Foenix Rising

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

(FLASH !)

Apr. 2024 / F14

This issue builds on the IRQ handler (published early last year) to construct something useful: a PS/2 keyboard state machine and a simple line editor for the F256 Jr. (or a F256K with a keyboard plugged into the mouse port). But first, we will study the pesky PS/2 protocol. And as usual, some nostalgia...

Getting from Q to RUN - a quick look at keyboard input across 3 types of machines

① Mechanical electronic typewriters -> ② Scanned matrix keyboards -> ③ Serial protocol PS/2 decoding

If a key is pressed and nobody is listening…

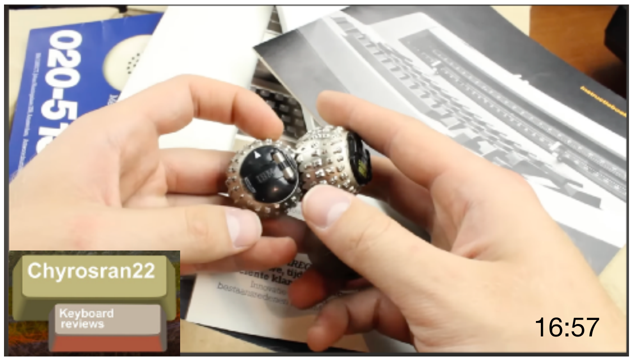
Mechanical keyboard fans are still at it; many are still buying, building, and appointing. Gamers, developers, writers, and of course, students and professionals all use a keyboard as the primary interface to the machine.

Tablets teased a glimpse at an alternate future, but they were only able to get so far. Cursed are the laptop keyboards and cherished are Cherry[1] MX keyswitches. But how did we get here?

My experience began with a Royal mechanical typewriter that I loved dearly. I was in the 5th grade and up to the usual; typing until 1am (after "lights out" on a school night). In a shared room with my younger sister, I was annoying, and my mom was not pleased. We lived in a small apartment on divorcee lane, and it was all bad. Mom barged into the room and catapulted the Royal to the far wall, cracking the metal chassis. The end! (another painful memory I'll never shake)

Three years later, my Dad loaned my Mom a company owned ① IBM Selectric (similar to the linked video below) so she could brush up on her typing skills before applying for a job with the IRS. The good old days.

This video, made by keyboard aficionado Chyrosran22, investigates the inner workings of the legendary IBM Selectric and adeptly highlights its mechanical digital to analog converters, levers, and *whiffletrees*. (If you don't have 16:57 for a full watch, find time to make it through half; you'll be rewarded handsomely with delicious sarcasm at ~ 8:27). Thomas does not disappoint.



In the days of single board microcomputers, terminals cost as much as a computer itself and by the time the hobbyist purchased a power supply, cassette interface and drive, several hundred dollars had been exchanged.

② General purpose electronic keyboards were more affordable than terminals, but they were not cheap. Apple succeeding in charging over $1,200 for their Apple II but Commodore, starting with the integrated PET, headed in the other direction and on their home computer gambit and used a low cost Mitsumi keyboard with an 8 x 8 key matrix (electronically organized in rows and columns).

The CPU (via 6526 Complex Interface Adapter) sensed cross-points in the matrix by scanning port addresses through software. The layout was designed to allow simultaneous combinations of shifted and CTRL'd keys.

The physical connection between the keyboard and the computer was a fairly clunky 20 position .100 pitch single-row connector, but it was reliable and low cost. Code within the Commodore Kernal took care of the rest, including an IRQ service routine that managed a 10-key buffer and provided kernel routines to 'get' chars.

There were no status lights and no tricks or features to speak of (just a physical shift/lock and a RESTORE key which had a fairly direct route to the CPU's NMI line).

Today, the Foenix F256 Jr. employs a similar scheme, however, the Jr's MicroKernel is more complex than the CBM kernel and is able to support PS/2 keyboards and the 20-pin Commodore keyboard simultaneously.

③ The PS/2 protocol standard (released by IBM in 1987 along with a new PC of the same name) is the subject of this article. FWIW, the PS/2 computer was dreadful.

Today, restoration junkies have access to a market full of low-cost devices to convert to and from 20-pin Commodore 64/VIC 20 (also 128D, SX64, and test rigs) **and** USB. There are also retro look-alike ITX cases like this one.

[1] Did you know: Cherry AG originated in Illinois in 1953 supplying automotive and then arcade industries. My sentimental favorite is the 'volcano' type, shown here.

