Foenix Rising

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

(FLASH !)

This early-access article focuses on MicroKernel DOS and related topics.

It was originally intended to be part of a multi-article FULL issue of Foenix Rising (and it will be); considering the amount of work and testing that was invested producing it, I thought it best to release it first as a FLASH! format article.

The next full version of Foenix Rising will be released on December 24th! I hope you like it (and this) ...

MicroKernel DOS - Hidden beneath SuperBASIC exists a set of system tools, providing support for disk operations and general utility. This article digs into sample code and assembler output, discussing how to use MicroKernel in the process.

Dec. 2023 / F12

This page is otherwise blank

:)

MicroKernal DOS & more …

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

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

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

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

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

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

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

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

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

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

Running .PGX files

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

convention

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

mind the gap; a space is required

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

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

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

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

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

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

Still, a 3rd type of binary file contains an auto-execute header, akin to the old Commodore 64 "CBM80" cartridge standard. The F256 auto-exec functionality is applicable across flash memory, flash cartridges, and ram memory; also to be discussed further another time...

MicroKernel DOS Command overview

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

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

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

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

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

Figure 3a - DOS help

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

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

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

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

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

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

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

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

retro remnant: land-line warfare

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

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

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

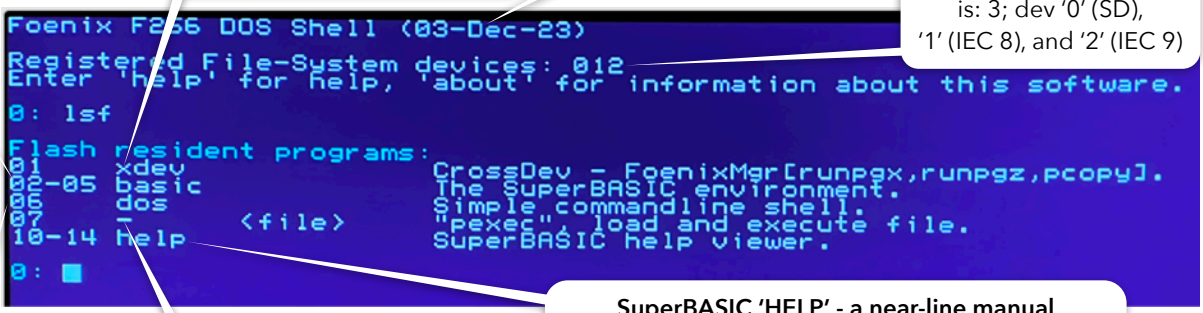
`keys` - invokes a utility which tests the use/action of your input devices (keyboard and joysticks) against kernel event routines. From a practical standpoint, this utility does not serve much of a purpose except to check for broken switches or keys, or to highlight anomalies in PS/2 support, but it's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

SuperBASIC
4 x 8k blocks containing binaries

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

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

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



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

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

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

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

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

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

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

MicroKernel Code samples

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

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

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

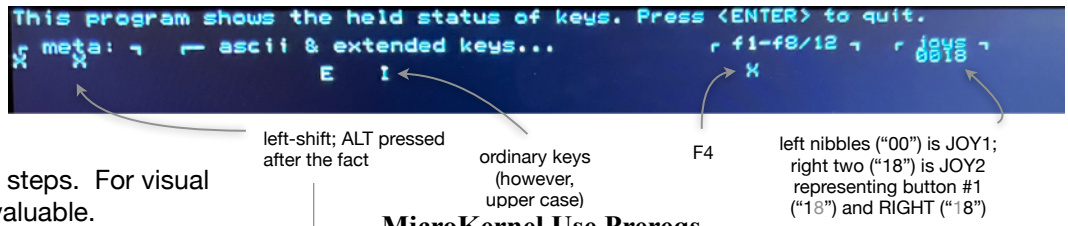
Let's conduct a mini case study of the DOS "keys"utility, (inside-out) in an effort to provide a MicroKernel starter app, and to answer one of the most oft asked questions about writing code for the F256: "how do I scan/read the keyboard and joysticks. Is there any example code?".

