



# Dragon's Lair on the TI-99/4A

(8 bits should still be enough for anyone)

# What's Dragon's Lair?



# What was the impact?

- Dragon's Lair had orders for over 8500 machines in the first year worth over \$34 million (US, 1983).
- Inspired roughly 45 other LaserDisc games (not all of which were released), including Cinematronics own follow-up Space Ace and sequel Dragon's Lair: Time Warp
- Ported to nearly every 16-bit home computer that could play the graphics, and inspired games were created for the 8-bit Coleco Adam, Commodore 64, and Sinclair Spectrum. Ports are still being released today.
- By mid-1984 however, the LaserDisc fad was already fading, and Space Ace and other games were released to far less success.
- Still, it inspired a Saturday morning cartoon and countless merchandise.

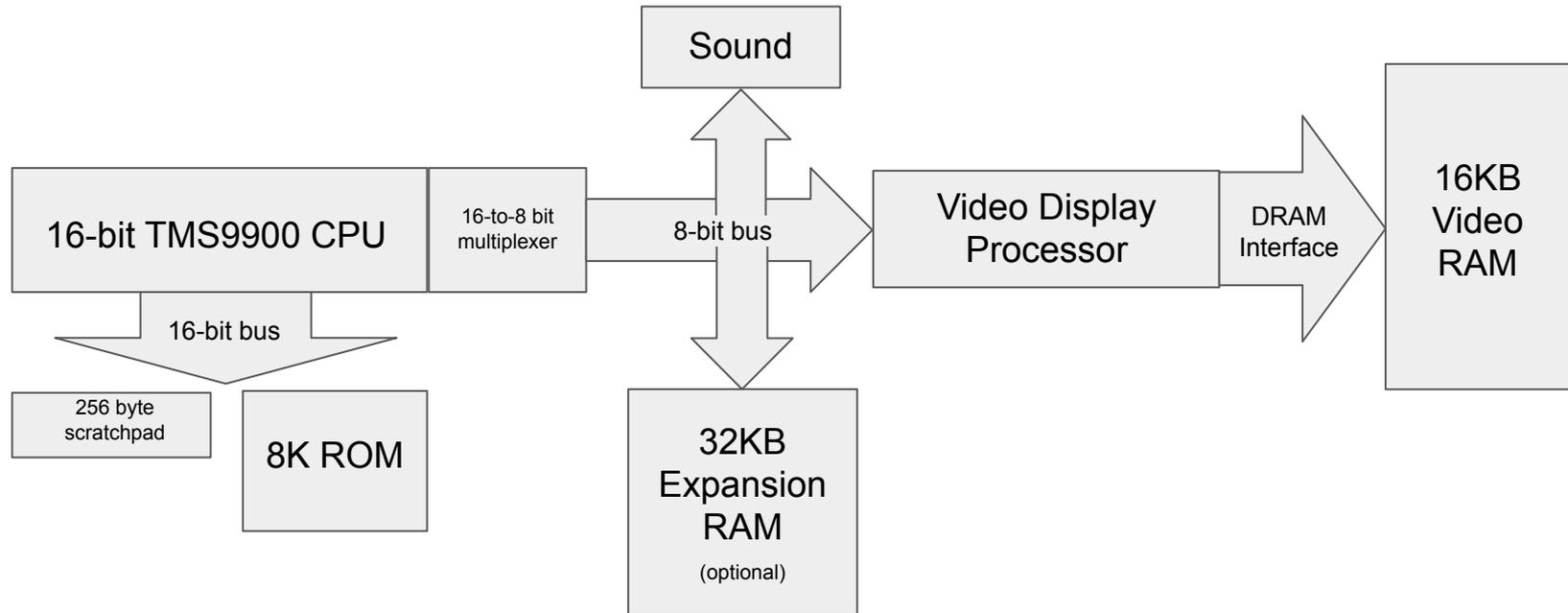
# Okay, so what's a TI-99/4A?

- Released in 1981 by Texas Instruments, this was an update of their 1979 TI-99/4
- The “home computer” attached to your television set and accepted software via cartridges or cassette (diskette came later).
- 3MHz TMS9900 (16-bit) CPU with 256x192, 15 color graphics, 256 bytes of CPU RAM & 16K of video RAM



# Wait, 8 bit or 16 bit?

Block diagram of the TI-99/4A components:

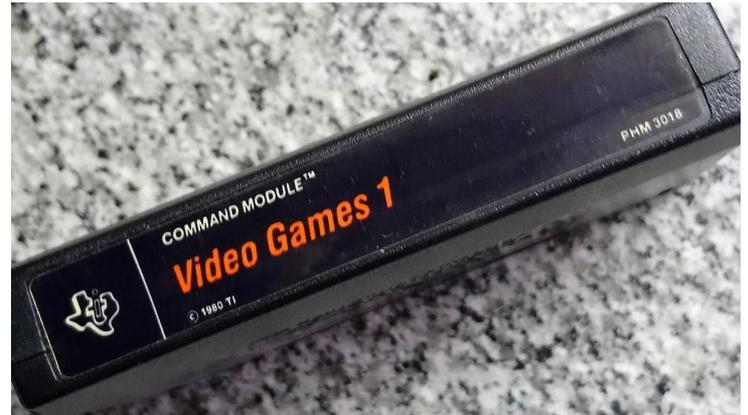
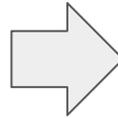


# What was its impact?

- Estimated to sell roughly 2.8 million units, although many were during the final fire sales. To compare, only roughly 100,000 expansion units were sold.
- Overall a large loss for Texas Instruments, who invested heavily in the home computer market and reportedly cancelled as many as 8 machines in development when they shut the division down.
- Most staying impact was probably the Speech Synthesizer, which, coupled with the game 'Parsec', seems to be the most common memory. Speech Synthesizers were free with six cartridge purchases, so many people had them.



So how do I get from this... to this?



# Engineering!

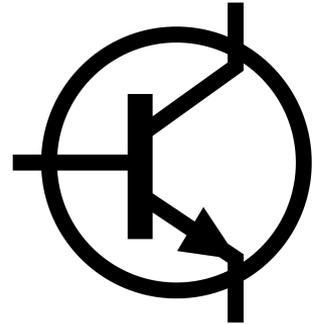
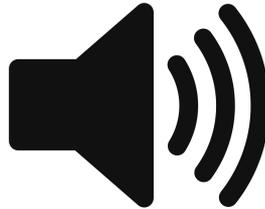
Start with the requisite back-of-the-napkin calculations:

- I tested that the TI could move about 1500 bytes per video frame (about 1/60th of a second) from the cartridge port to the video chip
- This works out to a byte every 11 microseconds, which is slow enough for the video chip to accept the data
- A video frame on the TI is 12,288 bytes, so full screen video should be able to get about 7 frames per second. I wanted 10, however, so I reduced the screen size in half for a rough estimate of 14 fps.
- I interleaved audio after every 4 bytes. This would give an audio rate of 18kHz but reduce the video rate to about 11 fps.

