A New Hope

Author : Andras Tantos

Previously I've written about my failed attempts at obtaining any SW for my Cray-1 project. And that I've found another soul out there doing the same thing and getting nowhere.

After two years and another kid, I remembered this old project of Chris and started wondering if he gotten any forward progress. So I checked back on his site, and lo and behold: he made some tremendous progress!

He received an old disk containing some Cray code. Now, this disk is for a CDC 9762 drive, an old 80MByte device. Back in those days Hard-drives actually had removable platters, and came in washing-machine size. So Chris tracked down an old drive and tried to get it working. I won't recount his whole story here (you can read it on his blog linked above) but the short of it is that he couldn't so he essentially took a 'magnetic photograph' of the disks. A bunch of 0-s and 1-s that only represent the magnitude of the signal coming off of the head amplifiers. They're not the actual 0-s and 1-s stored on the disk. They're not aligned to sectors. Or tracks for that matter. And there are gigabytes of them. This is the beginning of one particular scan:

00111111111000000001111111110000000111111111000000011111111110000

You get the idea…

With the help of another guy in Norway they successfully cracked the format and recovered the actual sectors. The first disk turned out to contain nothing terribly interesting, but he eventually got his hands on an actual OS backup disk. With his 'imaging HW' and recovery script he was able to recover the content of the disk and posted it on his site.

I got **very** excited. This is exactly what I was looking for! Well, almost… Reading his full post shows that the image is for a Cray X-MP, not a Cray-1S I originally intended to build. Still, there's enough documentation that I can probably succeed in at least making a simulator for the Cray X-MP. With a little luck I can even build an FPGA version of an X-MP machine, if the – even crazier – memory subsystem doesn't require a cycle-accurate re-implementation.

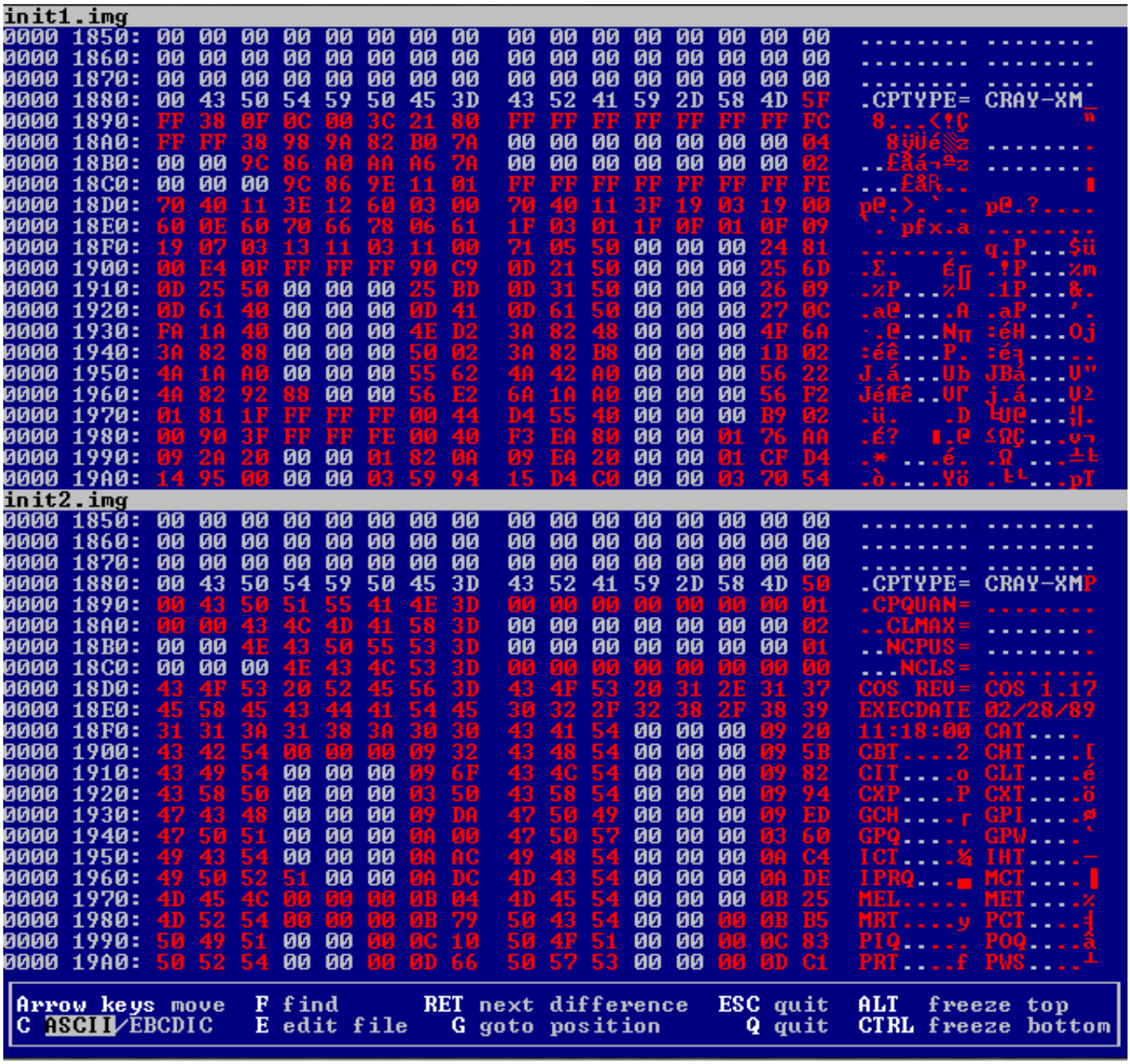
So I quickly downloaded his posted image and started poking around.

