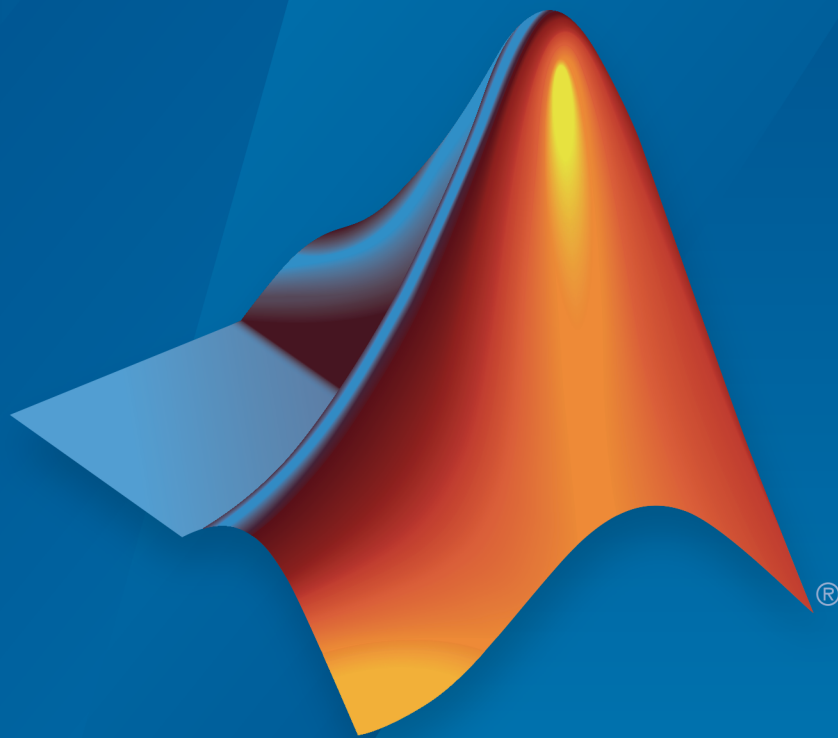# Audio System Toolbox™
Getting Started Guide

MATLAB®&SIMULINK®

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

Revision History

Contents

Introduction

Audio System Toolbox Product Description

Design and test audio processing systems

Audio System Toolbox™ provides algorithms and tools for the design, simulation, and desktop prototyping of audio processing systems. It enables low-latency signal streaming from and to audio interfaces, interactive parameter tuning, and automatic generation of audio plugins for digital audio workstations.

Audio System Toolbox includes libraries of audio processing algorithms (such as filtering, equalization, dynamic range control, and reverberation), sources (such as audio oscillators and wavetable synthesizers), and measurements (such as A- and C-weighting). Interfaces to external MIDI controls and low-latency audio drivers such as ASIO™, ALSA, and CoreAudio enable you to validate multichannel audio designs in MATLAB® or Simulink®. You can generate VST plugins from MATLAB code. For simulation acceleration or desktop prototyping, the toolbox supports C/C++ code generation.

Algorithms are available as MATLAB functions, System objects, and Simulink blocks.

Key Features

- VST plugin generation for digital audio workstations
- Interfaces to ASIO, ALSA, CoreAudio, and other low-latency audio drivers
- Interfaces to MIDI controls for real-time tuning of MATLAB and Simulink simulations
- Audio processing algorithms, sources, and measurements for crossover and equalization filtering, dynamic range control, reverberation, wavetable synthesis, and other tasks
- Support for C code generation

Acknowledgements

VST is a trademark and software of Steinberg Media Technologies GmbH.

ASIO is a trademark and software of Steinberg Media Technologies GmbH.

Export a MATLAB Plugin to a DAW

Export a MATLAB Plugin to a DAW

In this section...
"Plugin Development Workflow" on page 2-2
"Considerations When Generating Audio Plugins" on page 2-2
"How Audio Plugins Interact with the DAW Environment" on page 2-3

Audio System Toolbox enables generation of VST plugins from MATLAB source code by using the `generateAudioPlugin` function. The generated plugin is compatible with 32-bit and 64-bit Windows, and 64-bit Mac host applications. After you generate a VST plugin, you can use your generated audio plugin in a digital audio workstation (DAW).

Plugin Development Workflow

1   Design an audio plugin. For a tutorial on audio plugin architecture and design in the MATLAB environment, See "Design an Audio Plugin" on page 6-2.

2   Validate your audio plugin using the `validateAudioPlugin` function.

```
validateAudioPlugin myAudioPlugin
```

3   Test your audio plugin using Audio Test Bench.

```
audioTestBench myAudioPlugin
```

4   Generate your audio plugin using the `generateAudioPlugin` function.

```
generateAudioPlugin myAudioPlugin
```

5   Use your generated audio plugin in a DAW.

Considerations When Generating Audio Plugins

- Your plugin must be compatible with MATLAB code generation. See "MATLAB Programming for Code Generation" (MATLAB Coder) for more information.

- Your generated plugin must be compatible with DAW environments. The DAW environment:

  - Determines the sample rate and frame size at which a plugin is run, both of which are variable.

- Calls the reset function of your plugin at the beginning of each use and if the sample rate changes.
- Requires a consistent input and output frame size for the plugin's processing function.
- Must be synchronized with plugin parameters. Therefore, a plugin must not modify properties associated with parameters.
- Requires that plugin properties associated with parameters are scalar values.

Use the `validateAudioPlugin`, `Audio Test Bench`, and `generateAudioPlugin` tools to guide your audio plugin into a valid form capable of generation.

How Audio Plugins Interact with the DAW Environment

After you generate your plugin, plug it into a DAW environment. See documentation on your specific DAW for details on adding plugins.

The audio plugin in the DAW environment interacts primarily through the processing function, reset function, and interface properties of your plugin.

Initialization and Reset

- The DAW environment calls the reset function of the plugin the first time the plugin is used, or any time the sample rate of the DAW environment is modified. You can use the `getSampleRate` function to query the sample rate of the environment.

Processing

- The DAW environment passes a frame of an audio signal to the plugin. The DAW determines the frame size. If the audio plugin is a source audio plugin, the DAW does not pass an input audio signal.
- The processing function of your plugin performs the frame-based audio processing algorithm you specified, and updates internal plugin properties as needed. Plugins must not write to properties associated with parameters.
- The processing function of your plugin passes the processed audio signal out to the DAW environment. The frame size of the output signal must match the frame size of the input signal. If the audio plugin is a source audio plugin, you must use `getSamplesPerFrame` to determine the output frame size. Because the environment frame rate is variable, you must call `getSamplesPerFrame` for each output frame.
- Processing is performed iteratively frame by frame on an audio signal.

Tunability

• If you modify a parameter through the plugin dialog box, the synchronized public property updates at that time. You can use the `set` method of MATLAB classes to modify private properties.
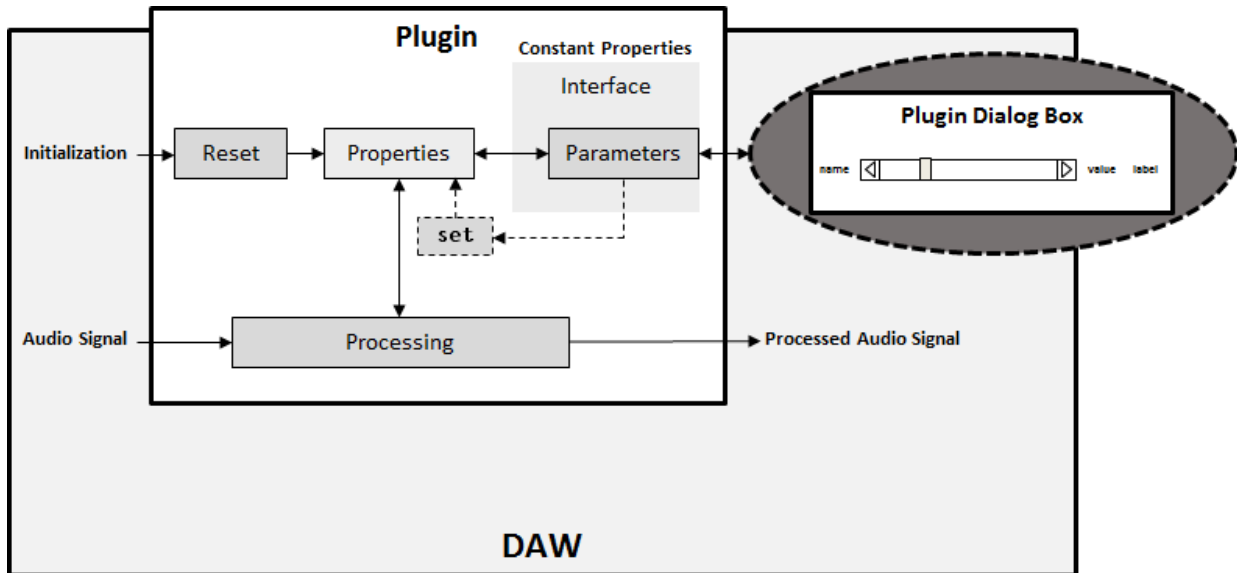
More About

• "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 4-2
• "Design an Audio Plugin" on page 6-2
• "Convert MATLAB Code to an Audio Plugin" on page 8-2

Audio I/O: Buffering, Latency, and Throughput

Audio I/O: Buffering, Latency, and Throughput

Audio System Toolbox is optimized for real-time stream processing. Its input and output System objects are efficient, low-latency, and they control all necessary parameters so that you can trade off between throughput and latency.

This tutorial describes how MATLAB software implements real-time stream processing. The tutorial presents key terminology and basic techniques for optimizing your stream processing algorithm. For more detailed technical descriptions and concepts, see the documentation for the audio I/O System objects used in this tutorial.

Input Audio Stream Signal

To acquire an audio stream from a file, use the dsp.AudioFileReader System object™. To acquire an audio stream from a device, use the audioDeviceReader System object.

This diagram and the ordered list that follows indicate the data flow when acquiring a monochannel signal with the `audioDeviceReader` System object. `audioDeviceReader` specifies the driver, the input device (sound card) and its attributes, buffer size, and provides diagnostic functionality.
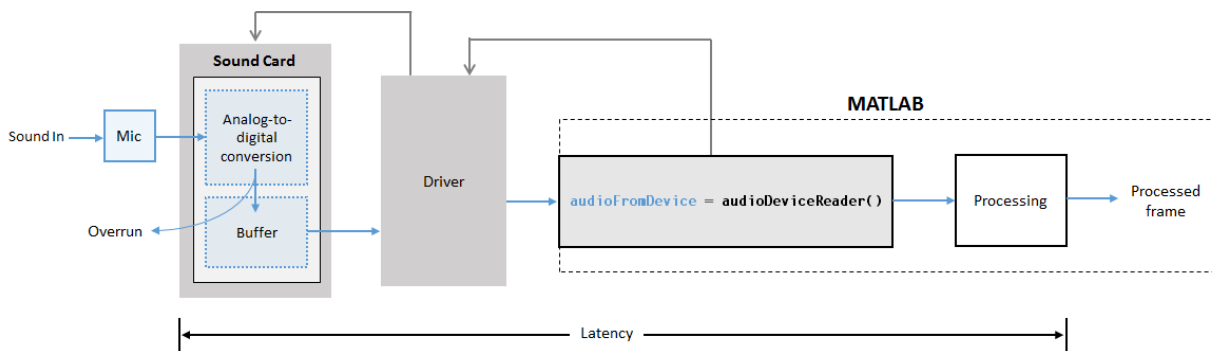
1. The microphone picks up the sound and sends a continuous electrical signal to your sound card.
2. The sound card performs analog-to-digital conversion at a sample rate, buffer size, and bit depth specified by your `audioDeviceReader` object.
3. The analog-to-digital converter writes audio samples into the sound card buffer. If the buffer is full, the new samples are dropped.

4 The `audioDeviceReader` uses the driver to pull the oldest frame from the sound card buffer iteratively.

Terminology and Techniques to Optimize Your `audioDeviceReader`

- Latency is measured as the time delay between audio entering the sound card to the frame output by the processing stage. To minimize latency, you can:

  - Optimize the processing stage. If your processing stage has reached a peak algorithmically, consider compiling your MATLAB code into C code using MATLAB Coder™.
  - Decrease the sample rate.
  - Decrease the frame size.

- Overrun refers to input signal drops. Input signal drops occur when the processing stage does not keep pace with the acquisition of samples. The number of samples overrun is returned when you call your `audioDeviceReader` object. To minimize overrun, you can:

  - Optimize the processing stage.
  - Decrease the sample rate.
  - Increase the frame size.

A typical workflow includes determining the minimum sample rate for your application, measuring overrun and latency, and then adjusting your `audioDeviceReader` properties. See audioDeviceReader for more information.

Output Audio Stream Signal

To send an audio stream to a file, use the dsp.AudioFileWriter System object. To send an audio stream to a device, use the audioDeviceWriter System object.

This diagram and the ordered list that follows indicate the data flow when playing a monochannel signal with the `audioDeviceWriter` System object. `audioDeviceWriter` specifies the driver, the output device (sound card) and its attributes, buffer size, and provides diagnostic functionality.
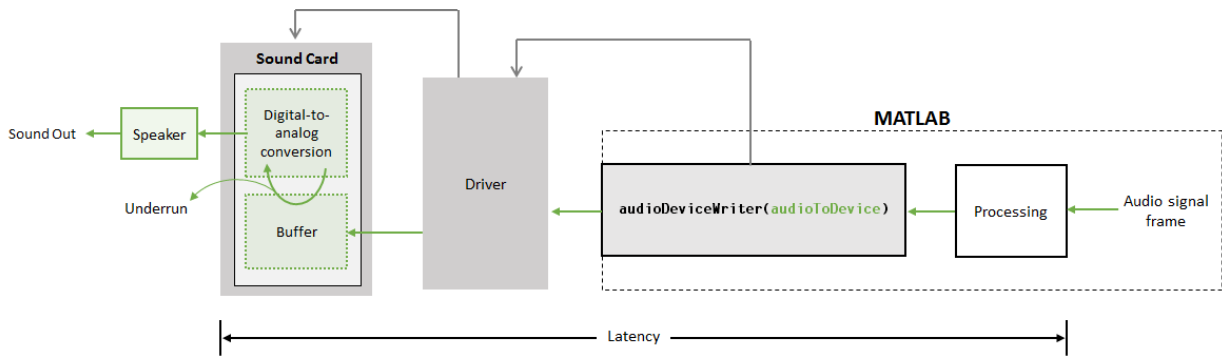
1. The processing stage passes a frame of variable length to the `audioDeviceWriter` System object.

2. `audioDeviceWriter` sends the frame to the sound card's buffer. Your `audioDeviceWriter` object specifies the sample rate, bit depth, and buffer size of the sound card.

3. The sound card pulls the oldest frame from the buffer and performs analog-to-digital conversion. The sound card sends the analog chunk to the speaker. If the buffer is empty when the sound card tries to pull from it, the sound card outputs a region of silence.

Terminology and Techniques to Optimize Your `audioDeviceWriter`

- Latency is measured as the time delay between the generation of an audio frame in MATLAB to the time audio is heard through the speaker. To minimize latency, you can:

  - Optimize the processing stage. If your processing stage has reached a peak algorithmically, consider compiling your MATLAB code into C code using MATLAB Coder.
  - Decrease the sample rate.
  - Decrease the frame size.

- Underrun refers to output signal silence. Output signal silence occurs when the buffer is empty when it is time for digital-to-analog conversion. The number of samples underrun is returned when you call your `audioDeviceWriter` object. To minimize underrun, you can:

  - Optimize the processing stage.

- Decrease the sample rate.
- Increase the frame size.
- Increase the buffer size. This approach applies only for audio signals with variable frame length.

See audioDeviceWriter for more information.

Synchronize Audio to and from Device

To simultaneously read from and write to a single audio device, use the audioPlayerRecorder object.

