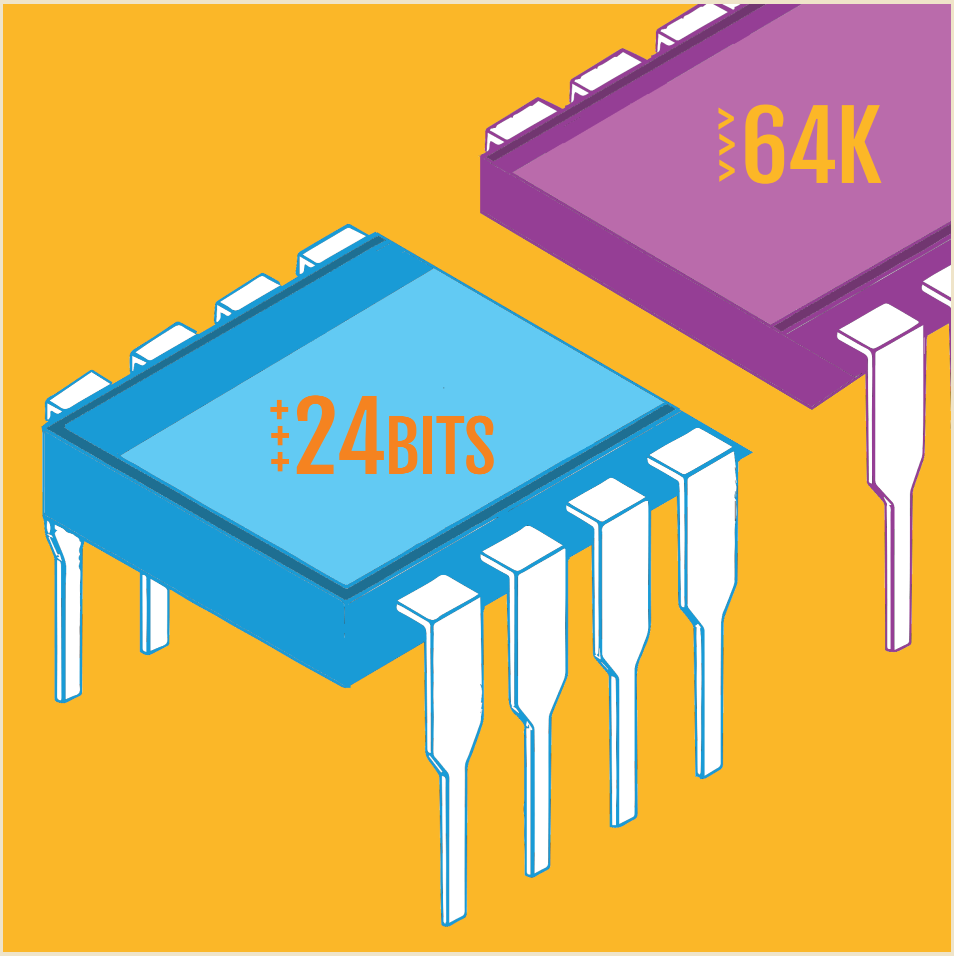
Apple II

Inside the apple][4



Inside the Apple II 4

APPLE Product # D2L0024
950-0018

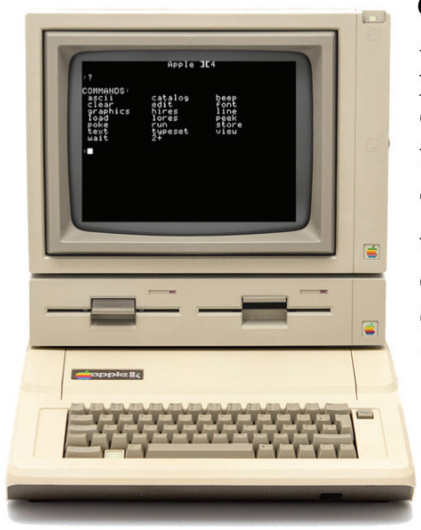
Printed in USA

What If?

What if "Apple II Forever" was true as of 1979, with the II Plus just the first in a long series of II model updates?

What if the 6502 was the first in a road map of updates that saw the 65xxx as the leading CPU of the 1980s?

What if there was a Markkula's Law, like Moore's Law, predicting the amount of RAM per year, avoiding a decade of banking memory to overcome the 64K limit of the 8-bit computer era?



```
              Apple ][4
:?
COMMANDS:
 ascii        catalog     beep
 clear        edit        font
 graphics     hires       line
 load         lores       peek
 poke         run         store
 text         typeset     view
 wait         2+
:█
```

What if Synertek had the capital and resources to produce a 24-bit address 6502 variant by 1980? And the foresight to partner with Apple for custom chips to optimize the II internals?

The result could have been the Apple II4, an 8-bit data / 24-bit address 652402 microprocessor with 96K-128K of RAM.

With a built-in, four-chip Disk II4 controller, complete with its own 6507, ROM, RAM, and Integrated Woz Machine, all driving double sided floppy drives.

Inside the Apple II4 tells that "what if" story, showing off what could have been, if only for a few changes to the timeline.

Best of all, this book isn't fan non-fiction as a manual to neverware, but instead nearly everything in this manual is realized in code, with every screenshot taken from an Apple II4 emulator, running hand-coded assembly running on an emulated 652402, all developed on a modern Apple M1-powered Mac Air.

github.com/lunarmobiscuit/izapple2, iz6502, and aCCembler

TABLE OF CONTENTS

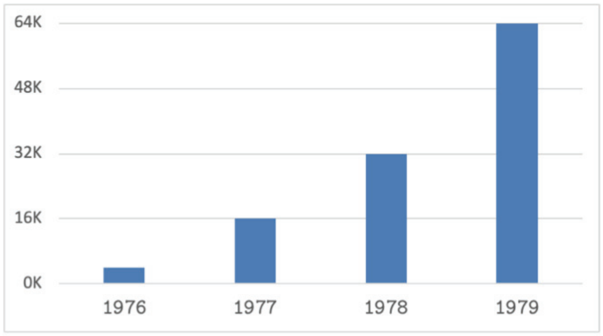
CHAPTER 1

BEYOND 64K

In 1965, Gordon Moore, one of the co-founders of Fairchild and Intel looked at the average number of transistors integrated on the computer chip, plotted out how that had changed over the previous eight years, and noticed that the number of transistors had been doubling every 18 months.

In 1978, Steve Jobs, co-founder of Apple, met with Gordon Moore and asked him how "Moore's Law" would apply to random access memory chips (a.k.a. RAM), one of Intel's main products. Moore asked Jobs how much RAM was included in each model of Apple computer.

The original Apple Computer kit in 1976 typically included just 4K or 8K of RAM. The Apple II, a year later, could manage between 4K and 48K, and by the end of 1977 16K was considered a small amount of memory. Two years later, 1979, the Apple II Plus shipped standard with 48K of RAM, with an additional 16K often added on through the Language Card, plus another 1K on the 80-column card.

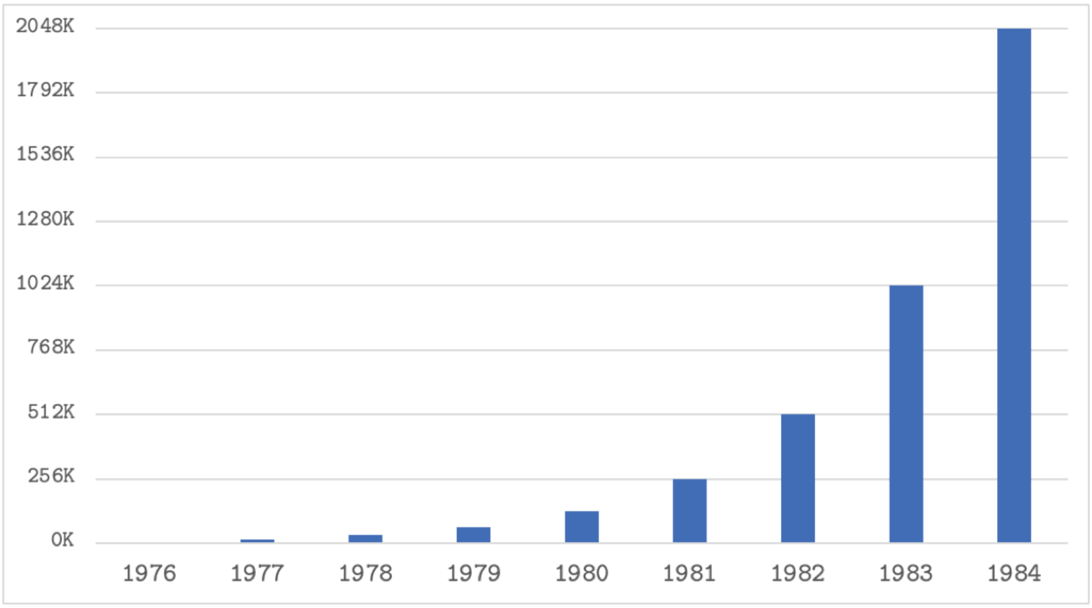


Common amounts of RAM

Plotting out these values made it obvious that RAM was growing even faster than Moore's general rule on transistors, doubling every 18 months.

If this trend continues through the 1980s, the 1984 Apple computer will include 2,048K of RAM, a.k.a. 2 megabytes. Even if the trend slows, by the end of the 1980s computers will have many megabytes of RAM, far more than the 64K that the original Apple II was designed, at most, to manage.

Which is why the Apple II4 has been designed to manage up to 16,384K of RAM (a.k.a. 16 megabytes), crashing through the 64K limit set not just the Apple II, but by all other microcomputers too.



Expected amounts of RAM, per computer

CHAPTER 2

THE 652402

The computer chip that powers all the previous Apple computers is the 6502. This chip was the breakthrough that made microcomputers affordable. The one big limitation of this chip is that it could manage only up to 64K of memory.

To overcome this barrier, the Apple II4 uses the Apple co-designed 652402 microprocessor. This chip is fully backward compatible with the 6502, allowing the Apple II4 to use any of the thousands of programs available for the Apple II and II Plus. This chip has just one new design feature, managing up to 16MB (M = megabyte = 1,024K) of memory.

Plus... the 652402 is twice as fast as the 6502, running at 2MHz, i.e. able to run 2 MILLION operations per second.

The INSIDE Story on the 652402

After that meeting with Gordon Moore, Steve Jobs set off on a mission to ensure future Apple computers would not be held back by the 64K limit embedded within the 6502's design. It didn't take long for Steve to find a sympathetic ear in Bob Schreiner, CEO of Synertek, the company that manufactures all of Apple's 6502s.

In their first meeting, Steve and Bob agreed on a design that would, as simply as possible, expand the 6502 from 16-bit addresses (64K) to 24-bit addresses (16,384K). Bill Mensch and a few of the original designers of the 6502 had recently joined Synertek. With their experience the 652402 was designed and ready within just nine months of that fateful meeting.

That same meeting created a plan for future microprocessor upgrades, to build upon this design partnership into the future, ensuring that the Apple computers would continue be at the forefront of personal and business computing.

Look for those changes in next year's Apple computers, as Steve Jobs mission includes shipping a new, improved computer EVERY YEAR, which is why the Apple II4 is here just a year after the Apple II Plus.

The 652402 includes all the same opcodes as the 6502:

```
ADC, AND, ASL, BCC, BCS, BEQ, BIT, BMI, BNE, BPL, BRK, BVC, BVS,
CLC, CLD, CLI, CLV, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX,
INY, JMP, JSR, LDA, LDX, LDY, LSR, NOP, ORA, PHA, PHP, PLA, PLP,
ROL, ROR, RTI, RTS, SBC, SEC, SED, SEI, STA, STX, STY, TAX, TAY,
TSX, TXA, TXS, TYA
```

All of instructions work exactly the same on the 652402 as the 6502, including all of the machine codes, the number of cycles, the instruction lengths, etc. One (and only one) opcode has been added. This allows the microprocessor to access memory using 24-bit addresses. The mnemonic for that operation is A24 (using the otherwise unused machine opcode $1F).

This single-byte opcode works differently from all the others. By itself it does nothing. But when placed before another opcode, it changes the behavior of that next opcode. If that opcode on the 6502 included two bytes to specify the address (a 16-bit address), then after an A24 that opcode requires three bytes (a 24-bit address).

To process this extra byte, the 652402 uses one extra clock cycle. For example, LDA $abcd is a 3-byte instruction that runs in 4 clock cycles. A24 LDA $abcdef is a 5-byte instruction that runs in 5 clock cycles. Similarly, JSR $abcd is a 3-byte instruction that runs in 6 clock cycles, while A24 JSR $abcdef is a 5-byte instruction that runs in 8 clock cycles. One extra cycle to read the extra byte, and one extra cycle to push a 24-bit return address on the stack.

If an opcode doesn't work with memory addresses, then A24 does nothing, and is ignored.

This style of opcode modifier is not part of the 6502 instruction set, but it is not novel. Other microprocessors have modifier opcodes and there they are called "prefix codes".

The A24 prefix code is how the 652402 is 100% backward compatible with the 6502. If the assembly code doesn't use the A24 mnemonic, then the microprocessor uses 16-bit addresses and behaves exactly like a 6502. 100%, cycle-by-cycle backward compatibility.

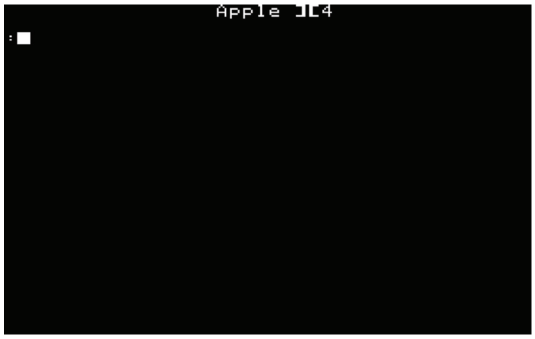
The only caveat for programmers to remember when using this prefix code is that A24 JSR $abcdef requires a matching A24 RTS, as A24 JSR $abcdef pushes a 24-bit return address while A24 RTS pops a 24-bit return address.

CHAPTER 3

OS4 & THE COMMAND LINE

The other big change between the Apple II4 previous Apple II computers is the more powerful, OS4 operating system and its new and improved command line. This follows the progression of previous Apple computers, from the Monitor in the original Apple kit to Integer BASIC in the Apple II to Applesoft BASIC and DOS 3.3 in the Apple II Plus.

When the Apple II4 is powered up (or reset), everything it needs to run the new command line is included in the new OS4 ROM.

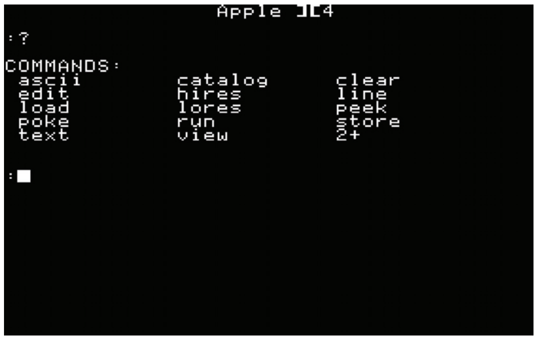


```
            Apple ][4
:
```

As soon as you turn on your Apple, the screen will clear, “Apple][4” will be drawn across the top of the screen, and the ":" prompt will be drawn, with a flashing '_' cursor ready for your command.

You don’t need to "boot" up the disk drive, hit reset, or do anything to get started.

To help you remember the various commands, type ‘?’ followed by RETURN. ***Note: you just type the ?, all by itself, with no quotes, as shown in the screen shot below.***



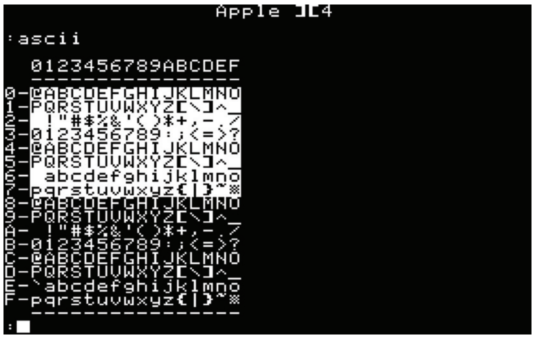
```
            Apple ][4
:?
COMMANDS:
 ascii       catalog     clear
 edit        hires       line
 load        lores       peek
 poke        run         store
 text        view        2+

:
```

The question mark command (‘?’) lists all the valid commands.

As you will see later in this manual, there will be other command lines with other prompts, and ‘?’ will similarly show which commands are valid for those prompts.

The first command on the list is “ascii”. Try it.

Type the letters a s c i i and then then press the RETURN key. If you are used to the Apple II, you might notice that there are both CAPITAL and lowercase letters on the Apple II4. The command line doesn’t care which you use. You press and hold the SHIFT key to type the uppercase letter. If you don’t press the SHIFT key, the letters you type are lowercase.

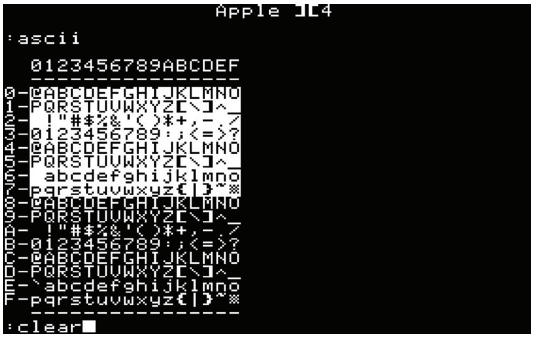


```
                 Apple ][4
:ascii
  0123456789ABCDEF
  ----------------
0-@ABCDEFGHIJKLMNO
1-PQRSTUVWXYZ[\]^_
2- !"#$%&'()*+,-./
3-0123456789:;<=>?
4-@ABCDEFGHIJKLMNO
5-PQRSTUVWXYZ[\]^_
6- abcdefghijklmno
7-pqrstuvwxyz{|}~▒
8-@ABCDEFGHIJKLMNO
9-PQRSTUVWXYZ[\]^_
A- !"#$%&'()*+,-./
B-0123456789:;<=>?
C-@ABCDEFGHIJKLMNO
D-PQRSTUVWXYZ[\]^_
E-`abcdefghijklmno
F-pqrstuvwxyz{|}~▒
  ----------------
:█
```

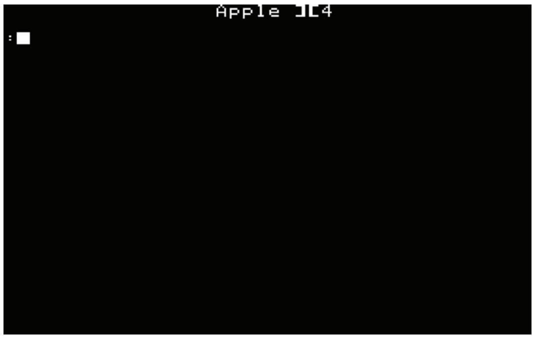
The ‘‘ascii’’ command displays all the letters, numbers, and symbols that the Apple II4 can display. The funny name comes from the American Standard for Character Interchange, a.k.a. ASCII.

Like the Apple II, the Apple II4 normally displays text as white dots on a black background, but it can also display ‘‘inverted’’ text as black dots on a white background, or ‘‘flash’’ the text between normal and inverted.

The next command to try is ‘‘clear’’. Type: c l e a r RETURN



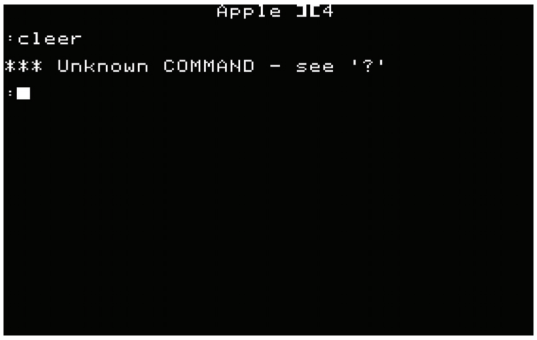
```
                 Apple ][4
:ascii
  0123456789ABCDEF
  ----------------
0-@ABCDEFGHIJKLMNO
1-PQRSTUVWXYZ[\]^_
2- !"#$%&'()*+,-./
3-0123456789:;<=>?
4-@ABCDEFGHIJKLMNO
5-PQRSTUVWXYZ[\]^_
6- abcdefghijklmno
7-pqrstuvwxyz{|}~▒
8-@ABCDEFGHIJKLMNO
9-PQRSTUVWXYZ[\]^_
A- !"#$%&'()*+,-./
B-0123456789:;<=>?
C-@ABCDEFGHIJKLMNO
D-PQRSTUVWXYZ[\]^_
E-`abcdefghijklmno
F-pqrstuvwxyz{|}~▒
  ----------------
:clear█
```



```
                 Apple ][4
:█
```

This command clears the screen. Mostly. The ‘‘Apple][4’’ stays visible at the top. Later you’ll see how that title changes when you run another program. You’ll also see later how the ‘‘clear’’ command is smart, clearing the graphics screens when you are displaying graphics or the text screen when viewing text.