There are holes everywhere!

According to Chris the disk should contain the OS code for the mainframe. In fact it should contain two images, built at slightly different times, and apparently with slightly different configuration parameters. In those days the OS shipped in source-code form, and you compiled it yourself for your machine, baking in (at least some) of the particularities of the machine configuration. Pretty far from plug-and-play…

The disk image contained a lot of other things too (we'll get back to them later), but first I wanted to see how the boot images looked like.

I've started comparing the two images and soon realized that they were very different:



In fact, too different to make any sense: one image contains strings, the other contains gibberish, but roughly in the same layout? Than, mysteriously, the two images become the same again, and the same strings appear in both!?

The only explanation I could find is that the image got corrupted in some way. I've done some other experiments – following binary structures that I could find on the disk – and convinced myself that the image is in fact full of faults. Enough faults that – unless I figure out a way to

correct them – I have no hope to get anything useful out.

So, I did the only sensible thing: asked Chris for the original ‘magnetic photographs’ of the disk and started working on a better recovery tool.

Disk recovery

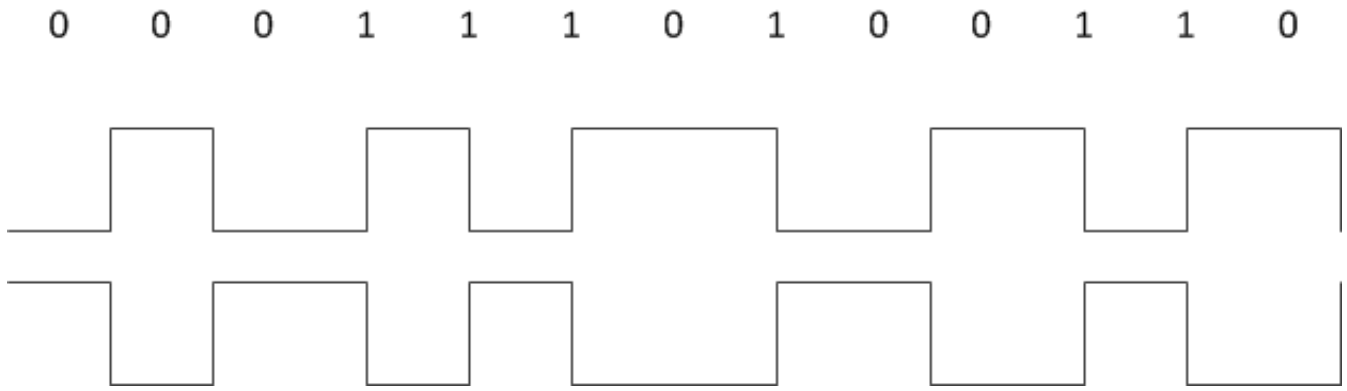
The CDC 9762 used several platters and heads to record it’s full 80MB of data. It organized the data in the usual ways: data grouped into sectors, recorded into tracks or cylinders by the heads on each platter:

(image source: wikipedia)

Each sector contained 4kBytes of data. For the actual information storage, it used MFM encoding. There are many ways to describe this encoding scheme, here's my attempt at it:

MFM modulation and reconstruction

The encoded stream contains low-to-high or high-to-low transitions at either the boundary or at the middle of a bit-time. The signal level has no meaning in the encoded signal, only the position of the transition. To encode a ‘1’, you put a transition at the center of the bit-time and no transition at the beginning of the bit-time. To encode a ‘0’ you put a transition at beginning of the bit-time if and only if the previous bit was also ‘0’. You never put a transition to the center of the bit-time of a ‘0’.



As I’ve said, the polarity doesn’t matter only the position of the edges.

The result of the scheme is that the length of a high or low pulse in the encoded stream is either 1, 1.5 or 2 bit-time.

One way of decoding an MFM stream is to use this interpretation: A 1-bit long pulse means that the next decoded bit is the same as the previous one. A 1.5-bit long pulse means that the next decoded bit is different from the previous one. A 2-bit long pulse means a 1-0-1 sequence.

Writing a decoder for something like this is rather straight-forward, provided you know two things: the bit-time, and the value of the start of the decoded sequence or – which is the same – which transitions are at the bit-boundary and which are at the center.

The bit-length problem

Let’s concentrate on the first problem: how to know the bit time, or to put it in another way, how to tell if a pulse is 1, 1.5 or 2 bit-time long?

When a digital signal is re-sampled, there is usually a 1-bit uncertainty about where the transition happen. So, even under ideal circumstances, there should be a 1-bit-time variation in the length of the pulses. In the picture below, both the green and red signals sample into the same bit-stream even though they’re shifted by almost a full bit.

If you plot the histogram of a perfect MFM signal, re-sampled at a significantly higher sampling rate, you should get something like this:

You can clearly see the groups corresponding to the 1, 1.5 and 2 bit-time long pulses. From this, it’s easy to recover the bit-time: calculate the local maximums (using some sort of interpolation) of the three ‘humps’, then divide the values by 1, 1.5 and 2 respectively, average them together and you have your answer.

Only when I’ve plotted the same of the sampled signals from the hard drive, I’ve got this:

It might not look horrible at first, but it is bad. While the same technique to calculate the bit-length works (you can see the ‘humps’), without a clear separation between the ‘humps’ it’s impossible to tell where the pulse lengths in the middle belong. Those in-between bars still represent hundreds or in some cases thousands of samples. And if I get only a single one wrong, the rest of the bit-stream could get reversed.

In other words, I’ll have to find a way to ‘clean’ this histogram up. If I can’t, that is if the signal is in fact that corrupted, all hope is lost. So I started thinking about what could cause the histogram to look like that.

One idea is that there is too much jitter in the signal, so the edges slide around randomly.

In this picture you see two bit-position where just a little bit of moving around of the edges (from green to red) causes a 0 to become a 1. These positions are marked with '?'.

There are entire books written about jitter alone, so I won't get into too much detail, but the canonical way of dealing with jittery signals is through PLLs. The trouble is that the re-sampling of the digital signal already introduces a lot of phase uncertainty (see above about the one sample period uncertainty) that would make implementing such PLLs in SW rather hard.

One possible cause of the jitter would be imperfections in the rotating speed of the platter. Either at the time of recording the information or during read-back. This is possible, especially since the two events are separated by more than 30 years, several thousand miles and are done on different drives. At the same time, why would the speed 'wobble'? These drives and platters are very heavy, with enormous inertia. It would take significant energy to **make** them wobble. Their natural tendency would be to run smoothly.

