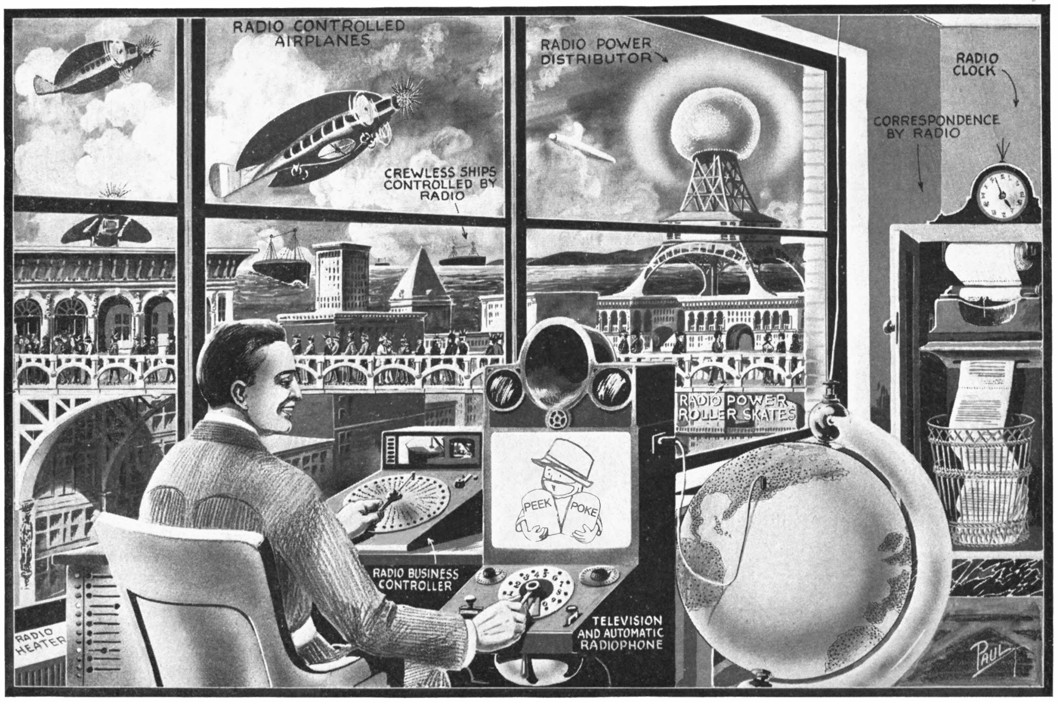
PoC || GTFO


RADIO CONTROLLED AIRPLANES
RADIO POWER DISTRIBUTOR
RADIO CLOCK
CORRESPONDENCE BY RADIO
CREWLESS SHIPS CONTROLLED BY RADIO
RADIO POWER ROLLER SKATES
PEEK
POKE
RADIO BUSINESS CONTROLLER
TELEVISION AND AUTOMATIC RADIOPHONE
RADIO HEATER
PAUL

IN THE THEATER OF LITERATE DISASSEMBLY,

PASTOR MANUL LAPHROAIG

AND HIS MERRY BAND OF

REVERSE ENGINEERS

LIFT THE WELDED HOOD FROM

THE ENGINE THAT RUNS THE WORLD!

Washington, District of Columbia

Funded by Single Malt as Midnight Oil and the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

PROPRIETARY INFORMATION OF
INFOCOM INC.
COMPANY CONFIDENTIAL

Это самиздат. He who has eyes to read, let him read!
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo10.pdf`. January 16, 2016.

Legal Note: The buying party agrees that *Pastor Manul Laphroaig and his merry band of Reverse Engineers lift the hood from the Engine That Runs the World* must be copied and shared with all neighbors, as defined by previously agreed-upon language, until the year 2104. The buying party also agrees that, at any time during the stipulated 88 year period, the seller may legally plan and attempt to execute one (1) heist or caper to steal back this issue of PoC‖GTFO, which, if successful, would return all ownership rights to the seller. Said heist or caper can only be undertaken by currently active clergy of the Church of the Weird Machines and/or neighbor Dan Kaminsky, with no legal repercussions.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo10.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`
`http://openwall.info/wiki/people/solar/pocorgtfo`

Technical Note: The polyglot file `pocorgtfo10.pdf` is valid as a PDF, as a ZIP file, and as an LSMV recording of a Tool Assisted Speedrun (TAS) that exploits Pokémon Red in a Super GameBoy on a Super NES. The result of the exploit is a chat room that plays the text of PoC‖GTFO 10:3.

Run it in LSNES with the Gambatte plugin, the Japanese version of the Super Game Boy ROM and the USA/Europe version of Pokémon Red.

```
./lsnes --library=gambatte/core.so
```

Printing Instructions: Pirate print runs of this journal are most welcome! PoC‖GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo10.pdf -o pocorgtfo10-book.pdf
```

1 Please stand; now, please be seated.

Neighbors, please join me in reading this eleventh release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our eleventh release, given on paper to the fine neighbors of Washington, D.C.

If you are missing the first ten issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth or eighth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer, the ninth in Montréal, or the tenth in Novi Sad or Stockholm.

Our sermon today, to be found on page 4, is a sordid tale in the style of a Dickensian ghost story. Pastor Laphroaig invites us to the anatomical theater, where helpless tamagotchis are disassembled in front of an audience, for *FUN*!

Page 7 contains a delightfully sophisticated and reliable exploit for Pokémon Red on the Super GameBoy, starting from a save-game glitch, then working forward through native Z80 code execution to native 65C816 code on the host Super NES. They do all of this on real hardware with scripted access to only the gamepad and the reset switch!

Keeping up our tradition of shipping in funky file formats, this PDF is a new polyglot! Page 24 contains the details for how this PDF is also an exploit, loading Pokémon Plays Twitch in the Lsnes emulator.

Micah Elizabeth Scott is becoming a regular contributor to this journal, and we eagerly await each of her submissions. Page 26 contains her notes on ARM's replacement for JTAG, called Single Wire Debug or SWD. Driving SWD from an Arduino, she's able to move the target machine like a marionette, scripted from literate HTML5 programming with powerful new elements such as `swd-hexedit`.

When we heard that Amanda Wozniak was contracted to reverse engineer a pregnancy test, but never paid for the work, we quickly scrounged up five Canadian loonies to buy the work as scrap. Page 32 contains her notes, and we'll happily pay five more loonies to the first use of this technology in a Hackaday marriage proposal or shotgun wedding.

On page 39, Peter Ferrie shares tricks for breaking the copy protection of dozens of Apple][games. When we told Peter to keep his notes to six pages, he laughed and dared us to find tricks worth cutting from his article. Accordingly, our cutting-room floor is empty and this article is the most complete collection of Apple][cracking techniques in modern publication.

Travis Goodspeed has been playing with Digital Mobile Radio (DMR) lately, a competitor to TETRA and P25 that is used for amateur radio, as well as trunked radio for businesses and cash-strapped police departments. Page 76 contains his notes for jailbreaking the Tytera MD380's bootloader, dumping all of protected memory, then patching its application to enable promiscuous mode. These tricks should also work on the CS700, CS750, and a variety of other DMR handhelds.

On page 88, the last and most important page, we pass around the collection plate. We don't need your dimes, but we'd love some nifty proofs of concept.

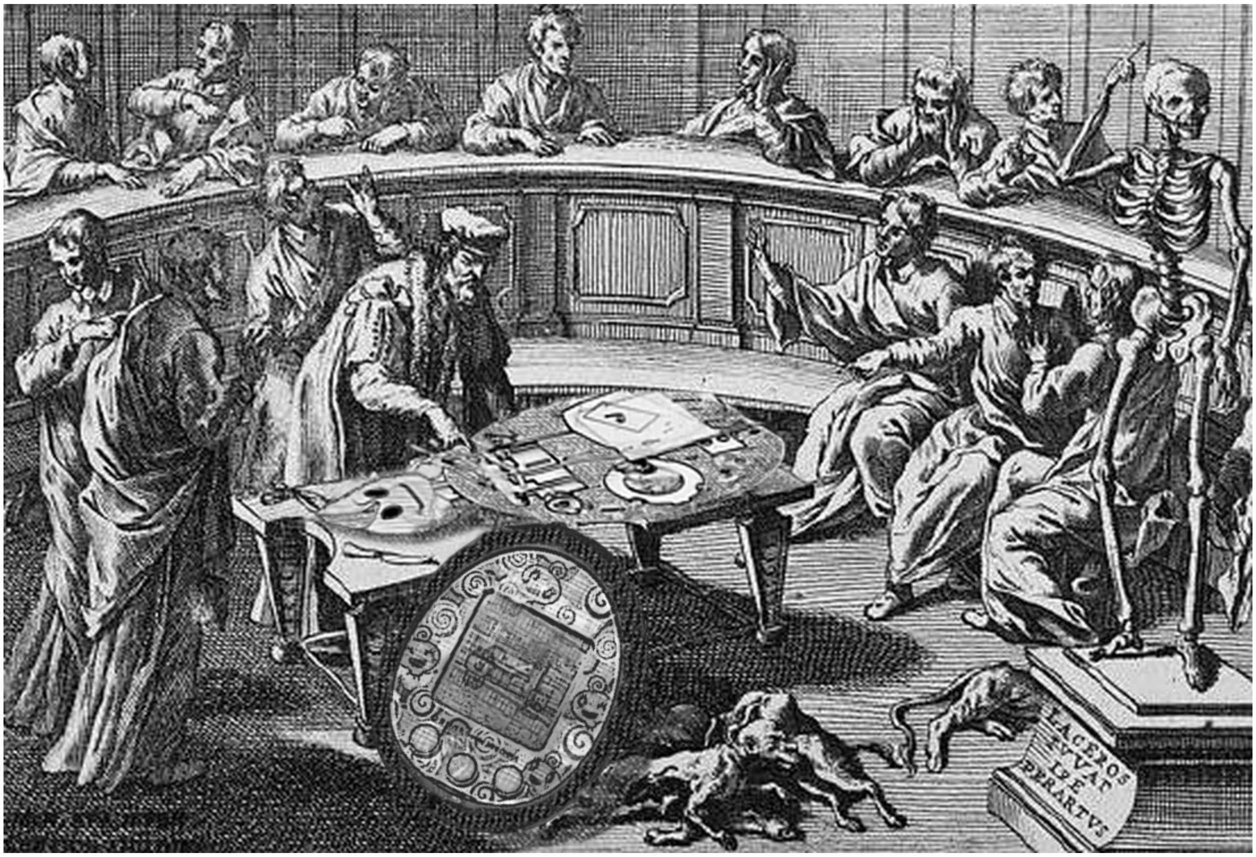
2 Three Ghosts and a Little, Brown Dog

a sermon by Pastor Manul Laphroaig

Rise, neighbors, and in the tradition of the season, let's have a conversation with spirits of the past, the present, and the future. We will head to a disreputable place, a place of controversy where, according to the best moral authorities, irresponsible people do foul things for fun—a place of scandalous, wholesale wickedness which must be stopped!

Yes, neighbors, we are heading to an *anatomical theater*, to observe its grim denizens at their grisly pastime. While some dissect carcasses, the rest watch from rows of seats. They call it learning and finding things out—even though most of what meets the eye looks like merely breaking things apart. They say they are making things better—even curing diseases!—though there are highly titled authorities with certified diplomas and ethically approved methodologies who make it their business to improve things "holistically," without all this disconcerting breakage and cutting things off. Truly, if this doesn't beg the question of "How is this allowed?" then what does?

There was a time, neighbors, when *anatomy* didn't mean trying to guess how a thing functioned by dissecting a specimen. When Andreas Vesalius published his classic human anatomy atlas with its absolute priority of dissection for learning what was and what was not true about the human body, his fixation on biological disassembly was a scandal. A proper anatomy book was understood to include Aristotle's four humors and a fair bit of astrology; imagine how regressive Vesalius' fixation on cutting things apart to find their function must have looked! Even when he became a royal court physician, other learned physicians called him a barber—for everyone knew that only barbers and sawbones used blades. Until Victorian times, a doctor was a gentleman,



and a surgeon wasn't. Testing the patient's urine was fine, but taking knives to one was simply below a proper doctor's station.

Vesalius' dissection-bound atlas became an instant hit, though. It turned out that going into specific techniques of dissection—place a rope here and a pulley there—so that others would replicate it was exactly what was needed; the venerable signs and elements, on the other hand, not so much. Which did not save Vesalius from having to undertake an emergency trip to far-away lands for an obscure reason, dying in abject poverty on the way. He died before the first dedicated anatomical theater was built in 1594, by which time anatomy finally meant what he had made it mean.

Ah, but that was then and now is now! The year is 1902, and *physiology* is the latest scandal. Again, moral delinquents and their supporters are doing something loathsome: vivisection. Again, they come up with excuses: it's all about finding out how things work, they say; some kind of knowledge that makes them different from the uninitiated, we hear. And even if there was knowledge to be gained, could it really be trusted to such an immature and irresponsible crowd? Stuck to their—not so innocent—toys and narrowly focused views, they can't even see the bigger ethical picture! They cater to and are occasionally catered by truly objectionable characters—and then have the gall to shrug it off. They talk about education, but who in their right mind would let them near children? Too bad there isn't a general law against them yet, and the establishment is dragging its feet (or even has its own uses for them, no doubt disgusting)—but the stride of social progress is catching up with them, and, with luck, there soon will be!

That was the year of high court drama, a pitched battle between people who each believed to embody "social progress" against "superstition" on both sides. It saw rallies by socialists and riots by medical students, scientists and suffragettes, British lords and Swedish feminists—and a lot more, including its own commemorative handkerchief merchandise. It is immortalized in history as *The Brown Dog affair*, one so dramatic that even the Wikipedia article about it makes for good reading. Incidentally, the experiment involved led to the discovery of hormones.

So says the Ghost of Science Past, but we bid him to haunt us no longer. There is another, more cheerful Spirit to occupy our attention—the Spirit of the Present. This is a more cheerful Spirit, involving pets only as cute pictures thereof—and lots of them!—much to the relief of those who think neither Schrödinger nor Pavlov would make good friends.

But this Spirit isn't left without attention from our moral betters. What about the children? What about the lowlives and the criminals whom we empower by our so-called knowledge? What about the bullies, the haters, the thieves, the spies, the despots, and even—the terrorists? Would a good thing be called *exploitation* or *pwnage*? This new reality is so scary to some people that their response goes straight to nuclear; they call for a *Manhattan project*, but what they really mean is "nuke it from orbit." To some, it's even about evil "techno-priests" hijacking "true social progress"—or at least it sells their books.

Nor is this Spirit's domain devoid of court drama, even in our enlightened times—although looking where we tend to fall on the scale between Vesalius and Lord Alverstone's Old Bailey, one begins to wonder just where the light is going. No wonder the Spirit of the Hacking Present looks somewhat frayed around the edges.

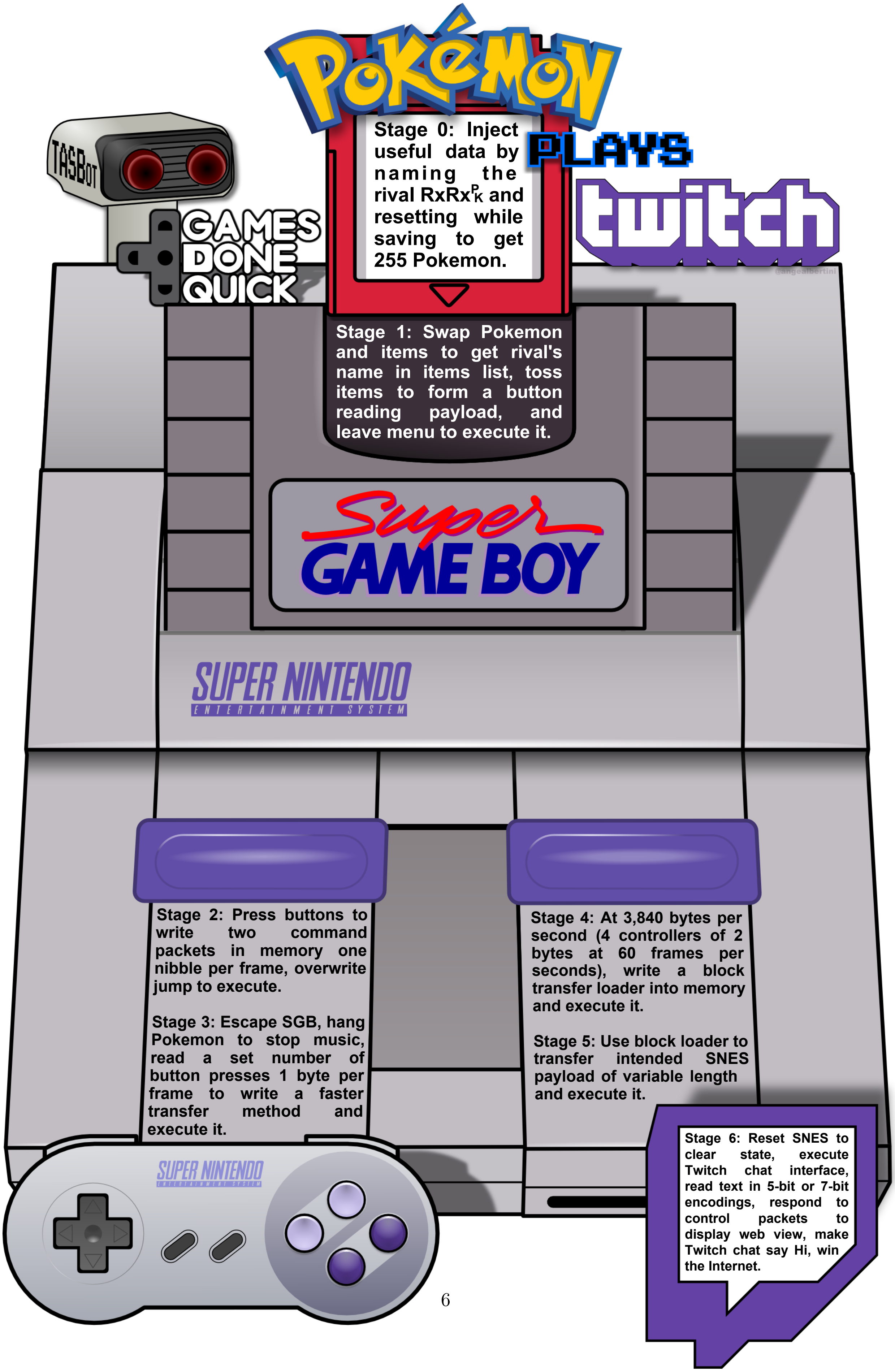
Why wait for the Specter of the Future to make an appearance? I say, neighbors, let's make like 1594 at the University of Padua—back when a university used to have quite a different place in this game of ghosts—and have our own Anatomical Theater, a Theater of Literate Disassembly!

Just as Knuth described Adventure with Literate Programming,[1] we'll weave together the disassembled code of a live subject with expert explanations of its deeper meaning. (Of course the best part might well be a one liner, but we'll save the reader hours of effort!) We'll weave a log and a transcript into an executable script that reproduces the cuts of a Master Surgeon, stroke by stroke.

It is high time. We have been doing our dissections alone—with none or few to watch and learn—long enough. Let other neighbors watch your disassembly, show them your technique, and let them get a good view and share the fun.

As the good old U. of Padua preserved its theater, so shall we! And then perhaps the Specter of the Future will smile upon us.

[1] `unzip pocorgtfo10.pdf adventure.pdf`


Pokémon
PLAYS
twitch
@angealbertini
TASBot
GAMES DONE QUICK
Super GAME BOY
SUPER NINTENDO ENTERTAINMENT SYSTEM
SUPER NINTENDO ENTERTAINMENT SYSTEM

Stage 0: Inject useful data by naming the rival RxRx^P_K and resetting while saving to get 255 Pokemon.

Stage 1: Swap Pokemon and items to get rival's name in items list, toss items to form a button reading payload, and leave menu to execute it.

Stage 2: Press buttons to write two command packets in memory one nibble per frame, overwrite jump to execute.

Stage 3: Escape SGB, hang Pokemon to stop music, read a set number of button presses 1 byte per frame to write a faster transfer method and execute it.

Stage 4: At 3,840 bytes per second (4 controllers of 2 bytes at 60 frames per seconds), write a block transfer loader into memory and execute it.

Stage 5: Use block loader to transfer intended SNES payload of variable length and execute it.

Stage 6: Reset SNES to clear state, execute Twitch chat interface, read text in 5-bit or 7-bit encodings, respond to control packets to display web view, make Twitch chat say Hi, win the Internet.

3 Pokémon Plays Twitch

by Allan Cecil (dwangoAC), Ilari Liusvaara (Ilari) and Jordan Potter (p4plus2)

For the Awesome Games Done Quick (AGDQ) 2015 charity marathon we exploited a chain of unmodified Nintendo game console components consisting of a Pokémon Red Game Boy cartridge in a Super Game Boy running in a Super Nintendo. We plugged the latter into custom hardware posing as a normal controller. In this *seven*-stage exploit, we corrupted a save file to give ourselves 255 Pokémon, swapped Pokémon, and tossed items to plant shellcode. We committed a series of atrocities using documented command packets and ultimately broke into the Super Nintendo's working RAM, where we wrote our own chat interface to display live contents of the Twitch chat and even a representation of a defaced website.

3.1 TAS'ing a Game to execute Arbitrary Code

TASVideos[2] hosts Tool-Assisted Speedruns of games that are created using an emulator with speed controls such as slow motion and frame advance, along with the ability to save and restore the state of the game (or, rather, of the entire console) at any time. TAS movie files consist of a list all of the button presses sent to the console every frame from the time it is powered on until the game is beaten. It aids our poor human reflexes, but it can do a lot more—like arbitrary code execution!

The first run on the site to use this ability to execute arbitrary code to jump to the credits of a game was Masterjun's Super Mario World run. Later, Bortreb used arbitrary code execution in a run of Pokémon Yellow, marking the first time external content was added to an existing game.

In late 2013, dwangoAC worked with Ilari and Masterjun to present a run at AGDQ 2014 that programmed the games Snake and Pong into Super Mario World on a real console using a replay device (also known as a "bot") from True.[3] This was a huge success and was covered by Ars Technica, but we knew that we could do even more, which ultimately led us to the project described in this article.[4]

3.2 The Game Choice

We started with an end-goal of executing arbitrary code on a Super Nintendo (SNES) using a Super Game Boy (SGB) cartridge as the entry point. We initially planned to use Pokémon Yellow based on Bortreb's exploit in that game, but quickly discovered that the SGB detection routine used by Pokémon Yellow is extremely broken and only worked on a real SGB by pure chance.[5] After looking at other options, we chose to use the Pokémon Red version, which uses a more reliable SGB detection routine that gets us access to the full SGB palette, a custom border, and consistent timing benefits we'll discuss later.[6] Using Pokémon

[2] `http://tasvideos.org`

[3] `http://truecontrol.org`

[4] It should also be noted that all recent AGDQ events have directly benefited the Prevent Cancer Foundation which was a huge motivator for several of us who worked on this project. The block we presented this exploit in at AGDQ 2015 helped raise over $50K and the marathon as a whole raised more than $1.5M toward cancer research, making this project a huge success on multiple levels.

[5] In brief, the detection routine is extremely sensitive to how many DMG clock cycles various operations take; the emulator is likely slightly inaccurate, which causes the detection to fail, but from looking at the behavior it seems like it "just works" on the real hardware. This is sheer luck, and the game developers likely never even knew it was so fragile.

[6] If the SGB BIOS does not find the special codes in the DMG game header that indicate it's an SGB-enabled game (`$146` equal to `$03`), it locks up the command channel until the game is reset, rendering any SGB based exploitation impossible. See `http://gbdev.gg8.se/wiki/articles/The_Cartridge_Header` for more details.

Red also has another added benefit in that the entire game has been skillfully disassembled.[7]

3.3 The Emulator

When we started this project in August 2014, the only emulator capable of emulating an SGB inside of an SNES at a low enough level for our needs was the BSNES emulator. Unfortunately, although BSNES is very accurate at emulating an SNES, it doesn't do a very good job of emulating an SGB. The Gambatte Dot-Matrix Game Boy (DMG) emulator is likewise very accurate, but is unable to emulate an SGB on its own. Ilari was able to create a hybrid emulation core using BSNES to emulate the SNES↔DMG interface chip while using Gambatte for DMG emulation. This was a considerable undertaking, but in the end the emulator was very usable, albeit somewhat slow, as properly emulating the synchronization between the SNES CPU and the DMG CPU is a challenge. Ilari continued to provide emulator development and scripting support throughout the project.

3.4 The Hardware

We didn't just want to exploit a game in the sandbox of a console emulator and call it a Proof of Concept. We wanted to do the job properly and create an actual exploit that would work on real hardware. Only one member of our team (dwangoAC) had all of the required hardware in one place, namely a SNES console, a SGB cartridge, a copy of Pokémon Red, and the replay device posing as a controller, also known as a "bot."[8] Because we wanted to stream data from an attached computer, we opted to use an older, serial-over-USB connected device, namely True's NES/SNES Replay Device. This choice of hardware had a few limitations but worked out well for the project in the end.

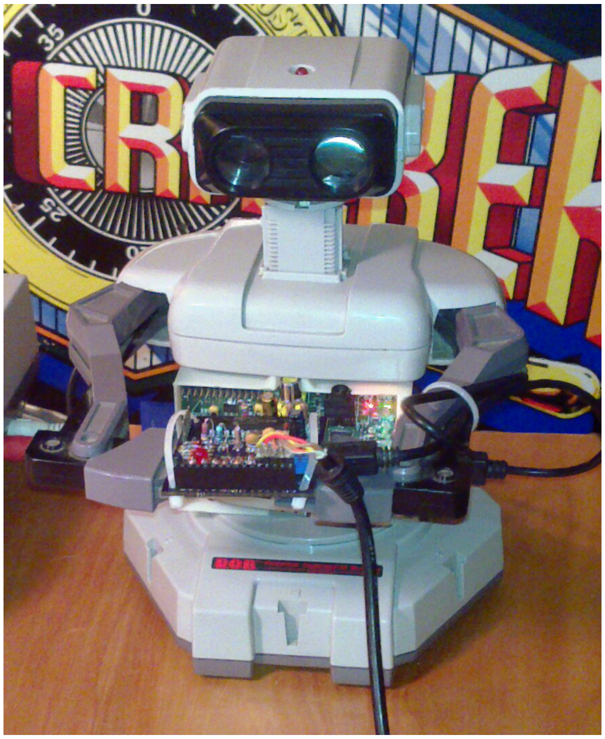


Figure 1 – The legendary TASBot

3.5 The Plan

We were initially unsure what kind of payload to create once we had gained the ability to execute arbitrary code on the SNES. Initially we investigated methods of showing crude video, but abandoned it after spending far too much time failing to increase the datarate and running into limits with the processing speed of the SNES's 65C816 CPU. An IRC discussion about Twitch Plays Pokémon[9] led dwangoAC and p4plus2 to brainstorm what it would take to incorporate similar elements into our payload, and the concept of *Pokémon Plays Twitch* was hatched—where a Pokémon character enters a Twitch chat room and "plays" Twitch. In the end, we took it to the next level by giving Red a voice in a chat interface on the SNES and giving TASBot, the robot holding the replay board, the ability to speak through *espeak* and argue with Red. There's much more to say about that, but let's first get to the point where we can execute arbitrary code!

[7] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/pokered-master.zip`

[8] The term "bot" was originally used because it's as if you have a robot playing the game for you; dwangoAC later attached one of these replay devices to a R.O.B. robot as shown in Figure 1 and after presenting Pong and Snake on SMW, the name TASBot came to be associated with the combination as described at `http://tasvideos.org/TASBot`.

[9] A way of crowdsourcing gameplay by parsing commands sent over IRC.

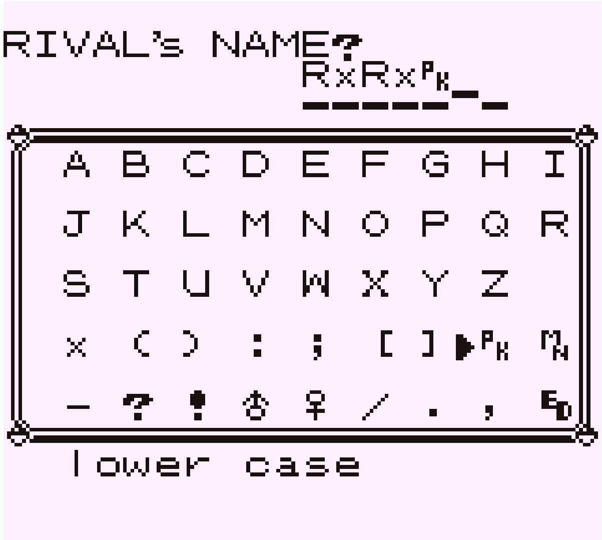


Figure 2 – A strange rival

3.6 Stage 0: Corrupting a save game.

(3–7 bytes per minute.)

We start the game by creating a save file, giving ourselves the default name of Red and naming our rival RxRxPK as shown in Figure 2. We then save the game as in Figure 3, but reset the console directly after it starts writing to the cartridge's SRAM. There is checksumming on most of the values in the save file but at least one value has no checksum at all, namely the byte at the start of the "party data" that stores the number of Pokémon that have been caught. By some chance, this value in SRAM (at `0xAF2C`, or `0x2F2C` when paged) is initially set to `FF`, so we wait long enough for valid name data and a save file header to be written before resetting. It is possible to do this with human reflexes as the window is approximately 20 ms but we opted to run a wire from our replay device to pin 19 on the expansion port on the underside of the SNES. This allowed us to trigger a reset by shorting the pin to ground, as shown in Figure 3.[10]

3.7 Stage 1: Writing Z80 assembly by swapping Pokémon and tossing items.

(30 bytes per second.)

After loading the game but before changing anything, the initial state of the GBBUS memory map is as follows:[11]

```
0xD163  Number of Pokemon caught,
        corrupted to 0xFF in Stage 0.
0xD164  Pokemon IDs (1 byte each),
        corrupted to 0xFF.
0xD16A  Sentinel byte (0xFF)
0xD16B  Pokemon Data (44 bytes each);
        all are corrupted to 0xFF.
0xD273  Pokemon original trainers;
        all are corrupted to 0xFF.
0xD2B5  Pokemon nicknames;
        all are corrupted to 0xFF.
0xD2F7  Pokemon owned bitmap (19 bytes);
        all zeroes.
0xD30A  Pokemon seen bitmap (19 bytes);
        all zeroes.
0xD31D  Number of items; initially 0
0xD31E  Items array; each entry is 2 bytes,
        an item ID and item count.
        After the last item, there is an FF.
        (Initially located at 0xD31E.)
0xD347  Money as Binary-Coded Decimal.
        (Initially 00 30 00, $3000.)
0xD34A  Rival's name. (Set during Stage 0,
        initially
        91 F1 91 F1 E1 50 00 00 00 00 00.)
0xD355  <misc data>
0xD36E  Map level script pointer.
        (Initially B0 40.)
```

We want to adjust some of these values to create a payload described in the next section, and the game conveniently provides three ways to arrange the state of memory.

- Swapping Pokémon: The game implements moving Pokémon around the list by swapping the raw contents of entries in the ID, Data, Original trainer, and nickname tables, which happens to offset data by an odd amount. Since we have 255 Pokémon instead of the 141 the game was originally limited to we can swap

[10] As with many exploits, the seed for this came from Bortreb's Pokémon Yellow exploit, which itself came from earlier discoveries of others. Masterjun adapted the exploit for Pokémon Red using the BizHawk DMG emulator and dwangoAC took this information and made the Stage 0 and Stage 1 movie file in LSNES.

[11] The same values can be found in the GBWRAM region at an offset of `-0xC000`, so the value for `0xD163` in GBBUS (which isn't visible in the LSNES memory editor) can instead be found at `0x1163` in GBWRAM. GBBUS addressing is used throughout for consistency with existing resources such as the pokered disassembly.

[12] This means the Pokémon data now extends past end of WRAM, and in fact the WRAM should in effect loop around, although this isn't used.

around the contents of anything in WRAM above `0xD164`.[12]

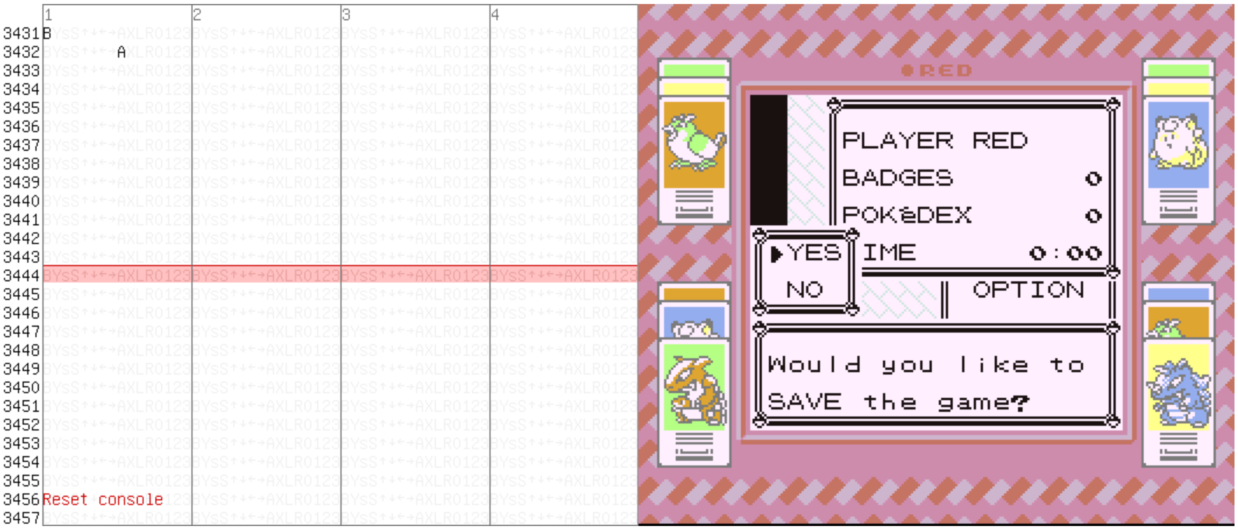
- Tossing items: Throwing away unwanted items decrements the second byte in an item's two-byte ID / Quantity pair. Unfortunately, there are some items that can't be tossed, either because the game prevents tossing them or because doing so softlocks or crashes the game.
- Swapping items: Items can be swapped around in the list of items, which normally just swaps the item data. If you swap two of the same item, the game tries to merge them by adding their counts and then shifting the item list. The shift adjusts the item count and writes a new sentinel item ID. (It doesn't touch either the item count in that slot or the old sentinel.)

Since we don't have any items, let's get some! Swapping the first Pokémon with the tenth causes the `FF`'s located at `0xD16B` through `0xD196` to be written to `0xD2F7` through `0xD322`. Per the memory map, the number of items is located at `0xD31D` and this is changed along with lots of other nearby addresses from `00` to `FF`, which causes the game to think we have 255 items. We eventually enter the item menu and proceed to toss a number of safe items, but—because we can only ever decrement the quantity byte in each item's ID/Quantity two-byte pair—we have to go back and swap Pokémon to make what was once an ID into an item count and vice versa.

In order to avoid softlocking the game, we have to walk through the sequence in a very particular order. There are several items that the game refuses to toss, some that crash the game if you try to toss them, some that can only be thrown once—after which all items afflicted with this condition can no longer be tossed. Some will crash the game simply by being in the menu even if you never even select them.

To work around these pitfalls, we prepare memory by doing several Pokémon and item swaps followed by an initial round of tossing, we go back to swap Pokémon in a way that realigns memory so we can now toss what used to be item IDs. We swap several Pokémon to relocate the Stage 1 code and create a swath of 0's in front of it, and at the very end we swap two identical items to shift memory two spaces back. That's a lot to take in in one sentence, so Figure 4 diagrams the complete list of changes we make showing the value changes as anchored initially from GBBUS `0xD349`.

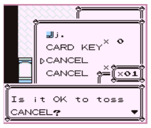
The primary purpose of all this swapping and tossing is to create a better way to craft our own



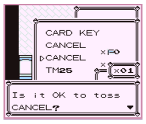
Would you like to SAVE the game?

Figure 3 – Corrupting a save game by pressing A to save, then resetting 24 frames later.

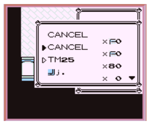
[13] The swap where j. is swapped with j. involves the pairs `00 00` and `00 F4`, but they turn into `00 63` and `00 91` because we abuse the fact that the game assumes a quantity of `00` is the same as `FF` and you can only have ninety-nine of any given item in one slot. This results in `FF` + `F4` = `1F3` which is larger than the sum mod 256 dec., at which point the game stores 63 in one

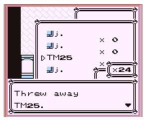


Is it OK to toss CANCEL?

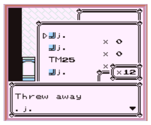


Is it OK to toss CANCEL?

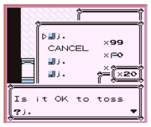




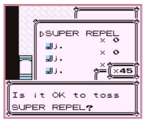
Threw away TM25.



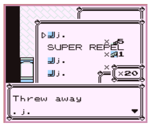
Threw away . j.



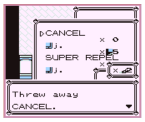
Is it OK to toss ?j.



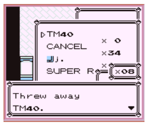
Is it OK to toss SUPER REPEL?



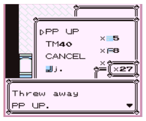
Threw away . j.



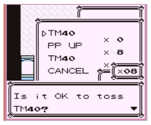
Threw away CANCEL.



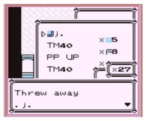
Threw away TM40.



Threw away PP UP.



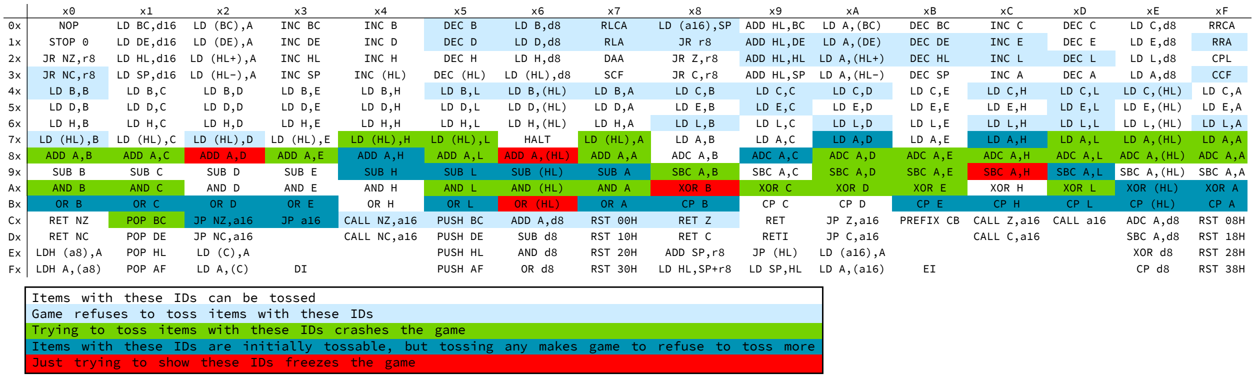
Is it OK to toss TM40?



Threw away . j.