Interrupt considerations

Before we get into the PS/2 protocol, let's talk more about interrupts. In the prior issue, we discussed the basics and included table 9.2 and 9.3 from the F256 manual which detailed bit values and address information.

We will continue the discussion by comparing the PS/2 keyboard interrupt (INT_PS2_KBD, represented by bit 2 or value 0x04) to the RTC periodic interval timer interrupt that we used to flash the colon (':') sprites in last issue's digital clock example.

Whatever we do, no conversation about interrupts is complete without discussing the SOF (start-of-frame) interrupt which is appealing since it's: a) deterministic from a timing perspective, occurring every screen frame without fanfare or dependency; b) is most similar to the prior gen of 8-bit machines and their resulting jiffy clock scheme; and c) in a graphics context, is useful as a dependent event to coordinate graphic objects update which reduces *tearing* (a phenomena that manifests as a glitchy visual artifact, caused when an object is being moved as the 'raster' crosses paths with it).

The following table compares the three interrupt types discussed above:

	SOF (start of frame)	RTC periodic interval timer	PS/2 keyboard
Type	fixed	static as bound to one of 15 intervals	event driven based on keyboard
Frequency	60 or 70 Hz. (display refresh)	(30.5175 *u*s. to 500 ms.)	upon keypress or release
Schedule	recurring (VICKY)	recurring (TI bq4802)	ad-hoc interactive
Operating model	1 per cycle (follows raster)	1 per cycle (timer based)	variable
Mask bit and group	0x01 of group 0	0x10 of group 1	0x04 of group 0

Do we really *need* interrupts to read the keyboard?

The short answer is "no". An improved answer is, it depends on your design goals and how much work needs to be orchestrated while a user is actively typing or during processing between user inputs.

We can avoid interrupts altogether and simply poll PS/2 KBD_IN and PS2_STAT addresses from the I/O bank page in a tight loop. The 6.29 MHz. CPU clock is fast enough for any human typist, even Mavis Beacon*. But just in case, VICKY manages a 16 **byte** buffer without our asking.

The *real* answer is we should use interrupts, and should also maintain our own keyboard buffer of reasonable length. In our code below, we maintain a 256 byte buffer.

Preposterous? Yes!; but we are aiming to demonstrate a scenario where it will matter.

Consider filtering Duke University's linux dictionary (400K) to remove all of the proper nouns. The `linuxwords` file contains more than 45,000 words which are newline delimited, starting with "Aarhus" (a city in Denmark) and ending with "Zurich". To complete this task, we will need to iterate through memory, managing banking (because 400K is far greater than addressable memory) and scan for newline characters, then check the value of the character following the newline, along the way, managing a set of pointers to copy back and squeeze out capitalized words. This will be time consuming.

If a user typed 'emergency' on the keyboard, VICKY's good graces would probably capture the 16 byte FIFO but not 16 characters. You would probably end up with something like [fragment], 'g', 'e', 'n', 'c', 'y' instead of the full word. Last I checked (now), whatever *that* is, it is not a word. But for your records, these are:

```
agency
astringency
contingency
emergency
exigency
```

About VICKY's 16 byte buffer

We'll examine the differences between scan codes, extended scan codes, and special (Pause/Break and Print Screen) sequences below, but the key point is, the PS/2 encoding standard produces variable length messages depending on the key pressed and (eventually) released. Thus, a normal keypress produces 3 bytes of output:

- 8-bit code representing the key pressed
- hex $F0 indicating a key release or 'off' state
- 8-bit code of the key released (same as above)

This translates to a minimum of 3 bytes per key and 16 divided by 3 equals at max, 5 unshifted keypresses hence the "[fragment]" above. Beyond this, VICKY keeps the train moving and runs in FIFO fashion, capturing the newest data and dropping the oldest.

It was not mentioned above, but instead of the PS/2 interrupt, we could opt for the 1/60th of a second interrupt (SOF) as our ancestors did; but that would be inefficient. Most of the time, we would check pending (as needed) and would find an empty keyboard buffer. Wasted cycles.

The approach we will take will mask all but two interrupts (the PS/2 keyboard, and the 125 ms. RTC periodic interval timer). Now let's move on to the instantiation code and discuss the logic and flow we will use to track events.