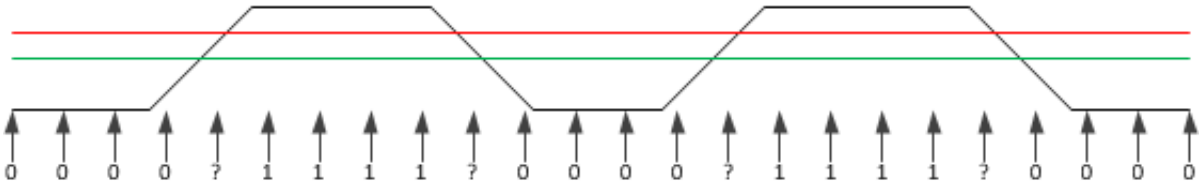
It is also possible that the signal is simply noisy, which results in the edges being sampled at different points.

Here you see how the addition of noise changes the sampling of (in this case) the rising edge on the signal.

Especially when the edge-rate of the signal is close to the bit-interval (which should be if the media is driven to it's capacity limits), this could be a cause of significant variations.

We'll see a little bit what can be done about this, but the key is taking multiple, uncorrelated samples.

The third idea is this: what if the comparison point wasn't perfectly in the middle of the signal swing? If that was the case, all the '0' pulses would be longer while all the '1' pulses would be shorter than they need to be. Or the other way around, depending on which way the comparison point have shifted:



This is easy to test: I just have to create a separate histogram for the '1'-pulses and the '0'-pulses:

Bingo!! This is something I can finally work with. You can see how the histogram of ‘0’-s (blue bars) and ‘1’-s (red bars) are shifted by about one sample-width.

Data-stream synchronization

Now, to the second problem: how to know which signal changes are on bit-boundaries and which are in the center. Or, in other words how to find if the decoded sequence starts with ‘0’ or ‘1’? (I’ll leave it as an exercise to the reader to prove that the two are the same).

The common technique (at least in PCs) seems to be that each sector starts with a set of 0-s (which translates into a series of 1-bit long pulses) followed by a special code 0xA1, with a missing clock

Why would this work? Notice, how a 2-bit-time pulse in the encoded stream corresponds to a 1-0-1 sequence? Every time you see such a sequence, you know the corresponding decoded

bits without ambiguity. You also know that the edges of the pulse must fall in the middle of the bit-time, not on the edge. You can use this as you synchronization point: from here on you know which edges are on bit-boundary and which pulses are in the center. In fact, if you see a 2-bit-time pulse where you expect a decoded 0, you know you have a decode error: all such sequences should start with 1. And this is where the missing clock in the special 0xA1 sequence comes in: it contains an artificial 2-bit long pulse in the wrong place. Because of that, this cannot be part of a normal data-sequence, so you know for sure that what you see is the beginning of a sector.

Here are some old MFM hard-drive controller manuals for reference:

- http://bitsavers.trailing-edge.com/pdf/seagate/ST412_OEMmanual_Apr82.pdf
- http://bitsavers.trailing-edge.com/pdf/westernDigital/WD100x/79-000029_WD1002S-WX2_XT_MFM_OEM_Manual_Jul85.pdf

Well, all this is nice theory, but this disk doesn't have any of these cool things. Here we only have a series of one-bit-time long pulses as the lead-in for the sectors, followed by the (binary) pattern of 0xf0. The reason they didn't have to use missing clock patterns and other synchronization method is that the designers reserved one of the surfaces of the disk pack is for servo information:

This servo surface contains signals for the head servo to find tracks, but also timing data to help locating the start of the sectors and the timing of the bits. This setup makes sense on large disk

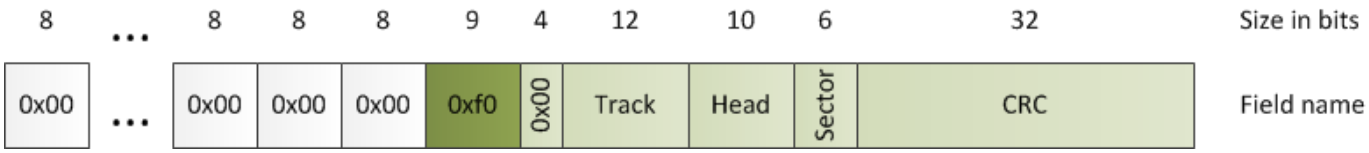
packs (there were drives working with packs containing 10 disks) where the overhead of a single servo surface is only 5%. With a small 3-disk pack like the one I have, the overhead of the servo surface is rather substantial. At any rate, this wasn't my design choice and it was made decades ago, so I'll just have to live with it. Furthermore, Chris's 'magnetic images' don't contain the servo head information, so we'll have to do without.