Feel free to try out the other commands too. There is no way to break or harm your Apple computer by typing a command. The worst case is you’ll get an *** UNKNOWN ERROR message, which will remind you to use the ‘?’ command to help you figure out what you misspelled.



```
                 Apple ][4
:cleer
*** Unknown COMMAND - see '?'
:█
```

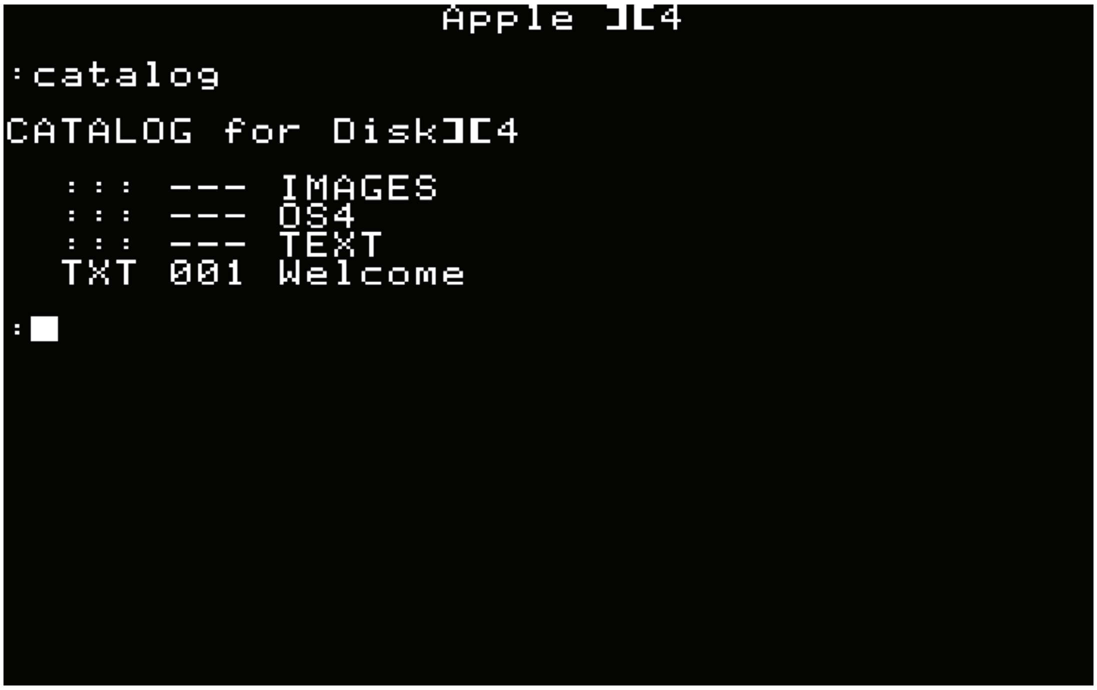
For example, ‘‘cleer’’ is not a valid command, as it is misspelled.

To help you further, the command line can “auto complete” your commands. Type just the ‘a’ key and then press the key labeled TAB. The Apple II4 will fill in “ascii” on the command line for you.

Try it with some other letters too. Or multiple letters. Type c l TAB and the Apple will fill in “clear” command.

The Apple II4 will also remember the previous two commands you typed, and you can re-enter them by pressing the UP ARROW key.

And, of course, sometimes you need to edit your commands before you hit RETURN, and you can do that with either the DELETE key or the left arrow key. Both will delete the last character on the command line.



```
                    Apple ][4
:catalog
CATALOG for Disk][4

  ::: --- IMAGES
  ::: --- OS4
  ::: --- TEXT
  TXT 001 Welcome

:
```

CHAPTER 4

DISK II4 & DOS4

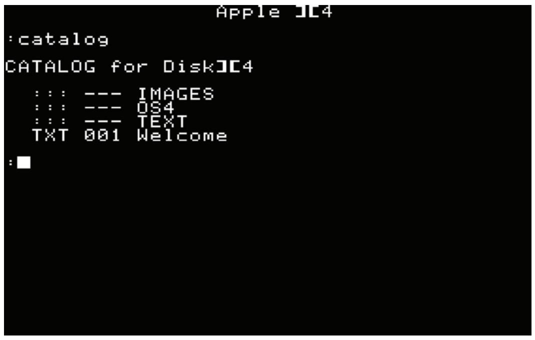
Disk][and DOS 3.2 were groundbreaking products when they were released back in 1978. The Disk][floppy drives were fast, held more storage than their competitors, and were a lot more affordable too.

The Apple II4 builds on that legacy with the new Disk II4 floppy disk drives and floppy disk control board. What is new? The floppy drives are now double sided, able to store 284K per disk, and the floppy drives now have their own 6502-like microprocessor, which greatly speeds up reading and writing, as well as the next command “catalog”.

The INSIDE Story on the Disk II4

In that fateful meeting of Steve Jobs and Bob Schreiner (CEO of Synertek) the two men didn’t just talk about the main microprocessor, they also talked about all the other logic chips in the Apple II, and together created a plan on greatly shrinking the number of chips in the computer by integrating those dozens of support chips into just a handful of chips.

One of those chips manages the larger 24-bit address space, implementing the new memory map (discussed in Chapter 6). Another simplifies the floppy disk logic invented by Steve Wozniak (Apple’s other co-founder) into a single chip, and couples that “Integrated Woz Machine” with a Synertek 6507 microprocessor, which is a lower-cost, simplified version of the 6502.



```
                Apple ][4
:catalog
CATALOG for Disk][4
  ::: --- IMAGES
  ::: --- OS4
  ::: --- TEXT
  TXT 001 Welcome
:■
```

Put the System disk into your Disk II4 floppy drive, then type: c a t a l o g RETURN, or more simply, c TAB RETURN. Almost instantly you will see a listing from that floppy.

What you might have noticed is that as soon as you closed the latch on the floppy drive, it spun up for a few seconds. That was the Disk II4 microprocessor reading the catalog of the floppy before you started typing the command. It was able to do that without the cursor even flickering because of that extra microprocessor.

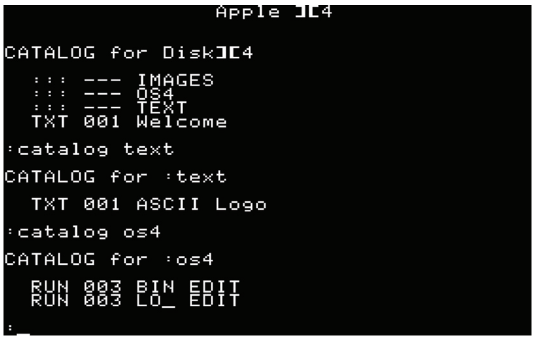
The CATALOG display will look familiar to anyone who has used DOS 3.2 or 3.3 on the Apple II or II Plus, but with a few small differences.

First, each floppy disk has a name. The name of the System disk isn't "System", it is "Disk][4", as seen in line "CATALOG for Disk][4".

Second, each entry has a three character file type instead of just one.

Third, the file size looks similar to DOS 3.2/3.3, but in OS4 this is the size in K rather than the number of sectors. OS4 is designed to support hard disks and other storage systems and hides sectors and other floppy-specific details.

Back to the file type, note how a few of the entries on the System disk have file type ":::" (three colons). These are sub-directories. In DOS4, you can organize your files into sub-directories. For simplicity, only the top-level catalog can have subdirectories.

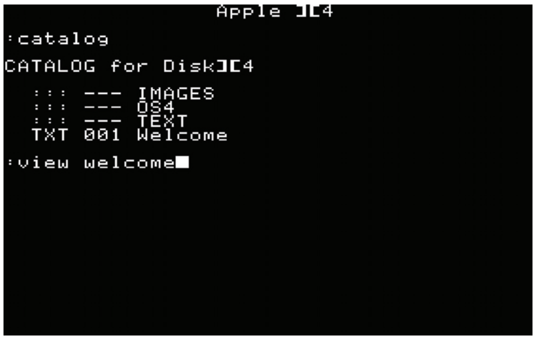


```
                Apple ][4
CATALOG for Disk][4
  ::: --- IMAGES
  ::: --- OS4
  ::: --- TEXT
  TXT 001 Welcome
:catalog text
CATALOG for :text
  TXT 001 ASCII Logo
:catalog os4
CATALOG for :os4
  RUN 003 BIN EDIT
  RUN 003 LO_ EDIT
:_
```

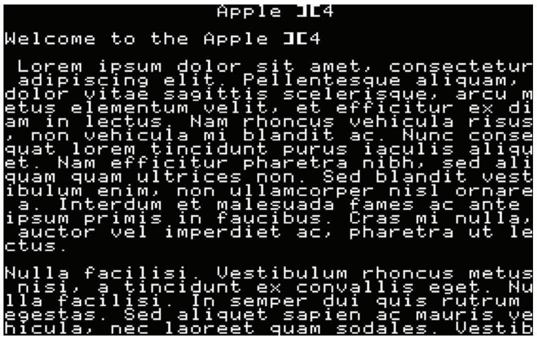
You can view the contents of these sub-directors by including their name in a "catalog" command, e.g., "catalog text" to see the files in the "text" directory, or "catalog os4" to see the to see the files in the OS4 directory.

The catalog command doesn't care if you type UPPERCASE or lowercase letters. All the commands on the Apple II4 are "case insensitive".

On the top-level catalog and in the TEXT directory you will find files with the TXT file type. These are text files. To view the "Welcome" text file, type "view welcome"



```
                Apple ][4
:catalog
CATALOG for Disk][4
  ::: --- IMAGES
  ::: --- OS4
  ::: --- TEXT
  TXT 001 Welcome
:view welcome
```

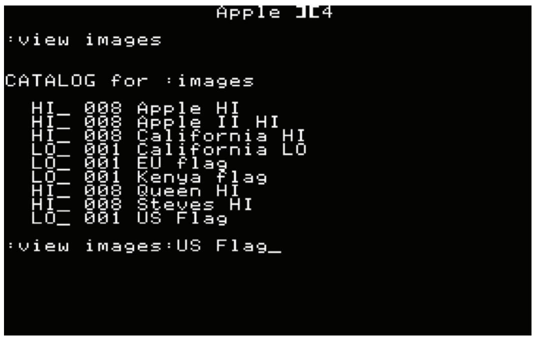


```
                Apple ][4
Welcome to the Apple ][4
 Lorem ipsum dolor sit amet, consectetur
 adipiscing elit. Pellentesque aliquam,
dolor vitae sagittis scelerisque, arcu m
etus elementum velit, et efficitur ex di
am in lectus. Nam rhoncus vehicula risus
, non vehicula mi blandit ac. Nunc conse
quat lorem tincidunt purus iaculis aliqu
et. Nam efficitur pharetra nibh, sed ali
quam quam ultrices non. Sed blandit vest
ibulum enim, non ullamcorper nisl ornare
 a. Interdum et malesuada fames ac ante
ipsum primis in faucibus. Cras mi nulla,
 auctor vel imperdiet ac, pharetra ut le
ctus.
Nulla facilisi. Vestibulum rhoncus metus
 nisi, a tincidunt ex convallis eget. Nu
lla facilisi. In semper dui quis rutrum
egestas. Sed aliquet sapien ac mauris ve
hicula, nec laoreet quam sodales. Vestib
```

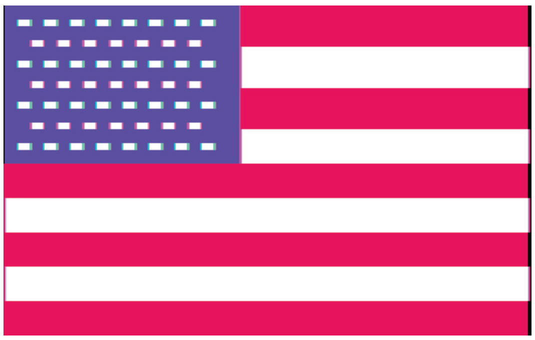
The contents of the file will display on the screen. This file is too long to fit on one screen, so once one screenful of information displays, it stops. Press any key to show the next screen. Repeat screen by screen until the end of the file to view the whole file.

Like on the Apple II and II Plus, if you want to stop in the middle of a file, press Control-C (hold down CONTROL and press C) or press the ESC key (a.k.a. the ESCAPE key).

The "view" command is much more powerful than just displaying text. Type "view images". "IMAGES" isn't a file, it is a sub-directory. The "view" command noticed the ":::" file type and ran the command "catalog images", as "catalog" is how you view a directory.



```
                Apple ][4
:view images

CATALOG for :images

  HI_ 008 Apple HI
  HI_ 008 Apple II HI
  HI_ 008 California HI
  LO_ 001 California LO
  LO_ 001 EU flag
  LO_ 001 Kenya flag
  HI_ 008 Queen HI
  HI_ 008 Steves HI
  LO_ 001 US Flag

:view images:US Flag_
```



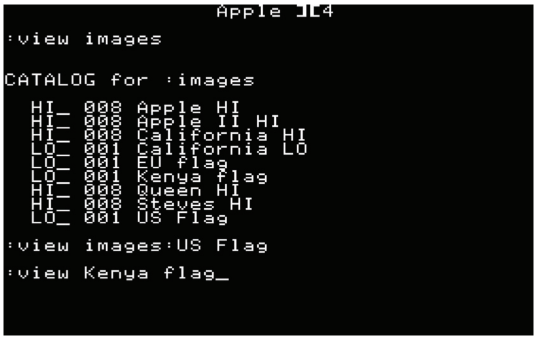
Even more impressive is when you type "view images:US Flag" followed by RETURN. View noticed that the "US Flag" file in the "IMAGES" directory was a LORES graphics image (file type "LO_"). The view command was smart, loading that image into the LORES graphics memory, and it changed the settings to display the second LORES graphics page. ***(Why the second LORES page? That will be covered by Chapter 6: II4 MEMORY MAP).***

Getting back to the TEXT page is simple. Press the ESC key.

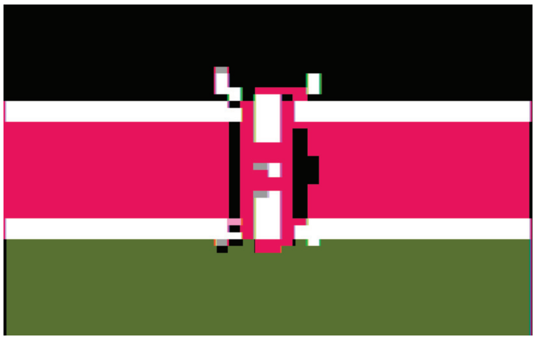
Press ESC one more time. The ESC key will toggle between the graphics mode and text mode, making is quick and easy to use the command line when viewing (or editing) graphics.

Back to the "view images:US Flag" command. The syntax used here is the sub-directory name followed by a colon (':') followed by the file name in the sub-directory, with no spaces before or after the color. That is how you fully specify the file you are trying to view (or edit or otherwise modify).

That said, type "view Kenya flag", hit RETURN.

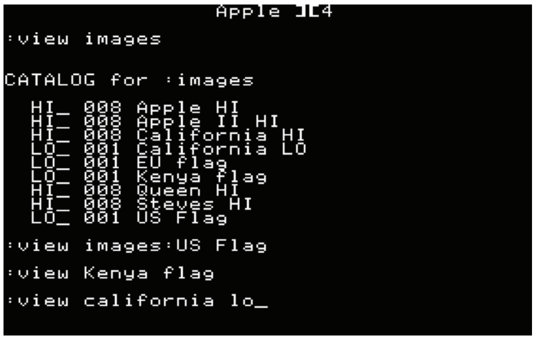


```
                Apple ][4
:view images

CATALOG for :images

  HI_ 008 Apple HI
  HI_ 008 Apple II HI
  HI_ 008 California HI
  LO_ 001 California LO
  LO_ 001 EU flag
  LO_ 001 Kenya flag
  HI_ 008 Queen HI
  HI_ 008 Steves HI
  LO_ 001 US Flag

:view images:US Flag

:view Kenya flag_
```

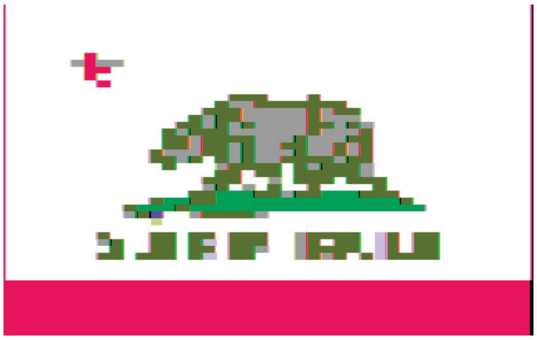


While DOS4 lets you create sub-directories, as long as the file name is unique, DOS4 will find it for you without requiring you to type the directory name and the colon.

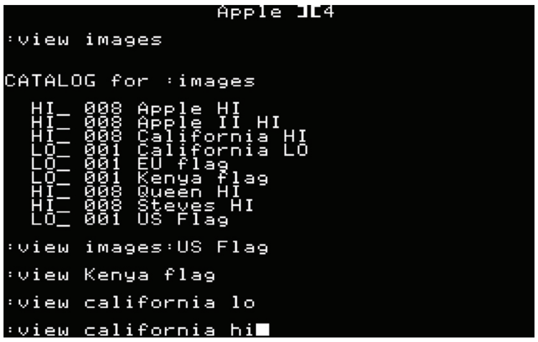
Another image to try is "view California LO"



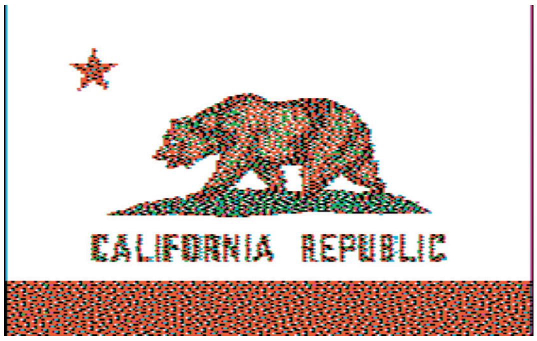
```
                Apple ][4
:view images

CATALOG for :images

  HI_ 008 Apple HI
  HI_ 008 Apple II HI
  HI_ 008 California HI
  LO_ 001 California LO
  LO_ 001 EU flag
  LO_ 001 Kenya flag
  HI_ 008 Queen HI
  HI_ 008 Steves HI
  LO_ 001 US Flag

:view images:US Flag

:view Kenya flag

:view california lo_
```



The LORES graphics resolution doesn't have enough pixels to clearly render the words CALIFORNIA REPUBLIC. But "view California HI" and a HIRES version of that flag will get loaded and displayed.



