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
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

Distributed under the Creative Commons License on page 5
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. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
* For any reuse or distribution, you must make clear to others the license terms of this work.
* Any of these conditions can be waived if you get permission from the copyright holder.

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

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

This page is not part of the original Wonderbook
Disclaimer

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

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

For the full legal code for this license see:

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

This page is not part of the original Wonderbook
The Woz Wonderbook

Table of Contents

Auto Repeat for Apple-II Monitor Commands
20 September 1977

Use of the Apple-II Mini-Assembler

Apple-II Pointers and Mailboxes

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

Using Apple-II Color Graphics

Adding Colors to Apple-II Hi-Res

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

Apple-II Cassette Article

Apple-II Floating Point Package

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

Apple-II 6502 Code Relocation Program
14 November 1977

Apple-II Renumbering and Appending BASIC Programs
15 November 1977

References
03 November 2004

Bill Goldberg Interview
19 April 2004

Credits

This page is not part of the original Wonderbook
The Woz Wonderbook

Auto Repeat for Apple-II Monitor Commands

20 September 1977

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\(\text{no blanks}\) (\text{\&} 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

Use of the

Apple-II Mini-Assembler

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

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

If the first character on the line is blank, the assembled instruction will be stored starting at the address immediately following the previously assembled instruction. If the first character is nonblank (and not "$"), the line is assumed to contain an assembly language instruction preceded by the instruction address (a hex number followed by a ";"). In either case, the instruction will be retyped over the line just entered in disassembler format to provide a visual check of what has been assembled. The counter that
keeps track of where the next instruction will be stored is the pseudo PC (Program Counter) and it can be changed by many
monitor commands (eg. 'L', 'T', ...). Therefore, it is advisable
to use the explicit instruction address mode after every monitor
command and, of course, when the Tiny assembler is first
entered.

Errors (unrecognized mnemonic, illegal format, etc.) are
signalled by a "beep" and a carrot ("^") will be printed be-
neath 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 pre-
   cede 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.
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
<table>
<thead>
<tr>
<th>addr</th>
<th>hex</th>
<th>dec</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td></td>
<td></td>
<td>Some buffer start point. Size class point has appeared one more.</td>
</tr>
<tr>
<td>In</td>
<td>200</td>
<td>640</td>
<td>line buffer</td>
</tr>
</tbody>
</table>

62, 6D -->

Top of AppleSoft BASIC program

73, 75

Top of variable area in AppleSoft
The Woz Wonderbook

DOCUMENT

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

This page is not part of the original Wonderbook
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 ¹⁄₄ inch of the remaining trace not still connected to pin 18. You may wish to tin it with solder since it will later be soldered to.

4. (Underside) Connect pin 18 of ROM 'D8' to pin 12 of ROM 'EO'
   (ground)

5. (underside) Connect pin 18 of ROM 'EO' to the trace which previously went to pin 18 of ROM 'D8' (and which should be pretinned if step 3 was followed).
6. (underside) Connect pin 21 of ROM 'D8' to pin 21 of ROM 'D0'. Then connect both of these to pin 24 of either ROM (VCC).

