20:04 Turtles All the Way Down

by Charles Mangin

Emulating an Apple II is a relatively straightforward proposition. The architecture is well-documented; the chips and logic are all well understood. It's a solved problem. All that remains is the choice of implementation.

The Apple II family of computers has been virtualized many times over, recreated in forms as varied as Javascript and Minecraft redstone logic. You can even tinker with Print Shop on your smartphone or play Wavy Navy in a web browser.

The program emulating the Apple may even be running inside a virtual machine of its own - a Parallels VM running Windows running AppleWin, itself hosted on a Mac running macOS, all to play an Apple II game. How far you can go along this chain is only limited by your imagination and available hardware. That whole macOS installation may be running in VirtualBox on a Linux host.

But can we go deeper?

Turns out, yes. Yes we can. In this PoC, I set out to add another layer or two to the this emulation lasagna by emulating an Altair 8800 on the Apple II.

The original S-100 machine, the Altair, boasts toggle switches, blinking LEDs, and not much more beyond that. Inside its industrial steel chassis lurks an Intel 8080 processor churning through bytecode at two MHz. With an addressable space of 64 kilobytes of memory, the 8080 contains seven eight-bit registers, a relocatable stack, and can access up to 256 I/O devices.

That seems easy enough to emulate on modern hardware, right? Compare those stats to the 6502 in the Apple II, however. The 6502 is also an eight-bit processor with 64k addressable memory, only three registers, a fixed 256-byte stack at `0x0100` and memory-mapped I/O.

September, October Super Special
Apple II 16K
$950.00 reg 1195 00

INTEGRAL DATA SYSTEMS

440G:	Paper Tiger with Graphics; 2K Buffer	**$950** reg. $1095
460:	Word Processing Quality	**$1099** reg. 1295
460G:	IDS 460 w/Graphics	**$1199** reg. 1395

Centronics 737 High Quality Dot Matrix	**$895** reg. 995.00
Apple Silentype Includes interface and graphic capabilities	**$535** reg. 595.00
Apple Parallel Int.	**$160** reg. $180
Apple Serial Int.	**$175** reg. $195
Centronics Parallel Int.	**$185** reg. $225

DOUBLE VISION	**$295.00**
DISK II	
with controller	**$525.00**
without controller	**$445.00**
MICROMODEM	**$325.00**
PASCAL	**$425.00**
LEEDEX MONITOR	**$140.00**
KG-12C Green Phosphor 12" Screen w/Glare Cover 18 MHz bandwidth	**$275.00**

16K RAMS for APPLE II TRS-80 $59

VERBATIM DISKS 10 for $27

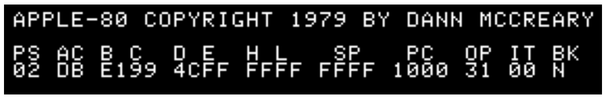
The Computer Stop
16919 Hawthorne Blvd
Lawndale, CA 90260
(213) 371-4010

MON. - SAT. 10-6

Luckily, much of the hard work was done for me in 1979, by Dann McCreary. He created an 8080 interpreter program for the KIM-1, a single-board 6502 computer with even fewer blinking lights and switches than the Altair. I found the binaries and source for SIM-80 in the usual way, through Google and the Internet Archive.

I set about cleaning up McCreary's 40 year old KIM-1 source code, ready to turn it to my will and port it to the Apple II. Once again, Dann had done the hard work for me. Apple-80 was a commercial release of SIM-80 for the Apple II, and I found a rip of the cassette, along with documentation, but no source, at `brutaldeluxe.fr`.

With the KIM-1 SIM-80 source on one hand, and a freshly disassembled binary of Apple-80 on the other, I was able to reproduce the source for Apple-80. My efforts then shifted to updating and augmenting it, relocating the code to run at boot from a ProDOS floppy instead of loading from cassette.



```
APPLE-80 COPYRIGHT 1979 BY DANN MCCREARY
PS AC B C  D E  H L   SP   PC  OP IT BK
02 DB E199 4CFF FFFF FFFF 1000 31 00 N
```

Apple-80 emulates the 8080 processor opcode-by-opcode, and provides a window into the inner workings of the processor as it operates, allowing a user to step and trace assembly code, modify register state directly, and read and write memory - but that's it. A single status line. I wanted more of the Altair experience. I wanted Blinkenlights.

The Apple II has a mixed low-resolution graphics and text mode, with 40 horizontal by 40 vertical rectangular pixels in 16 stunning colors, and four lines of 40-column text below. I designed a low-res screen version of the front plate of the Altair 8800 and scootched the Apple-80 status line into the "plus four" text lines.

