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

(FLASH !)

New year, new format. Welcome to Foenix Rising FLASH!. Each installment will feature one or more articles; more timely, easier to consume, and faster prod-to-press time. Same *great* taste, less filling.

Jan. 2023 / F5

Memory Management on the F256 Jr.

Learn more about VICKY's MMU and paging - squeezing a 64K bitmap into ~8K of memory

F256 - Balance by design

I recently sat down with my 12 year old, a pencil, and a blank sheet of paper. I was intent on conveying the triad of compute performance; that being, the balance between CPU, Memory, and I/O. My diagram premised that a bottleneck must exist within one component of the architecture; else, an infinite amount of work would complete instantaneously; physics will not allow this!

I explained to her that soon after the dawn of time (the January 1st, 1970 Unix epoch), microprocessor introduction and innovation was fast and furious, bringing predictive branching, instruction prefetch, multi-level caching, wider and faster buses, and media density improvement, stiction reduction… (my daughter walked out at that point; back to her Nintendo Switch).

Please afford me this restart: The F256 Jr. is well balanced. When Stefany sat down with Altium designer to create an affordable but capable 65C02 based platform, she considered not only component availability and a given price point, but rightsizing the graphic capabilities, I/O, audio functionality, and importantly, the need for a memory management unit (MMU).

As it turns out, a 6.3 MHz. clock is generous when combined with the power and performance of VICKY with her sprites, tiles, color palettes, and other features.

But from a bitmap graphics perspective, there is no getting around the fact that Foenix 24-bit color (in a 256 color palette) applied to a bitmapped screen requires as much as 76,800 bytes of memory. Against the 65C02's 64K footprint, this presents a challenge.

In the prior issue of Foenix Rising, we discussed the Jr.'s memory map and byte ranges that were generally available for use, versus "off-limits", or nearly so.

We are here today to discuss how to leverage the VICKY MMU in a bitmapped screen use case.

To restate the problem: [the lesser of the F256 Jr.'s bitmapped modes] 320 by 200 by a depth of 256 24-bit colors requires 64,000 bytes which, in a *perfect world*, is mapped in a contiguous block memory.

Spoiler: we do not live in a perfect world

Subtracting the space reserved for the 6502 stack, jump vectors, and MMU control registers leaves us with about 1K of memory for all of our user code and potentially, a kernel. This is far from ideal. But the F256 Jr. is not a traditional 6502 architecture. On the next page, we will introduce VICKYs MMU; and yes, it is 'to the rescue'.

The code example on the pages below was ported from Stephen Edwards's bouncing ball demo ('balls'), featured in an Advanced 6502 YouTube video tutorial produced for VCF, the Vintage Computer Federation. In less than 35 minutes, Stephen introduces the 6502 and discusses the intermediate and advanced coding techniques leveraged.

The demo employs a set of Apple II techniques to render shapes on a bitmapped display, effectively bouncing 30 independently moving objects across a high-resolution monochrome screen, complete with simulated gravity. It is a fair amount of work for an Apple II and to an extent, also challenging for the F256.

Considering ~1 MHz. of clock speed, vintage systems struggled to move data around a bitmapped field. In the beginning, there were no sprites and limited hardware to help. Everything was done using 74-series TTL circuits, discreet electronics, and in Apple's case. some innovative coding techniques.

In this article, we will combine Stephen's code (written for a 280 x 192 monochrome display) and Peter Weingartner's gradient tint bitmapped graphics example from chapter 4 of the F256 Jr. manual.

Across the next 7 pages, we will:

- Discuss VICKY's MMU in support of the F256 Jr. architecture, including extended memory and the [future] cartridge option
- Flash back to the early days of the Apple II and discuss the memory map challenge that the Apple II bestowed upon developers and how they dealt with it
- Discuss the basics of F256's bitmapped graphics, and apply Apple techniques to the Foenix by way of a 'balls' demo that goes a few steps further

The F256 Memory Map from an MMU architecture perspective

In Foenix Rising issue #4 (pgs. 4 and 17; item b.) we introduced, and later wrote code to manipulate, the F256 Jr.'s I/O banking. In the diagram below, we will illustrate this architecture pictorially on the left side of page, and then depict the general (RAM and FLASH) MMU banking against the backdrop of the full (potential) memory map.