SOUNDS GOOD!

# Obstacles to overcome

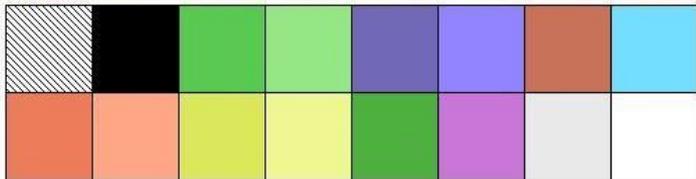
- Needed a way to convert true color images to reasonable quality 15 color images in the TI's unique bitmap graphics format.
- Needed a way to convert audio to a format consumable by the TI's tone generator sound chip.
- Needed some form of hardware to provide the A/V data to the TI's cartridge port.



# Graphics

The TI bitmapped graphics mode has a limitation or “clash” that forces every 8 pixels to be restricted to only 2 colors from a fixed 15 color palette.

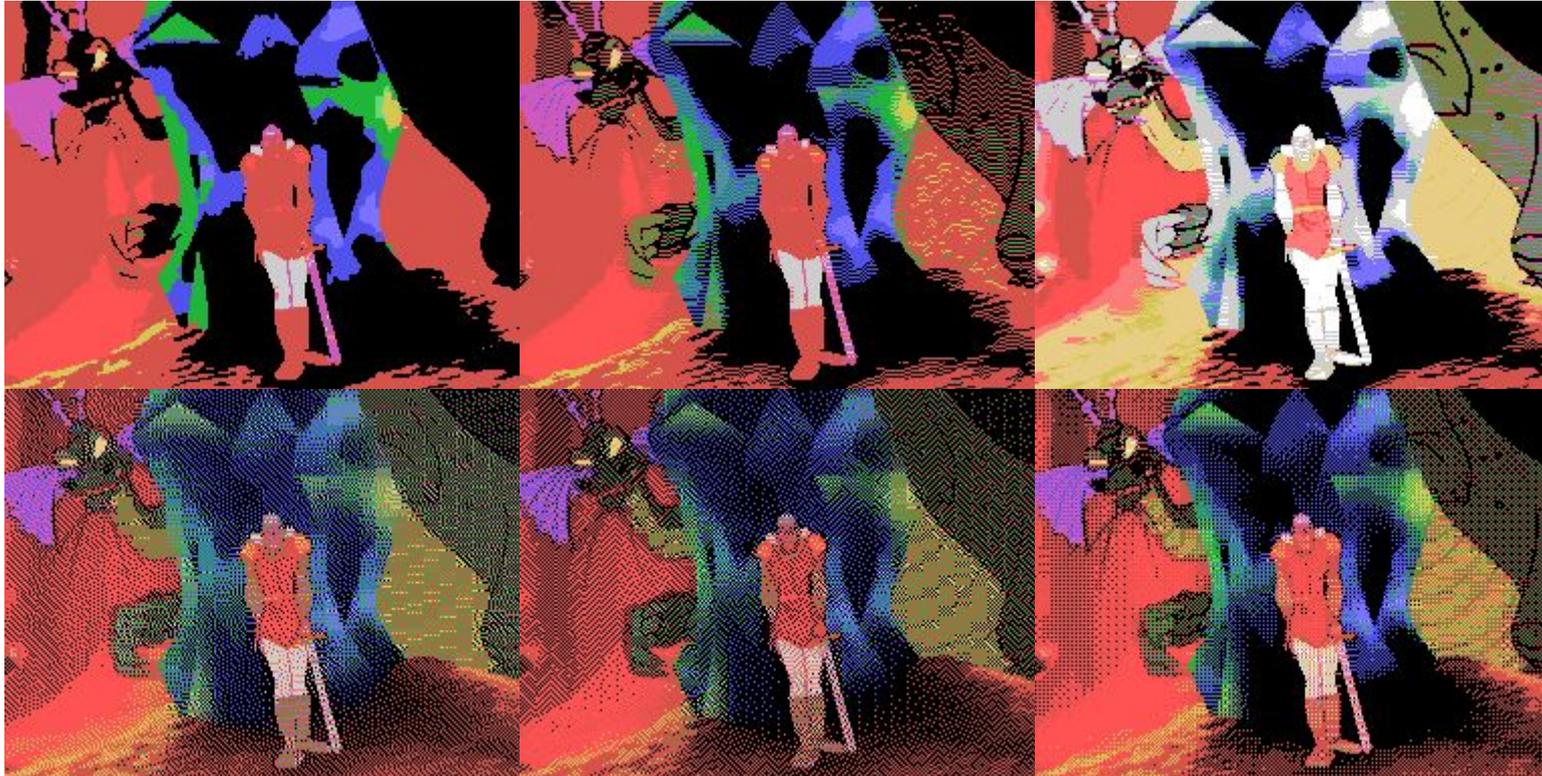
**FFBBFFBB** - (‘F’ - foreground color, ‘B’ - background color)



(the fixed 15 color palette - this is all we have!)

To deal with this, I developed a brute-force tool to search for the best matching patterns. However, coming up with pleasing dither patterns that align well with the clash is tricky.

# Evolution of the converter...



# Audio

The TI sound chip is a 4 channel sound generator with 15 discrete volume levels per channel. 3 of the channels are square wave tones, and one is a noise generator.

