Linux Hid Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION TO HID REPORT DESCRIPTORS

This chapter is meant to give a broad overview of what HID report descriptors are, and of how a casual (non-kernel) programmer can deal with HID devices that are not working well with Linux.

1.1 Manual parsing of HID report descriptors

Consider again the mouse HID report descriptor introduced in *Introduction to HID report descriptors*:

```
$ hexdump -C /sys/bus/hid/devices/0003\:093A\:2510.0002/report_descriptor
00000000  05 01 09 02 a1 01 09 01  a1 00 05 09 19 01 29 03  |..............).|
00000010  15 00 25 01 75 01 95 03  81 02 75 05 95 01 81 01  |..%.u.....u.....|
00000020  05 01 09 30 09 31 09 38  15 81 25 7f 75 08 95 03  |...0.1.8..%.u...|
00000030  81 06 c0 c0                                       |....|
00000034
```

and try to parse it by hand.

Start with the first number, 0x05: it carries 2 bits for the length of the item, 2 bits for the type of the item and 4 bits for the function:

```
+----------+
| 00000101 |
+----------+
        ^^
        ---- Length of data (see HID spec 6.2.2.2)
      ^^
      ------ Type of the item (see HID spec 6.2.2.2, then jump to 6.2.2.7)
  ^^^^
  --------- Function of the item (see HID spec 6.2.2.7, then HUT Sec 3)
```

In our case, the length is 1 byte, the type is `Global` and the function is `Usage Page`, thus for parsing the value 0x01 in the second byte we need to refer to HUT Sec 3.

The second number is the actual data, and its meaning can be found in the HUT. We have a `Usage Page`, thus we need to refer to HUT Sec. 3, "Usage Pages"; from there, one sees that `0x01` stands for `Generic Desktop Page`.

Moving now to the second two bytes, and following the same scheme, `0x09` (i.e. `00001001`) will be followed by one byte (`01`) and is a `Local` item (`10`). Thus, the meaning of the remaining four bits (`0000`) is given in the HID spec Sec. 6.2.2.8 "Local Items", so that we have a `Usage`. From HUT, Sec. 4, "Generic Desktop Page", we see that 0x02 stands for `Mouse`.

The following numbers can be parsed in the same way.

1.2 Introduction

HID stands for Human Interface Device, and can be whatever device you are using to interact with a computer, be it a mouse, a touchpad, a tablet, a microphone.

Many HID devices work out the box, even if their hardware is different. For example, mice can have any number of buttons; they may have a wheel; movement sensitivity differs between different models, and so on. Nonetheless, most of the time everything just works, without the need to have specialized code in the kernel for every mouse model developed since 1970.

This is because modern HID devices do advertise their capabilities through the *HID report descriptor*, a fixed set of bytes describing exactly what *HID reports* may be sent between the device and the host and the meaning of each individual bit in those reports. For example, a HID Report Descriptor may specify that "in a report with ID 3 the bits from 8 to 15 is the delta x coordinate of a mouse".

The HID report itself then merely carries the actual data values without any extra meta information. Note that HID reports may be sent from the device ("Input Reports", i.e. input events), to the device ("Output Reports" to e.g. change LEDs) or used for device configuration ("Feature reports"). A device may support one or more HID reports.

The HID subsystem is in charge of parsing the HID report descriptors, and converts HID events into normal input device interfaces (see *HID I/O Transport Drivers*). Devices may misbehave because the HID report descriptor provided by the device is wrong, or because it needs to be dealt with in a special way, or because some special device or interaction mode is not handled by the default code.

The format of HID report descriptors is described by two documents, available from the USB Implementers Forum HID web page address:

- the HID USB Device Class Definition (HID Spec from now on)
- the HID Usage Tables (HUT from now on)

The HID subsystem can deal with different transport drivers (USB, I2C, Bluetooth, etc.). See *HID I/O Transport Drivers*.

1.3 Parsing HID report descriptors

The current list of HID devices can be found at /sys/bus/hid/devices/. For each device, say /sys/bus/hid/devices/0003\:093A\:2510.0002/, one can read the corresponding report descriptor:

```
$ hexdump -C /sys/bus/hid/devices/0003\:093A\:2510.0002/report_descriptor
00000000  05 01 09 02 a1 01 09 01  a1 00 05 09 19 01 29 03  |..............).|
00000010  15 00 25 01 75 01 95 03  81 02 75 05 95 01 81 01  |..%.u.....u.....|
00000020  05 01 09 30 09 31 09 38  15 81 25 7f 75 08 95 03  |...0.1.8..%.u...|
00000030  81 06 c0 c0                                       |....|
00000034
```

Optional: the HID report descriptor can be read also by directly accessing the hidraw driver[1].

The basic structure of HID report descriptors is defined in the HID spec, while HUT "defines constants that can be interpreted by an application to identify the purpose and meaning of a data field in a HID report". Each entry is defined by at least two bytes, where the first one defines what type of value is following and is described in the HID spec, while the second one carries the actual value and is described in the HUT.

HID report descriptors can, in principle, be painstakingly parsed by hand, byte by byte.

A short introduction on how to do this is sketched in *Manual parsing of HID report descriptors*; you only need to understand it if you need to patch HID report descriptors.

In practice you should not parse HID report descriptors by hand; rather, you should use an existing parser. Among all the available ones

- the online USB Descriptor and Request Parser;
- hidrdd, that provides very detailed and somewhat verbose descriptions (verbosity can be useful if you are not familiar with HID report descriptors);

[1] read hidraw: see *HIDRAW - Raw Access to USB and Bluetooth Human Interface Devices* and file *samples/hidraw/hid-example.c* for an example. The output of hid-example would be, for the same mouse:

```
$ sudo ./hid-example
Report Descriptor Size: 52
Report Descriptor:
5 1 9 2 a1 1 9 1 a1 0 5 9 19 1 29 3 15 0 25 1 75 1 95 3 81 2 75 5 95 1 81 1 5 1 9 30 9 31 9 38␣
↪15 81 25 7f 75 8 95 3 81 6 c0 c0

Raw Name: PixArt USB Optical Mouse
Raw Phys: usb-0000:05:00.4-2.3/input0
Raw Info:
        bustype: 3 (USB)
        vendor: 0x093a
        product: 0x2510
...
```

- hid-tools, a complete utility set that allows, among other things, to record and replay the raw HID reports and to debug and replay HID devices. It is being actively developed by the Linux HID subsystem maintainers.

Parsing the mouse HID report descriptor with hid-tools leads to (explanations interposed):

```
$ ./hid-decode /sys/bus/hid/devices/0003\:093A\:2510.0002/report_descriptor
# device 0:0
# 0x05, 0x01,                    // Usage Page (Generic Desktop)        0
# 0x09, 0x02,                    // Usage (Mouse)                       2
# 0xa1, 0x01,                    // Collection (Application)            4
# 0x09, 0x01,                    // Usage (Pointer)                     6
# 0xa1, 0x00,                    // Collection (Physical)               8
# 0x05, 0x09,                    // Usage Page (Button)                10
```

what follows is a button

```
# 0x19, 0x01,                    // Usage Minimum (1)                  12
# 0x29, 0x03,                    // Usage Maximum (3)                  14
```

first button is button number 1, last button is button number 3

```
# 0x15, 0x00,                    // Logical Minimum (0)                16
# 0x25, 0x01,                    // Logical Maximum (1)                18
```

each button can send values from 0 up to including 1 (i.e. they are binary buttons)

```
# 0x75, 0x01,                    // Report Size (1)                    20
```

each button is sent as exactly one bit

```
# 0x95, 0x03,                    // Report Count (3)                   22
```

and there are three of those bits (matching the three buttons)

```
# 0x81, 0x02,                    // Input (Data,Var,Abs)               24
```

it's actual Data (not constant padding), they represent a single variable (Var) and their values are Absolute (not relative); See HID spec Sec. 6.2.2.5 "Input, Output, and Feature Items"

```
# 0x75, 0x05,                    // Report Size (5)                    26
```

five additional padding bits, needed to reach a byte

```
# 0x95, 0x01,                    // Report Count (1)                   28
```

those five bits are repeated only once

```
# 0x81, 0x01,                    // Input (Cnst,Arr,Abs)               30
```

and take Constant (Cnst) values i.e. they can be ignored.

```
# 0x05, 0x01,                    // Usage Page (Generic Desktop)       32
# 0x09, 0x30,                    // Usage (X)                          34
# 0x09, 0x31,                    // Usage (Y)                            36
# 0x09, 0x38,                    // Usage (Wheel)                        38
```

The mouse has also two physical positions (Usage (X), Usage (Y)) and a wheel (Usage (Wheel))

```
# 0x15, 0x81,                    // Logical Minimum (-127)               40
# 0x25, 0x7f,                    // Logical Maximum (127)                42
```

each of them can send values ranging from -127 up to including 127

```
# 0x75, 0x08,                    // Report Size (8)                      44
```

which is represented by eight bits

```
# 0x95, 0x03,                    // Report Count (3)                     46
```

and there are three of those eight bits, matching X, Y and Wheel.

```
# 0x81, 0x06,                    // Input (Data,Var,Rel)                 48
```

This time the data values are Relative (Rel), i.e. they represent the change from the previously sent report (event)

```
# 0xc0,                          // End Collection                       50
# 0xc0,                          // End Collection                       51
#
R: 52 05 01 09 02 a1 01 09 01 a1 00 05 09 19 01 29 03 15 00 25 01 75 01 95 03␣
↪81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81 25 7f 75 08 95 03 81␣
↪06 c0 c0
N: device 0:0
I: 3 0001 0001
```

This Report Descriptor tells us that the mouse input will be transmitted using four bytes: the first one for the buttons (three bits used, five for padding), the last three for the mouse X, Y and wheel changes, respectively.

Indeed, for any event, the mouse will send a *report* of four bytes. We can check the values sent by resorting e.g. to the *hid-recorder* tool, from hid-tools: The sequence of bytes sent by clicking and releasing button 1, then button 2, then button 3 is:

```
$ sudo ./hid-recorder /dev/hidraw1

....
output of hid-decode
....

#  Button: 1  0  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000000.000000 4 01 00 00 00
#  Button: 0  0  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000000.183949 4 00 00 00 00
#  Button: 0  1  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000001.959698 4 02 00 00 00
#  Button: 0  0  0 | # | X:    0 | Y:    0 | Wheel:    0
```

```
E: 000002.103899 4 00 00 00 00
#  Button: 0  0  1 | # | X:    0 | Y:    0 | Wheel:    0
E: 000004.855799 4 04 00 00 00
#  Button: 0  0  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000005.103864 4 00 00 00 00
```

This example shows that when button 2 is clicked, the bytes `02 00 00 00` are sent, and the immediately subsequent event (`00 00 00 00`) is the release of button 2 (no buttons are pressed, remember that the data values are *absolute*).

If instead one clicks and holds button 1, then clicks and holds button 2, releases button 1, and finally releases button 2, the reports are:

```
#  Button: 1  0  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000044.175830 4 01 00 00 00
#  Button: 1  1  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000045.975997 4 03 00 00 00
#  Button: 0  1  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000047.407930 4 02 00 00 00
#  Button: 0  0  0 | # | X:    0 | Y:    0 | Wheel:    0
E: 000049.199919 4 00 00 00 00
```

where with `03 00 00 00` both buttons are pressed, and with the subsequent `02 00 00 00` button 1 is released while button 2 is still active.

1.3.1 Output, Input and Feature Reports

HID devices can have Input Reports, like in the mouse example, Output Reports, and Feature Reports. “Output” means that the information is sent to the device. For example, a joystick with force feedback will have some output; the led of a keyboard would need an output as well. “Input” means that data come from the device.

“Feature”s are not meant to be consumed by the end user and define configuration options for the device. They can be queried from the host; when declared as *Volatile* they should be changed by the host.

1.4 Collections, Report IDs and Evdev events

A single device can logically group data into different independent sets, called a *Collection*. Collections can be nested and there are different types of collections (see the HID spec 6.2.2.6 “Collection, End Collection Items” for details).

Different reports are identified by means of different *Report ID* fields, i.e. a number identifying the structure of the immediately following report. Whenever a Report ID is needed it is transmitted as the first byte of any report. A device with only one supported HID report (like the mouse example above) may omit the report ID.