* Mavis Beacon was a fictional *character* from the popular "*teaches typing*" series. I was surprised to learn she is not a real person, but pleased by what I read in this article which describes the circumstances and impact of her persona. The original software was published in the mid '80s by "The Software Toolworks".

Interrupt registration and handler code description

The code below is similar to the instantiation code on page 6 of issue #6 where we installed a time-of-day clock update routine; the diff is called out below:

```
irqreg:

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

; Mask off all but PS/2 in group 0 and RTC in group 1
lda #$ff
pha                           ; added to save #$ff to the stack
and #~INT_PS2_KBD             ; added to prepare mask for group 0
sta INT_MASK_0
pla                           ; added to pull/seed mask for group 1
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
```

The handler routine is straightforward and does 3 things:

1. saves state* (partially), selects the I/O bank, determines if the interrupt was caused by the RTC timer or was PS/2, then branches accordingly.
2. if RTC (every 125 ms.), the routine decrements `BLNKCNT` towards zero, ultimately exclusive or'ing `BLNKFLG` using the same 3 lines of code from the sprite blink routine. When toggled to 1, it refreshes the upper 8 colors in the text RGB palette with data from colors 0 - 7; if toggled to 0, it acquires `BCKCOLR` and copies its RGB to LUT colors 8 - 15.

   During initialization, the rate is sync'd to match the VICKY cursor flash rate from bit 1 & 2 of `$D010` at either 1 sec. `%..00;` 1/2 sec. `%..01;` or 1/4 sec. `%..10` per cycle. (1/5th of a sec. is not supported) This may also be altered via new ctrlcode $16 (SYN).

   … however, if the interrupt was triggered by the PS/2 keyboard, it will blindly add scan codes to the 256 byte secondary keyboard buffer, placing data into memory as-is, updating the fill pointer accordingly.

   Since we are on "IRQ-time", we aim to process the most critical work quickly and then "get out" (we will convert from the scan code byte stream to ASCII chars in our *main* code when `getin` is called). This is where the "state machine" comes into play (pg. 6).

   Note that we own ~59/60th of the clock in the 'user' time domain; *interrupt-time* in this theoretical model is akin to 'system' time of a Unix system. Even though this is a simple 8-bit system, it's a good practice to approach time in this manner.
3. Finally, `return` cleans up loose ends by restoring registers, and `RTI`s back to the preempted program.

We will not say much about `outstrng` (issue #4) here, but will share that the handler code is all that is required to complete the character flash attribute feature.

It directly manipulates the text color LUT, therefore, will affect any character printed with our flash attribute.

The following draft code is about 90% there:

```
irqhand: pha
         lda MMU_IO_CTRL  <---   WHOOPS !!  I forgot to preserve
         pha                     and later restore x and y !!
         stz MMU_IO_CTRL

         lda #INT_RTC            ; Check for RTC flag
         bit INT_PEND_1
         beq chkps2              ; If not, check for PS/2

         sta INT_PEND_1          ; If so, clear RTC flag
         lda $d69d               ; Reset RTC status bits

         dec BLNKCNT
         beq flash
         jmp return

flash    lda FLSHRATE
         sta BLNKCNT
         lda BLNKFLG
         eor #$01
         sta BLNKFLG
         beq unflash

unflash  ldx #$00
_loop    lda palette,x
         sta TEXT_LUT_FG,x
         inx
         cpx #$40
         bne _loop
         bra return

flash    lda BCKCOLR
         ldx #$00
_loop    sta TEXT_LUT_FG+32,x
         cpx #$20
         bne _loop
         bra return

chkps2:  lda #INT_PS2_KBD        ; Check for KBD flag
         bit INT_PEND_0
         beq return              ; If not, return

         sta INT_PEND_0

_loop    ldy kendptr
         cpy khedptr
         beq return
         lda $D642               ; read PS/2 byte
         sta keybuf,y
         inc kendptr
         lda $D644               ; read PS/2 status
         and #$01                ; bit 0 = 0 if more
         beq _loop

return:  pla
         sta MMU_IO_CTRL
         pla
         rti
```

Next time, we will revisit, enhance, and refine an output routine; we will also tie in `getin` (discussed on pg. 7 below) and put a bow on our first complete program.

We are working towards a tiny set of character based I/O routines that when combined, will form a collection of moderately featured subroutines for program use. Call it a *Nano*Kernel. Keep your expectations low; we have to walk before we can run, and we are barely crawling!