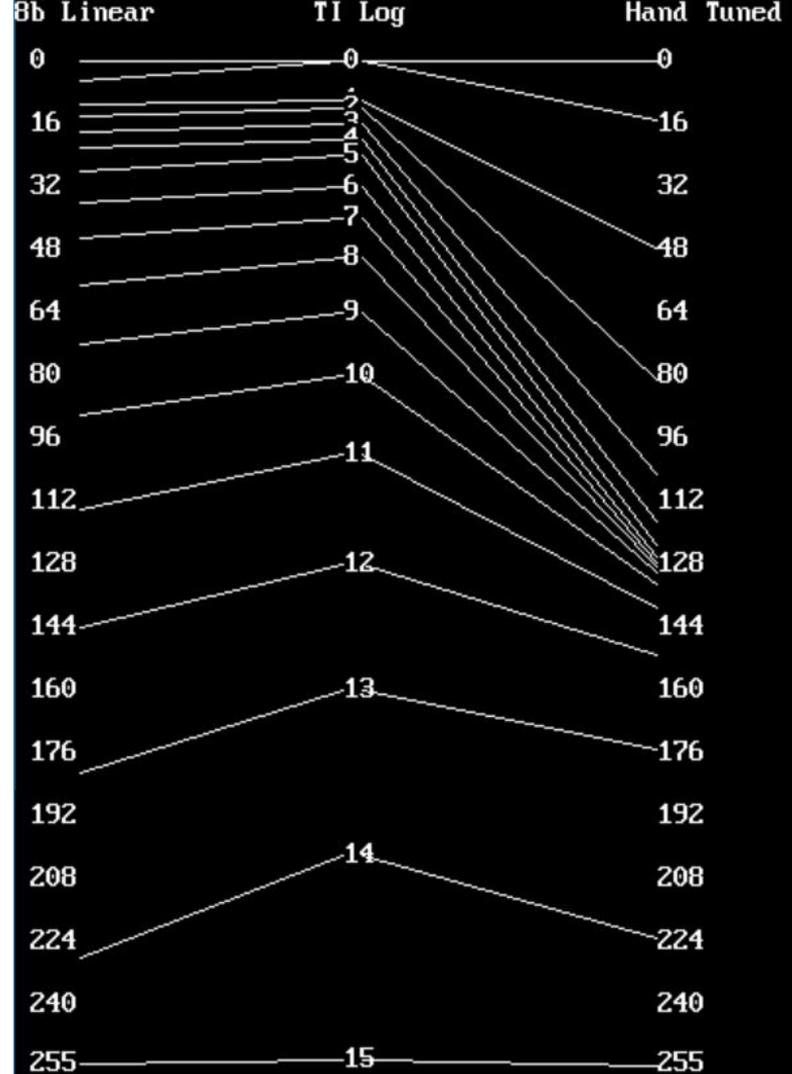
The chip can not stop the wave generator, so the only way to get samples out of it is to set to the maximum frequency (which is around 48kHz!) and modulate the volume levels.

The 4 bits of audio resolution are not evenly distributed, rather there is a logarithmic curve with most of the resolution in the quietest part of the range.

This all means NOISY. But the fact is, it's surprisingly recognizable!

# Mapping

I ended up converting to 8 bit audio on the PC, then mapping the 256 discrete levels to the 16 levels of the TI sound chip. Experimentation showed that while a closest-match mapping sounded best for very clean input audio, soft sounds would become very noisy. To make my converter work with most inputs as-is, I stretched the softer sounds to make them a little louder and added more detail in the midrange. This also helps balance the positive and negative side of the waveform.



# Testing!

At this point I had all I needed to create my first test video. I did a quick little video clip from Spaceballs as my proof of concept. This video was small enough to fit in the largest cart that already existed for the TI - 2 megabytes. This proved the concept, and let me tune the original calculations for reality, which turned out to be quite optimistic...



# Theory, meet reality!

- Video chip has set up time before you can write blocks of data (slows us down)
- Because my frame time (11fps) was slower than the video display (60fps), you could watch it paint. This meant an interleaving scheme was needed to hide this, meaning more setup time. (slows us down).
- You need set up time to select where you are reading data FROM (slows us down)
- The computer just wasn't as fast as I thought in real life (slows us down)
- And though I didn't know it till much later... writing to the sound chip is really, really slow. (The slowest operation on the motherboard - slows us down).
- Reality was about 8.6 fps, and about 13kHz audio.

# How to Paint in Real Time

The television display that a TI image is viewed on updates 60 times per second. My animation updates at less than 9 frames per second - this means that every frame of animation takes 6 video frames to finish drawing.

This is just a fact of life, so we have to live with it. But there's a wrinkle - the TI graphics consists of separate pattern and color tables, and we have to update each of them in turn. That would mean three frames drawing the pattern, and three frames drawing the color.

That means 1/3rd of a second during which one table is current and one table is out of date. That's lots of time to be able to see it as corruption.

# Interleaving fools the eye...

In order to make it less obvious that there are two display tables, we split the screen into 4 regions. Then we can update the color and pattern table for one region before moving on to the next region.



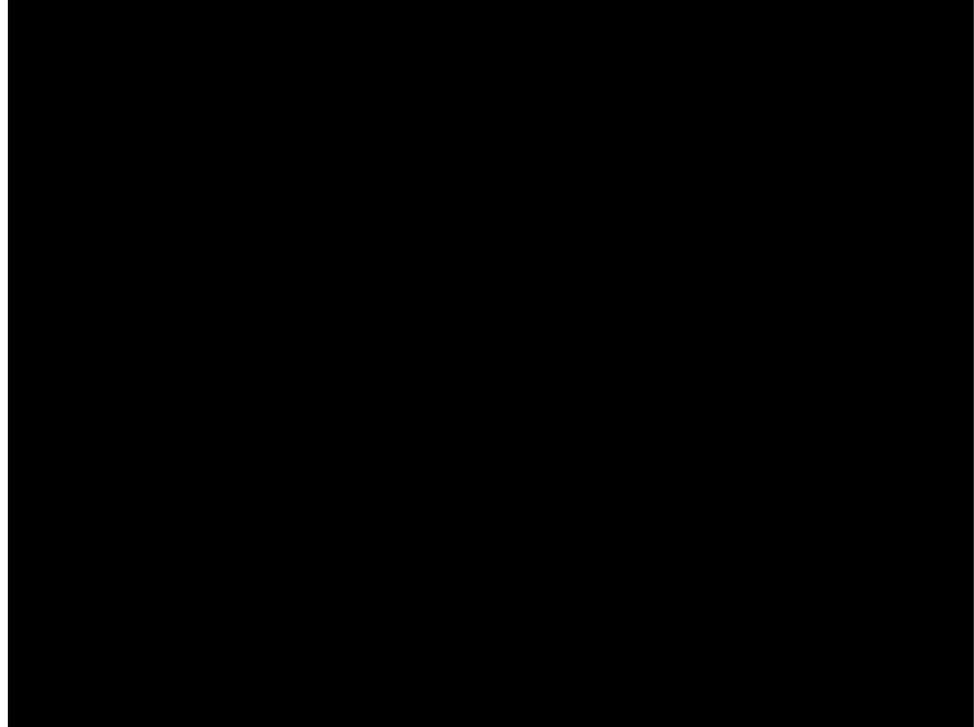
# GPU To the Rescue!

While the interleaving is notably better, there is a tradeoff. Every time you change the video memory address you want to write to, you need to set up the address registers again. This means that the more slices you have, the more time is spent just changing the address. This digs into your frame rate, since no matter what they are for, you can only write so many bytes per frame.

