Foenix Rising

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

Issue #4 - rev d. November / December 2022

Momentum

By the time you read this, Foenix fans world-wide will be receiving their F256 Jr. systems, joined by new A2560K owners and maybe even an early ship of a first-release GEN-X.

It has been a momentous 2nd half of 2022.

December marks my 9th month investing in research, writing and editing articles, including developing examples in support of Foenix Rising.

It began in the March/April time frame when I offered to represent Stefany's endeavor at VCF East, which led me to this Newsletter and the Foenix Marketplace.

The experience has taught me a bit about myself, specifically, the need to simplify, to focus, and to know when 'good enough' is good enough. I still have not learned my lesson.

This year-end issue debuts a new column "Intermediate Matters" and with it, introduces the F256 Jr. from an assembly language coding perspective.

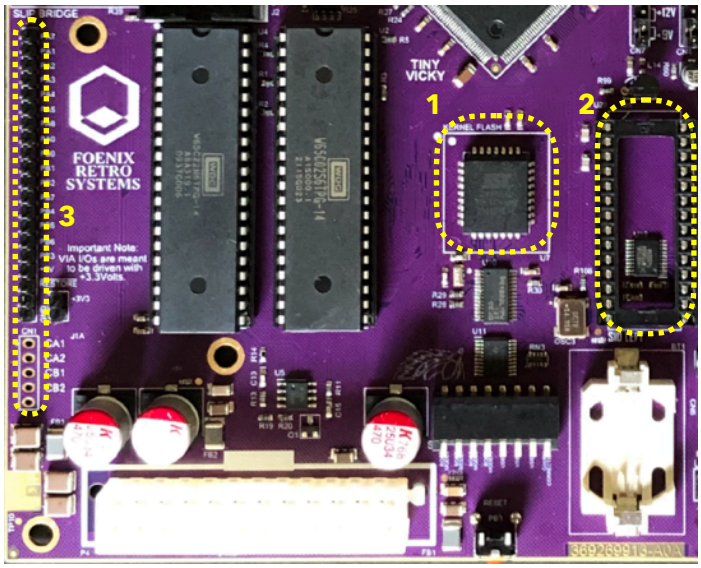
Beginning next year, we will be moving to a 'flash' format, where articles will be released more frequently, but on an individual basis. My hope is that a new format will encourage others to publish without the heavy lifting that goes into a multi-article 24 or 32 page issue.

At least that's the aim. What will actually occur depends on platform development from users like you!

Wishing you all a peaceful and productive end of year.

- EMwhite

A few of my favorite things



This is the last issue that will focus on the C256 Jr. Rev A.; not because I've fallen out-of-love; quite the opposite. It's because my F256 Rev B. has arrived. Here is a quick look at a few of the highlights.

1. FLASH memory - all Foenix machines have it, but none leverage it the way that the Jr. does. We'll focus on this aspect of the platform in an upcoming article, but the short story is any 8K block of FLASH can be banked into almost any 8K segment of the 64K memory map.
2. SID sockets - mine are populated with BackSIDs, but any SID will do. The SID is familiar, is leveraged by piles of code and tracks, and brings instant joy to girls and boys around the world.
3. The expansion possibilities offered through the 20 pin keyboard header, solder points CA1, CA2, CB1, CB2, the SLIP Bridge, and a socketed CPU for futures, are numerous.
4. The IEC connector (not shown) provides instant SD to any kernel with the means, and the Commodore legacy offers primitives for sequential files, relative files, and more.

VTOC - volume table of contents

Resources, publisher's notice, and 'fold-in' puzzle solved	2
Kernels, 'kernals', and the Foenix Jr's kernel(s) explained	3 - 6
Interview: Gadget from Discord, Foenix F256 Kernel developer	7 - 8
Retro distraction: A look at the Tandy Radio Shack (TRS-80) CoCo	9
Intermediate Matters #1: Computing large numbers and select assembly language examples to jumpstart your F256 Jr. development	10 - 25
Back Page - Vintage Advert Time Machine (Foenix F256K announced)	26

`git` and URL Resource Directory

Updated each issue, this space contains links to public Foenix related development efforts

Lib	**https://github.com/daschewie/a2560k-gcc**
Game	https://github.com/dtremblay/c256-tetris
Utility	https://github.com/dtremblay/c256-vgm-player
Game	https://github.com/dtremblay/fraggy
Utility	https://github.com/econtrerasd/playSong
Library	https://github.com/econtrerasd/VickyGraph
Kernel	https://github.com/ghackwrench/OpenKERNAL
Lang	**https://github.com/hth313/Calypsi-6502-Foenix**
Utility	https://github.com/hth313/Calypsi-Foenix-guide
Utility	**https://github.com/hth313/petit-fatfs-foenix-jr**
Lang	**https://github.com/paulscottrobson/superbasic**
Doc	**https://github.com/pweingar/C256jrManual**
Utility	**https://github.com/pweingar/FoenixMgr**
Env	**https://github.com/Trinity-11/FoenixIDE**
Utility	https://github.com/vinz6751/FoenixSamples
Env	**https://github.com/vinz6751/genxtos**
Library	**https://github.com/WartyMN/F256jr-cc65-lib**
Fonts	**https://github.com/WartyMN/Foenix-Fonts**

highlighted = mentioned this issue **bold** = newly added or updated

Links to other Foenix Resources:

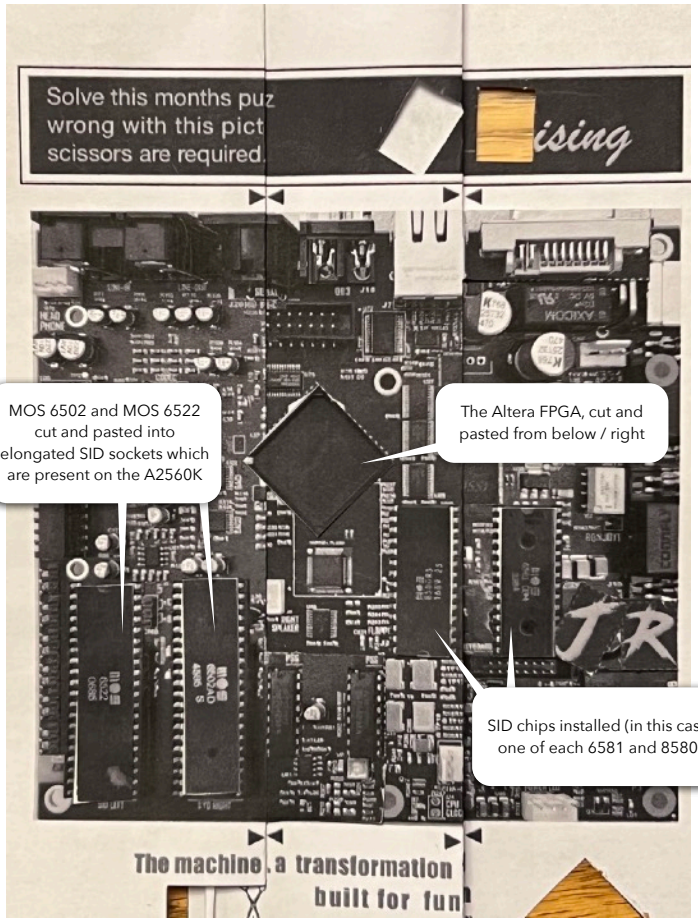
Foenix Retro Systems Home Page
Foenix Discord Invite
Stefany Allaire Patreon Page
Stefany Allaire Twitter
Foenix Marketplace content 'store'
VCF East 2022 Foenix Booth (virtual tour)

Foenix Rising is a user-supported, not-for-profit bimonthly hobbyist's newsletter published in Murray Hill, New Jersey, USA supporting Foenix Retro Systems products with a focus on software development & retro technology.

Distribution: ~2^{10}-1

Published by EMwhite (discord and elsewhere)
Motto: 'Beware of programmers with screwdrivers'

Correspondance: `commerce@emwhite.org`



The pic above looks a lot like a rough black & white rendition of a Rev. A C256 Jr., but it's not. It's the solution to the puzzle from Issue #3.

It's actually a picture of a piece of paper which was 'cut' and 'pasted', (with a scissor and tape) starting as a marked up picture of an A2560K circuit board provided on pg. 17 of issue #3.

When I was 12, I had an Uncle that resembled Uncle Ernie from 'Tommy'. He was a bad influence.

Upon each visit, my Uncle would present me and my step-brother with a stack of dodgy magazines, the titles of which, I will not mention. But I *will* mention that MAD magazine was included, and I fondly remembered the "fold-in" puzzle in the back.

Part 'B' of our puzzle is a fold-in that transforms the A2560K circuit board into a quasi-other PCB that when folded in, is transformed into a Foenix Jr.

Of course the tag-line below the picture went from "**The machine she designed for her own use, a transformation that few saw coming. Perfect fit and built for function; a development workstation that also says 'I want to play'**"... to the message you see above. Congrats to those of you that solved it!

Kernels, 'kernals', and the Jr. Foenix kernel(s) explained

What's in a name, where did they come from, and what do they want from us?

Kernel Origins

At least as far back as the late 1970s, the word 'kernel' has been associated with the **core** of an operating system. Coined alongside early versions of Bell Labs UNIX, the term has since become ubiquitous in association with Linux, Carnegie Mellon's Mach (NeXT and MacOS), and various micro-kernel architectures.

It was never 'core' as in, core memory (but on ancient systems, kernel-like code ran there). Kernels are at the center and provide *the* standard method for accessing (and sometimes sharing) system resources. All systems, regardless of OS rely on some amount of core code and for services to be as efficient and reliable as possible.

On many systems, the kernel is instantiated by a boot loader or startup sequence and spends its life (until shutdown) faithfully servicing the needs of programs. On Foenix platforms, the kernel is pulled (or banked in) from flash memory and its starting address is aligned with the processor reset vector, services are initialized, and control passed to a BASIC interpreter or a command shell. Everything in this legendary video (starring famous computer science pioneers) applies to what we do today. (gotta love the size, scale, and variety of the vintage terminals represented; and the beards)

Aboriginal operating systems booted from disk 'packs' or paper tape via "toggled-in" instructions that loaded single-minded programs; the earliest of code did not rely on kernels per se, but a handful of exec calls to accomplish the most basic of tasks: *put* and *get* from teletype (aka 'tty'), and *write* or *read* from storage.

The decades that followed the 1970s witnessed the evolution of kernels with first names of IRIX, Dynix, QNX, SCO, and dozens of others. Moore's law ushered in bigger and bigger systems, and kernels evolved further advancing memory paging, protection, and inter-process communications in shared multi-user environments.

Why all of this background? Because the kernels are coming, and because they are important. The F256 Jr. platform, while cute and entertainment-focused, can also be used as a serious machine with its MicroKernel which is complete with a network stack layer.

In the most simple sense, your relationship with one or more of the available Foenix kernels will improve your experience and productivity. You can live without a kernel (as we demonstrate in the *Intermediate Matters* column) but living **with** one will make your life easier, freeing you to focus on algorithms or creative endeavors.

Across the next two issues, we anticipate the release of a Fuzix port for Foenix and also, a modern new kernel named *MicroKernel*, designed and developed by Gadget from Discord. Stay tuned for details in the coming months and have a read of the interview with Gadget on pg. 8 below.

A short history of ROM based code and extensibility

The first affordable 8-bit microprocessor-based systems were limited. Product designers had to squeeze everything into a 32K, 48K, or at most, a 64K address space; they did not have access to fast or cost effective storage, and were therefore forced to rely on ROM chips for operating code. This meant that for 99% of the home computers shipped, the moment the three or four screws were tightened, the computer's fate was sealed.

Manufacturers such as Atari and Apple distributed systems incomplete, initially, relying on cartridge based BASIC in Atari's case, or in Apple's case, a limited Integer BASIC in ROM.

It took Atari three tries to get BASIC working; Rev. 'A' would lock-up if the user deleted a line of code that was exactly 256 bytes in length. Upon fixing this, Atari introduced a fun memory leak which added 16 bytes to a file every time it was saved, and a new problem; if a 256 char line was inserted in the screen editor, the same lockup would occur; Rev. 'C' was finally the charm.

Apple subsequently released a disk controller that made the coveted peripheral affordable, but it took a full year and then some to get there. This cleared the way for Applesoft (a vastly improved BASIC) and other languages. Of the many things Apple had going for them in these early years, Steve Wozniak's vision[1] and focus on extensibility was unmatched in this new industry.

Commodore, on the other hand, shipped the majority of their consumer systems more or less complete (but certainly, not perfect). They had a workable set of 8K ROMs and by the time the C64 was released, either could be banked in or out to allow access to the RAM below. The kernal also included access primitives to support their 'smart' peripherals[2] such as the 1541, and ultimately, the 1571 and 1581; the command set was based on the older IEEE-488 peripherals. Commodore drew heavily on the early PET series investment.

---

[1] In addition to Wozniak's Apple shipping full docs with their systems, they were also innovative by way of a peripheral card ROM and I/O space; the ability to boot via monitor from a serial port; and an inexpensive floppy controller based disk operating system (DOS). The openness of the platform fostered a vibrant market for 3rd party peripherals which literally paid for the rise of legendary vintage computer publications.

[2] Unlike Commodore's IEEE-488 parallel drives, the IEC serial protocol was excruciatingly slow due to a MOS 6522 hardware timing flaw. To compensate, they implemented *bit-banging* in software. Lots more to this story; here is an outstanding and amusing account.

Foenix owners are fortunate to not only be able to alter the footprint and contents of FLASH memory, but via Intel Quartus, can update the FPGA image, redefining hardware capabilities and features as they are released.

"You want a piece of this ?!!"

Kernels want what *most* humans want, peace-of-mind, material possessions, and to be loved; not necessarily in that order.

In a computer, this translates to consumption of just enough clock cycles to keep the house in order (negotiation with timing circuitry and peripherals), leveraging memory, sufficient to get the job done; and providing enough utility to make a programmers life easier. (on resource constrained 8-bit computers, bloat and greed is the enemy of 'good')

On 6502 systems, kernels have always appeared greedy in their consumption of zero page memory ($00 to $FF). This highly coveted block of memory is desirable due to the indirect indexed addressing modes which are only applicable here. Zero page family opcodes are generally more efficient than the addressing modes that manage data in the other 254 pages of memory (faster by ~25%; page crossing is expensive and will cost a full cycle!)

In the end, the kernel deserves this entitlement. Think about traditional IRQ code that scans the keyboard, advances cursor flash, services buffers, and updates the jiffy clock 60 times per second. Cycles add up quickly in this code loop; the kernel needs all the help it can get.

Likewise, printing a single character on the screen requires look-up tables, decision statements, paging, and more. The mundane task of scrolling even a single line of text requires moving 4K of screen memory (and the color map behind it). It's not complicated, but it costs and it needs to be efficient. The kernel's job is to provide utility and then to get out of the way and leave enough resources for 'user' programs. Ideally, this includes a large, contiguous block of RAM and free cycles per screen frame for meaningful programs (and games) that perform well.

