PAUL HAGSTROM, KANSASFEST 2022

MAKING AN APPLE /// ARCADE GAME

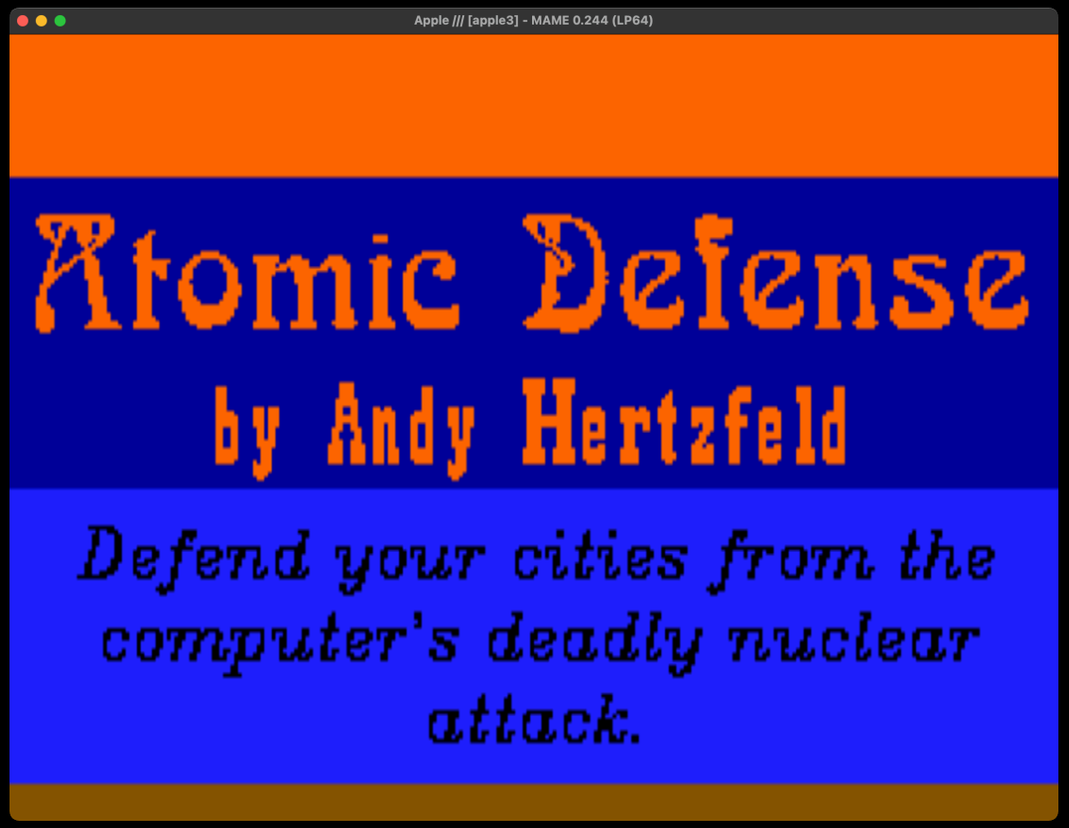
APPLE /// IMPROVEMENTS

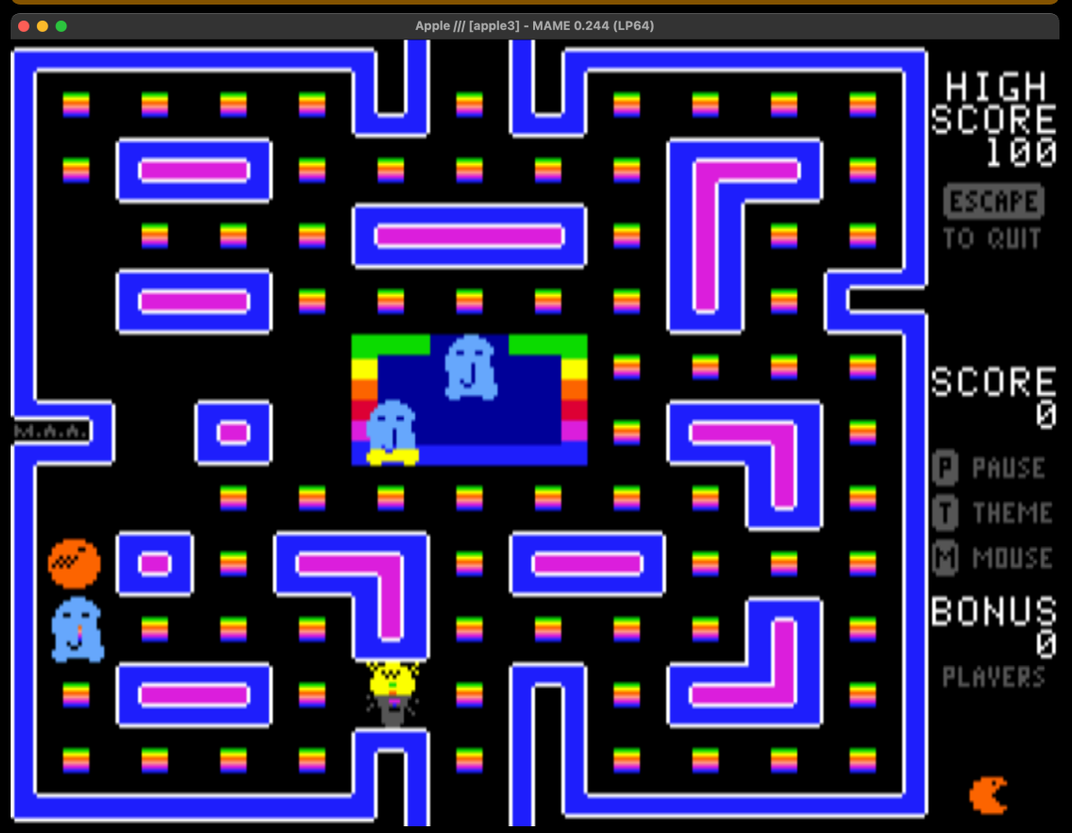
- 128K-256K RAM
- "2MHz" clock speed
- Serial port, Silentype printer port
- RGB output, 6 bit audio
- Clock nearly built in
- Custom text character sets
- Chainable disk drives, one built in.
- Better keyboard, numeric keypad
- 80 columns, lowercase, new graphics modes
- Emulation mode for Apple IIs circa 1980. 48K Apple II Plus.

PRETTY GREAT FOR WRITING GAMES?

- Sophisticated interrupts (VBL and more!)
- "Smooth scrolling" (hardware scrolling)
- 6-bit DAC audio
- "1.4MHz" speed
- 128K-256K memory
- Multiple zero pages, and stacks
- Customizable text fonts
- 140x192 16-color hires
  560x192 b/w super hires
  24x48 color text

BUT YET THERE ARE FEW





CUSTOMIZABLE CHARACTER SETS

- The Apple /// has its character set in RAM.
- Can animate REALLY FAST.
  - Change 56 pixels (7 across, 8 down) with a single write.
    Change an entire horizontal band 280x8 with 40 writes.
  - The on/off colors (16 for on, 16 for off) for all 56 pixels can be set with a single write.
- With this mechanism, you could write passable games even in Business BASIC. They're just text games.

TEXT-BASED GAMES

Apple /// [apple3] - MAME 0.244 (LP64)

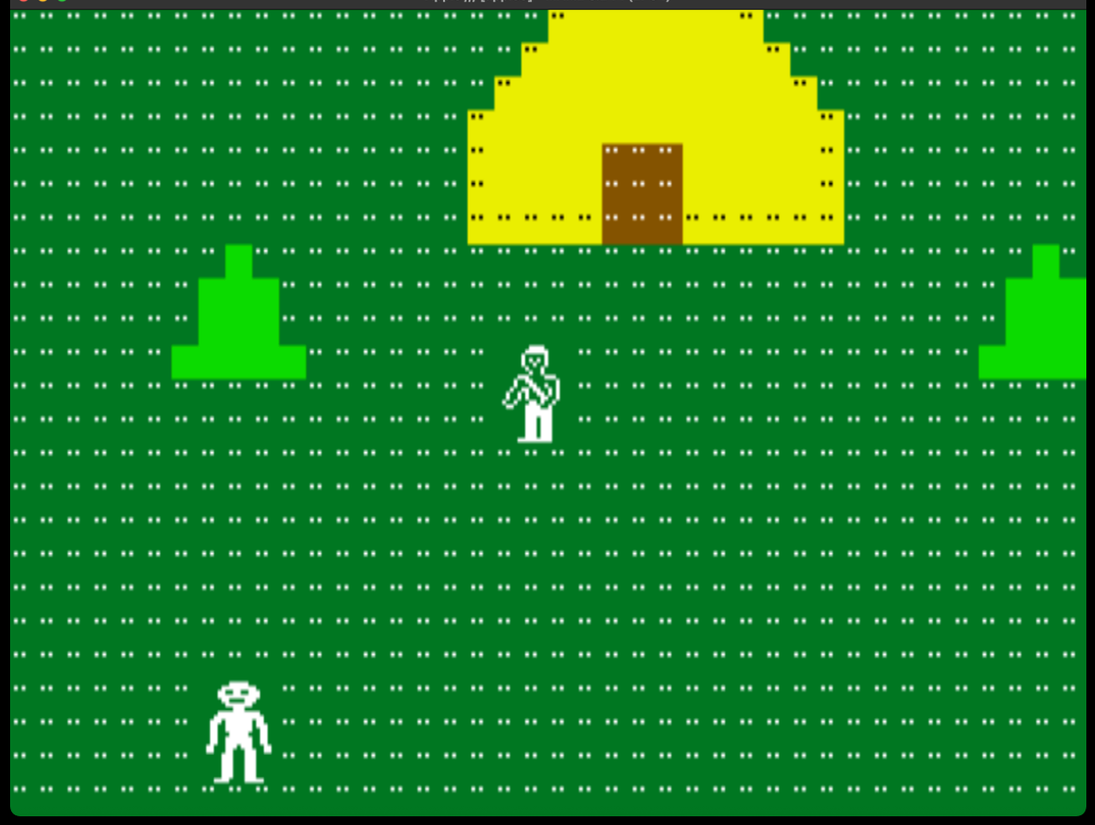
CAP'N MAGNETO

Cap'n Magneto --
A comic-book fantasy on a
Distant planet for the Apple ///

Program by Al Evans
Graphics by Celeste Evans

LOADING WORLD

Apple /// [apple3] - MAME 0.244 (LP64)



Apple /// [apple3] - MAME 0.244 (LP64)

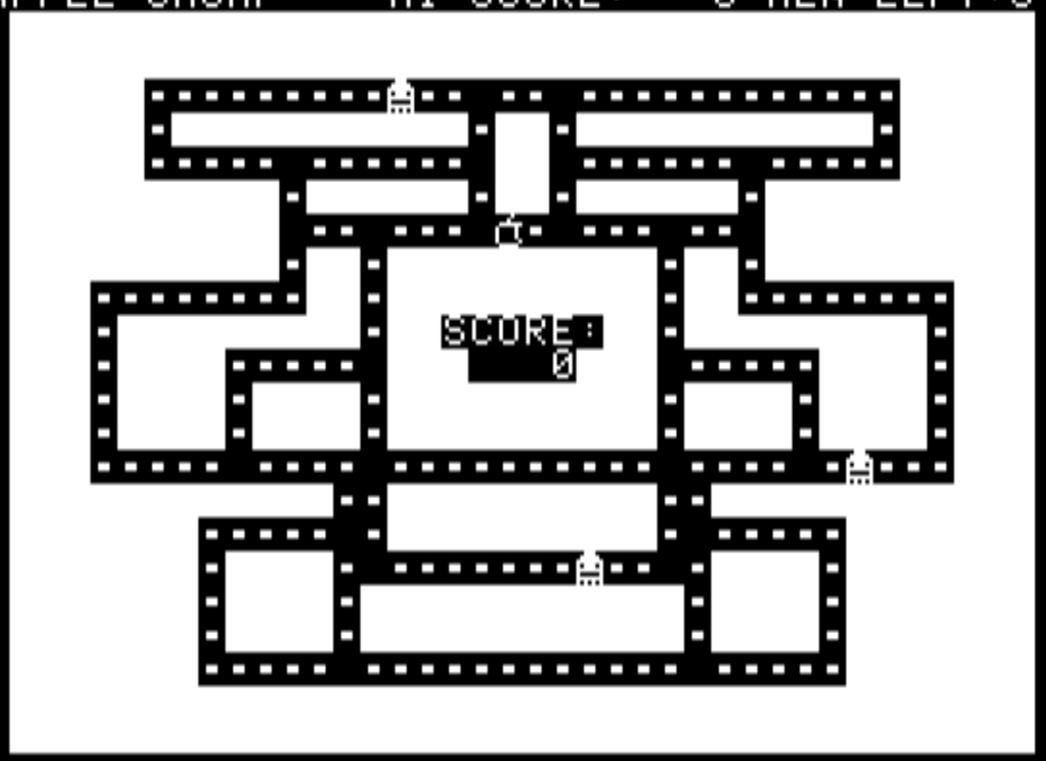
STROLL

CHOMP

Press a key to begin game

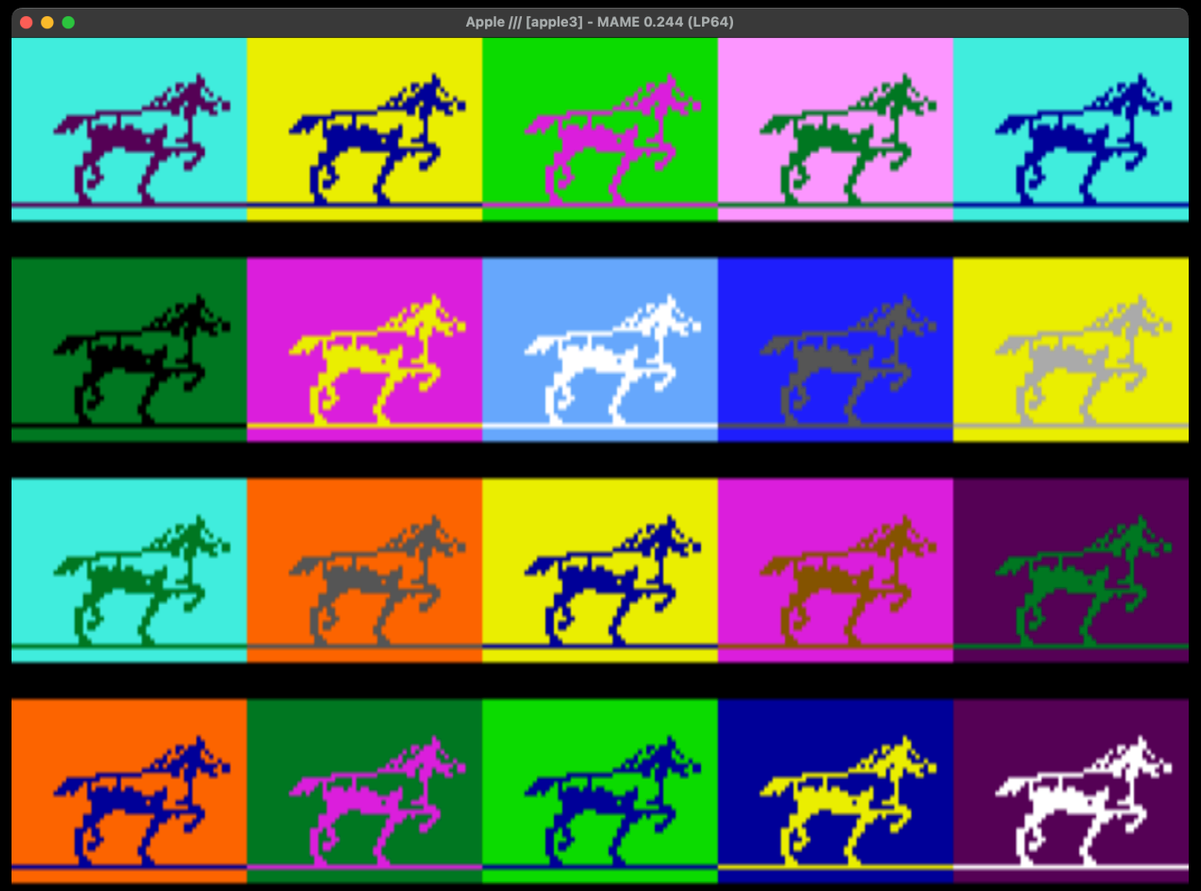
Apple /// [apple3] - MAME 0.244 (LP64)

APPLE CHOMP HI-SCORE: 0 MEN LEFT:3



RUNNING HORSES

- Andy Hertzfeld, Apple III System Demonstration



THE PLAN

- Write an arcade game in assembly language using the new graphics modes and enhanced features of the Apple ///.
- Of course, I hadn't written anything in assembly language on the Apple ///.
- Nor had I written any arcade games.
- Nor did I really know how the new graphics modes or enhanced features of the Apple /// work.
- This was a nearly foolproof plan.

