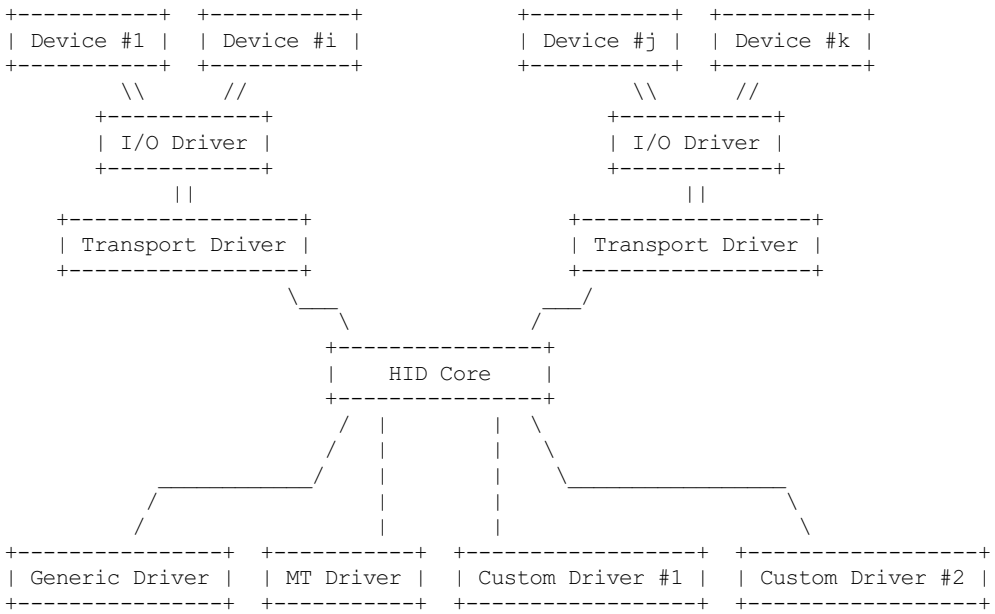
HID I/O Transport Drivers

The HID subsystem is independent of the underlying transport driver. Initially, only USB was supported, but other specifications adopted the HID design and provided new transport drivers. The kernel includes at least support for USB, Bluetooth, I2C and user-space I/O drivers.

1) HID Bus

The HID subsystem is designed as a bus. Any I/O subsystem may provide HID devices and register them with the HID bus. HID core then loads generic device drivers on top of it. The transport drivers are responsible for raw data transport and device setup/management. HID core is responsible for report-parsing, report interpretation and the user-space API. Device specifics and quirks are handled by all layers depending on the quirk.



Example Drivers:

- I/O: USB, I2C, Bluetooth-l2cap
- Transport: USB-HID, I2C-HID, BT-HIDP

Everything below "HID Core" is simplified in this graph as it is only of interest to HID device drivers. Transport drivers do not need to know the specifics.

1.1) Device Setup

I/O drivers normally provide hotplug detection or device enumeration APIs to the transport drivers. Transport drivers use this to find any suitable HID device. They allocate HID device objects and register them with HID core. Transport drivers are not required to register themselves with HID core. HID core is never aware of which transport drivers are available and is not interested in it. It is only interested in devices.

Transport drivers attach a constant "struct hid_ll_driver" object with each device. Once a device is registered with HID core, the callbacks provided via this struct are used by HID core to communicate with the device.

Transport drivers are responsible for detecting device failures and unplugging. HID core will operate a device as long as it is registered regardless of any device failures. Once transport drivers detect unplug or failure events, they must unregister the device from HID core and HID core will stop using the provided callbacks.

1.2) Transport Driver Requirements

The terms "asynchronous" and "synchronous" in this document describe the transmission behavior regarding acknowledgements. An asynchronous channel must not perform any synchronous operations like waiting for acknowledgements or verifications. Generally, HID calls operating on asynchronous channels must be running in atomic-context just fine. On the other hand, synchronous channels can be implemented by the transport driver in whatever way they like. They might just be the same as asynchronous channels, but they can also provide acknowledgement reports, automatic retransmission on failure, etc. in a blocking manner. If such functionality is required on asynchronous channels, a transport-driver must implement that via its own worker threads.

HID core requires transport drivers to follow a given design. A Transport driver must provide two bi-directional I/O channels to each HID device. These channels must not necessarily be bi-directional in the hardware itself. A transport driver might just provide 4 uni-directional channels. Or it might multiplex all four on a single physical channel. However, in this document we will describe them as

two bi-directional channels as they have several properties in common.