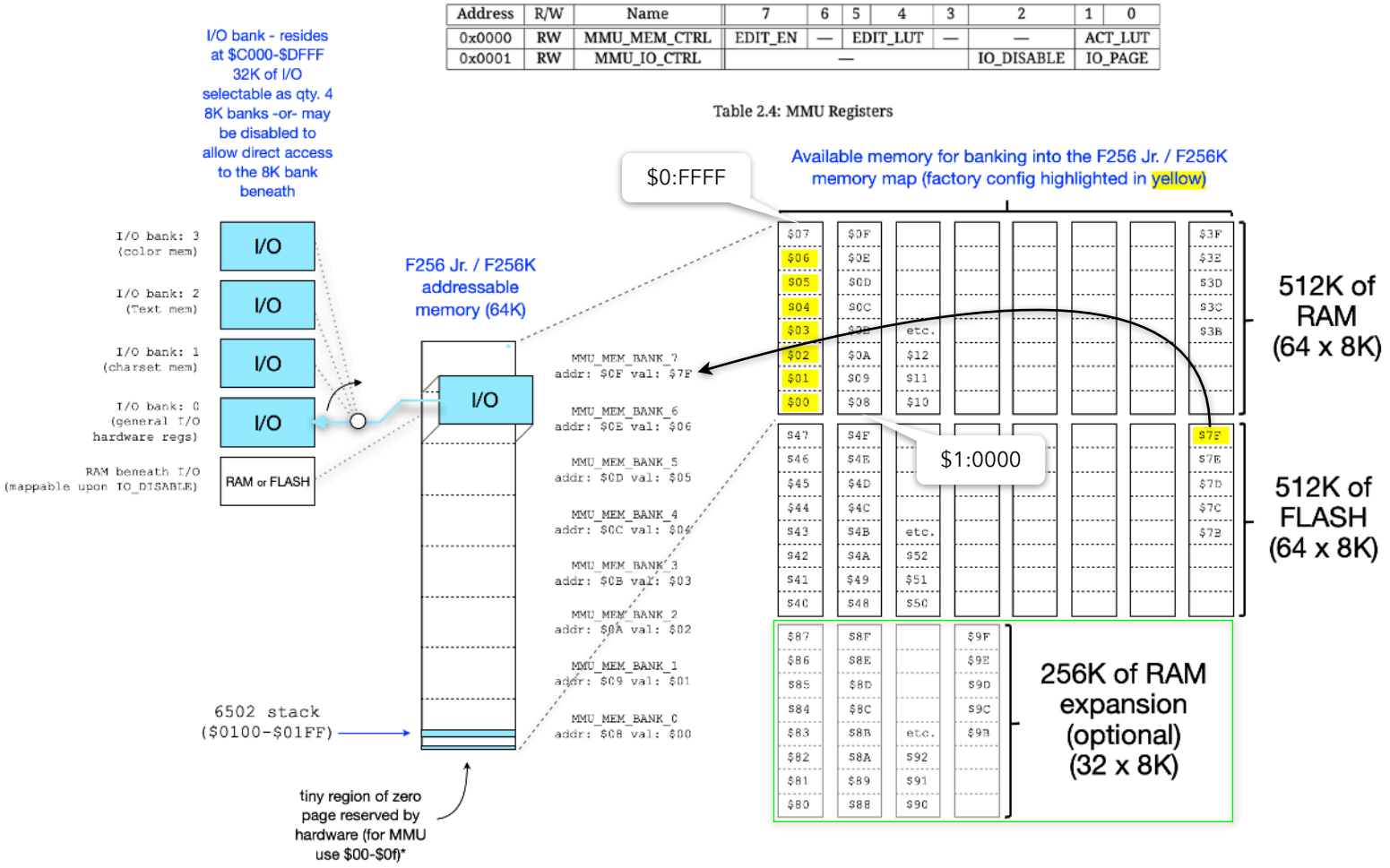
To review, a 65C02 CPU can only address 64K of memory, however, VICKY's MMU functionality, controlled via zero page locations $00 and $01, allows FOUR user controllable MMU *look up tables* (LUTs), each of which may mix and match up to 8 pages of memory in nearly any configuration from the total pool of available memory represented on the right side of the diagram below. Table 2.4, borrowed from Peter's F256 Jr. manual, details the register anatomy.

Eight zero page registers ($08 .. $0F), named `MMU_MEM_BANK_{x}`, correspond to the 8 banks of 8K in the platform's 64K memory map. In the time it takes to activate a LUT (store a value in $00; about 3 cycles), VICKY realigns memory to the desired configuration without missing a beat.

The MicroKernel leverages this feature extensively, swapping back and forth between MMU LUTs when select kernel calls are made. Most of the time, the MicroKernel stays out of the user's way, operating within a single 8K bank of code (from FLASH), and servicing simple user calls from a view of its RAM which is hiding beneath the I/O segment. Interrupts and more advanced user calls cause the kernel to save the user's state, switch to MMU LUT0 (where all of its code is accessible), perform the operation, and then seamlessly switch back to the user's LUT and code. When talking to the SD Card, the kernel makes a further switch into MMU LUT1 where 16k of FAT32 code and 8k of FAT32 RAM are mapped.

Address	R/W	Name	7	6	5	4	3	2	1	0
0x0000	RW	MMU_MEM_CTRL	EDIT_EN	—	EDIT_LUT		—	—	ACT_LUT	
0x0001	RW	MMU_IO_CTRL	—					IO_DISABLE	IO_PAGE	

Table 2.4: MMU Registers



The F256 RAM / Cartridge expansion interface

The F256 contains a slim PCIe x1 connector, capable of accommodating 256K of RAM or a rewritable (non-volatile) FLASH cartridge of up to 512K (pending release). The 256K RAM footprint is depicted within the green rectangle above.

