TinyCore 65c02 MicroKernel F256(K) Edition

Jessie Oberreuter
18-04-2023

The TinyCore 6502 MicroKernel for the Foenix F256/F256K line of computers is a powerful alternative to the typical BIOS style kernels that come with most 8-bit computers. This kernel offers the following features:

- An event-based programming model for real-time games and VMs.
- A network stack supporting multiple concurrent TCP and UDP sockets.
- Drivers for optional ESP8266 based wifi modules.
- Drivers for IEC drives.
- Drivers for Fat32 formatted SD Cards.
- Support for PS2 keyboards, Foenix keyboards, and CBM keyboards
- Support for PS2 scroll mice.
- Preemptive kernel multi-tasking (no need to yield).

There's a lot of love in here; I hope you will enjoy using it as much as I have enjoyed writing it!
— Gadget

Setup and Installation

Getting the Kernel

Kernel binaries may be obtained from either of the following repos:

- https://github.com/ghackwrench/F256_Jr_Kernel_DOS
- https://github.com/paulscottrobson/superbasic

The 'ghackwrench' repo contains the latest release. The 'paulscottrobson' repo contains the latest version verified to work with SuperBASIC.

Flashing the Kernel

The kernel consists of four 8k blocks which must be flashed into the last four blocks of the F256. Using the F256 Programmer Tool (Windows), the kernel needs to be flashed into Flash Blocks 0x3B–0x3F. Alternately, the SuperBASIC repo contains a script which uses a python based loader to install SuperBASIC along with the Kernel.

DIP Switches

The kernel assigns the DIP switches as follows:

1. Enable boot-from-RAM
2. Reserved (potentially for SNES gamepad support)
3. Enable SLIP based networking
4. Feather board installed (*e.g.* Huzzah 8266 WiFi)
5. SIDs are installed
6. F256: CBM keyboard installed; F256k: audio expansion installed.
7. ON: 640x480, OFF: 640x400 (*not yet implemented*)
8. Enable gamma color correction

When boot-from-RAM is enabled, the kernel will search the first 48k of RAM for pre-loaded programs before starting the first program on the expansion cartridge or the first program in flash.

Support Software

- The superbasic repo contains a powerful BASIC programming language which has been ported to run on the MicroKernel.
- The Kernel_DOS repo includes a simple DOS CLI which demonstrates most kernel functions.
- The Kernel_DOS repo also contains library and config files for compiling C programs with cc65 for use with the MicroKernel.

Memory Model

The F256 series machines minimally reserve addresses 0x00 and 0x01 for memory and I/O control. Programs may optionally enable additional hardware registers from 0x08–0x0f.

The MicroKernel uses 16 bytes of the Zero page, from 0xf0–0xff, for the `kernel.args` struct. Kernel calls do not expect their arguments to come in registers. Instead, all kernel arguments are passed by writing them to variables in `kernel.args`. The `A` register is used by all kernel calls. The `X` and `Y` registers, however, are always preserved. All kernel calls clear the carry on success and set the carry on failure. A few calls return a stream id in `A` as a convenience, but most of the time, `A` should just be considered undefined upon return from the kernel.

Outside of the above, user programs are free to use all addresses up to 0xC000 as they wish.

0xC000–0xDFFF is the I/O window. The I/O window can be disabled to reveal RAM underneath, but programs should refrain from doing so, as this RAM is used by the kernel. Programs are generally free to use the hardware as they see fit, but note that, by default, the kernel takes

ownership of the PS2 ports, the frame interrupt, and the RTC interrupt, and will take ownership of the serial port if either the SLIP or Feather DIP switches are set.

0xE000–0xFFFF contains the kernel itself. The kernel is considerably larger than 8kB, but it uses the F256 MMU hardware to keep its footprint in the user's memory map to a relative minimum.

The F256 machines support four concurrent memory maps. The kernel reserves map zero for itself, and map one for the Fat32 drivers; user programs are run from map three. Map two is potentially reserved for a hypervisor.

The kernel uses only two 8k blocks of RAM: block 6 (nominally at 0xC000), and block 7 (nominally at 0xE000). Everything else is free for use by user software.

