Foenix Rising

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

Issue #13 - Classic formula December 2023

Long format return

It's that time again! Not only is it the "*Toys 'R Us time of year*", it's also time for a full issue of Foenix Rising to close out 2023.

This year brought continued growth to Foenix platforms and to retro-tech in general. Lots of software projects, feature releases, and new hardware options to choose from.

Through November, five developer-centric FLASH! issues were published ranging from 6-11 pages in length; this issue returns to the classic multi-article format and more variety.

Guest contributor Ernesto Contreras is back with a graphically oriented feature. We also interview Boisy Pitre about development of his NitrOS-9 Level 2 port, time working at Microware, and his other interests.

If you've been around Foenix for a while but were not aware of this Newsletter, click here to download a full copy of Issue #4 (the most recent *full* issue - from Nov./Dec. of 2022). It's always interesting to see where we were 12 months ago. And don't forget our YouTube channel and Content Store; links on page 2.

Wishing you a relaxing year-end with family, friends, and your Foenix systems.

My winter break project

You should not be surprised to hear that I have other things going on in my life; we all do. So let's talk about something completely different; Commercial arcade games :)

While at VCF Midwest in September, a Foenix-friend and I visited the Galloping Ghost Arcade; said to be the largest in the U.S., with 958 games on the floor. Not only do they have all of the classics, but also, games you've never heard of.

Oddly, I was drawn like a magnet to a mid-'90s arcade driver called "The Great 1,000 Mile Rally", based on the Mille Miglia, released by a little known Japanese company called Kaneko.

As happens, sparks of mild interest lead inevitably to visions of arcade grandeur, and I found myself on eBay buying a Jamma equipped board-set; then to Amazon for a wiring harness and a new power supply. Then back to eBay for the beat-up hobo arcade steering wheel (and pedal) that you see on the right. Of course, I'll need to paint it red.

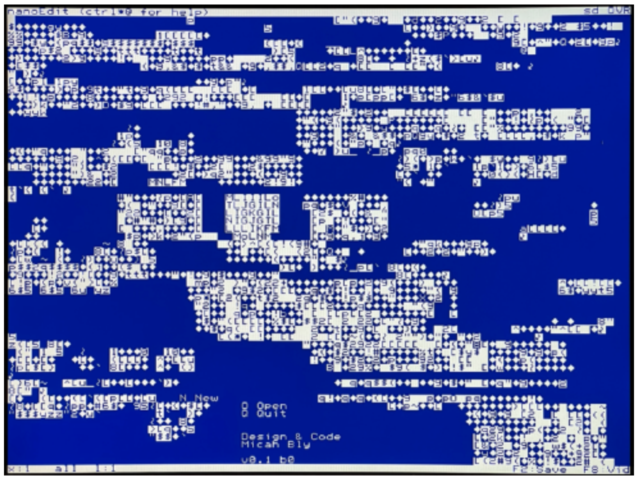
I thank this video for what will either turn into an expensive time sink, or an extremely satisfying addition to my home arcade that my family (me) will enjoy for years to come. Stay tuned for an update later …

VTOC - volume table of contents

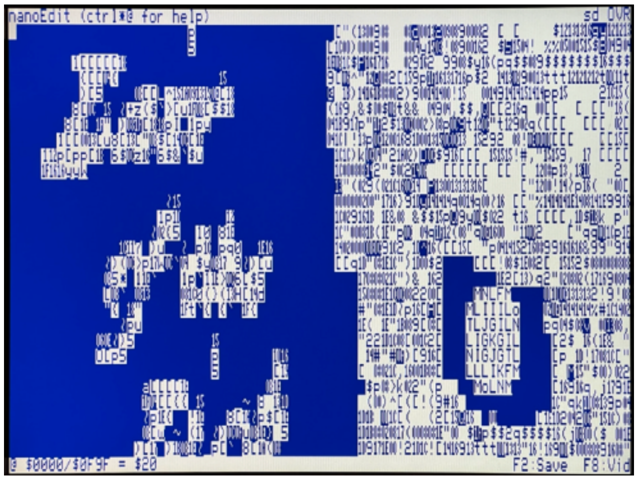
What do these pics have in common?

This is not a test, it's a humble brag and a bit of an advert - nanoEdit's 'data' mode take shape on pg. 23

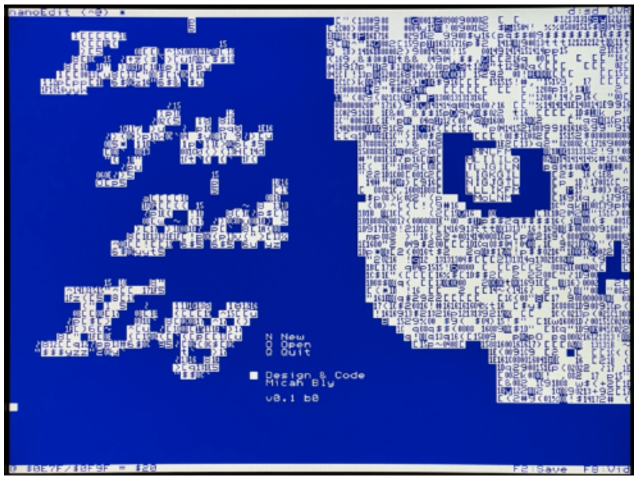
Micah Bly's soon to be released "Lair of the Lich King" is pictured on the bottom right. The 80 x 50 text file (4000 bytes) was harvested from a prior version and pulled into nanoEdit's data mode as a test case. nanoEdit is still in development and will be for some time, but as I hope you can see, it will be a great programmer's aid or general purpose text editor.



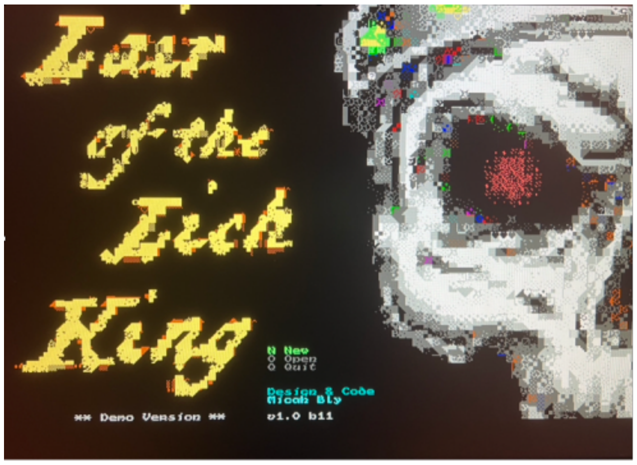
In unadorned 80 x 60 '**text**' mode (note the bottom left status line), data is all over the place; much of it unprintable, some replaced with tokens; word-wrap "does its worst".



In 80 x 30 '**data**' mode, the image begins to take shape. Unprintable characters are represented by custom chars indicating hexadecimal byte values. The aspect ratio is off, but you get the idea.



Same as above right but in 80 x 60 mode. Micah's game is the subject of a few jabs in the two-page article below and was the perfect example to demonstrate a few of nanoEdit's new features in context.



"Lair of the Lich King" is a multi-level rogue-like dungeon crawl adventure game that takes advantage of the F256 80 x 60 character screen and redefined characters. It has a deep rule set, leverages a random playfield generator, and will keep you amused, engaged, and terrified for hours.

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

Distribution: ~2^{10}-1

Published by EMwhite (discord and elsewhere)
Motto: 'Beware of programmers with screwdrivers'
other motto: '... because details matter'
Correspondance: `commerce@emwhite.org`

Foenix Resources for reference or to get started

- Foenix Retro Systems Home Page
- Foenix Discord Invite
- Stefany Allaire Patreon Page
- Stefany Allaire Twitter
- Foenix Marketplace content 'store': home of Foenix Rising
- Foenix Wiki (in-build but improving every day)
- GitHub hosted 'home' for doc, code, and data

MicroKernal DOS & more ...

A practical command primer, a look at binary file types, and sample MicroKernel code "for the rest of us"

(Thank you to Celton, dwsJason, and Gadget for helping me troubleshoot and ultimately, round out my knowledge on these topics)

Introduction - to use, type "`/DOS`" from SuperBASIC

By typing these four simple keystrokes and pressing <enter>, you can escape SuperBASIC and enter a world where you are afforded a variety of additional commands for managing your F256K, including:

- traditional disk functions including formatting
- viewing flash memory banks installed
- creating text files (BBS style) or dumping the contents of a file in ASCII or hexadecimal byte form
- testing the keyboard
- configuring the optional WiFi interface

This somewhat hidden layer sits somewhere between the utility of a Commodore'esque DOS wedge, and the command set included in MCP (the operating environment for A2560 family machines, written by Peter Weingartner).

The drive numbering system is identical to the convention used within SuperBASIC, namely, drive '0' for the built-in SD interface, and drive numbers '1', '2', and so on, for IEC bus connected peripherals.

Figure 3a represents MicroKernel DOS help; if you are running an older kernel and see something different, check pg. 2 for information on updating your system.

Pro tip: the update of FLASH blocks containing the F256 Kernel, DOS layer, SuperBASIC, and other operating code and utilities is pushed through the F256 USB-mini interface using a simple cable to your host. From a software perspective, you will need either the *FoenixMgr* framework (requires Python), or the Windows based *F256 Uploader* application*.

This is different from the VICKY (FPGA) update procedure which requires a 'blaster' device connected to the 10-pin (2 x 5) JTAG header and the use of the Intel/Altera Quartus software.

Running .PGX files

There is now a vetted method for executing .PGX or .PGZ files. You an do this from SuperBASIC's screen editor or from DOS (with a slight syntax variation). From SuperBASIC type the following:

convention

```
/- {program}.PGX
```

mind the gap; a space is required

A .PGX is an executable file similar to a '.com' or '.exe' on Microsoft DOS or CP/M systems. It might also be compared to a .PRG file on vintage Commodore systems, however the latter is primitive by comparison. At the lowest level, these types of files have headers identifying the load address. In the case of .PGX, this is also the execution address and qualifying machine information.

.PGZ files take this a step further and support multi-part or multi-segment files. The rest of this article will discuss .PGX files; we'll get to .PGZ another time.

Info on .PGX files can read about on the Foenix Wiki here. Here is a F256 appropriate sample header:

Offset	Count	Example	Purpose
0	3	"PGX"	Signature
3	1	$03	CPU
4	4	$08 $40 $-- $--	Destination addr
8	-	$20 $21 $12 ...	Data to load

Note: CPU type $01 (as detailed on the linked page) represents the WDC 65816 CPU; $02 is 68K family; $03 is the 6502, proper for the F256 platform.

Generic `.bin` files, on the other hand, are binary files with no identifying load address or embedded execution information. `.bin` files are either pushed into a specific address with the F256 Uploader/Updater utility or loaded to a specified address using BLOAD in SuperBASIC.

Still, a 3rd type of binary contains an auto-execute header, akin to the Commodore 64 "CBM80" cartridge standard. Called a 'KUP' (kernel-user-program) it will startup on reset when detected in low-RAM (if DIP 1 is set), expansion RAM, or in a flash cart or memory.

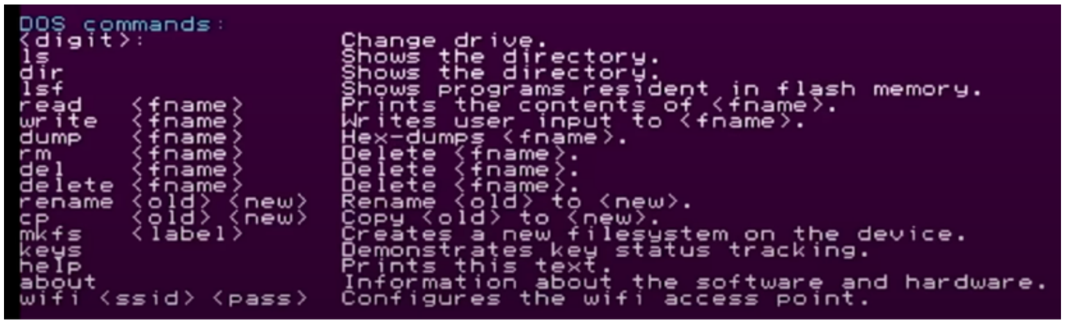
MicroKernel DOS Command overview

We won't cover all of the commands, but have comments on several and have included links to other sources.

`- {program}.PGX` - execute code from the default path. As from SuperBASIC, the space is required.

Change Drive - by typing a number [`0` .. `4`] followed by a colon, you change the drive which will be acted upon as default (the prompt will change). `0` corresponds to the built-in SD card (if present), and `1` .. `4` maps to IEC bus devices `8` .. `11` respectively.

`ls` or `dir` - either command will list the directory starting with the disk label (if assigned) followed by one line per file and its size in blocks in hexadecimal format.



```
DOS commands:
<digit>:                Change drive.
ls                      Shows the directory.
dir                     Shows the directory.
lsf                     Shows programs resident in flash memory.
read    <fname>         Prints the contents of <fname>.
write   <fname>         Writes user input to <fname>.
dump    <fname>         Hex-dumps <fname>.
rm      <fname>         Delete <fname>.
del     <fname>         Delete <fname>.
delete  <fname>         Delete <fname>.
rename  <old> <new>     Rename <old> to <new>.
cp      <old> <new>     Copy <old> to <new>.
mkfs    <label>         Creates a new filesystem on the device.
keys                    Demonstrates key status tracking.
help                    Prints this text.
about                   Information about the software and hardware.
wifi <ssid> <pass>      Configures the wifi access point.
```

Figure 3a - DOS help

One SD disk block is equivalent to 256 bytes of data, however, on IEC devices, a disk block is 254 bytes of data. Fun fact: the ancient Commodore disk format

* Perifractic demonstrates the Windows based F256 Uploader here. You can also use it to write your own programs to flash!

obeyed by JiffyDOS in your FNX1591 (and other IEC devices) used bytes 0 and 1 of each sector to link to the *next* track and sector. Track and Sector editors such as "Disk Doctor" pictured on page 16 of issue #4 gave the user the power to interrogate and edit data on disk sectors directly. Perhaps some enterprising individual will write a T+S editor for the F256 platform?