This diagram and the ordered list that follows indicate the data flow when playing and recording monochannel signals with the `audioPlayerRecorder` System object. `audioPlayerRecorder` specifies the driver, the input device (sound card) and its attributes, the buffer size, and provides diagnostic functionality.
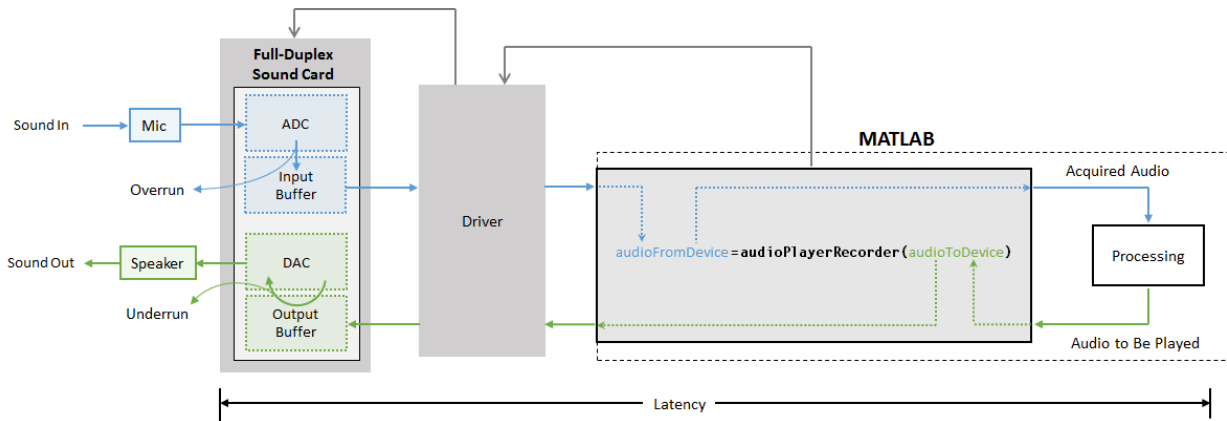
1   The microphone picks up the sound and sends a continuous electrical signal to your sound card. Simultaneously, the speaker plays an analog chunk received from the sound card.

2   The sound card performs analog-to-digital conversion of the acquired audio signal and writes the digital chunk to the input buffer. If the input buffer is full, the new samples are dropped. Simultaneously, the sound card pulls the oldest frame from the output buffer and performs digital-to-analog conversion of the next audio chunk to be played. If the output buffer is empty when the sound card tries to pull from it, the

sound card outputs a region of silence. The `audioPlayerRecorder` object specifies the sample rate, buffer size, and bit depth used for conversion.

3 The `audioPlayerRecorder` object returns the acquired audio signal to the MATLAB environment for processing. Simultaneously, the audio to be played is specified as an argument of the `audioPlayerRecorder` for playback in the next I/O cycle.

Terminology and Techniques to Optimize Your `audioPlayerRecorder`

- Output latency is measured as the time delay between the generation of an audio frame in MATLAB to the time that audio is heard through the speaker. To minimize latency, you can:

  - Optimize the processing stage. If your processing stage has reached a peak algorithmically, consider compiling your MATLAB code into C code using MATLAB Coder.
  - Decrease the sample rate.
 - Decrease the frame size.

- Input latency is measured as the time delay between when audio enters the sound card to when the frame is output by the processing stage. To minimize latency, you can:

  - Optimize the processing stage. If your processing stage has reached a peak algorithmically, consider compiling your MATLAB code into C code using MATLAB Coder.
  - Decrease the sample rate.
  - Decrease the frame size.

If properties and frame size remain consistent, the ratio of input latency to output latency is consistent between calls to the `audioPlayerRecorder`.

- Underrun refers to output signal silence. Output signal silence occurs when the buffer is empty and it is time for digital-to-analog conversion. The number of samples underrun is returned when you call your `audioPlayerRecorder` object. To minimize underrun, you can:

  - Optimize the processing stage.
  - Decrease the sample rate.
  - Increase the frame size.

- Increase the buffer size. This approach applies only for audio signals with variable frame length.

- Overrun refers to input signal drops. Input signal drops occur when the processing stage does not keep pace with the acquisition of samples. The number of samples overrun is returned when you call your `audioPlayerRecorder` object. To minimize overrun, you can:

  - Optimize the processing stage.
  - Decrease the sample rate.
  - Increase the frame size.

See Also

System Objects
dsp.AudioFileReader | dsp.AudioFileWriter | audioDeviceReader | audioDeviceWriter | audioPlayerRecorder

Blocks
Audio Device Reader | Audio Device Writer | From Multimedia File | To Multimedia File

More About

- "Real-Time Audio in MATLAB" on page 5-2
- "Real-Time Audio in Simulink" on page 7-2

What Are DAWs, Audio Plugins, and MIDI Controllers?

What Are DAWs, Audio Plugins, and MIDI Controllers?

In this section...
"Digital Audio Workstation (DAW)" on page 4-2
"Audio Plugins" on page 4-2
"Musical Instrument Digital Interface (MIDI)" on page 4-3

Digital Audio Workstation (DAW)

A digital audio workstation (DAW) is an electronic device or software application used to record, edit, and produce sound files. DAWs are controlled with a user interface. Most DAWs allow MIDI controls to tune parameters during live editing.

In the music industry, DAWs are typically used to acquire and save multiple tracks of audio recordings, and to mix, equalize, and add audio effects. DAWs generally have access to libraries of sounds and are used to create electronic music from scratch. Commercial DAWs, such as those found in recording studios, can be hardware integrated into computers.

DAWs are also used in the production of radio, television, film, podcasts, games, and anywhere complex manipulation of audio signals is needed.

DAWs generally support plugins, which are smaller pieces of software with unique functionality, therefore expanding the abilities of the DAW user.

Audio Plugins

Plugins are self-contained pieces of code that can be "plugged in" to DAWs to enhance their functionality. Generally, plugins fall into the categories of audio signal processing, analysis, or sound synthesis. Plugins usually specify a user-interface containing UI widgets, but the DAW interface might mask it. Typical plugins include equalization, dynamic range control, reverberation, delay, and virtual instruments.

To process streaming audio data, the DAW calls the plugin, passes in a frame of input audio data, and receives back a frame of processed output audio data. When a plugin parameter changes (for example, when you move a control on the plugin's UI), the DAW notifies the plugin of the new parameter value. Plugins usually have their own custom UI, but DAWs also provide a generic UI for all plugins.

Audio System Toolbox supports code generation to the most common plugin format, Steinberg's VST (Virtual Studio Technology). Audio System Toolbox also enables you to run and test externally authored VST and VST3 plugins directly in MATLAB.

For a discussion of plugin terminology and usage in the MATLAB environment, see "Design an Audio Plugin" on page 6-2.

Musical Instrument Digital Interface (MIDI)

Musical Instrument Digital Interface (MIDI) is a technical standard for communication between electronic instruments, computers, and related devices. MIDI carries event messages specific to audio signals, such as pitch and velocity, as well as control signals for parameters such as volume, vibrato, panning, cues, and clock signals to synchronize tempo.

MIDI controllers are devices that send MIDI messages. Common devices include electronic keyboards or surfaces with sliders, knobs, and buttons. For DAWs, MIDI controllers can be physical instantiations of functionality present in the DAW. The DAW user can interact using a keyboard and mouse and MIDI controllers.

More About

- "Design an Audio Plugin" on page 6-2
- "Host External Audio Plugins" on page 10-2
- "Audio Plugin Example Gallery"
- "Musical Instrument Digital Interface (MIDI)"
- "MIDI Control for Audio Plugins"

External Websites

- MIDI Manufacturers Association

Real-Time Audio in MATLAB

Real-Time Audio in MATLAB

In this section...
"Create a Development Test Bench" on page 5-2
"Quick Start Examples" on page 5-11

Audio System Toolbox is optimized for real-time audio processing. audioDeviceReader, audioDeviceWriter, audioPlayerRecorder, dsp.AudioFileReader, and dsp.AudioFileWriter are designed for streaming multichannel audio, and they provide all necessary parameters so that you can trade off between throughput and latency.

For information on real-time processing and tips on how to optimize your algorithm, see "Audio I/O: Buffering, Latency, and Throughput" on page 3-2.

This tutorial describes how you can implement audio stream processing in MATLAB. It outlines the workflow for creating a development test bench and provides examples for each stage of the workflow. Begin by inspecting the anatomy of a completed audio stream processing test bench, then walk through the example for a description of each stage.

Create a Development Test Bench

This tutorial creates a development test bench in five steps. You begin by constructing objects to input an audio signal to your test bench and output an audio signal from your test bench. You then create an audio stream loop that performs frame-based processing on your audio signal. To gain insight about your audio processing, you add scopes to visualize the input to and output from the audio stream loop. You then develop your audio processing algorithm. In the final stage, you make your processing algorithm tunable in real time.
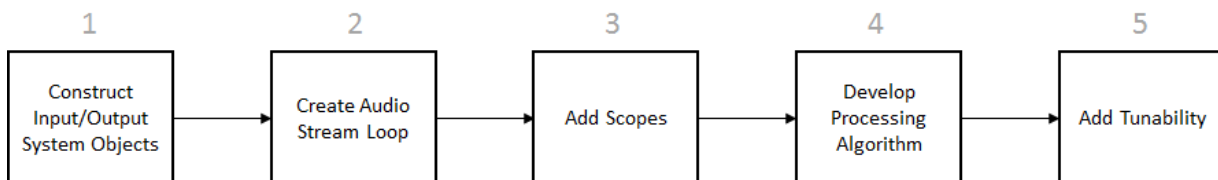
For an overview of how audio stream processing is implemented, inspect the anatomy of the completed audio stream processing test bench. To create this test bench, walk through the example for explanations and step-by-step instructions.

Anatomy of a Test Bench

```
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpan',16,...
    'BufferLength',1.5e6,...
    'YLimits',[-1 1]);
dRG = noiseGate(...
    'SampleRate',fileReader.SampleRate,...
    'Threshold',-25,...
    'AttackTime',10e-3,...
    'ReleaseTime',20e-3,...
    'HoldTime',0);

visualize(dRG);

configureMIDI(dRG);

while ~isDone(fileReader)
    signal = fileReader();
    noisySignal = signal + 0.0025*randn(frameLength,1);
    processedSignal = dRG(noisySignal);
    deviceWriter(processedSignal);
    scope([noisySignal,processedSignal]);
end

release(fileReader);
release(deviceWriter);
release(scope);
release(dRG);
```

Construct System objects for audio input/output

Construct System object for visualization

Construct System object for audio processing

Visualize audio processing algorithm

Tune audio processing algorithm with MIDI controls

Audio stream loop

Input signal frame to audio stream loop

Process signal frame in audio stream loop

Output processed signal frame from audio stream loop

Visualize processed and unprocessed signal frame

Completed Test Bench Code

Click here to open the file.

```
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpan',16,...
    'BufferLength',1.5e6,...
    'YLimits',[-1 1]);
dRG = noiseGate(...
```

```
    'SampleRate',fileReader.SampleRate,...
    'Threshold',-25,...
    'AttackTime',10e-3,...
    'ReleaseTime',20e-3,...
    'HoldTime',0);

visualize(dRG);

configureMIDI(dRG);

while ~isDone(fileReader)
    signal = fileReader();
    noisySignal = signal + 0.0025*randn(frameLength,1);
    processedSignal = dRG(noisySignal);
    deviceWriter(processedSignal);
    scope([noisySignal,processedSignal]);
end

release(fileReader);
release(deviceWriter);
release(scope);
release(dRG);
```

1. Construct Input/Output System Objects

Your audio stream loop can read audio directly from your device or from a file, and can write to a device or file. In this tutorial, you create an audio stream loop that reads audio frame by frame from a file, and outputs frame by frame to a device. See "Quick Start Examples" on page 5-11 for alternative input/output configurations.

Construct a `dsp.AudioFileReader` System object and specify a file. To reduce latency, specify a small frame size as a property of the `dsp.AudioFileReader` System object.

Also construct an `audioDeviceWriter` System object. Specify your `audioDeviceWriter` sample rate if the default of 44,100 Hz is not appropriate. If you do not modify the sample rate between input and output to your audio stream loop, use the sample rate of your input System object.

View Example Code

```
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
```

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

2. Create Audio Stream Loop

An audio stream loop refers to a programming loop that iteratively:

- Reads a frame of an audio signal
- Processes the audio signal frame
- Writes the audio signal frame

In this tutorial, the input to the audio stream loop is read from a file. The output from the audio stream loop writes to a device.

Create Audio Stream Loop with File Input and Device Output

To read a single frame of an audio file, call your `dsp.AudioFileReader` like a function without arguments. To read successive frames, call your `dsp.AudioFileReader` in the audio stream loop.

To write a single frame of an audio signal to your audio device, call your `audioDeviceWriter` like a function with the signal frame to output as the argument. To write successive frames, call your `audioDeviceWriter` in the audio stream loop.

View Example Code

```
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

while ~isDone(fileReader)                    %<---
    signal = fileReader();                   %<---
    deviceWriter(signal);                    %<---
end                                          %<---

release(fileReader);                         %<---
release(deviceWriter);                       %<---
```

All System objects have a `release` method. As a best practice, release your System object after use, especially when a System object is communicating with a hardware device such as your sound card.

3. Add Scopes

There are several scopes available to the Audio System Toolbox user. Two common scopes are the dsp.TimeScope and the dsp.SpectrumAnalyzer.

This tutorial uses the dsp.TimeScope System object to visualize the audio signal.

Add Time Scope

To display an audio signal in the time domain, construct a `dsp.TimeScope` System object. To aid visualization, specify necessary `dsp.TimeScope` properties, such as `TimeSpan`, `BufferLength`, and `YLimits`.

To display the current frame of a signal, call your `dsp.TimeScope` like a function with the signal frame to display as the argument. To display your signal in real time, call your `dsp.TimeScope` in the audio stream loop.

View Example Code

```
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...                    %<---
    'SampleRate',fileReader.SampleRate,...   %<---
    'TimeSpan',16,...                        %<---
    'BufferLength',1.5e6,...                 %<---
    'YLimits',[-1,1]);                       %<---

while ~isDone(fileReader)
    signal = fileReader();
    deviceWriter(signal);
    scope(signal);                           %<---
end

release(fileReader);
release(deviceWriter);
release(scope);                              %<---
```

4. Develop Processing Algorithm

In most applications, you want to process your audio signal in the audio stream loop. The processing stage can be

- An inline script in the audio stream loop
- A separate function called in the audio stream loop
- A System object called like a function in an audio stream loop

In this tutorial, you call the noiseGate System object like a function to process the signal in the audio stream loop.

Process Signal with `noiseGate`

Construct a `noiseGate` System object. Specify your `noiseGate` System object sample rate if the default of 44,100 Hz is not appropriate. As a best practice, use the sample rate of your input System object. To achieve the aims of your audio processing, specify necessary `noiseGate` properties, such as `Threshold`, `AttackTime`, `ReleaseTime`, and `HoldTime`.

To process the audio signal, call your `noiseGate` like a function in the audio stream loop.

In this tutorial, you add random Gaussian noise to the audio stream input to show a possible use case of the `noiseGate` System object.

View Example Code

```matlab
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpan',16,...
    'BufferLength',1.5e6,...
    'YLimits',[-1,1]);
dRG = noiseGate(...                                      %<---
    'SampleRate',fileReader.SampleRate,...               %<---
    'Threshold',-25,...                                  %<---
    'AttackTime',10e-3,...                               %<---
    'ReleaseTime',20e-3,...                              %<---
    'HoldTime',0);                                       %<---

while ~isDone(fileReader)
    signal = fileReader();
```

```matlab
        noisySignal = signal + 0.0025*randn(frameLength,1);     %<---
        processedSignal = dRG(noisySignal);                     %<---
        deviceWriter(processedSignal);                          %<---
        scope([noisySignal,processedSignal]);                   %<---
end

release(fileReader);
release(deviceWriter);
release(scope);
release(dRG);                                                   %<---
```

5. Add Tunability

MATLAB provides several options to interactively tune your algorithm with stream processing.