THE TALK

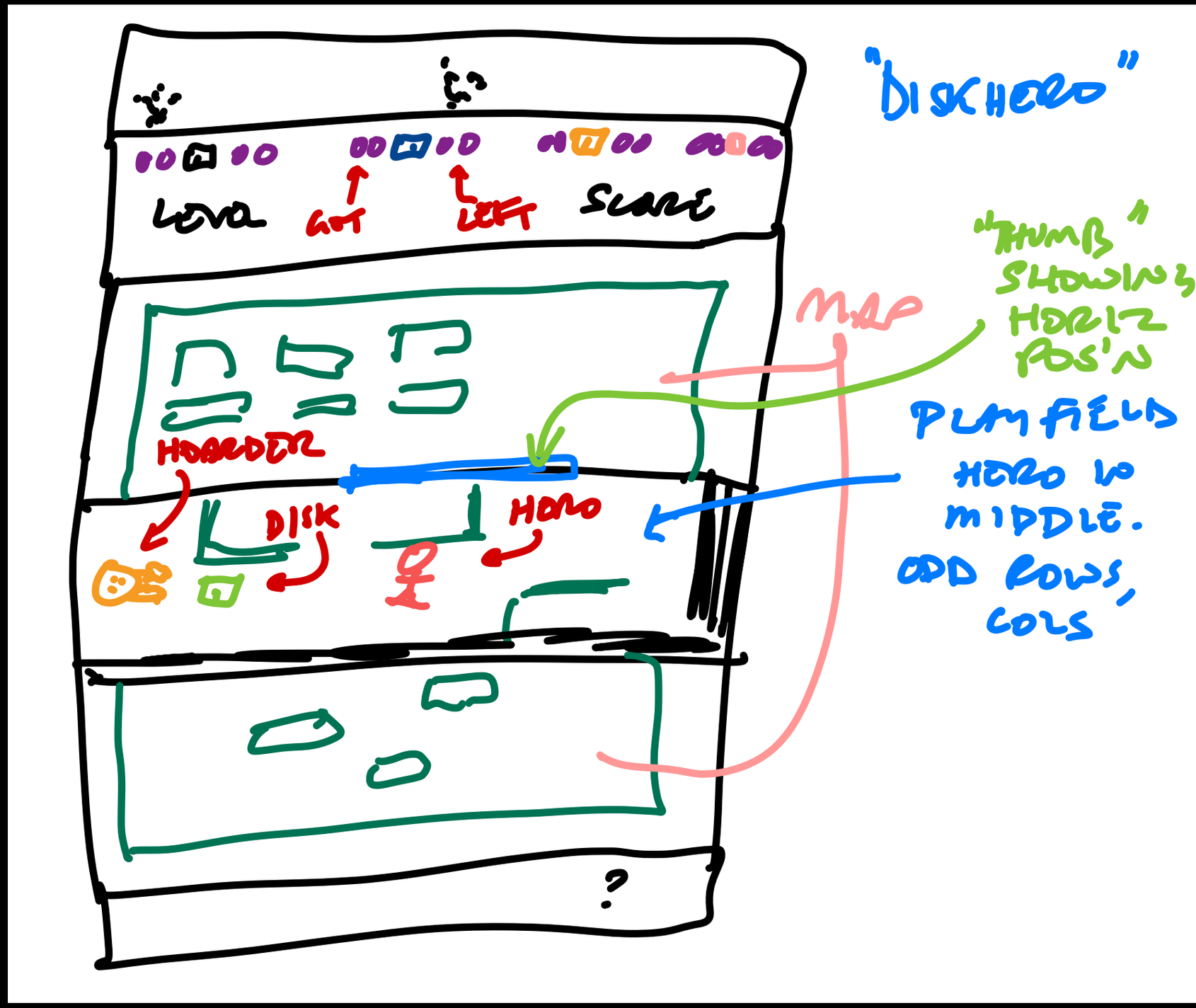
- Basic game plan
- Screen splitting, interrupts, blanking
- Dealing with memory
- Graphics modes
- Smooth scrolling
- Custom character sets
- Interrupt-driven audio

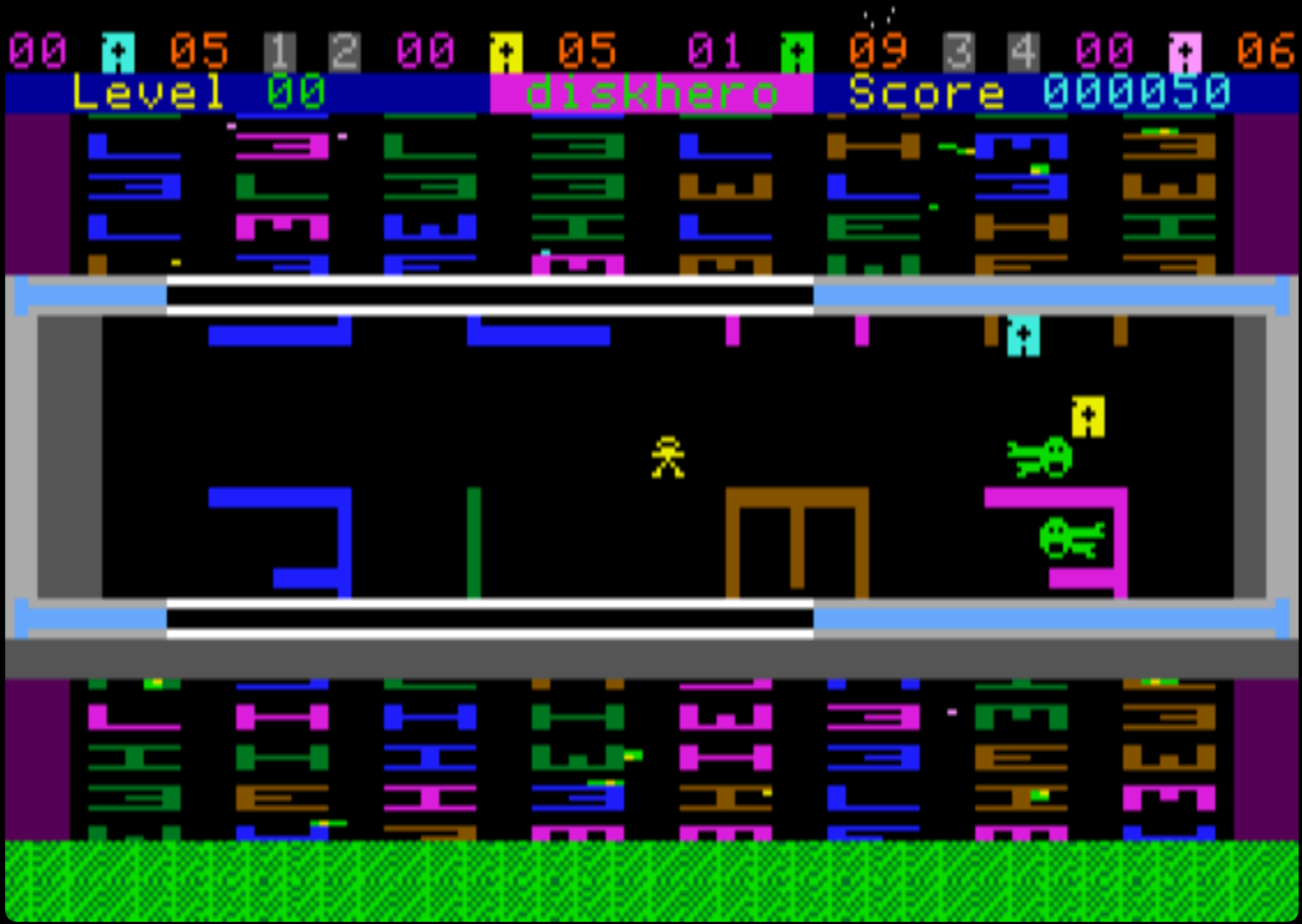
WHAT KIND OF GAME?

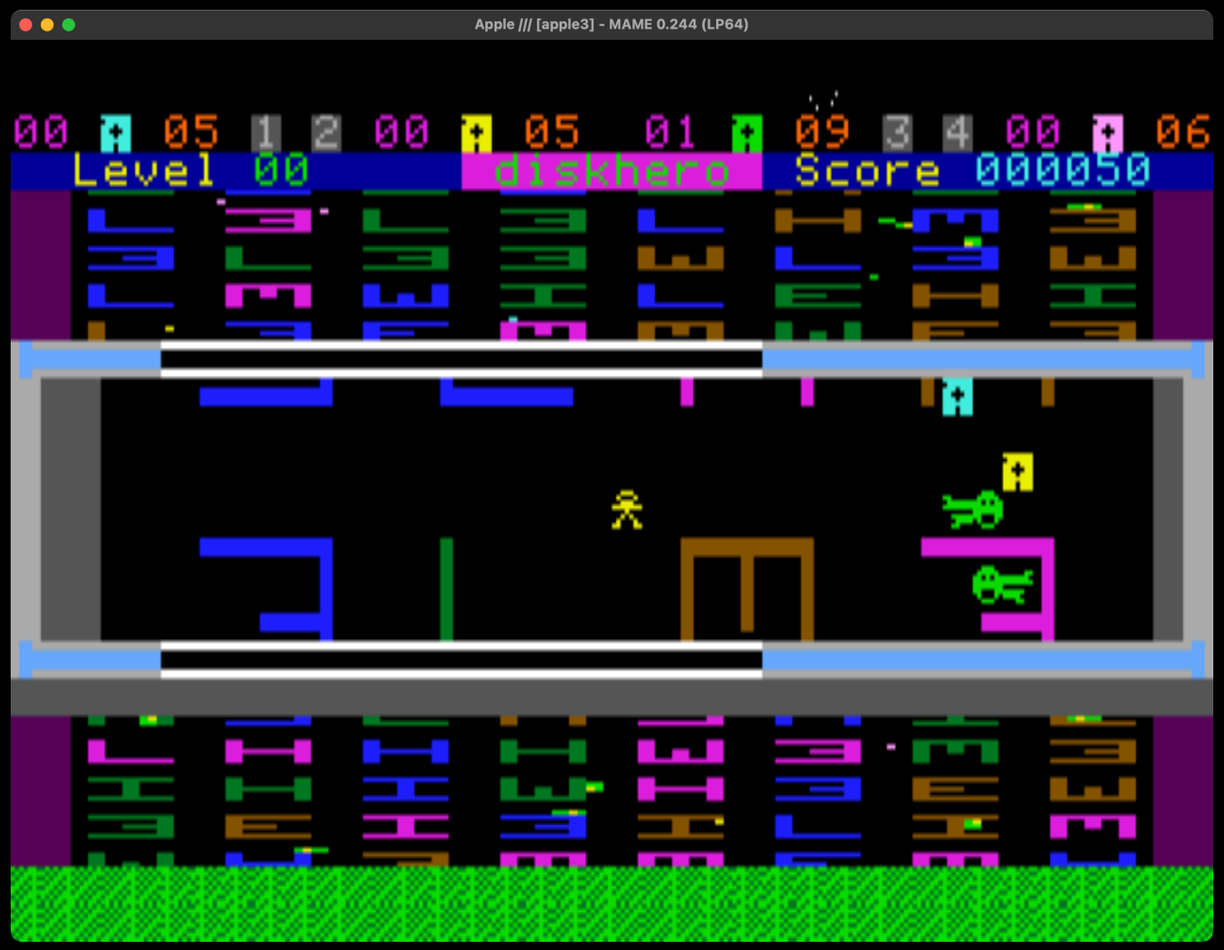
- Screen split into regions to use many different graphics modes at once, using the HBL signal.
- Use the new graphics modes (560 bw, 140 full color, 280 constrained color) and color text mode.
- Use smooth scrolling to shift pixels vertically, much faster than if they had to be drawn.
- Use custom character set to "fake" fast graphics.
- Use audio DAC for effects.

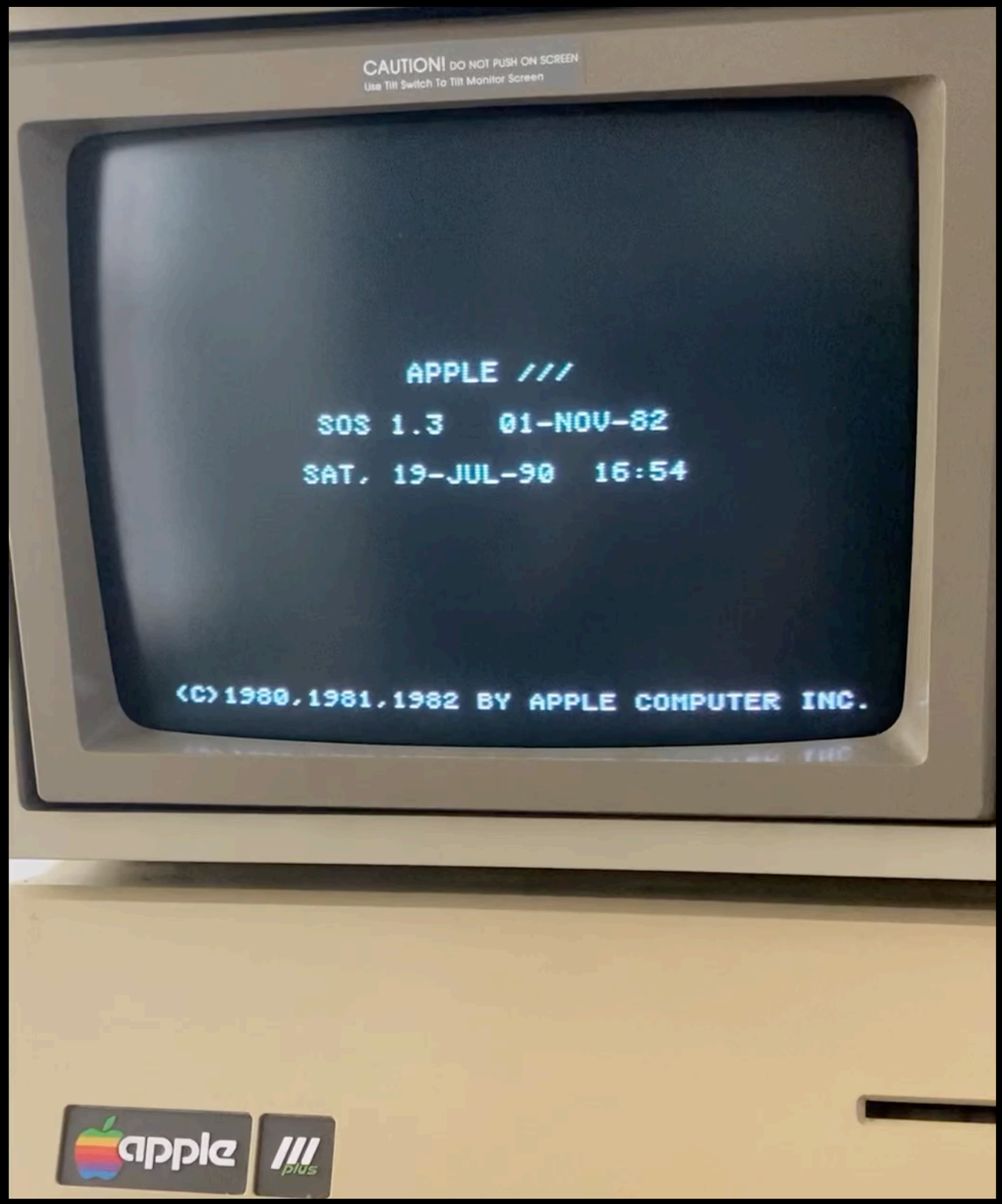
"DISKHERO"

- Disks scattered on map.
- You try to get them (to image them!) before the hoarders do.
- Hoarders seek high value disks, can be distracted by dropping one.
- A bit of a Dung Beetles vibe, zoomed in viewscreen on a smaller map.



