Sample Code

Creating an audio device driver

Implement a configurable audio input source as a driver extension that runs in user space in macOS and iPadOS.

Download

iOS 16.0+ | iPadOS 16.0+ | macOS 12.1+ | Xcode 16.0+

Overview

> **Note**
>
> This sample code project is associated with WWDC21 session Create audio drivers with DriverKit. This version updates the sample to run in iPadOS, as discussed in WWDC22 session Bring your driver to iPad with DriverKit.

This sample shows how to create an audio driver extension using the AudioDriverKit framework. The sample provides a C++ DriverKit implementation to publish a single audio device, output stream, input stream, volume control, and data-source selector control.

The sample implements a dynamic environment that can support multiple audio devices and any other audio objects the AudioDriverKit framework provides. The audio device provides the following features:

- A configurable input device volume
- A sine tone generator for the input stream's I/O buffer
- An output stream loopback to the input stream data-source selector control
- A sine tone frequency data-source selector control

- 44.1 and 48 kHz sample rates
- A mono channel of audio I/O in 16-bit, linear PCM format
- An example of a string-based custom property

AudioDriverKit is available in macOS, and in iPadOS 16 and later when running on an iPad device with an M-series chip. This sample project supports both platforms.

The sample app connects to the audio driver extension through a custom user client connection. The custom user client shows an example of how to change the data-source selector value or the sample rate directly on the audio driver extension. In macOS, the sample app also provides the installer for the driver. In iPadOS, you install the driver by enabling it in Settings.

Configure the sample code project

By default, the sample project uses manual code signing. If you just want to run the app to see how it works, select "Automatically manage signing" for each of the targets.

If you want to run the app with manual signing, do the following:

1. Create new bundle identifiers for the app and driver. The bundle identifiers included with the project already have App IDs associated with them, so you need unique identifiers to create your own App IDs. Use a reverse-DNS format for your identifiers, as described in Preparing Your App For Distribution. iPadOS also requires that your driver's bundle identifier begin with the host app's bundle identifier. For example, the default bundle identifiers are `com.example.apple-samplecode.SimpleAudio` for the app and `com.example.apple-samplecode.SimpleAudio.Driver` for the driver.
2. In the Xcode project, click the Signing & Capabilities tab for each of the three targets — driver, macOS app, and iOS app — and set the respective bundle identifier.
3. In the driver's `Info.plist` file, set the value of the `IOUserServerName` to the driver bundle identifier.
4. In `SimpleAudioDriverViewModel.swift`, make sure the string concatentation that initializes `dextIdentifier` matches the bundle identifer for the driver.
5. The sample app needs an explicit App ID and provisioning profile with the entitlements System Extension and Communicates with Drivers. For information about how to request entitlements, see Requesting Entitlements for DriverKit Development.
6. The sample driver needs an explicit App ID and provisioning profile with the following entitlements: `com.apple.developer.driverkit`, `com.apple.developer.driverkit.family.audio`, and `com.apple.developer.driverkit.allow-any-userclient-access`. This latter macOS-only entitlement allows any app to connect to the driver as a user client. Although this simplifies running the sample code, in your own apps you may prefer to use

`com.apple.developer.driverkit.userclient-access`. This entitlement goes on the app rather than the driver, and lists bundle identifiers of drivers it can connect to. If you don't intend for your driver to allow user client connections, just use the `com.apple.developer.driverkit.family.audio` entitlement.

7. For each of the App IDs you create in the previous steps, select Profiles to create a new provisioning profile. You need one for the macOS app, one for the iPadOS app, and one for the driver, which supports both macOS and iPadOS. When creating the driver's profile, be sure to select DriverKit App Development as the profile type.
8. Download each profile and add it to Xcode.
9. On the Signing & Capabilities tab, set each target to manual code signing and select its new profile.

Run the sample in macOS

To run the sample app in macOS, use the scheme selector to select the SimpleAudio (macOS) scheme and the My Mac destination. Build the target, then copy the app to the Applications folder and launch the app.

