Sample Code

Connecting a network driver

Create an Ethernet driver that interfaces with the system's network protocol stack.

Download

macOS 11.0+ | Xcode 14.3+ | DriverKit 20.2+

Overview

The `NetworkingDriverKitSample` project provides an example of how to write a driver with the NetworkingDriverKit APIs. Once installed and started, it appears like any other networking driver, viewable in IORegistry, System Settings, and the command-line interface `ifconfig`.

Use this sample as a guide for developing drivers for your own networking hardware. This sample is not hardware-dependent, and instead uses an internal timer to simulate receiving new packets. In a real-world driver, you'd connect the RX/TX submission and completion queues to real hardware transmit/receive descriptor rings supported by PCIDriverKit and USBDriverKit drivers.

Keep in mind that NetworkingDriverKit currently only supports developing Ethernet drivers.

The project contains two targets:

- NetworkingDriverKitSample — The sample driver, written in C++.
- NetworkingDriverKitSampleApp — A SwiftUI-based app that uses the SystemExtensions framework to install the driver.

Configure the sample code project

To run the sample code project, you first need to build and run `NetworkingDriverKitSample` App, which installs the dexts.

Start by choosing new bundle IDs for the app and driver. The bundle IDs included with the project are already associated with specific App IDs, so you need unique identifiers to create your own App IDs. Use a reverse-DNS format for your identifier (for more information, see Preparing your app for distribution). Then, in `DriverLoadingViewModel.swift`, edit the definition of `dext Identifier` to use the string you chose for your driver's bundle ID.

The installer and driver both need specific entitlements to run. For `NetworkingDriverKit SampleApp`, request the following entitlements:

- `System Extension Entitlement`
- `App Sandbox Entitlement`
- `com.apple.security.files.user-selected.read-only`

For the `NetworkingDriverKitSample` target, request the following:

- `com.apple.developer.driverkit`
- `com.apple.developer.driverkit.family.networking`

For more information on requesting entitlements, see Requesting Entitlements for DriverKit Development.

Next, log in to your Apple Developer account and navigate to the Identifiers list in the Certificates, IDs & Profiles section. From here, create new App IDs for `NetworkingDriverKitSampleApp` and `NetworkingDriverKitSample`.

For the Bundle ID field, choose Explicit, and use the bundle IDs you chose earlier. Then, for `NetworkingDriverKitSampleApp`, request the System Extension capability. For `Networking DriverKitSample`, request the DriverKit and DriverKit Family Networking capabilities (under Additional Capabilities).

Next, visit the Profiles section of the site to create new provisioning profiles.

1. For `NetworkingDriverKitSampleApp`, press the + icon to create a new profile.
2. On the Register a New Provisioning Profile page, choose macOS App Development and then Mac for the profile type.
3. On the next two pages, add any certificates and devices you want to include in the profile.
4. On the Additional Entitlements page, accept the default entitlements and click Continue.
5. Provide a name for the profile, click Generate, and download the profile to your computer. Drag and drop the file to Xcode to add it.

Repeat these same steps to create a profile for `NetworkingDriverKitSample`, with one key difference: on the Register a New Provisioning Profile page, choose DriverKit App Development.

Back in Xcode, do the following:

1. In the Project navigator's Signing & Capabilities tab, disable "Automatically manage signing," so that you can manually assign the provisioning profile. For each target, enter the unique bundle ID you chose earlier in the Bundle Identifier field, and use the Provisioning Profile popup to select the profile you downloaded in the previous two steps.

2. If you want to run `NetworkingDriverKitSampleApp` directly from Xcode, enter dext development mode with the Terminal command `systemextensionsctl developer on` (for more information, see Debugging and testing system extensions). Alternately, you can drag the built `DriverKitSample.app` from the build directory into the `/Applications` directory and run it from there.

Use the System Extensions framework to install the driver extension

The `NetworkingDriverKitSampleApp` target declares `NetworkingDriverKitSample` as a dependency, so building the app target builds the dext and its installer together. When it's running, `NetworkingDriverKitSampleApp` shows a single window with an Install Dext button.

The app uses the System Extensions framework to install and activate the dext. For more information, see `Installing System Extensions and Drivers`.

```
let request = OSSystemExtensionRequest
    .activationRequest(forExtensionWithIdentifier: dextIdentifier,
                       queue: .main)
request.delegate = self
OSSystemExtensionManager.shared.submitRequest(request)
```

> **Note**
>
> This call may prompt a System Extension Blocked dialog, which explains that `DriverKit SampleApp` tried to install a new system extension. To complete the installation, open System Settings and go to the Privacy & Security section. Find the section that explains that the system blocked `NetworkingDriverKitSampleApp`, and click Allow to complete the installation. To confirm installation of the `NullDriver` extension, run `systemextensionsctl list` in Terminal.

