Porting a Kernel Space Linux USB Driver to Android User Space

USB OTG is an intriguing feature of Android that perhaps hasn't gotten as much attention as it deserves. I suspect this may be in part due to developers not knowing where to start. When I partially ported a USB Wi-Fi driver back in 2012, there wasn't much in the way of guides online—I assembled the puzzle one disparate piece of information at a time. My hope is that this guide will bring all the relevant information together and ignite some more interest in the area.

First, some background. Android 3.1 introduced support for USB On-The-Go (OTG). This allows an Android device to act as the USB host (much like a desktop computer) rather than as a USB peripheral (e.g. USB storage drive). Essentially, this makes it possible to plug *other* USB peripherals into an Android device. This is interesting, as it means all manner of existing USB peripherals can be made to interface with Android devices. Some peripherals work out of the box. For example, many USB mice can be connected directly to an Android device and a cursor will appear on the screen. A lot of other peripherals however do not have native driver support built into the Android operating system.

The way to get peripherals to work with Android when there is no native driver is to use Android's USB Host API to reimplement the peripheral's driver in user space. The USB host API lets applications register to be notified when a specific peripheral is connected to the Android device. This essentially becomes the entry point into your custom user space driver. From here, you can perform arbitrary byte-level I/O with the peripheral. From the Android operating system's point of view, it has no idea what those bytes mean, but this obliviousness is what affords the freedom to implement a user space driver. Because the USB host API is an officially supported application API, there's no need to root the Android device in order to interface with peripherals. Moreover, it means user space drivers can be released on the Google Play Store.

Before we get too far into things, a word of a warning—USB OTG is still a bit flaky in the Android ecosystem. Some devices (e.g. Galaxy Nexus) support USB OTG beautifully. Some other devices mostly support it, but will require an external power source for power hungry peripherals. And some other handsets just don't properly support it at all. For handsets with no support, this is sometimes a hardware issue and sometimes a software issue. Some people have had luck with rooting certain handsets and patching the operating system drivers to add USB OTG support, but this isn't ideal. Other devices flat out won't work at the hardware level. So, even though Android >= 3.1 is the formal prerequisite, it can still be a bit of a mine field. Hopefully USB OTG matures on Android over time and by increasing awareness of the issues, perhaps this will help expedite that process.

One more thing—Linux drivers are typically written in C while the Android USB Host API is written in Java. You'll need to be at least familiar with both languages.

What follows in the rest of this article is my best recollection of the steps I took when writing my own first user space driver. If you find I've missed anything or have gotten something incorrect, let me know and I'll fix the article.

Finding a Peripheral's Existing Driver Source Code

So you have a USB peripheral that works on regular Linux and you'd like to port the driver to Android. To find the Linux source code that's handling the peripheral, you'll first need to gather some of the peripheral's metadata. Per the USB protocol, every USB peripheral reports a few key pieces of information to the Linux host upon connection. The host then uses this information to delegate responsibility for the peripheral to a specific driver. The first two pieces of information you'll need are:

1. **Vendor ID**: a 16-bit string which uniquely identifies the peripheral's vendor.
2. **Product ID**: a 16-bit string which uniquely identifiers the specific peripheral.

On Linux, you can retrieve the Vendor ID and Product ID with the `lsusb` command. Here's an example of the output on my machine:

```
~$ lsusb
Bus 002 Device 011: ID 0bda:8187 Realtek Semiconductor Corp. RTL8187 Wireless
Adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:8008 Intel Corp.
Bus 002 Device 002: ID 8087:8000 Intel Corp.
Bus 002 Device 003: ID 174c:2074 ASMedia Technology Inc.
Bus 002 Device 004: ID 0b05:17d0 ASUSTek Computer, Inc.
Bus 002 Device 006: ID 046d:0826 Logitech, Inc.
Bus 002 Device 007: ID 045e:071d Microsoft Corp.
Bus 002 Device 008: ID 046d:c531 Logitech, Inc.
```

In the output above, the Vendor ID and Product ID on each line are the colon-separated hex string. For example, the USB wireless card I had connected to my computer at the time has Vendor ID `0bda` and Product ID `8187`. If you can't identify the specific peripheral you're interested in porting, unplug it and run `lsusb` again to see which item is now omitted in the output. Keep a note of the Vendor ID and Product ID. For the time being they'll help us to find the peripheral's Linux driver source code. You'll also need them later on though as a way of registering your Android application to handle connected peripherals with that Vendor ID and Product ID.