> **Note**
>
> You can run the app directly from Xcode, without moving the app bundle to `/Applications` each time, by using the `systemextensionsctl` command to enable system extensions developer mode, as explained in Debugging and testing system extensions.

In macOS, the SimpleAudio app has two sections: Driver Manager, which installs the app, and User Client Manager, which interacts with the running driver. Under Driver Manager, click Install Driver. If a System Extension Blocked dialog appears, open System Settings and navigate to the Security & Privacy pane. Unlock the pane if necessary and click Allow to complete the installation. When installation completes, the Driver Manager status in the app displays the message "SimpleAudioDriver has been activated and is ready to use."

At this point, the sample's audio device is available to Core Audio. To inspect the newly installed device, use the Audio MIDI Setup app (`Applications/Utilities`), which shows the sine tone's frequency and sample rate. You can change these settings there, or in the SimpleAudio app's User Client Manager section. Click Open User Client to open a connection from the app to the driver. Then you can use the other buttons in this section to toggle the frequency and sample rate.

To hear the sine tone, open the QuickTime Player app and choose File > New Audio Recording to create a new recording window. Next to the Record button, change the device from the default

microphone to SimpleAudioDriver: Sine Tone 440 or SimpleAudioDriver: Sine Tone 660. Adjust the volume slider to hear the tone through your current audio output device.

To uninstall the driver, delete the sample app, which also stops and removes the driver extension (dext). You can also use `systemextensionsctl` from the command line to list and selectively uninstall system extensions like `SimpleAudioDriver`.

Run the sample in iPadOS

To run the sample app in iPadOS, connect an iPad device with an M-series chip to your Mac. Use the scheme selector to select the SimpleAudio (iOS) scheme and the name of your iPad as the destination. Run the app directly from Xcode to launch it on your iPad.

In iPadOS, the SimpleAudio app doesn't show the Driver Manager section because the app isn't responsible for installing the driver like it is in macOS. Instead, open the Settings app, navigate to Privacy & Security > Drivers, and enable the driver there.

After enabling the driver, return to the SimpleAudio app to open a user client connection and modify the device's frequency and sample rate.

When you finish using the driver, delete the app, which deletes the driver as well.

Create driver and device classes

To create an AudioDriverKit driver, the sample creates a driver that subclasses `IOUserAudioDriver`, and a device that subclasses `IOUserAudioDevice`. The dext's `Info.plist` file contains entries that identify the driver class to AudioDriverKit, which instantiates and initializes the driver. The sample's `Info.plist` file shows how this works: the `IOUserClass` key maps to the class name string `SimpleAudioDriver`, and `IOUserServerName` contains the bundle ID.

The driver subclass is the entry point into the dext, while the device subclass handles start and stop I/O-related messages, timestamps, and configuration messages. The device also owns various `IOUserAudioObject` instances for things like timer dispatch sources and `OSAction` references. In an actual hardware driver, the device class is also responsible for communication with the hardware over USB or PCI, and requires appropriate DriverKit entitlements for those transports. The sample doesn't actually connect to hardware, and instead provides a virtual device that generates a sine tone.

> **Note**
>
> When creating a virtual device, best practice is to use an Audio Server Driver Plug-in instead, as described in Creating an Audio Server Driver Plug-in. AudioDriverKit only supports physical audio devices.

