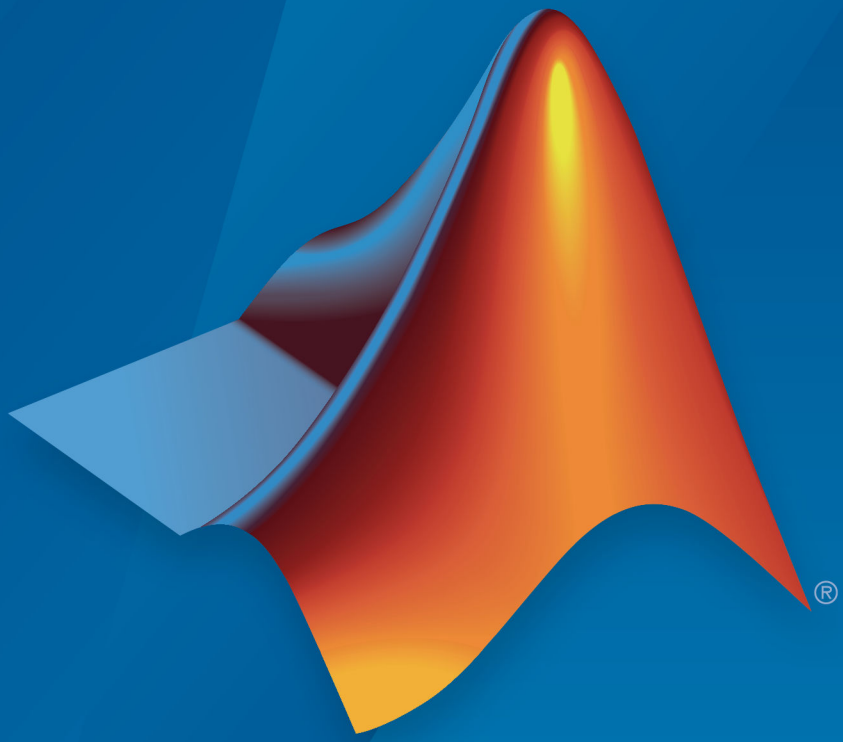
Mixed-Signal Blockset™

User's Guide



MATLAB® & SIMULINK®

R2019b

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

Revision History

March 2019	Online only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)

Contents

1 PLL Featured Examples

2 ADC Featured Examples

3 Mixing Analog and Digital Signals Featured Examples

4 PLL Block Level Examples

PLL Featured Examples

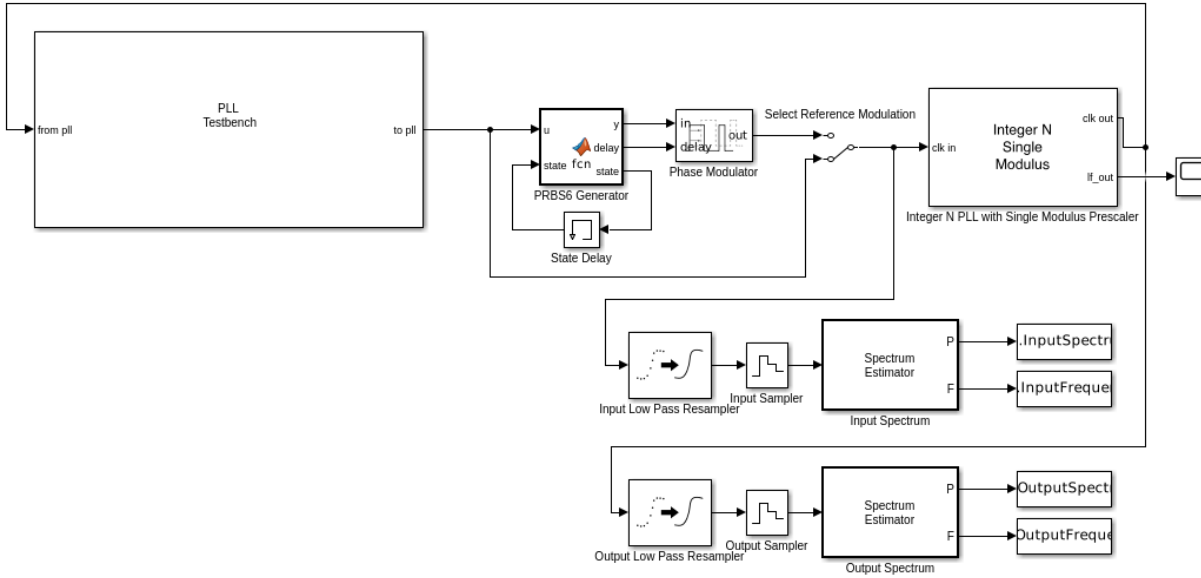
Phase Noise at PLL Output

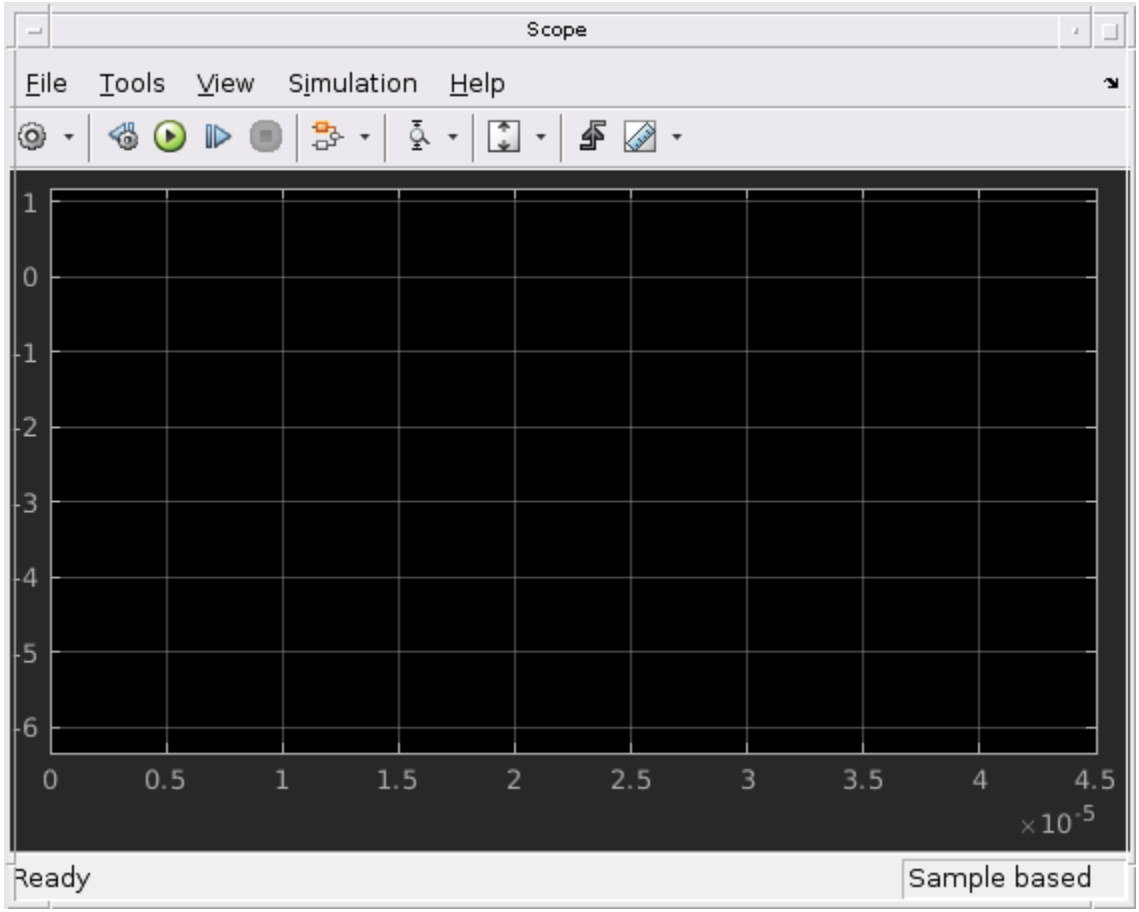
This example shows how to predict the phase noise at the output of a phase-locked loop (PLL), simulate the PLL using the PLL Testbench, and compare simulation results to theoretical predictions.

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. See Integer N PLL with Single Modulus Prescaler for more details.

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 used to convert 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 inroduced 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 ration on the PLL control loop's error function.
5. `TotalPhaseNoise.mat` - All phase noise sources enabled, and the prescaler ratio 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 scipt `prepareExpectedSpectra` computes the spectral density of the reference phase modulation and the VCO output phase noise, then 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) whereas the VCO phase noise is a stockastic 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 spectgral componentent 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 have been calculated
- `LoopZofs` - The transfer impedance of the loop filter as a function of 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 he 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 spectram has been 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 has been 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.

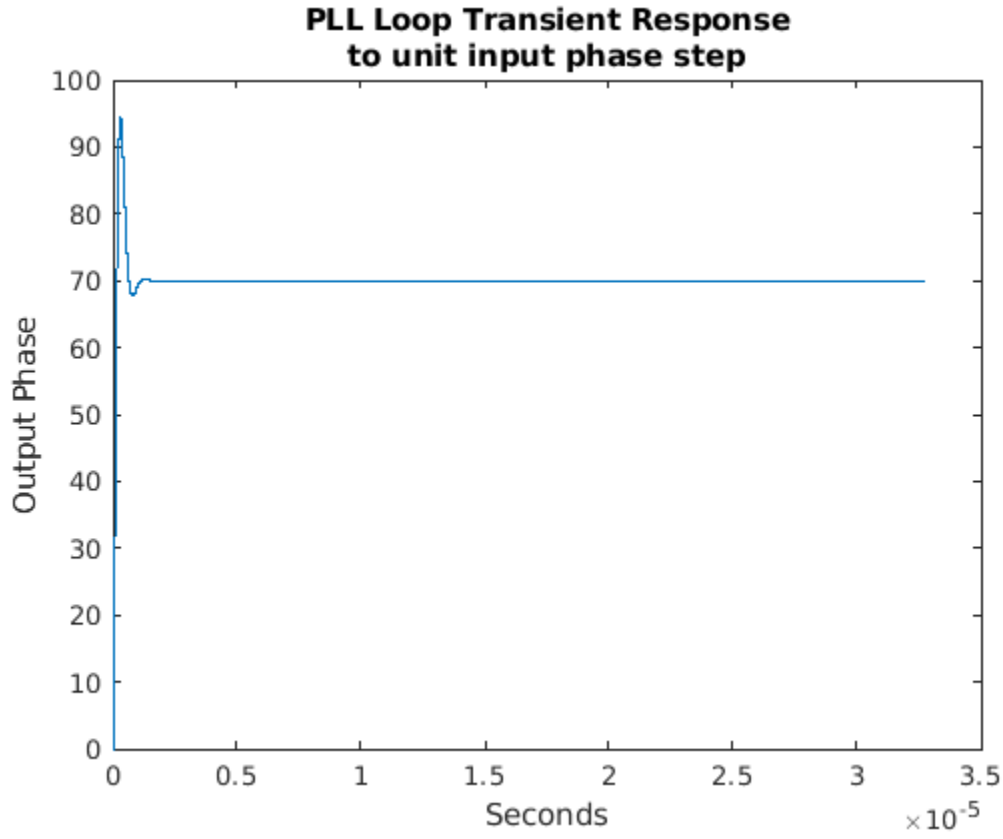
Start by looking at the closed loop response of the PLL. Consider three fundamental responses: transient response, transfer function, error function.

Transient response: In most PLL applications, the loop transient response is most impoortant when the loop is initially acquiring phase lock. For very small initial frequency

offsets or for relatively high closed loop bandwidth, the transient resopnse will predict 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.

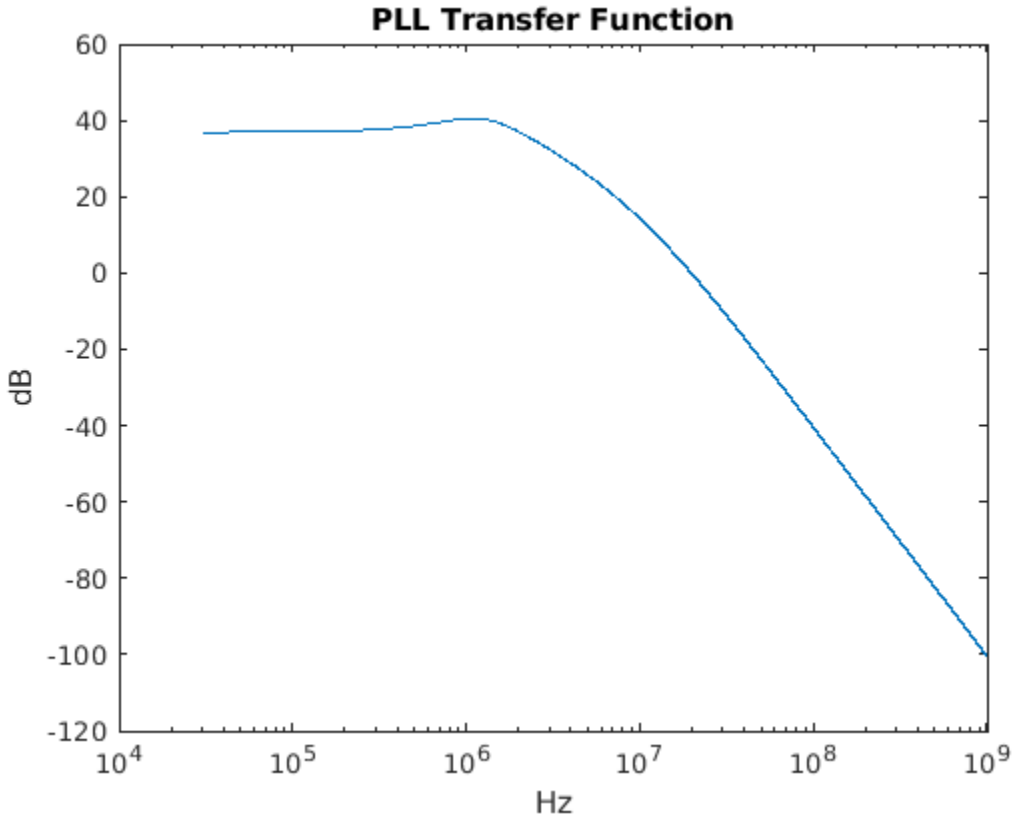
```
figure(1);
plot(LoopPhStep.Time,LoopPhStep.Data);
title({'PLL Loop Transient Response';'to unit input phase step'});
xlabel('Seconds');
ylabel('Output Phase');
```



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');
```

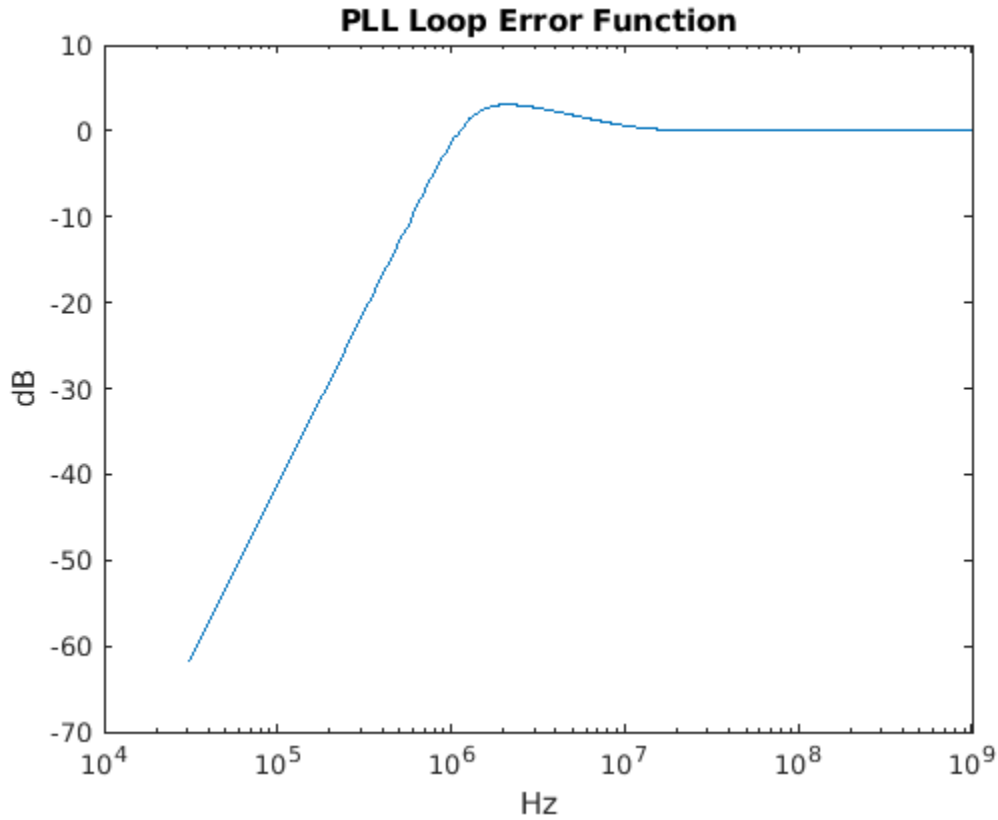


Error function: The PLL control loop is expected to reject phase noise generated inside the PLL itself, such as VCO phase noise. The transfer function between the VCO phase

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.

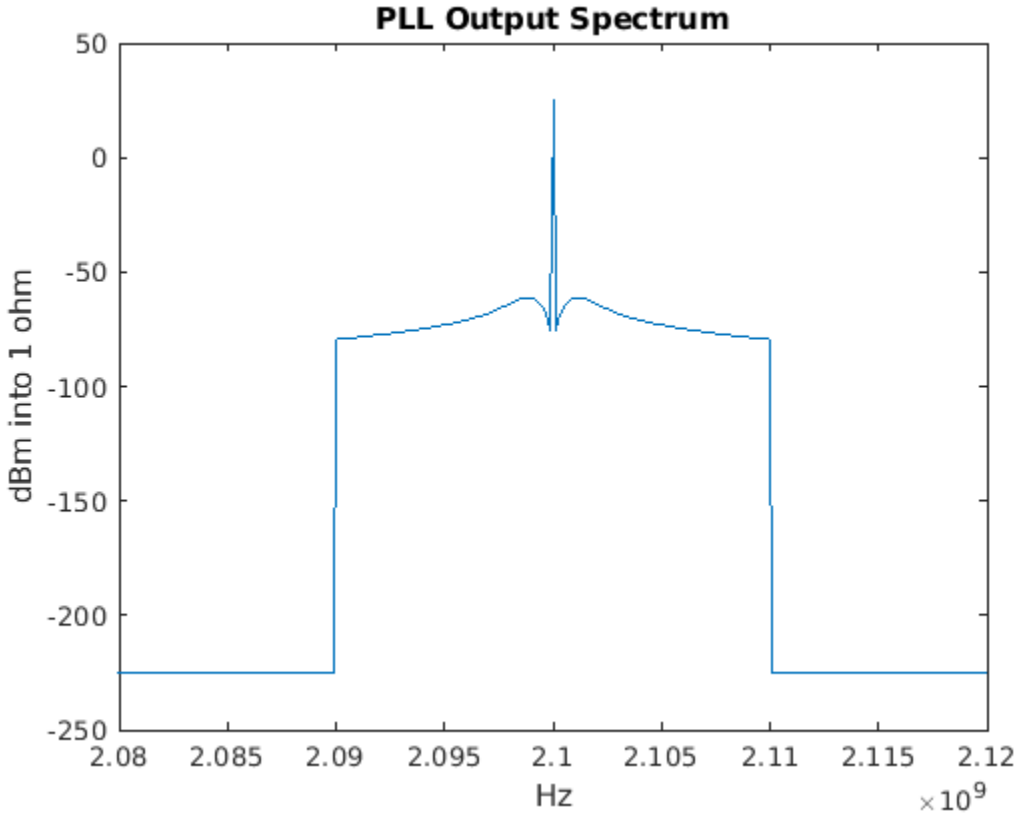
Create a log-log plot of the PLL loop error function.

```
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)]);
```



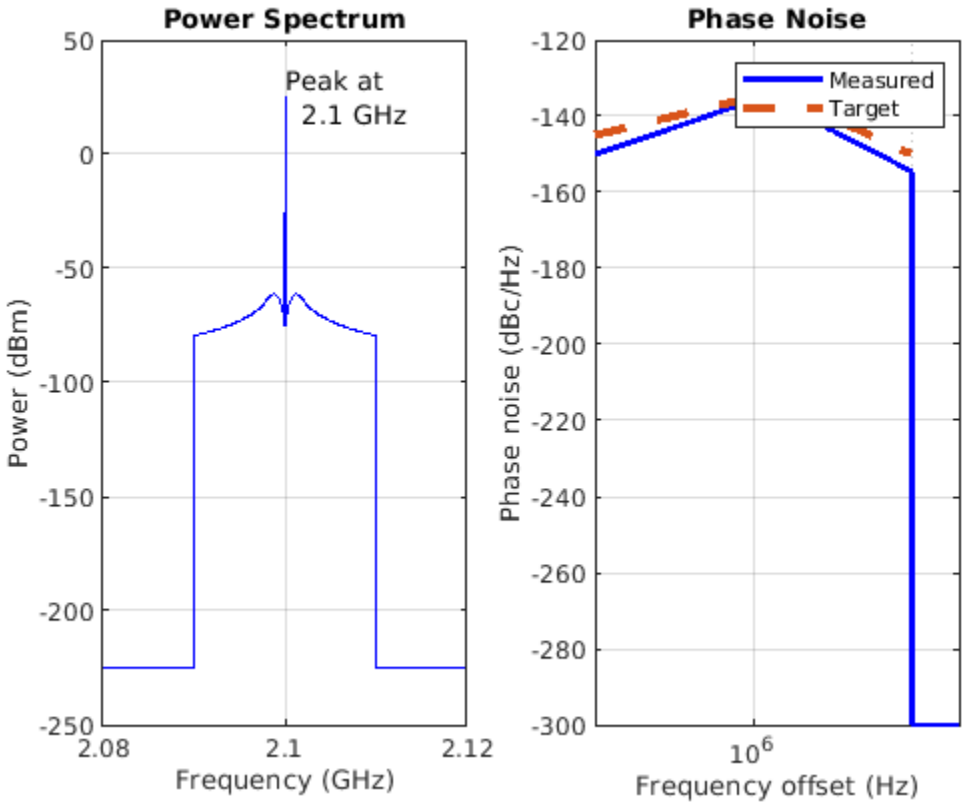
The `phaseNoiseMeasure` function is a callback function used by the PLL Testbench. It displays the target output phase noise spectral density along with the simulated or expected phase noise spectral density.

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 spcifies 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).');
```



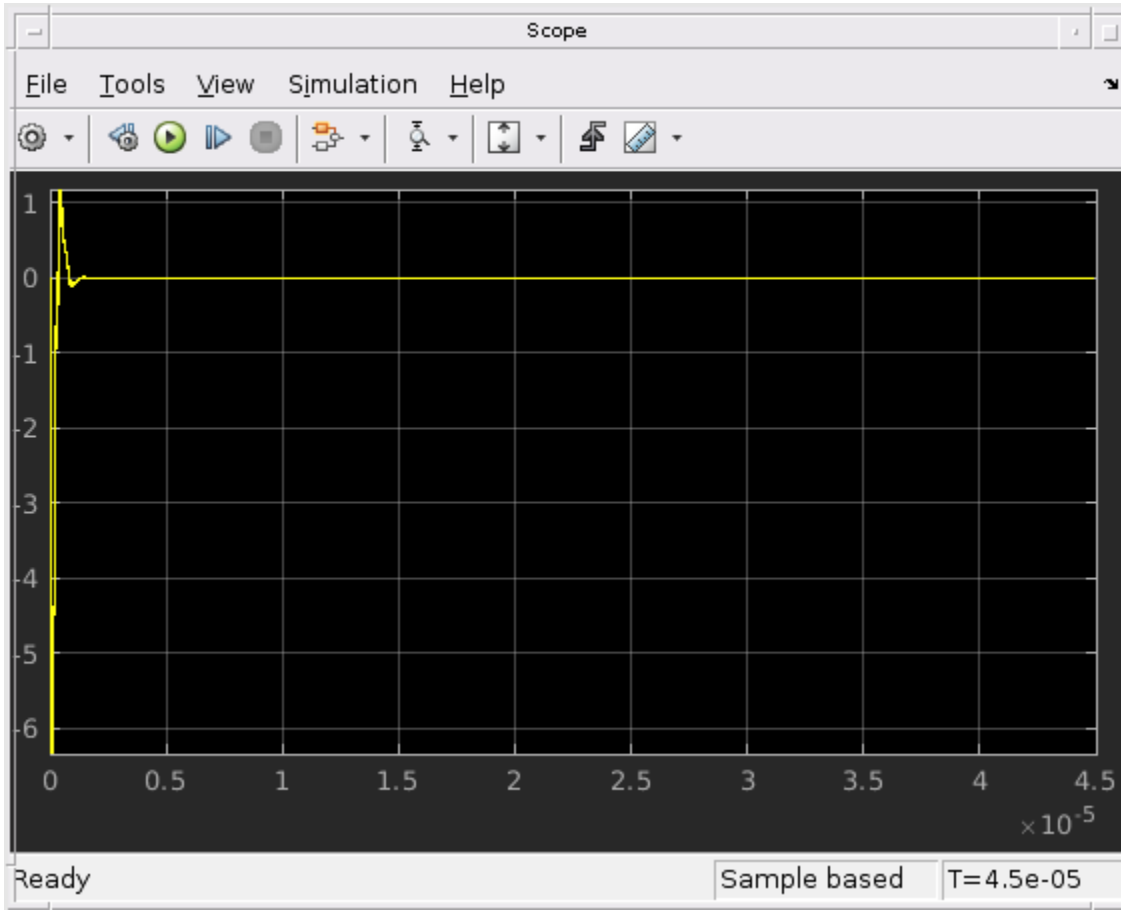
If the loop response isn't satisfactory, you might look at more detailed results such as the loop filter transimpedance (`LoopZofs`) to get additional insights that could help you improve the PLL design.

Run the Simultaion

While the configuration process is entirely straightforward, there are a lot of parameters to be configured. Use the `configureExamplePLL` function to configure the PLL Simulink model.

```
% Configure the system.
configureExamplePll;
% Execute
SimOut = sim('PllPhaseNoiseExample');
```

```
Warning: Unable to resolve the name autosar.api.Utils.initMessageStreamHandler.
```



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

- `SimInputFrequency` - A vector of frequencies for which the imput 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 resolutioon bandwidth of `CfgResBandwidth`)

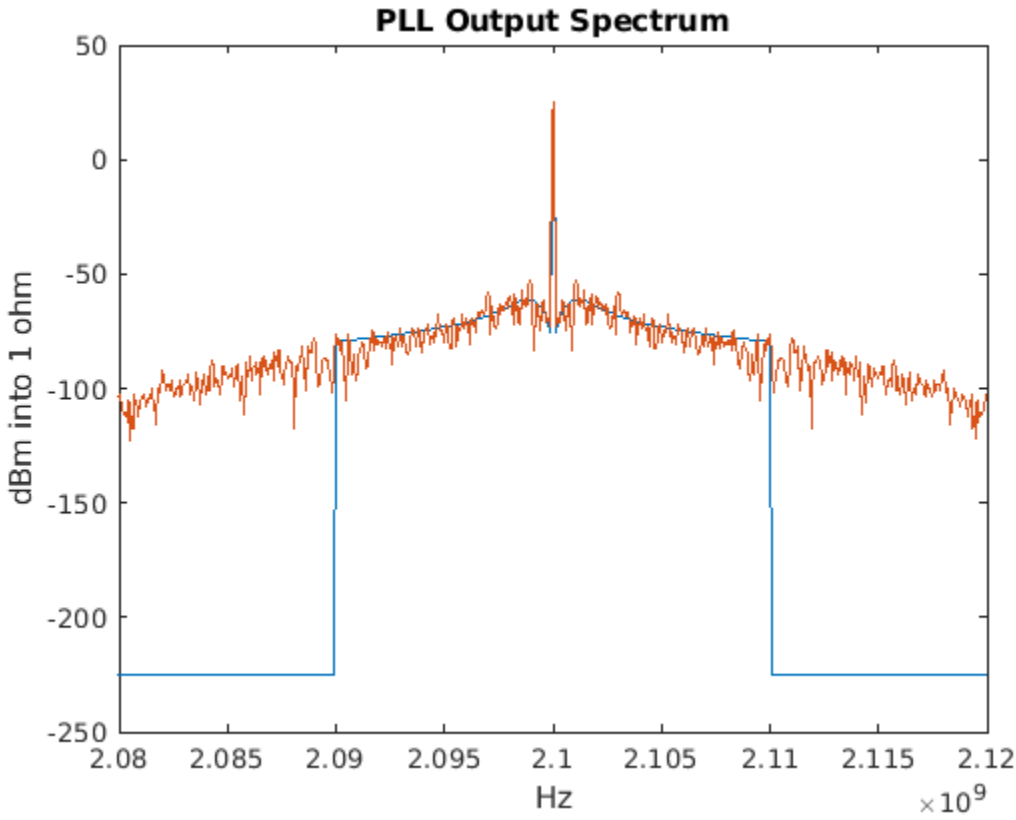
Make the results easier to plot from the workspace

```
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

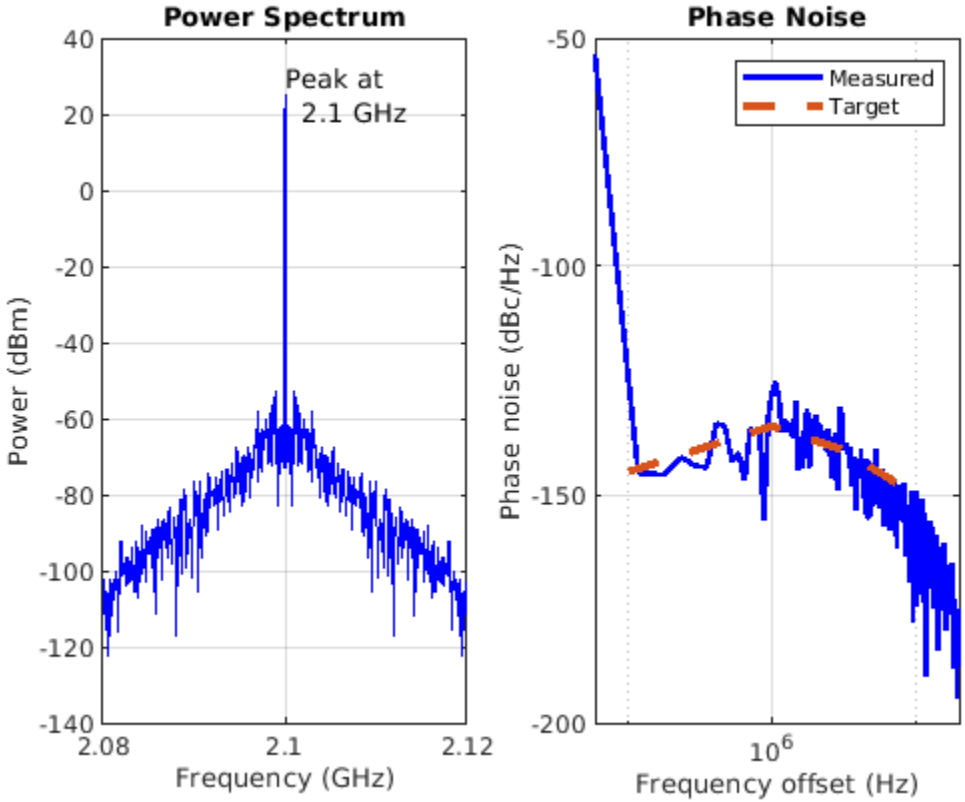
Compare expected and simulated phase noise at the PLL output.