Add User Interface

MATLAB provides several user interfaces (UI) to inspect and interact with your code. You can use:

- The Audio Test Bench, which provides UI-based exercises for audioPlugin classes and most Audio System Toolbox System objects.
- The built-in methods of Audio System Toolbox System objects for visualizing key characteristics of your processing algorithms. Then you can tune them in real time with MIDI controls.
- A custom-built user interface. See GUI Building (MATLAB) for a tutorial.

This tutorial uses the `visualize` method of the noiseGate System object to observe its static characteristics.

View Example Code

```matlab
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpan',16,...
    'BufferLength',1.5e6,...
    'YLimits',[-1,1]);
```

```matlab
dRG = noiseGate(...
    'SampleRate',fileReader.SampleRate,...
    'Threshold',-25,...
    'AttackTime',10e-3,...
    'ReleaseTime',20e-3,...
    'HoldTime',0);

visualize(dRG);                                         %<---

while ~isDone(fileReader)
    signal = fileReader();
    noisySignal = signal + 0.0025*randn(frameLength,1);
    processedSignal = dRG(noisySignal);
    deviceWriter(processedSignal);
    scope([noisySignal,processedSignal]);
end

release(fileReader);
release(deviceWriter);
release(scope);
release(dRG);
```

Add MIDI Controller

Many Audio System Toolbox System objects include methods that support MIDI controls. This tutorial uses the `configureMIDI` method of the noiseGate System object to synchronize your System object properties to MIDI controls.

Note: To use MIDI controls with System objects that do not have a `configureMIDI` method, see "Musical Instrument Digital Interface (MIDI)".

To control your `noiseGate` System object properties with a MIDI controller, connect the MIDI device to your computer.

The `configureMIDI` method enables you to synchronize properties to MIDI controls using a user interface or a script. This example synchronizes properties to a MIDI controller using a user interface.

Before calling your audio stream loop, call the `configureMIDI` method on your `noiseGate` System object. When you run your script, it does not advance until you have completed your configuration and closed the user interface. Once the user interface opens:

1 Select a property to synchronize by choosing it from the drop-down menu.

2 Move a MIDI control.

The `noiseGate` property in the drop-down menu and the MIDI control you moved are now synced. Repeat these steps for all properties you want to synchronize. Then click **OK**.

While your audio is stream processing, use your MIDI controller to adjust the `noiseGate` parameters in real time. In particular, toggle the MIDI control mapped to the `Threshold` property to attenuate the additive Gaussian noise in the signal.

View Example Code

```matlab
frameLength = 256;
fileReader = dsp.AudioFileReader(...
    'Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpan',16,...
    'BufferLength',1.5e6,...
    'YLimits',[-1,1]);
dRG = noiseGate(...
    'SampleRate',fileReader.SampleRate,...
    'Threshold',-25,...
    'AttackTime',10e-3,...
    'ReleaseTime',20e-3,...
    'HoldTime',0);

visualize(dRG);

configureMIDI(dRG);                                        %<---

while ~isDone(fileReader)
    signal = fileReader();
    noisySignal = signal + 0.0025*randn(frameLength,1);
    processedSignal = dRG(noisySignal);
    deviceWriter(processedSignal);
    scope([noisySignal,processedSignal]);
end

release(fileReader);
```

```
release(deviceWriter);
release(scope);
release(dRG);
```

Add UDP

You can use the User Datagram Protocol (UDP) within MATLAB for connectionless transmission, or to receive or transmit datagrams outside MATLAB. Possible applications include using MATLAB to tune your audio processing algorithm while playing and visualizing your audio in a third-party environment. See "Communicate Between a DAW and MATLAB Using UDP" for an example application of UDP communication.

Quick Start Examples

Audio Stream from Device to Device

This example uses the `audioDeviceReader` and `audioDeviceWriter` System objects to perform real-time I/O stream processing. The processing is limited to adding a gain. Click here to open the file.

```
%% Real-Time Audio Stream Processing
%
% The Audio System Toolbox provides real-time, low-latency processing of
% audio signals using the System objects audioDeviceReader and
% audioDeviceWriter.
%
% This example shows how to acquire an audio signal using your microphone,
% perform basic signal processing, and play back your processed
% signal.
%

%% Create input and output objects
deviceReader = audioDeviceReader;
deviceWriter = audioDeviceWriter('SampleRate',deviceReader.SampleRate);

%% Specify an audio processing algorithm
% For simplicity, only add gain.
process = @(x) x.*5;

%% Code for stream processing
% Place the following steps in a while loop for continuous stream
% processing:
```

```
%   1. Call your audio device reader like a function with no arguments to
%   acquire one input frame.
%   2. Perform your signal processing operation on the input frame.
%   3. Call your audio device writer like a function with the processed
%   frame as an argument.

disp('Begin Signal Input...')
tic
while toc<5
    mySignal = deviceReader();
    myProcessedSignal = process(mySignal);
    deviceWriter(myProcessedSignal);
end
disp('End Signal Input')

release(deviceReader)
release(deviceWriter)
```

Audio Stream from Device to File

This example uses the `audioDeviceReader` and `dsp.AudioFileWriter` System objects to perform real-time I/O stream processing. The processing is limited to adding a gain. Click here to open the file.

```
%% Real-Time Audio Stream Processing
%
% The Audio System Toolbox provides real-time, low-latency processing of
% audio signals using the System objects audioDeviceReader and
% dsp.AudioFileWriter.
%
% This example shows how to acquire an audio signal using your microphone,
% perform basic signal processing, and write your signal to a file.
%

%% Construct input and output objects
% Use the sample rate of your input as the sample rate of your output.
deviceReader = audioDeviceReader;
fileWriter = dsp.AudioFileWriter('SampleRate',deviceReader.SampleRate);

%% Specify an audio processing algorithm
% For simplicity, only add gain.
process = @(x) x.*5;

%% Code for stream processing
% Place the following steps in a while loop for continuous stream
```

```matlab
% processing:
%   1. Call your audio device reader like a function with no arguments to
%   acquire one input frame.
%   2. Perform your signal processing operation on the input frame.
%   3. Call your audio file reader like a function with the processed frame
%   as an argument.
%     Note: The file is named 'output.wav' and written to current folder by default.

disp('Begin Signal Input...')
tic
while toc<5
    mySignal = deviceReader();
    myProcessedSignal = process(mySignal);
    fileWriter(myProcessedSignal);
end
disp('End Signal Input')

release(deviceReader);
release(fileWriter);
```

Audio Stream from File to Device

This example uses the `dsp.AudioFileReader` and `audioDeviceWriter` System objects to perform real-time I/O stream processing. The processing is limited to adding a gain. Click here to open the file.

```matlab
%% Real-Time Audio Stream Processing
%
% The Audio System Toolbox provides real-time, low-latency processing of
% audio signals using the System objects dsp.AudioFileReader and
% audioDeviceWriter.
%
% This example shows how to acquire an audio signal using
% dsp.AudioFileReader, perform basic signal processing, and play your
% processed signal using audioDeviceWriter.
%

%% Construct input and output objects
% Use the sample rate of your input as the sample rate of your output.
fileReader = dsp.AudioFileReader('speech_dft.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

%% Specify an audio processing algorithm
% For simplicity, only add gain.
process = @(x) x.*5;
```

```
%% Code for stream processing
% Place the following steps in a while loop for continuous stream
% processing until dsp.AudioFileReader is done reading the file:
%   1. Call your audio file reader like a function with no arguments to
%   read one input frame.
%   2. Perform your signal processing operation on the input frame.
%   3. Call your audio device writer like a function with the processed
%   frame as an argument.

while ~isDone(fileReader)
    mySignal = fileReader();
    myProcessedSignal = process(mySignal);
    deviceWriter(myProcessedSignal);
end

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

More About

- "Real-Time Audio in Simulink" on page 7-2
- "Audio I/O: Buffering, Latency, and Throughput" on page 3-2
- "Musical Instrument Digital Interface (MIDI)"
- "Audio Test Bench Walkthrough"

Design an Audio Plugin

Design an Audio Plugin

In this section...

Role of Audio Plugins in Audio System Toolbox

The audio plugin is the suggested paradigm for developing your audio processing algorithm in Audio System Toolbox. Once designed, the audio plugin can be validated, generated, and deployed to a third-party digital audio workstation (DAW).

Additional benefits of developing your audio processing as an audio plugin include:

- Rapid prototyping using the Audio Test Bench
- Integration with MIDI devices
- Code reuse

Some understanding of object-oriented programming (OOP) in the MATLAB environment is required to optimize your use of the audio plugin paradigm. If you are unfamiliar with these concepts, see "Why Use Object-Oriented Design" (MATLAB).

For a review of audio plugins as defined outside the MATLAB environment, see "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 4-2

Defining Audio Plugins in the MATLAB Environment

In the MATLAB environment, an audio plugin refers to a class derived from the `audioPlugin` base class or the `audioPluginSource` base class.

MATLAB Environment
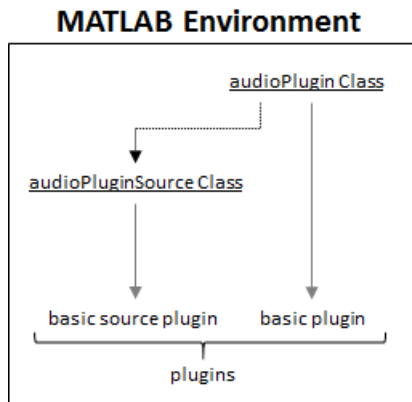
Audio System Toolbox documentation uses the following terminology:

- A *plugin* is any audio plugin that derives from the `audioPlugin` class or the `audioPluginSource` class.
- A *basic plugin* is an audio plugin that derives from the `audioPlugin` class.
- A *basic source plugin* is an audio plugin that derives from the `audioPluginSource` class.

Audio plugins can also inherit from `matlab.System`. Any object that derives from `matlab.System` is referred to as a System object. Deriving from `matlab.System` allows for additional functionality, including Simulink integration. However, manipulating System objects requires a more advanced understanding of OOP in the MATLAB environment.

See "Audio System Toolbox Extended Terminology" on page 6-20 for a detailed visualization of inheritance and terminology.

Design a Basic Plugin

In this example, you create a simple plugin, and then gradually increase complexity. Your final plugin uses a circular buffer to add an echo effect to an input audio signal. For additional considerations for generating a plugin, see "Export a MATLAB Plugin to a DAW" on page 2-2.

1   **Define a Basic Plugin Class.**   Begin with a simple plugin that copies input to output without modification.

```matlab
classdef myEchoPlugin < audioPlugin
    methods
        function out = process(~, in)
            out = in;
        end
    end
end
```

`myEchoPlugin` illustrates the two minimum requirements for audio plugin classes. They must:

- Inherit from `audioPlugin` class
- Have a `process` method

The `process` method contains the primary frame-based audio processing algorithm. It is called in an audio stream loop to process an audio signal over time.

By default, both the input to and output from the `process` method have two channels (columns). The number of input rows (frame size) passed to `process` is determined by the environment in which it is run. The output must have the same number of rows as the input.

2   **Add a Plugin Property.**   A property can store information in an object. Add a property, `Gain`, to your class definition. Modify your `process` method to multiply the input by the value specified by the `Gain` property.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
    properties                                %<---
        Gain = 1.5;                           %<---
    end                                       %<---
    methods
        function out = process(plugin, in) %<---
            out = in*plugin.Gain;          %<---
        end
    end
end
```

The first argument of the `process` method has changed from ~ to `plugin`. The first argument of `process` is reserved for the audio plugin object. If a variable is specified

as the first argument of `process`, then all `myEchoPlugin` properties are accessible in the `process` method.

3   **Add a Plugin Parameter.**   Plugin parameters are the interface between plugin properties and the plugin user. The definition of this interface is handled by `audioPluginInterface`, which holds `audioPluginParameter` objects. To associate a plugin property to a parameter, specify the first argument of `audioPluginParameter` as a character vector entered exactly as the property you want to associate. The remaining arguments of `audioPluginParameter` specify optional additional parameter attributes.

In this example, you specify a mapping between the value of the parameter and its associated property, as well as the parameter display name as it appears on a plugin dialog box. By specifying `'Mapping'` as `{'lin',0,3}`, you set a linear mapping between the `Gain` property and the associated user-facing parameter, with an allowable range for the property between 0 and 3.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Constant)                           %<---
        PluginInterface = audioPluginInterface(...  %<---
            audioPluginParameter('Gain',...         %<---
            'DisplayName','Echo Gain',...           %<---
            'Mapping',{'lin',0,3}))                 %<---
    end                                             %<---
    methods
        function out = process(plugin, in)
            out = in*plugin.Gain;
        end
    end
end
```

4   **Add Private Properties.**   Add properties to store a circular buffer, a buffer index, and the N-sample delay of your echo. Because the plugin user does not need to see them, make `CircularBuffer`, `BufferIndex`, and `NSamples` private properties. It is best practice to initialize properties to their type and size.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
```

```
        properties
            Gain = 1.5;
        end
        properties (Access = private)                          %<---
            CircularBuffer = zeros(192001,2);           %<---
            BufferIndex = 1;                            %<---
            NSamples = 0;                               %<---
        end                                             %<---
        properties (Constant)
            PluginInterface = audioPluginInterface(...
                audioPluginParameter('Gain',...
                'DisplayName','Echo Gain',...
                'Mapping',{'lin',0,3}))
        end
        methods
            function out = process(plugin, in)
                out = in*plugin.Gain;
            end
        end
    end
```

5 **Add an Echo.**   In the `process` method, write to and read from your circular buffer
   to create an output that consists of your input and a gain-adjusted echo. The first
   line of the `process` method initializes the output to the size of the input. It is best
   practice to initialize your output to avoid errors when generating plugins.

   View Code

```
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}))
    end
    methods
        function out = process(plugin, in)
```

```
        out = zeros(size(in));                              %<---
        writeIndex = plugin.BufferIndex;                    %<---
        readIndex = writeIndex - plugin.NSamples;           %<---
        if readIndex <= 0                                   %<---
            readIndex = readIndex + 192001;                 %<---
        end                                                 %<---
                                                            %<---
        for i = 1:size(in,1)                                %<---
            plugin.CircularBuffer(writeIndex,:) = in(i,:);  %<---
                                                            %<---
            echo = plugin.CircularBuffer(readIndex,:);      %<---
            out(i,:) = in(i,:) + echo * plugin.Gain;        %<---
                                                            %<---
            writeIndex = writeIndex + 1;                    %<---
            if writeIndex > 192001                          %<---
                writeIndex = 1;                             %<---
            end                                             %<---
                                                            %<---
            readIndex = readIndex + 1;                      %<---
            if readIndex > 192001                           %<---
                readIndex = 1;                              %<---
            end                                             %<---
        end                                                 %<---
        plugin.BufferIndex = writeIndex;                    %<---
    end
end
end
```

6 **Make the Echo Delay Tunable.** To allow the user to modify the NSamples delay of the echo, define a public property, Delay, and associate it with a parameter. Use the default audioPluginParameter mapping to allow the user to set the echo delay between 0 and 1 seconds.

Add a set method that listens for changes to the Delay property. Use the getSampleRate method of the audioPlugin base class to return the environment sample rate. Approximate a delay specified in seconds as a number of samples, NSamples. If the plugin user modifies the Delay property, set.Delay is called and the delay in samples (NSamples) is calculated. If the environment sample rate is above 192,000 Hz, the plugin does not perform as expected.