Startup

On power-on or reset, the kernel initializes the hardware according to the DIP switches and then searches various memory regions for the first program to run. If DIP1 is on, it first checks RAM blocks 1–5 for a pre-loaded binary. After that, it searches the expansion RAM/ROM blocks, and, finally, the on-board flash blocks. The first block found with a valid header will be mapped into MMU 3 and started. The header must appear at the very start of the block and contains the following fields:

```
Byte  0    signature: 0xF2
Byte  1    signature: 0x56
Byte  2    the size of program in 8k blocks
Byte  3    the starting slot of the program (ie where to map it)
Bytes 4-5  the start address of the program
Bytes 6-9  reserved
Bytes 10-  the zero-terminated name of the program.
```

Programming

With a 65c02 installed, the F256 machines can generally be treated as simple 6502 machines. Programs have full access to the RAM from 0x0000–0xBFFF, and full access to I/O from 0xC000–0xDFFF. More adventurous programs can enable the MMU registers and bank additional memory into any 8k slot below 0xC000. Really adventurous programs are free to disable interrupts, map out the kernel, and take complete control over the machine. Programs wishing to use the kernel, however, should be sure to keep interrupts enabled as much as possible, call `NextEvent` often enough that the kernel doesn't run out of event objects, and refrain from trashing MMU LUT0, MMU LUT1, and the RAM between 0xC000 and 0xFFFF.

The kernel interfaces are described below using their symbolic names. The actual values and addresses must be obtained by including either `api.asm` or `api.h` in your project.

Events

The F256 machines were designed for games, and the 6502 TinyCore MicroKernel was designed to match.

Games generally run in a simple loop:

1. Update the screen
2. Read the controls
3. Update the game state
4. Goto 1

The kernel supports this mode of operation by replacing step 2 above (read the controls) with a generic kernel call: `NextEvent`. `NextEvent` gets the next I/O event from the kernel's queue and copies it into a user provided buffer.

Setup

Events are 8-byte packets. Each packet contains four common bytes, and up to four event-specific bytes. The user's buffer for these bytes may be placed anywhere, but since they are accessed frequently, the zero page is a good place.

Before a program can receive events, it must tell the kernel where events should be copied. It does this by writing the address of an 8-byte buffer into the kernel's `arg` block:

```
    .section zp
event   .dstruct  kernel.event.event_t
    .send

    .section code

init_events

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

Handling

For games, and other real-time applications, a program will typically contain a single "handle_events" routine:

```
handle_events

    ; Peek at the queue to see if anything is pending
    lda kernel.events.pending  ; Negated count
    bpl _done

