Foenix Rising

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

Issue #2 August 2022

Twice the fun

(2x the content, same low price...)

In many parts of the world, August represents the gap between the heaviest vacation season of the year and school starting up again.

For those waiting on their Foenix to ship, this month will mark the long awaited arrival of machines including the first prod versions of the A2560K (featured in the last issue), and the C256 Jr. dev board, which we will spend some time on in this issue.

But what about the GEN X? Issue #3 will focus on it; with a view from the inside of the updated revision.

The other news this month is the opening of the Foenix Marketplace. We'll have a quick look at the ins & outs of how to find and download software for your Foenix machine.

And by request, *Foenix Redux*; a primer which covers the history, functional differences and commonalities between several of the Foenix sister systems.

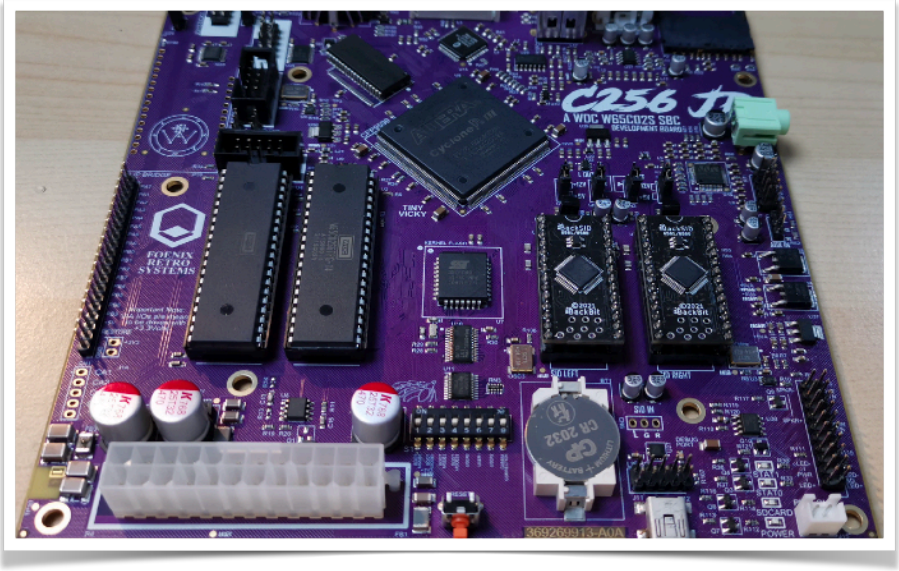
We are also featuring an article by our first contributing author, Ernesto Contreras, who has provided a Calypsi install guide and a C library for graphics named *VickyGraph*.

Finally, check out part 1 of our retro feature on COMPUTE magazine.

Thank you as always for your interest and support.

-EMwhite

Featured Photo - C256 Jr. dev board rev A



Junior - more than the sum of vintage parts

Revision 'A' of the Foenix C256 Jr. dev board supports old-school Commodore keyboards and IEC Serial bus peripherals, has a banked RAM scheme, is 65C02 based, and is 'just' the perfect little package. We introduced it briefly last issue and will have a closer look this month.

VTOC (volume table of contents)

`git` and URL Resource Directory

Updated each issue, this space will contain links to public facing Foenix related development efforts (useful to see how others have solved particular problems, to find docs or to see what is out there)

Lang	**https://github.com/paulscottrobson/basic02**
Compiler	https://github.com/hth313/Calypsi-m68k-Foenix
Compiler	https://github.com/hth313/Calypsi-65816-Foenix
Env	https://github.com/WartyMN/A2560-FoenixRetroOS
Env	https://github.com/Trinity-11/FoenixIDE
Env	**https://github.com/paulscottrobson/f68-emulator**
Game	https://github.com/dtremblay/fraggy
Game	https://github.com/dtremblay/c256-tetris
Kernel	https://github.com/pweingar/FoenixMCP
Kernel	**https://github.com/ghackwrench/OpenKERNAL**
Lang	https://github.com/daschewie/FoenixBasic68k
Library	**https://github.com/econtrerasd/VickyGraph**
Samp code	https://github.com/noyen1973/C256-Foenix
Utility	https://github.com/econtrerasd/Foenix-Sprite-Editor
Utility	https://github.com/dtremblay/c256-vgm-player
Utility	**https://github.com/pweingar/FoenixMgr**
Utility	**https://github.com/econtrerasd/playSong**

bold = newly added

Links to other Foenix Resources:

Foenix Retro Systems Home Page
Foenix Discord Invite
Stefany Allaire Patreon Page
Stefany Allaire Twitter

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 and related retro technologies.

Distribution: ~2^{10}-1

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

Correspondance: `commerce@emwhite.org`

Classic Arcade Games

H	C	L	A	R	K	A	N	O	I	D	E	U	S
G	W	B	N	T	C	O	M	M	A	N	D	O	Q
A	V	C	M	I	L	L	I	P	E	D	E	C	U
U	J	R	T	X	I	X	T	B	U	H	V	R	R
N	Z	A	X	X	O	N	W	E	T	T	V	A	T
T	W	G	Y	R	U	S	S	Z	H	R	C	Z	S
L	S	C	R	A	M	B	L	E	U	O	Y	Y	T
E	T	T	T	H	N	L	J	R	P	N	P	C	A
T	R	E	A	A	D	A	N	K	G	J	H	L	R
Q	Y	A	M	N	P	J	R	S	A	A	O	I	G
Q	V	J	I	P	K	P	A	T	L	A	E	M	A
I	I	X	O	D	E	V	E	G	A	V	N	B	T
N	G	X	G	P	E	S	U	R	G	H	I	E	E
O	U	T	R	U	N	N	T	Z	A	L	X	R	U

Find ~~19~~ 18 **single-word** coin-op game titles. I went ahead and circled my favorite from this list since:

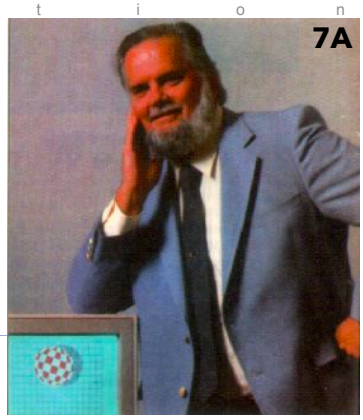
a) it's actually a two-word title but I wanted to give this unique classic a mention
b) I owe you one because I dropped the ball on last month's puzzle (pg. 3)

No PACMAN or ASTEROIDS above and many titles are staring right at you. But keep in mind, your notion of 'classic' may differ from the puzzle-master depending on your age, experiences, or where you live. We will dive briefly into unique features of a few of these in the next issue; there is something significant about many of these titles. I wager that Stefany and Gadget will do well on this activity :)

Next month, we will visit the developer of a *clean-room* version of a familiar classic which is well underway for C256 platforms (also applicable to the GEN X). No spoilers here but jump on Discord and you'll read some posts about it along with a `.hex` working demo that you can get your hands on as I write this. I just ran it and it is spectacular !!

Speaking of classics, in the September/October timeframe, we will dive in and recreate a portion of a pre classic-era arcade gem, *Atari Tank* (1974); 'Tank' is a precursor to an actual classic, *Battle Zone (1980)*.

'Tank' had cabinet-rumbling sound effects that Atari reused in its 3D vector graphics brethren. We will model the sound on a Moog Model 15, on discreet HW, and then attempt to recreate them using "Foenix sound"!!



1A: Mario's Nintendo father, credited for saving gaming

4A: Chuck E. Cheese founder and pong daddy

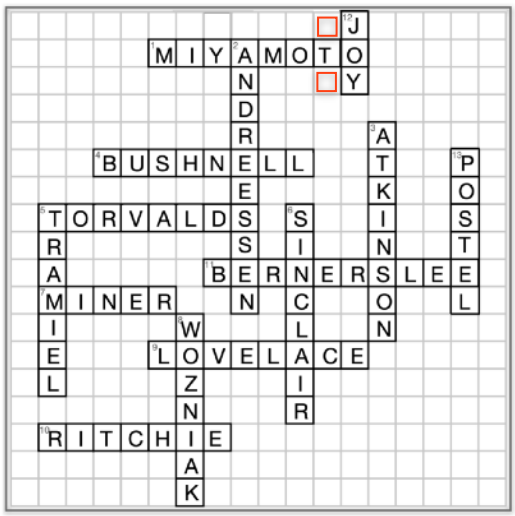
5A: First name shared with Peanuts Lucy's brother

7A: _ _ _ _ _ 2049'er and Lorraine founder

9A: Lady Ada's (not Limor Fried) surname

10A: Bell Labs Lang gifter and National Medal of Technology recipient

11A: Cubist inventor of W3 (suppress the hyphen, if there is one :)



2D: www HoF member and NCSA wunderkind

3D: Macintosh GUI King and Hypercard inventor

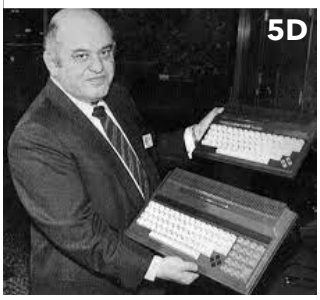
5D: "Jack Attack", known to market to the masses

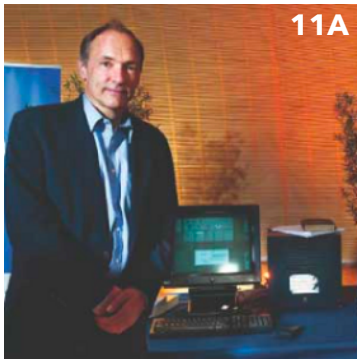
6D: Knighted Clive partnered with a US watch company to introduce a doorstop

8D: D-list' partner and Sharks 'super-fan'

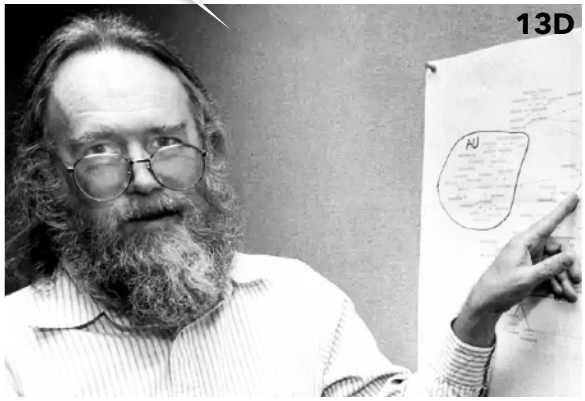
12D: Responsible for making 'Ed' visible. Co-founded a Unix workstation giant

13D: Root domain admin, SMTP RFC editor, and 'g-d of the internet'





click on pictures for brief, but interesting curated articles for "a reminder of their contributions"



It's true, Bill Joy's name (12 down) is spelled "J-O-Y", and not "J-T-Y". The Puzzle above was unsolvable as printed in issue #1. I'm sorry …

Foenix Redux

A not-so-brief history lesson for the uninitiated

I exchanged a few messages with a retro enthusiast from Thunder Bay, Ontario recently; he had interest in the history of Foenix. Where did it come from? What differentiates the products from one other? What is available now?

Admittedly, I take for granted the fact that some percentage of the population have been following the platform closely since the early days of the FMX, but a far larger group is just getting onboard and information, well… the internet.…

If you'll bear with me, I'd like to rewind the clock some four or five years and trace the origins of this platform, discuss some similarities and differences, and then quickly catch up to the currently shipping and soon-to-ship products including the recently announced C256 Jr.

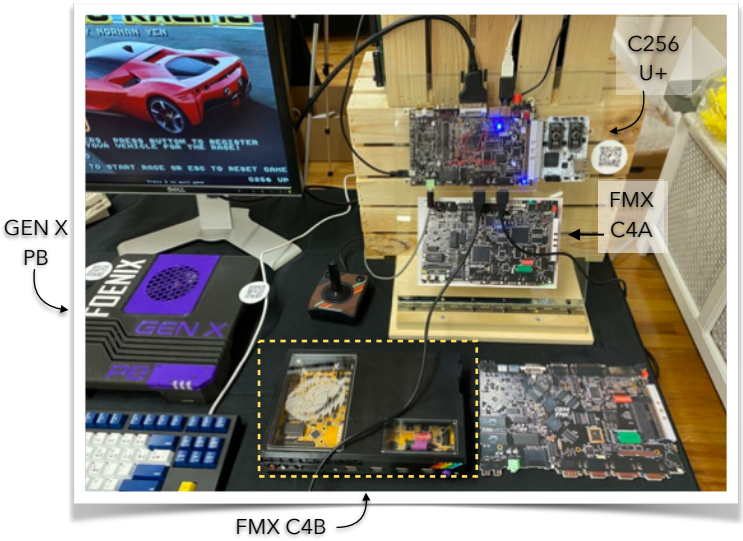
I *had* planned to depict a comprehensive memory map in 'this' space this month, but with the Rev. 2 dev platform of the C256 Jr. still taking shape, I've opted to put that discussion off a month and instead, will draw upon some of the text I complied for the VCF East event this past Spring for a more worthy cause. This will be repeat for some, but new to many.

The Beginning: Foenix Music eXtension (FMX)

The beginnings of Foenix can be traced to a conversation between Stefany Allaire and David Murray (The 8-bit guy) that occurred at a Portland retro conference. The story has been told many times but the short version is that a discussion about collaborating led to different paths which led to Stefany going out on her own to begin design and to build an operation that she would own end-to-end.

Here is an early video that begins with 'challenge accepted' then goes into the first of a few subsequent revisions of the first product (the C256) which eventually became the FMX.

C256



C256 Foenix Family photo (of sorts)

- C256U+ w/EXP-C100 eSID expansion card
- GEN X PB (Pizza Box) prototype in black and purple (in front of monitor)
- FMX C4B in original black 3D printed case with transparent windows. (this machine is treasured by some but in most cases, are sadly sitting on shelves, collecting dust)
- "picture" of an early PDIP based version of the FMX (see larger pic below)
- (just below the U+) 1/2 scale FMX (C4A)

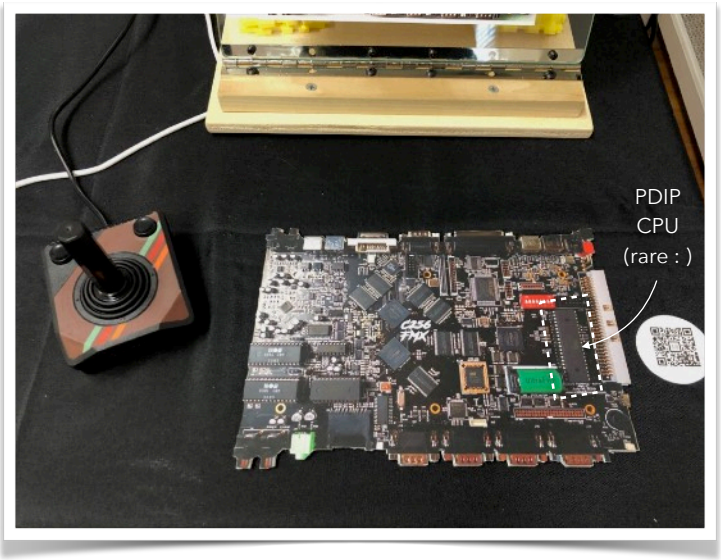
FMX

The original FMX Foenix (with roots in the C256) was designed, built, and improved upon in nearly a dozen iterations, then shipped to early adopters culminating in the GA release (see yellow dashed area at the bottom of the pic above).

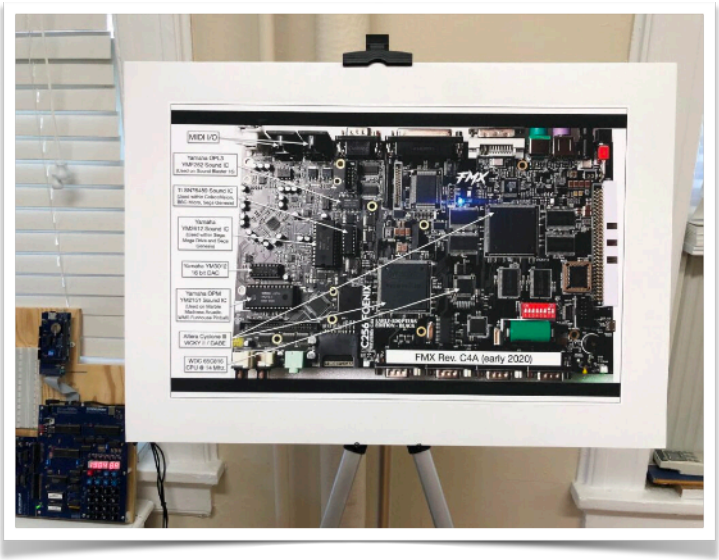
With a concentration on graphics and music, the Foenix platform offers retro enthusiasts a chance to get close to the hardware while enjoying a step-up in performance compared to the 1 or 2 MHz. systems that many retro enthusiasts hold dear. To paraphrase the sentiment behind Stefany's philosophy, she seeks to recreate the experience that many of us started with decades ago when we first switched on our 6502 based home computer and were faced with a blinking "**READY.**" prompt and nothing more than a reference manual with a few sample programs to key in.

All of the Foenix products are produced in Western British Columbia, Canada. Stefany designs the hardware (in many cases, 6 layer boards), develops the FPGA code, integrates classic ICs, selects components and designs and prints enclosures, solders and assembles, and ships worldwide.