- Interrupt Channel (intr): The intr channel is used for asynchronous data reports. No management commands or data acknowledgements are sent on this channel. Any unrequested incoming or outgoing data report must be sent on this channel and is never acknowledged by the remote side. Devices usually send their input events on this channel. Outgoing events are normally not sent via intr, except if high throughput is required.
- Control Channel (ctrl): The ctrl channel is used for synchronous requests and device management. Unrequested data input events must not be sent on this channel and are normally ignored. Instead, devices only send management events or answers to host requests on this channel. The control-channel is used for direct blocking queries to the device independent of any events on the intr-channel. Outgoing reports are usually sent on the ctrl channel via synchronous SET_REPORT requests.

Communication between devices and HID core is mostly done via HID reports. A report can be of one of three types:

- INPUT Report: Input reports provide data from device to host. This data may include button events, axis events, battery status or more. This data is generated by the device and sent to the host with or without requiring explicit requests. Devices can choose to send data continuously or only on change.
- OUTPUT Report: Output reports change device states. They are sent from host to device and may include LED requests, rumble requests or more. Output reports are never sent from device to host, but a host can retrieve their current state. Hosts may choose to send output reports either continuously or only on change.
- FEATURE Report: Feature reports are used for specific static device features and never reported spontaneously. A host can read and/or write them to access data like battery-state or device-settings. Feature reports are never sent without requests. A host must explicitly set or retrieve a feature report. This also means, feature reports are never sent on the intr channel as this channel is asynchronous.

INPUT and OUTPUT reports can be sent as pure data reports on the intr channel. For INPUT reports this is the usual operational mode. But for OUTPUT reports, this is rarely done as OUTPUT reports are normally quite scarce. But devices are free to make excessive use of asynchronous OUTPUT reports (for instance, custom HID audio speakers make great use of it).

Plain reports must not be sent on the ctrl channel, though. Instead, the ctrl channel provides synchronous GET/SET_REPORT requests. Plain reports are only allowed on the intr channel and are the only means of data there.

- GET_REPORT: A GET_REPORT request has a report ID as payload and is sent from host to device. The device must answer with a data report for the requested report ID on the ctrl channel as a synchronous acknowledgement. Only one GET_REPORT request can be pending for each device. This restriction is enforced by HID core as several transport drivers don't allow multiple simultaneous GET_REPORT requests. Note that data reports which are sent as answer to a GET_REPORT request are not handled as generic device events. That is, if a device does not operate in continuous data reporting mode, an answer to GET_REPORT does not replace the raw data report on the intr channel on state change. GET_REPORT is only used by custom HID device drivers to query device state. Normally, HID core caches any device state so this request is not necessary on devices that follow the HID specs except during device initialization to retrieve the current state. GET_REPORT requests can be sent for any of the 3 report types and shall return the current report state of the device. However, OUTPUT reports as payload may be blocked by the underlying transport driver if the specification does not allow them.
- SET_REPORT: A SET_REPORT request has a report ID plus data as payload. It is sent from host to device and a device must update its current report state according to the given data. Any of the 3 report types can be used. However, INPUT reports as payload might be blocked by the underlying transport driver if the specification does not allow them. A device must answer with a synchronous acknowledgement. However, HID core does not require transport drivers to forward this acknowledgement to HID core. Same as for GET_REPORT, only one SET_REPORT can be pending at a time. This restriction is enforced by HID core as some transport drivers do not support multiple synchronous SET_REPORT requests.

Other ctrl-channel requests are supported by USB-HID but are not available (or deprecated) in most other transport level specifications:

- GET/SET_IDLE: Only used by USB-HID and I2C-HID.
- GET/SET_PROTOCOL: Not used by HID core.
- RESET: Used by I2C-HID, not hooked up in HID core.
- SET_POWER: Used by I2C-HID, not hooked up in HID core.

2) HID API

2.1) Initialization

Transport drivers normally use the following procedure to register a new device with HID core:

```
struct hid_device *hid;
int ret;
```

```
hid = hid_allocate_device();
if (IS_ERR(hid)) {
        ret = PTR_ERR(hid);
        goto err_<...>;
}

strscpy(hid->name, <device-name-src>, sizeof(hid->name));
strscpy(hid->phys, <device-phys-src>, sizeof(hid->phys));
strscpy(hid->uniq, <device-uniq-src>, sizeof(hid->uniq));

hid->ll_driver = &custom_ll_driver;
hid->bus = <device-bus>;
hid->vendor = <device-vendor>;
hid->product = <device-product>;
hid->version = <device-version>;
hid->country = <device-country>;
hid->dev.parent = <pointer-to-parent-device>;
hid->driver_data = <transport-driver-data-field>;

ret = hid_add_device(hid);
if (ret)
        goto err_<...>;
```

Once hid_add_device() is entered, HID core might use the callbacks provided in "custom_ll_driver". Note that fields like "country" can be ignored by underlying transport-drivers if not supported.