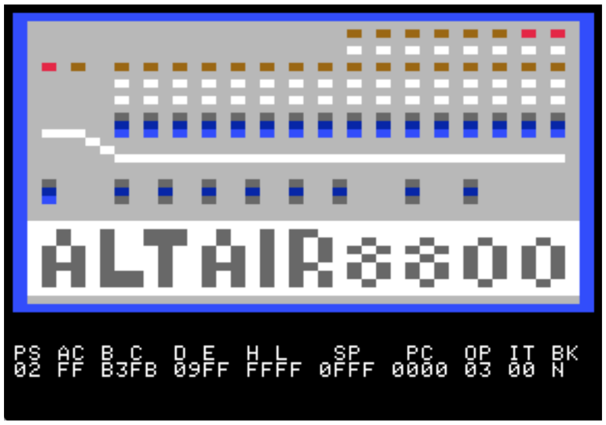
It was then a matter of animating the graphical front end of the newly dubbed Sim-8800.[11] The lights on the front of a real Altair reflect the status of the memory and address lines of the 8080, as well as other processor status bits. The switches are used to change and step through bytes of memory. I added hooks into the step and trace functions of the emulator core to change the proper pixels on the low res screen in order to simulate LEDs turning on and off, and toggling switches in up or down positions. Keyboard commands were then added to flip these virtual switches and change the bits in the emulated processor to the appropriate status.

I could now enter a program into the Sim-8800 the same way a hobbyist who had finally finished soldering together his Altair kit in late 1970s would have.

Byte by byte, flipping switches, and noting the pattern of LEDs, a test program is entered and then run. What better program to test with than the classic "Kill the Bit," which causes the processor to access memory at specific addresses, triggering lights on the front panel to rotate in a pattern.

This program and a more complex Pong-like game worked a treat. I had emulated the Altair out-of-the-box experience on an Apple II - almost.

[11] `unzip pocorgtfo20.pdf SIM8800.zip`



```
PS AC B C  D E  H L   SP   PC  OP IT BK
02 FF B3FB 09FF FFFF 0FFF 0000 03 00 N
```

Opcode Origami

Both the Apple II and virtual Altair were accessing the same 64K of memory space, with the Apple setting aside 4K of that for the Altair to play in - the range from `0x1000` to `0x1FFF`. Below that range lives the Apple's own zero page variables in use by ROM routines, the 6502's immobile stack, and the display buffer for text and low resolution screens. Above, at `0x2000`, sits the emulator program itself, an address set by ProDOS for any program that runs at boot.

The problem, at this point, was not that the Altair was limited to four virtual kilobytes, but that they started above `0x00`. The programs I entered all had to be rewritten, relocated to run at the higher address range, which limited me to very simple programs.

Additionally, any time the virtual 8080 stepped outside of its strict memory bounds, unpredictable crashes happened. If the 8080 program modified a portion of the emulator program by mistake, or ventured into ROM space and triggered one of the Apple's soft switches, all was lost.

Thus began a deep dive into the emulator core - all my changes up to this point had been to relocate routines or add my display functions on top of the existing pieces. Now I was going to have to rewrite portions of Dann McCreary's code to dynamically relocate everything by `0x1000` bytes. This way, an 8080 program designed to run at `0x00` could live in a real chunk of memory at `0x1000` and not interfere with the 6502 zero page.

Each operation of the 8080, and thus the SIM-80 emulator, essentially does one of three things: 1) read a chunk of memory into a register or register pair (`RP`), 2) write the contents of an `RP` to memory,

```
; 0000     org 0000
; 0000     21 00 00          lxi h,0          ;initialize counter
; 0003     16 80             mvi d,080h       ;set up initial display bit
; 0005     01 0E 00          lxi b,0eh        ;higher value = faster
; 0008     1A                beg: ldax d      ;display bit pattern on
; 0009     1A                ldax d           ;...upper 8 address lights
; 000A     1A                ldax d
; 000B     1A                ldax d
; 000C     09                dad b            ;increment display counter
; 000D     D2 08 00          jnc beg
; 0010     DB FF             in 0ffh          ;input data from sense switches
; 0012     AA                xra d            ;exclusive or with A
; 0013     0F                rrc              ;rotate display right one bit
; 0014     57                mov d,a          ;move data to display reg
; 0015     C3 08 00          jmp beg          ;repeat sequence
; 0018     end
```

Kill the Bit source, published by Dean McDaniel in 1975.

PCYACC™ Version 2.0

PROFESSIONAL LANGUAGE DEVELOPMENT TOOLKIT

Professional Version $395.00—Personal Version $139.00

Includes "Drop In" Language Engines for SQL, dBASE, POSTSCRIPT, HYPERTALK, SMALLTALK-80, C++, C, PASCAL, and PROLOG.

PCYACC Version 2.0 is a complete Language Development Environment that generates ANSI C source code from input Language Description Grammars for building Assemblers, Compilers, Interpreters, Browsers, Page Description Languages, Language Translators, Syntax Directed Editors, and Query languages. **Complete grammars, Lexical Analyzers, and Symbol Table Management for ANSI C, K&R C, ISO Pascal, dBASE III/Plus and IV, SQL, C++, Smalltalk-80, APPLE HyperTalk, C&M Prolog, YACC, LEX, and POSTSCRIPT are included. OS/2 and MAC versions are available.** Example application sources are provided to be used as skeletons for new programs. Examples include a desktop calculator, an Infix to Postfix Translator, a DBASE and SQL Syntax analyzer, an implementation of the PIC[ture] language, and a C++ to C translator.