`write {file}` - create/write a text file to disk using a "bbs" style line editor. Note that I am calling the line editor *bbs-style;* there is no doc suggesting this, but I have a few clues that Gadget spent a fair amount of time on `tty` connected systems in the good old days.

The line editor opens tabbed to column 4 (these spaces will not appear in your file) and allows printable ASCII characters to be entered, up to 75 characters per line. Each line is terminated with a carriage return aka ASCII `0x0d` (which **is** added to your file) and you may continue to append until '`.`' is entered on a blank line (the '`.`' and final return is **not** added to your file).

`dump {file}` - read and display hexadecimal values, 16 at a time until end of file is reached.

`read {file}` - read and display ASCII values until end-of-file is reached. Bonus feature: there is no harm is displaying binary data using this command since graphic glyphs are bound to all 256 ASCII values. The only character interpreted by DOS's simple output routine is ASCII 13 (carriage return). Hence, you will never experience the peculiar behavior such as screen clearing, odd colors or erratic cursor motion as experienced on legacy platforms or `vt*` type ASCII terminals.

To relive the power of leveraging inline control codes in a PETSCII context for constructive means, see this video :)

retro remnant: land-line warfare

My generation was scarred by 'inhabitants' purposely picking up the phone extension at home, intentionally interrupting the glory of 300 bps analog modem surfing. Characters became garbled, throwing ctrl-codes to the terminal, trashing the display or worse, switching to a foreign DEC character set. THE WORST, was having to restart a binary download!

`lsf` - generates a list of programs resident in flash memory. The graphic and callouts below provide an example 'build'. This might be considered an advanced topic, but it's good to have working knowledge for the simple reason that a misstep during a code push can 'brick' your system. But rest assured, mishaps are easily corrected and there are plenty of advanced users on Discord eager to help. In my case, I learned that uploading an 8,192 byte file of nulls could be used to disable an errant auto-start image Our community is a full circle of beginner, intermediate, and advanced users and developers. We depend upon, and help each other every day. Yet another reason why the Foenix platform is unique.

`rm` or `del` or `delete {file}` - allows deletion of a file. It is possible to explicitly identify a drive which is different from the default by preceding the file name with a drive number: "`rm 2:{file}`" as an example to remove the named *file* from IEC device #9.

`keys` - invokes a utility which tests the use/action of your input devices (keyboard and joysticks) against kernel event routines. From a practical standpoint, this utility does not serve much of a purpose except to check for broken switches or keys, or to highlight anomalies in PS/2 support, but it's an excellent example of working code which is built using MicroKernel services. Example sources are freely available in a GitHub repository, discussed on the next page and used as an example project in this article.

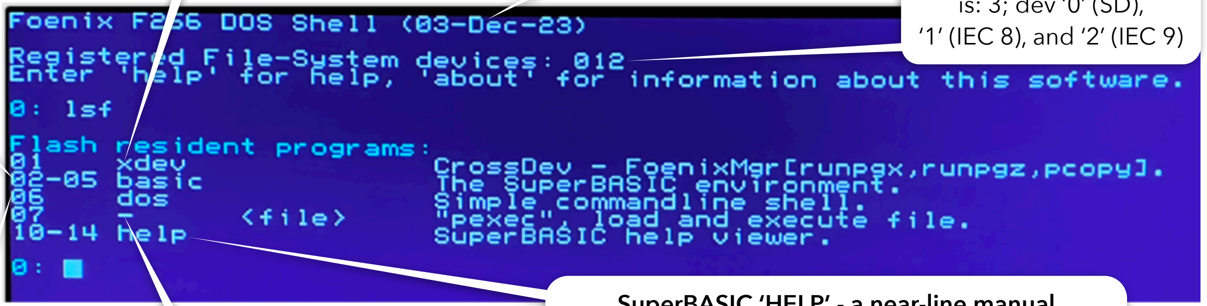
`lsf` **Command deep-dive**
Figure 4a

`xdev` framework
Tools to support cross-development and specifically, file transfer from host to your F256 platform SD device

This is the DOS shell release date
(corresponds to flash block 06 in this example; this is the binary for the DOS shell, itself)

`SuperBASIC`
4 x 8k blocks containing binaries

# of registered devices is: 3; dev '0' (SD), '1' (IEC 8), and '2' (IEC 9)



```
Foenix F256 DOS Shell (03-Dec-23)
Registered File-System devices: 012
Enter 'help' for help, 'about' for information about this software.
0: lsf
Flash resident programs:
01     xdev                  CrossDev - FoenixMgr[runpgx,runpgz,pcopy].
02-05  basic                 The SuperBASIC environment.
06     dos                   Simple commandline shell.
07     -        <file>       "pexec", load and execute file.
10-14  help                  SuperBASIC help viewer.
0: 
```

`DOS` binaries
(The interpreter and code described in this article)

'**–**' pexec binary
Callable from SuperBASIC or DOS to load and exec `.PGX` binaries

SuperBASIC 'HELP' - a near-line manual
A rudimentary viewer and the full text of most of the manual bound within this 5 block package; recallable by typing "**/HELP**" from SuperBASIC (note that if you have an unsaved program, you will be asked to confirm before exiting)

But where is MicroKernel? - `lsf` does not provide the visual; however, in the current distribution (as of December 2023), MicroKernel occupies 5 blocks ($3b .. $3f); ***Quiz***: *how much flash is left for the user? Answer: plenty (47 x 8k blocks !!)*

`wifi {ssid} {passcode}` - assists in configuring the ESP Feather board with your WiFi network and password, if installed. I do not own one (yet) but here are a pair of YouTube videos that discuss this topic:

- Foenix Discord user ‘PJW’ published this video in mid-July of 2023; access it here. This video focuses on the software aspect of the job and is highly detailed and comprehensive.
- Foenix Discord user ‘1Bit Fever Dreams’ published a video in early December of 2023; it can be accessed here. It covers a number of new use cases and a super close up view at soldering steps. For visual learners (I'm one), this will be invaluable.

I recommend watching both before attempting the upgrade. In addition, as pointed out in the 2nd video, there is a new option available to those purchasing their systems, to buy them WiFi enabled for a modest additional charge.

MicroKernel Code samples

When Gadget released DOS, her intention was twofold. On one hand, she knew that SuperBASIC was, well… basic; disk utilities were absent, as were utilities for developers interested in interrogating files (the ability to dump file data in hex format), for example.

She has also pointed to DOS as a kernel programming example; it demonstrates event use and device access code for at least 85% of the functions.

A third unintended benefit of hosting an evolving platform utility within DOS is to keep SuperBASIC confined and focused, thus leaving more memory for user-developed BASIC programs. This will allow DOS to expand over time, serving as a MCP-lite environment where other applications can be integrated and launched.

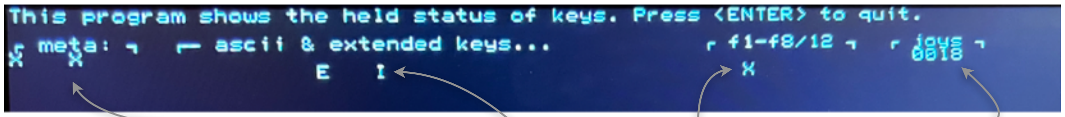
Let’s conduct a mini case study of the DOS “keys”utility, (inside-out) in an effort to provide a MicroKernel starter app, and to answer one of the most oft asked questions about writing code for the F256: “how do I scan/read the keyboard and joysticks. Is there any example code?”.

The answer is: “It’s easy”, and “absolutely”; not only are the full sources for DOS published in this publicly available GitHub repository, but we’ve extracted “keys” and wrapped it within a small .PGX executable, specifically to make it more lean and understandable.

I took some liberty in the process, simplifying some amount of Gadget’s original code to lessen the dependancies on associated display code, and have also reduced the use of sections and namespaces. I’ll point out a few highlights in the code listing below.

Before looking at the source code, I suggest running the DOS utility yourself by typing “keys” at the DOS prompt. Once in the app, press keys and various combinations of meta-keys (also, a joystick) and observe the behavior. Notice key combinations, and how it works from a mapping and user interface perspective. This will come in handy when you study the code later.

The following partial screen shot shows the finished product. A simple line of descriptive text was added to frame the data by ‘type’. This is just window dressing, but adds flair. Remember our motto: “details matter”.


```
This program shows the held status of keys. Press <ENTER> to quit.
X meta: ¬   ⌐ ascii & extended keys...              ⌐ f1-f8/12 ¬   ⌐ joys ¬
   X                                                                 0018
                 E   I                                    X
```

left-shift; ALT pressed after the fact
ordinary keys (however, upper case)
F4
left nibbles (“00”) is JOY1; right two (“18”) is JOY2 representing button #1 (“18”) and RIGHT (“18”)

MicroKernel Use Prereqs

Three steps, and then you may start calling functions to process events or otherwise address devices. It should be stated that not all of MicroKernel’s vectors leverage argument passing; `kernel.Display.Reset` at `$ffcc` for example, is one that does not. This is among the first things we do (line #20) in the code below (pg. 8).

Ok, here’s the drill:

Step #1: declare `event` struct vars - in 64tass, we do this by adding the following single statement to our source:

```
event .dstruct    kernel.event.event_t
```

This line informs the assembler of MicroKernel data structures (imported from the `api.asm` include file in step #3). Within your program, memory is required to house values such as `event.key.ascii` (see the example below). You might notice in reviewing this, a range of addresses overlap beginning at `$418c`; this memory is reused and is populated based on the event type. In your code, you simply access elements as appropriate after `NextEvent` returns a given type.

single occurrence of $4189-$418b

$418c is repeated

```
4189    event     .dstruct   kernel.event.event_t
4189    type      .byte   ?  ; enum above
418a    buf       .byte   ?  ; page id or zero
418b    ext       .byte   ?  ; page id or zero
418c    key       .dstruct   kernel.event.key_t
418c    keyboard  .byte   ?  ; keyboard ID
418d    raw       .byte   ?  ; Raw key ID
418e    ascii     .byte   ?  ; ASCII value
418f    flags     .byte   ?  ; Flags (META)
418c    joystick  .dstruct   kernel.event.joystick_t
418c    joy0      .byte   ?
418d    joy1      .byte   ?
418c    timer     .dstruct   kernel.event.timer_t
418c    value     .byte   ?
418d    cookie    .byte   ?
418c    tcp       .dstruct   kernel.event.tcp_t
418c    len       .byte   ?  ; Raw packet length
```

Step #2: Inform the kernel of the address of `event`. In step #1, we provided the assembler with details on how we intend to address data we are accessing, here we tell the kernel where it lives, using `<#` and `#>` directives.

The low-byte and high-byte values are stored to `kernel.args.event` and `kernel.args.event+1`.

There is nothing more to discuss here, it is taken care of on lines 24-27 and never worried about again.

Here is the code (again, from our '*sak*' example aka stand-alone-keys). This is unchanged from the original:

```
        lda    #<event
        sta    kernel.args.event+0
        lda    #>event
        sta    kernel.args.event+1
```

Step #3: Add `api.asm` to your Makefile, or 'include' it directly into your source code (this will vary based on the assembler being used; If you are a C lang developer, there are `.h` (header) and `.lib` files in the cc65 folder of the repo but here is how you do it in 64tass):

```
        .include     "kernel/api.asm"
```

That's it for setup. Here is a preview of 'use' code aka the event loop (discussed in detail in the code listing below):

```
_loop  jsr   kernel.Yield   <---
       jsr   kernelNextEvent
       bcs   _loop
       lda   event.type
       cmp   #kernel.event.key.PRESSED
       beq   _pressed
       cmp   #kernel.event.key.RELEASED
       beq   _released
       cmp   #kernel.event.JOYSTICK
       beq   _joy
       bra   loop
```

'yielding' is an interesting topic (to be discussed in future); for now just know that this gives some time back to the kernel

The following few sections delve into advanced use of 64tass and expand upon Gadget's use of the assembler's more interesting features. It is *not* absolutely necessary to gain this understanding to use MicroKernel effectively, but it will help you down the road, especially if the complexity of your own programs grow. It will also help you as you begin to borrow and maintain code from Gadget and other Foenix developers. I am just coming to grips with it, myself. In my opinion, it's worth the focus and effort.

Memory use and the manifestation of `event`

The following 3 pages contain an excerpt of the 64tass list output file for this project. If you've looked at these prior, you'll know they are useful for debugging, but tend to be oddly formatted and difficult to read; we've improved this by inserting line numbers, aligning cols, and annotating some key points including the struct hierarchy.

From a memory use perspective, you'll note that the code portion of the program (in blue text) is only 67 lines long, and it consumes 137 bytes of memory.

Meanwhile, data (highlighted in **bolded purple** text) is 256 bytes in total. In summary, here is the footprint:

.PGX header	8 bytes	\$4000 - \$4007
— code —	137 bytes	\$4008 - \$4090
'digits' data	16 bytes	\$4091 - \$40a0
Welcome text*	232 bytes (at end)	\$40a1 - \$4188

So, as far as 64tass is concerned, the binary file produced is 393 bytes. That's cool, but how do the `event` vars figure into the mix?

You'll have to study the .lst file to see what's really going on; the **orange** text on the next page represents the MicroKernel `event` struct definition in full, and as discussed, the act of adding `api.asm` to your code and declaring the datatype welcomes it to your program.

A key point to understand: the define struct directive (`.dstruct`) does not reserve any memory, it merely establishes the structure as a template so the assembler can do its thing and resolve references. If you look on the next page, you'll see that the section begins at address `$4189` which is the first byte following the welcome message termination null (`$00`) on line #89.

