

# The Woz Wonderbook

A compendium of notes, diagrams, articles, instructions and code  
that describes the Apple ][ computer and how to program it.

AUTHOR

Steve Wozniak ( [www.woz.org](http://www.woz.org) )

DOCUMENT DATES OF RECORD

September 20, 1977 - November 15, 1977

This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

# The Woz Wonderbook

## Introduction

A compendium of notes, diagrams, articles, instructions and code that describes the Apple ][ computer and how to program it.

### What is the Woz Wonderbook?

The Woz Wonderbook was pulled together from Steve Wozniak's file drawers in the Summer and Fall of 1977 and served as the key reference describing the Apple ][ for Apple's own employees. The Wonderbook served as a primary source for the first real Apple ][ manual, the Red Book, published in January 1978. Apple ][ sales were increasing since its introduction at the West Coast Computer Fair in April 1977 and Woz and a team at Apple used the Wonderbook to bridge the gap in documentation as Apple and Steve Jobs realized they had to create a more professional product and manuals. There was only one Woz Wonderbook in the Apple library. The Woz Wonderbook at the DigiBarn was one of only a few copies made of this master by Apple employees at the time for internal use.

### Facts about the Woz Wonderbook

Author:

Steve Wozniak ([www.woz.org](http://www.woz.org) <<http://www.woz.org/>>)

Document dates of record:

September 20, 1977-November 15, 1977

Owner:

DigiBarn Computer Museum ([www.digibarn.com](http://www.digibarn.com)),  
Curator Bruce Damer (<http://www.damer.com/>).

DigiBarn's pages on the Wonderbook including this version can be found at:

<http://www.digibarn.com/collections/books/woz-wonderbook/>

This Wonderbook was discarded by Apple Computer Inc. (<http://www.apple.com/>) and recovered by Bill Goldberg who later donated it to the DigiBarn Computer Museum.

This Wonderbook was scanned and resurrected in October 2004 into PDF format by David T Craig ([shirlgato@cybermesa.com](mailto:shirlgato@cybermesa.com)).

This digital rendition of the Woz Wonderbook is available for non-commercial, educational and research purposes with the requirement to provide attribution and share-alike under the Creative Commons license provided on page 5. All other uses require the agreement of the DigiBarn Computer Museum (contact through [www.digibarn.com](http://www.digibarn.com)).

This page is not part of the original Wonderbook

# The Woz Wonderbook

## Property Statement

This Woz Wonderbook is the property of the DigiBarn Computer Museum which is offering it under the following Creative Commons License found on page 5.

Under the terms of this license you must credit the DigiBarn Computer Museum, Steve Wozniak and Apple Computer, Inc. if whole or part of this Wonderbook is used for non commercial / educational or research purposes. All other uses require the agreement of the DigiBarn Computer Museum (contact through [www.digibarn.com](http://www.digibarn.com)).

We would like to acknowledge Bill Goldberg for providing us this copy of the Woz Wonderbook.

The author of the Woz Wonderbook is Steve Wozniak.

This page is not part of the original Wonderbook



C O M M O N S D E E D

Creative Commons

Attribution - Non Commercial - Share Alike 1.0

You are free:

- \* to copy, distribute, display, and perform the work
- \* to make derivative works

Under the following conditions:



Attribution. You must give the original author credit.



Noncommercial. You may not use this work for commercial purposes.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- \* For any reuse or distribution, you must make clear to others the license terms of this work.
- \* Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code.

This page is not part of the original Wonderbook

Disclaimer

The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) – it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license.

Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

For the full legal code for this license see:

<http://creativecommons.org/licenses/by-nc-sa/2.0/>

This page is not part of the original Wonderbook

# The Woz Wonderbook

## Table of Contents

Auto Repeat for Apple-II Monitor Commands  
20 September 1977

Use of the Apple-II Mini-Assembler

Apple-II Pointers and Mailboxes

Apple-II 2716 EPROM Adaptation ('D0' and 'D8' Sockets)  
18 November 1977

Using Apple-II Color Graphics

Adding Colors to Apple-II Hi-Res

Apple-II Disassembler Article (Apple-II MONITOR ROM)

Apple-II Cassette Article

Apple-II Floating Point Package

Apple-II Sweet-16 -- The 6502 Dream Machine

Apple-II 6502 Code Relocation Program  
14 November 1977

Apple-II Renumbering and Appending BASIC Programs  
15 November 1977

---

References

03 November 2004

Bill Goldberg Interview

19 April 2004

Credits

This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

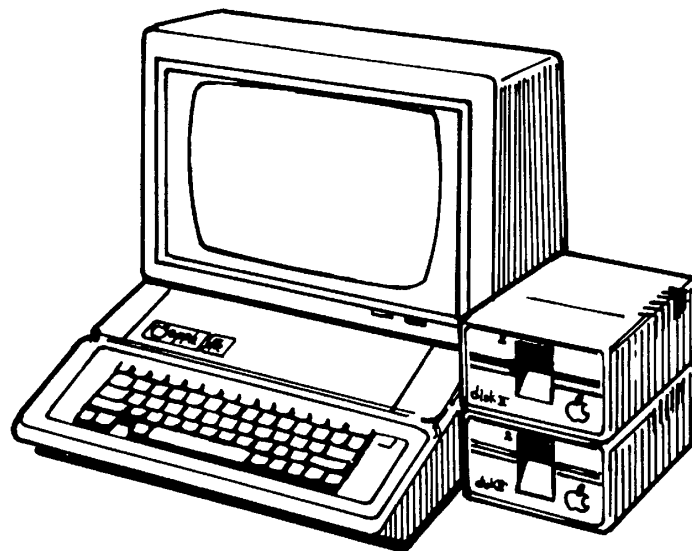


# The Woz Wonderbook

DOCUMENT

## Auto Repeat for Apple-II Monitor Commands

20 September 1977



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

9/20/77  
Woz

## AUTO REPEAT FOR APPLE -II MONITOR COMMANDS

It is occasionally desirable to automatically repeat a MONITOR command or command sequence on the APPLE II computer. For example, flaky (intermittently bad) RAM bits in the \$800 - \$FFF address range (\$ stands for hex) may be detected by verifying those locations with themselves using the MONITOR verify command:

```
*800<800.FFFV) ( ) is car ret)
```

Because this problem is intermittent, multiple verifications may be necessary before the problem is detected. Typing the verify command over and over is a tedious chore which may not even catch the bug, particularly since the RAMS are not fully exercised while the user is typing.

The APPLE - II MONITOR command input buffer begins at location \$200 and is scanned from beginning to end after the user finishes the line by typing a carriage return. An index to the next executable character of the buffer resides in location \$34 while any function is being executed. By adding the command '34:0' to the end of a MONITOR command sequence the user causes scanning to resume at the beginning. Because the '34:0' command leaves the MONITOR in 'store' mode, an 'N' command should begin the line. The following is an example of a command sequence which verifies locations \$800 - \$FFF with themselves, automatically repeating.

```
*N800<800.FFFV 34:0 ) ( ) is blank)
(Note that the trailing blank is necessary for this feature to
work properly)
```

Multiple command sequences accepted by the Apple II MONITOR may also be automatically repeated. For example, the following command sequence clears all bits in the address range \$400 - \$5FF, verifies these locations with themselves, sets them all to ones, verifies them again, and repeats:

```
*N400:0 ) N401<400.5FEM 400<400.5FFV 400:FF ) N401<400.5FEM
400<400.5FFV 34:0 )
) is necessary blank
) is car return
```

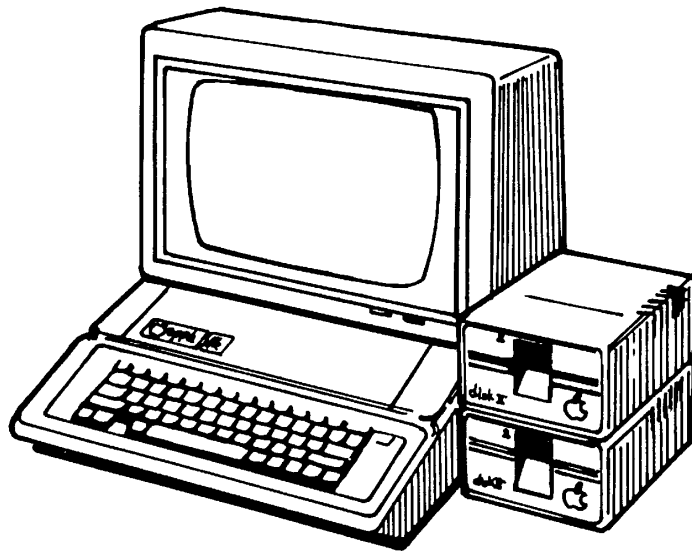
Because this example uses screen memory locations, it is observable on the display. The repeating command may be halted by hitting RESET. Since the cursor is only generated for keyboard entry, it will disappear while the example repeats.

This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

## Use of the Apple-II Mini-Assembler



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

The following section covers use of the Apple II mini-assembler only. It is not a course in assembly language programming. For a reference on programming the 6502 microprocessor, refer to the MOS Technology Programming manual. The following section assumes the user has a working knowledge of 6502 programming and mnemonics.

The Apple II mini-assembler is a programming aid aimed at reducing the amount of time required to convert a hand-written program to object code. The mini-assembler is basically a look-up table for opcodes. With it, you can type mnemonics with their absolute addresses, and the assembler will convert it to the correct object code and store it in memory.

Typing "F666G" will put the user in mini-assembler mode. While in this mode, any line typed in will be interpreted as an assembly language instruction, assembled, and stored in binary form unless the first character on the command line is a "\$".

If it is, the remainder of the line will be interpreted as a normal monitor command, executed, and control returned to assembler mode. To get out of the assembler mode, reset must be pushed.

If the first character on the line is blank, the assembled instruction will be stored starting at the address immediately following the previously assembled instruction. If the first character is nonblank (and not "\$"), the line is assumed to contain an assembly language instruction preceded by the instruction address (a hex number followed by a ":"). In either case, the instruction will be retyped over the line just entered in disassembler format to provide a visual check of what has been assembled. The counter that

keeps track of where the next instruction will be stored is the pseudo PC (Program Counter) and it can be changed by many monitor commands (eg. 'L', 'T', ...). Therefore, it is advisable to use the explicit instruction address mode after every monitor command and, of course, when the Tiny assembler is first entered.

Errors (unrecognized mnemonic, illegal format, etc.) are signalled by a "beep" and a carrot ("^") will be printed beneath the last character read from the input line by the mini-assembler.

The mnemonics and formats accepted by the mini assembler are the same as those listed by the 6502 Programmers Manual, with the following exceptions and differences:

1. All imbedded blanks are ignored, except inside addresses,
2. All addresses typed in are assumed to be in hex (rather than decimal or symbolic). A preceding "\$" (indicating hex rather than decimal or symbolic) is therefore optional, except that it should not precede the instruction address).
3. Instructions that operate on the accumulator have a blank operand field instead of "A".
4. When entering a branch instruction, following the branch mnemonic should be the target of the branch. If the destination address is not known at the time the instruction is entered, simply enter an address that is in the neighborhood, and later re-enter the branch instruction with the correct target address.  
NOTE: If a branch target is specified that is out of range, the mini-assembler will flag the address as being in error.



5. The operand field of an instruction can only be followed by a comment field, which starts with a semi-colon (";"). Obviously, the Tiny assembler ignores the field and in fact will type over it when the line is typed over in disassembler format. This "feature" is included only to be compatible with future upgrades including input sources other than the keyboard.
6. Any page zero references will generate page zero instruction formats if such a mode exists. There is no way to force a page zero address to be two bytes, even if the address has leading zeroes.

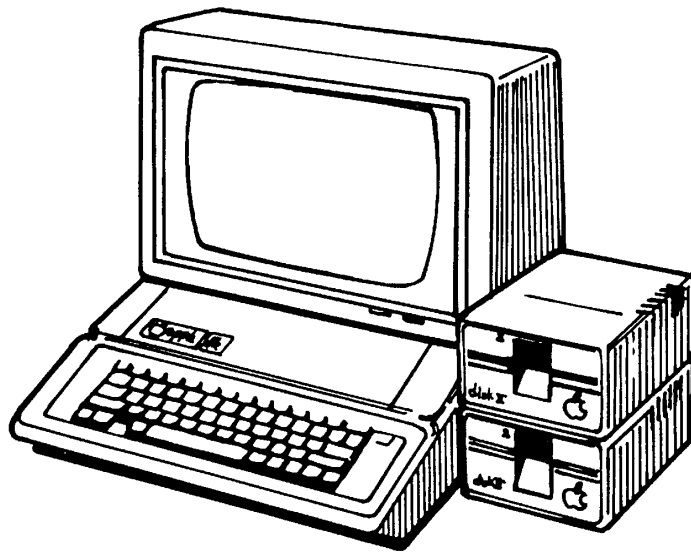
In general, to specify an addressing type, simply enter it as it would be listed in the disassembly. For information on the disassembler, see the monitor section.

This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

## Apple-II Pointers and Mailboxes



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

## POINTERS & MAILBOXES

<u>NAME</u>	<u>HEX</u>	<u>DEC</u>	<u>DESCRIPTION</u>
UNIT, L	700		Screen buffer start point. 7001 character when appear on screen.
IN	2000 - 2FF	512 - 767	Line buffer
6C, 6D	->		Top of Applesoft BASIC program
70, 71			Top of Variable Area in Applesoft

This page is not part of the original Wonderbook

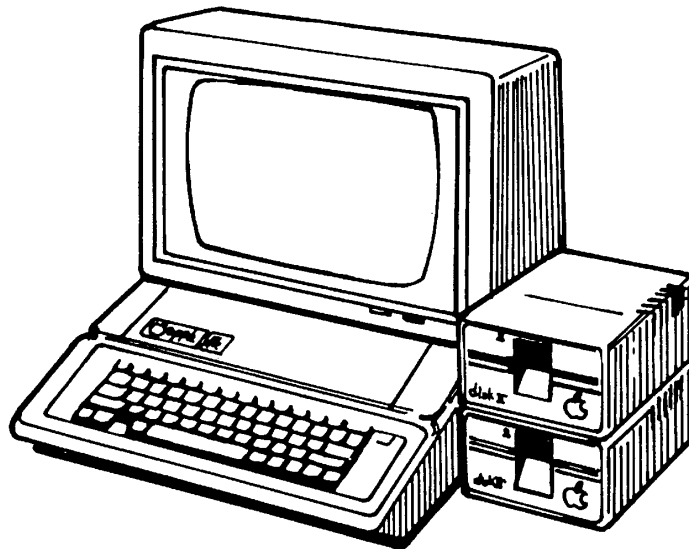
# The Woz Wonderbook

DOCUMENT

Apple-II 2716 EPROM Adaptation

( 'D0' and 'D8' Sockets )

18 November 1977



This page is not part of the original Wonderbook

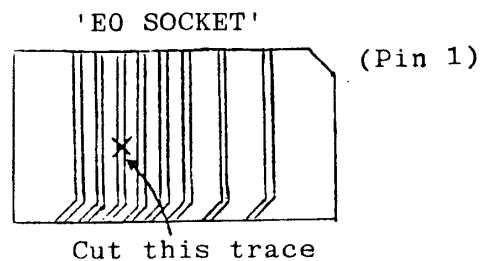
This page is not part of the original Wonderbook



11-18-77  
WOZ

APPLE-II 2716 EROM ADAPTATION  
( 'D0' and 'D8' sockets )

1. Remove the 'EO' ROM from its socket. On the top side of the board under the 'EO' socket, cut the ROM pin 18 jumper trace. Then reinsert the ROM. This cut will isolate pins 18 of ROMs 'D0' and 'D8' from pins 18 of the other ROMs. Reinsert the 'EO' ROM when done.



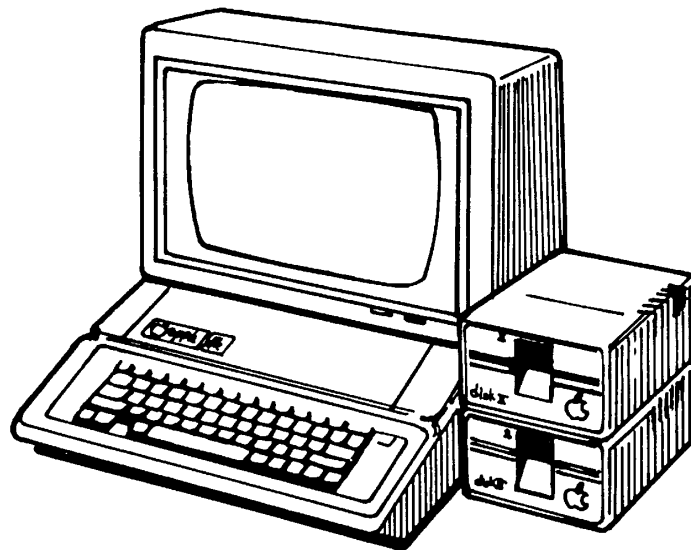
2. On the underside of the APPLE-II board, cut the traces connecting pin 20 to 21 of ROMs 'D0' and 'D8' only.
3. On the underside, cut the trace going to pin 18 of ROM 'D8' near the chip. Scrape solder resist off of approximately  $\frac{1}{4}$  inch of the remaining trace not still connected to pin 18. You may wish to tin it with solder since it will later be soldered to.
4. (Underside) Connect pin 18 of ROM 'D8' to pin 12 of ROM 'EO' (ground)
5. (underside) Connect pin 18 of ROM 'EO' to the trace which previously went to pin 18 of ROM 'D8' (and which should be pretinned if step 3 was followed).

6. (underside) Connect pin 21 of ROM 'D8' to pin 21 of ROM 'D0'.  
Then connect both of these to pin 24 of either ROM (VCC).
  
7. Note that the  $\overline{\text{INH}}$  control function (pin 32 on the APPLE-II I/O BUS connectors) will not disable the 2716 EROMs in the 'D0' and 'D8' ROM slots since pin 21 is a power supply pin and not a chip select input on the EROMs.

# The Woz Wonderbook

DOCUMENT

## Using Apple-II Color Graphics



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

### USING APPLE-II COLOR GRAPHICS

The APPLE-II color graphics hardware will display a 40H by 48V grid, each position, of which may be any one of 16 colors. The actual screen data is stored in 1K bytes of system memory, normally locations \$400 to \$7FF. (A dual page mode allows the user to alternatively display locations \$800 to \$BFF). Color displays are generated by executing programs which modify the 'screen memory'. For example, storing zeroes throughout locations \$400 to \$7FF will yield an all-black display while storing \$33 bytes throughout will yield an all-violet display. A number of subroutines are provided in ROM to facilitate useful operations.

The x-coordinates range from 0 (leftmost) to 39 (rightmost) and the y-coordinates from 0 (topmost) to 47 (bottommost). If the user is in the mixed graphics/text mode with 4 lines of text at the bottom of the screen, then the greatest allowable y-coordinate is 39.

The screen memory is arranged such that each displayed horizontal line occupies 40 consecutive locations. Additionally, even/odd line pairs share the same byte groups. For example, both lines 0 and 1 will have their leftmost point stored in the same byte, at location \$400; and their rightmost point stored in the byte at location \$427. The least significant 4 bits correspond to the even line and the most significant 4 bits to the odd line. The relationship between y-coordinates and memory addresses is illustrated on the following page.

COLOR GRAPHICS SCREEN MEMORY MAP

Y-coordinate

0 0 a b c d e f

BASE (leftmost) address

0 0 0 0 0 1 c d

GBASH

e a b a b 0 0 0

GBASL

Data byte

X X X X Y Y Y Y

odd      even  
line    line  
data    data

<u>LINE</u>	<u>BASE address(hex)</u>	<u>Secondary BASE address</u>
\$0,1	\$400	\$800
\$2,3	\$480	\$880
\$4,5	\$500	\$900
\$6,7	\$580	\$980
\$8,9	\$600	\$A00
\$A,B	\$680	\$A80
\$C,D	\$700	\$B00
\$E,F	\$780	\$B80
\$10,11	\$428	\$828
\$12,13	\$4A8	\$8A8
\$14,15	\$528	\$928
\$16,17	\$5A8	\$9A8
\$18,19	\$628	\$A28
\$1A,1B	\$6A8	\$AA8
\$1C,1D	\$728	\$B28
\$1E,1F	\$7A8	\$BA8
\$20,21	\$450	\$850
\$22,23	\$4D0	\$8D0
\$24,25	\$550	\$950
\$26,27	\$5D0	\$9D0
\$28,29	\$650	\$A50
\$2A,2B	\$6D0	\$AD0
\$2C,2D	\$750	\$B50
\$2E,2F	\$7D0	\$BD0

The APPLE-II color graphics subroutines provided in ROM use a few page zero locations for variables and workspace. You should avoid using these locations for your own program variables. It is a good rule not to use page zero locations \$20 to \$4F for any programs since they are used by the monitor and you may wish to use the monitor (for example, to debug a program) without clobbering your own variables. If you write a program in assembly language that you wish to call from BASIC with a CALL command, then avoid using page zero locations \$20 to \$FF for your variables.

Color Graphics  
Page Zero Variable Allocation

GBASL	\$26
GBASH	\$27
H2	\$2C
V2	\$2D
MASK	\$2E
COLOR	\$30

GBASL and GBASH are used by the color graphics subroutines as a pointer to the first (leftmost) byte of the current plot line. The (GBASL),Y addressing mode of the 6502 is used to access any byte of that line. COLOR is a mask byte specifying the color for even lines in the 4 least significant bits (0 to 15) and for odd lines in the 4 most significant bits. These will generally be the same, and always so if the user sets the COLOR byte via the SETCOLOR subroutine provided. Of the above variables only H2, V2, and MASK can be clobbered by the monitor.

Writing a color graphics program in 6502 assembly language generally involves the following procedures. You should be familiar with subroutine usage on the 6502.

1. Set the video mode and scrolling window (refer to the section on APPLE-II text features)
2. Clear the screen with a call to the CLRSCR (48-line clear) or CLRTOP (40-line clear) subroutines. If you are using the mixed text/graphics feature then call CLRTOP.
3. Set the color using the SETCOLOR subroutine.
4. Call the PLOT, HLINE, and VLINE subroutines to plot points and draw lines. The color setting is not affected by these subroutines.
5. Advanced programmers may wish to study the provided subroutines and addressing schemes. When you supply x- and y-coordinate data to these subroutines they generate BASE address, horizontal index, and even/odd mask information. You can write more efficient programs if you supply this information directly.



SETCOL subroutine (address \$F864)

Purpose: To specify one of 16 colors for standard resolution plotting.

Entry: The least significant 4 A-Reg bits contain a color code (0 to \$F). The 4 most significant bits are ignored.

Exit: The variable COLOR (location \$30) and the A-Reg will both contain the selected color in both half bytes, for example color 3 will result in \$33. The carry is cleared.

Example: (select color 6)

```
LDA #$6
JSR SETCOL ($F864)
```

note: When setting the color to a constant the following sequence is preferable.

```
LDA #$66
STA COLOR ($30)
```

PLOT subroutine (address \$F800)

Purpose: To plot a square in standard resolution mode using the most recently specified color (see SETCOL). Plotting always occurs in the primary standard resolution page (memory locations \$400 to \$7FF).

Entry: The x-coordinate (0 to 39) is in the Y-Reg and the y-coordinate (0 to 47) is in the A-Reg.

Exit: The A-Reg is clobbered but the Y-Reg is not. The carry is cleared. A halfbyte mask (\$F or \$F0) is generated and saved in the variable location MASK (location \$2E).

Calls: GBASCALC

Example: (Plot a square at coordinate (\$A,\$2C))

```
LDA #$2C      Y-coordinate
LDY #$A       X-coordinate
JSR PLOT (F800)
```

PLOT1 subroutine (address \$F80E)

Purpose: To plot squares in standard resolution mode with no Y-coordinate change from last call to PLOT. Faster than PLOT. Uses most recently specified COLOR (see SETCOL)

Entry: X-coordinate in Y-Reg (0 to 39)

Exit: A-Reg clobbered. Y-Reg and carry unchanged.

Example: (Plotting two squares - one at (3,7) and one at (9,7))

```
LDY, #$3      X-coordinate
LDA #$7       Y-coordinate
JSR PLOT      Plot (3,7)
LDY #$9       New X-coordinate
JSR PLOT1     Call PLOT1 for fast plot.
```

HLINE subroutine (address \$F819)

Purpose: To draw horizontal lines in standard resolution mode. Most recently specified COLOR (see SETCOL) is used.

Entry: The Y-coordinate (0 to 47) is in the A-Reg. The leftmost X-coordinate (0 to 39) is in the Y-Reg and the rightmost X-coordinate (0 to 39) is in the variable H2 (location \$2C). The rightmost x-coordinate may never be smaller than the leftmost.

Calls: PLOT, PLOT1

Exit: The Y-Reg will contain the rightmost X-coordinate (same as H2 which is unchanged). The A-Reg is clobbered. The carry is set.

Example: Drawing a horizontal line from 3(left X-coord) to \$1A (right X-coord) at 9 (Y-coord)

```
LDY #$3      Left
LDA #$1A     Right
STA H2       Save it
LDA #$9      Y-coordinate
JSR HLINE    Plot line
```

SCRN subroutine (address \$F871)

Purpose: To sense the color (0 to \$F) at a specified screen position.

Entry: The Y-coordinate is in the A-Reg and the X-coordinate is in the Y-Reg.

Exit: The A-Reg contains contents of screen memory at specified position. This will be a value from 0 to 15). The Y-Reg is unchanged and the 'N' flag is cleared (for unconditional branches upon return).

Calls: GBASCALC<sup>1</sup>

Example: To sense the color at position (5,7)

```
LDY #$5      X-coordinate
LDA #$7      Y-coordinate
JSR SCRN     Color to A-Reg.
```

GBASCALC subroutine (address \$F847)

Purpose: To calculate a base address within the primary standard resolution screen memory page corresponding to a specified Y-coordinate. Once this base address is formed in GBASL and GBASH (locations \$26 and \$27) the PLOT routines can access the memory location corresponding to any screen position by means of (GBASL),Y addressing.

Entry: (Y-coordinate)/2 (0 to \$17) is in the A-Reg. Note that even/odd Y-coordinate pairs share the same base address)

Exit: The A-Reg is clobbered and the carry is cleared. GBASL and GBASH contain the address of the byte corresponding to the leftmost screen position of the specified Y-coord.

Example: To access the byte whose Y-coordinate is \$1A and whose X-coordinate is 7.

```
LDA #$1A     Y-coordinate
LSR          Divide by 2
JSR GBASCALC Form base address.
LDY #$7      X-coordinate
LDA (GBASL),Y Access byte
```

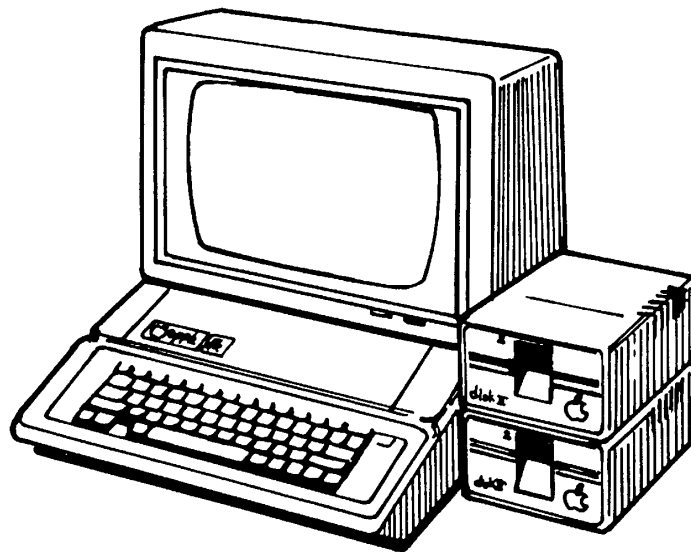
Note: For an even/odd Y-coord pair, the even-coord data is contained in the least significant 4 bits of the accessed byte and the odd-coord data in the most significant 4.

This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

## Adding Colors to Apple-II Hi-Res



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

ADDING COLORS TO APPLE-II HI-RES  
(nullifies warrantee)

1. Remove the APPLE-II PC board from its enclosure

- (a) Remove the ten (10) screws securing the plastic top piece to the metal bottom plate. Six (6) of these are flat-head screws around the perimeter of the bottom plate and four (4) are round-head screws located at the front lip of the computer. All are removed with a phillips head screwdriver. Do not remove the screws securing the power supply or nylon posts.
- (b) Lift the plastic top piece from the bottom plate while taking care not to damage the ribbon cable connecting the keyboard to the PC board. This cable will have to be disconnected from one or the other.
- (c) Disconnect the power supply from the PC board.
- (d) Remove the #8 nut and lockwasher securing the center of the PC board. These will not be found on the earlier APPLE-II computers.
- (e) Carefully disengage each of 6 nylon posts from the PC board. (7 on earlier versions).
- (f) Lift the PC board from the bottom plate.

2. Above the board wiring method

(a) Lift the following IC pins from their sockets.

A8-1  
A8-6  
A8-13  
A9-1  
A9-2  
A9-9

(b) Mount a 74LS74 (dual C-D flip-flop) and a 74LS02 (quad NOR gate) in the APPLE-II breadboard area (A11 to A14 region).

(c) Wire the following circuit (\* indicates that wiring is to a pin which is out of its socket).