    ; Get the next event.
    jsr kernel.NextEvent
```

```
    bcs _done

    ; Handle the event
    jsr _dispatch

    ; Continue until the queue is drained.
    bra handle_events

_done

    rts

_dispatch

    ; Get the event's type
    lda event.type

    ; Call the appropriate handler

    cmp #kernel.event.key.PRESSED
    beq _key_pressed

    cmp #kernel.event.mouse.DELTA
    beq _mouse_moved

    ...
    rts     ; Anything not handled can be ignored.
```

Other types of programs may eschew the use of a single central event handler, and instead work the queue only when waiting for events, and then only to handle the event types expected for the operation. The F256 cc65 kernel library does just this: when waiting for keypresses, it only handles key events; when waiting for data from a file, it only handles file.DATA/EOF/ERROR events and ignores all others. This approach is considerably simpler but also considerably less powerful.

Event types

Events belong to one of two categories: *Solicited Events* and *Unsolicited Events*.

Solicited Events are events which are generated in response to I/O requests (*e.g.* file Open/Read/Close calls). *Unsolicited Events* are generated by external devices such as keyboards, joysticks, and mice. *Unsolicited Events* are also generated whenever packets are received from the network.

Solicited Events

The *solicited events* are described in the *Kernel Calls* section below. The documentation for each kernel call that queues an event includes a description of the possible events.

Unsolicited Events

`event.key.PRESSED`
`event.key.RELEASED`

These events occur whenever a key is pressed or released:

- `event.key.keyboard` contains the id of the keyboard.
- `event.key.raw` contains the raw key-code (see `kernel/keys.asm`).
- `event.key.flags` is negative if this is a meta (non-ascii) key.
- `event.key.ascii` contains the ascii interpretation if available.

`event.mouse.DELTA`

This event is generated every time the mouse is moved or a button changes state.

- `event.mouse.delta.x` contains the x delta.
- `event.mouse.delta.y` contains the y delta.
- `event.mouse.delta.z` contains the z (scroll) delta.
- `event.mouse.delta.buttons` contains the button bits.

The buttons are encoded as follows:

- Bit 0 (1) is the inner-most button
- Bit 1 (2) is the middle button
- Bit 2 (4) is the outer-most button

The bits are set when the button is pressed and cleared when the button is released.
To change the "handedness" of the mouse, simply place it in whichever hand you prefer, and double-click!

`event.mouse.CLICKS`

This event reports mouse click (press-and-release) events. Whenever a mouse button is pressed, the kernel starts a 500ms timer and counts the number of times each button is both pressed and released. At the end of the 500ms, the counts are reported:

- `event.mouse.clicks.inner` contains the count of inner clicks.
- `event.mouse.clicks.middle` contains the count of middle clicks.
- `event.mouse.clicks.outer` contains the count of outer clicks.

A report of all zeros indicates that a press-and-hold is in progress; programs should consult the most recent event.`mouse.DELTA` event to determine which button(s) are being held.

`event.JOYSTICK+`

This event returns the state of the “buttons” for each of the two joysticks whenever either’s state changes:

- `event.joystick.joy0` contains the bits for Joystick 0.
- `event.joystick.joy1` contains the bits for Joystick 1.

The bits are set when the associated switch is pressed, and clear when released.

`event.TCP`
`event.UDP`
`event.ICMP`

These events are generated whenever a network packet of the given type is received. For TCP and UDP packets, a program can use the `kernel.Net.Match` call to see if the packet matches an open socket. Non-matching UDP packets can be ignored; network aware programs should respond to unmatched TCP packets by calling `kernel.Net.TCP. Reject`. TCP and UDP payloads may be read by calling `kernel.Net.TCP.Recv` and `kernel.Net.UDP.Recv` respectively. For ICMP packets, programs can get the raw data by calling `kernel.ReadData`.

`event.clock.TICK`

This event is generated every second for general time reference.

Kernel Calls

Generic Calls

NextEvent

Copies the next event from the kernel’s event queue into a user provided event buffer.

Input

- `kernel.args.events` points to an 8-byte buffer.

Output

- Carry cleared on success.
- Carry set if no events are pending.

Effect

- If there is an event in the queue, it is copied into the provided buffer.

Notes

- `kernel.args.events` is a reserved field in the argument block; nothing else uses this space, so you need only initialize this pointer once on startup.
- `kernel.args.pending` contains the count of pending events (negated for ease of testing with BIT). You can save considerable CPU time by testing `kernel.args.pending` and only calling `NextEvent` if it is non-zero.
- The kernel reports almost everything through events. Consider keeping the event buffer in the zeropage for efficient access.