Of course the operating environment, kernel, development tools, utilities, games, and demos are coming from all corners of the world and interest and adoption is increasing on a week-by-week basis. Collaborators Peter J. Weingartner and Daniel Tremblay have been working hand-in-hand with Stefany for years, and have built various kernels, virtual environments, and other software. Many more have joined recently and are producing software including highly interactive and well produced games for the community to enjoy.



Picture of an early FMX board. Note the 40 pin PDIP based 65816 near the expansion connector on the right. All other boards*, through the new GEN X utilize the QFP



This poster was created for the April 2022 VCF East conference and details features of the 'early adopter' FMX version C4A

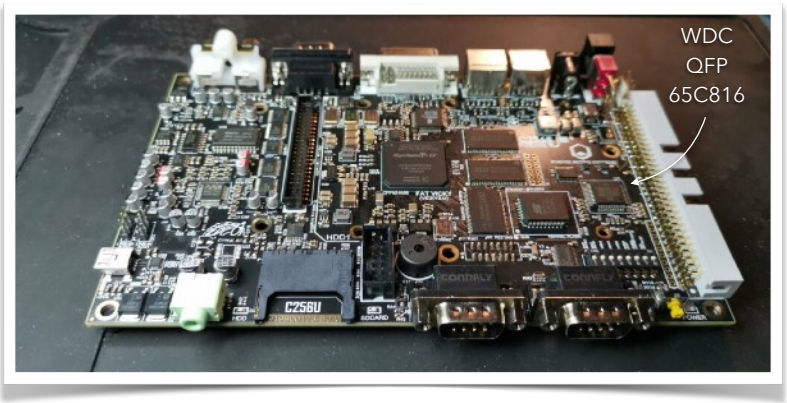
The machines in the middle: The Foenix C256U / U+ lead to the A2560 family

C256U

The everything-and-the-kitchen-sink feature list of the Foenix FMX was the best of times, and units were finally shipping in quantity; then COVID induced supply chain issues struck.

The FMX leveraged a diverse set of curated ICs including multiple ALTERA FPGAs and audio ICs that were suddenly difficult to find or prohibitively expensive.

In response, Stefany spun-up a new lower-cost "User" model, named the 'C256U'. The 'U' only required a single FPGA, did not leverage the SuperIO IC, had a limited set of physical Sound ICs, did not have MIDI, and had other limitations. But it enabled Stefany to keep shipping product and to put her technology in the hands of more people.



C256U - FMX heritage, WDC 65816 CPU @ 14 MHz. reduced chip count and lower cost

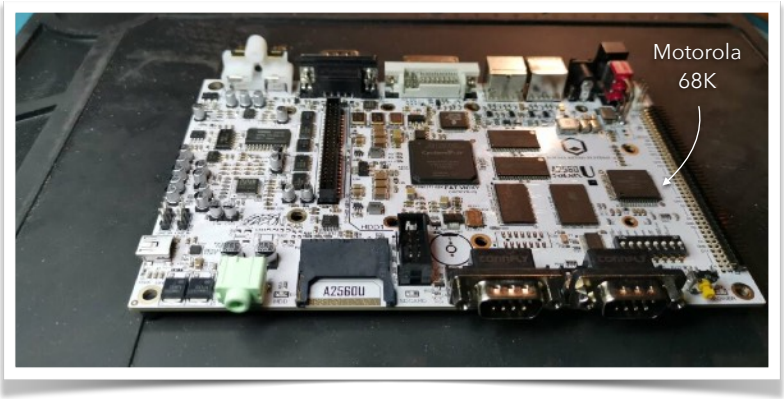
The C256U (pictured to the left) was also significant because it opened the door for use of the 20 MHz Motorola MC68SEC000 microprocessor (next page).

This was the first time a non MOS-lineage CPU was used in a Foenix design and at the time, it seemed like an interesting product, attracting a new group of retro enthusiasts to Foenix.

A2560U

Coined the A2560 series, the 'U' version (see pic on pg. 6) would lead to an integrated Keyboard 'K', an A2560X desktop version with a single 68040, and dual CPU switchable machine which could run a WDC 65816 and have a Motorola CPU (or Intel) footprint onboard. But we are getting ahead of ourselves; one step at a time.

* Rev A of the C256 Jr. dev board features both CPU footprints (the QFP pads are under the DIP socket) see pgs. 26-29



A2560U - sister to the C256U but leveraging a Motorola MC68SEC000 @ 20 MHz.

My own C256U+ paired with a custom WASD PS/2 Keyboard styled to match the A2560K keyboard

Both ‘U’ versions (C256 and A2560) were available at two price points; the 'U' had 2MB of system RAM, and the slightly more expensive 'U+' doubled the memory to 4MB.

To round out these offerings, a multi-piece 3D printed case (below) was designed, featuring register addressable LED lighting.

While many cases were sold, vendor manufacturing delays caused a backlog and as a result, many SBC (single-board-computer) versions of the Foenix ‘U' exist in the wild, including the C256U+ that I use for production and development of this newsletter (below, left).

C256 Family Operating Environment (aka OS)

All C256 family machines produced thus far ‘boot’ into BASIC816, a built-for-purpose BASIC language/environment that included a toolbox of graphics and memory access/DMA shims, a machine language monitor, and several ‘disk' commands for accessing either the SD card (all models), or the built in disk, for machines equipped.

Similar to early Commodore products, the C256 machines allow ‘open editing’ of BASIC code, meaning that you can move around the screen unencumbered up to the four corners of the screen.

Upon pressing enter, the text line under the cursor is interpreted in immediate mode, or ‘entered’ / tokenized, if beginning with a line number.

Though token based, programs saved by BASIC816 are written in clear text; handy for shuttling or editing on other platforms, including the IDE. This comes at the cost of processing time during load.

As full featured as it is (especially for graphics), BASIC816 is otherwise sparse. There are no ‘modern’ BASIC features (such as line renumbering) and no sound/music commands. And there are a few peculiarities to manage. But it works and as you’ll see in the Sprite Editor review below, it’s a perfectly feasible operating environment for lightweight work which is all that you can really expect from BASIC. Having said this, other versions of BASIC are under development and likely to be ported to other platforms; more to come here. But what about the kernel? We will need to save that for another time.

Shipping Today: The Foenix A2560K - Integrated Keyboard Model

A2560K

With the A2560U and U+ nearly complete, Stefany took an ambitious step forward to continue work on an integrated keyboard-based Foenix developed around one of her favorite processors, the Motorola 68040. Announced and shown in prototype form (over Zoom) at the October VCF East show in 2021, the A2560K has shipped its first 3 hand built models and the next four P&P built machines (see last issue), with a few dozen more presale units being assembled now.

The A2560K was featured in last month's newsletter both in pictures and in text, and we interviewed Peter Weingartner, MCP Kernel developer. So head to the Foenix Marketplace and download issue #1 if you have not yet done so. For a video tour, have a look at Peter's overview video. It is very well done.

Stefany describes the A2560K as "the computer I designed for myself", and it shows. It embodies features of earlier Foenix systems, design elements from prior generation products, and a look forward; with new capabilities and a rugged look.

In addition to a densely packed audio section footprint (physical ICs plus several FPGA based instances), the 'K' includes DIN based MIDI IN/OUT, the SuperIO IC with full featured serial ports, a parallel port, floppy and hard disk, RJ45 based Ethernet, and the next iteration of VICKY (video) including native dual-head video output.

All of this is supported by 64 MBytes of SDRAM, 4 MBytes of static RAM and an additional 4 MBytes of FLASH stuffed into a compact and attractive case inspired by the Amiga A600 and its ANSI keyboard, released in 1992.

A2560 Operating Environment (the MCP v1.0 release)

Unlike the C256 systems discussed thus far, A2560 machines boot to a traditional command-line interface with a handful of embedded commands for manipulating devices, for interrogating memory, and to test aspects of hardware.

There is also a micro-kernel interface which provides ~70 C library functions which can also be accessed via MC68000 assembly Trap#15 calls. An example of a C function is: `sys_txt_set_color(0, 6, 0);` which sets the cyan text against a black background on screen #0. Have a look at the github hosted repo here -> MCP for code and a .pdf doc.

MCP's shell also has a disassembler, `PEEK` and `POKE` commands (for 8, 16 or 32 bit values), and the ability to `type` a file to the screen, among other functions. In short, it's a combination of some of what Apple and Commodore offered, some of what CP/M and early DOS provided, and support for Foenix specific features.

Here is a selection of some of the available commands; you'll probably recognize many of them:

```
CALL {addr}   CD          CLS          COPY         DUMP         DIR          DISKREAD {sector}
GETTICKS      LOAD        POKE16       PWD          REN          SHOWINT      TEST {feature}
```

red (160) blue (160)

Just for fun, issuing "`POKE32 $fec0:0008 $00A0:00A0`" on a A2560K will change the power LED from green to a lovely lurid color. Blue more your thing? Try an argument of `$0000:00A0`. As Peter explained when interviewed in Issue #1, the intent of MCP was to strip away the layers and only provide the necessary primitives needed to manage the machine at a high level (or a low level, as the case may be).

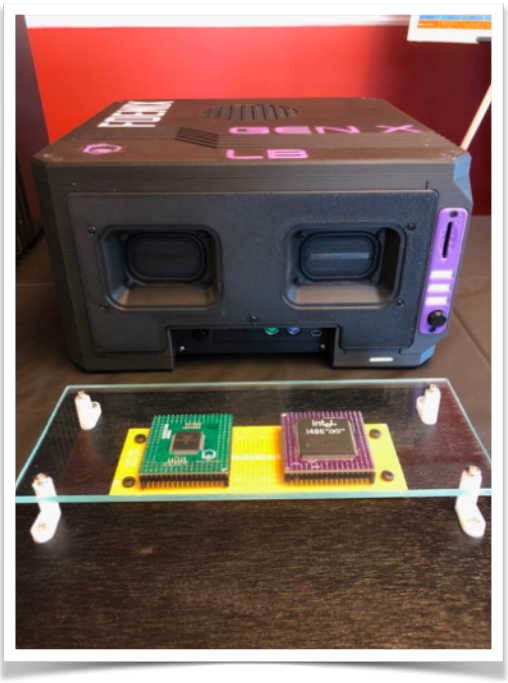
And similar to DOS or CP/M if you want to add a BASIC interpreter or a Text Editor, just drop the binary onto disk and MCP will search PATH to consider other "commands".

The A2560U, on the other hand, runs the predecessor to MCP (also developed by Peter), but MCP is being ported to the platform. This has not gotten in the way of software development or the availability of apps. The A2560U has an Atari ST (TOS) environment in active development, a MIDI note proof of concept, a BASIC language interpreter (and other languages), and a text editor with context highlighting and IDE capabilities, with more to come. There is also a virtual/IDE environment. With the first four production units shipped and received, I expect these pages to have much more A2560 content by the end-of-year and our fingers are crossed for a good Christmas Demo.

Shipping soon: The C256 GEN X Family

C256 GENX

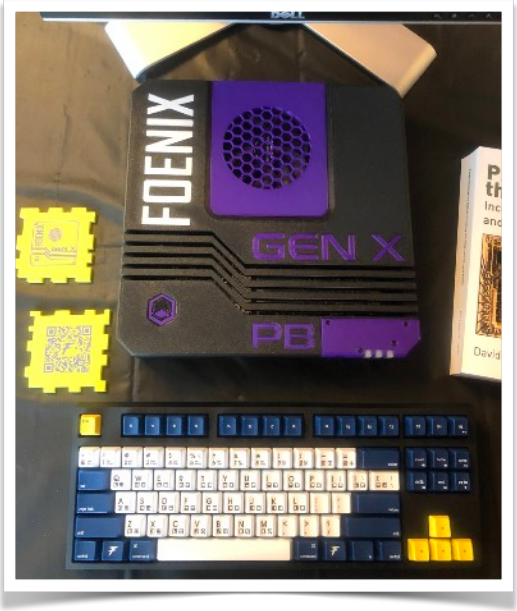
The GEN X is a multi-CPU capable, new-retro computer that ships with the WDC 65816 component-set embedded on the main board. This means the GEN X is ready to run the growing library of FMX and C256U/U+ programs.



And with the addition of a second processor (such as a MC680x0 processor or an Intel i486/DX2), you can leverage a secondary power-up mode and transform your GEN X into a different machine, with a different OS and kernel, but with the same hardware capabilities and I/O including MIDI.

The GEN X is available in 3 form factors, beginning with the 'Cube', an 11" x 11" x 11" enclosure. The Cube comes complete with a floppy drive, a hard disk, an internal power supply, and partially populated audio ICs. Similar to many of the other Foenix machines, there are also two sockets for SIM ICs which may be either 9V or 12V and can be bound to right, left, or a common audio channel.

Importantly, every GEN X model has two FPGA based Gideon SID instances in addition to a plethora of Yamaha TMS support which means that apps developed to use the onboard (emulated) sound will work across the range or systems down to the PB and likely, backwards to the C256U and FMX.



Next up is the 'Lunch Box' (LB) shown in the picture to the left. This picture was from the first public showing of the LB and you'll notice that it shares a lot with the Cube, occupying the same 11" x 11" footprint, except 1/2 of the height, at 5.5" tall. Absent is the floppy and some of the physical audio ICs, but otherwise, it is functionally identical to the Cube, same ports, memory, processor options, and build in amplifier, speakers, and controls.

Finally, there is the 'Pizza Box' (PB). Similar to the Lunch Box, except less expensive still, and smaller (< 2.5" tall). You'll notice that the distinctive stereo speakers are no longer onboard, nor are direct connect NES/SNES game ports (however, the Pizza Box does host 4 x Atari Joystick connections, two of which can host analog paddles or an external break-out box for NES / SNES controller connectivity).

Shipping next: The A2560X Family

A2560X

The A2560X, briefly mentioned above, looks and smells like a GEN X, however, is a single CPU machine and is powered by a 68040 @ 25 Mhz. It leverages the GEN X circuit board and shares input/output specifications, but a different population of ICs and is absent some of the unnecessary supporting infrastructure. At a high level, it wouldn't be incorrect to consider the A2560X a desktop or workstation version of the A2560K, but there are some differences.

The A2560X is in pre-order phase and is expected to ship in the next few months.

In Summary: Shipping and soon-to-ship products

There you have it, a brief but high-level history of Foenix Retro Systems. If you have the time and interest, there are literally dozens of hours of video available beginning with an early CRX presentation, live streaming events and updates, not to mention handful of seminars from Vintage Computer Federation (VCF) shows.

Head over to the C256Foenix.com home page for the latest information and see pgs. 26-28 below for a closer look at the C256 Jr. project.

Click here to head straight to the full specifications page.

App Review and First Look

Ernesto Contreras "Foenix Sprite Editor" for C256 platforms

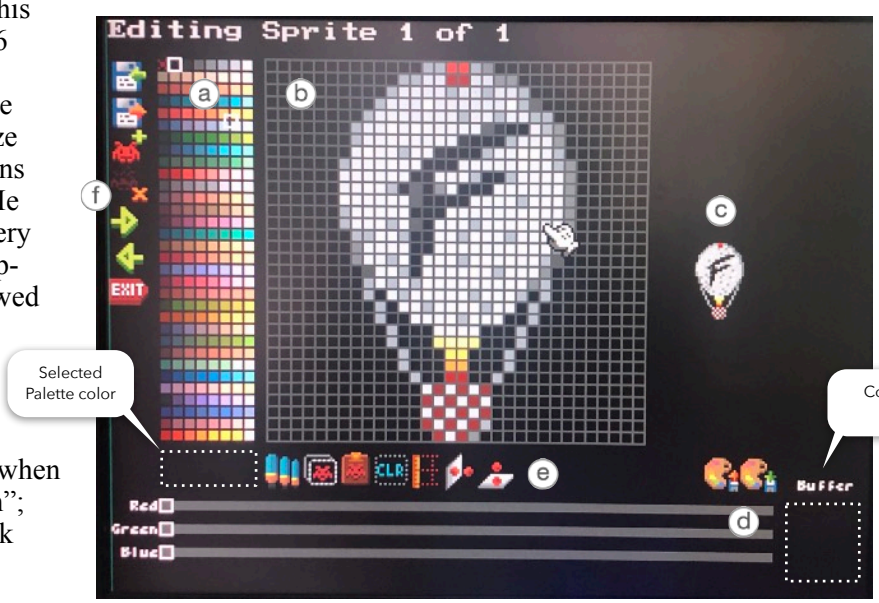
The first installment of *Beginner's Corner* recalled the "Up, up, and away" example from the original Commodore 64 User's Guide and passing reference was made to the old school way to design Sprites: graph paper.

In the early 80's the word "Sprite" was a new term but it didn't take long for computing journals to publish tools which allowed the user to move the cursor around a field of cells, one character space per pixel, to define the C64's 24 pixel by 21 pixel Sprite. Watching your Sprite take shape at high resolution to the right side of the screen was magical, and some of the editors provided functionality to double-size, bit shift, and x/y mirror.

The very last memory I have of my original Commodore 64 was selling it to my High School friend Jerry; it was complete with my 1541 Flash! equipped drive, an IEEE-488 interface and SFD-1001 disk drive, a standard RS-232 user port interface, the works. And the last program I demoed for him was not a game or EasyScript, but a Sprite editor; and I remember flying around the screen and toggling pixels on-and-off, amazing my friend. And yes, I regret selling it, to this day!

As mentioned last month, the first time I *felt* productive on this platform (a C256U+) was using the ***Foenix Sprite Editor***, the subject of this column.

Ernesto developed his editor in BASIC816 along with a few passages of machine language to optimize some of the functions in need of speed. He also pre-loaded a very useful color look-up-table (LUT), borrowed from from LOSPEC.