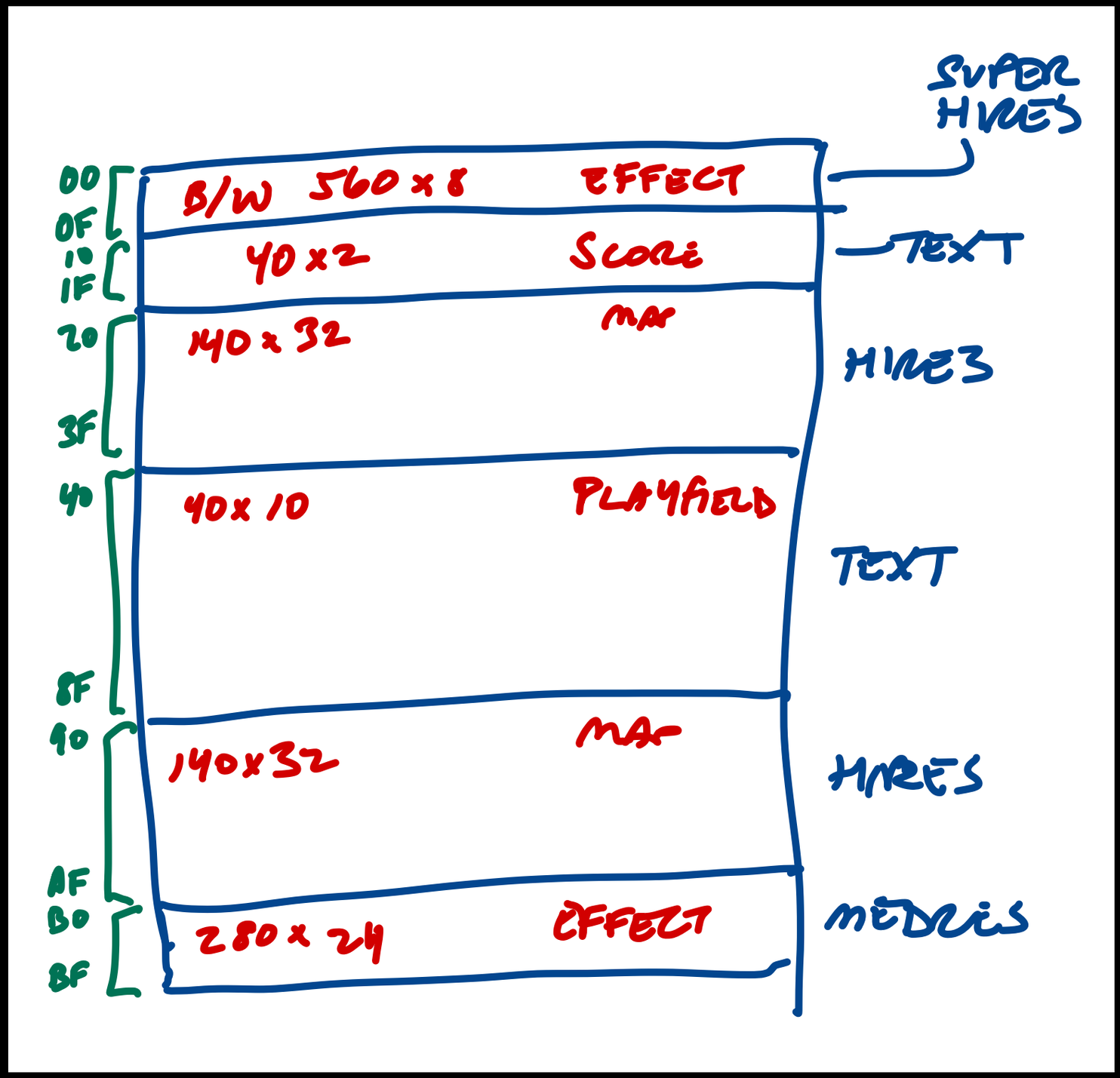

Level 00
diskhero
Score 000050


Apple /// [apple3] - MAME 0.244 (LP64)
Level 00
diskhero
Score 000050


CAUTION! DO NOT PUSH ON SCREEN
Use Tilt Switch To Tilt Monitor Screen
APPLE ///
SOS 1.3 01-NOV-82
SAT, 19-JUL-90 16:54
(C)1980,1981,1982 BY APPLE COMPUTER INC.
apple
plus

SCREEN REGIONS

- Top super hires region shows "splash" effect when inventory changes.
- Text region shows score and inventory.
- Hires regions show (portion) of larger map that scrolls.
- Text playfield uses custom characters as sprites for rapid animation.
- Medres region... never figured out what to do with that.



SCREEN SPLITTING

Have you ever wanted to create a display with both lo-res graphics and hi-res graphics on the same screen? Or graphics with more than just four lines of text at the bottom? Or how about text with four lines of graphics?

As we all know, the Apple II has only five display formats. It can display all lo-res graphics, all hi-res graphics, all text, lo-res with four lines of text at the bottom, or hi-res with four lines of text at the bottom. The latter two formats are sometimes called mixed modes because they allow, in a very restricted way, the mixing of graphics and text. But, according to page 12 of the *Apple II Reference Manual*, "There is no way to display both graphics modes at the same time." Well, not only are there ways of displaying both graphics modes on the same screen, it is also possible to display any combination of modes!

The technique of mixing display modes by the process of *screen splitting* is familiar to programmers who've used the Apple III, the Atari 400 and 800 machines, and several other computers. These machines contain special hardware that helps detect what is referred to as *vertical blanking* and *horizontal blanking*. What is not generally known is that the blanking can be detected by the Apple II, even though it lacks the special hardware found in those other machines.

```
100  HOME
200  FOR K = 0 TO 39
210  POKE 1448 + K,14 * 16
220  POKE 2000 + K,10 * 16
230  COLOR= K + 4
240  VLIN 25,45 AT K
250  NEXT K
300  VTAB 6: HTAB 17
310  PRINT "APPLE II"
400  CALL 768
500  GOTO 400
```

Listing 1.

```
0300-  8D 52 C0   STA  $C052
0303-  A9 E0      LDA  #$E0
0305-  A2 04      LDX  #$04
0307-  CD 51 C0   CMP  $C051
030A-  D0 F9      BNE  $0305
030C-  CA         DEX
030D-  D0 F8      BNE  $0307
030F-  A9 A0      LDA  #$A0
0311-  A2 04      LDX  #$04
0313-  CD 50 C0   CMP  $C050
0316-  D0 F9      BNE  $0311
0318-  CA         DEX
0319-  D0 F8      BNE
031B-  8D
```

Example Program. Before jumping into a technical discussion of the hows and whys of screen splitting, let's look at an example of screen splitting on the Apple II. Listings 1 and 2 present a short Applesoft main program and a machine language subroutine that the program calls.

Take a few moments now to turn on your Apple and enter these two programs. Don't worry if you don't understand machine language. Just go into the Monitor from Basic by typing *call−151* followed by the return key. Then start typing in the hexadecimal values for the listing 2 subroutine that starts at $0300:

```
300:8D 52 C0 A9 E0 A2
```

and so on followed by the return key.

Now run the Applesoft program. What do you see? (Nothing, if you didn't type in the listings correctly.) You should see a text message in the top half of the screen and lo-res color graphics in the bottom half. This is a display mode that's supposed to be impossible to create on the

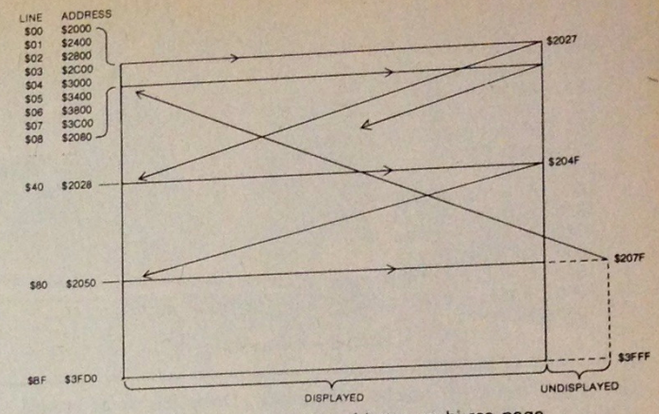


Figure 1. Memory mapping of bytes on hi-res page.

Apple maps its memory onto the display screen. (The latter information can be found on pages 14 through 21 of the *Apple II Reference Manual*.) The essence of what we need to know about hi-res in particular is shown in figure 1. Each line of the display is forty bytes long from left to right, and there are 192 such lines from top to bottom. The memory mapping seems somewhat haphazard: consecutive memory locations don't map onto consecutive lines of the display. Finally, for each set of 128 bytes of display memory only 120 bytes (three lines' worth) are displayed. The remaining eight bytes of the 128-byte set are never seen and are therefore sometimes referred to as the "undisplayed" or "unused" bytes. These undisplayed bytes all lie, conceptually, just off the bottom right-hand edge of the display, as shown in figure 1.

Text and lo-res both map in a way similar to hi-res, except that each cluster of eight lines now comes from one set of forty bytes instead of eight sets, and instead of the screen buffer being located at $2000 through $3FFF it lies at $0400 through $07FF. (Compare the *Apple II Reference Manual* pages 16 and 18 with page 21.)

Some Preliminary Insights. Let's try a few experiments that might give us some clues as to how screen splitting can be accomplished. From Basic type the command *call −151* (followed by return) to get into the Monitor. Next, clear the screen by issuing the escape−shift-P sequence. Now type *C051* followed by return. (Hitting return will always be assumed from now on.) The computer will probably display:

```
C051− A0
```

(If it doesn't, try typing *C051* again.)

Typing C051 from the Monitor is the way to turn on text mode if the computer is displaying graphics. But since we're already in text mode, nothing much happens—nothing much except that the contents of $C051 are displayed. But $C051 isn't supposed to be a readable address; it's merely a screen switch. So what does it mean for $C051 to contain $A0? Is it just a coincidence that $A0 is the hex code for an ASCII blank, and that most of the screen is also blank? What would happen if we typed C054? Or C056? Again, we tend to get $A0 if the screen is mostly blank.

Let's try another experiment. Again from the Monitor, type:

```
2000:73 2001< 2000.3FFEM
```

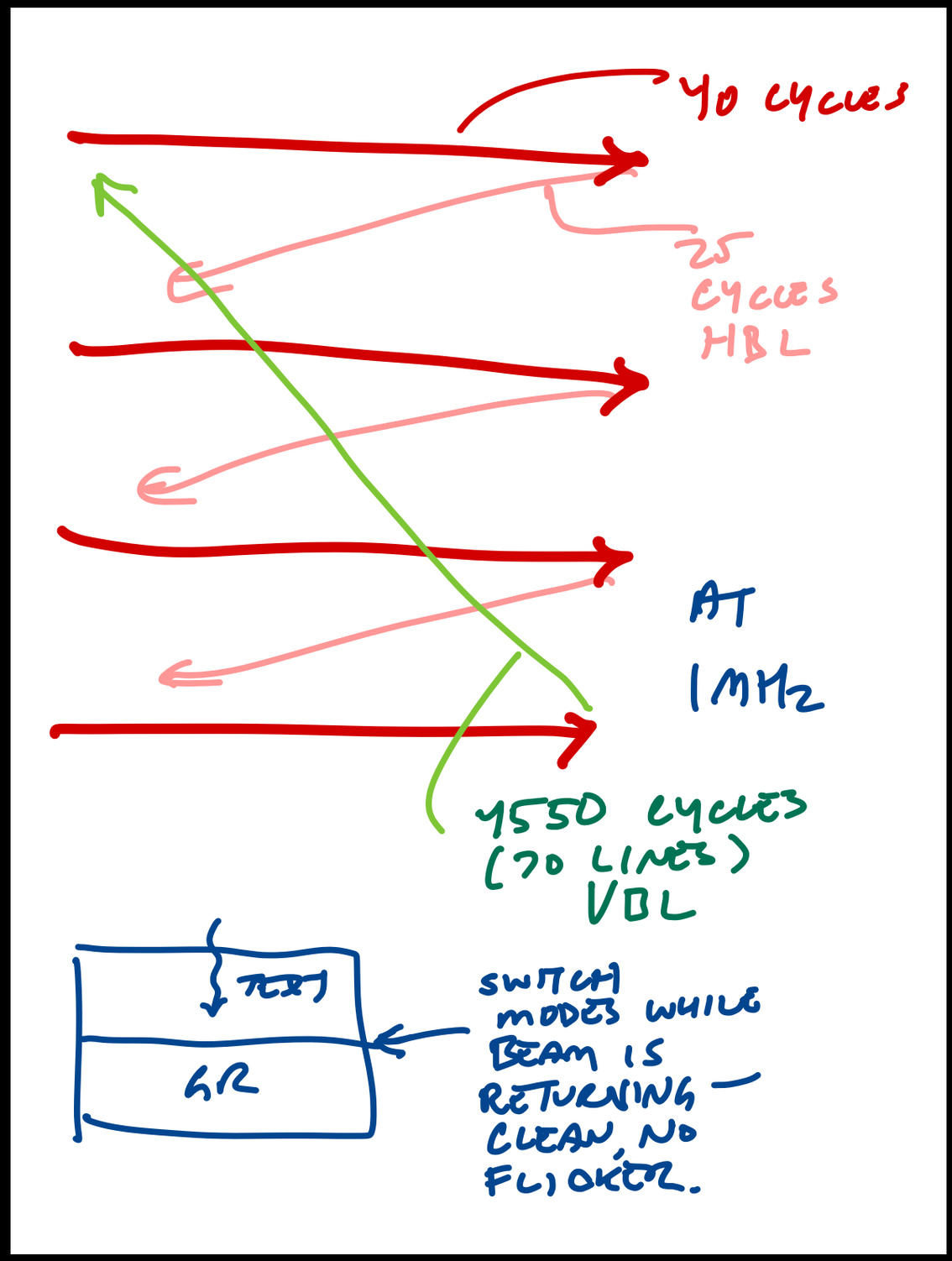
followed by:

```
C050  C053  C057
```

You should see some vertical hi-res lines with space for four lines of text at the bottom of the screen. Now type *C050*, or *C053*, or *C054*, or *C057*. Most of the time we now see $73 in the screen switch locations, and once in a while we see $A0. (Remember that the bottom four text lines on the screen are mostly blank.)

BLANKING INTERVALS

- Trick to screen splitting is to make changes when the beam is off. HBL or VBL.
- Timing is predictable, but on Apple II you don't have a VBL or HBL signal, so requires some trickery.
- The Apple /// has VBL interrupts. *And* HBL interrupts. Piece of cake.



6522 VIAS

- Two VIA chips deal with most of the interrupts, memory banking, zero page placement.

- These are pretty general-purpose devices. They have timers and counters and input/output registers. The Apple /// connects them to specific things.

- You talk to them via the addresses FFDx and FFEx.

Rockwell

R6522
VERSATILE INTERFACE ADAPTER (VIA)

DESCRIPTION

The R6522 Versatile Interface Adapter (VIA) is a very flexible I/O control device. In addition, this device contains a pair of very powerful 16-bit interval timers, a serial-to-parallel/parallel-to serial shift register and input data latching on the peripheral ports. Expanded handshaking capability allows control of bidirectional data transfers between VIA's in multiple processor systems.

Control of peripheral devices is handled primarily through two 8-bit bidirectional ports. Each line can be programmed as either an input or an output. Several peripheral I/O lines can be controlled directly from the interval timers for generating programmable frequency square waves or for counting externally generated pulses. To facilitate control of the many powerful features of this chip, an interrupt flag register, an interrupt enable register and a pair of function control registers are provided.