ReadData

Copies data from the kernel buffer associated with the current event into the user's address space.

Input

- `kernel.args.buf` points to a user buffer.
- `kernel.args.buflen` contains the number of bytes to copy (0=256).

Output

- Carry cleared.

Effect

- The contents of the current event's primary buffer are copied into the provided user buffer.
- If the event doesn't contain a buffer, zeros are copied.

ReadExt

Copies extended data from the kernel buffer associated with the current event into the user's address space.

Input

- `kernel.args.buf` points to a user buffer.
- `kernel.args.buflen` contains the number of bytes to copy (0=256).

Output

- Carry cleared.

Effect

- The contents of the current event's primary buffer are copied into the provided user buffer.
- If the event doesn't contain an extended data buffer, zeros are copied.

Notes

- Events with extended data are relatively rare. Typical examples include file meta-information and bytes 256...511 of a 512 byte raw-sector read.

Yield

Yields the CPU to the kernel. This is typically used to expedite the processing of network packets. It is never required.

Input

- none

Output

- none

Effects

- none

RunBlock

Transfers execution to the program found in the given memory block.

Input

- `kernel.args.run.block_id` contains the number of the block to execute.

Output

- On success, the call doesn't return.
- Carry set on error (block doesn't contain a program).

RunNamed

Transfers execution to the first program found with the given name.

Input

- `kernel.args.buf` points to a buffer containing the name of the program to run.
- `kernel.args.buflen` contains the length of the name.

Output

- On success, the call doesn't return.
- Carry set on error (a program with the provided name was not found).

Notes

- The name match is case-insensitive.

FileSystem Calls

FileSystem.MkFS

Creates a new filesystem on the given device.

Input

- `kernel.args.fs.mkfs.drive` contains the device number (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.fs.mkfs.label` points to a buffer containing the new drive label.
- `kernel.args.fs.mkfs.label_len` contains the length of the label buffer (0=0).
- `kernel.args.fs.mkfs.cookie` contains a user-provided cookie for matching against completion events.

Output

- Carry cleared on success.
- Carry set on error (device doesn't exist, kernel is out of streams).

Events

- The kernel will queue an `event.fs.CREATED` event on success.
- The kernel will queue an `event.fs.ERROR` event on failure.

Notes

- This is presently an atomic, blocking call on both IEC and Fat32, and it can take a *long* time to complete. While running, your program will not be able to work the event queue, so pretty much the whole system will grind to a halt. Until this changes, it's best if interactive programs and operating systems avoid this call.

File Calls

Open

Opens a file for read, append, or create/overwrite. The file should not be concurrently opened in another stream.

Input

- `kernel.args.file.open.drive` contains the drive ID (0 = SD card, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.file.open.fname` points to the name of the file.
- `kernel.args.file.open.fname_len` contains the length of the name.
- `kernel.args.file.open.mode` contains the access mode (0 = read, 1 = write, 2 = append).
- `kernel.args.file.open.cookie` contains an optional, user provided value which will be returned in subsequent events related to this file.

Output

- Carry cleared on success, and `A` contains the stream ID for the file.
- Carry set if the drive isn't found, or if the kernel lacks sufficient resources to complete the call.

Events

- On a successful open/create/append, the kernel will queue a `file.OPENED` event.
- For read or append, if the file does not exist, the kernel will queue a `file.NOT_FOUND` event.
- File events contain the stream id (in `event.file.stream`) and the user supplied cookie (in `event.file.cookie`).

Notes

- The kernel supports a maximum of 20 concurrently opened files (including directories and rename/delete operations) across all devices.
- The IEC driver supports a maximum of 8 concurrently opened files (not counting directories and rename/delete operations) per device.
- Fat32 preserves case when creating files, and uses case-insensitive matching when opening files.
- IEC devices vary in their handling of case. Users will just need to live with this.

- Unlike other kernel calls, most file operations are blocking. In the case of IEC, this is due to the fact that an interrupt driven interface was not available at the time of writing (it is now, so IEC can be improved). In the case of Fat32, the fat32.s package we're using contains its own, blocking SPI stack (which may be replaced in the future). Fortunately, most operations are fast enough that events won't be dropped.

