Foenix Rising

A newsletter for developers, users, and enthusiasts of Foenix Retro Systems products

(FLASH !)

Aug. 2023 / F9

This article begins with a story of how the need for a simple utility turned into a quest to deliver new operating environment for a computer that already had one. Along the way, we acknowledges 'giants', as in "standing on the shoulders of ...".

While plenty existed (magnetic core memory, logic gate modules, electricity) before CPUs were affordable, the advent of just about everything else that mattered became available during the golden-age of microcomputers (1975-1990). Our shared history is as storied as it is important, so we will point out a few signposts en route. Enjoy!

Making your own tools / a convergence of needs

The machine shop

As a youngster, I was fortunate to have been exposed to influences that would end up invaluable to my future, and in ways I never imagined. Rarely, do we see *it* coming.

For me, #1 was my step-father, Bert. When he wasn't yelling at me to get my broken-down 'parts-car' out of the yard, he was clocking overtime as a laborer working for U.S. Navy contractor Grumman Aerospace.

I never understood what Bert did for Grumman precisely, (he was 20 years older than my mom and didn't talk much) but his toolbox and bench were full of machined aluminum parts, fasteners, and industrial tools. He was less than thrilled that I poked around in his stash because I misused his tools[1] and rarely put anything back in place.

Born in 1926, Bert was a high-school dropout and he reportedly lied about his age to enlist in the Navy during WWII. All I knew was, the guy was a legend. Each summer, he would drag our family to the "Grumman Family Picnic & Air Show" where several thousand Long Islanders witnessed might & machinery in the form of the F14 Tomcat and a dozen other aircraft flying for the entertainment of a massive and proud community. Grumman also powered a bustling local electronics industry; talk about being born in the right place and time.

An obvious influence was education, but not in the way you might expect. As a middle school student in the 70s, girls and boys in the Northeast U.S. were led through Industrial Arts training, which included metal shop, wood shop, and in high school, drafting and machine drawing. These were the skills that educators felt were important for our generation.

What does any of this have to do with Foenix computers? Loosely, everything. Sometimes the universe grants you gifts and sometimes, it gives you something better; pieces of a puzzle, leaving the rest up to you.

In the first few issues of this Newsletter, we used BASIC816 to build a few lightweight utilities; one to display sprite data as text and one to manipulate color palette encoding; we also leveraged Unix shell commands hosted on another platform to manipulate and visualize 8-bit fonts and 24-bit color. It's all about taking what you *do* have to create something that you *don't* have.

In this article, we will begin a build of the ultimate microcomputer accompaniment, a 6502 machine language (ML) monitor for the Foenix F256 platform. The plan:

this issue
- ✓ Write a memory examination utility from scratch, but tailored for the F256 platform. We will do so in support of a fork of a kernel environment that is also being developed; stay up to date on my YouTube channel: 8-Bit Wall of Doom) for more insight.
- ✓ Take Steve Wozniak's original Apple I disassembler as published in Dr. Dobb's journal, and port it to the F256.

next issue
- Modernize the disassembler to recognize the full WDC 65C02 instruction set (8 new instructions plus added addressing modes for the original MOS6502 opcodes).
- Consider if a simple version can run under TinyCore/ MicroKernel and discuss what a minimum viable kernel ought to include from a services perspective.

later
- Port and integrate Charles Brannon's SpeedScript editor (because it is tiny, refined, and well documented).
- Port and integrate a File Manager with 'launch' capabilities from my own 6502 Source code (1986)
- with more to come…

… but first, some history:

In the beginning: The MOS Technology KIM-1

By the middle of 1975, MOS was distributing samples of their first soon-to-be blockbuster product: the 6502 CPU. They were also readying the KIM-1 for release.

The KIM-1 was a single board development system for would-be 6502 hobbyists and integrators, and it was literally named after its key feature; it had a keypad for hexadecimal input and a 6 digit hexadecimal LED display, plus a monitor *program* masked into the MOS 6530 RRIOT IC. (The 6530 preceded the famous MOS 6522 and shared some of its features; some say these ICs are as important as the CPU; more on them another time.)

