# Audio System Toolbox™

User's Guide

MATLAB® & SIMULINK®

R2017a

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.
3 Apple Hill Drive
Natick, MA 01760-2098

*Audio System Toolbox™ User's Guide*

### Revision History

| | | |
|---|---|---|
| March 2016 | Online only | New for Version 1.0 (Release 2016a) |
| September 2016 | Online only | Revised for Version 1.1 (Release 2016b) |
| March 2017 | Online only | Revised for Version 1.2 (Release 2017a) |

# Contents

# 4 Use the Audio Test Bench

# 5 Audio Plugin Example Gallery

# 6 Equalization

# 13 Tips and Tricks for Plugin Authoring

1

# Dynamic Range Control

# Dynamic Range Control

*Dynamic range control* is the adaptive adjustment of the dynamic range of a signal. The dynamic range of a signal is the logarithmic ratio of maximum to minimum signal amplitude specified in dB.

You can use dynamic range control to:

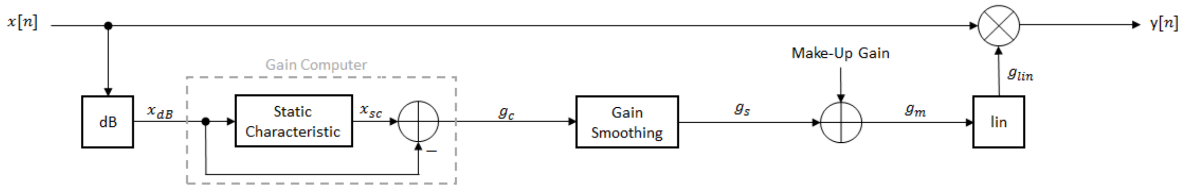
- Match an audio signal level to its environment
- Protect AD converters from overload
- Optimize information
- Suppress low-level noise

Types of dynamic range control include:

- Dynamic range compressor — Attenuates the volume of loud sounds that cross a given threshold. They are often used in recording systems to protect hardware and to increase overall loudness.
- Dynamic range limiter — A type of compressor that brickwalls sound above a given threshold.
- Dynamic range expander — Attenuates the volume of quiet sounds below a given threshold. They are often used to make quiet sounds even quieter.
- Noise gate — A type of expander that brickwalls sound below a given threshold.

This tutorial shows how to implement dynamic range control systems using the compressor, expander, limiter, and noiseGate System objects from Audio System Toolbox. The tutorial also provides an illustrated example of dynamic range limiting at various stages of a dynamic range limiting system.

The diagram depicts a general dynamic range control system.



In a dynamic range control system, a gain signal is calculated in a sidechain and then applied to the input audio signal. The sidechain consists of:

- Linear to dB conversion: $x \rightarrow x_{dB}$
- Gain computation, by passing the dB signal through a static characteristic equation, and then taking the difference: $g_c = x_{sc} - x_{dB}$
- Gain smoothing over time: $g_c \rightarrow g_s$
- Addition of make-up gain (for compressors and limiters only): $g_s \rightarrow g_m$
- dB to linear conversion: $g_m \rightarrow g_{lin}$
- Application of the calculated gain signal to the original audio signal: $y = g_{lin} \times x$

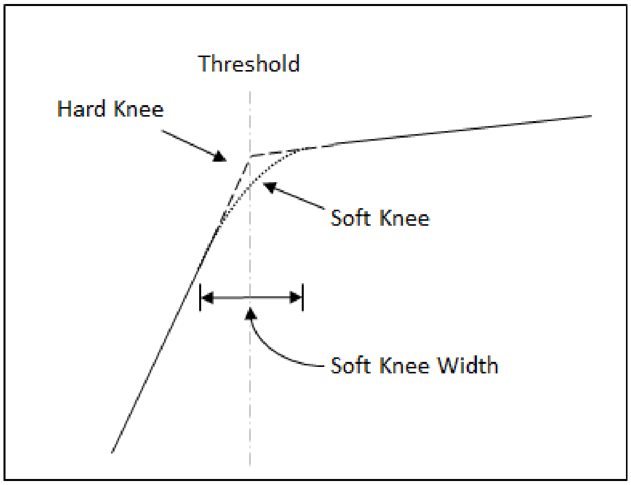
## Linear to dB Conversion

The gain signal used in dynamic range control is processed on a dB scale for all dynamic range controllers. There is no reference for the dB output; it is a straight conversion: $x_{dB} = 20\log_{10}(x)$ . You might need to adjust the output of a dynamic range control system to the range of your system.

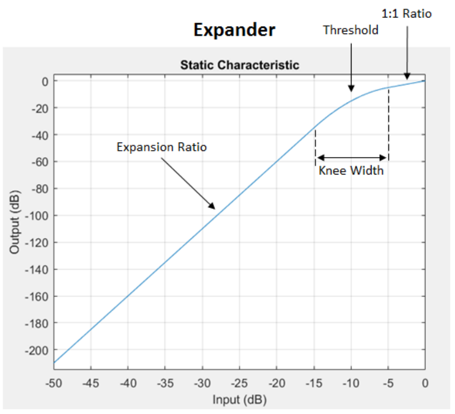
## Gain Computer

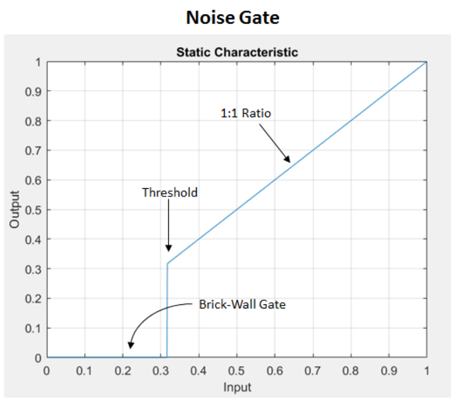
The gain computer provides the first rough estimate of a gain signal for dynamic range control. The principal component of the gain computer is the static characteristic. Each type of dynamic range control has a different static characteristic with different tunable properties:

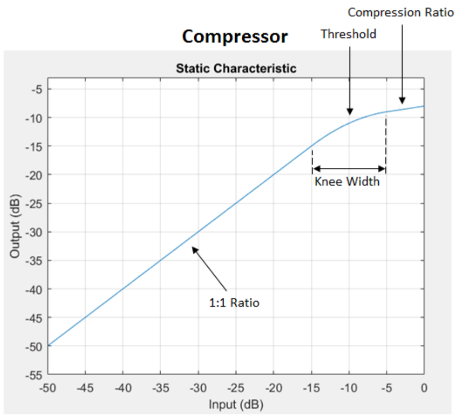
- `Threshold` — All static characteristics have a threshold. On one side of the threshold, the input is given to the output with no modification. On the other side of the threshold, compression, expansion, brickwall limiting, or brickwall gating is applied.
- `Ratio` — Expanders and compressors enable you to adjust the input-to-output ratio of the static characteristic above or below a given threshold.
- `KneeWidth` — Expanders, compressors, and limiters enable you to adjust the knee width of the static characteristic. The knee of a static characteristic is centered at the threshold. An increase in knee width creates a smoother transition around the threshold. A knee width of zero provides no smoothing and is known as a *hard knee*. A knee width greater than zero is known as a *soft knee*.

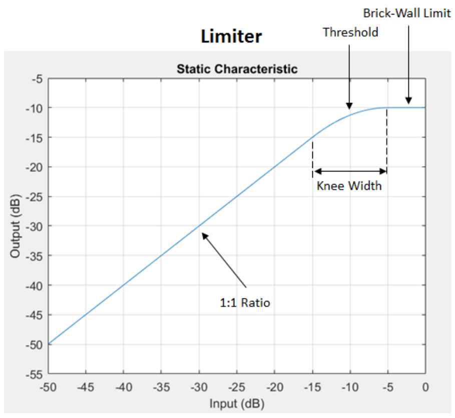


In these static characteristic plots, the expander, limiter, and compressor each have a 10 dB knee width.









## Gain Smoothing

All dynamic range controllers provide gain smoothing over time. Gain smoothing diminishes sharp jumps in the applied gain, which can result in artifacts and an unnatural sound. You can conceptualize gain smoothing as the addition of impedance to your gain signal.

The `expander` and `noiseGate` objects have the same smoothing equation, because a noise gate is a type of expander. The `limiter` and `compressor` objects have the same smoothing equation, because a limiter is a type of compressor.

The type of gain smoothing is specified by a combination of attack time, release time, and hold time coefficients. Attack time and release time correspond to the time it takes the gain signal to go from 10% to 90% of its final value. Hold time is a delay period before gain is applied. See the algorithms of individual dynamic range controller pages for more detailed explanations.

### Smoothing Equations

#### `expander` and `noiseGate`

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & if\ (C_A > k)\ \&\ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & if\ \ C_A \le k \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & if\ (C_R > k)\ \&\ (g_c[n] \le g_s[n-1]) \\ g_s[n-1] & if\ \ C_R \le k \end{cases}$$

- $\alpha_A$ and $\alpha_R$ are determined by the sample rate and specified attack and release time:

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right), \quad \alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right)$$

- $k$ is the specified hold time in samples.
- $C_A$ and $C_R$ are hold counters for attack and release, respectively.

#### `compressor` and `limiter`

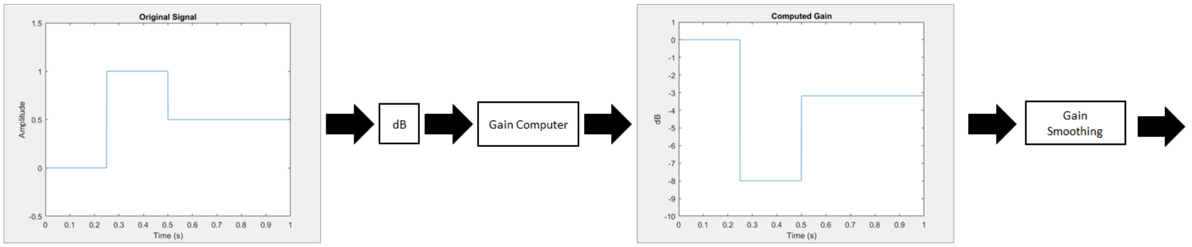
$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & for\ g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & for\ g_c[n] \le g_s[n-1] \end{cases}$$

- $\alpha_A$ and $\alpha_R$ are determined by the sample rate and specified attack and release time:

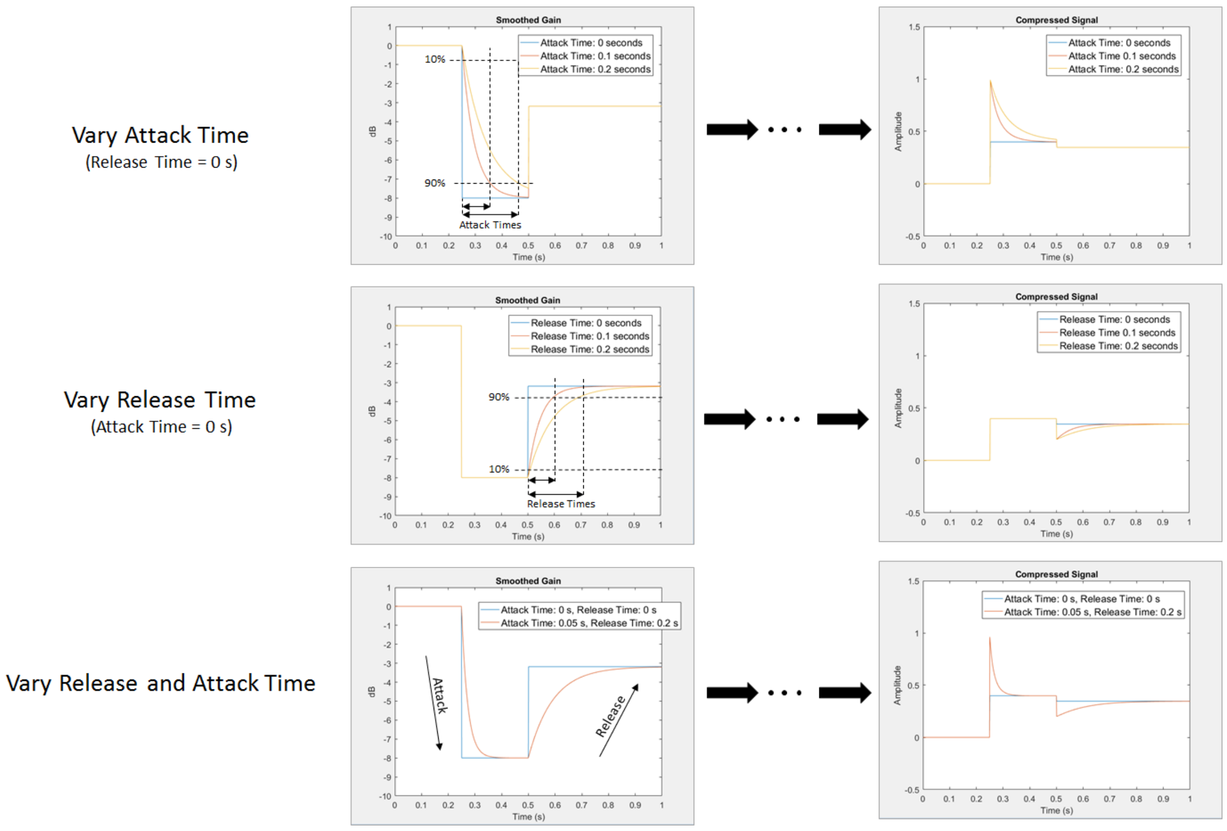
$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right), \quad \alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right)$$

### Gain Smoothing Example

Examine a trivial case of dynamic range compression for a two-step input signal. In this example, the compressor has a threshold of –10 dB, a compression ratio of 5, and a hard knee.



Several variations of gain smoothing are shown. On the top, a smoothed gain curve is shown for different attack time values, with release time set to zero seconds. In the middle, release time is varied and attack time is held constant at zero seconds. On the bottom, both attack and release time are specified by nonzero values.



## Make-Up Gain

Make-up gain applies for compressors and limiters, where higher dB portions of a signal are attenuated or brickwalled. The dB reduction can significantly reduce total signal power. In these cases, make-up gain is applied after gain smoothing to increase the signal power. In Audio System Toolbox, you can specify a set amount of make-up gain or specify the make-up gain mode as `'auto'`.

The `'auto'` make-up gain ensures that a 0 dB input results in a 0 dB output. For example, assume a static characteristic of a compressor with a soft knee:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases}$$

*T* is the threshold, *W* is the knee width, and *R* is the compression ratio. The calculated auto make-up gain is the negative of the static characteristic equation evaluated at 0 dB:

$$-x_{sc}(0) = \begin{cases} 0 & \frac{W}{2} < T \\ \frac{\left(\frac{1}{R} - 1\right)\left(T - \frac{W}{2}\right)^2}{2W} & -\frac{W}{2} \le T \le \frac{W}{2} \\ T + \frac{T}{R} & -\frac{W}{2} > T \end{cases}$$

## dB to Linear Conversion

Once the gain signal is determined in dB, it is transferred to the linear domain:

$g_{lin} = 10^{g_m/20}$.

## Apply Calculated Gain

The final step in a dynamic control system is to apply the calculated gain by multiplication in the linear domain.

## Example: Dynamic Range Limiter

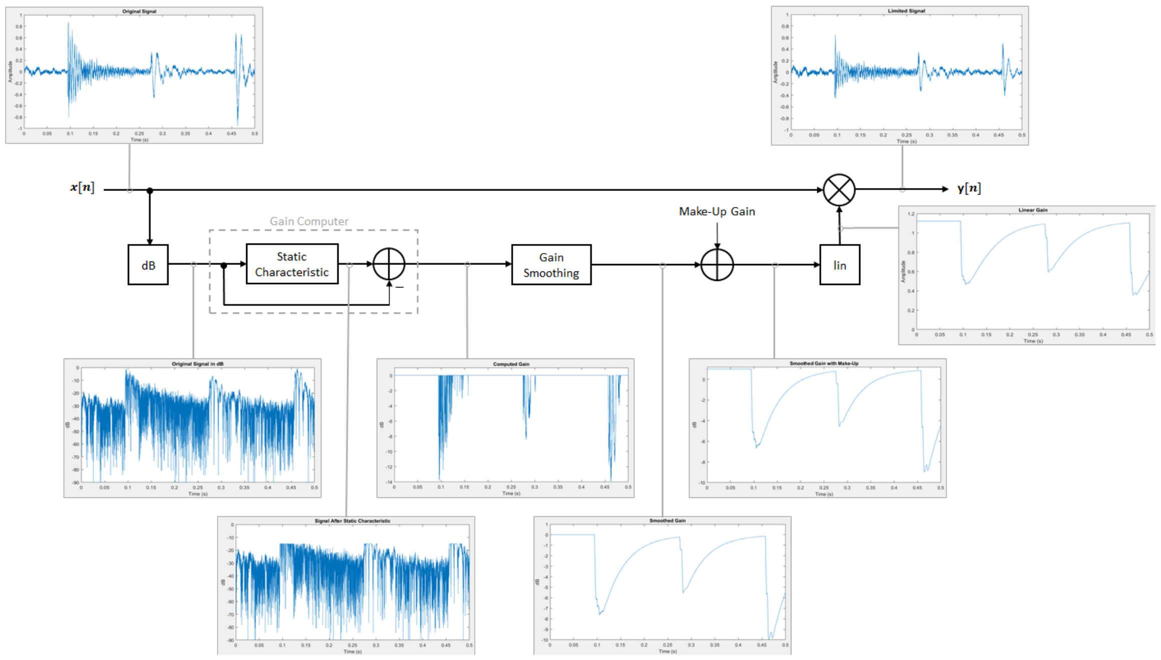
The audio signal described in this example is a 0.5 second interval of a drum track. The limiter properties are:

- Threshold = –15 dB
- Knee width = 0 (hard knee)
- Attack time = 0.004 seconds
- Release time = 0.1 seconds
- Make-up gain = 1 dB

To create a `limiter` System object™ with these properties, at the MATLAB® command prompt, enter:

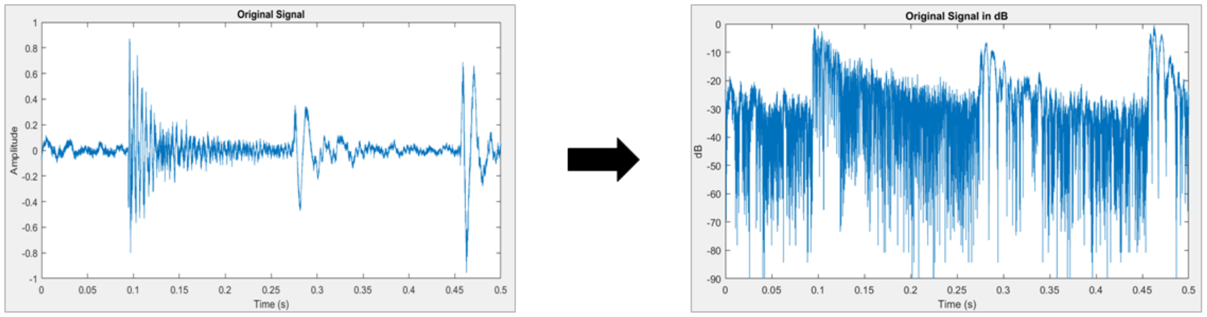
```
dRL = limiter('Threshold',-15,...
              'KneeWidth',0,...
              'AttackTime',0.004,...
              'ReleaseTime',0.1,...
              'MakeUpGainMode','property',...
              'MakeUpGain',1);
```

This example provides a visual walkthrough of the various stages of the dynamic range limiter system.



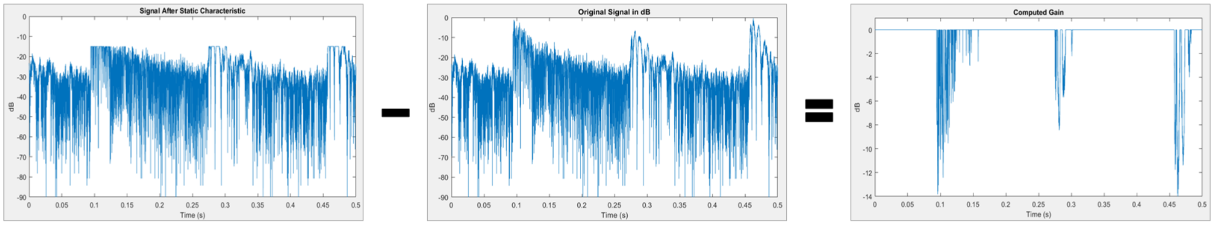
## Linear to dB Conversion

The input signal is converted to a dB scale element by element.



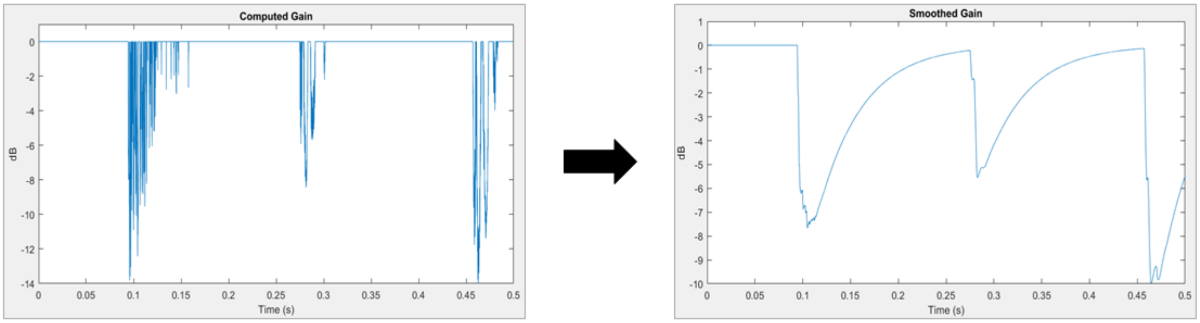
### Gain Computer

The static characteristic brickwall limits the dB signal at –15 dB. To determine the dB gain that results in this limiting, the gain computer subtracts the original dB signal from the dB signal processed by the static characteristic.



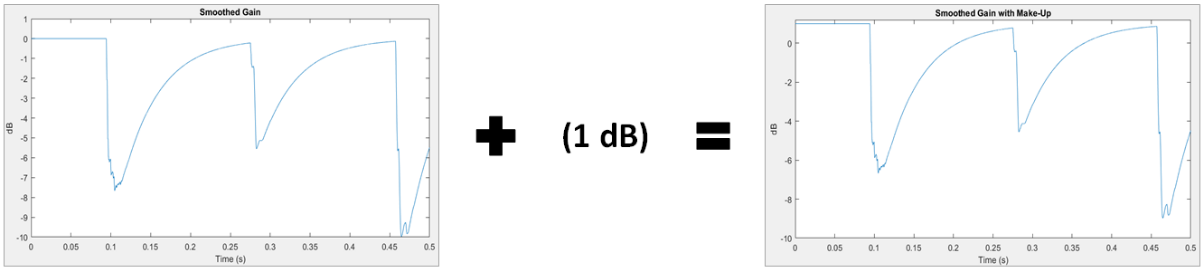
### Gain Smoothing

The relatively short attack time specification results in a steep curve when the applied gain is suddenly increased. The relatively long release time results in a gradual diminishing of the applied gain.



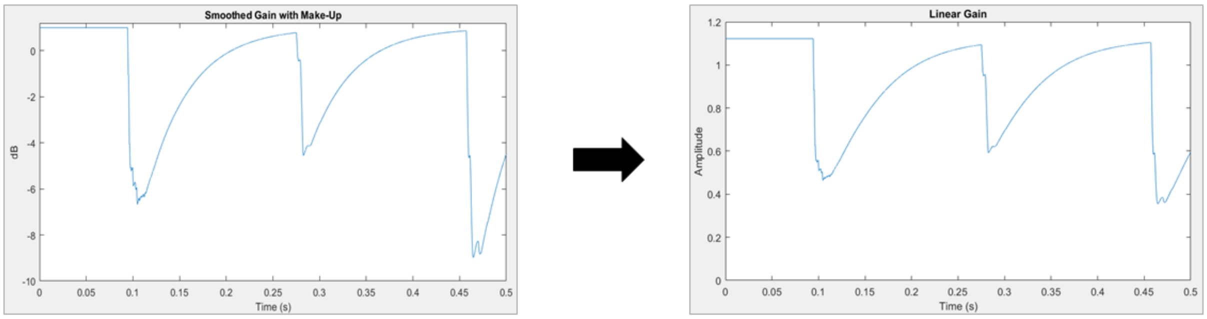
### Make-Up Gain

Assume a limiter with a 1 dB make-up gain value. The make-up gain is added to the smoothed gain signal.



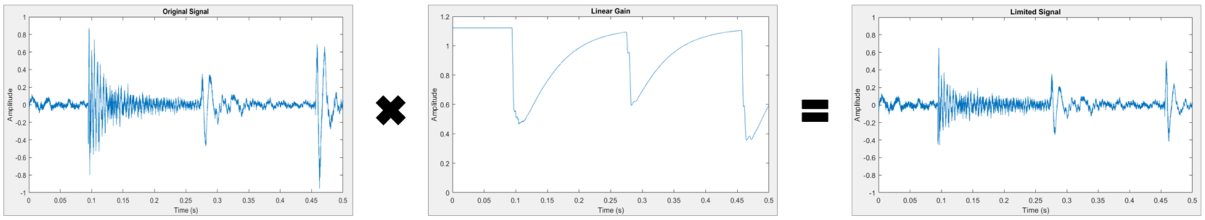
## dB to Linear Conversion

The gain in dB is converted to a linear scale element by element.



## Apply Calculated Gain

The original signal is multiplied by the linear gain.



## References

[1] Zolzer, Udo. "Dynamic Range Control." *Digital Audio Signal Processing*. 2nd ed. Chichester, UK: Wiley, 2008.

[2] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. “Digital Dynamic Range Compressor Design—A Tutorial And Analysis.” *Journal of Audio Engineering Society*. Vol. 60, Issue 6, pp. 399–408.

## See Also

**Blocks**
Compressor | Expander | Limiter | Noise Gate

**System Objects**
compressor | expander | limiter | noiseGate

## More About

- “Multiband Dynamic Range Compression”

2

# MIDI Control for Audio Plugins

# MIDI Control for Audio Plugins

## MIDI and Plugins

MIDI control surfaces are commonly used in conjunction with audio plugins in digital audio workstation (DAW) environments. Synchronizing MIDI controls with plugin parameters provides a tangible interface for audio processing and is an efficient approach to parameter tuning.

In the MATLAB environment, audio plugins are defined as any valid class that derives from the `audioPlugin` base class or the `audioPluginSource` base class. For more information about how audio plugins are defined in the MATLAB environment, see "Design an Audio Plugin".

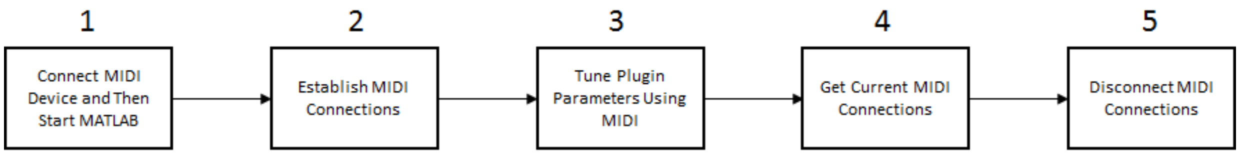
## Use MIDI with MATLAB Plugins

The Audio System Toolbox product provides three functions for enabling the interface between MIDI control surfaces and audio plugins:

- `configureMIDI` — Configure MIDI connections between audio plugin and MIDI controller.
- `getMIDIConnections` — Get MIDI connections of audio plugin.
- `disconnectMIDI` — Disconnect MIDI controls from audio plugin.

These functions combine the abilities of general MIDI functions into a streamlined and user-friendly interface suited to audio plugins in MATLAB. For a tutorial on the general functions and the MIDI protocol, see "Musical Instrument Digital Interface (MIDI)" on page 3-2.

This tutorial walks you through the MIDI functions for audio plugins in MATLAB.


