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

(FLASH !)

Picking up from January, we will examine aspects of development & design and then dip into the code used to accumulate and convert BCD values to address pointers for sprite numerals. (who doesn't love sprite numerals?)

Feb. 2023 / F6

Joining BCD encoding and 19-bit (?!) graphics addressing

A quick look at a few blocks of code from last month's "balls" demo and then… RTC and Interrupts use

Binary Coded Decimal (again)

Discussed prior, the BCD mode of the 6502 CPU gives the programmer an efficient and low-touch method to perform arithmetic on base-10 numbers and does so in a form that is readable to humans and easily convertible for output (or to map into memory).

In issue #4, we dove into BCD, working with massive numbers in the process. Last month (issue #5), we used BCD to accumulate memory banking stats for the F256 Jr. platform in a bitmapped graphics memory banking use case (counting up to 100,000,000 per mem bank metric).

As simple as BCD is to use, connecting it to graphic data might appear unnatural, or at a minimum, non-trivial; but in fact, the opposite is true. Powers-of-two bit-shift operations and friendly memory alignment help make our job easier.

In this FLASH! feature, we discuss the code used in the telemetry subroutine of the 'balls' demo starting with the no-frills version. The assembly code on page 2 and 3 gets the job done, but there is always room for improvement.

Creating and locating graphic source data

The set of purple on green sprites used in our demo was created using ASEPRITE in the form of a 32 x 352 pixel graphic image; in figure 1a, the '0..4', '5..9', and the comma are stacked end-to-end in memory, creating a tall column with a stride of 32 pixels (the blue ASEPRITE gridlines are for reference and do not exist in the data).

The workflow of: 'design' -> 'save' -> 'load' -> 'export' create Foenix load-ready sprite and color palette files, easily wrangled using 64TASS. As written, the compiler directives below pull data into our project starting at address $8000. Omitting the '`* =`' address will encode data *inline*, and we can move it later (we have a choice). In our code, we will point to it using the following low-byte/high-byte notation: '`#<sprbmp`' and '`#>sprbmp`'

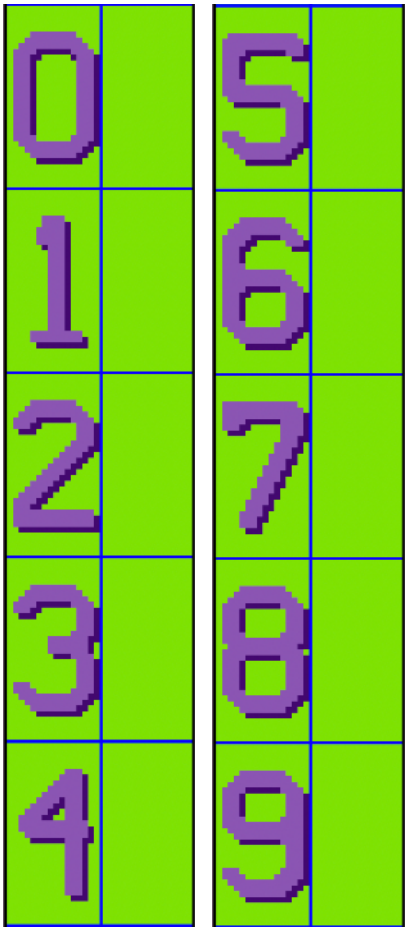
```
* = $8000

sprbmp  .binary "Sprite-10nums.data",0,11264
sprpal  .binary "Sprite-10nums.data.pal",0,1024
```

Once data is where we need it, all that is required to instantiate is to set *x* and *y* locations, set the address ($8000 in last months code) and set the sprite enable bit. (size, priority, and LUT defaults of 0 bring us across the finish line)

As luck has it, 32 pixels x 32 pixels is exactly 1,024 bytes, a size that we are pleased to work with since the offset between $8000, $8400, $8800, etc. lends favorably to the simple shift & add algorithm covered on page 2.

All in all, the approach below is a "good enough is good enough" version and is the actual code present in the telemetry version of the balls demo published last month.