Consider the following HID report descriptor:

```
05 01 09 02 A1 01 85 01 05 09 19 01 29 05 15 00
25 01 95 05 75 01 81 02 95 01 75 03 81 01 05 01
```

```
09 30 09 31 16 00 F8 26 FF 07 75 0C 95 02 81 06
09 38 15 80 25 7F 75 08 95 01 81 06 05 0C 0A 38
02 15 80 25 7F 75 08 95 01 81 06 C0 05 01 09 02
A1 01 85 02 05 09 19 01 29 05 15 00 25 01 95 05
75 01 81 02 95 01 75 03 81 01 05 01 09 30 09 31
16 00 F8 26 FF 07 75 0C 95 02 81 06 09 38 15 80
25 7F 75 08 95 01 81 06 05 0C 0A 38 02 15 80 25
7F 75 08 95 01 81 06 C0 05 01 09 07 A1 01 85 05
05 07 15 00 25 01 09 29 09 3E 09 4B 09 4E 09 E3
09 E8 09 E8 09 E8 75 01 95 08 81 02 95 00 81 01
C0 05 0C 09 01 A1 01 85 06 15 00 25 01 75 01 95
01 09 3F 81 06 09 3F 81 06 09 3F 81 06 09 3F 81
06 09 3F 81 06 09 3F 81 06 09 3F 81 06 09 3F 81
06 C0 05 0C 09 01 A1 01 85 03 09 05 15 00 26 FF
00 75 08 95 02 B1 02 C0
```

After parsing it (try to parse it on your own using the suggested tools!) one can see that the device presents two `Mouse` Application Collections (with reports identified by Reports IDs 1 and 2, respectively), a `Keypad` Application Collection (whose report is identified by the Report ID 5) and two `Consumer Controls` Application Collections, (with Report IDs 6 and 3, respectively). Note, however, that a device can have different Report IDs for the same Application Collection.

The data sent will begin with the Report ID byte, and will be followed by the corresponding information. For example, the data transmitted for the last consumer control:

```
0x05, 0x0C,        // Usage Page (Consumer)
0x09, 0x01,        // Usage (Consumer Control)
0xA1, 0x01,        // Collection (Application)
0x85, 0x03,        //   Report ID (3)
0x09, 0x05,        //   Usage (Headphone)
0x15, 0x00,        //   Logical Minimum (0)
0x26, 0xFF, 0x00,  //   Logical Maximum (255)
0x75, 0x08,        //   Report Size (8)
0x95, 0x02,        //   Report Count (2)
0xB1, 0x02,        //   Feature (Data,Var,Abs,No Wrap,Linear,Preferred State,
↪No Null Position,Non-volatile)
0xC0,              // End Collection
```

will be of three bytes: the first for the Report ID (3), the next two for the headphone, with two (`Report Count (2)`) bytes (`Report Size (8)`), each ranging from 0 (`Logical Minimum (0)`) to 255 (`Logical Maximum (255)`).

All the Input data sent by the device should be translated into corresponding Evdev events, so that the remaining part of the stack can know what is going on, e.g. the bit for the first button translates into the `EV_KEY/BTN_LEFT` evdev event and relative X movement translates into the `EV_REL/REL_X` evdev event”.

1.5 Events

In Linux, one /dev/input/event* is created for each Application Collection. Going back to the mouse example, and repeating the sequence where one clicks and holds button 1, then clicks and holds button 2, releases button 1, and finally releases button 2, one gets:

```
$ sudo libinput record /dev/input/event1
# libinput record
version: 1
ndevices: 1
libinput:
  version: "1.23.0"
  git: "unknown"
system:
  os: "opensuse-tumbleweed:20230619"
  kernel: "6.3.7-1-default"
  dmi: "dmi:bvnHP:bvrU77Ver.01.05.00:bd03/24/2022:br5.0:efr20.
↪29:svnHP:pnHPEliteBook64514inchG9NotebookPC:pvr:rvnHP:rn89D2:rvrKBCVersion14.
↪1D.00:cvnHP:ct10:cvr:sku5Y3J1EA#ABZ:"
devices:
- node: /dev/input/event1
  evdev:
    # Name: PixArt HP USB Optical Mouse
    # ID: bus 0x3 vendor 0x3f0 product 0x94a version 0x111
    # Supported Events:
    # Event type 0 (EV_SYN)
    # Event type 1 (EV_KEY)
    #   Event code 272 (BTN_LEFT)
    #   Event code 273 (BTN_RIGHT)
    #   Event code 274 (BTN_MIDDLE)
    # Event type 2 (EV_REL)
    #   Event code 0 (REL_X)
    #   Event code 1 (REL_Y)
    #   Event code 8 (REL_WHEEL)
    #   Event code 11 (REL_WHEEL_HI_RES)
    # Event type 4 (EV_MSC)
    #   Event code 4 (MSC_SCAN)
    # Properties:
    name: "PixArt HP USB Optical Mouse"
    id: [3, 1008, 2378, 273]
    codes:
      0: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] # EV_SYN
      1: [272, 273, 274] # EV_KEY
      2: [0, 1, 8, 11] # EV_REL
      4: [4] # EV_MSC
    properties: []
  hid: [
    0x05, 0x01, 0x09, 0x02, 0xa1, 0x01, 0x09, 0x01, 0xa1, 0x00, 0x05, 0x09,␣
↪0x19, 0x01, 0x29, 0x03,
    0x15, 0x00, 0x25, 0x01, 0x95, 0x08, 0x75, 0x01, 0x81, 0x02, 0x05, 0x01,␣
↪0x09, 0x30, 0x09, 0x31,
    0x09, 0x38, 0x15, 0x81, 0x25, 0x7f, 0x75, 0x08, 0x95, 0x03, 0x81, 0x06, 0xc0, 0xc0
  ]
  udev:
    properties:
    - ID_INPUT=1
    - ID_INPUT_MOUSE=1
    - LIBINPUT_DEVICE_GROUP=3/3f0/94a:usb-0000:05:00.3-2
  quirks:
  events:
  # Current time is 12:31:56
  - evdev:
    - [  0,      0,   4,   4,      30] # EV_MSC / MSC_SCAN                 30 (obfuscated)
    - [  0,      0,   1, 272,       1] # EV_KEY / BTN_LEFT                  1
    - [  0,      0,   0,   0,       0] # ------------ SYN_REPORT (0) ---------- +0ms
  - evdev:
    - [  1, 207892,   4,   4,      30] # EV_MSC / MSC_SCAN                 30 (obfuscated)
    - [  1, 207892,   1, 273,       1] # EV_KEY / BTN_RIGHT                 1
    - [  1, 207892,   0,   0,       0] # ------------ SYN_REPORT (0) ---------- +1207ms
  - evdev:
    - [  2, 367823,   4,   4,      30] # EV_MSC / MSC_SCAN                 30 (obfuscated)
    - [  2, 367823,   1, 272,       0] # EV_KEY / BTN_LEFT                  0
    - [  2, 367823,   0,   0,       0] # ------------ SYN_REPORT (0) ---------- +1160ms
  # Current time is 12:32:00
  - evdev:
    - [  3, 247617,   4,   4,      30] # EV_MSC / MSC_SCAN                 30 (obfuscated)
    - [  3, 247617,   1, 273,       0] # EV_KEY / BTN_RIGHT                 0
    - [  3, 247617,   0,   0,       0] # ------------ SYN_REPORT (0) ---------- +880ms
```

Note: if `libinput record` is not available on your system try using `evemu-record`.

1.6 When something does not work

There can be a number of reasons why a device does not behave correctly. For example

- The HID report descriptor provided by the HID device may be wrong because e.g.
  - it does not follow the standard, so that the kernel will not able to make sense of the HID report descriptor;
  - the HID report descriptor *does not match* what is actually sent by the device (this can be verified by reading the raw HID data);

- the HID report descriptor may need some "quirks" (see later on).

As a consequence, a `/dev/input/event*` may not be created for each Application Collection, and/or the events there may not match what you would expect.

1.6.1 Quirks

There are some known peculiarities of HID devices that the kernel knows how to fix - these are called the HID quirks and a list of those is available in *include/linux/hid.h*.

Should this be the case, it should be enough to add the required quirk in the kernel, for the HID device at hand. This can be done in the file *drivers/hid/hid-quirks.c*. How to do it should be relatively straightforward after looking into the file.

The list of currently defined quirks, from *include/linux/hid.h*, is

HID_QUIRK_NOTOUCH:
HID_QUIRK_IGNORE: ignore this device
HID_QUIRK_NOGET:
HID_QUIRK_HIDDEV_FORCE:
HID_QUIRK_BADPAD:
HID_QUIRK_MULTI_INPUT:
HID_QUIRK_HIDINPUT_FORCE:
HID_QUIRK_ALWAYS_POLL:
HID_QUIRK_INPUT_PER_APP:
HID_QUIRK_X_INVERT:
HID_QUIRK_Y_INVERT:
HID_QUIRK_SKIP_OUTPUT_REPORTS:
HID_QUIRK_SKIP_OUTPUT_REPORT_ID:
HID_QUIRK_NO_OUTPUT_REPORTS_ON_INTR_EP:
HID_QUIRK_HAVE_SPECIAL_DRIVER:
HID_QUIRK_INCREMENT_USAGE_ON_DUPLICATE:
HID_QUIRK_FULLSPEED_INTERVAL:
HID_QUIRK_NO_INIT_REPORTS:
HID_QUIRK_NO_IGNORE:
HID_QUIRK_NO_INPUT_SYNC:

Quirks for USB devices can be specified while loading the usbhid module, see `modinfo usbhid`, although the proper fix should go into hid-quirks.c and **be submitted upstream**. See Documentation/process/submitting-patches.rst for guidelines on how to submit a patch. Quirks for other busses need to go into hid-quirks.c.

1.6.2 Fixing HID report descriptors

Should you need to patch HID report descriptors the easiest way is to resort to eBPF, as described in *HID-BPF*.

Basically, you can change any byte of the original HID report descriptor. The examples in samples/hid should be a good starting point for your code, see e.g. *samples/hid/hid_mouse.bpf.c*:

```
SEC("fmod_ret/hid_bpf_rdesc_fixup")
int BPF_PROG(hid_rdesc_fixup, struct hid_bpf_ctx *hctx)
{
  ....
     data[39] = 0x31;
     data[41] = 0x30;
  return 0;
}
```

Of course this can be also done within the kernel source code, see e.g. *drivers/hid/hid-aureal.c* or *drivers/hid/hid-samsung.c* for a slightly more complex file.

Check *Manual parsing of HID report descriptors* if you need any help navigating the HID manuals and understanding the exact meaning of the HID report descriptor hex numbers.

Whatever solution you come up with, please remember to **submit the fix to the HID maintainers**, so that it can be directly integrated in the kernel and that particular HID device will start working for everyone else. See Documentation/process/submitting-patches.rst for guidelines on how to do this.

1.6.3 Modifying the transmitted data on the fly

Using eBPF it is also possible to modify the data exchanged with the device. See again the examples in *samples/hid*.

Again, **please post your fix**, so that it can be integrated in the kernel!

1.6.4 Writing a specialized driver

This should really be your last resort.

CHAPTER

TWO

CARE AND FEEDING OF YOUR HUMAN INTERFACE DEVICES

2.1 Introduction

In addition to the normal input type HID devices, USB also uses the human interface device protocols for things that are not really human interfaces, but have similar sorts of communication needs. The two big examples for this are power devices (especially uninterruptible power supplies) and monitor control on higher end monitors.

To support these disparate requirements, the Linux USB system provides HID events to two separate interfaces: * the input subsystem, which converts HID events into normal input device interfaces (such as keyboard, mouse and joystick) and a normalised event interface - see Documentation/input/input.rst * the hiddev interface, which provides fairly raw HID events

The data flow for a HID event produced by a device is something like the following:

```
usb.c ---> hid-core.c  ----> hid-input.c ----> [keyboard/mouse/joystick/event]
                        |
                        |
                         --> hiddev.c ----> POWER / MONITOR CONTROL
```

In addition, other subsystems (apart from USB) can potentially feed events into the input subsystem, but these have no effect on the HID device interface.

2.2 Using the HID Device Interface

The hiddev interface is a char interface using the normal USB major, with the minor numbers starting at 96 and finishing at 111. Therefore, you need the following commands:

```
mknod /dev/usb/hiddev0 c 180 96
mknod /dev/usb/hiddev1 c 180 97
mknod /dev/usb/hiddev2 c 180 98
mknod /dev/usb/hiddev3 c 180 99
mknod /dev/usb/hiddev4 c 180 100
mknod /dev/usb/hiddev5 c 180 101
mknod /dev/usb/hiddev6 c 180 102
mknod /dev/usb/hiddev7 c 180 103
mknod /dev/usb/hiddev8 c 180 104
mknod /dev/usb/hiddev9 c 180 105
mknod /dev/usb/hiddev10 c 180 106
mknod /dev/usb/hiddev11 c 180 107
mknod /dev/usb/hiddev12 c 180 108
mknod /dev/usb/hiddev13 c 180 109
mknod /dev/usb/hiddev14 c 180 110
mknod /dev/usb/hiddev15 c 180 111
```

So you point your hiddev compliant user-space program at the correct interface for your device, and it all just works.