```
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
```

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.

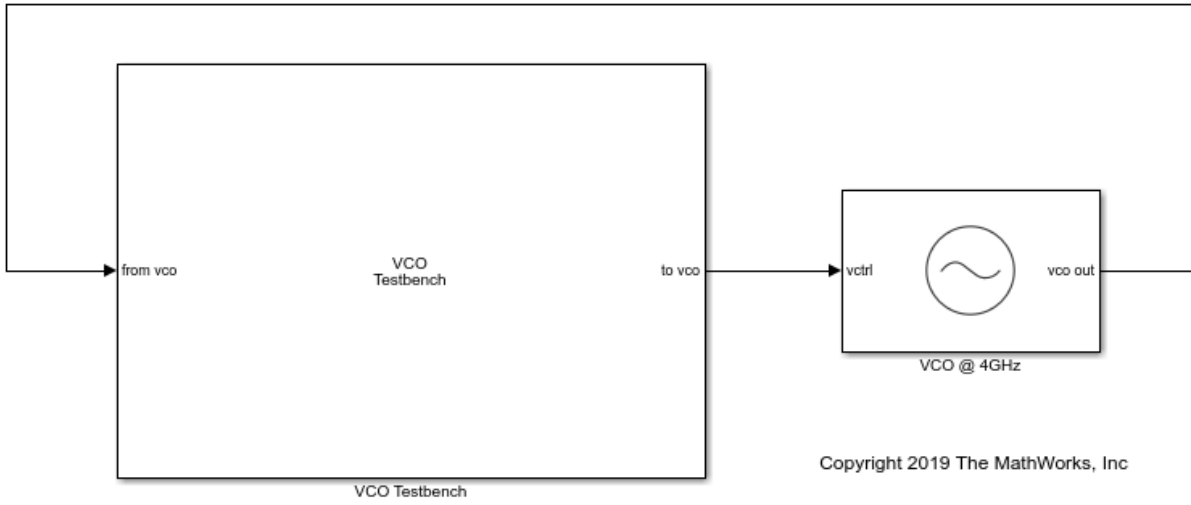
You can use PLL models to explore and design different loop filters, simulate different operating frequencies, determine different divider ratios, or assess the frequency synthesizer performance once embedded in a larger system. For example, Mixed-Signal Blockset PLL models can be helpful if you need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

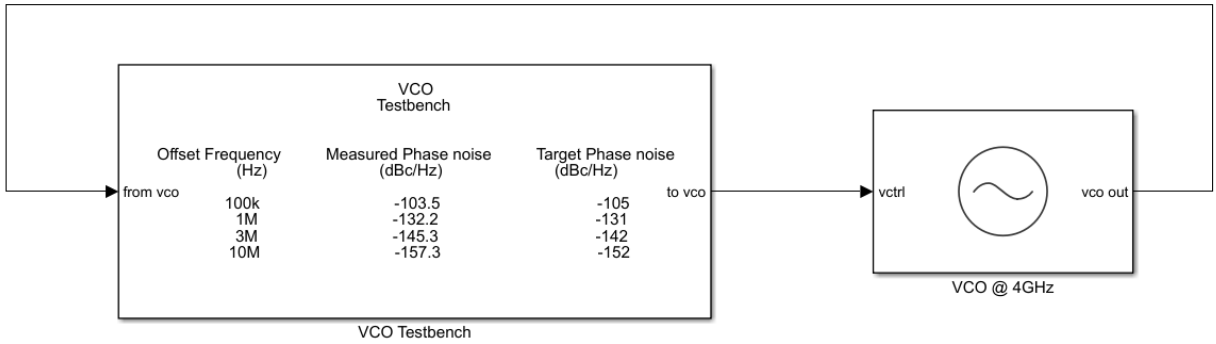
$$F = Kvco * Vctrl + Fo$$

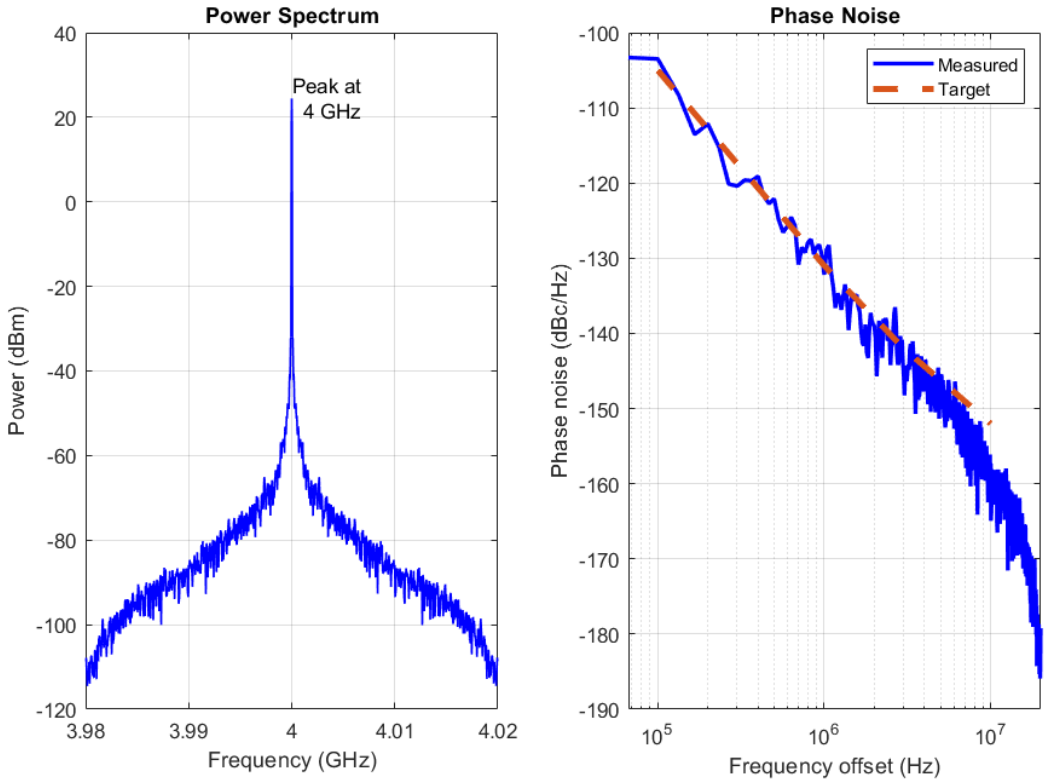
Substituting the values for the output frequency, the free running frequency, and the voltage sensitivity, the equation gives a control voltage of be 5 V.

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 **Sampling frequency** to four times the lock frequency, `4*4` GHz, and 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.



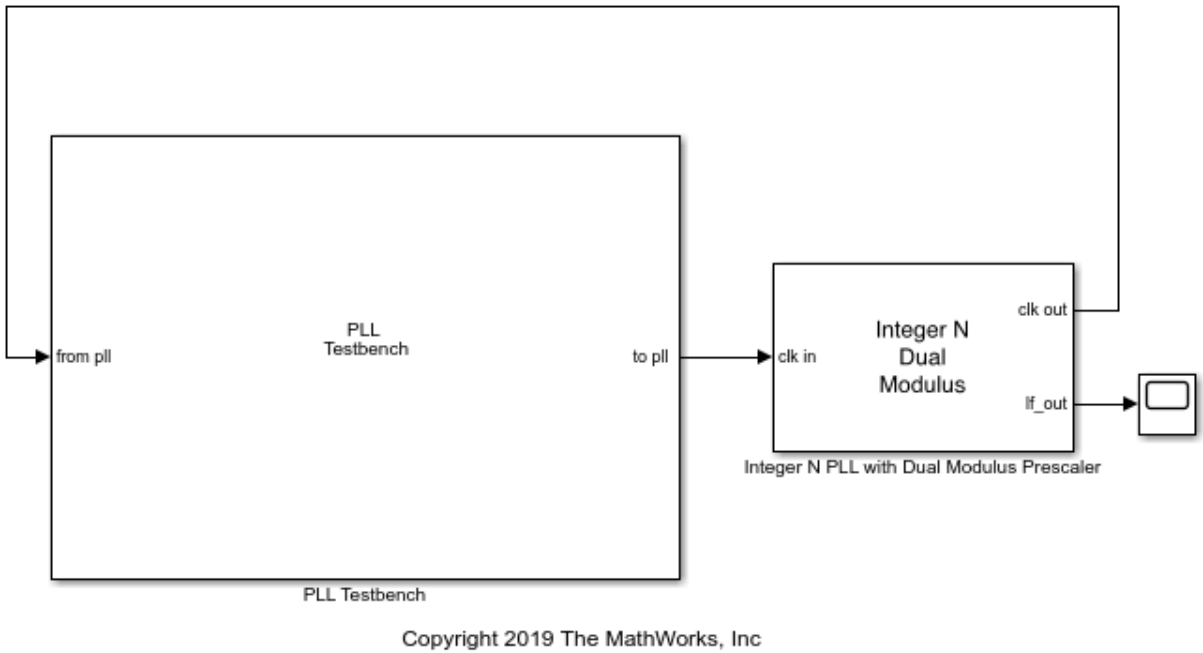


You can now proceed with modeling the complete PLL system and verify its phase noise performance.

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:

$$\frac{4.2e9}{1.6e6} = 2625$$

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

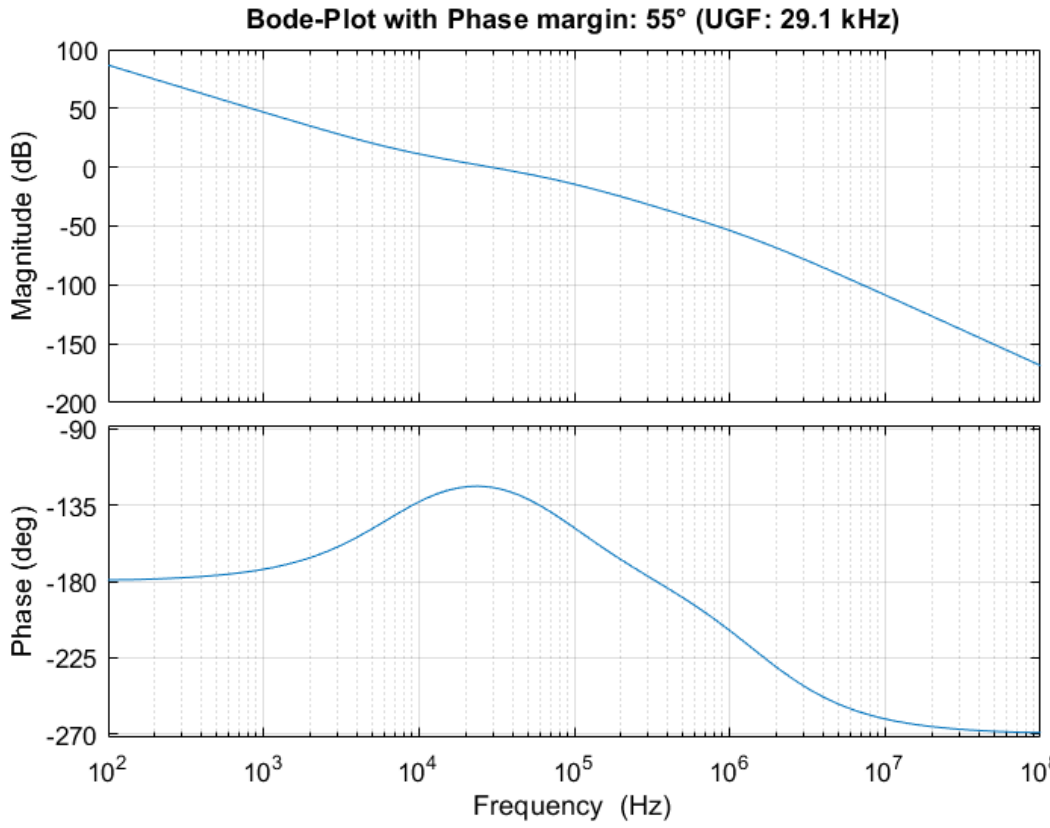
C2 = `2.2e-9` F

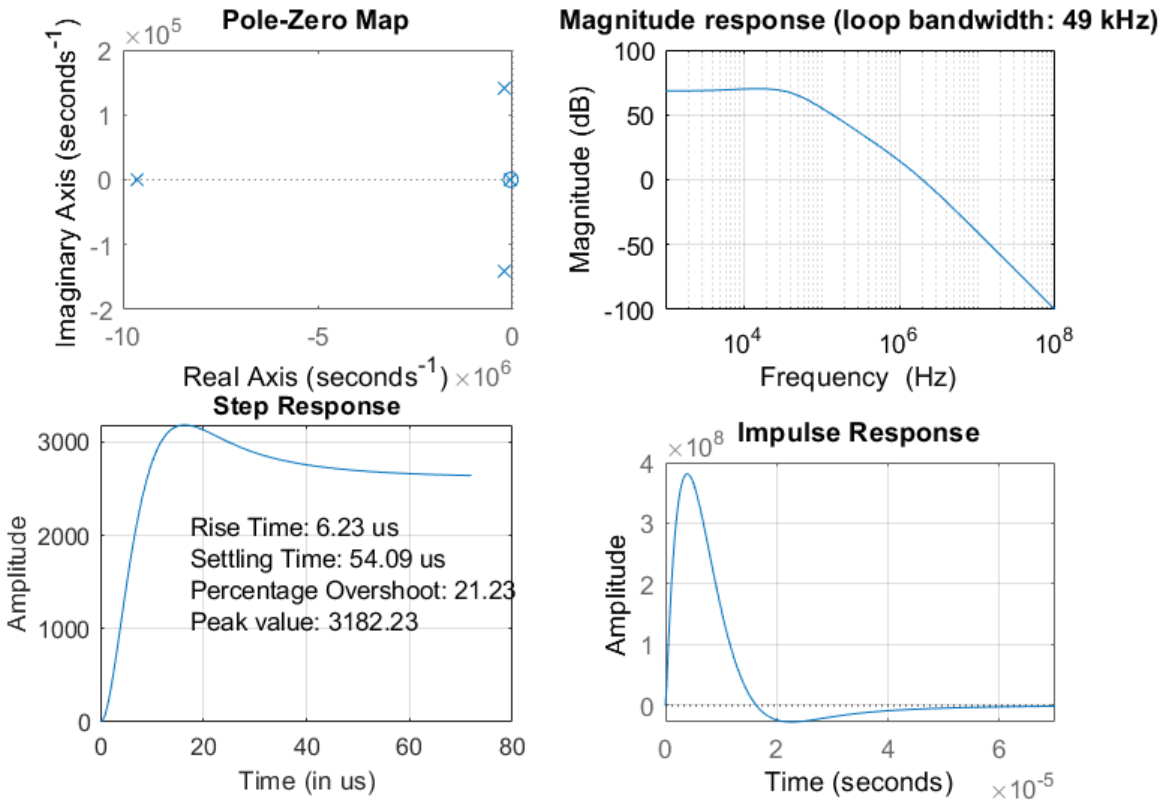
C3 = `100e-12` F

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

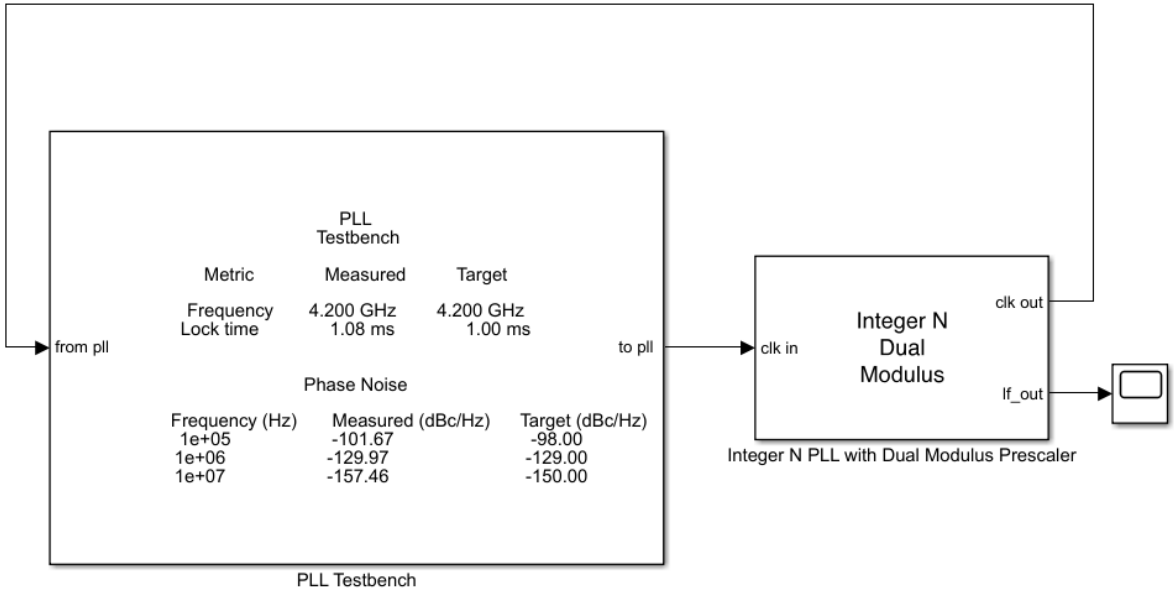
Connect the PLL to a Mixed-Signal Blockset PLL Testbench to validate its performance. The testbench defines the stimuli used to test the PLL. In this case, use a square waveform with 1.6 MHz frequency.

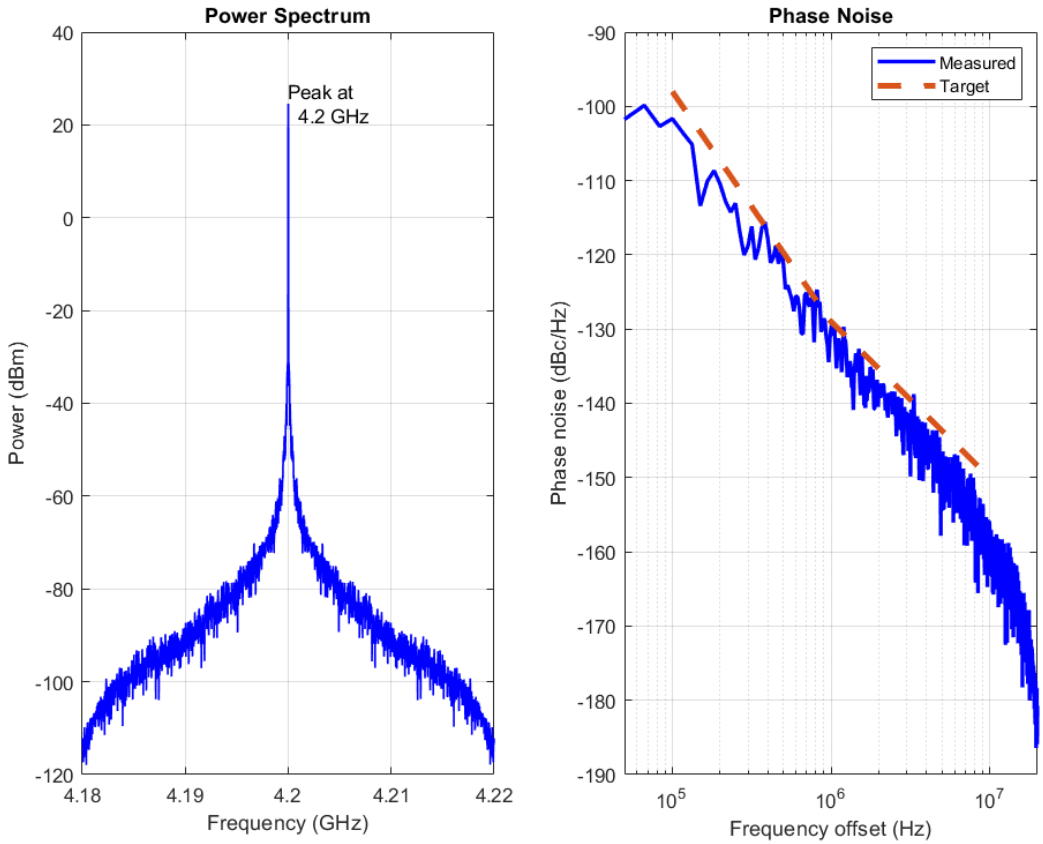
Set up the testbench to measure operating frequency, lock time, and phase noise. The data sheet specifies a 1 ms lock time with 1 ppm frequency error, that is to say 4.2 kHz (Table 5). Provide the same error tolerance. For measuring the phase noise, use a configuration similar to the one used for the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

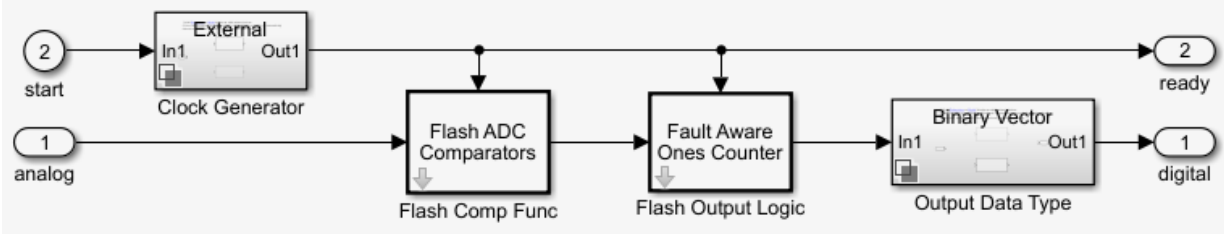
ADC Featured Examples

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.

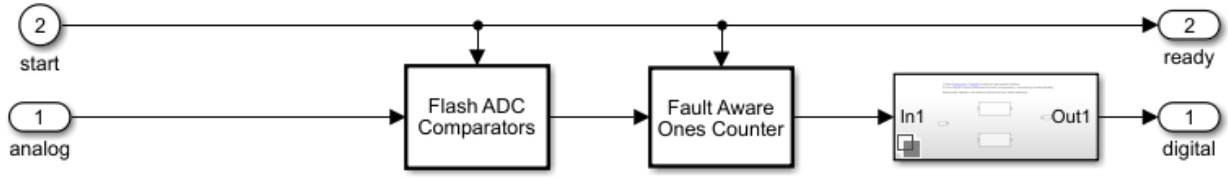
Customize Flash ADC

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 two major components:

- Flash ADC Comparators
- Fault Aware Ones Counter



Flash ADC Comparators

An *N*-bit flash ADC uses 2^{Nbits} comparators in parallel. The Flash ADC Coparators 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

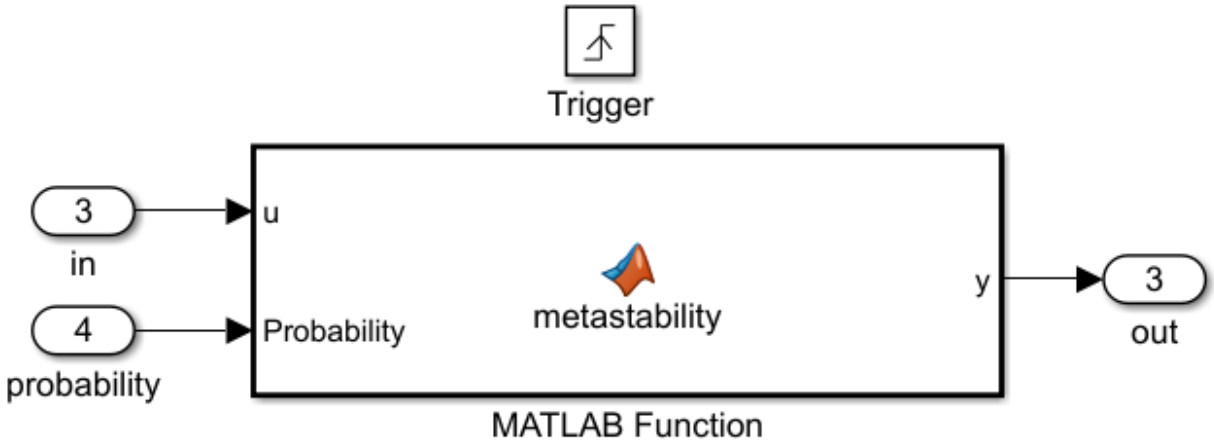
The Fault Aware Ones Counter subsystem implements the impairments in the flash ADC architecture. Real ADCs handle conversion from thermometer to binary through logic circuits. This subsystem takes the sum-of-elements of the vector stored by the comparators and applies that sum to a lookup table to simulate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

Implement Metastability Probability as an Impairment to Flash ADC

To add metastability impairment, place a triggered susbsystem 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.

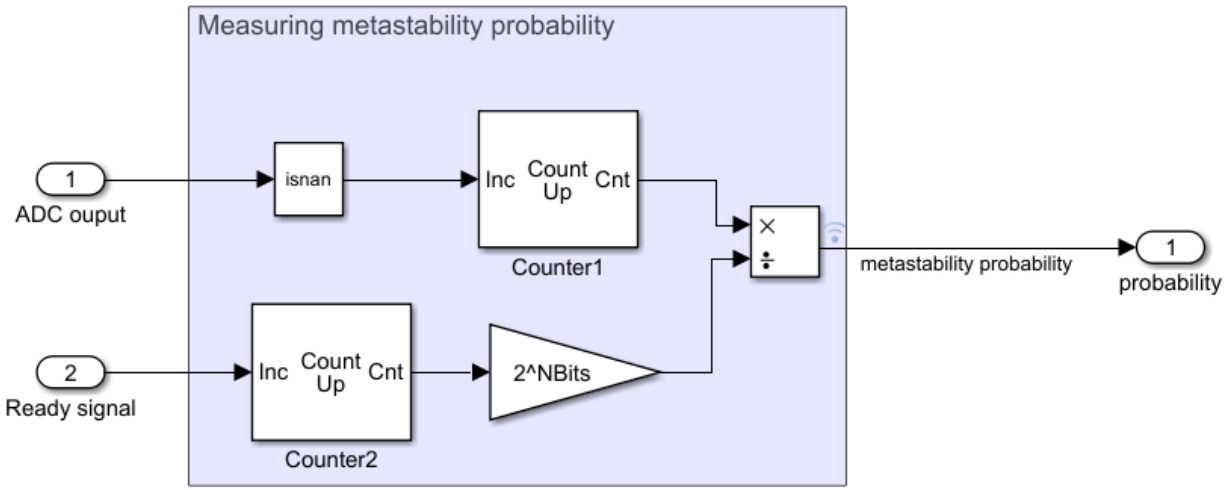
```
%  function y = metastability(u, Probability)
%   mult = ones(size(u));
%   mult(rand(size(u)) < Probability(1)) = NaN; % metastability = NaN
%   y = u .* mult;
%  end
```



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:



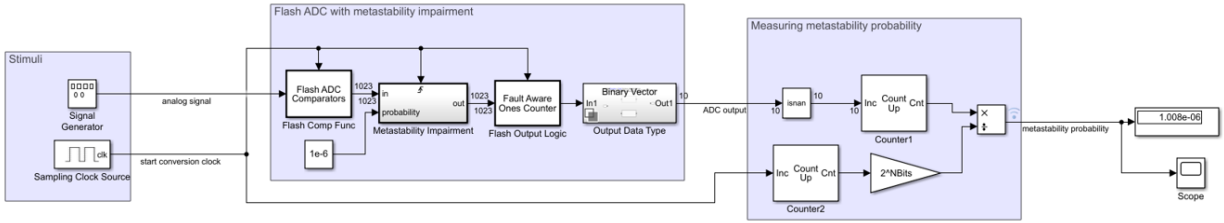
The Inports are:

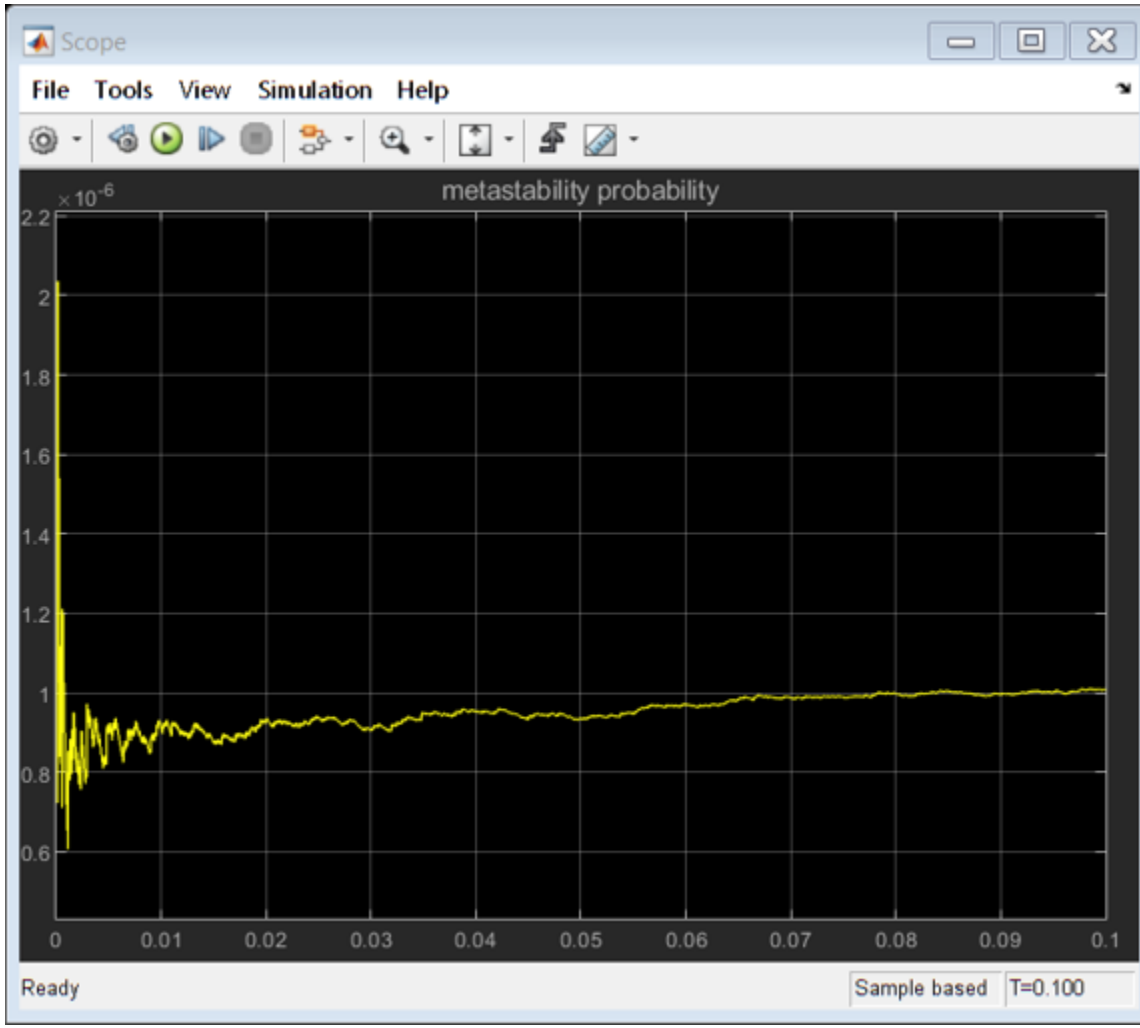
- ADC output- Receives the output digital code generated by the flash ADC.
- Ready signal- Receives the ready signal which represents the rate at which the digital conversion is taking place. The digital code gets generated at each rising edge of the signals received by 'Ready signal' port.

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,'/Scope']);
sim(model1);
```