[1] For a *great* 'Bert' story, see the bottom of pg. 6

In 1975, machine language monitors for microcomputers were a *new* idea. The IMSAI 8080 and Altair 8800 (two other hobbyist microcomputer offerings) were based on the Intel 8080 CPU and only offered toggle switch input and a set of single element LEDs for status. If it mattered (it didn't), Digital Equipment Corp (DEC) might have suggested that the IMSAI interface was a rip-off of the PDP-8 mini released a decade earlier.

The KIM-1 seemed primitive but delivered a level of satisfaction and immediacy akin to the much sought after x-ray vision superpower (sensationalized in super-hero comic book ads), otherwise known as the ability to slide a virtual window over something (memory) to see its contents, otherwise obscured; to set or advance a program counter; or to modify memory by keying in code and data to ultimately execute. Ladies and Gentleman, the monitor had arrived.

A recommended read, aptly named the First Book of KIM, was published by Hayden and edited and compiled by Jim Butterfield and others; it brought the KIM-1 from science project to a near-friendly consumer product. The ambitious preface said it all:

> ***Dedicated to the person who just purchased a KIM-1 and doesn't know what to do with it . . .***

The book included technical detail written in Jim Butterfield's approachable style followed by dozens of ready-to-run amusements, data acquisition and computational programs and tools, plus encouraging commentary. Here's an excerpt from "`LOOKING AT MEMORY`" on the first page:

If you've just turned your KIM system on, press the RS (Reset) button to get things started. Hit the following keys: AD (for ADDRESS) 0 0 0 0. You've just entered the address of memory cell 0000, the lowest numbered one in memory. The display will show 0000 (the number you entered) on the left. On the right, you'll see the contents of cell 0000: it will be a two digit number. That number might be anything to start with; let's change it.

Press key DA (for DATA). Now you're ready to change the contents of cell 0000. Key in 44, for example, and you'll see that the cell contents have changed to 44.

Hit the + button, and KIM will go to the next address. As you might have guessed, the address following 0000 is 0001. You're still in DATA mode (you hit the DA key, remember?), so you can change the contents of this cell. This time, put in your lucky number, if you have one. Check to see that it shows on the right hand part of the display.

This kind of memory - the kind you can put information into - is called RAM, which stands for Random Access Memory. Random access means this: you can go to any part of memory you like, directly, without having to start at the lowest address and working your way through. Check this by going straight up to address 0123 and looking at its contents (key AD 0 1 2 3); then address 0000 (key AD 0 0 0 0), which should still contain the value 44 that we put there.

As the world learned more about the 6502 and the KIM-1, Steve Wozniak and others began publishing fully commented assembly code in educationally focused articles which were generally applicable to kit systems (recall, there was no Atari or Apple just yet). Much of what was published was distributed through grass-roots newsletters and one such publication, Dr. Dobb's Journal, became an aggregation point for these efforts, including the disassembler that we cover below.

In Volume 1 of Hayden's article reprint of Dr. Dobb's, there are 1/2 a dozen 6502 related pieces, the first of which concerned capturing of breakpoint (`BRK`) data by John Zeigler, originally published in March of 1976 at the Homebrew Computer Club. The last 6502 article of the first year of Dr. Dobb's was entitled "A 6502 Disassembler from Apple" and it was published in September of 1976 by Steve Wozniak and Allen Baum.

In the 12 months following the release of the KIM-1, the Apple 1, Commodore PET; and ultimately, the Apple II were released (~6 mos. apart). Each system included a monitor in ROM, but that changed in 1980 when the VIC-20 was released (thanks Jack!).

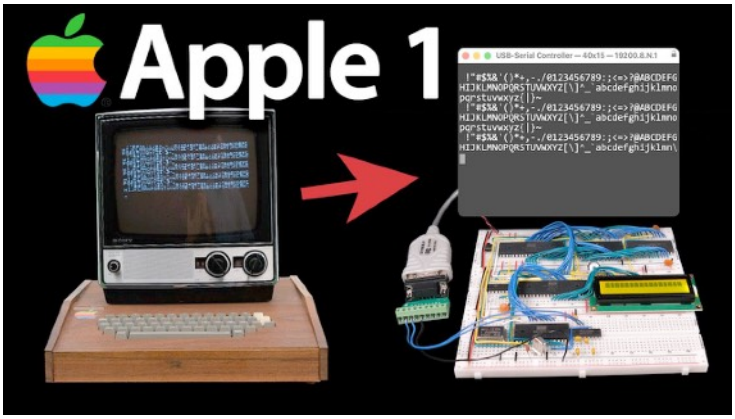
Meanwhile, Apple II's `CALL -151` implementation stood out. The monitor itself was nothing special, but Woz's ingenuity led to a method to redirect I/O from analog tape and then to support the yet to be invented SuperSerial card that hobbyists across the next decade would depend upon for off-host backup & restore and eventually, boot disk creation for a systems otherwise left for dead. This bare metal foundational capability existed across the Apple II+, IIe, IIc, and even the 65816 based IIGS.

