HID-over-GATT

HOGP

To be recognised as an HID, a device must implement the HID-over-GATT Profile (https://developer.bluetooth.org/TechnologyOverview/Pages/HOGP.aspx), which means at least the following services:

- HID.
- Battery.
- Device information.

BLE HID Service

All of the structure formats described in HID are used in HID-over-GATT.

The nomenclature is not ideal, though:

- Report Map: what the USB HID calls Report Descriptor.
- Report Reference Characteristic Descriptor is the BLE way of setting a report characteristic's metadata. It contains the type (Input/Output/Feature) and ID of a report.

The HID Service defines the following characteristics:

- *Protocol Mode*: the default is Report mode, but you can change that to Boot mode.
- *Report Map*: the HID Report descriptor, defining the possible format for Input/Output/Feature reports.
- *Report*: a characteristic used as a vehicle for HID reports. Unlike USB, where the ID is sent as a prefix when there is more than one report per type, the ID is stored in a characteristic descriptor. This means that there will be one characteristic per report described in the Report Map.
- *Boot Keyboard Input Report*: when the device is a keyboard, it must define boot reports.
- *Boot Keyboard Output Report*.
- *Boot Mouse Input Report*.
- *HID Information*: HID version, localization and some capability flags.
- *HID Control Point*: inform the device that the host is entering or leaving suspend state.

Instead of USB interrupt pipes, input reports are sent using notifications. They can also be read by the host.

Implementation with BLE API

Implementation with BLE_API

A custom HID device will need to inherit from HIDServiceBase and provide it with the necessary informations:

- A report map (USB's report descriptor). In the following example, we will use the keyboard map described in appendix B.1 of the USB HID specification.
- The report arrays and their sizes.
- The report rate, when reports need to be sent asynchronously. To start with a simple example, we'll only use synchronous reports.

We want to send the string "Hello" to an OS. We can first write a method `putc` that takes a char as argument and returns a status code. This function must send two reports: one that means "key down" and one that means "key up".

Given a `keymap` array that associates characters to their keycode, we can write `putc` as follows:

```
int KeyboardService::putc(char c)
{
    static uint8_t report[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    int err = 0;

    report[0] = keymap[c].modifier;
    report[2] = keymap[c].usage;

    err = send(report);
    if (err)
        return err;

    report[0] = 0;
    report[2] = 0;

    return send(report);
}
```

And a string could be sent with this:

```
void KeyboardService::puts(const char *s)
{
    for (; *s; s++)
        putc(*s);
}
```

There are two major issues with this code:

1. If the "key up" report fails, the key will seem to be stuck until the next report, from the OS' point of view.
2. Since we're using a very limited amount of resources, reports will soon fail if we attempt to send them sequentially.

To illustrate those issues, let's take as example a nRF51 chip with a S110 SoftDevice. The firmware is using seven buffers to store outgoing notifications. For a call to `keyboardService.puts("Hello")`, `putc` will be called for each character until the end of string, before returning and allowing the main loop to inspect

BLE events. Calls to `send()` will start to fail during the fourth report, since the seven notification buffers will be in use. Instead of seeing "Hello" on the OS, we'll see "Helllllllll...", which won't look good in a demo.

KeyboardService (classKeyboardService.html)

KeyboardService (classKeyboardService.html) uses a buffer to dissociate calls to `putc` from the `send` thread. This thread is provided by `HIDServiceBase` in the form of a ticker. When enabled, it calls the `sendCallback` method at a rate specified with the `reportTickerDelay` parameter.