Stimuli
Signal Generator
analog signal
Sampling Clock Source
clk
start conversion clock
Flash ADC with metastability impairment
Flash ADC Comparators
1023
1023
Flash Comp Func
in
probability
out
Metastability Impairment
1e-6
1023
1023
Fault Aware Ones Counter
Flash Output Logic
Binary Vector
In1
Out1
Output Data Type
10
ADC output
Measuring metastability probability
isnan
10
10
Inc Count Up Cnt
Counter1
Inc Count Up Cnt
Counter2
2^NBits
metastability probability
1.008e-06
Scope

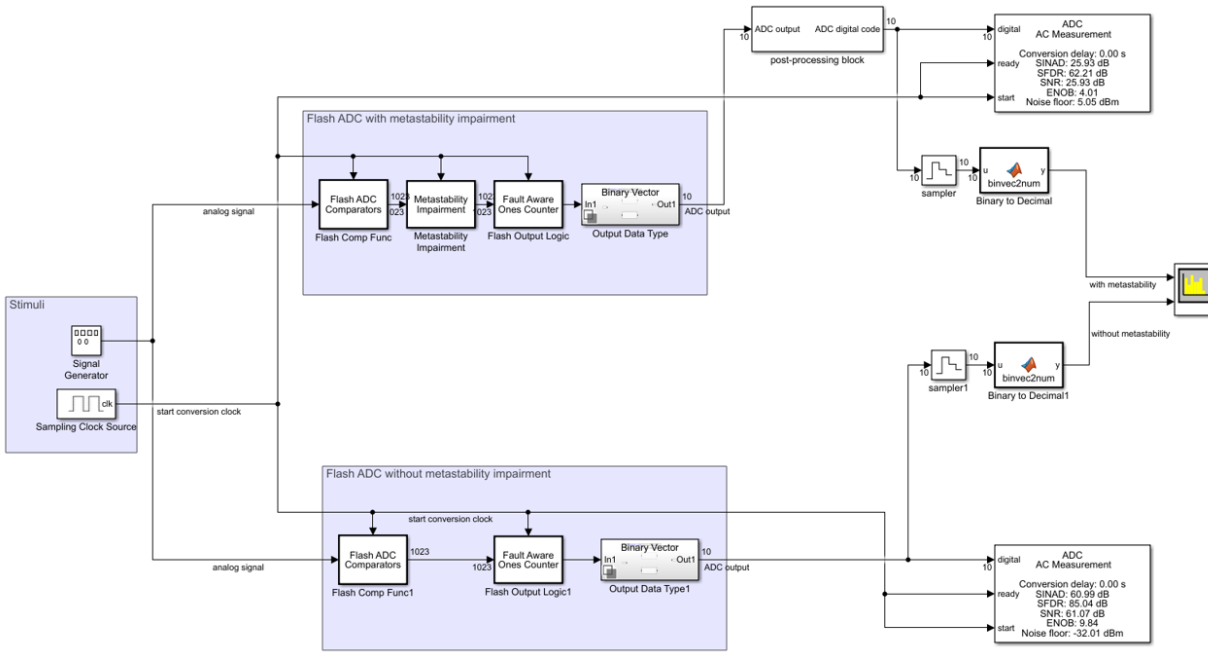


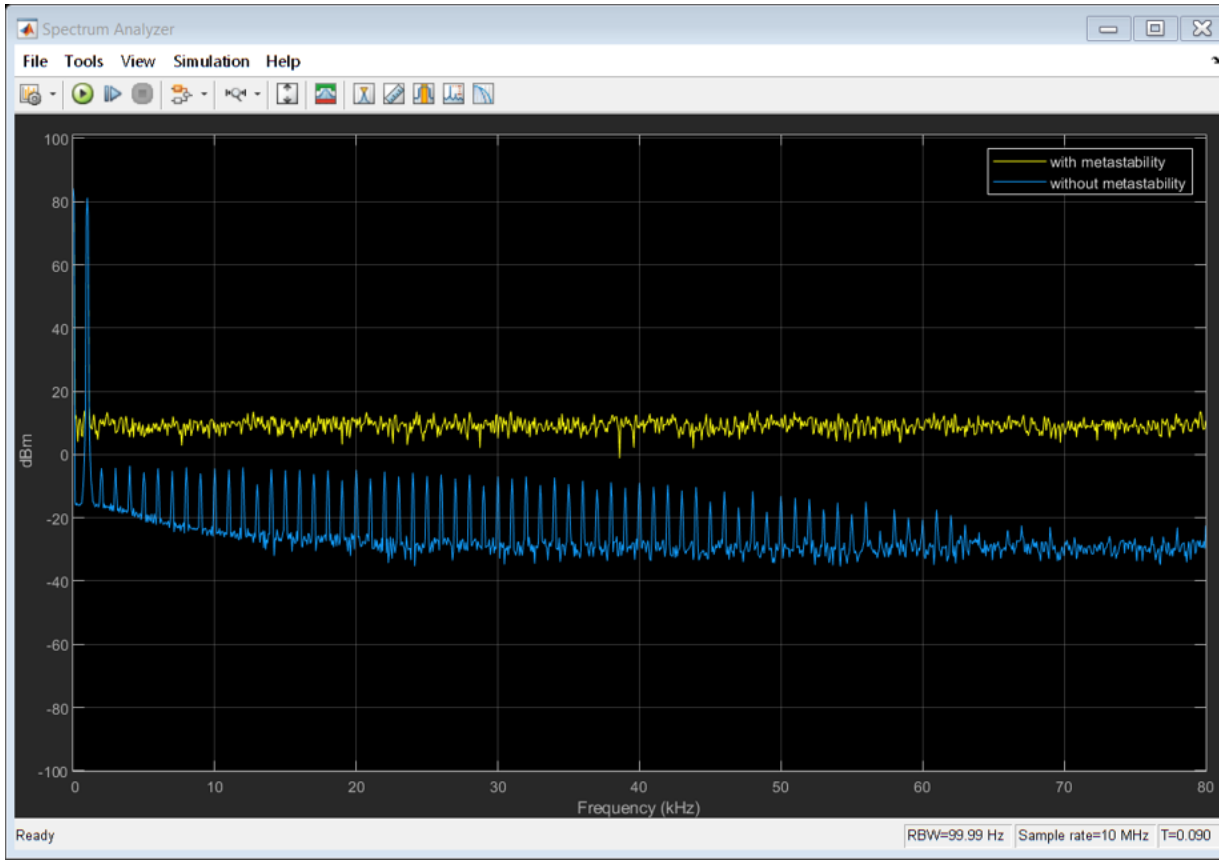
Effect of Metastability on Dynamic Performance of ADC

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);
```




Spectrum Analyzer
File Tools View Simulation Help
with metastability
without metastability
dBm
100
80
60
40
20
0
-20
-40
-60
-80
-100
0
10
20
30
40
50
60
70
80
Frequency (kHz)
Ready
RBW=99.99 Hz
Sample rate=10 MHz
T=0.090

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

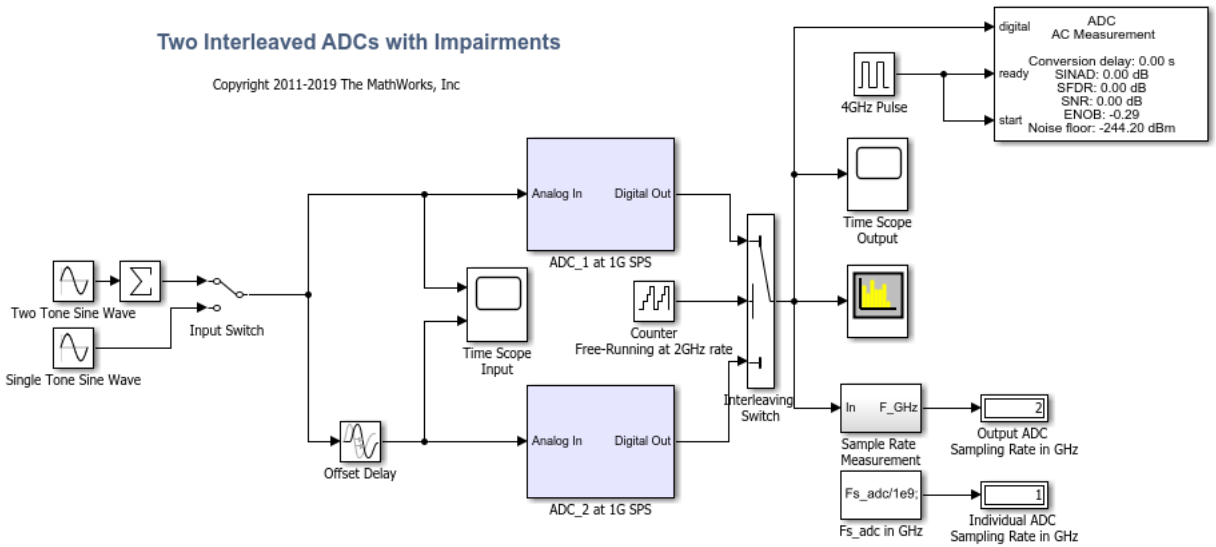
In this example, interleave two simple ADCs based on the model ADC with impairments to create the equivalent of one ADC operating at 2X the individual ADC sampling rate. Use a two-tone test signal at `200` MHz and `220` MHz as the input to verify the distortion introduced by the ADC operation.

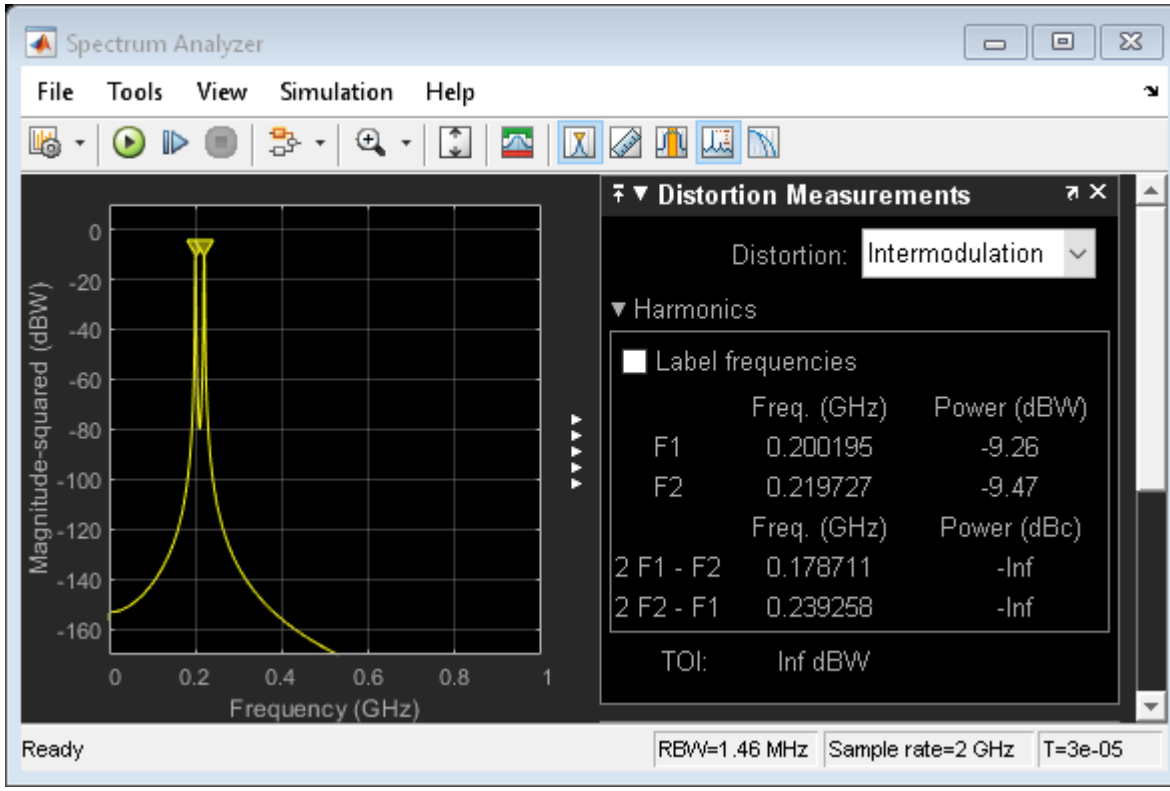
```
model = 'interleaved_adc';
open_system(model)
set_param([model '/ADC_1 at 1G SPS'],'jitter','off');
set_param([model '/ADC_1 at 1G SPS'],'nonlinearity','off');
set_param([model '/ADC_1 at 1G SPS'],'quantization','off');

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)
```


Two Interleaved ADCs with Impairments
Copyright 2011-2019 The MathWorks, Inc
Two Tone Sine Wave
Single Tone Sine Wave
Input Switch
Offset Delay
Analog In
Digital Out
ADC_1 at 1G SPS
ADC_2 at 1G SPS
Time Scope Input
Counter Free-Running at 2GHz rate
Interleaving Switch
4GHz Pulse
digital
ADC AC Measurement
ready
start
Conversion delay: 0.00 s
SINAD: 0.00 dB
SFDR: 0.00 dB
SNR: 0.00 dB
ENOB: -0.29
Noise floor: -244.20 dBm
Time Scope Output
In
F_GHz
Sample Rate Measurement
Output ADC Sampling Rate in GHz
Fs_adc/1e9;
Fs_adc in GHz
Individual ADC Sampling Rate in GHz



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.

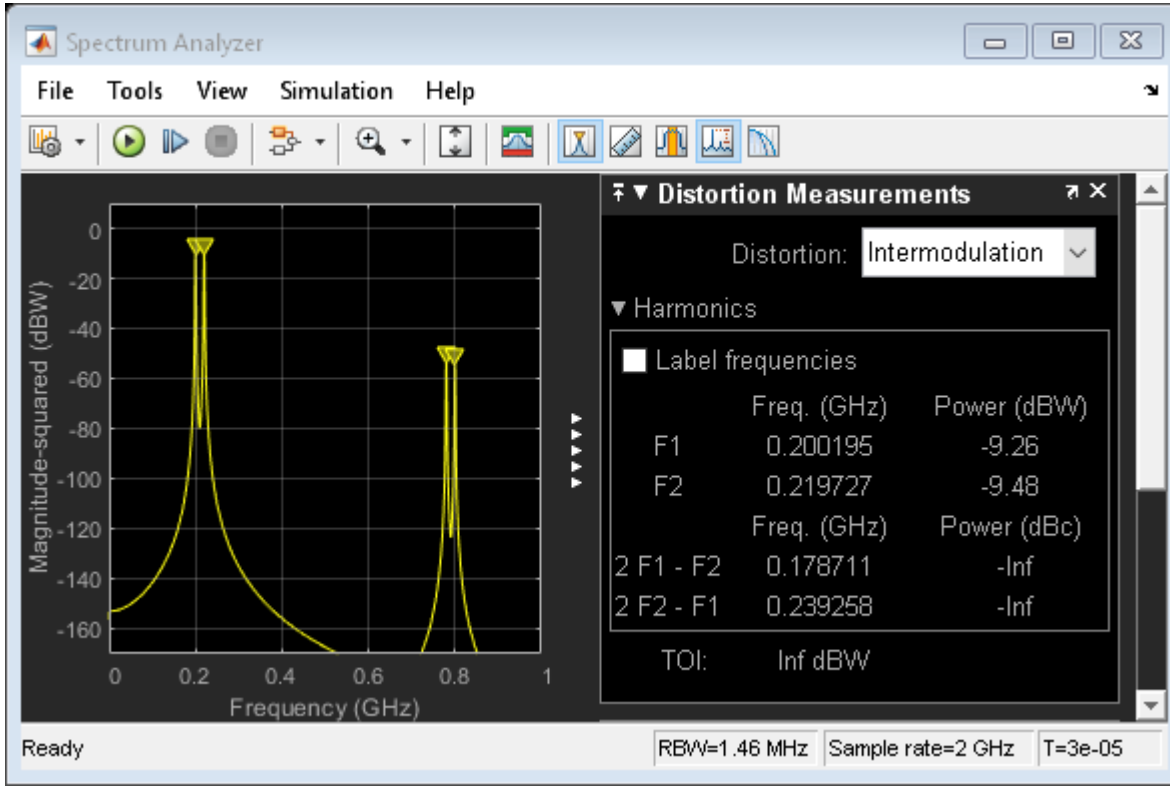
Timing Imperfection

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 SP Devices: spdevices.com/index.php/interleaving

```
sim(model)
```



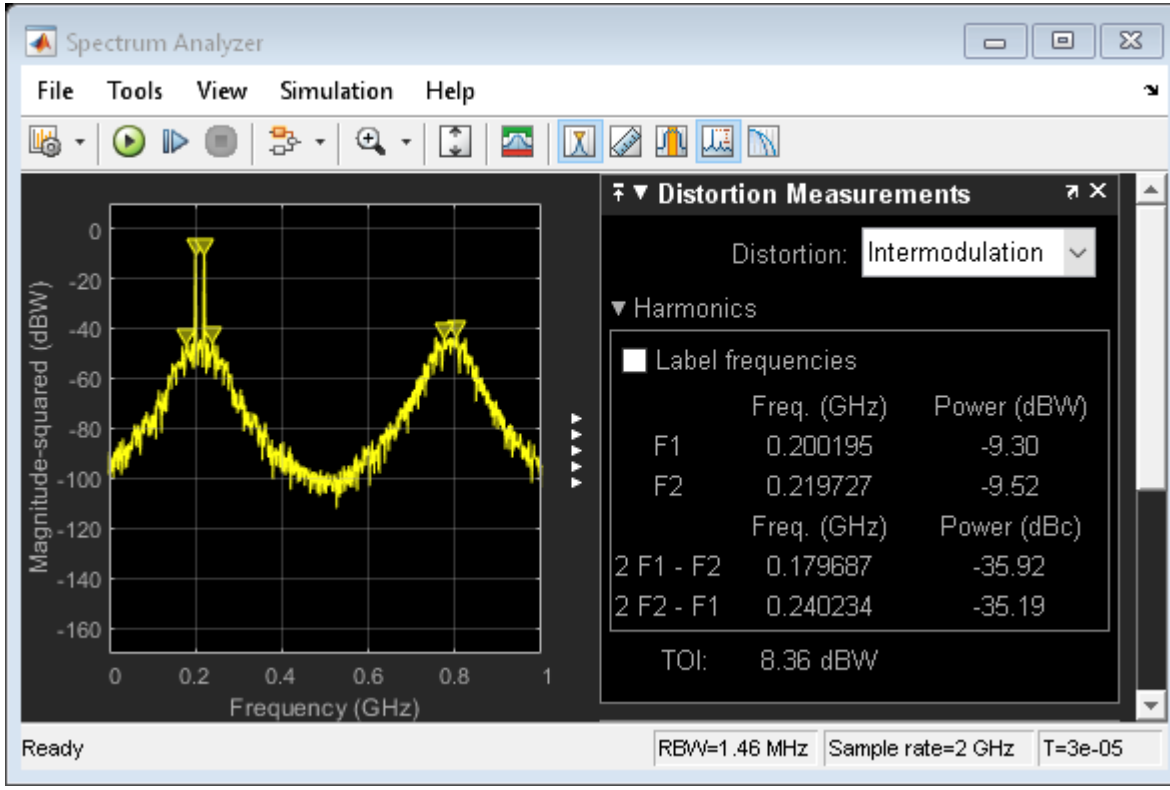
Effect of Aperture Jitter

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)
```



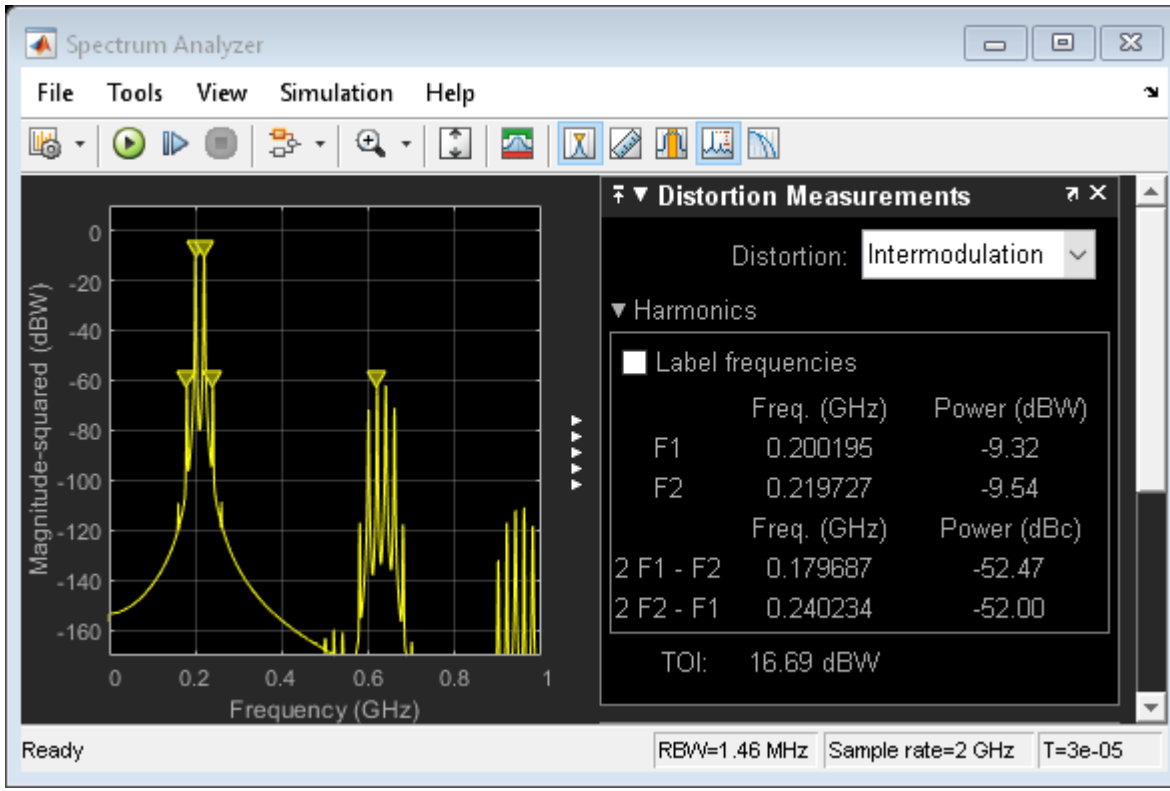
Effect of Nonlinearity

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');
```

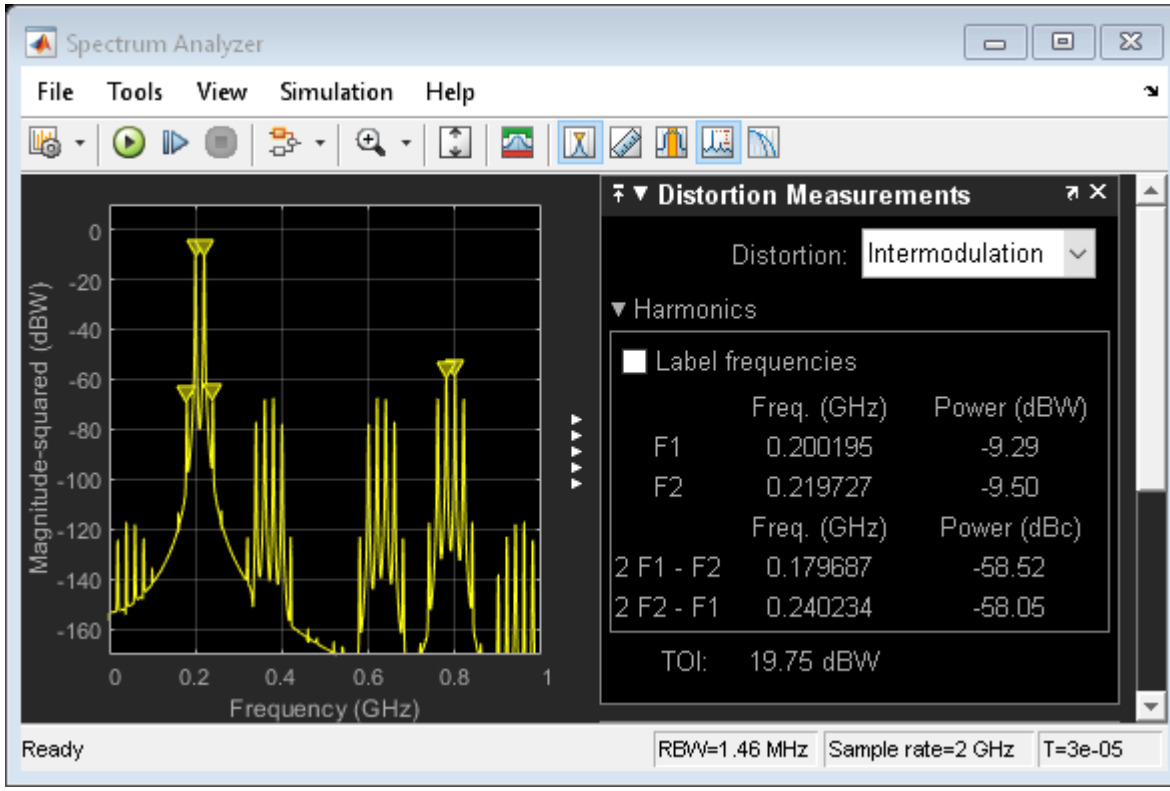
The spectrum now shows 3rd order IMD products around two tones and harmonically related spurs around the 600 MHz region.

```
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)
```



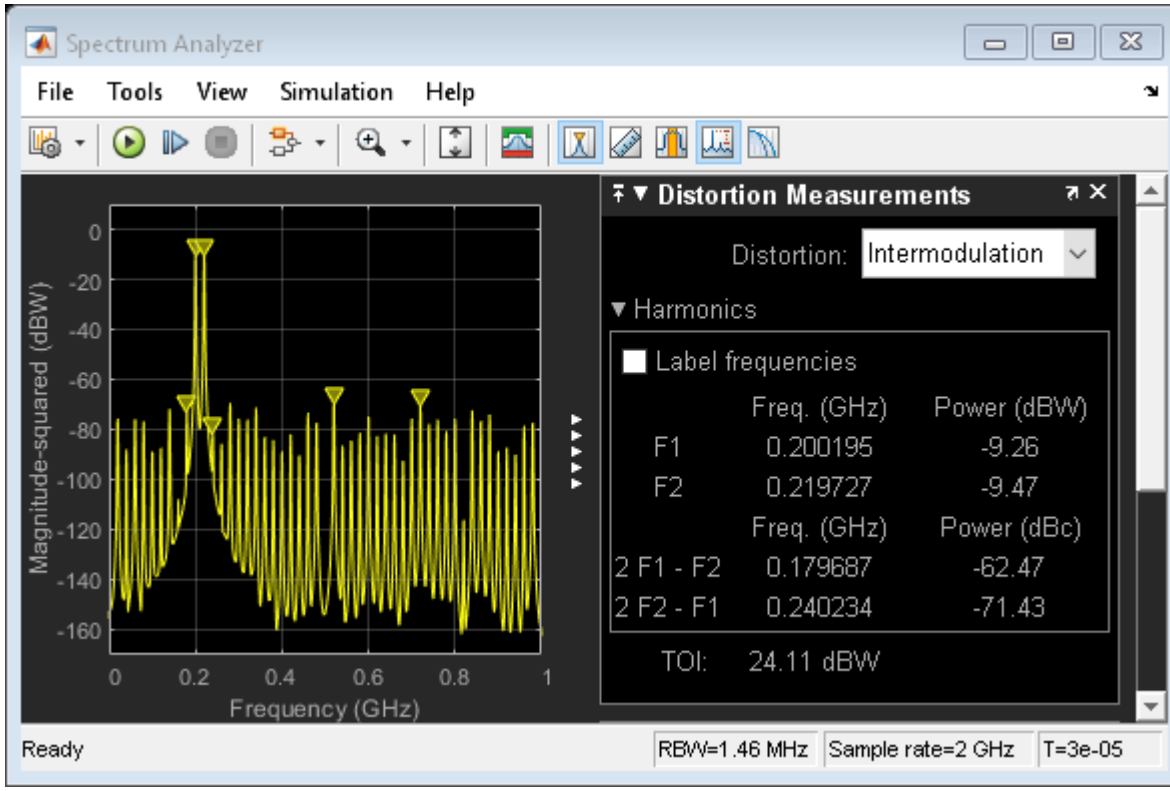
Effect of Quantization and Saturation

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');
```

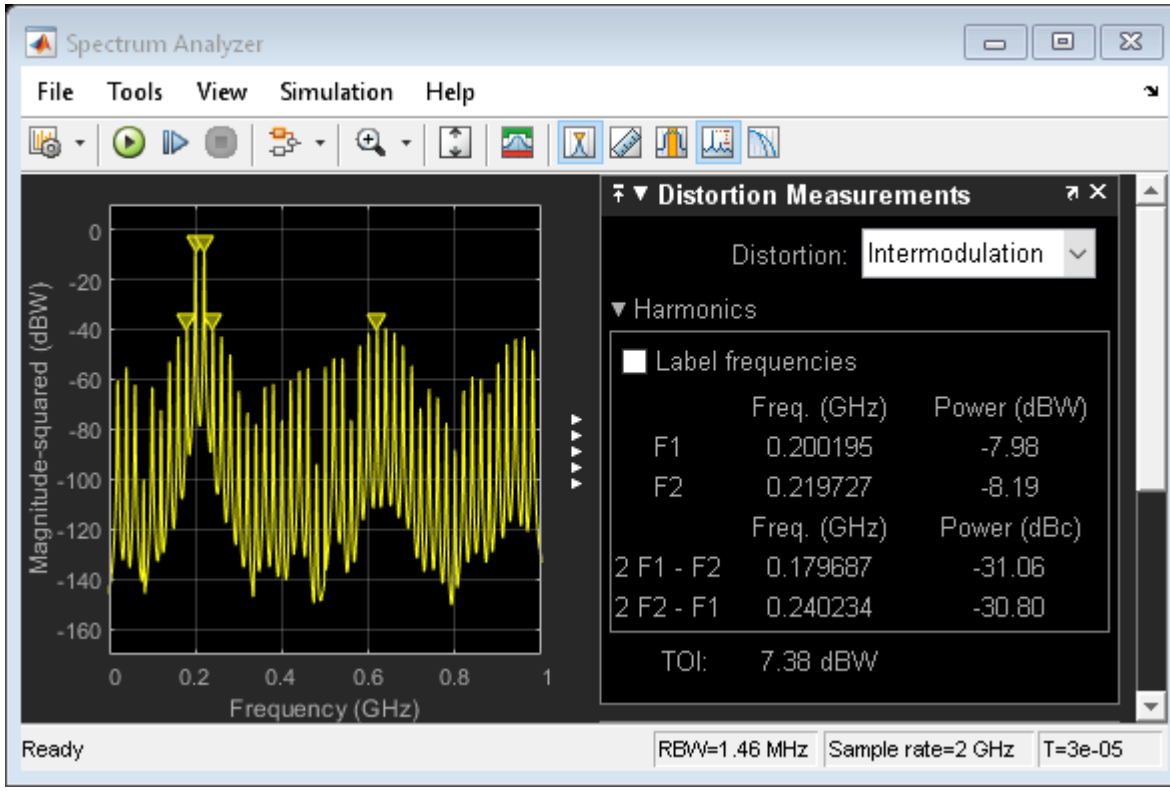
The spectrum shows the noise floor increasing as an effect of quantization.

```
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');
```