Load the app with a LOAD "SPREDIT.BAS" and when complete, type "run"; you'll notice a quick video mode shift, followed by a grid rendering reminiscent of the old HES GridRunner game (but cooler).

Everything is managed and edited from a single screen. Notice (clockwise): (a) the color palette on the left, (b) the Sprite editing field, (c) the rendered Sprite on the right, (d) palette load/save and custom color sliders. Below editing field is (e) a set of Sprite editing tools, and finally, (f) main program functions for (top to bottom) saving/loading, adding/deleting, moving to next/previous Sprite, and an 'exit' to BASIC. A status line above indicates the Sprite number being edited (from 1 to 255). I've highlighted two areas (dotted regions) which show the selected color and the copy/paste buffer contents, used for copying a Sprite to a new workspace. This is useful when wanting to try alternate coloring schemes or editing multiple frames for animation.

How to use: Use the mouse to select/deselect pixels and colors from the choice of 255 (plus transparent) on the left. You may also custom 'mix' your own colors via positioning a set of three horizontal sliders at the bottom of the screen. Doing so will adjust the RGB range from 0..255 and you'll see the new color take its place in the palette selector and in the large annotated rectangle box just to the left of the tools.

One enhancement that I would like to see is for the color bars to 'jump' to the value of the selected color; doing so would allow the user to easily tweak a hue.

Saving Sprites for later use is as simple as clicking the picture of the disk with the left-facing green arrow, and providing a filename. All Sprite data is written to a single binary file, with 1024 bytes per Sprite definition + one leading byte at the start of file representing the number of Sprites within. Note that this is not a PGX or PGZ file so you will need to 'load' the file into system memory at a given address before moving to VRAM (will will discuss this in Beginner's Corner on pg. 25 below).

What I like most about the editor: It is easy to use, mouse driven, comes loaded with a great palette of colors which reduces toil, allowing the user to get on with the creative process. It's also very easy to use the resulting data either with a handful of BASIC816 commands (see last and this issue's "Beginner's Corner") or within your own assembly language routines.

What's missing? A pixel shift (moving the whole field or a selected region bitwise) in the primary directions would be nice, especially useful when duplicating Sprites for animation. A smooth, or *n* (9 or 18) degree rotate would be icing on the cake. (Ernesto does provide a 180' *x* and *y* flip and it works well). I may be pitching in to help develop some of these enhancements in future.

What I *really* like most about the editor? The fact that it is written it BASIC and is therefore easily extensible for practical use or to learn from. Also, Ernesto has inadvertently set a standard on how to organize Sprite and color LUT (palette) data, including a file format for multiple Sprites. If you have a look at the "BUG13.SPR" file included within the "BUGDEMO" distribution (on the Foenix Marketplace), you'll begin to realize that this is the beginning of a growing library of content.

An alternative?: There are many possible alternatives (not counting graph paper) to edit and manage Sprites and Color LUTs, and they include familiar titles such as GIMP and Unity and lesser known titles such as Piskel, Aesprite, LOSPEC, Gamemaker, PLAYCANVAS, and more.

Many have video tutorials specifically for Sprite use (some are generic images editors), most are free, and some are open source and go to great lengths to make the source code available such as this GNU example.

Ultimately, you will need to get your Sprite data into binary format and onto a Foenix platform. You may opt to dive into each tool to learn about its native format, explore conversion routines that manage color palette export as well, then write some Python code or maybe some shell scripts. Or (*punchline*), use this native tool, and spend more time working 'on platform' to make the world a better place.

Related topics and further discussion: In last month's issue, we mentioned leveraging Sprites for non-movable objects and at the time, we were referring to the use of Sprites for buttons on a tool bar. The thought was that a clicked-upon control could easily be animated to appear 'pressed' or in a shadow. Another approach could be to render (move) a single Sprite down and to the right a few pixels, complete with sound effect or better, *a serial port connected haptic clicker hack!*

If there is one thing, aside from the obvious 'everything' that Foenix platforms improve upon, it's the number of Sprites and amount of memory available for user programs and data. So it is not much of a stretch to consider using Sprites within utility apps or lightweight educational games; and next month we will do just that.

Something else stumbled upon between publishing the prior issue and starting to write this issue was a happy accident brought about while troubleshooting the FPGA code that I had loaded into my machine that I suspected was causing problems (it was). I actually had two somewhat nonstandard pieces of code in my machine; a newer (and less stable) FPGA load that supported the EXP-C100-ESID (dual SID and CAT5 Ethernet expansion card), and a custom build of a kernel that supported a UDP listener.

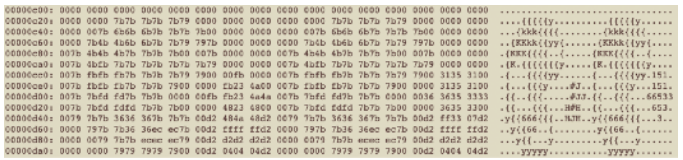
It had been months since I used this hardware and upon trying a new WASD keyboard, I noticed some odd behavior during editing BASIC816 programs. For a while I lived with it. But ultimately (while using the Sprite Editor) I experienced a lockup.

To my chagrin, I had invested a good hour or two editing Sprites and failed to save them, but I had a bright idea !!

All Foenix platforms come equipped with a USB debug host interface which can be used for content (.hex or binary) push, to command the CPU, or in this case, to interrogate memory.

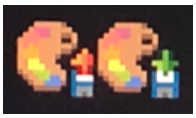
You see, regardless of the state of the CPU, the USB port combined with FPGA code is autonomous and has access to memory; so I investigated pulling a region of memory (observed in Ernesto's BASIC code) that I thought might have the 1K Sprite segments I was after.

It didn't work. As it turns out, VRAM (Video) is a different block of memory and is not available to the debug interface; but in poking around, I stumbled upon something interesting that I hadn't noticed prior. Have a look and **see if you see what I see?**

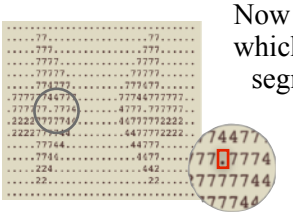


This is the output of a `xxd -c 32 {filename}` executed at the `bash` shell of a MacOS terminal. The significance of the "`-c 32`" represents what might be referred to as the 'stride' of the image aka 32 bytes (1 byte per pixel) Combined with the precise starting address, we see a clear image instead of indiscernible garbage (the worst kind of garbage). More on '*stride*' in *Beginner's Corner* further down in this newsletter).

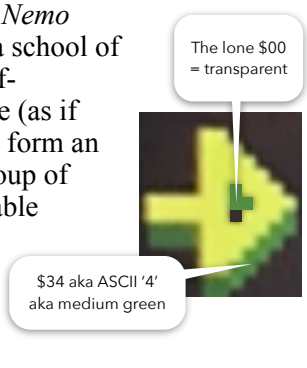
No, it's not a kidney-bean, these are tool icons from Ernesto's editor and specifically, shape definitions (packed into the native 32 x 32 pixel Sprite).



Load the file "ICONS.SPR" into the Sprite Editor to see for yourself. You wouldn't be incorrect to call these 'tiles' but this has nothing to do with VICKY II tiles.



Now consider the example to the left which reminded me of the moonfish segment in Disney's *Finding Nemo* where a school of fish self-organize (as if pixels) to form an arrow, or in this case, where a group of 7's, 4's, 2's, and a single unprintable character (hex $00 or "[.]" above) defines the 3D arrow on the right (Sprite Editor - "Next Sprite" button).



It takes more than a bitmap definition to make this look good. You also need the correct color palette. Otherwise, color $37 (aka #55 aka ASCII '7') and color $32 might default to black, while $34 is set to brown; sounds innocuous enough, but on a black background, this would look more like an angry eyebrow.

So don't forget the color palette; you can either load, run, and exit Ernesto's Sprite Editor; or load and run last issue's "`I01P11A.BAS`".

The code excerpt below was borrowed from Ernesto's Editor. It loads his Sprite definition file, and displays the 16 x 16 bit green arrow portion of the 32 x 32 sprite.

```
100 GRAPHICS 47+512
110 BITMAP 0, 1, 0, &hB00000 : SY%=5
120 SETBORDER 0:CLRBITMAP 0
3510 BLOAD "ICONS.SPR", &h100000-1
3610 IPOS%= &hB00000 + (SY%*15) *320
3620 MEMCOPY RECT &h100400,16,16,32 TO RECT IPOS%,16,16,320
```

After execution, you'll need to find a blank line to type "**GRAPHICS 47**" on in order to get back to a usable text screen; the "**+512**"on line 100 pushes VICKY II into 320 x 240 (double pixel) mode where things get messy fast.

Thus concludes another use of the Sprite Editor: for creating and managing bitmap content that has nothing to do with movable objects.

But wait, there's more! Add the following two lines:

```
3505 FOR x%= 1 to 200
3630 NEXT
```

… and then add a "**+ x%**" to line 3610 so it looks like:

```
3610 IPOS%= &hB00000 + (SY%*15) *320 + x%
```

... and run it again. You should see the following:

Pretty cool, huh? Recall that this is NOT a sprite. 'Yes', Ernesto used his Sprite Editor to create the arrow; 'Yes', we loaded it into memory, then to VRAM as if we were going to instantiate a Sprite; and 'yes', we are using the same color look-up-table (aka palette) used for Sprites, but we are, in fact, redrawing or *re-stamping* the shape 200 times to move left to right.

Important to point out that it *appears* to be moving smoothly, cleanly, and not 'smearing'. And all of this is true. But we've taken zero care to make this happen, yet it is behaving like a Sprite.

So what's going on? The leftmost column of pixels contains hex $00 which, based on LUT rules is 'transparent'. This is the single reserved location in each LUT that may not contain color. We could have chosen color $01 which (in Ernesto's table) is legit black; but doing so would paint a thick black stripe across a screen that might otherwise tarnish a lovely bitmap image, or soil a sky blue background. This is better.

Said another way, the bitmap shape, due to the way that it leverages colors, cleans up after itself. This should remind Apple II fans of working with *Shape Tables*.

One famous monochrome Apple II bitmap graphics trick is to exclusive-or (the 6502 opcode is `XOR`) a shape pixel with a background resulting in an 'only'-or situation. If both pixels are 'on', a reverse bit will result.

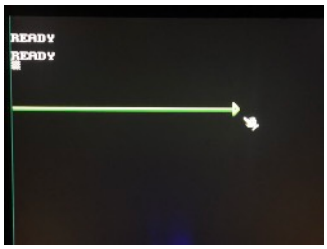
Also worth mentioning, that this is a fair amount of work for a lone CPU; far more than increasing a small number of registers for Sprite movement. Early 80's computers often had a single, built-for-purpose IC for '*assist*' (e.g. the VIC-II). The CPU was otherwise lonely. Foenix platforms have VICKY for graphics, sound, and other functions, not the least of which is a DMA engine.

Every iteration of the loop (all 200 of them), not only costs cycles in BASIC interpretation of the FOR / NEXT loop itself, but also computational time to solve the math equation on line **3610** and also the **MEMCOPY** (and the cycles it requires) on line **3620**. But at 14 Mhz, this is faster than a Commodore 64 moved a single Sprite in BASIC, so at least we have that going for us :)

Something else to try. Let's modify this code slightly to discount the left column of pixel and see what happens. We can do this by modifying a single statement:

```
3620 MEMCOPY RECT &h100401,15,16,32 TO RECT IPOS%,15,16,320
```

Now run this again and see the following:



There's the *smear* we referred to previously.

One last trick; let's push this out 4 more pixels. You can probably guess what type of effect we are going after here.

```
3620 MEMCOPY RECT &h100405,11,16,32 TO RECT IPOS%,11,16,320
```

That's all for now. Next month, we will leverage a few of these techniques in a simple game.

We will also discuss a few Sprite Editor enhancements then see if we can actually code them. Should be fun !!

✦ We dive deeper into **MEMCOPY** and related graphics topics in *Beginner's Corner* on pgs. 18-25 below.

Installing the Calypsi "C" Compiler in Windows

Written by Ernesto Contreras

Introduction

So you own of a Foenix FMX, Foenix U+, or a Gen X?, first of all Congratulations on your acquisition!, this is an exciting retro computer that is just waiting to be explored and exploited!

As a new user your first exploration of the capabilities will be with the default BASIC included in the machine, but as exciting as BASIC is, some programs and specially games need a bit more Oomph to really shine, so what we need is a compiler to access the real power of the machine.

There are a few available compilers for this processor (for example 64Tassfor assembler or the WDC provided Compiler for C), but in this guide we are going to concentrate on how to install the Calypsi WDC 65816 Compiler & Tool Chain on Windows and do a quick test to compile the demo **Hello World** program.

Why Choose Calypsi?

Calypsi is not one, but a series of C and assembly language cross compilers (there are versions available for the MOS 6502 / WDC 65816 and Motorola 68000) & tool chains aimed towards the retro and hobby communities by Håkan Thörngren.

Its targets are conveniently matching the current offering of processors of the Foenix Series of computers by Stefany Allaire, so by learning Calypsi you could have one compiler & toolchain that targets the complete series.

Other highlights of the compiler's features:

- ISO C 99 compiler. This is a freestanding implementation with many features you will typically find in a hosted compiler.
- Fully re-entrant code model.
- Support for all integer types up to 64 bits `long long`.
- Floating point supported (32 and 64 bits IEEE-754).
- Full support for `struct`, `union`, `typedef` and what you expect to find in C.
- Support for (stack allocated) variable sized arrays.
- Optimizing compiler that can output source level debugging information.
- Source code debugger included.
- Support for ELF/DWARF, hex output as well as various target specific output formats.

Installing the Calypsi Tool Chain

To get the compiler & tool chain head to the official page for ***Calypsi Tool Chains:*** https://www.calypsi.cc/

a. Scroll Down on the page to the WDC 65816 target and download the following files:

   i. ***Hello World for C256U***

   ii. Download the ***current user guide***

   iii. Download the ***Windows*** version of the compiler

b. Install the downloaded file ***"Calypsi-65816-3.6.4.msi"**** by double clicking in the file and follow the on screen instructions to complete the install.

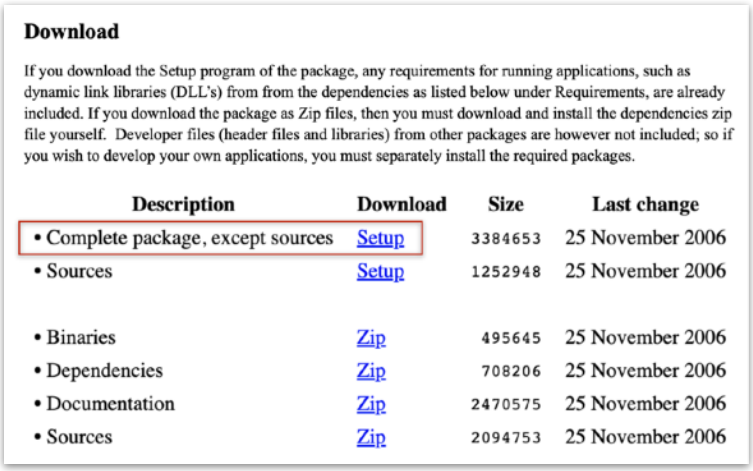
** current version as of this guide is 3.6.4*

Installing GNU Make for Windows

Since the examples provided with the "Hello World for C256" use a Makefile to control the compiling and linking of files you need to download a Make utility, if you don´t already have one installed I suggest to use the GNU ***Make for Windows*** utility that you can get from:

http://gnuwin32.sourceforge.net/packages/make.htm

a. Scroll down on the page and Download from the option: “Complete package, except sources”.



Download

If you download the Setup program of the package, any requirements for running applications, such as dynamic link libraries (DLL's) from from the dependencies as listed below under Requirements, are already included. If you download the package as Zip files, then you must download and install the dependencies zip file yourself. Developer files (header files and libraries) from other packages are however not included; so if you wish to develop your own applications, you must separately install the required packages.

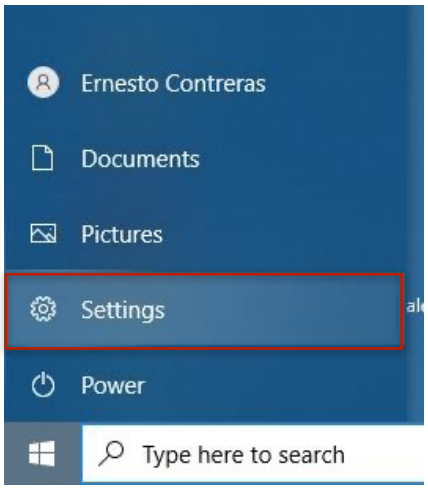
Description	Download	Size	Last change
• Complete package, except sources	Setup	3384653	25 November 2006
• Sources	Setup	1252948	25 November 2006
• Binaries	Zip	495645	25 November 2006
• Dependencies	Zip	708206	25 November 2006
• Documentation	Zip	2470575	25 November 2006
• Sources	Zip	2094753	25 November 2006

b. Install the file “make-3.81.exe” by double clicking on it and follow the on screen instructions to complete the install.

