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)

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/>)

Document dates of record:

September 20, 1977-November 15, 1977

Owner:

DigiBarn Computer Museum (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).

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).

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).

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
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:

**BY:**

Attribution. You must give the original author credit.

**Noncommercial:**

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

**Share Alike:**

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
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 (no blanks) (\ 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 handwritten 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 "F6666G" 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

<table>
<thead>
<tr>
<th>MOV</th>
<th>DEC</th>
<th>DESCRIPTEON</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>480</td>
<td>Some buffered output from class not seen appear on screen.</td>
</tr>
<tr>
<td>200</td>
<td>512</td>
<td>line buffer.</td>
</tr>
<tr>
<td>32, 64</td>
<td>768</td>
<td>Top of Applesoft BASIC program.</td>
</tr>
<tr>
<td>72, 71</td>
<td></td>
<td>Top of 'variable area' in Applesoft.</td>
</tr>
</tbody>
</table>
This page is not part of the original Wonderbook
The Woz Wonderbook

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
APPLE-II 2716 EROM ADAPTATION
('DO' 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 'DO' and 'D8' from pins 18 of the other ROMS. Reinsert the 'EO' ROM when done.

   'EO SOCKET'

   (Pin 1)

   Cut this trace

2. On the underside of the APPLE-II board, cut the traces connecting pin 20 to 21 of ROMs 'DO' 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 1 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 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

<table>
<thead>
<tr>
<th>BASE address(hex)</th>
<th>Secondary BASE address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0_{1,1}$</td>
<td>$800$</td>
</tr>
<tr>
<td>$2_{3}$</td>
<td>$880$</td>
</tr>
<tr>
<td>$4_{5}$</td>
<td>$900$</td>
</tr>
<tr>
<td>$6_{7}$</td>
<td>$980$</td>
</tr>
<tr>
<td>$8_{9}$</td>
<td>$A00$</td>
</tr>
<tr>
<td>$A_{B}$</td>
<td>$A80$</td>
</tr>
<tr>
<td>$C_{D}$</td>
<td>$B00$</td>
</tr>
<tr>
<td>$E_{F}$</td>
<td>$B80$</td>
</tr>
<tr>
<td>$10_{11}$</td>
<td>$828$</td>
</tr>
<tr>
<td>$12_{13}$</td>
<td>$8A8$</td>
</tr>
<tr>
<td>$14_{15}$</td>
<td>$928$</td>
</tr>
<tr>
<td>$16_{17}$</td>
<td>$9A8$</td>
</tr>
<tr>
<td>$18_{19}$</td>
<td>$A28$</td>
</tr>
<tr>
<td>$1A_{1B}$</td>
<td>$AA8$</td>
</tr>
<tr>
<td>$1C_{1D}$</td>
<td>$B28$</td>
</tr>
<tr>
<td>$1E_{1F}$</td>
<td>$BA8$</td>
</tr>
<tr>
<td>$20_{21}$</td>
<td>$850$</td>
</tr>
<tr>
<td>$22_{23}$</td>
<td>$8D0$</td>
</tr>
<tr>
<td>$24_{25}$</td>
<td>$950$</td>
</tr>
<tr>
<td>$26_{27}$</td>
<td>$9D0$</td>
</tr>
<tr>
<td>$28_{29}$</td>
<td>$A50$</td>
</tr>
<tr>
<td>$2A_{2B}$</td>
<td>$AD0$</td>
</tr>
<tr>
<td>$2C_{2D}$</td>
<td>$B50$</td>
</tr>
<tr>
<td>$2E_{2F}$</td>
<td>$BD0$</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Color Graphics Page Zero Variable Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBASEL</td>
</tr>
<tr>
<td>GBASH</td>
</tr>
<tr>
<td>H2</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>MASK</td>
</tr>
<tr>
<td>COLOR</td>
</tr>
</tbody>
</table>

GBASEL and GBASH are used by the color graphics subroutines as a pointer to the first (leftmost) byte of the current plot line. The (GBASEL),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)


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 #$3A     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

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

```
LDY #35         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
Adding Colors
to
Apple-II Hi-Res
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).
Mount a 7400 (Dual CMOS Flip Flip) and a 74LS20
Junction Logic in the breadboard area (Pl-814).

Wire the following circuit (× indicates that wire goes
across the 0-4* pin that is not out of its socket.)

[Diagram of wiring connections]
-30-17

To set correct timing conditions
in VSYNC on standard (60 Hz)

\[ V2 \rightarrow 12 \]

\[ V1 \rightarrow 4 \]

\[ \text{C17 \rightarrow Sync} \]

\[ C1 \rightarrow 6 \]

\[ VB \rightarrow 2 \]

\[ \text{C11 \rightarrow \text{Sync}} \]
The Woz Wonderbook

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
DISASSEMBLER ARTICLE

(pertains to APPLE-II MONITOR ROM)
APPLE DISASSEMBLER

1. Description. This subroutine package is used to display single or sequential 6502
instructions in mnemonic form. The subroutines are intended to disassemble
and debugging aids but ladies with more general usage (assemblers) are included.
The subroutines occupy one page (256 bytes) and ladies 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 hex bytes; (3) 3 character mnemonic, or "??" for invalid ops (which assume a length of 1 byte); and (4) address field, in one of the following formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Address Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty)</td>
<td>Invalid, Implied, Accumulator</td>
</tr>
<tr>
<td>$12</td>
<td>Page zero.</td>
</tr>
<tr>
<td>$1234</td>
<td>Absolute, Branch (target pointed)</td>
</tr>
<tr>
<td>#1,2</td>
<td>Immediate</td>
</tr>
<tr>
<td>$12, X</td>
<td>Zero page, indexed by X</td>
</tr>
<tr>
<td>$12, Y</td>
<td>Zero page, indexed by Y</td>
</tr>
<tr>
<td>$1234, X</td>
<td>Absolute, indexed by X</td>
</tr>
<tr>
<td>$1234, Y</td>
<td>Absolute, indexed by Y</td>
</tr>
<tr>
<td>(1,234)</td>
<td>Indirect.</td>
</tr>
<tr>
<td>($12, X)</td>
<td>Indexed Indirect.</td>
</tr>
<tr>
<td>($12, Y)</td>
<td>Indirect Indexed.</td>
</tr>
</tbody>
</table>

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) **DSMBS**: Disassembles and displays 26 sequential instructions beginning at
the address specified by the page zero variables PCL and PCr. For example, if called with 102 in PCl and 595 in PCr, 26
instructions beginning at address $35D2 will be disassembled. PCl and
PCr are updated to contain the address of the last disassembled
instruction. Must be called with $5A02 in hexadecimal mode ($5A
is hex $b2, bit clear). All processor registers are altered (except 5 - stack pointer). Uses INSTDSP and PCADS.

(b) **INSTDSP**: Disassembles and displays a single instruction whose address is
specified by PCl and PCr. Must be called in hexadecimal mode.
All processor registers (except 5) are altered. Uses PCADS, PRPC, PRBLNK, PRBLZ, PRNTAX, PRBITE, 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) **PRNTAX**: 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) **PRNYX**: 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 ($P in hex
blanks). Alters A, clears X. Uses CHAROUT.

(i) **PRBL3**: Outputs a character from the A register followed by X-1 blanks. In
other words, X specifies the total number of character output ($X-1$ in
blanks). Alters A, clears X. Uses CHAROUT.
4. Running as a program. The following program will run a disassembly.

```
9F0 20 0 8  JSR DSBML
9F3 4C 15 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 9F0R (on APPLE-1 system). 20 instructions will be disassembled. Hitting 9F3 again will give the next 20, etc.

Cassette tapes supplied for the ACE-1 (APPLE Cassette Interface) are intended to be loaded from 1500 to E9FF.

5. Non-APPLE systems. Source and object code supplied occupies pages 5 and 6. All code is on page 5. Tables on page 6. These tables may be relocated as will MODE, MODE2, CHART, CHART2, MNEM, and MNAME. The code may also be relocated. Be careful if you use pages 0 or 1. Pages 5 is the subroutine return stack and page 6 must contain 7 variables (if use DSBML). These may be relocated on page 6 but PCL must always immediately precede PCH for (page, y) addressing.

```
{ 140  FORMAT }
141  LENGTH  [Used in INSIDE, DSBML]
142  XMNEM   
143  RMMEM    
144  PCL     [Used by PCL, INSIDE, DBMML]
145  PCH     
146  COUNT?  [Used by DBMML only]
```
(j) PCADJ: \( (PCL, PCH) + 1 + \) (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).
3. Modifications.

(a) To change '+' to '=' for immediate mode change location $9355$ 
(on code enciOSed) from a $A3$ to a $BD$.

(b) To skip the '$+$' (meaning hex) preceding disassembled values make the 
following changes:

- $946$ : $81$ (was $81$)
- $947$ : $82$ (was $82$)
- $94C$ : $91$ (was $91$)
- $94D$ : $92$ (was $92$)
- $94E$ : $96$ (was $96$)
- $950$ : $85$ (was $85$)
- $951$ : $9D$ (was $9D$)
- $952$ : $8A$ (was $8A$)
- $953$ : $8A$ (was $8A$)

(c) To have address field of accumulator addressed instructions print as '@'.

1. Must skip $+$ preceding disassembled values by making modification
   (b) above.

2. Change the following locations.

- $949$ : $88$ (was $88$)
- $952$ : $C1$ (was $A4$)

(d) To add R0 and addressing modes change the following locations.

- $901$ : $06$ (was $06$)
- $902$ : $06$ (was $06$)
- $911$ : $02$ (was $02$)
- $916$ : $16$ (was $16$)
- $91F$ : $16$ (was $16$)
- $916$ : $67$ (was $67$)
- $911$ : $87$ (was $87$)
DSMVL  LDA #13
STA COUNT

DSMVL2 JSR INSTDSP
JSR PCADJ
STA PCL
STY PCH
DEC COUNT

BNE DSMVL2

INSTDSP
JSR PRPC
LDA (PCL,X)
TAY
LDA
LSR
BCC IEVEN
LSR
BCS ERR
CMP #$22
BEQ ERR
AND #$7
ORA #$80
IEVEN
LSR
TAX
LDA MODE,X
LSR
BCS RTMODE
LSR
LSR
LSR
RSR
29 F
RTMODE AND #$F
LDY #$80
ERR

LDY
AA
AA
GETFMT
TAX
LDA MODE2,X
STA FORMAT
AND #$13

Count for 20 instruction disassembly.

Disassemble + display one instruction.

Update PCL, PCH to next instruction.

Done first 19 instructions?

Yes, loop. Else disassemble 20th.

Print PCL, PCH.

Get op code.

Even/odd test.

b1 test.

XXXXXXXX instruction invalid.

10001001 instruction invalid.

Mask 3 bits for address mode and add indexing offset.

LSB into carry for left/right test below.

Index into address mode table.

If carry set use LSD for print format index.

If carry clear use MSD.

Mask for 4-bit index.

$0 for invalid opcodes.

Substitute $80 for all invalid opcodes.

Set print format index to 0.

Index into print format table.

Save for address field formatting.

Mask for 2-bit length (W:1b to 13:2b).