The F18A is a video enhancement by Matthew Hagerty of [codehackcreate.com](http://codehackcreate.com). While its major claim to fame is the VGA output, the hidden gem of this device is the embedded GPU with an extra 2k of RAM. The GPU allows direct access to video memory, meaning that writing to the video and color table individually is essentially free.

# GPU Block Writing

This video demonstrates the difference between the standard video update and the F18A GPU update in slow motion. On the left are the two video chips, with pattern data on the top and color data on the bottom. As they are updated one byte at a time, real time samples of the display appear on the right side. The GPU's rapid update of both tables at the same time means that 'sparkles' caused by mismatched color and pattern data are almost non-existent.



# But How Does it Sound?

To generate sound in sync with the video, the audio is interleaved with the video. Every fifth byte of data is transferred to the audio chip directly. The transfer rate of the hardware dictates the playback rate.

One issue is that the write to the sound chip is 3-5 times slower than the write to the video chip, which means that after every four bytes there is a small delay. However, seen in groups of 5 bytes, the overall output is evenly spaced. In Dragon's Lair, this happens roughly 13,000 times per second, for a 13khz audio playback rate.

# Now I needed hardware...

The TI's cartridge space is 8 kilobytes. My estimates for Dragon's Lair showed that it was going to need about 80 **megabytes**. There was not much available that could do it.

I spent a lot of time with paper and datasheets to see whether SD cards or Compact Flash could manage it. They easily had the needed throughput, but setup times were too long, and the TI did not have any storage space for caching data (even if the video player could handle the additional overhead).

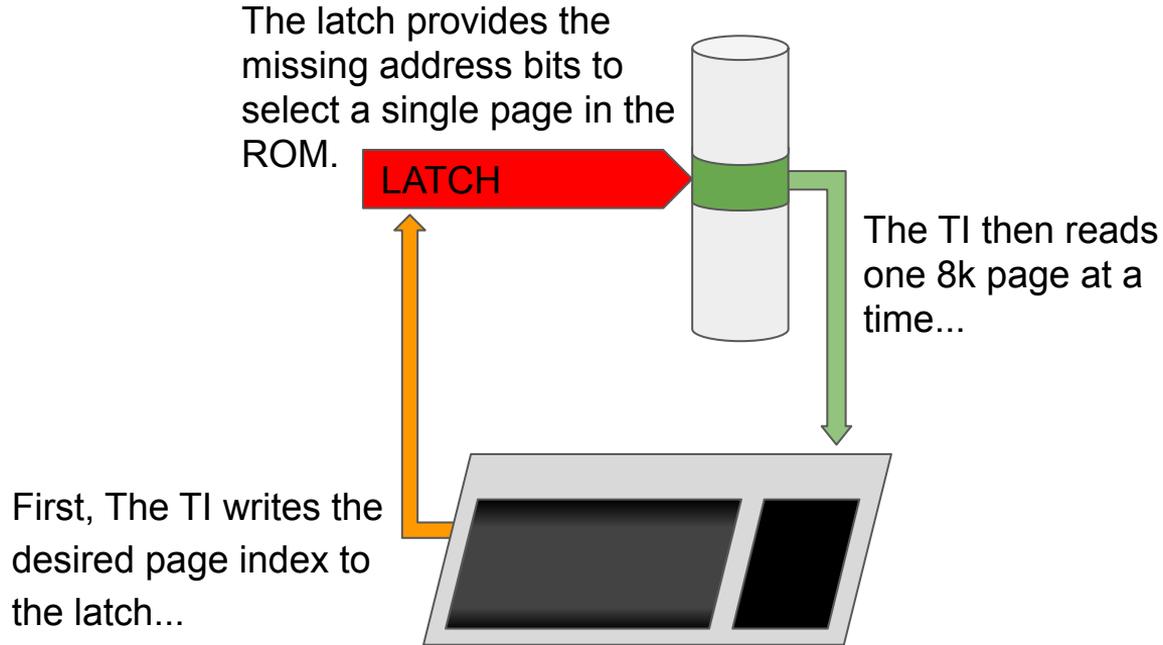
I could conceivably have lost the setup times in the beginning of the first frame of a sequence, but I decided to fall back on a simpler scheme...

# Hardware Selection

While in Shanghai, I came across a 128MB parallel flash chip. Parallel flash is becoming less common as more and more systems use high speed serial memory, but it's ideal for interfacing to older systems. However, to interface 128MB to the TI was going to take a lot of connections to a larger latch than was ever used before - in fact 14 bits would need to be added to the address bus. (We need 27 bits, the TI itself only presents 13!)

This large latch would allow us to *page* the large ROM into memory, a bite-sized 8k at a time. Fan-made hardware had already taken the paging up to 2MB, so it was just an extension of the existing scheme.

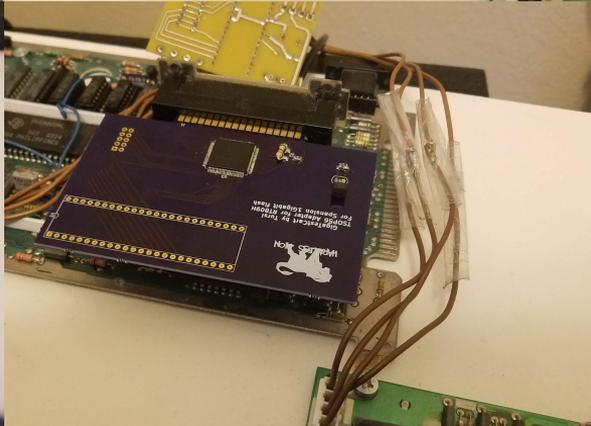
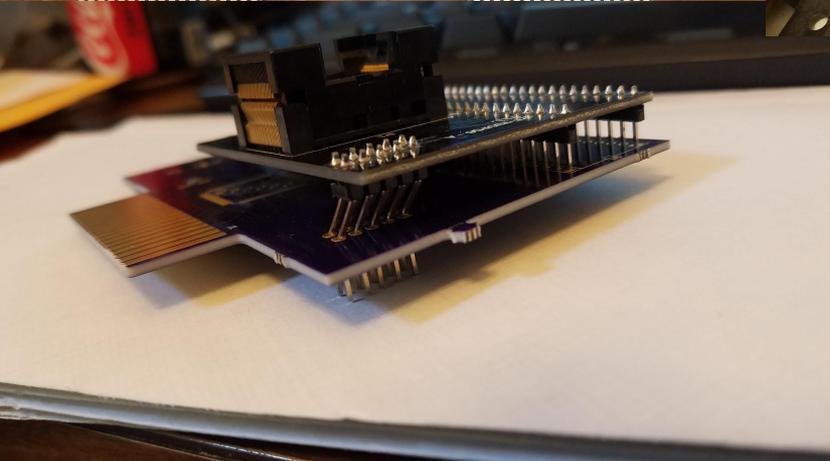
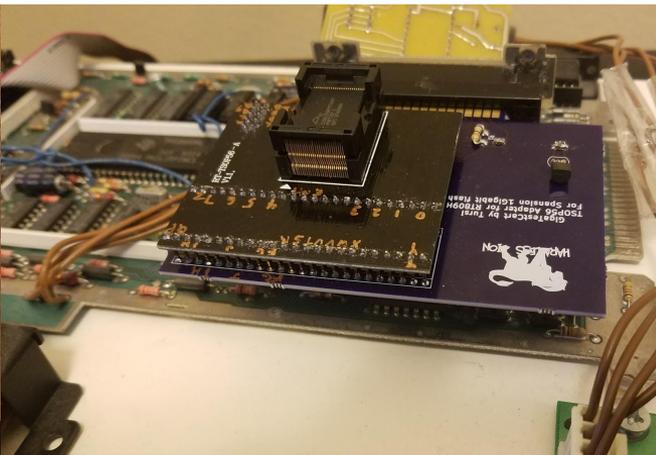
# How to flip pages...