We send key reports with KeyboardService (classKeyboardService.html) through its putc or printf methods. For example, with `kbdService.printf("Hello world!")`, the string "Hello world!" will go in a circular buffer, and the ticker will consume from this buffer every 20ms. First, the letter 'H' will be sent by writing the following values into the inputReport characteristic:

```
[0x2, 0, 0x0b, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

If the first report fails, the callback puts it back in the circular buffer and will try again on next call. Same goes for the empty report.

On the next tick, we'll send two reports for the letter 'e', and so on.

MouseService (classMouseService.html)

A mouse will need to send reports at regular interval, because the OS will only move the cursor upon receiving an input report. As explained in HID, our report will contain three bytes; one bitmap contains the button status, the next two are signed and represent the immediate speed.

Note that since we're using GATT notifications, there is no way to know if the OS got the message and understood it correctly.

Support in common operating systems

Bluetooth Low Energy support is still at an early stage, and the HID service is even less supported. At this stage, we can only say that examples work relatively well on Linux and Android.

Windows

Untested.

Mac OS

- The keyboard and mouse examples work on MacOSX 10.10. The OS takes complete control over HID Services, so you can connect to a device using the bluetooth system panel.
- All tests failed on MacOSX 10.9; it seems to simply ignore all input reports.
    - Mouse report map has been tested with USB HID and worked well. So we can assume that's not the issue.
    - Examples often manage to connect, but sending HID reports doesn't have any apparent effect. One way of "solving" this on MacOSX 10.10 was to for security on Device Information Service. Security requirements need to be investigated.

iOS

Untested.

Android

Android may be running either Bluez (see Linux), or the custom BlueDroid implementation. Either way, all initial tests succeeded: HID keyboard and mouse over BLE worked great. You should be able to just connect to HID devices from the bluetooth configuration panel.

Linux

Bluez (http://www.bluez.org/download/) is the way to go on Linux. Bluetooth support up until version 4.2 is in the Kernel, and userspace implementation depends on the distribution.

On Archlinux, for instance, systemd launches the Bluetooth daemon, which communicates with the kernel driver. Any client can then send commands using dbus. One such client is bluetoothctl, another is blueman.

I used the Bluetooth page on the Archlinux wiki (https://wiki.archlinux.org/index.php/Bluetooth) as a starting point. It's a good source of info about Bluez userspace.

The HID part is a bit more intricate:

- Bluetooth packets go through Bluez and Bluetooth daemon. The driver itself doesn't handle GATT.
- The daemon recognizes GATT packets associated with the HID Service.
- HID reports are sent to UHID (Userspace HID), and re-routed through the generic HID component in the kernel.
- Everything goes through the input drivers and out to the userspace again.

HID Subsystem with BLE in Linux
(Most of this diagram is courtesy of the kernel's Documentation/hid/hid-transport.txt)

I'm still having issues when a device is disconnected: the daemon seems unable to register the kernel hidraw device again on its own. My current solution is to remove the pairing infos manually with bluetoothctl's `remove` command and let the agent do the rest.

ARM mbed Enabled (https://www.mbed.com/en/about-mbed/mbed-enabled/)

Jobs (https://www.mbed.com/en/about-mbed/jobs/)

Events (https://www.mbed.com/en/about-mbed/events/)

Blog (https://www.mbed.com/en/about-mbed/blog/)

Technologies (https://www.mbed.com/en/technologies/)

Connectivity (https://www.mbed.com/en/technologies/connectivity/)

Security (https://www.mbed.com/en/technologies/security/)

Partners (https://www.mbed.com/en/partners/)

Our Partners (https://www.mbed.com/en/partners/our-partners/)

Become a Partner (https://www.mbed.com/en/partners/become-partner/)

Development (https://www.mbed.com/en/development/)

Hardware (https://www.mbed.com/en/development/hardware/)

Software (https://www.mbed.com/en/development/software/)

Cloud (https://www.mbed.com/en/development/cloud/)

Getting Started (https://www.mbed.com/en/development/getting-started/)

Community and Help (https://www.mbed.com/en/development/community-help/)

mbed Classic Developer site(https://www.mbed.com/en/development/developer-classic-site/)

Docs Home (/) | Terms (https://www.mbed.com/en/about-mbed/terms-use/) | Privacy (https://www.mbed.com/en/about-mbed/privacy/) | Cookies (https://www.mbed.com/en/about-mbed/cookie-policy/)