After installation, the driver appears in System Settings as Ethernet Adapter (enXX), where *XX* is a number, like `en17`. You can also see the extension in Terminal with the command `systemextensionsctl list`, which lists all running system extensions. When the driver is running, it has an entry like the following, but with a different `teamID` value:

```
~ % systemextensionsctl list
1 extension(s)
--- com.apple.system_extension.driver_extension
enabled    active    teamID    bundleID (version)    name    [state]
*    *    A123456789    com.example.apple-samplecode.NetworkingDriverKitSample (1.0/
```

You can also inspect its entry in the I/O Registry with the Terminal command `ioreg`. It appears in the registry tree with an entry like the following:

```
    | +-o NetworkingDriverKitSample  <class IOUserNetworkEthernet, id 0x100028cee, r
```

Start the driver by creating transmit and receive queues

At startup, NetworkingDriverKit calls the following methods on a driver as part of its lifecycle:

- `init` — The object initializer, which a driver uses to allocate and initialize its instance variables.
- `Start` — The message that indicates NetworkingDriverKit matched a provider for the driver. Drivers use this method to reset hardware and pepare it for operation.
- `SetPowerState` — An indication that the provider's power state is changing. Drivers can use the first call to the method for setup, but typically do so in `Start()`. Instead, use this callback to get into and out of a safe state given the power setting.
- `SetInterfaceEnable` — This call enables or disables the Ethernet service. A hardware driver uses this callback to bring the hardware up or down based on the Boolean `isEnable` parameter.

The sample driver does its setup work in the `Start` method. The driver uses this opportunity to set up queues to ensure that calls to and from NetworkingDriverKit are thread-safe. First, the sample creates a primary dispatch queue, which it calls `Default`.

```
ret = CopyDispatchQueue("Default", &ivars->dsQueue);
if (ret != kIOReturnSuccess)
    goto fail;
```

The `Start` method then uses this dispatch queue, stored in `ivars->dsQueue`, to create the transmit submission queue. Since the `.iig` file declares a method named `TxPacketAvailable`, DriverKit creates a method called `CreateActionTxPacketAvailable`. This method creates the action that DriverKit calls when packets are available to transmit. The following setup in `Start`

creates a queue and calls `SetDataAvailableHandler` to tell the queue to use the newly created action. The actual implementation of `TxPacketAvailable` appears later in this article.

```
ret = CreateActionTxPacketAvailable(0, &ivars->txPacketAction);
if (ret != kIOReturnSuccess)
    goto fail;

ret = IOUserNetworkTxSubmissionQueue::Create(
    ivars->pool, this, 8, 0, ivars->dsQueue, &ivars->txsQueue);
if (ret != kIOReturnSuccess)
    goto fail;

DLOG("==> %p (%p)", this, provider);

ret = ivars->txsQueue->CopyDataQueue(&dataQueue);
if (ret != kIOReturnSuccess)
    goto fail;

DLOG("==> %p (%p)", this, provider);

ret = dataQueue->SetDataAvailableHandler(ivars->txPacketAction);
if (ret != kIOReturnSuccess)
    goto fail;
```

Next, the `Start` method creates three queues — one for transmit completion, one for receive submission, and one for receive completion — and associates each of them with the dispatch queue, `ivars->dsQueue`.

```
ret = IOUserNetworkTxCompletionQueue::Create(
    ivars->pool, this, 8, 0, ivars->dsQueue, &ivars->txcQueue);
if (ret != kIOReturnSuccess)
    goto fail;

DLOG("==> %p (%p)", this, provider);

ret = IOUserNetworkRxSubmissionQueue::Create(
    ivars->pool, this, 8, 0, ivars->dsQueue, &ivars->rxsQueue);
if (ret != kIOReturnSuccess)
    goto fail;

DLOG("==> %p (%p)", this, provider);
```

```
ret = IOUserNetworkRxCompletionQueue::Create(
    ivars->pool, this, 8, 0, ivars->dsQueue, &ivars->rxcQueue);
if (ret != kIOReturnSuccess)
    goto fail;
```

To mimic reception of data from the network, the sample creates a timer. Since the timer needs a callback action, the `.iig` declares a `RecieveTimer` method, which prompts DriverKit to provide a `CreateActionReceiveTimer` method. The following listing calls that method to create the action, then sets it as the handler for an `IOTimerDispatchSource` stored in `ivars->receiveTimerSource`.

```
ret = IOTimerDispatchSource::Create(ivars->dsQueue, &ivars->receiveTimerSource);
if (ret != kIOReturnSuccess)
    goto fail;

status = CreateActionReceiveTimer(sizeof(void *), &ivars->receiveTimer);
if (ret != kIOReturnSuccess)
    goto fail;

status = ivars->receiveTimerSource->SetHandler(ivars->receiveTimer);
if (ret != kIOReturnSuccess)
    goto fail;
```