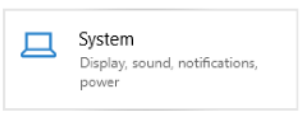
Configure the PATH to Bin directory of the Calypsi compiler

The Calypsi compiler is not automatically added to the Path and you need to do so if you want make to be able to find the compiler and linker from the directory of your projects, to do so follow the instructions below:

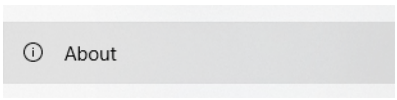
1. Click on the Windows Start Icon and click on the Gear Icon to enter ***”Settings”*** :



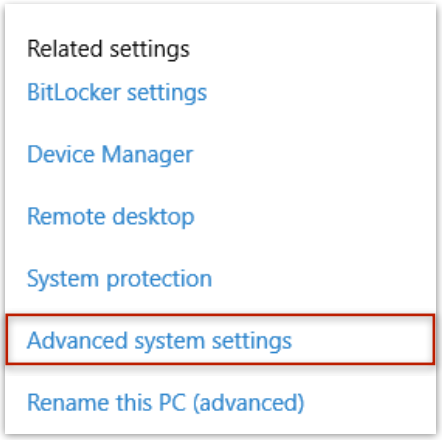
2. On Settings click the “***System***” Option



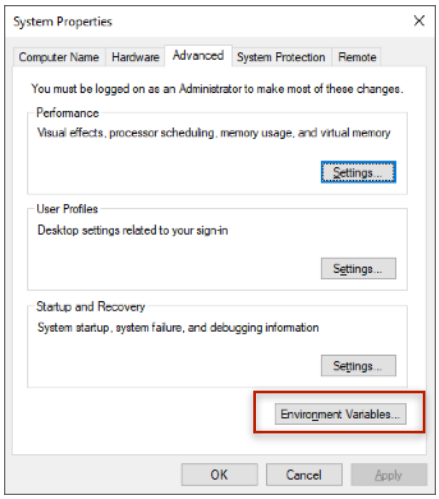
3. On the System Settings page click on the “***About***” option from the left Menu:



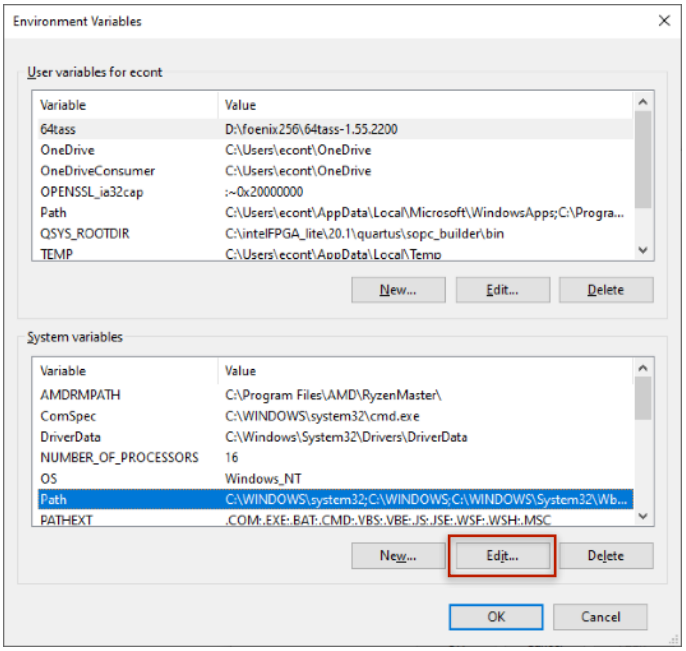
4. On the “About” page click on the option “***Advanced system settings***” that appears on “Related settings” on the right side of the screen:



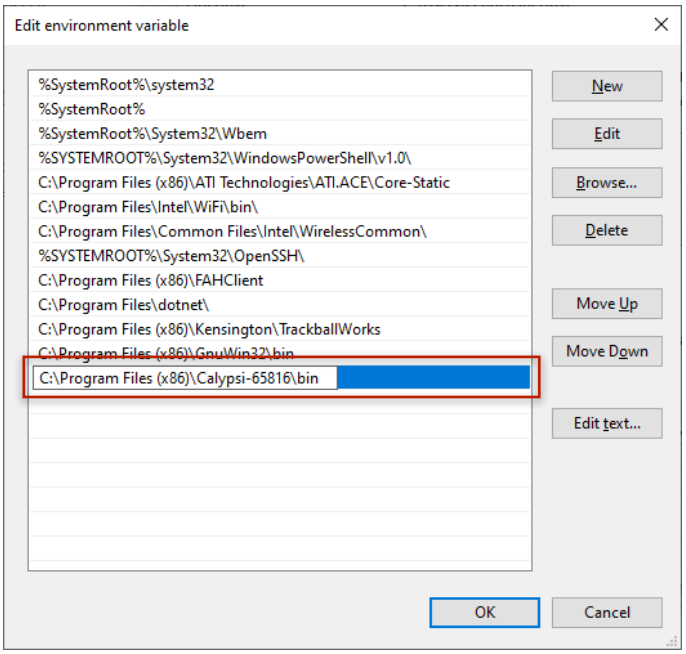
5. You should’ve made it into the “System Properties” dialog, please locate and click on the “Environment Variables...” button (at the bottom of the dialog form)



6. Search through the items in the "***System variables***" list (the bottom list). Click on the "***Path***" variable and press the "***Edit***" button:



7. Click on the "New" button and add the path where the "Calypsi-65816\bin" directory is installed (usually "C:\Program Files (x86)\Calypsi-65816\bin" but please verify it!)



8. Verify if a Path to the "***make***" utility was added to the environment variables (usually "***C:\Program Files (x86)\GnuWin32\bin***"), if it's not there please add an appropriate entry following the same steps that you just used to add the "*Calypsi-65816\bin*" path.

9. Press "Ok" to close the "Edit Environment variable" dialog.Close all other dialogs.

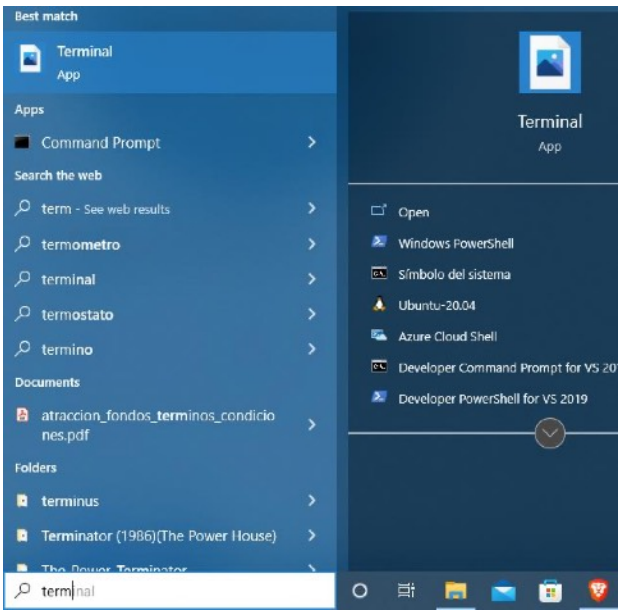
10. Close all other dialogs

11. The configuration is now complete!

Testing Hello World for C256U

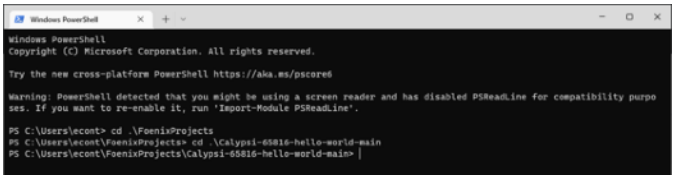
1. Unpack the zip file from the "Hello World for C256U" into a directory

2. Open a windows terminal, I recommend the Windows "Terminal App", to find it click on the windows start Icon and type ***"Terminal"***

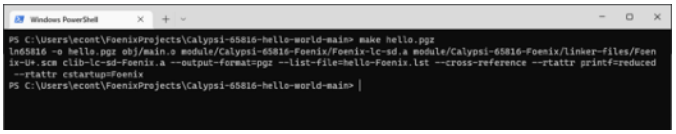


3. Open the terminal program (and I suggest that you pin it in the taskbar to locate it easily!)

4. Move to the directory where you unzipped the "Hello World for C256U" file



5. Type "make hello.pgz" to build an executable PGZ file



If all goes well the "hello.pgz" will be compiled in a zinch!

6. Copy the program to an SD card to try it on your Foenix Computer.

7. To execute the Program in your FOENIX type:

```
BRUN "hello.pgz"
```

Your Foenix Computer should respond:

```
Hello World!
```

Congratulations, Calypsi is installed and working!

Recommended Next Steps

1. Read the ***user guide*** that comes with Calypsi and asked you to download, it is indeed a very comprehensive document that will help you understand the compiler, linker and all its functions.

2. If you find the compiler useful, please support the Author, there is information on how to do donations though Paypal on the web page https://www.calypsi.cc/

3. Stay tuned for more Guides!

F.R.

Library Look-see - VickyGraph

by Ernesto Contreras

VickyGraph is a C library for Foenix C256 systems, developed for use with Håkan Thörngren's Calypsi C compiler. The VickyGraph library was written by Ernesto Contreras.

Base functionality includes:

- Initializing the VICKY II graphics engine
- Selecting one of several supported display modes
- Drawing, and to a lesser extent, reading from the bitmap display via a set of graphic primitives.

The VICKY II FPGA contains a powerful graphics engine which supports Sprites, Tilesets, Color LUTs and other features however, from a bitmap graphics perspective, it's a blank canvas. Said another way, there is no support for graphic shapes or plotting; the user must directly address memory in order to have graphics rendered. This is where VickyGraph comes in.

The following list details the objective and usage for each of the 11 provided graphic primitives:

Erase Bitmap
Objective: clears the current bitmap page
Usage: `clrBitmap()`

Point
Objective: sets pixel at (x,y) with color (col)
Usage: `plot (x, y, col)`

Line
Objective: Draw a line from (x0,y0) to (x1,y1) using color (col)
Usage: `plot_line (x0, y0, x1, y1, col)`

Rectangle
Objective: Draw a hollow rectangle from (x0,y0) to (x1,y1) using color (col)
Usage: `plot_rectangle (x0, y0, x1, y1, col)`

Filled Rectangle
Objective: Draw a filled rectangle from (x0,y0) to (x1,y1) using color (col)
Usage: `plot_solid_rectangle (x0, y0, x1, y1, col)`

Bezier Curve
Objective: draws a curved line from (x0,y0) to (x1,y1) using color (col).
Usage: `plot_bezier (x0, y0, xc0, yc0, xc1, yc1, x1, y1, col)`

Circle
Objective: draws a hollow circle onscreen centered on (x,y) with radius (r) using color (col)
`plot_circle (x, y, r, col)`

Filled Circle
Objective: draws a solid circle onscreen centered on (x,y) with radius (r) using color (col)
Usage: `plot_solid_circle (x, y, r, col)`

Ellipse
Objective: draws a hollow ellipse centered on (x,y) with width (a) and height (b) using color (col)
Usage: `plot_ellipse (x0, y0, a, b, col)`

Filled Ellipse
Objective: draws a solid ellipse centered on (x,y) with width (a) and height (b) using color (col)
Usage: `plot_solid_ellipse (x0, y0, a, b, col)`

Fill Area
Objective: fills an area of pixels that share the same color as the pixel at (x,y) with a new color (col)
Usage: `floodFill (x, y, col)`

Public repository: https://github.com/econtrerasd/VickyGraph

We will be putting Ernesto's VickyGraph library to the test in a future issue of Foenix Rising. This has been a *Library Look-see*.

F.R.

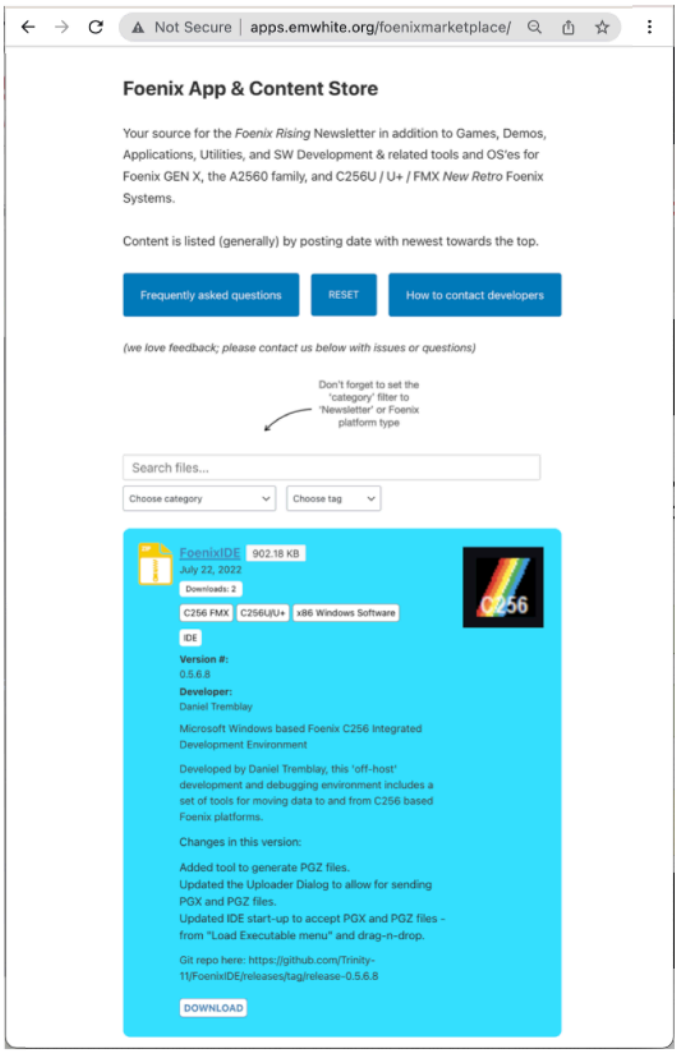
Quick take - A look at the Foenix Marketplace

The ins and outs of downloading software for Foenix platforms

In late July, following release of issue #1 of this journal, we launched the Foenix Marketplace. The Marketplace is more of a *content repository* than a hub for commerce, in fact there is no commerce permitted. Here is a quick Q&A style overview.

Where is it? In a secure location in Northern Virginia at a famous Cloud provider hosting site. But you can access it at: http://apps.emwhite.org/foenixmarketplace

What it is? A web browser accessible site to download binary coded apps in .hex, .pgx, and .pgz formats (including bitmap image and Sprite data); it is also home to *this* Foenix Rising Newsletter and includes code examples (primarily, short BASIC programs from our "Beginner's Corner" column); It is also a place to download IDEs and Utilities.



Each file has a representative 'profile' which identifies the version number, developer, a brief description, and (where available) a thumbnail screenshot and links to a github repo.

At some point, we hope to support native downloads directly to Foenix platforms. While we do not anticipate running a full TCP/IP stack on Foenix, there are a few ideas kicking around to potentially relay content between an internet connected modern host and your Foenix.

What Foenix platforms are supported? At the moment, most of the content in the Marketplace is for C256 systems (FMX and the C256U/U+) but we will actively be adding software for other platforms (including the A2560 family) at the end of August and into September.

How are files organized?

Content may be filtered and/or selected 4 ways:

a) Free browsing (*default*) - when first connected, there are no filters in place and content is paginated. The bottom of each screen hosts the page picker from pg. 1 to pg. *n*. Scroll, choose, and click to download. (pagination only works in 'free browsing' mode).

✦ The color scheme of the Foenix Marketplace evokes the cabinet palette of the original Nintendo Donkey Kong Arcade machine; cabinet build forums cite color recipes and Sherman Williams paint formulas; We settled on RBG 33E2FF.

b) Select by category - a common use of *category* is to focus by platform such as “FMX”, however, there are categories that are not platforms at all such as“Newsletter”, “Sprite”, or “Windows software”.

c) Filter by tag - a *tag* is a genre or an attribute that might be a “Game” or a “Demo” or include “.PGZ”. Most files have more than one tag but only one tag may be selected for filtering.

Therefore, Category = “C256U/U+” and Tag = “Game” will yield all games for the identified platform.

d) Full text search by keyword(s) - will search across filenames, descriptions, and developer names, without prejudice. Search terms may be any part of a word or multiple words however, multiple words (spaces permitted) must be an exact match. As you type characters, the number of matching files will be dynamically updated. CAUTION: Using text search with Category and/or Tag is *unreliable*; when in doubt, click on the “RESET” button to re-load the form.

How often is new content added?

Generally monthly, but as more content is published, the frequency of updates will increase. We are just getting started.

What is the default ‘listing order’ of content? Can I request to see ‘new’ only?

Content is generally listed as ‘newest’ first, but there are some exceptions. As explained in the FAQ, there is no user tracking or cookies; downloading is completely anonymous; so there is no way to notify or alert that something new has been added. We may consider such a feature in the future. We are trying to keep it simple for now.

Why are some files .zip’d?

Some content is packaged within platform independent .zip files. Of course, there is no native Foenix zip/unzip/pkzip (yet). But some apps are more complex than a single .PGX or .BAS. An example of this is the Foenix Sprite Editor discussed elsewhere in this Newsletter which includes assembly source files, a ‘make’ batch file, assembled machine code, a sample Sprite and other files within a .zip file.

I am a developer, how can I publish code to make it available for others?

Reach out to `EMwhite` via the feedback link, see page 2 (above), or find me on the Foenix Retro Discord channel. We love not only complete and working apps, but demos, partially working apps, amusements, and code examples. Everything helps the cause.

In closing, have a look through the FAQ, which contains additional information. We anticipate a heavy uptick in content as end of year approaches.

Beginner's Corner

More on MEMCOPY and Color LUT

In this month's column, we will continue the discussion with a shallow dive into the important BASIC816 MEMCOPY command and talk a little more about Color look up tables (LUT) and specifically, how to leverage Sprite / LUT data after being saved from a Sprite Editor. And 'yes', more balloon action.

