Foenix Rising

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

(FLASH !)

Nov. 2023 / F11

One of my closest friends claims to be a full-stack developer but **hates** front-end development. She loves data, software architecture, control planes, all of that back-endy stuff. I never understood her way of thinking; I just thought it was snobbery.

I'm here to report that I've come around to see her point of view. Even the most sparse 8-bit character based UI requires research, rewrite, refinement, and 'performance tuning' (note the quotes). In this issue, we unpack a number of old school interface features and code that hopefully helps you avoid my mistakes.

The simplicity of an 80 x 30 character based screen gone awry

Strengths and weaknesses of 8-bit microcomputers

Our machines are good at bit manipulation, branching on various states (via status register bits), incrementing memory, pushing and popping from the stack, and even keeping track of game score and extremely large decimal numbers (via BCD). And that's just the CPU. Around 1980, custom ICs empowered 8-bit systems to go further supporting sprites, blitters, MMUs, audio and more.

But our machines are challenged to count and perform math on large binary numbers, to move large amounts of data efficiently, and (due to a small set of registers) track several variables without continuously fetching and storing to memory, which tends to get expensive.

Great things are possible, but at some point, small systems run out of memory, processor cycles, or both.

Before I jumped into this project, I felt it would be easy to support the handful of character modes that the F256 offers; simply mask a VICKY master control register (`$d001`), then set or clear `DBL_X` and `DBL_Y` bits.

What I neglected to consider was the fact that changing modes breaks general screen navigation and scrolling. You can experience this yourself in SuperBASIC, discussed in this YouTube video. For nanoEdit, it also complicated resize and layout, so we had to compensate.

Citing one instance, status lines, a poor choice would be to keep all indicators in columns 1-40, leaving the right half of 80 character status lines empty. So we coded a lightweight method to recognize a token (' | ')to identify a split point, which right justified text as needed.

The SpeedScript interface was sparse out of necessity. In many ways, nanoEdit faces a similar list of constraints. Feature cost; memory, CPU, coding, and refactoring time; more than anticipated.

In issue #10*, I quoted Richard Mansfield, (the author of COMPUTE!'s *Machine Language for Beginners*) acknowledging that he grossly underestimated (by 4x) the amount of time required to write his all-assembly game. I'm not there yet, but I'm getting closer, and I don't like it.

In this issue, we'll dive into ~~four~~ [make that] five 'features' of the upcoming beta of nanoEdit and discuss the complexity of each and how demons were bested.

a. How to render a *QR code* along with text on the screen (used in the welcome message).

b. How to fake *alternate pitch text*, where standard characters were too wide for the allotted space.

c. How to organize code and data for single-use/*burn after reading*, thereby saving 2,952 bytes between initialization and normal operation.

d. How to efficiently *cycle through on-screen options*, using a simple list and 'poked' screen codes.

\+ e. How to *leverage the 'Math Block' 16-bit division* operator for a low-budget percentage calculation (instead of lengthy & expensive 'classic' assembly language). See pg. 9. snuck

Not my first rodeo

I'm not going to lie, I had a head start. 40 years ago, I composed my opus, an all-assembly language Terminal Emulator for the Commodore 64. It had a reverse engineered Punter file transfer protocol built in, a full featured disk and data menu, Hayes Smartmodem(tm) support, a CBM ASCII color & PETSCII graphics mode, and a **multi-file select interface** which, taken by itself, was close to the best piece of code I've ever assembled or compiled (because it was well over my pay grade and a miracle that I managed to get it working at all). for **Dennis**

Also, my terminal emulator had a 'frozen' status line above a scrolling 24 line display, embedded state indicators, and an overlay 'help' screen. nanoEdit includes these features specifically.

But there is plenty new here, made more complex by the work required to develop a ported Commodore kernal `CHROUT` routine.

Echoing the sentiment on the lead-in above, I hope some of this code and some of the techniques are valuable to you; if not, perhaps my foibles will be entertaining.

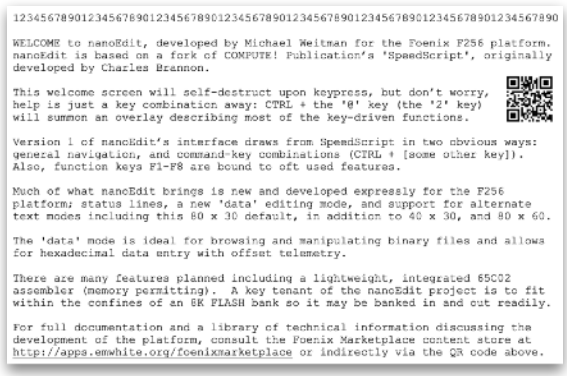
*Grab issue #10 here if you missed it. It contains an important introduction to SpeedScript and a look at COMPUTE Publications.
As an added bonus, you'll get to read a fun backstory behind a **Dennis** Hayes user feedback gripe (with Compute) and about his legendary product.

Getting started: (a.) QR codes are fun (or are they)?

Thank COVID for making the QR code ubiquitous. Every restaurant, at least in my part of the world, used them. It must have been a boon for the printing industry and for web hosting of online menus.

Today, there are free online QR generation sites such as this one, and python libraries such as this one.

But how does one go about getting a code into a Foenix machine for display on the screen as a 'see more' link.



Desired effect - prototype of the nanoEdit startup screen

The 'easy' way is to use a modern image manipulation tool and some python code off-host to produce a load-ready binary file or a sequence of ASCII `.byte` statements converted to source code.

The low-budget way is to do what I did. Reduce the encoded URL to the smallest feasible footprint (a 25 x 25 pixel QR type-no:119), model it on screen using the Foenix Sprite Editor to test readability with varied smartphones, choose a suitable color combination to improve clarity, then, armed with a picture of the QR code, recruit an unsuspecting child to read all of those 1's and 0's aloud while typing them into source code.

If they ask questions, it will open the door for you to give an hour long lecture on vintage compute. Winning!