Assuming that you have a hiddev compliant user-space program, of course. If you need to write one, read on.

2.3 The HIDDEV API

This description should be read in conjunction with the HID specification, freely available from https://www.usb.org, and conveniently linked of http://www.linux-usb.org.

The hiddev API uses a read() interface, and a set of ioctl() calls.

HID devices exchange data with the host computer using data bundles called “reports”. Each report is divided into “fields”, each of which can have one or more “usages”. In the hid-core, each one of these usages has a single signed 32-bit value.

2.3.1 read():

This is the event interface. When the HID device’s state changes, it performs an interrupt transfer containing a report which contains the changed value. The hid-core.c module parses the report, and returns to hiddev.c the individual usages that have changed within the report. In its basic mode, the hiddev will make these individual usage changes available to the reader using a struct hiddev_event:

```
struct hiddev_event {
    unsigned hid;
    signed int value;
};
```

containing the HID usage identifier for the status that changed, and the value that it was changed to. Note that the structure is defined within <linux/hiddev.h>, along with some other useful #defines and structures. The HID usage identifier is a composite of the HID usage page shifted to the 16 high order bits ORed with the usage code. The behavior of the read() function can be modified using the HIDIOCSFLAG ioctl() described below.

2.3.2 ioctl():

This is the control interface. There are a number of controls:

HIDIOCGVERSION

- int (read)

Gets the version code out of the hiddev driver.

HIDIOCAPPLICATION

- (none)

This ioctl call returns the HID application usage associated with the HID device. The third argument to ioctl() specifies which application index to get. This is useful when the device has more than one application collection. If the index is invalid (greater or equal to the number of application collections this device has) the ioctl returns -1. You can find out beforehand how many application collections the device has from the num_applications field from the hiddev_devinfo structure.

HIDIOCGCOLLECTIONINFO

- struct hiddev_collection_info (read/write)

This returns a superset of the information above, providing not only application collections, but all the collections the device has. It also returns the level the collection lives in the hierarchy. The user passes in a hiddev_collection_info struct with the index field set to the index that should be returned. The ioctl fills in the other fields. If the index is larger than the last collection index, the ioctl returns -1 and sets errno to -EINVAL.

HIDIOCGDEVINFO

- struct hiddev_devinfo (read)

Gets a hiddev_devinfo structure which describes the device.

HIDIOCGSTRING

- struct hiddev_string_descriptor (read/write)

Gets a string descriptor from the device. The caller must fill in the "index" field to indicate which descriptor should be returned.

HIDIOCINITREPORT

- (none)

Instructs the kernel to retrieve all input and feature report values from the device. At this point, all the usage structures will contain current values for the device, and will maintain it as the device changes. Note that the use of this ioctl is unnecessary in general, since later kernels automatically initialize the reports from the device at attach time.

HIDIOCGNAME

- string (variable length)

Gets the device name

HIDIOCGREPORT

- struct hiddev_report_info (write)

Instructs the kernel to get a feature or input report from the device, in order to selectively update the usage structures (in contrast to INITREPORT).

HIDIOCSREPORT

- struct hiddev_report_info (write)

Instructs the kernel to send a report to the device. This report can be filled in by the user through HIDIOCSUSAGE calls (below) to fill in individual usage values in the report before sending the report in full to the device.

HIDIOCGREPORTINFO

- struct hiddev_report_info (read/write)

Fills in a hiddev_report_info structure for the user. The report is looked up by type (input, output or feature) and id, so these fields must be filled in by the user. The ID can be absolute -- the actual report id as reported by the device -- or relative -- HID_REPORT_ID_FIRST for the first report, and (HID_REPORT_ID_NEXT | report_id) for the next report after report_id. Without a priori information about report ids, the right way to use this ioctl is to use the relative IDs above to enumerate the valid IDs. The ioctl returns non-zero when there is no more next ID. The real report ID is filled into the returned hiddev_report_info structure.

HIDIOCGFIELDINFO

- struct hiddev_field_info (read/write)

Returns the field information associated with a report in a hiddev_field_info structure. The user must fill in report_id and report_type in this structure, as above. The field_index should also be filled in, which should be a number from 0 and maxfield-1, as returned from a previous HIDIOCGREPORTINFO call.

HIDIOCGUCODE

- struct hiddev_usage_ref (read/write)

Returns the usage_code in a hiddev_usage_ref structure, given that its report type, report id, field index, and index within the field have already been filled into the structure.

HIDIOCGUSAGE

- struct hiddev_usage_ref (read/write)

Returns the value of a usage in a hiddev_usage_ref structure. The usage to be retrieved can be specified as above, or the user can choose to fill in the report_type field and specify the report_id as HID_REPORT_ID_UNKNOWN. In this case, the hiddev_usage_ref will be filled in with the report and field information associated with this usage if it is found.

HIDIOCSUSAGE

- struct hiddev_usage_ref (write)

Sets the value of a usage in an output report. The user fills in the hiddev_usage_ref structure as above, but additionally fills in the value field.

HIDIOGCOLLECTIONINDEX

- struct hiddev_usage_ref (write)

Returns the collection index associated with this usage. This indicates where in the collection hierarchy this usage sits.

HIDIOCGFLAG

- int (read)

HIDIOCSFLAG

- int (write)

These operations respectively inspect and replace the mode flags that influence the read() call above. The flags are as follows:

HIDDEV_FLAG_UREF

- read() calls will now return struct hiddev_usage_ref instead of struct hiddev_event. This is a larger structure, but in situations where the device has more than one usage in its reports with the same usage code, this mode serves to resolve such ambiguity.

HIDDEV_FLAG_REPORT

- This flag can only be used in conjunction with HIDDEV_FLAG_UREF. With this flag set, when the device sends a report, a struct hiddev_usage_ref will be returned to read() filled in with the report_type and report_id, but with field_index set to FIELD_INDEX_NONE. This serves as additional notification when the device has sent a report.

CHAPTER

THREE

HIDRAW - RAW ACCESS TO USB AND BLUETOOTH HUMAN INTERFACE DEVICES

The hidraw driver provides a raw interface to USB and Bluetooth Human Interface Devices (HIDs). It differs from hiddev in that reports sent and received are not parsed by the HID parser, but are sent to and received from the device unmodified.

Hidraw should be used if the userspace application knows exactly how to communicate with the hardware device, and is able to construct the HID reports manually. This is often the case when making userspace drivers for custom HID devices.

Hidraw is also useful for communicating with non-conformant HID devices which send and receive data in a way that is inconsistent with their report descriptors. Because hiddev parses reports which are sent and received through it, checking them against the device's report descriptor, such communication with these non-conformant devices is impossible using hiddev. Hidraw is the only alternative, short of writing a custom kernel driver, for these non-conformant devices.

A benefit of hidraw is that its use by userspace applications is independent of the underlying hardware type. Currently, hidraw is implemented for USB and Bluetooth. In the future, as new hardware bus types are developed which use the HID specification, hidraw will be expanded to add support for these new bus types.

Hidraw uses a dynamic major number, meaning that udev should be relied on to create hidraw device nodes. Udev will typically create the device nodes directly under /dev (eg: /dev/hidraw0). As this location is distribution- and udev rule-dependent, applications should use libudev to locate hidraw devices attached to the system. There is a tutorial on libudev with a working example at:

```
http://www.signal11.us/oss/udev/
https://web.archive.org/web/2019*/www.signal11.us
```

3.1 The HIDRAW API

3.2 read()

read() will read a queued report received from the HID device. On USB devices, the reports read using read() are the reports sent from the device on the INTERRUPT IN endpoint. By default, read() will block until there is a report available to be read. read() can be made non-blocking, by passing the O_NONBLOCK flag to open(), or by setting the O_NONBLOCK flag using fcntl().

On a device which uses numbered reports, the first byte of the returned data will be the report number; the report data follows, beginning in the second byte. For devices which do not use numbered reports, the report data will begin at the first byte.

3.3 write()

The write() function will write a report to the device. For USB devices, if the device has an INTERRUPT OUT endpoint, the report will be sent on that endpoint. If it does not, the report will be sent over the control endpoint, using a SET_REPORT transfer.

The first byte of the buffer passed to write() should be set to the report number. If the device does not use numbered reports, the first byte should be set to 0. The report data itself should begin at the second byte.

3.4 ioctl()

Hidraw supports the following ioctls:

HIDIOCGRDESCSIZE:
Get Report Descriptor Size

This ioctl will get the size of the device's report descriptor.

HIDIOCGRDESC:
Get Report Descriptor

This ioctl returns the device's report descriptor using a hidraw_report_descriptor struct. Make sure to set the size field of the hidraw_report_descriptor struct to the size returned from HIDIOCGRDESCSIZE.

HIDIOCGRAWINFO:
Get Raw Info

This ioctl will return a hidraw_devinfo struct containing the bus type, the vendor ID (VID), and product ID (PID) of the device. The bus type can be one of:

```
- BUS_USB
- BUS_HIL
- BUS_BLUETOOTH
- BUS_VIRTUAL
```

which are defined in uapi/linux/input.h.

HIDIOCGRAWNAME(len):
Get Raw Name

This ioctl returns a string containing the vendor and product strings of the device. The returned string is Unicode, UTF-8 encoded.

HIDIOCGRAWPHYS(len):
Get Physical Address

This ioctl returns a string representing the physical address of the device. For USB devices, the string contains the physical path to the device (the USB controller, hubs, ports, etc). For Bluetooth devices, the string contains the hardware (MAC) address of the device.

HIDIOCSFEATURE(len):
Send a Feature Report

This ioctl will send a feature report to the device. Per the HID specification, feature reports are always sent using the control endpoint. Set the first byte of the supplied buffer to the report number. For devices which do not use numbered reports, set the first byte to 0. The report data begins in the second byte. Make sure to set len accordingly, to one more than the length of the report (to account for the report number).

HIDIOCGFEATURE(len):
Get a Feature Report

This ioctl will request a feature report from the device using the control endpoint. The first byte of the supplied buffer should be set to the report number of the requested report. For devices which do not use numbered reports, set the first byte to 0. The returned report buffer will contain the report number in the first byte, followed by the report data read from the device. For devices which do not use numbered reports, the report data will begin at the first byte of the returned buffer.

HIDIOCSINPUT(len):
Send an Input Report

This ioctl will send an input report to the device, using the control endpoint. In most cases, setting an input HID report on a device is meaningless and has no effect, but some devices may choose to use this to set or reset an initial state of a report. The format of the buffer issued with this report is identical to that of HIDIOCSFEATURE.

HIDIOCGINPUT(len):
Get an Input Report

This ioctl will request an input report from the device using the control endpoint. This is slower on most devices where a dedicated In endpoint exists for regular input reports, but allows the host to request the value of a specific report number. Typically, this is used to request the initial states of an input report of a device, before an application listens for normal reports via the regular device read() interface. The format of the buffer issued with this report is identical to that of HIDIOCGFEATURE.

HIDIOCSOUTPUT(len):
Send an Output Report

This ioctl will send an output report to the device, using the control endpoint. This is slower on most devices where a dedicated Out endpoint exists for regular output reports, but is added for completeness. Typically, this is used to set the initial states of an output report of a device, before an application sends updates via the regular device write() interface. The format of the buffer issued with this report is identical to that of HIDIOCSFEATURE.

HIDIOCGOUTPUT(len):
Get an Output Report

This ioctl will request an output report from the device using the control endpoint. Typically, this is used to retrieve the initial state of an output report of a device, before an application updates it as necessary either via a HIDIOCSOUTPUT request, or the regular device write() interface. The format of the buffer issued with this report is identical to that of HIDIOCGFEATURE.

3.5 Example

In samples/, find hid-example.c, which shows examples of read(), write(), and all the ioctls for hidraw. The code may be used by anyone for any purpose, and can serve as a starting point for developing applications using hidraw.

Document by:

Alan Ott <alan@signal11.us>, Signal 11 Software

CHAPTER

FOUR

HID SENSORS FRAMEWORK

HID sensor framework provides necessary interfaces to implement sensor drivers, which are connected to a sensor hub. The sensor hub is a HID device and it provides a report descriptor conforming to HID 1.12 sensor usage tables.