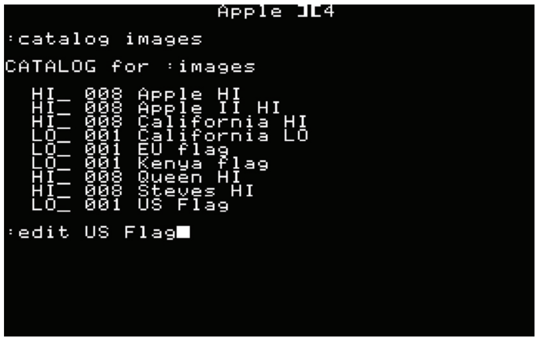
```
                Apple ][4
:view images

CATALOG for :images

  HI_ 008 Apple HI
  HI_ 008 Apple II HI
  HI_ 008 California HI
  LO_ 001 California LO
  LO_ 001 EU flag
  LO_ 001 Kenya flag
  HI_ 008 Queen HI
  HI_ 008 Steves HI
  LO_ 001 US Flag

:view images:US Flag

:view Kenya flag

:view california lo

:view california hi█
```



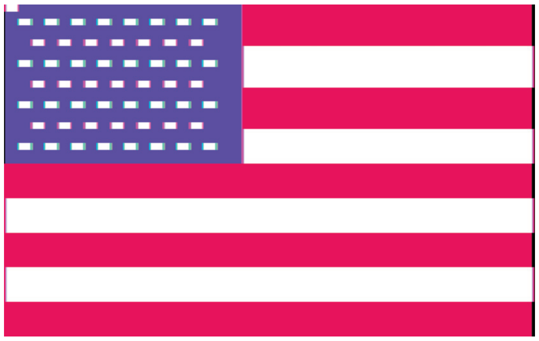
Again, the ESC key toggles between TEXT mode and in this case, HIRES graphics mode. If you want to flip between the two graphics, type the command "lores" to go back to the LORES graphic. If you do that while the HIRES graphic is loaded, then ESC will toggle between LORES and HIRES graphics modes.

In that case, to get back to TEXT mode, the command is "text".

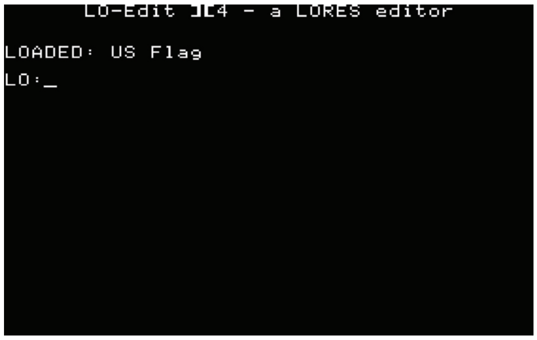
Now for a bit more fun, type "edit US Flag"



```
                Apple ][4
:catalog images
CATALOG for :images
  HI_ 008 Apple HI
  HI_ 008 Apple II HI
  HI_ 008 California HI
  LO_ 001 California LO
  LO_ 001 EU flag
  LO_ 001 Kenya flag
  HI_ 008 Queen HI
  HI_ 008 Steves HI
  LO_ 001 US Flag
:edit US Flag█
```



The LORES graphics file is shown like before, but there is an extra white dot in the upper left corner, and if you use the UP, DOWN, LEFT, and RIGHT buttons on the keyboard, that dot moves around the screen. That dot is the cursor for editing LORES graphics!



```
      LO-Edit ][4 - a LORES editor
LOADED: US Flag
LO:_
```

After moving the cursor around the screen, press ESC. Like with "view US Flag" the display will toggle back to the TEXT view, but look carefully and you'll notice a few differences.

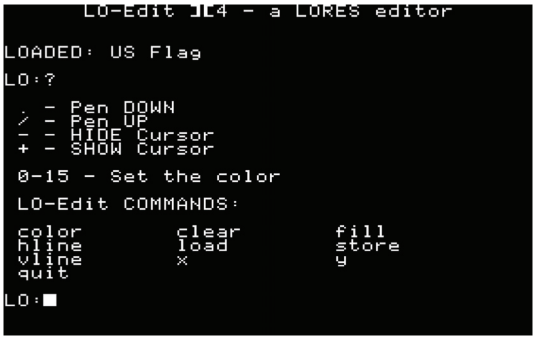
The prompt by the flashing cursor is now "LO:" instead of just ":" and the top of the screen doesn't say "Apple][4", it instead says "LO-Edit - a LORES editor". You loaded and started running a program from the System disk! Which program? "OS4:LO_ EDIT"

Just as the view command looked at the file type to figure out how to display the file, the edit command is similarly smart. But unlike view, edit doesn't have any built-in editors. Instead, it looks in the "OS4" subdirectory for a program named "XYZ EDIT" where "XYZ" matches the file type of the file you specified.

If you had instead typed "edit California HI", the results would have been "*** No EDIT program was found" as there is no program in the OS4 directory named "HI_ EDIT".

You can tell when a file is a program as the file type is "RUN"

Back to the LORES editor, type ? RETURN to see the list of commands that are valid in this program.



```
        LO-Edit ][4 - a LORES editor

LOADED: US Flag

LO:?

 . - Pen DOWN
 / - Pen UP
 - - HIDE Cursor
 + - SHOW Cursor

 0-15 - Set the color

 LO-Edit COMMANDS:

 color        clear        fill
 hline        load         store
 vline        x            y
 quit

LO:■
```

We said there would be other command lines, and here is the first example. You cannot run the catalog, view, or edit commands from this command line, but you can run other commands relevant to LORES graphics like: color, hline, vline, x, and y.

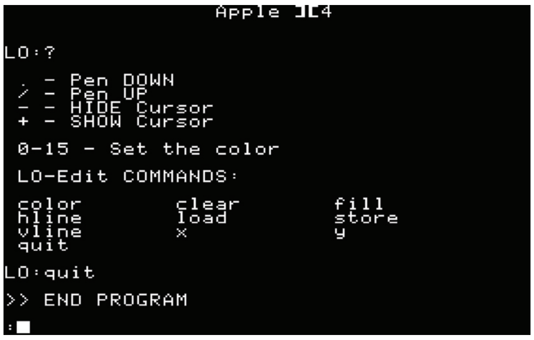
There are a few other keys that are special, displayed above the list of commands. When viewing the graphic, pressing '.' colors in pixels as you move the cursor, and '/' stops doing that. The '-' key hides the cursor, and the '+' shows it again (as does UP, DOWN, LEFT, or RIGHT).

Type "color 0" or "color 1" or any number up to "color 15" to set the color that is drawn when the "pen" is down. Or just type 0 RETURN, 9 RETURN, 1 1 RETURN, 1 5 RETURN, etc. to change the color without typing the word "color" first.

"x 10" moves the cursor to column 10. "y 24" moves the cursor to row 24.

"hline 5,20" draws a horizontal line on the row where the cursor is located from column 5 through column 20, in the current color. "vline 0,47" draws a vertical line from the top of the screen to the bottom, in the column where the cursor is located, in the current color.

"store my image" creates a new file named "my image" and saves the image into that file. "load California lo" loads and displays the LORES California flag image.

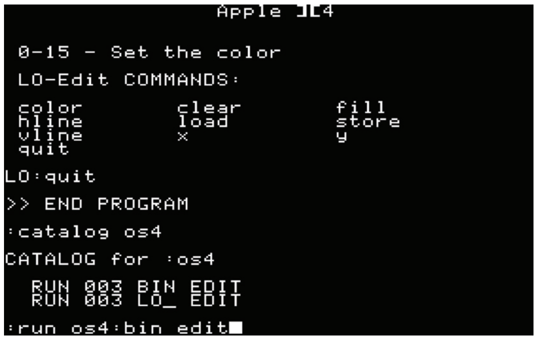


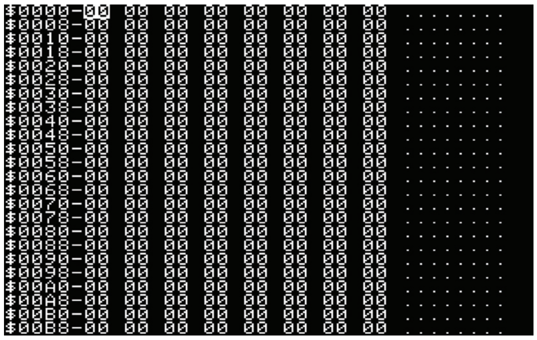
```
                Apple ][4

LO:?

 . - Pen DOWN
 / - Pen UP
 - - HIDE Cursor
 + - SHOW Cursor

 0-15 - Set the color

 LO-Edit COMMANDS:

 color        clear        fill
 hline        load         store
 vline        x            y
 quit

LO:quit

>> END PROGRAM

:■
```

When you are done editing, use the "quit" command to return to the OS4 command line.

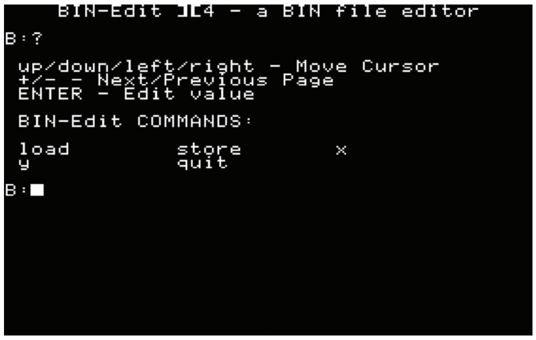
You'll know quit worked as the words ">> END PROGRAM" will be drawn on the screen, the prompt before the cursor will again be a ':', and "Apple][4" will appear at the top of the screen.

There is another program in the OS4 directory, “BIN EDIT”, but there are no “BIN” files to edit. To try out this program, type “run OS4:BIN EDIT”.


```
                Apple ][4

 0-15 - Set the color
 LO-Edit COMMANDS:

 color        clear        fill
 hline        load         store
 vline        x            y
 quit
LO:quit
>> END PROGRAM
:catalog os4
CATALOG for :os4
  RUN 003 BIN EDIT
  RUN 003 LO_ EDIT
:run os4:bin edit
```


```
$0000-00 00 00 00 00 00 00 00 ........
$0008-00 00 00 00 00 00 00 00 ........
$0010-00 00 00 00 00 00 00 00 ........
$0018-00 00 00 00 00 00 00 00 ........
$0020-00 00 00 00 00 00 00 00 ........
$0028-00 00 00 00 00 00 00 00 ........
$0030-00 00 00 00 00 00 00 00 ........
$0038-00 00 00 00 00 00 00 00 ........
$0040-00 00 00 00 00 00 00 00 ........
$0048-00 00 00 00 00 00 00 00 ........
$0050-00 00 00 00 00 00 00 00 ........
$0058-00 00 00 00 00 00 00 00 ........
$0060-00 00 00 00 00 00 00 00 ........
$0068-00 00 00 00 00 00 00 00 ........
$0070-00 00 00 00 00 00 00 00 ........
$0078-00 00 00 00 00 00 00 00 ........
$0080-00 00 00 00 00 00 00 00 ........
$0088-00 00 00 00 00 00 00 00 ........
$0090-00 00 00 00 00 00 00 00 ........
$0098-00 00 00 00 00 00 00 00 ........
$00A0-00 00 00 00 00 00 00 00 ........
$00A8-00 00 00 00 00 00 00 00 ........
$00B0-00 00 00 00 00 00 00 00 ........
$00B8-00 00 00 00 00 00 00 00 ........
```

In a few seconds, up pops the BIN Editor. Like the LORES editor, press ESC to see the command line and use the ‘?’ command to list the valid commands.


```
    BIN-Edit ]L4 - a BIN file editor
B:?

 up/down/left/right - Move Cursor
 +/- - Next/Previous Page
 ENTER - Edit value
 BIN-Edit COMMANDS:

 load         store        x
 y            quit
B:
```

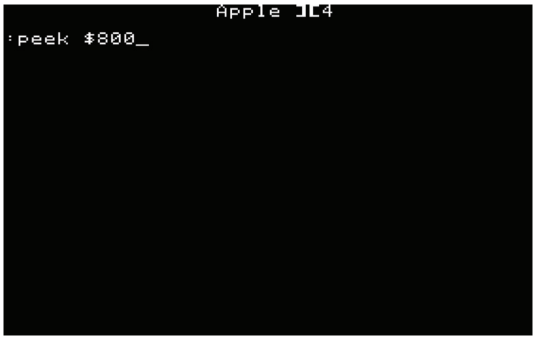
This program has a cursor, seen as the inverted text, and again, UP, DOWN, LEFT, RIGHT move the cursor.

When viewing the file, RETURN lets you edit the value pointed to by the cursor, and then RETURN lets you save that value.

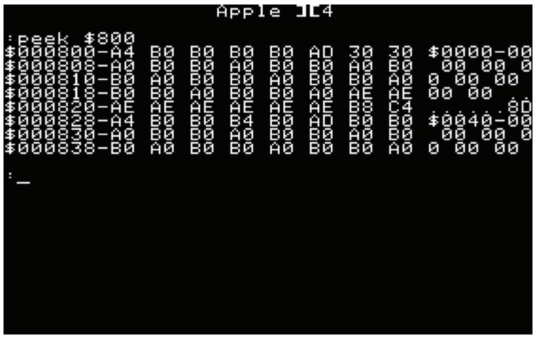
The ‘x’ and ‘y’ commands again let you move the cursor from the command line, and again the ‘store’ and ‘load’ commands will let you save and load files. In this case, BIN files, a.k.a. binary files.

And to keep these command-line programs consistent, when you are done editing your BIN file, use the “quit” command to return to the OS4 command line.

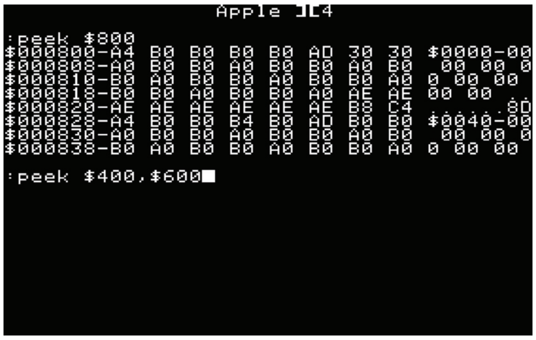
Back at the OS4 prompt, you can view the memory of the Apple II4 using the "peek" command. Peek will display the memory starting at address zero if you do not specify an address. You can specify addresses using either decimal or hexadecimal values. Hexadecimal values are specified by typing a '$' before the value, e.g. $ 8 0 0 for memory location $800 = 2048.



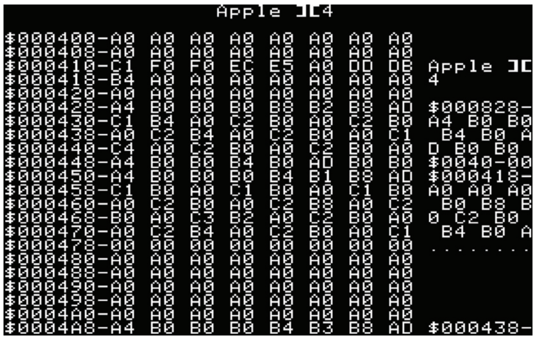
```
                    Apple ][4
:peek $800_
```



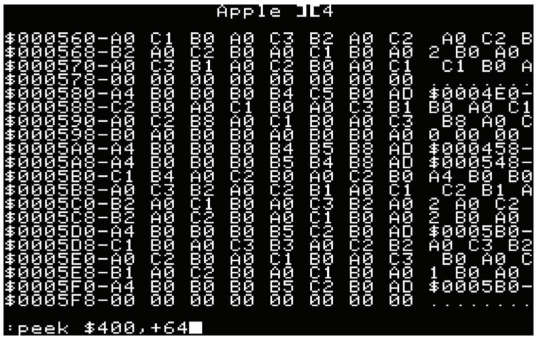
```
                    Apple ][4
:peek $800
$000800-A4 B0 B0 B0 B0 AD 30 30 $0000-00
$000808-A0 B0 B0 A0 B0 B0 A0 B0  00 00 0
$000810-B0 A0 B0 B0 A0 B0 B0 A0 0 00 00
$000818-B0 B0 A0 B0 B0 A0 AE AE 00 00 ..
$000820-AE AE AE AE AE AE B8 C4 ......8D
$000828-A4 B0 B0 B4 B0 AD B0 B0 $0040-00
$000830-A0 B0 B0 A0 B0 B0 A0 B0  00 00 0
$000838-B0 A0 B0 B0 A0 B0 B0 A0 0 00 00

:_
```



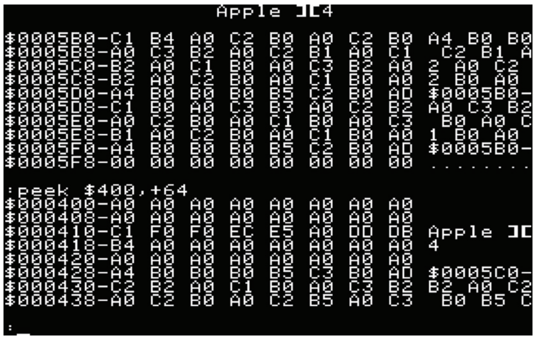
```
                    Apple ][4
:peek $800
$000800-A4 B0 B0 B0 B0 AD 30 30 $0000-00
$000808-A0 B0 B0 A0 B0 B0 A0 B0  00 00 0
$000810-B0 A0 B0 B0 A0 B0 B0 A0 0 00 00
$000818-B0 B0 A0 B0 B0 A0 AE AE 00 00 ..
$000820-AE AE AE AE AE AE B8 C4 ......8D
$000828-A4 B0 B0 B4 B0 AD B0 B0 $0040-00
$000830-A0 B0 B0 A0 B0 B0 A0 B0  00 00 0
$000838-B0 A0 B0 B0 A0 B0 B0 A0 0 00 00

:peek $400,$600█
```



```
                    Apple ][4
$000400-A0 A0 A0 A0 A0 A0 A0 A0
$000408-A0 A0 A0 A0 A0 A0 A0 A0
$000410-C1 F0 F0 EC E5 A0 DD DB Apple ][
$000418-B4 A0 A0 A0 A0 A0 A0 A0 4
$000420-A0 A0 A0 A0 A0 A0 A0 A0
$000428-A4 B0 B0 B0 B8 B2 B8 AD $000828-
$000430-C1 B4 A0 C2 B0 A0 C2 B0 A4 B0 B0
$000438-A0 C2 B4 A0 C2 B0 A0 C1  B4 B0 A
$000440-C4 A0 C2 B0 A0 C2 B0 A0 D B0 B0
$000448-A4 B0 B0 B4 B0 AD B0 B0 $0040-00
$000450-A4 B0 B0 B0 B4 B1 B8 AD $000418-
$000458-C1 B0 A0 C1 B0 A0 C1 B0 A0 A0 A0
$000460-A0 C2 B0 A0 C2 B8 A0 C2  B0 B8 B
$000468-B0 A0 C3 B2 A0 C2 B0 A0 0 C2 B0
$000470-A0 C2 B4 A0 C2 B0 A0 C1  B4 B0 A
$000478-00 00 00 00 00 00 00 00 ........
$000480-A0 A0 A0 A0 A0 A0 A0 A0
$000488-A0 A0 A0 A0 A0 A0 A0 A0
$000490-A0 A0 A0 A0 A0 A0 A0 A0
$000498-A0 A0 A0 A0 A0 A0 A0 A0
$0004A0-A0 A0 A0 A0 A0 A0 A0 A0
$0004A8-A4 B0 B0 B0 B4 B3 B8 AD $000438-
```



```
                    Apple ][4
$000560-A0 C1 B0 A0 C3 B2 A0 C2  A0 C2 B
$000568-B2 A0 C2 B0 A0 C1 B0 A0 2 B0 A0
$000570-A0 C3 B1 A0 C2 B0 A0 C1  C1 B0 A
$000578-00 00 00 00 00 00 00 00 ........
$000580-A4 B0 B0 B0 B4 C5 B0 AD $0004E0-
$000588-C2 B0 A0 C1 B0 A0 C3 B1 B0 A0 C1
$000590-A0 C2 B8 A0 C1 B0 A0 C3  B8 A0 C
$000598-B0 A0 B0 B0 A0 B0 B0 A0 0 00 00
$0005A0-A4 B0 B0 B0 B4 B5 B8 AD $000458-
$0005A8-A4 B0 B0 B0 B5 B4 B8 AD $000548-
$0005B0-C1 B4 A0 C2 B0 A0 C2 B0 A4 B0 B0
$0005B8-A0 C3 B2 A0 C2 B1 A0 C1  C2 B1 A
$0005C0-B2 A0 C1 B0 A0 C3 B2 A0 2 A0 C2
$0005C8-B2 A0 C2 B0 A0 C1 B0 A0 2 B0 A0
$0005D0-A4 B0 B0 B0 B5 C2 B0 AD $0005B0-
$0005D8-C1 B0 A0 C3 B3 A0 C2 B2 A0 C3 B2
$0005E0-A0 C2 B0 A0 C1 B0 A0 C3  B0 A0 C
$0005E8-B1 A0 C2 B0 A0 C1 B0 A0 1 B0 A0
$0005F0-A4 B0 B0 B0 B5 C2 B0 AD $0005B0-
$0005F8-00 00 00 00 00 00 00 00 ........

:peek $400,+64█
```



```
                    Apple ][4
$0005B0-C1 B4 A0 C2 B0 A0 C2 B0 A4 B0 B0
$0005B8-A0 C3 B2 A0 C2 B1 A0 C1  C2 B1 A
$0005C0-B2 A0 C1 B0 A0 C3 B2 A0 2 A0 C2
$0005C8-B2 A0 C2 B0 A0 C1 B0 A0 2 B0 A0
$0005D0-A4 B0 B0 B0 B5 C2 B0 AD $0005B0-
$0005D8-C1 B0 A0 C3 B3 A0 C2 B2 A0 C3 B2
$0005E0-A0 C2 B0 A0 C1 B0 A0 C3  B0 A0 C
$0005E8-B1 A0 C2 B0 A0 C1 B0 A0 1 B0 A0
$0005F0-A4 B0 B0 B0 B5 C2 B0 AD $0005B0-
$0005F8-00 00 00 00 00 00 00 00 ........

:peek $400,+64
$000400-A0 A0 A0 A0 A0 A0 A0 A0
$000408-A0 A0 A0 A0 A0 A0 A0 A0
$000410-C1 F0 F0 EC E5 A0 DD DB Apple ][
$000418-B4 A0 A0 A0 A0 A0 A0 A0 4
$000420-A0 A0 A0 A0 A0 A0 A0 A0
$000428-A4 B0 B0 B0 B5 C3 B0 AD $0005C0-
$000430-C2 B2 A0 C1 B0 A0 C3 B2 B2 A0 C2
$000438-A0 C2 B0 A0 C2 B5 A0 C3  B0 B5 C

:_
```

You can also specify the number of bytes to display, specifying that either as a second address or as the number of bytes to display. In the latter case, prefix the number of bytes with a '+'.