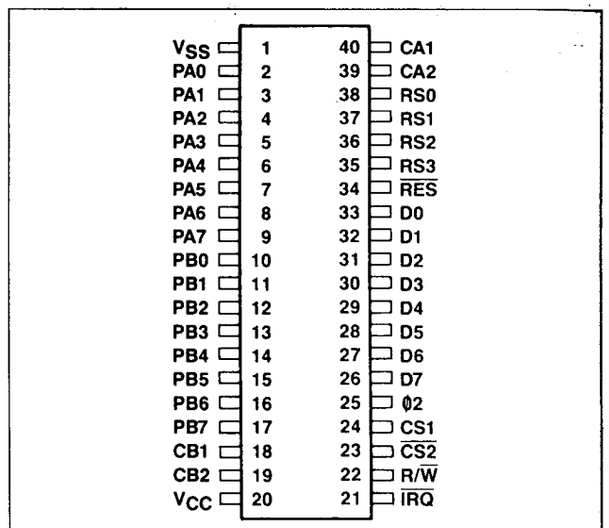
FEATURES

- Two 8-bit bidirectional I/O ports
- Two 16-bit programmable timer/counters
- Serial data port
- TTL compatible
- CMOS compatible peripheral control lines
- Expanded "handshake" capability allows positive control of data transfers between processor and peripheral devices.
- Latched output and input registers
- 1 MHz and 2 MHz operation
- Single +5V power supply

ORDERING INFORMATION

Part Number:
R6522

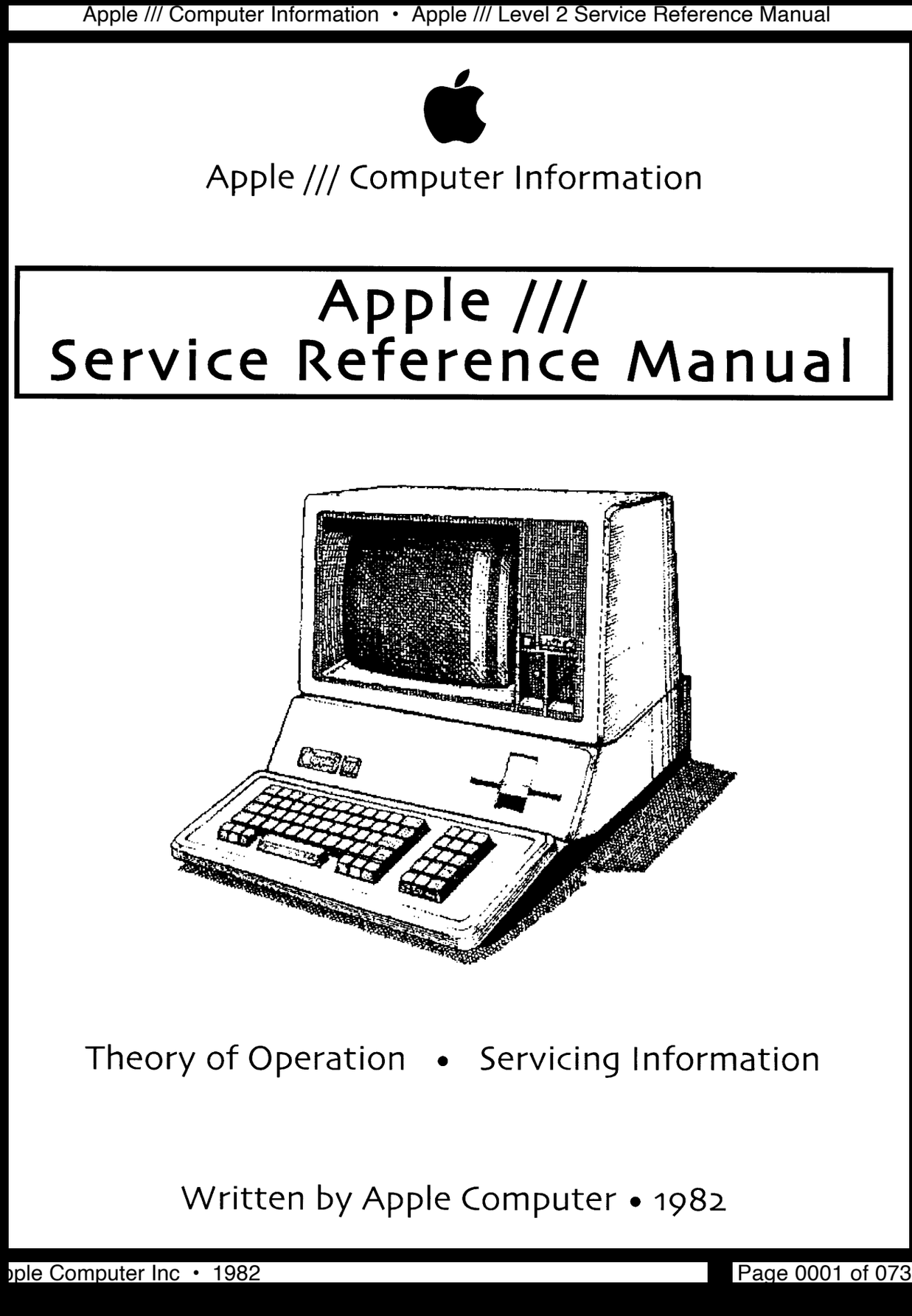
- Temperature Range
  - Blank = 0°C to +70°C
  - R = −40°C to +85°C
- Package
  - C = Ceramic
  - P = Plastic
- Frequency
  - No Letter = 1 MHz
  - A = 2 MHz



R6522 Pin Configuration

APPLE /// LEVEL 2 SERVICE REFERENCE MANUAL

- This is pretty much the only detailed reference to how all of this stuff works.
- It is pretty informal in places. And wasn't really available to regular people in the 1980s.
- There was kind of a lot of figuring things out needed.
  - Rob Justice's disassembly of Andy Hertzfeld's *Atomic Defense* was very helpful in getting started!



INTERRUPTS

- FFCD is the IRQ vector. If we put JMP Somewhere in there, the 6502 will jump there whenever an interrupt is encountered.

- The interrupt handler needs to save and restore the state (AXY), figure out which interrupt woke it up.

- By reading FFED (the E-VIA interrupt enable register), you can see what caused the interrupt, and by writing there you can clear it.

- Relevant interrupts here are HBL ($20), VBL ($10), Keyboard ($01), Timer ($02).

INTERRUPT TIMING

- Saving and restoring registers burns cycles. TXA, TAX, TYA, TAY are 2, PHA is 3, PLA is 4! RTI (return from interrupt) is 6.

- HBLs occur every 65 cycles, and you have 25 cycles while the blanking is actually happening. It is basically unworkable to count HBLs and switch graphics modes whenever you reach the scan line. By the time you're ready to switch, the HBL is over.

- However: You can set the VIA up to count HBLs, and report in only every, say, 8. That leaves time to restore the context after switching modes, and for doing the actual game logic between interrupts.

"2MHZ"

- The Apple /// runs its 6502 at 2MHz.
- Sort of. Sometimes.
- It runs at 1MHz when the video memory is going out. Even when the processor is set to 2MHz.

- 192 lines of video display (65 cycles each, 12480 cycles total)
- 70 lines of VBL (4550 cycles at 1MHz, so about 9100 cycles at 2MHz).
- 73% of the time, running at 1MHz, 27% of the time running at 2MHz.

"2MHZ" AND MAME

- MAME does not presently emulate the downshift for video memory.
- On a real ///, ideally shift mode between 65 and 80 cycles in (during the HBL after the one that triggered the interrupt). MAME is going to run twice as fast, getting through 130-160 cycles in that same time.
- Easy to wind up with something that looks fine in MAME and gets the splits all wrong on real hardware because HBLs were missed during processing.
- To get something that runs on both MAME and real hardware, must switch FAST because they get further apart quite quickly. (Also easy to fool yourself into thinking your game runs fast enough during development!)
- Very important to set the next 8-line HBL timer immediately in the handler.

INTERRUPT HANDLER

- This is what I have as of now. Not sure it's as good as it could be. Resets the timer by 30 cycles.

- Switches modes in 23 cycles by referring to modes by branch offset and modifying itself.

- There's a reason I did not lean more on the stack or zero page, which is...

diskhero

buildmap.s | interrupts.s | diskhero.inc | diskhero

```
53  ; mode 1 (a3 medres) lines B0-BF (10) 15-17  medium res something
54
55  ; Getting the HBL timer reset is urgent enough that I will do that
56  ; properly stashing the environment.  To dodge an inaccuracy with
57  ; present does not downshift to 1MHz during video drawing) I need
58  ; reset within the first 32 cycles ideally (MAME running at double
59  ; cycles before first HBL would be missed).  Next most crucial thi
60  ; video modes fast because real hardware is already progressing do
61
62  nothbljmp:  jmp nothbl          ; 19 once departing from here.  Ju
63
64  inthandle:  sta IntAStash       ; 4 save A
65              clc                 ; 2 assume by default that this is
66              lda RE_INTFLAG      ; 4 identify the interrupt we got
67              and #$20            ; 2 is it HBL after all?
68              beq nothbljmp       ; 2/3 branch+jump off to the rest
69              sta RE_INTFLAG      ; 4 clear the HBL interrupt
70              lda #$07            ; 2 reset the timer2 flag for 8 HB
71              sta RE_T2CL         ; 4
72              lda #0              ; 2 and...
73              sta RE_T2CH         ; 4 go! [30 to get to this point,
74              stx IntXStash       ; 4
75              lda NextMode        ; 4
76              sta modebranch + 1  ; 4 modify next instruction to go
77  modebranch: bne ismode1         ; 3 [45 to here]
78  ismode0:    bit D_TEXT          ;mode 0 - +00 - 40 char A3 text [1
79              bit D_NOMIX
80              bit D_LORES
81              jmp isddone
82  ismode1:    bit D_TEXT          ;mode 1 - +18 - medres [15 cycles]
83              bit D_NOMIX
84              bit D_HIRES
85              jmp isddone
86  ismode2:    bit D_GRAPHICS      ;mode 2 - +24 - super hires [15 cy
87              bit D_MIX
88              bit D_HIRES
89              jmp isddone
90  ismode3:    bit D_TEXT          ;mode 3 - +30 - A3 hires (3 cycles
```

Ln 94, Col 16 +0, -0 650

diskhero – zsh

```
(base) hagstrom@ThatmOneMini diskhero % ca65 diskhero.s
(base) hagstrom@ThatmOneMini diskhero % ld65 -o diskhero.bin -C apple3big.cf
(base) hagstrom@ThatmOneMini diskhero % ac -d test.po SOS.INTERP
```

POINTABLE ZERO-PAGE AND STACK

- Apple /// allows you to designate any page as"zero page" and "stack."
- The 6502 can refer to any address in a 64K space using 16-bit addresses, but two "pages" of 256 bytes are special.
- On the Apple II, the "zero page" occupies addresses $00 to $FF. The 6502 has instructions that can interact with these addresses faster, because it only takes one byte to specify an address.
- The "stack" occupies $100-1FF, and is a LIFO data store that the 6502 can stash information in fairly quickly (using pushing and pull instructions).
- On the Apple ///, you can specify which page is the zero page, it does not need to be $00. So you can interact fast with any page you choose. Like, say, the graphics buffer.

POINTABLE ZERO-PAGE AND STACK

- To set the zero page, you store the value in $FFD0.
- You then have a choice about the stack. It can either be
  - adjacent to the zero page
    (ZP EOR #$01, above ZP when odd, below ZP when even)
  - at $100
- If ZP is in video memory, true $100 is probably better—since alt stack would ALSO be in the graphics buffer. Constrains JSR and RTS along with PHA and PLA.
- Chosen via a bit in the Environment Register ($FFDF).

PUSHING TO THE SCREEN

```
 LDA  #$C1        2
 LDX  #$27        2
:STA  $0400,X     5
 DEX              2
 BPL  :-          2

                364
```

- You can fill the first text line ($400-427) with "A"s faster. Plus, the 6502 is clocked at closer to 2MHz.

```
 LDA  #$04        2
 STA  $FFD0       4

 LDA  #$C1        2
 LDX  #$27        2
:STA  $00,X       4
 DEX              2
 BPL  :-          2

     6+324  =  330
```

```
 LDA  #$05        2
 STA  $FFD0       4
 LDA  $FFDF       4
 ORA  #$20        2
 STA  $FFDF       4

 LDA  #$C1        2
 LDX  #$27        2
 TXS              2
:PHA              3
 DEX              2
 BPL  :-          2

    16+286  =  300
```

PUSHING PIXELS

- In various places I use either pushing to the screen via the stack, or writing to the screen using the zero page.
- But: this means that when an interrupt arrives, I don't know where the stack or zero page are. They might be onscreen. And it costs more cycles to save state, set them to a known value, and restore them than it does just to use absolute 16-bit addresses.

128-256K

- 6502 can see 64K at once.
- But we want more.
- Bank switching: 6502 refers to something within 64K, but Apple /// positions that 64K window over a larger RAM area.

apple computer inc.

Memory Map

128K

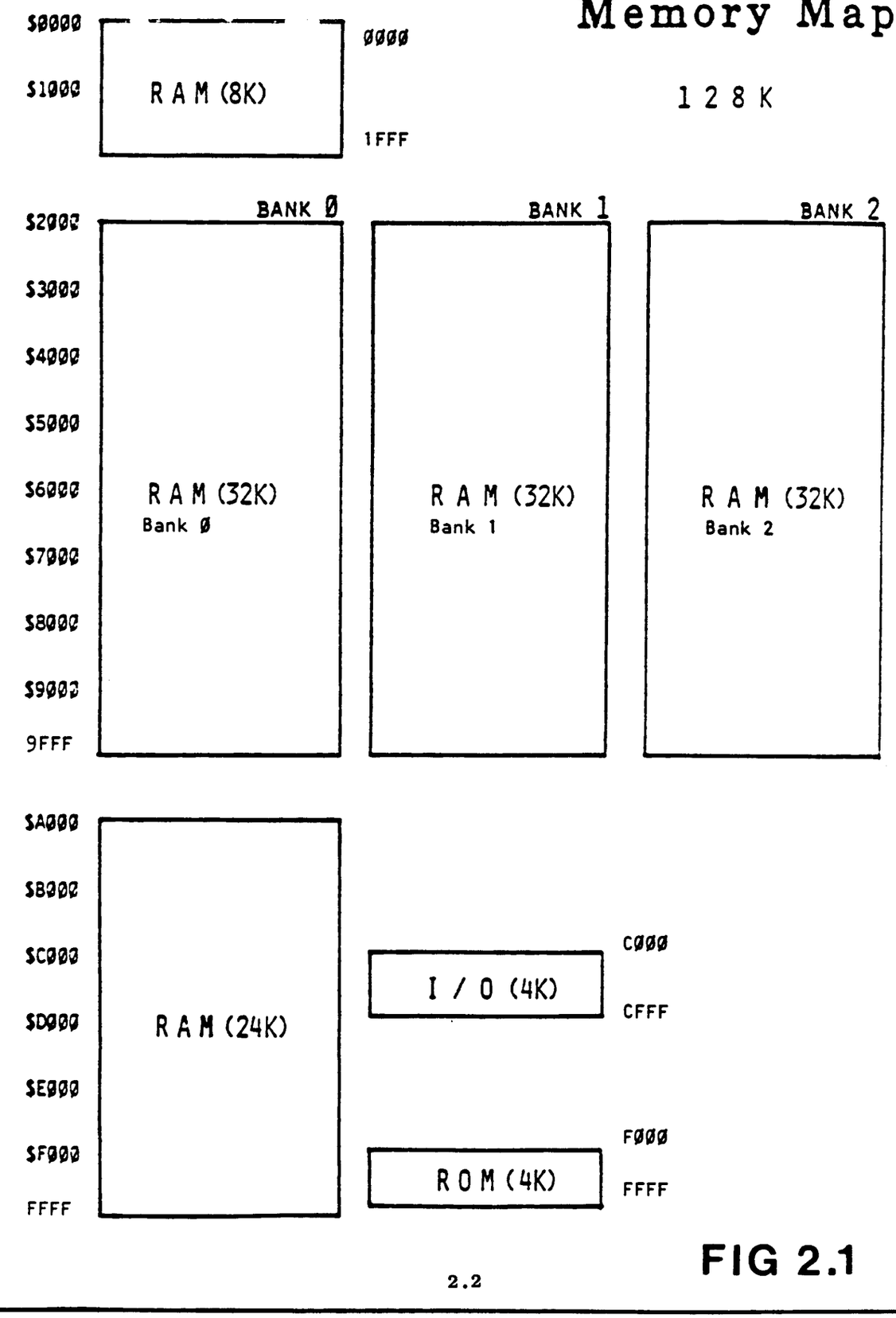
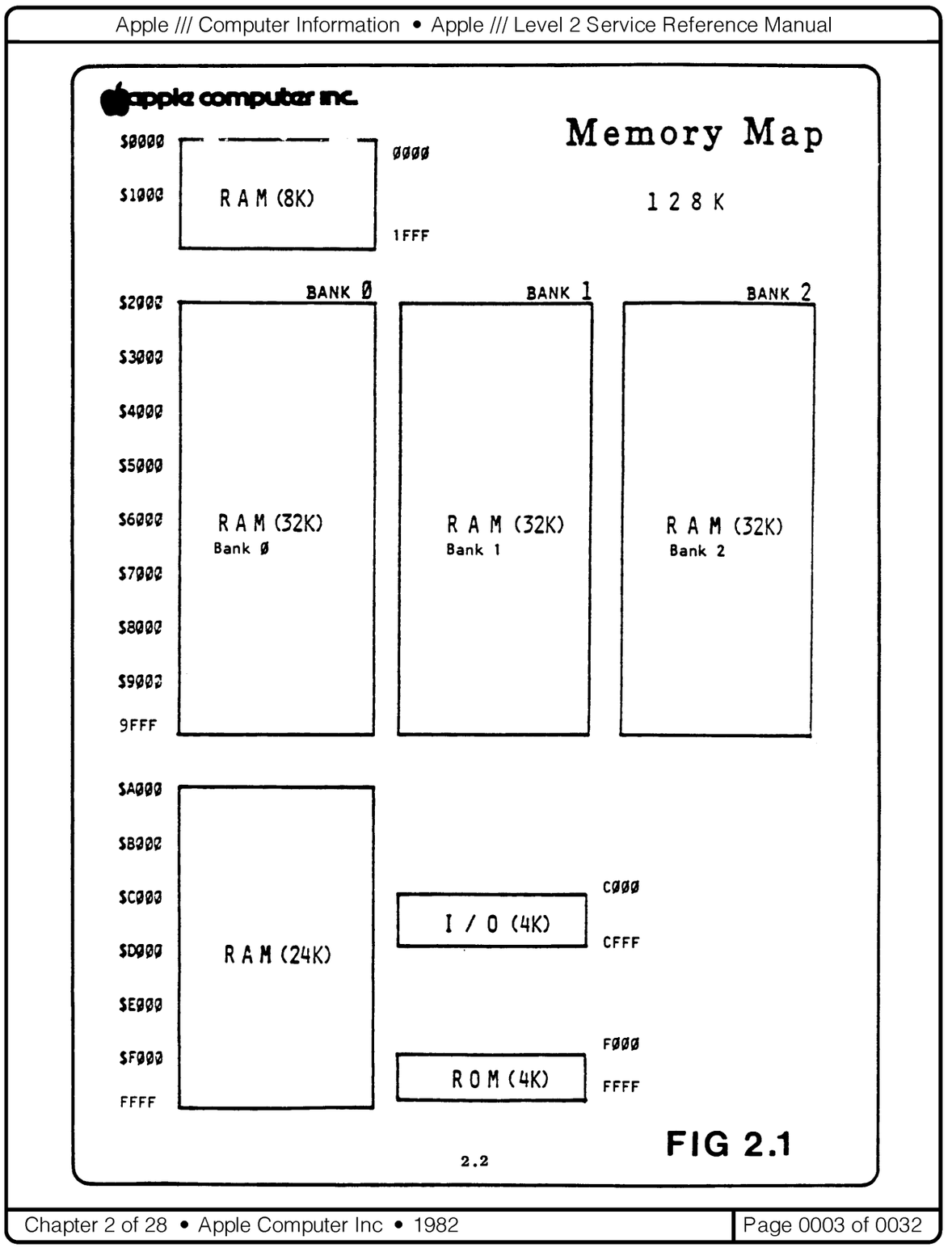


