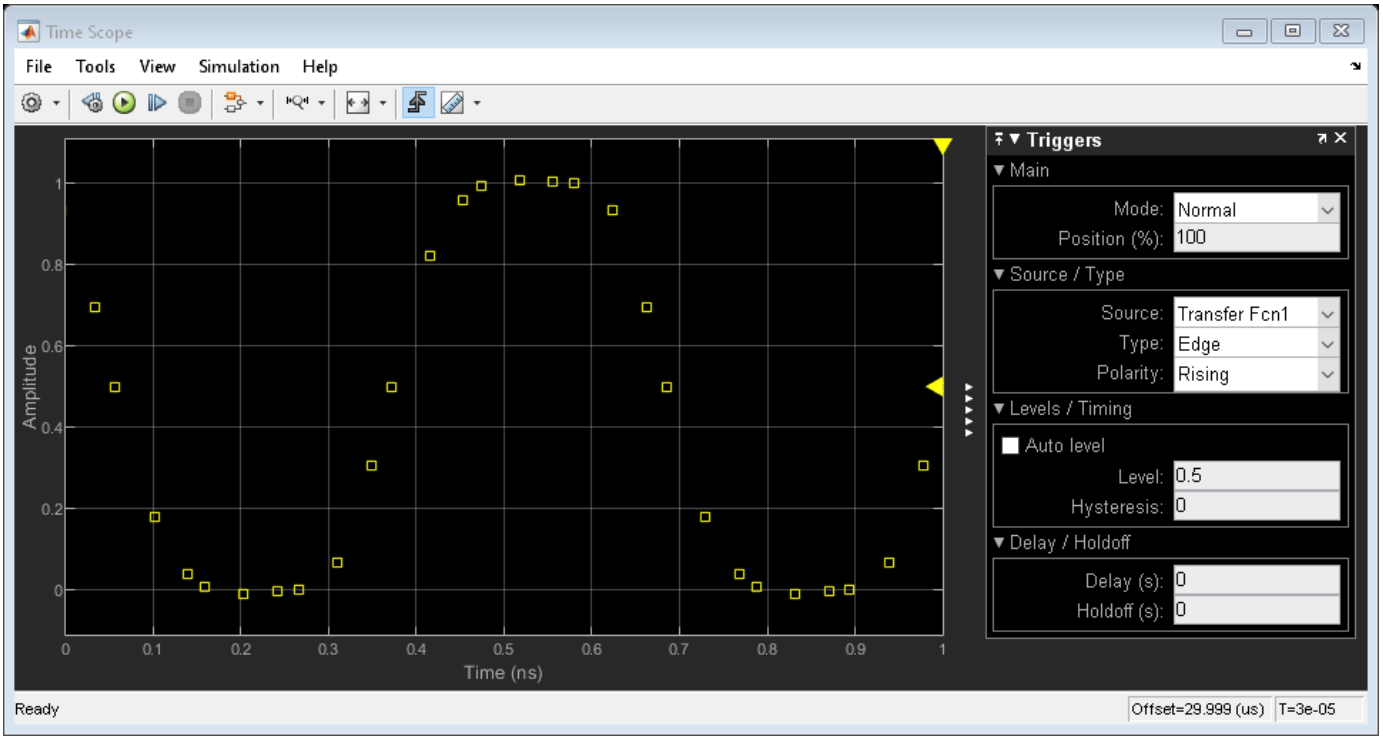
In this section, model the response of the ring oscillator stages using a continuous time Transfer Function block with a fourth order Bessel-Thompson response to approximate a constant slew rate response.

Set the configuration for the fourth order Bessel-Thompson rise/fall response.

```
den = getBesselDenominator(3e9);
set_param('OdeWaveform/Transfer Fcn','Denominator',mat2str(den));
set_param('OdeWaveform/Transfer Fcn1','Denominator',mat2str(den));
den = getBesselDenominator(3.1e9);
set_param('OdeWaveform/Transfer Fcn2','Denominator',mat2str(den));
```

Run the modified ODE-based model. Note the slight rounding of the onset of the switching edges, similar to the waveforms produced in the “Digital Timing Using Fixed Step Sampling” on page 3-7 example.

```
sim('OdeWaveform');
```



Continuous Time with Solver in Auto Mode

In this section, change the solver configuration and observe the change in results.

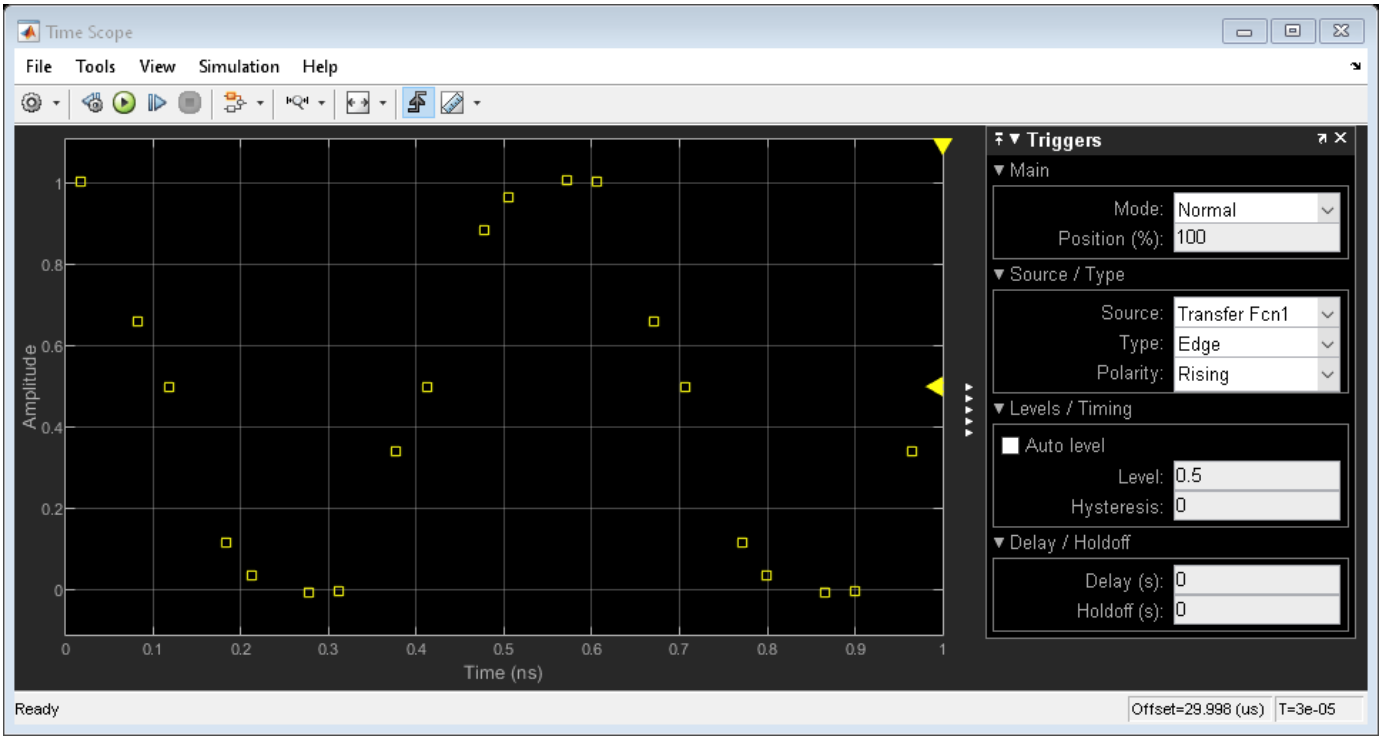
For circuits which are adequately described by linear, time invariant models, the combination of fixed step and variable step discrete sample times, as described in the **Combined Fixed Step and Digital Timing** section may be the simplest way to get reliable results. However, for circuits which must be modeled by a nonlinear or time-varying model, the ODE-based solution is the only viable option. In such cases, you should vary the maximum error tolerance, maximum step size or choice of solver in the solver configuration dialog and compare the results to the behavior you expect.

Maintain the model configuration of the previous section but change the solver's **Relative Tolerance** from `1e-9` to `auto`.

```
set_param('OdeWaveform','RelTol','auto');
```

Run the model with the auto solver setting. Observe both the change in period of oscillation and in wave shape.

```
sim('OdeWaveform');
```



See Also

More About

- “Logic Timing Simulation” on page 3-12
- “Digital Timing Using Fixed Step Sampling” on page 3-7

Digital Timing Using Fixed Step Sampling

This example shows how to model a three stage ring oscillator using a combination of fixed step and variable step discrete sample times.

This example is the second of three examples that uses a three stage ring oscillator model to explore the range of options for simulating the analog applications of digital circuits. The delays in each stage determine the ring oscillator's output frequency, making the accurate modeling of these delays essential to the simulation of the circuit.

The first example in the sequence, “Logic Timing Simulation” on page 3-12 contains background information about this example. First read that example if you have not already done so. After completing this example, you can move forward with the “Digital Timing Using Solutions to Ordinary Differential Equations” on page 3-2 example.

For each stage of the ring oscillator, the Logic Decision block converts the input into a saturated variable step discrete signal and the Slew Rate block converts the output to an analog fixed step discrete signal. Some delay in the Logic Decision block is unavoidable; however most of the delay is introduced by the Slew Rate block.

The Logic Decision block generates a variable step discrete sample at its output in response to any threshold crossing it detects at its input.

For a fixed step discrete input sample time, the threshold crossing time is determined by linear interpolation between the two most recent samples. The output sample is delayed by one sample because the block does not have access to the fixed in minor step services of an ODE solver. In the modeling of a circuit, this delay must represent either the delay of input stages in a multi-stage transistor circuit or RC transmission line routing delay.

For a fixed step input, the precision of the threshold crossing time reported by the Logic Decision block depends on the ratio of the spectral content of the signal to the Nyquist frequency defined by sample rate. For a sine wave at 0.25 times the Nyquist frequency (8x oversampling), the maximum error in the reported threshold crossing time is 1% of a sample interval. For 0.1 times the Nyquist frequency, the maximum error is 0.15% of a sample interval. For applications requiring greater precision, such as evaluating low level phase noise at the output of a PLL, an approach that depends only on variable step sampling may produce more precise results.

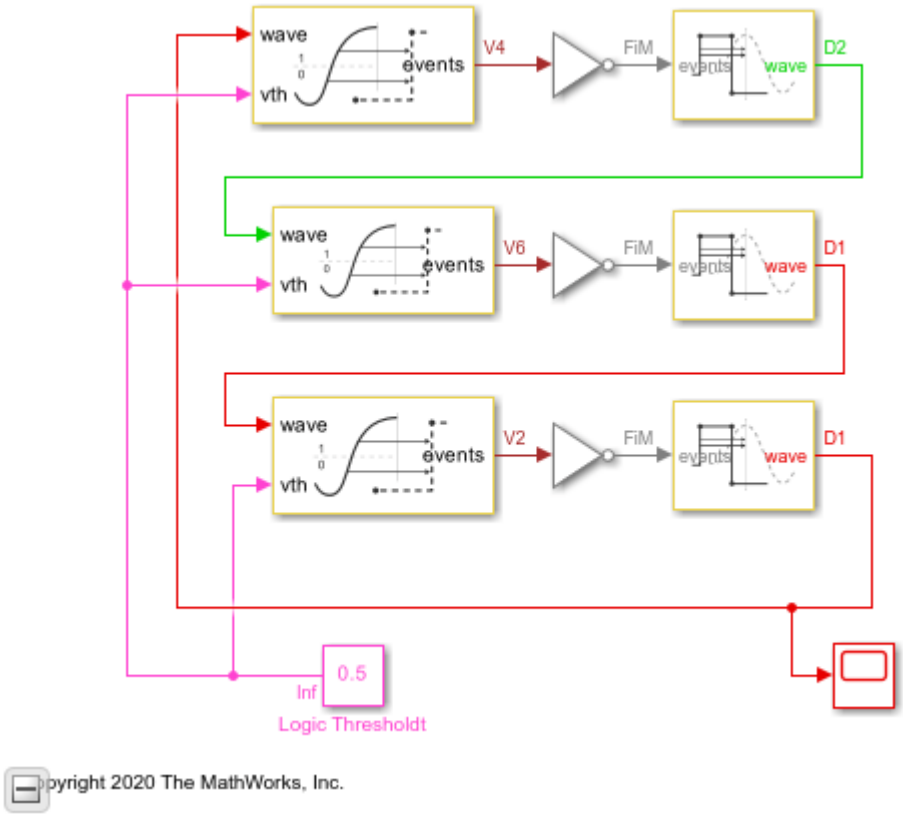
For a variable step input, the output of the Logic Decision block is delayed by the minimum delay parameter for the block.

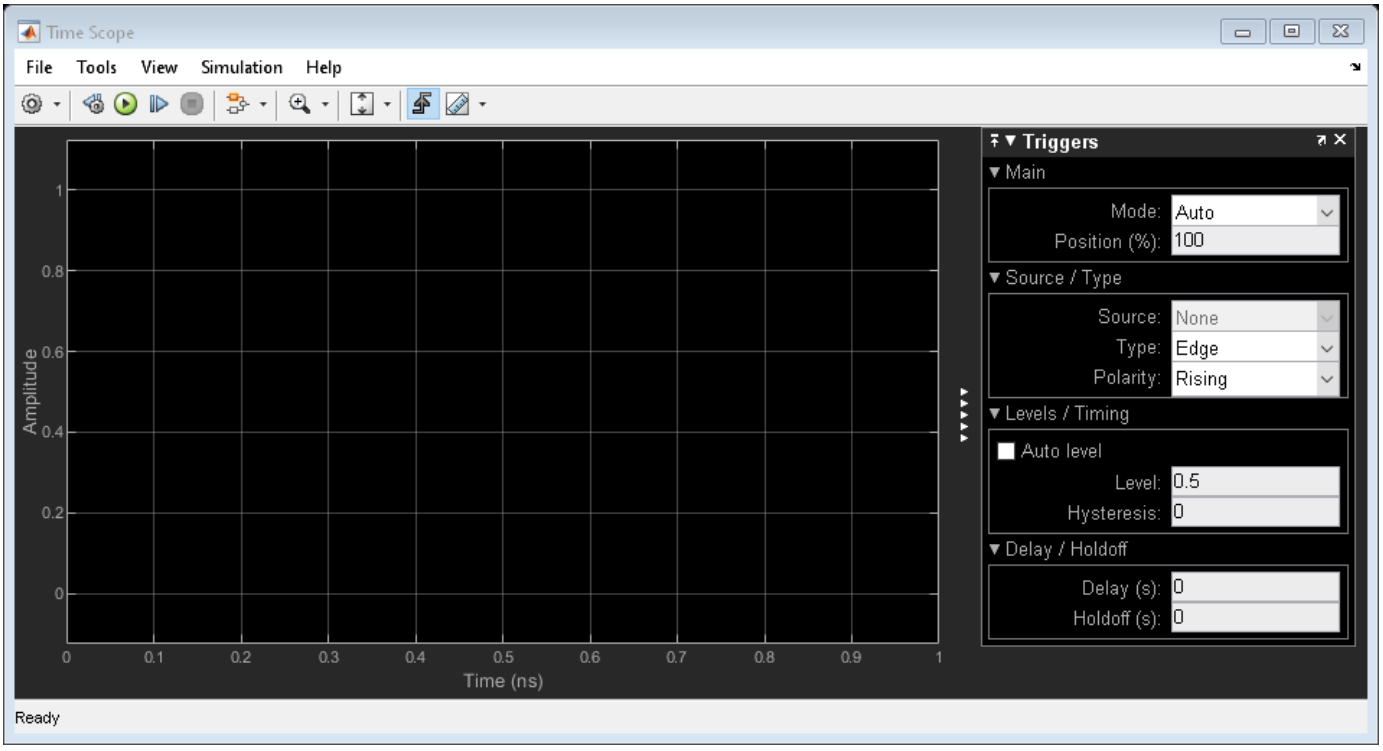
The Slew Rate block implements a linear time invariant transfer function that can be applied to either a variable step or fixed step input signal, producing a fixed step discrete output signal with a sample time that was set by the Slew Rate block. The delay of the Slew Rate block is a mixture of:

- Constant delay such as might occur in a multi-stage transistor circuit or RC routing delay
- Nearly constant slew rate such as would be typical of a saturated transistor driving a capacitive load
- Exponential decay such as would be typical of an RC circuit

Load the mixed analog/digital model and update the model to display sample times.

```
open_system('AnalogWaveform');
set_param(gcs,'SimulationCommand','update');
```





Slew Rate Block in Default Sampling Mode

In this section, use the **Default** sampling mode of the Slew Rate block to model the response of a circuit whose slew rate is nearly constant, such a saturated transistor driving a capacitive load.

For this section, the Slew Rate blocks are configured to their **Default** sampling mode, which maximizes the portion of the delay due to nearly constant slew rate (saturated transistor) and minimizes the delay due to constant delay or exponential decay.

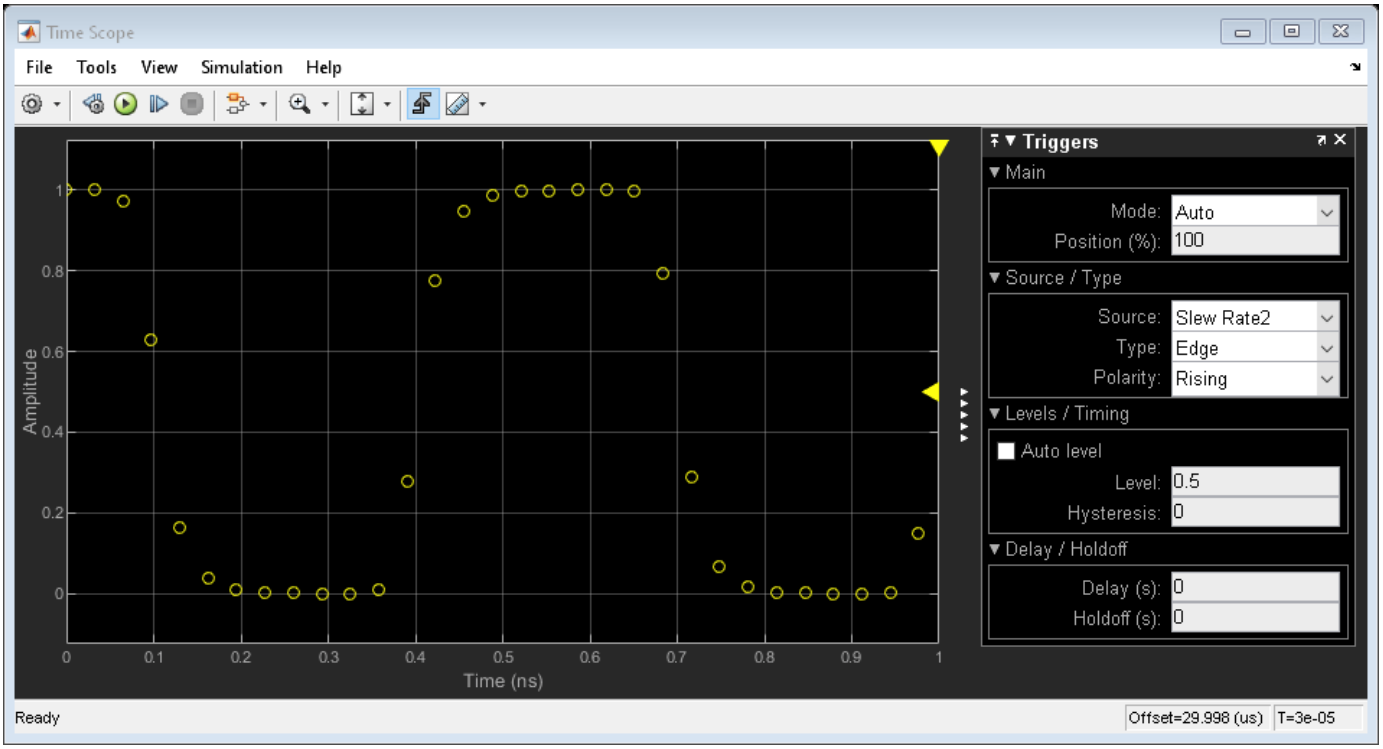
The delay for one logic stage is set to a slightly different value than for the other stages so that the model enters the correct mode of oscillation.

Since the model contains no differential equations, the solver is Variable Step Discrete.

Note in the response that the onset of the switching edges is slightly rounded. In a real circuit, this would typically be due to the RC response of the routing.

Run the mixed analog/digital model with default sample times.

```
sim('AnalogWaveform');
```



Slew Rate Block in Advanced Sampling Mode

In this section, use the **Advanced** sampling mode of the Slew Rate block to model circuits whose response is primarily a decaying exponential.

Choose the **Advanced** mode for the Slew Rate block sampling and set the **Maximum frequency of interest** to a value that is high enough to make the delay of the Slew Rate block due primarily to exponential decay. This choice also minimizes the delay of the Logic Decision block.

Note in the response that the onset of the switching edges is relatively sharp.

Configure the mixed analog/digital model to approximate single pole response.

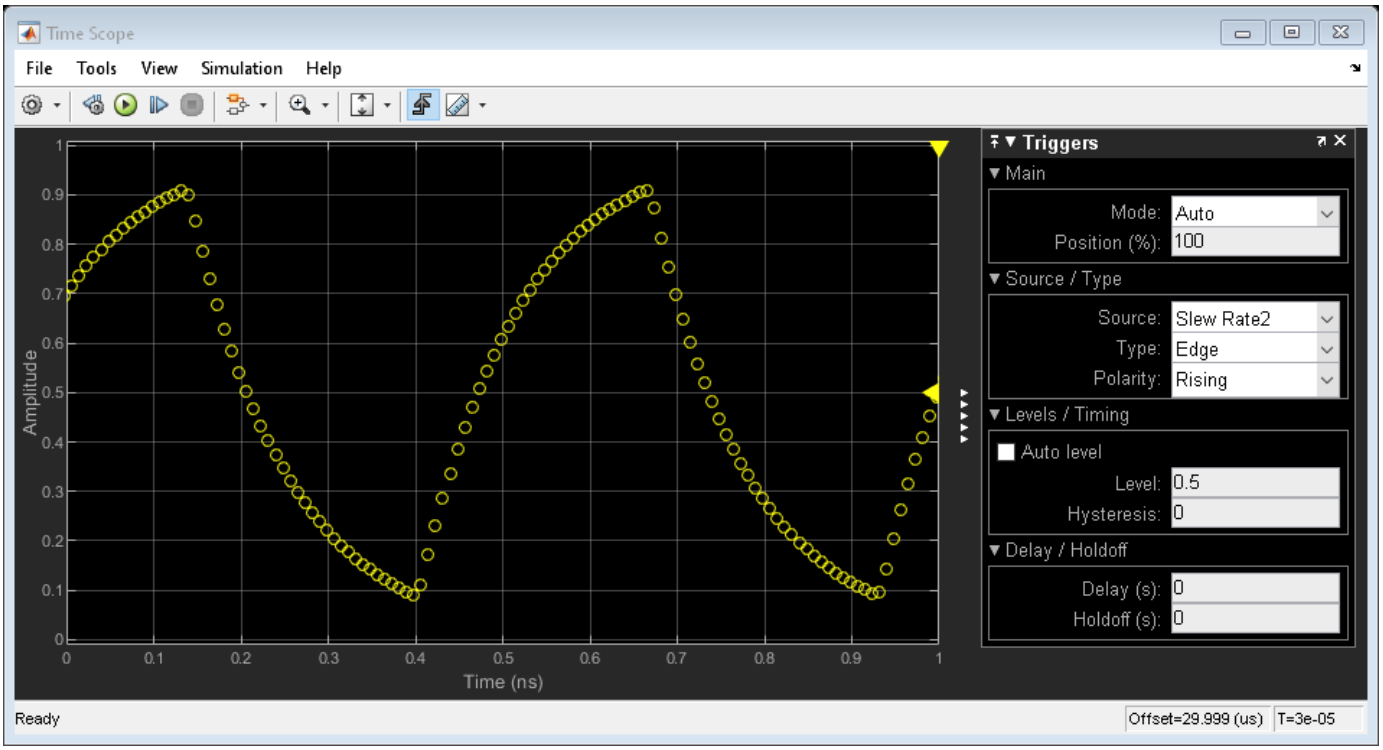
```
% Slew Rate1
set_param('AnalogWaveform/Slew Rate1','DefaultOrAdvanced','Advanced');
set_param('AnalogWaveform/Slew Rate1','MaxFreqInterest','20e9');
set_param('AnalogWaveform/Slew Rate1','RisePropDelay','92e-12');
set_param('AnalogWaveform/Slew Rate1','RiseTime','15.5e-11');
% Slew Rate2
set_param('AnalogWaveform/Slew Rate2','DefaultOrAdvanced','Advanced');
set_param('AnalogWaveform/Slew Rate2','MaxFreqInterest','20e9');
set_param('AnalogWaveform/Slew Rate2','RisePropDelay','92e-12');
set_param('AnalogWaveform/Slew Rate2','RiseTime','15.5e-11');
% Slew Rate3
set_param('AnalogWaveform/Slew Rate3','DefaultOrAdvanced','Advanced');
set_param('AnalogWaveform/Slew Rate3','MaxFreqInterest','20e9');
set_param('AnalogWaveform/Slew Rate3','RisePropDelay','95e-12');
set_param('AnalogWaveform/Slew Rate3','RiseTime','16e-11');
```

Update the diagram to show the revised sample times in the sample time legend.

```
set_param(gcs,'SimulationCommand','update');
```

Run he mixed analog/digital model with emphasized one pole response.

```
sim('AnalogWaveform');
```



See Also

Logic Decision | Slew Rate

More About

- “Logic Timing Simulation” on page 3-12
- “Digital Timing Using Solutions to Ordinary Differential Equations” on page 3-2

Logic Timing Simulation

This example shows how to use the Variable Pulse Delay block to create accurate timing models of logic circuits.

This example is the first of three examples that use a three stage ring oscillator model to explore the range of options for simulating the analog applications of digital circuits. The delays in each stage determine the ring oscillator's output frequency, making the accurate modeling of these delays essential to the simulation of the circuit. This first example shows how to produce digital waveforms with accurate timing by using variable step discrete sampling.

The second and third examples both show how to produce analog waveforms with accurate shape and timing. The “Digital Timing Using Fixed Step Sampling” on page 3-7 example shows how to model the three stage ring oscillator using a combination of fixed step and variable step discrete sample times. The “Digital Timing Using Solutions to Ordinary Differential Equations” on page 3-2 shows modelling the three stage ring oscillator using models defined by ordinary differential equations (ODE). You should study this example before studying the other two.

The delays in this model are introduced by the Variable Pulse Delay blocks from the Utilities library of the Mixed-Signal Blockset™, with the delay defined by a separate input to the block. The initial output values for the Variable Pulse Delay blocks are set to guarantee oscillation. The initial output values for two of the blocks are kept at the default value of zero while the initial output for the third block is set to one.

The oscilloscope is configured to display the samples as a scatter graph, with no rendering between samples. Different sample times make different assumptions about the signal value between samples such as:

- Zero Order Hold (ZOH) — The signal value is assumed to equal the value of the most recent sample.
- First Order Hold (FOH) — The signal value is assumed to vary linearly from one sample to the next.
- Nyquist limited — The signal is assumed to have zero spectral content above a frequency equal to one half of a fixed sample rate.
- Taylor series — For each major sample step, an ODE solver produces a polynomial that approximates the signal value over that time interval.

The oscilloscope block bases its rendering on these assumptions. You must focus on the samples themselves and understand explicitly the assumptions that different sample times make.

The samples displayed on the oscilloscope show a single sample for each logic switching event. These samples are generated by the Variable Pulse Delay blocks. Every time a Variable Pulse Delay block receives a sample, it generates a new event at a time equal to the sample time plus the value at the **delay** input port.

As indicated by the sample time color coding, the output sample time for the inverters is Fixed In Minor Step (FIM). This means that each inverter produces an output sample value for every major sample time in the model, regardless whether or not that sample time is used at an input port of the gate.

This FIM behavior is typical of most logic blocks; however you should pay special attention to sample time propagation in triggered subsystems such as D flip-flops. If the trigger input uses a fixed step

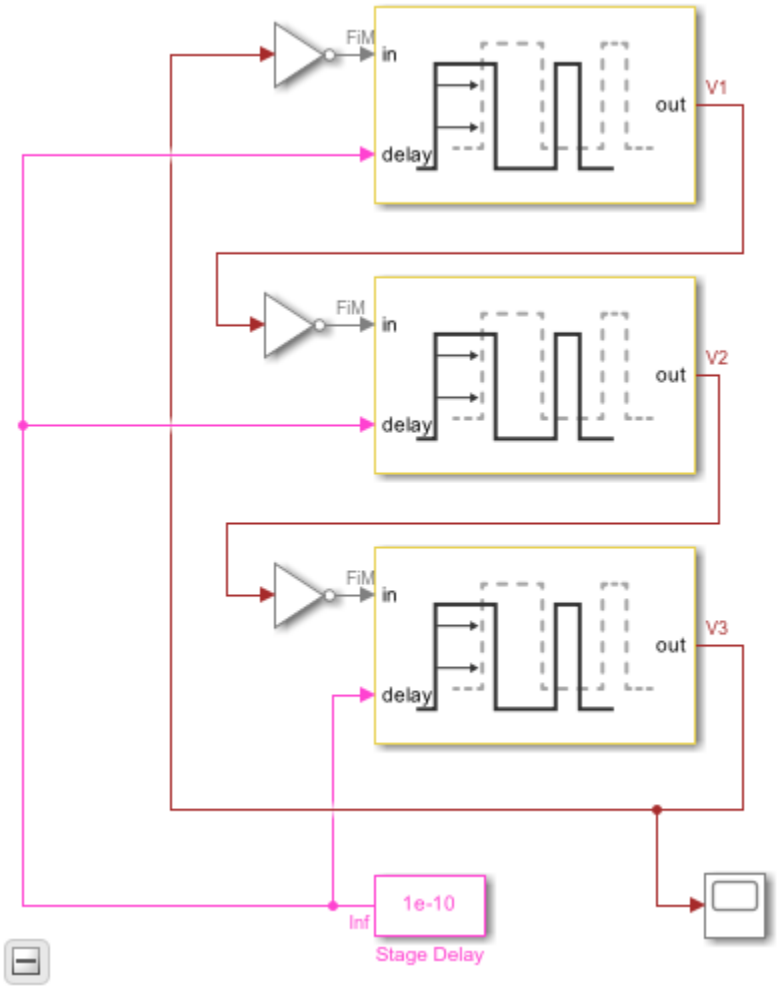
discrete sample time, then any input which is not synchronous with that sample time may not be processed correctly. The triggered subsystem can be forced to operate in FIM mode by triggering it with a variable step discrete trigger such as would be produced by the Variable Pulse Delay block or the Logic Decision.