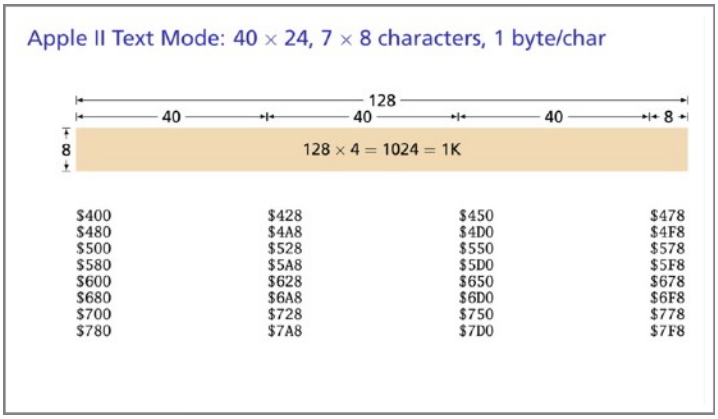
While the Jr. form factor does not host a 'cartridge slot' per se, the adapter interface is identical to the F256K. Regardless of F256 model, either form of media will appear beginning at bank $80. The optional cartridge is ideal for software distribution or for organizing your own data and binaries; yet another choice to add to the SD Card and IEC peripherals.

Revisiting Apple's text and graphics for comparison

If you watched the linked 'balls' video above, you absorbed a break-neck paced primer on 6502 and an introduction to Apple II graphics. If you didn't watch it, you should; not only because it's interesting and well produced, but because it focuses on 6502 history and the assembly language applicable to our F256 platforms.

The front half of the video discusses language heuristics, use, and theory; the back applies it to the aforementioned bouncing ball demo which we will port.

The images and slide assets represented within the video are available from archive.org via this link, so you can download them for additional study; I've pulled one slide to facilitate discussion and to highlight one piece of a thorny problem that we will need to deal with.



Representation of the Apple II text-to-memory alignment; notice 8 bytes at the end of each 128 byte (3 screen line) block; (this is the simpler of the two challenges)

The task in this case is to reshuffle and step over the 8 'wasted' bytes at the end of each 120 bytes. We are not troubled that 8 bytes are essentially wasted; rather, that we have to compensate for complexity of interleaving. As explained in the video, this issue also applies to the Apple II high-res mode, but to a greater extent.

A looped iterator with conditional math can be leveraged to resolve this, but is too expensive considering the need for performance. A better option is to throw memory[1] at the problem via the table discussed in the video.

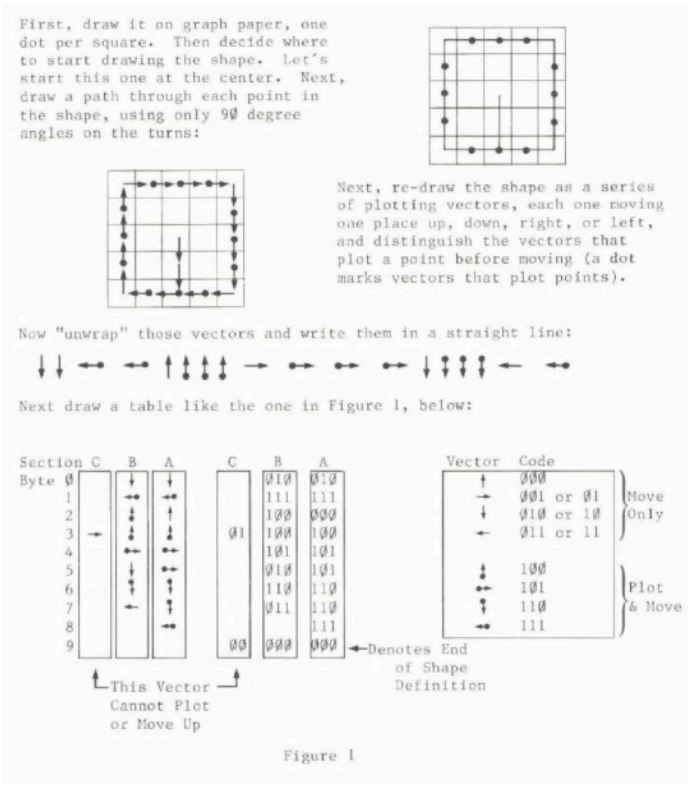
Shapes without sprites; nothing was easy in 1977

The Apple II did not have hardware based player/missile graphics or sprites; this tech was far from common in 1977, but video consoles and commercial arcade were beginning to innovate in this area. Apple's killer feature was high-resolution color. (competitors were peddling low-res monochrome character graphics, at best)

By 1978, Applesoft BASIC was released and considered stable. And it introduced integrated *Shape Tables* to BASIC. Five commands leveraging a cryptic set of vector rules, coded in 6502 and callable. DRAW, ROTate, SCALE, and others were just a call away. But golly was it slow. Turtle-graphics slow!

Shapes such as a square could be defined and moved around the screen; but for any real action, developers came to rely on 'pre-rendered' and in some cases, 'pre-shifted' shapes accompanied by optimized assembly language to drive them. The balls demo does this 30-fold.