1
Connect MIDI Device and Then Start MATLAB
2
Establish MIDI Connections
3
Tune Plugin Parameters Using MIDI
4
Get Current MIDI Connections
5
Disconnect MIDI Connections

## 1. Connect MIDI Device and Then Start MATLAB

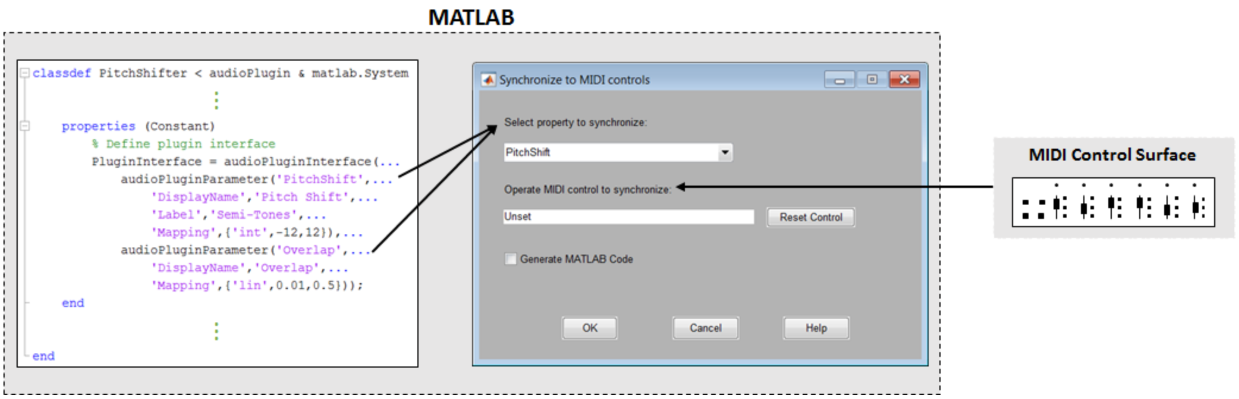
Before starting MATLAB, connect your MIDI control surface to your computer and turn it on. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart MATLAB with the device already connected.

## 2. Establish MIDI Connections

Use `configureMIDI` to establish MIDI connections between your default MIDI device and an audio plugin. You can use `configureMIDI` programmatically, or you can open a user interface (UI) to guide you through the process. The `configureMIDI` UI reads from your audio plugin and populates a drop-down list of tunable plugin properties. You are then prompted to move individual controls on your MIDI control surface to associate the position of each control with the normalized value of each property you select. For example, create an object of `audiopluginexample.PitchShifter` and then call `configureMIDI` with the object as the argument:

```
ctrlPitch = audiopluginexample.PitchShifter;
configureMIDI(ctrlPitch)
```

The Synchronize to MIDI controls dialog box opens with the tunable properties of your plugin automatically populated. When you operate a MIDI control, its identification is entered into the **Operate MIDI control to synchronize** box. After you synchronize tunable properties with MIDI controls, click **OK** to complete the configuration. If your MIDI control surface is bidirectional, it automatically shifts the position of the synchronized controls to the initial property values specified by your plugin.


MATLAB
Synchronize to MIDI controls
Select property to synchronize:
PitchShift
Operate MIDI control to synchronize:
Unset
Reset Control
Generate MATLAB Code
OK
Cancel
Help
MIDI Control Surface

```
classdef PitchShifter < audioPlugin & matlab.System
                          ⋮
    properties (Constant)
        % Define plugin interface
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('PitchShift',...
                'DisplayName','Pitch Shift',...
                'Label','Semi-Tones',...
                'Mapping',{'int',-12,12}),...
            audioPluginParameter('Overlap',...
                'DisplayName','Overlap',...
                'Mapping',{'lin',0.01,0.5}));
    end
                          ⋮
end
```

To open a MATLAB function with the programmatic equivalent of your actions in the UI, select the **Generate MATLAB Code** check box. Saving this function enables you to reuse your settings and quickly establish the configuration in future sessions.

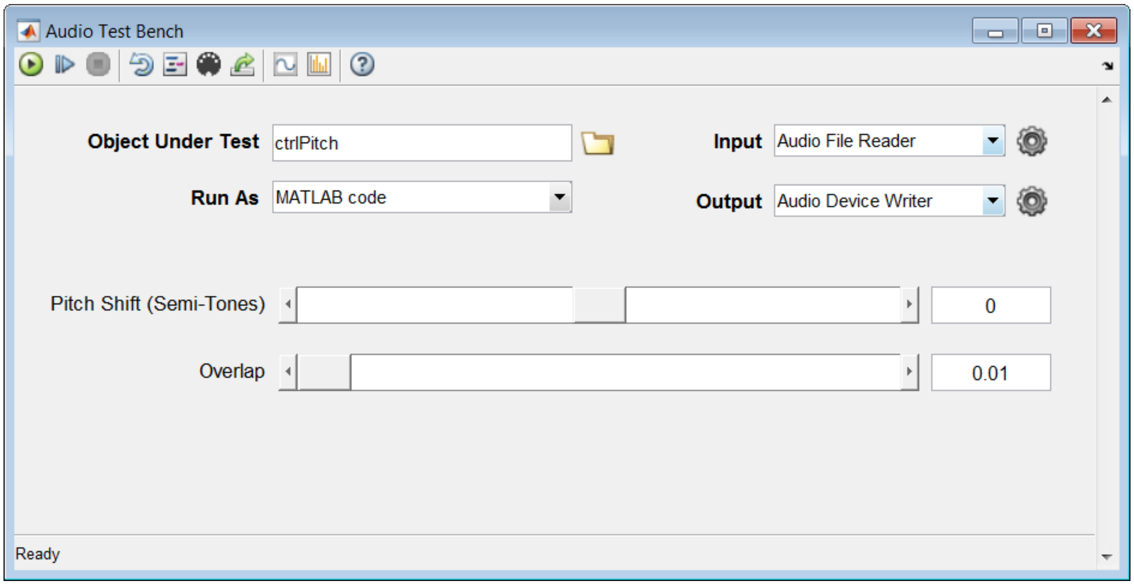
### 3. Tune Plugin Parameters Using MIDI

After you establish connections between plugin properties and MIDI controls, you can tune the properties in real time using your MIDI control surface.

Audio System Toolbox provides an all-in-one app for running and testing your audio plugin. The test bench mimics how a DAW interacts with plugins.

Open the **Audio Test Bench** for your `ctrlPitch` object.

```
audioTestBench(ctrlPitch)
```



Audio Test Bench

Object Under Test: ctrlPitch

Run As: MATLAB code

Input: Audio File Reader

Output: Audio Device Writer

Pitch Shift (Semi-Tones): 0

Overlap: 0.01

Ready

When you adjust the controls on your MIDI surface, the corresponding plugin parameter sliders move. Click ▶ to run the plugin. Move the controls on your MIDI surface to hear the effect of tuning the plugin parameters.

To establish MIDI connections and modify existing ones, click the Synchronize to MIDI Controls ● button to open a `configureMIDI` UI.

Alternatively, you can use the MIDI connections you established in a script or function. For example, run the following code and move your synchronized MIDI controls to hear the pitch-shifting effect:

```
fileReader = dsp.AudioFileReader(...
    'Filename','Counting-16-44p1-mono-15secs.wav');
deviceWriter = audioDeviceWriter;

% Audio stream loop
while ~isDone(fileReader)
    input = fileReader();
    output = ctrlPitch(input);
    deviceWriter(output);
    drawnow limitrate; % Process callback immediately
end

release(fileReader);
release(deviceWriter);
```

#### 4. Get Current MIDI Connections

To query the MIDI connections established with your audio plugin, use the `getMIDIConnections` function. `getMIDIConnections` returns a structure with fields corresponding to the tunable properties of your plugin. The corresponding values are nested structures containing information about the mapping between your plugin property and the specified MIDI control.

```
connectionInfo = getMIDIConnections(ctrlPitch)
```

```
connectionInfo = 

  struct with fields:

    PitchShift: [1×1 struct]
       Overlap: [1×1 struct]
```

```
connectionInfo.PitchShift
```

```
ans = 

  struct with fields:

            Law: 'int'
            Min: -12
            Max: 12
```

```
    MIDIControl: 'control 1081 on 'BCF2000''
```

### 5. Disconnect MIDI Surface

As a best practice, release external devices such as MIDI control surfaces when you are done.

```
disconnectMIDI(ctrlPitch)
```

## See Also

**Apps**
Audio Test Bench

**Classes**
audioPlugin | audioPluginSource

**Functions**
`configureMIDI` | `disconnectMIDI` | `getMIDIConnections`

## More About

- "What Are DAWs, Audio Plugins, and MIDI Controllers?"
- "Musical Instrument Digital Interface (MIDI)" on page 3-2
- "Design an Audio Plugin"
- "Host External Audio Plugins"

## External Websites

- http://www.midi.org

# Musical Instrument Digital Interface

# Musical Instrument Digital Interface (MIDI)

**In this section...**

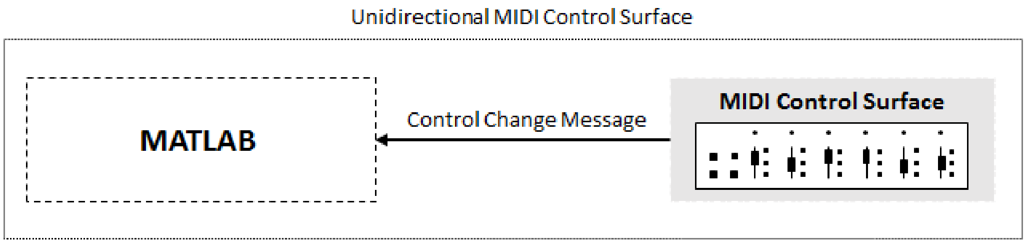
## About MIDI

Musical Instrument Digital Interface (MIDI) was originally developed to interconnect electronic musical instruments. This interface is flexible and has uses in applications far beyond musical instruments. Its simple unidirectional messaging protocol supports many different kinds of messaging. One kind of MIDI message is the *Control Change message*, which is used to communicate changes in controls, such as knobs, sliders, and buttons.

## MIDI Control Surfaces

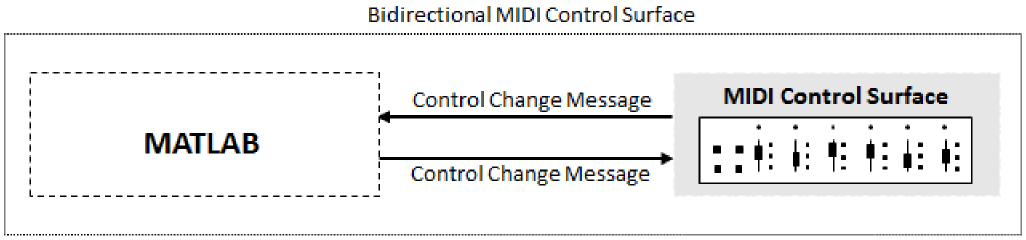
A *MIDI control surface* is a device with controls that sends MIDI Control Change messages when you turn a knob, move a slider, or push a button on its surface. Each Control Change message indicates which control changed and what its new position is.

Because the MIDI messaging protocol is unidirectional, determining a particular controller position requires that the receiver listen for Control Change messages that the controller sends. The protocol does not support querying the MIDI controller for its position.



The simplest MIDI control surfaces are unidirectional: They send MIDI Control Change messages but do not receive them. More sophisticated control surfaces are bidirectional:

They can both send and receive Control Change messages. These control surfaces have knobs or sliders that can operate automatically. For example, a control surface can have motorized sliders or knobs. When it receives a Control Change message, the appropriate control moves to the position in the message.

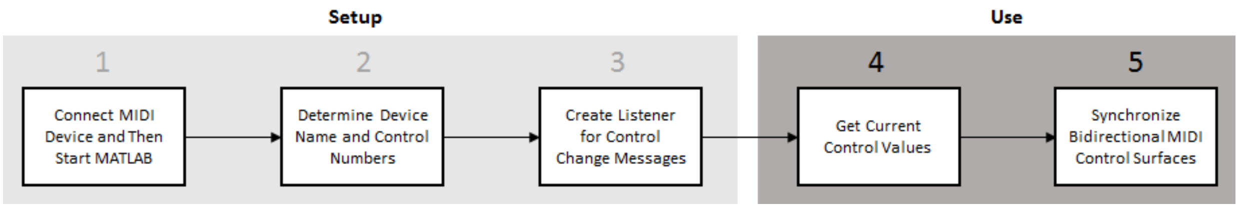


## Use MIDI Control Surfaces with MATLAB and Simulink

The Audio System Toolbox product enables you to use MIDI control surfaces to control MATLAB programs and Simulink® models by providing the capability to listen to Control Change messages. The toolbox also provides a limited capability to send Control Change messages to support synchronizing MIDI controls. This tutorial covers general MIDI functions. For functions specific to audio plugins in MATLAB, see "MIDI Control for Audio Plugins" on page 2-2. The Audio System Toolbox general interface to MIDI control surfaces includes five functions and one block:

- `midiid` — Interactively identify MIDI control.
- `midicontrols` — Open group of MIDI controls for reading.
- `midiread` — Return most recent value of MIDI controls.
- `midisync` — Send values to MIDI controls for synchronization.
- `midicallback` — Call function handle when MIDI controls change value.
- MIDI Controls (block) — Output values from controls on MIDI control surface. The MIDI Controls block combines functionality of the general MIDI functions into one block for the Simulink environment.

This diagram shows a typical workflow involving general MIDI functions in MATLAB. For the Simulink environment, follow steps 1 and 2, and then use the MIDI Controls block for a user-interface guided workflow.


Setup
Use
1
Connect MIDI Device and Then Start MATLAB
2
Determine Device Name and Control Numbers
3
Create Listener for Control Change Messages
4
Get Current Control Values
5
Synchronize Bidirectional MIDI Control Surfaces

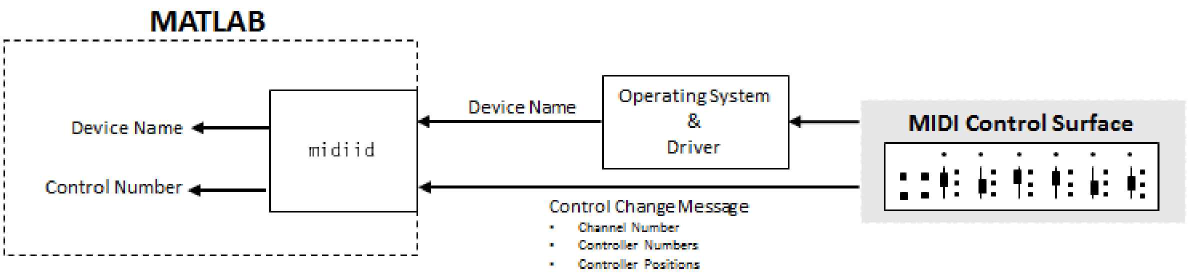
### 1. Connect MIDI Device and Then Start MATLAB

Before starting MATLAB, connect your MIDI control surface to your computer and turn it on. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart MATLAB with the device already connected.

### 2. Determine Device Name and Control Numbers

Use the `midiid` function to determine the device name and control numbers of your MIDI control surface. After you call `midiid`, it continues to listen until it receives a Control Change message. When it receives a Control Change message, it returns the control number associated with the MIDI controller number that you manipulated, and optionally returns the device name of your MIDI control surface. The manufacturer and host operating system determine the device name. See “Control Numbers” on page 3-10 for an explanation of how MATLAB calculates the control number.

To set a default device name, see “Set Default MIDI Device” on page 3-10.



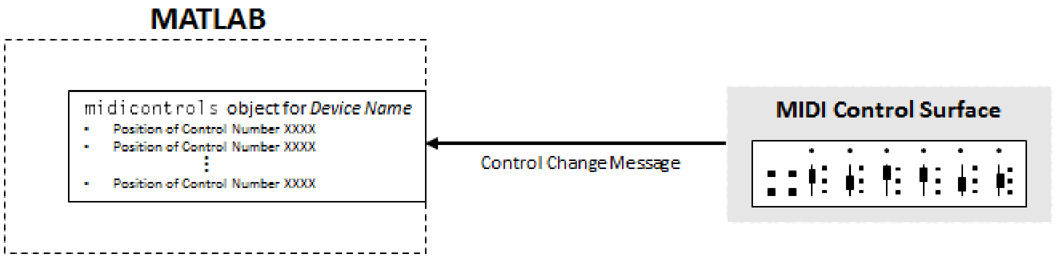
#### View Example

Call `midiid` with two outputs and then move a controller on your MIDI device. `midiid` returns the control number specific to the controller you moved and the device name of the MIDI control surface.

```
[controlNumber,deviceName] = midiid;
```

### 3. Create Listener for Control Change Messages

Use the `midicontrols` function to create an object that listens for Control Change messages and caches the most recent values corresponding to specified controllers. When you create a `midicontrols` object, you specify a MIDI control surface by its device name and specific controllers on the surface by their associated control numbers. Because the `midicontrols` object cannot query the MIDI control surface for initial values, consider setting initial values when creating the object.


MATLAB
midicontrols object for *Device Name*
Position of Control Number XXXX
Position of Control Number XXXX
Position of Control Number XXXX
Control Change Message
MIDI Control Surface

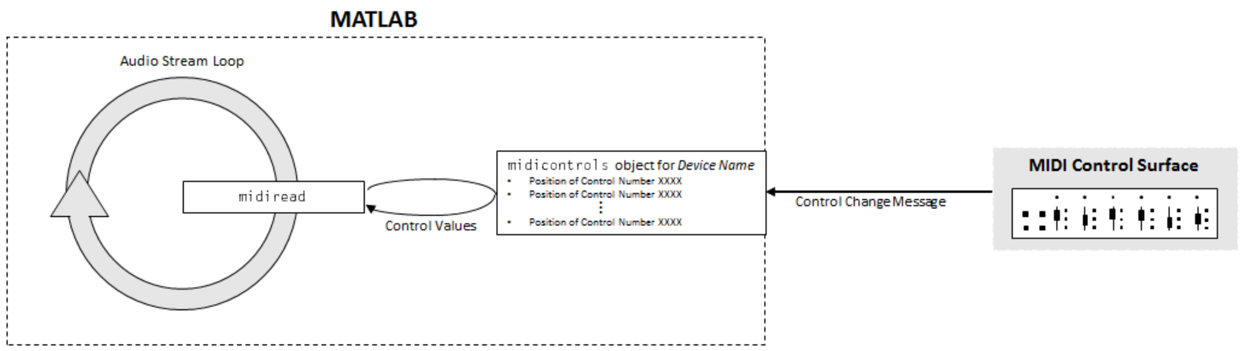
#### View Example

Identify two control numbers on your MIDI control surface. Choose initial control values for the controls you identified. Create a `midicontrols` object that listens to Control Change messages that arrive from the controllers you identified on the device you identified. When you create your `midicontrols` object, also specify initial control values. These initial control values work as default values until a Control Change message is received.

```
controlNum1 = midiid;
[controlNum2,deviceName] = midiid;
initialControlValues = [0.1,0.9];

midicontrolsObject = midicontrols([controlNum1,controlNum2],initialControlValues,'MIDII
```

### 4. Get Current Control Values

Use the `midiread` function to query your `midicontrols` object for current control values. `midiread` returns a matrix with values corresponding to all controllers the `midicontrols` object is listening to. Generally, you want to place `midiread` in an audio stream loop for continuous updating.


MATLAB
Audio Stream Loop
midiread
Control Values
midicontrols object for Device Name
Position of Control Number XXXX
Position of Control Number XXXX
Position of Control Number XXXX
Control Change Message
MIDI Control Surface

#### View Example

Place `midiread` in an audio stream loop to return the current control value of a specified controller. Use the control value to apply gain to an audio signal.

```
[controlNumber, deviceName] = midiid;
initialControlValue = 1;
midicontrolsObject = midicontrols(controlNumber,initialControlValue,'MIDIDevice',device

% Create a dsp.AudioFileReader System object™ with default settings. Create
% an audioDeviceWriter System object and specify the sample rate.
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

% In an audio stream loop, read an audio signal frame from the file, apply
% gain specified by the control on your MIDI device, and then write the
% frame to your audio output device. By default, the control value returned
% by midiread is normalized.
while ~isDone(fileReader)
    audioData = step(fileReader);

    controlValue = midiread(midicontrolsObject);
    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    play(deviceWriter,audioDataWithGain);
end

% Close the input file and release your output device.
release(fileReader);
release(deviceWriter);
```

### 5. Synchronize Bidirectional MIDI Control Surfaces

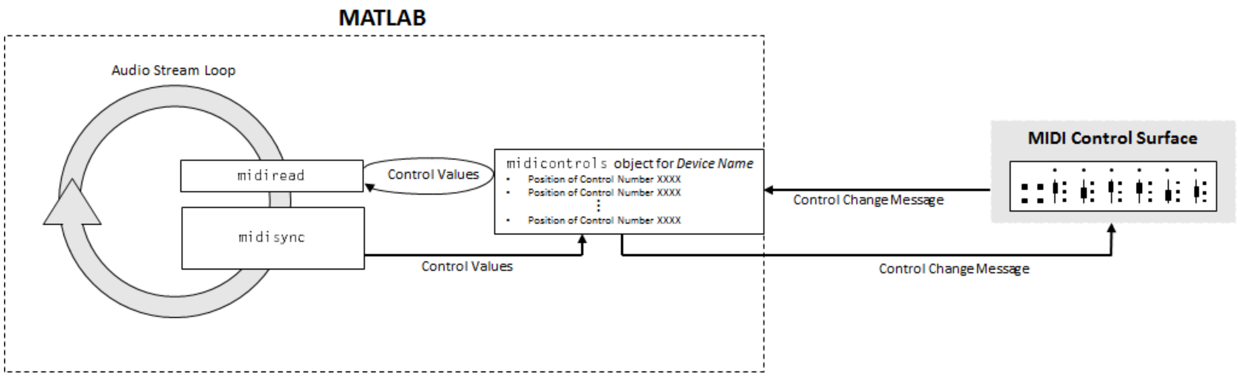
You can use `midisync` to send Control Change messages to your MIDI control surface. If the MIDI control surface is bidirectional, it adjusts the specified controllers. One important use of `midisync` is to set the controller positions on your MIDI control surface to initial values.

#### View Example

In this example, you initialize a controller on your MIDI control surface to a specified position. Calling `midisync(midicontrolsObject)` sends a Control Change message to your MIDI control surface, using the initial control values specified when you created the `midicontrols` object.

```
[controlNumber,deviceName] = midiid;
initialControlValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialControlValue,'MIDIDevice',device

midisync(midicontrolsObject);
```

Another important use of `midisync` is to update your MIDI control surface if control values are adjusted in an audio stream loop. In this case, you call `midisync` with both your `midicontrols` object and the updated control values.



### View Example

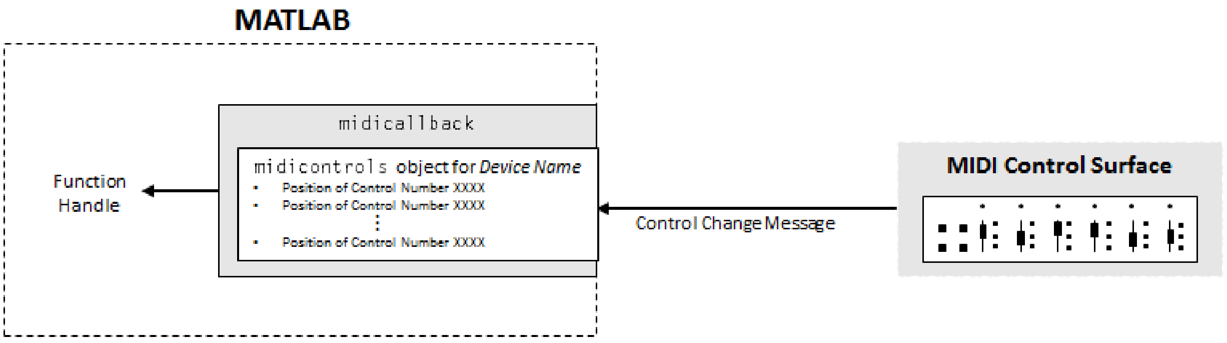
In this example, you check the normalized output volume in an audio stream loop. If the volume is above a given threshold, `midisync` is called and the MIDI controller that controls the applied gain is reduced.

```
[controlNumber, deviceName] = midiid;
initialControlValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialControlValue);
fileReader = dsp.AudioFileReader('Ambiance-16-44p1-mono-12secs.wav');
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

% Synchronize specified initial value with the MIDI control surface.
midisync(midicontrolsObject);

while ~isDone(fileReader)
    audioData = step(fileReader);
    controlValue = midiread(midicontrolsObject);
    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    % Check if max output is above a given threshold.
    if max(audioDataWithGain) > 0.7

        % Force new control value to be nonnegative.
        newControlValue = max(0,controlValue-0.5);

        % Send a Control Change message to the MIDI control surface.
        midisync(midicontrolsObject,newControlValue)
        end

        play(deviceWriter,audioDataWithGain);
    end

    release(fileReader);
    release(deviceWriter);
```

`midisync` is also a powerful tool in systems that also involve user interfaces (UIs), so that when one control is changed, the other control tracks it. Typically, you implement such tracking by setting callback functions on both the `midicontrols` object (using `midicallback`) and the UI control. The callback for the `midicontrols` object sends new values to the UI control. The UI uses `midisync` to send new values to the `midicontrols` object and MIDI control surface. See `midisync` for examples.

### Alternative to Stream Processing

You can use `midicallback` as an alternative to placing `midiread` in an audio stream loop. If a `midicontrols` object receives a Control Change message, `midicallback` automatically calls a specified function handle. The callback function typically calls `midiread` to determine the new value of the MIDI controls. You can use this callback when you want a MIDI controller to trigger an action, such as updating a UI. Using this approach prevents having a MATLAB program continuously running in the command window.


MATLAB
midicallback
midicontrols object for Device Name
Position of Control Number XXXX
Position of Control Number XXXX
Position of Control Number XXXX
Function Handle
Control Change Message
MIDI Control Surface

### Set Default MIDI Device

You can set the default MIDI device in the MATLAB environment by using the `setpref` function. Use `midiid` to determine the name of the device, and then use `setpref` to set the preference.

```
[~,deviceName] = midiid

Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

deviceName =

BCF2000

setpref('midi','DefaultDevice',deviceName)
```

This preference persists across MATLAB sessions, so you only have to set it once, unless you want to change devices.

If you do not set this preference, MATLAB and the host operating system choose a device for you. However, such autoselection can cause unpredictable results because many computers have “virtual” (software) MIDI devices installed that you might not be aware of. For predictable behavior, set the preference.

### Control Numbers

MATLAB defines control numbers as (*MIDI channel number*) × 1000 + (*MIDI controller number*).

- *MIDI channel number* is the transmission channel that your device uses to send messages. This value is in the range 1–16.
- *MIDI controller number* is a number assigned to an individual control on your MIDI device. This value is in the range 1–127.

Your MIDI device determines the values of *MIDI channel number* and *MIDI controller number*.

## See Also

**Blocks**
MIDI Controls

**Functions**
`midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

## More About

- “What Are DAWs, Audio Plugins, and MIDI Controllers?”
- “Real-Time Audio in MATLAB”
- “MIDI Control for Audio Plugins” on page 2-2

## External Websites

- http://www.midi.org

# Use the Audio Test Bench

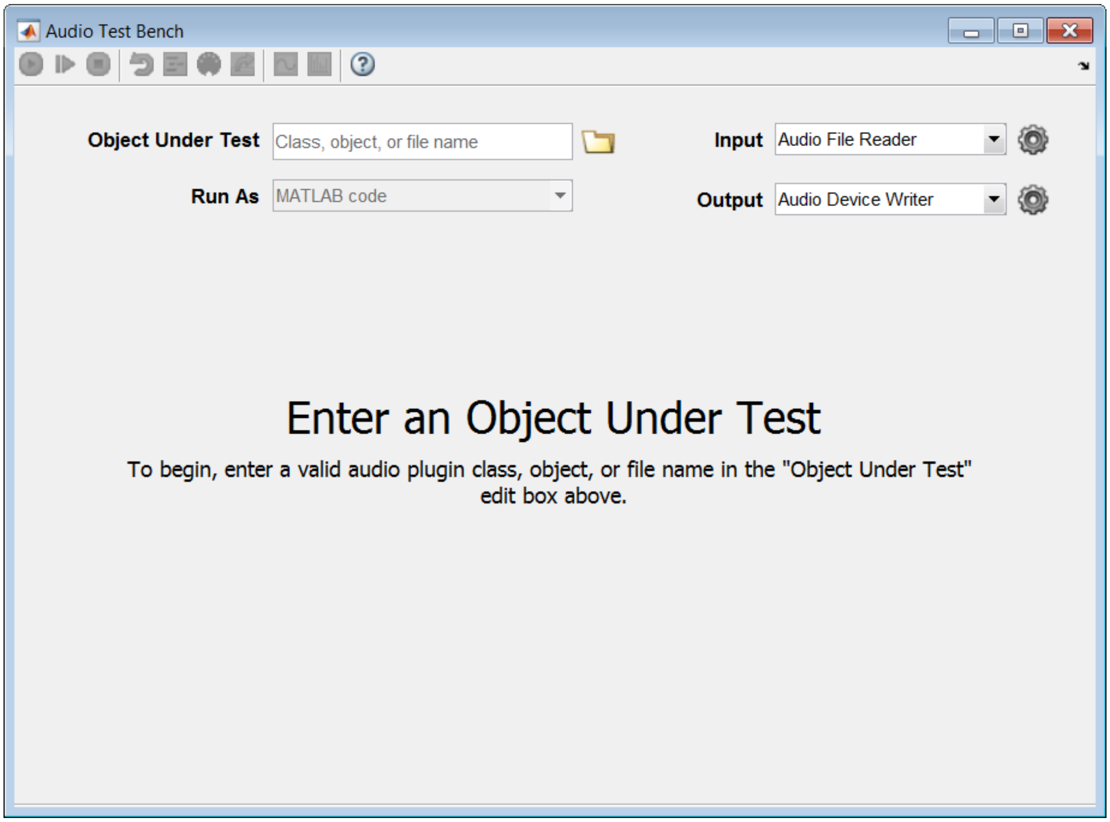
## Audio Test Bench Walkthrough

In this tutorial, explore key functionality of the Audio Test Bench.

