Mode 10 — The Variable Text Mode

How the text modes became data, and how to program them.

Until now every text mode on the SD-8516 was a separate piece of code. Mode 1 drew 40×25 with 8×8 cells, mode 2 drew 80×25, mode 6 drew 80×25 with 9×16 VGA cells, and each one existed because a different set of numbers had been compiled into a different renderer. Adding a mode meant adding a mode.

That is no longer true. A text mode is now described by a small block of registers in RAM, and the renderer reads them. Modes 1, 2 and 10 are the same renderer with different numbers in front of it. **Mode 10 is the mode with no numbers of its own** — it draws whatever the registers currently say, so you can define your own geometry, place the screen anywhere in memory, and point at your own character set.

Background: what changed in mode 1

The interesting part of this change is how little of it was new.

The KERNAL had *already* been keeping the text geometry in RAM for a long time. `int10_set_video_mode` (INT `$10`, AH=`$40`) writes `VIDEO_COLUMNS`, `VIDEO_ROWS`, `VIDEO_CHAR_WIDTH`, `VIDEO_CHAR_HEIGHT`, `VIDEO_TEXT_BASE` and `VIDEO_COLOR_BASE` every time you change mode, and `term.sda` reads them constantly — `VIDEO_COLUMNS` alone is referenced dozens of times. The screen editor, the cursor logic and the scroller were all descriptor-driven already.

The renderer was the one component that ignored all of it. It contained:

```
const int cols = 40, rows = 25;              // hardcoded geometry
u8  charCode  = RAM[VIDEO_TEXT  + charIndex]; // hardcoded $01F000
u8  colorByte = RAM[VIDEO_COLOR + charIndex]; // hardcoded $01F800
u32 charRomBase = CHAR_ROM + charCode * 8;    // hardcoded $01E000, 8 rows
```

So mode 1 was not really “a mode”. It was the descriptor the KERNAL was already maintaining, plus a renderer that refused to read it.

Making mode 1 read the registers instead of its own constants is the whole change. Two fields had to be added, because they were the two the descriptor had never had a place to record:

- **where the glyphs live** — previously the fixed address `$01E000`
- **how wide a text row is in memory** — previously assumed equal to the

column count

With those in place, mode 1 renders identically (this was verified pixel-by-pixel, see verification), mode 2 becomes the same renderer with `VIDEO_COLUMNS=80`, and mode 10 becomes possible at no extra cost.

The cost on the KERNAL side was **four instructions**.

The text mode descriptor

Register	Address	Size	Meaning
VIDEO_MODE	$01EF00	1	Current mode number. The host reads the low 4 bits.
VIDEO_COLUMNS	$01EF01	1	Characters per row (visible)
VIDEO_ROWS	$01EF02	1	Rows on screen
VIDEO_CHAR_WIDTH	$01EF03	1	Character cell width in pixels
VIDEO_CHAR_HEIGHT	$01EF04	1	Character cell height in pixels
VIDEO_FONT_BASE	$01EEF6	3	**NEW.** Where the font is (see VIDEO_FONT_KIND)
VIDEO_FONT_KIND	$01EF7F	1	**NEW.** 0 = glyph bitmaps at that address, 1 = font *filename* at that address
VIDEO_ROW_STRIDE	$01EEF9	2	**NEW.** Bytes from one text row to the next
VIDEO_TEXT_BASE	$01EF63	3	Address of the character plane
VIDEO_COLOR_BASE	$01EF66	3	Address of the colour plane

The screen resolution is not a register. It is derived:

```
pixel width  = VIDEO_COLUMNS * VIDEO_CHAR_WIDTH
pixel height = VIDEO_ROWS    * VIDEO_CHAR_HEIGHT
```

Why stride is separate from columns

VIDEO_ROW_STRIDE is normally equal to VIDEO_COLUMNS — every preset mode sets it that way when you select it — and if you set it smaller it is silently raised to the column count. **Mode 10 is the exception: it leaves the stride alone**, so a wider buffer survives being selected. Making it **larger** is the interesting case: it gives you a text buffer wider than the screen, with the extra columns sitting off-stage to the right. Advancing VIDEO_TEXT_BASE by one byte then scrolls the whole display sideways by one character, with new content already in place. The same trick vertically — add VIDEO_ROW_STRIDE to the base — scrolls up by one row without moving any screen memory.