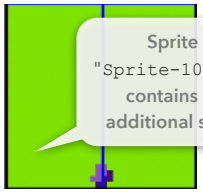
figure 1a



It performed well, but we had to limit the qty. of `NBALLS` rendered to 32.

Our next release will improve the design, cutting telemetry CPU use to a fraction (1/16th estimated).

In March, we will be publishing a "how-to" guide (an easy to follow workflow) for adding your own bitmap data (or assets from elsewhere) into your programs using a low-cost option: ASEPRITE and a no-cost classic: GIMP (the Gnu Image Manipulation Program).



Code example 1 - `telem01` (tallying counters with BCD math)

The code below represents the full end-to-end bank 'adder' subroutine which is used to accumulate 4 BCD digit pairs for a given (passed in via accumulator) bank number. We start by saving a (the accumulator) and in addition, push the x and y registers onto the stack because the calling code needs them back in the shape they sent them.

Unlike other CPUs, the 6502 is limited to only 3 'working' registers, necessitating store and load from memory. The trick is to orchestrate transfers between registers, the stack, and memory in a way that supports 'clean' code; otherwise known as *reliable*, *easy to follow*, and *adequately performant*.

Some of the VICKY I/O registers are write-only, but those that are read/write let us check state without using memory to manage (store and read); in other words, you can query and act on a value easily, and need not worry about drift.

But nothing is free; on the F256 platform, you are going to have to get used to managing I/O banks. This means establishing a policy to either keep the most oft accessed bank 'in' or alternatively, always set it based on 'need'. As an example, a text editor might opt to keep the *text* page banked in while a graphics or device-heavy use case might choose to keep the *I/O* bank resident.

If you read issue #4, you might recognize that the add #$00 with carry methodology used below is similar to the De Jong code discussed in the Mersenne Prime example. This is a common way to conduct multi-byte addition (whether in BCD mode or standard, non-decimal base 2). The `adc #$00` seems peculiar, but is the 6502 'way' to add a carry bit forward.

Word of caution: the code below will *not* work on an old school 6502, since it uses the `phx` and `phy` opcodes which are 'new' to the WDC 65C02 (also available on WDC 65C816 processors). If this were running on a Commodore 64 or PET, ATARI, or Apple, you would need to do the register shuffle; transfer y to a with `tya`; push from the accumulator to the stack; then commute x via `txa`, then push *that* to the stack, then do the reverse before returning. Bill Mensch and his friends were good enough to squeeze a few new opcodes and addressing modes into the WDC version of the classic MOS CPU.

This subroutine is called every time a bank is accessed in the `xordraw` ball loop (8x for each ball) otherwise known as quite frequently! Due to this heavy use, the 28 innocent-enough looking instructions chew up a pile of cycles in aggregate, so we short circuit the call to it when patching for non-telemetry mode.

Unfortunately, the calling code is located in a fairly tight code block which is not conducive to the patching method discussed in Issue #5; so our only option to disable without a test and branch statement is to overwrite the bytes at `ptchpt1` (patch-point 1) with three `nop` opcodes. They are about 3x less expensive than a load, compare, branch and about 10x fewer cycles than running this gauntlet unconditionally.

This can be made slightly more efficient by removing the **bolded** decimal mode set and clear instructions below, but we are being courteous (for now) and leaving the modes and registers as we found them.

For this code to make sense, an understanding of the `BANKCTR` data structure is required. Simply, it is 32 bytes of memory or 8 sets of 4 byte counters where each 4 BCD bytes (when combined) can track up to qty. 8, base 10 digits of bank access; see the depiction in figure 2a and the on-screen representation in figure 2b. Note the transposition of BCD bytes between the two.

```
telem01 pha              ; preserve a, y, x
        phy
        phx
        asl a
        asl a            ; shift twice to mult index by 4
        tay              ; accum contains bank # offset, transfer to index
        sed              ; set decimal mode
        clc              ; clear carry
        lda BANKCTR,y    ; load low-byte BCD pair otherwise known as (xx,xxx,x99)
        adc #1           ; add 1 ... we are here because a bank was retrieved
        sta BANKCTR,y    ; store back
        iny
        lda BANKCTR,y    ; get next BCD pair (xx,xx9,9xx)
        adc #$00         ; add carry in case the prior went from 99 to 00
        sta BANKCTR,y.   ; store it back
        iny
        lda BANKCTR,y    ; same for (xx,99x,xxx)
        adc #$00
        sta BANKCTR,y
        iny
        lda BANKCTR,y    ; same for (99,xxx,xxx)
        adc #$00
        sta BANKCTR,y
        cld
        plx
        ply
        pla
        rts
```