### Choose Object Under Test

1 To open the **Audio Test Bench**, at the MATLAB command prompt, enter:

```
audioTestBench
```



Object Under Test: Class, object, or file name

Run As: MATLAB code

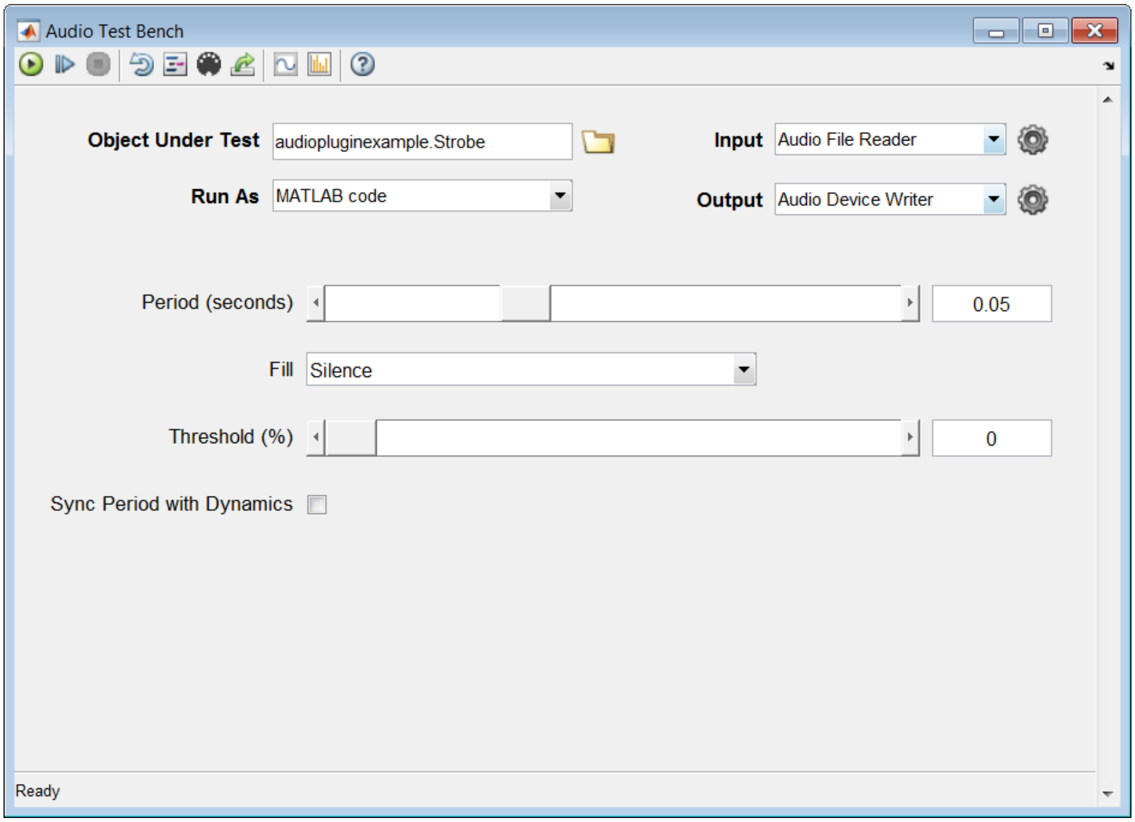
Input: Audio File Reader

Output: Audio Device Writer

Enter an Object Under Test

To begin, enter a valid audio plugin class, object, or file name in the "Object Under Test" edit box above.

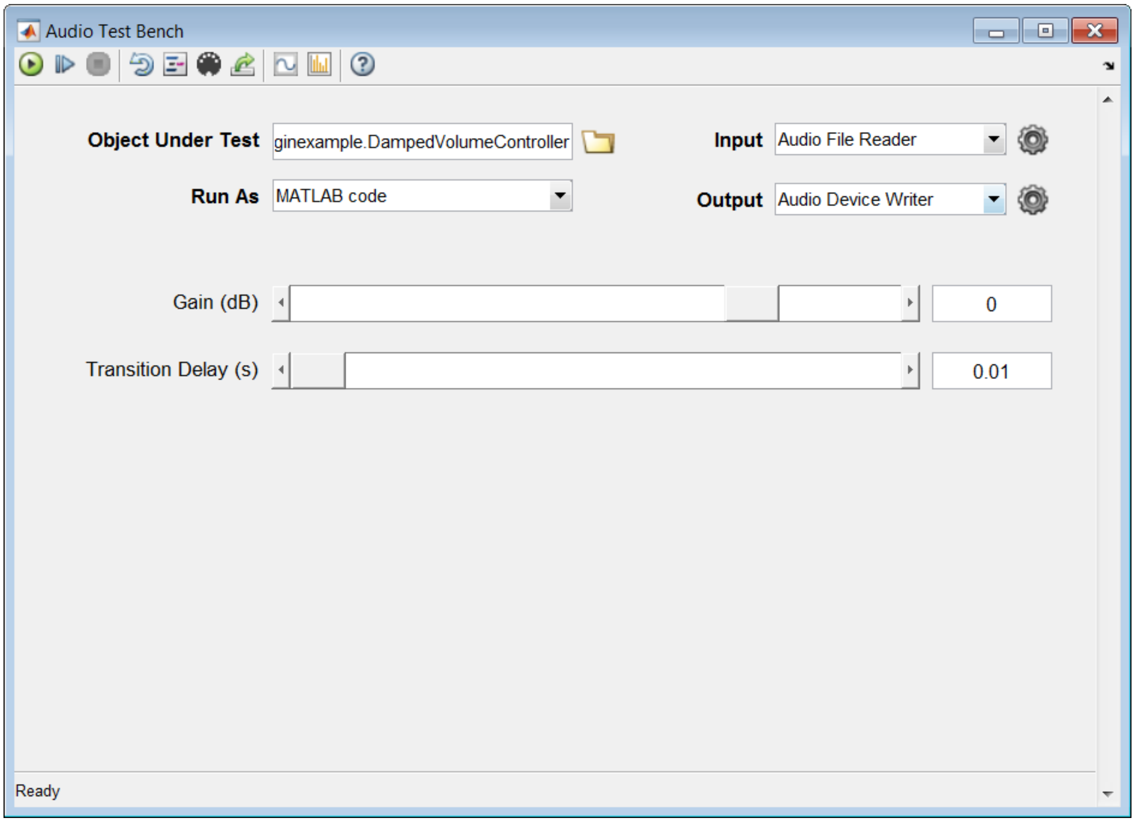
2 In the **Object Under Test** box, enter `audiopluginexample.Strobe` and press **Enter**. The **Audio Test Bench** automatically populates itself with the tunable parameters of the `audiopluginexample.Strobe` audio plugin.



Object Under Test: audiopluginexample.Strobe
Run As: MATLAB code
Input: Audio File Reader
Output: Audio Device Writer
Period (seconds): 0.05
Fill: Silence
Threshold (%): 0
Sync Period with Dynamics

The mapping between the tunable parameters of your object and the UI widgets on the **Audio Test Bench** is determined by `audioPluginInterface` and `audioPluginParameter` in the class definition of your object.

**3** In the **Object Under Test** box, enter `audiopluginexample.DampedVolumeController` and press **Enter**. The **Audio Test Bench** automatically populates itself with the tunable parameters of the `audiopluginexample.DampedVolumeController` audio plugin.



**Object Under Test** ginexample.DampedVolumeController

**Run As** MATLAB code

**Input** Audio File Reader

**Output** Audio Device Writer

Gain (dB) 0

Transition Delay (s) 0.01

Ready

## Run Audio Test Bench

To run the **Audio Test Bench** for your plugin with default settings, click ▶. Move the sliders to modify the **Gain (dB)** and **Transition Delay (s)** parameters while streaming.

To stop the audio stream loop, click ■. The MATLAB command line and objects used by the test bench are now released.

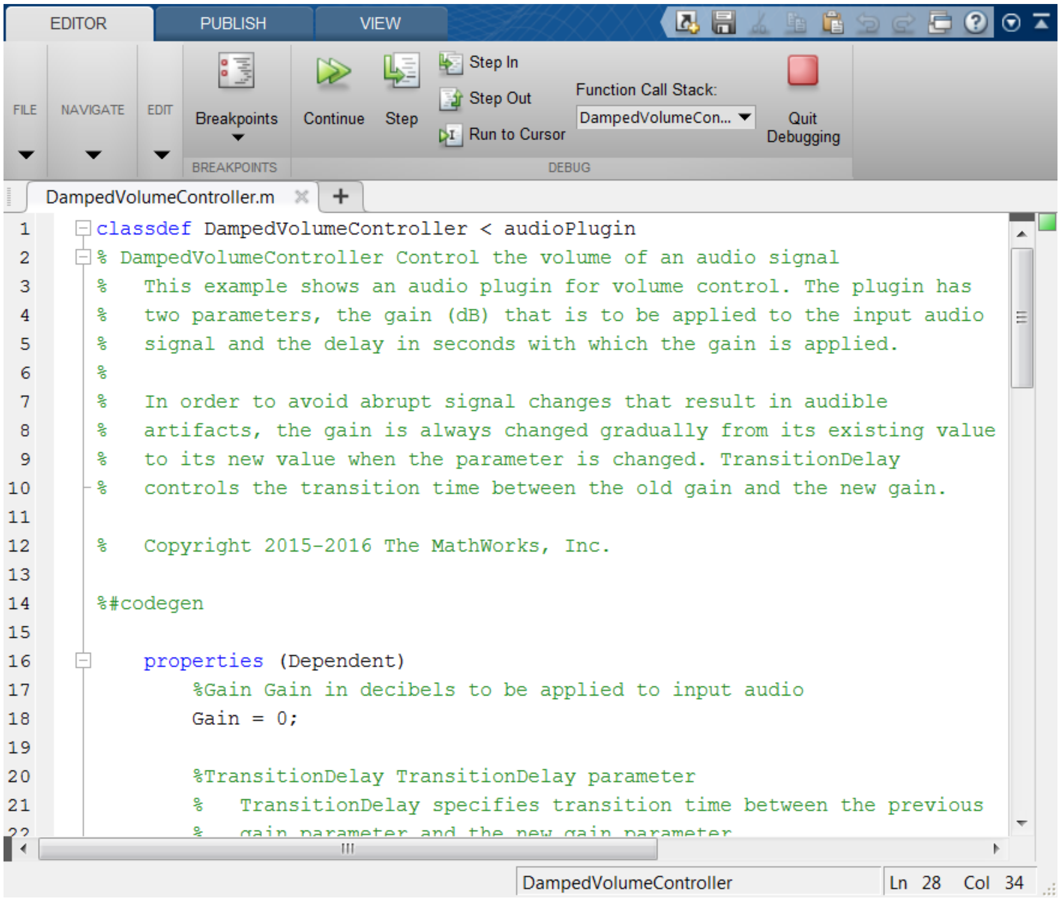
To reset internal states of your audio plugin and return the sliders to their initial positions, click ↺.

Click ▶ to run the audio test bench again.

## Debug Source Code of Audio Plugin

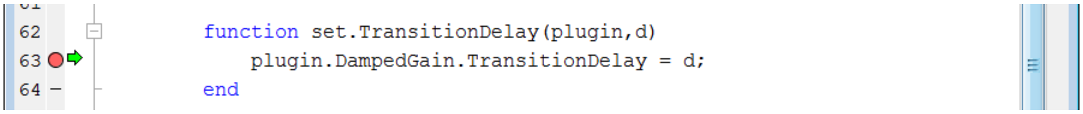
To pause the **Audio Test Bench**, click ⏸.

To open the source file of your audio plugin, click .



```
classdef DampedVolumeController < audioPlugin
% DampedVolumeController Control the volume of an audio signal
%   This example shows an audio plugin for volume control. The plugin has
%   two parameters, the gain (dB) that is to be applied to the input audio
%   signal and the delay in seconds with which the gain is applied.
%
%   In order to avoid abrupt signal changes that result in audible
%   artifacts, the gain is always changed gradually from its existing value
%   to its new value when the parameter is changed. TransitionDelay
%   controls the transition time between the old gain and the new gain.

%   Copyright 2015-2016 The MathWorks, Inc.

%#codegen

    properties (Dependent)
        %Gain Gain in decibels to be applied to input audio
        Gain = 0;

        %TransitionDelay TransitionDelay parameter
        %   TransitionDelay specifies transition time between the previous
        %   gain parameter and the new gain parameter
```

You can inspect the source code of your audio plugin, set breakpoints on it, and modify the code. Set a breakpoint at line 63, and then click ▶ on your **Audio Test Bench**.

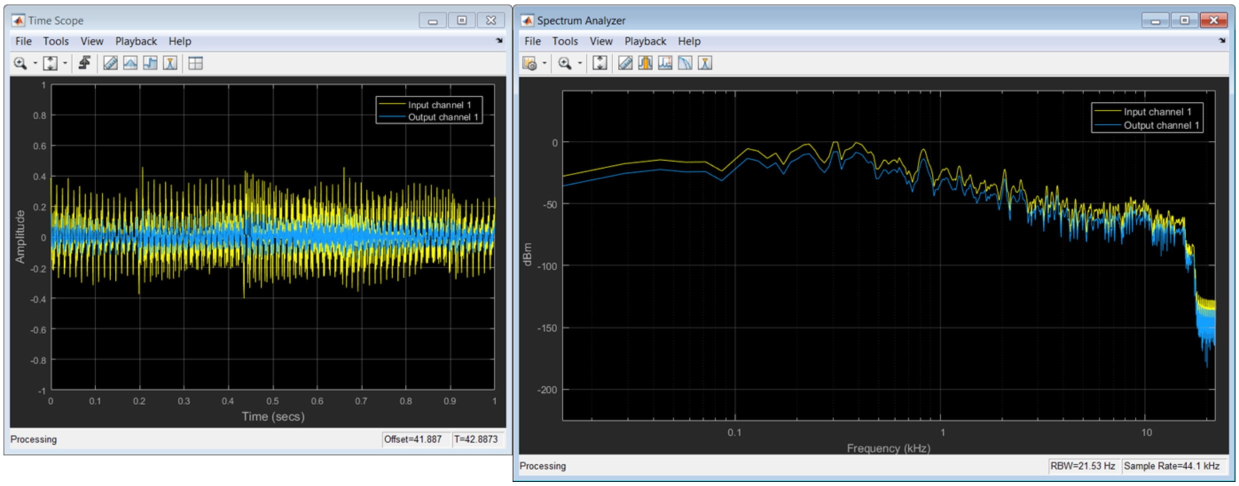


```
    function set.TransitionDelay(plugin,d)
        plugin.DampedGain.TransitionDelay = d;
    end
```

The **Audio Test Bench** runs your plugin until it reaches the breakpoint. To reach the breakpoint, move the **Transition Delay (s)** slider on the UI. To quit debugging, remove the breakpoint. In the MATLAB editor, click **Quit Debugging**.

## Open Scopes

To open a time scope to visualize the time-domain input and output for your audio plugin, click . To open a spectrum analyzer to visualize the frequency-domain input and output, click .



To release objects and stop the audio stream loop, click .

## Configure Input to Audio Test Bench

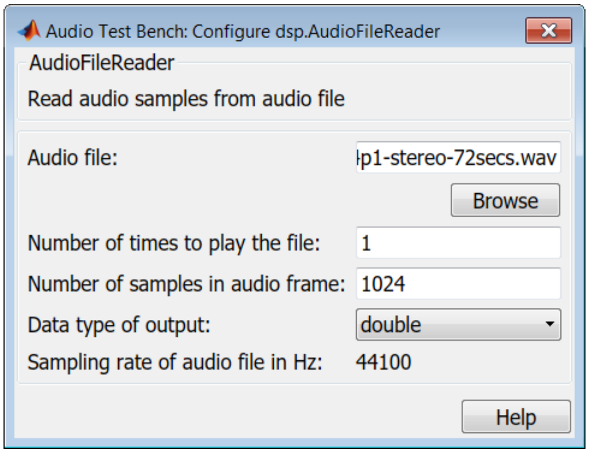
The **Input** list contains these options:

- `Audio File Reader` — dsp.AudioFileReader
- `Audio Device Reader` — audioDeviceReader

- `Audio Oscillator` — audioOscillator
- `Wavetable Synthesizer` — wavetableSynthesizer
- `Chirp Signal` — dsp.Chirp
- `Colored Noise` — dsp.ColoredNoise

1 Select `Audio File Reader`.

2 Click to open a UI for `Audio File Reader` configuration.



Audio Test Bench: Configure dsp.AudioFileReader

AudioFileReader

Read audio samples from audio file

Audio file: ‹p1-stereo-72secs.wav

Number of times to play the file: 1

Number of samples in audio frame: 1024

Data type of output: double

Sampling rate of audio file in Hz: 44100

You can enter any file name included on the MATLAB path. To specify a file that is not on that MATLAB path, specify the file path completely.

3 In the **Name of audio file from which to read** box, enter: `RockDrums-44p1-stereo-11secs.mp3`

Press **Enter**, and then exit the `Audio File Reader` configuration UI. To run the audio test bench with your new input, click .

To release your output object and stop the audio stream loop, click .

## Configure Output from Audio Test Bench

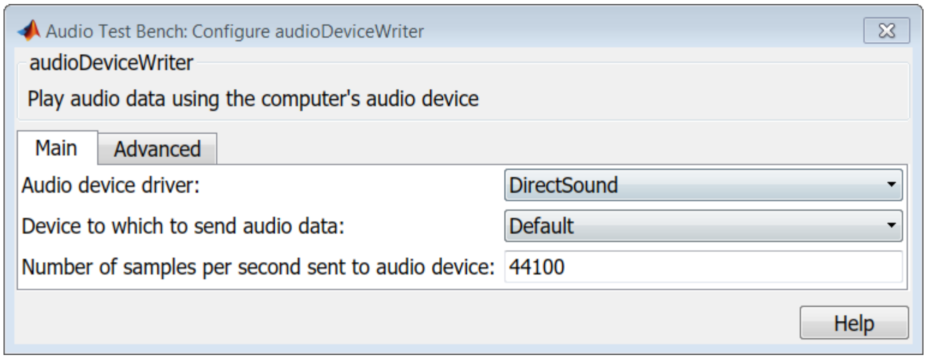
The **Output** list contains these options:

- `Audio Device Writer` — audioDeviceWriter

- `Audio File Writer` — dsp.AudioFileWriter
- `Both` — audioDeviceWriter and dsp.AudioFileWriter

Choose to output to device and file by selecting `Both` from the **Output** menu.

To open a UI for `Audio Device Writer` and `Audio File Writer` configuration, click ⚙.


Audio Test Bench: Configure audioDeviceWriter

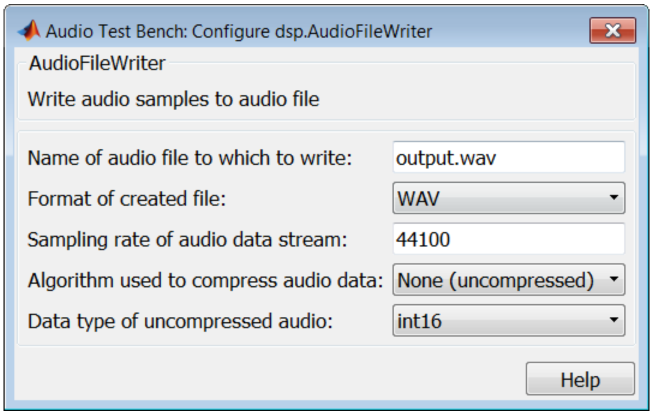
audioDeviceWriter

Play audio data using the computer's audio device

Audio device driver: DirectSound

Device to which to send audio data: Default

Number of samples per second sent to audio device: 44100


Audio Test Bench: Configure dsp.AudioFileWriter

AudioFileWriter

Write audio samples to audio file

Name of audio file to which to write: output.wav

Format of created file: WAV

Sampling rate of audio data stream: 44100

Algorithm used to compress audio data: None (uncompressed)

Data type of uncompressed audio: int16

## Synchronize Plugin Property with MIDI Control

If you have a MIDI device connected to your computer, you can synchronize plugin properties with MIDI controls. To open a MIDI configuration UI, click ●. Synchronize the `Gain` and `TransitionDelay` properties with MIDI controls you choose. Click **OK**.

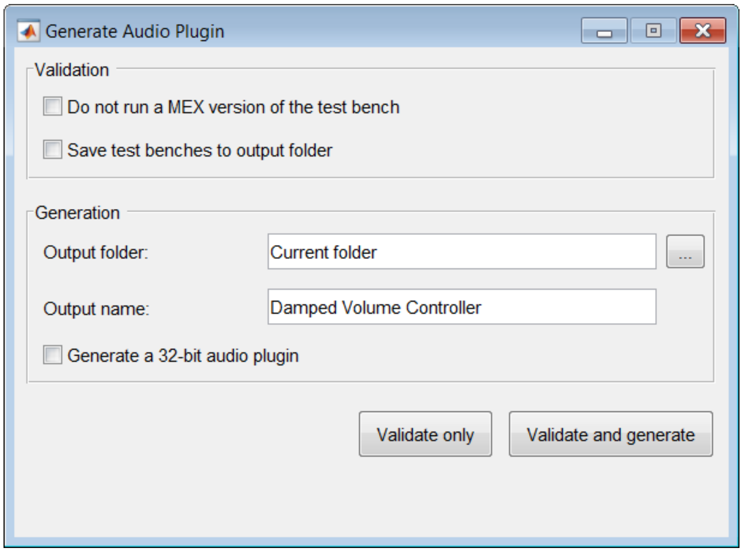
See `configureMIDI` for more information.

## Play the Audio and Save the Output File

To run your audio plugin, click . Adjust your plugin properties in real time using your synchronized MIDI controls and UI sliders. Your processed audio file is saved to the current folder.

## Validate and Generate Audio Plugin

To open the validation and generation dialog box, click .



Generate Audio Plugin

Validation

- [ ] Do not run a MEX version of the test bench
- [ ] Save test benches to output folder

Generation

Output folder: Current folder

Output name: Damped Volume Controller

- [ ] Generate a 32-bit audio plugin

You can validate only, or validate and generate your MATLAB audio plugin code in VST 2 plugin format. The **Generate a 32-bit audio plugin** check box is available only on win64 machines. See `validateAudioPlugin` and `generateAudioPlugin` for more information.

## See Also

**Apps**
Audio Test Bench

**Functions**
`generateAudioPlugin` | `validateAudioPlugin`

**Classes**
audioPlugin

## More About

- "Design an Audio Plugin"
- "Audio Plugin Example Gallery" on page 5-2
- "Export a MATLAB Plugin to a DAW"

5

# Audio Plugin Example Gallery

## Audio Plugin Example Gallery

Use these Audio System Toolbox plugin examples as buliding blocks in larger systems, as models for design patterns, or as benchmarks for comparison. Search the plugin descriptions to find an example that meets your needs.

### Audio Effects

### Filters

### Gain Control

### Communicate Between MATLAB and DAW

### Music Information Retrieval

### Speech Processing

### Audio Plugin Examples

| For a list of available audio plugins, see the online documentation. |
|---|

### See Also

**Apps**
Audio Test Bench

**Classes**
audioPlugin | audioPluginSource

**Functions**
`audioPluginInterface` | `audioPluginParameter`

### More About

- "Audio Test Bench Walkthrough" on page 4-2

- “Design an Audio Plugin”

# Equalization

# Equalization

Equalization (EQ) is the process of weighting the frequency spectrum of an audio signal.

You can use equalization to:

- Enhance audio recordings
- Analyze spectral content

Types of equalization include:

- Lowpass and highpass filters — Attenuate high frequency and low frequency content, respectively.
- Low-shelf and high-shelf equalizers — Boost or cut frequencies equally above or below a desired cutoff point.
- Parametric equalizers — Selectively boost or cut frequency bands. Also known as peaking filters.

This tutorial describes how to create equalizers using these Audio System Toolbox design functions: `designParamEQ`, `designShelvingEQ`, and `designVarSlopeFilter`. The toolbox also contains the `multibandParametricEQ` System object, which combines the filter design functions into a multiband parametric equalizer. For a tutorial focused on using these functions in MATLAB, see “Parametric Equalizer Design”.

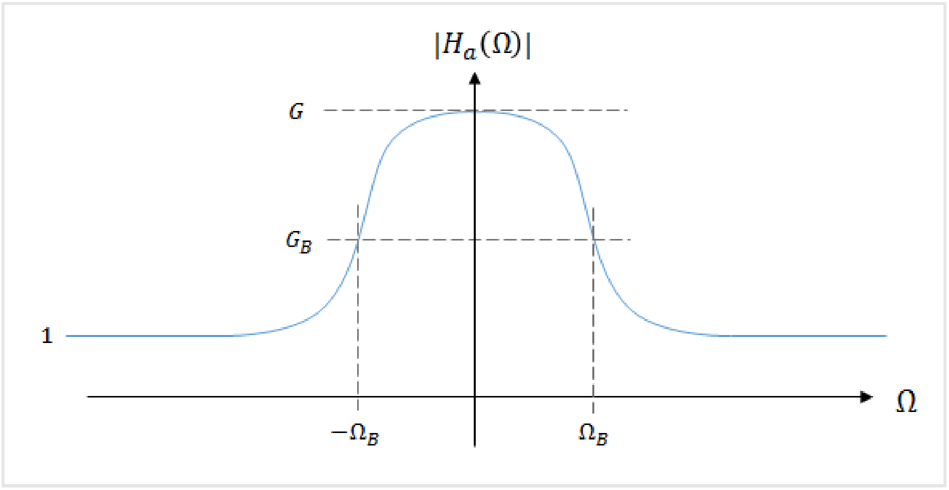
## Equalization Design Using Audio System Toolbox

### EQ Filter Design

Audio System Toolbox design functions use the bilinear transform method of digital filter design to determine your equalizer coefficients. In the bilinear transform method, you:

1. Choose an analog prototype.
2. Specify filter design parameters.
3. Perform the bilinear transformation.

### Analog Low-Shelf Prototype



Audio System Toolbox uses the high-order parametric equalizer design presented in [1]. In this design method, the analog prototype is taken to be a low-shelf Butterworth filter:

$$H_a(s) = \left[\frac{g\beta + s}{\beta + s}\right]^r \prod_{i=1}^{L}\left[\frac{g^2\beta^2 + 2gs_i\beta s + s^2}{\beta^2 + 2s_i\beta s + s^2}\right]$$

- $L$ = Number of analog SOS sections
- $N$ = Analog filter order
- $r = \begin{cases} 0 & \text{N even} \\ 1 & \text{N odd} \end{cases}$
- $g = G^{1/N}$

- $\beta = \Omega_B \times \left(\sqrt{\dfrac{G^2 - G_B^2}{G_B^2 - 1}}\right)^{-1/N} = \tan\left(\pi \dfrac{\Delta\omega}{2}\right) \times \left(\sqrt{\dfrac{G^2 - G_B^2}{G_B^2 - 1}}\right)^{-1/N}$, where *Δω* is the desired digital bandwidth

- $s_i = \sin\left(\dfrac{(2i-1)\pi}{2N}\right), \quad i = 1,2,\ldots,L$

For parametric equalizers, the analog prototype is reduced by setting the bandwidth gain to the square root of the peak gain ($G_B$ = *sqrt(G)*).

After the design parameters are specified, the analog prototype is transformed directly to the desired digital equalizer by a bandpass bilinear transformation:

$$s = \frac{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}{1 - z^{-2}}$$

$\omega_0$ is the desired digital center frequency.