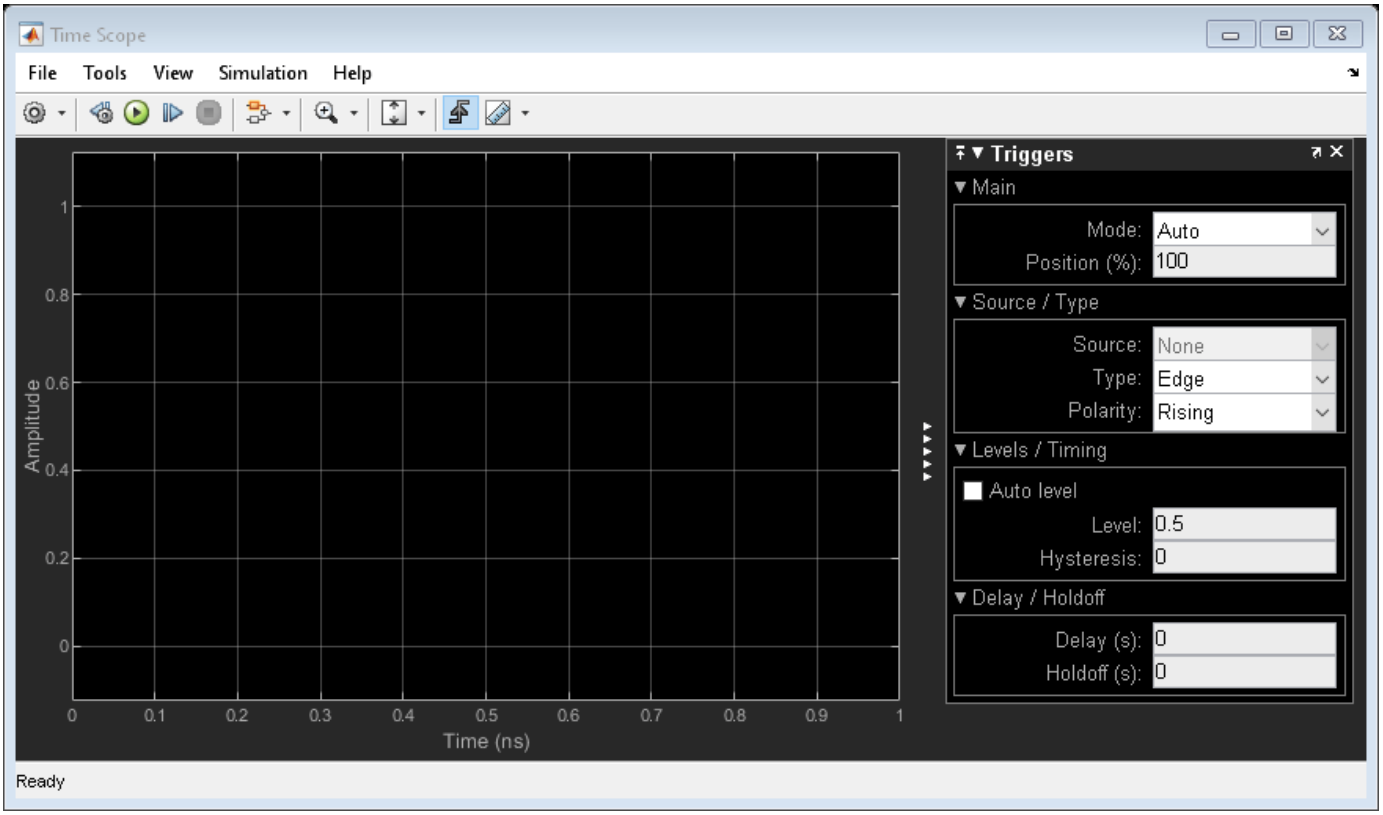
Since the model does not contain any differential equations, the solver is **Variable Step Discrete**.

The **Stage Delay** is set to `100` ps, resulting in a half period of precisely `300` ps and a period of `600` ps, as demonstrated in the simulation output.

Load the logic timing model and update the model to display sample times.

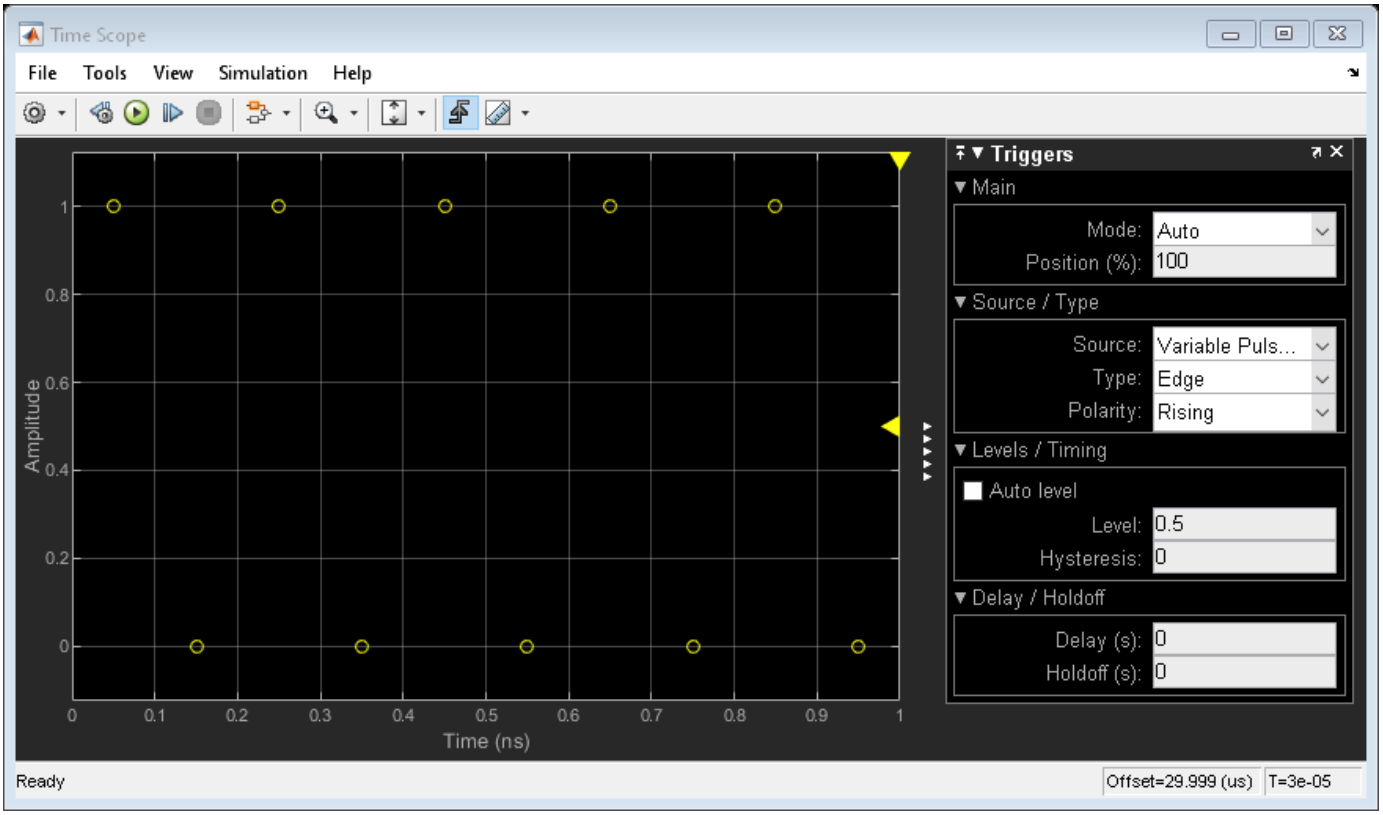
```
open_system('LogicTiming');
set_param(gcs,'SimulationCommand','update');
```





Run the logic timing model.

```
%%.
sim('LogicTiming');
```



See Also

Variable Pulse Delay

More About

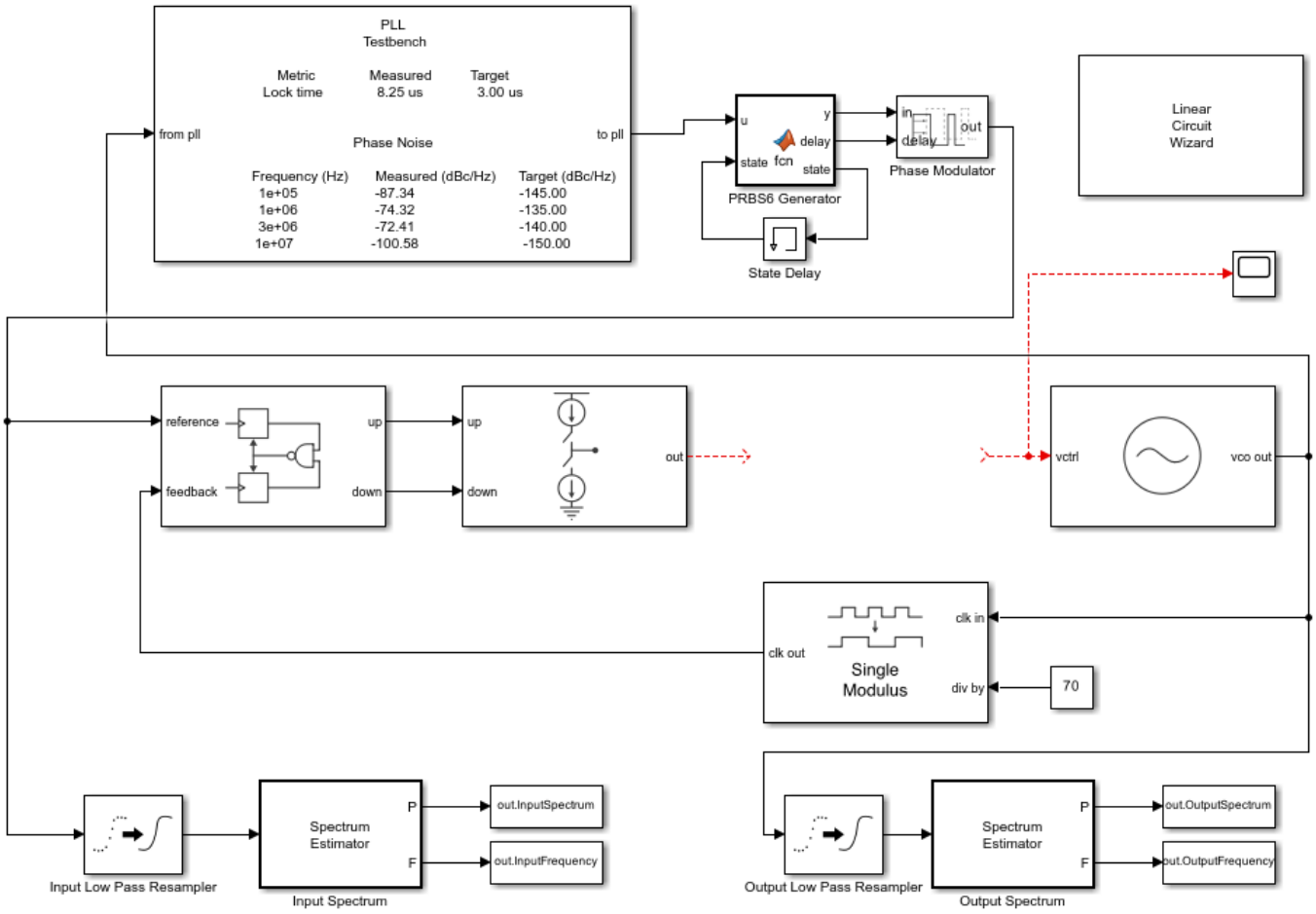
- “Digital Timing Using Fixed Step Sampling” on page 3-7
- “Digital Timing Using Solutions to Ordinary Differential Equations” on page 3-2

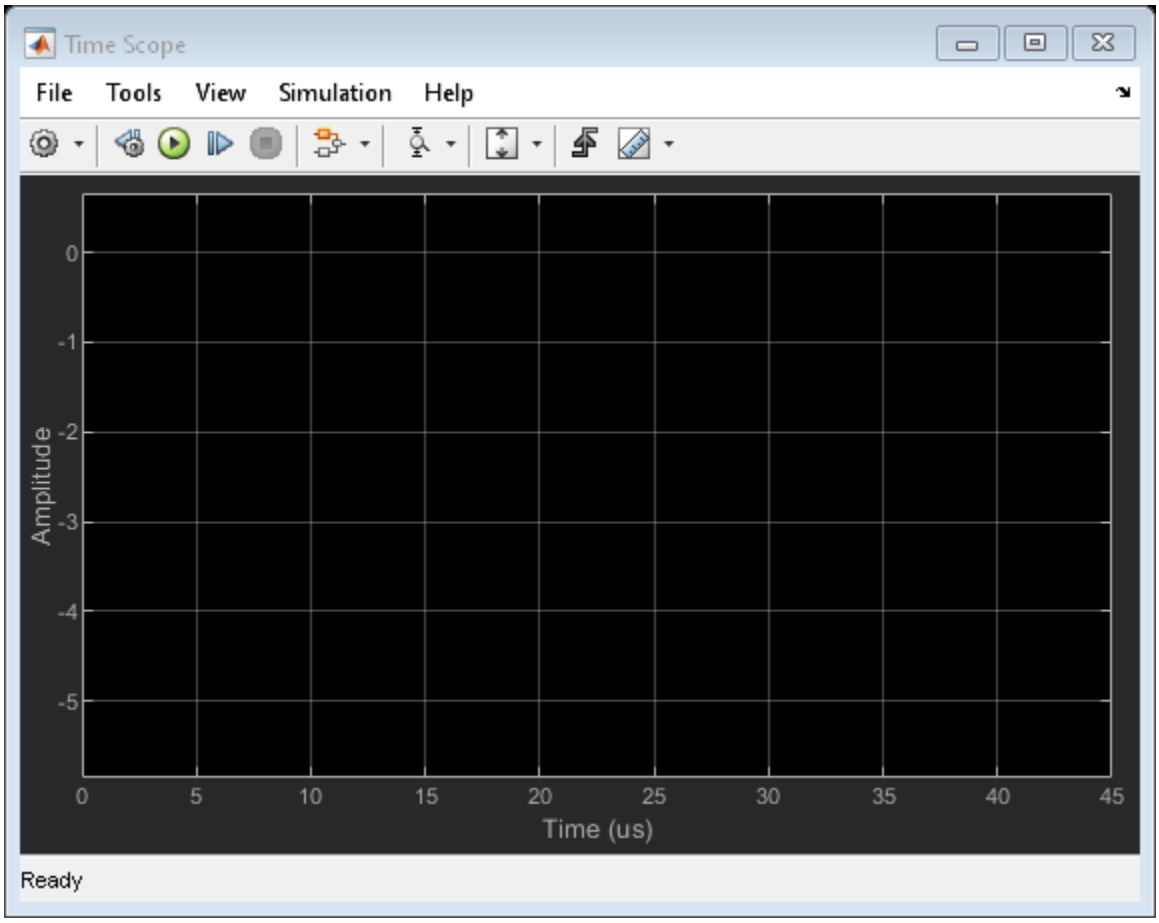
Circuit Design Details Affect PLL Performance

This example shows how to use the Linear Circuit Wizard block to evaluate the effect of loop filter circuit design details on the performance of a phase-locked loop.

Analog circuit imperfections such as circuit element limitations, element value variations, layout parasitics, and device noise can all measurably affect system-level performance. To release a system design for production, you need to evaluate the effects of these analog circuit design details to confirm that the manufactured system meets its performance requirements. For linear, time-invariant analog circuits, the Linear Circuit Wizard block can help by directly solving the detailed circuit equations and packaging the solution in the form of behavioral blocks that will execute efficiently in a Simulink® model.

Open the model `PllAnalogCircuitExample` attached to this example.





The model defines an integer-N single modulus PLL using the basic building blocks from Mixed-Signal Blockset™. The loop filter for the PLL is designed using the Linear Circuit Wizard block. For more information on the PLL model, see “Phase Noise at PLL Output” on page 1-2.

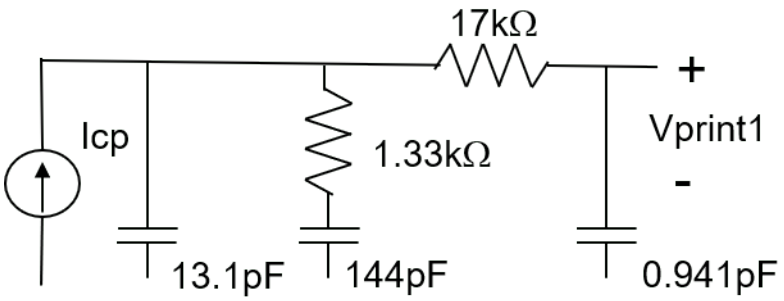
The PLL Testbench supplies a reference input signal for the PLL. This reference signal is modulated by a PRBS6 phase modulation that is used to measure the closed loop phase noise transfer function of the PLL. The input spectrum is evaluated by a spectrum estimator.

The output spectrum is measured both by the PLL Testbench and by a spectrum estimator like the one used to measure the input spectrum.

The model includes an oscilloscope to measure the PLL loop lock time.

Initial System Design

The chosen loop filter design is a third order passive loop filter, **R2** = `1.33` kΩ, **R3** = 17 kΩ, **C1** = `13.1` pF, **C2** = 144 pF, and **C3** = `0.941` pF [1].



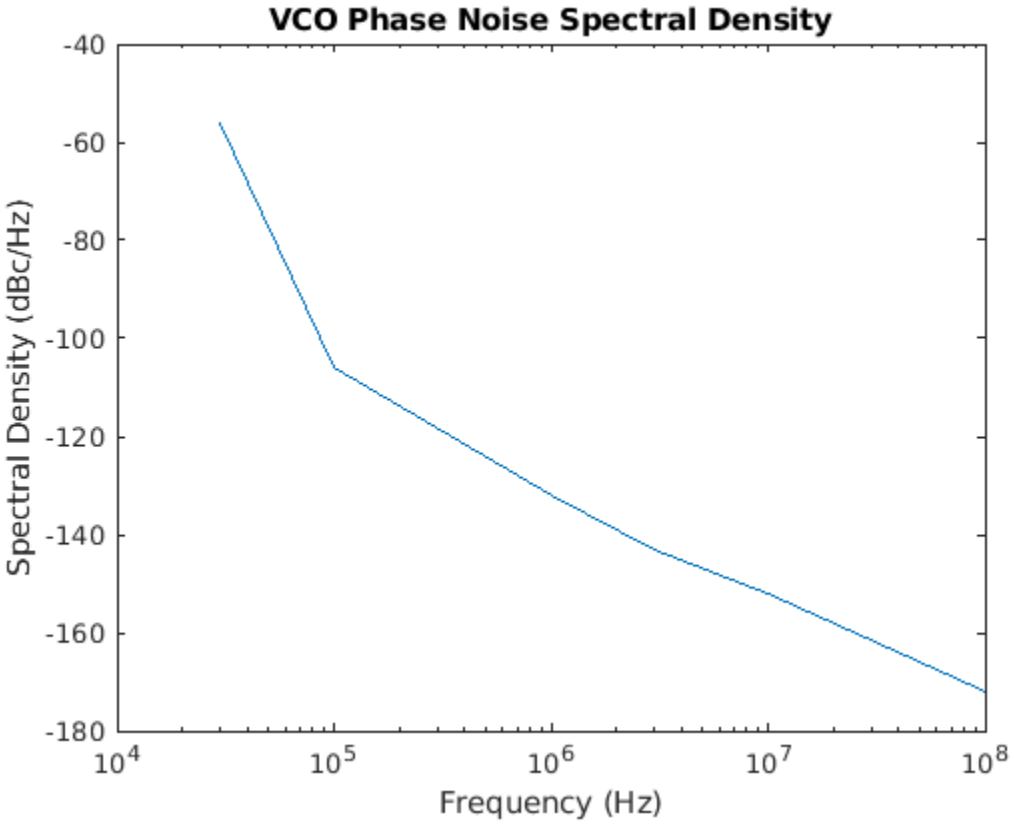
Third order passive loop filter

The other primary loop design parameters are:

- Reference frequency: `30` MHz
- Prescaler divider ratio: `70`
- VCO sensitivity: `100` MHz/V
- Charge pump output current: `1` mA

The primary circuit impairments are:

- VCO phase noise



- Charge pump imbalance: `0.1` mA
- Charge pump leakage: `0.01` mA
- Reference PRBS6 modulation peak level: `-60` dBc/Hz

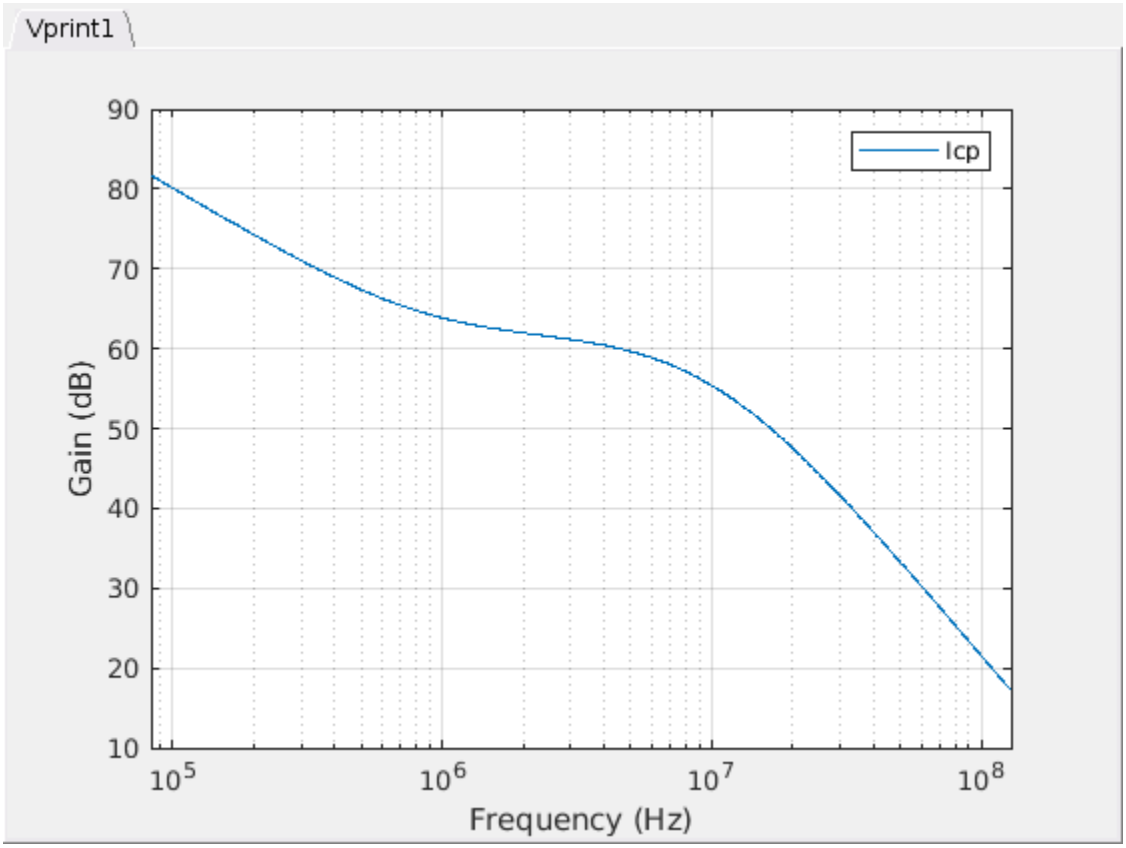
To create the loop filter block, start by creating or obtaining a SPICE netlist description of the circuit. This example uses a third order passive loop filter defined in the SPICE netlist file `3rdOrderLoopFilter.sp`. This netlist includes the independent current source **Icp** to define a current input port and a **.print** statement to define a voltage output port.

```
* Third order passive loop filter
* for preliminary system definition
Icp N1 0
C1 N1 0 13.1p
R2 N1 N2 1.33k
C2 N2 0 144p
R3 N1 N3 17k
C3 N3 0 0.941p
.print V(n3 0)
```

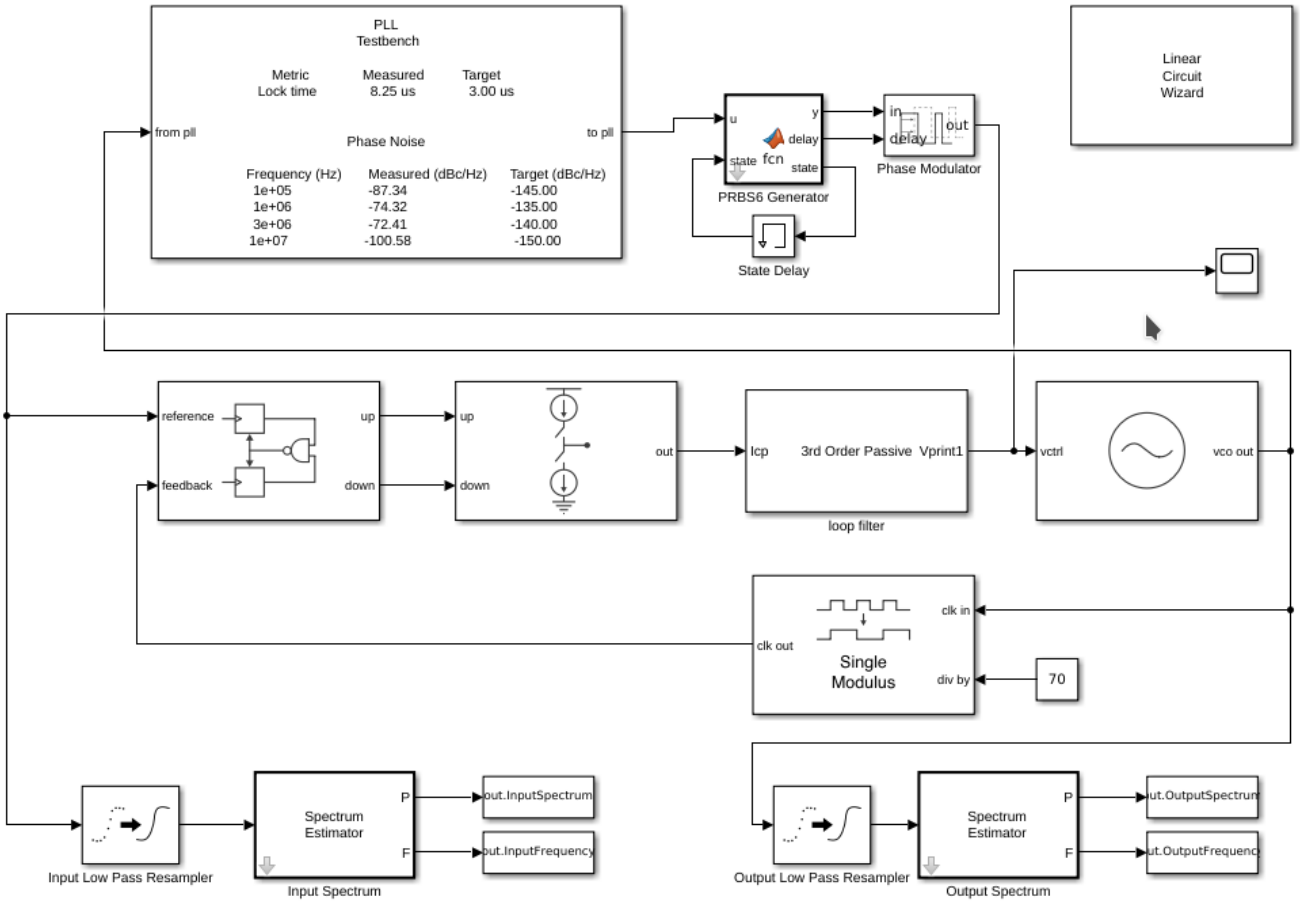
In the Linear Circuit Wizard block parameters dialog box, set the **Circuit design name** to `'3rd Order Passive'`, **Block name** to `'loop filter'`, and **Netlist file name** to `'3rdOrderLoopFilter.sp'`. Click the **Parse netlist file and redefine ports** button.

The **Port Definition** and **Device Noise Generators** tabs become visible after the netlist has been parsed. Review the content of the **Port Definition** tab to confirm that the port definitions are correct.

Click the **Plot transfer functions** button and review the resulting plot.

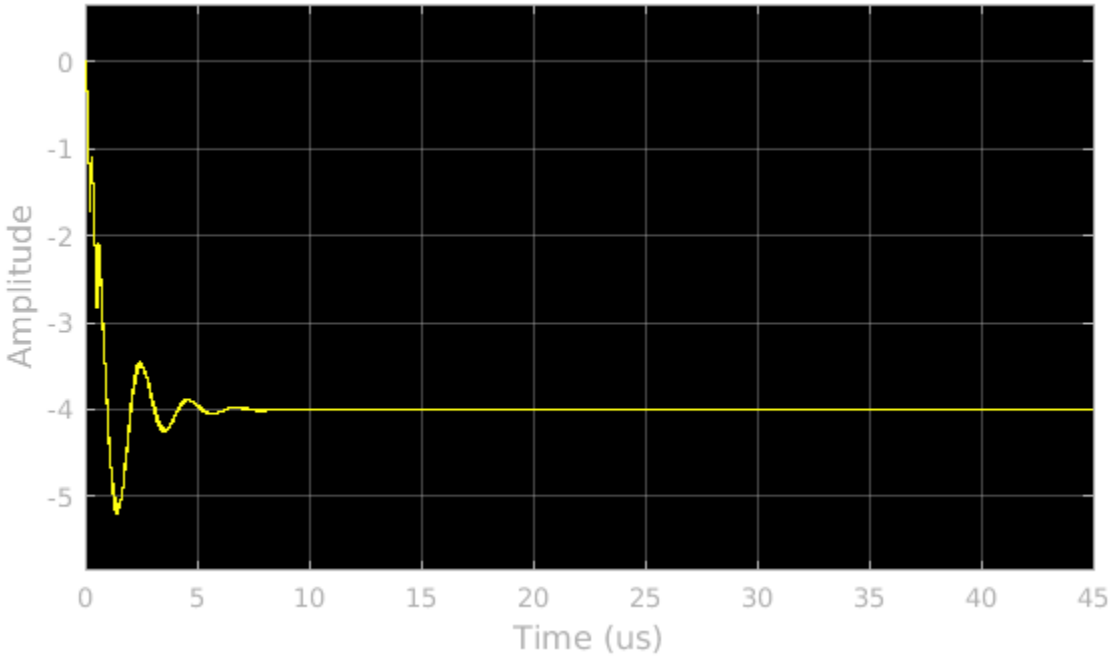


Click the **Build/modify block** button and connect the resulting loop filter block between the output of the charge pump and the VCO control voltage.