Description from the HID 1.12 “HID Sensor Usages” specification: “Standardization of HID usages for sensors would allow (but not require) sensor hardware vendors to provide a consistent Plug And Play interface at the USB boundary, thereby enabling some operating systems to incorporate common device drivers that could be reused between vendors, alleviating any need for the vendors to provide the drivers themselves.”

This specification describes many usage IDs, which describe the type of sensor and also the individual data fields. Each sensor can have variable number of data fields. The length and order is specified in the report descriptor. For example a part of report descriptor can look like:

```
   INPUT(1)[INPUT]
 ..
    Field(2)
      Physical(0020.0073)
      Usage(1)
        0020.045f
      Logical Minimum(-32767)
      Logical Maximum(32767)
      Report Size(8)
      Report Count(1)
      Report Offset(16)
      Flags(Variable Absolute)
..
..
```

The report is indicating “sensor page (0x20)” contains an accelerometer-3D (0x73). This accelerometer-3D has some fields. Here for example field 2 is motion intensity (0x045f) with a logical minimum value of -32767 and logical maximum of 32767. The order of fields and length of each field is important as the input event raw data will use this format.

4.1 Implementation

This specification defines many different types of sensors with different sets of data fields. It is difficult to have a common input event to user space applications, for different sensors. For example an accelerometer can send X,Y and Z data, whereas an ambient light sensor can send illumination data. So the implementation has two parts:

- Core HID driver
- Individual sensor processing part (sensor drivers)

4.1.1 Core driver

The core driver (hid-sensor-hub) registers as a HID driver. It parses report descriptors and identifies all the sensors present. It adds an MFD device with name HID-SENSOR-xxxx (where xxxx is usage id from the specification).

For example:

HID-SENSOR-200073 is registered for an Accelerometer 3D driver.

So if any driver with this name is inserted, then the probe routine for that function will be called. So an accelerometer processing driver can register with this name and will be probed if there is an accelerometer-3D detected.

The core driver provides a set of APIs which can be used by the processing drivers to register and get events for that usage id. Also it provides parsing functions, which get and set each input/feature/output report.

4.1.2 Individual sensor processing part (sensor drivers)

The processing driver will use an interface provided by the core driver to parse the report and get the indexes of the fields and also can get events. This driver can use IIO interface to use the standard ABI defined for a type of sensor.

4.2 Core driver Interface

Callback structure:

```
Each processing driver can use this structure to set some callbacks.
      int (*suspend)(..): Callback when HID suspend is received
      int (*resume)(..): Callback when HID resume is received
      int (*capture_sample)(..): Capture a sample for one of its data fields
      int (*send_event)(..): One complete event is received which can have
                             multiple data fields.
```

Registration functions:

```
int sensor_hub_register_callback(struct hid_sensor_hub_device *hsdev,
                      u32 usage_id,
                      struct hid_sensor_hub_callbacks *usage_callback):
```

Registers callbacks for a usage id. The callback functions are not allowed to sleep:

```
int sensor_hub_remove_callback(struct hid_sensor_hub_device *hsdev,
                      u32 usage_id):
```

Removes callbacks for a usage id.

Parsing function:

```
int sensor_hub_input_get_attribute_info(struct hid_sensor_hub_device *hsdev,
                      u8 type,
                      u32 usage_id, u32 attr_usage_id,
                      struct hid_sensor_hub_attribute_info *info);
```

A processing driver can look for some field of interest and check if it exists in a report descriptor. If it exists it will store necessary information so that fields can be set or get individually. These indexes avoid searching every time and getting field index to get or set.

Set Feature report:

```
int sensor_hub_set_feature(struct hid_sensor_hub_device *hsdev, u32 report_id,
                      u32 field_index, s32 value);
```

This interface is used to set a value for a field in feature report. For example if there is a field report_interval, which is parsed by a call to sensor_hub_input_get_attribute_info before, then it can directly set that individual field:

```
int sensor_hub_get_feature(struct hid_sensor_hub_device *hsdev, u32 report_id,
                      u32 field_index, s32 *value);
```

This interface is used to get a value for a field in input report. For example if there is a field report_interval, which is parsed by a call to sensor_hub_input_get_attribute_info before, then it can directly get that individual field value:

```
int sensor_hub_input_attr_get_raw_value(struct hid_sensor_hub_device *hsdev,
                      u32 usage_id,
                      u32 attr_usage_id, u32 report_id);
```

This is used to get a particular field value through input reports. For example accelerometer wants to poll X axis value, then it can call this function with the usage id of X axis. HID sensors can provide events, so this is not necessary to poll for any field. If there is some new sample, the core driver will call registered callback function to process the sample.

4.2.1 HID Custom and generic Sensors

HID Sensor specification defines two special sensor usage types. Since they don't represent a standard sensor, it is not possible to define using Linux IIO type interfaces. The purpose of these sensors is to extend the functionality or provide a way to obfuscate the data being communicated by a sensor. Without knowing the mapping between the data and its encapsulated form, it is difficult for an application/driver to determine what data is being communicated by the sensor. This allows some differentiating use cases, where vendor can provide applications. Some common use cases are debug other sensors or to provide some events like keyboard attached/detached or lid open/close.

To allow application to utilize these sensors, here they are exported using sysfs attribute groups, attributes and misc device interface.

An example of this representation on sysfs:

```
/sys/devices/pci0000:00/INT33C2:00/i2c-0/i2c-INT33D1:00/0018:8086:09FA.0001/
↪HID-SENSOR-2000e1.6.auto$ tree -R
.
│   ├──  enable_sensor
│   │   ├── feature-0-200316
│   │   │   ├── feature-0-200316-maximum
│   │   │   ├── feature-0-200316-minimum
│   │   │   ├── feature-0-200316-name
│   │   │   ├── feature-0-200316-size
│   │   │   ├── feature-0-200316-unit-expo
│   │   │   ├── feature-0-200316-units
│   │   │   ├── feature-0-200316-value
│   │   ├── feature-1-200201
│   │   │   ├── feature-1-200201-maximum
│   │   │   ├── feature-1-200201-minimum
│   │   │   ├── feature-1-200201-name
│   │   │   ├── feature-1-200201-size
│   │   │   ├── feature-1-200201-unit-expo
│   │   │   ├── feature-1-200201-units
│   │   │   ├── feature-1-200201-value
│   │   ├── input-0-200201
│   │   │   ├── input-0-200201-maximum
│   │   │   ├── input-0-200201-minimum
│   │   │   ├── input-0-200201-name
│   │   │   ├── input-0-200201-size
│   │   │   ├── input-0-200201-unit-expo
│   │   │   ├── input-0-200201-units
│   │   │   ├── input-0-200201-value
│   │   ├── input-1-200202
│   │   │   ├── input-1-200202-maximum
│   │   │   ├── input-1-200202-minimum
│   │   │   ├── input-1-200202-name
│   │   │   ├── input-1-200202-size
│   │   │   ├── input-1-200202-unit-expo
│   │   │   ├── input-1-200202-units
│   │   │   ├── input-1-200202-value
```

Here there is a custom sensor with four fields: two feature and two inputs. Each field is represented by a set of attributes. All fields except the "value" are read only. The value field is a read-write field.

Example:

```
/sys/bus/platform/devices/HID-SENSOR-2000e1.6.auto/feature-0-200316$ grep -r .␣
↪*
feature-0-200316-maximum:6
feature-0-200316-minimum:0
feature-0-200316-name:property-reporting-state
feature-0-200316-size:1
feature-0-200316-unit-expo:0
feature-0-200316-units:25
feature-0-200316-value:1
```

How to enable such sensor?

By default sensor can be power gated. To enable sysfs attribute "enable" can be used:

```
$ echo 1 > enable_sensor
```

Once enabled and powered on, sensor can report value using HID reports. These reports are pushed using misc device interface in a FIFO order:

```
/dev$ tree | grep HID-SENSOR-2000e1.6.auto
│   │   │   ├── 10:53 -> ../HID-SENSOR-2000e1.6.auto
│   ├──  HID-SENSOR-2000e1.6.auto
```

Each report can be of variable length preceded by a header. This header consists of a 32-bit usage id, 64-bit time stamp and 32-bit length field of raw data.

CHAPTER

FIVE

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.

5.1 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.

```
+-----------+  +-----------+            +-----------+  +-----------+
| Device #1 |  | Device #i |            | Device #j |  | Device #k |
+-----------+  +-----------+            +-----------+  +-----------+
         \\      //                              \\      //
       +------------+                          +------------+
       | I/O Driver |                          | I/O Driver |
       +------------+                          +------------+
             ||                                      ||
    +------------------+                    +------------------+
    | Transport Driver |                    | Transport Driver |
    +------------------+                    +------------------+
                      \___                ___/
                          \              /
                         +----------------+
                         |    HID Core    |
                         +----------------+
                          /  |        |  \
                         /   |        |   \
            ____________/    |        |    \_________________
           /                 |        |                      \
          /                  |        |                       \
+----------------+  +-----------+  +------------------+  +------------------+
| Generic Driver |  | MT Driver |  | Custom Driver #1 |  | Custom Driver #2 |
+----------------+  +-----------+  +------------------+  +------------------+
```

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.

5.1.1 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.

5.1.2 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 inde-

pendent 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.

5.2 2) HID API

5.2.1 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.

5.2.2 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!

5.2.3 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>

CHAPTER

SIX

HID-BPF

HID is a standard protocol for input devices but some devices may require custom tweaks, traditionally done with a kernel driver fix. Using the eBPF capabilities instead speeds up development and adds new capabilities to the existing HID interfaces.

6.1 When (and why) to use HID-BPF

There are several use cases when using HID-BPF is better than standard kernel driver fix:

6.1.1 Dead zone of a joystick

Assuming you have a joystick that is getting older, it is common to see it wobbling around its neutral point. This is usually filtered at the application level by adding a *dead zone* for this specific axis.

With HID-BPF, we can apply this filtering in the kernel directly so userspace does not get woken up when nothing else is happening on the input controller.

Of course, given that this dead zone is specific to an individual device, we can not create a generic fix for all of the same joysticks. Adding a custom kernel API for this (e.g. by adding a sysfs entry) does not guarantee this new kernel API will be broadly adopted and maintained.

HID-BPF allows the userspace program to load the program itself, ensuring we only load the custom API when we have a user.

6.1.2 Simple fixup of report descriptor

In the HID tree, half of the drivers only fix one key or one byte in the report descriptor. These fixes all require a kernel patch and the subsequent shepherding into a release, a long and painful process for users.

We can reduce this burden by providing an eBPF program instead. Once such a program has been verified by the user, we can embed the source code into the kernel tree and ship the eBPF program and load it directly instead of loading a specific kernel module for it.

Note: distribution of eBPF programs and their inclusion in the kernel is not yet fully implemented

6.1.3 Add a new feature that requires a new kernel API

An example for such a feature are the Universal Stylus Interface (USI) pens. Basically, USI pens require a new kernel API because there are new channels of communication that our HID and input stack do not support. Instead of using hidraw or creating new sysfs entries or ioctls, we can rely on eBPF to have the kernel API controlled by the consumer and to not impact the performances by waking up userspace every time there is an event.

6.1.4 Morph a device into something else and control that from userspace

The kernel has a relatively static mapping of HID items to evdev bits. It cannot decide to dynamically transform a given device into something else as it does not have the required context and any such transformation cannot be undone (or even discovered) by userspace.

However, some devices are useless with that static way of defining devices. For example, the Microsoft Surface Dial is a pushbutton with haptic feedback that is barely usable as of today.

With eBPF, userspace can morph that device into a mouse, and convert the dial events into wheel events. Also, the userspace program can set/unset the haptic feedback depending on the context. For example, if a menu is visible on the screen we likely need to have a haptic click every 15 degrees. But when scrolling in a web page the user experience is better when the device emits events at the highest resolution.

6.1.5 Firewall

What if we want to prevent other users to access a specific feature of a device? (think a possibly broken firmware update entry point)

With eBPF, we can intercept any HID command emitted to the device and validate it or not.

This also allows to sync the state between the userspace and the kernel/bpf program because we can intercept any incoming command.

6.1.6 Tracing

The last usage is tracing events and all the fun we can do we BPF to summarize and analyze events.

Right now, tracing relies on hidraw. It works well except for a couple of issues:

1. if the driver doesn't export a hidraw node, we can't trace anything (eBPF will be a "god-mode" there, so this may raise some eyebrows)
2. hidraw doesn't catch other processes' requests to the device, which means that we have cases where we need to add printks to the kernel to understand what is happening.

6.2 High-level view of HID-BPF

The main idea behind HID-BPF is that it works at an array of bytes level. Thus, all of the parsing of the HID report and the HID report descriptor must be implemented in the userspace component that loads the eBPF program.