Steve Wozniak and his monitor related efforts

Volumes have been written about Wozniak's embedded Apple 1 monitor. Built from 124 lines of assembly language source (248 bytes of object/machine code), the very first Apple computer came with the monitor stored in upper ROM occupying the last page of memory from $FF00 to $FFF7 (Woz used all but 2 bytes of space, not including the 6 bytes worth of 6502 vectors).

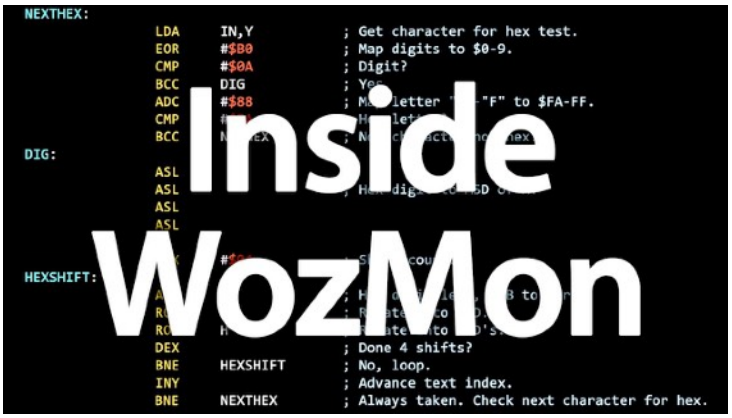
With only one page of memory 'used', and no graphics or kernel to speak of, the Apple 1 must have felt spacious. How on earth would a user ever use up to 64K of RAM? Not only were programs scarce, but digital content did not yet exist, except within research and government agencies. Of course RAM was prohibitively expensive in those days and the most dense RAM footprint that was feasible provided 4K or 8K (multiples more than the KIM-1, but otherwise tiny). Tailored for terminal input/output, the new monitor was a fond departure from 'toggle-in' and keypad interfaces.

For our project, we developed our own monitor (you'll see why); but the following two recently published Ben Eater videos provide a full retrospective of this first noteworthy terminal based effort. Ben examines Woz's early work by way of the Apple 1 and specifically, the famous first monitor affectionally known as 'WozMon'.



Part 1 - Ben Eater's Intro; Apple 1 history included (~15 mins.)

In the following video which I consider Part 2, Ben navigates a deep dive into the WozMon code; this is the part that most impresses me, and it's not just Woz. It's also Ben. He breaks down complex subject matter expertly. If at some point in watching this, you feel lost, hop back to Ben's channel and check out his 6502 series (playlist). It will take some discipline to power through it, but it's well worth the time investment and you need not purchase the kit. You can (I did), but even if you just follow along at home, you'll feel like you own one. That is the magic of Ben's immersive presentation style:



Part 2 - Ben's 'Inside', deep dive (~37 mins.)

What about the rest of the world?

As a teenager learning assembly language in the early '80s, monitors were mysterious to me. Since they were distributed as machine code, they gave zero insight into how they worked. From a terse list of commands they provided the power to produce programs from a small footprint without the fanfare of full assemblers and the cost of 'development environments' that tended be as expensive as the computers they ran on.

But monitors were (since day one) free; and while there were commercial products such as HesMon by Human Engineered Software (developer of GridRunner) or as a utility within another cartridge such as Epyx Fastload, scant documentation meant the learning curve was steep. Oh, another thing… YouTube did not yet exist and Al Gore was flunking College Science during the formative ARPAnet years. He did not invent the internet :)

But Woz and a colleague named Allen Baum, on the other hand, wrote (and shared with the world) an early 6502 disassembler toolkit, free to anybody that wanted to type in the hex digits and save it off to tape.