This transformation doubles the filter order. Every first-order analog section becomes a second-order digital section. Every second-order analog section becomes a fourth-order digital section. Audio System Toolbox always calculates fourth-order digital sections, which means that returning second-order sections requires the computation of roots, and is less efficient.

### Digital Coefficients

The digital transfer function is implemented as a cascade of second-order and fourth-order sections.

$$H(z) = \left[\frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{1 + a_{01}z^{-1} + a_{02}z^{-2}}\right]^r \prod_{i=1}^{L}\left[\frac{b_{i0} + b_{i1}z^{-1} + b_{i2}z^{-2} + b_{i3}z^{-3} + b_{i4}z^{-4}}{1 + a_{i1}z^{-1} + a_{i2}z^{-2} + a_{i3}z^{-3} + a_{i4}z^{-4}}\right]$$

The coefficients are given by performing the bandpass bilinear transformation on the analog prototype design.

| Second-Order Section Coefficients | Fourth-Order Section Coefficients |
|---|---|
| $D_0 = \beta + 1$<br>$b_{00} = (1 + g\beta) / D_0$<br>$b_{01} = -2\cos(\omega_0) / D_0$<br>$b_{02} = (1 - g\beta) / D_0$<br>$a_{01} = -2\cos(\omega_0) / D_0$<br>$a_{02} = (1 - \beta) / D_0$ | $D_i = \beta^2 + 2s_i\beta + 1$<br>$b_{i0} = \left(g^2\beta^2 + 2gs_i\beta + 1\right) / D_i$<br>$b_{i1} = -4c_0 (1 + gs_i\beta) / D_i$<br>$b_{i2} = 2\left(1 + 2\cos^2(\omega_0) - g^2\beta^2\right) / D_i$<br>$b_{i3} = -4c_0 (1 - gs_i\beta) / D_i$<br>$b_{i4} = \left(g^2\beta^2 - 2gs_i\beta + 1\right) / D_i$<br>$a_{i1} = -4c_0 (1 + s_i\beta) / D_i$<br>$a_{i2} = 2\left(1 + 2\cos^2(\omega_0) - \beta^2\right) / D_i$<br>$a_{i3} = -4\cos(\omega_0)(1 - s_i\beta) / D_i$<br>$a_{i4} = \left(\beta^2 - 2s_i\beta + 1\right) / D_i$ |

**Biquadratic Case**

In the biquadratic case, when $N = 1$, the coefficients reduce to:

$$D_0 = \frac{\Omega_B}{\sqrt{G}} + 1$$

$$b_{00} = \left(1 + \Omega_B\sqrt{G}\right) / D_0, \quad b_{01} = -2\cos(\omega_0) / D_0, \quad b_{02} = \left(1 - \Omega_B\sqrt{G}\right) / D_0$$

$$a_{01} = -2\cos(\omega_0) / D_0, \quad a_{02} = \left(1 - \frac{\Omega_B}{\sqrt{G}}\right) / D_0$$

Denormalizing the $a_{00}$ coefficient, and making substitutions of $A = sqrt(G)$, $\Omega_B \cong \alpha$ yields the familiar peaking EQ coefficients described in [2].

Orfanidis notes the approximate equivalence of $\Omega_B$ and $\alpha$ in [1].

By using trigonometric identities,

$$\Omega_B = \tan\left(\frac{\Delta\omega}{2}\right) = \sin(\omega_0)\sinh\left(\frac{\ln 2}{2} B\right),$$

where $B$ plays the role of an equivalent octave bandwidth.

Bristow-Johnson obtained an approximate solution for $B$ in [4]:

$$B = \frac{\omega_0}{\sin\omega_0} \times BW$$

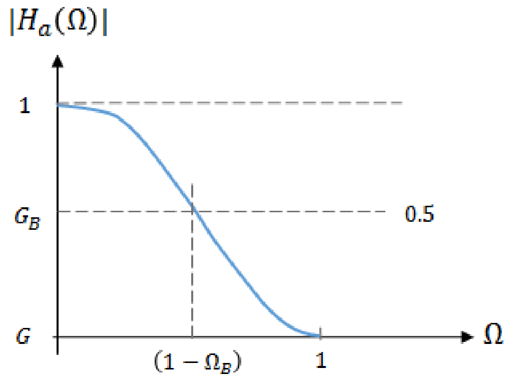
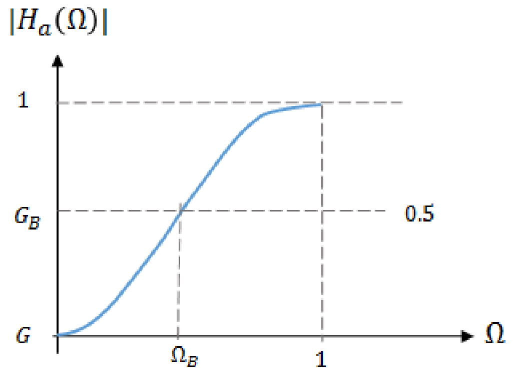
Substituting the approximation for $B$ into the $\Omega_B$ equation yields the definition of $\alpha$ in [2]:

$$\alpha = \sin(\omega_0)\sinh\left(\frac{\ln 2}{2} \times \frac{\omega_0}{\sin\omega_0} \times BW\right)$$

## Lowpass and Highpass Filter Design

### Analog Low-Shelf Prototype

To design lowpass and highpass filters, Audio System Toolbox uses a special case of the filter design for parametric equalizers. In this design, the peak gain, $G$, is set to 0, and $G_B^{\,2}$ is set to 0.5 (–3 dB cutoff). The cutoff frequency of the lowpass filter corresponds to 1 – $\Omega_B$. The cutoff frequency of the highpass filter corresponds to $\Omega_B$.

| Lowpass Form | Highpass Form |
|---|---|
|  |  |

### Digital Coefficients

The table summarizes the results of the bandpass bilinear transformation. The digital center frequency, $\omega_0$, is set to $\pi$ for lowpass filters and 0 for highpass filters.

| Second Order Section Coefficients | Fourth Order Section Coefficients |
|---|---|
| $D_0 = 1 + \tan\left(\pi\frac{\Delta\omega}{2}\right)$<br>$b_{00} = 1 / D_0$<br>$b_{01} = -2\cos(\omega_0) / D_0$<br>$b_{02} = 1 / D_0$<br>$a_{01} = -2\cos(\omega_0) / D_0$<br>$a_{02} = \left(1 - \tan\left(\pi\frac{\Delta\omega}{2}\right)\right) / D_0$ | $D_i = \tan^2\left(\pi\frac{\Delta\omega}{2}\right) + 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1$<br>$b_{i0} = 1 / D_i$<br>$b_{i1} = -4\cos(\omega_0) / D_i$<br>$b_{i2} = 2\left(1 + 2\cos^2(\omega_0)\right) / D_i$<br>$b_{i3} = -4\cos(\omega_0)_0 / D_i$<br>$b_{i4} = 1 / D_i$<br>$a_{i1} = -4\cos(\omega_0)\left(1 + s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right) / D_i$<br>$a_{i2} = 2\left(1 + 2\cos^2(\omega_0) - \tan^2\left(\pi\frac{\Delta\omega}{2}\right)\right) / D_i$<br>$a_{i3} = -4\cos(\omega_0)\left(1 - s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right) / D_i$<br>$a_{i4} = \left(\tan^2\left(\pi\frac{\Delta\omega}{2}\right) - 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1\right) / D_i$ |

## Shelving Filter Design

### Analog Prototype

Audio System Toolbox implements the shelving filter design presented in [2]. In this design, the high-shelf and low-shelf analog prototypes are presented separately:

$$H_L(s) = A\left(\frac{As^2 + \left(\sqrt{A}/Q\right)s + 1}{s^2 + \left(\sqrt{A}/Q\right)s + A}\right) \qquad H_H(s) = A\left(\frac{s^2 + \left(\sqrt{A}/Q\right)s + A}{As^2 + \left(\sqrt{A}/Q\right)s + 1}\right)$$

For compactness, the analog filters are presented with variables $A$ and $Q$. You can convert $A$ and $Q$ to available Audio System Toolbox design parameters:

$$A = 10^{G/40}$$
$$\frac{1}{Q} = \sqrt{\left(A + {}^{1}\!/_{A}\right)\left({}^{1}\!/_{slope} - 1\right) + 2}$$

After you specify the design parameters, the analog prototype is transformed to the desired digital shelving filter by a bilinear transformation with prewarping:

$$s = \left(\frac{z-1}{z+1}\right) \times \left(\frac{1}{\tan\left(\frac{\omega_0}{2}\right)}\right)$$

### Digital Coefficients

The table summarizes the results of the bilinear transformation with prewarping.

| | |
|---|---|
| Low-Shelf | $b_0 = A\left((A+1) - (A-1)\cos(\omega_0) + 2\alpha\sqrt{A}\right)$<br>$b_1 = 2A\left((A-1) - (A+1)\cos(\omega_0)\right)$<br>$b_2 = A\left((A+1) - (A-1)\cos(\omega_0) - 2\alpha\sqrt{A}\right)$<br>$a_0 = (A+1) + (A-1)\cos(\omega_0) + 2\alpha\sqrt{A}$<br>$a_1 = -2\left((A-1) + (A+1)\cos(\omega_0)\right)$<br>$a_2 = (A+1) + (A-1)\cos(\omega_0) - 2\alpha\sqrt{A}$ |
| High-Shelf | $b_0 = A\left((A+1) + (A-1)\cos(\omega_0) + 2\alpha\sqrt{A}\right)$<br>$b_1 = -2A\left((A-1) + (A+1)\cos(\omega_0)\right)$<br>$b_2 = A\left((A+1) + (A-1)\cos(\omega_0) - 2\alpha\sqrt{A}\right)$<br>$a_0 = (A+1) - (A-1)\cos(\omega_0) + 2\alpha\sqrt{A}$<br>$a_1 = 2\left((A-1) + (A+1)\cos(\omega_0)\right)$<br>$a_2 = (A+1) - (A-1)\cos(\omega_0) - 2\alpha\sqrt{A}$ |

Intermediate Variables

$$\alpha = \frac{\sin(\omega_0)}{2}\sqrt{\left(A + \frac{1}{A}\right)\left(\frac{1}{slope} - 1\right) + 2A}$$

$$\omega_0 = 2\pi \frac{Cutoff\ Frequency}{Fs}$$

## References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026–1046.

[2] Bristow-Johnson, Robert. "Cookbook Formulae for Audio EQ Biquad Filter Coefficients." Accessed March 02, 2016. http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt.

[3] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[4] Bristow-Johnson, Robert. “The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers.” Presented at the 97th Convention of the AES, San Francisco, November 1994, AES Preprint 3906.

## See Also

### Functions
`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

### System Objects
multibandParametricEQ

## More About

- “Parametric Equalizer Design”

# Deployment

## Functions and System Objects Supported for MATLAB Coder

If you have a MATLAB Coder license, you can generate C and C++ code from MATLAB code that contains Audio System Toolbox functions and System objects. For more information about C and C++ code generation from MATLAB code, see the MATLAB Coder documentation. For more information about generating code from System objects, see "System Objects in MATLAB Code Generation" (MATLAB Coder).

The following Audio System Toolbox functions and System objects are supported for C and C++ code generation from MATLAB code.

| Name | Remarks and Limitations |
|---|---|
| **Audio I/O and Waveform Generation** | |
| audioPlayerRecorder | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| audioDeviceReader | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| audioDeviceWriter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| wavetableSynthesizer | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| audioOscillator | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| **Audio Processing Algorithm Design** | |
| `designVarSlopeFilter` | Supports MATLAB Function block: Yes |
| `designParamEQ` | Supports MATLAB Function block: Yes |
| `designShelvingEQ` | Supports MATLAB Function block: Yes |

| Name | Remarks and Limitations |
|---|---|
| `integratedLoudness` | Supports MATLAB Function block: Yes |
| crossoverFilter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| compressor | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| expander | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| noiseGate | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| limiter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| multibandParametricEQ | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| octaveFilter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| weightingFilter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |

| Name | Remarks and Limitations |
|---|---|
| loudnessMeter | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: No<br><br>Dynamic Memory Allocation must not be turned off. |
| reverberator | "System Objects in MATLAB Code Generation" (MATLAB Coder)<br><br>Supports MATLAB Function block: Yes |
| **Audio Plugins** | |
| `audioPluginInterface` | Supports MATLAB Function block: Yes |
| `audioPluginParameter` | Supports MATLAB Function block: Yes |
| audioPlugin | Supports MATLAB Function block: Yes |
| audioPluginSource | Supports MATLAB Function block: Yes |

## Functions and System Objects Supported for MATLAB Compiler

If you have a MATLAB Compiler license, you can generate standalone applications that contain Audio System Toolbox functions, System objects, and classes.

The following Audio System Toolbox functions, System objects, and classes are supported for generating standalone applications from MATLAB code.

| Name | Remarks and Limitations |
|---|---|
| **Audio I/O and Waveform Generation** | |
| audioPlayerRecorder | |
| audioDeviceReader | |
| audioDeviceWriter | |
| wavetableSynthesizer | |
| audioOscillator | |
| **Audio Processing Algorithm Design** | |
| `designVarSlopeFilter` | |
| `designParamEQ` | |
| `designShelvingEQ` | |
| `integratedLoudness` | |
| crossoverFilter | |
| compressor | |
| expander | |
| noiseGate | |
| limiter | |
| multibandParametricEQ | |
| octaveFilter | |
| weightingFilter | |
| loudnessMeter | |
| reverberator | |
| **Simulation, Tuning, and Visualization** | |

| Name | Remarks and Limitations |
|---|---|
| midiid | |
| midicontrols | |
| midiread | |
| midisync | |
| midicallback | |
| getMIDIConnections | |
| configureMIDI | User interface not supported. |
| disconnectMIDI | |
| **Audio Plugins** | |
| audioPluginInterface | |
| audioPluginParameter | |
| audioPlugin | |
| audioPluginSource | |

# Desktop Real-Time Audio Acceleration with MATLAB Coder

This example shows how to accelerate a real-time audio application using C code generation with MATLAB® Coder™. You must have the MATLAB Coder™ software installed to run this example.

## Introduction

Replacing parts of your MATLAB code with an automatically generated MATLAB executable (MEX-function) can speed up simulation. Using MATLAB Coder, you can generate readable and portable C code and compile it into a MEX-function that replaces the equivalent section of your MATLAB algorithm.

This example showcases code generation using an audio notch filtering application.

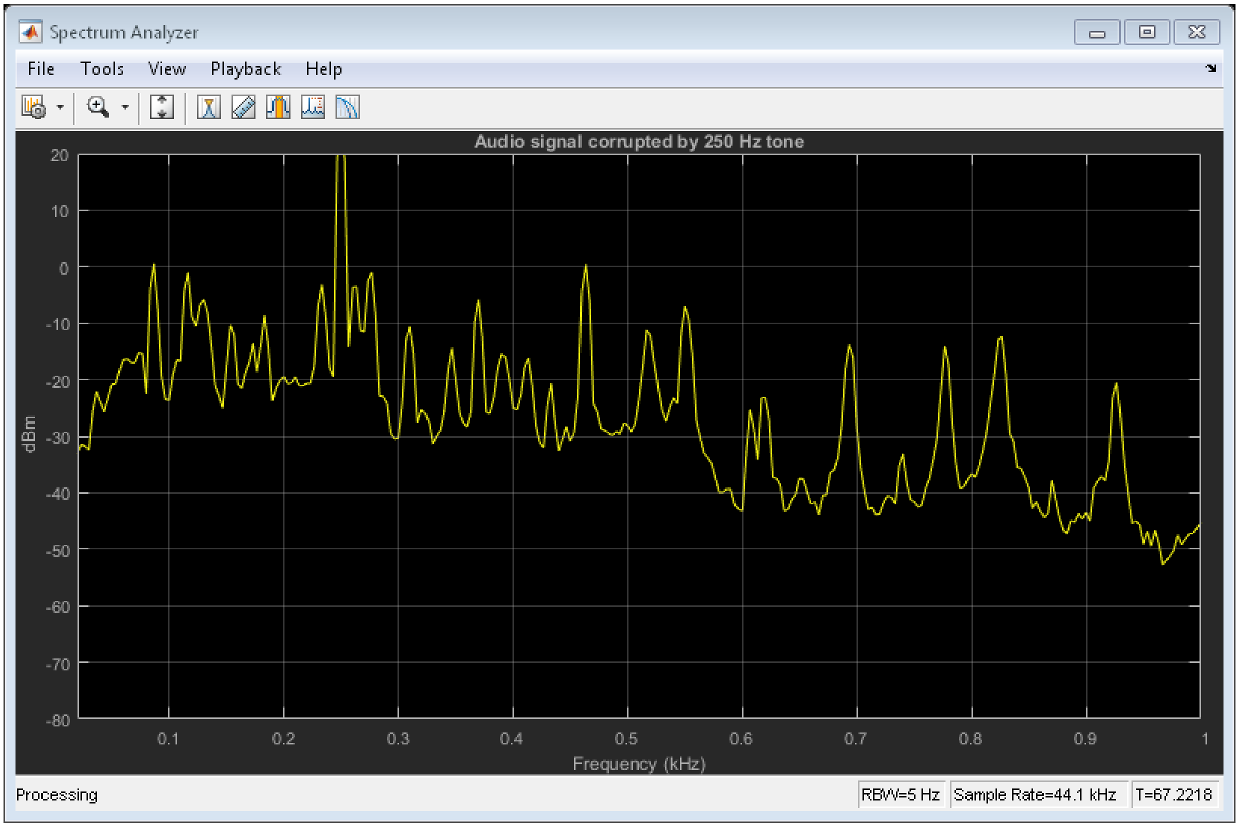
## Notch Filtering

A notch filter is used to eliminate a specific frequency from a signal. Typical filter design parameters for notch filters are the notch center frequency and the 3 dB bandwidth. The center frequency is the frequency at which the filter has a linear gain of zero. The 3 dB bandwidth measures the frequency width of the notch of the filter computed at the half-power or 3 dB attenuation point.

The helper function used in this example is helperAudioToneRemoval. The function reads an audio signal corrupted by a 250 Hz sinusoidal tone from a file. helperAudioToneRemoval uses a notch filter to remove the interfering tone and writes the filtered signal to a file.

You can visualize the corrupted audio signal using a spectrum analyzer.

```
scope = dsp.SpectrumAnalyzer('SampleRate',44.1e3,...
    'RBWSource','Property','RBW',5,...
    'PlotAsTwoSidedSpectrum',false,...
    'SpectralAverages',10,...
    'FrequencySpan','Start and stop frequencies',...
    'StartFrequency',20,...
    'StopFrequency',1000,...
    'Title','Audio signal corrupted by 250 Hz tone');
reader = dsp.AudioFileReader('guitar_plus_tone.ogg');

while ~isDone(reader)
    audio = reader();
    scope(audio(:,1));
    end
```



### C Code Generation Speedup

Measure the time it takes to read the audio file, filter out the interfering tone, and write the filtered output using MATLAB code. Because helperAudioToneRemoval writes an audio file output, you must have write permission in the current directory. To ensure write access, change directory to your system's temporary folder.

```
mydir = pwd; addpath(mydir); cd(tempdir);
tic;
helperAudioToneRemoval;
t1 = toc;
fprintf('MATLAB Simulation Time: %d\n',t1);
```

```
MATLAB Simulation Time: 3.898437e+00
```

Next, generate a MEX-function from `helperAudioToneRemoval` using the MATLAB Coder function, codegen.

```
codegen helperAudioToneRemoval
```

Measure the time it takes to execute the MEX-function and calculate the speedup gain with a compiled function.

```
tic;
helperAudioToneRemoval_mex
t2 = toc;
fprintf('Code Generation Simulation Time: %d\n',t2);
fprintf('Speedup factor: %6.2f\n',t1/t2);

cd(mydir);
```

```
Code Generation Simulation Time: 3.961503e+00
Speedup factor:   0.98
```

## Related Examples

- “Generate Standalone Executable for Parametric Audio Equalizer”
- “Deploy Audio Applications with MATLAB Compiler”

## More About

- “Functions and System Objects Supported for MATLAB Coder” on page 7-2
- “Functions and System Objects Supported for MATLAB Compiler” on page 7-5

# Audio I/O User Guide

## Run Audio I/O Features Outside MATLAB and Simulink

You can deploy these audio input and output features outside the MATLAB and Simulink environments:

### System Objects

- audioPlayerRecorder
- audioDeviceReader
- audioDeviceWriter
- dsp.AudioFileReader
- dsp.AudioFileWriter

### Blocks

- Audio Device Reader
- Audio Device Writer
- From Multimedia File
- To Multimedia File

The generated code for the audio I/O features relies on prebuilt dynamic library files included with MATLAB. You must account for these extra files when you run audio I/O features outside the MATLAB and Simulink environments. To run a standalone executable generated from a model or code containing the audio I/O features, set your system environment using commands specific to your platform.

| Platform | Command |
|---|---|
| Mac | `setenv DYLD_LIBRARY_PATH "${DYLD_LIBRARY_PATH}:$MATLABROOT/bin/maci64" (csh/tcsh)`<br><br>`export DYLD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MATLABROOT/bin/maci64 (Bash)` |
| Linux | `setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$MATLABROOT/bin/glnxa64 (csh/tcsh)` |

| Platform | Command |
|---|---|
| | `export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:$MATLABROOT/bin/ glnxa64 (Bash)` |
| Windows | `set PATH=%PATH%;%MATLABROOT%\bin \win64` |

The path in these commands is valid only on systems that have MATLAB installed. If you run the standalone app on a machine with only MCR, and no MATLAB installed, replace `$MATLABROOT/bin/...` with the path to the MCR.

To run the code generated from the above System objects and blocks on a machine does not have MCR or MATLAB installed, use the `packNGo` function. The `packNGo` function packages all relevant files in a compressed zip file so that you can relocate, unpack, and rebuild your project in another development environment with no MATLAB installed.

You can use the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility. For more details on how to pack the code generated from MATLAB code, see “Package Code for Other Development Environments” (MATLAB Coder). For more details on how to pack the code generated from Simulink blocks, see the `packNGo` function.

## More About

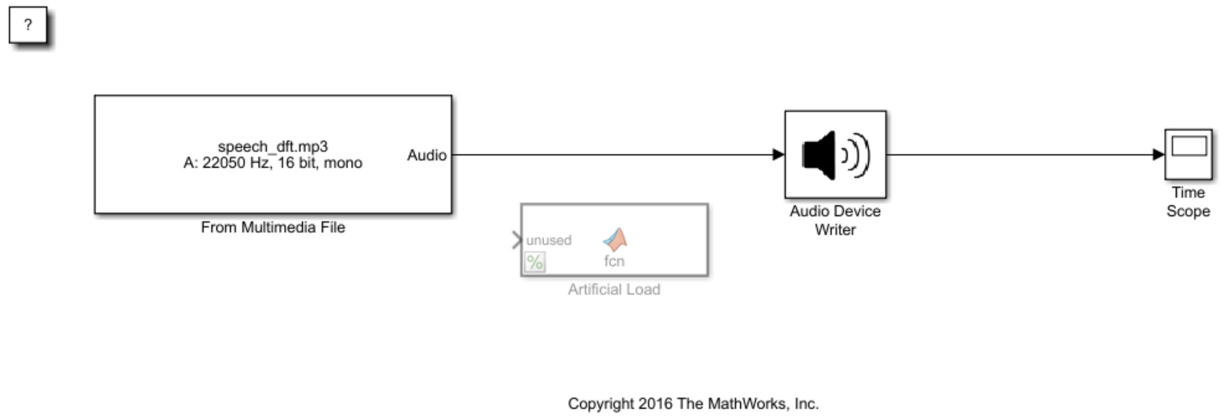
- “MATLAB Programming for Code Generation” (MATLAB Coder)

9

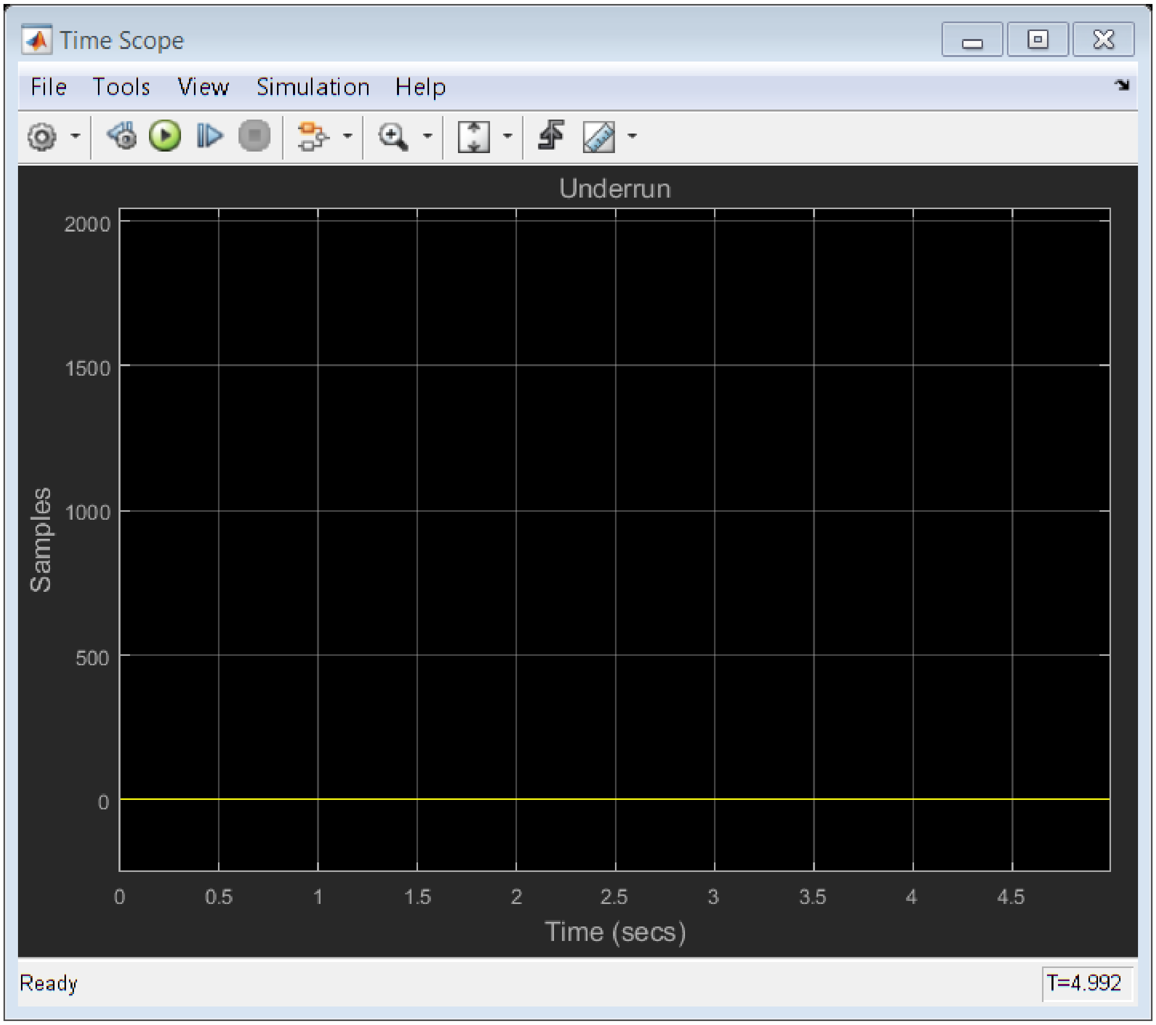
# Block Example Repository

## Decrease Underrun

Examine the Audio Device Writer block in a Simulink® model, determine underrun, and decrease underrun.



1. Run the model. The Audio Device Writer sends an audio stream to your computer's default audio output device. The Audio Device Writer block sends the number of samples underrun to your Time Scope.



2. Uncomment the Artificial Load block. This block performs computations that slow the simulation.

3. Run the model. If your device writer is dropping samples:

a. Stop the simulation.

b. Open the From Multimedia File block.

c. Set the **Samples per frame** parameter to `1024`.

d. Close the block and run the simulation.

If your model continues to drop samples, increase the frame size again. The increased frame size increases the buffer size used by the sound card. A larger buffer size decreases the possibility of underruns at the cost of higher audio latency.