The timer actually starts later, in `SetInterfaceEnable`, after enabling the transmit and receive queues seen earlier.

```
now = clock_gettime_nsec_np(CLOCK_UPTIME_RAW);
deadline = now + 1000 * kMillisecondScale;
ret = ivars->receiveTimerSource->WakeAtTime(kIOTimerClockUptimeRaw, deadline, 0);
if (ret != kIOReturnSuccess)
    goto disable;
```

Receive packets in an action callback

When the timer fires, it calls the sample's `ReceiveTimer` callback. For the purposes of the sample project, this creates a fake ICMP request packet that it can submit to the receive-completion queue. For each packet dequeued from the `IOUserNetworkRxSubmissionQueue`, this method performs the following steps:

- Copies in a block of static data called `echoRequest`

- Sets the packet's data offset, data length, and link header length
- Enqueues the packet in the `IOUserNetworkRxCompletionQueue`
- Deallocates the packet
- Resets the timer for the next simulated receive-packets event

If any of the mutations to the packet fail, the sample deallocates the packet without enqueueing it. The timer update occurs in either case.

```
dequeueCount = ivars->rxsQueue->DequeuePackets(packets, 8);

linkHeaderLength = 0;
for (i = 0; i < dequeueCount; i++) {
    packet = packets[i];
    good_packet = true;
    dataAddr = (uint8_t *)packet->getDataVirtualAddress();
    dataOffset = packet->getDataOffset();

    DLOG("dataAddr = %p dataOffset = %llu", dataAddr, dataOffset);

    pktBuffer = (decltype(pktBuffer))(uintptr_t)(dataAddr + dataOffset);

    bcopy(echoRequest, pktBuffer, sizeof(echoRequest));

    ret = packet->setDataOffset(dataOffset);
    good_packet &= (ret == kIOReturnSuccess);

    ret = packet->SetLinkHeaderLength(linkHeaderLength);
    good_packet &= (ret == kIOReturnSuccess);

    ret = packet->setDataLength(sizeof(echoRequest));
    good_packet &= (ret == kIOReturnSuccess);

    if (good_packet) {
        DLOG("enqueue - packet[%d] = %p", i, packet);

        ret = ivars->rxcQueue->EnqueuePacket(packet);
        if (ret != kIOReturnSuccess) {
            ivars->pool->DeallocatePacket(packet);
            LOG("Enqueue failed dropping pkt\n");
        }
    } else {
```

```
        ivars->pool->DeallocatePacket(packet);
        LOG("Packet setup failed dropping pkt\n");
    }
}

now = clock_gettime_nsec_np(CLOCK_UPTIME_RAW);
deadline = now + 5ULL * kSecondScale;
ret = ivars->receiveTimerSource->WakeAtTime(kIOTimerClockUptimeRaw, deadline, 0);
if (ret != kIOReturnSuccess) {
    DLOG("error setting interrupt read timer 0x%08x\n", ret);
}
```

Transmit packets in an action callback

The `Start` method created the action `TxPacketAvailable` to handle callbacks when the networking stack places packets on the transmit-submssion queue. Like the packet-receive handler, this method dequeues available packets, this time from the `IOUserNetworkTxSubmissionQueue`, and loops over them. For the purposes of the sample, this method implementation just logs the data address, data offset, and link header length of each packet, and enqueues it in the `IOUserNetworkTxCompletionQueue`.

```
dequeueCount = ivars->txsQueue->DequeuePackets(packets, 8);

linkHeaderLength = 0;

if (dequeueCount) {
    for (i = 0; i < dequeueCount; i++) {
        packet = packets[i];

        DLOG("dequeue - TX packet[%d] = %p", i, packet);

        dataAddr = (uint8_t *)packet->getDataVirtualAddress();
        dataOffset = packet->getDataOffset();

        ret = packet->GetLinkHeaderLength(&linkHeaderLength);

        DLOG("dataAddr = %p dataOffset = %llu linkHeaderLength = %d", dataAddr, data

        ret = ivars->txcQueue->EnqueuePacket(packet);
        if (ret != kIOReturnSuccess) {
            ivars->pool->DeallocatePacket(packet);
            LOG("Returning Tx Packet failed just return to pool\n");
```

```
        }
    }
}
```

Remove the running driver

When shipping a DriverKit driver, people delete the driver by removing the parent app from their `/Applications` directory. If you're using dext developer mode to build and run the driver from Xcode, then you need to remove the driver manually.

To remove the driver, use the `systemextensionsctl uninstall` command, passing the team ID and the driver's bundle ID. To look up these identifiers, use the `systemextensionsctl list` command to return both of these values. Invoke the `uninstall` command as follows:

```
~ % systemextensionsctl uninstall A123456789 com.example.apple-samplecode.Networking
```

After authorization – via password, Touch ID, or an equivalent – the driver disappears from the System Settings network pane, and will no longer appear in the I/O Registry.