```
     Address ## ID ## ID ## ID ## ID ## ID ## ID ## ID

     0xD34A  00 91 F1 91 F1 E1 50 00 00 00 00 00 00 00
Pokemon 1 ↔ 10 ⇒datastart = 0xD349
item    3 ↔ 5  ⇒datastart = 0xD347
Pokemon 3 ↔ 6  ⇒datastart = 0xD331
item    3 ↔ 4  ⇒datastart = 0xD32F
     0xD32F  00 91 F1 91 F1 E1 50 00 00 00 00 00 00 00
              toss 1 item
     0xD32F  00 91 F0 91 F1 E1 50 00 00 00 00 00 00 00
                    toss 1 item
     0xD32F  00 91 F0 91 F0 E1 50 00 00 00 00 00 00 00

     0xD32F  00 91 F0 E1 50 91 F0 00 00 00 00 00 00 00

     0xD32F  00 91 F0 00 00 91 F0 00 00 00 00 E1 50 00
                                            toss 24 items
     0xD32F  00 91 F0 00 00 91 F0 00 00 00 00 E1 38 00
                               toss 12 items
     0xD32F  00 91 F0 00 00 91 F0 00 F4 00 00 E1 38 00
(same ID swap)
     0xD32F  00 91 F0 00 63 91 F0 00 91 00 00 E1 38 00
                   toss 20 items
     0xD32F  00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 00
Pokemon   4 ↔  5 ⇒datastart = 0xD324
(even address, so now ID and ## are shifted)
     Address ID ## ID ## ID ## ID ## ID ## ID ## ID ##

     0xD324  00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 00
                                               toss 45 items
     0xD324  00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 D3
                                         toss 20 items
     0xD324  00 91 F0 00 4F 91 F0 00 91 00 00 CD 38 D3
                                  toss 222 items
     0xD324  00 91 F0 00 4F 91 F0 00 91 22 00 CD 38 D3
                             toss 8 items
     0xD324  00 91 F0 00 4F 91 F0 F8 91 22 00 CD 38 D3
                      toss 27 items
     0xD324  00 91 F0 00 4F 76 F0 F8 91 22 00 CD 38 D3
                 toss 8 items
     0xD324  00 91 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
          toss 27 items
     0xD324  00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
Pokemon  -8 ↔ -7   ⇒datastart = 0xD350
Pokemon   3 ↔  4   ⇒[0xD35B] = 00
Pokemon   4 ↔  5   ⇒[0xD366] = 00
Pokemon   5 ↔  2   ⇒datastart = 0xD366
Pokemon   2 ↔ -11  ⇒[0xD2CC] = 00
Pokemon -11 ↔ -9   ⇒[0xD32E] = 00
item      4 ↔  5   ⇒datastart = 0xD362
     0xD362  00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
```

Figure 4 – Pokémon and items are re-arranged in memory to create shellcode.



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	🟦 DEC B	🟦 LD B,d8	🟦 RLCA	🟦 LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	🟦 DEC D	🟦 LD D,d8	🟦 RLA	🟦 JR r8	🟦 ADD HL,DE	🟦 LD A,(DE)	🟦 DEC DE	🟦 INC E	DEC E	LD E,d8	🟦 RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	🟦 ADD HL,HL	🟦 LD A,(HL+)	🟦 DEC HL	🟦 INC L	🟦 DEC L	LD L,d8	CPL
3x	🟦 JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	🟦 CCF
4x	🟦 LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	🟦 LD B,L	🟦 LD B,(HL)	🟦 LD B,A	🟦 LD C,B	🟦 LD C,C	🟦 LD C,D	LD C,E	🟦 LD C,H	🟦 LD C,L	🟦 LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	🟦 LD E,C	LD E,D	LD E,E	LD E,H	🟦 LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	🟦 LD L,B	LD L,C	🟦 LD L,D	LD L,E	🟦 LD L,H	🟦 LD L,L	LD L,(HL)	🟦 LD L,A
7x	🟦 LD (HL),B	LD (HL),C	🟦 LD (HL),D	LD (HL),E	🟩 LD (HL),H	🟩 LD (HL),L	HALT	🟩 LD (HL),A	LD A,B	LD A,C	🔵 LD A,D	LD A,E	🔵 LD A,H	🟩 LD A,L	🟩 LD A,(HL)	🟩 LD A,A
8x	🟩 ADD A,B	🟩 ADD A,C	🟥 ADD A,D	🟩 ADD A,E	🔵 ADD A,H	🟩 ADD A,L	🟥 ADD A,(HL)	🟩 ADD A,A	ADC A,B	🔵 ADC A,C	🟩 ADC A,D	🟩 ADC A,E	🟩 ADC A,H	🟩 ADC A,L	🟩 ADC A,(HL)	🟩 ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	🔵 SUB H	🔵 SUB L	🔵 SUB (HL)	🔵 SUB A	🟩 SBC A,B	SBC A,C	🟩 SBC A,D	🟩 SBC A,E	🟥 SBC A,H	🔵 SBC A,L	SBC A,(HL)	SBC A,A
Ax	🟩 AND B	🟩 AND C	AND D	AND E	AND H	🟩 AND L	🟩 AND (HL)	🟩 AND A	🟥 XOR B	🟩 XOR C	🟩 XOR D	🟩 XOR E	XOR H	🟩 XOR L	🔵 XOR (HL)	🔵 XOR A
Bx	🔵 OR B	🔵 OR C	🔵 OR D	🔵 OR E	OR H	🔵 OR L	🟥 OR (HL)	🔵 OR A	🔵 CP B	CP C	CP D	🔵 CP E	🔵 CP H	🔵 CP L	🔵 CP (HL)	🔵 CP A
Cx	RET NZ	🟩 POP BC	🔵 JP NZ,a16	🔵 JP a16	🟦 CALL NZ,a16	🟦 PUSH BC	🟦 ADD A,d8	🟦 RST 00H	🟦 RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

Items with these IDs can be tossed
🟦 Game refuses to toss items with these IDs
🟩 Trying to toss items with these IDs crashes the game
🔵 Items with these IDs are initially tossable, but tossing any makes game to refuse to toss more
🟥 Just trying to show these IDs freezes the game

Figure 5 – Item IDs can double as Z80 opcodes.

code—as it would be quite tedious to use this method to do anything longer.[13] Here's a disassembly of what we've been able to write so far, starting from 0xD361.

```
  0xD362   00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
                  ⇓
        LR35902 shellcode at 0xD361:
          30 00      JR NC,0        // nop
Player's   76         HALT           // wait for frame
starting money F0 F8      LDH A, (0xF8)  // load input
          4F         LD C,A         // save in C
          76         HALT           // wait for frame
          F0 F8      LDH A, (0xF8)  // load input
          91         SUB C          // decode opcode
          22         LD (HL+),A     // stage2[HL++] = A
          00         NOP
          CD 38 D3   CALL 0xD338    // call stage2
```

Everything up to this point has been the process of writing Stage 1, but now it's time to walk through executing it, although some of the shortcuts we took require a bit of explanation.

First, the reason 0xD361 contains 30 is because the beginning of the Stage 1 data is mostly copied from the field that holds the rival name—which happens to be directly preceded by the player's money. Of this quantity we see the last two out of three bytes represented here in BCD format; the full value 00 30 00 starts at 0xD360. It would take extra effort to eliminate the 30 00 portion, but because that sequence is effectively a NOP, we leave it be.

To reduce the number of bytes that needed to be modified, we used several clever tricks. The code that jumps to this point sets HL to the jump target address, and HL is a canonical pointer register that can be written to. We reused 0xD36E (the map level script pointer) as the loop jump address; upon exiting the menu, the map level script pointer is loaded and called, so it loads the value in 0xD36E into HL and jumps to it.

```
1041   LD  HL,  0xD36E
1044   LD  A, (HL+)
1045   LD  H, (HL)
1046   LD  L ,A
1047   LD  DE,  0x104C
104A   PUSH DE
104B   JP  (HL)  ;  [D36E]
```

Stage 1's purpose is to read the buttons being held down on the controller and write them somewhere, eventually executing what we've written using this slightly more efficient method than twiddling with Pokémon and items. At a high level, this code will read a byte from the controller on one frame, read another byte from the controller on the next frame, subtract the two, store the result at a given memory offset and repeat, successively storing values one byte at a time in order in memory, and ultimately executing said bytes.

The game's NMI (Non-Maskable Interrupt) routine writes a bitmap of the current buttons being held down during each frame (mapped as the buttons ABsSRLUD from lowest to highest bit) to 0xFFF8, and HALT is used to wait for the next frame. Unfortunately, the SGB BIOS cancels out LEFT+RIGHT or UP+DOWN if they are pressed simultaneously and instead converts those bits to 0's. To work around it, our short routine reads two frames and combines the values in a way that can yield arbitrary bytes. Because of restrictions on

item and $190 \mathit{mod} \mathtt{FF} = \mathtt{91}$ is stored as the remainder in the other.

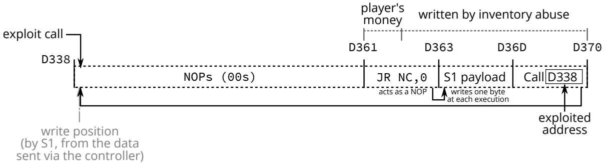
[14] There is no working way to ADD the two reads because we can't write that opcode. Due to byte restrictions, we can't use JP either, but CALL is close enough. See Figure 5.

which bytes we can create, we use LD C,A to store the first value and then SUB C to combine them.[14]

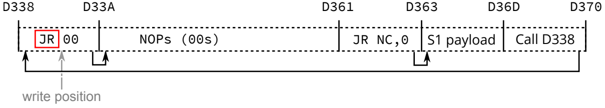
Using this method, we write the following data to 0xD338, which is executed every frame; that is to say, it is repeatedly executed even before it is fully written!

```
18 27 <= first jump
3E 1C CD AF 00 21 4D D3 CD EB 5F 2E 58 00 CD
     EB 5F 18 FE 79 00 18 00 06 AD 12 42 30
    FB 40 91 18 42 00 00 18 00 00 00 <=
    Stage 2 payload
18 D7 <= second jump
```

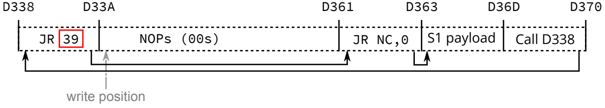
The memory range from 0xD338 to D360 contains only 00's and forms a cascade of harmless NOP instructions. This is critical, because this entire section is executed every time a byte is written; this also means we have to be extremely careful with what we write, to avoid executing an incomplete Stage 2 that causes a crash. The solution is to write a jump instruction of 18 27 into the first two bytes—which will skip execution of Stage 2 while it is being constructed. The sequence 18 27 can't be entered in one frame, but fortunately the incomplete form, 18 00, is effectively a NOP instruction. This gives us the full range from 0xD33A to 0xD360 where we can write whatever we want with impunity, and HL points to 0xD33A.


player's money
written by inventory abuse
exploit call
D338
D361
D363
D36D
D370
NOPs (00s)
JR NC,0
S1 payload
Call D338
acts as a NOP
writes one byte at each execution
exploited address
write position (by S1, from the data sent via the controller)

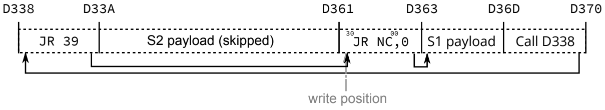
We write 0x18 (JR x) into current write position and advance write position:



We write 0x27 into current write position, turning the first instruction into a nontrivial jump.

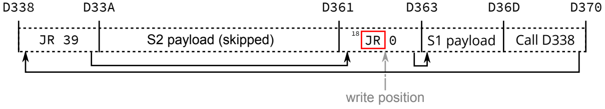


We write the Second Stage to D33A–D360 which is jumped over and not executed. This takes 39 iterations through the loop.



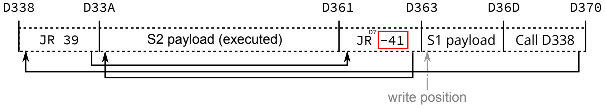
After this, we somehow need to jump to the newly completed Stage 2. The HL now points to 0xD360 and the next byte we poke is 18, which turns the first instruction in the Stage 1 code into JR 0, which is still effectively a NOP.

We write 18 (JR x) to current position, turning the 30 into 18, acting as a JR 0 instruction.



We write D7 into 0xD362, which modifies the instruction to be JR -41, which jumps to 0xD33A, the start of the second payload. After one more call into 0xD338 and the subsequent jump to 0xD360, the execution jumps to the Second Stage.

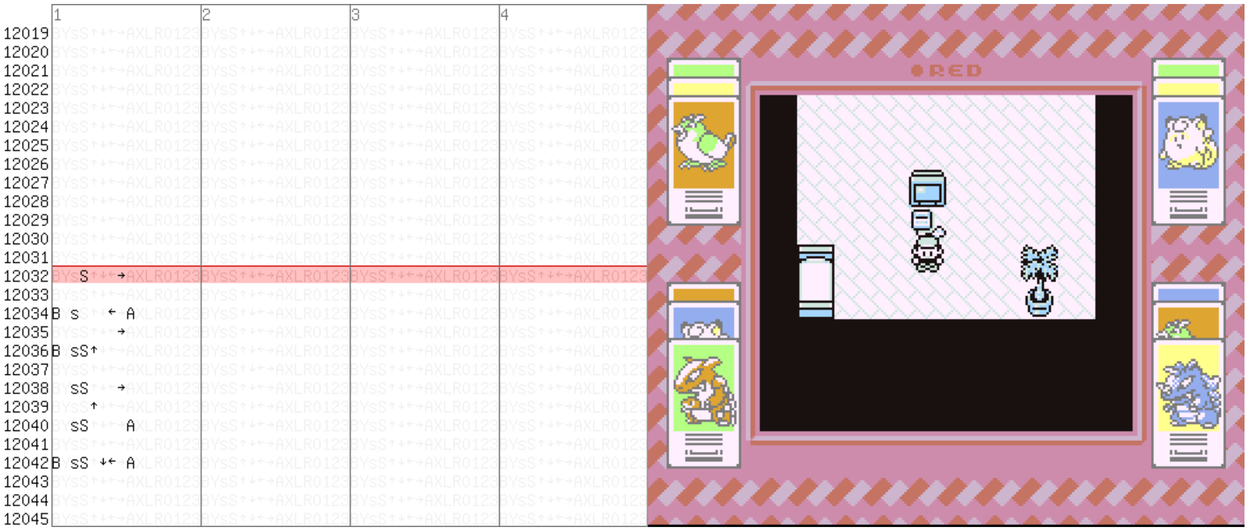
We write D7 (-41) to current position, turning the jump into a jump to execute the Stage 2:



One last note before moving on to what Stage 2 will do for us: as with most things in this exploit, entering the Stage 2 payload isn't as straightforward as it should be, and this time it's because the SNES and the DMG run at different clock speeds and framerates. It takes 351,120 cycles for the DMG to run one frame, and 357,366 for the SNES to run one frame. Each side polls the inputs once per their frame, and the SNES side updates the inputs that the DMG side reads once per frame. Since each SNES frame takes slightly longer, the SNES regularly skips updating inputs for one full DMG frame, causing the input to be duplicated.[15]

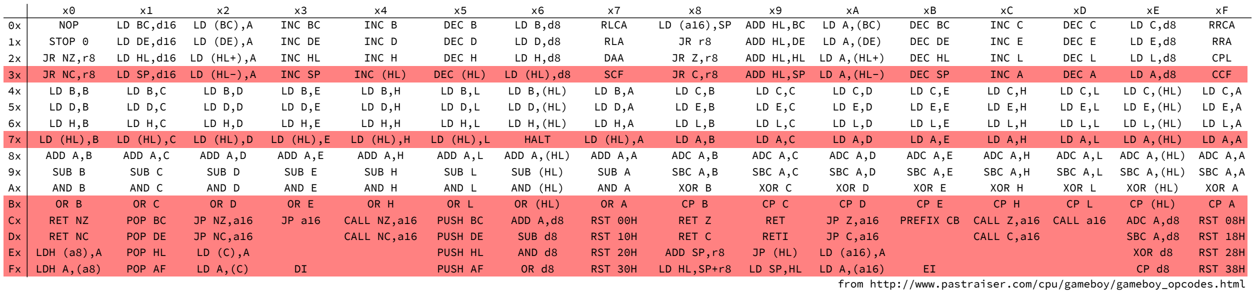
This clock skew slip happens about every 56 DMG frames. (Sometimes it's 57 frames between slips due to slipping.) It takes a full 86 frames to input the Stage 2 sequence because there are 39 bytes of payload plus 4 bytes total for prologue and epilogue jump instructions, and each byte takes 2 frames to enter as a result of working around L+R and U+D combinations being nulled out. This means we have to cope with at least one clock skew slip and because 86 isn't that much bigger than 2*56

[15] This has implications even outside of TAS'ing: If you hold a button for a single frame you risk that input being lost (if the previous frame had no buttons being pressed, that single frame's press could be overwritten with the no buttons pressed frame from before) or your buttons could be held for an extra frame (even though you let go, you hit right before the skew so your buttons are sent for an additional frame). Both of these behaviors could cause a talented realtime player to question their abilities as they wouldn't have any idea that the console had been the cause of their input being wrong.



```
      1                  2                  3                  4
12019
12020
12021
12022
12023
12024
12025
12026
12027
12028
12029
12030
12031
12032    S   →
12033
12034 B s   ← A
12035        →
12036 B sS↑
12037
12038   sS   →
12039     ↑
12040   sS    A
12041
12042 B sS ↓← A
12043
12044
12045
```

Figure 6 – Sending payload (combos injected by first controller)



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

from http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Figure 7 – Z80 opcodes that can be sent through SGB input filtering.

the slip position must be relatively near the middle to avoid having to deal with two slips.[16]

3.8 Stage 2: Sending packets to escape SGB from very little space.

We have just 39 bytes to work with in the Stage 2 payload we just wrote and we need to make the most out of every last byte. Fortunately, Pokémon Red already contains a routine that sends a command packet into the SNES. The catch is the code to send that packet is in another ROM bank (`0x1C`) that we need to switch to. While the ROM bank can be switched by a single write, the game NMI routine (which runs every frame) does not save the bank - it switches to one stored in another memory address instead. Two writes are needed to reliably change the bank which would take too much space; however, the common part of ROM (mapped regardless of the bank) has a function that does something, then switches banks and returns. That function makes for a very useful gadget! The entry address for this function is `0x00AF`, with register `A` holding the bank number.

[16] The movie we used was 2(prologue)+5(banksetting)+6(packetsend)+5(packetsend)+1(nop-for-slip)+2(hang)+11(packet1)+7(packet2)+2(unused)+2(epilogue)=43 bytes. We later discovered a different method where the smallest possible extended payload would have been 2(prologue)+5(banksetting)+6(packetsend)+2(hang)+13(packet)+2(epilogue)=30 bytes which is still too much to input without a slip due to our data rate being restricted to one nibble at a time, although the packet data's `0x00` portion could potentially be used for this purpose.

[17] It could be possible to use just one, by putting the NMI routine in a memory-mapped SGB packet register, but we decided not to, as we would need full exploit abilities just to test if this method actually works because the emulator isn't accurate enough to test with.

We need to send two separate command packets, described below.[17] The packets aren't a full 16 bytes in length like they appear to be, but 11 and 7 bytes; the tails of the packets are ignored, so we let the packet payloads overrun into whatever happens to be next. After sending the packets, we have no use for the DMG anymore, so we hang the Z80 by entering a tight loop.

The following Stage 2 assembly code is loaded into `0xD33A—D360`.

```
; The gadget takes a new bank number in A.
3E 1C       LD A, #$1C
; Call the bankswitch gadget.
CD AF 00    CALL $00af
; The address of the first packet to send.
21 4D D3    LD HL, packet1
; Call packet send routine.
CD EB 5F    CALL $5feb

; The low byte of address of the 2nd packet.
; used to compensate input slipping.
2E 58       LD L, 0x58
00          NOP
; Call packet send routine.
CD EB 5F    CALL $5feb

18 FE       JR -2       ; Hang the DMG.

packet1: ; 0xd34d
  DB 0x79, 0x00, 0x18, 0x00, 0x06, 0xad,
     0x12, 0x42, 0x30, 0xfb, 0x40

packet2: ; 0xd358
  DB 0x91, 0x18, 0x42, 0x00, 0x00, 0x18,
     0x00, 0x00, 0x00
```

Originally, the `LD L, 0x58 ; NOP` sequence was `LD HL, 0xD358` but we discovered that the transfer routine leaves the upper eight bits of the address in the `H` register at the end of the transfer. The transfer end of the packet at `0xD34D` will be `0xD35D`, so the `H` register will be `D3`, which is exactly the value we want for the next packet, so we can save one byte by just loading the `L` register. The saved byte can taken to be `NOP (00)`.

The repeated input can land on two inputs of the same byte, or the last input of one byte and first input of next. The latter is much better, since for any byte pair, it is possible to construct three valid inputs. However, the first is much worse: The byte will be forced to `00`, and even more unfortunately, the frame rules always cause the duplication to occur in a bad way. The `00` freed from only loading L is close enough to the middle that this byte can be targeted for duplication. It turned out that the emulator doesn't emulate the input slipping quite accurately and we (dwangoAC) had to do a lot of tedious trial and error testing to time the input correctly.[18] The offset between emulator and real hardware turned out to be eight frames, which we adjusted by adding eight frames of no input into the file sent to the bot prior to exiting the menu.

3.9 Exploiting DMG→SGB command packets for gaining a foothold on SNES

The Super Game Boy command packet protocol has two nifty commands for gaining control of the SNES. `0x79` writes arbitrary data to an arbitrary memory location, while `0x91` sets the NMI vector and jumps to an arbitrary address. Both commands are real, documented command packets; they are not undocumented debug commands.

Since the Stage 2 executing on the DMG is so small we needed to minimize the number of packets required. The SNES's controller registers are memory-mapped I/O registers that automatically update each video frame when enabled. It is possible to execute code from those registers but it isn't particularly easy to do so, largely because it is very unsafe to execute anything from those registers when they are in the middle of an update. (There are all sorts of intermediate stages.)

The solution is to find some way for the SNES CPU to waste time during that update elsewhere. The NMI vector and the NMI handler are perfect for this: when enabled, it starts running just before the register starts updating, so we just need an NMI handler that wastes somewhere between roughly 4 and 260 scanlines so it hits after the current NMI returns but before the next NMI starts. Scanning descriptions of various SNES I/O registers, a useful one seems to be `$4212`, which has bit 7 set when the console is performing a vertical retrace. The NMI occurs immediately after the vertical retrace starts and the retrace lasts for about 40 scanlines, so waiting for `$4212` bit 7 to clear works out perfectly. Since the retrace bit is bit 7 and the SNES CPU happens to be in a mode where the `A` regis-

[18] Each blind test took about 5 minutes, as we had to play back the entire movie before reaching the point where we could determine if it worked and we weren't entirely certain it would work at all, but eventually we discovered the correct offset.

[19] Based on the setting of a flags register bit that selects between an 8– and 16–bit `A` register.

ter is 8 bits wide,[19] numbers with bit 7 set show as negative, so it's trivial to branch on those using `BMI` instruction. Handily enough, the `LDA` instruction that loads the memory address into the `A` register sets the condition flags, so we can just loop around that one instruction using `BMI`.

After the loop, we must return from the NMI. This is done using the `RTI` instruction, so the final NMI handler looks like:

```
loop:
AD 12 42 LDA $4212 ;Read 0x4212.
30 FB    BMI loop  ;Loop while bit 7 is set.
40       RTI       ;Return from NMI.
```

This handler trashes the `A` register, which is generally considered bad style, but we can get away with doing that.

We send two packets; the first one writes six bytes (`AD 12 42 30 FB 40`) into the memory address `0x001800`. This is the NMI routine.

```
79                  ; Write Memory
00 18 00            ; Target Address
06                  ; Size
AD 12 42 30 FB 40   ; Content
```

ASH is
playing the SNES!

Figure 8 – Inception

The second one jumps to `0x004218` (which is the start of the controller registers), with the NMI vector set to `0x001800` (which points to the routine we just wrote).[20]

```
91          ; Jump
18 42 00    ; Jump Target
00 18 00    ; NMI Vector
```

3.10 Stage 3: From stable loop in autopoller registers to loading payloads.

(480 bytes per second; 60 payload bytes per second.)

We have transferred control flow to controller registers, but we aren't done just yet. The controller registers are only eight bytes in size, and normally not all bits are even controllable. However, there are some tricks we can play to control all the bits. First, even though a standard SNES controller only has 12 buttons, the autopoller reads all 16 bits. Normally the last four bits are controller type identification bits. Since those bits are read from the controller, the controller can set those bits to whatever it likes, including changing those bits every frame. Second, the last four bytes of the register are read from the second data line that is normally not connected to anything unless there is a multitap device. It isn't possible to just connect a multitap device whenever we like as the game will softlock. Fortunately, it is possible to just connect the second controller so that it shares all the other pins (+5V, ground, latch and clock), but use the second data pin instead the first.

These two tricks allow controlling all 128 bits in the controller registers which gives us 8 bytes of data per frame. While this is a huge improvement over our Stage 1 effective data rate of a nibble per frame it still only amounts to a datarate of 300 bytes per frame because three of those 8 bytes need to be used for looping in the controller registers, leaving only five bytes usable. (Although, as you'll see, only one byte of payload data can be sent per frame.)

Specifically, to loop successfully in the controller registers we need to wait for the NMI induced interrupt in order to avoid the NMI happening at an unpredictable instruction (because the NMI trashes `A`) and then jump to the start of the controller register. Then there is issue that NMI is not initially

[20] We considered putting the NMI code into the SGB packet receive buffer, which is a memory-mapped I/O register (and presumably can be executed by the CPU). We decided against this since the SGB emulation in BSNES is quite questionable and we didn't know if it would work, largely due to the difficulty of testing it.

enabled, even if the handler is set, so the first frame has to enable the NMI handler. Fortunately, this can be done rather compactly:

```
loop:
A9 81       LDA #$81
8D 00 42    STA $4200   ; Set 0x4200 = 0x81 (
     autopoller enabled, IRQ disabled, NMI
     enabled)
CB          WAI
80 F8       BRA loop
```

Since the code is idempotent, this is good time to switch from sending input in once per frame to sending input in once per latch poll. The way the SGB BIOS polls the controllers is completely crazy, often polling more than once per frame, polling too many bits, trying to poll but leaving the latch held high, etc. Because this is a somewhat common problem even in other games, the bot connected to the controller ports has a mode where it synchronizes what input to send based on the edge of each video frame (i.e. 60ths of a second in a polling window) by keeping track of how much time has elapsed; if the game asks for input more than once on the same frame we give it that frame's input again until we know it is time for the next frame's polls, which means we can follow the polling no matter how crazy it is. The obvious tradeoff is that this mode is limited to 8 bytes per frame with 4 controllers attached, so we need to switch the bot's mode to one that is strictly polling based, sending the next set of button presses on each latch. Making that transition can be a bit glitchy considering it was added as a firmware hack but because this piece of code is idempotent we can just spam the same input several times as we only need it to hit in the range. This happens from frame 12117 to 12212 in the movie.

We now have a stable loop in the controller registers that we can use to poke some code into RAM. The five bytes per frame is enough to write one byte per frame into an arbitrary address in first 8kB of the SNES's RAM:

```
LDA #$xx
STA $yyyy
```

This assembles to five bytes, `A9 xx 8D yy yy`. Finally, after the writes, we can use `JML` (four bytes) to jump to the desired address. Since the DMG is still playing some annoying tunes, the first order of business is to try to crash it. Writing `00` to the clock control/reset register at `0x6003` should do the trick by stopping the DMG clock, and in fact this works in the LSNES emulator, but on a real console the annoying tunes keep playing until the DMG corrupts itself enough to crash completely.[21]

3.11 Stage 4: Increasing the datarate even further.

(3840 bytes per second.)

One byte per frame is rather slow as it would take us several minutes to write our payload at that speed so we poke the following routine (Stage 4) that reads 8 bytes per frame from the autopoller registers and writes it sequentially to RAM, starting from `0x1A00` until `0x1B1F` into address `0x19000`.

```
SEP #$30      ;Set 8-bit A and X/Y
LDA #$01      ;Set 0x4200 = 0x01
              ;(autopoller en, NMI dis)
STA $4200
REP #$10      ;Set 16-bit X/Y, keep A 8-bit.
LDY #$1A00    ;Load address to write to.
 wait_vblank_start:
LDA $4212     ;Wait until vblank starts.
BPL wait_vblank_start
 wait_vblank_end:
LDA $4212     ;Wait until vblank ends, so the
              ;new controller value arrives.
BMI wait_vblank_end
LDX #$4218    ;Start address of controller reg
     .
LDA #$00      ; MVN copies 16-bit amount of
     bytes, even with A being 8 bit.
XBA        ; So ensure that the high bits are
     zero.
LDA #$07      ; A = 7, copy 8 bytes.
PHB        ; MVN changes the data bank
     register, so save it.
MVN $7E,$00     ; Copy the 8 bytes from 0
     x4218 to RAM. Y is auto-incremented.
PLB        ; Restore the data bank register.
CPY #$1B20     ; Have we reached 0x1820?
BNE wait_vblank_start   ; If no, wait a frame
      and read again.
JML $7E1A08     ; Jump to read payload.
```

As machine code, `e2 30 a9 01 8d 00 42 c2 10 a0 00 1a ad 12 42 10 fb ad 12 42 30 fb`

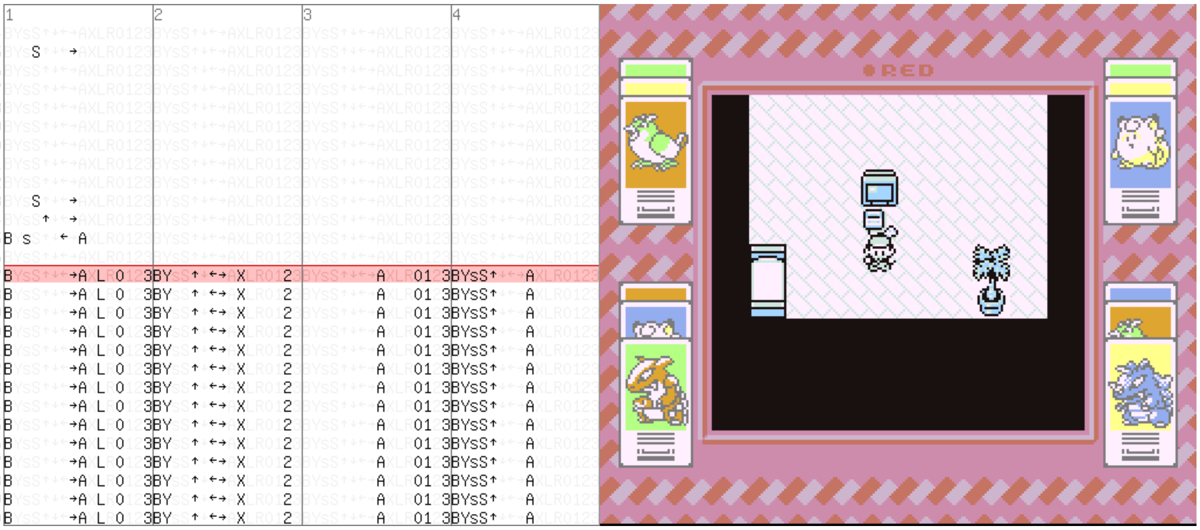
[21] It's not a surprise that it behaves differently in the emulator, as the SGB emulation accuracy in BSNES is questionable in a lot of places; it's possible that the emulator is triggered on a different edge of the clock than real hardware or something similar. Regardless, on real hardware the DMG eventually crashes in a way that makes it stop producing sound and while it's about the equivalent of driving a car into a brick wall instead of hitting the brakes it at least gets the job done.

`a2 18 42 a9 00 eb a9 07 8b 54 7e 00 ab c0 20 1b d0 e4 5c 08 1a 7e.`

Why jump to eight bytes after the start of the payload? It turns out that code loads some junk from what is previously in the controller registers on the first frame, so we just ignore the first few bytes and start the payload code afterwards. Eight bytes per frame still isn't fast enough, so the routine this code pokes into RAM is another loader routine that uses serial controller registers to read eight bytes eight times per frame, for total of 64 bytes per frame.

Let's take a look at the Stage 5 payload:

```
; 0000 => Current transfer address.
; 0002 => Transfer end address.
; 0004 => Blocks to transfer.
; 0006 => Current transfer bank.
; 0008 => 0: Transfer not in progress.
;         1: Transfer in progress.
; 000C => Blocks transferred.
; 0010 => Jump vector to next in chain.
; 0020-0027 => Buffer
; 0080-00BF => Buffer.

Start:
NOP          ; 8 NOPs, for the junk at start.
NOP
NOP
NOP
NOP
NOP
NOP
NOP
SEI
LDA #$00      ; Autopoll off, NMI and IRQ off.
STA $4200

REP #$30         ; 16-bit A/X/Y.

LDA #$0000       ; Initially no transfer.
STA $0008

frame_loop:

SEP #$20
not_in_vblank: ; Wait until next vblank ends
LDA $4212
BPL not_in_vblank
in_vblank:
LDA $4212
BMI in_vblank
REP #$20

LDA #$0008
STA $0004
LDA #$0000
STA $000C

rx_block:
LDA #$0001
STA $4016
LDX #$0003
latch_high_wait:
DEX
BNE latch_high_wait
STZ $4016
LDX #$0004
latch_low_wait:
DEX
BNE latch_low_wait

LDA #$0000
STA $0020
STA $0022
STA $0024
```



```
     1                2                3                4
12104
12105   S   →
12106
12107
12108
12109
12110
12111
12112
12113   S   →
12114    ↑  →
12115B s   ← A
12116
12117B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12118B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12119B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12120B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12121B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12122B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12123B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12124B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12125B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12126B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12127B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12128B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12129B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
12130B      →A L 0  3BY  ↑ ←→ X    2         A   01 3BYsS↑    A
```

Figure 9 – Now using four controllers!

```
STA $0026

LDY #$0010
read_loop:
LDA $4016
PHA
; Bit 0 => 0020, Bit 1 => 0024,
; Bit 8 => 0022, Bit 9 => 0026
BIT #$0001
BNE b0nz
LDA $0020
ASL A
BRA b0d
b0nz:
LDA $0020
ASL A
EOR #$0001
b0d:
STA $0020

PLA
PHA
BIT #$0002
BNE b1nz
LDA $0024
ASL A
BRA b1d
b1nz:
LDA $0024
ASL A
EOR #$0001
b1d:
STA $0024

PLA
PHA
BIT #$0100
BNE b8nz
LDA $0022
ASL A
BRA b8d
b8nz:
LDA $0022
ASL A
EOR #$0001
b8d:
STA $0022

PLA
BIT #$0200
BNE b9nz
LDA $0026
ASL A
BRA b9d
b9nz:
LDA $0026
ASL A
EOR #$0001
b9d:
STA $0026

DEY
BNE read_loop

;Move the block from 0020 to its final place
LDA $000C
ASL A
ASL A
ASL A
CLC
ADC #$0080
TAY
LDX #$0020
LDA #$0007
MVN $00, $00

; Increment the counter at 000C,
; decrement the count at 0004.
; If no more blocks, exit.
LDA $000C
INA
STA $000C
LDA $0004
DEA
STA $0004
BEQ exit_rx_loop
JMP rx_block
exit_rx_loop:

LDA $0008
BNE doing_transfer
; Okay, setup transfer.
LDA $0082
CMP #$FF
BMI not_jump
; This is jump, copy the address.
STA $12
LDA $0080
STA $10
BRA out
not_jump:
LDA $0080     ; Starting address.
STA $0000
LDA $0082     ; Bank.
STA $0006
LDA $0084     ; Ending address.
STA $0002

; Self-modify the move.
LDX #move_instruction
LDA $0006
AND #$FF
STA $01,X

; Enter transfer.
LDA #$0001
STA $0008

; See you next frame.
JMP no_reset_transfer

doing_transfer:

; Copy the stuff to its final place in WRAM.
LDY $0000
LDX #$0080
LDA #$003F
PHB
move_instruction:
MVN $40,$00     ; Bogus bank, will be
```

```
      modified.
PLB
TYA
STA $0000
CMP $0002
BNE no_reset_transfer
STZ $0008      ; End transfer.
no_reset_transfer:
; Next frame.
JMP frame_loop
out:
JMP [$10]
```

3.12 Stage 5: Transfers of data in blocks with headers.

(3,840 bytes per second.)

This routine is rather complex, so let's review some of its trickier parts.

The serial protocol works by first setting the latch bit (bit 0) in 0x4016, then clearing it, then reading the appropriate number of times from 0x4016 (port #1) and 0x4017 (port #2). Bit 0 of the read result is the first data line value, while bit 1 is the second data line value. After each read, the line is automatically clocked so the next bit is read. The two port latch lines are connected together; bit 0 of 0x4016 controls both.

The bot is slow, so we wait after setting/clearing the latch bit. We properly reassemble the input in the usual order of the controller registers, since we have CPU time available to do that. Since we read 16-bit quantities, port 0x4017 is read as high 8 bits, so the data lines there appear as bits 8 and 9.

To handle large payloads, the payload is divided into blocks with headers. Each header tells where the payload is to be written, or, if it is the last block, where to begin execution.

The routine uses self-modifying code: The source and destination banks in MVN are fixed in code, but this code is dynamically rewritten to refer to correct target bank.

3.13 Automating the Movie Creation

Since manually editing, recompiling and transforming inputs gets old very fast when iterating payload ROMs, tools to automate this are very useful. This is the whole reason for having Stage 5 use block headers. Furthermore, to not have one person doing the work every time, it's helpful to have a tool that even script-kiddies can run. The tool to do this is a Lua script that runs inside the emulator (The LSNES emulator has built-in support for running Lua scripts, with all sorts of functions for manipulating the emulator.)

```
dofile("sgb-arbitrarywrite.lua");

make_movie = function(filename)
        write_sgb_data("stage4.dat");
        write_8bytes_data("stage5.dat");
        write_xfer_block(filename, 0x8000, 0x7E8000, 0x4000, 8);
        write_xfer_block(filename, 0x10000, 0x7F8000, 0x7A00, 8);
        write_jump_block(0x7E8051, 8);
        print("Done");
end
```

This code, the main Lua script, refers to four external files. "stage4.dat" contains the memory writes to load the Stage 4 payload from Section 3.11 while executing in the controller registers.

This file contains the Stage 4 payload, plus the ill-fated attempt to shut up the DMG. (As noted previously, it dies on its own later.) The first line containing 0x001900 is the address to jump to after all bytes are written.

2) "stage5.dat", which is the machine code corresponding to the Stage 5 loader.

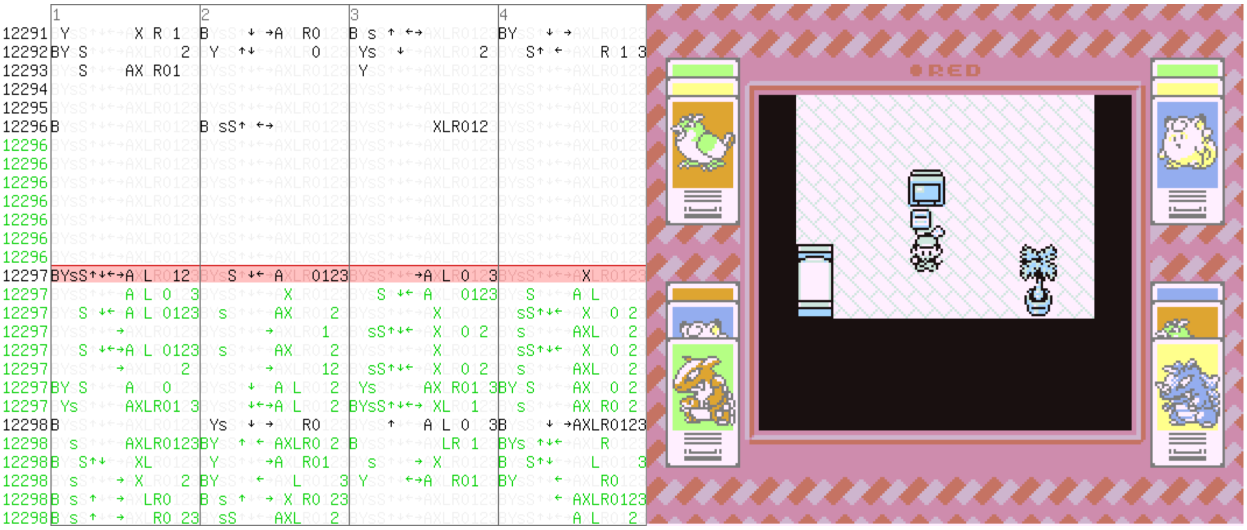
3) A filename taken as a parameter, which is the payload ROM to use. As you can see, the Lua script fixes the memory mappings, but this is okay, as those are not difficult to modify.

The specified memory mappings copy a sixteen kilobyte byte region starting from file offset 0x8000 into 0x7E8000, and the 0x7A00 byte region starting from offset 0x10000 into 0x7F8000. (The first 32kB is assumed to contain initialization code for stand-alone testing, but we don't care about that.)

4) "sgb-arbitrarywrite.lua", which is just a function library.