Published in the September 1976 issue of Dr. Dobb's Journal (printed on page 22-25) "A 6502 Disassembler from Apple" squeezed a surprising amount of utility into a single page of code but then used an additional page for tables of data including packed mnemonics, encoded addressing modes, and some wizardry.

Fast forward to today, and *assemblers* are at last, free! Merlin32, descendant of the original Apple II assembler by Glen Bredon is a full-featured cross-assembler available for Linux, MacOS, and Windows. 64tass, cc65, and others are also free and are well supported.

But a noteworthy platform example for the Commodore 64, still relevant today, is Turbo Macro Pro, originally written by Wolfram Roemhild. Robin from the **YouTube channel "8-bit Show and Tell"** is an avid user and features TMP across many of his videos; this one in particular provides a good beginners introduction. Bookmark this moment, it will be relevant later!

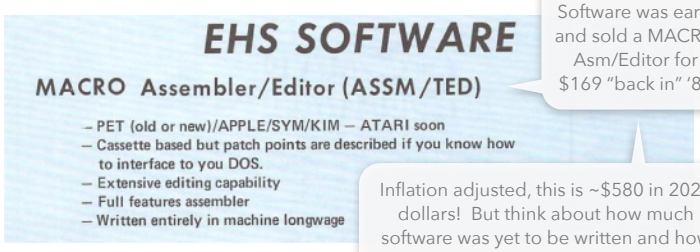
Basic monitor operation and differentiating features

In primitive form, a monitor allows examination of memory contents and may also include functions to display registers including the program counter, status byte, and stack pointer. Monitors historically operated with hexadecimal values exclusively, accepting whitespace delimited two-digit numbers strung together as input. Ultimately, there is the 'go' command ('G') which executes code from a given address.

Fancier efforts tack on interpreted ASCII display views that even allowed search and edit of text; and as mentioned, some include disassemblers and even simple assemble utilities with relative branch offset resolution.

An important aspect of any development or debugging environment is how recovery and workflow is dealt with. Assembled machine code is unforgiving and it's easy to experience a crash or lock-up. Restoring (via NMI vector) to a stable operating base, or via documented recovery address is an important 'tool' to have in your arsenal, especially while debugging. More on this subject to be discussed in future.

Across the next pages, we will discuss the development of a purpose built F256 monitor and a port of the Wozniak disassembler toolkit from Dobb's.



Retro distraction
a vintage ad from 1981

Introducing nanoMon

nanoMon is being designed to function as an embedded monitor providing data interrogation, manipulation, and load and save for the F256 platform. Longer term, the project might consider on-platform assembly and debugging but at the moment, sample prototype code is being developed to measure what fits into an 8K envelope and what can be delivered by end-of -year.

It is anticipated that at *first release*, nanoMon will be a CLI (versus an open screen editor) based monitor, coded to work within the native TinyCore MicroKernel present in every F256 (K and Jr.) platform out there today. As such, it will include a simple line editor and up/down command execution similar to Foenix MCP or a Unix shell.

Longer term, it is envisioned that nanoMon's true strength lay in its integration in an operating environment geared towards developers which will leverage a Commodore clean-room kernel written to run natively on the F256. The screen shots on pg. 5 below are from a prototype of this platform.

What's in a name; 'nano'?

Contrary to the size-obsessed '80s where something deemed full-featured became "Super" and something crowned super became "Super+", we decided to go in the other direction. MicroKernel is the default kernel on the F256 platform, and since *micromon* is already taken, nanoMon is our choice.

The general philosophy is as follows: If Commodore and others were able to more-or-less fit within the confines of an ~8K kernal ROM and an ~8K application ROM (BASIC 2.0 in their case), nanoMon should as well, thus leaving more resources and less complexity for the developer. Code requiring access to a simple kernel would therefore have all except 8K plus two 256 byte pages of high memory and two, 256 byte pages of non-zero page low memory. Ideally, part of this code will be relocatable as Jim Butterfield's Supermon+ was.

nanoMon features

For the monitor aware, some of these commands will be familiar to you; but there are new commands as well which support F256 specific features such as display stride, I/O bank selection, and VICKY registers interpretation.

Here is a short list of implemented and planned commands, followed by screen shots of a work in progress:

[l] `oad` - `.PGX` file load with optional addr override

[bl] `oad` - address agnostic data into a given address

[cl] `oad` - Commodore format load (headed by low-byte/high-byte pair)

[hl] `oad` - `.HEX` encoded file load

[s] `ave` - address agnostic data save to default device

[g] `o` - execute code at given address (also [e] `xec`). Entering `'g'` with no arguments will use the program counter address.