After initialization, DriverKit calls the driver's `Start` method. The implementation in `SimpleAudioDriver` creates and configures the `SimpleAudioDevice` instance and, if successful, calls `RegisterService` to let the system know the driver is running.

```
kern_return_t SimpleAudioDriver::Start_Impl(IOService* in_provider)
{
    bool success = false;
    auto device_uid = OSSharedPtr(OSString::withCString(kSimpleAudioDriverDeviceUID)
    auto model_uid = OSSharedPtr(OSString::withCString("SimpleAudioDevice-Model"), O
    auto manufacturer_uid = OSSharedPtr(OSString::withCString("Apple Inc."), OSNoRet
    auto device_name = OSSharedPtr(OSString::withCString("SimpleAudioDevice"), OSNoR

    kern_return_t error = Start(in_provider, SUPERDISPATCH);
    FailIfError(error, , Failure, "Failed to start Super");

    // Get the service's default dispatch queue from the driver object.
    ivars->m_work_queue = GetWorkQueue();
    FailIfError(ivars->m_work_queue.get() == nullptr, error = kIOReturnInvalid, Fail

    // Allocate and configure audio devices as necessary.
    ivars->m_simple_audio_device = OSSharedPtr(OSTypeAlloc(SimpleAudioDevice), OSNoP
    FailIfNULL(ivars->m_simple_audio_device.get(), error = kIOReturnNoMemory, Failur

    success = ivars->m_simple_audio_device->init(this, false, device_uid.get(), mode
    FailIf(success == false, error = kIOReturnNoMemory, Failure, "Failed to init Sim

    ivars->m_simple_audio_device->SetName(device_name.get());

    // Add the device object to the driver.
    AddObject(ivars->m_simple_audio_device.get());

    // Register the service.
    error = RegisterService();
    FailIfError(error, , Failure, "failed to register service!");

    return kIOReturnSuccess;

Failure:
    return error;
}
```

Implement a user client interface

There are two dictionaries in the `Info.plist` file that define how the driver acts as a user client to the Core Audio Hardware Abstraction Layer (HAL) and to other apps. The first dictionary, `IOUserAudioDriverUserClientProperties`, maps `IOClass` to `IOUserUserClient` and `IOUserClass` to `IOUserAudioDriverUserClient`. This allows the HAL to connect to the driver. To support user client connections from apps, the sample also defines a custom user client class. The dictionary for the custom user client has the key `SimpleAudioDriverUserClient Properties`, and its `IOUserClass` has the value `SimpleAudioDriverUserClient`, a custom subclass of `IOUserClient`. Drivers that don't accept user client connections from apps don't need this second dictionary.

When the HAL requires a new user client connection to the dext, it calls the driver's `NewUser Client` method. In the sample, the implementation of this method serves two purposes. If the incoming client type is `kIOUserAudioDriverUserClientType`, then this is a request from the HAL. In this case, the driver just forwards the call to the `IOUserAudioDriver` superclass. For other client types, such as apps connecting to the driver, it uses the `SimpleAudioDriverUser ClientProperties` values from the `Info.plist` file to create an instance of the custom `SimpleAudioDriverUserClient` class.

```
kern_return_t SimpleAudioDriver::NewUserClient_Impl(uint32_t in_type, IOUserClient**
{
    kern_return_t error = kIOReturnSuccess;

    // Have the superclass create the IOUserAudioDriverUserClient object
    // if the type is kIOUserAudioDriverUserClientType.
    if (in_type == kIOUserAudioDriverUserClientType)
    {
        error = super::NewUserClient(in_type, out_user_client, SUPERDISPATCH);
        FailIfError(error, , Failure, "Failed to create user client");
        FailIfNULL(*out_user_client, error = kIOReturnNoMemory, Failure, "Failed to
    }
    else
    {
        IOService* user_client_service = nullptr;
        error = Create(this, "SimpleAudioDriverUserClientProperties", &user_client_s
        FailIfError(error, , Failure, "failed to create the SimpleAudioDriver user c
        *out_user_client = OSDynamicCast(IOUserClient, user_client_service);
    }

Failure:
    return error;
}
```

Create audio objects in the device initializer

The device class manages the `IOUserAudioStream` interfaces that perform audio I/O. It can also contain controls and custom properties that interact with the audio stream.