F256 Jr. kernels - something old, something new, something borrowed, something RGB

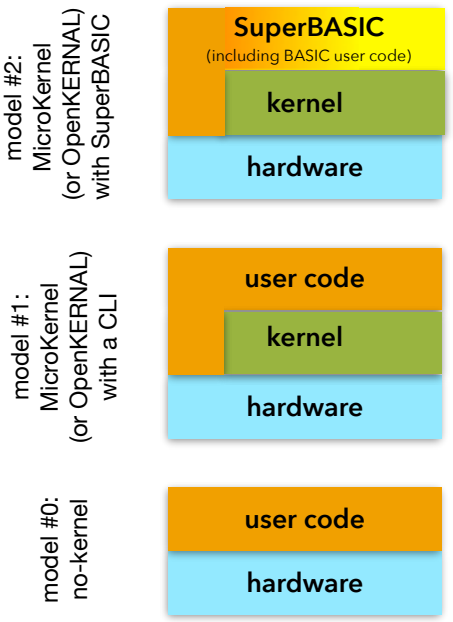
The Jr. borrows plenty from the Foenix lineage that brought us to this point. Old? Clearly, the interface tech, the fact that it will accommodate SID ICs and a 40+ year old peripheral standard (IEC) and RS-232 over DB9.

But the 'new' is the exciting part. Partly due to necessity, partly due to focusing on utility, the Jr. is new in the way that memory is dealt with, and the focus on managing blocks of 8K pages, all of which may be banked in from a generous pool of 512K of flash memory or from the onboard or expansion RAM.

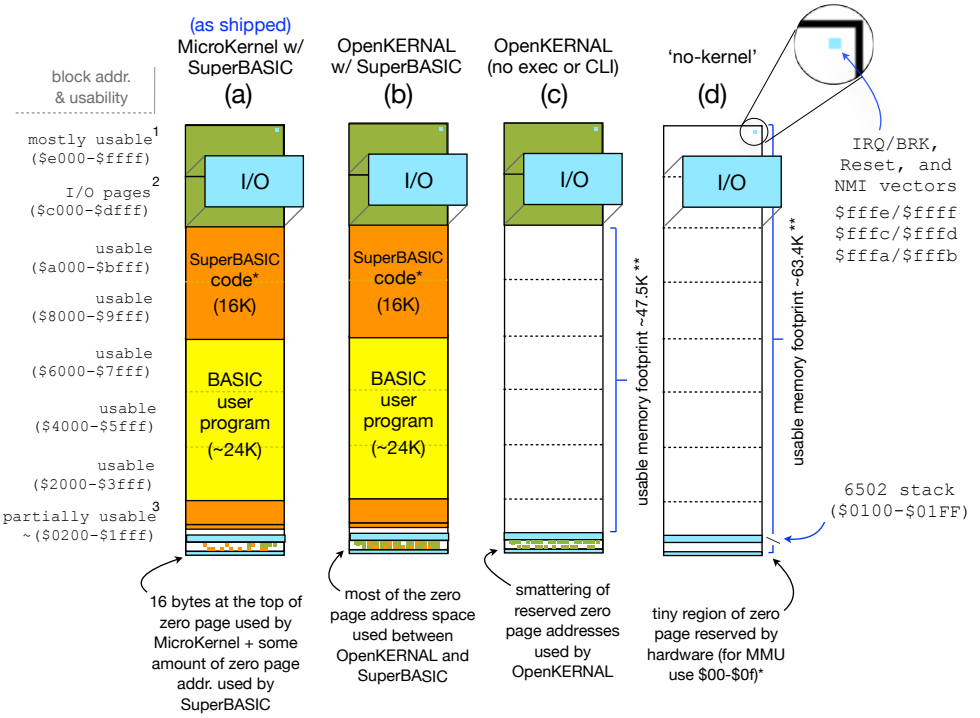
The architecture lays the groundwork for a multi-boot environment with different memory and kernel config options that may one day allow booting to a native MicroKernel **or** to OpenKERNAL **or** to a FUZIX core, **or** just a CLI from dip switches or a boot manager, user controlled. Prior Foenix systems had flash, of course, but none were opinionated in its best use.

Regardless, there are tenets inherited from the original 6502 and conventions established by '80s and in some cases, '90s systems. For the sake of discussion, consider the following memory maps, based on test builds released for comment across the prior several months.

Three models



Four memory map examples



[1] all but the NMI, RESET, IRQ/BRK vectors (6 bytes)

[2] mapped memory beneath I/O blocks is usable (but often used by kernel implementations)

[3] regardless of kernel (or in the case of memory maps a. and b. above, SuperBASIC), the 6502 stack and 16 bytes of zero page memory is reserved

* while SuperBASIC occupies a 16K footprint, its codebase is larger and dynamically extends into additional blocks of flash memory

** may be virtually extended via flash or extended SRAM banking

Initial ship brings in alpha/16 of SuperBASIC on top of MicroKernel 8-Dec-22. When turned on, the user is greeted by a lovely new splash screen, and a fancy new character set. Something like:

Three F256 Jr. memory models to discuss

The left side of the prior page introduces three memory model layouts, each of which has upper-layer user code (orange); that accesses hardware (blue); either through or without the aid of a kernel (green).

A practical example of model #1 might be an educational game written in assembly language that prints text to the screen (using kernel calls), but also accesses sprite registers directly for display and manipulation.

The SuperBASIC example (#2) depicts a BASIC interpreter and editing environment occupying 'user' space (orange), but also BASIC code (yellow) within it. It is common for integrated development environments, graphics or music editors, and productivity software to dedicate large amounts of memory to documents, songs, or in this case, BASIC language source.

Four memory map examples

The leftmost memory map (a) uses MicroKernel and SuperBASIC. This is the 'as shipped' config today. MicroKernel is a new and more modern architecture that ships with the F256 Jr. MicroKernel provides (for the first time) support for legacy Commodore devices and an IP stack which may be attached over the DB9 wired serial header or via an optional SLIP bridge WiFi interface (sold by a 3rd party; detail forthcoming). We will be discussing MicroKernel in the next issue of Foenix Rising but Gadget makes mention of it in the context of her 'dream' kernel/VM on the pages below.

The next example (b) depicts a SuperBASIC build sitting on top of the Commodore-like "OpenKERNAL". This most closely resembles a Commodore 64 and prior PET machines, with lower memory shared by a kernel and BASIC, upper memory occupied by I/O and ROM, and the space between available for user programs. This model is mid-build but prototypes have been released.

The third example (c) is identical to the second, except without SuperBASIC. This footprint is ideal for running boot-on-reset assembly language builds that kickstart Commodore-like code which depend on a kernel.

The fourth example (d) on the right is a "no-kernel" model; this is as vanilla as it gets with only 16 bytes of zero page memory, 6 bytes of 6502 upper vector memory, and the single page (256 byte) 6502 stack reserved. The Mersenne Prime discussion featured in *Intermediate Matters* is built on the no-kernel model.

Memory Map Detail

Each details a 64K footprint residing beneath a banked I/O area (four pages of I/O for managing vital system functions which sits atop of 8K of RAM). This region exists at $C000-$DFFF.

The F256 Jr.'s memory manager permits mixing and banking of the Jr.'s 512K of Flash memory, 256K (or optionally* the full 512K) of SRAM memory, and the aforementioned 8K I/O segment.

In the OpenKERNAL examples, the kernel itself reserves 2 x 8K segments of memory, one of which is the SRAM that sits below the I/O bank and the other from $E000 - $FFFF. (in the 'no-kernel' model, there is no such footprint)

In the SuperBASIC example, orange areas are occupied by SuperBASIC itself (either code or housekeeping data) and the 24K of RAM marked in yellow represents memory for BASIC code which is tokenized and stored in a highly efficient format.

The following represents other restrictions, caveats, and notes for each of the memory map examples above:

- All four memory maps are restricted from using zero page addresses $00 through $0F, reserved by Foenix hardware for banking / DMA control.
- No application may use the 6502 stack area for anything other than its intended purpose (caring for the return addresses of subroutine `JSR` / `RTS` and honoring developer push and pull actions) or else!
- The OpenKERNAL model and to a larger extent, the SuperBASIC model, use a portion of (or the majority of) precious zero page memory. This is quite similar to the way the Commodore 64 managed memory, leveraging locations $00 and $01 to control ROM and I/O banking. Between BASIC and the Commodore KERNAL, nearly all of zero page memory on the C64 was accounted for and it doesn't stop there. The original Commodore 64 BASIC V2 used the pages of memory extending most of the way up to the bottom of screen memory, located from $0400-$07E8. On the F256 Jr., SuperBASIC reserves portions of memory up through $1FFF for its own tables and housekeeping. (Worth noting that MicroKernel is much more lean in this regard. more on this in the next issue.)
- Finally, at the top of memory exists a non-negotiable set of three vectors (represented in the map diagram by tiny blue dots the upper right corner). These are little endian low byte/high byte pairs which dictate the address that will be stuffed into the program counter (PC) at CPU reset (address $FFFC / $FFFD) and vectors for non-maskable and IRQ interrupts. See this link for additional information on this topic.

A quick look at OpenKERNAL (a work in progress)

Based on the lineage of Commodore ROM based kernels, OpenKERNAL is a clean-room version for the F256 Jr. platform, with support for Commodore's IEC based peripherals, keyboard, joystick, and 6522 timers and interrupts (including support for the 20-pin CBM keyboard header). And of course, it supports the Foenix character video generator and the PS/2 keyboard as well.

OpenKERNEL does not support datasette tape devices. Most importantly, OpenKERNAL is a work in progress; but the aim is to be faithful to as many of the 39 original calls as possible. Here is a preview of what we might expect and a brief introduction of how to use it:

Call name	routine called (see github source here)	
SCINIT	jmp	scinit
IOINIT	jmp	io.ioinit
RAMTAS	jmp	ramtas
RESTOR	jmp	restor
VECTOR	jmp	vector
SETMSG	jmp	setmsg
LSTNSA	jmp	iec.lstnsa
TALKSA	jmp	iec.talksa
MEMBOT	jmp	membot
MEMTOP	jmp	memtop
SCNKEY	jmp	scnkey
SETTMO	jmp	iec.settmo
IECIN	jmp	iec.iecin
IECOUT	jmp	iec.iecout
UNTALK	jmp	iec.untalk
UNLSTN	jmp	iec.unlstn
LISTEN	jmp	iec.listen
TALK	jmp	iec.talk
READST	jmp	iec.readst
SETLFS	jmp	io.setlfs
SETNAM	jmp	io.setnam
OPEN	jmp	io.open
CLOSE	jmp	io.close
CHKIN	jmp	io.chkin
CHKOUT	jmp	io.chkout
CLRCHN	jmp	io.clrchn
CHRIN	jmp	io.chrin
CHROUT	jmp	io.chrout
LOAD	jmp	iec.load
SAVE	jmp	iec.save
SETTIM	jmp	settim
RDTIM	jmp	rdtim
STOP	jmp	keyboard.stop
GETIN	jmp	io.getin
CLALL	jmp	io.clall
UDTIM	jmp	udtim
SCREEN	jmp	screen
PLOT	jmp	plot
IOBASE	jmp	iobase

Example code #1 - "Hello A" (???)

The famous Commodore kernel example involves loading a PETSCII value ($41 or 'A') into the accumulator and calling `CHROUT` at $FFD2.

Doing so will print a character on the screen. This is the 6502 equivalent of "Hello World"* by Brian Kernighan.

```
LDA #$41     ;  65 decimal aka "A"
JSR $FFD2    ;  call CHROUT
```

But let's dig deeper!

Example code #2 - "Low budget DOS wedge"

The following 6502 code leverages kernal calls (highlighted in yellow to the left) for a *DOS disk 'wedge'* use-case. This will work on a 'real' 1541 or an SD2IEC.

For those unfamiliar with Commodore disk devices, they are *smart* devices; meaning they obey instructions received via formatted text strings across the secondary address command channel #15 as in:

"{*logical channel #*}, {*drive #, usually '8'*}, **15**"

Parameter strings begin with a command (e.g. "s" or "scratch" to erase a file), followed by a "**:**" following by arguments. The "duplicate" command, (only applicable to double drive systems) will asynchronously copy entire disks without host control other than to initiate the task.

Immediately checking the error channel will not return control to the computer until the command completes, but otherwise, the computer is free to tend to other matters while the copy is proceeding. See the single-drive Commodore 1541 manual for other commands and reference, here. Sequential files and Relative files are powerful and easy to use features that did not exist on traditional MFM drive units. More on these (potentially) in a future *Back Page* article. The legacy is interesting.

The code below calls an `INPUT` subroutine (not shown), which in-turn calls `GETIN` to accept a string of up to 36 characters then sets up and opens device 8.

A simplified BASIC version of this code might be:

```
INPUT "ENTER WEDGE COMMAND", A$
OPEN 15, 8, 15: PRINT #15, A$: CLOSE 15
```

```
WEDGE LDA #36
      JSR INPUT    ; allow up to 36 chars
      BEQ DONE     ; if no input, branch
      LDX #8       ; drive variable to open = 8
      LDA #15      ; logical file # to open
      TAY          ; secondary address
      JSR SETLFS   ; log file kernal call
      LDA CHCNTR   ; # of chars in name
      LDX #$00     ; low byte of buffer
      LDY #$02     ; high byte of buffer
      JSR SETNAM   ; name kernal call
      JSR OPEN     ; open 'file'
      LDA #15      ; logical file #15
      JSR CLOSE    ; close channel
DONE  RTS
```

* Published publicly in the 1978 "The C Programming Language", otherwise known as the "K&R book", the seminal *Hello World* example has been used as a first example in every programming language for the past 40 or 50 years. In 1972, while at Bell Labs, Brian Kernighan internally documented the same program for the BCPL language (predecessor to C).

A few words with 'Gadget' from Discord

Foenix F256 Kernel developer talks OpenKERNAL, MicroKernel, and more

In issue #1 of Foenix Rising, we interviewed Peter Weingartner, developer of the MCP kernel for the A2560K and prior work on the FMX kernel, not to mention BASIC816, video tutorials and more.

This month, we interview the designer and developer of the first pair of F256 Jr. kernels. Under the Discord ID 'Gadget', she has been collaborating with Stefany for several months, the result of which is OpenKERNAL (discussed above) and her masterpiece in progress, MicroKernel.

I sat down with Gadget over zoom and captured the dialog below. Worth noting that our discussion took place in mid-September (3 months ago as I write this) but much of what was discussed is still relevant, so we will roll with it and provide updates as matters evolve. The Rev B. prod release is shipping this week with an early version of MicroKernel and Paul Robson's SuperBASIC onboard !!

EMW: My understanding is that you created a clean room kernel for the junior based on entry point vectors of the Commodore 64. How complete is it and what is left to do?

GH: The CBM kernel is really a BIOS with a unified I/O system bolted on the side. Almost all of the calls are implemented (minus the STOP call and real time clock calls). The KERNAL provides two interfaces to the IEC bus: a BIOS level load/save interface, and an I/O level open/read/write/close channel interface.

The BIOS level load is implemented; save isn't supported *yet* but it's really just waiting for the next FPGA release. The unified I/O interface to the IEC bus is not yet implemented, and I haven't yet pulled in the RS-232 driver. But it's all coming soon.

EMW: How large will it be in total (bytes of 6502 binary) and what is the optimal location for it based on the memory management scheme of the Jr.

GH: It lives in exactly the same place that the CBM kernel lives ($E500-$FFFF). That's "out of the way" for most purposes, and as in the past, user programs are free to map it in and out as they desire.