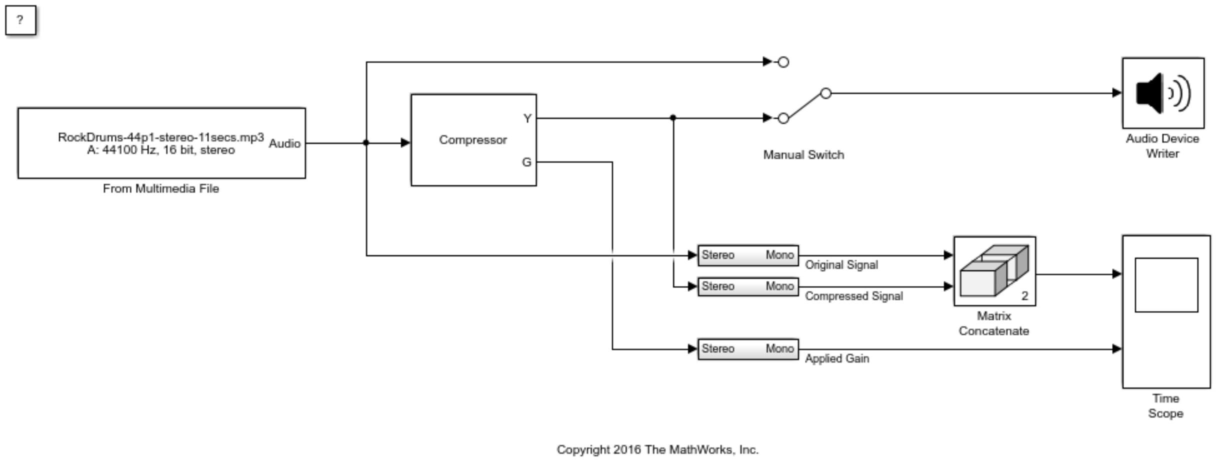
## See Also

From Multimedia File | Time Scope

# Block Example Repository

## Suppress Loud Sounds

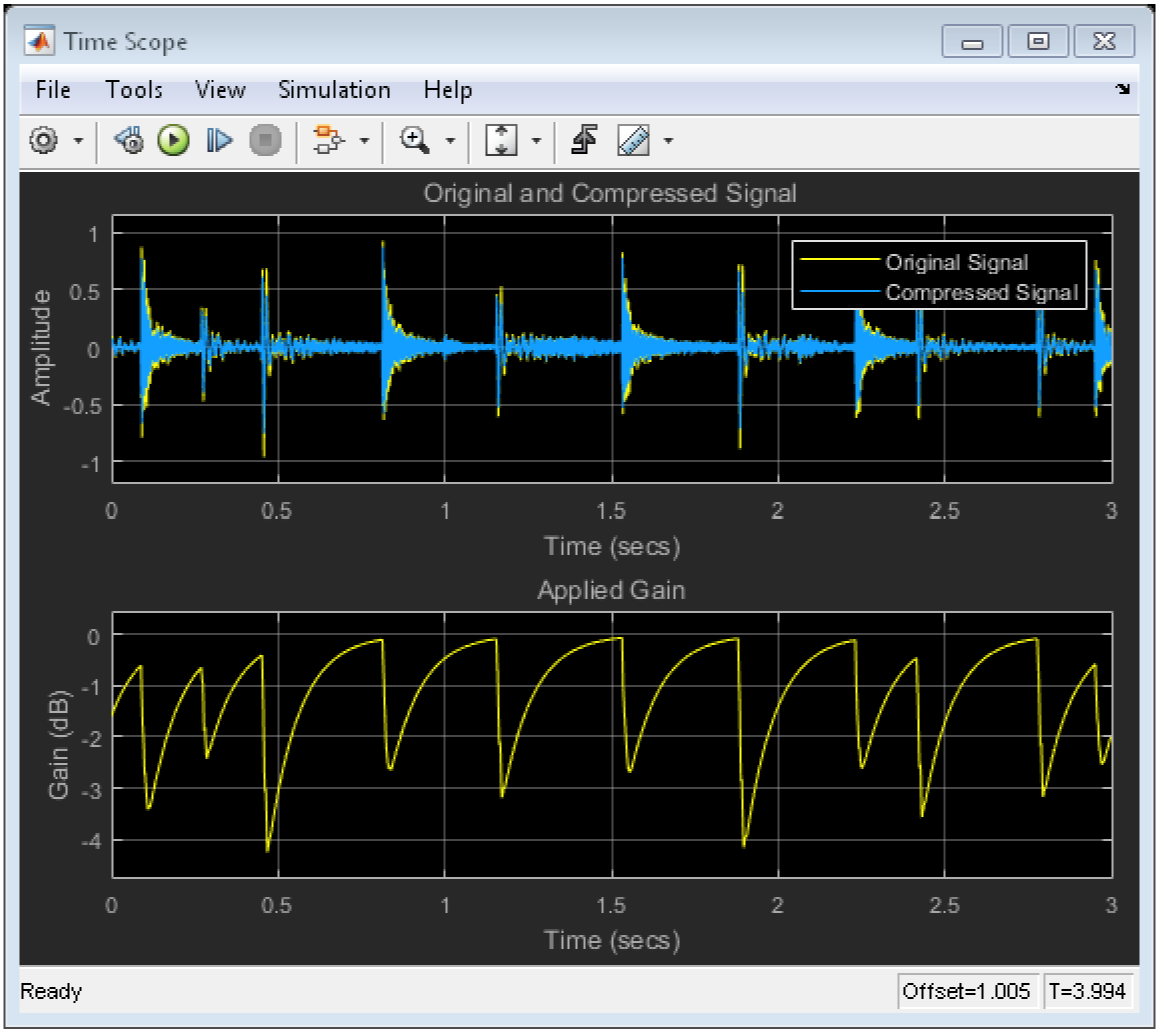
Use the Compressor block to suppress loud sounds and visualize the applied compression gain.



1. Open the Time Scope and Compressor blocks.

2. Run the model. To switch between listening to the compressed signal and the original signal, double-click the Manual Switch block.

3. Observe how the applied gain depends on compression parameters and input signal dynamics by tuning the Compressor block parameters and viewing the results on the Time Scope.



## See Also

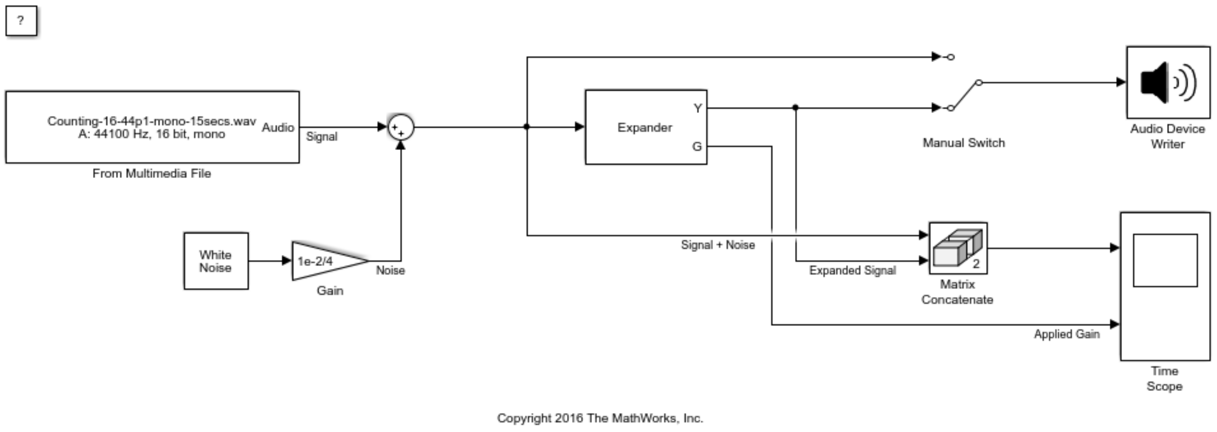
Audio Device Writer | Compressor | From Multimedia File | Matrix Concatenate | Time Scope

## More About

- "Dynamic Range Control" on page 1-2

## Attenuate Low-Level Noise

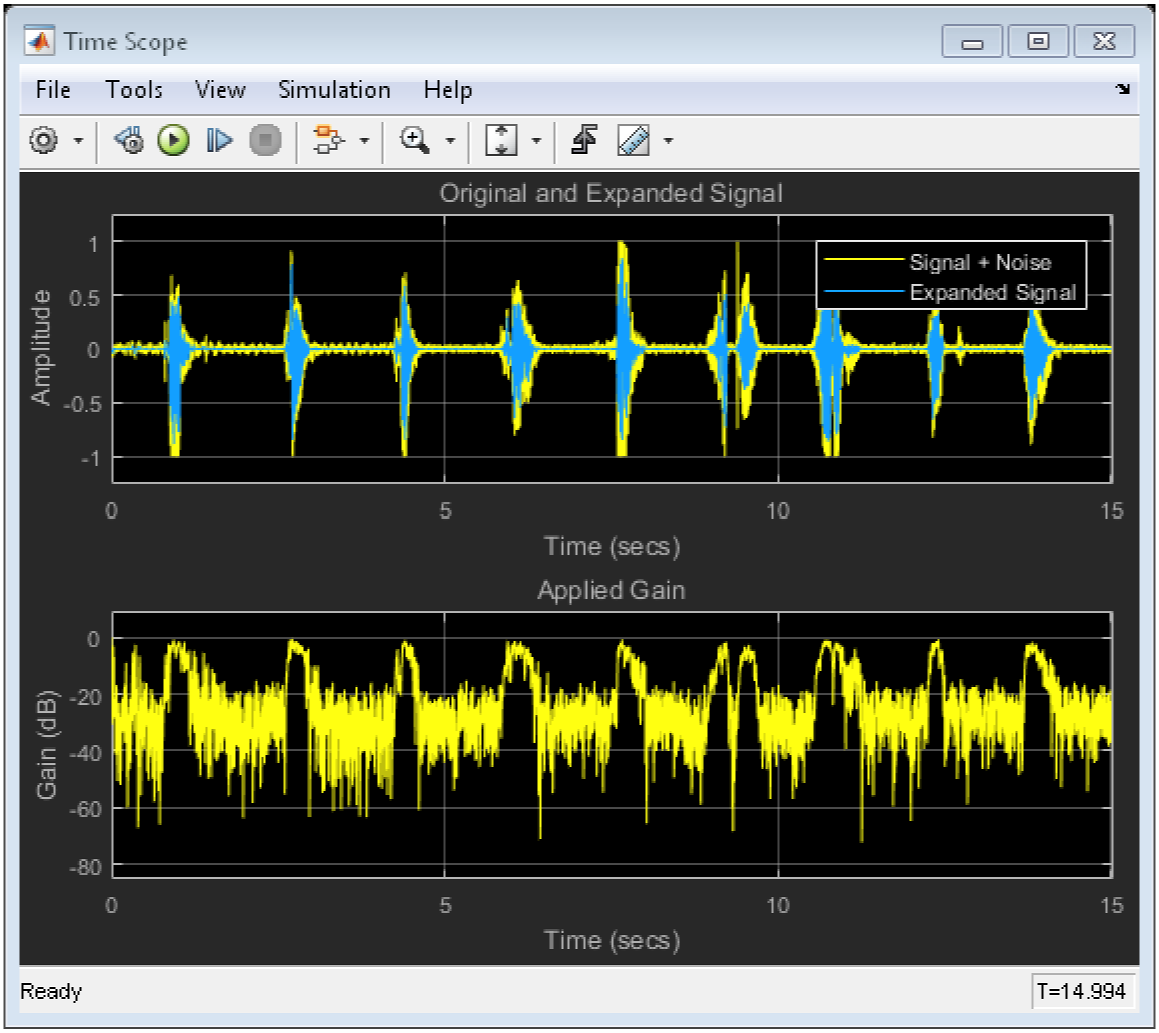
Use the Expander block to attenuate low-level noise and visualize the applied dynamic range control gain.



1. Open the Time Scope and Expander blocks.

2. Run the model. To switch between listening to the expanded signal and the original signal, double-click the Manual Switch block.

3. Observe how the applied gain depends on expansion parameters and input signal dyanmics by tuning the Expander block parameters and viewing the results on the Time Scope.



## See Also

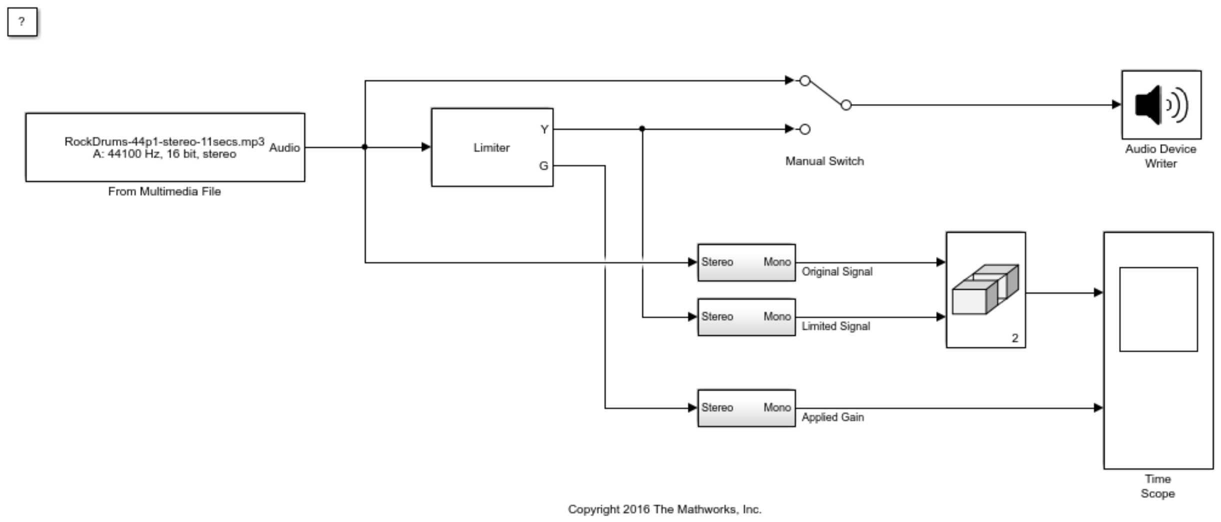
Audio Device Writer | Colored Noise | Expander | From Multimedia File | Matrix Concatenate | Time Scope

## More About

- "Dynamic Range Control" on page 1-2

## Suppress Volume of Loud Sounds

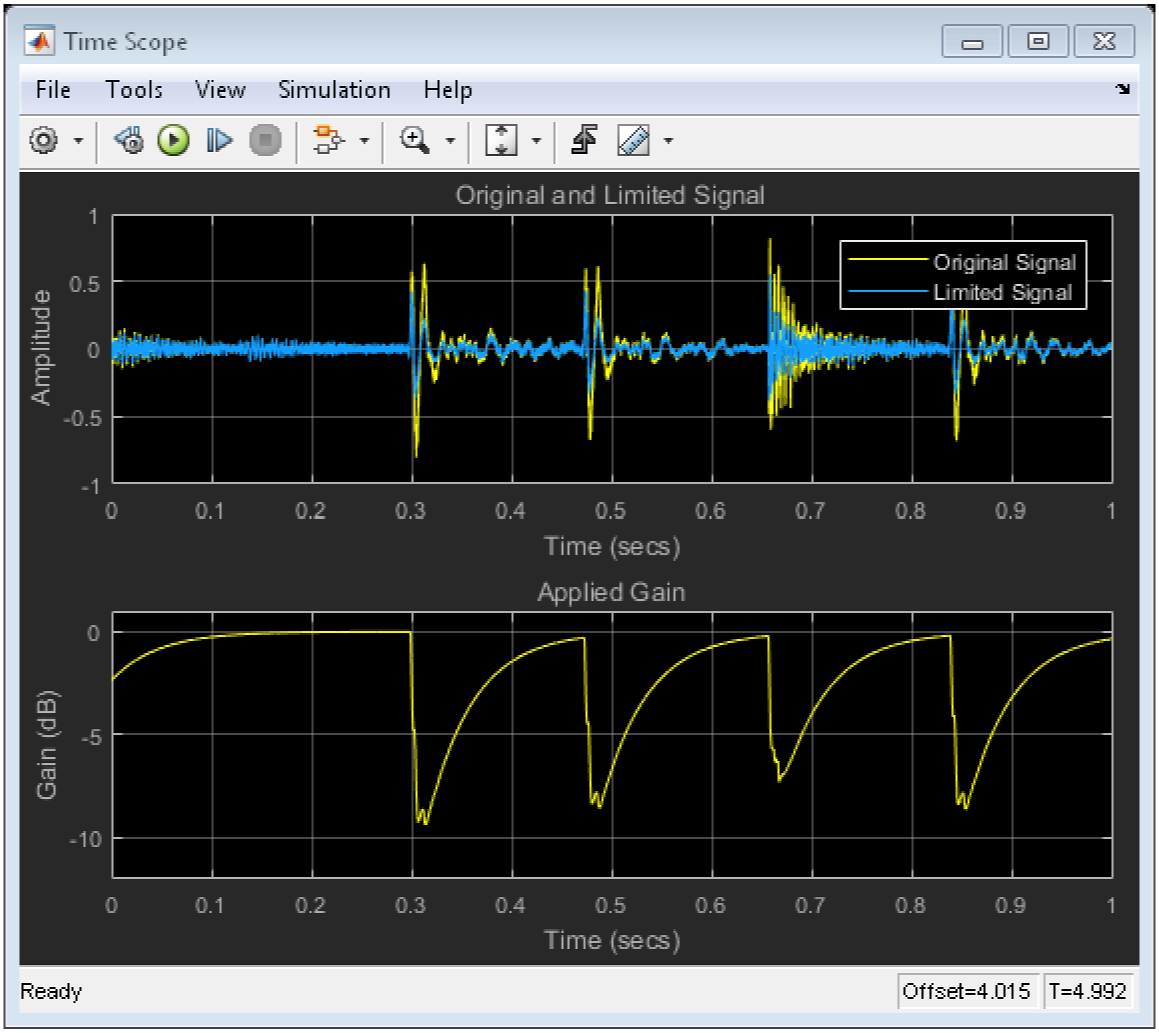
Suppress the volume of loud sounds and visualize the applied dynamic range control gain.



1. Open the Time Scope and Limiter blocks.

2. Run the model. To switch between listening to the gated signal and the original signal, double-click the Manual Switch block.

3. Observe how the applied gain depends on dynamic range limiting parameters and input signal dynamics by tuning Limiter block parameters and viewing the results on the Time Scope.



## See Also

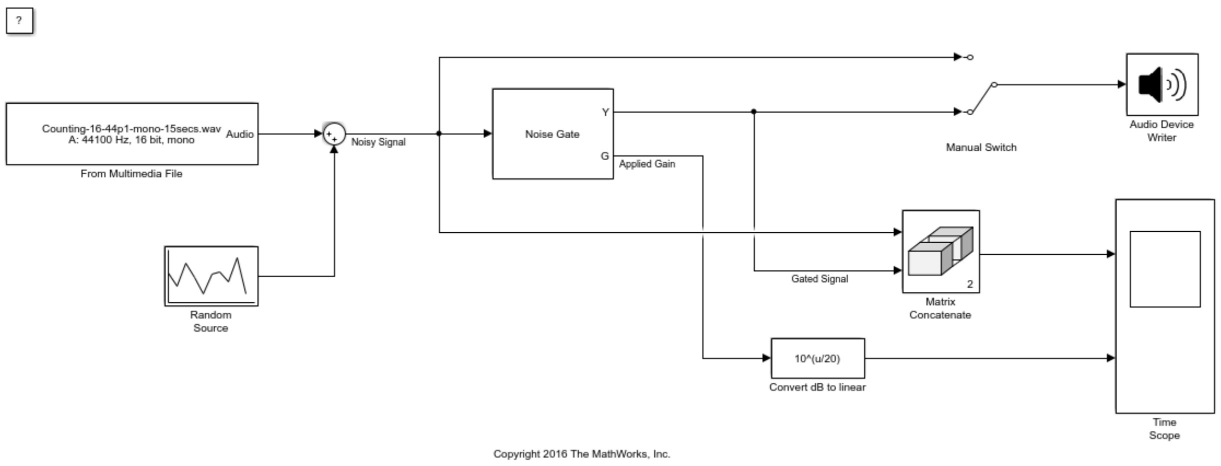
Audio Device Writer | From Multimedia File | Limiter | Matrix Concatenate | Time Scope

## More About

- “Dynamic Range Control” on page 1-2

## Gate Background Noise

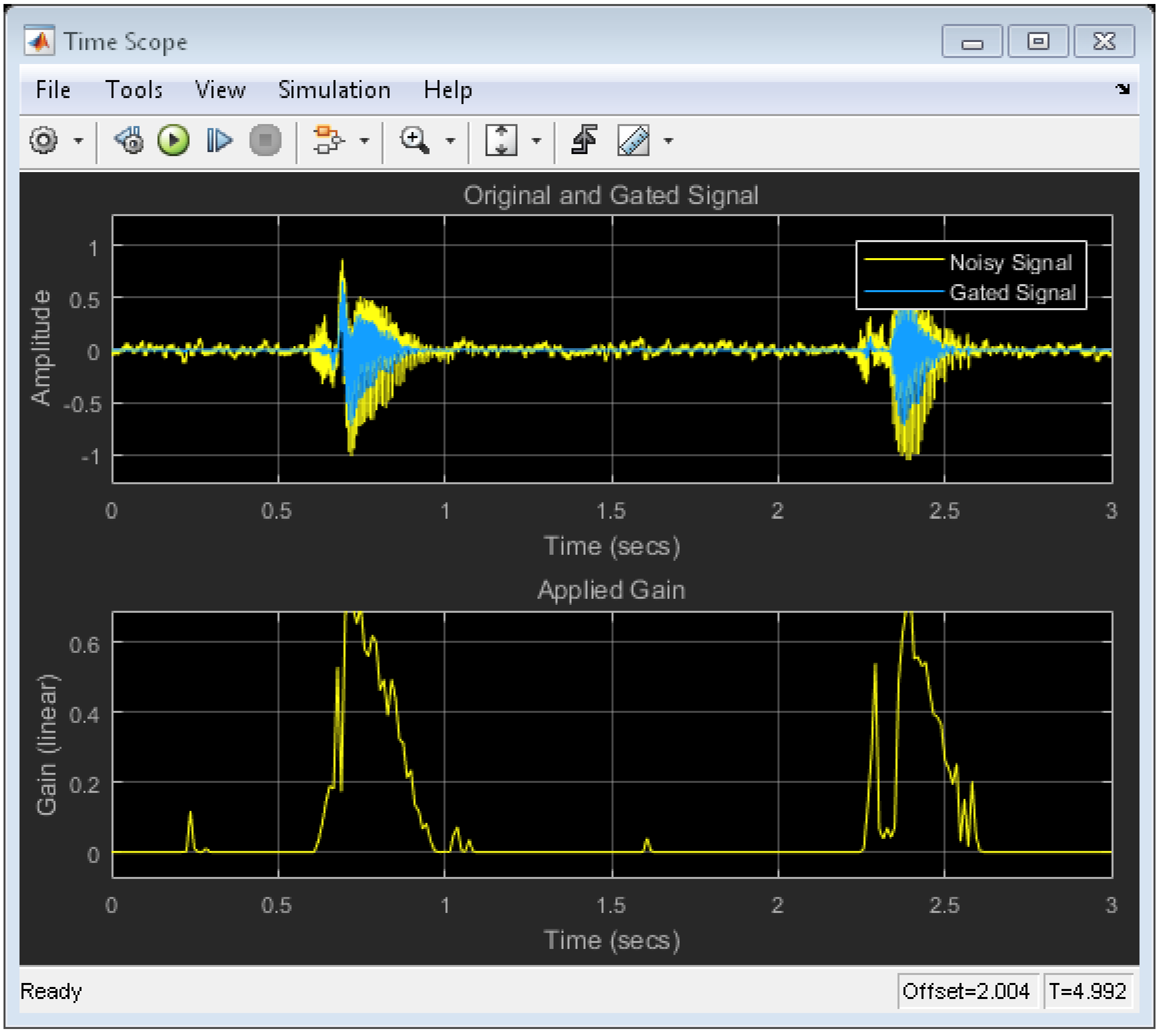
Apply dynamic range gating to remove low-level noise from an audio file.



1. Open the Time Scope and Noise Gate blocks.

2. Run the model. To switch between listening to the gated signal and the original signal, double-click the Manual Switch block.

3. Observe how the applied gain depends on noise gate parameters and input signal dynamics by tuning Noise Gate block parameters and viewing the results on the Time Scope.



## See Also

Audio Device Writer | From Multimedia File | Matrix Concatenate | Noise Gate | Random Source | Time Scope

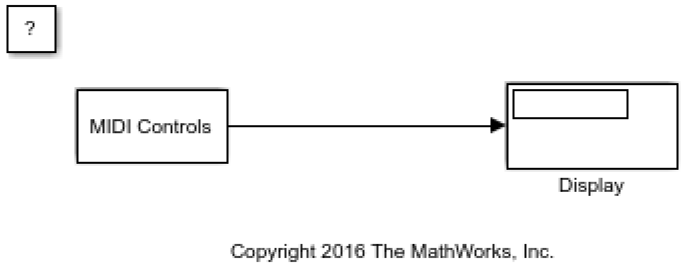
## More About

- “Dynamic Range Control” on page 1-2

## Output Values from MIDI Control Surface

The example shows how to set the MIDI Controls block parameters to output control values from your MIDI device.

1. Connect a MIDI device to your computer and then open the model.



2. Run the model with default settings. Move any controller on your default MIDI device to update the Display block.

3. Stop the simulation.

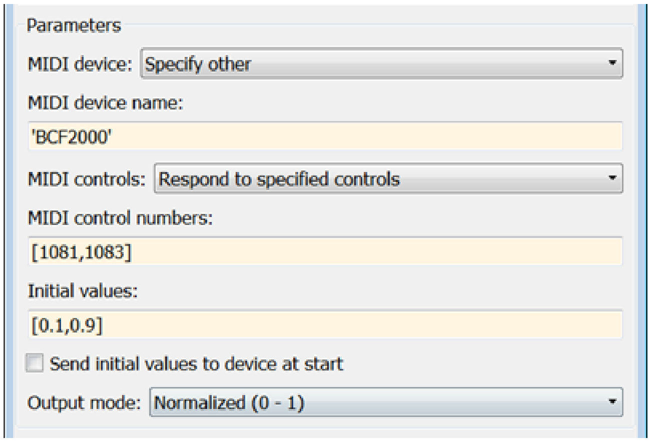
4. At the MATLAB™ command line, use `midiid` to determine the name of your MIDI device and two control numbers associated with your device.

5. In the MIDI Control block dialog box, set **MIDI device** to `Specify other` and enter the name of your MIDI device.

6. Set **MIDI controls** to `Respond to specified controls` and enter the control numbers determined using `midiid`.

7. Specify initial values as a vector the same size as **MIDI control numbers**. The initial values you specify are quantized according to the MIDI protocol and your particular MIDI surface.

The dialog box shows sample values for a `'BCF2000'` MIDI device with control numbers `1081` and `1083`.


Parameters

MIDI device: Specify other

MIDI device name:

'BCF2000'

MIDI controls: Respond to specified controls

MIDI control numbers:

[1081,1083]

Initial values:

[0.1,0.9]

☐ Send initial values to device at start

Output mode: Normalized (0 - 1)

8. Click **OK**, and then run the model. Verify that the Display block shows initial values and updates when you move the specified controls.