[m] `em` - display and interrogate memory according to a given stride mode at 'next' address if not specified

[d] `is` - disassemble from a given address, or from 'next' (program counter); currently using the Wozniak/ Baum code referenced in the article above

`str` - advance display stride mode (command will return the 3 char identifier of the new mode) which will include any of the following 8 options (see pg. 5):

> `'16B'` (default) - displays a conventional hybrid mode of hex bytes plus interrogated ASCII, 16 bytes across
>
> `'08B'` - same as above but 8 bytes across
>
> `'g4B'` - graphical 4 byte mode (displays binary byte data as tied to a gradient palette color to false greyscale (4 shades plus transparent) based on a simple algorithm. Ideal for 32 x 32 sprites
>
> `'g3B'` - same as above but suitable for 24 x 24 sprites
>
> `'g2B'` - same as above but suitable for 16 x 16 sprites or a subset of tile data
>
> `'g1B'` - same as above but for 8 x 8
>
> `'bit'` - binary mode; similar to above but data is depicted with a single byte per line with open or filled graphic `o` characters. Suitable for general binary data manipulation or live font editing
>
> `'24B'` - wide format (no ASCII displayed) - a good mode for viewing a full 256 byte page of data on just two screens and useful in the case of context sensitive highlighting (from hunt/search)

`i/o` - advance I/O bank mode (determines which page will be visible to `mem` functions aka: `reg`ister memory, `fon`t memory, `txt` memory, or `col` or memory

[h] `unt` - hunt for data (enables, or if blank, disables, context highlighting)

[f] `ill` - fill memory with a value

[r] `eg` - enhanced register display

[b] `reak` - set breakpoint address

`a` - print or set accumulator value (instantiated at `'g'`)

`x` - print or set x register value (instantiated at `'g'`)

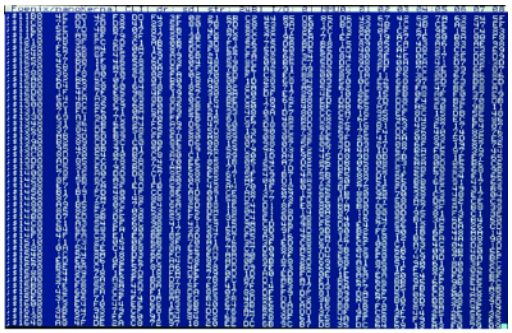
`y` - print or set y register value (instantiated at `'g'`)

`sec`, `clc`, `sed`, `cld` - set or clear select status flags (instantiated at `'g'`).

`e` [x] `it` - return to calling program

Example `mem` screen shots in various stride modes

#1: 24B mode - packs 1,560 bytes worth of data on a single 80 x 60 screen. Most useful when looking for search terms (which would be highlighted in reverse field); much ground can be covered quickly with forward and backward scrolling in this mode.



In the Apple 1 ad on the last page of this issue, you'll see marketing touting "192 memory locations displayed at once".

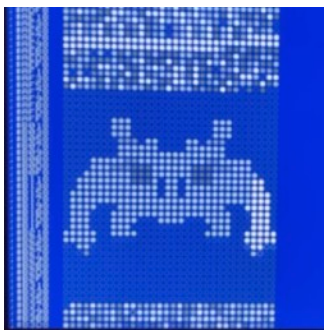
It took almost 50 years, but we finally beat it :)

#2: 16B mode - the default, contains 16 bytes left to right with ASCII characters represented. Allows full editing with hunt (search) highlighting supported.

#3: 08B mode - more targeted and closer to the famous 40-column displays of the early '80s. This is an ideal mode for double-wide/double-high text for those of us (me) that are eyesight challenged from looking at tiny text for too many years.

The "READY" prompt is ceremonial. nanoMon is written in assembly language :)

#4: g4B mode - graphics interpolation at 32-bits (4 Bytes). Assuming a gradian palette, one of 5 symbols will be rendered; a zero (transparent in the palette), 3 shades of grey, and white represent the equivalent of 2 bit+ greyscale shading.