mult'ing **a** (passed in bank index) by 4 gets us to the desired 'record' aka group of 4 BCD digits

The lone +1 math add

Each `iny` Increments index to the more significant BCD byte

```
BANKCTR:

83 60 01 00 ; bank 0
.. .. .. .. ; bank 1 (not shown)
.. .. .. .. ; bank 2 (not shown)
.. .. .. .. ; bank 3 (not shown)
.. .. .. .. ; bank 4 (not shown)
.. .. .. .. ; bank 5 (not shown)
.. .. .. .. ; bank 6 (not shown)
20 84 06 00 ; bank 7
```

figure 2a

We do not display bank counters 1..6 (just 0 and 7) but we *do* tally them; called from `ftchbnk`

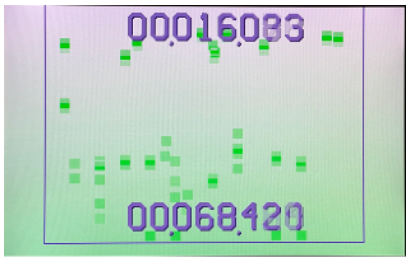


figure 2b

Code example 2 - `bcdbnk0` (converting from BCD values to sprite memory pointers)

This code example is the 2nd half of the equation. We are accumulating the totals above and do so in a neat, self-contained subroutine. The input to `telem01` above is the bank #, and the routine manipulates the 32 byte `BANKCTR` data structure; otherwise, it minds its business.

Here (in `bcdbnk0`), we read from `BANKCTR` bytes and update the sprites which are already on the screen.

The phrase *data structure* (above) is being used loosely; there is no such thing in 6502 and most of the programmers I know typically do not bother with object notation or namespaces that some of the newer fangled assemblers now support. But acting as if you have C lang style structs and organizing memory and code with the tenets of higher level languages in mind can help large projects stay organized. :)

> The primitive nature of 6502 assembly language places the onus to maintain order directly on the programmer. Absent are linters and compiler warnings, and CPU based memory protection does not exist in this universe.
>
> We sit directly on hardware, coercing circuits to do things that fulfill and exceed the original CPU designer's dreams. Development is enjoyable because it is challenging. Cycle-optimized elegance is the reward.

The following heavily commented code renders one pair of sprite numerals on-screen. Subsequent instances are identical, with the exception of the `LDA` (from `BANKCTR` in this case) and a pair of `STA` instructions (one per digit) to 'offset 2' of each sprite. (see the yellow highlight in table 5.1 on pg. 5 below) On pg. 6, you can see similar code to display digits of the onscreen TOD clock which is updated from an interrupt handler.

Table 5.1, from the F256 documentation, describes the sprite register format. The sprite address pointer is formed from 19 significant bits spread across 3 bytes. We only care about the middle byte and have initialized the high and low to 0 (see page 4). This will give us access to address in range of `$00:xx00`

despite what looks like a 65C816 formatted 24 bit address, the F256 only uses 2 of the 3 high order bits to reach *all* of memory. We (mortals) cannot address more than 64K but VICKY can! We are pointing to offsets ('xx') that range from $80 up through $BC aka 16 sprites of 1K.

As discussed, each 8 digit counter represents one bank and each is ordered and encoded as 21 43 65 87. This is followed by the same sequence for bank 1, and so on.