Why the font base had to be relocatable

The old character ROM lived at $01E000 and was exactly 2048 bytes: 256 glyphs of 8 rows. That is not a spare address, it is a **ceiling**. A 256-glyph 8×16 font needs 4096 bytes, which would run from $01E000 straight into the interrupt vector table at $01E900.

So taller cells were not merely inconvenient under the old scheme, they were impossible. VIDEO_FONT_BASE removes the ceiling: the font can live wherever there is room, including in your own program's memory.

Selecting a preset mode

Unchanged, and still the normal way to get a classic screen:

```
    LDA $4001       ; AH=$40 set video mode, AL=$01
    INT $10
```

- mode 1 — 40×25, 8×8 cells, 320×200
- mode 2 — 80×25, 8×8 cells, 640×200
- mode 6 — 80×25, 9×16 cells, 720×400 (see three kinds of mode)

From Stellar BASIC, `MODE 1`, `MODE 2`, `MODE 6` and `MODE 10` do the same thing (`MODE 40` and `MODE 80` are accepted as aliases for 1 and 2).

These presets write the whole descriptor for you. Everything below is about what happens when you write it yourself.

Programming mode 10

Mode 10 writes no preset — geometry, both planes, the row stride and the font are left exactly as the program set them. Selecting it clears the screen, and that is all it does. The sequence is always the same:

1. fill in the descriptor registers
2. set the mode to 10

```
; ---- an 80x50 screen, 640x400 pixels, planes in bank 2 ----

    LDAL #80
    STAL [@VIDEO_COLUMNS]
    LDAL #50
    STAL [@VIDEO_ROWS]
    LDAL #8
    STAL [@VIDEO_CHAR_WIDTH]
    LDAL #8
    STAL [@VIDEO_CHAR_HEIGHT]

    LDT  #80                    ; stride = columns (no off-screen margin)
    STT  [@VIDEO_ROW_STRIDE]

    LDELM $020000               ; character plane
    STELM [@VIDEO_TEXT_BASE]
    LDELM $021000               ; colour plane
    STELM [@VIDEO_COLOR_BASE]

    ; VIDEO_FONT_BASE is not written here, so whatever font is currently
    ; selected stays in effect -- after boot that is the system font

    LDA $400A                   ; AH=$40 set mode, AL=$0A (10)
    INT $10
```

The screen is 4000 characters, so the two planes need 4000 bytes each; putting them at `$020000` and `$021000` keeps them clear of the 4K video window at `$01F000`, which is only large enough for the classic modes.

Worked examples

Goal	Columns	Rows	Cell	Result
Classic 40 column	40	25	8×8	320×200
Classic 80 column	80	25	8×8	640×200
80×50 “small text”	80	50	8×8	640×400
Wide terminal	132	43	8×8	1056×344
Chunky/large text	40	25	16×16	640×400

All of these have been run. Nothing in the renderer is specific to any of them.

Reading the descriptor back

INT `$10` already provides accessors, and they remain the polite way to find the screen:

AH	Returns
$41	current video mode
$42	rows
$43	columns
$44	pointer to the character plane
$45	pointer to the colour plane
$46	is this a text mode?
$47	ELM = `VIDEO_FONT_BASE`, AL = `VIDEO_FONT_KIND` (clobbers A)

Character and colour data

Both planes are indexed the same way:

```
offset = row * VIDEO_ROW_STRIDE + column
```

The character plane holds one byte per cell: the glyph number, 0-255.

The colour plane holds one attribute byte per cell, packed **BBBBFFFF**:

- low nibble = foreground colour index (0-15)
- high nibble = background colour index (0-15)

Both indexes select from the 16-entry palette at `VIDEO_PALETTE` (`$01D700`), offset by the palette number for the current mode:

```
entry = VIDEO_PALETTE + (colour_mode * 64) + (index * 4)
```

Each entry is 4 bytes; the first three are red, green and blue.

