UHID - User-space I/O driver support for HID subsystem

UHID allows user-space to implement HID transport drivers. Please see hid-transport.rst for an introduction into HID transport drivers. This document relies heavily on the definitions declared there.

With UHID, a user-space transport driver can create kernel hid-devices for each device connected to the user-space controlled bus. The UHID API defines the I/O events provided from the kernel to user-space and vice versa.

There is an example user-space application in ./samples/uhid/uhid-example.c

The UHID API

UHID is accessed through a character misc-device. The minor number is allocated dynamically so you need to rely on udev (or similar) to create the device node. This is /dev/uhid by default.

If a new device is detected by your HID I/O Driver and you want to register this device with the HID subsystem, then you need to open /dev/uhid once for each device you want to register. All further communication is done by read()'ing or write()'ing "struct uhid_event" objects. Non-blocking operations are supported by setting O_NONBLOCK:

```
struct uhid_event {
        __u32 type;
        union {
                struct uhid_create2_req create2;
                struct uhid_output_req output;
                struct uhid_input2_req input2;
                ...
        } u;
};
```

The "type" field contains the ID of the event. Depending on the ID different payloads are sent. You must not split a single event across multiple read()'s or multiple write()'s. A single event must always be sent as a whole. Furthermore, only a single event can be sent per read() or write(). Pending data is ignored. If you want to handle multiple events in a single syscall, then use vectored I/O with readv()/writev(). The "type" field defines the payload. For each type, there is a payload-structure available in the union "u" (except for empty payloads). This payload contains management and/or device data.

The first thing you should do is send a UHID_CREATE2 event. This will register the device. UHID will respond with a UHID_START event. You can now start sending data to and reading data from UHID. However, unless UHID sends the UHID_OPEN event, the internally attached HID Device Driver has no user attached. That is, you might put your device asleep unless you receive the UHID_OPEN event. If you receive the UHID_OPEN event, you should start I/O. If the last user closes the HID device, you will receive a UHID_CLOSE event. This may be followed by a UHID_OPEN event again and so on. There is no need to perform reference-counting in user-space. That is, you will never receive multiple UHID_OPEN events without a UHID_CLOSE event. The HID subsystem performs ref-counting for you. You may decide to ignore UHID_OPEN/UHID_CLOSE, though. I/O is allowed even though the device may have no users.

If you want to send data on the interrupt channel to the HID subsystem, you send a HID_INPUT2 event with your raw data payload. If the kernel wants to send data on the interrupt channel to the device, you will read a UHID_OUTPUT event. Data requests on the control channel are currently limited to GET_REPORT and SET_REPORT (no other data reports on the control channel are defined so far). Those requests are always synchronous. That means, the kernel sends UHID_GET_REPORT and UHID_SET_REPORT events and requires you to forward them to the device on the control channel. Once the device responds, you must forward the response via UHID_GET_REPORT_REPLY and UHID_SET_REPORT_REPLY to the kernel. The kernel blocks internal driver-execution during such round-trips (times out after a hard-coded period).

If your device disconnects, you should send a UHID_DESTROY event. This will unregister the device. You can now send UHID_CREATE2 again to register a new device. If you close() the fd, the device is automatically unregistered and destroyed internally.

write()

write() allows you to modify the state of the device and feed input data into the kernel. The kernel will parse the event immediately and if the event ID is not supported, it will return -EOPNOTSUPP. If the payload is invalid, then -EINVAL is returned, otherwise, the amount of data that was read is returned and the request was handled successfully. O_NONBLOCK does not affect write() as writes are always handled immediately in a non-blocking fashion. Future requests might make use of O_NONBLOCK, though.

UHID_CREATE2:
: This creates the internal HID device. No I/O is possible until you send this event to the kernel. The payload is of type struct uhid_create2_req and contains information about your device. You can start I/O now.

UHID_DESTROY:
: This destroys the internal HID device. No further I/O will be accepted. There may still be pending messages that you can receive with read() but no further UHID_INPUT events can be sent to the kernel. You can create a new device by sending UHID_CREATE2 again. There is no need to reopen the character device.