## See Also

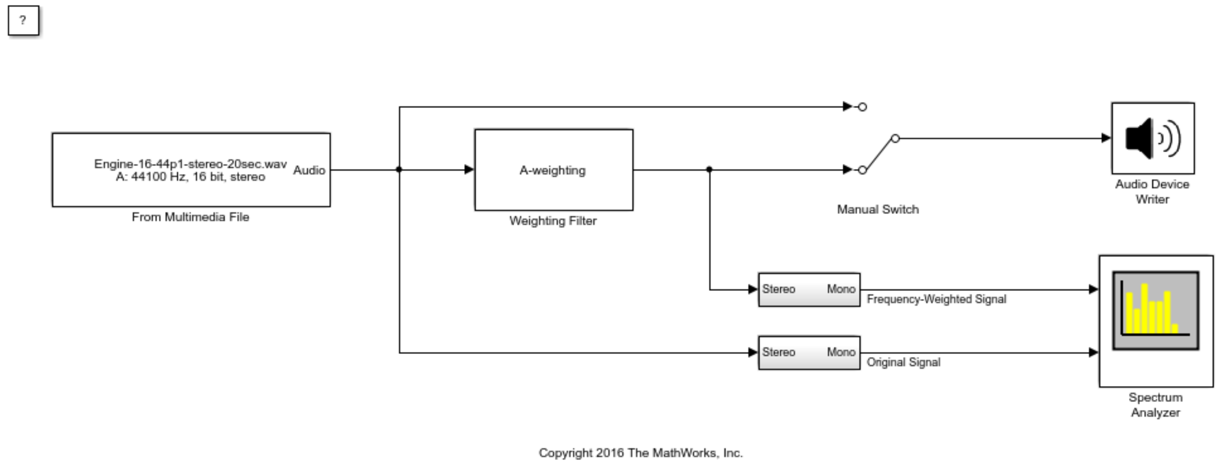
Audio Device Writer | From Multimedia File | Matrix Concatenate | MIDI Controls | Time Scope

## More About

- "Musical Instrument Digital Interface (MIDI)" on page 3-2

## Apply Frequency Weighting

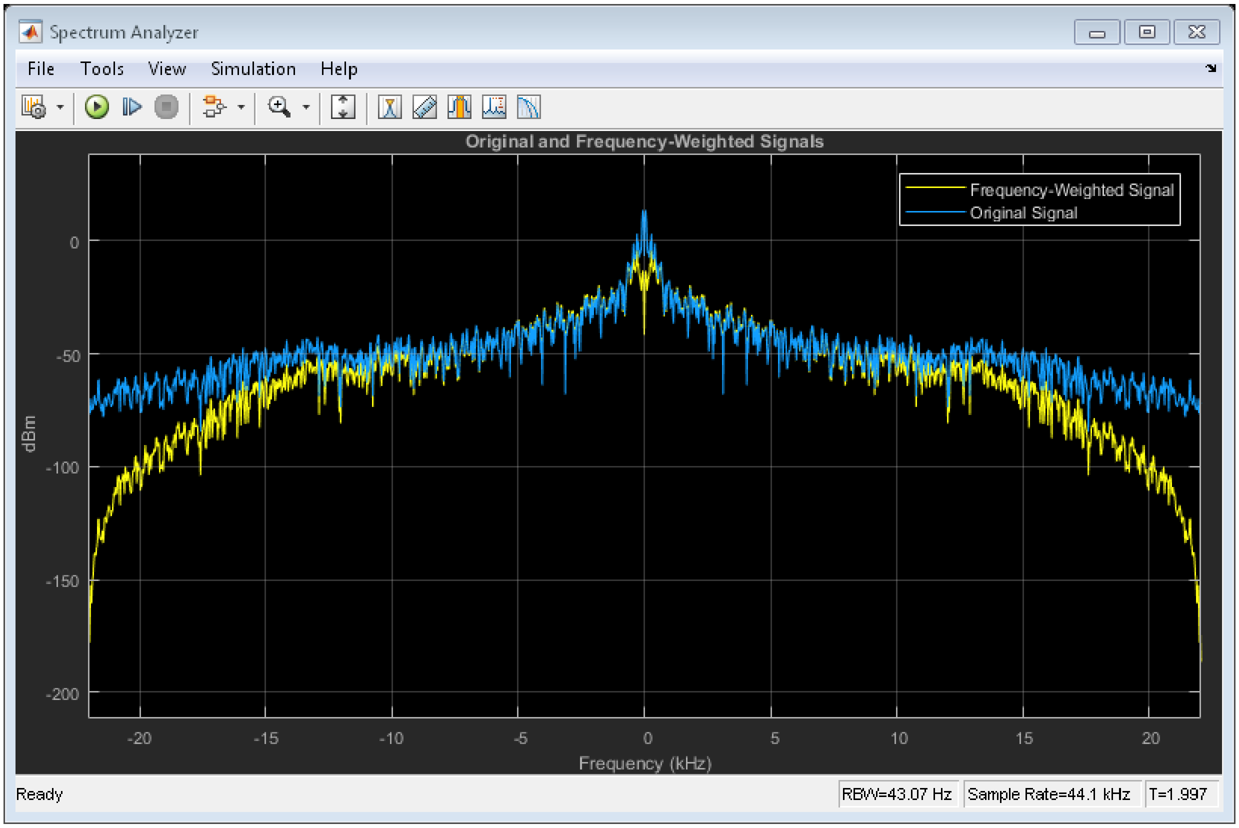
Examine the Weighting Filter block in a Simulink® model and tune parameters.



1. Open the Spectrum Analyzer block.

2. Run the model. Switch between listening to the frequency-weighted signal and the original signal by double-clicking the Manual Switch block.

3. Stop the model. Open the Weighting Filter block and choose a different weighting method. Observe the difference in simulation.

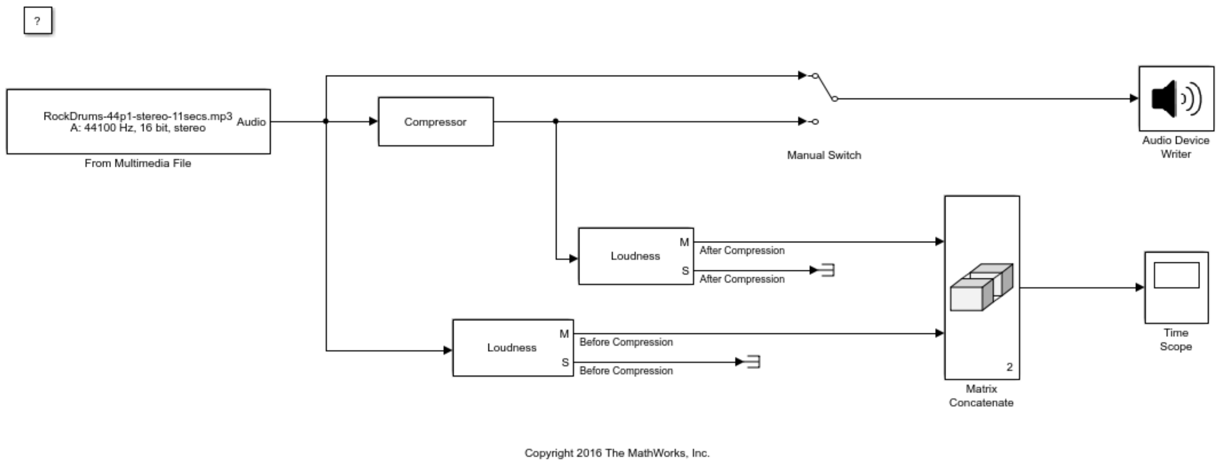


## See Also

Audio Device Writer | From Multimedia File | Spectrum Analyzer | Weighting Filter

## Compare Loudness Before and After Audio Processing

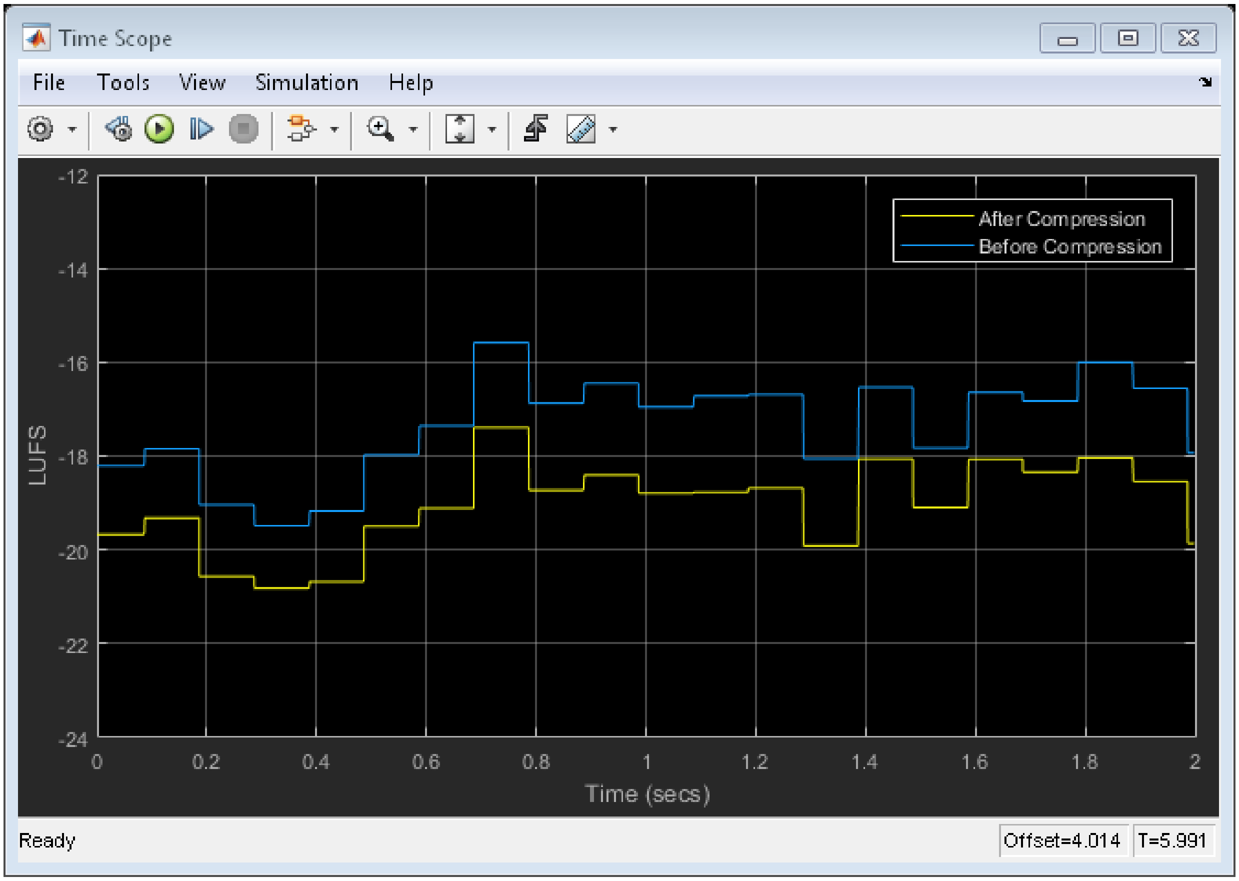
Measure momentary and short-term loudness before and after compression of a streaming audio signal in Simulink®.



1. Open the Time Scope and Compressor blocks.

2. Run the model. To switch between listening to the compressed signal and the original signal, double-click the switch.

3. Observe the effect of compression on loudness by tuning the Compressor block parameters and viewing the momentary loudness on the Time Scope block.



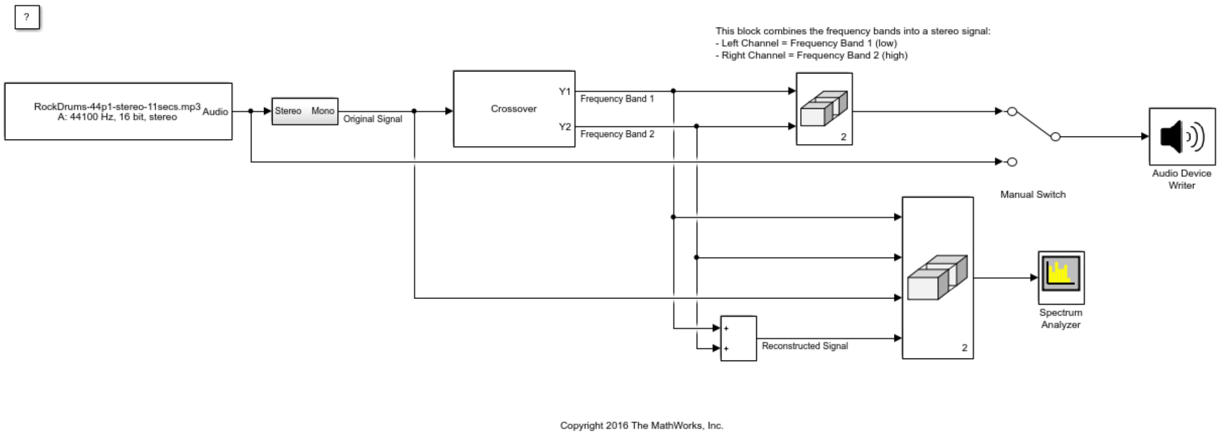
4. Stop the model. For both Loudness blocks, replace momentary loudness with short-term loudness as input to the Matrix Concatenate block. Run the model again and observe the effect of compression on short-term loudness.

## See Also

Audio Device Writer | Compressor | From Multimedia File | Loudness Meter | Matrix Concatenate | Time Scope

## Two-Band Crossover Filtering for a Stereo Speaker System

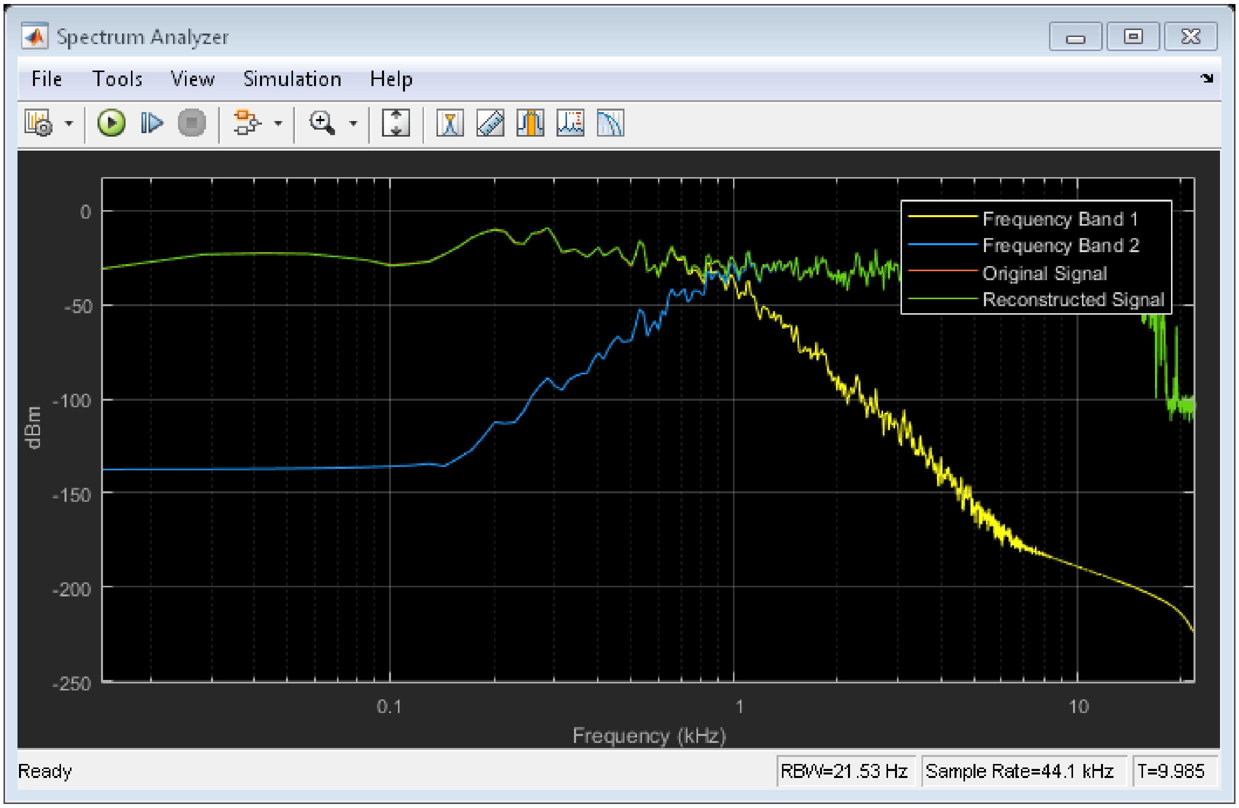
Divide a mono signal into a stereo signal with distinct frequency bands. To hear the full effect of this simulation, use a stereo speaker system, such as headphones.


?
This block combines the frequency bands into a stereo signal:
- Left Channel = Frequency Band 1 (low)
- Right Channel = Frequency Band 2 (high)
RockDrums-44p1-stereo-11secs.mp3
A: 44100 Hz, 16 bit, stereo
Audio
Stereo Mono
Original Signal
Crossover
Y1
Y2
Frequency Band 1
Frequency Band 2
2
Manual Switch
Audio Device Writer
Spectrum Analyzer
Reconstructed Signal
2
Copyright 2016 The MathWorks, Inc.

1. Open the Spectrum Analyzer and Crossover Filter blocks.

2. Run the model. To switch between listening to the filtered and original signal, double-click the Manual Switch block.

3. Tune the crossover frequency on the Crossover Filter block to listen to the effect on your speakers and view the effect on the Spectrum Analyzer block.

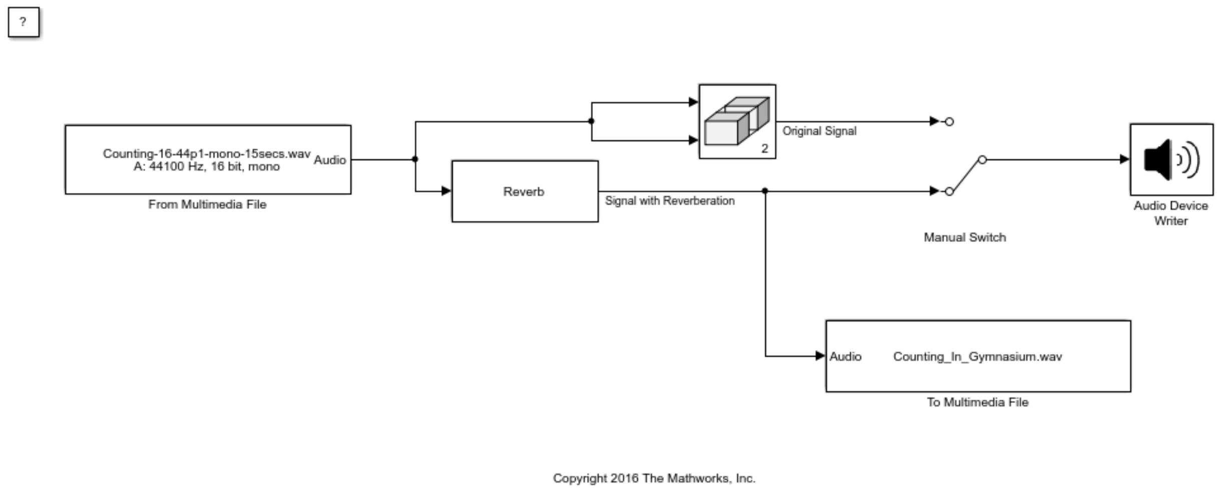


## See Also

Audio Device Writer | Crossover Filter | From Multimedia File | Matrix Concatenate | Spectrum Analyzer

## Mimic Acoustic Environments

Examine the Reverberator block in a Simulink® model and tune parameters. The reverberation parameters in this model mimic a large room with hard walls, such as a gymnasium.


?
Counting-16-44p1-mono-15secs.wav
A: 44100 Hz, 16 bit, mono
Audio
From Multimedia File
2
Original Signal
Reverb
Signal with Reverberation
Manual Switch
Audio Device Writer
Audio
Counting_In_Gymnasium.wav
To Multimedia File
Copyright 2016 The Mathworks, Inc.

1. Run the simulation. Listen to the audio signal with and without reverberation by double-clicking the Manual Switch block.

2. Stop the simulation.

3. Disconnect the From Multimedia File block so that you can run the model without recording.

4. Open the Reverberator block.

5. Run the simulation and tune the parameters of the Reverberator block.

6. After you are satisfied with the reverberation environment, stop the simulation.

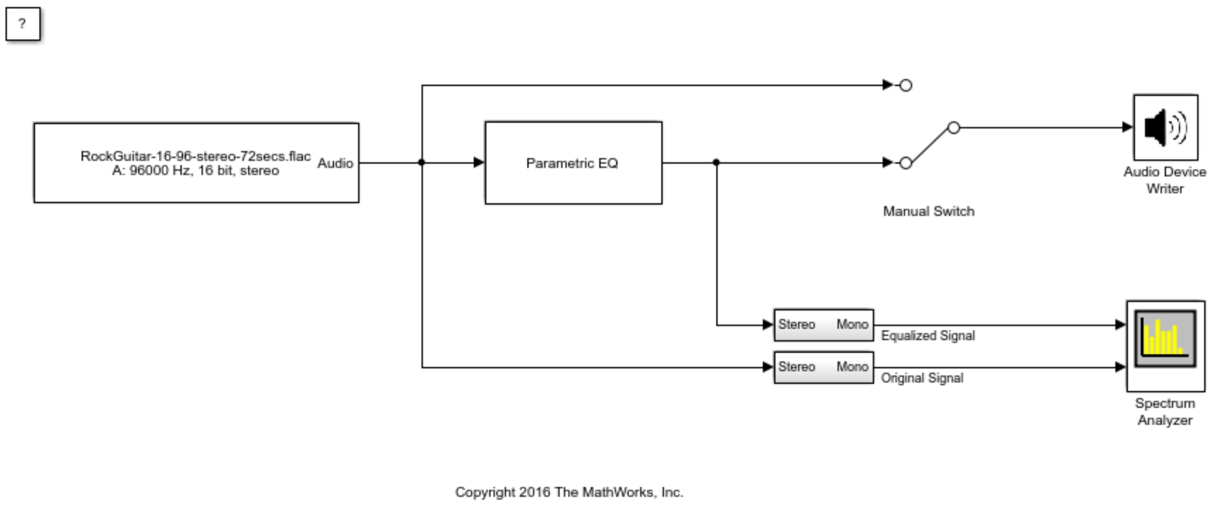
7. Reconnect the To Multimedia File block. Rename the output file with a description to match your reverberation environment, and rerun the model.

## See Also

Audio Device Writer | From Multimedia File | Matrix Concatenate | Reverberator | To Multimedia File

## Perform Parametric Equalization

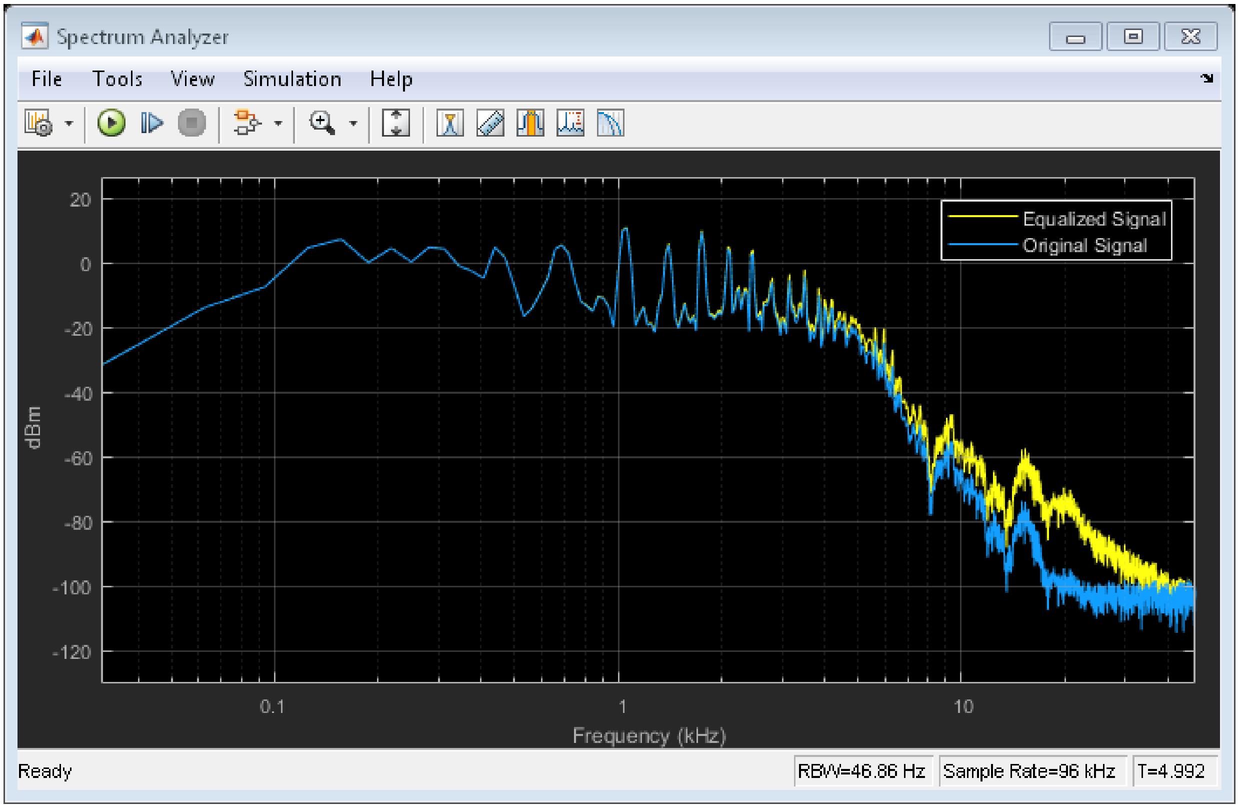
Examine the Parametric EQ Filter block in a Simulink® model and tune parameters.



1. Open the Spectrum Analyzer and Parametric EQ Filter blocks.

2. In the Parametric EQ Filter block, click **View Filter Response**. Modify parameters of the parametric equalizer and see the mangitude response plot update automatically.

3. Run the model. Tune parameters on the Parametric EQ Filter to listen to the effect on your audio device and see the effect on the Spectrum Analyzer display. Double-click the Manual Switch block to toggle between the original and equalized signal as output.

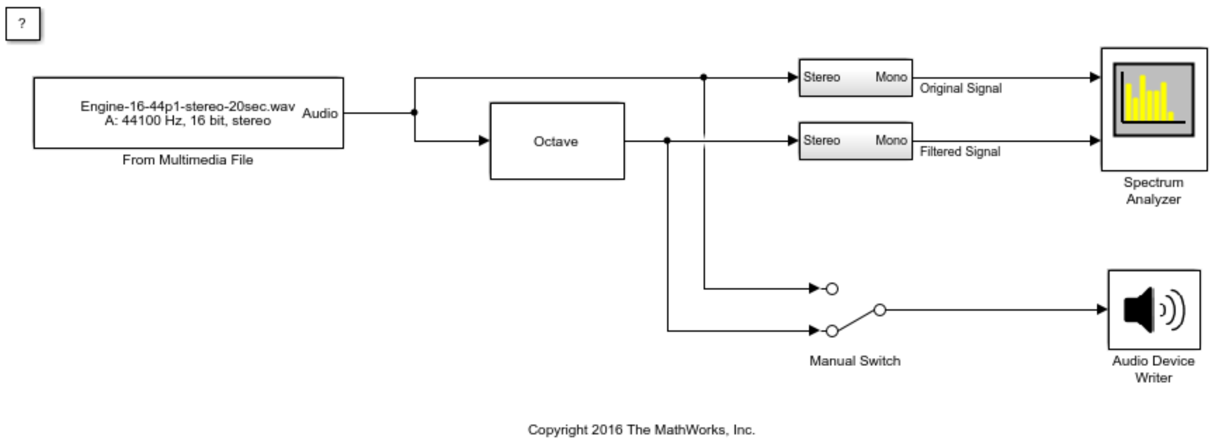


## See Also

Audio Device Writer | From Multimedia File | Matrix Concatenate | Parametric EQ Filter | Spectrum Analyzer

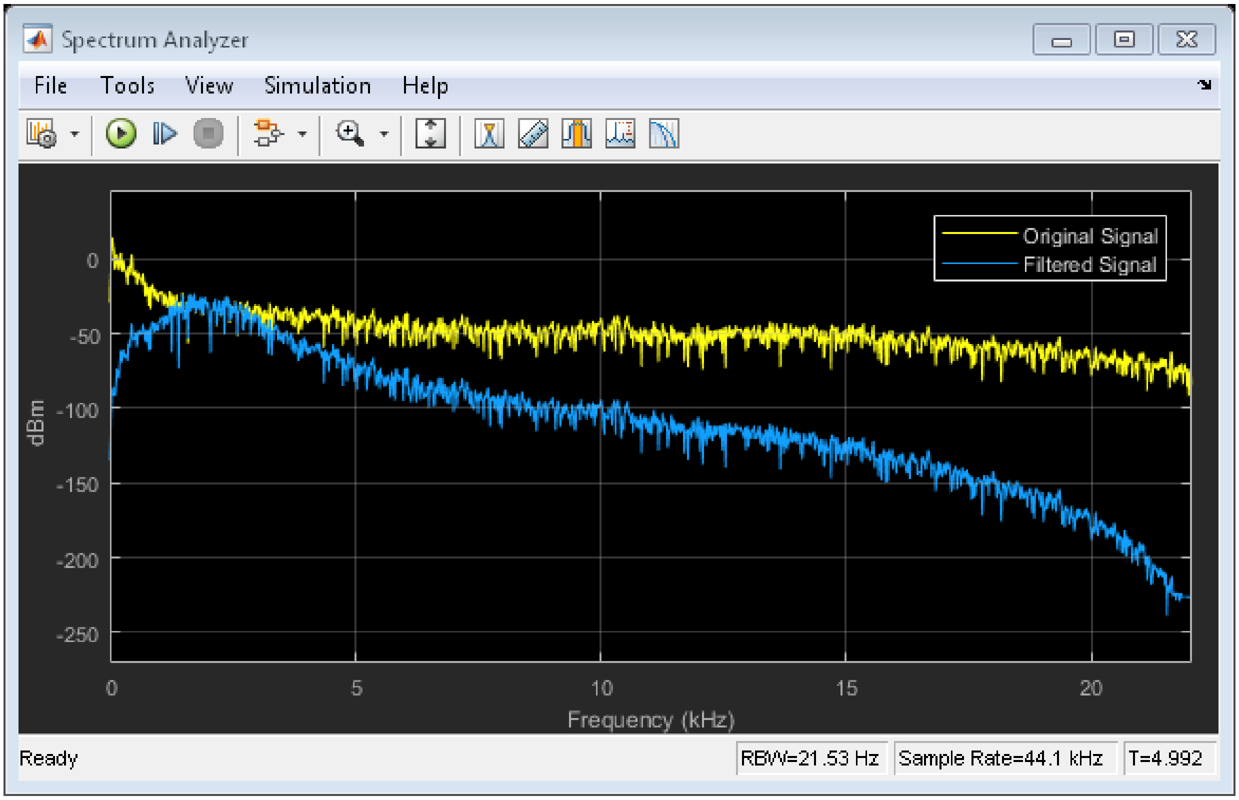
## Perform Octave Filtering

Examine the Octave Filter block in a Simulink® model and tune parameters.