- ■ Lexical Analyzer Generator ABRAXAS PCLEX is included
- ■ Quick Syntax analysis option
- ■ Optional Abstract Syntax Tree
- ■ Advanced Error recovery Support Provided
- ■ Manual "Compiler Construction with PC'S" included
- ■ All examples include FULL SOURCE
- ■ 30 day money back guarantee
- ■ No charge for shipping anywhere in the world

To order, please call

1-800-347-5214

ABRAXAS™ SOFTWARE, INC.
7033 S W Macadam Ave. Portland, OR 97219 USA
TEL (503) 244-5253 · FAX (503) 244-8375
AppleLink D2205 · MCI ABRAXAS

or 3) carry out some manipulation of bytes within the registers. There are a handful of other unique opcodes that have different effects, but the bulk of the opcodes fit into one of those three categories.

Any routine instructing the emulator to read from memory or write to memory (including the program counter [PC] that keeps up with the current instruction address) had to be modified. I added `0x1000` to the `PC` for reads, then subtracted `0x1000` for execution. Writes were handled similarly, adding `0x1000` in order to write the correct real addresses.

As each edge case was found, the off-by-one errors began to fall, and soon I could run rudimentary programs again - this time, as they were originally written. There was one binary beastie I wanted to tackle in particular, but it would require having some means of doing input and output. The next goal was something slightly more complicated than turning LEDs on and off.

Talk To Me

The first peripheral most Altair owners would add to their machines was some sort of input and output beyond the built-in LEDs and switches. A paper tape reader and teletype printer opened up a world of possibilities beyond Kill the Bit, and turned the hobbyist curiosity into a truly useful home computer - for those homes that could accommodate a clanging, clacking teletype. These were connected to the Altair with a serial board, the 88-SIO or later 88-2SIO.

Once again diving into the Internet Archive, I surfaced with complete documentation of the 88-SIO board, including full assembly and installation instructions as well as theory of operation. Most importantly, a table of the status bits was included, and assembly listings of programs for testing the board. Bonus!

The internal workings of the SIO are not important, or indeed that complicated. In order to take in bytes from the outside world, or emit them back out again, the SIO utilizes two of the 8080's I/O ports. One is used for status, both setting and reading, the other for transmitting and receiving bytes. Being the first such device available for the Altair, those functions default to ports `0x00` and `0x01` respectively.

Emulating the external teletype functions, I used the Apple II's built-in ROM functions. Any bytes received from the virtual SIO are simply printed to the screen through the "character output" or `COUT` function call. This handles everything from scrolling the text window, to wrapping text at 40 (or later, 80) characters, to linefeeds and carriage returns. Reading the keyboard buffer at `0xC000` provides input to the SIO, one byte at a time.

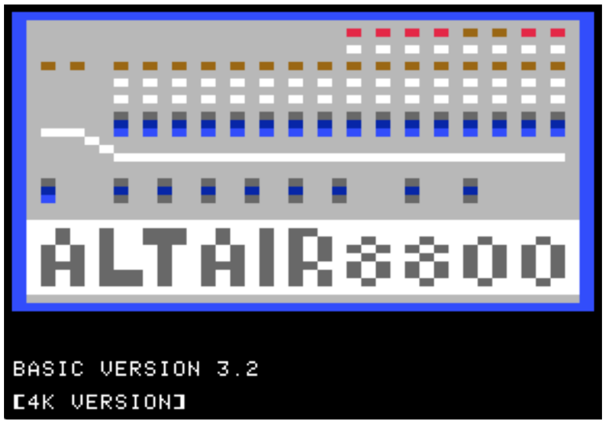
I added a code to the emulation routines handling the `OUT` and `IN` 8080 opcodes to make them call my virtual SIO subroutines. These subroutines in turn set the proper status bits, indicating that the card is either ready to receive or ready to send. As far as the virtual Altair is concerned, it's connected to a ridiculously fast serial board that never has to wait for a byte to buffer, and it's always in sync with the receiving printer.

Ya BASIC

Microsoft, at the time styled Micro-Soft, was formed in order to sell a BASIC interpreter to MITS after the Altair was revealed. Their initial product ran in 4K (check) and needed only a serial connected teletype for I/O (check).[12]

The program itself is much too large to enter by hand. While I could transfer the bytes in one at a time through the virtual paper tape machine I had created with the emulated SIO, I took a shortcut instead. I cheated and had ProDOS load BASIC into the virtual Altair's memory directly. When Sim-8800 booted up, BASIC was already sitting at `0x00`, ready to run.

And run it did. The first time the prompt spat out the bottom of the Apple II screen, asking me how much memory the system had, I grinned like a fool.



```
BASIC VERSION 3.2
[4K VERSION]
```

[12]`http://altairbasic.org/`

I could now create and run a program in an interpreted language created by a program running on a virtual 8080 processor, emulated by another program running on a 6502 processor.

Then the text scrolled past the four lines at the bottom of the mixed low res graphics screen, and I coded up a full-screen switch.

Then the default line length turned out longer than the 40 columns of the Apple II standard text mode, and I knocked together a switch to set 80 column text mode.

But can we go deeper?

With 4K of virtual memory, and the optional trigonometric and random functions turned on, BASIC was left with a meager 726 bytes of memory to run programs. This was a significant roadblock to many ambitious Altair owners in their day as well, and was cause for many memory upgrades.