FIG 2.1

2.2

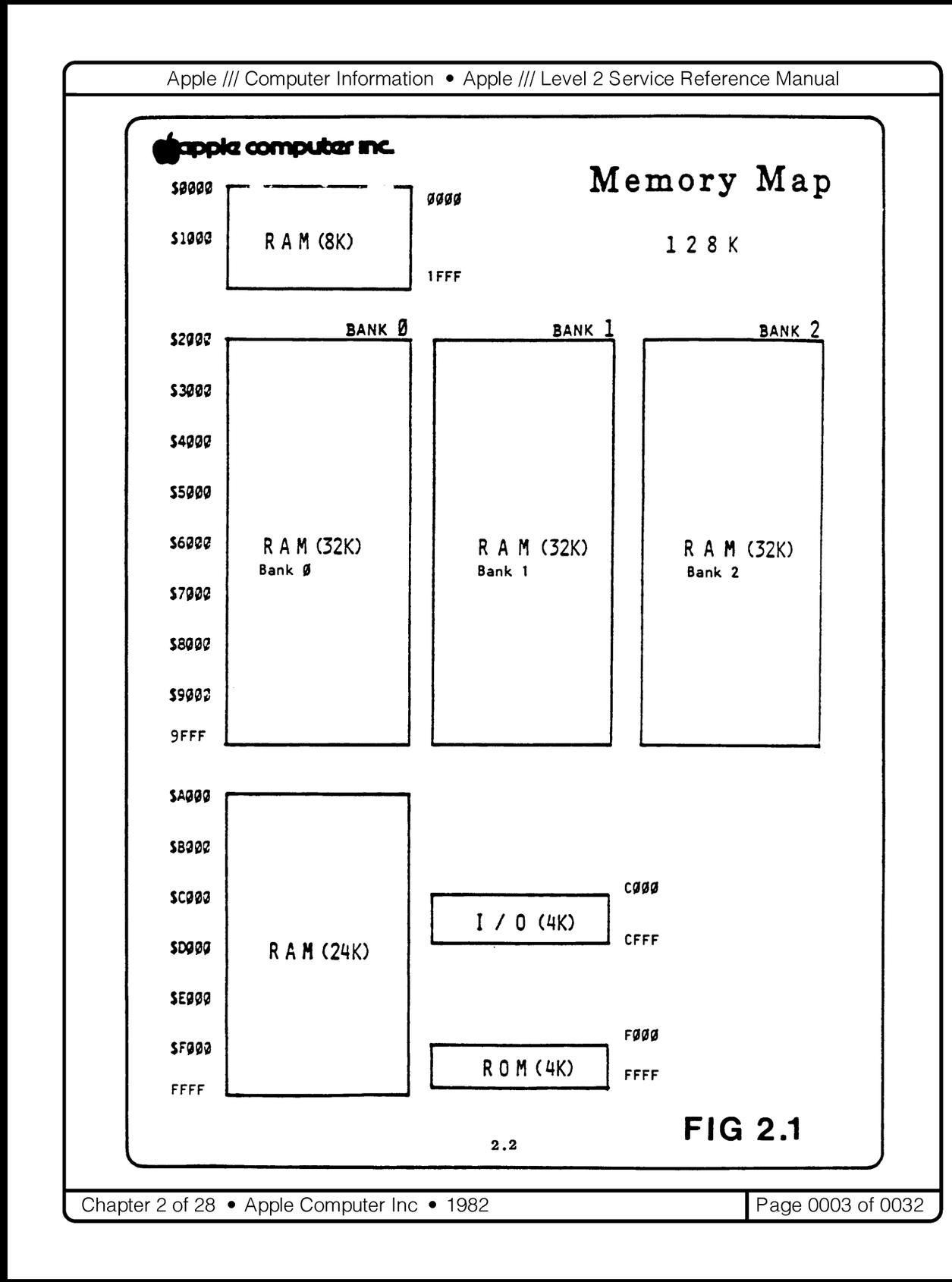
BANK REGISTER

- FFEF: bank register. Controls what bank is switched into the address space.
  - 0=bank 0, 1=bank 1...
- The "easy" way to deal with banked memory is to pick which bank is in 2000-9FFF and then manipulate memory in there. Swap banks if you want to manipulate other memory.



BANK REGISTER

- Except where is your program?
- Text memory (400-C00) lives below the banked area, always there, but might be on screen.
- Graphics memory (2000-9FFF) lives in bank 0 (only!).
- SOS claims A000 up, text memory (800-FFF) and zero pages and stacks eat up a lot below 2000.
- Not much to work with, and God Help You if you switch out the bank your code is running in. SOS 1.3 actually doesn't start until $B800 though.



EXTENDED ADDRESSING

- There is a special addressing trick to read/write data in other banks.

- The 6502 can only address 8 bits, but the Apple /// under certain circumstances will take the 6502's address and combine it with a bank address, and fake the 6502 out.

- The 6502 asks for the byte at $1000. The Apple /// checks the bank address, grabs the $1000 byte from the selected byte, and hands it to the 6502. The 6502 just thinks it is $1000, it knows nothing of banks.

- Same basic trickery as the relocatable ZP/stack.

EXTENDED ADDRESSING

- If your ZP is pointed at $1A00 (the default SOS provides for user programs), then:

- Store lower 16 bits of address at a ZP pointer, say $20.

- Store bank address in the $1600 page, parallel to the high byte of the pointer. $1621.

- Using addressing mode (ZP), y will interact with memory thus designated. Only that addressing mode.

READING FROM BANK 2, 3, 0.

```
LDA #$1A
STA $FFD0   ;ZP=1A

LDA #$00
STA $20     ;ADDRL
LDA #$90
STA $21     ;ADDRH
LDA #$82    ;BANK 2
STA $1621   ;XBYTE
LDY #$00
LDA ($20), Y
```

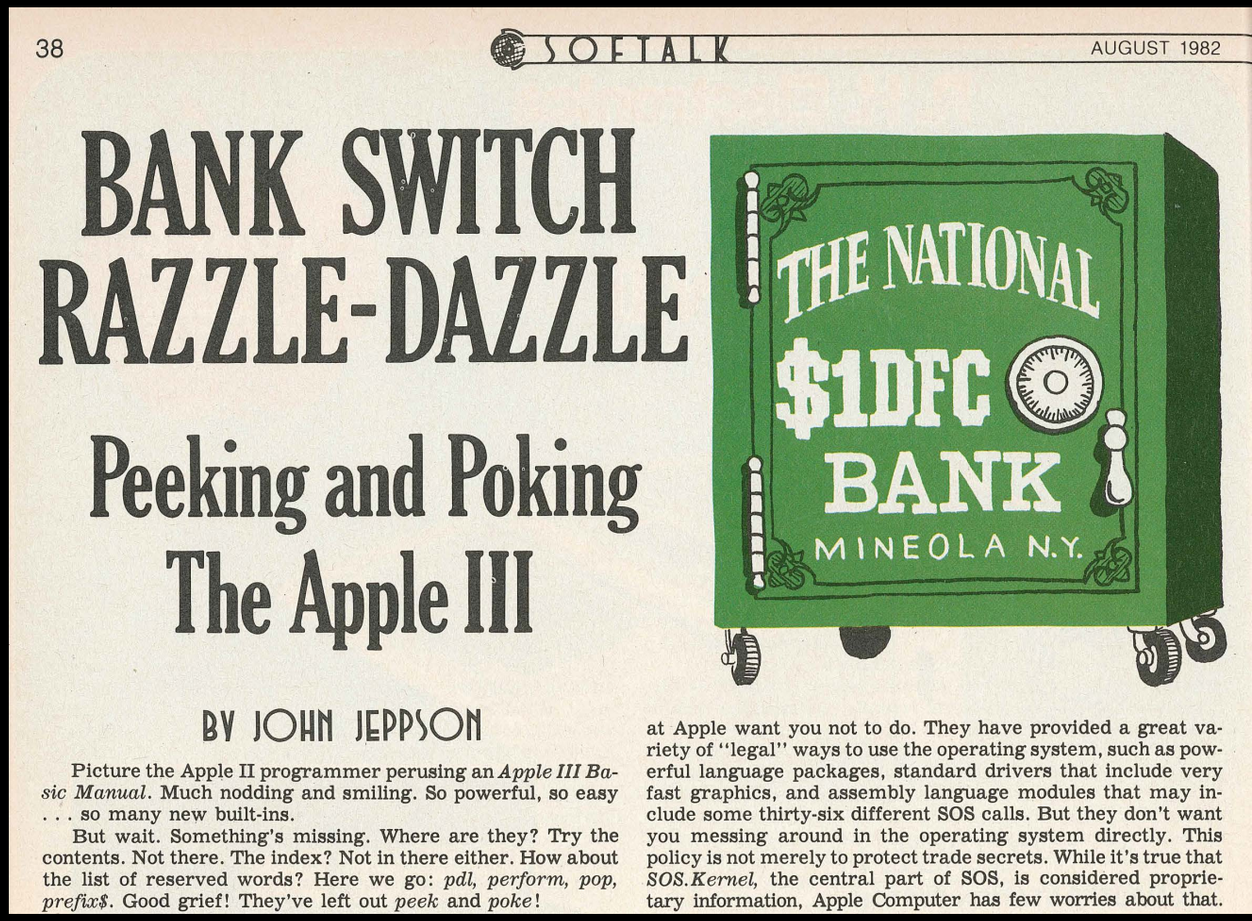
```
LDA #$1A
STA $FFD0   ;ZP=1A

LDA #$00
STA $20     ;ADDRL
LDA #$10
STA $21     ;ADDRH
LDA #$83    ;BANK 3
STA $1621   ;XBYTE
LDY #$00
LDA ($20), Y
```

- Bank in X-byte is $0000-7FFF. Next bank up is in $8000-FFFF.
- "Bank 2" address $9000 = "bank 3" address $1000.
- "Bank F" is special, regular memory map with bank 0 in $2000-$9FFF and the unbanked memory around it. Only parallel way to access first $100 bytes of bank 0, also allows reading data "under" the VIAs in $FFDx and $FFEx.

POINTER + $1601??

- Why pointer+$1601?
- Why "if your ZP is at $1A00"?
- Actual X-byte is at ZP EOR $0C00. Which is $1600 for $1A00. But $1000 for $1C00.
- ALSO, Extended addressing only kicks in with ZP between $18 and $1F. So: not if you've pointed your ZP at video memory.



38 SOFTALK AUGUST 1982

BANK SWITCH RAZZLE-DAZZLE

Peeking and Poking The Apple III

BY JOHN JEPPSON

Picture the Apple II programmer perusing an *Apple III Basic Manual*. Much nodding and smiling. So powerful, so easy . . . so many new built-ins.

But wait. Something's missing. Where are they? Try the contents. Not there. The index? Not in there either. How about the list of reserved words? Here we go: *pdl, perform, pop, prefix$*. Good grief! They've left out *peek* and *poke*!

at Apple want you not to do. They have provided a great variety of "legal" ways to use the operating system, such as powerful language packages, standard drivers that include very fast graphics, and assembly language modules that may include some thirty-six different SOS calls. But they don't want you messing around in the operating system directly. This policy is not merely to protect trade secrets. While it's true that *SOS.Kernel*, the central part of SOS, is considered proprietary information, Apple Computer has few worries about that.

(John Jeppson, *Bank Switch Razzle-Dazzle: Peeking and Poking the Apple III*, **Softalk**, Aug 1982)

EXTENDED ADDRESSING

- Video text memory on the Apple II: $400-7FF.
- Video circuit pulls values from text memory while the CPU is busy, deposits them on the screen.
- Can display either text page 1 ($400-7FF) or text page 2 ($800-BFF). They're located at XOR $0C00 from each other.

EXTENDED ADDRESSING

- To get 80-column text, both pages are used, and the values are interleaved. $400-7FF has the even characters, $800-BFF has the odd characters.
- The hardware has to do this all at the same time, so every fetch grabs BOTH $400 and $800, then BOTH $401 and $801, etc.
- That is, every fetch of $XXXX also grabs $XXXX XOR $0C00 in case it is needed.

EXTENDED ADDRESSING

- Suppose you do a LDA ($20), y.
- That will use the bytes in $20 and $21 as a 16-bit address, and then you will load the accumulator with what is in that address (plus y).
- That's going to grab $0C20 and $0C21 in the shadows as well. Though that's not super useful, those are sitting in text page 2 and we can't just freely change them.

EXTENDED ADDRESSING

- However you can point the "zero page" to whatever page you want.
- If you point the "zero page" to $1A00, then when it goes out to fetch $20, it's really going out to $1A20. Page 00011010. XOR with 00001100 yields page 00010110. Page 16. And the shadow system will grab $1620 at the same time.

EXTENDED ADDRESSING

- In summary:
  - if you use an indirect y-indexed zero-page addressing mode,
  - when the zero page is pointed somewhere between $18 and $1F inclusive,
  - then the X-byte is used to determine the bank you are reading/writing.

NEW VIDEO MODES

24x80 color text (16 foreground and background colors)

560x192 monochrome

140x192 with 16 colors *anywhere*

280x192 with 16 colors or grays.

apple computer inc.

APPLE /// VIDEO

INTRODUCTION

The Apple /// has 11 defined video modes of operation. There are 5 Apple][modes and 6 new Apple /// modes. There are now 3 text modes and 8 graphics modes. Though the Apple /// can emulate all of the Apple][video modes, there are many differences in the video hardware between the Apple][and Apple ///, including:

- o 80 column text with full upper and lower case character capability
- o New color text mode
- o Super high resolution black and white graphics
- o 2 new color hires modes

AND

- o A modifiable character set

The modifiable character set is a major new feature of the Apple ///. You can now change the character set by changing the pattern in the character generator. This is possible because of a ram, instead of a fixed rom configuration.

There are also improved video outputs. An NTSC (National Television Standards Committee) composite Black and White and color composite, plus the primary video signals, are available at the back panel for mixing into the input of a high quality RBG monitor.

The Apple][emulation mode has the very same video modes as the Apple][. The Apple ///, while in its native mode, can have the following modes.

40-COL TEXT

- 24x40 Apple II
  - familiar ($400, $800)
- 24x40 Apple ///
  - one page has the text
  - the other has the colors (high=bg, low=fg)
- Both can "flip."

40 Character Apple ///

This second 40 character text mode is the most interesting and, in a way, the most powerful. This is the only color text mode. It has the same screen resolution as the Apple][, and the same video attributes. BUT it also has the ability to select both the color of the foreground (dots) and the color of the background. Sixteen (16) colors are available as in the Apple][Lores Graphics.