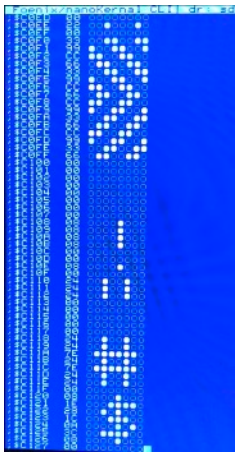


As opposed to 'bit' mode (#5), color data on the Foenix platform is described by 1 byte per pixel where each byte value indexes into a 24-bit color LUT. Visually, this would otherwise appear as random data.

This mode portrays 8-bit, 16-bit, 24-bit, and 32-bit square graphic assets (sprites and tiles) using a simple false-color mapping. The 4 variations are named based on the # of bytes as `gxB` (where *x* = '1' .. '4').

This mode, while complete as a proof of concept, is a work in progress and will depend on memory available, reduce/expand and shift functions are being considered to allow this feature to apply to bitmapped assets and to provide true-palette mapping. This will be discussed in an upcoming video on my 8-Bit Wall of Doom channel.

#5: bit mode - useful for toggling bits in data or visualizing and editing fonts (monochrome assets); this mode is similar to #4 above, however has a fixed stride of a single byte and is most applicable to bank 1 font memory located at $c000 as shown in the screen capture below.



Feel like living dangerously and editing fonts 'live'? Have at it!

Since the F256 has two font sets to choose from (and since the yet to be released nanoKernal CLI environment provides char set swapping with a single function key keystroke), you can leverage the this feature to edit using one font, and quickly toggle to view the results without the dilemma of transforming 'the' character set into symbols from the Mamluk dynasty (said another way, no Rosetta Stone is required). Since you are editing data in native form, you may use other monitor features to edit, relocate (copy), and save to disk for later loading into your own program; all without the aid of a purpose built font editor. Native tools rule!

The disassembler that led a legendary disassembler

History tells the story of how a principal of the Toronto Pet User's Group (TPUG), leveraged Woz and Baum's disassembler to form the basis of 'SuperMon'.

Of course, I'm talking about Jim Butterfield. SuperMon has been thoroughly written about and enhanced up to its final '+' form which was revamped to look and feel like the monitor built in the Commodore 128.

SuperMon+ has also been torn down, analyzed, and documented by developers, and in fact, is available in heavily commented source thanks to J.B. Langston's GitHub **here**. While it no longer resembles the simple monitor that Wozniak and Baum created, the core use of tables to pack opcode data remains in tact.

nanoMon, on the other hand, contains a direct port of the original Woz & Baum disassembler. The screen capture below (pg. 6) is rendered in F256 'double-y' character mode (80 x 30). Rather than dive into the original code in this article, we'll simply table the topic until next issue where we extend the disassemble routine to capably interpret the set of 'new' opcodes

brought forth with WDC's 65C02 in addition to providing support for added addressing modes of the original MOS 6502 opcodes.

MOS 6502 vs. WDC 65C02

The graphic to the right depicts an overlay of a modified pagetable.com image which is a color-coded MOS 6502 opcode chart placed on top of an opcode chart from the 'modern' WDC 65C02 documentation. The result is a clear picture of the new instructions.

Boxes with non-colored opcodes are 'new' instructions and will be added in the next phase of this project.

An obvious example of this is the oft used `BRA` opcode ($80) bordered in red, which is "branch unconditional" relative and takes a space in the table that was previously unpopulated.

Time and space permitting, we may visit the madness behind control-word formulation with another look at Stephen Edwards Advanced 6502 Assembly Language video. In it, he talks about patterns of instruction groups and bitwise isolation of modes and ultimately a few anomalies (made worse with the 65C02 additions!)

Disassembler porting, briefly discussed

To the right is a capture of the Woz & Baum disassembler ported to the F256 running inside of the nanoKernal CLI environment (loaded with sample code at $1000).

As we will discuss in updates to this project, one of the delivery objectives is to be able to run a subset of Commodore 64 character mode applications unmodified. There will be a long list of exceptions, but the sample code loaded is a 'first' try of sorts. (more on this endeavor and the work required next time)

Aside from a few simple formatting changes, the only significant change required to port this code to the F256 platform was to resolve the character output routine (the Apple 1 did not use standard ASCII, but it was close).