Remediating this limitation in my emulated Altair meant moving my program from `0x2000` to a spot higher in memory. This entailed writing a small program that would load at boot time into `0x2000`, then load Sim-8800 from disk into a higher memory location and hand off control. The loader, its job complete, would get clobbered by the next phase, which loaded a more complex, 8K BASIC into memory.

But why stop there? The Apple II has 64K of memory space, albeit in a rather hodgepodge arrangement.

As outlined by Gary B. Little in *Inside the Apple IIe*, reproduced on page 20, the first roughly 4K of RAM is associated with zero page variables, stack, and text/graphics buffers. On the higher end is the ROM, the 4K at `0xC000` for memory-mapped I/O and peripheral cards, and everything else above `0xBF00` is used by ProDOS. All this leaves about 36K of usable space on a standard 64K Apple II system. If I could keep my program, including the graphics for the virtual Altair front panel, at less than 4K, I could emulate a 32K 8080 system on a 64K 6502.

And so I did. All my code and data lived at `0x9000` through `0xBF00`, with plenty of room to spare, while Sim-8800 addresses everything from `0x1000` through `0x8FFF`, and pretends it's `0x0000` to `0x07FFF`.

32K felt luxurious compared to the 4K I had previously eeked out a working program in, so I was happy with it for a while. I found a chess program built for the 8080, and played a few moves against it. I even worked out a way to load text files from floppy disk into the emulated paper tape reader, meaning I no longer needed to type in ever more complicated BASIC programs.

And if I ever wanted to save one of those programs back out from the emulator, I could. Well. Um. Paper tape? Oh.

Back Off - I'm A Scientist

The next obvious peripheral most Altair owners would have sprung for in those early days of home computing was a floppy drive. At 8" across, these disks were truly floppy, contrasted to the comparably compact 5.25" "mini" floppy disks that would come later.

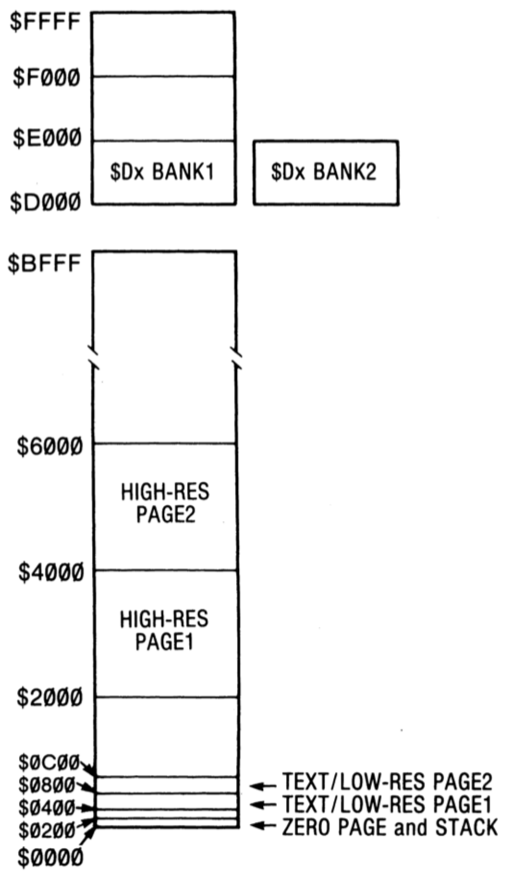
The 88-DCDD (sensing a naming convention here?) was the 8" floppy drive of choice for those early machines, and came, like the 88-SIO, with a complete set of assembly instructions and tables of I/O bytes. Credit, once again, to the Internet Archive for the documentation.

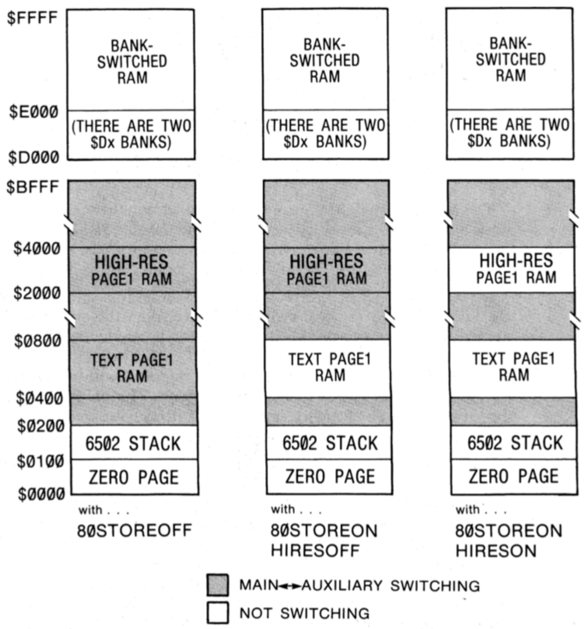
8" Altair disks are preserved for the ages in archived DSK files. Thankfully for me, the DSK format is a byte-for-byte image of what one would find on the disk itself, contiguous and without preamble. The physical format allows for 77 tracks of 32 hard-defined sectors, each with 137 bytes of data - 128 bytes with a small lead-in and out, plus space for a checksum - for a total of 330K of data per DSK.

The Apple II generally boots from 140K 5.25" floppies - you may sense a problem here.

