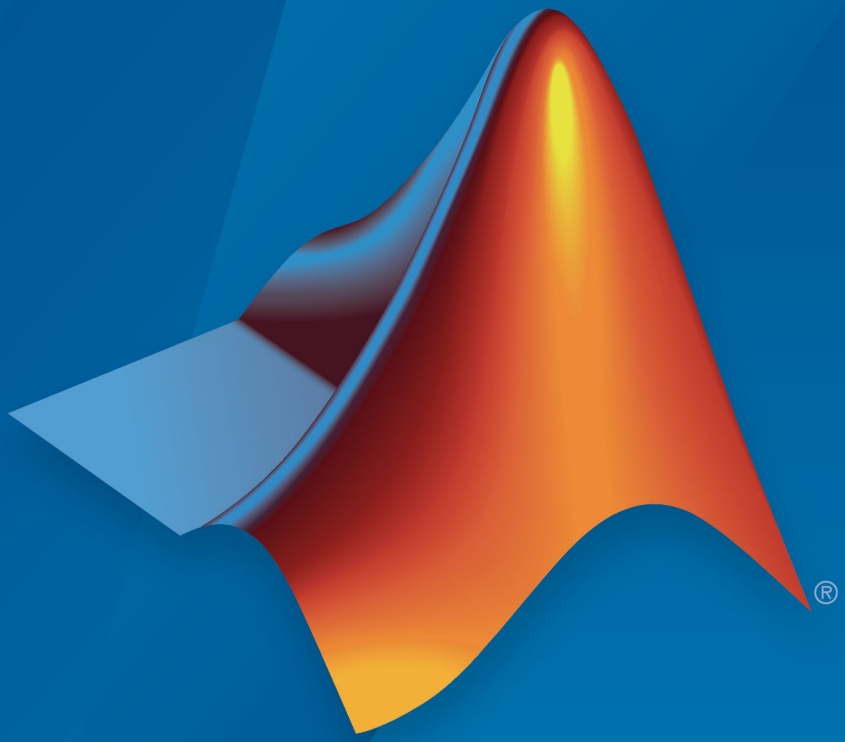
# Mixed-Signal Blockset™

User's Guide



MATLAB® & SIMULINK®

R2019a

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) |

# Contents

# PLL Featured Examples

# Phase Noise at PLL Output

This example shows how to measure and analyze the phase noise at the output of a phase-locked loop (PLL) using the PLL testbench.

The five sections of this example demonstrate three phase noise effects, individually and collectively:

1. Numerical noise baseline
2. The effect of reference phase noise only
3. The effect of VCO phase noise only
4. The effect of VCO phase noise subsampling by the feedback prescaler
5. The combined effect of all three phase noise sources.

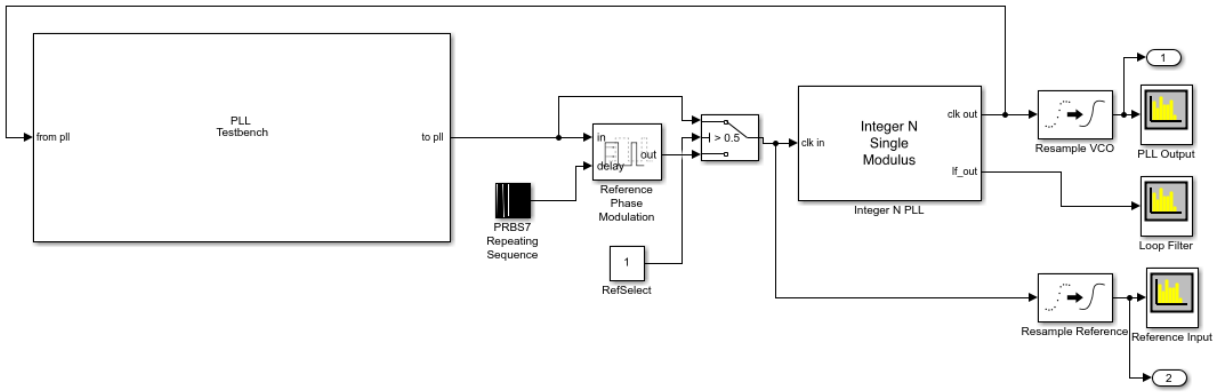
The PLL Testbench generates a reference signal and measures the phase noise at the PLL output.

The reference phase modulation in this model is used to contrast the response to reference phase variation with the response to VCO phase noise. The PLL Testbench does not currently include a specfication of reference phase noise; however the user can insert reference phase noise between the PLL Testbench output and the PLL reference input, as is done in this example.

The Resample blocks at the inputs of two of the spectrum analyzers convert the variable step discrete sampled signals to the fixed step discrete sampled signals the spectrum analyzer requires. The Resample block is a block inside the Slew Rate block found in the Mixed Signal Blockset Utilities library. It is a low pass filter whose impulse response is described by a closed form equation. It is therefore able to calculate an output sample value for any sample time step, independent of the input sample time step.

The model uses a reference frequency **fref** of `30` MHz and a feedback divider ratio **N** of `70` for all but one section. The output frequency **fout** is **N*x*fref**, or `2.1` GHz for those sections.

```
% Make sure genPrbs7 has been run.
if ~exist( 'prbs7', 'var' ) || ~exist( 'prbs7time', 'var' )
    [prbs7, prbs7time] = genPrbs7(30e6,500e3,-110);
end
% Make sure PllPhaseNoiseExample.slx has been loaded.
open_system('PllPhaseNoiseExample.slx');
```



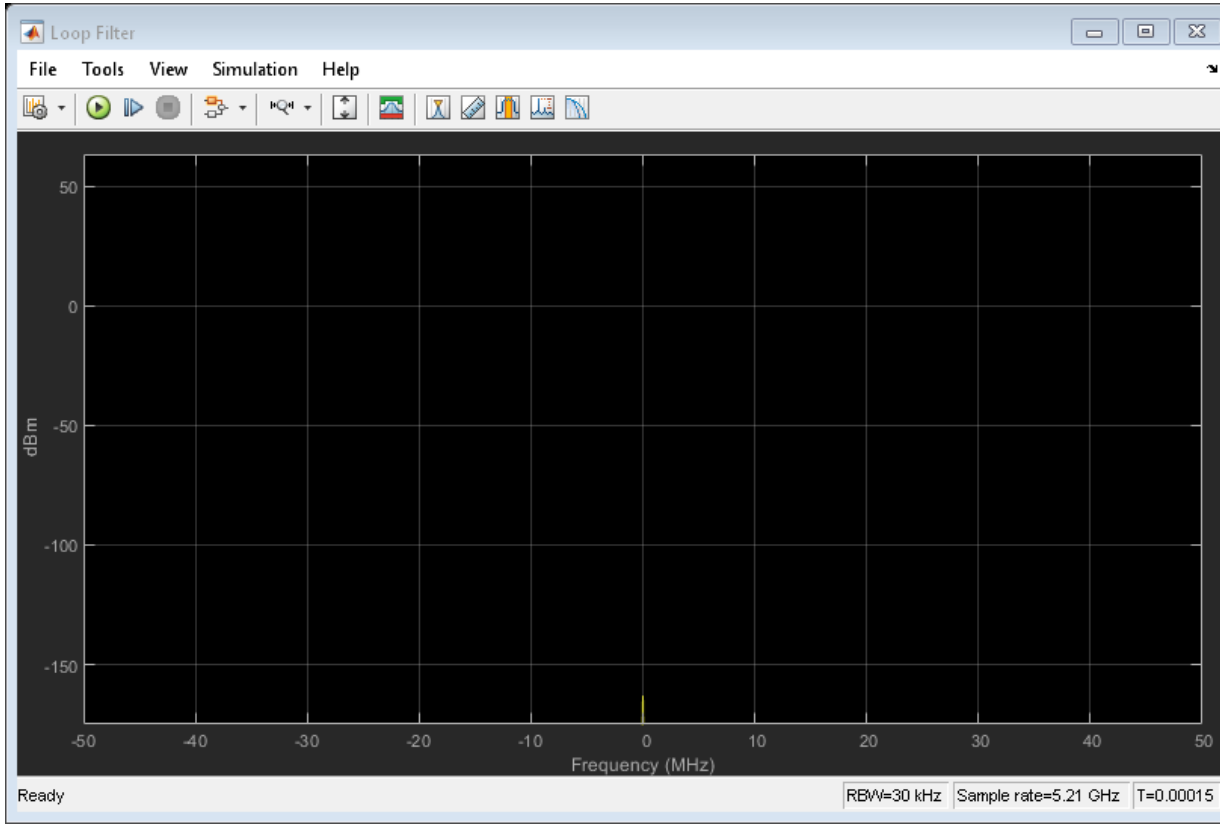

### Baseline

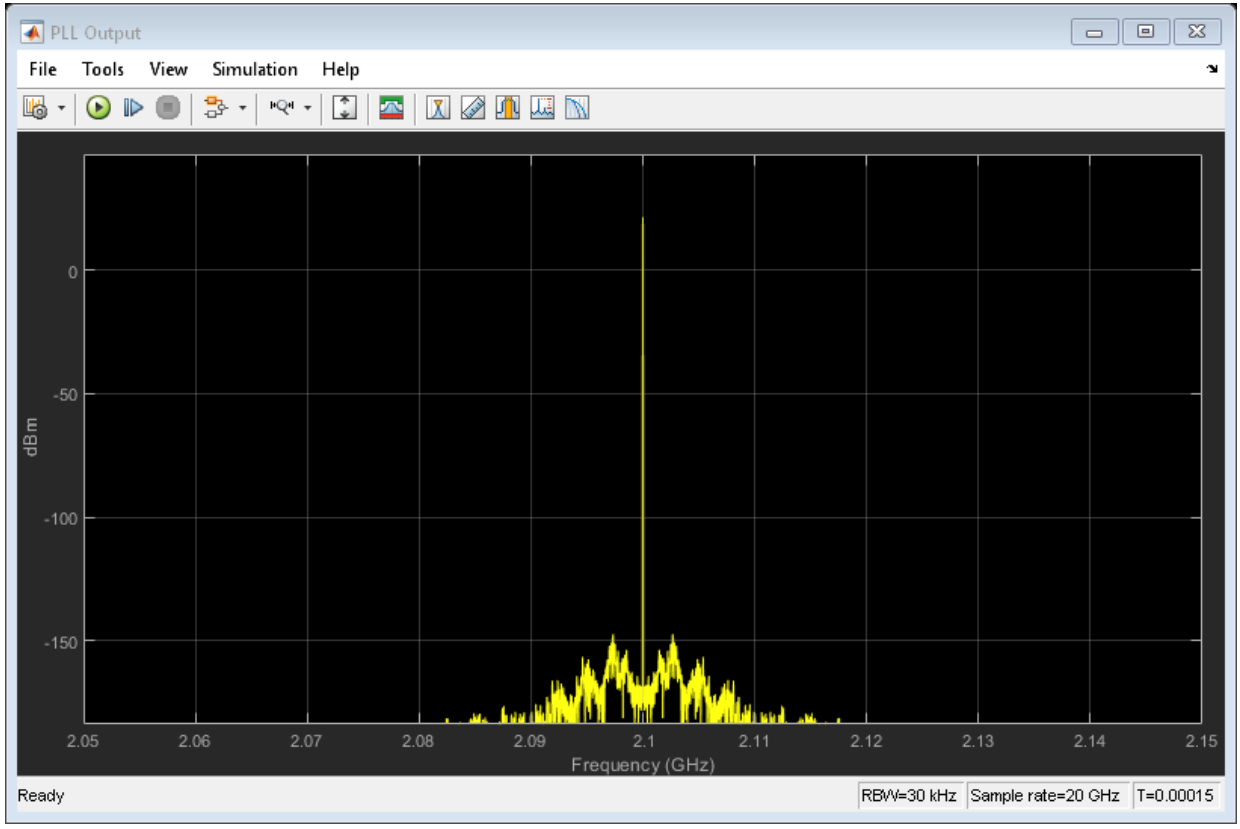
This section measures the limitations of the example model due to numerical noise or other inaccuracies. All of the sources of phase noise are disabled and the phase noise is measured to establish the baseline phase noise level. When sources of phase noise are enabled in subsequent sections, the effect of the phase noise source is the difference between the measured phase noise and the baseline phase noise level measured in this section.

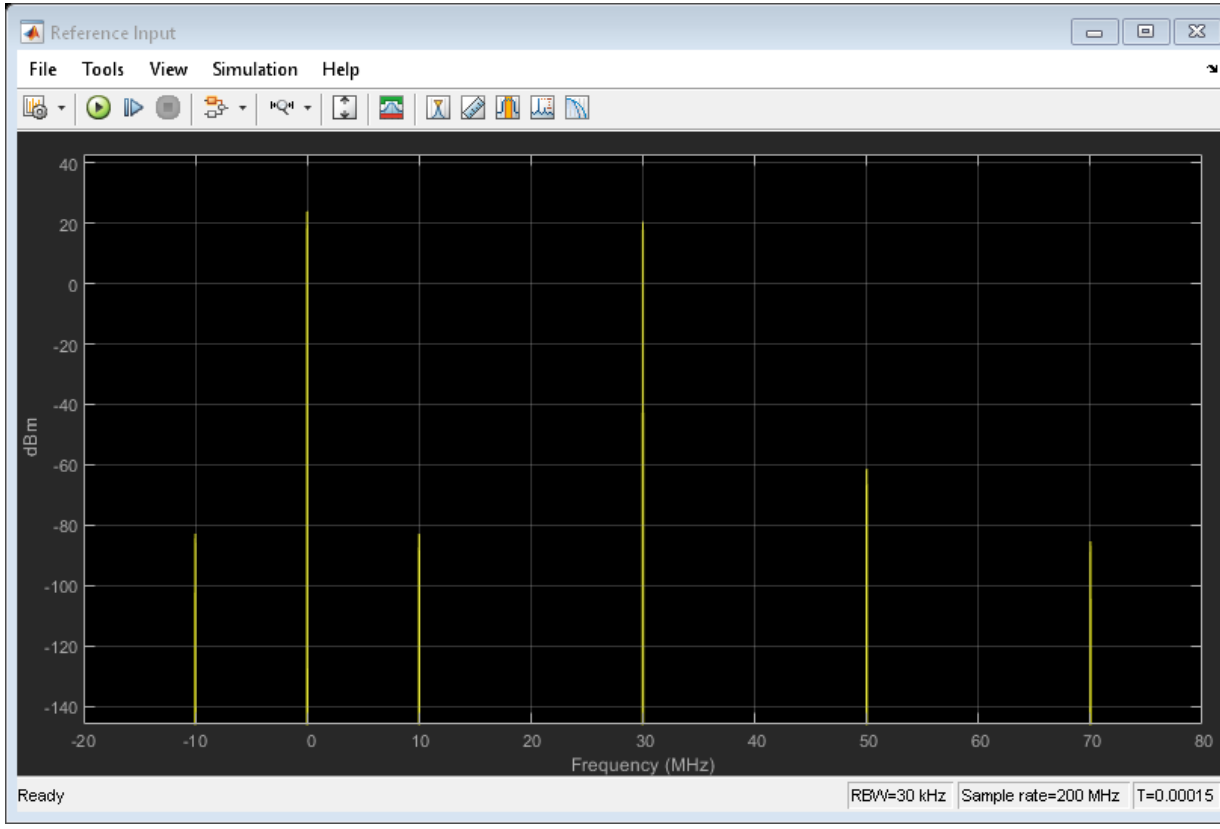
- VCO free-run frequency: **N*x*fref**
- Feedback prescaler ratio: **N**
- VCO phase noise: Disabled
- Reference modulation: Through path selected

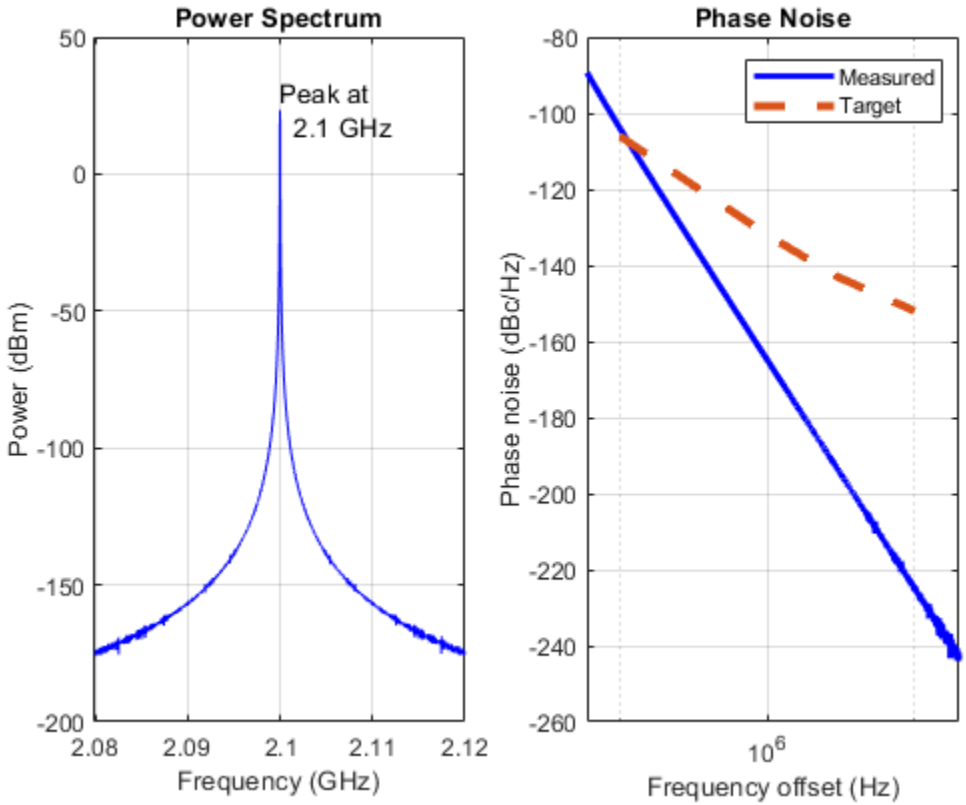
The target phase noise shown in the PLL Testbench phase noise plot is a PLL Testbench parameter. In this example, it is not actually a performance target, but rather was set equal to the VCO phase noise for comparison to the phase noise at the output of the PLL.

```
% Configure the model
configurePhaseNoiseExample( 'PllPhaseNoiseExample', 2.1e9, 70, 'off', '1');
% Run the simulation
sim('PllPhaseNoiseExample.slx');
% Plot the PLL Testbench phase noise analysis
plotPhaseNoiseAnalysis();
```


Loop Filter
File
Tools
View
Simulation
Help
50
0
-50
-100
-150
dBm
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=5.21 GHz
T=0.00015


PLL Output
File
Tools
View
Simulation
Help
0
-50
-100
-150
dBm
2.05
2.06
2.07
2.08
2.09
2.1
2.11
2.12
2.13
2.14
2.15
Frequency (GHz)
Ready
RBW=30 kHz
Sample rate=20 GHz
T=0.00015


Reference Input
File
Tools
View
Simulation
Help
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=200 MHz
T=0.00015




### Reference Source Phase Modulation

In this section, measure the amplification of phase modulation at the reference input of the PLL.