The BASIC language, unlike 'C' or Assembly Language doesn't bother the programmer *much* with memory management. Yes, it's possible to exhaust memory with an extremely long BASIC program; yes, you do need to keep track of BASIC variables and pay some attention to data types; and yes, DIM'd arrays can be cumbersome, but garbage collection and strings pretty much take care of themselves and *floats* (not named as such) can serve as looping iterators without fuss.

The Commodore 64 and to some extent, the VIC 20 (both with BASIC 2.0), required extensive use of POKE and PEEK in order to take advantage of interesting features beyond just printing a character on the screen or reading a key or input from a keyboard or a peripheral.

By the time BASIC 7.0 was released (with the Commodore 128), just about everything you would ever want to do on the platform had an associated BASIC command, and knowledge of the memory maps and registers were no longer vital for survival. BASIC 7.0 had 168 commands in its arsenal. BASIC 10.0 for the fabled Commodore 65 (never released by Commodore) had 185 commands, supposedly.

On Foenix C256 platforms, we have BASIC816 and it features 90-something commands, operators, and function (versus Commodore BASIC 2.0 which had 71). Remove Commodore's legacy based I/O commands and you are really comparing about 90 to 60 which means that you've got far more utility here; and that is wonderful news !! Or is it?

As introduced in this column last month, we are able to do a fair amount with Sprites without even thinking about using POKE. And while we have not yet spoken about Tiles or Bitmap graphics in this column, you'll be happy to hear that BASIC816 support looks pretty good and some of what we've already learned is applicable.

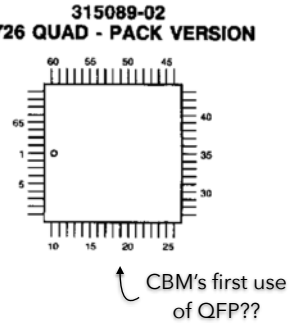
Music and sound? That's another topic and a long story, and I'm afraid that the answer is that the audio control capabilities of BASIC816 are about as good as SID control was within Commodore BASIC 2.0. POKE and PEEK will again be your best friends :)

Where does this leave us? In my opinion, it actually leaves us in **good** shape where 'good shape' equals learning quick access to commands for some of the heavy lifting required for an introduction to VICKY II graphic features, but ultimately, we will need to put in some work and learn how to read memory maps and spend time in the ML monitor. This column will help you get there. And these new skills will be transferable as we move to different languages ('C' or Assembly) on a given platform or a different Foenix computer (such as the C256 Jr., still under development).

DMA and why we need it (MEMCOPY to the rescue)

Graphics are about moving data (often, densely packed, large amounts of data) from point 'a' to point 'b', and on the Foenix, to do this efficiently, we ask the VICKY II DMA primitives for help. Of course, we are doing this *for* VICKY, meaning that we are placing graphic data into her view (in Video RAM aka "VRAM") so she can manage it, and ultimately, create the graphic signal output that a video monitor can understand in order to please humans with lovely and satisfying images in return.

DMA as a technology has been around for years and while the first 8-bit consumer systems did not have the luxury, 3rd party products and subsequent 16-bit systems leveraged these circuits heavily. Commodore's MOS outfit created an IC called the MOS 8726 between 1984 and 1985 and it was used within Commodore's 1750 and 1764 RAM expansion units and others.



Of course, by the time the Amiga was unveiled, terms like 'blitter' were more common and *Agnus* (one of Amiga's famous custom chips) had a big role in allowing the Amiga to raise and lower the desktop screen with ease and without interrupting the infamous bouncing ball demo; a feat that would otherwise fully consume the majority of machines resources, 68K CPU or not !!

Fast forward to today (we live in the future) and we've got DMA functionality built into the VICKY II. And the BASIC816 MEMCOPY command harnesses this power to setup and pass the acceleration through to your BASIC program.

✦ Distraction: What's the diff between a programmer and a developer? A programmer writes programs, a developer is used to chemically transform a film based latent image into a visible negative.

MEMCOPY in practice

On the last page of the Foenix Sprite Editor review (pgs. 9-11) above, you can see that we iteratively 'stamp' the image of the green arrow onto the bitmap screen using MEMCOPY which, using the RECT directive, copies a 2D (rectangle) region of data to VRAM. The loop that does this iterates 200 times (once per horizontal pixel).

The shape is small (16 x 16 pixels), but in aggregate, 200 times *anything* is a large enough task to tax an interpretive language like BASIC and really consume some wall-clock time.

Let's capture the elapsed time, move on to a 2nd example, and then examine a non-DMA (PEEK and POKE) approach.

Example 1: (this is the second example on the left side of pg. 11 above; here, we are just discussing the loop)

```
3505 FOR X%= 1 to 200
3610 IPOS%= &hB00000 + (SY%*15) *320 + X%
3620 MEMCOPY RECT &h100400, 16, 16, 32 TO RECT IPOS%, 16, 16, 320
3630 NEXT
```

Result: the green arrow moves 200 pixels from the left edge of the screen to position 200 of the 320 pixel screen width
Wall clock time: *3.88 seconds*

What if we expanded this to traverse the width of a higher resolution screen. To do so, we need to remove bit 10 (the '512') from the GRAPHICS command on line 100, change the loop length, and modify the stride from 320 to 640.

NOTE: we are running this loop to 624 because 624 + the 16 pixel width of the arrow will land it right at the edge of the screen, but still in view. (It actually appears a few pixels away because the right most edge of the arrow is transparent).

Example 2: (same as example 1, except with the modification noted; highlighted in blue)

```
3505 FOR X%= 1 to 624
3610 IPOS%= &hB00000 + (SY%*15) *640 + X%
3620 MEMCOPY RECT &h100400, 16, 16, 32 TO RECT IPOS%, 16, 16, 640
3630 NEXT
```

Result: the green arrow moves 624 pixels from the left edge of the screen to the right border of the 640 x 480 screen
Wall clock time: *10.76 seconds*

This next example is the piece de la resistance; that is, a fair representation demonstrating just how challenging work of this nature can be to an interpretive language like BASIC. Mind you, neither the original (example 1 above), nor this example, is optimized and to be honest, I've taken liberties with Ernesto's original code; it was never intended to be performant. His code found a clever use of Sprite editor data to create icons on a bitmap screen. Nevertheless, these examples do a good job at demonstrating the power of DMA and for us, how powerful MEMCOPY can be.

In **Example 3**: (using PEEK and POKE instead of MEMCOPY. Example 1 vs. example 3 diff highlighted in red)

```
3505 FOR X%= 1 to 200
3510 FOR YPIX% = 0 to 15
3515 FOR XPIX% = 0 to 15
3610 IPOS%= &hB00000 + (SY%*15) *320 + (YPIX%*320) + (XPIX%) + X%
3620 POKE IPOS%, PEEK(&H100400 + (YPIX%*32) + XPIX%)          ; MEMCOPY line was located here
3630 NEXT : NEXT : NEXT
```

Result: the green arrow moves 200 pixels from the left edge of the screen to position 200 of 320 using nested loops.
Wall clock time: ***222.51*** *seconds.* You are not seeing things, we pulled out MEMCOPY and replaced it with PEEK and POKE commands and a 2 dimension loop, and now it takes nearly 4 minutes instead of 3.8 seconds to accomplish the same task.

In essence, all this block of code does is read 256 bytes (16 pixel x 16 pixel), then writes 256 bytes, and does so 200 times. In aggregate, it reads 51.2K of data and writes 51.2K of data which is greater than the total amount of usable memory in a Commodore 64 (w/kernal and BASIC banked in). The combined nested loops impress nobody, but are necessary because we need a rectangle copy. Of course there is room for optimization in the calculation, but the 'as is' comparison is staggering.

Hopefully, this gives you a better appreciation of MEMCOPY's value and power. In theory, a VRAM to VRAM copy should be quicker than the SRAM (System RAM) to VRAM use; but the improvement is not noticeable since each iteration is waiting on the handoff between BASIC and the FPGA processor. Next month, we will consider writing this block move in pure 65816 assembly (with and without the VDMA assist; I predict interesting results). Let's close out this topic for now with a look at the 'manual' page.

MEMCOPY **syntax**

Borrowing a/the page from the BASIC816 manual, here is the syntax with a few callouts to hopefully make your life easier. It took me a little while to wrap my head around this topic.

MEMCOPY <source> TO <destination>

Request a copy of one block of memory to another block using the C256's DMA capabilities. In the C256, DMA transfers can work with two different memory topologies. A block of memory to be copied can be a linear (or "1D") block of contiguous bytes, or it can be a rectangular (or "2D") block of bytes, which can be a smaller region of a larger rectangular block. An example of this latter topology is when a program wants to copy a 32 pixel by 32 pixel block out of a 640 by 480 image.

In order to specify the block topology, **<source>** and **<destination>** must use one of two forms:

LINEAR <address>, <length> This structure specifies that the block is 1D or linear and starts at **<address>** and consists of **<length>** contiguous bytes. (See: Figure 1)

RECT <address>, <width>, <height>, <stride> This structure specified that the block is a 2D or rectangular block. The block starts at **<address>** and is **<width>** bytes wide and **<height>** bytes high. The block is part of a potentially larger rectangular "image" that is **<stride>** bytes wide. (See: Figure 2)

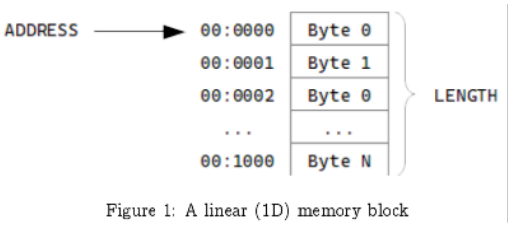


Figure 1: A linear (1D) memory block

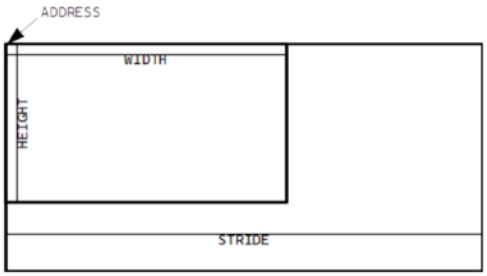


Figure 2: A rectangular (2D) memory block

Within certain limits, source and destination can use two different memory topologies, so the structures of source and destination do not need to match. An example where this might be useful is when a program loads a small icon file that is 32 by 32 pixels and then copies it to video RAM for display on a 640 by 480 screen:

```
MEMCOPY LINEAR &h010000, 1024 TO RECT &hB00000,32,32,640
```

```
3620 MEMCOPY RECT &h100400,16,16,32 TO RECT IPOS%,16,16,640
```

This statement will copy a 32 by 32 pixel image starting at 0x010000 to 0xB00000 as a 32 by 32 block in a larger 640 by 480 image.

There are two system limitations involving the addresses involved. The C256 has two different types of RAM: system RAM, and video RAM. System RAM (SRAM) is that memory from 0x000000 to 0x3FFFFF and can be used for all code and data. This memory is not directly accessible by the graphics subsystem, however. Video RAM (VRAM) is that memory from 0xB00000 to 0xEFFFFF and is exclusively for the graphics subsystem. All graphical data to be displayed must be placed in this block of memory.

Any memory copy involving SRAM involves stopping the CPU for the duration of the copy. This means any **MEMCOPY** transferring blocks between SRAM addresses, from SRAM to VRAM, or from VRAM to SRAM will halt the processor while the transfer is happening. While transfers will be quick, the C256 will not be able to service interrupts while this is happening. VRAM to VRAM transfers, do not stop the processor so interrupts can still be processed (although the BASIC program will still pause while the transfer is happening).

Another limitation is that while transfers between the two different memories can use different memory topologies as shown above, transfers within a single memory type have to use the same memory topology. For example, the following statement is legal:

```
MEMCOPY LINEAR &h010000, 1024 TO LINEAR &h020000, 1024
```

```
40 MEMCOPY LINEAR &h100001, 1024 TO LINEAR &hB00000, 1024
```

But this statement will result in an Illegal Argument error, since it is trying to use two different memory topologies within the same type of memory (SRAM in this case):

```
MEMCOPY LINEAR &h010000, 1024 TO RECT &h020000, 32, 32, 640
```

In our example above, the green arrow shape data embedded in the Sprite is 16 x 16 pixels and has a 'stride' of 32 bytes (eg, the offset to where the next row of useful data begins) and copies to a destination with a stride of 320 pixels (aka, the screen width at 47+512 resolution), In one of the examples above, we have a 640 byte stride since we run the draw across a wider screen (GRAPHICS 47 mode).

This is very similar to what we do on line 40 of a few of our examples from last issue. In our case, we are copying from &h100001 to 'skip' the first byte of the file (metadata from the Foenix Sprite Editor). We are also moving from the middle of memory (1 MB mark) to VRAM (&hB00000) aka (11 MB)

Palette Manipulation

Now let's switch gears a bit. If you read *Beginner's Corner* in issue #1, you'll recall that we had our Balloon Sprite defined and in memory, we loaded a Color LUT from BASIC language DATA statements, and we wrote a few simple loops to move a single balloon around the screen.

Creating Sprites is fun and working with graphics in general, is enjoyable. But it can also be frustrating because there is a fair amount going on *behind the scenes*. Note the key phrase, "behind the scenes". Since data is encoded in bits and bytes and since it in no way resembles the end-product, we are going to need a good understanding of how to interpret color data so we can use it to our benefit. On the next few pages, we will share one approach.

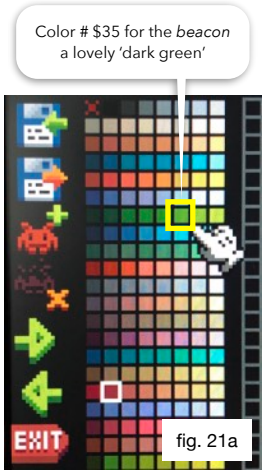


fig. 21a

(an alternate approach is to jump into a time machine and teleport to the year 2030 where we will have a Foenix with 4 screens and every tool imaginable; unfortunately, my time machine is in the shop, so instead we will create a few BASIC programs to assist with data gathering).

Choosing colors from the aforementioned Sprite Editor color palette is quick and easy but knowing which of the colors you chose (many are similar looking, especially in high resolution) after the fact is near impossible unless you record the coordinates of colors on the 32 x 8 grid (see figure 21a), and then find a way to decode the BASIC data statements, counting to the precise position ("seven hundred and twelve, seven hundred and thirteen"…); get the picture?

In last month's column, example "`I01P11A.BAS`" loaded LUT 0 with a full set of 255 colors which were identical to the default palette of the Foenix Sprite Editor. If you looked at the code, you saw that a FOR loop took a pile of values (768 of them) ranging from 0 to 255, and used SETCOLOR to build LUT 0.

This month, we will dynamically alter (**overwrite**) a few of the colors in the LUT by redefining them on the fly with new values in order to affect subtle animation of our Foenix Balloon; specifically, the *dark red beacon* at the top (which we will flash *dark green*), and all of the colors of the *flame* at the bottom (6 of them). To try this at home, you will need* "`FBALLOO2.SPR`" (the footnote below will lead you to the "why necessary").

Here's a question: considering that we are about to overwrite values in the original LUT, how might we undo an update later? The bad news is there is no mechanism for this, they are gone forever. (cue sad trombone). Therefore, in addition to selecting new values, we are going to need to record the original RGB values. "LUT LOAD" will help.

Tool #1: "`LUT LOAD.BAS`"

This tool loads the palette <u>while</u> presenting the RGB color values on the screen so you can record them (aka, write them down). This utility, along with the "DUMP SPR" utility on the next page completes the picture.

Figure 21b is a bit of an eye chart, but here is what is going on. "LUT LOAD" displays pages (8 of them), each with 32 colors per page, showing the hex and decimal color numbers and their corresponding RGB values.

The BASIC program that does this is short (11 lines long not counting the DATA statements) and we've seen line 110 and 130 prior. Here it's wrapped in a FOR / NEXT loop. We are using variable `c%` and incrementing it on line 130.

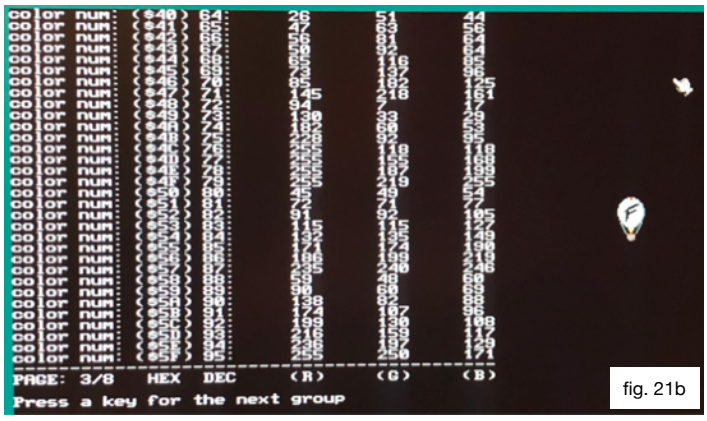


fig. 21b

```
100 CLS : c%=0                                                  ; c% or color # is the iterator
110 READ r%, g%, b%                                             ; same command as used last issue examples
120 PRINT "color num: ($"; HEX$(c%); ")"; c%; ":", r%, g%, b%   ; nicely formatted values, courtesy of BASIC
130 SETCOLOR 0, c%, r%, g%, b% : c%= c% +1                      ; same command as used last issue examples
140 IF c% MOD 32 <> 0 THEN 110                                  ; we use this for pagination (32 per screen)
150 PRINT "-----------------------------------------"
160 PRINT "PAGE:"; INT(c%/32); "/8   HEX  DEC"," (R)"," (G)"," (B)"  ; this (and the line above) prints the footer
170 IF c% = 256 THEN 200                                        ; detect 'end' and if so, jump to line 200
180 PRINT : PRINT "Press a key for the next group"              ; else wait print the 'wait' message
190 GET a$ : CLS : GOTO 110                                     ; get a key, clear screen, and loop back
200 END
```

*Treasure hunt… page 25 speaks to the need for "`FBALLOO2.SPR`"

Now that we have our alternate colors and the equally important original colors, we will construct a table for documentation purposes (see pg. 23). Just like memory, data written on paper without context does not mean much. They are just a bunch of values. It's important to be organized.