View Code

```
classdef myEchoPlugin < audioPlugin
```

6-7

```matlab
properties
    Gain = 1.5;
    Delay = 0.5;          %<---
end
properties (Access = private)
    CircularBuffer = zeros(192001,2);
    BufferIndex = 1;
    NSamples = 0;
end
properties (Constant)
    PluginInterface = audioPluginInterface(...
        audioPluginParameter('Gain',...
        'DisplayName','Echo Gain',...
        'Mapping',{'lin',0,3}),...          %<---
        audioPluginParameter('Delay',...    %<---
        'DisplayName','Echo Delay',...      %<---
        'Label','seconds'))                 %<---
end
methods
    function out = process(plugin, in)
        out = zeros(size(in));
        writeIndex = plugin.BufferIndex;
        readIndex = writeIndex - plugin.NSamples;
        if readIndex <= 0
            readIndex = readIndex + 192001;
        end

        for i = 1:size(in,1)
            plugin.CircularBuffer(writeIndex,:) = in(i,:);

            echo = plugin.CircularBuffer(readIndex,:);
            out(i,:) = in(i,:) + echo*plugin.Gain;

            writeIndex = writeIndex + 1;
            if writeIndex > 192001
                writeIndex = 1;
            end

            readIndex = readIndex + 1;
            if readIndex > 192001
                readIndex = 1;
            end
        end
        plugin.BufferIndex = writeIndex;
```

```matlab
            end
            function set.Delay(plugin, val)                          %<---
                plugin.Delay = val;                                  %<---
                plugin.NSamples = floor(getSampleRate(plugin)*val);  %<---
            end                                                      %<---
        end
    end
```

7   **Add a Reset Function.**   The `reset` method of a plugin contains instructions
to reset the plugin between uses or when the environment sample rate changes.
Because `NSamples` depends on the environment sample rate, update its value in the
`reset` method.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
        Delay = 0.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}),...
            audioPluginParameter('Delay',...
            'DisplayName','Echo',...
            'Label','seconds'))
    end
    methods
        function out = process(plugin, in)
            out = zeros(size(in));
            writeIndex = plugin.BufferIndex;
            readIndex = writeIndex - plugin.NSamples;
            if readIndex <= 0
                readIndex = readIndex + 192001;
            end

            for i = 1:size(in,1)
                plugin.CircularBuffer(writeIndex,:) = in(i,:);
```

```
                        echo = plugin.CircularBuffer(readIndex,:);
                        out(i,:) = in(i,:) + echo*plugin.Gain;

                        writeIndex = writeIndex + 1;
                        if writeIndex > 192001
                            writeIndex = 1;
                        end

                        readIndex = readIndex + 1;
                        if readIndex > 192001
                            readIndex = 1;
                        end
                    end
                    plugin.BufferIndex = writeIndex;
                end
                function set.Delay(plugin, val)
                    plugin.Delay = val;
                    plugin.NSamples = floor(getSampleRate(plugin)*val);
                end
                function reset(plugin)                                          %<---
                    plugin.CircularBuffer = zeros(192001,2);                    %<---
                    plugin.NSamples = floor(getSampleRate(plugin)*plugin.Delay);%<---
                end                                                             %<---
            end
        end
```

Click here to open the completed plugin in MATLAB.

Design a System Object Plugin

You can map the basic plugin to a System object plugin. Note the differences between the two plugin types:

- A System object plugin inherits from both the `audioPlugin` base class and the `matlab.System` base class, not just `audioPlugin` base class.
- The primary audio processing method of a System object plugin is named `stepImpl`, not `process`.
- The reset method of a System object is named `resetImpl`, not `reset`.
- Both `resetImpl` and `stepImpl` must be defined as protected methods.
- System objects enable alternatives to the `set` method. For more information, see processTunedPropertiesImpl.

System Object Plugin

```matlab
classdef myEchoSystemObjectPlugin < audioPlugin & matlab.System
    properties
        Gain = 1.5;
        Delay = 0.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}),...
            audioPluginParameter('Delay',...
            'DisplayName','Echo',...
            'Label','seconds'))
    end
    methods (Access = protected)
        function out = stepImpl(plugin, in)
            out = zeros(size(in));
            writeIndex = plugin.BufferIndex;
            readIndex = writeIndex - plugin.NSamples;
            if readIndex <= 0
                readIndex = readIndex + 192001;
            end

            for i = 1:size(in,1)
                plugin.CircularBuffer(writeIndex,:) = in(i,:);

                echo = plugin.CircularBuffer(readIndex,:);
                out(i,:) = in(i,:) + echo * plugin.Gain;

                writeIndex = writeIndex + 1;
                if writeIndex > 192001
                    writeIndex = 1;
                end

                readIndex = readIndex + 1;
                if readIndex > 192001
                    readIndex = 1;
```

```
                end
            end
            plugin.BufferIndex = writeIndex;
        end
        function resetImpl(plugin)
            plugin.CircularBuffer = zeros(192001,2);
            plugin.NSamples = floor(getSampleRate(plugin) * plugin.Delay);
        end
    end
    methods
        function set.Delay(plugin, val)
            plugin.Delay = val;
            plugin.NSamples = floor(getSampleRate(plugin) * val);
        end
    end
end
```

Click here to open the file.

Quick Start Basic Plugin

Template

```
classdef myBasicPlugin < audioPlugin
    % myBasicPlugin is a template basic plugin. Use this template to create
    % your own basic plugin.

    properties
        % Use this section to initialize properties that the end-user interacts
        % with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does not
        % interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public property
        % with a tunable parameter.
    end
    methods
        function out = process(plugin, in)
            % This section contains instructions to process the input audio
```

```matlab
            % signal. Use plugin.MyProperty to access a property of your
            % plugin.
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin between
            % uses or if the environment sample rate changes.
        end
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
    end
end
```

Annotated Example

This basic plugin enables the user to tune a damped applied gain. Click here to open the file.

```
classdef myBasicPlugin < audioPlugin              ← Inherit from audioPlugin base class.
    properties
        GaindB = 0;
    end
    properties (Access=private)                        If the plugin user does not interact with
        GainMemory = 1;                           ←   property directly, specify as a private
        GainLinear = 1;                                property.
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('GaindB',...
            'DisplayName','Damped Gain',...
            'Label','dB',...
            'Mapping',{'pow',1/3,-60,20}));
    end
    methods
        function out = process(plugin,in)
            gain = 0.5*(plugin.GainLinear + plugin.GainMemory);    Main processing algorithm. The method
            out = in*gain;                                     ←  must be named process.

            plugin.GainMemory = gain;
        end
        function reset(plugin)                               Called at the beginning of each use, or when
            plugin.GainMemory = 1;                        ←  the environment sample rate changes. The
        end                                                 function must be named reset.
        function set.GaindB(plugin, val)
            plugin.GaindB = val;                             Called if specified property, GaindB, is
            plugin.GainLinear = 10^(val/20);              ←  modified.
        end
    end
end
```

Define properties to hold data.

Define plugin interface as a **Constant** property. The property must be named **PluginInterface**.

Define methods to update and process data.

Quick Start Basic Source Plugin

Template

```
classdef myBasicSourcePlugin < audioPluginSource
    % myBasicSourcePlugin is a template for a basic source plugin. Use this
    % template to create your own basic source plugin.

    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does
        % not interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
```
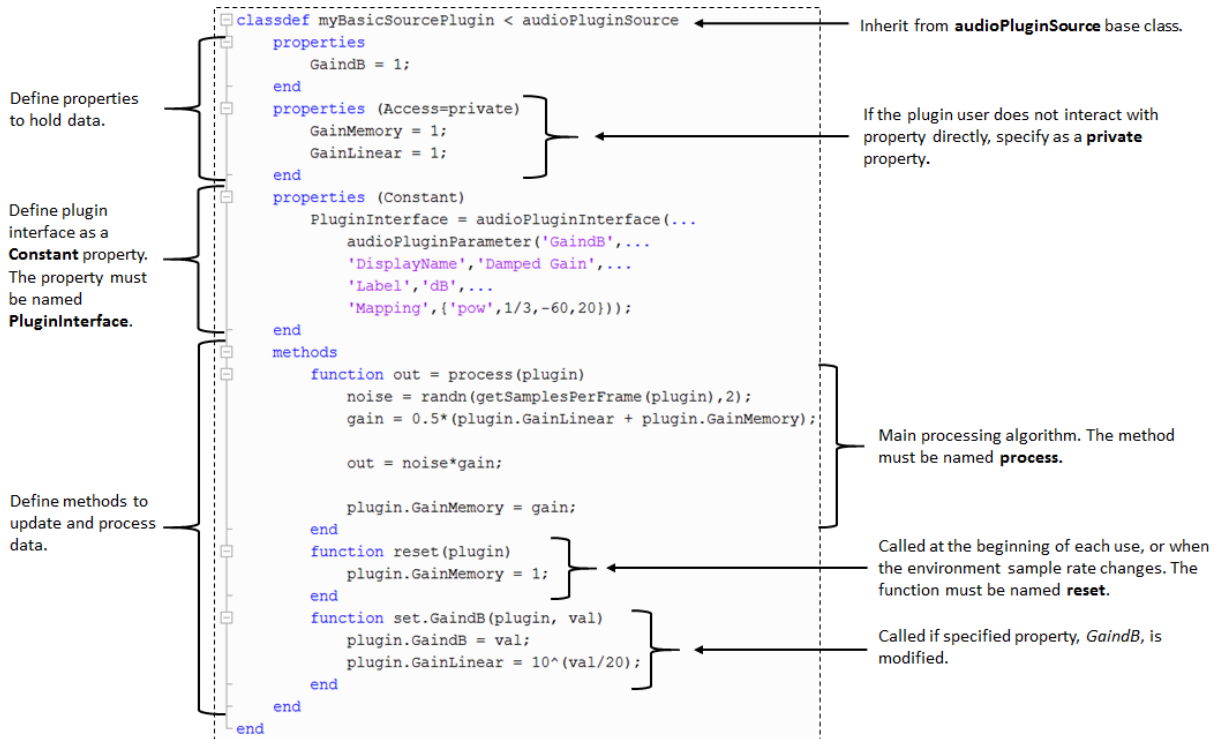
```matlab
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
    end
    methods
        function out = process(plugin)
            % This section contains instructions to produce the output
            % audio signal. Use plugin.MyProperty to access a property of
            % your plugin. Use getSamplesPerFrame(plugin) to get the frame
            % size used by the environment.
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin
            % between uses, or when the environment sample rate changes.
        end
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
    end
end
```

Annotated Example

This basic source plugin enables the user to tune the damped gain of a noise signal. Click here to open the file.

Define properties to hold data.

Define plugin interface as a **Constant** property. The property must be named **PluginInterface**.

Define methods to update and process data.

```
classdef myBasicSourcePlugin < audioPluginSource
    properties
        GaindB = 1;
    end
    properties (Access=private)
        GainMemory = 1;
        GainLinear = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('GaindB',...
            'DisplayName','Damped Gain',...
            'Label','dB',...
            'Mapping',{'pow',1/3,-60,20}));
    end
    methods
        function out = process(plugin)
            noise = randn(getSamplesPerFrame(plugin),2);
            gain = 0.5*(plugin.GainLinear + plugin.GainMemory);

            out = noise*gain;

            plugin.GainMemory = gain;
        end
        function reset(plugin)
            plugin.GainMemory = 1;
        end
        function set.GaindB(plugin, val)
            plugin.GaindB = val;
            plugin.GainLinear = 10^(val/20);
        end
    end
end
```

Inherit from **audioPluginSource** base class.

If the plugin user does not interact with property directly, specify as a **private** property.

Main processing algorithm. The method must be named **process**.

Called at the beginning of each use, or when the environment sample rate changes. The function must be named **reset**.

Called if specified property, *GaindB*, is modified.

Quick Start System Object Plugin

Template

```
classdef mySystemObjectPlugin < audioPlugin & matlab.System
    % mySystemObjectPlugin is a template for System object plugins.
    % Use this template to create your own System object plugin.

    properties
        % Use this section to initialize properties that the end-user interacts
        % with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does not
        % interact with directly.
    end
    properties (Constant)
```
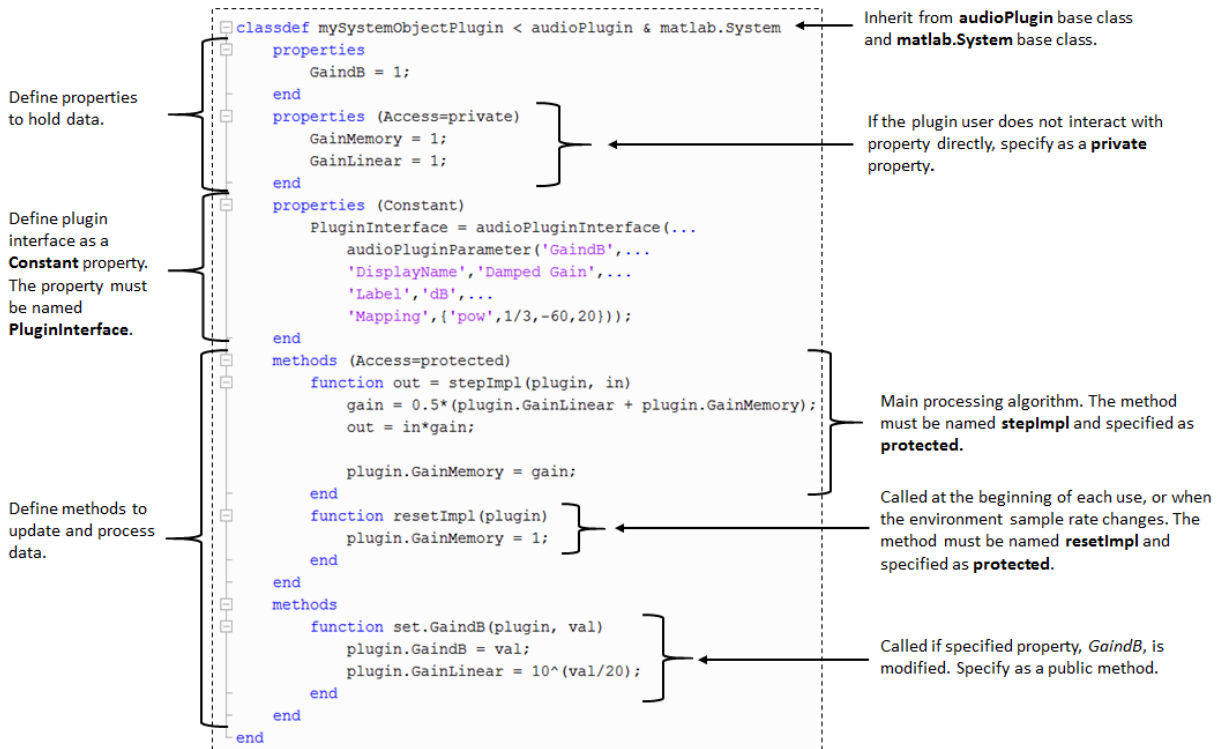
```matlab
            % This section contains instructions to build your audio plugin
            % interface. The end-user uses the interface to adjust tunable
            % parameters. Use audioPluginParameter to associate a public property
            % with a tunable parameter.
        end
    methods (Access = protected)
        function out = stepImpl(plugin)
            % This section contains instructions to process the input audio
            % signal. Use plugin.MyProperty to access a property of your
            % plugin.
        end
        function resetImpl(plugin)
            % This section contains instructions to reset the plugin between
            % uses or if the environment sample rate changes.
        end
    end
    methods
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the specified
            % property is modified. Properties associated with parameters are updated
            % automatically. Use the set method to execute more complicated
            % instructions.
        end
    end
end
```

Annotated Example

This System object plugin enables the user to tune a damped applied gain. Click here to open the file.

```
classdef mySystemObjectPlugin < audioPlugin & matlab.System
    properties
        GaindB = 1;
    end
    properties (Access=private)
        GainMemory = 1;
        GainLinear = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('GaindB',...
            'DisplayName','Damped Gain',...
            'Label','dB',...
            'Mapping',{'pow',1/3,-60,20}));
    end
    methods (Access=protected)
        function out = stepImpl(plugin, in)
            gain = 0.5*(plugin.GainLinear + plugin.GainMemory);
            out = in*gain;

            plugin.GainMemory = gain;
        end
        function resetImpl(plugin)
            plugin.GainMemory = 1;
        end
    end
    methods
        function set.GaindB(plugin, val)
            plugin.GaindB = val;
            plugin.GainLinear = 10^(val/20);
        end
    end
end
```

Inherit from **audioPlugin** base class and **matlab.System** base class.

Define properties to hold data.

If the plugin user does not interact with property directly, specify as a **private** property.

Define plugin interface as a **Constant** property. The property must be named **PluginInterface**.

Main processing algorithm. The method must be named **stepImpl** and specified as **protected**.

Called at the beginning of each use, or when the environment sample rate changes. The method must be named **resetImpl** and specified as **protected**.

Define methods to update and process data.

Called if specified property, *GaindB*, is modified. Specify as a public method.

Quick Start System Object Source Plugin

Template

```
classdef mySystemObjectSourcePlugin < audioPluginSource & matlab.System
    % mySystemObjectPlugin is a template for System object source plugins.
    % Use the template to create your own System object source plugin.