Here are some overall design goals:

- the main code will fit entirely within the 8K bank beneath `$C000` and will include a resident monitor.
- the keyboard buffer, IRQ routine and jump vector table will sit adjacent to the 6502 stack and will occupy no more than the 512 bytes from `$0200-$03FF`. It will consume 16 bytes of zero page memory from `$20-$2F`.
- Later, IEC support will be added for 1591 and Commodore 1541 compatible drives including SD2IECs. Goal = do this by the year 2030.

*The 65C02 interrupt microcode preserves the program counter (PC), status register (S), and the implicated BCD (D) flag, but we still need to save registers and anything else we meddled with in darkness.

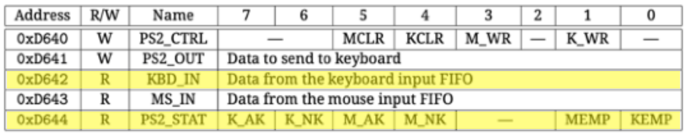
About that PS/2 keyboard interface

We don't have time or space to discuss the inner workings of the PS/2 keyboard protocol but there are excellent resources and YouTube content available.

Ben Eater's "Keyboard Interface" PS/2 masterpiece is a must watch, and I urge you to spend the better part of 30 minutes doing so. Consider it pre-req to running the F256 PS/2 keyboard sniffer app that we are coding this month.

Or, if time is tight, fast-forward and start at 21:40 and just watch the last 10 minutes; Ben talks about keycodes specifically. You can leave the framing and shift register TTL electronics for another day.

The core of our keyboard handler routine on pg. 3 above leverages `$D642` and `$D644` registers. As we have prior, the following table was borrowed from the F256 manual and it details the PS/2 mouse and keyboard interface:

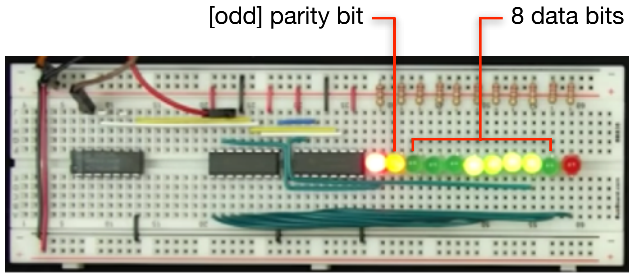


Address	R/W	Name	7	6	5	4	3	2	1	0
0xD640	W	PS2_CTRL	—		MCLR	KCLR	M_WR	—	K_WR	—
0xD641	W	PS2_OUT	Data to send to keyboard							
0xD642	R	KBD_IN	Data from the keyboard input FIFO							
0xD643	R	MS_IN	Data from the mouse input FIFO							
0xD644	R	PS2_STAT	K_AK	K_NK	M_AK	M_NK	—		MEMP	KEMP

Table 13.1: PS/2 Port Registers

Today, we are only interested in the FIFO (first in/first out) queue associated with KBD_IN, and bit 0 (KEMP) of the PS2_STAT register; it will contains a bit value of 1 when the keyboard is empty.

If you watched the video above, you'll be pleased to know that VICKY dispenses with the framing and parity bits for us, depositing only the 8 bit byte of each partial into the KBD_IN register. Pictorially, we are talking about the green LEDs:



The key being pressed here is $1E, or the number '2' (non-keypad). Note that the yellow LED is illuminated in order to create "odd parity" (there are 4 green bits on). These 9 bits are book-ended with the red start/stop bits totaling 11 bits.

The code on lines 36-44 above reads a byte (we wouldn't be here unless there was *at least 1 byte pending*), places it directly into our y-indexed keyboard buffer then checks to see if another key is pending*; if so, it branches to the local `_loop`; else, it exits.

Along the way, we increment a zero-page variable `KEYPTR`, which is our index, pointing to the next byte in our 256 byte queue.

Earlier in this article (on pg. 2), we mentioned that scan codes vary in length depending on message type. They range from a single byte (scroll lock), to as many as 10 bytes (the print screen key) between the key press and release.