This ordering may seem odd, but it is the nature of BCD to have the low nibble of each byte represent the lower order place (the ones place in "21") with the high nibble of the byte representing the 10's place or the 2 in "21". It's also the nature of loops and indexes (the y index, for example) to increment. Finally, it is the nature of 6502's little endian encoding to order low-to-high.

Indeed, processor addressing modes depend upon (require) low/high, and assemblers follow the rules; but much of the rest is convention. BCD is only concerned with adding two 8-bit values and will modify carry accordingly. That's it. It's as simple as it is brilliant.

In the screen shot in figure 2b, bank 0 counter = 16,083 (padded zeros not reflected here) and bank 7 = 68,420.

If this were bank 0 and 1, `BANKCTR` memory would be ordered as follows:

	83	60	01	00	20	84	06	00
This represents byte #:	0	1	2	3	4	5	6	7

. . . and this would continue three more times for a total of 32 bytes.

To repeat, the `AND`'ing, `LSR`'ing, and math to clear carry and add hex #$80 in this implementation adjusts the middle byte of the 3 byte sprite pointer which, by design, begins at address of $8000 and aligns to 0 byte boundaries. In the Mersenne prime example in issue #4, De Jong does something different; he adds #$30 (48 decimal) to convert to ASCII, then prints to the screen using the kernel output of the mighty Rockwell AIM-65.

In a subsequent version of the balls demo, we will refactor 120 lines of code into just a handful of instructions but will *spend* a ton of memory using 128 sprite definitions. The result, from a performance perspective, will be greased lightning as compared to the original which is long and boring, but works.

We could go in the other direction and optimize for memory, using a table and indexed retrieval but that would require looping and additional cycles. Longer term, the aim is to *gamify* the demo and we will need the cycles for collision detection and more.

single load

two store

```
bcdbnk0  lda BANKCTR     ; load bank 0 counter; as noted, this code block repeats for BANKCTR+1, BANKCTR+2, and BANKCTR+3
         pha             ; save a temporarily so we can get it quickly for the low nibble calc; our example is 83 60 01 00
         and #$f0        ; mask off the high nibble only            ; start with  1000 0000 as example (nibble starts with 8)
         lsr a           ; shift right (aka div by 2)               ; right =     0100 0000 nibble is now 4
         lsr a           ; shift right again to div by 2 again      ; right =     0010 0000 nibble is now 2
         clc             ; prepare to add by clearing the carry bit
         adc #$80        ; add $80, the start of the Sprite bank    ; after add = 1010 0000 or $A0 hex which is the 8th sprite
         sta $d94a       ; store to pointer for the tens digit to mid address of sprite #8
         pla             ; low nibble retrieve from stack (the unmolested BANKCTR)

         and #$0f        ; mask off the low nibble then do the same as above but shifting left instead of right
         asl a
         asl a
         clc
         adc #$80
         sta $d942       ; store to pointer for the ones digit into mid addr for sprite #8
```

Code example 3 - `sprload` (sprite loader / copy routine: to load or not to load?)

There is nothing terribly innovative about the code below, but it is relevant in discussions focused on graphic data of one type or another. You've got data, your F256 is hungry for data, and you've got to load or move and store it and then tell VICKY where to find it.

The code is near identical to the character set 'load' routine that we used in the De Jong Mersenne Prime example in issue #4. In that program, we let 64TASS incorporate the old school Commodore PET font inline with our code and then copied it up into I/O bank 1 of $C000. But let's start over.

The routine uses two pairs of zero page (low byte/high byte) addresses and exploits the indirect y-indexed addressing mode to move sprite data from a given location in memory to the desired destination ($8000) that we depend upon in other parts of the program (the `bcdbnk*` sprite pointing code discussed on page 3).

A mishap in this calculation will yield visual garbage (aka, the math in the above code to *not* point to the correct sprite, or to any sprite for that matter). Sometimes this is entertaining; most often, it's frustrating. A miscalculation can also be the cause of a nasty system crash or wild mishaps in cases where opcodes and data gets trampled upon.