```
--sgb-arbitrarywrite.lua
lo = function(a) return bit.band(a, 0xFF); end
mid = function(a) return bit.band(bit.lrshift(a, 8), 0xFF); end
hi = function(a) return bit.band(bit.lrshift(a, 16), 0xFF); end

set8 = function(obj, port, controller, index, val)
  for i=0,7 do obj:set_button(port, controller, index + i, bit.test_all(bit.lshift(val, i), 128)); end
end
```

```

add_frame=function(a, b, c, d, e, f, g, h, sync)
  local frame = movie.blank_frame();
  frame:set_button(0, 0, 0, sync);
  set8(frame, 1, 0, 0, b);
  set8(frame, 1, 0, 8, a);
  set8(frame, 1, 1, 0, f);
  set8(frame, 1, 1, 8, e);
  set8(frame, 2, 0, 0, d);
  set8(frame, 2, 0, 8, c);
  set8(frame, 2, 1, 0, h);
  set8(frame, 2, 1, 8, g);
  movie.append_frame(frame);
end

write_sgb_data = function(filename)
  local jump_address = nil;
  local file, err = io.open(filename);
  if not file then error(err); end
  for i in file:lines() do
    if i == "" then
    elseif not jump_address then
      jump_address = tonumber(i);
    else
      local a, b = string.match(i, "(%w+)%s+(%w+)");
      a = tonumber(a);
      b = tonumber(b);
      add_frame(0xA9, b, 0x8D, lo(a), mid(a), 0xCB, 0x80, 0xF8, true);
    end
  end
  add_frame(0x5C, lo(jump_address), mid(jump_address), hi(jump_address), 0, 0, 0x80, 0xF8, true);
  file:close();
end
write_8bytes_data = function(filename)
  local file, err = io.open(filename);
  if not file then error(err); end
  while true do
    local data = file:read(8);
    if not data then break; end
    local a, b, c, d, e, f, g, h = string.byte(data, 1, 8);
    add_frame(a, b, c, d, e, f, g, h, true);
  end
  file:close();
end

write_xfer_block = function(filename, fileoffset, targetaddress, size, speed)
  local file, err = io.open(filename);
  if not file then error(err); end
  file:seek("set", fileoffset);
  while size % (8 * speed) ~= 0 do size = size + 1; end
  local endaddr = bit.band(targetaddress + size, 0xFFFF);
  --Write the header.
  add_frame(lo(targetaddress), mid(targetaddress), hi(targetaddress), 0, lo(endaddr), mid(endaddr), 0, 0, true);
  for i=2,speed do add_frame(0, 0, 0, 0, 0, 0, 0, 0, false); end

  --Write actual data.
  for i = 0,size/8-1 do
    local data = file:read(8);
    if data == nil then data = string.char(0, 0, 0, 0, 0, 0, 0, 0); end
    while #data < 8 do data = data .. string.char(0); end
    local a, b, c, d, e, f, g, h = string.byte(data, 1, 8);
    add_frame(a, b, c, d, e, f, g, h, i %
```



```
     |1               |2               |3               |4
12291| Y       X R 1  |B    ↓ →A  R0   |B s ↑ ←→        |BY   ↓ →        
12292|BY S          2 | Y  ↑↓      0   | Ys  ↓        2 |   S↑ ←    R 1 3
12293|   S    AX R01  |                | Y              |                
12294|                |                |                |                
12295|                |                |                |                
12296|B               |B sS↑ ←→        |         XLR012 |                
12296|                |                |                |                
12296|                |                |                |                
12296|                |                |                |                
12296|                |                |                |                
12296|                |                |                |                
12296|                |                |                |                
12296|                |                |                |                
12297|BYsS↑↓←→A L  12 |   S ↓← A   0123|       →A L 0  3|         X      
12297|        A L 0  3|         X      |   S ↓← A   0123|   S    A L     
12297|   S ↓← A L 0123|  s     AX    2 |         X      |  sS↑↓←  X  0 2 
12297|       →        |       →     1  |  sS↑↓←  X  0 2 |  s     AXL   2 
12297|   S ↓←→A L 0123|  s     AX    2 |         X      |  sS↑↓←  X  0 2 
12297|       →      2 |       →     12 |  sS↑↓←  X  0 2 |  s     AXL   2 
12297|BY S    A   0   |     ↓  A L   2 | Ys     AX R01 3|BY S    AX  0 2 
12297| Ys     AXLR01 3|     ↓←→A L   2 |BYsS↑↓←→ XL  1  |  s     AX R0 2 
12298|B               | Ys  ↓ →   R0   |    ↑   A L 0  3|B    ↓ →AXLR0123
12298|  s     AXLR0123|BY  ↑ ← AXLR0 2 |B         LR 1  |BYs ↑↓←    R    
12298|B  S↑↓    XL    | Y     →A  R01  |  s    → X      |B  S↑↓    L    3
12298|  s    → X    2 |BY    ←   L    3| Y    ←→A  R01  |BY    ←    R0   
12298|B s ↑  →  LR0   |B s ↑  → X R0 23|                |      ←AXLR0123
12298|B s ↑  →   R0 23|  sS    AXL   2 |                |        A L   2 
```

Figure 10 – Why should we wait for next frame? Go sub-frame! (in green)

```
    speed == 0);
  end
  file:close();
end

write_jump_block = function(address, speed)
  add_frame(lo(address), mid(address), hi(
    address), 1, 0, 0, 0, 0, true);
  for i=2,speed do add_frame(0, 0, 0, 0, 0,
    0, 0, 0, false); end
end
```

This script assumes that the loaded movie causes the SNES to jump into controller registers and then enable NMI, using the methods described earlier. It appends the rest of the stages and payload to the movie. Also, since it edits the loaded input, it is possible to just load state near the point of gaining control of the SNES and then append the payload for very fast testing. (Otherwise it would take about two minutes for it to reach that point when executing from the start.)

3.14 Stage 6: Twitch Chat Interface

After successfully transferring our payload, execution of the exploit payload (created by p4plus2) can officially begin. There are three primary states to the final payload: (1) Reset, (2) the Chat Interface, and (3) a TASVideos Webview.

3.14.1 The Reset

Because much of the hardware state is either unknown or unreliable at the point of control transfer we need to initialize much of the system to a known state. On the SNES this usually implies setting a myriad of registers from audio to display state, but also just as important is clearing out WRAM such that a clean slate is presented to the payload. Once we have a cleared state it is possible to perform screen setup.

In the initial case we set the tile data and tilemap VRAM addresses and set the video made to `0x01`, which gives us two layers of 4–bit depth (Layers 1 and 2) and a single layer of 2–bit depth, Layer 3.

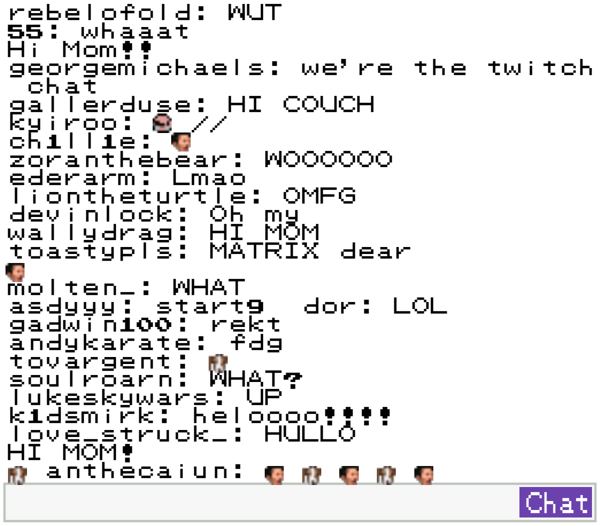
Layer 1 is used as a background which displays the chat interface, while Layer 2 is used for emoji and text. Layer 3 is unused. A special case for the text and emoji however is Red's own text which is actually present on the sprite layer, allowing code to easily update that text independently.

3.14.2 The Chat Interface

Now that we have the screen itself set up and able to run we need to stream data from Twitch chat to the SNES. But we only have 64 bytes per frame available to support emoji as well as the alphabet, numbers, various symbols, and even special triggers for controlling the payload execution. This complexity quickly bogged down our throughput per frame, so we created special encodings for performance! On average the most common characters will be `a-z` in lower case, which conveniently fit into a 5–bit encoding with several more character to spare.

The SNES has both 16–bit and 8–bit modes, so in 16–bit mode we can easily process three characters with a bit to spare! But what about the rest of our character space? Well, we have a single bit remaining and can set it to allow the remaining characters to be alternatively encoded. The alternate encoding allowed for two 7 bit characters, with an additional toggle bit on the second character.

```
BXXXXXXX XXXXXXXX
if(E) goto special_encoding
if(!E) goto normal_encoding
    normal_encoding:
        0AAAAABB BBBCCCCC
        A = full character 1
        B = full character 2
        C = full character 3
    special_encoding:
        1XXXXXXX SXXXXXXX
        if(S) goto special_command
        if(!S) goto read_two_characters
        read_two_characters:
            1AAAAAAA 0BBBBBBB
            A = full character 1
            B = full character 2  (used for
    Red's text)
        special_command:
            1AAAAAAA 1BBBBBBB
            A = full character 1
            B = Command byte
```



```
rebelofold: WUT
55: whaaat
Hi Mom!!
georgemichaels: we're the twitch
 chat
gallerduse: HI COUCH
kyiroo:  //
chillie: 
zoranthebear: WOOOOOO
ederarm: Lmao
liontheturtle: OMFG
devinlock: Oh my
wallydrag: HI MOM
toastypls: MATRIX dear

molten_: WHAT
asdyyy: start9  dor: LOL
gadwin100: rekt
andykarate: fdg
tovargent: 
soulroarn: WHAT?
lukeskywars: UP
kidsmirk: heloooo!!!!
love_struck_: HULLO
HI MOM!
 anthecaiun: 
```

Figure 11 – Twitch chat!

The most important command was `EE`, chosen very arbitrarily, which meant "transition state." The state transition would then toggle between the TASVideos website and chat interface. Also worth noting is that any character with a value of `00` was considered a null character and was not displayed for synchronization purposes.

3.15 The Website

The website itself is not very complicated, rather just interesting to mention to take advantage of mode `0x03` which allowed us to render a 256–color image, rather than the standard 16–color images from the prior section. The only caveat was that we had to make a quick tool to remove duplicate tiles to optimize the tile data to fit in VRAM. Background colors were controlled by tweaking the palette data rather than the image itself, as the SNES is very poor at manipulating raw tile data due to its planar pixel format.

3.16 Outside of the SNES

The bot was connected to the console through the controller ports and a single wire going to the reset pin on the expansion board, meaning that from an external perspective the hardware was completely unmodified. The bot itself was connected by a USB serial interface to a MacBook Pro running Linux. The source of the button presses being sent to the bot was in the form of a continuous bitstream representing the state of all buttons for each frame. Once the payload was fully written and the Twitch chat interface was complete the bitstream transitioned from being pre-created movie content to a bitstream in the format the chat interface payload needed it in, with 5-bit and 7-bit encodings for characters and emoji. This was controlled by the python scripts[22] that relied on a script to identify when Red, the player inside of the Pokémon Red game, said various things. The script also triggered things that TASBot, the robot holding the replay device, would say via the use of espeak, which allowed us to create a conversation between TASBot and Red.

Finally, as part of the script we predefined periods where we would "deface" the TASVideos website by changing it to different colors; this worked by showing an image on the SNES as well as literally defacing the actual website. Finally, the script was built with the ability to send commands to a serial-controlled camera, but truth be told we ran out of time to test it so we used a bit of stage magic to pretend like Twitch chat was interacting with the camera by typing directions to move it, and we had a helpful volunteer running the camera for us.

3.17 Live Performance

These exploits were unveiled at AGDQ 2015. They were streamed live to over 100,000 people on January 4th with a mangled Python script that didn't trigger the text for Red properly, then again on January 11th with the full payload. The run was very well received and garnered press coverage from Ars Technica[23] among others and resulted in substantially more interest in TASBot and the art of arbitrary code execution on video games than had existed previously. Most importantly, the TAS portions of the marathon where the exploit was featured helped raise over fifty thousand dollars directly to the Prevent Cancer Foundation. Overall, the project was a resounding success, well worth the substantial effort that our team put into it.

[22] `https://github.com/TheAxeMan301/PptIrcBot`

[23] Pokémon Plays Twitch: How a Robot got IRC Running on an Unmodified SNES by Kyle Orland

4 This PDF is also a Gameboy exploit that displays the "Pokémon Plays Twitch" article!

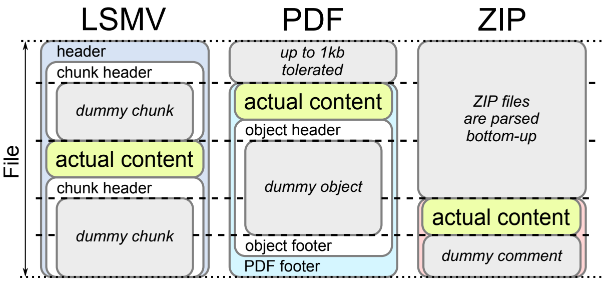
The idea for this polyglot is to embed the contents of the previous article in this fine issue of PoC||GTFO in such a way that it shows on when played as an LSNES movie. So now you can use your copy of the journal to exploit your hardware and read "Pokémon Plays Twitch" on your TV. This way, we hope to start a tradition of articles being viewable on the hardware of the article!

LSNES supports two kinds of movie files, which might better be thought of as input recording files. The older format is ZIP based and formally specified, while the new one is binary and custom. The new binary format has no official specs, but starting a PDF with a ZIP signature would now trigger Adobe's blacklist—clearly, someone at the company must have disliked something about one of our previous releases. So the new, non-ZIP LSMV binary format is the one that we'll use.

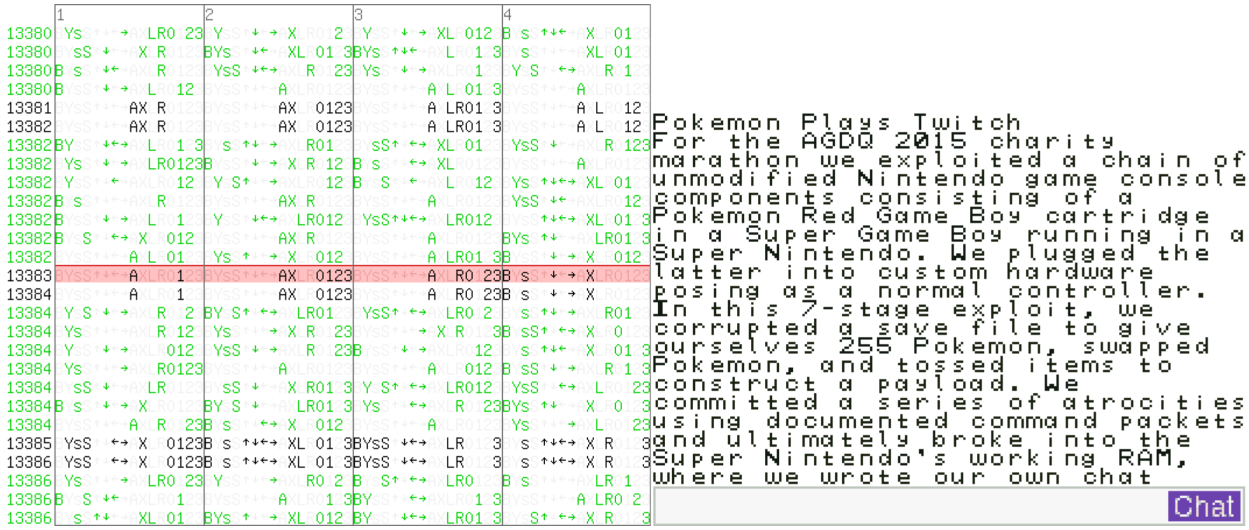
The buffers for read and write calls for movie data are straight out of the movie data in memory. One unintended benefit of the new format is that it is much easier to write from SIGSEGV or similar signal handlers. (The memory allocator cannot be trusted.)

The binary LSMV format is chunk-based. The "`lsmv`" magic must be at offset 0; we can't have any appended data. So the PDF header and content must be added in a dummy chunk early in the LSMV, and the ZIP and PDF footer must be added at the end of the file, in another dummy chunk (see included diagram).

A clean version of the LSMV file has been submitted to TASVideos.[24] You can play this polyglot on a modified LSNES with the hybrid emulation core using BSNES and Gambatte or, if you have the required hardware, on the real stuff!



Be warned that none of these approaches is trivial. We include detailed howtos with the zip contents of this issue.[25]



```
      |1               |2               |3               |4               |
13380 | Ys    →  LR0 23| Y   ↓ → X    2 | Y   ↓ → XL 012 |B s ↑↓←  X  01  |
13380 |  sS ↓   X R    |BYs  ↓←  XL 01 3|BYs ↑↓←   L  1 3|  s      XL 01  |
13380 |B s  ↓←    R    | YsS ↓←→   R  23| Ys  ↓ →     1  | Y S  ←→   R 1  |
13380 |B    ↓ →  L  12 |        A       |        A L 01 3|        A       |
13381 |        AX R    |        AX  0123|        A LR01 3|        A L  12 |
13382 |        AX R    |        AX  0123|        A LR01 3|        A L  12 |
13382 |BY   ↓←→  L  1 3|  s ↑↓ →   R01  |  sS↑ ←→ XL 01  | YsS ↓     R 123|
13382 | Ys  ↓ →  LR0123|B    ↓ → X R 12 |B s   ←→ XL     |        A       |
13382 | Y    ←      12 | Y S↑  →     12 |B  S  ←   L  12 | Ys ↑↓←→ XL 01  |
13382 |B s        R    |        AX R    |        A       | YsS ↓←      12 |
13382 |B    ↓ →  L  1  | Y   ↓←→  LR012 | YsS↑↓←→  LR012 |    ↑↓←→ XL 01 3|
13382 |B  S  ←→ X  012 |        AX R    |        A       |BYs ↑↓    LR01 3|
13382 |        A L 01  | Ys ↑  → X  012 |        A LR01 3|B    ↓ → X  012 |
13383 |        A    1  |        AX  0123|        A  R0 23|B s  ↓ → X      |
13384 |        A    1  |        AX  0123|        A  R0 23|B s  ↓ → X      |
13384 | Y S ↓ →   R  2 |BY S↑ ←→  LR01  | YsS↑ ←→  LR0 2 |  s ↑↓ →   R01  |
13384 | Ys        R 12 | Ys    → X R  23|       → X R   3|B sS↑ ←→ X  0   |
13384 | Y   ↓ →    012 | YsS ↓ →   R  23|B    ↓ →     12 |  s ↑↓←  X  01 3|
13384 | Ys    →   R0123|        A       |        A   012 |B sS ↓ →   R 1 3|
13384 |  sS ↓    LR    |  sS ↓   X R01 3| Y S↑ ←→  LR012 | YsS  ←→  L   23|
13384 |B s  ↓ → X      |BY S ↓    LR01 3| Ys   ←→   R  23|BYs ↑↓   X  0  3|
13384 |        A  R  23|B s   ←→ X  012 |        A       | Ys    →  L   23|
13385 | YsS  ←→ X  0123|B s ↑↓←→ XL 01 3|BYsS ↓←→  LR   3|  s ↑↓←→ X R   3|
13386 | YsS  ←→ X  0123|B s ↑↓←→ XL 01 3|BYsS ↓←→  LR   3|  s ↑↓←→ X R   3|
13386 | Ys    →  LR0 23| Y     →   R0 2 |B  S↑ ←→  LR0   |B s       LR 1  |
13386 |B  S ↓←      1  |        A    1 3|BY    ←→     1 3|        A LR0 2 |
13386 |  s ↑↓   XL 01  |BYs ↑  → XL 012 |BY   ↓←→  LR01 3|   S↑ ←→ X R   3|
```

```
Pokemon Plays Twitch
For the AGDQ 2015 charity
marathon we exploited a chain of
unmodified Nintendo game console
components consisting of a
Pokemon Red Game Boy cartridge
in a Super Game Boy running in a
Super Nintendo. We plugged the
latter into custom hardware
posing as a normal controller.
In this 7-stage exploit, we
corrupted a save file to give
ourselves 255 Pokemon, swapped
Pokemon, and tossed items to
construct a payload. We
committed a series of atrocities
using documented command packets
and ultimately broke into the
Super Nintendo's working RAM,
where we wrote our own chat
```

[24] http://tasvideos.org/4947S.html

[25] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/sgbhowto.pdf`

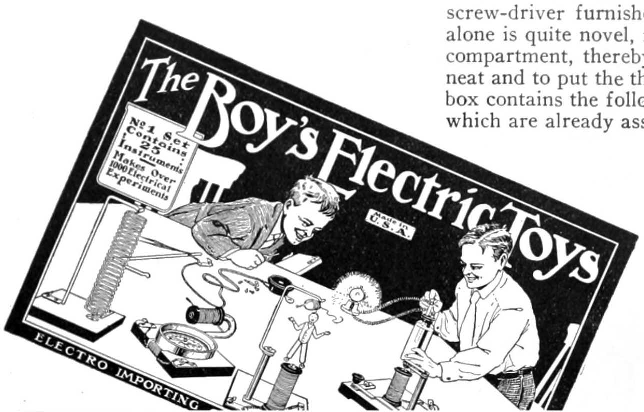
"The Boy's Electric Toys"

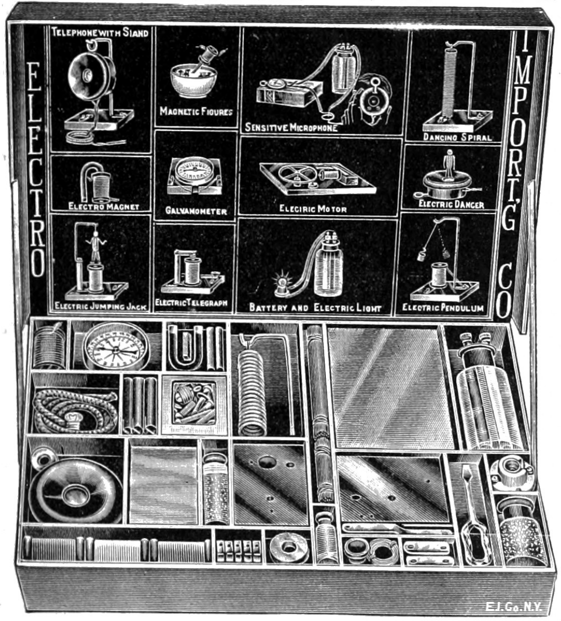
There have been other electrical experimental outfits on the market thus far, but we do not believe that there has ever been produced anything that comes anywhere near approaching the new experimental outfit which we illustrate herewith.

"The Boy's Electric Toys" is unique in the history of electrical experimental apparatus, as in the small box which we offer enough material is contained TO MAKE AND COMPLETE OVER TWENTY-FIVE DIFFERENT ELECTRICAL APPARATUS without any other tools, except a screw-driver furnished with the outfit. The box construction alone is quite novel, inasmuch as every piece fits into a special compartment, thereby inducing the young experimenter to be neat and to put the things back from where he took them. The box contains the following complete instruments and apparatus which are already assembled:

Student's chromic plunge battery, compass-galvanometer, solenoid, telephone receiver, electric lamp. Enough various parts, wire, etc., are furnished to make the following apparatus:

Electromagnet, electric cannon, magnetic pictures, dancing spiral, electric hammer, galvanometer, voltmeter, hook for telephone receiver, condenser, sensitive microphone, short distance wireless telephone, test storage battery, shocking coil, complete telegraph set, electric riveting machine, electric buzzer, dancing fishes, singing telephone, mysterious dancing man, electric jumping jack, magnetic geometric figures, rheostat, erratic pendulum, electric butterfly, thermo electric motor, visual telegraph, etc., etc.


The Boy's Electric Toys
No 1 Set Contains 25 Instruments
Makes Over 1000 Electrical Experiments
Made in U.S.A.
ELECTRO IMPORTING



No. EX2002

This does not by any means exhaust the list, but a great many more apparatus can be built actually and effectually.

With the instruction book which we furnish, one hundred experiments that can be made with this outfit are listed, nearly all of these being illustrated with superb illustrations. We lay particular stress on the fact that no other materials, goods or supplies are necessary to perform any of the one hundred experiments or to make any of the 25 apparatus. Everything can be constructed and accomplished by means of this outfit, two hands, and a screw-driver. Moreover this is the only outfit on the market to-day in which there is included a complete chromic acid plunge battery, with which each and everyone of the experiments can be performed. No other source of current is necessary.

Moreover, the outfit has complete wooden bases with drilled holes in their proper places, so that all you have to do is to mount the various pieces by means of the machine screws furnished with the set.

The outfit contains 114 separate pieces of material and 24 pieces of finished articles ready to use at once.

The box alone is a masterpiece of work on account of its various ingenious compartments, wherein every piece of apparatus fits.

Among the finished material the following parts are included: Chromic salts for battery, lamp socket, bottle of mercury, core wire (two different lengths), a bottle of iron filings, three spools of wire, carbons, a quantity of machine screws, flexible cord, two wood bases, glass plate, paraffine paper, binding posts, screw-driver, etc., etc. The instruction book is so clear that anyone can make the apparatus without trouble, and besides a section of the instruction book is taken up with the fundamentals of electricity to acquaint the layman with all important facts in electricity in a simple manner.

All instruments and all materials are well finished and tested before leaving the factory. We guarantee satisfaction.

We wish to emphasize the fact that anyone who goes through the various experiments will become proficient in electricity and will certainly acquire an electrical education whibh cannot be duplicated except by frequenting an electrical school for some months.

The size over all of the outfit is 14 x 9 x 2¼. Shipping weight, 8 lbs.

No. EX2002 "The Boy's Electric Toys," outfit as described $5.00

"The Livest Catalog in America"

Our big, new electrical cyclopedia No. 19 is waiting for you. Positively the most complete Wireless and electrical catalog in print today. 228 Big Pages, 600 illustrations, 500 instruments and apparatus, etc. Big "Treatise on Wireless Telegraphy." 20 FREE coupons for our 160-page FREE Wireless Course in 20 lessons. FREE Cyclopedia No. 19 measures 7 x 5¼". Weight ½ lb. Beautiful stiff covers.

Now before you turn this page write your name and address on margin below, cut or tear out, enclose 6 cts. stamps to cover mail charges, and the Cyclopedia is yours by return mail.

THE ELECTRO IMPORTING CO.
231 Fulton Street, New York City

ELECTRO IMPORTING CO., 231 Fulton St., N.Y.

5 SWD Marionettes; or, The Internet of Unsuspecting Things

by Micah Elizabeth Scott

Greetings, neighbors! Let us today gather to celebrate the Internet of Things. We live in a world where nearly any appliance, pet, or snack food can talk to the Cloud, which sure is a disarming name for this random collection of computers we've managed to network together. I bring you a humble PoC today, with its origins in the even humbler networking connections between tiny chips.

5.1 Firmware? Where we're going, we don't need firmware.

I've always had a fascination with debugging interfaces. I first learned to program on systems with no viable debugger, but I would read magazines in the nineties with articles advertising elaborate and pricey emulator and in-circuit debugger systems. Decades go by, and I learn about JTAG, but it's hard to get excited about such a weird, wasteful, and under-standardized protocol. JTAG was designed for an era when economy of silicon area was critical, and it shows.

More years go by, and I learn about ARM's Serial Wire Debug (SWD) protocol. It's a tantalizing thing: two wires, clock and bidirectional data, give you complete access to the chip. You can read or write memory as if you were the CPU core, in fact concurrently while the CPU core is running. This is all you need to access the processor's I/O ports, its on-board serial ports, load programs into RAM or flash, single-step code, and anything else a debugger does. I took my first dive into SWD in order to develop an automated testing infrastructure for the Fadecandy LED controller project. There was much yak shaving, but the result was totally worthwhile.

More recently, Cortex-M0 microcontrollers have been showing up with prices and I/O features competitive with 8-bit microcontrollers. For example, the Freescale MKE04Z8VFK4 is less than a dollar even in single quantities, and there's a feature-rich development board available for $15. These micros are cheaper than many single-purpose chips, and they have all the peripherals you'd expect from an AVR or PIC micro. The dev board is even compatible with Arduino shields.

In light of this economy of scale, I'll even consider using a Cortex-M0 as a sort of I/O expander chip. This is pretty cool if you want to write microcontroller firmware, but what if you want something without local processing? You could write a sort of pass-through firmware, but that's extra complexity as well as extra timing uncertainty. The SWD port would be a handy way to have a simple remote-controlled set of ARM peripherals that you can drive from another processor.

Okay! So let's get to the point. SWD is neat, we want to do things with it. But, as is typical with ARM, the documentation and the protocols are fiercely layered. It leads to the kind of complexity that can make little sense from a software perspective, but might be more forgivable if you consider the underlying hardware architecture as a group of tiny little machines that all talk asynchronously.

The first few tiny machines are described in the 250-page ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 tome.[26] It becomes apparent that the tiny machines must be so tiny because of all the architectural flexibility the designers wanted to accommodate. To start with, there's the Debug Port (DP). The DP is the lower layer, closest to the physical link. There are different DPs for JTAG and Serial Wire Debug, but we only need to be concerned with SWD.

We can mostly ignore JTAG, except for the process of initially switching from JTAG to SWD on

[26] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031c/index.html`

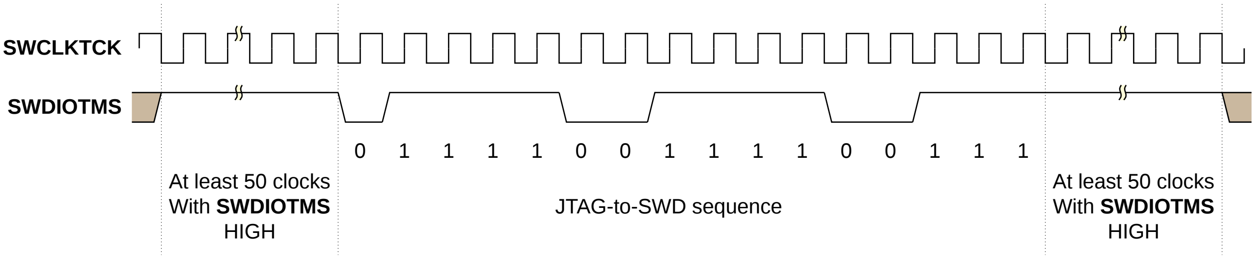

SWCLKTCK
SWDIOTMS
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
At least 50 clocks With **SWDIOTMS** HIGH
JTAG-to-SWD sequence
At least 50 clocks With **SWDIOTMS** HIGH

Figure 12 – JTAG-to-SWD sequence timing

systems that support both options. SWD's clock matches the JTAG clock line, and SWD's bidirectional data maps to JTAG's TMS signal. A magic bit sequence in JTAG mode on these two pins will trigger a switch to the SWD mode, as shown in Figure 12.

SWD will look a bit familiar if you've used SPI or I2C at all. It's more like SPI, in that it uses a fast and non-weird clocking scheme. Each processor's data sheet will tell you the maximum SWD speed, but it's usually upwards of 20 MHz. This hints at why the protocol includes so many asynchronous layers: the underlying hardware operates on separate clock domains, and the debug port may be operating much faster or slower than the CPU clock.

Whereas SPI typically uses separate wires for data in and out, SWD uses a single wire (it's in the name!) and relies on a "turnaround" period to switch bus directions during one otherwise wasted clock cycle that separates groups of written or returned bits. These bit groups are arranged into tiny packets with start bits and parity and such, using turnaround bits to separate the initial, data, and acknowledgment phases of the transfer. For example, see Figures 13 and 14 to execute read and write operations and for all the squiggly details on these packets, the tome has you covered starting with Figure 4-1.

These low-level SWD packets give you a memory-like interface for reading and writing registers; but we're still a few layers removed from the kind of registers that you'd see anywhere else in the ARM architecture. The DP itself has some registers accessed via these packets, or these reads and writes can refer to registers in the next layer: the Access Port (AP).

The AP could really be any sort of hardware that needs a dedicated debug interface on the SoC. There are usually vendor specific access ports, but usually you're talking to the standardized MEM-AP which gives you a port for accessing the ARM's AHB memory bus. This is what gives the debugger a view of memory from the CPU's point of view.

Each of these layers are of course asynchronous. The higher levels, MEM-AP and above, tend to have a handshaking scheme that looks much like any other memory mapped I/O operation. Write to a register, wait for a bit to clear, that sort of thing. The lower level communications between DP and AP needs to be more efficient, though, so reads are pipelined. When you issue a read, that transaction will be returning data for the previous read operation on that DP. You can give up the extra throughput in order to simplify the interface if you want, by explicitly reading the last result (without starting a new read) via a Read Buffer register in the DP.

This is where the Pandora's Box opens up. With the MEM-AP, this little serial port gives you full access to the CPU's memory. And as is the tradition of the ARM architecture, pretty much everything is memory-mapped. Even the CPU's registers are indirectly accessed via a memory mapped debug controller while the CPU is halted. Now everything in the thousands of pages of Cortex-M and vendor-specific documentation is up for grabs.

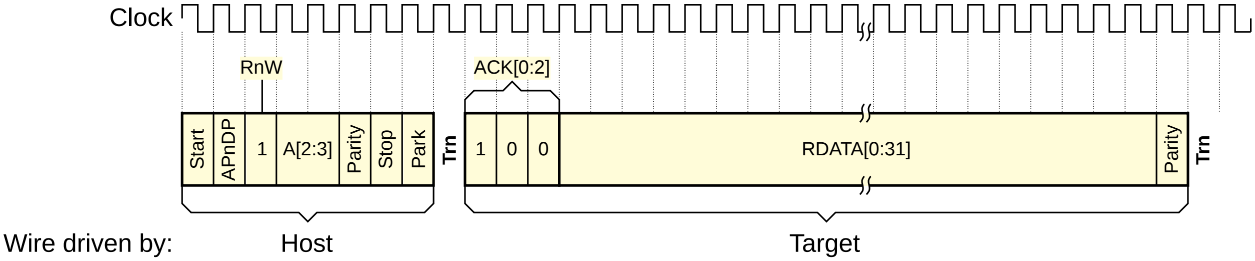


Figure 13 – Serial Wire Debug successful read operation

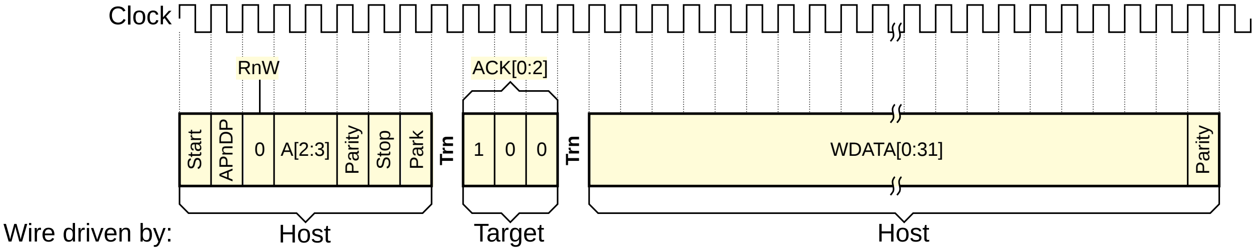


Figure 14 – Serial Wire Debug successful write operation

5.2 Now I'm getting to the point.

I like making tools, and this seems like finally the perfect layer to use as a foundation for something a bit more powerful and more explorable. Combining the simple SWD client library I'd written earlier with the excellent Arduino ESP8266 board support package, attached you'll find `esp8266-arm-swd`,[27] an Arduino sketch you can load on the $5 ESP8266 Wi-Fi microcontroller. There's a README with the specifics you'll need to connect it to any ARM processor and to your Wi-Fi. It provides an HTTP GET interface for reading and writing memory. Simple, joyful, and roughly equivalent security to most Internet Things.

These little HTTP requests to read and write memory happen quickly enough that we can build a live hex editor that continuously scans any visible memory for changes, and sends writes whenever any value is edited. By utilizing all sorts of delightful HTML5 modernity to do the UI entirely client-side, we can avoid overloading the lightweight web server on the ESP8266.

This all adds up to something that's I hope could

[27] `unzip pocorgtfo10.zip esp8266-arm-swd.zip`