It's also worth mentioning that our alternate colors do not need to be defined elsewhere in the LUT, but there's nothing wrong if they are. This is a bit confusing, but not if we think of a LUT as a massive box of crayons or colored pencils and accept the reality that many of them are rarely, if ever used (such as APRICOT or SEA FOAM) and yet, we may have doubles of some. What we are doing is replacing the color in slot 73 or in the case of water colors, taking liberties.

On pg. 21 we identified a nice green color from the Sprite Editor palette but we didn't change our Sprite because the starting value of the beacon, as defined, is dark red ($49; see yellow box at the top of figure 22a below);

The important part is capturing the values using "LUT LOAD" above. We could have just as easily chose a Pantone color found on one of the many internet color guides in the same way that I chose *Donkey Kong blue* for the Foenix Marketplace color. The important part is that we know how to 'mix' our chosen colors. Now we just have to confirm that we are leveraging them as we thought when we originally defined our Sprite.

Tool #2: "DUMP SPR.BAS"

This program was printed on page 12 of Issue #1 though we did not discuss it. "DUMP SPR" interprets Sprite data from SRAM and dumps it to the screen, presenting each *set* pixel with its color number in hexadecimal format.

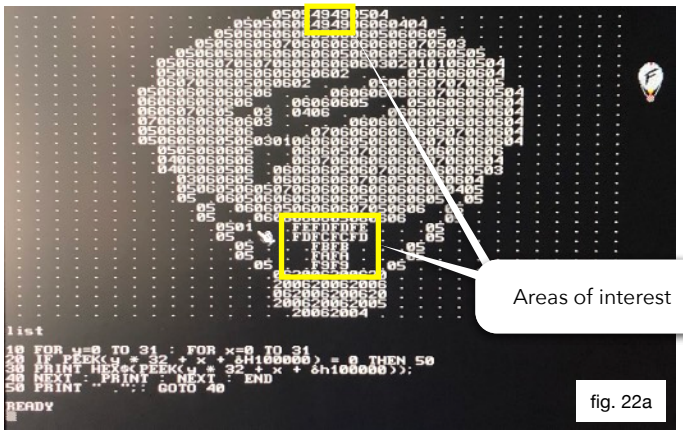


fig. 22a

The following 5-line program does the work, peeling back the lovely colorful exterior to instead show the skeletal structure of hexadecimal values representing not colors, but values that index into the color LUT. We are utilizing hexadecimal because an 8 bit value represented in hex occupies 2 characters consistently and makes better use of screen real-estate; and thankfully, BASIC816 affords us a handy BASIC command for this purpose; HEX$(). If we are going to graduate to Assembly language, begin to use the MONITOR, and learn how to leverage the dozens (if not hundreds) of registers that your Foenix computer has to offer, we should start getting familiar with this notation. When I was in grade school in New York in the late 70's, they taught us hexadecimal notation in math class. (also the metric system).

Here is the program listing, complete with copious comments. It's a bit dense here in the text; maybe copy and paste it to an editor and then modify it by inserting spaces to make it more readable:

```
10 FOR y% =0 to 31 : FOR x %=0 to 31      ; nested loop runs for each row (y%) and each column (x%) 32 x 32
20 IF PEEK((y% * 32 + x + &H100000) = 0 THEN 50   ; for each iteration, see if the value of the byte is 0. If it is, goto line 50
30 PRINT HEX$(PEEK(y% * 32 + x% + &H100000));     ; otherwise, print hex value peeked from (offset + base). Note the ';'
40 NEXT : PRINT : NEXT : END              ; close the inner (x%) loop, print a blank line per row, close outer loop, end
50 PRINT " .";: GOTO 40                   ; print a " ." when a non-colored pixel is encountered. The semicolon
                                            prevents BASIC from printing a 'return', which is its default for each line
                                            printed. If we didn't do this, every printed value (1024 of them) would
                                            result in a column of hex digits that would scroll continuously !!
```

This program would be two lines shorter but we are opting to do something special when the transparent 'color' (value 0) is present. This is useful because all unused pixels and all intentionally transparent pixels contain 0's. Remember, this is **computer memory** we are talking about, so every location *has* to contain *some* value.

Memory can either contain uninitialized 'garbage' or whatever random data may have been present from a past use, or preferably, it contains *expected* data such as values we placed there or values that we will interpret for goodness.

These two programs do not use POKE, only PEEK, and DATA and READ statement, so you can consider them read-only and harmless. The only harm is keying in the incorrect address or botching up the formula. Such a mistake will only send us on a wild goose-chase of madcap colors. If during experimentation, you are struck with unexpected results, don't give up. We all learn by making mistakes and recovering from those mistakes. Do keep a 'prime' copy of your LUT. It's important that what you see in the Sprite Editor (or your tool of choice) has a data set that you can use reliably.

In our next issue, we will discuss a few enhancements for the Foenix Sprite Editor and also how to leverage saved palettes. There is no Deluxe Paint for Foenix yet, but it's not crazy to expect to see King Tutankhamun at some point.

Colors organized, and a simple program to put them to use

	hex - - LUT - -	dec	RGB color value	
dark red	$49	73	130, 33, 29	2 colors for flashing the beacon
dark green	(also $49; shared* w/above)		125, 164, 45	
red	$F9	249	156, 51, 39	6 colors for cycling the flame on ascent
dark orange	$FA	250	191, 90, 62	
orange	$FB	251	233, 134, 39	
yellow	$FC	252	255, 177, 8	
light yellow	$FD	253	255, 207, 5	
lighter yellow	$FE	254	255, 240, 43	

*See lines 1030 and 1040 below

What does this program do / what doesn't it do?

This program moves the Foenix Balloon diagonally within the bounds of a 640 x 480 screen. This means that when the balloon heads in a given direction, it needs to a) sense the practical edge of the screen and in that eventuality, b) change direction on either (or rarely, both *x* and *y*) axis. We could have run the *x* location down to 0 which is actually off screen (under the border, depending on the graphics mode, if we wanted to), but we are opting to define a top (`tip%`), bottom (`bot%`), and then characterize the right edge as (`right%`) and reeling it back 'in' to the (`left%`) edge of the screen.

Why name a variable "`tip%`"? Fun fact: an earlier version of this program used "`top%`", which was intuitive, but the BASIC816 tokenize routine assumed I wanted "`TOp%`" as in:

> "FOR x% = 1 TO p%" - this is NOT what I wanted because we are not using a FOR loop; but BASIC just blindly saw the characters 't' and 'o' and assumed we were. The lesson learned is: don't use "`to`" or any reserved words in your variable names. The parsers we are accustomed to in 2022 don't translate to primitive languages. They never had the time or memory to deal with context.

Example "`I02P23A.BAS`" - There is a more elegant way to move a Sprite diagonally, but this works!

```
100 x%=26 : y%=32 : as%=0 : in%=0 : be%=0      ; Init starting position, as%(cent), in%(ward), be%(acon) flags
105 tip%=64 : left%=58 : bot%=448 : right%=614 ; identify top, left, bottom, and right bounds

110 IF x% MOD 40 = 0 THEN 1020                 ; alternate beacon every 40 horizontal pixels
120 SPRITEAT 0, x%, y%                         ; position Sprite 0 with the updated coordinates
125 FOR n = 1 TO 50 : NEXT                     ; time delay (will need to shorten this as complexity increases)

130 IF as%=1 THEN 910                          ; if ascending branch to 910
140 y%= y% +1 : IF y%= bot% THEN 160           ; otherwise inc y% and check if at bottom, if so, then 160
150 GOTO 700                                   ; else goto 700 which is the x axis inc/dec block
160 as%= 1                                     ; if at the bottom, reverse direction and fall to the x axis code

700 IF in%= 1 THEN 730                         ; if x direction is 'inward' aka, decreasing then 730
710 x%= x% +1 : IF x%= right% THEN 770         ; otherwise inc x% and check if at right, if so then 770
720 GOTO 110                                   ; else go back to top, we're done
730 x%= x% -1 : IF x%= left% THEN 760          ; since x direction is 'inward', dec x%, if at left border then 760
740 GOTO 110                                   ; else go back to top, we're done
760 in%= 0 : GOTO 110                          ; since at left, reverse dir then to top, we're done
770 in%= 1 : GOTO 110                          ; since at right, reverse dir then to top, we're done

910 y%= y% -1 : IF y%= tip% THEN 950           ; since ascent, dec y% and if at top then 950
920 GOSUB 2000 : GOTO 700                      ; else gosub 2000 (flame cycle subroutine) then onto x axis code
950 as%= 0 : GOTO 700                          ; since at top, reverse direction then onto x axis code

1020 IF be%= 1 THEN 1040                       ; if beacon flag (be%) is on, then 1040
1030 SETCOLOR 0, 73, 125, 164, 45 : be%= 1     ; set color $49 (4 pixel block @ top of balloon) to green, set beacon flag
1035 GOTO 120                                  ; to top
1040 SETCOLOR 0, 73, 130, 33, 29 : be%= 0      ; set color $49 to dark red, clear beacon flag
1050 GOTO 120                                  ; to top
2000 RETURN                                    ; placeholder for flame cycle subroutine
```

More about this BASIC program

There are numerous approaches for *iterating* and testing, even in a language as simple as BASIC. Adding decision statements with embedded calculations to the mix introduces time variability that can add up in unexpected ways.

Aside from having to detect the screen edges, this program is made complex because we are animating the balloon's flame only when ascending. This works in our favor since in the natural world, gravitational forces work against thrust.

Unfortunately, the way we've coded the other element of animation is flawed; the flashing beacon is nice and all, but as written, it will flash more slowly on ascent because the amount of time required to cycle the flame colors means less *x* movement per second. We cycle the beacon each 40 pixels of *x* movement, so less overall motion across a given time span means more time between each flash when the balloon is going up.

We can compensate for this by adding a line or two of code that uses a different MOD value on ascent, but there is no time for science here (calculating jiffy clock and the 'cost' of work), so one solution would be to approximate a fixed value that is calculated only after the rest of the program has been written. Another approach is to bind the beacon flashing to an interrupt which is jiffy clock based. We may do this another time.

In the early days of game development, programmers literally calculated the cost of routines by tallying cycles per instruction and then added balance with compensating delay loops. Games that execute time critical code in order to maintain smooth player / adversary motion need to deal with this at grand scale; think about the complexity of a game like the arcade classic, *Galaga,* which can have a highly variable number of enemies (each with different 'flight path' and rate of movement) and a variable number of missiles on the screen at any one time.

Leveraging hardware features (such as interrupts) or algorithms that orchestrate timing in concert with a free running clock is effective, but BASIC is not exactly well suited for this type of programming. For this we will need some assembly language; if interested, Pater Weingartner addresses this topic in one of his 65816 tutorial series videos, here.

BASIC816, "We've got a problem"

I've always been keen on off-the-beaten-path algorithms, and the solution to the *self-inflicted challenge* (animate a Sprite without the use of several, or a dozen different Sprite definitions) was something that I thought I had a good solution to but unfortunately, it exposed a flaw in BASIC816 that I did not anticipate.

I planned on leveraging a single dimension array of integers of RGB values and a simple formula wrapped in a loop that applied newly selected values to each of the 6 palette colors used in the Foenix Balloon flame. But in testing, I found a corner case (a bug in BASIC816) that has not been resolved (I confirmed this with the developer) and with no time left, I had to cut the cord. What I will do is take a moment and let you know where I was going with it.

The approach:

- Add initialization of 0 for variable `fla%` (flame index) to line 100
- Add a DIM for color array `col%(18)` to line 100
- Add a GOSUB 3000 (to initialize the array) to line 100
- Add the following to the program

```
2000 n%=0
2010 SETCOLOR 0,249+n%,col%(fla%*3), col%(fla%*3+1), col%(fla%*3+2)
2015 n%=n%+1: if n%=6 goto 2050
2020 fla%=fla%+1: if fla%=6 then 2040
2030 goto 2010
2040 fla%=0: goto 2010
2050 return
```

Each GOSUB call of this routine iterates `n%` 6 times (0..5), redefining colors $F9, $FA, $FB, $FC, $FD, and $FE with different sets of colors based on the index of variable `fla%`. At least in theory, this is how it would work. Since I was unable to run the code, I could not confirm whether the algorithm was convincing. We may need to rework the order of the colors or alter the flame definition such that the color spread is more random, versus just being graduated vertically.

```
3000 col%(0)= 156 : col%(1)= 51 : col%(2)= 39          ; initialize red color
3010 col%(3)= 191 : col%(4)= 90 : col%(5)= 62          ; initialize dark orange color
3020 col%(6)= 233 : col%(7)= 134 : col%(8)= 39         ; initialize orange
3030 col%(9)= 255 : col%(10)= 177 : col%(11)= 8        ; initialize yellow
3040 col%(12)= 255 : col%(13)= 207 : col%(14)= 5       ; initialize light yellow
3050 col%(15)= 255 : col%(16)= 240 : col%(17)= 43      ; initialize lighter yellow
```

Why "FBALLOO2.SPR"?

When defining the original Foenix Balloon (months ago), I blindly chose colors that I thought looked good together without thinking forward that I would be programmatically manipulating data in any particular way. As a result, I chose the same color for part of the flame as for the beacon at the top of the balloon.

Fast forward to this issue and I suddenly have the need to dynamically redefine the color palette in order to provide the appearance of the beacon flashing. This would have peculiar results anyplace else that I used the same color 'dark red', should it start alternating with 'dark green'.

To compensate for this, I redefined this portion of the Sprite and saved it with an 8th character of "2". In doing so, I also took the opportunity to patch up the way that the flame area was organized. I also stumbled upon something unexpected (that I did *not* correct).

I inadvertently set a few pixels using color $01 which pertain to black in our palette. Against a black background, it is indistinguishable from color $00 (aka transparent) and I didn't realize it while editing; but it is obvious, thanks to our "DUMP SPR.BAS" example (figure 25a).

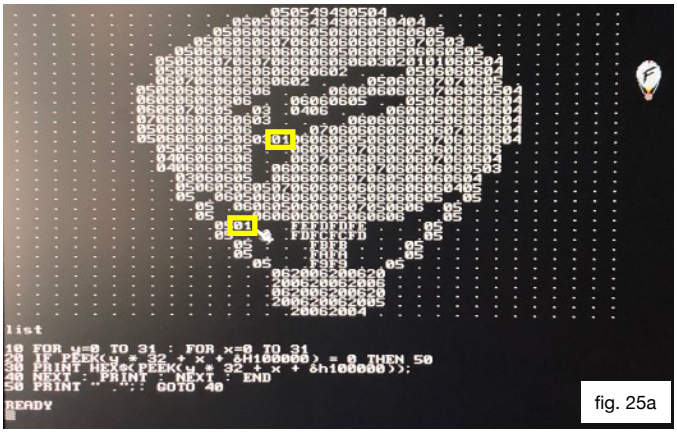


fig. 25a

For the work we are doing here, it won't matter, but should we at some point navigate our balloon across a light blue background or in front of some clouds, it might appear odd.

What I should have done was consistently set the 'F' area as actual black $01 and confirmed that everywhere else (between the balloon and the ropes and in unused areas of the Sprite) was transparent $00.

The other thing I should have done, was to slightly redefine the color palette so that the cycle series $F9 through $FE was pushed one position such that the color sequence instead run from hex $FA to $FF. Having values pressed against byte boundaries is sometime handy. Thinking a few steps ahead (now in hind site!), if I move to assembly language, I can increment the value and then check for status 'carry' in a single branch instruction versus comparing against a particular value $FF as being the end of the range, before resetting. In the end, it won't matter, but these are the things that you should consider when working with binary data or any data.

A quick note on loading binary data into memory

Because we've yet to touch on it directly, let's talk about loading graphics data into memory from BASIC. Simply, the BLOAD command performs a binary load of a file to a given address. The following text is from the BASIC816 .pdf:

> BLOAD <filename> [, <destination>]
>
> Loads a binary file into memory starting at the specified <destination> address in memory. If there is no provided <destination>, the file format loaded must have its own destination address embedded within it. Currently, only the PGX format is supported, but others may be supported in future.

As used within our programs, it is common to see BLOAD "ICONS.SPR", &h100000-1 and since we are loading binary data (not an application), the destination address is necessary. The hexadecimal memory address (note the '&h') of 0x100000 is followed by a '-1' simply to compensate for the fact that .SPR files have a leading byte. You don't always want an offset, but in this case, we need one because the leading byte of the Sprite file is extraneous.

Final thoughts for now

Between now and next issue, I'll be releasing (to the Foenix Marketplace) a set of 12 Sprites that we will use in the next issue. With it, we will be able to take care of the heavy lifting and resolve the timing variability and the fact that DIM doesn't work as expected.