3. Remove the 74LS125 and the standard APPLE II  
 (A11-114) sockets.

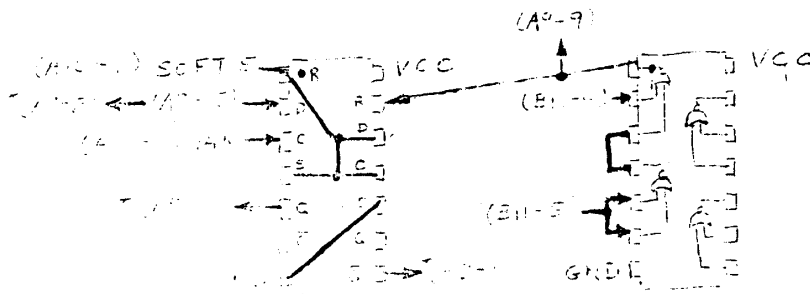
4. Cut the traces on the board wiring

5. Lift the following pins from their sockets.

- AP-1
- AP-6
- AP-13
- AP-1
- AP-6
- AP-8

6. Mount a 74LS174 (Dual C-D Flip Flop) and a 74LS02  
 (Dual NOR gate) in the breadboard area (A11-114)

7. Wire the following circuit (\* indicates that wire goes  
 to a pin which is out of its socket)

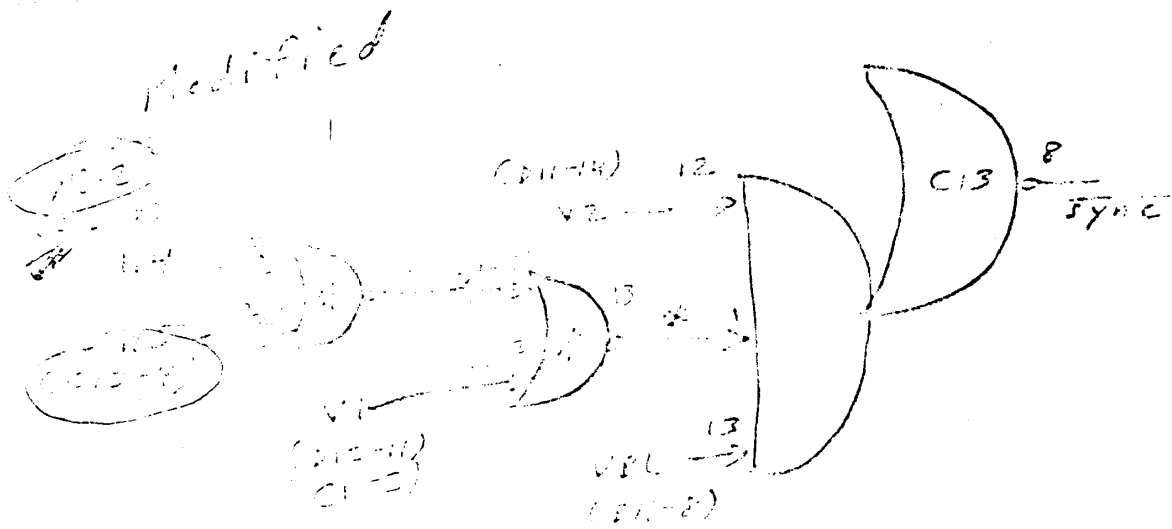
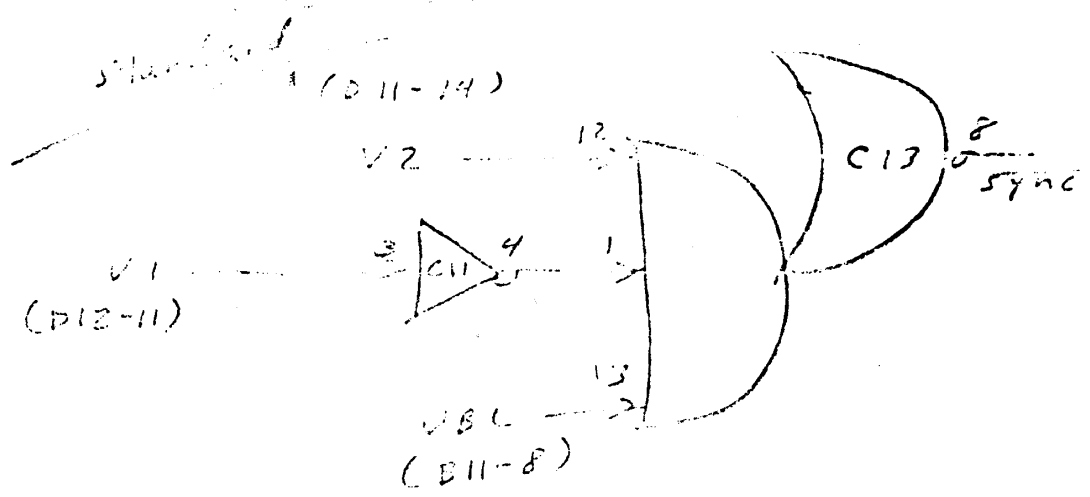


(A11-114) → (AP-6)

(A11-114) → (AP-13)

8-30-77 Woz

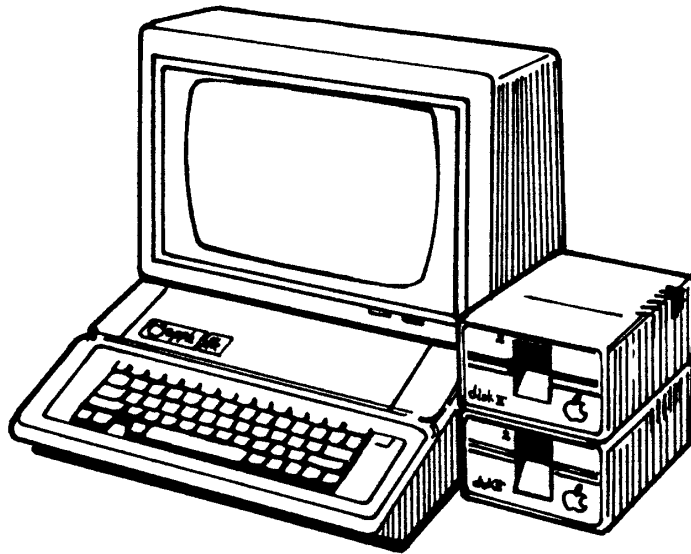
A 2 to 1 input Meqnc. variations  
 in VSYND on standard (60 Hz)  
 APPLE-IT



# The Woz Wonderbook

DOCUMENT

Apple-II  
Disassembler Article  
( Apple-II MONITOR ROM )



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

D I S A S S E M B L E R   A R T I C L E

(pertains to APPLE-II MONITOR ROM)

## APPLE DISASSEMBLER

Allen Gamm Steve Wozniak

1. Description. This subroutine package is used to display single or sequential 68000 instructions in mnemonic form. The subroutines are tailored to disassemble and debugging aids but tables with more general usage (assemblers) are included. The subroutines occupy one page (256 bytes) and tables most of another. Seven page zero locations are used.

2. Features. Four output fields are generated for each disassembled instruction: (1) Address of instruction, in hexadecimal (hex); (2) hex code listing of instruction, 1 to 4 bytes; (3) 3 character mnemonic, or "???" for invalid ops (which assume a length of 1 byte); and (4) Address fields, in one of the following formats.

<u>Format</u>	<u>Address Mode</u>
(empty)	Invalid, Implied, Accumulator
#i2	Page zero.
#i234	Absolute, Branch (target primed)
#i12	Immediate
i12, X	Zero page, indexed by X
i12, Y	Zero page, indexed by Y
i234, X	Absolute, indexed by X
i234, Y	Absolute, indexed by Y
(i1234)	Indirect
(i12, X)	Indexed Indirect
(i12), Y	Indirect Indexed

Note that unlike MOS TECHNOLOGY assemblers, which use "A" for accumulator addressing, the APPLE disassembler outputs an empty field to avoid confusion and facilitate byte counting.

3. Usage. The following subroutine entries are useful.

- (a) **DSMBL:** Disassembles and displays 20 sequential instructions beginning at the address specified by the page zero variables PCL and PCH. For example, if called with 1D2 in PCL and 158 in PCH, 20 instructions beginning at address 158D2 will be disassembled. PCL and PCH are updated to contain the address of the last disassembled instruction. Must be called with 6502 in hexadecimal mode (0's shift bit clear). All processor registers are altered (except S - stack pointer). Uses INSTDSP and PCAD5.
- (b) **INSTDSP:** Disassembles and displays a single instruction whose address is specified by PCL and PCH. Must be called in hexadecimal mode. All processor registers (except S) are altered. Uses PCAD5, PRPC, PRBLNK, PRBLZ, PRNTAX, PRBYTE, and CHAROUT.
- (c) **PRPC:** Outputs a carriage return, 4 hex digits corresponding to PCH and PCL, a dash, and 3 blanks. Alters A, clears X. Uses PRNTAX and CHAROUT.
- (d) **PRNTX:** Outputs the contents of X as two hex digits. Alters A. Uses CHAROUT.
- (e) **PRNTAX:** Outputs two hex digits for the contents of A, then two hex digits for the contents of X. A is altered. Uses CHAROUT.
- (f) **PRNTYX:** Same as PRNTAX except that Y and X are output. Alters A. Uses CHAROUT.
- (g) **PRBLNK:** Outputs 3 blanks. Alters A, clears X. Uses CHAROUT.
- (h) **PRBLZ:** Outputs the number of blanks specified by the contents of X (0 for no blanks). Alters A, clears X. Uses CHAROUT.
- (i) **PRBL3:** Outputs a character from the A register followed by X-1 blanks (0 for other words, X specifies the total number of characters output, 0's for no blanks). Alters A, clears X. Uses CHAROUT.

4. Running as a program.

The following program will run a disassembly.

```
9F0 20 0 8 JSR DSMBL
9F3 4C 1F FF JMP MONITOR
```

Supplied on APPLE-1 cassette tapes.

First, put the starting address of code you want disassembled in PCL (low order byte) and PCH. (high order byte). Then type 9F0 R (CR) (on APPLE-1 system). 20 instructions will be disassembled. Hitting R (CR) again will give the next 20, etc.

Cassette tapes supplied for the ACI-1 (APPLE Cassette Interface) are intended to be loaded from 8500 to 89FF.

5. Non-APPLE systems.

Source and object code supplied occupies pages 8 and 9. All code is on page 8, tables on page 9. (these tables may be relocated at will; MODE, MODE2, CHAR1, CHAR2, MNEML, and MNEMR. The code may also be relocated. Be careful if you use pages 0 or 1. Page 1 is the subroutine return stack and page 0 must contain 7 variables (to use DSMBL). These may be relocated on page 0 but PCL must always immediately precede PCH for (rpage), Y addressing.

	{ 840	FORMAT }	
locations	{ 841	LENGTH }	Used by INSDSP, DSMBL
used	{ 842	LMNEM }	
by	{ 843	RMNEM }	
supplied	{ 844	PCL }	used by READ, INSDSP, DSMBL
code.	{ 845	PCH }	
	{ 846	COUNT }	used by DSMBL only.



(j) PCADJ:  $(PCL, PCH) + 1 + (\text{contents of page zero variable LENGTH}) \rightarrow Y \ \& \ A$   
(low order byte in Y). For example, if  $PCL = \$D2$ ,  $PCH = \$38$ ,  
and  $LENGTH = 1$  (corresponding to a 2 byte instruction), PCADJ will  
leave  $Y = \$D4$  and  $A = \$38$ . X is always loaded with PCH.

(k) PCADJ2: Same as PCADJ except that A is used in place of LENGTH.

(l) PCADJ3: Same as PCADJ2 except that the increment (+1) is specified  
by the carry (set = +1, clear = +0).

5. Modifications.

- (a) To change '#' to '=' for immediate mode change location \$955 (on code enclosed) from a \$A3 to a \$BD
- (b) To skip the '\$' (meaning hex) preceding disassembled values make the following changes.

946: ~~01~~ (was 81)  
 947: ~~02~~ (was 82)  
 94C: 11 (was 91)  
 94D: 12 (was 92)  
 94E: ~~06~~ (was 56)  
 950: ~~05~~ (was 55)  
 951: 1D (was 9D)  
 95B: ~~00~~ (was A4)  
 95C: ~~00~~ (was A4)

- (c) To have address field of accumulator-addressed instructions print as 'A'.
  - (1) Must skip \$ preceding disassembled values by making modification
  - (b) above.
  - (2) Change the following locations.

949: ~~80~~ (was 00)  
 957: C1 (was A4)

- (d) To add ROP and addressing modes change the following locations.

901: 9C (was 00) 919: 02 (was 00)  
 902: 26 (was 00) 91A: 71 (was 70)  
 91F: B3 (was B0)  
 91D: 85 (was 00)  
 91E: 09 (was 00)

300	A9	DSMBL	800	LDA	#13	Count for Z0 instruction disassembly.
302	55		STA	COUNT		
304	20	DSMBL2	JSR	INSTDSP		Disassemble + display one instruction.
307	20		JSR	PCADJ		
30A	55		STA	PCL		Update PCL, H to next instruction.
30C	84		STY	PCH		
30E	C6		DEC	COUNT		Done first 19 instructions?
310	D0		BNE	DSMBL2		Yes, loop. Else disassemble Z0th.
312	20	INSTDSP	JSR	PRPC		Print PCL, PCH.
315	A1		LDA	(PCL,X)		Get op code.
317	A8		TAY			
318	4A		LSR	IEVEN		Even/odd test.
319	90		BCC			b, test.
31B	4A		LSR			xxxxxxx11 instruction invalid.
31C	B0		BCS	ERR		10001001 instruction invalid.
31E	C9		CMP	#22		Mask 3 bits for address mode and
320	F0		BEQ	ERR		add indexing offset.
322	29		AND	#7		LSB into carry for left/right test below.
324	9		ORA	#80		
326	4A	IEVEN	LSR			
327	AA		TAX			
328	BD		LDA	MODE,X		
32B	B0		BCS	RTMODE		Index into address mode table.
32D	4A		LSR			If carry set use LSD for print
32E	4A		LSR			format index.
32F	4A		LSR			If carry clear use MSD.
330	4A		LSR			
331	29	RTMODE	AND	#F		Mask for 4-bit index.
333	D0		BNE	GETFMT		\$0 for invalid opcodes.
335	A0		LDY	#80		Substitute \$80 for all invalid opcodes.
337	A9		LDA	#50		Set print format index to 0.
339	AA		TAX			
33A	BD		LDA	MODE2,X		Index into print format table.
33D	55		STA	FORMAT		Save for address field formatting.
33E	29		AND	#13		Mask for 2-bit length (0=1 byte, 1=2 byte, 2=3 byte)

'6

Op code.	Mask if for IXXXIØIØ test.	Save it.	Op code to A again.	Form index into mnemonic table.
841	85	41	841 STA	LENGTH
843	98	8F	TYA	
844	29		AND #8F	
846	AA		TAX	
847	98		TYA	
848	AØ	3	LDY #3	
84A	EØ	8A	CPX #8A	
84C	FØ	B	BEQ MNNDX3	
84E	4A	8	LSR	
84F	9Ø		BCC MNNDX3	
851	4A		LSR	
852	4A		LSR	
853	9	2Ø	ORA #2Ø	
855	88		DEY	
856	DØ	FA	BNE MNNDX2	
858	C8		INY	
859	88		DEY	
85A	DØ	F2	BNE MNNDX1	
85C	48		PHA	
85D	B1	44	LDA (PCL),Y	
85F	2Ø	DC FF	JSR PRBYTE	
862	A2	1	#2 LDX #1	
864	2Ø	E6 8	JSR PRBL2	
867	C4	41	CPY LENGTH	
869	C8		INY	
86A	9Ø	F1	BCC PROP	
86C	A2	3	LDX #3	
86E	CØ	4	CPY #4	
87Ø	9Ø	F2	BCC PROPBL	
872	68		PLA	
873	A8		TAY	
874	B9	5E 9	LDA MNEML,Y	
877	85	42	STA LMNEM	
879	B9	9E 9	LDA MNEMR,Y	
87C	85	43	STA RMNEM	

1. IXXXIØIØ → ØØIØIØXXX
2. XXXYYØIØ → ØØ1111XXX
3. XXXYYØIØ → ØØ11ØØXXX
4. XXXYYØIØ → ØØ1ØØØXXX
5. XXXXXØØØ → ØØØØXXX

Save mnemonic table index.

Print instruction (1 to 3 bytes) in a 12-character field.

Character count for mnemonic print.

Recover mnemonic index.

Fetch 3 character (packed in 2 bytes) mnemonic.

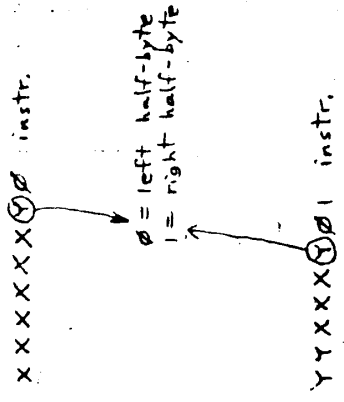
'6

87E	A9	0	PRMNI	LDA	##0		
880	A0	5		LDY	##5		
882	6	43	PRMN2	ASL	RMNEM		Shift 5 bits of character into A. (clears carry)
884	26	42		ROL	LMNEM		
886	2A			ROL			
887	88			DEY			
888	D0	F8		BNE	PRMN2		Add "?" offset.
88A	69	BF		ADC	##BF		Output a character of mnemonic.
88C	20	EF		JSR	CHAROUT		
88F	CA			DEX			
890	D0	EC		BNE	PRMNI		
892	20	E4		JSR	PRBLNK		Output 3 blanks.
895	A2	6		LDX	##6		Count for 6 print format bits.
897	E0	3	PRADR1	CPX	##3		
899	D0	12		BNE	PRADR3		If X=3 then print address val.
89B	A4	41		LDY	LENGTH		No print if LENGTH=0 (1 byte instr.)
89D	F0	E		BEQ	PRADR3		
89F	A5	40	PRADR2	LDA	FORMAT		Handle relative addressing mode special (print target, not displacement).
8A1	C9	E8		SALCMP	##E8		
8A3	B1	44		LDA	(PCL),Y		
8A5	B0	1C		BCS	RELADR		
8A7	20	DC		JSR	PRBYTE		Output 1- or 2-byte address (more significant byte first).
8AA	88			DEY			
8AB	D0	F2		BNE	PRADR2		
8AD	6	40	PRADR3	ASL	FORMAT		Test next print format bit.
8AF	90	E		BCC	PRADR4		If 0, don't print corresponding chars.
8B1	BD	51		SALDA	CHAR1-1,X		
8B4	20	EF		JSR	CHAROUT		Output 1 or 2 chars (if char from CHAR2 is zero, don't output it).
8B7	BD	57		LDA	CHAR2-1,X		
8BA	F0	3		BEQ	PRADR4		
8BC	20	EF		JSR	CHAROUT		
8BF	CA		PRADR4	DEX			
8C0	D0	D5		BNE	PRADR1		Return when done 6 format bits.
8C2	60			RTS			

8C3	20	F2	8	RELADR	JSR	PCADJ3	PCL, PCH + Displacement + 1. to A, Y.
8C6	AA			TAX			
8C7	E8			INX			
8C8	D0	1		BNE	PRNTYX		+1 to X, Y.
8CA	C8			INY			
8CB	98			TYA			
8CC	20	DC	FF	JSR	PRBYTE		Output target address of branch and return.
8CF	8A			TXA			
8D0	4C	DC	FF	JMP	PRBYTE		
8D3	A9	8D		LDA	#\$8D		
8D5	20	EF	FF	JSR	CHAROUT		Output carriage return.
8D8	A5	45		LDA	PCH		
8DA	A6	44		LDX	PCL		
8DC	20	CC	8	JSR	PRNTAX		Output PCH & PCL.
8DF	A9	AD		LDA	#\$AD		
8E1	20	EF	FF	JSR	CHAROUT		Output "-" Blank count.
8E4	A2	3		LDX	#\$3		
8E6	A9	A0		LDA	#\$A0		
8E8	20	EF	FF	JSR	CHAROUT		Output a blank.
8EB	CA			DEX			
8EC	D0	F8		BNE	PRBL2		Loop until count = 0.
8EE	60			RTS			
8EF	A5	41		LDA	LENGTH		0 = 1 byte, 1 = 2 byte, 2 = 3 byte.
8F1	38			SEC			
8F2	A4	45		LDY	PCH		
8F4	AA			TAX			Test displ. sign. (for rel. branch).
8F5	10	1		BPL	PCADJ4		Extend neg- by decrementing PCH.
8F7	88			DEY			
8F8	65	44		ADC	PCL		
8FA	90	1		BCC	RTSI		PCL + LENGTH (or displ.) + 1 to A. Carry into Y (PCH).
8FC	C8			INY			
8FD	60			RTS			

381 40 SHEETS 3 SQUARE  
 43 500 170 SHEETS 3 SQUARE  
 27,000 210 SHEETS 3 SQUARE

Address	Hex	Mode	DFB	Op	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	OpA	OpB	OpC	OpD	OpE	OpF	OpG	OpH	OpI	OpJ	OpK	OpL	OpM	OpN	OpO	OpP	OpQ	OpR	OpS	OpT	OpU	OpV	OpW	OpX	OpY	OpZ	
900	40		DFB																																				
904	D0		DFB																																				
908	30		DFB																																				
90C	D0		DFB																																				
910	40		DFB																																				
914	D0		DFB																																				
918	40		DFB																																				
91C	D0		DFB																																				
920	00		DFB																																				
924	D0		DFB																																				
928	11		DFB																																				
92C	D0		DFB																																				
930	10		DFB																																				
934	D0		DFB																																				
938	10		DFB																																				
93C	D0		DFB																																				
940	62		DFB																																				
944	00		DFB																																				
948	21		DFB																																				
94C	81		DFB																																				
950	82		DFB																																				
954	00		DFB																																				
958	00		DFB																																				
95A	59		DFB																																				
95B	4D		DFB																																				
95C	91		DFB																																				
95D	92		DFB																																				
95E	86		DFB																																				
95F	4A		DFB																																				
960	85		DFB																																				
961	9D		DFB																																				



- ERR
- IMM
- Z-PG
- ABS
- IMPL
- ACC
- (Z-PG, X)
- (Z-PG), Y
- Z-PG, X
- ABS, X
- ABS, Y
- (ABS)
- Z-PG, Y
- REL



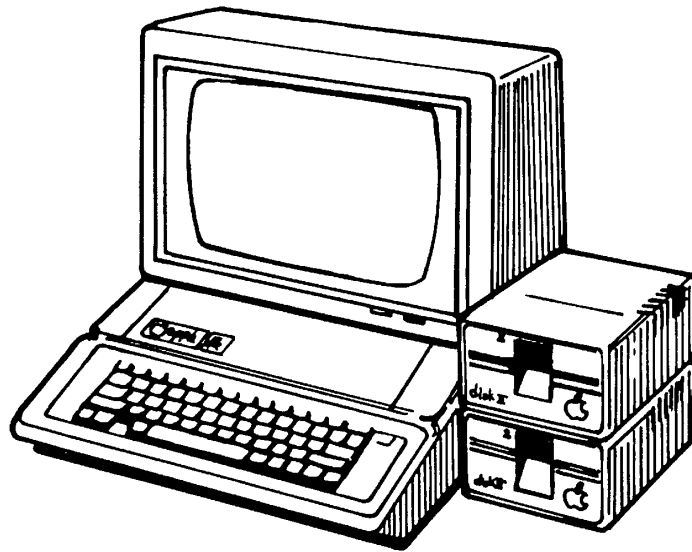


# The Woz Wonderbook

DOCUMENT

Apple-II

Cassette Article



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

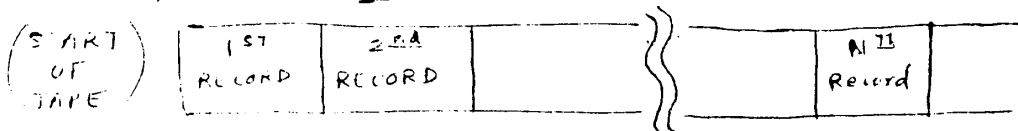
C A S S E T T E   A R T I C L E

(Intro)

The standard audio cassette recorder is rapidly becoming the most popular mass storage peripheral in video-based hobby systems. Many vendors supply their program libraries in cassette form at modest cost. Herein is presented a hardware/software package developed for APPLE-1 systems but easily modified to work on other 6502 and 6800 systems. It is simple, versatile, fast, and inexpensive.

FILES

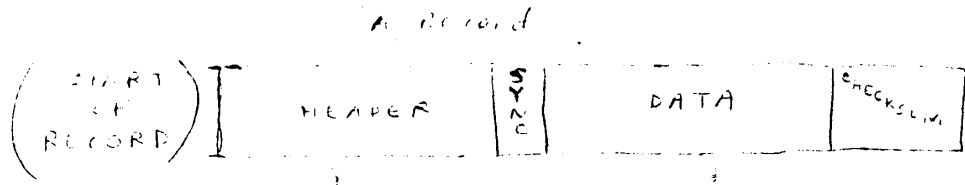
A file is generally a complete program with associated data. Although any number may be recorded on a single tape, one is ~~suggested~~ <sup>recommended</sup> to facilitate locating it. Obviously it should begin at the very beginning of the cassette!

A FILE

Each record within a file contains one contiguous block of data. Thus if a program begins at address  $E000$  (hex) and its data is located ~~at~~ beginning at address  $0100$  (hex) then a 2 record file may be used. <sup>to store it</sup> Either record may appear first on the tape.

RECORDS

Each record of a file is independent of all others. Each may be read from a 'cold start' of the recorder, and the recorder may <sup>be</sup> stopped in-between any pair of records. A header precedes data on the record to insure the recorder reaches speed. A sync bit precedes the data and indicates its start. A checksum byte is recorded after all data bytes for ~~error~~ error detection.



HEADER

The header consists of a .5 second to 20 second square wave, to allow the recorder to reach speed and the 'head circuits' to lock on. The READ RECORD algorithm is such that the header beginning may contain 'junk'.

First Record Header: Approx 10 seconds

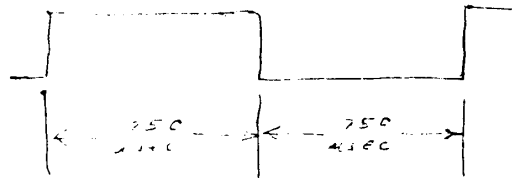
to bypass tape leader.

Other Record Headers: .5 to 20 seconds,

depending on user needs such as whether the recorder will be stopped prior to the record.

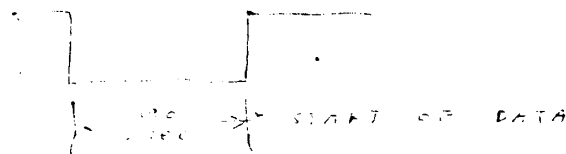
Header Bit (long 1)

(If not, .5 sec or more)



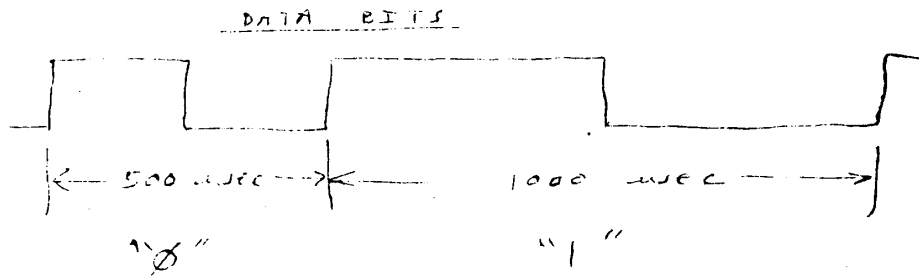
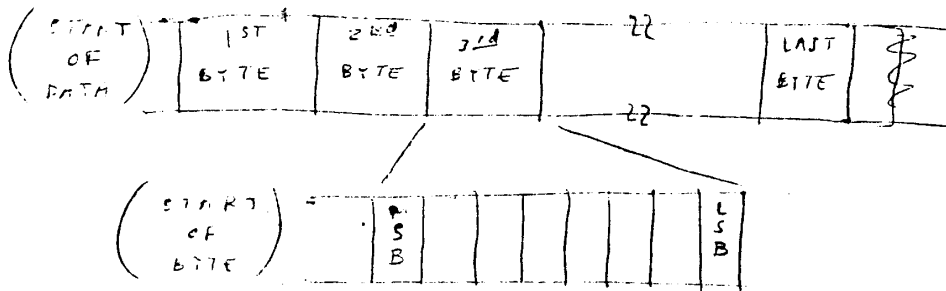
SYNC

To announce 'start of data' a half-cycle of 'short 0' is the sync bit.



DATA

the first byte recorded is ~~typically~~ <sup>usually</sup> from the lowest address. The last one is from the highest address. Each byte is recorded most-significant-bit first, least-significant-bit last. The average transfer rate is 180 bytes per second.