```
<ul>
  <li>
    Turn the LED
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100800"> red </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200800"> green </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300000"> blue </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200000"> cyan </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100000"> pink </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00000000"> whiteish </a>, or
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300800"> off </a>
  </li>
  <li>
    Now <a is="swd-async-action" href="/api/halt"> halt the CPU </a> and let's have some scratch RAM:
    <p>
      <swd-hexedit addr="0x20000000" count="32"></swd-hexedit>
    </p>
  </li>
  <li>
    <a is="swd-async-action" href="/api/mem/write?0x20000000=0x22004b0a&.=0x4a0a601a&.=0x601a4b0a&.=0x4a0b4b0a&.=0x4b0b6013&.=0x2b003b01&.=0x2380d1fc&.=0x6013035b&.=0x3b014b07&.=0xd1fc2b00&.=0x46c0e7f0&.=0x40048008&.=0x00300800&.=0x400ff014&.=0x00200800&.=0x400ff000&.=0x00123456&.=0x7fffffbc&.=0x00000001">
      Load a small program
    </a>
    into the scratch RAM
  </li>
  <li>
    <a is="swd-async-action" href="/api/reg/write?0x3c=0x20000000"> Set the program counter </a>
    (<span is="swd-hexword" src="/api/reg" addr="0x3c"></span>)
    to the top of our program
  </li>
    <li>
      The PC <i>sample</i> register (<span is="swd-hexword" addr="0xe000101c"></span>)
      tells you where the <i>running</i> CPU is
    </li>
    <li>
      <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0001"> Let the CPU run! </a>
      (or try a <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0005"> single step </a>)
    </li>
    <li>
      While the program is running, you can modify its delay value:
      <span is="swd-hexword" addr="0x20000040"></span>
    </li>
</ul>
```

Figure 15 – Single Wire Debug from HTML5

be used for a kind of *literate* reverse engineering and debugging, in the way Knuth imagined literate programming. When trying to understand a new platform, the browser can become an ideal sandbox for both investigating and documenting the unknown hardware and software resources.

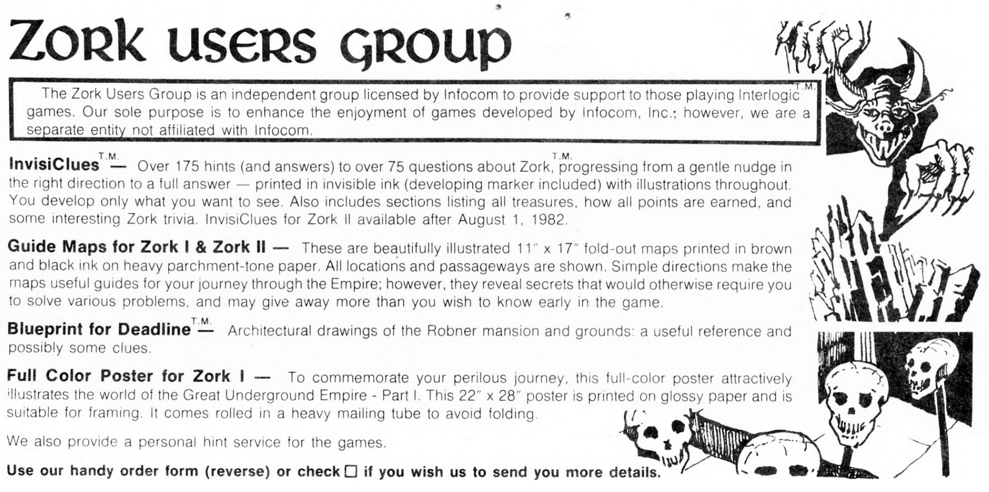
The included HTML5 web app, served by the Arduino sketch, uses some Javascript to define custom HTML elements that let you embed editable hex dumps directly into documentation. Since a register write is just an HTTP GET, hyperlinks can cause hardware state changes or upload small programs.

There's a small example of this approach on the "Memory Mapped I/O" page, designed for the $15 Freescale FRDM-KE04Z board. This one is handy as a prototyping platform, particularly since the I/O is 5V tolerant and compatible with Arduino shields. Figure 15 contains the HTML5 source for that demo.

This sample uses some custom HTML5 elements defined in `/script.js`: `swd-async-action`, `swd-hexedit`, and `swd-hexword`. The `swd-async--action` isn't so exciting, it's really just a special kind of hyperlink that shows a pass/fail result without navigating away from the page. The `swd-hexedit` is also relatively mundane; it's just a shell that expands into many `swd-hexword` elements. That's where the substance is. Any `swd--hexedit` element that's scrolled into view will be refreshed in a continuous round-robin cycle, and the content is editable by default. These become simple but powerful tools.

5.3 Put a chip in it!

While the practical applications of `esp8266-arm-swd` may be limited to education and research, I think it's an interesting Minimum Viable Internet Thing. With the ESP8266 costing only a few dollars, anything with an ARM microcontroller could become an Internet Thing with zero firmware modification, assuming you can find the memory addresses or hardware registers that control the parts you care about. Is it practical? Not really. Secure? Definitely not! But perhaps take a moment to consider whether it's really any worse than the other solutions at hand. Is ARM assembly and HTML5 your kind of fun? Please send pull requests. Happy hacking



HERE YOU 'LEARN BY DOING'

The Only Way to Learn Electricity

The only way you can become an expert is by doing the very work under competent instructors, which you will be called upon to do later on. In other words, *learn by doing*. That is the method of the New York Electrical School.

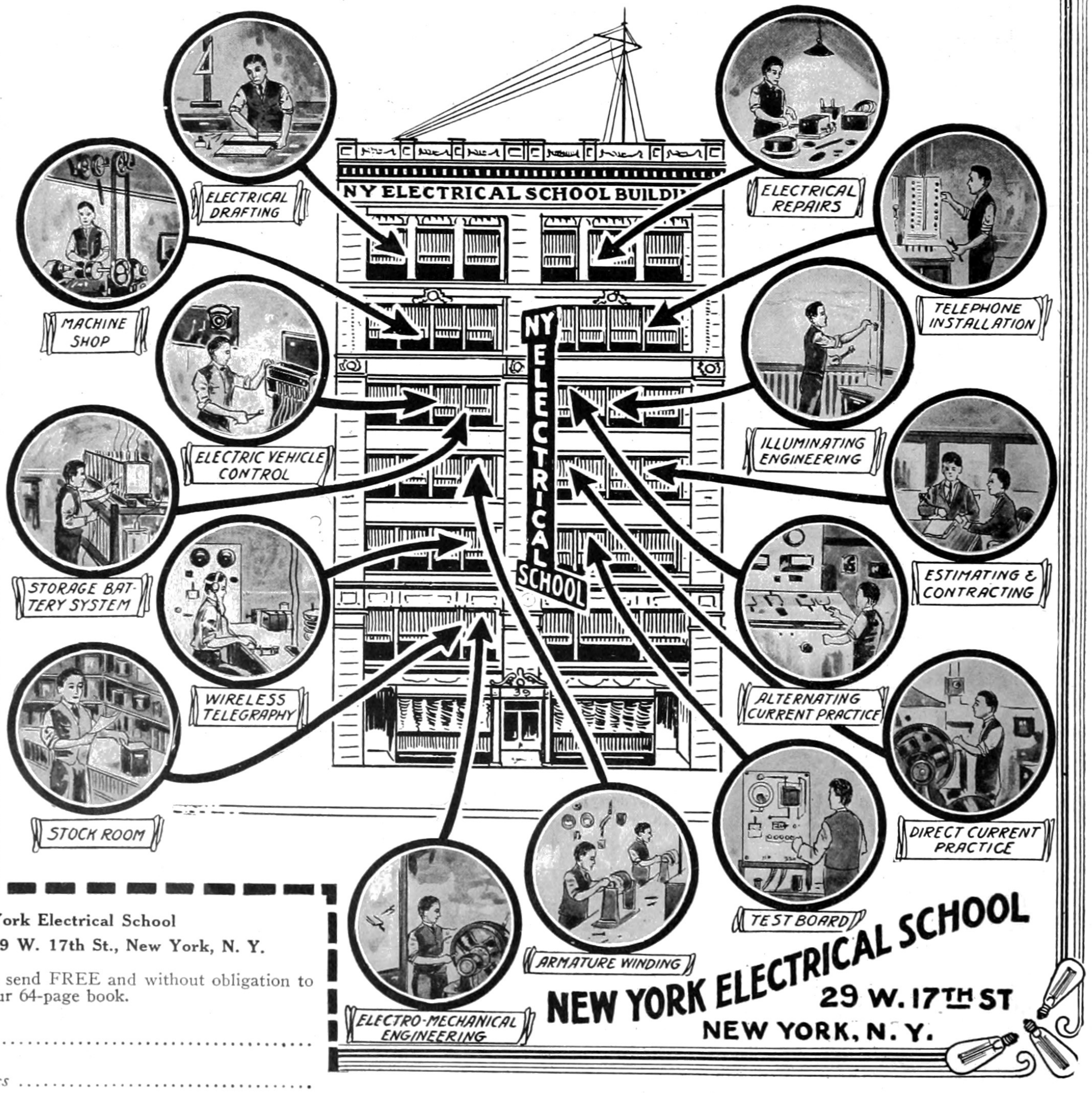
Five minutes of actual practice properly directed is worth more to a man than years and years of book study. Indeed, Actual Practice is the only training of value, and graduates of New York Electrical School have proved themselves to be the only men that are fully qualified to satisfy EVERY demand of the Electrical Profession.

At this "Learn by Doing" School a man acquires the art of Electrical Drafting; the best business method and experience in Electrical Contracting, together with the skill to install, operate and maintain all systems for producing, transmitting and using electricity. A school for Old and Young. Individual instruction.

And Now

If you have an ambition to make a name for yourself in the electrical field you will want to join the New York Electrical School. It will be an advantage to you to start at once. Hurry and send for our 64-page book which tells you all about the school, with pictures of our equipment and students at work, and a full description of the course. You need not hesitate to send for this book. It is FREE to everyone interested in electricity. It will not obligate you to send for it. Send the coupon or write us a letter. But write us *now* while you are thinking about the subject of electricity.

School open to visitors 9 A. M. to 9 P. M.



NEW YORK ELECTRICAL SCHOOL
29 W. 17TH ST
NEW YORK, N. Y.

New York Electrical School
29 W. 17th St., New York, N. Y.

Please send FREE and without obligation to me your 64-page book.

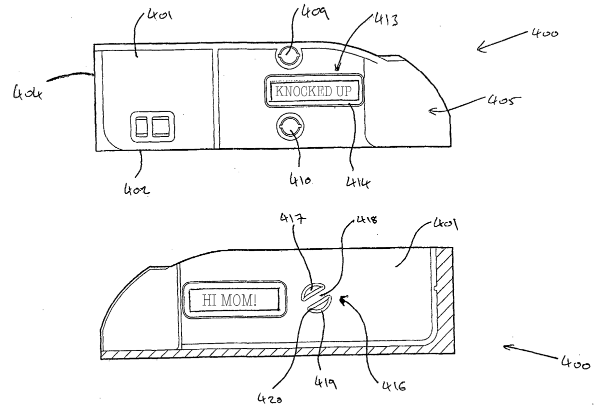
Name ....

Address ....

6 Reversing a Pregnancy Test; or, Bitch better have my money!

by Amanda Wozniak

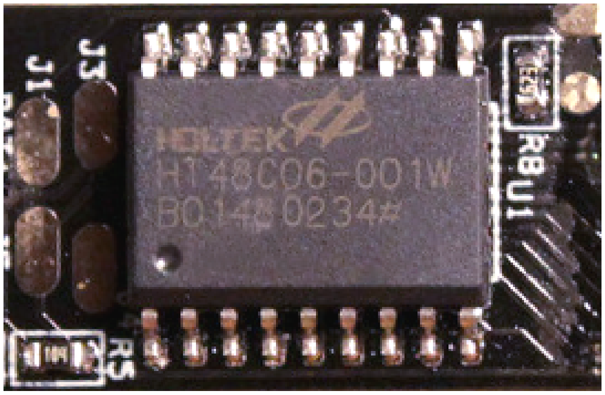
The adventure started like most adventures do—in a dark bar near a technical institute over pints of IPA. An serial entrepreneur plied me with compliments, alcohol and assurances of a budget worthy of my hourly rate to take an off-the shelf device and build a sales-pitch demo in support of his natal company's fund-raising and growth plan. The goal was to take approximately zero available fabrication resources other than myself and spend a couple of months to make a universally approachable, easy to use demonstration prototype for a (now utterly defunct) startup's flow strip technology with a hack-a-thon patented Internet-of-Things interface. The target was an entry straight out of PC Magazine's *The Secret World of Embedded Computers*, the thing no active neighbor should be without—a handy-dandy off the shelf CVS digital pregnancy test.



6.1 Fast, Cheap, and Easy

Head on down to your local pharmacy, and virtually every store will carry a nifty brand of digital pregnancy tests. All of these tests are basically identical (inside and out), and the marketing strategy is simple. Humans are bad at reading analog inputs, so when your time comes, let technology ease your mind whether you, the user is stressed to the breaking point trying to get pregnant or if you're in the boat of desperately hoping you're sterile. "Oh god, it's been three seconds. Or minutes? Wait? What happened to space time. Is there one blue line? Two? I feel faint. Fish? Fuck! I'm pregnant with mutant fish babies."[28]

Now, it doesn't matter which brand you buy for this exercise—as far as I can tell, they're all based on the same two-chip solution built around a Holtek HT48C06 microprocessor. And you can guess at the function without cracking the case – just go buy one (for extra bonus points, look as underaged as possible) and look at the test strips themselves.



Remember, this OTS technology is extra cool because back in the day, instead of peeing on a stick, women suspected of pregnancy[29] had to have their urine injected into a rabbit in order to assess pregnancy before the onset of "the quickening." If you think it's hard telling the difference between '+' and '–', you definitely haven't had to divine your future livelihood from the appearance of leporid entrails. And for extra bonus by the Theory Of Cyber-Extension, every time you use a digital pregnancy test, a cute bunny Tamagotchi is saved from certain death.

6.2 Basics of the Test

Each strip has an absorbent area (that you pee on) and a clear window where the test results show up. One stripe is a control stripe that 'fires' (changes color) in any liquid from water to bourbon, and the other one is a test stripe that only fires when sufficient concentrations of the hormone hCG are present

[28] The mutant fish baby thing is kind of true according to developmental biology, but that's not really our focus today.

[29] *Fun fact*: Eve was the first hacker and Cain was her first 0-day. Humankind is the ultimate Trojan. Since Cain was such a dick in the Biblical sense, the hacking community has carried his mark of social stigma until this very day.

in the fluid sample. (hCG stands for Human Chorionic Gonadotropin, named because scientists snicker at words like "gonad.") You can use the strips without the digital tester, because all you're being sold is a device that will load in one of the basic strips, and monitor the control and test stripes, and return three results: ERROR, NOT or PREGNANT. It turns out that $50 and getting at least one pregnant woman to pee on a test strip can end up for an entertaining couple of evenings at the old workbench.

Following these instructions, with enough time, patience and abstinence, you'll be able to make your own legitimate-looking pregnancy test that works on men and women alike! Or jazz it up to say "`HI MOM`" in no time.

6.3 Teardown

To open the case of a digital pregnancy test (DPT), take a nickel or quarter, place it in the detent in the injection molded case, and gently twist. The model of DPT I did most of my work with was the generic "CVS Clear Results," test – the mechanical specifics may vary from brand to brand, but the nicest part of the cheap injection-molded plastic is that the shell parts are universally thin-walled and toleranced to snap-fit together, which makes it easy to snap them apart without visibly damaging the case.

Inside that case, there will be a circuit board that has another multi-piece injection-molded assembly of ABS plastic, press-fitted into mounting holes on the PCB. This is the test strip alignment/ejection mechanism.[30] For my purposes, I removed this semi-destructively, by twisting off the retention pins on the back side of the PCB. I wanted to save the housing for when I rebuilt the test with my own internal electronics, to be virtually indistinguishable from the stock pregnancy test but with added entrepreneurial functions. This strategic re-use of injection molded parts and hard-to-design mechanisms adds that special professional flair to demonstration prototypes.

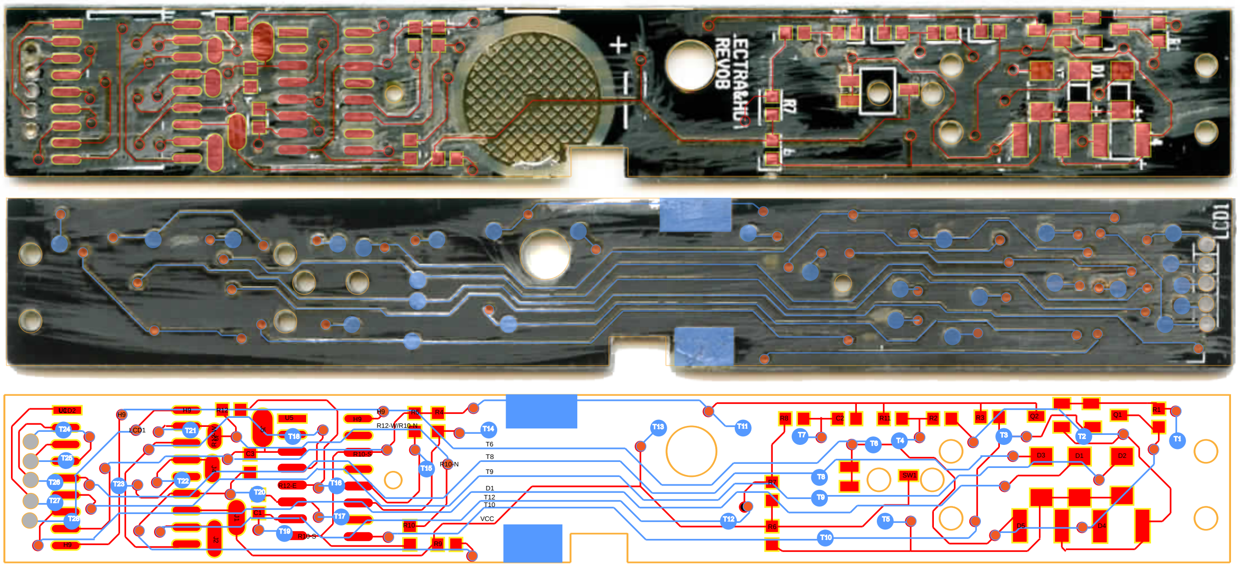
Once you've got the holder off, you'll uncover an activation switch and the analog optical sensor (made of two photodiodes and three LEDs), a PLL (used only for its voltage-controlled oscillator) IC, the aforementioned Holtek HT48C06, a 3V battery and a custom LCD. You can either look up the battery type to confirm it's 3V, or just read the CE-mark label on the outside of the DPT that lists the part number, lot data, confirmation that this test is made by SPD GmbH out of Geneva, Switzerland (made in China), and that the test runs on 3V DC. Safety first, kids. Also convenient: if you peel up this label, you'll see holes in a pattern of the case that line up with un-tinned pads on the PCB. These are the calibration and test points for the Holtek, which means if you prefer firmware reverse-engineering to hardware reverse-engineering, you can go fiddle with the insides *from* the outside.

By the by, that label isn't tamper-evident. You can easily replace it. Don't get any ideas!

6.4 Schematic

Flick the little button, and you'll see the whole test light up (with or without a strip). The LEDs strobe, the LCD thoughtfully blinks its "thinking" icon, and a scope or DMM will show plenty of pin activity until the test errors out because you just set it off

[30] `unzip pocorgtfo10 pregpatent.pdf`



without a valid test strip. I could have started probing there, but I realized that an optical test requires a dark environment, and I wanted to bring my test wires out through the conveniently placed unit-test-and-programming holes on the case. My ultimate goal was to test the unit under multiple conditions to determine the internal logic. That meant making a schematic.

I don't enjoy tracing out circuits with dark soldermask, and the DPTs are relatively cheap, so I gathered up the pinouts for each IC and then did my physical net trace using graphic design tools.

Step 1. Desolder all components from the PCB.

Step 2: Scrub the pads with solder wick to get them nice and flat.

Step 3. Using a razor blade or fine-grit sandpaper, sand off the soldermask with loving attention on both sides of the PCB.

Step 4. Scan the PCB with high contrast.

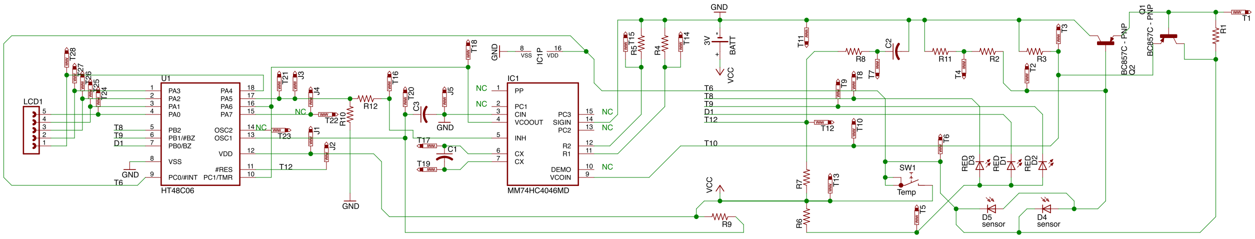
Step 5. Import the scans into an illustration tool of your choice. Color code the top vs. bottom scans to match your preferred layout scheme. Drop circles on the vias—*first*. Then add the IC and passive pins. Then add your traces. Use the vias to register the two images on top of one another for a single layout trace.

Step 6. Annotate the trace with the reference designators from an intact PCB. Add your own net names and pin labels. Use this to build a reference schematic.

6.5 Let's Skip the Firmware

Let's walk through what this sweet little circuit is up to.

First off, the Holtek micro is always on, albeit in sleep mode. The battery is sized for the shelf life of the device plus a couple of uses (three strips ship with each one). When a test strip is placed in the tester, it mechanically triggers the switch which a) flags an interrupt to the microcontroller to wake it up out of sleep mode and b) enables power to the PLL and sense circuitry that would not otherwise be powered. If you remove the test strip mid-test, it cuts power to the PLL and the micro will error out, making it a bit of a pain to work with. Meh,



meh, power-saving feature and fault reporting during foreseeable misuse.

Once all supplies are up, the Holtek samples the state of the optical sensor four times a second for twenty iterations, averaging the samples. In order to sample the test strip, the Holtek drives the LEDs and then reads back the output state of the photodetector, using the voltage-controlled-isolator (VCO) sub-function of that phase-lock-loop IC. The role of the VCO is to convert the analog voltage from the photodetector into a square wave for easy edge counting. Higher voltage implies a higher frequency of edges. Because the micro controls the LED excitation timing, it can easily tell by edge counts what color test strip the LEDs might be illuminating. It's pretty nifty.

Because I wanted to build new electronics to fit inside the case of the original DPT and reproduce a function similar to the original hardware and firmware, I dove into the deeper specifics of how the DPT detects whether one or two blue stripes show up in that plastic clear-view window. The secret is stereoscopic vision enabled by time-division multiplexing and the physical layout of the optosensor. The three LEDs are interdigitated with two parallel photodiodes that are the base current sources in a PNP common emitter amplifier (D4, D5, Q2). The Holtek enables each of the 3 LEDs (D1, D2, D3) sequentially using a 25% LOW duty cycle waveform at 10kHz. The LEDs are strobed in a round-robin fashion and the Holtek samples the result via the VCO.

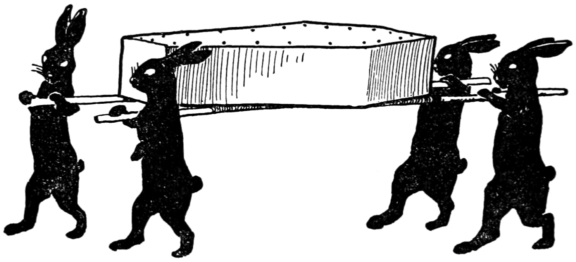
When any one of the three LEDs is strobing, the induced current in the photodiode causes the filter cap on the output of Q2 to charge. The LED's light causes charging, while discharging occurs while the LED is off. Because the Holtek excites the LEDs intermittently, the output of the photodetector is a sawtooth wave. The period of the sawtooth is the LED drive interval, while the peak and trough of the sawtooth wave correspond to the colorimetric intensity of the test stripe that appears and/or the amount of mis-alignment between the photodetector and the LED array.

But how does this produce stereoscopic vision, you ask?

For the same background test strip, when D1 is on, the sawtooth peak-to-peak amplitude will be different than when D3 is on, giving the sensor some ability to resolve spatial light sources. Because the LEDs are independently addressable, it also means that the Holtek can discriminate between a colored stripe hanging over D5 (stripe #1) versus one hanging over D4 (stripe #2). Also, all apologies for the fact that the reference designator order for the diodes makes no physical sense. It's not how I'd design the board, but it apparently took eight revisions for the manufacturer to get this far.

6.6 Schrödinger's Rabbit

Okay, so if you're pregnant, it works like this.



Just kidding, folks—here's what the DPT is doing.

	Photodetectors			Test Stripe	
	D3	D1	D2	ST1	ST2
PREGO	L	H	L	CNTRL	PREGO
CNTRL	L	H	H	CNTRL	...
ERROR	H	H	L	...	PREGO
BLANK	H	H	H	...	...

Remember that a high PD voltage implies more edges counted by the Holtek per excitation cycle. The Holtek uses this *and* sequencing to tell if you're pregnant. Based on the chemistry of the test stripe, the test expects the CNTRL stripe to fire first. If only the CNTRL stripe fires—congratulations, you aren't pregnant! Again, due to chemistry, the PREGO stripe ought to always fire second, if at all. If the stripes fire out of order, that's an error. If the PREGO stripe fires but the CNTRL stripe doesn't, that's an error. If no stripe fires, that's an error.

The factors that contribute to setting the DETECT vs. NO-DETECT threshold for "how many edges do I expect to count if the rabbit died" are (1) the distance from each of the three LEDs to each of the two sensors, (2) the intensity of the LEDs, (3) the color of the LEDs (as that corresponds to the sensitivity of the sensors for a given wavelength of light), (4) the placement of the stripes (if they appear) with respect to the two photodiodes, and (5) the color of the stripe and the saturation of the stripe. Because process controls on LEDs are fucking horrible, each test has to be individually calibrated after assembly.

But that's good news for us!

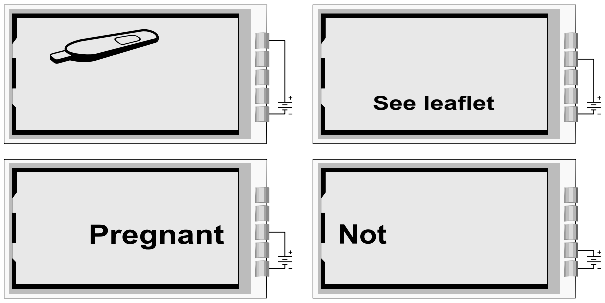
6.7 Hands-On Hacking

Let's be honest, you don't want to come up with a new set of guts to shove into the case of a digital pregnancy test relabeled `0xBEEF` and `0xCAFE` for maximum entertainment and confusion to potential investors! You just want to have fun with the available raw materials that God and your local drugstore have provided.

Each element of the LCD for the digital pregnancy test is custom, just like an old Tamagotchi. That means one pin polarizes the layer with the test logo artwork on it. A second layer covers "`SEE LEAFLET`" for reporting error states, a third conveys "`NOT`" and a fourth, "`PREGNANT`." A given layer is active when the phase of the drive pin is 180 degrees out of phase with the COMMON pin.

So, let's go through the pins that make this happen.

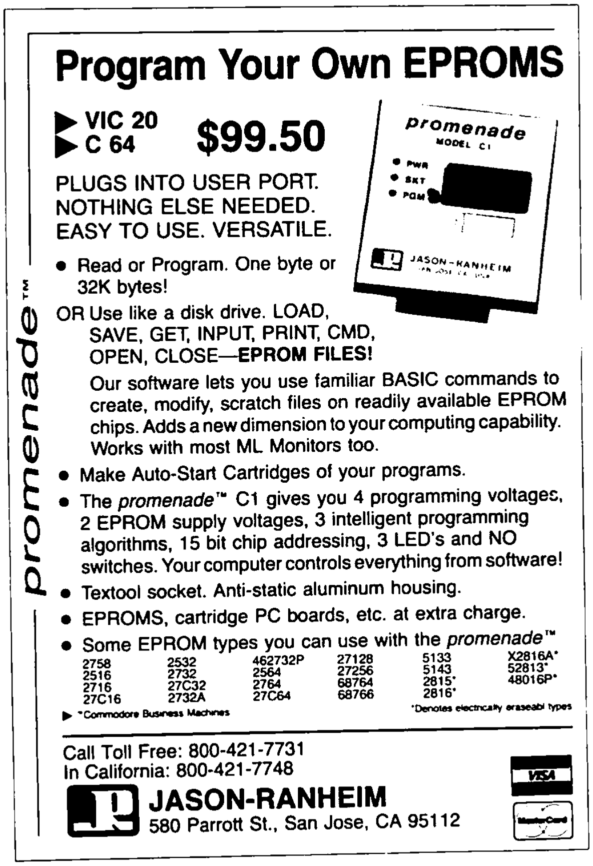
LCD Pin	Image
1	Common
2	"`NOT`"
3	"`PREGNANT`"
4	"`SEE LEAFLET`"
5	Logo



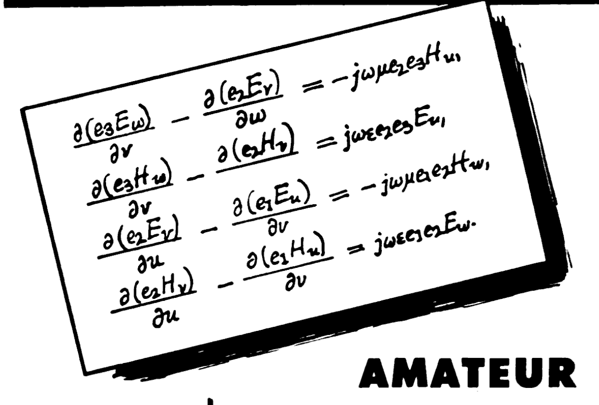
Pin 1 is the rightmost pin if you're looking at the LCD face and the pins are at the top of the package, opposite the reference designator. Make sure to not just short pins—you actually have to lift and move any pins you might be interested in swapping around. Cut a wire here, tack in a jumper there. Mix and match, and get ready to have a ball! Dance a jig! I mean, shoot, a fella could have a pretty good weekend in Vegas with all that.

At the time I was doing this work, the Holtek micro wasn't available for purchase from Digikey or Mouser, so in a fit of intellectual incuriosity, I didn't bother to crack it. Outcome: I can't give you any information on its internals other than what I've inferred from reverse-engineering the rest of the circuit. I'd love to see it done, though—just because the programming physical interface is obfuscated in the primary datasheet doesn't mean it's impossible. If I were doing this twice, I'd start with the ICE. The correct ICE tool for the job, assuming you're into that, is the CICE48U000006A. In the interest of speed, I based my redesign on a PIC16F1933 and a character LCD that fit nicely in the same window as the original.

The demo worked, but I never got paid. So, demo code and hardware design files are available for any neighbor who wants to buy me a beer.
Cheers!
–w0z


promenade
MODEL C1
PWR
SKT
PGM
JASON-RANHEIM
promenade™

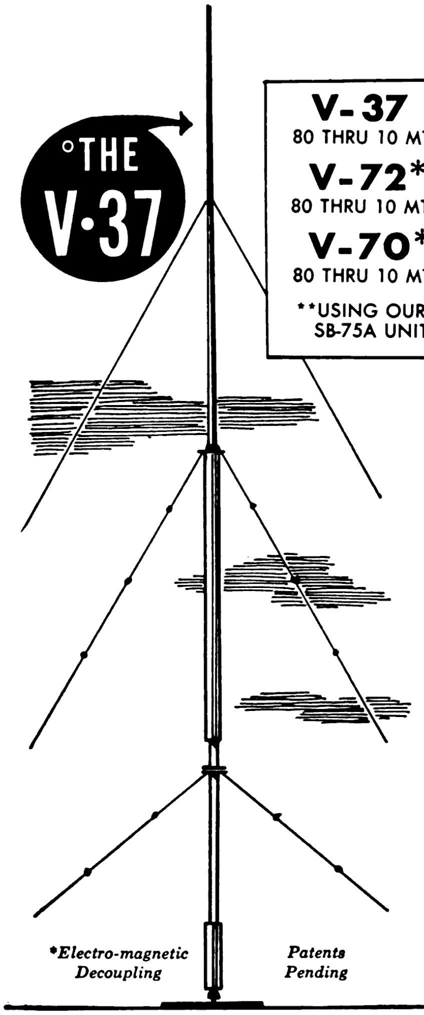
Your Rig is only as effective as the Antenna you tie it to!


$\frac{\partial(e_3E_w)}{\partial v} - \frac{\partial(e_2E_v)}{\partial w} = -j\omega\mu e_2e_3H_u$
$\frac{\partial(e_3H_w)}{\partial v} - \frac{\partial(e_2H_v)}{\partial w} = j\omega\varepsilon e_2e_3E_u$
$\frac{\partial(e_2E_v)}{\partial u} - \frac{\partial(e_1E_u)}{\partial v} = -j\omega\mu e_1e_2H_w$
$\frac{\partial(e_2H_v)}{\partial u} - \frac{\partial(e_1H_u)}{\partial v} = j\omega\varepsilon e_1e_2E_w$

Out of **ANTENNA ENGINEERING LABORATORIES,** where Radiation experts and Scientists have developed the E. D.* principle for Military, Commercial and Marine use, comes a

RADICALLY NEW ALL-BAND "E. D." AMATEUR ROBOT SKYHOOK!

● This New, All-band Antenna, precision-manufactured by **ANTENNA ENGINEERING COMPANY,** does exactly what has long been considered a virtual IMPOSSIBILITY.°



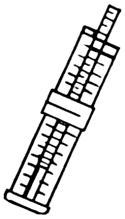
V-37
80 THRU 10 MTRS

V-72**
80 THRU 10 MTRS

V-70**
80 THRU 10 MTRS

**USING OUR SB-75A UNIT

Do you want



AUTOMATIC all-band coverage including Novice, C.D. & MARS

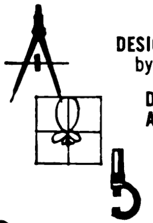
AUTOMATIC IMPEDANCE-MATCHING on EVERY BAND

AUTOMATIC Radiation-pattern Control

AUTOMATIC Colinear Array on 15 and 10 meters (V-37)

ALL with maximum operational **EFFICIENCY** and convenience

Then YOU want—and can NOW HAVE—your CHOICE of a VARIETY OF MODELS of "E.D." All-Banders which have been



DESIGNED for **AMATEUR SERVICE** by Antenna Scientists

DEVELOPED for **HAMS** at the **A. E. C. ANTENNA LABORATORY**

PRECISION-MANUFACTURED for Quality Control at the **A. E. C. FACTORY.**

Sooooo

For Ham Radio at its BEST on your Xmtr & Rcvr
For a THRILL as New & Potent as an "H" bomb
For the TOPS in operating efficiency & convenience

WRITE US FOR DETAILS, LITERATURE AND PRICES

ATTENTION AMATEUR RADIO CLUBS! If you would like one of our Representatives to discuss the vitally-important subject of Amateur Antennas, their problems and how they can be solved, write us for an appointment to address your Members.

ANTENNA ENGINEERING COMPANY
5021 WEST EXPOSITION BLVD., LOS ANGELES 16, CALIF.
TELEPHONE: REpublic 4-7807

**Electro-magnetic Decoupling*

Patents Pending

Peeks,Pokes and Pirates

Disk Layout

A 5.25-inch floppy disk has 35 tracks, numbered $00 to $22 (hex). The format of each track is disk-specific. Most disks split each track into 16 "sectors," but older disks use 13 sectors per track. Some games use 12, 11, or 10. Newer games can squeeze up to 18 sectors in a single track! Just figuring out how data is stored on disk can be a challenge.

Disk Control

Disk control is through "soft-switches," not function calls:

$C080-7,X	move drive arm (phase 0 off/on, phase 1 off/on... until 3)
$C088,X	turn off drive motor
$C089,X	turn on drive motor
$C08C,X	read raw nibble from disk
$C08D,X	reset data latch (used in desync nibble checks)

(X = boot slot x $10)

Disk Boot

A disk is booted in stages, starting from ROM:

$C600	**ROM**	finds track 0 and reads sector 0 into **$800**
$0801	**RAM**	re-uses part of **$C600** code to read more sectors (usually into **$B600+**)
$B700	**RAM**	uses RWTS at **$B800+** to read rest of disk

tip: **$C600** is read-only. But the code there is surprisingly flexible;
It will run at **$9600**, **$8600**, even **$1600**. If you copy it to RAM,
you can insert your own code before jumping to **$0801**.

Prologue And Epilogue

Many protected disks start with DOS 3.3 and change prologue/epilogue values. Here's where to look:

	0x	read	write
	D5	$B955	$BC7A
prologue	AA	$B95F	$BC7F
/	96	$B96A	$BC84
ADDRESS			
\	DE	$B991	$BCAE
epilogue	AA	$B99B	$BCB3
	EB	-----	$BCB8

	0x	read	write
	D5	$B8E7	$B853
prologue	AA	$B8F1	$B858
/	AD	$B8FC	$B85D
DATA			
\	DE	$B935	$B89E
epilogue	AA	$B93F	$B8A3
	EB	-----	$B8A8

Know Your Tools

Every pirate needs:

- a NIBBLE EDITOR for inspecting raw nibbles and determining disk structure (Copy II Plus, Nibbles Away, Locksmith)
- a SECTOR EDITOR for searching, disassembling, patching sector-based disks (Disk Fixer, Block Warden, Copy II Plus)
- a DEMUFFIN TOOL for converting disks to a standard format (Advanced Demuffin, Super Demuffin)
- a FAST DISK COPIER for backing up your work-in-progress! (Locksmith Fast Disk Backup, FASTDSK, Disk Muncher)

Common Code Obfuscation

Apples have a built-in "monitor" and naive disassembler.
Confusing this disassembler is not hard!

Self-modifying code

```
BB03-  4E 06 BB   LSR  $BB06     ← modifies the next instruction
BB06-  71 6E      ADC  ($6E),Y
BB08-  0A         ASL
BB09-  BB         ???
```

By the time **$BB06** is executed...

```
BB03-  4E 06 BB   LSR  $BB06
BB06-  38         SEC            ← the code has changed!
BB07-  6E 0A BB   ROR  $BB0A
```

Branches into the middle of an instruction

```
AEB5-  A0 02      LDY  #$02
AEB7-  8C EC B7   STY  $B7EC
AEBA-  88         DEY
AEBB-  8C F4 B7   STY  $B7F4
AEBE-  88         DEY
AEBF-  F0 01      BEQ  $AEC2     ← Y = 0 here, so this branches...
AEC1-  6C 8C F0   JMP  ($F08C)
AEC4-  B7         ???
AEC5-  8C EB B7   STY  $B7EB

AEBF-  F0 01      BEQ  $AEC2
AEC1-  6C
AEC2-  8C F0 B7   STY  $B7F0     ← ...to here (JMP is never executed)
AEC5-  8C EB B7   STY  $B7EB
```

Manual stack manipulation

```
0800-  A9 51      LDA  #$0F      ← push address to stack ($0FFF)
0802-  48         PHA
0803-  A9 8E      LDA  #$FF
0805-  48         PHA
0806-  20 5D 6A   JSR  $080C     ← call subroutine (also pushes to stack)
0809-  4C 00 08   JMP  $0800
080C-  68         PLA            ← remove address pushed by JSR
080D-  68         PLA
080E-  60         RTS            ← "return" to $0FFF+1 = $1000
```

JMP at **$0809** is never executed! Execution continues at **$1000**.

Undocumented opcodes

```
0801-  74         ???            ← huh?
0802-  4C B0 1C   JMP  $1CB0
```

$74 is an undocumented 6502 opcode that does nothing, but takes a one-byte operand. Here is what actually executes:

```
0801-  74 4C      DOP  $4C,X
0803-  B0 1C      BCS  $0821     ← actually a branch-on-carry (not a JMP)
```

JMP at **$0802** is never executed!

with apologies to Beagle Bros.

4am
to deprotect
and preserve

7 A Brief Description of Some Popular Copy-Protection Techniques on the Apple][Platform

by Peter Ferrie (qkumba, san inc)

7.1 Ancient history

I've been. . . let's call it "preserving" software since about 1983, albeit under a different name. However, the most interesting efforts have been recent, requiring skills that I definitely didn't have until now: I am the author of the only two-side 16-sector conversion of Prince of Persia[31], the six-side 16-sector conversion of The Toy Shop[32], the single file conversion of Joust, Moon Patrol, and Mr. Do!, as well as the DOS and ProDOS file-based conversions of Aquatron, Conan[33], The Goonies, Jungle Hunt, Karateka, Lady Tut (including the long-lost ending from side B), Mr. Do!, Plasmania, and Swashbuckler, to name a few. I am also the only one to crack Rastan cleanly on the IIGS, just 25 years late.[34] Yes, I do 16-bit, too.

I've spent 13 years writing articles for the Virus Bulletin[35] journal. My faithful readers will recognise the style.

7.2 Isn't it ironic

4am[36] declined to write this document himself, but his work and approval inspired me to do it instead. Since his collection is so varied, and his write-ups so detailed, they served as a rich source of information, which I coupled with my own analyses, to fill in the gaps for titles that I don't have. Everyone knows already that he's funny, but he's also quite friendly and very generous. Together, we corrected a few mistakes in the write-ups, so I gave something back. I even consider us friends now, so I think that I got the better deal.

While I don't *regret* writing this paper, I do have to say that, considering the time and effort that it required, he probably made a wise decision. . . ;-)

I have tried to associate at least one example of a real program for each technique, but in Section 7.20 you'll find some nifty new protection techniques that I've developed just for this paper.

7.3 Why why why?

Why the Apple][? It's because I grew up with the Apple][, I learned to code on the Apple][, I *know* the Apple][.

Why now? Because the disks that were fresh when the Apple][was current are failing, and if we do not work to preserve them now, some of the titles will be lost forever.

This paper is dedicated to anyone who has an interest in helping to preserve what's left, I sincerely hope it may help to recognise and defeat the copy-protection that they have come across.

7.4 Okay, let's split

We can separate copy protection into two categories; they are either *What You Have* or *What You Know*. What You Have protections are generally protected disks, while What You Know protections are gener-

[31] `http://pferrie.host22.com/misc/lowlevel14.htm`

[32] `http://pferrie.host22.com/misc/lowlevel15.htm`

[33] `http://pferrie.host22.com/misc/lowlevel16.htm`

[34] `http://www.hackzapple.com/phpBB2/viewtopic.php?t=952`

[35] `http://www.virusbtn.com`

[36] `https://archive.org/details/apple_ii_library_4am`

ally off-disk, such as requests to type in a word from the manual.

What You Know protections come in several forms. One is an explicit challenge with immediate effect; you must answer now to continue. Another is an explicit challenge with delayed effect; if you answer incorrectly now, the game becomes unplayable later. Yet another is an implicit challenge; in order to proceed, you should perform an action as described in the manual, but the game will *appear* to be playable without it.

Infocom were infamous for their use of all three:

Starcross issued a direct challenge with immediate effect, and you could not even leave the second room without typing the correct co-ordinates from the star chart.[37]

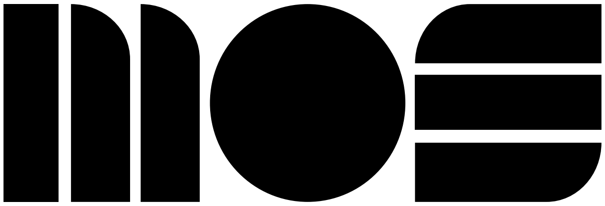
Spellbreaker[38] issued a direct challenge with delayed effect, along the lines of "name the wizard who..." Any name from their word list is accepted, but an incorrect answer results in the player receiving the wrong key. This key cannot unlock a critical door much later in the game, causing the character to be killed instead.

Border Zone made use of an implicit challenge. It required reading the manual in order to know the correct words to excuse yourself — Oopzi Dazi![39]— after bumping into someone, in order to establish contact with the friendly spy. Failure to make contact within the allotted time ended the game.

Brøderbund's Prince of Persia had a variety of delayed effects, depending on which of the several copy protection checks failed. One of them included crashing immediately before showing the closing scene upon winning the game. That is, after completing *fourteen levels*!

However, the What You Have is perhaps the more interesting, given the vast number of possibilities.

7.5 Accept your limitations



The first important component that we will consider in the Apple][is the MOS 6502 or 65C02 CPU. These CPUs have no separation of code and data. That is, they are a Von Neumann, not Harvard architecture. All memory and I/O addresses are executable, and everything that is not in ROM is writable, including the stack.

Since the stack is writable directly, it introduces the possibility of tricks relating to transfer of control. (§7.14.) Since the stack is executable, it introduces the possibility of hosting code. (§7.18.5.)

The CPU has no prefetch queue, only a single prefetched byte of the next instruction (which is why the minimum instruction execution time is two cycles—one for the instruction, and one for the prefetch), as the last stage in the execution of the current instruction. This introduces the possibility of self-modifying code, including the next instruction to execute, because any memory write will have completed before the prefetch occurs. (§7.15.2.)

7.6 Lay it out for me

The second important component that we will consider in the Apple][is the Disk][controller. The Disk][controller is a peripheral which is placed in a slot. It exposes an interface through memory-mapped I/O, so the various soft-switches can be read and written, just like regular RAM. The interface looks like accesses to `$C0sX`, where `s` is `#$80` plus the slot times 16, and `X` is the switch to access.

The Disk][controller runs independently of the CPU. Once the drive is turned on and spinning the disk, the drive will continue to spin the disk until the drive is turned off again. The drive rotates the disk at a fixed speed—approximately 300 RPM, and five rotations per second, which works out to be 200ms per rotation. However, the speed varies somewhat from drive to drive. For 5.25" disks, the data density is equal across all tracks. At 300 RPM, each

[37] `http://infocom.elsewhere.org/gallery/starcross/starcross-map.gif`

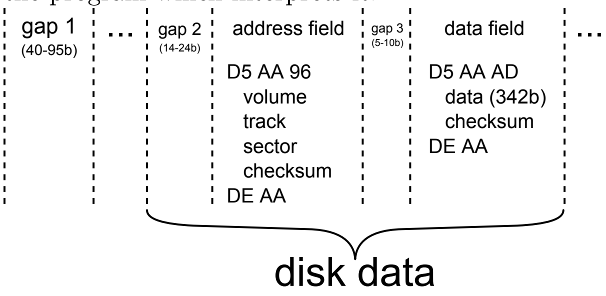
[38] `http://gallery.guetech.org/spellbreaker/spellbreaker.html`

[39] `http://infodoc.plover.net/manuals/temp/borderzo.pdf` p19

track holds 50000 bits, which is equal to 6250 8-bit nibbles.

The data on a disk is simply a stream of bits to be read. For a 5.25" disk, those bits are usually gathered into 16 sectors of 256 bytes each, spread across 35 tracks—$256 \times 16 \times 35 = 143,360$ bytes, or 140kb. When reading from a disk, the Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. As a result, programmers must count CPU cycles carefully to avoid missing nibbles fetched by the controller.

The Disk][controller cannot tell you on which track the head resides. It also cannot tell you on which sector the head resides. (The Shugart SA400 on which the Disk][controller is based does have this capability via index detector circuits, but that feature was removed from the Disk][controller to reduce the cost to manufacture it.) As a result, sectors are usually prepended with a structure known as the "address field", which holds the sector's track and sector number. The controller does not need or use this information. Only the boot PROM makes use of it when requested to read a sector. Beyond that, the information exists solely for the purpose of the program which interprets it.



Following the address field that defines a sector's location on the disk, there is another structure known as the "data field", which holds the sector body. One reason for the separate address and data fields is to allow the sector body to be skipped, as opposed to stored and then decoded, in the event that the sector address is not the desired one. Another reason is that it allows a sector to be updated in-place, by overwriting the data field only, instead of rewriting the entire track to update all of the sectors.

(If the sector were a single structure, the CPU time required to verify that the desired sector has been found is so long that the write would begin after the start of the sector body and extend beyond the original end of the sector, overwriting part of the following sector.)

Between the sectors are dead space, which can be filled with a sequence of self-synchronizing values, timing bits, and protection-specific bytes.

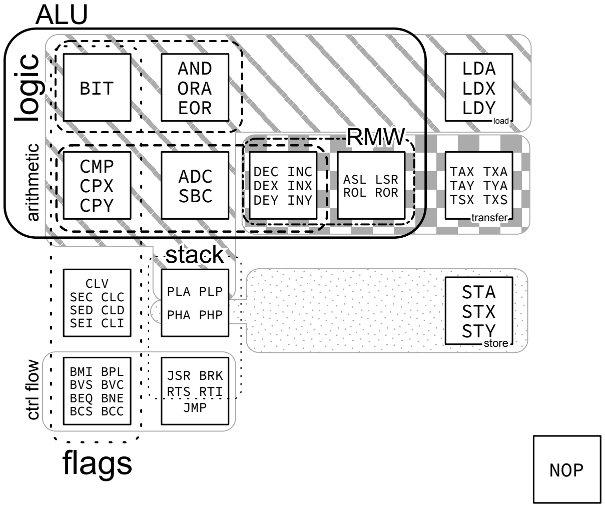
The two structures that define a sector are each bounded by a prologue and an epilogue. The prologues for the address and data fields are composed of three values. Two of those values are never used in the sector body, to distinguish the structures from the sector body, and the third value is different between the two structures, to distinguish them from each other. The epilogues for the address and data fields are composed of two values. One of those values is common to both epilogues but never used in the sector body, to distinguish it from the sector data.

The Disk][controller cannot even tell you where it is within the bitstream. The problem is that the stream does not have an explicit start and end. Instead, a specific sequence must be laid on the track, to form an implicit start. That way, the hardware can find the start of the stream reliably. These values are the "self-synchronizing values." For DOS 3.3, and systems with a compatible sector format, the self-synchronising values are composed of a minimum of five ten-bit "`FF`"s. A ten-bit "`FF`" is eight bits of one followed by two bits of zero. Self-synchronising values are usually placed before both structures that define a sector, to allow synchronisation to occur at any point on the disk. However, this is not a requirement if read-performance is not a consideration.[40] That is, the fewer the number of self-synchronizing values that are present, the more data that can be placed on a track. However, the fewer the number of self-synchronizing values that are present, the more the controller must read before it can enter a synchronized state, and then start

[40] It is a requirement if the data field can be written independently of its address field. Since the write is not guaranteed to begin on a byte boundary, the self-synchronizing values are required for the controller to synchronize itself when reading the data again.

to return meaningful data.

Finally, the Disk][controller can write—but not read reliably—arbitrary eight-bit values. Instead, for reading each eight-bit value, only seven of the bits can be used—the top bit must always be set, in order for the hardware to know when all eight bits have been read, without the overhead of having to count them. (See §7.10.15 for a deeper discussion about an effect made possible by the lack of a counter.) In addition to requiring the top bit to be set, there should not be more than two consecutive zero-bits in a row for the modern drive. (The original disk system did not allow even that. See §7.10.13 for a deeper discussion about the effect of excessive zeroes⊛)



7.7 Copy me, I want to travel

Now that we understand the format of data on the disk, we consider the ways in which that data can be copied.

First is the sector-copier. It relies on sectors being well-defined, and requires knowing only the values for the prologues and epilogues. The sectors are copied one at a time in sequential order, for each of the tracks on the disk, discarding the data between the sectors, and writing new self-synchronizing values instead. Some sector-copiers rely on DOS to perform the writing. In order for that to work, the disk must be formatted first, because that kind of sector-copier will not write new address fields to the disk. Instead, it will reuse the existing ones, since only the data field needs to be updated to place a sector on a track. In any case, the sector-copier cannot deal easily with deviations from the standard format, and requires a lot of interaction to copy sectors for which the prologue and/or epilogue values are not constant. Some sector-copiers can be directed to ignore the sectors that they cannot read, but obviously this can lead to important data being missed.

Second is the track-copier. It also relies on sectors being well-defined, with known the values for the prologues and epilogues. However, it reads the sectors in the order in which they arrive, and then writes the entire track in one pass[41], by itself. It shares the same limitations as the sector-copier regarding reading sectors and discarding the data between them, but it keeps the sectors in the same order as they were originally, which can be important. (§7.10.9.)

Third is the bit-copier. Unlike the previous two, it makes as few assumptions as possible about the data on the disk. Instead, it treats tracks as the bitstream that they are, and attempts to measure the length of the track while reading.[42] It intends to write the track exactly as it appears on the disk, including the data between the sectors, in one pass. Some bit-copiers can be directed to copy the additional zero-bits in the stream, but there is a limit to how reliably these bits can be detected, and the method to detect them can be exploited. Some bit-copiers can be directed to attempt to reproduce the layout of the disk across track boundaries. See sections 7.10.12 and 7.11.3.

The most important point about copiers in general is that there is simply no way to read data off of a disk with 100% accuracy, unless you can capture the complete bitstream on the disk itself, which can be done only with specialised hardware. There is no way for software alone to read all of the bits explicitly and understand how the controller will behave while parsing them⊛

[41] As opposed to reading the sectors in sequential order, and then writing the entire track—that would only make it a sector-copier with a faster write routine.

[42] A sector-copier can use the collection of sectors as a basic track length; the bit-copier has no such luxury. Instead, it is left to "guess", and might be forced to discard or insert additional data to reconstruct a track of the same length. The difference occurs when the rotation speed of the drive that is being used to make the copy is not the same as that of the drive that was used to make the original.

7.8 Super-super decoder ring

Despite the quite strict requirements regarding the format of data on the disk, DOS introduced two additional requirements regarding the format of data within a sector. The first requirement is that there must not be more than one pair of zero-bits in the value. The second requirement is that there be at least one pair of consecutive one-bits, excluding the sign bit.

If we ignore the DOS requirements for the moment, and consider instead all possible values which comply with the hardware requirement to have no more than two consecutive zero-bits, then there are 81 legal values.

