Note/1 MIDI Driver Report

This driver is a continuation of the work set forth by Kelly Hirai and Nathan Lay. The driver they parted with was one that had a fairly workable functionality for writing to the device, but the reading from the device was not doing anything. My work was mostly taking what was already there and hopefully figuring out the last mystery of the Note/1 driver.

My first endeavor was to get the device to compile on the machines under kernel 2.6.25.3. This was not too difficult, as there was only one or two API changes that were minor, as well as a few deprecated libraries that needed to be removed. At the moment there is still a “kernel section mismatch” warning when I compile the code, but it does not seem to change how the functions work (everything works as intended).

I tested the functionality that was said to work as heavily as I could and I found something interesting. At the end of each put() sequence (such as “F0 00 00 37 01 00 F7”), an additional data byte “FE” was inserted at the end, no matter what the sequence is. I can only guess what this means, but it seems fairly obvious (coming from the fact that I blew up the driver when I blocked this byte). This seems to be a terminate byte, signaling to the device that it has all the data it is going to get.

With this in mind, I decided to write the above sequence to the device (which happens to be the sysex command for “Return ROM version” ). When I did this, I got a rather shocking thing to happen: a response from the device, but it was infinite looping the same byte. After many reboots, I discovered that the response byte was in fact the proper first byte of the response. Using the functions that were already there (note1_get_adv in particular) I was able to make the byte advance, and get all the response bytes. I played around with other sysex commands (found in the sysex.txt file in the DOS drivers) and got responses from them too.

Seeing that the put() functions were working, and the get() functions were sort of working, it has lead me to believe that there is some form of structure to getting the device to respond to write things into the buffer. Since the sysex commands trigger a write into the buffer (multiple sysex commands triggered WILL queue themselves consecutively), I believe that in order to put something in the device’s buffer to read form has to be done in a very structured way, as the rest of it is done, but I am not certain of this, more disassembly/reverse engineering will be necessary to determine if this is the case.
I have a few additional things for those who might work on the device in the future. The first is that if there is anything in the buffer when you unload the driver it will blow up, and cause a need for a reboot. Second, the /dev file is strangely missing for this device (I haven’t really looked into why, as I was focused on trying to do what I could with reading from the device, but it could be useful in the future). Also, there are functions that were existing that could prove to be very useful (such as note1_pre_get) in future rewrites /cleanups of the code. The note1_get_adv function might also be better served to be a void returning function, instead of returning the value of the TX register. I left that function alone because I could not decide which manner would be better to rewrite the functions so that they would perform well, while being as easy to maintain as possible. One thing I did not try, but might be worth testing is to see what happens to the device when there is no data on the line, and there is an attempt to advance the byte that the device is pointing at. I am unsure how this might respond though, as the buffer appears to know when it is empty or full, and handles that on its own.