In this case, there are 16,384 pages to get 128MB of ROM.



# Prototypes...



# CPLDs and VHDL

To get both the relatively large latching logic as well as voltage conversion between the old TI's 5 volts and the modern memory chip's 3.3 volts, I selected a CPLD. At \$3, it was cheaper than the numerous discrete chips that it would otherwise take.

I had to learn some basic VHDL to do this, which is one of several languages used to describe circuits to the computer, which can then be implemented on CPLDs, FPGAs, and sometimes other devices.

Once the basic latch was working, I was also able to add emulation of TI's proprietary boot chips, which makes my cart work on later hardware, all within the CPLD.

# Sample of the VHDL

```
-- handle the ROM latch on write
PROCESS (ALL)
BEGIN
    -- capture on rising edge of WE (if ROM is active)
    IF (rising_edge(ti_we)) THEN
        IF (ti_rom='0') THEN
            -- we dont capture the TI lsb because it ALWAYS
            -- changes due to the 16->8 bit multiplexer
            -- remember TI bit order - 0 is MSB
            latch(11) <= ti_adr(3); -- MSB
            latch(10) <= ti_adr(4);
            latch(9) <= ti_adr(5);
            latch(8) <= ti_adr(6);
            latch(7) <= ti_adr(7);
            latch(6) <= ti_adr(8);
            latch(5) <= ti_adr(9);
            latch(4) <= ti_adr(10);
            latch(3) <= ti_adr(11);
            latch(2) <= ti_adr(12);
            latch(1) <= ti_adr(13);
            latch(0) <= ti_adr(14); -- LSB

            -- two extra bits come from the data bus
            latch(13) <= ti_data(6); -- MSB
            latch(12) <= ti_data(7); -- LSB

            -- two bits of chip select (for 512MB mode over 4 chips)
            chip(1) <= ti_data(4); -- MSB
            chip(0) <= ti_data(5); -- LSB
        END IF;
    END IF;
END PROCESS;
```

## Booting on QI 2.2

Just before the end, TI revised their console with a “version 2.2” boot that locked out ROM-only cartridges (commonly attributed as a direct attempt to lock out AtariSoft games). The only way to boot was to implement TI’s proprietary “GROM” memories, which contained a multiplexed data/address bus and a built-in address counter with integrated select and automatic increment. The cost of implementing this in discrete logic was too much for most third party developers, who just abandoned the machine.

So, of course, I was challenged to implement this as well, so that this new cartridge “could have worked” back then, giving a bit of serendipity to the people who grew up with one of those limited consoles.

# GROM Memory

GROMs had a built-in 13-bit address bus along with 3 bits of hard-wired address match. This let them respond on 8k divisions (although official GROMs had only 6k of actual memory with the remaining 2k wasted). Through an 8-bit bus and 2 control pins, the address could be read or written, or data could be read. These memories were also slow, clocked by a 400khz clock, and so had a busy signal that could halt the CPU, and all pins were attached to a shared bus that all GROMs responded to.

I had previously implemented an emulation using an Atmel AVR. So of course I said it would be easy.

# Scaling it down...

My AVR implementation provided 128k of GROM memory in a single package, but the slow execution of a software implementation made it much more tolerant of the noisy TI memory bus. The exact concepts used for software did not work as well under a CPLD.

To deal with the noisy bus, I needed to implement delays after detecting logic level switches (to give the bus time to become stable). This took logic resources from the CPLD, which was already tight for resources.

Next I needed a memory latch, to hold the internal address bus. And here I already ran into a problem.

# Latching the GROM Address

The 13 bit address bus of the GROM took too many resources in the CPLD - it simply didn't fit. I did experiment with trying to share the ROM latch, but it was getting tricky and reduced the utility of the system (switching between memory types would then require special code to restore the correct memory bank).

Ultimately, I did a number of experiments. I finally settled on an 8-bit address latch - this gave me 256 bytes of addressable GROM. To deal with the 2-byte sequence of setting the address (so I know whether MY address is being accessed), I captured a single "set/not set" bit from the outgoing bytes (which would otherwise become the MSB of the address word). This gave me the GROM emulation, in a limited way, with just 9 bits of data stored.

# Reality Steps In Again

Now that I could finally move everything to physical hardware - I ran into a small challenge... the hardware didn't work.

I wrote diagnostic software, and found, bizarrely, that everything seemed to be working, except that the flash chip would randomly return empty data for entire 128KB blocks of data (far larger than the TI's 8KB window!) This correlated with the size of an erase sector, but other than that, I had no clues. Furthermore, the bad sectors moved with every power cycle, and all other data was 100% correct.

Read your datasheets.

# Finally, the software!?

With all the technology pieces in place, I still had to create a game.

I first needed to convert all the video files, which I extracted from the HD release for PC (they were conveniently in WMV). FFMPEG, SOX, my own image converter and a handful of custom tools provided me with the TI video data.