- The color resolution can be selected for each character and can change for each character.
- It is interesting to note that by down loading a character set, a new low resolution graphics mode can be manufactured from a text mode.

The page mode is different for this mode since both pages are used at once. Why? Because the first page contains the character data and the second page contains the color information. The page 2 mode reverses the mapping, that is, the characters in page 2 are stored where the color was stored in page 1, and vice versa.

In the color byte, bits 4-7 set the foreground color and bits 0-3 set the background color. The mapping between color and character is 1:1. That is, a character located in 0409, for example, has its foreground color determined by the byte in location 0809.

In the page 1 mode the mapping is as follows:

0400-07FF contain the characters

0800-0BFF contain the color information.

In the page 2 mode:

0800-0BFF contain the characters.

0400-07FF contains the color.

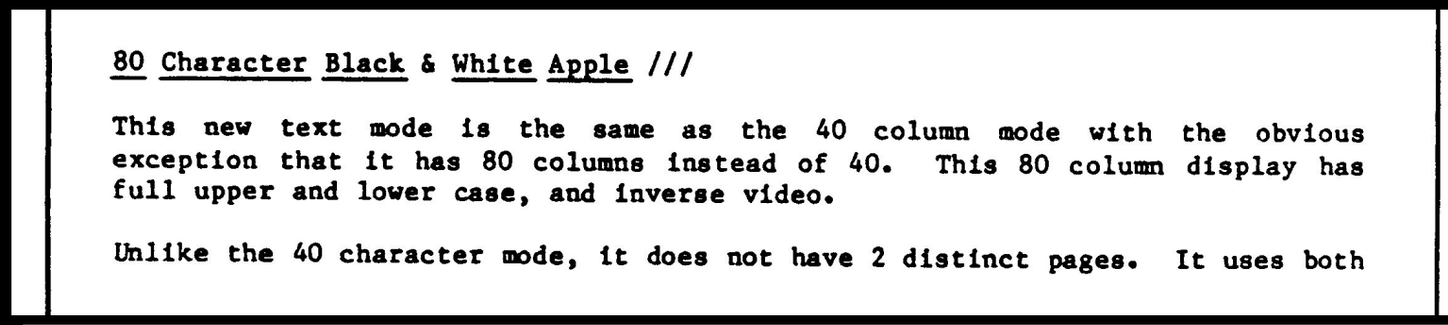
40 Character Apple][

This mode is equivalent to the Apple][text mode. The only difference is it has upper and lower characters.

- The screen is divided into 40 horizontal columns and 24 vertical lines.
- The characters are usually white dots on a black background.
- This mode has inverse video and flashing characteristics.
- This mode has no color.
- This mode has two screen pages mapped into memory:
  - Page 1 is located at 0400-07FF
  - Page 2 is located at 0800-0BFF.

80-COL TEXT
BW HIRES 280X192

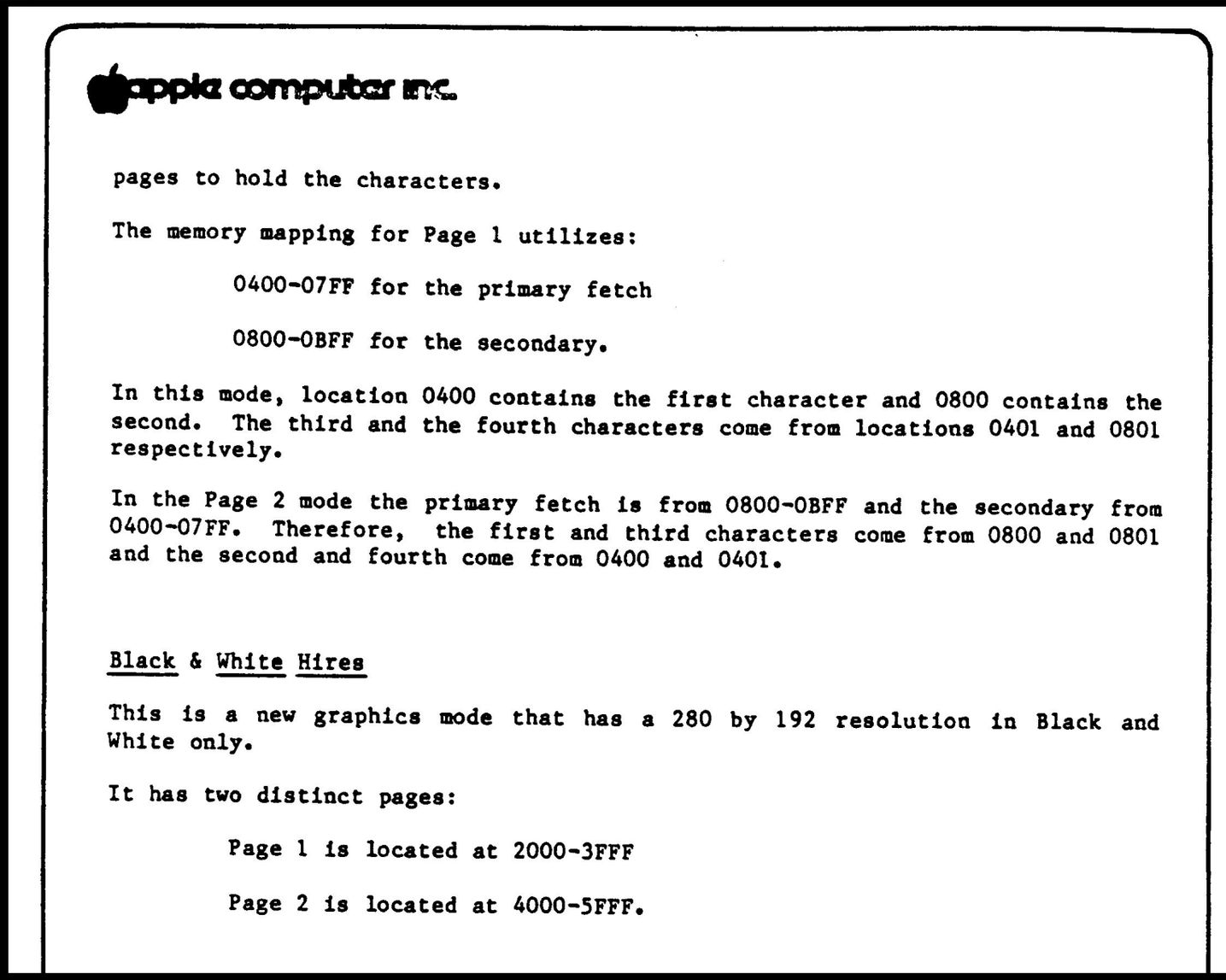
- 80-col text. Characters interleaved.
- $400 has the even characters, $800 has the odd ones.
- BW Hires 280x192
- Familiar $2000, $4000. The Apple II hires mode. Can flip.



80 Character Black & White Apple ///

This new text mode is the same as the 40 column mode with the obvious exception that it has 80 columns instead of 40. This 80 column display has full upper and lower case, and inverse video.

Unlike the 40 character mode, it does not have 2 distinct pages. It uses both



apple computer inc.

pages to hold the characters.

The memory mapping for Page 1 utilizes:

0400-07FF for the primary fetch

0800-0BFF for the secondary.

In this mode, location 0400 contains the first character and 0800 contains the second. The third and the fourth characters come from locations 0401 and 0801 respectively.

In the Page 2 mode the primary fetch is from 0800-0BFF and the secondary from 0400-07FF. Therefore, the first and third characters come from 0800 and 0801 and the second and fourth come from 0400 and 0401.

Black & White Hires

This is a new graphics mode that has a 280 by 192 resolution in Black and White only.

It has two distinct pages:

Page 1 is located at 2000-3FFF

Page 2 is located at 4000-5FFF.

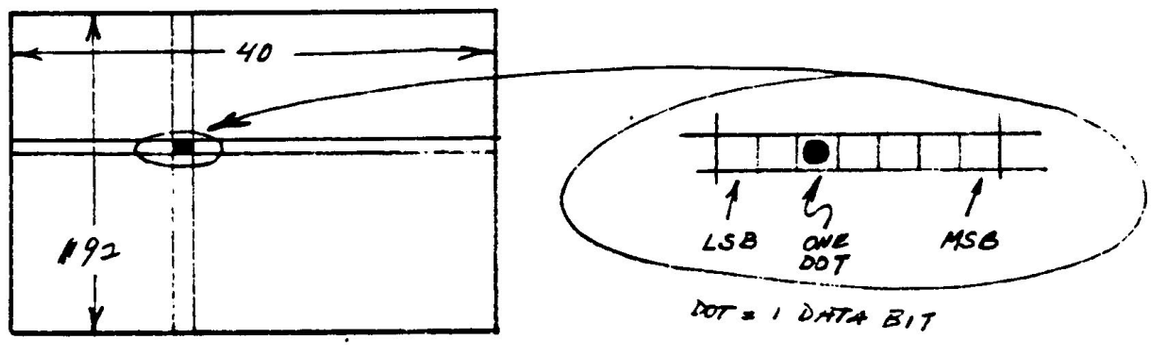
16-COLOR MEDRES

- 280x192 16 col/grey
- 40x192 in groups of 7.
- $2000 are the pixels
- $4000 are the colors
- Or they can be flipped, though not very usefully.
- This is the highest resolution 16-color mode, but colors are constrained, colors can only be set for each group of 7.

Medium Resolution 16 Color Graphics Apple ///

This is a new graphics mode for the Apple ///. It has the same dot resolution as the Apple][Hires (280 by 192), but it has an expanded color capability of 16 background colors. The B/W Output will yield 16 levels of grey scale.

The screen is divided into a 40 wide by 192 high matrix. That is, the color selection for foreground and 16 background can change for each 7 dot [ooooooo] pixel segment. You can think of each segment as a one-bit-high slice across a character space, as illustrated below.



6.4

The memory mapping is as follows:

Page 1:	2000-3FFF	each byte represents 7 pixels in the segment
	4000-5FFF	each byte represents the foreground and background colors for the corresponding 2000-3FFF byte.
Page 2:	2000-3FFF	each byte represents the colors
	4000-5FFF	each byte represents 7 pixels.

SUPER HIRES

- 560x192 B&W, in blocks of 7 that are drawn alternately (page 1) from $2000 and $4000.

- MSB is not displayed (maybe a place to hide data?). LSB is leftmost. Page 2 from $6000-9FFF, so CAN FLIP.

Super Hires Apple ///

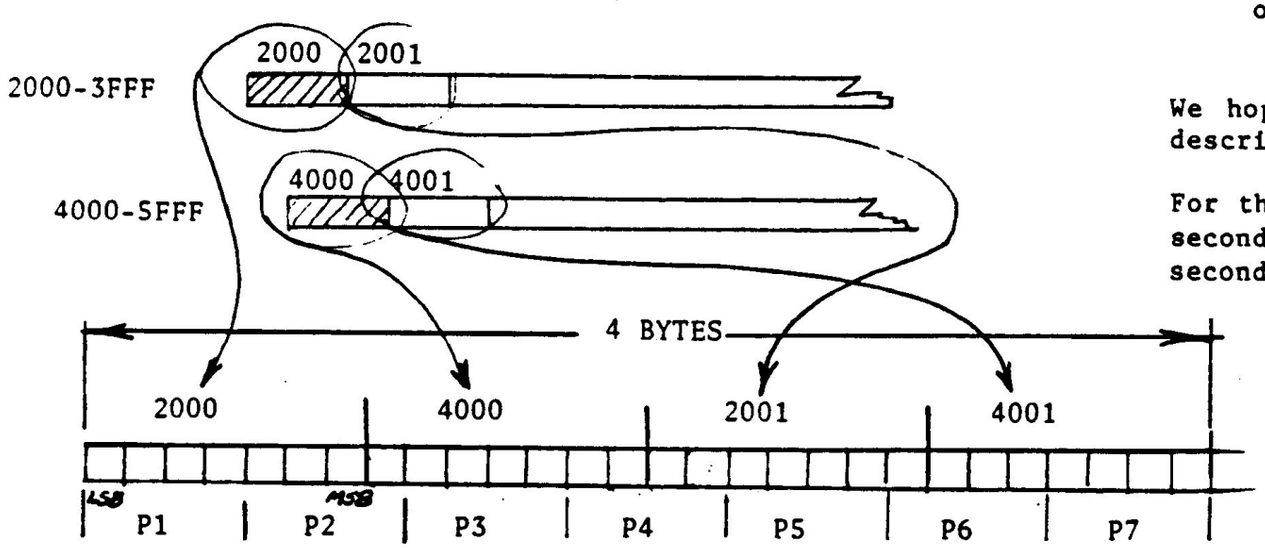
This is the Apple /// Hires equivalent of 80 character mode. It is a Black and White mode which has the dot resolution of 560 Horizontal by 192 vertical spaces.

There are two distinct screen pages, each with a primary and secondary page. Because it is like the 80 character modes, this mode draws its information from alternating ram. Each memory byte contributes 7 pixels. In Page 1 mode, the primary contains the odd dot groups and the secondary contains the even dot groups. The primary (first 7 pixels) is located at 2000-3FFF, and the secondary (second 7 pixels) is found at 4000-5FFF. In Page 2 the primary is at 6000-7FFF, and the secondary is 8000-9FFF.

In each byte the Most Significant Bit (MSB) is ignored and the data is displayed with the Least Significant Bit (LSB) first from left to right.

APPLE /// HIRES

- 140x192, 16 colors. "A bit difficult to master. Good luck!" 4 color bits per pixel. CAN FLIP.
  - 0: **2000**: 00001111
  - 1: **2000**: 01110000 + **4000**: 00000001
  - 2: **4000**: 00011110
  - 3: **4000**: 01100000 + **2001**: 00000011
  - 4: **2001**: 00111100
  - 5: **2001**: 00000001 + **4001**: 00000111
  - 6: **4001**: 01111000



Apple /// Hires

This is the third new graphics mode. It has 140 by 192 pixel resolution, and 1 of 16 color selection per pixel. In this mode the pixel is formed by a group of four dots of the same color.

There are two distinct screen pages in this mode but the mapping of the individual pages is, at first encounter, a bit difficult to master. Good luck!

- The display dot represents a sequence of 4 data bits in the ram display area.
- Two rams are used starting at 2000 and 4000 respectively and alternate bytes are fetched from each ram area.
- In any video mode only 7 of the 8 bits of each byte are displayed.

With this information in mind...and remembering that each pixel in this mode is made from 4 bits...you can see that you need 4 bytes of information to get 7 pixels. The way in which these bytes map into picture elements is shown below.

It is apparent, from the diagram, that picture elements overlap the byte boundaries for 7 picture elements and 4 bytes. The basic pattern then repeats.

The four bytes are shifted out in a fashion similar to the other Apple /// modes:

- The first byte comes from the primary and the second byte comes from the secondary.
- The first byte contains the first pixel and the second byte comes from the secondary.
- The first byte contains the first pixel and 3 bits of the second pixel.
- The second byte contains the fourth bit of the second pixel, the third pixel, and the first two bits of the fourth pixel.
- The third byte contains the last two bits of the fourth pixel, the fifth pixel, and the first bit of the sixth pixel.
- The fourth byte contains the last three bits of the sixth pixel and the entire seventh pixel.

We hope the preceding diagram will help you picture what has already been described.

For this mode, Page 1 is mapped with the primary fetch in 2000-3FFF, and the secondary in 4000-5FFF. In Page 2 the primary is in 6000-7FFF, and the secondary is in 8000-7FFF.

VIDEO MEMORY FOR REGIONS

- Band 0: Super hires: 2000-5FFF
- Band 1: Text: 400-BFF
- Band 2: Hires: 2000-5FFF
- Band 3: Text: 400-BFF
- Band 4: Hires: 2000-5FFF
- Band 5: Medres: 2000-5FFF
- Fortunately, all graphics modes have the same scan line organization. So some parts of 2000-5FFF will have super hires data, some will have hires data, some will have medres data. I did not attempt to flip graphics bands, though could have used 6000-9FFF for that in parallel.