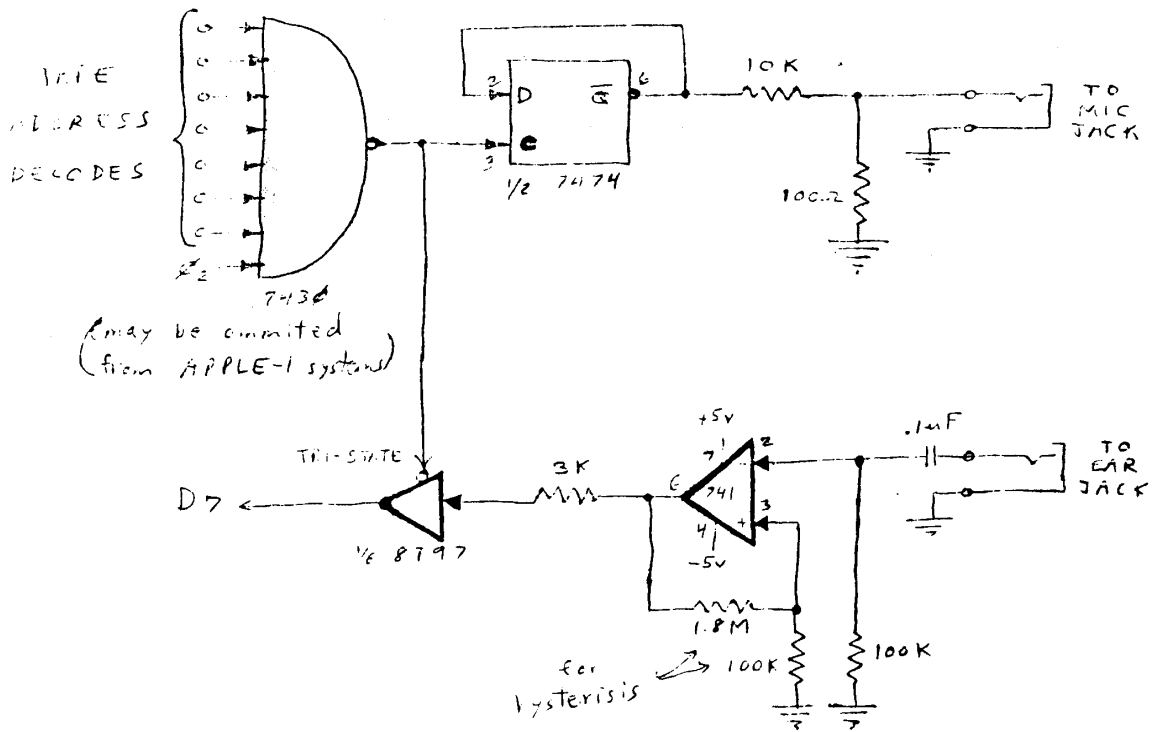
CHECKSUM

The checksum byte immediately follows the last data byte and is recorded in the same 0-1 format. It is the inverse of the logical exclusive-or of all data bytes of the record.

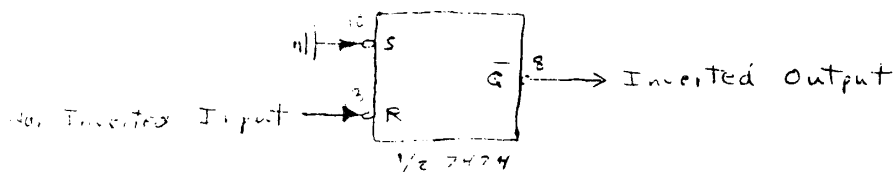
EXAMPLE

DATA BYTE 1 =	10011101
DATA BYTE 2 =	00111011
EX-OR =	10100110
CHECKSUM =	01011001

HARDWARE



- Notes:
- (1) An existing input port may be used in place of the 8T97.
  - (2) Any decoded address strobe (glitch-free) may be used in place of the 7438
  - (3) If an inverter is desired for address decoding, the unused half of the 7474 may be used.





SOFTWARE

Listings are included for subroutines which read and write records and bits. Because all timing is performed in software, interrupts should be disabled while using these routines.

Writing a bit is accomplished as follows:

- (1) User initializes the Y-REG to a value indicating 'number of counts to tapeout toggle'. This value will vary according to the path length since the prior tapeout toggle. Carry is cleared to write a "0" and set to write a "1".
- (2) Subroutine WRBIT is called. It will time out (based on Y-REG count) and toggle the tapeout line, then return with the CARRY and A-REG unchanged, the X-REG decremented, and the Y-REG cleared. Zero and Neg flags will reflect the result of decrementing the X-REG. This is useful as a bit count.

Reading a bit is accomplished as follows:

(1) The Y-REG is initialized to a value indicating 'number of counts since last tapin toggle' where 'toggle' means edge sensed. This value will vary according to the path taken since prior tapin toggle.

(2) RDBIT subroutine is called. It will loop while waiting for a toggle of the tapin signal, while decrementing the Y-REG once every 12  $\mu$ sec. After sensing the toggle, a comparison on the Y-REG sets the carry:

0 means toggle came 'early'

1 means toggle came 'late'.

RDBIT is an entry which calls RDBIT twice. In this usage, the Y-REG is decremented once every 12  $\mu$ sec for a full cycle (two <sup>tapin</sup> toggles).

The final carry state indicates whether a 0 (short cycle) or 1 (long cycle) was read. The A-REG is used, Y-REG modified, X-REG unchanged.  
Note: Register location for 'LASTIN' must be provided

### Reading a byte:

- (1) Initialize Y-REG as in reading a bit (taking extra path lengths in mind).
- (2) call RDBYTE. A byte is read and left in the A-REG. X is cleared.

### Writing a Record:

- (1) user initializes the page  $\emptyset$  pointers (A1L, A1H) and (A2L, A2H) to the starting and ending addresses of a block of data to be written. These addresses must be in standard binary form.
- (2) call WRITE
  - (a) 10-second header is written.
  - (b) sync bit written.
  - (c) Data block written. (A1L, A1H) pointer is incremented until it is greater than (A2L, A2H).  
All registers ~~are used.~~  
~~PARIT circuit are used.~~
  - (d) checksum is written.
  - (e) sound BELL

### Reading a Record :

- (1) Initialize (AIL, AIH) and (AZL, AZH) to the starting and ending addresses for the block of data to be read.
- (2) call READ
  - (a) Look for toggle on tape in line.
  - (b) waits 3 seconds for tape to reach speed.
  - (c) Look for Tapein toggle.
  - (d) Scan header half-bit by half-bit waiting for sync bit.
  - (e) Read data block, advancing pointer (AIL, AIH) until greater than (AZL, AZH)
  - (f) Read ~~pa~~ checksum byte. If mismatch then print "ERR"
  - (g) sound BELL.

Note that all registers and page ~~o~~ locations LASTIN and CHKSUM are used

0000 should now contain a negative value (0F-FF hex). If so, your hardware is working.

Writing a <sup>single-record</sup> Tape

- (1) Initialize a block of memory to be written.
- (2) Enter the cassette ~~TWRITE~~ routines by hand. You may wish to store these programs permanently on PROM or EPROM.
- (3) Initialize locations 3C and 3D <sup>(A1L and A1H)</sup> to the 16-bit starting address for the data block to be written. The low-order half of the address must be in A1L, the high-order half in A1H.
- (4) Initialize A2L and A2H <sup>(locations)</sup> (3E and 3F) to the 16-bit ending address for the data block.
- (5) Store the following program in memory

```
TWRITE JSR WRITE 20
      JMP MON 4C IF FF
```

- (6) Run TWRITE. Immediately after typing the run command, start the recorder. It must be in the RECORD mode with the mic <sup>jack</sup> cable connected to the interface. The mic cable should be removed prior to writing. When done, the cursor will return. Allow 10 seconds for the header and 5 to 10 seconds for

0000 should now contain a negative value (00-FF hex). If so, your hardware is working.

Writing a <sup>single-record</sup> Tape

- (1) Initialize a block of memory to be written.
- (2) Enter the cassette ~~WRITE~~ routines by hand. You may wish to store these programs permanently on PROM or EPROM.
- (3) Initialize locations 3C and 3D <sup>(A1L and A1H)</sup> to the 16-bit starting address for the data block to be written. The low-order half of the address must be in A1L, the high-order half in A1H.
- (4) Initialize A2L and A2H <sup>(locations 3E and 3F)</sup> to the 16-bit ending address for the data block.

(5) Store the following program in memory

```

TURITE JSR WRITE 20
        JMP MON   4C IF FF
    
```

(6) Run TURITE. Immediately after typing the run command, start the recorder. It must be in the RECORD mode with the MIC <sup>jack</sup> cable connected to the interface. When the MIC cable is removed prior to writing. When done, the cursor will return. Allow 10 seconds for the tape to be written to 10 records for

## Reading a ~~single~~ single-record Tape

- (1) Enter the cassette routines into memory (if not already there).
- (2) Initialize AIL, AII, AZL and AZH as for writing tapes.
- (3) store the following program in memory.

```

TREAD   JSR   READ   20
        JMP   MON    4C  IF  FF
  
```

- (4) Run TREAD. Immediately after typing the run command, start the recorder in play mode. The tape should be rewound prior to reading. The volume setting should be nominal and the EAR jack connected to the interface.
- (5) when done each record, the ~~ERR~~ cursor will return. The word ERR will appear if the checksum<sup>byte</sup> doesn't match the data read. If you read fewer than the total number of data bytes on the record, this will occur. If you try to read more bytes than are on the record, the program may hang necessitating a system RESET.

Variable Allocation

Page & workspace should be assigned for the following variables:

```

AIL
AIH
AZL
AZH
LASTIN
CHKSUM

```

The only restriction is that AIL must immediately precede AIH and AZL must immediately precede AZH otherwise you may assign these variables differently than the provided listing.

~~user interface on non-APPLE~~

User supplied subroutines

For ERR printout and BELL prompts, the user must provide a character out subroutine, COUNT. The assembly listing provided uses the APPLE-1 entry point FFEF for this subroutine, you may substitute your own. The A, X- and Y-REG must not be disturbed by this subroutine. The byte to be output is passed in the A-REG.



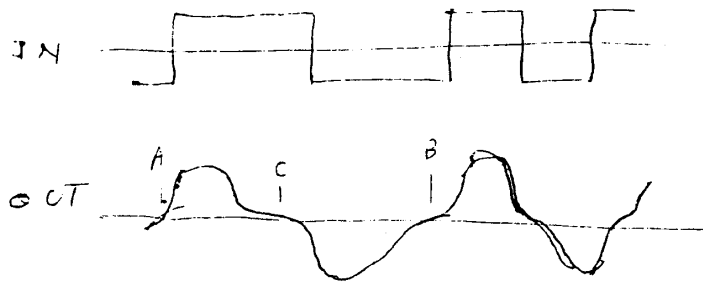
Writing and Reading Multi-Record Tapes

To write and read multi-record tapes the user must supply a program which sets up the 'start' and 'end' pointers (A1L, A1H) and (A2L, A2H), calls <sup>the</sup> READ or WRITE subroutine, then repeats the address pointers <sup>retry</sup> and subroutine call for all further records. <sup>Even</sup> If the tape is not stopped, it is permissible to spend a small amount of time calculating between records, since the first part of the header is ignored.

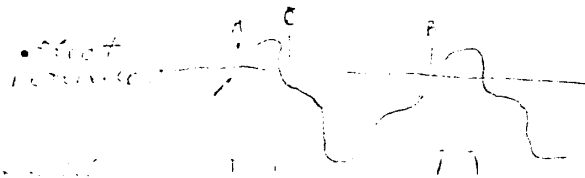
## RELIABILITY

I have tested the interface at APPLE over millions of bits without failure. I have used the cheapest tapes I could find and the cheapest recorders. The test patterns were representative of random data. What were some of the considerations?

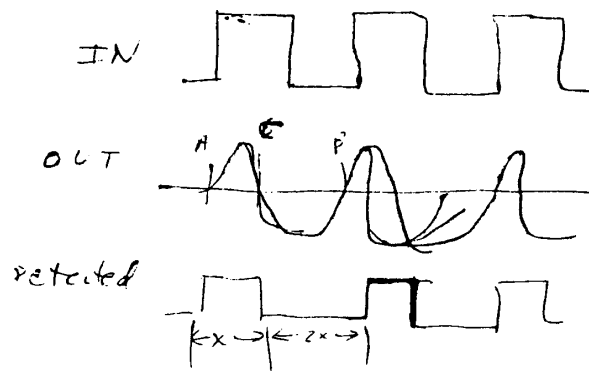
First, lets look at a typical input/output wave forms:



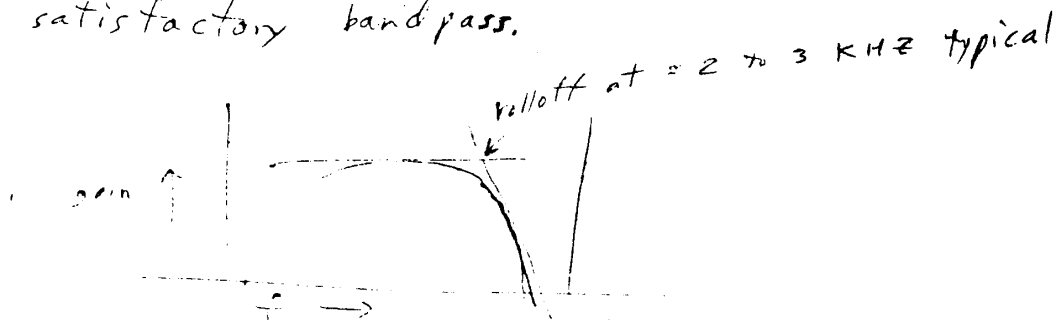
It can be seen that zero crossings of the output are ~~quite~~<sup>only</sup> very approximate due to high-frequency cutoff. Slight differentiation of this signal, coupled with hysteresis (Schmitt-trigger action) were included in the interface zero-crossing detector. Due to the nature of the recording format (one full cycle per data bit) there can be no average DC offset of the signal being read. The effect of a DC offset is to vary the zero-crossing detection point.



To counteract certain types of distortion (including a DC offset) present in some recorders, a data bit is sampled over a full cycle, never over a half-cycle. (From A to B on distorted waveform above.) My 'favorite' recorder outputs a square wave as a rectangle wave (below) yet works reliably with this interface.



Reading a string of zeroes or a string of ones presents no major problem. A major problem does crop up when the data contains mixes which ~~show up in~~ cheap recorders but not good ones. This has to do with the ~~soft~~ read and write amplifiers within the recorder. Virtually all recorders have a satisfactory bandpass.



Many ~~Apple II~~ FII type cassette interfaces use a 'high tone' in the range of 2 kHz. The gain of the recorder is satisfactory in this range but not the relative phase shift between the two tones used. ~~Apple II~~ The read (and write) amplifiers in the recorder delay the two fundamental tones by

ENTRY TO VERIFY A CASSETTE TAPE IN MEM WITH A PROGRAM (CALL 043)

```

0070- 85 D8 STX $08 Reserve X-REG
0071- 38 SEC
0083- A2 FF LDX #FFF
0082- 85 4D LDA #4D, X Calc length
0084- F5 DB SBC #DB, X in (CE, CF)
0085- 95 CF STA #CF, X
0089- E9 INX
0089- F0 F7 BEQ #0082 set Adr's for
0088- 20 1E F1 JSR #F11E → CE, CF read
008E- 20 9C 03 JSR #009C → Read/Verify
0091- A2 01 LDX #01
0093- 20 2C F1 JSR #F12C set Adr's for
0095- 20 9C 03 JSR #009C → Program Read
0099- A6 D8 LDX $08 restore X-REG
0099- 68 RTS

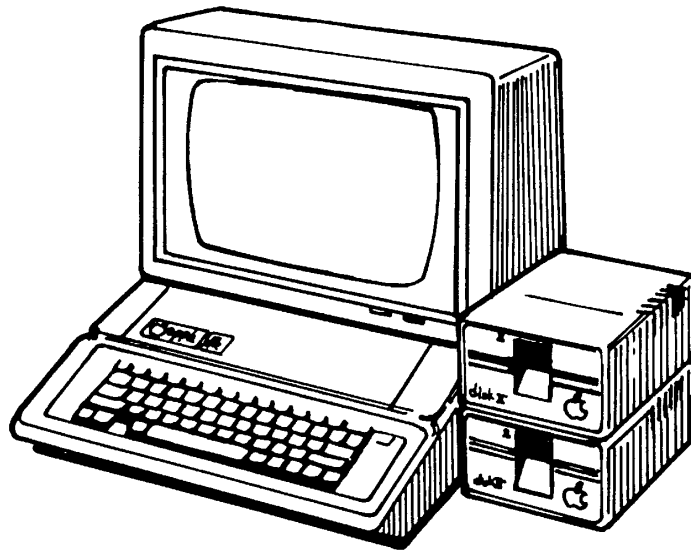
009C- 20 FA FC JSR #FCFA
009F- A9 16 LDA #16
00A1- 20 09 FC JSR #F0C9 Synchronize
00A4- 85 2E STA #2E on
00A6- 20 FA FC JSR #FCFA header
00A9- 90 24 LDY #24
00AB- 20 FD FC JSR #FCFD
00AC- 80 F9 BCS #03A9
00B0- 20 FD FC JSR #FCFD
00B3- A0 3B LDY #3B
00B5- 20 EC FC JSR #FDEC - Read a byte
00B9- F0 BE BEQ #03C8 (Always)
00BA- 45 2E EOR #2E (checksum)
00BC- 85 2E STA #2E
00BE- 20 BA FC JSR #FCBA - Incr A1, compare
00C1- A0 34 LDY #34 to A2 (get carry)
00C0- 90 FD BCC #03B5 one less than READ
00C5- 40 25 FF JNE #FF25 - Loop until A1 > A2
00C8- EA NOP → sound BELL
00C9- EA NOP when done
00CA- EA NOP after CHKSUM verify.
00CB- EA NOP extra delay to equalize
00CC- 01 30 CMP #30, Y thing (T121212)
00CD- F0 EB BEQ #03EA (Byte matches)
00CF- 48 PHA
00D0- 20 2D FF JSR #FF2D (output 'ERR')
00D3- 20 92 FD JSR #F092 (output (A1) - ")
00D6- 81 30 LDA #30, Y } output contents
00D8- 20 DA FD JSR #FD0A of A1
00DB- A9 A0 LDA #A0
00DE- 20 ED FD JSR #F0ED "y"
00E0- A9 A0 LDA #A0
00E2- 20 ED FD JSR #F0ED "c"
00E5- 68 PLA } output byte
00E6- 20 DA FD JSR #FD0A from tape
00E9- A9 A0 LDA #A0 } "y"
00EB- 20 ED FD JSR #F0ED
00EE- A9 80 LDA #80 } Car. RTN
00F0- 40 ED FD JNE #F0ED and return
00F3- A9 80 LDA #80 - Not used
00F5- 40 ED FD JNE #F0ED ← YC from
00F8- 40 9C 03 JSR #009C monitor
    
```

This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

## Apple-II Floating Point Package



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook



FLOATING POINT PACKAGE

The mantissa-exponent, or 'floating point', numerical representation is widely used by computers to express values with a wide dynamic range. With floating point representation, the number  $7.5 \times 10^{22}$  requires no more memory to store than the number 75 does. We have allowed for binary floating point arithmetic on the APPLE-II computer by providing a useful subroutine package in ROM, which performs the common arithmetic functions. Maximum precision is retained by these routines and overflow conditions such as 'divide by zero' are trapped for the user. The 4-byte floating point number representation is compatible with future APPLE products such as floating point BASIC.

A small amount of memory in page zero is dedicated to the floating point workspace, including the two floating-point accumulators, FP1 and FP2. After placing operands in these accumulators, the user calls subroutines in the ROM which perform the desired arithmetic operations, leaving results in FP1. Should an overflow condition occur, a jump to location \$3F5 in RAM is executed, allowing a user routine to take appropriate action.

## FLOATING POINT REPRESENTATION



## 1. Mantissa

The floating point mantissa is stored in two's complement representation with the sign at the most significant bit (MSB) position of the high-order mantissa byte. The mantissa provides 24 bits of precision, including sign, and can represent 24-bit integers precisely. Extending precision is simply a matter of adding bytes at the low-order end of the mantissa.

Except for magnitudes less than  $2^{-128}$  (which lose precision) mantissas are normalized by the floating point routines to retain maximum precision. That is, the numbers are adjusted so that the upper two high-order mantissa bits are unequal.

### High-order Mantissa Byte

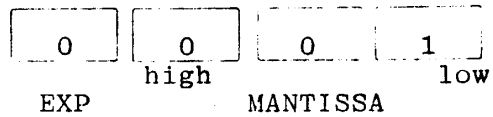
01.XXXXXX	Positive mantissa.
10.XXXXXX	Negative mantissa.
00.XXXXXX	Unnormalized mantissa, exponent = -128.
11.XXXXXX	

## 2. Exponent.

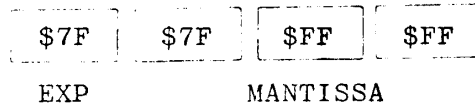
The exponent is a binary scaling factor (power of two) which is applied to the mantissa. Ranging from -128 to +127, the exponent is stored in standard two's complement representation except for the sign bit which is complemented. This representation allows direct comparison of exponents since they are stored in increasing numerical sequence. The most negative exponent, corresponding to the smallest magnitude, -128, is stored as \$00 (\$ means hexadecimal) and the most positive, +127, is stored as \$FF (all ones).

<u>Exponent</u>	<u>Stored As</u>
+1	10000001 (\$81)
+2	10000010 (\$82)
+3	10000011 (\$83)
-1	01111111 (\$7F)
-2	01111110 (\$7E)
-3	01111101 (\$7D)

The smallest magnitude which can be represented is  $+2^{-150}$ .



The largest positive magnitude which can be represented is  $+2^{128}-1$ .



## FLOATING POINT REPRESENTATION EXAMPLES

<u>Decimal Number</u>	<u>Hex Exponent</u>	<u>Hex Mantissa</u>	
+ 3	81	60 00 00	$(1.1_2 \times 2^1)$
+ 4	82	40 00 00	$(1.0_2 \times 2^2)$
+ 5	82	50 00 00	$(1.01_2 \times 2^2)$
+ 7	82	70 00 00	$(1.11_2 \times 2^2)$
+12	83	60 00 00	$(1.10_2 \times 2^3)$
+15	83	78 00 00	$(1.111_2 \times 2^3)$
+17	84	44 00 00	$(1.0001_2 \times 2^4)$
+20	84	50 00 00	$(1.01_2 \times 2^4)$
+60	85	78 00 00	$(1.111_2 \times 2^5)$
- 3	81	A0 00 00	
- 4	81	80 00 00	
- 5	82	B0 00 00	
- 7	82	90 00 00	
-12	83	A0 00 00	
-15	83	88 00 00	
-17	84	BC 00 00	
-20	84	B0 00 00	
-60	85	88 00 00	

FLOATING POINT SUBROUTINE DESCRIPTIONS

FCOMPL subroutine (address \$F4A4)

Purpose: FCOMPL is used to negate floating point numbers.

Entry: A normalized or unnormalized value is in FP1 (floating point accumulator 1).

Uses: NORM, RTLOG.

Exit: The value in FP1 is negated and then normalized to retain precision. The 3-byte FP1 extension, E, may also be altered but FP2 and SIGN are not disturbed. The 6502 A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed.

Caution: Attempting to negate  $-2^{128}$  will result in an overflow since  $+2^{128}$  is not representable, and a jump to location \$3F5 will be executed, with the following contents in FP1.

FP1:	0	\$80	0	0
	X1	M1		

Example: Prior to calling FCOMPL, FP1 contains +15.

FP1:	\$83	\$78	0	0	(+15)
	X1	M1			

After calling FCOMPL as a subroutine, FP1 contains -15.

FP1:	\$83	\$88	0	0	(-15)
	X1	M1			

FADD subroutine (address \$F46E)

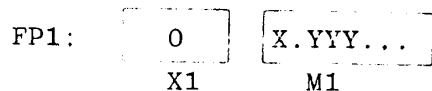
Purpose: To add two numbers in floating point form.

Entry: The two addends are in FP1 and FP2 respectively. For maximum precision, both should be normalized.

Uses: SWPALGN, ADD, NORM, RTLOG.

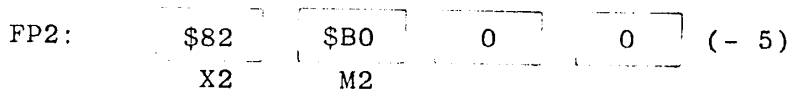
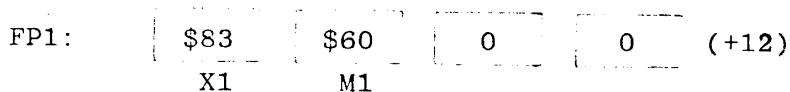
Exit: The normalized sum is left in FP1. FP2 contains the addend of greatest magnitude. E is altered but SIGN is not. The A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed. The sum mantissa is truncated to 24 bits

Caution: Overflow may result if the sum is less than  $-2^{128}$  or greater than  $+2^{128}-1$ . If so, a jump to location \$3F5 is executed leaving 0 in X1, and twice the proper sum in the mantissa M1. The sign bit is left in the carry, 0 for positive, 1 for negative.

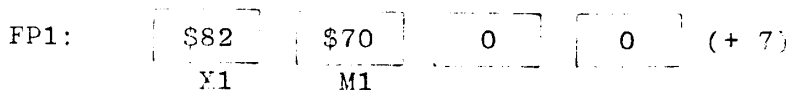


(For carry=0, true sum =  $+X.YYY... \times 2^{128}$ .)

Example: Prior to calling FADD, FP1 contains +12 and FP2 contains -5.



After calling FADD, FP1 contains +7 (FP2 contains +12).



FSUB subroutine (address \$F468)

Purpose: To subtract two floating point numbers.

Entry: The minuend is in FP1 and the subtrahend is in FP2. Both should be normalized to retain maximum precision prior to calling FSUB.

Uses: FCOMPL, ALGNSWP, FADD, ADD, NORM, RTLOG.

Exit: The normalized difference is in FP1 with the mantissa truncated to 24 bits. FP2 holds either the minuend or the negative subtrahend, whichever is of greater magnitude. E is altered but SIGN and SCR are not. The A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed.

Cautions: An exit to location \$3F5 is taken if the result is less than  $-2^{128}$  or greater than  $+2^{128}-1$ , or if the subtrahend is  $-2^{128}$ .

Example: Prior to calling FSUB, FP1 contains +7 (minuend) and FP2 contains -5 (subtrahend).

FP1:	<table border="1"><tr><td>\$82</td></tr></table>	\$82	<table border="1"><tr><td>\$70</td></tr></table>	\$70	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(+7)
\$82									
\$70									
0									
0									
	X1	M1							

FP2:	<table border="1"><tr><td>\$82</td></tr></table>	\$82	<table border="1"><tr><td>\$B0</td></tr></table>	\$B0	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(-5)
\$82									
\$B0									
0									
0									
	X2	M2							

After calling FSUB, FP1 contains +12 and FP2 contains +7.

FP1:	<table border="1"><tr><td>\$83</td></tr></table>	\$83	<table border="1"><tr><td>\$60</td></tr></table>	\$60	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(+12)
\$83									
\$60									
0									
0									
	X1	M1							

FMUL subroutine (address \$F48C)

Purpose: To multiply floating point numbers.

Entry: The multiplicand and multiplier must reside in FP1 and FP2 respectively. Both should be normalized prior to calling FMUL to retain maximum precision.

Uses: MD1, MD2, RTLOG1, ADD, MDEND.

Exit: The signed normalized floating point product is left in FP1. M1 is truncated to contain the 24 most significant mantissa bits (including sign). The absolute value of the multiplier mantissa (M2) is left in FP2. E, SIGN and SCR are altered. The A- and X-REGs are altered and the Y-REG contains \$FF upon exit.

Cautions: An exit to location \$3F5 is taken if the product is less than  $-2^{128}$  or greater than  $+2^{128}-1$ .

Notes: FMUL will run faster if the absolute value of the multiplier mantissa contains fewer '1's than the absolute value of the multiplicand mantissa.

Example: Prior to calling FMUL, FP1 contains +12 and FP2 contains -5.

FP1:	\$83	\$60	0	0	(+12)
	X1	M1			

FP2:	\$82	\$B0	0	0	(- 5)
	X2	M2			

After calling FMUL, FP1 contains -60 and FP2 contains +5.

FP1:	\$85	\$88	0	0	(-60)
	X1	M1			

FP2:	\$82	\$50	0	0	(+ 5)
	X2	M2			



FDIV subroutine (address \$F4B2)

Purpose: To perform division of floating point numbers.

Entry: The normalized dividend is in FP2 and the normalized divisor is in FP1.

Exit: The signed normalized floating point quotient is left in FP1. The mantissa (M1) is truncated to 24 bits. The 3-bit M1 extension (E) contains the absolute value of the divisor mantissa. MD2, SIGN, and SCR are altered. The A- and X-REGs are altered and the Y-REG is cleared.

Uses: MD1, MD2, MDEND.

Cautions: An exit to location \$3F5 is taken if the quotient is less than  $-2^{128}$  or greater than  $+2^{128}-1$ .

Notes: MD2 contains the remainder mantissa (equivalent to the MOD function). The remainder exponent is the same as the quotient exponent, or 1 less if the dividend mantissa magnitude is less than the divisor mantissa magnitude.

Example: Prior to calling FDIV, FP1 contains -60 (dividend) and FP2 contains +12 (divisor).

FP1:	<table border="1"><tr><td>\$85</td></tr></table>	\$85	<table border="1"><tr><td>\$88</td></tr></table>	\$88	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(-60)
\$85									
\$88									
0									
0									
	X1	M1							

FP2:	<table border="1"><tr><td>\$83</td></tr></table>	\$83	<table border="1"><tr><td>\$60</td></tr></table>	\$60	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(+12)
\$83									
\$60									
0									
0									
	X1	M1							

After calling FMUL, FP1 contains -5 and M2 contains 0.

FP1:	<table border="1"><tr><td>\$82</td></tr></table>	\$82	<table border="1"><tr><td>\$B0</td></tr></table>	\$B0	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>0</td></tr></table>	0	(- 5)
\$82									
\$B0									
0									
0									
	X1	M1							

FLOAT subroutine (address \$F451)

Purpose: To convert integers to floating point representation.

Entry: A signed (two's complement) 2-byte integer is stored in M1 (high-order byte) and M1+1 (low-order byte). M1+2 must be cleared by the user prior to entry.