    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does
        % not interact with directly.
    end
```

```matlab
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
    end
    methods (Access = protected)
        function out = stepImpl(plugin)
            % This section contains instructions to produce the output
            % audio signal. Use plugin.MyProperty to access a property of
            % your plugin. Use getSamplesPerFrame(plugin) to get the frame
            % size used by the environment.
        end
        function resetImpl(plugin)
            % This section contains instructions to reset the plugin
            % between uses or if the environment sample rate changes.
        end
    end
    methods
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
    end
end
```

Annotated Example

This System object source plugin enables the user to tune the damped gain of a noise signal. Click here to open the file.

```
classdef mySystemObjectSourcePlugin < audioPluginSource & matlab.System
    properties
        GaindB = 0;
    end
    properties (Access=private)
        GainMemory = 1;
        GainLinear = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('GaindB',...
            'DisplayName','Damped Gain',...
            'Label','dB',...
            'Mapping',{'pow',1/3,-60,20}));
    end
    methods (Access=protected)
        function out = stepImpl(plugin)
            noise = randn(getSamplesPerFrame(plugin),2);
            gain = 0.5*(plugin.GainLinear + plugin.GainMemory);
            out = noise*gain;

            plugin.GainMemory = gain;
        end
        function resetImpl(plugin)
            plugin.GainMemory = 1;
        end
    end
    methods
        function set.GaindB(plugin, val)
            plugin.GaindB = val;
            plugin.GainLinear = 10^(val/20);
        end
    end
end
```

Inherit from **audioPluginSource** base class and **matlab.System** base class.

Define properties to hold data.

If the plugin user does not interact with property directly, specify as a **private** property.

Define plugin interface as a **Constant** property. The property must be named **PluginInterface**.

Main processing algorithm. The method must be named **stepImpl** and specified as **protected**.

Define methods to update and process data.

Called at the beginning of each use, or when the environment sample rate changes. The method must be named **resetImpl** and specified as **protected**.

Called if specified property, *GaindB*, is modified. Specify as a public method.

Audio System Toolbox Extended Terminology

In the MATLAB environment, an audio plugin refers to a class derived from the `audioPlugin` base class or the `audioPluginSource` base class. Audio plugins can also inherit from `matlab.System`. Any object that derives from `matlab.System` is referred to as a System object. Deriving from `matlab.System` allows for additional functionality, including Simulink integration. However, manipulating System objects requires a more advanced understanding of OOP in the MATLAB environment.

MATLAB Environment
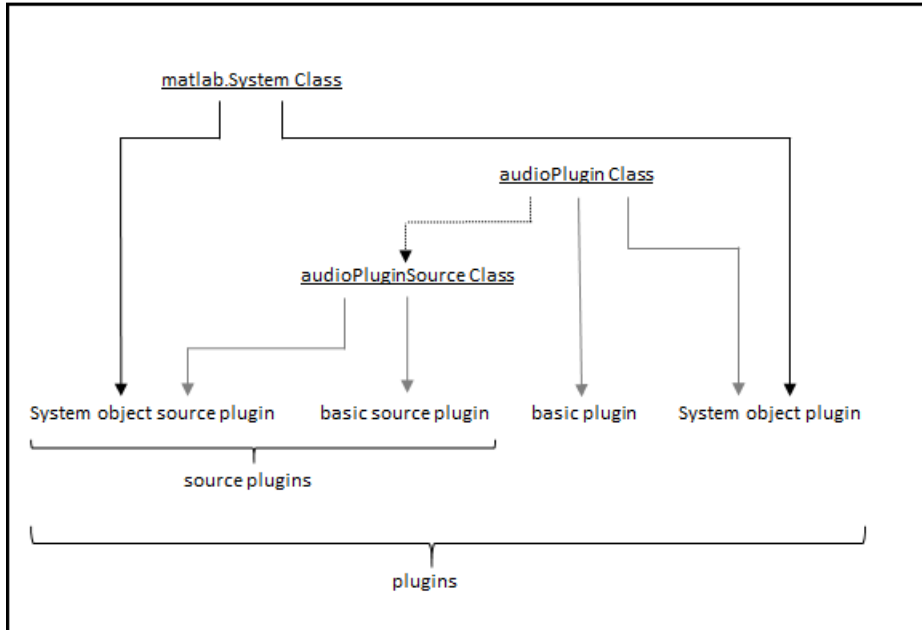
More About

- "Convert MATLAB Code to an Audio Plugin" on page 8-2
- "Convert Audio Plugin System Objects to Simulink Blocks" on page 9-2
- "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 4-2
- "Export a MATLAB Plugin to a DAW" on page 2-2

Real-Time Audio in Simulink

Real-Time Audio in Simulink

In this section...
"Create Model Using Audio System Toolbox Simulink Model Templates" on page 7-2
"Add Audio System Toolbox Blocks to Model" on page 7-3
"Recommended Settings for Audio Signal Processing" on page 7-6

Create Model Using Audio System Toolbox Simulink Model Templates

The Audio System Toolbox Simulink model templates let you automatically configure the Simulink environment for audio signal processing. See "Recommended Settings for Audio Signal Processing" on page 7-6. These templates enable reuse of settings, including configuration parameters. For more information on Simulink model templates, see "Create a Model" (Simulink) in the Simulink documentation.

To create a model using the Audio System Toolbox Simulink model templates:

1 Open the Simulink Start Page by typing `simulink` at the MATLAB command prompt.
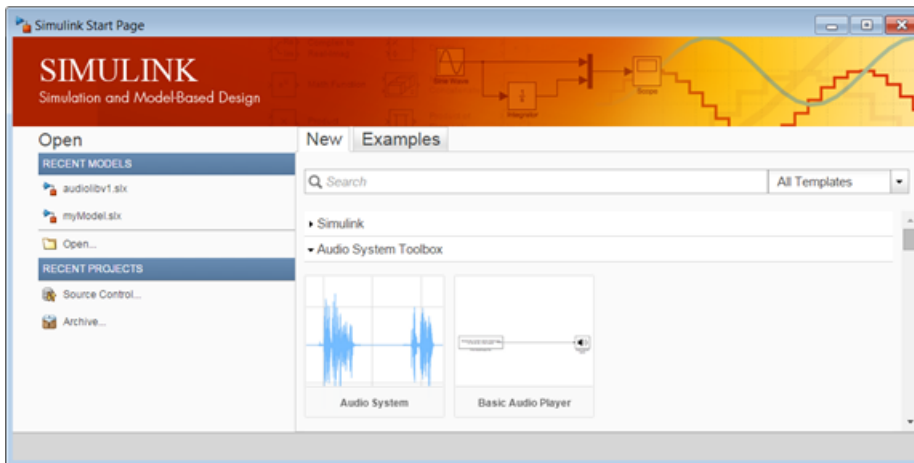
2 Under Audio System Toolbox, click the model template you want, and then click

 .

The two Audio System Toolbox Simulink model templates are:

- Audio System – Creates a blank model configured with settings recommended for Audio System Toolbox.

- Basic Audio Player – Creates an audio model configured with settings recommended for Audio System Toolbox. This model uses a From Multimedia File block to read multimedia files, and a Audio Device Writer block to send sound data to the default audio device of your computer. Adjust the model as needed to model your audio system. For example, to process live audio input, replace the From Multimedia File block with an Audio Device Reader block.
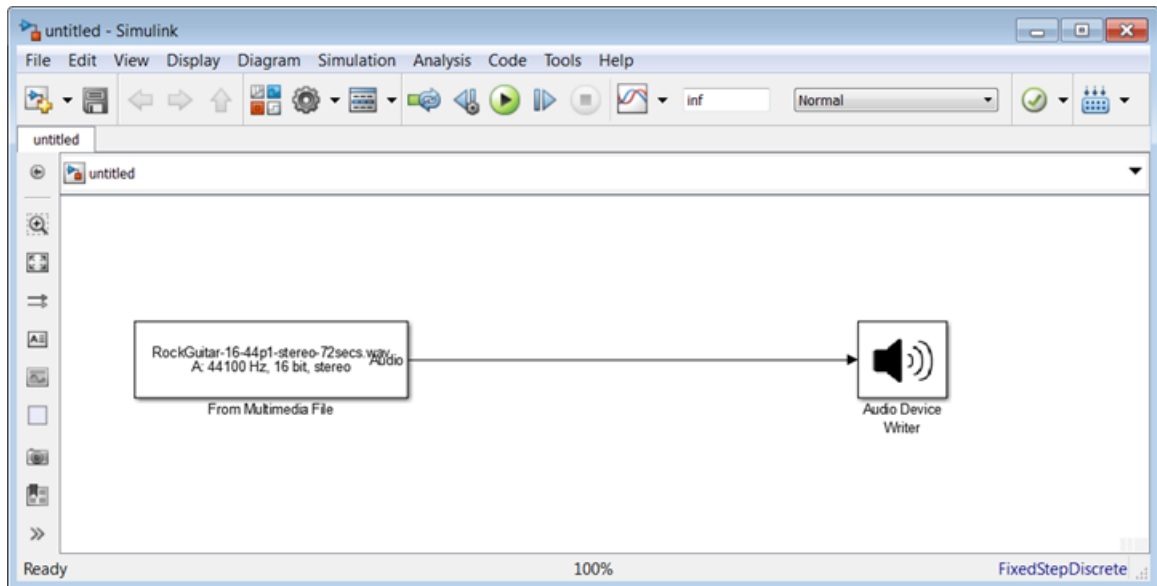
Add Audio System Toolbox Blocks to Model

1 Create a model using an Audio System Toolbox template.

2 Open the Simulink Library Browser and select Audio System Toolbox.
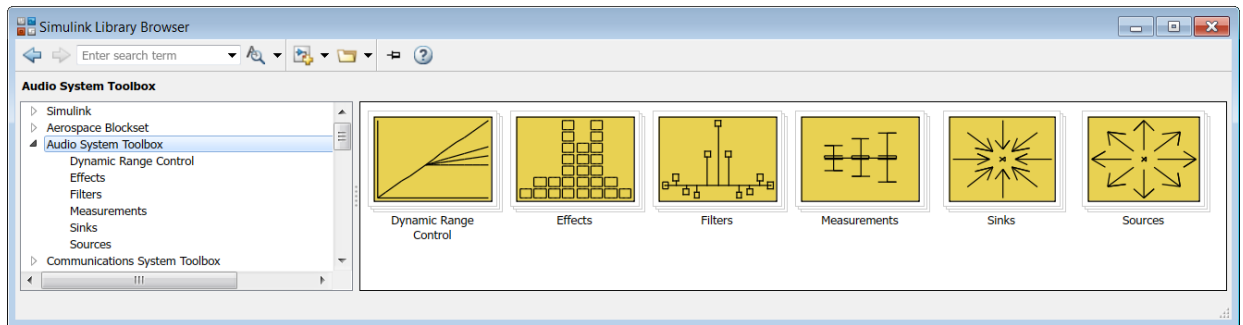
3 The Audio System Toolbox Block Library has six categories: Dynamic Range Control, Effects, Filters, Measurements, Sinks, and Sources. Select a block from one of the categories, and add it to your model.

4 In this example, a Compressor is added to the model by dragging and dropping from the Dynamic Range Control category of the Simulink Library Browser.
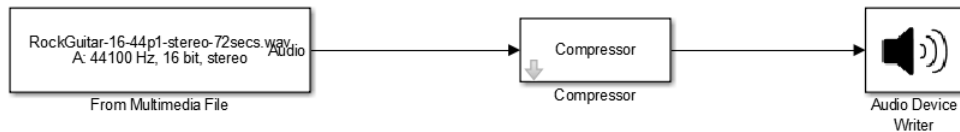
5 To run your model, click the ▶ button.

6 Open a block parameter user interface by double-clicking the block. You can modify parameters while the model runs. For example, if you added a Compressor block, you can adjust the **Threshold (dB)** dial to compress the dynamic range of your audio signal.
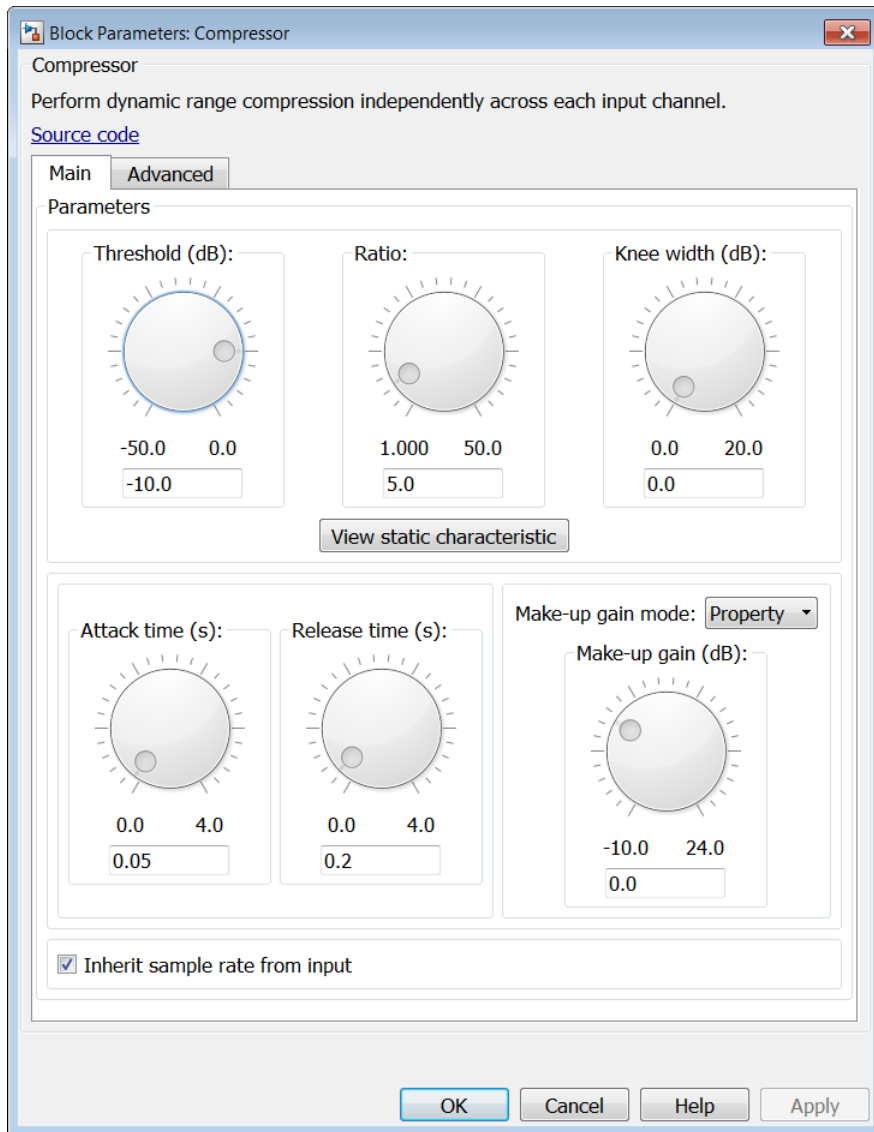
7 Running a model in the Simulink environment does not save the model. Save your model by clicking the 🖫 button.

Recommended Settings for Audio Signal Processing

Configuration Parameter	Setting
SingleTaskRateTransMsg	`error`
multiTaskRateTransMsg	`error`
Solver	`fixedstepdiscrete`
EnableMultiTasking	`Off`
StartTime	`0.0`
StopTime	`inf`
FixedStep	`auto`
SaveTime	`off`
SaveOutput	`off`
AlgebraicLoopMsg	`error`
SignalLogging	`off`
FrameProcessingCompatibilityMsg	`error`

More About

- "Real-Time Audio in MATLAB" on page 5-2
- "Convert Audio Plugin System Objects to Simulink Blocks" on page 9-2

8

Convert MATLAB Code to an Audio Plugin

Convert MATLAB Code to an Audio Plugin

Audio System Toolbox supports several approaches for the development of audio processing algorithms. Two common approaches include procedural programming using MATLAB scripts and object-oriented programming using MATLAB classes. The audio plugin class is the suggested paradigm for developing your audio processing algorithm in Audio System Toolbox. See "Design an Audio Plugin" on page 6-2 for a tutorial on the structure, benefits, and uses of audio plugins.

This tutorial presents an existing algorithm developed as a MATLAB script, and then walks through the steps to convert the script to an audio plugin class. Use this tutorial to understand the relationship between procedural programming and object-oriented programming. You can also use this tutorial as a template to convert any audio processing you developed as MATLAB scripts to the audio plugin paradigm.

Inspect Existing MATLAB Script

The MATLAB script has these sections:

A **Variable Initialization.**   Variables are initialized with known values, including the number of samples per frame (`frameSize`) for frame-based stream processing.

B **Object Construction.**

  - Two audioOscillator System objects — Construct to create time-varying gain control signals.
  - dsp.AudioFileReader System object — Construct to read an audio signal from a file.
  - audioDeviceWriter System object — Construct to write an audio signal to your default audio device.

C **Audio Stream Loop.**   Mixes stereo channels into a mono signal. The mono signal is used to create a new stereo signal. Each channel of the new stereo signal oscillates in applied gain between 0 and 2, with a respective 90-degree phase shift.

View Code

Click here to open this example.

