



Apple 2  
Technical Manual

Apple 2  
Programmer's Aid  
# 1

Document 030-0026-01  
1978

Author:  
Apple Computer, Inc.

# PROGRAMMER'S AID #1

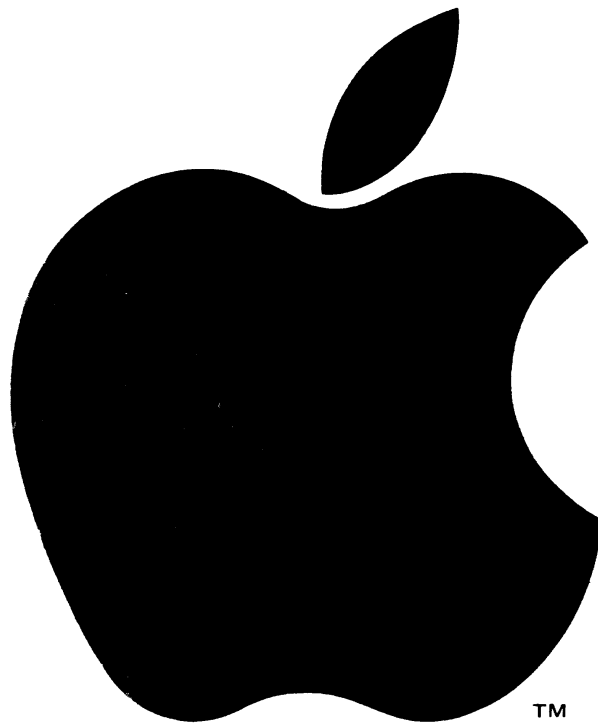
## INSTALLATION AND OPERATING MANUAL



# Apple Utility Programs

"A2MAN 030-0026-01 1 1-COV.PICT" 6385 KB 2001-07-22 dpi: 600h x 600v pix: 2446h x 3487v

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



TM

**Apple Utility Programs**

**DAVID T. CRAIG**

Published by  
APPLE COMPUTER INC.  
10260 Bandley Drive  
Cupertino, California 95014  
(408) 996-1010

All rights reserved. No part of this publication  
may be reproduced without the prior written  
permission of APPLE COMPUTER INC. Please  
call (408) 996-1010 for more information.

©1978 by APPLE COMPUTER INC.

Reorder APPLE Product #A2L0011  
(030-0026)

# TABLE OF CONTENTS

## INTRODUCTION

- XI Features of Programmer's Aid #1
- XII How to install the Programmer's Aid ROM

## CHAPTER 1

## RENUMBER

- 2 Renumbering an entire BASIC program
- 2 Renumbering a portion of a BASIC program
- 4 Comments

I

# CHAPTER 2

## APPEND

- 6 Appending one BASIC program to another
- 6 Comments

||

# CHAPTER 3

## TAPE VERIFY (BASIC)

- 8 Verifying a BASIC program SAVED on tape
- 8 Comments



# CHAPTER 4

## TAPE VERIFY (Machine Code or Data)

- 1Ø Verifying a portion of memory saved on tape
- 1Ø Comments



# CHAPTER 5

## RELOCATE

- 12 Part A: Theory of operation
  - 12 Relocating machine-language code
  - 13 Program model
  - 14 Blocks and Segments
  - 15 Code and Data Segments
  - 16 How to use the Code-Relocation feature
  
- 18 Part B: Examples of Code relocation
  - 18 Example 1. Straightforward relocation
  - 19 Example 2. Index into Block
  - 20 Example 3. Immediate address reference
  - 20 Example 4. Unusable Block ranges
  - 21 Example 5. Changing the page zero variable allocation
  - 22 Example 6. Split Blocks with cross-referencing
  - 23 Example 7. Code deletion
  - 24 Example 8. Relocating the APPLE II Monitor (\$F800-\$FFFF)  
to run in RAM (\$800-\$FFF)
  