```
10010010 (92)  10101101 (AD)  11001110 (CE)  11101011 (EB)
10010011 (93)  10101110 (AE)  11001111 (CF)  11101100 (EC)
10010100 (94)  10101111 (AF)  11010010 (D2)  11101101 (ED)
10010101 (95)  10110010 (B2)  11010011 (D3)  11101110 (EE)
10010110 (96)  10110011 (B3)  11010100 (D4)  11101111 (EF)
10010111 (97)  10110100 (B4)  11010101 (D5)  11110010 (F2)
10011001 (99)  10110101 (B5)  11010110 (D6)  11110011 (F3)
10011010 (9A)  10110110 (B6)  11010111 (D7)  11110100 (F4)
10011011 (9B)  10110111 (B7)  11011001 (D9)  11110101 (F5)
10011100 (9C)  10111001 (B9)  11011010 (DA)  11110110 (F6)
10011101 (9D)  10111010 (BA)  11011011 (DB)  11110111 (F7)
10011110 (9E)  10111011 (BB)  11011100 (DC)  11111001 (F9)
10011111 (9F)  10111100 (BC)  11011101 (DD)  11111010 (FA)
10100100 (A4)  10111101 (BD)  11011110 (DE)  11111011 (FB)
10100101 (A5)  10111110 (BE)  11011111 (DF)  11111100 (FC)
10100110 (A6)  10111111 (BF)  11100100 (E4)  11111101 (FD)
10100111 (A7)  11001001 (C9)  11100101 (E5)  11111110 (FE)
10101001 (A9)  11001010 (CA)  11100110 (E6)  11111111 (FF)
10101010 (AA)  11001011 (CB)  11100111 (E7)
10101011 (AB)  11001100 (CC)  11101001 (E9)
10101100 (AC)  11001101 (CD)  11101010 (EA)
```

If we introduce the first of the DOS requirements that there not be more than one pair of zero-bits, then there are only 72 compliant values, as we see here:

```
10010101 (95)  10110010 (B2)  11010010 (D2)  11101011 (EB)
10010110 (96)  10110011 (B3)  11010011 (D3)  11101100 (EC)
10010111 (97)  10110100 (B4)  11010100 (D4)  11101101 (ED)
10011010 (9A)  10110101 (B5)  11010101 (D5)  11101110 (EE)
10011011 (9B)  10110110 (B6)  11010110 (D6)  11101111 (EF)
10011101 (9D)  10110111 (B7)  11010111 (D7)  11110010 (F2)
10011110 (9E)  10111001 (B9)  11011001 (D9)  11110011 (F3)
10011111 (9F)  10111010 (BA)  11011010 (DA)  11110100 (F4)
10100101 (A5)  10111011 (BB)  11011011 (DB)  11110101 (F5)
10100110 (A6)  10111100 (BC)  11011100 (DC)  11110110 (F6)
10100111 (A7)  10111101 (BD)  11011101 (DD)  11110111 (F7)
10101001 (A9)  10111110 (BE)  11011110 (DE)  11111001 (F9)
10101010 (AA)  10111111 (BF)  11011111 (DF)  11111010 (FA)
10101011 (AB)  11001010 (CA)  11100101 (E5)  11111011 (FB)
10101100 (AC)  11001011 (CB)  11100110 (E6)  11111100 (FC)
10101101 (AD)  11001101 (CD)  11100111 (E7)  11111101 (FD)
10101110 (AE)  11001110 (CE)  11101001 (E9)  11111110 (FE)
10101111 (AF)  11001111 (CF)  11101010 (EA)  11111111 (FF)
```

If we introduce the second of the DOS requirements that there be at least one pair of consecutive one-bits, excluding the sign bit, then there are only 64 compliant values:

```
10010110 (96)  10110100 (B4)  11010110 (D6)  11101101 (ED)
10010111 (97)  10110101 (B5)  11010111 (D7)  11101110 (EE)
10011010 (9A)  10110110 (B6)  11011001 (D9)  11101111 (EF)
10011011 (9B)  10110111 (B7)  11011010 (DA)  11110010 (F2)
10011101 (9D)  10111001 (B9)  11011011 (DB)  11110011 (F3)
10011110 (9E)  10111010 (BA)  11011100 (DC)  11110100 (F4)
10011111 (9F)  10111011 (BB)  11011101 (DD)  11110101 (F5)
10100110 (A6)  10111100 (BC)  11011110 (DE)  11110110 (F6)
10100111 (A7)  10111101 (BD)  11011111 (DF)  11110111 (F7)
10101011 (AB)  10111110 (BE)  11100101 (E5)  11111001 (F9)
10101100 (AC)  10111111 (BF)  11100110 (E6)  11111010 (FA)
10101101 (AD)  11001011 (CB)  11100111 (E7)  11111011 (FB)
10101110 (AE)  11001101 (CD)  11101001 (E9)  11111100 (FC)
10101111 (AF)  11001110 (CE)  11101010 (EA)  11111101 (FD)
10110010 (B2)  11001111 (CF)  11101011 (EB)  11111110 (FE)
10110011 (B3)  11010011 (D3)  11101100 (EC)  11111111 (FF)
```

That leaves us with eight values for which there is not more than one pair of zero-bits, but also not one pair of consecutive one-bits, excluding the sign bit. DOS reserves some of these value for a separate purpose.

```
10010101 (95)
11010010 (D2)
11010100 (D4)
11010101 (D5)
10100101 (A5)
10101001 (A9)
10101010 (AA)
11001010 (CA)
```

That leaves us with 17 values for which there are not more than two consecutive zero-bits, which seems like a missed opportunity for a better encoding:

```
10010010 (92)  10101001 (A9)  11100100 (E4)
10010011 (93)  10101010 (AA)
10010100 (94)  11001001 (C9)
10010101 (95)  11001010 (CA)
10011001 (99)  11001100 (CC)
10011100 (9C)  11010010 (D2)
10100100 (A4)  11010100 (D4)
10100101 (A5)  11010101 (D5)
```

Having exactly 64 entries in the table allows us to represent all of the values using six bits. That leads us to an encoding method known as “6-and-2 Group Code Recording (GCR)” or more commonly “6-and-2” encoding.

In “6-and-2” encoding, an eight-bit value is split into two parts, where the high six bits are separated from the low two bits. (The disk system for which DOS 3.2 was first written had an additional restriction that did not allow consecutive zero-bits, and so used “5-and-3” encoding for the same purpose.) To encode an entire sector, each of the two-bit values are gathered together, such that three of them form another six-bit value in reverse order, and are stored first, followed by each of the regular six-bit values. Prior to storing any of the values, they must be transformed into the values in our table of 64 nibbles. This is done by using the original value as an index into the nibble table, and writing the value from the table instead.

When we place the original value beside the nibble value, the table looks like this:

00 = 96	10 = B4	20 = D6	30 = ED
01 = 97	11 = B5	21 = D7	31 = EE
02 = 9A	12 = B6	22 = D9	32 = EF
03 = 9B	13 = B7	23 = DA	33 = F2
04 = 9D	14 = B9	24 = DB	34 = F3
05 = 9E	15 = BA	25 = DC	35 = F4
06 = 9F	16 = BB	26 = DD	36 = F5
07 = A6	17 = BC	27 = DE	37 = F6
08 = A7	18 = BD	28 = DF	38 = F7
09 = AB	19 = BE	29 = E5	39 = F9
0A = AC	1A = BF	2A = E6	3A = FA
0B = AD	1B = CB	2B = E7	3B = FB
0C = AE	1C = CD	2C = E9	3C = FC
0D = AF	1D = CE	2D = EA	3D = FD
0E = B2	1E = CF	2E = EB	3E = FE
0F = B3	1F = D3	2F = EC	3F = FF

DOS reserved two values from our fourth table—#$AA and #$D5—for the prologue signatures. These values are good candidates for the purpose of identifying the headers, because they do not conform to the "at least one pair of consecutive one-bits" criterion, and thus do not conflict with the entries in the "nibbilisation" table. It is not a coincidence that they have alternating bit values; #$D5 is #$55 without the sign bit. By reserving these values, it ensures that the bitstream generated by arbitrary sector data cannot contain a long string of ones (prevented by reserving #$FF), or alternating zeroes and ones (prevented by reserving #$AA and #$D5), regardless of the user's data.

The third value of the prologue signature (#$96 or #$AD) need be unique only between the headers, in order to distinguish between the two. The combination of unique values and non-unique values still produces a unique sequence.

DOS reserved one value from our fourth table—#$AA—for the second byte of the epilogue signatures, for the same reason as for the prologue. The first byte of the epilogue signature need not be unique with respect to sector data (because the combination of unique values and non-unique values still produces a unique sequence), but obviously it must not match the first byte of the prologue, because the third byte of the epilogue (intended to be #$EB) is written sometimes with only limited success (and it is never verified for this reason), and so could potentially be read as the third byte of a prologue instead, with unpredictable results.

The decoding process requires a reverse transformation, via a table which is typically filled with all of the values in a six-bit number. (See the sections on Race Conditions and SpiraDisc for two counter-examples.) The layout of the table is the special thing, though—the nibbles that are read from disk are used as an index into the table, in order to recover the original six-bit value. So the table has gaps between some of the values, because the legal values of the nibbles are not consecutive.

Note that convention is a powerful force. There is no reason for the table to have the nibbilisation entries in that order, or to exclude #$AA or #$D5 (or any of the other 15 entries from the last table) from the set. Further, according to John Brooks, it is possible to use all 81 values from our first table, combined with a special encoding method, which would increase the data density by 105.5%, and potentially even more.[43]

7.9 Write-protection

The absolute simplest possible protection against a copy is to check if the disk is write-protected. The vast majority of owners of duplicated software won't bother to write-protect the disk. If the disk is not write-protected, then the image is considered to be a copy, rather than the original.

Alien Addition uses this technique.

```
;assumes slot 6
7975    LDA   $C0ED       ;request status
7978    LDA   $C0EE       ;read status
797B    BPL   $7985       ;taken if write-
    enabled
```

A more generic version of the technique is slightly longer:

```
0000    LDX   $2B         ;fetch slot (x16)
0002    LDA   $C08D, X    ;request status
0005    LDA   $C08E, X    ;read status
0008    BPL   $0008       ;hang if write-
    enabled
```

7.10 Sector-level protections

7.10.1 Altered prologue/epilogue

This is one of the simpler techniques available, and was used by many titles. Standard DOS 3.3 uses

[43] http://www.bigmessowires.com/2015/08/27/apple-ii-copy-protection/#comment-227325

the sequence #$D5 #$AA #$96 to identify the address field prologue, #$D5 #$AA #$AD to identify the data field prologue, and #$DE #$AA to identify both of the epilogues. Of course, it is possible to choose from the 17 values from our fifth table, for either the first two bytes of the prologue values, or the second byte of the epilogue. It is also possible to choose from among the 81 values from our first table, for either the third byte of the prologue, or the first byte of the epilogue.

Most commonly, only one value is changed in the prologue or epilogue, and that same value is used for every sector on every track of the disk.

Lucifer's Realm uses this technique; the epilogue was changed from #$DE #$AA to #$DF #$AA.

The Tracer Sanction extended the technique by carrying a table of values, and using a different value for each track.

Masquerade extended the technique to the sector level, by requiring that each even sector has one value, and each odd sector has another value. The routine extracts bit zero of the sector number, and then inverts it, to create the key which is applied to the identification byte. Thus, even sectors use #$D5 (the standard value), and odd sectors use #$D4. This is necessary because sector zero of track zero must have the regular value in order to be readable by the boot PROM.

The Coveted Mirror used exactly the same technique–and almost the exact same code–at only the track level.

Due to size limitations, the boot PROM does not verify the epilogue bytes[44] allowing all sectors on all tracks—including the boot sector itself—to be protected. The most common technique involved altering the epilogue values to something other than the default value. This protection cannot be reproduced by a sector-copier or track-copier, which requires the default values to be seen, because they will fail to copy the sector. Operation Apocalypse uses this technique.

Given that the boot PROM does not verify the epilogue bytes, a very light protection technique is to change the epilogue values to something other than the default values for sector zero of track zero only, leaving all other sectors readable. This protection cannot be reproduced by a sector-copier or track-copier which requires the default values to be seen, because they will fail to copy the boot-sector, leaving the disk unusable. Alien Addition makes use of this technique.

A common technique to defeat this protection is to ignore read errors for all sectors, in the hope that it is caused by the non-default epilogue values alone. However, given the degrading state of floppy disks these days, ignoring read errors can hide the fact that the disk is truly failing.

The address field contains more than just the track and sector numbers. It also contains a volume number. This value can be used as a quick method to determine which disk from a set is currently inserted into the drive. However, support for it—even in DOS—is poor. So many programs, including DOS itself, assume that the volume number is the default value. When it is changed, the read fails. By hard-coding the new value in DOS, the disk will be readable only by itself. Algebra Arcade uses this technique.

This technique can also be used in a slightly different way. Since each sector can have its own volume number, any value can be put there, as long as the program is aware of that fact.

Randamn sets the volume number to a checksum calculated from the current track and sector, and hangs if the values do not match.

Both the address field and data field contain a checksum of the data that precede it, prior to the epilogue. The checksum algorithm is usually a rolling exclusive-OR of each of the bytes, with a zero seed. However, there is no requirement that either of these things is used, for sectors other than sector zero of track zero. For other sectors, the seed can be set to any value, and the algorithm can be a cumulative ADD or anything else at all. This protection cannot be reproduced by a sector-copier or track-copier which relies on the regular algorithm, because the disk will appear to be corrupted.

Hellfire Warrior uses a slight variation on this technique. It maintains a counter at address $40, which coincides with the track number which is stored by the boot PROM. In order to break out of the loop that reads sectors into memory, the program requests the boot PROM to read a sector with an intentionally bad checksum. This causes the boot PROM to rewrite the value at address $40. The new value is exactly what the program requires as the exit condition. This protection cannot be reproduced by a sector-copier or track-copier, because they will fail to copy this sector, resulting in a disk that has only sectors with good checksums. The disk

[44] It also ignores the address field checksum and volume number.

will not boot because it will never exit the loop.

The volume number is normally an eight-bit value. For efficiency of encoding it, DOS uses a “4-and-4” encoding, where the four odd bits are separated from the low even bits, and converted to nibbles. To recombine them, it is a simple matter to shift the nibble holding the odd bits (“abcd”) one to the left, resulting in an encoding that looks like “a1b1c1d1”, and then to `AND` the result with the nibble holding the even bits (“efgh”), whose encoding that looks like “1e1f1g1h”. This method requires 16 bytes to describe the address field. Since the track, sector, and checksum, are known to fit into six bits each, it is easy to see that if the volume number is disregarded, a “6-and-0” encoding can be used instead. This method requires only four nibbles to describe the address field. Algernon uses this technique.

The entries in the address field have a defined order because the boot PROM needs to read them to identify sector zero of track zero, and any other sector which the PROM is asked to read. However, it is possible to change the order of the entries for other sectors on the disk, and then to read the sectors manually.

7.10.2 Fewer sectors

The major reason for using 16 sectors per track is because that is the maximum number that can fit within the standard format created by DOS 3.3. DOS 3.2 supported only 13 sectors per track, because of the limitation of the hardware regarding consecutive zeroes. Copy protection techniques are free to use fewer sectors than either of those values.

Wavy Navy uses ten sectors per track, while Olympic Decathlon uses eleven and Karateka uses a dozen. The sectors in these examples are all the regular size, but encoded in a wasteful manner. (Primarily the “4-and-4” encoding was used because the decoder is very small, but sometimes “5-and-3” because the decoder looks weird when compared with the more familiar “6-and-2” encoding.) The wasteful encoding is the reason for the reduced sector count; there really isn’t more room for more sectors.

7.10.3 More sectors

The standard DOS 3.3 format disk uses 16 individual sectors per track, with relatively large gaps between the sectors. Consider how much space would be available if those sectors were combined into a single large sector, with a single field that combines both address (specifically, only the track number) and data fields. Yes, it would require reading the entire track in order to find the field again once the track had been verified, but for some applications, performance is not that critical. This is what Infocom did, on programs such as A Mind Forever Voyaging. Once the track had been found, and the data field found again, then the program read (and discarded) sectors sequentially until the required one was found. Again, if the performance is not that critical, the fact that the routine can fetch only one sector at a time is not an issue. In fact, the implementation works well enough for the text-adventure scenario in which it was used. Since the user will be reading the text while additional text is loading, the time required for that loading goes mostly unnoticed.

Consider how much space would be available if those gaps were reduced to the minimum of five self-synchronizing values before the address field prologue, with just a few bytes of gap between the address and data headers. Then reducing the prologue byte count from three to two, and the epilogue byte count from two to one. Consider how much space would be available by merging groups of sectors. If you converted the track into six sectors of three times the size, you would have RWTS18. This is a good compromise between speed and density. On one side, having fewer sectors means less processing; and on the other side, having more sectors means less latency to find a sector. The RWTS18 routine also supports “read scattering” by assigning a dummy write address to the pages that aren’t needed.

This second technique was used very heavily by Brøderbund, on programs such as Airheart (and even three years later, on Prince of Persia), but other companies made use of it, too, such as Infogrames in Hold-Up. Interestingly, in the case of Airheart, after compressing the title screen to reduce its size on the disk, the rest of the game fit on a regular 16-sector disk.

7.10.4 Big sectors

There is no requirement to define multiple sectors per track. It is possible to define a single sector that spans the entire track.[45] However, there can be a significant time penalty while reading such a track, because it requires up to one complete rotation in order to find the start of the sector.

Lady Tut uses a single sector per track, at a size equivalent to eleven 256-bytes sectors.

7.10.5 Encoded sectors

As noted previously, there is no reason for a disk to use our sixth table—there is no reason to have the nibbilisation entries in that order, nor even to use those values at all. Any alteration to the table results in a disk that can be copied freely, but whose contents cannot be read from the outside. Further, the DOS on such a disk cannot write files from the inside to the outside. The reason why the read would fail is because the standard table would be applied to data that requires the alternative table to decode, resulting in the wrong decoding. The reason why the write would fail is because the alternative table would be applied to data that requires the standard table to encode, resulting in the wrong encoding.

Maze Craze Construction Set uses an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

Bop'N Wrestle uses the regular nibble table (and a standard DOS 3.3), but in reverse order.

7.10.6 Duplicated sectors

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have multiple sectors with the same number.[46] There are numerous ways in which they could be distinguished, such as by the volume number. A protection technique could set every sector number to the same value in the address field. It could set them all to zero, provided that the checksum algorithm is changed, so that the boot PROM will read successfully only the true sector zero, in order to boot the disk. It could also use the volume number from the address field as the page number in which to write the sector data. This would be a very compact way to load data without the need to pass the address as a parameter to the loader.

Math Blaster has two sectors numbered zero on track zero. The program distinguishes between them by examining the first nibble after the address field epilogue, but the checksum of the second sector zero also fails verification, which is why the boot PROM does not see it. This protection cannot be reproduced by a sector-copier or track-copier, because those copiers will write only a single sector zero to a track. It is unpredictable which of the two sector zeroes would be written, but even if the true one is chosen, the copy is revealed by the program missing the duplicated sector.

7.10.7 Sector numbering

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have sectors whose number is not in the range of zero to 15.[47] Any eight-bit value can be used, as long as the program is expecting it. This protection cannot be reproduced by a sector-copier, because the copier will not copy those sectors at all.

7.10.8 Sector location

The address field carries the track and sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible for a sector to "lie" about its location on the disk. For example, the address field of sector three on track zero could label itself as sector zero on track three. This protection cannot be reproduced by a sector-copier which relies on DOS to perform the write, because they will

[45] This would be the equivalent of about 18.5 256-bytes sectors in "6-and-2" encoding. Using 19 sectors is possible, if the full range of values from the first figure is used, but it introduces a problem to identify the start of the sector, since there are no single values that can be reserved exclusively. One possible solution is to find a sequence which cannot appear in user-data due to particular characteristics of the decoding process. Just because it is possible, it doesn't mean that it's easy.

[46] The same is true for the track number, and Jumble Jet has multiple tracks which claim to be track zero.

[47] The same is true for the track number. That is, a number which is not in the range of zero to 34.

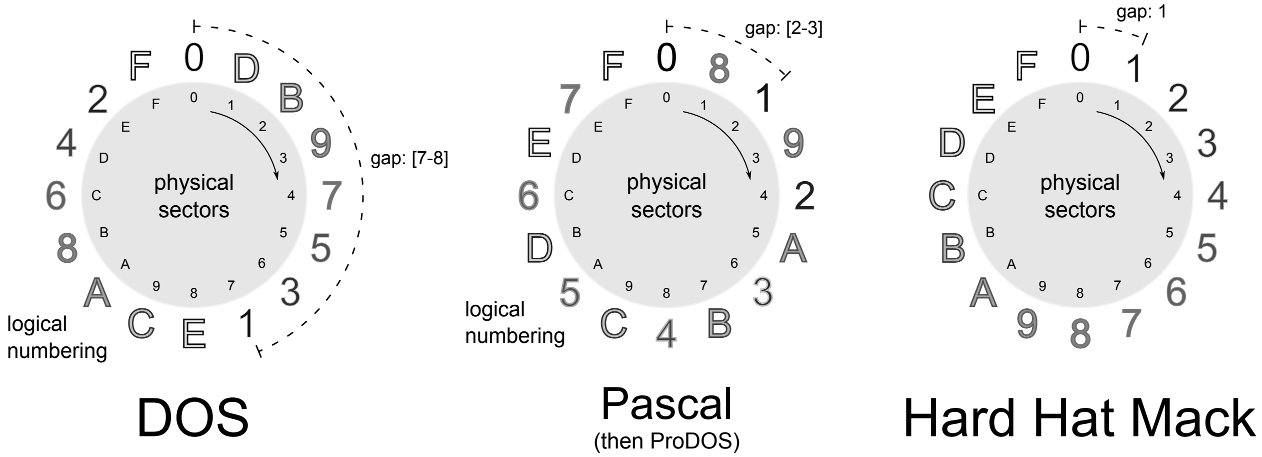


Figure 16 – Floppy sectors interleaving.

not duplicate this information, because DOS will fill in the address field by itself when placing the sector on the disk. Thus, a program that seeks to a track that contains "misplaced" sectors will not find any misplaced sectors, or will receive the wrong content instead.

Discover uses this technique; it changes the identity of one particular sector in the sector interleave table, on one particular track.

7.10.9 Synchronised sectors

Since the approximate rotation speed of the drive is known (∼300 RPM), it becomes possible to place sectors at specific locations on a track, such that they have a special position relative to other sectors on the same track. This is difficult to reproduce because of the delay that is introduced while a sector-copier is writing the data.

Hard Hat Mack takes this to the extreme, by requiring that one track has all 16 sectors in incremental order. This protection is highly unlikely to be reproduced by using a sector-copier, because after factoring in the rotation speed of the drive, the next sector is more likely to be placed halfway around the disk.

7.10.10 Bad sectors

Some protections rely on the fact that intentionally bad sectors (for example, checksum mismatch in the simplest case, but potentially physical damage could be used, too) should return a read error.

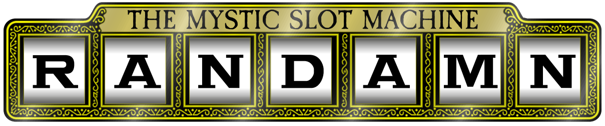
Drelbs uses this technique. This protection cannot be reproduced even with a bit-copier, because the copy will have no sectors that cannot be read.

7.10.11 Dead-space bytes

The data for a sector is well defined, but apart from the optional presence of the self-synchronizing values, the data between sectors is not defined at all. As a result, it is not often copied, either. It is possible to place specific counts of specific values in this location, which can be checked later. A program can detect a copy by the absence or wrong count of the special values.

Randamn checks the value of the byte immediately before the prologue of a particular sector, and reboots if the value looks like a self-synchronizing value. (A bit-copier might insert this values when asked to match the track length, and a sector-copier would always insert the value.)

Binomial Multiplication counts the number of values that appear between the address field epilogue and the data field prologue, and between the data field epilogue and the next sector address field prologue, for all of the sectors on a particular track. This protection cannot be reproduced by a sector-copier or a track-copier, because those copiers will discard the original data between the sectors.



7.10.12 Timing bits

The Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. The significant part of that statement is "once a one-bit is seen." It is possible to intentionally introduce "timing" (zero) bits into the stream in order to delay the reset. For each zero-bit that is present, the previous value will be held for another eight CPU cycles. For code that is not expecting these zero-bits to be present, a nibble that is being held back will be indistinguishable from a nibble that has newly arrived.

Creation uses this technique. It looks like this:

```
;wait for nibble to arrive
B94F    LDA   $C08C,X
B952    BPL   $B94F
;watch for #$D5
B954    CMP   #$D5
B956    BNE   $B948
;delay to ensure > 4 cycles will elapse
;before the next read occurs
B958    NOP
;read data latch
B959    LDA   $C08C,X
;check if nibble has changed
;if zero-bit is present,
;then read value lasts longer
B95C    CMP   #$D5
B95E    BEQ   $B972
```

Hacker II requires a pattern of zero-bits to be present in the stream. The effect of the delayed shift becomes clear when we count cycles.

```
;initialise mask
403A    LDA   #$08
...
;wait for nibble to arrive
4044    LDY   $C08C,X
4047    BPL   $4044         ;2 cycles
;watch for #$FB
4049    CPY   #$FB          ;2 cycles
404B    BNE   $403A         ;2 cycles
;not a do-nothing instruction!
;exists to be timing-identical
;to the BEQ at $4062
404D    BEQ   $404F         ;3 cycles
404F    NOP                 ;(2 cycles)
4050    NOP                 ;(2 cycles)
;read data latch
4051    LDY   $C08C,X       ;(4 cycles)
;check how many bits have shifted in
4054    CPY   #$08
;shift carry into A
4056    ROL
;until a set bit is shifted out
;(takes five rounds)
4057    BCS   $4064
;wait for nibble to arrive
4059    LDY   $C08C,X
405C    BPL   $4059         ;2 cycles
;watch for #$FF
405E    CPY   #$FF          ;2 cycles
4060    BNE   $403A         ;2 cycles
4062    BEQ   $404F         ;3 cycles
;wait for nibble to arrive
4064    LDY   $C08C,X
4067    BPL   $4064
;remember its value
4069    STY   $07
;check if proper pattern was seen
;(alternating zero-bit yes and no)
406B    CMP   #$0A
406D    BNE   $403A
;wait for nibble to arrive
406F    LDA   $C08C,X
4072    BPL   $406F
;checksum against previous value
;both must be #$FF to pass
4074    SEC
4075    ROL
4076    AND   $07
4078    EOR   #$FF
407A    BEQ   $4080
```

The timing loop is long enough for four nibbles to be shifted in if no zero-bit is present, resulting in a value of at least `#$08`. (Specifically the right-hand "`F`" from the value "`FF`".) If a zero-bit is present, then fewer than four nibbles will be shifted in, resulting in a value of less than `#$08`. This explains the "`CPY #$08`" instruction at `$4054`. It is checking if a one-bit has been shifted in four times or three times.

The "`CMP #$0A`" instruction at `$406B` is checking the final results of the multiple `CPY`s that were made. In binary, the results look like `01010` but prior to that, the results progress like this:

0001000**0**
001000**01**
01000**010**
1000**0101**
000**01010**

That means it is expecting the first pass to have a value of less than eight (carry clear), then a value of at least eight (carry set), then a value of less than eight (carry clear), then a value of at least eight (carry set), and finally a value of less than eight (carry clear), followed by two "`FF`"s. That requires the stream to look like `FB 0 FF FF 0 FF FF 0 Fx`

FF FF⊗

7.10.13 Floating bits

What happens if more than two consecutive zero-bits are present in a stream? Something random. The Automatic Gain Control circuit will eventually insert a one-bit because of amplified noise. It might happen immediately after the second zero-bit, or it might happen after several more zero-bits. The point is that reading that part of the stream repeatedly will yield different responses⊗

Mr. Do! uses this technique.

```
;set counter to be used later
0710    LDY    #$06
...
;set state
0713    LDA    #$FF
0715    STA    $07C2
;wait for nibble to arrive
0718    LDA    $C088,X
071B    BPL    $0718
;watch for #$D5
071D    CMP    #$D5
071F    BNE    $0718
;wait for nibble to arrive
0721    LDA    $C088,X
0724    BPL    $0721
;watch for #$9B
0726    CMP    #$9B
0728    BNE    $071D
;wait for nibble to arrive
072A    LDA    $C088,X
072D    BPL    $072A
;watch for #$AB
072F    CMP    #$AB
0731    BNE    $071D
;wait for nibble to arrive
0733    LDA    $C088,X
7036    BPL    $0733
;watch for #$B2
0738    CMP    #$B2
073A    BNE    $071D
;wait for nibble to arrive
073C    LDA    $C088,X
073F    BPL    $073C
;watch for #$9E
0741    CMP    #$9E
0743    BNE    $071D
;wait for nibble to arrive
0745    LDA    $C088,X
0748    BPL    $0745
;watch for #$BE
074A    CMP    #$BE
074C    BNE    $071D
;wait for nibble to arrive
074E    LDA    $C088,X
0751    BPL    $074E
;loop six times
0753    DEY
0754    BNE    $074E
;change state
0756    INC    $07C2
0759    BNE    $2761
;store last read value on first pass
075B    STA    $07C3
;allow complete revolution and read again
075E    JMP    $071D
;check last read value on subsequent pass
;must be different from the first pass
0761    CMP    $07C3
0764    BNE    $0771
;retry up to four times
0766    INC    $07C2
0769    LDA    $07C2
076C    CMP    #$08
076E    BNE    $271D
```

On the first pass, the program watches for the sequence `$#D5 #$9B #$AB #$B2 #$9E #$BE`, skips the next five nibbles, and then reads and saves the sixth nibble. On subsequent passes, the program watches again for the sequence `$#D5 #$9B #$AB #$B2 #$9E #$BE`, skips the next five nibbles, and then reads and compares the sixth nibble against the sixth nibble that was read initially. The value that is read will always be a legal value, but on the original disk, with multiple zero-bits in the stream, the value that was read in one of the subsequent passes will not match the value that was read in the first pass. No matter how many extra zero-bits existed in the stream, the bit-copier will not write them out. Instead, it will "freeze" the appearance of the stream, and normalise it so that there are no more than two zero-bits emitted. As a result, the sixth nibble that was read will have the same value for all passes, and therefore fail the protection check⊗

7.10.14 Nibble count

Since a track is simply a stream of bits, it is possible to control the layout of the values in that stream, as long as it follows the rules of the hardware. The number of self-sychronizing values can be reduced to a single set of the minimum number, if performance is not a consideration. That means there are no other zero-bits present on the track. However, a bit-copier cannot detect the zero-bits reliably (neither their presence, nor their number), so it is left to guess if the value `#$FF` must be stored using eight or ten bits. (That is, if it is a data nibble or a self-synchronizing value.) If there are enough `#$FF` bytes on a track, and if the bit-copier assumes that every one of them must be ten bits wide, then it is possible that the bit-copier will write more data than can fit on the track, resulting in part of the track being overwritten when the revolution completes before the write completes.

As a separate technique, it is also possible to reduce the speed of the drive while writing the data to the original disk, resulting in a track that is so dense, that the data cannot fit on a disk when written at regular speed. This is known as a "fat" track.

The more common technique is to simply use a sequence of nibbles with enough zero-bits between them, that the "delayed fetch" effect is triggered. (§7.10.12.) When the zero-bits are present, and if the fetch is fast enough (that is, it polls the QA switch of the Data Register while the top bit is clear, stores the fetched value, and then resumes polling), then there will appear to be more nibbles of a particular value than really exist, because the next bit will not be ready to shift in. A program that counts the number of nibbles will see more nibbles in the copy than in the original.

If the fetch is slow enough... now, this is an interesting case. Bit-copiers try to read the data as quickly as it comes in. This is done not by polling the QA switch of the Data Register, but by checking if the top bit is already set, in an unrolled loop, like this:

```
;2 cycle delay so
;shift might finish
TDL1      NOP
;try to detect timing bit
LDA $C0EC, X
BMI TDS2
TDL2      LDA $C0EC, X
BMI TDS2
;timing bit probably present
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
;3 cycle penalty if taken!
BPL TDL2
TDS2      STA ($0), Y
...
RTS
;store value with timing bit
;loses one bit as a result
TDS3      AND #$7F
STA ($0), Y
...
RTS
```

This code is a disassembly from Essential Data Duplicator (E.D.D.), but apart from the BPL instruction, it is shared by Copy][+. (Someone copied!) Normally, a nibble will be shifted in before TDL2 completes, so that TDS2 is reached, and the nibble is stored intact. However, by using only six fetches, the code is vulnerable to a well-placed timing bit, such that the BPL will be reached just before the last bit of the nibble is shifted in. That three-cycle time penalty when the branch is taken is just enough that, when combined with the two-cycle instruction before it, the shift will complete, and the four CPU cycles will elapse, before the next read occurs. The result is that the nibble is missed, and the next few nibbles that arrive will reach TDS3 instead, losing one bit each. When those data are written to disk by the bit-copier, the values will be entirely wrong.

Create With Garfield: Deluxe Edition uses this technique. (The original Create With Garfield uses an entirely different protection.) It has one track that is full of repeated sequences. Each of the sequences has a prologue of five bytes in length. Every second one of the prologues has a timing bit after each of the five bytes in the prologue. In the middle of the track is a collection of bytes which do not match the sequence, so the track is essentially split into two groups of these repeated sequences. The size of the two groups is the same. When the bit-copier attempts to read the data, the timing bits cause about half of the sequences to be lost. What remain are far fewer sequences than exist on the original disk. (Enough of them that the bit-copier mistakenly believes that it has copied the track successfully.) A program can detect a copy by the small count of these sequences. This technique is likely to have been created to defeat E.D.Dṡpecifically, but Copy][+ is also affected. However, the protection can be reproduced with the use of a peripheral that connects to the drive controller (and thus see the zero-bits for exactly what they are), or by inserting an additional fetch in the software.

7.10.15 Bit-flip, or defeat bit-copiers with this one weird trick

Deeply technical content follows. Prepare yourself!

Let's take this simple sentence (sorry, but it's the best example that I could create at the time):
ITHASGOTTOBETHISLANDAHEAD

And split it according to some potential word boundaries:
IT HAS GOT TO BE THIS LAND AHEAD

Now we skip a bit:

OTTO BETH ISLAND AHEAD

A bit more:

TO BETH ISLAND AHEAD

A bit more still:

BET HIS L AND A HEAD

Okay, that last one doesn't make much sense, but I wanted a sentence which could be read differently, depending on where you started reading, as opposed to a series of arbitrary overlapping words. In any case, it's clear that depending on where you start reading, you can get vastly different results. Something similar is possible while reading the bitstream from the disk. After a nibble is shifted in (determined by the top bit being set), and the four CPU cycles have elapsed, and once the one-bit is seen, then the QA switch of the Data Register is set to zero. The absence of a counter allows the hardware to be fooled about how many bits have been read. Specifically, the controller can be convinced to discard some of the bits that it has read from the disk while forming a nibble, and then the starting position within the stream will be shifted accordingly. This is possible with a single instruction, in conjunction with an appropriate delay.

After issuing an access of Q6H (`$C08D`+(*slot* × 16)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer (LSS) continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. Let's see that in action.

Tinka's Mazes does it this way, beginning with some preamble code which is common to many programs that used this technique.

```
BB6A    LDY     #0
;wait for nibble to arrive
BB6C    LDA     $C08C,X
BB6F    BPL     $BB6C
BB71    DEY
;retry up to 256 times
BB72    BEQ     $BBBB
;watch for #$D5
BB74    CMP     #$D5
BB76    BNE     $BB6C
BB78    LDY     #0
;wait for nibble to arrive
BB7A    LDA     $C08C,X
BB7D    BPL     $BB7A
BB7F    DEY
;retry up to 256 times
BB80    BEQ     $BBBB
;watch for #$E7
BB82    CMP     #$E7
BB84    BNE     $BB7A
;wait for nibble to arrive
BB86    LDA     $C08C,X
BB89    BPL     $BB86
;watch for #$E7
BB8B    CMP     #$E7
BB8D    BNE     $BBBB
;wait for nibble to arrive
BB8F    LDA     $C08C,X
BB92    BPL     $BB8F
;watch for #$E7
BB94    CMP     #$E7
BB96    BNE     $BBBB
```

Here is the switch:

```
;trigger desync
BB98    LDA    $C08D,X
BB9B    LDY    #$10
;delay to ensure > 4 cycles will elapse
;before the next read occurs
BB9D    BIT    $6
;wait for nibble to arrive
BB9F    LDA    $C08C,X
BBA2    BPL    $BB9F
BBA4    DEY
;retry up to 16 times
BBA5    BEQ    $BBBB
;watch for #$EE
BBA7    CMP    #$EE
BBA9    BNE    $BB9F
BBAB    LDY    #7
;wait for nibble to arrive
BBAD    LDA    $C08C,X
BBB0    BPL    $BBAD
;compare backwards against the list at $BBC1
;E7 FC EE E7 FC EE EE FC
BBB2    CMP    ($48),Y
BBB4    BNE    $BBBB
BBB6    DEY
BBB7    BPL    $BBAD
;pass
BBB9    CLC
BBBA    RTS
BBBB    DEC    $50
;retry if count remains
BBBD    BNE    $BB57
;fail
BBBF    SEC
BBC0    RTS
BBC1    .BYTE    $FC,$EE,$EE,$FC,$E7,$EE,$FC,
    $E7
```

But wait, there's more! To see the bitstream on disk, it looks like `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? Since the magic is in the timing of the reads, we must count cycles:

```
BB8F    LDA    $C08C,X
BB92    BPL    $BB8F     ;2 cycles
BB94    CMP    #$E7      ;2 cycles
BB96    BNE    $BBBB     ;2 cycles
BB98    LDA    $C08D,X   ;4 cycles
BB9B    LDY    #$10      ;2 cycles
BB9D    BIT    $6        ;3 cycles
;total: 15 cycles
```

Time passes...

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it looks like the following, with the seemingly redundant zero-bits in bold.
11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **0** 11100111 11100111
However, by skipping the first three bits, the stream looks like this:
00 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**0**1110 *0* 11111100 111...

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `E7 FC EE E7 FC EE EE FC`, and we have our magic values.

Programs from Epyx that use this protection do not compare the values in the pattern. Instead, the values are used as a key to decode the rest of the data that are loaded. This hides the expected values, and causes the program to crash if they are altered.

The Thunder Mountain version of Dig Dug uses a slight variation on the technique, including a different preamble and switch. The company seems to have kept the variation to themselves. (Bop'N Wrestle from 1986 uses the same altered version, and comes from Mindscape, but Mindscape owned the Thunder Mountain label, so the connection is clear.)[48] That version looks like this:

```
0224 LDY #$00
;wait for nibble to arrive
0226 LDA $C08C,X
0229 BPL $2226
022B DEY
;retry up to 256 times
022C BEQ $2275
022E CMP #$AD
0230 BNE $2226
```

A different prologue value is checked, allowing the bitstream to begin like a regular sector: `D5 AA AD`...

Here is the switch:

```
;trigger desync
0252 LDA $C08D,X
```

[48] Interestingly, one title from Thunder Mountain and released in the same year is known to use the regular version. It is entirely possible that the alternative version was developed in-house to avoid paying royalties to protect other products.