Unlike the very low level, random reference phase noise that is typical of most practical applications, this example uses PRBS7 reference phase modulation. This phase modulation produces discrete spectral components that contrast with the smooth spectral density of the VCO phase noise and in-band phase noise generated internal to the PLL. Such a reference phase modulatioon might be used in a spread spectrum application, though usually at a much higher modulation amplitude.

- VCO free-run frequency: **N*x*fref**

- Feedback prescaler ratio: **N**
- VCO phase noise: Disabled
- Reference modulation: *Phase modulated reference selected*

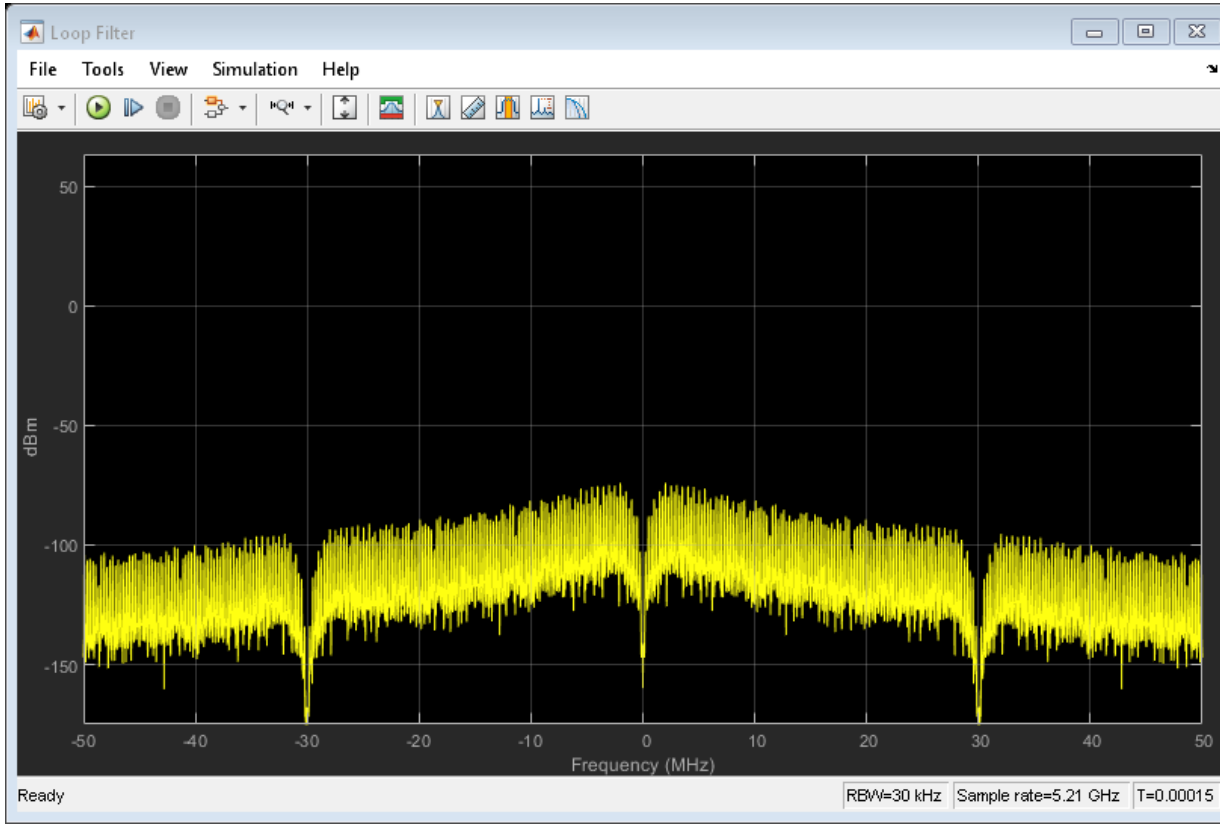
The reference input spectrum has many closely spaced spectral components of nearly the same amplitude across the entire spectrum.

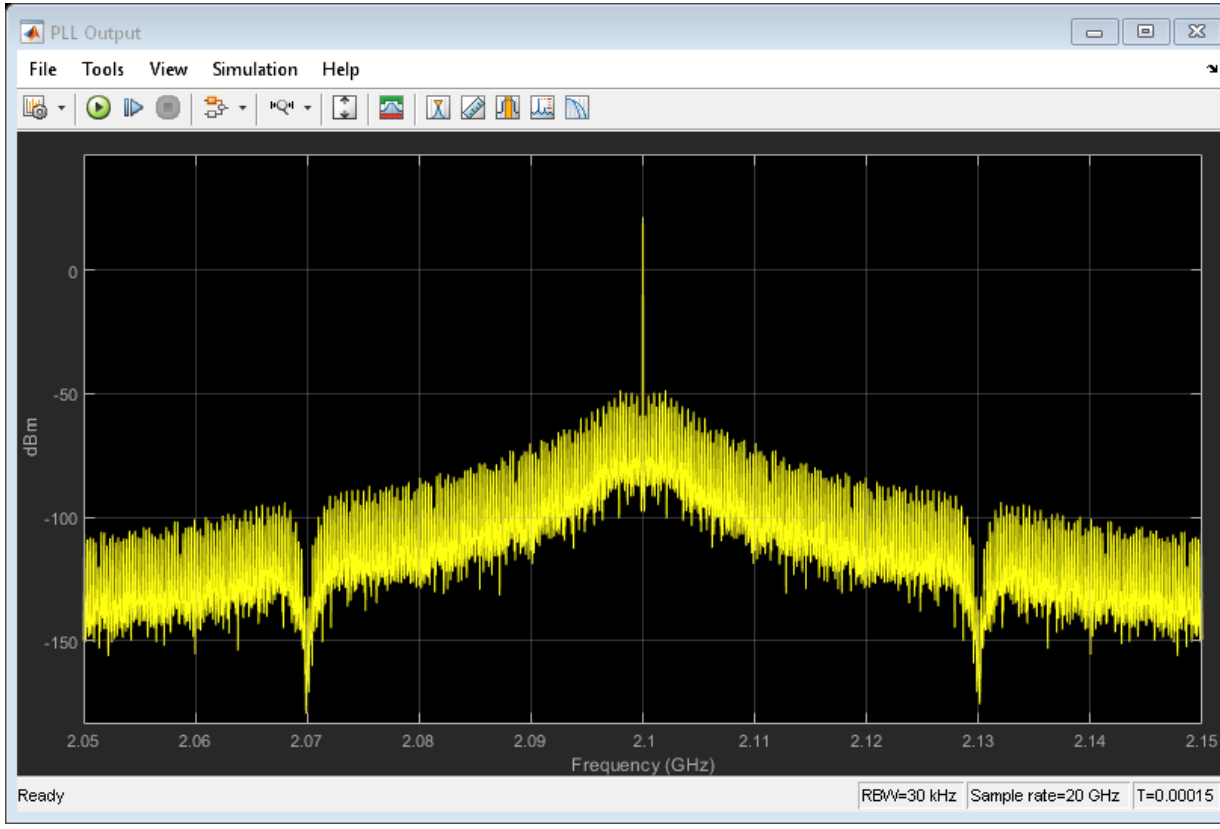
The PLL output spectrum with reference phase modulation includes spectral components at the same frequency offsets as for the modulated reference input, but at very different levels. The spectral components within the PLL loop bandwidth are amplified by the feedback prescaler divider ratio. Beyond the loop bandwidth, the spectral components fall off much more rapidly than the spectral components at the input, due to the filtering of the PLL control loop.

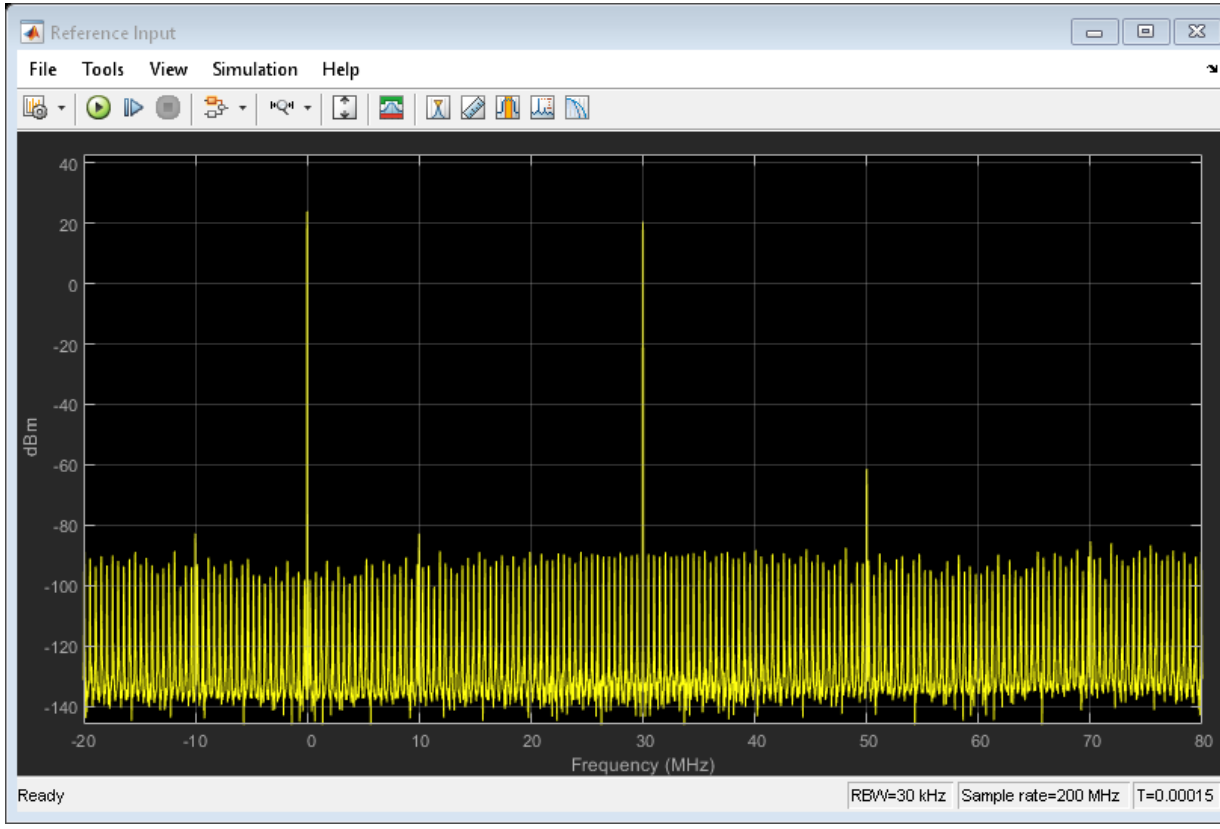
The spectrum at the output of the loop filter includes spectral components at the same frequency offsets as for the modulated reference input, but at levels that are very different from either the reference input or the PLL output. The loop filter output is integrated by the VCO; and in order to produce spectral components that are nearly coonstant within the PLL control loop bandwidth, the level of the spectral components at the loop filter output must be proportional to the frequency offset.

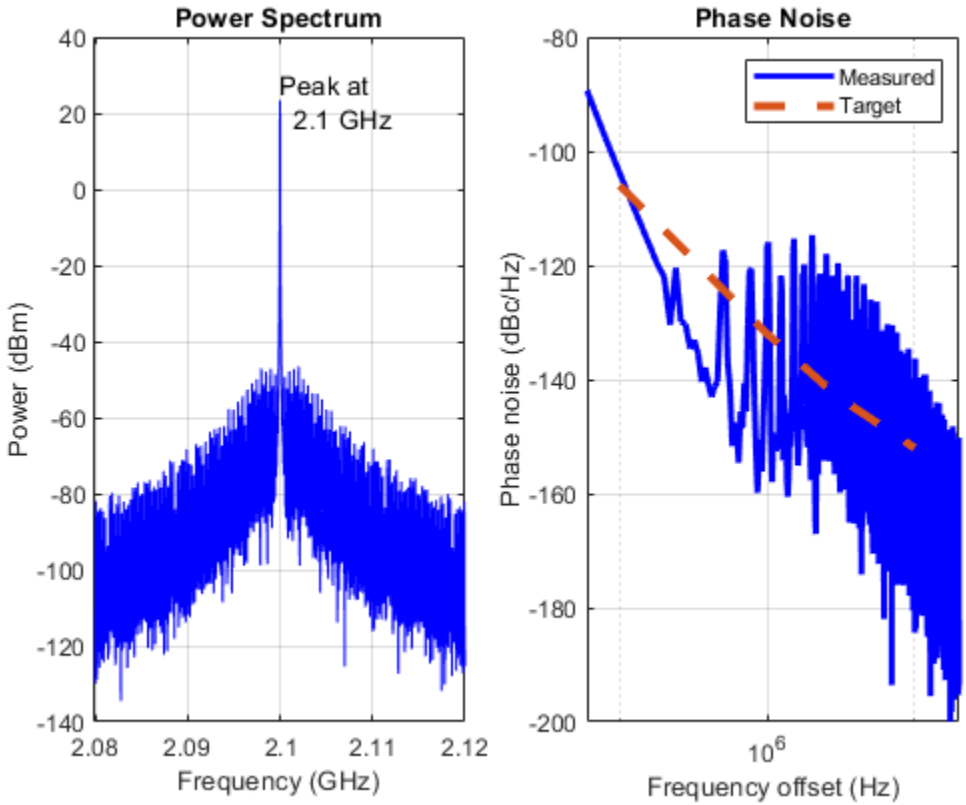
**Optional:** In the mask for the PLL, go to the loop filter tab and change the loop bandwidth (e.g., reduce the loop bandwidth by a factor of the), then re-run this section. Note the effect of the PLL control loop bandwidth on the spectrum at the PLL output.

```
% Configure the model.
configurePhaseNoiseExample( 'PllPhaseNoiseExample', 2.1e9, 70, 'off', '0');
% Run the simulation
sim('PllPhaseNoiseExample.slx');
% Plot the PLL Testbench phase noise analysis
plotPhaseNoiseAnalysis();
```


Loop Filter
File
Tools
View
Simulation
Help
50
0
-50
-100
-150
dBm
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=5.21 GHz
T=0.00015


PLL Output
File
Tools
View
Simulation
Help
0
-50
-100
-150
dBm
2.05
2.06
2.07
2.08
2.09
2.1
2.11
2.12
2.13
2.14
2.15
Frequency (GHz)
Ready
RBW=30 kHz
Sample rate=20 GHz
T=0.00015


Reference Input
File
Tools
View
Simulation
Help
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=200 MHz
T=0.00015




### VCO Phase Noise

In this section, observe the direct output of VCO phase noise as filtered by the PLL loop response. This is done by selecting the unmodulated reference, setting the feedback prescaler divider ratio to one, setting the VCO free running frequency equal to the reference frequency, and enabling the VCO phase noise. The loop filter design is automatically updated so you can compare the results with other feedback prescaler ratios.

- VCO free-run frequency: **fref**
- Feedback prescaler ratio: 1
- VCO phase noise: *Enabled*

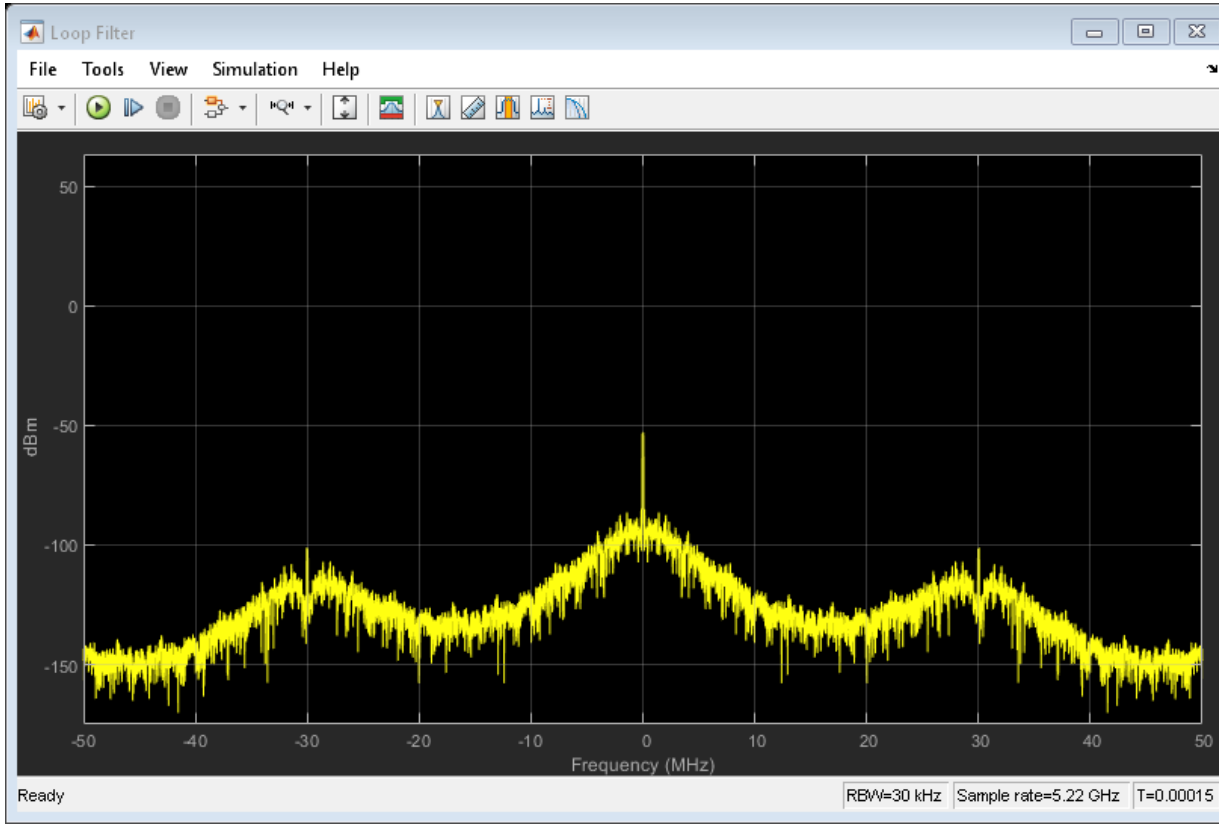
- Reference modulation: *Through path selected*