- 25 Part C: Further details
  - 25 Technical Information
  - 26 Algorithm used by the Code-Relocation feature
  - 27 Comments

v

# CHAPTER 6

## RAM TEST

- 30 Testing APPLE's memory
- 31 Address ranges for standard memory configurations
- 32 Error messages
  - Type I - Simple error
  - Type II - Dynamic error
- 33 Testing for intermittent failure
- 34 Comments

# CHAPTER 7

## MUSIC

- 36 Generating musical tones
- 37 Comments

# CHAPTER 8

## HIGH-RESOLUTION GRAPHICS

- 40 Part A: Setting up parameters, subroutines, and colors
  - 40 Positioning the High-Resolution parameters
  - 41 Defining subroutine names
  - 41 Defining color names
  - 42 Speeding up your program
- 43 Part B: Preparing the screen for graphics
  - 43 The INITIALization subroutine
  - 43 Changing the graphics screen
  - 44 CLEARing the screen to BLACK
  - 44 Coloring the BackGrouND
- 45 Part C: PLOTting points and LINEs
- 46 Part D: Creating, saving and loading shapes
  - 46 Introduction
  - 47 Creating a Shape Table
  - 53 Saving a Shape Table
  - 54 Loading a Shape Table
  - 55 First use of Shape Table
- 56 Part E: Drawing shapes from a prepared Shape Table
  - 55 Assigning parameter values: SHAPE, SCALE and ROTation
  - 57 DRAWing shapes
  - 58 Linking shapes: DRAW1
  - 59 Collisions
- 60 Part F: Technical information
  - 60 Locations of the High-Resolution parameters
  - 61 Variables used within the High-Resolution subroutines
  - 62 Shape Table information
  - 63 Integer BASIC memory map
- 64 Part G: Comments

# APPENDIX II

## SOURCE ASSEMBLY LISTINGS

65	High-Resolution Graphics	\$D000-\$D3FF
76	Renumber	\$D400-\$D4BB
79	Append	\$D4BC-\$D4D4
80	Relocate	\$D4DC-\$D52D
82	Tape Verify (BASIC)	\$D535-\$D553
83	Tape Verify (6502 Code & Data)	\$D554-\$D5AA
84	PAM Test	\$D5BC-\$D691
87	Music	\$D717-\$D7F8

# APPENDIX I

## SUMMARY OF PROGRAMMER'S AID COMMANDS

92	Renumber
92	Append
92	Tape Verify (BASIC)
93	Tape Verify (Machine Code and Data)
93	Relocate (Machine Code and Data)
94	RAM Test
94	Music
95	High-Resolution Graphics
96	Quick Reference to High-Resolution Graphics Information

# INTRODUCTION

## FEATURES OF PROGRAMMER'S AID #1

Programmer's Aid #1 combines several APPLE II programs that Integer BASIC programmers need quite frequently. To avoid having to load them from a cassette tape or diskette each time they are used, these programs have been combined in a special read-only memory (ROM) integrated circuit (IC). When this circuit is plugged into one of the empty sockets left on the APPLE's printed-circuit board for this purpose, these programs become a built-in part of the computer the same way Integer BASIC and the Monitor routines are built in. Programmer's Aid #1 allows you to do the following, on your APPLE II:

- Chapter 1. Renumber an entire Integer BASIC program, or a portion of the program.
- Chapter 2. Load an Integer BASIC program from tape without erasing the Integer BASIC program that was already in memory, in order to combine the two programs.
- Chapter 3. Verify that an Integer BASIC program has been saved correctly on tape, before the program is deleted from APPLE's memory.
- Chapter 4. Verify that a machine-language program or data area has been saved correctly on tape from the Monitor.
- Chapter 5. Relocate 6502 machine-language programs.
- Chapter 6. Test the memory of the APPLE.
- Chapter 7. Generate musical notes of variable duration over four chromatic octaves, in five (slightly) different timbres, from Integer BASIC.
- Chapter 8. Do convenient High-Resolution graphics from Integer BASIC.