Uses: NORM1.

Exit: The normalized floating point equivalent is left in FP1. E, FP2, SIGN, and SCR are not disturbed. The A-REG contains a copy of the high-order mantissa byte upon exit but the X- and Y-REGs are not disturbed. The carry is cleared.

Notes: To float a 1-byte integer, place it in M1+1 and clear M1 as well as M1+2 prior to calling FLOAT.

FLOAT takes approximately 3 msec. longer to convert zero to floating point form than other arguments. The user may check for zero prior to calling FLOAT and increase throughput.

```

*
* LOW-ORDER INTEGER BYTE IN A-REG
* HIGH-ORDER BYTE IN Y-REG
*
85 FA      XFLOAT   STA   M1+1
84 F9                      STY   M1      INIT MANT1.
A0 00                      LDY   #$0
84 FB                      STY   M1+2
05 D9                      ORA   M1      CHK BOTH BYTES
D0 03                      BNE   TOFLOAT  FOR ZERO.
85 F8                      STA   X1      IF SO, CLR X1
60                      RTS          AND RETURN.
4C 51 F4    TOFLOAT   JMP   FLOAT  ELSE FLOAT INTEGER.
    
```

(FLOAT continued)

Example: Float +274 (\$0112 hex)

Calling sequence

A0	01	LDY	#\$01	HIGH-ORDER INTEGER BYTE
A9	12	LDA	#\$12	LOW-ORDER INTEGER BYTE
84	F9	STY	M1	
85	FA	STA	M1+1	
A9	00	LDA	#\$00	
85	F8	STA	M1+2	
20	51	F4	JSR	FLOAT

Upon returning from FLOAT, FP1 contains the floating point representation of +274.

FP1:	\$88	\$44	\$80	0	(+274)
	X1	M1			

FIX subroutine (address \$F640)

Purpose: To extract the integer portion of a floating point number with truncation (ENTIER function).

Entry: A floating point value is in FP1. It need not be normalized.

Uses: RTAR.

Exit: The two-byte signed two's complement representation of the integer portion is left in M1 (high-order byte) and M1+1 (low-order byte). The floating point values +24.63 and -61.2 are converted to the integers +24 and -61 respectively. FP1 and E are altered but FP2, E, SIGN and SCR are not.

The A- and X-REGs are altered but the Y-REG is not.

Example: The floating point value +274 is in FP1 prior to calling FIX.

FP1:	\$88	\$44	\$80	0	(+274)
	X1	M1			

After calling FIX, M1 (high-order byte) and M1+1 (low-order byte) contain the integer representation of +274 (\$0112).

FP1:	\$8E	\$01	\$12	0
	X1	M1		

Note: FP1 contains an unnormalized representation of +274 upon exit.

AUXILLIARY SUBROUTINES.

NORM subroutine (address \$F463)

Purpose: To normalize the value in FP1, thus insuring maximum precision.

Entry: A normalized or unnormalized value is in FP1.

Exit: The value in FP1 is normalized. A zero mantissa will exit with X1=0 ( $2^{-128}$  exponent). If the exponent on exit is -128 (X1=0) then the mantissa (M1) is not necessarily normalized (with the two high-order mantissa bits unequal). E, FP2, SIGN, and SCR are not disturbed. The A-REG is disturbed but the X- and Y-REGs are not. The carry is set.

Example: FP1 contains +12 in unnormalized form (as  $.0011_2 \times 2^6$ ).



Upon exit from NORM, FP1 contains +12 in normalized form (as  $1.1_2 \times 2^3$ ).



NORM1 subroutine (address \$F455)

Purpose: To normalize a floating point value in FP1 when it is known the exponent is not -128 (X1=0) upon entry.

Entry: An unnormalized number is in FP1. The exponent byte should not be 0 for normal use.

Exit: The normalized value is in FP1. E, FP2, SIGN, and SCR are not disturbed. The A-REG is altered but the X- and Y-REGs are not.

ADD subroutine (address \$F425)

Purpose: To add the two mantissas (M1 and M2) as 3-byte integers).

Entry: Two mantissas are in M1 (through M1+2) and M2 (through M2+2). They should be aligned, that is with identical exponents, for use in the FADD and FSUB subroutines.

Exit: The 24-bit integer sum is in M1 (high-order byte in M1, low-order byte in M1+2). FP2, X1, E, SIGN, and SCR are not disturbed. The A-REG contains the high-order byte of the sum, the X-REG contains \$FF, and the Y-REG is not altered. The carry is the '25th' sum bit.

Example: FP1 contains +5 and FP2 contains +7 prior to calling ADD.

FP1	\$82	\$50	0	0	(+ 5)
	X1	M1			

FP2	\$82	\$70	0	0	(+ 7)
-----	------	------	---	---	-------

Upon exit, M1 contains the overflow value for +12.

Note that the sign bit is incorrect. This is taken care of with a call to the right shift routine.

FP1	\$82	\$C0	0	0	(+12)
-----	------	------	---	---	-------

ABSWAP subroutine (address \$F437).

Purpose: To take the absolute value of FP1 and then swap FP1 with FP2. Note that two sequential calls to ABSWAP will take the absolute values of both FP1 and FP2 in preparation for a multiply or divide.

Entry: FP1 and FP2 contain floating point values.

Exit: The absolute value of the original FP1 contents are in FP2 and the original FP2 contents are in FP1. The least significant bit of SIGN is complemented if a negation takes place (if the original FP1 contents are negative), by means of an increment. SCR and E are used. The A-REG contains a copy of X2, the X-REG is cleared, and the Y-REG is not altered.

RTAR subroutine (address \$F47D)

Purpose: To shift M1 right one bit position while incrementing X1 to compensate for scale. This is roughly the opposite of the NORM subroutine.

Entry: A normalized or unnormalized floating point value is in FP1.

Exit: The 6-byte field MANT1 and E is shifted right one bit arithmetically and X1 is incremented by 1 to retain proper scale. The sign bit of MANT1 (MSB of M1) is unchanged. FP2, SIGN, and SCR are not disturbed. The A-REG contains the least significant byte of E (E+2), the X-REG is cleared, and the Y-REG is not disturbed.

RTAR subroutine (continued)

Caution: If X1 increments to 0 (overflows) then an exit to location \$3F5 is taken, the 'A-REG contains the high-order MANT1 byte, M1, and X1 is cleared. FP2, SIGN, SCR, and the X- and Y-REG's are not disturbed.

Uses: RTLOG

Example: Prior to calling RTAR, FP1 contains the normalized value -7.



After calling RTAR, FP1 contains the unnormalized value -7 (note that precision is lost off the low-order end of M1).



Note: M1 sign bit is unchanged.



RTLOG subroutine (address \$F480)

Purpose: To shift the 6-byte field MANT1 and E one bit to the right (toward the least significant bit). The 6502 carry bit is shifted into the high-order M1 bit.

This is useful in correcting binary sum overflows.

Entry: A normalized or unnormalized floating point value is in FP1. The carry must be cleared or set by the user since it is shifted into the sign bit of M1.

Exit: Same as RTAR except that the sign bit of M1 is not preserved (it is set to the value of the carry bit on entry).

Caution: Same as RTAR.

Example: Prior to calling RTLOG, FP1 contains the normalized value -12 and the carry is clear.

FP1:	\$83	\$A0	0	0	(-12)
	X1	M1			

After calling RTLOG, M1 is shifted one bit to the right and the sign bit is clear. X1 is incremented by 1.

FP1:	\$84	\$50	0	0	(+20)
	X1	M1			

Note: The bit shifted off the end of MANT1 is rotated into the high order bit of the 3-byte extension E. The 3-byte E field is also shifted one bit to the right.

RTLOG1 subroutine (address \$F484)

Prupose: To shift MANT1 and E right one bit without adjusting X1. This is used by teh multiply loop. The carry is shifted into the sign bit of MANT1.

Entry: M1 and E contain a 6-byte unsigned field. E is the 3-byte low-order extension of MANT1.

Exit: Same as RTLOG except that X1 is not altered and an overflow exit cannot occur.

MD2 subroutine (address \$F4E2)

Purpose: To clear the 3-byte MANT1 field for FMUL and FDIV, check for initial result exponent overflow (and underflow), and initialize the X-REG to \$17 for loop counting.

Entry: The X-REG is cleared by teh user since it is placed in the 3 bytes of MANT1. The A-REG contains the result of an exponent addition (FMUL) or subtraction (FDIV). The carry and sign status bits should be set according to this addition or subtraction for overflow and underflow determination.

Exit: The 3 bytes of M1 are cleared (or all set to the contents of the X-REG on entry) and the Y-REG is loaded with \$17. The sign bit of the A-REG is complemented and a copy of the A-aEG is stored in X1. FP2, SIGN, SCR, and the X-REG are not disturbed.

Uses: NORM.

Caution: Exponent overflow results in an exit to location \$3F5. Exponent underflow results in an early return from the

MD2 subroutine (continued)

calling subroutine (FDIV or FMUL) with a floating point zero in FP1. Because MD2 pops a return address off the stack, it may only be called by another subroutine.

1:49 P. M., 10/3/1977

FLOATING POINT ROUTINES

PAGE: 1

```

1 *****
2 *
3 * APPLE-II FLOATING *
4 * POINT ROUTINES *
5 *
6 * COPYRIGHT 1977 BY *
7 * APPLE COMPUTER INC. *
8 *
9 * ALL RIGHTS RESERVED *
10 *
11 * S. WOZNAK *
12 *
13 *****
14 TITLE "FLOATING POINT ROUTINES"
15 SIGN EPZ $F3
16 X2 EPZ $F4
17 M2 EPZ $F5
18 X1 EPZ $F8
19 M1 EPZ $F9
20 E EPZ $FC
21 OVLOC EQU $3F5
22 ORG $F425
F425: 18 23 ADD CLC CLEAR CARRY.
F426: A2 02 24 LDX ##2 INDEX FOR 3-BYTE ADD.
F428: B5 F9 25 ADD1 LDA M1,X
F42A: 75 F5 26 ADC M2,X ADD A BYTE OF MANT2 TO MANT1.
F42C: 95 F9 27 STA M1,X
F42E: CA 28 DEX INDEX TO NEXT MORE SIGNIF. BYT
F430: 10 F7 29 BPL ADD1 LOOP UNTIL DONE.
F432: 06 F3 30 RTS RETURN
F434: 20 37 F4 31 MUL1 ASL SIGN CLEAR LSB OF SIGN.
F437: 24 F9 32 ABSWAP JSR ABSWAP ABS VAL OF M1, THEN SWAP WITH
F439: 10 05 33 BIT M1 MANT1 NEGATIVE?
F43B: 20 A4 F4 34 BPL ABSWAP1 NO, SWAP WITH MANT2 AND RETURN
F43E: E6 F3 35 JSR FCOMPL YES, COMPLEMENT IT.
F440: 30 37 ABSWAP1 SEC INCR SIGN, COMPLEMENTING LSB
F441: A2 04 38 SWAP LDX ##4 SET CARRY FOR RETURN TO MUL/DI
F443: 94 FB 39 SWAP1 STY E-1,X INDEX FOR 4-BYTE SWAP.
F445: B5 F7 40 LDA X1-1,X SWAP A BYTE OF EXP/MANT1 WITH
F447: B4 F3 41 LDY X2-1,X EXP/MANT2 AND LEAVE A COPY OF
F449: 94 F7 42 STY X1-1,X MANT1 IN E (3 BYTES). E+3 USL
F44B: 95 F3 43 STA X2-1,X
F44D: CA 44 DEX ADVANCE INDEX TO NEXT BYTE.
F44E: D0 F3 45 BNE SWAP1 LOOP UNTIL DONE.
F450: 60 46 RTS RETURN
F451: A9 8E 47 FLOAT LDA #38E INIT EXP1 TO 14,
F453: 85 F3 48 STA X1 THEN NORMALIZE TO FLOAT.
F455: A5 F9 49 NORM1 LDA M1 HIGH-ORDER MANT1 BYTE.
F457: 09 C0 50 CMP #9C0 UPPER TWO BITS UNEQUAL?
F459: 30 C0 51 BMI RTS1 YES, RETURN WITH MANT1 NORMAL
F45B: C6 F8 52 DEC X1 DECREMENT EXP1.
F45D: 06 FB 53 ASL M1+2
F45F: 26 FA 54 ROL M1+1 SHIFT MANT1 (3 BYTES) LEFT.
    
```

FLOATING POINT ROUTINES

PAGE: 2

49 P. M., 10/3/1977

461:	26 F9	55		ROL	M1	
463:	A5 F8	56	NORM	LDA	X1	EXP1 ZERO?
	D0 EE	57		BNC	NORM1	NO, CONTINUE NORMALIZING.
467:	60	58	RTS1	RTS		RTS RETURN.
468:	20 A4 F4	59	F2SUB	JSR	FCOMPL	COMPL MANT1, CLEARS CARRY UNLESS
46B:	20 7B F4	60	SWPALGN	JSR	ALGNSWP	RIGHT, SHIFT MANT1 OR SWAP WITH
46E:	A5 F4	61	FADD	LDA	X2	
470:	C5 F8	62		CMP	X1	COMPARE EXP1 WITH EXP2.
472:	D0 F7	63		BNE	SWPALGN	IF #, SWAP ADDEND OR ALIGN MANT
474:	20 25 F4	64		JSR	ADD	ADD ALIGNED MANTISSAS.
477:	50 EA	65	ADDEND	BVC	NORM	NO OVERFLOW, NORMALIZE RESULT
479:	70 05	66		BVS	RTLOG	OV: SHIFT M1 RIGHT, CARRY INTO
47B:	90 C4	67	ALGNSWP	BCC	SWAP	SWAP IF CARRY CLEAR.
		68	*			ELSE SHIFT RIGHT ARITH.
47D:	A5 F9	69	RTAR	LDA	M1	SIGN OF MANT1 INTO CARRY FOR
47F:	0A	70		ASL	A	RIGHT ARITH SHIFT.
480:	E6 F8	71	RTLOG	INC	X1	INCR X1 TO ADJUST FOR RIGHT SHI
482:	F0 75	72		BEQ	OVFL	EXP1 OUT OF RANGE.
484:	A2 FA	73	RTLOG1	LDX	#\$FA	INDEX FOR 6-BYTE RIGHT SHIFT.
486:	76 FF	74	RDR1	ROR	E+3, X	
488:	E8	75		INX		NEXT BYTE OF SHIFT.
489:	D0 FB	76		BNL	ROR1	LOOP UNTIL DONE.
48B:	60	77		RTS		RETURN.
48C:	20 32 F4	78	FMUL	JSR	MD1	ABS VAL OF MANT1, MANT2.
48F:	65 F8	79		ADC	X1	ADD EXP1 TO EXP2 FOR PRODUCT EX
491:	20 E2 F4	80		JSR	MD2	CHECK PROD. EXP AND PREP. FOR M
494:	18	81		CLC		CLEAR CARRY FOR FIRST BIT.
495:	20 84 F4	82	MUL1	JSR	RTLOG1	M1 AND E RIGHT (PROD AND MPLIEH
497:	90 03	83		BCC	MUL2	IF CARRY CLEAR, SKIP PARTIAL RE
49A:	20 25 F4	84		JSR	ADD	ADD MULTIPLICAND TO PRODUCT.
49D:	88	85	MUL2	DEY		NEXT MUL ITERATION.
49E:	10 15	86		BPL	MUL1	LOOP UNTIL DONE.
4A0:	46 F3	87	MDEND	LSR	SIGN	TEST SIGN LSB.
4A2:	90 BF	88	NORMX	BCC	NORM	IF EVEN, NORMALIZE PROD, ELSE CO
4A4:	38	89	FCOMPL	SEC		SET CARRY FOR SUBTRACT.
4A5:	A2 03	90		LDX	#\$3	INDEX FOR 3-BYTE SUBTRACT.
4A7:	A9 00	91	COMPL1	LDA	#\$0	CLEAR A.
4A9:	F5 F8	92		SBC	X1, X	SUBTRACT BYTE OF EXP1.
4AB:	95 F8	93		STA	X1, X	RESTORE IT.
4AD:	CA	94		DEX		NEXT MORE SIGNIFICANT BYTE.
4AE:	D0 F7	95		BNE	COMPL1	LOOP UNTIL DONE.
4B0:	F0 C5	96		BEQ	ADDEND	NORMALIZE (OR SHIFT RT IF OVFL)
4B2:	20 32 F4	97	FDIV	JSR	MD1	TAKE ABS VAL OF MANT1, MANT2.
4B5:	E5 F8	98		SBC	X1	SUBTRACT EXP1 FROM EXP2.
4B7:	20 E2 F4	99		JCR	MD2	SAVE AS QUOTIENT EXP.
4BA:	38	100	DIV1	SEC		SET CARRY FOR SUBTRACT.
4BB:	A2 02	101		LDX	#\$2	INDEX FOR 3-BYTE SUBTRACTION.
4BD:	B5 F5	102	DIV2	LDA	M2, X	
4BF:	F5 FC	103		SBC	E, X	SUBTRACT A BYTE OF E FROM MANT.
4C1:	48	104		PHA		SAVE ON STACK.
4C2:	CA	105		DEX		NEXT MORE SIGNIFICANT BYTE.
4C3:	10 F8	106		BPL	DIV2	LOOP UNTIL DONE.
4C5:	A2 FD	107		LDX	#\$FD	INDEX FOR 3-BYTE CONDITIONAL M1
4C7:	68	108	DIV3	PLA		PULL BYTE OF DIFFERENCE OFF S.

FLOATING POINT ROUTINES

PAGE: 3

1:47 P. M., 10/3/1977

```

F408: 90 02 109      BCC DIV4      IF M2<E THEN DON'T RESTORE M2
F40A: 95 F8 110      STA M2+3,X
F40C: E8 111      DIV4      INX      NEXT LESS SIGNIFICANT BYTE.
      D: D0 F8 112      BNE DIV3      LOOP UNTIL DONE.
F40F: 26 FB 113      ROL M1+2
F4D1: 26 FA 114      ROL M1+1      ROLL QUOTIENT LEFT, CARRY INTO
F4D3: 26 F9 115      ROL M1
F4D5: 06 F7 116      ASL M2+2
F4D7: 26 F6 117      ROL M2+1      SHIFT DIVIDEND LEFT.
F4D9: 26 F5 118      ROL M2
F4DB: E0 1C 119      BCS OVFL      OVFL IS DUE TO UNNORMED DIVIS
F4DD: 88 120      DEY      NEXT DIVIDE ITERATION.
F4DE: D0 DA 121      BNL DIV1      LOOP UNTIL DONE 23 ITERATIONS
F4E0: F0 BE 122      BEQ MDEND     NORM. QUOTIENT AND CORRECT SIG
F4E2: 86 FB 123      MD2      STX M1+2
F4E4: 86 FA 124      STX M1+1      CLEAR MANT1 (3 BYTES) FOR MUL
F4E6: 86 F9 125      STX M1
F4E8: B0 OD 126      BCS OVCHK     IF CALC. SET CARRY, CHECK FOR
F4EA: 30 04 127      BMI MD3      IF NEG THEN NO UNDERFLOW.
F4EC: 68 128      PLA      POP ONE RETURN LEVEL.
F4ED: 68 129      PLA
F4EE: 90 B2 130      BCC NORMX     CLEAR X1 AND RETURN.
F4F0: 49 80 131      MD3      EOR #$80     COMPLEMENT SIGN BIT OF EXPONEN
F4F2: 85 F8 132      STA X1      STORE IT.
F4F4: A0 17 133      LDY #$17     COUNT 24 MUL/23 DIV ITERATIONS
F4F6: 60 134      RTS      RETURN.
F4F7: 10 F7 135      OVCHK      BPL MD3      IF POSITIVE EXP THEN NO OVFL.
F4F9: 4C F5 03 136      OVFL      JMP OVLOC
      137      ORG $F63D
      D: 20 7D F4 138      FIX1      JSR RTAR
F640: A5 F8 139      FIX      LDA X1
F642: 10 13 140      BPL UNDFL
F644: 09 8E 141      CMP #$8E
F646: D0 F5 142      BNE FIX1
F648: 24 F9 143      BIT M1
F64A: 10 0A 144      BPL FIXRTS
F64C: A5 FB 145      LDA M1+2
F64E: F0 06 146      BEQ FIXRTS
F650: E6 FA 147      INC M1+1
F652: D0 02 148      BNE FIXRTS
F654: E6 F9 149      INC M1
F656: 60 150      FIXRTS     RTS      RTS
F657: A9 00 151      UNDFL     LDA #$0
F659: 85 F9 152      STA M1
F65B: 85 FA 153      STA M1+1
F65D: 60 154      RTS

```

\*\*\*\*\*SUCCESSFUL ASSEMBLY: NO ERRORS

CROSS-REFERENCE: FLOATING POINT ROUTINES

ABSWAP	F437	0032																		
ABSWAP1	F440	0034																		
ADD	F425	0064	0084																	
ADD1	F428	0029																		
ADDEND	F477	0096																		
ALGNSWP	F47B	0060																		
COMPL1	F4A7	0095																		
DIV1	F46A	0121																		
DIV2	F46D	0106																		
DIV3	F4C7	0112																		
DIV4	F4CC	0109																		
E	00FC(Z)	0039	0074	0103																
FADD	F46E																			
FDDMPL	F4A4	0035	0059																	
FDIV	F4B2																			
FIX	F640																			
FIX1	F63D	0142																		
FIXRTS	F656	0144	0146	0148																
FLOAT	F451																			
FMUL	F48C																			
FSUB	F468																			
M1	00F9(Z)	0025	0027	0033	0049	0053	0054	0055	0069	0113	0114	0115								
		0123	0124	0125	0143	0145	0147	0149	0152	0153										
M2	00F5(Z)	0026	0102	0110	0116	0117	0118													
MD1	F432	0078	0097																	
MD2	F4E2	0080	0099																	
MD3	F4F0	0127	0135																	
MEND	F4A0	0122																		
MUL1	F495	0036																		
MUL2	F49D	0083																		
NORM	F463	0065	0088																	
NORM1	F455	0057																		
NORMX	F4A2	0130																		
OVCHK	F4F7	0126																		
OVFL	F4F9	0072	0119																	
OVLUC	03F5	0136																		
RDR1	F486	0076																		
RTAR	F47D	0138																		
RTLOG	F480	0066																		
RTLOG1	F484	0082																		
RTR1	F467	0051																		
SIGN	00F3(Z)	0031	0036	0087																
SWAP	F441	0067																		
SWAP1	F443	0045																		
SWFALGN	F46B	0063																		
UNLFL	F657	0140																		
X1	00F8(Z)	0040	0042	0048	0052	0056	0062	0071	0079	0092	0093	0098								
		0132	0139																	
X2	00F4	0041	0043	0061																

This page is not part of the original Wonderbook

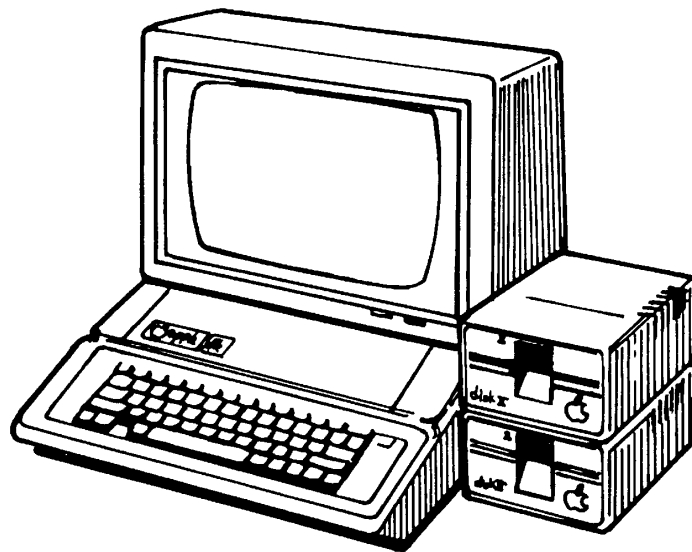


# The Woz Wonderbook

DOCUMENT

Apple-II

Sweet-16 -- The 6502 Dream Machine



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

## SWEET16 - THE 6502 DREAM MACHINE

While writing APPLE BASIC for a 6502 microprocessor I repeatedly encountered a variant of MURPHY'S LAW. Briefly stated, any routine operating on 16-bit data will require at least twice the code that it should. Programs making extensive use of 16-bit pointers (such as compilers, editors, and assemblers) are included in this category. In my case, even the addition of a few double-byte instructions to the 6502 would have only slightly alleviated the problem. What I really needed was a 6502/RCA 1800 hybrid - a powerful 8-bit data handler complemented by an easy to use processor with an abundance of 16-bit registers and excellent pointer capability. My solution was to implement a non-existent (meta) 16-bit processor in software, interpreter style, which I call SWEET16.

SWEET16 is based around sixteen 16-bit registers (R0-R15), actually 32 memory locations. R0 doubles as the SWEET16 accumulator (ACC), R15 as the program counter (PC), and R14 as the status register. R13 holds compare instruction results and R12 is the subroutine return stack pointer if SWEET16 subroutines are used. All other SWEET16 registers are at the user's unrestricted disposal.

SWEET16 instructions fall into register and non-register categories. The register ops specify one of the sixteen registers to be used as either a data element or a pointer to

data in memory depending on the specific instruction. For example, INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register ops take 1 byte of code each. The non-register ops are primarily 6502 style branches with the second byte specifying a +127 byte displacement relative to the address of the following instruction. Providing that the prior register op result meets a specified branch condition, the displacement is added to SWEET16's PC, effecting a branch.

SWEET16 is intended as a 6502 enhancement package, not a stand-alone processor. A 6502 program switches to SWEET16 mode with a subroutine call and subsequent code is interpreted as SWEET16 instructions. The non-register op RTN returns the user program to 6502 mode after restoring the internal register contents (A, X, Y, P, and S). The following example illustrates how to use SWEET16.

300	B9 00 02	LDA	IN,Y	Get a char.
303	C9 CD	CMP	"M"	"M" for move?
305	D0 09	BNE	NOMOVE	No, skip move.
307	20 89 F6	JSR	SW16	Yes, call SWEET16.
30A	41	MLOOP	LD @R1	R1 holds source address.
30B	52	ST	@R2	R2 holds dest. address.
30C	F3	DCR	R3	Decrement length.
30D	07 FB	BNZ	MLOOP	Loop until done.
30F	00	RTN		Return to 6502 mode.
310	C9 C5	NOMOVE	CMP "E"	"E" char?
312	D0 13	BEQ	EXIT	Yes, exit.
314	C8	INY		No, continue

NOTE: Registers A, X, Y, P, and S are not disturbed by SWEET16.

## INSTRUCTION DESCRIPTIONS

The SWEET16 opcode list is short and uncomplicated. Excepting relative branch displacements, hand assembly is trivial. All register opcodes are formed by combining two hex digits, one for the opcode and one to specify a register. For example, opcodes 15 and 45 both specify register R5 while codes 23, 27 and 29 are all ST ops. Most register ops are assigned in complementary pairs to facilitate remembering them. Thus LD and ST are opcodes 2n and 3n respectively, while LD @ and ST @ are codes 4n and 5n.

Opcodes 0 to C (hex) are assigned to the thirteen non-register ops. Except for RTN (opcode 0), BK (0A), and RS (B), the non-register ops are 6502 style relative branches. The second byte of a branch instruction contains a +127 byte displacement value (in two's complement form) relative to the address of the instruction immediately following the branch. If a specified branch condition is met by the prior register op result, the displacement is added to the PC effecting a branch. Except for BR (Branch always) and BS (Branch to Subroutine), the branch opcodes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are opcodes 4 and 5 while Branch if Zero and Branch if NonZero are opcodes 6 and 7.

SWEET16 OP CODE SUMMARY

<u>Register Ops</u>		<u>Non-register Ops</u>	
1n	SET Rn, Constant (Set)	00	RTN (Return to 6502 mode)
2n	LD Rn (Load)	01	BR ea (Branch always)
3n	ST Rn (Store)	02	BNC ea (Branch if No Carry)
4n	LD @Rn (Load indirect)	03	BC ea (Branch if Carry)
5n	ST @Rn (Store indirect)	04	BP ea (Branch if Plus)
6n	LDD @Rn (Load double indirect)	05	BM ea (Branch if Minus)
7n	STD @Rn (Store double indirect)	06	BZ ea (Branch if Zero)
8n	POP @Rn (Pop indirect)	07	BNZ ea (Branch if NonZero)
9n	STP @Rn (Store pop indirect)	08	BMI ea (Branch if Minus 1)
An	ADD Rn (Add)	09	BNMI ea (Branch if Not Minus 1)
Bn	SUB Rn (Sub)	0A	BK (Break)
Cn	POPD @Rn (Pop double indirect)	0B	RS (Return from Subroutine)
Dn	CPR Rn (Compare)	0C	BS ea (Branch to Subroutine)
En	INR Rn (Increment)	0D	(Unassigned)
Fn	DCR Rn (Decrement)	0E	(Unassigned)
		0F	(Unassigned)

REGISTER OPS

SET Rn, Constant 1n low high (Set)  
constant

The 2-byte constant is loaded into Rn (n = 0 to F, hex) and branch conditions set accordingly. The carry is cleared.

Example

15 34 A0                    SET R5, A034    R5 now contains A034

LD Rn 2n (Load)

The ACC (R0) is loaded from Rn and branch conditions set according to the data transferred. The carry is cleared and the contents of Rn are not disturbed.

Example

15 34 A0                    SET R5, A034  
 24                            LD R5                    ACC now contains A034

ST Rn 3n (Store)

The ACC is stored into Rn and branch conditions set according to the data transferred. The carry is cleared and the ACC contents are not disturbed.