EMW: I read a thread suggesting that your TRS-80 CoCo experience figured into the memory management design. Tell us about it.

GH: I wanted an MMU design that had enough "windows" to efficiently implement my VM, I wanted something fast (8K is a good balance between MMU LUT size and utility), something simple (instead of odd-ball windows, just divide the whole address space up evenly), and something documented (and I could provide chapter and verse for the CoCo 3's MMU). I also wanted a minimum of two LUTs so I wouldn't need to reprogram the MMU table on every interrupt. Stef gave us four, which is fantastic!

EMW: I know that you've been working with Paul Scott Robson's SuperBASIC as well; how is that moving along and in the end, do you envision the Junior experience to be a switch on to BASIC welcome screen affair, similar to the computers we grew up with?

GH: Paul's the go-to guy for SuperBASIC, but yes, I believe boot-to-BASIC is still the desired out-of-the-box experience.

EMW: Given available resources, what additional feature(s) would you like to add?

GH: OpenKERNAL will most likely be limited to IEC devices, because the only file level calls in the CBM KERNAL are LOAD and SAVE.

It may be possible to support motherboard FAT32 SDC devices in the future, but only for the LOAD and SAVE. For more advanced uses, the MicroKernel offers a more conventional set of filesystem abstractions which can directly support IEC and FAT devices.

We're planning to compile FatFS as a stand-alone application that the kernel can run when it wants to talk to the SDC on the motherboard. It's a full implementation, including partition support, but you'll need to use the MicroKernel

* What's in a name? Check this link (a Disney fandom site, for a profile)

to take full advantage of it. Here is a link: http://elm-chan.org/fsw/ff/00index_e.html

EMW: How did you first get started and how did that lead to your profession?

GH: I used an Apple II at school, then to a 16K or 32K CoCo 1 at home and eventually, a CoCo 3 running OS9 Level II.

The Motorola 6809 was a CPU ahead of its time. It had two stacks, was designed to run FORTH, it had all of these cool position independent addressing modes. The guys who built it really had a dream.

EMW: I wasn't Bill Mensch, was it?

GH: It was a different team at Motorola. Most of the team that eventually ended up at Commodore came from the 6800 group, I think.

Ultimately, the CoCo 3 came out and featured a reasonable graphics chip. It wasn't great, but it was better. What the Coco 3 *did* have was an MMU that supported an extended addressing range up to 512K.

EMW: Let's talk about life after 'retirement'. I'm using air quotes here because you don't quite seem retired to me. Considering how busy you are *in* retirement, have you ever thought about what you would do if you were not wrapped up with Foenix platforms?

GH: Heh, if I were truly retired, I'd be focused on martial arts and music. Instead, I still have this kernel/VM/OS dream that I want to release across the world's platforms :).

EMW: If you could focus on writing one piece of software for the Junior, what would it be?

GH: I have a dream of doing a MMORPG for the C64 and for the Foenix machines!

EMW: Is there any game, utility, or application for the early Commodores that you would like to port or improve upon for a Foenix platform?

GH: I wasn't a Commodore person until just a few years ago, when, having discovered that I could get my VM running in 64K on a CoCo, I ported it to the C64 hoping to have an audience. If anything, my retro-dream is still to see my VM fully up and running on these 64K platforms.

EMW: Given your experience, what part of tech would you have liked to have skipped and what new skills that you passed on, would you have liked to have learned more about?

GH: Everything 'web' has been a waste of time for me. I would like to have spent some time in the 3d MMORPG game space.

EMW: Lightning round: desert island computer?

GH: Tough! I adore my 12" MacBook, but it's difficult to truly claim ownership of a SOC x86 or ARM, so I'd like to go with a SPARC ISA machine: it's a lovely ISA.

EMW: Are you talking about Sun specifically?

GH: I don't think about computer brands, I think about how much fun are they to code on. SPARC has a really pretty assembly language. It's similar to 68K, but it's RISC.

EMW: Favorite instrument?

GH: I'd have to go with my Baldwin grand piano, though it's hard to beat the intimacy and portability of my favorite penny whistle!

EMW: And what is your favorite music; one recording or perhaps a box set that you could take with you to a deserted island for the proverbial 3-hour tour?

GH: I'd have to take the Virginia Sil'hooettes discography: fun, innocent music that I love to sing along with while I'm getting dressed in the morning.

EMW: Any favorite movies or books?

GH: No movies, but I'd take the *Ancillary Justice* trilogy (books) instead. They aren't at the very top of my favorite books list, but I think they would be more enjoyable to re-read.

EMW: Favorite food (ethnicity or specialty)?

GH: Sushi, om nom nom!!!

EMW: Ok, last one... favorite video game?

GH: Coin-op Asteroids. Otherwise, I still love to lose myself in the original Guild Wars MMORPG on modern hardware!

EMW: Thank you Gadget!!

Retro distraction: what do Isaac Asimov and Gadget have in common?

Apparently, the computer below !!

Asimov, famed Science Fiction writer, was a spokesperson for Tandy's TRS-80 line just as Captain Kirk (William Shatner) served for Commodore's paper and TV ad campaigns. Asimov and Shatner invited us to the future, and Gadget, just a kid at the time, jumped on board.

The original CoCo was accessible and affordable and with the eventual release of the CoCo 3, it was powerful; with up to 512K of memory, an MMU, and a software controlled clock speed doubler. It was also capable of running a *grown-up* operating system developed by a small Iowa based company, specifically for the MC6809; OS/9. Some of what we are about to behold on the Jr. platform is based on Gadget's early work on this TRS-80 platform.

In our discussion, she said that the CoCo, based on the Motorola reference design, had very limited graphics or sound and as a result she "lucked out"; meaning, she was compelled to focus on algorithms, memory management schemes and OS related disciplines. This equipped her for a career designing and coding embedded systems.

The 6809 was more advanced than the stalwart Z80 and industry darling 6502, but fell short of commercial success. It was available as an add-on card on a pre-*BBC Micro* from Acorn, and within an equally niche computer, the Commodore SuperPET (as the 2nd processor! Commodore would repeat this trick in the C128; neither was particularly effective).

But the biggest 6809 use-case was unexpected. It was commercial arcade, the most noteworthy of which included Williams Joust, Defender, Robotron and Konami Time Pilot and Gyruss. It was also used in dozens of Williams Pinball machines.

Will the 6809 rise from cryogenic slumber to see another day? It's looking that way. Nothing certain yet, but Stefany is already working on an FPGA core for the Jr's 40 pin CPU footprint. (see the bottom of pg. 25 for a few spy photos)

It was not uncommon for '80s celebs to endorse computer products, or technology in general. There are countless examples, some of which make little or no sense, some of which border on disturbing. But these two were good:

As happens, late night web browsing is a bad practice. I stumbled upon Dom DeLuise trying to sell me an NCR PC, the cast of M*A*S*H peddling IBM PS/2s in TV commercials, and worse. But **the worst** was "the time" Magnavox dressed Leonard Nimoy up in white John Travolta hot pants and a '70s mustache. That was it for me.

Intermediate Matters - using Binary Coded Decimal

Another look at Dr. Marvin L. De Jong's Prime Number Generator, ported to the Foenix F256 Jr.

subtitle: "1963 called and they want their largest known Mersenne prime number back"

Picking up from where Beginner's Corner left off, this column is aimed at novice programmers interested in learning assembly language beyond simple load (`LDA`/peek), store (`STA`/poke), and branch.

With an initial focus on 6502 assembly language, we will start with Dr. De Jong's somewhat low-budget code example and turn it into a big Hollywood production. (or a character-based equivalent of that)

What we will actually do is take the original published source which was developed for the Rockwell AIM-65*, dissect its workings, then bolt-on some old school features. Along the way we will discuss mathematics, optimization, and a handful of F256 Jr. hardware features. Buckle up for more vintage fun.

In issue #2, we briefly discussed a 1979 COMPUTE magazine article written by a Physics and Mathematics professor named Dr. Marvin L. De Jong.

In his original article, De Jong discussed "the advantage in speed that [MOS Technology 6502] machine language offers" versus an Apple II BASIC program in generating an extremely large number. At the time of writing, the Mersenne prime of 2^{11213} -1 was the 3rd or 4th largest known prime number; discovered in 1963 by Donald Gillies.

The next Mersenne prime would not be *found* for 8 more years (1971) when an IBM 360/91 mainframe happened upon a prime containing twice as many digits (6,002).

If you are not familiar with Mersenne and his 'primes', have a look at this brief 5 minute video; it explains.

The fact that Mersenne prime numbers are rooted in powers of 2 make them fairly easy to calculate. But notice the operative word, "calculate". We are **not** finding (proving) large primes with an 8-bit computer, we are merely ***generating*** found numbers. And we will be doing so using the 6502 BCD processor feature. We will also discuss the challenges when NOT using BCD.

There is no shortage of irony attached to this subject, the least of which is the fact that we've dug up a 40+ year old, 2 page article to write a 6 1/2 page article. Or that the CPU which De Jong based his article upon is still available, newly manufactured, for about $10 USD. It is also peculiar that Commodore Business Machines (parent of MOS Technology) started as an office supply company selling Japanese made calculators, typewriters, and furniture before acquiring MOS Technology.

Jack Tramiel's purchase of MOS was reported as an attempt to protect his ability to source calculator ICs (the market was in the midst of collapse thanks to Texas Instruments entering the business). Jack didn't know that the 6502 CPU would ignite a home computer revolution, even though Chuck Peddle was leading him in that direction.

Fun fact: As mentioned above, Donald B. Gillies (Princeton University PhD - 1953) was given credit for the discovery of 2^{11213} -1 while teaching at the University of Illinois. (he also discovered the prior two Mersenne primes). His work was so highly lauded, the University changed their postmark (below) for several years:

Why should we care? As always, we aim to connect the dots backwards, and in this case, beyond the 6502 and to a subject that might appear difficult for an 8-bit computer to comprehend, that being: large number calculation. It is an excuse to learn more about our collective and storied past, and our task at hand; to do something unexpected with a Foenix platform and leverage it as a learning vehicle.

What we will be covering:

- Overview of Dr. Marvin L. De Jong's program, including a brief description of the Rockwell AIM-65 system.
- Description of a 'like' implementation, which was ported to a Commodore SX-64 via HES-MON cartridge.
- A look at portions of: `F11213JR.BIN` - the first program published for the Foenix C256 Jr., available on the Foenix Marketplace; In this tutorial we will:

this issue:
- Use character display 'as' calc and counter memory
- Manage the Jr.'s I/O memory banking (examples for all four of the 'page 6' features)
- Use a redefined character set (PETSCII) on the Jr. (and load binary files into 64TASS) without hassle
- Write a low budget `goto x,y` kernel-like function to print text messages to a given location, sans kernel

next issue:
- Simulate old school BBS output for fun (not profit)
- Read the real-time clock and leverage VIA 6522 timers from assembly language
- Access the Jr.'s PCB mounted DIP switches - the aboriginal user interface (earn your pocket protector!)
- and more (see *closing thoughts* on pg. 25)

*At launch (~1977) the AIM-65 cost $345 and included a single-step debugging monitor; it also had a 40 char LED display, a thermal printer, and a full QWERTY keyboard. Just a year or two prior, HP was selling the HP-67 programmable mag strip calculator for $450!

A 'small' Mersenne example

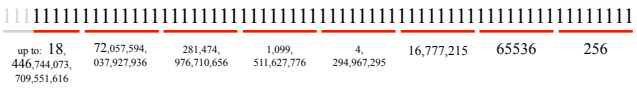
140 years ago, a Russian clergyman and mathematician named Ivan Mikheevich Pervushin discovered the 9th Mersenne prime (2^{61}-1) or: 2,305,843,009,213,693,951.

I like this one because it's ~eight 8's and pronounceable:

Two quintillion, three hundred and five quadrillion, eight hundred and forty-three trillion, nine billion, two hundred and thirteen million, six hundred and ninety-three thousand, nine hundred and fifty-one (base 10).

Something else about Mersenne primes, all bits are '1'. Think about that. Think about the fact that as far back as the 15th century and in ancient times prior, mathematicians were discovering binary patterns before they knew what binary was!

I rather like this representation of 2^{61}-1:



Q: Why do computers love Mersenne primes?

A: Because, you guessed it; they are "powers of 2 minus 1"and as such, they are easily represented in binary, or hexadecimal. They are also easy to **generate**. Let's compute 2^{61}-1. We can do it with registers, fifteen 6502 instructions, and 8 bytes of memory.

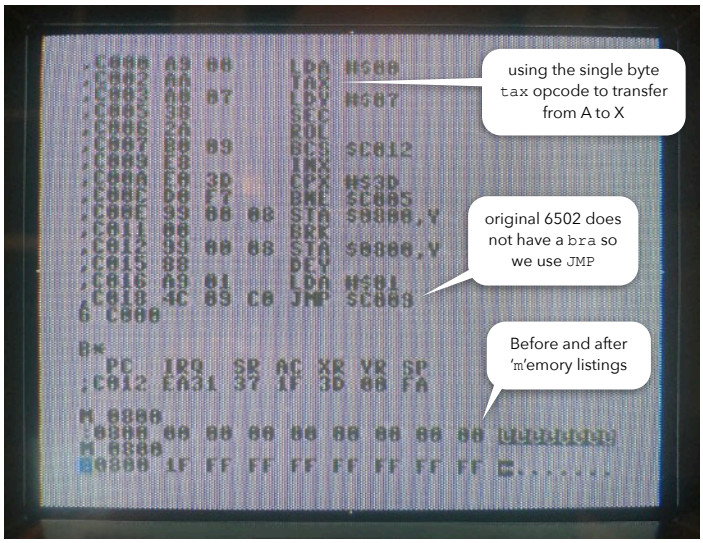
Note that this version of the program uses the *new* 65C02 instruction `bra` which branches unconditionally. It is one byte less than the 6502 equiv (`jmp`). The other nice thing about using this relative branch instruction is the code is now 100% relocatable.

How it works: After initialization, the carry flag is set and bits are rotated (right to left), from the carry into the least significant bit of the accumulator until each byte is 'full'; then a whole (full) byte, indexed by the `y` register is stored; this continues until the `x` register counter matches the value on the **`cpx`** compare on the 8th line of this program. When complete, the last byte of leftover bits is stored at position $0800.

```
start   lda #$00      ; initialize counters
        ldx #$00      ; tax will save a byte
        ldy #$07      ; done after 7 full bytes
loop    sec           ; preload carry w/1 bit
        rol a         ; rotate accum. from carry
        bcs storbyt   ; when byte is full
return  inx
        cpx #$3D      ; aka 61 as in 2^61
        bne loop
        sta $0800,y   ; store remaining bits
end     brk
storbyt sta $0800,y   ; store whole bytes
        dey
        lda #$01      ; prime the accumulator
        bra return

at 'start':  0800 00 00 00 00 00 00 00 00
at 'end':    0800 1F FF FF FF FF FF FF FF
```

Here is the same code, keyed into a Commodore SX-64 via the HES MON monitor cartridge. Note the callouts below:



iPhone photo of the mighty SX-64 and its 5" display

Next steps - method and algorithm to display 2^{61}-1

There's a video for that, sort of. Ben Eater spends 40+ minutes and writes well over 100 lines of 6502 code as part of his "Build a 6502 Computer" series. In this tutorial, he discusses the theory of operation and writes code to convert a modest (16 bit) binary number for LCD display output.