But joking aside, it took about 20 minutes to type in and another 20 to fix mistakes. Next time: must. use. tools.

```
qr_sprdata  .byte  $01,$01,$01,$01,$01,$01,$01,$00,$01,$00,$01,$00,$01,$01,$00,$00
            .byte  $00,$00,$01,$01,$01,$01,$01,$01,$00,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$00,$00,$00,$00,$00,$01,$00,$01,$00,$01,$00,$01,$01,$00,$00
            .byte  $01,$00,$01,$00,$00,$00,$00,$00,$01,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$00,$01,$01,$01,$00,$01,$00,$01,$00,$01,$01,$00,$01,$00,$01
            .byte  $01,$00,$01,$00,$01,$01,$01,$00,$01,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$00,$01,$01,$01,$00,$01,$00,$00,$01,$00,$01,$01,$01,$00,$01
            .byte  $01,$00,$01,$00,$01,$01,$01,$00,$01,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$00,$01,$01,$01,$00,$01,$00,$01,$01,$01,$01,$00,$01,$00,$00
            .byte  $01,$00,$01,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$00,$00,$00,$00,$00,$01,$00,$00,$01,$01,$01,$01,$00,$01,$00
            .byte  $01,$00,$01,$00,$00,$00,$00,$00,$01,$00,$00,$00,$00,$00,$00,$00
            .byte  $01,$01,$01,$01,$01,$01,$01,$00,$01,$00,$01,$00,$01,$00,$01,$00
            .byte  $01,$00,$01,$01,$01,$01,$01,$01,$01,$00,$00,$00,$00,$00,$00,$00
            .byte  $00,$00,$00,$00,$00,$00,$00,$00,$00,$01,$00,$00,$01,$01,$00,$00
            .byte. $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
```

Resulting data (this is a quarter of it; 256 bytes of the 1K):

Nothing is easy

In a prior article, we mixed text and sprites in an RTC 'am/pm' clock tutorial (issue 6). It worked out well and the result was pleasing. What I didn't realize at the time was how uninformed I was in dealing with layers and specifically, the fact that while sprites may be mixed in and through layers of tiles and bitmap graphics, text is *always* on top. I was successful using layers in issue 5, when I bounced bitmapped rendered objects behind sprite numerals, but suddenly, I was having trouble.

This was a different challenge: When the character background color is anything but palette color `0` (transparent), none of the graphic objects behind the text are visible and this doesn't jibe with our desire to let a `CHROUT`-style output routine obey color change and reverse field attributes alongside a sprite use case.

Recall, the F256 does not have a screen color in the traditional sense; there is no `$d021` (Commodore 64) background color. Instead, VICKY supports varying screen colors *per* character, if desired.

I chased this around in circles for 2 evenings until I recalled a conversation that Gadget, Stefany and I had about a year ago. At the time, Micah (we sent a car to find him) was crafting all of those lovely glyphs in the F256 character set.

I was proposing a reverse field complement to the F256 set and Gadget insisted that the need could be satisfied by manipulating foreground/background colors and she was right; but the scheme produces an unwanted side effect. With a colored background (which is opaque), characters always block the sprite, bitmap, and tile layers from being seen.

The fix; four choices:

a. Enable a **bitmap screen** and fill it with bits strapped to a LUT of the desired background color, then ensure that none of the characters have a background color (their background LUT is all color `0`). We will still be able to provide the important [to the core SpeedScript code] reverse field characters because we created them within the default set.

b. Do likewise, except with a single solid **tile and a tile map** across the screen. This will not require nearly as much memory as the 64,000 byte bitmap.

c. **Punch holes** in the character screen (the spaces directly in front of the QR code pictured) and choose 'off' bits that match the background color. We'll need to precisely align sprite x/y coordinates of the 32 x 32 pixel sprite (to that of the characters).

d. Use 15 **redefined chars*** instead, and juxtapose the redefined chars to form the desired QR image.

Option 'a' and 'b' are expensive and require code and data to setup and option 'd' is silly (but might be a fun science project for a kids hackathon); So option 'c' it is.

The good news is we are only dealing with this QR code business during the first run of the program. There is no need to fuss about color parity between LUTs because this environment is controlled and follows initialization.

* Why 15 characters and not 16? Because we can re-use the vertical bar in the left pair of 'finder pattern' blocks if we right justify the graphic. "*... because every byte and cycle counts*"

Now is a good time to point out that on the F256, char color LUTs (`$D800` of `MMU_IO_CTRL` bank `0`) are different from the general graphics color LUTs (which reside starting from `$D000` of `MMU_IO_CTRL` bank `1`).

I'm mentioning this because coordination of text and graphic assets will require duplicate entries. Here is the manual excerpt for each:

Note: addresses are in I/O bank 0

Index	R/W	Foreground	Background	0	1	2	3
0	W	0xD800	0xD840	BLUE_0	GREEN_0	RED_0	X
1	W	0xD804	0xD844	BLUE_1	GREEN_1	RED_1	X
2	W	0xD808	0xD848	BLUE_2	GREEN_2	RED_2	X
3	W	0xD80C	0xD84C	BLUE_3	GREEN_3	RED_3	X
4	W	0xD810	0xD850	BLUE_4	GREEN_4	RED_4	X
5	W	0xD814	0xD854	BLUE_5	GREEN_5	RED_5	X
6	W	0xD818	0xD858	BLUE_6	GREEN_6	RED_6	X
7	W	0xD81C	0xD85C	BLUE_7	GREEN_7	RED_7	X
8	W	0xD820	0xD860	BLUE_8	GREEN_8	RED_8	X
9	W	0xD824	0xD864	BLUE_9	GREEN_9	RED_9	X
10	W	0xD828	0xD868	BLUE_10	GREEN_10	RED_10	X
11	W	0xD82C	0xD86C	BLUE_11	GREEN_11	RED_11	X
12	W	0xD830	0xD870	BLUE_12	GREEN_12	RED_12	X
13	W	0xD834	0xD874	BLUE_13	GREEN_13	RED_13	X
14	W	0xD838	0xD878	BLUE_14	GREEN_14	RED_14	X
15	W	0xD83C	0xD87C	BLUE_15	GREEN_15	RED_15	X

(above) Text LUTs (note only 16 colors for foreground/background)

Address	R/W	Purpose
0xD000	R/W	Graphics CLUT 0
0xD400	R/W	Graphics CLUT 1
0xD800	R/W	Graphics CLUT 2
0xDC00	R/W	Graphics CLUT 3

(left) Graphic LUTs (note: there are four available, each with 256 slots of 4 bytes each)

Note: these addresses are in I/O bank 1

Here is the sprite graphic data load code; we discussed a similar routine in prior issues:

```
sprinit         lda     #<qr_sprdata
                sta     FROML
                lda     #>qr_sprdata
                sta     FROMH
                lda     $80
                sta     DESTH
                stz     DESTL
                ldy     #$00
sprloop         lda     (FROML),y
                sta     (DESTL),y
                iny
                bne     sprloop
                inc     FROMH
                inc     DESTH
                lda     DESTH
                cmp     #$04
                bne     sprloop
```

Here the sprite meta-data instantiation code; we've used this routine in the past as well:

```
                ldx     #$00
sprattr_outer   ldy     #$00
sprattr_inner   lda     qr_sprattr,x
                sta     $d900,x
                inx
                iny
                cpy     #$08
                bne     sprattr_inner
                lda     qr_sprattr,x
                cmp     #$ff
                bne     sprattr_outer
```

That's all there is to it. We define the sprite, load it into memory and instantiate it using a standard method.