$12 $12 b7 b7 b7 b7 b7 b7 b7 b7
STA \text{LENGTH}
\text{TYA}\n\text{AND} \#\$8\text{F} \quad \text{Op code.}
\text{Mask it for 1XXX1010 test.}
\text{TAX}\n\text{TYA}\n\text{LDY} \#3\n\text{CPX} \#\$8\text{A}\n\text{BEQ MNNDX3}
\text{MNNDX1} \text{LSR}\n\text{BCC MNNDX3}\quad \text{Form index into mnemonic table.}
\text{LDA (PCL),Y}\n\text{ORA} \#\$2\text{0}\n\text{DEY}\n\text{BNE MNNDX2}\n\text{INY}\n\text{DEY}\n\text{BNE MNNDX1}\quad \text{Save mnemonic table index.}
\text{LDA (PCL),Y}\n\text{JSR PRBYTE}\n\text{LDA} \#\$1\n\text{JSR PRBL2}\n\text{CPY LENGTH}\n\text{INY}\n\text{BCC PROP}\n\text{LDA} \#\$3\n\text{CPY} \#\$4\n\text{BCC PROPBL}\n\text{PLA}\n\text{TAY}\n\text{LDA MNEML,Y}\n\text{STA LNMEM}\n\text{LDA MNEMR,Y}\n\text{STA RMNEM}
| SC3  | 2ø  F2 8 | RELADR SC3 JSR PCADJ3 | PCL, PCH + Displacement + 1 to A, Y. |
| SC6  | AA       | TAX                   |                                          |
| SC7  | E8       | INX                   |                                          |
| SC8  | DØ 1     | BNE PRNTYX            | +1 to X, Y.                              |
| SC9  | C8       | INY                   |                                          |
| SCB  | 98       | PRNTYX TYA           |                                          |
| SCC  | 2ø DC FF | PRNTAX JSR PRBYTE    | Output target address of branch and return. |
| SCF  | 8A       | PRNTX TXA            |                                          |
| SDØ  | 4C DC FF |                      |                                          |
| SDB  | A9 8D    | PRPC LDA #$8D         |                                          |
| SDC  | 2ø EF FF | JSR CHAROUT          | Output carriage return.                 |
| SDE  | A5 45    | LDA PCH               |                                          |
| SDA  | A6 44    | LDX PCL               |                                          |
| SDC  | 2ø CC 8  | JSR PRNTAX           | Output PCH & PCL.                       |
| SDF  | A9 AD    | LDA #$AD             |                                          |
| SE1  | 2ø EF FF | REI JSR CHAROUT      | Output "-".                             |
| SE4  | A2 3     | PRBLNK LDA #$3       | Blank count.                            |
| SE6  | A9 AØ    | PRBL2 LDA #$AØ       |                                          |
| SE7  | 2ø EF FF | PRBL3 JSR CHAROUT    | Output a blank.                         |
| SE8  | CA       | DEX                   |                                          |
| SEC  | DØ F8    | BNE PRBL2            | Loop until count = 0.                   |
| SEL  | 6Ø       | RTS                   |                                          |
| SEF  | A5 41    | PCADJ LDA LENGTH     | Ø = 1 byte, 1 = 2 byte, 2 = 3 byte.     |
| SF1  | 38       | PCADJ2 SEC           | Test displ. sign. (for rel. branch).    |
| SF2  | A4 45    | PCADJ3 LDY PCH       | Extend neg. by decrementing PCH.        |
| SF4  | AA       | TAX                   |                                          |
| SF5  | 16 1     | BPL PCADJ4           |                                          |
| SF6  | 88       | DEY                   |                                          |
| SF8  | 65 44    | PCADJ4 ADC PCL       |                                          |
| SFH  | 9Ø 1     | BCC RTSI             | PCL + LENGTH (or displ.) + 1 to A.     |
| SFC  | C8       | INY                   | Carry into Y (PCH).                     |
| SFD  | 6Ø       | RTSI RTS             |                                          |
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
CASSETTE ARTICLE
The standard audio cassette recorder is rapidly becoming the most popular mass storage peripheral in micro-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 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 E0000 (hex) and its data is located at beginning at address 01000 (hex) then a record file may be used. 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 be stopped in between any pair of records. A header precedes data on the record to insure the recorder reaches speed. A sync byte precedes the data and indicates its start. A checksum byte is recorded after all data bytes for error detection.
A record

\[
\begin{array}{c|c|c|c}
\text{HEADER} & \text{SYNC} & \text{DATA} & \text{CHECKSUM} \\
\end{array}
\]

HEADER

The header consists of a .5 second to 20 second space move 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)

\[
\begin{array}{c}
\begin{array}{c}
\text{5000} \\
\text{2500} \\
\text{1250} \\
\text{625} \\
\text{0} \\
\end{array}
\end{array}
\]

SYNC

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

\[
\begin{array}{c}
\begin{array}{c}
\text{500} \\
\text{250} \\
\text{125} \\
\text{62} \\
\text{31} \\
\text{0} \\
\end{array}
\end{array}
\]
In the first byte recorded is typically 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 186 bytes per second.

**Checksum**

The checksum byte immediately follows the last data byte and is recorded in the same 8-bit 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
Data byte 3 = 10100110
Checksum = 01010011
Notes: (1) An existing input port may be used in place of the 8797.

(2) Any decoded address strobe (glitch-free) may be used in place of the 7436.

(3) If an inverter is desired for address decoding, the unused half of the 7434 may be used.

\[ \text{Invert Input} \rightarrow \text{Inverted Output} \]
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 counter.
Reading a bit is accomplished as follows:

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

2. RDBIT subroutine is called. It will loop while waiting for a toggle of the tapein signal, while decrementing the Y-REG once every 12 usec. 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 usec for a full cycle (two toggles).

The final carry state indicates whether a 0 (short cycle) or 1 (long cycle) was read. The A-REG is used, hence modified, X-REG unchanged.

Note: base & location for 'LASTIN' must be provided.
reading a byte:

1. Initialize A-REG as if 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 0 pointers
   (AIL, AIH) 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. (AIL, AIH)
   pointer is incremented until it
   is greater than (A2L, A2H).
   All registers are zeroed.
   (d) checksum is written.
   (e) sound BELL
Read a Record:

1. Initialize (AIL, A1H) 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, A1H) until greater than (AZL, AZH)
   f. Read page checksum byte. If mismatch then print "ERR"
   g. Sound BELL

Note that all registers on page 6 locations LASTIN and CHKSUM are used.
cover should not contain a negative value (wi - ri hex). If so, your hardware is working.

---

Writing a Tape

(1) Initialize a block of memory to be written.

(2) Enter the cassette routines by hand. You may wish to store these programs permanently on PROM or EROM.

(3) Initialize locations 3C and 3D 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 (3E and 3F) to the 16-bit ending address for the data block.

(5) Store the following program in memory

```
WRITE JSR WRITE 26 JMP NON 4C 1F F1
```

(6) Run WRITE. Immediately after typing the run command, start the recorder.

It must be in the RECORD mode with the run jack connected to the interface. The recorder must be turned on prior to writing.

Note: the recorder will return after 10 seconds for the header, and 5 to 10 seconds for
Code should not contain a negative value (FF FF hex). If so, your hardware is
working.

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 ERROM.

3. Initialize locations 3C and 3D to the 16-bit starting address for the data
block to be written. The low-order half of the address must be in A0L, the
high-order half in A0H.

4. Initialize A2L and A2H (locations 3E and 3F) to the 16-bit ending address for the
data block.

5. Store the following program in memory

   WRITE JSR WRITE 27
   JMP MON 4C IF FF

6. Run WRITE. Immediately after typing
the run command, start the recorder.
It must be in the record mode with
its out jack connected to the interface.
With done, the recorder will return. Allow 10
seconds for the program to load.

Writing a tape
(1) Enter the cassette routines into memory (if not already there).

(2) Initialize A1, A1H, A2, and A2H as for writing tapes.

(3) Store the following program in memory:

```
READ JSR READ 28 FF
```

(4) Run READ. 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 cursor will return. The word error will appear if the checksum 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 requiring a system reset.
Variable Allocation

Page & workspace should be assigned for the following variables:

AIL
A1H
A2L
A2H
LASTIN
CHKSUM

The only restriction is that AIL must immediately precede A1H and A2L must immediately precede A2H; otherwise you may assign these variables differently than the provided listing.

User interface on non-APPLE

User supplied subroutines

For USER printout and BELL prompts, the user must provide a character output subroutine, COUT. The assembly listing provided uses the APPLE-I entry point EFFE for this subroutine; you may substitute your own. The A3 X- and Y-areas must not be disturbed by this subroutine. The byte to be output is passed in the A-AREA.
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 (AIL, A1H) and (AZL, AZH), calls the READ or WRITE subroutine, then repeats the address pointer and subroutine call for all further records. Even if the tape is not stripped, it is permissible to spend a small amount of time calculating between records, since the first part of the header is ignored.
I have tested the interface at Apple over millions of bits without failure. I have used the cheapest types I could find and the cheapest recorders. The test patterns were representative of random data. What were some of the considerations?

First, let's look at a typical input/output wave forms:

\[ \text{IN} \quad \begin{array}{c|c|c|c} \hline \text{A} & \text{B} & \text{C} & \text{D} \\ \hline \end{array} \]

\[ \text{OUT} \quad \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \end{array} \]

It can be seen that echo crossings of the output are quite approximated 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 rogue 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. (Compare 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 zeros or a string of ones presents no major problem. A major problem does creep up when the data contains mixes which show up in cheap recorders but not good ones. This has to do with the read and write amplifiers within the recorder. Virtually all recorders have a satisfactory bandpass.

Rolloff at 2 to 3 kHz typical

Distributed under the Creative Commons License on page 5
Page 0076 of 0213
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
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

- Exponent
- Signed Mantissa
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,
   | 11.XXXXXX  | exponent = -128.

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 $00$ ($S$ means hexadecimal) and the most positive, +127, is stored as $FF$ (all ones).
<table>
<thead>
<tr>
<th>Exponent</th>
<th>Stored As</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>10000001 ($81)</td>
</tr>
<tr>
<td>+2</td>
<td>10000010 ($82)</td>
</tr>
<tr>
<td>+3</td>
<td>10000011 ($83)</td>
</tr>
<tr>
<td>-1</td>
<td>01111111 ($7F)</td>
</tr>
<tr>
<td>-2</td>
<td>01111110 ($7E)</td>
</tr>
<tr>
<td>-3</td>
<td>01111101 ($7D)</td>
</tr>
</tbody>
</table>

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

```
0 0 0 0 0 0 0 0 1
EXP high MANTISSA low
```

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

```
$7F$ $7F$ $FF$ $FF$
EXP MANTISSA
```
# Floating Point Representation Examples

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

FP1:  \[ \begin{array}{c}
X.YYY... \\
X1 \\
M1 \end{array} \]

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

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

FP1:  \[ \begin{array}{c}
$S3$ \\
$60$ \\
0 \\
0 \end{array} \] (+12)

FP2:  \[ \begin{array}{c}
$S2$ \\
$B0$ \\
0 \\
0 \end{array} \] (-5)

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

FP1:  \[ \begin{array}{c}
$S2$ \\
$70$ \\
0 \\
0 \end{array} \] (+7)
FSUB subroutine (address $F168)

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).

\[
\begin{array}{c|c|c|c|c|c|c}
\text{FP1:} & \$82 & \$70 & 0 & 0 & (7) \\
& X1 & M1 & & & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{FP2:} & \$82 & \$80 & 0 & 0 & (-5) \\
& X2 & M2 & & & \\
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c|c|c}
\text{FP1:} & \$83 & \$60 & 0 & 0 & (12) \\
& X1 & M1 & & & \\
\end{array}
\]
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: \[
\begin{array}{c}
\$83 \\
X1 \\
\$60 \\
M1 \\
\$0 \\
\$0 \\
\end{array}
\] (+12)

FP2: \[
\begin{array}{c}
\$82 \\
X2 \\
\$B0 \\
M2 \\
\$0 \\
\$0 \\
\end{array}
\] (-5)

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

FP1: \[
\begin{array}{c}
\$85 \\
X1 \\
\$88 \\
M1 \\
\$0 \\
\$0 \\
\end{array}
\] (-60)

FP2: \[
\begin{array}{c}
\$82 \\
X2 \\
\$50 \\
M2 \\
\$0 \\
\$0 \\
\end{array}
\] (+5)
FDIV subroutine (address SP4B2)

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).

\[
\begin{align*}
\text{FP1:} & \quad \begin{array}{c}
$85 \\
X1 \\
$88 \\
M1 \\
0 \\
0 \\
\end{array} & \quad \text{(-60)} \\
\text{FP2:} & \quad \begin{array}{c}
$83 \\
X1 \\
$60 \\
M1 \\
0 \\
0 \\
\end{array} & \quad \text{(+12)}
\end{align*}
\]

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

\[
\begin{align*}
\text{FP1:} & \quad \begin{array}{c}
$82 \\
X1 \\
$80 \\
M1 \\
0 \\
0 \\
\end{array} & \quad \text{(-5)}
\end{align*}
\]
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 XPFLOAT 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: $80 \quad \begin{array}{c} \$88 \\ X1 \end{array} \quad \begin{array}{c} \$44 \\ M1 \end{array} \quad \begin{array}{c} \$80 \\ 0 \end{array} \quad (+274)$
**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.

\[
\begin{array}{c|c|c|c|c}
\text{FP1:} & \$88 & \$44 & \$80 & 0 \\
\hline
\text{X1} & \text{M1}
\end{array}
\] (+274)

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

\[
\begin{array}{c|c|c|c|c}
\text{FP1:} & \$8E & \$01 & \$12 & 0 \\
\hline
\text{X1} & \text{M1}
\end{array}
\]

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

NORM subroutine (address SF463)

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 \(0.0011_2 \times 2^6\)).

\[
\begin{array}{c|c|c|c}
\text{FP1} & \$86 & \$0C & 0 & 0 \\
\hline
\text{X1} & & & 0 & 0 \\
\text{M1} & & & & (+12)
\end{array}
\]

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

\[
\begin{array}{c|c|c|c}
\text{FP1} & \$83 & \$60 & 0 & 0 \\
\hline
\text{X1} & & & 0 & 0 \\
\text{M1} & & & & (+12)
\end{array}
\]

NORM1 subroutine (address SF455)

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.

\[
\begin{array}{c}
FP1 \\
X1 \\
M1 \\
\end{array}
\begin{array}{c}
\$82 \\
\$50 \\
0 \\
0 \\
\end{array}
\quad (+5)
\]

\[
\begin{array}{c}
FP2 \\
\$82 \\
\$70 \\
0 \\
0 \\
\end{array}
\quad (+7)
\]

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.

\[
\begin{array}{c}
FP1 \\
\$82 \\
\$C0 \\
0 \\
0 \\
\end{array}
\quad (+12)
\]
**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.