Example

25                            LD R5                    Copy the contents  
 36                            ST R6                    of R5 to R6.

LD @Rn

4n

(Load indirect)

The low-order ACC byte is loaded from the memory location whose address resides in Rn and the high-order ACC byte is cleared. Branch conditions reflect the final ACC contents which will always be positive and never minus 1. The carry is cleared. After the transfer, Rn is incremented by 1.

Example

15 34 A0	SET R5, A034	
45	LD @R5	ACC is loaded from memory location A034 and R5 is incremented to A035.

ST @Rn

5n

(Store indirect)

The low-order ACC byte is stored into the memory location whose address resides in Rn. Branch conditions reflect the 2-byte ACC contents. The carry is cleared. After the transfer, Rn is incremented by 1.

Example

15 34 A0	SET R5, A034	Load pointers R5 and R6 with A034 and 9022.
16 22 90	SET R6, 9022	
45	LD @R5	Move a byte from location A034 to location 9022.
56	ST @R6	Both pointers are incremented.





POP @Rn

8n

(Pop indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn, after Rn is decremented by 1 and the high order ACC byte is cleared. Branch conditions reflect the final 2-byte ACC contents which will always be positive and never minus 1. The carry is cleared. Because Rn is decremented prior to loading the ACC, single byte stacks may be implemented with the ST @Rn and POP @Rn ops (Rn is the stack pointer).

Example

15 34 A0	SET R5, A034	Init stack pointer.
10 04 00	SET R0, 4	Load 4 into ACC.
35	ST @R5	Push 4 onto stack.
10 05 00	SET R0, 5	Load 5 into ACC.
35	ST @R5	Push 5 onto stack.
10 06 00	SET R0, 6	Load 6 into ACC.
35	ST @R5	Push 6 onto stack.
85	POP @R5	Pop 6 off stack into ACC.
85	POP @R5	Pop 5 off stack.
85	POP @R5	Pop 4 off stack.

STP @Rn

9n

(STORE POP indirect)

The low order ACC byte is stored into the memory location whose address resides in Rn after Rn is decremented by 1. Then the high-order ACC byte is stored into the memory location whose address resides in Rn after Rn is again decremented by 1. Branch conditions will reflect the 2-byte ACC contents which are not modified. STP @Rn and POP @Rn are used together to move data blocks beginning at the greatest address and working down. Additionally, single-byte stacks may be implemented with the STP @Rn and LDA @Rn ops.

Example

14	34	A0	SET R4, A034	Init pointers.
15	22	90	SET R5, 9022	
84			POP @R4	Move byte from A033
95			STP @R5	to 9021.
84			POP @R4	Move byte from A032
95			STP @R5	to 9020.

ADD Rn

An

(Add)

The contents of Rn are added to the contents of the ACC (R0) and the low-order 16 bits of the sum restored in ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents.

Example

10 34 76	SET R0, 7634	Init R0 (ACC)
11 27 42	SET R1, 4227	and R1.
A1	ADD R1	Add R1 (sum = B85B, carry clear)
A0	ADD R0	Double ACC (R0) to 70B6 with carry set.



POPD @Rn

Cn

(POP Double-byte indirect)

Rn is decremented by 1 and the high-order ACC byte is loaded from the memory location whose address now resides in Rn. Then Rn is again decremented by 1 and the low-order ACC byte is loaded from the corresponding memory location. Branch conditions reflect the final ACC contents. The carry is cleared. Because Rn is decremented prior to loading each of the ACC halves, double-byte stacks may be implemented with the STD @Rn and POPD @Rn ops (Rn is the stack pointer).

Example

15 34 A0	SET R5, A034	Init stack pointer.
10 12 AA	SET R0, AA12	Load AA12 into ACC.
75	STD @R5	Push AA12 onto stack.
10 34 BB	SET R0, BB34	Load BB34 into ACC.
75	STD @R5	Push BB34 onto stack.
10 56 CC	SET R0, CC56	Load CC56 into ACC.
C5	POPD @R5	Pop CC56 off stack.
C5	POPD @R5	Pop BB34 off stack.
C5	POPD @R5	Pop AA12 off stack.

CPR Rn

Dn

(Compare)

The ACC (R0) contents are compared to Rn by performing the 16-bit binary subtraction ACC-Rn and storing the low order 16 difference bits in R13 for subsequent branch tests. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents then the carry is set, otherwise it is cleared. No other registers, including ACC and Rn, are disturbed.

Example

15 34 A0	SET R5, A034	Pointer to memory.
16 BF A0	SET R6, A0BF	Limit address.
10 00 00	LOOP SET R0, 0	Zero data.
75	STD @R5	Clear 2 locs, incr R5 by 2.
25	LD R5	Compare pointer R5
D6	CPR R6	to limit R6.
02 F8	BNC LOOP	Loop if carry clear.

INR Rn

En

(Increment)

The contents of Rn are incremented by 1. The carry is cleared and other branch conditions reflect the incremented value.

Example

15 34 A0	SET R5, A034	Init R5 (pointer)
10 00 00	SET R0, 0	Zero to R0.
55	ST @R5	Clears loc A034 and incrs R5 to A035.
E5	INR R5	Incr R5 to A036
55	ST @R5	Clears loc A036 (not A035)

DCR Rn

Fn

(Decrement)

The contents of Rn are decremented by 1. The carry is cleared and other branch conditions reflect the decremented value.

Example (Clear 9 bytes beginning at loc A034)

15 34 A0	SET R5, A034	Init pointer.
14 09 00	SET R4, 9	Init count.
10 00 00	SET R0, 0	Zero ACC.
55	LOOP ST @R5	Clear a mem byte.
F4	DCR R4	Decr. count.
07 FC	BNZ LOOP	Loop until zero.



## NON-REGISTER INSTRUCTIONS

RTN

00

(Return to 6502 mode)

Control is returned to the 6502 and program execution continues at the location immediately following the RTN instruction. The 6502 registers and status conditions are restored to their original contents (prior entering SWEET16 mode)

BR ea

01

d

(Branch Always)

An effective address (ea) is calculated by adding the signed displacement byte (d) to the PC. The PC contains the address of the instruction immediately following the BR, or the address of the BR op plus 2. The displacement is a signed twos complement value from -128 to +127. Branch conditions are not changed. Note that effective address calculation is identical to that for 6502 relative branches. The hex add and subtract features of the APPLE-II monitor may be used to calculate displacements.

$$d = \$80 \quad ea = PC + 2 - 128$$

$$d = \$81 \quad ea = PC + 2 - 127$$

$$d = \$FF \quad ea = PC + 2 - 1$$

$$d = \$00 \quad ea = PC + 2 + 0$$

$$d = \$01 \quad ea = PC + 2 + 1$$

$$d = \$7E \quad ea = PC + 2 + 126$$

$$d = \$7F \quad ea = PC + 2 + 127$$
Example

\$300: 01 50      BR \$352

BNC ea 02 d (Branch if No Carry)

A branch to the effective address is taken only if the carry is clear, otherwise execution resumes as normal with the next instruction. Branch conditions are not changed.

BC ea 03 d (Branch if Carry set)

A branch is effected only if the carry is set. Branch conditions are not changed.

BP ea 04 d (Branch if Plus)

A branch is effected only if the prior 'result' (or most recently transferred data) was positive. Branch conditions are not changed.

Example (Clear mem from loc. A034 to A03F)

```

15 34 A0          SET  R5, A034  Init pointer.
14 3F A0          SET  R4, A03F  Init limit.
10 00 00  LOOP   SET  R0, 0
55                ST   @R5      Clear mem byte, incr R5.
24                LD   R4      Compare limit to
D5                CPR  R5      pointer.
04 F8             BP   LOOP     Loop until done.
    
```

BM ea 05 d (Branch if Minus)

A branch is effected only if the prior 'result' was minus (negative, MSB = 1). Branch conditions are not changed.

BZ ea                    06 d                    (Branch if Zero)

A branch is effected only if the prior 'result' was zero.  
Branch conditions are not changed.

BNZ ea                    07 d                    (Branch if NonZero)

A branch is effected only if the prior 'result' was  
non-zero. Branch conditions are not changed.

BM1 ea                    08 d                    (Branch if Minus 1)

A branch is effected only if the prior 'result' was  
minus 1 (\$FFFF hex). Branch conditions are not changed.

BNM1 ea                    09 d                    (Branch if Not Minus 1)

A branch is effected only if the prior 'result' was not  
minus 1 (\$FFFF hex). Branch conditions are not changed.

BRK                                    0A                                    (Break)

A 6502 BRK (break) instruction is executed. SWEET16  
may be reentered nondestructively at SW16D after correcting  
the stack pointer to its value prior executing the BRK.

RS                    OB            (Return from SWEET16 Subroutine)

RS terminates execution of a SWEET16 subroutine and returns to the SWEET16 calling program which resumes execution (in SWEET16 mode). R12, which is the SWEET16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BS    ea                    OC d (Branch to SWEET16 Subroutine)

A branch to the effective address (PC + 2 + d) is taken and execution is resumed in SWEET16 mode. The current PC is pushed onto a 'SWEET16 subroutine return address' stack whose pointer is R12, and R12 is incremented by 2. The carry is cleared and branch conditions set to indicate the current ACC contents.

Example (Calling a 'memory move' subroutine to move A034-A03B to 3000-3007)

300:	15 34 A0	SET	R5, A034	Init pointer 1.
303:	14 3B A0	SET	R4, A03B	Init limit 1.
306:	16 00 30	SET	R6, 3000	Init pointer 2.
309:	0C 15	BS	MOVE	Call move subroutine.
	.			
	.			
	.			
320:	45	MOVE	LD @R5	Move one
321:	56		ST @R6	byte.
322:	24		LD R4	
323:	D4	CPR	R5	Test if done.
324:	04 FA	BP	MOVE	Return.
326:	0B		RS	

## THEORY OF OPERATION

SWEET16 execution mode begins with a subroutine call to SW16. The user must insure that the 6502 is in hex mode upon entry. All 6502 registers are saved at this time, to be restored when a SWEET16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30 usec may be saved by entering at SW16 + 3. Because this might cause an inadvertant switch from hex to decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 6502 registers, SWEET16 initializes its PC (R15) with the subroutine return address off the 6502 stack. SWEET16's PC points to the location preceding the next instruction to be executed. Following the subroutine call are 1-, 2-, and 3-byte SWEET16 instructions, stored in ascending memory locations like 6502 instructions. The main loop at SW16B repeatedly calls the 'execute instruction' routine at SW16C which examines one opcode for type and branches to the appropriate subroutine to execute it.

Subroutine SW16C increments the PC (R15) and fetches the next opcode which is either a register op of the form OP REG with OP between 1 and 15 or a non-register op of the form 0 OP with OP between 0 and 13. Assuming a register op, the register specification is doubled to account for the 2-byte SWEET16 registers and placed in the X-Reg for indexing. Then the instruction type is determined. Register ops place the doubled register specification in the high order byte of R14 indicating

the 'prior result register' to subsequent branch instructions. Non-register ops treat the register specification (right-hand half-byte) as their opcode, increment the SWEET16 PC to point at the displacement byte of branch instructions, load the A-Reg with the 'prior result register' index for branch condition testing, and clear the Y-Reg.

#### WHEN IS AN RTS REALLY A JSR?

Each instruction type has a corresponding subroutine. The subroutine entry points are stored in a table which is directly indexed into by the opcode. By assigning all the entries to a common page only a single byte of address need be stored per routine. The 6502 indirect jump might have been used as follows to transfer control to the appropriate subroutine.

```
LDA #ADRH      High-order address byte.
STA IND+1
LDA OPTBL,X    Low-order byte.
STA IND
JMP (IND)
```

To save code the subroutine entry address (minus 1) is pushed onto the stack, high-order byte first. A 6502 RTS (ReTurn from Subroutine) is used to pop the address off the stack and into the 6502 PC (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction!

### OPCODE SUBROUTINES

The register op routines make use of the 6502 'zero page indexed by X' and 'indexed by X indirect' addressing modes to access the specified registers and indirect data. The 'result' of most register ops is left in the specified register and can be sensed by subsequent branch instructions since the register specification is saved in the high-order byte of R14. This specification is changed to indicate R0 (ACC) for ADD and SUB instructions and R13 for the CPR (compare) instruction.

Normally the high-order R14 byte holds the 'prior result register' index times 2 to account for the 2-byte SWEET16 registers and thus the LSB is zero. If ADD, SUB, or CPR instructions generate carries, then this index is incremented, setting the LSB.

The SET instruction increments the PC twice, picking up data bytes in the specified register. In accordance with 6502 convention, the low-order data byte precedes the high-order byte.

Most SWEET16 nonregister ops are relative branches. The corresponding subroutines determine whether or not the 'prior result' meets the specified branch condition and if so update the SWEET16 PC by adding the displacement value (-128 to +127 bytes).

The RTN op restores the 6502 register contents, pops the subroutine return stack and jumps indirect through the SWEET16 PC. This transfers control to the 6502, at the instruction immediately following the RTN instruction.

The BK op actually executes a 6502 break instruction (BRK), transferring control to the interrupt handler.

Any number of subroutine levels may be implemented within SWEET16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the SWEET16 subroutine capability is used since it is utilized as the automatic subroutine return stack pointer.

#### MEMORY ALLOCATION

The only storage that must be allocated for SWEET16 variables are 32 consecutive locations in page zero for the SWEET16 registers, four locations to save the 6502 register contents, and a few levels of the 6502 subroutine return address stack. If you don't need to preserve the 6502 register contents, delete the SAVE and RESTORE subroutines and the corresponding subroutine calls. This will free the four page zero locations ASAV, XSAV, YSAV, and PSAV.

#### USER MODIFICATIONS

You may wish to add some of your own instructions to this implementation of SWEET16. If you use the unassigned opcodes \$OE and \$OF, remember that SWEET16 treats these as 2-byte instructions. You may wish to handle the break instruction as a SWEET16 call, saving two bytes of code each time you transfer into SWEET16



mode. Or you may wish to use the SWEET16 BK (Break) op as a 'CHAROUT' call in the interrupt handler. You can perform absolute jumps within SWEET16 by loading teh ACC (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

SWEET16 INTERPRETER

PAGE: 1

45 P. M. , 10/3/1977

```

1 *****
2 *
3 *   APPLE-II PSEUDO *
4 *   MACHINE INTERPRETER *
5 *
6 *   COPYRIGHT 1977 *
7 *   APPLE COMPUTER INC *
8 *
9 *   ALL RIGHTS RESERVED *
10 *
11 *   S. WOZNIAK *
12 *
13 *****
14 TITLE "SWEET16 INTERPRETER"
15 ROL      EPZ  $0
16 ROH      EPZ  $1
17 R14H     EPZ  $1D
18 R15L     EPZ  $1E
19 R15H     EPZ  $1F
20 S16PAG   EQU  $F7
21 SAVE     EQU  $FF4A
22 RESTORE  EQU  $FF3F
23          ORG  $F689
F689: 20 4A FF 24 SW16 JSR  SAVE
F68C: 68          25 PLA
F68D: 65 1E      26 STA  R15L
F68F: 68          27 PLA
F690: 35 1F      28 STA  R15H
F692: 20 98 F6 29 SW16B JSR  SW16C
F695: 4C 92 F6 30 JMP  SW16B
F698: E6 1E      31 SW16C INC  R15L
F69A: D0 02      32 BNE  SW16D
F69C: E6 1F      33 INC  R15H
F69E: A9 F7      34 SW16D LDA  #S16PAG
F6A0: 48          35 PHA
F6A1: A0 00      36 LDY  #$0
F6A3: B1 1E      37 LDA  (R15L),Y
F6A5: 29 0F      38 AND  #$F
F6A7: 0A          39 ASL  A
F6A8: AA          40 TAX
F6A9: 4A          41 LSR  A
F6AA: 51 1E      42 EOR  (R15L),Y
F6AC: F0 0B      43 BED  TOBR
F6AE: 86 1D      44 STX  R14H
F6B0: 4A          45 LSR  A
F6B1: 4A          46 LSR  A
F6B2: 4A          47 LSR  A
F6B3: AB          48 TAY
F6B4: B9 E1 F6 49 LDA  OPTBL-2,Y
F6B7: 40          50 PHA
F6B8: 60          51 RTS
F6B9: E6 1E      52 TOBR INC  R15L
F6BB: D0 02      53 BNE  TOBR2
F6BD: E6 1F      54 INC  R15H

```

```

PRESERVE 6502 REG CONTENTS
INIT SWEET16 PC
FROM RETURN
ADDRESS
INTERPRET AND EXECUTE
ONE SWEET16 INSTR.
INCR SWEET16 PC FOR FEICH
PUSH ON STACK FOR RTS
FEICH INSTR
MASK REG SPECIFICATION
DOUBLE FOR 2-BYTE REG'S
TO X-REG FOR INDEXING
NOW HAVE OPCODE
IF ZERO THEN NON-REG OP
INDICATE PRIOR RESULT REG
OPCODE*2 TO LSB'S
TO Y-REG FOR INDEXING
LOW-ORDER ADR BYTE
ONIO STACK
GOTO REG-OP ROUTINE
INCR PC

```

SWEET16 INTERPRETER

PAGE: 2

1:45 P.M., 10/3/1977

F6BF:	BD E4 F6 55	T0BR2	LDA	BRTBL,X	LOW-ORDER ADR BYTE
F6C2:	48	56	PHA		ONTO STACK FOR NON-REG OP
F6C3:	A5 1D	57	LDA	R14H	'PRIOR RESULT REG' INDEX
F6C5:	4A	58	LSR	A	PREPARE CARRY FOR BC, BNC.
F6C6:	60	59	RTS		GOTO NON-REG OP ROUTINE
F6C7:	68	60	RTNZ	PLA	POP RETURN ADDRESS
F6C8:	68	61	PLA		
F6C9:	20 3F FF	62	JSR	RESTORE	RESTORE 6502 REG CONTENTS
F6CC:	6C 1E 00	63	JMP	(R15L)	RETURN TO 6502 CODE VIA PC
F6CF:	B1 1E	64	SETZ	LDA (R15L),Y	HIGH-ORDER BYTE OF CONST
F6D1:	95 01	65	STA	ROH,X	
F6D3:	88	66	DEY		
F6D4:	B1 1E	67	LDA	(R15L),Y	LOW-ORDER BYTE OF CONSTANT
F6D6:	95 00	68	STA	ROL,X	
F6D8:	98	69	TYA		Y-REG CONTAINS 1
F6D9:	38	70	SEC		
F6DA:	65 1E	71	ADC	R15L	ADD 2 TO PC
F6DC:	85 1E	72	STA	R15L	
F6DE:	90 02	73	BCC	SETZ	
F6E0:	E6 1F	74	INC	R15H	
F6E2:	60	75	SETZ	RTS	
F6E3:	02	76	OPTBL	DFB SET-1	(1X)
F6E4:	F9	77	BRTBL	DFB RTN-1	(0)
F6E5:	04	78		DFB LD-1	(2X)
F6E6:	9D	79		DFB BR-1	(1)
F6E7:	0D	80		DFB ST-1	(3X)
F6E8:	9E	81		DFB BNC-1	(2)
F6E9:	25	82		DFB LDAT-1	(4X)
F6EA:	AF	83		DFB BC-1	(3)
F6EB:	16	84		DFB STAT-1	(5X)
F6EC:	B2	85		DFB BP-1	(4)
F6ED:	47	86		DFB LDDAT-1	(6X)
F6EE:	B9	87		DFB BM-1	(5)
F6EF:	51	88		DFB STDAT-1	(7X)
F6F0:	00	89		DFB BZ-1	(6)
F6F1:	2F	90		DFB POP-1	(8X)
F6F2:	09	91		DFB BNZ-1	(7)
F6F3:	5B	92		DFB STPAT-1	(9X)
F6F4:	D2	93		DFB BM1-1	(8)
F6F5:	85	94		DFB ADD-1	(AX)
F6F6:	DD	95		DFB BNM1-1	(9)
F6F7:	6E	96		DFB SUB-1	(EX)
F6F8:	05	97		DFB BK-1	(A)
F6F9:	33	98		DFB POPD-1	(CX)
F6FA:	E8	99		DFB RS-1	(B)
F6FB:	70	100		DFB CPR-1	(DX)
F6FC:	93	101		DFB BS-1	(C)
F6FD:	1E	102		DFB INR-1	(EX)
F6FE:	E7	103		DFB NUL-1	(D)
F6FF:	65	104		DFB DCR-1	(FX)
F700:	E7	105		DFB NUL-1	(E)
F701:	E7	106		DFB NUL-1	(UNUSED)
F702:	E7	107		DFB NUL-1	(F)
F703:	10 CA	108	SET	BPL SETZ	ALWAYS TAKEN

## SWSET16 INTERPRETER

1:45 P. M., 10/3/1977

PAGE: 3

```

F705: 85 00 109 LD LDA ROL, X
      110 BK EQU #-1
F707: 85 00 111 STA ROL
F709: 85 01 112 LDA ROH, X MOVE RX TO RO
F70B: 85 01 113 STA ROH
F70D: 60 114 RTS
F70E: A5 00 115 ST LDA ROL
F710: 95 00 116 STA ROL, X MOVE RO TO RX
F712: A5 01 117 LDA ROH
F714: 95 01 118 STA ROH, X
F716: 60 119 RTS
F717: A5 00 120 STAT LDA ROL
F719: 81 00 121 STAT2 STA (ROL, X) STORE BYTE INDIRECT
F71B: A0 00 122 LDY #$0
F71D: 84 1D 123 STAT3 STY R14H INDICATE RO IS RESULT REG
F71F: F6 00 124 INK INC ROL, X
F721: D0 02 125 BNE INK2 INCR RX
F723: F6 01 126 INC ROH, X
F725: 60 127 INK2 RTS
F726: A1 00 128 LDAT LDA (ROL, X) LOAD INDIRECT (RX)
F728: 85 00 129 STA ROL TO RO
F72A: A0 00 130 LDY #$0
F72C: 84 01 131 STY ROH ZERO HIGH-ORDER RO BYTE
F72E: F0 ED 132 BEQ STAT3 ALWAYS TAKEN
F730: A0 00 133 POP LDY #$0 HIGH ORDER BYTE = 0
F732: F0 06 134 BEQ POP2 ALWAYS TAKEN
F734: 20 66 F7 135 POPD JSR DCR DECR RX
F737: A1 00 136 LDA (ROL, X) POP HIGH-ORDER BYTE @RX
F739: A8 137 TAY SAVE IN Y-REG
F73A: 20 66 F7 138 POP2 JSR DCR DECR RX
F73D: A1 00 139 LDA (ROL, X) LOW-ORDER BYTE
F73F: 85 00 140 STA ROL TO RO
F741: 84 01 141 STY ROH
F743: A0 00 142 POP3 LDY #$0 INDICATE RO AS LAST
F745: 84 1D 143 STY R14H RESULT REG
F747: 60 144 RTS
F748: 20 26 F7 145 LDDAT JSR LDAT LOW BYTE TO RO, INCR RX
F74B: A1 00 146 LDA (ROL, X) HIGH-ORDER BYTE TO RO
F74D: 85 01 147 STA ROH
F74F: 4C 1F F7 148 JMP INK INCR RX
F752: 20 17 F7 149 STAT JSR STAT STORE INDIRECT LOW-ORDER
F755: A5 01 150 LDA ROH BYTE AND INCR RX. THEN
F757: 81 00 151 STA (ROL, X) STORE HIGH-ORDER BYTE.
F759: 4C 1F F7 152 JMP INK INCR RX AND RETURN
F75C: 20 66 F7 153 STAT JSR DCR DECR RX
F75E: A5 00 154 LDA ROL
F761: 81 00 155 STA (ROL, X) STORE RO LOW BYTE @RX
F763: 4C 43 F7 156 JMP POP3 INDICATE RO AS LAST RSLT RE
F766: 85 00 157 DCR LDA ROL, X
F768: D0 02 158 BNE DCR2 DECR RX
F76A: D6 01 159 DEC ROH, X
F76C: D6 00 160 DCR2 DEC ROL, X
F76E: 60 161 RTS
F76F: A0 00 162 SUB LDY #$0 RESULT TO RO

```

SWEET16 INTERPRETER

PAGE 4

1:45 P.M., 10/3/1977

F771:	38	163	CFR	SEC		NOTE Y-REG = 13*2 FOR CFR
F772:	A5 00	164		LDA	ROL, X	
F774:	F5 00	165		SBC	ROL, X	
F776:	99 00 00	166		STA	ROL, Y	RO-RX TO RY
F779:	A5 01	167		LDA	ROH	
F77B:	F5 01	168		SBC	ROH, X	
F77D:	99 01 00	169	SUB2	STA	ROH, Y	
F780:	98	170		TYA		LAST RESULT REG*2
F781:	69 00	171		ADC	#\$0	CARRY TO LSB
F783:	85 1D	172		STA	R14H	
F785:	60	173		RTS		
F786:	A5 00	174	ADD	LDA	ROL	
F78C:	75 00	175		ADC	ROL, X	
F78A:	85 00	176		STA	ROL	RO-RX TO RO
F78C:	A5 01	177		LDA	ROH	
F78E:	75 01	178		ADC	ROH, X	
F790:	A0 00	179		LDY	#\$0	RO FOR RESULT
F792:	F0 09	180		BEQ	SUB2	FINISH ADD
F794:	A5 1E	181	BS	LDA	R15L	NOTE X-REG IS 12*2!
F796:	20 19 F7	182		JSR	STAT2	PUSH LOW PC BYTE VIA R12
F799:	A5 1F	183		LDA	R15H	
F79B:	20 19 F7	184		JSR	STAT2	PUSH HIGH-ORDER PC BYTE
F79E:	10	185	BR	CLC		
F79F:	B0 0E	186	BNC	BCS	BNC2	NO CARRY TEST
F7A1:	B1 1E	187	BR1	LDA	(R15L), Y	DISPLACEMENT BYTE
F7A3:	10 01	188		BPL	BR2	
F7A5:	88	189		DEY		
F7A6:	65 1E	190	BR2	ADC	R15L	ADD TO PC
F7A8:	85 1E	191		STA	R15L	
F7AA:	98	192		TYA		
F7AB:	65 1F	193		ADC	R15H	
F7AD:	85 1F	194		STA	R15H	
F7AF:	60	195	BNC2	RTS		
F7B0:	B0 EC	196	BC	BCS	BR	
F7B2:	60	197		RTS		
F7B3:	0A	198	BP	ASL	A	DOUBLE RESULT-REG INDEX
F7B4:	AA	199		TAX		TO X-REG FOR INDEXING
F7B5:	B5 01	200		LDA	ROH, X	TEST FOR PLUS
F7B7:	10 ES	201		BPL	BR1	BRANCH IF SO
F7B9:	60	202		RTS		
F7BA:	0A	203	BM	ASL	A	DOUBLE RESULT-REG INDEX
F7BB:	AA	204		TAX		
F7BC:	B5 01	205		LDA	ROH, X	TEST FOR MINUS
F7BE:	30 E1	206		BMI	BR1	
F7C0:	60	207		RTS		
F7C1:	0A	208	BZ	ASL	A	DOUBLE RESULT-REG INDEX
F7C2:	AA	209		TAX		
F7C3:	B5 00	210		LDA	ROL, X	TEST FOR ZERO
F7C5:	15 01	211		ORA	ROH, X	(BOTH BYTES)
F7C7:	F0 D8	212		BEQ	BR1	BRANCH IF SO
F7C9:	60	213		RTS		
F7CA:	0A	214	BNZ	ASL	A	DOUBLE RESULT-REG INDEX
F7CB:	AA	215		TAX		
F7CD:	B5 00	216		LDA	ROL, X	TEST FOR NONZERO

SWEET16 INTERPRETER

PAGE: 5

1:45 P. M., 10/3/1977

```

F7DE: 15 01 217      ORA  R0H, X      (BOTH BYTES)
F7DD: 00 0F 218      BNE  BR1        BRANCH IF SO
F7DC: 60           219      RTS
F7DB: 0A           220      ASL  A          DOUBLE RESULT-REG INDEX
F7DA: AA           221      TAX
F7D9: B5 00       222      LDA  R0L, X    CHECK BOTH BYTES
F7D7: 35 01       223      AND  R0H, X    FOR $FF (MINUS 1)
F7D9: 49 FF       224      EOR  #$FF
F7DB: F0 C4       225      BEQ  BR1        BRANCH IF SO
F7DD: 60           226      RTS
F7DC: 0A           227      ASL  A          DOUBLE RESULT-REG INDEX
F7DF: AA           228      TAX
F7E0: B5 00       229      LDA  R0L, X
F7E2: 35 01       230      AND  R0H, X    CHK BOTH BYTES FOR NO $FF
F7E4: 49 FF       231      EOR  #$FF
F7E6: D0 B9       232      BNE  BR1        BRANCH IF NOT MINUS 1
F7E8: 60           233      RTS
F7E9: A2 18       234      LDX  #$18      12*2 FOR R12 AS STK PNTR
F7EB: 20 66 F7    235      JSR  DCR        DECR STACK POINTER
F7EE: A1 00       236      LDA  (R0L, X)  POP HIGH RETURN ADR TO PC
F7F0: 85 1F       237      STA  R15H
F7F2: 20 66 F7    238      JSR  DCR        SAME FOR LOW-ORDER BYTE
F7F5: A1 00       239      LDA  (R0L, X)
F7F7: 85 1E       240      STA  R15L
F7F9: 60           241      RTS
F7FA: 4C C7 F6    242      JMP  RTNZ
*****SUCCESSFUL ASSEMBLY: NO ERRORS
    
```