The video highlights a few examples of this scheme and Ben flashes this cheat sheet a few times. To review, I've compiled the following table which cites several of the same examples but adds a few noteworthy combinations that are more relevant to 8-bit retro computers.

	type of key or activity	# of bytes	'on' value aka *make* code	off value aka *break* code
Scroll Lock	'special'	1	$7E	n/a
lower case 'a'	standard keycode	3	$1C	$F0, $1C
'c'	standard keycode	3	$21	$F0, $21
Shift (left)	standard keycode	3	$12	$F0, $12
Ctrl (left)	standard keycode	3	$14	$F0, $14
Left Arrow	extended keycode	5	$E0, $6B	$F0, $E0, $6B
Enter	extended keycode	5	$E0, $5A	$F0, $E0, $5A
capital 'A'	multikey sequence	6	$12, $1C (shift), (letter 'a')	$F0, $1C, $F0, $12
^c (ctrl c)	multikey sequence	6	$14, $21 (ctrl), (letter 'c')	$F0, $21, $F0, $14
Pause/ Break	'special'	8	$E1, $14, $77, $E1, $F0, $14, $E0, $77	n/a

*It's unlikely that there be a single byte in VICKY's keyboard buffer so we may as well check KEMP and read whatever is pending while we are here. There is less overhead checking KEMP 'that one-last-time' than there is exiting the handler and winding up here again. In addition to the outer-most reaches of the handler code, the CPU itself incurs some amount of overhead every time the IRQ is raised.

A quick review of [generic] stacks and queues*

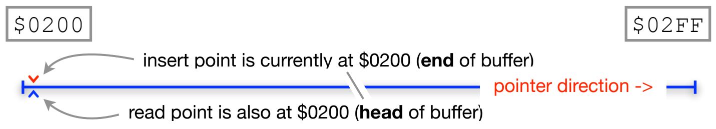
One of the most basic data structures in software is a stack and the 6502 and other processors and languages rely on them heavily. They require a consecutive range of bytes for 'pushed' data, and a single *pointer* (typically one byte) which points to memory locations where data will be pushed to or pulled from.

In the context of the 6502 processor, when you cause a byte to be *pushed* onto the stack, the stack pointer move by one position (for illustration, if it started at $FF, it will now contain $FE); *pull* from the stack and the pointer moves one position in the opposite direction (it increases by one). At one extreme, the pointer = $FF and the stack is empty; on the other, the stack pointer is $00 and the stack is absolutely full. Decremented further, and the pointer will wrap around the famous "stack overflow" condition results. Stacks are known as LIFO data structures; last-in-first-out. For a visual representation of a stack, see pgs. 9-15 from Philip Koehn's Johns Hopkins University materials here.

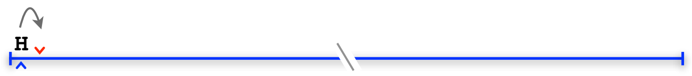
A queue is different and requires two pointers; an 'end' of queue, and the point in memory where data will be read from, or the 'head' of queue. Think of a line of customers; you get on line at the end; the next customer is serviced from the head of the line.

When 'in' = 'out', the queue is empty. If the two pointers cross, the equivalent of an overflow condition can occur.

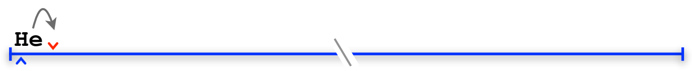
Well architected queues (when full) simply will not allow additional data elements to be added. They either return an error condition or alternatively, continue operation but do so without changing the distance between the head and end pointers. Consider the following examples:


$0200
$02FF
insert point is currently at $0200 (end of buffer)
pointer direction ->
read point is also at $0200 (head of buffer)

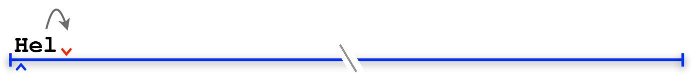
Upon initialization, 'end' = 'head' and the queue is empty.



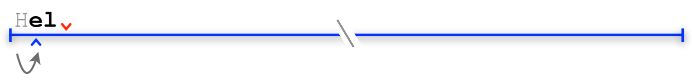
With one byte ('`H`') added, 'end' is advanced 1 byte (from $0200 to $0201).



The next byte has been added ('`e`'), 'end' is again advanced (from $0201 to $0202). No reading as of yet.



Now ('`l`') is added ('end' is now $0203).



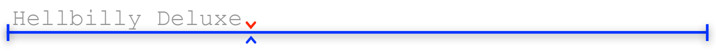
Finally, the '`H`' character is read and 'head' is advanced. The char is still in memory, but the queue contains "`el`".

Finer points of an 8-bit keyboard queue use-case

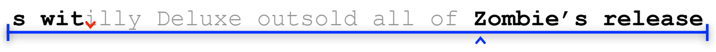
Unlike a stack (which adds and reads based on a single pointer position), a queue adds and reads from different pointers and they [typically] move independently.