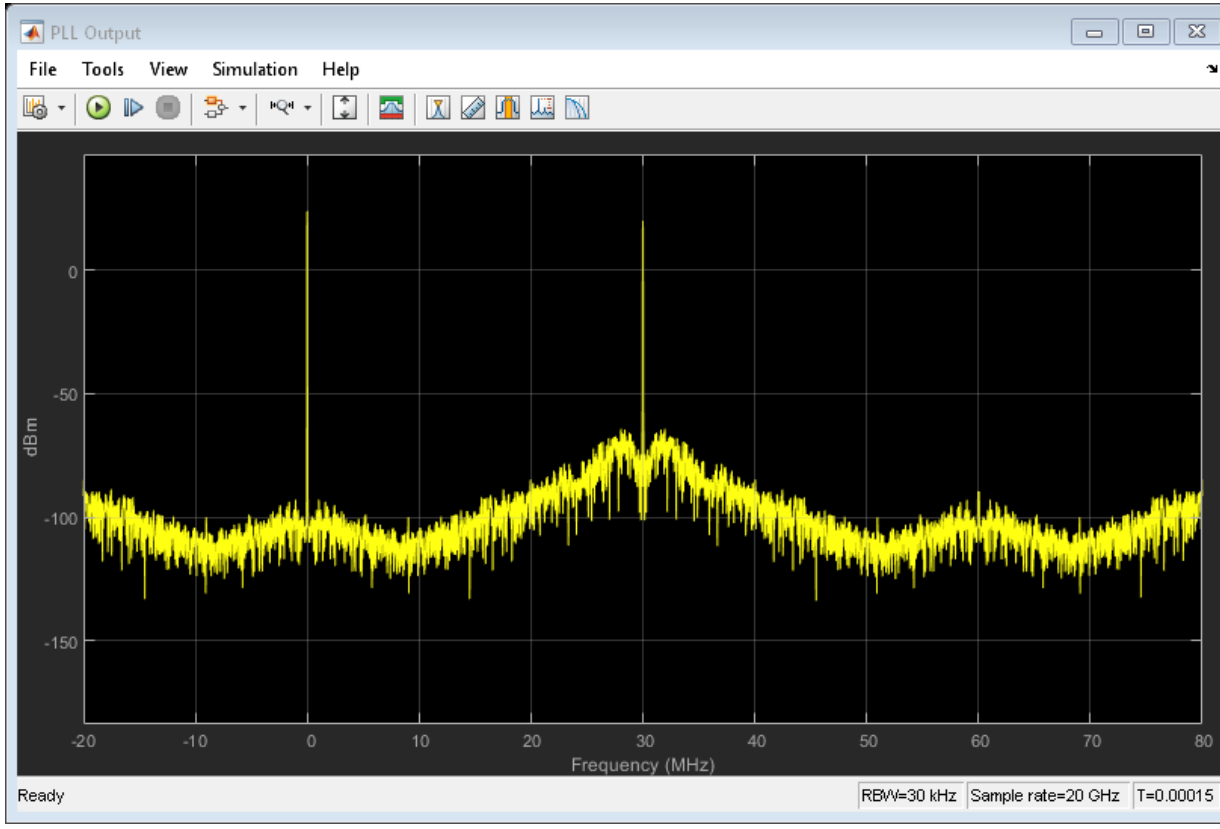
In both the PLL output spectrum and the PLL Testbench phase noise analysis, the phase noise sidebands nearest the main spectral component are significantly reduced. The sidebands beyond the loop bandwidth decay more slowly.

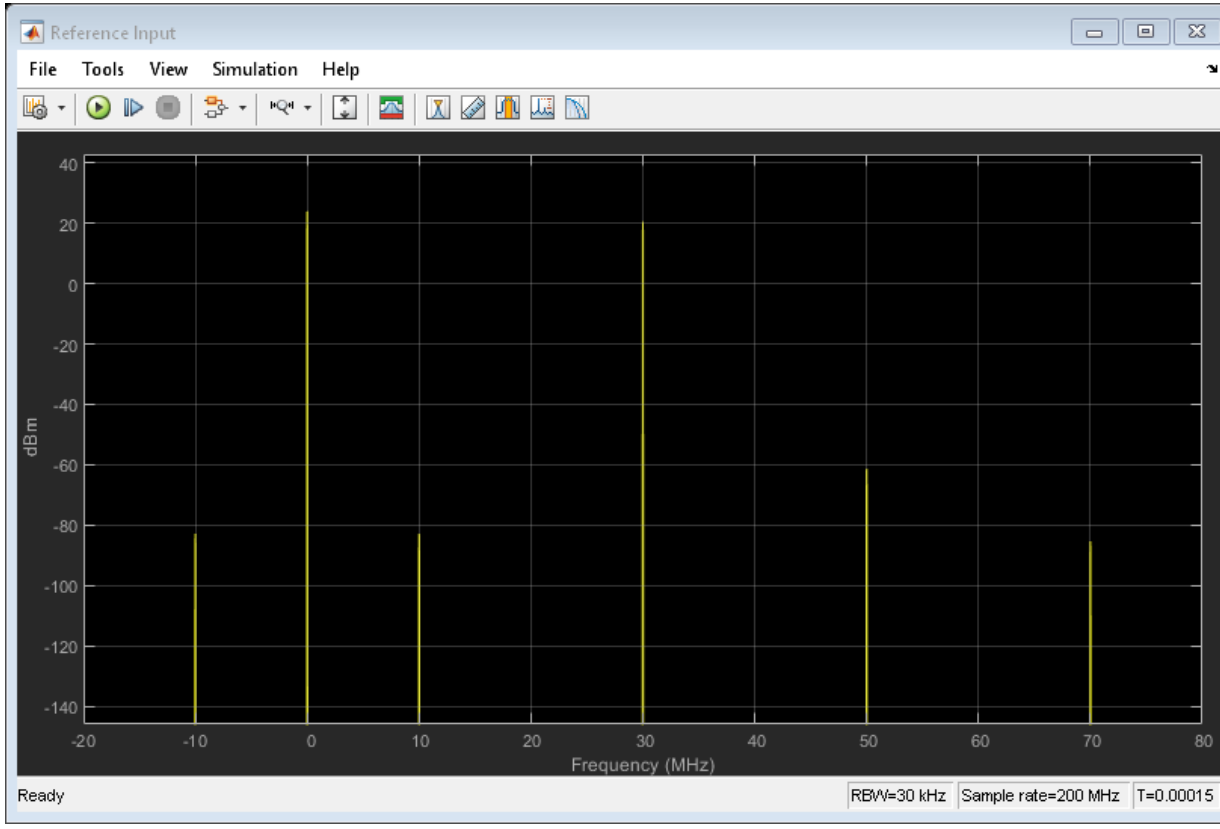
In the loop filter output spectrum, the spectral density at small frequency offsets is quite significant. The loop filter output cancels out a significant portion of the VCO phase noise close to the main output signal.

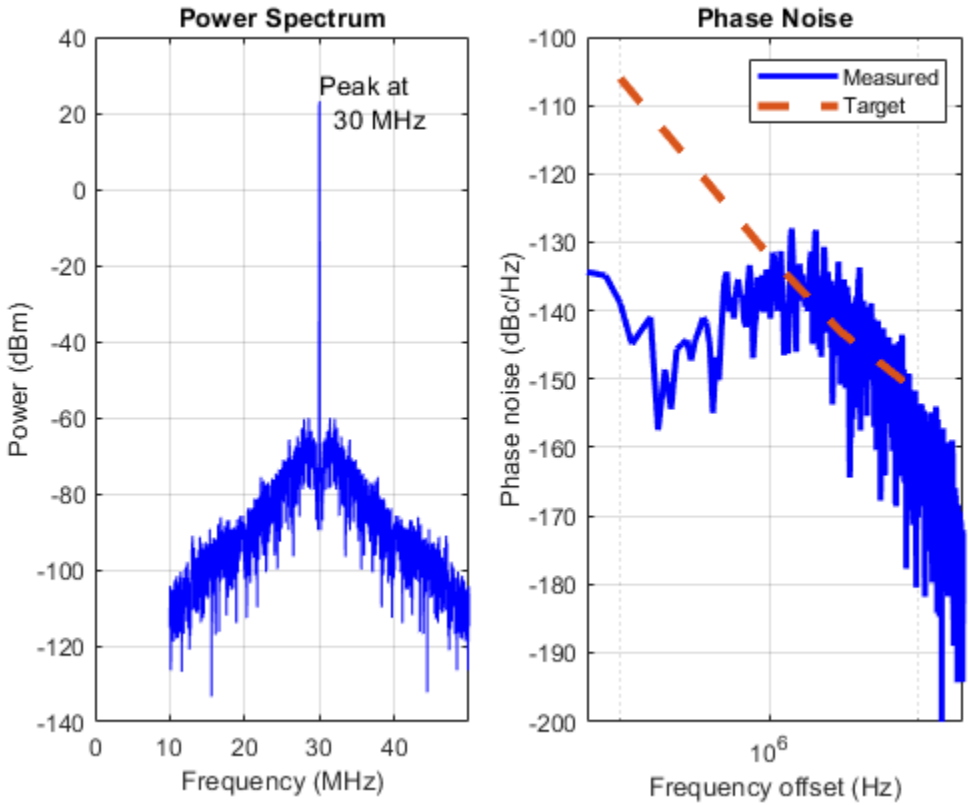
**Optional:** Run with different loop bandwidths, phase margins or filter orders and the the effect on the PLL output spectrum.

```
% Configure the model.
configurePhaseNoiseExample( 'PllPhaseNoiseExample', 30e6, 1, 'on', '1');
% Run the simulation
sim('PllPhaseNoiseExample.slx');
% Plot the PLL Testbench phase noise analysis
plotPhaseNoiseAnalysis();
```


Loop Filter
File
Tools
View
Simulation
Help
50
0
-50
-100
-150
dBm
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=5.22 GHz
T=0.00015


PLL Output
File
Tools
View
Simulation
Help
0
-50
-100
-150
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=20 GHz
T=0.00015


Reference Input
File
Tools
View
Simulation
Help
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=200 MHz
T=0.00015




### In-band Phase Noise

In this section, generate an in-band noise floor through the feedback prescaler. The PLL control loop filters the newly introduced phase noise. In a PLL with feedback prescaler ratio greater than one, the phase noise at the output of the feedback prescaler is a subsampled version of the VCO phase noise. The phase noise that the PLL control loop is responding to is therefore not a perfect replica of the VCO phase noise, resulting in incomplete suppression of the VCO phase noise. The resulting tracking error is commonly referred to as an in-band noise floor.

For this section, the feedback divider ratio is set to `70`, resulting in a PLL output frequency of `2.1` GHz. The unmodulated reference is chosen and the VCO phase noise is enabled.

- VCO free-run frequency: **N*x*fref**
- Feedback prescaler ratio: **N**
- VCO phase noise: Enabled
- Reference modulation: Through path selected

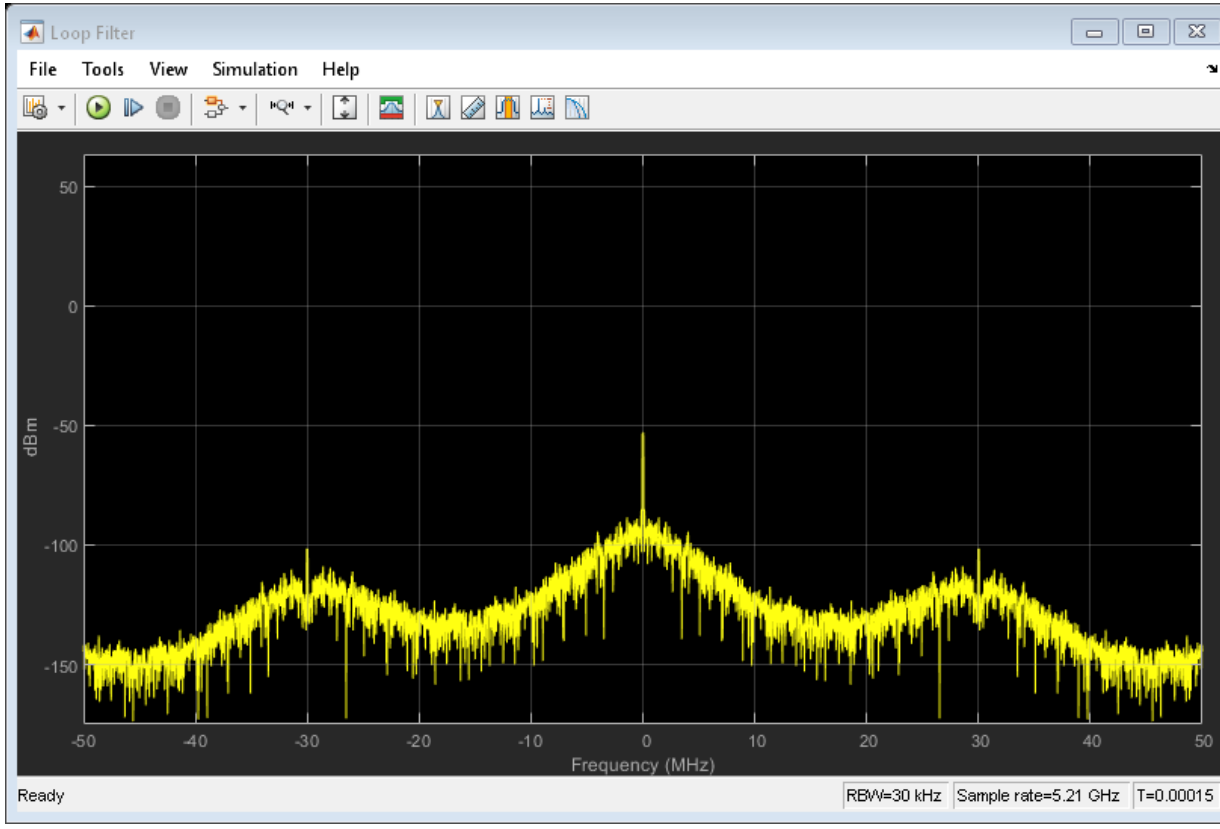
In contrast to the case with unity feedback prescaler ratio, the PLL output phase noise sidebands nearest the main spectral component are essentially constant. This is what is referred to as an "in-band noise floor".

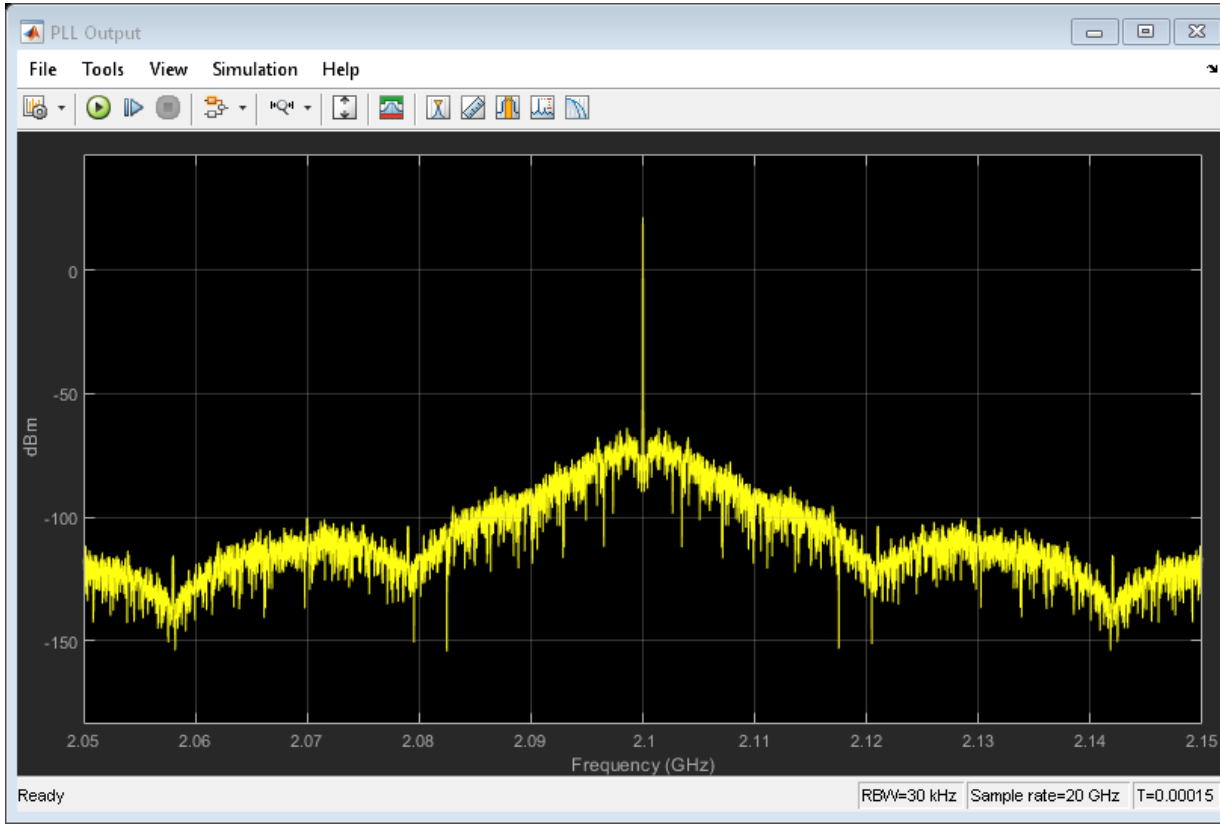
In the loop filter output spectrum, the spectral density at small frequency offsets is significant; however in this case the cancellation of the close-in VCO phase noise by the loop filter output is not as effective as in the case of a divider ration equal to one.