1. Open the Octave Filter block and click **Visualize filter response**. Tune parameters on the Octave Filter dialog. The filter response visualization updates automatically. If you break compliance with the ANSI S1.11-2004 standard, the filter mask is drawn in red.

2. Run the model. Open the Spectrum Analyzer block. Tune parameters on the Octave Filter block to listen to the effect on your audio device and see the effect on the Spectrum Analyzer display. Switch between listening to the filtered and unfiltered audio by double-clicking the Manual Switch block.

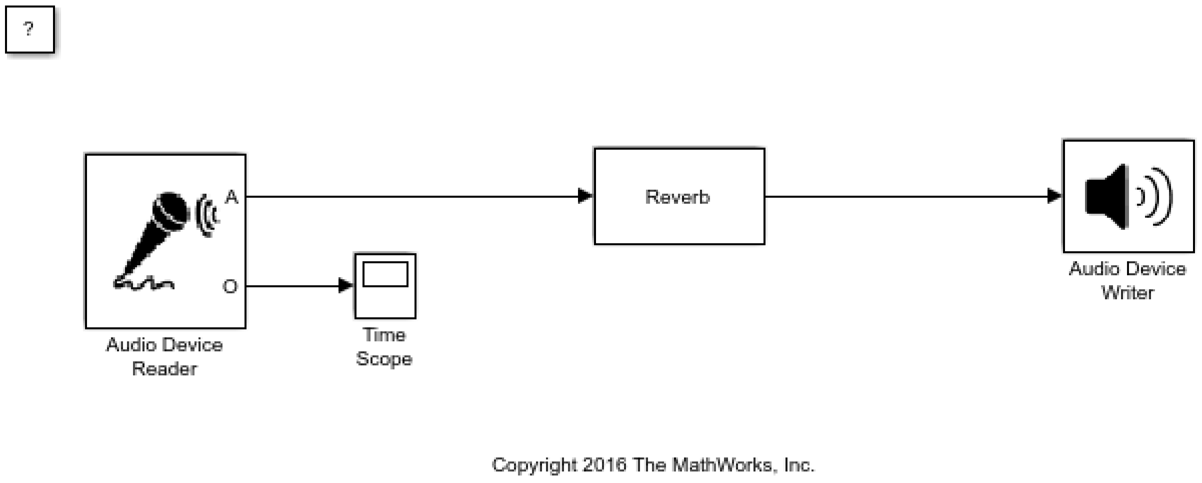


## See Also

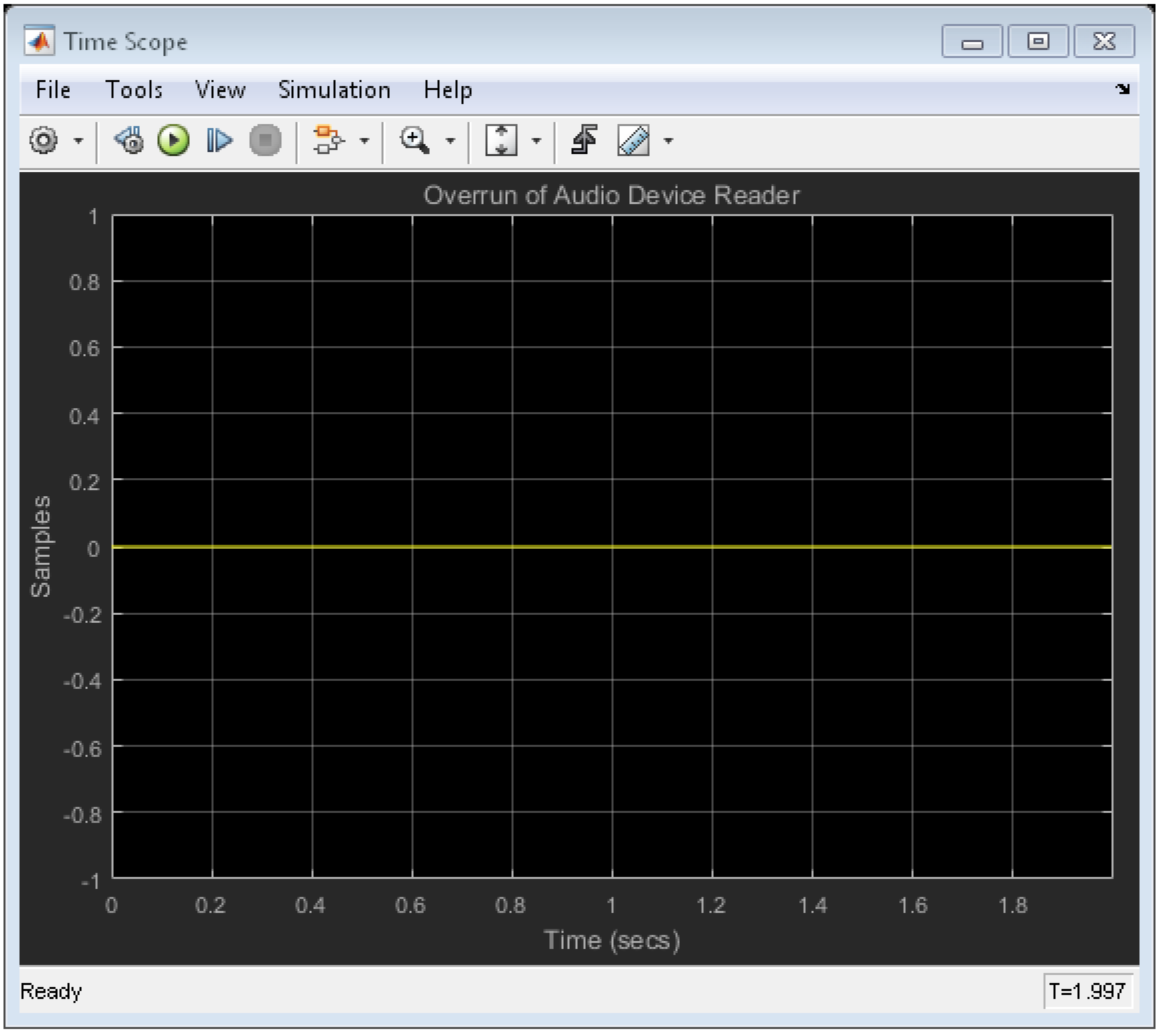
Audio Device Writer | From Multimedia File | Octave Filter | Spectrum Analyzer

## Read from Microphone and Write to Speaker

Examine the Audio Device Reader block in a Simulink® model, modify parameters, and explore overrun.



1. Run the model. The Audio Device Reader records an audio stream from your computer's default audio input device. The Reverberator block processes your input audio. The Audio Device Writer block sends the processed audio to your default audio output device.



2. Stop the model. Open the Audio Device Reader block and lower the **Samples per frame** parameter. Open the Time Scope block to view overrun.

3. Run the model again. Lowering the **Samples per frame** decreases the buffer size of your Audio Device Reader block. A smaller buffer size decreases audio latency while increasing the likelihood of overruns.

## See Also

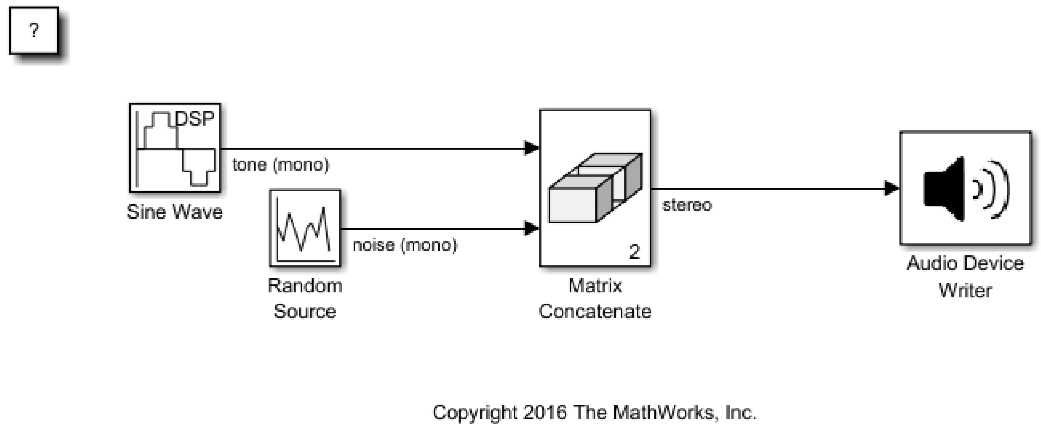
Audio Device Reader | Audio Device Writer | Reverberator | Time Scope

## More About

- “Audio I/O: Buffering, Latency, and Throughput”

## Channel Mapping

Examine the Audio Device Writer block in a Simulink® model and specify a nondefault channel mapping.



1. Run the simulation. The Audio Device Writer sends a stereo audio stream to your computer's default audio output device. If you are using a stereo audio output device, such as headphones, you can hear a tone from one speaker and noise from the other speaker.

2. Specify a nondefault channel mapping:

a. Stop the simulation.

b. Open the Audio Device Writer block to modify parameters.

c. On the **Advanced** tab, clear the **Use default mapping between columns of input of this block and sound card's output channels** parameter.

d. Specify the **Device output channels** in reverse order: `[2,1]`. If you are using a stereo output device, such as headphones, you hear that the noise and tone have switched speakers.

### See Also

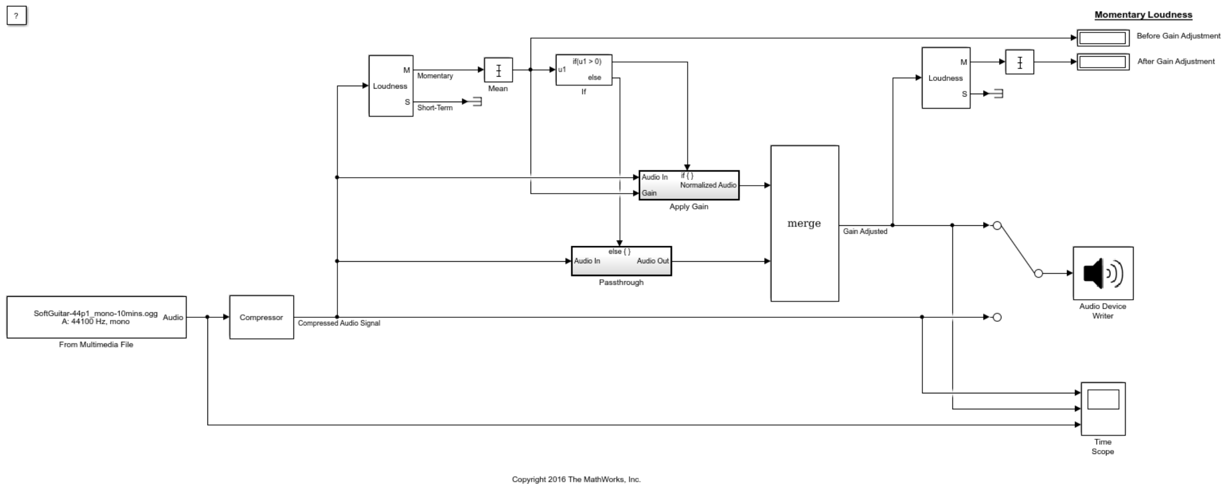
Audio Device Writer | Matrix Concatenate | Random Source | Sine Wave

## More About

- “Audio I/O: Buffering, Latency, and Throughput”

## Trigger Gain Control Based on Loudness Measurement

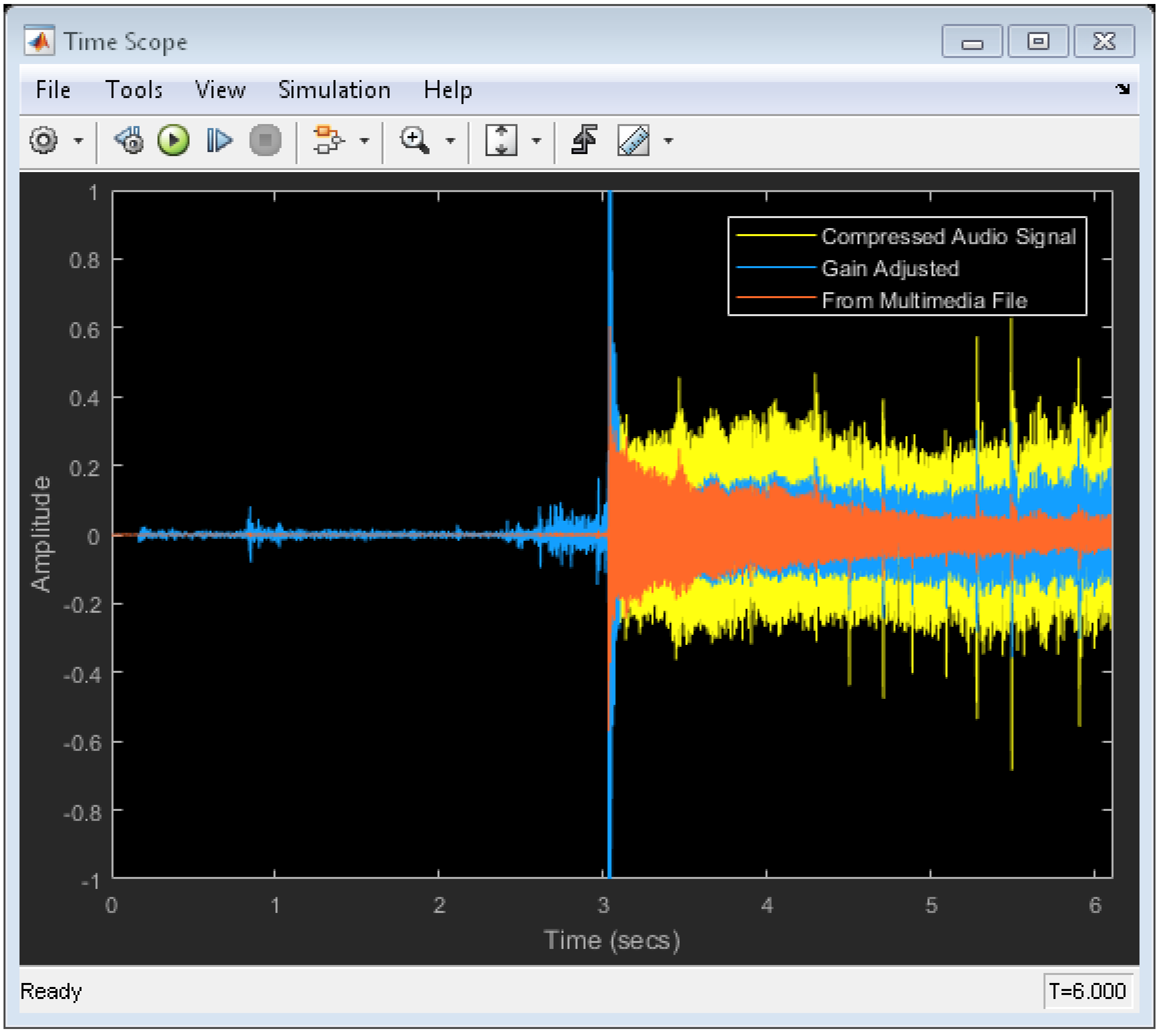
This model enables you to apply dynamic range compression to an audio signal while staying inside a preset loudness range. In this model, a Compressor block increases the loudness and decreases the dynamic range of an audio signal. A Loudness Meter block calculates the momentary loudness of the compressed audio signal. If momentary loudness crosses a -23 LUFS threshold, an enabled subsystem applies gain to lower the corresponding level of the audio signal.



1. Open the Time Scope and Compressor blocks.

2. Run the model. To switch between listening to the compressed signal with and without gain adjustment, double-click the switch.

3. To observe the effect of compression on loudness, tune the Compressor block parameters and view the compressed audio signal on the Time Scope block.



## See Also

**Blocks**
Audio Device Writer | Compressor | From Multimedia File | Loudness Meter | Time Scope

**System Objects**
loudnessMeter

**Functions**
`integratedLoudness`

## More About

- "Loudness Normalization in Accordance with EBU R 128 Standard"

# Real-Time Parameter Tuning

## Real-Time Parameter Tuning

*Parameter tuning* is the ability to modify parameters of your audio system in real time while streaming an audio signal. In algorithm development, tunable parameters enable you to quickly prototype and test various parameter configurations. In deployed applications, tunable parameters enable users to fine-tune general algorithms for specific purposes, and to react to changing dynamics.

Audio System Toolbox is optimized for parameter tuning in a real-time audio stream. The System objects, blocks, and audio plugins provide various tunable parameters, including sample rate and frame size, making them robust tools when used in an audio stream loop.

To optimize your use of Audio System Toolbox, package your audio processing algorithm as an audio plugin. Packaging your audio algorithm as an audio plugin enables you to prototype your algorithm using the Audio Test Bench. The **Audio Test Bench** creates a user interface (UI) for tunable parameters, enables you to specify input and output from your audio stream loop, and provides access to analysis tools such as the time scope and spectrum analyzer. Packaging your code as an audio plugin also enables you to quickly synchronize your parameters with MIDI controls. For more information, see “Design an Audio Plugin” and “Audio Test Bench Walkthrough” on page 4-2.

Other methods to create UIs in MATLAB include:

- App Designer — Development environment for a large set of interactive controls with support for 2-D plots. See “Create a Simple App Using App Designer” (MATLAB) for more information.
- GUIDE — Drag-and-drop environment for laying out user interfaces with support for any type of plot. See “Create a Simple App Using GUIDE” (MATLAB) for more information.
- Programmatic workflow — Use MATLAB functions to define your app element-by-element. This tutorial uses a programmatic approach.

See “Ways to Build Apps” (MATLAB) for a more detailed list of the costs and benefits of the different approaches to parameter tuning.

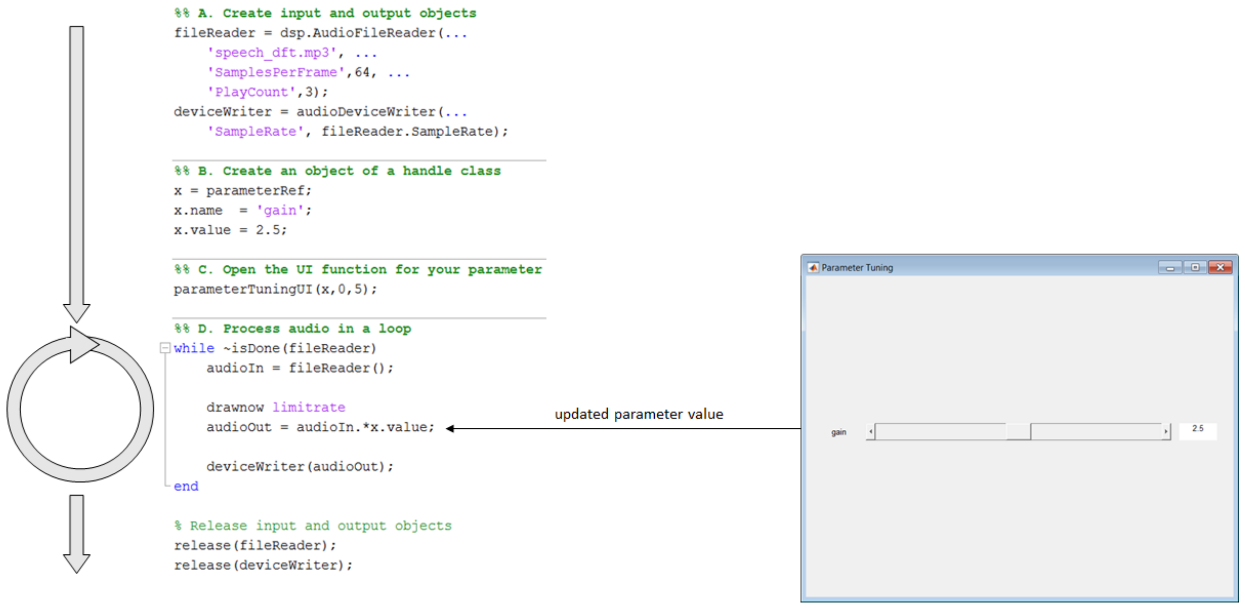
### Programmatic Parameter Tuning

In this tutorial, you tune the value of a parameter in an audio stream loop.

This tutorial contains three files:

1 `parameterRef` — Class definition that contains tunable parameters
2 `parameterTuningUI` — Function that creates a UI for parameter tuning
3 `AudioProcessingScript` — Script for audio processing

Inspect the diagram for an overview of how real-time parameter tuning is implemented. To implement real-time parameter tuning, walk through the example for explanations and step-by-step instructions.



```
%% A. Create input and output objects
fileReader = dsp.AudioFileReader(...
    'speech_dft.mp3', ...
    'SamplesPerFrame',64, ...
    'PlayCount',3);
deviceWriter = audioDeviceWriter(...
    'SampleRate', fileReader.SampleRate);

%% B. Create an object of a handle class
x = parameterRef;
x.name  = 'gain';
x.value = 2.5;

%% C. Open the UI function for your parameter
parameterTuningUI(x,0,5);

%% D. Process audio in a loop
while ~isDone(fileReader)
    audioIn = fileReader();

    drawnow limitrate
    audioOut = audioIn.*x.value;

    deviceWriter(audioOut);
end

% Release input and output objects
release(fileReader);
release(deviceWriter);
```

## 1. Create Class with Tunable Parameters

To tune a parameter in an audio stream loop using a UI, you need to associate the parameter with the position of a UI widget. To associate a parameter with a UI widget, make the parameter an object of a handle class. Objects of handle classes are passed by reference, meaning that you can modify the value of the object in one place and use the updated value in another. For example, you can modify the value of the object using a slider on a figure and use the updated value in an audio processing loop.

Save the `parameterRef` class definition file to your current folder.

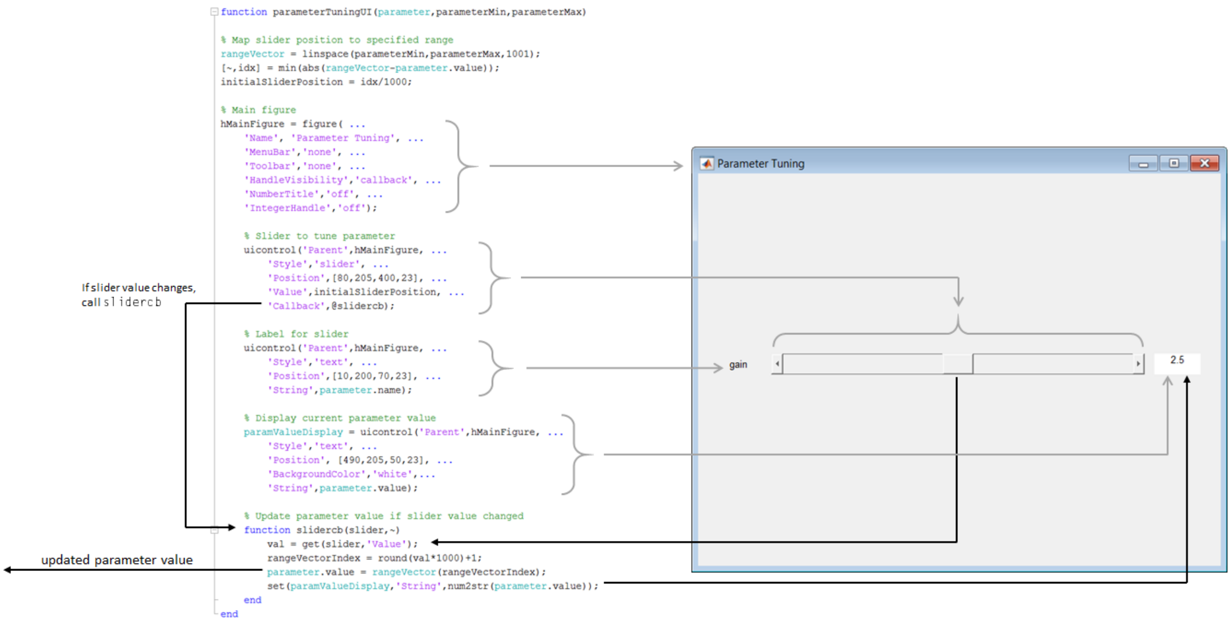
```
classdef parameterRef < handle
    properties
        name
```

```
            value
        end
    end
```

Objects of the `parameterRef` class have a `name` and `value`. The `name` is for display purposes on the UI. You use the `value` for tuning.

### 2. Create Function to Generate a UI

The `parameterTuningUI` function accepts your parameter, specified as an object handle, and the desired range. The function creates a figure with a slider associated with your parameter. The nested function, `slidercb`, is called whenever the slider position changes. The slider callback function maps the position of the slider to the parameter range, updates the value of the parameter, and updates the text on the UI. You can easily modify this function to tune multiple parameters in the same UI.



```
function parameterTuningUI(parameter,parameterMin,parameterMax)

% Map slider position to specified range
rangeVector = linspace(parameterMin,parameterMax,1001);
[~,idx] = min(abs(rangeVector-parameter.value));
initialSliderPosition = idx/1000;

% Main figure
hMainFigure = figure( ...
    'Name', 'Parameter Tuning', ...
    'MenuBar','none', ...
    'Toolbar','none', ...
    'HandleVisibility','callback', ...
    'NumberTitle','off', ...
    'IntegerHandle','off');

    % Slider to tune parameter
    uicontrol('Parent',hMainFigure, ...
        'Style','slider', ...
        'Position',[80,205,400,23], ...
        'Value',initialSliderPosition, ...
        'Callback',@slidercb);

    % Label for slider
    uicontrol('Parent',hMainFigure, ...
        'Style','text', ...
        'Position',[10,200,70,23], ...
        'String',parameter.name);

    % Display current parameter value
    paramValueDisplay = uicontrol('Parent',hMainFigure, ...
        'Style','text', ...
        'Position', [490,205,50,23], ...
        'BackgroundColor','white',...
        'String',parameter.value);

    % Update parameter value if slider value changed
    function slidercb(slider,~)
        val = get(slider,'Value');
        rangeVectorIndex = round(val*1000)+1;
        parameter.value = rangeVector(rangeVectorIndex);
        set(paramValueDisplay,'String',num2str(parameter.value));
    end
end
```

### Save `parameterTuningUI` to Current Folder

Open `parameterTuningUI` and save it to your current folder.

```
function parameterTuningUI(parameter,parameterMin,parameterMax)
```

```
% Map slider position to specified range
rangeVector = linspace(parameterMin,parameterMax,1001);
[~,idx] = min(abs(rangeVector-parameter.value));
initialSliderPosition = idx/1000;

% Main figure
hMainFigure = figure( ...
    'Name', 'Parameter Tuning', ...
    'MenuBar','none', ...
    'Toolbar','none', ...
    'HandleVisibility','callback', ...
    'NumberTitle','off', ...
    'IntegerHandle','off');

    % Slider to tune parameter
    uicontrol('Parent',hMainFigure, ...
        'Style','slider', ...
        'Position',[80,205,400,23], ...
        'Value',initialSliderPosition, ...
        'Callback',@slidercb);

    % Label for slider
    uicontrol('Parent',hMainFigure, ...
        'Style','text', ...
        'Position',[10,200,70,23], ...
        'String',parameter.name);

    % Display current parameter value
    paramValueDisplay = uicontrol('Parent',hMainFigure, ...
        'Style','text', ...
        'Position', [490,205,50,23], ...
        'BackgroundColor','white', ...
        'String',parameter.value);

    % Update parameter value if slider value changed
    function slidercb(slider,~)
        val = get(slider,'Value');
        rangeVectorIndex = round(val*1000)+1;
        parameter.value = rangeVector(rangeVectorIndex);
        set(paramValueDisplay,'String',num2str(parameter.value));
    end
end
```

### 3. Create Script for Audio Processing

The audio processing script:

**A** Creates input and output objects for an audio stream loop.

**B** Creates an object of the handle class, `parameterRef`, that stores your parameter name and value.

**C** Calls the tuning UI function, `parameterTuningUI`, with your parameter and the parameter range.

**D** Processes the audio in a loop. You can tune your parameter, `x`, in the audio stream loop.

#### Run `AudioProcessingScript`

Open `AudioProcessingScript`, save it to your current folder, and then run the file.

```
%% A. Create input and output objects
fileReader = dsp.AudioFileReader(...
    'speech_dft.mp3', ...
    'SamplesPerFrame',64, ...
    'PlayCount',3);
deviceWriter = audioDeviceWriter(...
    'SampleRate', fileReader.SampleRate);

%% B. Create an object of a handle class
x = parameterRef;
x.name  = 'gain';
x.value = 2.5;

%% C. Open the UI function for your parameter
parameterTuningUI(x,0,5);

%% D. Process audio in a loop
while ~isDone(fileReader)
    audioIn = fileReader();

    drawnow limitrate
    audioOut = audioIn.*x.value;

    deviceWriter(audioOut);
end

% Release input and output objects
```

```
release(fileReader);
release(deviceWriter);
```

While the script runs, move the position of the slider to update your parameter value and hear the result.

