(D)ISKETTE (O)RGAN

Preface

In February 2011, I was cleaning out my closet when I found an old stack of computer hardware in the corner. It had been sitting there for a few years because I had intended to build something out of it, but never did get around to it. For whatever reason, I finally decided to do it then and there. A few days later I recorded and uploaded a demo video to YouTube, and to my surprise, it got a lot of attention!

Literally hundreds of people have been asking me how it all works under the hood, so I've finally put some materials together and slapped them up on the web for all to see. I apologize if it's a little rough at the moment, but I'll make refinements down the road as needed. If you have any questions or comments, feel free to let me know!

Introduction

The (d)iskette (O)rgan is a MIDI-controlled 4-voice polyphonic instrument. Now anyone can enjoy the subtle sputter and cough of a quartet of dying floppy drives frantically seeking nonexistent data... in the key of any. Plug it in, turn it on, grab your MIDI keyboard and press any key to jam!

(YouTube demo video)

Theory

Under the hood of a floppy drive, a magnetizable disk rotates at a fixed speed while a read/write head travels back and forth along its radius. The read/write head is actuated by a stepper motor, which is a special type of electric motor that rotates in discrete steps. It's the rhythmically pulsed movements of the transmission system between the read/write head and stepper motor that cause the delicious buzz and grind you know and love.

Beyond that, floppy drives aren't terribly sophisticated devices. They need to be instructed when to spin the disk, how far to move the data head, when to read or write, and you have to keep track of all this. There is no data formatting done for you, and you literally have to lay it down bit-by-bit. Obviously this is annoying and potentially error-prone for its intended purpose, but *great* for screwing around decades later after the format is long obsolete. So I like to pretend that the original architects of the technology and the pushers who kept it around so long sort of had me in mind.

To tame the outrageous buzzing noises into the delicate crooning of fine organ music, I had to capture and domesticate several temperamental drives from the wild. Their motherboards typically only have litters of one or two, and they certainly don't teach the drive cubs how to perform parlor tricks. To train *four* of them to *play music*, I had to build a custom controller interface to mimic their natural pack social order.

So basically, I've built an ersatz floppy drive controller that doesn't read or write data at all. Instead, it waits for MIDI note data, determines the corresponding frequency, and repeatedly instructs one of the drives to move its data head back and forth at precisely that frequency.

Limitations

Floppy drives were meant to transfer data to and from diskettes. If you want to do something other than transfer data to or from a diskette—like play music, for example—you may be better served by something like a piano.

Among other possible issues:

- Each drive has unique tonal properties.
- Each drive has a different (not necessarily contiguous) range of pitch.
- A single drive may sound different between play sessions.

- Floppy drives may fail prematurely and without warning.
- Some floppy drives may refuse to work at all in this setup.
- The main control firmware is very much a work-in-progress.

That being said, you might be surprised what you can get out of something like this.

PC Components

Case

Although not strictly required, a sturdy AT-style PC case is the simplest way to keep everything neat, tidy, and in one place. They also generally come with a…

Power Supply

Several floppy drives may need several watts of power. Rather than build a custom supply from scratch, I was happy to put the existing AT supply from my case to good use. While a PC power supply is easily overkill, it guarantees the capacity and connectors you need right out of the box. You can't beat the convenience.

> There is no need to open up or otherwise tamper with the power supply in any way. You could drastically shorten the operational lifetime of both it and yourself.

Drives

This is easily the most important factor to determining what sounds you'll get. The only way to know for sure is to find as many as you can, and test each one.

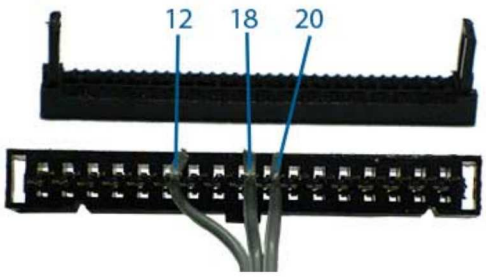
On an interesting side note, a lot of people joked that I should use some old 8" drives for extra bass. I didn't have any to test, but between the 5¼" and 3½" drives, the 3½" drives produced the most audible low frequencies, and the 5¼" drives produced the cleanest and highest notes. I hypothesized that the older drives were more over-engineered than their modern descendents, allowing them to operate farther out of spec, i.e. play higher notes. The newer drives are flimsier by comparison, and literally rattled better at low frequencies.