For reference, here is a page from the Applesoft programing manual that discusses the use of vectors in defining a shape; once defined, a shape may be manipulated with SCALE and ROT commands. Tedious? yup. Powerful? Sort of. Usable? Not for our purposes; and many developers felt this way.



I am writing to you from the 21st century where we acknowledge that we are spoiled. But in our example, we will more or less pretend that it is 1977 and leverage the algorithm developed for the original Apple version.

We will ignore the fact that we have 64 sprites at our disposal and other tricks; indeed, we have a clock that is ~6x as fast as the original Apple II, but we also are dealing with ~8x as much data (a byte per pixel) and we will need to be frugal in how we spend cycles.

And as always, there are a few self-imposed constraints that we will introduce, to be addressed next time.

With this background, we are ready to get started. But first a note of thanks to Stephen Edwards. I stumbled upon his video a year ago, and leverage it heavily for this installment of the Foenix Rising Project.

Stephen Edwards is a CAL Berkeley PhD and tenured Computer Science Professor teaching at Columbia University in New York. Having met at a recent VCF, we've kept in touch and he was good enough to grant me permission to feature his work in this tutorial.

[1] Via low byte / high byte table of 192 x 2 (384 bytes)

Porting notes

With an aim to honor the original, we limited our scope to a) accommodate platform differences (hardware and constants), b) perform bit / byte conversation, and to c) leverage the VICKY MMU for bank swap.

Here is a short list of some of the work tackled:

- Changes to port from the Merlin source to 64TASS including zero page addressing for dynamic allocation (step around reserved memory) to accommodate screen geometry diffs, video mode init, palette instantiation, and startup of the *no-kernel* machine.
- Center a framed window on the 320 pixel wide Foenix screen. Since we could not easily support a 9-bit column model (without major surgery), we swung the resolution from 280 x 192 to 256 x 192.
- Modification to border draw (`vline` and `hline`) routines, using purple and appropriate dimensions.
- Adjust the `LKHI` and `LKLO` tables to accommodate paged (versus absolute) addressing. Each of the eight 8K memory banks map into address $2000.
- Addition of the MMU subroutine for frame draw which, upon select, *fetches* the associated 8K bank (see fig. 4a) into $2000 via `MEM_MMU_BANK_1`. There are several instances of playfield lines which *span* 8K boundaries (e.g. the beginning of the 320 pixel line is at the end of one 8K bank and continues through the start of an adjacent bank). We discuss this on pg. 6.
- Finally, modify the `xorball` routine by a) adding inline `ftchbnk` code and b) expanding code to work with 64 whole bytes per object rather than bits. This requires added instructions, and leaves us with ATARI PONG shaped 'balls' rather than slightly rounded edge balls. We will cover this in detail on the next page (it was messy and somewhat expensive but was the correct choice for our minimum viable version).

Platform comparison

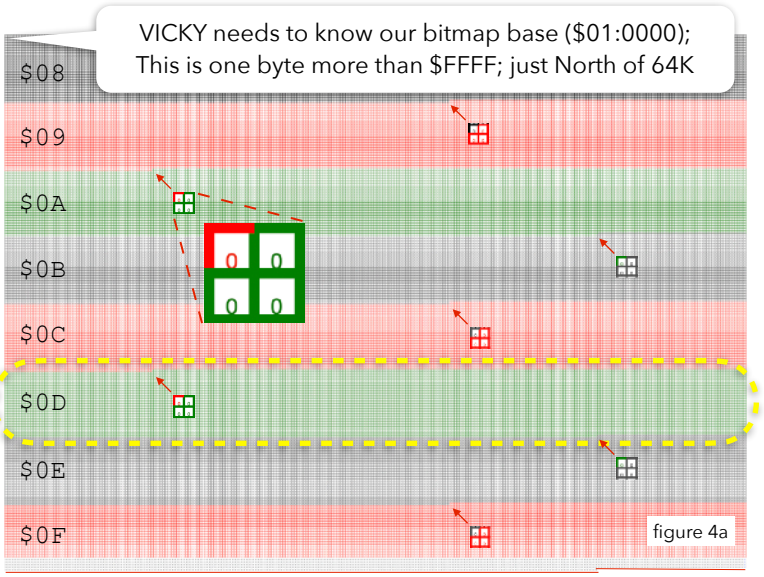
Here is a quick table outlining differences in platform specifications. This is a partial list of capabilities; five of these six measures are relevant to our discussion:

	Apple II	Foenix F256 Jr.
CPU	1.023 MHz MOS 6502	6.29 MHz WDC 65C02
Primary RAM	[up to] 64K	512K nominal
Nominal resolution	280 x 192 (when monochrome)	320 x 200 (we use 256 x 200)
Colors at max resolution	1	palette of 255 of 16.7MM RGB
Character matrix	7 x 8 pixel (40 x 24)	8 x 8 pixel (40 x 25 / 80 x 50 / 80 x 60)
Add'l graphics modes and features (out of scope[1])	Limited color at 1/2 HIRES resolution (140 x 192)	sprites, tiles, layers, scrolling, raster interrupts, DMA, math block

The Apple II was also burdened in a sense, since it has a ROM based kernel and Integer BASIC image onboard; it also has an IRQ servicing routine to scan the keyboard and perform general housekeeping.