To unregister a device, use:

```
hid_destroy_device(hid);
```

Once hid_destroy_device() returns, HID core will no longer make use of any driver callbacks.

2.2) hid_ll_driver operations

The available HID callbacks are:

```
int (*start) (struct hid_device *hdev)
```

Called from HID device drivers once they want to use the device. Transport drivers can choose to setup their device in this callback. However, normally devices are already set up before transport drivers register them to HID core so this is mostly only used by USB-HID.

```
void (*stop) (struct hid_device *hdev)
```

Called from HID device drivers once they are done with a device. Transport drivers can free any buffers and deinitialize the device. But note that ->start() might be called again if another HID device driver is loaded on the device.

Transport drivers are free to ignore it and deinitialize devices after they destroyed them via hid_destroy_device().

```
int (*open) (struct hid_device *hdev)
```

Called from HID device drivers once they are interested in data reports. Usually, while user-space didn't open any input API/etc., device drivers are not interested in device data and transport drivers can put devices asleep. However, once ->open() is called, transport drivers must be ready for I/O. ->open() calls are nested for each client that opens the HID device.

```
void (*close) (struct hid_device *hdev)
```

Called from HID device drivers after ->open() was called but they are no longer interested in device reports. (Usually if user-space closed any input devices of the driver).

Transport drivers can put devices asleep and terminate any I/O of all ->open() calls have been followed by a ->close() call. However, ->start() may be called again if the device driver is interested in input reports again.

```
int (*parse) (struct hid_device *hdev)
```

Called once during device setup after ->start() has been called. Transport drivers must read the HID report-descriptor from the device and tell HID core about it via hid_parse_report().

```
int (*power) (struct hid_device *hdev, int level)
```

Called by HID core to give PM hints to transport drivers. Usually this is analogical to the ->open() and ->close() hints and redundant.

```
void (*request) (struct hid_device *hdev, struct hid_report *report,
                 int reqtype)
```

Send a HID request on the ctrl channel. "report" contains the report that should be sent and "reqtype" the request type. Request-type can be HID_REQ_SET_REPORT or HID_REQ_GET_REPORT.

This callback is optional. If not provided, HID core will assemble a raw report following the HID specs and send it via the ->raw_request() callback. The transport driver is free to implement this asynchronously.

```
int (*wait) (struct hid_device *hdev)
```

Used by HID core before calling ->request() again. A transport driver can use it to wait for any pending requests to complete if only one request is allowed at a time.

```
int (*raw_request) (struct hid_device *hdev, unsigned char reportnum,
                    __u8 *buf, size_t count, unsigned char rtype,
                    int reqtype)
```

Same as ->request() but provides the report as raw buffer. This request shall be synchronous. A transport driver must not use ->wait() to complete such requests. This request is mandatory and hid core will reject the device if it is missing.

```
int (*output_report) (struct hid_device *hdev, __u8 *buf, size_t len)
```

Send raw output report via intr channel. Used by some HID device drivers which require high throughput for outgoing requests on the intr channel. This must not cause SET_REPORT calls! This must be implemented as asynchronous output report on the intr channel!

```
int (*idle) (struct hid_device *hdev, int report, int idle, int reqtype)
```

Perform SET/GET_IDLE request. Only used by USB-HID, do not implement!

2.3) Data Path

Transport drivers are responsible of reading data from I/O devices. They must handle any I/O-related state-tracking themselves. HID core does not implement protocol handshakes or other management commands which can be required by the given HID transport specification.

Every raw data packet read from a device must be fed into HID core via hid_input_report(). You must specify the channel-type (intr or ctrl) and report type (input/output/feature). Under normal conditions, only input reports are provided via this API.

Responses to GET_REPORT requests via ->request() must also be provided via this API. Responses to ->raw_request() are synchronous and must be intercepted by the transport driver and not passed to hid_input_report(). Acknowledgements to SET_REPORT requests are not of interest to HID core.

Written 2013, David Herrmann <dh.herrmann@gmail.com>