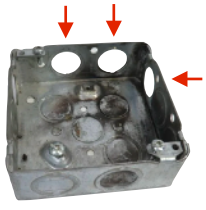
One matter that puzzled me initially was a missing subroutine in the Dobb's listing. Then it struck me, the code assumed that it was running *on* an actual Apple 1, hence, Wozmon was expected to be resident in ROM and the disassembler, a user-mode application. (Wozmon was not a kernel per se but contained the system output function similar to $F000 (`PRINT`) on the Rockwell AIM-65 (written about previously):

`CHAROUT` - existed in WozMon, however, needed `SEC` and `SBC #$80`, then a call our own `CHROUT` which manages our screen handler and character text modes.

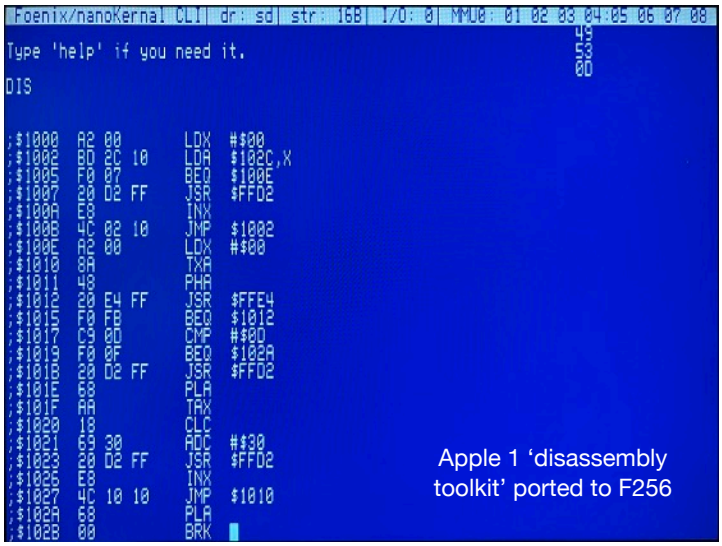
`PRBYTE` - (16 lines) existed in WozMon and contains an efficient algorithm to to print the accumulator value as two printable hexadecimal nibbles.

Sweet arcade cash: 14 x .25 = $3.50

Cost (me) = nothing



pagetable.com - Ultimate opcode reference, superimposed with the WDC 65C02 documentation opcode table



Which 'Bert' story is best?

In 1979, a kid with no quarters (me) stumbled upon a 'golden-ticket' idea. Pry as many of the coin-sized circular knockouts from *all* of Bert's electrical boxes, file off the rough edge, and head out to play Space Invaders and Atari Sprint 2 at Modell's Department Store.

It took a year for him to realize what I had done; I was helping him mix concrete in the basement when he discovered the large box full of Swiss cheese remnants. I think his words were something to the effect of "Son of a bitch"! (Bert was never short of one-liners).

Of course the scheme only worked because the coin mechs of the day did not feature magnetic or weight/ inertia reject. My next scheme involved a metal washer and a piece of fishing line. I never got that one to work.

Vintage advert time machine - despite producing only about 200 units, this very first Apple Computer ad provided the look and professionalism akin to anything Apple would create in their first 5 years of Apple II product production and beyond. The combination of Wozniak's knowhow and Jobs's vision was unstoppable; and they were just getting started!

Apple Introduces the First Low Cost Microcomputer System with a Video Terminal and 8K Bytes of RAM on a Single PC Card.

The Apple Computer. A truly complete microcomputer system on a single PC board. Based on the MOS Technology 6502 microprocessor, the Apple also has a built-in video terminal and sockets for 8K bytes of on-board RAM memory. With the addition of a keyboard and video monitor, you'll have an extremely powerful computer system that can be used for anything from developing programs to playing games or running BASIC.

Combining the computer, video terminal and dynamic memory on a single board has resulted in a large reduction in chip count, which means more reliability and lowered cost. Since the Apple comes fully assembled, tested & burned-in and has a complete power supply on-board, initial set-up is essentially "hassle free" and you can be running within minutes. At $666.66 (including 4K bytes RAM!) it opens many new possibilities for users and systems manufacturers.

You Don't Need an Expensive Teletype.