Note: if your APPLE has the firmware APPLESOFT card installed, its switch must be down (in the Integer BASIC position) for Programmer's Aid #1 to operate.

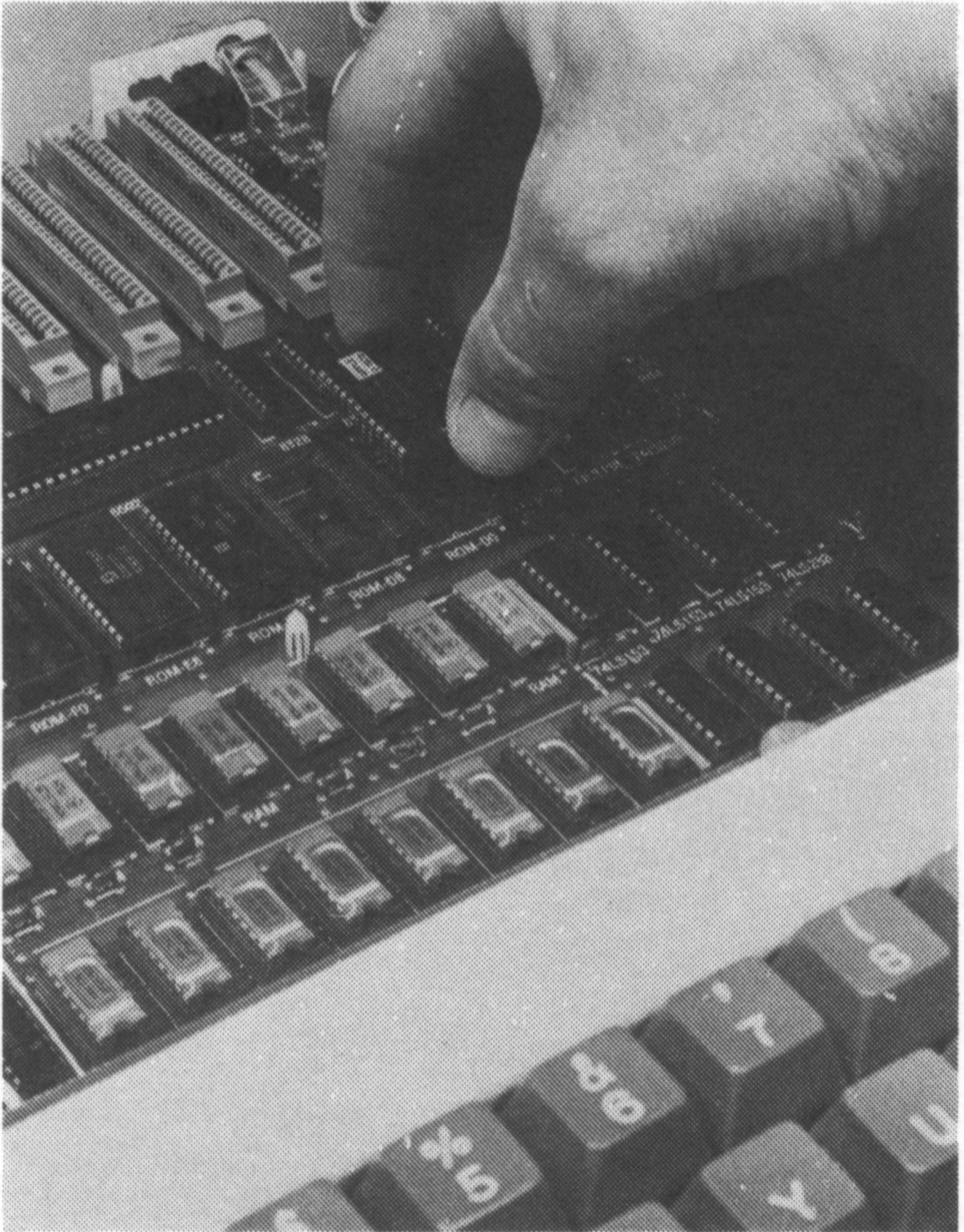
## HOW TO INSTALL THE PROGRAMMER'S AID ROM

The Programmer's Aid ROM is an IC that has to be plugged into a socket on the inside of the APPLE II computer.