Once the sprites are copied, we can use the memory it previously occupied to overlay code or to store and use other data. In the '80s, games such as *Space Taxi* would use this technique to load additional game levels and to maximize graphics and game code for a richer experience. Modern-vintage style games such as *Lair of the Lich King* by Micah Bly and *PETSCII Robots* by David Murray also use this technique.

The bottom-left of page 1 illustrates an alternative; the "no-load" method. In this case, 64TASS assembler directives are used to align data with desired starting addresses. The assembler calculates and pads the object file to ensure alignment.

The benefit is we avoid having to write and execute code to perform data copy; the drawback is our binary file is potentially bloated. We did this in the original version of our balls demo which explains why our file size was a whopping 32K. At one end of the binary ($8000, also the load address) was graphic sprite data; in the middle was the $E000 machine code which included a modest amount of embedded table data; at the top end were 6502 vectors between $FFFA-$FFFF. All in all, 32K for the bouncing ball demo was wasteful, and I am sorry. :)

In a perfect world, we aim to organize our code, data, variables, and other assets so they reside in a tight blob* of memory that can be loaded from storage, initialized, and executed with a minimum of fuss and resources. Usually, it's preferable to have a smaller file to load and code to relocate, where applicable. Initialization routines that perform the copy are usually faster than file load routines, even from SD Card. At least, that's the way it has been.

In our next version, we will load and locate larger chunks of graphic data using VICKY's DMA to do the heavy lifting; it is capable of moving many MB/sec. which is lightyears faster than a single-MHz. 8-bit CPU. For science, we will measure the time it takes to move 32K of data using standard 6502 instructions (the code below) versus using DMA. We will do this for science, because we can. We are computer scientists!

```
sprload     lda #<sprbmp
            sta FROM_PTR
            lda #>sprbmp
            sta FROM_PTR+1
            lda #$80
            sta TO_PTR+1
            stz TO_PTR
            ldy #$00
sprloop     lda (FROM_PTR),y
            sta (TO_PTR),y
            iny
            bne sprloop
            inc FROM_PTR+1
            inc TO_PTR+1
            lda TO_PTR+1
            cmp #$ac
            bne sprloop
            rts
```

We load/copy pages from: `sprbmp` to: `$8000` ...

... and stop at page `$ac`. The math is `$ac` - `$80` (hex) = `$2c` (hex) aka 44 decimal. 44 divided by 4 pages of 256 = 11 (1K per sprite)

(but note, we start copying from a 0th page for sprite #0; the final iteration of consequence, considering the ^2 math is the 10th for sprite # 10; 0 to 10 is 11

The outer loop iterates and *then* tests/compare to `$ac`. The loop is actually copying from `$8000` to `$abff`.

* ... with the exception of the BASIC816 examples published in issues 1 through 3, all of our code has been in .bin 'blob' files. Shortly, we will be moving to .PGX and .PGZ files, the latter of which allows for multi-segment relocation upon load. Until then, we will make believe it is still 1982 and move data to where we need it to be.

Here are the pointer definitions for `FROM_PTR` and `TO_PTR`. Note the `.fill` and the use of '`?`' instead of hard addresses.

```
* = $0
.fill           16      ; reserved for MMU_MEM_CTRL ($00) and MMU_IO_CTRL ($01) plus MMU_MEM_BANK_0 through MMU_MEM_BANK_7
.dsection       zp      ; declare section named "zp" aka zero page (from $0; well… from $10 since we reserve 16 bytes)

.section zp
  FROM_PTR      .word ? ; two bytes which will be our low-byte / high-byte pair
  TO_PTR        .word ? ; same
  BLNKFLG       .byte ? ; blinking colon state
  BANKCTR
.fill           32      ; 32 byte 'data structure'
.send                   ; zp (zero-page) section end
```

Code example 4 - `sprinit` (initializing graphics mode and sprites)

In this example we configure VICKY's master control register to enable sprite graphics then populate registers for 18 sprites in the process, re-using some of the digit placement data from the bank counters of issue #5's demo. We also prepare for RTC display; specifically, the time-of-day, replacing the bank 7 counter with colon delimited digit pairs.