## See Also

Audio Test Bench

## More About

- “Real-Time Audio in MATLAB”
- “Design an Audio Plugin”
- “Audio Test Bench Walkthrough” on page 4-2
- “Create a Simple App Using App Designer” (MATLAB)
- “Create a Simple App Using GUIDE” (MATLAB)
- “Ways to Build Apps” (MATLAB)

# Sample Audio Files

# Sample Audio Files

Use these audio files as input to your audio system.

## See Also

**Functions**
`audioread`

**System Objects**
dsp.AudioFileReader

**Blocks**
From Multimedia File

## More About

- “Audio I/O: Buffering, Latency, and Throughput”

13

# Tips and Tricks for Plugin Authoring

# Tips and Tricks for Plugin Authoring

To author your algorithm as an audio plugin, you must conform to the audio plugin API. When authoring audio plugins in the MATLAB environment, keep these common pitfalls and best practices in mind.

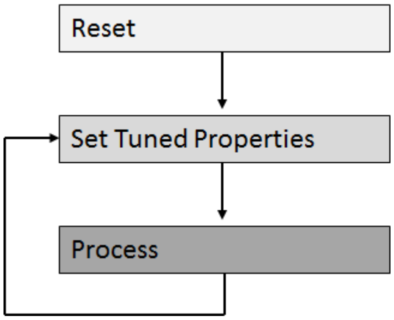
To learn more about audio plugins in general, see “Design an Audio Plugin”.

## Avoid Disrupting the Event Queue in MATLAB

When the Audio Test Bench runs an audio plugin, it sequentially:

1. Calls the reset method
2. Sets tunable properties associated with parameters
3. Calls the process method

While running, the **Audio Test Bench** calls in a loop the process method and then the `set` methods for tuned properties. The plugin API does not specify the order that the tuned properties are set.



It is possible to disrupt the normal methods timing by interrupting the event queue. Common ways to accidently interrupt the event queue include using a `plot` or `drawnow` function.

**Note:** `plot` and `drawnow` are only available in the MATLAB environment. `plot` and `drawnow` cannot be included in generated plugins. See “Separate Code for Features Not Supported for Plugin Generation” on page 13-6 for more information.

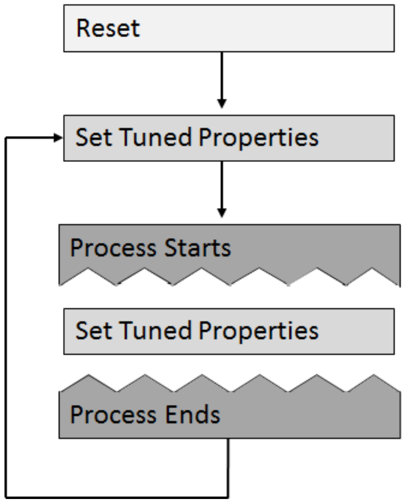
In the following code snippet, the gain applied to the left and right channels is not the same if the associated `Gain` parameter is tuned during the call to process:

```
...
L = plugin.Gain*in(:,1);
drawnow
R = plugin.Gain*in(:,2);
out = [L,R];
...
```

**See Full Code**

```
classdef badPlugin < audioPlugin
    properties
        Gain = 0.5;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)

            L = plugin.Gain*in(:,1);

            drawnow

            R = plugin.Gain*in(:,2);

            out = [L,R];
        end
        function set.Gain(plugin,val)
            plugin.Gain = val;
        end
    end
end
```

The author interrupts the event queue in the code snippet, causing the `set` methods of properties associated with parameters to be called while the process method is in the middle of execution.



Depending on your processing algorithm, interrupting the event queue can lead to inconsistent and buggy behavior. Also, the `set` method might not be explicit, which can make the issue difficult to track down. Possible fixes for the problem of event queue disruption include saving properties to local variables, and moving the queue disruption to the beginning or end of the process method.

### Save Properties to Local Variables

You can save tunable property values to local variables at the start of your processing. This technique ensures that the values used during the process method are not updated within a single call to process. Because accessing the value of a local variable is cheaper than accessing the value of a property, saving properties to local variables that are accessed multiple times is a best practice.

```
...
gain = plugin.Gain;
L = gain*in(:,1);
drawnow
R = gain*in(:,2);
out = [L,R];
...
```

### See Full Code

```
classdef goodPlugin < audioPlugin
    properties
```

```
        Gain = 0.5;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            gain = plugin.Gain;

            L = gain*in(:,1);

            drawnow

            R = gain*in(:,2);

            out = [L,R];
        end
        function set.Gain(plugin,val)
            plugin.Gain = val;
        end
    end
end
```

### Move Queue Disruption to Bottom or Top of Process Method

You can move the disruption to the event queue to the bottom or top of the process method. This technique ensures that property values are not updated in the middle of the call.

```
...
L = plugin.Gain*in(:,1);
R = plugin.Gain*in(:,2);
out = [L,R];
drawnow
...
```

### See Full Code

```
classdef goodPlugin < audioPlugin
    properties
        Gain = 0.5;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)

            L = plugin.Gain*in(:,1);

            R = plugin.Gain*in(:,2);

            out = [L,R];

            drawnow
        end
        function set.Gain(plugin,val)
            plugin.Gain = val;
        end
    end
end
```

## Separate Code for Features Not Supported for Plugin Generation

The MATLAB environment offers functionality not supported for plugin generation. You can mark code to ignore during plugin generation by placing it inside a conditional statement by using `coder.target`.

```
...
    if coder.target('MATLAB')
    ...
    end
...
```

If you generate the plugin using `generateAudioPlugin`, code inside the statement `if coder.target('MATLAB')` is ignored.

For example, dsp.TimeScope is not enabled for code generation. If you run the following plugin in MATLAB, you can use the visualize function to open a time scope that plots the input and output power per frame.

### See Full Example Code

```
classdef  pluginWithMATLABOnlyFeatures < audioPlugin
    properties
        Threshold = -10;
    end
    properties (Access = private)
        aCompressor
        aScope
        SamplesPerFrame = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Threshold','Mapping',{'lin',-60,20}));
    end
    methods
        function plugin = pluginWithMATLABOnlyFeatures
            plugin.aCompressor = compressor;
            setup(plugin.aCompressor,[0,0])
        end
        function out = process(plugin,in)
            out = plugin.aCompressor(in);

            % The contents of this if-statement are ignored during plugin
            % generation.
            if coder.target('MATLAB')
                if ~isempty(plugin.aScope) && isvalid(plugin.aScope)
                    numSamples = size(in,1);

                    % The time scope object is not enabled for
                    % variable-size signals. Call release if the samples
                    % per frame is changed.
                    % Because this code is intended for use in MATLAB only, it
                    % is okay to call release on the time scope object. Do
                    % not call release on a System object in
                    % generated code.
                    if plugin.SamplesPerFrame(1) ~= numSamples
                        release(plugin.aScope)
                        plugin.SamplesPerFrame = numSamples;
                    end

                    power = 20*log10(mean(var(in)))*ones(numSamples,1);
                    adjustedPower = 20*log10(mean(var(out)))*ones(numSamples,1);
                    plugin.aScope([power,adjustedPower]);
                end
            end
        end
        function reset(plugin)
            fs = getSampleRate(plugin);
            plugin.aCompressor.SampleRate = fs;
            reset(plugin.aCompressor)

            % The contents of this if-statement are ignored during plugin
```

```
                % genderation.
                if coder.target('MATLAB')
                    if ~isempty(plugin.aScope)
                        % Because this code is intended for use in MATLAB only, it
                        % is okay to call release on the time scope object. Do
                        % not call release on a System object in
                        % generated code.
                        release(plugin.aScope)
                        plugin.aScope.SampleRate = fs;
                        plugin.aScope.BufferLength = 2*fs;
                    end
                end
            end
            function visualize(plugin)
                % Visualization function. This function is public in the MATLAB
                % environment. Because the plugin does not call this function directly, the
                % by generateAudioPlugin.

                % Create a time scope object for visualization in the MATLAB
                % environment.
                plugin.aScope = dsp.TimeScope( ...
                    'SampleRate',getSampleRate(plugin), ...
                    'TimeSpan',1, ...
                    'YLimits',[-40,0], ...
                    'BufferLength',2*getSampleRate(plugin), ...
                    'TimeSpanOverrunAction','Scroll', ...
                    'YLabel','Power (dB)');
                show(plugin.aScope)
            end
            function set.Threshold(plugin,val)
                plugin.Threshold = val;
                plugin.aCompressor.Threshold = val;
            end
        end
    end
```

## Implement Reset Correctly

A common error in audio plugin authoring is misusing the reset method. Valid uses of the reset method include:

- Clearing state
- Passing down calls to reset to component objects

- Updating properties which depend on sample rate

Invalid use of the reset method includes setting the value of any properties associated with parameters. Do not use your reset method to set properties associated with parameters to their initial conditions. Directly setting a property associated with a parameter causes the property to be out of sync with the parameter. For example, the following plugin is an example of incorrect use of the reset method.

```
classdef badReset < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
        function reset(plugin) % <-- Incorrect use of reset method.
            plugin.Gain = 1;   % <-- Never set values of a property that is
        end                    %     associated with a plugin parameter.
    end
end
```

## Implement Plugin Composition Correctly

If your plugin is composed of other plugins, then you must pass down the sample rate and calls to reset to the component plugins. Call setSampleRate in the reset method to pass down the sample rate to the component plugins. To tune parameters of the component plugins, create an audio plugin interface in the composite plugin for tunable parameters of the component plugins. Then pass down the values in the `set` methods for the associated properties. The following is an example of plugin composition that was constructed using best practices.

### Plugin Composition Using Basic Plugins

```
classdef compositePlugin < audioPlugin
    properties
        PhaserQ  = 1.6;
        EchoGain = 0.5;
    end
    properties (Access = private)
```

```
        aEcho
        aPhaser
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('PhaserQ', ...
                'DisplayName','Phaser Q', ...
                'Mapping',{'lin',0.5, 25}), ...
            audioPluginParameter('EchoGain', ...
                'DisplayName','Gain'));
    end
    methods
        function plugin = compositePlugin
            % Construct your component plugins in the composite plugin's
            % constructor.
            plugin.aPhaser = audiopluginexample.Phaser;
            plugin.aEcho   = audiopluginexample.Echo;
        end
        function out = process(plugin,in)
            % Call the process method of your component plugins inside the
            % call to the process method of your composite plugin.
            x = process(plugin.aPhaser,in);
            y = process(plugin.aEcho,x);
            out = y;
        end
        function reset(plugin)
            % Use the setSampleRate method to set the sample rate of
            % component plugins and pass the call to reset down.
            fs = getSampleRate(plugin);

            setSampleRate(plugin.aPhaser, fs)
            setSampleRate(plugin.aEcho, fs)

            reset(plugin.aPhaser)
            reset(plugin.aEcho);
        end
        % Use the set method of your properties to pass down property
        % values to your component plugins.
        function set.PhaserQ(plugin,val)
            plugin.PhaserQ = val;
            plugin.aPhaser.QualityFactor = val;
        end
        function set.EchoGain(plugin,val)
            plugin.EchoGain = val;
            plugin.aEcho.Gain = val;
        end
    end
end
```

Plugin composition using System objects has these key differences from plugin composition using basic plugins.

- Immediately call `setup` on your component System object after it is constructed. Construction and setup of the component object occurs inside the constructor of the composite plugin.
- If your component System object requires sample rate information, then it has a sample rate property. Set the sample rate property in the reset method.

### Plugin Composition Using System Objects

```
classdef compositePluginWithSystemObjects < audioPlugin
    properties
        CrossoverFrequency  = 100;
        CompressorThreshold = -40;
    end
    properties (Access = private)
        aCrossoverFilter
        aCompressor
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('CrossoverFrequency', ...
                'DisplayName','Crossover Frequency', ...
                'Mapping',{'lin',50, 200}), ...
            audioPluginParameter('CompressorThreshold', ...
                'DisplayName','Compressor Threshold', ...
                'Mapping',{'lin',-100,0}));
    end
    methods
        function plugin = compositePluginWithSystemObjects
            % Construct your component System objects within the composite
            % plugin's constructor. Call setup immediately after
            % construction.
            %
            % The audio plugin API requires plugins to declare the number
            % of input and output channels in the plugin interface. This
            % plugin uses the default 2-in 2-out configuration. Call setup
            % with a sample input that has the same number of channels as
```

```
                % defined in the plugin interface.
                %
                sampleInput = zeros(1,2);

                plugin.aCrossoverFilter = crossoverFilter;
                setup(plugin.aCrossoverFilter,sampleInput)

                plugin.aCompressor = compressor;
                setup(plugin.aCompressor,sampleInput)
            end
            function out = process(plugin,in)
                % Call your component System objects inside the call to
                % process of your composite plugin.
                [band1,band2] = plugin.aCrossoverFilter(in);
                band1Compressed = plugin.aCompressor(band1);
                out = band1Compressed + band2;
            end
            function reset(plugin)
                % Set the sample rate properties of your component System
                % objects.
                fs = getSampleRate(plugin);

                plugin.aCrossoverFilter.SampleRate = fs;
                plugin.aCompressor.SampleRate = fs;

                reset(plugin.aCrossoverFilter)
                reset(plugin.aCompressor);
            end
            % Use the set method of your properties to pass down property
            % values to your component System objects.
            function set.CrossoverFrequency(plugin,val)
                plugin.CrossoverFrequency = val;
                plugin.aCrossoverFilter.CrossoverFrequencies = val;
            end
            function set.CompressorThreshold(plugin,val)
                plugin.CompressorThreshold = val;
                plugin.aCompressor.Threshold = val;
            end
        end
    end
```

## Address "A set method for a non-Dependent property should not access another property" Warning in Plugin

It is recommended that you suppress the warning when authoring audio plugins.

The following code snippet follows the plugin authoring best practice for processing changes in parameter property `Cutoff`.

```
...
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Cutoff', ...
            'Label','Hz',...
            'Mapping',{'log',20,2000}));
    end
    methods
        function y = process(plugin,x)
            [y,plugin.State] = filter(plugin.B,plugin.A,x,plugin.State);
        end

        function set.Cutoff(plugin,val)
            plugin.Cutoff = val;
            [plugin.B,plugin.A] = highpassCoeffs(plugin,val,getSampleRate(plugin)); % <
        end
    end
...
```

### See Full Code Example

```
classdef highpassFilter < audioPlugin
    %------------------------------------------------------------------------
    % Public Properties - End user interacts with these
    %------------------------------------------------------------------------
    properties
        Cutoff = 20;
    end

    %------------------------------------------------------------------------
    % Private Properties - Used for internal storage
    %------------------------------------------------------------------------
    properties (Access = private)
        State = zeros(2);
        B     = zeros(1,3);
        A     = zeros(1,3);
    end

    %-----------------------------------------------------------------------
    % Constant Properties - Used to define plugin interface
    %-----------------------------------------------------------------------
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Cutoff', ...
            'Label','Hz', ...
            'Mapping',{'log',20,2000}));
    end

    methods
        %-------------------------------------------------------------------
        % Main processing function
        %-------------------------------------------------------------------
        function y = process(plugin,x)
            [y,plugin.State] = filter(plugin.B,plugin.A,x,plugin.State);
        end

        %-------------------------------------------------------------------
        % Set Method
        %-------------------------------------------------------------------
        function set.Cutoff(plugin,val)
            plugin.Cutoff = val;
            [plugin.B,plugin.A] = highpassCoeffs(plugin,val,getSampleRate(plugin)); % <
        end

        %-------------------------------------------------------------------
        % Reset Method
        %-------------------------------------------------------------------
        function reset(plugin)
            plugin.State = zeros(2);
            [plugin.B,plugin.A] = highpassCoeffs(plugin,plugin.Cutoff,getSampleRate(plu
        end
    end
    methods (Access = private)
        %-------------------------------------------------------------------
        % Calculate Filter Coefficients
        %-------------------------------------------------------------------
        function [B,A] = highpassCoeffs(~,fc,fs)
            w0    = 2*pi*fc/fs;
            alpha = sin(w0)/sqrt(2);
            cosw0 = cos(w0);
            norm  = 1/(1+alpha);
            B     = (1 + cosw0)*norm * [.5 -1 .5];
            A     = [1 -2*cosw0*norm (1 - alpha)*norm];
        end
    end
end
```

The highpassCoeffs function might be expensive, and should be called only when necessary. You do not want to call highpassCoeffs in the process method, which runs in the real-time audio processing loop. The logical place to call highpassCoeffs is in set.Cutoff. However, mlint shows a warning for this practice. The warning is intended to help you avoid initialization order issues when saving and loading classes. See “Avoid Property Initialization Order Dependency” (MATLAB) for more details. The solution recommended by the warning is to create a dependent property with a get method and compute the value there. However, following the recommendation complicates the design and pushes the computation back into the real-time processing method, which you are trying to avoid.

You might also incur the warning when correctly implementing plugin composition. For an example of a correct implementation of composition, see “Implement Plugin Composition Correctly” on page 13-9.

## Use System Object That Does Not Support Variable-Size Signals

The audio plugin API requires audio plugins to support variable-size inputs and outputs. For a partial list of System objects that support variable-size signals, see “Variable-Size Signal Support DSP System Objects” (DSP System Toolbox). You might encounter issues if you attempt to use objects that do not support variable-size signals in your plugin.

For example, dsp.AnalyticSignal does not support variable-size signals. The BrokenAnalyticSignalTransformer plugin uses a dsp.AnalyticSignal object incorrectly and fails the validateAudioPlugin test bench:

```
validateAudioPlugin BrokenAnalyticSignalTransformer
```

```
Checking plug-in class 'BrokenAnalyticSignalTransformer'... passed.
Generating testbench file 'testbench_BrokenAnalyticSignalTransformer.m'... done.
Running testbench...
Error using dsp.AnalyticSignal/parenReference
Changing the size on input 1 is not allowed without first calling the release() method
```

```
Error in BrokenAnalyticSignalTransformer/process (line 13)
                analyticSignal = plugin.Transformer(in);

Error in testbench_BrokenAnalyticSignalTransformer (line 61)
        o1 = process(plugin, in(:,1));

Error in validateAudioPlugin
```

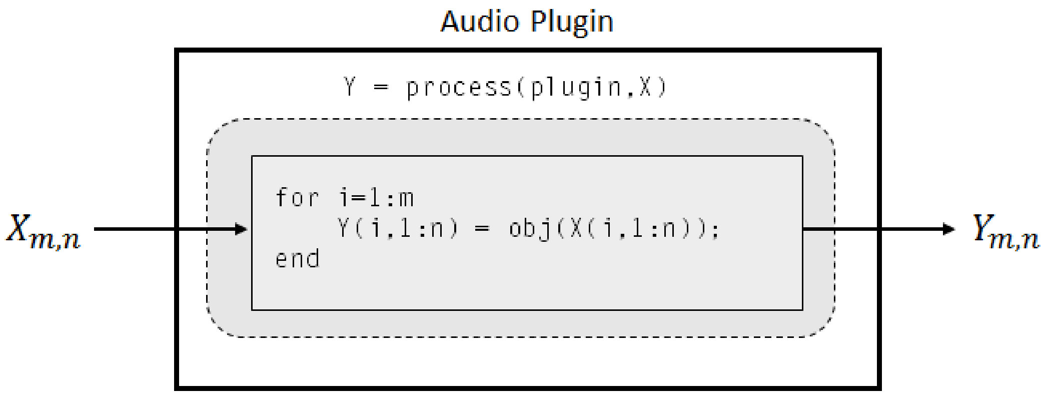
#### See `BrokenAnalyticSignalTransformer` Code

```
classdef BrokenAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
    end
    methods
        function plugin = BrokenAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
        end
        function out = process(plugin,in)
                analyticSignal = plugin.Transformer(in);
                realPart = real(analyticSignal);
                imaginaryPart = imag(analyticSignal);
                out = [realPart,imaginaryPart];
        end
    end
end
```

If you want to use the functionality of a System object that does not support variable-size signals, you can buffer the input and output of the System object, or always call the object with one sample.

### Always Call the Object with One Sample

You can create a loop around your call to an object. The loop iterates for the number of samples in your variable frame size. The call to the object inside the loop is always a single sample.



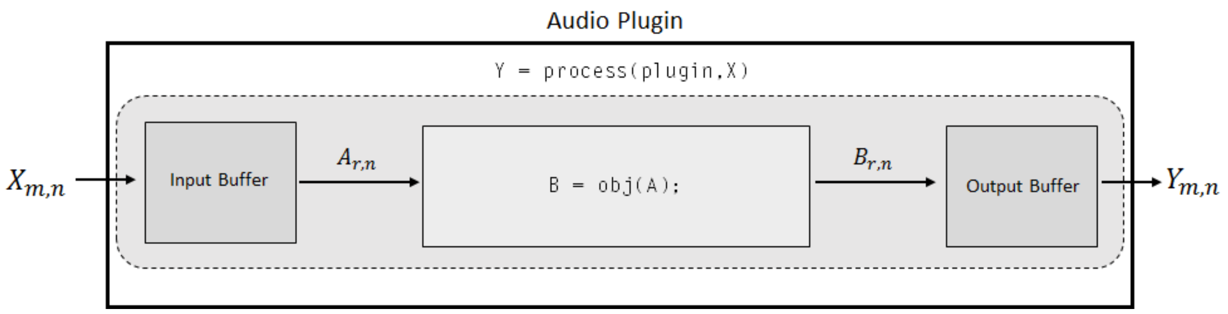
**See Full Code Example**

```
classdef ExpensiveAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
    end
    methods
        function plugin = ExpensiveAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
        end
        function out = process(plugin,in)
            analyticSignal = complex(zeros(size(in,1),1),0);
            for i = 1:size(in,1)
                analyticSignal(i,:) = plugin.Transformer(in(i,1));
            end
            out = [real(analyticSignal),imag(analyticSignal)];
        end
    end
end
```

**Note:** Depending on your implementation and the particular object, calling an object sample by sample in a loop might result in significant computational cost.

### Buffer Input and Output of Object

You can buffer the input to your object to a consistent frame size, and then buffer the output of your object back to the original frame size. The dsp.AsyncBuffer System object is well-suited for this task.



### See Full Code Example

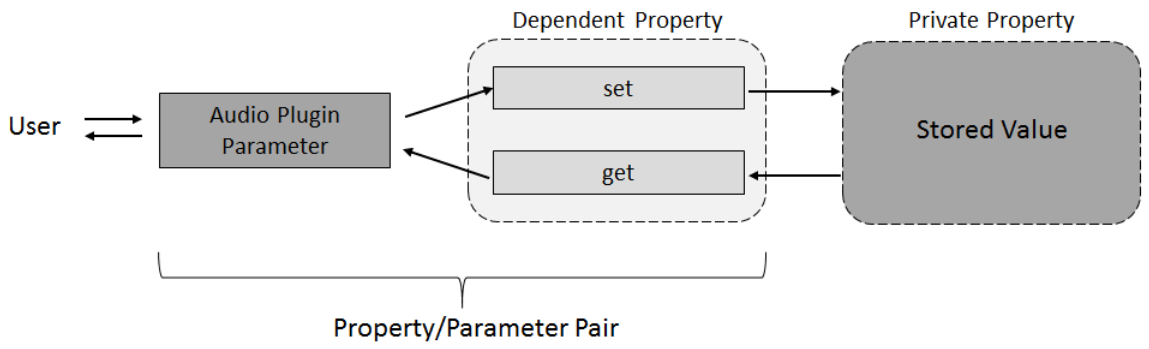
```
classdef DelayedAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
        InputBuffer
        OutputBuffer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
        MinSampleDelay = 256;
    end
    methods
        function plugin = DelayedAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
            setup(plugin.Transformer,ones(plugin.MinSampleDelay,1));

            plugin.InputBuffer = dsp.AsyncBuffer;
            setup(plugin.InputBuffer,1);

            plugin.OutputBuffer = dsp.AsyncBuffer;
            setup(plugin.OutputBuffer,[1,1]);
        end
        function out = process(plugin,in)
            write(plugin.InputBuffer,in);
            while plugin.InputBuffer.NumUnreadSamples >= plugin.MinSampleDelay
                x = read(plugin.InputBuffer,plugin.MinSampleDelay);
                analyticSignal = plugin.Transformer(x(1:plugin.MinSampleDelay,:));
                write(plugin.OutputBuffer,[real(analyticSignal),imag(analyticSignal)])
            end

            if plugin.OutputBuffer.NumUnreadSamples >= size(in,1)
                out = read(plugin.OutputBuffer,size(in,1));
            else
                out = zeros(size(in,1),2);
            end
        end
        function reset(plugin)
            reset(plugin.Transformer)
            reset(plugin.InputBuffer)
            reset(plugin.OutputBuffer)
        end
    end
end
```

**Note:** Use of the asynchronous buffering object forces a minimum latency of your specified frame size.

## Using Enumeration Parameter Mapping

It is often useful to associate a property with a set of strings or character vectors. However, restrictions on plugin generation require cached values, such as property values, to have a static size. To work around this issue, you can use a separate enumeration class that maps the strings to the enumerations, as described in the `audioPluginParameter` documentation.

Alternatively, if you want to avoid writing an enumeration class and keep all your code in one file, you can use a dependent property to map your parameter names to a set of values. In this scenario, you map your enumeration value to a value that you can cache.



**See Full Code Example**

```
classdef pluginWithEnumMapping < audioPlugin
    properties (Dependent)
        Mode = '+6 dB';
    end
    properties (Access = private)
        pMode = 1; % '+6 dB'
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Mode',...
                'Mapping',{'enum','+6 dB','-6 dB','silence','white noise'}));
    end
    methods
        function out = process(plugin,in)
            switch (plugin.pMode)
                case 1
                    out = in * 2;
                case 2
                    out = in / 2;
                case 3
                    out = zeros(size(in));
                otherwise % case 4
                    out = rand(size(in)) - 0.5;
            end
        end
        function set.Mode(plugin,val)
            validatestring(val,{'+6 dB','-6 dB','silence','white noise'},'set.Mode','Mo
            switch val
                case '+6 dB'
```

```
                    plugin.pMode = 1;
                case '-6 dB'
                    plugin.pMode = 2;
                case 'silence'
                    plugin.pMode = 3;
                otherwise % 'white noise'
                    plugin.pMode = 4;
            end
        end
        function out = get.Mode(plugin)
            switch plugin.pMode
                case 1
                    out = '+6 dB';
                case 2
                    out = '-6 dB';
                case 3
                    out = 'silence';
                otherwise % case 4
                    out = 'white noise';
            end
        end
    end
end
```

## More About

- “Design an Audio Plugin”
- “Audio Plugin Example Gallery” on page 5-2
- “Export a MATLAB Plugin to a DAW”