1. Turn off the power switch on the back of the APPLE II. This is important to prevent damage to the computer.
2. Remove the cover from the APPLE II. This is done by pulling up on the cover at the rear edge until the two corner fasteners pop apart. Do not continue to lift the rear edge, but slide cover backward until it comes free.
3. Inside the APPLE, toward the right center of the main printed-circuit board, locate the large empty socket in Row F, marked "ROM-D0".
4. Make sure that the Programmer's Aid ROM IC is oriented correctly. The small semicircular notch should be toward the keyboard. The Programmer's Aid ROM IC must match the orientation of the other ROM ICs that are already installed in that row.
5. Align all the pins on the Programmer's Aid ROM IC with the holes in socket D0, and gently press the IC into place. If a pin bends, remove the IC from its socket using an "IC puller" (or, less optimally, by prying up gently with a screwdriver). Do not attempt to pull the socket off the board. Straighten any bent pins with a needlenose pliers, and press the IC into its socket again, even more carefully.
6. Replace the cover of the APPLE, remembering to start by sliding the front edge of the cover into position. Press down on the two rear corners until they pop into place.
7. Programmer's Aid #1 is installed; the APPLE II may now be turned on.





"A2MAN 030-0026-01 2-12xiiP.PICT" 6016 KB 2001-07-22 dpi: 1200h x 1200v pix: 2195h x 2809v

# CHAPTER **1**

# RENUMBER

- 2 Renumbering an entire BASIC program
- 2 Renumbering a portion of a BASIC program
- 4 Comments

## RENUMBERING AN ENTIRE BASIC PROGRAM

After loading your program into the APPLE, type the

CLR

command. This clears the BASIC variable table, so that the Renumber feature's parameters will be the first variables in the table. The Renumber feature looks for its parameters by location in the variable table. For the parameters to appear in the table in their correct locations, they must be specified in the correct order and they must have names of the correct length.

Now, choose the number you wish assigned to the first line in your renumbered program. Suppose you want your renumbered program to start at line number 1000. Type

START = 1000

Any valid variable name will do, but it must have the correct number of characters. Next choose the amount by which you want succeeding line numbers to increase. For example, to renumber in increments of 10, type

STEP = 10

Finally, type the this command:

CALL -10531

As each line of the program is renumbered, its old line number is displayed with an "arrow" pointing to the new line number. A possible example might appear like this on the APPLE's screen:

```
7->1000
213->1010
527->1020
698->1030
13000->1040
13233->1050
```

## RENUMBERING PORTIONS OF A PROGRAM

You do not have to renumber your entire program. You can renumber just the lines numbered from, say, 300 to 500 by assigning values to four variables. Again, you must first type the command

CLR

to clear the BASIC variable table.

The first two variables for partial renumbering are the same as those for renumbering the whole program. They specify that the program portion, after renumbering, will begin with line number 200, say, and that each line's number thereafter will be 20 greater than the previous line's:

```
START = 200
STEP = 20
```

The next two variables specify the program portion's range of line numbers before renumbering:

```
FROM = 300
TO = 500
```

The final command is also different. For renumbering a portion of a program, use the command:

```
CALL -10521
```

If the program was previously numbered

```
100
120
300
310
402
500
2000
2022
```

then after the renumbering specified above, the APPLE will show this list of changes:

```
300->200
310->220
402->240
500->260
```

and the new program line numbers will be

```
100
120
200
220
240
260
2000
2022
```

You cannot renumber in such a way that the renumbered lines would replace, be inserted between or be intermixed with un-renumbered lines. Thus, you cannot change the order of the program lines. If you try, the message

```
*** RANGE ERR
```

is displayed after the list of proposed line changes, and the line numbers themselves are left unchanged. If you type the commands in the wrong order, nothing happens, usually.