7. Note that the 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

<table>
<thead>
<tr>
<th>Y-coordinate</th>
<th>00abcdef</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE (leftmost) address</td>
<td>eabab000</td>
</tr>
<tr>
<td>Data byte</td>
<td>XXXXYYY</td>
</tr>
<tr>
<td>odd line</td>
<td>even line</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LINE</th>
<th>BASE address (hex)</th>
<th>Secondary BASE address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0,1</td>
<td>$400</td>
<td>$800</td>
</tr>
<tr>
<td>$2,3</td>
<td>$480</td>
<td>$880</td>
</tr>
<tr>
<td>$4,5</td>
<td>$500</td>
<td>$900</td>
</tr>
<tr>
<td>$6,7</td>
<td>$580</td>
<td>$980</td>
</tr>
<tr>
<td>$8,9</td>
<td>$600</td>
<td>$A00</td>
</tr>
<tr>
<td>$A,B</td>
<td>$680</td>
<td>$A80</td>
</tr>
<tr>
<td>$C,D</td>
<td>$700</td>
<td>$B00</td>
</tr>
<tr>
<td>$E,F</td>
<td>$780</td>
<td>$B80</td>
</tr>
<tr>
<td>$10,11</td>
<td>$428</td>
<td>$828</td>
</tr>
<tr>
<td>$12,13</td>
<td>$4A8</td>
<td>$8A8</td>
</tr>
<tr>
<td>$14,15</td>
<td>$528</td>
<td>$928</td>
</tr>
<tr>
<td>$16,17</td>
<td>$5A8</td>
<td>$9A8</td>
</tr>
<tr>
<td>$18,19</td>
<td>$628</td>
<td>$A28</td>
</tr>
<tr>
<td>$1A,1B</td>
<td>$6A8</td>
<td>$AA8</td>
</tr>
<tr>
<td>$1C,1D</td>
<td>$728</td>
<td>$B28</td>
</tr>
<tr>
<td>$1E,1F</td>
<td>$7A8</td>
<td>$BA8</td>
</tr>
<tr>
<td>$20,21</td>
<td>$450</td>
<td>$850</td>
</tr>
<tr>
<td>$22,23</td>
<td>$4D0</td>
<td>$8D0</td>
</tr>
<tr>
<td>$24,25</td>
<td>$550</td>
<td>$950</td>
</tr>
<tr>
<td>$26,27</td>
<td>$5D0</td>
<td>$9D0</td>
</tr>
<tr>
<td>$28,29</td>
<td>$650</td>
<td>$A50</td>
</tr>
<tr>
<td>$2A,2B</td>
<td>$6D0</td>
<td>$AD0</td>
</tr>
<tr>
<td>$2C,2D</td>
<td>$750</td>
<td>$B50</td>
</tr>
<tr>
<td>$2E,2F</td>
<td>$7D0</td>
<td>$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</th>
<th>Page Zero Variable Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBASL</td>
<td>$26$</td>
</tr>
<tr>
<td>GBASH</td>
<td>$27$</td>
</tr>
<tr>
<td>H2</td>
<td>$2C$</td>
</tr>
<tr>
<td>V2</td>
<td>$2D$</td>
</tr>
<tr>
<td>MASK</td>
<td>$2E$</td>
</tr>
<tr>
<td>COLOR</td>
<td>$30$</td>
</tr>
</tbody>
</table>

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

1. Set the video mode and scrolling window (refer to the section on APPLE-II text features)
2. Clear the screen with a call to the CLRSCR (48-line clear) or CLRTOP (40-line clear) subroutines. If you are using the mixed text/graphics feature then call CLRTOP.
3. Set the color using the SETCOLOR subroutine.
4. Call the PLOT, HLINE, and VLINE subroutines to plot points and draw lines. The color setting is not affected by these subroutines.
5. Advanced programmers may wish to study the provided subroutines and addressing schemes. When you supply x- and y-coordinate data to these subroutines they generate BASE address, horizontal index, and even/odd mask information. You can write more efficient programs if you supply this information directly.
SETCOL subroutine (address $F864)

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

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

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

Example: (select color 6)

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

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

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

PLOT subroutine (address $F800)