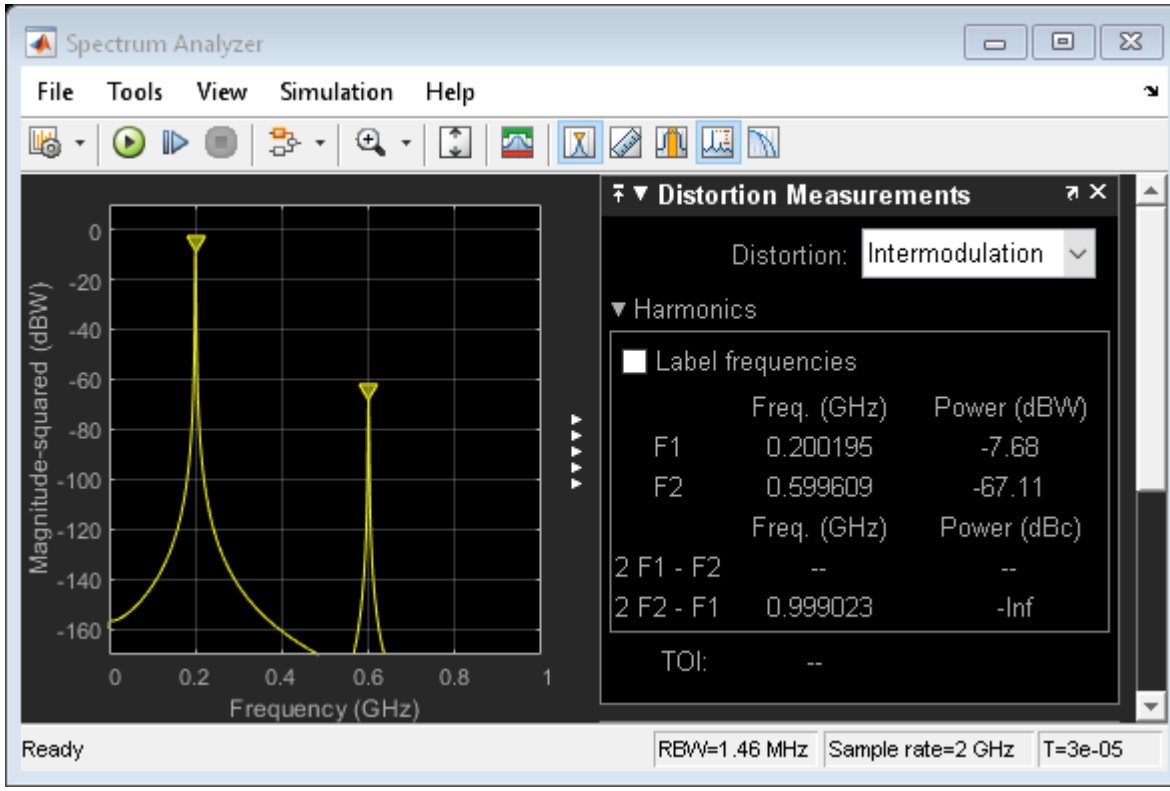
The ADC AC Measurement block from the Mixed-Signal Blockset™ measures conversion delay, SINAD (the ratio of signal to noise and distortion), SFDR, SNR (Signal to Noise Ratio), ENOB and the ADC's output noise floor.

```
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: '316.5751'
          SFDR: '58.7451'
         SINAD: '58.7451'
          ENOB: '9.466'
    NoiseFloor: '-504.6523'
      MaxDelay: '0'
     MeanDelay: '0'
      MinDelay: '0'
```



Design and Evaluate Successive Approximation ADC Using Stateflow

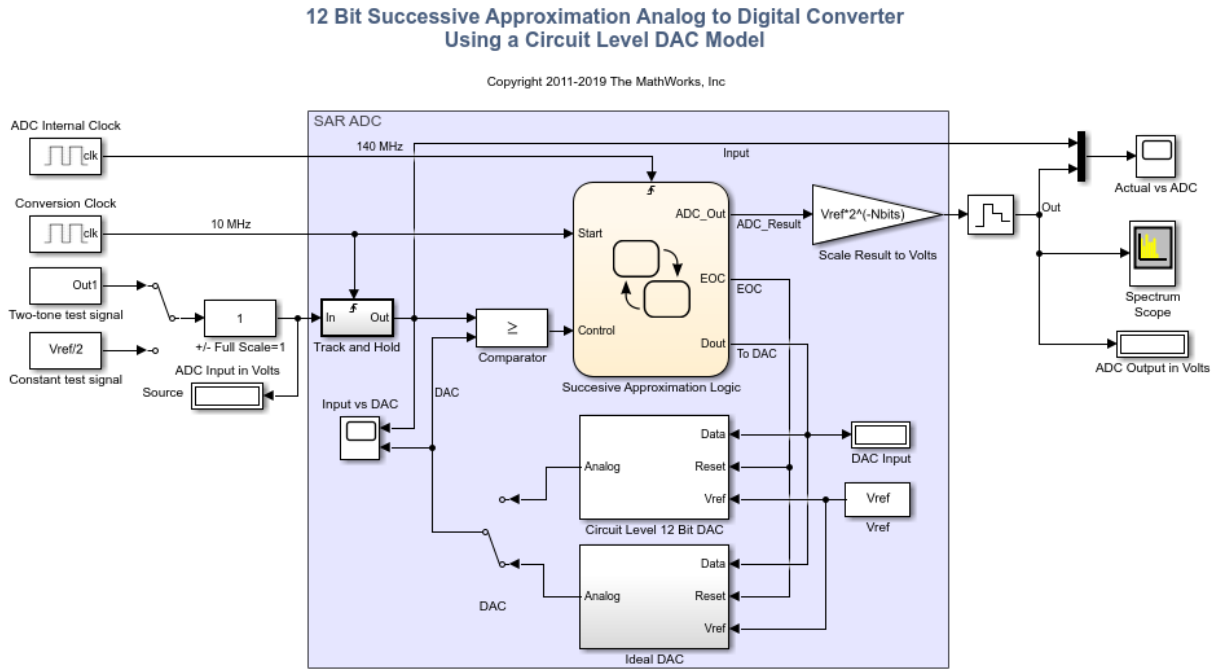
This Successive Approximation Register (SAR) ADC model demonstrates a 12 bit converter with a circuit-level DAC model.

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)
```



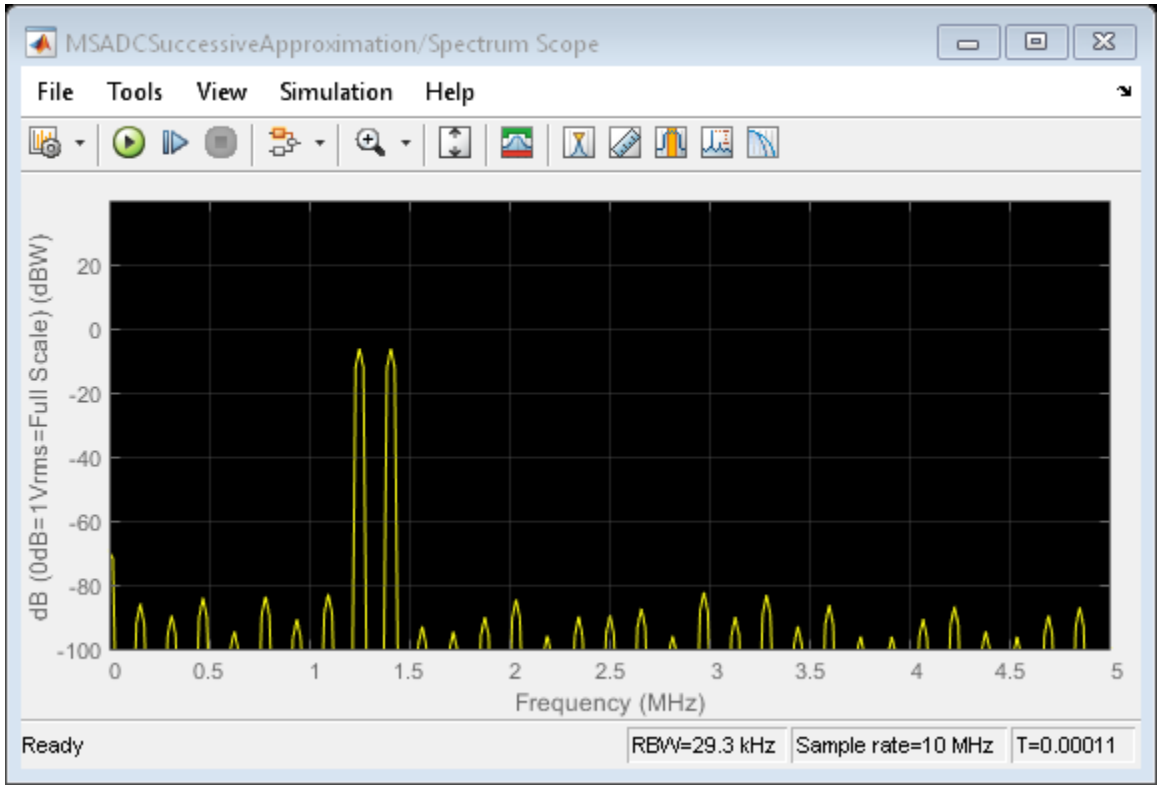
Set the switches to their default positions, selecting the two-tone source and the ideal DAC model.

```
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);
```



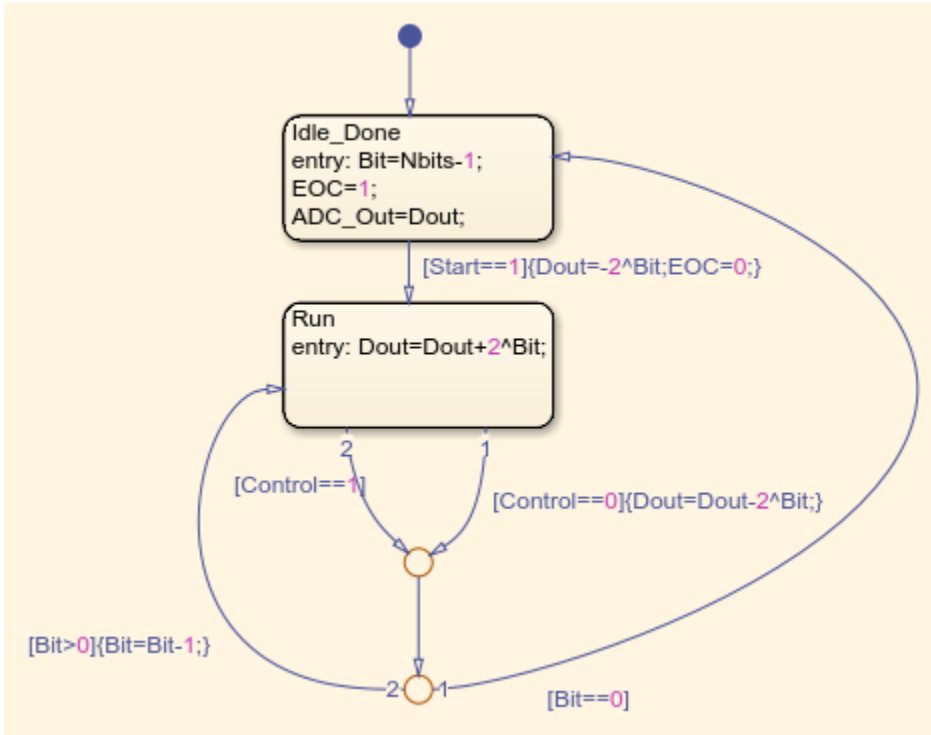
Define the number of bits (*NBits*) and ADC conversion rate (*Fs*) in the MATLAB® workspace. The ADC operating clock rate is determined from *Nbits* and *Fs* .

```
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 '/Succesive 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.

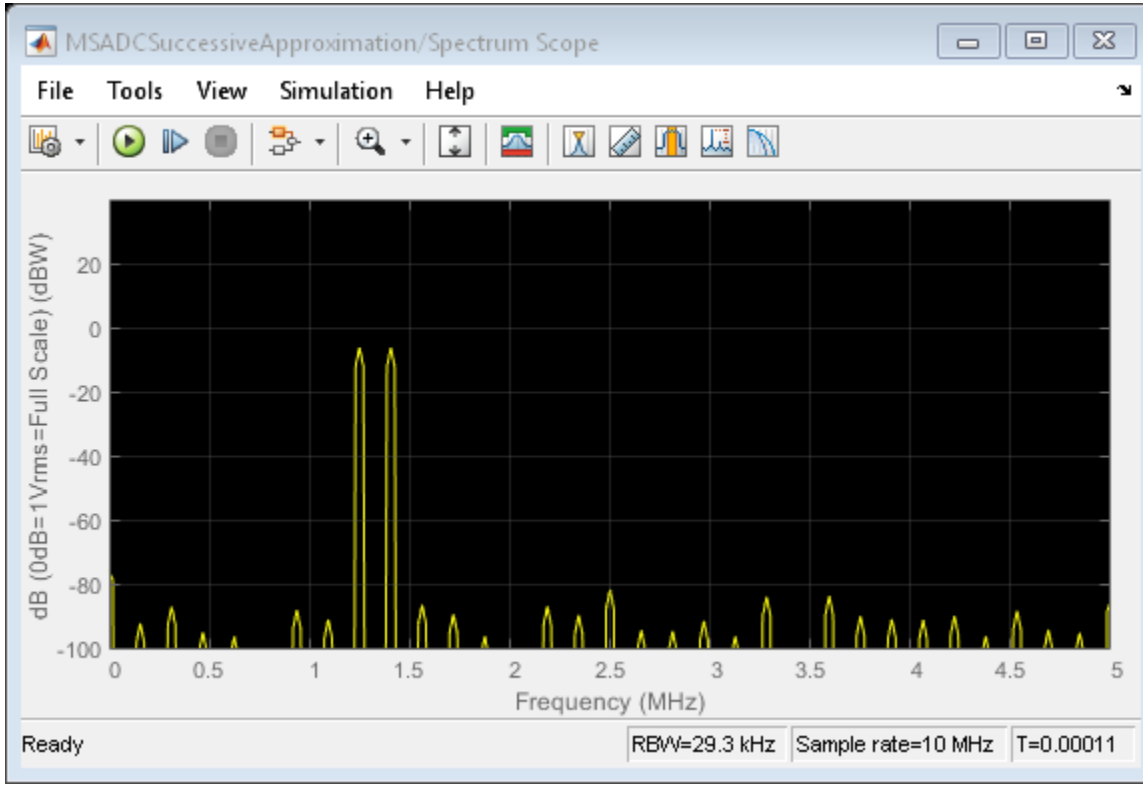
There are two versions of a digital to analog converter (DAC) in this model, one at the circuit-level and the second representing ideal DAC behavior. The ideal DAC block takes the input count and multiplies it by

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

to generate the output comparison voltage [1].

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

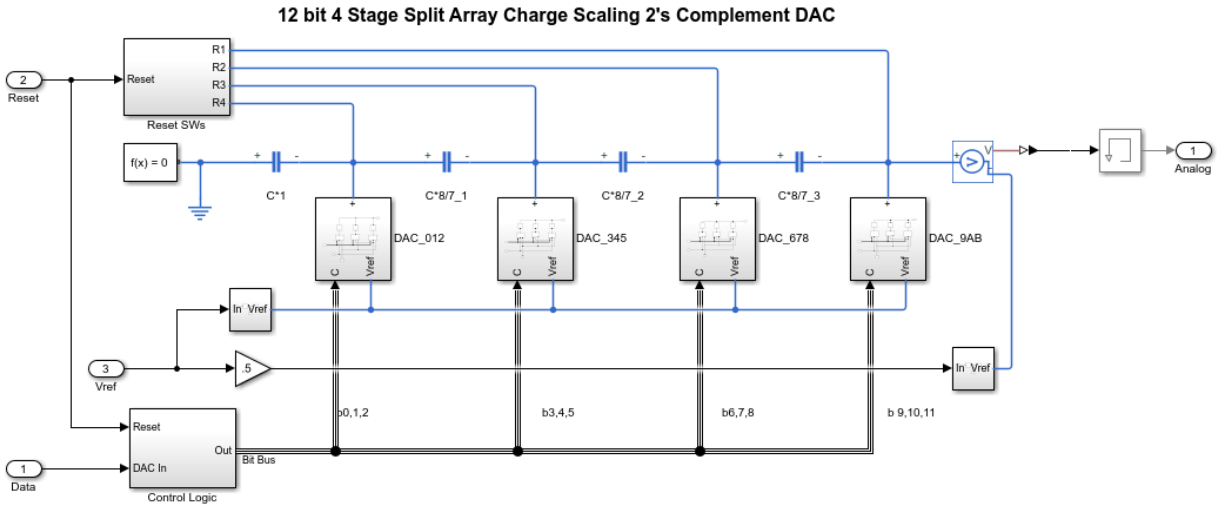
```
set_param([model '/DAC'],'sw','1');
sim(model);
```



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

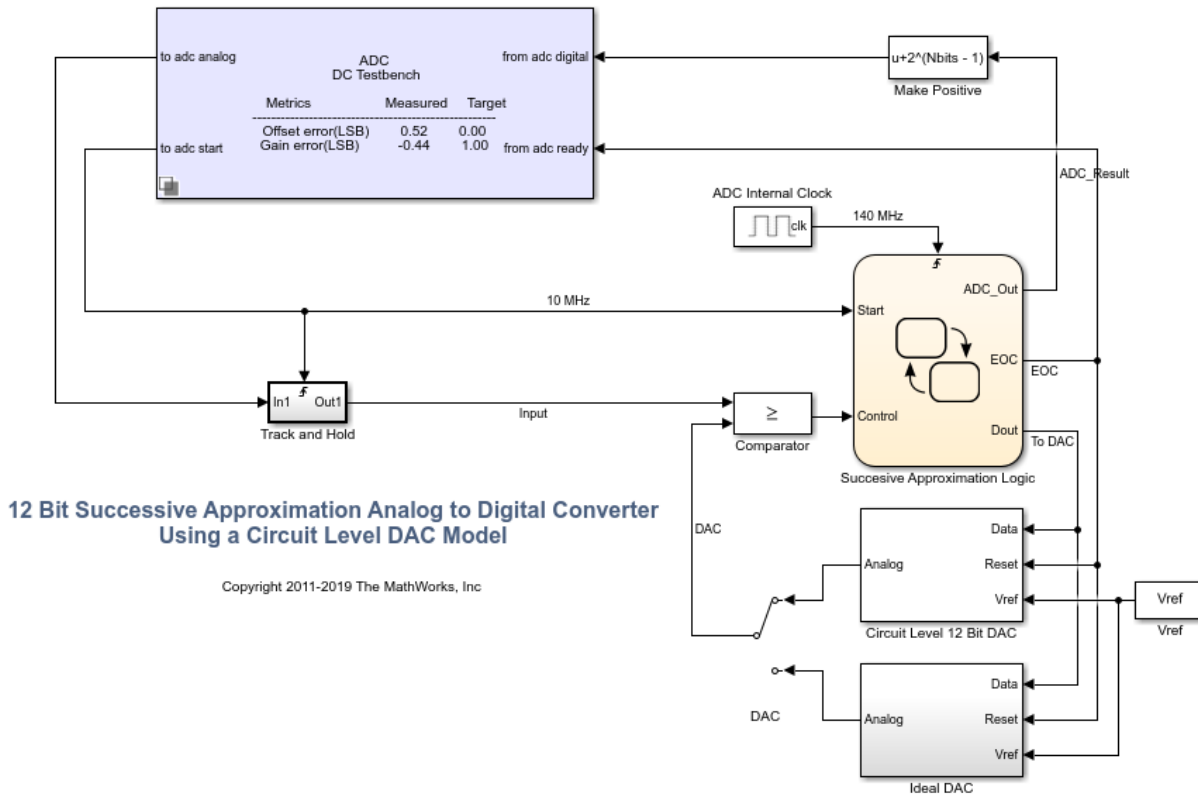
switched to ground rather than data, thus draining the capacitors of charge in preparation for the next approximation. This effectively drains the capacitive network of charge preparing it for the next input sample.

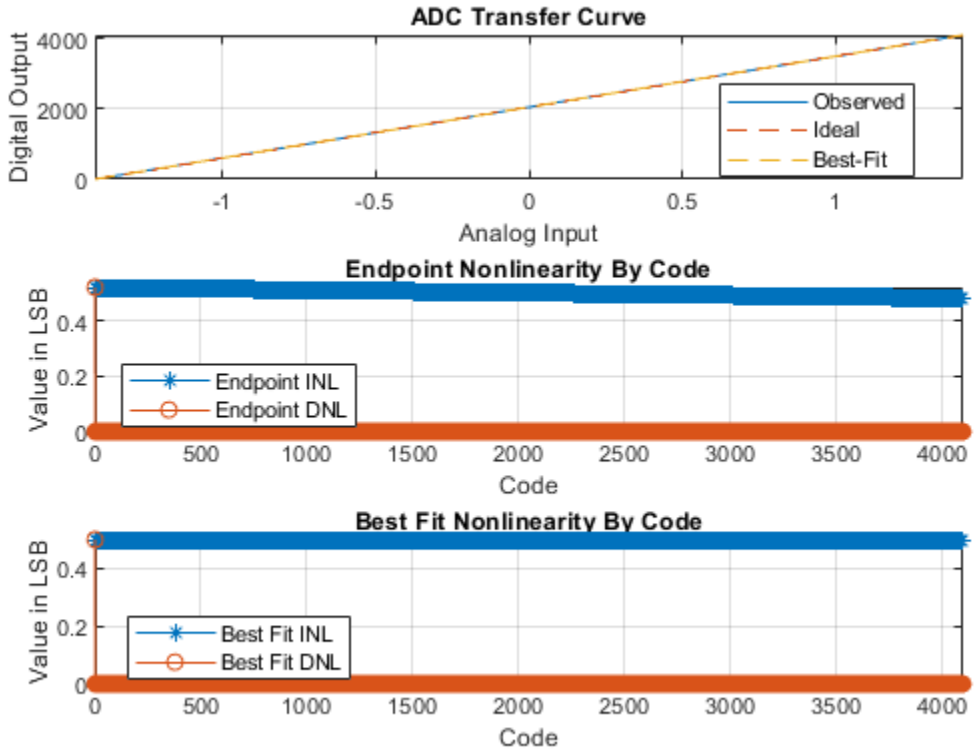
Measurement Testbench

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.

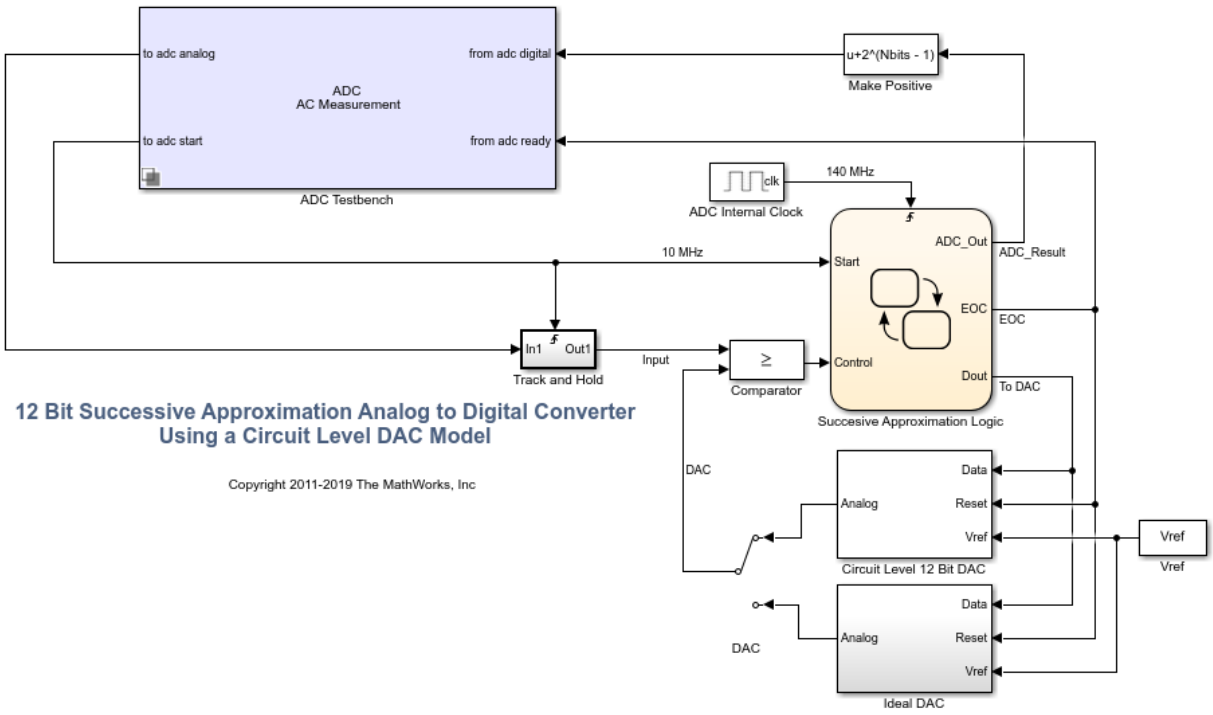
```
bdclose(model);
model = 'MSADCSAR_DC';
open_system(model);
sim(model);
```


to adc analog
ADC
DC Testbench
from adc digital
Metrics
Measured
Target
Offset error(LSB)
0.52
0.00
Gain error(LSB)
-0.44
1.00
to adc start
from adc ready
u+2^(Nbits - 1)
Make Positive
ADC_Result
ADC Internal Clock
clk
140 MHz
10 MHz
ADC_Out
Start
EOC
EOC
In1
Out1
Track and Hold
Input
≥
Comparator
Control
Dout
To DAC
Succesive Approximation Logic
12 Bit Successive Approximation Analog to Digital Converter
Using a Circuit Level DAC Model
Copyright 2011-2019 The MathWorks, Inc
DAC
Data
Analog
Reset
Vref
Circuit Level 12 Bit DAC
Vref
Vref
DAC
Data
Analog
Reset
Vref
Ideal DAC



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');
```



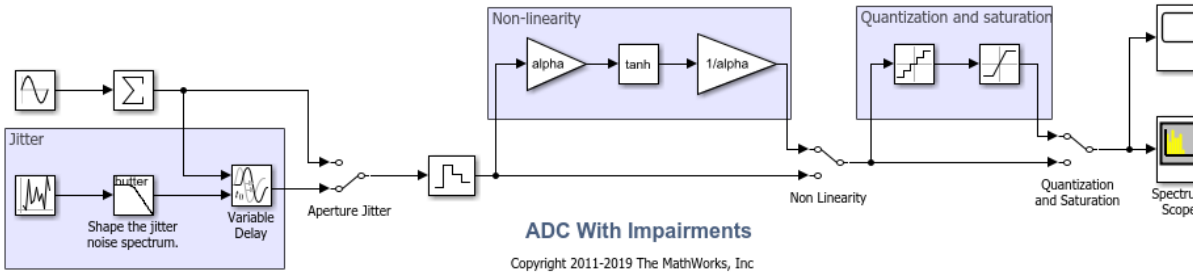
References

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>

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)
```



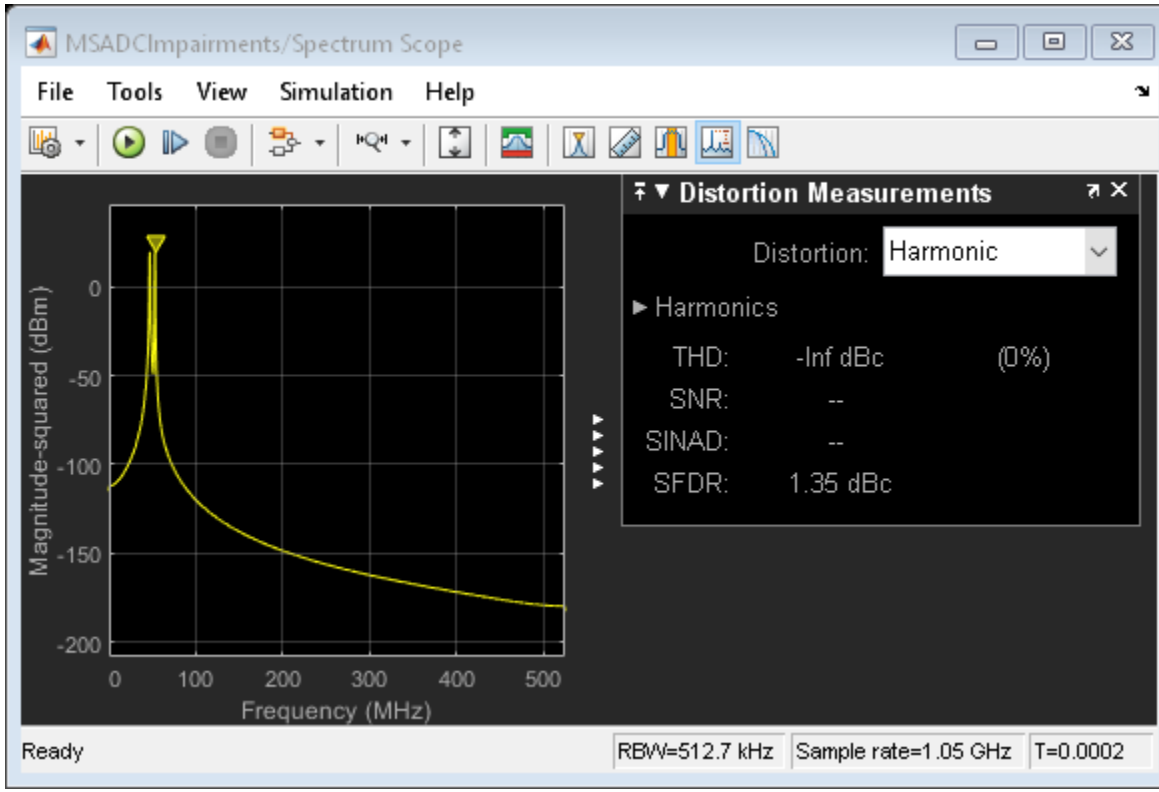
To observe the behavior of an ideal ADC, bypass the impairments using the switches. Set the Sine Wave source to generate two tones as an input signal.

```
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);
```

```
### Building the Accelerator target for model: MSADCImpairments
### Successfully built the Accelerator target for model: MSADCImpairments
```