In 'balls', sprite registers were populated through a lengthy sequence of fall-through immediate mode load and store instructions. Here, we pull sprite config data from `.byte` statements using a loop to increment sprite-by-sprite until a `$FF` is found in the first column of the table. Since bit 7 of byte 0 is unused (see the legend beneath table 5.2), the existence of this value will not occur in normal use. The commented source and table 5.1 below, covers the rest.

The Foenix Marketplace contains a single .zip file containing the source, assembled binary, and an *updated* sprite set (a few special symbols are added to the original set of numerals; the original comma has been shifted from the 11th position to the 16th to preserve the natural ASCII order of a subset of the special symbols for which the all important ':' is part of)

```
start    stz MMU_IO_CTRL
         lda #$0C                ; use #$2C for mixed bit map and sprite mode
         sta VKY_MSTR_CTRL_0     ; Save that to VICKY master control register 0
         ldx #$00
sproutr  ldy #$00
sprinnr  lda sprdata,x
         sta $D900,x
         inx
         iny
         cpy #$08                ; process 1 'record'
         bne sprinnr
         lda sprdata,x           ; pre-check for end
         cmp #$FF
         bne sproutr
```

$D900 is the base address of the sprite register range (manual chapter 5)

Offset	R/W	7	6	5	4	3	2	1	0
0	W	—	SIZE		LAYER		LUT		ENABLE
1	W	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
2	W	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
3	W			—			AD18	AD17	AD16
4	W	X7	X6	X5	X4	X3	X2	X1	X0
5	W	X15	X14	X13	X12	X11	X10	X9	X8
6	W	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
7	W	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Table 5.1: Sprite Registers for a Single Sprite

```
;          __.|sz|sz|lay|lay|lut|lut|enable (see legend)
;         |
;         |      addr-high            x-low
;         |    addr-mid   \          /   x-high
;         | addr-low   \   \        /   /   y-low
;         |_____   \   \   \      /   /   /   y-high
;               \   \   \   \    /   /   /   /
sprdata  .byte $01,$00,$80,$00,$FF,$00,$52,$00      ; sprite 0  : ones
         .byte $01,$00,$80,$00,$ED,$00,$52,$00      ; sprite 1  : tens
         .byte $01,$00,$80,$00,$DB,$00,$52,$00      ; sprite 2  : hundreds
         .byte $01,$00,$80,$00,$C9,$00,$52,$00      ; sprite 3  : thousands
         .byte $01,$00,$80,$00,$B7,$00,$52,$00      ; sprite 4  : ten-thousands
         .byte $01,$00,$80,$00,$A5,$00,$52,$00      ; sprite 5  : hundred-thousands
         .byte $01,$00,$80,$00,$93,$00,$52,$00      ; sprite 6  : millions
         .byte $01,$00,$80,$00,$81,$00,$52,$00      ; sprite 7  : ten-millions

         .byte $01,$00,$BC,$00,$CB,$00,$52,$00      ; sprite 8  : thousands-comma
         .byte $01,$00,$BC,$00,$95,$00,$52,$00      ; sprite 9  : millions-comma

         .byte $01,$00,$80,$00,$F2,$00,$BA,$00      ; sprite 10 : ones-seconds
         .byte $01,$00,$80,$00,$DF,$00,$BA,$00      ; sprite 11 : tens-seconds
         .byte $01,$00,$80,$00,$C9,$00,$BA,$00      ; sprite 12 : ones-minutes
         .byte $01,$00,$80,$00,$B7,$00,$BA,$00      ; sprite 13 : tens-minutes
         .byte $01,$00,$80,$00,$A1,$00,$BA,$00      ; sprite 14 : ones-hours
         .byte $01,$00,$80,$00,$90,$00,$BA,$00      ; sprite 15 : tens-hours

         .byte $01,$00,$A8,$00,$CA,$00,$BA,$00      ; sprite 16 : min/sec-colon
         .byte $01,$00,$A8,$00,$A2,$00,$BA,$00      ; sprite 17 : hr/min-colon
         .byte $FF                                  ; end-of-data token
```