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

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

Exit: The A-Reg is clobbered but the Y-Reg is not. The carry is cleared. A halfbyte mask ($F or $FO) 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 ($800)
```
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 #$1A  Right
STA H2  Save it
LDA #$9  Y-coordinate
JSR HLINE  Plot line
SCRN subroutine (address $F871)

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

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

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

Calls: GBASCALC

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.
LDA #$7 X-coordinate
LDA (GBASL),Y Access byte

Note: For an even/odd Y-coord pair, the even-coord data is contained in the least significant 4 bits of the accessed byte and the odd-coord data in the most significant 4.
This page is not part of the original Wonderbook
The Woz Wonderbook

DOCUMENT

Adding Colors to Apple-II Hi-Res

This page is not part of the original Wonderbook
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).
1. Mount a 4007 (Dual 2-N Flip Flop) and a 24LS03
and wire them in the breadboard area (A12 - A14);

2. wire the following circuit (* indicates that wire goes
   to a pin which is out of its socket)
It is a fixed timing condition in VSYNC on scan standard (60 Hz)

Modified
The Woz Wonderbook

DOCUMENT

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

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 \textit{C64} instructions in mnemonic form. The subroutines are intended to disassemble and debugging aids but may be used by those with more general output (assemblers) are required. The subroutines occupy one page (256 bytes) and leave 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, 2-1 bytes; (3) 3 character mnemonic, or "??" for invalid eps (which assume a length of \textit{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 provided)</td>
</tr>
<tr>
<td>#12</td>
<td>Immediate</td>
</tr>
<tr>
<td>$12, X</td>
<td>Zero page, indexed by $A</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>(1234)</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) **DISMBA**: Disassembles and displays 20 sequential instructions beginning at the address specified by the page zero variables PCL and PCD. For example, if called with 1D2 in PCL and 382 in PCD, 20 instructions beginning at address $381D will be disassembled. PCL and PCD are updated to contain the address of the last disassembled instruction. Must be called with EOSR in hexadecimal mode ("S" status bit clear). All processor registers are altered (except S - stack pointer). Uses INSTDSP and PCADS.

(b) **INSTDSP**: Disassembles and displays a single instruction whose address is specified by PCL and PCD. Must be called in hexadecimal mode. All processor registers (except S) are altered. Uses PCADS, PRPC, PRBLNK, PRBL2, PRNTAX, PRB1TE, and CHAROUT.

(c) **PRPC**: Outputs a carriage return, 4 hex digits corresponding to PCL and PCD, 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) **PRNTXY**: 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) **PRBL2**: Outputs the number of blanks specified by the contents of X (6 to 256 blanks). Alters A, clears X. Uses CHAROUT.

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

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

Cassette tapes supplied for the ACL-1 (APPLE Cassette Interface) are intended to be loaded from F500 to F9FF.

5. Non-APPLE systems.

Source and object code supplied occupies pages 5 and 9. All code is on page 5. Tables on page 4. These tables may be relocated at will: MODE, MODE2, CHART, CHAR2, MNEM2, and MNEMA. 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 variables (to use DSMBL). These may be relocated on page 6 but PCL must always immediately precede PCL in (page, Y addressing).

```
{ 140 FORMAT }
{ 141 LENGTH } Used to INSINS, DSMBL
{ 142 EMNEM } 1
{ 143 RMNEM } 1
{ 144 PCL } Used by COUNT, INSINS, DSMBL
{ 145 PCL } COUNT and 1, DSMBL only
{ 146 COUNT }
(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 \(Y = \$F14\) and \(A = \$38\). \(X\) is always loaded with \(PCH\).

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

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

(a) To change "#" to '=' for immediate mode change location $955
    (in code enclosed) 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)
    948 : 83    (was 91)
    949 : 12    (was 92)
    94A : 16    (was 5A)
    950 : 85    (was 55)
    951 : 1D    (was 9D)
    952 : 08    (was A9)
    953 : 09    (was A9)

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

  (1) Must skip $ preceding disassembled values by making modification
      (b) above.
  (2) Change the following locations:

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

(d) To add REG and addressing modes change the following locations:

    90: 20    (was 01)
    91: 26    (was 0C)
    91: 02    (was 02)
    91: 16    (was 16)
    91: 13    (was 13)
    91: 81    (was 81)
    91: 01    (was 01)
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Assembler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>DSMBL</td>
<td>Count for 20 instruction disassembly.</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>LDA #$13</td>
<td>Disassemble + display one instruction.</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>STA COUNT</td>
<td>Update PCL, M to next instruction.</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>JSR INSTDSP</td>
<td>Done first 19 instructions?</td>
</tr>
<tr>
<td>1</td>
<td>06</td>
<td>STA PCL</td>
<td>Yes, loop. Else disassemble 20th.</td>
</tr>
<tr>
<td>0</td>
<td>07</td>
<td>STY PCH</td>
<td>Print PCL, PCH.</td>
</tr>
<tr>
<td>0</td>
<td>08</td>
<td>DEC COUNT</td>
<td>Get op code.</td>
</tr>
<tr>
<td>2</td>
<td>0A</td>
<td>JSR PCADJ</td>
<td>Even/odd test.</td>
</tr>
<tr>
<td>1</td>
<td>0B</td>
<td>BCC IEVEN</td>
<td>b, test.</td>
</tr>
<tr>
<td>0</td>
<td>0C</td>
<td>BSR ERR</td>
<td>XXXXXXXX instruction invalid.</td>
</tr>
<tr>
<td>0</td>
<td>0D</td>
<td>CMP #$22</td>
<td>10001001 instruction invalid.</td>
</tr>
<tr>
<td>1</td>
<td>0E</td>
<td>BEQ ERR</td>
<td>Mask 3 bits for address mode and add indexing offset.</td>
</tr>
<tr>
<td>0</td>
<td>0F</td>
<td>AND #$7</td>
<td>LSB into carry for left/right test below.</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>ORA #$80</td>
<td>Index into address mode table.</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>IEVEN</td>
<td>If carry set use LSD for print format index.</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>LDA MODE,X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>BCS RTMODE</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>17</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1A</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1B</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1C</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1D</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1E</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1F</td>
<td>LSR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>20</td>
<td>RTMODE</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>21</td>
<td>AND #$F</td>
<td>Mask for 4-bit index.</td>
</tr>
<tr>
<td>0</td>
<td>22</td>
<td>BNE GETFMT</td>
<td>$0 for invalid opcodes.</td>
</tr>
<tr>
<td>0</td>
<td>23</td>
<td>A6 80</td>
<td>Substitute $80 for all invalid opcodes.</td>
</tr>
<tr>
<td>0</td>
<td>24</td>
<td>ERR</td>
<td>Set print format index to $0.</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>LDY #$80</td>
<td>Index into print format table.</td>
</tr>
<tr>
<td>0</td>
<td>26</td>
<td>LDA #$0</td>
<td>Save for address field formatting.</td>
</tr>
<tr>
<td>0</td>
<td>27</td>
<td>GETFMT</td>
<td>Mask for 2-bit length ($1, $2, $4, $8).</td>
</tr>
<tr>
<td>0</td>
<td>28</td>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>29</td>
<td>LDA MODE2,X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2A</td>
<td>STA FORMAT</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2B</td>
<td>AND #$13</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STA LENGTH
TYA
AND #$8F
TAX
TYA
LDY #$3
CPX #$3A
BEQ MNNDX3

MNNDX1
LSR
BCC MNNDX3

MNNDX2
LSR
ORA #$20
DEY
BNE MNNDX2

MNNDX3
DEY
BNE MNNDX1

Save mnemonic table index.

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

Character count for mnemonic print.

Recover mnemonic index.