**HTAR 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.

\[
\begin{array}{cccc}
\text{FP1} & \$83 & \$A0 & 0 & 0 \\
\text{X1} & \text{M1} & \\
\end{array}
\]

(-7)

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

\[
\begin{array}{cccc}
\text{FP1} & \$84 & \$D0 & 0 & 0 \\
\text{X1} & \text{M1} & \\
\end{array}
\]

(-7)

Note: M1 sign bit is unchanged.
RTLOG subroutine (address SF480)

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.

\[
\begin{array}{c}
\text{FP1:} \\
\hline
\text{X1} & \text{M1} \\
\$83 & \$A0 & 0 & 0 \\
\end{array}
\]

(-12)

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

\[
\begin{array}{c}
\text{FP1:} \\
\hline
\text{X1} & \text{M1} \\
\$84 & \$50 & 0 & 0 \\
\end{array}
\]

(+20)

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)

Purpose: To shift MANT1 and E right one bit without adjusting X1. This is used by the 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 the 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.
FLOATING POINT ROUTINES

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

1
2 *
3 * APPLE-II FLOATING *
4 * FONKI ROUTINES *
5 *
6 * COPYRIGHT 1977 BY *
7 * APPLE COMPUTER INC. *
8 *
9 * ALL RIGHTS RESERVED *
10 *
11 * S. WOZNIAK *
12 *
13 *************************************
14 TITLE "FLOATING FONKI ROUTINES"

15 SIGN  EPI $F3
16 X2   EPI $F4
17 M2   EPI $F5
18 X1   EPI $F8
19 M1   EPI $F9
20 E    EPI $FC
21 OVLOC EOU $3F5
22 ORG $F425

F425:  18  23 ADD CLC CLEAR CARRY.
F426:  A2  02  24 LDX #$2 INDX FOR 3-BYTE ADD.
F428:  B5  F9  25 ADD1 LDA M1.X ADD A BYTE OF MAN12 TO MAN1,
F42A:  75  F5  26 ADC M2.X
F42C:  95  F9  27 STA M1.X
F42E:  CA  28 DEX INDX TO NEXT MORE SIGNIF. BYT.
-40F:  10  F7  29 BPL ADD1 LOOP UNTIL DONE.
1:  60  30 RTS RETURN
-432:  06  F3  31 MUL ASL SIGN CLEAR LSB OF SIGN.
-434:  20  37 F4  32 JCR ASBSWAP ABS VAL OF M1, THEN SWAP WITH
-437:  24  F9  33 ABSWAP BIT M1 MAN1 NEGATIVE?
439:  10  05  34 BPL ABSWAP1 NO, SWAP WITH MAN12 AND RETURN.
-43B:  20  A4 F4  35 JSR FINCPL YES, COMPLEMENT IT.
-43E:  E6  F3  36 INC SIGN INC SIGN, COMPLEMENTING LSB.
-440:  30  37 ABSWAP1 SEC SET CARRY FOR RETURN TO MUL/DIV.
-441:  A2  04  38 LDX #$4 INDEX FOR 4-BYTE SWAP.
-443:  94  F8  39 SWAP1 STY E-1.X
445:  B5  F7  40 LDA X1-1.X SWAP A BYTE OF EXP/MAN11 WITH
-447:  B4  F3  41 LDY X2-1.X EXP/MAN12 AND LEAVE A COPY OF,
-449:  94  F7  42 STY X1-1.X MAN11 IN E (3 BYTES). E+3 US.
-44B:  95  F3  43 STA X2-1.X
-44D:  CA  44 DEX ADVANCE INDX TO NEXT BYTE.
-44E:  D0  F3  45 BNE SWAP1 LOOP UNTIL DONE.
-450:  60  46 RTS RETURN
-451:  A7  8E  47 FLOAT LDA #$8E INIT EXP1 TO 14.
-453:  85 F8  48 STA X1 THEN NORMALIZE TO FLOAT.
-455:  A5 F9  49 NORM1 LDA M1 HIGH-ORDER MAN1 BYTE.
-457:  09 CO  50 CMF #$C0 UPEK TWO BITS UNEQUAL?
-459:  30 OC  51 BMI RTS1 YES, RETURN WITH MAN11 NORMAL.
-45B:  06 F8  52 DEC X1 DECREMENT EXP1.
-45D:  06 FB  53 ASL M1+2
-45F:  26 FA  54 POL MI+1 SHIFI MAN1 (3 BYTES) LEFT.
### FLOATING POINT ROUTINES

**Page 2**

| Address | opcode | mnemonic | operation | condition | address | instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>49F000</td>
<td>00000</td>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td>E0</td>
<td>MULB A</td>
</tr>
<tr>
<td>49F000</td>
<td>00000</td>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td>E0</td>
<td>MULB A</td>
</tr>
<tr>
<td>49F000</td>
<td>00000</td>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td>E0</td>
<td>MULB A</td>
</tr>
<tr>
<td>49F000</td>
<td>00000</td>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td>E0</td>
<td>MULB A</td>
</tr>
<tr>
<td>49F000</td>
<td>00000</td>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td>E0</td>
<td>MULB A</td>
</tr>
</tbody>
</table>

**Notes:**
- This code is intended for assembly language programming.
- The operations include basic arithmetic such as addition, subtraction, and multiplication.
- The conditions indicate the flow of the program based on the results of the operations.

---

**Distributed under the Creative Commons License on page 5**

**Page 0101 of 0213**
FLOATING POINT ROUTINES

1.4  F. M., 10/3/1977

PAGE 3

F4C8: 90 02 109  BCC DIV4
F4CA: 8F F8 110  STA M2+3, X
F4CC: E8 111  DIV4
F4CF: D0 18 112  INX
F4D2: 26 FB 113  BNE DIV3
F4D4: 26 FA 114  ROL M1+1
F4D7: 26 FA 115  ROL M1
F4D9: 26 FA 116  ASL M2+2
F4DB: 26 FA 117  ROL M2+1
F4DE: 26 F6 118  ROL M2
F4DF: 60 1C 119  BCS OVFL
F4E8: 88 120  DEY
F4EC: D0 UA 121  BNE DIV1
F4F0: F0 BE 122  BER MUEND
F4F4: 86 FB 123  MUX
F4F6: 86 FD 124  STX M1+2
F4F8: 86 F8 125  STX M1
F4FA: 60 0D 126  BCS OVCHK
F4FE: 30 04 127  BMI MU3
F500: 68 128  PLA
F50C: 68 129  PLA
F510: 90 B2 130  BCC NORMX
F51F: 49 30 131  MUX
F524: F3 8E 132  EOR #590
F528: 85 F8 133  STA M1
F52C: A0 17 134  LDY #17
F530: 40 F7 135  OVCHK
F534: 40 F5 03 136  OVFL
F538: 10 03 137  JMP OVLOC
F53C: ORG #F63D
F540: 5F F8 139  FIX
F544: 10 13 140  BPL UNDFL
F548: 09 8E 141  CMP #58E
F54C: D0 F5 142  BNE FIX1
F550: 24 F9 143  BIT M1
F554: A5 F8 144  BPL FIXRTS
F558: A5 F8 145  LDA M1+2
F55C: EC F0 06 146  BEQ FIXRTS
F560: E6 FA 147  INC M1+1
F564: D0 02 148  BNE FIXRTS
F568: E6 F9 149  INC M1
F56C: 50 F5 150  FIXRTS
F570: 99 00 151  UNDFL
F574: 85 F9 152  STA M1
F578: 85 FA 153  STA M1+1
F57C: 60 154  RTS

*******************************************************************
SUCCESSFUL ASSEMBLY: NO ERRORS
CROSS-REFERENCE: FLOATING POINT ROUTINES

ABS\WAP F437 0032
ARB\WAP F440 0034
ADD F425 0064 0084
ADD1 F429 0029
ADDLND F477 0096
ALIGNSWP F47B 0060
COMPL1 F4A7 0095
DIV1 F46A 0121
DIV2 F48D 0106
DIV3 F4C7 0112
DIV4 F4CC 0109
L 00FC(Z) 0039 0074 0103
FADD F46E
FCOMPL F4A4 0035 0059
FDIV F4B2
FIX F640
FIX1 F63D 0142
FIXRTS F656 0144 0146 0148
FLOAT F451
FMUL F43C
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
MULND F4A0 0122
MUL1 F495 0006
MUL2 F49D 0083
NORM F463 0065 0088
NORM1 F455 0057
NORMX F4A2 0130
OVCHK F4F7 0126
OVFL F409 0072 0119
OVLOC 03F5 0136
RDN1 F436 0076
RTAR F47D 0138
HTLOG F480 0066
HTLOG1 F484 0082
HT1 F467 0051
SIUN 00F3(Z) 0031 0036 0087
SWAP F441 0067
SWAP1 F443 0045
SWP+ON F46B 0063
UNFL F657 0140
X1 00F8(Z) 0040 0042 0048 0052 0056 0062 0071 0079 0092 0093 0098
    0132 0139
X2 00F4 0041 0043 0061
The Woz Wonderbook

DOCUMENT

Apple-II

Sweet-16 -- The 6502 Dream Machine

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 Sub-routine), 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 OPCODE SUMMARY**

<table>
<thead>
<tr>
<th>Register Ops</th>
<th>Non-register Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1n SET Rn, Constant (Set)</td>
<td>00 RTN (Return to 6502 mode)</td>
</tr>
<tr>
<td>2n LD Rn (Load)</td>
<td>01 BR ea (Branch always)</td>
</tr>
<tr>
<td>3n ST Rn (Store)</td>
<td>02 BNC ea (Branch if No Carry)</td>
</tr>
<tr>
<td>4n LD @Rn (Load indirect)</td>
<td>03 BC ea (Branch if Carry)</td>
</tr>
<tr>
<td>5n ST @Rn (Store indirect)</td>
<td>04 BP ea (Branch if Plus)</td>
</tr>
<tr>
<td>6n LDD @Rn (Load double indirect)</td>
<td>05 BM ea (Branch if Minus)</td>
</tr>
<tr>
<td>7n STD @Rn (Store double indirect)</td>
<td>06 BZ ea (Branch if Zero)</td>
</tr>
<tr>
<td>8n POP @Rn (Pop indirect)</td>
<td>07 BNZ ea (Branch if NonZero)</td>
</tr>
<tr>
<td>9n STP @Rn (Store pop indirect)</td>
<td>08 BM1 ea (Branch if Minus 1)</td>
</tr>
<tr>
<td>An ADD Rn (Add)</td>
<td>09 BNM1 ea (Branch if Not Minus 1)</td>
</tr>
<tr>
<td>Bn SUB Rn (Sub)</td>
<td>0A BK (Break)</td>
</tr>
<tr>
<td>Cn POPD @Rn (Pop double indirect)</td>
<td>0B RS (Return from Subroutine)</td>
</tr>
<tr>
<td>Dn CPR Rn (Compare)</td>
<td>0C BS ea (Branch to Subroutine)</td>
</tr>
<tr>
<td>En INR Rn (Increment)</td>
<td>0D (Unassigned)</td>
</tr>
<tr>
<td>Fn DCR Rn (Decrement)</td>
<td>0E (Unassigned)</td>
</tr>
<tr>
<td></td>
<td>0F (Unassigned)</td>
</tr>
</tbody>
</table>
REGISTER OPS

SET Rn, Constant

\[
\begin{array}{ccc}
\text{ln} & \text{low} & \text{high} \\
\text{constant} & & \\
\end{array}
\]

(Set)

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

\[
\begin{array}{c}
2n \\
\end{array}
\]

(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

\[
\begin{array}{c}
3n \\
\end{array}
\]

(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
45

SET R5,A034
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
16 22 90

SET R5, A034
SET R6, 9022
45

Load pointers R5 and R6 with A034 and 9022.