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

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

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

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



MicroKernel Use Prereqs

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

Ok, here's the drill:

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

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

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

single occurrence of $4189-$418b

$418c is *repeated*

```
4189    event     .dstruct   kernel.event.event_t
4189    type      .byte   ?  ; enum above
418a    buf       .byte   ?  ; page id or zero
418b    ext       .byte   ?  ; page id or zero

418c    key       .dstruct   kernel.event.key_t
418c    keyboard  .byte   ?  ; keyboard ID
418d    raw       .byte   ?  ; Raw key ID
418e    ascii     .byte   ?  ; ASCII value
418f    flags     .byte   ?  ; Flags (META)

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

418c    timer     .dstruct   kernel.event.timer_t
418c    value     .byte   ?
418d    cookie    .byte   ?

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

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

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

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

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

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

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

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

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

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

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

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

Memory use and the manifestation of `event`

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

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

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

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

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

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

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

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

Where are the MicroKernel Vectors? (entry points)

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

What about writing to devices and setting attributes?

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

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

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

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

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

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

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

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

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

06 .dsection   code

07 .virtual    $0000    ; Zero page
08 >0000 mmu_ctrl       .byte       ?
09 >0001 io_ctrl        .byte       ?
10 >0002 reserved       .fill       6
11 >0008 mmu            .fill       8
12 >0010 printptr       .word       ?
13 .dsection   dp
14 .endv

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

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

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

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

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

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

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

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

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

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

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

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

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

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

use example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
16 keys                 .namespace

17 .section  code
18 .4008 a9 02          lda #$02                cmd         lda     #2
19 .400a 85 01          sta $01                             sta     io_ctrl
20 .400c 20 cc ff       jsr $ffcc                           jsr     kernel.Display.Reset
21 .400f a2 a1          ldx #$a1                            ldx     #<welcomemsg
22 .4011 a0 40          ldy #$40                            ldy     #>welcomemsg
23 .4013 20 80 40       jsr $4080                           jsr     printmsg
24 .4016 a9 89          lda #$89                            lda     #<event
25 .4018 85 f0          sta $f0                             sta     kernel.args.events+0
26 .401a a9 41          lda #$41                            lda     #>event
27 .401c 85 f1          sta $f1                             sta     kernel.args.events+1
28 .401e 20 0c ff       jsr $ff0c               _loop       jsr     kernel.Yield
29 .4021 20 00 ff       jsr $ff00                           jsr     kernel.NextEvent
30 .4024 b0 f8          bcs $401e                           bcs     _loop
31 .4026 ad 89 41       lda $4189                           lda     event.type
32 .4029 c9 08          cmp #$08                            cmp     #kernel.event.key.PRESSED
33 .402b f0 1e          beq $404b                           beq     _pressed
34 .402d c9 0a          cmp #$0a                            cmp     #kernel.event.key.RELEASED
35 .402f f0 16          beq $4047                           beq     _released
36 .4031 c9 04          cmp #$04                            cmp     #kernel.event.JOYSTICK
37 .4033 f0 02          beq $4037                           beq     _joy
38 .4035 80 e7          bra $401e                           bra     _loop
39 .4037 a2 00          ldx #$00                _joy        ldx     #0
40 .4039 ad 8c 41       lda $418c                           lda     event.joystick.joy0
41 .403c 20 66 40       jsr $4066                           jsr     print_hex
42 .403f ad 8d 41       lda $418d                           lda     event.joystick.joy1
43 .4042 20 66 40       jsr $4066                           jsr     print_hex
44 .4045 80 d7          bra $401e                           bra     _loop
45 .4047 a9 20          lda #$20                _released   lda     #' '
46 .4049 80 11          bra $405c                           bra     _show
47 .404b ac 8e 41       ldy $418e               _pressed    ldy     event.key.ascii
48 .404e c0 0d          cpy #$0d                            cpy     #13
49 .4050 f0 12          beq $4064                           beq     _done
50 .4052 a9 58          lda #$58                            lda     #'X'
51 .4054 2c 8f 41       bit $418f                           bit     event.key.flags
52 .4057 30 03          bmi $405c                           bmi     _show
53 .4059 ad 8e 41       lda $418e                           lda     event.key.ascii
54 .405c ac 8d 41       ldy $418d               _show       ldy     event.key.raw
55 .405f 99 f0 c0       sta $c0f0,y                         sta     $c0f0,y
56 .4062 80 ba          bra $401e                           bra     _loop
```