Microcomputers, since they have limited resources, have always had to strike a balance between memory usage, code efficiency, and speed. Since we are having this conversation in 2022 and not 1982, we can take comfort in the knowledge that we have many times more memory (and CPU power) than the systems that many of us grew up using.

So moving along to the 'ultimate' solution; we will define two sets of 6 Sprites; one that is leveraged when the balloon is ascending, and one set for use when descending. This approach will cost about 11K of additional Sprite data but sets us up nicely to leverage an extremely simple piece of code versus the madness I dragged you through on the prior pages.

Jack Tramiel would have really liked the C256 Jr. concept. It leverages time tested tech that Commodore developed across its most profitable years and adds openness, allowing the buyer decide how to finish the final 10% of their machine. This is where "for the masses" meets the 2020's.

Go crazy with a high-end, LED hologram projection mini-ITX case, squeeze it into an old (or new-vintage) keyboard case, or buy a $29 Pico-style ITX power supply from Amazon and screw a few standoffs to a piece of plywood. (that's what I'm going to do*)

"TINY VICKY" will feel familiar to anybody with Foenix experience; and for those that have used the Commodore VIC-II chip, VICKY will feel like a well appointed sports car.

Such a photogenic little one

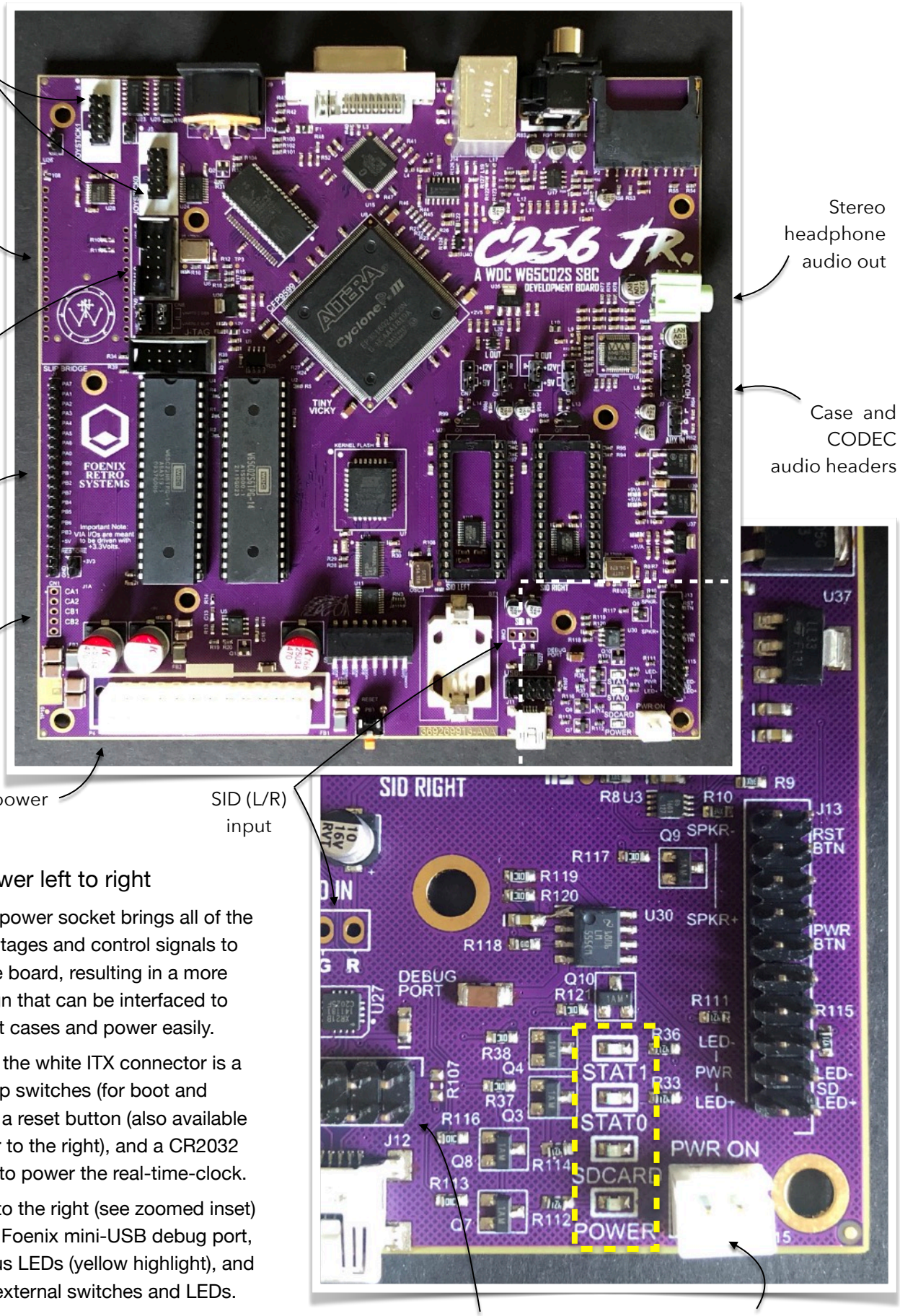
The C256 Jr. wows its adoring fans with each snap of the photo shutter. Each corner of the board is a conversation piece.

Since there is an IEC port and a 20-pin CBM keyboard header onboard, lots of hardware and software interfacing options are possible. As a C256U+ owner, It's tempting to say "less is more"; but with this proof-of-concept/dev board, *more is more!* It checks boxes that I have not heard in conversation since the early (pre FMX) days of Foenix, and does so at an especially attractive price.

The following pages detail a few of the C256 Jr's assets in photo form. Stay tuned for the next two issues where we will be discussing kernel development and a preview of the final prod specifications which are coming together now.

✦ ... what I'll really going to do is dust off my bench shear, my No. 16 punch, and a 48" box/pan brake, then ignore my family until my youngest son says "hey Dad, 1979 called, they want their middle school metal shop skills back"

Power, control, and header overview

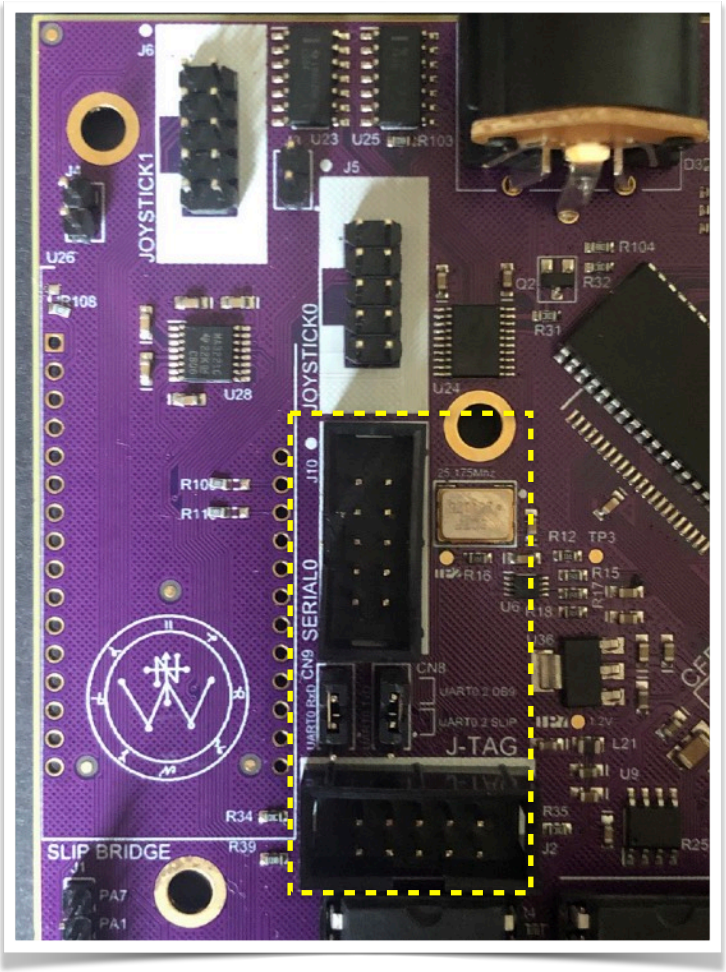


Lower left to right

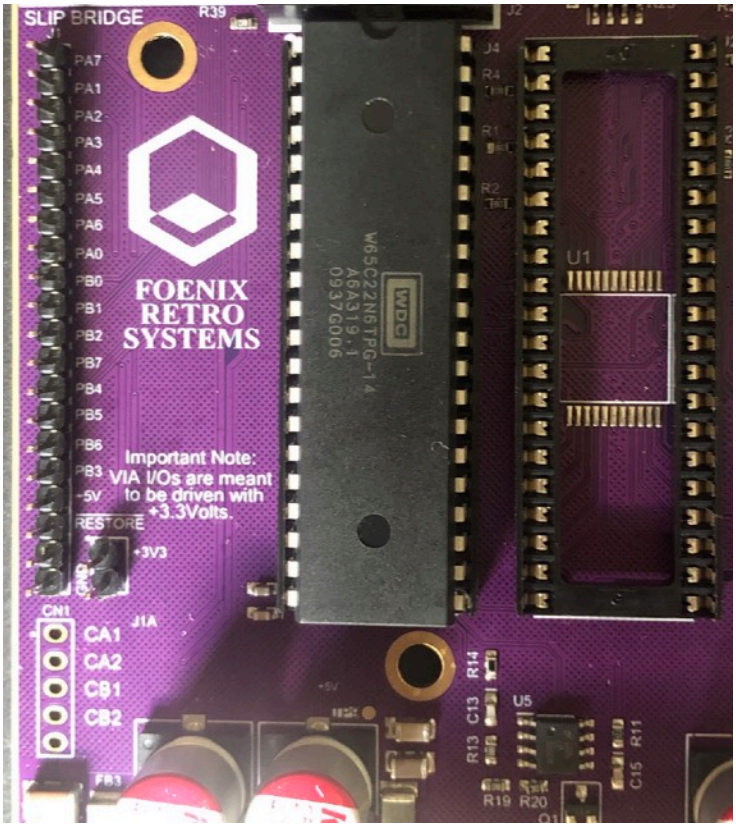
The 24-pin ITX power socket brings all of the necessary voltages and control signals to and from the board, resulting in a more simple design that can be interfaced to aftermarket cases and power easily.

To the right of the white ITX connector is a bank of dip switches (for boot and configuration), a reset button (also available on the header to the right), and a CR2032 battery holder to power the real-time-clock.

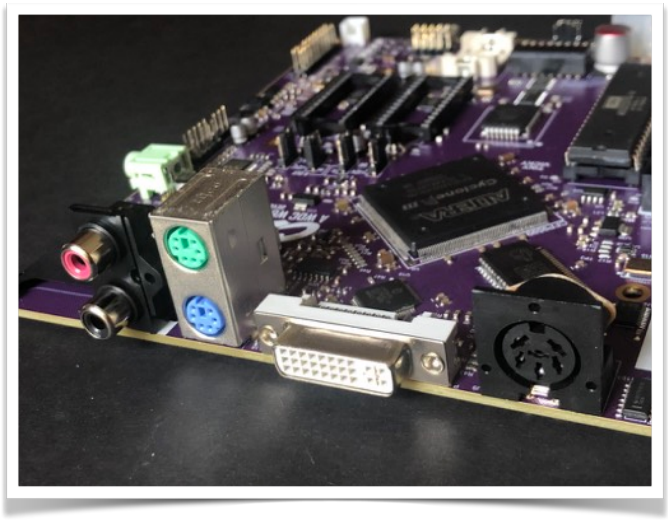
Moving further to the right (see zoomed inset) is the standard Foenix mini-USB debug port, a series of status LEDs (yellow highlight), and a header for external switches and LEDs.



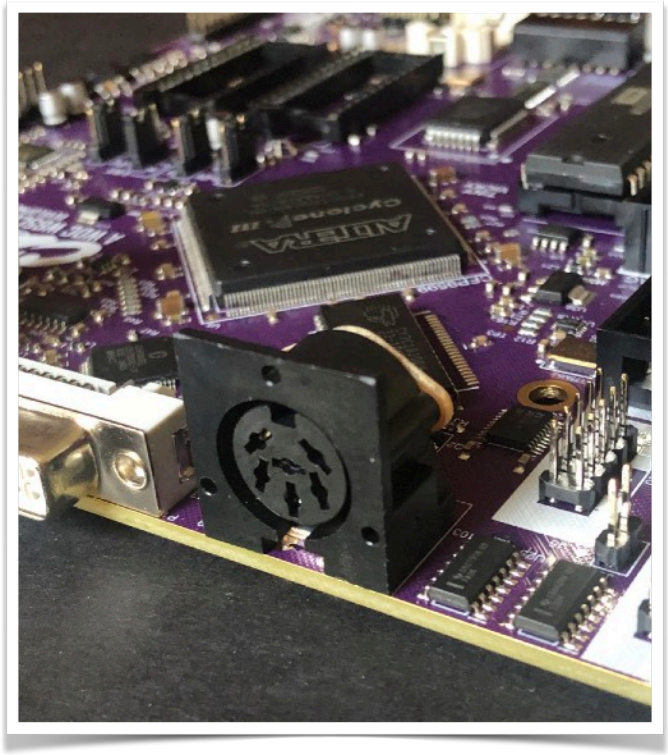
JTAG, jumpers, and headers for serial vs. “SLIP bridge”



65C02 pulled to reveal this dev board’s QFP footprint



SD card (left, partially obscured), stereo audio line out, PS/2 mouse and keyboard; DVI-I video, and 6-pin DIN IEC (Commodore serial peripheral port)



IEC evokes W-H-Y? So many people ask “why bother” with this slow and antiquated interface?

If you have to ask, you don’t understand. Many in this community possess a collection of aged IEC peripherals and some are *building* their own (new).

Well before SCSI, before SATA, before IDE, there were ‘*proprietary* standards’ and Commodore just happened to have the coolest (borrowed from IEEE-488). Overkill, ‘yup’. Dog slow, ‘check’. But by far, the coolest. How else could my High School have shared an 8250 dual disk drive and a daisy wheel printer between 6 PET Computers “back in ’82”?

Back Page - Foenix Issue #2 takes a look at COMPUTE Magazine issue #2

Compute began as a Newsletter, exclusively for the Commodore PET

In this first of two parts, we'll take a look at the origins of what would become a flagship computer magazine and touch on a few products, distractions, and technologies that fostered the growth of an industry

I was at the right place at the right time 'recently' when a Minnesota based retro enthusiast offered up "old issues" of his vintage computer magazine collection for free. He wasn't keen on cataloging them ahead of shipping, or entertaining questions about individual issues, but the first to take them all and pay shipping, got them. He was clear, however, that he was keeping issue #1 of each publication. I may have paid $30 for shipping of a 12" x 12" box, the details are foggy. This *recent* event was 15 years ago.

25 years prior to that, I was fortunate to have access to some of these publications when they were new, and like many across those years, the usefulness of having a monthly journal deliver a continuous supply of articles chock full of technical detail, tutorials, and type-in programs, could not be matched. Of course, there were ads; tons and tons of ads which today, provide fodder for articles in their own right (such as last month's *Back Page* on "Skyles 1541 Flash!"), or amazement ("I can't believe such a product existed, wow!"), and also amusement ("what?! People paid that much for …"). It was a different time and place then exists today, or has existed since the birth of the modern Internet.

There was a time when two streams of content coexisted peacefully; a healthy print publishing industry, and an emerging pre-commerce (largely, University and Government funded) Internet, which offered managed Gopher, Archie, [and most of all] Usenet News services to those of us with access to Unix or VMS based school or work computers.

SGML (structured generalized markup language) hadn't yet led to a definition of HTML, and NeXT Computer would not have been invented for another 6-8 years and as such, Sir Berners-Lee was nowhere near prototyping his first HTTP Web Server.

You see, in the mid-to-late 80's and early 90s, Internet content was actually 'free' and printed material was worth the cost of subscription. BBS'es weren't bad either, and other options were emerging as well (such as Compuserve, Fidonet, Qlink, and moderated forums such as "The Well" aka *The Whole Earth 'Lectronic Link*).

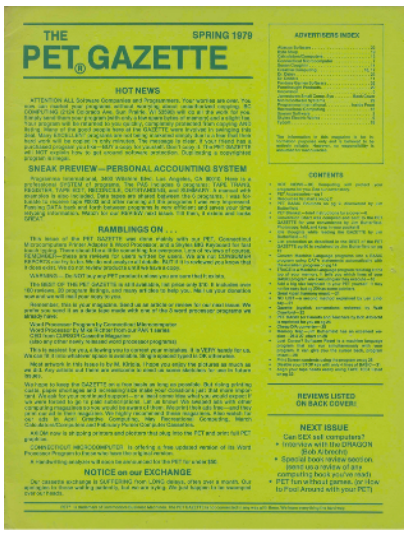
It is ironic that just about all of the vintage print based material is catalogued and available for free today, meanwhile, you cannot even click on a recently published news article (or YouTube vid) of any kind without hitting a paywall or being force-fed "news" about how a single mother of 3 discovered an indispensable teeth-whitening trick or "32 child celebs from the 90s - you won't believe what they look like today !!". Ok, enough of that… I do apologize.

The point of this article is a recollection of better times, and Compute Magazine joined TPUG, Transactor and others to report and disseminate news and innovation from a burgeoning industry, the first of its kind.

The beginning - not the beginning

In doing research for this article, I came to learn that even issue #1 of Compute, was not issue #1. Compute was the 2nd iteration of a publication that began through the efforts of one Len Lindsay, who in April of 1978, decided to put his College Newsletter Editing skills to work; to aggregate Commodore PET resources and information into a folded 5.5" x 8.5" monthly titled the "PET GAZETTE".