For example, in the dead zone joystick from above, knowing which fields in the data stream needs to be set to `0` needs to be computed by userspace.

A corollary of this is that HID-BPF doesn't know about the other subsystems available in the kernel. *You can not directly emit input event through the input API from eBPF*.

When a BPF program needs to emit input events, it needs to talk with the HID protocol, and rely on the HID kernel processing to translate the HID data into input events.

6.3 Available types of programs

HID-BPF is built "on top" of BPF, meaning that we use tracing method to declare our programs.

HID-BPF has the following attachment types available:

1. event processing/filtering with `SEC("fmod_ret/hid_bpf_device_event")` in libbpf
2. actions coming from userspace with `SEC("syscall")` in libbpf
3. change of the report descriptor with `SEC("fmod_ret/hid_bpf_rdesc_fixup")` in libbpf

A hid_bpf_device_event is calling a BPF program when an event is received from the device. Thus we are in IRQ context and can act on the data or notify userspace. And given that we are in IRQ context, we can not talk back to the device.

A syscall means that userspace called the syscall BPF_PROG_RUN facility. This time, we can do any operations allowed by HID-BPF, and talking to the device is allowed.

Last, hid_bpf_rdesc_fixup is different from the others as there can be only one BPF program of this type. This is called on probe from the driver and allows to change the report descriptor from the BPF program. Once a hid_bpf_rdesc_fixup program has been loaded, it is not possible to overwrite it unless the program which inserted it allows us by pinning the program and closing all of its fds pointing to it.

6.4 Developer API:

6.4.1 User API data structures available in programs:

struct **hid_bpf_ctx**

User accessible data for all HID programs

Definition:

```
struct hid_bpf_ctx {
    __u32 index;
    const struct hid_device *hid;
    __u32 allocated_size;
    enum hid_report_type report_type;
    union {
        __s32 retval;
        __s32 size;
    };
};
```

Members

index

program index in the jump table. No special meaning (a smaller index doesn't mean the program will be executed before another program with a bigger index).

hid

the struct hid_device representing the device itself

allocated_size

Allocated size of data.

This is how much memory is available and can be requested by the HID program. Note that for HID_BPF_RDESC_FIXUP, that memory is set to 4096 (4 KB)

report_type

used for hid_bpf_device_event()

{unnamed_union}

anonymous

retval
Return value of the previous program.

size
Valid data in the data field.

Programs can get the available valid size in data by fetching this field. Programs can also change this value by returning a positive number in the program. To discard the event, return a negative error code.

`size` must always be less or equal than `allocated_size` (it is enforced once all BPF programs have been run).

Description

`data` is not directly accessible from the context. We need to issue a call to `hid_bpf_get_data()` in order to get a pointer to that field.

All of these fields are currently read-only.

enum **hid_bpf_attach_flags**
flags used when attaching a HIF-BPF program

Constants

HID_BPF_FLAG_NONE
no specific flag is used, the kernel choses where to insert the program

HID_BPF_FLAG_INSERT_HEAD
insert the given program before any other program currently attached to the device. This doesn't guarantee that this program will always be first

HID_BPF_FLAG_MAX
sentinel value, not to be used by the callers

6.4.2 Available tracing functions to attach a HID-BPF program:

noinline int **hid_bpf_device_event**(struct *hid_bpf_ctx* *ctx)
Called whenever an event is coming in from the device

Parameters

struct hid_bpf_ctx *ctx
The HID-BPF context

Description

return `0` on success and keep processing; a positive value to change the incoming size buffer; a negative error code to interrupt the processing of this event

Declare an `fmod_ret` tracing bpf program to this function and attach this program through *hid_bpf_attach_prog()* to have this helper called for any incoming event from the device itself.

The function is called while on IRQ context, so we can not sleep.

noinline int **hid_bpf_rdesc_fixup**(struct *hid_bpf_ctx* *ctx)
Called when the probe function parses the report descriptor of the HID device

Parameters

`struct hid_bpf_ctx *ctx`
The HID-BPF context

Description

return 0 on success and keep processing; a positive value to change the incoming size buffer; a negative error code to interrupt the processing of this event

Declare an `fmod_ret` tracing bpf program to this function and attach this program through *`hid_bpf_attach_prog()`* to have this helper called before any parsing of the report descriptor by HID.

6.4.3 Available API that can be used in all HID-BPF programs:

__u8 ***`hid_bpf_get_data`**(struct *hid_bpf_ctx* *ctx, unsigned int offset, const size_t rdwr_buf_size)

Get the kernel memory pointer associated with the context **ctx**

Parameters

`struct hid_bpf_ctx *ctx`
The HID-BPF context

`unsigned int offset`
The offset within the memory

`const size_t rdwr_buf_size`
the const size of the buffer

Description

returns `NULL` on error, an __u8 memory pointer on success

6.4.4 Available API that can be used in syscall HID-BPF programs:

int **`hid_bpf_attach_prog`**(unsigned int hid_id, int prog_fd, __u32 flags)

Attach the given **prog_fd** to the given HID device

Parameters

`unsigned int hid_id`
the system unique identifier of the HID device

`int prog_fd`
an fd in the user process representing the program to attach

`__u32 flags`
any logical OR combination of *`enum hid_bpf_attach_flags`*

Description

returns an fd of a bpf_link object on success (> `0`), an error code otherwise. Closing this fd will detach the program from the HID device (unless the bpf_link is pinned to the BPF file system).

struct *hid_bpf_ctx* ***hid_bpf_allocate_context**(unsigned int hid_id)

Allocate a context to the given HID device

Parameters

unsigned int hid_id
the system unique identifier of the HID device

Description

returns A pointer to *struct hid_bpf_ctx* on success, NULL on error.

void **hid_bpf_release_context**(struct *hid_bpf_ctx* *ctx)

Release the previously allocated context **ctx**

Parameters

struct hid_bpf_ctx *ctx
the HID-BPF context to release

int **hid_bpf_hw_request**(struct *hid_bpf_ctx* *ctx, __u8 *buf, size_t buf__sz, enum hid_report_type rtype, enum hid_class_request reqtype)

Communicate with a HID device

Parameters

struct hid_bpf_ctx *ctx
the HID-BPF context previously allocated in *hid_bpf_allocate_context()*

__u8 *buf
a PTR_TO_MEM buffer

size_t buf__sz
the size of the data to transfer

enum hid_report_type rtype
the type of the report (HID_INPUT_REPORT, HID_FEATURE_REPORT, HID_OUTPUT_REPORT)

enum hid_class_request reqtype
the type of the request (HID_REQ_GET_REPORT, HID_REQ_SET_REPORT, ...)

Description

returns 0 on success, a negative error code otherwise.

6.5 General overview of a HID-BPF program

6.5.1 Accessing the data attached to the context

The struct hid_bpf_ctx doesn't export the data fields directly and to access it, a bpf program needs to first call *hid_bpf_get_data()*.

offset can be any integer, but size needs to be constant, known at compile time.

This allows the following:

1. for a given device, if we know that the report length will always be of a certain value, we can request the data pointer to point at the full report length.

The kernel will ensure we are using a correct size and offset and eBPF will ensure the code will not attempt to read or write outside of the boundaries:

```
__u8 *data = hid_bpf_get_data(ctx, 0 /* offset */, 256 /* size */);

if (!data)
    return 0; /* ensure data is correct, now the verifier knows we
               * have 256 bytes available */

bpf_printk("hello world: %02x %02x %02x", data[0], data[128], data[255]);
```

2. if the report length is variable, but we know the value of X is always a 16-bit integer, we can then have a pointer to that value only:

```
__u16 *x = hid_bpf_get_data(ctx, offset, sizeof(*x));

if (!x)
    return 0; /* something went wrong */

*x += 1; /* increment X by one */
```

6.5.2 Effect of a HID-BPF program

For all HID-BPF attachment types except for *hid_bpf_rdesc_fixup()*, several eBPF programs can be attached to the same device.

Unless HID_BPF_FLAG_INSERT_HEAD is added to the flags while attaching the program, the new program is appended at the end of the list. HID_BPF_FLAG_INSERT_HEAD will insert the new program at the beginning of the list which is useful for e.g. tracing where we need to get the unprocessed events from the device.

Note that if there are multiple programs using the HID_BPF_FLAG_INSERT_HEAD flag, only the most recently loaded one is actually the first in the list.

SEC("fmod_ret/hid_bpf_device_event")

Whenever a matching event is raised, the eBPF programs are called one after the other and are working on the same data buffer.

If a program changes the data associated with the context, the next one will see the modified data but it will have *no* idea of what the original data was.

Once all the programs are run and return 0 or a positive value, the rest of the HID stack will work on the modified data, with the size field of the last hid_bpf_ctx being the new size of the input stream of data.

A BPF program returning a negative error discards the event, i.e. this event will not be processed by the HID stack. Clients (hidraw, input, LEDs) will **not** see this event.

SEC("syscall")

syscall are not attached to a given device. To tell which device we are working with, userspace needs to refer to the device by its unique system id (the last 4 numbers in the sysfs path: /sys/bus/hid/devices/xxxx:yyyy:zzzz:0000).

To retrieve a context associated with the device, the program must call *hid_bpf_allocate_context()* and must release it with *hid_bpf_release_context()* before returning. Once the context is retrieved, one can also request a pointer to kernel memory with *hid_bpf_get_data()*. This memory is big enough to support all input/output/feature reports of the given device.

SEC("fmod_ret/hid_bpf_rdesc_fixup")

The hid_bpf_rdesc_fixup program works in a similar manner to .report_fixup of struct hid_driver.

When the device is probed, the kernel sets the data buffer of the context with the content of the report descriptor. The memory associated with that buffer is HID_MAX_DESCRIPTOR_SIZE (currently 4kB).

The eBPF program can modify the data buffer at-will and the kernel uses the modified content and size as the report descriptor.

Whenever a SEC("fmod_ret/hid_bpf_rdesc_fixup") program is attached (if no program was attached before), the kernel immediately disconnects the HID device and does a reprobe.

In the same way, when the SEC("fmod_ret/hid_bpf_rdesc_fixup") program is detached, the kernel issues a disconnect on the device.

There is no detach facility in HID-BPF. Detaching a program happens when all the user space file descriptors pointing at a program are closed. Thus, if we need to replace a report descriptor fixup, some cooperation is required from the owner of the original report descriptor fixup. The previous owner will likely pin the program in the bpffs, and we can then replace it through normal bpf operations.

6.6 Attaching a bpf program to a device

libbpf does not export any helper to attach a HID-BPF program. Users need to use a dedicated syscall program which will call hid_bpf_attach_prog(hid_id, program_fd, flags).

hid_id is the unique system ID of the HID device (the last 4 numbers in the sysfs path: /sys/bus/hid/devices/xxxx:yyyy:zzzz:0000)

progam_fd is the opened file descriptor of the program to attach.

flags is of type enum hid_bpf_attach_flags.

We can not rely on hidraw to bind a BPF program to a HID device. hidraw is an artefact of the processing of the HID device, and is not stable. Some drivers even disable it, so that removes the tracing capabilities on those devices (where it is interesting to get the non-hidraw traces).

On the other hand, the hid_id is stable for the entire life of the HID device, even if we change its report descriptor.

Given that hidraw is not stable when the device disconnects/reconnects, we recommend accessing the current report descriptor of the device through the sysfs. This is available at `/sys/bus/hid/devices/BUS:VID:PID.000N/report_descriptor` as a binary stream.

Parsing the report descriptor is the responsibility of the BPF programmer or the userspace component that loads the eBPF program.

6.7 An (almost) complete example of a BPF enhanced HID device

Foreword: for most parts, this could be implemented as a kernel driver

Let's imagine we have a new tablet device that has some haptic capabilities to simulate the surface the user is scratching on. This device would also have a specific 3 positions switch to toggle between *pencil on paper*, *cray on a wall* and *brush on a painting canvas*. To make things even better, we can control the physical position of the switch through a feature report.

And of course, the switch is relying on some userspace component to control the haptic feature of the device itself.

6.7.1 Filtering events

The first step consists in filtering events from the device. Given that the switch position is actually reported in the flow of the pen events, using hidraw to implement that filtering would mean that we wake up userspace for every single event.

This is OK for libinput, but having an external library that is just interested in one byte in the report is less than ideal.