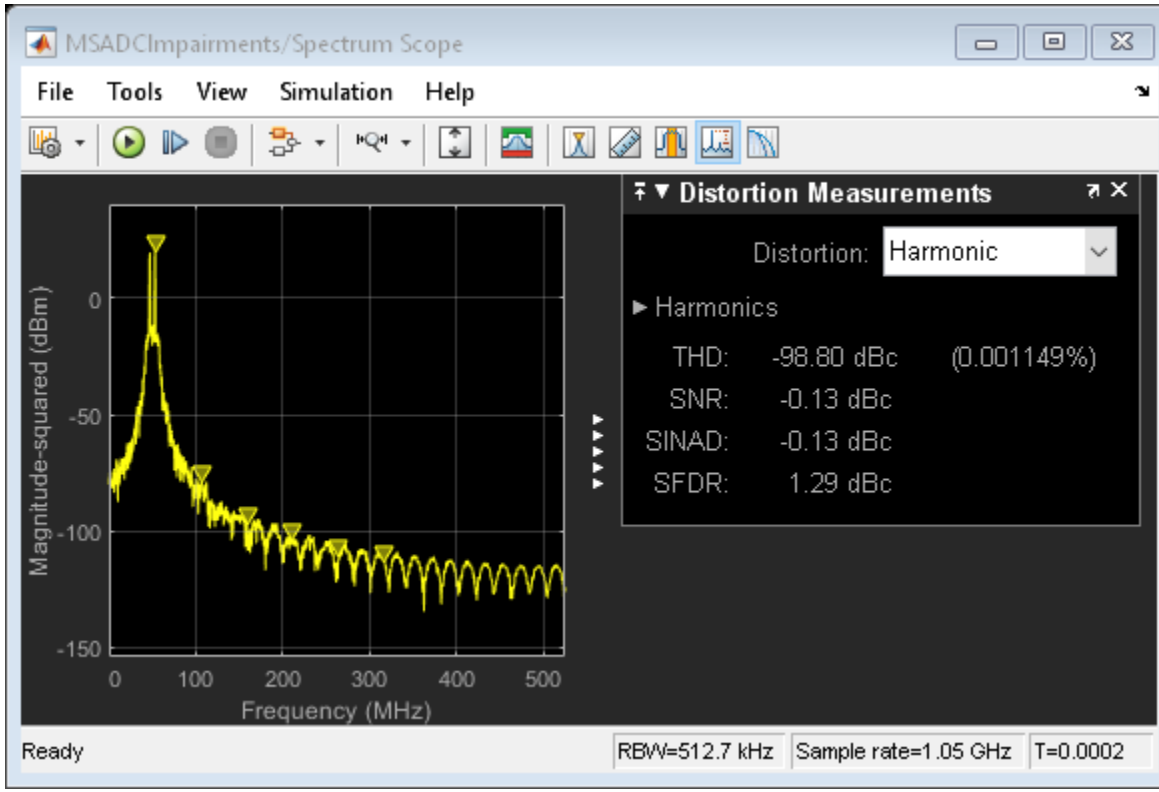
Effect of Aperture Jitter

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);
```



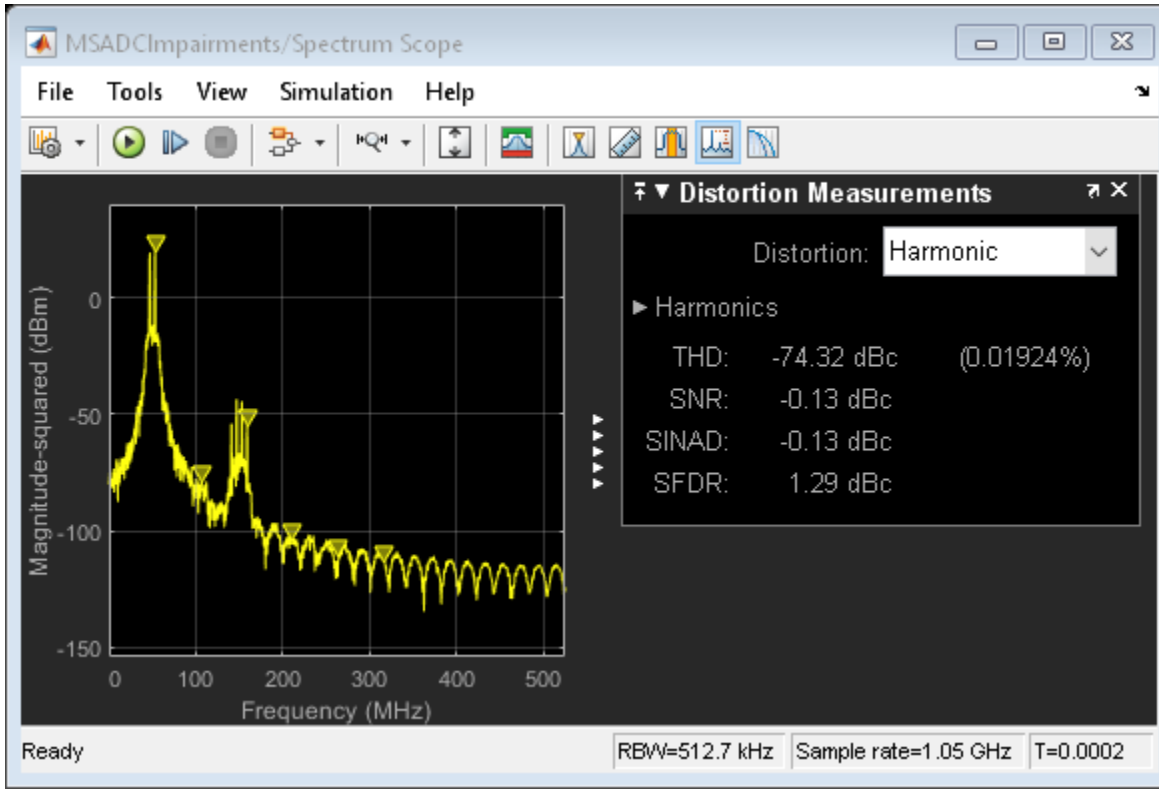
Effect of Nonlinearity

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);
```



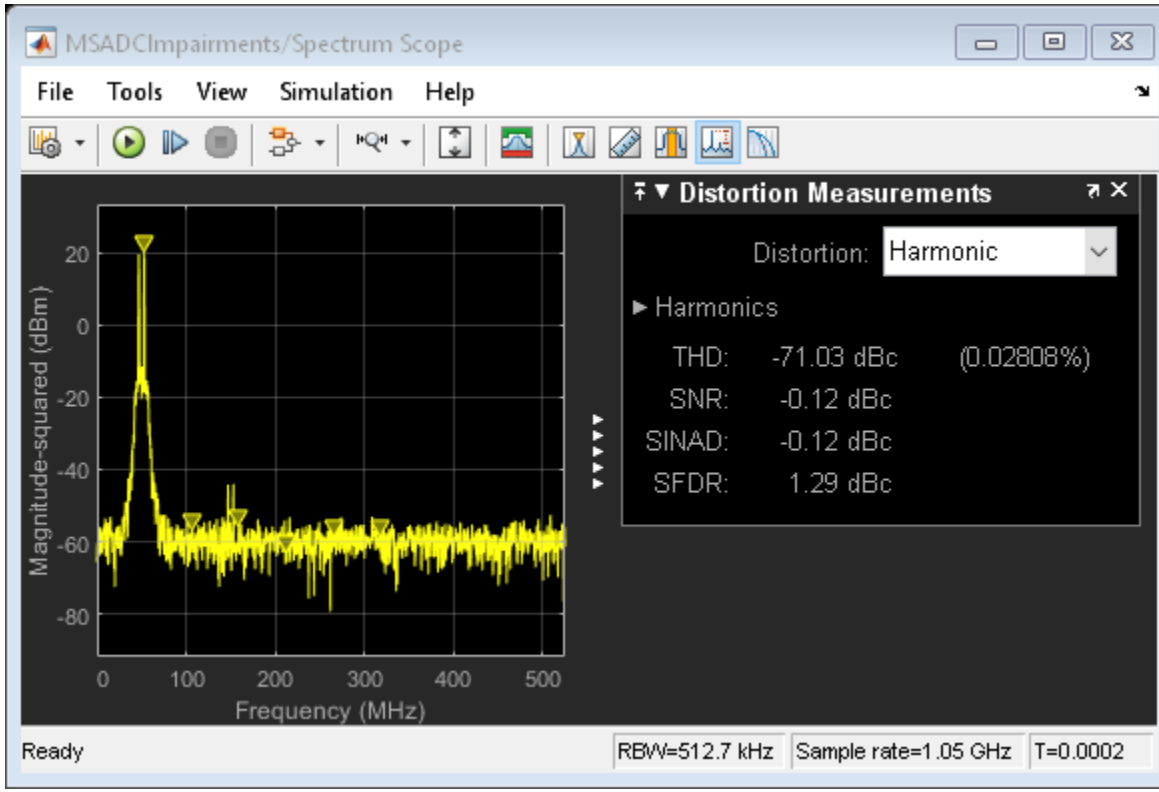
Effect of Quantization and Saturation

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);
```

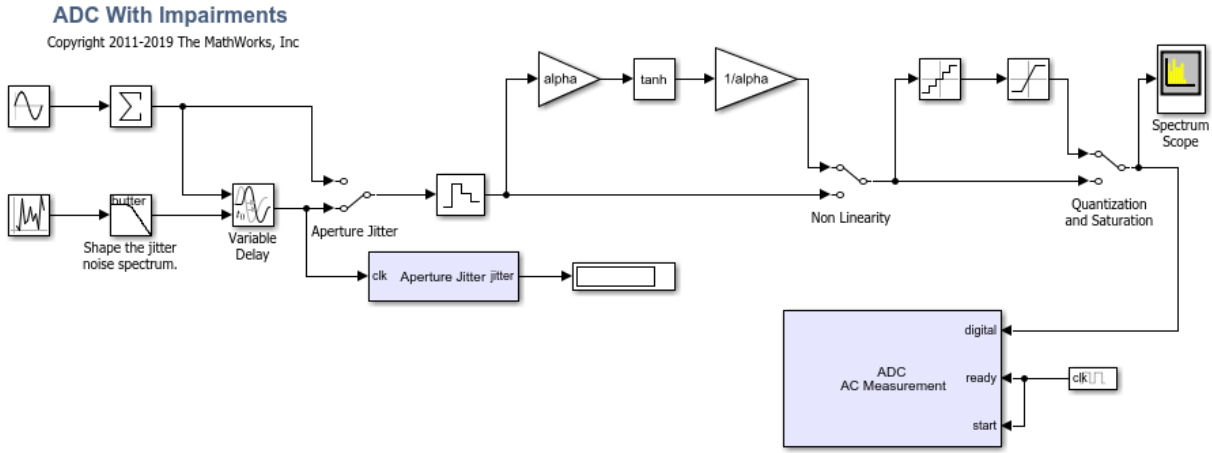


ADC AC Measurements

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.

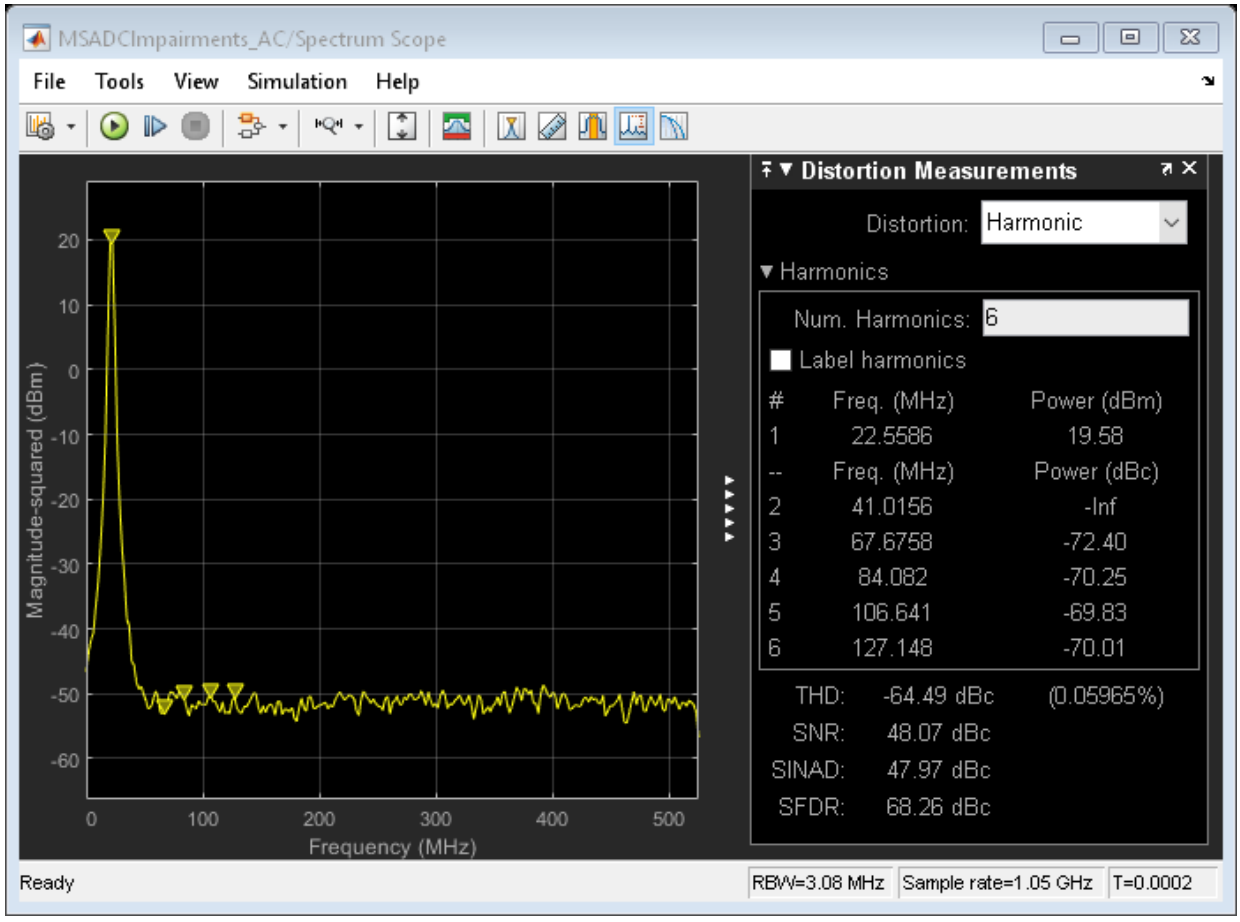
```
bdclose(model);
model = 'MSADCImpairments_AC';
open_system(model);
```




```
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 `50` ps.

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;
```

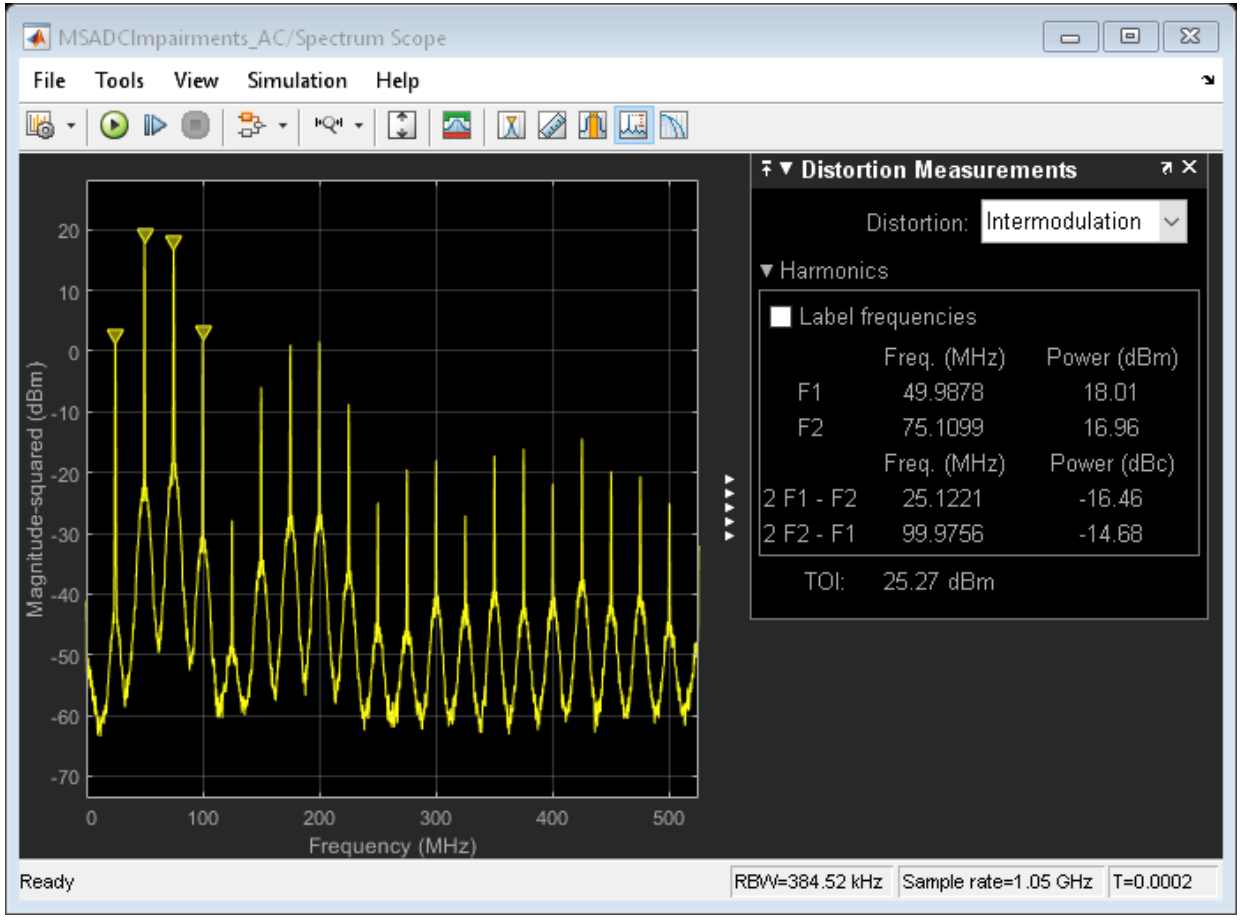
Use a two tone test signal as input to the ADC for the intermodulation measurements.

```
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.

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 block. 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.

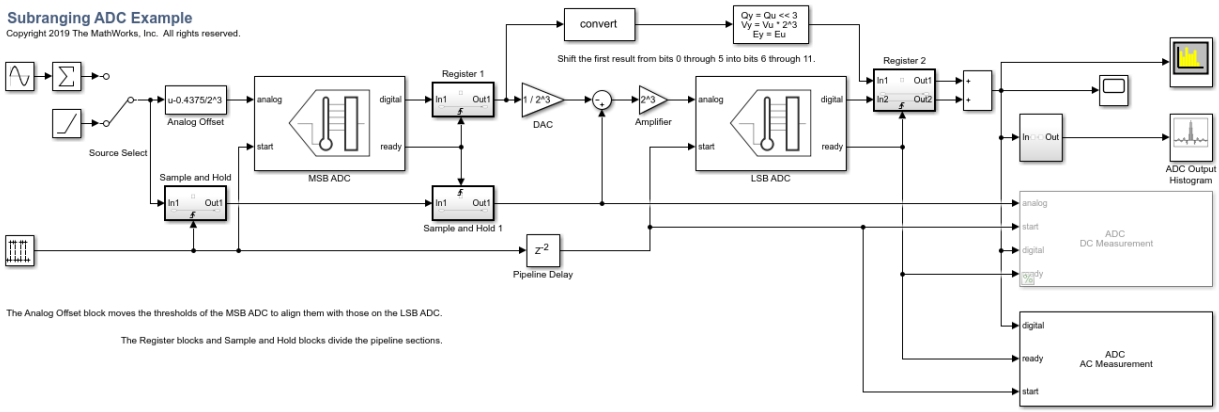
The model is based on the following Analog Devices tutorial: ADC Architectures V: Pipelined Subranging ADCs.

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 quantize this error, 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';
load_system(model);

set_param(model, 'StopTime', '0.001');
set_param([model '/ADC AC Measurement'], 'Commented', 'off');
set_param([model '/ADC DC Measurement'], 'Commented', 'on');
set_param([model '/Subsystem/Buffer'], 'N', '0.001 * 2 * Fs')
set_param([model '/Time Scope'], 'TimeSpan', '1e-6');
set_param([model '/Time Scope'], 'TimeSpanOverrunAction', 'Scroll');

open_system(model);
```



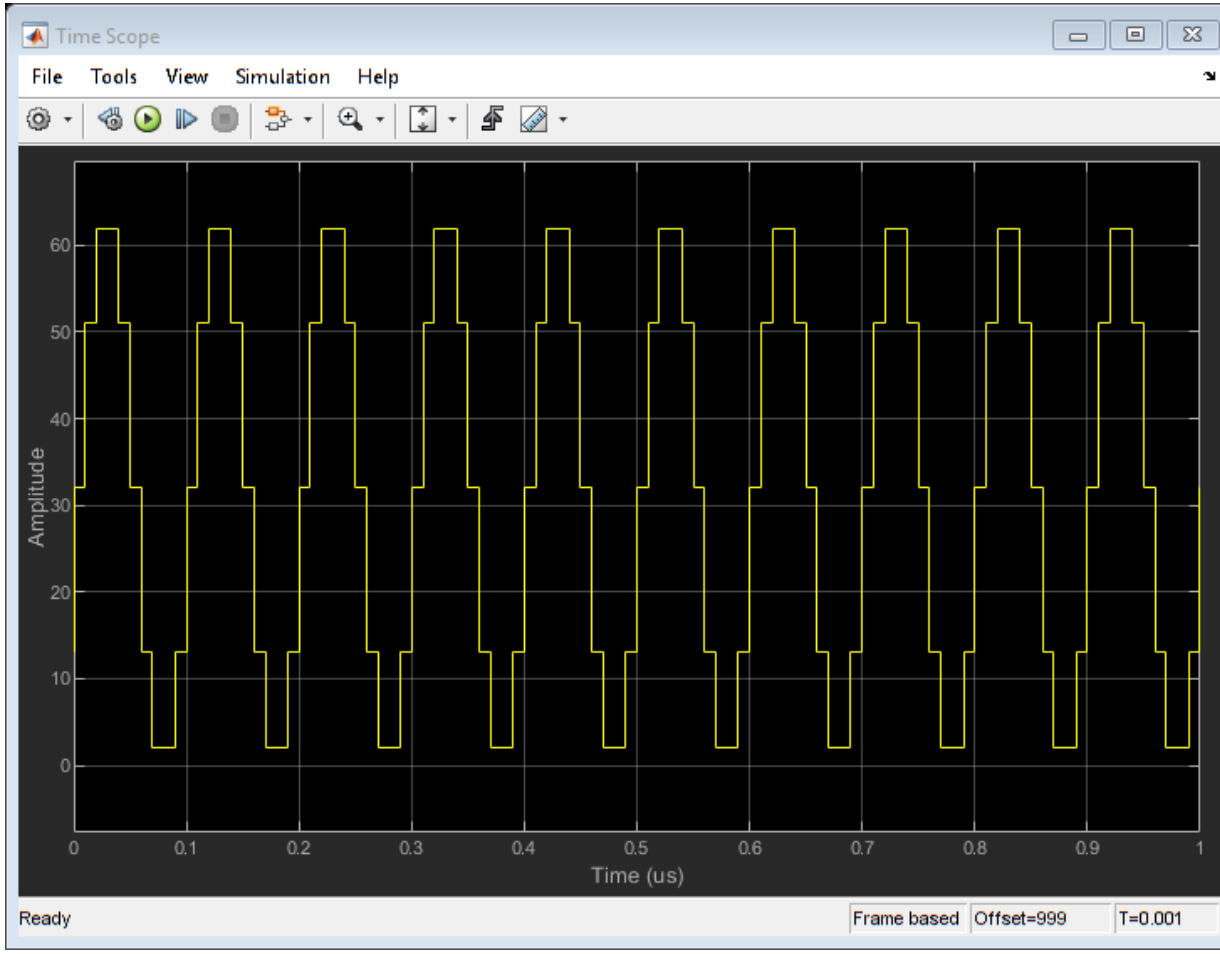
Dynamic Testing

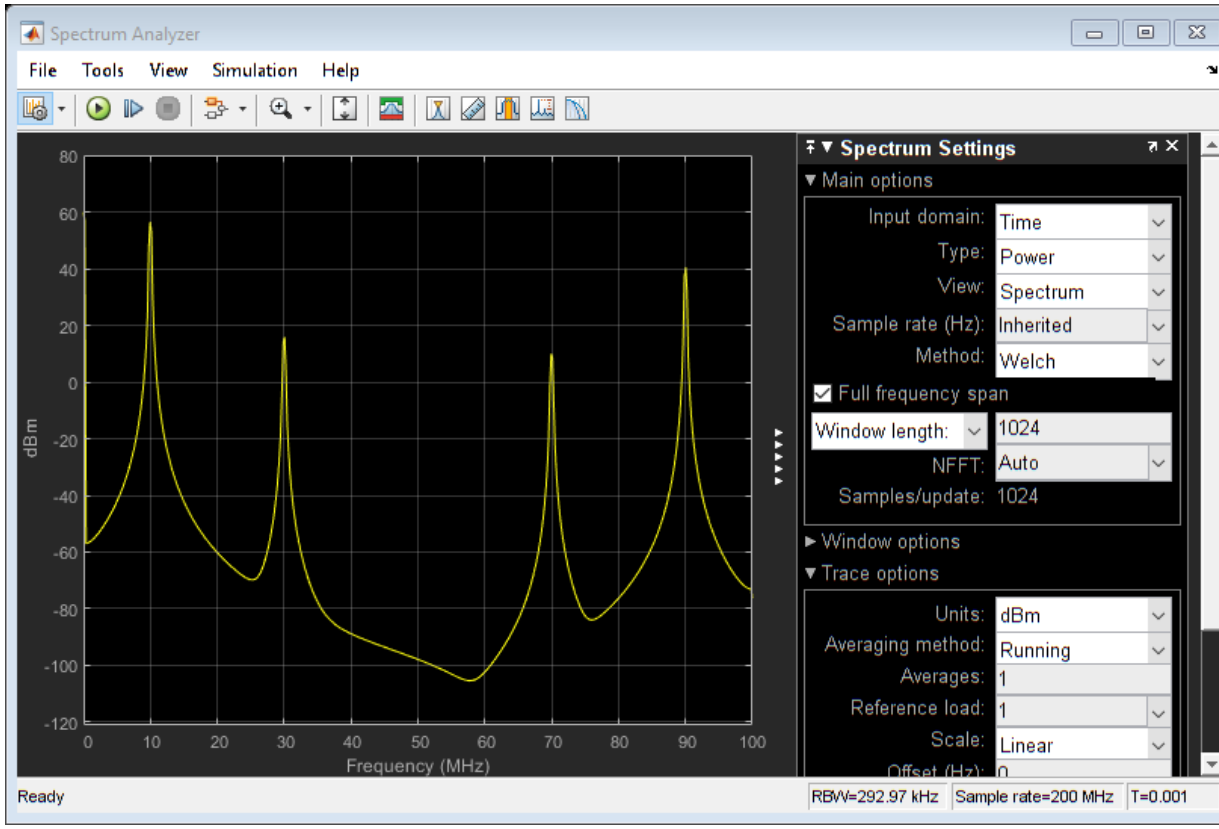
This model uses one of two test sources. Use a sine wave for dynamic testing, e.g. ENoB, SNR, and SFDR. Use a ramp signal for static testing, e.g. missing codes and nonlinearity quantifiers.

To determine SNR, ENOB and other dynamic characteristics of the subranging ADC, use the ADC AC Measurement block from the Mixed-Signal Blockset™. Select he correct switch position to use the Sine Wave as input source.

