

The Note/1 Driver Design

Introduction

The Music Quest Note/1 midi engine is a small midi controller that communicates over the parallel port. It features two midi ports, one for input and one for output. It also features 16 midi-enabled filters and the ability to share the port with a printer. The actual hardware is a black box, it has an Intel PROM chip that holds the mysterious workings of the device itself. What *is* known is that the device has a 32 byte buffer and appears stateless. The device itself supports reads and writes and is possibly full duplex. Writes are done byte-wise over the data register while reads are done 2 bits at a time over the status register. The status register uses 4 bits:

0x08 - Data bit 1
0x20 - Data bit 0
0x40 - Data RX
0x80 - Data TX

The control register bit meanings are still not fully understood but only the 4 lower bits are ever used. Additionally, the device can deliver interrupts but those appear to only happen on read. Likewise, there is a certain discipline manipulating these registers. Usually a write to control and/or data takes place, then the status is examined. However, there appear to be exceptions to this rule where strange bytes are flashed over control and data registers.

The Development Method

The driver was written in revisions. There were only 4 revisions when this paper was written. The first revision was written for a rough idea of how data and functions were to be laid out while the second revision was a complete re-write, it was intended to be a skeleton for future revisions. The third and fourth revisions were improvements over the second and third revisions respectively, these featured layers of API not present in the previous. These include parport, character device, and ALSA abstraction. This is a summary of all the work done in each revision:

Revision 1:

- Basic module, implemented parport

Revision 2:

- Rewrite
- Implemented parport
- Implemented character device abstraction for parent and child devices.
- Added a single ioctl command for probing.
- Wrote note1ctl that commands a parent node to probe for newly attached Note/1 devices.

Revision 3:

- Added /proc entry that displays device information for each attached Note/1 device.
- Removed character device support for child nodes, this wasn't needed.
- Added ALSA support.

Revision 4:

- Changed locking methods for parport.
- Improved read/write ALSA methods.

The Design

Initially one of the major design concerns was the ability to stack on the existing parport driver. Along with this concern was the highly unlikely event that somebody might have more than one Note/1 device attached to the system. Luckily, the parport documentation guarantees us that there will be no more than PARPORT_MAX parallel ports supported. For the PC, PARPORT_MAX defaults to 16. So, in light of this information, the driver holds an array of notel_device pointers set to the size of PARPORT_MAX. On initialization this array is NULL'd out.

```
struct notel_device {
    struct semaphore sem; /* For locking */
    struct pardevice* par; /* Pointer to our port */
    struct snd_rawmidi* rmidi; /* ALSA Raw Midi */
    struct snd_card* card; /* Our Sound Card */
    int busy, claimed, opened; /* parport preempt lock */
};

static struct notel_device* notel_devs[ PARPORT_MAX ];
```

This array is primarily used for lookup and cleanup as it is used to manage our memory. But before we can make any use of this, the driver must be registered for parport.

```
struct parport_driver {
    const char *name;
    void (*attach) (struct parport *);
    void (*detach) (struct parport *);
    struct parport_driver *next;
};

int parport_register_driver (struct parport_driver *driver);
```

The parport_register_driver is used to make the parport driver aware of our driver as well as provide access to the parallel port. The attach and detach functions our ways to make our driver aware of the parallel ports available and the attach function is triggered for all available ports as soon as the driver is registered. Likewise the detach function is called when a parallel port is no longer available to the system or when parport_unregister_driver is called. The attach function is a perfect place to probe for devices, in fact that's exactly what happens.

Our real memory management happens in the attach function. First, notel_getadd()


```

    if ( snd_card_register( (*p)->card ) ) {
        printk( KERN_ALERT "%s: Unable to register the card!\n",
                NOTE1_DEVNAME );
        snd_card_free( (*p)->card );
        *p = NULL;
        return;
    }
}
if ( !notel_register_port( *p, port ) && notel_probe( *p ) )
    notel_unregister_port( *p );
else
    printk( "Found device!\n" );
}

```

It is of note that we rely on `snd_card_new` to allocate our memory for convenience reasons. This, of course was only recent to Revision 3 since ALSA API was added in that revision, before it used `kmalloc`. One caveat to this memory management technique is that the initialization requires us to register the card, even if there might be no Note/1 devices connected. Registering the card makes our callbacks available to ALSA (they will always fail if there is no device).