LD @R5

Move a byte from location A034 to location 9022. Both pointers are incremented.

56

ST @R6
LDD @Rn

\[ 6n \]  

(Load double-byte indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn and Rn is then incremented by 1. The high order ACC byte is loaded from the memory location whose address resides in the (incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the final ACC contents. The carry is cleared.

Example

15 34 A0  SET R5, A034
65       LDD @R5  The low-order ACC byte is loaded from location A034, the high-order byte from location A035. R5 is incremented to A036.

STD @Rn

\[ 7n \]  

(Store double-byte indirect)

The low-order ACC byte is stored into the memory location whose address resides in Rn and Rn is then incremented by 1. The high-order ACC byte is stored into the memory location whose address resides in (the incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the ACC contents which are not disturbed. The carry is cleared.

Example

15 34 A0  SET R5, A034
16 22 90  SET R6, 9022
65       LDD @R5
76       STD @R6  Load pointers R5 and R6 with A034 and 9022. Move double byte from locations A034 and A035 to locations 9022 and 9023. Both pointers are incremented by 2.
POP @Rn

(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
10 04 00
35
10 05 00
35
10 06 00
35
85
85
85

SET R5, A034 Init stack pointer.
SET R0, 4 Load 4 into ACC.
ST @R5 Push 4 onto stack.
SET R0, 5 Load 5 into ACC.
ST @R5 Push 5 onto stack.
SET R0, 6 Load 6 into ACC.
ST @R5 Push 6 onto stack.
POP @R5 Pop 6 off stack into ACC.
POP @R5 Pop 5 off stack.
POP @R5 Pop 4 off stack.
STP @Rn

(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
to 9021.
95          STP @R5   Move byte from A032
to 9020.
ADD Rn  \\
\text{An} \quad \quad \quad \quad \quad (\text{Add})

The contents of Rn are added to the contents of the ACC (RO) 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 \hspace{1cm} \text{SET RO, 7634} \quad \text{Init RO (ACC)}
11 27 42 \hspace{1cm} \text{SET R1, 4227} \quad \text{and R1.}
A1 \hspace{1cm} \text{ADD R1} \quad \text{Add R1 (sum = B85B, carry clear)}
A0 \hspace{1cm} \text{ADD RO} \quad \text{Double ACC (RO) to 70B6 with carry set.}
SUB Rn   -- Rn

(Subtract)

The contents of Rn are subtracted from the ACC contents by performing a two's complement addition:

\[
\text{ACC} \leftarrow \text{ACC} + \overline{Rn} + 1
\]

The low order 16 bits of the subtraction are restored in the ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents. 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. Rn is not disturbed.

Example

10 34 76 \hspace{1em} \text{SET} \hspace{0.5em} R0, \hspace{0.5em} 7634 \hspace{0.5em} \text{Init} \hspace{0.5em} R0 \hspace{0.5em} (\text{ACC})
11 27 42 \hspace{1em} \text{SET} \hspace{0.5em} R1, \hspace{0.5em} 4227 \hspace{0.5em} \text{and} \hspace{0.5em} R1.
A1 \hspace{1em} \text{SUB} \hspace{0.5em} R1 \hspace{0.5em} \text{Subtract} \hspace{0.5em} R1 \hspace{0.5em} (\text{diff} = 340D \hspace{0.5em} \text{with carry set})
A0 \hspace{1em} \text{SUB} \hspace{0.5em} R0 \hspace{0.5em} \text{Clears} \hspace{0.5em} \text{ACC} \hspace{0.5em} (R0)
POP D @Rn  

\[
\begin{array}{c}
\text{Cn} \\
\end{array}
\]  

(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

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 incr R5 to A035.
      INR R5  Incr R5 to A036
55      ST @R5  Clears loc A036 (not A035)

DCR Rn

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

\[
\begin{array}{c}
\text{00}
\end{array}
\]
(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

\[
\begin{array}{c}
\text{01} \quad \text{d}
\end{array}
\]
(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 two's 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.

\[
\begin{align*}
d &= 80 & ea &= PC + 2 - 128 \\
d &= 81 & ea &= PC + 2 - 127 \\
d &= FF & ea &= PC + 2 - 1 \\
d &= 00 & ea &= PC + 2 + 0 \\
d &= 01 & ea &= PC + 2 + 1 \\
d &= 7E & ea &= PC + 2 + 126 \\
d &= 7F & ea &= PC + 2 + 127
\end{align*}
\]

Example

\[
\begin{array}{c}
\text{300: 01 50} & \text{BR } \text{352}
\end{array}
\]
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 pointer.
D5 CPR R5
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  
\[ \text{06} \quad \text{d} \quad \]  
(Branch if Zero)
A branch is effected only if the prior 'result' was zero.  
Branch conditions are not changed.

BNZ ea  
\[ \text{07} \quad \text{d} \quad \]  
(Branch if NonZero)
A branch is effected only if the prior 'result' was non-zero. Branch conditions are not changed.

BM1 ea  
\[ \text{08} \quad \text{d} \quad \]  
(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  
\[ \text{09} \quad \text{d} \quad \]  
(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  
\[ \text{0A} \quad \]  
(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

0B
(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

0C 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 ensure 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 inadvertent 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 #ADRHI  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 $0E and $0F, 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.
45 P.M., 10/3/1977

***************
* APPLE-II PSEUDO *
* MACHINE INTERPRETER *
* COPYRIGHT 1977 *
* APPLE COMPUTER INC *
* ALL RIGHTS RESERVED *
* S. WOZNIAK *
*
***************