You can find other approaches in the pair of Lance Leventhal books (the original Osborne McGraw-Hill "6502 Assembly Language Programming" and "6502 Assembly Language Subroutines"). Both are out of print and available on `archive.org`.

Now let's move on to an easier way to convert a machine representation of a number to base 10 output so humans can digest them. This leads us to a major benefit of BCD and the reason why we are here.

BCD - an easier way

Generally speaking, hardware beats software. History has proven this again and again; in the mid '80s, history was being made; recording indelible examples.

It would be difficult to imagine the C64 without the VIC-II (and its sprites) or its SID chip, or the Amiga without its blitter functionality (within Agnus).

The original MOS 6502 had ~3,500 transistors but the BCD functionality was squeezed into the arithmetic logic unit without noticeably affecting the size of the die. Functionality was made available to programmers via the introduction of only two additional single byte opcodes, `SED` and `CLD`. (set / clear decimal mode)

Other microprocessors (such as the M6800 and even the original Intel 4004) had implemented BCD but not as elegantly as the MOS Technology implementation. Hence their patent US3991307A.

The famous Arcade game use case

As I was researching content for this article, I revisited Michael Steil's 6502 YouTube lecture where he dismissed BCD as "quite boring", assuming it was "for financial stuff".

Atari thought differently and in fact, used it for score-keeping on the original Atari Asteroids arcade machine, which leveraged the original MOS 6502.

```
7397: F8     SED                    ;set decimal mode
7398: 75 52  ADC ply1ScoreTens,X    ;add to players score, tens
739A: 95 52  STA ply1ScoreTens,X    ;
739C: 90 12  BCC $73B0              ;increase in thous.? no, branch
739E: B5 53  LDA ply1ScoreThous,X   ;current players score, thous.
73A0: 69 00  ADC #$00               ;add in the carry
73A2: 95 53  STA ply1ScoreThous,X   ;
73A4: 29 0F  AND #$0F               ;will be 0 if 10,000 pts. reached
73A6: D0 08  BNE $73B0              ;branch if not 10K for bonus ship
73A8: A9 B0  LDA #$B0               ;len. of time to play free ship
73AA: 85 68  STA sndTimeBonusShip   ;  sound into timer
73AC: A6 18  LDX curPlayer          ;current player
73AE: F6 57  INC ply1CurShips,X     ;award bonus ship
73B0: D8     CLD                    ;clear decimal mode
73B1: 60     RTS
```

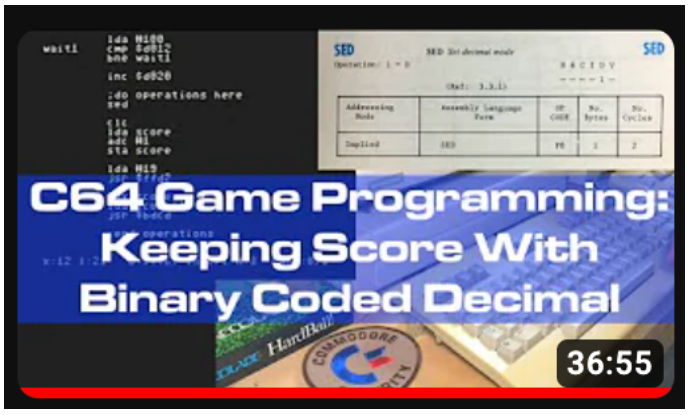
Excerpt of original source from computerarcheology.com

Fact is, as a control word actuated machine instruction, you can do no better than to use the MOS Technology implementation of BCD for real time calc and display use cases, games among them. For this type of use case (and considering the alternative; writing 100 lines of assembly), this might be considered the equivalent of having an onboard FPU in the pre-Pentium Intel CPUs.

An interesting footnote to this topic; I exchanged a few rounds of mail with Bill Mensch and he shared that due to the tightness of timing, Atari could not use the [then, new] WDC 65C02 since math operations took an additional cycle which threw off the Asteroids timing loop. This was remedied in the 65C816 and represented the 3rd time that Bill redesigned the implementation.

Another game related use case

Here is a score keeping related tutorial that is very well done from Robin of "8-Bit Show And Tell". It is worth the 37 minute view (click it to open).



8-Bit Show And Tell is one of my favorite retro resources. With nearly 50,000 subscribers and over 5 million views, Robin delivers educational content, expertly edited, without fanfare

Enter Dr. De Jong and his BCD interests

Before the internet and before BBSes, electronics magazines marketed SBCs (single board computers) and kit computers to the technical and academic community. This population of pioneers was significantly smaller than "the masses" which would ultimately consume home computers by the tens of millions.

Marvin De Jong was an educator by day but also published books and papers on topics such as "Chaos and the Simple Pendulum" and "Mathematica For Calculus-based Physics". And like many in the scientific community, he was a computer hobbyist as well, publishing a handful of books on Apple II and Commodore 64 Assembly Language.

But he started with single board computers including the KIM-1 and the Rockwell AIM-65 and published what I consider **the** 6502 compendium entitled "Programming and Interfacing the 6502 With Experiments". It is long out of print but you can have a read on archive.org here.

As discussed in Issue #2 (see pg. 32) of Foenix Rising, De Jong also published several COMPUTE magazine articles including a 4-page floating point to BCD conversion piece that might be worth a look.

What does De Jong's Prime Number program do

Let's start with what it does not do; It does not take any 'input' whatsoever; instead, it uses hard coded 'stop values' encoded in a set of nested decision statements which compare (`CMP`) for immediate mode values of #$01, #$12, and #$13. This represents the exponent of 2^{11213}.

While it's processing, the original program spins quietly for 12 minutes or so (at 1 MHz), then fireworks! Actually, there are no fireworks, but there is output in the form of ASCII characters beginning with dozens of zeros (ASCII "0"), followed by the the most significant digits of 2,814 onwards. 3,376 digits later, it's done and a `BRK` instruction drops control back to the monitor.

We'll talk about De Jong's Rockwell AIM-65 on the next page or two, but suffice to say, it was a primitive machine with a 20 character display and a 20 character wide thermal (5 x 7 character matrix) printer.

The print routine (which ultimately calls the ROM based kernel output routine at $F000) is 29 of the total 85 program instructions and the initialization code is 22 lines long; leaving a mere 34 lines to generate the prime.

The program listing as published is heavily commented including symbolic labels for branch and jump targets. There is no memory map per se, but the 1/2 page article that accompanies the code does a fair job introducing the memory requirements and variables.

I had a go at entering and assembling the listing with 64TASS for the C256 Foenix Jr. Rev. A dev board and had a number of problems starting with the fact that the printed version of the source code did not differentiate between immediate mode and zero page addressing.

References such as `CMP $10` actually meant to identify an immediate compare of `#$10`, elsewhere, a load of $01 as in `LDA $01` actually meant zero page $01.

Luckily, the assembly byte-code output was included in the left margin of the article so I was able to look up the opcodes and ascertain the addressing modes from there.

An additional challenge was the source computer system (the Rockwell AIM-65) afforded the use of all zero page addresses; not so, for the Foenix; (addresses range $00 through $0F) this range is reserved for memory banking and DMA, so this had to be relocated.

A final challenge had to do with 'me' and my particular Jr. board which I damaged while inserting one of the power supplies that I was testing. This resulted in a number of the legs of the FPGA pulling off the board which I ultimately fixed, but in doing so, created some amount of instability that manifests as component temperatures change. Again, this was caused by me when I inadvertently flexed the board. But it still works.

Challenges aside, I prevailed, improving my narrow pitch surface mount soldering skills along the way.

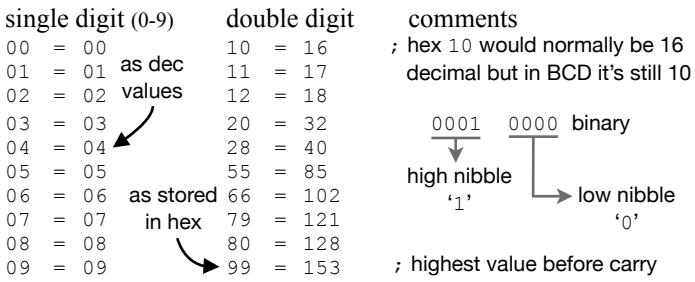
Memory use and 'the' calculation

De Jong mentions allocating $0400 to $0FFF to hold "the number". After zeroing, computation begins. The seed value of 1 is stored in location $0400 and is doubled, carrying forward for every double digit BCD byte across 3K (or 3072 bytes) of 'calc memory'.

As 1 turns to 2, then to 4, and 8 and passes 10 on its way to 16 decimal, the value of the byte is adjusted by the BCD microcode. 16 hex would normally represent 22!

But armed with the understanding that each nibble (4 bits) of the byte can only represent values (digits) 0 through 9, matters improve. You will always (only) see decimal digits 0 through 9 and while in decimal mode, the CPU treats addition and subtraction as such.

To illustrate here are several examples:



What to know about math in BCD mode: There is actually nothing to know. `ADC` (add with carry) and `SBC` (subtract with carry) simply work. The CPU's ALU performs the math, adjusting as you might expect for base 10. Printing them to the screen will requires the use of `LSR` bit shifts and a logical `AND`. (more on this later)

De Jong's source code with callouts

Inner loop - traverses each page of the 3,072 byte buffer (can represents up to or 6,144 digits)
Outer loop - tracked with the LO, MID, and HI counters; counts from 1 to 11,213

```
$0208  A9 00   START   LDA $00        Load pointers to number table.
020A  85 04           STA TABLE
020C  A9 04           LDA $04
020E  85 05           STA TABLE+1
0210  A0 00           LDY $00         Initialize Y index to zero to
0212  A9 00   NEXT    LDA $00         clear all table locations to zero.
0214  91 04   LOOP    STA (TABLE),Y   Put zero in each table location.
0216  C8             INY             Increment Y to fill page with zeros.
0217  D0 FB           BNE LOOP
0219  E6 05           INC TABLE+1     Go to the next page in the table.
021B  A5 05           LDA TABLE+1     Are all the pages completed?
021D  C9 10           CMP $10
021F  90 F1           BCC NEXT        No. Then fill another page.
0221  A9 04           LDA $04         Yes.  Reset pointers to the base
0223  85 05           STA TABLE+1     address of the table.
0225  F8             SED             All subsequent additions will be
0226  A9 01           LDA $01         in decimal.
0228  8D 00 04        STA TABLO       Start with one in lowest digit of
022B  A9 00           LDA $00         the table.
022D  85 00           STA LO          Initialize the addition counter to
022F  85 01           STA MID         zero;  three locations ($00,$01,$02)
0231  85 02           STA HI          in page zero.
0233  18      COUNT  CLC             Clear carry for additions.
0234  A9 01           LDA $01         Increment the addition counter,
0236  65 00           ADC LO          LO, MID, and HI each time the number
0238  85 00           STA LO          is added to itself.
023A  A5 01           LDA MID         Carry from LO addition into MID.
023C  69 00           ADC $00
023E  85 01           STA MID         Result into MID.
0240  A5 02           LDA HI          Carry from MID addition into HI.
0242  69 00           ADC $00
0244  85 02           STA HI          Result into HI.
0246  18             CLC             Clear carry for adding THE NUMBER.
0247  B1 04   PAGAD   LDA (TABLE),Y   Get a piece of THE NUMBER.
0249  71 04           ADC (TABLE),Y   Add it to itself.
024B  91 04           STA (TABLE),Y   Store THE NUMBER.
024D  C8              INY             Increment Y to repeat the addition
024E  D0 F7           BNE PAGAD       for an entire page of memory.
0250  E6 05           INC TABLE+1     Increment the page number.
0252  08             PHP             Store P to keep track of any carry.
0253  A5 05           LDA TABLE+1     Have we finished adding the entire
0255  C9 10           CMP $10         table?
0257  B0 04           BCS DOWN        Yes.  Then check to see if we have
0259  28             PLP             added enough times.  No.  Add more.
025A  4C 47 02        JMP PAGAD

025D  A9 04   DOWN    LDA $04         Reset table pointer.
025F  85 05           STA TABLE+1
0261  A5 00           LDA LO          Check add counter.  Is it equal to
0263  C9 13           CMP $13         011213?
0265  D0 CC           BNE COUNT
0267  A5 01           LDA MID
0269  C9 12           CMP $12
026B  D0 C6           BNE COUNT
026D  A5 02           LDA HI
026F  C9 01           CMP $01
0271  90 C0           BCC COUNT
0273  18              CLC             Subtract one from 2^11213 to get
0274  AD 00 04        LDA TABLO       THE PRIME NUMBER.
0277  E9 00           SBC $00
0279  8D 00 04        STA TABLO
027C  A9 0F           LDA $0F         Point to the top of the table to
027E  85 05           STA TABLE+1     read THE PRIME NUMBER out from
0280  A0 FF   UP      LDY $FF         the most-significant digit to the
0282  B1 04   THERE   LDA (TABLE),Y   least-significant digit.
0284  A2 FE           LDX $FE         Convert the BCD digits to ASCII.
0286  48              PHA             Save two digits on the stack.
0287  4A              LSR A           Get the most-significant nibble.
0288  4A              LSR A           Move it into the low-order nibble.
0289  4A              LSR A
028A  4A              LSR A
028B  18      HERE    CLC
028C  69 30           ADC $30         Here we have an ASCII digit so
028E  20 00 F0        JSR PRINT       jump to the output routine.
0291  E8              INX             Do we need to get another digit?
0292  F0 06           BEQ AHED        No.
0294  68              PLA             Yes.  Get the digits.
0295  29 0F           AND $0F         Mask the high-order nibble.
0297  4C 8B 02        JMP HERE        Convert it to ASCII.
029A  88      AHED    DEY             Get some more of the number from
029B  C0 FF           CPY $FF         the same page.
029D  D0 E3           BNE THERE
029F  C6 05           DEC TABLE+1     Change pages.
02A1  A5 05           LDA TABLE+1
02A3  C9 04           CMP $04
02A5  B0 D9           BCS UP
02A7  00              BRK             Back to the monitor.
```

This is the test / reset portion of the outer loop whose job is to reset the page table pointer and look for the combination of low, medium, and high values, aka the desired exponent

Output loop

Doubling code (3 instructions which load, add, store); executed once for each 3072 byte iteration, 11,213 times; `TABLE` is the low-byte/high-byte page pointer ($04, $05 indirect indexed by the Y register); this starts the PAGAD inner loop

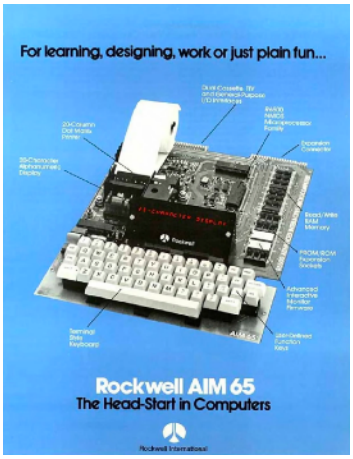
Quick detour: Rockwell's AIM-65

In 1975, MOS Technology released the KIM-1 as a development board alongside their 6502 processor. The name 'KIM' was catchy, but in fact, was an acronym for the killer feature: keyboard-input-monitor.