CHAPTER 5

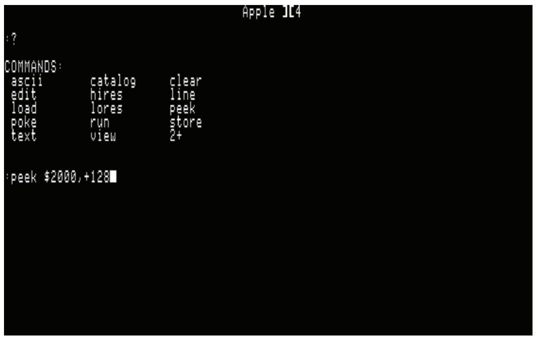
80-COLUMN TEXT

A very common expansion card for the Apple II Plus is the 80-column card. When enabled, this displays an 80x24 character page of text instead of the built-in 40x24 characters.

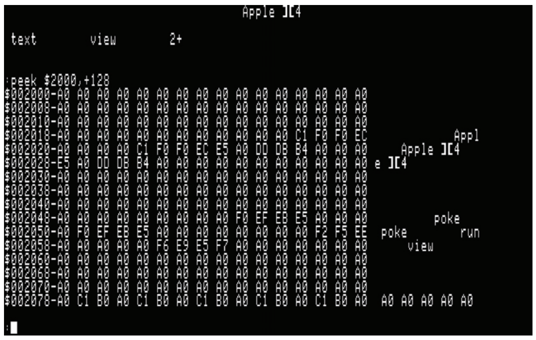
The 80-column card stores the extra 960 bytes of text in a bank of memory outside the main 64K. This design choice makes it challenging for programs that want to support both 40-column and 80-column text.

The Apple II4 implements the same 80-columns behavior, but as a built-in function of the computer, within the standard memory map, reusing the memory normally used by page one of the HIRES graphics mode.

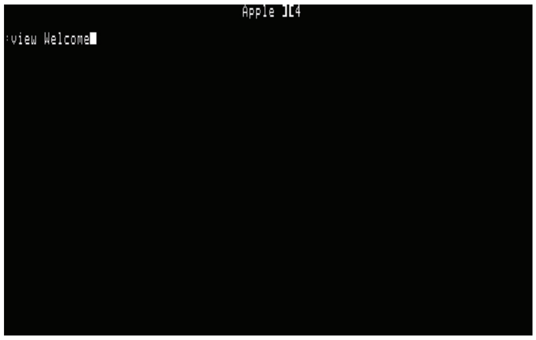
To switch to display 80 column text, use the “text 80” command.



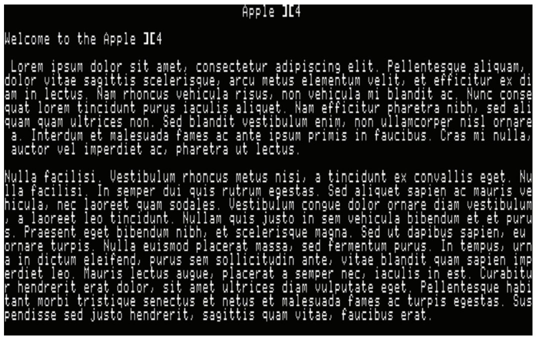
```
                                    Apple ][4
:?

COMMANDS:
 ascii       catalog     clear
 edit        hires       line
 load        lores       peek
 poke        run         store
 text        view        2+


:peek $2000,+128█
```



```
                                    Apple ][4
 text        view        2+


:peek $2000,+128
$002000-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002008-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002010-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002018-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 C1 F0 F0 EC              Appl
$002020-A0 A0 A0 A0 C1 F0 F0 EC E5 A0 DD DB B4 A0 A0 A0       Apple ][4
$002028-E5 A0 DD DB B4 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 e ][4
$002030-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002038-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002040-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002048-A0 A0 A0 A0 A0 A0 A0 A0 A0 F0 EF EB E5 A0 A0 A0         poke
$002050-A0 F0 EF EB E5 A0 A0 A0 A0 A0 A0 A0 A0 F2 F5 EE  poke        run
$002058-A0 A0 A0 A0 A0 F6 E9 E5 F7 A0 A0 A0 A0 A0 A0 A0      view
$002060-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002068-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002070-A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
$002078-A0 C1 B0 A0 C1 B0 A0 C1 B0 A0 C1 B0 A0 C1 B0 A0  A0 A0 A0 A0 A0

:█
```



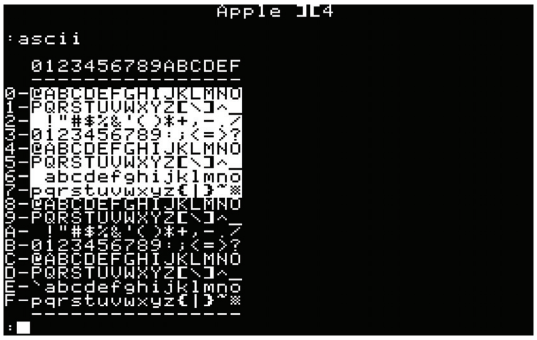
```
                                    Apple ][4
:view Welcome█
```



```
                                    Apple ][4
Welcome to the Apple ][4

 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque aliquam,
dolor vitae sagittis scelerisque, arcu metus elementum velit, et efficitur ex di
am in lectus. Nam rhoncus vehicula risus, non vehicula mi blandit ac. Nunc conse
quat lorem tincidunt purus iaculis aliquet. Nam efficitur pharetra nibh, sed ali
quam quam ultrices non. Sed blandit vestibulum enim, non ullamcorper nisl ornare
 a. Interdum et malesuada fames ac ante ipsum primis in faucibus. Cras mi nulla,
 auctor vel imperdiet ac, pharetra ut lectus.

Nulla facilisi. Vestibulum rhoncus metus nisi, a tincidunt ex convallis eget. Nu
lla facilisi. In semper dui quis rutrum egestas. Sed aliquet sapien ac mauris ve
hicula, nec laoreet quam sodales. Vestibulum congue dolor ornare diam vestibulum
, a laoreet leo tincidunt. Nullam quis justo in sem vehicula bibendum et et puru
s. Praesent eget bibendum nibh, et scelerisque magna. Sed ut dapibus sapien, eu
ornare turpis. Nulla euismod placerat massa, sed fermentum purus. In tempus, urn
a in dictum eleifend, purus sem sollicitudin ante, vitae blandit quam sapien imp
erdiet leo. Mauris lectus augue, placerat a semper nec, iaculis in est. Curabitu
r hendrerit erat dolor, sit amet ultrices diam vulputate eget. Pellentesque habi
tant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sus
pendisse sed justo hendrerit, sagittis quam vitae, faucibus erat.
```

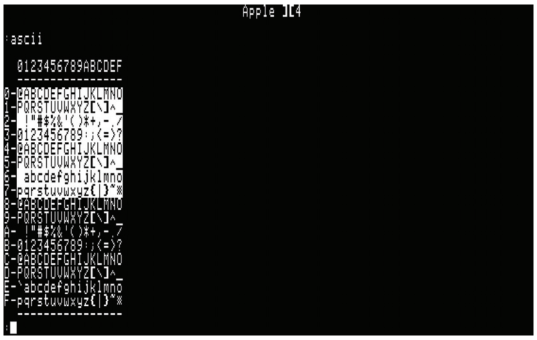
All the commands act the same as in 40 column mode, except that more text is visible on the screen. Some commands, e.g. ‘peek’ will change their format, in this case displaying 16 bytes per row instead of 8.

To switch back to 40 columns, the command is “text 40”.



```
                              Apple ][4
:ascii

   0123456789ABCDEF
   ----------------
0-@ABCDEFGHIJKLMNO
1-PQRSTUVWXYZ[\]^_
2- !"#$%&'()*+,-./
3-0123456789:;<=>?
4-@ABCDEFGHIJKLMNO
5-PQRSTUVWXYZ[\]^_
6- abcdefghijklmno
7-pqrstuvwxyz{|}~※
8-@ABCDEFGHIJKLMNO
9-PQRSTUVWXYZ[\]^_
A- !"#$%&'()*+,-./
B-0123456789:;<=>?
C-@ABCDEFGHIJKLMNO
D-PQRSTUVWXYZ[\]^_
E-`abcdefghijklmno
F-pqrstuvwxyz{|}~※
   ----------------
:
```

40 column text



```
                                        Apple ][4
:ascii

   0123456789ABCDEF
   ----------------
0-@ABCDEFGHIJKLMNO
1-PQRSTUVWXYZ[\]^_
2- !"#$%&'()*+,-./
3-0123456789:;<=>?
4-@ABCDEFGHIJKLMNO
5-PQRSTUVWXYZ[\]^_
6- abcdefghijklmno
7-pqrstuvwxyz{|}~※
8-@ABCDEFGHIJKLMNO
9-PQRSTUVWXYZ[\]^_
A- !"#$%&'()*+,-./
B-0123456789:;<=>?
C-@ABCDEFGHIJKLMNO
D-PQRSTUVWXYZ[\]^_
E-`abcdefghijklmno
F-pqrstuvwxyz{|}~※
   ----------------
:
```

80 column text

CHAPTER 6

THE 24-BIT MEMORY MAP

With up to 16,384K of memory the Apple II4 memory map is a lot less of a squash and a squeeze as compared with the Apple II and II Plus.

```
+----------------------------------------------+
$0000-$00EF - Zero page
$00F0-$00FF - OS4 ‘‘registers’’
$0100-$01FF - 6502 JSR/RTS stack
$0200-$03FF - unused
$0400-$07FF - TEXT page 1 (and LORES page 1)
$0800-$0BFF - LORES page 2 (and TEXT page 2)
$0C00-$03FF - unused
$2000-$3FFF - TEXT80 and HIRES page 1
$4000-$5FFF - HIRES page 2
$6000-$BFFF - unused
$C000-$CFFF - Soft switches and Expansion cards
$D000-$DFFF - Command line and subroutine parameters
$E000-$FCFF - unused
$FE00-$FEFF - Command line input buffers
$FF00-$FFFF - Global variables
+----------------------------------------------+
$10000-RAM    - RUN programs and VIEW/EDIT files
$FF000-$FFFFF - OS4 ROM
+----------------------------------------------+
```

On both the 6502 and 652402, the first 256 bytes of memory are special, with opcodes that can read, write, and manipulate these values using 1-byte addresses instead of 2-bytes or 3-bytes. Bill Mensch, one of the designers of the 6502 called this feature ‘‘addressable registers’’, as in the microprocessor didn’t have just the three A, X, and Y registers as taught in the assembly code manuals, but another 256 registers using these special ‘‘zero page’’ addressing modes.

The Apple II4 and OS4 follow Mr. Mensch’s vision and leaves 240 of the 256 bytes of this precious ‘‘zero page’’ memory space for programmers to use as registers or faster-to-access variables.

The TEXT, LORES, and HIRES memory pages are located at the same addresses as on the Apple II, with page one at $400, $400, and $2000 respectively. The ‘‘view’’ and ‘‘edit’’ programs for LORES graphics use LORES page 2 instead of page 1 to allow for toggling between TEXT and LORES graphics. LORES page 2 starts at address $800. TEXT page 2 is not used by OS4.

The TEXT80 page is stored in memory $2000-$27F0, overlapping page one of the HIRES graphics. In this case, if you ‘‘view’’ a HIRES graphics file it will be seen as random characters on the 80 column text screen, and if you enter commands in ‘‘text 80’’ mode, you will see random colors appear on the ‘‘hires’’ page.

Addresses $C000-$CFFF are unchanged from the Apple II, except the soft-switches to enable and disable TEXT80 mode are $C068 and $C060. On the Apple II and II Plus these are used to read and write from the cassette tape. With Disk II4 as standard, the Apple II4 does not have a built-in ability to read or write to a cassette tape.

Addresses $D000-$DFFF are used to pass parameters into the command line, and to/from the built-in library of subroutines stored in ROM.

$E000 is an unused block available to programs.

Addresses $FE00-$FEFF hold the text from the command line along with copies of the previous two commands entered on the command line. On the Apple II, this input buffer is located down at address $200-$2FF. It had to be near the start of memory on the original Apple II as it had to fit within the 4K of memory of the earliest Apple computers.

Similarly, previous Apple II computers put their global variables down in addresses $300-$3FF. The Apple II4 moves those ‘‘out of the way’’ up in addresses $FF00-$FFFF, where the previous Apple IIs stored their ROM.

All of the above memory addresses are within the first 64K. $10000 is the first address newly available given the 24-bit addresses of the 652402, and the memory map from there onward is simple. OS4 loads programs from disk to location $10000 and the program can use any location afterward as one big memory space, a.k.a. a memory ‘‘heap’’. The size of the loaded program can be found in RUNLENGTH ($FF16), the first available address rounded up to the nearest $100 bytes after RUNLENGTH can be found at HEAPSTART ($FF13), and the size of the installed RAM can be found in RAMTOP ($FF16). Each of these values is a 24-bit value (3-bytes stored in little-endian, e.g. $abcdef stored in three consecutive bytes as $ef $cd $ab).

Finally, the OS4 ROM begins at address $FF0000 and ends at the last addressable memory location, $FFFFFF.

CHAPTER 6

FOCUS ON FILES

From the list of commands, it may be obvious that there is a shift in focus between OS4 and previous Apple computer command lines. The Apple II4 is focused on files, rather than memory.

The INSIDE Story on files

Steve Jobs made another fateful trip in 1978, this one to AT&T's famous Bell Labs, the research lab where the transistor was first invented. On that trip, the researchers at Bell Labs showed Steve their latest and greatest software inventions, centered around the UNIX operating system.

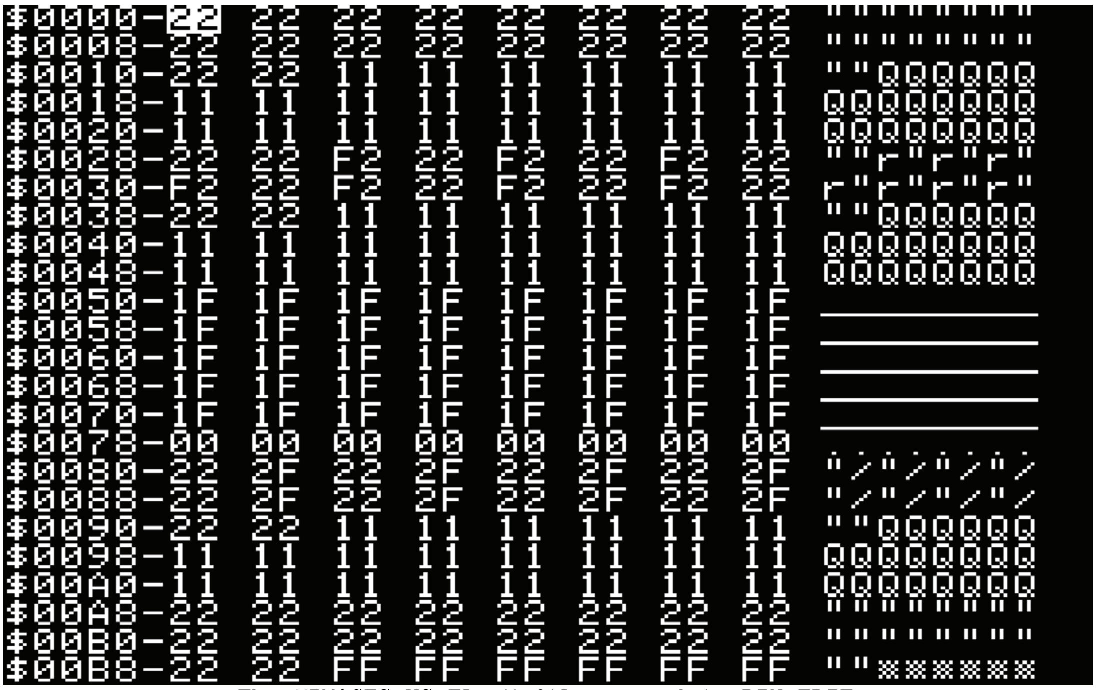
UNIX was designed for minicomputers, but by 1978 the Apple II was nearly as powerful as the Digital Equipment PDP/8, the computer that UNIX was original built on, back in the mid-1970s.

Not all of UNIX is applicable to microcomputers, but one design feature is quite applicable now that the Disk][4 is a standard feature on every Apple computer. UNIX's focus on files.

This design feature explains why “view” and “edit” do what they do in OS4. Earlier it was shown how “edit” looks for a matching file in the OS4 sub-directory called XYZ EDIT, where XYZ matches the file format. The “view” has built-in viewers for TXT, LO_, HI_, and BIN files. For all others it follows the same process as “edit”, looking for XYZ VIEW in the OS4 directory.

Programmers can create their own file formats, their own VIEW-only programs, and their own EDIT programs.

Alternatively, the “run” command can be used to load and run a program, and a file name can be entered on the command line to optionally open and act on a specific file. For example, “run BIN EDIT,US Flag” will load and run the BIN EDIT program, which will in turn open and edit the US Flag file. Not as a LORES graphics image, but as a binary (BIN) file.



```
$0000-22 22 22 22 22 22 22 22 """"""""
$0008-22 22 22 22 22 22 22 22 """"""""
$0010-22 22 11 11 11 11 11 11 ""QQQQQQ
$0018-11 11 11 11 11 11 11 11 QQQQQQQQ
$0020-11 11 11 11 11 11 11 11 QQQQQQQQ
$0028-22 22 F2 22 F2 22 F2 22 ""r"r"r"
$0030-F2 22 F2 22 F2 22 F2 22 r"r"r"r"
$0038-22 22 11 11 11 11 11 11 ""QQQQQQ
$0040-11 11 11 11 11 11 11 11 QQQQQQQQ
$0048-11 11 11 11 11 11 11 11 QQQQQQQQ
$0050-1F 1F 1F 1F 1F 1F 1F 1F ________
$0058-1F 1F 1F 1F 1F 1F 1F 1F ________
$0060-1F 1F 1F 1F 1F 1F 1F 1F ________
$0068-1F 1F 1F 1F 1F 1F 1F 1F ________
$0070-1F 1F 1F 1F 1F 1F 1F 1F ________
$0078-00 00 00 00 00 00 00 00
$0080-22 2F 22 2F 22 2F 22 2F "/"/"/"/
$0088-22 2F 22 2F 22 2F 22 2F "/"/"/"/
$0090-22 22 11 11 11 11 11 11 ""QQQQQQ
$0098-11 11 11 11 11 11 11 11 QQQQQQQQ
$00A0-11 11 11 11 11 11 11 11 QQQQQQQQ
$00A8-22 22 22 22 22 22 22 22 """"""""
$00B0-22 22 22 22 22 22 22 22 """"""""
$00B8-22 22 FF FF FF FF FF FF ""▒▒▒▒▒▒
```

The ‘‘IMAGES:US Flag’’ file opened in BIN EDIT

CHAPTER 7

BASIC

Applesoft BASIC has also been updated on the Apple II4, but not dramatically. The new big features are: support for lowercase, support for hexadecimal numbers, and support for programs larger than 64K in size.