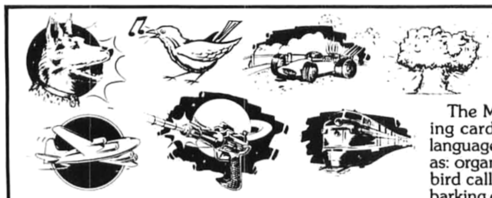
Luckily, my choice of ProDOS for booting the Apple II allowed me to leverage its ability to boot from hard drive volumes up to 32 MB. Today, those volumes generally live on some sort of solid state storage device, like a CFFA-3000. In fact, I hadn't touched a real floppy disk in this whole process - all of my disk storage for the Apple II was emulated by either a CFFA or a Floppy Emu, both of which present solid state storage media (Compact Flash or SD card) to the Apple as if it is a floppy disk or spinning drive.

The storage issue resolved, I could focus on the actual emulation. Having tackled the SIO emulation, the DCDD was a relative breeze - that is, if a scorching hurricane of sand and broken glass could be called a "breeze."





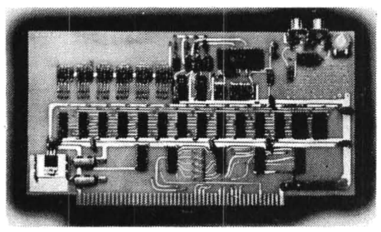
Apple IIe Memory Maps.
Reprinted from *Inside the Apple IIe* by Gary B. Little.



New and Unusual SOUNDS for your Computer $149.95

The Microsounder is an S-100 compatible sound generating card that can be programmed in BASIC or assembly language. Three to five lines of code generates such sounds as: organ music, sirens, phasers, shotguns, explosions, trains, bird calls, helicopters, race cars, airplanes, machine guns, barking dogs, and many thousands more. Only a few minutes of time is needed to patch the sound code into existing programs.

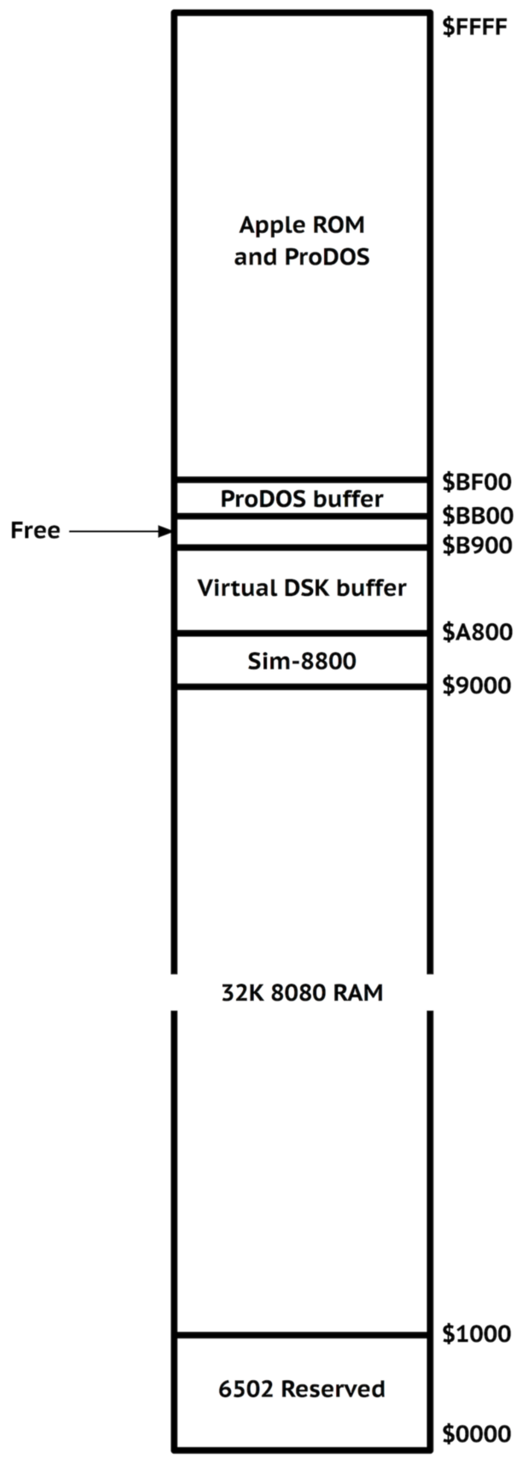
The Microsounder is assembled and tested, and comes complete with sample code, two game programs, and two utility programs for creating almost any sound.



BOOTSTRAP ENTERPRISES INC.

BOOTSTRAP ENTERPRISES INC.
100 North Central Expwy., Richardson, TX 75080
(214) 238-9262

Name ______
Address ______
City ______
State ______ Zip ______
Add $4.95 for Postage & Handling
☐ Check Enclosed   Texas Residents add 5% Sales Tax
☐ VISA # ______
☐ MASTERCHARGE # ______
Exp. Date ______



My decision to tie every `IN` and `OUT` opcode to the SIO emulation came back to bite me here, and I was forced to rip out vital chunks of code in order to rebuild them in a new, better abstracted image. Now, in addition to an infinitely fast serial port, the Altair was connected to a floppy drive with near-zero seek time spinning at roughly 3.75 million RPM.