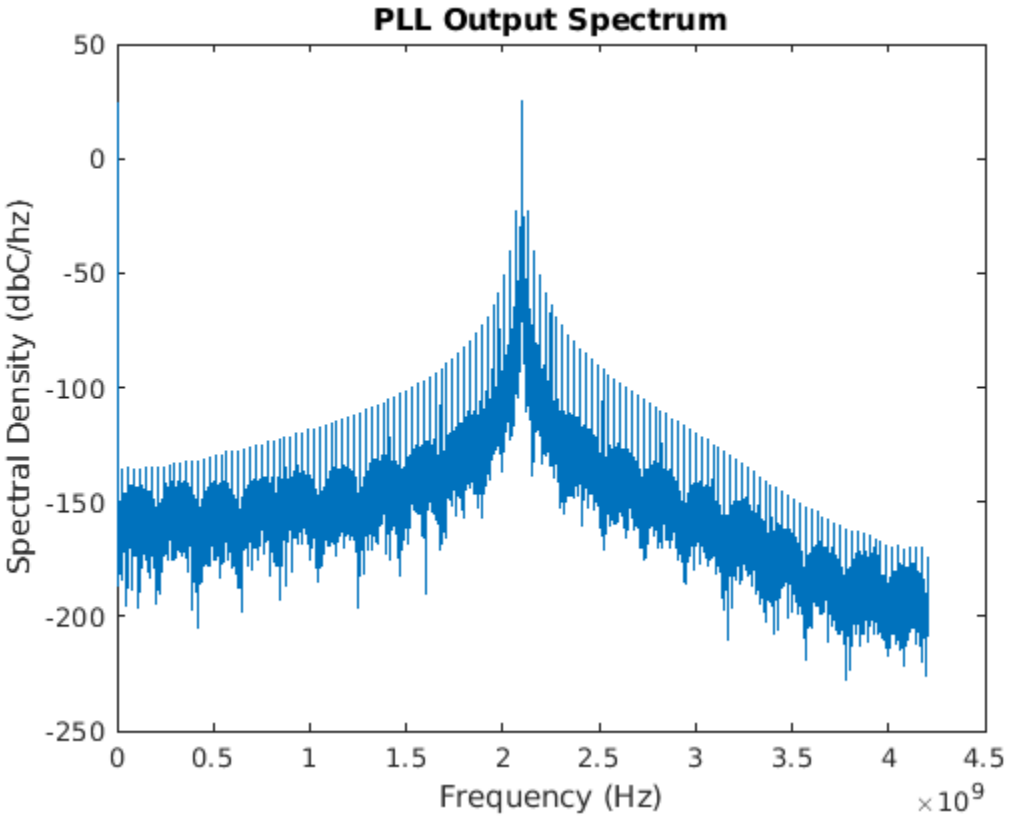


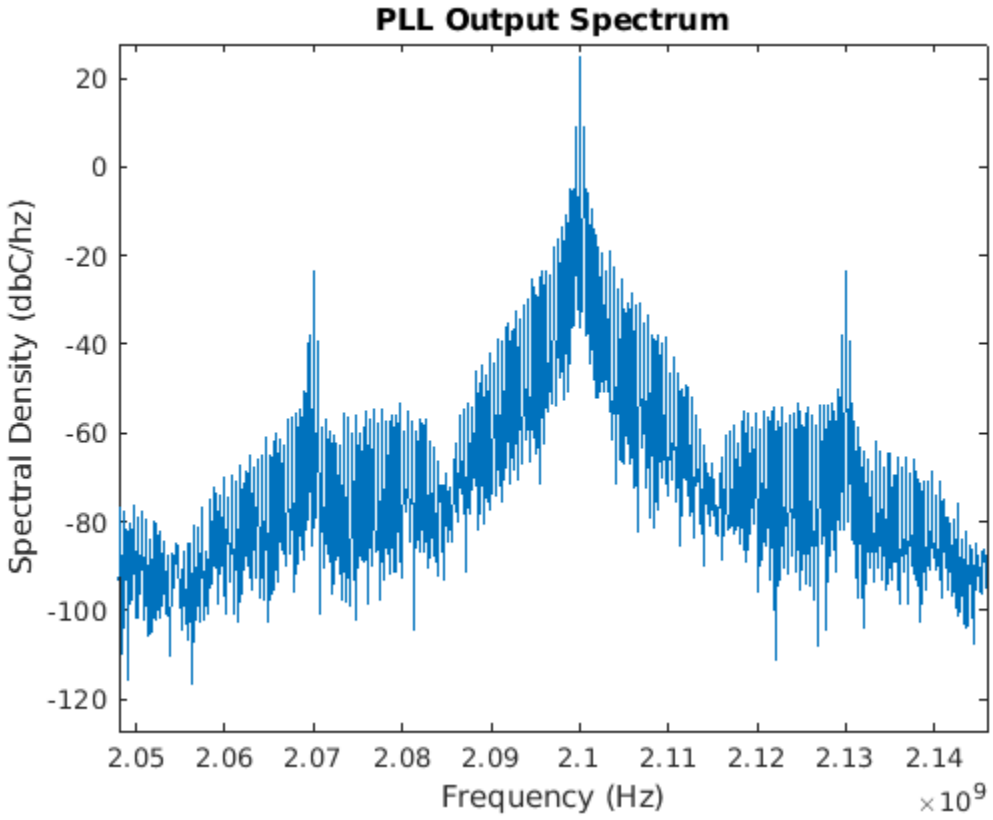
In the configuration of this model, the charge pump impairments are activated. Therefore, the charge pump defines a fixed step discrete sample time to drive the loop filter. Enabling the impairments also provides the level of detail needed in the later stages of development, at the cost of increased simulation run time. If the charge pump impairments are disabled, then the loop filter can be configured to provide its own sample time. However, a lowpass resampler, such as that used in the loop filter block in the “Building Blocks” library, is required to convert from the variable step discrete sample time of the charge pump without impairments to the fixed step discrete sample time of the loop filter.

Run the simulation.



To plot the PLL output spectrum, you can use the `plotPllOutputSpectrum` helper script attached to this example. The resulting figures highlight the spurious responses at `30` MHz intervals due to the charge pump imbalance, and the output spectrum due to the reference phase modulation. You can create simulations that highlight other effects by modifying the level of these and other impairments.


PLL Output Spectrum
Spectral Density (dbC/hz)
50
0
-50
-100
-150
-200
-250
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Frequency (Hz)
×10^9



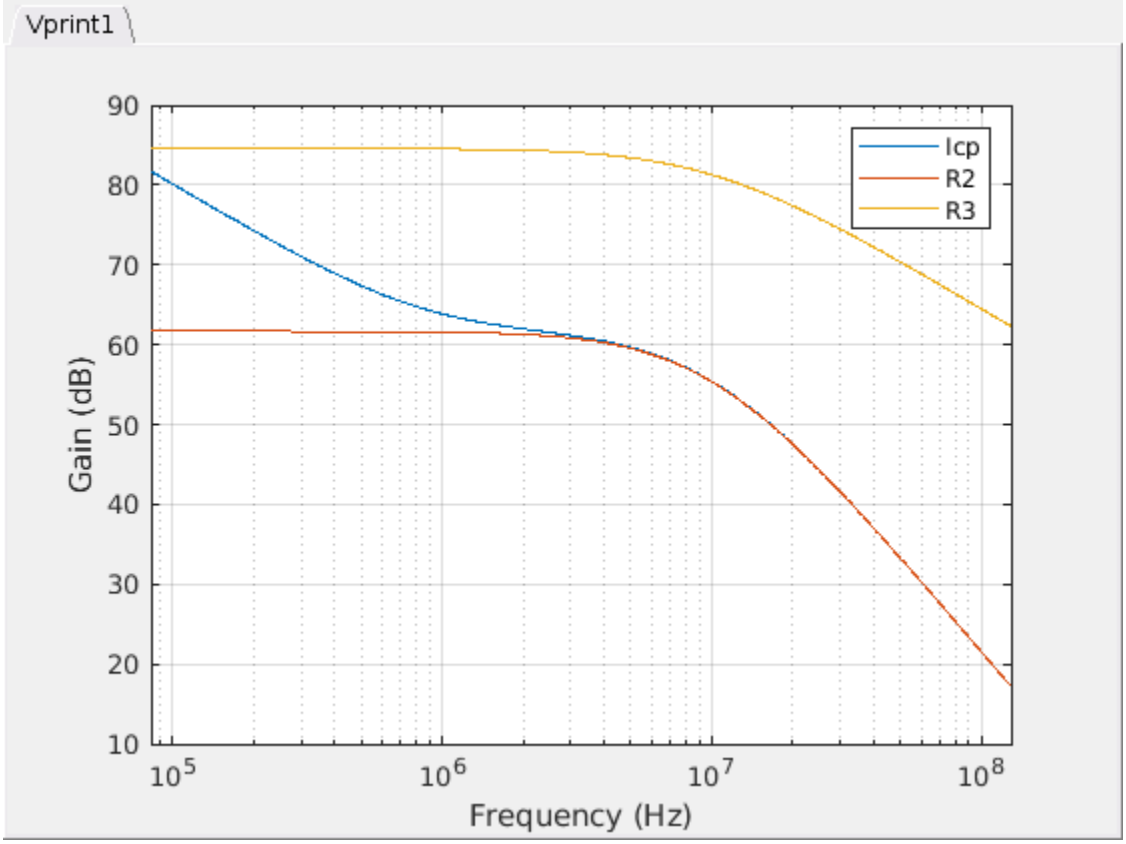
Add Device Noise

Add to the simulation the effect of device noise in the loop filter.

Enable and control the addition of device noise to the PLL model using the **Device Noise Generators** tab in the Linear Circuit Wizard block parameters dialog box. For **Circuit Element** R2 and R3, select **Enable device noise generator** and set the **Corner frequency (Hz)** to `10000` to include a flicker noise corner frequency of `10` kHz.

As soon as you change the definition of the block, for example by enabling device noise, the block mask displays a warning message indicating that the generated block does not reflect the latest changes. Apply the latest changes to the generated loop filter block by clicking the **Build/modify block** button. The warning message is removed and the block now includes the addition of the device noise.

Click the **Plot transfer functions** button. The transfer functions now include the transfer function from each device noise source to the output of the loop filter.

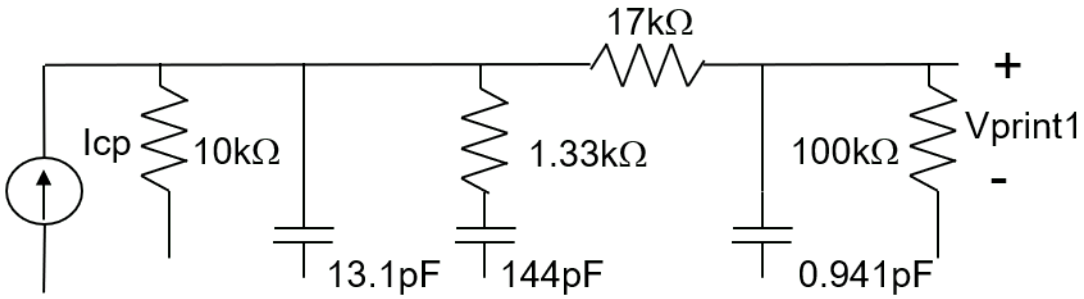


Re-run the simulation to include the effect of device noise in the results. To get a clearer evaluation of the effect of the device noise, set the charge pump current impairments to zero while leaving the charge pump impairments enabled, and disable the VCO phase noise. However if you do so, then set the charge pump impairments back to their original value and enable the VCO phase noise for the later sections of this example.

Practical Circuit Design

Detailed circuit designs that include the effects of circuit element limitations and layout parasitics typically only become available late in the development of a product. At that time, you should incorporate the detailed circuit design of critical components into the system model to confirm that the as-designed system is ready for production.

As a simple example of the types of circuit effects that should be included in the detailed as-designed model, add a charge pump output impedance of `10` kΩ and a VCO control voltage input impedance of `100` kΩ to the third order passive filter model.



Third order passive loop filter with input/output impedances

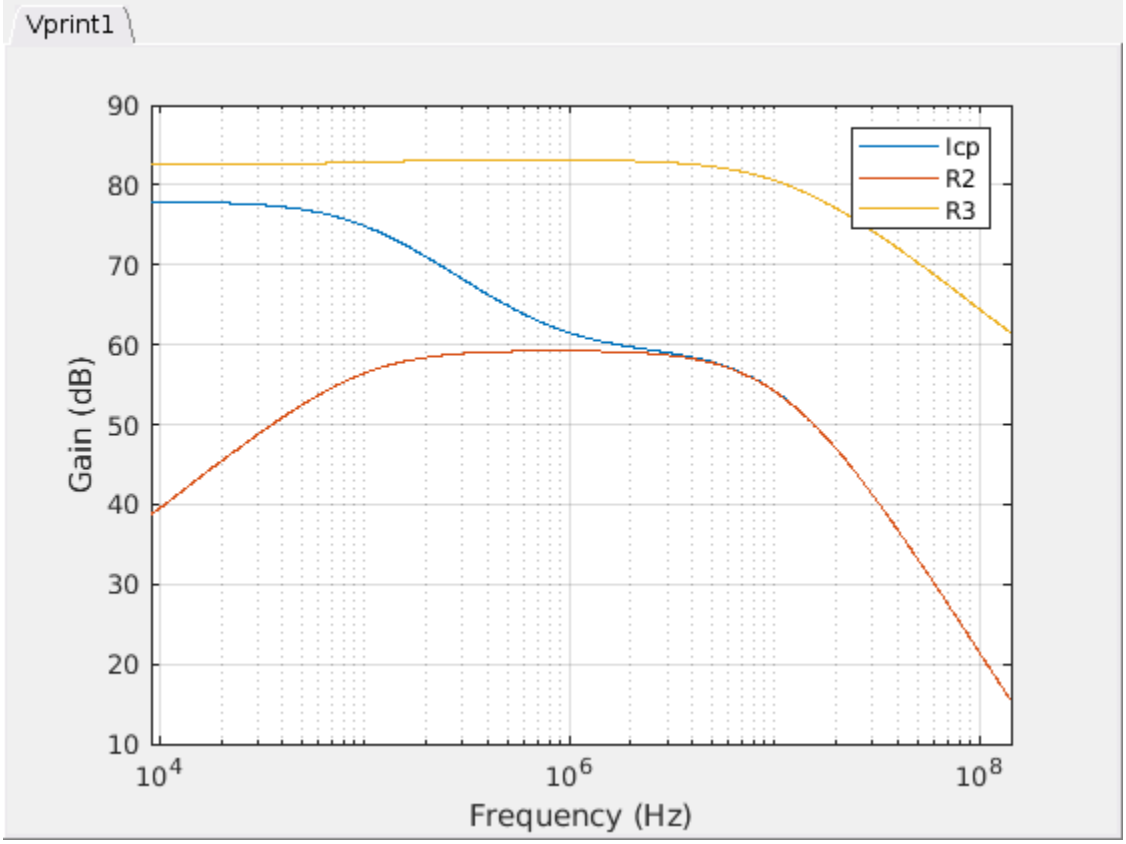
The associated SPICE netlist, as supplied in the file `3rdOrderCPLoading.sp` is:

```
* Third order passive loop filter
* with charge pump output impedance
Icp N1 0
Rs N1 0 10k
C1 N1 0 13.1p
R2 N1 N2 1.33k
C2 N2 0 144p
R3 N1 N3 17k
C3 N3 0 0.941p
R1 N3 0 100k
.print V(N3 0)
```

This schematic and netlist also illustrate an important principle when multiple circuit blocks are to be cascaded. You can cascade multiple linear circuit blocks created by the Linear Circuit Wizard block. The accuracy of your result depends on the accuracy of the modeling of circuit loading at both the input and output of each circuit block.

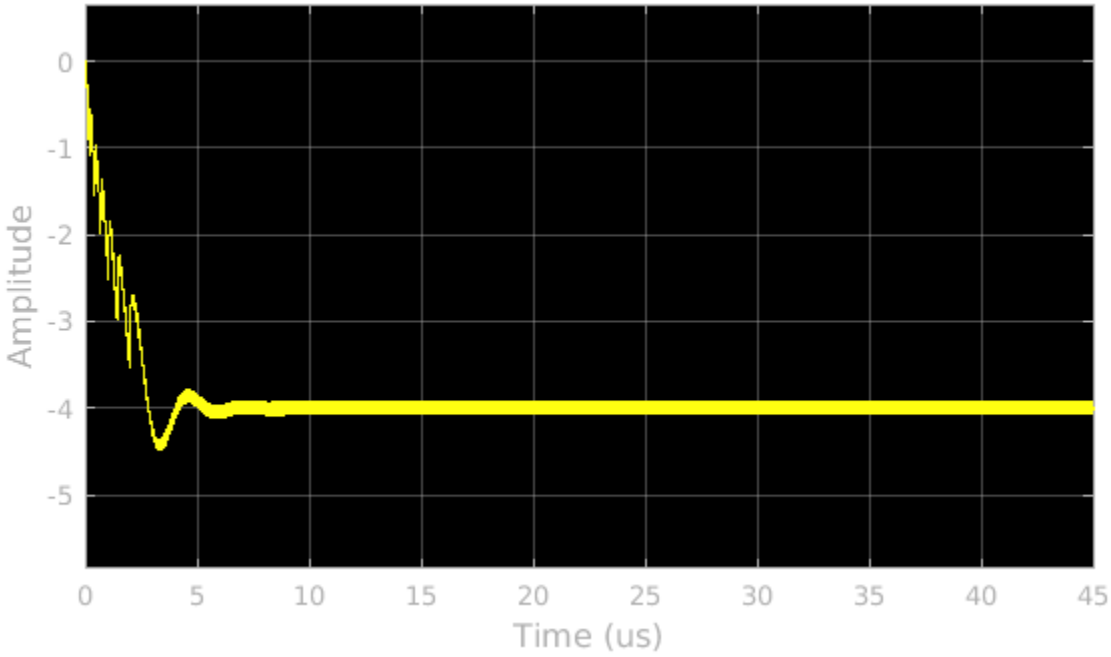
To evaluate the effect of loop filter circuit loading, change **Netlist file name** to `'3rdOrderCPLoading.sp'` in the Linear Circuit Wizard parameters dialog box and click the **Build/ modify block** button.

Plot the resulting transfer functions.

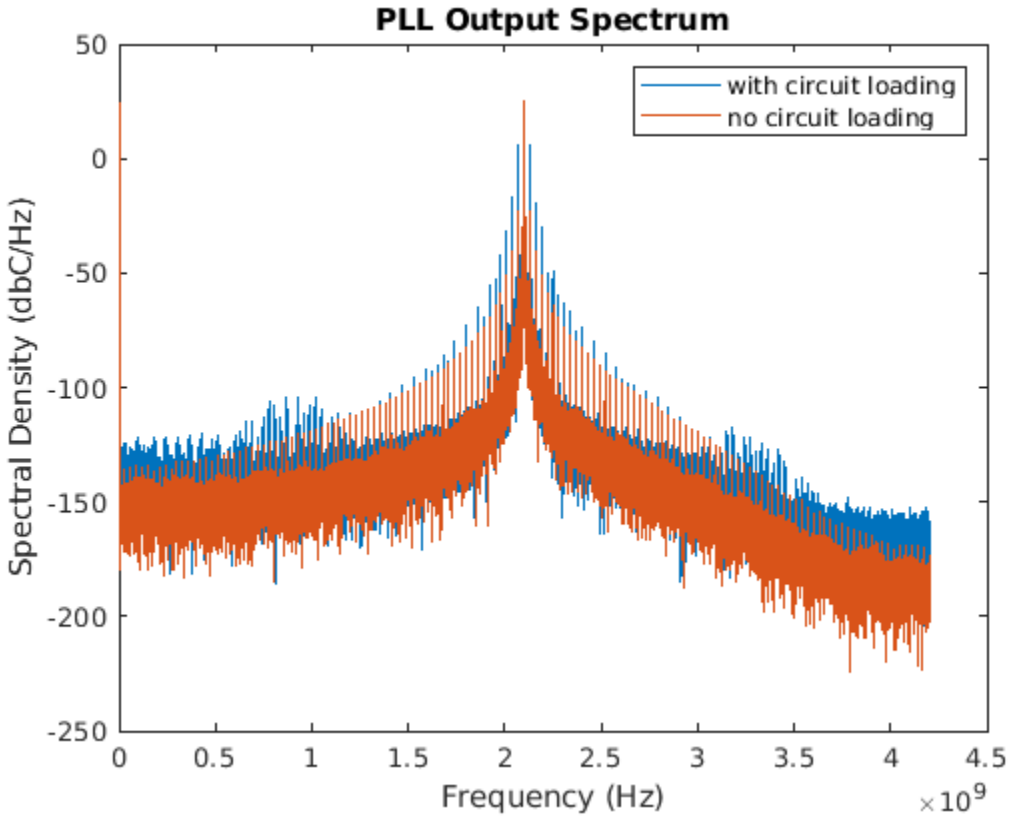


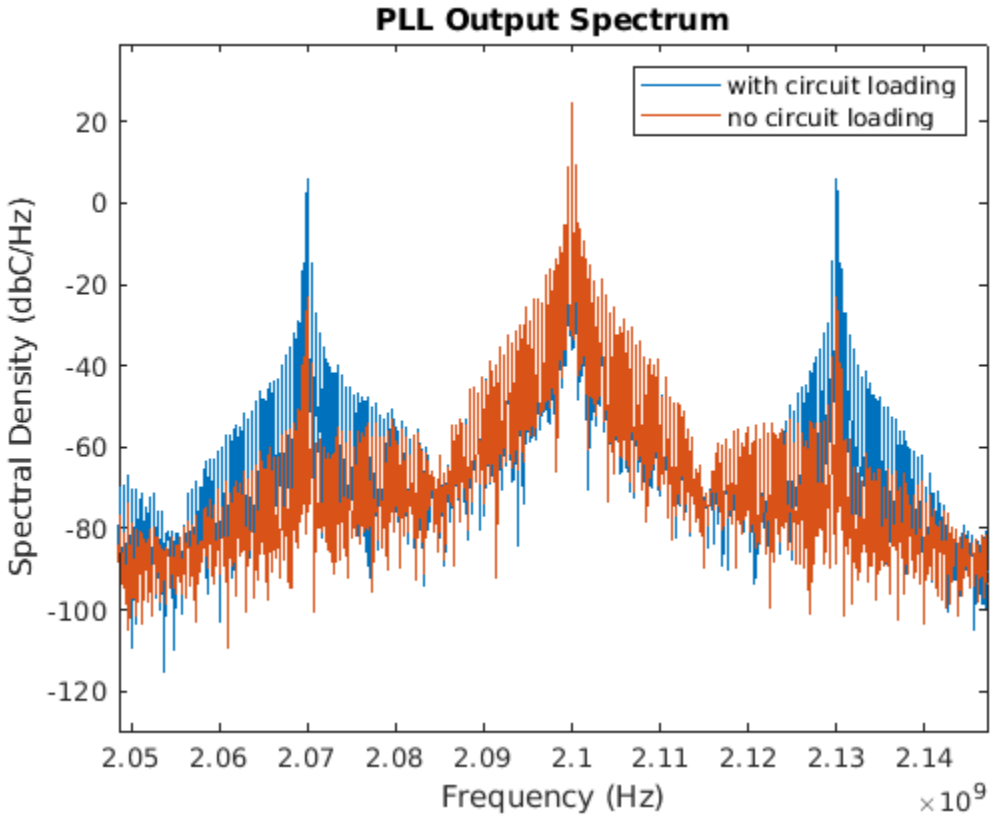
Re-run the simulation to include the effect of circuit design details in the results.

The shape of the loop acquisition response has changed. There are more cycle slips during loop acquisition, but much less overshoot. The resulting lock time remains approximately the same as the lock time for the initial system design. Further, there is significantly more noise in the steady state loop filter output, and the additional noise appears to have a more or less constant amplitude.



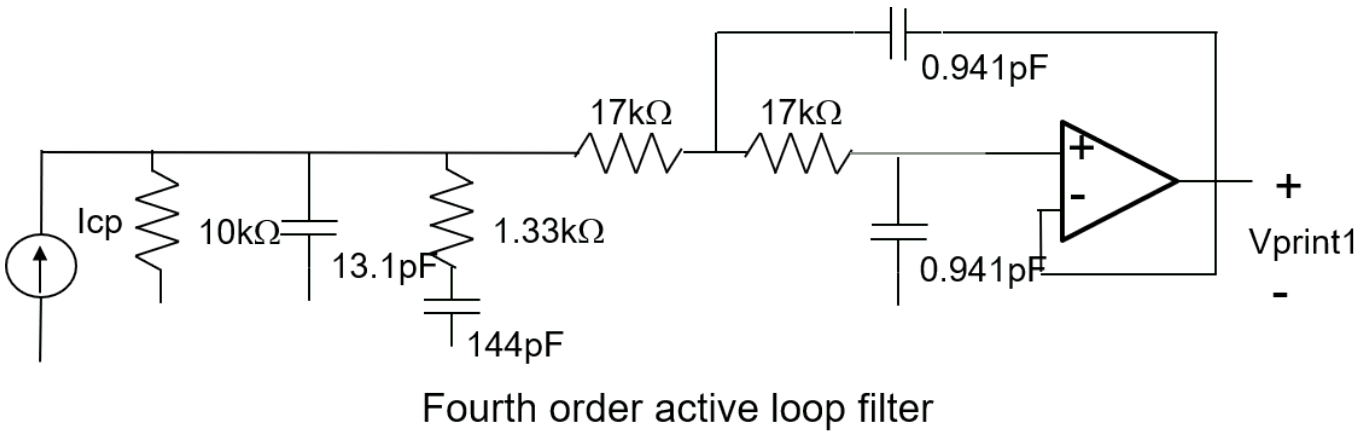
Plot the PLL output spectrum using the `plotPllOutputSpectrum` helper script. The primary impact of the circuit loading is a substantial increase in the spurious responses.





Enhanced Circuit Design

In this loop filter design, the last RC section of the loop filter with loading is replaced by a Sallen and Keye active filter. This circuit design introduces a pair of resonant poles with a modest Q factor.



Fourth order active loop filter

The associated netlist, as supplied in the file `4thOrderActiveFilter.sp`, is:

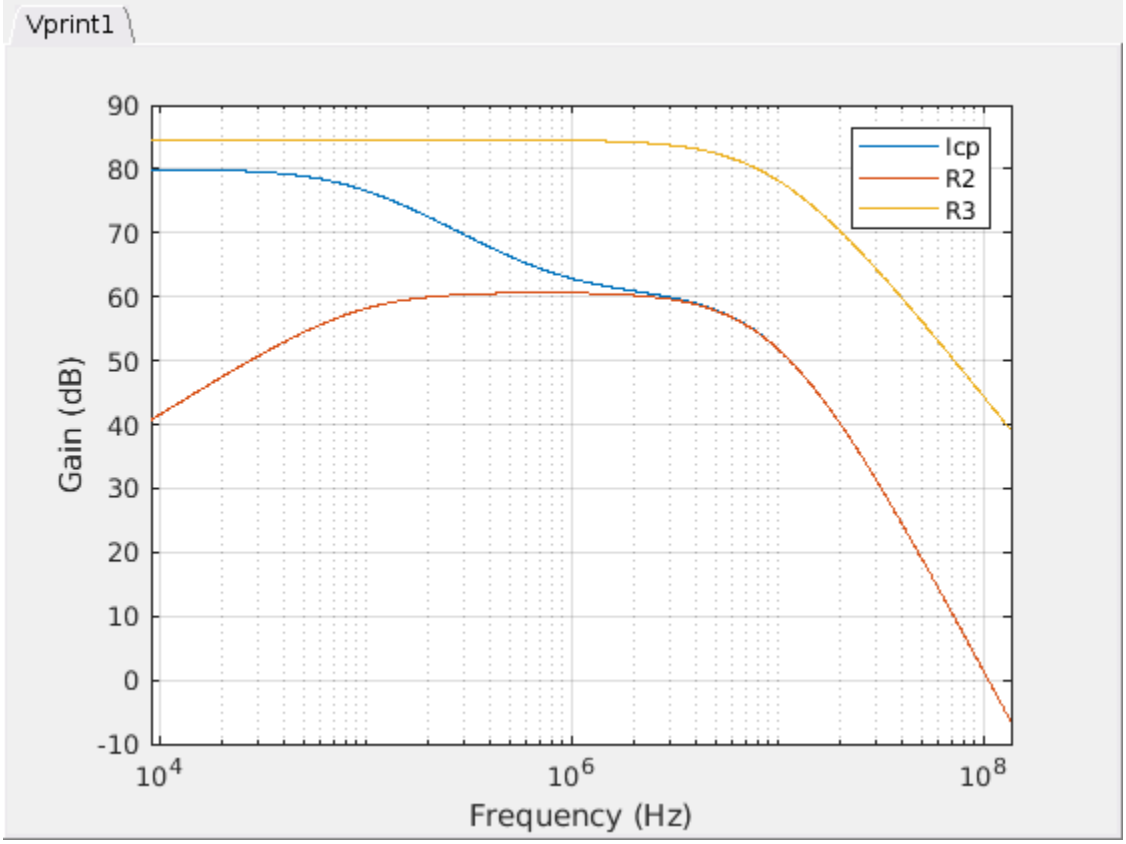
```
* Fourth order loop filter with Sallen and Keye output section
Icp N1 0 1e-3
Rs N1 0 10k
C1 N1 0 13.1p
```

```
R2 N1 N2 1.33k
C2 N2 0 144p
R3 N1 N4 17k
C3 N4 N3 0.941p
R4 N4 N5 17k
C4 N5 0 0.941p
E1 N3 0 LAPLACE N5 N3 6.3e7/6.3e4 1
.PRINT V(N3)
```

The operational amplifier in this circuit is represented as a voltage controlled voltage source. The open loop response of this amplifier is modeled using the LAPLACE keyword and the expression "`6.3e7/6.3e7 1`". This expression describes a rational transfer function with a numerator equal to `6.3e7` and the denominator `(s+6.3e4)`. In other words, the amplifier has an open loop DC gain of `1000` and a pole at `10` kHz. This syntax can readily describe transfer functions with more poles and zeros.

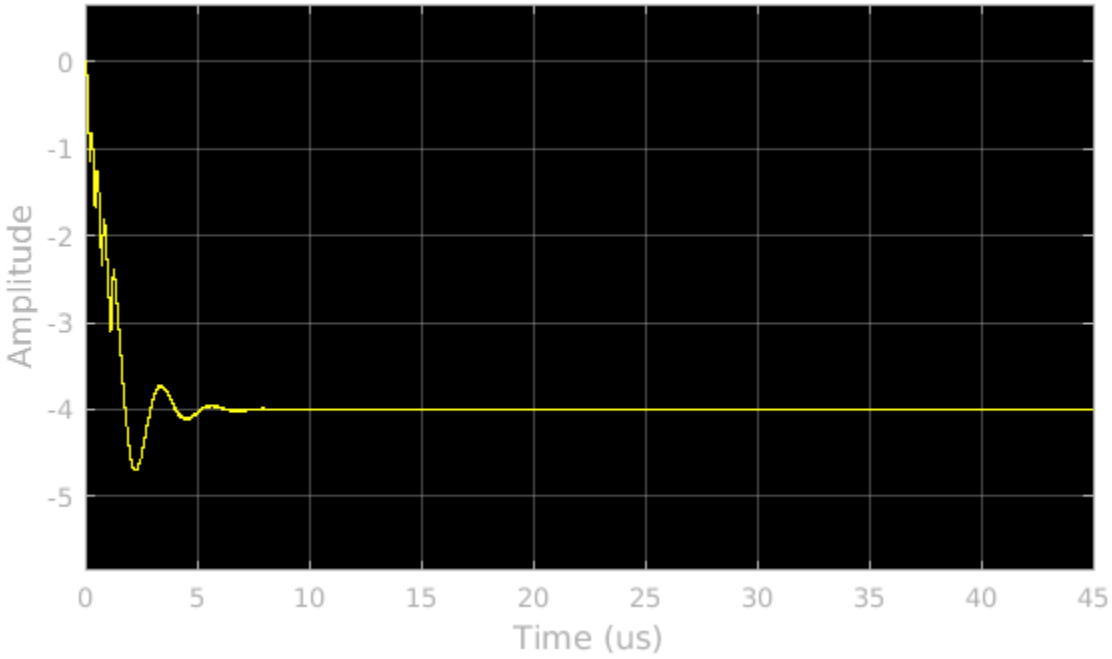
In the Linear Circuit Wizard block parameters dialog box, set **Circuit design name** to `'4th Order Active'` and **Netlist file name** to `'4thOrderActiveFilter.sp'`.

Plot the transfer functions. Although the low frequency response closely resembles the response of the passive filter with loading, the high frequency response rolls off much more rapidly.