In the sample, the `SimpleAudioDevice` initializer method declares the stream format to use for `IOUserAudioStream` objects: single-channel, PCM, using 16-bit native-endian signed integer. It also sets two available sample rates — `44100.0` and `48000.0` — which a person using the sample app can toggle.

```
double sample_rates[] = {kSampleRate_1, kSampleRate_2};
SetAvailableSampleRates(sample_rates, 2);
SetSampleRate(kSampleRate_1);
const auto channels_per_frame = 1;
IOUserAudioChannelLabel input_channel_layout[channels_per_frame] = { IOUserAudioChar
IOUserAudioChannelLabel output_channel_layout[channels_per_frame] = { IOUserAudioCha

IOUserAudioStreamBasicDescription stream_formats[] =
{
    {
        kSampleRate_1, IOUserAudioFormatID::LinearPCM,
        static_cast<IOUserAudioFormatFlags>(IOUserAudioFormatFlags::FormatFlagIsSigr
        static_cast<uint32_t>(sizeof(int16_t)*channels_per_frame),
        1,
        static_cast<uint32_t>(sizeof(int16_t)*channels_per_frame),
        static_cast<uint32_t>(channels_per_frame),
        16
    },
    {
        kSampleRate_2, IOUserAudioFormatID::LinearPCM,
        static_cast<IOUserAudioFormatFlags>(IOUserAudioFormatFlags::FormatFlagIsSigr
        static_cast<uint32_t>(sizeof(int16_t)*channels_per_frame),
        1,
        static_cast<uint32_t>(sizeof(int16_t)*channels_per_frame),
        static_cast<uint32_t>(channels_per_frame),
        16
    },
};
```

AudioDriverKit maps the memory of these streams to the Core Audio HAL. In an actual hardware driver, this memory needs to be the same I/O memory the system uses for DMA to hardware.

```
OSSharedPtr<IOBufferMemoryDescriptor> output_io_ring_buffer;
OSSharedPtr<IOBufferMemoryDescriptor> input_io_ring_buffer;
const auto buffer_size_bytes = static_cast<uint32_t>(in_zero_timestamp_period * size
error = IOBufferMemoryDescriptor::Create(kIOMemoryDirectionInOut, buffer_size_bytes,
FailIf(error != kIOReturnSuccess, , Failure, "Failed to create output IOBufferMemory

error = IOBufferMemoryDescriptor::Create(kIOMemoryDirectionInOut, buffer_size_bytes,
FailIf(error != kIOReturnSuccess, , Failure, "Failed to create input IOBufferMemoryD

// Create an output/input stream object and pass in the I/O ring buffer memory descr
ivars->m_output_stream = IOUserAudioStream::Create(in_driver, IOUserAudioStreamDirec
FailIfNULL(ivars->m_output_stream.get(), error = kIOReturnNoMemory, Failure, "failed

ivars->m_input_stream = IOUserAudioStream::Create(in_driver, IOUserAudioStreamDirect
FailIfNULL(ivars->m_input_stream.get(), error = kIOReturnNoMemory, Failure, "failed

//  Configure stream properties: name, available formats, and current format.
ivars->m_output_stream->SetName(output_stream_name.get());
ivars->m_output_stream->SetAvailableStreamFormats(stream_formats, 2);
ivars->m_stream_format = stream_formats[0];
ivars->m_output_stream->SetCurrentStreamFormat(&ivars->m_stream_format);

ivars->m_input_stream->SetName(input_stream_name.get());
ivars->m_input_stream->SetAvailableStreamFormats(stream_formats, 2);
ivars->m_input_stream->SetCurrentStreamFormat(&ivars->m_stream_format);

// Add a stream object to the driver.
error = AddStream(ivars->m_output_stream.get());
FailIfError(error, , Failure, "failed to add output stream");

error = AddStream(ivars->m_input_stream.get());
FailIfError(error, , Failure, "failed to add input stream");
```

Create standard controls in the device

AudioDriverKit provides `IOUserAudioControl` objects for standard user interface to an audio device. Along with general controls for a toggle, slider, or selection interface to device properties, there are standard controls for volume and stereo pan. The sample driver adds an instance of the volume control, `IOUserAudioLevelControl`, in its initializer, which provides the volume slider in Audio MIDI Setup in macOS.