CROSS-REFERENCE:		SWEET16 INTERPRETER											
ADD	F736	0094											
BC	F7B0	0083											
BK	F706	0097											
BM	F7EA	0087											
BM1	F7D3	0093											
BNC	F79F	0081											
BNC2	F7AF	0186											
BNM1	F7DE	0095											
BNZ	F7CA	0091											
BP	F7B3	0085											
BR	F79E	0079	0196										
BR1	F7A1	0201	0206	0212	0218	0225	0232						
BR2	F7A6	0188											
BXTBL	F6E4	0055											
BS	F794	0101											
BZ	F7C1	0089											
CBR	F771	0100											
DCR	F766	0104	0135	0138	0153	0235	0238						
DCR2	F76C	0158											
INR	F71F	0102	0148	0152									
INR2	F725	0125											
LD	F705	0078											
LDDAT	F726	0082	0145										
LDDAT	F748	0086											
NULL	F7E8	0103	0105	0106	0107								
OPTBL	F6E3	0049											
POP	F730	0090											
POP2	F73A	0134											
POP3	F743	0156											
POPD	F734	0098											
ROL	0001(Z)	0065	0112	0113	0117	0118	0126	0131	0141	0147	0150	0159	
		0167	0168	0169	0177	0178	0200	0205	0211	0217	0223	0230	
ROL	0000(Z)	0068	0109	0111	0115	0116	0120	0121	0124	0128	0129	0136	
		0139	0140	0146	0151	0154	0155	0157	0160	0164	0165	0166	
		0174	0175	0176	0210	0216	0222	0229	0236	0239			
R14H	001D(Z)	0044	0057	0123	0143	0172							
R15H	001F(Z)	0028	0033	0054	0074	0183	0193	0194	0237				
R15L	001E	0026	0031	0037	0042	0052	0063	0064	0067	0071	0072	0181	
		0187	0190	0191	0240								
RESTORE	F73F	0062											
RS	F7E9	0099											
RTN	F7FA	0077											
RTNZ	F6C7	0242											
S16PAG	00F7	0034											
SAVE	F74A	0024											
SET	F703	0076											
SLT2	F6E2	0073											
SELZ	F6CF	0108											
ST	F70E	0080											
STAT	F717	0084	0149										
STAT2	F719	0182	0184										
STAT3	F71D	0132											
SUDDAT	F752	0088											
SUDDAT	F75C	0092											
SUB	F76F	0096											
SUB2	F77D	0180											
SW16	F6B9												
SW16B	F6B2	0030											
SW16C	F6B3	0029											
SW16D	F6B5	0032											

CROSS REFERENCE	ADDRESS	INTERPRETER
T0BR	F6B9	0043
T0BR2	F6BF	0053



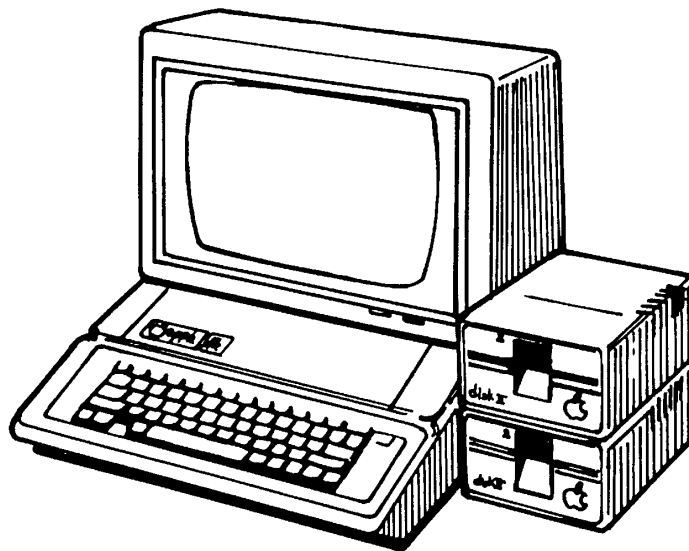
# The Woz Wonderbook

DOCUMENT

Apple-II

6502 Code Relocation Program

14 November 1977



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

A 6502  
CODE RELOCATION  
PROGRAM  
for the  
APPLE - II COMPUTER

S. Wozniak (WOZ)  
November 14, 1977

APPLE-II MACHINE CODE RELOCATION PROGRAM

Quite frequently I have encountered situations calling for relocation of machine language (not BASIC) programs on my 6502-based APPLE-II computer. Relocation means that the new version must run properly from different memory locations than the original. Because of the relative branch instruction, certain small 6502 programs need simply be moved and not altered. Others require only minor hand modification, which is simplified on the APPLE-II by the built-in disassembler which pinpoints absolute memory reference instructions such as JMPs and JSRs. However, most of the situations which I have encountered involved rather lengthy programs containing multiple data segments interspersed with code. For example, I once spent over an hour to hand-relocate the 8K byte APPLE8II monitor and BASIC to run from RAM addresses and at least one error probably went by undetected. That relocation can now be accomplished in a couple minutes using the relocation program described herein.

The following situations call for program relocation:

- (1) Two programs which were written to run in identical locations must now reside and run in memory concurrently.
- (2) A program currently runs from ROM. In order to modify its operation experimentally, a version must be generated which runs from RAM (different addresses).
- (3) A program currently running in RAM must be converted to run from EPROM or ROM addresses.
- (4) A program currently running on a 16K machine must be relocated in order to run on a 4K machine. Furthermore, the relocation may have to be performed on the smaller machine.
- (5) Due to memory mapping differences, a program running on an APPLE-I (or other 6502 based) computer falls into unusable address space on an APPLE-II (or other) computer.
- (6) Due to operating system variable assignment differences either the page-zero or non-page-zero variable allocation for a specific program may have to be modified when moving the program from one make of computer to another.
- (7) A program exists as several chunks strewn about memory which must be combined in a single, contiguous block.

- (8) A program has outgrown the available memory space and must be relocated to a larger 'free' space.
- (9) A program insertion or deletion requires a chunk of the program to move a few bytes up or down.
- (10) On a whim, the user wishes to move a program.

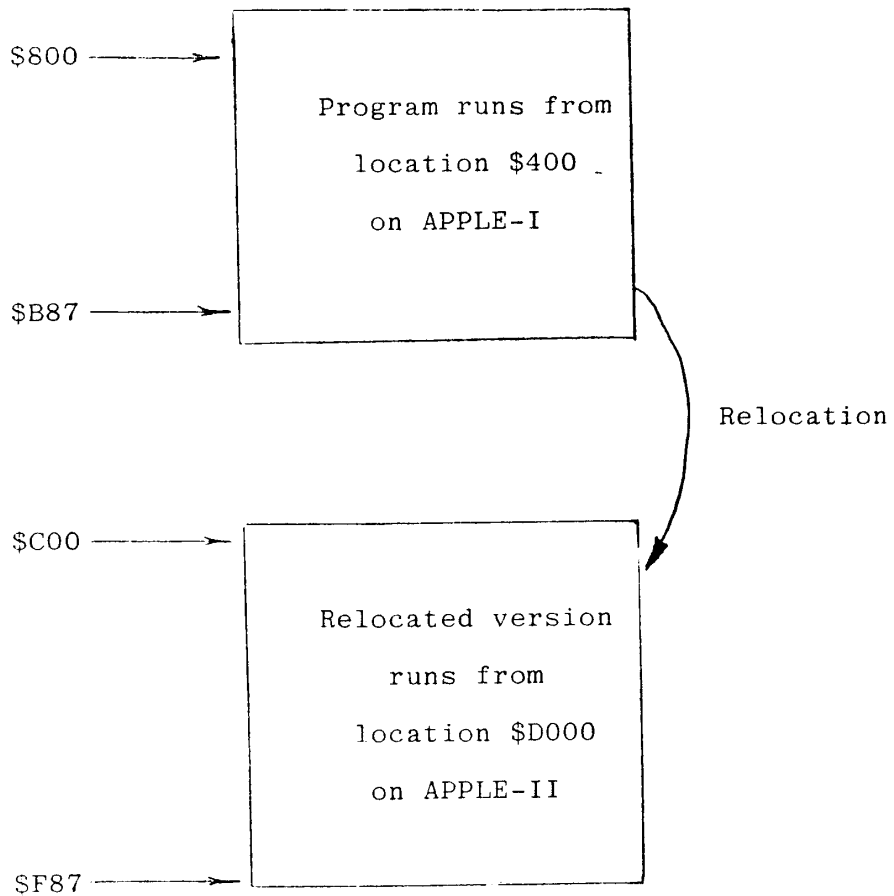
## PROGRAM MODEL

It is easy to visualize relocation as taking a program which resides and runs in a 'source block' of memory and creating a modified version in a 'destination block' which runs properly. This model dictates that the relocation must be performed in an environment in which the program may in fact reside in both blocks. In many cases, the relocation is being performed because this is impossible. For example, a program written to begin at location \$400 on an APPLE-I (\$ stands for hex) falls in the APPLE-II screen memory range. It must be loaded elsewhere on the APPLE-II prior to relocation.

A more versatile program model is as follows. A program or section of a program runs in a memory range termed the 'source block' and resides in a range termed the 'source segments'. Thus a program written to run at location \$400 may reside at location \$800. The program is to be relocated so that it will run in a range termed the 'destination block' although it will reside in a range termed 'destination segments' (not necessarily the same). Thus a program may be relocated such that it will run from location \$D000 (a ROM address) yet reside beginning at location \$C00 prior to being saved on tape or used to burn EPROMs (obviously, the relocated program cannot immediately reside at locations reserved for ROM). In some cases the source and destination segments may overlap.

BLOCKS AND SEGMENTS EXAMPLE

Location during  
Relocation



SOURCE BLOCK: \$400-\$787

DEST BLOCK: \$D000-\$D387

SOURCE SEGMENTS: \$800-\$B87

DEST SEGMENTS: \$C00-\$F87



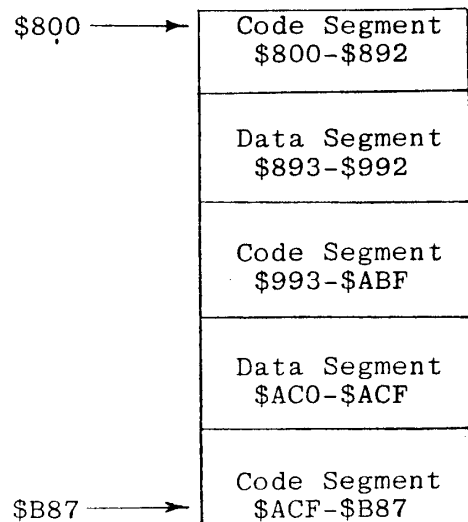
THE RELOCATION ALGORITHM

- (1) Set SOURCE PTR to beginning of source segment and DEST PTR to beginning of destination segment.
- (2) Copy 3 bytes from source segment (using SOURCE PTR) to temp INST area.
- (3) Determine instruction length from opcode (1, 2, or 3 byte).
- (4) If two byte instruction with non-zero-page addressing mode (immediate or relative) then go to (7).
- (5) If two byte instruction then clear 3rd byte so address field is 0-255 (zero page).
- (6) If address field (2nd and 3rd bytes of INST area) falls within source block, then substitute
$$\text{ADR} - \text{SOURCE BLOCK BEGIN} + \text{DEST BLOCK BEGIN}$$
- (7) Move 'length' bytes from INST area to dest segment (using DEST PTR). Update SOURCE and DEST PTRs by length.
- (8) If SOURCE PTR is less than or equal to SOURCE SEGMENT END then goto (2), else done.

## DATA SEGMENTS

The problem with relocating a large program all at once is that data (tables, text, etc.) may be interspersed throughout the code. Thus data may be 'relocated' as though it were code or might cause some code not to be relocated due to boundary uncertainty introduced when the data takes on the multi-byte attribute of code. This problem is circumvented by considering the 'source segments' and 'destination segments' sections to contain both code and data segments.

## CODE AND DATA SEGMENTS EXAMPLE



The source code segments are relocated to the 'destination segments' area and the source data segments are moved. Note that several commands will be necessary to accomplish the complete relocation.

USAGE

1. Load RELOC by hand or off tape into memory locations \$3A6-\$3FA. Note that locations \$3FB-\$3FF are not disturbed by tape load versions to insure that the APPLE-II interrupt vectors are not clobbered. The monitor user function Y<sup>C</sup> (Control-Y) will now call RELOC as a subroutine at location \$3F8.
2. Load the source program into the 'source segments' area of memory if it is not already there. Note that this need not be where the program normally runs.
3. Specify the source and destination block parameters, remembering that the blocks are the locations that the program normally runs from, not the locations occupied by the source and destination segments during the relocation. If only a portion of a program is to be relocated then that portion alone is specified as the block.

\* DEST BLOCK BEG < SOURCE BLOCK BEG . END Y<sup>C</sup> \*

Note that the syntax of this command closely resembles that of the MONITOR 'MOVE' command. The initial '\*' is generated by the MONITOR, not typed by the user.

4. Move all data segments and relocate all code segments in sequential (increasing address) order.

First Segment (if CODE)

\* DEST SEGMENT BEG < SOURCE SEGMENT BEG . END Y<sup>C</sup>

First Segment (if DATA)

\* DEST SEGMENT BEG < SOURCE SEGMENT BEG . END M

Subsequent segments (if CODE)

\* . SOURCE SEGMENT END Y<sup>C</sup> (Relocation)

Subsequent segments (if DATA)

\* . SOURCE SEGMENT END M (Move)

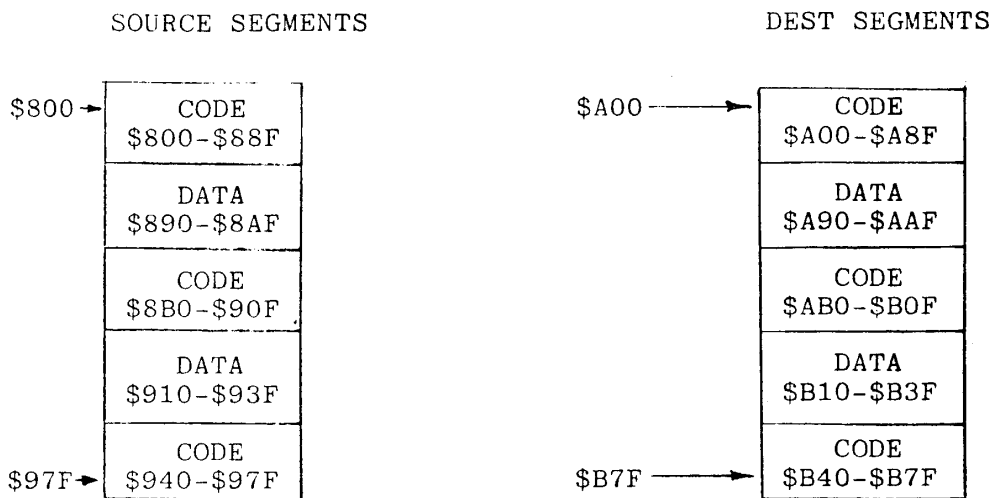
Note that it is wise to prepare a list of segments (code and data) prior to relocation.

If the relocation is performed 'in place' (SOURCE and DEST SEGMENTS reside in identical locations) then the SOURCE SEGMENT BEG parameter may be omitted from the first segment relocate (or move).

EXAMPLES

1. Straightforward Relocation

Program A resides and runs in locations \$800-\$97F. The relocated version will reside and run in locations \$A00-\$B7F.



SOURCE BLOCK \$800-\$97F

DEST BLOCK \$A00-\$B7F

SOURCE SEGMENTS \$800-\$97F

DEST SEGMENTS \$A00-\$B7F

(a) Load RELOC

(b) Define blocks

\* A00 < 800 . 97F Y<sup>C</sup> \*

(c) Relocate first segment (code).

\* A00 < 800 . 88F Y<sup>C</sup>

(d) Move and relocate subsequent segments in order.

- \* . 8AF M (data)
- \* . 90F Y<sup>C</sup> (code)
- \* . 93F M (data)
- \* . 97F Y<sup>C</sup> (code)

Note that step (d) illustrates abbreviated versions of the following commands:

- \* A90 < 890 . 8AF M (data)
- \* AB0 < 8B0 . 90F Y<sup>C</sup> (code)
- \* B10 < 910 . 93F M (data)
- \* B40 < 940 . 97F Y<sup>C</sup> (code)

## 2. Index into block

Assume that the program of example 1 uses an indexed reference into the data segment at \$890 as follows:

```
LDA 7B0,X
```

The X-REG is presumed to contain \$E0-\$FF. Because \$7B0 is outside the source block, it will not be relocated. This may be handled in one of two ways.

- (a) The exception is fixed by hand, or
- (b) The block specifications begin one page lower than the addresses at which the original and relocated programs begin to account for all such 'early references'. In step (b) of example (1) change to:

```
* 900 < 700 . 97F YC *
```

Note that program references to the 'prior page' (in this case the \$7XX page) which are not intended to be relocated will be.

### 3. Immediate Address References

Assume that the program of example (1) has an immediate reference which is an address. For example,

```
LDA  #3F
STA  LOC0
LDA  #08
STA  LOC1
JMP  (LOC0)
```

In this example, the LDA #08 will not be changed during relocation and the user will have to hand-modify it to 0A.

### 4. User function ( $Y^C$ ) programs

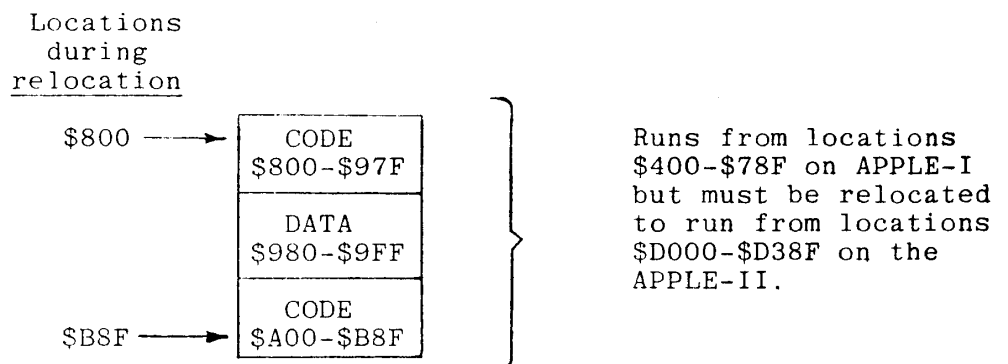
Relocating programs such as RELOC introduces another irregularity. Because RELOC uses the MONITOR user function command ( $Y^C$ ) its entry point must remain fixed at 3F8. The rest of RELOC may be relocated anywhere in memory (which is trivial since RELOC contains no absolute memory references other than the JMP at 3F8). The user must leave the JMP at 3F8 undisturbed or find some way other than  $Y^C$  to pass parameters.



5. Unusable block ranges

A program was written to run from locations \$400-\$78F on an APPLE-I. A version which will run in ROM locations \$D000-\$D38F must be generated. The source (and destination) segments may reside in locations \$800-\$B8F on the APPLE-II where relocation is performed.

SEGMENTS, SOURCE AND DEST



SOURCE BLOCK \$400-\$78F

DEST BLOCK \$D000-\$D38F

SOURCE SEGMENTS \$800-\$B8F

DEST SEGMENTS \$800-\$B8F

- (a) Load RELOC
- (b) Load original program into locations \$800-\$B8F (despite the fact that it doesn't run there).
- (c) Specify block parameters (i.e. where the original and relocated versions will run)

\* D000 < 400 . 78F Y<sup>C</sup> \*

(d) Move and relocate all segments in order.

```
* 800 < 800 . 97F YC (first segment, code)
* . 9FF M (data)
* . B8F YC (code)
```

Note that because the relocation is done 'in place' the SOURCE SEGMENT BEG parameter is the same as the DEST SEGMENT BEG parameter (\$800) and need not be specified. The initial segment relocation command may be abbreviated as follows:

```
* 800 <. 97F YC
```

6. The program of example (1) need not be relocated but the page zero variable allocation is from \$30 to \$3F. Because these locations are reserved for the APPLE-II system monitor, the allocation must be changed to locations \$80-\$8F. The source and destination blocks are thus not the program but rather the variable area.

```
SOURCE BLOCK $20-$2F          DEST BLOCK $80-$8F
SOURCE SEGMENTS $800-$97F    DEST SEGMENTS $800-$97F
```

(a) Load RELOC

(b) Define blocks

```
* 80 < 20.2F YC *
```

(c) Relocate code segments and move data segments in place.

```
* 800 <.88F YC (code)
* . 8AF M (data)
* . 90F YC (code)
* . 93F M (data)
* . 97F YC (code)
```



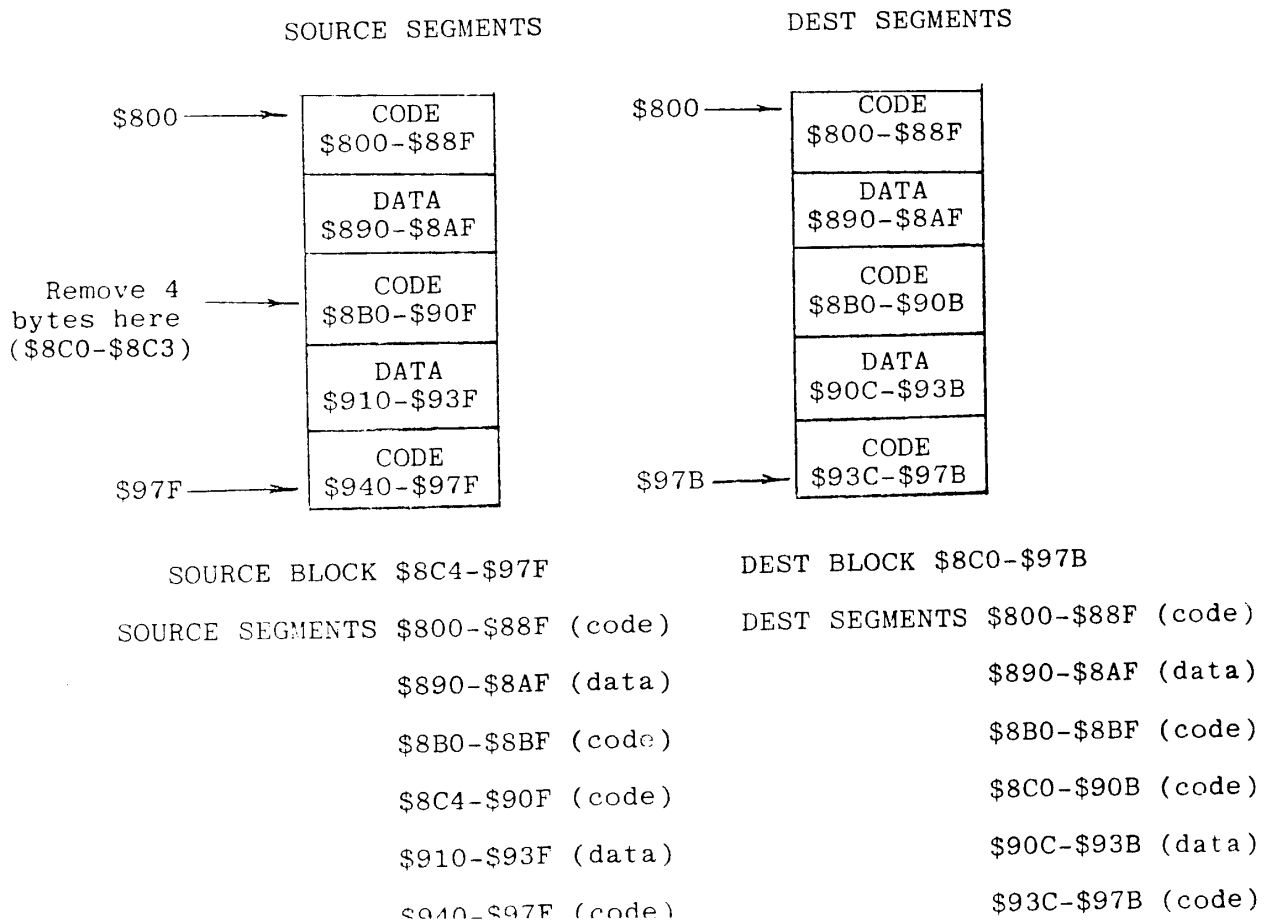
(c) Relocate each of the two programs individually. Program A must be relocated even though it does not move.

- \* 800 < . 8A6 Y<sup>C</sup> (program A, 'in place')
- \* 8A6 < 900 . 9F1 Y<sup>C</sup> (program B, not 'in place')

Note that any data segments within the two programs would necessitate additional relocation and move commands.

8. Code deletion.

4 bytes of code are to be removed from within a program and the program is to contract accordingly.



(a) Load RELOC

(b) Define blocks

```
* 8C0 < 8C4 . 97F YC *
```

(c) Relocate code segments and move data segments in ascending address sequence.

```
* 800 <. 88F YC (code, 'in place')
* . 8AF M (data)
* . 8BF YC (code)
* 8C0 < 8C4 . 90F YC (code, not 'in place')
* . 93F M (data)
* . 97F YC (code)
```

(d) Relative branches crossing the deletion boundary will be incorrect since the relocation process does not modify them (only zero-page and absolute memory references). The user must patch these by hand.

9. Relocating the APPLE-II monitor (\$F800-\$FFFF) to run in RAM  
 (\$800-\$FFF)

SOURCE BLOCK \$F700-\$FFFF                      DEST BLOCK \$700-\$FFF  
 (see example (2) )

SOURCE SEGMENTS \$F800-\$F961 (code)	DEST SEGMENTS \$800-\$961 (code)
\$F962-\$FA42 (data)	\$962-\$A42 (data)
\$FA43-\$FB18 (code)	\$A43-B18 (code)
\$FB19-\$FB1D (data)	\$B19-\$B1D (data)
\$FB1E-\$FFCB (code)	\$B1E-\$FCB (code)
\$FFCC-\$FFFF (data)	\$FCC-\$FFF (data)

IMMEDIATE ADDRESS REFS (see example (3) )

\$FFBF

\$FEA8

(more if not relocating to page boundary)

(a) Load RELOC

(b) Block parameters

\* 700 < F700 . FFFF Y<sup>C</sup> \*

(c) Segments

\* 800 < F800 . F961 Y<sup>C</sup> (first segment, code)  
 \* . FA42 M (data)  
 \* . FB18 Y<sup>C</sup> (code)  
 \* . FB1D M (data)  
 \* . FFCB Y<sup>C</sup> (code)  
 \* . FFFF M (data)

(c), Immediate address references

\* FBF ; E (was \$FE)

\* EA8 ; E (was \$FE)

## OTHER 6502 SYSTEMS

The following details illustrate features specific to the APPLE-II which are used by RELOC. If adapted to other systems, the convenient and flexible parameter passing capability of the APPLE-II monitor may be sacrificed.

1. The APPLE-II monitor command

\*  $A_4 < A_1 . A_2 Y^C$  ( $A_1$ ,  $A_2$ , and  $A_4$  are addresses)  
 vectors to location \$3F8 with the value  $A_1$  in locations \$3C (low) and \$3D (high),  $A_2$  in locations \$3E (low) and \$3F (high), and  $A_4$  in locations \$42 (low) and \$43 (high). Location \$34 (YSAV) holds an index to the next character of the command buffer (after the  $Y^C$ ). The command buffer (IN) begins at \$200.

2. If  $Y^C$  is followed by an '\*' then the block parameters are simply preserved as follows:

<u>Parameter</u>	<u>Preserved at</u>	<u>SWEET16 Reg Name</u>
DEST BLOCK BEG	\$8, \$9	TOBEG
SOURCE BLOCK BEG	\$2, \$3	FRMBEG
SOURCE BLOCK END	\$4, \$5	FRMEND

3. If  $Y^C$  is not followed by and '\*' then a segment relocation is initiated at RELOC2 (\$3BB). Throughout,  $A_1$  (\$3C, \$3D) is the source segment pointer and  $A_4$  (\$42, \$43) is the destination segment pointer.



4. INSDS2 is an APPLE-II monitor subroutine which determines the length of a 6502 instruction in the variable LENGTH (location \$2F) given the opcode in the A-REG.

<u>Instruction type</u>	<u>LENGTH</u>
Invalid	0
1 byte	0
2 byte	1
3 byte	2

5. The code from XLATE to SW16RT (\$3D9-\$3E6) uses the APPLE-II 16-bit interpretive machine, SWEET16. The target address of the 6502 instruction being relocated (locations \$C low and \$D high) occupies the SWEET16 register named ADR. If ADR is between FRMBEG and FRMEND (inclusive) then it is replaced by ADR - FRMBEG + TOBEG.
6. NXTA4 is an APPLE-II monitor subroutine which increments A1 (source segment index) and A4 (destination segment index). If A1 exceeds A2 (source segment end) then the carry is set, otherwise it is cleared.

