Sample Code

Creating a MIDI device driver

Implement a configurable virtual MIDI driver as a driver extension that runs in user space in macOS and iPadOS.

Download

DriverKit 24.0+ | iOS 18.0+ | iPadOS 18.0+ | macOS 15.0+ | Xcode 16.0+

Overview

This sample code project shows how to create a MIDI driver extension using the MIDIDriverKit framework. It provides a C++ DriverKit implementation that allows you to publish a single MIDI device with one port that contains a MIDI source and a MIDI destination, as well as a Swift client app that installs and configures the drivers.

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

- A port
- A virtual MIDI source
- A virtual MIDI destination
- A toggle for the offline property
- The possibility to add or remove a port

MIDIDriverKit is available in macOS, and in iPadOS 18 and later when running on an iPad device with an M-series chip. This sample code project supports both platforms.

The sample app connects to the MIDI driver extension through a custom user client connection. The custom user client shows an example of how to toggle the offline property and add or remove

a port on the MIDI 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 code 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 in 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. Additionally, iPadOS 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.CreatingMIDIDriverSampleApp` for the app and `com.example.apple-samplecode.ExampleDriver.Driver` for the driver.

2. In the Xcode project, click the Signing & Capabilities tab for each of the two targets — the driver and the macOS/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 `CreatingMIDIDriverSampleAppViewModel.swift`, make sure the string concatentation that initializes `dextIdentifier` matches the bundle identifier for the driver.

5. Create an explicit App ID and provisioning profile for the sample app with the entitlements System Extension and Communicates with Drivers. For information about how to request entitlements, see Requesting Entitlements for DriverKit Development.

6. Create an explicit App ID and provisioning profile for the sample driver with the following entitlements: `com.apple.developer.driverkit`, `com.apple.developer.driverkit.family.midi`, 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.midi` 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. Click the Signing & Capabilities tab, set each target to manual code signing, and select its new profile.

Run the sample app in macOS

To run the sample app in macOS, use the scheme selector to select the CreatingMIDIDriverSampleApp 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 CreatingMIDIDriverSampleApp 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 Login Items & Extensions 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 “CreatingMIDIDriverSampleAppDriver has been activated and is ready to use.”

At this point, the sample’s MIDI device is available to CoreMIDI. To inspect the newly installed device, use the Audio MIDI Setup app (`Applications/Utilities`), which shows the CreatingMIDIDriverSampleAppDevice on the MIDI pane with one port. You can change the number of ports and toggle the offline property there, or in the CreatingMIDIDriverSampleApp 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 offline property and to add or remove ports.

To use the driver, you can use any app that supports MIDI input and output, and select CreatingMIDIDriverSampleAppDevice’s sources or destinations.

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 `com.example.apple-samplecode.ExampleDriver.Driver`.

Run the sample app 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 CreatingMIDIDriverSampleApp 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 CreatingMIDIDriverSampleApp app doesn't show the Driver Manager section because the app isn't responsible for installing the driver like it is in macOS. Instead, choose Settings > Privacy & Security > Drivers, and enable the driver there.

After enabling the driver, return to the CreatingMIDIDriverSampleApp app to open a user client connection and modify the device's offline property or change the number of ports.

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

Create driver and device classes

To create a MIDIDriverKit driver, the sample creates a driver that subclasses `IOUserMIDIDriver`, and a device that subclasses `IOUserMIDIDevice`. The dext's `Info.plist` file contains entries that identify the driver class to MIDIDriverKit, 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 `CreatingMIDIDriverSampleAppDriver`, 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 and configuration messages. The device also owns various `IOUserMIDIObject` instances for things like timer dispatch sources. 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 with virtual MIDI sources and MIDI destinations.