- The kernel does not lock files and does not check to see if a file is already in use. Should you attempt to concurrently open the file in two or more streams, the resulting behavior is entirely up to the device (IEC) or the file-system driver (fat32.s). The above statement that the file should not be concurrently opened should be treated as a strong warning.

- *Open for append is not yet implemented*

File.Read

Reads bytes from a file opened for reading.

Input

- `kernel.args.file.read.stream` contains the stream ID of the file (the stream is returned in the `file.OPENED` event and from the `File.Open` call itself).

- `kernel.args.file.read.buflen` contains the requested number of bytes to read.

Output

- Carry cleared on success.

- Carry set on error (file is not opened for reading, file is in the EOF state, the kernel is out of event objects).

Events

- On a successful read, the kernel will queue an `event.file.DATA` event (see `ReadData`). `event.file.data.requested` contains the number of bytes requested; `event.file.data.read` contains the number of bytes actually read.

- On EOF, the kernel will queue an `event.file.EOF` event.

- On error, the kernel will queue an `event.file.ERROR` event.

- In all cases, the event will also contain the stream id in `event.file.stream` and the the user's cookie in `event.file.cookie`.

Notes

- As with POSIX, the kernel may return fewer bytes than requested. This is not an error, and does not imply that the file has reached EOF (the kernel will queue an `event.file.EOF` on EOF). When this happens, the user is expected to simply call `File.Read` again to get more data.
- IEC devices don't report file-not-found until a read is attempted. To work around this issue, when talking to an IEC device, File.Open performs a one-byte read during the open. This single byte is later returned in the file's first `event.file.DATA` response (it does not attempt to merge this byte into the next full read).
- To reduce the risk of losing events while reading data from IEC devices, IEC read transfers are artificially limited to 64 bytes at a time. This limit may be lifted once we have interrupt driven IEC transfers.

File.Write

Writes bytes to a file opened for writing.

Input

- `kernel.args.file.write.stream` contains the stream ID of the file (the stream is returned in the `file.OPENED` event and from the `File.Open` call itself).
- `kernel.args.file.write.buf` points to the buffer to write.
- `kernel.args.file.write.buflen` contains the requested number of bytes to write.

Output

- Carry cleared on success.
- Carry set on error (file is not opened for writing, the kernel is out of event objects).

Events

- On a successful write, the kernel will queue an `event.file.WROTE` event. `event.file.wrote.requested` contains the number of bytes requested; `event.file.wrote.wrote` contains the number of bytes actually written.
- On error, the kernel will queue an `event.file.ERROR` event.
- In all cases, the event will also contain the stream id in `event.file.stream` and the user's cookie in `event.file.cookie`.

Notes

- As with POSIX, the kernel may write fewer bytes than requested. This is not an error. When it happens, the user is expected to simply call `File.Write` again to write more data.
- To reduce the risk of losing events while reading data from IEC devices, IEC write transfers are artificially limited to 64 bytes at a time. This limit may be lifted once we have interrupt driven IEC transfers.

File.Seek

Changes the position of the next read or write within a file.

Input

- `kernel.args.file.seek.stream` contains the stream ID of the file (the stream is returned in the `file.OPENED` event and from the `File.Open` call itself).
- `kernel.args.file.seek.offset` contains the 32-bit file offset.

Output

- Carry cleared on success.
- Carry set on error (the kernel is out of event objects).

Events

- On a successful write, the kernel will queue an `event.file.SEEK` event.
- On error, the kernel will queue an `event.file.ERROR` event.
- In all cases, the event will also contain the stream id in `event.file.stream` and the the user's cookie in `event.file.cookie`.

Events

- This is presently implemented only for fat32. It is potentially available on the SD2IEC.

File.Close

Closes an open file.

Input

- `kernel.args.file.close.stream` contains the stream ID of the file.

Output

- Carry cleared on success.
- Carry set on error (kernel is out of event objects).

Events

- The kernel will always queue an `event.file.CLOSED`, even if an I/O error occurs during the close. The event will contain the stream id in `event.file.stream` and the user's cookie in `event.file.cookie`.

Notes

- The kernel will always queue an `event.file.CLOSED`, even if an I/O error occurs during the close.
- Upon a successful call to `File.Close`, no further operations should be attempted on the given stream (the stream ID will be returned to the kernel's free pool for use by subsequent file operations).

File.Rename

Renames a file. The file should not be in use.

Input