The following code example creates the audio level control with a default level of -6.0 decibels (dB), and a range of -96.0 to 0.0 dB. Like all audio controls, the level control has an element and scope to set; these properties have the same meaning as the `AudioUnitElement` and `AudioUnitScope` of an `AUAudioUnit` in AudioToolbox. In this case, the element `IOUserAudioObjectPropertyElementMain` affects the entire control, and the scope `IOUserAudioObjectPropertyScope::Input` indicates that this control affects input from the device.

```
// Create the volume control object for the input stream.
ivars->m_input_volume_control = IOUserAudioLevelControl::Create(in_driver,
                                                                true,
                                                                -6.0,
                                                                {-96.0, 0.0},
                                                                IOUserAudioObjectPro
                                                                IOUserAudioObjectPro
                                                                IOUserAudioClassID::
FailIfNULL(ivars->m_input_volume_control.get(), error = kIOReturnNoMemory, Failure,
ivars->m_input_volume_control->SetName(input_volume_control_name.get());

// Add the volume control to the device object.
error = AddControl(ivars->m_input_volume_control.get());
FailIfError(error, , Failure, "failed to add input volume level control");
```

Create custom properties to control the device

The sample creates two custom control properties for its virtual device. In the code example below, the sample creates a property address reference for a property selector, using the main element and global scope. It then creates local pointer variables for the property selector, the data, and an optional qualifier that provides further detail about how to use the property data.

```
IOUserAudioObjectPropertyAddress prop_addr = {
    kSimpleAudioDriverCustomPropertySelector,
    IOUserAudioObjectPropertyScope::Global,
    IOUserAudioObjectPropertyElementMain };
OSSharedPtr<IOUserAudioCustomProperty> custom_property = nullptr;
OSSharedPtr<OSString> qualifier = nullptr;
OSSharedPtr<OSString> data = nullptr;
```

Later in the device's initializer, the following code example creates the custom property from the property address, setting two qualifier/data pairs before adding the custom property.

```
custom_property = IOUserAudioCustomProperty::Create(in_driver,
                                                    prop_addr,
                                                    true,
                                                    IOUserAudioCustomPropertyDataTyp
                                                    IOUserAudioCustomPropertyDataTyp

// Set the qualifier and data-value pair on the custom property.
qualifier = OSSharedPtr(OSString::withCString(kSimpleAudioDriverCustomPropertyQualit
data = OSSharedPtr(OSString::withCString(kSimpleAudioDriverCustomPropertyDataValue0)
custom_property->SetQualifierAndDataValue(qualifier.get(), data.get());

// Set another qualifier and data-value pair on the custom property.
qualifier = OSSharedPtr(OSString::withCString(kSimpleAudioDriverCustomPropertyQualit
data = OSSharedPtr(OSString::withCString(kSimpleAudioDriverCustomPropertyDataValue1)
custom_property->SetQualifierAndDataValue(qualifier.get(), data.get());
AddCustomProperty(custom_property.get());
```

> **Note**
>
> iPadOS doesn't support custom properties and qualifiers.

Handle the callback to start device I/O

Because the sample project doesn't connect to a hardware device, it uses timers and actions in place of hardware interrupts and DMA. When the HAL attempts to start I/O on the device, it calls `SimpleAudioDevice::StartIO`. AudioDriverKit provides this method to signal the driver to perform any necessary calls to start I/O on the device. The sample project uses this signal to start its timers.