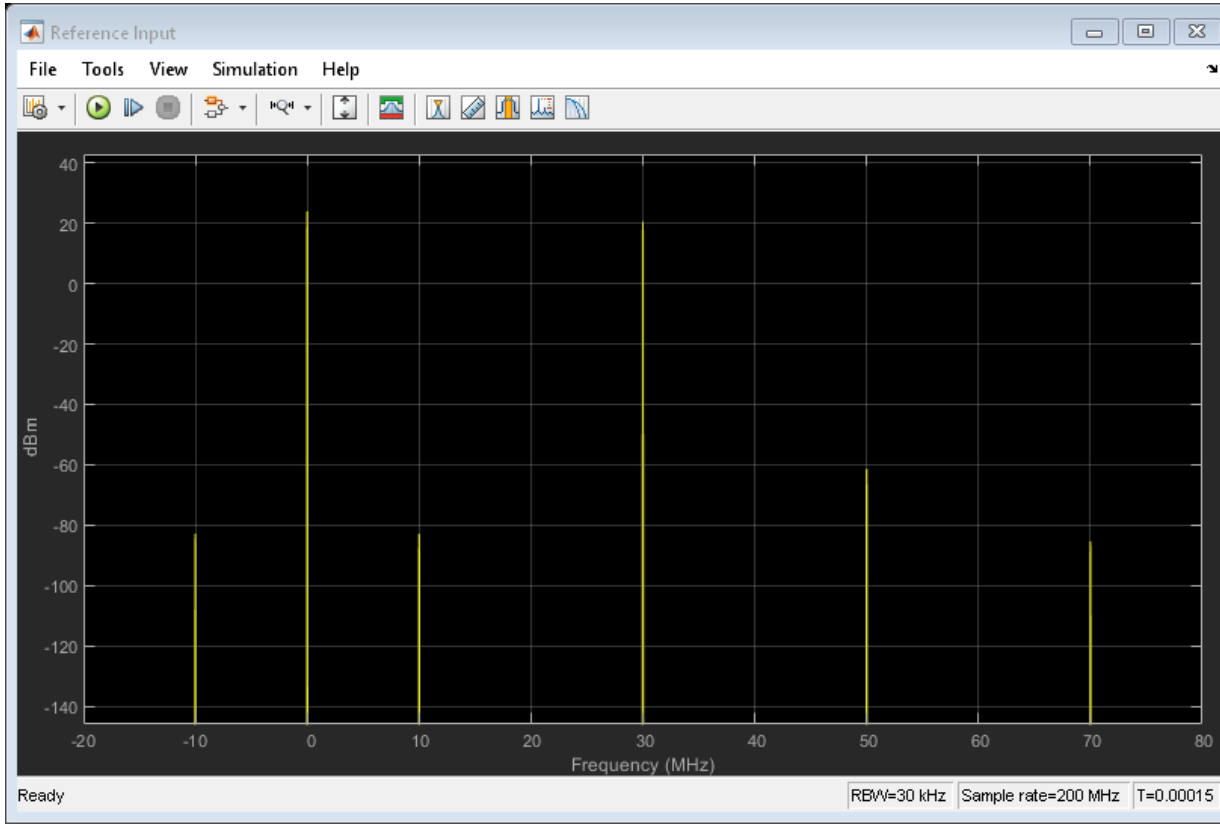
**Optional:** Experiment with other divider ratios and note how the in-band noise floor changes.

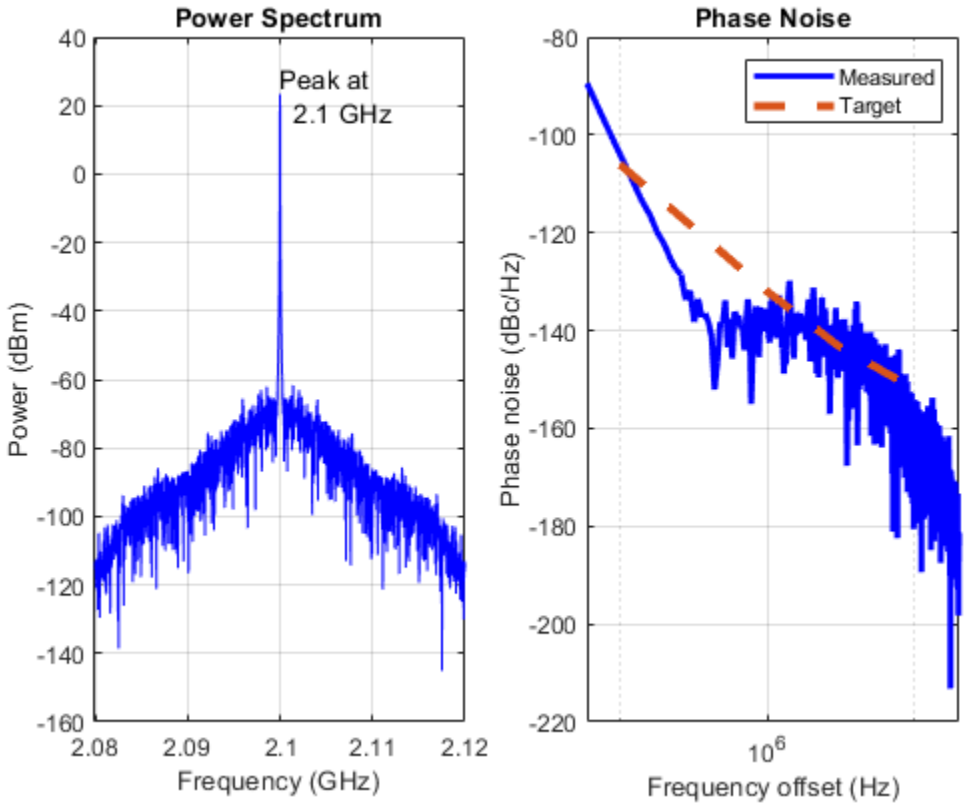
**Optional:** Run width different loop bandwidths, phase margins or filter orders and note the effect on the in-band noise floor.

```
% Configure the model.
configurePhaseNoiseExample( 'PllPhaseNoiseExample', 2.1e9, 70, 'on', '1');
% Run the simulation
sim('PllPhaseNoiseExample.slx');
% Plot the PLL Testbench phase noise analysis
plotPhaseNoiseAnalysis();
```


Loop Filter
File
Tools
View
Simulation
Help
50
0
-50
-100
-150
dBm
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=5.21 GHz
T=0.00015


PLL Output
File
Tools
View
Simulation
Help
0
-50
-100
-150
dBm
2.05
2.06
2.07
2.08
2.09
2.1
2.11
2.12
2.13
2.14
2.15
Frequency (GHz)
Ready
RBW=30 kHz
Sample rate=20 GHz
T=0.00015


Reference Input
File
Tools
View
Simulation
Help
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=200 MHz
T=0.00015



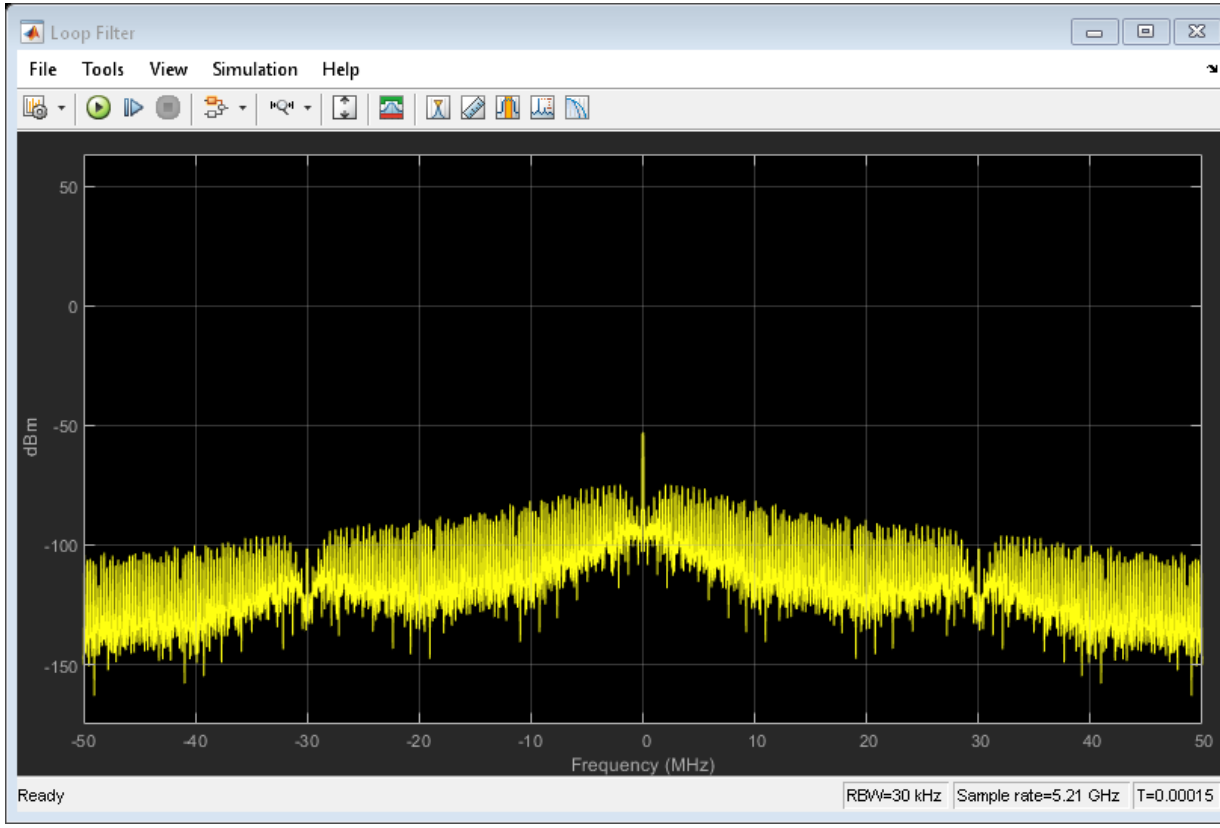

### Total Phase Noise

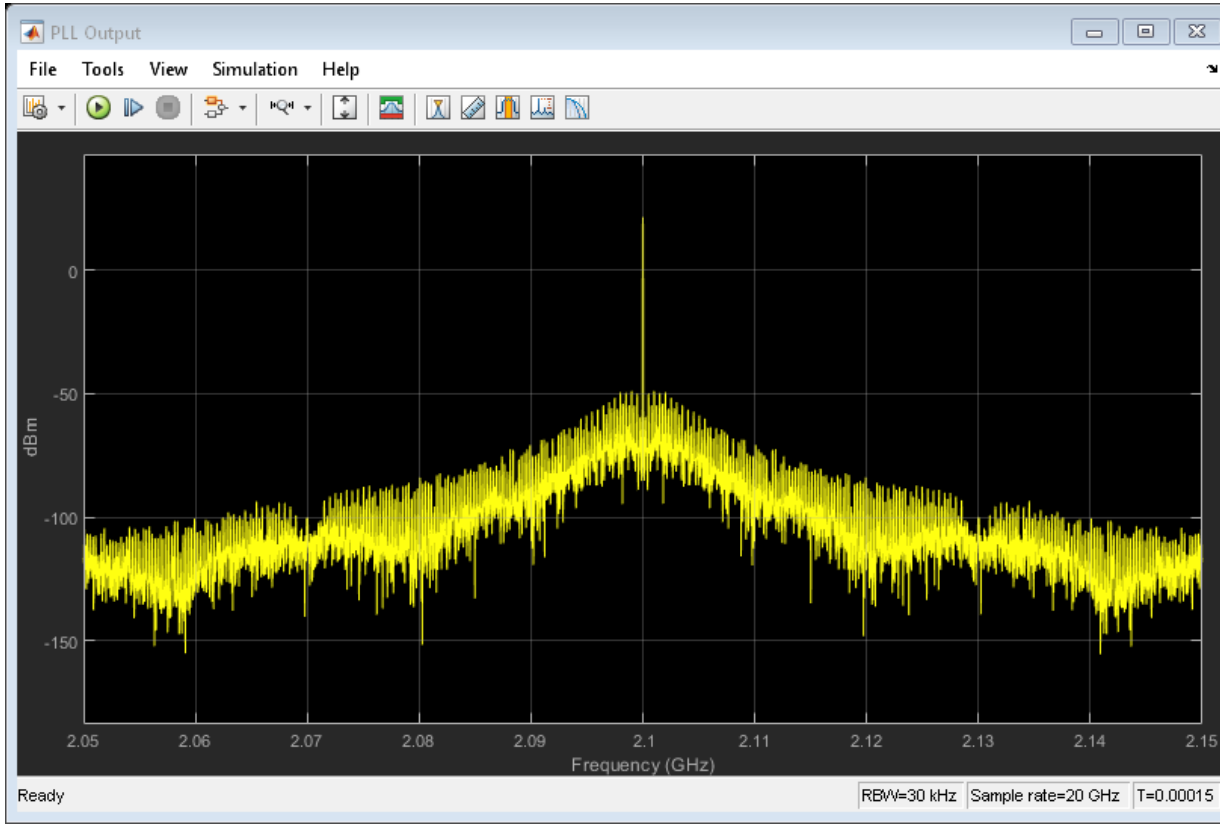
In this section, observe the total effect of all phase noise components at the PLL output.

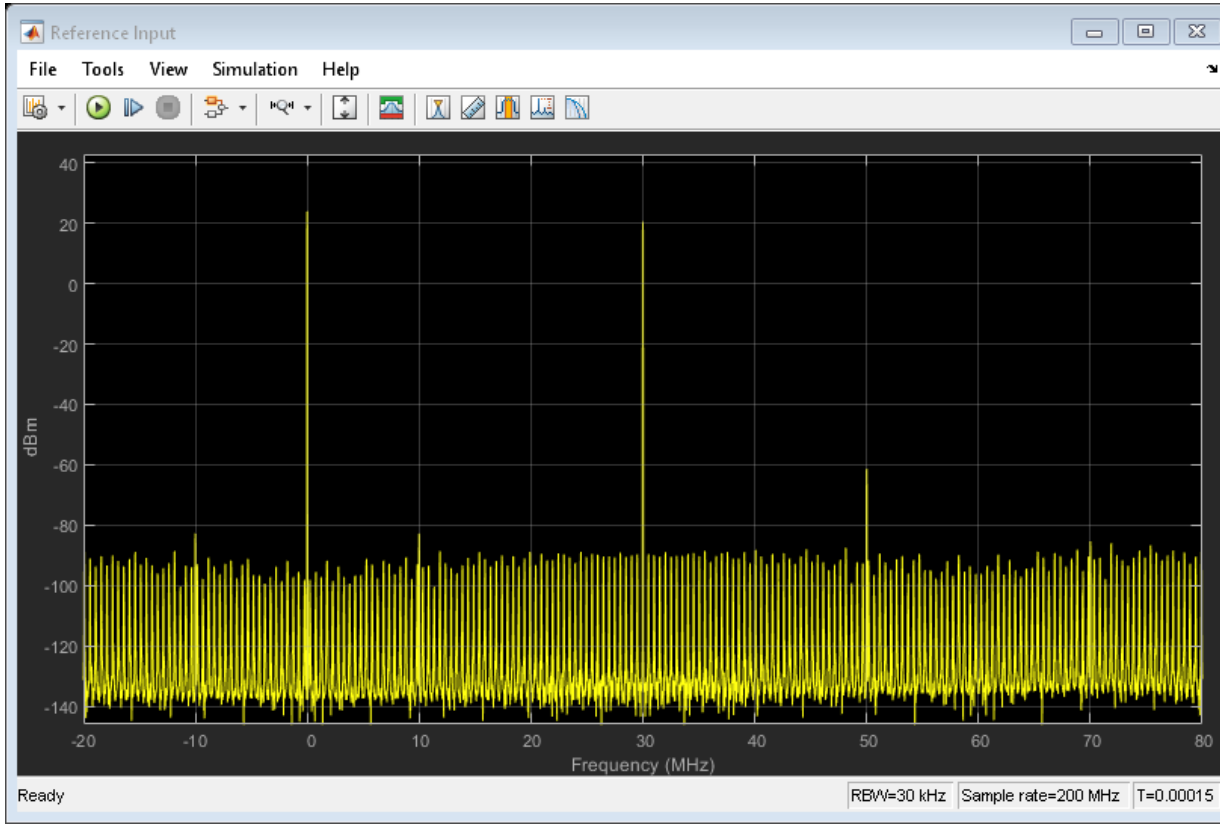
- VCO free-run frequency: **N*x*fref**
- Feedback prescaler ratio: **N**
- VCO phase noise: Enabled
- Reference modulation: *Phase modulated reference selected*

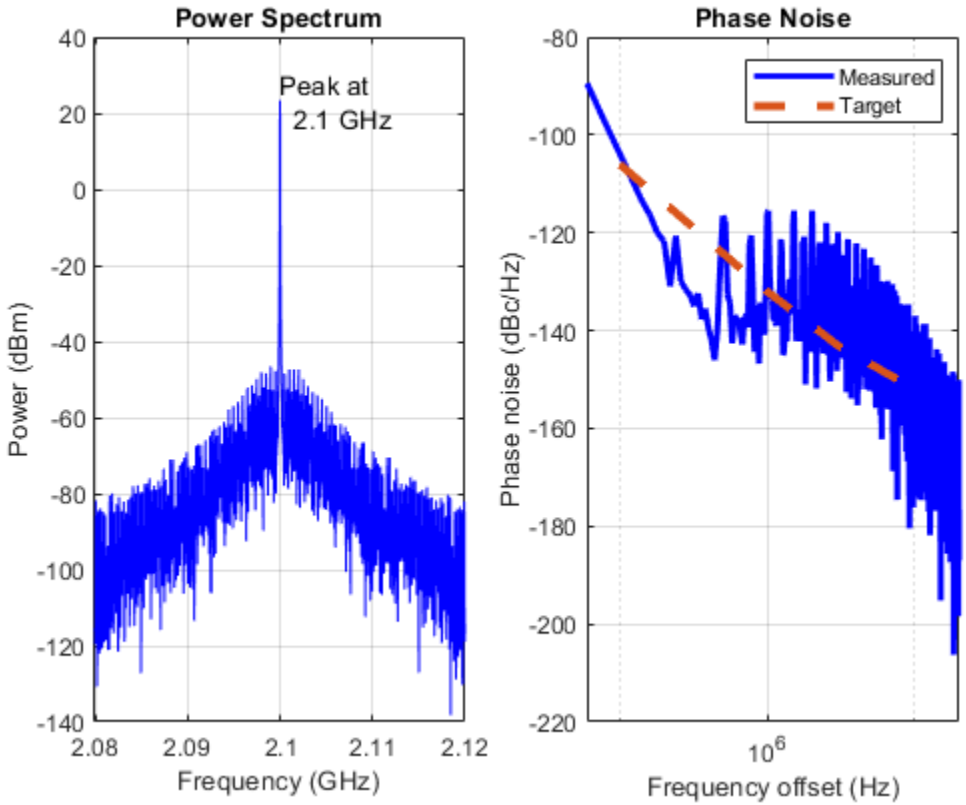
```
% Configure the model.
configurePhaseNoiseExample( 'PllPhaseNoiseExample', 2.1e9, 70, 'on', '0');
% Run the simulation
sim('PllPhaseNoiseExample.slx');
```

```
% Plot the PLL Testbench phase noise analysis
plotPhaseNoiseAnalysis();
```