6502 RELOCATION SUBROUTINE

4:36 P.M., 11/10/1977

PAGE: 1

```
1 TITLE '6502 RELOCATION SUBROUTINE'
2 *****
3 *
4 * 6502 RELOCATION *
5 * SUBROUTINE *
6 * *
7 * 1. DEFINE BLOCKS *
8 * *A4<A1.A2 ^Y *
9 * (^Y IS CRTL-Y) *
10 * *
11 * 2. FIRST SEG *
12 * *A4<A1.A2 ^Y *
13 * (IF CODE) *
14 * *
15 * *A4<A1.A2 M *
16 * (IF MOVE) *
17 * *
18 * 3. SUBSEQUENT SEGS *
19 * *.A2 ^Y OR *.A2 M *
20 * *
21 * WOZ 11-10-77 *
22 * APPLE COMPUTER INC. *
23 * *
24 *****
25 PAGE
```

```
4:36 P.M., 11/10/1977          RELOCATION SUBR EQUATES          PAGE: 2
26      SUBTTL RELOCATION SUBR EQUATES
27      R1L          EPZ  $2          SWEET16 REG 1.
28      INST        EPZ  $B          3-BYTE INST FIELD.
29      LENGTH      EPZ  $2F        LENGTH CODE.
30      YSAV        EPZ  $34        CMND BUF POINTER.
31      A1L         EPZ  $3C        APPLE-II MON PARAM AREA.
32      A4L         EPZ  $42        APPLE-II MON PARAM REG 4
33      IN          EQU  $200       MON CMND BUF.
34      SW16        EQU  $F689      SWEET16 ENTRY.
35      INSDS2      EQU  $F88E      DISASSEMBLER ENTRY.
36      NXTA4       EQU  $FCB4      POINTER INCR SUBR.
37      FRMBEG      EPZ  $1         SOURCE BLOCK BEGIN.
38      FRMEND      EPZ  $2         SOURCE BLOCK END.
39      TOBEG       EPZ  $4         DEST BLOCK BEGIN.
40      ADR         EPZ  $6         ADR PART OF INST.
41      PAGE
```

6502 RELOCATION SUBROUTINE

PAGE: 3

4:36 P.M., 11/10/1977

```

42      SUBTTL 6502 RELOCATION SUBROUTINE
43      ORG    $3A6
03A6:   A4 34   44      RELOC    LDY    YSAV      CMND BUF POINTER.
03A8:   B9 00 02 45      LDA    IN,Y      NEXT CMND CHAR.
03AB:   C9 AA   46      CMP    #$AA      '*'?
03AD:   D0 0C   47      BNE    RELOC2    NO, RELOC CODE SEG.
03AF:   E6 34   48      INC    YSAV      ADVANCE POINTER.
03B1:   A2 07   49      LDX    #$7
03B3:   B5 3C   50      INIT    LDA    A1L,X    MOVE BLOCK PARAMS
03B5:   95 02   51      STA    R1L,X    FROM APPLE-II MON
03B7:   CA     52      DEX
03B8:   10 F9   53      BPL    INIT      AREA TO SW16 AREA.
03BA:   60     54      RTS           R1=SOURCE BEG, R2=
03BB:   A0 02   55      RELOC2  LDY    #$2      SOURCE END, R4=DEST BEG
03BD:   B1 3C   56      GETINS  LDA    (A1L),Y  COPY 3 BYTES TO
03BF:   99 0B 00 57      STA    INST,Y   SW16 AREA.
03C2:   88     58      DEY
03C3:   10 F8   59      BPL    GETINS
03C5:   20 8E F8 60      JSR    INSDS2   CALCULATE LENGTH OF
03C8:   A6 2F   61      LDX    LENGTH   INST FROM OPCODE.
03CA:   CA     62      DEX           0=1 BYTE, 1=2 BYTE,
03CB:   D0 0C   63      BNE    XLATE    2=3 BYTE.
03CD:   A5 0B   64      LDA    INST
03CF:   29 0D   65      AND    #$D      WEED OUT NON-ZERO-PAGE
03D1:   F0 14   66      BEQ    STINST   2 BYTE INSTS (IMM).
03D3:   29 08   67      AND    #$8      IF ZERO PAGE ADR
03D5:   D0 10   68      BNE    STINST   THEN CLEAR HIGH BYTE.
03D7:   85 0D   69      STA    INST+2
03D9:   20 89 F6 70      XLATE  JSR    SW16     IF ADR OF ZERO PAGE
03DC:   22     71      LD     FRMEND   OR ABS IS IN SOURCE
03DD:   D6     72      CPR    ADR      (FRM) BLOCK THEN
03DE:   02 06   73      BNC    SW16RT  SUBSTITUTE ADR-
03E0:   26     74      LD     ADR      SOURCE BEG+DEST BEG.
03E1:   B1     75      SUB    FRMBEG
03E2:   02 02   76      BNC    SW16RT
03E4:   A4     77      ADD    TOBEG
03E5:   36     78      ST     ADR
03E6:   00     79      SW16RT  RTN
03E7:   A2 00   80      STINST  LDX    #$0
03E9:   B5 0B   81      STINS2  LDA    INST,X
03EB:   91 42   82      STA    (A4L),Y COPY LENGTH BYTES
03ED:   E8     83      INX     OF INST FROM
03EE:   20 B4 FC 84      JSR    NXTA4   SW16 AREA TO
03F1:   C6 2F   85      DEC    LENGTH  DEST SEGMENT. UPDATE
03F3:   10 F4   86      BPL    STINS2  SOURCE, DEST SEGMENT
03F5:   90 C4   87      BCC    RELOC2  POINTERS. LOOP IF NOT
03F7:   60     88      RTS     BEYOND SOURCE SEG END.
03F8:   4C A6 03 90      USRLOC  ORG    $3F8
*****SUCCESSFUL ASSEMBLY: NO ERRORS

```

```
CROSS-REFERNCE: 6502 RELOCATION SUBROUTINE
ALL      003C      0050 0056
A4L      0042      0082
ADR      0006      0072 0074 0078
FRMBEG   0001      0075
FRMEND   0002      0071
GETINS   03BD      0059
IN        0200      0045
INIT     03B3      0053
INSDS2   F88E      0060
INST     000B      0057 0064 0069 0081
LENGTH   002F      0061 0085
NXTA4    FCB4      0084
R1L      0002      0051
RELOC    03A6      0090
RELOC2   03BB      0047 0087
STINS2   03E9      0086
STINST   03E7      0066 0068
SW16     F689      0070
SW16RT   03E6      0073 0076
TOBEG    0004      0077
USRLOC   03F8
XLATE    03D9      0063
YSAV     0034      0044 0048
FILE:
```

This page is not part of the original Wonderbook

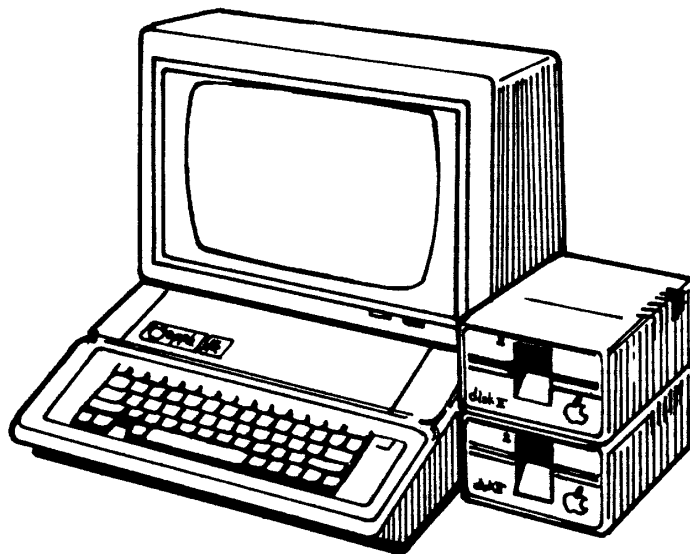
# The Woz Wonderbook

DOCUMENT

Apple-II

Renumbering and Appending  
BASIC Programs

15 November 1977



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook



R E N U M B E R I N G   A N D   A P P E N D I N G

B A S I C   P R O G R A M S

on the

A P P L E - I I   C O M P U T E R

S. Wozniak (WOZ)

November 15, 1977

## RENUMBERING AND APPENDING APPLE-II BASIC PROGRAMS

The answer to the question "what do 5, 11, 36, 150, 201, and 588 have in common?" is given as "adjacent rooms in the Warsaw Hilton"<sup>1</sup> but might just as well be "adjacent line numbers in my last BASIC program." The laws of entropy insure that the line numbers of a debugged and operational BASIC program give the appearance of having been selected by a KENO machine.\* Many a time I have spent an extra hour to retype a finished program while spacing the line numbers evenly just to make it 'look good'.

Another difficulty which I have experienced is joining two BASIC programs into a single, larger one. This 'append' operation is easier to accomplish by hand than renumbering. The sophisticated user can examine the BASIC memory map and perform some manual manipulations to join the programs providing that the line numbers do not overlap. Still, the manual append operation is highly prone to error.

---

<sup>1</sup> The Official Polish/Italian Joke Book, L. Wilde, Pinnacle Books, New York, N.Y., 1973, p. 17

\* In fact, while several texts detail how the boundary conditions of a KENO game lead to predictable outcomes, finished programs seldom exhibit this property.

The APPLE-II BASIC user now has a solution to these needs in the form of a hand- or tape-loadable program, RENUM/APPEND, described herein. The CALL command is used to activate one of three machine level programs. The renumber operation (RENUM) requires user specification of the original line number range over which renumbering is to occur, the new initial line number to be applied to the range, and the new line number increment to use. The example below specifies that lines 200 to 340 be renumbered starting with 100 and spaced by 10's.

```
RANGE BEGIN  200
RANGE END    340
NEW BEGIN    100
NEW INCREMENT 10
```

A second RENUM entry renumbers the entire program, relieving the user of the need to specify the range begin and end parameters. The append operation (APPEND) reads the second user (BASIC) program off tape with the first in memory.

Renumber and append error conditions (memory full and line number overlap) are detected just as in BASIC. In case of error the user is notified and no program alteration occurs.

## USING RENUM/APPEND

1. Load RENUM/APPEND (\* 300.3D4 R)

Note that the high-order bytes of page 3 are not loaded, preventing inadvertant alteration of the interrupt and user function (YC) vectors. The '\*' is generated by the MONITOR, not the user.

2. Load a BASIC program.
3. To renumber entire program:

```
POKE 2, START L      User must supply low and high bytes
POKE 3, START H      of new STARTing line number.

POKE 4, INCR L       User must supply low and high bytes
POKE 5, INCR H       of new line number INCRement.

CALL 768             (does not alter locations 2-5)
```

Note: START L is equivalent to START MOD 256

START H is equivalent to START / 256

4. To renumber a range of the program

```
POKE 2, START L
POKE 3, START H

POKE 4, INCR L
POKE 5, INCR H

POKE 6, RANGE START L  User must supply low and high bytes
POKE 7, RANGE START H  of renumber range starting line number.

POKE 8, RANGE END L    User must supply low and high bytes
POKE 9, RANGE END H    of renumber range ending line number.

CALL 776               (does not alter locations 2-9)
```

5. To append program #2 (larger line numbers) to program #1  
(smaller line numbers):

(a) Load program #2

(b) CALL 956

Be sure you are running the tape of program #1 as this  
command will load it.

(c) If you get a memory full error then use the command  
CALL 973 to recover the original program.

### ERRORS

1. If not enough free memory exists to contain the line number table during pass 1 of RENUM then the message '(beep) \*\*\* MEM FULL ERR' is displayed and no renumbering occurs. The same message is displayed if not enough free memory exists to hold the product of an APPEND. In the case of APPEND, the user will have to type the BASIC command CALL 973 to recover his original program. The user can free additional memory by eliminating all active BASIC variables with the CLR command.
2. If renumbering results in a line number overlap (detected during pass 1 of RENUM) then the message '(beep) \*\*\* RANGE ERR' is displayed and no renumbering occurs. This error may mean that one or more parameters were not specified or were incorrectly specified.

### CAUTIONS

1. When appending a program, always load the one with greater line numbers first.
2. The user must be aware that branch target expressions may not be renumbered. For example, the statement GO TO ALPHA will not be modified by RENUM. The statement GO TO 100 + ALPHA will be modified only to reflect the new line number assigned to the old line 100.

## APPLE-II BASIC STRUCTURE

An understanding of the internal representation of a BASIC program is necessary in order to develop RENUMBER and APPEND algorithms. Figure 1 illustrates the significant pointers for a program in memory. Variable and symbol table assignment begins at the location whose address is contained in the pointer LOMEM (\$4A and \$4B where '\$' stands for hex). This is \$800 (2048) on the APPLE-II unless changed by the user with the LOMEM: command. A second pointer, PV (Variable Pointer, at \$CC and \$CD) contains the address of the location immediately following the last location allocated to variables. PV is equal to LOMEM if no variables are actively assigned as is the case after a NEW, CLR, or LOMEM: command. As variables are assigned, PV increases.

The BASIC program is stored beginning with the lowest numbered line at the location whose address is contained in the pointer PP (Program Pointer, at \$CA and \$CB). The pointer HIMEM (\$4C and \$4D) contains the address of the location immediately following the last byte of the last line of the program. This is normally the top of memory unless changed by the user with the HIMEM: command. As the program grows, PP decreases. PP is equal to HIMEM if there is no program in memory. Adequate checks in the BASIC insure that PV never exceeds PP. This in essence says that variables and program are not permitted to overlap.

Lines of a BASIC program are not stored as they were originally entered (in ASCII) on the APPLE-II due to a pre-translation stage. Internally each line begins with a length byte which may serve as a link to the next line. The length byte is immediately followed by a two-byte line number stored in binary, low-order byte first. Line numbers range from 0 to 32767. The line number is followed by 'items' of various types, the final of which is an 'end-of-line' token (\$01). Refer to figure 2.

Single bytes of value less than \$80 (128) are 'tokens' generated by the translator. Each token stands for a fixed unit of text as required by the syntax of the language BASIC. Some stand for keywords such as PRINT or THEN while others stand for punctuation or operators such as ',' or '+'.

Integer constants are stored as three consecutive bytes. The first contains \$B0-\$B9 (ASCII '0'-'9') signifying that the next two contain a binary constant stored low-order byte first. The line number itself is not preceded by \$B0-\$B9. All constants are in this form including line number references such as 500 in the statement GO TO 500. Constants are always followed by a token. Although one or both bytes of a constant may be positive (less than \$80) they are not tokens.



Variable names are stored as consecutive ASCII characters with the high order bit set. The first character is between \$C1 and \$DA (ASCII 'A'-'Z'), distinguishing names from constants. All names are terminated by a token which is recognizable by a clear high-order bit. The '\$' in string names such as A\$ is treated as a token.

String constants are stored as a token of value \$28 followed by ASCII text (with high-order bits set) followed by a token of value \$29. REM statements begin with the REM token (\$5D) followed by ASCII text (with high-order bits set) followed by the 'end-of-line' token.

Figure 1 - MEMORY MAP

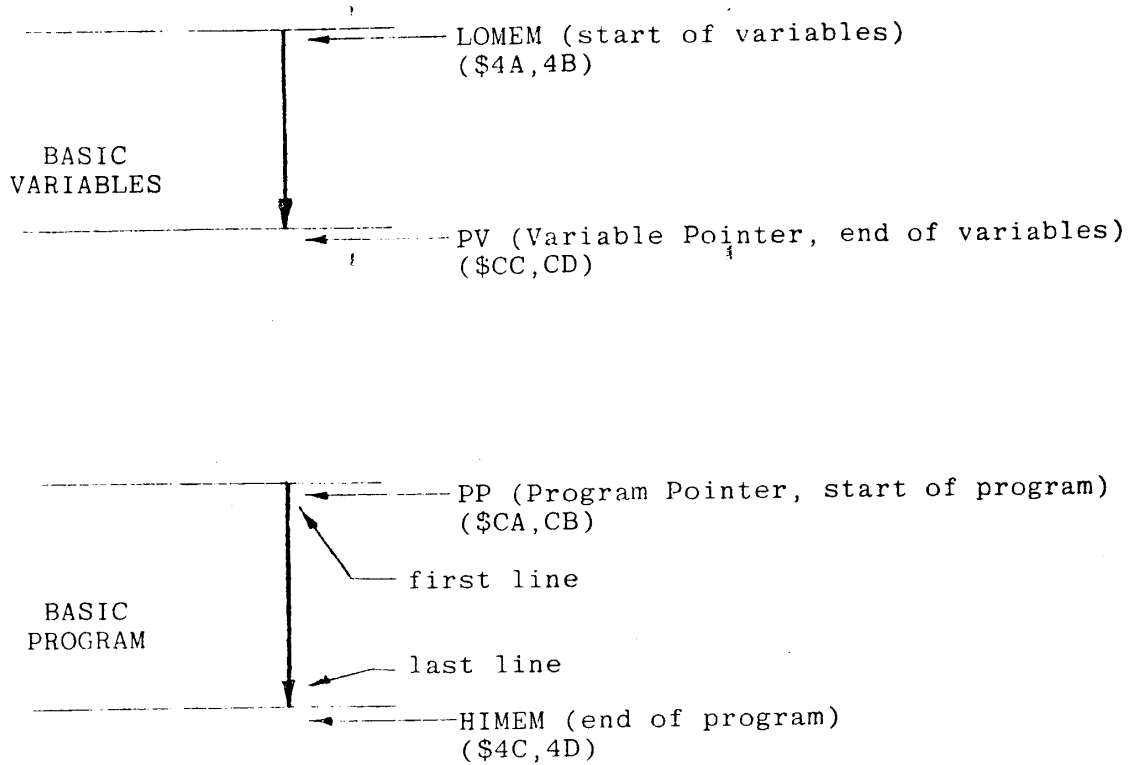
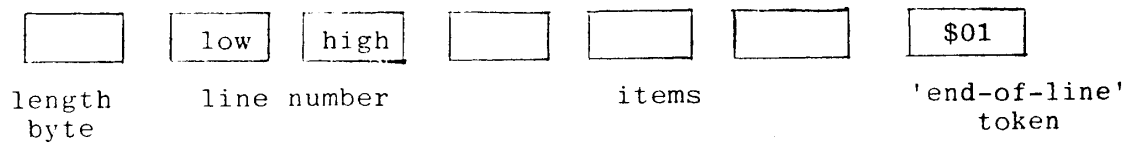


Figure 2 - LINE REPRESENTATION





## RENUMBER - THEORY OF OPERATION

Because of the rigid internal representation of APPLE-II BASIC programs (insured by the translator syntax check) writing a renumber program was a somewhat easier task than it would have been on many small BASIC's. Fortunately all constants in APPLE-II BASIC (including line number references) are preconverted to binary.

The normal renumber subroutine entry point is RENUM (\$308). The RENX entry (\$300) conveniently sets the renumber range for the user such that the entire program will be renumbered. RENUM extensively uses SWEET16, the code-saving 16-bit interpretive machine built into the APPLE-II.<sup>1</sup> Occasional 6502 code is interspersed throughout RENUM for even greater code efficiency.

RENUM scans the entire program from beginning to end twice. During pass 1 a line number table is built containing all line numbers of the program found to be within the renumber range. This table begins at the address specified by the BASIC variable pointer, PV, and is limited in length by the program pointer, PP. Each entry is two bytes long. A memory full error occurs if not enough free memory is available for the table.

---

<sup>1</sup> Byte Magazine, Nov. 1977, pp.

As line numbers are entered in the table corresponding new line numbers are generated and both new and old are displayed. Should the new line numbers result in an 'out of ascending sequence' condition, then a range error occurs and renumbering is terminated. It is assumed that the line numbers of the original program are in ascending sequence.

The purpose of pass 2 is to scan the entire BASIC program while updating all references of line numbers found in the table to new assignments. Aside from the line numbers themselves, the line number references sought are identified as constants immediately preceded by one of the following tokens:

GOTO

GOSUB

THEN lno

LIST

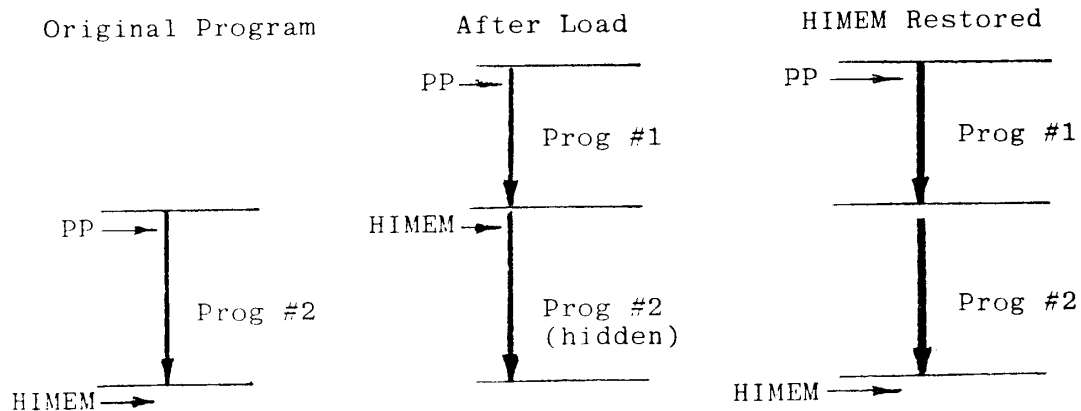
LIST

No other statement normally permitted within an APPLE-II BASIC program may contain a line number reference. No errors will occur during pass 2.

Exceptions such as empty line number table and null program are properly considered by both passes of RENUM.

When APPEND is called, the user program with larger line numbers will be in memory and the one with smaller line numbers will be read off tape. The current program resides between two pointers, PP and HIMEM. HIMEM is preserved and set to the value contained in PP. This 'hides' the original program and prepares to load a new one immediately above it in memory.

The BASIC load subroutine is called and a normal memory full error condition will result if not enough free memory is available to contain both programs. If this error occurs then the original program will still be hidden. Fortunately, it can be recovered by calling the tail end of APPEND at \$3CD which simply restores HIMEM. If the load is successful then HIMEM is restored to its original value and both programs will be joined. No line number overlap check is performed.



## RENUMBER EXAMPLE

Original	Renumber lines 100-110 to start at 150 spaced by 10
>LIST	>POKE 2, 150 MOD 256
1 GOTO 100	>POKE 3, 150 / 256
2 GOSUB 103	>POKE 4, 10 MOD 256
3 IF TRUE THEN 107	>POKE 5, 10 / 256
4 LIST 109,110	>POKE 6, 100 MOD 256
100 REM	>POKE 7, 100 / 256
103 REM	>POKE 8, 110 MOD 256
107 REM	>POKE 9, 110 / 256
109 REM	>CALL 776
110 REM	100->150
200 FOR I=1 TO 10	103->160
210 PRINT I	107->170
220 NEXT I	109->180
230 GOTO 1	110->190
	>LIST
	1 GOTO 150
	2 GOSUB 160
	3 IF TRUE THEN 170
	4 LIST 180,190
	150 REM
	160 REM
	170 REM
	180 REM
	190 REM
	200 FOR I=1 TO 10
	210 PRINT I
	220 NEXT I
	230 GOTO 1

RENUMBER EXAMPLE (cont)

Renumber lines 100-110 to start at  
10 spaced by 5

```
>POKE 2, 10 MOD 256
```

```
>POKE 3, 10 / 256
```

```
>POKE 4, 5 MOD 256
```

```
>POKE 5, 5 / 256
```

```
>CALL 768
```

```
1->10
```

```
2->15
```

```
3->20
```

```
4->25
```

```
150->30
```

```
160->35
```

```
170->40
```

```
180->45
```

```
190->50
```

```
200->55
```

```
210->60
```

```
220->65
```

```
230->70
```

```
>LIST
```

```
10 GOTO 30
```

```
15 GOSUB 35
```

```
20 IF TRUE THEN 40
```

```
25 LIST 45,50
```

```
30 REM
```

```
35 REM
```

```
40 REM
```

```
45 REM
```

```
50 REM
```

```
55 FOR I=1 TO 10
```

```
60 PRINT I
```

```
65 NEXT I
```

```
70 GOTO 10
```



APPEND EXAMPLE

```
>LIST
  100 REM
  200 REM THE ORIGINAL PROGRAM
  300 REM

>CALL 956

>LIST
  10 REM
  20 REM THIS PROGRAM CAME FROM TAPE
  30 REM
  100 REM
  200 REM THE ORIGINAL PROGRAM
  300 REM
```

```
APPLE-II BASIC RENUMBER/APPEND SUBROUTINES
9:53 A.M., 11/21/1977
1 TITLE 'APPLE-II BASIC RENUMBER/APPEND SUBROUTINES'
2 *****
3 *
4 * APPLE-II BASIC *
5 * RENUMBER AND APPEND *
6 * SUBROUTINES *
7 *
8 * RENUMBER *
9 * NEW INITIAL (2,3) *
10 * NEW INCR (4,5) *
11 * RANGE BEG (6,7) *
12 * RANGE END (8,9) *
13 *
14 * USE RENX ENTRY *
15 * FOR RENUMBER ALL *
16 *
17 * WOZ 11/16/77 *
18 * APPLE COMPUTER INC. *
19 *
20 *****
21 PAGE
```

6502 EQUATES

9:53 A.M., 11/21/1977

PAGE: 2

22	SUBTTL	6502 EQUATES	
23	ROL	EPZ \$0	LOW-ORDER SW16 R0 BYTE
24	ROH	EPZ \$1	HI-ORDER.
25	R11L	EPZ \$16	LOW-ORDER SW16 R11 BYTE
26	R11H	EPZ \$17	HI-ORDER.
27	HIMEM	EPZ \$4C	BASIC HIMEM POINTER.
28	PPL	EPZ \$CA	BASIC PROG POINTER.
29	PVL	EPZ \$CC	BASIC VAR POINTER.
30	MEMFULL	EQU \$E36B	BASIC MEM FULL ERROR.
31	PRDEC	EQU \$E51B	BASIC DECIMAL PRINT SUB
32	RANGERR	EQU \$EE68	BASIC RANGE ERROR.
33	LOAD	EQU \$F0DF	BASIC LOAD SUBR.
34	SW16	EQU \$F699	SWEET16 ENTRY.
35	CROUT	EQU \$FD8E	CHAR RET SUBR.
36	COU	EQU \$FDED	CHAR OUT SUBR.
37		PAGE	

9:53 A.M., 11/21/1977

SWEET16 EQUATES

PAGE: 3

38	SUBTTL	SWEET16	EQUATES	
39	ACC	EPZ	\$0	SWEET16 ACCUMULATOR.
40	NEWLOW	EPZ	\$1	NEW INITIAL LNO.
41	NEWINCR	EPZ	\$2	NEW LNO INCR.
42	LNLOW	EPZ	\$3	LOW LNO OF RENUM RANGE.
43	LNHI	EPZ	\$4	HI LNO OF RENUM RANGE.
44	TBLSTRT	EPZ	\$5	LNO TABLE START.
45	TBLNDX1	EPZ	\$6	PASS 1 LNO TBL INDEX.
46	TBLIM	EPZ	\$7	LNO TABLE LIMIT.
47	SCR8	EPZ	\$8	SCRATCH REG.
48	HMEM	EPZ	\$8	HIMEM (END OF PRGM).
49	SCR9	EPZ	\$9	SCRATCH REG.
50	PRGNDX	EPZ	\$9	PASS 1 PROG INDEX.
51	PRGNDX1	EPZ	\$A	ALSO PROG INDEX.
52	NEWLN	EPZ	\$B	NEXT 'NEW LNO'.
53	NEWLN1	EPZ	\$C	PRIOR 'NEW LNO' ASSIGN
54	TBLND	EPZ	\$6	PASS 2 LNO TABLE END.
55	PRGNDX2	EPZ	\$7	PASS 2 PROG INDEX.
56	CHRO	EPZ	\$9	ASCII '0'.
57	CHRA	EPZ	\$A	ASCII 'A'.
58	MODE	EPZ	\$0	CONST/LNO MODE.
59	TBLNDX2	EPZ	\$B	LNO TBL IDX FOR UPDATE
60	OLDLN	EPZ	\$D	OLD LNO FOR UPDATE.
61	STRCON	EPZ	\$B	BASIC STR CON TOKEN.
62	REM	EPZ	\$C	BASIC REM TOKEN.
63	R13	EPZ	\$D	SWEET16 REG 13 (CPR RE
64	THEN	EPZ	\$D	BASIC THEN TOKEN.
65	LIST	EPZ	\$D	BASIC LIST TOKEN.
66	SCRC	EPZ	\$C	SCRATCH REG FOR APPEND
67			PAGE	

```

                                APPLE-II BASIC RENUMBER SUBROUTINE - PASS 1
9:53 A.M., 11/21/1977                                PAGE: 4
                                SUBTTL APPLE-II BASIC RENUMBER SUBROUTINE - PASS 1
                                68
                                69
0300: 20 89 F6 70   RENX      JSR   $300
                                71   SUB   ACC      OPTIONAL RANGE ENTRY.
0303: B0           72   ST   LNLOW    SET LNLOW=0,
0304: 33           73   ST   LNHI     LNHI=$FFFF
0305: 34           74   DCR   LNHI
0306: F4           75   RTN
0307: 00           76   RENUM     JSR   SW16
0308: 20 89 F6 76   SET   SCR8,HIMEM
030B: 18 4C 00 77   LDD  @SCR8    BASIC HIMEM POINTER
030E: 68           78   ST   HMEM     TO HMEM.
030F: 38           79   SET   SCR9,PVL+2
0310: 19 CE 00 80   POPD @SCR9    BASIC VAR PTR TO
0313: C9           81   ST   TBLSTRT  TBLSTRT AND TBLNDX1.
0314: 35           82   ST   TBLNDX1
0315: 36           83   LD   NEWLOW   COPY NEWLOW (INITIAL)
0316: 21           84   ST   NEWLN    TO NEWLN.
0317: 3B           85   ST   NEWLN1
0318: 3C           86   POPD @SCR9    BASIC PROG PTR
0319: C9           87   ST   TBLIM   TO TBLIM
031A: 37           88   ST   PRGNDX  AND PRGNDX.
031B: 39           89   LD   PRGNDX
031C: 29           90   CPR   HMEM     IF PRGNDX >= HMEM
031D: D8           91   BC   PASS2   THEN DONE PASS 1.
031E: 03 46       92   ST   PRGNDX1
0320: 3A           93   LD   TBLNDX1
0321: 26           94   INR   ACC     IF < 2 BYTES AVAIL IN
0322: E0           95   CPR   TBLIM   LNO TABLE THEN RETURN
0323: D7           96   BC   MERR    WITH 'MEM FULL' MSG.
0324: 03 38       97   LD   @PRGNDX1
0326: 4A           98   ADD   PRGNDX   ADD LEN BYTE TO
0327: A9           99   ST   PRGNDX   PROG INDEX.
0328: 39          100  LDD  @PRGNDX1   LINE NUMBER.
0329: 6A          101  CPR   LNLOW    IF < LNLOW THEN
032A: D3          102  BNC  PIB     GO TO PIB.
032B: 02 2A       103  CPR   LNHI    IF > LNHI THEN
032D: D4          104  BNC  PIA     GO TO PIC.
032E: 02 02       105  BNZ  PIC
0330: 07 30       106  STD  @TBLNDX1  ADD TO LNO TABLE.
0332: 76          107  RTN
0333: 00          108  LDA   ROH     *** 6502 CODE ***
0334: A5 01       109  LDX   ROL
0336: A6 00       110  JSR   PRDEC    PRINT OLD LNO '->' NEW
0338: 20 1B E5 111  LDA   #SAD    (RO,R11) IN DECIMAL.
033B: A9 AD       112  JSR   COUT
033D: 20 ED FD 113  LDA   #SBE
0340: A9 BE       114  JSR   COUT
0342: 20 ED FD 115  LDA   RI1H
0345: A5 17       116  LDX   RI1L
0347: A6 16       117  JSR   PRDEC
0349: 20 1B E5 118  JSR   CROUT
034C: 20 8E FD 119  JSR   SW16+3
034F: 20 8C F6 120  LD   NEWLN    *** END 6502 CODE ***
0352: 2B          121