We are not relying on this and in fact, do not even have operating code resident in our memory map. We also have IRQs disabled. The goal is to run on bare metal to the greatest extent possible.

[1] considering the diff (and 45 years) between these two platforms, the selected scope is a good center point; we will tackle some of the more progressive VICKY features soon



What's going on here?

- 64,000 mapped pixels representing the 320 x 200 bitmapped screen, pieced together in 8K banks
- VICKY sees bitmapped memory as a contiguous block; registers `VKY_BMO_ADDR_L`, `M`, and `H` at $D101 to $D103 respectively, point to the base addr. of $01:0000
- The colors to the left are arbitrary other than red, green, and gray having the same offset/transition point. (you will see some of this in `LKHI` and `LKLO` tables as well)
- The highlighted green bank ($0D) represents a bank *hypothetically* mapped to $2000 (as the 65C02 sees it) but this memory is really $01:A000 - $01:BFFF
- The bottom of the screen (8th bank) is only partially full (see the light grey area at the bottom of the last red stripe. This is because the screen data is less than 64K (base 2). This area is unused. (see the thin red jagged border)

fig. 4a - Foenix F256 bitmap display memory and addressing notes (revisit page 2 for relevant callouts)

About fetch-bank (`ftchbnk`)

In the balls video, references are made to friendly and unfriendly numbers and about Woz's design decisions in mapping memory to video, use of a 4-bit adder circuit, and related matters.

This topic is steeped in history. Aboriginal 8-bit design engineers struggled to deliver innovation at low cost. Components and board real estate was expensive. The punchline is that neither the Apple II, nor F256 "make life easy for the programmer". For this example, we could have used sprites, DMA, tiles, the math block, and other features; that would have made life easier.

Where a fairly basic lookup table was helpful in managing around the Apple II's foibles, we will adopt and code a feature which tracks horizontal video scan lines (192 out of 200 of them) to reconcile the fact that hires memory is physically stored outside of the 16 bit addressable range; and we will use VICKY's MMU to do it. To honor the original program, we will use the same table lookup methodology; in fact, we need it. This is the purpose of the fetch-bank routine and its associated 192 byte pre-calculated line-to-bank table.

Simply, armed with the screen line number (available in `HGRY`), a call to `ftchbnk` indexes into the n^{th} byte of our table and returns the bank number which is plugged into the MMU LUT for the associated memory bank.

The tables we used were derived using good old fashioned Visicalc, I mean, Microsoft Excel :). You can download the table from the Foenix Marketplace; inputs including the baseline of $2000, 192 lines, a 32 pixel offset (for centering), and 320 pixel 'stride' between rows was encoded within a few simple Excel formulas. We could have written Python to do this, but didn't want to waste any brainpower or cycles on that; I'd rather invest energy in 6502 coding.

A few 192 columns of data later (we don't utilize the last 8 horizontal lines), our tables are complete and a simple copy/paste into `.text` statements in the source replaces the `hex` directive of the Merlin assembler.

Ball versus PONG ball

Stephen's code goes to some length to use shapes with rounded corners, actuating pixel shifting to create the effect of smooth motion on the horizontal axis. The technique is similar to smooth scrolling on the Commodore 64 which leveraged the VIC-II hardware for smooth (pixel-by-pixel) movement and every 8 steps, shifts the entire character field one whole character, resetting the fine pixel shift. We've just got 320 good old fashioned pixels to deal with, so an easy change to the constant allowed us to keep most of the code as is. Some of it behaves differently, but the difference is not visible.

When we revisit this project in future, we will convert to something a bit more interesting via animation (perhaps a tribute to Drelbs; anybody remember that?). For now, the color palette is enough and we will invest our efforts in performance measurement and code up a simple UI.

The new enhancements will inevitably require use of VICKY DMA and specifically, its 2d copy, but this option is too creepy for today (Halloween is 10 months away).

Otherwise, the original shifting code was left in place, but the image is a full byte per pixel (`%11111111`) so has no effect. The result is a square object that moves smoothly with no beginning or end, and it's a solid color.

We do like PONG around here. Jay Miner (yet another pioneering and innovator) was instrumental in the development of the TIA IC within the VCS (2600) and was indirectly responsible for some of the earliest home console games. Of course, Jay went on to Amiga, developing some industry leading capabilities that were far ahead of their time.

Byte versus bit per pixel and an efficiency give-back

For better or worse, the F256 screen is ~8x as large from a memory perspective due to the aforementioned color. In order to compensate for the difference, we had to transform the following code to deal with 8 bytes at a time rather than one byte for each of two original *coarse* ball footprints (`BALL0` and `BALL1`).

To resolve this, we ended up giving back some of our 6.29 MHz. performance advantage in the worst passage of code (the tight and fast bitmap rendering routine).