SIZE		Meaning
0	0	32 × 32
0	1	24 × 24
1	0	16 × 16
1	1	8 × 8

Table 5.2: Sprite Sizes

```
byte 0 legend
  7 bit - unused
6, 5 bit - size (see table 5.2)
4, 3 bit - layer (0..3)
2, 1 bit - color lut (0..3)
   0 bit - enable (1 = on)
```

We are angling to tie the RTC periodic interrupt timer to query the time-of-day of the RTC, itself. The "PIE" periodic-interrupt-enable bit turns on this function, obeying the "RS" rate nibble which can be set from 500 ms. all the way down to 30.5175 μs. (microseconds). The fact that we are notified every 1/2 second comes in handy not just because it's a reasonable interval to update the clock, but because it provides a lever to use an extremely simple piece of code to flash the ':' sprites between the digit pairs.

The Texas Instruments bq4802 does most of the work, but VICKY integrates the clock ICs signaling into its interrupt scheme and maps the R/W registers into memory on the I/O page of $C000 at documented addresses.

The use of IRQ handlers for graphics versus device servicing purposes is a broad topic and we won't do it justice here; but to preface the conversation, sometimes a use-case calls for deterministic time intervals, useful for graphics and game action; other times, IRQs are used to read device queues to prevent overflow conditions. The WDC 65C22 circuit also has timers that we can use for interrupts, but the RTC is easy and reliable.

Operating systems have multiple interrupts active simultaneously and yet, the 6502 has but a single interrupt request line, so how is this managed? Again, VICKY "to the rescue"; as you will see on pg. 6, she manages over a dozen interrupts across 3 bytes worth of mask and pending bits, so we can specify which event(s) will even bother the IRQ line and once triggered, identify the responsible party. In the 'no-kernel' model, we commandeer the entire machine but mask and check 'pending' as if there *might* be other interrupts in use; it is good practice to do so.

Code example 5 - `irqreg` and `irqhand` (real time clock interrupt registration and handler)

The code on the left installs the handler, masking all but the RTC interrupt; the code on the right is the handler itself, and does three things: a) checks to see if the interrupt is a result of the RTC timer (and not a spurious serial port or PS/2 interrupt); it b) toggles two sprite enable bits leveraging the `eor` instruction; and c) clears the interrupt and reads the RTC status bit, thus clearing its trigger in order to get ready for the next. For more information, see the article on setting the RTC on the C256U+ by Ernesto Contreras (issue #3) and Peter Weingartner's F256 manual (chapter 10 - tracking time).

```
irqreg:

sei
lda #<irqhand
sta VIRQ          ; $fffe
lda #>irqhand
sta VIRQ+1        ; $ffff

; Mask off all but the RTC interrupt
lda #$ff
sta INT_MASK_0
and #~INT_RTC
sta INT_MASK_1

; Clear all pending interrupts
lda #$ff
sta INT_PEND_0
sta INT_PEND_1

; Re-enable IRQ handling
cli
```

This code (a simplified example) might look familiar; it was stolen from the F256 github example

Pro tip: save yourself some typing and grab it, and other manual examples here

The 64TASS tilde '~' assembler directive inverts the bit mask ensuring that only the RTC interrupt is registered (all of `MASK_0` is blocked)

```
irqhand:

        pha
        lda MMU_IO_CTRL
        pha
        stz MMU_IO_CTRL

        lda #INT_RTC              ; Check for RTC flag
        bit INT_PEND_1
        beq return                ; If not, return
        sta INT_PEND_1            ; If so, clear RTC flag
        lda $d69d                 ; Reset RTC status bits

        lda BLNKFLG
        eor #$01
        sta $d980                 ; sprite enable for colon
        sta $d988                 ; .. and for other colon

        lda $d690                 ; seconds
        pha                       ; digit
        and #$f0                  ; pair
        lsr a
        lsr a
        clc
        adc #$80
        sta $d95a
        pla
        and #$0f
        asl a
        asl a
        clc
        adc #$80
        sta $d952

        lda $d692         ; minutes
        pha               ; digit
        and #$f0          ; pair
        lsr a
        lsr a
        clc
        adc #$80
        sta $d96a
        pla
        and #$0f
        asl a
        asl a
        clc
        adc #$80
        sta $d962

        lda $d694         ; hours
        pha               ; digit
        and #$70          ; pair
        lsr a
        lsr a
        clc
        adc #$80
        sta $d97a
        pla
        and #$0f
        asl a
        asl a
        clc
        adc #$80
        sta $d972

return: pla
        sta MMU_IO_CTRL
        pla
        rti
```