Examine the following scenarios and behaviors which are modeled using a 48-byte queue depicted below:

- Queue pointers will advance in one direction only (increasing; or left-to-right in our diagrams); but they will eventually wrap back to zero; this is by design. Importantly, the end-ptr may not pass the head-ptr.
- By checking to see if 'end' = 'head' in our code (below), we can trap "full queue" conditions and prohibit additional adding, thus avoiding corruption that is sure to result when we lose bytes at the head end.
- 16 and 32 bit architectures (and high level languages such as 'C' and Python) can more easily support large queues without hardship, but here in 8-bit land, we would need to manage carry bits and 16-bit pointers.
- The following depicts an empty queue, after processing the name of Rob Zombie's most acclaimed solo effort:



- The following illustrates a queue which has wrapped but is still coherent. You'll note that the old data is present (depicted in grey) but as far as the pointers and algorithm are concerned, it does not exist.



  We have read up to and including "`all of `". The text which is still in the queue and valid to be read includes "`Zombie's releases wit`". As rest has already been processed; rendered in grey text.

- Should the buffer fill (when `kendptr` = `khedptr` as tested on line 37 below), our handler will exit. Regardless of address, the situation is as follows:
  - `kendptr` advanced following the most recently added keystroke (byte), so this address is in theory, the 'next' byte in the buffer *to be* written to.
  - `khedptr` is pointing to a byte in the buffer that has not been read yet, but it is valid and waiting.
  - this condition will persist (no more bytes added and no change in pointers) until user code calls the `getin` subroutine, which reads the aforementioned but-not-yet-read byte and advances `khedptr` by 1. At this point, the keyboard buffer will again be open for business, but only for 1 byte (for now…).

The following is an excerpt from `irqhand` on page 3:

```
_loop     ldy kendptr
37        cpy khedptr              ; test for full buffer
38        beq return
39        lda $D642                ; read PS/2 byte
40        sta keybuf,y
41        inc kendptr
42        lda $D644                ; read PS/2 status
43        and #$01                 ; bit 0 = 0 if more
44        beq _loop
```

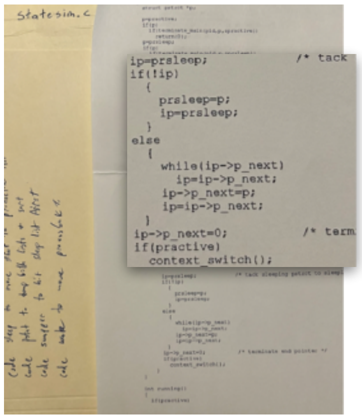
*Our keyboard handler will need to work with keycodes (not clear text); but the queue algorithm is identical

State Machines: my first, favorite, and the PS/2

In the early '90s, I attended a class at NYU based on Goodheart & Cox's "The Magic Garden Explained". It was a Unix Kernel internals class but you'd never guess that from the book title.

The final project was to write a *State Simulator* in the 'C' language. Interestingly, the course had no prerequisite for proficiency in 'C' which to me, was amusing; some of my classmates were not amused.

my NYU code (cryptic)



```
ip=prsleep;                                /* tack
if(!ip)
  {
    prsleep=p;
    ip=prsleep;
  }
else
  {
    while(ip->p_next)
      ip=ip->p_next;
    ip->p_next=p;
    ip=ip->p_next;
  }
ip->p_next=0;                              /* term
if(practive)
  context_switch();
```

death by pointers (but it worked)

The program (statesim.c) had to simulate a Unix OS, including a working shell that allowed processes to be init'd, forked, managed to and from background/ foreground and terminated; it had to support signals (SIGWAIT/semaphores), process scheduling & priority, and had to display 'state', including consumption of resources and availability.

It was my first experience having to consider and track resources then develop a system to manage them from nothing but a written specification and the input/output functions of `<stdio.h>`. Of course it was theoretical, but it was challenging and profoundly impactful to me.

I did not think much of it at the time, but looking back, each process was in one state at any particular point in time (sleeping, running in the foreground, running in the background, waiting on a message, terminating). It was a state machine simulation before I knew such a concept existed. The title should have been a clue (it wasn't).