It is strongly suggested to take care and **avoid** using this memory for something else such as storing local/temp variables. Doing so would be the equivalent of declaring a `struct` in the C language, calling `malloc()` to grab memory from the heap (thus obtaining a pointer to it), then writing to the address explicitly elsewhere in your program (creating a mess). So use the '?' directive religiously to avoid this. 64tass will keep track of memory, and keep you out of trouble.

Where are the MicroKernel Vectors? (entry points)

`api.asm` also includes directives to calculate vectors along with a slew of constant definitions to make life easier. We stop short of including it in this article, but you'll see the addresses explicitly detailed within a contiguous area of memory near the end of bank 7 (`$e000-$ffff`) in the .lst.

What about writing to devices and setting attributes?

Also in the .lst (also not captured below) you'll see a ton of detail concerning devices that accept parameters such as network devices, common disk functions, timer settings, and char 'drawing'. In this case, MicroKernel uses a structure similar to `event_t`, but living in a dedicated and reserved area of memory at the top of zero page (`$F0-$FF`); the important point aside from the fact that the overlap scheme is similar, is that this 16 byte area of memory is the **only** portion of zero page that MicroKernel needs. The "memory model" section of this README, covers this topic in detail. The doc is a must-read; it is concise and extremely well written.

Our example doesn't use any of these vectors so we won't cover it in this article, but have a look at the full repo, and look specifically at `reader.asm` and `cmd_wifi.asm` for disk read and TCP/IP examples, respectively.

*`welcomemsg` contains whitespace padding, required so text wraps at 80 cols (yes, this is wasteful). This approach was chosen to keep the display routine dead-simple, versus being clever with pointers, interpretation of new lines, etc.

In the real world, you might use a simple or full featured display library as Gadget's original "keys" program did.

```
; 64tass Turbo Assembler Macro V1.58.2974 listing file
; 64tass -I . -C -Wall -Wno-shadow -x --verbose-list -b -L standalonekeys.lst -o standalonekeys.bin standalonekeys.asm api.asm

; Wed Dec 06 18:55:20 2023
;Offset ;Hex ;Monitor ;Source

;******  Processing input file: standalonekeys.asm

00 .cpu    "65c02"
01 *        = $4000

02 >4000 50 47 58       .text     "PGX"        ; Signature
03 >4003 03             .byte     $03          ; machine type
04 >4004 08 40          .word     keys.cmd     ; execution addr
05 >4006 00 00          .word     0            ; 3rd and 4th byte of starting addr not used on 8-bit systems

06 .dsection   code

07 .virtual    $0000    ; Zero page
08 >0000 mmu_ctrl       .byte       ?
09 >0001 io_ctrl        .byte       ?
10 >0002 reserved       .fill       6
11 >0008 mmu            .fill       8  ;merely (and merrily) reserves addresses.  Aside
12 >0010 printptr       .word       ?  ;  from $00 and $01, $08..$0f are the most critical
13 .dsection   dp
14 .endv

15 .4189 event                 .dstruct    kernel.event.event_t
   >4189 type                  .byte   ?   ; Enum above
   >418a buf                   .byte   ?   ; page id or zero
   >418b ext                   .byte   ?   ; page id or zero
   .union

   .418c key                   .dstruct    kernel.event.key_t
   >418c keyboard              .byte   ?   ; Keyboard ID
   >418d raw                   .byte   ?   ; Raw key ID
   >418e ascii                 .byte   ?   ; ASCII value
   >418f flags                 .byte   ?   ; Flags (META)
   =$80 META                   = $80       ; Meta key; no associated ASCII value
   .ends

   .418c mouse ①               .dstruct    kernel.event.mouse_t
   .union

   .418c delta                 .dstruct    kernel.event.m_delta_t
   >418c x                     .byte   ?
   >418d y                     .byte   ?
   >418e z                     .byte   ?
   >418f buttons               .byte   ?
   .ends

   .418c clicks ②              .dstruct    kernel.event.m_clicks_t
   >418c inner ③               .byte   ?
   >418d middle                .byte   ?
   >418e outer                 .byte   ?
   .ends
   .endu
   .ends

   .418c joystick              .dstruct    kernel.event.joystick_t
   >418c joy0                  .byte   ?
   >418d joy1                  .byte   ?
   .ends

   .418c udp                   .dstruct    kernel.event.udp_t
   >418c token                 .byte   ?   ; TODO: break out into fields
   .ends

   .418c tcp                   .dstruct    kernel.event.tcp_t
   >418c len                   .byte   ?   ; Raw packet length.
   .ends

   .418c file                  .dstruct    kernel.event.file_t
   >418c stream                .byte   ?
   >418d cookie                .byte   ?

   .union
   .418e data                  .dstruct    kernel.event.fs_data_t
   >418e requested             .byte   ?   ; Requested number of bytes to read
   >418f read                  .byte   ?   ; Number of bytes actually read
   .ends
```

(01) Origin address of `$4000` chosen because it will co-exist with SuperBASIC

(02-05) The PGX signature occupies 8 bytes. When loaded into memory, these bytes are not placed in memory, only our program will (beginning at $4008).

(07-14) This virtual definition reserves the first 16 bytes of memory while 'naming' `mmu_ctrl` and `io_ctrl`, only the latter of which is used in this program. On line 12, we define `.word printptr`, used by `printmsg`.

(15) As discussed above, this single line is expanded to the full `event` structure up to line 16 on the next page

use example

The `mouse` struct, at ① will be populated with data upon occurrence of a **DELTA** event or a **CLICKS** event. The **upper case** constants are built at assemble time from `api.asm`. A compare example in context for a click event would be:

```
cmp #kernel.event.mouse.CLICKS
```

From a data hierarchy perspective, `mouse` is within the overarching `event` struct that we instantiated on line (15), and `clicks` ② contains `inner` ③ within. We would reference this byte as:

```
lda event.mouse.clicks.inner
```

It contains the # of clicks detected (single, double, or triple; 1, 2*, or 3 respectively)

It could be said that *nested structs* create hierarchies while *unions* (in green) facilitate overlap. To quote the 64tass manual (v1.59 r3120):

"Unions can be used for overlapping data as the compile offset and program counter remains the same on each line. Therefore the length of a union is the length of its longest item."

Combined, these features provide something between C language structs and the dot notation used in Python, Java, or C++. Powerful !!

* For source, .lst, and binary of *sak, see the* Foenix Marketplace here… You'll be pleased to see that mouse events have been added to the code. One killer (undocumented) kernel feature: it auto-detects 'handedness' (pg. 9) on double-click. Gadget thinks of everything!

```
.418e wrote            .dstruct    kernel.event.fs_wrote_t
>418e requested        .byte   ?   ; Requested number of bytes to read
>418f wrote            .byte   ?   ; Number of bytes actually read
.ends
.endu
.ends

.418c directory        .dstruct    kernel.event.dir_t
>418c stream           .byte   ?
>418d cookie           .byte   ?
.union

.418e volume           .dstruct    kernel.event.dir_vol_t
>418e len              .byte   ?   ; Length of volname (in buf)
>418f flags            .byte   ?   ; block size, text encoding
.ends

.418e file             .dstruct    kernel.event.dir_file_t
>418e len              .byte   ?
>418f flags            .byte   ?   ; block scale, text encoding, approx sz
.ends

.418e free             .dstruct    kernel.event.dir_free_t
>418e flags            .byte   ?   ; block scale, text encoding, approx sz
.ends
.endu
.ends

.418c timer            .dstruct    kernel.event.timer_t
>418c value            .byte   ?
>418d cookie           .byte   ?
.ends
.endu
.ends
```

```
16 keys                   .namespace
```

🟨 = MicroKernel Vectors

```
17 .section  code
18 .4008 a9 02            lda #$02          cmd        lda     #2
19 .400a 85 01            sta $01                      sta     io_ctrl
20 .400c 20 cc ff         jsr $ffcc                    jsr     kernel.Display.Reset
21 .400f a2 a1            ldx #$a1                     ldx     #<welcomemsg
22 .4011 a0 40            ldy #$40                     ldy     #>welcomemsg
23 .4013 20 80 40         jsr $4080                    jsr     printmsg
24 .4016 a9 89            lda #$89                     lda     #<event
25 .4018 85 f0            sta $f0                      sta     kernel.args.events+0
26 .401a a9 41            lda #$41                     lda     #>event
27 .401c 85 f1            sta $f1                      sta     kernel.args.events+1
28 .401e 20 0c ff         jsr $ff0c         _loop      jsr     kernel.Yield
29 .4021 20 00 ff         jsr $ff00                    jsr     kernel.NextEvent
30 .4024 b0 f8            bcs $401e                    bcs     _loop
31 .4026 ad 89 41         lda $4189                    lda     event.type
32 .4029 c9 08            cmp #$08                     cmp     #kernel.event.key.PRESSED
33 .402b f0 1e            beq $404b                    beq     _pressed
34 .402d c9 0a            cmp #$0a                     cmp     #kernel.event.key.RELEASED
35 .402f f0 16            beq $4047                    beq     _released
36 .4031 c9 04            cmp #$04                     cmp     #kernel.event.JOYSTICK
37 .4033 f0 02            beq $4037                    beq     _joy
38 .4035 80 e7            bra $401e                    bra     _loop
```

Discussed on pg. 6 above, the kernel clears the carry flag when an event is pending.

On line (31), the event type is loaded into the accumulator then compared with constants to branch on events

```
39 .4037 a2 00            ldx #$00          _joy       ldx     #0
40 .4039 ad 8c 41         lda $418c                    lda     event.joystick.joy0
41 .403c 20 66 40         jsr $4066                    jsr     print_hex
42 .403f ad 8d 41         lda $418d                    lda     event.joystick.joy1
43 .4042 20 66 40         jsr $4066                    jsr     print_hex
44 .4045 80 d7            bra $401e                    bra     _loop
```

`_joy` calls `print_hex` for each of the two joysticks (when either triggers an event) then branches back to the event loop (`_loop`)

```
45 .4047 a9 20            lda #$20          _released  lda     #' '
46 .4049 80 11            bra $405c                    bra     _show
47 .404b ac 8e 41         ldy $418e         _pressed   ldy     event.key.ascii
48 .404e c0 0d            cpy #$0d                     cpy     #13
49 .4050 f0 12            beq $4064                    beq     _done
50 .4052 a9 58            lda #$58                     lda     #'X'
51 .4054 2c 8f 41         bit $418f                    bit     event.key.flags
52 .4057 30 03            bmi $405c                    bmi     _show
53 .4059 ad 8e 41         lda $418e                    lda     event.key.ascii
54 .405c ac 8d 41         ldy $418d         _show      ldy     event.key.raw
55 .405f 99 f0 c0         sta $c0f0,y                  sta     $c0f0,y
56 .4062 80 ba            bra $401e                    bra     _loop
```

`_pressed` exits upon <return> and cleverly prints meta key flags, then keypresses beginning at screen location `$c0f0`.

The original “keys” program leveraged `display.asm` to write to the screen indirectly.

This was simplified to a static location to reduce overhead.

```
.4064 18            clc                   _done       clc
.4065 60            rts                               rts

.4066 48            pha                   print_hex   pha
.4067 4a            lsr a                             lsr     a
.4068 4a            lsr a                             lsr     a
.4069 4a            lsr a                             lsr     a
.406a 4a            lsr a                             lsr     a
.406b 20 75 40      jsr $4075                         jsr     _digit
.406e 68            pla                               pla
.406f 29 0f         and #$0f                          and     #$0f
.4071 20 75 40      jsr $4075                         jsr     _digit
.4074 60            rts                               rts

.4075 5a            phy                   _digit      phy
.4076 a8            tay                               tay
.4077 b9 91 40      lda $4091,y           lda         digits,y
.407a 7a            ply                               ply
.407b 9d 31 c1      sta $c131,x                       sta     $c131,x
.407e e8            inx                               inx
.407f 60            rts                               rts

.4080 86 10         stx $10               printmsg    stx     printptr
.4082 84 11         sty $11                           sty     printptr+1
.4084 a0 00         ldy #$00                          ldy     #$00
.4086 b1 10         lda ($10),y           printloop   lda     (printptr),y
.4088 f0 06         beq $4090                         beq     exitprint
.408a 99 00 c0      sta $c000,y                       sta     $c000,y
.408d c8            iny                               iny
.408e 80 f6         bra $4086                         bra     printloop
.4090 60            rts                   exitprint   rts

  >4091 30 31 32 33 34 35 36 37 38 39 61 62 63 64 65 66
digits      .text       "0123456789abcdef"
  >40a1 54 68 69 73 20 70 72 6f 67 72 61 6d 20 73 68 6f     >4121 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
  >40b1 77 73 20 74 68 65 20 68 65 6c 64 20 73 74 61 74     >4131 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
  >40c1 75 73 20 6f 66 20 6b 65 79 73 2e 20 50 72 65 73     >4141 a0 20 6d 65 74 61 3a 20 a1 20 20 a0 96 20 61 73
  >40d1 73 20 3c 45 4e 54 45 52 3e 20 74 6f 20 71 75 69     >4151 63 69 69 20 26 20 65 78 74 65 6e 64 65 64 20 6b
  >40e1 74 2e 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4161 65 79 73 2e 2e 2e 20 20 20 20 20 20 20 20 20 20
  >40f1 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4171 20 a0 20 66 31 2d 66 38 2f 31 32 20 a1 20 20 a0
  >4101 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4181 20 6a 6f 79 73 20 a1 00
  >4111 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
welcomemsg  .text       "This program shows the held status of keys. Press <ENTER> to quit.             "
            .text       "                                                                               "
            .text       $a0," meta: ",$a1,"  ",$a0,$96," ascii & extended keys...          ",$a0," f1-f8/12 ",$a1
            .text.      "  ",$a0," joys ",$a1,$0
.send
.endn
```

Exits

This simple print routine is single purpose and replaces the `.mkstr` macro (part of `display.asm`).

Characters are stored, indexed by y, beginning at the hard-coded `$c000` address. The text message below is nearly 3 full screen lines in length.