TITLE "SWEET16 INTERPRETER"

```
ROL 20 E9 FF 24 PLA
ROL 00 E9 FF 25 STA R15L
F60D: 65 5 1 26 PLA
F60F: 68 27 STA R15H
F61: 85 1F 28 STA R15H
F692: 1E 29 SW16B
F695: 92 30 INC R15L
F698: 86 31 SW16C
F69A: 9F 32 INC R15H
F69D: 86 33 INC R15H
F69F: A7 34 SW16D
F6A0: 4B 35 LDA #5
F6A1: 00 36 LDA (R15L),Y
F6A3: 81 37 AMU #F
F6A5: 20 38 A5L A
F6A7: 0A 39 TAX
F6A8: 40 40 LSR A
F6AB: 41 41 EOR (R15L),Y
F6AC: 51 42 BEQ TBDR
F6AD: 0B 43 STX R14H
F6AE: 1D 44 LSR A
F6AF: 5D 45 LSR A
F6B0: 4A 46 LSR A
F6B2: 4A 47 TAY
F6B3: 43 48 LDA OPTBL-2,Y
F6B4: E9 E1 FF 49 PHA
F6B7: 4D 50 RTS
F6B8: 6B 51 INC R15L
F6BB: 6E 52 TBDR
F6BD: D0 53 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

DOUBLE FOR 2-BYTE REG'S TO X-REG FOR INDEXING

NOW HAVE_OPCODE IF ZERO THEN NON-REG_OP
INUCATE PRIOR RESULT REG'

OPCODE#2 TO LSB'S

TO Y-REG FOR INDEXING

LOW-ORDER ADR BYTE

ON TO STACK

GOTO REG-OP ROUTINE

INCR PC
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F66F</td>
<td>BD E4 F6 55</td>
<td>TOER2: LDA BRTBL, X</td>
</tr>
<tr>
<td>F670</td>
<td>48</td>
<td>PLA: LOW-ORDER ADDR BYTE</td>
</tr>
<tr>
<td>F672</td>
<td>A5 1D 57</td>
<td>PHA: ONI0 STACK FOR NON-REG OP</td>
</tr>
<tr>
<td>F674</td>
<td>4A 58</td>
<td>LDA R14H: 'PRIOR RESULT REG' INDEX</td>
</tr>
<tr>
<td>F676</td>
<td>60</td>
<td>LSR A: PREPARE CARRY FOR BC, ENC.</td>
</tr>
<tr>
<td>F678</td>
<td>63</td>
<td>RTS: DTO NON-REG OP HOU1NL</td>
</tr>
<tr>
<td>F67B</td>
<td>68</td>
<td>RTNZ: POP RETURN ADDRESS</td>
</tr>
<tr>
<td>F67D</td>
<td>68</td>
<td>PLA:lä</td>
</tr>
<tr>
<td>F67F</td>
<td>68</td>
<td>PLA:lä</td>
</tr>
<tr>
<td>F681</td>
<td>20 3F FF 62</td>
<td>JSR RESTORE: RESTORE 6502 REG CONTENTS</td>
</tr>
<tr>
<td>F683</td>
<td>6C 1E 00 63</td>
<td>JMP (R15L): RETURN TO 6502 CODE VIA PC</td>
</tr>
<tr>
<td>F685</td>
<td>B1 1E 64</td>
<td>SETZ: LDA (R15L), Y</td>
</tr>
<tr>
<td>F687</td>
<td>95 01 65</td>
<td>STA RCH, X: HIGH-ORDER BYTE OF CONST</td>
</tr>
<tr>
<td>F689</td>
<td>38 66</td>
<td>DEY:</td>
</tr>
<tr>
<td>F68B</td>
<td>B1 1E 67</td>
<td>LDA (R15L), Y: LOW-ORDER BYTE OF CONSTANT</td>
</tr>
<tr>
<td>F68D</td>
<td>95 00 68</td>
<td>STA ROL, X: Y-REG CONTAINS 1</td>
</tr>
<tr>
<td>F68F</td>
<td>93 69</td>
<td>TYA:</td>
</tr>
<tr>
<td>F691</td>
<td>38 70</td>
<td>SEC:</td>
</tr>
<tr>
<td>F693</td>
<td>65 1E 71</td>
<td>ADC R15L: ADD 2 TO PC</td>
</tr>
<tr>
<td>F695</td>
<td>85 1E 72</td>
<td>STA R15L:</td>
</tr>
<tr>
<td>F697</td>
<td>90 02 73</td>
<td>BCC SET2:</td>
</tr>
<tr>
<td>F699</td>
<td>E6 1F 74</td>
<td>INC R15H:</td>
</tr>
<tr>
<td>F69B</td>
<td>60 75</td>
<td>RTS: SET2</td>
</tr>
<tr>
<td>F69D</td>
<td>02 76</td>
<td>OPTBL: DBF SET-1 (1X)</td>
</tr>
<tr>
<td>F69F</td>
<td>F9 77</td>
<td>ERTBL: DBF RTN-1 (0)</td>
</tr>
<tr>
<td>F6A1</td>
<td>C4 78</td>
<td>DFB: LD-1:</td>
</tr>
<tr>
<td>F6A3</td>
<td>9D 79</td>
<td>DBF: BR-1:</td>
</tr>
<tr>
<td>F6A5</td>
<td>C0 80</td>
<td>DBF: ST-1:</td>
</tr>
<tr>
<td>F6A7</td>
<td>9E 81</td>
<td>DBF: BNC-1:</td>
</tr>
<tr>
<td>F6A9</td>
<td>25 82</td>
<td>DBF: LDAT-1:</td>
</tr>
<tr>
<td>F6AB</td>
<td>16 84</td>
<td>DBF: BC-1:</td>
</tr>
<tr>
<td>F6AC</td>
<td>B2 85</td>
<td>DBF: LDAT-1:</td>
</tr>
<tr>
<td>F6AD</td>
<td>47 86</td>
<td>DBF: STAT-1:</td>
</tr>
<tr>
<td>F6AE</td>
<td>B9 87</td>
<td>DBF: BM-1:</td>
</tr>
<tr>
<td>F6AF</td>
<td>51 88</td>
<td>DBF: STDAT-1:</td>
</tr>
<tr>
<td>F6B1</td>
<td>C0 89</td>
<td>DBF: BP-1:</td>
</tr>
<tr>
<td>F6B3</td>
<td>C9 91</td>
<td>DBF: ENZ-1:</td>
</tr>
<tr>
<td>F6B5</td>
<td>5B 92</td>
<td>DBF: STPAT-1:</td>
</tr>
<tr>
<td>F6B7</td>
<td>D2 93</td>
<td>DBF: BM1-1:</td>
</tr>
<tr>
<td>F6B9</td>
<td>85 94</td>
<td>DBF: ADD-1:</td>
</tr>
<tr>
<td>F6BB</td>
<td>DD 95</td>
<td>DBF: BN1M1-1:</td>
</tr>
<tr>
<td>F6BC</td>
<td>6E 96</td>
<td>DBF: SUB-1:</td>
</tr>
<tr>
<td>F6BD</td>
<td>05 97</td>
<td>DBF: BK-1:</td>
</tr>
<tr>
<td>F6BF</td>
<td>33 98</td>
<td>DBF: POPD-1:</td>
</tr>
<tr>
<td>F6C0</td>
<td>E8 99</td>
<td>DBF: RS-1:</td>
</tr>
<tr>
<td>F6C2</td>
<td>70 100</td>
<td>DBF: CPR-1:</td>
</tr>
<tr>
<td>F6C4</td>
<td>93 101</td>
<td>DBF: BS-1:</td>
</tr>
<tr>
<td>F6C6</td>
<td>1E 102</td>
<td>DBF: INR-1:</td>
</tr>
<tr>
<td>F6C8</td>
<td>E7 103</td>
<td>DBF: NUL-1:</td>
</tr>
<tr>
<td>F6CA</td>
<td>65 104</td>
<td>DBF: DCR-1:</td>
</tr>
<tr>
<td>F6CC</td>
<td>E7 105</td>
<td>DBF: NUL-1:</td>
</tr>
<tr>
<td>F6CE</td>
<td>701</td>
<td>DBF: NUL-1:</td>
</tr>
<tr>
<td>F6CF</td>
<td>702</td>
<td>DBF: NUL-1:</td>
</tr>
<tr>
<td>F703</td>
<td>10 CA 108</td>
<td>SET BPL SET1: ALWAYS TAKEN</td>
</tr>
<tr>
<td>Line</td>
<td>Address</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>B5 00</td>
<td>LD</td>
</tr>
<tr>
<td>2</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>EK</td>
</tr>
<tr>
<td>4</td>
<td>00 111</td>
<td>STA</td>
</tr>
<tr>
<td>5</td>
<td>70 01</td>
<td>LDA</td>
</tr>
<tr>
<td>6</td>
<td>01 113</td>
<td>STA</td>
</tr>
<tr>
<td>7</td>
<td>0D 114</td>
<td>RTS</td>
</tr>
<tr>
<td>8</td>
<td>70 0E</td>
<td>A5 00</td>
</tr>
<tr>
<td>9</td>
<td>115</td>
<td>ST</td>
</tr>
<tr>
<td>10</td>
<td>116</td>
<td>LDA</td>
</tr>
<tr>
<td>11</td>
<td>112</td>
<td>STA</td>
</tr>
<tr>
<td>12</td>
<td>70 117</td>
<td>RTS</td>
</tr>
<tr>
<td>13</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>70 17</td>
<td>A5 00</td>
</tr>
<tr>
<td>15</td>
<td>120</td>
<td>LDA</td>
</tr>
<tr>
<td>16</td>
<td>71 00</td>
<td>121</td>
</tr>
<tr>
<td>17</td>
<td>121</td>
<td>STA</td>
</tr>
<tr>
<td>18</td>
<td>118</td>
<td>MOVE RX TO RO</td>
</tr>
<tr>
<td>19</td>
<td>71 1B</td>
<td>A0 00</td>
</tr>
<tr>
<td>20</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>71 1D</td>
<td>84 10</td>
</tr>
<tr>
<td>22</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>71 1F</td>
<td>F6 00</td>
</tr>
<tr>
<td>24</td>
<td>124</td>
<td>INC</td>
</tr>
<tr>
<td>25</td>
<td>72 21</td>
<td>D0 02</td>
</tr>
<tr>
<td>26</td>
<td>125</td>
<td>INK2</td>
</tr>
<tr>
<td>27</td>
<td>72 23</td>
<td>F6 01</td>
</tr>
<tr>
<td>28</td>
<td>126</td>
<td>INC</td>
</tr>
<tr>
<td>29</td>
<td>72 25</td>
<td>60 00</td>
</tr>
<tr>
<td>30</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>72 26</td>
<td>A1 00</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>LDA</td>
</tr>
<tr>
<td>33</td>
<td>72 28</td>
<td>85 00</td>
</tr>
<tr>
<td>34</td>
<td>129</td>
<td>ROL</td>
</tr>
<tr>
<td>35</td>
<td>72 2A</td>
<td>A0 00</td>
</tr>
<tr>
<td>36</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>72 2C</td>
<td>84 01</td>
</tr>
<tr>
<td>38</td>
<td>131</td>
<td>ROH</td>
</tr>
<tr>
<td>39</td>
<td>72 2D</td>
<td>E0 00</td>
</tr>
<tr>
<td>40</td>
<td>132</td>
<td>DRO</td>
</tr>
<tr>
<td>41</td>
<td>72 30</td>
<td>A0 00</td>
</tr>
<tr>
<td>42</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>72 32</td>
<td>F0 00</td>
</tr>
<tr>
<td>44</td>
<td>134</td>
<td>BEQ</td>
</tr>
<tr>
<td>45</td>
<td>72 34</td>
<td>20 10 F7 135</td>
</tr>
<tr>
<td>46</td>
<td>135</td>
<td>JSR</td>
</tr>
<tr>
<td>47</td>
<td>72 37</td>
<td>A1 00</td>
</tr>
<tr>
<td>48</td>
<td>136</td>
<td>(ROL, X)</td>
</tr>
<tr>
<td>49</td>
<td>72 39</td>
<td>A3 00</td>
</tr>
<tr>
<td>50</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>72 3A</td>
<td>20 66 F7 138</td>
</tr>
<tr>
<td>52</td>
<td>139</td>
<td>JSR</td>
</tr>
<tr>
<td>53</td>
<td>72 3D</td>
<td>A1 00</td>
</tr>
<tr>
<td>54</td>
<td>138</td>
<td>(ROL, X)</td>
</tr>
<tr>
<td>55</td>
<td>72 3F</td>
<td>85 00</td>
</tr>
<tr>
<td>56</td>
<td>140</td>
<td>ROL</td>
</tr>
<tr>
<td>57</td>
<td>72 41</td>
<td>84 01</td>
</tr>
<tr>
<td>58</td>
<td>141</td>
<td>ROH</td>
</tr>
<tr>
<td>59</td>
<td>72 43</td>
<td>A0 00</td>
</tr>
<tr>
<td>60</td>
<td>142</td>
<td>LDY</td>
</tr>
<tr>
<td>61</td>
<td>72 45</td>
<td>84 10</td>
</tr>
<tr>
<td>62</td>
<td>143</td>
<td>R14H</td>
</tr>
<tr>
<td>63</td>
<td>72 47</td>
<td>60 00</td>
</tr>
<tr>
<td>64</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>72 48</td>
<td>20 26 F7 145</td>
</tr>
<tr>
<td>66</td>
<td>145</td>
<td>JSR</td>
</tr>
<tr>
<td>67</td>
<td>72 4B</td>
<td>A1 00</td>
</tr>
<tr>
<td>68</td>
<td>146</td>
<td>(ROL, X)</td>
</tr>
<tr>
<td>69</td>
<td>72 4D</td>
<td>85 01</td>
</tr>
<tr>
<td>70</td>
<td>147</td>
<td>ROH</td>
</tr>
<tr>
<td>71</td>
<td>72 4E</td>
<td>4C 1F F7 148</td>
</tr>
<tr>
<td>72</td>
<td>148</td>
<td>JMP</td>
</tr>
<tr>
<td>73</td>
<td>72 52</td>
<td>20 17 F7 149</td>
</tr>
<tr>
<td>74</td>
<td>149</td>
<td>JSR</td>
</tr>
<tr>
<td>75</td>
<td>72 55</td>
<td>A5 01</td>
</tr>
<tr>
<td>76</td>
<td>150</td>
<td>ROH</td>
</tr>
<tr>
<td>77</td>
<td>72 57</td>
<td>31 00</td>
</tr>
<tr>
<td>78</td>
<td>151</td>
<td>(ROL, X)</td>
</tr>
<tr>
<td>79</td>
<td>72 59</td>
<td>4C 1F F7 152</td>
</tr>
<tr>
<td>80</td>
<td>152</td>
<td>JMP</td>
</tr>
<tr>
<td>81</td>
<td>72 5C</td>
<td>20 66 F7 153</td>
</tr>
<tr>
<td>82</td>
<td>153</td>
<td>JSR</td>
</tr>
<tr>
<td>83</td>
<td>72 5F</td>
<td>05 00</td>
</tr>
<tr>
<td>84</td>
<td>154</td>
<td>ROL</td>
</tr>
<tr>
<td>85</td>
<td>72 61</td>
<td>81 00</td>
</tr>
<tr>
<td>86</td>
<td>155</td>
<td>(ROL, X)</td>
</tr>
<tr>
<td>87</td>
<td>72 63</td>
<td>4C 43 F7 156</td>
</tr>
<tr>
<td>88</td>
<td>156</td>
<td>JMP</td>
</tr>
<tr>
<td>89</td>
<td>72 66</td>
<td>E5 00</td>
</tr>
<tr>
<td>90</td>
<td>157</td>
<td>ROH</td>
</tr>
<tr>
<td>91</td>
<td>72 68</td>
<td>D0 02</td>
</tr>
<tr>
<td>92</td>
<td>158</td>
<td>DCR2</td>
</tr>
<tr>