Plus, following the file-centric model, the “run” command is smart, looking at the file type to determine if a file is a BASIC program (type BAS), and if so, loading and running it inside the new BASIC interpreter.

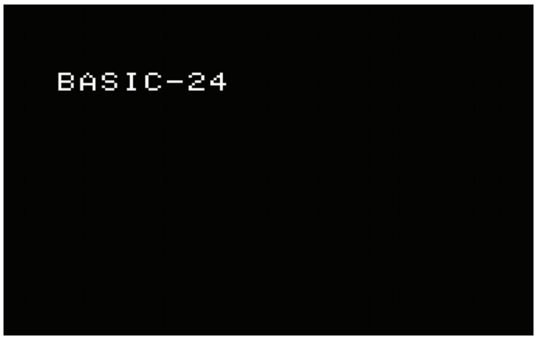
That interpreter is built into the OS4 ROM.

However, unlike the Apple II, where the ‘>’ command line was also the Integer BASIC interpreter, or the Apple II Plus, where the ‘]’ command line was also the Applesoft BASIC interpreter, on the Apple II4, you have to run the ‘basic’ command to start the BASIC interpreter.

We made that change for two reasons. One, because with the file-centric operating system, the expectation is that the BASIC program will be edited in a text editor and saved as a TXT file, rather than entered line by line inside the BASIC interpreter. Two, because there are other languages available for the Apple II, with more on their way.

And most importantly, three, because more and more owners of Apple computers are not programmers or computer hobbyists like the initial customers. More and more computer owners are office workers or other professionals, who expect the computer to first and foremost run programs that are created by professional programmers. With that, programming languages are now just one more program that can be run, rather than a central activity of an Apple computer.

Beyond running and editing BASIC programs, there are a few other small changes. First, PEEK, POKE, and all other values can be specified in hexadecimal, using #$ as a prefix, e.g. the value 32769 can be entered as #$8001. Second, BASIC programs can be megabytes in size, limited only by the amount of RAM installed on the Apple][4. BASIC programs are loaded into memory location $10000 and use the memory between HEAPSTART and RAMTOP to store the program and its variables.


```
BASIC-24
```

CHAPTER 8

ASSEMBLER

The Apple II ROM included the “mini assembler”. The Apple II4 ROM includes a full, multi-pass symbolic assembler.

Most professional software programs are written in assembly code, not BASIC. With up to 1,024K of memory installed on an Apple II4, complex programs can now be edited, compiled, and debugged without needing to “cross compile” the assembly code on a minicomputer or mainframe computer.

To write a program in assembly, use the text editor to write your program and save it as a TXT file. Then type a s s e m b l e followed by a space and a filename for the assembled program, followed by a comma and the filename of the text file to assemble (followed by more commas and more filenames if you have split your program into multiple files) all followed by RETURN. The assembler will create a file of type RUN with your assembled program if there are no errors, or it will display those errors on the screen for you to fix.

Once assembled, you can “run FILENAME” to run and test your program.

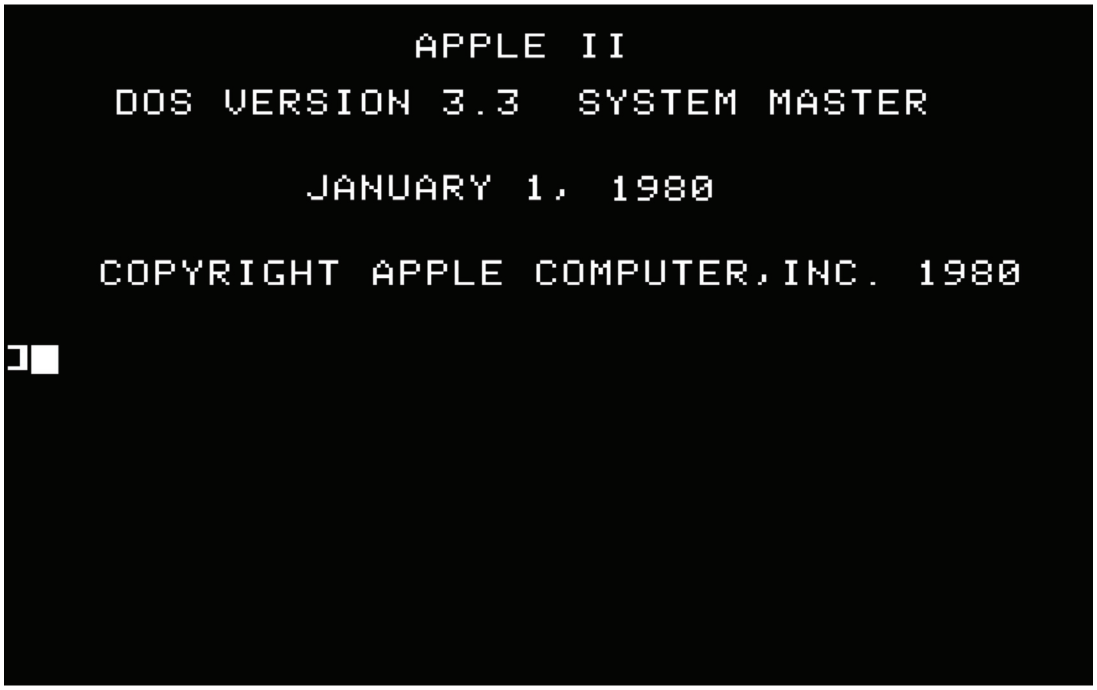
See the “Apple][4 Assembler” manual for details on the syntax of this assembler.

```
Assembly code
```

CHAPTER 9

APPLE II PLUS

To support ***all*** of the legacy programs from the Apple II and II Plus, the Apple II4 includes the “2+” command. Type 2 + RETURN and the memory map switches back to the that of the Apple II Plus, the ROMs from the II Plus swap into memory locations $D000-$FFFF, the keyboard switches to all UPPERCASE, the floppy disks switch to be single sided, the computer boots from the floppy disk as if it was an Apple II Plus, and the microprocessor speed drops down to 1MHz.



```
                 APPLE II
    DOS VERSION 3.3  SYSTEM MASTER

           JANUARY 1, 1980

   COPYRIGHT APPLE COMPUTER,INC. 1980

]█
```

In short, every Apple II4 is also a 64K Apple II Plus.

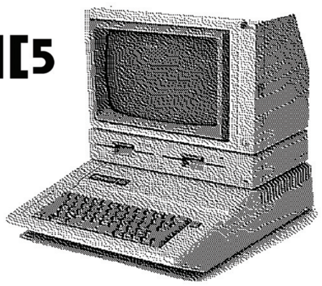
apple][5

AFTER THE II4

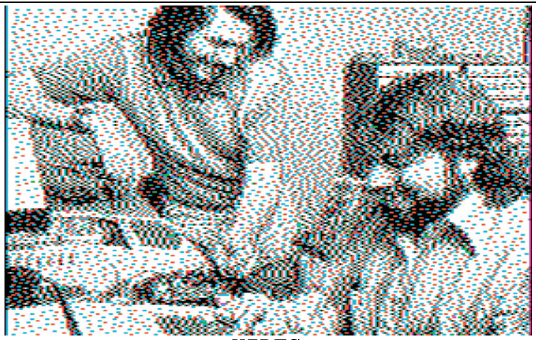
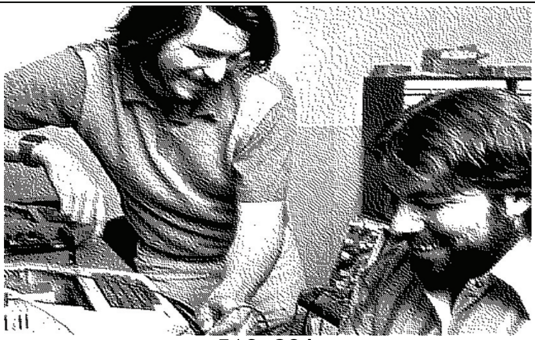
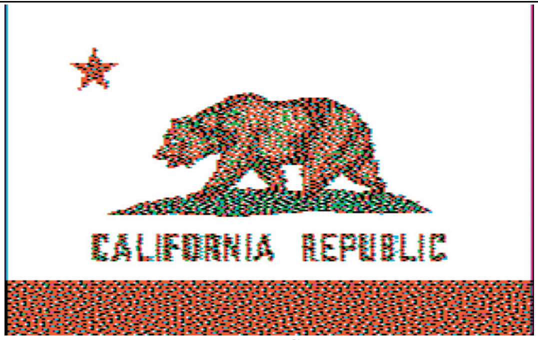
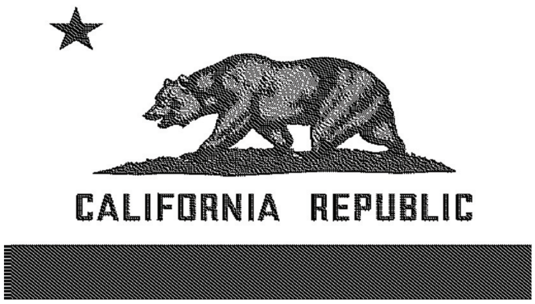
THE APPLE II5

Back in 1980 we promised a new Apple computer every year, and here in 1981 the new model is the Apple II5. In keeping with the double meanings of the names, the II5 is not only the next number is the sequence, but the ‘5’ represents the new super high resolution 512x384 pixel graphics mode.

apple][5

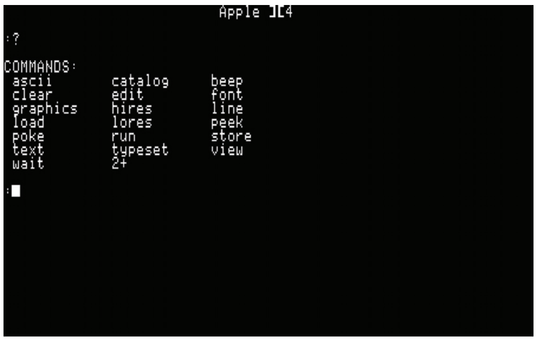


Here are a few images: HIRES (280x192) and GRAPHICS (512x384)

 HIRES	 512x384
 HIRES	 512x384

The quality of the new GRAPHICS mode is obvious, but you may notice that those images are black and white. Yes, that is the tradeoff. A single image with 512x384 black and white pixels requires 24K of memory. Just four colors would double that to 48K, and the 16 colors of LORES memory would require 96K just to store one graphics image. Future Apple][models will add color, as memory sizes increase.

TEXT64



```
                                Apple ][4

:?

COMMANDS:
 ascii          catalog        beep
 clear          edit           font
 graphics       hires          line
 load           lores          peek
 poke           run            store
 text           typeset        view
 wait           2+

:█
```

Along with the new graphics mode is a new TEXT mode as well. When you power up the Apple II5, you may notice that the text is smaller than the 40-column TEXT of the II4, but not as squished as the 'text 80' mode.

This new TEXT is 64-columns wide, and the same 24-rows as before. The new mode is far easier to read than 'text 80'. 64 columns was chosen as 512 divided by 8 is 64, and each character is 8 bits wide.

The 'text 64' memory use the same section of RAM as the 'text 80' memory, both begin at address $2000 (overlapping HIRES page 1), with TEXT64 ending at $25FF.

Beyond that, all the built-in commands display just as well with 64 columns as they do with 40 or 80 columns.

3MHz

The only other change from the II4 to the II5 is another boost in speed, with the II5 running at 3MHz, 50% faster than the II4, three times faster than the II and II Plus. And, as predicted, the amount of RAM in the average II5 is more than the II4, with 256K now affordable and 128K standard.

AFTER THE II4

THE APPLE II6

Moving on to 1982, the Apple II6 is another year with a major upgrade. The big change this year is the 652424 microprocessor, which leapfrogs Apple Computer beyond the age of 8-bit computing and beyond 16-bits too, jumping ahead to 24-bits.

Once again, this new microprocessor is 100% backward compatible with both the 652402 and the original 6502, ensuring the tens of thousands of legacy programs can continue to run.

CMOS

Externally, the 652424 acts like a bigger, better 652402, but internally it is a totally new design, based on the new Complementary Metal Oxide Semiconductor technology (a.k.a. CMOS).

This lowers the power consumption of the 652424 significantly vs. the 652402, and with that, the clock speed has been boosted as high as 8MHz, more than twice as fast as the Apple II5, four times faster than the II4, and eight times faster than the II and II Plus.

But that speed does cost more, so the Apple II6 is the first Apple computer where the customer can pick the speed of the microprocessor, with choices of 4MHz, 6MHz, and 8MHz.

THE 652424 PREFIX CODES

What makes the 652424 a 24-bit computer is that it can add, subtract, load, and store 24-bits in a single operation. It can do the same with 16-bits and 8-bits, making it optimal for processing both text characters and numbers, and a wide variety of small, medium, and large values.

This flexibility is implemented using a method similar to the A24 opcode from the 652402. The 652424 adds new prefix codes: R16 and R24 which change the size of the CPU registers for the next operations.

For example, R16 LDA #$1234 will load the 16-bit value $1234 into the A register. Similarly, R24 LDA #1048576 will load the 24-bit value 1,048,576 into the A register. The prefix code defining the width of the A register for that LDA operation, filling in any unused bits with zeros.

The X and Y registers can also each store an 8-bit, 16-bit, or 24-bit value. R16 LDX $2000 will load a 16-bit value from address $2000 into the X register. R16 INX will then increment the 16-bit value in the X register. R16 TXA followed by R16 TAY will copy that 16-bit value from the X register to the A register and then to the Y register.

The R24 prefix code acts similarly, except that the values loaded are 24-bits (3 bytes). In all these cases, the value is stored in little endian, e.g. $abcdef is stored in consecutive memory addresses as $ef $cd $ab.

It is also possible to mix the 24-bit addresses with 16-bit and 24-bit registers. For that, rather than requiring two prefix codes, the W16 and W24 prefix codes have been added. These are just like R16 and R24 except that the address in the opcode is 24-bits wide instead of 16-bits.

For example, W24 LDA $123400 loads a 24-bit value from the 24-bit memory address $123400. W24 ADC $10400 adds the 24-bit value stored in address $10400 to the A register.

652424 ASSEMBLY

This may sound complicated, but the assembler has been updated to take care of the prefix codes for assembly language programmers. Rather than specifying prefix codes, the assembly mnemonics are augmented with suffixes.

LDA.a24, STA.a24, ADC.a24, JMP.a24 specifies that the opcode address is 24-bits. LDA.b or LDA.r8 loads an 8-bit value, LDA.w or LDA.r16 loads a 16-bit value, and LDA.t or LDA.r16 load a 24-bit value.

The suffixes can have suffixes, e.g., LDA.a24.w loads a 16-bit value from a 24-bit memory address and STA.a24.t stores a 24-bit value into a 24-bit memory address.

That too may at first sound a little complicated, but it becomes second nature after a little practice as programmers plan which memory locations store 8-bit, 16-bit, and 24-bit values, and at any given time plans out the width of the values in the A, X, and Y registers.

A BIGGER STACK

The S register gets an upgrade too. The 652424 is no longer limited to just 256 bytes of stack space. Upon power up or reset, the S register is set to 8-bits and memory location $1FF to ensure backward compatibility with the 6502.

But the ROM in the Apple II6 includes the sequence R16 SWS, using the new SWS opcode to change the width of the S register to 16-bits, and then initializes the stack address to $FEFF. The width of the S register is specified by the prefix code (or lack thereof) before the SWS opcode. R24 SWS sets the S register to be 24-bits wide and SWS alone resets the width to 8-bits.

SWS does not set the value of the S register. TXS is used for that, with the sequence LDX.w #$FEFF (which is assembled into R16 LDX #$FEFF) followed by TXS.w (assembled into R16 TXS) to move the value into the S register.

No matter the width of the S register, PHA.w, PHA.t, PLA.w, and PLA.t will push and pull 16-bit or 24-bit values from the stack.

OTHER NEW 652424 OPCODES

In addition to wider registers, the 652424 adds a few new opcodes and one new addressing mode.

PHX, PLX, PHY, and PLY push and pull values from the X and Y registers to the stack.

STZ stores the value #0, with address modes for zero-page and other addresses, with and without adding the X index register.

INC and DEC increment and decrement the A register.

SL8 and SR8 shift the A register left and right 8-bits at a time. This is helpful when processing 16-bit or 24-bit values byte by byte.

XSL and YSL shift the X or Y register left by one bit, filling in the least significant bit with a zero and ignoring the value of the most significant bit. This is useful for indexed address mode when tables include 16-bit values, as the index needs to be multiplied by two as each entry is two bytes long.

ADX, ADY, and AXY add the value of X to A, Y to A, or both X and Y to A. This is especially useful with 16-bit and 24-bit values, as the X and Y registers can act as address registers rather than just index registers. Complex structured data can be easily managed with just a few opcodes using a little addition, as well as allowing X and Y to store temporary values that would otherwise have to be stored in memory when the A register is otherwise needed.

Three existing opcodes get two brand new addressing modes: LDA X, LDA XY, STA X, STA XY, and CMP X, CMP XY. In these modes, the address for the load or store or compare is either stored in the X register alone, or is the sum of the X and Y registers.

The XY mode is similar to the 6502's (zp),Y indirect indexed address mode, without having to use the zero page. The X addressing mode is similar to a (zp) addressing mode, again without using the zero page, and without adding the value of the Y register. Given a 24-bit value in the X register can hold any address, and given INX.t can increment the X register, this new address mode along with the INX opcode is often sufficient for iterating through an array of values, without needing Y as an index register.

JMP and JSR get small updates, with indirect JMP (addr) and JSR (addr) address modes. These are used to simplify the assembly code for jumping into the ROM subroutines.

Finally, BRA is added, branching <u>a</u>lways, no matter the state of the 652424 status flags. This makes it far simpler to write relocatable code. So does the A24 extension to all of the BRx opcodes, increasing the distance the branches can branch from 127 bytes (plus or minus) to 32,767 (plus or minus).

24-BIT BASIC

Apple II6 expands BASIC to support 24-bit values too. Numeric variables can store values between -8,388,608 and 8,388,607, up from the range -32,768 to 32,767.

AFTER THE II4

THE APPLE II7

By 1983, the Apple II7 introduces a massive change to Apple Computer, this time in software rather than hardware.

The INSIDE Story of the walk in the PARC

In 1980, Steve Jobs and a few Apple software and hardware engineers were invited to visit Xerox's Palo Alto Research Center (a.k.a. Xerox PARC), located a few towns over from Apple's headquarters in California. Given Xerox's focus on copiers, not much was expected from this trip.

To say the visit exceeded expectations is an understatement.

This research lab had been tasked with building the computer systems from ten years into the future. What Steve and team saw that day totally blew their minds, as Xerox had succeeded in that mission.

The downside was that much of what Xerox had built couldn't be commercialized, as a computer half as powerful as their lab machines would cost $10,000 or more. But that didn't mean there were not a few concepts that could be commercialized, and the first of those appeared in the II5 in 1981, the 512x384 graphics mode, as a core piece of the future of computing was centered on high resolution graphics.

What Xerox did with those graphics took a few more years for Apple to replicate. The graphical user interface.

Apple II7 was fully backward compatible with the II4, II5, and II6, but OS7 was vastly different. Upon power up, OS7 displayed the "Finder", a graphical user interface that used icons and windows rather than a command line. Rather than a flashing prompt, there is a pointer on the screen that moves when the "mouse" is moved. Files and open and programs are run by pointing to an icon representing that file or program and clicking the button on the mouse.

To make the most of this new computing paradigm, the II7 included the 7/7 suite of programs: Write, Draw, Calc, Graph, Calendar, Contacts, and List.