Interpreting Events

Now that we are detecting events, what are we going to do with them? Before deciding, we need to understand the circumstances that caused the kernel to inform us that a particular event has occurred. This primer will help.

JOYSTICK events are triggered each time a state change occurs, meaning, each time a button (or directional switch) is detected, **and** each time the state changes due to release of the button or joystick back to center. You can exercise this yourself with *sak* by moving a joystick in a particular direction or pressing a button, holding it, then releasing.

DELTA x and y events (mouse movement) measure velocity from a center of `0` either positive (`$01`, `$02`, `$03` … increasing) or negative (`$ff`, `$fe`, `$fd` …, decreasing). Down (y axis) and to the Right (x axis) are positive. The z axis is triggered by the scroll wheel (if your mouse has one), and reads `$ff` when scrolled forward and `$01` when scrolled backwards, regardless of velocity. buttons returns bit values of bit 0 = inner; bit 1 = outer; bit 2 = center aka, byte values of 1, 2, or 4. This byte returns to 0 when released, triggering an event for the action and another when there is no action.

Mouse CLICKS events are more complicated because they track single, double, and triple click. When actuated for either of these three conditions, the stack waits ~0.5s to determine how many (clicks) have occurred on the inner, middle, or outer button. At this point, a value of 1, 2, or 3 is placed into the appropriate register. Contrary to DELTA or JOYSTICK events, there is no 'unclick' event sent for CLICKS. You are notified of the last compound click event, that's it.

The notion of inner and outer are relative to the handedness of the mouse. While righty is the default (with the inner being the left button), all it takes is a double-click on the [then] outer button to reverse the orientation; doing so will turn your mouse into a left-handed HID (human interface device). Double-click the left button to return to 'righty'.

There is more to discuss on this topic, but we will have to save it for another time. It is highly recommended that you pull the binary and source from the marketplace, try the .PGX, study the source code, and then embark on your own path.

F256 Binary Files and Headers

A quick look at the ins and outs of 3 types of binary files including .PGX and the $F2, $56 auto-execute variety

On prior pages, we discussed the .PGX format but there are others to mention. Differences aside, most have two things in common: a) somebody though it a good idea to define a format and b) it wasn't just an idea; one or more people invested effort to create a loader, editor, or attribute viewer. With sound design and some amount of good fortune, formats and standards catch on; in rare cases, they form the basis of a solid ecosystem that contributes to make a good platform, great.

The most simple header I'm aware of, was the Commodore .PRG format. Kernal LOAD and SAVE vectors counted on it to inform the loading of binary and BASIC programs, disk directories, and even app data (e.g. SpeedScript documents). It did so with only two bytes of meta-data, representing the load address in low-byte/high-byte format. Unfortunately, nothing within this thin spec identified the execution address, and if you are familiar with the C64 and its ancestors, you'll probably know that the characters 'P', 'R', 'G' are not part of the filename at all; they manifested from byte 0 of a file's directory entry, occupying a column in the '$' output along with SEQ and REL. In the old days, if you didn't know where or how to load something, detective work was necessary. Proprietary loaders adorned nearly every multi-file program, especially commercial titles. Thankfully, standards have evolved since the '80s and our lives are improved in this regard.

Foenix platforms do not have an 8.3 filename spec or a system extension registry. Extensions are merely a few courteous characters at the end of a filename. In this one-pager, we will profile a few of these formats. Ultimately, the job is the same; to load (one or sometimes, many) files into memory where they belong so a program or the user can use them.

	.PGX	$F2, $56 auto-exec aka 'KUP'	Generic binary

Byte format	`Byte 0    signature: "PGX"`
	`Byte 3    $03          CPU type`
	`Byte 4    $00 $40      Destination addr.`
	`Byte 6    $-- $--      (unused)`
	`Byte 8-   $20 $21 $12  (data)`
		`Byte  0    signature: $F2, $56`
		`Byte  2    size of program in 8k blocks`
		`Byte  3    starting slot`
		`Bytes 4-5  start address (to be exec'd)`
		`Bytes 6-9  reserved`
		`Bytes 10-  zero-terminated program name`
		`Byte  x-   (data)`
			None - these binaries are unstructured, but common convention on 6502 platforms has been to align the desired start address to the reset vector ($fffc).
			In some cases, developers have created their own formats. We'll point to a few app specific use-cases below.

Description	PGX files (see pg. 3) contain identifying information and a load address and can start a program composed of a single segment. PGZ files are similar, but have provisions for multiple segments, loaded across memory (see links below).	Auto-exec KUP binaries, marked by $F2, $56 (aka F256), in bytes 0 and 1 are scanned for by the kernel at start-up. The MicroKernel code that manages startup is MMU 'slot' aware. KUP = kernel-user-program	Loosely, any file type may be called *binary*. More commonly, the name refers to non-ASCII data that is largely unprintable. A char set 'font' file is an example of one, as is machine code or sprite graphic data loaded by SuperBASIC, noted below.

How to load and execute	Upon load, data is placed in memory at byte offset $08 (the header is processed, but not stored to memory).
	When the application ends (with an `rts`), the program that initiated the load may either resume control, or reset the system.
	Here is the "from SuperBASIC" form, discussed on pg. 3:
	convention
	`/- {program}.PGX`
	mind the gap; a space is required
		At detection (during boot or reset), *x* number of blocks (byte 2), consecutive from the signature block, are moved to slot *y* (byte 3) and execution begins from the low-byte/high-byte address in bytes 4 & 5.
		Combined with the ability to flash blocks to cartridge or onboard flash (via the F256 Uploader) *and* use the MicroKernel DOS `lsf` command (see figure 4a), this capability is of great use to developers and users interested in customizing their system startup and onboard toolset.
			Absent identifying information indicating how or where to load or execute, the parent or calling program must load the data into memory at a prescribed address. Within SuperBASIC, this can be done as follows:
			`BLOAD "{filename}", {[$]addr}`
			… and if code, it can be executed using:
			`CALL {[$]addr}`
			There are several examples present in the *Foenix F256 Graphics Toolkit* by Ernesto Contreras (see link below).

.asm directives	`* = $4000`
	`.text   "PGX"`
	`.byte   $03`
	`.word   keys.cmd`
	`.word   0`
		`* = $4000`
		`.text   $f2,$56`
		`.byte   1`
		`.byte   2`
		`.word   keys.cmd`
		`.word   0`
		`.text   "sak - standalone keys",$0`
			`Not applicable*`
			*see pg. 22 for a look at how the 65C02 CPU (sans kernel) boots, the old fashioned way.

See also	https://wiki.c256foenix.com/index.php?title=Executable_binary_file	https://github.com/ghackwrench/F256_Jr_Kernel_DOS/blob/main/kernel/README.md (see the "**Startup**" section)	http://apps.emwhite.org/shared-files/770/?F256-GraphicToolkit-11.zip&download=1 See line 3500 onwards in the Foenix Sprite Editor "`spreditjr.bas`" for the BLOAD sprite load code. See line 3650 onwards in the Foenix Font Editor "`fontjr.bas`" charset load / relocate code.

https://github.com/FoenixRetro/Documentation/blob/main/f256/programming-file-formats.md

(applies to both)

A conversation with Boisy Pitre - CoCo historian and developer of NitrOS-9 for the FNX6809 equipped F256 platform

In issue #4 (December, 2022), we interviewed Gadget, developer of MicroKernel for the F256 and other efforts. The interview touched on her early experience with the Tandy Color Computer (a.k.a. CoCo), coding in Motorola 6809 assembly language, and its influence, which prepared her for a career of embedded software development and led to a collaboration with Stefany on the design of the F256 MMU.

It did not take long for Stefany to take this one step further when she created a 6809 compatible core on FPGA in a 40-pin package that when paired with a custom version of VICKY, gave Foenix developers the opportunity to write Motorola 6809 assembly language while enjoying all of the other F256 graphic, audio, and connectivity niceties. Issue #4 also had a 'sneak peek' of the FNX6809 (reprinted on pg. 14).

This month, we are closing the loop with a conversation with Boisy Pitre; author, developer, and primary contributor to the NitrOS-9 project. When Boisy caught wind of the availability of the FNX6809 option for the F256, he jumped on board and has been working tirelessly to port NitroOS-9 to the platform (Level 2 is ready for user test now!!).

This is the first 'real' operating system for a Foenix computer. Unix fans will recognize similarities between their beloved OS and the features that OS-9 brings to the table; CoCo and 6809 fans will likewise be attracted to the F256 platform for its speed and its modern features. There is alot to like...

EMWhite: Thank you for taking the time to discuss the platform and your contribution. I'm excited by what I saw in one of your demos and have been reading about the CoCo for some time, including in your book. Up until VCF Midwest this year, I had never met a CoCo or a CoCo enthusiast in person. I now have newfound respect for both.

If we could start at the beginning, what was your first job in tech; maybe as a teenager?

Boisy Pitre: My first job in tech was with Microware Systems Corporation, maker of OS-9. This was in 1992. The company started in 1977 and was past the 6809 phase of their business when I joined. At the time, they were heavily into the Motorola 68K and x86 platforms, and were looking at porting to the Motorola PowerPC and other processors. The MC68K version of OS-9 is still being marketed and sold by the current owners of Microware.

EMW: What was your first experience with computers in general; either at School or home?

BP: When I was 14, a friend of mine from church had an older brother that had the original CoCo and when the CoCo 2 came out, he handed it down to my friend. This was around 1983. Of course, we had an Atari VCS at home, but this was my first real computer experience.

EMW: Did your high school offer computer based curriculum or machines for use?

BP: My High School had a Commodore PET in the library that was not even functional. In my senior year, I took a Computer Applications course based on MS-DOS on "Leading Edge" brand PC clones.

But having a CoCo at home by that time (since 1985), I learned to program on my own. And in school, I joined FBLA (Future Business Leaders of America) and participated in a Computer Programming competition where I placed first in my region. Then I moved up to the State finals, and also placed first. I did not place in the Nationals, but it was a great experience and I knew at that point what I wanted to do professionally.

EMW: Did you take CompSci in College?

BP: I did, I earned my Bachelors and Masters degree in Computer Science and am currently working on completing my PhD. It will be the capstone of my career, once achieved.

EMW: Can you explain in layman's terms what NitrOS-9 is and how it came about?

BP: NitrOS-9 is a freely available version of the Radio Shack version of OS-9 which a group of folks deconstructed, fully commented, and made available in source. Members of the project then wrote cross-development tools on modern systems to assemble the code.

But the original NitrOS-9 got its start as a platform for the 6309 CPU (a souped up version of the 6809). It was developed by CoCo enthusiasts as a closed-source commercial product and sold in the early 1990s.

Sometime after 2000, the 6809 project began in similar fashion to the 6309 effort, by disassembling the original code.

EMW: How many people actively contribute?

BP: About 6-8 people are still involved and actively maintaining it. (here is the GitHub link) Do you have the FNX6809?

EMW: I do, though I haven't tried it, yet; I've been really busy with my stock 65C02 based Jr., which is where I do all of my development and Newsletter work. There is so much to do and I've got so much in motion, but the short answer is, I have three F256 machines and the silver cased Jr. (of course) will run NitrOS-9 at some point. My FNX6809 is still in the packaging Stefany shipped it in. I only took it out once, to show it at VCF East in April.

EMW: Earlier, you mentioned you taught yourself BASIC. What was your first experience programming; after BASIC, that is?

BP: I started learning the C Language in the '80s, then C++ in college. From there, I moved to Objective-C on the Apple Platform. The next logical step was Apple's Swift.

Editor's note: Boisy and I did not talk about it, but I'm aware of his *Swift for Beginners* book (see pic and link, below). Published in 2014 by Peachpit Press, the second edition is still available on Amazon and seems to be well received (4.3 / 5 on Goodreads with 27 ratings, and another 44 on Amazon's site).

Personally, the last development I did on Mac was when it was called a "Macintosh", and Lightspeed C was the language of choice. At the time, I remember the enigma known as the resource fork and how lost I was. In reading Boisy's book description here, I'm thinking a restart might be in order. Have a look.

EMW: As far as your day job is concerned, do you code professionally at the moment?

BP: I'm a professional writer and developer, putting together sample projects to teach other developers how to leverage different frameworks, so that's a "yes". But I continue to develop and contribute to personal projects such as the DriveWire server which runs on Unix, Mac, and Windows platforms.

EMW: What is *DriveWire*?

BP: In a nutshell, DriveWire lets NitrOS-9 platforms share disk, network resources, serial ports, and other devices with modern computers over a serial link and this now includes the F256 platform, when running NitrOS-9.

EMW: How fast is the 'last-mile' link to the Foenix machine?

BP: 230,400 bps.

EMW: I know you've written a few books as well; we'll get to them in a moment; but here's a tough question for you: Day job aside, how do you rank your interests, considering your time appears to be a split multiple ways (vintage historian, software engineer, freelance technology author)?

BP: I consider myself a software engineer first, a writer second, and a historian third. I've come to enjoy the historian role; I find it neglected in computer science.

EMW: And I either read somewhere, or heard you speak of your personal collection of vintage CoCo gear. What are the highlights?

BP: I have about 30 CoCo machines including a Deluxe CoCo which is an unreleased model. I also have a CoCo 4 mockup case, which is one of a kind, and the DIY kit that Tandy sold, which is based on the CoCo 2. Kits from those days required soldering of through-hole components. Finally, I've got the original four CoCo 3 prototypes that Microware used to develop OS-9; two are PAL and two are NTSC.

EMW: I might have to introduce you to Commodoreman (Chris) on the Discord forum. I think he now has some competition, from a vintage collection point of view!

EMW: Ok, here comes a few random questions: If you have a second hobby or passion, what is it?

BP: In addition to restoring old CoCo systems, I've had a few projects restoring commercial Namco Arcade games, specifically, PacMan and Super-PacMan. I like it as it combines a number of interests including electronics but also woodworking and general painting and restoration.

EMW: Do you play any musical instruments or have any hidden talents.