The only easy part of the disk emulation comes thanks to the hard sectoring of the disks. While the actual data on disk is interleaved to give the computer time to process data from sector `N` before being presented with the data on sector `N+1`, the hardware treats the sectors as numbered sequentially. Interleaving is handled by the software, so I didn't need to build an interleave table. It's also up to the program reading the data on disk to build and decode any checksums on the data, tasking the drive only with reliably reading and writing bytes.

To present the Sim-8800 with bytes from a virtual disk, I needed to load in data from the DSK file on a real disk (in the way that an SD card emulating a spinning drive is a "real" disk). To do this, ProDOS can read arbitrary pieces of a file, given a starting byte offset and a length. To properly emulate a spinning disk, I load in one full 4,384 (32 x 137) byte track at a time into memory. This is queued 1K at a time by ProDOS into a buffer before being moved into place. If you can tell I'm running out of bytes to shove things into, you're still not wrong.

When the Altair starts asking for data, there's no way to tell what track it's looking for, or what sector. The virtual DCDD simply increments the track number and grabs 4.3K from the DSK, overwriting the previous track's data, when Sim8800 tells it to step the motor inward by a track. Then, when Sim8800 reads the status byte for the drive, the DCDD increments the sector by one. This way, the program loading data only needs to wait a few virtual CPU cycles for the proper sector to come by.

And then, there's the bootstrapping problem. Whereas the Altair knew what to do when told to run BASIC, that was because I was loading BASIC into virtual memory before the Altair booted. With a program on disk, I was no longer able to cheat to get by. I needed a bootloader. Luckily, the internet provided again. The same site I kept coming back to for DSK files and other information not easily found on `archive.org` had a variety of boot ROMs for the Altair - `deramp.com`.

I acquired a proper bootloader, which was now loaded into memory at boot time, much like a ROM board used a real Altair owner. Booting from the ROM is easy, only requiring the computer to examine the proper place in memory - a simple incantation consisting of flipping the front panel switches, and then telling the machine to run. The loader relocates itself in memory away from ROM space, modifying itself as necessary along the way, based on the front panel switch settings, and finally runs at its new location.

This pass accesses the disk at track zero, sector zero, and loads data from disk into memory at `0x00`. After reaching the end of track zero, the loader hands off control to the program at `0x00`, which is then responsible for loading the remainder of the operating system from the disk.

After some additional effort to get the virtualized DCDD to write data back to a DSK file, I was able to read, run, and save BASIC programs stored on a DSK under a Disk BASIC and Altair DOS. I could now run an interpreted program loaded into an operating system in 32K of virtual memory on an emulated 2 MHz 8080 from an emulated 8" floppy disk which was really a file inside another file on an SD card emulating a spinning hard drive feeding data into an Apple II with 64K of RAM and a 1MHz 6502.

Real-Time C

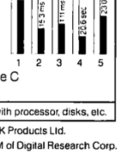
for 8080, Z80

A Run-Time Library for Whitesmiths' C 2.1

- ■ Fast execution
- ■ ROMable
- ■ No royalties
- ■ Fully reentrant machine support
- ■ CP/M file support
- ■ Error checking
- ■ Usable with our AMX Multitasking Executive

Benchmarks

1. Int to ASCII conv.
2. Long to ASCII conv.
3. Long random number generator
4. Double 20 x 20 matrix multiply
5. File copy (16kb)



4 Mhz Z80, 8" SD diskette. Times may vary with processor, disks, etc.

AMX and Real-Time C are trademarks of KADAK Products Ltd.
A-Natural is TM of Whitesmiths Ltd. CP/M is TM of Digital Research Corp. Z80 is TM of Zilog Corp.

Real-Time C	**$ 95**
manual only	**$ 25**
source code	**$950**
Intel mnemonic to A-Natural converter	**$ 50**

KADAK Products Ltd.
206–1847 W. Broadway Avenue
Vancouver, B.C., Canada V6J 1Y5
Telephone: (604) 734-2796
Telex: 04-55670

Catch All That?

But, again, can we go deeper? The answer is yes, but first, a bit of a diversion:

"If you wish to make an apple pie from scratch, you must first invent the universe." - Dr. Carl Sagan, 1980

To paraphrase Dr. Sagan, in order to play a computer game, you must first invent the computer. To this end, in 1979 the authors of what would eventually become the Infocom interactive fiction title Zork, manifested from pure imagination and no small amount of magic a virtual computer to run it on. They called it the "Z-Machine."

Much has been written about this virtual machine, its antecedents and its successors. Several versions of the Z-Machine were created, and even today there is a vibrant community of authors and creators who still program for it. The fabled machine does not exist in a physical form of chips and wires, but only in the imagination.

Imagine a computer - depending on the accuracy and veracity of your imagination, you may come up with something that contains a processor, memory, storage, and some forms of input and output. Good imagining, neighbor!

In order for this imaginary machine to function in the real world, and run the programs, it must be implemented in code on an actual computer. Z-Machine interpreters, or programs that emulate a virtual Z-Machine, have been written for nearly any platform you can think of. An atypical, but not unheard-of system for running Zork in its heyday might have been an Altair 8800. Now imagine one of those.