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

```
kern_return_t IMPL(CreatingMIDIDriverSampleAppDriver, Start)
{
    DebugMsg("+");
    bool success = true;

    OSSharedPtr<OSString> deviceName(OSString::withCString("CreatingMIDIDriverSample
    OSSharedPtr<OSString> modelUID(OSString::withCString("CreatingMIDIDriverSampleAp
    OSSharedPtr<OSString> manufacturerUID(OSString::withCString("Apple Inc."), OSNoF
```

```
    kern_return_t error = Start(provider, SUPERDISPATCH);
    if (error)
    {
        DebugMsg("Failed to start Super");
        goto Failure;
    }

    // Get the service's default dispatch queue from the driver object.
    ivars->mWorkQueue = GetWorkQueue();
    if (ivars->mWorkQueue.get() == nullptr)
    {
        DebugMsg("Failed to get default work queue");
        error = kIOReturnInvalid;
        goto Failure;
    }

    ivars->mCreatingMIDIDriverSampleAppDevice = OSSharedPtr(OSTypeAlloc(CreatingMIDI
    if (ivars->mCreatingMIDIDriverSampleAppDevice.get() == nullptr)
    {
        DebugMsg("Failed to allocate CreatingMIDIDriverSampleAppDevice");
        error = kIOReturnNoMemory;
        goto Failure;
    }

    success = ivars->mCreatingMIDIDriverSampleAppDevice->init(this, deviceName.get()
    if (!success)
    {
        DebugMsg("Failed to init CreatingMIDIDriverSampleAppDevice");
        error = kIOReturnNoMemory;
        goto Failure;
    }

    AddObject(ivars->mCreatingMIDIDriverSampleAppDevice.get());

    // Register the service
    error = RegisterService();
    if (error)
    {
        DebugMsg("Failed to register service");
        goto Failure;
    }

    return kIOReturnSuccess;
```

```
Failure:

    ivars->mCreatingMIDIDriverSampleAppDevice.reset();
    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 CoreMIDI and to other apps. The first dictionary, `IOUserMIDIDriverUserClientProperties`, maps `IOClass` to `IOUserUserClient` and `IOUserClass` to `IOUserMIDIDriverUserClient`. This allows CoreMIDI 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 `CreatingMIDIDriverSampleAppUserClientProperties`, and its `IOUserClass` has the value `CreatingMIDIDriverSampleAppDriverUserClient`, a custom subclass of `IOUserClient`. Drivers that don't accept user client connections from apps don't need this second dictionary.

When CoreMIDI requires a new user client connection to the dext, it calls the driver's `NewUserClient` method. In the sample, the implementation of this method serves two purposes. If the incoming client type is `kIOUserMIDIDriverUserClientType`, this is a request from CoreMIDI. In this case, the driver just forwards the call to the `IOUserMIDIDriver` superclass. For other client types, such as apps connecting to the driver, it uses the `CreatingMIDIDriverSampleAppUserClientProperties` values from the `Info.plist` file to create an instance of the custom `CreatingMIDIDriverSampleAppDriverUserClient` class.

```
kern_return_t IMPL(CreatingMIDIDriverSampleAppDriver, NewUserClient)
{
    DebugMsg("type: %u out_user-client: %p", type, userClient);
    kern_return_t error = kIOReturnSuccess;

    //  Have the superclass create the `IOUserMIDIDriverUserClient` object if the ty
    //  kIOUserMIDIDriverUserClientType.
    if (type == kIOUserMIDIDriverUserClientType)
    {
        error = super::NewUserClient(type, userClient, SUPERDISPATCH);
        if (error)
        {
            DebugMsg("Failed to create user client");
            goto Failure;
        }
        if (*userClient == nullptr)
```

```
        {
            DebugMsg("Failed to create user client");
            error = kIOReturnNoMemory;
            goto Failure;
        }
    }
    else
    {
        IOService* userClientService = nullptr;
        error = Create(this, "CreatingMIDIDriverSampleAppUserClientProperties", &use
        if (error != kIOReturnSuccess)
        {
            DebugMsg("failed to create the CreatingMIDIDriverSampleAppDriver user-cl
            goto Failure;

        }
        *userClient = OSDynamicCast(IOUserClient, userClientService);
    }

Failure:
    return error;
}
```

Create MIDI objects and set properties in the device initializer

The device class manages the `IOUserMIDIEntity` interfaces containing `IOUserMIDISource` and `IOUserMIDIDestination` objects that perform MIDI I/O.

In the sample, the `CreatingMIDIDriverSampleAppDevice` initializer method declares one instance of `IOUserMIDIEntity`, including one of each `IOUserMIDISource` and `IOUserMIDIDestination` objects.

```
    auto entityName = CreateEntityName(1);
    auto entity = IOUserMIDIEntity::Create(driver,
                                           this,
                                           entityName.get(),
                                           IOUserMIDIProtocolID::MIDIProtocol_2_0,
                                           1, 1);
    AddEntity(entity.get());

    SetupEntities();
```

```
    SetProperty(IOUserMIDIProperty::Offline, offline.get());
```

Set up I/O for MIDI sources and MIDI destinations

To receive MIDI data coming from CoreMIDI, each destination needs to set the I/O block. The sample routes each destination to its corresponding source to implement a virtual driver.