[WARNING: this paragraph is vague—if this is an area you have expertise in, please email me a better description of this step so I can include it]. With the Vendor ID and Product ID in hand, you can now establish which C source file is the driver entry point for your peripheral on Linux. To do this, follow the steps outlined in this Stack Overflow answer. Note my comment on this answer hasn't received a response. Again, if anyone would like to provide clarification on this step, I'll look to add it to this article. The rest of the article assumes you've been able to identify the "main" source file for your peripheral's driver.

Navigating the Peripheral's Linux Driver Source Code

You've located your peripheral driver's main source file—let's call it `foo.c`. When your peripheral is connected, Linux delegates responsibility for the peripheral to the code in this file. The obvious next question is, *where in this file does the driver begin executing?* What you should look for in `foo.c` is a call to `module_usb_driver()`. `module_usb_drver()` accepts a struct of type `usb_driver`, so you should see in the call that something is being passed in. This struct

has a number of fields, but the one of immediate interest is `.probe`. `.probe` is a function pointer to the method that initially gets called. In terms of controlling the peripheral, this is where things kick off. To give an example, at time of writing, here is what can be found in `/drivers/net/wireless/rtl818x/rtl8187/dev.c`:

```
static struct usb_driver rtl8187_driver = {
        .name           = KBUILD_MODNAME,
        .id_table       = rtl8187_table,
        .probe          = rtl8187_probe,
        .disconnect     = rtl8187_disconnect,
        .disable_hub_initiated_lpm = 1,
};

module_usb_driver(rtl8187_driver);
```

Take note of the line `.probe = rtl8187_probe,`. Sure enough, there's a function named `rtl8187_probe` in the same file that defines the code to be executed when this specific driver is invoked by the Linux operating system. It's at this point that the driver logic becomes specific to the attached peripheral. In other words, what you find in the probe function will be determined by what work the driver needs to do when your specific peripheral is first connected. In the remainder of this section I simply identify methods I found useful for navigating the driver source code.

A tool I found particularly useful for analysing the source code of my USB wireless LAN adapter was CTAGS. Not to drift too far off topic, but for the uninitiated, CTAGS is a program that can accept a source code directory as input and product an index of the source code as output. This index includes information such as the file and line number of function calls. The index itself is designed to be computer-readable, not human readable. Text editors like Emacs and vi use the index to allow navigating the codebase through keyboard shortcuts. For me, the indispensable feature of CTAGS was being able to recursively jump into and out of function definitions. Essentially I used CTAGS to generate an index over the entire Linux source code, so I was then able to perform this navigation through Emacs and vi keyboard shortcuts. Setting up CTAGS and choice of text editor however are beyond the scope of this article.

The reason I heavily advocate CTAGS (or similar) for reading USB driver code is that there are often many function calls separating a top-level function in the main driver source file from where the actual byte-level data is transmitted or received. When porting the driver to Android user space, it's ultimately this byte-level communication you'll need to emulate, so being able to quickly navigate in and out of function calls is very useful. A final word of warning on CTAGS: it's not always 100% reliable. I once spent a whole day trying to establish why a line of source code seemed to indicate a `0x0060` should be sent over the wire while the corresponding USB packet dump indicated it was actually a `0x0040`. It came down to two different drivers with the same function name—CTAGs happened to have taken me to the wrong file—I was reading the code for a very similar but different driver.

One thing in particular to look out for in the source code is the term `urb`. The Linux kernel has a USB subsystem which handles low level I/O between the host and peripheral. This subsystem speaks in terms of URBs (USB Request Blocks). For example, you might see calls to

`usb_submit_urb`, at which point you know you're pretty close to the code doing the actual I/O. At this point, don't get too caught up in completely understanding the source code. Simply take a cursory look over the code, digging into some of the function definitions along the way. The next steps will help verify that how you believe the driver behaves based on the source code is how the driver actually behaves in practice.

Packet Dumping USB

You'll notice that at this point we haven't talked much about the USB protocol. The truth is, I'm not an expert on the protocol; in fact, I only have a rudimentary working knowledge of the basic transfer types, which, it turns out, is sometimes enough. Rather than diving into the technical details of the USB protocol here, I strongly suggest you instead consult a book on the protocol. I've found the first few chapters of USB Complete contain enough information to get up and running. The most important thing is understanding the different types of USB transfers and how they're used. Alternatively, you might refer to the free USB in a NutShell website. Once you understand the basics, you'll be able to read USB packet dumps.

