The Discovery Process and Implementation of midi on the Music Quest Note/1 Parallel Port Adapter.

Kelly Hirai hirai@cs.fsu.edu Nathan Lay nslay@cs.fsu.edu
For Linux Device Drivers Summer 2006. Instructor: Dr. Ted Baker

The Note/1 itself

The ALSA is a 80C251 CPU based parallel to midi processor running on a 12 MHz clock. The 80C251 is available with up to 4Kb of inboard EPROM and 256 bytes of RAM. A parallel i/o bus and a full duplex serial port are built on chip. [1] The Note/1 surrounds this chip with some inverters and multiplexers so that it can share the port with a standard parallel port line printer. For the midi side, the midi-in is protected by the usual optoisolator, the midi-out has the usual current limiting resistors.

One would think this overkill for a simple serial to parallel message pass thru scheme but, Note/1 is reported to implement event filtering and channel remapping by receiving specially marked midi sysex messages. It is also able to report its current ROM version, test its internal RAM, and do a checksum on its ROM through the use of midi sysex messages. [2]

Trees we have barked up..

The Note/1 was designed in 1991 by the Music Quest company under the leadership of Paul Messick. Messick went on to write the book Maximum Midi [3], targeting windows 95 C developers writing midi applications. He has since gone into photography and has not responded to e-mails.

Music Quest was bought out by Opcode Systems, which was subsequently bought by Gibson (Gibson Guitars). All of whom have been unreachable.

Voyetra's DOS based sequencer, Sequencer Plus, came bundled with a driver for the Note/1. Voyetra merged with Turtle Beach. When asked about possible documentation for the Note/1, they responded that all they could offer us was all ready on the site. From them we obtained all the dos software. [4]

User space tools for midi.

ALSA, Advanced Linux Sound Architecture. provides a standard mechanism for accessing sound and midi devices. The alsa-utils package provides some basic command line utilities for reading and writing to raw midi ports. The most useful to this project being:

- amidi: to monitor a midi port, or to send raw hex to a midi port.

- aplaymidi: to play a midi file through specific midi port, in real time
- aseqdump: records midi messages with time stamps in human readable form

The other important tool to get this project right was to grab a sound card (an ens1371 it turn out) with an MPU-401 midi port (joystick port) and get the ALSA drivers for it working. Cabling the midi ports from the Note/1 to the 1371, we could now send and detect midi messages to and from the Note/1.

First naive attempts.

The first logical thing to do was to send random data to the parallel data register and see if anything turned up. Using the raw parallel port interface module parport.h we sent a series of bytes through the Note/1 and listened from the 1371. By some miracle, we received the hex 0xff from the midi-out port, indicating that the device was alive. Unfortunately, we were not able to reproduce this, and considered it a fluke. So we resolved to get the DOS software and make sure the device worked.

DOS emulation, dosemu/freedos, qemu/dr.dos with the Voyetra drivers and sequencer

Dosemu installed the voyetra sequencer flawlessly. The drivers loaded without a hitch. Running the voyetra MIDITEST.EXE while monitoring the 1371 produced the following output: F0 00 00 37 01 16 00 F7 from the Note/1's midi-out port. This is a midi sysex message telling the Note/1 to report its rom version number. After re-connecting the Note/1's midi-in to its midi-out, MIDITEST.EXE was unable to succeed under dosemu. We speculated that irqs, and latency ere contributing to the program being unable to read from the midi-in port of the Note/1. We resolved to try qemu, another emulator.

Qemu, had difficulty loading the voyetra software. We couldn't change floppies under it. In order to see if read was working, we built a DOS partition on the hard drive and booted directly into DOS. Under these conditions the Note/1 passed MIDITEST.EXE and ran smoothly. Seeing this, we booted back into Linux and qemu'd the DOS partition. Qemu generated the same signals on the midi-out port of the Note/1, but was also unable to get MIDITEST.EXE to pass. Nate felt confident he could hack the sources of qemu to report the contents of the parallel port activity, so we did that.

Logging parallel port activity with qemu.

The code modifications to qemu are in from the file vl.c They produced a timestamped log of the transactions of the parallel port. Running the Voyetra software has the following phases: booting, loading the driver VAPIN1.EXE, MIDITEST.EXE timer tests, and MIDITEST.EXE i/o tests. The following chunks were then carefully analyzed by expert an expert in musical analysis. The boot process was ignored mostly because it's probably part of the power up self test process. The timer tests we difficult to make heads or tails but may turn out to be

an important part of the read method. The write method turns out to be fairly straight forward. Here is an annotated excerpt from the log:

```
8 PPRSTATUS ERROR SELECT PAPEROUT BUSY 0xb8  ---- good to go.
u: 22                                        --- f0 when data present
                                            --- or more likely
                                           --- buffer full

8 PPWDATA 0xfc  -----  data1
u: 13
8 PPWDATA 0xf0  ----- data2
u: 16
8 PPWCONTROL STROBE INIT 0x05   ----
u: 19                               |- signals processor
8 PPWCONTROL INIT 0x04          ----
u: 12
8 PPWDATA 0xf3   ---  data3
u: 10
8 PPWDATA 0xff   --- blank register
u: 27
```