UHID_INPUT2:

> You must send UHID_CREATE2 before sending input to the kernel! This event contains a data-payload. This is the raw data that you read from your device on the interrupt channel. The kernel will parse the HID reports.

UHID_GET_REPORT_REPLY:

> If you receive a UHID_GET_REPORT request you must answer with this request. You must copy the "id" field from the request into the answer. Set the "err" field to 0 if no error occurred or to EIO if an I/O error occurred. If "err" is 0 then you should fill the buffer of the answer with the results of the GET_REPORT request and set "size" correspondingly.

UHID_SET_REPORT_REPLY:

> This is the SET_REPORT equivalent of UHID_GET_REPORT_REPLY. Unlike GET_REPORT, SET_REPORT never returns a data buffer, therefore, it's sufficient to set the "id" and "err" fields correctly.

read()

read() will return a queued output report. No reaction is required to any of them but you should handle them according to your needs.

UHID_START:

> This is sent when the HID device is started. Consider this as an answer to UHID_CREATE2. This is always the first event that is sent. Note that this event might not be available immediately after write(UHID_CREATE2) returns. Device drivers might require delayed setups. This event contains a payload of type uhid_start_req. The "dev_flags" field describes special behaviors of a device. The following flags are defined:
>
> - UHID_DEV_NUMBERED_FEATURE_REPORTS
> - UHID_DEV_NUMBERED_OUTPUT_REPORTS
> - UHID_DEV_NUMBERED_INPUT_REPORTS
>
> > Each of these flags defines whether a given report-type uses numbered reports. If numbered reports are used for a type, all messages from the kernel already have the report-number as prefix. Otherwise, no prefix is added by the kernel. For messages sent by user-space to the kernel, you must adjust the prefixes according to these flags.

UHID_STOP:

> This is sent when the HID device is stopped. Consider this as an answer to UHID_DESTROY.
>
> If you didn't destroy your device via UHID_DESTROY, but the kernel sends an UHID_STOP event, this should usually be ignored. It means that the kernel reloaded/changed the device driver loaded on your HID device (or some other maintenance actions happened).
>
> You can usually ignore any UHID_STOP events safely.

UHID_OPEN:

> This is sent when the HID device is opened. That is, the data that the HID device provides is read by some other process. You may ignore this event but it is useful for power-management. As long as you haven't received this event there is actually no other process that reads your data so there is no need to send UHID_INPUT2 events to the kernel.

UHID_CLOSE:

> This is sent when there are no more processes which read the HID data. It is the counterpart of UHID_OPEN and you may as well ignore this event.

UHID_OUTPUT:

> This is sent if the HID device driver wants to send raw data to the I/O device on the interrupt channel. You should read the payload and forward it to the device. The payload is of type "struct uhid_output_req". This may be received even though you haven't received UHID_OPEN yet.

UHID_GET_REPORT:

> This event is sent if the kernel driver wants to perform a GET_REPORT request on the control channel as described in the HID specs. The report-type and report-number are available in the payload. The kernel serializes GET_REPORT requests so there will never be two in parallel. However, if you fail to respond with a UHID_GET_REPORT_REPLY, the request might silently time out. Once you read a GET_REPORT request, you shall forward it to the HID device and remember the "id" field in the payload. Once your HID device responds to the GET_REPORT (or if it fails), you must send a UHID_GET_REPORT_REPLY to the kernel with the exact same "id" as in the request. If the request already timed out, the kernel will ignore the response silently. The "id" field is never re-used, so conflicts cannot happen.

UHID_SET_REPORT:

> This is the SET_REPORT equivalent of UHID_GET_REPORT. On receipt, you shall send a SET_REPORT request to your HID device. Once it replies, you must tell the kernel about it via UHID_SET_REPORT_REPLY. The same restrictions as for UHID_GET_REPORT apply.

Written 2012, David Herrmann <dh.herrmann@gmail.com>