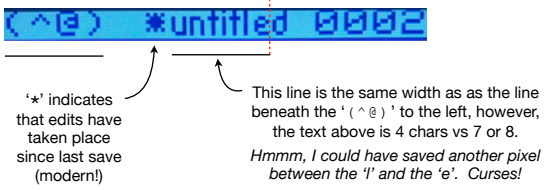
(b.) Faking condensed text

Monospaced fonts are great and all, but it's a bit of a stretch to use the adjective (monospaced) and noun (font) as if we had a choice; all text on late '70s and early '80s machines was monospaced. And they did not have fonts, they had *character sets*. Eventually, the Apple II and others ran Word Processors with limited fonts for output, and some had on-screen rendering, but not without heavy lifting which consumed resources.

nanoEdit is a text editor, not a Word Processor, but it supports an 80 character screen by default. Press F8 and you can instantly pop into a world of 40 x 30 text. There are many cases where this is preferred. Offering both complicates the desire to squeeze static text and dynamic messages into one or more status lines.

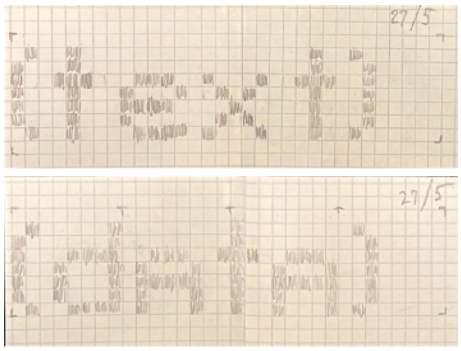
Since the birth of the CRT, the goal has been to present the user with a clean and intuitive interface; complete (in this case) with useful data on status lines.

The text below on the left is a string of sorts; four 'normal' characters followed by a space and an asterisk (noting that the file has not yet been saved), followed by the word "untitled". Look closely and you can see that the text is a bit odd; you might even suggest that it is proportionally spaced.



To prepare, we used Ernesto's Foenix Character Editor to create the 'untitled' text, and ultimately, to squeeze 6 characters with whitespace at each end into less than 4 and one half characters of 'on' bits (37 pixels in total).

We could have done this with bitmap graphics or tiles on a color-matched status line (per item 'a.' above) but text mode, while mono-chrome per character, is 2x the resolution (640 x 480).


Hand drawn prototypes

The approach used is low budget, but it gets the job done and once the data is loaded into the set, we can dispose it, and the load code (see item 'c.' below).

But it's not all peaches and gravy. This approach contradicts a statement made on page 5 of Issue #10 where I boasted that we would not alter other glyphs or try to recreate the Commodore PETSCII char set.

We will still honor the latter part of this agreement, but a hard violation of characters numbered 6 to 31 was

required; first to create our newline character (ASCII 31, discussed last issue), but also to move and redefine a number of line draw characters for dialogue box like piece parts (around the help screen); to create an edged vertical bar (where a centered separator felt too tight), and above for the 'text' and 'data' editing mode labels; the gratuitous 'untitled' label was just for fun; we allotted 16 characters for a file name and easily could have written this out in standard ASCII text.

At the bottom of this page, you can see a depiction of our full character set. Init code does the dirty work and a bank of high RAM saves the original set(s) for restoration upon exit (an alternate character set and the `FON_SET` feature will be used for *data* editing mode, to be covered in the next issue).

How to place re-defined characters on the screen

Placing the new characters on-screen is as simple as storing the ASCII value into screen memory at `$C000` when `MEM_IO_CTRL` is equal to `#$02`. Our normal screen output routine will not readily print these (since non-printable codes are interpreted as cursor and color controls) but we created a `rawout` vector, also to be discussed in future. Here is the simple math:

```
$c000 + (row * scrnwidth) + column
```

nanoEdit uses F6 to toggle the edit mode and the following code swaps the status line chars as needed (for simplicity, this is a 40 column only example).

```
modetxtdat      .byte $0b,$0c,$0d,$0e,$07,$08,$09,$0a

bumpmode        lda     mode
                eor     #$01
                sta     mode

updstat40       asl     a
                asl     a
                tay
                lda     modetxtdat,y
                sta     $c49c
                lda     modetxtdat,y+1
                sta     $c49d
                lda     modetxtdat,y+2
                sta     $c493
                lda     modetxtdat,y+3
                sta     $c494
                rts
```

ASCII values

This code takes the mode (0 or 1), shifts left twice (mult. by 4), then transfers the product to the `y`-register (offset into `modtxtdat`). Simple.

For illustration; if you were trying to impress your friends, you might try a label def instead of a series of hard addresses, or better yet, do some math based on row and column. But this works as well!

'Creating' our own reverse field chars, revisited

This code is similar to last month's example, however, is one instruction shorter. The reason for the revisit is to discuss convention. As you will see, it is somewhat unconventional and/or can be considered incomplete, unless you've played with this type of code, prior.

Execution absent the pre-req (see 'a.' below) will either locked up your machine, or produce a screen full of garbage or interesting colors.

Three matters to discuss on this topic:

a. the prerequisite - `MEM_IO_CTRL` (`$01`) must be set to `#$01`; this exposes FONT memory to the CPU. Often, this bank will default to `#$00` (I/O addresses) and depending on the type of app, screen update intensive code might default to `#$02` or `#$03` for ready access to the character or color matrix.

b. this code is *self-modifying* (versus using zero-page indirect indexed addressing) - in the example, we begin with two 16-bit addresses (source and target for our font byte copy) initialized `$bf00` and `$c300` respectively. These addresses are useless until the first increment (3rd and 4th instruction). The routine loops 'outer' 4 times and 'inner' 256 times, tallying 1,024 bytes of movement, indexed by y.

   The fundamental difference from the original code (issue #10, pg. 5) is we count **down** from 3 to 0; versus counting up); this allows us to remove the `cmp #$04` and change the branch from a not-equal (`bne`) to a branch-positive (`bpl`); We also move the `ldy #$00` to the top. The code iterates just the same, but is one instruction and 16 cycles shorter. Will the two bytes or 16 cycles matter? No :)

c. Importantly, this code is 'single use' - since it modifies itself, it can only run once. We could initialize it at the start, but if we wasted instructions doing that, we could have just as well used zero page indirect indexed. As you'll read below, this code will be disposed of, so it won't matter.

```
fontrvs         ldx #$03
                ldy #$00
fontouter       inc fontinner+2
                inc fontupper+2
fontinner       lda $bf00,y
                eor #$ff
fontupper       sta $c300,y
                iny
                bne fontinner
                dex
                bpl fontouter
                rts
```

The most important thing to know about the F256 character sets is they require no additional memory once defined. And VICKY supports two full 2K sets built into one of the $c000 banks. For more info on this topic (in a SuperBASIC context), see the following **YouTube video** and the two that follow.

The nanoEdit character set (for now) - appears mirrored through character 128, but ASCII `0` is missing (the British Pound sign is not included above); this image was generated with the SuperBASIC `CPRINT` command which stubbornly refuses to print a 'null'

We opted to stick with this Commodore'esque reverse field scheme so the core code could run as is. It also burns far fewer machine cycles to XOR bit-7 of the char under the cursor than to shift color attributes to reverse foreground and background repeatedly.