I then needed to go through these images, frame by frame, and note every frame index that marked a notable action point. In particular, begin and end of each clip, and begin and end of each point input became valid.

```
0176      aldrawbridge.bin
0176-018A Resurrection
018B-01BF Castle intro

01C0-01DA Approach drawbridge (drawbridge is technically a lost scene)
01DB-01DF Fall through
01E0-01E9 Monsters approach
01EA-01EF Approaching - press sword
01F0-01F7 Swing sword
01F8-01FF Move up/right needed
0200-0219 Climb up and run in

021A-0231 Doors slamming shut

0232-023E Enter vestibule
023F-0244 Crumbling
0245-0255 Stumbling (down/right needed, UL die)
0255-025A Collapsing (Move right needed, UL die)
025B-026C Escaping

026D-028B Game over

028C-0299 Caught by drawbridge monster

029A-02AB Falling into pit
```

# What about porting?

Although Dragon's Lair is well documented, two things made a direct port less helpful:

- None of the disassembled listings I could find covered the scene data
- The laserdisc frames would not line up with my converted frames anyway

Thus, the inputs and timing would need to be redetermined. Fortunately for me, the people at dragons-lair-project.com have detailed breakdowns of all the scenes, sequencing, inputs, and scoring. While I made some interpretations, having this reference made the job **possible**.

*Also, porting is not any fun compared to writing it yourself...*

# Making it interactive

I had a video player, so the first step was to get it reading the joystick.

To read the joystick on the TI hardware, you need to use a special I/O hardware system called the 'CRU'. There are just two steps to get input:

- Set the output CRU bits to select the joystick
- Read the input CRU bits

Fortunately there was enough time in delays between the screen output blocks to read the joystick in one game frame, check for new inputs, and respond if they match an input mask (for 'valid' directions). (I also had to switch to keyboard and check for the soft reset sequence, which is a similar task.)

# Scripting the game

In order to keep the complexity of managing hundreds of little clips within dozens of scenes, each with six possible outcomes, I created a simple engine that handled the game as a set of scripts, with one script for each scene. A script is simply a list of clips to play, with a link to to next clip based on input.

```
** start frame, frame count (byte), death? (byte), valid joystick (byte), good joystick (byte),  
** hint sprites, up scene, down scene, left scene, right scene, fire scene, timeout scene, score
```

```
SCENEDATA
```

```
DATA >0176,21 * resurrection  
DATA >026D,31 * game over
```

```
** scenes **
```

```
A01EA DATA >01C0,>2A00,>0000,HINTNONE,0,0,0,0,0,A01EA,49 * drawbridge through monsters approach  
A01F0 DATA >01EA,>0600,>0101,HINTFIRE,0,0,0,0,0,A01F0,A028C,0 * approaching, press sword  
A01F8 DATA >01F0,>0800,>0000,HINTNONE,0,0,0,0,0,A01F8,49 * swing sword  
A0200 DATA >01F8,>0800,>1414,HINTUP,A0200,0,0,A0200,0,A028C,0 * move up/right needed  
A021A DATA >0200,>1A00,>0000,HINTNONE,0,0,0,0,0,A021A,49 * climb up and run in  
DATA >021A,>1800,>0000,HINTNONE,0,0,0,0,0,0,0 * slamming doors
```

```
** deaths **
```

```
A028C DATA >028C,>0Eff,>0000,HINTNONE,0,0,0,0,0,0,0 * caught by drawbridge monster
```

# Game Engine

The game engine itself has two main paths, which select different sequences of scenes (depending on whether we are representing the arcade version, or the optimized home version). Scenes are cued up into a simple list, which is walked through by the scene selection code until all lives are exhausted, or the list is completed. In most cases, a death causes the scene to be added to the end of the list, to be completed before the player is permitted into the Dragon's Lair.

The only real challenge here was memory management. The TI-99/4A, unexpanded, has only 256 bytes of CPU RAM (and 16KB of video RAM). Up until this point, all game data had been maintained in CPU RAM. The lists, however, had to be stored in video RAM.

# Diagnostics

With the issues I'd had with the hardware, and with the need to solder by hand, I needed a way to confirm that the cartridge was working 100%.

To this end, I added a small checksum to each page of the cartridge, and built code to verify it. The original arcade game used photographed sheets of paper for the progress screens, so I made similar screens for my own, and integrated CPU, sound and control tests just to be similar to the arcade game.



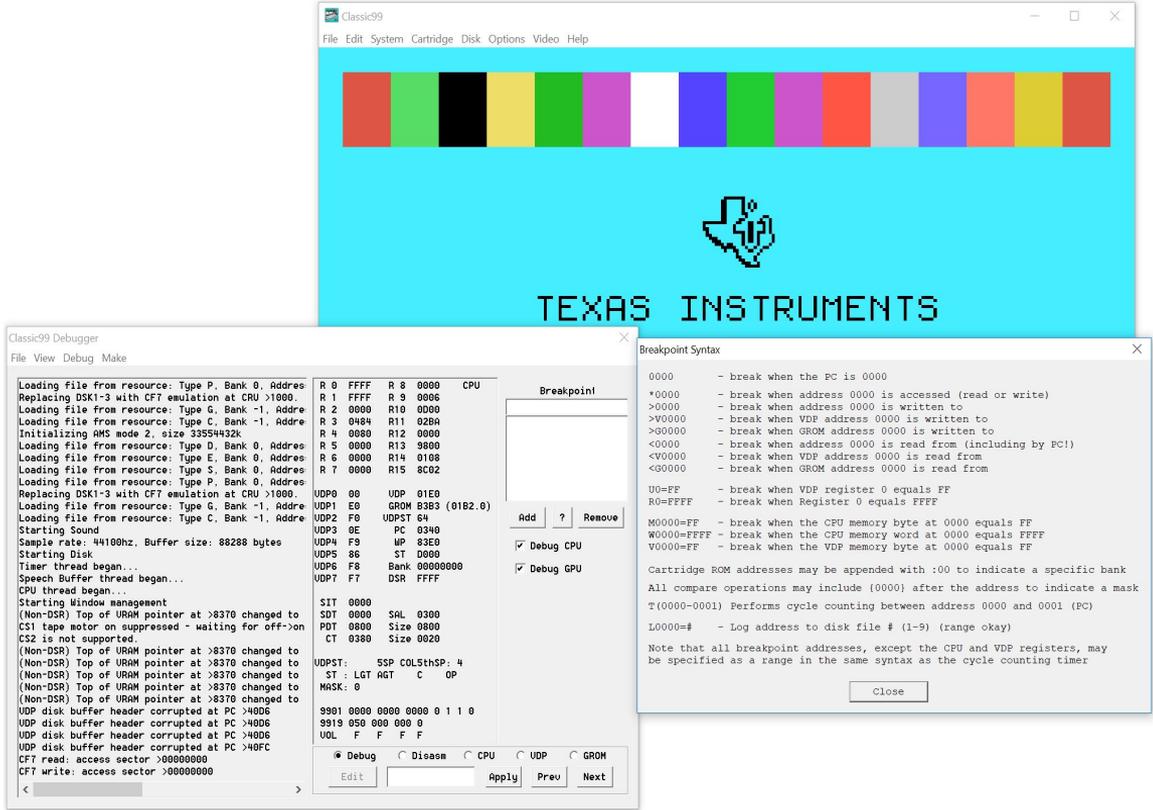
# Debugging

Although the code itself was not very complicated, there was a lot of it, spread out across a large system. More importantly, there was a lot of data. Keeping the code in manageable pieces, and using scripts to keep the build organized - these help but only take you so far. Eventually, you have bugs and need to solve them.