```
0255 LDY #$10
;no delay instruction in this version
;wait for nibble to arrive
0257 LDA $C08C,X
025A BPL $2257
025C DEY
;retry up to 16 times
025D BEQ $2275
;watch for #$E7 instead, but it's not a ``
    true'' E7
025F CMP #$E7
0261 BNE $2257
;and double the size of the pattern to match
0263 LDY #$0F
```

The bitstream on disk looks like `D5 AA AD` [many `96`s] `E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. The desync timing is only 12 cycles, but the required pattern is not found right away, so the delay is not as interesting. In binary, the stream looks like 11100111 11100111 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:
00 11111100 11111100 111**00**111 (← E7, but not aligned) *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111...

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `FC` (ignored) `FC` (ignored) `E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE`, a very smooth sequence indeed. Put simply, each single bold zero-bit sequence results `EE` being seen, and every double bold zero-bit sequence results in `E7` being seen, allowing easy control over exactly how smooth the sequence is.

1-2-3 Sequence Me uses the same technique but with different values:

```
;wait for nibble to arrive
BA5B     LDA    $C08C,X
BA5E     BPL    $BA5B
;watch for #$AA
BA60     CMP    #$AA
BA62     BEQ    $BA7A
...
BA7A     LDY    #$02
;trigger desync
BA7C     LDA    $C08D,X
;delay while status is loaded
BA7F     PHA
;balance stack
BA80     PLA
;wait for nibble to arrive
BA81     LDA    $C08C,X
BA84     BPL    $BA81
;watch for #$BB
BA86     CMP    #$BB
BA88     BEQ    $BA8F
BA8A     DEY
;retry if count remains
BA8B     BPL    $BA81
;fail
BA8D     BMI    $BA77
;wait for nibble to arrive
BA8F     LDA    $C08C,X
BA92     BPL    $BA8F
;watch for #$F9
BA94     CMP    #$F9
BA96     BNE    $BA77
```

That stream looks like `AA EB 97 DF FF` with some harmless zero-bits in between. Now let's count the cycles:

```
BA5B     LDA    $C08C,X
BA5E     BPL    $BA5B        ;2 cycles
BA60     CMP    #$AA         ;2 cycles
BA62     BEQ    $BA7A        ;3 cycles
...
BA7A     LDY    #$02         ;2 cycles
BA7C     LDA    $C08D,X      ;4 cycles
BA7F     PHA                 ;3 cycles
;total: 16 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 16 CPU cycles is enough for four bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it would look like this:
11101011 **0** 10010111 **0** 11011111 **00** 11111111
with the seemingly redundant zero-bits in bold. However, by skipping the first four bits, the stream looks like this:
1011**0**100 10111**0**11 *0* 11111**00**1 111111...

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to `B4 (ignored) BB F9 Fx`, and we have our magic values.

The 4th R: Reasoning uses another variation of this technique. Instead of matching the values explicitly, it watches for the data field on a particular sector, waits for three nibbles and three bits to pass, and then reads and stores the next 16 nibbles in an array. Then it calculates a checksum of those 16 nibbles, and uses the checksum as an index into the table of those 16 nibbles, to fetch two 8-bit keys in a row. The table is treated as a circular list, so if the index were 15, then the two keys would be formed by fetching the last entry in the array and the first entry in the array. The keys are used to decipher the other nibbles that are read from all of the other sectors on the disk. It looks like this:

```
;wait for nibble to arrive
BB63    LDA    $C08C,X
BB66    BPL    $BB63
;wait for nibble to leave
;if zero-bit is present,
;then read value lasts longer
BB68    LDA    $C08C,X
BB6B    BMI    $BB68
;wait for nibble to arrive
BB6D    LDA    $C08C,X
BB70    BPL    $BB6D
;trigger desync
BB72    STA    $C08D,X
;delay to reduce number of times
;that branch will be taken
BB75    NOP
;wait for status value to leave
;if zero-bit is present,
;then read value lasts longer
BB76    LDA    $C08C,X
BB79    BMI    $BB76
;wait for next nibble to arrive
BB7B    LDA    $C08C,X
BB7E    BPL    $BB7B
```

That stream looks like `CF CF 9E FD ED BB E6 B6 ED FB FC EB DF DE D3 D9 FF D9 DD D7` with some harmless zero-bits in between. Now let's count those cycles:

```
BB63    LDA    $C08C,X
BB66    BPL    $BB63
BB68    LDA    $C08C,X
BB6B    BMI    $BB68
BB6D    LDA    $C08C,X
BB70    BPL    $BB6D      ;2 cycles
BB72    STA    $C08D,X    ;5 cycles
BB75    NOP               ;2 cycles
BB76    LDA    $C08C,X    ;4 cycles
;but +4 cycles for each time reached
;because of zero-bit
BB79    BMI    $BB76      ;2 cycles
;but +3 cycles for each time
;BMI is taken because of zero-bit
;total 15 (or 22 or even 29) cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. A delay of 22 CPU cycles would normally be enough for five bits to be shifted in. However, if the delay is caused by the presence of a zero-bit, then it behaves as though the delay were only 18 CPU cycles, which is enough for four bits to be shifted in. A delay of 29 CPU cycles is enough for seven bits to be shifted in. However, if the delay is caused by the presence of a second zero-bit, then it behaves as though the delay were only 21 CPU cycles, which is enough for five bits to be shifted in. In any case, the routine is written to discard a fixed number of regular bits, along with any zero-bits that are also present. Back to our stream, in binary, it would look like this:
11001111 11001111 **0** 10011110 11111101 **0** 11101101
10111011 11100110 10110110 11101101 11111011 **0**
11111100 11101011 11011111 11011110 11010011
11011001 11111111 11011001 11011101 **0** 11010111
with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:
0 1111**0**100 11110111 11101**0**11 10110110 11101111
10011010 11011011 10110111 111011**0**1 11111001
11010111 10111111 10111101 10100111 10110011
11111111 10110011 1011101**0** 11010111

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to `F4 F7` (both ignored) `EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA`. The trailing values are stored backwards, and the checksum is `#$67`. The low four bits (7) are the index into the table, and the values at offset 7 and 8 are `#$D7` and `#$F9`.

A bit-copier that misses any of these zero-bits will write a track whose length and contents do not match the original.

7.10.16 Race conditions

Page 4 of the Software Control of the Disk][or IWM Controller document states that "The Disk][controller hardware will keep the ENABLE/ signal to its active low state for approximately one second after the execution of the motor off instruction, therefore read/write can be performed reliably within this period." So, a program can issue the motor off instruction, and then read sector data successfully for up to one second afterwards.

This behavior functions as a very nice anti-debugging mechanism, since single-stepping through the disk access code, after the motor-off instruction has been issued, will cause the time period to be exceeded. Thus, the disk won't be readable at that time. Sherwood Forest uses this technique.

Page 4 of the Software Control of the Disk][or IWM Controller document also states that "...the program should verify that the motor is spinning by monitoring the change in data pattern read from the drive." That is to say, while the drive is spinning, the value will change. Once the drive stops spinning, the value will not change anymore.

Lady Tut uses this technique. It issues the motor-off instruction, and then reads continually from the drive until it sees two consecutive bytes of the same value. The program assumes at that point that the drive is no longer spinning. Periodically thereafter, the program reads from the QA switch of the Data Register, and compares the newly read value with the initially read value. If a different value is seen, then the program triggers a reboot.

In section 9-14 of Understanding the Apple][, Jim Sather says, "any even address could be used to load data from the data register to the MPU, although `$C088` ...would be inappropriate." It might be considered inappropriate because of the one-second window noted previously, but that's exactly how the program Mr. Do! uses it. By reading from `$C088`, the program is able to issue the motor off instruction, and fetch the data at the same time. It is compact and useful for anti-debugging.

Faster pussycat

Another kind of race condition revolves around how quickly the data can be read from the disk. Borrowed Time, for example, reads an entire track in one revolution. In an interview for the Open Apple podcast, Rebecca Heineman says that she performs the decoding while the seek is in progress. While this is certainly possible, it would incur the significant overhead of having to store all 16 of the two-bit arrays—a total of 1.3kB! — before any decoding could occur. Of course, this is not what was done. Instead, each sector is read individually, but the denibbilisation is interleaved with the read. It means that the sector is decoded directly into memory, with only 86 bytes of overhead for a single two-bit array, and the use of two tables of 106 bytes and 256 bytes respectively. It is obviously fast enough to catch the next sector that arrives

The code looks like this, after validating the data field prologue:

```
0946    LDY     #$AA
;zero rolling checksum
0948    LDA     #0
094A    STA     $26
;wait for nibble to arrive
094C    LDX     $C0EC
094F    BPL     $94C
;index into table of offsets of structures
0951    LDA     $A00,X
;store offset
0954    STA     $200,Y
;update rolling checksum
0957    EOR     $26
;fetch 86 times
0959    INY
095A    BNE     $94A
095C    LDY     #$AA
095E    BNE     $963
;store decoded value
0960    STA     $9F55,Y
;wait for nibble to arrive
0963    LDX     $C0EC
0966    BPL     $963
;update rolling checksum
0968    EOR     $A00,X
;fetch structure offset, bits 0-1
096B    LDX     $200,Y
;merge first member of two-bit structure
;with six-bit value to recover eight-bit
    value
096E    EOR     $B00,X
;loop 86 times
0971    INY
0972    BNE     $960
;save 85th decoded value for last
0974    PHA
;clear low two bits
0975    AND     #$FC
0977    LDY     #$AA
;wait for nibble to arrive
0979    LDX     $C0EC
097C    BPL     $979
;update rolling checksum
097E    EOR     $A00,X
;fetch structure offset, bits 2-3
0981    LDX     $200,Y
;merge second member of two-bit structure
;with six-bit value to recover eight-bit
    value
0984    EOR     $B01,X
;store decoded value
0987    STA     $9FAC,Y
;loop 86 times
098A    INY
098B    BNE     $979
;wait for nibble to arrive
098D    LDX     $C0EC
0990    BPL     $98D
;clear low two bits
0992    AND     #$FC
0994    LDY     #$AC
;update rolling checksum
0996    EOR     $A00,X
;fetch structure offset, bits 4-5
```

```
;offset -2 to account for Y+2
0999      LDX      $1FE,Y
;merge third member of two-bit structure
;with six-bit value to recover eight-bit
    value
099C      EOR      $B02,X
;store decoded value
099F      STA      $A000,Y
;wait for nibble to arrive
09A2      LDX      $C0EC
09A5      BPL      $9A2
;loop 84 times
09A7      INY
09A8      BNE      $996
;clear low two bits
09AA      AND      #$FC
;update rolling checksum
09AC      EOR      $A00,X
;restore slot to X
09AF      LDX      $2B
;retry if checksum mismatch
09B1      TAY
09B2      BNE      $9BD
;wait for nibble to arrive
09B4      LDA      $C0EC
09B7      BPL      $9B4
;check only first epilogue byte
09B9      CMP      #$DE
09BB      BEQ      $9BF
09BD      SEC
09BE      .BYTE $24
09BF      CLC
;store 85th decoded value
09C0      PLA
09C1      LDY      #$55
09C3      STA      ($44),Y
09C5      RTS
```

The exact way in which the technique works is as follows. First, each of the two-bit values is read into memory, but instead of storing them directly, the values are used as an index into the 106-bytes table. The 106-bytes table serves two purposes. The first, in the context of the two-bit values, is as an array of offsets within the 256-bytes table. The second, in the context of the six-bit values, is as an array of pre-shifted values for the six-bit nibbles. The 256-bytes table is composed of groups of two-bit values in all possible combinations for each of the three positions in a nibble. To produce the eight-bit value, each of the pre-shifted six-bit values is `OR`ed with the corresponding two-bit value. It is unknown why the 85th value is treated separately from the rest in that code; it could certainly be decoded at the same time, saving five lines.

With the benefit of determination to improve it, and the ability to do so, I rewrote this loader to decode all of the bytes directly, reduced the size of the code, and made it even faster. I call it "0boot."[49] Then I reduced the overhead to just two bytes, if page `$BF` is not the destination. I call that one "q-boot."[50] The two tables are still 106 bytes and 256 bytes respectively. It might appear that the second table can be reduced to 192 bytes, since the other 64 bytes are unused. However, it is not possible for this algorithm, because the alignment is required to supply the pre-shifted values. If the table were reduced in size, then additional operations would be required to reproduce the effect of the shift, and which would take longer to execute than the time available before the next nibble arrived.

Interestingly, Heineman claims to have created and released the technique in 1980,[51] but it was apparently not until 1984 that she used it in a release herself. It certainly existed in 1980, though. Automated Simulations (which later became Epyx) included the technique with the programs Hellfire Warrior and Rescue At Rigel. In 1983, Free Fall Associates (founded by the co-founder of Automated Simulations, whose last name begins with "Free", and a programmer whose last name ends with "Fall") included the technique with the programs Murder on the Zinderneuf and Archon. (Apparently they took it with them, as Epyx did not use it again.) Also in 1983, Apple included the technique in ProDOS. In 1985, Brøderbund included the technique with the program Captain Goodnight. According to Roland Gustafsson, Apple supplied that code.[52]

[49] `http://pferrie.host22.com/misc/0boot.zip`

[50] `http://pferrie.host22.com/misc/qboot.zip`

[51] Personal communication

[52] Personal communication

Also interestingly, whoever included it in the Free Fall Associates programs either did not understand it, or just did not want to touch it—there, the loader has been patched to require page-aligned reads, but the code still performs the initialisation for arbitrary addressing. Twelve lines of code could have been removed from that version. The Interplay programs that use the technique also require page-aligned reads, but do not have the unnecessary initialisation code.

Quote of the day by Olivier Guinart, "It's ironic that the race condition would be used by a program called Borrowed Time."

7.11 Track-level protections

7.11.1 Track length

The length of a track might not be constant across all of the tracks on a disk. The speed of the drive is the primary reason: the faster the drive, the shorter the track (that is, fewer nibbles can be written) because of the larger gaps between the nibbles.

Wizardry determines the length of the track, by measuring the time between succeeding arrivals of sector zero, and then calculates the deviation from the expected value. This deviation value is applied to the length of several other tracks, and the result is compared against the expected lengths. If the length of the track is not within the range that is expected, then the program hangs. This protection cannot be reproduced by a sector-copier or track-copier, because they will discard the original data between the sectors, thus altering the length of the track. A bit-copier can usually reproduce this protection because it writes the entire track mostly as it appeared originally, so the track length is at least similar to the original.

7.11.2 Track positioning

The stepper motor in the Disk][system is composed of four magnets. To advance a whole track requires activating and deactivating two phases in the proper order, and with a sufficient delay, for each track to step. To step to a later track, the next phase must be activated while the other phases are deactivated. To step to an earlier track, the previous phase must be activated while the other phases are deactivated. As might be expected, activating and then deactivating only one of the phases will cause the stepper to stop half-way between two tracks. This is a half-track position. It is even possible to produce quarter-track stepping reliably, by performing the half-track stepping method, but with a smaller delay. Depending on the hardware, it can also be done by activating two of the phases, and then deactivating only one of them. This last technique is used by Spiradisc. (§7.11.9.)

The issue with half-track and quarter-track positioning is that data written to these partial track positions will cause signal interference with data written to the neighbouring half-track or quarter-track at the same relative position. To avoid unintentional cross-talk, data can be written to only part of the track such that there is no overlap, or placed at least three-quarters of a track apart. (The reliability of three-quarter tracks is questionable.)

The maximum amount of data that can be placed at partial-track intervals is proportional to the stepping—a quarter of a track for each of four consecutive quarter-tracks, half of a track for each of two consecutive half-tracks, or a full track for consecutive three-quarter-tracks. There can be a significant performance hit to access the data, too—it requires an almost complete rotation to reach the start of the data on subsequent tracks if the maximum density is used, because the seek time is long enough that the start will be missed on the first time around. As a result, the most common amount that is used is only a quarter of the track, and placed far enough around the track that the read can be performed almost continuously. Programs that make use of partial tracks usually include a standard format of individual sectors, so the only trick to the protection is the location of the data on the disk.

Agent USA uses the half-track technique with five sectors per track.

Championship Lode Runner uses an alternating quarter-track technique with just two sectors per track but of twice the size. While loading, the access alternates between the neighbouring quarter-tracks, resulting in the drive "chattering", but allowing the sectors to be spaced only half of a rotation apart. In both cases of the programs here, it results in an extremely fast load time because of the reduced head movement.

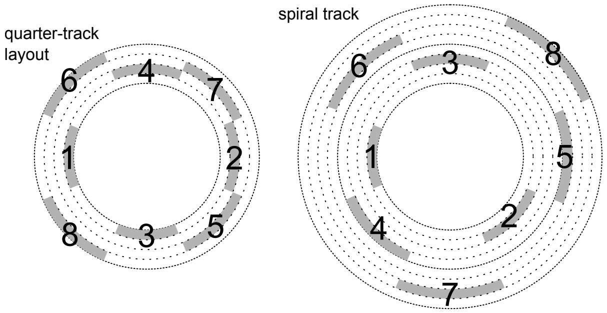
In this case, the protection is the use of partial tracks. Copy programs which do not copy the partial tracks (and copying partial tracks is not the default behavior) will fail to reproduce the protection.

7.11.3 Synchronised tracks

If the approximate rotation speed of the drive is known, then it becomes possible to place sectors at specific locations on tracks, such that they have a special position relative to sectors on other tracks. This technique is identical to synchronized sectors, except that it spans tracks, making it even more difficult to reproduce, because it is difficult to determine the relative position of sectors across tracks. Unlike "spiral tracking" (§7.11.4), this technique limits itself to checking for the existence of particular sectors, rather than actually reading them.

Blazing Paddles uses this technique. Once it finds sector zero on track zero, as a known starting point, it seeks to track one, reads the address field of the next sector to arrive, and then compares it to an expected value. If the proper sector is found, then the program seeks to track two, reads the address field of the next sector to arrive, and compares it to an expected value. If the proper sector is found, then the program seeks to track three. This is repeated over eight tracks in total. It means that the original disk has one sector placed at a specific location on each of eight consecutive tracks, relative to sector zero of track zero, such that it factors in how much the disk rotates during the time that the controller takes to move the head from track zero. It also supports slight variations in rotation speed, such that the read can begin anywhere after the address field for the previous sector, without failing the protection.

7.11.4 Track spiralling



"Track spiralling" or "spiral tracking" is a technique whereby the data is placed in partial-track intervals, but treated as a complete track. By measuring the time to move the head to a partial-track, the position on the track can be known, such that the next sector to be read will have a predictable number, and therefore can be read without validation, once the start of the sector is found. A copy of the disk will not place the data at the same relative position, causing the protection to fail. The stepping in spiral tracking goes in only one direction. A visualisation of the data access would look like a broken spiral, hence the name.

One major problem with spiral tracking is that variations in rotation speed can result in the read missing its queue and not finding the expected sector. For 30 years, I believed a claim[53] that the program Captain Goodnight uses this technique. It doesn't. The Observatory uses a spiral pattern for faster loading, but still verifies the sector number first. However, the program LifeSaver uses true spiral tracking.

7.11.5 Track arcing

"Track arcing" uses the same principle as spiral tracking, but instead of stepping in only one direction, it reaches a threshold and then reverses direction.

7.11.6 Track mirroring

Track mirroring should be placed conceptually between synchronized tracks and spiral tracking. As

[53] From a cracker whose crack-screens were displayed only by pressing a particular key-sequence during the boot. They were known as "Hidden Pages" (Imagine that—a cracker who didn't want to brag openly!) Both of the programs Captain Goodnight and Where In The World Is Carmen Sandiego (first release) use alternating quarter-tracks—the same technique as in the program Championship Lode Runner. (The former two were released within a year of the latter one.) The sectors are placed in a N/S/E/W orientation on the first two tracks, a NW/SE/NE/SW orientation on the next two tracks, and then back to the N/S/E/W orientation on the next two tracks, and so on. The loader will allow an entire revolution to pass, if necessary, in order to find the requested sector. The tracks are synchronized, however, because they must be to avoid cross-talk. (§7.11.7.)

with synchronized tracks, it expects a particular sector to be found after stepping across multiple tracks. As with spiral tracking, it reads the sector data. However, unlike spiral tracking, it verifies that the contents of that sector match exactly the contents of all of the other sectors that are synchronized similarly across the tracks.

The Toy Shop uses this technique. It reads three consecutive quarter-tracks in RWTS18 format, and verifies that they all fully readable and have a valid checksum. This is possible only because they are identical in their content and position. The contents of the last quarter-track are used to boot the program. A funny thing occurs when the program is converted to a NIB image: the protection is defeated transparently, because NIB images do not support partial tracks, so the attempt to read consecutive quarter-tracks will always return identical data, exactly as the protection requires.

Pinball Construction Set uses this technique. It reads a sector then activates a phase to advance the head, and then proceeds to read a sector *while the head is moving.* The head continues to drift over the track while the sector is being read. After reading the sector, the program deactivates the phase, reads another sector, and then completes the move to the next track. Once there, it reads a sector. It activates a phase to retreat the head, and then performs the same trick in reverse, until the start of the track is reached again. It performs this sequence four times across those two tracks, which makes the drive hiss. The program is able to read the sector as continuous data because the disk has consecutive quarter-tracks that are identical in their content and position.

7.11.7 Cross-talk

While cross-talk is normally something to be avoided, it can serve as a copy-protection mechanism, by intentionally allowing it to occur. It manifests itself in a manner similar to the effect of having excessive consecutive zero-bits being present in the stream, where reading the same stream repeatedly will yield different values. The lack of such an effect indicates the presence of a copy.

7.11.8 More tracks

Many disk drives had the ability to seek beyond track 34, and many disks also carried more than 35 tracks. However, since DOS could not rely on the presence of either of these things, it did not offer support for them. Some copy programs did not support the copying of additional tracks for the same reason. Of course, programmers who did not use DOS had no such limitation. While the actual number of available tracks could vary up to 40 or even 42, it was fairly safe to assume that at least one track existed, and could be read by direct use of the disk drive.

Faial uses this technique to place data on track 35.

7.11.9 SpiraDisc

No description of copy-protection techniques could be complete without including SpiraDisc. This program was a protection technology that introduced the idea of spiral tracking, though the implementation is not spiral tracking as we would describe it today. It is, in fact, a precise placement of multiple sectors on quarter-tracks, such that there is no cross-talk while reading them, but without a specific order. The major deviation from the current idea of spiral tracking is that there is no synchronization of the sectors beyond avoiding cross-talk. The program will allow a complete rotation of the disk to occur, if necessary, while searching for the required sector.

The first-stage boot loader is a single sector that is “4-and-4” encoded, and 768 bytes long. The second stage loader is composed of ten regular sectors that are “6-and-2” encoded. They are read one by one—there is no read-scattering here to speed up the process. Thereafter, reads use an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

The encoding is not “6-and-2”, either, it is “6-and-0” encoding. This requires 344 bytes per sector, instead of the regular 342 bytes. The decoder overwrites the addresses `$xxAA` and `$xxAB` (the program supports only page-aligned reads) twice in order to compensate for the additional bytes. The decoding is interleaved, so there is no denibbilisation pass.

The “6-and-0” encoding works by using the six-bit nibble as an alternating index into one of the arrays of six-bit or two-bit values. The code is both much faster (no fetching of the two-bit array) and much smaller (two-thirds of the size) than the one described in Race Conditions,(§7.10.16) but the decoding tables occupy 1.5kb of memory. The memory layout might have been chosen to avoid a timing penalty due to page-crossing accesses. However, the penalty has no effect on the performance of the routine because the code must still spend time waiting for the bytes to arrive from disk. Therefore, the tables could have been combined into a 512-byte region instead, which is a closer match to the memory usage of the routine described in Race Conditions.

A Spiradisc-protected disk uses four sectors per track, but since the track stepping is quartered, the data density is equivalent to a single 16-sector track. Each sector has a unique prologue value to identify itself. When a read is requested, if a sector cannot be found on the current track, then the program advances the drive head by one quarter-track, and then attempts the read again. If the read fails again, then the program retreats the drive head by one quarter-track, and then attempts the read again. If the read still fails, then the program retreats the drive head by another quarter-track, and then attempts the read again. If the read fails at this point, then the disk is considered to be corrupted.

Given the behaviour of the read request, the data might not be stored on consecutive quarter-tracks. Instead, they might zig-zag across a span of up to three quarter-tracks. This is another deviation from the idea of spiral tracking. By coincidence, the movement is very similar to the one in the program Captain Goodnight and other Brøderbund titles.

Copying a SpiraDisc-protected disk is difficult because of the potential for cross-talk which would corrupt the sectors when they are read back. However, images produced by an E.D.Dċard will work in emulators, if the copy parameters are set correctly.

When run, the program decodes selected pages of itself, based on an array of flags, and also re-encodes those pages after use, to prevent dumping from memory. The decoding is simply an exclusive-OR of each byte with the value `#$AC`, exclusive-ORed with the index within the page.

At start-up, the program profiles the system: scanning the slot device space, and records the location of devices for which the first 17 bytes are constant (that is, they return the same value when read more than once), and which do not have eight bytes that match the first one within those 17 bytes. For example, Mockingboard has memory-mapped I/O space in that region, which are mostly zeroes. The program calculates and stores a checksum for slot devices which pass this check. The store was supposed to happen only if the checksum did not match certain values, but the comparison is made against a copyright string instead of an array of checksums. The first time around, all values are accepted. During subsequent profiling, the value must match exactly.

The program checks if bank one is writable, after attempting to write-enable it, and sets a flag based on the result. The program checksums the F8 and F0 ROM BIOS codes, watches for particular checksums, and sets flags based on the result. The original version of the program (as seen in 1981, used on the program Jawbreaker) actually *required* that the ROM BIOS code match particular checksums—either the original Apple][or the Apple][+—otherwise the program simply wiped memory and rebooted. (This prevented protected programs from running on the Apple][e or the Apple][c.) The no-doubt numerous compatibility problems that resulted from this decision led to the final check being discarded (as seen in 1983, used on the program Maze Craze Construction Set, but quite possibly even earlier), though the rest of the profiling remains. However, having even one popular title that didn't work on more modern machines was probably sufficient to turn publishers entirely off the use of the program.

The program probes all of memory by writing a zero to every second byte. However, it skips pages `#0`, `#2`, `#4-7`, and `#$A8-C0`, meaning that it writes data to all slot devices, with unpredictable results. The program also re-profiles the system upon receiving each request to read tracks. This re-profiling is intended to defeat memory dumps that are produced by NMI cards, and which are then transferred to another machine, as the second machine might have different hardware options.

The program also checksums the boot PROM prior to disk reads, and requires that it matches one particular checksum—that of the Disk][system—otherwise the program wipes memory and reboots. (This prevents protected programs from running on the Apple][GS.)

Interestingly, despite all of the checks of the environment, the program does not protect itself against tampering, other than using encoded pages. The memory layout is data on pages `#$A8-B1`, and code on pages `#$B2-BF`. The data pages are very sparse, leaving plenty of room for a boot tracer to intercept execution and disable protections.

The program uses a quarter-track stepping algorithm that activates two phases, and then deactivates only one of them. According to Roland

Gustafsson, this stepping technique allows for more precise positioning of the drive head, but it does not work on Rana drives. It was for this reason that he used the reduced-delay technique instead. (§7.11.2.) The reduced-delay technique is apparently the only one which works on an Apple][c, as well. Spiradisc predated the Apple][c by about two years, so it was just bad luck that an incompatible technique was chosen.

7.12 Illegal opcodes

The 6502 CPU has 151 documented instructions. There are quite a few additional instruction encodings for which the results could be considered useful, if the side-effects (e.g. memory and/or register corruption, or long execution time) were also acceptable. In some cases, the instructions were used to obfuscate the meaning of the code, since they would not be disassembled correctly. Some of these undocumented instructions were replaced in the 65C02 CPU with documented instructions with different behaviors, and without the unfortunate side-effects. In some cases, the code that used the undocumented instructions was not affected because the results of the undocumented instructions were discarded, and the documented replacement did not introduce especially unwanted behavior. Note that the instructions that were not replaced will cause the 65C02 CPU to hang.

The Datasoft version of the program Dig Dug uses this technique. It begins with an instruction which used to behave as a two-byte `NOP`, but which is now a zero-page `STZ` instruction. Since the program does not make use of the zero-page at that time, the store has no side-effects. It looks like this in 6502 mode:

```
0801     74             ???
0802     4C  B0  58     JMP     $58B0
```

In 65C02 mode, the same machine code interpreted differently.

```
0801     74  4C         STZ     $4C
0803     B0  58         BCS     $85D
```

Beer Run uses this technique, but was unfortunate enough to choose an instruction which was not defined on the 65C02 CPU, so the program does not work on a modern machine. The code is run with the carry set much earlier in the flow, as a side-effect of executing a routine in the ROM BIOS. It is possible that the authors were not even aware of the fact.

```
051B     LDX     #$00
. . .
051F     LDA     #$00
0521     STA     $00
. . .
;FF  00  00
0525     ISC     $0000 ,X
```

which, when executed, does this:

```
INC     $0000 ,X
SBC     $0000 ,X
```

X is zero, so `$00` is first incremented to `#$01`, and then subtracted from `A`. `A` is zero before the subtraction, so it becomes `#$FF`. The resulting `#$FF` is used as a key to decipher some values later.

7.13 CPU bugs(!)

The original 6502 CPU had a bug where an indirect `JMP (xxFF)` could be directed to an unexpected location because the MSB will be fetched from address `xx00` instead of page `xx+1`. Randamn relies on this behavior to perform a misdirection, by placing a dummy value at offset zero in page `xx+1`, and the real value at address `xx00`.

While not a bug, but perhaps an undocumented feature—the breakpoint bit is always set in the status register image that is placed on the stack by the `PHP` instruction. Lady Tut relies on this behavior to derive a decryption key.

There is also a class of alternative behaviours between the 6502 and the 65C02 CPUs, particularly regarding the Decimal flag. For example, the following sequence will yield different values between the two CPUs: $1B on a 6502, and $0B on a 65C02. These days, it would be used as an emulator detection method. Try it in your favorite emulator to see what happens.

```
SED
SEC
LDA #$20
SBC #$0F
```

7.14 Magic stack values

One way to obfuscate the code flow is through the use of indirect transfers of control. Rescue At Rigel fills the stack entirely with the sequence #$12 #$11 #$10, and then performs an RTI without setting the stack pointer to a constant value. Of course, it works reliably.

Since there are only three values in the sequence, there should be only three cases to consider. If the stack pointer were #$F6 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1F7. If the stack pointer were #$F7 at the time of executing the RTI instruction, then this causes the value #$11 and $1210 to be fetched from $1F8. If the stack pointer were #$F8 at the time of executing the RTI instruction, then this causes the value #$10 and $1112 to be fetched from $1F9. The program has an RTS instruction at the first and last of those locations. That yields two more cases to consider. The RTS at $1011 transfers control to $1112+1. The RTS at $1112 transfers control to $1210+1. That leaves one more case to consider. The program has an RTS instruction at $1113. The RTS at $1113 transfers control to $1211. So, both $1210 and $1211 are reachable this way. Both addresses contain a NOP instruction, to allow the code to fall through to the real entrypoint

Note the phase "there should be." There is one special case. The remainder of 256 divided by three is one. What is in that one byte? It's the value #$10. So the first and last byte of the stack page is #$10, introducing an additional case. If the stack pointer were #$FD at the time of executing the RTI instruction, then this causes the value #$11 and $1010 to be fetched from $1FE. The program has an RTS instruction at $1010. The RTS at $1010 transfers control to $1112+1. The RTS at $1113 transfers control to $1211.

That's not all! We can construct an even longer chain. If the stack pointer were #$F9 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1FA. The RTS at $1011 transfers control to $1112+1, but the RTS at $1113 causes the stack pointer to wrap around. The CPU fetches both #$10 values, so the RTS at $1113 transfers control to $1010+1. The RTS at $1011 transfers control again to $1112+1. The RTS at $1113 finally transfers control to $1211.

Championship Lode Runner has a smaller chain. It uses only two values on the stack: $3FF and $400. An RTS transfers control to $3FF+1. The program has an RTS at $400. The RTS at $400 transfers control to $400+1, the real entrypoint.

7.15 Obfuscation

7.15.1 Anti-disassembly (aka WTF?

This technique is intended to prevent casual reading of the code—that is, static analysis, and specifically targeting linear-sweep disassemblers—by inserting dummy opcodes into the stream, and using branch instructions to pass over them. At the time, recursive-descent disassembly was not common, so the technique was extremely effective.

Wings of Fury uses this technique, even for its system detection. The initial disassembly follows, with undocumented instructions such as RLA.

```
9600    ORA    (0,X)
9602    LDY    #$10
9604    BPL    $9616
9606    RLA    ($10,X)
9608    NOP
960A    BEQ    $95AC
960C    NOP
960E    STY    $84
9610    STY    $18
9612    CLC
9613    CLC
```

```
9614      BNE      $961C
9616      CLC
9617      CLC
9618      BNE      $960B
961A      SRE      ($51),Y
961C      STY      $C009
961F      STX      $20,Y
9621      ORA      ($10),Y
9623      CPX      $84
9625      STA      $C008
9628      BEQ      $9672
962A      LDA      $C088,X
962D      ORA      ($18),Y
962F      ORA      ($10),Y
9631      ASL
9632      LDX      #$27
9634      ASL
9635      ASL
9636      LDY      #$10
9638      BPL      $9630
963A      BRK
963B      JMP      $93BD
963E      TYA
963F      STA      $400,X
9642      BNE      $964C
9644      BRK
```

Upon closer examination, we see the branch instruction at `$9604` is unconditional, because the value in the `Y` register is positive. That leads to the branch at `$9618`. This branch is also unconditional, because the value in the `Y` register is not zero. That takes us into the middle of an instruction at `$960B`, and requires a second round disassembly:

```
;store #$64 at $84
960B      LDY      #$64
960D      STY      $84
;four dummy instructions
960F      STY      $84
9611      CLC
9612      CLC
9613      CLC
;unconditional branch
;because Y is not zero
9614      BNE      $961C
...
;switch to auxiliary memory bank, if available
961C      STY      $C009
;store alternative value at $84 ($20+#$64=$84)
961F      STX      $20,Y
;dummy instruction
9621      ORA      ($10),Y
;compare the two values
;will differ in 64kb environment
9623      CPX      $84
;switch to main memory bank
9625      STA      $C008
;branch if 128kb memory exists
9628      BEQ      $9672
;turn off the drive
962A      LDA      $C088,X
;dummy instruction
962D      ORA      ($18),Y
;dummy instruction masks real instruction
962F      ORA      ($10),Y
;dummy instruction in first pass
;opcode parameter in second pass
9631      ASL
;length of error message
9632      LDX      #$27
;two dummy instructions
9634      ASL
9635      ASL
9636      LDY      #$10
;unconditional branch
;because Y is positive
9638      BPL      $9630
963A      BRK
963B      JMP      $93BD
963E      TYA
963F      STA      $400,X
9642      BNE      $964C
9644      BRK
```

A third round disassembly:

```
;unconditional branch
;because Y is positive
9630      BPL      $963C
...
;message text
963C      LDA      $9893,X
;write to the screen
963F      STA      $400,X
;unconditional branch
;because A is not zero
9642      BNE      $964C
```

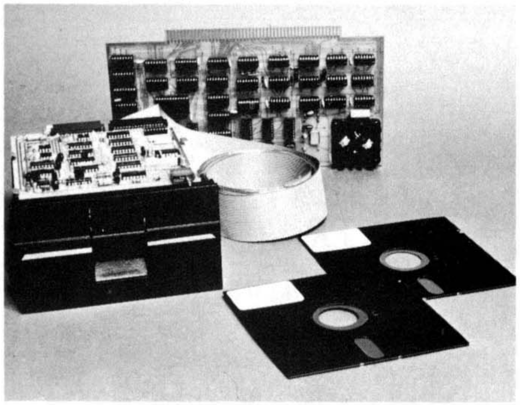
The obfuscated code only gets worse from there, but the intention is clear already.

7.15.2 Self-modifying code

As the name implies, this technique relies on the ability of code to modify itself at runtime, and to have the modified version executed. A common use of the technique is to improve performance by updating an address with a loop during a memory copy, for example. However, from the point of view of copy-protection, the most common use is to change the code flow, or to act as a light encoding layer. Self-modifying code can be used to interfere with debuggers, because a breakpoint that is placed on the modified instruction might be overwritten directly, thus removing it, and resulting in uncontrolled execution; or turned into an entirely unrelated (and possibly meaningless or even harmful) instruction, with unpredictable results.

Aquatron hides its protection check this way. The initial disassembly looks like this, complete with undocumented instructions such as ISB:

```
9600   DEC   $9603
9603   ISB   $9603
9606   LDA   $9628
9609   EOR   #$C9
960B   BNE   $960E
960D   JSR   $288D
9610   STX   $18,Y
9612   BNE   $9615
9614   JMP   $29A0
9617   TYA
9618   BCC   $961B
961A   JSR   $59
961D   STX   $99,Y
961F   BRK
9620   STX   $C8,Y
9622   BNE   $9617
9624   TYA
9625   BPL   $9628
9627   JMP   $2960
```



Upon closer examination, we see references to instructions at “hidden” offsets, and of course, the direct modification of the instruction at `$9603`.

Second round disassembly:

```
9600    DEC    $9603
;-> INC $9603
;undo self-modification and continue
9603    ISB    $9603
9606    LDA    $9628
9609    EOR    #$C9
;unconditional branch
;because A is not zero
960B    BNE    $960E
960D    .BYTE  $20
;replace instruction below
960E    STA    $9628
9611    CLC
;unconditional branch
;because A is not zero
9612    BNE    $9615
9614    .BYTE  $4C
9615    LDY    #$29
9617    TYA
9618    BCC    $961B
961A    .BYTE  $20
;decode and store
961B    EOR    $9600,Y
961E    STA    $9600,Y
9621    INY
9622    BNE    $9617
9624    TYA
;unconditional branch
;because Y is positive
9625    BPL    $9628
9627    .BYTE  $4C
;self-modified by $960E to $A9 on first pass
;restored to $60 on second pass
9628    RTS
;decoded by $961B-9620 on first pass
;re-encoded on second pass
9629    .BYTE  $29
```

Now we can see the decryption routine. It decodes the bytes at `$9629-96FF`, which contained a check for a sector with special format. If the checked passes, then the routine at `$9600` is run again, which reverses the changes that had been made — the bytes at `$9629-96FF` are encoded again, and the routine exits via the RTS instruction at `$9628`.

7.15.3 Self-overwriting code

When self-modification is taken to the extreme, the result is self-overwriting code. There, the RWTS routine reads sector data over itself, in order to change the execution behavior, and potentially remove user-defined modifications such as breakpoints or detours. LifeSaver uses this technique. The loader enters a loop which has no apparent exit condition. Instead, the last sector to be read from disk contains an identical copy of the loader code, except for the last instruction which branches to a new location upon completion. When combined with a critically timing-dependent technique, such as reading a sector while the head is moving, it becomes extremely difficult to defeat.

7.15.4 Encryption and compression

Encryption (or, more correctly, enciphering) of code was a popular technique, but the keys were always very weak. The enciphering usually consisted of an exclusive-OR of the byte with a fixed key. In some cases, the key was a rolling value taken from the byte just deciphered. In some rarer cases, multiple keys were used.

Goonies uses a rotate operation. However, since the 6502 CPU does not have a plain rotate instruction—only rotate with carry — the program must set the carry bit correctly prior to the operation. The program does it this way:

```
;save value
0405    PHA
;extract carry bit
0406    LSR
;restore value
0407    PLA
;rotate with carry
0408    ROR
```

Compression of graphics was necessary to reduce the size of the data on disk, and to decrease load times, since the reduced disk access more than made up for the time spent to decompress the graphics. The most common compression technique was Run-Length Encoding (RLE), using a stream derived from every second horizontal byte, or vertical columns. More advanced compression, such as something based on Lempel-Ziv, was generally considered to be too slow to use.

Perhaps based on the assumption that LZ-based compression was too slow, compression of code seems to have been entirely absent until recently—all of my releases use my decompressor for aPLib[54], for an almost exact or even slightly reduced load time, which shows that the previous assumption was quite wrong. Others have had success with my decompressor for LZ4[55] when used for graphics. A more recent LZ4-based project is also showing promise.[56]

7.16 Virtual machines

One of the most powerful forms of obfuscation is the virtual machine. Instead of readable assembly language that we can recognise, the virtual machine code replaces instructions with bytes whose meaning might depend on the parameters that follow them. Electronic Arts were famous for their use of pseudo-code (p-code) to hide the protection routines in programs such as Archon and Last Gladiator. That virtual machine was even ported to the Commodore 64 platform.

Last Gladiator uses a top-level virtual machine that has 17 instructions. The instructions look like this:

```
00      JMP
01      CALL NATIVE
02      BEQ
03      LDA IMM
04      LDA ABSOLUTE
05      JSR
06      STA ABSOLUTE
07      SBC IMM
08      JMP NATIVE
09      RTS
0A      LDA ABSOLUTE, A ;p-code A register
0B      ASL
0C      INC ABSOLUTE
0D      ADC ABSOLUTE
0E      XOR ABSOLUTE
0F      BNE
10      SBC ABSOLUTE
11      MOVS
```

It has the ability to transfer control into 6502 routines, via the instructions that I named "call native" and "jmp native." The parameters to the instructions were XORed with different values to make the disassembly even more difficult. Since the virtual machine could read arbitrary memory, it was used to access the soft-switches, in order to turn the drive on and off. Once past the first virtual machine, the program ran a second one. The second virtual machine is interesting for one particular reason. While it looks identical to the first one, it's not exactly the same. For one thing, there are only 13 instructions. For another, two of them have swapped places:

```
0A      INC ABSOLUTE
0B      nothing
0C      LDA ABSOLUTE, A ;p-code A register
```

HARD HAT MACK

These two engines were not the only ones that Electronic Arts used, either. Hard Hat Mack uses a version that had twelve instructions.

```
00      JMP
01      CALL NATIVE
02      BEQ
03      LDA IMM
04      LDA ABSOLUTE
05      JSR
06      STA ABSOLUTE
07      SBC IMM
08      JMP NATIVE
09      RTS
0A      LDA ABSOLUTE, A ;p-code A register
0B      ASL
```

Following that virtual machine was yet another variation. This one has only eleven instructions. Nine of the instructions are identical in value to the previous virtual machine. The differences are that "ASL" is missing, and the "LDA ABSOLUTE, A" instruction is now "INC ABSOLUTE."

However, in between those two virtual machines was an entirely different virtual machine. It is a stack-based engine that uses function pointers instead of byte-code. It looks like this, if you'll forgive handler address in place of names I wasn't able to identify.

