

[HIDDriverKit](#) / Handling Keyboard Events from a Human Interface Device

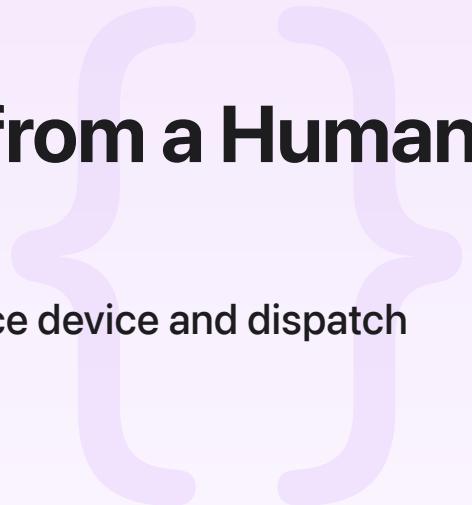
Sample Code

# Handling Keyboard Events from a Human Interface Device

Process keyboard-related data 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 HIDKeyboardDriver sample implements an event service that processes keyboard-related data. The event service is a subclass of [IOUserHIDEEventService](#), 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 presses or releases a key, the keyboard 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 keyboard hardware, update the `IOKitPersonalities` dictionary in the driver's `Info.plist` file. The `HIDKeyboardDriver` 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 `HIDKeyboardApp` 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. `HIDKeyboardApp` attempts to install the `HIDKeyboardDriver` 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 HIDKeyboardDriver 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 [IOHIDEElement](#) objects.
3. It caches the subset of element objects that contain keyboard 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 stop 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 keyboard data.

```
kern_return_t  
IMPL(HIDKeyboardDriver, 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 [IOUserHIDEEventService](#).

## Identify Keyboard-Related Elements

The [IOUserHIDEEventService](#) class automatically handles incoming reports from the device, turning the raw bytes of the report into a set of [IOHIDEElement](#) objects. Each element object contains details about a particular piece of data that the device supports. For example, some elements from a keyboard contain the current state of a specific key.

At startup, the sample calls [parseKeyboardElement](#) for all relevant element objects. That method checks the HID-defined usage value for each element, and saves a reference to all keyboard elements. During subsequent parsing, the event service examines only the objects in its cached collections, instead of all element objects.

```

bool HIDKeyboardDriver::parseKeyboardElement(IOHIDEElement *element)
{
    bool result = false;
    uint32_t usagePage = element->getUsagePage();
    uint32_t usage = element->getUsage();

    // Determine whether the element contains keyboard-related data.
    if (usagePage == kHIDPage_KeyboardOrKeypad) {
        if (usage >= kHIDUsage_KeyboardA &&
            usage <= kHIDUsage_KeyboardRightGUI) {
            result = true;
        }
    }

    if (!result) {
        return false;
    }

    if (!_keyboard.elements) {
        _keyboard.elements = OSArray::withCapacity(4);
    }

    // Save a reference to the element for later.
    _keyboard.elements->setObject(element);

exit:
    return result;
}

```

## Dispatch a Keyboard Event Using the Element Data

When HID hardware detects changes in its state, it reports the details of those changes to the system. When the system receives a new report from the device, it forwards that report to the relevant drivers. In a custom subclass of `IOUserHIDEEventService`, the `handleReport` method receives the report data and processes it.

The `HIDKeyboardDriver` class overrides `handleReport` and dispatches new reports immediately to its custom `handleKeyboardReport` method. This custom method iterates over the array of cached keyboard elements and uses each element's timestamp and report ID to determine whether the element contains new data. If it does, the `handleKeyboardReport` method calls the inherited `dispatchKeyboardEvent` method to dispatch the event information to the system.

```

// Get the previous value of the element.
preValue = element->getValue(kIOHIDValueOptionsFlagPrevious) != 0;
value = element->getValue(0) != 0;

// If the element's value didn't change, skip it.
if (value == preValue) {
    continue;
}

// If the code reaches this point, the element has new data, so
// dispatch an event with the new value to the system.
usagePage = element->getUsagePage();
usage = element->getUsage();

os_log(OS_LOG_DEFAULT,
    "Dispatching key with ussagPage: 0x%02x usage: 0x%02x value: %d",
    usagePage, usage, value);
dispatchKeyboardEvent(timestamp, usagePage, usage, value, 0, true);

```

Unlike other inherited methods, the [IOHIDEEventService](#) class defines the [dispatchKeyboardEvent](#) method as a local method to the driver. Because it is a local method, the sample doesn't use the [SUPERDISPATCH](#) macro to call it.

---

## See Also

### Essentials

[com.apple.developer.driverkit.transport.hid](#)

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

{ } Handling Stylus Input from a Human Interface Device

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