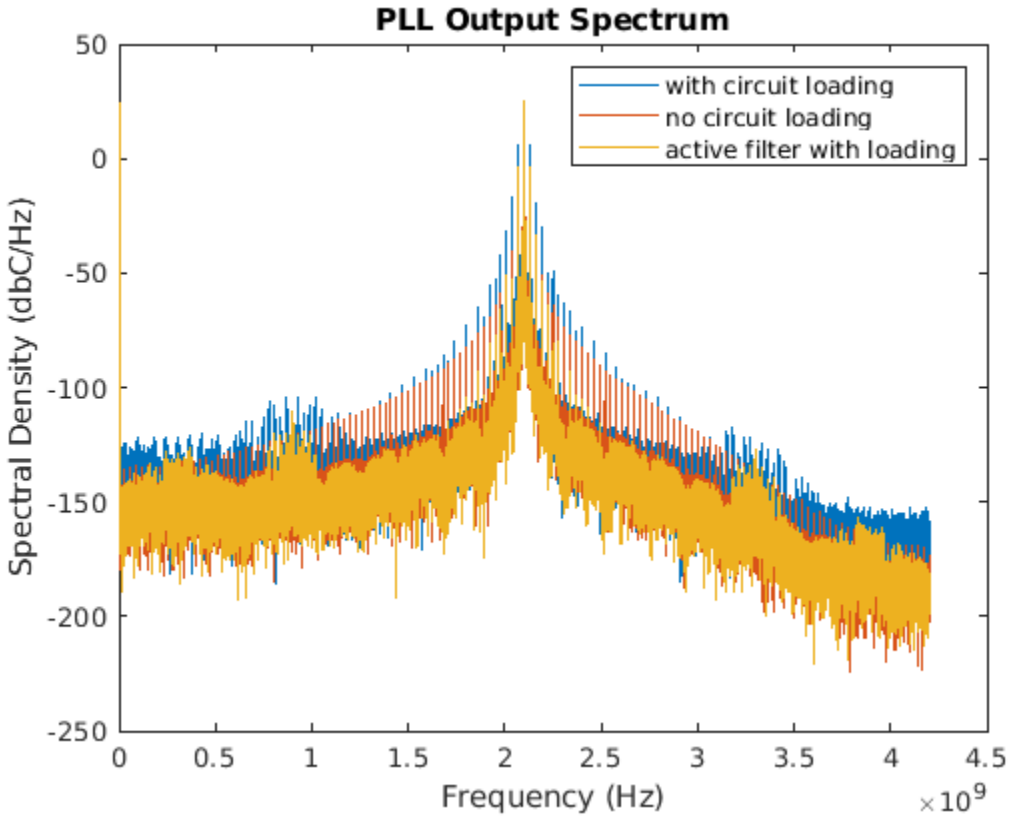


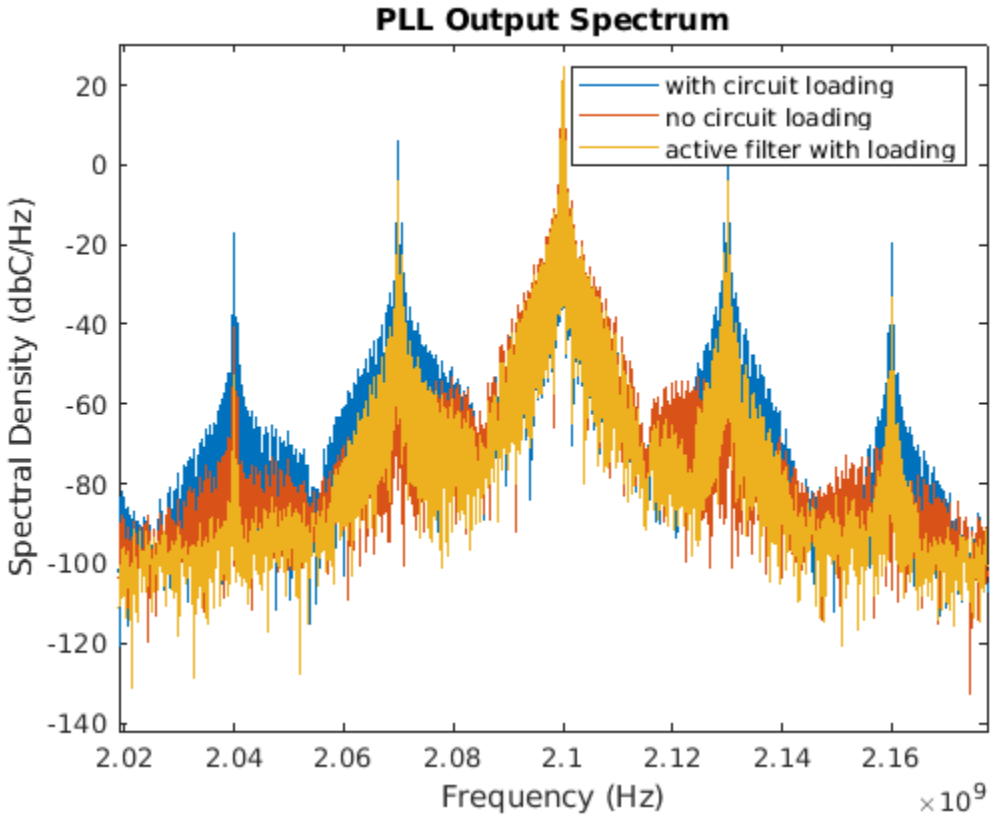
Click the **Build/modify block** button. The text on the block icon changes to match the revised circuit design name.

Re-run the simulation to evaluate the effect of the active loop filter on the PLL system performance. The loop acquisition response is similar to those for the other cases studied but the loop filter output is much smoother.



Plot the PLL output spectrum using the `plotPllOutputSpectrum` helper script. The output spectrum reflects the improvements to the system performance. Specifically, the out-of-band spurious responses are dramatically reduced and the in-band response remains essentially unchanged.


PLL Output Spectrum
with circuit loading
no circuit loading
active filter with loading
Spectral Density (dbC/Hz)
50
0
-50
-100
-150
-200
-250
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Frequency (Hz)
×10^9



References

1. Dean Bannerjee. *PLL Performance, Simulation, and Design*, 4th edition.

See Also

Linear Circuit Wizard

Related Examples

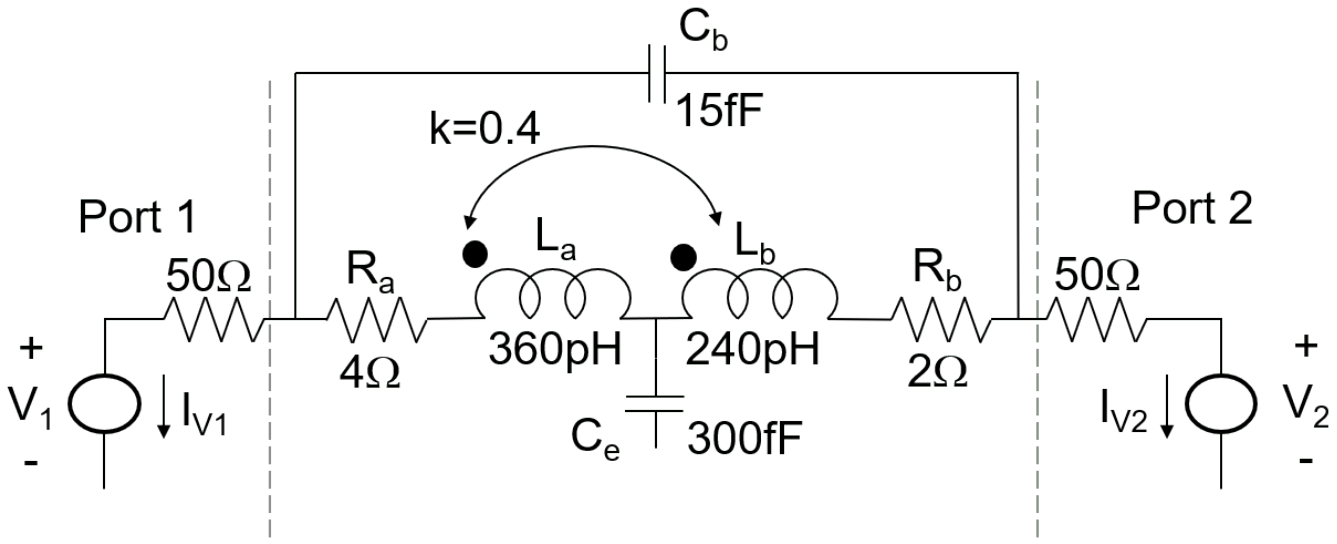
- “Phase Noise at PLL Output” on page 1-2
- “Analyze T-Coil Circuit” on page 3-33

Analyze T-Coil Circuit

This example shows how to use the Linear Circuit Wizard block to analyze a T-coil impedance matching circuit and generate S-parameter data from the result.

The T-coil circuit is used in many applications to impedance match a capacitive load, such as an electrostatic discharge (ESD) protection device, over as broad a frequency band as possible [1], [2]. You need to include the effects of a T-coil circuit in the model of any broadband or microwave channel that uses it. Depending on the application, you may insert the transfer function of the circuit into a model of the channel or insert the S-parameters of the circuit into the RF analysis of the channel.

This example analyzes the T-coil circuit shown in this schematic diagram. Capacitor **Ce** is the capacitor to be impedance matched [3]. The schematic includes the equivalent circuits for two S-parameter ports. For more about S-parameter ports, see the **S-Parameters** section of the example.



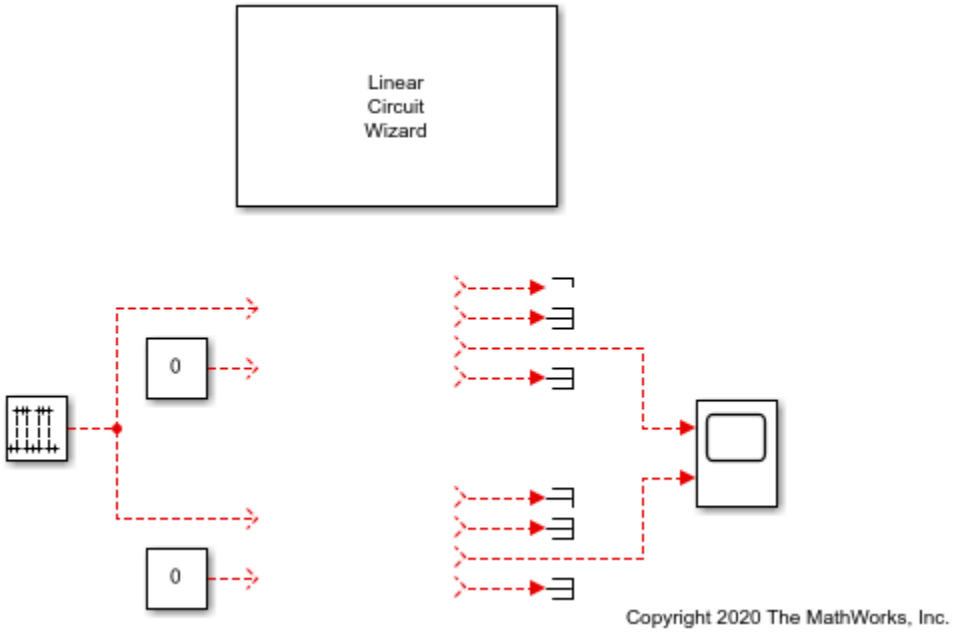
The file `TCoil.sp` contains the SPICE netlist for the T-coil circuit.

```
* SPICE netlist for T-coil example
* The circuit itself
Ra 2 3 4
La 3 4 360e-12
Cb 2 6 15e-15
Kab La Lb 0.4
Ce 4 0 300e-15
Lb 4 5 240e-12
Rb 5 6 2
* Create an S-parameter port by driving with a voltage source
* and load resistor, and then measure the voltage and current
* at the circuit end of the load resistor.
*
* Port 1 definition
V1 1 0
R1 1 2 50
.print V(2 0) I(V1)
```

```
* Port 2 definition
V2 7 0
R2 6 7 50
.print V(6 0) I(V2)
.end
```

Open the `TCoil` model. The model contains the Linear Circuit Wizard that you can use to create Linear Circuit blocks for the T-coil circuit and a baseline circuit without impedance matching. The model also contains a test setup used to compare the pulse response of the T-coil circuit to the baseline circuit. The time scope from the DSP System Toolbox library is used rather than the scope from the Simulink library because its rendering of the waveforms is smoother in this application.

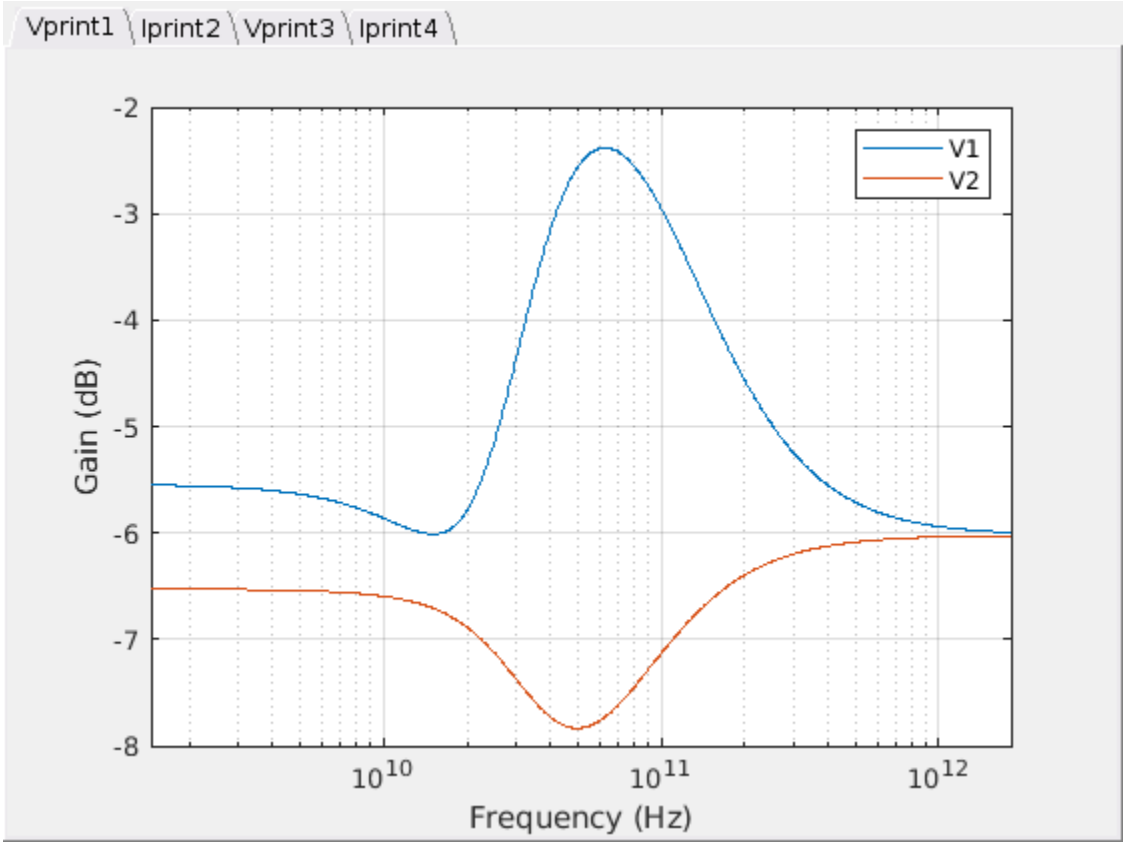
```
open_system('TCoil.slx');
```



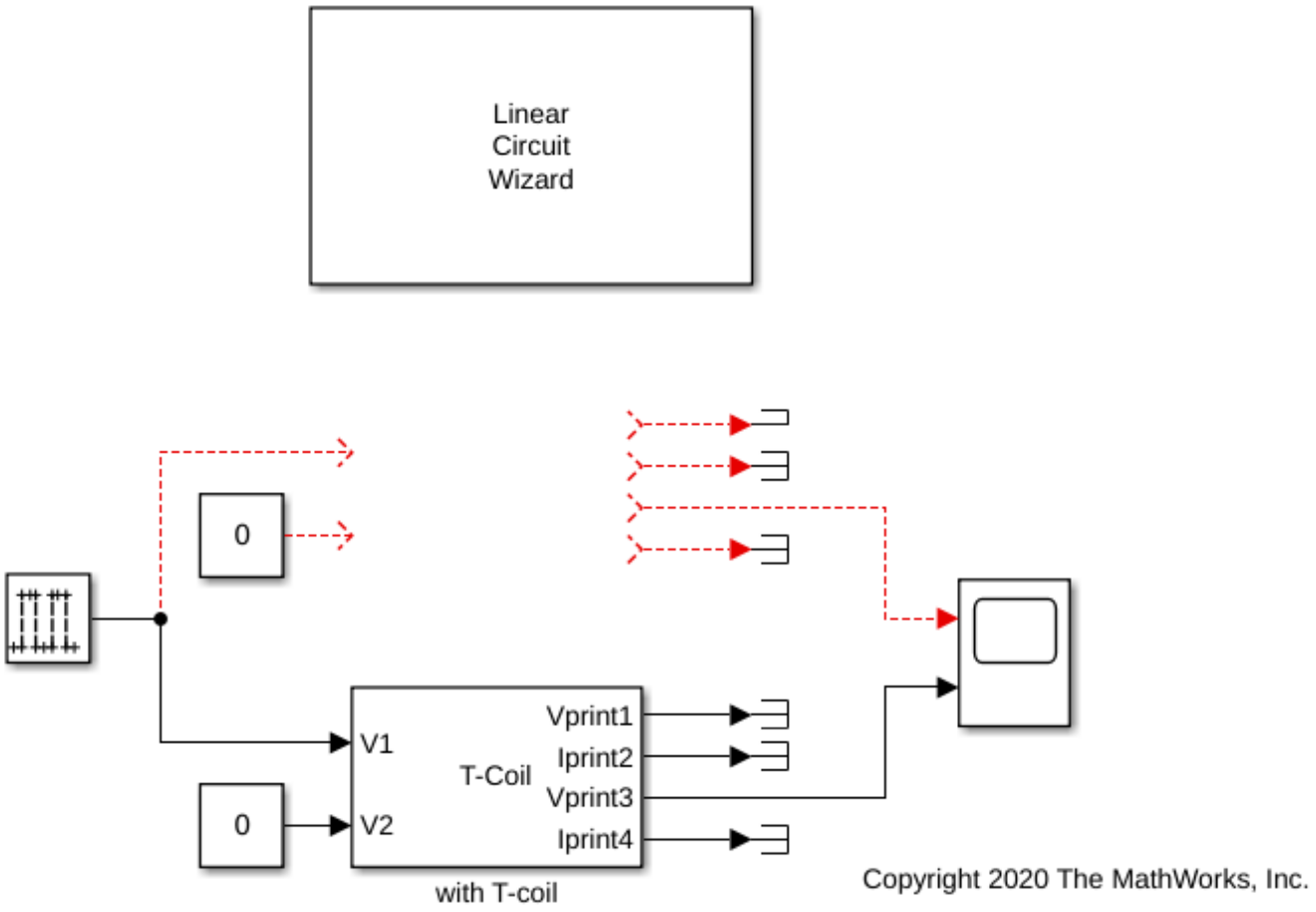
Select the Linear Circuit Wizard block. In the block mask or the Property Inspector, set the **Circuit design name** to `'T-Coil'` and the **Block name** to `'with T-coil'`.

Parse the SPICE netlist for the T-coil circuit by clicking the **Parse netlist file and redefine ports** button. The **Port Definition** and **Device Noise Generators** tabs become visible in the block parameter dialog box. If desired, review the port settings and device noise settings.

Display the frequency response of the T-coil circuit by clicking the **Plot transfer functions** button.



Build a Linear Circuit block for the T-coil circuit by clicking the **Build/modify block** button. Connect the block into the lower channel in the model. The active input port is **V1** and the active output port is **Vprint3**.



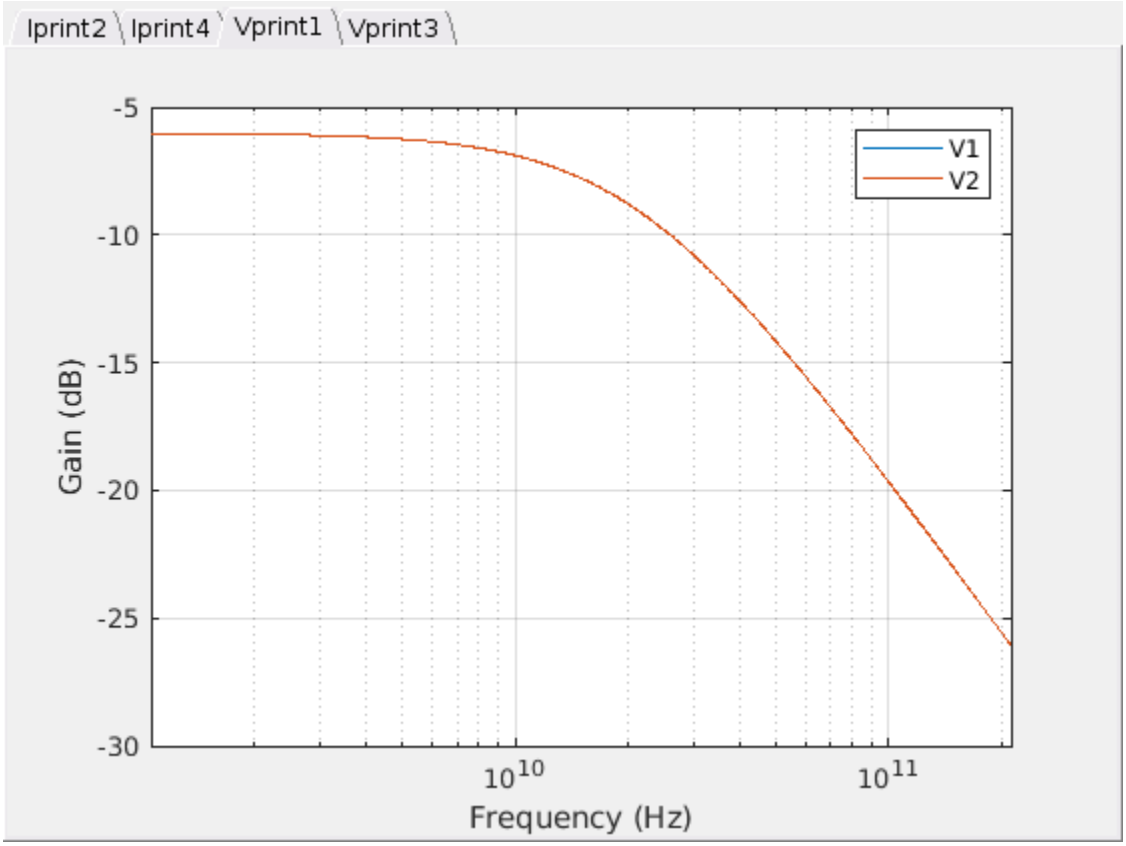
This example compares the T-coil circuit against a baseline circuit without impedance matching. The baseline circuit is a shunt capacitor connected to two S-parameter ports. The file `NoTCoil.sp` contains the SPICE netlist for this circuit.

```
* SPICE netlist for T-coil example
* The load parasitic capacitance without matching circuit
Ce 4 0 300e-15

* Create an S-parameter port by driving with a voltage source
* and load resistor, and then measure the voltage and current
* at the circuit end of the load resistor.
*
* Port 1 definition
V1 1 0
R1 1 4 50
.print V(4 0) I(V1)
* Port 2 definition
V2 7 0
R2 4 7 50
.print V(4 0) I(V2)
.end
```

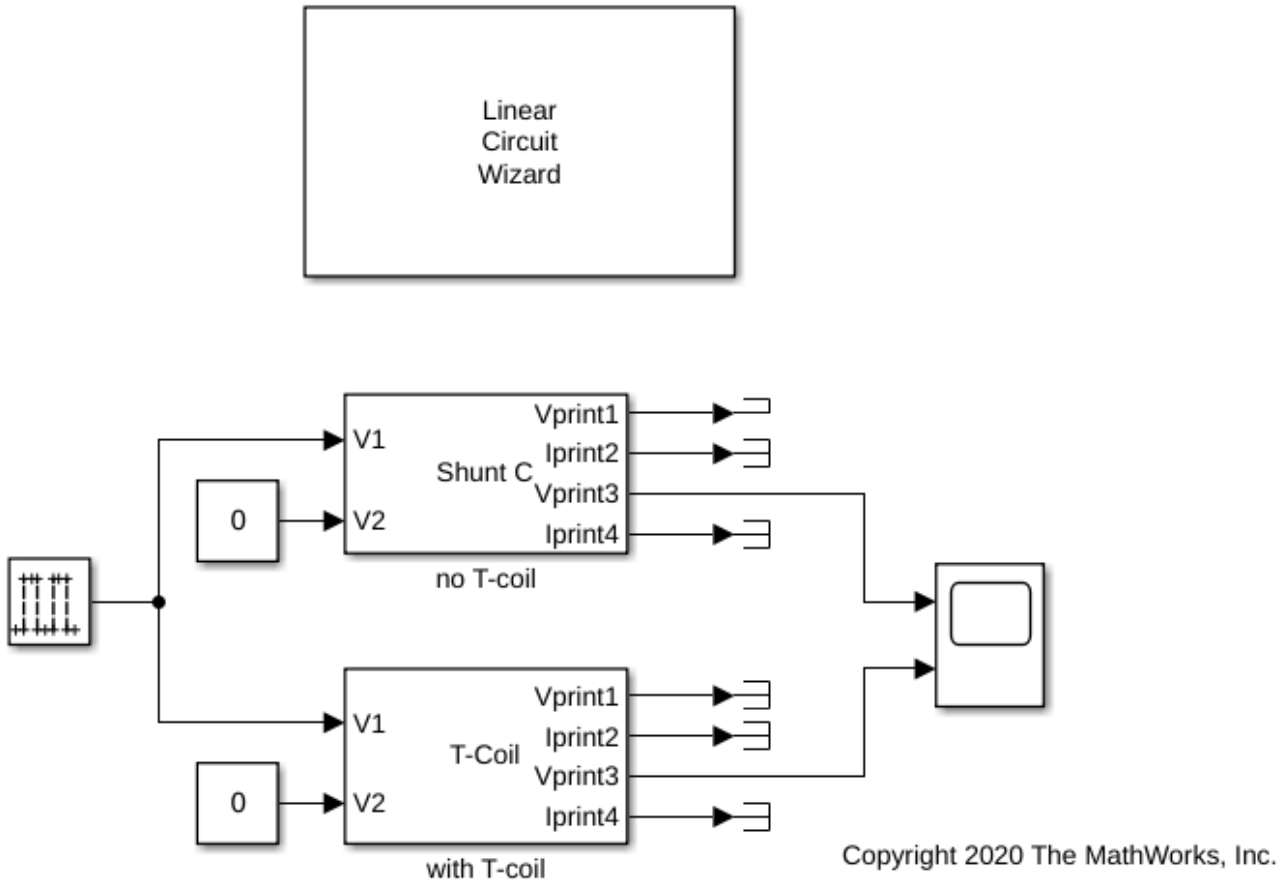
Return to the Linear Circuit Wizard block. In the block mask or the Property Inspector, set the **Circuit design name** to `'Shunt C'`, the **Block name** to `'no T-coil'`, and the **Netlist file name**

to `'NoTCoil.sp'`. Parse the netlist for the shunt capacitor circuit by clicking the **Parse netlist and redefine ports** button.

Display the frequency response of the shunt capacitor circuit by clicking the **Plot transfer functions** button. The response of the shunt capacitor falls off much faster at high frequencies than the response of the T-coil circuit.

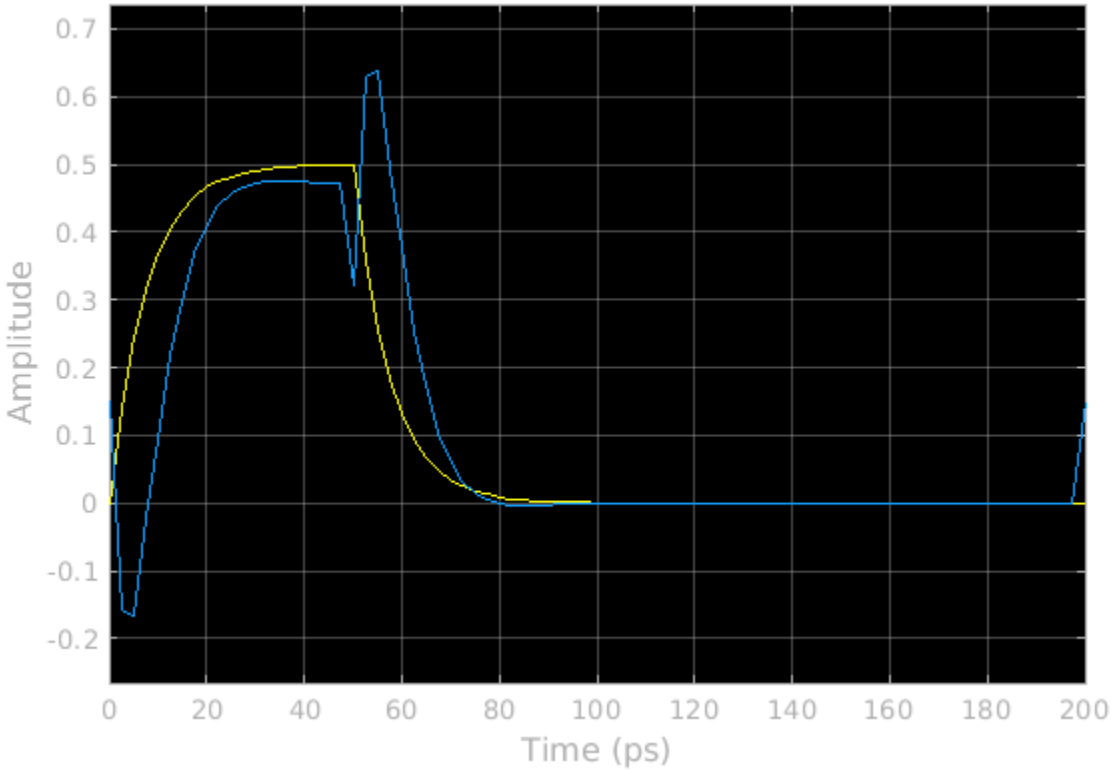


Build a Linear Circuit block for the shunt capacitor by clicking the **Build/modify block** button. Connect this block into the upper channel in the model. The active input port is **V1** and the active output port is **Vprint3**.



Run the simulation. In this scope display, the yellow trace is the pulse response of the shunt capacitor without impedance matching and the blue trace is the pulse response of the T-coil circuit.

Although the T-coil circuit introduces a little delay, the resulting pulse response is more desirable for data transmission, with a flat top and more compact rising and falling edges.



S-Parameters

The Linear Circuit Wizard defines ports as being either input or output and as being either voltage or current. In contrast, S-parameters are expressed in terms on incoming waves and outgoing waves at each port, based on an assumed transmission line impedance at each port. You therefore need some specific assumptions and some conversion to obtain S-parameters for a linear circuit.

The example makes these assumptions:

1. Each port is connected to a source/load impedance of **Z0** (typically `50` Ω).
2. Each port is driven by an independent voltage source driving a resistor whose value is equal to the source/load impedance. This voltage source is an input voltage port.
3. The voltage at each port is available as an output. This is a voltage output port.
4. The current flowing through each source/load resistor is available as an output. This is a current output port. In the context of the SPICE netlist, this current is measured by the independent voltage source that defines the input voltage port.

Given these assumptions, you can transform the port node voltages and currents into incoming and outgoing waves to produce S-parameters. Given incoming wave voltage w_i and outgoing wave voltage w_o at a port with characteristic impedance Z_0, and remembering that the port current is defined as an output current of the circuit, the port voltage and port current are:

$$v = w_i + w_0$$

$$i = -\frac{w_i}{Z_0} + \frac{w_o}{Z_0}$$

Solving these equations yields the resulting waves:

$$w_i = \frac{v - Z_o i}{2}$$

$$w_o = \frac{v + Z_o i}{2}$$

The S parameter matrix is then a matrix of outgoing wave amplitudes divided by incoming wave amplitudes.