In the sample, the `StartIO` implementation calls `GetIOMemoryDescriptor` to get the streams' memory descriptors, and then creates an `IOMemoryMap` with `CreateMapping`. After setting up the mapping, the sample uses a private helper method, `StartTimers`, to configure and enable the time sources and actions to generate timestamps and fill out the input audio buffer.

```
kern_return_t SimpleAudioDevice::StartIO(IOUserAudioStartStopFlags in_flags)
{
    DebugMsg("Start I/O: device %u", GetObjectID());

    __block kern_return_t error = kIOReturnSuccess;
    __block OSSharedPtr<IOMemoryDescriptor> input_iomd;
    __block OSSharedPtr<IOMemoryDescriptor> output_iomd;
```

```
    ivars->m_work_queue->DispatchSync(^(){
        //  Tell IOUserAudioObject base class to start I/O for the device.
        error = super::StartIO(in_flags);
        FailIfError(error, , Failure, "Failed to start I/O");

        output_iomd = ivars->m_output_stream->GetIOMemoryDescriptor();
        FailIfNULL(output_iomd.get(), error = kIOReturnNoMemory, Failure, "Failed to
        error = output_iomd->CreateMapping(0, 0, 0, 0, 0, ivars->m_output_memory_map
        FailIf(error != kIOReturnSuccess, , Failure, "Failed to create memory map fr

        input_iomd = ivars->m_input_stream->GetIOMemoryDescriptor();
        FailIfNULL(input_iomd.get(), error = kIOReturnNoMemory, Failure, "Failed to
        error = input_iomd->CreateMapping(0, 0, 0, 0, 0, ivars->m_input_memory_map.a
        FailIf(error != kIOReturnSuccess, , Failure, "Failed to create memory map fr

        // Start the timers to send timestamps and generate sine tone on the stream
        StartTimers();
        return;

    Failure:
        super::StopIO(in_flags);
        ivars->m_output_memory_map.reset();
        ivars->m_input_memory_map.reset();
        return;
    });

    return error;
}
```

An `IOUserAudioDevice` is a subclass of `IOUserAudioClockDevice`, and as such, it's responsible for timekeeping between the driver and the hardware device. AudioDriverKit enables this with the methods `UpdateCurrentZeroTimestamp` and `GetCurrentZeroTimestamp`. The framework handles the timestamps atomically, and the HAL uses the sample time-host time pair to run and synchronize I/O. Therefore, it's vital to track the hardware clock's timestamps as closely as possible.

In the case of the sample device, timers and actions simulate calls from a hardware device. These actions manage the zero timestamp values. The device class's initializer creates an `IOTimerDispatchSource` to serve as the timer. Then it creates an action to invoke a callback named `ZtsTimerOccurred`, which simulates the handling of a hardware callback.

```
// Initialize the timer that stands in for a real interrupt.
error = IOTimerDispatchSource::Create(ivars->m_work_queue.get(), &zts_timer_event_so
FailIfError(error, , Failure, "failed to create the ZTS timer event source");
ivars->m_zts_timer_event_source = OSSharedPtr(zts_timer_event_source, OSNoRetain);

// Create a timer action to generate timestamps.
error = CreateActionZtsTimerOccurred(sizeof(void*), &zts_timer_occurred_action);
FailIfError(error, , Failure, "failed to create the timer event source action");
ivars->m_zts_timer_occurred_action = OSSharedPtr(zts_timer_occurred_action, OSNoReta
ivars->m_zts_timer_event_source->SetHandler(ivars->m_zts_timer_occurred_action.get()
```

In the `StartTimers` method, which `StartIO` calls earlier, the sample calls `UpdateCurrentZeroTimestamp` to update the pair of values that represents the sample time and host time. Then it starts the timer that the sample creates in the previous code example, using `mach_absolute_time` and host ticks from the device to schedule the timed callback.

```
// Clear the device's timestamps.
UpdateCurrentZeroTimestamp(0, 0);
auto current_time = mach_absolute_time();

// Start the timer. The first timestamp occurs when the timer goes off.
ivars->m_zts_timer_event_source->WakeAtTime(kIOTimerClockMachAbsoluteTime, current_t
ivars->m_zts_timer_event_source->SetEnable(true);
```