Using the built-in video terminal and keyboard interface, you avoid all the expense, noise and maintenance associated with a teletype. And the Apple video terminal is six times faster than a teletype, which means more throughput and less waiting. The Apple connects directly to a video monitor (or home TV with an inexpensive RF modulator) and displays 960 easy to read characters in 24 rows of 40 characters per line with automatic scrolling. The video display section contains its own 1K bytes of memory, so all the RAM memory is available for user programs. And the Keyboard Interface lets you use almost any ASCII-encoded keyboard.

The Apple Computer makes it possible for many people with limited budgets to step up to a video terminal as an I/O device for their computer.

No More Switches, No More Lights.

Compared to switches and LED's, a video terminal can display vast amounts of information simultaneously. The Apple video terminal can display the contents of 192 memory locations at once on the screen. And the firmware in PROMS enables you to enter, display and debug programs (all in hex) from the keyboard, rendering a front panel unnecessary. The firmware also allows your programs to print characters on the display, and since you'll be looking at letters and numbers instead of just LED's, the door is open to all kinds of alphanumeric software (i.e., Games and BASIC).

8K Bytes RAM in 16 Chips!

The Apple Computer uses the new 16-pin 4K dynamic memory chips. They are faster and take ¼ the space and power of even the low power 2102's (the memory chip that everyone else uses). That means 8K bytes in sixteen chips. It also means no more 28 amp power supplies.

The system is fully expandable to 65K via an edge connector which carries both the address and data busses, power supplies and all timing signals. All dynamic memory refreshing for both on and off-board memory is done automatically. Also, the Apple Computer can be upgraded to use the 16K chips when they become available. That's 32K bytes on-board RAM in 16 IC's—the equivalent of 256 2102's!

A Little Cassette Board That Works!

Unlike many other cassette boards on the marketplace, ours works every time. It plugs directly into the upright connector on the main board and stands only 2" tall. And since it is very fast (1500 bits per second), you can read or write 4K bytes in about 20 seconds. All timing is done in software, which results in crystal-controlled accuracy and uniformity from unit to unit.

Unlike some other cassette interfaces which require an expensive tape recorder, the Apple Cassette Interface works reliably with almost any audio-grade cassette recorder.

Software:

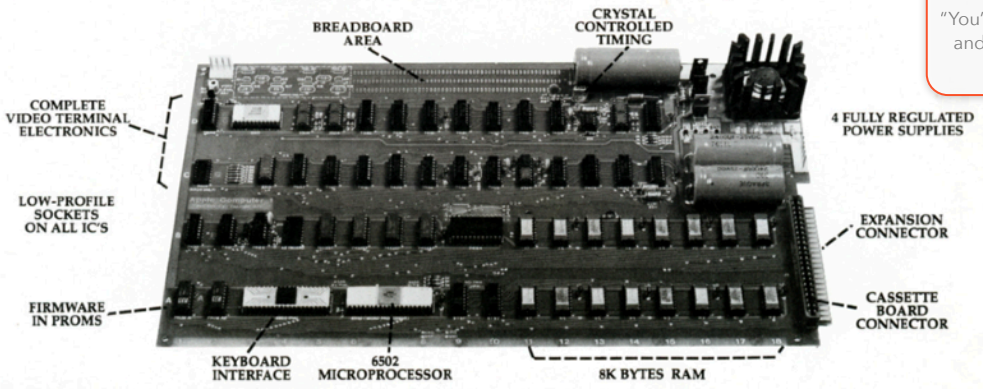
A tape of **APPLE BASIC** is included free with the Cassette Interface. Apple Basic features immediate error messages and fast execution, and lets you program in a higher level language immediately and without added cost. Also available **now** are a dis-assembler and many games, with many software packages, (including a macro assembler) in the works. And since our philosophy is to provide software for our machines free or at minimal cost, you won't be continually paying for access to this growing software library.

The Apple Computer is in stock at almost all major computer stores. (If your local computer store doesn't carry our products encourage them or write us direct). **Dealer inquiries invited.**

Key points…
'no more lights'
'192 locations displayed'
'firmware in ROM'
'enter in hex' and …
"You'll be looking at letters and numbers instead of just LEDs"

Byte into an Apple $666.66*

* includes 4K bytes RAM



APPLE Computer Company • 770 Welch Rd., Palo Alto, CA 94304 • (415) 326-4248

OCTOBER 1976 — CIRCLE NO. 7 ON INQUIRY CARD — INTERFACE AGE 11