Rather than rely on front panel switches (as the IMSAI 8080 & ALTAIR 8800 did a year or so prior), the KIM-1 boasted six 7-segment LEDs and a hexadecimal keypad along with a ROM based machine language monitor. Steve Wozniak reportedly used the KIM for his earliest work and published a few Dr. Dobbs Journal articles for the early 6502 systems, the KIM-1 among them.

As innovative and useful as the KIM-1 was, the addition of a terminal was important; there was only so much one could accomplish keying in hex digits and interfacing (as a human) with a 6 digit display (though Jim Butterfield and others published a pile of games and amusements in the First Book of KIM).

Enter Rockwell. Not only did they 2nd source and later enhance the original 6502, but they released a vastly improved development platform, the AIM-65, and it had a strong following.



As sold (sans case)

(check this link for issue number 2 of the official Rockwell AIM 65 newsletter, "Interactive")

Advantages over the KIM-1 included a 40 character, 16 segment LED display, a line editor oriented machine language monitor, and a 40 character (though small) thermal printer. It also boasted a full ASCII keyboard and had 5 ROM sockets (three were free for user ROMs) which could host languages such as BASIC, FORTH, and Pascal.


The AIM 65's amazing 16-segment LED could display all 64 ASCII characters

Of course at some point, cross-compilers became a necessity for serious development; but there is nothing like an integrated keyboard and apps in ROM (akin to *flash* in Foenix systems) for developing "on platform".

Ethan Dicks (Commodore and maker enthusiast) interfaced a 1541 drive to his AIM 65 !! Have a look at his 6502 tribute, from VCF East 2020 (virtual).

The centre for computing history has an excellent collection of Rockwell AIM 65 artifacts and info including an index of magazine articles.

Commodore SX-64 version of De Jong's original code

In this section, we'll discuss the steps required in order to re-platform the original code to another 6502 system. De Jong's original article mentions: "Owners of other systems can simply use their own output subroutine". Unfortunately, it took a bit more work than *just* changing the output routine.

In fairness, the additional work was to accommodate futures that De Jong would not have seen coming. At the time of his writing, 6502 systems were extremely primitive and it was not uncommon to 'own' zero page and therefore have little to navigate around.

Above, we touched on the need to relocate a some of the zero page address variables in order to accommodate the Foenix Jr. (reserved locations $00 through $0F). This also applies to vintage Commodore platforms. The C64 reserves $00 for the 6510 data direction register and address $01 for LORAM, HIRAM, and CHAREN banking, not to mention some cassette control lines.

To remedy this (at least for the SX-64 test) we used zero page locations of $60, $61, and $62 for `LO`, `MID`, and `HI` (originally $00, $01, and $02).

But we were able to leave the `TABLE` low-byte/high-byte indirect pointer at $04 and $05 and in fact, left the calc buffer at $0400 - $0FFF.

On the SX-64, $0400 is the start of screen memory and continues through a small unused section, through sprite shape pointers, and into BASIC program memory. We own this space!

Finally, we relocated the program itself to $1808 (which is $1600 higher than the starting location of the original AIM-65 program, which started at $0208. We do this for two reasons; first, it is important to get our code up and out of the way of [still, other] addresses required for this machines interrupt handler and also away from the screen buffer; secondly, since I entered and modified this code in a rudimentary ML monitor; I needed to know exactly which address I was changing when I made the last group of modifications (below), and it was easier to track with an offset high-byte and a matching low-byte.

The final modifications included one `JSR`, two `JMP`s, and three `LDA` and `STA` *absolute* address references. The 'absolute' addressing mode is as it sounds; the address is explicit and final. It is not subject to indexing by a register and regardless of the location in memory, it will still point to the same address. As such, we had to manually patch these 3 byte instructions after the code was keyed in.

Due to the amount of screen real estate that this gen Commodore platform offered (40 x 25 or 1,000 bytes), I was not interested in running the display output portion of the program, just the calc portion. My aim was to a) get it working and b) record the run time in minutes and seconds which was 12 minutes and 40 seconds (close to De Jong's quoted 11 minute run time for 2^{11000}. I should mention that I did not disable the IRQ interrupt.

Overview of `F11213JR.BIN`. Details, lots.

'F' is for Foenix and 11213 is our 1963 prime as in 2^{11213}. This .bin file is on the Foenix Marketplace today but you'll need to insure that your system does not boot to kernel.

The pic below is a PET look/feel which, for best results (my opinion) leverages the 2K PETSCII PET font; a black screen and a green (phosphor, of course) text foreground complete the look. (see item 'f.' below)

Execution begins with the start address pushed into the boot vector of $FFFC (and $FFFD), and some housekeeping is in order. A color palette is initialized and the screen is cleared with colors plugged in (more on this below), then the DIP switch selectable font is loaded into page 3, and the glorious descriptive text is drawn on the screen.

Each text string is written to `x` and `y` coordinates as defined in the .byte definition itself. Remember, we do not have a kernel here so had to write the slimmest version of one.

In this application, the "byte counter" ending values are plugged in based on the setting of DIP switch 7.

The '@' sign, '0', and 'S' representing high (`HI`), mid (`MID`), and low (`LO`) value variables are updated in real time as the calc memory proceeds.

The upper portion of the screen **is** calc memory which begins with a $01 in location $C000 (the Foenix Jr.'s screen memory) versus $0400 on the AIM and Commodore 64 machines.

The field of green '@' signs signifies that memory is initialized at zero. If you've been around ASCII for a while, you'll know that the '@' symbol is $40 hex (64 decimal) which is just below the alpha range. In VICKY (as with Commodore prior), values repeat such that writing a zero to screen memory produces an '@' and writing a 1 yields an 'A'.

Program run has 3 distinct phases; a) initialization which was briefly described above and completes in a fraction of a second; b) calculation, which we will talk about shortly; and c) output, which displays 'the number' in the output buffer window which is conveniently 256 characters in size.

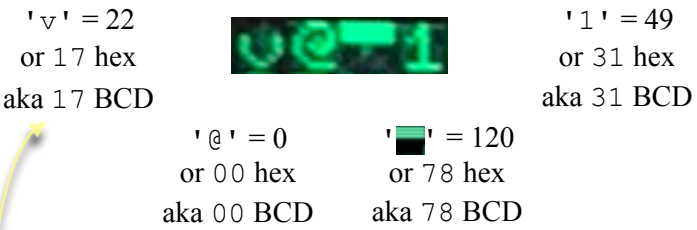
We will discuss more about initialization when we look at the actual code below, but let's start with by discussing calc memory, herein referred to as "The Matrix". (sorry)

What's in The Matrix?

More than meets the eye, but not this much:

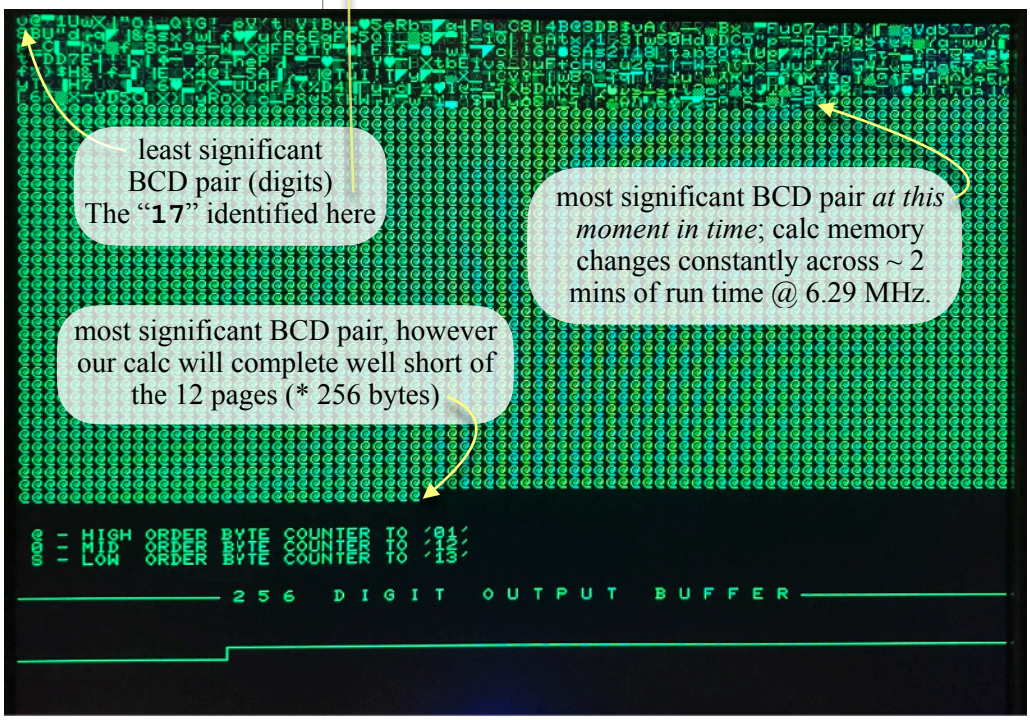
What you are looking at is the equivalent of phosphor glow on a CRT or in this case, an iPhone catching a DVI-I display fading, more irony. Very few of those characters actually exist in the wild.

Let's look closely at the first four characters (see screenshot below). The punchline is the first 8 (least significant) digits are packed into 4 BCD bytes as 17007831. Oriented correctly (if this were the entire number), the answer would be 31,780,017. (not a prime)



Clear as mud? Remember, in De Jong's algorithm above, he begins with an 'a' or a `01` in the first memory location, then doubles it. Upon each doubling, he adds with carry, which spills to the next (left) nibble then to the byte to the right and this math is carried on without regard to the size of the number in memory.

Upon each cycle of 12 pages of calculations, `LO` is incremented until it turns from `99` (BCD) to `100` and is carried to `MID`, etc. This continues until `01 12 13`.



Displaying the results

At the bottom of his COMPUTE article, De Jong cautions: "*P.S. A lot of leading zeros get printed before the number starts*".

Said another way, for each unchanged '@' sign, two '0' numerals will be printed, beginning with the most significant potential numeral, backwards to the first non-zero digit. At this point, the large prime begins printing.

In the original Rockwell example, this must have been annoying (I'll have to ask Ethan).

In our program however, we suppress this by means of self-modifying code (more on this next time), but this is the magic moment by which we print out the prime that our adoring public has been waiting for, so we take our time getting there.

I thought it would be fun to pile vintage on top of vintage and leverage something akin to 1200 baud output. I want the user to appreciate the number, or at least catch the first few digits of this truly massive number.

Once the number is fully rendered (anywhere from a single digit of 2^2 -1 or 3, to 1971's crown jewel, 2^{19937} -1 which is 6,002 digits long, you'll get a special surprise (decade appropriate). In the next issue, we'll discuss how we did it.

The remainder of this article will focus on code and coding in general. Each of the twelve examples labeled ('a' .. 'l') will discuss one aspect of 6502 development, a new or interesting Foenix Jr. feature, or both. Next issue, we will finish with the full program listing.

Thank you to **Gadget** for helping me troubleshoot some of the issues I was having with my Jr. (after I damaged it), including providing the code samples that kickstarted this project. Also to **Dr. Marvin L. De Jong** (for his original magazine articles, his books, and most of all, for his dedication to math and science education).

---

a. Using display memory as variable or data storage

It's difficult to peer into memory and view data as it changes; in fact, it can be become a burden or even a full time job. In modern IT, analytics is a specialty in itself; telemetry services, observability, logging, and log pattern searching requires specialized skills and tools.

In our 8-bit world, matters are simple but viewing changing values across time brings unique challenges because facilities to log remotely (or to disk) are scarce and at any moment, the system is subject to locking up, vaporizing precious state data during debugging.

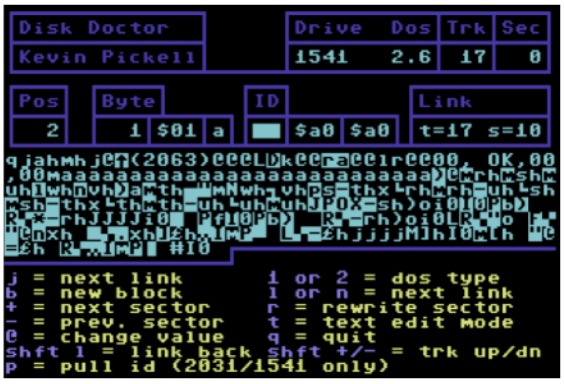
Using display memory to store data, changes this. Screen memory is, after all, just plain memory; It just happens to be mapped into video circuit (VICKY's) view.

Considering its 320 x 240 bitmapped display, the Jr. might seem limited, but it boasts multiple text modes including a large 80 x 60 screen. This is greater than 4x the size of early '80s machines and in this program, we use it for the visualizing calc memory, watching variables increment, and displaying the calculated number.

Screen memory is mapped from $C000 to $D2BF on I/O page 2 and the upper left corner (coordinate 0, 0 or the 0th column and 0th row) is the first byte of this memory range. It is important to note that the byte value to font mapping may not produce visually useful data. This is 100% dependent on the font in use. Contrary to traditional application development, using screen memory to display data requires nothing more than locating variables and buffers into this range; there are no print statements or kernel output routines.

We will touch on fonts below, but it's worth mentioning that Commodore fonts are well suited for this kind of work since they sport a high number of printable characters. (to be discussed next issue)

In the early '80s, several utility program used screen memory and line draw characters effectively. Here are two early examples:



Ex 1. Kevin Pickell's "Disk Doctor" utility

Track and Sector editors such as "Disk Doctor" were popular tools-of-trade in the '80s. Notice the 256 byte buffer in the center of the screen.

The first byte of the sector, a 'q' and the second, a 'j', corresponds to the link to track 17 and sector 10. Using Disk Doctor, you could live-edit text, modify the starting address of programs, access hidden regions of disk. Handy !!

Disk Doctor was unique because it allowed free text (or numeric value) editing within the displayed block (the cyan region above). I remember following the sector links from the 1541 disk directory (starting on track 18, sector 1) into program binaries and editing text messages, investigating and defeating copy protection schemes, and learning how block allocation maps worked and how file types (PRG, SEQ, etc.) were encoded. Disk Doctor-like tools were indispensable.

Ex 2. Michael Weitman's "M-Term" terminal emulator

This screenshot shows a Punter Protocol file transfer in progress. A filename called "it" (of course !!) is 6 'good' blocks into a transfer.

In this example, the current block is read from device #2 (the modem) and stored directly to screen memory before being committed to disk.

The mild entertainment of the visual helps distract from the snails-pace 300 baud transfer rate.

---

b. Managing the Jr.'s I/O memory banking

The modern WDC 65C02 processor used in the Jr. has many advantages over its MOS 6502 ancestor. But one thing that has not changed is the addressing scheme. It still uses 16 address lines (pins 9 - 25) and thus can only access 64K of memory. Great for 1982, not for 2022.

VICKY's MMU to the rescue; Through a set of zero page addresses, four mapping lookup tables provide a high performance and flexible scheme allowing 8K pages to bank in from SRAM, flash, and I/O.

Today, we will discuss I/O, which on the Jr. is split across four 8K banks (all of which attach at $C000) as follows:

I/O Bank 0 (`%00000000`): Devices including the stereo SID sockets, FPGA based PSG cores, the CODEC, UART (serial or SLIP port), timers, DIP switches, screen color lookup tables and other functions.