Write is a WYSIWYG text editor. What You See Is What You Get, displaying the text using multiple typefaces, with a ruler to set margins and tabs, embedding graphics alongside the text. It allows a single person to replace a team of typesetters and makes "desktop publishing" possible.

Draw is the tool for creating and editing those graphics. It too is WYSIWYG, with tools like pens and paint brushes, all drive by the mouse.

Calc is a spreadsheet, which like Write can display labels and values using multiple sizes of typefaces, make spreadsheets not just functional, but also pretty.

Graph creates pie charts, line and bar graphs, and scatter plots from the spreadsheet data. These graphs can be embedded within Write embedded into spreadsheets, or imported into Draw.

Calendar is an electronic day planner, storing appointments and meetings.

Contacts is an electronic Rolodex.

List is an electronic "to do" list.

Calendar, Contacts, and List all work together, allowing calendar events to reference contacts and to embed TODO items. Contacts can include embedded TODO items and will show upcoming events for that contact.

All seven of these programs support the "clipboard", allowing information to be copied from application to application.

And sharing is made simpler as the Apple II7 includes two hardware upgrades too:

The first is the move to the 6524T4, fully backward compatible with the 652424, but with four microprocessor "threads". This is like having four separate microprocessors, one to run OS7 and three more, each able to run another program simultaneously. Or in other words, the Apple II7 can "multi task", running up to three programs at once, seamlessly.

The other change is the Integrated Atkinson Chip, a.k.a. the "Bitblt" processor or "graphics accelerator". This chip takes the burden of drawing the text, lines, circles, and rectangles on the graphics screen, speeding up the computer tremendously.

With these changes, the Apple computers are years ahead of the competition, with the easiest to use and most powerful hardware and software in the industry.

MEANWHILE…

THE APPLE /// (AND MORE)

It is 1980 and Apple Computer has two new models for sale, the Apple II4 for hobbyists and home use, and the Apple /// for business.

The big differences are that the /// has a new design, with an integrated double-sided floppy disk drive, four expansion slots instead of eight, has no backward compatibility with the II or II Plus.

The /// is designed to work with a monochrome monitor, which makes 80-column text crisp and clear, but which cannot display color. The /// thus does not support any of the legacy 40-column text or LORES or HIRES graphics from the II, with 80-column text as default.

The Apple /// runs OS///, which uses a similar style of command line as the II4, the same file centric design, the same smart commands like ‘‘view’’ and ‘‘edit’’, but which adds the ability to store sub-directories within sub-directories.

The Apple /// includes a built-in date-time chip. OS/// uses this chip to note when a file is modified.

The Apple /// uses the same 652402 as the II4, but runs the microprocessor at 2.5MHz, 25% faster than the II4 and two and a half times faster than the II and II Plus.

As a serious business computer, the /// comes standard with 128K of RAM, upgradable to up to 1,024K (a.k.a. 1 megabyte), and the built-in Disk/// controller supports up to three external floppy or hard disk drives in addition to the one built-in floppy drive.

The same annual upgrade schedule applies to the /// series of computers as to the II series.

In 1981, the ///a arrives, with the big upgrade the addition of the 512x384 high resolution graphics, identical in functionality to the II5. The 652402 that year is sped up to 4MHz, and the standard RAM grows to 192K.

In 1982, the ///b arrives, with a color monitor and 512x384 graphics that supports up to 16 colors. A second built-in floppy disk drive is included, with an option of swapping that out for a 5 megabyte hard disk drive.

In 1983, the ///c ships standard with 3.5'' floppy drive, internal space for a 10 megabyte hard disk, a mouse, a graphical user interface, and includes the full 7/7 suite of business applications.

...

By the mid 1980s, both the II and /// series supports 1024x768 graphics with 16 million colors, a ''graphics accelerator'' chip, and a 6548T16 with 48-bit addresses, virtual memory, 8/16/24/32-bit registers, and multi-tasking using 16 CPU-resident threads.

The internal rivalry between the II team and /// team helped push both products to excellence, keeping Apple far ahead of the competition. That competition grew when the ''Macintosh'' team adding a third product line, one that took a vastly different path, focusing on computers with no screens, no keyboards, and no mouse, but instead which sat in the closet and served data over the local area network to the IIs and ///s that filled the offices.

The next internal competitor was the ''Dynabook'' team, headed by former Xerox Parc researcher Alan Kay. The Dynabooks are battery-powered portable computers, with wireless connections to the Macintosh network, with touchscreens, and variations of the graphical user interface that made the computer feel more like pen and paper.

And finally, video games. One of Woz's requirements for the first-ever Apple computer was that it run the game Breakout, a game that Woz worked on prior to Apple when he worked at Atari.

One day Woz and Atkinson were playing around with other uses of the Bitblt chip, whipped up a circuit to drive 512x384 color graphics on a television by reusing a small amount of RAM 24 times each time a screen was drawn, and the result was the ''Pippin'' gaming machine.

The Pippin has incredible high resolution, 16 million colors, and runs sixteen times faster than an Atari 2600, but costs only $299 thanks to the reuse of all the Apple][components. With that, Apple took over the home video game market.

The commonality of all these products was the integration of cutting-edge hardware along with software that borrowed the best ideas from across the whole industry, implementing those ideas into easy to use, practical solutions that non-experts could understand.

THE STORY

DOWN THE RABBIT HOLE

Stepping back to the 2020s… where did this alternative history come from?

My first computer was an Apple II Plus, acquired over 40 years ago as a birthday present from my parents at age 13. I have fond memories of that computer, but back then, in the early 1980s, living in the suburbs of New York City, I had limited knowledge to how the computer actually worked, and no access to the books that were written explaining the details.

I hadn't given the Apple II much thought until early in 2022 when I discovered copies of the 1970s, 1980s, and 1990s BYTE magazine on archive.org. My first software company, Nimble, founded in 1992, was inspired by the February 1991 issue of BYTE. That issue had a series of articles talking about what comes next after personal computers and laptops. They predicted pen-based computers. They were right, but that transition took place starting in 2007 with Apple's iPhone and 2010 with Apple's iPad.

After re-reading that issue, I searched for the 1977 issue where Woz explained the features of the then brand-new Apple II.

What caught my eye between that span of 14 years was the growth in RAM in the computers being advertised between the articles. Looking back at these old issues, the ads were just as interesting as the articles, as the ads showcased the state of the art, and quite often, the big visions that turned out to be totally wrong.

Back in 1977 there were S100 bus expansion cards with 16K, 32K and 64K of RAM. They were expensive, but they were for sale. My Apple II Plus from a few years later had 48K installed on the motherboard, and I quickly picked up a Language Card at a computer trade show to add another 16K. All that made me wonder why no one back in the late 1970s had pushed the 8-bit chip makers for more address space.

That question led me down the rabbit hole of the history of the 8080, Z80, 6800, and 6502. Diving into 6502 instruction set and the Visual 6502 dissection of the chip design, it seemed quite reasonable to squeeze in 8 additional bits of address, expanding the addresses from 16-bits to 24-bits, a few extra states in the state machine to manage the extra byte, and extend the 6502 design to what I call a 652402.

To prove the simplicity of the changes, I found a Verilog implementation of the 6502, refreshed my knowledge of Verilog (which was just one undergraduate class 30 years prior and one graduate class 20 years prior), and over one weekend had a working 652402. The changes were as minimal, as expected.

With that design in hand, I found Bill Mensch's email address and sent him a message entitled, "A question only Bill Mensch can answer". Within 24 hours, he answered. In summary, MOS saw the bigger market of microcontrollers vs. microprocessors, and in the microcontroller market, 64K was sufficient. That, and he said none of the computer companies asked for 24-bit addresses.

That sounded reasonable from MOS' perspective, but later I found another answer, that most of the team that designed the 6502 left MOS by 1978, and thus even if they wanted to upgrade their design, they no longer had the talent in house to do that. Mensch had left too, founding Western Design Center in 1978.

I could have stopped with those answers, but this didn't explain why neither Intel nor Zilog expanded the 8080, 8086, or Z80 with larger address spaces either. The only reasonable explanation was that that in foresight they all saw 16-bit CPUs as the next step in microcomputers, despite the 16-bit CPUs also having just 16-bit addresses. Except for Motorola, which leapfrogged the whole industry in 1979 with the 32-bit 68000 CPU.

Eventually I found an interview with the 68000 team, who explained that they hadn't enjoyed losing every important socket in the 8-bit computer era, purposefully bringing a design to management that would best Intel, Zilog, National, and every other contender for the expected, upcoming, 16-bit era. That strategy worked, with the 68000 winning every socket except the IBM PC (which apparently they would have won too if not for a few thousand test chips that IBM requested, which Motorola couldn't provide in time).

The 68000 was familiar to me, as I coerced my parents into buying me a 128K Macintosh in 1986, which was quickly upgraded to 512K. When I started college, the Sun workstations and HP workstations at Carnegie Mellon were running on the 68020, and my summer job for most of college was an Apple-funded project which booted Mach (Carnegie Mellon's UNIX) on Macintosh IIs, running MacOS 7 as a user-level application on top of Mach. A decade later Apple's own OS X would repeat that design pattern, but on the PowerPC.

At this point in the rabbit hole, I had put the 24-bit address space question aside, as it seemed the answer was simple. Apple knew about the 68000 chip, had picked it for the Lisa, and it included a 24-bit address bus with 32-bit addresses.

But I couldn't get the 6502 out of my mind. Over another weekend I asked the question, how complicated would it be to expand the 6502 registers to 24-bits? Turns out, not very hard, at least not in Verilog. That again was a weekend project, with less than 100 lines of code and a handful of new states in the state machine. All with a much more orthogonal design than WDCs 65C816 and with 100% backward compatibility to the 6502 and 652402.

One learning in this nostalgic look at the late 1970s, early 1980s computer industry was the reminder of waterfall design. The norm of that era was an expectation of big step changes between design iterations, rather than the lean, incremental steps that are the norm since the dot com era of the 1990s.

Given the challenges of manual chip design and layout, waterfall makes logical sense for hardware, but Apple's strategy of abandoning the Apple II product line, expecting the Apple /// to fill all demand, in hindsight, was a foolish decision. At a minimum Apple should have divided up the team into business and consumer, with the /// aiming at the business market while allowing the II team to keep iterating on the successful design. Even better, there should have been a third team focused on chips, disks, and other shared technology, so that both product lines could benefit from double sided floppy drives, hard disk drives, better printers, monitors, etc.

Months after falling into the rabbit hole, it felt like I was out. I had reasonable answers to my question, at least from Apple's perspective, and I had working Verilog versions of my designs. I was done. I moved on to look for the next rabbit hole. Then a few months later I happened across one more article about the history of the Apple II, and rather than the typical story about how Woz picked the MOS 6502, this article talked about how Apple computers used the Synertek 6502B.

Synertek? Who was Synertek? What did they have to do with the 6502?

Hiding in plain site on Wikipedia is the story of Steve and Steve walking into Bob Schreiner's office asking for a line of credit to buy his 6502s. The Synertek 6502B. In the footnotes of that Wikipedia article is an interview with Bob, talking about that meeting, explaining why he said yes, and then many pages later, SURPRISE, it turns out Steve Jobs talked to Bob Schreiner just one other time.

It turns out Steve Jobs asked Synertek to build a 16-bit 6502!

So, in the end, it turns out someone did ask for an upgraded 6502. That someone was none other than Steve Jobs. That ask didn't go to MOS or WDC and thus this story was never known to Bill Mensch. That ask was literally just a footnote of history in a footnote in a Wikipedia article.

With that, I felt a little vindicated, but I also felt like my journey out of the rabbit hole was an illusion, as it felt like a little unused, never-to-be complied Verilog code was insufficient to show what could have been. Another weekend later I found the iz6502 emulator written in Go, I learned Go, and a week later had the 652402 emulated. A week after that, all the 652424 features were added, including the 8/16/24-bit adjustable-sized registers. And a week after that, the izapple2 emulator was running using that new CPU.

That was nice, but at that point it simply proved the 652424 was 100% backward compatible, as it was just running Apple II Plus and IIe ROMs. So I started writing OS4. That led me to relearn how challenging it is to write thousands of lines of assembly code.

Rather than relearn the old methods of unstructured code, I instead created the aCCembler, an assembler that takes the structural elements from a C compiler but the mnemonics from an assembler. That in itself was a month-long project, and another big lesson in hindsight vs. foresight, as nothing in my design was unknown in the 1970s, but again everyone was thinking about jumping to much higher-level languages rather than making assembly 10x faster to write, debug, and maintain.

With a working aCCembler in Go running on a modern ARM-based Mac Air, I had the ability to iterate the write / assemble / run / test process within seconds. That made writing OS4 a relatively easy and fun project. It has been over 30 years since I coded as a hobby instead of as a job, and it is enjoyable to see the visible progress as OS4 emerged from ideas to pixels on the emulated low-res screen.

The results of that work are documented as *Inside the Apple II4*, to show it to others who also grew up with an Apple II as their personal computer, and to see what might have been, if only Bob or Bill or anyone else in the 6502 world had the foresight and resources to out-design Motorola one more time.

And what might have been may have used a different CPU, but physically these variations may have looked familiar to the real timeline.

BONUS

THE ACCEMBLER

The INSIDE Story on the ACCembler

Beyond the file-centric model of OS4, one other huge innovation came from the trip to Bell Labs, the ACCembler. AT&T's UNIX operating system was written in a computer programming language called "C". C was an enormous step up from assembly language. A high-level language, but a very different high level as compared with BASIC.

C was designed for professional computer programmers, using computers with a lot of memory, running on CPUs with a lot more functionality than the 6502 or 652402. Because of this, and because C was designed around the UNIX operating system, programs in C tend to be much larger than programs written in BASIC and much much larger than programs written in assembly.

For these, and many other reasons, C wasn't a great fit for the Apple][4, but there were still a multitude of lessons learned that were compiled (pun intended) into the ACCembler, which, as its name implies, is a cross between an assembler of assembly language and a C compiler.

The core ideas.

The design of the ACCembler is wrapped around a few core ideas:

1. Assembly language for speed and efficiency.
2. Structured programming.
3. High level structure + low level assembly language.
4. Named variables.
5. Subroutine parameters and return values.
6. Data blocks.

1. Assembly language for speed and efficiency.

No high level language will ever be as small, fast, and efficient as well-written assembly code. The ACCembler doesn't try and compete in any of those areas. Instead, most of the code for an ACCembler program is written in assembly, with the same mnemonics and identical syntax to assembly language. For example:

```
LDA #' 'H          ; ERASE THE CURRENT CHARACTER
LDY @TXTPOS
DEY
STA (@TXTPTR),Y
```

```
/*
 *  The code begins with a JUMP TABLE
 */
sub Start() @$FF0000 {
     jmp Reset
}

/*
 *  RST vector points here
 */
sub Reset() @$FF0400 {
     ...
     r16                          ; set the stack to be 16-bit wide
     sws
     ldx.w #TOP_OF_STACK          ; initialize the stack at $FEFF
     txs.w
     ...
     ; drop through to CommandLoop
}

/*
 *  Interpret and run commands
 *  echoing the keys as they are typed and store in @TYPBUFFER
 */
sub CommandLoop() {
     jsr ClearInputBuffers
     LOOP {
          jsr SetScreenBase       ; Set the @TXTPTR based on @TXTROW
          lda @TXTPOS
          IF == {
               lda @PMPTCHR       ; Draw the prompt
               jsr PrintChar
               jsr PrintCursor    ; Draw the cursor
          }
          LOOP {
               bit KBD            ; check keyboard for next key
               IF - {
                    BREAK
               }
          }

          lda KBD                 ; get the key code and ack the read
          bit KBDSTRB
          IF == $8D {             ; 13 = CR ($80 | $0D in high ASCII)
               ...
          }
     }
     ; loop forever
}
```

2. Structured programming.

BASIC includes many of the core ideas of structured programming, but C takes this idea much further, splitting programs into a collection of “functions” (similar to BASIC’s subroutines), and commonly splitting programs into multiple files as well.

The ACCembler does the same. Blocks of code are split into named SUB sections.

Within a function, C includes loops and conditionals that can each include blocks of code. C uses the { and } characters to mark the start and end of those blocks. The ACCembler does the same, copying all of that syntax.

See the example ACCembler code at the end of this chapter for the use of SUB { ... } blocks and other {} blocks.

3. High level structure + low level assembly language.

Inside the blocks, C programs look a lot more like BASIC than assembly. Instead, most of ACCembler code is assembly code. All of the mnemonics of assembly instructions are allowed within ACCembler programs. All of the commonly used techniques for writing assembly are allowed. But in addition, C keywords like FOR, BREAK, IF, and RETURN are allowed too.

These keywords make ACCembler code much easier to understand and modify than typical assembly code. A very simple example:

ASSEMBLY
```
KBDWAIT  BIT KBD       ; WAIT TILL NEXT KEY TO RESUME
         BPL KBDWAIT   ; WAIT FOR KEYPRESS
```

ACCEMBLER
```
LOOP {
     BIT KBD           ; CHECK KEYBOARD FOR NEXT KEY
     IF - {
          BREAK
     }
}
```

The assembly code loop requires a label, which then doubles as an address that is often used elsewhere in the assembly code, creating unstructured jumps that are a challenge to follow when trying to understand or modify other people’s code.

Furthermore, while expert assembly programmers see BPL (branch if positive) and know this loops until the opposite is true (when the value is negative), this opposite logic confuses new assembly programmers, and especially causes issues on edge cases like when the value is zero.

The ACCembler code is obviously a loop, thanks to the LOOP syntax, and obviously the loop stops when the value is negative. And every programmer knows zero isn't negative.

This is a rather trivial example but take a look at the example ACCembler code at the end of this chapter to see for yourself how much more obvious the structure of the code is when split into {...} blocks and organized by LOOP, FOR, and IF keywords.

4. Named variables.

Most assemblers include labels, which can be used to specify addresses, including addresses used to hold variables. The challenge in doing that in most assembly programs is that all of those variables are defined at the start of the program.

In a high-level language like BASIC or C, variables can be defined inside subroutines. The ACCembler allows that too. Which then allows the same address to be reused over and over in different subroutines.

And yes, that can cause bugs at times, but with sufficient care in the structure of the program, that is easy to avoid.

In the following example, two values are specified, "I" stored in zero page address #$04 and "I4" stored in zero page address #$05.

```
SUB COMMANDLOOP() {
     VAR I    = @$04         ; LOOP VARIABLE
     VAR I4   = @$05         ; I * 6

     FOR @I = 0 TO N_COMMANDS-1 {
          LDA @I             ; LOAD THE LOOP VARIABLE INTO A
          A <<= 2            ; A SHIFTED LEFT TWICE = A * 4
          STA @I4            ; STORE I*4 INTO I4
          ...
     }
}
```

The variables are defined using the VAR keyword inside the named SUB subroutine. Those variables can only be used within that one named subroutine. It is a syntax error while compiling if the variables are named elsewhere, although the same names can be reused in other subroutines.

"Global" variables can be named at the start of the ACCembler code, outside of all subroutines.

5. Subroutine parameters and return values.

Subroutines can specify parameters by name too. The simplest version of these simply inform the programmer whether registers A, X, or Y should include certain values before the subroutine is called. E.g.:

```
SUB SETSCREENMODE(MODE A) {
     LDX @DISPMODE      ; REMEMBER THE PREVIOUS DISPLAY MODE
     STX @PREVDISP

     STA @DISPMODE      ; STORE THE NEW DISPLAY MODE
     ...
}
```

Parameters don't have to be passed in registers. The ACCembler allows parameters to be passed using memory $D000-$DFFF, which on the memory map is labeled "Command line and subroutine parameters". E.g.:

```
/*
 *  COMPARE TWO STRINGS
 *  RETURN IN A = #FF IF MATCH, #00 IF NOT
 */
SUB COMPARETEXT(STR_A %%7.T, STR_B %%7.3.T) {
     FOR Y = 0 TO 127 {
          LDX.T @STR_A
          LDA.A24 XY
          LDX.T @STR_B
          CMP.A24 XY
          IF != {
               RETURN 0
          }
          IF == 0 {
               LDA.A24 XY
               IF == 0 {
                    RETURN $FF
               }
               RETURN 0
          }
     }
     RETURN 0
}
```

The ACCembler uses a syntax prefixed by %% to specify the memory locations, rather than using actual address values. The syntax is %%N.M, where N is a value from 0 through 7 and M is a value from 0 through 255. These values map to memory location $DNMM. E.g. %%0 = $D000, %%1.3 = $D103, %%7.64 = $D180, etc.