BP: I do. I play Cajun music, which is local to South Louisana and specifically, I play fiddle, accordion, and guitar. I play in jam sessions with friends all the time. Many of them are craftsman and make their own instruments.

EMW: Were you exposed to any interesting operating systems or technology early in your career that you found influential?

BP: I was first exposed to a port of Unix called ESIX on an Intel 386 platform in college. A professor had it running.

By the time I got to MicroWare, every developer had a Sun 3/80 on their desk running SunOS v3. I'd like to pick one of those up, someday.

EMW: Can you give me the NitrOS-9 elevator pitch; what are 3 killer features of the platform that qualify it as a real operating system.

BP: Well, it supports priority based, preemptive multitasking with aging through a fairly sophisticated scheduler. It adopts the Unix mode of unified I/O meaning it can treat devices and files similarly such that they feel the same to an application. Also, it supports kernel modules that can be loaded at boot or at run-time (via the command line) dynamically. This provides great flexibility such that you can have one build that is full featured with all of the modules resident, and another that is light weight, running only from memory with minimal drivers.

EMW: I should share that my personal favorite vintage system is my NeXT Cube for many of the same reasons. It has a real operating system; so I see the appeal and benefit. Despite the 68040 in my NeXT, it's dog slow. NitrOS-9, since it is character based is probably a different story.

EMW: What is left to do on the F256? Is Level 2 the end? What differentiates the Levels?

BP: Well, Level 3 was more of an experimental OS that somebody constructed. It utilized a different way of organizing memory, but it's not something I'm looking to tackle. Level 1 is restricted to 64K including the OS and all processes. Level 2 allows up to 64K for the operating system and then another 64K per process upwards of 2MB in total, which is plenty for the F256, since it has of 512K of RAM.

At the moment, I'm working on BASIC09, which is full featured and feels somewhat like what I've seen from SuperBASIC; they are both procedural. I'm not sure how far I'll get but I'm going for parity for features like DMA, which SuperBASIC front ends it with MEMCOPY.

There are still some questions to be resolved about the best way to distribute the environment and I've been talking to Stefany about whether a FLASH cartridge and an SD package would be a good turnkey arrangement. But it's ready for use now if anybody wants to grab it and build it.

EMW: How difficult was the port for the F256 platform, and how long did it take you?

BP: NitrOS-9 was designed to be portable, so the work was isolated. I needed code to talk to the hardware, specifically the MMU, RAM, FLASH, the SD interface, the real-time-clock, screen, keyboard, and serial port.

Stefany was very helpful in moving the memory map around with a custom FPGA build, and this was necessary for the Level 2 port, in terms of how the hardware was exposed.

From a code base size, the total of all sources and examples in the GitHub repo is staggering. I used a quick `xargs grep` to remove lines beginning with '*', ';', or '#' and the total returned was 670,279 lines. Mind you, this is for all nine platforms.

EMW: How much time did it take you to get to this point?

BP: Work started in September in earnest and within 2 months, Level 1 was running. Today, Level 2 is stable and live. You can download it now.

EMW: How much effort or expertise is involved in getting it built and running?

BP: A set of instructions has been pushed to the new Wiki (as of this week). Otherwise, all of the tools required to build the platform are maintained in the GitHub.

EMW: Amazing. I can't wait to get onboard. My next few months are booked but I'll look forward this and learning more about DriveWire as well.

EMW: I know you are still pushing, but looking back at your career, the projects you've participated in, and your own personal development, are you surprised by the trajectory of your career and achievements?

BP: When I was in high school, a teacher asked us to write down where we thought we would be in 10 or 20 years. I've been thinking about that.

If I could go back to my 17 year old self and look forward, I think I not only met, but exceeded what I had written down for that exercise.

EMW: I've had similar thoughts but #1 is feeling as if I was born at the right time in history, which is to say, at the advent of the microprocessor.

I also reflect back across my career and remember the first time I thought to myself: "I can't believe I'm getting paid for this".

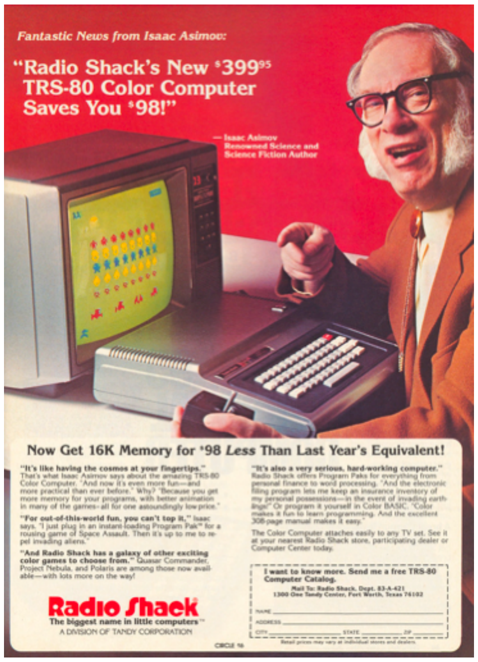
Boisy and I spoke some more and I didn't capture it all, but you get the idea. His work on our platform is significant and there is alot of history and technology to jump into. I'll have more on this topic in a few months. I can't thank Boisy enough for his time and effort.

In addition to Boisy's interest and experience with OS-9 and the CoCo platform in general, he also co-authored a book on the subject, available on Amazon and elsewhere.

"CoCo: The Colorful History of Tandy's Underdog Computer" is ~190 pages of dense information and considers technical details of the hardware, software, operating code intricacies, in addition to the genesis of the Tandy products, personalities involved, industry lore, clones, and more.

Any fan of vintage or retro will look at the index and want a copy. I bought mine following VCF Midwest, after mentioning Boisy's name to a CoCo enthusiast and learning of it.

One year ago this week, as published in Foenix Rising issue #4


Fantastic News from Isaac Asimov:
"Radio Shack's New $399.95 TRS-80 Color Computer Saves You $98!"
— Isaac Asimov Renowned Science and Science Fiction Author
Radio Shack
The biggest name in little computers™
A DIVISION OF TANDY CORPORATION

Now Get 16K Memory for $98 *Less* Than Last Year's Equivalent!

"It's like having the cosmos at your fingertips." That's what Isaac Asimov says about the amazing TRS-80 Color Computer. "And now it's even more fun—and more practical than ever before." Why? "Because you get more memory for your programs, with better animation in many of the games- all for one astoundingly low price."

"For out-of-this-world fun, you can't top it," Isaac says. "I just plug in an instant-loading Program Pak™ for a rousing game of Space Assault. Then it's up to me to repel invading aliens."

"And Radio Shack has a galaxy of other exciting color games to choose from." Quasar Commander, Project Nebula, and Polaris are among those now available—with lots more on the way!

"It's also a very serious, hard-working computer." Radio Shack offers Program Paks for everything from personal finance to word processing. "And the electronic filing program lets me keep an insurance inventory of my personal possessions—in the event of invading earthlings!" Or program it yourself in Color BASIC. "Color makes it fun to learn programming. And the excellent 308-page manual makes it easy."

The Color Computer attaches easily to any TV set. See it at your nearest Radio Shack store, participating dealer or Computer Center today.

I want to know more. Send me a free TRS-80 Computer Catalog.
Mail To: Radio Shack, Dept. 83-A-421
1300 One Tandy Center, Fort Worth, Texas 76102

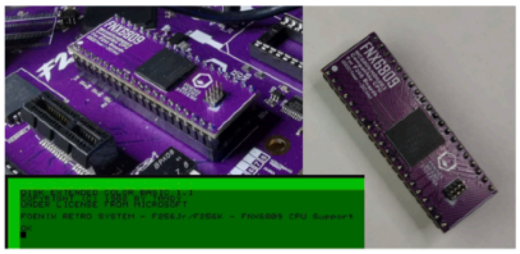
NAME ____
ADDRESS ____
CITY ____ STATE ____ ZIP ____

Retail prices may vary at individual stores and dealers.

CIRCLE 16

At the end of an interview with Gadget (on page 9 of Issue #4), I wrote a bit more about her involvement with the CoCo platform and took a tongue-in-cheek look at some retro adverts including this one with Isaac Asimov entitled "*Fantastic News from…*"

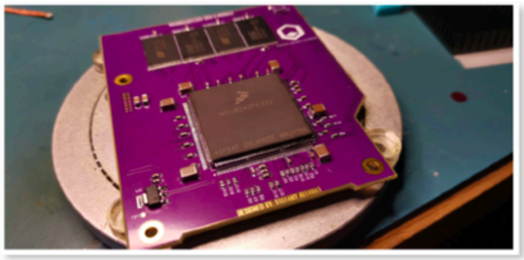
Spy photos from Foenix Labs



```
DISK EXTENDED COLOR BASIC 1.1
COPYRIGHT (C) 1982 BY TANDY
UNDER LICENSE FROM MICROSOFT

FOENIX RETRO SYSTEM - F256Jr/F256K - FNX6809 CPU Support
OK
```

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



MC68040 - 3.3V A2560X CPU module

On page 25, I had some spare room and asked Stefany for a few "spy photos" that I could fill some space with. She offered pics of her [then] prototype FNX6809. At the time, she had just coded the mock-up screen shot that you see in green.

Illustrating 8-bit dreams...

Article and accompanying SuperBASIC code written by Ernesto Contreras

Contributing developer Ernesto Contreras is the author of the original “Foenix Sprite Editor”, developed in BASIC816 for the Foenix C256U+ platform. He subsequently retooled and enhanced its capabilities, and ported the application to SuperBASIC on the F256 platform. Combined with his “Font” (aka character set) Editor and a full featured Tile Map Editor, the most recent versions of each have been packaged and released as the “Foenix F256 Graphics Toolkit”.

Ernesto’s work is unique because it is often graphical or musical in nature and tool focused, but also, because it is complete. Features are fully implemented, and apps often include documentation which acts as a user guide and a reference guide, documenting the byte format layout for data files his app creates.

Humble Beginnings in the ‘70s

Let’s go back to some the oldest home platform hardware created for 8-bit gaming, the ATARI 2600. By looking at its library of games you would believe that it had some half decent graphical capabilities, but even though a lot was accomplished with it, you should know that its hardware only supported displaying **five** interactive objects at any one time: two "player" sprites, two "missiles” and one "ball." (you can almost imagine how the original designers’ specs were inspired by Pong).

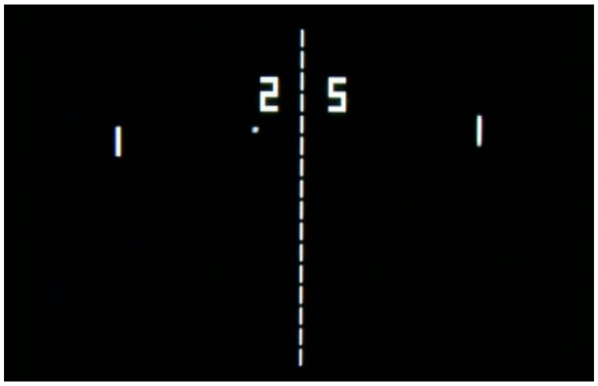


Figure 15a - **ATARI Pong**

So, the question is: how the ATARI programmers created half decent graphics with these limitations? Well… ATARI 2600 programmers were a sneaky bunch and implemented a lot of tricks and hacks, some of these were rudimentary “tile-like” capabilities with their limited hardware support.

To create these tile-style graphics with the hardware restrictions, the programmers designed their playfield carefully, and drew it using multiple instances of one of their available elements. This approach needed careful planning of the correct position of each element, either the missiles and/or ball. Once the electron beam had drawn it, the program could re-use this missile or ball element to draw it again, at a different position by shifting its horizontal or vertical position and redrawing it. Since the previous (missile or ball) image had already been drawn on the screen, the original one would not disappear until the electron gun came back around to redraw the screen. By doing this along with a lot of planning, and careful timing, programmers could create repetitive images that filled the screen.

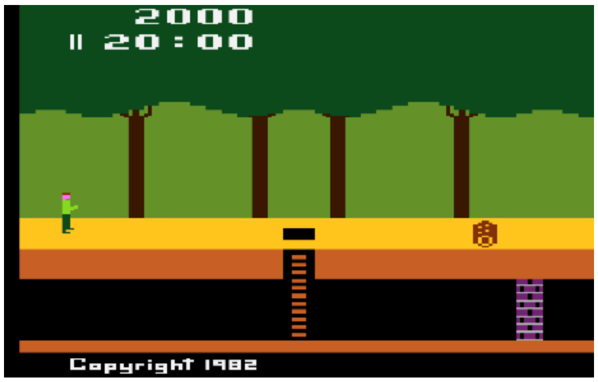


Figure 15b - **Activison’s Pitfall**

Use of these techniques allowed designers to create scenes on the VCS that were significantly more detailed than the hardware maker had ever imagined. Consider the jungle scene from Pitfall. The bottom part of the trees is drawn with missiles; they are used as primitive tiles creating the mirrored curves below the treetops. To complement the illusion, the details of the tree's branches are drawn using “static” sprites.

And if you are wondering, the iconic swinging vine was created with multiple “ball” elements, with its position shifted slightly on the x-axis as needed for each subsequent scan line to give the illusion of a swinging movement.

If you want to know more on how the ATARI 2600 accomplished some of its tricks you can read the book “Racing the Beam: The Atari Video Computer System.”

Moving Up in the ‘80s

Coin-ops by the mid ‘80s had left the graphics of their ‘70s counterparts in the dust. Image technologies had evolved rapidly, allowing more engaging gameplay, liberating games from the boundaries of one screen. Now the playfield was scrollable and ready to reveal more colorful detailed terrain in all directions, which blew our minds away!