Len's first four issues were indeed published monthly, but success turned to growth which grew to become a burden, and it forced a seasonal issue frequency and ultimately, pushed Len to turn the publication over to "Small System Services", who had reached out to him with interest right about the time that he was considering calling it quits. Len went on to contribute to Compute as an author and an editor; You can read the 'early' days of Len's story here (from Compute Issue #1, pg. 65).


THE PET GAZETTE

SPRING 1979

HOT NEWS

SNEAK PREVIEW—PERSONAL ACCOUNTING SYSTEM

RAMBLINGS ON . . .

ADVERTISERS INDEX

CONTENTS

REVIEWS LISTED ON BACK COVER!

NEXT ISSUE

NOTICE on our EXCHANGE

Len Lindsay's publication just prior to transition to 'COMPUTE.'

A very quick look at COMPUTE Issue #1

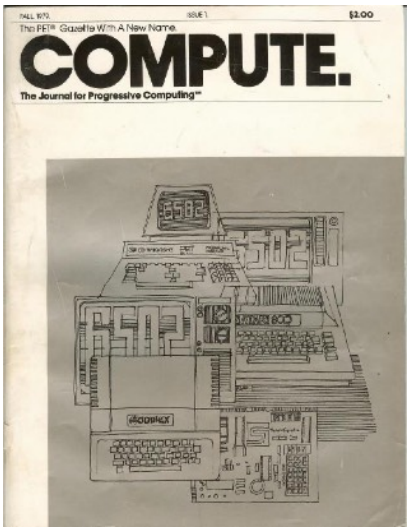
The PET GAZETTE closed with a "Super-issue", so it was only fitting that Compute begin with a seasonal (rather than monthly) "Fall of 1979" blockbuster of its own.

Issue #1 of Compute was composed of 108 pages and contained 46 articles, 10 of which were product reviews plus a group of 4 passages representing Part 1 of Len's Word Processor run-down; There was also a hodgepodge of topics including some educationally focused, general 6502 assembly language, a few articles on the BASIC language, one on printing 'pin-feed' labels, and an intro to a computer science level look at sorting algorithms, to name several.

Issue #1 of Compute broke from the Gazette's Commodore PET *only* format, but PET related topics still dominated the issue and it attracted a growing list of contributors such as Jim Butterfield, and numerous others.

Two of the more interesting articles to me (because of what came prior and what was to occur in the next 10 years) were Len Lindsay's "The Evolution of a Magazine" retrospective, and the very much forward-looking "Publisher's Remarks" editorial, which included Editor Robert Lock's welcome notice, and framed his hopes for the new publication.

The two-part "Sorting Sorts" is probably my single favorite article because it was rare in the early days of this publication to see advanced software development or data structure topics interspersed with consumer and hobbyist content. In these early years, enthusiasts with SBCs were incorporating circuits that were on-par with industry quality integration (probably because many worked in electrical engineering in the first place), but software coming from 8-bit systems was just getting started.



COMPUTE. Issue #1 - note the reference to PET Gazette

COMPUTE Issue #2 - size, shape, and distractions

The remainder of this article discusses Issue #2 of "COMPUTE. - The Journal for Progressive Computing™" as published by Small System Services in Greensboro North Carolina, USA. We will continue on this topic in Part 2 of Foenix Rising #3.

Issue #2 was published as the "January/February 1980" issue; at least, it was an issue said to represent the first two calendar months of what was to be a monumental decade for the magazine and the industry.

Robert Lock, editor, quoted the time to publish an issue as a 4-5 month endeavor so it's clear to see that with the exception of some last minute product release bulletins, the publishing workflow of the day didn't square well with the turnaround time offered by printers of this small but growing quantity.

The newsstand cost of Compute was $2.00 and Issue #2 was 116 pages long including a 16 page buyers guide coined the "After Christmas Buyers Guide" (not to be confused with Issue #1's "Christmas Buyers Guide" which was nearly identical :)

COMPUTE. Issue #2

Loosely, the issue had a preamble containing a table of contents, editor notes, and miscellany, with the remainder of the magazine split into two parts, bisected by the aforementioned buyers guide. Said guide looked as if it was printed inexpensive copy paper, the type you used in your home printer when you're out of white paper (it was light blue).

The buyers guide was more of a disorganized selection of ads than anything intentionally structured, but looked at individually, there were some gems within the guide and throughout the magazine; one example was the Eastern House Software's "PET Breadboard Kit".

For $39.95 USD (postage included), the buyer gets a 'kit' composed of a breadboard and mounting plate, a 36" ribbon cable with 24 pin DIP plug, a connector for the PET's IEEE / User port, and and a clip lead, for power. "*Now you can build those circuits you see in magazines easily and quickly!*", the advert urged.

Later in the same issue, Eastern House Software advertised some *actual* software: the MAE (PET Macro Assembler, for $169.95); this was the predecessor to the two-pass Commodore branded "Commodore 64 Macro Assembler", one that I relied heavily upon a few years later. It's fun to connect the dots backwards to see the origins of where some of these companies originated.

One of the first pieces of reader feedback was from a guy named Dennis C. Hayes (yes, *that* D. C. Hayes). He was writing in response to an issue #1 review of one of his products. Really, he was writing to plug his other products but also to complain (mildly) about the fact that his $395 Apple Micromodem™ product and accessories were reviewed "without being noted as trademarked products" (tsk tsk), but also that the magazine did not refer to his outfit by its proper registered name, "D. C. Hayes and Associates, Inc.".

A vintage detour - the rise and fall of D.C. Hayes and their Smartmodem^{tm}

fig. 30a

Of course, from proprietary beginnings (Apple specific ROM encoded signaling of the Micromodem), Hayes and his associates would go on and revolutionize the modem industry with the release of the Smartmodem^{tm} (1981) and its "ATtention" command set method of controlling it, which obviated the need for complex and proprietary interfacing in favor of simple text commands that were interpreted by the device over RS-232 serial lines.

The Smartmodem was expensive, but the "AT" protocol made the Hayes usable to Commodore, Atari, Apple, dumb terminals or Unix hosts without fuss. And this feature-set was ripped off by every manufacturer including Commodore with their 1200 baud modem.

This facet of the peripheral industry grew massively right through the PC clone revolution, though Hayes was wiped off the map after a few poorly timed bets on ISDN and a fierce battle with US Robotics, not to mention a growing field of also-ran, low-cost manufacturers including one with an ominous name: "Prometheus". Hayes never saw them coming; had they been on the lookout for the "Titan who was chained and tortured by Zeus for stealing fire from heaven and giving it to humankind", we wouldn't be having this conversation. :) Again, I apologize.

Hayes also botched licensing opportunities and mismanaged the handling of patents and lawsuits, which makes Dennis Hayes letter to Compute even more ironic.

I still have my *Smartmodem 1200* and it still works to this day. It proudly powered my Commodore 64 based BBS for a few years when I was in Berkeley; I still have not located the sister product, the *Hayes Stack Chronograph* (see figure 30a). It could be that Hayes did not sell many; the Chronograph was an expensive solution looking for a problem.

As Foenix owners are likely aware, all Foenix computers ship with an RTC (real time clock) circuit; feed your Foenix a CR2032 battery and you've essentially reproduced what Hayes once sold for $249 USD (about $800 in 2022 dollars), but directly memory mapped as binary coded decimal.

The Hayes Chronograph on the other hand, would communicate over 300 and 1200 baud serial lines with a host computer to exchange 'real time' chrono data from this single purpose product. Once configured, you could ask it "ATRT" and retrieve "123015P" (12:30pm and 15 seconds) over the serial port. You could also set an alarm and have the chrono raise the ring-indicator (pin 22) high when the set time had been triggered.

I suppose there were practical uses for it, but there were better things to spend money on in the 80s. Such as a computer. You might create a 'time card' application for a small business, coordinate transmission of file transfers for off-peak hours to save toll charges, or guarantee that logged events (such as user logons) were recorded according to an out-of-band, but accurate and controlled time source. The industry and the pages of Compute, BYTE, and other magazines were full of products like the Chronograph.

COMPUTE had articles too

The first half of Issue #2 contained just three or four general (multi-platform applicable) articles, and many that were PET specific. More than likely, the editors had to push content around in an attempt to have the *Buyers Guide* act as a separator between the general magazine and the Gazette portion, the latter of which was indeed platform focused. Compute did something similar in Issue #1 as well, but quickly gave up on the idea eventually.

In the early months of the publication, they struggled to get their hands on Apple and Atari content, logging 2.5 pages for the Apple and 7.5 pages for Atari in this issue (though the first 4.5 pgs was a comparison of Atari and PET BASICs).

Compute would not have this problem for long; by mid way through their first year, the quality and quantity of articles improved dramatically and they likely had a backlog of cutting room floor content to choose from. They eventually started paying authors by the page ($50), and by January of 1981, had moved to a monthly format.

Here is a list of of articles from the first 'half' of Issue #2, a few of which are tagged (x) for discussion below.

	Sorting Sorts, Part 2 by Rick and Belinda Hulon	PET*	3.5 pgs
	Memory Partition of Basic Workspace by Harvey B. Herman	PET	2 pgs
	Home Accounting, Plus An Easier Method of Saving Data by Robert W. Baker	PET	4 pgs
	Word Processing. A User Manual of Reviews - Part 2 by Len Lindsay	PET	5 pgs
	Book Review: 6502 Assembly Language Programming by Jim Butterfield	general 6502	1 pg
(x)	Machine Language Versus BASIC: Prime Number Generation / AIM 65 by Marvin L De Jong	AIM 65*	2 pgs
	BASIC Memory Map (Page 0): Aim, Kim, Sym, PET, Apple compiled by Jim Butterfield	Multiple	1 pg
(x)	Ramblin' by Roy O'Brien	PET	1 pg
	The Learning Lab by Marlene Pratto	PET*	1 pg
	Micros and the Handicapped by The Delmarva Computer Club	general	1 pg

* one or more of the examples referenced are applicable or convertible to other platforms

A closer look at Marvin L De Jong's Prime Number Generation article

Dr. Marvin De Jong was a Physics professor at The School of the Ozarks in Pt. Lookout, Missouri and authored a series of books on Computer Science, Physics, and Mathematics in addition to a handful of articles for various publications.

In this article, Dr. Marvin provides commented 6502 source code and an explanation supporting an algorithm to calculate prime numbers of the form 2^N-1.

The code provided was designed for an AIM 65, but the platform specific portion is limited to a single JSR to print the ASCII character (number) from the accumulator to the screen; this code should easily port to a Commodore 64, 1 Mhz. reference system using CHROUT $FFD2 and to a C256U Foenix system using PUTC $00:1018.

The author introduces the subject recalling that one of his students was searching for 'perfect numbers' and wrote a BASIC program for an Apple that took 11 hrs to produce a desired result. In response, Dr. De Jong rewrote the program in assembly language and executed it on the AIM 65, requiring only 11 *minutes* of run time.

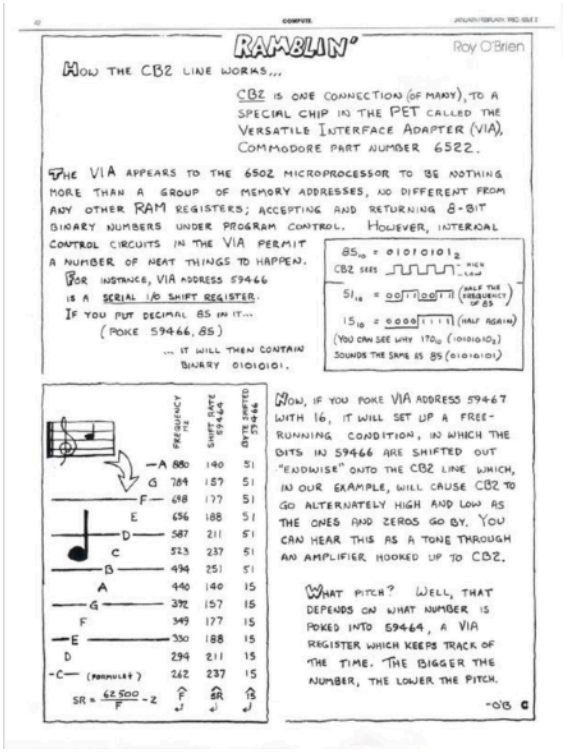
Commented source is provided for the 86 instruction program; based on the text, the program requires an additional 3K of working space in order to store the resulting number in BCD. I won't pretend to be able to understand how he goes about generating said large prime numbers, but you can :). What I will do is assemble it and post it on the Foenix Marketplace as part of the September update. I'll include relative timing in the notes on the Marketplace as well.

Following this (his first article published in Compute), Dr. De Jong went on to write others including:

- Experimenting With The 6551 ACIA - March, 1981
- Computer Communications Experiments - March, 1981
- A Floating-Point Binary To BCD Routine - April, 1981
- A General Purpose BCD-To-Binary Routine - October, 1981
- Plotting Polar Graphs With The Apple II - February 1982

"RAMBLIN'" by Roy O'Brien - what is this all about?!

Roy O'Brien's article stands out, not only because is one of the few music related articles in this early day of Compute, but because his article is entirely hand penned, including a table detailing musical notes, frequency in Hz., and shift rates for addresses that map to the PETs 6522 versatile interface adapter (VIA) to yield magic.


RAMBLIN'

Roy O'Brien

HOW THE CB2 LINE WORKS...

CB2 IS ONE CONNECTION (OF MANY), TO A SPECIAL CHIP IN THE PET CALLED THE VERSATILE INTERFACE ADAPTER (VIA), COMMODORE PART NUMBER 6522.

THE VIA APPEARS TO THE 6502 MICROPROCESSOR TO BE NOTHING MORE THAN A GROUP OF MEMORY ADDRESSES, NO DIFFERENT FROM ANY OTHER RAM REGISTERS; ACCEPTING AND RETURNING 8-BIT BINARY NUMBERS UNDER PROGRAM CONTROL. HOWEVER, INTERNAL CONTROL CIRCUITS IN THE VIA PERMIT A NUMBER OF NEAT THINGS TO HAPPEN. FOR INSTANCE, VIA ADDRESS 59466 IS A SERIAL I/O SHIFT REGISTER. IF YOU PUT DECIMAL 85 IN IT... (POKE 59466,85)

... IT WILL THEN CONTAIN BINARY 01010101.

$85_{10} = 01010101_2$

CB2 SEES

$51_{10} = 00110011$ (HALF THE FREQUENCY OF 85)

$15_{10} = 00001111$ (HALF AGAIN)

(YOU CAN SEE WHY 170_{10} (10101010_2) SOUNDS THE SAME AS 85 (01010101)

	FREQUENCY Hz	SHIFT RATE 59466	BYTE SHIFTED 59466
A	880	140	51
G	784	157	51
F	698	177	51
E	656	188	51
D	587	211	51
C	523	237	51
B	494	251	51
A	440	140	15
G	392	157	15
F	349	177	15
E	330	188	15
D	294	211	15
C (FORMULA?)	262	237	15

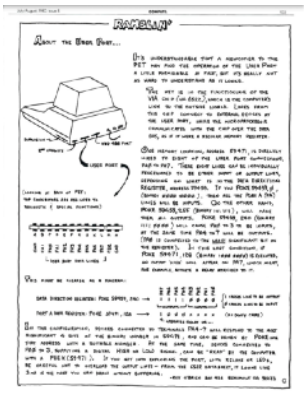
$SR = \frac{62,500}{F} - 2$

NOW, IF YOU POKE VIA ADDRESS 59467 WITH 16, IT WILL SET UP A FREE-RUNNING CONDITION, IN WHICH THE BITS IN 59466 ARE SHIFTED OUT "ENDWISE" ONTO THE CB2 LINE WHICH, IN OUR EXAMPLE, WILL CAUSE CB2 TO GO ALTERNATELY HIGH AND LOW AS THE ONES AND ZEROS GO BY. YOU CAN HEAR THIS AS A TONE THROUGH AN AMPLIFIER HOOKED UP TO CB2.

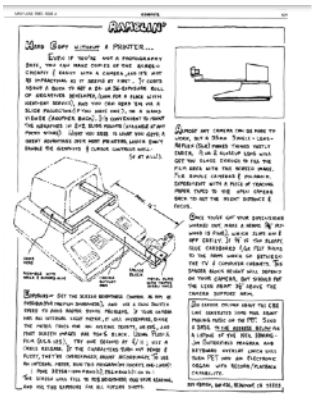
WHAT PITCH? WELL, THAT DEPENDS ON WHAT NUMBER IS POKED INTO 59464, A VIA REGISTER WHICH KEEPS TRACK OF THE TIME. THE BIGGER THE NUMBER, THE LOWER THE PITCH.

-OB ©

The 2nd reason this articles appeals to me is that the VIC20 (which would have basic tone generation capabilities), was not even announced at the time of this writing and while other platforms from competing manufacturers did offer audio capabilities, this method generated a fair amount of excitement for PET owners.


RAMBLIN'

About The Book Over...


RAMBLIN'

Using Copy Withhold A Printer...

Finally, this article appeals to me because it is hand drawn; the publisher probably had no other means of easily producing the table and diagrams.

Mr. O'Brien would go on and publish more in subsequent issues; to the left is a sample of two others, published during Compute's first year. The camera idea is particularly entertaining; look for these on archive.org.

Next issue, we will pick up where we left off, examining a few more vintage ads and looking more closely at the 'Gazette' section of Compute, issue #2. Until then …