<td>93</td>
<td>72 6A</td>
<td>D6 01</td>
</tr>
<tr>
<td>94</td>
<td>159</td>
<td>ROH, X</td>
</tr>
<tr>
<td>95</td>
<td>72 6C</td>
<td>D6 00</td>
</tr>
<tr>
<td>96</td>
<td>160</td>
<td>DEC</td>
</tr>
<tr>
<td>97</td>
<td>72 6E</td>
<td>60 00</td>
</tr>
<tr>
<td>98</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>72 6F</td>
<td>A0 00</td>
</tr>
<tr>
<td>100</td>
<td>162</td>
<td>SUB</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SWEETIE INTERFASTER

1: 45 P.M., 10/6/1977

F7/1: 38 163 164 165 166 'SEC
F7/2: 55 162 163 164 165 'LDA ROL
F7/4: 55 162 163 164 165 'SBC ROL
F7/6: 99 162 163 164 165 'STA ROL
F7/7: 167 165 'LDA RCH
F7/8: 167 166 'SBC RCH
F7/9: 167 165 'STA RCH
F7/10: 169 'TYA
F7/11: 169 'ADCB 0
F7/13: 172 'STAR14H
F7/15: 170 'RTS
F7/16: 170 'ADD
F7/17: 170 'ROR X TO RO
F7/18: 171 'ADCRX TO RO
F7/19: 172 'LDA ROL
F7/20: 173 'ADCROL
F7/21: 174 'RTS
F7/22: 175 'ADD
F7/23: 176 'ROR X TO RO
F7/24: 177 'LDA RCH
F7/25: 178 'RTS
F7/26: 179 'LDY
F7/27: 180 'FINISH ADD
F7/28: 181 'LDA R15L
F7/29: 182 'NOTE X-REG IS 12*2!
F7/30: 183 'JSR STAT2
F7/31: 184 'PUSH LOW PC BYTE VIA R12
F7/32: 185 'LDA R15H
F7/33: 186 'JSR STAT2
F7/34: 187 'PUSH HIGH-ORDER PC BYTE
F7/35: 188 'BRCLC
F7/36: 189 'EC
F7/37: 190 'BC
F7/38: 191 'BC
F7/39: 192 'BPL BR2
F7/40: 193 'DEY
F7/41: 194 'ADCR15H
F7/42: 195 'RTS
F7/43: 196 'BC
F7/44: 197 'BC
F7/45: 198 'BC
F7/46: 199 'ASL A
F7/47: 200 'DOUBLE RESULT-REG INDEX
F7/48: 201 'TO X-REG FOR INDEXING
F7/49: 202 'TEST FOR PLUS
F7/50: 203 'BRANCH IF SO
F7/51: 204 'RTS
F7/52: 205 'TEST FOR MINUS
F7/53: 206 'BRANCH IF SO
F7/54: 207 'RTS
F7/55: 208 'ASL A
F7/56: 209 'DOUBLE RESULT-REG INDEX
F7/57: 210 'TEST FOR ZERO
F7/58: 211 'BRANCH IF SO
F7/59: 212 'RTS
F7/60: 213 'TEST FOR NONZERO
F7/61: 214 'ASL A
F7/62: 215 'DOUBLE RESULT-REG INDEX
F7/63: 216 'RTS
SWEET 16 INTERPRETER

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

PAGE: 5

F/V0: 15 01  217  CMA  ROH. X   (BOTH BYTES)
F/V1: DO LF  218  ENE BR1   BRANCH IF SO
F/V2: 60 219  RTS
F/V3: 0A 220  BMI1  ASL A   DOUBLE RESULT-REG INDEX
F/V4: AA 221  TAX
F/V5: BB 00  222  LDA ROL. X   CHECK BOTH BYTES
F/V7: 35 '01  223  TAND ROH. X   FOR $FF (MINUS 1)
F/V9: 49 FF  224  EOR #$FF
F/VB: FO C4  225  DEQ BR1   BRANCH IF SO
F/VD: 60 226  RTS
F/VE: 0A 227  BNM1  ASL A   DOUBLE RESULT-REG INDEX
F/VF: AA 228  TAX
F/V0: BB 00  229  LDA ROL. X   CHK BOTH BYTES FOR NO $FF
F/V2: 35 01  230  ANU ROH. X
F/V4: 49 F F  231  EOR #$FF
F/V6: DO F9  232  ENE BR1   BRANCH IF NOT MINUS 1
F/V8: 60 233  NULL  RTS
F/V9: A2 18  234  RXL  JSR DCR   DECR STACK POINTER
F/VA: 20 66 F7  235  LDA (ROH. X)   POP HIGH RETURNADR TO PC
F/VB: A1 00  236  STX R15H
F/V3: 20 66 F7  238  JSR DCR   SAME FOR LOW-ORDER BYTE
F/V5: A1 00  239  LDA (ROH. X)
F/V7: 85 1E  240  STA R15L
F/V9: 60 241  RTS
F/VA: 4C C7 F6  242  RTN   JMP RTNZ

***** SUCCESSFUL ASSEMBLY: NO ERRORS
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>F7E6</td>
<td>0074</td>
</tr>
<tr>
<td>BC</td>
<td>F7B0</td>
<td>0083</td>
</tr>
<tr>
<td>BK</td>
<td>F7B6</td>
<td>0097</td>
</tr>
<tr>
<td>BM</td>
<td>F7BA</td>
<td>0087</td>
</tr>
<tr>
<td>BM1</td>
<td>F7B3</td>
<td>0093</td>
</tr>
<tr>
<td>ENC</td>
<td>F79F</td>
<td>0001</td>
</tr>
<tr>
<td>ENC2</td>
<td>F7AF</td>
<td>0136</td>
</tr>
<tr>
<td>L/BM1</td>
<td>F7DE</td>
<td>0075</td>
</tr>
<tr>
<td>LNZ</td>
<td>F7CA</td>
<td>0091</td>
</tr>
<tr>
<td>BP</td>
<td>F7B3</td>
<td>0055</td>
</tr>
<tr>
<td>BR</td>
<td>F7E7</td>
<td>0079</td>
</tr>
<tr>
<td>BR1</td>
<td>F7A1</td>
<td>0201</td>
</tr>
<tr>
<td>BR2</td>
<td>F7A6</td>
<td>0138</td>
</tr>
<tr>
<td>BRN/TBL</td>
<td>F6E4</td>
<td>0055</td>
</tr>
<tr>
<td>BS</td>
<td>F794</td>
<td>0101</td>
</tr>
<tr>
<td>B7</td>
<td>F7C1</td>
<td>0089</td>
</tr>
<tr>
<td>CIR</td>
<td>F771</td>
<td>0100</td>
</tr>
<tr>
<td>DCR</td>
<td>F766</td>
<td>0104</td>
</tr>
<tr>
<td>DCN2</td>
<td>F76C</td>
<td>0150</td>
</tr>
<tr>
<td>INR</td>
<td>F71F</td>
<td>0102</td>
</tr>
<tr>
<td>INR2</td>
<td>F725</td>
<td>0125</td>
</tr>
<tr>
<td>LD</td>
<td>F705</td>
<td>0078</td>
</tr>
<tr>
<td>LDAT</td>
<td>F726</td>
<td>0062</td>
</tr>
<tr>
<td>LDUDAT</td>
<td>F748</td>
<td>0056</td>
</tr>
<tr>
<td>NH1</td>
<td>F7E8</td>
<td>0103</td>
</tr>
<tr>
<td>OPT/TBL</td>
<td>F6E3</td>
<td>0049</td>
</tr>
<tr>
<td>PUP</td>
<td>F730</td>
<td>0090</td>
</tr>
<tr>
<td>PUP2</td>
<td>F73A</td>
<td>0134</td>
</tr>
<tr>
<td>PUP3</td>
<td>F743</td>
<td>0156</td>
</tr>
<tr>
<td>PUP4</td>
<td>F734</td>
<td>0098</td>
</tr>
<tr>
<td>ROL1</td>
<td>0001Z</td>
<td>0065</td>
</tr>
<tr>
<td>ROL2</td>
<td>0000Z</td>
<td>0068</td>
</tr>
<tr>
<td>R14H</td>
<td>001DZ</td>
<td>0044</td>
</tr>
<tr>
<td>R15H</td>
<td>001FZ</td>
<td>0023</td>
</tr>
<tr>
<td>R15L</td>
<td>001EZ</td>
<td>0026</td>
</tr>
<tr>
<td>RESTORE</td>
<td>F73F</td>
<td>0062</td>
</tr>
<tr>
<td>R6</td>
<td>F7E9</td>
<td>0099</td>
</tr>
<tr>
<td>RTN</td>
<td>F7FA</td>
<td>0077</td>
</tr>
<tr>
<td>RTNZ</td>
<td>F6C7</td>
<td>0242</td>
</tr>
<tr>
<td>S16PA6</td>
<td>00F7</td>
<td>0034</td>
</tr>
<tr>
<td>SAVE</td>
<td>F74A</td>
<td>0024</td>
</tr>
<tr>
<td>SET</td>
<td>F703</td>
<td>0076</td>
</tr>
<tr>
<td>SLT2</td>
<td>F6E2</td>
<td>0073</td>
</tr>
<tr>
<td>SEL2</td>
<td>F6CF</td>
<td>0108</td>
</tr>
<tr>
<td>ST</td>
<td>F70E</td>
<td>0080</td>
</tr>
<tr>
<td>STAT</td>
<td>F717</td>
<td>0084</td>
</tr>
<tr>
<td>STAT2</td>
<td>F719</td>
<td>0182</td>
</tr>
<tr>
<td>STAT3</td>
<td>F71D</td>
<td>0132</td>
</tr>
<tr>
<td>SUAT</td>
<td>F752</td>
<td>0008</td>
</tr>
<tr>
<td>SUHAT</td>
<td>F75C</td>
<td>0092</td>
</tr>
<tr>
<td>SUB</td>
<td>F76F</td>
<td>0096</td>
</tr>
<tr>
<td>SUB2</td>
<td>F77D</td>
<td>0180</td>
</tr>
<tr>
<td>SW16</td>
<td>F639</td>
<td>0098</td>
</tr>
<tr>
<td>SW16B</td>
<td>F632</td>
<td>0430</td>
</tr>
<tr>
<td>SW16C</td>
<td>F638</td>
<td>0029</td>
</tr>
<tr>
<td>SW16D</td>
<td>F632</td>
<td>0032</td>
</tr>
<tr>
<td>Code Reference</td>
<td>Opcode</td>
<td>CR3-20</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>SBR</td>
<td>F6B9</td>
<td>0043</td>
</tr>
<tr>
<td>SBR2</td>
<td>F6BF</td>
<td>0053</td>
</tr>
</tbody>
</table>
The Woz Wonderbook

DOCUMENT

Apple-II

6502 Code Relocation Program

14 November 1977

This page is not part of the original Wonderbook
A 6502

CODE RELLOCATION 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

$800

Program runs from location $400 on APPLE-I

$B87

Relocation

$C00

Relocated version runs from location $D000 on APPLE-II

$F87

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

ADR - SOURCE BLOCK BEGIN + 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

\[
\begin{array}{|c|}
\hline
$800 & \text{Code Segment} \\
800-892 \\
$893-992 \\
993-$ABF \\
$AC0-ACF \\
\hline
\end{array}
\]

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 YC (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 YC *

   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 YC

First Segment (if DATA)

* DEST SEGMENT BEG = SOURCE SEGMENT BEG . END M

Subsequent segments (if CODE)

* . SOURCE SEGMENT END YC (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 ommitted 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.

<table>
<thead>
<tr>
<th>SOURCE SEGMENTS</th>
<th>DEST SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800 \rightarrow$ CODE $\rightarrow$ $A00$ CODE $\rightarrow$ $A00$-$_{SA8F}$</td>
<td></td>
</tr>
<tr>
<td>$800$-$88F$</td>
<td>$A00$-$88F$</td>
</tr>
<tr>
<td>DATA $\rightarrow$ DATA $\rightarrow$ $A90$-$AAF$</td>
<td></td>
</tr>
<tr>
<td>$890$-$8AF$</td>
<td>$A90$-$AAF$</td>
</tr>
<tr>
<td>CODE $\rightarrow$ CODE $\rightarrow$ $AB0$-$B0F$</td>
<td></td>
</tr>
<tr>
<td>$8B0$-$890F$</td>
<td>$AB0$-$B0F$</td>
</tr>
<tr>
<td>DATA $\rightarrow$ DATA $\rightarrow$ $B10$-$B3F$</td>
<td></td>
</tr>
<tr>
<td>$910$-$93F$</td>
<td>$B10$-$B3F$</td>
</tr>
<tr>
<td>CODE $\rightarrow$ CODE $\rightarrow$ $B40$-$B7F$</td>
<td></td>
</tr>
<tr>
<td>$940$-$97F$</td>
<td>$B40$-$B7F$</td>
</tr>
</tbody>
</table>

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 YC *
(c) Relocate first segment (code).
   * A00 < 800 . 88F YC
(d) Move and relocate subsequent segments in order.

* . 8AF M (data)
* . 90F YC (code)
* . 93F M (data)
* . 97F YC (code)

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

* A90 < 890 . 8AF M (data)
* ABO < 8B0 . 90F YC (code)
* B10 < 910 . 93F M (data)
* B40 < 940 . 97F YC (code)
2. Index into block

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

\[ \text{LDA 7B0,X} \]

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 LOCO
    LDA #$08
    STA LOC1
    JMP (LOCO)

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 (YC) programs

Relocating programs such as RELOC introduces another irregularity. Because RELOC uses the MONITOR user function command (YC) 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 YC 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**

<table>
<thead>
<tr>
<th>Locations during relocation</th>
<th>Runs from locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S000$</td>
<td>$400-$78F on APPLE-I</td>
</tr>
<tr>
<td>$S800-$97F</td>
<td>but must be relocated</td>
</tr>
<tr>
<td>DATA $980-$9FF</td>
<td>to run from locations</td>
</tr>
<tr>
<td>CODE $A00-$B8F</td>
<td>$D000-$D38F on the</td>
</tr>
<tr>
<td>$B8F$</td>
<td>APPLE-II.</td>
</tr>
</tbody>
</table>

SOURCE BLOCK $400-$78F        DEST BLOCK $D000-$D38F