If you've ever used Wireshark to capture network traffic, you'll know how useful a packet dump can be. It's essentially a debugger for the network. Thankfully, Linux comes equip with a command that will let you do the same sort of thing for USB. It's called `usbmon`. On some Linux distributions, to get access to the `usbmon` command you'll first need to load the module with `sudo modprobe usbmon`. To avoid dumping *all* USB traffic, you'll want to target only the specific bus your peripheral is running on. The bus information was in fact shown earlier in our call to `lsusb`. For example, my device is operating on bus 002:

```
Bus 002 Device 011: ID 0bda:8187 Realtek Semiconductor Corp. RTL8187 Wireless
Adapter
```

At this point, unplug the USB peripheral. Now, with the bus number in hand, you can start capturing the USB packets, with a call like:

```
sudo cat /sys/kernel/debug/usb/usbmon/2u > /tmp/my_peripheral.mon.out
```

Note the `2` in `/sys/kernel/debug/usb/usbmon/2u` is indicating bus 2. So to monitor bus 1, you would instead use `/sys/kern...mon/1u`.

If I recall correctly, it's important to ensure that the peripheral of interest is the only peripheral operating on the given bus (which you can verify by reading the output of `lsusb`). For example, if a USB mouse is connected to the same bus, the trace will be mixing packets from the two devices. You may need to experiment with plugging the peripheral into different USB ports to find an empty bus. More information about the `usbmon` command can be found here.

What I ended up doing with the packet dump produced by my wireless USB peripheral was to print the ~250 pages of hex and work through it in lockstep with the Linux source code. This turned out extremely useful in cross-referencing the pattern of USB messages observed over the wire against the Linux driver source code. Every time I found a perceived discrepancy between the two, I'd carefully work through the code until I could see how the two were consistent.

Adding Logging to the Kernel Driver

Another approach I found quite useful in analyzing the kernel driver was to introduce logging statements into the driver source code. This allowed me to log variable values as the driver executed, which I would again try to match to hex codes in the packet dump.

Under Linux, you can use the `printk()` function to print to the kernel log. You may find that there are already calls to `printk()` dispersed throughout the driver you're analyzing. Essentially I downloaded the Linux source tree, made the relevant modifications to the code, compiled the modified driver and inserted the new module into the running operating system. If your peripheral's driver is a kernel module (as was the case with mine), you can perform this operation without needing to reboot the computer. First however, you'll need to establish the name of the kernel module that corresponds to your driver. The unsophisticated, though effective method I used to achieve this was to:

1. Restart Linux without the peripheral connected
2. Run `lsmod` and keep note of the results
3. Plug in the peripheral
4. Run `lsmod` and compare the output to Step 2.

If your driver is indeed a module which is dynamically loaded the first time the peripheral is plugged in, you should see changes between the two runs of `lsmod` which give away the name. For example, the module name for my wireless USB peripheral is `rtl8187` and appears in the output of Step 4 but not Step 2. If this doesn't work for you, perhaps your driver isn't implemented as a dynamically loaded module. Again, if anyone would like to contribute as to how this scenario should be addressed, please contact me so that I can update the article.

With the module name in hand, `cd` to the root of your modified Linux source directory (e.g. `~/linux-source`), then execute the following:

```
~/linux-source$ make
~/linux-source$ make ./drivers/path/to/your/driver.ko
~/linux-source$ rmmod <your_module_name>
~/linux-source$ insmod ./drivers/path/to/your/driver.ko
```

The calls to `rmmod` and `insmod` should replace the old module in the running kernel with your modified module. The output of calls to `printk()` can be found in `/var/log/kern.log`.

Porting the Code to Android

What's outlined above is a general guide to finding, exploring and analyzing the behavior of an existing Linux USB peripheral driver. With these tools in hand, it's time to finally begin porting the code over to Android. Perhaps surprisingly, this is actually now the easy part. If you've followed faithfully the steps above, you will have a reasonable idea about how your chosen peripheral works, or at least some idea of the messages you need to emulate on the wire. From here, you can simply follow the Android USB Host API Documentation. The documentation describes everything you need to know about registering your Android application to be notified when your target peripheral is attached based on the Vendor ID and Product ID. All of the methods used to initiate I/O between

the Android device and the peripheral are also documented. Again, it’s important to understand the basics of the USB transfer types which are not covered in this article. Once you have this knowledge however, everything else becomes easier.

Conclusion

Hopefully this article demystifies what’s involved in porting a USB driver from Linux kernel space to Android user space. Again, this article has been written based on my own experience—if you have particular domain expertise in some of the areas I’ve glossed over, please get in touch so that I can improve the article.