= MicroKernel Vectors

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

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

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

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

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

This was simplified to a static location to reduce overhead.

```
57 .4064 18            clc                  _done       clc                    ; Exits
58 .4065 60            rts                              rts

59 .4066 48            pha                  print_hex   pha
60 .4067 4a            lsr a                            lsr     a
61 .4068 4a            lsr a                            lsr     a
62 .4069 4a            lsr a                            lsr     a
63 .406a 4a            lsr a                            lsr     a
64 .406b 20 75 40      jsr $4075                        jsr     _digit
65 .406e 68            pla                              pla
66 .406f 29 0f         and #$0f                         and     #$0f
67 .4071 20 75 40      jsr $4075                        jsr     _digit
68 .4074 60            rts                              rts

69 .4075 5a            phy                  _digit      phy
70 .4076 a8            tay                              tay
71 .4077 b9 91 40      lda $4091,y          lda         digits,y
72 .407a 7a            ply                              ply
73 .407b 9d 31 c1      sta $c131,x                      sta     $c131,x
74 .407e e8            inx                              inx
75 .407f 60            rts                              rts

76 .4080 86 10         stx $10              printmsg    stx     printptr       ; This simple print routine is single
77 .4082 84 11         sty $11                          sty     printptr+1     ; purpose and replaces the .mkstr
78 .4084 a0 00         ldy #$00                         ldy     #$00           ; macro (part of display.asm).
79 .4086 b1 10         lda ($10),y          printloop   lda     (printptr),y   ;
80 .4088 f0 06         beq $4090                        beq     exitprint      ; Characters are stored, indexed by y,
81 .408a 99 00 c0      sta $c000,y                      sta     $c000,y        ; beginning at the hard-coded $c000
82 .408d c8            iny                              iny                    ; address. The text message below is
83 .408e 80 f6         bra $4086                        bra     printloop      ; nearly 3 full screen lines in length.
84 .4090 60            rts                  exitprint   rts

   >4091 30 31 32 33 34 35 36 37 38 39 61 62 63 64 65 66
85 digits      .text       "0123456789abcdef"

   >40a1 54 68 69 73 20 70 72 6f 67 72 61 6d 20 73 68 6f     >4121 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
   >40b1 77 73 20 74 68 65 20 68 65 6c 64 20 73 74 61 74     >4131 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
   >40c1 75 73 20 6f 66 20 6b 65 79 73 2e 20 50 72 65 73     >4141 a0 20 6d 65 74 61 3a 20 a1 20 20 a0 96 20 61 73
   >40d1 73 20 3c 45 4e 54 45 52 3e 20 74 6f 20 71 75 69     >4151 63 69 69 20 26 20 65 78 74 65 6e 64 65 64 20 6b
   >40e1 74 2e 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4161 65 79 73 2e 2e 2e 20 20 20 20 20 20 20 20 20 20
   >40f1 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4171 20 a0 20 66 31 2d 66 38 2f 31 32 20 a1 20 20 a0
   >4101 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20     >4181 20 6a 6f 79 73 20 a1 00
   >4111 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

86 welcomemsg  .text       "This program shows the held status of keys. Press <ENTER> to quit.              "
87             .text       "                                                                                "
88             .text       $a0," meta: ",$a1,"  ",$a0,$96," ascii & extended keys...          ",$a0," f1-f8/12 ",$a1
89             .text.      "  ",$a0," joys ",$a1,$0
90 .send
91 .endn
```

Interpreting Events

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

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

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

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

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

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