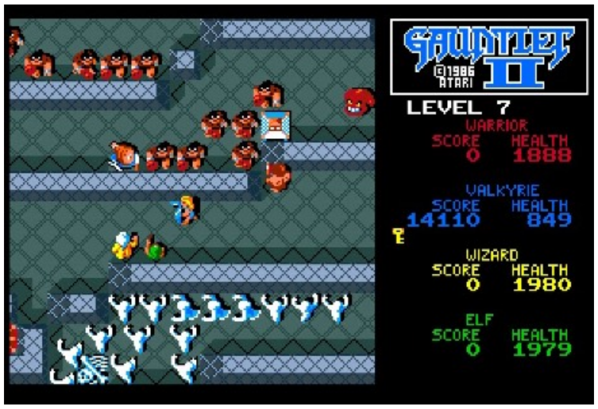


Figure 16a - **ATARI Gauntlet 2**

But how was all this graphic wizardry accomplished? How could games be so colorful, detailed and moving so fast with the limited resources of 8-bit machines of the era?

We were all aware of Bitmap Graphics since the 1970s, but moving bitmap graphics was prohibitively expensive in computational resources since it involved too many operations on too much data.

Well, the answer is simple. Hardware evolved using the '70s trick of reusing an element many times, tiles now were formerly supported in hardware as reusable graphic components that, along with a tile map, could direct how to arrange graphic objects onscreen.

Many of the Classic video games in the '80s benefitted from tiles, helping to push the envelope in graphical terms throughout the decade without taxing too much the CPUs of the era.

Figure 16b - **Capcom's Commando**

"Tile maps are a very popular technique in 2D game development, consisting of building the game world or level map out of small, regular-shaped images called tiles. This results in performance and memory usage gains — big image files containing entire level maps are not needed, as they are constructed by small images or image fragments multiple times."

Source: https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps

More Improvements: Multiple Tile Layers

By the mid '80s, additional tile layer support in hardware allowed game companies to unlock not only more detailed graphics, but additional effects.

For example, in *Rastan Saga*, two tile layers are used to create a parallax effect to simulate depth. A tile layer with the mountains and statues in the background moves at a slower pace than the front tile layer, which contains the playfield that the player interacts with.

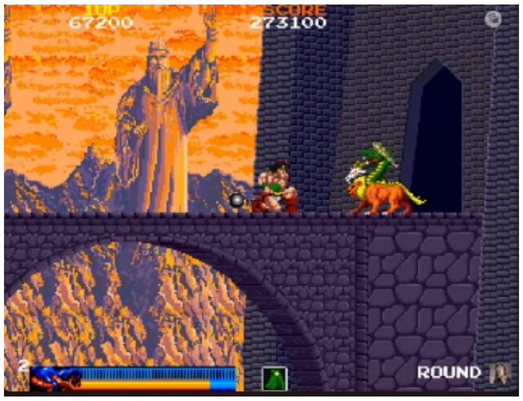


Figure 16b - **Rastan Saga**

A very nice 3D effect for the era, indeed!

Moving slightly ahead, by the middle of the '80s, improvements in CPU speed and addressable memory allowed more tile layers to be implemented.

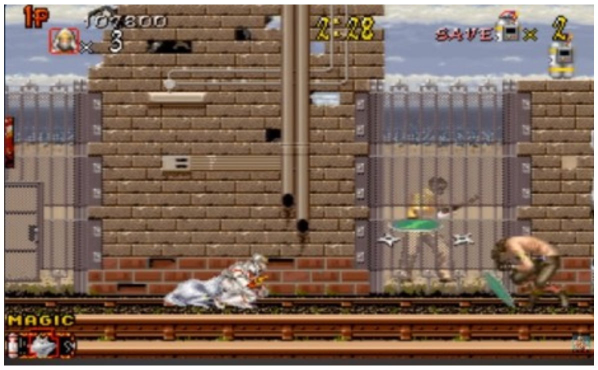


Figure 16c - **Shadow Dancer**

More Layers would not only improve visuals such as improving the 3D parallax effects with more tile layers moving at different speeds, but would also allow game designers to expand the actual game experience by permitting the player to handle the action in different planes, just as demonstrated in *Shinobi* or *Shadow Dancer* (above), where the player must alternate moving between the forward and back planes to attack all enemies and reach all goals needed to advance to the next stage.

Three tile layers are used for this effect; one with the background (again moving slowly) another in the middle (seen above as the fence that separates the back and forward planes), and a foreground tile layer for the forward plane.

Replicating The Graphical Identity of '80s video games on the Foenix F256 platform

After this brief tour on graphic capabilities of arcades in the '80s, we now have some basis to discuss the F256 graphic capabilities. Stephany ("The Mistress of all Villainy") surely has her own say about how or why she selected the features she included, but I believe that she wanted to replicate some period correct features.

Color	Four 256 Indexed color palettes Selectable per each tile in Tile Map or per bitmap plane, allowing up to 1024 colors on screen.
3 Graphical Layers	Up to three simultaneous stacked graphical layers Selectable between any combination of 3 tile layers or 2 bitmaps
Bitmap Planes	Up to 2 bitmap planes at either 320x240 at 60Hz or 320x200 at 70Hz Important: All other screen elements are adjusted to selected resolution/refresh rate
Tile Sets	Up to eight 256 Tile Sets, supporting either 8x8 tiles or 16x16 tiles Tile size configurable per each Tile Set/Tile Map
Tile Maps	Up to three Tile Maps Maximum size of each tile map is 256x256 tiles
Smooth Scrolling	Scrollable Tile Maps With Independent scrollable speed & direction per tile map
Sprites	Up to 64 sprites* (size selectable between 32x32, 24x24, 16x16 or 8x8) supported at once per scanline, configurable to be below or on top of any layer *Sprites are multiplexable so the total amount on screen can be easily doubled or tripled!
Text	1 Text Layer above all other layers 8x8 Font size with a few options for different number of screen rows/columns

F256 Graphic Capabilities
(Tiny Vicky – FPGA graphics engine)

* *Tiny Vicky has more capabilities, but we'll concentrate only on these, since they are relevant for this article*

This feature set would make it possible to implement, almost perfectly, the latest game example "Shadow Dancer", allowing 3 tile map planes with up to 1024 colors on screen (ok, the arcade has 4096 colors but 1024 is not bad!) with enough sprites configurable to allow the player and enemies to move between the back and forward plane along with bullets and ninja stars flying around.

A better comparison, more suitable for arcade enthusiasts, would be that the F256's graphics capabilities are comparable to that of the two SEGA platforms listed below (along with a list of the arcade hits of each platform, to provide more context). These examples cover the period from 1987-1992.

SEGA system 16B: *System 16 - Sega System 16B Hardware (Sega))*

VIDEO RESOLUTION : 320 X 224
COLORS : 4096
BOARD COMPOSITION : MOTHER BOARD + ROM BOARD
HARDWARE FEATURES : 128 SPRITES ON SCREEN AT ONE TIME, 2 TILE LAYERS, 1 TEXT LAYER, 1 SPRITE LAYER WITH HARDWARE SPRITE ZOOMING, TRANSLUCENT SHADOWS
EXAMPLE ARCADE HITS: ALIEN SYNDROME, ALTERED BEAST, E-SWAT, GOLDEN AXE, SHINOBI

SEGA system 18: *System 16 - Sega System 18 Hardware (Sega)*

VIDEO RESOLUTION : 320 X 224
COLORS : 4096
BOARD COMPOSITION : MAIN BOARD + ROM BOARD
HARDWARE FEATURES : 128 SPRITES ON SCREEN AT ONE TIME, 4 TILE LAYERS, 1 TEXT LAYER, 1 SPRITE LAYER WITH HARDWARE SPRITE ZOOMING, TRANSLUCENT SHADOWS
EXAMPLE ARCADE HITS: ALIEN STORM, MICHAEL JACKSON'S MOONWALKER, SHADOW DANCER

Hardware sprite 'zooming' and translucent shadows are not supported by Tiny Vicky (The core graphics engine of the F256) today. But I don't lose hope that some of these features will be supported on Vicky III, currently "in development" for the Foenix GENX, which, by the way, will support SEGA's choice of the *Motorola's 680XX* family CPUs used on System 16 / System 18.

Art on the F256

Before we start manipulating memory registers to enable graphic layers, color palettes, tile maps and scrolling, we need to address a small logistics problem:

The F256 doesn't have a full-fledged paint program to produce graphics (yet). And chances are, you have the graphics that you want to use to build your tile map on a PC or Mac. The challenge will be getting those graphics into the F256.

To address this problem, I'll discuss two matters in the following sections:

1. How to turn your source image into an 8-bit indexed Windows Bitmap with the help of GIMP
2. Decoding this bitmap file with a SuperBASIC program in the F256 and saving the image and palette as binary files

Preparing the Image

To turn the image into a suitable bitmap we are going to be using the program "Gimp", since it's a powerful Image Editor that works on Windows, Linux, and Mac platforms, and best of all, it's free!

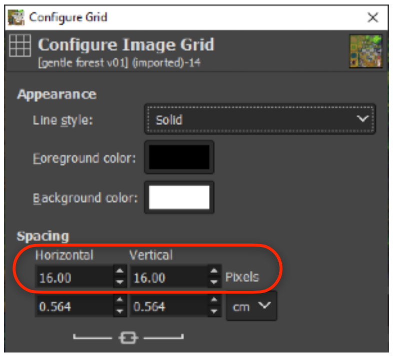
- Download Windows & Linux Gimp versions from GIMP - Downloads … or
- Download the MAC version from the developer's version GIMP - Development Downloads

Once Installed, launch Gimp and follow these instructions:

1. Open or Create a source image for your tiles in Gimp.

2. The obvious choice would be to make the image 256x256 pixels (16x16 tiles), but the Jr resolution is 320x240, so the *y* size won't be enough to fit in one screen!
3. An alternate size that would fit in one screen is 272x240 (17x15 tiles); you would need to create or rearrange your tiles in such a file.
4. This size is ideal for use with the *Tile Editor* from the *F256 Graphic Toolkit* since it allows up to 255 tiles out of 256. The Tile Editor always forces tile 0 as a blank tile and loads all tiles into memory from tile 1 to 255.
5. Optional Step - Configure Grid | Snap to Grid

   Select the following options from the Menu: Image | Configure Grid



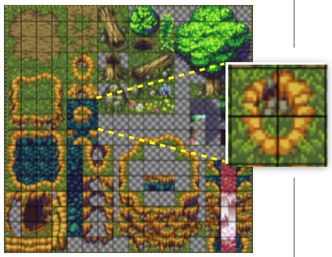
Configure Grid

Configure Image Grid
[gentle forest v01] (imported)-14

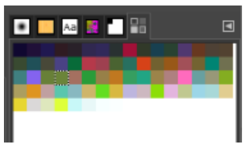
Appearance
Line style: Solid
Foreground color:
Background color:

Spacing
Horizontal 16.00 Vertical 16.00 Pixels
0.564 0.564 cm

   - On the Dialog that appears adjust grid to 16x16 or 8x8 pixels.
   - To show the Grid, select the following Menu Items: View | Show Grid
   - Having the grid helps if you are creating a tile set from scratch to dimension your images properly.
   - If you are rearranging your tiles in a different size image (as suggested on step 1), you'll benefit from enabling "Snap-to-Grid", this way Gimp ensures that when you are copying & pasting images they are always aligned with the Grid. Do this with the following menu items View | Snap-to-Grid.



6. Convert the Image to Indexed 8-bit color by using the following Menu Options: Image | Mode | Indexed.
   - On the dialog box select the following options:
   - Generate Optimum palette.
   - Maximum number of colors 255
7. Save the Indexed Bitmap file by using the following Menu Items: File | Export As…
   - Remove the current extension of the file and change it to `.bmp`
   - A message will appear informing that "Cannot export indexed image with transparency in BMP format – alpha channel will be ignored."
   - Don't mind the warning and click on "OK".
   - The file will be created correctly.
8. Optional Step - To view the indexed file palette generated, select the following Menu Options: Windows | Dockable Dialogs | Color Map



   - On the left side, the palette will appear
   - You can alt-click on any color and select "rearrange color map" to show all colors and rearrange them using drag-and-drop. Use this to make sure your transparent color is color #0

Understanding & Importing the Bitmap File

A bitmap file structure is simple enough to understand so that a simple program (even in BASIC) from an 8-bit computer can read it, and well, learning a bit more about bitmaps won't hurt!

With that settled, let's discuss the structure of the BMP file: First, a BMP file incorporates two headers:

A 14 byte BITMAPFILEHEADER that specifies the type of bitmap file, the size of the file, and the position (offset) in the file where the pixel data begins.

A second header, known in general as a DIB header, supplies technical information needed to render the image, such as bits-per-pixel, image height and width in pixels, and other exotic data such as compression method and halftoning algorithms (when applicable). There used to be different DIB headers for Windows and OS/2, but now only Windows remains.

The following 3 1/2 pages include file header formats and SuperBASIC code with commentary.

A full version of Ernesto's code (pg. 20) can be downloaded from the Foenix Marketplace at:

http://apps.emwhite.org/foenixmarketplace/

Bitmap File Header

Offset hex	Offset Dec	Field Size	Purpose
00	0	2 bytes	The header field used to identify the BMP and DIB file is 0x42 0x4D in hexadecimal, contains BM in ASCII when the bitmap is generated in Windows.
02	2	4 bytes	The size of the BMP file in bytes
06	6	2 bytes	Reserved - actual value depends on the application that creates the image, if created manually can be 0
08	8	2 bytes	Reserved - actual value depends on the application that creates the image, if created manually can be 0
0A	10	4 bytes	The offset, i.e., starting address, of the byte where the bitmap image begins (pixel array data)

Windows Bitmaps are nowadays the only BMP format used; OS/2 was the other OS using bitmaps but since it's no longer used, we will focus on the Windows version of the DIB header.

