Sample Code

Handling Stylus Input from a Human Interface Device

Process stylus-related input from a human interface device and dispatch events to the system.

Download

DriverKit 19.0+ | macOS 10.15+ | Xcode 12.0+

Overview

The human interface device (HID) specification defines how hardware, such as keyboards and mice, communicates information to a host computer. HID hardware comes in a variety of types, and corresponds to an expected type of usage. Each device communicates this usage information, along with data values, to the host computer. A driver processes the data and uses it to dispatch relevant events to the operating system.

The HIDStylusDriver sample implements an event service that processes input from a drawing stylus, such as Apple Pencil. The event service is a subclass of `IOUserHIDEventService`, which processes the incoming device data and turns it into a set of easily accessible element objects. The sample iterates over these objects looking for changes to the data. For example, when the user moves the stylus or changes its twist or tilt, the stylus reports that change to the sample's event service. The sample forwards the data to the system as part of an event, which the system then dispatches to relevant apps.

For details about working with HID hardware, see the HID specification at https://www.usb.org/.

Configure the Sample Code Project

You can't use automatic code signing for this sample app. You must create an explicit App ID and provisioning profile, and your provisioning profile must contain the following set of entitlements:

- `com.apple.developer.driverkit.family.hid.eventservice`
- `com.apple.developer.driverkit.transport.hid`
- `com.apple.developer.driverkit`

Request these entitlements from Apple, and use them to configure a provisioning profile for the sample. See Requesting Entitlements for DriverKit Development.

To test this sample with custom stylus hardware, update the `IOKitPersonalities` dictionary in the driver's `Info.plist` file. The `HIDStylusDriver` personality contains hypothetical values of the kind of keys to include. Change the values of the `VendorID` and `ProductID` keys to match values from your own custom hardware. You can also change the `PrimaryUsagePage` and `PrimaryUsage` keys to support different device usages. Leave the other keys unchanged.

The normal installation of a DriverKit extension includes security checks to validate the DriverKit extension. During development, you typically disable these checks to speed up the turnaround time for your builds. Disable these checks for the sample app by doing the following:

1. Disable system integrity protection (SIP) on your system.

2. Run `systemextensionsctl developer on` from Terminal to enable developer mode.

Enabling developer mode allows you to run and debug the StylusApp in place, without moving it to one of your system's Applications folders. You must also disable SIP to skip the notarization checks that normally occur when installing DriverKit extensions. StylusApp attempts to install the HIDStylusDriver extension at launch time. When installation is successful, the system asks you to open Security & Privacy system preferences. In the General tab, allow the system to install the driver.

After you install the DriverKit extension, you can verify its installation by running the `systemextensionsctl list` command in Terminal. You can also use that tool to uninstall your extension or reset the state of your system extensions.

Note: If an error occurs during the installation process, the app writes an appropriate error message to the Xcode console. If you get an unknown error, verify that the OSBundleUsageDescription key in the driver's `Info.plist` file has the correct spelling.

For additional information, see Debugging and Testing System Extensions.

Start Up the Event Service

After matching an event service to a device, the system calls the `Start` method of that service. The `Start` method verifies that the event service is able to run, and puts it into the running state.

The `Start` method of `HIDStylusDriver` performs three tasks:

1. It calls the `Start` method of its parent class.

2. It calls the `getElements` method to create the initial set of `IOHIDElement` objects.

3. It caches the subset of element objects that contain stylus data.

After each step, the `Start` method checks the result to see if the step was successful. If any step fails, the sample calls the inherited `Stop` method to terminate the event service. For example, it stops the event service if it is unable to retrieve the element objects or if none of the objects contains stylus data.

```
kern_return_t
IMPL(HIDStylusDriver, Start)
{
    kern_return_t ret;

    ret = Start(provider, SUPERDISPATCH);
    if (ret != kIOReturnSuccess) {
        Stop(provider, SUPERDISPATCH);
        return ret;
    }
```

Notice that the implementation of the `Start` method includes the `IMPL` macro instead of the normal list of parameters. This macro provides binding between the kernel (which calls the method), and the method itself (which runs in user space). The `SUPERDISPATCH` macro provides a similar binding in the other direction. The sample uses it to call inherited methods that run in the kernel, such as the `Start` method of `IOUserHIDEventService`.

Identify Stylus-Related Elements

The `IOUserHIDEventService` class automatically handles incoming reports from the device, turning the raw bytes of the report into a set of `IOHIDElement` objects. Each element object contains details about a particular piece of data that the device supports. For example, some elements from a stylus contain the position of the stylus, its tilt, or the amount of pressure at its tip.