But by far, my *favorite* state machine of all was the 1974 Gottlieb Electro-Mechanical machine that an eBay seller at the other end of New Jersey was showing me (but would not sell to me because it was already sold!). It contained a massive array of latch relays, stepper units, mechanical flip/flops, score motors and reels, binary and decagon counters, and solenoids, all to track gameplay as a single steel ball raced around the playfield striking targets for recreation. It was Pinball.

And here we are, the subject of the moment is a keyboard state machine. When combined with a handful of other routines, it will contribute to a minimum viable platform to capture keyboard input and display multicolor text. With it, we can write something simple and useful.

A state machine is an algorithm that can track matters and be in one of a number of valid states at a given time. It can also change from state-to-state in response to external inputs. A *finite* state machine is defined by a list of known states (including an initial state) and the conditions for each transition between. This is, more or less, the Wikipedia definition and it is lofty; especially as captured on this page. But it's a framework for tying user actions to desired *outcomes* and it will serve its purpose.

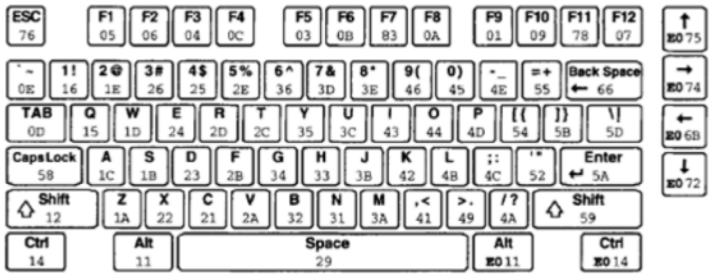
In the context of the F256 integrated keyboard, one outcome might be to change the color of the cursor from WHITE to CYAN; one might be to terminate a running program; one might be to pause screen output or trap to enter a machine language monitor for debugging, or to do something special when a sequence of keys are pressed.

These examples are very old-school but novel on our platform; to my knowledge, none of this is supported by the F256 today; but that is about to change.

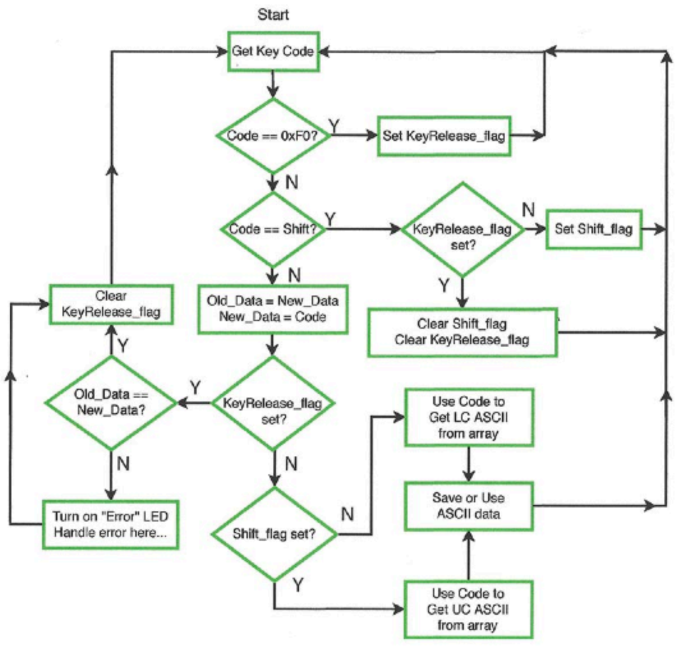
For this part of the project, we begin with an initial state:

- all locking keys 'off'
- no keys pressed
- keyboard initialized (PS/2 keyboard)

On the prior pages, we discussed and provided references to understand the PS/2 protocol and a method to get the next byte and place it in-queue. To accomplish our goal, we will track the state of pressed and released keys so we know which keys were shifted (for upper case letters, of course; but also for '$' instead of the numeral '4' when appropriate. This diagram is a good for our purposes:

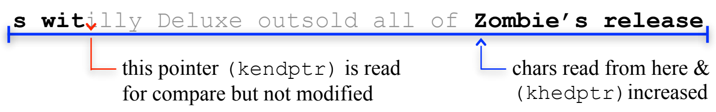


The flowchart below, borrowed from a Jon Titus article, is also good-enough. Once implemented, it will convert the PS/2 byte-stream into upper and lower case ASCII. It's not awesome, but we can improve it over time. One notable omission is support for CTRL key combinations that we (I) love. We will need to add support for that in our code.