As you'll see next time, we use a different character set in 'data' editing mode; in the 2nd set, the ASCII chars appear in their rightful place, but we replaced lower order glyphs with control symbols to remove the guesswork from visualizing byte values while editing binary files.

(c.) Burn after reading: a memory use discussion

nanoEdit could be thought of as a successor to SpeedScript for the simple matter that it contains a core of the original code. It also aims to meet several of the same design objectives that were likely being discussed in Compute's Raleigh, North Carolina headquarters:

1. speed (its in the name) - response time must not feel sluggish. This was challenging back in '82 and it still is. In our next issue we will touch up the text store methodology, its limits, and how we improved it).
2. functionality - the more utility, the better, provided it does not get in the way of performance or contribute to bloat. (while Brannon & co. sought to fulfill word processing needs, nanoEdit is intended to be a general purpose editor that will eventually grow into an on-platform IDE). There is much left to do.
3. efficiency - it must fit in a small footprint, leaving more memory for documents and data.

   In addition (and just as SpeedScript did with Commodore's R6510 location $01), nanoEdit takes advantage of machine specific features such as memory banking, the use of flash, and soon, DMA.

Ultimately, the project aims to encourage a wide group of users to make nanoEdit part of their workflow and to to co-exist with SuperBASIC.

nanoEdit also differs since it is built to execute from a bank of flash memory, and since it stores its non-zero page variable data in a clean RAM bank acquired at runtime. Above all, the #1 design goal is to keep the code within a single 8K bank of memory (SpeedScript was ~6K).

Thankfully, the F256 is ~6x more performant by clock speed than the original SpeedScript target platform (the Commodore 64). But more features means more code and we are still bound to a 64K address space.

To deliver on our goals, we will leverage a simple compression algorithm and also, throw away a fair amount of the startup data including the welcome message, font sources, sprite data for the QR code, and various bits of load and copy code. This approach is less lofty, but not much different than a rocket launch that leverages a booster stage to reach a certain altitude, and then sheds aeroshells and other components which are no longer required for space flight.

We'll also do as SpeedScript did and leverage a startup signature and code that sets, and (upon subsequent execution) recognizes that initialization has already run.

SpeedScript did this to allow re-entry, meaning that if you quit the program and returned back to BASIC, you could return and re-start where you left off.

Because Charles Brannon seemed like a swell guy, we will leverage his method in the form of the same byte signature (`$CB`, his initials in hex) stored in a variable called `FIRSTRUN`). We will use it as well.

We will also do as SpeedScript did and leverage a label positioned just beyond the 'keep' code which is important to the editor, but prior to all of the disposable assets (code and data).

Here is how SpeedScript went about it:

SpeedScript used `END` to identify the start of memory which followed its last variable; we used '`CODEEND`'.

```
LDA   #>END
CLC
ADC   #1
STA   TEXSTART+1
LDA   #$CF
STA   TEXEND+1
```

Brannon's scheme grabs the high byte of `END` adding one 6502 'page' to mark a 'clean' starting point for text memory. His text end high byte is `$CF` which is the last page of Commodore 64 high RAM.

Finally, we will unpack from flash into RAM in our release version; this step is necessary because our flash footprint is smaller than the sum of all of our code and data. RAM is acquired from the extended pool, above high memory at `01:0000`, one MMU bank at a time.

Longer term, we will leverage the decompression algorithm discussed below and will look to support overlays (for an assembler, as one example). For now, beta versions of nanoEdit will be distributed as monolithic `.bin` files with accompanying Commodore-linked ABI print and load/save routines. This is a journey, not a sprint.

Otherwise, there is nothing else left to do. Part of the SpeedScript initialization code (which is also used by the erase text buffer command) will literally stomp all over the code and data that exists beyond `CODEEND`.

For your records, here is a tally of data and code [so far] that we will leave behind (2,952 bytes *unpacked*):

`welcometext` : text (1,450) : displayed once
`spritedata` : graphic data (1,032) : same
`sprinit` : code (84) : same
`fontinit` : code (237) : run-once copy and move
`returnchar` : glyph data for above (128)
`fontsources` / `fonttargets` : table data (21)

A quick word on pack/unpack

A modest compression routine is in the works, derived from a paper written by *Tommy et al* in 2018, published in the *Journal of Physics - Conference Series*.

The algorithm uses a value differencing scheme which we will adapt to leverage 64tass `.lst` files. More on this next time. But if you are curious, have a look at the pseudocode description on page 3 here, which we will code simply in 65C02 assembly. The packing routine is being written in Python to leverage the list output.

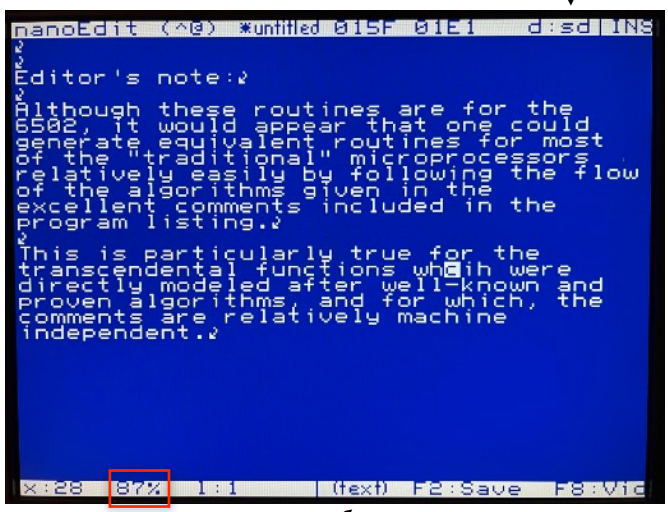
(d.) Cycling text using data and poked screen codes

In a humble brag on page one, I mentioned my experience managing status lines in a program I wrote on my childhood computer, a Commodore 64. Looking at my code today (which I have in MAE source format), I am impressed and embarrassed, simultaneously.

I recently explained to Stefany that my 6502 coding skills are equivalent to that of an undisciplined 17 year old because I literally picked it up again about a year or so ago, after not touching it for many years.

But a long career in Tech has taught me a few things. So I am here to share an evolved form of that old code, updated to be presentable as used in three features (see callouts below).

(1) Typing mode: "**INS**" for insert; "**OVR**" for overwrite; *toggles*
(2) Device: "**sd**" or an IEC drive number ["**08**".. "**11**"]; *cycles*



(3) Editor mode: Custom char "**text**" or "**data**";

(Figure 1)

Of course, there are differences between each of the three features, but they are handled similarly. Each has state-change code, screen-poke code, and follows convention for efficient 'use' code.

State-change code: typically a stub or a short piece of code responsible for toggling a byte or advancing an index that tracks a setting. It is a good practice to optimize code flow (see *use*). For the sake of discussion, a state change is an update which is typically (but not always) triggered by a user keypress.