PLL Output
File
Tools
View
Simulation
Help
0
-50
-100
-150
dBm
2.05
2.06
2.07
2.08
2.09
2.1
2.11
2.12
2.13
2.14
2.15
Frequency (GHz)
Ready
RBW=30 kHz
Sample rate=20 GHz
T=0.00015


Reference Input
File
Tools
View
Simulation
Help
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
-20
-10
0
10
20
30
40
50
60
70
80
Frequency (MHz)
Ready
RBW=30 kHz
Sample rate=200 MHz
T=0.00015


Power Spectrum
Peak at
2.1 GHz
Power (dBm)
Frequency (GHz)
40
20
0
-20
-40
-60
-80
-100
-120
-140
2.08
2.09
2.1
2.11
2.12
Phase Noise
Measured
Target
Phase noise (dBc/Hz)
Frequency offset (Hz)
-80
-100
-120
-140
-160
-180
-200
-220
10^6

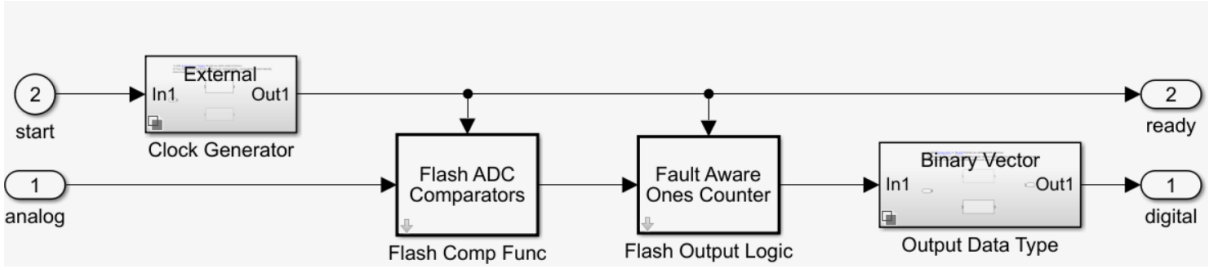
# ADC Featured Examples

# Effect of Metastability Impairment in Flash ADC

This example shows how to customize a flash ADC by adding the metastability probability as an impairment and how to measure the said impairment. This is to validate our impairment implementation. The example also shows the effect of metastability on the dynamic performance (AC analysis) of the flash ADC. When the digital output from a comparator is ambiguous (neither '1' nor '0') , 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.

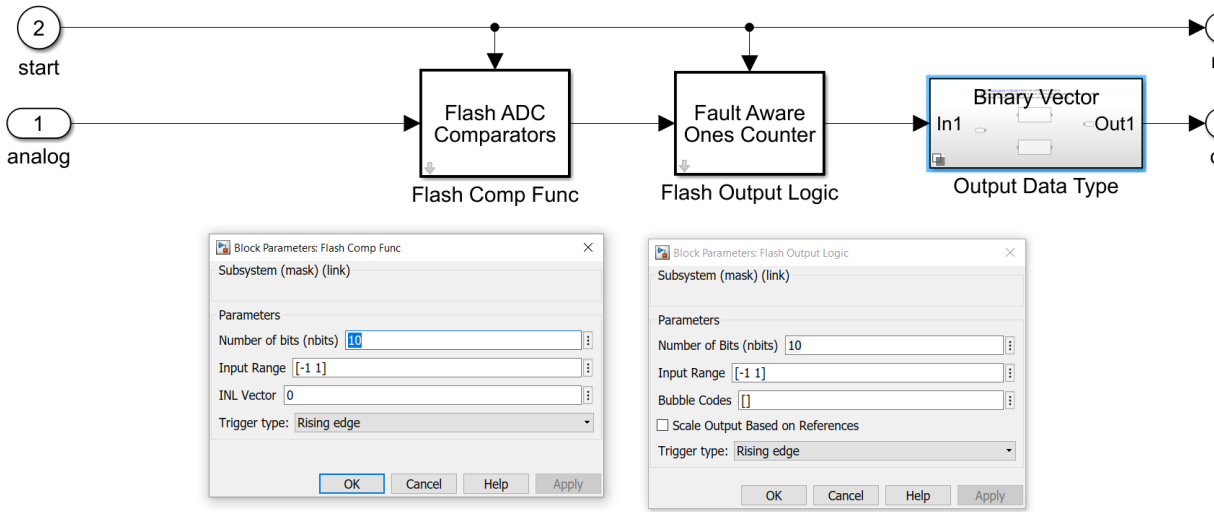
### Customize the flash ADC

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 paste the complete structure to another new blank canvas.




For this example, we delete the 'Clock Generator' block as it is not used to provide the start conversion clock. An external stimuli block is used for that purpose. Barring the clock generator block, flash ADC consists of 2 major components:

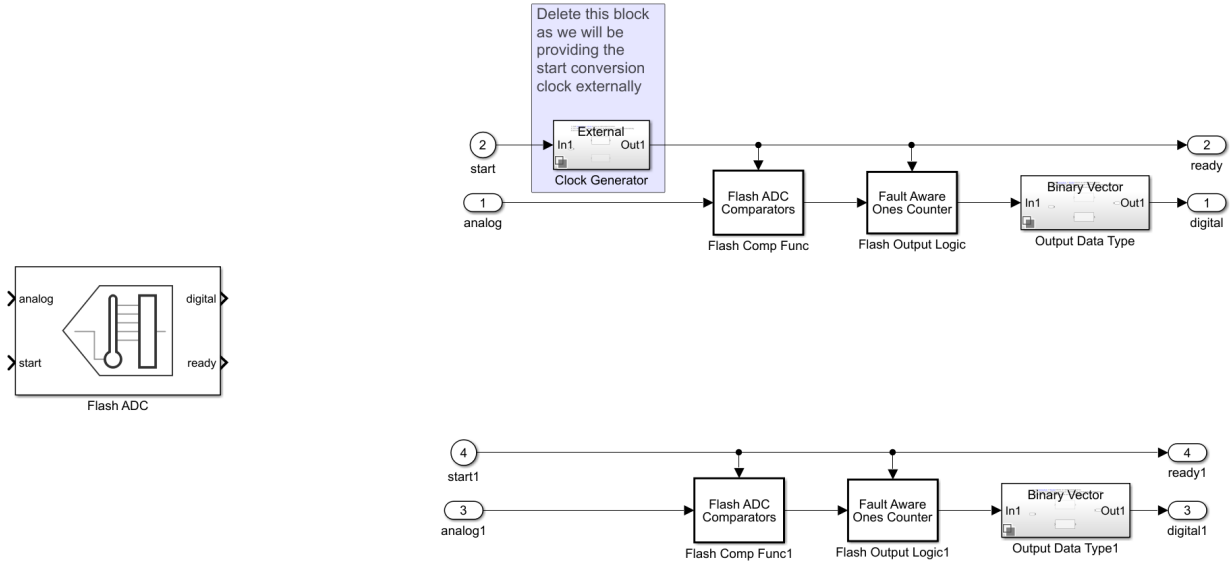
- Flash ADC comparators: The flash ADC uses 2^NBits comparators in parallel. The block itself is based on MATLAB code. Before simulation starts, they calculate the individual reference voltages and store them in a vector. On every specified edge, the input is compared to the references using MATLAB's ability to compare vectors. This generates thermometer code in much the same way the real flash ADC does, without the lag from (NBits) individual comparator blocks in the model. Various parameters of its mask will be configured as follows




1. Number of bits: 10 (we will be creating the example for a 10-bit ADC)
2. Input range: [-1 1] (the default setting)
3. INL vector: 0 (we want to disable this impairment to see the real effect of metastability)

- Fault Aware Ones Counter: This block is an impairment and the other major half of the flash ADC architecture rolled into one. Real ADCs handle conversion from thermometer to binary through logic circuits. This block just takes the sum-of-elements of the vector. It then applies that sum to a lookup table to simulate missing codes, otherwise known as bubbles. The various parameters for this block are like flash ADC comparator block mentioned above. The only additional parameter is the missing codes which is entered as an empty vector.

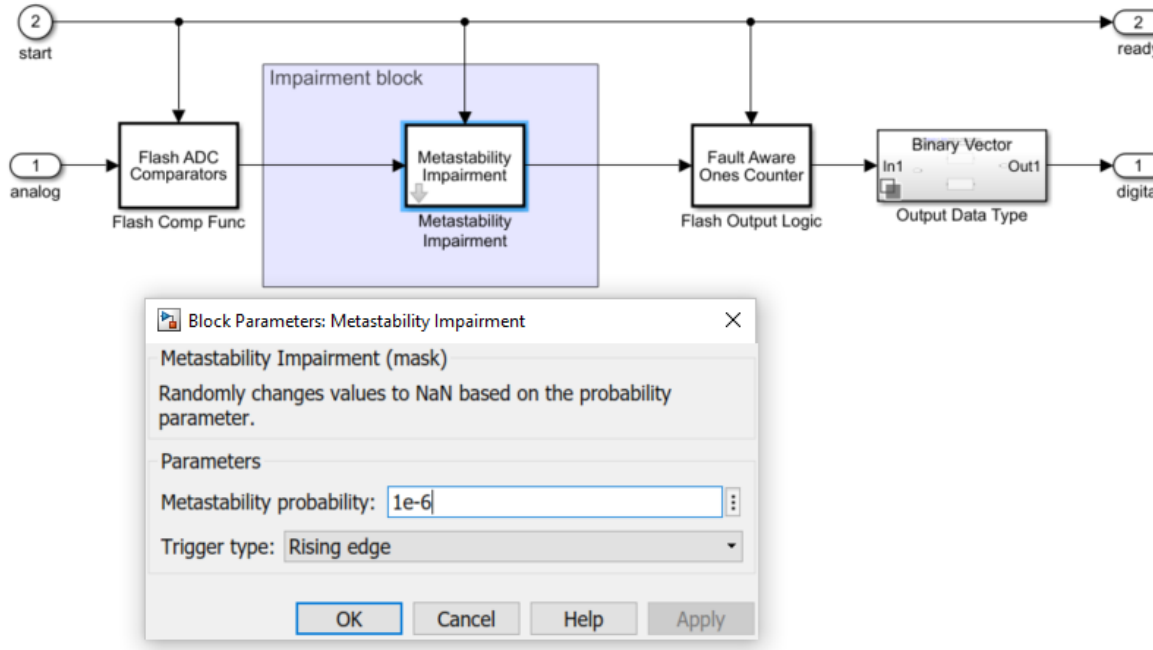
What we see from the image below is a top view of the flash ADC block on the left side and its inner structure pasted on the top right side. Bottom right is the structure with only the important components required for this example.




### Implement Metastability Probability as an Impairment to Flash ADC

To add metastability impairment we place a MATLAB function block soon after the flash ADC comparators as shown in the figure below. This block sets thermometer code signals to NaN with a probability specified in its mask using a uniform random number generator. It resets them on the next relevant edge. MATLAB code behind the impairment block is as follows:

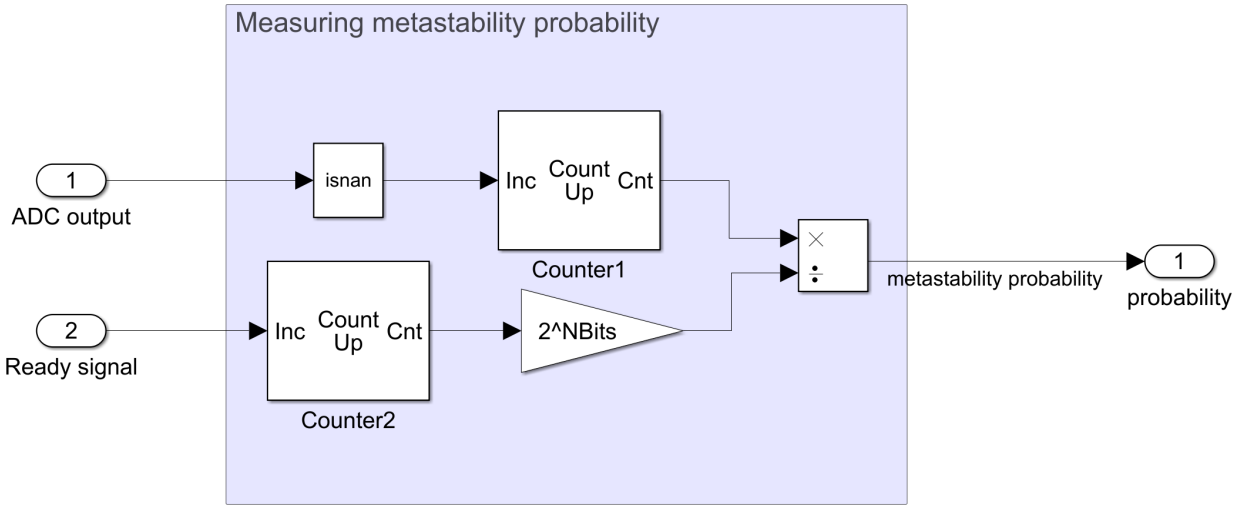
```
%  function y = metastability(u, Probability)
%   mult = ones(size(u));
%   mult(rand(size(u)) < Probability(1)) = NaN; % metastability = NaN
%   y = u .* mult;
%  end
```




As you can observe from the image above the mask of the impairment block requires one parameter i.e. metastability probability that user wants to implement in the flash ADC block.

### Implement Measuring Metastability Probability

The idea behind measuring metastability impairment is very simplistic. We 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 shown below.

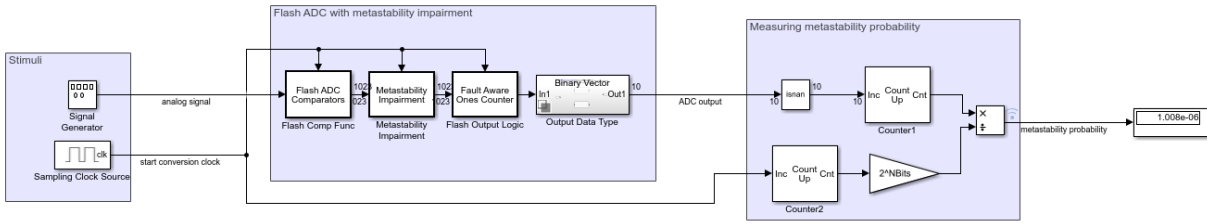


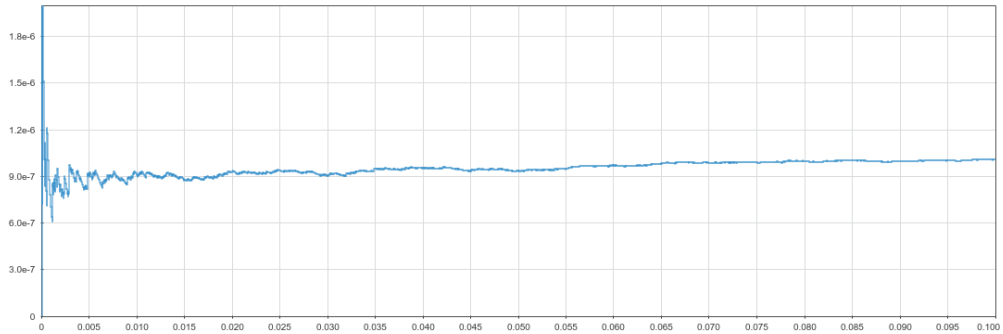

The Inports are: * ADC output: This port takes in the output digital code generated by the flash ADC. * Ready signal: This port takes in the ready signal signifying the rate at which the digital conversion is taking place. at each rising edge of the Ready signal digital code gets generated.