After initialization, the device attempts to register a port to itself with `notel_register_port` so it can then attempt to claim the port as well as register parport specific callbacks (such as an interrupt handler). This function only relies on `parport_register_device`:

```

struct pardevice *parport_register_device(struct parport *port,
                                         const char *name,
                                         preempt_func preempt,
                                         wakeup_func wakeup,
                                         irq_func irq,
                                         int flags,
                                         void *handle);

```

Where `preempt` is the parallel preemption function, `wakeup` is a function that reacts on an available port and `irq` is the interrupt handler. The `preempt` function will surrender the port to another driver if it is not in use, likewise, the `wakeup` function will be triggered if a port becomes available. The Note/1 driver only uses `preempt` and `irq` since it is able to share the port with a printer and delivers interrupts. If this succeeds, `attach` is able to call the probing function which determines if the port is to remain associated to the device or not.

Put aside until now, the `attach` function also sets up the appropriate ALSA callbacks on initialization. Since our device is independent of any recognized sound chip, we must use the Raw MIDI API that ALSA provides. The callback functions associated with Raw MIDI include:

- Input/Output open - Opens a stream
- Input/Output close - Closes a stream
- Input/Output trigger - Triggers input and output for a stream
- and Output drain - Dumps the remaining output buffer to the device.

Since our device is byte stream oriented, it does not need a drain function. This is because `trigger` is guaranteed to take care of that detail for us.

To initialize the driver for use with ALSA, one must initialize a `snd_card`. A `snd_card` is a structure that describes a list of sound interfaces through components. One of those components include the Raw MIDI interface. Then one must register a `snd_rawmidi` structure into the card and associate the `snd_rawmidi_ops` input and output callbacks into the `snd_rawmidi` structure. Finally, `snd_card_register` makes the card available to the system for use, its important that this is done *after* initializing the `snd_card`.

There are two types of locking needed for this driver. There is the usual kernel lock (e.g. semaphore, mutex, spinlock ...), and to protect the claimed parallel port there is a busy counter. Since there is no real locking issue with concurrency, the big problem lay with port preemption. Initially, the `notel_device` structure had a mode mask that described the various states of the devices. These states included: opened, busy, and claimed. To protect the previous mode, functions that *wanted* to change the mode needed to store the current mode. Unfortunately, the problem with this approach became apparent with concurrency. Suppose the device is idle and its opened for input. It stores the current idle mode and sets the mode to busy. Now suppose, at the same time, the device is opened for output, it stores the current busy mode and sets the mode to busy again (which has no effect). Now, when the input method is finished it resets the previous mode it stored, which happens to be the idle mode, despite the fact that the output method intended to be busy too. So, to remedy this issue a busy counter is instead used. Whenever a function intends to be busy, it just increments the busy counter, and when its finished, decrements the counter, promising that the driver's mode is not compromised. To reiterate, this busy counter protects the port from preemption.

```
#define notel_isbusy( dev )                ( (dev)->busy )
#define notel_setbusy( dev )              ( ++(dev)->busy )
#define notel_unsetbusy( dev )            ( --(dev)->busy )

int notel_preempt( void* handle ) {
    struct notel_device* dev = (struct notel_device*)handle;
    if ( notel_isbusy( dev ) )
        return -1;
    notel_unsetclaim( dev );
    return 0;
}
```

When the cleanup routine is run, the module unregisters the associated `snd_card` structures (from the `notel_devs` array) with `snd_card_free`. It is of note to mention that this method removes the ALSA callbacks as well as free the memory for our `notel_device` structure.

```
void __exit notel_exit( void ) {
    struct notel_device** p;
#ifdef NOTE1_DEBUG
    printk( KERN_NOTICE "notel_exit() called!\n" );
#endif
    parport_unregister_driver( &notel_ppd );
}
```

```
for ( p = notel_devs; p < notel_devs+PARPORT_MAX; ++p )
    if ( *p ) {
        snd_card_free( (*p)->card );
    }

cdev_del( &notel_cdev );
unregister_chrdev_region( notel_major, NOTE1_DEVS );
remove_proc_entry( NOTE1_DEVNAME, NULL );
}
```