Fetch 3 character (packed in 2 bytes) mnemonic.
S7E A9 Ø
S80 AØ 5
S82 6 43
S84 26 42
S86 2A
S87 88
S88 1Ø F8
S8A C9 BF
S8C 2Ø EF FF
S8E CA
S90 DØ EC
S92 2Ø E4 8
S95 A2 6
S97 EØ 3
S99 DØ 12
S9B A4 41
S9D FØ Ø
S9F A5 4Ø
SA1 C9 E8
SA3 B1 44
SA5 BØ 1C
SA7 2A DC FF
SA9 88
SAB DØ F2
SAD 6 40
SAF 9Ý E
SB1 BD 51 9
SB4 2Ø EF FF
SB7 BD 57 9
SBA FØ 3
SBC 2Ø EF FF
SBF CA
SC0 DØ D5
SC2 6Ø
PRMN1 LDA #$Ø
PRMN2 ASL RMNEM
ROL LMNEM
DEY
BNE PRMN2
ADC #$BF
JSR CHAROUT
DEX
89Ø BNE PRMN1
JSR PRBLNK
LDX #$6
CPX #$3
BEQ PRADR3
PRADR2 LDA FORMAT
PRADR1 CPX #$3
BNE PRADR3
LDY LENGTH
BEQ PRADR3
Output 6 blanks.
Count for 6 print format bits.
If x=3 then print address val.
No print if LENGTH=Ø (1 byte instr.)
Handle relative addressing mode
special (print target, not displacement).
Output 1 or 2-byte address (more
significant byte first).
Test next print format bit.
If Ø, don't print corresponding chars.
Output 1 or 2 chars (if char from
CHAR2 is zero, don't output it).
Return when done 6 format bits.
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8C3</td>
<td>2F F2 8</td>
<td>RELADR 8C3 JSR PCADJ3</td>
<td>PCL, PCH + Displacement + 1 to A, Y.</td>
</tr>
<tr>
<td>8C6</td>
<td>A8</td>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td>8C7</td>
<td>B8</td>
<td>INX</td>
<td></td>
</tr>
<tr>
<td>8C8</td>
<td>D0 1</td>
<td>BNE PRNTYX</td>
<td>+1 to X, Y.</td>
</tr>
<tr>
<td>8CA</td>
<td>C8</td>
<td>INY</td>
<td></td>
</tr>
<tr>
<td>8CB</td>
<td>9B</td>
<td>PRNTYX TYA</td>
<td></td>
</tr>
<tr>
<td>8CC</td>
<td>20 DC FF</td>
<td>PRNTAX JSR PRBYTE</td>
<td>Output target address of branch and return.</td>
</tr>
<tr>
<td>8CF</td>
<td>8A</td>
<td>PRNTX TXA</td>
<td></td>
</tr>
<tr>
<td>8D0</td>
<td>4C DC FF</td>
<td>80H JMP PRBYTE</td>
<td></td>
</tr>
<tr>
<td>8D3</td>
<td>A9 8D</td>
<td>PRPC LDA #8D</td>
<td></td>
</tr>
<tr>
<td>8D5</td>
<td>20 EF FF</td>
<td>JSR CHAROUT</td>
<td>Output carriage return.</td>
</tr>
<tr>
<td>8D8</td>
<td>A5 45</td>
<td>LDA PCH</td>
<td></td>
</tr>
<tr>
<td>8DA</td>
<td>A6 44</td>
<td>LDX PCL</td>
<td></td>
</tr>
<tr>
<td>8DC</td>
<td>20 CC F8</td>
<td>JSR PRNTAX</td>
<td>Output PCH &amp; PCL.</td>
</tr>
<tr>
<td>8DF</td>
<td>A9 AD</td>
<td>LDA #AD</td>
<td></td>
</tr>
<tr>
<td>8E1</td>
<td>20 EF FF</td>
<td>JSR CHAROUT</td>
<td>Output &quot;-&quot;.</td>
</tr>
<tr>
<td>8E4</td>
<td>A2 3</td>
<td>PRBLNK LDX #3</td>
<td>Blank count.</td>
</tr>
<tr>
<td>8E6</td>
<td>A9 AØ</td>
<td>PRBL2 LDA #AØ</td>
<td></td>
</tr>
<tr>
<td>8EB</td>
<td>CA</td>
<td>DEX</td>
<td></td>
</tr>
<tr>
<td>8EC</td>
<td>D0 F8</td>
<td>BNE PRBL2</td>
<td>Loop until count = 0.</td>
</tr>
<tr>
<td>8EE</td>
<td>60</td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>8EF</td>
<td>A5 41</td>
<td>PCADJ LDA LENGTH</td>
<td>Ø = 1 byte, 1 = 2 byte, 2 = 3 byte.</td>
</tr>
<tr>
<td>8F1</td>
<td>38</td>
<td>PCADJ2 SEC</td>
<td>Test displ. sign. (for rel. branch).</td>
</tr>
<tr>
<td>8F2</td>
<td>45</td>
<td>PCADJ3 LDY PCH</td>
<td></td>
</tr>
<tr>
<td>8F4</td>
<td>AA</td>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td>8F5</td>
<td>10 1</td>
<td>BPL PCADJ4</td>
<td>Extend neg. by decrementing PCH.</td>
</tr>
<tr>
<td>8F6</td>
<td>88</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>8F8</td>
<td>65 44</td>
<td>PCADJ4 ADC PCL</td>
<td></td>
</tr>
<tr>
<td>8FA</td>
<td>7A 1</td>
<td>BCC RTS1</td>
<td>PCL + LENGTH (or displ.) + 1 to A.</td>
</tr>
<tr>
<td>8FC</td>
<td>CB</td>
<td>INY</td>
<td>Carry into Y (PCH).</td>
</tr>
<tr>
<td>8FD</td>
<td>60</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>
The Woz Wonderbook

DOCUMENT

Apple-II

Cassette Article

This page is not part of the original Wonderbook
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!

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 indicate the recorder transfer speed. A sync bit precedes the data and indicates its start. A termination byte is recorded after all data bytes for each block of information.
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 square wave, to allow the recorder to reach speed and the head circuits to lock on. The READ/RECORD algorithm is such that the header, beginning may contain 'junk'.

First Record Header: Approx 10 seconds to bypass tape leader.

Other Record Headers: .5 to 20 seconds, depending on user needs such as whether the recorder will be stopped prior to the record.

**HEADER BIT (\(\text{SYNC} = 1\))**

\[
\begin{array}{c}
\text{SYNC} \quad \text{SYNC} \\
\text{SYNC} \quad \text{SYNC}
\end{array}
\]

**SYNC**

To ensure start of data, a half cycle of \(\text{SYNC} = 1\) is the sync bit.
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 180 bytes per second.