**Simulation for Metastability Measurement**

Below is the model combining the customized flash ADC with its output connected to the metastability probability measurement system. In the model we have a 10-bit flash ADC with metastability probability of 1e-6 added. the stimuli block generates an analog signal of 100Hz and a start conversion clock with a frequency of 100MHz (this will be ADC's rate of operation). We have a dashboard scope which provides the behavior of the probability number over time. There is a display block also to provide the current probability being measured by the subsystem. One needs to run the simulation for a longer period to see the probability number getting settled to the desired value (1e-6 in this case)

```
NBits=10;
model1='flashAdc_metastability.slx';
open_system(model1);
sim(model1);
```

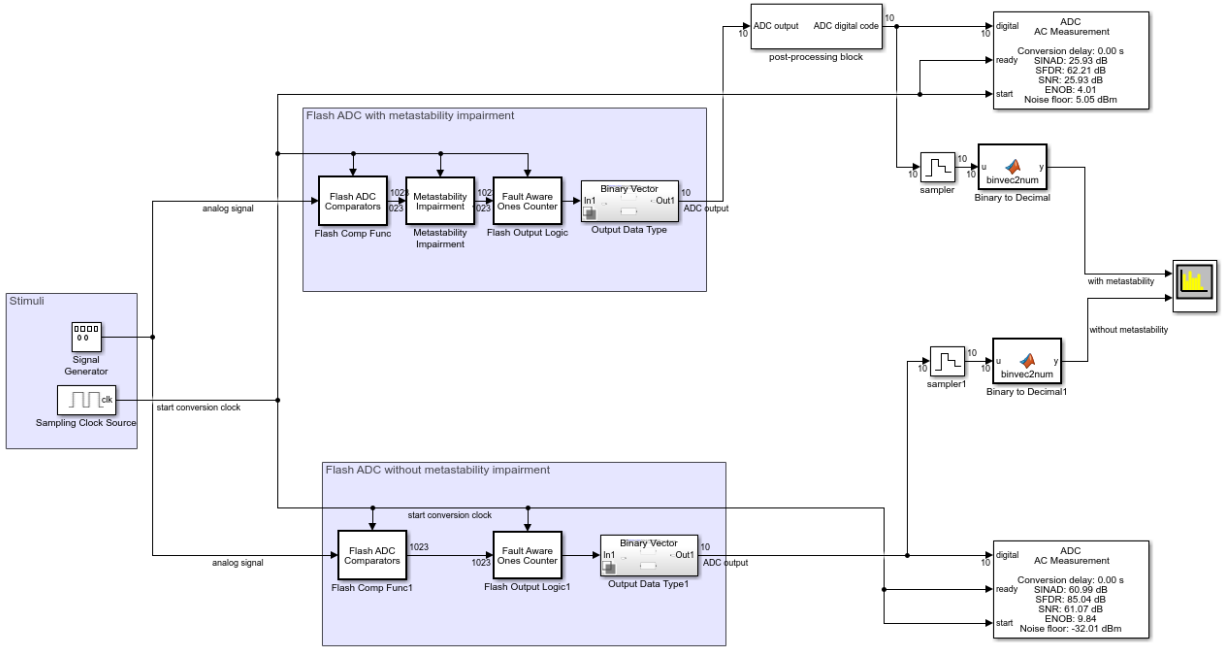


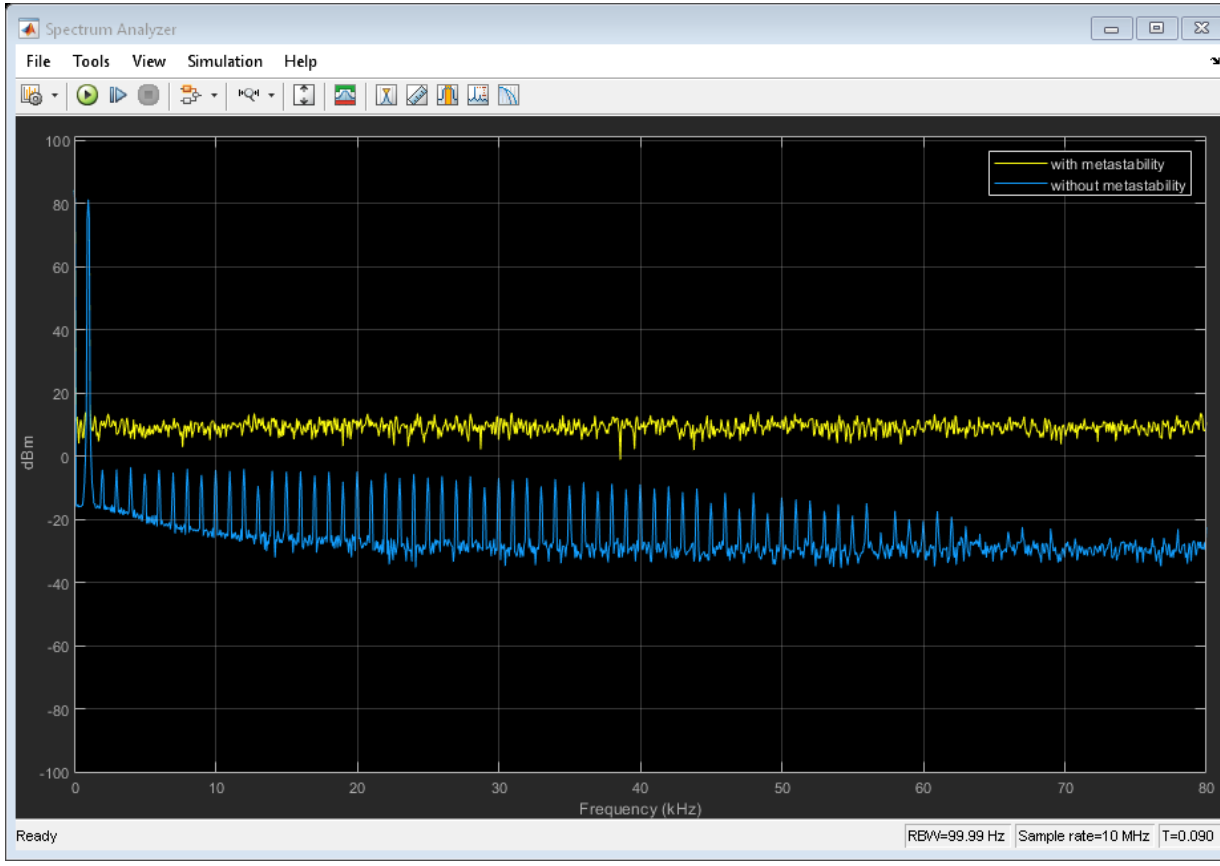




### Effect of Metastability on Dynamic Performance of ADC

Here we 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 it. We have a post processing block that takes in the impaired digital output and converts the NaNs to a 0. This is because the digital output with NaNs can't be recognized by a spectrum analyzer as valid signal for spectral analysis. We also hookup ADC AC measurement block to observe various performance metrics like SNR, ENOB, noise floor etc. The simulation results show the AC analysis brings 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);
```


Stimuli
Signal Generator
Sampling Clock Source
clk
analog signal
start conversion clock
Flash ADC with metastability impairment
Flash ADC Comparators
Flash Comp Func
Metastability Impairment
Metastability Impairment
Fault Aware Ones Counter
Flash Output Logic
Binary Vector
In1
Out1
Output Data Type
ADC output
post-processing block
ADC output
ADC digital code
digital
ready
start
ADC AC Measurement
Conversion delay: 0.00 s
SINAD: 25.93 dB
SFDR: 62.21 dB
SNR: 25.93 dB
ENOB: 4.01
Noise floor: 5.05 dBm
sampler
binvec2num
Binary to Decimal
with metastability
without metastability
sampler1
binvec2num
Binary to Decimal1
Flash ADC without metastability impairment
start conversion clock
Flash ADC Comparators
Flash Comp Func1
Fault Aware Ones Counter
Flash Output Logic1
Binary Vector
Output Data Type1
ADC output
analog signal
digital
ready
start
ADC AC Measurement
Conversion delay: 0.00 s
SINAD: 60.99 dB
SFDR: 85.04 dB
SNR: 61.07 dB
ENOB: 9.84
Noise floor: -32.01 dBm


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

# Mixing Analog and Digital Signals Featured Examples

# Digital Timing using Solutions to Ordinary Differential Equations

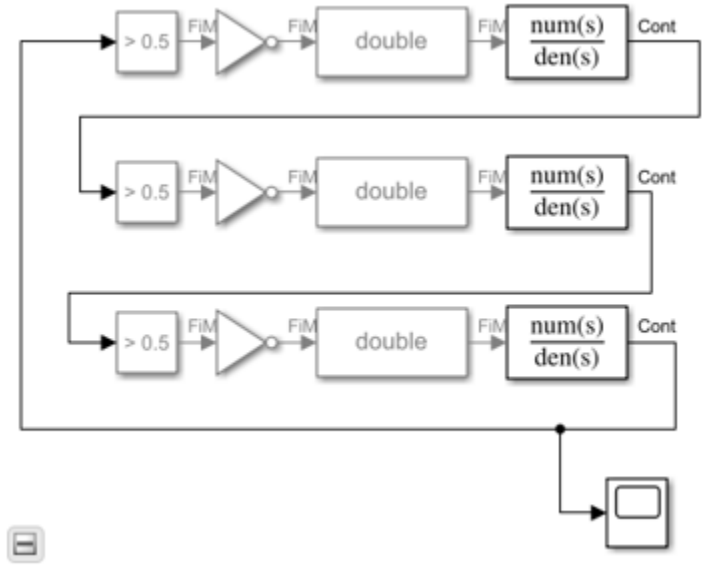
This example shows how to model a three stage ring oscillator using models defined by ordinary differential equations (ODE).

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');
```