Glyph data format

Glyphs are bitmaps, one bit per pixel, most significant bit leftmost.

```
bytes per glyph row = (VIDEO_CHAR_WIDTH + 7) / 8      (rounded up)
bytes per glyph     = VIDEO_CHAR_HEIGHT * bytes per glyph row
address of glyph N  = VIDEO_FONT_BASE + N * bytes per glyph
```

For the standard 8×8 font that is 1 byte per row and 8 bytes per glyph, which is exactly the old layout — existing fonts need no conversion. A 9×16 cell needs 2 bytes per row and 32 bytes per glyph, and the renderer reads the top 9 bits of those 16 and ignores the rest.

The font is always 256 glyphs, so a font occupies `256 * VIDEO_CHAR_HEIGHT * bytes-per-row` bytes.

Using your own font

Point `VIDEO_FONT_BASE` at your data and it is used immediately — there is no copy step and no "install font" call:

```
    LDELM @my_font
    STELM [@VIDEO_FONT_BASE]
```

The system font is data too. It lives in the KERNAL as `petscii_data` and `VIDEO_FONT_BASE` is pointed at it during boot, so it is redefinable in exactly the same way as yours.

The font base is sticky

`VIDEO_FONT_BASE` is written **exactly once**, during boot, where it is pointed at `petscii_data`. No mode change ever writes it — not `int10_set_video_mode`, not any of the per-mode setup routines. Two things follow from that:

- Leaving it alone gives you **whatever font was last selected**, not

specifically the system font. Immediately after boot those are the same

```
  thing, which is why the example above works.
* A font you install **stays installed across mode changes**. Set a custom
  font in mode 10, switch to mode 1, and mode 1 will draw with your font.
```

If you intend to put the system font back afterwards, **save the old pointer before you overwrite it**. There is currently no call that will tell you what it was: AH=$47 reports a fixed per-mode address rather than `VIDEO_FONT_BASE`.

```
    LDELM [@VIDEO_FONT_BASE]    ; save the current font
    STELM [@saved_font]
```

```
    LDELM @my_font              ; install ours
    STELM [@VIDEO_FONT_BASE]
    ; ...

    LDELM [@saved_font]         ; put it back
    STELM [@VIDEO_FONT_BASE]
```

To change a single character rather than the whole set, INT $10 AH=$48 (`font_set_char`) writes 8 bytes of glyph data, and AH=$49 (`glyph_pack`) builds a glyph from an 8×8 grid of characters, which is the convenient one to call from BASIC.

Using a font file instead of a bitmap

A text mode's glyphs can come from RAM or from a font file on the host, and `VIDEO_FONT_KIND` says which. `VIDEO_FONT_BASE` is a pointer either way — only the thing it points *at* changes:

VIDEO_FONT_KIND	**VIDEO_FONT_BASE points at**
0	glyph bitmaps, in the layout above
1	a NUL-terminated filename

```
    ; ---- 80x25 with 9x16 cells, drawn with a real VGA typeface ----
    LDAL #80
    STAL [@VIDEO_COLUMNS]
    LDAL #25
    STAL [@VIDEO_ROWS]
    LDAL #9
    STAL [@VIDEO_CHAR_WIDTH]
    LDAL #16
    STAL [@VIDEO_CHAR_HEIGHT]

    LDELM @font_name            ; -> "PxPlus_IBM_VGA_9x16.ttf", 0
    STELM [@VIDEO_FONT_BASE]
    LDAL #1                     ; 1 = the pointer is a filename
    STAL [@VIDEO_FONT_KIND]

    LDA $400A
    INT $10
    ...

font_name:
    .bytes "PxPlus_IBM_VGA_9x16.ttf", 0
```

Notes on file fonts:

- The **point size follows `VIDEO_CHAR_HEIGHT`**. The cell size still

governs layout; glyphs are clipped or padded into the cell, so the two do

```
  not have to agree perfectly.
```

```
* The name is a **plain filename, looked for in the assets directory**. Any
  directory part is discarded — a program cannot reach elsewhere on the
host.
* The font is rendered **once** and cached. It is rebuilt only when the
  filename or the cell size changes; rasterising 256 glyphs is far too slow
  to do per frame.
* If the font cannot be loaded the reason is reported once and the previous
  frame stays on screen, rather than the display going blank.
```

Internally a file font is converted into exactly the bitmap layout described above, so nothing else about the mode behaves differently.