`8` is the file descriptor, `PPWDATA` signifies `(W)`riting the the `DATA` register of the parallel port. `0xnn` is the actual byte sent or read. `u:` marks the microseconds between instructions. if an `s:` appears before it, it identifies the seconds component of the delay. The above segment we have identified as the write loop. This loops with the following data:

data1	data2	data3
0xfc	0xf0	0xf3
0xcc	0x00	0x33
0xcc	0x00	0x33
0xff	0x37	0x37
0xcd	0x01	0x33
0xde	0x16	0x37
0xcc	0x00	0x33
0xff	0xf7	0xf7

It turns out, `data1` is `data2` or'd with `0xcc` and that `data3` is `data2` or'd with `0x33`. Looking at this bitwise, `0xcc` is `11001100` and `0x33` is `00110011`. This suggests that there is some timing specific select circuits operating inside the Note/1. The `data2` column contains the sysex command, "report rom version". This paradigm is implemented in the function:
`note1_put(struct parport *port);` in file `note1_get_set.c`

Further on down the log we find another loop that appears to contain the data read back. This part is receiving the byte `0xf7`, the start of a sysex message.

```
8 PPWCONTROL AUTOFD INIT 0x06     --- any data???
u: 13
8 PPRSTATUS SELECT ACK BUSY 0xd0  --- yes.
u: 13
8 PPWCONTROL INIT 0x04            --- ok.
u: 14
8 PPRSTATUS ERROR SELECT PAPEROUT ACK BUSY 0xf8
```

```
u: 12
8 PPWDATA 0xfd
u: 13
8 PPRSTATUS ERROR SELECT PAPEROUT ACK BUSY 0xf8 --- 11
u: 12
8 PPWDATA 0xfc
u: 11
8 PPRSTATUS ERROR SELECT PAPEROUT ACK BUSY 0xf8 --- 11
u: 14
8 PPWDATA 0xfe
u: 11
8 PPRSTATUS ERROR SELECT ACK BUSY 0xd8         --- 01
u: 13
8 PPWDATA 0xff
u: 11
8 PPRSTATUS ERROR SELECT PAPEROUT ACK BUSY 0xf8  --- 11
u: 12
```

The status register appears to be laid out as follows:

```
bit   code  meaning...
0     --    unused
1     --    unused
2     --    unused
3     d0    low order data bit
4     --    ? always 1 dunno....
5     d1    high order data bit
6     Rx    read ready  1 = data on the line, 0 = empty
7     Tx    write ready 1 = ready to transmit, 0 = not ready
```

Writing the following bytes to the data register seems to select which bit pairs are being read:

```
byte  bits_appearing_in_status_register
0xfd  ..10....
0xfc  10......
0xfe  ....10..
0xff  ......10
```

This paradigm is implemented in the function:

`u_int8_t note1_get(struct parport *port);` and
`u_int8_t note1_get_adv(struct parport *port);`
in file `note1_get_set.c`

The former returns the get value. The latter performs what we felt would advance the input buffer to the next byte. It returns the content of the status register, which is used to signal if there is still data on the line.

Results.

The put() seems to be working as expected. In raw parport module tests, the get has successfully read up to about 48 bytes but then suddenly quits. Inside the ALSA reading trigger hook, get() doesn't return any data and keeps the module from unloading. We have scanned the parport and system logs for clues to as to

why this is. The lack of a get() function motivates the following strategy toward developing a complete driver.

TODO

Perhaps when you do a get() and there is not data on the line, it puts the system in a funk that requires some sort of reset. Or possibly, every time you reach the end of the Note/1's internal read buffer, you have to manually repoint to the beginning of the buffer again. Figure this out.

Interrupts are being generated by the device, as reported in the system log during the pre-ALSA tests. Maybe ALSA, not having been given an irq handler can't get started when its time to read from the device. Or maybe it is polling the device into a bad state. Perhaps the device is waiting for some handshake before it will hand over the data that has been signaled present by an interrupt. We don't know. Figure this out.

The things we write to the control register: 0x00, 0x04 0x06 we have been using colloquially and it would be nice to know the meaning of the bit patterns.

When there is no data on the line, the data bits in the status register could have some possible meaning regarding other states of the processor.

Sometimes the routine writes successive 0x00, 0xff to the data port. We'd be interested in why.

The timing test part of the log begins with a preamble, then iterates some fairly tight loops of write data, write control, read status. We'd like to know what this is really doing. Really we should implement this in a function and try calling it during the ALSA's open, drain, and close functions.

We have yet to really tune the delays between reads and writes.

Cites and Sources:

[1] http://www.datasheetcatalog.com/datasheets_pdf/I/P/8/0/IP80C521.shtml
[2] file SYSEX.TXT in ftp://ftp.voyetra.com/pub/voy/seqplus/seq_gold.zip
[3] https://secure.manning.com/books/messick
[4] http://support.turtlebeach.com/site/kb_ftp/340.asp

Team blog, https://bugtrack.alsa-project.org/wiki/wikka.php?wakka=MidiQuest
Alsa Documentation,
http://alsa-project.org/~iwai/writing-an-alsa-driver/index.html
Kernel Sources