```
set_param([model '/Source Select'],'sw','1');
set_param([model '/Sine Wave'], 'Frequency', '2 * pi * 10e6');
set_param([model '/Sine Wave'], 'Amplitude', '0.5');
set_param([model '/Sine Wave'], 'Bias', '0.5');

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

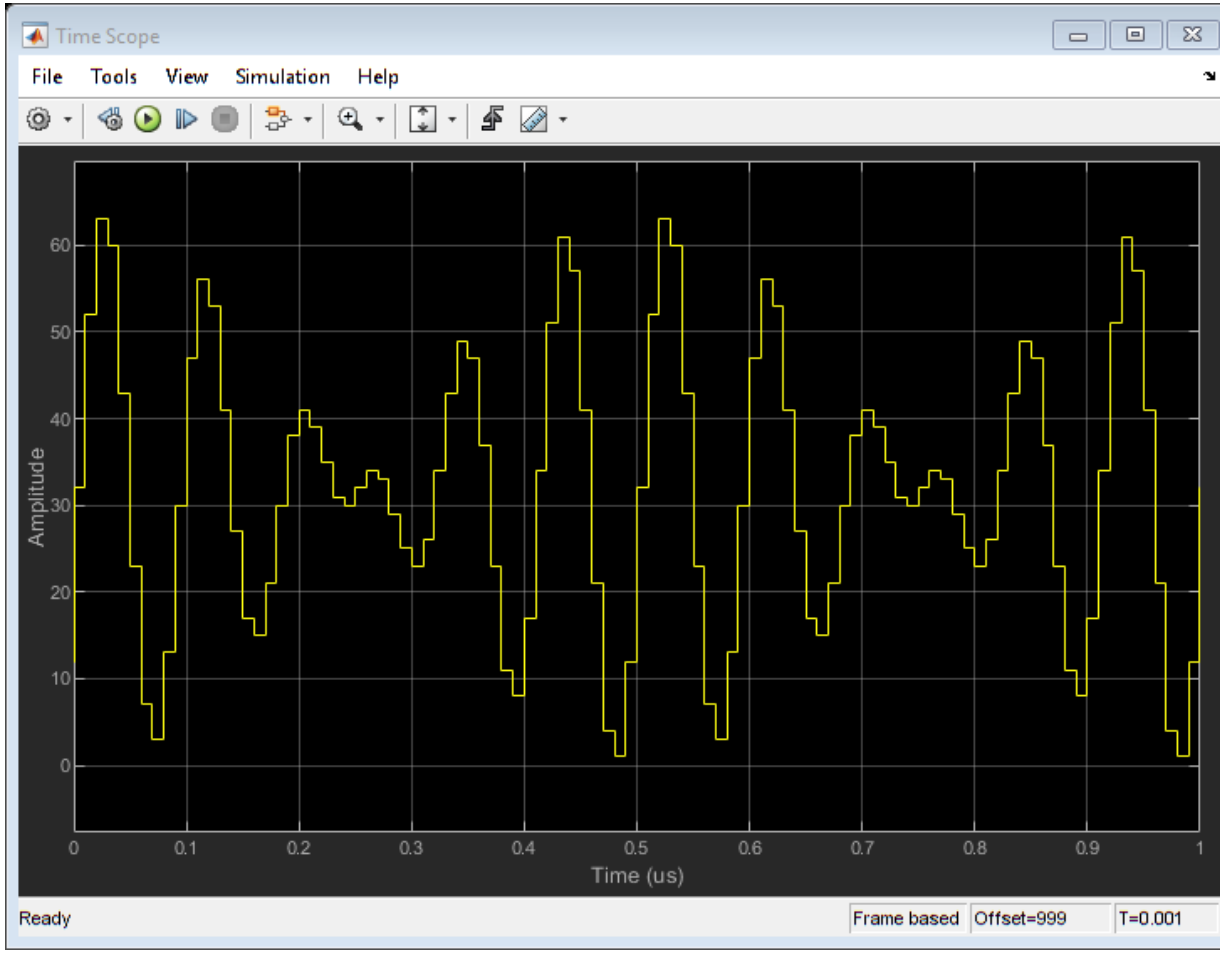

Time Scope
File Tools View Simulation Help
Amplitude
60
50
40
30
20
10
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Time (us)
Ready
Frame based
Offset=999
T=0.001

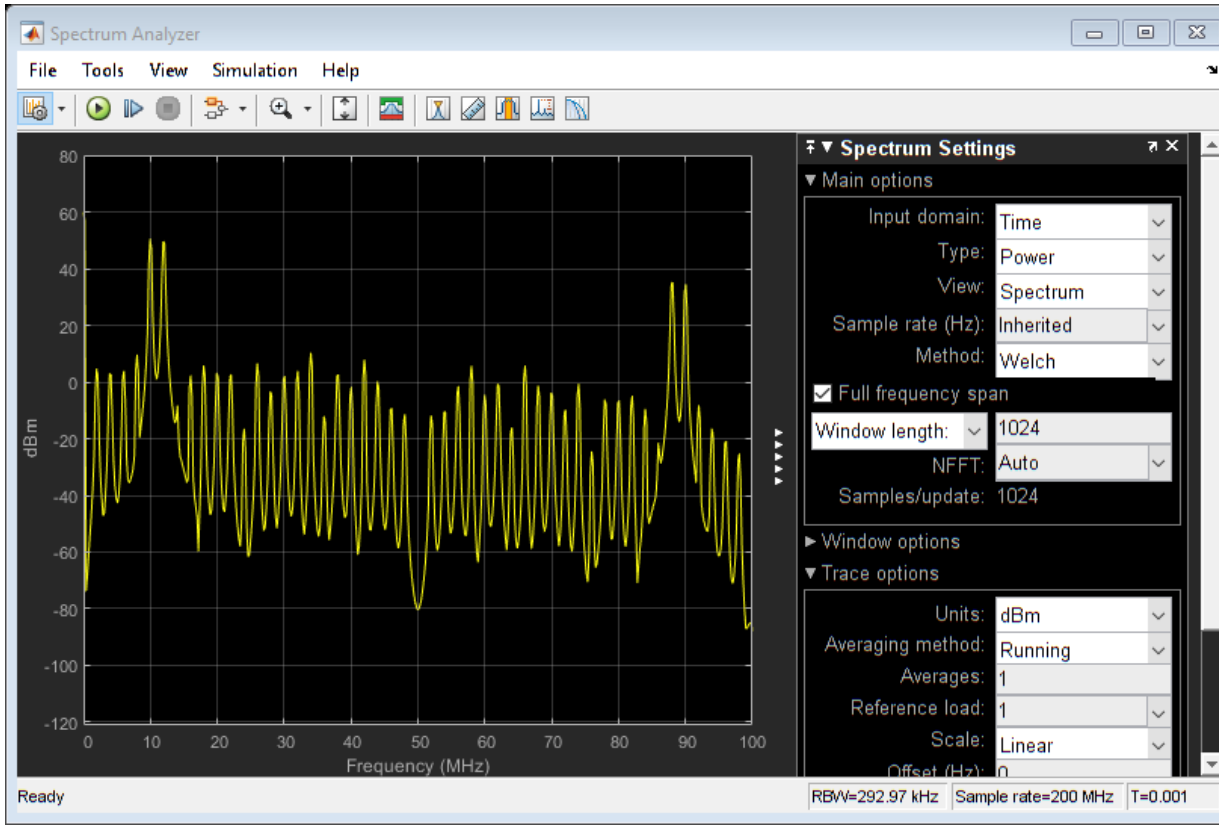


Add another sine wave to the input to test intermodulation distortion. Observe and measure the results using the spectrum analyzer.

```
set_param([model '/Sine Wave'], 'Frequency', '2 * pi * [10, 12] * 1e6');
set_param([model '/Sine Wave'], 'Amplitude', '0.25');
set_param([model '/Sine Wave'], 'Bias', '0.25');

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
Frame based
Offset=999
T=0.001



Missing Code Analysis

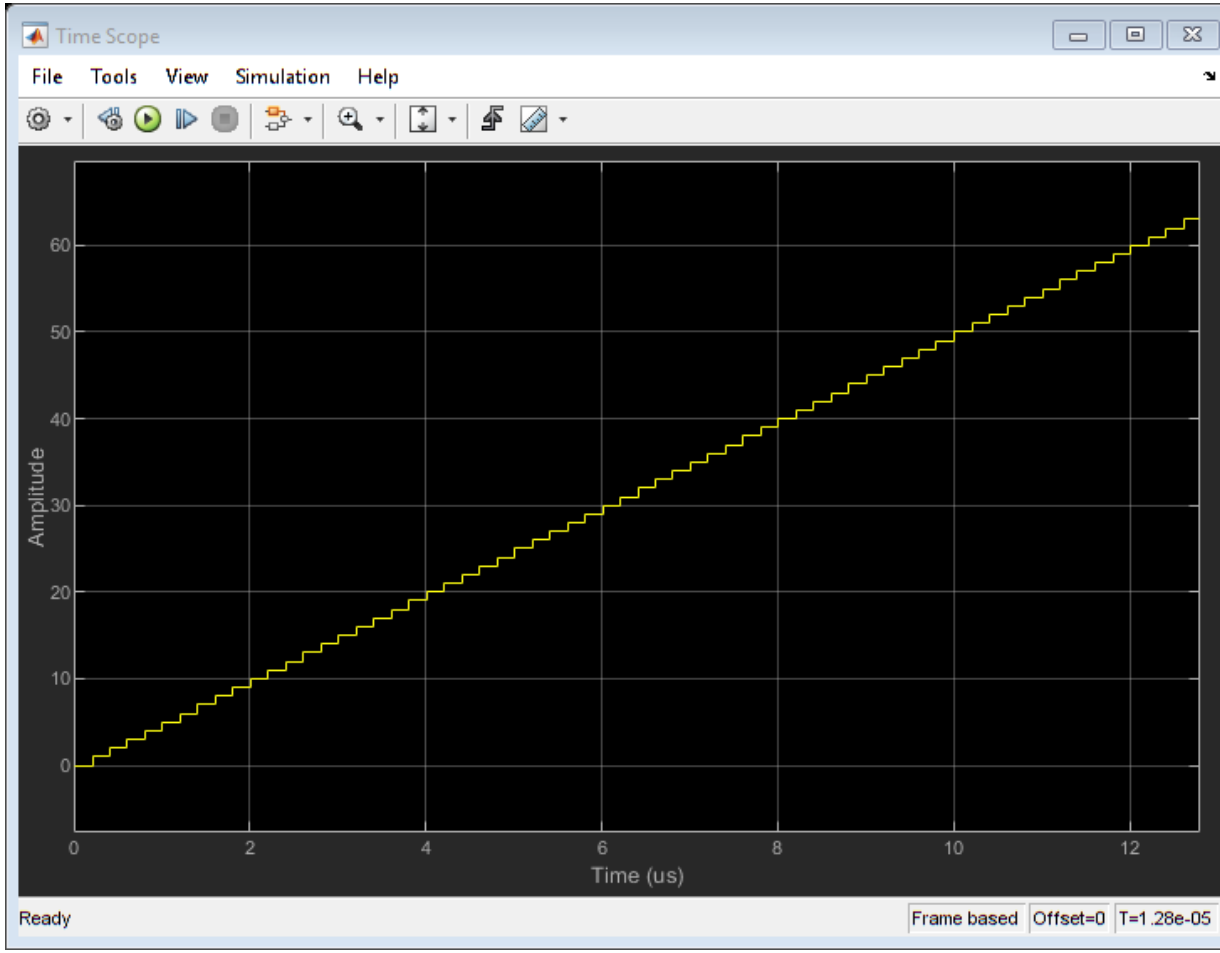
This example uses a histogram block to plot the frequency of occurrence of each ADC output code. Uncomment the histogram by right-clicking on it and selecting Uncomment from the menu. Use the Ramp block as the input source by setting the input switch to the proper position.

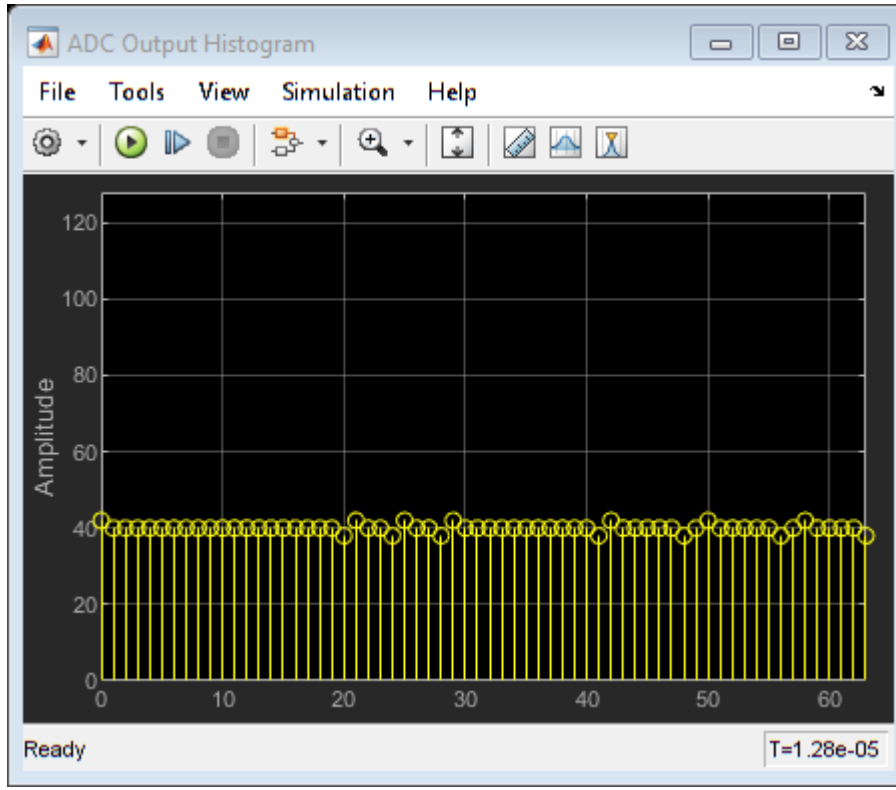
Because there is no need to collect multiple periodograms for DC measurements, change the simulation time to `1.28e-5` s, then comment out the ADC AC Measurement block and the Spectrum Analyzer. To collect quantitative data to go with the histogram, uncomment the ADC DC Measurement block.

```
set_param([model '/Source Select'],'sw','0');
set_param(model, 'StopTime', '1.28e-5');
```

```
set_param([model '/Subsystem/Buffer'], 'N', '1.28e-5 * 2 * Fs')
set_param([model '/ADC AC Measurement'], 'Commented', 'on');
set_param([model '/ADC DC Measurement'], 'Commented', 'off');
w = warning('off', 'msblks:msblksMessages:ADCDataTruncated');
set_param([model '/Time Scope'], 'TimeSpan', '1.28e-5');
set_param([model '/Time Scope'], 'TimeSpanOverrunAction', 'Wrap');

close_system([model '/Spectrum Analyzer']);
open_system([model '/ADC Output Histogram']);
sim(model);
```



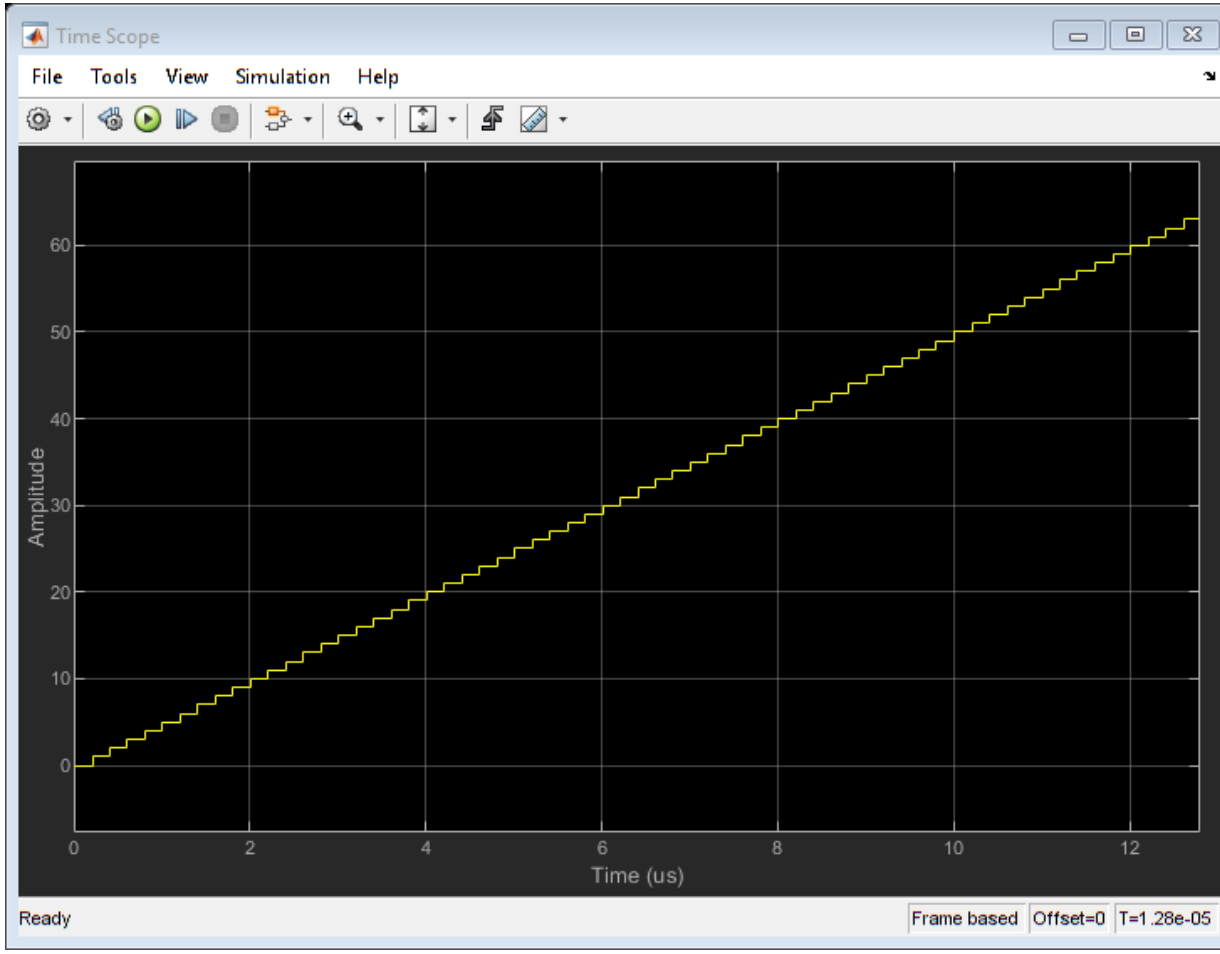


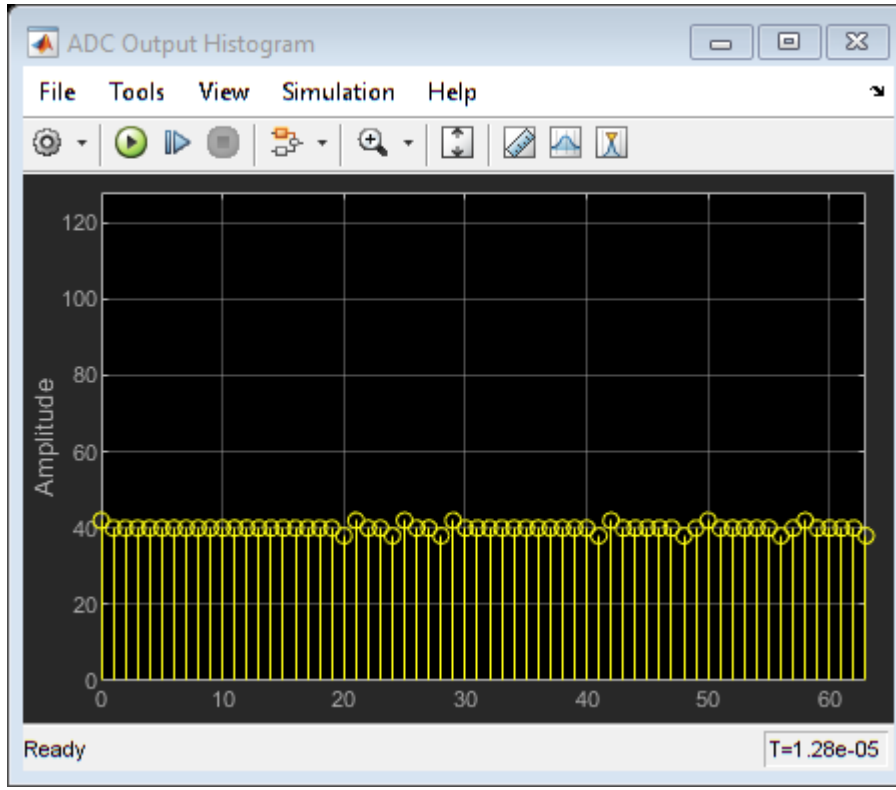
You can observe how the second stage corrects errors from the first by adding **Offset Error** and **Gain Error** in the impairments tab of the MSB ADC. Any fraction of an LSB introduced as a form of gain or offset error results in an imbalance or missing code in the ADC output histogram. Anything other 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.

Errors of up to 1 LSB in the MSB ADC are corrected by the extra bit of the LSB ADC. Larger errors influence the output.

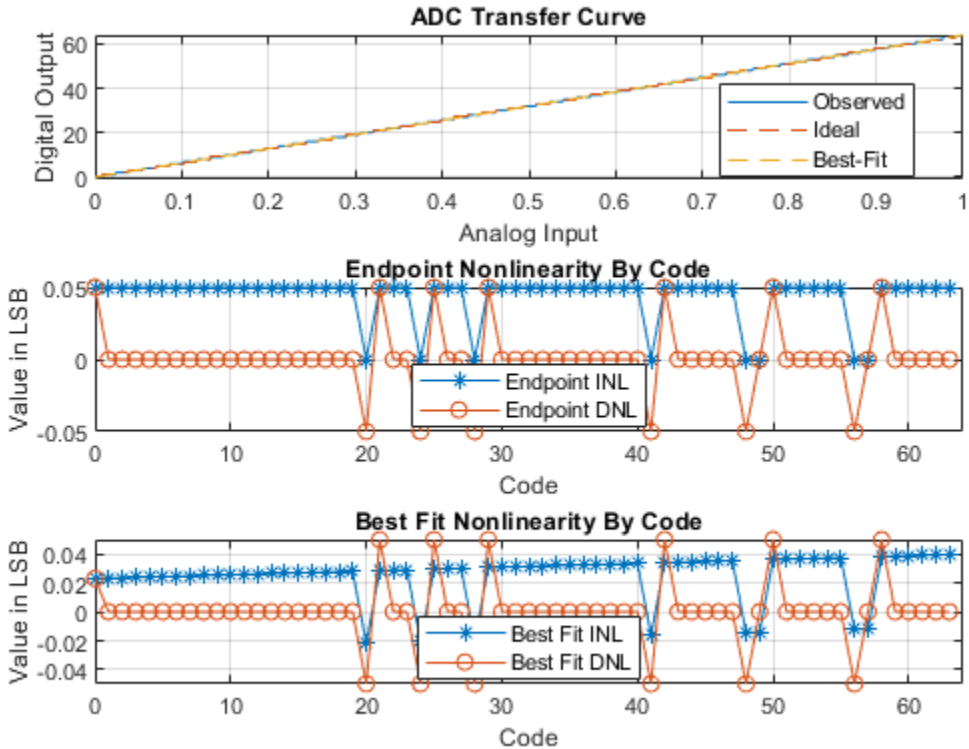
```
set_param([model '/MSB ADC'],'OffsetError','0.5');
set_param([model '/MSB ADC'],'GainError','0.3');

sim(model);
```


Time Scope
File
Tools
View
Simulation
Help
Amplitude
60
50
40
30
20
10
0
0
2
4
6
8
10
12
Time (us)
Ready
Frame based
Offset=0
T=1.28e-05



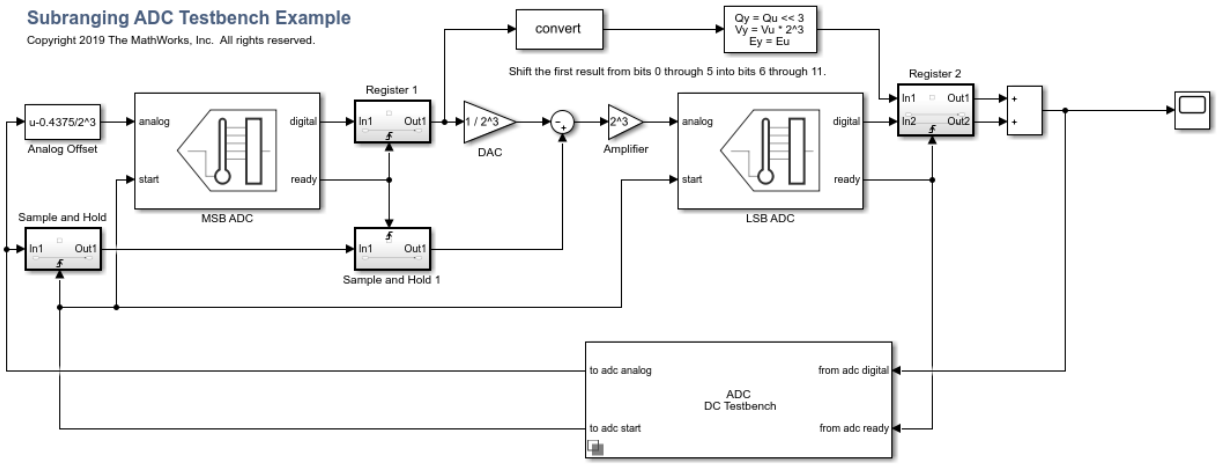
View the results of the ADC DC Measurement block by clicking the Plot measurement results button in the block mask.



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.

```
model = 'subranging_adc_testbench';
open_system(model);
warning(w.state, 'msblks:msblksMessages:ADCDataTruncated');
```



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:

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

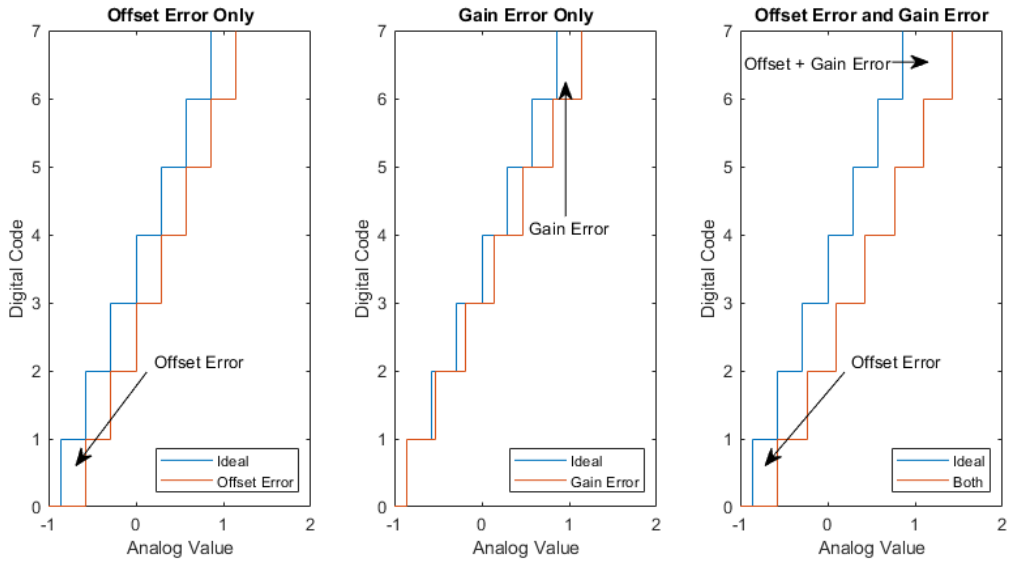
Calculate Offset Error and Gain Error

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 E
plotAdcTcForExample(TC0Digital, TC0Analog, TC1Analog, TC2Analog, TC3Analog, Nbits);
```

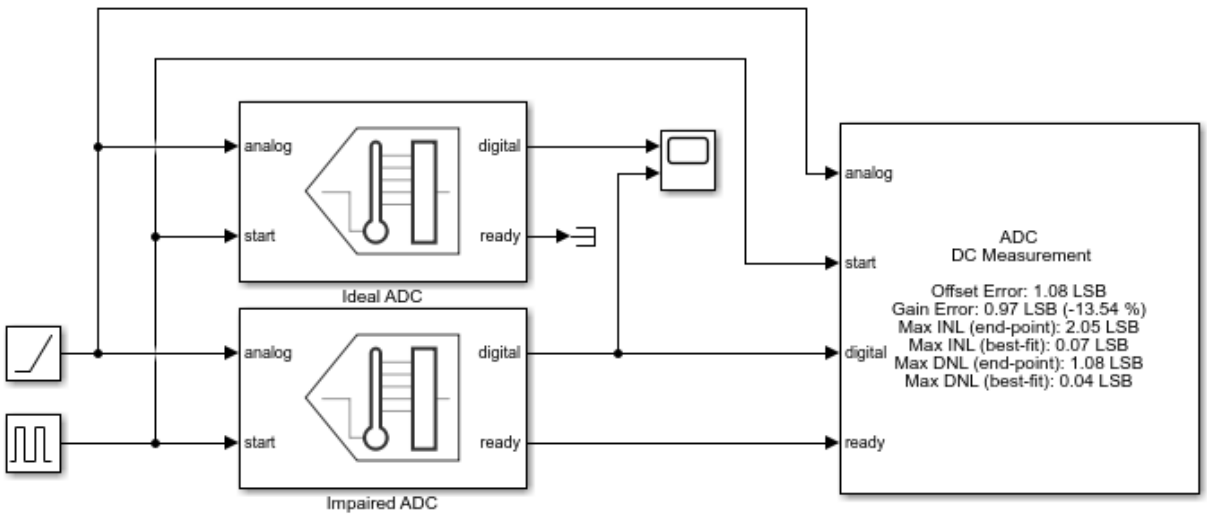


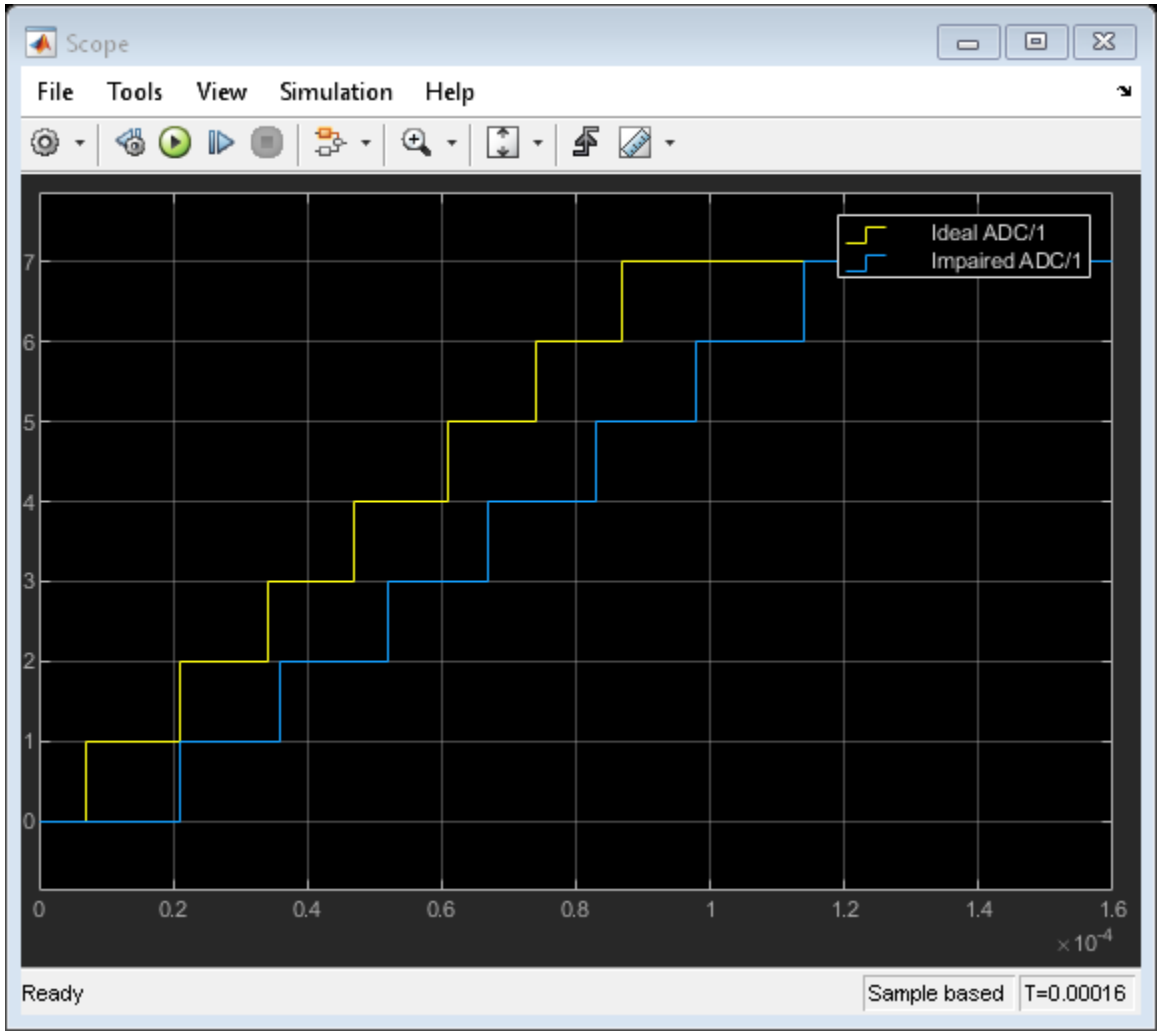
The sum of offset error and gain error is known as full-scale error. In monopolar converters zero error, essentially offset error defined at analog level `0`, is identical to regular offset error. In bipolar converters such as those above, offset error and zero error are different quantities.