Screen-poke update code can range from simple to complex and depending on the type of feature and importance of timing to an event loop, there will be occasions to leverage pseudo-timers or interrupts which defer or batch updates as we do in major item (e.) below.

Example (1) in this part of the discussion merely updates 3 characters ('OVR' or 'INS'). This code takes no time at all to execute and is encapsulated in the keyboard control loop (meaning, it will not be executed until actuated).

The opposite of this type of feature is code that might perform an expensive calculation or loop through memory. In such cases, it's best to defer until the keyboard is idle or at worst, update every half second or every few seconds. The "87%" highlighted in red is an example of this, as you will see.

Use code, meaning the part of the program that will read a value and act upon it.

Generally speaking, if the 'test' is in a loop where performance is important, the value used should rely on a binary `0`, to fall through so the branch is less commonly taken. The opposite of this is branching more often than not, which will cost an additional cycle (and potentially one more if a page boundary is crossed). It would appear silly to open `.lst` files to check and adjust page alignment, but it was a common practice on early 8-bit realtime embedded systems.

SpeedScript drops the ball here. They set their default typing mode to overwrite and in the code they leverage the zero status flag for branch without a compare, but then they branch by 'default' when they should have done the opposite.

With the background behind us, let's examine the feature callouts on the left: (1), (2), and (3).

(1) Typing mode: Insert / Overwrite

SpeedScript's core initializes to *overwrite* mode which is toggled with CTRL-'o' (customary in ancient word processors and screen editors). It also supports an insert mode, however, operating in this mode has a performance penalty* when working with large files.

SpeedScript offered no obvious indication of which mode was selected but we will do as classic PC apps did; with status line text.

State-change code:

```
instgl          lda INSMODE     ; mode variable
                eor #$0e
                sta INSMODE
  asserted rts  jmp updatestats
```

Screen-poke update code:

```
                ldx scrnpokeofs ; used in bumpdrive
                lda INSMODE
                beq ovrpoke
inspoke         lda #'I'
                sta $c025,x
                lda #'N'
                sta $c026,x
                lda #'S'
                sta $c027,x
                bra done
ovrpoke         lda #'O'
                sta $c025,x
                lda #'V'
                sta $c026,x
                lda #'R'
                sta $c027,x     [ LOW BUDGET CODE ALERT ]
```

Use code excerpt:

```
                :
                lda INSMODE  ┐
                bne notinst  ┘ Test
doins           jsr inschar  ┐
                pla          │
                cmp #95      │ Action code, if orchestrated well,
                bne putchar  ┘ will fall-through from the above test
                :            ¦
```

* Next issue, we will discuss they ins and outs of the insert *problem*. You'll know when your files are getting too large for comfort.

(2) Device: “sd” or IEC drive number

This setting, cycled with CTRL-‘d’, uses three tables and two variables in an obvious way. Here is the data:

```
drvtxtl         .text $00, 's', '1', '1', '0', '0'
drvtxth         .text $00, 'd', '1', '0', '9', '8'
drvval          .byte $00,  0 ,  11,  10,  9 ,  8
```

Initialized to SD

State-change code:

```
bumpdrive       dec drive_idx
                bne grabdrive
                lda #$05
                sta drive_idx
grabdrive       ldy drive_idx
                lda drvval,y
                sta drivenum
                jmp updatestats
```

The blue box (indexed by drive_idx) slides to the left and rotates to position 5 (drivenum = 8)

x already contains the scrnpokeofs from the routine above; this adjusts for 40 or 80 cols

Screen-poke update code:

```
drivepoke       ldy drive_idx
                lda drvtxtl,y   ; low or 1st digit
                sta $c022,x
                lda drvtxth,y   ; high or 2nd
                sta $c023,x
                lda #$18        ; vertical bar char
                sta $c024,x
```

Use code excerpt:

```
dirload         ldx drivenum
                bne sd_dir      ; if drive is 0, SD
                lda #$01        ; else IEC setup
                ldy #$00
                jsr SETLFS      ; equiv of 1,x,0
                lda filenamelen ; is 1
                ldx #<filename  ; buffer of '$'
                ldy #>filename
                jsr SETNAM
                jmp LOAD        ; loads directory

sd_dir          ldx #FAT_INIT
                jsr SDCARD
                bcc _out
                lda #0
                ldx #FAT_CTX_NEW
                jsr SDCARD
                bcc _out
                sta fat_ctx
                ldx #FAT_CTX_SET
                jsr SDCARD
                bcc _out
                lda #0
                jsr SETNAM
                ldx #FAT_DOPEN
                jsr SDCARD
                bcc _out
                :
```

[GADGET'S HIGH BUDGET CODE]

There is *a-lot* going on here, but the interesting part is the state-change code and the data. We are cycling, and therefore, iterate values (in this case, in reverse). 6502 `dec` and `inc` instruction are wonderful because they set or clear the zero flag without a ‘load’.

When zero is encountered, we set the index to 5 to reset it to the top of the array. In either case, we end up at `grabdrive` which grabs the new drive number that the use code can readily use.

There are several ways to do this, but we opt for a simple index with data; best of all, `drivenum` is usable without interpretation or a long string of compares. It’s trivial code, but it’s efficient and easy enough to follow.

(*continued on pg. 8 below*)

A Nostalgic Detour - twice the code, 4x the fun

For old times sake, let’s examine the approach I took when I was 18 years old, faced with similar requirements. Life was simple back then, no SD cards to worry about, and I had no clue how to use compiler directives, so I did it the hard way.

```
2347  DRCHN  LDA DRIVE
2348         CMP #$0B
2349         BNE DRCHN1
2350         LDA #$08
2351         STA DRIVE
2352         JMP DRCHN2
2353  DRCHN1 INC DRIVE
2354  DRCHN2 JMP DRIQUR
2355
2356  DRIQUR LDA DRIVE
2357         JSR HEX2AS
2358         TXA
2359         CLC
2360         ADC #$30
2361         STA $0887
2362         TYA
2363         CLC
2364         ADC #$30
2365         STA $0888
2366         RTS
2367
2368  ;-CONVERT HEX TO ASCII
2369  HEX2AS LDX #$00
2370         STX $02
2371  CONVE1 CMP #$0A
2372         BCC CONVE2
2373         INC $02
2374         SEC
2375         SBC #$0A
2376         JMP CONVE1
2377  CONVE2 PHA
2378         LDA $02
2379         TAX
2380         PLA
2381         TAY
2382         RTS
```

2348: `DRIVE` is checked for `11` (the max) and if so, reset to `8`; else, branch to `DRCHN1` where `DRIVE` is incremented.

2354: Note the `JMP` and wonder ‘why!’

2356: `DRIQUR` calls `HEX2AS` to clumsily convert the `DRIVE` byte to digits returned in x and y (in case `DRIVE` is `10` or `11`). ASCII 30 is added and the screen is poked.