The checksum byte immediately follows the last data byte and is recorded in the same 8-1 format. It is the inverse of the logical exclusive-or of all data bytes of the record.

Example:

<table>
<thead>
<tr>
<th>Data byte 1</th>
<th>Data byte 2</th>
<th>Data byte 3</th>
<th>Data byte 4</th>
<th>Data byte 5</th>
<th>Data byte 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010110</td>
<td>01101101</td>
<td>01111001</td>
<td>10100010</td>
<td>01101010</td>
<td>10101101</td>
</tr>
</tbody>
</table>

Checksum: 01111001
Notes:

1. An existing input port may be used in place of the 8797.

2. Any decoded address strobe (glitch tyrant) may be used in place of the 7432.

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

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

1. The Y-REG is initialized to a value indicating '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, X-REG is shifted, X-REG unchanged.

Note: Tape & location for 'LASTIN' must be provided.
Feeding a Byte:

1. Initialize i-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 saved.
   (d) Checksum is written.
   (e) Sound BELL
Reading a Record:

1. Initialize (A1L, A1H) and (A2L, A2H)
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) Wait 3 seconds for tape to reach
        speed.
   (c) Look for tape in toggle.
   (d) Scan header half-bit by half-bit
        waiting for sync bit.
   (e) Read data block, advancing pointer
        (A1L, A1H) until greater than
        (A2L, A2H)
   (f) Read pc checksum byte. If mismatch
       then print "ERR"
   (g) Sound BELL.

Note that all registers and page 0
locations LAST IN and CHKSUM are used.
cassette should not contain a negative value
($5 - 11$ bty). 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 28 JMP NON 4C IF FF`

6. Run `WRITE`. Immediately after typing
   the `run` command, start the recorder,
   it must be in the RECORD mode with
   the tape cable connected to the interface.
   If the tape is not properly seated, the tape
   will not play. Allow 10
   seconds for the tape, and 5 to 10 seconds for
code should not contain a negative value
(FF - FF 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 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 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 2A
JMP MON 4C 1F FF

(6) Run WRITE. Immediately after typing
the run command, start the recorder.
It must be in the RECORD mode with
its mic jack connected to the interface.
If it is on, it will record properly. If
not, the recorder will remain silent. Allow 10
seconds for

Reading a Tape

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

(2) Initialize ALL, AH, A0, and A2H as for writing tapes.

(3) Store the following program in memory.

READ JSR READ 20 E0 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 err 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:

A1L
A1H
A2L
A2H
LASTIN
CHKSUM

The only restriction is that A1L 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 CKE printout and BELL prompts, the
user must provide a character output subroutine
COUT. The assembly listing provided uses the
APPLE-1 entry point FFEF for this subroutine,
you may substitute your own. The A3 X- and Y-KE
must not be disturbed by this subroutine. The byte
to be output is passed in the A-KEA.
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, AILH) and (AZL, AZLH), calls the TRAN or WRITE subroutine, then repeats the address pointers and subroutine call for all further records. If the tape is not striped it is permissible to spend a small amount of time calculating between records, since the first part of the header is ignored.
RELIABILITY

I have tested the interface at Apple over millions of bits without failure. I have used the cheapest 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:

IN  \[ \text{Inh} \quad \text{Out} \quad \text{Out} \quad \text{Out} \]

It can be seen that zero crossings of the output are only very approximate due to high-frequency cutoff, slight differentiation of this signal, coupled with hysteresis (Schmitt-trigger action) were included in the interface zero-crossing detector. Due to the nature of the recording format (one full cycle per data bit) there can be no negative DC offset of the signal being read. The effect of a DC offset is to vary the zero-crossing detection point.
To counteract certain types of distortion (including a DC offset) present in some recorders, a data bit is sampled over a full cycle, never over a half-cycle. From A to B on distorted waveform above, my favorite recorder outputs a square wave as a rectangle wave (below) yet works reliably with this interface.

Reading a string of zeroes or a string of ones presents no major problem. A major problem does crop up when the data contains mixes which show up in cheap recorders but not good ones. This has to do with the amplifier read and write amplifiers within the recorder. Virtually all recorders have a satisfactory bandpass. Roll-off at \( f = 2 \pm 3 \text{ KHz} \) typical.