The netlist `TCoil.sp` satisfies this set of assumptions for S-parameter **Port 1** through the following group of statements:

```
* Port 1 definition
V1 1 0
R1 1 2 50
.print V(2 0) I(V1)
```

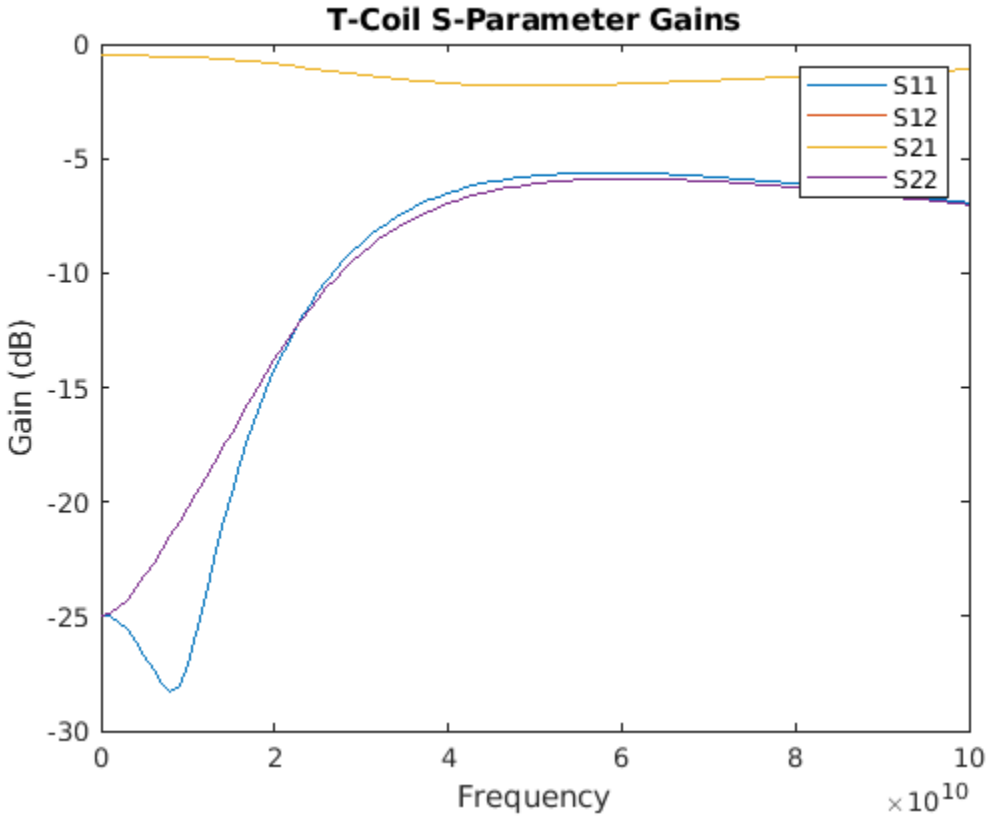
V1 defines the voltage input port and **R1** is the source/load resistor. The `.print` statement defines a voltage output port from node 2 to the return node and a current output port whose output is the current flowing through the voltage source **V1**.

A similar set of statements defines S-parameter **Port 2**.

You can calculate the S-parameter data by first defining a desired frequency scale and then using the `generateSParameterData` helper function. Parse the netlist of interest and then click the **Export poles and zeros** button. Obtain the S-parameter data from DC to 100 GHz in 1 GHz steps.

```
freq = (0:100)*1e9;
[data,errmsg] = generateSParameterData('TCoil/Linear Circuit Wizard',freq);
```

The resulting data structure is directly compatible with the `S-parameters` object and the `rfwrite` function in the RF Toolbox™.



References

1. Razavi, Behzad. "The Bridged T-Coil [A Circuit for All Seasons]." IEEE Solid-State Circuits Magazine 7, no. 4 (Fall 2015): 9–13. https://doi.org/10.1109/MSSC.2015.2474258.
2. Ross, Bob. "T-Coil Topics." DesignCon IBIS Summit, Santa Clara, Ca. February 3, 2011. https://ibis.org/summits/feb11/ross.pdf
3. Kossel, Marcel, Christian Menolfi, Jonas Weiss, Peter Buchmann, George von Bueren, Lucio Rodoni, Thomas Morf, Thomas Toifl, and Martin Schmatz. "A T-Coil-Enhanced 8.5Gb/s High-Swing Source-Series-Terminated Transmitter in 65nm Bulk CMOS." In 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, 110–599, 2008. https://doi.org/10.1109/ISSCC.2008.4523081.

See Also

Linear Circuit Wizard | `sparameters` | `rfwrite`

Related Examples

- “Circuit Design Details Affect PLL Performance” on page 3-16

Design Inverting Amplifier

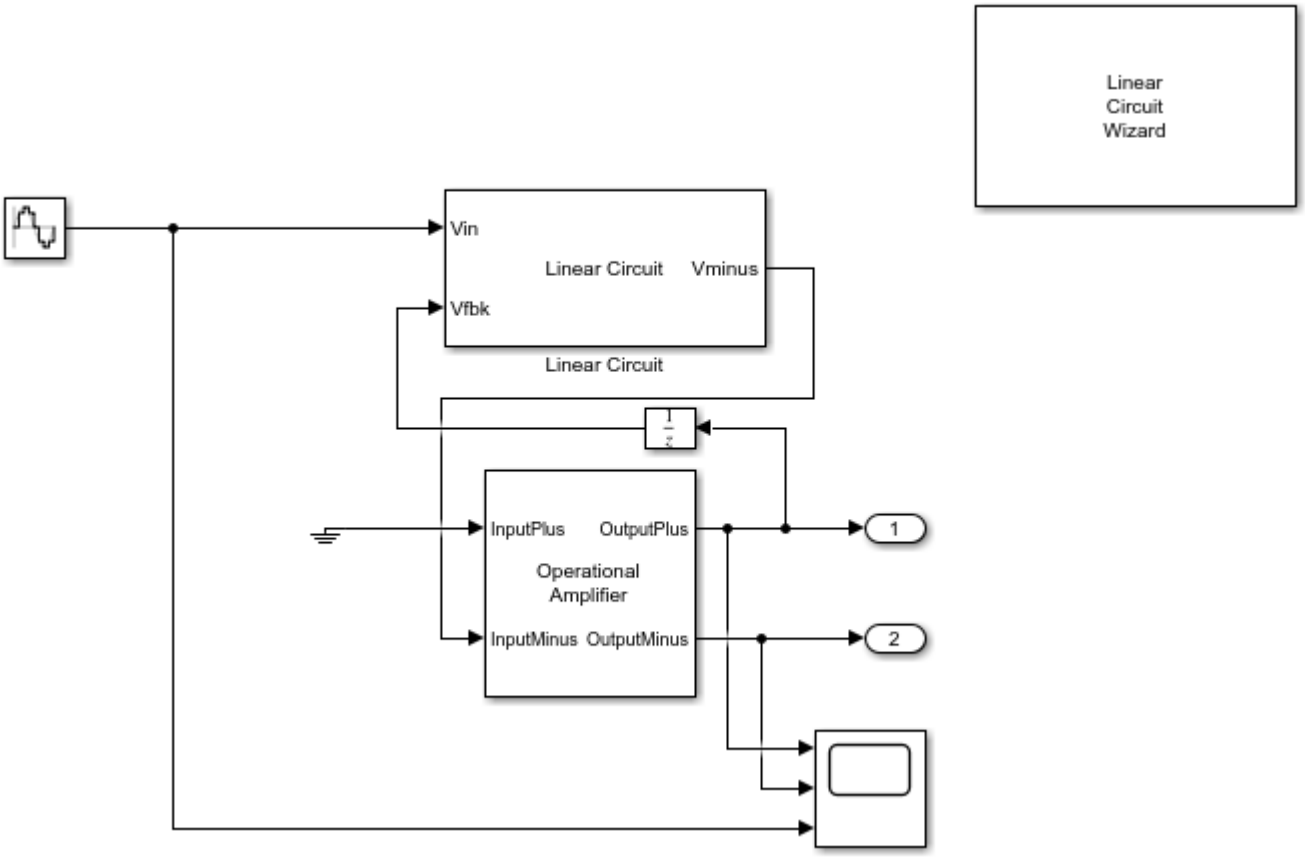
This example shows how to model an inverting amplifier circuit using the Operational Amplifier block and the Linear Circuit wizard block.

Inverting Amplifier

The inverting amplifier circuit contains an op-amp and a feedback network consisting of passive resistors. In this example, the Linear Circuit Wizard block is used to model the feedback network. The Linear Circuit Wizard uses a SPICE netlist describing the feedback stage from the output of the op-amp to its input.

Open the model `Inverting_Amplifier` attached to this example.

```
open_system('Inverting_Amplifier');
```



Operational Amplifier

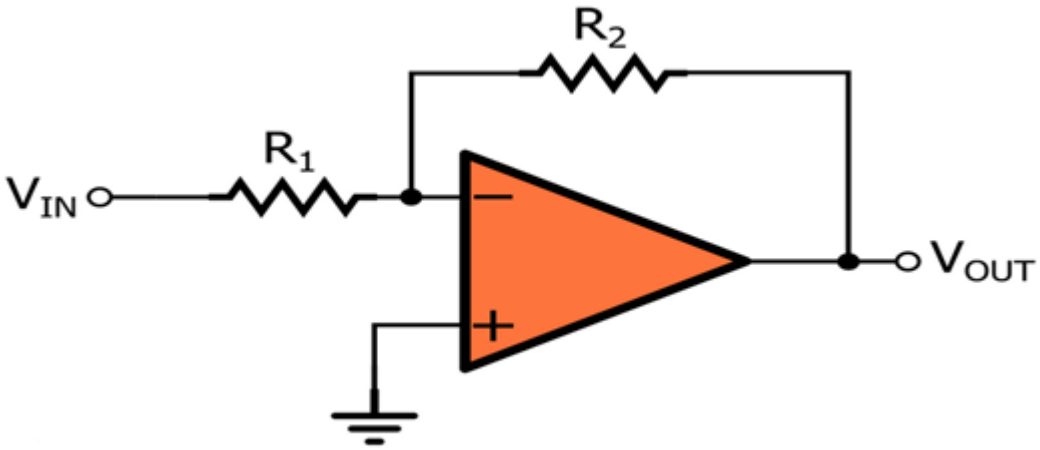
The op-amp used in this example is a double pole circuit defined from the circuit parametes. In this example, the **Input offset voltage (V)** is set to `0`, the **Output resistance (Ohms)** is set to `80`, and the **Open loop gain (v/v)** is set to `855e3`. The **Unity Gain Bandwidth (Hz)** is `1e8` Hz and the **Maximum Tail Current (A)** is `100e-6`.

Linear Circuit Wizard

The Linear Circuit Wizard block uses a SPICE netlist to generate a linear circuit block. You can specify the SPICE netlist file name and click on the **Build/modify block** button to create a circuit block that models the SPICE netlist. The netlist attached to this example is **InvertingAmplifier.sp**.

Operation

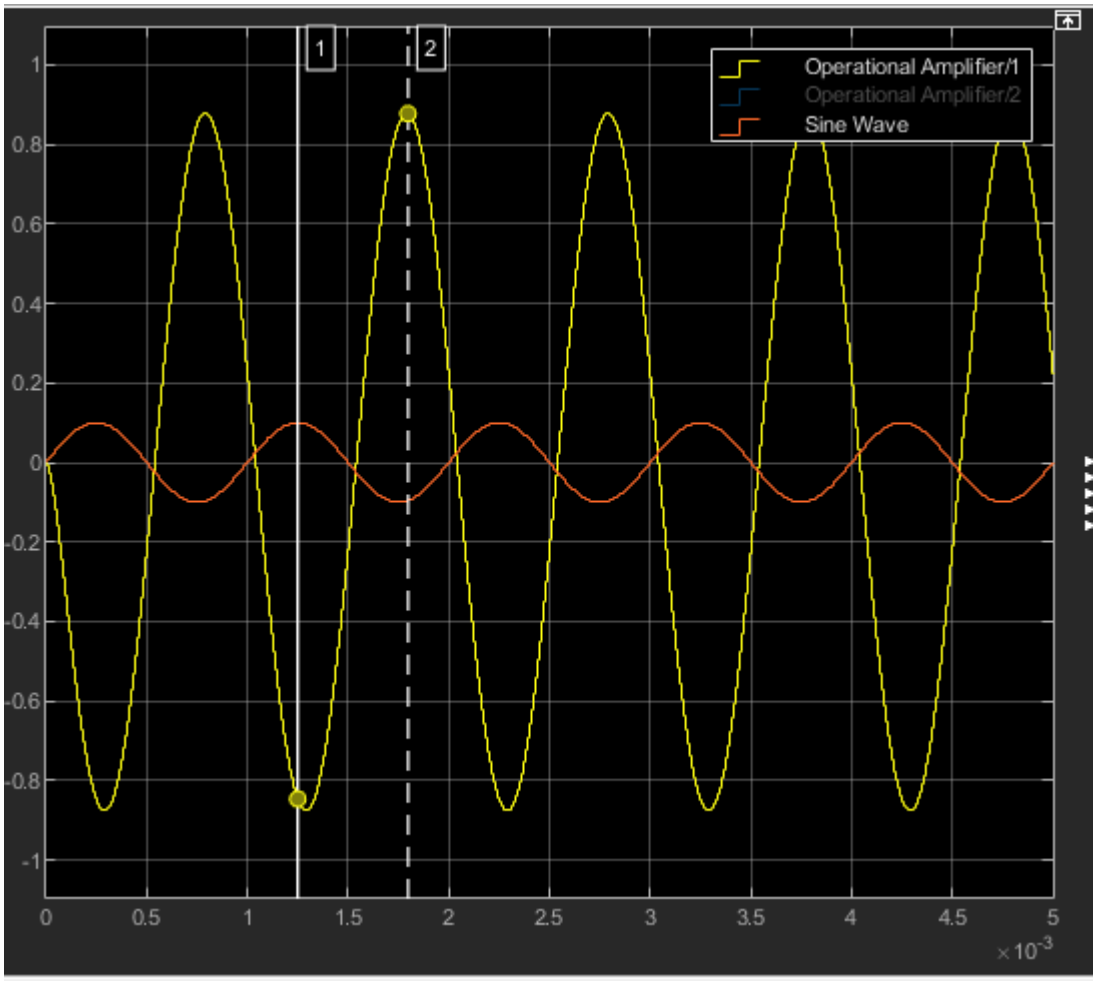
The inverting amplifier circuit is designed with an Op-amp and a feedback network consisting of two resistors R2 and R1.



The feedback from Op-amp output to the input terminal is modeled in the attached SPICE file (R2 = 20KOhms, R1 = 2KOhms).

The closed loop gain of the inverting amplifier is -R2/R1, which is equal to -10. The input stimulus applied to this model has a sample rate of `1e-8`. The same sample rate is used in the operational amplifier.

An input signal of 0.1V sine wave with 1KHz frequency is provided to the circuit model. The input and corresponding output are seen on the scope.



See Also

Linear Circuit Wizard | Operational Amplifier

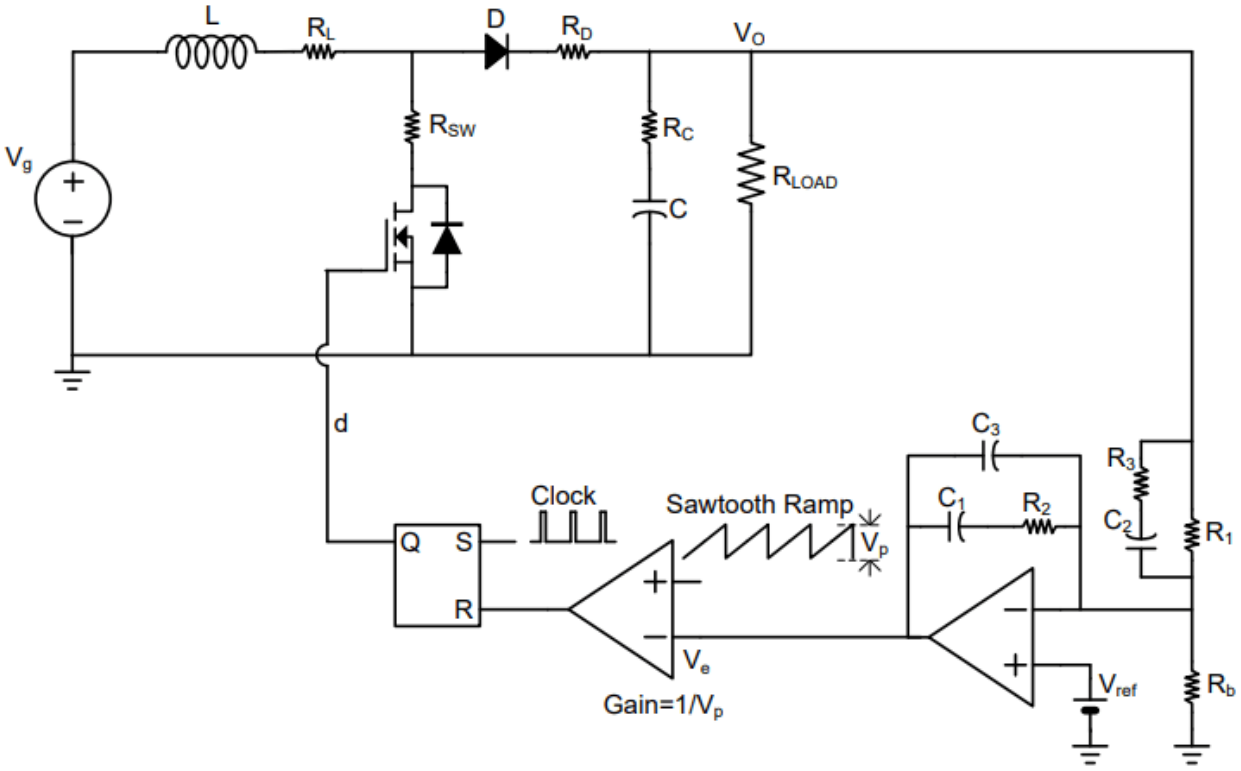
Feedback Amplifier Design for Voltage-Mode Boost Converter

This example shows how to tune the components of a power supply controller to control the output voltage of a boost converter using loop-shape design and fixed-structure tuning methods. This workflow is demonstrated using a boost converter model and a type-III controller.

You need a Mixed Signal Blockset® license to run this example.

Power Train

This example uses the boost converter and feedback amplifier circuit defined in [1].



The power supply system consists of a voltage source, boost power train, load, feedback amplifier and pulse width modulator. The boost power train converts the voltage from the voltage source *Vg* to a voltage *Vo* driving a load *RLoad* in parallel with a load current *ILoad*. This conversion is controlled by the pulse duty cycle `d` at the output of the pulse width modulator, which is in turn controlled by the feedback amplifier. The feedback amplifier senses the output voltage and attempts to make it a fixed multiple of the reference voltage *Vref*, in response to changes in the source voltage, load current, and load resistance.

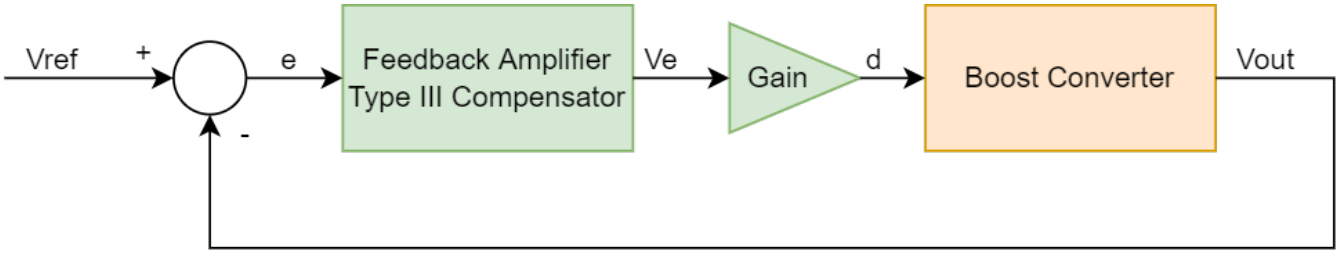
Boost Converter Model

Create a transfer function model for the boost converter with defined component values and at an operating point specified using the duty cycle and output voltage values observed in [2]. Here, use the `getBoostConverterPlant` helper function (available in the file directory for this example) to set up this model. For more information, see “Design Controller for Power Electronics Model Using Frequency Response Data” (Simulink Control Design).

```
boostConverterPlant = getBoostConverterPlant();
boostConverterPlant.InputName = {'d'};
boostConverterPlant.OutputName = {'Vo'};
```

Tunable Control Architecture with Type-III Compensator

Now, set up a closed-loop system for the boost converter using a type-III compensator as a feedback amplifier and a simple gain block as the modulator gain, as shown in the following architecture diagram.



The objective is to tune the resistance (`R1`, `R2`, `R3`) and capacitance (`C1`, `C2`, `C3`) values of the type-III compensator which amplifies the error between the reference voltage and output voltage. A type-III compensator can increase phase in the mid-frequency range while improving both low and high frequency responses.

Create Tunable Linear System for Type-III Compensator

The type-III compensator structure is defined in the netlist file `TypeIII_simple.sp`. Run the `getControlModel` helper function, which uses a Linear Circuit Wizard block to parse the file and extract a tunable linear model. Configure the compensator using the `configureTunableBlock` helper function, and use a tunable gain in series to get the final `controlBlock`.

```
modelName = 'TypeIIICompensator';
load_system(modelName);
lcwBlock = [modelName,'/Linear Circuit Wizard'];

compensator = getControlModel(lcwBlock,{'R1','R2','R3','C1','C2','C3'});
compensator = configureTunableBlock(compensator);

K = realp('K',-1);
K.Minimum = -1.2;
K.Maximum = 0;

controlBlock = K*compensator;
controlBlock.InputName = {'e'};
controlBlock.OutputName = {'d'};
```

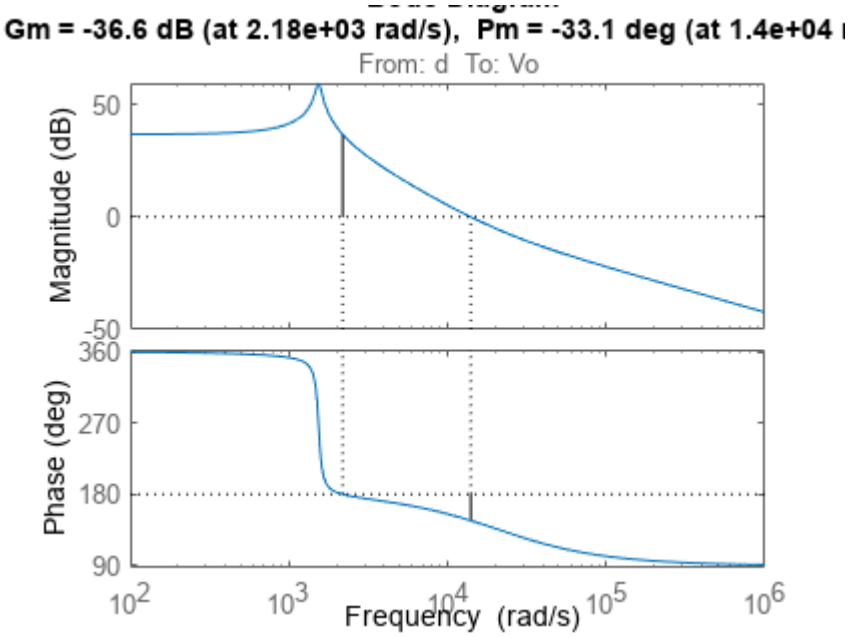
Connect Tunable and Fixed Blocks to Create Closed Loop System

```
eSum = sumblk('e = Vref - VoMeasured');
dSum = sumblk('VoMeasured = dVo + Vo');

closedLoopSystem = connect(controlBlock,boostConverterPlant,eSum,dSum,{'Vref','dVo'},'VoMeasured
```

Loop-Shaping Design

First, view the stability margins and frequency response of the open-loop boost converter plant.

```
figure;
margin(boostConverterPlant);
```



Design Requirements

Use the following design requirements to define a stable target loop shape:

- Increase gain at low frequencies. This improves the transient performance of reference voltage and output voltage disturbance rejection steady state error.
- Add phase in middle frequencies to increase bandwidth (reduce response time) and achieve stability with positive gain and phase margins.
- Reduce gain at high frequencies to make the closed loop system robust by attenuating oscillations in the control signal created by introduced harmonics and noisy measurements in the output voltage.

Use the helper function `getTuningGoals` to create the following tuning goal objects, which help you define constraints and objectives when you tune the system.

- `marginsGoal` - `TuningGoal.Margins` (Control System Toolbox) with a gain margin constraint of 5 dB and phase margin constraint of 30 degrees. These constraints enforce stability and also help you visualize, using `viewGoal`, how much uncertainty the loop can tolerate at different frequencies before going unstable.
- `minLoopGainGoal` - `TuningGoal.MinLoopGain` (Control System Toolbox) with an integral action gain profile and a bandwidth of 1 KHz. The constraint is enforced in the low frequency range (10 Hz to 200 Hz).
- `maxLoopGainGoal` - `TuningGoal.MaxLoopGain` (Control System Toolbox) with a double integrator gain profile and a roll-off of -40 dB/decade, enforced in the high frequency range (1.5 KHz to 15 KHz).

```
[marginsGoal, minLoopGainGoal, maxLoopGainGoal] = getTuningGoals();
```

Tune System

Now, use `systune` to tune the system, that is, compute the values of the tunable parameters (`C1`, `C2`, `C3`, `R1`, `R2`, `R3`, and `K`) such that the open loop response meets the desired design requirements.

First, create a `systuneOptions` object and adjust the values for the minimum decay rate and maximum spectral radius to suit tuning for high bandwidth loop shapes. Also, reduce the relative tolerance criteria for termination.

```
opts = systuneOptions;
opts.MinDecay = 1e-15;
opts.MaxRadius = 1e15;
opts.SoftTol = 1e-10;
```

Tune the system using `systune`, with the margin goals enforced as objectives (soft goals) and minimum and maximum loop gain goals enforces as constraints (hard goals). The tuning returns a converged result that satisfies the hard goals and optimizes the soft goals. The best achieved values for the soft and hard goals are both less than 1.

```
tunedClosedLoopSystem = systune(closedLoopSystem,marginsGoal,[minLoopGainGoal,maxLoopGainGoal],op
```

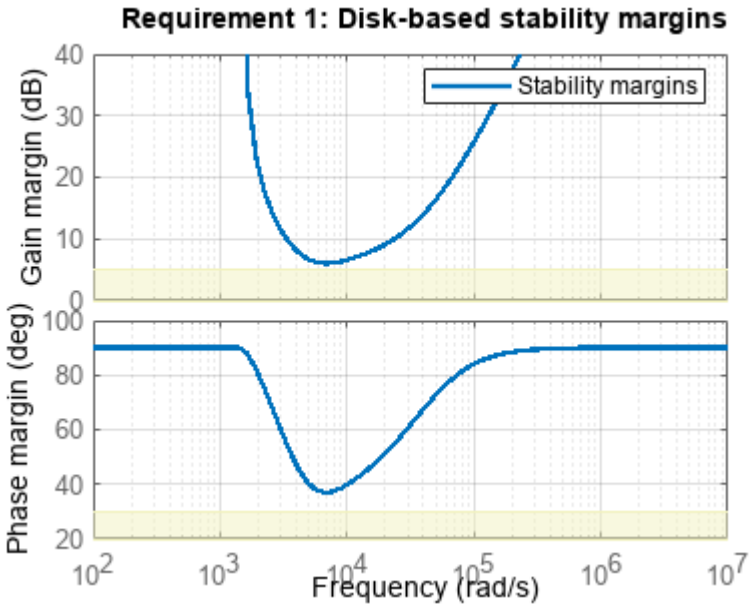
```
Final: Soft = 0.838, Hard = 0.99841, Iterations = 1232
```

Analyze Results

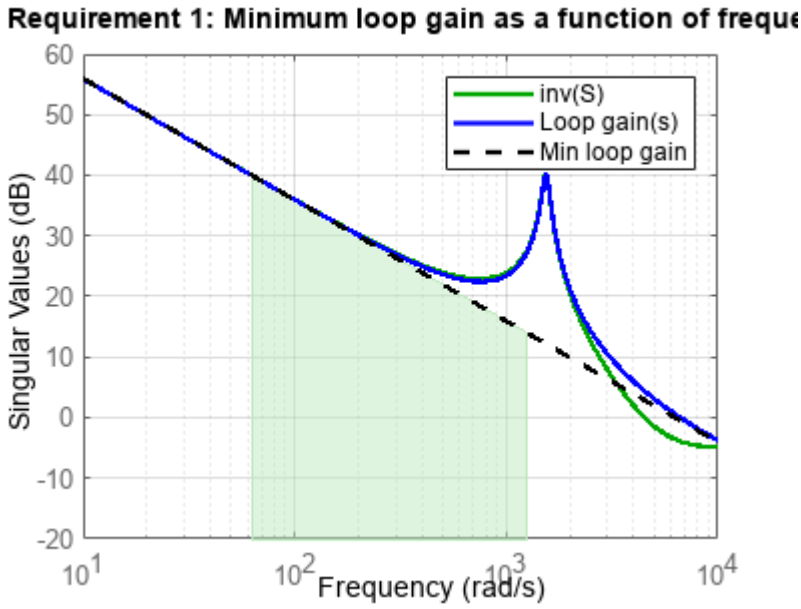
View Tuned System Results Against Desired Specifications

Create plots of the tuned system against the tuning goals to indicate how closely it meets the desired specifications. The shaded regions in each plot represent where the tuning goal is violated. The plots show that the constraints of tracking error and stability margins are met, while the objectives of minimum and maximum loop gains in specific frequency ranges are optimized.