Windows DIB Header

Offset (hex)	Offset (dec)	Size (bytes)	Purpose
0E	14	4	the size of this header, in bytes
12	18	4	the bitmap width in pixels (signed integer)
16	22	4	the bitmap height in pixels (signed integer)
1A	26	2	the number of color planes (must be 1)
1C	28	2	the number of bits per pixel, which is the color depth of the image. Typical values are 1, 4, 8, 16, 24 and 32.
1E	30	4	the compression method being used. See the next table for a list of possible values
22	34	4	the image size. This is the size of the raw bitmap data; a dummy 0 can be given for BI_RGB bitmaps.
26	38	4	the horizontal resolution of the image. (pixel per metre, signed integer)
2A	42	4	the vertical resolution of the image. (pixel per metre, signed integer)
2E	46	4	the number of colors in the color palette, or 0 to default to 2^n
32	50	4	the number of important colors used, or 0 when every color is important; generally ignored

*Compression method parameters and halftoning will be omitted for simplicity since we won't be using compressed bitmaps and halftoning was only used for black and white images.

Believe it or not, the only relevant fields for decoding a Windows Indexed Bitmap file are the those shaded in grey in both tables, as we will demonstrate with the code in the next section.

Finally, Some Code (NEW)!

Good things come to those who wait, so here it is the code to import an indexed bitmap (up to 320x240 in size)

```
10    dword=alloc(4):pokew dword,0
20    cls :bitmap on :bitmap clear 0
30    input "bitmap filename (.bmp will be added):";a$
40    print "Loading bitmap..."
50    try bload a$+".bmp",$30000 to ec
60    if ec<>0 then cls :print "Error: File does not exist!":end
70    wpeek($3000E):hl=dwval:rem "DIB Header length"
75    wpeek($30012):xs=dwval:rem "X size"
80    wpeek($30016):ys=dwval:rem "Y size"
85    wpeek($3001C):bits=dwval:rem "Color Bits"
90    wpeek($3000A):rem "Offset to Pixel data start"
95    print "bitmap size x:";xs,"y:";ys,bits;" bit color"
100   orig=$30000+dwval:dest=$10000
110   for y=ys-1 downto 0
120   memcopy orig,xs to dest+(y*320)
130   orig=orig+xs
140   next
145   if option$<>"m"
150   ?1=1
160   for a=0 to 255
165   wpeek($3000E+hl+(a*4))
170   poke $D000+(a*4),peek(dword):poke $D001+(a*4),peek(dword+1)
175   poke $D002+(a*4),peek(dword+2)
180   next
185   bitmap off
190   ?1=1:for c=0 to 1023:?($7800+c)=?($D000+c):next :?1=0
195   b$=a$+".pal":print "."
197   print "saving palette.. please wait.."
200   try bsave b$,$7800,1024 to ec
210   if ec<>0
220   print "Palette Save error, check device or media":end
225   else
227   print "Palette saved as ";b$
230   endif
240   endif
245   b$=a$+".fbmp"
247   print "saving bitmap (320x240) please wait this takes a while..."
250   try bsave b$,$10000,320*240 to ec
260   if ec<>0
270   print "Bitmap Save error, check device or media":end
275   else
276   print "Bitmap saved as ";b$
280   endif
290   ?1=0
300   end
10100 proc wpeek(a)
10110 memcopy a,4 to dword
10120 dwval=peekw(dword)
10130 endproc
```

Let's analyze a few lines from the program:

Line 50 loads the image at $30000, we have a nice chunk of memory from $30000-$7FFFF which is not used in SuperBASIC (that's plenty of memory to load bitmaps!)

```
50    try bload a$+".bmp",$30000 to ec
```

In lines 70-90 we get the double word values at offsets $0E, $12, $16 and $1C which correspond to the shaded offsets in the WINDOWS DIB HEADER.

```
70    wpeek($3000E):hl=dwval:rem "DIB Header length"
75    wpeek($30012):xs=dwval:rem "X size"
80    wpeek($30016):ys=dwval:rem "Y size"
85    wpeek($3001C):bits=dwval:rem "Color Bits"
90    wpeek($3000A):rem "Offset to Pixel data start"
```

Finally on line 90 we get a double word (which corresponds to an address offset) at which the pixel data begins as marked on the shaded entry in the BITMAP FILE HEADER, this address is stored in variable `dwval`.

NOTE: `wpeek` is not a SuperBASIC keyword, but rather a **Procedure** that gets bytes from memory addresses outside the first 64k of memory

```
90    wpeek($3000A):rem "Offset to Pixel data start"
```

In lines 100-140 we copy the bitmap data to the BITMAP screen area in SuperBASIC ($10000) using the DMA via the memcopy function in SuperBASIC; you will notice that we go backwards starting by the bottom line and going up, since bitmaps are stored upside down in bitmap files.

```
100   orig=$30000+dwval:dest=$10000
110   for y=ys-1 downto 0
120       memcopy orig,xs to dest+(y*320)
130       orig=orig+xs
140   next
```

On Indexed bitmap files, the palette data resides after the DIB header data, so we use lines 150-180 to copy the palette data from the appropriate area (considering the DIB header length, since different programs produce a longer or shorter header!) into I/O page 1.

```
150   ?1=1
160   for a=0 to 255
165       wpeek($3000E+hl+(a*4))
170       poke $D000+(a*4),peek(dword):poke $D001+(a*4),peek(dword+1)
175       poke $D002+(a*4),peek(dword+2)
180   next
```

Lines 185 turns off the image (to help the user view text messages either confirming the operation or reporting errors)

```
185   bitmap off
```

Line 190 deserves its own explanation since it manually copies the values from I/O page 1 corresponding to the palette to an area ($7800) at the end of SuperBASIC reserved memory for programs and then BSAVES the palette from that area. This is done this way since you can't BSAVE directly from the memory in the I/O Pages.

```
190   ?1=1:for c=0 to 1023:?($7800+c)=?($D000+c):next :?1=0
```

Lines 195-230 save the palette file using the same name of the Bitmap file but adding a `.PAL` extension. Text messages appropriate to the success or failure of the operation are displayed.

```
195   b$=a$+".pal":print "."
197   print "saving palette.. please wait.."
200   try bsave b$,$7800,1024 to ec
210   if ec<>0
220       print "Palette Save error, check device or media":end
225   else
227       print "Palette saved as ";b$
230   endif
```

Lines 240-300 save the Pixel data of the Bitmap with a `.FBMP` extension.

```
245   b$=a$+".fbmp"
247   print "saving bitmap (320x240) please wait this takes a while..."
250   try bsave b$,$10000,320*240 to ec
260   if ec<>0
270       print "Bitmap Save error, check device or media":end
275   else
276       print "Bitmap saved as ";b$
280   endif
290   ?1=0
300   end
```

Regardless of the original bitmap size, the full screen 320x240 is saved so that the user can just use the following commands to load it from SuperBASIC:

```
bitmap on
bitmap clear 0
bload "FILENAME.FBMP", $10000
```

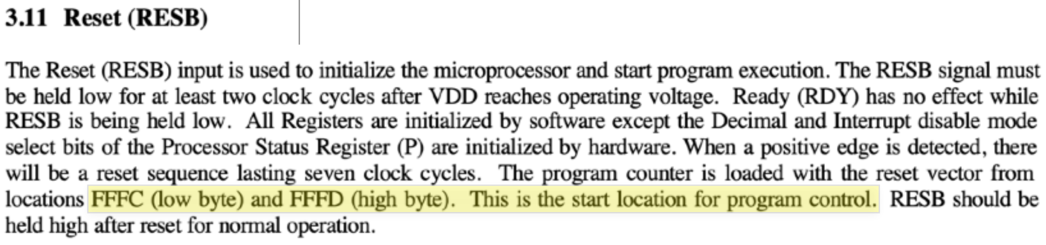
This article got a bit long, so we'll stop here. Next time, we'll address how to load the palette and slice this bitmap into tiles. Finally, we explain how to use the layering system, select tile layers, and play with tile maps to create a multilayer graphics scene worthy of a 1990 arcade.

F.R.

What happens when a 65C02 CPU is reset ?

Booting a 65C02 computer requires coordination between control lines, clock, and memory, but how does the CPU know what code to run?

A look at the WDC spec sheet unravels this mystery. The short answer is the low-byte/ high-byte address, accessible when address lines select `$FFFC` then `$FFFD`, gets to run.


3.11 Reset (RESB)

The Reset (RESB) input is used to initialize the microprocessor and start program execution. The RESB signal must be held low for at least two clock cycles after VDD reaches operating voltage. Ready (RDY) has no effect while RESB is being held low. All Registers are initialized by software except the Decimal and Interrupt disable mode select bits of the Processor Status Register (P) are initialized by hardware. When a positive edge is detected, there will be a reset sequence lasting seven clock cycles. The program counter is loaded with the reset vector from locations FFFC (low byte) and FFFD (high byte). This is the start location for program control. RESB should be held high after reset for normal operation.

On the F256, the story is a bit more complicated because the platform has a MMU primed from the VICKY FPGA which dictates which bank of memory aligns to slot 7 (`$e000`-`$ffff`). And depending on DIP switches (and if a Jr., the boot mode jumper), the system may boot from RAM via code that landed through the USB DEBUG port. Under normal conditions, FLASH contains MicroKernel which says, "I'll take it from here…".

The greatest YouTube video on the subject was produced by Ben Eater. You can see him exercise his breadboard CPU while 'sniffing' data and address lines. The entire video is ~27 minutes long, but click this link, to be taken to the punchline and within 3 minutes, it will all be clear.

nanoEdit 'data' mode preview

A viewer and editor for binary files. Use it for header manipulation or interrogation & machine language snooping

The tl;dr: nanoEdit (in development now), has a data mode that allows viewing and editing of unstructured binary files, useful for modifying data or creating binary files. Upon release, nanoEdit will join the growing list of F256 software, on a collective mission to support on-platform development.

Introduction

On the pages above, we discussed binary file types including KUP autostart files, PGX files, and others. In addition to the popular standards and known types, exists a category of binary files containing some semblance of structure, if not, embedded machine code.

Filetypes in this in-between category might include BMP derivative files (discussed in Ernesto's article), application specific files such as those produced by the Foenix Sprite Editor, and maybe even the *Lair of the Lich King* high score file: just because you are ranked *"3rd class dung wrangler"*, doesn't mean that you can't edit your way onto the LotLK list of "Top 10 Least Pathetic Souls"!

Fact is, absent a built-for-purpose editor for every data type imaginable, at some point it will be desirable to view and modify any file, regardless of format.

Back in '82

Forty-five (pushing fifty) years ago, machine language monitors served this function. They were one of a few go-to tools that could be used to edit binary files. To do so, the user had to load a file into memory at a given address, modify memory as needed, and then save the file to back to disk, identifying the starting and ending addresses in the process.

On page 4, we touched upon the lore of *Track & Sector Editors*. Their capabilities were impressive for the time, but unless you yearn for nostalgia, the technology is not relevant today.

Data mode of nanoEdit is closer to a machine language monitor in operation, but easier to navigate. If you must have your T+S Editor, there is nothing stopping an enterprising individual from writing a wedge that speaks directly to the FNX1591 and populates document memory of nanoEdit with sector buffers. I smell a hackathon project.

Use cases - Three to discuss:

1. File copy - **this** three minute long video sparked my interest in creating data mode. Despite looking at the SpeedScript source code for a few weeks (at that point), I failed to realize that one of Charles Brannon's features was somewhat unintended. His memory model and display routine could handle all 8-bit values (`$00` .. `$ff`) without fail. He did this, not for file copy or to remain open for other uses, but so he could assign ASCII values 128-255 for inline printer formatting tokens, or for output to disk.

   Since we had no desire to support printing, we merely changed the behavior for one special character (end-of-paragraph) and defeated two display formatting routines, and data mode was born. (see "data editing model" for more on this topic).

   The summary is, this use case is realized no differently than in the linked video, it just looks different: a file is opened, optionally edited, then saved.

   Having said this, nanoEdit offers enhancements. Our file save function allows writing to an alternate device (SpeedScript only supported [D]isk drive 8 or [T]ape). For an append use case (as demonstrated in the video), we can add a second file at the cursor position as well. But unlike SpeedScript's handling, nanoEdit will not disregard the remainder of the existing file, unless you direct it to; INS / OVR mode is obeyed, the latter of which adopts the default posture, else, files are inserted at the cursor position.
2. Data patch - let's say I had the need to replace text embedded within a binary structure with alternate values. Say… well, I don't know… grasping at straws here… oh… Let's say I load the file lk.pgZ and replace every occurrence of "Micah Bly" with "Michael W". Who's going to miss a few characters?

   In all seriousness, standard editing features work within data mode including search and replace for printable text. There are a few added features as well, such as the ability to enter a hex value at the cursor position or to hunt for an *n* byte long binary value (expressed in multi-byte hex). Handy.
3. 'Code' grab - the 'narrow' feature of data mode (ctrl-'n') combined with 'unassemble' (ctrl-'u'), invokes a split screen displaying assembly language in the right 20% (16 columns) of the 80 column screen. Grabbing (via ctrl-'g') will snag the displayed section of code to the kill buffer for recall or saving with or without append, and there is more to come.

On the next page, we'll discuss a few more details of data mode and provide a screen capture of the work in progress. There is alot of development required to deliver on these features, but we are off to a good start.

A font problem, and a fix

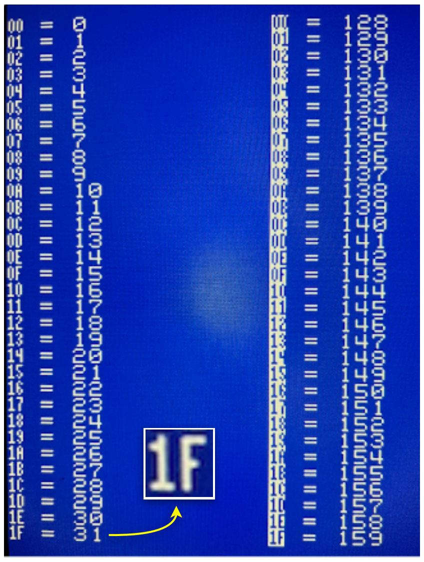
As discussed in issue #10 and #11, the nanoEdit project chose to adopt a Commodore-like reverse field scheme. This allowed us to preserve the core of the SpeedScript edit, store, and render code.