Consider the following:

```
; (GBASL, GBASH) = row address
; Y = byte offset into the row
; X = index into sprite tables

        lda (GBASL),y   ; 5 cycles
        eor BALL0,x     ; 4
        sta (GBASL),y   ; 6
        iny             ; 2
        lda (GBASL),y   ; 5
        eor BALL1,x     ; 4
        sta (GBASL),y   ; 6
```

32 cycles, single pass

vs:

```
        lda #$08        ; 2
        sta XSPCNTR     ; 3
xsploop lda (GBASL),y   ; 5
        eor BALL0,x     ; 4
        sta (GBASL),y   ; 6
        iny             ; 2
        dec XSPCNTR     ; 5
        bne xsploop     ; 3, if branch is
                          taken; otherwise 2
```

25 cycles by count
Iterates 8x

Total: 204 cycles

In a sense, this is an apples-to-oranges comparison because the requirements differ, but the aim is be as close in cycle usage to the first block of code as possible.

As written, the F256 version consumes 6.375x as many cycles in this passage of code, and does so for each horizontal stripe of each ball on the screen. There has got to be a better way, and the better way is: brute force.

What if we used 8 groups of `lda`, `eor`, `sta`, `iny` instructions (**bolded** above) instead? The answer is, it will save plenty (68 cycles) and reduce the difference to ~4x. It consumes 48 more bytes but we are saving 56 bytes by not using the `BALL1` data.

Other performance notes and a pesky issue to resolve

The fetch-bank routine, since it's called for each horizontal line of every ball, has a cost; but we would not be able manipulate memory anywhere near as fast without it. The time it takes VICKY to map in a new MMU look up table, or even to instantiate changes to an existing table is immeasurable (the first instruction to attempt to measure it will consume more cycles than for VICKY to just do it).

When we made the design decision to not use the math block features of VICKY (explained briefly below), we gave up on the dream of only using 8K of our base memory for the $2000-$3FFF partial bitmap footprint.

The *seams* between memory banks exist in 7 zones vertically. This is where a horizontal line is shared between two 8K banks (see the stepped lines and call-outs in figure 4a). Left alone, this casts a dark line, visible as an object passes through. This is problematic.

It occurs because bank numbers are based on line numbers and we make the incorrect assumption that all pixels on a given line are in the same bank. It does not pose a problem for ~25 of every 26 lines, but is a corner case that we will need to make accommodations for.

There are at least four ways to solve for this:

a) write conditional code to detect and accommodate these anomalies (which will occur hundreds or thousands of times per second)

b) unconditionally fetch the +1 bank; then allow the existing `(GBASL),y` code to do its job

c) perform the lookup and fetch for every pixel, not just once per line per object (this is inefficient, and ugly)

d) use VICKY's *math block* feature, even though we said we wouldn't. Remember the premise; keep as much of the original code as possible; tables and all.

The winner is (b). It's easy and efficient, however on the surface, it seems wasteful. If we used most of main memory for code and data on a normal 6502 machine, wasting the balance of 8K would be a tragedy. But here, the time it takes to quickly bank in, access memory, and bank out the added 8K is negligible; relatively speaking. It is akin to doing a pile of work in zero time! We own this machine and have loads of resources. Life is good!

A few words about math block capabilities:

- The math block is a set of VICKY functions that perform high speed 16-bit multiplication and division
- The math block also provides complex and useful utility built upon these functions and in this context, the *Bitmap Coordinate calc* function returns the memory bank, offset, and 18-bit absolute address of any requested pixel

See figure 7.5 of the F256 manual for more detail

Telemetry - counter accumulation and display

This first version of the Foenix port includes a telemetry hook and assets to display the frequency of MMU access with a pair of onscreen counters (see pic on pg. 8).

The upper counter is associated with bank $08 (first 25 lines of the screen) and the lower, bank $0F. Counters are updated once per *frame*[1]. You'll notice updates to the upper increase only as objects cross into that bank of memory. (the bottom counter is more active).

The standard, optimized version of this program includes a few bytes of placeholder code that allow telemetry to be enabled without branching or decision statements.

The following is a theoretical example of how this works:

```
        and #7
        bne xsplot      ← original code
plug    rts (1 byte)
        nop (1 byte)    ; normally we will never get here
        nop (1 byte)
        rts
```

… and plugged w/3 bytes, [`20, <talyrtn, >talyrtn`]:

```
        and #7
        bne xsplot
new     jsr talyrtn     ; patched 3 bytes to jsr to the costly
instruction
        rts             ;  tally routine, then rts as above
```

To revert, opcode and operand bytes are pulled from a `.byte $60, $EA, $EA` statement and plugged into memory, reclaiming the efficiency of the original.

The MVP release does not include an option for switching back and forth but a subsequent version will have a basic user interface to control this, and more.