Actually, no need to imagine. I already had a virtual Altair 8800. Dare I dream? Could it run Zork?

In a word: No. Not yet.

Giving It All I've Got

In order to run Zork on an Altair, said Altair must have some kind of text terminal (check), a floppy disk to read and write the program files (check) and be running the CP/M operating system (hmm...). Digital Research's CP/M was a contemporary of and competitor against Micro-Soft's DOS, and early versions exist that will barely squeak by with just 24K of memory.

I should note here that at each point in my journey, I found and fixed numerous bugs in my code, and limitations of the original Apple-80 emulator core. These were flaws were revealed by the ever expanding and complex convolutions I was forcing upon it. 8K BASIC uncovered issues with repositioning the stack pointer; Disk BASIC had trouble with reading from virtual disk, and Altair DOS with writing to it. At multiple stops along the way, I was forced to backtrack - faced with the consequences of fixing a load-bearing bug, while wondering how this whole thing had even worked in the first place.

Debugging my own 6502 spaghetti code is one thing, my head was swimming trying to understand what the emulated 8080 code was intended to do, while also handling translation of memory addresses from virtual to real.

`Deramp.com` provided a DSK of 24K CP/M, version 1.4, which ran like a champ as I put it through some limited testing. The distribution on the DSK was intended to be used to make another bootable disk, rather than used by itself, but it worked as proof of concept that Sim-8800 could, indeed, run CP/M.

But 32K just wasn't going to suffice. In fact, CP/M 1.4 wouldn't cut it, either. According to my research, I was going to need at least 48K minimum, and CP/M 2.2 for the Z-Machine interpreter.

As I've demonstrated, on a typical 64K Apple II system, there's no way to load up 48K of anything, let alone leave room for an emulator program to manage it all. I would have to revise my minimum system requirements for running Sim-8800.

Zoom and Enhance

Enter the Apple IIe. While the base system still faces the typical 64K limitation, a common upgrade for the IIe is an 80-column card with an additional 64K of "auxiliary" memory on board. 64 glorious kilobytes of usable RAM, at my fingertips! Why not just run the emulator itself in main memory, and shuttle the virtual memory into the aux memory on the card? Because that would be too simple.

You see, in order to access that auxiliary memory outside the 64K limit on an eight-bit system, one must perform bank switching. Chunks of memory are turned off and others turned on in their place. This process is handled through soft switches, memory locations in the ROM area that inform the processor how to perform whenever they are accessed. You can't have access to both aux and main RAM at the same time. My code would need to exist in both places at once in order to continuously maintain control.

Add to this the fact that the Apple mirrors portions of the main memory in auxiliary, so that when banked out, the processor still has access to the peripheral ROM, zero page and stack, among other things. The end result is about 32K of usable memory in the aux space to add to the 32K I was using in main memory. I had my 64K. Only, like Waffle House hash browns, it was scattered, smothered and chunked.

I endeavored once again to dynamically remap the 8080 virtual memory, retracing the paths I had forged in my previous efforts. This time, in addition to shifting all the virtual addresses up `0x1000` real bytes (to make room for 6502 zero page, etc.) I was bank switching any virtual address above `0x7FFF` into the auxiliary space. Once there, the address would need to be shifted down `0x8000` bytes again, since aux space counts up from zero. Then, everything gets shifted up again another `0x1000` bytes, since the 6502 zero page is mirrored in aux.

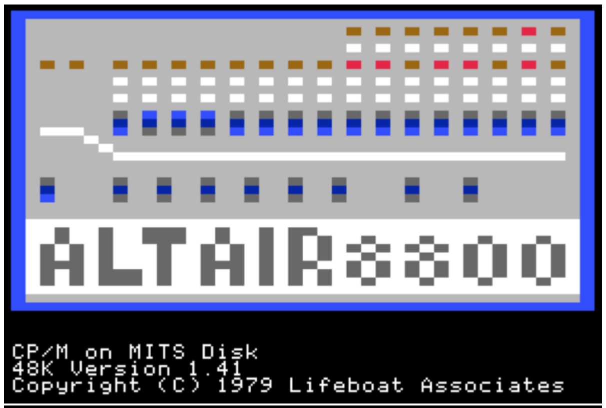
All of these mathematical gymnastics need to happen any time the virtual 8080 accesses any virtual address, whether it's the PC fetching an opcode, reading bytes, or writing bytes in memory. Keeping this all straight in my head was nigh impossible, and it led to some frustrating, if spectacular crashes, as virtual programs that used to run perfectly well in 32K suddenly overran the emulator's bounds.

I loaded in and bootstrapped CP/M 1.4 from a DSK intended for a 48K system. It worked!

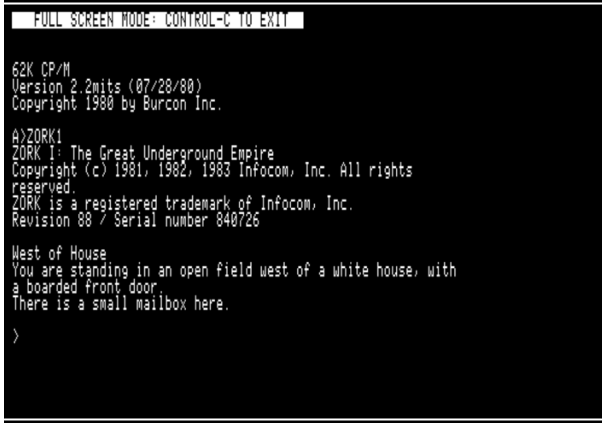
With some trepidation, I pointed the emulated disk drive at a file named `ZORK.DSK` and booted once more.