## COMMENTS:

1. If you do not CLR before renumbering, unexpected line numbers may result. It may or may not be possible to renumber the program again and save your work.
2. If you omit the START or STEP values, the computer will choose them unpredictably. This may result in loss of the program.
3. If an arithmetic expression or variable is used in a GOTO or GOSUB, that GOTO or GOSUB will generally not be renumbered correctly. For example, GOTO TEST or GOSUB 10+20 will not be renumbered correctly.
4. Nonsense values for STEP, such as 0 or a negative number, can render your program unusable. A negative START value can renumber your program with line numbers above 32767, for what it's worth. Such line numbers are difficult to deal with. For example, an attempt to LIST one of them will result in a >32767 error. Line numbers greater than 32767 can be corrected by renumbering the entire program to lower line numbers.
5. The display of line number changes can appear correct even though the line numbers themselves have not been changed correctly. After the \*\*\* RANGE ERR message, for instance, the line numbers are left with their original numbering. LIST your program and check it before using it.
6. The Renumber feature applies only to Integer BASIC programs.
7. Occasionally, what seems to be a "reasonable" renumbering does not work. Try the renumbering again, with a different START and STEP value.

# CHAPTER 2

## APPEND

- 6 Appending one BASIC program to another
- 6 Comments

## APPENDING ONE BASIC PROGRAM TO ANOTHER

If you have one program or program portion stored in your APPLE's memory, and another saved on tape, it is possible to combine them into one program. This feature is especially useful when a subroutine has been developed for one program, and you wish to use it in another program without retyping the subroutine.

For the Append feature to function correctly, all the line numbers of the program in memory must be greater than all the line numbers of the program to be appended from tape. In this discussion, we will call the program saved on tape "Program1," and the program in APPLE's memory "Program2."

If Program2 is not in APPLE's memory already, use the usual command

```
LOAD
```

to put Program2 (with high line numbers) into the APPLE. Using the Renumber feature, if necessary, make sure that all the line numbers in Program2 are greater than the highest line number in Program1.

Now place the tape for Program1 in the tape recorder. Use the usual loading procedure, except that instead of the LOAD command use this command:

```
CALL -11076
```

This will give the normal beeps, and when the second beep has sounded, the two programs will both be in memory. If this step causes the message

```
*** MEM FULL ERR
```

to appear, neither Program2 nor Program1 will be accessible. In this case, use the command

```
CALL -11059
```

to recover Program2, the program which was already in APPLE's memory.

### COMMENTS:

1. The Append feature operates only with APPLE II Integer BASIC programs.
2. If the line numbers of the two programs are not as described, expect unpredictable results.

# CHAPTER **3**

## **TAPE VERIFY (BASIC)**

- 8 Verifying a BASIC program SAVED on tape
- 8 Comments



## VERIFYING A BASIC PROGRAM SAVED ON TAPE

Normally, it is impossible (unless you have two APPLES) to know whether or not you have successfully saved your current program on tape, in time to do something about a defective recording. The reason is this: when you SAVE a program on tape, the only way to discover whether it has been recorded correctly is to LOAD it back in to the APPLE. But, when you LOAD a program, the first thing the APPLE does is erase whatever current program is stored. So, if the tape is bad, you only find out after your current program has been lost.

The Tape Verify feature solves this problem. Save your current program in the usual way:

```
SAVE
```

Rewind the tape, and (without modifying your current program in any way) type the command

```
CALL -10955
```

Do not press the RETURN key until after you start the tape playing. If the tape reads in normally (with the usual two beeps), then it is correct. If there is any error on the tape, you will get a beep and the ERR message. If this happens, you will probably want to try re-recording the tape, although you don't know for sure whether the Tape Verify error means that the tape wasn't recorded right or if it just didn't play back properly. In any case, if it does verify, you know that it is good.