Not ideal, as it leaves the polarity to anybody's guess. Worse: if this sequence appears in the middle of sector (which it can because they didn't employ the missing-clock trick) we might synchronize to the wrong polarity. This makes it harder (but not impossible) and more ambiguous to recover sector starts if we ever loose sync.

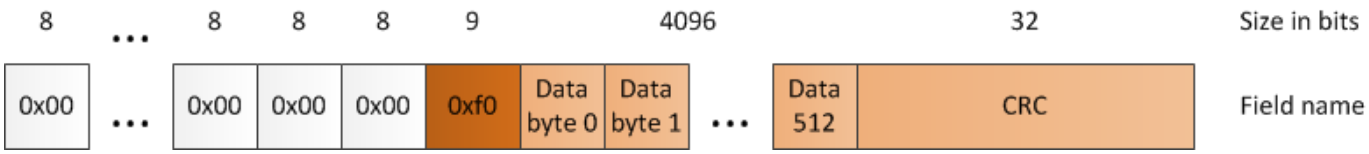
It seems the designers assumed a '0x00' as the synchronization pattern, at least that's how things work out. And as far as the ambiguity goes, we can use the structure of the sector header (where sector, track and head-group IDs are recorded) and the CRC at the end to increase our confidence that we in fact recovered a real sector.

Sector structure

Talking about structure, each sector is built up from two portion. A sector header and a sector data. Each portion has it's own lead-in and 'sync' byte, followed by the actual data and terminated by a 32-bit CRC. The sector header has the following structure:

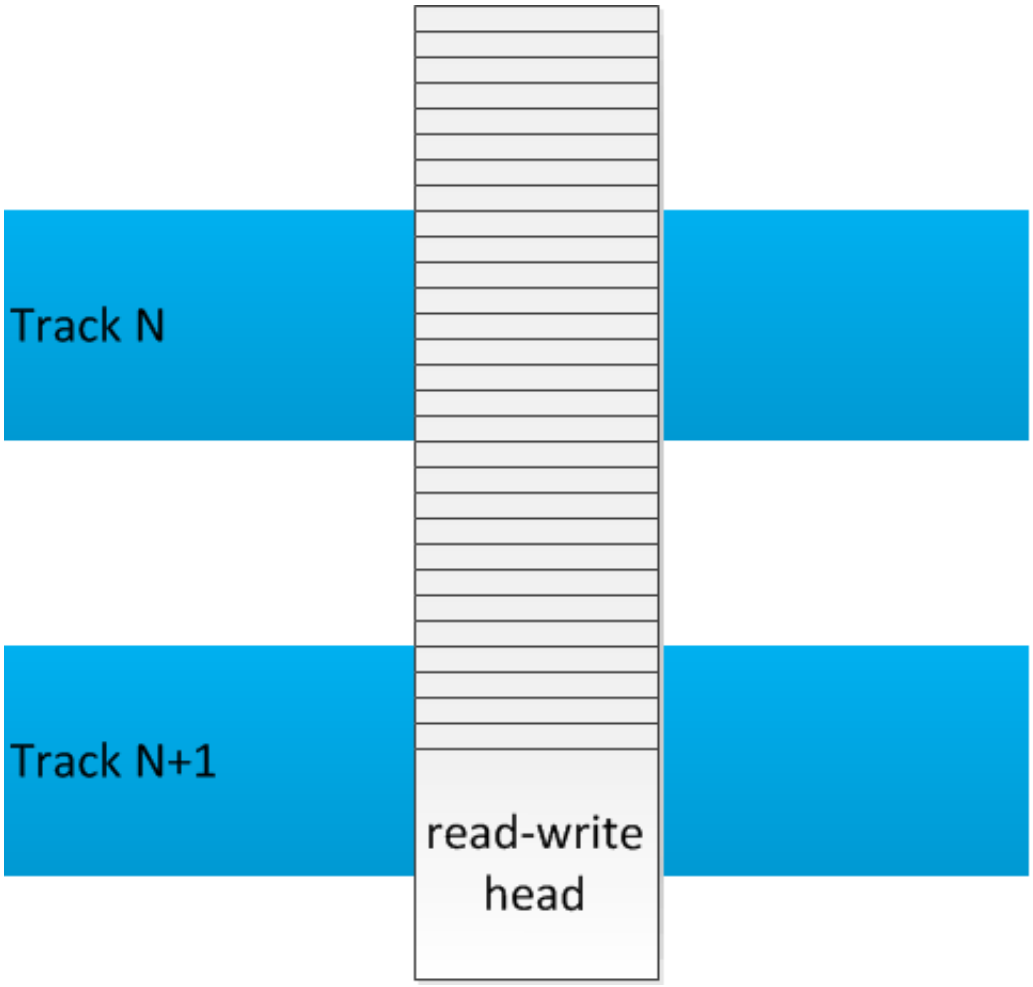


The sector data looks like this:



On to recovery

With all this, I finally had full sectors reliably recovered. In fact, I had way too many sectors: every 'scan' that Chris did involves roughly four revolutions of the platter (that is at least 3 copies of every sector) and his stepper motor stepped way finer than the track-width of the recording heads, so I had several slightly different samples of the same track.



As Chris's stepper pushed the head through the surface it gradually got closer and closer to the center of a recorder track, than passed over it, eventually crossing over the next one. The blue stripe is the actual track and its width is the width of the head. As he steps further and further away from the center of the track, the head overlaps less and less with the original recording. Still, for the scans that are fairly close to the center of the track have a reasonable signal-to-noise ratio and at least some data can be recovered from them.

So, what to do with all this data? Well, one obvious thing is to see if they are identical: if all of the successful recoveries of a sector are identical, I could be quite certain I've gotten it right.

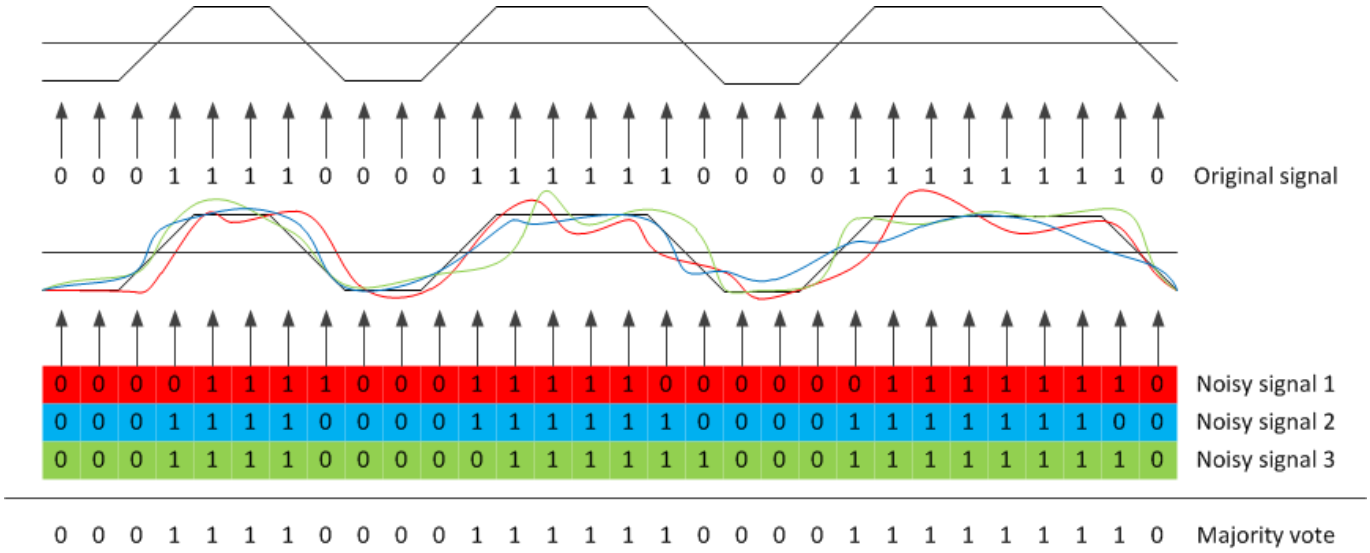
At this point, my success rate was around 90%. Pretty good, you might think, but it still means that anything significantly longer than 10 sectors would still have 'holes' – unrecovered sectors – in them. Which is pretty much everything. So I needed to do better. Much better in fact.