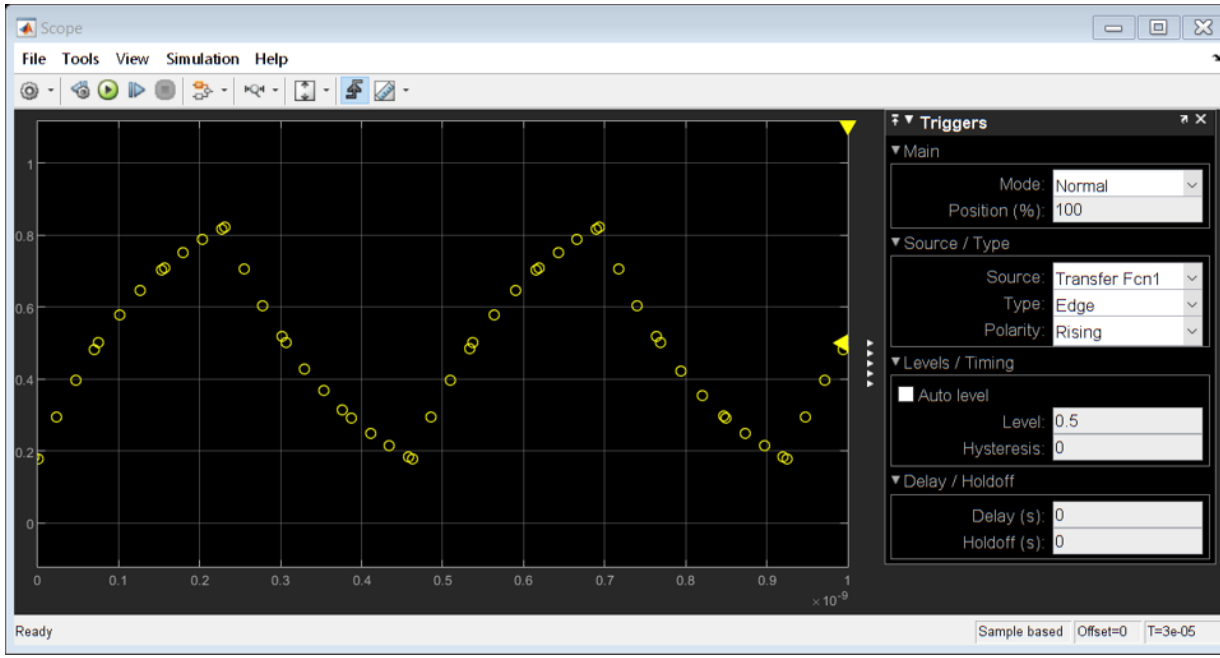

### Continuous Time Model Using Exponential Decay

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');
```



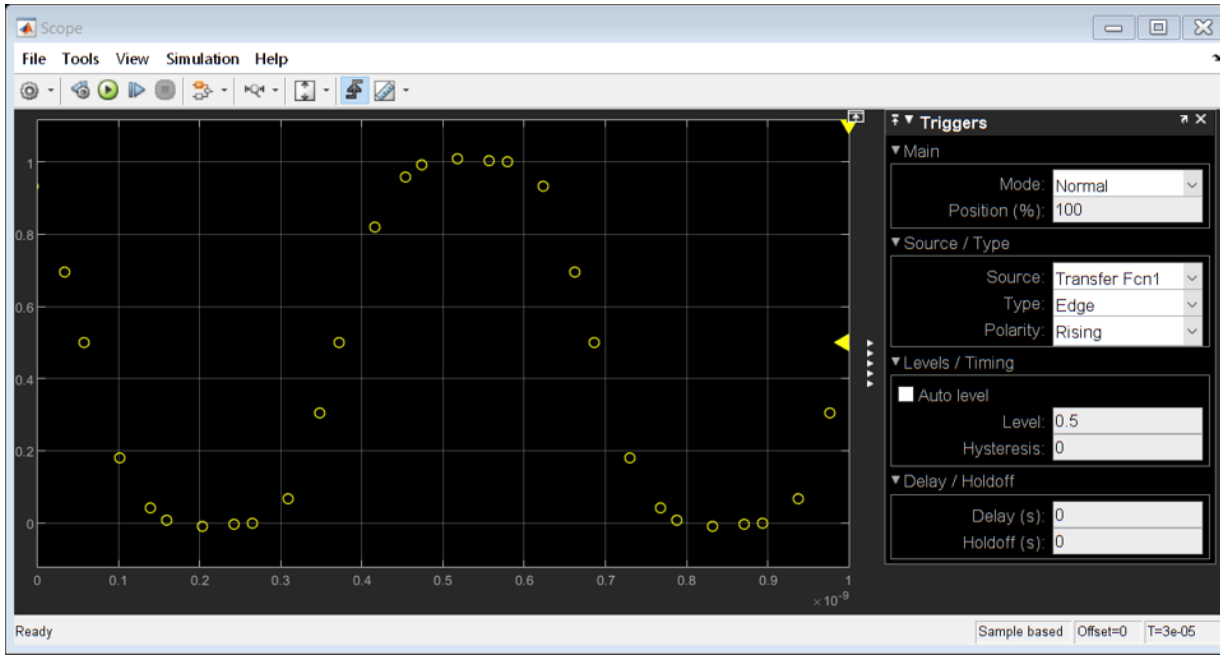

### Continuous Time Model with Nearly Constant Slew Rate

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**.

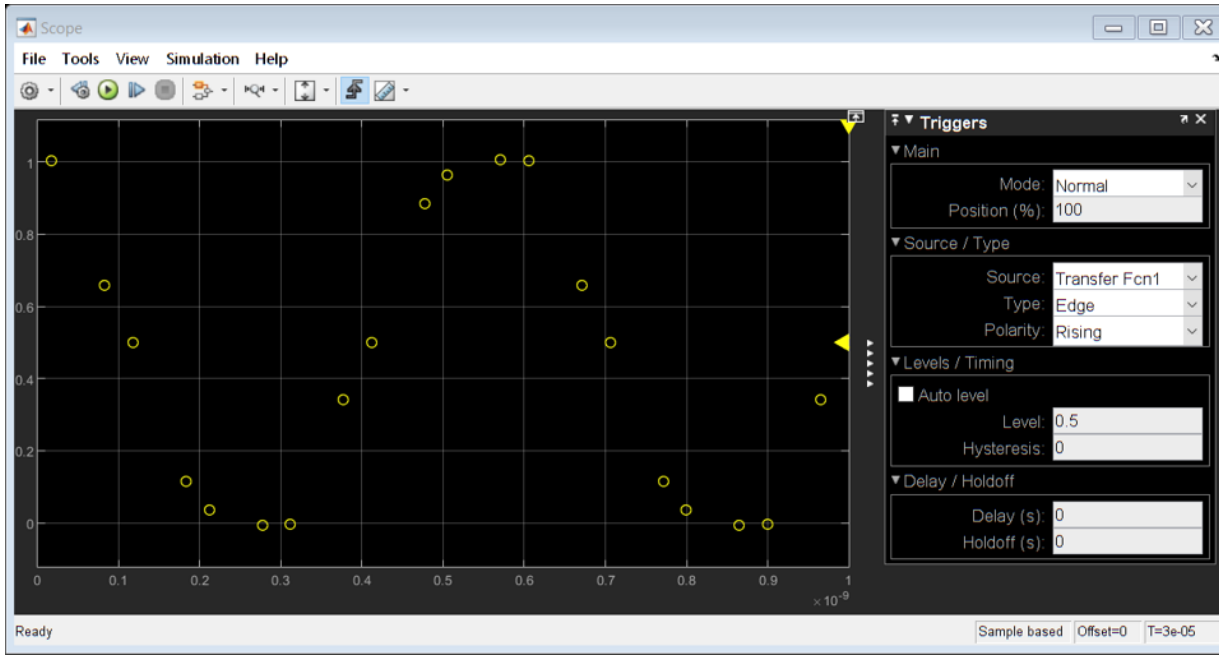
Observe both the change in period of oscillation and in wave shape.

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.

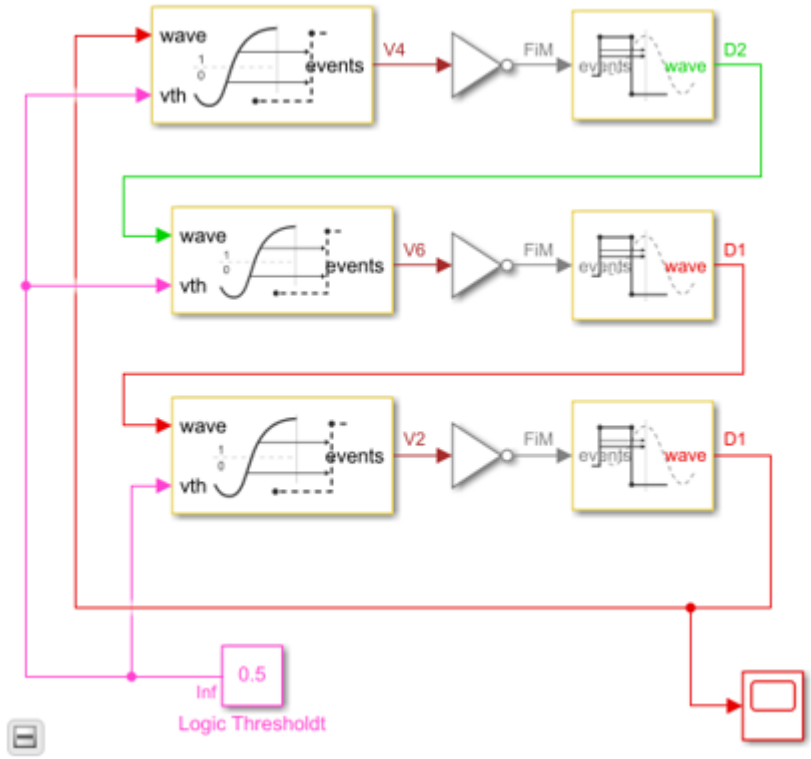
For a variable step input, the output of the Logic Decision block is delayed by the minimum delay parameter for the block.

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.

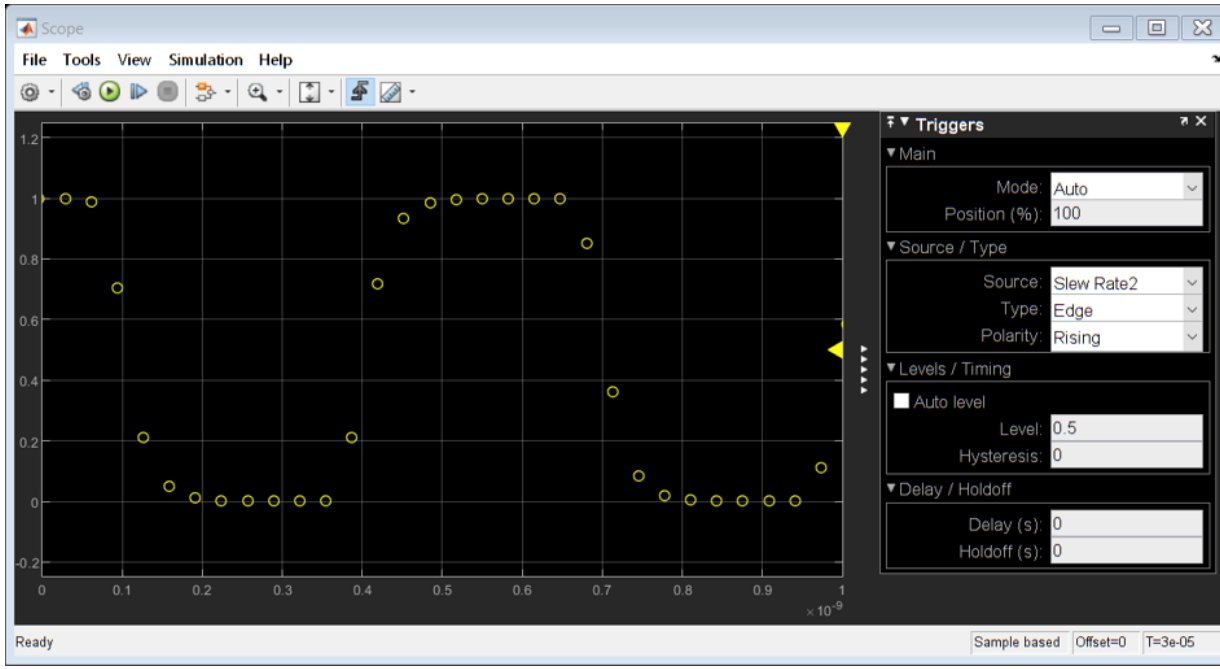
For this section, the Slew Rate blocks have been configured to their **Default** sampling mode, which maximizes the portion of the delay due to nearly constant slew rate (saturated transistor) and minimizes the delay due to constant delay or exponential decay.

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

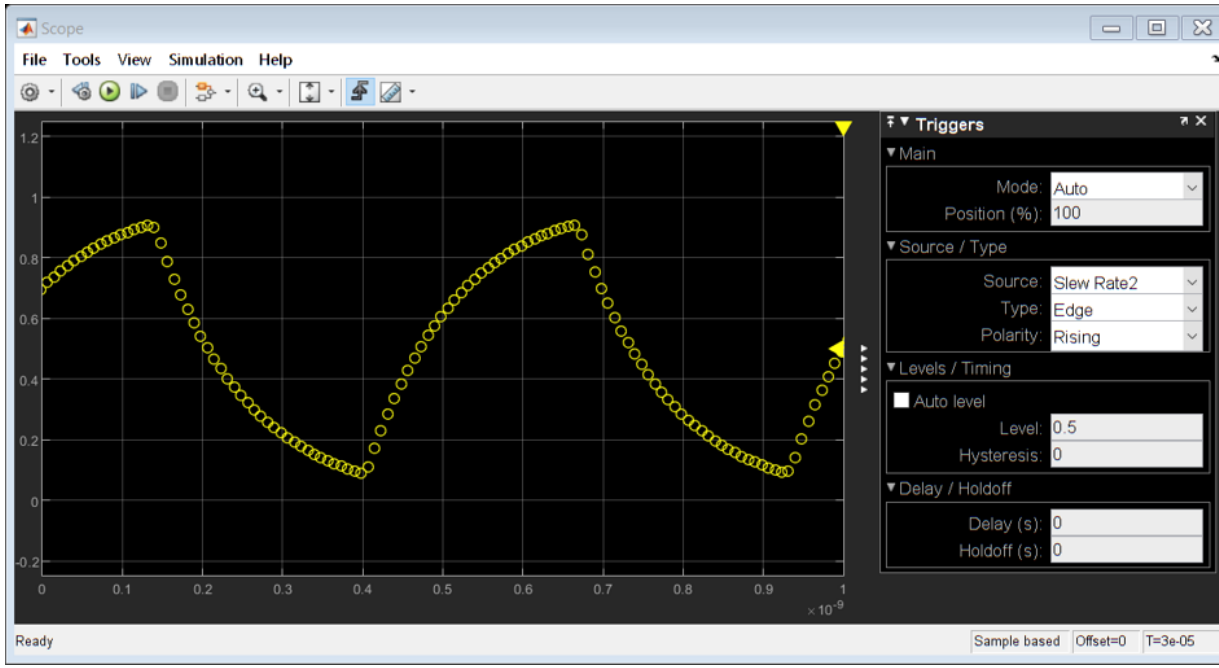
In this section, use the **Advanced** sampling mode of the Slew Rate block to model circuits whose response is primarily a decaying exponential.

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.

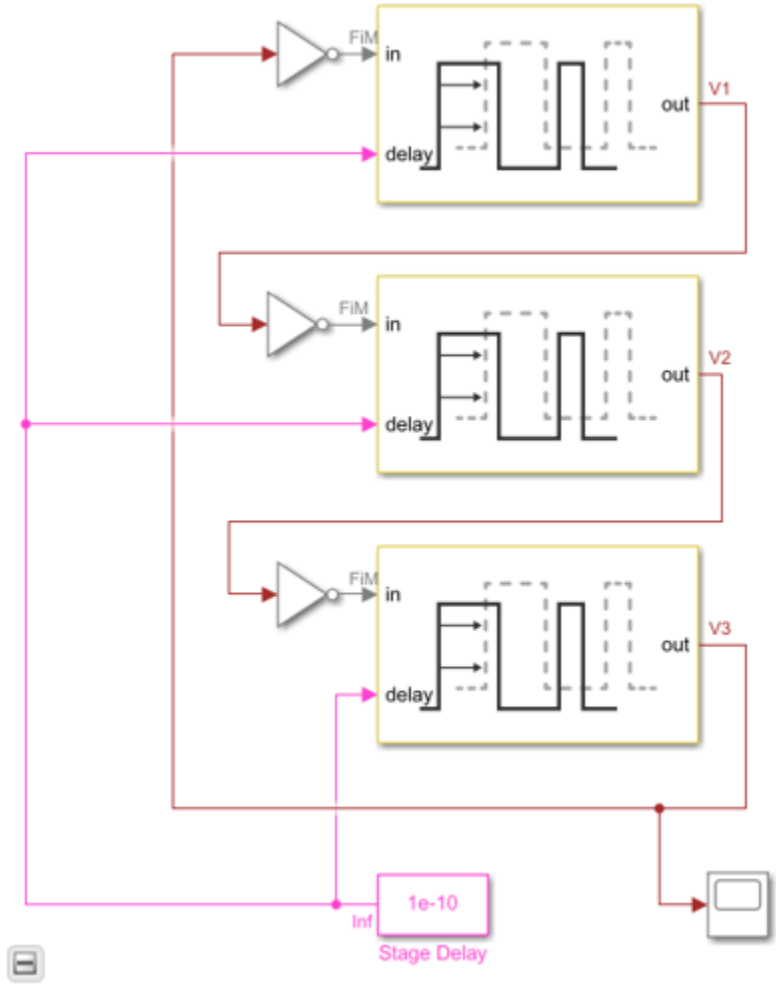
As indicated by the sample time color coding, the output sample time for the inverters is Fixed In Minor Step (FIM). This means that each inverter will produce an output sample value for every major sample time in the model, regardless whether or not that sample time is used at an input port of the gate.

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).

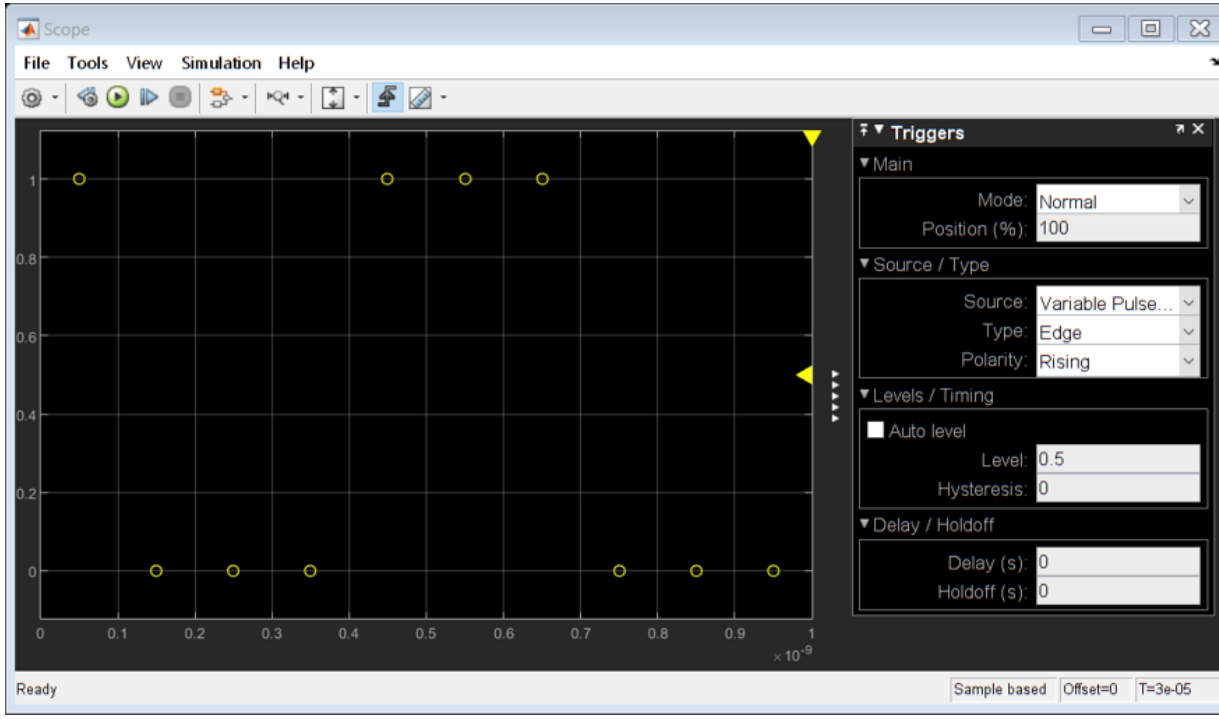
Since the model does not contain any differential equations, the solver is **Variable Step Discrete**.

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

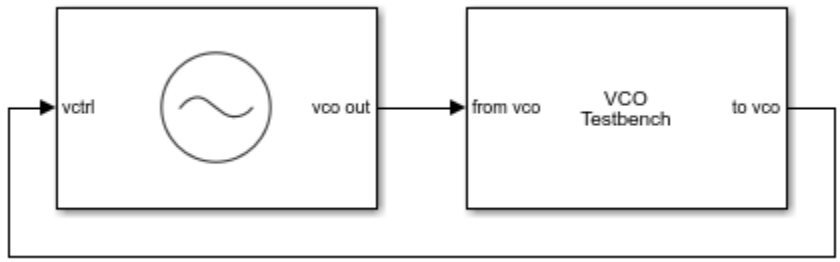
# PLL Block Level Examples

# 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.

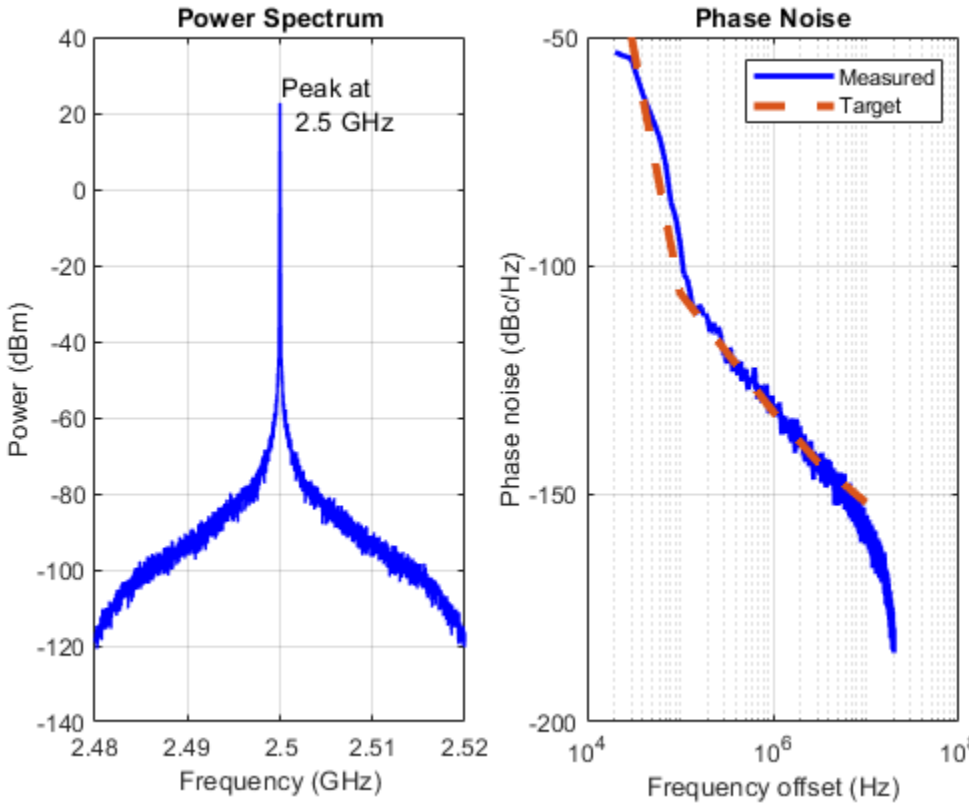
```
open_system('vcoPhaseNoise.slx')
```




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.



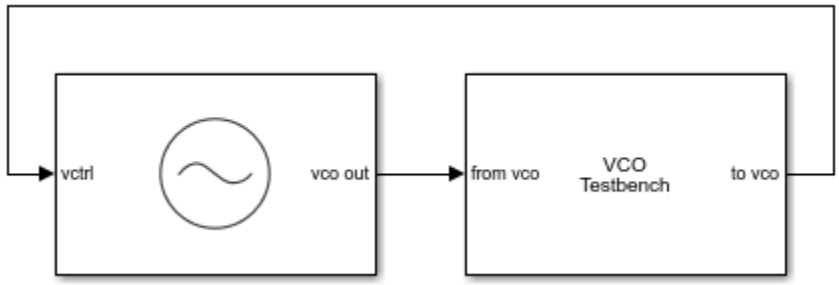

The operating frequency matches the predicted frequency `2.5` GHz. The measured phase noise profile also matches the target profile.

# Finding Voltage Sensitivity and Quiescent Frequency of VCO

This example shows how to find VCO metrics such as voltage sensitivity ($K_{vco}$) and quiescent frequency or free running frequency ($F_o$).

Open the model `vcoCharacteristics` . The model consists of a VCO block and a VCO Testbench.

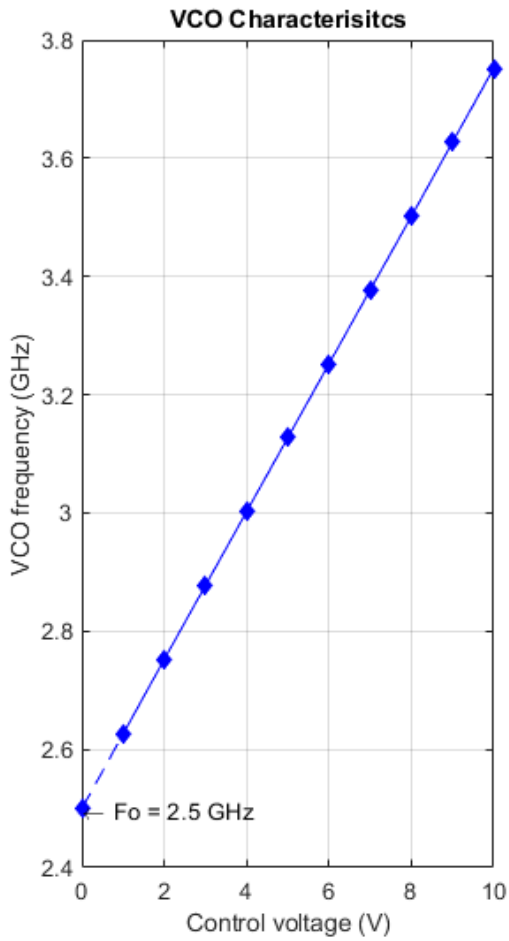
```
open_system('vcoCharacteristics.slx')
```

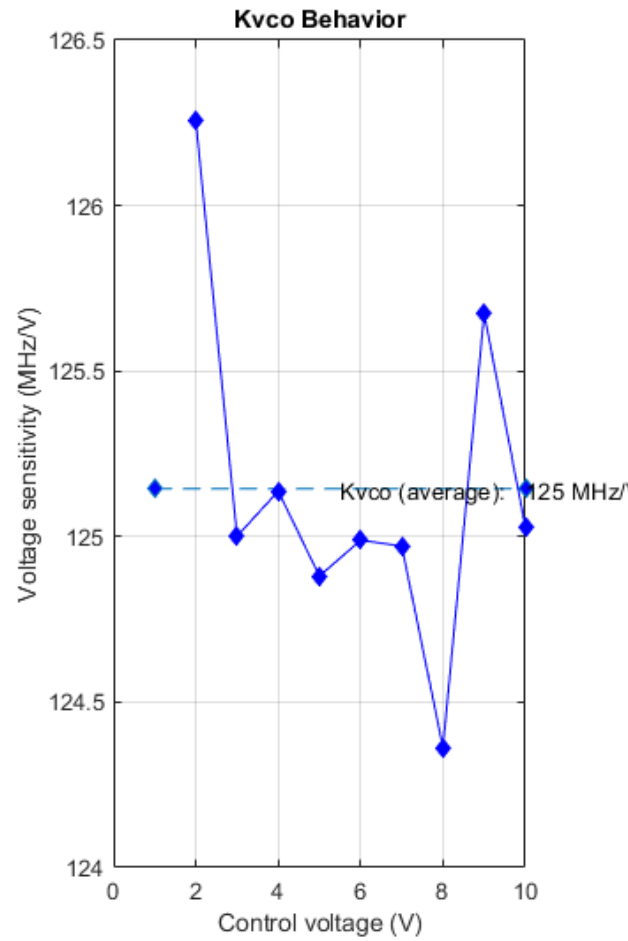



The testbencch is set to measure the **Kvco and Fo** metric of the VCO in **Measurement** option. **Range of control voltage (V)** provided to the input of VCO is set to `[1 10]`.

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.

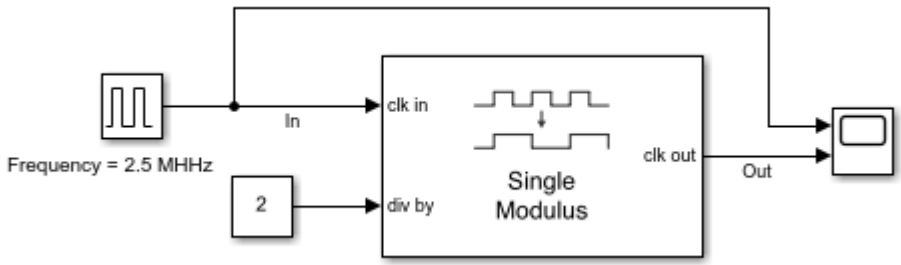

VCO Characterisitcs
VCO frequency (GHz)
Control voltage (V)
Fo = 2.5 GHz
3.8
3.6
3.4
3.2
3
2.8
2.6
2.4
0
2
4
6
8
10


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

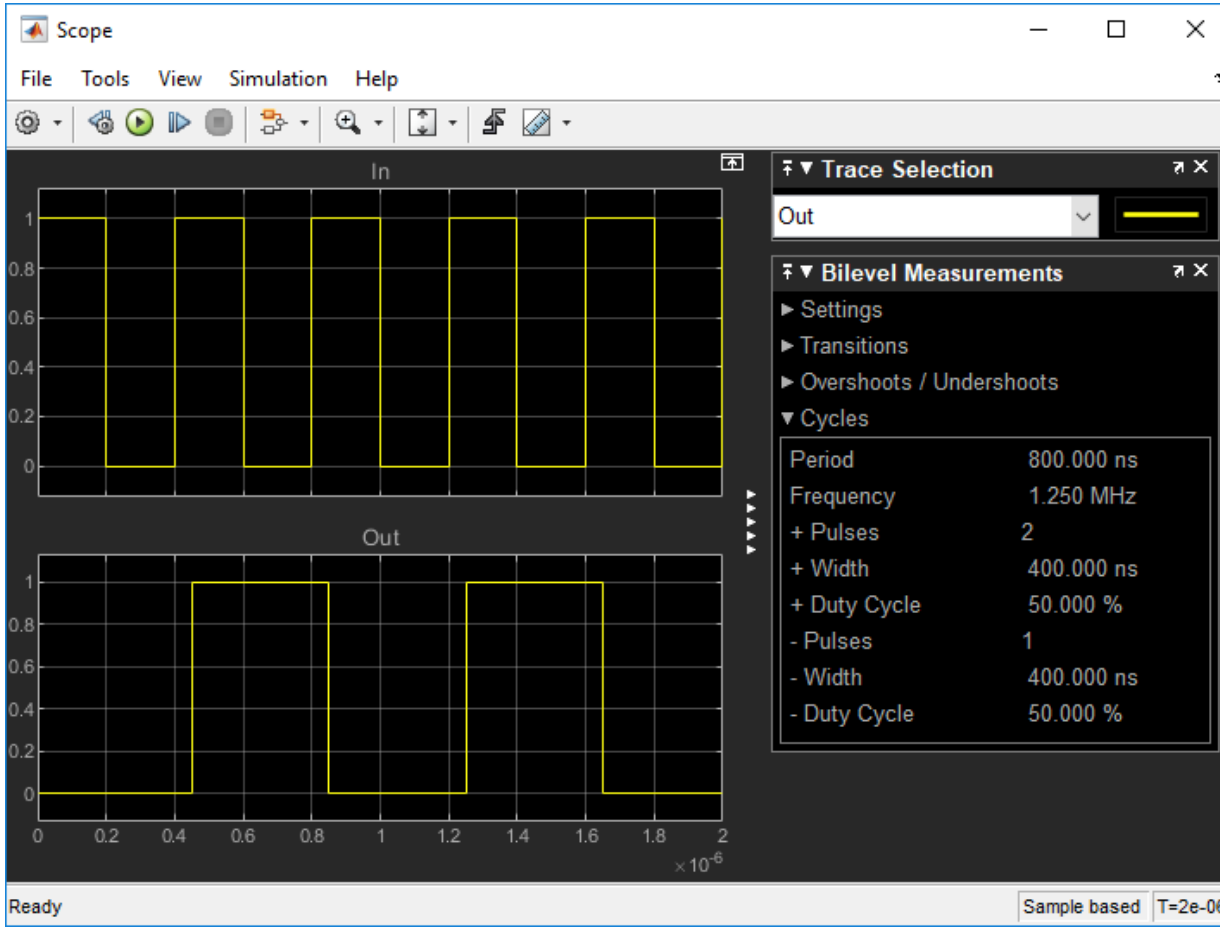
Open the model `singleModulusPrescaler`. The model consists of a Pulse Generator and a Single Modulus Prescaler block.