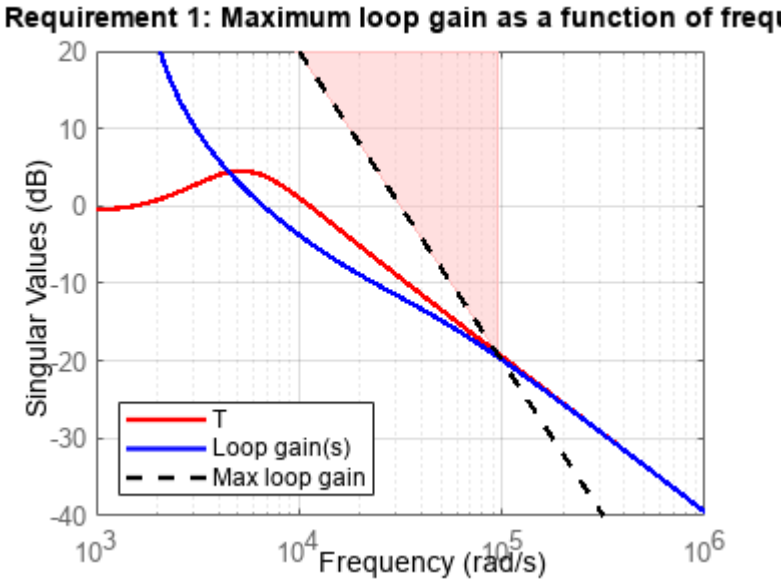
```
viewGoal(marginsGoal,tunedClosedLoopSystem);
```




```
viewGoal(minLoopGainGoal,tunedClosedLoopSystem);
```




```
viewGoal(maxLoopGainGoal,tunedClosedLoopSystem);
```

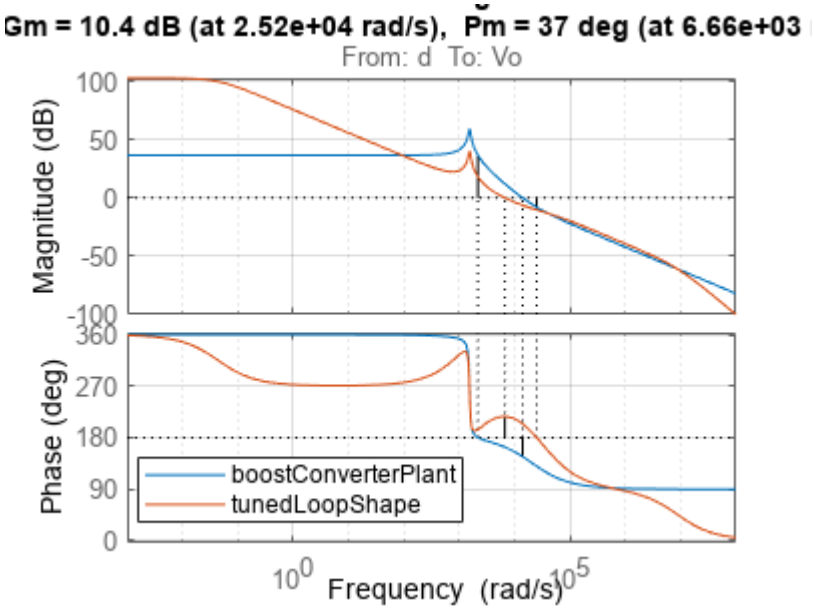


View Tuned Loop Shape and Stability Margins

The margin plot of the tuned open loop system indicates the bandwidth and robustness along with the achieved gain and phase margins.

```
tunedLoopShape = getLoopTransfer(tunedClosedLoopSystem,'d',-1);
figure;
margin(boostConverterPlant);
hold on;
margin(tunedLoopShape);
```

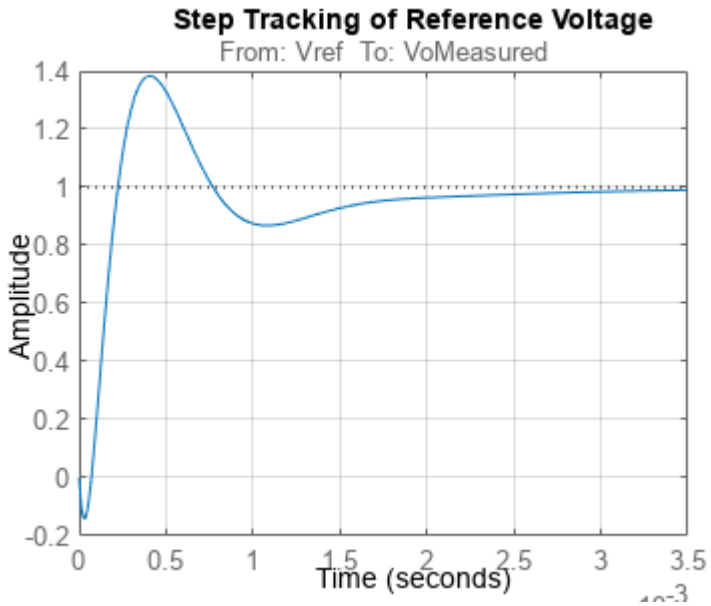
```
grid on;
legend('show','Location','best');
```



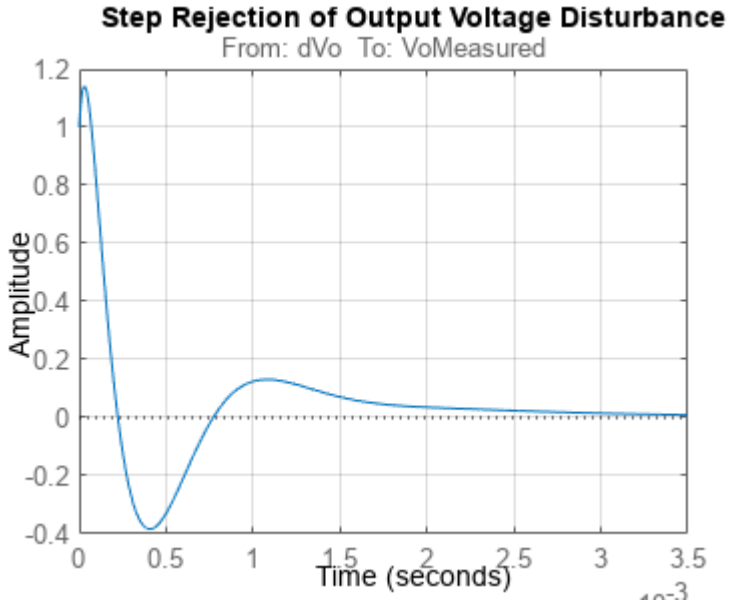
View Step Responses of Closed Loop System

Step response plots of reference voltage tracking and output voltage disturbance tracking show that the response settles without any steady state error with an acceptably small settling time.

```
figure;
step(getIOTransfer(tunedClosedLoopSystem,'Vref','VoMeasured'));
title('Step Tracking of Reference Voltage');
grid on;
```



```
figure;
step(getIOTransfer(tunedClosedLoopSystem,'dVo','VoMeasured'));
title('Step Rejection of Output Voltage Disturbance');
grid on;
```



Tuned Component Values

Get the transfer function from the controller input `e` to the duty cycle `d` and display the tuned component values.

```
tunedCompensator = getIOTransfer(tunedClosedLoopSystem,'e','d','d');
showTunable(tunedCompensator);
```

```
C1 = 5.62e-08
-----------------------------------
C2 = 2.56e-09
-----------------------------------
C3 = 1e-11
-----------------------------------
K = -1.2
-----------------------------------
R1 = 2.37e+05
-----------------------------------
R2 = 1.12e+04
-----------------------------------
R3 = 1e+04
```

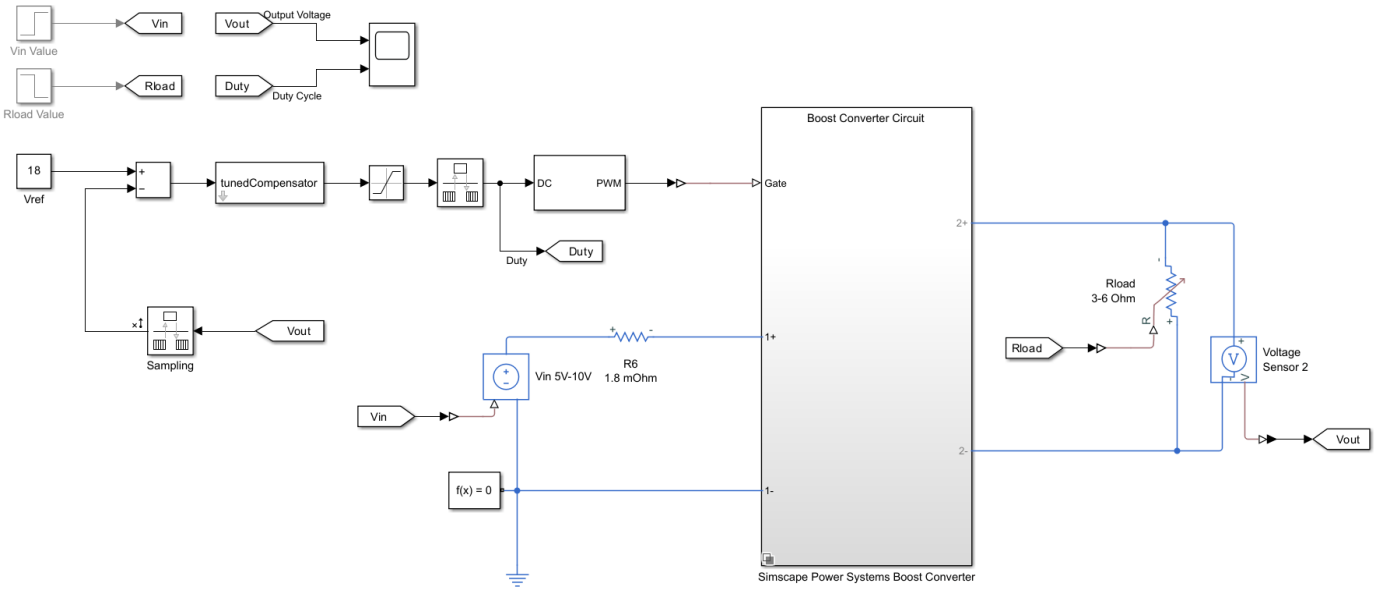
Simulation Results

Simulation Result with Tuned Controller

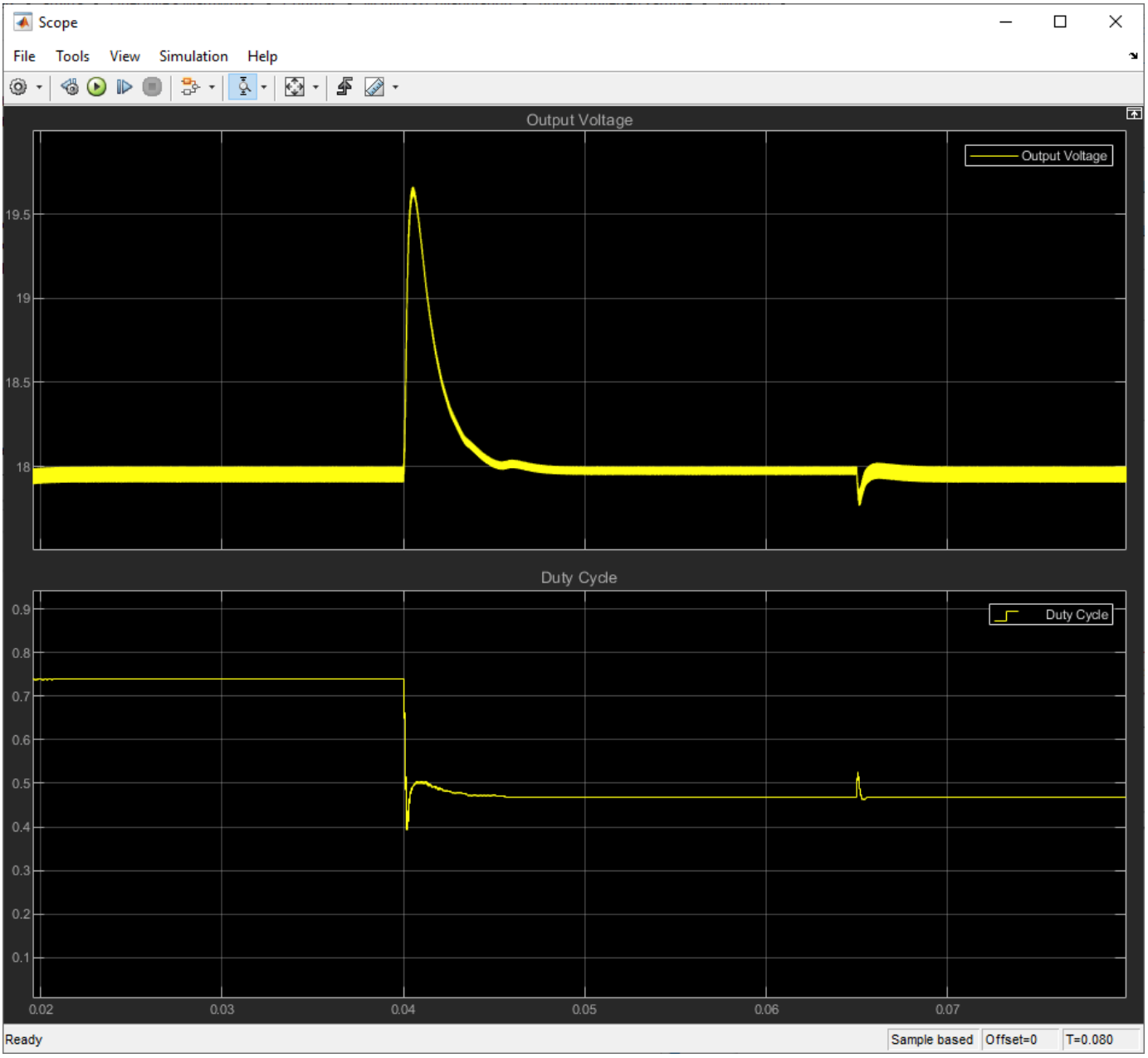
```
open_system("VoltageControlledBoostConverter.slx");
```

Use an LTI System Block to implement the type-III compensator and simulate the model to examine the performance. The model uses the following disturbances:

- Line disturbance at t = 0.04 seconds, which increases the input voltage `Vin` from 5 V to 10 V.
- Load disturbance at t = 0.065 seconds, which increases the load resistance `RLoad` from 3 ohms to 6 ohms.



The results show that the tuned feedback amplifier rejects the line and load disturbances well.



References

[1] Lee, S. W. "Practical Feedback Loop Analysis for Voltage-Mode Boost Converter." Application Report No. SLVA633. Texas Instruments, January 2014. www.ti.com/lit/an/slva633/slva633.pdf.

[2] Ahmadi, Reza, and Mehdi Ferdowsi, "Modeling Closed-Loop Input and Output Impedances of DC-DC Power Converters Operating inside DC Distribution Systems." *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014*. IEEE, March 2014. https://ieeexplore.ieee.org/document/6803449.

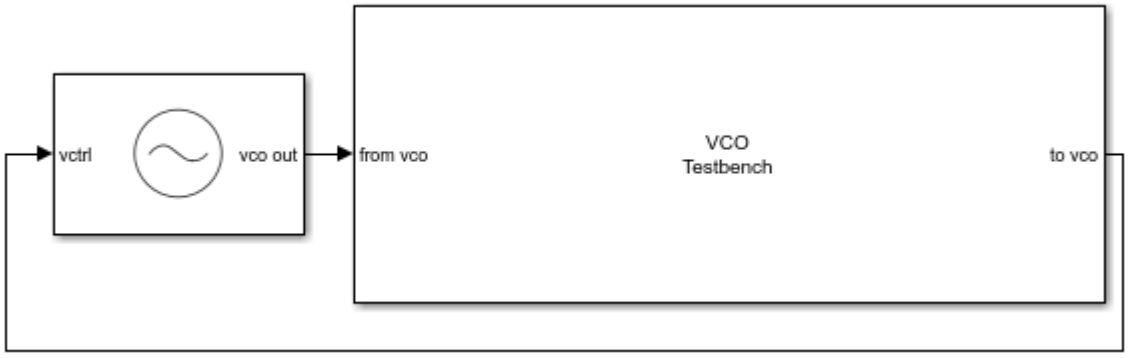
PLL Block Level Examples

Measuring VCO Phase Noise to Compare with Target Profile

This example shows how to validate the phase noise profile of a VCO device under test (DUT) using a VCO Testbench.

Open the model `vcoPhaseNoise` . The model consists of a VCO block and a VCO Testbench.

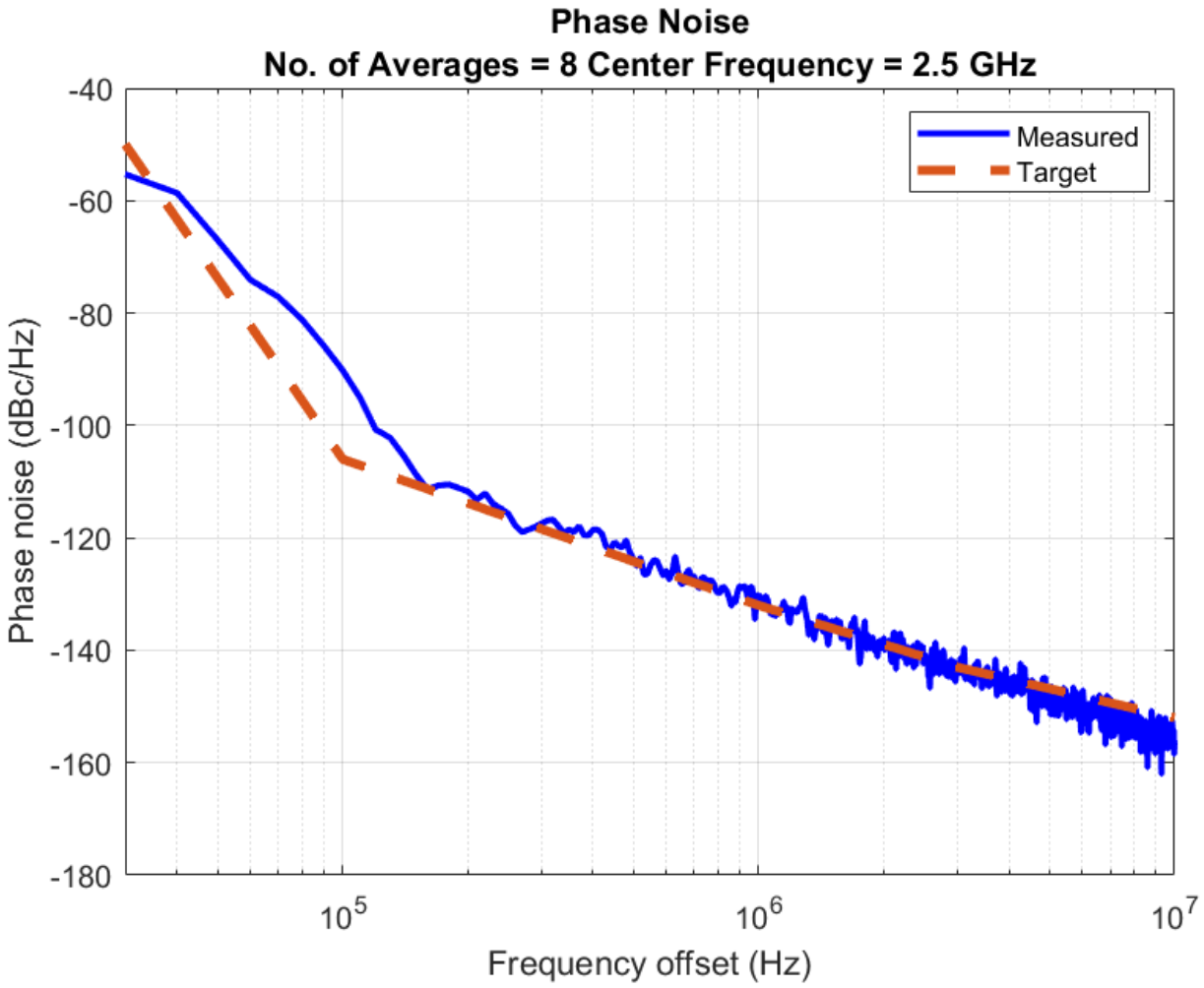
```
open_system('vcoPhaseNoise.slx')
```



The voltage sensitivity of the VCO is set to `1.25e6` Hz, and the free running frequency is `2e9` Hz.

The testbencch is set to measure the **Phase noise** metric of the VCO in the **Measurement** option. The `Control voltage` provided to the input of VCO is `4` V. So, the VCO operates at `2.5` GHz frequency. Click the **Autofill setup parameters** button to automatically calculate the **Resolution bandwidth (Hz)**, and **No. of spectral averages**.

Run simulation for `8e-4` s, as recommended in the **Block Parameters** dialog box. Double click the VCO Testbench block to open the **Block Parameters** dialog box. Click on the **Plot measurement** button.



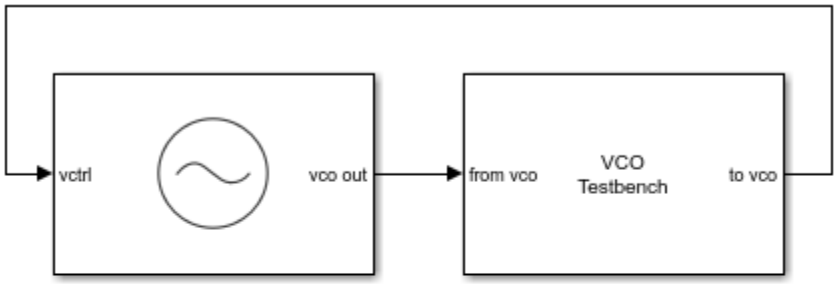
The operating frequency matches the predicted frequency `2.5` GHz. The measured phase noise profile also matches the target profile.

Finding Voltage Sensitivity and Quiescent Frequency of VCO

This example shows how to find VCO metrics such as voltage sensitivity (Kvco) and quiescent frequency or free running frequency (Fo).

Open the model `vcoCharacteristics`. The model consists of a VCO block and a VCO Testbench.

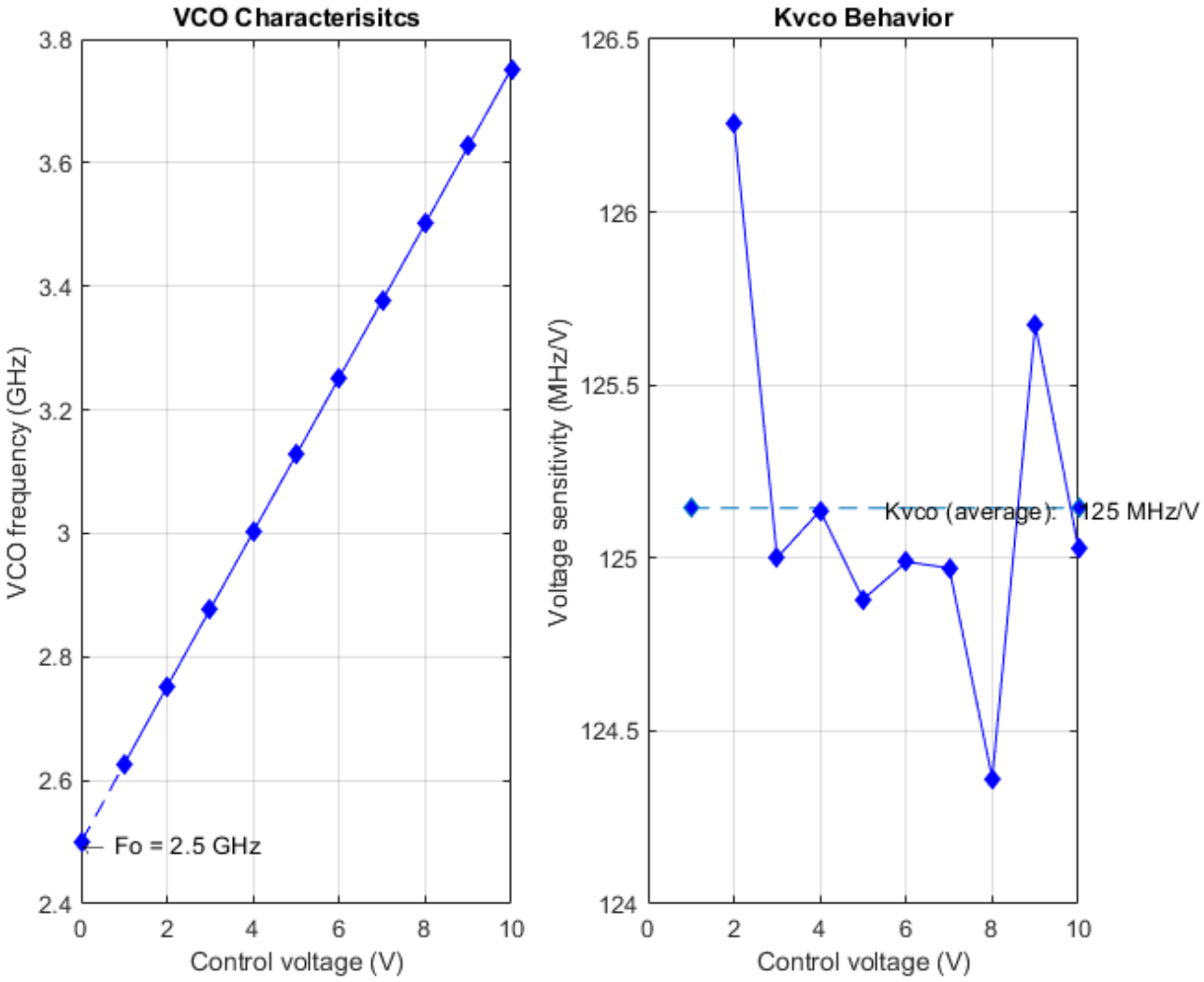
```
open_system('vcoCharacteristics.slx')
```



The testbench is set to measure the **Kvco and Fo** metric of the VCO in **Measurement** option. **Range of control voltage (V)** provided to the input of VCO is set to `[1 10]`.

Run the model for `1.2e-3` s. Double click the VCO Testbench to open the **Block Parameters** dialog box. Click on the **Plot measurement** button.

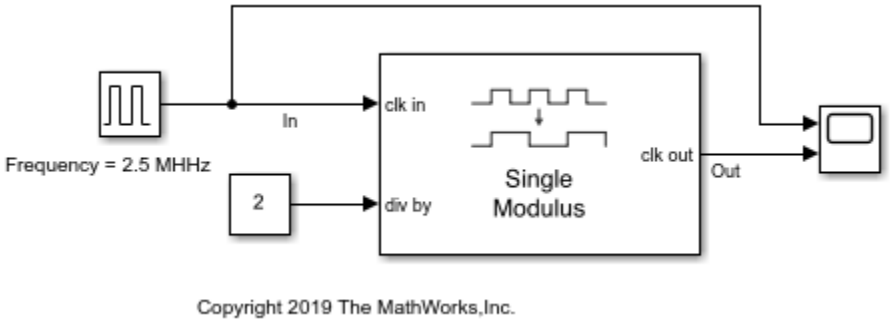
From the simulation, the free running frequency is `2.5` GHz, and voltage sensitivity is `125` MHz/V.


VCO Characterisitcs
VCO frequency (GHz)
Control voltage (V)
Fo = 2.5 GHz
Kvco Behavior
Voltage sensitivity (MHz/V)
Control voltage (V)
Kvco (average):

Frequency Division Using Single Modulus Prescaler

Open the model `singleModulusPrescaler`. The model consists of a Pulse Generator and a Single Modulus Prescaler block.

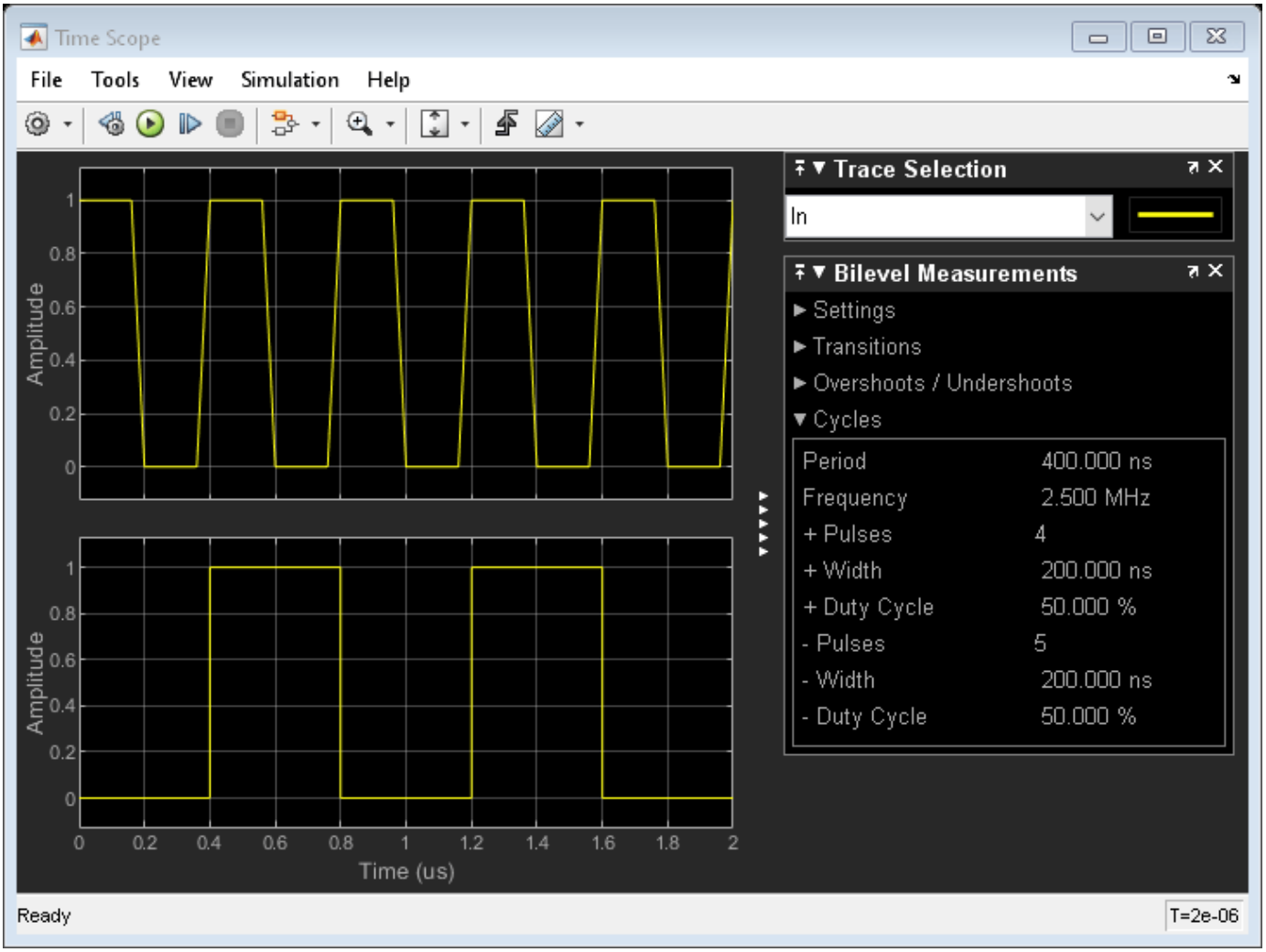
```
model='singleModulusPrescaler';
open_system(model)
```



The period of the incoming pulse at the **clk in** port is `4e-7` s. So, the incoming signal has a frequency of `2.5` MHz. The div-by value is set at `2`.

Run the simulation for `2e-6` s. The frequency of the output signal is `1.25` MHz.

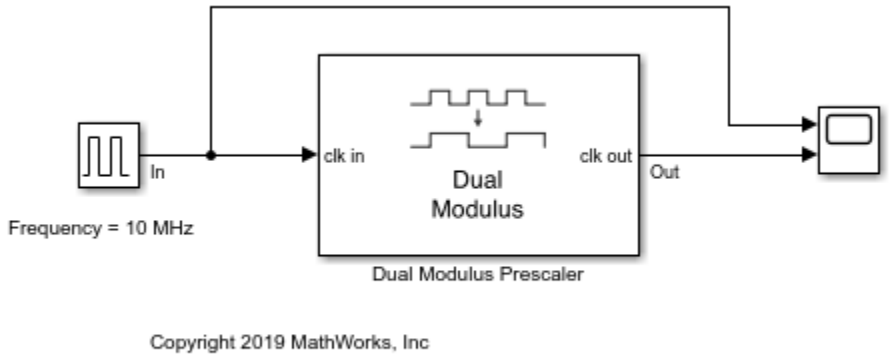
```
sim(model);
```


Time Scope
File
Tools
View
Simulation
Help
Trace Selection
In
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 400.000 ns
Frequency 2.500 MHz
+ Pulses 4
+ Width 200.000 ns
+ Duty Cycle 50.000 %
- Pulses 5
- Width 200.000 ns
- Duty Cycle 50.000 %
Amplitude
Time (us)
Ready
T=2e-06

Frequency Division Using Dual Modulus Prescaler

Open the model `Dual_Modulus_Prescaler_Ex`. The model consists of a Pulse Generator and a Dual Modulus Prescaler block.