Looking at the failures (and their corresponding histograms) it turned out that the 'humps' are still too wide in some cases. My original code rejected samples in the middle: that is pulses that are more than +/-1 sample away from the peak of the humps. Maybe that was too strict? So my next step was to relax the rejection criteria and instead of flat-out rejecting these samples, I just assigned a low confidence value to them. If I end up with a good CRC in the end, I should be

still good, plus, I have several copies of the same sector, so I can keep the recovery with the highest confidence value. This helped somewhat – as expected – but still not good enough.

What else can we do? The next idea is to combine several samples of the same sector to increase the signal-to-noise ratio. This is when I started working on the noise hypothesis: the widened humps in the histogram are caused by noise coupled into the signal. If I take several samples of the same signal edge, the coupled noise should be different in each case. By using noise-filtering techniques I should be able to better recover the true edge.

One simple technique is to use majority voting: if I had three different samples of the same signal (with different noise on top) at least two of the samples should agree: choose the majority voted value, and hopefully, you suppress the erroneous one.



Since Chris recorder about four revolutions of the platters, in each scan we have at least three, maybe even four versions of the same sector, depending on where the recording cuts off. The problem is: we have to find the corresponding bits, in other words we have to estimate how many bit-times a full revolution takes. The way I went about it is to look at successfully recovered sectors, and if I have two instances of – say – sector 12 form the same scan, this should give me some idea of the timing of the repetitions. Of course, things aren't that simple: nobody guarantees that the successful recoveries are one revolution away: they could be two or three revolutions away as well. But with some fiddling around I have been able to come up with limits to categorize these distances.

Another problem is if more then one sector-pairs can be recovered from a scan, there are multiple estimates for the revolution size. I've used simple averaging to come up with the refined value. Not the best, but will do. By the way: another way would be use the auto-correlation of the scans to determine the revolution size, but that's very time-consuming on such large samples, so I decided to do that only if other, faster methods don't succeed.

A final problem is that the revolution size recovered this way is only an estimate, so the program

tries a few values around this estimate, each time ‘folding’ the scan back on itself and using majority voting to come up with a single digital value for each bit.

This method finally lead to success – in the expense of several hours of run-time. A total of 672 missing sectors of which 160 is in the last track, which is completely missing. This is from a total of 26336 sectors, or 98% recovery rate.

Results and the road from here

Could I do even better? Yes, I’m sure I can. There are several other things that could be improved: the revolution-size for example is not necessarily a multiple of the sample-period. This would lead to gradual shifting of the folding, making the edges actually even more spread-out than originally.

Another idea is that since there are several scans of the same track, slightly offset-ed, it should also be possible to combine those into a single scan (and fold them) greatly increasing the number of samples for each bit, further reducing the effect of noise.

It’s also possible that bandwidth-limits in the channel cause intersymbol interference which leads to additional jitter (and remember, the channel in this case involves thirty years in a garage). Tricky adaptive filtering could estimate the channel characteristics and compensate for – some of – these effects.

These techniques however have to wait: I have a good enough reconstruction of the disk image to start poking at it and I’m eager to figure out what’s on it!

For closing, here’s the result of all my labor:

The source code for the decoder:
http://www.modularcircuits.com/download/cray_files/disk_decode.zip

The input datasets (huge!!!) can be downloaded from here:
http://www.modularcircuits.com/download/cray_files/mag_photo/

Finally, you can find the reconstructed disk image here:
http://www.modularcircuits.com/download/cray_files/cos_117_disk.zip