```
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 `2.5`.

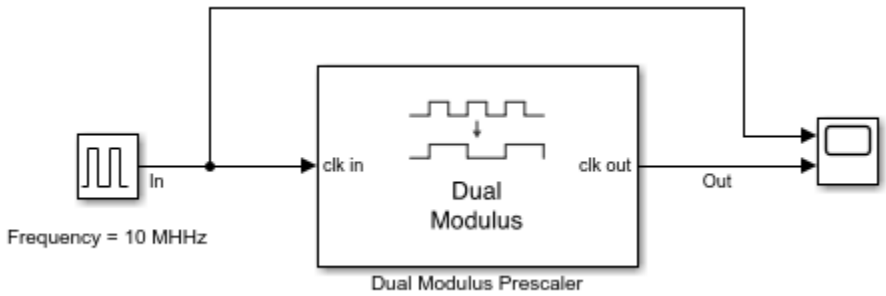
Run the simulation for `2e-6` s. The frequency of the output signal is `1.25` MHz.


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

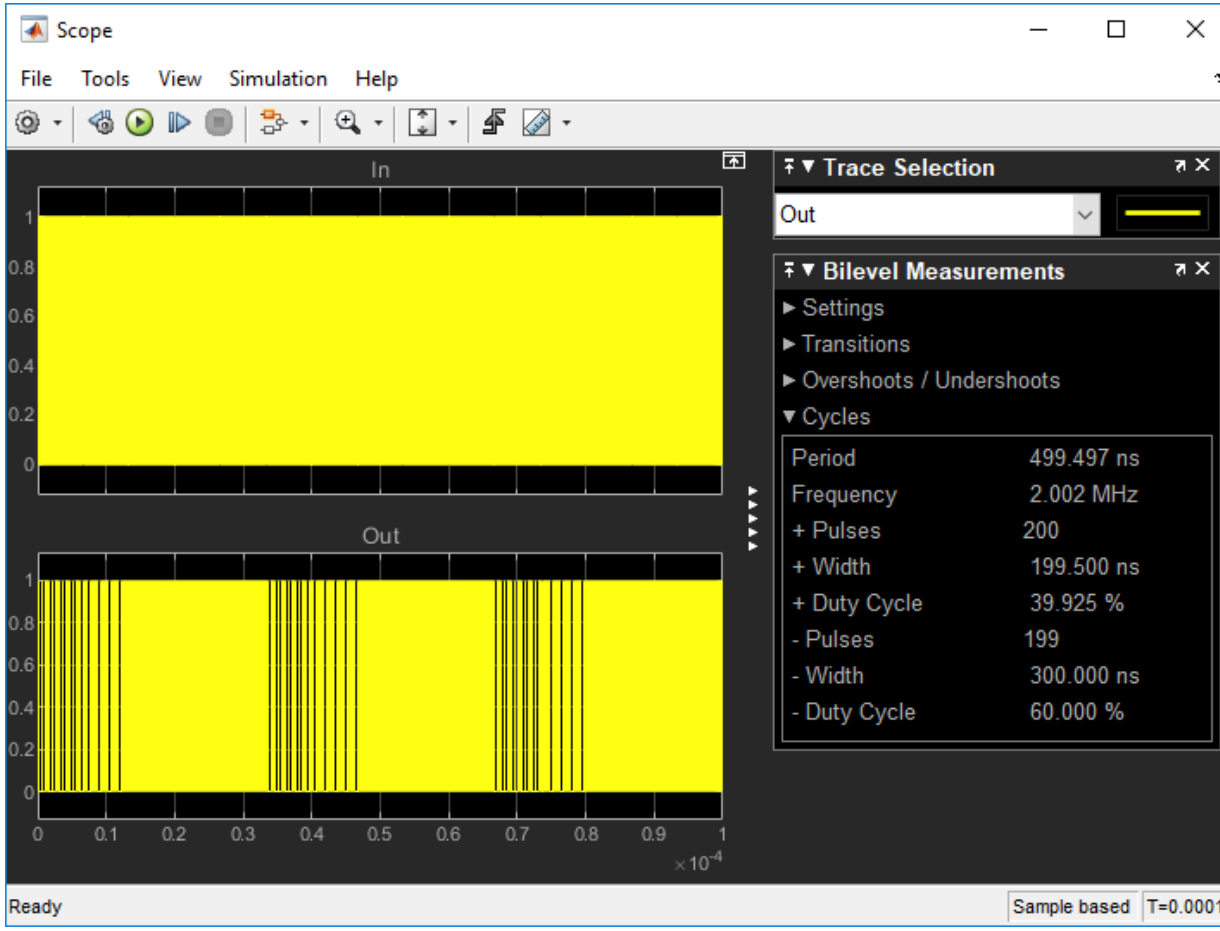
Open the model `Dual_Modulus_Prescaler_Ex`. The model consists of a Pulse Generator and a Dual Modulus Prescaler block.

```
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`.

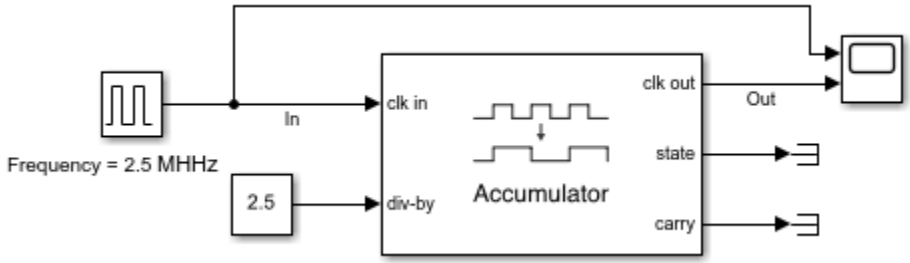
Run the simulation for `1e-4` s. The frequency of the output signal is `2.002` MHz.


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

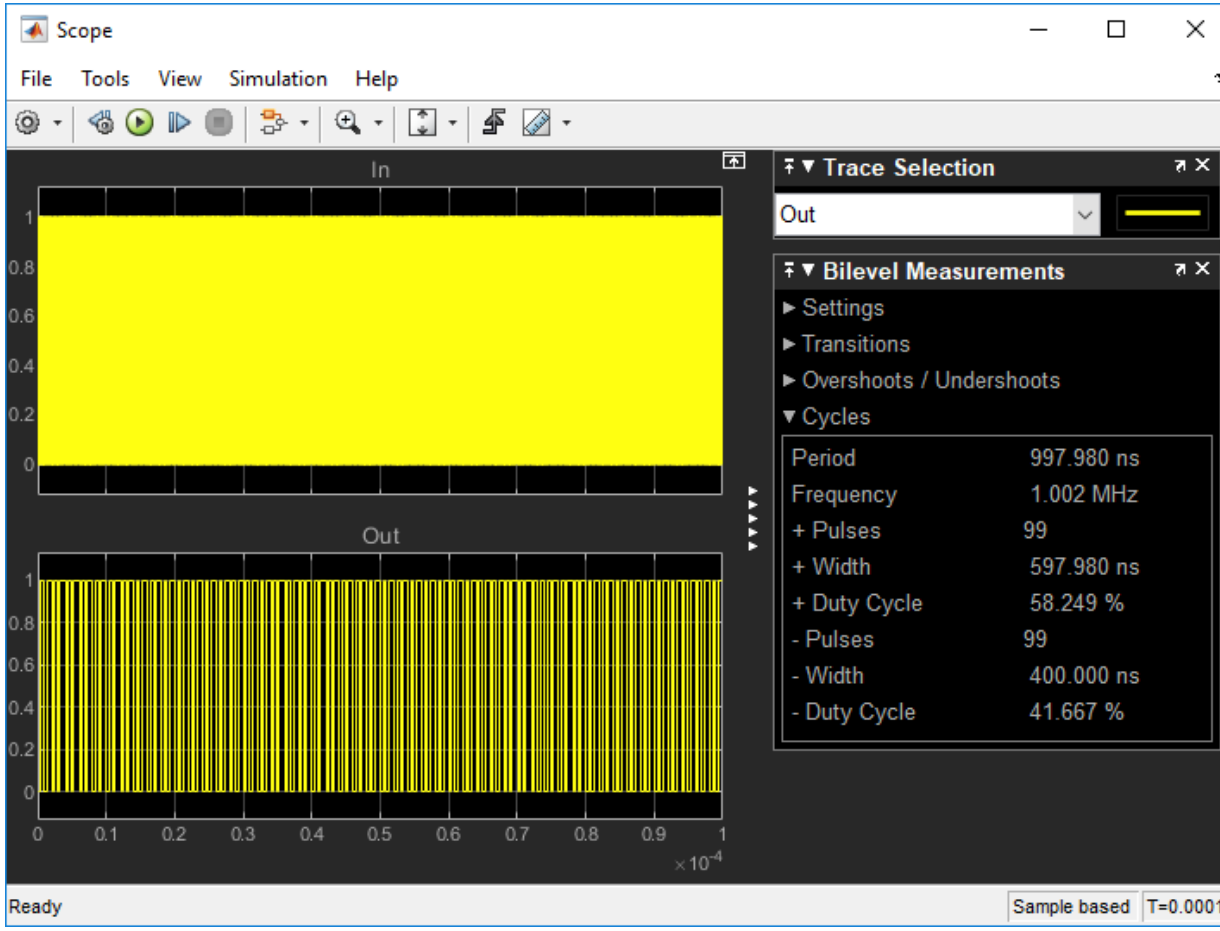
Open the model `fractionalClockDivider_w_Accumulator`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

```
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`.

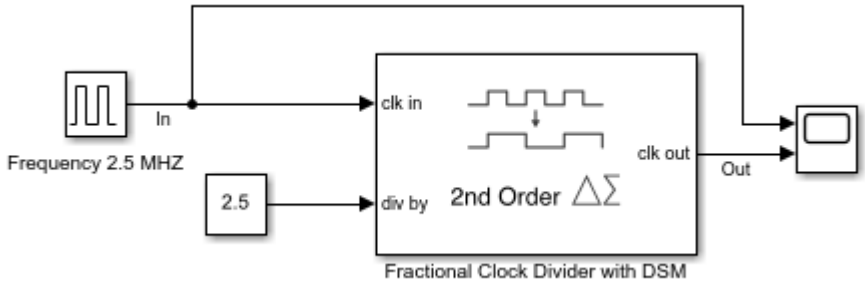
Run the simulation for `1e-4` s. The frequency of the output signal is `1.002` MHz.


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

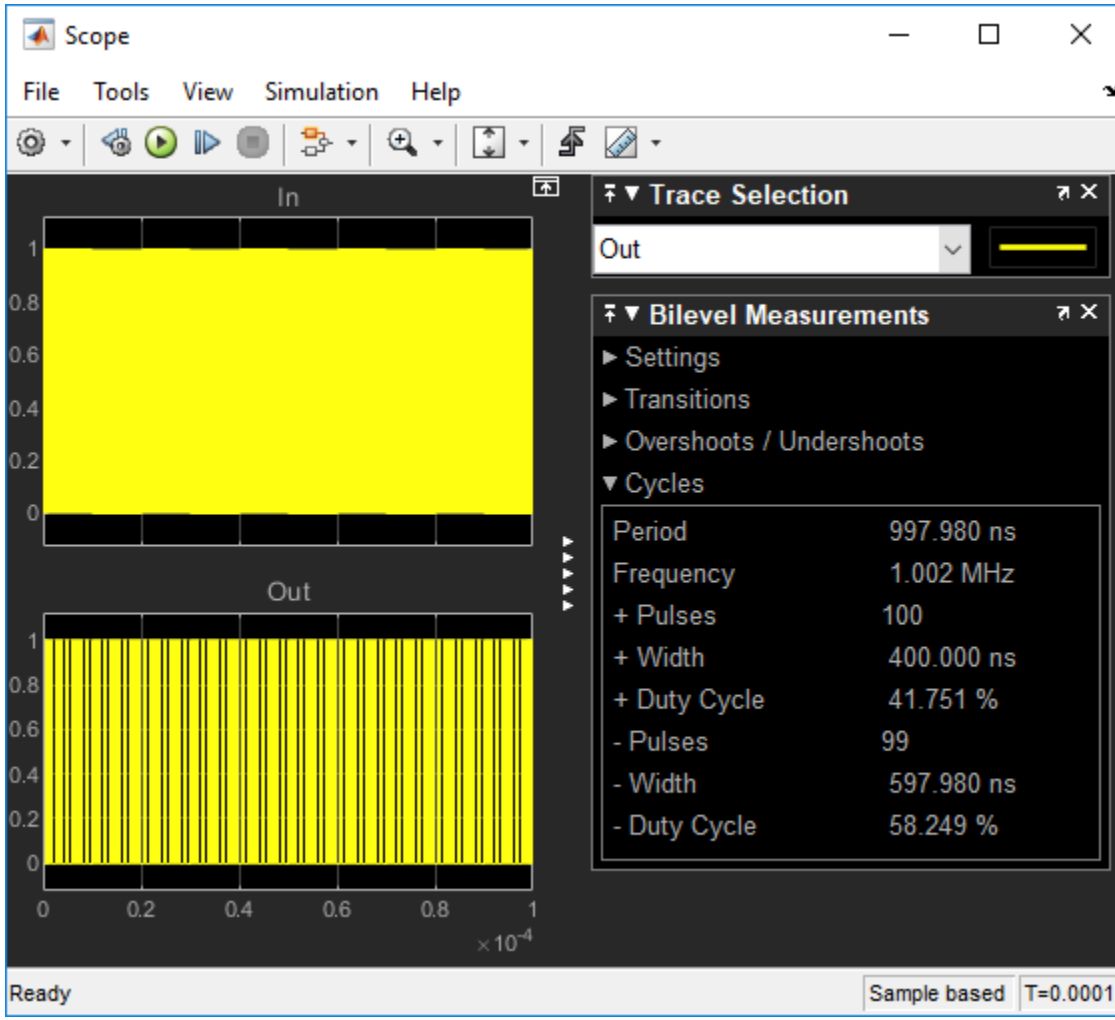
Open the model `fractionalClockDivider_w_DSM`. The model consists of a Pulse Generator and a Fractional Clock Divider with Accumulator block.

```
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