```
%% Section A: Variable Initialization

% Specify frequency of gain oscillation.
Frequency = 1;
```

```matlab
% Determine sample rate of audio file (input audio signal).
fileInfo = audioinfo(...
    'RockGuitar-16-44p1-stereo-72secs.wav');
sampleRate = fileInfo.SampleRate;

% Specify size of frame to read in from audio file.
frameSize = 256;

%% Section B: Object Construction

Sine = audioOscillator(...
    'DCOffset',1,...
    'SamplesPerFrame',frameSize,...
    'Frequency',Frequency,...
    'SampleRate',sampleRate);

Cosine = audioOscillator(...
    'DCOffset',1,...
    'PhaseOffset',0.5,...
    'Frequency',Frequency,...
    'SamplesPerFrame',frameSize,...
    'SampleRate',sampleRate);

fileReader = dsp.AudioFileReader(...
    'Filename',fileInfo.Filename,...
    'SamplesPerFrame',frameSize);

deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

%% Section C: Audio Stream Loop

while ~isDone(fileReader)

    % Read in one frame of audio signal from file.
    in = fileReader();

    % Mix stereo input to mono.
    mono = 0.5*sum(in,2);

    % Get current frame of Sine and Cosine gain functions.
    gainLeft = Sine();
    gainRight = Cosine();
```

```
% Process signal by multiplying by variable gain matrix.
out = [mono,mono] .* [gainLeft,gainRight];

% Write one frame of audio signal to device.
deviceWriter(out);

end
```

Convert MATLAB Script to Plugin Class

This tutorial converts a MATLAB script to an audio plugin class in six steps. You begin by creating a skeleton of a basic audio plugin class, and then map sections of the MATLAB script to the audio plugin class.
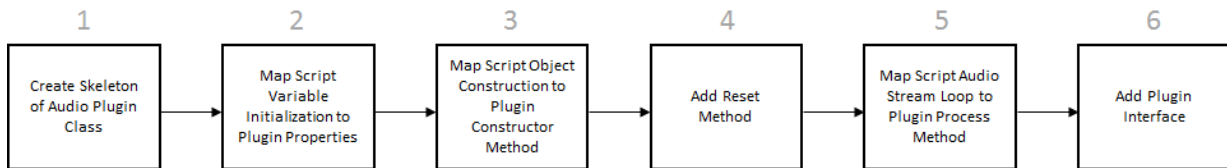
For an overview of how a MATLAB script is converted to a plugin class, inspect the script to plugin visual mapping. To perform this conversion, walk through the example for explanations and step-by-step instructions.

MATLAB Script

```
%% Section A: Variable Initialization
% Specify frequency of gain oscillation.
Frequency = 1;

% Determine sample rate of audio file.
fileInfo = audioinfo(...
    'RockGuitar-16-44p1-stereo-72secs.wav');
sampleRate = fileInfo.SampleRate;

% Specify frame size read from audio file.
frameSize = 256;

%% Section B: Object Construction
Sine = audioOscillator(...
    'DCOffset',1,...
    'Frequency',Frequency,...
    'SamplesPerFrame',frameSize,...
    'SampleRate',sampleRate);

Cosine = audioOscillator(...
    'DCOffset',1,...
    'PhaseOffset',0.5,...
    'Frequency',Frequency,...
    'SamplesPerFrame',frameSize,...
    'SampleRate',sampleRate);

fileReader = dsp.AudioFileReader(...
    'Filename',fileInfo.Filename,...
    'SamplesPerFrame',frameSize);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

%% Section C: Audio Stream Loop
while ~isDone(fileReader)
    in = fileReader();
    % Process audio signal.
    mono = 0.5*sum(in,2);
    gainLeft = Sine();
    gainRight = Cosine();
    out = [mono,mono].*[gainLeft,gainRight];
    deviceWriter(out);
end
```
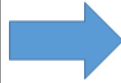
Plugin Class Definition

```
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Frequency',...
            'DisplayName','Oscillation Frequency',...
            'Label','Hz',...
            'Mapping',{'lin',0.01,10}))
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
            frameSize = size(in,1);

            plugin.Sine.Frequency = plugin.Frequency;
            plugin.Cosine.Frequency = plugin.Frequency;

            plugin.Sine.SamplesPerFrame = frameSize;
            plugin.Cosine.SamplesPerFrame = frameSize;

            mono = 0.5*sum(in,2);
            gainLeft = step(plugin.Sine);
            gainRight = step(plugin.Cosine);
            out = [mono,mono].*[gainLeft,gainRight];
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
            plugin.Cosine.SampleRate = getSampleRate(plugin)
        end
    end
end
```

1. Create Skeleton of Audio Plugin Class

Begin with the basic skeleton of an audio plugin class. This skeleton is not the minimum required, but a common minimum to create an interesting audio plugin. See "Design an Audio Plugin" on page 6-2 for the minimum requirements to create a basic audio plugin.

View Code

```
classdef gainOscillator < audioPlugin
    % gainOscillator Phase-shifted stereo gain oscillation.
    %   The process method mixes stereo channels into a mono signal. The
```

```
%   mono signal is used to create a stereo signal, with each channel
%   oscillating in gain between zero and two, with a respective 90
%   degree phase shift.

properties
    % Use this section to initialize properties that the end-user
    % interacts with.
end
properties (Access = private)
    % Use this section to initialize properties that the end-user does
    % not interact with directly.
end
properties (Constant)
    % This section contains instructions to build your audio plugin
    % interface. The end-user uses the interface to adjust tunable
    % parameters. Use audioPluginParameter to associate a public
    % property with a tunable parameter.
end
methods
    function out = process(plugin, in)
        % This section contains instructions to process the input audio
        % signal. Use plugin.MyProperty to access a property of your
        % plugin.
    end
    function reset(plugin)
        % This section contains instructions to reset the plugin
        % between uses or if the environment sample rate changes.
    end
end
end
```

2. Map Script Variable Initialization to Plugin Properties

Properties allow a plugin to store information across sections of the plugin class definition. If a property has access set to private, the property is not accessible to the end user of a plugin. Variable initialization in a script maps to plugin properties.

- A valid plugin must allow input to the `process` method to have a variable frame size. Frame size is determined for each input frame in the `process` method of the plugin. Because frame size is used only in the `process` method, you do not declare it in the properties section.

- A valid audio plugin must allow input to the `process` method to have a variable sample rate. The `reset` method of a plugin is called when the environment changes

the sample rate. Determine the sample rate in the `reset` method using the `getSampleRate` method inherited from the `audioPlugin` base class.

- The objects used by a plugin must be declared as properties to be used in multiple sections of the plugin. However, the constructor method of a plugin performs object construction.

View Code

```
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;  %<---
    end
    properties(Access = private)
        Sine            %<---
        Cosine          %<---
    end
    properties (Constant)
    end
    methods
        function out = process(plugin,in)
        end
        function reset(plugin)
        end
    end
end
```

3. Map Script Object Construction to Plugin Constructor Method

Add a constructor method to the methods section of your audio plugin. The constructor method of a plugin has the form:

```
function plugin = myPluginClassName
    % Instructions to construct plugin object.
end
```

If your plugin uses objects, construct them when the plugin is constructed. Set nontunable properties of objects used by your plugin during construction.

In this example, you construct the `Sine` and `Cosine` objects in the constructor method of the plugin.

View Code

```
classdef gainOscillator < audioPlugin
    properties
```

```matlab
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator        %<---
            plugin.Sine = audioOscillator(...    %<---
                'DCOffset',1);                   %<---
            plugin.Cosine = audioOscillator(...  %<---
                'DCOffset',1,...                 %<---
                'PhaseOffset',0.5);              %<---
        end                                      %<---
        function out = process(plugin,in)
        end
        function reset(plugin)
        end
    end
end
```

4. Add Reset Method

The `reset` method of a plugin is called every time a new session is started with the plugin, or when the environment changes sample rate. Use the `reset` method to update the `SampleRate` property of your `Sine` and `Cosine` objects. To query the sample rate, use the `getSampleRate` base class method.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator
```

```matlab
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);     %<---
            plugin.Cosine.SampleRate = getSampleRate(plugin);   %<---
        end
    end
end
```

5. Map Script Audio Stream Loop to Plugin Process Method

The contents of the audio stream loop in a script maps to the `process` method of an audio plugin, with these differences:

- A valid audio plugin must accept a variable frame size, so frame size is calculated for every call to the `process` method. Because frame size is variable, any processing that relies on frame size must update when input frame size changes.
- The environment handles the input and output to the `process` method.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
```

```
        end
        function out = process(plugin,in)
            frameSize = size(in,1);                        %<---

            plugin.Sine.SamplesPerFrame = frameSize;       %<---
            plugin.Cosine.SamplesPerFrame = frameSize;     %<---

            mono = 0.5*sum(in,2);                          %<---
            gainLeft = step(plugin.sine);                  %<---
            gainRight = step(plugin.cosine);               %<---
            out = [mono,mono].*[gainLeft,gainRight];       %<---
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
            plugin.Cosine.SampleRate = getSampleRate(plugin);
        end
    end
end
```

6. Add Plugin Interface

The plugin interface lets users view the plugin and tune its properties. Specify
`PluginInterface` as an `audioPluginInterface` object that contains an
`audioPluginParameter` object. The first argument of `audioPluginParameter` is the
property you want to synchronize with a tunable parameter. Choose the name to display,
label the units, and set the parameter range. This example uses 0.1 to 10 as a reasonable
range for the `Frequency` property. Write code so that during each call to the `process`
method, your `Sine` and `Cosine` objects are updated with the current frequency value.

View Code

```
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...             %<---
            audioPluginParameter('Frequency',...               %<---
            'DisplayName','Oscillation Frequency',...          %<---
            'Label','Hz',...                                   %<---
```

```
            'Mapping',{'lin',0.01,10}))                        %<---
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
            frameSize = size(in,1);

            plugin.Sine.Frequency = plugin.Frequency;        %<---
            plugin.Cosine.Frequency = plugin.Frequency;      %<---

            plugin.Sine.SamplesPerFrame = frameSize;
            plugin.Cosine.SamplesPerFrame = frameSize;

            mono = 0.5*sum(in,2);
            gainLeft = step(plugin.Sine);
            gainRight = step(plugin.Cosine);
            out = [mono,mono].*[gainLeft,gainRight];
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
            plugin.Cosine.SampleRate = getSampleRate(plugin);
        end
    end
end
```

Click here to open the completed plugin example.

Once your audio plugin class definition is complete:

1 Save your plugin class definition file.
2 Validate your plugin using `validateAudioPlugin`.
3 Prototype it using Audio Test Bench.
4 Generate is using `generateAudioPlugin`.

More About

• "Real-Time Audio in MATLAB" on page 5-2

Convert Audio Plugin System Objects to Simulink Blocks

Convert Audio Plugin System Objects to Simulink Blocks

You can convert System object audio plugins to blocks for real-time parameter tuning in Simulink. Use this workflow to convert your own System object plugins to Simulink blocks, or to convert any of the System object plugins found in the "Audio Plugin Example Gallery".

Open the Basic Audio Player Template in Simulink

On the Simulink Start Page, under Audio System Toolbox, click **Basic Audio Player**. See "Real-Time Audio in Simulink" on page 7-2 for a tutorial on using Simulink model templates.
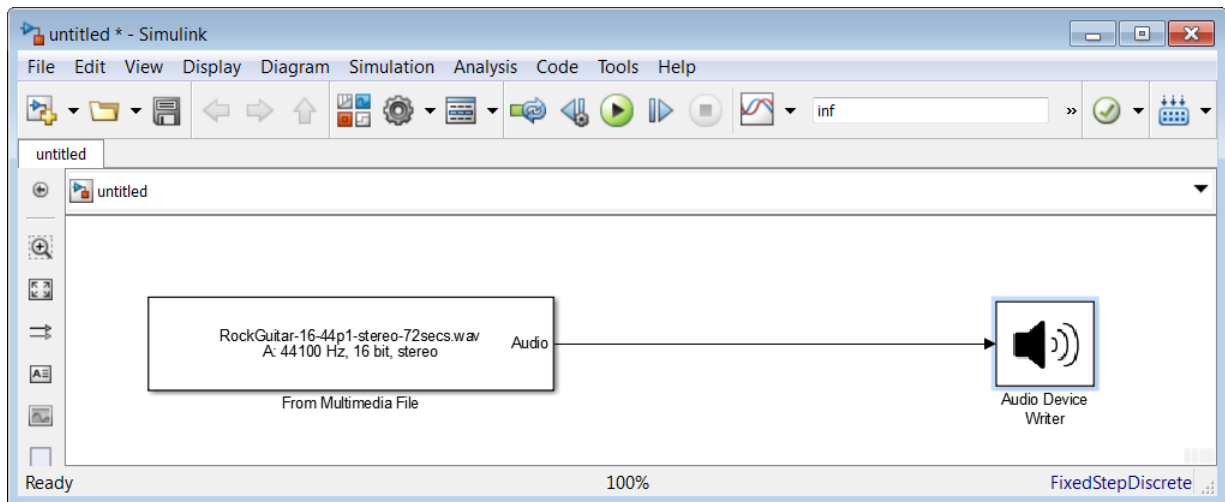
Import Audio Plugin Functionality

To import System object plugins into Simulink, use the MATLAB System block. This block is compatible with System object plugins but not basic plugins. See "Design an Audio Plugin" on page 6-2 for more information about defining plugins in MATLAB.

1   Add the System object plugin used in this example to the MATLAB path. At the command prompt, enter:

```
addpath(fullfile(matlabroot,'/examples/audio'))
```

2   From the Simulink / User-Defined Functions library, drag a MATLAB System block to your model.

3   In the MATLAB System block, enter the name of your System object: `SoundPosition`
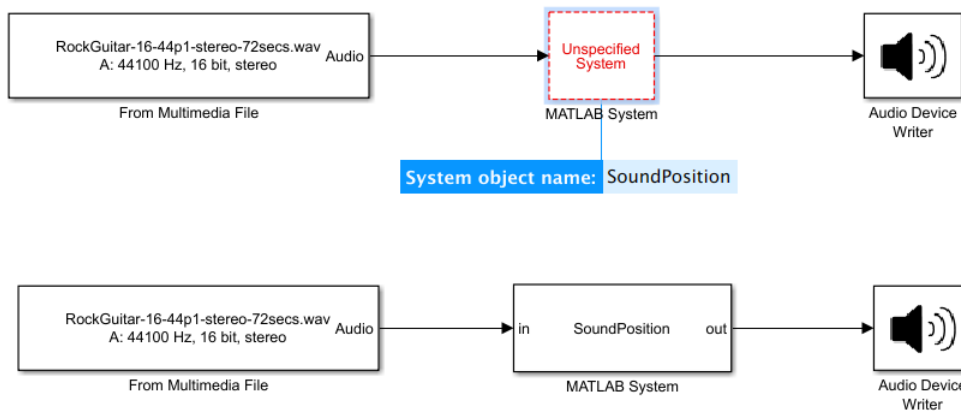
The `SoundPosition` audio plugin enables you to tune two parameters: stereo width, and panning.

Create an Audio Plugin Block Interface

When you import a plugin into a Simulink model, the plugin parameters are set to the initial values defined in the properties section of the plugin class. To use dials for tunable parameters, create a custom interface by using a block mask. See "Masking Fundamentals" (Simulink) for more information.

1   Open the SoundPosition block.

    a   Set **Width** to the variable W.

    b   Set **Pan** to the variable P.

Parameters	
Width:	W
Pan:	P
PluginInterface:	audioPluginInterface('InputChann
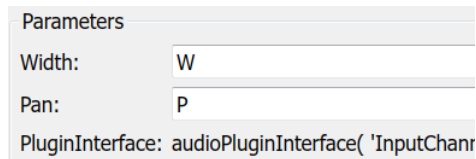
   c   Click **OK**.

2   Make your SoundPosition block a subsystem. With the SoundPosition block selected, from the Simulink Editor menu, select **Diagram** > **Subsystem & Model Reference** > **Create Subsystem from Selection**.

RockGuitar-16-44p1-stereo-72secs.wav
A: 44100 Hz, 16 bit, stereo   Audio

From Multimedia File

In1   Out1

Subsystem

Audio Device
Writer
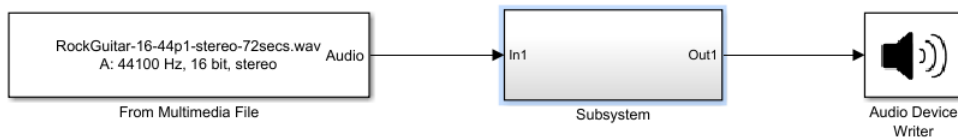
3   Add a mask to your Subsystem block. Select your Subsystem block, and then from the Simulink Editor menu, select **Diagram** > **Mask** > **Create Mask**.

4   In the Mask Editor, click the **Parameters & Dialog** tab.

5   Add a dial to the dialog box for controlling stereo width. From the **Controls** pane, drag a **Dial** to the **Dialog box** pane. Then, in the **Property editor** pane, set these properties:

- **Name** — W
- **Value** — 2
- **Prompt** — Stereo width
- **Type** — dial
- **Minimum** — 0
- **Maximum** — 4
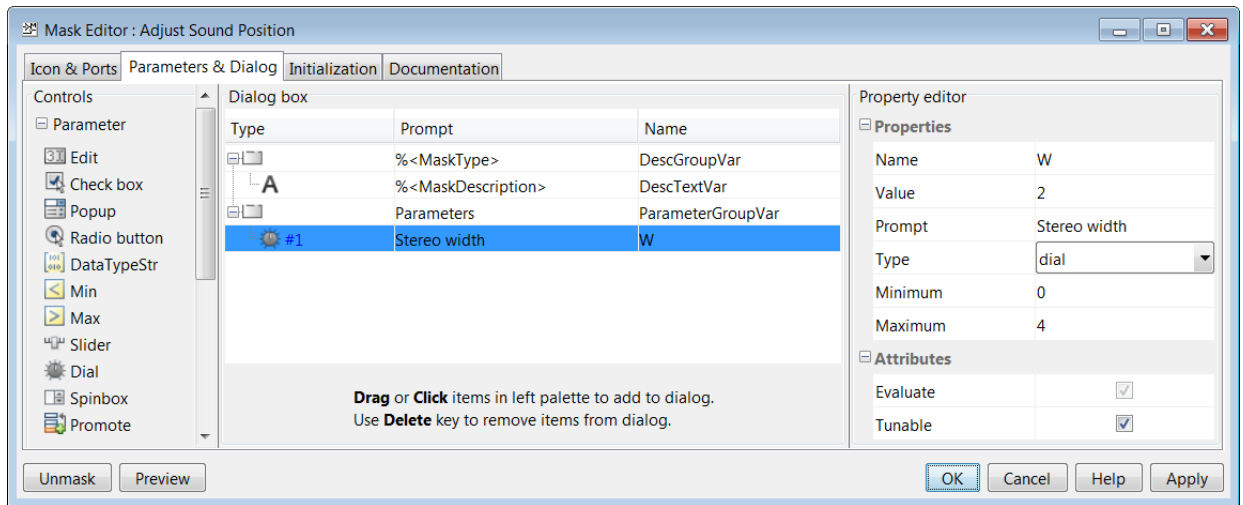
6 To control the panning, add another dial to the dialog box. From the **Controls** pane, drag a **Dial** to the **Dialog box** pane. Then, in the **Property editor** pane, set these properties:

- **Name** — P
- **Value** — 0
- **Prompt** — Pan
- **Type** — dial
- **Minimum** — -1
- **Maximum** — 1
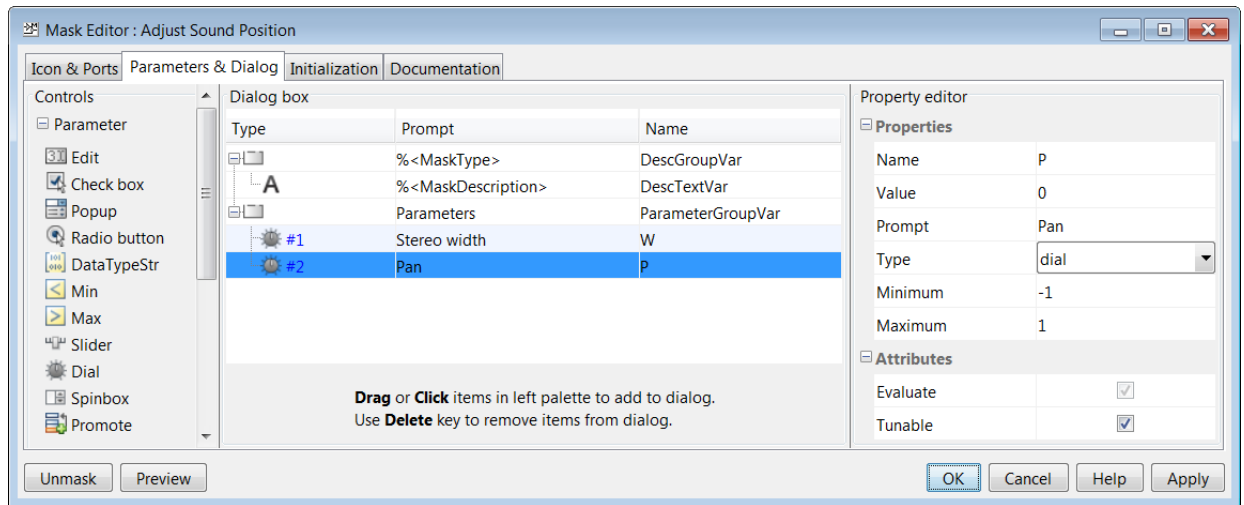
7 Click **OK**.

Run the Model

1 Open the From Multimedia File block.

    a To modify the frame size used in your model, set **Samples per audio channel** to `256`.

    b To hear the effect of the stereo widening, specify an audio file with a distinct stereo field recording. Set **File name** to `FunkyDrums-44p1-stereo-25secs.mp3`.

    c Click **OK**.

2 To open the parameter controls of your SoundPosition block, double-click the Subsystem block.
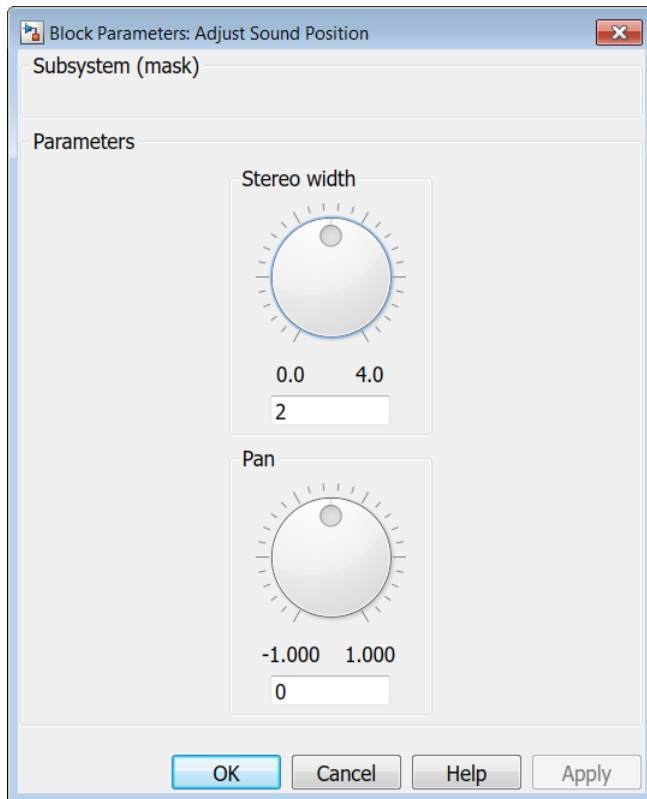
3 Run your model. To hear the effect of your audio plugin, open the Subsystem block and modify the **Stereo width** and **Pan** parameters in real time.

Open the completed model.

After you complete this tutorial, it is a best practice to undo the modification to the MATLAB path. At the command prompt, enter:

```
rmpath(fullfile(matlabroot,'/examples/audio'))
```

More About

·   "Design an Audio Plugin" on page 6-2

·   "Audio Plugin Example Gallery"

- "Real-Time Audio in Simulink" on page 7-2

Host External Audio Plugins

Host External Audio Plugins

You can host VST and VST3 plugins in MATLAB by using the `loadAudioPlugin` function from Audio System Toolbox.

MATLAB

External Audio Plugin
VST or VST3

`loadAudioPlugin`

Hosted Plugin Object
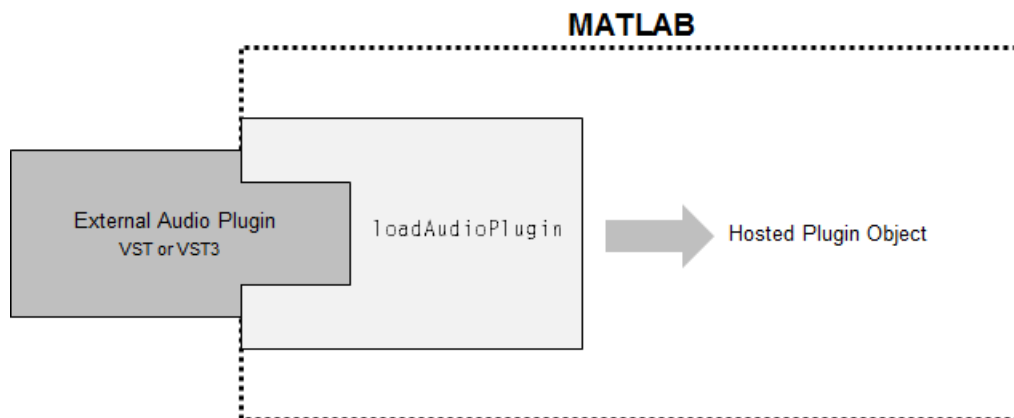
After you load an external audio plugin, you interact with it by setting and getting parameters. You process audio through its main audio processing algorithm.

MATLAB

Get Parameter

Parameter X   Parameter X Value

Hosted Plugin Object
- Parameter 1, Parameter 1 Value
- Parameter 2, Parameter 2 Value
  ⋮
- Parameter N, Parameter N Value

Audio Signal Input Frame ⟶   ⟶ Audio Signal Output Frame

Parameter X, Parameter X Value
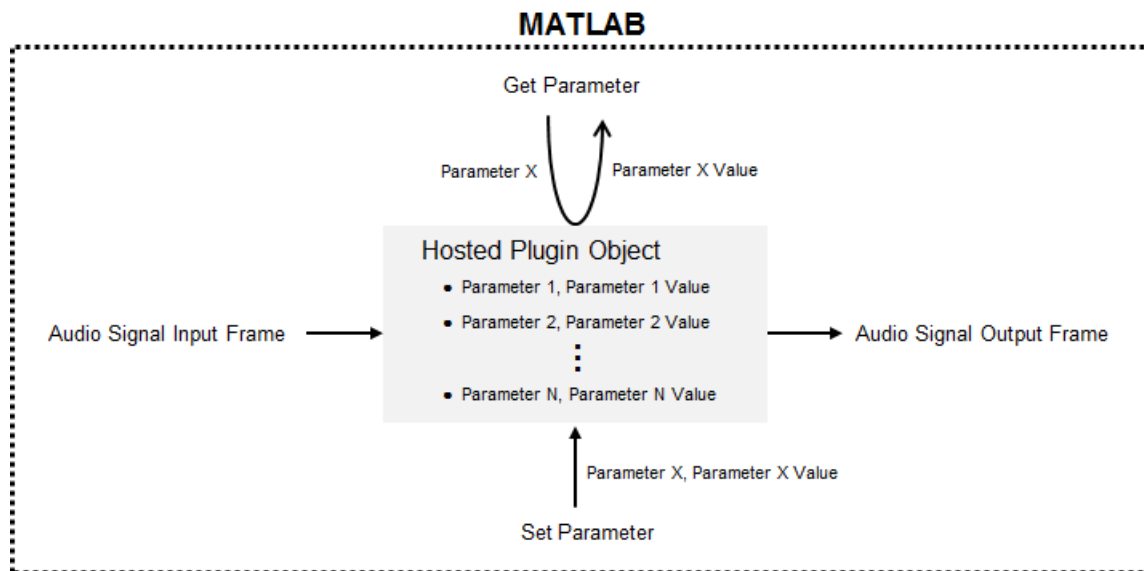
Set Parameter

This tutorial walks you through the process of hosting an externally authored VST plugin and interacting with the plugin at the MATLAB command line:

1 Load the plugin.
2 Get and set plugin parameters.
3 Use the plugin to process audio.
4 Analyze the plugin.

In this tutorial, you host a plugin from the suite of ReaPlugs VST plugins distributed by Cockos Incorporated. To download the ReaPlugs VST FX Suite for your system, follow the instructions on the REAPER website. A 64-bit Windows platform is used in this tutorial. The `loadAudioPlugin` function cannot load 32-bit plugins.

You can also interact with an externally authored VST audio plugin graphically, using the **Audio Test Bench**. See the Audio Test Bench documentation for examples.

Host External Audio Plugin Tutorial

1. Load External Audio Plugin

Use the `loadAudioPlugin` function to host the `ReaDelay` VST plugin. Specify the complete path to the plugin or place the plugin in your current folder. In this example, the plugin is placed in the current folder and loaded by specifying the file name only.