Cables

I attempted to find the cleanest and easiest way to hook the drives up. As it turns out, I only needed to use three signal wires out of the whole interface. I didn't want to add multiple bulky connectors to my little circuit, nor did I want to permanently solder three wires directly to each drive. The most reliable and convenient compromise was to modify an existing 34-conductor floppy cable.

Although 3½" drives have a *pin connector*, and 5¼" drives have an *edge connector*, they're otherwise the same thing as far as pin numbers and signals.

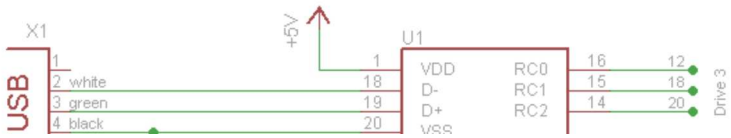
As you can see, I separated a group of 3 wires, and reconnected them to the appropriate pins in the connector.

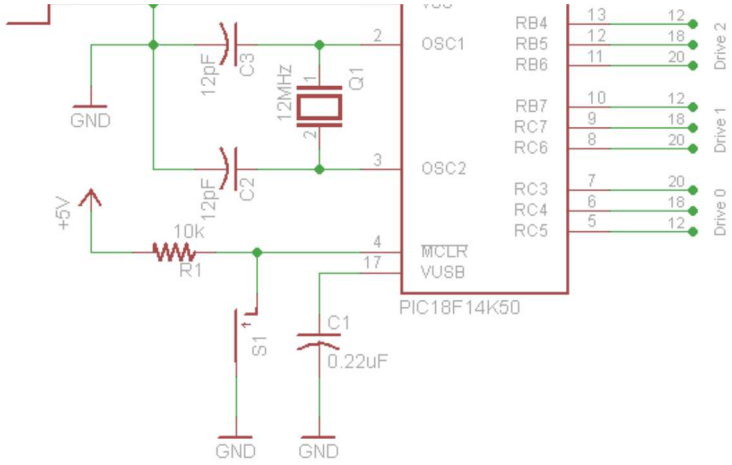


Controller Board

The circuit is totally bare-bones, and can live quite happily on a solderless breadboard. But in the end I opted to solder everything to a small Radioshack IC PC board (part #276-159B) because I shuffled it around a lot in the course of testing.

Schematic





Note: S1 is the PC reset button, and is used to quickly access the microcontroller update mode.

Parts List

- 1 x PIC18F14K50
- 1 x 12 MHz crystal
- 2 x 12 pF capacitor
- 1 x 0.22 µF capacitor
- 1 x 10 kΩ resistor
- 1 x cannibalized USB plug

Firmware

The hardware is as simple as it can get, but all the real action happens in the firmware. It's coded in C, and takes advantage of Microchip's USB stack library which makes MIDI-over-USB a breeze. (I mentioned elsewhere that this could also be adapted for genuine MIDI with a few extra parts, but I'll leave that as an exercise for someone else because I run all my MIDI into my PC anyway.)

There are two sections of code that make the magic happen: the main program loop determines what notes should play, and when; and a high-speed interrupt service routine drives the hardware at the requested frequencies. Here is a rough breakdown of what's going on:

Main Program Loop

After everything is all initialized and running, the main() loop waits for MIDI packets to arrive over the USB link. Once intercepted, the packet(s) are searched for MIDI NOTE ON and NOTE OFF messages; anything not explicitly handled is discarded. When a NOTE ON or NOTE OFF message is found (a MIDI key is pressed or released), the corresponding note number is stored in or removed from a global variable where it is processed at the next scheduled interrupt.