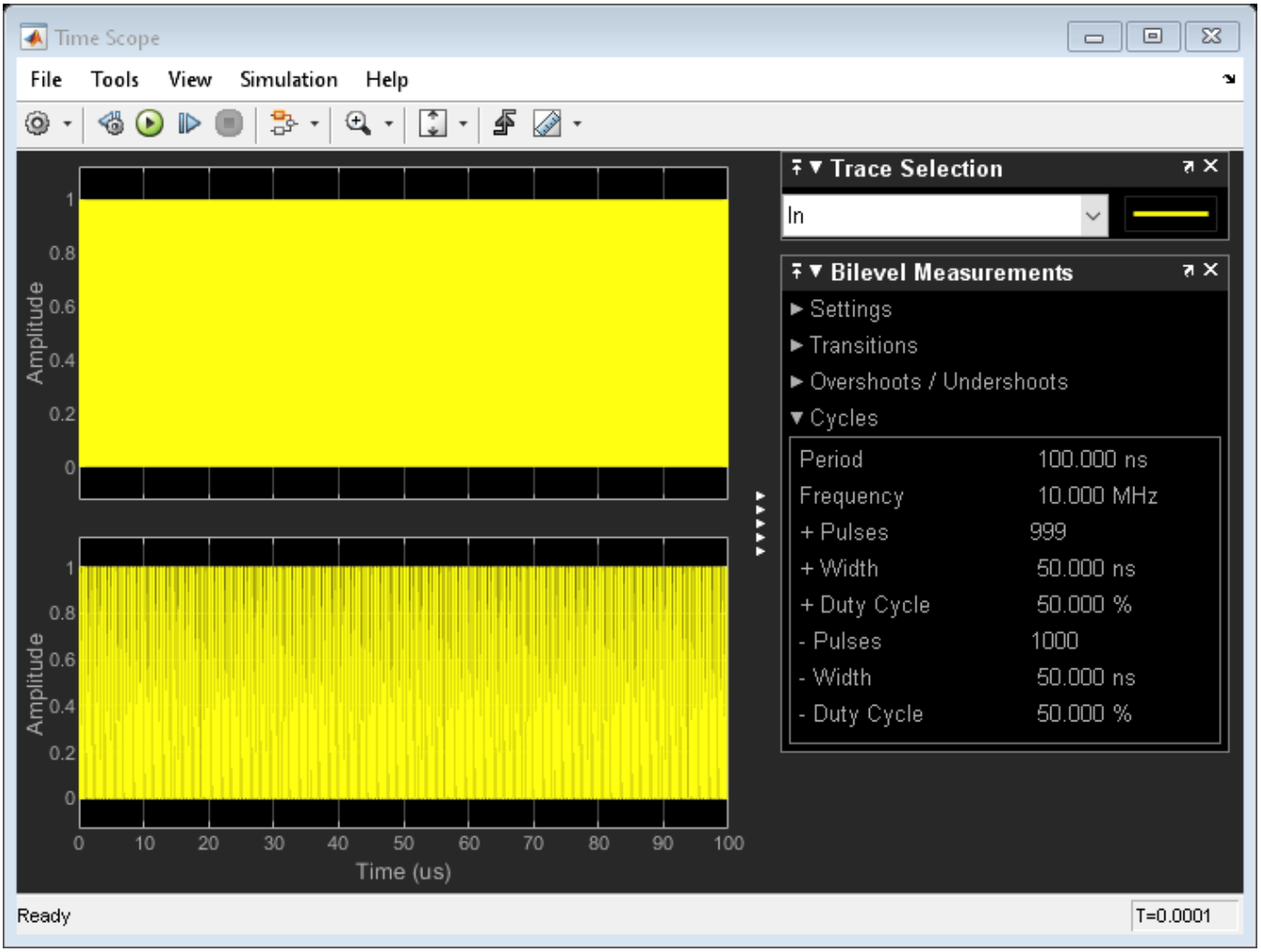
```
model='Dual_Modulus_Prescaler_Ex';
open_system(model)
```



The period of the incoming pulse at the **clk in** port is `1e-7` s. So, the incoming signal has a frequency of `10` MHz. The **Program counter value**, **Prescaler divider value**, and **Swallow counter value** are `4`, `1`, and `1`, respectively. The effective clock divider value of the dual modulus prescaler is `5`.

Run the simulation for `1e-4` s. The frequency of the output signal is `2.002` MHz.

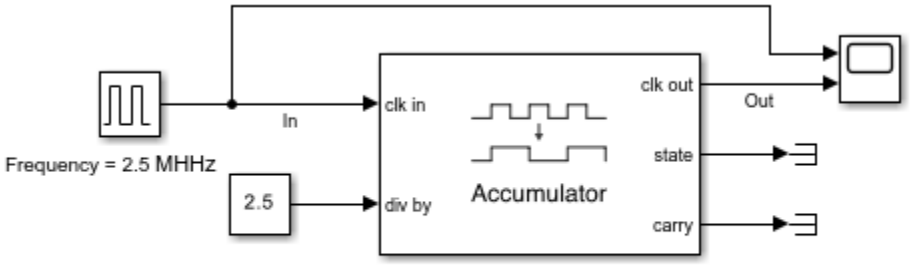
```
sim(model);
```


Time Scope
File
Tools
View
Simulation
Help
Trace Selection
In
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 100.000 ns
Frequency 10.000 MHz
+ Pulses 999
+ Width 50.000 ns
+ Duty Cycle 50.000 %
- Pulses 1000
- Width 50.000 ns
- Duty Cycle 50.000 %
Amplitude
Time (us)
Ready
T=0.0001

Frequency Division Using Fractional Clock Divider with Accumulator

Open the model `fractionalClockDivider_w_Accumulator`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

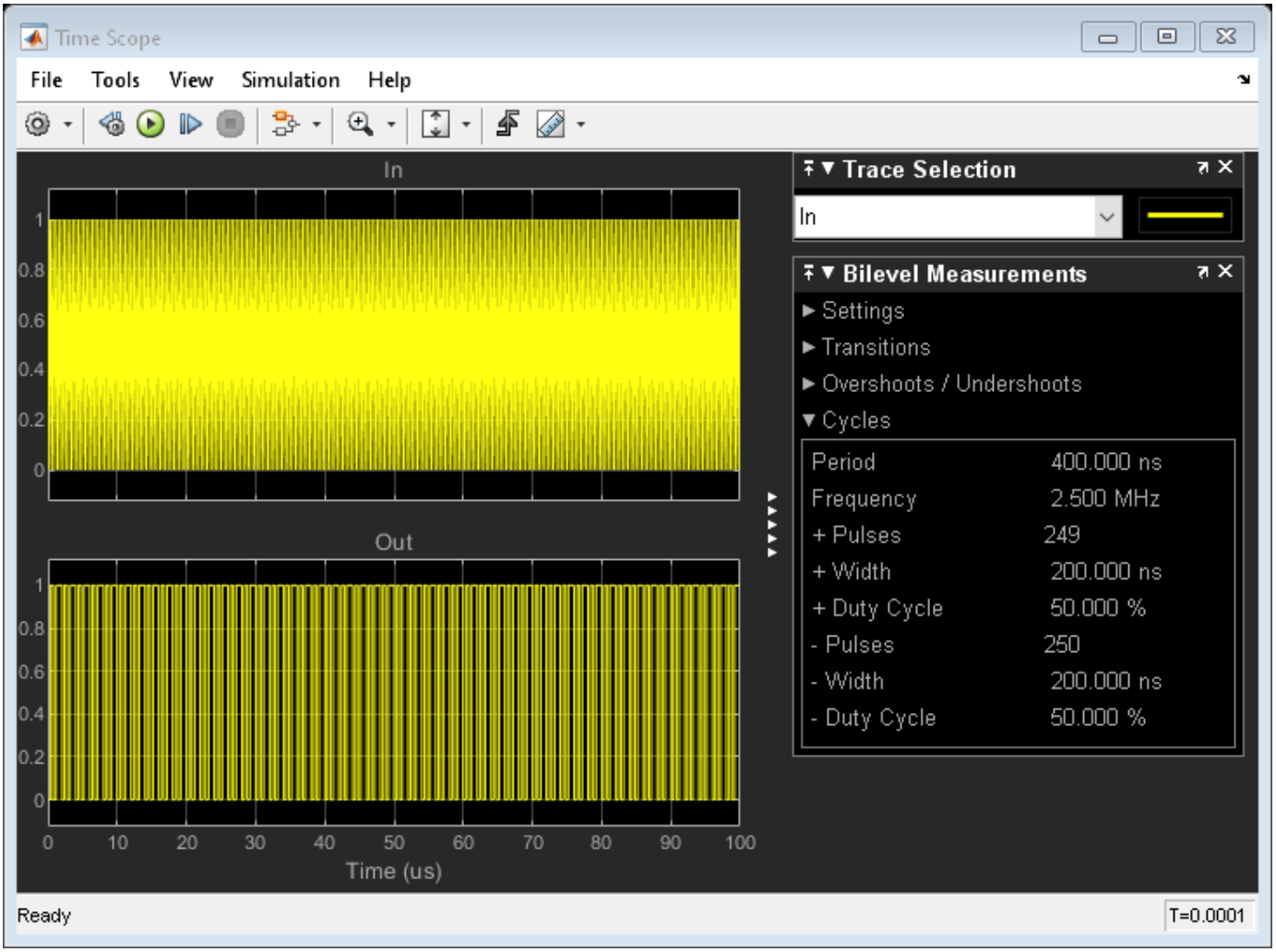
```
model='fractionalClockDivider_w_Accumulator';
open_system(model)
```



The period of the incoming pulse at the **clk in** port is `4e-7` s. So, the incoming signal has a frequency of `2.5` MHz. The div-by value is set at `2.5`.

Run the simulation for `1e-4` s. The frequency of the output signal is `1.002` MHz.

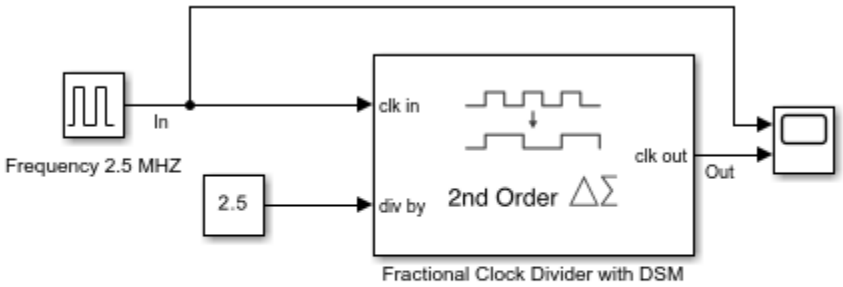
```
sim(model);
```


Time Scope
File
Tools
View
Simulation
Help
In
Out
Time (us)
Trace Selection
In
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 400.000 ns
Frequency 2.500 MHz
+ Pulses 249
+ Width 200.000 ns
+ Duty Cycle 50.000 %
- Pulses 250
- Width 200.000 ns
- Duty Cycle 50.000 %
Ready
T=0.0001

Frequency Division Using Fractional Clock Divider with DSM

Open the model `fractionalClockDivider_w_DSM`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

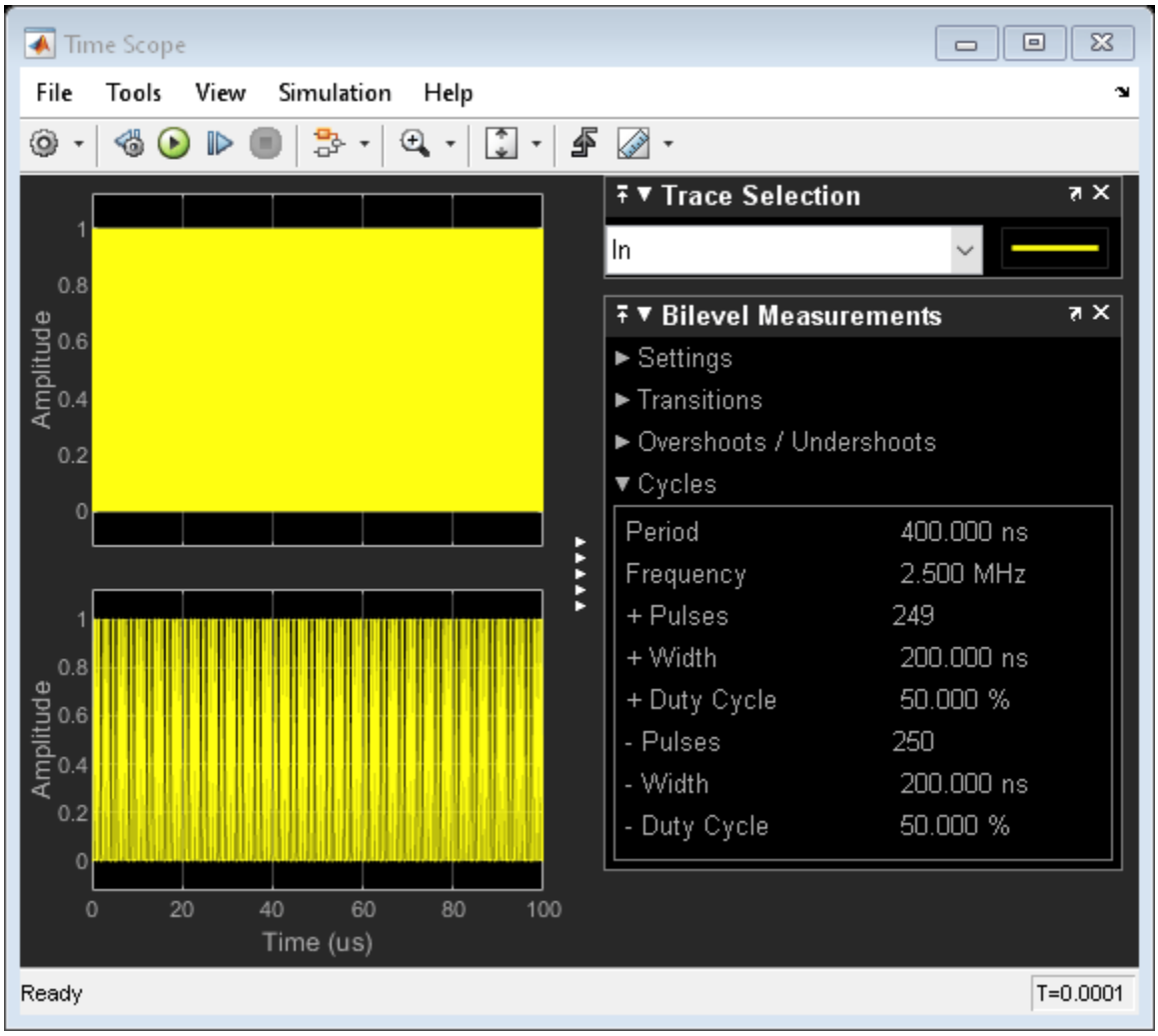
```
model='fractionalClockDivider_w_DSM';
open_system(model)
```



The period of the incoming pulse at the **clk in** port is `4e-7` s. So, the incoming signal has a frequency of `2.5` MHz. The div-by value is set at `2.5`. The clock divider uses a second order delta sigma modulator.

Run the simulation for `1e-4` s. The frequency of the output signal is `1.002` MHz.

```
sim(model);
```


Time Scope
File
Tools
View
Simulation
Help
Trace Selection
In
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 400.000 ns
Frequency 2.500 MHz
+ Pulses 249
+ Width 200.000 ns
+ Duty Cycle 50.000 %
- Pulses 250
- Width 200.000 ns
- Duty Cycle 50.000 %
Amplitude
Time (us)
Ready
T=0.0001

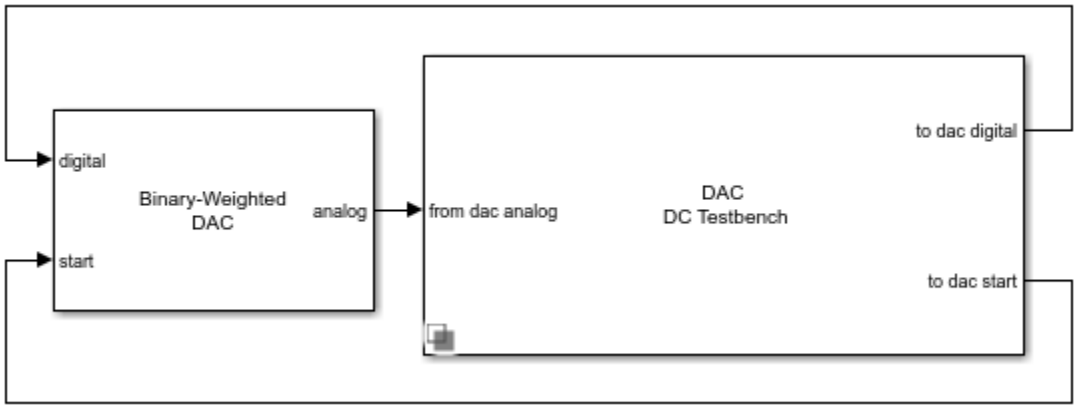
Data Converter Block Level Examples

Measure Offset and Gain Error of Binary Weighted DAC

This example shows how to find the offset and gain errors of a binary weighted DAC block.

Open the model `dac_dc_error`. The model consists of a Binary Weighted DAC block and a DAC Testbench.

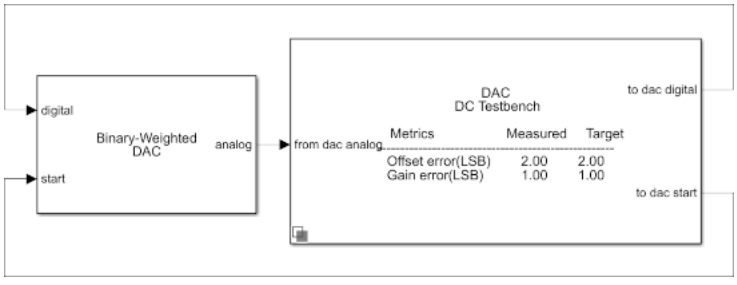
```
model = 'dac_dc_error';
open_system(model)
```



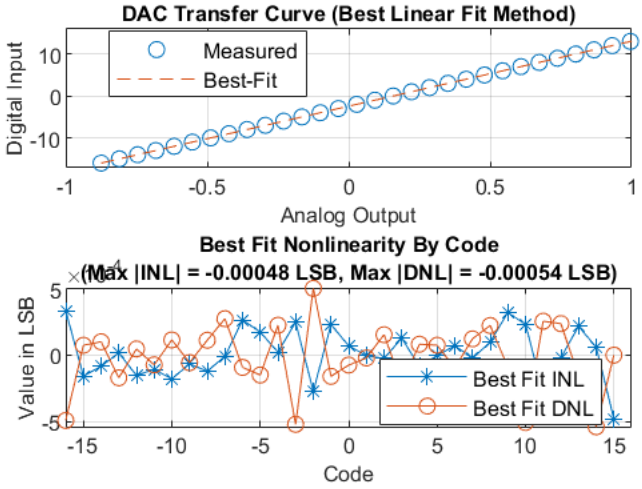
The parameters under the **General** tab of the Binary Weighted DAC are kept at their default values. The impairments are enabled in the **Impairments** tab, and the value of the offset error and gain error are set to `2` LSB and `1` LSB, respectively.

In the DAC Testbench block, the **Measurement** option is set to `DC`. In the **Setup** tab, the **Autofill setup parameter** button is used to automatically propagate the setup parameters from the DAC block. In the **Target Metric** tab, the **Autofill target metric** button is used to automatically set the target offset and gain errors from the DAC block. All other parameters are kept at their default values.

Run the simulation for `7.68e-04` s. The measured offset and gain errors are `2.00` LSB and `1.00` LSB, respectively.



The DAC transfer curve matches the best linear fit method. Using best fit method, maximum INL is `-3.3e-4` LSB and maximum DNL is `4.6e-4` LSB.


DAC Transfer Curve (Best Linear Fit Method)
Measured
Best-Fit
Digital Input
10
0
-10
-1
-0.5
0
0.5
1
Analog Output

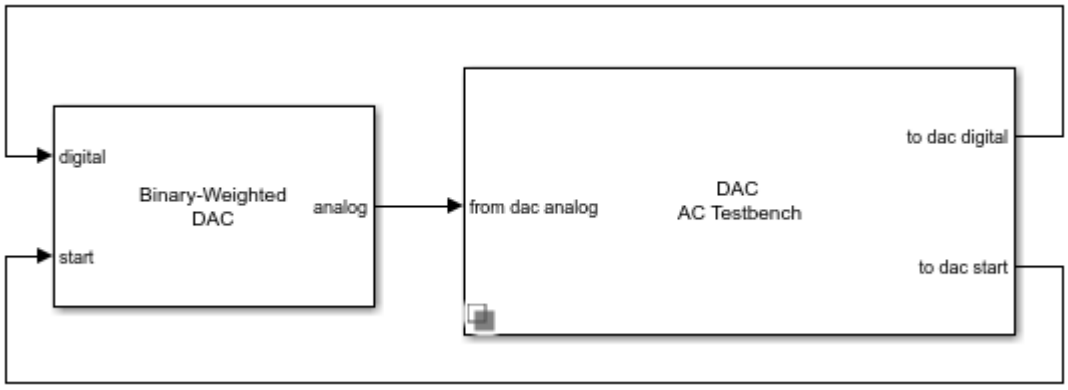
Best Fit Nonlinearity By Code
(Max |INL| = -0.00048 LSB, Max |DNL| = -0.00054 LSB)
×10^-4
5
0
-5
Value in LSB
Best Fit INL
Best Fit DNL
-15
-10
-5
0
5
10
15
Code

Measure AC Performance Metrics of Binary Weighted DAC

This example shows how to find the AC performance metrics such as SNR, SINAD, SFDR, ENOB, noise floor and settling time of a binary weighted DAC block.

Open the model `dac_ac_error`. The model consists of a Binary Weighted DAC block and a DAC Testbench.

```
model = 'dac_ac_error';
open_system(model)
```

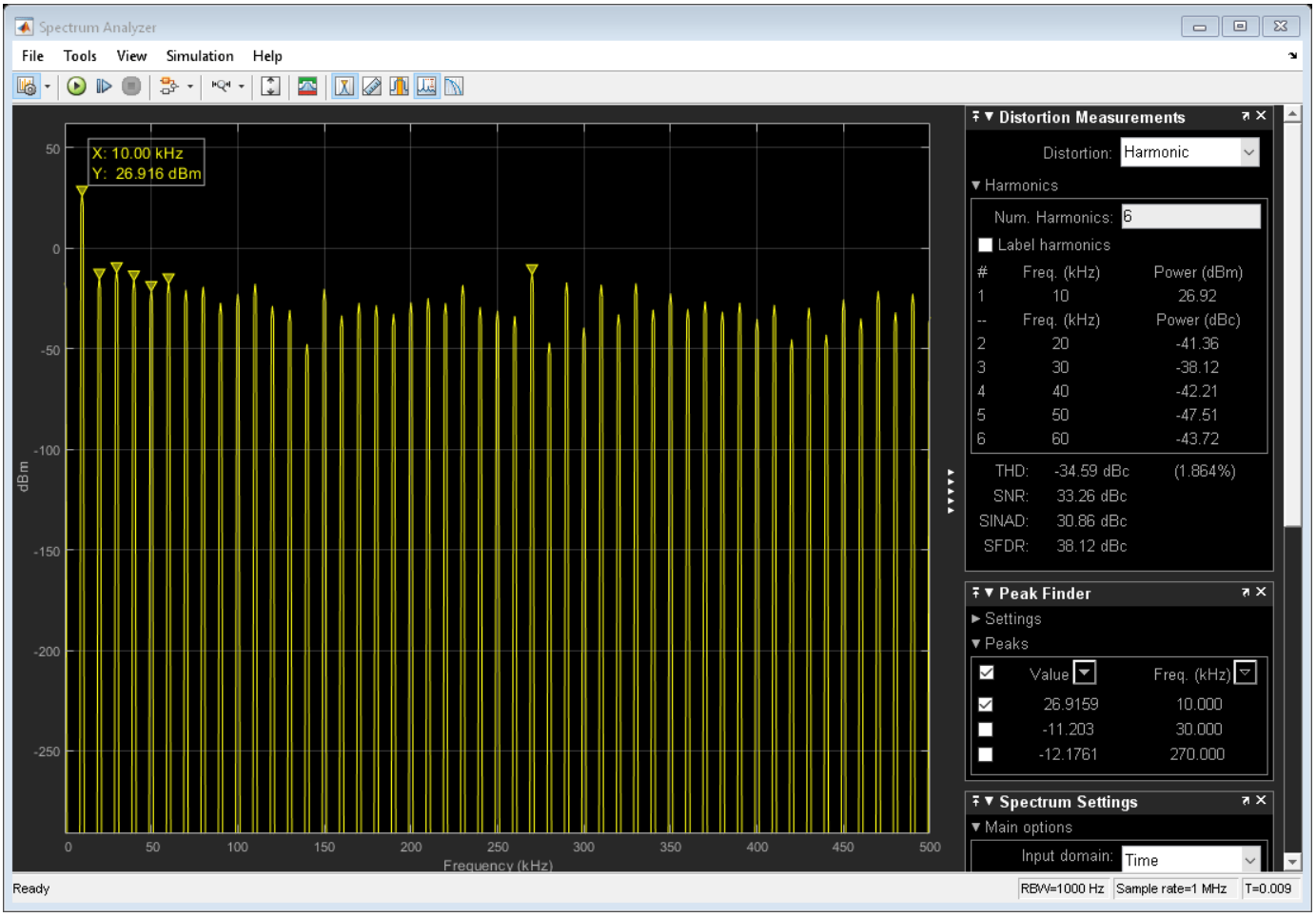


The parameters of the Binary Weighted DAC are kept at their default values.

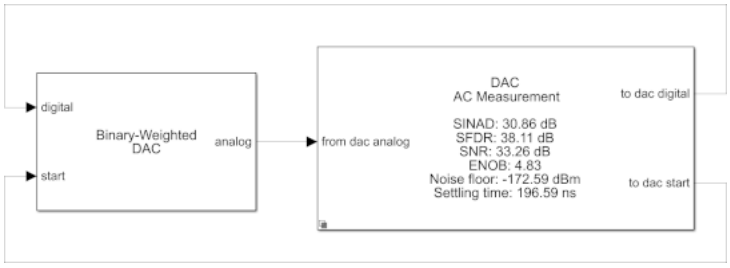
In the DAC Testbench block, the **Measurement** option is set to `AC`. In the **Setup** tab, the **Autofill setup parameter** button is used to automatically propagate the setup parameters from the DAC block. **Show spectrum analyzer during simulation** option is also checked.

Run the simulation for `9e-3` s.

```
sim(model);
```



The measured AC performance metrics are displayed on the DAC Testbench block.

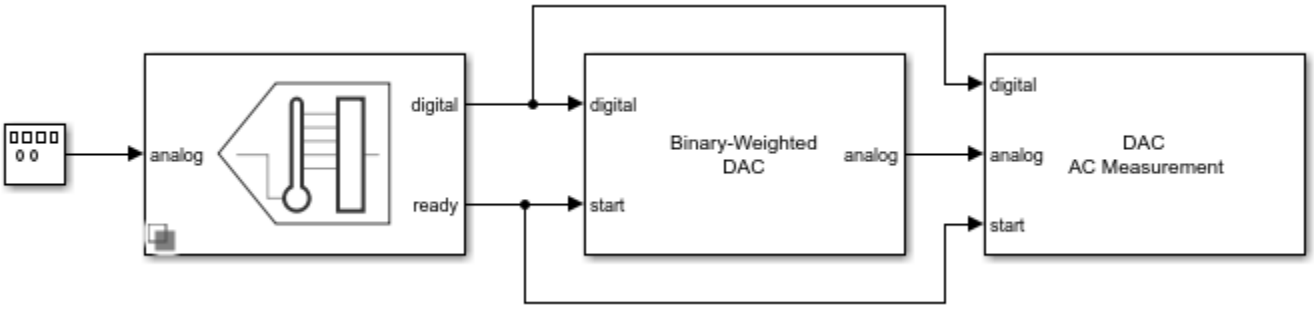


Measure AC Performance Metrics of DAC Using DAC AC Measurement

This example shows how to find the AC performance metrics such as SNR, SINAD, SFDR, ENOB, and noise floor using a DAC AC Measurement block.

Open the model `dac_ac_measure`. The model consists of a Signal Generator, a Flash ADC, a Binary Weighted DAC block, and a DAC AC Measurement block.

```
model = 'dac_ac_measure';
open_system(model)
```



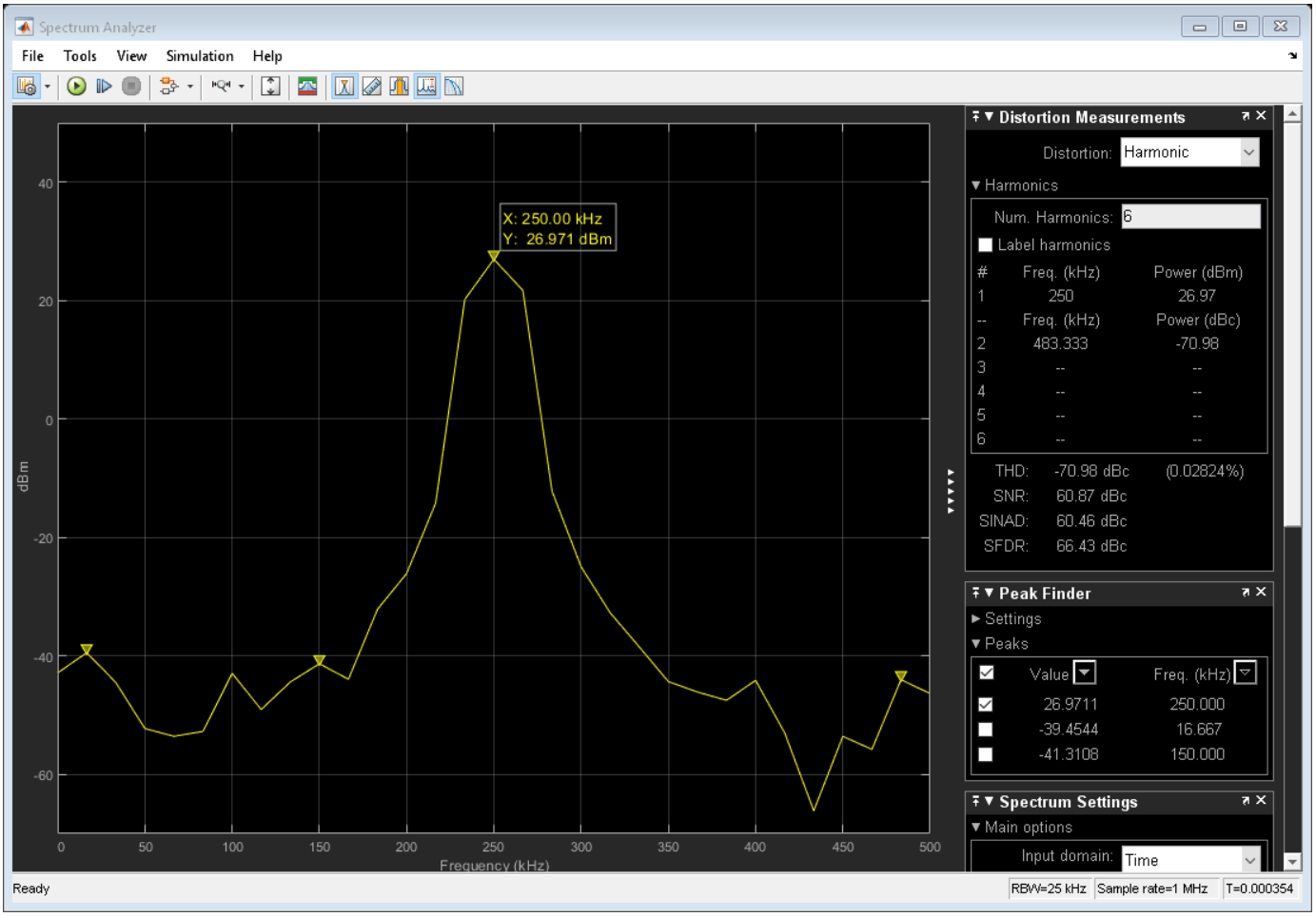
The Flash ADC acts as the input to the Binary Weighted DAC. The ADC uses an internal start clock whose **Conversion start frequency (Hz)** is `1e6` and **RMS aperture jitter (s)** is `1e-12`. The input analog frequency to the Flash ADC from the Signal Generator is `250.98` kHz.

The **Number of bits** of the Binary Weighted DAC is set to `10`. All other parameters are kept at their default values.

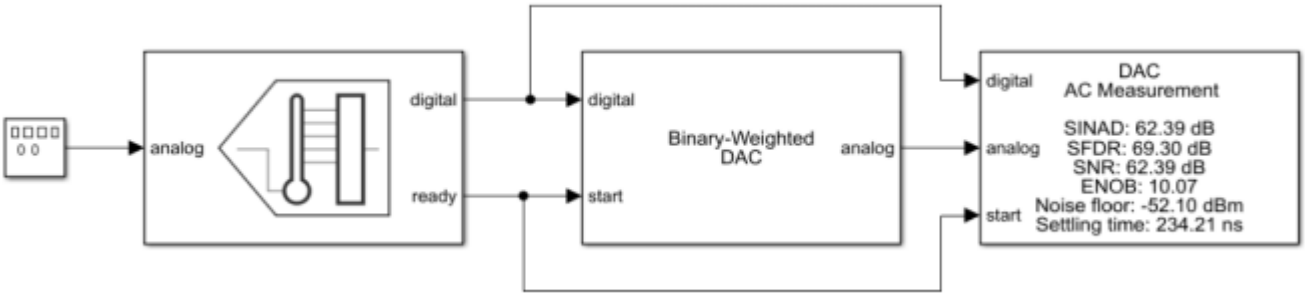
The **Digital signal frequency (Hz)** of the DAC AC Measurement block is `2.5098e5` and the **Start conversion frequency (Hz)** is `1e6`.