```

```

                APPLE-II BASIC RENUMBER SUBROUTINE - PASS 1
9:53 A.M., 11/21/1977
0353: 3C      122      ST  NEWLN1      COPY NEWLN TO NEWLN1
0354: A2      123      ADD NEWINCR    AND INCR NEWLN BY
0355: 3B      124      ST  NEWLN      NEWINCR.
0356: 0D      125      NUL           (WILL SKIP NEXT INST).
0357: D1      126  PIB   CPR  NEWLOW    IF LOW LNO < NEWLOW
0358: 02 C2   127      BNC PASS1     THEN RANGE ERR.
035A: 00      128  RERR  RTN           PRINT 'RANGE ERR' MSG
035B: 4C 68 EE 129      JMP  RANGERR  AND RETURN.
035E: 00      130  MERR  RTN           PRINT 'MEM FULL' MSG
035F: 4C 6B E3 131      JMP  MEMFULL  AND RETURN.
0362: EC      132  PIC   INR  NEWLN1    IF HI LNO <= MOST RECH
0363: DC      133      CPR  NEWLN1    NEWLN THEN RANGE ERR
0364: 02 F4   134      BNC  RERR
                135      PAGE
    
```

```

                                APPLE-II BASIC RENUMBER SUBROUTINE - PASS 2
9:53 A.M., 11/21/1977
                                PAGE: 6
                                SUBTTL APPLE-II BASIC RENUMBER SUBROUTINE - PASS 2
0366: 19 B0 00 137 PASS2      SET  CHRO,$B0      ASCII '0'
0369: 1A C1 00 138          SET  CHRA,$C1      ASCII 'A'
036C: 27          139 P2A      LD   PRGNDX2
036D: D8          140          CPR  HMEM          IF PROG INDEX = HIMEM
036E: 03 63      141          BC   DONE          THEN DONE PASS 2.
0370: E7          142          INR  PRGNDX2      SKIP LEN BYTE.
0371: 67          143          LDD  @PRGNDX2     LINE NUMBER.
0372: 3D          144 UPDATE  ST   OLDLN       SAVE OLD LNO.
0373: 25          145          LD   TBLSTRT
0374: 3B          146          ST   TBLNDX2     INIT LNO TABLE INDEX.
0375: 21          147          LD   NEWLOW      INIT NEWLNI TO NEWLOW.
0376: 1C 00 00 148          SET  NEWLNI,0     (WILL SKIP NEXT 2 INSN)
                                149          ORG  *-2
0377: 2C          150 UD2      LD   NEWLNI
0378: A2          151          ADD  NEWINCR     ADD INCR TO NEWLNI.
0379: 3C          152          ST   NEWLNI
037A: 2B          153          LD   TBLNDX2     IF LNO TBL IDX = TBLND
037B: B6          154          SUB  TBLND       THEN DONE SCANNING
037C: 03 07      155          BC   UD3         LNO TABLE.
037E: 6B          156          LDD  @TBLNDX2     NEXT LNO FROM LNO TABL
037F: BD          157          SUB  OLDLN       LOOP TO UD2 IF NOT SAE
0380: 07 F5      158          BNZ  UD2         AS OLDLN.
0382: C7          159          POPD @PRGNDX2    REPLACE OLD LNO WITH
0383: 2C          160          LD   NEWLNI     CORRESPONDING NEW L
0384: 77          161          STD  @PRGNDX2
0385: 1B 28 00 162 UD3      SET  STRCON,$28   STR CON TOKEN.
0388: 1C 00 00 163          SET  MODE,0      (SKIPS NEXT 2 INSTR'S)
                                164          ORG  *-2
0389: 67          165 GOTCON  LDD  @PRGNDX2
038A: FC          166          DCR  MODE          IF MODE = 0 THEN UPDAE
038B: 08 E5      167          BMI  UPDATE     LNO REF.
038D: 47          168 ITEM    LD   @PRGNDX2     BASIC ITEM.
038E: D9          169          CPR  CHRO
038F: 02 09      170          BNC  CHKTOK     CHECK TOKEN FOR SPECIA
0391: DA          171          CPR  CHRA          IF >= '0' AND < 'A' TH
0392: 02 F5      172          BNC  GOTCON     SKIP CONST OR UPDAE
0394: F7          173 SKPASC  DCR  PRGNDX2
0395: 67          174          LDD  @PRGNDX2     SKIP ALL NEG BYTES OF
0396: 05 FC      175          BM   SKPASC     STR CON, REM, OR NAM
0398: F7          176          DCR  PRGNDX2
0399: 47          177          LD   @PRGNDX2
039A: DB          178 CHKTOK  CPR  STRCON     STR CON TOKEN?
039B: 06 F7      179          BZ   SKPASC     YES, SKIP SUBSEQUEN
039D: 1C 5D 00 180          SET  REM,$5D
03A0: DC          181          CPR  REM          REM TOKEN?
03A1: 06 F1      182          BZ   SKPASC     YES, SKIP SUBSEQUEN
03A3: 08 13      183          BMI  CONST     GOSUB, LOOK FOR LNO.
03A5: FD          184          DCR  R13
03A6: FD          185          DCR  R13          (TOKEN $5F IS GOTO)
03A7: 06 0F      186          BZ   CONST     THEN LNO, LOOK FOR LNO.
03A9: 1D 24 00 187          SET  THEN,$24
03AC: DD          188          CPR  THEN
03AD: 06 09      189          BZ   CONST     THEN LNO, LOOK FOR LNO.

```

APPLE-II BASIC RENUMBER SUBROUTINE - PASS 2

9:53 A.M., 11/21/1977

PAGE: 7

03AF:	F0	190	DCR	ACC	
03B0:	06 BA	191	BZ	P2A	EOL (TOKEN \$01)?
03B2:	ID 74 00	192	SET	LIST,\$74	
03B5:	BD	193	SUB	LIST	SET MODE = 0 IF LIST
03B6:	09 01	194	BNMI	CONTS2	OR LIST COMMA (\$73,8
03B8:	BQ	195	SUB	ACC	CLEAR MODE FOR LNO
03B9:	3C	196	ST	MODE	UPDATE CHECK.
03BA:	01 D1	197	BR	ITEM	CHECK NEXT BASIC ITEM.
		198	PAGE		



APPLE-II BASIC APPEND SUBROUTINE

9:53 A.M., 11/21/1977

PAGE: 8

```
199  SUBTTL APPLE-II BASIC APPEND SUBROUTINE
03B0: 20 89 F6 200  APPEND      JSR  SW16
03B1: 1C 4E 00 201          SET  SCRC,HIMEM+2
03C2: CC          202          POPD #SCRC      SAVE HIMEM.
03C3: 38          203          ST   HMEM
03C4: 19 CA 00 204          SET  SCR9,PPL
03C7: 69          205          LDD  #SCR9      SET HIMEM TO PRESERVE
03C8: 7C          206          STD  #SCRC      PROGRAM.
03C9: 00          207          RTN
03CA: 20 DF F0 208          JSR  LOAD      LOAD FROM TAPE.
03CD: 20 89 F6 209          JSR  SW16
03D0: CC          210          POPD #SCRC      RESTORE HIMEM TO SHOW
03D1: 28          211          LD   HMEM      BOTH PROGRAMS
03D2: 7C          212          STD  #SCRC      (OLD AND NEW).
03D3: 00          213  DONE   RTN      RETURN.
03D4: 60          214          RTS
*****SUCCESSFUL ASSEMBLY: NO ERRORS
```

CROSS-REFERENCE: APPLE-II BASIC RENUMBER/APPEND SUBROUTINES

ACC	0000	0071 0095 0190 0195
APPEND	033C	
CRSTOK	039A	0170
CRRO	0009	0137 0169
CHRA	000A	0138 0171
CONTS2	03B9	0194
CONTST	03B8	0183 0186 0189
COUT	FD5D	0113 0115
COOUT	FD5E	0119
DOSE	03D3	0141
GOYCON	0389	0172
HITEM	004C	0077 0201
HMEM	0008	0079 0091 0140 0203 0211
ITEM	038D	0197
LIST	000D	0192 0193
LXHI	0004	0073 0074 0104
LXLOW	0003	0072 0102
LOAD	F0DF	0208
MEMFULL	E36B	0131
MERR	035E	0097
MODE	000C	0163 0166 0196
NEWINCR	0002	0123 0151
NEWLN	000B	0085 0121 0124
NEWLNI	000C	0086 0122 0132 0133 0148 0150 0152 0160
NEWLOW	0001	0084 0126 0147
OLDLN	000D	0144 0157
PIA	0332	0105
PIB	0357	0103
PIC	0362	0106
PRA	036C	0191
PASS1	031C	0127
PASS2	0366	0092
PPL	00CA	0204
PRDEC	E51B	0111 0118
PRGNDX	0009	0089 0090 0099 0100
PRGNDX1	000A	0093 0098 0101
PRGNDX2	0007	0139 0142 0143 0159 0161 0165 0168 0173 0174 0176 0177
FVL	00CC	0080
ROH	0001	0109
ROL	0000	0110
RIIH	0017	0116
RIIL	0016	0117
RI3	000D	0184 0185
RANGERR	EE68	0129
REM	000C	0180 0181
RENUM	0308	
RENX	0300	
RERR	035A	0134
SCR8	0008	0077 0078
SCR9	0009	0080 0081 0087 0204 0205
SCRC	000C	0201 0202 0206 0210 0212
SKPASC	0394	0175 0179 0182
STRCON	000B	0162 0178
SW16	F589	0070 0076 0120 0200 0209
TBLIM	0007	0088 0096
TBLND	0006	0154
TBLNDX1	0006	0083 0094 0107
TBLNDX2	000B	0146 0153 0156
TBLSTRT	0005	0082 0145
TREN	000D	0187 0188

CROSS REFERENCE: APPLE-II BASIC RENUMBER/APPEND SUBROUTINES  
UD2           0377       0158  
UD3           0385       0155  
UPDATE       0372       0167

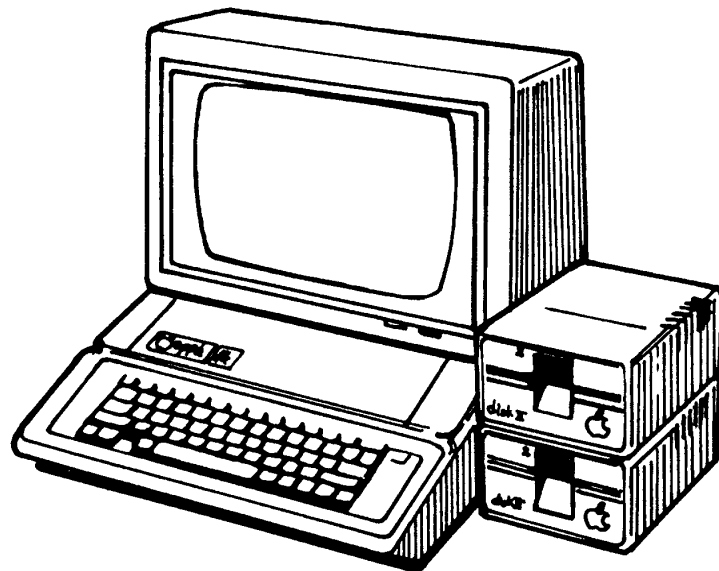
This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

## References

03 November 2004



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

# The Woz Wonderbook

## References

David T Craig • 03 November 2004

Here is a list of Apple Computer and Apple-II computer technical and historical reference materials that may prove beneficial to readers of the Woz Wonderbook who want to know more about the details behind this document and the Apple-II computer in the late 1970's.

These more polished references originated in publicly published Apple Computer documents, magazine articles, and Apple-II enthusiast private materials.

David T Craig (shirlgato@cybermesa.com) has digital copies of all of these materials. These materials may possibly be provided with DigiBarn's Woz Wonderbook digital materials via its web site or a CD.

### SYSTEM DESCRIPTION: THE APPLE-II

Steve Wozniak • BYTE Magazine • May 1977

This description of the Apple-II computer by its main designer provides a concise description of this computer's technical features.

### MICROCOMPUTER FOR USE WITH VIDEO DISPLAY

Steve Wozniak • US Patent 4,136,359 • 23 January 1979

This is Apple Computer's patent for the Apple-II computer assigned to Steve Wozniak. Dry reading, but has some good Apple-II technical information. Available on the US Patent Office web site <http://www.uspto.gov/patft/>.

### APPLE-II HISTORY

Steven Weyhrich • <http://apple2history.org/history/> • 1991-2003

This great web site contains a cornucopia of accurate Apple-II historical information. If you want to learn about the origins of Apple Computer and the Apple-I and Apple-II computers, this is the place to go. Also available on the internet at <http://www.blinkenlights.com/classiccmp/apple2history.html>.

### SWEET-16: THE 6502 DREAM MACHINE

Steve Wozniak • BYTE Magazine • November 1977

This is Steve Wozniak's comprehensive description of his SWEET-16 16-bit byte-code "meta microprocessor" interpreter built into the Apple-II Integer BASIC ROM. Wozniak's Apple-II system description in BYTE May 1977 also has a short description of SWEET-16.

This page is not part of the original Wonderbook

## APPLE-II REFERENCE MANUAL ("RED BOOK")

Apple Computer • January 1978

This is Apple Computer's first published technical reference manual for the Apple-II computer. It is commonly referred to as the "Red Book" because it has a red cover. The Red Book's contents (155 pages) were based on the Woz Wonderbook but in a more polished format, but is not as comprehensive or readable as the later Apple-II reference manuals. A good PDF scan of the Red Book can be found on the internet at <http://bitsavers.org/pdf/apple/> along with several other older Apple-II manuals.

## APPLE-II REFERENCE MANUAL

Apple Computer • 1979 • Document # 030-0004-01

This is Apple Computer's first revision of the Apple-II Red Book. This 275 page manual is much improved over the Red Book and tremendously improved over the Woz Wonderbook materials. Note the Apple document number (030-0004-01) which indicates this is a technical manual (030), is document number 4 (0004), and is revision 1 (01) which means this is Apple's 4th published manual.

## APPLE-II MINI MANUAL

Apple Computer • 1977-1978

This 68 page manual from Apple Computer appears to be the predecessor to the Red Book from 1978. As such, I would date this manual in the 1977-1978 range. More complete and more detailed than the Woz Wonderbook, but not as good as the Red Book. A good PDF scan of this manual can be found on the internet at <http://bitsavers.org/pdf/apple/>.

## THE WOZ PAK ]

Call-A.P.P.L.E. Magazine • 15 November 1979

This 138 page document contains a large number of technical documents about the Apple-II computer courtesy of Apple Computer and Call-A.P.P.L.E. magazine. This is better organized and more comprehensive than the Woz Wonderbook or the Red Book, but not as good as the Apple-II Reference Manual from 1979. Contains a detailed article on the Apple-II floating point package.

## PEEKING AT CALL-A.P.P.L.E.

Call-A.P.P.L.E. Magazine • 1978 and 1979

This 2 volume set (volume 1 dated 1978 has 92 pages, volume 2 dated 1979 has 206 pages) contains lots of Apple Computer re-produced technical information and original Call-A.P.P.L.E. magazine information. Well worth reading.

This page is not part of the original Wonderbook



## PROGRAMMER'S AID #1: INSTALLATION AND OPERATING MANUAL

Apple Computer • 1978 • Document # 030-0026-01

This 113 page Apple manual describes the special programming built into the Programmer's Aid #1 ROM chip (there was never an Aid #2 chip AFAIK). Includes several 6502 assembly language programs by Steve Wozniak which used his SWEET-16 16-bit byte-code interpreter. Includes more polished information for the Integer BASIC renumber and append programs described in the Woz Wonderbook.

## FLOATING POINT ROUTINES FOR THE 6502

Steve Wozniak & Roy Rankin

Dr. Dobb's Journal of Computer Calisthenics & Orthodontia • August 1976

This is an article on the Apple-II floating point package pre-dating the Woz Wonderbook. Has more details about this package than the Wonderbook. Available on the internet at [www.strotmann.de/twiki/bin/view/APG/AsmAppleFloatingPoint](http://www.strotmann.de/twiki/bin/view/APG/AsmAppleFloatingPoint). Concerning authorship of this floating point package, web site <http://linux.monroecc.edu/~paulrsm/dg/dg32.htm> says Wozniak wrote the core package routines (e.g. ADD) and Rankin wrote the transcendental routines (e.g. LOG).

## DISASSEMBLER PROGRAM FOR THE 6502

Steve Wozniak & Allen Baum

Dr. Dobb's Journal of Computer Calisthenics & Orthodontia • September 1976

This is an article on the Apple-II 6502 disassembler pre-dating the Woz Wonderbook. Available on the internet at <http://users.telenet.be/kim1-6502/kun/i14/p06.html>.

## THE APPLE II PLUS PERSONAL COMPUTER SYSTEM

Apple Computer • November 1981

This is Apple Computer's data sheet for the Apple-II Plus computer, the successor to the Apple-II computer. Shows how some of the enhancement ideas documented in the Woz Wonderbook and the Red Book were implemented by Apple.

## PRELIMINARY APPLE BASIC USERS MANUAL

Apple Computer • October 1976

This 16 page manual seems to be Apple Computer's first user manual for its Apple-II Integer BASIC programming language. The Woz Wonderbook is very lacking in Integer BASIC information for the user. A good PDF scan of this manual can be found on the internet at <http://bitsavers.org/pdf/apple/>.

This page is not part of the original Wonderbook

## APPLE TECH NOTES

Apple Computer and the International Apple Core (IAC) • July 1982

This 500 page document contains an extensive collection of Apple Computer technical notes from 1982 covering the Apple-II and Apple-III computer families. Many Apple-II hardware, software and documentation errata details are here. Includes articles about the Apple-II mini-assembler and cassette interfacing. A treasure trove of early Apple system technical information.

## APPLE-II SYSTEM MONITOR ROM LISTING

Apple Computer • 1977

For detailed information about the internal software workings of the Apple-II computer the source listing for the Apple-II System Monitor ROM is the key. Available in the Apple-II reference manual dated 1979 or on the internet at <http://members.buckeye-express.com/marksm/6502/>.

## STEVE WOZNIAK INTERVIEW: HOMEBREW TO CHAMPAGNE

Apple Orchard Magazine • Spring 1981

An early interview with Steve Wozniak in which he provides contemporary details about Apple Computer's origins and early days.

## STEVE WOZNIAK INTERVIEW: THE APPLE STORY

BYTE Magazine • December 1984

A great interview with Steve Wozniak by BYTE magazine with lots of Apple Computer and Apple-II information. Also includes a retrospective on SWEET-16, Wozniak's 16-bit byte-code interpreter. This is available on the internet at <http://apple2history.org/museum/articles/byte8412/byte8412.html>.

## STEVE WOZNIAK INTERVIEW: STEVE WOZNIAK UNBOUND

SlashDot Interview • January 2000

<http://slashdot.org/interviews/00/01/07/1124211.shtml>

This 2000 interview of Steve Wozniak contains some good 24 year recollections about Apple Computer's origins and early years.

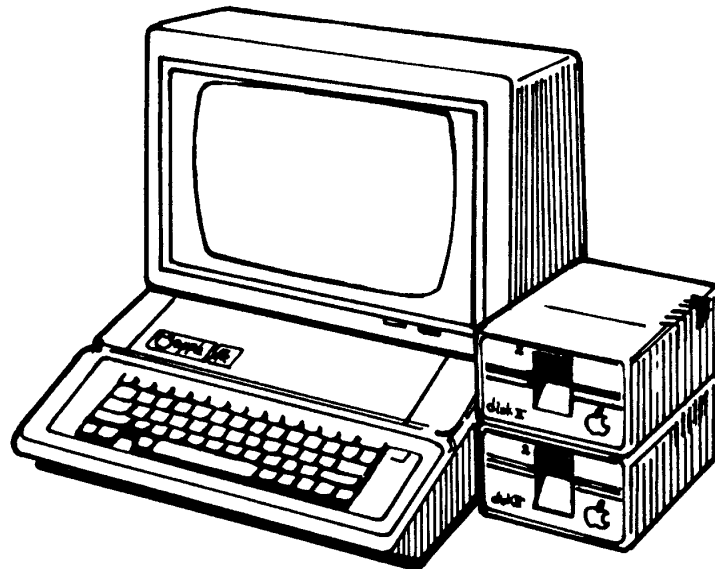
This page is not part of the original Wonderbook

# The Woz Wonderbook

DOCUMENT

Bill Goldberg Interview

19 April 2004



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

# The Woz Wonderbook

## Bill Goldberg Interview

Bruce Damer • 19 April 2004

### Source:

<http://www.digibarn.com/collections/books/woz-wonderbook/goldberg-on-woz-wonderbook.mp3> (3.3 MB file)

### Transcript created by

David T Craig <shirlgato@cybermesa.com> -- 02 November 2004

Interviewer: Bruce Damer <bdamer@digitalspace.com>  
Interviewee: Bill Goldberg <billau@coastside.net>

Interview duration: 3:39 minutes

### BACKGROUND

The "Woz Wonderbook" was a compilation of notes from Steve Wozniak's filing cabinet that served as the first documentation and technical support manual for the Apple II computer (before the more famous "red book" of January 1978). Bill Goldberg, longtime Apple employee, donated his copy of the Wonderbook to the DigiBarn (thanks Bill!). At the time he was at Apple there was only a single copy of this thick binder of photocopied notes, diagrams and such to be found in the Apple library. Bill, being in the technical support role and a natural pack rat, made a copy of the Wonderbook.

### INTERVIEW TRANSCRIPT

**BILL GOLDBERG:** Here it's faded. This is the Woz Wonderbook. And its disorganized but I found the copy of this in the Apple library and immediately made some copies of it.

**BRUCE DAMER:** So this was before the Red Book?

**BILL GOLDBERG:** This is what the Red Book was made from.

**BRUCE DAMER:** Oh gosh.

**BILL GOLDBERG:** Actually, I've got one or two Red Books for you.

**BRUCE DAMER:** Wonderful, because the Red Book we have is on loan.

**BILL GOLDBERG:** Actually, in Service Engineering we would get the leftovers of things. People would say "we don't need any more of this". So we had two cases of Red Books and a few of us in the department said "Hmm, these are worth something" and we divided them up.

This page is not part of the original Wonderbook

BRUCE DAMER: Wow.

BILL GOLDBERG: So, anyway, in here you will find some of the stuff typed, a number of different articles, but you will also find, unfortunately the xerox did the best job it could and it has faded over the years, but there's handwritten notes.

BRUCE DAMER: So Woz wrote these notes?

BILL GOLDBERG: Uh-Hmm [yes]. Here's a listing with some hand disassembly and his comments. Article on the disassembler.

BRUCE DAMER: So this is Woz's hand notes?

BILL GOLDBERG: Well, it's hand notes, it's various articles.

BRUCE DAMER: Here's a disassembled disassembler.

BILL GOLDBERG: Uh ha. But all written by hand.

BRUCE DAMER: Written by hand. Yup.

BILL GOLDBERG: And let's see. Here for instance, here's an article on the cassette system.

BRUCE DAMER: Ok.

BILL GOLDBERG: We (he?) gave up on using the cassette, but this actually is his handwritten notes on the cassette system. So ...

BRUCE DAMER: This is a big book. He must have sat for hours writing this down.

BILL GOLDBERG: You know, somebody just went through a file drawer of his notes and put it in a binder.

BRUCE DAMER: Oh.

BILL GOLDBERG: And there was only one of these in the Apple library. So ...

BRUCE DAMER: Wow.

BILL GOLDBERG: Either I or one of my colleagues checked it out and made some copies because this was going to disappear into obscurity.

BRUCE DAMER: This is the Woz Wonderbook?

BILL GOLDBERG: This is what it was called on the spine.

BRUCE DAMER: This would have been [19]77?

This page is not part of the original Wonderbook

BILL GOLDBERG: Uhm, actually the first article on the first of this has a date of 9/20/77 [20 September 1977]. So, but this is just a collection of a lot of different ... this actually goes into explaining ...

BRUCE DAMER: Yeah ...

BILL GOLDBERG: Yup. The detail that, you know ... I'm sure some of this is hideously proprietary but who will ever know.

BRUCE DAMER: Well, not at this point.

BILL GOLDBERG: Ok, so that's the Woz ...

BRUCE DAMER: Woz ...

BILL GOLDBERG: Actually, it won't hurt to write on the spine ... here take a pen, so its in your handwriting. That was my handwriting, so there's nothing special about that.

###

This page is not part of the original Wonderbook

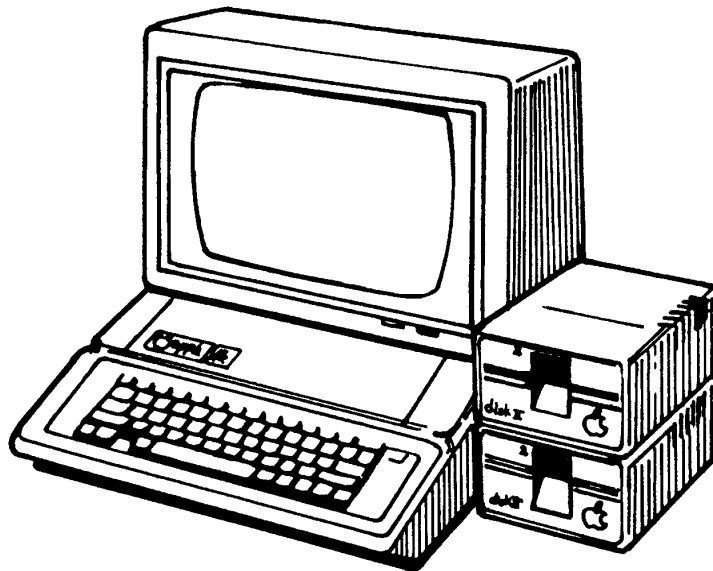
This page is not part of the original Wonderbook



# The Woz Wonderbook

DOCUMENT

## Credits



This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

# The Woz Wonderbook

## Credits

Thanks to Bill Goldberg for donating this copy of the Woz Wonderbook.

The DigiBarn Computer Museum and Curator Bruce Damer for providing it to the education and research community.

David T Craig is to be thanked for resurrecting the Wonderbook into a modern digital format.

And of course, thanks to Steve Wozniak for creating the Woz Wonderbook!



*Steve Wozniak, Co-founder Apple*

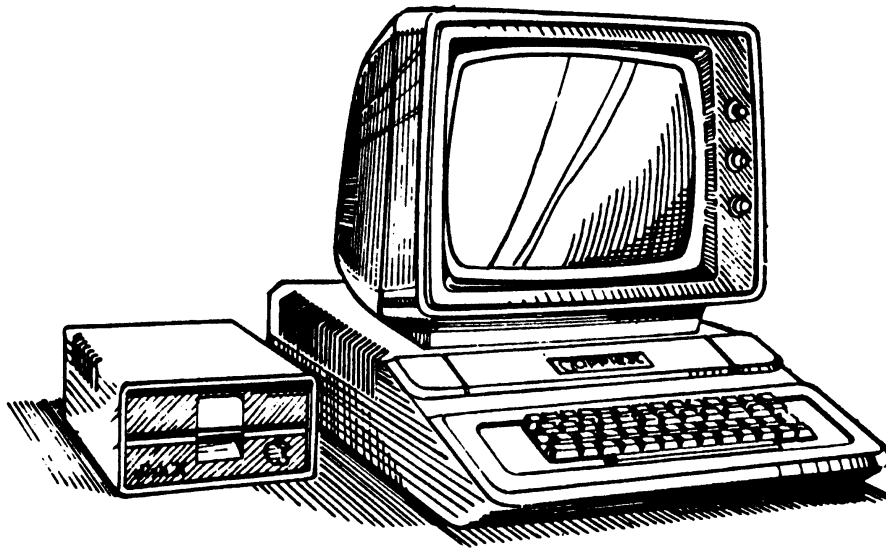
Steve Wozniak circa 1977 and 1981

This page is not part of the original Wonderbook

This page is not part of the original Wonderbook

# The Woz Wonderbook

The End



This page is not part of the original Wonderbook