SCROLLING GRAPHICS, II VS ///

👎

👍

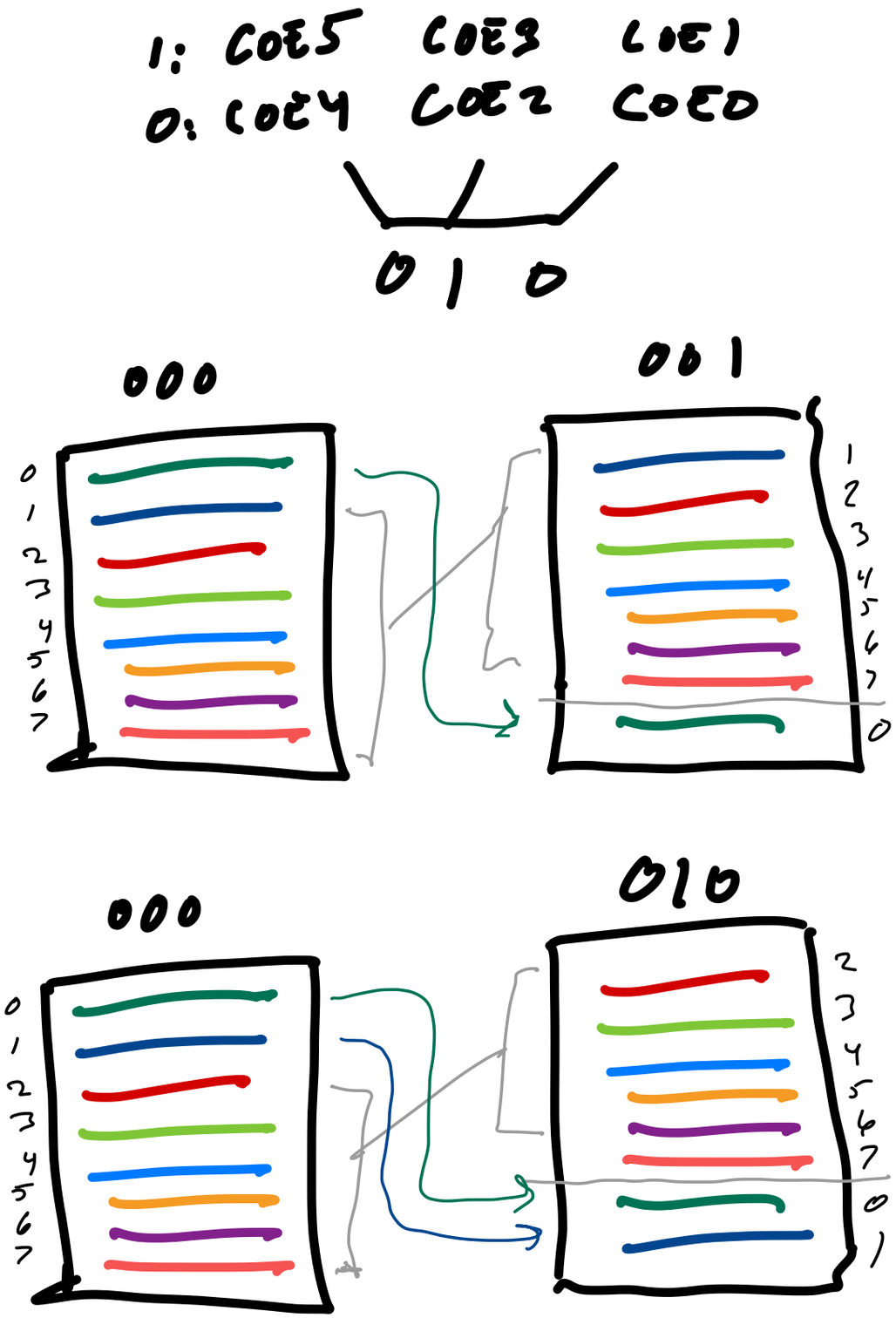
SMOOTH SCROLLING

The scrolling operation which is used is somewhat unusual in that each line of the display is separately moved up (line-by-line) with one line of data in memory being moved for each frame. This technique provides a uniform, esthetically pleasing, scroll. Scrolling the screen one line per frame can be achieved by moving all the data in the memory into a new position for each frame. This would be very time consuming and impractical. With the described technique, only one-eighth of the data in the memory is moved for each new frame.

Referring to the adder **121**, as mentioned, the signals V_A, V_B V_C are the three least significant vertical counter bits from the counter **58**. These bits or counts, by way of example, represent the 8 horizontal lines of each character. In adder **12**, a 3-bit digital signal, VA1, VB1 and VC1, is added to the count from counter **58**. This 3-bit signal is constant during each frame, however, it is incremented for each new frame.

During a first frame, 000 is added to the vertical count. During a second frame, 001 is added; and during a third frame, 010 is added, and so on. By adding this digital signal to the count from counter **58**, the addresses to the memory are changed in the vertical sense. During the first frame when 000 is added, the display remains unaffected. During the next frame, when 001 is added to the vertical count, instead of first displaying the first line of a character, the second line of each character is displayed at the top of each character space and each subsequent line of the character is likewise moved up one line. If data in memory is not moved, the first line of the character would appear at the bottom of each character. Note when 001 is added to 111 from the counter, 000 results. Thus, the first line of characters would be addressed when the beam is scanning the eighth line of characters. To prevent this, the data corresponding to the first line of each character is moved in memory for this frame. The first line of one character is moved up and becomes the bottom line of the character directly above it. When 010 is added, the process is again repeated. For example, the third line of each character is first displayed in each character space and the second line of each character is moved up to become the bottom line of the character directly above it. This process is repeated to scroll the data. The movement of data in memory is controlled by the CPU in a well-known manner.

SMOOTH SCROLLING



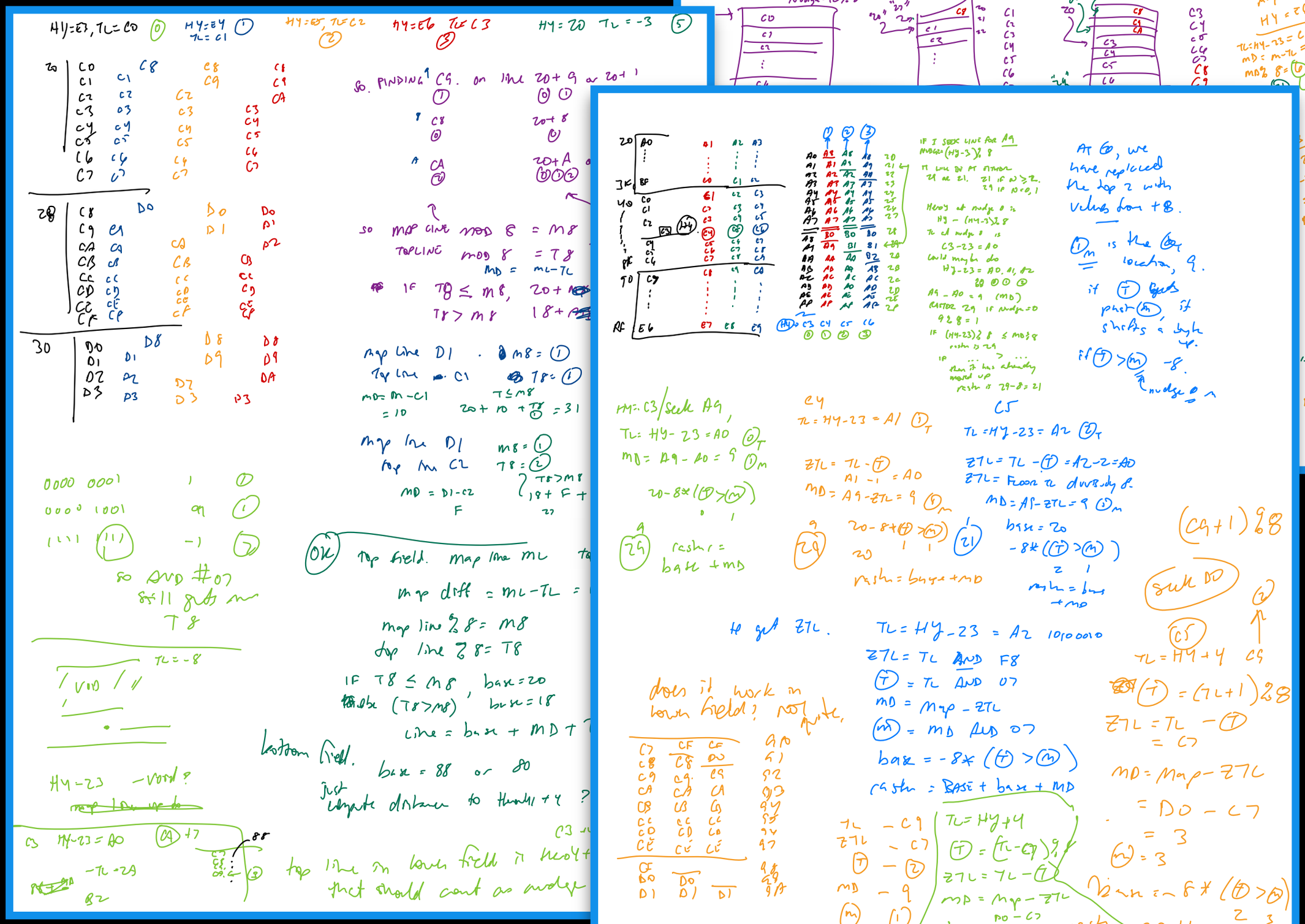
thus select different rows within the memory of FIG. **6**.

The scrolling operation which is used is somewhat unusual in that each line of the display is separately moved up (line-by-line) with one line of data in memory being moved for each frame. This technique provides a uniform, esthetically pleasing, scroll. Scrolling the screen one line per frame can be achieved by moving all the data in the memory into a new position for each frame. This would be very time consuming and impractical. With the described technique, only one-eighth of the data in the memory is moved for each new frame.

Referring to the adder **121**, as mentioned, the signals V_A, V_B V_C are the three least significant vertical counter bits from the counter **58**. These bits or counts, by way of example, represent the 8 horizontal lines of each character. In adder **12**, a 3-bit digital signal, VA**1**, VB**1** and VC**1**, is added to the count from counter **58**. This 3-bit signal is constant during each frame, however, it is incremented for each new frame.

During a first frame, 000 is added to the vertical count. During a second frame, 001 is added; and during a third frame, 010 is added, and so on. By adding this digital signal to the count from counter **58**, the addresses to the memory are changed in the vertical sense. During the first frame when 000 is added, the display remains unaffected. During the next frame, when 001 is added to the vertical count, instead of first displaying the first line of a character, the second line of each character is displayed at the top of each character space and each subsequent line of the character is likewise moved up one line. If data in memory is not moved, the first line of the character would appear at the bottom of each character. Note when 001 is added to 111 from the counter, 000 results. Thus, the first line of characters would be addressed when the beam is scanning the eighth line of characters. To prevent this, the data corresponding to the first line of each character is moved in memory for this frame. The first line of one character is moved up and becomes the bottom line of the character directly above it. When 010 is added, the process is again repeated. For example, the third line of each character is first displayed in each character space and the second line of each character is moved up to become the bottom line of the character directly above it. This process is repeated to scroll the data. The movement of data in memory is controlled by the CPU in a well-known manner.

ROUGH SAILING



SMOOTH SCROLLING VS HIRES VS MAME

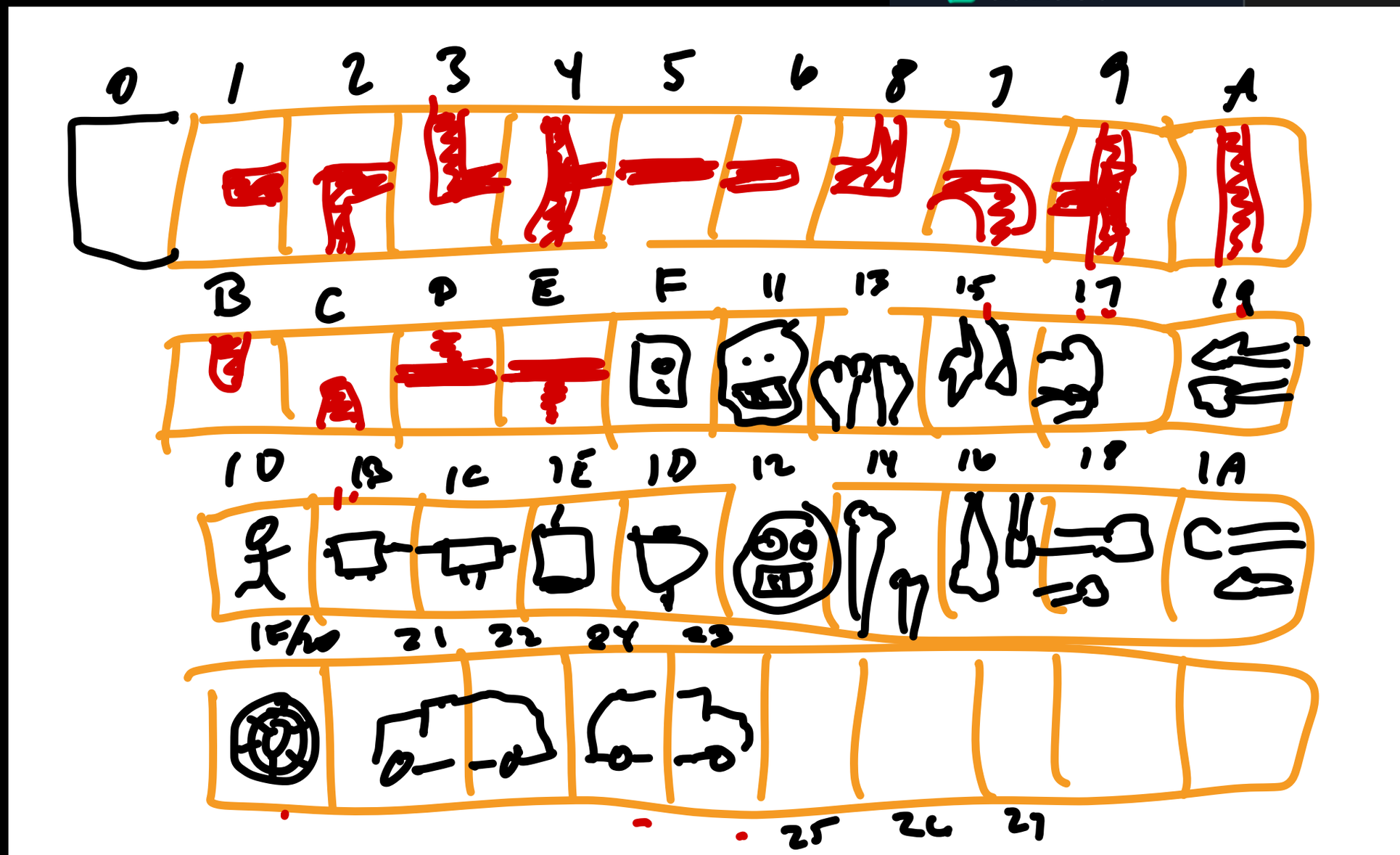
- Smooth scrolling only functions for "HIRES" display modes.
- Surprisingly, not for text modes. So it cannot be used to make a smoothly scrolling terminal.
- MAME will do it for text modes too, but scrolls things SIDEWAYS in text mode. Real hardware does not do this. Also means I need to turn off smooth scroll in non-hires regions.