Rules the descriptor must satisfy

A descriptor is checked before it is used. If it fails, the previous working geometry is kept and a message is printed once — the display will not follow you off a cliff, and the renderer never reads outside RAM.

- `VIDEO_COLUMNS` and `VIDEO_ROWS` must both be non-zero
- cell width and height must be non-zero, and no larger than 32×32
- the character plane must fit: `VIDEO_TEXT_BASE + VIDEO_ROWS * VIDEO_ROW_STRIDE`
- the colour plane must fit, by the same measure
- the glyph data must fit: `VIDEO_FONT_BASE + 256 * bytes-per-glyph`

A stride smaller than the column count is corrected up to the column count rather than rejected.

Three kinds of mode

It is worth being explicit about the taxonomy, because the descriptor applies to two of the three:

Kind	Where glyphs/pixels come from	Modes
Text, RAM font	`VIDEO_FONT_BASE`, `VIDEO_FONT_KIND=0`	1, 2, 10
Text, file font	`VIDEO_FONT_BASE`, `VIDEO_FONT_KIND=1`	10
Text, file font (fixed)	a TrueType file named in the emulator	6
Framebuffer	pixel data in RAM	3, 4, 5, 7, 8

Mode 6 is a text mode in every respect that matters — same character plane, same colour plane, same attribute format, same geometry registers — but its glyphs are rasterised from `PxPlus_IBM_VGA_9x16.ttf` rather than read from RAM. That is a feature, not an oversight: it is how you get a real VGA typeface.

Mode 6 still names its font inside the emulator rather than through the descriptor, so it remains a mode of its own. Mode 10 can now reach the same place under program control: set 9×16 cells, point `VIDEO_FONT_BASE` at the filename and set `VIDEO_FONT_KIND` to 1.

Framebuffer modes are a separate discussion and are not covered here.

Verification

The conversion was not judged by eye. The emulator's control console has a `vtest` command that renders both the original fixed-geometry mode and the new descriptor-driven renderer from the same RAM into separate buffers and compares them byte for byte:

- mode 1 — identical over all 64000 pixels
- mode 2 — identical over all 128000 pixels

This is also what proved the old font copy at `$01E000` and the master `petscii_data` blob agreed, which is what makes reading glyphs directly from the blob safe.

Experimenting from the console

Running the emulator with `-c` gives a command console on stdin. It can install a descriptor without writing a program:

```
  text <cols> <rows> <cell_w> <cell_h> [text_base] [colour_base] [font_base]
[stride]
```

For example:

```
  text 80 50 8 8 $020000 $021000
  text 132 43 8 8 $020000 $022000
  text 40 25 16 16
```

Also useful: `vtest [mode]` to run the comparison above, `sym <name>` to look up a KERNAL label's address, and `peek`/`poke`/`dump` to inspect the descriptor registers directly.

Current status

- modes 1, 2 and 10 all run through the descriptor-driven renderer
- mode 10 is selectable from a program (`LDA $400A` / `INT $10`), from

BASIC (`MODE 10`), and clears the screen like any other mode

- every preset mode writes `VIDEO_ROW_STRIDE`; mode 10 preserves it
- fonts may be relocated and redefined through `VIDEO_FONT_BASE`
- a text mode may draw with a host font file via `VIDEO_FONT_KIND=1`
- AH=$47 reports the real font pointer and its kind, in every mode

Still to come:

- There is no single call that installs a whole descriptor. A convenience

function (AH=$4A) has been proposed but not written; for now, write the

```
  registers.
* Mode changes do not restore the font. Only boot writes
  ''VIDEO_FONT_BASE'', so a custom font survives a switch back to mode 1.
  Whether presets should reset it is undecided — see [[#the font base is
sticky]].
* Mode 6 still names its font inside the emulator rather than through the
  descriptor.
```

From:
https://www.appledog.ca/wiki/ - **Appledog**

Permanent link:
https://www.appledog.ca/wiki/doku.php?id=sd:mode_10

Last update: **2026/09/07 10:22**