- `kernel.args.file.rename.drive` contains the drive ID (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.file.rename.cookie` contains a user supplied cookie for matching the completed event.
- `kernel.args.file.rename.old` points to a file path containing the name of the file to rename.
- `kernel.args.file.rename.old_len` contains the length of the path above.
- `kernel.args.file.rename.new` points to the new name for the file (*NOT* a new *path*!)
- `kernel.args.file.rename.new_len` contains the length of the new name above.

Output

- Carry cleared on success.
- Carry set on error (device not found; kernel out of events).

Events

- On successful completion, the kernel will queue a `file.RENAMED` event.
- On failure, the kernel will queue a `file.ERROR` event.
- In either case, `event.file.cookie` will contain the cookie supplied above.

Notes

- Rename semantics are up to the device (in the case of IEC) or the fat32.s driver (in the case of Fat32). The kernel doesn't even look at the arguments to see if they are sane. This means you may or may not be able to, say, change the case of letters in a filename with a single rename call; you may need to rename to a temp name, and then rename to the case-corrected name.
- Similarly, the kernel doesn't actually check if the file is in-use or not. Whether this matters is up to the device or file-system driver; treat the above statement that the file should not be in use as a strong recommendation.
- Rename is *NOT* a 'move'. You can only rename a file in place. If you want to move it, you will need to copy and then delete.

File.Delete

Deletes a file. The file should not be in use.

Input

- `kernel.args.file.delete.drive` contains the drive ID (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.file.delete.cookie` contains a user supplied cookie for matching the completed event.
- `kernel.args.file.delete.fname` points to a file path containing the name of the file to delete.
- `kernel.args.file.delete.fname_len` contains the length of the path above.

Output

- Carry cleared on success.
- Carry set on error (device not found; kernel out of events).

Events

- On successful completion, the kernel will queue a `file.DELETED` event.
- On failure, the kernel will queue a `file.ERROR` event.
- In either case, `event.file.cookie` will contain the cookie supplied above.

Notes

- Case matching is up to the device (IEC) or the file-system driver (fat32.s).
- Delete semantics are up to the device (in the case of IEC) or the fat32.s driver (in the case of Fat32). The kernel doesn't even look at the path to see if it is sane.
- Similarly, the kernel doesn't actually check if the file is in-use or not. Whether this matters is up to the device or file-system driver; treat the above statement that the file should not be in use as a strong recommendation.

Directory Calls

Directory.Open

Opens a directory for reading.

Input

- `kernel.args.directory.open.drive` contains the device id (0 = SD, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.directory.open.path` points to a buffer containing the path.
- `kernel.args.directory.open.lan_len` contains the length of the path above. May be zero for the root directory.
- `kernel.args.directory.open.cookie` contains a user-supplied cookie for matching the completed event.

Output

- Carry cleared on success; `A` contains the stream id.
- Carry set on error (device not found, kernel out of event or stream objects).

Events

- On successful completion, the kernel will queue an `event.directory.OPENED` event.
- On error, the kernel will queue an `event.directory.ERROR` event.
- In either case, `event.directory.cookie` will contain the above cookie.

Notes

- The IEC protocol only supports reading one directory per device at a time. The kernel does not, however, prevent you from trying. Consider yourself warned.

Directory.Read

Reads the next directory element (volume name entry, file entry, bytes-free entry).

Input

- `kernel.args.directory.stream` contains the stream id.

Output

- Carry cleared on success.
- Carry set on error (EOF has occurred or the kernel is out of event objects).

Events

- The first read will generally queue an `event.directory.VOLUME` event. `event.directory.volume.len` will contain the length of the volume name. Call `ReadData` to retrieve the volume name.
- Subsequent reads will generally queue an `event.directory.FILE` event. `event.directory.file.len` will contain the length of the filename. `ReadData` will retrieve the file name; `ReadExt` will retrieve the meta-data (presently just the sector count).
- The last read before EOF will generally queue an `event.directory.FREE` event. `event.directory.free.free` will contain the number of free sectors on the device.
- The final read will queue an `event.directory.EOF` event.
- Should an error occur while reading the directory, the kernel will queue an `event.directory.ERROR` event.

Notes

- The IEC protocol does not support multiple concurrent directory reads.
- Attempting to open files while reading an IEC directory have been known to result in directory read errors (cc65 errata).