Finally - after revealing yet another edge case, and guiding me to yet another flaw in my math related to the virtual stack pointer, which took me two days to find and fix - it worked.

I was west of a white house. I took the lamp and the sword. I killed the troll and got lost in the maze of twisty passages, all alike.



```
CP/M on MITS Disk
48K Version 1.41
Copyright (C) 1979 Lifeboat Associates
```



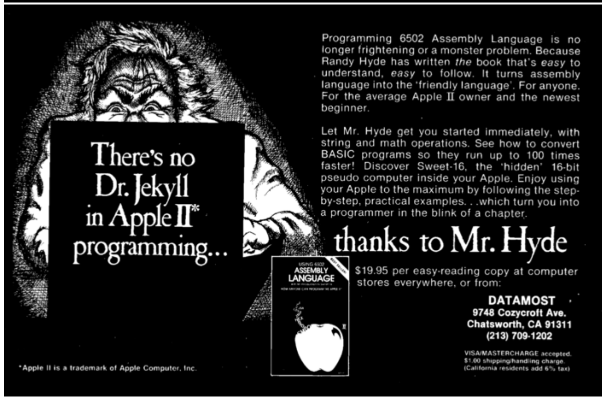
```
 FULL SCREEN MODE: CONTROL-C TO EXIT


62K CP/M
Version 2.2mits (07/28/80)
Copyright 1980 by Burcon Inc.

A>ZORK1
ZORK I: The Great Underground Empire
Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights
reserved.
ZORK is a registered trademark of Infocom, Inc.
Revision 88 / Serial number 840726

West of House
You are standing in an open field west of a white house, with
a boarded front door.
There is a small mailbox here.

>
```



There's no Dr. Jekyll in Apple II* programming...

Programming 6502 Assembly Language is no longer frightening or a monster problem. Because Randy Hyde has written *the* book that's *easy* to understand, *easy* to follow. It turns assembly language into the 'friendly language'. For anyone. For the average Apple II owner and the newest beginner.

Let Mr. Hyde get you started immediately, with string and math operations. See how to convert BASIC programs so they run up to 100 times faster! Discover Sweet-16, the 'hidden' 16-bit pseudo computer inside your Apple. Enjoy using your Apple to the maximum by following the step-by-step, practical examples. . .which turn you into a programmer in the blink of a chapter.

thanks to Mr. Hyde

$19.95 per easy-reading copy at computer stores everywhere, or from:

DATAMOST
9748 Cozycroft Ave.
Chatsworth, CA 91311
(213) 709-1202

VISA/MASTERCHARGE accepted.
$1.00 shipping/handling charge.
(California residents add 6% tax)

*Apple II is a trademark of Apple Computer, Inc.

I was playing a game written for an imaginary computer, which was being emulated by CP/M with 64K of contiguous virtual memory on a virtual 2 MHz 8080 CPU loading data from a 330K eight-inch virtual floppy, itself emulated by a 1MHz 6502 Apple IIe with 128K of bank-switched memory, loading data from a DSK file held on an SD card pretending to be a spinning hard drive. Did I miss anything?

Oh yes. All of this was running inside the emulator Virtual][on my Mac.

You see, aside from my earliest versions of Sim-8800, the whole development process was done on my Mac, the part of the Apple II played by Virtual][, a most excellent emulator by Gerard Putter.

My workflow begins in BareBones' BBEdit, where I write the assembly code. This is assembled into a binary by Merlin32 by Brutal Deluxe. Merlin32 is a modern command line rewrite of Merlin, an assembler that ran on Apple systems. The binary, and other files like `CPM.DSK`, are compiled into a 2MG disk image by CiderPress, which only runs on Windows, or WINE, in my case.

The 2MG is loaded into an emulated CFFA-3000 in Virtual][. Yes, it emulates the card emulating a hard drive. This way, disk access is even faster than simply emulating the hard drive, as Virtual][strives for accuracy in all things, even disk access latency.

Which brings me to a note about speed - you may have asked yourself somewhere while reading this missive, "just how fast can a 1MHz CPU emulate a 2MHz one?" The answer is slowly, unusably slowly. The only way any of the Altair software is even remotely tolerable, from 4K BASIC all the way up to Zork, is through the speed boost of emulation in Virtual][. In emulation, I can choose to be cycle accurate, pinning the emulate 6502 at a precise 1.023 MHz, or I can press a button and run the emulation as fast as my 2.3GHz i7 can handle.

Early on, I ran a benchmark to see just how slowly the Sim-8800 emulation really ran. I knew it took sometimes several hundred 6502 cycles to emulate a single 8080 cycle, drastically more if I was updating the graphics display at the same time. A simple prime number finding BASIC program, which on a real Altair should take 80 seconds or so, instead took 3 hours, 25 minutes without acceleration.

But can we go deeper?

Probably, but you might get eaten by a grue.