When the `ZtsTimerOccurred` action fires, it gets the last zero timestamp value from the device by calling `GetCurrentZeroTimestamp`. If this is the first timestamp, it uses `mach_absolute_time` as the anchor time. Otherwise, it updates the timestamps by the zero timestamp period and host ticks per buffer. Either way, it updates the device's timestamps with a call to `UpdateCurrentZeroTimestamp`. Finally, it sets the timer to wake up in the future for the next zero timestamp.

```
void    SimpleAudioDevice::ZtsTimerOccurred_Impl(OSAction* action, uint64_t time)
{
    // Get the current time.
    auto current_time = time;

    // Increment the timestamps...
    uint64_t current_sample_time = 0;
    uint64_t current_host_time = 0;
    GetCurrentZeroTimestamp(&current_sample_time, &current_host_time);
```

```
    auto host_ticks_per_buffer = ivars->m_zts_host_ticks_per_buffer;

    if(current_host_time != 0)
    {
        current_sample_time += GetZeroTimestampPeriod();
        current_host_time += host_ticks_per_buffer;
    }
    else
    {
        // ...but not if it's the first one.
        current_sample_time = 0;
        current_host_time = current_time;
    }

    // Update the device with the current timestamp.
    UpdateCurrentZeroTimestamp(current_sample_time, current_host_time);

    // Set the timer to go off in one buffer.
    ivars->m_zts_timer_event_source->WakeAtTime(kIOTimerClockMachAbsoluteTime,
                                                current_host_time + host_ticks_per_b
}
```

Use real-time callbacks to perform signal processing in the driver

For drivers that need to perform signal processing, AudioDriverKit provides real-time callbacks. The driver registers a block that the system calls in a real-time context whenever an I/O operation occurs on the `IOUserAudioStream` buffers for the device.

In the sample code, this is how `SimpleAudioDriver` creates its sine tone. It declares its callback in the `SimpleAudioDevice` initalizer.

```
io_operation = ^kern_return_t(IOUserAudioObjectID in_device,
                              IOUserAudioIOOperation in_io_operation,
                              uint32_t in_io_buffer_frame_size,
                              uint64_t in_sample_time,
                              uint64_t in_host_time)
{
```

The block receives a reference to the device, the operation it's performing, the buffer size, and the sample and host times. `SimpleAudioDriver` checks that the operation is `IOUserAudioIOOperationBeginRead`, and if it is, it fills its audio buffers with signal data. The data

is either loopback from the audio output, or a programmatically generated sine tone that results from a call to a private method, `GenerateToneForInput`.

Because this callback block runs on a real-time thread, it must not perform any lengthy or indeterminate operations. This includes things like allocating memory, acquiring locks, calling Objective-C or Swift methods, and performing file system or network I/O.

To set this block as the callback, the sample calls the `IOUserAudioDevice` method `SetIOOperationHandler`.

```
this->SetIOOperationHandler(io_operation);
```

Access DMA audio buffers

As mentioned previously, a private method called `GenerateToneForInput` creates the sine tone. This is where the sample simulates writing audio data to DMA, and thereby delivers it to the hardware.

This method starts by checking that the `m_input_memory_map` that `StartIO` creates is valid. If so, it uses the memory map buffer length and stream format to calculate the length in samples for the I/O buffer. Because the sample project supports only signed, 16-bit PCM audio, it recasts the buffer to an `int_16` pointer.

With the calculated buffer length and the pointer ready, it's possible to fill the buffer with the sine tone. The sample starts by getting the current volume control gain as a scalar value. Next, it loops for the number of samples necessary to fill the buffer. In the loop, it calculates a sine value for each sample and applies the volume gain, then writes this value as a signed, 16-bit integer to all the channels in the buffer's format.

```
void SimpleAudioDevice::GenerateToneForInput(double in_tone_freq, size_t in_sample_t
{
    // Fill out the input buffer with a sine tone.
    if (ivars->m_input_memory_map)
    {
        // Get the pointer to the I/O buffer and use stream format information
        // to get the buffer length.
        const auto& format = ivars->m_stream_format;
        auto buffer_length = ivars->m_input_memory_map->GetLength() / (format.mBytes
        auto num_samples = in_frame_size;
        auto buffer = reinterpret_cast<int16_t*>(ivars->m_input_memory_map->GetAddre

        // Get the volume control dB value to apply gain to the tone.
        auto input_volume_level = ivars->m_input_volume_control->GetScalarValue();
```

```
        for(size_t i = 0; i < num_samples; i++)
        {
            float float_value = input_volume_level * sin(2.0 * M_PI * in_tone_freq *
            int16_t integer_value = FloatToInt16(float_value);
            for (auto channel_index = 0; channel_index < format.mChannelsPerFrame; c
            {
                auto buffer_index = (format.mChannelsPerFrame * (in_sample_time + i)
                buffer[buffer_index] = integer_value;
            }
            ivars->m_tone_sample_index += 1;
        }
    }
}
```