This is where I laughed (in retrospect). I must have had my “*Mapping the Commodore 64*” book by this time. How else did I know that (2370) zero page `$02` was “unused”?

This was life in ‘84 or ‘85. Someday, I will produce a YouTube demo featuring my Terminal, and I’ll be Commodore famous!

Of course at this point I can find plenty wrong with even this snippet of teenage code. Regardless, I’m not sure I’ve ever been more proud of a project of this magnitude, since. The blue line numbers from VS Code on the left should give an indication of the sheer amount of work involved. In total it was > 4,000 lines of assembly language and data with minimal comments (and no style).

Considering I was self-taught, had limited resources, and was a freshman in College with only rudimentary dev experience (PET BASIC 4.0 in 10th grade *Computer Math* class and Pascal in *AP Computer Science* in 11th grade), I still can’t believe I persevered and got it to work. Equally shocking; I still have the `.asm` source.

To say that these works are akin to Richard Mansfield’s all-assembly language game, would not be too much of a stretch. To connect the dots backwards, it’s only been 2 years on either side of a 40-year slumber between my being inspired by Richard Mansfield book, and today.

Time passes quickly. Do something useful with it, even if it’s just for you. I hope that you are finding yourself on a satisfying journey with your Foenix machine(s). I am.

(3) Editor mode: custom char "text" or "data"

Pressing F6 toggles the editing mode between two states which dictates the editing behavior of nanoEdit. The visuals and goal are similar to item (1) above, but there is a key difference.

We are not poking printable ASCII text onto the status line (such as the chars 'I', 'N', 'S' above); instead, we are poking *custom* characters (glyphs) numbered in the lower ASCII range. Traditionally, these characters are not printable, but on the F256, this range contains an organized collection of graphic characters which we have redefined. Here is an edited (spun) view of the `.asm` source glyph def.

It's a bit like one of those mid-'90s magic-eye posters:

```
.byte   %01000001 00000000 10000000 01000000
.byte   %10000001 00000000 10000000 00100000
.byte   %10001111 00111001 11001110 00100000
.byte   %10010001 00000100 10000001 00100000
.byte   %10010001 00111100 10001111 00100000
.byte   %10010001 01000100 10010001 00100000
.byte   %01001111 00111100 10001111 01000000
.byte   %00000000 00000000 00000000 00000000
          ascii $0b  ascii $0c  ascii $0d  ascii $0e
```

Of course, we could grab each character using an immediate mode `lda` as we did in the example above, but there is a better way; a one dimensional array of values (see `modetxtdata` below) and simple bit shifts to multiply the `0` or `1` stored in `mode` by four (the length of the glyph 'string'). This value is transferred to the `y` index and then used to grab values in the array. Best of all, this method is also scalable should we add modes.

There is some complexity to deal with here; per design, glyphs appear on the lower status line (not the upper where addresses are absolute, always starting at `$c000`). On the lower line, we not only have to adjust for the number of columns (40 vs. 80 as we did above), but we'll need to adjust vertically, to account for the difference between the 30th or 60th row.

Computing the address would normally take some doing, but we will leverage a *rewritten* version of the classic `PLOT` routine, named `SCRNADDR`.

The OpenFNXkernal version `PLOT` uses bitwise math (`asl` and `rol`) in a tight, single-pass routine. It works well and is efficient, but only supports an 80 x 60 screen. Commodore's `PLOT` used a table of address values and some logic to account for the inevitable 80 character long *logical* line (despite a fixed 40 x 25 screen). We'll lean towards the latter, but code it simply. More on this next time.

Here is the editing mode glyph data:

```
modetxtdat      .byte $0b,$0c,$0d,$0e,$07,$08,$09,$0a
```

State-change code (suitable for 2 modes as written):

```
bumpmode        lda mode        ; mode variable
                eor #$01        ; will be 1 or 0
                sta mode
                jmp updatestats
```

Screen-poke update code:

```
                lda mode
                asl a           ; first shift is x 2
                asl a           ; second is x 4

                ldx #20         ; x coord
to be discussed ldy scrnheight  ; y coord (29 or 59)
next issue      jsr scrnaddr    ; returns address
                stx TEMP        ; re-use of zp pair
                sty TEMP+1      ;  indirect indexed

                tax
                ldy #$00
modetxtloop     lda modetxtdat,x
                sta (TEMP),y
                inx
                iny
                cpy #$04
                bne modetxtloop
```

Use code excerpt:

```
                :
                lda MODE
                bne hexdisplay
                inc columnnum
                :
```

The 'use' code is not important; simply, the lines highlighted in yellow are all we need to do in order to act upon the setting. The default value, optional branch, and fall-through nature is optimal. Per our prior discussion, we `bne` without the need to compare for our default use case (normal editing).

Throughout nanoEdit, we employ these techniques frequently. In one instance, we populate and indirect `jmp` vector, to prevent having to constantly check and branch; elsewhere, the code above is used simply.

In summary - the big deal

There is no big deal or 'aha' moment here. Being consistent and reusing code that is easy to modify and works can help transform a mundane task into something between fun and challenging, and ultimately, satisfying. That's the point.

In an earlier issue, I took a stab at distilling the joy of being a 6502 programmer. It continues to be a learning experience for me; something between art and science. The fact that our platform is in its infancy and offers a wealth of graphics and audio features keeps it interesting.

With [now] four different CPUs in my collection of Foenix machines (the latest being the FNX6809), I'm most comfortable on the F256 but will be shifting gears to the A2560 family sometime next year. It will be a challenge for me, but I'll be back.

I hope to release nanoEdit in beta form by years end, but a Christmas demo keeps trying to lure me away. nanoEdit looks nothing like my original minds-eye view of what it would be and in this case, it's a good story. Feature development and refinement yields a better product.

What are you working on? Reach out through the Discord `newsletter-appstore` channel or via DM. I'd be happy to write an article about one of your projects.

Calculating percentage of two 16-bit unsigned values

It's not obvious, but despite trying to keep nanoEdit 'light', we borrow from classic editors such as `vi` (sorry) and Emacs (Gadget, don't get your hopes up!) In this article, we replicate a feature of the latter and leverage 16-bit division in the process.

The 'text' editing mode includes a '%' numeric on the lower status line representing the cursor position. Calculating this **should** be simple (for a 5th grader):

cursor position aka byte offset from start ÷ filesize

Here are two examples in the context of our use case:

a. cursor at the 33rd position of a 252 byte file; so 33 divided by 252 = .1309 = 13%. Simple…

b. cursor at 60,731 of a 65,535 byte file. The answer is .9266 or 93% if we round up. Easy right? Nope, not for an 8-bit system.

The 6502 cannot divide natively. Instead, we will need to employ looping subtraction, ideally carrying the remainder forward. But it gets worse…

The literal translation of *percent* is "per 100" (but natively, our machine is integer binary), therefore, the quotient will always be 0 and the remainder will **always** equal the numerator; that's not very useful.