In C, parameters are passed into functions on a stack, and this syntax lets a programmer more or less copy that idea, but manually, as long as the various subroutines are organized into layers. The top layer of subroutines should use the %%1 values, and should not call each other. The next layer of subroutines should use %%2 values, and again make sure not to call each other, nor any higher level subroutine. The naming system allows for seven layers of subroutines and up to 255 bytes of parameters per layer. That should be more than sufficient.

The reason for the manual system is that the 6502 and 652402 have no simple method for looking up a value relative to the top of the stack. The mini computers that run UNIX do have an addressing mode to make that easy.

Similarly, C can pass values back on the stack. The ACCembler instead uses register A. An ACCembler program can use the RETURN keyword to store a constant into register A and RTS out of the subroutine. Or the program can store the value itself and just use the RETURN keyword on its own with no value to generate the RTS assembly language.

6. Data blocks.

Assembly programs usually put all the data addresses and assembler-directives for data constants at the top of the assembly file. That was needed when assembler ran through the program file once, assembling from top to bottom. For structured programs, and for complex programs written in multiple files, that isn't always the most logical way to organize the data portion of a program.

To help with that logic, the ACCembler includes DATA blocks. Each block can specify a name, an optional starting memory address, whether the data is a BYTE, WORD or 24-bit values, and using {}s cleanly delineate the values to be stored. E.g.:

```
/*
 *  THE ADDRESSES OF THE 24 TEXT ROWS
 */
DATA TEXTSCREENBASE @$FF8000 WORD {
     $0400, $0480, $0500, $0580, $0600, $0680, $0700, $0780
     $0428, $04A8, $0528, $05A8, $0628, $06A8, $0728, $07A8
     $0450, $04D0, $0550, $05D0, $0650, $06D0, $0750, $07D0
}
```

The data block can also store a string of characters, with the ACCembler automatically added a zero termination byte.

```
/*  THE TITLE OF THE RESET SCREEN */
DATA HELLO STRING {
     "APPLE ][4"
}
```

More...

There is quite a lot more functionality in the ACCembler than mentioned here, but those are the core ideas that show how the ACCembler is far more than an assembler and a big step toward a high level language, while not giving up any of the size, speed, or efficiency of assembly language.

If you didn't notice, the other obvious syntax are the various methods for commenting the code. The /* ... */ syntax is copied from C, as is the "//" syntax, while the ';' from assembly is allowed too.

OS4 was written using the ACCembler, and the result of that effort clearly shows how this level of structure helps make the code more readable and far far far more easily modified. Especially when multiple programmers are working together on such a complex program as an operating system.

To see this yourself, the following pages includes a portion of that source code.

```
/*
 *  ROM for the Apple ][4
 *
 */

#include "OS4:globals.ac"

const N_COMMANDS   = 21  // The number of commands in CommandList


/*
 *  The code begins with a JMPs to the OS functions
 *  followed by a table of addresses for JSR ($aaaa) indirect calls
 */
sub Start() @$FF0000 {
     jmp Reset
}

/*
 *  RST vector points here
 */
sub Reset() @$FF0400 {
     cld
     sta LANGSET             ; turn on ][+ "upper" 16K (write twice)
     sta LANGSET
     r16                     ; set the stack to be 16-bit wide
     sws
     ldx.w #TOP_OF_STACK     ; initialize the stack at $FEFF
     txs.w

     lda TXTSET              ; Start in TEXT mode
     lda TXTPAGE1            ;  Page 1
     lda SETAN0              ; AN0 = TTL hi
     lda SETAN1              ; AN1 = TTL hi
     lda CLRAN2              ; AN2 = TTL lo
     lda CLRAN3              ; AN3 = TTL lo
     lda CLRROM              ; turn off extension ROM
     bit KBDSTRB             ; clear keyboard

     lda #2
     sta @TXTROW             ; Start at TEXT row 2
     stz @TXTPOS             ; Start at TEXT position 0

     jsr ClearLoRes          ; Clear LORES page 2
     jsr ClearGraphics       ; Clear GRAPHICS
                             ;  (which overlaps with HIRES page 2)

     lda #DISP_80COL         ; Setup and clear 80-col TEXT screen
     jsr SetScreenMode
     ldx #0
     jsr ClearScreen         ; Clear 80-col TEXT page

     lda #DISP_TEXT_1        ; Start in TEXT mode, page 1
     jsr SetScreenMode
     ldx #0
     jsr ClearScreen    ; Clear TEXT page 1
     jsr AppleII4       ; draw the header

     lda #DISP_64COL    ; Setup and clear 64-col TEXT screen
     jsr SetScreenMode
     ldx #0
     jsr ClearScreen    ; Clear 64-col TEXT page
     jsr AppleII4       ; draw the header on the 64-col page

     lda #'_'           ; Initialize the cursor
     ora #$40
     sta @CURCHR

     lda #':'H          ; Initialize the prompt
     sta @PMPTCHR

     lda.w #II4GR_START ; Reset GRPTR to GRAPHICS memory
     sta.w @GRPTR
     stz.w @GRX         ; Reset GRAPHICS coordinate to 0,0
     stz.w @GRY
     lda #$FE           ; Default to 16 pixel tall ROM Font
     sta @GRFONT

     lda.t #_64K        ; Reset the memory heap to $10000 (64K)
     sta.t @HEAPSTART
     stz.t @RUNLENGTH

     jsr FindTopOfRAM   ; search for the highest RAM address
     ; drop through to CommandLoop
}

/*
 *  Interpret and run commands
 *  echoing the keys as they are typed and store in @TYPBUFFER
 */
sub CommandLoop() {
     jsr ClearInputBuffers
     LOOP {
          jsr SetScreenBase  ; Set the @TXTPTR based on @TXTROW
          lda @TXTPOS
          IF == {
               lda @PMPTCHR       ; Draw the prompt
               jsr PrintChar
               jsr PrintCursor    ; Draw the cursor
          }
          LOOP {
               bit KBD       ; check keyboard for next key
               IF - {
                    BREAK
               }
          }
```

```
lda KBD            ; get the key code and ack the read
bit KBDSTRB
IF == $8D {        ; 13 = CR ($80 | $0D in high ASCII)
     jsr ClearCursor
     jsr NextLine
     jsr CurrentToPrevInputBuffer
     jsr CommandLine
     jsr ClearCurrentInputBuffer
     CONTINUE
}
IF == $FF {        ; 127 = DEL ($80 | $7F in high ASCII)
     lda @TXTPOS
     IF > 1 {
          jsr ClearCursor    ; Erase the cursor
          lda #' 'H          ; Erase the current character
          ldy @TXTPOS
          dey
          sta (@TXTPTR),y
          dey  ; Delete the last character in the text buffer
          lda #0
          sta @TYPBUFFER,y
          dec @TXTPOS        ; Decrement the text position
          jsr PrintCursor    ; Draw the cursor
     }
     CONTINUE
}
IF == $88 {        ; 8 = BS ($80 | $08 in high ASCII)
     lda @TXTPOS
     IF > 1 {
          jsr ClearCursor    ; Erase the cursor
          lda #' 'H          ; Erase the current character
          ldy @TXTPOS
          dey
          sta (@TXTPTR),y
          dey  ; Delete the last character in the text buffer
          lda #0
          sta @TYPBUFFER,y
          dec @TXTPOS        ; Decrement the text position
          jsr PrintCursor    ; Draw the cursor
     }
     CONTINUE
}
IF == $89 {        ; 9 = Tab ($80 | $09 in high ASCII)
     jsr CompleteCommand
     CONTINUE
}
IF == $8B {        ; 11 = Up Arrow ($80 | $0B in high ASCII)
     jsr PrevToCurrentInputBuffer
     jsr InputBufferToTextScreen
     jsr PrintCursor         ; Draw the cursor
     CONTINUE
}
IF == $9B {        ; 27 = ESC ($80 | $1B in high ASCII)
     ldx @TXTROW             ; Clear the current text buffer
```

```
                jsr ClearRow
                jsr ClearCurrentInputBuffer
                lda #0
                sta @TXTPOS         ; Reset to position 0

                lda @PREVDISP       ; Return to the previous diplay mode
                                    ;  (likely TEXT page 1 or 80col)
                jsr SetScreenMode

                CONTINUE
            }
            IF > $9F {     ; >31 = Not a control character
                           ; ($80 | $1F in high ASCII)
                ldy @TXTPOS
                jsr PrintChar
                cpy #40             ; Ignore IF >= column 40
                IF < {
                    and #$7F        ; store (low) ASCII in TEXT buffer
                    dey             ; Y-1 as TEXT position starts at 1,
                                    ;  making space for the prompt
                    sta @TYPBUFFER,y
                    jsr PrintCursor
                }
            }
        }
        ; loop forever
    }


    /*
     *  Lookup and execute the typed command
     */
    sub CommandLine() {
        var I     = @$04        ; loop variable
        var I6    = @$05        ; I * 6

        lda @TYPBUFFER          ; Check for blank line
        if (==) {
            RETURN
        }

        lda.t #@TYPBUFFER
        sta.t %%7
        jsr ParseInputParams

        lda.t %%0.1             ; strA for CompareText
        sta.t %%7
        FOR @I = 0 TO N_COMMANDS-1 {
            lda @I              ; X = @M$04 * 6 = (@M$04 * 4) + (@M$04 * 2)
            A <<= 2
            sta @I6
            lda @I
            A <<= 1
```

```
          clc
          adc @I6
          sta @I6
          tax

          lda.t CommandList,X     ; CommandList[X].string
          sta.t %%7.3
          jsr CompareTextCI
          IF - {
               ldx @I6
               lda.t CommandList+3,X   ; CommandList[X].func
               sta.t %R0
               jsr.a24 (%R0)
               RETURN
          }
     }

     ldx.t %%0.1                  ; Check for all spaces
     ldy #0
     LOOP {
          lda.a24 XY
          if (==) {
               RETURN
          }
          cmp #' '
          if (!=) {
               lda.t #UnknownCommandErr
               sta.t %%7
               jsr PrintError
               RETURN
          }
          iny
     }
}

/*
 *  Try and complete the command
 */
sub CompleteCommand() {
     var I     = @$04             ; loop variable
     var I6    = @$05             ; I * 6

     lda #@TYPBUFFER              ; %%7 <- TEXT buffer
     sta.t %%7
     FOR @I = 0 TO N_COMMANDS-1 {
          lda @I             ; X = @M$04 * 6 = (@M$04 * 4) + (@M$04 * 2)
          A <<= 2
          sta @I6
          lda @I
          A <<= 1
          clc
          adc @I6
          sta @I6
          tax
          lda.t CommandList,X     ; CommandList[X].string
          sta.t %%7.3
          jsr CompareStartOfTextCI
          IF - {
               ldx @I6
               lda.t CommandList,X     ; %%0.1 <- CommandList[X].string
               sta.t %%7
               lda #@TYPBUFFER         ; %%7.3 <- TEXT buffer
               sta.t %%7.3
               jsr CopyString
               lda.w @TXTPTR           ; @TXTPTR <- Current line on
                                       ; TEXT screen
               inc.w                   ;  +1 for the cursor
               sta.t %%7.3
               jsr CopyStringHigh
               iny
               sty @TXTPOS             ; CopyString leaves the length
                                       ; of the string in Y
               jsr PrintCursor
               RETURN
          }
     }

     RETURN
}

#include "OS4:util.ac"
#include "OS4:graphics.ac"
#include "OS4:disk4.ac"
#include "OS4:commands.ac"
#include "OS4:view.ac"
#include "OS4:data.ac"

/*
 *  IRQ vector points here
 */
sub Reset() @$FFFF00 {
     a24
     rti
}

/*
 *  The 6502 vectors
 */
data Vectors @$FFFFF7 u24 {
     $000000  ; NMI
     $FF0000  ; RESET
     $FFFF00  ; IRQ
}
```

```
FILE: OS4:globals.ac

/*
 *  ROM for the Apple ][4
 *
 *  Constants and globals
 *
 */

const KBD          = $c000  // R last key pressed + 128
const KBDSTRB      = $c010  // RW keyboard strobe
const TAPEOUT      = $c020  // RW toggle casSette tape output
const SPKR         = $c030  // RW toggle speaker
const TXTCLR       = $c050  // RW display graphics
const TXTSET       = $c051  // RW display text
const MIXSET       = $c053  // RW display split screen
const TXTPAGE1     = $c054  // RW display page 1
const TXTPAGE2     = $c055  // RW display page 2
const LORES        = $c056  // RW display lo-res graphics
const HIRES        = $c057  // RW display lo-res graphics
const SETAN0       = $c058  // RW annunciator 0 off
const SETAN1       = $c05a  // RW annunciator 1 off
const CLRAN2       = $c05d  // RW annunciator 2 on
const CLRAN3       = $c05f  // RW annunciator 3 on
const TXT80CLR     = $c060  // RW 80column mode off (was casette in)
const TXT80SET     = $c068  // RW 80column mode on (was casette in)
const TXT64CLR     = $c06E  // RW 64column mode off (was DUP PDL2)
const TXT64SET     = $c06F  // RW 64column mode on (was DUP PDL3 on)
const PADDL0       = $c064  // R analog input 0
const PTRIG        = $c070  // RW analog input reset
const LANGCLR      = $c082  // Disable "Language card" RAM
const LANGSET      = $c083  // Enable "Language card" RAM (2x write)
const CLRROM       = $cfff  // disable slot C8 ROM

const TOP_OF_STACK = $fdff  // Put the stack in the "Language card" RAM

const RSTVECTOR    = $fffc  // Apple ][ 6502 reset vector
const PWREDUP      = $03f4  // Apple ][ stores #$A5 to soft reboot

global TXTPTR      = @$fe.w // The address of the current line of text
                            //  (2 bytes as the TEXT screen is <$FFFF)
global TXTWIDTH    = @$fd   // The width of the current screen
                            //  (typically #40 or #80 columns)
global TXTROW      = @$ff00 // The current row on the text screen
global TXTPOS      = @$ff01 // The current position in the text screen
global TXTPGROWS   = @$ff02 // The number of rows output to the screen
                            //  (used for pagination)
global DISPMODE    = @$ff03 // The current display mode
                            //  (see DISP_xxxx constants)
global PREVDISP    = @$ff04 // The previous display mode
                            //  (see DISP_xxxx constants)
global CURCHR      = @$ff05 // The character to draw for the cursor
global PMPTCHR     = @$ff06 // The character to draw for the prompt
global GRPTR       = @$ff20.w    // The address of the current
                                 //  row of graphics
                                 //  (2 bytes as GRAPHICS is <$FFFF)
global GRX         = @$ff22.w    // The x coordinate for
                                 //  drawing II4 GRAPHICS
global GRY         = @$ff24.w    // The y coordinate for
                                 //  drawing II4 GRAPHICS
global GRFONT      = @$ff26      // The current font ID

const _64K         = $010000
global RAMTOP      = @$ff10.t    // The highest RAM memory address
global HEAPSTART   = @$ff13.t    // The start of the heap
                                 //  (initially $10000, then
                                 //    the page after the
                                 //     loaded program)
global RUNLENGTH   = @$ff16.t    // The length of the loaded program

global TYPBUFFER   = @$fe00      // $FE00-$FE4F holds
                                 //  the line of text being typed
global TYPBUFFER2  = @$fe50      // $FE50-$FEBF holds
                                 //  the previous line of text typed
global TYPBUFFER3  = @$feA0      // $FEA0-$FEEF holds
                                 //  the previous PREVIOUS line of text

const DISP_PAGE_1       = $00    // MSB is page 1|2
const DISP_PAGE_2       = $80
const DISP_40COL        = $00    // MSB-1 is 40|80 column
const DISP_64COL        = $48
const DISP_80COL        = $40
const DISP_TEXT         = $00
const DISP_TEXT_MASK    = $7F
const DISP_LORES        = $20
const DISP_HIRES        = $10
const DISP_II4          = $08

const DISP_TEXT_1  = $00         // PAGE 1 | 40COL | TEXT
const DISP_TEXT_2  = $80         // PAGE 2 | 40COL | TEXT
const DISP_LORES_1 = $20         // PAGE 1 | LORES
const DISP_LORES_2 = $A0         // PAGE 2 | LORES
const DISP_HIRES_1 = $10         // PAGE 1 | HIRES
const DISP_HIRES_2 = $90         // PAGE 2 | HIRES

const LORES_START  = $800
const HIRES_START  = $2000
const II4GR_START  = $4000
const II4GR_END    = $B800

const PAGINATE_LINES_PER_PAGE = 22
```

```
FILE: OS4:data.ac

/*
 *  ROM for the Apple ][4
 *
 *  Data blocks
 *
 */


/*
 *  The ROM vector table - a.k.a. all the OS4 subroutines
 */
data ROMVectors @$FF0005 u24 {
     Wait
     SetScreenMode
     SetScreenBase
     GetChar
     PrintChar
     PrintSpace
     PrintCursor
     ClearCursor
     PrintString
     PrintError
     PrintHexDigit
     PrintHexByte
     PrintHexWord
     PrintHex24
     ResetPaginate
     PaginateLine
     NextLine
     ScrollScreen
     ClearRow
     ClearScreen
     ClearLores
     ClearHires
     ClearGraphics
     CompareText
     CompareTextCI
     CompareStartOfText
     CompareStartOfTextCI
     CopyString
     CopyStringHigh
     ParseInputParams
     ParseNumber
     ParseHexNumber
     ParseDecimalNumber
     Times10
     Modulo
     PlayBeep
}

/*   The OS4 subroutines for Disk4 */
data DOSVectors @$FF0100 u24 {
     Disk4Name
     Disk4CatalogStart
     Disk4CatalogNext
     Disk4Exists
     Disk4Create
     Disk4Open
     Disk4Close
     Disk4Read
     Disk4Write
     Disk4C800
     Disk4PrintError
}

/*   The OS4 subroutines for the BitBlt4 chip */
data GraphicsVectors @$FF0200 u24 {
     GrSetBase
     GrSetFont
     GrSetX
     GrSetY
     GrTypesetChar
     GrTypesetString
     GrTypesetSpace
     GrTypesetCharBig
}
```

```
/*
 *  The addresses of the 24 TEXT rows
 */
data TextScreenBase @$FF8000 word {
     $0400, $0480, $0500, $0580, $0600, $0680, $0700, $0780
     $0428, $04A8, $0528, $05A8, $0628, $06A8, $0728, $07A8
     $0450, $04D0, $0550, $05D0, $0650, $06D0, $0750, $07D0
}

/*
 *  The addresses of the 24 64-column TEXT rows
 */
data Text64ScreenBase word {
     $2000, $20c0, $2180, $2240, $2300, $23c0, $2480, $2540,
     $2040, $2100, $21c0, $2280, $2340, $2400, $24c0, $2580,
     $2080, $2140, $2200, $22c0, $2380, $2440, $2500, $25c0
}

/*
 *  The addresses of the 24 80-column TEXT rows
 */
data Text80ScreenBase word {
     $2000, $2100, $2200, $2300, $2400, $2500, $2600, $2700,
     $2050, $2150, $2250, $2350, $2450, $2550, $2650, $2750,
     $20a0, $21a0, $22a0, $23a0, $24a0, $25a0, $26a0, $27a0
}

/*
 *  The offsets for the first 64 of 384 rows in the GRAPHICS screen
 */
data GraphicsOffsets word {
     $0000, $0040, $0080, $00C0, $0100, $0140, $0180, $01C0
     $0200, $0240, $0280, $02C0, $0300, $0340, $0380, $03C0
     $0400, $0440, $0480, $04C0, $0500, $0540, $0580, $05C0
     $0600, $0640, $0680, $06C0, $0700, $0740, $0780, $07C0
     $0800, $0840, $0880, $08C0, $0900, $0940, $0980, $09C0
     $0A00, $0A40, $0A80, $0AC0, $0B00, $0B40, $0B80, $0BC0
     $0C00, $0C40, $0C80, $0CC0, $0D00, $0D40, $0D80, $0DC0
     $0E00, $0E40, $0E80, $0EC0, $0F00, $0F40, $0F80, $0FC0
}


/*  The title of the reset screen */
data Hello string {
     "Apple ][4"
}

/*  The header when listing commands */
data CommandListStr string {
     "COMMANDS:"
}

/*  The program has ended */
data EndProgramStr string {
     ">> END PROGRAM"
}

/*  The headers when printing the ASCII charset */
data ASCIIHeaderStr string {
     "  0123456789ABCDEF"
}
data ASCIIDashesStr string {
     "  ----------------"
}

/*  The header for the catalog */
data CatalogHeaderStr string {
     "CATALOG for "
}

/*  Error string for load, view, etc. */
data UnknownCommandErr string {
     "*** Unknown COMMAND - see '?'"
}
data NoFilenameErrStr string {
     "*** FILENAME must be specified"
}
data NoFontNameErrStr string {
     "*** FONT NAME must be specified"
}
data NoFileTypeErrStr string {
     "*** FILE TYPE must be specified"
}
```

```
data NoAddressErrStr string {
       "*** ADDRESS must be specified"
}
data NoValueErrStr string {
       "*** VALUE must be specified"
}
data NoLengthErrStr string {
       "*** LENGTH must be specified"
}
data EndBeforeStartErrStr string {
       "*** The END value is less than START"
}
data NoEditorForDirectoriesErr string {
       "*** You cannot EDIT a directory"
}
data NoEditorFoundErr string {
       "*** No EDIT program was found"
}
data NoFontFoundErr string {
       "*** The FONT was NOT found"
}
data NotFontFileErr string {
       "*** The file is not of type FNT"
}
data FontLoadFailedErr string {
       "*** FONT load FAILED"
}
data LoadedStr string {
       "LOADED "
}
data LoadedBytesToStr string {
       " bytes to $"
}

data NotRUNFileError string {
       "*** FILE Must be a RUN file"
}

/*  General error message for bad argument */
data InvalidValue string {
       "*** INVALID VALUE"
}
```

```
/*  The header for the catalog */
data Disk4ErrorStr string {
       "*** DISK4 ERROR"
}
data Disk4ErrorNotFoundStr string {
       "*** DISK4: FILE NOT FOUND"
}
data Disk4ErrorExistsStr string {
       "*** DISK4: FILE ALREADY EXISTS"
}
data Disk4ErrorReadStr string {
       "*** DISK4: READ ERROR"
}

/*  The waveform of the beep tone */
data BeepTimes byte {
       220, 220, 220, 60, 60, 60, 60, 60,
}


/*
 *  Built in fonts
 */
data ROM_Fonts @$FFA000 u24 {
       $FFA020,
       $FFA600
       $FFB400
       $FFC900
}
data FONT_Apple7x8 @$FFA020 file { "fonts/charset7x8.fnt" }
data FONT_Apple14x16 @$FFA600 file { "fonts/charset14x16.fnt" }
data FONT_Chicago @$FFB400 file { "fonts/Chicago.fnt" }
data FONT_apple @$FFC900 file { "fonts/Motter.fnt" }
```

```
/*
 *  The list of commands
 */
data CMD_Help @$FFF800 string { "?" }
data CMD_Ascii string { "ascii" }
data CMD_Beep string { "beep" }
data CMD_Catalog string { "catalog" }
data CMD_Clear string { "clear" }
data CMD_Edit string { "edit" }
data CMD_Font string { "font" }
data CMD_Graphics string { "graphics" }
data CMD_Hires string { "hires" }
data CMD_Line string { "line" }
data CMD_Load string { "load" }
data CMD_Lores string { "lores" }
data CMD_Peek string { "peek" }
data CMD_Poke string { "poke" }
data CMD_Run string { "run" }

/*
 *  The table of command names and address vectors
 */
data CommandList u24 {
       CMD_Help, DoHelp
       CMD_Ascii, DoAscii
       CMD_Catalog, DoCatalog
       CMD_Beep, PlayBeep
       CMD_Clear, DoClear
       CMD_Edit, DoEdit
       CMD_Font, DoFont
       CMD_Graphics, DoGraphics
       CMD_Hires, DoHires
       CMD_Line, DoLine
       CMD_Load, DoLoad
       CMD_Lores, DoLores
       CMD_Peek, DoPeek
       CMD_Poke, DoPoke
       CMD_Run, DoRun
}
```