Run the simulation for `0.0003586` s.

```
sim(model);
sc = get_param([model '/DAC AC Measurement/DAC AC Measurement/Spectrum '...
     'Analyzer'], 'ScopeConfiguration');
sc.YLimits = [-70 50];
snapnow;
```



The measured AC performance metrics are displayed on the DAC AC Measurement block.

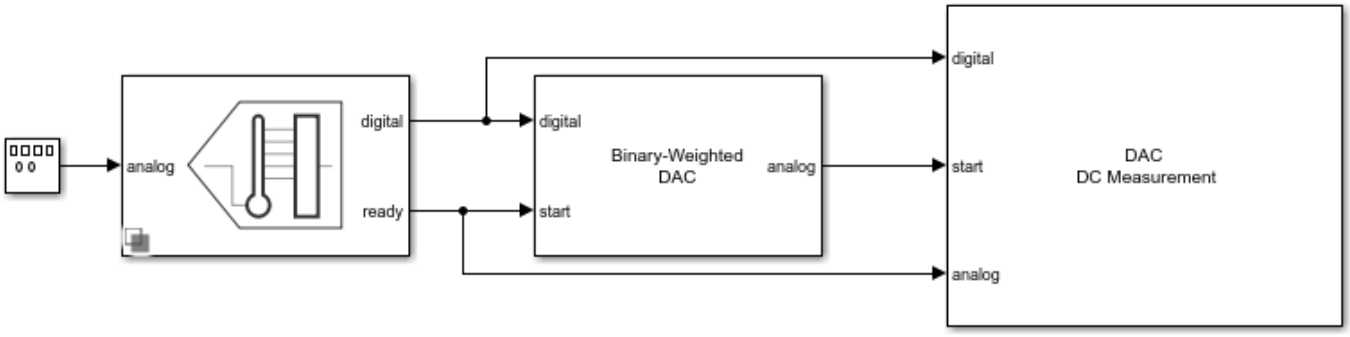


Measure DC Performance Metrics Using DAC DC Measurement

This example shows how to find DC performance metrics such as offset error, gain error, INL, and DNL.

Open the model `dac_dc_measure`. The model consists of a Signal Generator, a Flash ADC, a Binary Weighted DAC block, and a DAC AC Measurement block.

```
model = 'dac_dc_measure';
open_system(model)
```

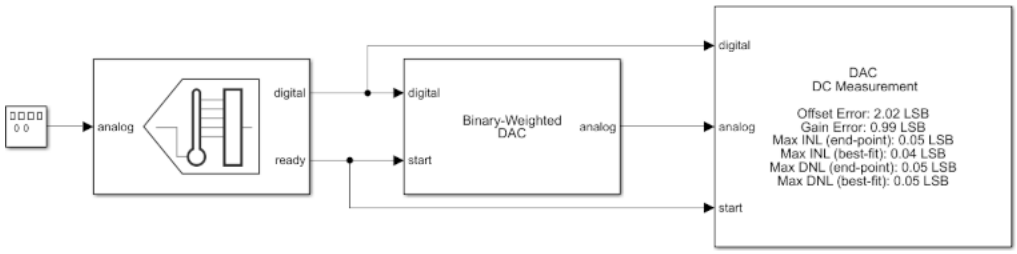


The Flash ADC acts as the input to the Binary Weighted DAC. The ADC uses an internal start clock whose **Conversion start frequency (Hz)** is `1e6` and **RMS aperture jitter (s)** is `1e-12`. The input analog frequency to the Flash ADC from the Signal generator is `900` Hz.

The **Number of bits** of the Binary Weighted DAC is set to `10`. The impairments are enabled in the **Impairments** tab, and the value of the offset error and gain error are set to `2` LSB and `1` LSB, respectively. All other parameters are kept at their default values.

The **Number of bits** and **Conversion start frequency (Hz)** of the DAC DC Measurement block are `10` and `1e6`, respectively. The **Settling time (s)** of the DAC DC Measurement block is set to `4e-7`, twice the **Settling time (s)** of the Binary Weighted DAC to ensure the entire transition period is not considered as part of the steady-state DC measurements.

Run the simulation for `0.02558` s. The measured AC performance metrics are displayed on the DAC AC Measurement block.



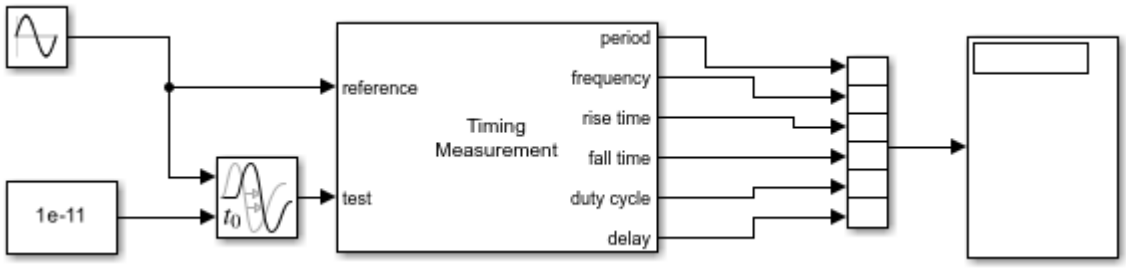
Utilities Block Level Examples

Measure Timing Metrics of Periodic Signal

Measure the period, frequency, rise time, fall time, and duty cycle of a periodic signal. Also measure the delay of the input signal with respect to a reference signal

Open the model `time_metrics_measure`. The model consists of a Timing Measurement block. A period sine wave of `50e6` Hz is applied at the reference input signal. A variable time delay of `1e-11` s is applied at the test input port. The Timing Measurement block is set to measure the period, frequency, 10%-90% rise time, 90%-10% fall time, duty cycle, and the delay with respect to the test input port.

```
model = 'time_metrics_measure';
open_system(model)
```



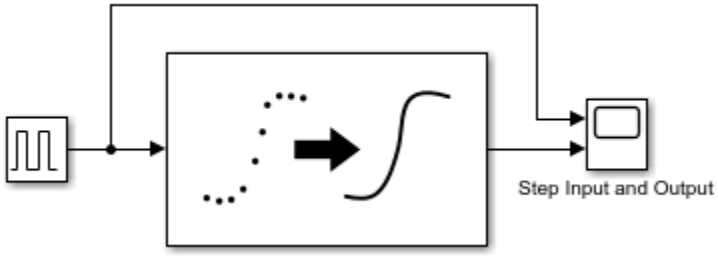
Run the model for `100` microseconds. The results are displayed on a Display block.

- Period = 0.2 microseconds
- Frequency = 50 MHz
- Rise time = 5.903 nanoseconds
- Fall time = 5.903 nanoseconds
- Duty cycle = 0.5
- Delay = 10 picoseconds

Define Output Samples in Lowpass Resampler

To demonstrate the variable step discrete behavior of the Lowpass Resampler block, open the model `LowpassResamplerOutputSampling`.

```
model = 'LowpassResamplerOutputSampling';
open_system(model)
```



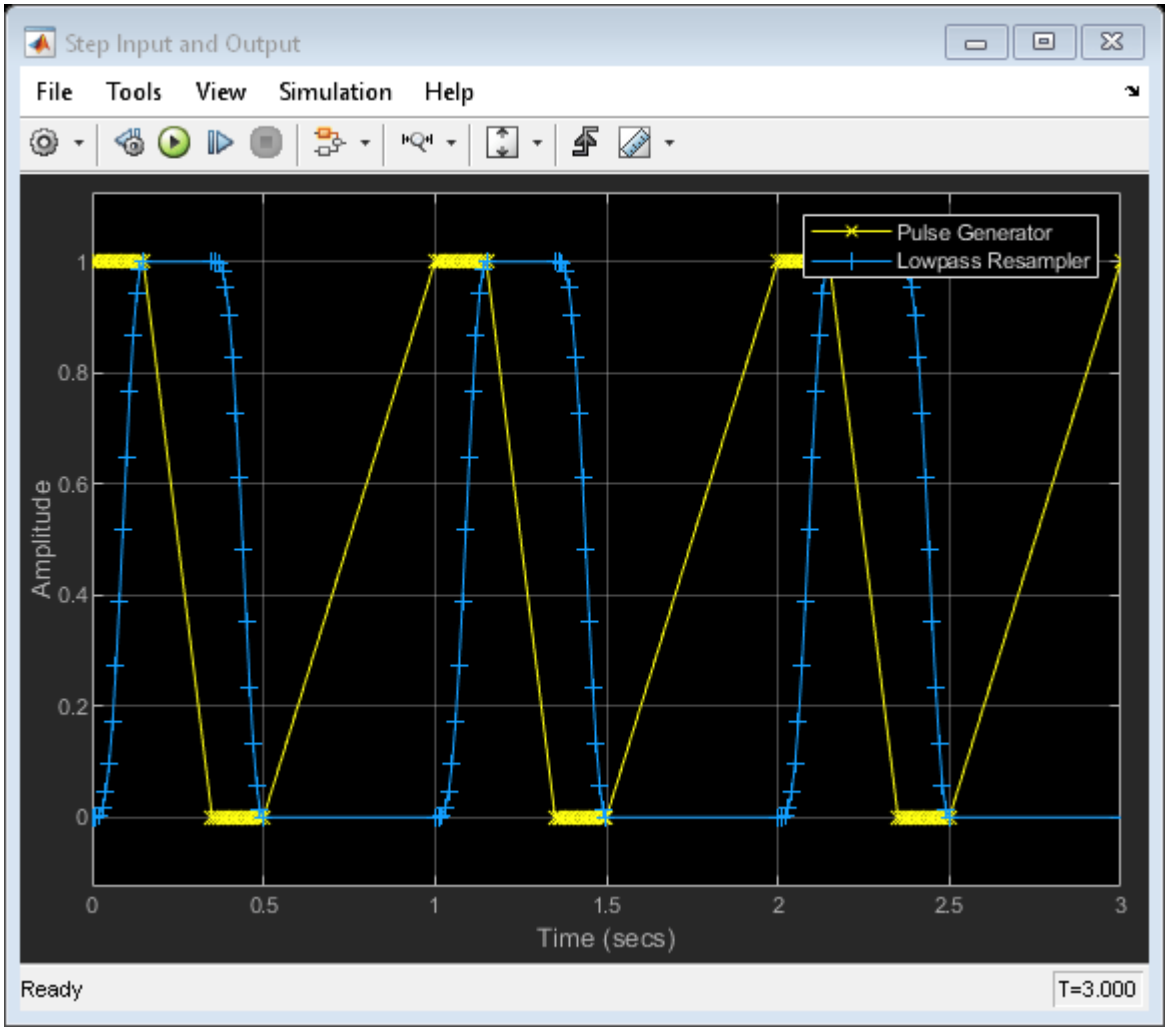
The model consists of a Pulse Generator and a Lowpass Resampler. The pulse generator generates a pulse of 1 Hz with a duty cycle of 35%. The **Pulse type** is selected as `Time based`. In the Lowpass Resampler, the **Output rise/fall time** is set to `0.05` s, the **Number of samples of delay** is set to `1`, the **Output sample time** is set to `Variable step discrete`, and **Samples out per rise/fall time** is set to 5.

In the Model Settings, the **Max step size** of the solver is set to 3, the simulation stop time. As a result, the only sample times are those generated by the model and the sample time display is readily understandable.

Run the simulation.

```
sim(model)
```

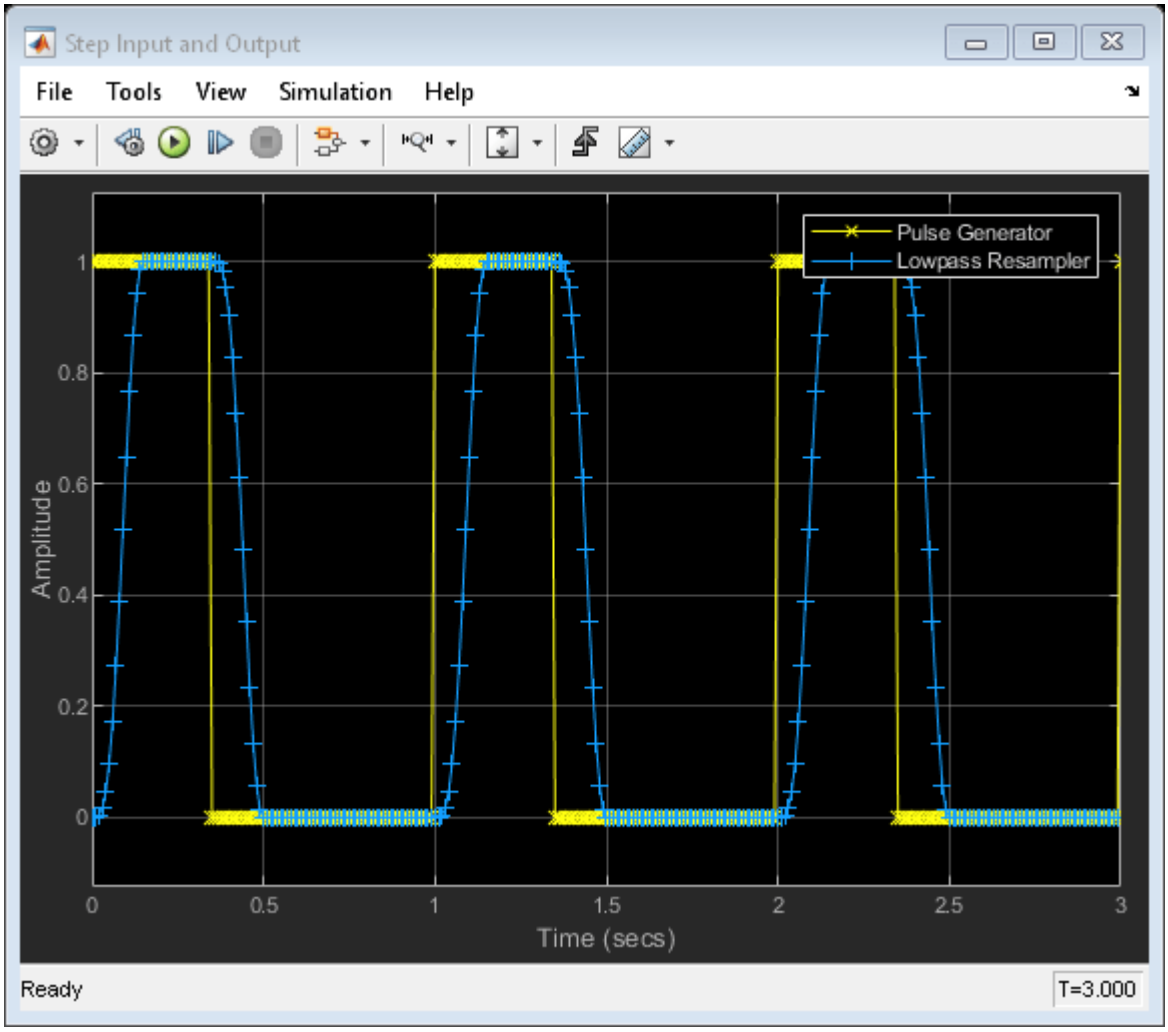
```
ans =

  Simulink.SimulationOutput:
                  tout: [97x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```



To demonstrate the fixed step discrete behavior of the Lowpass Resampler block, set the **Output sample time** is set to `Fixed step discrete`. Rerun the simulation and observe the additional samples produced by the model.

```
set_param([model '/Lowpass Resampler'],'OutputTsType','Fixed step discrete');
sim(model)
```

```
ans =

  Simulink.SimulationOutput:
                  tout: [301x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```


Step Input and Output
File
Tools
View
Simulation
Help
Pulse Generator
Lowpass Resampler
Amplitude
Time (secs)
0
0.2
0.4
0.6
0.8
1
0
0.5
1
1.5
2
2.5
3
Ready
T=3.000

Mix Analog and Digital Signals

Model Linear Circuit Response from SPICE Netlist

In this section...

The Linear Circuit Wizard block parses a SPICE netlist to model the response of a desired linear circuit such as a custom filter design or a circuit with parasitics. The block supports a limited number of SPICE netlist constructs, specifically conforming to Berkeley SPICE syntax [1] or the HSPICE syntax [2]. Edit your SPICE netlist so that the desired circuit is described at the top level of the netlist by supported netlist constructs. The block ignores any statements that use unsupported netlist syntaxes.

The SPICE netlist can contain passive devices, independent sources, controlled sources, and subcircuits.

If present, independent sources are interpreted as input ports by the Linear Circuit Wizard block. The name of the input port is the designator for the circuit element, the port type matches the source type, and the nodes are the nodes to which the source is connected. For each such input port, the stimulus is the Simulink® signal connected to the port, not the description provided by the SPICE netlist.

The output ports are defined by .PRINT or .PLOT statements conforming to the HSPICE syntax. Because the SPICE statements do not supply port names for probed voltages, the Linear Circuit Wizard block supplies default names for voltage output ports. Current output ports are named after the independent voltage source used to sense the current. The netlist must contain all of the circuit nodes and circuit elements required to support the output ports.

The Linear Circuit Wizard block does not support mathematical functions.

Supported SPICE Syntax

Statements can be continued onto multiple lines by starting each continuation line with a plus sign ("+"). For example, these two statements are equivalent.

`E1 2 0 LAPLACE 3 4 6.3e7/6.3e4 1`	`E1 2 0 LAPLACE 3 4` `+ 6.3e7/6.3e4 1`

Three forms of comment are supported.

- Block comment. The comment block starts with the line `#com` and ends with the line `#endcom`. For example:

  ```
  #com
  When you’re lying awake with a dismal headache
  and repose is taboo’d by anxiety,
  I conceive you may use any language you choose
  to indulge in without impropriety.
  #endcom
  ```

- Line comment. The comment is a line that starts with an asterisk ("*"). For example:

  ```
  * Gilbert, Iolanthe, act 2, “Love, unrequited, robs me of my rest”, line 5
  ```

- End of line comment. The comment follows a pipe symbol ("|") and is ignored. For example:

```
.end | That's all, folks.
```

The node labeled "`0`" is defined as the common return node for the circuit.

Component	Syntax	Comment
Passive Devices		
Resistor	R<name> <+ node> <- node> <value>	Create a resistor with a specific value.
Capacitor	C<name> <+ node> <- node> <value>	Create a capacitor of a specific value.
Inductor	L<name> <+ node> <- node> <value>	Create an inductor of a specific value.
Mutual inductance	K<name> <inductor name> <inductor name> <value>	Create a mutual inductance coupling between two inductors. For each inductor, current flow in the positive direction is from the first node to the second node in the statement that creates the inductor. The circuit element value is the mutual inductance factor k, where $0 < k < 1$.
Independent Sources		
Independent voltage source	V<name> <+ node> <- node>	Create an input voltage port in the Simulink block.
Independent current source	I<name> <+ node> <- node>	Create an input current port in the Simulink block.
Controlled sources		
Voltage controlled voltage source	E<name> <+ node> <- node> <+ control node> <- control node> <gain> E<name> <+ node> <- node> LAPLACE <+ control node> <- control node> <k0, k1, ...km>/ <d0,d1,...dn>	Create a voltage controlled voltage source. The gain can be a constant value or a rational transfer function. See "LAPLACE Keyword" on page 7-5 for the transfer function definition.

Component	Syntax	Comment
Current controlled current source	F<name> <+ node> <- node> <voltage source name> <gain> F<name> <+ node> <- node> LAPLACE <voltage source name> <k0, k1, ...km>/<d0,d1, ...dn>	Create a current controlled current source. The gain can be a constant value or a rational transfer function. The current through the controlling voltage source determines the output current. The controlling source must be an independent voltage source, although it does not need to have a zero DC value. See “LAPLACE Keyword” on page 7-5 for the transfer function definition.
Voltage controlled current source	G<name> <+ node> <- node> <+ control node> <- control node> <gain> G<name> <+ node> <- node> LAPLACE <+ control node> <- control node> <k0, k1, ...km>/ <d0,d1,...dn>	Create a voltage controlled current source. The gain can be a constant value or a rational transfer function. The current through the controlling voltage source determines the output current. The controlling source must be an independent voltage source, although it does not need to have a zero DC value. See “LAPLACE Keyword” on page 7-5 for the transfer function definition.
Current controlled voltage source	• H<name> <+ node> <- node> <voltage source name> <gain> • H<name> <+ node> <- node> LAPLACE <voltage source name> + <k0, k1, ... km>/<d0,d1,...dn>	Create a current controlled voltage source. The gain can be a constant value or a rational transfer function. See “LAPLACE Keyword” on page 7-5 for the transfer function definition.
Subcircuit		

Component	Syntax	Comment
Subcircuit	X<name> <external node names> <subcircuit name> <parameter list>	Create an instance of a subcircuit. The number of external nodes must equal the number of nodes in the subcircuit definition. Entries in the parameter list take the form <name> = <value>.
Commands		
.PARAM	.param <param name> = <expression>	Definable parameter
.SUBCKT	.subckt <subname> <external node names> <parameter list>	Begin the definition of a subcircuit. The number of external nodes is arbitrary. Entries in the parameter list take the form <name> = <value>. Subcircuit definitions can be nested.
.ENDS	.ends .ends <subname>	Directive to end a subcircuit definition.
.END	.end	Directive to end the netlist. Optional command.
.INC	.inc[lude] <file name>	Directive to include contents of an external netlist
.PRINT	.print V<name>(<one or two nodes>) | I<name>(<voltage source name>)	Directive to define output ports. Multiple voltage and current outputs can be defined in a single statement. Port names are optional but recommended.
.PLOT	.plot V(<one or two nodes>) | I(<voltage source name>)	Directive to define output ports. Multiple voltage and current outputs can be defined in a single statement.

LAPLACE Keyword

The LAPLACE keyword defines a Laplace domain transfer function for a controlled voltage or current source. The transfer function is defined by a series of transfer function numerator coefficients, from constant term to highest power of *s*, followed by a forward slash ('/'), followed by a series of denominator coefficients from constant term to highest power of *s*.

References

[1] "SPICE Circuit Components." http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/UserGuide/elements_fr.html.

[2] "HSPICE® User Guide: Simulation and Analysis." https://cseweb.ucsd.edu/classes/wi10/cse241a/assign/hspice_sa.pdf.

See Also

Linear Circuit Wizard

Related Examples

- “Circuit Design Details Affect PLL Performance” on page 3-16

More About

- “Converting a SPICE Netlist to Simscape Blocks” (Simscape Electrical)

Define Device Noise Using Linear Circuit Wizard

You can use the Linear Circuit Wizard block to generate device noise and model the response of a linear circuit. The device noise generators are built inside the MATLAB System block that the Linear Circuit Wizard configures, and include flicker noise as well as uniformly distributed Gaussian noise.

Note The Linear Circuit Wizard also offers a **Device noise source** type of input port through which you can inject your own model of device noise. For any given device, you should use either a **Device noise source** input port or an internal device noise generator, but not both.

Device List

The device noise generators are designed to model the current noise produced by resistors and semiconductor devices such as MOSFETs. In the **Device Noise Generators** tab of the Linear Circuit Wizard, you can select from a list of resistors and controlled current sources, which are the circuit elements for which device noise can be modeled. To model a source of device noise, choose the circuit element from the list, enable its device noise generator, and set the parameters for that device noise generator. To disable a device noise generator, select the circuit element and disable its device noise generator.

Device Noise Specification

You can configure a device noise generator using three parameters: **Noise coefficient**, **Corner frequency**, and **Independent noise source/Noise generator seed**.

Noise Coefficient

The **Noise coefficient** parameter defines the spectral density of the generated noise through the equation

$$i^2 = 4kTG$$

where i^2 is the noise spectral density in A^2/Hz, k is Boltzmann's constant, T is the temperature in degrees Kelvin, and G^2 is the noise coefficient in units of conductance (Siemens or 1/Ohms).

For resistors, the device noise is Johnson/Nyquist noise, and the noise coefficient is $G = 1/R$, where R is the conductance of the resistor. The Linear Circuit Wizard performs this calculation from the resistor value in the netlist and sets the noise coefficient to the resulting conductance.

For MOSFETs, the noise coefficient is $G = \gamma g_{dso}$, where γ has a lower bound of 2/3 for MOSFETs with relatively long channel length, and can be 1.5 or greater for short channel MOSFETs. g_{dso} is the output conductance at zero drain bias. Because γ is process dependent, you must perform this calculation and enter the resulting noise coefficient.

Corner Frequency

The **Corner frequency** parameter defines the flicker noise for the device. Flicker noise spectral density increases as approximately 1/f. The corner frequency is the frequency at which the flicker noise is approximately equal to the spectral density of the uniformly distributed thermal noise. The corner frequency is process dependent, but typically has a value around 1 kHz.

Within the MATLAB System block configured by the Linear Circuit Wizard, the frequency response of the flicker noise is implemented by a filter which accurately creates the $1/f$ spectral density from the

corner frequency to four decades below the corner frequency. At even lower frequencies, the modeled flicker noise is essentially constant.

Independent Noise Source/Noise Generator Seed

While most sources of device noise are statistically independent of each other, there are cases in which the same noise process is affects the outputs of multiple devices. For the most common case, enable **Independent noise source**. However, when multiple devices must be driven by the same stochastic process, disable **Independent noise source** on the device noise generator for each of those devices and set **Noise generator seed** to the same positive integer. The resulting device noise generators share the same stochastic process but are statistically independent of all other device noise generators.

See Also

Linear Circuit Wizard

More About

- “Circuit Design Details Affect PLL Performance” on page 3-16

Ports Supported in Linear Circuit Wizard

The Linear Circuit Wizard block uses the port definitions you supply either through the SPICE netlist or the Linear Circuit Wizard's parameter dialog to supply ports on the blocks it produces that are compatible with signals from other Simulink blocks.

Block Ports

The input ports on the blocks that the Linear Circuit Wizard produces accept input signals that represent either a voltage or a current. Similarly, the output ports produce signals which represent either a voltage or a current. Both input and output signals must use a fixed step discrete sample time equal to the block's sample time.

There are five types of ports.

Port Type	Port Circuit Definition	Definition from Netlist
Voltage input	The input signal is the node voltage of a positive circuit node minus the node voltage of a negative circuit node.	Independent voltage source
Current input	The input signal is a current flowing into a positive circuit node and back out through a negative circuit node.	Independent current source
Voltage output	The output signal is the node voltage of a positive circuit node minus the node voltage of a negative circuit node.	V(<node> <node>) clause in .print or .plot statement
Current output	The output signal is the current flowing through a voltage source from the positive output node to the negative output node. The voltage source must be defined in the SPICE netlist.	I(<voltage source>) clause in .print or .plot statement
Device noise source	The input signal is a device noise current that is assumed to be flowing through a resistor or controlled current source in the circuit. The device must be defined in the SPICE netlist. The goal is to make it easier for you to apply your own device noise model to the circuit.	

Parameter Dialog

Through the mask parameter dialog of the Linear Circuit Wizard, you can add or delete ports and define the order in which the ports are listed.

The parameter dialog displays the type, name, and nodes of a single port, along with an indication of that port's position in the list of ports. If no ports have been defined yet, then the type, name, and nodes are blank and require your input. Choose a port from the port list, modify the port definition as desired, and then choose an action.

Port list: Each entry in the port list is a number representing the position in the list, followed by the port name.

Port type: Choose one of the five port types described in the table above.

Port name: The port name is a character array obtained by applying the `eval` function to the parameter value that you specify. You can specify either a literal character array by enclosing the name in single quotes (for example, `'port1'`) or the name of a workspace variable.

Port nodes: Select the port nodes from a list. The contents of the list are derived from the netlist, and depend on the port type you have selected.

- For the **Voltage input**, **Current input**, and **Voltage output** port types, the list contains the names of circuit nodes.
- For the **Current output** port type, the list contains the names of voltage sources.
- For the **Device noise source** port type, the list contains the designators of circuit elements that could be sources of device noise.

Actions: The actions enabled depend on the content of the port list.

- **Create**: Add a new port to the list, based on the port definition currently displayed.
- **Save**: Save the currently displayed port definition to its current location in the port list.
- **Delete**: Remove the currently displayed port from the port list.
- **Move up**: Move the currently displayed port up one position in the port list.
- **Move down**: Move the currently displayed port down one position in the port list.

Port Report

You can obtain a table displaying the ports you've defined by clicking the **Export ports** button from the . This creates the struct array `LinearCircuitWizardPorts` in your base workspace. Display or inspect this struct array to see your port definitions in a table.

See Also

Linear Circuit Wizard

More About

- "Circuit Design Details Affect PLL Performance" on page 3-16

MATLAB Systems Generated from Linear Circuit Wizard

You can use a single Linear Circuit Wizard block to configure one or more MATLAB System blocks to be integrated into a larger Simulink model. The input and output ports of each MATLAB System block will implement the ports defined in the SPICE netlist and parameter dialog, the block icon will be the **Circuit design name** specified in the parameter dialog, and the name given to the block in your model will be the **Block name** specified in the parameter dialog. Each MATLAB System block will have a fixed step discrete sample time that can either be inherited from another block in the model or set by the MATLAB System block itself.

Verify MATLAB System Block Configuration

Define and verify the configuration of a MATLAB System block in the Linear Circuit Wizard before building it. You can also use the Linear Circuit Wizard to modify the configuration of a MATLAB System block that it has already constructed by setting **Block name** to the name of the block to be modified.

You can verify the **Circuit design name**, **Block name**, **Netlist file name**, port definitions, and device noise generators. You can also verify the response of the block using either transfer function plots or a transfer function pole/zero report. Both options are available as soon as the netlist file is parsed and at least one output port is defined.

Plot transfer functions: Click the **Plot transfer functions** button to obtain a plot of all of the transfer functions the block will implement. This produces a figure window with a separate tab for each output port. Each output port tab plots the transfer function magnitude from each input port and device noise generator to that output port. The plot scale is decibels of gain versus log frequency, making it easier to identify approximate pole and zero locations as well as overall gain.

Export poles and zeros: Click the **Export poles and zeros** button to obtain a report of the transfer function poles and zeros. This produces the struct array `LinearCircuitWizardPoleZero` in your base workspace. The fields of each struct are:

- Poles: A row vector of complex pole locations in the Laplace domain
- Zeros: A cell matrix in which each cell is a row vector of complex zero locations in the Laplace domain, one for each channel through the block
- Gains: A matrix of real-valued gains, one for each channel through the block

Build MATLAB System Block

To configure a new or existing MATLAB System block to reflect the configuration currently defined in the Linear Circuit Wizard, click the **Build/modify block** button. If in doubt, look for a warning message in the Linear Circuit Wizard parameter dialog indicating that the block does not reflect the current configuration. Clicking the **Build/modify block** button will resolve the warning.

Sample Time

The sample time of the MATLAB System block is specified in terms of a sample rate (frequency) rather than directly as a sample time. You can configure the sample time of the MATLAB System block through its parameter dialog. The default condition is for the block to inherit its sample time. To set the sample time of the block directly, disable **Inherit sample rate from input**. This enables the **Sample rate** parameter. Set the sample rate to the desired frequency.

MATLAB System blocks configured by the Linear Circuit Wizard often model analog circuits that are combined with other blocks that model digital circuits. Whereas analog circuits are usually modeled using a fixed step discrete sample time, digital circuits are usually modeled using a variable step discrete sample time. For more information on combining fixed step discrete sample time with variable step discrete sample time, see the example “Digital Timing Using Fixed Step Sampling” on page 3-7.

See Also

Linear Circuit Wizard

More About

- “Circuit Design Details Affect PLL Performance” on page 3-16
- “Digital Timing Using Fixed Step Sampling” on page 3-7