In the old days, you were stuck with changing the screen color, text on screen, beeps from the speaker, maybe even LEDs if you were clever enough. This could help you infer what was actually going wrong in most cases, with enough iterations...

# Emulation

Today emulation is advanced enough to make a lot of this easier. Classic99 performs checks for common illegal operations, allows insight of the system memory and registers (permitting modification or logging), plus a large array of breakpoint options - including hardware breakpoints that would be difficult or impossible on real hardware.



# Support the masses...

With the game largely working, the last task was to add keyboard mode. This concerned me, because the playback code was carefully cycle counted.

While the joystick required setting a single CRU select, and reading once for all the bits, the layout of the keyboard required three sets of this to get the same result (at least with keys that are useful to the average user).

Ultimately, I gained back a few cycles by removing the edge detection code. Rather than looking for new keys, I allowed the keyboard version to just check the currently held key at any given point. It was still a few hundred microseconds slower per frame, but this turned out to be neither visible nor audible.

# ROM layout

Since I had lots of space available on the cartridge, I didn't need to be stingy with my cartridge layout. To keep things simple, the three main programs (keyboard, joystick, and diagnostics) all have their own 8k page.

In addition, every scene table, and a little bit of support code to queue it up, also gets its own 8k page. This means that a scene can be selected by simply selecting a ROM page, and jumping to a fixed address.

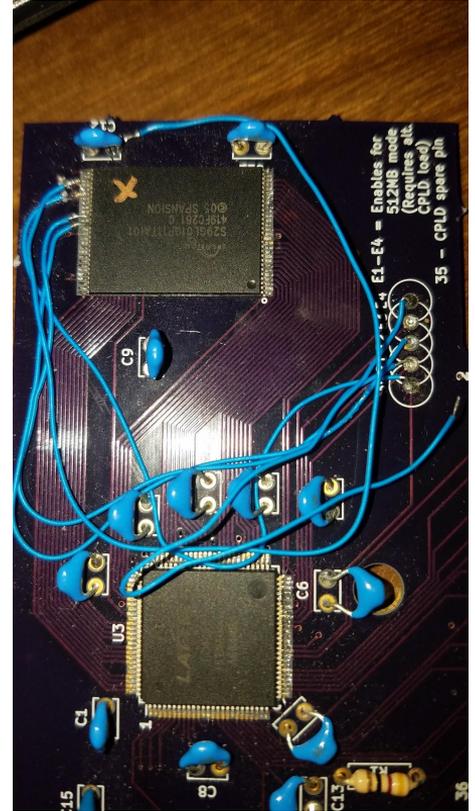
Most of the pages have lots of space free.



## Version 2

After I started shipping, I found that a large percentage of the flash chips I had simply would not program in my EPROM programmer. Rather than come up very short, I decided to get a quick-turnaround set of PCBs from a US company. I was able to find a modern version of the flash chip, and layout a new PCB that I hoped would allow me to program the flash chip in-circuit, allowing it to be fully manufactured.

Of course, while I was waiting, I needed to make it possible to program and test the carts...



# How to program in-circuit...

To program the flash in-circuit, I needed to solve three issues:

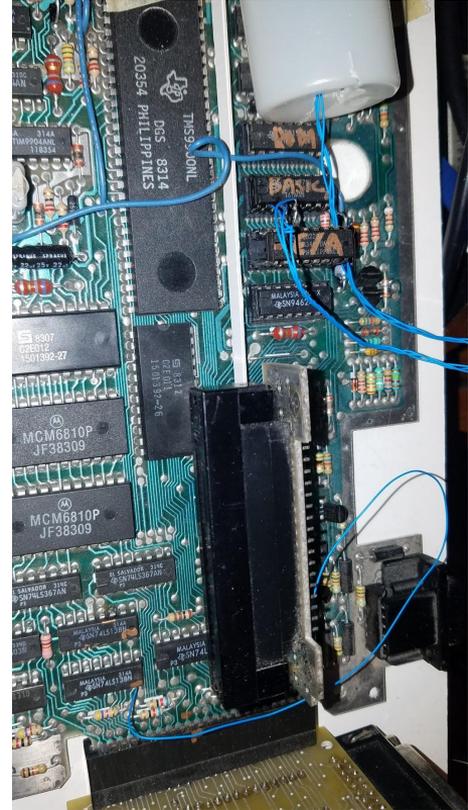
- How do I interface with the card edge connector?
- How do I get 80MB of data into that card edge connector?
- How do I get the data from the card edge connector to the flash chip?



# Interfacing with the card edge connector

Although I considered numerous options, including feeding the data through the JTAG port under the CPLD and building an interface for the PC, ultimately I decided that the best interface for the cartridge's card edge connector would be the TI itself.

To make this useful, I would need to figure out how to load software when the cart port was occupied. The ultimate solution was to disassemble an Editor/Assembler cartridge, commonly used to load assembly code, and connect it directly to the motherboard...



# Getting 80MB into the Cart

The TI is a rather small memory system by today's specs, but I did have one large storage device - a compact flash adapter. However, it was designed to throw away half of the 16-bit data, and to emulate only three 90k floppy disks.

After tracing through its BIOS with the debugger in my emulator, I was able to get the information I needed to talk directly to the card. I decided to try and get direct 8-bit access in order to make writing the data from the PC simpler. After some experimentation, I was able to get this to work, although I found that some cards did not work in 8-bit mode correctly. But, I only needed one that did.



# Getting the data to the flash chip

This was the largest challenge. In order to program a flash chip, precise sequences of writes to specific addresses need to be performed. The TI hardware meant that every 16-bit access was broken into two 8-bit accesses (with no ability to perform only one or the other).

