Contents

1 Abstract

In this essay I try to reason with myself as to what the best design decisions for this firmware should be. It starts by introdcuing the good ol', honest to God, USB keyboard. No fancy firmware just a simple application of the HID class from the USB protocol. It covers its limitations and how a custom firmware might help. Finally it also helps me test my new emacs latex workflow.

2 HID class keyboard

2.1 Introduction

The USB protocol defines the HID class which in turn provides a standard for generic USB keyboards to follow. The standard defines certain codes refered to as "HID key codes" defined under the "HID Usage Table" document. All a generic USB keyboard needs to do in order to actually function is implement the basic USB functionalities (transactions, enumerations, yada yada) and generate reports containing those keycodes and boom, you are typing. The implementation part is rather straight forward, albeit the USB protocol isn't.

Notice that the keycodes, while containing most of the used characters, does not contain all ASCII let alone Unicode characters, what's up with that, you might ask. Well, simply put, that document would be a lot longer if every character were to be implemented by the HID class. That still doesn't answer how we are even possible to print input accented characters and what not, most notably on ISO and international layouts since ANSI keyboard layout doesn't have accented characters to begin wtih. To answer

that we must consider the selected keyboard layout under the Operating System.

2.2 Physical and logical layouts

A course divergence is now nescessary. Let's talk about layouts, specifically what they are and how they are implemented under the HID class. A quick google search shows the difference between the ANSI and ISO layout. The characters printed by some keys are different (number row), some extra keys are added on the ISO layout and so on. Ignoring the extra keys for a moment, what is the difference between the firmware of an ANSI and ISO keyboard? Absolutely none, which is beautiful and confusing at the same time. Let's consider how things are happening at a firmware level for a second. User presses 'a' key, firmware detects it and sends* the key to the computer, well we are in luck because the 'a' key is the same in both layouts but how about the '`' key, more specifically Shift + '`' (shift and other modifiers will be covered soon). Well now what, that key in the ISO layout is the negation character while in ANSI is ~ so what is actually being sent. The keyboard will send the exact same keycode on both layouts, the keycode of the ~ character. The same is true for every other key that is present in both layouts, in short, their physical disposition about the keyboard will - more or less - determine its keycode and the keycode is the same as it would be in under the ANSI keyboard layout. The exception to the rule is the extra keys, which get a special keycode for them such as "0x64" lovely named as "Keyboard Non-US \ and |", very descriptive yet not very helpful. So how is it that the computer prints the right character? Your best friend, the Operating System of course. The logical layouts selected on your computer such as "US-English" or "ISO-English" is responsible for detecting the keycodes and making sure the correct characters are displayed in your machine. Those layouts do more than just intercept and translate keycodes, they can also add a few cool features such as dead keys (ie a key that does nothing until another key is pressed). That awesome feature is not even mentioned in the HID Usage Tables, maybe it is though since I haven't read all of it, point is: that is done by your lovely OS and that's not all, most OSes give the option for a custom user layout (yes, **even** windows. Check this puppy out: `https://www.microsoft.com/en-us/download/details.aspx?id=22339`). Which means you can create your own dead keys, make all sorts of crazy layouts. Heck, you can even add greek letters to it! I'll stop here however since this gets outside my scope of knowledge because I don't know which characters are actually available, maybe it has full UTF-8 support maybe it doesn't,

point is it opens a world of possibilities.

The points just made are very important since it shows that despite the limited ammount of keycodes defined under the HID Table most characters are still printable, all thanks to the Operating System. What's the problem then? Well, for one - it's not very portable. Sure you can do your own crazy layout but that's stuck to your box and it depends **heavily** on your operating system. Different operating systems and under same layouts can send different keys! Don't believe me? Get yourself Brazilian ANBT-2 keyboard. The \ key will send different symbols under Linux (tested on Ubuntu) and Windows while the logical layout is set to US-English, same keycode different OS implementation (though that's an edge case since that keycode is not even part of the ANSI layout).

*Yes, the usb protocol is host controlled but such details are not relevant for the current discussion.

2.3 Modifiers

Now, modifiers! The Ctrl, Shift, Alt and Windows - or GUI as it is called on the HID Table - keys on that 5 bucks USB keyboard off of Amazon. What's up with those and how do they work? It's honestly a really neat and beautiful implementation - at least in my opinion. It should be made clear that they are keys on their own! Though a single Ctrl key press doesn't really do anything under normal OS conditions, it is still a key with a unique keycode. The thing is, the Operating System interprets the 'a' differently if the Ctrl key is pressed. So by pressing Ctrl + a in your keyboard, you're in fact sending the Ctrl keycode and the 'a' keycode, this fine point is **very** important, especially once we start talking about layers under custom firmwares. The same is true for all other 7 modifier keys (oh yeah, it's 8 total. Left Ctrl has a keycode and so does Right Ctrl, though most of the time they are equivalent). This allows our little keyboard to have a really large number of unique key + modifier combinations since modifiers also combine (Ctrl+Alt, Ctrl+Shift, etc). At a low level the way it works is: the report sent to the computer has a Modifier Byte and each of the eight modifiers represent a bit on that byte. If the bit is 1 then the key is pressed.

2.4 Modifiers II

The importance of modifiers cannot be overlooked. It is a simple concept but a powerful one. It turns the arrangement of keys from rather limited (in terms of keybinds and customization) into something extremely powerful

with a myriad of combinations. In the spirit of keeping the tradition of modifiers, I used some of their principles as motivation for my initial design decision in earlier versions of the firmware, decision which I then started to question. In the interest of making this firmware a good piece of software, I decided to go back to the drawing board and start from bottom up, I wanted to really understand what made a keyboard a keyboard thus this essay. The surprising thing is, I gave a shot at abandoing my whole pre-conceptions and tried a fresh start, I have reached the same conclusions. The way modifiers are implemented under the HID Class is elegant and I will - once again - use it as my motivation for this firmware.

The characteristics that make these modifiers powerful are:

- Modifier combinations. Combining modifiers is a simple way to expand the ammount of key combinations available. Multiple modifiers can be combined together (Shift + Crtl, Alt + Shift ...), each modifier functions on their own and releasing a modifier removes that modifier from the next USB report.

- Key interactions. Due to how they are implemented modifier keys effectively change all the other keys in the keyboard. Since modifiers are kept track of in a different fashion as to other keys they allow for different end results. While an 'a' and a 'b' don't produce any special interactions a Ctrl 'a' does and that is - partialy - thanks to the fixed position of the modifier in the USB report which facilitates tracking whether the modifier is on or off.

It is under those keypoints that I chose to base my design decisions regarding layers and layer keys on this custom firmware. Following a similar philosophy and usability that is present in every generic keyboard, while extending on it, it is possible to make truly flexible mappings for custom keyboards.

2.5 Conclusion

The rather clever implementation of modifier keys significantly increases the usability of a generic keyboard and they are - in a sense - "special" since that's how they were implemented to be. They are the only keys that interact with other keys in the keyboard, they make the 'a' no longer behave like 'a' and while pressing 'a' + 's' under a generic keyboard doesn't really do anything unexpected, what if it did? What if we could build on and expand all the possibilities modifier keys gives us? Custom layouts gives great power

and flexibility to users but it can only do so much. For a truly customizable experience we need to resort to the keyboard itself, let's make a crazy firmware!

3 Custom firmware