I/O Bank 1 (`%00000001`): Font memory (the base for custom characters) and graphics color LUTs

I/O Bank 2 (`%00000010`): Text display char memory

I/O Bank 3 (`%00000011`): Text display color memory (index into foreground/background LUTs).

To use: Upon write to `MMU_IO_CTRL` ($01), the selected 8K bank is attached at $C000-$DFFF. For example:

```
LDA #$02    ; select "page 2" text display
STA $01     ; store in register
LDA #$08    ; load 'h' into accumulator
STA $C000   ; store to screen memory 0, 0
LDA #$09    ; load 'i' into accumulator
STA $C001   ; store to screen memory 1, 0
```

This example enables text display memory and writes 'hi' in the first two positions. We will examine a more full fledged printing example in detail, below.

c. Controlling devices & resources from your own programs (addressing hardware directly)

If you read the "Kernel" article on pg. 3 or if you are already familiar with this topic, you'll know a kernel is *user callable code that controls resources of a computer.*

Since we are focusing on the F256 Jr., the resources in question include the usual suspects: the keyboard, character display, and probably disk. But on the Jr., it might also include the serial port or optional WiFi.

Then there is another class of features and devices that begins with joysticks and gaming controllers and moves on through audio, graphics, and 6522 timing circuits.

And we would be remiss if we did not mention the RTC (real-time clock circuit), the CODEC (D/A, converter, mixer), and other features and addressable components.

Most of these devices are not kernel managed at all, but directly addressed at documented memory locations, otherwise referred to as registers.

In the old days, I/O devices and custom chips would be hard-wired through TTL logic ICs to occupy an address range but on Foenix systems, the FPGA arbitrates between buses and devices (some virtual, some physical), solving for differences in clocking and negotiating streaming from serial to parallel and back.

The following assembly language code will instruct a SID chip inserted in the left socket to play a simple bell tone*. The **highlighted** lines represent registers; consult the F256 Jr. manual for a full map.

```
PLAYBEL LDA #$00
        TAX
LOOP1   STA $D400,X
        INX
        CPX #$17   ; init most of the regs
        BNE LOOP1
        LDA #$32   ; ~5th octave G
        STA $D401  ;   stuffed in freq hi byte
        LDA #$69
        STA $D405  ; attack / decay
        LDA #$8A
        STA $D406  ; sustain / release
        LDA #$4C
        STA $D418  ; hi-pass / volume (12)
        LDA #$60
        STA $D416  ; filt cutoff (high)
        LDA #$11
        STA $D417  ; resonance control reg
        LDA #$11
        STA $D404  ; triangle / gate-on ("key on")

        LDY #$00   ; useless delay loop...
        LDX #$00   ;   useless because it merely
LOOP2   INX        ;   wastes cycles, iterating
        BNE LOOP2  ;   32 * 256 = 8192 times
        INY
        CPY #$20   ; at varied clock speeds
        BNE LOOP2  ; this will no longer work

        LDA #$10
        STA $D404  ; "key off"
        RTS
```

*pre-req: initialized CODEC, physical 6581/8580 SID or a clone in the left socket, and I/O bank 0 selected via `MMU_IO_CTRL`

d. Character output, a how-to (and what for) guide

In item ‘a.’ above, we discussed the benefit of using screen memory as variable and buffer storage to assist in debugging during development, for utility/functionality, and/or for mild entertainment purposes.

This section, which admittedly, is a bit longer than originally planned, discusses the ins, outs, art, and history of displaying text on a screen.

The basics of printing to screen requires two, or optionally, three pieces of information:

1. What character to print?
2. Where on the screen will the character be written?
3. How do attributes need to be altered in coordination with writing to character memory (if at all).

The “**what**” is obvious; in the most simplistic example, we are talking about a single character within the traditional printable alphanumeric range including special symbols such as ‘!’, ‘#’, and ‘$’. Depending on implementation, this might also include extended characters between values 128 and 255. In this range you are likely to find line draw characters, smily faces, or something obscure that somebody felt was important at the time.

The “**where**” touches upon the concept of a logical ‘cursor’ also known as the insertion point. We are not necessarily talking about the physical cursor character which might appear as a thick underscore (underbar) or a reverse field block; these are artifacts of '70s and ‘80s terminals; but the same concept applies. We are talking about the screen location where the next character will be printed.

In older terminals, the physical/visible cursor was always on, whether displaying characters on the screen or while waiting for input; the nostalgia of watching characters render from left-to-right and top-to-bottom led by a cursor at moderate speed is heartwarming. On modern machines (including Foenix), the screen is painted so quickly, text appears magically and of course, does not leverage a cursor. Cursors are, of course, important for input. We will discuss input, physical cursors, and something called *curses* in part II of this article. Today, we are focusing on output and printing.

Terminals are beginning to feel foreign to modern computers; but just like the IBM Mainframes that I last touched in my college years, they have not gone away.

Physical CRT terminals may be gone, but there are several examples of how terminal and serial technology is still relevant. Here are a few examples:

In a cloud paradigm (Amazon Web Services), a virtual web based terminal is spawned to connect to your Linux EC2 or Lightsail instance. The shell is still the sys admin’s home. Yes, it is tunneled through SSL via a window in a Chrome browser, but the Linux instance on the other side thinks a physical terminal is connected, and the host still obeys XON / XOFF, ctrl-c, and more.

Likewise, within a MacOS or Windows desktop, users of modern development environments such as Python and NodeJS depend on a local *Terminal* app and an ecosystem of tools that take user input from a command line and deliver output to a character based window with capabilities that mirror vintage terminals.

Finally, the USB to Serial connection to your Foenix debug port is indispensable. You use it to push `.hex` formatted code or, via specialized commands, stop the CPU, pull a range of memory, or update kernel flash. All of this occurs over a high speed serial terminal interface created for the earliest 6502 systems (KIM-1).

The teletype (tty) was based on the typewriter, and essentially had two features (not counting the bell):

- it could print a character and advance something called “the carriage” one position. (yes, the carriage *carried* the paper)
- It could return the carriage, or in our case, the logical cursor to the beginning of the line

With this background behind use, let’s talk about the way that screen memory maps to the display.

Video modes and the x / y grid:

The F256 Jr. support 4 character based video modes for each of two refresh rates. They represent an 80 x 60 screen (or an 80 x 50 screen) and derivations as follows:

@ 60 Hz. 80 × 60, 40 × 60, 80 × 30, 40 × 30
@ 70 Hz. 80 × 50, 40 × 50, 80 × 25, 40 × 25

The upper, left hand corner of the screen, regardless of resolution, is $C000.

To compute the starting address of the 2nd line of text, the developer must sense the video mode, and from it, determine the width of the screen (or set it yourself, even if it's the default to be safe).

The following table, borrowed from the F256 Jr. memory map/manual outlines the bit mapping for the VICKY Master Control Register.

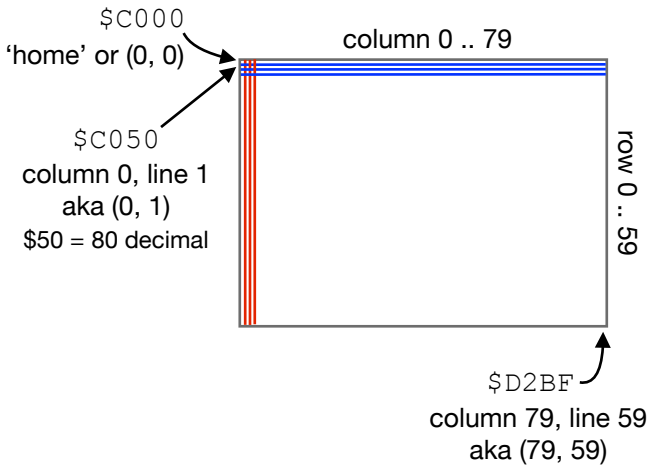
Address	R/W	7	6	5	4	3	2	1	0
0xD000	R/W	X	GAMMA	SPRITE	TILE	BITMAP	GRAPH	OVRLY	TEXT
0xD001	R/W	X					DBL_Y	DBL_X	CLK_70

Forgetting about bits 3 .. 7, text mode is enabled by setting bit 0 of location $D000 to 1; 70 Hz. May be selected by setting bit 0 of location $D001 to 1; and either mode may be selected in double-wide (or high) by setting bit 1 and/or bit 2 of location $D001 to 1.

These settings will dictate the amount of text and dimensions of the screen, and importantly, the “stride” of a line (critical for calculations but not used in our examples).

We discussed *stride* in a graphics context in issue #2 on pgs. 10 and 19; have a look, as this concept will be important as we move on to advanced topics later.

The following graphic details screen locations HOME (0, 0), column 0 - row 1, and the last character of the screen assuming an 80 x 60 text mode @ 60 Hz.



To select this screen mode (monitor withstanding), execute the following instructions:

```
lda $01     ;load MMU_IO_CTRL
pha         ;push to stack to save state
stz $01     ;store 0 to select I/O block
lda #$01    ;select text mode exclusively
sta $d000   ;store to VICKY control reg. 0
stz $d001   ;store zero VICKY control reg. 1
pla         ;pull value to restore state
sta $01     ;store to MMU_IO_CTRL
```

The following table defines starting addresses for the first 39* lines of the screen.

Screen row #	Starting address (hex)	Screen row #	Starting address (hex)	Screen row #	Starting address (hex)
0	$C000	13	$C410	26	$C820
1	$C050	14	$C460	27	$C870
2	$C0A0	15	$C4B0	28	$C8C0
3	$C0F0	16	$C500	29	$C910
4	$C140	17	$C550	30	$C960
5	$C190	18	$C5A0	31	$C9B0
6	$C1E0	19	$C5F0	32	$CA00
7	$C230	20	$C640	33	$CA50
8	$C280	21	$C690	34	$CAA0
9	$C2D0	22	$C6E0	35	$CAF0
10	$C320	23	$C730	36	$CB40
11	$C370	24	$C780	37	$CB90
12	$C3C0	25	$C7D0	38	$CBE0

*for brevity, 39 rows are shown; (continued in the code on the right)

The following string output routine on-a-budget is tailored for a single page application (no scrolling). It relies on a static color palette, initialized along with screen `clear` by a subroutine (see 'e.' below).

The only 'feature' is a plotting function which relies on the first two bytes of the text message to hold an x, y location where the first character should be printed. Some math is required (called out in comments).

The routine will print until a null is encountered. No error checking is performed; so messages longer than 255 bytes will loop endlessly.

This routine relies on two pairs of zero page pointers (`from_ptr` and `to_ptr`) and the zero page indirect y-indexed addressing mode, but is otherwise basic. Later, we will enhance functionality and rewrite the `ctrlcode` and `txtcolor` routines for added functionality.

Line numbers are for reference only; the code posted on the Foenix Market-place is sequenced differently.

```
1                from_ptr = $16          ;zero page 16 and 17
2                to_ptr = $18            ;zero page 18 and 19
3                defcolor = $0200        ;variable for normal
4                pencolor = $0201        ;variable for current
5                ldx message                     ;x location (10)
6                ldy message+1           ;y location (9)
7                lda #<message+2         ;low byte of string
8                sta from_ptr
9                lda #>message+2         ;high byte of string
10               sta from_ptr+1
outstrng         stx x_loc               ;save x a variable
12               tya                     ;move y to accum.
13               asl a                   ;mult x 2 w shift left
14               tay                     ;xfer to y for index
15               lda scrntab+1,y         ;get high byte of row
16               sta to_ptr+1
17               lda scrntab,y           ;get low byte of row
18               adc x_loc               ;add x value
19               sta to_ptr
20               lda to_ptr+1
21               adc #0                  ;take care of carry…
22               sta to_ptr+1            ;just in case
23               stz from_offset
24               stz to_offset
txtloop          ldy from_offset
26               lda (from_ptr),y        ;copy loop
27               beq txtdone             ;if end of line (null)
28               cmp #$20                ;compare to " "
29               bcc ctrlcode            ;placeholder for now
30               ldy to_offset
31               sta (to_ptr),y          ;else store to screen
32               jsr txtcolor            ;currently an rts
33               inc to_offset
ctrlcode         inc from_offset
35               jmp txtloop
txtcolor
txtdone          rts
message          .text $0A,$09,"Hello, y'all",$00
scrntab          .word $CC30 | .word $CFA0
                 .word $CC80 | .word $CFF0
                 .word $CCD0 | .word $D040
                 .word $CD20 | .word $D090
                 .word $CD70 | .word $D0E0
                 .word $CDC0 | .word $D130
                 .word $CE10 | .word $D180
                 .word $CE60 | .word $D1D0
                 .word $CEB0 | .word $D220
                 .word $CF00 | .word $D270
                 .word $CF50
```

39th row continues from here (but we started at 0 so really, it's the 40th-50th)

this notation won't work; (the assembler will bark) it's just for illustrative purposes; the actual code will include the complete 60 line table

It took a while to get here, but let's talk about the '**how**'.

How shall characters be rendered on the screen. What color should they be printed with, and against what background? If double-wide characters are supported, how do we invoke this mode? Does a character set exist to allow underlined characters or reverse field? So many questions …

Few of us had the opportunity to work with the hardwired, impact driven terminals of the early '70s. In the time between characters leaping off the print platen and into a cathode ray tube, enterprising engineers were thinking about how to support the conventions of the past while trying to innovate and navigate a future.

ROM based character sets brought opportunity for text highlighting in the form of bolded, reverse field, and in some cases, embedded underline characters. Solid state CRT terminals supported cursor movement in two or four directions and could seek to start-of-line, end-of-line, top-of-form (screen), etc. Ultimately, terminals supported character-based windows and overlays, scroll and no-scroll zones, programmable status bars and more.

By the birth of the personal computer, this innovation led to a commonly accepted way to tackle these challenges, and it didn't take much for the earliest 8-bit machines to add line draw, cursor control, and other capabilities. The seemingly simple task of writing text to a screen was no longer a simple map/lookup, calculate address, and write to memory.

Of course, the Apple I had none of this utility. A portion of Steve Wozniak's board was dedicated to "Terminal" functions; it included a (really) dumb terminal with three features: A genuine ASCII keyboard, an accompanying opinionated (though upper-case only) character set, and a hardware based Signetics character generator, which together was capable of printing from left to right and from top to bottom. The Apple I featured a 40 x 24 line screen and a 5 x 7 bit font. It also supported "auto-scrolling"; apparently, a big deal at the time.

Apple II's ROM based 'Integer' BASIC had a VTAB command that could move the cursor to a given line on the screen, but the world would have to wait for Applesoft BASIC for real power; namely, commands for HOME (clear), HTAB, INVERSE, FLASH and NORMAL text directives.

Increasing our budget… a little

Let's expand upon our 'on-a-budget' output routine above and consider implementing a BOLD highlighting capability and a FLASH feature. I'll mention here that FLASH is a bit tricky because there is no hardware based (or FPGA) mechanism to accomplish this; we will need to write this ourselves and leverage IRQ interrupts and an enumerated set of palette entries for it.

BOLD is also tricky, because we are limited to characters which are only 8 pixels square so there is no feasible way to thicken or antialias a font. This would be doable if we restricted ourselves to white and gray.