Interrupt Service Routine

The interrupt is triggered periodically at a rate of 100 kHz (once every 10 µs), and is directly responsible for making the drives buzz at the necessary times and frequencies. It's kind of brute-force but at 10 µs granularity, it's possible to multiplex four different floppy drives, at four different audible frequencies, totally asynchronously. (Ideally, things would be far simpler if each drive had a dedicated hardware timer, but there aren't enough available.)

So every 10 µs the interrupt routine increments a counter for each drive that should be playing a note. Each drive's counter value is compared to a reference value in a lookup table (conveniently indexed by MIDI note number; 0-127). If the drive counter is greater than or equal to the reference value, the drive stepper is pulsed once, and the counter is reset to zero.

Counter Reference Values

Since we're lacking a hardware floating point unit, the 128 MIDI note frequency reference values were precomputed and stored as (100,000 ÷ frequency). In other words, the period (T) expressed as tens of microseconds, rounded to the nearest whole integer.

I generated the values with a quick Perl script:

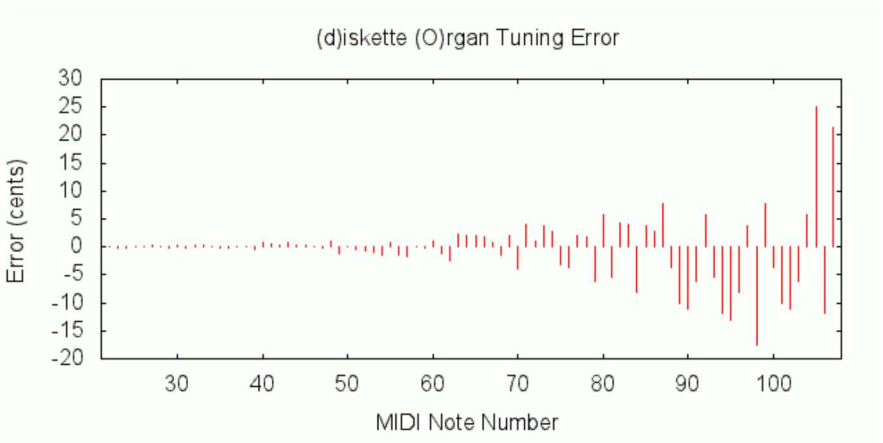
```
#!/usr/local/bin/perl

foreach (@values = 0..127) {
    $_ = 440 * (2 ** (($_ - 69) / 12));      # en.wikipedia.org/wiki/MIDI_Tuning_Standard
    $_ = 100000 / $_;                        # multiply period by 10^5
    $_ = int($_ + 0.5);                      # round to nearest integer
}

print join(", ", @values), "\n";
```

Tuning Error

Because of the way the drive steppers are timed, there is increasingly significant inaccuracy as the frequency increases. The following graph reveals the exact tuning error at each MIDI note on an 88-key keyboard:



As you can see, the error is ±5 cents pretty much up to around note 80. The usable note range fits well within the accurately tuned frequencies shown above. (For reference, middle C is note 60.) For all intents and purposes, this is a very accurate sounding instrument.

Source Code and Hex File

Disclaimer: the source code is incredibly messy and littered with old comments, old code, unused variables, inconsistencies, and other nastiness. This is only provided as a reference to the brave; download at your own risk. Hopefully it'll help you in your own endeavors. :) If you're a particularly special breed of masochist who wishes to actually compile it, you'll also need Microchip's MPLAB IDE, and the Microchip Application Libraries, v2010-10-19.

Because of the licensing, I shouldn't distribute the "complete" out-of-the-box sources. You'll have to go into the Microchip library source tree, make a copy of the "USB Device - Audio - MIDI\Firmware" directory, and merge my source files into it. Sorry!

(d)iskette (O)rgan source code

The hex file can be programmed over USB on a chip preloaded with the Microchip USB bootloader, using the bootloader program (source and executable found in the Microchip Application Libraries).

(d)iskette (O)rgan hex file