```
9DF2      .WORD xsave_retpc
9DF4      .WORD xpush_imm
9DF6      .WORD $95FF
9DF8      .WORD xpush_imm
9DFA      .WORD $A600
9DFC      .WORD xchkstk_vars
9DFE      .WORD xbeq_rel
9E00      .WORD 4
9E02      .WORD xdo_copy_prot
9E04      .WORD xjmp_retpc
```

[54] http://pferrie.host22.com/misc/aplibunp.zip
[55] http://pferrie.host22.com/misc/lz4unp.zip
[56] https://github.com/fadden/fhpack

This virtual machine is Forth. Amnesia, including its copy-protection (What You Know style), was written entirely in Forth. The Toy Shop used another virtual machine, which combined byte-code and function pointers, depending on which function was called, and all mixed freely with native code. Its identity is not known.

Of course, the most famous of all virtual machines is the one inside Pascal, an ancestor of Delphi that was very widely used in the eighties. Wizardry is perhaps the most well-known Pascal program on the Apple][system, and the Pascal virtual machine made it a simple task to port the program to other platforms. The advantage of a virtual machine is that only the interpreter must be ported, rather than the entire system. Since the language is much higher-level than assembly language, it also allows for a faster development time. It also makes de-protecting a program much harder

7.17 ROM regions

The Apple][ROM BIOS is full of little routines whose intention is clear, but whose meaning can be changed depending on the context. That leads into an interesting area of obfuscation and indirection. For our first example, there is a routine to save the register contents. It is used by the ROM BIOS code when a breakpoint occurs. It has the side-effect of returning the status register in the `A` register. That allows a program to replace the instruction pair `PHP; PLA` with the instruction `JSR $FF4A` for the same primary effect (it has the side-effect of altering several memory locations), but one byte larger.

For our second example, there is a routine to clear the primary text screen. Since the Apple][has a text and graphics mode that share the same memory region, there is one routine for clearing the screen while in text mode, and another for clearing the screen while in graphics mode. However, it is possible to use the graphics routine to clear the screen even while in text mode. That allows a program to replace `JSR $FC58` with `JSR $F832` for the same major effect. (It has the side-effect of altering several memory locations.)

For our third example, there is a routine to compare two regions of memory. It is used primarily to ensure that memory is functioning correctly. However, it can also be used to detect alterations that as those produced by a user attempting to patch a program. All that is required is to set the parameters correctly, like this:

```
LDA    #>beghi
STA    $3D
LDA    #<beglo
STA    $3C
LDA    #>endhi
STA    $3F
LDA    #<endlo
STA    $3E
LDA    #>cmphi
STA    $43
LDA    #<cmplo
STA    $42
JSR    $FE36
```

For our fourth example, there is an `RTS` instruction at a known location. A jump to this instruction will simply return. It is usually used to determine the value of the Program Counter. However, it can just as easily be used to hide a transfer of control, taking into account that the destination address must be one less than the true value, like this to jump to `$200`:

```
LDA    #$01
PHA
LDA    #$FF
PHA
JMP    $FF58
```

And so on. The first three examples are taken from Lady Tut, though in the third example, the parameters are also set in an obfuscated way, using shifts, increments, and constants. The fourth is taken from Mr. Do!.

7.18 Sensitive memory locations

There are certain regions in memory, in which modifications can be made which will cause intentional side-effects. The side-effects include code-destruction when viewed, or automatic execution in response to any typed input, among other things. The zero-page is a rich source of targets, because it is shared by so many things.

The most commonly altered regions follow.

7.18.1 Scroll window

When the monitor is active, the scrollable region of the screen can be adjusted to allow "fixed" rows and/or columns. The four locations, left (`$20`), width (`$21`), top (`$22`), and bottom (`$23`) can also be adjusted. A program can protect itself from debugging attempts by altering these values to make a very small window, or even to cause overlapping regions that will cause memory corruption if scrolling occurs.

7.18.2 I/O vectors

There are two I/O vectors in the Apple][, one for output—`CSW ($36-37)`, and one for input—`KSW ($38-39)`. CSW is invoked whenever the ROM BIOS routine COUT is called to display text. KSW is invoked whenever the ROM BIOS routine RDKEY is called to wait for user input. Both of these vectors are hooked by DOS in order to intercept commands that are typed at the prompt. Both of these vectors are often forcibly restored to their default values to unhook debuggers. They are sometimes altered to point to disk access routines, to prevent user interaction. Championship Lode Runner uses the hooks for disk access routines in order to load the level data from the disk.

7.18.3 Monitor

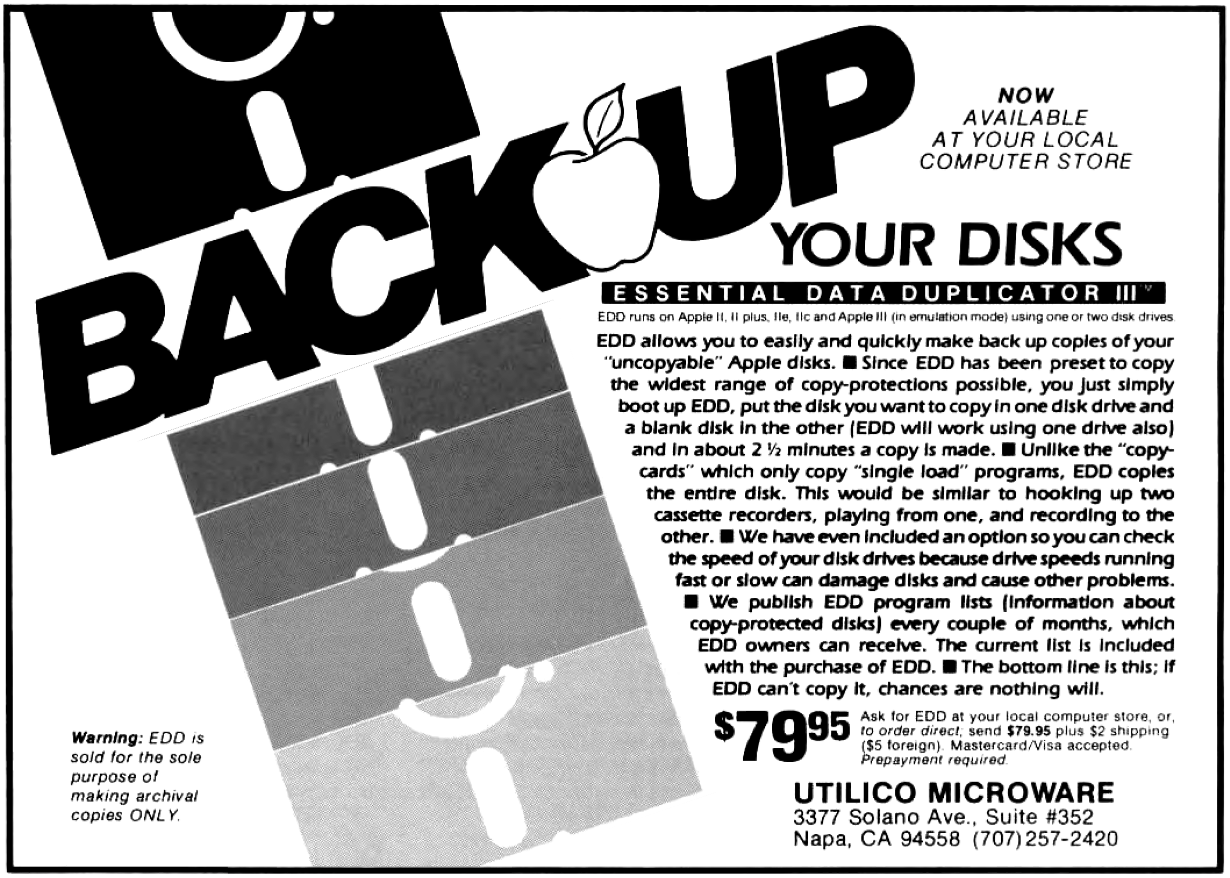
The monitor prompt allows a user to view and alter memory, and execute subroutines. It uses several zero-page addresses in order to do this. Anything that is stored in those locations (`$31`, `$34-35`, `$3A-43`, `$45-49`) will be lost when the monitor becomes active. In addition, the monitor uses the ROM BIOS routine RDKEY. RDKEY provides a pseudo-random number generator, by measuring the time between keypresses. It stores that time in `$4E-4F`.

Falcons uses address `$31` to hold the rolling checksum, and checks if `$47` is constant after initialising it.

Classmate uses addresses `$31` and `$4E` to hold two of the data field prologue bytes.

7.18.4 The "LOCK" mystery

There is a special memory location in Applesoft (`$D6`) which is named the "AppleSoft Mystery Pa-



rameter" in What's Where In The Apple. It is also named "LOCK" in the Applesoft Internals disassembly, which gives a better idea of its purpose. When set to #$80, all Applesoft commands are interpreted as meaning "RUN." This prevents any user interaction at the Applesoft prompt. Tycoon uses this technique.

7.18.5 Stack

The stack is a single 256-bytes page ($100-1FF) in the Apple][. Since the standard Apple][environment does not have any source of interrupts, the stack can be considered to be a well-defined memory region. This means that code and data can be placed on the stack, and run from there, without regard to the value of the stack pointer, and modifications will not occur unexpectedly. (The effect on the stack of subroutine calling is an expected modification.) If an interrupt occurred, then the CPU would save the program counter and status register on the stack, thus corrupting the code or data that existed below the current stack pointer. (The corruption can even be above the stack pointer, if the stack pointer value is low enough that it wraps around!) Correspondingly, any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code or data that exist below the current stack pointer. Choplifter uses this technique.

7.18.6 Stack pointer

Since the standard Apple][environment does not have any source of interrupts, the stack pointer can be considered to be a register with well-defined value. This means that its value remains under program control at all times and that it can even be used as a general-purpose register, provided that the effect on the stack pointer of subroutine calling is expected by the program. Beer Run uses this technique.

LifeSaver also uses this technique for the purpose of obfuscating a transfer of control—the program checksums the pages of memory that were read in, and then uses the result as the new stack pointer, just prior to executing a "return from subroutine" instruction. Any alteration to the data, such as the insertion of breakpoints or detours, results in a different checksum and unpredictable behavior.

7.18.7 Input buffer

The input buffer is a single 256-bytes page ($200-2FF) in the Apple][. Code and data can be placed in the input buffer, and run from there. However, anything that the user types at the prompt, and which is routed through the ROM BIOS routine GETLN ($FD6A), will be written to the input buffer. Any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code in the input buffer. Karateka uses this technique.

7.18.8 Primary text screen

The primary text screen is a set of four 256-bytes pages ($400-7FF) in the Apple][. Code and data can be placed in the text screen memory, and run from there. The visible screen was usually switched to a blank graphics screen prior to that occurring, to avoid visibly displaying garbage, and perhaps causing the user to think that the program was malfunctioning. Obviously, any user interaction that occurs through the ROM BIOS routines, such as breaking to the prompt and typing commands, will cause corruption of the code in the text screen. Joust uses this technique to hold essential data.

7.18.9 Non-maskable interrupt vector

When a non-maskable interrupt (NMI) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at $FFFA-FFFB, and then starts executing from the specified address. The ROM BIOS handler immediately transfers control to the code at $3FB-3FD, which is usually a jump instruction to the complete NMI handler. For programs that were very heavily protected, such that inserting breakpoints was difficult because of hooked CSW and KSW vectors, for example, one alternative was to "glitch" the system by using a NMI card to force a NMI to occur. However, that technique required direct access to memory in order to install the jump instruction at $3FB-3FD, since the standard ROM BIOS does not place one there.

On a 64kb Apple][, the ROM BIOS could be copied into banked memory and made writable. The BIOS NMI vector could then be changed directly, potentially bypassing the user-defined NMI vector completely.

7.18.10 Reset vector

On a cold start, and whenever the user presses Ctrl-Reset, the Apple][reads the vector at `$FFFC-FFFD`, and then starts executing from the specified address. If the Apple][is configured with an Autostart ROM, then the warm-start vector at `$3F2-3F3` is used, if the "power-up" byte at `$3F4` matched the exclusive-OR of `#$A5` with the value at `$3F3`[57]. The values at `$3F2-3F4` are always writable, allowing a program to protect itself against a user pressing Ctrl-Reset in order to gain access to the monitor prompt, and then saving the contents of memory. The typical protected program response to Ctrl-Reset was to erase all of memory and then reboot.

On a 64kb Apple][, the ROM can be copied into banked memory and made writable. When the user presses Ctrl-Reset on an Apple][+, the ROM BIOS is not banked in first, meaning that the cold-start reset vector can be changed directly, and will be used, potentially bypassing the warm-start reset vector completely. On an Apple][e or later, the ROM BIOS is banked in first, meaning that the modified BIOS cold-start reset vector will never be executed, and so the warm-start reset vector cannot be overridden.

7.18.11 Interrupt request vector

Despite not having a source of interrupts in the default configuration, the Apple][did offer support for handling them. When an interrupt request (IRQ) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at `$FFFE-FFFF`, and then starts executing from the specified address. However, there is also a special case IRQ, which is triggered by the `BRK` instruction. This instruction is a single-byte breakpoint instruction, and is intended for debugging purposes. The ROM BIOS handler checks the source of the interrupt, and transfers control to the vector at `$3FE-3FF` if the source was an external interrupt. On the Autostart ROM, the ROM BIOS handler transfers control to the vector at `$3F0-3F1` if the source was a breakpoint. (Pre-Autostart ROMs simply dumped the register values to the screen, and then dropped to the monitor prompt instead.) The values at `$3F0-3F1`, and `$3FE-3FF` are always writable, allowing a program to protect itself against a user inserting breakpoints in order to break when execution reaches the specified address. The typical protected program response to breakpoints was to erase all of memory and then reboot. An alternative protection is to point `$3F0-3F1` to another `BRK` instruction, to produce an infinite loop and hang the machine. Bank Street Writer III uses this technique.

On a 64kb Apple][, the ROM BIOS can be copied into banked memory and made writable. The BIOS IRQ vector can then be changed directly, potentially bypassing the user-defined IRQ vector completely.

7.19 Catalog tricks

7.19.1 Control-"Break"

On a regular DOS disk, there is a sector called the Volume Table Of Contents (VTOC), which describes the starting location (track and sector) of the catalog, among other things. The catalog sectors contain the list on the disk of files which are accessible by DOS. For a file-based program, apart from the DOS and the catalog-related structures, all other content is accessible through the files listed in the catalog. DOS "knows" the track which holds the VTOC, since the track number (usually `#$11`) is hard-coded in DOS itself, and sector zero is assumed to be the one that holds the VTOC.

Since the files are listable, they can also be loaded from the original disk, and then saved to a copy of the disk. One way to prevent that is to insert control-characters in the filenames. Since control-characters are not visible from the DOS prompt, any attempt to load a file, using the name exactly as it appears, will fail.

Classmate uses this technique. It is also possible to embed backspace characters into the filename. Filenames with backspace characters in them cannot be loaded from the prompt. Instead, a Basic program must be written with printable characters as placeholders, and then the memory image must be altered to replace them with backspace characters®

7.19.2 Now you see it

Since the VTOC also carries the sector of the catalog, it can be altered to point to another location within the track that holds the VTOC. That causes

[57] This is true only when the full warm-start vector is not `#$00 #$E0 #$45` (`$E000` and `#$45`). If the vector is `$E000` and `#$45`, then the cold-start handler will change it to `$E003`, and resume execution from `$E000`. This behavior could have been used as an indirect transfer of control on the Apple][+, by jumping back to the cold-start handler, which would look like an infinite loop, but it would actually resume execution from `$E003`.

the disk to display a “fake” catalog, while allowing a program to access the real catalog sectors directly.

The Toy Shop uses this technique to show the program title, copyright, and author credits.

7.19.3 Now you don’t

Since DOS carries a hard-coded track number for the VTOC, it is easy to patch DOS to look at a different track entirely. The original default track can then be used for data. Any attempt to show the catalog from a regular DOS disk will display garbage.

Ali Baba uses this technique, by moving the entire catalog track to track five.

7.20 Basic tricks

7.20.1 Line linking

Circularly

In Basic on the Apple][, each line contains a reference to the next line to list. As such, several interesting effects are possible. For example, the listing can be made circular, by pointing to a previous line, causing an infinite loop of listing. The simplest example of that looks like this:

```
801:01 08 00 00 3A 00 00 00
```

This program contains one line whose line number is zero, and whose content is a single “:”. An attempt to list this program will show an infinite number of “0 :” lines. However it can be executed without issue.

Missing

The listing can be forced to skip lines, by pointing to a line that appears after the next line, like this:

```
801:10 08 00 00 3A 00 10 08 01 00 BA 22
80D:31 22 00 16 08 02 00 3A 00 00 00
```

Listing the program will show two lines:

```
0  :
2  :
```

However, there is a second line (numbered “one”) which contains a PRINT statement. Running the program will display the text in line one.

Out-of-order

The listing can list lines in an order that does not match the execution, for example, backwards:

```
801:13 08 03 00 BA 22 30 22 00 1C 08 01 00 BA
22
810:31 22 00 0A 08 03 00 BA 22 32 22 00 00 00
```

This program contains three lines, numbered from zero to two. The list will show the second and third lines in reverse order. The illusion is completed by altering the line number of the first line to a value larger than the other lines. However, the execution of the first line first cannot be altered in this way.

Out-of-bounds

The listing can even be forced to fetch from arbitrary memory, such as the graphics screen or the memory-mapped I/O space:

```
801:55 C0 00 00 3A 00 00 00
```

This program contains a single line whose line number is zero, and whose content is a single “:”. An attempt to list this program will cause the second text screen to be displayed instead, and the machine will appear to crash. Further misdirection is possible by placing an entirely different program at an alternative location, which will be listed instead⊛

Imagine the feeling when the drive light turns itself on while the program is being listed!

It might even be possible to create a program with lines that touch the memory-mapped I/O space, and activate or deactivate a stepper-motor phase. If those lines were listed in a specific order, then the drive could be enticed to move to a different track. That track could lie about its position on the disk, but carry alternative content to the proper track, resulting in perhaps subtly different behavior. Are we having fun yet?

7.20.2 Start address

The first line of code to execute can be altered dynamically at runtime, by a “`POKE 103, <low addr>`” and/or “`POKE 104, <high addr>`”, followed by a “RUN” command. Math Blaster uses this technique.

7.20.3 Line address

Normally, the execution will generally proceed linearly through the program (excluding instructions that legally transfer control, such as subroutine calls and loops), regardless of the references to individual lines. However, the next line (technically, the next token) to execute can be altered dynamically at runtime, by a “POKE 184, <low addr>”. The first value at the new location must be a ’:’ character. For example, this program:

```
0 POKE 184,14 : END : PRINT "!"
```

will skip the “`END`” token and print the ’!’ instead. It is also possible to alter the high address by a “`POKE 185, <high address>`” as well, but it requires that the second POKE is placed at the new location, which is determined by the new value of the high address and the old value of the low address. It cannot be placed immediately after the address of the first POKE, because that location will not be accessed anymore.

7.20.4 “REM crash”

```
801:0E 08 00 00 B2 0D 04 50 52 23 36 0D 00 00
00
```

This program contains one line, which looks like the following, where the “^” character stands for the Control key.

```
0 REM^M^DPR#6^M
```

When listed with DOS active, it will trigger a reboot. It works because the same I/O routine is used for displaying the text as for typing commands from the keyboard. Zardax uses this technique.

7.20.5 Self-modification

A program can even modify itself dynamically at runtime. For example, this program will display “2” instead of “1”. The address of the POKE corresponds to the location of the text in memory.

```
0 POKE 2064,50 : PRINT "1"
```

A program can also extend its code dynamically at runtime:

```
0 DATA 130,58
1 FOR I=0 TO 1 : READ X : POKE 2086+I,X :
```

A “FOR” loop must be terminated by a “NEXT” token, in order to be legal code. Notice that the program does not contain a “NEXT” token, as expected. Instead, the values in the DATA line supply the “NEXT” token and a subsequent “:”. The inclusion of a “:” allows extending the line further, simply by adding more values to the “DATA” line and altering the corresponding address of the “POKE”.

By using this technique, even entirely new lines can be created.

7.21 Rastan

Rastan is mentioned here only because it is a title for an Apple][system (okay, the IIGS) that carried the means to bypass its own copy-protection! The program contained two copy-protection techniques. One was a disk verification check, which executed shortly after inserting the second disk. The other was a checksum routine which performed part of the calculation between each graphics frame, until it formed the complete value. If the match failed, only then would it display a message. It means that the game would run for a little while before failing, making it extremely difficult to determine where the check was performed.

7.21.1 The Rastan backdoor

In order to avoid waiting for the protection check every time a new version of the code was built, the author[58] inserted a “backdoor” routine which executed before the first protection check could run. The backdoor routine had the ability to disable both protection checks in memory, as well as to add new functionality, such as invincibility and level warping. And where was this backdoor routine located? Inside the highscore file!

Yes. The highscore file had a special format, whereby code could be placed beginning at the third byte of the file. As long as the checksum of the file was valid (an exclusive-OR of every byte of the file yielded a zero), the code would be executed.

Here is the dispatcher code in Rastan:

```
.A16
;checksum data
2000D       JSR       $21216
;note this address
20010       JSR       $2D1C2
```

[58] https://twitter.com/JBrooksBSI

Here is the checksum routine:

```
.A16
;source address
21216     TXA
;taken if no highscore file
21217     BEQ     $21240
;length of data
21219     LDA     $0,X
2121D     TAY
2121E     SEP     #$20
.A8
21220     PHX
;checksum seed
21221     LDA     #0
;checksum data
21223     EOR     $0,X
21227     INX
21228     DEY
21229     BNE     $21223
2122B     PLX
2122C     REP     #$30
.A16
2122E     AND     #$FF
;taken if bad checksum, no copy
21231     BNE     $21240
;length of data
21233     LDA     $0,X
21237     DEC
21238     LDY     #$D1C0
;copy to $2D1C0
2123B     MVN     #2, #0
2123E     PHK
2123F     PLB
21240     RTS
```

We can see that the data are copied to `$2D1C0`, the first word is the length of the data, and the first byte after the length (so `$2D1C2`) is executed directly in 16-bit mode. By default, the file carried an immediate return instruction, but it could have been anything, including this:

```
;always pass protection
;(BRA $+$0F)
2D1C2     LDA     #$0D80
2D1C5     STA     $22004
;always pass checksum
;(BRA $+$19)
2D1C8     LDA     #$1780
2D1CB     STA     $3CAD0
2D1CE     RTS
```

7.22 Conclusion

There were many tricks used to protect programs on the Apple][, and what is listed here is not even all of them. Copy-protection and cracking were part of a never-ending cycle of invention and advances on both sides. As the protectors came to understand the hardware more and more, they were able to develop techniques like delayed fetch, or consecutive quarter-tracks. The crackers came up with NMI cards, and the mighty E.D.D. In response, the protectors hooked the NMI vector and exploited a vulnerability in E.D.D.'s read routine. (This is my absolute favorite technique.) The crackers just boot-traced the whole thing.

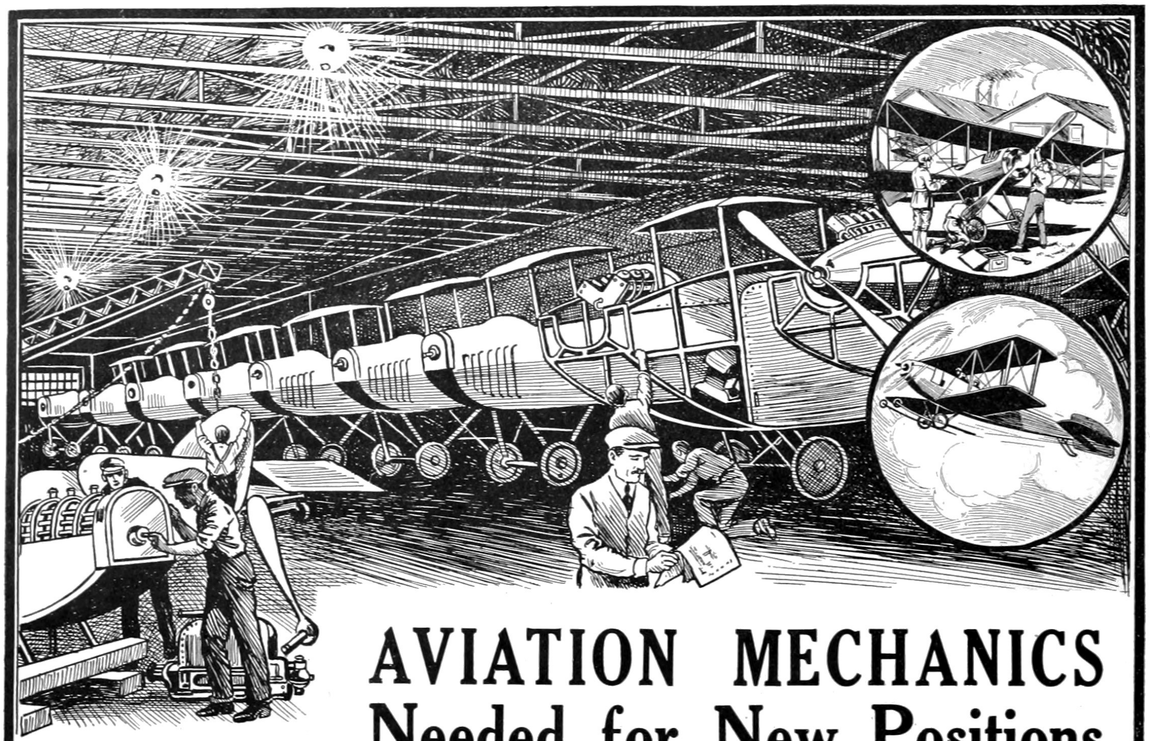
We can only stand and admire the ingenuity and inventiveness of the protectors like Roland Gustafsson or John Brooks. They were helped by the openness of the Apple][platform and especially its disk system. Even today, we see some of the same styles of protections—anti-disassembly, self-modifying code, compression, and, of course, anti-debugging.

The cycle really is never-ending.

7.23 Acknowledgements

Thanks to William F. Luebbert for What's Where In The Apple, and Don Worth and Pieter Lechner for Beneath Apple DOS. Both books have been on my bookshelf since 1983, and were consulted very often while writing this paper.

Thanks to reviewers 4am, Olivier Guinart, and John Brooks, for their invaluable input⊛



AVIATION MECHANICS
Needed for New Positions

Thousands of splendid new positions now opening up everywhere in this amazing new field. New Airplane factories being built—automobile and other plants in all parts of the country being converted to turn out vast fleets of Airplanes for our armies in Europe. And only a few hundred expert Airplane Mechanics available, although thousands are needed. And this is only the beginning. Already airplane mail routes are being planned for after the war and thousands upon thousands of flying machines will be wanted for express and passenger carrying service.

Not in a hundred years has any field of endeavor held out such wonderful chances to young men as are offered to you today in the Aviation Industry. Resolve now to change your poorly paid job for a big paying position with a brilliant future. Send the coupon today for Special limited offer in Practical Aeronautics and the Science of Aviation and prepare yourself in a few short months to double or treble your present salary.

Earn $50 to $300 per week as

Aeronautical Instructor	$60 to $150 per week
Aeronautical Engineer	$100 to $300 per week
Aeronautical Contractor	Enormous profits
Aeroplane Repairman	$60 to $75 per week
Aeroplane Mechanician	$40 to $60 per week
Aeroplane Inspector	$50 to $75 per week
Aeroplane Salesman	$5000 per year and up
Aeroplane Assembler	$40 to $65 per week
Aeroplane Builder	$75 to $200 per week

What Our Students Say:

Mr. Stanfield Fries
Fort Bliss, Tex.
My estimation of the new course is excellent; it could positively not be any better.

Mr. Z. Purdy
Shreveport, La.
It is hard to believe that lessons on such a subject could be gotten up in such an interesting manner.

Mr. Lloyd Royer
Haigler, Neb.
I can hardly thank you enough for the way you have personally taken up my enrollment.

Mr. Mayne Eble
Manistee, Mich.
I believe I learn more from my lessons than an aviator who takes his first lesson with an airman in an aeroplane.

We Teach You By Mail

IN YOUR SPARE TIME AT HOME

Our new, scientific Course has the endorsement of airplane manufacturers, aeronautical experts, aviators and leading aero clubs. Every Lesson, Lecture, Blue-Print and Bulletin is self-explanatory. You can't fail to learn. No book study. No schooling required. Lessons are written in non-technical, easy-to-understand language. You'll not have the slightest difficulty in mastering them. The Course is absolutely authoritative and right down to the minute in every respect. Covers the entire field of Practical Aeronautics and Science of Aviation in a thorough practical manner. Under our expert direction, you get just the kind of practical training you must have in order to succeed in this wonderful industry.

Special Offer NOW! *SEND THIS COUPON TODAY*

It is our duty to help in every possible way to supply the urgent need for graduates of this great school. We have facilities for teaching a few more students, and to secure them quickly we are making a remarkable Special Offer which will be withdrawn without notice. Write today—or send the coupon—for full particulars. Don't risk delay. Do it now.

AMERICAN SCHOOL OF AVIATION
431 S. Dearborn Street Dept. 7442 CHICAGO, ILL.

American School
of Aviation
431 S. Dearborn Street
Dept. 7442
Chicago, Illinois

Without any obligations on my part, you may send me full particulars of your course in Practical Aeronautics and your Special LIMITED Offer.

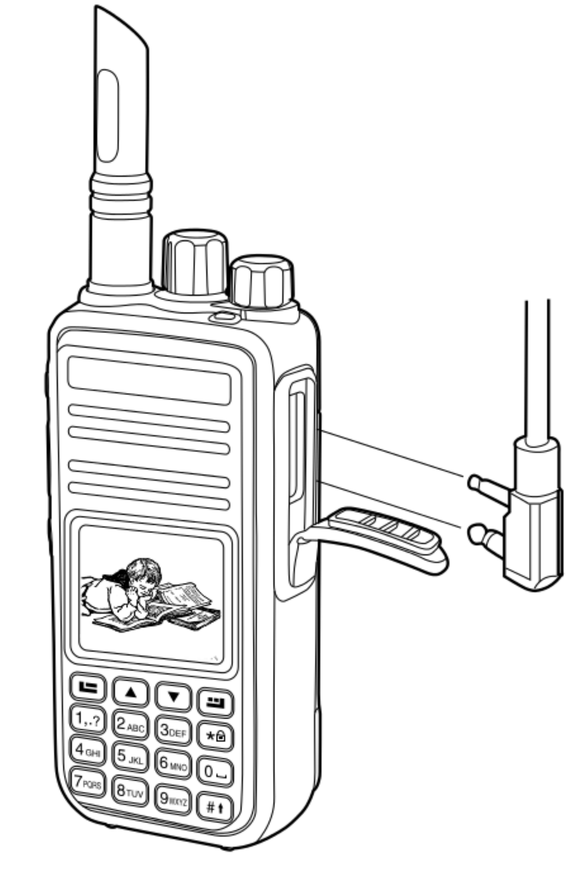
Name..............................

Address..............................

....

8 Reverse Engineering the Tytera MD380

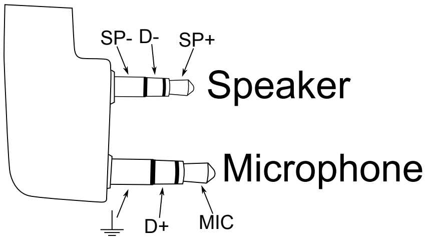
by Travis Goodspeed KK4VCZ,
with kind thanks to DD4CR and W7PCH.



The following is an adventure of reverse engineering the Tytera MD380, a digital hand-held radio that can be had for barely more than a hundred bucks. In this article, I explain how to read and write the radio's configuration over USB, and how to break the readout protection on its firmware, so that you fine readers can write your own strange and clever software for this nifty gizmo. I also present patches to promiscuously receive audio from unknown talkgroups, creating the first hardware scanner for DMR. Far more importantly, these notes will be handy when you attempt to reverse engineer something similar on your own.

This article does not go into the security problems of the DMR protocol, but those are sufficiently similar to P25 that I'll just refer you to *Why (Special Agent) Johnny (Still) Can't Encrypt* by Sandy Clark and Friends.[59]

8.1 Hardware Overview



The MD380 is a hand-held digital voice radio that uses either analog FM or Digital Mobile Radio (DMR). It is very similar to other DMR radios, such as the CS700 and CS750 from Connect Systems.[60]

DMR is a trunked radio protocol using two-slot TDMA, so a single repeater tower can be used by one user in Slot 1 while another user is having a completely different conversation on Slot 2. Just like GSM, the tower coordinates which radio should transmit when.

The CPU of this radio is an STM32F405 from STMicroelectronics. This contains a Cortex M4, so all instructions are Thumb and all function pointers are odd. The LQFP100 package of this chip is used. It has a megabyte of Flash and 192 kilobytes of RAM. The STM32 has both JTAG and a ROM bootloader, but both of these are protected by a Readout Device Protection (RDP) feature. In Section 8.8, I'll show you how to bypass these protections and jailbreak your radio.

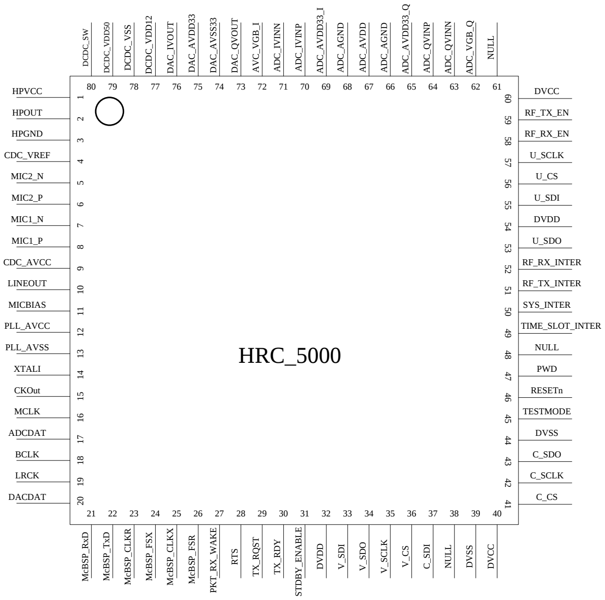
There is also a radio baseband chip, the HR C5000. At first I was reconstructing the pinout of this chip from the CS700 Service Manual, but the full documentation can be had from DocIn, a Chinese PDF sharing website. 中国排名第一。

Aside from a bunch of support components that we can take for granted, there is an SPI Flash chip for storing the codeplug. "Codeplug" is a Motorola term for the radio settings, such as frequencies, contacts, and talk groups; I use the term here to distinguish the radio configuration in SPI Flash from the

[59] `unzip pocorgtfo10.pdf p25sec.pdf #from Proceedings of the 20th Usenix Security Symposium in 2011`

[60] The folks at Connect Systems are nice and neighborly, so please buy a radio from them.

code and data in CPU Flash.



8.2 A Partial Dump

From `lsusb -v` on Linux, we can see that the device implements USB DFU, most likely as a fork of some STMicro example code. The MD380 appears as an STMicro DFU device with storage for Internal Flash and SPI Flash with a VID:PID of `0483:df11`.

```
iMac% dfu−util −list
Found DFU: [0483:df11]
     devnum=0, cfg=1, intf=0, alt=0,
     name="@Internal Flash
            /0x08000000/03*016Kg"
Found DFU: [0483:df11]
     devnum=0, cfg=1, intf=0, alt=1,
     name="@SPI Flash Memory
            /0x00000000/16*064Kg"
```

Further, the `.rdt` codeplug files are SPI Flash images in the DMU format, which is pretty much just wrapper with a bare minimum of metadata around a flat, uncompressed memory image. These codeplug files contain the radio's contact list, repeater frequencies, and other configuration info. We'll get back to this later, as what we really want to do is dump and patch the firmware.

Unfortunately, dumping memory from the device by the standard DFU protocol doesn't seem to yield useful results, just the same repeating binary string, regardless of the alternate we choose or the starting position.

```
iMac% dfu−util −d 0483:df11 −−alt 1 −s 0:0x200000 −U
     first1k.bin
Filter on vendor = 0x0483 product = 0xdf11
Opening DFU capable USB device... ID 0483:df11
Run−time device DFU version 011a
Found DFU: [0483:df11] devnum=0, cfg=1, intf=0, alt=1,
          name="@SPI Flash Memory /0x00000000/16*064Kg"
Claiming USB DFU Interface...
Setting Alternate Setting #1 ...
Determining device status: state = dfuUPLOAD−IDLE
aborting previous incomplete transfer
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 011a
Device returned transfer size 1024
Limiting default upload to 2097152 bytes
bytes_per_hash=1024
Starting upload: [####...####] finished!
iMac% hexdump first1k.bin
0000000 30 1a 00 20 15 56 00 08 29 54 00 08 2b 54 00 08
0000010 2d 54 00 08 2f 54 00 08 31 54 00 08 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 33 54 00 08
0000030 35 54 00 08 00 00 00 00 83 30 00 08 37 54 00 08
0000040 61 56 00 08 65 56 00 08 69 56 00 08 5b 54 00 08
...
00003c0 10 eb 01 60 df f8 34 1a 08 60 df f8 1c 0c 00 78
00003d0 40 28 c0 f0 e6 81 df f8 24 0a 00 68 00 f0 0e ff
00003e0 df e1 df f8 10 1a 09 78 a2 29 0f d1 df f8 f8 19
00003f0 09 68 02 29 0a d1 df f8 00 0a 02 21 01 70 df f8
... [same 1024 bytes repeated]
```

In this brave new world, where folks break their bytes on the little side by order of Golbasto Momarem Evlame Gurdilo Shefin Mully Ully Gue, Tyrant of Lilliput and Eternal Enemy of Big Endians and Blefuscu, to break them on the little side, it's handy to spot four byte sequences that could be interrupt handlers. In this case, what we're looking at is the first few pointers of an interrupt vector table. This means that we are grabbing memory from the beginning of internal flash at `0x08000000`!

Note that the data repeats every kilobyte, and also that `dfu-util` is reporting a transfer size of 1,024 bytes. The `-t` switch will order `dfu-util` to dump more than a kilobyte per transfer, but everything after the first transfer remains corrupted.

This is because `dfu-util` isn't sending the proper commands to the radio firmware, and it's getting the page as a bug rather than through proper use of the protocol. (There are lots of weird variants of DFU, created by folks only using DFU with their own tools and never testing for compatibility with each other. This variant is particularly weird, but manageable.)

8.3 Tapping USB with VMWare

Before going further, it was necessary to learn the radio's custom dialect of DFU. Since my Total Phase USB sniffers weren't nearby, I used VMWare to sniff the transactions of both the MD380's firmware updater and codeplug configuration tools.

I did this by changing a few lines of my VMWare `.vmx` configuration to dump USB transactions out to `vmware.log`, which I parsed with ugly regexes in Python. These are the additions to the `.vmx` file.

```
monitor = "debug"
usb.analyzer.enable = TRUE
usb.analyzer.maxLine = 8192
mouse.vusb.enable = FALSE
```

The logs showed that the MD380's variant of DFU included non-standard commands. In particular, the LCD screen would say "PC Program USB Mode" for the official client applications, but not for any 3rd party application. Before I could do a proper read, I had to find the commands that would enter this programming mode.

DFU normally hides extra commands in the `UPLOAD` and `DNLOAD` commands when the block address is less than two. (Hiding them in blocks `0xFFFF` and `0xFFFE` would make more sense, but if wishes were horses, then beggars would ride.)

To erase a block, a DFU host sends `0x41` followed by a little endian address. To set the address pointer (block 2's address), the host sends `0x21` followed by a little endian address.

In addition to those standard commands, the MD380 also uses a number of two-byte (rather than five-byte) `DNLOAD` transactions, none of which exist in the standard DMU protocol. I observed the following, which I still only partially understand.

Non-Standard DNLOAD Extensions

`91 01`	Enables programming mode on LCD.
`a2 01`	Seems to return model number.
`a2 02`	Sent only by config read.
`a2 31`	Sent only by firmware update.
`a2 03`	Sent by both.
`a2 04`	Sent only by config read.
`a2 07`	Sent by both.
`91 31`	Sent only by firmware update.
`91 05`	Reboots, exiting programming mode.

8.4 Custom Codeplug Client

Once I knew the extra commands, I built a custom DFU client that would send them to read and write codeplug memory. With a little luck, this might have given me control of firmware, but as you'll see, it only got me half way.

Because I'm familiar with the code from a prior target, I forked the DFU client from an old version of Michael Ossmann's Ubertooth project.[61]

Sure enough, changing the VID and PID of the `ubertooth-dfu` script was enough to start dumping memory, but just like `dfu-util`, the result was a repeating sequence of the first block's contents. Because the block size was 256 bytes, I received only the first `0x100` bytes repeated.

Adding support for the non-standard commands in the same order as the official software, I got a copy of the complete 256K codeplug from SPI Flash instead of the beginning of Internal Flash. Hooray!

To upload a codeplug back into the radio, I modified the `download()` function to enable programming mode and properly wait for the state to return to `dfuDNLOAD_IDLE` before sending each block.

This was enough to write my own codeplug from one radio into a second, but it had a nasty little bug! I forgot to erase the codeplug memory, so the radio got a bitwise AND of two valid codeplugs.[62]

A second trip with the USB sniffer shows that these four blocks were erased, and that the upload address must be set to zero *after* the erasure.
`0x00000000 0x00010000 0x00020000 0x00030000`

Erasing the blocks properly gave me a tool that correctly reads and writes the radio codeplug!

8.5 Codeplug Format

Now that I could read and write the codeplug memory of my MD380, I wanted to be able to edit it. Parts of the codeplug are nice and easy to reverse, with strings as UTF16L and numbers being either integers or BCD. Checksums don't seem to matter, and I've not yet been able to brick my radios by uploading damaged firmware images.

The Radio Name is stored as a string at `0x20b0`, while the Radio ID Number is an integer at `0x2080`. The intro screen's text is stored as two strings at `0x2040` and `0x2054`.