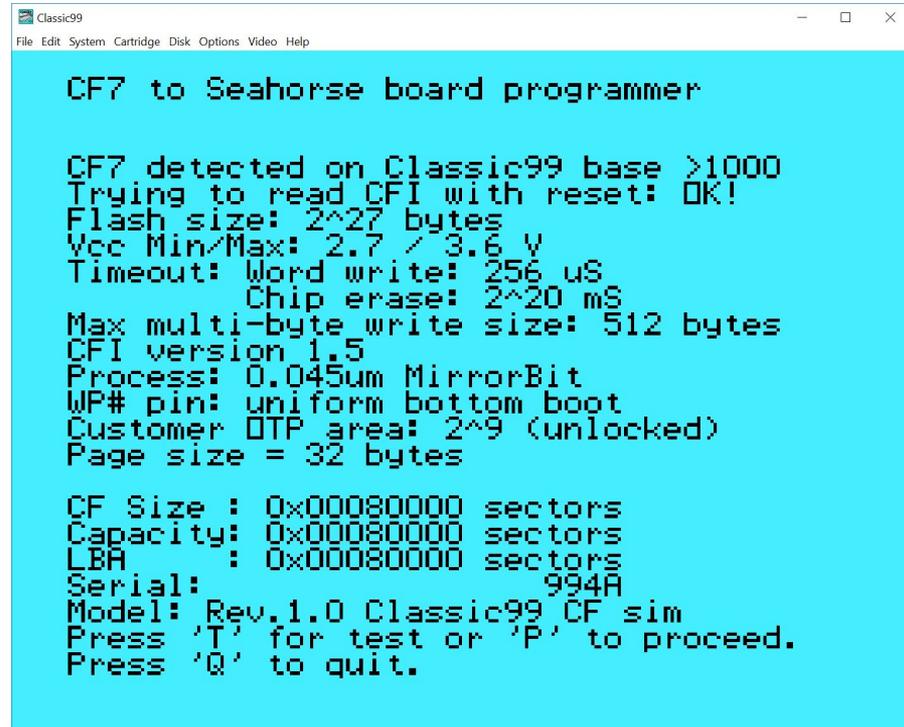
Furthermore, I still needed to control my 14-bit latch so that I could access the entire chip rather than only 8k of it.

I cheated and modified my console hardware. Using the GROM select line, I repurposed it as a second memory select line, at a different address. I then created a new CPLD load that would relay writes to the new address directly to the flash chip, while writes to the old address would manipulate the latch as before.

In order to control whether MSB, LSB, or both would be sent to the flash chip, I defined a couple of extra latch bits. Since I could remove GROM emulation for this programming load (and put it in later for runtime), there was space for it.

# Finally, the software again

Finally, I just needed to put all the pieces together. I updated my emulator with (sketchy) emulation of the compact flash card and the new flash chip itself, so that I could build and test the software before I had the hardware in hand. It took a few passes, but this helped a great deal, because when I got the hardware... there was a problem.



```
Classic99
File Edit System Cartridge Disk Options Video Help

CF7 to Seahorse board programmer

CF7 detected on Classic99 base >1000
Trying to read CFI with reset: OK!
Flash size: 2^27 bytes
Vcc Min/Max: 2.7 / 3.6 V
Timeout: Word write: 256 uS
          Chip erase: 2^20 mS
Max multi-byte write size: 512 bytes
CFI version 1.5
Process: 0.045um MirrorBit
WP# pin: uniform bottom boot
Customer OTP area: 2^9 (unlocked)
Page size = 32 bytes

CF Size : 0x00080000 sectors
Capacity: 0x00080000 sectors
LBA      : 0x00080000 sectors
Serial:   994A
Model: Rev.1.0 Classic99 CF sim
Press 'T' for test or 'P' to proceed.
Press 'Q' to quit.
```

# Hello, reality...

Upon arrival, the new cartridges didn't work. Using a direct debugger, I was not able to edit any of the flash bytes, even though everything else appeared to work.

Eventually, I found that I had tied the write protect pin to active on the old board. It didn't matter on the old board, since I programmed the flash chip externally, but the new chips had the boot block at the beginning of the flash, and so the first 128KB was write protected.

Better still, the necessary pin was routed under the flash chip, so could not be easily cut. All I would have needed to do was cut the trace, but it was inaccessible.

*Label your schematic components.*

# Making room...

There was only one option - move the code to ignore the first 128KB. There was plenty of space in the ROM, but I needed to change all the page index references.

*(Use named values, not magic numbers. I had most offsets as equates but would have loved it if they ALL were...)*

Fortunately, the GROM boot code was located at the top of the GROM, so required no changes. But it could have easily been moved with a change to the CPLD. Ultimately, and with the help of emulation, everything was working.

Of course, the first cartridge took over 4 hours to program. This was a bit more than I expected, and with over 100 carts to go, it felt like a bit too long...

# One more round of optimizations...

The programming code contained three important loops:

- Read from the CF card into a memory buffer
- Write the flash chip from the memory buffer
- Verify the flash against the memory buffer

The TI's 256 bytes of CPU RAM, called the scratchpad, would today probably be called a cache. Like most cache, it's faster than the rest of the memory in the system, being fully 16 bit (instead of 8 like the rest of the system). Moving the loops into the scratchpad RAM brought the programming time down to 90 minutes.

# And the end...

During the whole of the version 2 phase, I was still programming, building, and testing original cartridges in order to get shipments out to people. In the end, the last shipments went out just about a week after my license expired (I had checked with Digital Leisure in advance if it'd be okay to finish my orders). Given the large number of unexpected challenges, I'm happy enough with this outcome.

Although I didn't plan for the round number, in the end 150 modules were produced - 50 more than my initial estimate. 125 were boxed (mostly because I only had that many extra posters for the box contents).

I'm also quite happy to be done. At least, with this project. :)

# Tools and software used

Classic99 (Emulation), Visual Studio (C++ for tools), GCC (C for other tools), SOX (for audio conversion), ffmpeg (for video conversion and extraction, ImageMagick (for image resizing), Convert9918 (for image conversion), dircmd (for batching), KiCAD (for schematic capture and PCB layout), xdt99 (for assembler), Photoshop (for image manipulation), Krita (for more image manipulation), cmd (for batching), ispLEVER (for CPLD development, program and test), Rigol (oscilloscope), Aoyue (rework station), Hakko (soldering iron), AmScope (microscope), Extech (bench power), TI-99/4A (for final test and programming), F18A (for testing GPU code), MiniPEB (for compact flash interface), OSHPark (for prototypes), PCBExpress (for prototypes), pcbcart (for initial PCBs), PCB4U (final PCBs), Digikey (components), Mouser (components), Vistaprint (posters and postcards), Packola (boxes), StickerGiant (cartridge labels), Spansion (flash), Lattice (CPLD).