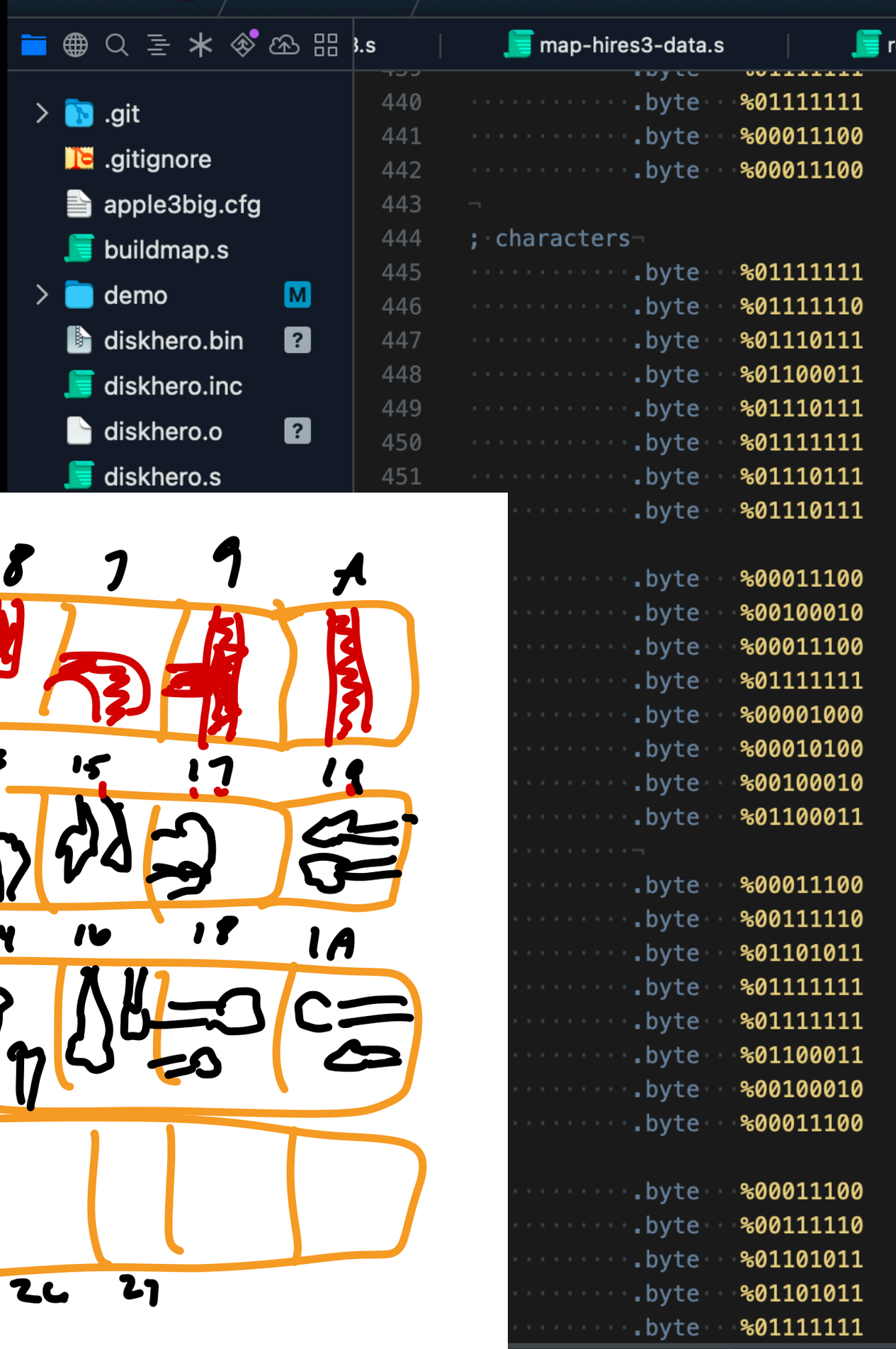
VIDEO MODES

APPLE III

	OUTPUTS	H6 (9334) pin 7	H6 (9334) pin 5	H6 (9334) pin 4	.6 (9334) pin 6
	INPUTS TO ROM G5 (0032)	ROM G5 pin 2	ROM G5 pin 1	ROM G5 pin 23	ROM G5 pin 4
PAGE 1		Ø 1 C056/C057 HIRES	Ø 1 C052/C053 MIX	Ø 1 C050/C051 GR/TEXT	Ø 1 C054/C055 GP1~2
40 CHAR AII (B/W)		Ø	Ø	Ø	Ø
40 CHARSARA (Color)		Ø	Ø	1	Ø
80 CHAR (B/W)		Ø	1	Ø	Ø
80 CHAR (B/W)		Ø	1	1	Ø
AII HIRES (250x192, B/W)		1	Ø	Ø	Ø
FGD/BKGD HIRES (280x192, 16 colours)		1	Ø	1	Ø
SUPER HIRES (560x192, B/W)		1	1	Ø	Ø
140 x192 AHIRES (140x192, color)		1	1	1	Ø

CUSTOM FONT


0 1 2 3 4 5 6 8 7 9 A
B C D E F 11 13 15 17 19
1D 1B 1C 1E 1D 12 14 16 18 1A
1F 21 22 24 23
25 26 27


map-hires3-data.s
.git
.gitignore
apple3big.cfg
buildmap.s
demo
diskhero.bin
diskhero.inc
diskhero.o
diskhero.s
; characters
.byte %01111111
.byte %00011100
.byte %00011100
.byte %01111111
.byte %01111110
.byte %01110111
.byte %01100011
.byte %01110111
.byte %01111111
.byte %01110111
.byte %01110111
.byte %00011100
.byte %00100010
.byte %00011100
.byte %01111111
.byte %00001000
.byte %00010100
.byte %00100010
.byte %01100011
.byte %00011100
.byte %00111110
.byte %01101011
.byte %01111111
.byte %01111111
.byte %01100011
.byte %00100010
.byte %00011100
.byte %00011100
.byte %00111110
.byte %01101011
.byte %01101011
.byte %01111111

CHANGING THE FONT

- Lots of articles and manuals talk about changing the font. But not really how.

- Apple provides a way using a driver that lets you point at the font data. That's pretty much the only way anyone talks about. I am not using drivers.

- The font data does not live in addressable RAM. You cannot change it on the fly. You have to stage it into some hidden RAM space by putting in special memory areas and telling the Apple /// to start loading the data.

CHANGING THE FONT

- Though I do not know all the details, the transfer clearly happens by leveraging the scan through video memory.
- The procedure is to place 8 characters' worth of data into the text page screen holes, turn on the "scan font data" switch, and wait for at least one full VBL cycle (when the video data has all been transferred at least once). Then, move on to the next set of 8 characters.
- You have 128 characters (the other 128 are inverse).

SCREEN HOLES

- Each triplet of lines on the text page has a "screen hole." That is because each line is 40 characters wide, but each 128 byte boundary is lined up to the left edge. So:
  - first line: 0 to 39
  - second line (down the screen a ways): 40-79
  - third line (down the screen further): 80-119
  - not displayed (screen holes): 120-127
- More concretely, screen holes are:
  - $478-47F, $578-57F, $678-67F, $778-77F
  - $4F8-4FF, $5F8-5FF, $6F8-6FF, $7F8-7FF
- Data in here will not affect the screen as it is being drawn.

FONT DATA

- We have 64 bytes of screen hole space (8 chunks of 8 bytes). That is enough to hold font data for 8 characters (font data is 7x8, high bit is unused).

- You'd think that each screen hole would contain one character.

- Or maybe that they'd be organized down the screen, with the 8th character in the 8th byte, 7th in the 7th, etc., with the first raster line in the first screen hole.

- But it is neither of these things. Nor did I find it written down anywhere. The monitor ROM sets up the characters at startup, but the data is compressed in a way that makes it opaque. I determined the actual layout basically through trial and error.

FONT DATA LAYOUT

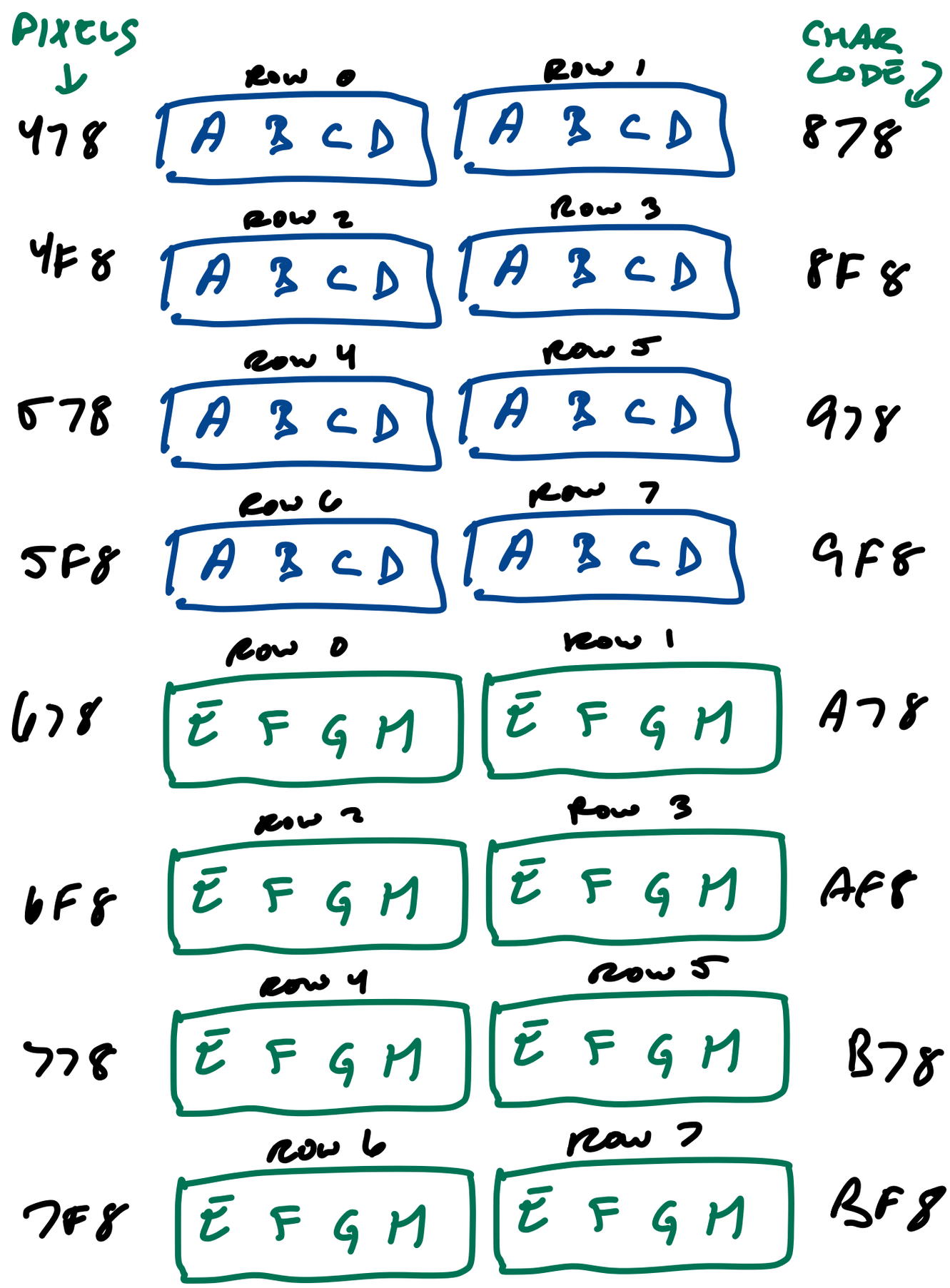
- In fact, each screen hole contains two bytes of four different characters. First 4 screen holes contain the first four characters, last 4 screen holes contain the last four characters. Because Apple can always find a way to interleave things just one more level.

- The screen holes in page 1 ($400) contain the character data. The screen holes in page 2 ($800) contain (very redundant) index data indicating which character this is. So if you are updating character $01, you have to put 8 $01 values in the page 2 addresses corresponding to the 8 addresses holding the pixel data.

  - It is likely that this system is "dumb" and you could scatter different characters' data around somewhat differently (like: have the last bytes of the first screen hole contain the first line of character $02 and the second line of character $03, though it is hard to see why you would).

CUSTOM FONT DATA

- Put data in screen holes.
- Touch $C0DB to turn on font transfer
- Wait for a full VBL cycle (two interrupts)
- Touch $C0DA to turn off font transfer
- Anything you don't change doesn't change (stays as monitor ROM or SOS set it up), doesn't hurt anything to change something to the same thing it was.



6-BIT AUDIO DAC

- Often mentioned, but with very little information on how it works.
- PB of the FFEx VIA is accessed by $FFE0. What this is telling us is that the lower 6 bits of what you write to $FFE0 go to the DAC.
- Incidentally, the PB6/IO Count Line turns out to be the HBL. Not written down ANYWHERE I don't think. I only know this HBL trick from Rob Justice's observations of what Atomic Defense was doing.

PB Port Description

The first 6 lines of the B port are configured to be outputs. They are inputs to the sound Generator.

- o The tone generated at the speaker can be varied by changing the bit values of these lines.
- o There are 127 possible tone combinations; the missing one turns the tone off completely.

PB6 is connected to the I/O Count line. Depending on the device in the slots, the VIA may be programmed to count a certain number of pulses generated or to determine that only one pulse occurred. Either way, the VIA will generate an IRQ and set the appropriate bit flag.

The last bit is used to monitor the NMI (Non Maskable Interrupt) line generated by the devices in the I/O slots.

WHAT IS AN AUDIO DAC ANYWAY?

- I had no idea how this was supposed to work. What do those 6 bits encode?

- My current guess: It seems to encode essentially an amplitude, and it's fairly straightforward. You produce a sine wave by raising its value up and then down in a sinusoidal pattern. Do that repeatedly and the frequency dictates the pitch.

- The document says there are 127 different possible tone combinations, but I can only count to 64 with 6 bits. This is a typo right? Or maybe I still don't quite understand what those 6 bits do.

RUNNING AUDIO

- It appears that the basics of running audio is pretty simple. You just put the amplitude into the audio register. But if you want to have a reasonable pitch range and smooth sounding audio, you need to do that OFTEN and REGULARLY.

- An interrupt sounds perfect for this, except that it takes so long to get in and out of an interrupt handler there's no time left for anything else.

Unfortunately, according to the **SOS Device Driver Writer's Guide**, the minimum response time to call an interrupt handler is about 160 microseconds, and another 115 microseconds is required to return from the interrupt handler to whatever was happening before the interrupt occurred. So even though my interrupt handler takes only 15 microseconds to execute, the total time required to update the speaker is 290 microseconds. Since the voltage on the speaker has to be changed twice per audio cycle, a sound with a frequency of about 1700 Hz consumes ALL of the computer's processing time—not exactly an improvement.

ON THREE

/// /// /// /// /// /// /// /// ///

/// to the Max #4 ...And Things That Go GLEEP In The Night

by Al Evans

PIGGYBACKING AUDIO

- I do have a regularly-firing interrupt, though. It goes off with certain HBLs in order to do the display mode switches. Since I'm in there anyway, may as well update the audio then.

- Originally, I had the interrupts fire only when I needed to switch, but changed it so that it will fire every 8 scan lines, whether a switch is needed or not. That makes it regular.

- Except during VBL, which covers the same time as 70 scan lines (or 8-9 audio samples). So, during VBL I set up a counting timer that goes off at approximately the same rate, and do just the audio. Costly, but workable.

GAMEPLAY

- After all the technical stuff, also needed to address gameplay. How to move? How to make the hoarders move? How to keep track of score?

- Keyboard control is pretty simple, keyboard generates an interrupt, which stores the pressed key somewhere the main code can read it.

- The basic game is mostly event driven, sitting in a loop that just waits to see if it is supposed to quit, redrawing the playfield and the score.

- The VBL generates the game clock that triggers characters to move.

DEVELOPMENT

- When I started this, I also didn't know anything about ca65, but I used that as my cross-assembler. Just from the command line, I wasn't about to also learn how to use Xcode. I may not be using it particularly correctly. There are probably fancier things I can do with the memory and segment configuration.

- The program is all in one binary file. Stored to the disk as SOS.INTERP which gets it to boot. Although A000-B7FF should be safe (though not endorsed by Apple), the program got big enough I had to move it down. Meaning that I had to put all the execute-once setup stuff early, so it was ok for a bank to switch overtop of it.

- All it takes to make a bootable program is to compile the binary file to SOS.INTERP (which includes a header) and store it on a disk (i.e. with AppleCommander).

PROGRAMMING NOTES

- Using a standard lookup table for hires Y-coordinates, I generally set the ZP to the page in graphics memory, and then wrote to graphics with ZP opcodes, leaving the stack at $100 so that it wouldn't also be onscreen.

- Using the stack to push to graphics memory can save some cycles over using ZP, but then you are limited to using just $78-7F and $F8-FF in ZP (landing in screen holes), since if the stack is in graphics memory, ZP will also be.

- It would be convenient if extended addressing worked, so you could read from bank 2 and push into graphics memory—but it doesn't. Extended addressing requires ZP and stack to be between 18-1F, not on the graphics pages.

PROGRAMMING NOTES

- The colors MAME produces don't really match the ones I see on the Color Monitor 100. The colors look better on the real Apple ///.

- Also, MAME runs faster, discernibly. The audio sounds better in MAME.

- Which is really because MAME is operating "too well." It's going too fast, the real Apple /// slows down to 1MHz 73% of the time, whereas MAME just plows ahead at 2MHz. Meaning that fixing MAME is kind of adding code to make it worse. Which I guess is kind of fitting for Apple /// emulation.

CURRENT STATUS

- This is the sort of thing that always could be more finished.
- Not bug-free yet at time of recording. Traveling downward can sometimes lead to a crash/hang, and occasionally the audio sound effects overpower the mode switching and the screen briefly displays garbage.
- Not really a way to win or lose yet. The hoarders are supposed to head for the disk that's closest to them, but it's not clear that they do. That needs to work for the ability to drop a distractor disk to work. The map might be too large to be fun, maybe should start significantly smaller.
- Would be nice to think of something to do with the lower medres region apart from showing a grassy pattern.

FINDING IT

- The code will be available on github to look at (under account paulhagstrom).
- Will try to see if the MAME-in-a-browser on the Internet Archive will allow this to be played without installing anything.
- Part of the point of doing this was to provide an example of how the various Apple /// technologies could be used. To help future others or future me, by having at least something that shows how these things are done. So not everyone needs to keep banging their heads against the level 2 service reference manual.