SOURCE SEGMENTS $S00-$B8F     DEST SEGMENTS $S00-$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 YC *
(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)
7. Split blocks with cross-referencing

Program A resides and runs in locations $800-$8A6. Program B resides and runs in locations $900-$9F1. A single, contiguous program is to be generated by moving program B so that it immediately follows program A. Each of the programs contains memory references within the other. It is assumed that the programs contain no data segments.

<table>
<thead>
<tr>
<th>SOURCE SEGMENTS</th>
<th>DEST SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800 → Program A</td>
<td>$800 → Program A</td>
</tr>
<tr>
<td>$800-$8A6</td>
<td>$800-$8A6</td>
</tr>
<tr>
<td>$8A6 → Unused</td>
<td>$8A6 → Program B</td>
</tr>
<tr>
<td>$900 → Program B</td>
<td>$8A7 → $8A7</td>
</tr>
<tr>
<td>$9F1 → $900-$9F1</td>
<td>$8A7 → $998</td>
</tr>
</tbody>
</table>

SOURCE BLOCK $900-$9F1

DEST BLOCK $8A7-$998

SOURCE SEGMENTS $800-$8A6 (A)

DEST SEGMENTS $800-$8A6 (A)

$900-$9F1 (B)

$8A7-$998 (B)

(a) Load RELOC

(b) Define blocks (program B only)

* 8A7 < 900 . 9F1 YC *
(c) Relocate each of the two programs individually. Program A must be relocated even though it does not move.

* 800 <. 8A6 YC  (program A, 'in place')
* 8A6 < 900 . 9F1 YC  (program B, not 'in place')

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


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

<table>
<thead>
<tr>
<th>SOURCE BLOCK $8C4-$97F</th>
<th>DEST BLOCK $8C0-$97B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SOURCE SEGMENTS</strong></td>
<td><strong>DEST SEGMENTS</strong></td>
</tr>
<tr>
<td>$800-$88F (code)</td>
<td>$800-$88F (code)</td>
</tr>
<tr>
<td>$890-$8AF (data)</td>
<td>$890-$8AF (data)</td>
</tr>
<tr>
<td>$8B0-$8BF (code)</td>
<td>$8B0-$8BF (code)</td>
</tr>
<tr>
<td>$8C4-$90F (code)</td>
<td>$8C0-$90B (code)</td>
</tr>
<tr>
<td>$910-$93F (data)</td>
<td>$90C-$93B (data)</td>
</tr>
<tr>
<td>$940-$97F (code)</td>
<td>$93C-$97B (code)</td>
</tr>
</tbody>
</table>
(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
(see example (2))

DEST BLOCK $700-$FFF

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^C *

(c) Segments

* 800 < F800 . F961 Y^C (first segment, code)
* . FA42 M (data)
* . FB18 Y^C (code)
* . FB1D M (data)
* . FFCB Y^C (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

   \[ \* A_d < A_1 . A_2 \ Y^C \]  \(A_1, A_2, \text{and} \ A_d \text{are addresses}\)

   vectors to location $3F8$ with the value $A_1$ in locations $3C$ (lo and $3D$ (high), $A_2$ in locations $3E$ (low) and $3F$ (high), and $A_d$ 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Preserved at</th>
<th>SWEET16 Reg Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEST BLOCK BEG</td>
<td>$8, 9$</td>
<td>TOBEG</td>
</tr>
<tr>
<td>SOURCE BLOCK BEG</td>
<td>$2, 3$</td>
<td>FRMBEG</td>
</tr>
<tr>
<td>SOURCE BLOCK END</td>
<td>$4, 5$</td>
<td>FRMEND</td>
</tr>
</tbody>
</table>

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 $A_d$ ($42, 43$) 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.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>0</td>
</tr>
<tr>
<td>1 byte</td>
<td>0</td>
</tr>
<tr>
<td>2 byte</td>
<td>1</td>
</tr>
<tr>
<td>3 byte</td>
<td>2</td>
</tr>
</tbody>
</table>

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

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

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

26 SUBTTL RELOCATION SUBR EQUATES
27 RL1L EPZ $2 SWEET16 REG 1.
28 INST EPZ $8 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 CMDN BUF.
34 SW16 EQU $F689 SWEET16 ENTRY.
35 INSNS2 EQU $F88E DISASSEMBLER ENTRY.
36 NXTA4 EQU $FCB4 POINTER INCR SUBR.
37 FMBEG 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

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

PAGE: 3

SUBTL 6502 RELOCATION SUBROUTINE

ORG $3A6

CMND BUF POINTER.

LDD YSAV

NEXT CMND CHAR.

CMP #$A

' * '?

BNE RELOC2

NO, RELOC CODE SEG.

INC YSAV

ADVANCE POINTER.

LDX #$7

MOVE BLOCK PARAMS

STA R1L,X

FROM APPLE-II MON

DEX

AREA TO SW16 AREA.

BPL INIT

R1=SOURCE BEG, R2=

RTS

SOURCE END, R4=DEST BEG.

COPY 3 BYTES TO

GETINS LDA (A1L),Y

SW16 AREA.

DEY

CALCULATE LENGTH OF

INSTR FROM OPCODE.

XRDS2

0=1 BYTE, 1=2 BYTE,

LDX LENGTH

2=3 BYTE.

DEX

WEED OUT NON-ZERO-PAGE

BNE XLATE

2 BYTE INSTRS (IMM).

BNE INST

IF ZERO PAGE ADR

AND #$8

THEN CLEAR HIGH BYTE.

BNE STINST

IF ADR OF ZERO PAGE

STA INST+2

OR ABS IS IN SOURCE

JSR SW16

(FRM) BLOCK THEN

LD FRMEND

SUBSTITUTE ADR-

CPR ADR

SOURCE BEG+DEST BEG.

SUB FMBEGIN

RTN

ADD TOBEG

SW16 AREA TO

STA (A4L),Y

COPY LENGTH BYTES

OF INST FROM

STA INST,X

SW16 AREA TO

STINS2 LDA INST,X

OF INSTR TO

STINS LDX #$0

DEST SEGMENT. UPDATE

STINST

SOURCE, DEST SEGMENT

LDX #S0

POINTERS. LOOP IF NOT

ADD TOBEG

BEYOND SOURCE SEG END.

STA (A1L),Y

ENTRY FROM MONITOR.

INX

SUCCESSFUL ASSEMBLY: NO ERRORS

ORG $3F8

USRLOC JMP RELOC

03E6: 00 79 SW16RT RTN

03E7: A2 00 80 STINST LDY #$0

03E8: 4C A6 03 30 USRLOC JMP RELOC

03E9: 91 42 82 STA (A4L),Y

03ED: E8 83 INX

03EE: 20 B4 FC 84 JSR NXTA4

03F0: 03 F3: 10 F4 86 BPL STINS2

03F5: 90 C4 87 BCC RELOC2

03F7: 60 88 RTS

03F9: 4C A6 03 90 USRLOC JMP RELOC

03F8: 4C A6 03 90 USRLOC JMP RELOC

03B8: A0 02 55 BPL GETINS

03B9: 5A 5E 03 90 USRLOC JMP RELOC

03BA: 60 54 RTS

03BB: A0 02 55 RELOC2 LDX #$2

03BD: B1 3C 56 GETINS LDA (A1L),Y

03BF: F9 0B 00 57 STA INST,Y

03C0: 88 58 DEY

03C1: 10 F8 59 GETINS BPL

03C2: 20 8E F8 60 JSR INSDS2

03C3: 06 A6 2F 61 LDX LENGTH

03C4: 02 09 62 DEX

03C5: D0 0C 63 BNE XLATE

03C6: 05 A5 64 LDA INST

03C7: 29 0D 65 AND #$D

03C8: F0 14 66 BEQ STINST

03C9: 29 08 67 AND #$8

03CA: D0 10 68 BNE STINST

03CB: 85 0D 69 STA INST+2

03CC: 20 89 F6 70 XLATE JSR SW16

03CD: 22 71 LD FRMEND

03CE: D6 72 CPR ADR

03CF: 02 06 73 BNE SW16RT

03D0: 26 74 LD ADR

03D1: B1 75 SUB FMBEGIN

03D2: 02 02 76 BNE SW16RT

03D3: A4 77 ADD TOBEG

03D4: 36 78 ST ADR

03D5: 00 79 SW16RT RTN

03D6: A2 00 80 STINST LDY #$0

03D7: B5 81 STINS2 LDA INST,X

03D8: 91 42 82 STA (A4L),Y

03D9: 9A 83 INX

03DA: 20 B4 FC 84 JSR NXTA4

03DB: C6 2F 85 DEC LENGTH

03DC: 10 F4 86 BPL STINS2

03DD: 90 C4 87 BCC RELOC2

03DE: 60 88 RTS

03DF: 4C A6 03 90 USRLOC JMP RELOC

***** SUCCESSFUL ASSEMBLY: NO ERRORS
<table>
<thead>
<tr>
<th>CROSS-REFERENCE: 6502 RELOCATION SUBROUTINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1L</td>
</tr>
<tr>
<td>A4L</td>
</tr>
<tr>
<td>ADR</td>
</tr>
<tr>
<td>FRMBEG</td>
</tr>
<tr>
<td>FRMEND</td>
</tr>
<tr>
<td>GETINS</td>
</tr>
<tr>
<td>IN</td>
</tr>
<tr>
<td>INIT</td>
</tr>
<tr>
<td>INSDS2</td>
</tr>
<tr>
<td>INST</td>
</tr>
<tr>
<td>LENGTH</td>
</tr>
<tr>
<td>NXTA4</td>
</tr>
<tr>
<td>R1L</td>
</tr>
<tr>
<td>RELOC</td>
</tr>
<tr>
<td>RELOC2</td>
</tr>
<tr>
<td>STINS2</td>
</tr>
<tr>
<td>STINST</td>
</tr>
<tr>
<td>SW16</td>
</tr>
<tr>
<td>SW16RT</td>
</tr>
<tr>
<td>TOBEG</td>
</tr>
<tr>
<td>USRLOC</td>
</tr>
<tr>
<td>XIMATE</td>
</tr>
<tr>
<td>YSAV</td>
</tr>
<tr>
<td>FILE:</td>
</tr>
</tbody>
</table>
Apple-II
Renumbering and Appending
BASIC Programs
15 November 1977
RENUMBERING AND APPENDING

BASIC PROGRAMS

on the

APPLE-II COMPUTER

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" 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.

---


* 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 develope 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 preceeded 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

LOMEM (start of variables) ($4A,4B)

PV (Variable Pointer, end of variables) ($CC,CD)

PP (Program Pointer, start of program) ($CA,CB)

first line

last line

HIMEM (end of program) ($4C,4D)

Figure 2 - LINE REPRESENTATION

length byte    low    high    items    $01
line number   'end-of-line' token
Figure 3 - ITEMS

<table>
<thead>
<tr>
<th>Constant:</th>
<th>low</th>
<th>high</th>
<th>value</th>
<th>positive token</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B0$-$B9$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name (ABC):</th>
<th>$C1$</th>
<th>$C2$</th>
<th>$C3$</th>
<th>negative ASCII</th>
<th>positive token</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>String Constant (&quot;123&quot;):</th>
<th>$28$</th>
<th>$B1$</th>
<th>$B2$</th>
<th>$B3$</th>
<th>$29$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>quote token</td>
<td>negative ASCII</td>
<td>quote token</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REM:</th>
<th>$5D$</th>
<th></th>
<th></th>
<th></th>
<th>$01$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>REM token</td>
<td>negative ASCII</td>
<td>'end-of-line' token</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tokens:</th>
<th>$00$-$7F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO TO</td>
<td>$5F$</td>
</tr>
<tr>
<td>GOSUB</td>
<td>$5C$</td>
</tr>
<tr>
<td>THEN ln</td>
<td>$24$</td>
</tr>
<tr>
<td>LIST</td>
<td>$74$</td>
</tr>
<tr>
<td>LIST ,</td>
<td>$75$</td>
</tr>
<tr>
<td>STR CON</td>
<td>$28$</td>
</tr>
<tr>
<td>REM</td>
<td>$5D$</td>
</tr>
<tr>
<td>EOL</td>
<td>$01$</td>
</tr>
</tbody>
</table>

(tokens used by RENUMBER)
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.1 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.

---

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.

Original Program