Regardless, let's discuss options for invoking extended attributes via inline characters stream. We'll touch upon each (of 3) briefly and then move on to a specification:

1. Use an inline **non-printable** control character with a value of less than 32 decimal. Expressed within an assembly `.text` directive, we will use $11 for FLASH and $15 for NORMAL. Example follows:

```
.text    $0A,$09,"Please ",$12,"FLASH ME"
         ,$15," - thank you, kindly!",$00
```

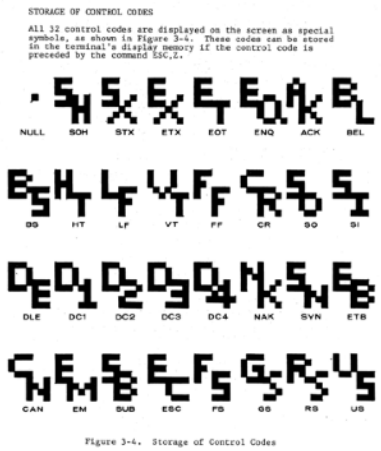
   (we will leverage this scheme in our not-quite-on-a-budget algorithm on page 22)

2. Use a **sequence of codes** which are *escaped* for example, "{*ESC*}`[1M`" to represent the a BOLD directive. The benefit of this type of control sequence is ease of detection (always starting with ASCII 27) and also, it allows for an extensive library of features since it is multi-character; the downside is twofold: a) if you really want to 'send' an escape, you'll need to escape it by duplicating the code (a minor nuisance); and b) it requires more chars for the directive (4) and gets ugly in source. This happens to be the Digital Equipment (DEC) standard, leveraged in the VT100 specification.

3. Use a **printable graphic characters** which, by convention, can be embedded within a string. It should be a character you would not normally type. Commodore chose this route with the PET, VIC 20, C64, C128, C16, and Plus/4 line of products. By accident or design, it was a well architected scheme and it stood the test of time.

   See the Leonard Tramiel callout in the FONT discussion (item 'f.') below.

Retro flashback ASCII terminals such as early ADDS Regent and DEC VT (video terminals) included ctrl-code selectable debug modes which, when invoked, output printable glyphs for ASCII values 0 .. 31; the chart to the right from a vintage terminal doc leveraged a font for this purpose.


STORAGE OF CONTROL CODES

All 32 control codes are displayed on the screen as special symbols, as shown in Figure 3-4. These codes can be stored in the terminal's display memory if the control code is preceded by the command ESC,Z.

NULL SOH STX ETX EOT ENQ ACK BEL

BS HT LF VT FF CR SO SI

DLE DC1 DC2 DC3 DC4 NAK SYN ETB

CAN EM SUB ESC FS GS RS US

Figure 3-4. Storage of Control Codes

This was useful in debugging encoding issues, while building Unix TERMCAP entries, or just for sport.

It was not uncommon for terminals to have such a debug mode; it **was** uncommon for a manufacturer to dedicate bit mapped ROM space for this range. The ASCII backspace, bell, line feed, and carriage return, are among the more common single character non-printable ASCII characters that are still relevant.

Medium budget kernel output routine specification

Hex	ASCII	Feature
$07	BEL	Plays SID bell sound (see pg. 18)
$11	DC1	CLEAR SCREEN*
$12	DC2	FLASH on
$13	DC3	RVS field on
$15	NAK	NORMAL (resets mode attributes)
$18	CAN	BLK text color
$19	EM	WHT text color
$1A	SUB	RED text color
$1B	ESC	CYN text color
$1C	FS	PUR text color
$1D	GS	GRN text color
$1E	RS	BLU text color
$1F	US	YEL text color

We support 8 text colors; (with a catch and a hidden feature) the upper 8 are for flashing; an IRQ shim controls a counter based on the VICKY text cursor flash rate (see `doc table 3.5`) which changes the FG color to match its BG

necessity; mother of invention

This routine is deemed *medium* budget because it includes a few interesting features that you might expect from an output routine, but it falls short in completeness and has limitations since attributes are mutually exclusive (e.g. it is not possible to have reverse-field flashing; you can only have one of the two, or normal text. A color an be normal or reverse field or flashing.

This scheme includes something old and something new. The *old* is obvious, it's the bell. It's a relic of days past; ^g for the OGs. The *new* is also something old, but it's new again as of this week. It is support for text colors represented on keycaps, recently announced for the F256K (see pg. 26). Foenix systems have always had wide ranging color support, but *key* color selection is again front and center.

Make color easy again; change your default color with a simple keystroke and `poke` a background color and border just because you can!

Text mode palettes

The F256 Jr. supports a 16 color palette for the text foreground and a separate 16 color palette for the text background. They need not contain the same data, but for our purposes, we will make the first 8 of each identical so we can orchestrate a true reverse field effect.

We've covered this prior (in Beginner's Corner issue #2, pgs. 21-25 and in issue #3 pgs. 22-23) but to quickly recap, each color is composed of a 24-bit value represented as RGB. Including the alpha channel byte (currently not implemented in VICKY), each color requires 4 bytes. 16 colors * 4 bytes = 64 bytes or $40 bytes hexadecimal. Importantly, they are ordered (in memory) in reverse; aka (B)lue, (G)reen, (R)ed, (A)lpha.

The foreground color LUT is located at $D400 and the background color LUT is located adjacent, at $D440.

Once established*, *double nibble bytes*** stored in I/O bank 3 (see 'b.' on pg. 18) correspond to characters stored in bank 2 on a 1:1 basis.

*code to do this (top) requires I/O bank 0 selected

```
init_pal    ldx     #0                  $D800
_loop       lda     _palette,x
            sta     TEXT_LUT_FG,x
            sta     TEXT_LUT_BG,x
            inx                                 I stole this from Gadget
            cpx     #64
            bne     _loop
            rts                 $D840

_palette
      .dword $000000    ;C64 black (BLK)
      .dword $ffffff    ;C64 white (WHT)
      .dword $68372b    ;C64 red (RED)
      .dword $70a4b2    ;C64 cyan (CYN)
      .dword $6f3d86    ;C64 purple (PUR)
      .dword $588d43    ;C64 green (GRN)
      .dword $352879    ;C64 blue (BLU)
      .dword $b8c76f    ;C64 yellow (YEL)
      .dword $6f4f25    ;C64 orange (ORN)
      .dword $433900    ;C64 brown (BRN)
      .dword $9a6759    ;C64 pink (PNK)
      .dword $444444    ;C64 dark gray (DK GRY)
      .dword $6c6c6c    ;C64 gray (GRY)
      .dword $9ad284    ;C64 light green (LT GRN)
      .dword $6c5eb5    ;C64 light blue (LT BLU)
      .dword $959595    ;C64 light gray (LT GRY)
```

… and these from Paul's github

About the code

The core of the code on page 20 does not need to change but we will resolve two labels, currently non-functional stubs. We've already got a trap for null (end of string) on line 25 and a `bcc` branch to `ctrlcode` on line 27 for char values less than 32. We also have a `jsr` to a subroutine called `txtcolor` which takes care of color for printable characters (discussed below).

From a house keeping perspective, all we need to do is increment the `y` register to keep our index moving and ultimately, `jmp` or branch back to `txtloop` on line 24.

The `ctrlcode` routine uses a series of compare and branch instructions. If the feature list was longer and more varied, a jump table with a linear search would be appropriate. But we've organized our control codes in such a way that a `cmp` with `#$18` and accompanying `bcc` branches to the `setcolor` routine where a simple index into the predefined palette is used. Let's knock off these routines one-by-one (starting with the most simple):

```
bell      jsr playbel
40        jmp txtloop

clrscrn   jsr clear
42        jmp txtloop

normal    lda defcolor
44        sta pencolor
45        jmp txtloop

flash     lda pencolor
47        ora #$08
48        sta pencolor
49        jmp txtloop
```

**double-nibble as in `$0C` where `'0'` hex represents the 0th foreground color (BLK) and `'C'` = background (GRAY)

Reverse Field - a slightly more difficult problem

```
reverse  lda pencolor
51       asl a
52       bcc rev_bit1
53       ora #%00000001
rev_bit1 asl a
55       bcc rev_bit2
56       ora #%00000001
rev_bit2 asl a
58       bcc rev_bit3
59       ora #%00000001
rev_bit3 asl a
61       bcc rev_done
62       ora #%00000001
rev_done sta pencolor
64       jmp txtloop
```

This code might seem repetitive, it is. Is it efficient? Good question. Have a look at the alternative below and think about which is better. As is, the above example is 15 instructions, 33 bytes in length, and in a worst case, requires ~35 cycles. It's subtle, but the worst case in this example is to NOT branch (all bits need to be commuted to the right nibble); this consumes one more cycle (the `ora` consumes 2 cycles) than a branch.

The routine below is shorter and may appear more efficient, but it's nearly twice as long in cycles; again, considering a worst case scenario which is unlikely and actually pointless*. The routine below is 10 instructions, 23 bytes in length, and consumes ~59 cycles. We burn a number of cycles getting to and returning from the `setbit` branch (and this option is taken every time in a worst case scenario). Ideally, it's good to know the probability/distribution of your data when you design a default path. In this case, it's actually arbitrary as it depends on color use. In the end, the example above is preferred. Sometimes, simpler is better.

As a piece of code that we will only execute when reverse field is invoked, efficiency won't matter; but if code like this was embedded in an IRQ routine and iterating hundreds of times, variability or bloat could affect the stability of the system or at a minimum, squander resources. We will cover IRQs next time; they will be instrumental in 'animating' the FLASH routine.

```
reverse     lda pencolor
            ldx #$04
rev_loop    asl a
            bcs setbit
rev_incr    dex
            bne rev_loop
rev_done    sta pencolor
            jmp txtloop
setbit      ora #%00000001
            jmp rev_incr
```

We are nearly done with the setup, decision flow, and method of tracking attributes. You are probably getting closer to guessing how this works. Let's have a look at the main branch code and then discuss.

```
ctrlcode from_offset
66       cmp #BEL
67       beq bell
68       cmp #CLR
69       beq clrscrn
70       cmp #NOR
71       beq normal
72       cmp #FLA
73       beq flash
74       cmp #RVS
75       beq reverse
76       cmp #BLK
77       bcc txtloop        ; if < lowest (black)
78       sec
79       sbc #$18
80       tax
81       lda colors,x
82       sta pencolor
83       jmp txtloop
```

Theory of operation

`defcolor = pencolor` is written to all locations of color memory when the screen is cleared and `pencolor` is returned to `defcolor` anytime the NORMAL control code is embedded. This might seem like a peculiar move since we are updating every character's color memory every time, but we are taking this step in case screen memory is written to directly as explained in item "a." above. One such use case might be a text windowing environment.

`pencolor` is what is unconditionally written to color memory as text is rendered on the screen; the output routine merely writes the character to screen memory, flips the I/O bank to text display color, and stores the current color for the corresponding character.

When RVS field is requested, the pencolor is altered such that the upper and lower order nibbles (4-bits) are swapped. The code to the left pushes 4 bits left-wise, rotating the 7th-to-carry then into the bit 0 via an `ORA`.

FLASH does something we've addressed prior. In issue #2 of beginners corner, we altered the color LUT for sprites by periodically changing the color of the Foenix Balloon beacon; here, we are managing a counter tied to the IRQ (every 1/60th of a second), and when at the FLASH 'point', will change the upper 8 (unusable) pen colors to have the same RGB as their background. There are many ways to accomplish this but this was fun to write. It comes a the expense of wasting 8 of our screen colors, however. As mentioned, "on a budget".

An alternative could be to maintain list of screen start and stop locations to be flashed, and literally erase these characters and re-draw them. (this is madness).

Another is to only permit flashing for one or two colors; this would limit the waste, but is restrictive.

Let's finishing things up for now with a complete `txtcolor` routine, and code to `clear` the screen.

*with all bits set, we are rotating $FF to be… also $FF. Pointless, but in this case it is not worth additional code that would otherwise address corner cases.

Text color - managing color display memory

Lines 28 and 29 of the code on pg. 19 determines whether or not the next character in the string is a control code character or not; if so, `ctrlcode` is called; if not, however, text display memory is updated via this code on line 31:

```
31          sta (to_ptr),y      ; store to screen
```

… of course we still need to update color memory!

The beginning of the code block on lines 11-22 does all of the work to convert the *x* and *y* location to a memory address and writes the low-byte/high-byte pointer pair in `to_ptr`. As luck would have it, color memory exists on a parallel plane to character memory, so we merely need to change the I/O bank (3 = color mem) and then change it back (2 = text mem) before returning. Line 80 should look familiar to the line above. (it's identical)

Here is the complete color subroutine:

```
txtcolor lda #$03
85          sta $01
86          lda pencolor
87          ldy to_offset
88          sta (to_ptr),y
89          lda #$02
90          sta $01
91          rts
```

Could this be made more simple? Of course it could; we could smash it into the code block above and avoid the `jsr` and `rts` but it's good practice to isolate routines that will be shared; or to avoid complexity to make code more readable. (our aim)

e. Clear screen subroutine

This routine clears the screen by writing ASCII 32 (" ") to text memory and `pencolor` to color memory. We do this in two passes and each pass uses the main loop, and a secondary, small loop.

The `clear` portion of this routine loads the accumulator with the desired character value and takes care of the I/O bank (2 = char mem) before calling `fill`.

Next, it then loads the accumulator with `pencolor`, makes it the default (`defcolor`), sets the I/O bank (3 = color mem), and again calls fill.

There are at least 4 ways to do this (to be discussed next time) but this one is the most fun as you will see.

```
clear       lda #$20
93          ldx #$02
94          stx $01
95          jsr fill
96          lda pencolor
97          sta defcolor
98          ldx #$03
99          stx $01
100         jsr fill
101         ldx #$02
102         stx $01
103         rts
```

In the old days when screens had 960 or 1,000 characters, a loop with indexed `sta` statements could take care of this without calisthenics. One could merely use store instructions indexed on consecutive pages:

```
loop        sta $c000,x     ; or $0400 on a C64, etc.
            sta $c100,x
            sta $c200,x
            sta $c300,x
            inx
            bne loop
```

This will work on a Foenix 40 x 25 screen but not on the ole' C64 where we knowingly stomp on 24 bytes of memory that does not belong to us; sprite pointers!

On a Commodore 64, screen RAM runs from $0400-$07e7 and it is followed by 16 bytes which are unused, and then 8 bytes for sprite pointers. This might be ok if we are not using sprites, but it's sloppy. A better method is to move the `sta $c300,x` to a small loop that will spin for $E7 iterations to clean up the remaining bytes.

Here comes the fun. On a F256 Jr., we will need nearly 4x the number of absolute store statements (15 of them) to accommodate the 80 x 60 screen. This is boring. Let's use 15 lines of code to do something crazy instead.

The following code runs a fill loop that stores full pages worth of the passed in accumulator values in the inner loop and then increments the high byte of the `sta` such that the $c0 page advances to $c1 and so on. This outer loop continues until high byte = $d2 and then we clean up with a small loop similar to what we discussed above (for 192 iterations). Finally, we 'fix' the high byte that is part of the original code before exiting.

```
fill        ldx #$00
filloop     sta $c000,x
106         inx
107         bne filloop
108         inc filloop+2
109         ldx filloop+2
110         cpx #$d2
111         bne filloop
112         ldx #$c0
smfill      sta $d1ff,x
114         dex
115         bne smfill
116         ldx #$c0
117         stx filloop+2
118         rts
```

Inner loop - no `cmp` needed; the zero flag resets us for the next page

Outer loop - bumps the high-byte address until $d200 is reached

This will restore the $c0 high byte that we started with (for next time !!) . If we don't do this, we'll create a mess just like C programmers that don't mind their *p's and queues.