Identical code blocks which differ only in **source** (mem addr of RTC value), and sprite **target** (mid sprite-addr ptrs)

Wait! Why #$70 and not #$F0?

Because 'bit 7' of the hour register represents AM/PM. If we don't mask it, an 8 will appear in the 10's digit and that would be weird !!

Bit	Name	Purpose
0x01	INT_VKY_SOF	TinyVicky Start Of Frame interrupt.
0x02	INT_VKY_SOL	TinyVicky Start Of Line interrupt
0x04	INT_PS2_KBD	PS/2 keyboard event
0x08	INT_PS2_MOUSE	PS/2 mouse event
0x10	INT_TIMER_0	TIMER0 has reached its target value
0x20	INT_TIMER_1	TIMER1 has reached its target value
0x40	RESERVED	
0x80	Cartridge	Interrupt asserted by the cartidge

Table 9.2: Interrupt Group 0 Bit Assignments

Bit	Name	Purpose
0x01	INT_UART	The UART is ready to receive or send data
0x02	RESERVED	
0x04	RESERVED	
0x08	RESERVED	
0x10	INT_RTC	Event from the real time clock chip
0x20	INT_VIA0	Event from the 65C22 VIA chip
0x40	INT_VIA1	F256K Only: Local keyboard
0x80	INT_SDC_INS	User has inserted an SD card

Table 9.3: Interrupt Group 1 Bit Assignments

Group	Address	Name
0	0xD660	INT_PENDING_0
	0xD664	INT_POLARITY_0
	0xD668	INT_EDGE_0
	0xD66C	INT_MASK_0
1	0xD661	INT_PENDING_1
	0xD665	INT_POLARITY_1
	0xD669	INT_EDGE_1
	0xD66D	INT_MASK_1
2	0xD662	INT_PENDING_2
	0xD666	INT_POLARITY_2
	0xD66A	INT_EDGE_2
	0xD66E	INT_MASK_2

Table 9.1: Interrupt Registers

The tables above represent the anatomy of 2 of the 3 interrupt bytes; (the 3rd, not shown, includes 4 bits for low level IEC control and data lines)

The table to the left details memory addresses and the PENDING and MASK bytes leveraged in our code.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD690	R/W	Seconds	0	second 10s digit			second 1s digit			
0xD691	R/W	Seconds Alarm	0	second 10s digit			second 1s digit			
0xD692	R/W	Minutes	0	minute 10s digit			minute 1s digit			
0xD693	R/W	Minutes Alarm	0	minute 10s digit			minute 1s digit			
0xD694	R/W	Hours	AM/PM	0	hour 10s digit		hour 1s digit			

Table 10.2: Real Time Clock Registers

What's next? - dealing with a growing backlog

It's getting longer (we've gone off-roading again), but to join a few threads together, the following represents best-current-thinking of how to bridge recently discussed topics.

We covered item 'i.' (The RTC) above but still need to address issue #4's 'g.' (keyboard input) along with item 'k.', which revisits redefined characters - "VIC-20 style"; So the early March feature will be focused on the PS/2 keyboard.

After that, we will discuss graphic (sprite and bitmap) workflow and then pickup the Flash! issue #5 'balls' demo and item 'l.' (reading ATARI style joysticks) with some of the discussed telemetry enhancements and a low (no) budget gamification of a platform classic, just in time for VCF East, April 14-16. At least that's the plan for now!