```
PP      Prog #2
|      |
|      | HIMEM
```

After Load

```
PP      Prog #1
|      |
|      | Prog #2 (hidden)
```

HIMEM Restored

```
PP      Prog #1
|      |
|      | Prog #2
```

HIMEM

```
|      |
|      | HIMEM
```
RENUMBER EXAMPLE

Original

>LIST
1 GOTO 100
2 GOSUB 103
3 IF TRUE THEN 107
4 LIST 109,110
100 REM
103 REM
107 REM
109 REM
110 REM
200 FOR I=1 TO 10
210 PRINT I
220 NEXT I
230 GOTO 1

POKE 3, 150 MOD 256
POKE 3, 150 / 256
POKE 4, 10 MOD 256
POKE 5, 10 / 256
POKE 6, 100 MOD 256
POKE 7, 100 / 256
POKE 8, 110 MOD 256
POKE 9, 110 / 256

CALL 776
100->150
103->160
107->170
109->180
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


TITLE 'APPLE-II BASIC RENUMBER/APPEND SUBROUTINES'

1  *******************************
2  *
3  * APPLE-II BASIC *
4  * RENUMBER AND APPEND *
5  * SUBROUTINES *
6  *
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 * JOZ 11/16/77 *
18 * APPLE COMPUTER INC.*
19 *
20 *******************************
21 "PAGE"

Distributed under the Creative Commons License on page 5

22 SUBTTL 6502 EQUATES
23 ROL EPZ $0
24 ROR EPZ $1
25 RRL EPZ $16
26 RRRH EPZ $17
27 HIREPZ $40
28 PPL EPZ $8A
29 PUL EPZ $8C
30 MEMFULL EQU $E368
31 PDECC EQU $E518
32 RANGE R EQU $E68
33 LOAD EQU $F0DF
34 SV16 EQU $F639
35 COUT EQU $FDBE
36 COUT EQU $FDED
37 PAGE
SWEET16 EQUATES


PAGE 3

38 SUB TTL SWEET16 EQUATES

39 ACC EPZ $0 SWEET16 ACCUMULATOR.
40 NEWLOW EPZ $1 NEW INITIAL LNO.
41 NEWINCR EPZ $2 NEW LNO INCR.
42 LNFLW EPZ $3 LOW LNO OF RENUM RANGE.
43 LNH1 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 SCR0 EPZ $8 SCRATCH REG.
48 SCR9 EPZ $9 HMEM (END OF PRGM).
49 SCR0 EPZ $10 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 TBLNO EPZ $6 PASS 2 LNO TABLE END.
55 PRGNDE2 EPZ $7 PASS 2 PROG INDEX.
56 CHR0 EPZ $9 ASCII '0'.
57 CHR1 EPZ $A ASCII 'A'.
58 MODE EPZ $0 CONST/LNO MODE.
59 TBLNDX2 EPZ $8 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 SRC0 EPZ $C SCRATCH REG FOR APPEND.
APPLE-II BASIC RENUMBER SUBROUTINE - PASS 1


68 SUBTL APPLE-II BASIC RENUMBER SUBROUTINE - PASS 1
69 ORG $300

0300: 20 89 F6 70 RENX JSR SW16
0303: 80 71 SUB ACC
0304: 33 72 ST LNW
0305: 34 73 ST LNH
0306: F4 74 DCR LNH
0307: 00 75 RTN
0308: 20 89 F6 76 RENUM JSR SW16
030B: 18 4C 00 77 SET SCR8, HIMEM
030F: 68 78 LDD @SCR8
0310: 38 79 ST HIMEM
0313: E9 81 POPD @SCR9
0314: 35 82 ST TBLSTART
0315: 36 83 ST TBLNDXI
0316: 21 84 LD NEVLOW
0317: 3B 85 ST NEVLN
0318: 3C 86 ST NEVLN
0319: C9 87 POPD @SCR9
031A: 37 88 ST TBLIM
031B: 39 89 ST PRGNDX
031C: 29 90 PASS1
031D: D8 91 CPR HIMEM
031E: 03 46 92 BC PASS2
0320: 3A 93 ST PRGNDX1
0321: 26 94 LD TBLNDXI
0322: 20 95 INR ACC
0323: D7 96 CPR TBLIM
0324: 03 38 97 BC MERR
0326: 4A 98 LD @PRGNDX1
0327: A9 99 ADD PRGNDX
0328: 39 100 ST PRGNDX
0329: 6A 101 LD @PRGNDX1
032A: D4 102 CPR LNW
032B: 02 2A 103 BNC P1B
032D: 04 104 CPR LNH
032E: 02 02 105 BNC PIA
0330: 07 30 106 BNZ PIC
0332: 76 107 PIA STD @TBLNDXI
0333: 00 108 RTN
0334: A5 01 109 LDA R0H
0336: A6 00 110 LDX ROL
0338: 20 18 E5 111 JSR PRDEC
033B: A9 AD 112 LDA $AD
033D: 20 ED FD 113 JSR COUT
0340: A9 BE 114 LDA $BE
0342: 20 ED FD 115 JSR COUT
0345: A5 17 116 LDA Ri1H
0347: A6 16 117 LDX Ri1L
0349: 20 18 E5 118 JSR PRDEC
034C: 20 8E FD 119 JSR COUT
034F: 20 8C F6 120 JSR SW16+3
0352: 28 121 LD NEVLN

+++ END 6502 CODE +++
<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0353</td>
<td>3C</td>
<td>ST</td>
<td>NEWLN1</td>
<td>COPY NEWLN TO NEWLN1</td>
</tr>
<tr>
<td>0354</td>
<td>A2</td>
<td>ADD</td>
<td>NEWINCR</td>
<td>AND INCR NEWLN BY</td>
</tr>
<tr>
<td>0355</td>
<td>3B</td>
<td>ST</td>
<td>NEWLN</td>
<td>NEWINCR.</td>
</tr>
<tr>
<td>0356</td>
<td>0D</td>
<td>NUL</td>
<td></td>
<td>(WILL SKIP NEXT INST).</td>
</tr>
<tr>
<td>0357</td>
<td>D1</td>
<td>PIB</td>
<td></td>
<td>IF LOW LNO &lt; NEWLOW</td>
</tr>
<tr>
<td>0358</td>
<td>02 02</td>
<td>BNC</td>
<td>PASSI</td>
<td>THEN RANGE ERR.</td>
</tr>
<tr>
<td>035A</td>
<td>00</td>
<td>RERR</td>
<td>RTN</td>
<td>PRINT 'RANGE ERR' MSG</td>
</tr>
<tr>
<td>035B</td>
<td>4C 68 EE</td>
<td>JMP</td>
<td>RANERR</td>
<td>AND RETURN.</td>
</tr>
<tr>
<td>035E</td>
<td>00</td>
<td>MERR</td>
<td>RTN</td>
<td>PRINT 'MEM FULL' MSG</td>
</tr>
<tr>
<td>035F</td>
<td>4C 68 E3</td>
<td>JMP</td>
<td>MEMFULL</td>
<td>AND RETURN.</td>
</tr>
<tr>
<td>0362</td>
<td>EC</td>
<td>PIC</td>
<td>INR</td>
<td>IF HI LNO &lt;= MOST RECH</td>
</tr>
<tr>
<td>0363</td>
<td>DC</td>
<td>GPR</td>
<td>NEWLN1</td>
<td>NEWLN THEN RANGE ERR</td>
</tr>
<tr>
<td>0364</td>
<td>02 F4</td>
<td>BNC</td>
<td>RERR</td>
<td>PAGE</td>
</tr>
</tbody>
</table>
APPLE-II BASIC RENUMBER SUBROUTINE - PASS 2


0366: 19 20 00 137 PASS2
0369: 1A C1 00 138 SET CHR,80
036C: 27 139 P2A ASCII '0'
036D: D8 140 SET CHR,81
036E: 03 63 141 ASCII 'A'
0370: 07 142 LD PGNNDX2
0371: 67 143 CPR HIMEM IF PROG INDEX = HIMEM
0372: 3D 144 INR PGNNDX2 SKI LEN BYTE
0373: 25 145 LDD #PGNNDX2 LINE NUMBER
0374: 3B 146 ST TBLNDX2 INIT LNO TABLE INDEX
0375: 21 147 LD NEVLOW INIT NEVLN1 TO NEVLOW
0376: IC 00 00 148 (WILL SKIP NEXT 2 INSR
0377: 2C 150 UD2
0378: A2 151 ADD NEVNL1 INCR TO NEVLN1
0379: 3C 152 ADD NEVNL1
037A: 2B 153 ST NEVNL1
037B: B6 154 LD TBLNDX2 IF LNO TBL IDX = TBLN
037C: 03 07 155 SUB TBLND THEN DONE SCANNING
037D: 6B 156 BC UD3 LNO TABLE
037E: BD 157 SUB TBLND NEXT LNO FROM LNO TABLE
0380: 07 F5 158 BNZ UD2 IF NOT SAE
0382: C7 159 POPD #PGNNDX2 AS OLDLN
0383: 2C 160 LD NEVNL1 replace OLDLN with
0384: 77 161 STD #PGNNDX2 CORRESPONDING NEW LNO
0385: IB 28 00 162 UD3 SET STRCON,28 STR CON TOKEN
0386: IC 00 00 163 SET MODE,0 (SKIPS NEXT 2 INSR'S
0387: 67 165 GOTCON LDD #PGNNDX2
0388: FC 166 DCR MODE
0389: 08 E5 167 BM1 UPDATE
038A: 47 168 ITEM LD #PGNNDX2 BASIC ITEM
038B: D9 169 CPR CHRO
038C: DA 170 CPR CHR
038D: 02 F5 172 CPR CHKCON
038E: F7 173 SKPASC DCR PRGNDX2
038F: 67 174 LDD #PGNNDX2 SKIP ALL NEG BYTES OF
0390: 05 FC 175 BM SKPASC STR CON, REM, OR NAME
0391: F7 176 DCR PRGNDX2
0392: 49 177 LD #PGNNDX2
0393: DB 178 CHKCON CPR STRCON
0394: 06 F7 179 CPR STRCON
0395: 1C 5D 00 180 SET REM,5D
0396: DC 181 CPR RSM
0397: 06 182 SET REM,5D
0398: 08 13 183 BM1 CONTST
0399: FD 184 DCR R13
039A: 06 185 DCR R13 (TOKEN $5F IS GOTO)
039B: 06 186 BM2 CONTST
039C: DD 187 SET THEN,$24
039D: 06 188 CPR THEN
039E: 06 189 SE CONTST THEN LNO, LOOK FOR LNO.
### APPLE-II BASIC RENUMBER SUBROUTINE - PASS 2

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Value</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>03AF</td>
<td>F0</td>
<td>190</td>
<td>DCR ACC</td>
<td></td>
</tr>
<tr>
<td>03B0</td>
<td>06 BA</td>
<td>191</td>
<td>BZ P2A</td>
<td>EOL (TOKEN $01)</td>
</tr>
<tr>
<td>03B2</td>
<td>1D 74 00</td>
<td>192</td>
<td>SET LIST,$74</td>
<td></td>
</tr>
<tr>
<td>03B3</td>
<td>BD</td>
<td>193</td>
<td>SUB LIST</td>
<td>SET MODE = 0 IF LIST</td>
</tr>
<tr>
<td>03B4</td>
<td>09 01</td>
<td>194</td>
<td>BNMI CONTS2</td>
<td>OR LIST COMMA ($73,</td>
</tr>
<tr>
<td>03B5</td>
<td>BQ</td>
<td>195</td>
<td>CONTST SUB ACC</td>
<td>CLEAR MODE FOR LNO</td>
</tr>
<tr>
<td>03B6</td>
<td>JC</td>
<td>196</td>
<td>CONTS2 ST MODE</td>
<td>UPDATE CHECK</td>
</tr>
<tr>
<td>03B7</td>
<td>01 DI</td>
<td>197</td>
<td>BR ITEM</td>
<td>CHECK NEXT BASIC ITEM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>198</td>
<td>PAGE</td>
<td></td>
</tr>
</tbody>
</table>
APPLE-II BASIC APPEND SUBROUTINE


PAGE: 9

199 SUBTL APPLE-II BASIC APPEND SUBROUTINE

033C: 20 69 F6 200 APPEND JSR SW16
033F: 1C 4E 00 201 SET SCRC,HIMEM+2
03C2: C0 202 POPD #SCRC SAVE HIMEM
03C3: 38 203 ST HMEM
03C4: 19 CA 00 204 SET SC9,PPL
03C7: 69 205 LDD #SC9 SET HIMEM TO PRESERVE
03C8: 7C 206 STD #SC9 PROGRAM
03C9: 00 207 RTN
03CA: 20 DF F0 208 JSR LOAD LOAD FROM TAPE
03CD: 20 69 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
<table>
<thead>
<tr>
<th>CROSS REFERENCE</th>
<th>APPLE-II BASIC RENUMBER/APEND SUBROUTINES</th>
</tr>
</thead>
<tbody>
<tr>
<td>UD2</td>
<td>0377 0158</td>
</tr>
<tr>
<td>UD3</td>
<td>0385 0155</td>
</tr>
<tr>
<td>UPDATE</td>
<td>0372 0167</td>
</tr>
</tbody>
</table>
The Woz Wonderbook

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


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 bytecode "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.
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. Concerning authorship of this floating point package, website http://linux.monroeccc.edu/~paulrsmp/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/kiml-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
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@coastsides.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
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