For that, we can create a basic skeleton for our BPF program:

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

/* HID programs need to be GPL */
char _license[] SEC("license") = "GPL";

/* HID-BPF kfunc API definitions */
extern __u8 *hid_bpf_get_data(struct hid_bpf_ctx *ctx,
                              unsigned int offset,
                              const size_t __sz) __ksym;
extern int hid_bpf_attach_prog(unsigned int hid_id, int prog_fd, u32 flags) __
↪ksym;

struct {
      __uint(type, BPF_MAP_TYPE_RINGBUF);
      __uint(max_entries, 4096 * 64);
} ringbuf SEC(".maps");

struct attach_prog_args {
      int prog_fd;
      unsigned int hid;
      unsigned int flags;
      int retval;
};

SEC("syscall")
int attach_prog(struct attach_prog_args *ctx)
{
      ctx->retval = hid_bpf_attach_prog(ctx->hid,
                                        ctx->prog_fd,
                                        ctx->flags);
      return 0;
}

__u8 current_value = 0;

SEC("?fmod_ret/hid_bpf_device_event")
int BPF_PROG(filter_switch, struct hid_bpf_ctx *hid_ctx)
{
      __u8 *data = hid_bpf_get_data(hid_ctx, 0 /* offset */, 192 /* size */);
      __u8 *buf;

      if (!data)
              return 0; /* EPERM check */

      if (current_value != data[152]) {
              buf = bpf_ringbuf_reserve(&ringbuf, 1, 0);
              if (!buf)
                      return 0;

              *buf = data[152];

              bpf_ringbuf_commit(buf, 0);

              current_value = data[152];
      }

      return 0;
}
```

To attach `filter_switch`, userspace needs to call the `attach_prog` syscall program first:

```
static int attach_filter(struct hid *hid_skel, int hid_id)
{
      int err, prog_fd;
      int ret = -1;
      struct attach_prog_args args = {
              .hid = hid_id,
      };
      DECLARE_LIBBPF_OPTS(bpf_test_run_opts, tattrs,
              .ctx_in = &args,
              .ctx_size_in = sizeof(args),
      );

      args.prog_fd = bpf_program__fd(hid_skel->progs.filter_switch);

      prog_fd = bpf_program__fd(hid_skel->progs.attach_prog);

      err = bpf_prog_test_run_opts(prog_fd, &tattrs);
      if (err)
              return err;

      return args.retval; /* the fd of the created bpf_link */
}
```

Our userspace program can now listen to notifications on the ring buffer, and is awaken only when the value changes.

When the userspace program doesn't need to listen to events anymore, it can just close the returned fd from `attach_filter()`, which will tell the kernel to detach the program from the HID device.

Of course, in other use cases, the userspace program can also pin the fd to the BPF filesystem through a call to `bpf_obj_pin()`, as with any bpf_link.

6.7.2 Controlling the device

To be able to change the haptic feedback from the tablet, the userspace program needs to emit a feature report on the device itself.

Instead of using hidraw for that, we can create a `SEC("syscall")` program that talks to the device:

```
/* some more HID-BPF kfunc API definitions */
extern struct hid_bpf_ctx *hid_bpf_allocate_context(unsigned int hid_id) __
↪ksym;
extern void hid_bpf_release_context(struct hid_bpf_ctx *ctx) __ksym;
extern int hid_bpf_hw_request(struct hid_bpf_ctx *ctx,
                              __u8* data,
                              size_t len,
                              enum hid_report_type type,
                              enum hid_class_request reqtype) __ksym;


struct hid_send_haptics_args {
        /* data needs to come at offset 0 so we can do a memcpy into it */
        __u8 data[10];
        unsigned int hid;
};

SEC("syscall")
int send_haptic(struct hid_send_haptics_args *args)
{
        struct hid_bpf_ctx *ctx;
```

```
      int ret = 0;

      ctx = hid_bpf_allocate_context(args->hid);
      if (!ctx)
              return 0; /* EPERM check */

      ret = hid_bpf_hw_request(ctx,
                               args->data,
                               10,
                               HID_FEATURE_REPORT,
                               HID_REQ_SET_REPORT);

      hid_bpf_release_context(ctx);

      return ret;
}
```

And then userspace needs to call that program directly:

```
static int set_haptic(struct hid *hid_skel, int hid_id, __u8 haptic_value)
{
      int err, prog_fd;
      int ret = -1;
      struct hid_send_haptics_args args = {
              .hid = hid_id,
      };
      DECLARE_LIBBPF_OPTS(bpf_test_run_opts, tattrs,
              .ctx_in = &args,
              .ctx_size_in = sizeof(args),
      );

      args.data[0] = 0x02; /* report ID of the feature on our device */
      args.data[1] = haptic_value;

      prog_fd = bpf_program__fd(hid_skel->progs.set_haptic);

      err = bpf_prog_test_run_opts(prog_fd, &tattrs);
      return err;
}
```

Now our userspace program is aware of the haptic state and can control it. The program could make this state further available to other userspace programs (e.g. via a DBus API).

The interesting bit here is that we did not created a new kernel API for this. Which means that if there is a bug in our implementation, we can change the interface with the kernel at-will, because the userspace application is responsible for its own usage.

CHAPTER

SEVEN

UHID - USER-SPACE I/O DRIVER SUPPORT FOR HID SUBSYSTEM

UHID allows user-space to implement HID transport drivers. Please see *HID I/O Transport Drivers* 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

7.1 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.

7.2 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.

7.3 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>

CHAPTER

EIGHT

ALPS HID TOUCHPAD PROTOCOL

8.1 Introduction

Currently ALPS HID driver supports U1 Touchpad device.

U1 device basic information.

Vendor ID	0x044E
Product ID	0x120B
Version ID	0x0121

8.2 HID Descriptor

Byte	Field	Value	Notes
0	wHIDDescLength	001E	Length of HID Descriptor : 30 bytes
2	bcdVersion	0100	Compliant with Version 1.00
4	wReportDescLength	00B2	Report Descriptor is 178 Bytes (0x00B2)
6	wReportDescRegister	0002	Identifier to read Report Descriptor
8	wInputRegister	0003	Identifier to read Input Report
10	wMaxInputLength	0053	Input Report is 80 Bytes + 2
12	wOutputRegister	0000	Identifier to read Output Report
14	wMaxOutputLength	0000	No Output Reports
16	wCommandRegister	0005	Identifier for Command Register
18	wDataRegister	0006	Identifier for Data Register
20	wVendorID	044E	Vendor ID 0x044E
22	wProductID	120B	Product ID 0x120B
24	wVersionID	0121	Version 01.21
26	RESERVED	0000	RESERVED

8.3 Report ID

ReportID-1	(Input Reports)	(HIDUsage-Mouse) for TP&SP
ReportID-2	(Input Reports)	(HIDUsage-keyboard) for TP
ReportID-3	(Input Reports)	(Vendor Usage: Max 10 finger data) for TP
ReportID-4	(Input Reports)	(Vendor Usage: ON bit data) for GP
ReportID-5	(Feature Reports)	Feature Reports
ReportID-6	(Input Reports)	(Vendor Usage: StickPointer data) for SP
ReportID-7	(Feature Reports)	Flash update (Bootloader)

8.4 Data pattern

Case1	ReportID_1	TP/SP	Relative/Relative
Case2	ReportID_3 ReportID_6	TP SP	Absolute Absolute

8.5 Command Read/Write

To read/write to RAM, need to send a command to the device.

The command format is as below.

DataByte(SET_REPORT)

Byte1	Command Byte
Byte2	Address - Byte 0 (LSB)
Byte3	Address - Byte 1
Byte4	Address - Byte 2
Byte5	Address - Byte 3 (MSB)
Byte6	Value Byte
Byte7	Checksum

Command Byte is read=0xD1/write=0xD2.

Address is read/write RAM address.

Value Byte is writing data when you send the write commands.

When you read RAM, there is no meaning.

DataByte(GET_REPORT)

Byte1	Response Byte
Byte2	Address - Byte 0 (LSB)
Byte3	Address - Byte 1
Byte4	Address - Byte 2
Byte5	Address - Byte 3 (MSB)
Byte6	Value Byte
Byte7	Checksum

Read value is stored in Value Byte.

Packet Format Touchpad data byte -----------------

•	b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	SW6	SW5	SW4	SW3	SW2	SW1
2	0	0	0	Fcv	Fn3	Fn2	Fn1	Fn0
3	Xa0_7	Xa0_6	Xa0_5	Xa0_4	Xa0_3	Xa0_2	Xa0_1	Xa0_0
4	Xa0_15	Xa0_14	Xa0_13	Xa0_12	Xa0_11	Xa0_10	Xa0_9	Xa0_8
5	Ya0_7	Ya0_6	Ya0_5	Ya0_4	Ya0_3	Ya0_2	Ya0_1	Ya0_0
6	Ya0_15	Ya0_14	Ya0_13	Ya0_12	Ya0_11	Ya0_10	Ya0_9	Ya0_8
7	LFB0	Zs0_6	Zs0_5	Zs0_4	Zs0_3	Zs0_2	Zs0_1	Zs0_0
8	Xa1_7	Xa1_6	Xa1_5	Xa1_4	Xa1_3	Xa1_2	Xa1_1	Xa1_0
9	Xa1_15	Xa1_14	Xa1_13	Xa1_12	Xa1_11	Xa1_10	Xa1_9	Xa1_8
10	Ya1_7	Ya1_6	Ya1_5	Ya1_4	Ya1_3	Ya1_2	Ya1_1	Ya1_0
11	Ya1_15	Ya1_14	Ya1_13	Ya1_12	Ya1_11	Ya1_10	Ya1_9	Ya1_8
12	LFB1	Zs1_6	Zs1_5	Zs1_4	Zs1_3	Zs1_2	Zs1_1	Zs1_0
13	Xa2_7	Xa2_6	Xa2_5	Xa2_4	Xa2_3	Xa2_2	Xa2_1	Xa2_0
14	Xa2_15	Xa2_14	Xa2_13	Xa2_12	Xa2_11	Xa2_10	Xa2_9	Xa2_8
15	Ya2_7	Ya2_6	Ya2_5	Ya2_4	Ya2_3	Ya2_2	Ya2_1	Ya2_0
16	Ya2_15	Ya2_14	Ya2_13	Ya2_12	Ya2_11	Ya2_10	Ya2_9	Ya2_8
17	LFB2	Zs2_6	Zs2_5	Zs2_4	Zs2_3	Zs2_2	Zs2_1	Zs2_0
18	Xa3_7	Xa3_6	Xa3_5	Xa3_4	Xa3_3	Xa3_2	Xa3_1	Xa3_0
19	Xa3_15	Xa3_14	Xa3_13	Xa3_12	Xa3_11	Xa3_10	Xa3_9	Xa3_8
20	Ya3_7	Ya3_6	Ya3_5	Ya3_4	Ya3_3	Ya3_2	Ya3_1	Ya3_0
21	Ya3_15	Ya3_14	Ya3_13	Ya3_12	Ya3_11	Ya3_10	Ya3_9	Ya3_8
22	LFB3	Zs3_6	Zs3_5	Zs3_4	Zs3_3	Zs3_2	Zs3_1	Zs3_0
23	Xa4_7	Xa4_6	Xa4_5	Xa4_4	Xa4_3	Xa4_2	Xa4_1	Xa4_0
24	Xa4_15	Xa4_14	Xa4_13	Xa4_12	Xa4_11	Xa4_10	Xa4_9	Xa4_8
25	Ya4_7	Ya4_6	Ya4_5	Ya4_4	Ya4_3	Ya4_2	Ya4_1	Ya4_0
26	Ya4_15	Ya4_14	Ya4_13	Ya4_12	Ya4_11	Ya4_10	Ya4_9	Ya4_8
27	LFB4	Zs4_6	Zs4_5	Zs4_4	Zs4_3	Zs4_2	Zs4_1	Zs4_0

SW1-SW6:
SW ON/OFF status

Xan_15-0(16bit):
X Absolute data of the “n”th finger

Yan_15-0(16bit):
Y Absolute data of the “n”th finger

Zsn_6-0(7bit):
Operation area of the “n”th finger

8.6 StickPointer data byte

•	b7	b6	b5	b4	b3	b2	b1	b0
Byte1	1	1	1	0	1	SW3	SW2	SW1
Byte2	X7	X6	X5	X4	X3	X2	X1	X0
Byte3	X15	X14	X13	X12	X11	X10	X9	X8
Byte4	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
Byte5	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8
Byte6	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0
Byte7	T&P	Z14	Z13	Z12	Z11	Z10	Z9	Z8

SW1-SW3:
SW ON/OFF status

Xn_15-0(16bit):
X Absolute data

Yn_15-0(16bit):
Y Absolute data

Zn_14-0(15bit):
Z

CHAPTER

NINE

INTEL INTEGRATED SENSOR HUB (ISH)

A sensor hub enables the ability to offload sensor polling and algorithm processing to a dedicated low power co-processor. This allows the core processor to go into low power modes more often, resulting in increased battery life.

There are many vendors providing external sensor hubs conforming to HID Sensor usage tables. These may be found in tablets, 2-in-1 convertible laptops and embedded products. Linux has had this support since Linux 3.9.

Intel® introduced integrated sensor hubs as a part of the SoC starting from Cherry Trail and now supported on multiple generations of CPU packages. There are many commercial devices already shipped with Integrated Sensor Hubs (ISH). These ISH also comply to HID sensor specification, but the difference is the transport protocol used for communication. The current external sensor hubs mainly use HID over I2C or USB. But ISH doesn't use either I2C or USB.

9.1 1. Overview

Using a analogy with a usbhid implementation, the ISH follows a similar model for a very high speed communication:

```
-----------------               ----------------------
|    USB HID    |       -->     |    ISH HID         |
-----------------               ----------------------
-----------------               ----------------------
|  USB protocol |       -->     |    ISH Transport   |
-----------------               ----------------------
-----------------               ----------------------
|  EHCI/XHCI    |       -->     |    ISH IPC         |
-----------------               ----------------------
      PCI                                PCI
-----------------               ----------------------
|Host controller|       -->     |    ISH processor   |
-----------------               ----------------------
     USB Link
-----------------               ----------------------
| USB End points|       -->     |    ISH Clients     |
-----------------               ----------------------
```

Like USB protocol provides a method for device enumeration, link management and user data encapsulation, the ISH also provides similar services. But it is very light weight tailored to manage and communicate with ISH client applications implemented in the firmware.

The ISH allows multiple sensor management applications executing in the firmware. Like USB endpoints the messaging can be to/from a client. As part of enumeration process, these clients are identified. These clients can be simple HID sensor applications, sensor calibration applications or sensor firmware update applications.

The implementation model is similar, like USB bus, ISH transport is also implemented as a bus. Each client application executing in the ISH processor is registered as a device on this bus. The driver, which binds each device (ISH HID driver) identifies the device type and registers with the HID core.

9.2 2. ISH Implementation: Block Diagram

```
       ---------------------------
      |  User Space Applications  |
       ---------------------------

----------------IIO ABI----------------
       --------------------------
      |  IIO Sensor Drivers     |
       --------------------------
       --------------------------
      |        IIO core         |
       --------------------------
       --------------------------
      |   HID Sensor Hub MFD    |
       --------------------------
       --------------------------
      |       HID Core          |
       --------------------------
       --------------------------
      |   HID over ISH Client   |
       --------------------------
       --------------------------
      |   ISH Transport (ISHTP) |
       --------------------------
       --------------------------
      |      IPC Drivers        |
       --------------------------
OS
---------------- PCI -----------------
Hardware + Firmware
       ----------------------------
      | ISH Hardware/Firmware(FW) |
       ----------------------------
```

9.3 3. High level processing in above blocks

9.3.1 3.1 Hardware Interface

The ISH is exposed as "Non-VGA unclassified PCI device" to the host. The PCI product and vendor IDs are changed from different generations of processors. So the source code which enumerates drivers needs to update from generation to generation.

9.3.2 3.2 Inter Processor Communication (IPC) driver

Location: drivers/hid/intel-ish-hid/ipc

The IPC message uses memory mapped I/O. The registers are defined in hw-ish-regs.h.

3.2.1 IPC/FW message types

There are two types of messages, one for management of link and another for messages to and from transport layers.

TX and RX of Transport messages

A set of memory mapped register offers support of multi-byte messages TX and RX (e.g. IPC_REG_ISH2HOST_MSG, IPC_REG_HOST2ISH_MSG). The IPC layer maintains internal queues to sequence messages and send them in order to the firmware. Optionally the caller can register handler to get notification of completion. A doorbell mechanism is used in messaging to trigger processing in host and client firmware side. When ISH interrupt handler is called, the ISH2HOST doorbell register is used by host drivers to determine that the interrupt is for ISH.

Each side has 32 32-bit message registers and a 32-bit doorbell. Doorbell register has the following format:

```
Bits 0..6: fragment length (7 bits are used)
Bits 10..13: encapsulated protocol
Bits 16..19: management command (for IPC management protocol)
Bit 31: doorbell trigger (signal H/W interrupt to the other side)
Other bits are reserved, should be 0.
```

3.2.2 Transport layer interface

To abstract HW level IPC communication, a set of callbacks is registered. The transport layer uses them to send and receive messages. Refer to struct ishtp_hw_ops for callbacks.

9.3.3 3.3 ISH Transport layer

Location: drivers/hid/intel-ish-hid/ishtp/

3.3.1 A Generic Transport Layer

The transport layer is a bi-directional protocol, which defines: - Set of commands to start, stop, connect, disconnect and flow control (see ishtp/hbm.h for details) - A flow control mechanism to avoid buffer overflows

This protocol resembles bus messages described in the following document: http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/dcmi-hi-1-0-spec.pdf "Chapter 7: Bus Message Layer"

3.3.2 Connection and Flow Control Mechanism

Each FW client and a protocol is identified by a UUID. In order to communicate to a FW client, a connection must be established using connect request and response bus messages. If successful, a pair (host_client_id and fw_client_id) will identify the connection.

Once connection is established, peers send each other flow control bus messages independently. Every peer may send a message only if it has received a flow-control credit before. Once it has sent a message, it may not send another one before receiving the next flow control credit. Either side can send disconnect request bus message to end communication. Also the link will be dropped if major FW reset occurs.

3.3.3 Peer to Peer data transfer

Peer to Peer data transfer can happen with or without using DMA. Depending on the sensor bandwidth requirement DMA can be enabled by using module parameter ishtp_use_dma under intel_ishtp.

Each side (host and FW) manages its DMA transfer memory independently. When an ISHTP client from either host or FW side wants to send something, it decides whether to send over IPC or over DMA; for each transfer the decision is independent. The sending side sends DMA_XFER message when the message is in the respective host buffer (TX when host client sends, RX when FW client sends). The recipient of DMA message responds with DMA_XFER_ACK, indicating the sender that the memory region for that message may be reused.

DMA initialization is started with host sending DMA_ALLOC_NOTIFY bus message (that includes RX buffer) and FW responds with DMA_ALLOC_NOTIFY_ACK. Additionally to DMA address communication, this sequence checks capabilities: if the host doesn't support DMA, then it won't send DMA allocation, so FW can't send DMA; if FW doesn't support DMA then it won't respond with DMA_ALLOC_NOTIFY_ACK, in which case host will not use DMA transfers. Here ISH acts as busmaster DMA controller. Hence when host sends DMA_XFER, it's request to do host->ISH DMA transfer; when FW sends DMA_XFER, it means that it already did DMA and the message resides at host. Thus, DMA_XFER and DMA_XFER_ACK act as ownership indicators.

At initial state all outgoing memory belongs to the sender (TX to host, RX to FW), DMA_XFER transfers ownership on the region that contains ISHTP message to the receiving side, DMA_XFER_ACK returns ownership to the sender. A sender need not wait for previous DMA_XFER to be ack'ed, and may send another message as long as remaining continuous memory in its ownership is enough. In principle, multiple DMA_XFER and DMA_XFER_ACK messages may be sent at once (up to IPC MTU), thus allowing for interrupt throttling. Currently, ISH FW decides to send over DMA if ISHTP message is more than 3 IPC fragments and via IPC otherwise.

3.3.4 Ring Buffers

When a client initiates a connection, a ring of RX and TX buffers is allocated. The size of ring can be specified by the client. HID client sets 16 and 32 for TX and RX buffers respectively. On send request from client, the data to be sent is copied to one of the send ring buffer and scheduled to be sent using bus message protocol. These buffers are required because the FW may have not have processed the last message and may not have enough flow control credits to send. Same thing holds true on receive side and flow control is required.

3.3.5 Host Enumeration

The host enumeration bus command allows discovery of clients present in the FW. There can be multiple sensor clients and clients for calibration function.

To ease implementation and allow independent drivers to handle each client, this transport layer takes advantage of Linux Bus driver model. Each client is registered as device on the transport bus (ishtp bus).

Enumeration sequence of messages:

- Host sends HOST_START_REQ_CMD, indicating that host ISHTP layer is up.
- FW responds with HOST_START_RES_CMD
- Host sends HOST_ENUM_REQ_CMD (enumerate FW clients)
- FW responds with HOST_ENUM_RES_CMD that includes bitmap of available FW client IDs
- For each FW ID found in that bitmap host sends HOST_CLIENT_PROPERTIES_REQ_CMD
- FW responds with HOST_CLIENT_PROPERTIES_RES_CMD. Properties include UUID, max ISHTP message size, etc.
- Once host received properties for that last discovered client, it considers ISHTP device fully functional (and allocates DMA buffers)

9.3.4 3.4 HID over ISH Client

Location: drivers/hid/intel-ish-hid

The ISHTP client driver is responsible for:

- enumerate HID devices under FW ISH client
- Get Report descriptor
- Register with HID core as a LL driver
- Process Get/Set feature request
- Get input reports

9.3.5 3.5 HID Sensor Hub MFD and IIO sensor drivers

The functionality in these drivers is the same as an external sensor hub. Refer to *HID Sensors Framework* for HID sensor Documentation/ABI/testing/sysfs-bus-iio for IIO ABIs to user space.

9.3.6 3.6 End to End HID transport Sequence Diagram

```
HID-ISH-CLN                    ISHTP                    IPC                   ␣
 ↪         HW
        |                        |                       |                    ␣
 ↪          |
        |                        |                       |-----WAKE UP---------
 ↪---------->|
        |                        |                       |                    ␣
 ↪          |
        |                        |                       |-----HOST READY------
 ↪---------->|
        |                        |                       |                    ␣
 ↪          |
        |                        |                       |<----MNG_RESET_
 ↪NOTIFY_ACK----- |
        |                        |                       |                    ␣
 ↪          |
        |                        |<----ISHTP_START------ |                    ␣
 ↪          |
        |                        |                       |                    ␣
 ↪          |
        |                        |<-----------------HOST_START_RES_CMD---------
 ↪----------|
        |                        |                       |                    ␣
 ↪          |
        |                        |------------------QUERY_SUBSCRIBER-----------
 ↪---------->|
        |                        |                       |                    ␣
 ↪          |
        |                        |------------------HOST_ENUM_REQ_CMD----------
 ↪---------->|
        |                        |                       |                    ␣
 ↪          |
        |                        |<-----------------HOST_ENUM_RES_CMD----------
 ↪----------|
        |                        |                       |                    ␣
 ↪          |
        |                        |------------------HOST_CLIENT_PROPERTIES_REQ_
 ↪CMD------>|
        |                        |                       |                    ␣
 ↪          |
        |                        |<-----------------HOST_CLIENT_PROPERTIES_RES_
 ↪CMD-------|
        |      Create new device on in ishtp bus        |                    ␣
```

```
↪           |
        |                        |                       |                    ␣
↪           |
        |                        |------------------HOST_CLIENT_PROPERTIES_REQ_
↪CMD------>|
        |                        |                       |                    ␣
↪           |
        |                        |<-----------------HOST_CLIENT_PROPERTIES_RES_
↪CMD-------|
        |       Create new device on in ishtp bus        |                    ␣
↪           |
        |                        |                       |                    ␣
↪           |
        |                        |--Repeat HOST_CLIENT_PROPERTIES_REQ_CMD-till␣
↪last one--|
        |                        |                       |                    ␣
↪           |
     probed()
        |----ishtp_cl_connect--->|----------------- CLIENT_CONNECT_REQ_CMD-----
↪---------->|
        |                        |                       |                    ␣
↪           |
        |                        |<----------------CLIENT_CONNECT_RES_CMD------
↪----------|
        |                        |                       |                    ␣
↪           |
        |register event callback |                       |                    ␣
↪           |
        |                        |                       |                    ␣
↪           |
        |ishtp_cl_send(
        HOSTIF_DM_ENUM_DEVICES)  |----------fill ishtp_msg_hdr struct write to␣
↪HW-----  >|
        |                        |                       |                    ␣
↪           |
        |                        |                       |<-----IRQ(IPC_
↪PROTOCOL_ISHTP---|
        |                        |                       |                    ␣
↪           |
        |<--ENUM_DEVICE RSP------|                       |                    ␣
↪           |
        |                        |                       |                    ␣
↪           |
for each enumerated device
        |ishtp_cl_send(
        HOSTIF_GET_HID_DESCRIPTOR|----------fill ishtp_msg_hdr struct write to␣
↪HW-----  >|
        |                        |                       |                    ␣
↪           |
        ...Response
        |                        |                       |                    ␣
 ↪          |
for each enumerated device
        |ishtp_cl_send(
     HOSTIF_GET_REPORT_DESCRIPTOR|--------------fill ishtp_msg_hdr struct␣
 ↪write to HW-- >|
        |                        |                       |                    ␣
 ↪          |
        |                        |                       |                    ␣
 ↪          |
 hid_allocate_device
        |                        |                       |                    ␣
 ↪          |
 hid_add_device                  |                       |                    ␣
 ↪          |
        |                        |                       |                    ␣
 ↪          |
```

9.3.7 3.7 ISH Debugging

To debug ISH, event tracing mechanism is used. To enable debug logs:

```
echo 1 > /sys/kernel/tracing/events/intel_ish/enable
cat /sys/kernel/tracing/trace
```

9.3.8 3.8 ISH IIO sysfs Example on Lenovo thinkpad Yoga 260

```
root@otcpl-ThinkPad-Yoga-260:~# tree -l /sys/bus/iio/devices/
/sys/bus/iio/devices/
├── iio:device0 -> ../../../devices/0044:8086:22D8.0001/HID-SENSOR-200073.9.
 ↪auto/iio:device0
│   ├── buffer
│   │   ├── enable
│   │   ├── length
│   │   └── watermark
...
│   ├── in_accel_hysteresis
│   ├── in_accel_offset
│   ├── in_accel_sampling_frequency
│   ├── in_accel_scale
│   ├── in_accel_x_raw
│   ├── in_accel_y_raw
│   ├── in_accel_z_raw
│   ├── name
│   ├── scan_elements
│   │   ├── in_accel_x_en
│   │   ├── in_accel_x_index
│   │   ├── in_accel_x_type
│   │   ├── in_accel_y_en
│   │   ├── in_accel_y_index
│   │   ├── in_accel_y_type
│   │   ├── in_accel_z_en
│   │   ├── in_accel_z_index
│   │   └── in_accel_z_type
...
│   │   ├── devices
│   │   │   │   ├── buffer
│   │   │   │   │   ├── enable
│   │   │   │   │   ├── length
│   │   │   │   │   └── watermark
│   │   │   │   ├── dev
│   │   │   │   ├── in_intensity_both_raw
│   │   │   │   ├── in_intensity_hysteresis
│   │   │   │   ├── in_intensity_offset
│   │   │   │   ├── in_intensity_sampling_frequency
│   │   │   │   ├── in_intensity_scale
│   │   │   │   ├── name
│   │   │   │   ├── scan_elements
│   │   │   │   │   ├── in_intensity_both_en
│   │   │   │   │   ├── in_intensity_both_index
│   │   │   │   │   └── in_intensity_both_type
│   │   │   │   ├── trigger
│   │   │   │   │   └── current_trigger
...
│   │   │   │   ├── buffer
│   │   │   │   │   ├── enable
│   │   │   │   │   ├── length
│   │   │   │   │   └── watermark
│   │   │   │   ├── dev
│   │   │   │   ├── in_magn_hysteresis
│   │   │   │   ├── in_magn_offset
│   │   │   │   ├── in_magn_sampling_frequency
│   │   │   │   ├── in_magn_scale
│   │   │   │   ├── in_magn_x_raw
│   │   │   │   ├── in_magn_y_raw
│   │   │   │   ├── in_magn_z_raw
│   │   │   │   ├── in_rot_from_north_magnetic_tilt_comp_raw
│   │   │   │   ├── in_rot_hysteresis
│   │   │   │   ├── in_rot_offset
│   │   │   │   ├── in_rot_sampling_frequency
│   │   │   │   ├── in_rot_scale
│   │   │   │   ├── name
...
│   │   │   │   ├── scan_elements
│   │   │   │   │   ├── in_magn_x_en
│   │   │   │   │   ├── in_magn_x_index
│   │   │   │   │   ├── in_magn_x_type
│   │   │   │   │   ├── in_magn_y_en
│   │   │   │   │   ├── in_magn_y_index
│   │   │   │   │   ├── in_magn_y_type
│   │   │   │   │   ├── in_magn_z_en
│   │   │   │   │   ├── in_magn_z_index
│   │   │   │   │   ├── in_magn_z_type
│   │   │   │   │   ├── in_rot_from_north_magnetic_tilt_comp_en
│   │   │   │   │   ├── in_rot_from_north_magnetic_tilt_comp_index
│   │   │   │   │   └── in_rot_from_north_magnetic_tilt_comp_type
│   │   │   │   ├── trigger
│   │   │   │   │   └── current_trigger
...
│   │   │   │   ├── buffer
│   │   │   │   │   ├── enable
│   │   │   │   │   ├── length
│   │   │   │   │   └── watermark
│   │   │   │   ├── dev
│   │   │   │   ├── in_anglvel_hysteresis
│   │   │   │   ├── in_anglvel_offset
│   │   │   │   ├── in_anglvel_sampling_frequency
│   │   │   │   ├── in_anglvel_scale
│   │   │   │   ├── in_anglvel_x_raw
│   │   │   │   ├── in_anglvel_y_raw
│   │   │   │   ├── in_anglvel_z_raw
│   │   │   │   ├── name
│   │   │   │   ├── scan_elements
│   │   │   │   │   ├── in_anglvel_x_en
│   │   │   │   │   ├── in_anglvel_x_index
│   │   │   │   │   ├── in_anglvel_x_type
│   │   │   │   │   ├── in_anglvel_y_en
│   │   │   │   │   ├── in_anglvel_y_index
│   │   │   │   │   ├── in_anglvel_y_type
│   │   │   │   │   ├── in_anglvel_z_en
│   │   │   │   │   ├── in_anglvel_z_index
│   │   │   │   │   └── in_anglvel_z_type
│   │   │   │   ├── trigger
│   │   │   │   │   └── current_trigger
...
│   │   │   │   ├── buffer
│   │   │   │   │   ├── enable
│   │   │   │   │   ├── length
│   │   │   │   │   └── watermark
│   │   │   │   ├── dev
│   │   │   │   ├── in_anglvel_hysteresis
│   │   │   │   ├── in_anglvel_offset
│   │   │   │   ├── in_anglvel_sampling_frequency
│   │   │   │   ├── in_anglvel_scale
│   │   │   │   ├── in_anglvel_x_raw
│   │   │   │   ├── in_anglvel_y_raw
│   │   │   │   ├── in_anglvel_z_raw
│   │   │   │   ├── name
│   │   │   │   ├── scan_elements
│   │   │   │   │   ├── in_anglvel_x_en
│   │   │   │   │   ├── in_anglvel_x_index
│   │   │   │   │   ├── in_anglvel_x_type
│   │   │   │   │   ├── in_anglvel_y_en
│   │   │   │   │   ├── in_anglvel_y_index
│   │   │   │   │   ├── in_anglvel_y_type
│   │   │   │   │   ├── in_anglvel_z_en
│   │   │   │   │   ├── in_anglvel_z_index
│   │   │   │   │   └── in_anglvel_z_type
│   │   │   │   ├── trigger
│   │   │   │   │   └── current_trigger
...
```

AMD SENSOR FUSION HUB

AMD Sensor Fusion Hub (SFH) is part of an SOC starting from Ryzen-based platforms. The solution is working well on several OEM products. AMD SFH uses HID over PCIe bus. In terms of architecture it resembles ISH, however the major difference is all the HID reports are generated as part of the kernel driver.

10.1 Block Diagram

```
    ---------------------------------
    |  HID User Space Applications  |
    - -------------------------------

---------------------------------------------
     ---------------------------------
    |               HID Core          |
     ---------------------------------

     ---------------------------------
    |     AMD HID Transport           |
     ---------------------------------

     --------------------------------
    |             AMD HID Client     |
    |       with HID Report Generator|
     --------------------------------

     --------------------------------
    |     AMD MP2 PCIe Driver        |
     --------------------------------
OS
---------------------------------------------
Hardware + Firmware
     --------------------------------
     |     SFH MP2 Processor         |
     --------------------------------
```

10.2 AMD HID Transport Layer

AMD SFH transport is also implemented as a bus. Each client application executing in the AMD MP2 is registered as a device on this bus. Here, MP2 is an ARM core connected to x86 for processing sensor data. The layer, which binds each device (AMD SFH HID driver) identifies the device type and registers with the HID core. Transport layer attaches 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. AMD HID Transport layer implements the synchronous calls.

10.3 AMD HID Client Layer

This layer is responsible to implement HID requests and descriptors. As firmware is OS agnostic, HID client layer fills the HID request structure and descriptors. HID client layer is complex as it is interface between MP2 PCIe layer and HID. HID client layer initializes the MP2 PCIe layer and holds the instance of MP2 layer. It identifies the number of sensors connected using MP2-PCIe layer. Based on that allocates the DRAM address for each and every sensor and passes it to MP2-PCIe driver. On enumeration of each sensor, client layer fills the HID Descriptor structure and HID input report structure. HID Feature report structure is optional. The report descriptor structure varies from sensor to sensor.

10.4 AMD MP2 PCIe layer

MP2 PCIe Layer is responsible for making all transactions with the firmware over PCIe. The connection establishment between firmware and PCIe happens here.

The communication between X86 and MP2 is split into three parts. 1. Command transfer via the C2P mailbox registers. 2. Data transfer via DRAM. 3. Supported sensor info via P2C registers.

Commands are sent to MP2 using C2P Mailbox registers. Writing into C2P Message registers generates interrupt to MP2. The client layer allocates the physical memory and the same is sent to MP2 via the PCI layer. MP2 firmware writes the command output to the access DRAM memory which the client layer has allocated. Firmware always writes minimum of 32 bytes into DRAM. So as a protocol driver shall allocate minimum of 32 bytes DRAM space.

10.5 Enumeration and Probing flow

```
HID              AMD            AMD                        AMD -PCIe                ␣
↪MP2
Core         Transport      Client layer                     layer                  ␣
↪ FW
 |                |              |                            |                  |
 |                |              |                 on Boot Driver Loaded         |
 |                |              |                            |                  |
 |                |              |                        MP2-PCIe Int           |
 |                |              |                            |                  |
 |                |              |---Get Number of sensors-> |                  |
 |               |              |                       Read P2C              |
 |               |              |                        Register             |
 |               |              |                           |                 |
 |               |              | Loop(for No of Sensors)   |                 |
 |               |              |----------------------|    |                 |
 |               |              | Create HID Descriptor|    |                 |
 |               |              | Create Input  report |    |                 |
 |               |              |  Descriptor Map      |    |                 |
 |               |              |  the MP2 FW Index to |    |                 |
 |               |              |   HID Index          |    |                 |
 |               |              | Allocate the DRAM    |  Enable              |
 |               |              |       address        |  Sensors             |
 |               |              |----------------------|    |                 |
 |               | HID transport|                           |    Enable       |
 |               |<--Probe------|                           |---Sensor CMD--> |
 |               | Create the   |                           |                 |
 |               | HID device   |                           |                 |
 |               |    (MFD)     |                           |                 |
 |               | by Populating|                           |                 |
 |               |  the HID     |                           |                 |
 |               |  ll_driver   |                           |                 |
 | HID           |              |                           |                 |
 |  add          |              |                           |                 |
 |Device         |              |                           |                 |
 |<------------- |              |                           |                 |
```

10.6 Data Flow from Application to the AMD SFH Driver

```
           |              |              |                           |       ␣
↪         |
           |              |              |                           |       ␣
↪         |
           |              |              |                           |       ␣
↪         |
           |              |              |                           |       ␣
↪         |
           |              |              |                           |       ␣
↪         |
           |HID_req       |              |                           |       ␣
↪         |
           |get_report    |              |                           |       ␣
↪         |
           |------------->|              |                           |       ␣
↪         |
           |              | HID_get_input|                           |       ␣
↪         |
           |              |  report      |                           |       ␣
↪         |
            |              |------------->|----------------------|  |       ␣
 ↪         |
            |              |              | Read the DRAM data for|  |       ␣
 ↪         |
            |              |              | requested sensor and |  |       ␣
 ↪         |
            |              |              | create the HID input |  |       ␣
 ↪         |
            |              |              | report               |  |       ␣
 ↪         |
            |              |              |----------------------|  |       ␣
 ↪         |
            |              |Data received |                         |       ␣
 ↪         |
            |              | in HID report|                         |       ␣
 ↪         |
To          |<-------------|<-------------|                         |       ␣
 ↪         |
Applications|              |              |                         |       ␣
 ↪         |
    <-------|              |              |                         |       ␣
 ↪         |
```

INDEX