```
void CreatingMIDIDriverSampleAppDevice::SetupEntities()
{
    ivars->mDestinations = OSSharedPtr(OSArray::withCapacity(1), OSNoRetain);

    GetEntities()->iterateObjects(^bool(OSObject* object){
        auto e = OSDynamicCast(IOUserMIDIEntity, object);
        if (e != nullptr)
        {
            auto source = e->GetSource(0);
            auto destination = e->GetDestination(0);
            auto ioBlock = ^kern_return_t(IOUserMIDIUMPWord const* umpWords, size_t
                return source->Send(umpWords, numWords);
            };
            destination->SetIOBlock(ioBlock);
        }
        return false;
    });
}
```

Start device I/O

When CoreMIDI attempts to start I/O on the device, it calls `CreatingMIDIDriverSampleAppDevice::StartIO`. MIDIDriverKit provides this method to signal the driver to perform any necessary calls to start I/O on the device.

```
kern_return_t CreatingMIDIDriverSampleAppDevice::StartIO()
{
    DebugMsg("StartIO: device %u", GetObjectID());

    __block kern_return_t error = kIOReturnSuccess;

    ivars->mWorkQueue->DispatchSync(^{
        // Tell IOUserMIDIObject base class to start I/O for the device.
```

```
        error = super::StartIO();
        if (error) {
            DebugMsg("Failed to start I/O, error %d", error);
            super::StopIO();
        }
    });

    if (error == kIOReturnSuccess) {
        auto offline = OSSharedPtr(OSNumber::withNumber(uint64_t{0}, 32), OSNoRetair
        SetProperty(IOUserMIDIProperty::Offline, offline.get());
    }

    return error;
}
```

Handle configuration changes

At this point, the driver and device can supply a MIDI port with one source and one destination 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 `IOUserMIDIDevice` support this ability:

- `RequestDeviceConfigurationChange` — A driver calls this method on the device prior to any configuration action. MIDIDriverKit temporarily shuts down ports — calling the device's `StopIO` callback — so that the device class can perform the configuration change.
- `PerformDeviceConfigurationChange` — MIDIDriverKit 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 the number of ports, source, and destination, or perform other changes that are only safe while I/O isn't occurring. After this method returns, MIDIDriverKit 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 number of ports provides an example of how to perform a configuration change. When a person taps the Add Port button, the app makes a user client call to the driver's `HandleAddPort` method. The driver calls into the device's `AddPort` method, which calls `RequestDeviceConfigurationChange`. The latter tells MIDIDriverKit to shut down I/O and then make a callback to `PerformDeviceConfigurationChange`.

```
kern_return_t CreatingMIDIDriverSampleAppDriver::HandleAddPort()
{
```

```
    __block kern_return_t ret = kIOReturnSuccess;
    ivars->mWorkQueue->DispatchSync(^(){
        ret = ivars->mCreatingMIDIDriverSampleAppDevice->AddPort();
    });
    return ret;
}
```

```
kern_return_t CreatingMIDIDriverSampleAppDevice::AddPort()
{
    auto changeInfo = OSSharedPtr(OSString::withCString("Add Port"), OSNoRetain);
    if (GetDeviceIsRunning())
    {
        return RequestDeviceConfigurationChange(kAddPortConfigChangeAction, changeIn
    }
    else
    {
        return PerformDeviceConfigurationChange(kAddPortConfigChangeAction, changeIn
    }
}
```

The implementation of `PerformDeviceConfigurationChange` starts by logging a string it receives from the initial callback in the app. Then it adds the new `IOUserMIDIEntity` with each `IOUserMIDISource` and `IOUserMIDIDestination`. Assuming this succeeds, it sets up the callback blocks for I/O. Finally, it calls the superclass's implementation of `PerformDeviceConfigurationChange`.

```
kern_return_t CreatingMIDIDriverSampleAppDevice::PerformDeviceConfigurationChange(
        uint64_t changeAction, OSObject* changeInfo)
{

    DebugMsg("change action %llu", changeAction);
    kern_return_t ret = kIOReturnSuccess;
    switch (changeAction) {
        // Add custom configuration change handlers.
        case kAddPortConfigChangeAction:
        {
            if (changeInfo)
            {
                auto changeInfoString = OSDynamicCast(OSString, changeInfo);
                DebugMsg("%s", changeInfoString->getCStringNoCopy());
            }
```

```
            auto entities = GetEntities();
            if (entities.get() != nullptr) {
                auto index = entities->getCount() + 1;
                auto entityName = CreateEntityName(index);
                auto entity = IOUserMIDIEntity::Create(ivars->mDriver.get(),
                                                       this,
                                                       entityName.get(),
                                                       IOUserMIDIProtocolID::MIDIPro
                                                       1, 1);
                AddEntity(entity.get());
                SetupEntities();
            }
            break;
        }

        default:
            ret = super::PerformDeviceConfigurationChange(changeAction, changeInfo);
            break;
    }
    return ret;
}
```

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

See Also

Essentials

`com.apple.developer.driverkit.family.midi`

A Boolean value that indicates whether to match the driver against devices that support MIDI.