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 '/Scope']);
sim(model);
```





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.

Compare SAR ADC to Ideal ADC

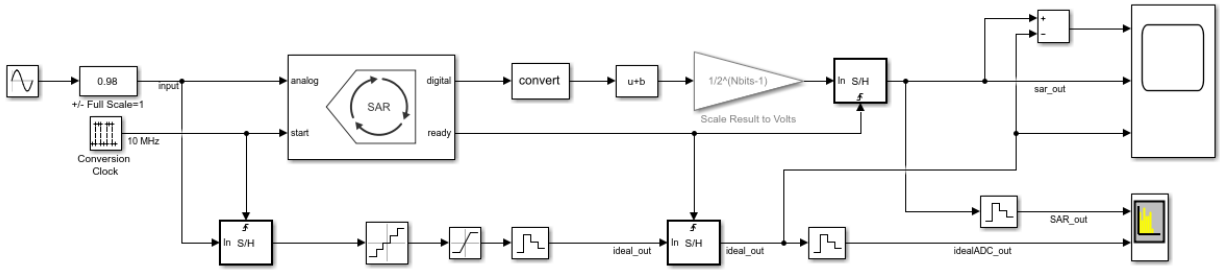
This example shows a comparison of the SAR ADC from the Mixed-Signal Blockset™ to the ideal ADC model with impairments presented in Analyzing a Simple ADC with Impairments.

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.

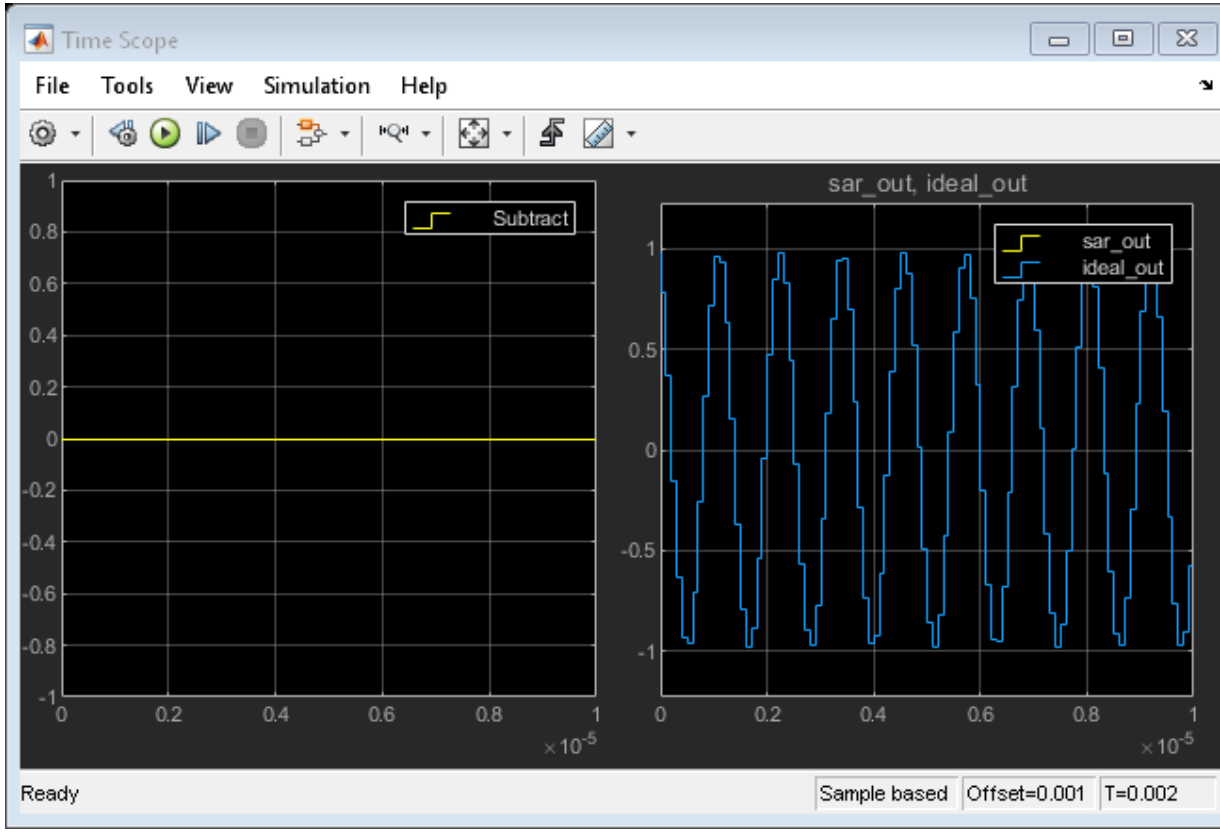
The SAR ADC outputs an unsigned, 16-bit integer. The minimum value of the converter's output is 0 and maximum value is 2^ **NBits** - 1. This is less than the maximum value of the integer as long as the integer has more bits than the converter. The Bias and Scale Results to Volts blocks scale the former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



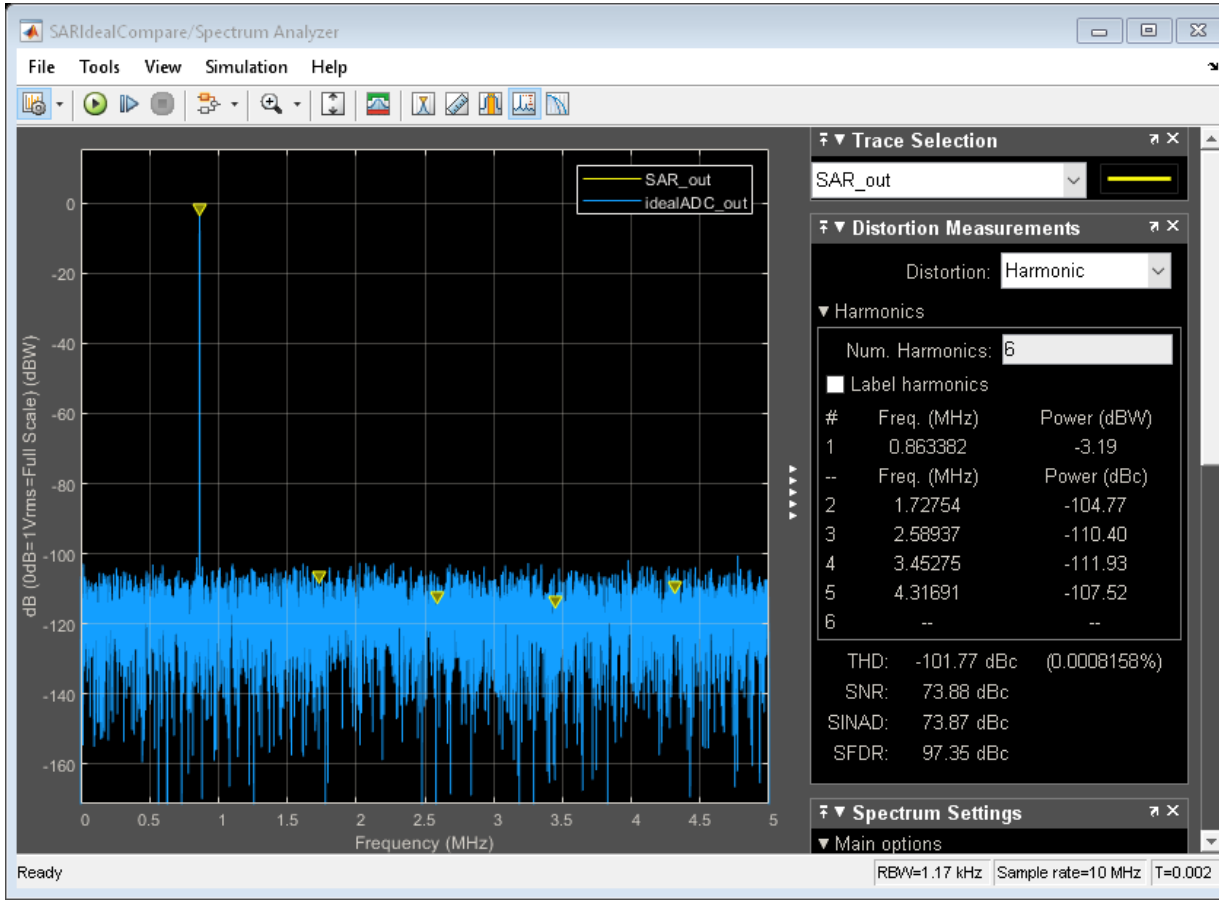
The Time Scope block shows the difference between the two outputs on the left and the two outputs themselves on the right.

```
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.

Mixing Analog and Digital Signals Featured Examples

Digital Timing using Solutions to Ordinary Differential Equations

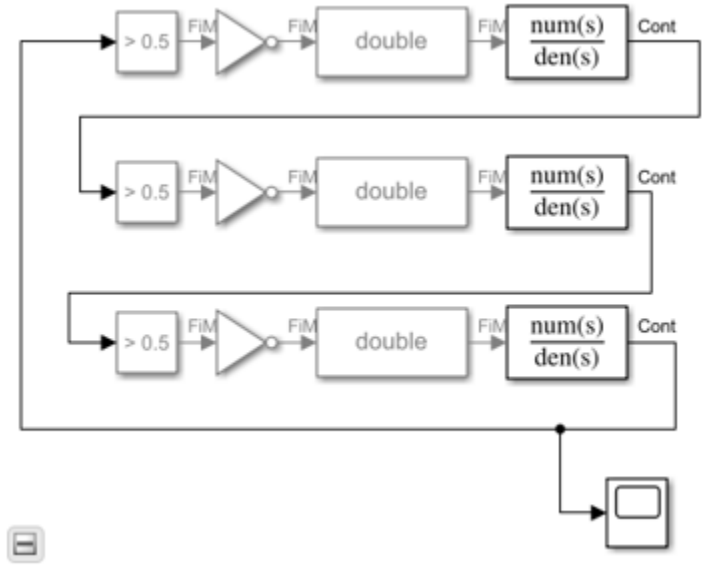
This example shows how to model a three stage ring oscillator using models defined by ordinary differential equations (ODE).

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 frquency, making the accurate modeling of these delays essential to the the simulation of the circuit.

The first and second examples in the sequence contain background that will help you get the most out of this example. You should read them first, in sequence, if you haven't 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');
```



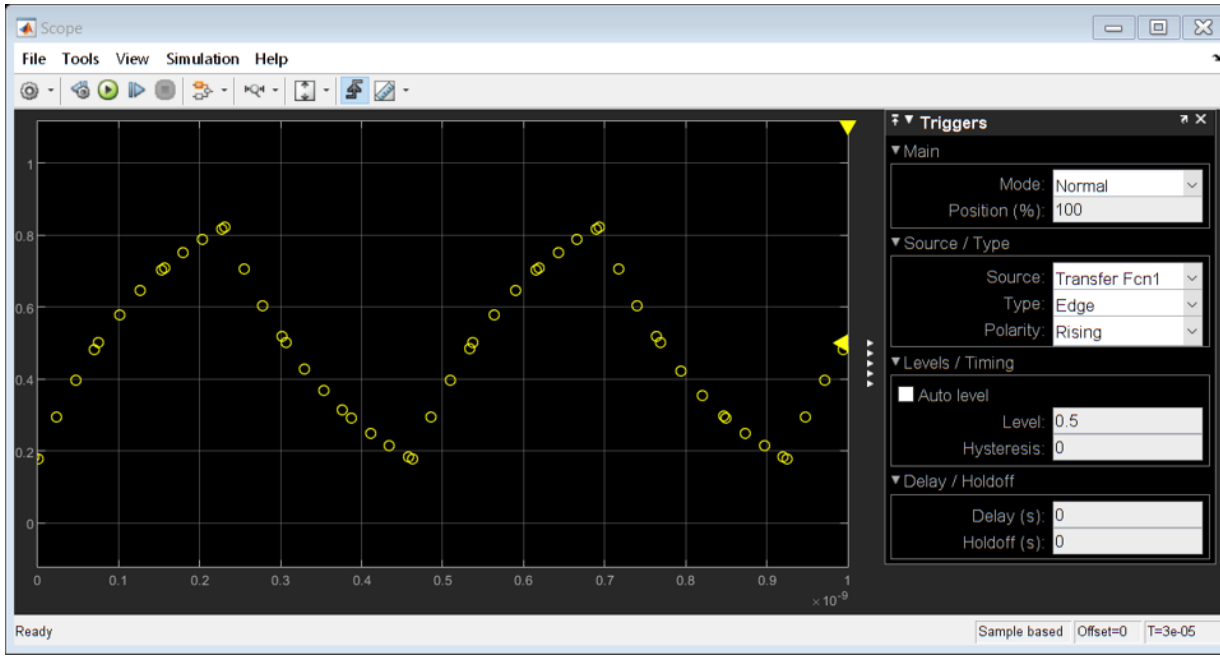
Continuous Time Model Using Exponential Decay

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 single pole rise/fall response.
sim('OdeWaveform');
```



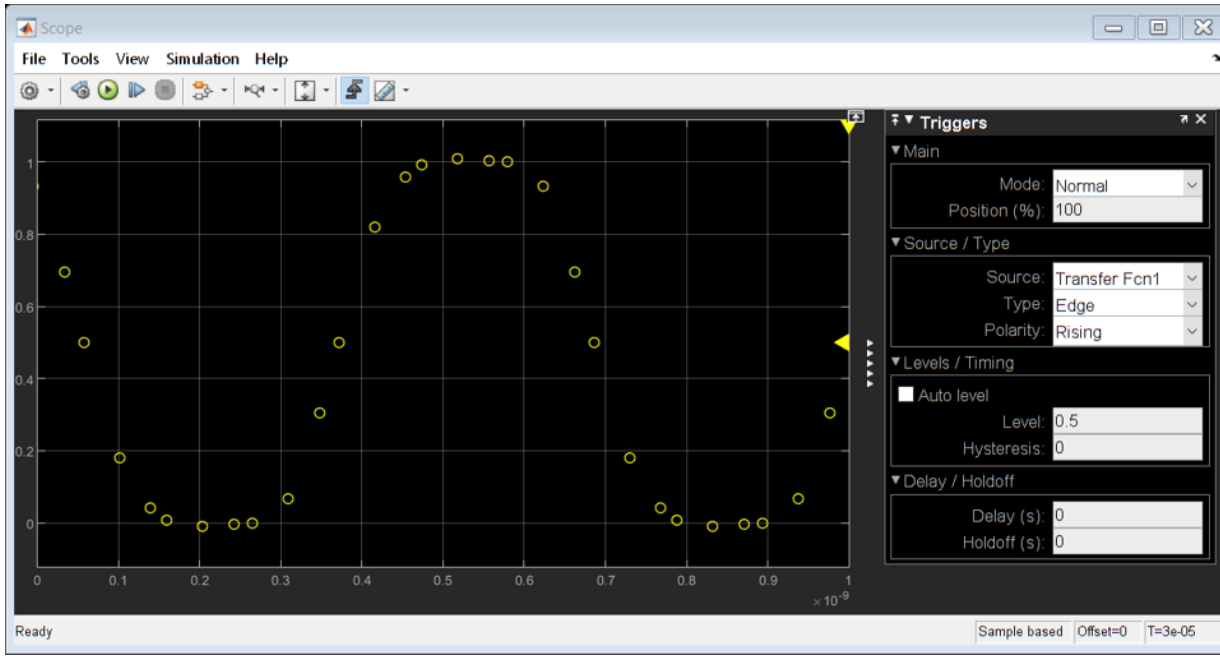
Continuous Time Model with Nearly Constant Slew Rate

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.

Note the slight rounding of the onset of the switching edges, similar to the waveforms produced in the **Digital Timing using Fixed Step Sampling** example.

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

```
Warning: Undefined variable "autosar" or class
"autosar.api.Utils.initMessageStreamHandler".
```



Continuous Time with Solver in auto Mode

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

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

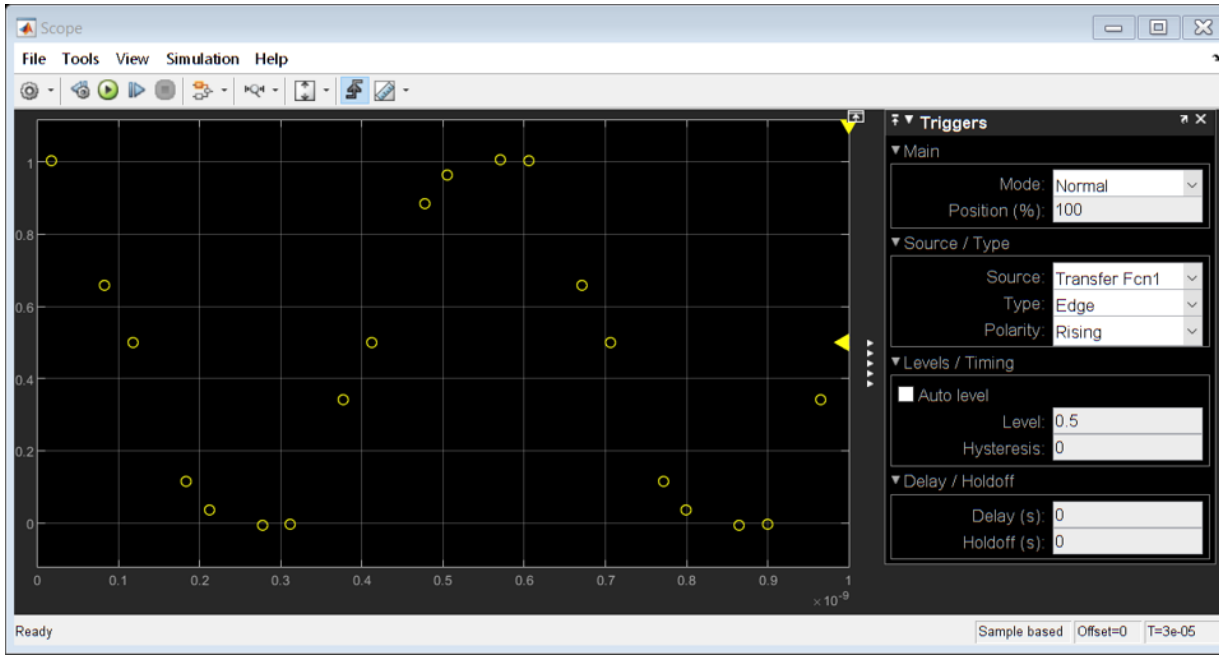
Observe both the change in period of oscillation and in wave shape.

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.

```
% Change the solver's relative tolerance to 'auto'.
set_param('OdeWaveform','RelTol','auto');
```

```
% Run the model with the auto solver setting.
sim('OdeWaveform');

Warning: Undefined variable "autosar" or class
"autosar.api.Utils.initMessageStreamHandler".
```



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 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 frquency, making the accurate modeling of these delays essential to the the simulation of the circuit.

The first example in the sequence contains background that will help you get the most out of this example. You should read it first, if you haven't already done so.

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.

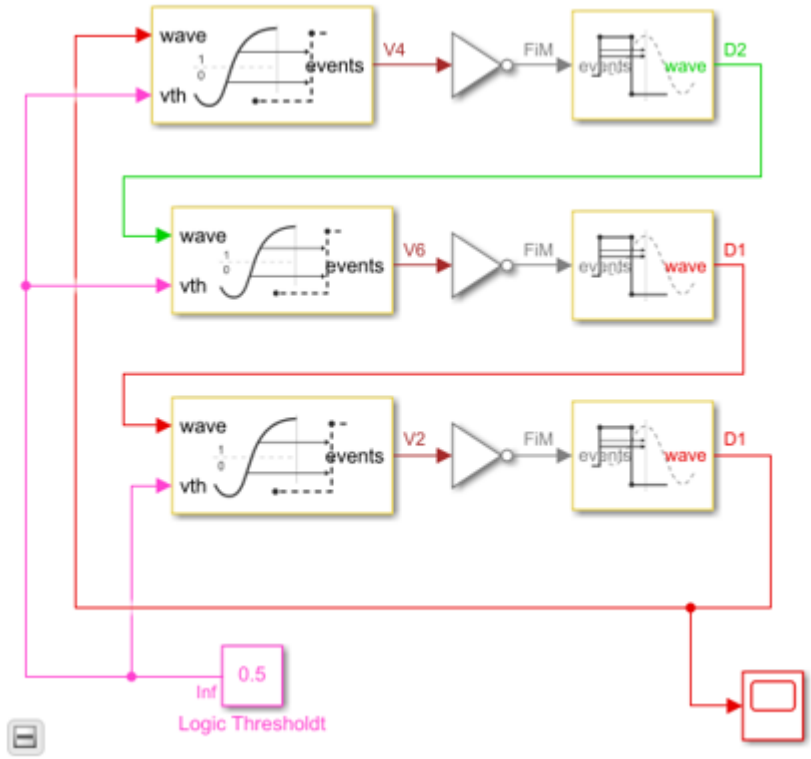
For a variable step input, the output of the Logic Decision block is delayed by the minimum delay parameter for the block.

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.

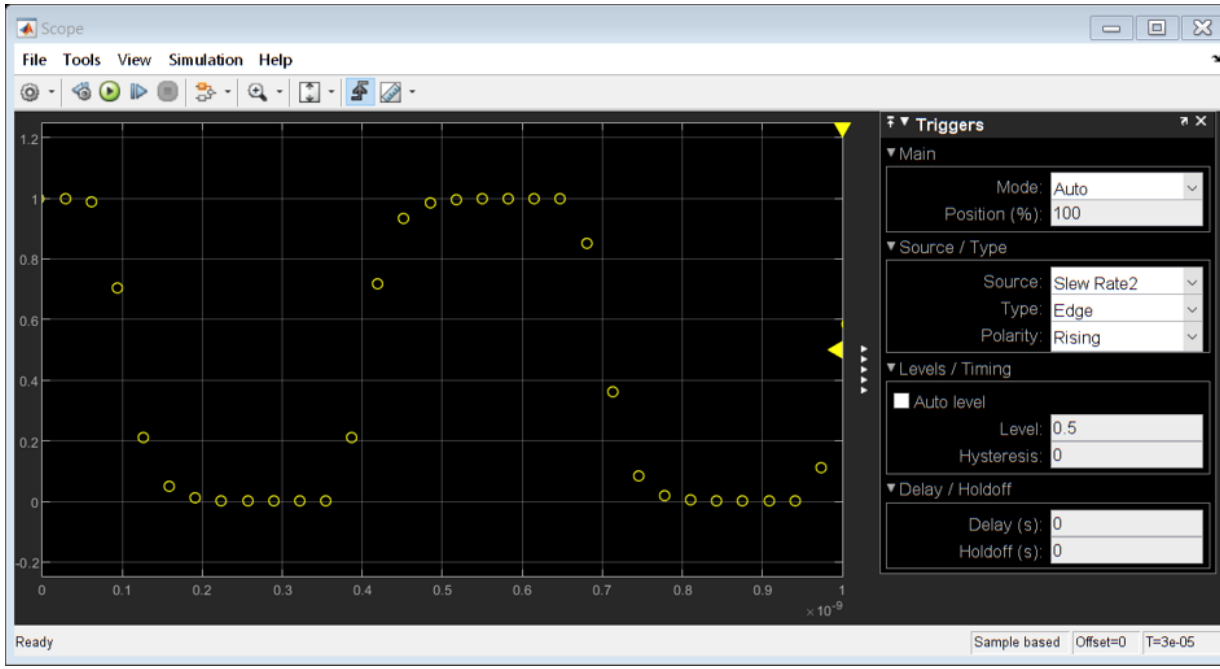
For this section, the Slew Rate blocks have been configured to their **Default** sampling mode, which maximizes the portion of the delay due to nearly constant slew rate (saturated transistor) and minimizes the delay due to constant delay or exponential decay.

The delay for one logic stage has been set to a slightly different value than for the other stages so that the model will enter 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

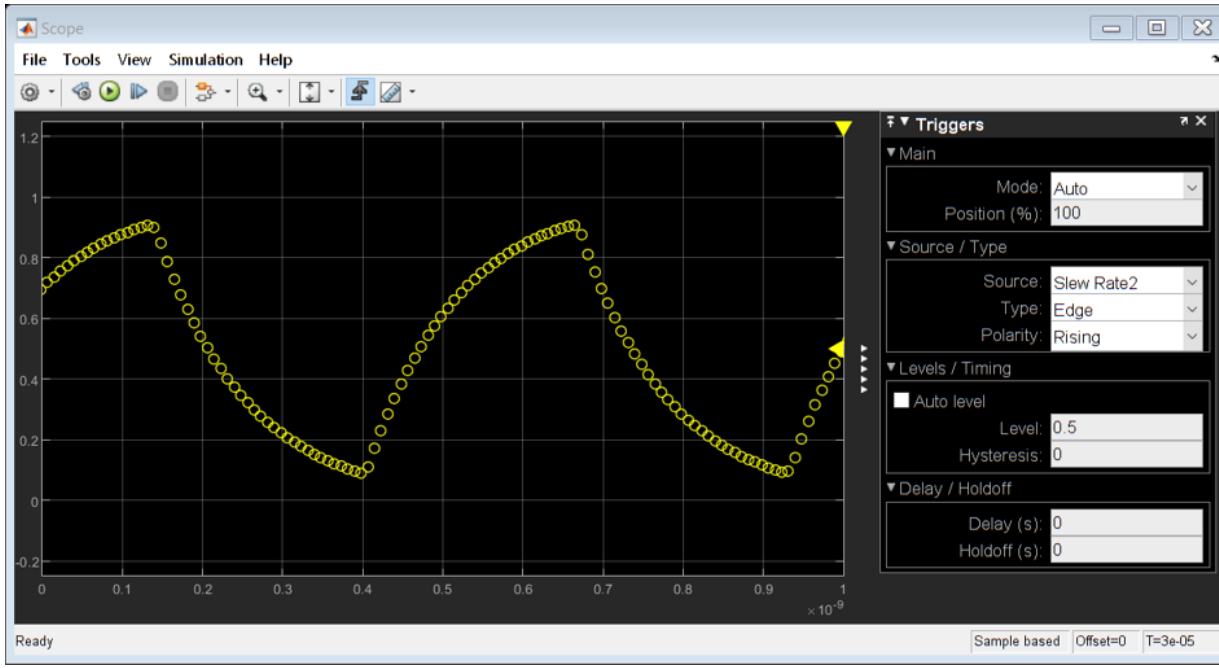
In this section, use the **Advanced** sampling mode of the Slew Rate block to model circuits whose response is primarily a decaying exponential.

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 revised sample times in sample time legend.
set_param(gcs,'SimulationCommand','update');
% Run he mixed analog/digital model with emphasized one pole response
sim('AnalogWaveform');
```

```
Warning: Undefined variable "autosar" or class
"autosar.api.Utils.initMessageStreamHandler".
```


Scope
File Tools View Simulation Help
Triggers
Main
Mode: Auto
Position (%): 100
Source / Type
Source: Slew Rate2
Type: Edge
Polarity: Rising
Levels / Timing
Auto level
Level: 0.5
Hysteresis: 0
Delay / Holdoff
Delay (s): 0
Holdoff (s): 0
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
×10^-9
Ready
Sample based
Offset=0
T=3e-05

Logic Timing Simulation

This example shows how to use the Variable Pulse Delay block from the Mixed Signal Library 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 frquency, making the accurate modeling of these delays essential to the the simulation of the circuit.

The second and third examples both show how to produce analog waveforms with accurate shape and timing. 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 Mixed-Signal Blockset's Utilities library, 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 value for two of the blocks is set to 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 explicity 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.

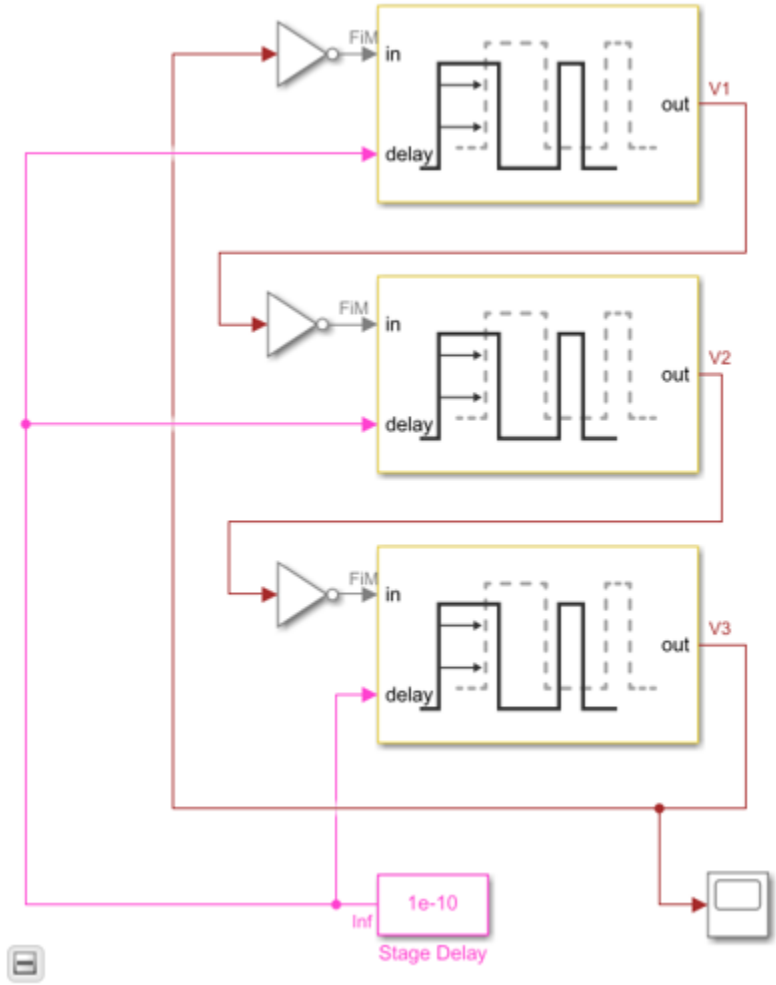
As indicated by the sample time color coding, the output sample time for the inverters is Fixed In Minor Step (FIM). This means that each inverter will produce an output sample value for every major sample time in the model, regardless whether or not that sample time is used at an input port of the gate.

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 block (also from the Mixed-Signal Blockset's Utilities library).

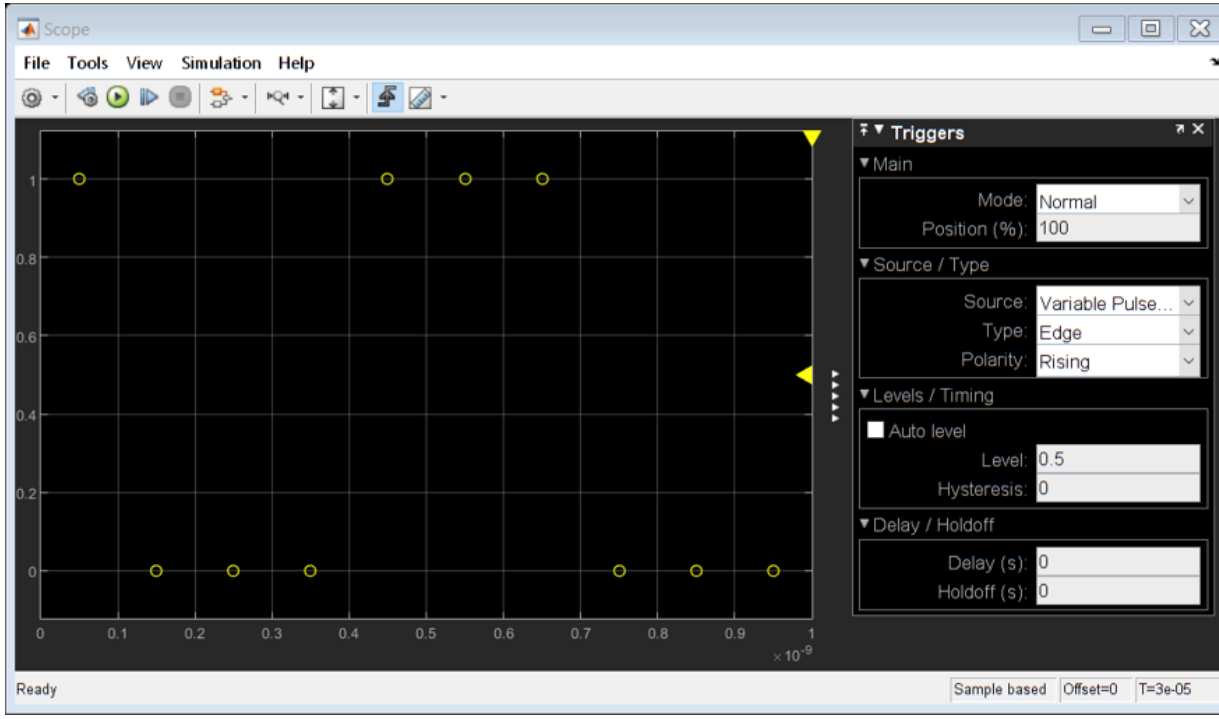
Since the model does not contain any differential equations, the solver is **Variable Step Discrete**.

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');
```


Scope
File Tools View Simulation Help
Triggers
Main
Mode: Normal
Position (%): 100
Source / Type
Source: Variable Pulse...
Type: Edge
Polarity: Rising
Levels / Timing
Auto level
Level: 0.5
Hysteresis: 0
Delay / Holdoff
Delay (s): 0
Holdoff (s): 0
1
0.8
0.6
0.4
0.2
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
×10^-9
Ready
Sample based
Offset=0
T=3e-05

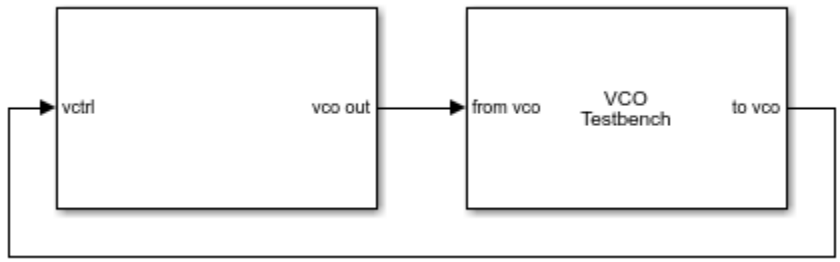
PLL Block Level Examples

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.