In doing so, we moved a number of line draw characters from the native F256 character set to the lower range of the set, intended for menus and dialogue boxes. In a prior version of nanoEdit they were in the `$14` - `$1f` range and included: . We also redefined characters to enhance the interface, originally `$07` - `$13` to: .

Cute… but in data mode, this complicates the desire to display all 256 byte values with glyphs. Many are aware that a tilde: '`~`' is ASCII 127 and '`A`' is 65, but few will remember what this is: . (spoiler… it is `$1d` in the nanoEdit character set, `$9a` in the F256 char set and `$b2` in PETSCII. So much for standards)!

It gets worse. How do you feel about this character: ? It's `$0e`, of course! Ridiculous.

By leveraging the seldom used F256 alternate character set feature (`FON_SET` @ bit 5 of `$d001`), a 2nd set of glyphs is defined representing ASCII value `$00` - `$1f` and `$80` - `$9f`. So instead of seeing the end of paragraph arrow: , we get a hexadecimal "`1f`" as called out below. This is better.



```
00 = 0      00 = 128
01 = 1      01 = 129
02 = 2      02 = 130
03 = 3      03 = 131
04 = 4      04 = 132
05 = 5      05 = 133
06 = 6      06 = 134
07 = 7      07 = 135
08 = 8      08 = 136
09 = 9      09 = 137
0A = 10     0A = 138
0B = 11     0B = 139
0C = 12     0C = 140
0D = 13     0D = 141
0E = 14     0E = 142
0F = 15     0F = 143
10 = 16     10 = 144
11 = 17     11 = 145
12 = 18     12 = 146
13 = 19     13 = 147
14 = 20     14 = 148
15 = 21     15 = 149
16 = 22     16 = 150
17 = 23     17 = 151
18 = 24     18 = 152
19 = 25     19 = 153
1A = 26     1A = 154
1B = 27     1B = 155
1C = 28     1C = 156
1D = 29     1D = 157
1E = 30     1E = 158
1F = 31     1F = 159
```

Figure 24a - 'data' mode chars `$00-$1f` and `$80-$9f`

Wait, not so fast. What occurs should we load a binary file into memory containing unprintable byte values and then try to edit the doc in 'text' mode?

For starters, the load routine scans input data and selects the appropriate mode (text, the default). But, the user is not restricted from switching.

The answer is, we display it with a token as a special character, otherwise impossible to type in text mode; the "diamond" character of Leonard Tramiel's famous Commodore card-suit set gets the job.

Data Editing Model

SpeedScript and therefore, nanoEdit's text mode, has two and only two rendering features built into the 'refresh' routine. Discussed above, the first invokes an end of paragraph action when character `$1f` is encountered. The second feature affects word-wrap by scanning backwards from column 40 to column 1 in search of a space (ASCII 32). If found, the line from that point forward is forced to the next line; if no space is found, the text occupies all 40 columns of that screen line; the routine continues in-kind on the following line until the full screen is rendered.

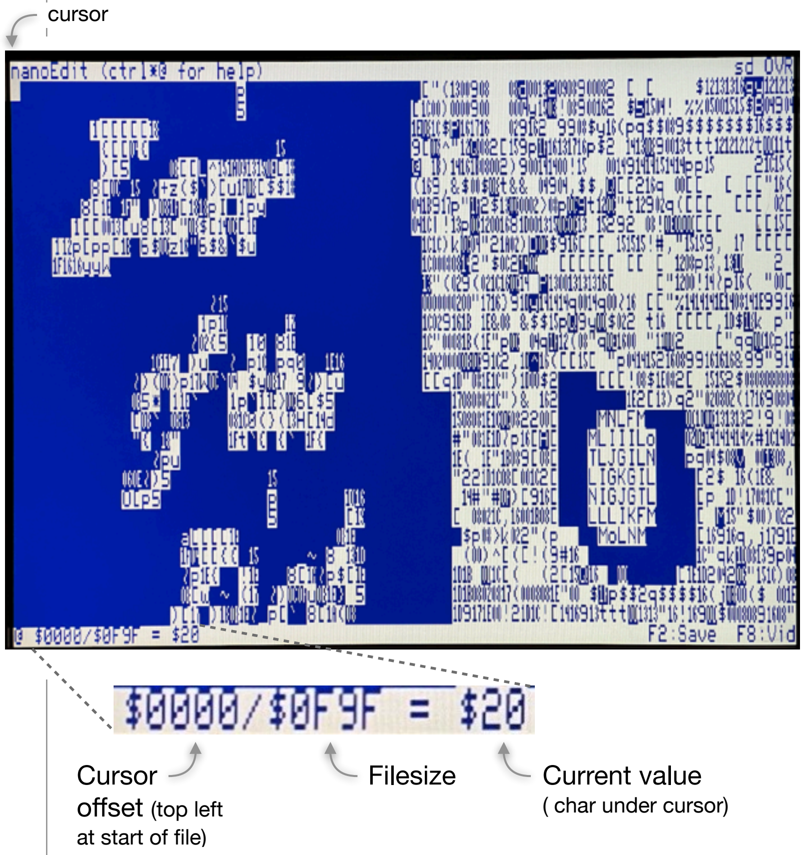
In our data mode, we simply defeat both of these features, utilizing every character of every line and as you'll see in the next section, this serves well in tracking offset addresses (especially in [n]arrow mode, which is 64 bytes of 80 wide).

Unlike SpeedScript, cursor movement is simple. Logic to seek the next paragraph and sentence is removed. Finally, data mode supports a few extended commands, which we mention below and will dive into next time.

On screen status

The following screen capture reflects the current build of data mode. You'll notice that rather than showing the cursor position in terms of column number and percentage through a file (the subject of issue #11's 16-bit division discussion), data mode status lines display the cursor position as an offset in bytes from the beginning of the file along with the file size (also in hex), and the character under the cursor.

The upper status line retains its job as in text mode, and is likewise used to prompt for functions only available to data mode; this includes [v]alue entry of hexadecimal bytes and the ability to search via the [h]ex hunt feature.



cursor
nanoEdit (ctrl+@ for help)
sd OVR
$0000/$0F9F = $20
F2:Save F8:Vid
Cursor offset (top left at start of file)
Filesize
Current value (char under cursor)

Next steps for the nanoEdit project

Next month, we will return to the FLASH format as we continue to make progress on the nanoEdit project. For now, have a look at this video (and potentially others) as working prototypes become ready for demo.

Nostalgia dialed up to 11: Commodore's Christmas Demo Rides Again

40 years of obsession has led to this… "the demo made me do it !!"

I admit it: I've been obsessed with the Commodore Christmas Demo since seeing it for the first time (1984) at a computer store in Selden, New York.

I was a College freshman at the time living in Berkeley, California and had returned home for the holiday. While on break, I wandered into the store (still trying to find the name of it) to see what was happening.

I'll cut the story short, except to say that a gift of sorts led me to obsess over the greatness of this demo for most of my adult life.

On second thought, I need to share … one of the guys that worked there looked the other way so I could make a copy. "… well, I really shouldn't, but it's Christmas so go ahead; don't let the owner catch you, I'll get in trouble…". Nearly 40 years after losing my copy, I bought a cartridge from an Etsy seller; in a way, the Universe has been after me to do this for a long time.

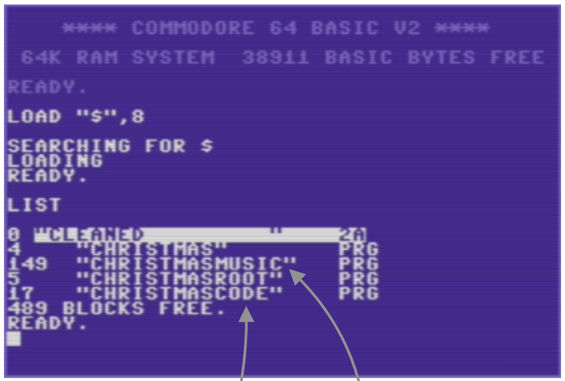
There are a few halfway-decent videos that mention this demo, but no deep dives into how it was created or what makes it tick. You can find my hackathon result on the Foenix Marketplace, run-able as a .PGX for the SID equipped F256 Jr. or any F256K. I'll place the assets into GitHub when I have time.

And when I really have time, I'll write a story about how I attacked the job and how I was able to get *part* of it working in 4 days. It was an unmitigated disaster, but I learned alot and had fun. Isn't that the point?

Merry Christmas
EMwhite (Michael)

"CHRISTMAS" is the file you load and run. It's a BASIC program that asks if you would like to run once (1), or continuously (0), and pokes that value into 40959.

It then loads the MUSIC file, CODE file, and main (BASIC) ROOT program and slams "RUN" into the keyboard buffer.


```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
LOAD "$",8

SEARCHING FOR $
LOADING
READY.

LIST

0 "CLEANED          " 2A
4    "CHRISTMAS"        PRG
149  "CHRISTMASMUSIC"   PRG
5    "CHRISTMASROOT"    PRG
17   "CHRISTMASCODE"    PRG
489 BLOCKS FREE.
READY.
```

"CHRISTMASROOT" coordinates the scenes. It is written in BASIC and uses a series of variables with SYS calls to paint each screen and control the action.

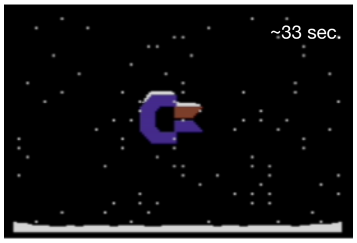
"CHRISTMASCODE" is primarily machine language but does contain data including the famous 'sales pitch' text. It loads from `$c000-$cfff`.

"CHRISTMASMUSIC" is the largest file, weighing in at 149 blocks or ~36K. It loads into memory from `$0d00 - $9fff`. The name is misleading since it contains code for two of the scenes and most of the graphic and character screen data.

Interpreting machine code is a lit fuse wearing on your patience and determination. One of my mentors once proclaimed: "*grit* dictates who succeeds and who quits. It's not intelligence or skill; it's the desire and determination to keep charging !!"

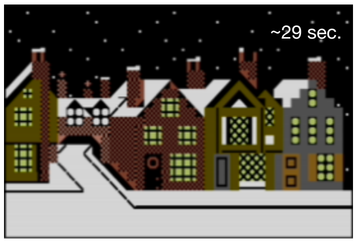
Intro - white noise

- Animated chars, smoothish scrolling, sprites (3 color CBM logo)
- Most graphically complex scene due to accumulating snow with 'smooth' scrolling. Would require the use of tiles on the F256.
- "Soundtrack", as it were, is filter swept noise w/modulation.



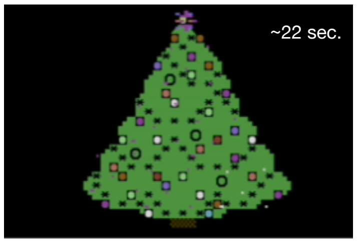
Village scene - Good Christian

- Fixed chars, animated char snow
- The snow animation uses some intelligence, and is shared across 3 scenes. All 3 have color artifact errors (especially Frosty).
- The soundtrack melody is multipart choral of similarly tuned oscillators.



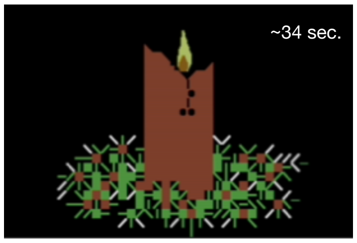
Tree scene - Jingle Bells

- Fixed chars, flashing sprites (2 color tree ornaments)
- Of course the soundtrack is iconic.
- Visually, the scene leverages a field of simple sprites which flash once per note phrase. The code is peculiar and the most simple; to be covered in a future article.



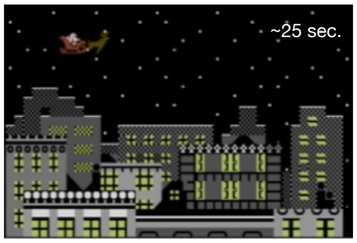
Candle scene - Silent Night

- Fixed chars, sprite animation (2 single color flame elements)
- Candle flame is composed of two sprites, 8 frames per
- The soundtrack sounds as if it's using heavily filtered single part simple chords. We will verify this in a feature on SID music in future.



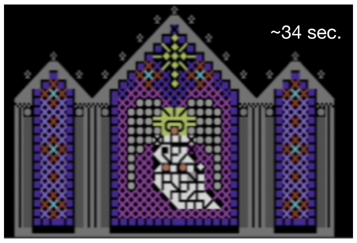
City scene - Rudolf

- Fixed chars, animated char snow, sprite movement and animation (4 colors including flashing nose)
- Rudolf, Santa, and his sled are composed of multiple sprites and is x and y expanded on the 2nd pass.
- Soundtrack is similar to Jingle Bells in Frosty in structure and tone.



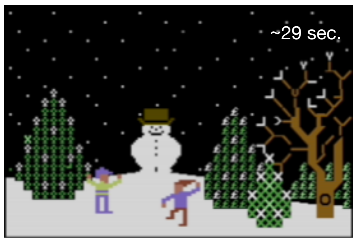
Angel scene - Hark

- Fixed chars
- The only scene with no sprites or character animation.
- This song is beautiful but could use more adventurous oscillator and filtering choices. Perhaps we will present a remix arrangement next year and lengthen the sequence.



Snowman scene - Frosty

- Fixed chars, animated char snow, sprite animation (6 colors)
- Leverages double-x, double-y expanded sprites
- Soundtrack utilizes an alternating 1-2 bass sequence and a simple, single-voice legato lead line, The character on the right appears to be winding up to throw a snowball !



Not depicted: The "sales pitch" scene, which boasts J. S. Bach's Invention #13, the most ambitious track of them all; aside from mid-play re-voicing (which became common as SID music evolved, this track exploits the SID's ring modulator in various forms. It is represented in FOENIXMAS23.