```
#seekto 0x5F80;
struct {
  ul24 callid;    //DMR Account Number
  u8   flags;     //c2 private, no tone
                  //e1 group, with rx tone
  char name[32];  //U16L chars
} contacts[1000];
```

[61] In particular, I used r543 of the old SVN repository, a version from 4 July 2012.

[62] See PoC||GTFO 2:5.

[63] `http://chirp.danplanet.com`

CHIRP,[63] a ham radio application for editing radio codeplugs, has a bitwise library that expects memory formats to be defined as C structs with base addresses. By loading a bunch of contacts into my radio and looking at the resulting structure, it was easy to rewrite it for CHIRP.

Repeatedly changing the codeplug with the manufacturer's application, then comparing the hexdumps gave me most of the radio's important features. Patience and a few more rounds will give me the rest of them, and then my CHIRP plugin can be cleaned up for inclusion.

Unfortunately, not everything of importance exists within the codeplug. It would be nice to export the call log or the text messages, but such commands don't exist and the messages themselves are nowhere to be found inside of the codeplug. For that, we'll need to break into the firmware.

8.6 Dumping the Bootloader

Now that I had a working codeplug tool, I'd like a cleartext dump of firmware. Recall from Section 8.2 that forgetting to send the custom command `0x91 0x01` leaves the radio in a state where the beginning of code memory is returned for every read. This is an interrupt table!

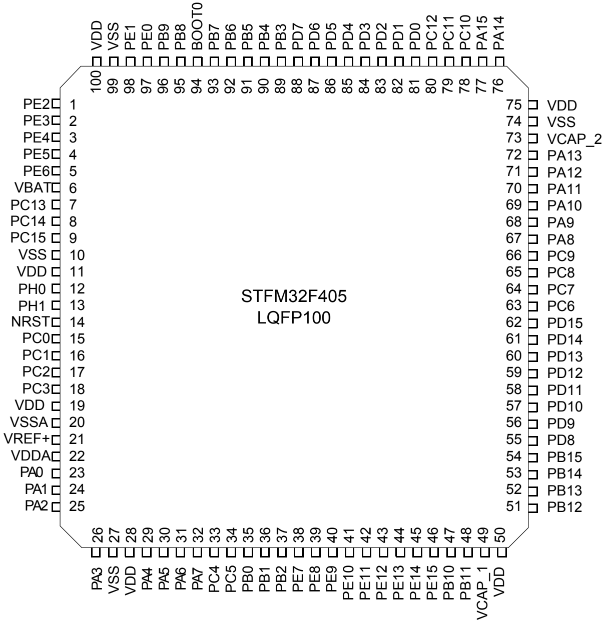
MD380 Recovery Bootloader Interrupts

`0x20001a30`	Top of the call stack.
`0x08005615`	Reset Handler
`0x08005429`	Non-Maskable Interrupt (NMI)
`0x0800542b`	Hard Fault
`0x0800542d`	MMU Fault
`0x0800542f`	Bus Fault
`0x08005431`	Usage Fault

From this table and the STM32F405 datasheet, we know the code flash begins at `0x08000000` and RAM begins at `0x20000000`. Because the firmware updater only writes to regions at and after `0x0800C000`, we can guess that the first 48k are a recovery bootloader, with the region after that holding the application firmware. As all of the interrupts are odd, and because the radio uses a Cortex M4 core, we know that the firmware is composed exclusively of Thumb (and Thumb2) code, with no old fashioned ARM instructions.

Sure enough, I was able to dump the whole bootloader by reading a single page of `0xC000` bytes from the application mode. This bootloader is the one used for firmware updates, which can be started by holding PTT and the unlabeled button above it when turning on the power switch.[64]

This trick doesn't expose enough memory to dump the application, but it was valuable to me for two very important reasons. First, this bootloader gave me some proper code to begin reverse engineering, instead of just external behavioral observations. Second, the recovery bootloader contains the keys and code needed to decrypt an application image, but to get at that decrypted image, I first had to do some soldering.

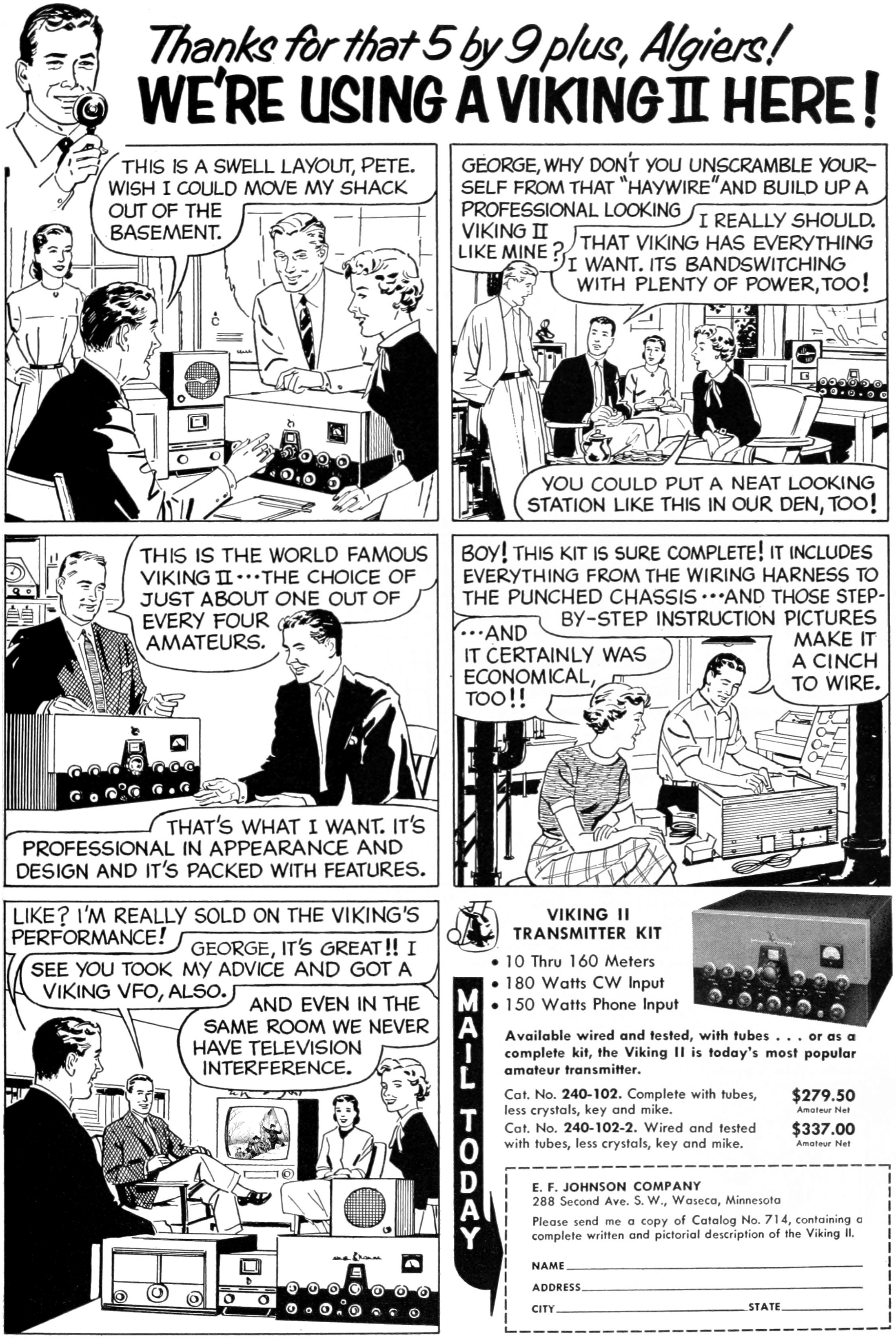


8.7 Radio Disassembly (BOOT0 Pin)

As I stress elsewhere, the MD380 has *three* applications in it: (1) Tytera's Radio Application, (2) Tytera's Recovery Bootloader, and (3) STMicro's Bootloader ROM. The default boot process is for the Recovery Bootloader to immediately start the Radio Application unless Push-To-Talk (PTT) and the button above it are held during boot, in which case it waits to accept a firmware update. There is no key sequence to start the STMicro Bootloader ROM, so a bit of disassembly and soldering is required.

This ROM contains commands to read and write all of memory, as well as to begin execution at any arbitrary address. These commands are initially locked down, but in Section 8.8, I'll show how to get around the restrictions.

[64] Transfers this large work on Mac but not Linux.


Thanks for that 5 by 9 plus, Algiers!
WE'RE USING A VIKING II HERE!
THIS IS A SWELL LAYOUT, PETE. WISH I COULD MOVE MY SHACK OUT OF THE BASEMENT.
GEORGE, WHY DON'T YOU UNSCRAMBLE YOURSELF FROM THAT "HAYWIRE" AND BUILD UP A PROFESSIONAL LOOKING VIKING II LIKE MINE?
I REALLY SHOULD. THAT VIKING HAS EVERYTHING I WANT. ITS BANDSWITCHING WITH PLENTY OF POWER, TOO!
YOU COULD PUT A NEAT LOOKING STATION LIKE THIS IN OUR DEN, TOO!
THIS IS THE WORLD FAMOUS VIKING II ··· THE CHOICE OF JUST ABOUT ONE OUT OF EVERY FOUR AMATEURS.
THAT'S WHAT I WANT. IT'S PROFESSIONAL IN APPEARANCE AND DESIGN AND IT'S PACKED WITH FEATURES.
BOY! THIS KIT IS SURE COMPLETE! IT INCLUDES EVERYTHING FROM THE WIRING HARNESS TO THE PUNCHED CHASSIS ··· AND THOSE STEP-BY-STEP INSTRUCTION PICTURES MAKE IT A CINCH TO WIRE.
··· AND IT CERTAINLY WAS ECONOMICAL, TOO!!
LIKE? I'M REALLY SOLD ON THE VIKING'S PERFORMANCE!
GEORGE, IT'S GREAT!! I SEE YOU TOOK MY ADVICE AND GOT A VIKING VFO, ALSO.
AND EVEN IN THE SAME ROOM WE NEVER HAVE TELEVISION INTERFERENCE.
MAIL TODAY

VIKING II TRANSMITTER KIT

- 10 Thru 160 Meters
- 180 Watts CW Input
- 150 Watts Phone Input

Available wired and tested, with tubes . . . or as a complete kit, the Viking II is today's most popular amateur transmitter.

Cat. No. **240-102.** Complete with tubes, less crystals, key and mike.	**$279.50** Amateur Net
Cat. No. **240-102-2.** Wired and tested with tubes, less crystals, key and mike.	**$337.00** Amateur Net

E. F. JOHNSON COMPANY
288 Second Ave. S. W., Waseca, Minnesota

Please send me a copy of Catalog No. 714, containing a complete written and pictorial description of the Viking II.

NAME ____________________

ADDRESS ____________________

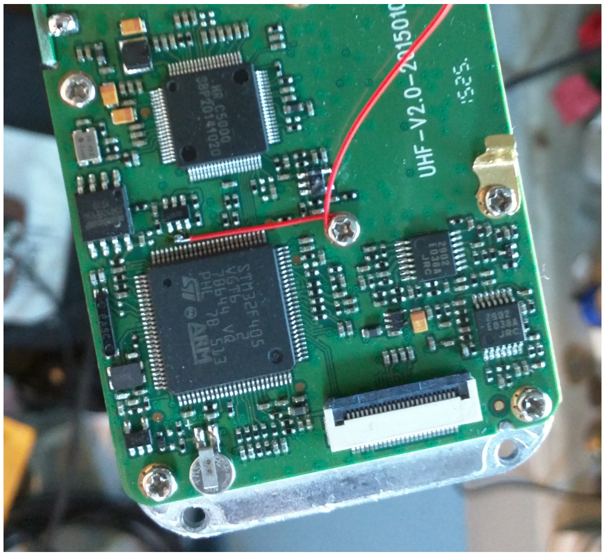
CITY ____________________ STATE ____________

To open your radio, first remove the battery and the four Torx screws that are visible from the back of the device. Then unscrew the antenna and carefully pry off the two knob covers. Beneath each knob and the antenna, there are rings that screw in place to secure them against the radio case; these should be moved by turning them counter-clockwise using a pair of sturdy, dull tweezers.

Once the rings have been removed, the radio's main board can be levered up at the bottom of the radio, then pulled out. Be careful when removing it, as it is attached with a Zero Insertion Force (ZIF) connector to the LCD/Keypad board, as well as by a short connector to the speaker.

The STMicro Bootloader is started by pulling the BOOT0 pin of the STM32F405 high while restarting the radio. I did this by soldering a thin wire to the test pad near that pin, wrapping the wire around a screw for strain relief, then carefully feeding it out through the microphone/speaker port.

(An alternate method involves removing BOOT0's pull-down resistor, then fly-wiring it to the pull-up on the PTT button. Thanks to tricky power management, this causes the radio to boot normally, but to *reboot* into the Mask ROM.)



8.8 Bootloader RE

Once I finally had a dump of Tytera's bootloader, it was time to reverse engineer it.[65]

The image is 48K in size and should be loaded to 0x08000000. Additionally, I placed 192K of RAM at 0x20000000. It's also handy to create regions for the I/O banks of the chip, in order to help track those accesses. (IDA and Radare2 will think that peripherals are global variables near 0x40000000.)

After wasting a few days exploring the command set, I had a decent, if imperfect, understanding of the Tytera Bootloader but did not yet have a cleartext copy of the application image. Getting a bit impatient, I decided to patch the bootloader to keep the device unprotected while loading the application image using the official tools.

I had to first explore the STM32 Standard Peripheral Library to find the registers responsible for locking the chip, then hunt for matching code.

```
/* STM32F4xx flash regs from stm32f4xx.h */
#@0x40023c00
typedef struct {
  __IO uint32_t ACR;      //access ctrl    0x00
  __IO uint32_t KEYR;     //key            0x04
  __IO uint32_t OPTKEYR;  //option key     0x08
  __IO uint32_t SR;       //status         0x0C
  __IO uint32_t CR;       //control        0x10
  __IO uint32_t OPTCR;    //option ctrl    0x14
  __IO uint32_t OPTCR1;   //option ctrl 1  0x18
} FLASH;
```

[65] The MD5 of my image is 721df1f98425b66954da8be58c7e5d55, but you might have a different one in your radio.

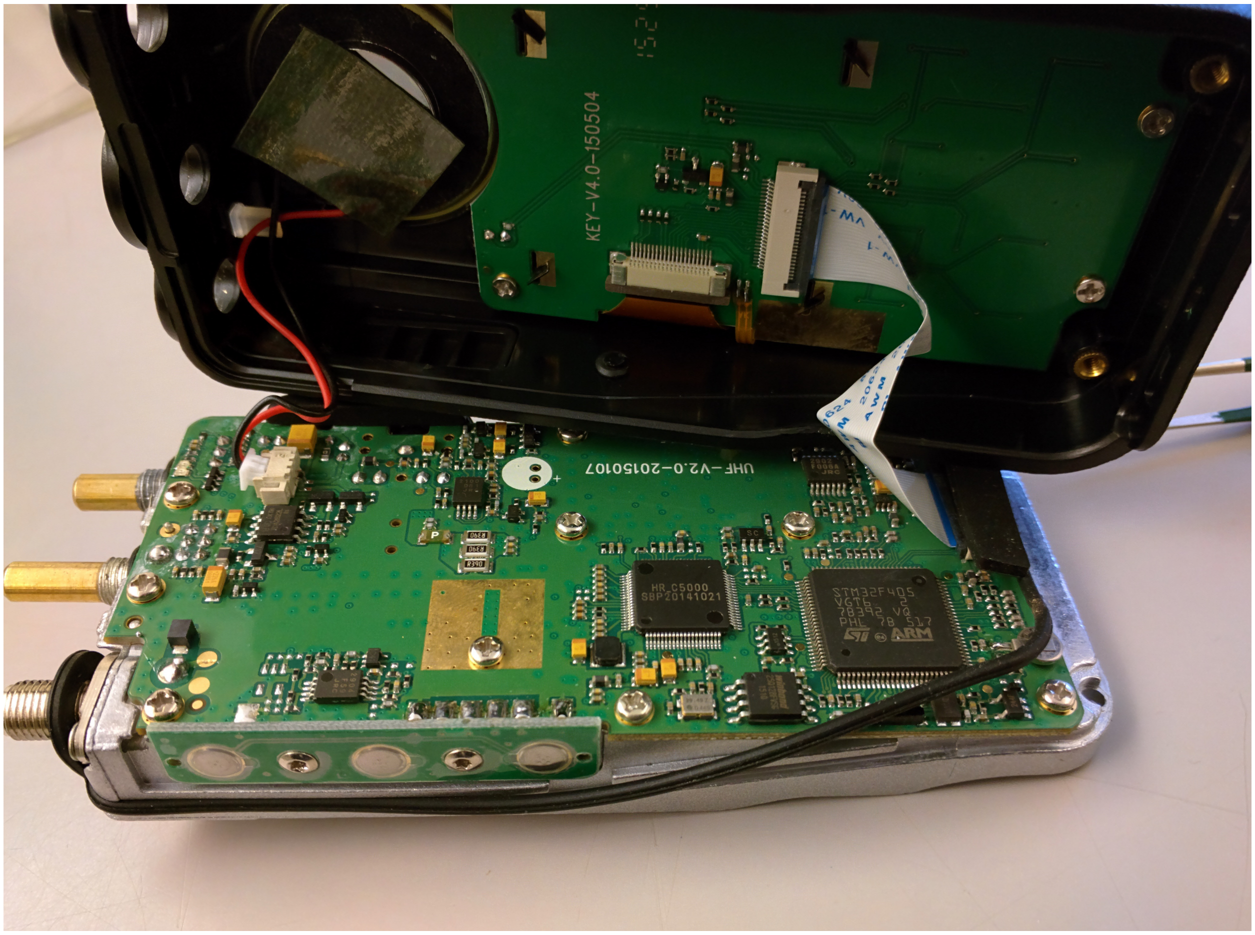
The way flash protection works is that byte 1 of FLASH->OPTCR (at 0x40023C15) is set to the protection level. 0xAA is the unprotected state, while 0xCC is the permanent lock. Anything else, such as 0x55, is a sort of temporary lock that allows the application to be wiped away by the Mask ROM bootloader, but does not allow the application to be read out.

Tytera is using this semi-protected mode, so you can pull the BOOT0 pin of the STM32F4xx chip high to enter the Mask ROM bootloader.[66] This process is described in Section 8.7.

Sure enough, at 0x08001FB0, I found a function that's very much like the example FLASH_OB_RDPConfig function from stm32f4xx_flash.c. I call the local variant rdp_lock().

```
/* Sets the read protection level.
 * OB_RDP specifies the protection level.
 *        AA: No protection.
 *        55: Read protection memory.
 *        CC: Full chip protection.
 * WARNING: When enabling OB_RDP level 2
 *          it's no longer possible to go
 *          back to level 1 or 0.
 */
void FLASH_OB_RDPConfig(uint8_t OB_RDP){
  FLASH_Status status = FLASH_COMPLETE;

  /* Check the parameters */
  assert_param(IS_OB_RDP(OB_RDP));

  status = FLASH_WaitForLastOperation();
  if(status == FLASH_COMPLETE)
    *(__IO uint8_t*)
      OPTCR_BYTE1_ADDRESS = OB_RDP;
}
```

[66] Confusingly enough, this is the *third* implementation of DFU for this project! The radio application, the recovery bootloader, and the ROM bootloader all implement different variants of DFU. Take care not to confuse the them.



This function is called from `main()` with a parameter of `0x55` in the instruction at `0x080044A8`.

```
       0x080044a0    fdf7a0fd        bl rdp_isnotlocked
       0x080044a4    0028            cmp r0, 0
   ,=< 0x080044a6    04d1            bne 0x80044b2
   |   ; Change this immediate from 0x55 to 0xAA
   |   ; to jailbreak the bootloader.
   |   0x080044a8    5520            movs r0, 0x55
   |   0x080044aa    fdf781fd        bl rdp_lock
   |   0x080044ae    fdf78bfd        bl rdp_applylock
   `-> 0x080044b2    fdf776fd        bl 0x8001fa2
       0x080044b6    00f097fa        bl bootloader_pin_test
```

Patching that instruction to instead send `0xAA` as a parameter prevents the bootloader from locking the device. (We're just swapping `aa 20` in where `55 20` used to be.)

```
iMac% diff old.txt jailbreak.txt
< 00044a0 fd f7 a0 fd 00 28 04 d1
          55 20 fd f7 81 fd fd f7
---
> 00044a0 fd f7 a0 fd 00 28 04 d1
          aa 20 fd f7 81 fd fd f7
```

8.9 Dumping the Application

Once I had a jailbroken version of the recovery bootloader, I flashed it to a development board and installed an encrypted MD380 firmware update using the official Windows tool. Sure enough, the application installed successfully!

After the update was installed, I rebooted the board into its ROM by holding the `BOOT0` pin high. Since the recovery bootloader has been patched to leave the chip unlocked, I was free to dump all of Flash to a file for reverse engineering and patching.

8.10 Reversing the Application

Reverse engineering the application isn't terribly difficult, provided a few tricks are employed. In this section, I'll share a few; note that all pointers in this section are specific to Version 2.032, but similar functionality exists in newer firmware revisions.

At the beginning, the image appears almost entirely without symbols. Not one function or system call comes with a name, but it's easy to identify a few strings and I/O ports. Starting from those, related functions—those in the same `.C` source file—are often located next to one another in memory, providing hints as to their meaning.

The operating system for the application is an ARM port of MicroC/OS-II, an embedded real-time operating system that's quite well documented in the book of the same name by Jean J. Labrosse. A large function at `0x0804429C` that calls the operating system's `OSTaskCreateExt` function to make a baker's dozen of threads. Each of these conveniently has a name, conveniently describing the system interrupt, the real-time clock timer, the RF PLL, and other useful functions.

As I had already reverse engineered most of the SPI Flash codeplug, it was handy to work backward from codeplug addresses to identify function behavior. I did this by identifying `spiflash_read` at `0x0802fd82` and `spiflash_write` at `0x0802fbea`, then tracing all calls to these functions. Once these have been identified, finding codeplug functions is easy. Knowing that the top line of startup text is 32 bytes stored at `0x2040` in the codeplug, finding the code that prints the text is as simple as looking for calls to `spiflash_read(&foo, 0x2040, 20)`.

Thanks to the firmware author's stubborn insistence on 1-indexing, many of the structures in the codeplug are indexed by an address just before the real one. For example, the list of radio channel settings is an array that begins at `0x1ee00`, but the functions that access this array have code along the lines of `spiflash_read(&foo, 64*index+0x1edc0, 64)`.

One mystery that struck me when reverse engineering the codeplug was that I didn't find a missed call list or any sent or received text messages. Sure enough, the firmware shows that text messages are stored after the end of the 256K image that the radio exposes to the world.

Code that accesses the C5000 baseband chip can be reverse engineered in a similar fashion to the codeplug. The chip's datasheet[67] is very well handled by Google Translate, and plenty of dandy functions can be identified by writes to C5000 registers of similar functions.

Be careful to note that the C5000 has multiple memories on its primary SPI bus; if you're not careful, you'll confuse the registers, internal RAM, and the Vocoder buffers. Also note that a lot of registers are missing from the datasheet; please get in touch with me if you happen to know what they do.

Finally, it is crucially important to be able to sort through the DMR packet parsing and construction routines quickly. For this, I've found it handy

[67] `unzip pocorgtfo10.pdf hrc5000.pdf`

to keep paper printouts of the DMR standard, which are freely available from ETSI.[68] Link-Local addresses (LLIDs) are 24 bits wide in DMR, and you can often locate them by searching for code that masks against 0xFFFFFF.[69]

8.11 Patching for Promiscuity

While it's fun to reverse engineer code, it's all a bit pointless until we write a nifty patch. Complex patches can be introduced by hooking function calls, but let's start with some useful patches that only require changing a couple of bits. Let's enable promiscuous receive mode, so the MD380 can receive from all talk groups on a known repeater and timeslot.

In DMR, audio is sent to either a Public Talkgroup or a Private Contact. These each have a 24-bit LLID, and they are distinguished by a bit flag elsewhere in the packet. For a concrete example, 3172 is used for the Northeast Regional amateur talkgroup, while 444 is used for the Bronx TRBO talkgroup. If an unmodified MD380 is programmed for just 3172, it won't decode audio addressed to 444.

There is a function at 0x0803ec86 that takes a DMR audio header as its first parameter and decides whether to play the audio or mute it as addressed to another group or user. I found it by looking for access to the user's local address, which is held in RAM at 0x2001c65c, and the list of LLIDs for incoming listen addresses, stored at 0x2001c44c.

To enable promiscuous reception to unknown talkgroups, the following talkgroup search routine can be patched to always match on the first element of listengroup[]. This is accomplished by changing the instruction at 0x0803ee36 from 0xd1ef (JNE) to 0x46c0 (NOP).

```
for ( i = 0; i < 0x20u; ++i ){
  if ( (listengroup[i] & 0xFFFFFF)
      == dst_llid_adr ) {
    something = 16;
    recognized_llid_dst = dst_llid_adr;
    current_llid_group = var_lgroup[i+16];
    sub_803EF6C();
    dmr_squelch_thing = 9;
    if ( *(v4 + 4) & 0x80 )
      byte_2001D0C0 |= 4u;
    break;
  }
}
```

A similar JNE instruction at 0x0803ef10 can be replaced with a NOP to enable promiscuous reception of private calls. Care in real-world patches should be taken to reduce side effects, such as by forcing a match only when there's no correct match, or by skipping the missed-call logic when promiscuously receiving private calls.

8.12 DMR Scanning

After testing to ensure that my patches worked, I used Radio Reference to find a few local DMR stations and write them into a codeplug for my modified MD380. Soon enough, I was hearing the best gossip from a university's radio dispatch.[70]

Later, I managed to find a DMR network that used the private calling feature. Sure enough, my radio would ring as if I were the one being called, and my missed call list quickly grew beyond my two local friends with DMR radios.

8.13 A New Bootloader

Unfortunately, the MD380's application consumes all but the first 48K of Flash, and that 48K is consumed by the recovery bootloader. Since we neighbors have jailbroken radios with a ROM bootloader accessible, we might as well wipe the Tytera bootloader and replace it with something completely new, while keeping the application intact.

Luckily, the fine folks at Tytera have made this easy for us! The application has its own interrupt table at 0x0800C000, and the RESET handler—whose address is stored at 0x0800C004—automatically moved the interrupt table, cleans up the stack, and performs other necessary chores.

```
//Minimalist bootloader.
void main(){
  //Function pointer to the application.
  void (*appmain)();
  //The handler address is the stored in the
  //interrupt table.
  uint32_t *resethandler =
        (uint32_t*) 0x0800C004;
  //Set the function pointer to that value.
  appmain = (void (*)()) *resethandler;
  //Away we go!
  appmain();
}
```

[68] ETSI TS 102 361, Parts 1 to 4.

[69] In assembly, this looks like LSLS r0, r0, #8; LSRS r0, r0, #8.

[70] Two days of scanning presented nothing more interesting than a damaged elevator and an undergrad too drunk to remember his dorm room keys. Almost gives me some sympathy for those poor bastards who have to listen to wiretaps.

8.14 Firmware Distribution

Since this article was written, DD4CR has managed to free up 200K of the application by gutting the Chinese font. She also broke the (terrible) update encryption scheme, so patched or rewritten firmware can be packaged to work with the official updater tools from the manufacturer.

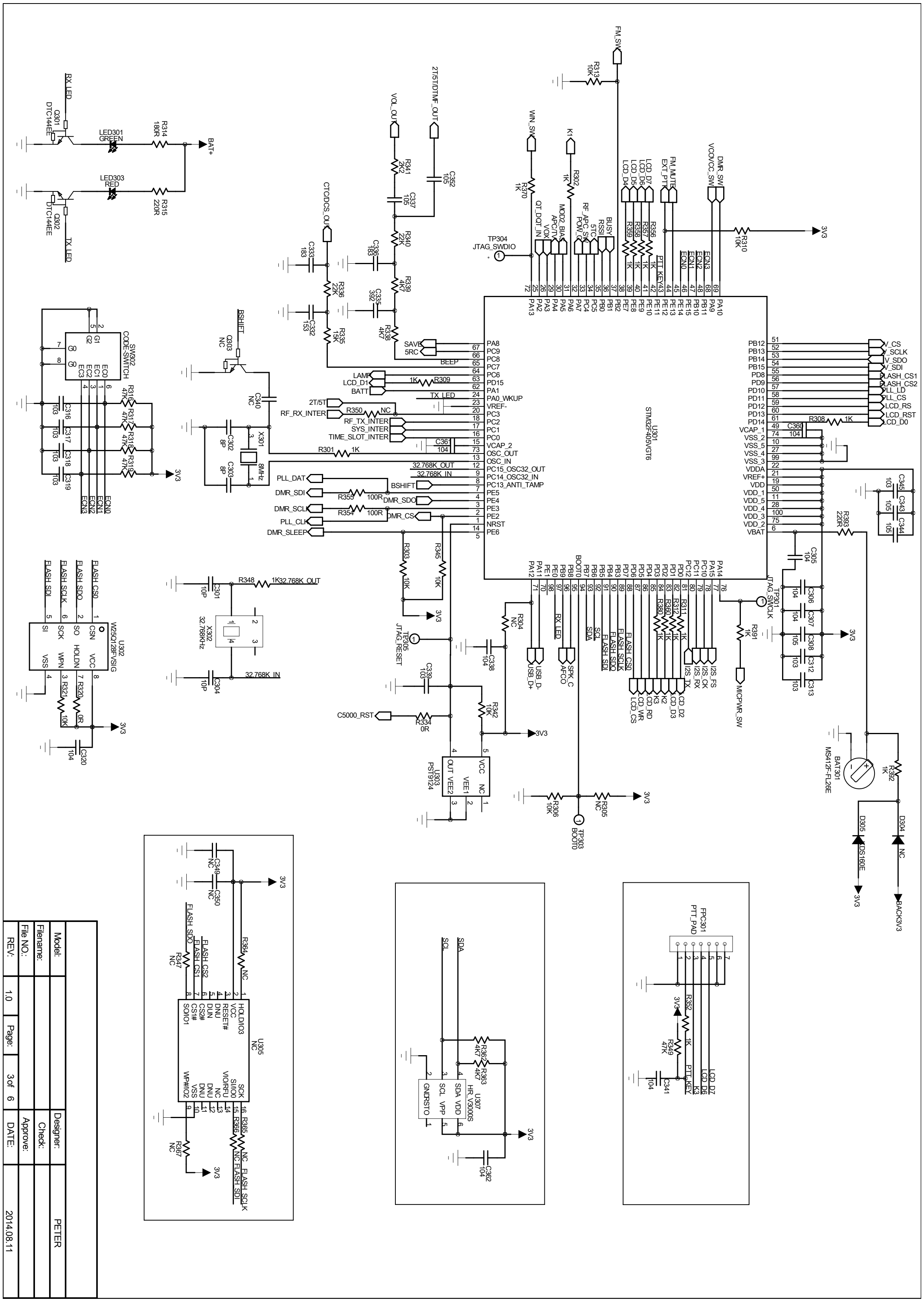
Patrick Hickey W7PCH has been playing around with from-scratch firmware for this platform, built around the FreeRTOS scheduler. His code is already linking into the memory that DD4CR freed up, and it's only a matter of time before fully-functional community firmware can be dual-booted on the MD380.

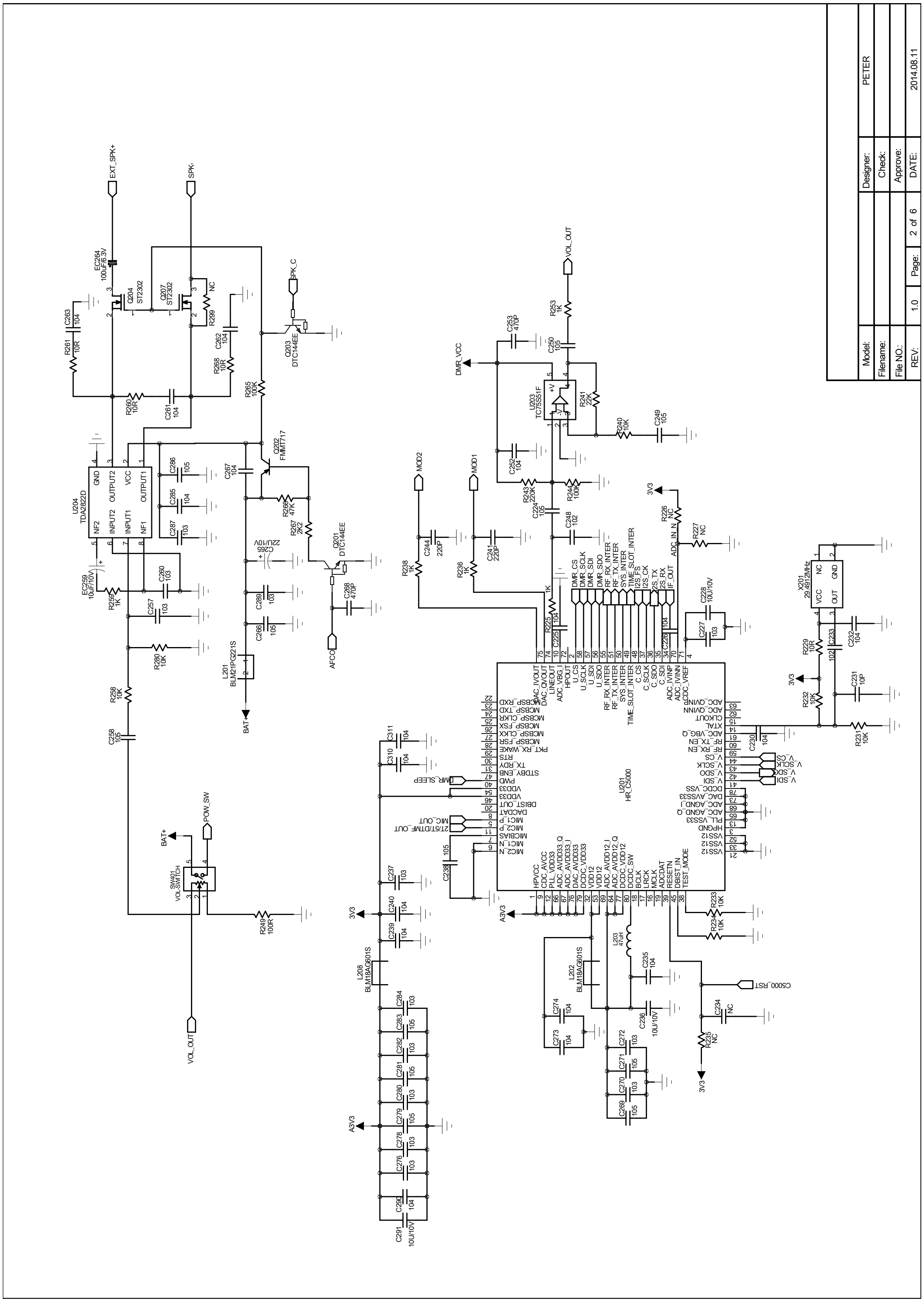
– – – –   – – –   – – – –

In this article, you have learned how to jailbreak your MD380 radio, dump a copy of its application, and begin patching that application or writing your own, new application.

Perhaps you will add support for P25, D-Star, or System Fusion. Perhaps you will write a proper scanner, to identify unknown stations at a whim. Perhaps you will make DMR adapter firmware, so that a desktop could send and receiver DMR frames in the raw over USB. If you do any of these things, please tell me about it!

Your neighbor,
Travis


U301
STM32F405VGT6
U302
W25Q128FVSIG
U303
PST9124
U305
NC
U307
HR_V3000S
SW302
CODE-SWITCH
X301
8MHz
X302
32.768KHz
BAT301
MS412F-FL26E
FPC301
PTT_PAD
LED301
GREEN
LED303
RED
Q301
DTC144EE
Q302
DTC144EE
Q303
NC
D304
NC
D305
DS160E
TP301
JTAG_SWCLK
TP303
BOOT0
TP304
JTAG_SWDIO
TP305
JTAG_RESET
RX_LED
TX_LED
BAT+
3V3
BACK3V3
FM_SW
WN_SW
K1
VOL_OUT
2T/5T/DTMF_OUT
CTCDCS_OUT
BSHIFT
DMR_SW
VCOVCC_SW
FM_MUTE
EXT_PTT
LCD_D7
LCD_D6
LCD_D5
LCD_D4
MOD2_BIAS
APC/TV
VOX
QT_DQT_IN
RF_APC_SW
STO
POW_C
BUSY
RSSI
PTT_KEY
ECN0
ECN1
ECN2
ECN3
V_CS
V_SCLK
V_SDO
V_SDI
FLASH_CS1
FLASH_CS2
PLL_LD
PLL_CS
LCD_RS
LCD_RST
LCD_D0
SAVE
5RC
BEEP
LAMP
LCD_D1
BATT
2T/5T
RF_RX_INTER
RF_TX_INTER
SYS_INTER
TIME_SLOT_INTER
32.768K_OUT
32.768K_IN
PLL_DAT
DMR_SDI
DMR_SDO
DMR_SCLK
DMR_CS
PLL_CLK
DMR_SLEEP
C5000_RST
USB_D-
USB_D+
AFCO
SPK_C
SCL
SDA
RX_LED
FLASH_CS0
FLASH_SCLK
FLASH_SDO
FLASH_SDI
I2S_FS
I2S_CK
I2S_RX
I2S_TX
LCD_D2
LCD_D3
LCD_RD
LCD_WR
LCD_CS
MICPWR_SW
PA8 PC9 PC8 PC7 PC6 PD15 PA1 PA0_WKUP VREF- PC3 PC2 PC1 PC0 VCAP_2 OSC_OUT OSC_IN PC15_OSC32_OUT PC14_OSC32_IN PC13_ANTI_TAMP PE5 PE4 PE3 PE2 NRST PE6
PB12 PB13 PB14 PB15 PD8 PD9 PD10 PD11 PD12 PD13 PD14 VCAP_1 VSS_2 VSS_5 VSS_4 VSS_3 VDDA VREF+ VDD VDD_1 VDD_5 VDD_4 VDD_3 VDD_2 VBAT
PA13 PA3 PA2 PA4 PA5 PA6 PA7 PC4 PC5 PB0 PB1 PB2 PE7 PE8 PE9 PE10 PE11 PE12 PE13 PE14 PE15 PB10 PB11 PA9 PA10
PA14 PA15 PC10 PC11 PC12 PD0 PD1 PD2 PD3 PD4 PD5 PD6 PD7 PB3 PB4 PB5 PB6 PB7 BOOT0 PB8 PB9 PE0 PE1 PA11 PA12
CS# SO WP# VSS VCC HOLD# SCK SI
HOLD/IO3 VCC RESET# DNU DNU CS2# CS1# SO/IO1 SCK SI/IO0 VIO/RFU NC DNU DNU VSS WP#/IO2
SCL SDA VDD VPP GND/STO
VCC OUT VEE1 VEE2 NC
G1 G2 G0 EC0 EC1 EC2 EC3
R301 1K
R302 1K
R303 10K
R304 NC
R305 NC
R306 10K
R308 1K
R309 1K
R310 10K
R311 1K
R313 10K
R314 180R
R315 220R
R316 47K
R317 47K
R318 47K
R319 47K
R320 0R
R321 10K
R334 0R
R335 15K
R336 22K
R338 4K7
R339 4K7
R340 22K
R341 2K2
R342 10K
R345 10K
R347 NC
R348 1K
R349 47K
R350 NC
R352 1K
R353 100R
R354 100R
R356 1K
R357 1K
R358 1K
R359 1K
R360 1K
R362 4K7
R363 4K7
R364 NC
R365 NC
R366 NC
R367 NC
R370 1K
R380 1K
R391 1K
R392 1K
R398 220R
C301 10P
C302 8P
C303 8P
C304 10P
C305 104
C306 104
C307 104
C308 105
C312 103
C313 103
C316 103
C317 103
C318 103
C319 103
C320 104
C332 153
C333 183
C335 392
C336 183
C337 105
C338 104
C339 103
C340 NC
C341 104
C345 103
C343 105
C344 105
C349 NC
C350 NC
C352 105
C360 104
C361 104
C362 104
Model:
Filename:
File NO.:
REV: 1.0
Page: 3of 6
Designer: PETER
Check:
Approve:
DATE: 2014.08.11


Model:
Filename:
File NO.:
REV: 1.0
Page: 2 of 6
Designer: PETER
Check:
Approve:
DATE: 2014.08.11
U201
HR_C5000
U204
TDA2822D
U203
TC75S51F
X201
29.4912MHz
SW401
VOL SWITCH
Q201
DTC144EE
Q202
FMMT717
Q203
DTC144EE
Q204
ST2302
Q207
ST2302
EC264
1000uF/6.3V
EC259
10uF/10V
C265
22U/10V
L201
BLM21PG221S
L202
BLM18AG601S
L208
BLM18AG601S
L203
47nH
EXT_SPK+
SPK-
SPK_C
AFCO
BAT-
BAT+
POW_SW
VOL_OUT
MOD1
MOD2
DMR_VCC
3V3
A3V3
MIC_OUT
2T/5T/DTMF_OUT
DMR_SLEEP
C5000_RST
ADC_IN_N
V_CS
V_SCLK
V_SDO
V_SDI
DMR_CS
DMR_SCLK
DMR_SDI
DMR_SDO
RF_RX_INTER
RF_TX_INTER
SYS_INTER
TIME_SLOT_INTER
I2S_FS
I2S_CK
I2S_TX
I2S_RX
IF_OUT

9 Tithe us your Alms of 0day!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

Howdy, neighbor!

One Sunday, a man and his son were hiking the Appalachian Trail, when they came upon a small church in rural New Hampshire. The boy insisted, so the father begrudgingly attended the morning service. Because he forgot to bring cash, the father fished a dime out of his pocket for the collection plate.

After the service, when they were walking back to the woods, the father started griping. "The sermon was too long," he said, "and the hymns were off key!"

After the few minutes of silence, the boy spoke up. "Dad, I think it was pretty good for a dime!"

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Do pick on quick, clever trick and explain it in a few pages. Teach me how to repair Dakarand from PoC‖GTFO 1:2 and 2:9. Show me a fancy game in a boot sector, like PoC‖GTFO 3:8. Port the worst features from Visual Basic to C, like PoC‖GTFO 8:8. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.