`getin` – **Reading from `keybuf` (low budget version)**

Continuing the discussion from the queue illustrations on pg. 5, calling `getin` will return the 'next' character ('**Z**' in 'Zombie') from the keyboard buffer pointed to by var `khedptr`. This is the queue head. If buffer is empty (`kendptr` = `khedptr`), 0 is returned.


s with lly Deluxe outsold all of Zombie's release
this pointer (kendptr) is read for compare but not modified
chars read from here & (khedptr) increased

With data in `keybuf`, the 8 lines of code takes care of it. Remember, the interrupt handler at the bottom of pg. 5 (detailed in full on pg. 3) is the 1st half of the picture. It **fills** the keyboard buffer; this code **empties** it, one byte at a time as `getin` is called.

```
getin   ldy khedptr      ; load index in case char
        cpy kendptr      ; check for empty buffer
        beq _return      ; branch if empty
        lda keybuf,y     ;   else, load from buffer
        inc khedptr      ; advance head ptr
        rts
_return lda #$00         ; if no keys in buffer, null
        rts
```

The Commodore Kernal version of '`getin`' returns values which match PETSCII (ASCII) keypresses. We've not yet dealt with the complexity of the PS/2 keycodes, but the code will work just the same; it's just returning the next byte from the queue via (a)ccumulator.

To get actual ASCII and support special keys, we need to build further. We can write a more robust, monolithic version of `getin` (which applies the logic in figure 6a), or we can leave this subroutine simple. For today, we will choose the latter because it isolates the functionality and makes debugging more simple.

`getline` – **Read a line of text using `getin`**

This subroutine lets the user to enter a string of variable length characters up to *n* (but not longer than 254), storing the entered text in the low byte (y register) / high byte (x register) which points to `inbuff`. An input limit (# of chars) may be specified in the accumulator (a).

- if the user types a backspace key (keycode $66), the string will be shortened by 1 character.
- if the user presses the ESCape key (keycode $76), the buffer will zero and be set to null.
- if a carriage return (enter) is pressed (keycode $5A), the accumulator will be set to the length of the entered string, a null ($00) will be stored in the string buffer (at length + 1), and the routine will exit.
- extended keycodes (alt key, and function keys) are discarded/ignored for now, but we will handle the CTRL key.
- the CapsLock key toggles caps only. It's not a shift lock, it's CapsLock which is the way modern keyboards work. But contrary to PCs, the combo CapsLock'd+shift will not result in lower case.

Next time; more discussion on this topic, a look at some working code, and an intro to a familiar character output routine (CHROUT), complete with some bells and whistles.

In retro-retrospect: One year ago

A few words about this article and a recently published set of videos that might be of interest

Work on this article began in March of 2023 but had to be put on hold to spare cycles for prep of the Foenix exhibit at VCF East. The preparation included hardware, software, and several hundred pages of printed handouts.

Three Foenix systems were ready along with an Apple II+, monitors, and accessories, when an untimely stumble off a commuter rail platform landed me in the hospital for a MRI which led way to a spinal procedure, too many meds, two months of disability, and 6 months of physical therapy. My writing career was over.

Fast forward to today and I'm writing again and ready for VCF East, but this time as a volunteer. My back is in better shape thanks to that wake-up call, and I've picked up part of what I left behind (finishing this article). Several other efforts are still waiting for their turn. I'll get to them eventually.

The impetus for rescuing this work was a series of Discord posts seeking a low-budget (sans kernel) input output routine. Note that we are not discounting the usefulness of Microkernel, it's just that sometimes, simple is better and a great way to learn about a machine is to get in there and build from the hardware on up.

It is in this spirit that I'd like to draw your attention to Ben Eater's works (again) and his recent videos where he ports Wozmon, pulls and assembles a version of Microsoft BASIC, and ultimately, writes an interrupt based keyboard input routine, similar to what we just discussed.

Had I completed this article a year ago, I would have beaten Ben at this one tiny thing! Maybe next time.

To add to the "Keyboard Interface" YouTube video link on page 4, here is a short list of Ben's latest. Whether or not you buy his kit from Jameco, his videos are relevant to our platform, and extremely well produced. Enjoy!

"Running Apple 1 Software on a breadboard (Wozmon)"

"Adapting Wozmon for the breadboard 6502"

"A simple BIOS for my breadboard computer"

"Running MSBASIC on my breadboard 6502 computer"

"How input buffering works"