To get around this, we could change our approach and use two division calculations as follows:

cursor position in memory ÷ (calculated filesize ÷ 100)

Div #2 Div #1

Would BCD will help? Well, it would, but setup is complicated and each *next* digit of precision is another run through the gauntlet, which consumes clock cycles.

The goal is to estimate* the % through the file being edited (highlighted in red)

From a design perspective, placing such an indicator on the status line establishes an assumed contract: **users expect real time updates**. Unfortunately, this is exacerbated by the fact that humans can type at a rapid rate, and this leads to interface latency.

We need performant division. We could consult stack overflow or GitHub, but instead, we will look to an old friend: *"6502 Assembly Language Subroutines"*, by Lance A Leventhal. It hasn't disappointed yet.

After typing in the code (pg. 240-248) and pruning unneeded subroutines (signed portions), we are left with the following code. Never mind that the type is small, the point is, that this runnable excerpt will require > 128 bytes of memory. (the full routine which would have included signed division code, weighs in at 293 bytes plus an additional 13 bytes of variable data):

```
dvsor       .word   ?
dvend       .word   ?
            .word   ?
retadr      .word   ?
rsltix      .byte   ?

udiv16      lda #0
            beq udivmd
urem16      lda #2
udivmd      sta rsltix

            ; save return addr
            pla
            sta retadr
            pla
            sta retadr+1

            ;get divisor
            pla
            sta dvsor
            pla
            sta dvsor+1

            ;get dividend
            pla
            sta dvend
            pla
            sta dvend+1

            ; perform division
            jsr udiv
            bcc divok
diver       jmp erexit
divok       jmp okexit

; unsigned division
udiv        lda #0
            sta dvend+2
            sta dvend+3

            ; check for div by zero
            lda dvsor
            ora dvsor+1
            bne okudiv
            sec
            rts

okudiv      ldx #16
divlp       rol dvend
            rol dvend+1
            rol dvend+2
            rol dvend+3

chklt       sec
            lda dvend+2
            sbc dvsor
            tay
            lda dvend+3
            sbc dvsor+1
            bcc deccnt
            sty dvend+2
            sta dvend+3

deccnt      dex
            bne divlp
            rol dvend
            rol dvend+1
            clc
            rts

erexit      lda #0
            sta dvend
            sta dvend+1
            sta dvend+2
            sta dvend+3
            sec
            bcs dvexit

okexit      clc

dvexit      ldx rsltix
            lda dvend+1,x
            pha
            lda dvend,x
            pha
            lda retadr+1
            pha
            lda retadr
            pha
            rts
```

A heavily edited [but runnable] excerpt of UDIV16, an unsigned division subroutine as published in Leventhal & Saville's "*6502 Assembly Language Subroutines*" - Osborne/McGraw-Hill 1982

Based on the doc, this routine consumes upwards of 1,160 cycles. On a 1 MHz. CPU, this can be significant. On a 6.29 MHz. CPU, it *also* could be!

How? Because an 80 x 60 screen is more than 4x the size of most 1980s systems, and because the core SpeedScript code recalculates and redraws the screen constantly (every 1/2 a second or cursor flash), so the CPU is quite busy.

How can we implement this 2-step formula given what we've already discussed and still retain some level of performance?

Now is a good time to reintroduce you to VICKY. She is here to help:

Math Block - What is it and what is it good for?

Three things to know:

1. [similar to other VICKY features] it saves cycles and code by greatly accelerating workload that was traditionally difficult and *expensive* on 8-bit CPUs.
2. It is simple to use; just plug values into registers (memory locations) and read the result; before you even begin to setup your own code to measure performance, your answer is ready to be read (within a 6502 clock cycle or two).

3. Use it or lose it! HDL code within FPGAs consume gates which are limited in supply. Hardware designers (Stefany, in this case) often add features and refactor code; a perceived lack of interest in a particular feature can lead to its untimely demise!

   Fallen brethren included signed math functions and tragically, the "Bitmap Coordinate Math Block"; returned a bank number and offset of an identified pixel of a bitmapped screen. In a future article, we will highlight this forgotten/expunged feature and discuss an algorithm to perform this function using other parts of the Math Block.

Here are the basics on how to use unsigned division using `140009 B0` which was released in late August. Addresses change over time so if you've printed the doc, be sure that what you have matches the code level (noted on SuperBASIC's start screen as "hardware").

Address	R/W	Name	Data
0xDE04	R/W	DIVU_DEN_L	Unsigned Denominator Low Byte
0xDE05	R/W	DIVU_DEN_H	Unsigned Denominator High Byte
0xDE06	R/W	DIVU_NUM_L	Unsigned Numerator Low Byte
0xDE07	R/W	DIVU_NUM_H	Unsigned Numerator High Byte
0xDE14	R	QUOU_LL	Quotient of NUM/DEN (unsigned) low byte
0xDE15	R	QUOU_LH	Quotient of NUM/DEN (unsigned) high byte
0xDE16	R	REMU_HL	Remainder of NUM/DEN (unsigned) low byte
0xDE17	R	REMU_HH	Remainder of NUM/DEN (unsigned) low byte

The F256 includes a built-in math coprocessor for integer math. This coprocessor provides fast 16-bit unsigned multiplication and division. As well as a 32-bit adder. The use of this coprocessor is straightforward: both operands are written to the appropriate registers and then the result is read for the corresponding answer register. The math units are completely separate blocks using separate registers, so they function independently.

Unsigned Division registers and narrative from Peter's documentation

Implementation details

We will need to perform some setup using 16-bit subtraction in order to prepare the inputs for the Math Block. Let's examine two near-identical pieces of code:

- First: compute the document size in memory; above, we talk about the size of a 'file'. In reality, we are talking about a document in memory. The data we need for the division calculation is derived from pairs of 16-bit variables that the original SpeedScript code defined as follows:

```
TEXSTART    .word    ; start of memory
LASTLINE    .word    ; end of memory
CURR        .word    ; cursor (current)
```

To determine the size of the document, we will need to subtract the memory address of the first byte of document memory, `TEXSTART`, from the last byte of memory used, `LASTLINE`. These are SpeedScript's variable names hence the upper case and peculiar naming :)

We will store the result in a new variable, a 16-bit variable called `memused`. Here is the code:

```
compute_sz    stz memused+1
              lda LASTLINE
              sec
              sbc TEXSTART
              sta memused
              bcs noborrow_sz
              lda #$ff
              sta memused+1
noborrow_sz   lda LASTLINE+1
              sec
              sbc TEXSTART+1
              clc
              adc memused+1
              sta memused+1
```

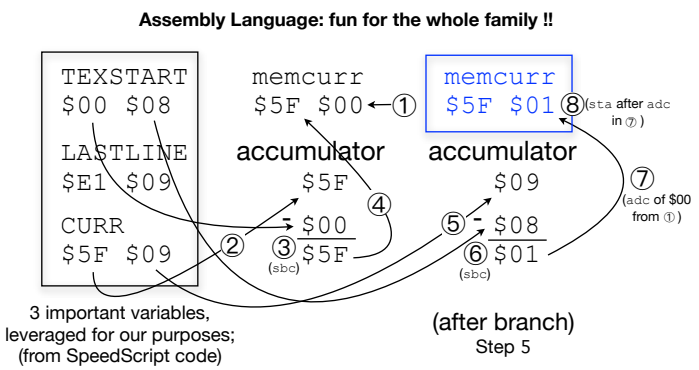
May as well acknowledge here that the `lda #$ff` is unorthodox. The intent is to engineer a 'borrow' by artificially creating a `-1`. By using a 255 in the high byte of `memused`, an add of the difference between total and beginning of memory permits the high-order borrow to manifest correctly. Below, we do the same with the next calculation.