We could have done this the "6502 way", leveraging a pair of zero page addresses as we did with `to_ptr` above, but this approach is eyebrow raising and a somewhat advanced topic, so it's worth challenging ourselves to mess with memory and sleigh the one-off-error demons in the process.

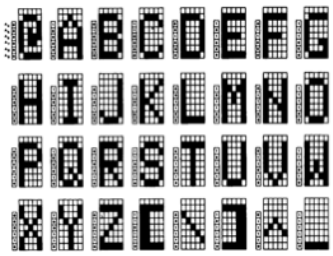
We will take this one step further next issue when we modify code branching for an expendable (single-use) use case to improve performance in a loop that iterates a few thousand times. Saved (or spent) cycles can really add up! It is unlikely that anybody will ever see your code, but you'll know it's there.

f. Using redefined characters (aka fonts)

There is only so much you can do with an 8 x 8 typeface. Across the years, numerous examples of 8-bit machine character sets have been used and abused.

In this section, we will cover three historically significant examples, talk about resources where you can nab your own fonts, and then examine code for use in your programs (on the F256 Jr., redefined character sets are easier than you might expect).

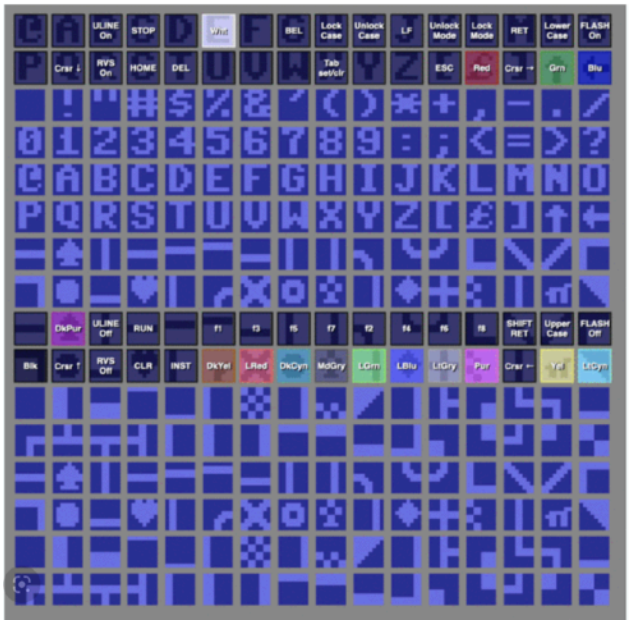
Example 1: Apple I and the aforementioned **Signetics 2513** character generator. Billed as an 8 x 5, the top line was always blank in order to provide vertical spacing between characters. On the left is a taste (1/2) of the set from the original Signetics documentation (linked above). 64 characters in total; nothing but upper case alpha + numeric + ASCII special symbols. This was the starting point.



Example 2: Apple II (and the II+) added inverse and flashing in addition to a 4th repeated block of the same character set for no known reason (not shown). Apple was not ready to support lower case text yet (the keyboard wasn't yet equipped for it either). But reverse field and flashing (with Applesoft support to leverage the new video modes) was a step in the right direction. The *II*e delivered lower case for the first time for Apple, however graphics mode kludges and add-on cards with keyboard mods were popular in the II+.

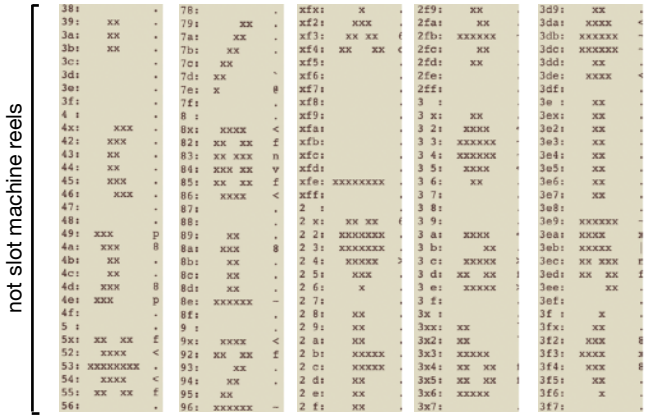
Example 3: Commodore PETSCII was designed jointly by Leonard Tramiel and Chuck Peddle as part of the integration of Microsoft BASIC, released in 1977 on the first Personal Electronic Transactor (PET). In a wicked twist that cursed Commodore for years, they based the original set on the 1963 version of ASCII (which did not have lower case) rather than the 1967 version that everybody else was using. This meant that the default set would have upper case in the correct sequence but in a mixed mode (upper/lower), Commodore opted to retain compatibility with their upper case (and graphic) only character set and swap cases such that programs encoded for the default upper case would default to lower in mixed case mode. (The opposite would have had encoded text in broken graphic chars which would be unacceptable. It wasn't the end of the world, but Commodore was definitely the outlier.

Whether or not they made up for it by affording a rich graphic set is up for debate and argument. One thing Commodore did do well was to incorporate a data entry and encoding methodology into the kernel, place symbols on keys, and (in 1980) extend the set to support color. The *quoted* input mode, frustrating at first, was instrumental in putting all of those graphic symbols, colors, and cursor control into the hands of the masses.



In case you wondered where the '↑' came from, thank the 1963 ASCII standard. In 1967 it became the caret '^' we know today. Check out **this** retrospective from University of Turku and Aalto University in Poland, which lead me to **this** amazing online PETSCII editor.

An outstanding resource for font data can be sourced from **this github repo**. I stumbled upon it from **this** ATARI site. If you are Unix savvy, a pulled `.FNT` from the github produces viewable output as follows:




```
xxd -g1 -b -c1 ACADEMY.FNT | tr 0 ' ' |
tr 1 'x'
```

(my SparcStation is gone!; this is MacOS*)

Provided the font is 8 x 8, it will be load-ready into the F256 Jr. without fuss. The following section explains.

* if you grew up (or worked professionally) with Unix, you are (I am) utterly thankful that MacOS is based on MACH, which was derived from Carnegie Mellon's version of Bell Labs Unix. Thank NeXT and Steve Jobs for that one.

The F256 Jr. supports a single character set which is instantiated by VICKY (and potentially altered by the kernel) during initialization.

The dimension of the set is 8 (pixels wide) x 8 (pixels high) x 256 (characters) such that a stream of bits and bytes are ordered with $C000 containing the top 8 pixels of the first character, $C001 containing the 2nd of 8 rows of pixels for the first character, and so on.

The character set sits in memory (I/O block 1) between $C000 and $C7FF. To load a new set into memory, simply copy from anywhere into this region but be sure that zero page $01, the `MMU_IO_CTRL`, contains #$01.

Important to take care that your alphanumeric characters align with the representative values else the machine could be rendered unusable (if typing 'a' places an 's' on the screen, or worse). Note that the Foenix platform does not currently have an NMI bound hot key (that on Commodore platforms would reset the system to its standard ROM based character set).

What's on your F256? Depending on your FPGA load and kernel flash, you may have the following official released character set (as of end of December 2022).

This set features alphanumerics with 'g', 'j', 'p', 'q', and 'y' descenders, a set of dithered fill patterns, graduated horizontal blocks in two styles, card suits, and other useful glyphs. Unlike PETSCII, characters are ordered such that familiar symbols are adjacent to each other. Created by Foenix community member and Discord user Micah, you can secure it from his **github here**.

Incorporating a font file within 64TASS source

The following assembler directive will load the identified file into memory as follows:

```
charset .binary "petscii.bin", 0, 2048
```

The following code, performs the copy, overwriting the default character set with characters of your choice:

```
            lda #<charset
            sta FR_PTR
            lda #>charset
            sta FR_PTR+1
            lda #$C0
            sta TO_PTR+1
            stz TO_PTR
            ldy #$00
charloop    lda (FR_PTR),y
            sta (TO_PTR),y
            iny
            bne charloop
            inc FR_PTR+1
            inc TO_PTR+1
            lda TO_PTR+1
            cmp #$c4
            bne charloop
```

don't forget to set (and later reset) to I/O bank 1

Editing characters within source

One other method that you might find useful is to express bitmaps in binary using assembler directives. A suitable tile editor would be better, but this WYSIWYG approach will suffice in a pinch.

The following was pulled from the ATARI (400/800 family) font set; the full character set is available on the Foenix Marketplace.

```
;$3F-underline   ;$40-heart card  ;$41-mid left win
.byte %00000000  .byte %00000000  .byte %00011000
.byte %00000000  .byte %00110110  .byte %00011000
.byte %00000000  .byte %01111111  .byte %00011000
.byte %00000000  .byte %01111111  .byte %00011111
.byte %00000000  .byte %00111110  .byte %00011111
.byte %00000000  .byte %00011100  .byte %00011000
.byte %11111111  .byte %00001000  .byte %00011000
.byte %00000000  .byte %00000000  .byte %00011000
```

Closing thoughts for now…

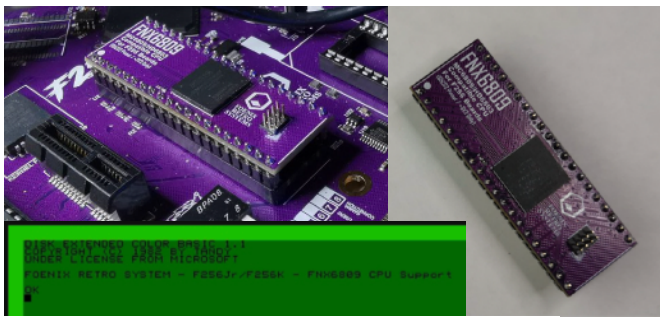
It is our hope that you find some of this entertaining, educational, or maybe even usable in your code.

We will continue this discussion in the next issue with the following topics:

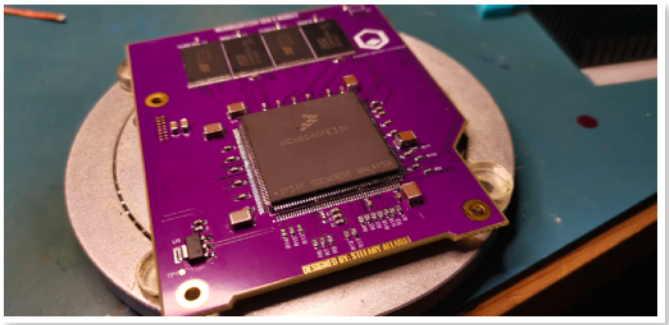
g. keyboard input on a budget (again, sans kernel)
h. Reading DIP switches
i. Interpreting the real-time clock circuit
j. Leveraging the VIA 6522 timers
k. VIC-20 style bit-mapped graphics
l. Reading ATARI style joysticks

F.R.

Spy photos from Foenix Labs



FNX6809 - MC6809 implemented in FPGA
(a drop in replacement for the WDC65C02 for F256 Jr.)



MC68040 - 3.3V A2560X CPU module

Back Page - Vintage Advert Time Machine

Happening now - the **F256K desktop announced and on offer**

What does $595 buy? In December of 1982, the answer was the Commodore machine pictured below (left). When compared to Apple, IBM, Tandy and others, the Commodore 64 was a bargain. Not only did it boast 64K of RAM (others struggled to make 32K or 48K affordable), but the capabilities of the machine were class-leading in nearly every regard.

Through the insistence of Michael Tomczyk (Commodore marketing guru) the company secured the back page of popular computer magazines and was relentless in product development, distribution, and advertising. This, combined with an in-house chip fabrication capability and Jack Tramiel's shrewd business practices, Commodore dominated the market for many years. They leveraged an already established PET dealer network at first, then sold through a growing population of computer stores, and ultimately, toy stores worldwide.

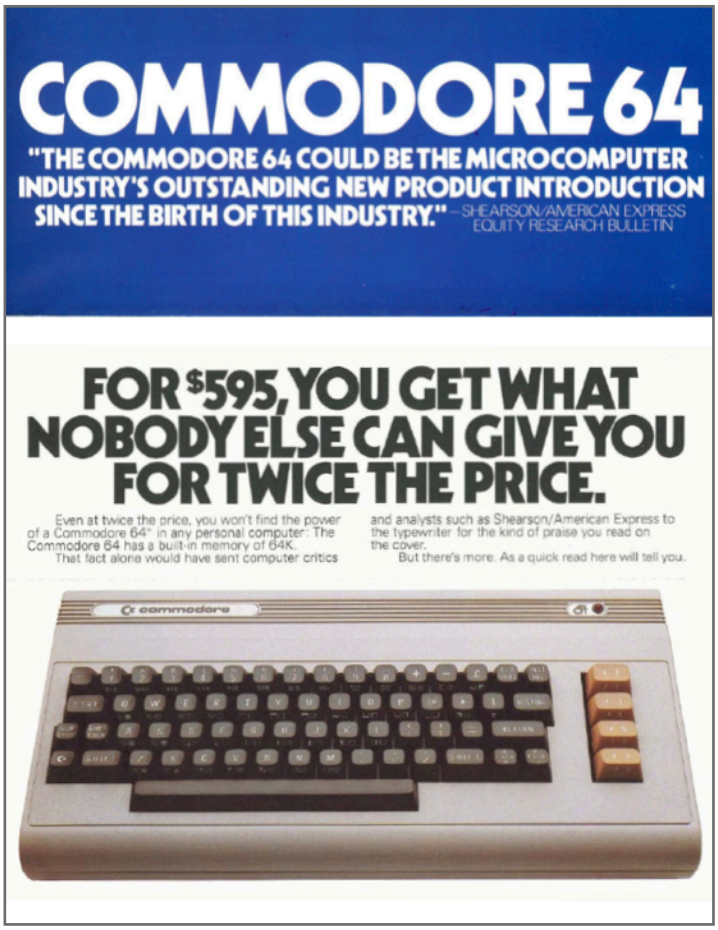
Fast forward to today, and the inflation calculator suggests that $595 in 1982 dollars adjusts to $1,835 in 2022 dollars.

For approximately 1/3rd of this inflation adjusted price, $595 in 2022 currency gains access to the natural successor to the C64, with vastly expanded and added capabilities.

On December 20th, Stefany Allaire of Foenix Retro Systems announced the upcoming release of the F256K computer.

Your $595 (USD) secures a place in line for the newly announced machine. Created in the spirit of the C64, the F256K features a modern mechanical keyboard (with your choice of keyswitch type), and all of the F256 Jr. features spun into a slim desktop style case, complete with a cartridge/RAM expansion slot. With a WDC 65C02 CPU running at 6.29 MHz., accommodations for two SID chips (BYOS*), a powerful FPGA based graphics engine with dual embedded PSG sound chip instances, Commodore standard IEC peripheral support, multiple text modes (up to 80 x 60), DVI-I video output, and much more; the combination of features and cost make it feel like Christmas 1982 all over again.

Then (1982)



Now (2022)

The first round offering is available for sale for a limited time, for April 2023 shipping. See www.c256foenix.com/f256k for details and full specifications.

*BYOS: bring-your-own-SID; the F256 supports +9V or +12V SIDs, not included