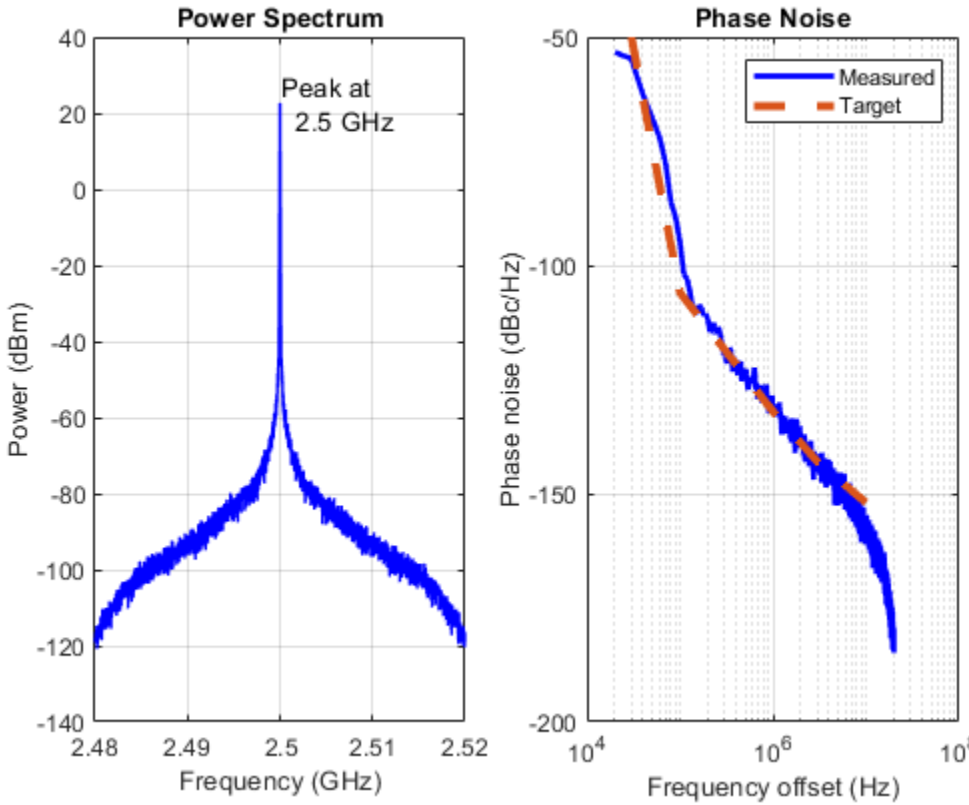
```
open_system('vcoPhaseNoise.slx')
```



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 **Sampling frequency (Hz)**, **Resolution bandwidth (Hz)**, and **No. of spectral averages**.

Run simulation for `2.4e-3` 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.



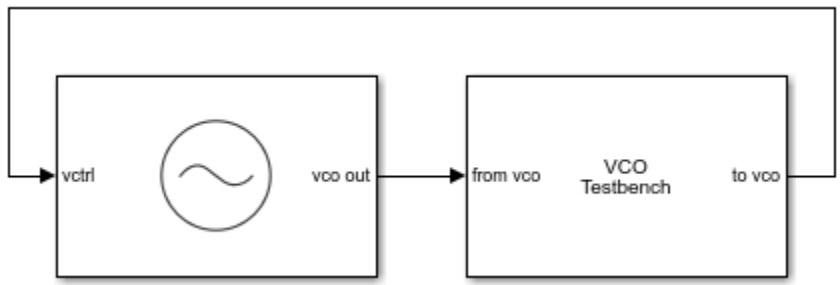
The operating frequency matches the predicted frequency `2.5` GHz. The measured phase noise profile also matches the target profile.

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.

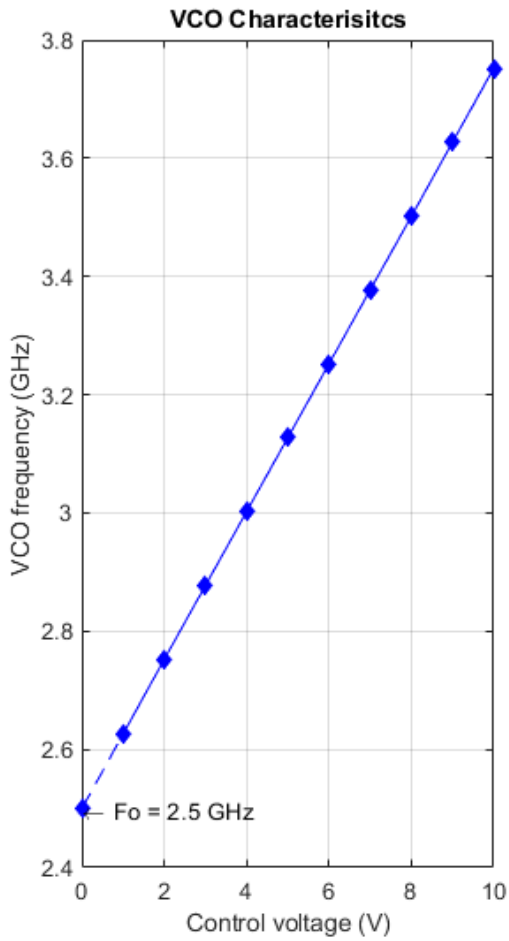
```
open_system('vcoCharacteristics.slx')
```

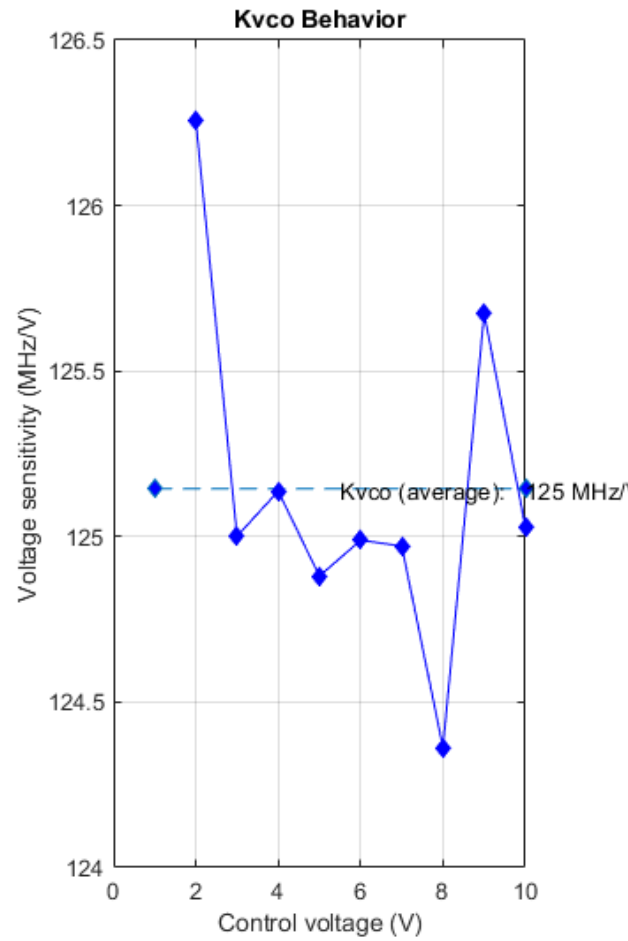


The testbench is set to measure the **Kvco and Fo** metric of the VCO in **Measurement** option. **Range of control voltage (V)** provided to the input of VCO is set to `[1 10]`.

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.

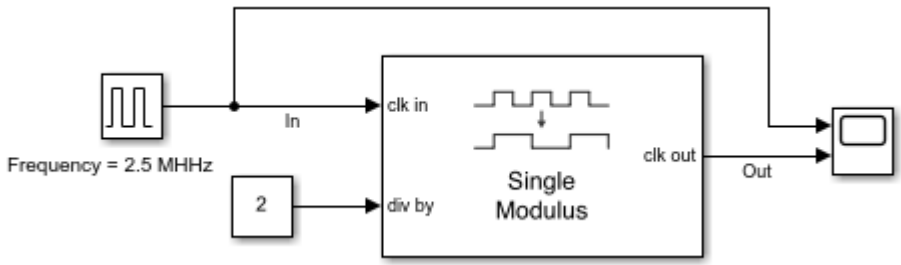

VCO Characterisitcs
VCO frequency (GHz)
Control voltage (V)
Fo = 2.5 GHz
3.8
3.6
3.4
3.2
3
2.8
2.6
2.4
0
2
4
6
8
10


Kvco Behavior
Voltage sensitivity (MHz/V)
Control voltage (V)
Kvco (average):
125 MHz/V
126.5
126
125.5
125
124.5
124
0
2
4
6
8
10

Frequency Division Using Single Modulus Prescaler

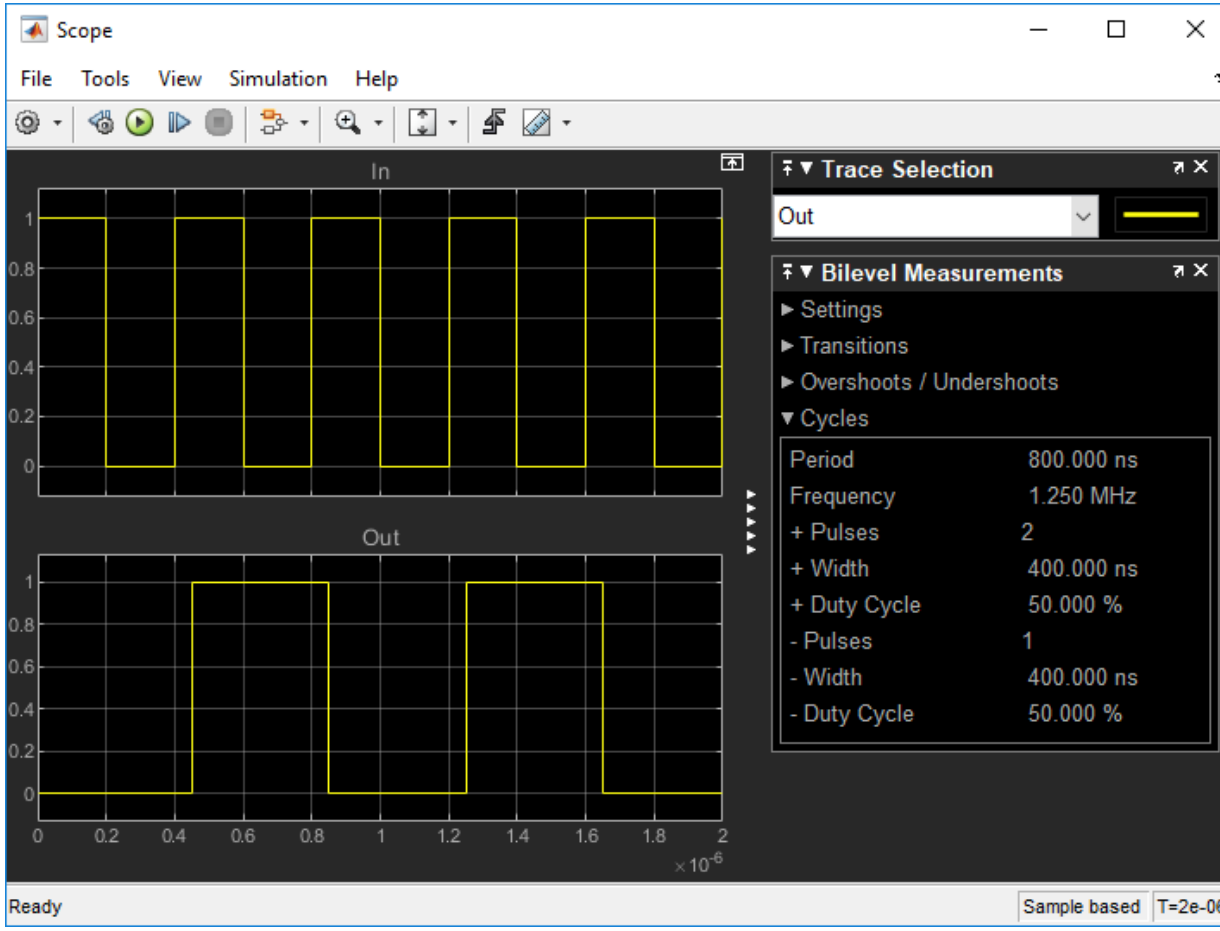
Open the model `singleModulusPrescaler`. The model consists of a Pulse Generator and a Single Modulus Prescaler block.

```
open_system('singleModulusPrescaler.slx')
```



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`.

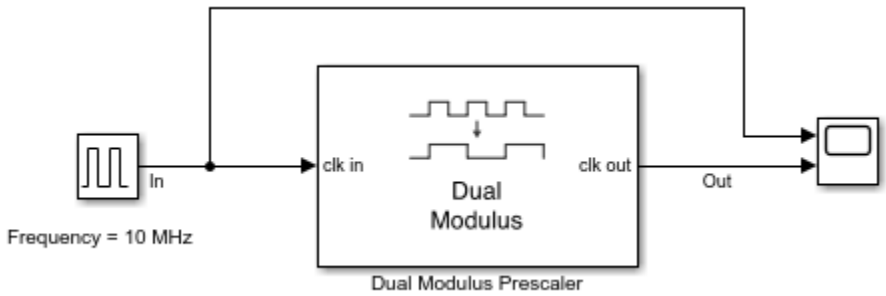
Run the simulation for `2e-6` s. The frequency of the output signal is `1.25` MHz.


Scope
File
Tools
View
Simulation
Help
In
Out
Trace Selection
Out
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 800.000 ns
Frequency 1.250 MHz
+ Pulses 2
+ Width 400.000 ns
+ Duty Cycle 50.000 %
- Pulses 1
- Width 400.000 ns
- Duty Cycle 50.000 %
Ready
Sample based
T=2e-06

Frequency Division Using Dual Modulus Prescaler

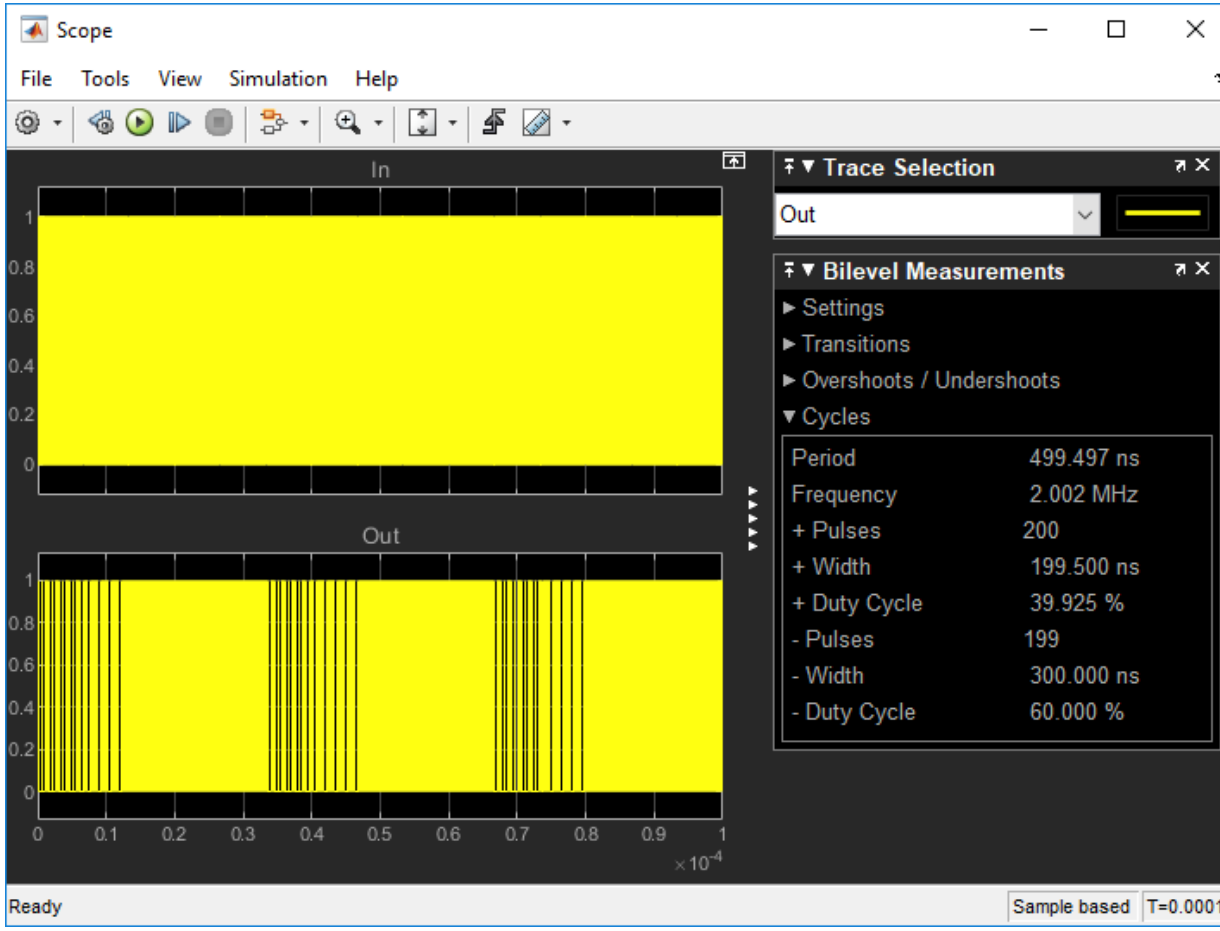
Open the model `Dual_Modulus_Prescaler_Ex`. The model consists of a Pulse Generator and a Dual Modulus Prescaler block.

```
open_system('Dual_Modulus_Prescaler_Ex.slx')
```



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`.

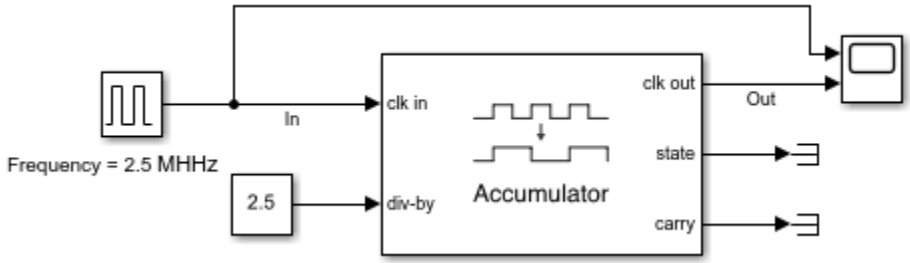
Run the simulation for `1e-4` s. The frequency of the output signal is `2.002` MHz.


Scope
File
Tools
View
Simulation
Help
In
Out
Trace Selection
Out
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 499.497 ns
Frequency 2.002 MHz
+ Pulses 200
+ Width 199.500 ns
+ Duty Cycle 39.925 %
- Pulses 199
- Width 300.000 ns
- Duty Cycle 60.000 %
Ready
Sample based
T=0.0001

Frequency Division Using Fractional Clock Divider with Accumulator

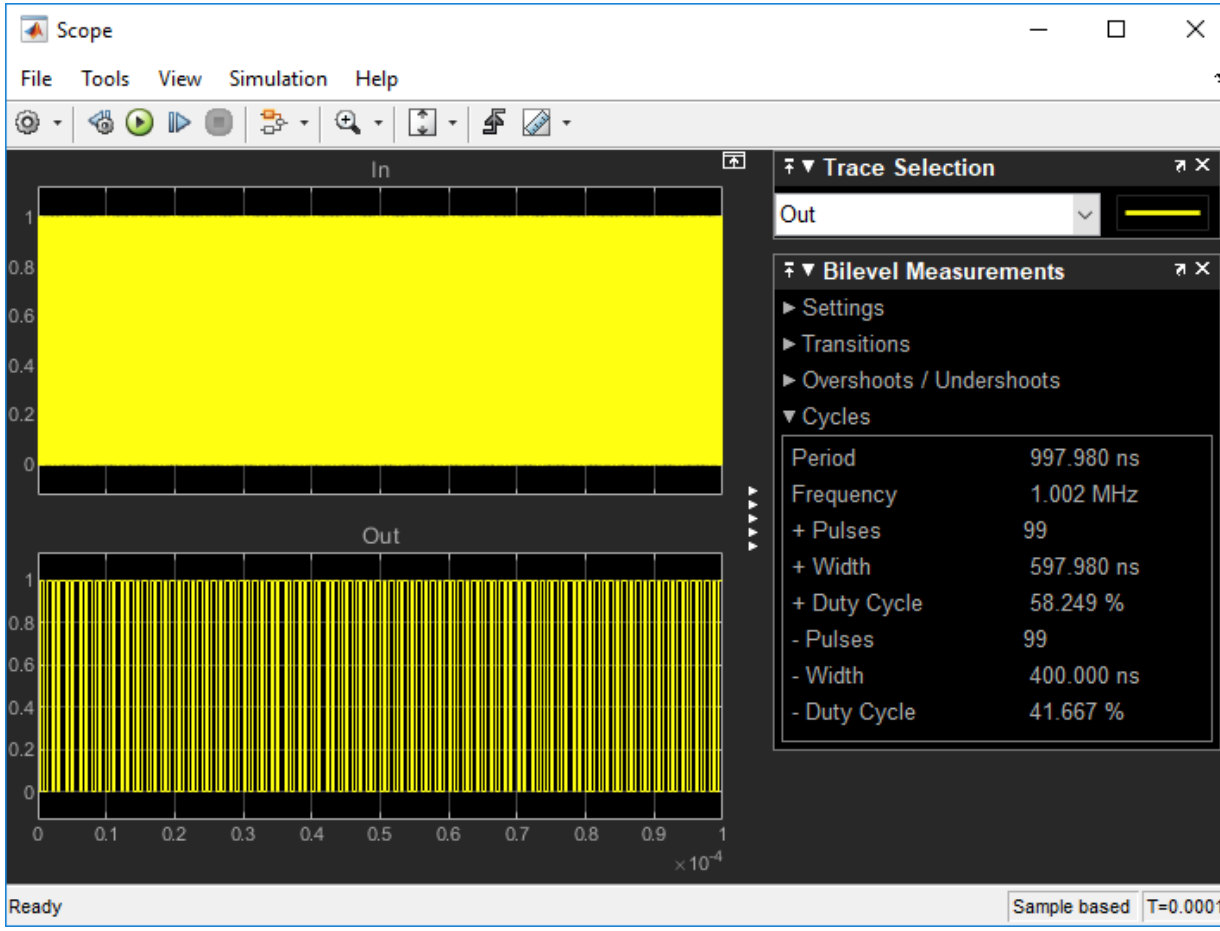
Open the model `fractionalClockDivider_w_Accumulator`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

```
open_system('fractionalClockDivider_w_Accumulator.slx')
```



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`.

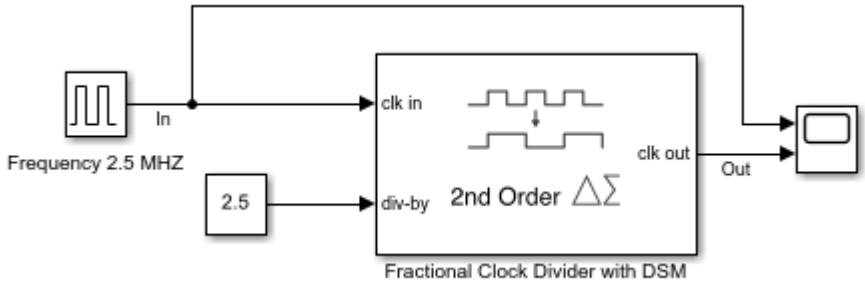
Run the simulation for `1e-4` s. The frequency of the output signal is `1.002` MHz.


Scope
File
Tools
View
Simulation
Help
In
Out
Trace Selection
Out
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 997.980 ns
Frequency 1.002 MHz
+ Pulses 99
+ Width 597.980 ns
+ Duty Cycle 58.249 %
- Pulses 99
- Width 400.000 ns
- Duty Cycle 41.667 %
Ready
Sample based
T=0.0001

Frequency Division Using Fractional Clock Divider with DSM

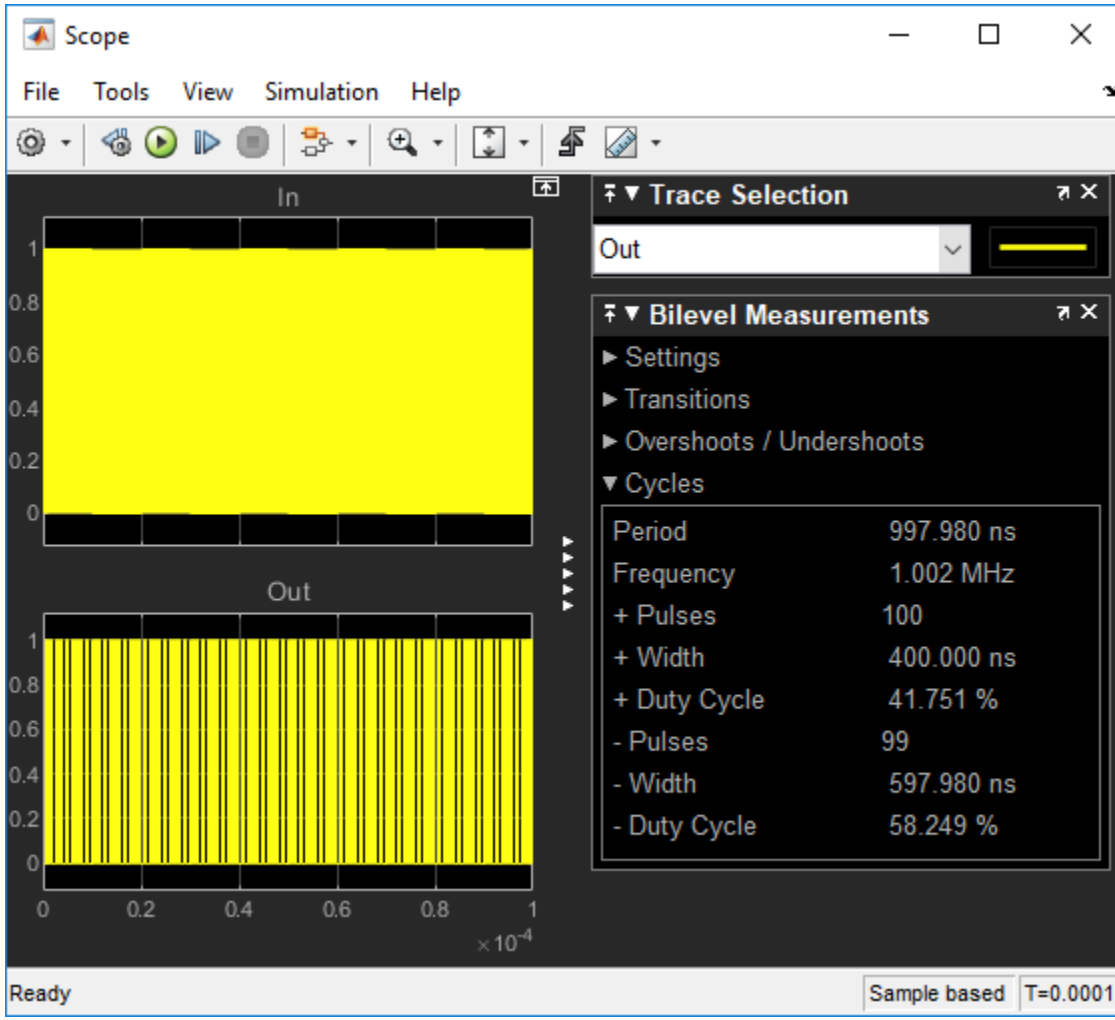
Open the model `fractionalClockDivider_w_DSM`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

```
open_system('fractionalClockDivider_w_DSM.slx')
```



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.


Scope
File
Tools
View
Simulation
Help
In
Out
Trace Selection
Out
Bilevel Measurements
Settings
Transitions
Overshoots / Undershoots
Cycles
Period 997.980 ns
Frequency 1.002 MHz
+ Pulses 100
+ Width 400.000 ns
+ Duty Cycle 41.751 %
- Pulses 99
- Width 597.980 ns
- Duty Cycle 58.249 %
Ready
Sample based
T=0.0001