We will also be implementing a FPS calculation; the count of frames divided by whole seconds as tracked against the RTC (real time clock) circuit. It is anticipated that partial seconds against a low number of frames will yield an FPS that is not very accurate since we will only look to fetch time every 256 interrupts (4-5 seconds); but as soon as a few dozen seconds elapse, objects reach equilibrium and FPS stabilizes. (evident by the bottom group of objects losing height to creep along the baseline)

Today, you can increase the number of objects by changing `NBALLS` (default 30) and re-assembling. By choosing a high number (maybe 90), you'll be able to clearly see that the simulated gravity routine is not very smooth. Fast moving objects approach the bottom by skipping pixels; as they reach apogee, they slow to single pixel movement. The action might be a bit jumpy, but your eye is naturally drawn to the top of the screen, where pattern spotting becomes an enjoyable pastime.

An alternative to this type of gravity algorithm is to write the be-all/end-all time slicing engine to smoothly dispatch clock cycles, prioritizing the movement of faster moving objects a few pixels at a time before slower moving objects are dealt with.

[1] a 'frame' is defined as a full render cycle [ball 1..*n*]

BCD, sprites, and additional bitmap window dressing

We use BCD math for tallying bank fetch stats in the telemetry version of the code. We will not be covering it in this article, but it will be clear to you when you see it (in code comments). If you are new to 6502 or have not worked with BCD, see *Foenix Rising* Issue #4 for a tutorial in our 'Intermediate Matters' article; it explains.

Sprites are used to display text (numerals) but as a reminder, not for any of the bouncing objects. In this version of the project, we have a somewhat simple, but expensive algorithm for displaying the counters.

Once instantiated, sprites offer a highly performant method to move or locate objects on screen and only require an update to *x* and *y* location registers or in our case, memory pointers to a given digit.

In this version of the our program, we use 20 overlapped 32 x 32 sprites which point to a numeral or the comma and our sprite data is 11K in size. Another choice might have been to use standard ASCII characters but that would cost more in terms of string processing.

In future, we will leverage even more banking for an absolutely massive set (101K) of sprites in an effort to reduce the update code to a small handful of lines (for performance benefit). We will have our frame per second (FPS) code running as well and expect the difference to be measurable, and interesting.

[future] **Interrupt use**

If you are familiar with 6502 IRQs, you'll know that on many systems, every 1/60th of a second, the CPU stops in its tracks and jumps to the vector identified by the low-byte/high-byte pair stored in $FFFE/$FFFF.

IRQs are traditionally used for servicing devices such as a keyboard or a serial port, but a developer can append a modest amount of code and use it to our benefit. This is an ideal way to perform pseudo time-slicing in an architecture that otherwise does not support multitasking or threading.

The original version of the program does not use IRQs but as our workload becomes more complex, more of a focus will be placed on efficiency and the goal will be to use a handler to update the counters a few times per second rather than hundreds of time per second. This will provide more time for the task at hand, to smoothly move (and ultimately, animate) objects on the screen.

About color

If you were quizzed about the original balls demo, you'd insist that white balls are rendered against a black background, and you would be correct; except for the part where the `XOR` routine encounters pixels from two balls in the same screen location (this occurs more frequently than you might expect). In such cases, a white ball is rendered in black on top of a 2nd white ball. Stephen's video provides a good explanation of how this works here.

Our demo (since it is color), appears more complex because we have a fancy gradient background.

But what you are actually looking at is a full screen bitmap image rendered with pixels whose color is incremented every other horizontal line using a dynamically populated, but creamy smooth color palette.

Care was taken to choose a palette that was not precisely consecutive because doing so would have created a *foggy* zone around line #128 (2/3rds down our screen) where positive and negative colors are so close to each other, that the rendered object is near invisible. The palette populate routine ensures that the color is cycled every third line instead of every line.

Also worth mentioning that we artificially populated a handful of purple hues for the frame border, numerals, and dark shadows behind the numerals at the beginning and end of the palette. We don't worry about these colors being chosen by the `XOR` routine since we've made efforts to start the screen gradient beginning with color #4.

Now think back to the Apple example; white objects against a black, empty screen. Our screen on the other hand is 100% full of pixels at all times, it's just the colors that are changing to give the appearance of square objects that are bouncing. Interestingly, we did not have to add any code to get this to work; Stephen's original algorithm does it all.

Final thoughts - a personal story

In the early '80s, the Public Library in my small town had two computers; a well equipped Apple II, and a VIC 20 with a datasette. It also had a subscription to a British Broadcasting Corporation show called "The Computer Programme" which was distributed on VCR tape. After school, my friend John and his brother Gerry and I would watch and be amazed by the content which all seemed very futuristic; and specifically, the Acorn BBC Micro.

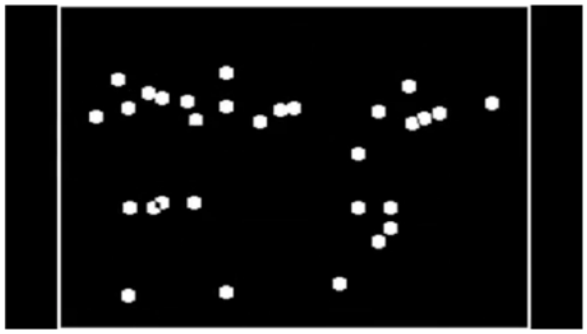
Every visit ended with a few games on the Apple II. I remember this time in my life as if it were yesterday; (who can forget playing *Choplifter* with an analog joystick). When I was accepted into AP Computer Science in 11th grade (Pascal), my Father bought me a Commodore 64, and I never looked back.