Handle configuration changes

At this point, the driver and device can supply an audio stream as if it's coming from an external device. One other task a driver needs to support is handling configuration changes from the device. Three methods from `IOUserAudioClockDevice` support this ability:

- `RequestDeviceConfigurationChange` — A driver calls this method on the device prior to any configuration action. AudioDriverKit temporarily shuts down the audio stream — calling the device's `StopIO` callback — so that the device class can perform the configuration change.
- `PerformDeviceConfigurationChange` — AudioDriverKit calls this method after stopping any running I/O, signaling to the device class that it can perform its configuration change. This is where the device can change sample rate and format, or perform other changes that are only safe while I/O isn't occurring. After this method returns, AudioDriverKit restarts I/O if necessary, calling the device's `StartIO` callback.
- `AbortDeviceConfigurationChange` — A driver calls this method to stop a change from a request to `RequestDeviceConfigurationChange`. The sample doesn't need to perform any additional work to implement this method, so it just calls its superclass's implementation.

In the sample code project, changing the sample rate provides an example of how to perform a configuration change. When a person taps the Toggle Sample Rate button, the app makes a user client call to the driver's `HandleTestConfigChange` method. The driver calls `RequestDeviceConfigurationChange`, which tells AudioDriverKit to shut down I/O and then make a callback to `PerformDeviceConfigurationChange`.

```
kern_return_t SimpleAudioDriver::HandleTestConfigChange()
{
    auto change_info = OSSharedPtr(OSString::withCString("Toggle Sample Rate"), OSNo
```

```
    return ivars->m_simple_audio_device->RequestDeviceConfigurationChange(k_custom_c
}
```

The implementation of `PerformDeviceConfigurationChange` starts by logging a string it receives from the initial callback in the app. Then it toggles between one of two preset sample rate values, and sets the new sample rate on the clock device with `SetSampleRate`. Assuming this succeeds, it then sets the sample rate on both the input and output streams with `DeviceSampleRateChanged`. Finally, it calls the superclass's implementation of `PerformDeviceConfigurationChange`.

```
kern_return_t SimpleAudioDevice::PerformDeviceConfigurationChange(uint64_t change_ac
{
    DebugMsg("change action %llu", change_action);
    kern_return_t ret = kIOReturnSuccess;
    switch (change_action) {
            // Add custom config change handlers.
        case k_custom_config_change_action:
        {
            if (in_change_info)
            {
                auto change_info_string = OSDynamicCast(OSString, in_change_info);
                DebugMsg("%s", change_info_string->getCStringNoCopy());
            }

            // Toggle the sample rate of the device.
            double rate_to_set = static_cast<uint64_t>(GetSampleRate()) != static_ca
            ret = SetSampleRate(rate_to_set);
            if (ret == kIOReturnSuccess)
            {
                // Update the stream formats with the new rate.
                ret = ivars->m_input_stream->DeviceSampleRateChanged(rate_to_set);
                ret = ivars->m_output_stream->DeviceSampleRateChanged(rate_to_set);
            }
        }
            break;

        default:
            ret = super::PerformDeviceConfigurationChange(change_action, in_change_i
            break;
    }

    // Update the cached format.
```

```
    ivars->m_stream_format = ivars->m_input_stream->GetCurrentStreamFormat();

    return ret;
}
```

When this method returns, the configuration change is complete, and the system resumes I/O with the device.

See Also

Essentials

`IOUserAudioObject`

The base class for most classes in the framework.

`IOUserAudioDriver`

A DriverKit provider object that manages communications with an audio device.

`DriverKit Audio Family`

A Boolean value that indicates whether the device supports audio functionality.