- Next: compute the cursor position within the file or document; SpeedScript tracks the cursor position in memory in `CURR` (another 16-bit `.word`). To solve for the percentage, we will need to compute `memcurr`, it will contain the offset into the total:

```
compute_cu  1 stz memcurr+1
            2 lda CURR
              sec
            3 sbc TEXSTART
            4 sta memcurr
              bcs noborrow_cu
              lda #$ff
              sta memcurr+1
noborrow_cu 5 lda CURR+1
              sec
            6 sbc TEXSTART+1
              clc
            7 adc memcurr+1
            8 sta memcurr+1
```

The following is a walk through of `compute_cu` from beginning to the end. Step numbers are inserted above assuming no borrow. We begin with telemetry on the left, and will end with a computed `memcurr` in blue.

This scenario is based on the screen shot on page 6. The upper status line contains two 16-bit hex values representing the calculated cursor position `$015F` and the calculated document size `$01E1` poked to the screen for debug* purposes). The choice of `$0800` below is arbitrary, but not unrealistic.



*stay tuned! `memused` and `memcurr` will receive a promotion and get their permanent spot on the status line of nanoEdit's 'data' mode, discussed next issue

The great divide

Now that setup is complete, we will load VICKY's registers and perform **two** division operations, necessary per the discussion on page 9.

The code below calculates a **coarse** percentage using two unsigned 16-bit divisions and the Math Block; the result is subsequently converted into printable characters for the percentage shown on the status line (pg. 6 fig. 1).

Additional code (not shown) replaces the numerals and '%' character with "Top" or "Bot" when appropriate.

The quotient from "Div #1" is used as a denominator for "Div #2" and the final answer is deposited in `QUOU_LL` and `QUOU_HH`, then passed to `hex2asc` (not shown).

```
calcpercent
                lda memused+1
                sta DIVU_NUM_H
                lda memused
                sta DIVU_NUM_L      ; numerator
Div #1:
                stz DIVU_DEN_H
                lda #100
                sta DIVU_DEN_L      ; denominator

                lda QUOU_LH
                sta DIVU_DEN_H
                lda QUOU_LL
                sta DIVU_DEN_L      ; denominator
Div #2:
Quotient in QUOU_LL
and QUOU_HH;
remainder is
ignored
                lda memcurr+1
                sta DIVU_NUM_H
                lda memcurr
                sta DIVU_NUM_L      ; numerator
```

$$\frac{252}{100} = 2 \text{ R } 52$$

$$\frac{33}{2} = 16$$

vs. 13%? meh
(ex. 'a' on pg. 9)

As cautioned, this is a *coarse* percentage; we lose precision by ignoring the remainder and a chance at further refinement, but this example data (from pg. 9) works out well enough. It is few percentage points off, but is acceptable.

We could improve the math by rounding the result of the first division (rather than taking the integer) by inserting a small amount of code at the blue '>'above. We'll just compare the remainder (only the lower byte is needed) with 50 and branch accordingly, we'll either add 1 to the low byte of the denominator, or not; *or* we could add complexity with a different formula altogether.

There are always tradeoffs to consider. On one hand, Math Block greatly simplifies our code, but on the other, we should be wary that without further action, we will execute whatever calculation we come up with every time a key is pressed, otherwise known as often.

SpeedScript's code leveraged a Commodore 64 specific jiffy register to flash a fake cursor and squander the idle time between.

nanoEdit does something more primitive but better, and it will never get in the way of typing.

It works as follows: if we see that there is no key waiting to be serviced, we `inc` a variable and then `bne` back to the keyboard 'get' routine if < 256.

The net effect is, as long as the user is typing rapidly, the update code will not execute, but as soon as they pause (even the slightest bit) the update takes place. Here is the code:

```
wait            jsr GETIN
                bne keypress     ; service keystroke
                inc telemcounter
                bne wait         ; if not 256 (zero)
                jmp (telemupdate)
```

Two matters to highlight. First, the absolutely minimal amount of cycles that are added after each consecutive empty keyboard buffer scan.

When nothing is pending, 8 cycles are consumed (incrementing `telemcounter`) before the keyboard buffer is checked again. After 256 iterations of no keyboard activity (or about 2,048 cycles) the onscreen counters are updated.

In 40 x 30 mode, status line values update continuously despite nonsensical rapid-fire typing. In 80 x 30, the screen update code places enough stress on the system to back up the keyboard buffer, forget about this telemetry code; so the desired effect is realized.

There is more to do in this category, such as to develop a methodology to count newline characters (lines) (SpeedScript has no such telemetry), to manage a timer for auto-save, and to check for SD insertion events, but that comes later. This method suits our needs for now.

Next time

All I can promise at this point is… more. In addition to a few of the to-be-continued items from this issue, we have a dual-charset dialog box routine to discuss, the pack/unpack compression methodology, and the nanEdit "data" mode, which is a large topic by itself.

We've also got an article authored by Ernesto Contreras on the way entitled "Illustrating 8-bit dreams…"

A TinyByte - Wozniak's 6502 Floating Point code

Part of my job is research, and while working on code and material for this issue, I couldn't help but dive into a piece of work published in the very first year of Dr. Dobb's Journal (one of my favorite vintage resources).

In August of 1976, Woz co-authored an article for Dr. Dobb's along with a Stanford Ph.D student named Roy Rankin entitled "*Floating Point Routines for the 6502*".

The article portion is nothing more than a brief intro, written by the editor but it's followed by an extremely dense code listing (with copious comments), as was Woz's style. The article indicates that Woz wrote all of the Floating point functions and Rankin, the supporting routines including EXP, LOG, etc. The text captured in the screen shot on pg. 6 above is from the editor of the piece, presumed to be Jim C. Warren, Jr.

I own the first six seminal volumes from the Hayden Press compilations. They are priceless. (Volume 7 begins with the PC revolution) Lucky you, they are available on archive.org. Here is a direct link to the article; have a read, type in the code, and party.