At startup, the sample calls `parseDigitizerElement` for all relevant element objects. That method collects the related elements for a specific type of device input. Although the sample normally handles stylus input, it can also handle touch input. During subsequent parsing, the event service examines only the objects in its cached collections, instead of all element objects.

```
bool HIDStylusDriver::parseDigitizerElement(IOHIDElement *element)
{
    bool result = false;
    IOHIDElement *parent = element;
    IOHIDDigitizerCollection *collection = NULL;

    if (element->getType() > kIOHIDElementTypeInput_ScanCodes) {
        return false;
    }

    // Find the top-level collection element.
    while ((parent = parent->getParentElement())) {
        IOHIDElementCollectionType collectionType = parent->getCollectionType();
        uint32_t usagePage = parent->getUsagePage();
        uint32_t usage = parent->getUsage();

        if (usagePage != kHIDPage_Digitizer) {
            continue;
        }

        if (collectionType == kIOHIDElementCollectionTypeLogical ||
            collectionType == kIOHIDElementCollectionTypePhysical) {
            if (usage >= kHIDUsage_Dig_Stylus &&
                usage <= kHIDUsage_Dig_GestureCharacter) {
                break;
            }
        } else if (collectionType == kIOHIDElementCollectionTypeApplication) {
            if (usage >= kHIDUsage_Dig_Digitizer &&
                usage <= kHIDUsage_Dig_DeviceConfiguration) {
                break;
            }
        }
    }

    // Ignore elements that aren't in an appropriate collection.
    if (!parent) {
        return false;
    }

    switch (element->getUsagePage()) {
        case kHIDPage_GenericDesktop:
            switch (element->getUsage()) {
                case kHIDUsage_GD_X:
```

```
            case kHIDUsage_GD_Y:
            case kHIDUsage_GD_Z:
                if (element->getFlags() & kIOHIDElementFlagsRelativeMask) {
                    return false;
                }
                break;
        }
        break;
}

if (!_digitizer.collections) {
    _digitizer.collections = OSArray::withCapacity(4);
    if (!_digitizer.collections) {
        return false;
    }
}

// Find the collection the element belongs to.
for (unsigned int i = 0; i < _digitizer.collections->getCount(); i++) {
    IOHIDDigitizerCollection *tmp = OSDynamicCast(IOHIDDigitizerCollection,
                                                  _digitizer.collections->getOb

    if (!tmp) {
        continue;
    }

    if (tmp->getParentCollection() == parent) {
        collection = tmp;
        break;
    }
}

// If an appropriate parent collection wasn't found, create one.
if (!collection) {
    IOHIDDigitizerCollectionType type = kIOHIDDigitizerCollectionTypeStylus;

    switch (parent->getUsage()) {
        case kHIDUsage_Dig_Puck:
            type = kIOHIDDigitizerCollectionTypePuck;
            break;
        case kHIDUsage_Dig_Finger:
        case kHIDUsage_Dig_TouchScreen:
        case kHIDUsage_Dig_TouchPad:
```

```
                type = kIOHIDDigitizerCollectionTypeFinger;
                break;
            default:
                break;
        }

        // Create the new collection object.
        collection = IOHIDDigitizerCollection::withType(type, parent);
        if (!collection) {
            return false;
        }

        _digitizer.collections->setObject(collection);
        collection->release();
    }

    // Add the element to the collection.
    collection->addElement(element);
    result = true;

exit:
    return result;
}
```

Dispatch an Event When the Stylus Data Changes

When HID hardware detects changes in its state, it reports the details of those changes to the host computer. The host forwards each new report to the relevant drivers for handling. In a custom subclass of `IOUserHIDEventService`, the `handleReport` method receives the report data and processes it. For example, a driver might use custom data provided by the device to dispatch a modified event to the system.

The `HIDStylusDriver` class dispatches events as-is to the system. Upon receiving a report, the sample iterates over the cached elements and calls the `createStylusDataForDigitizerCollection` method for each one. That method determines whether the element contains new data, and returns a valid structure if it does.

```
void HIDStylusDriver::handleDigitizerReport(uint64_t timestamp,
                                            uint32_t reportID)
{
    if (!_digitizer.collections) {
        return;
```

```
    }

    for (unsigned int i = 0; i < _digitizer.collections->getCount(); i++) {
        IOHIDDigitizerCollection *collection = OSDynamicCast(IOHIDDigitizerCollectio
                                                             _digitizer.collections-
        IOHIDDigitizerStylusData *stylusData = NULL;

        if (!collection) {
            continue;
        }

        stylusData = createStylusDataForDigitizerCollection(collection,
                                                             timestamp,
                                                             reportID);

        if (stylusData) {
            printStylus(stylusData);
            dispatchDigitizerStylusEvent(timestamp, stylusData);
            IOFree(stylusData, sizeof(IOHIDDigitizerStylusData));
        }
    }
}
```

Unlike other inherited methods, the `dispatchDigitizerStylusEvent` method of `IOHIDEventService` runs locally in the driver's process space, not in the kernel. DriverKit annotates such methods by appending the `LOCAL` or `LOCALONLY` macro to the method definition. When calling such methods, the sample uses the standard calling semantics for inherited methods, and doesn't include the `SUPERDISPATCH` macro.

See Also

Essentials

`com.apple.developer.driverkit.transport.hid`

A Boolean value that indicates whether the driver communicates with human interface devices.

{} Handling Keyboard Events from a Human Interface Device

Process keyboard-related data from a human interface device and dispatch events to the system.