- SD2IEC devices cap the sectors-free report at 65535 sectors.

Directory.Close

Input

- `kernel.args.directory.stream` contains the stream id.

Output

- Carry cleared on success.
- Carry set on error (kernel is out of event objects).

Events

- The kernel will always queue an `event.directory.CLOSED` event, even if an error should occur.

Notes

- Do not attempt to make further calls against the same stream id after calling `Directory.Close`—the stream will be returned to the kernel for allocation to subsequent file operations.

Directory.MkDir

Creates a sub-directory.

Input

- `kernel.args.directory.mkdir.drive` contains the device id (0 = SD, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.directory.mkdir.path` points to a buffer containing the path.
- `kernel.args.directory.mkdir.path_len` contains the length of the path above. May be zero for the root directory.
- `kernel.args.directory.mkdir.cookie` contains a user-supplied cookie for matching the completed event.

Output

- Carry cleared on success.
- Carry set on error (device not found, kernel out of event or stream objects).

Events

- On successful completion, the kernel will queue an `event.directory.CREATED` event.
- On error, the kernel will queue an `event.directory.ERROR` event.
- In either case, `event.directory.cookie` will contain the above cookie.

Notes

- Not supported by all IEC devices.
- The IEC protocol only supports one command (mkdir, rmdir, rename, delete) per device at a time. The kernel does not, however, prevent you from trying. Consider yourself warned.

Directory.RmDir

Deletes a sub-directory.

Input

- `kernel.args.directory.rmdir.drive` contains the device id (0 = SD, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.directory.rmdir.path` points to a buffer containing the path.
- `kernel.args.directory.rmdir.path_len` contains the length of the path above. May be zero for the root directory.
- `kernel.args.directory.rmdir.cookie` contains a user-supplied cookie for matching the completed event.

Output

- Carry cleared on success.
- Carry set on error (device not found, kernel out of event or stream objects).

Events

- On successful completion, the kernel will queue an `event.directory.CREATED` event.
- On error, the kernel will queue an `event.directory.ERROR` event.
- In either case, `event.directory.cookie` will contain the above cookie.

Notes

- Not supported by all IEC devices.
- The kernel does not ensure that the directory is empty – that is left to the device (IEC) or driver (fat32).
- The IEC protocol only supports one command (mkdir, rmdir, rename, delete) per device at a time. The kernel does not, however, prevent you from trying. Consider yourself warned.

Network Calls—Generic

Network.Match

Determines if the packet in the current event belongs to the provided socket.

Input

- `kernel.args.net.socket` points to either a TCP or UDP socket.

Output

- Carry cleared if the socket matches the packet.
- Carry set if the socket does not match the packet.

Network Calls—UDP

Network.UDP.Init

Initializes a UDP socket in the user's address space.

Input

- `kernel.args.net.socket` points to a 32 byte UDP socket structure.
- `kernel.args.net.dest_ip` contains the destination address.
- `kernel.args.net.src_port` contains the desired source port.
- `kernel.args.net.dest_port` contains the desired destination port.

Output

- Carry clear (always succeeds).

Events

- None

Notes

- Opening a UDP socket *does not* create a packet filter for the particular socket. Instead, ALL UDP packets received by the kernel are queued as events; it is up to the user program to accept or ignore each in turn.

Network.UDP.Send

Writes data to a UDP socket.

Input

- `kernel.args.net.socket` points to a 32 byte UDP socket structure.
- `kernel.args.net.buf` points to the send buffer.
- `kernel.args.net.buf_len` contains the length of the buffer (0 = 256, but see the notes section)

Output

- Carry cleared on success.
- Carry set on error (kernel is out of packet buffers).

Events

- None

Notes

- By design, the kernel limits all network packets to 256 bytes. This means the maximum payload for a single UDP packet is 228 bytes.
- **Bug**: the kernel doesn't currently stop you from trying to send more than 228 bytes. Attempts to do so will result in corrupt packets.

Network.UDP.Recv

Reads the UDP payload from an `event.network.UDP` event.

Input

- `kernel.args.net.buf` points to the receive buffer.
- `kernel.args.net.buflen` contains the size of the receive buffer.

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of bytes copied from the event (0 = 0).
- Carry set on failure (event is not a network.UDP event).

Notes

- The full packet may be read into the user's address space by calling `kernel.ReadData`.

Network Calls—TCP

Network.TCP.Open

Initializes a TCP socket in the user's address space.

Input

- `kernel.args.net.socket` points to a 256 byte TCP socket structure (includes a re-transmission queue).