BONUS

THE 650024

The Rabbit Hole Once More…

The 652402 proved that the 6502 CPU could be expanded to 24-bit address space to create a far more powerful microcomputer. But that design has quite a few limitations, including just three general purpose registers and an 8-bit data bus. Once 24-bit addresses became the norm, the next logical design step was a fully 24-bit CPU.

What makes an 8-bit CPU 8-bit is not just the width of the data bus and not just the width of the registers, but also the width of the instructions. A 16-bit CPU has 16-bit instructions. A 24-bit CPU has 24-bit instructions.

Rather than using that expanded space to add million of new instructions, that expanded space can be used to reference a multitude of registers.

The 650024 is a 24-bit CPU that replicates the core instructions of the 6502, but which includes 256 registers instead of just A, X, and Y. Each instruction is 3 bytes long (24-bits), with the first byte encoding the operation, the second encoding one register address, and the third encoding a second register address (or for some opcodes, other values).

operation	register A	register B

For example, ***ADD24 R6 R9*** adds the 24-bit value in register 6 to the 24-bit value in register 9, storing the result in register 9. Similarly, the instruction ***OR16 R125 R250*** performs a logical or on the 16-bit values in registers 125 and 250, storing the result in register 250.

How can a CPU have 256 registers? Simple. The 6502 designers talked about how the zero page address mode was akin to addressable registers. The 650024 builds upon that idea, using memory addresses 0-255 as the register storage, in RAM, outside the CPU.

As the speed of future 650024 implementations increase, a subset of that memory space can be stored within the CPU to ensure access to the registers keep up with the CPU speed. Someday, when CPUs have hundreds of thousands of transistors, all 256 bytes of registers can be included within the CPU itself.

Note that the register set is not 256 24-bit registers, but 256 bytes of storage, addressable as 256 8-bit registers or 128 16-bit registers or 85 24-bit registers, or any mix of sizes. The register name is the memory address of the first byte of the register, with the bytes stored in little endian, i.e. the least significant byte first and the most significant byte last, the same way the 6502 reads and writes 16-bit values in RAM and the same way the 652402 reads and writes 16-bit and 24-bit values in RAM.

Each 650024 instruction that uses registers comes in three sizes: OP8, OP16, and OP24, specifying whether the value in the registers are accessed as 8-bit, 16-bit, or 24-bit values. Both registers must be the same size. Thus, if an 8-bit counter is being added to a 24-bit address, the 8-bit value must use three consecutive register addresses, with the two most significant bytes set to zero.

Just as not every 6502 instruction references a register, the same is true for the 650024. The instructions that use two registers include: ADC, ADD, SBC, SUB, AND, EOR, OR, CMP, and CP. The following instructions use a single register: ADI, BIT, DEC, INC, PSH, PLL, SBI, SL, SR, RL, RR, and CPU. Some of the LD and ST address modes use one register, some use two, and some use none. Similarly, the address modes vary for JMP and JSR. And some instructions do not use any registers: RTS, RTI, BRK, CLR, SET, PHF, PLF, and all the various BRAnch instructions.

If you are familiar with the 6502 instructions, most of these opcodes should look familiar. The 650024's instructions are very similar, with a few of the names simplified, a few extra variations added, and thanks to the 3-byte instruction length, some instruction families simplified down to a single instruction.

Load/Store

Let's start by looking at the instructions for loading values from memory into registers and storing values from registers into memory. This is the biggest change from the 6502. The only instructions that access memory are LD (load) and ST (store). All other instructions can only modify the values in registers or modify the state of the CPU, with the exception of the PSH (push) and PLL (pull) instruction for modifying the stack.

MODE	WIDTH	SYNTAX	HEX	LEN
Immediate	8-bits	LD #$45 Rb	$10	3
Immediate	16-bits	LD #$4567 Rb	$11	6
Immediate	24-bits	LD #$456789 Rb	$12	6
Absolute	8-bits	LD8 $654321 Rb	$14	6
Absolute	16-bits	LD16 $654321 Rb	$15	6
Absolute	24-bits	LD24 $654321 Rb	$16	6
Register	8-bits	LD8 Ra Rb	$18	3
Register	16-bits	LD16 Ra Rb	$19	3
Register	24-bits	LD24 Ra Rb	$1A	3
Reg+Next	8-bits	LD8 Ra+ Rb	$1C	3
Reg+Next	16-bits	LD16 Ra+ Rb	$1D	3
Reg+Next	24-bits	LD24 Ra+ Rb	$1E	3

The LD (load) operations have four addressing modes: Immediate, Absolute, Register, or Register+Next.

Immediate mode loads a value stored in the opcode into a register. This value is stored in the second register byte for 8-bit values. For 16-bit and 24-bit values, this is stored in the 3 bytes after the opcode, making the whole instruction 6 bytes long. This is similar to the 6502 whose instructions can be 1-byte, 2-byte, or 3-bytes long.

Absolute mode loads the value stored in the specified memory location. The memory address is always specified as a 24-bit address, and thus the whole instruction is always 6 bytes long.

Register mode loads the value into register B that is stored in the memory address specified by register A.

Register+Next mode is the same as ***Register*** mode, except that the address stored in register A is incremented by 1, 2, or 3, matching the width of the value being loaded. This mode allows for tight loops that load a series of values into a set of registers, without having to ADD or INC the index register A in a separate operation.

The address modes for the ST (store) operations are the same as for LD, except that there is no ***Immediate*** mode for storing values. To store a specific value, the value must first be loaded into a register.

MODE	WIDTH	SYNTAX	HEX	LEN
Absolute	8-bits	ST8 $654321 Rb	$14	6
Absolute	16-bits	ST16 $654321 Rb	$15	6
Absolute	24-bits	ST24 $654321 Rb	$16	6
Register	8-bits	ST8 Ra Rb	$18	3
Register	16-bits	ST16 Ra Rb	$19	3
Register	24-bits	ST24 Ra Rb	$1A	3
Reg+Next	8-bits	ST8 Ra+ Rb	$1C	3
Reg+Next	16-bits	ST16 Ra+ Rb	$1D	3
Reg+Next	24-bits	ST24 Ra+ Rb	$1E	3

Copying Registers

The value in one register can be copied into another register using the CP opcode, which comes in three sizes: 8-bits, 16-bits, and 24-bits.

MODE	SYNTAX	HEX	LEN
8-bits	CP8 Ra Rb	$20	3
16-bits	CP16 Ra Rb	$21	3
24-bits	CP24 Ra Rb	$22	3

Stack

The PSH and PLL opcodes push and pull registers from the stack. Each operation comes in three sizes: 8-bits, 16-bits, and 24-bits.

MODE	SYNTAX	HEX	LEN
8-bits	PSH8 Ra	$30	3
16-bits	PSH16 Ra	$31	3
24-bits	PSH24 Ra	$32	3
8-bits	PLL8 Ra	$34	3
16-bits	PLL16 Ra	$35	3
24-bits	PLL24 Ra	$36	3

The PHF and PLF opcodes push and pull the processor flags from the stack. This is always an 8-bit value.

PHF (PusH processor Flags)	$3E	3
PLF (PuLl processor Flags)	$3F	3

Note that the stack pointer is the 24-bit register R250. Using a register for this value eliminates the need for special opcodes to manage the stack pointer.

Arithmetic

The operations for adding (ADC and ADD) and subtracting (SBC and SUB) all specify two registers, A and B, and all store the result in register B. All of these operations also specify a width, 8-bit, 16-bit, or 24-bit.

MODE	SYNTAX	HEX	LEN
8-bits	ADC Ra Rb	$54	3
16-bits	ADC Ra Rb	$55	3
24-bits	ADC Ra Rb	$56	3
8-bits	ADD Ra Rb	$50	3
16-bits	ADD Ra Rb	$51	3
24-bits	ADD Ra Rb	$52	3
8-bits	SBC Ra Rb	$5C	3
16-bits	SBC Ra Rb	$5D	3
24-bits	SBC Ra Rb	$5E	3
8-bits	SUB Ra Rb	$58	3
16-bits	SUB Ra Rb	$59	3
24-bits	SUB Ra Rb	$5A	3

Both addition and subtraction have opcodes with and without the use of the carry flag. This saves the extra operation for clearing or setting the carry flag that is often found in 6502 assembly code. While that extra operation is just one extra byte on a 6502, it is three extra bytes on a 650024 as the smallest operations are three bytes long.

There is also an ADI opcode for adding an 8-bit value to the value in a register and SBI opcode for subtracting an 8-bit value from a register. This allows for a 3-byte operation to increment a register value by 2, to jump between 16-bit values or by 3, to jump between 24-bit values without having to use multiple INC instructions.

MODE	SYNTAX	HEX	LEN
8-bits	ADI #V Rb	$60	3
16-bits	ADI #V Rb	$61	3
24-bits	ADI #V Rb	$62	3
8-bits	SBI #V Rb	$64	3
16-bits	SBI #V Rb	$65	3
24-bits	SBI #V Rb	$66	3

Increment and Decrement

All 650024 compilers should provide mnemonics for INC and DEC, encoding the opcodes as ADI and SBI instructions with 1 as the immediate value.

Shift and Roll

The 650024 includes a symmetric set of opcodes for shifting and rolling the bits within a register. Each shift or roll shifts one bit left or right. The roll opcodes use the carry flag for a 9th, 17th, or 25th bit.

MODE	SYNTAX	HEX	LEN
8-bits	SL Ra	$70	3
16-bits	SL Ra	$71	3
24-bits	SL Ra	$72	3
8-bits	SR Ra	$74	3
16-bits	SR Ra	$75	3
24-bits	SR Ra	$76	3
8-bits	RL Ra	$78	3
16-bits	RL Ra	$79	3
24-bits	RL Ra	$7A	3
8-bits	RL Ra	$7C	3
16-bits	RL Ra	$7D	3
24-bits	RL Ra	$7E	3

Bit test

The 650024 includes the BIT opcode for setting various processor flags, like the 6502, but only with values stored in registers.

MODE	SYNTAX	HEX	LEN
8-bits	BIT Ra	$D4	3
16-bits	BIT Ra	$D5	3
24-bits	BIT Ra	$D6	3

Bitwise Logic

The operations bitwise logic (AND, OR, and EOR) all specify two registers, A and B, and all store the result in register B. All of these operations also specify a width, 8-bit, 16-bit, or 24-bit.

MODE	SYNTAX	HEX	LEN
8-bits	AND A B	$40	3
16-bits	AND A B	$41	3
24-bits	AND A B	$42	3
8-bits	OR Ra Rb	$44	3
16-bits	OR Ra Rb	$45	3
24-bits	OR Ra Rb	$46	3
8-bits	EOR Ra Rb	$48	3
16-bits	EOR Ra Rb	$49	3
24-bits	EOR Ra Rb	$4A	3

Processor Flags

The 650024 includes the same processor flags as the 6502, with an additional X flag that is not changed by any opcodes other than SET and CLR, usable by the programmer as a programmable flag.

MNEMONIC	HEX
CLR f (CLeaR flags)	$8E
SET f (SET flags)	$8F

Rather than individual opcodes for each flag, the CLR and SET opcodes can set or clear one or more flags in a single operation, with a specified bitmask to determine which flags are updated. The syntax is the name of the flag, e.g. CLR INZCVX to clear all of the flags or SET C to set the carry flag.

Various opcodes set or clear the C, V, Z, and N flags, just like on the 6502. The biggest difference is that the width of the register determines how many bits or which bits in the opcode value to set the flags. Like the 6502, the CMP opcode can set these flags, as can other opcodes. Unlike the 6502, the 650024's CMP opcode only compares two registers or one register with an immediate, 8-bit value.

MODE	SYNTAX	HEX	LEN
8-bits	CMP Ra Rb	$80	3
16-bits	CMP Ra Rb	$81	3
24-bits	CMP Ra Rb	$82	3
8-bits	CMP #V Rb	$84	3
16-bits	CMP #V Rb	$85	3
24-bits	CMP #V Rb	$86	3

Branches

The 650024 includes the same set of branches as the 6502, plus BRA to always branch. All branches are relative to the current program counter. All the branches can either be 3-bytes long with a jump forward or backward 16383 bytes or a 6-byte long instruction with a jump forward or backward 16777215 bytes. The compiler is expected to set the most significant bit of the second instruction byte to encod the length.

MNEMONIC	HEX
BRA (Branch Always)	$90
BEQ (Branch on EQual)	$91
BNE (Branch on Not Equal)	$92
BCC (Branch on Carry Clear)	$93
BCS (Branch on Carry Set)	$94
BPL (Branch on PLus)	$95
BMI (Branch on MInus)	$96
BVC (Branch on oVerflow Clear)	$97
BVS (Branch on oVerflow Set)	$98
BXC (Branch on X flag Clear)	$99
BXS (Branch on X flag Set)	$9A

Jumps

The JMP opcode has three address modes for specifying the address of the next opcode. Absolute mode specifies a memory address. Register mode uses the 24-bit value stored in the specified register. Reg+Reg mode adds the 24-bit values of two registers and jump to that address, leaving the values in the two registers unchanged.

MODE	SYNTAX	HEX	LEN
Absolute	JMP $654321	$04	6
Register	JMP Ra	$05	3
Reg+Reg	JMP Ra+Rb	$06	3

Subroutines

The JSR opcode has the same three address modes as JMP, plus two more that use relative addresses, similar to the branch opcodes. JSRS is a 3-byte opcode that specifies an address forward or backward 32767 bytes from the current program counter. JSRL is a 6-byte opcode that allows for jumps forward or backward 16777215 bytes.

MODE	SYNTAX	HEX	LEN
Absolute	JSR $654321	$08	6
Register	JSR Ra	$09	3
Reg+Reg	JSR Ra+Rb	$0A	3
Relative	JSRS +disp	$0B	3
Relative	JSRL +disp	$0C	6

All JSR opcodes push the 24-bit address of the next instruction on the stack. The RTS opcode pulls a 24-bit value from the stack and sets the program counter to that value.

MODE	SYNTAX	HEX	LEN
Implied	RTS	$60	3

Interrupts

When the interrupt pin is set, the 650024 pushes an 8-bit value with the processor flags on the stack, followed by the 24-bit address of the current opcode. The RTI opcode pulls those values, resets the program counter and processor flags, and continues processing where the interrupt occurred.

MODE	SYNTAX	HEX	LEN
Implied	RTI	$01	3

Break

The BRK opcode causes a software-defined interrupt.

MODE	SYNTAX	HEX	LEN
Implied	BRK	$00	3

No operation

The NOP opcode performs no operation. Unlike on the 6502, where each NOP consumes one cycle, on the 650024, the NOP opcode can specify up to 16383 cycles. Interrupts can interrupt a NOP, shortening that delay.

MODE	SYNTAX	HEX	LEN
1-cycle	NOP	$03	3
N-cycles	NOP n	$03	3

CPU model & features

The CPU opcode load the CPU features into a register, storing the CPU version number in the least significant byte, the number of registers built into CPU in the second byte, and the third byte reserved for future use.

MODE	SYNTAX	HEX	LEN
24-bits	CPU Ra	$20	3

The Stack Pointer is R250

The stack pointer is embedded within the CPU, but it is also addressable as the 24-bit R250 in any of the operations that specify a register. The 24-bit R250, the 16-bit R250 and R251, and 8-bit R250, R251, and R253 are all readable and writable by any of the opcodes that use registers.

This allows for addressing modes that are relative to the top of the stack, and it allows for popping large blocks off the stack without requiring a loop or sequence of PLL operations.

Some high-level languages like C use the stack for storage without subroutines, and both the stack-relative address modes

The Program Counter is R253

The program counter is embedded within the CPU, but it is also addressable as R253 in any of the operations that specify a register.

This allows for addressing modes that are relative to the currently running program, which is useful for relocatable code.

Do note that the 24-bit R253, 16-bit R253 and R254, and 8-bit R253, R254, and R255 may be implemented as read-only values by the CPU, with the JMP and JSR opcodes as the only method for writing a new value to the program counter.

appleII4
github.com/lunarmobiscuit/izapple2 & iz6502
LOAD
128K
100
50
0
600-2026-00
apple computer inc.™