Nearly 40 years later, I bought my first Apple II. I've since picked up adapter cards and add-ons including an SD Card interface, an 80 column card, SuperSerial, the Language card, not to mention an analog joystick, a set of original paddles, manuals, and more.

By any measure, the original Apple II had a 5 year head start on the Commodore 64 and due to this, was quickly eclipsed by it. But not before it (and a few other home computers), established a bustling industry of software, hardware, and accessory vendors, some of which are still in business today.

Many of the games that I played as a youth were originally developed for the Apple II. In our next issue, we will honor one of them, so stay tuned for that.

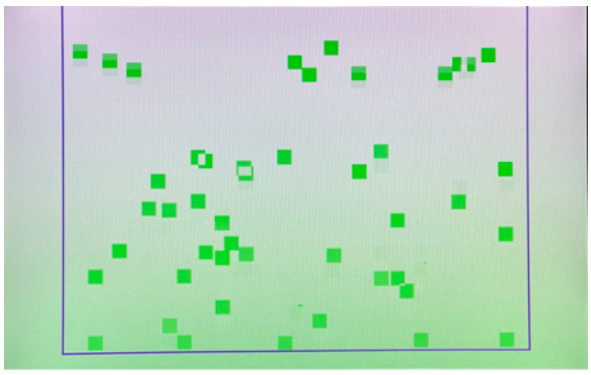
Original 'balls' by Stephen Edwards



Apple II monochrome @ 280 x 192 w/ 30 balls

- Note the XOR overlap which yields the visual artifact as explained in the linked video
- The bitmaps for the rounded edge balls are stored within the assembly language source code
- The NBALLS constant is set to 30 balls and the main loop iterates through the range, rendering each ball in its new onscreen position by XOR'ing (to erase it) and then XOR'ing again in its new position

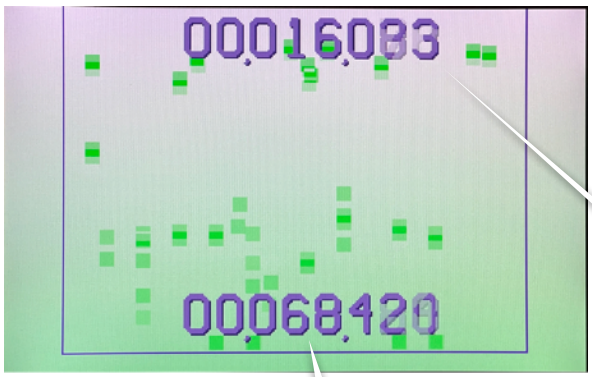
F256 Jr. port (minimum viable)



F256 Color version (MVP) w/ 48 PONG balls

- This version leverages a gradient tinted hires 'background', which as discussed, is really the foreground
- The balls more closely resemble PONG balls (they are square) but are controlled by the same vertical velocity algorithm but slightly less active horizontal movement, a byproduct of the conversion to Foenix
- Despite the overhead required to manage color, the MVP version mobilizes 48 balls with absolute ease

F256 Jr. port (with simple counters)



F256 Color version (limited telemetry) w/ 32 objects

- As above, but leverages BCD counters resulting in one 'add with carry' for each bank-fetch retrieval
- Shadowed numerals were built using Aseprite as 16 x 32 pixel two-color sprites. Maximum count is 99MM
- The on-screen counters are updated at each start of cycle (once a full run of 32 PONG balls are refreshed)
- Internal counting for all 8 banks at 4 double-nibble precision is tallied, however, only bank $08 (upper most) and bank $0F (lowest) are displayed.

Bank $0F hosts the last 25 or 26 scan lines and is mapped to $2000 each time a portion of a PONG ball is rendered there.

BCD addition adds 1 with each bank fetch and every 'frame', we update the counters. When the program begins, counting is staggered at first, but once the objects reach equilibrium, the counting progresses consistently and at a furious pace.

The total count at this lowest altitude crosses 10,000 in less than 10 seconds. In reality, the count is DOUBLE this figure since the '+1' bank is also retrieved (discussed on pg. 6 above).

As I sit here typing (1am), my machine just crossed 26.5 million banks in the bottom region. This approaches more than 200 million banked pages in total and by the time I wake up, it will cross a billion. The F256 is an amazing machine !!

Bank $08, which lives at $01:0000 is updated in batched fashion as each group of objects reach the top region of the bitmap display.

Only a portion of the balls have enough velocity to maintain this height, and due to variation in bounce patterns, they arrive in groups.

Visually, it appears as if the numerals are incrementing as the objects make contact; this is not far from the truth since the numerals are 23 pixels in height (they 'sit' squarely atop this bank)