```
hostedPlugin = loadAudioPlugin('readelay-standalone.dll');
```

Call the hosted plugin to display basic information:

```
hostedPlugin
```

```
hostedPlugin =

  VST plugin 'ReaDelay (ReaPlugs Edition)'  2 in, 2 out

          Parameter    Value     Display
         _____
      1          Wet:  1.0000    +0.0 dB
      2          Dry:  1.0000    +0.0 dB
      3   1: Enabled:  1.0000      ON
      4    1: Length:  0.0000    0.0 ms
```

```
    5    1: Length:    0.0156    4.00 8N
 7 parameters not displayed. Use dispParameter(hostedPlugin) to see all 12 params.
```

The first line displays the plugin type, plugin display name, and the number of input and output channels for the main audio processing algorithm of the plugin. If you are hosting a source plugin, the number of output channels and the default samples per frame are displayed.

By default, only the first five parameters are displayed. To display all parameters of the hosted plugin, click `See all 12 params`.

The table provides the parameter index, parameter name, normalized parameter value, displayed parameter value, and the displayed parameter value label.

The *normalized parameter value* is always in the range [0,1] and generally corresponds to the position of a user interface (UI) widget in a DAW or the position of a MIDI control on a MIDI control surface. The *parameter display value* is related to the normalized parameter value by an unknown mapping internal to the plugin, and typically reflects the value used internally by the plugin for processing.
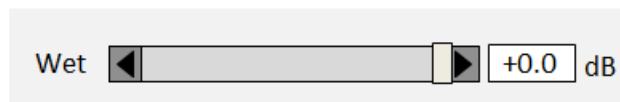
2. Set and Get Hosted Plugin Parameter Values

You can use `getParameter` and `setParameter` to interact with the parameters of the hosted plugin. Using `getParameter` and `setParameter` is the programmatic equivalent of moving widgets in a UI or controls on a MIDI control surface. A typical DAW UI provides the parameter name, a visual representation of the normalized parameter value, the displayed parameter value, and the displayed parameter value label.
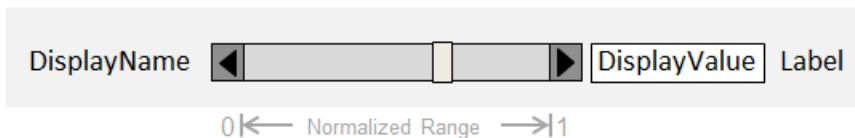
For example, the `Wet` parameter of `readelay-standalone.dll` has a normalized parameter value of `1` and a display parameter value of `+0.0`. The `Wet` parameter might be displayed in a DAW as follows:

With Audio System Toolbox, you can use `getParameter` to return the normalized parameter value and additional information about a single hosted plugin parameter. You can specify which parameter to get by the parameter index.

```
parameterIndex = 1;
[normParamValue,paramInfo] = getParameter(hostedPlugin,parameterIndex)


normParamValue =

    1


paramInfo =

  struct with fields:

    DisplayName: 'Wet'
```

```
           DisplayValue: '+0.0'
                 Label: 'dB'
```

You can use `setParameter` to set a normalized parameter value of your hosted plugin. You can specify which parameter to set by its parameter index.

```
normParamValue = 0.5;
setParameter(hostedPlugin,parameterIndex,normParamValue)
```

Setting the normalized parameter value to 0.5 is equivalent to setting the indicator to the center of a slider in a DAW.
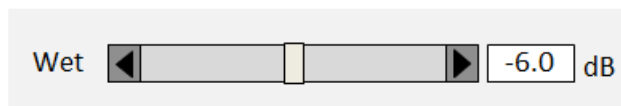
To verify the new normalized parameter value for `Wet`, use `getParameter`.

```
parameterIndex = 1;
[normParamValue,paramInfo] = getParameter(hostedPlugin,parameterIndex);
```

The `DisplayValue` for the `Wet` parameter updates from `+0.0` to `-6.0` because you set the corresponding normalized parameter value. The relationship between the displayed value and the normalized value is determined by an unknown mapping that is internal to the hosted plugin.

3. Use Hosted Plugin to Process Audio

To process an audio signal with the hosted plugin, use `process`.

```
audioIn = [1,1];
audioOut = process(hostedPlugin,audioIn);
```

Audio plugins are designed for variable-frame-based processing, meaning that you can call `process` with successive audio input frames of different lengths. The hosted plugin saves the internal states required for continuous processing. To process an audio signal read from a file and then written to your audio output device, place your hosted plugin in an audio stream loop. Use dsp.AudioFileReader and audioDeviceWriter objects as the input and output to your audio stream loop, respectively. Set the sample rate of the hosted plugin to the sample rate of the audio file by using `setSampleRate`.

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

```
sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
setSampleRate(hostedPlugin,sampleRate);

while ~isDone(fileReader)
    audioIn = fileReader();

    % The hosted plugin requires a stereo input.
    stereoAudioIn = [audioIn,audioIn];

    x = process(hostedPlugin,stereoAudioIn);

    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

You can modify parameters in the audio stream loop. To control the `Wet` parameter of your plugin in an audio stream loop, create an audioOscillator System object™. Use the `fileReader`, `deviceWriter`, and `hostedPlugin` objects you created previously to process the audio.

```
osc = audioOscillator('sine',...
    'Frequency',10,...
    'Amplitude',0.5,...
    'DCOffset',0.5',...
    'SamplesPerFrame',fileReader.SamplesPerFrame,...
    'SampleRate',sampleRate);

while ~isDone(fileReader)
    audioIn = fileReader();

    controlSignal = osc();
    setParameter(hostedPlugin,1,controlSignal(1));

    stereoAudioIn = [audioIn,audioIn];
    x = process(hostedPlugin,stereoAudioIn);
    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

4. Analyze Hosted Plugin

You can use the Audio System Toolbox measurement and visualization tools to display behavior information about your hosted plugin. To display the input and output of your hosted audio plugin, create a dsp.TimeScope object. Create a loudnessMeter object and use the 'EBU Mode' visualization to monitor loudness output by the hosted plugin. Use the `fileReader`, `deviceWriter`, `osc`, and `hostedPlugin` objects you created previously to process the audio.

```
scope = dsp.TimeScope(...
    'SampleRate',sampleRate,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',5,...
    'BufferLength',5*2*sampleRate,...
    'YLimits',[-1 1]);

loudMtr = loudnessMeter('SampleRate',sampleRate);
visualize(loudMtr)

while ~isDone(fileReader)
    audioIn = fileReader();

    controlSignal = osc();
    setParameter(hostedPlugin,1,controlSignal(1));

    stereoAudioIn = [audioIn,audioIn];
    x = process(hostedPlugin,stereoAudioIn);

    loudMtr(x);
    scope([x(:,1),audioIn(:,1)])

    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
release(scope)
release(loudMtr)
```
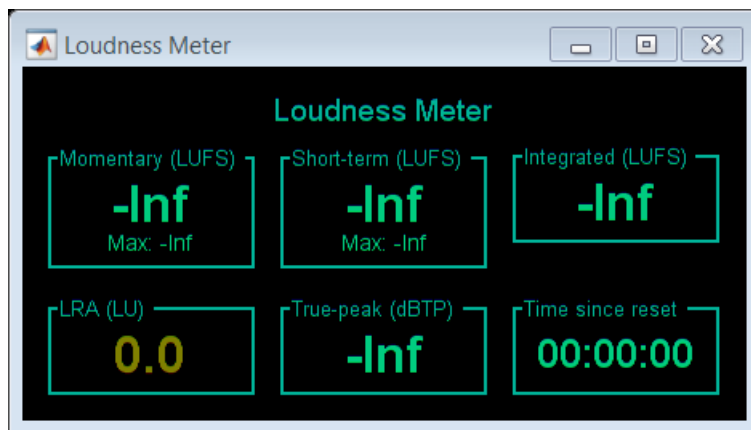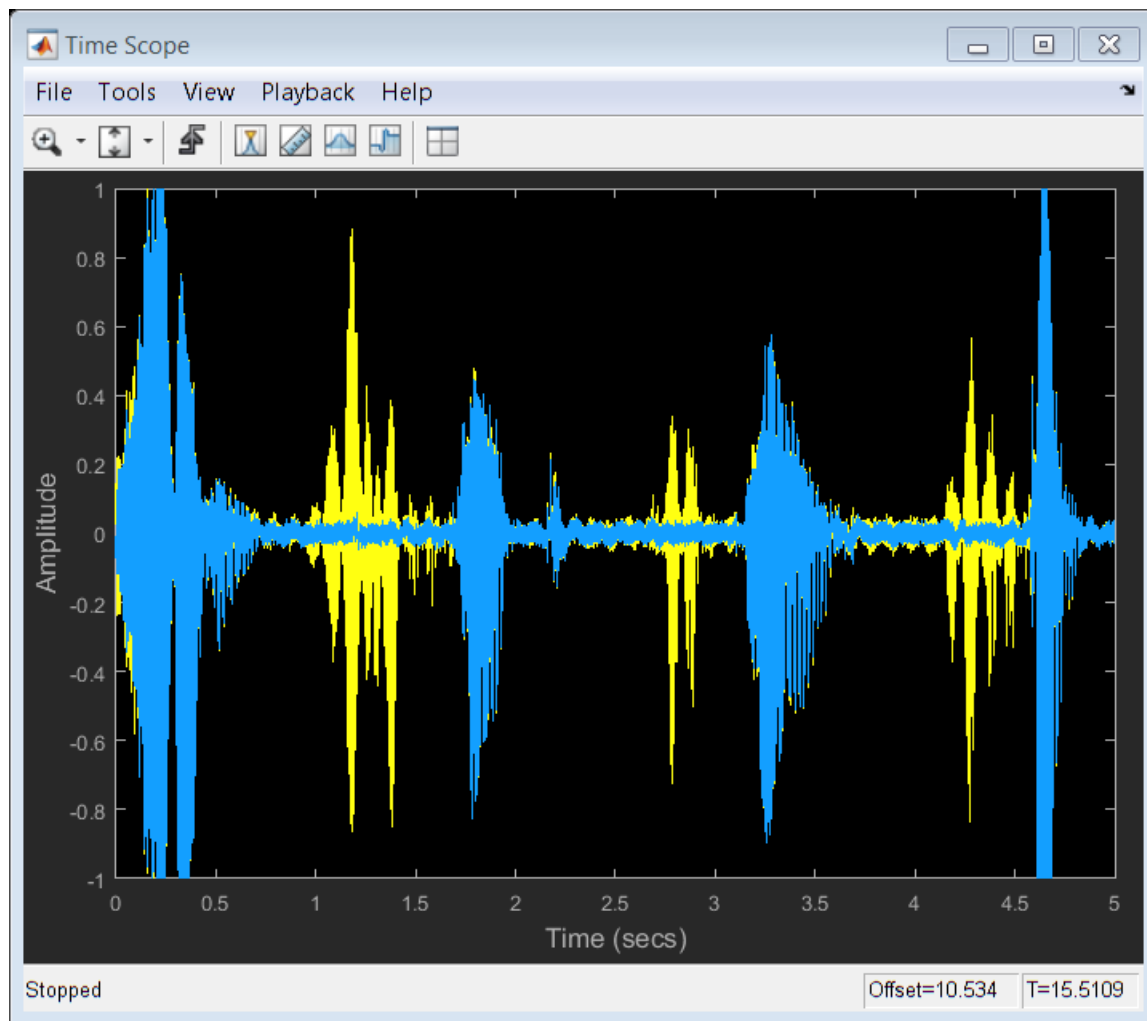
You can investigate the relationship between the normalized hosted plugin parameter values and the displayed values by creating a sweeping function.

For example, save the `displayParameterMapping` function to your current folder. This function is for example purposes and might not perform as expected with all hosted plugins.

```matlab
function displayParameterMapping(hPlugin,prmIndx)
x = 0:0.001:1; % Normalized parameter range

[~,prmInfo] = getParameter(hPlugin,prmIndx);
if isnan(str2double(prmInfo.DisplayValue))
    % Non-Numeric Displays - prints normalized parameter range associated
    % with string
    setParameter(hPlugin,prmIndx,0);
    [~,prmInfo] = getParameter(hPlugin,prmIndx);
    txtOld = prmInfo.DisplayValue;
    oldIndx = 1;

    for i = 2:numel(x)
        setParameter(hPlugin,prmIndx,x(i))
        [~,prmInfo] = getParameter(hPlugin,prmIndx);
        txtNew = prmInfo.DisplayValue;
        if ~strcmp(txtNew,txtOld)
            fprintf('%s: %g - %g\n',txtOld, x(oldIndx),x(i-1));
            oldIndx = i;
            txtOld = txtNew;
        end
    end
    fprintf('%s: %g - %g\n',txtOld, x(oldIndx),x(i));
else
    % Numeric Displays - plots normalized parameter value against displayed
    % parameter value
    y = zeros(numel(x),1);
    for i = 1:numel(x)
        setParameter(hPlugin,prmIndx,x(i))
        [~,prmInfo] = getParameter(hPlugin,prmIndx);
        y(i) = str2double(prmInfo.DisplayValue);
    end
    if any(isnan(y))
        warning('NaN detected in numeric display.')
    end
    plot(x,y)
    xlabel('Normalized Parameter Value')
    ylabel(['Displayed Parameter Value (',prmInfo.Label,')'])
    title(prmInfo.DisplayName)
end

end
```
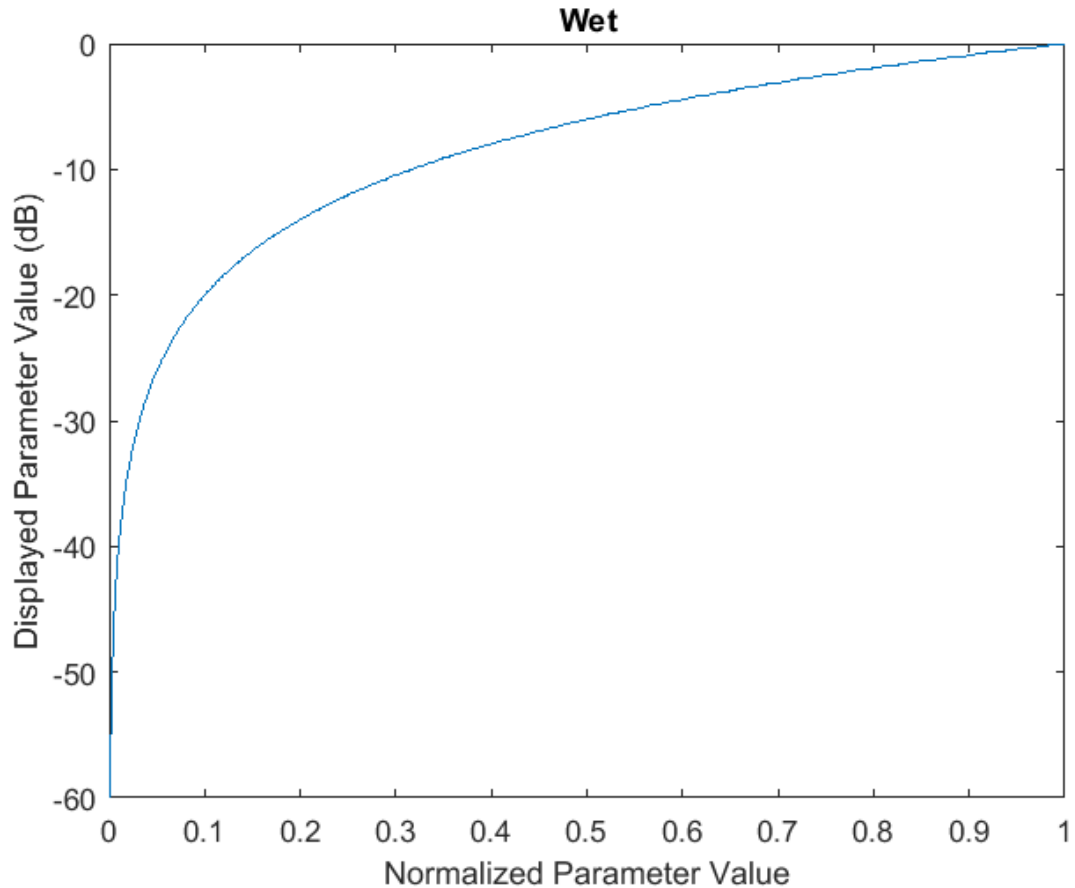
Call the `displayParameterMapping` function with the hosted plugin and a parameter index.

```
displayParameterMapping(hostedPlugin,1)
```


If you use the `displayParameterMapping` function with a nonnumeric parameter, the relationship displays in the Command Window:

```
displayParameterMapping(hostedPlugin,3)
```

```
OFF: 0 - 0.499
```

```
ON: 0.5 - 1
```

See Also

Functions
loadAudioPlugin

Classes
externalAudioPlugin | externalAudioPluginSource

More About

- "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 4-2
- "Design an Audio Plugin" on page 6-2