- `kernel.args.net.dest_ip` contains the destination address.
- `kernel.args.net.dest_port` contains the desired destination port.

Output

- Carry clear (always succeeds).

Events

- None

Notes

- Opening a TCP socket *does not* create a packet filter for the particular socket. Instead, ALL TCP packets received by the kernel are queued as events; it is up to the user program to accept or reject each in turn; see `Network.Match`.
- The kernel does not prevent you from binding multiple sockets to the same endpoint, but you shouldn't.

Network.TCP.Accept

Accepts a new connection from the outside world.

Input

- The current event is a TCP SYN packet.
- `kernel.args.net.socket` points to a 256 byte TCP socket structure which will represent the accepted connection.

Output

- Carry clear: socket initialized
- Carry set: event is not a TCP SYN packet.

Notes

- Accepting a TCP socket *does not* create a packet filter for the particular socket. Instead, ALL TCP packets received by the kernel are queued as events; it is up to the user program to accept or reject each in turn; see `Network.Match`.
- The kernel does not prevent you from binding multiple sockets to the same endpoint, but you shouldn't.
- As with all things IP, the kernel will try to reply; it will not, however, retry on its own.

Network.TCP.Reject

Rejects a connection from the outside world. May also be used to forcibly abort an existing connection.

Input

- The current event is a TCP event.

Output

- Carry clear if the reset was sent.
- Carry set on error (out of network buffers).

Notes

- TCP applications should reject all TCP packets that don't match an existing socket.
- The kernel does not prevent you from rejecting valid packets.

Network.TCP.Send

Writes data to a TCP socket.

Input

- `kernel.args.net.socket` points to the socket.
- `kernel.args.net.buf` points to the send buffer.
- `kernel.args.net.buf_len` contains the length of the buffer (0 = 0; may be used to force re-transmission).

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of byte accepted.
- Carry set on error (socket not yet open, user has closed this side of the socket, kernel is out of packet buffers).

Events

- None

Notes

- The socket presently contains a 128 byte transmit queue. This queue is forwarded by both `Network.TCP.Send` and by `Network.TCP.Recv`. The remote host must ACK the bytes in the transmit queue before more bytes become available in the queue.

Network.TCP.Recv

- Reads the TCP payload from an `event.network.TCP` event.
- ACKs valid packets from the remote host.
- Maintains the state of the socket.

Input

- `kernel.args.net.socket` points to the socket.
- `kernel.args.net.buf` points to the receive buffer.
- `kernel.args.net.buflen` contains the size of the receive buffer.

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of bytes copied from the event (0 = 0). `A` contains the socket state.
- Carry set on failure (event is not a network.TCP event).

Notes

- The full packet may be read into the user's address space by calling `kernel.ReadData`.

Network.TCP.Close

Tells the remote host that no more data will be sent from this side of the socket. The remote host is, however, free to continue sending until it issues a close.

Input

- `kernel.args.net.socket` points to the socket.

Output

- Carry cleared on success.
- Carry set on failure (kernel is out of buffers).

Display Calls

Display.Reset

Resets the display resolution and colors and disables the mouse and cursor. Does NOT reset the font.

Input

- None

Output

- None

Notes

- In the future, the kernel should include support for re-initializing the default fonts.

Display.GetSize

Returns the size of the current text display.

Input

- None

Output

- `kernel.args.display.x` contains the horizontal size.
- `kernel.args.display.y` contains the vertical size.

Display.DrawRow

Copies the user provided text buffer to the screen (from left to right) starting at the provided coordinates and using the provided color buffer.

Input

- `kernel.args.display.x` contains the starting x coordinate.
- `kernel.args.display.y` contains the starting y coordinate.
- `kernel.args.display.text` points to the text data.
- `kernel.args.display.color` points to the color data.
- `kernel.args.buflen` contains the length of the buffer.

Output

- Carry cleared on success.
- Carry set on error (x/y outside of the screen)

Notes

- Probably isn't yet checking the coordinate bounds or clipping.

Display.DrawColumn

Copies the user provided text buffer to the screen (from top to bottom) starting at the provided coordinates and using the provided color buffer.

Input

- `kernel.args.display.x` contains the starting x coordinate.
- `kernel.args.display.y` contains the starting y coordinate.
- `kernel.args.display.text` points to the text data.
- `kernel.args.display.color` points to the color data.
- `kernel.args.buflen` contains the length of the buffer.

Output

- Carry cleared on success.
- Carry set on error (x/y outside of the screen)

Notes

- *Not yet implemented.*