### COMMENTS:

1. This works only with Integer BASIC programs.
2. Any change in the program, however slight, between the time the program is SAVED on tape and the time the tape is verified, will cause the verification to fail.

# CHAPTER 4

## TAPE VERIFY

### (Machine Code or Data)

- 1Ø Verifying a portion of memory saved on tape
- 1Ø Comments

## VERIFYING A PORTION OF MEMORY SAVED ON TAPE

Users of machine-language routines will find that this version of the Tape Verify feature meets their needs. Save the desired portion of memory, from address1 to address2, in the usual way:

```
address1 . address2 W return
```

Note: the example instructions in this chapter often include spaces for easier reading; do not type these spaces.

Rewind the tape, and type (after the asterisk prompt)

```
D52EG return
```

This initializes the Tape Verify feature by preparing locations \$3F8 through \$3FA for the ctrl Y vector. Now type (do not type the spaces)

```
address1 . address2 ctrl Y return
```

and re-play the tape. The first error encountered stops the program and is reported with a beep and the word ERR. If it is not a checksum error, then the Tape Verify feature will print out the location where the tape and memory disagreed and the data that it expected on the tape.

Note: type "ctrl Y" by typing Y while holding down the CTRL key; ctrl Y is not displayed on the TV screen. Type "return" by pressing the RETURN key.

### COMMENTS:

Any change in the specified memory area, however slight, between the time the program is saved on tape and the time the tape is verified, will cause the verification to fail.

# CHAPTER 5

## RELOCATE

- 12 Part A: Theory of operation
  - 12 Relocating machine-language code
  - 13 Program model
  - 14 Blocks and Segments
  - 15 Code and Data Segments
  - 16 How to use the Code-Relocation feature
  
- 18 Part B: Examples
  - 18 Example 1. Straightforward relocation
  - 19 Example 2. Index into Block
  - 20 Example 3. Immediate address reference
  - 20 Example 4. Unusable Block ranges
  - 21 Example 5. Changing the page zero variable allocation
  - 22 Example 6. Split Blocks with cross-referencing
  - 23 Example 7. Code deletion
  - 24 Example 8. Relocating the APPLE II Monitor (\$F800-\$FFFF)  
to run in RAM (\$800-\$FFF)
  
- 25 Part C: Further details
  - 25 Technical information
  - 26 Algorithm used by the Code-Relocation feature
  - 27 Comments

# PART A: THEORY OF OPERATION

## RELOCATING MACHINE-LANGUAGE CODE

Quite frequently, programmers encounter situations that call for relocating machine-language (not BASIC) programs on the 6502-based APPLE II computer. Relocation implies creating a new version of the program, a version that runs properly in an area of memory different from that in which the original program ran.

If they rely on the relative branch instruction, certain small 6502 programs can simply be moved without alteration, using the existing Monitor Move commands. Other programs will require only minor hand-modification after Monitor Moving. These modifications are simplified on the APPLE II by the built-in disassembler, which pinpoints absolute memory-reference instructions such as JMP's and JSR's.

However, sometimes it is necessary to relocate lengthy programs containing multiple data segments interspersed with code. Using this Machine-Code Relocation feature can save you hours of work on such a move, with improved reliability and accuracy.

The following situations call for program relocation:

1. Two different programs, which were originally written to run in identical memory 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 a different set of addresses in RAM.
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. Because of memory-mapping differences, a program that ran on an APPLE I (or other 6502-based computer) falls into unusable address space on an APPLE II.
6. Because different operating systems assign variables differently, either 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, which exists as several chunks strewn about memory, 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" memory space.
9. A program insertion or deletion requires a portion of the program to move a few bytes up or down.
10. On a whim, the user wishes to move a program.

## PROGRAM MODEL

Here is one simple way to visualize program relocation: starting with a program which resides and runs in a "Source Block" of memory, relocation creates a modified version of that program which resides and runs properly in a "Destination Block" of memory.

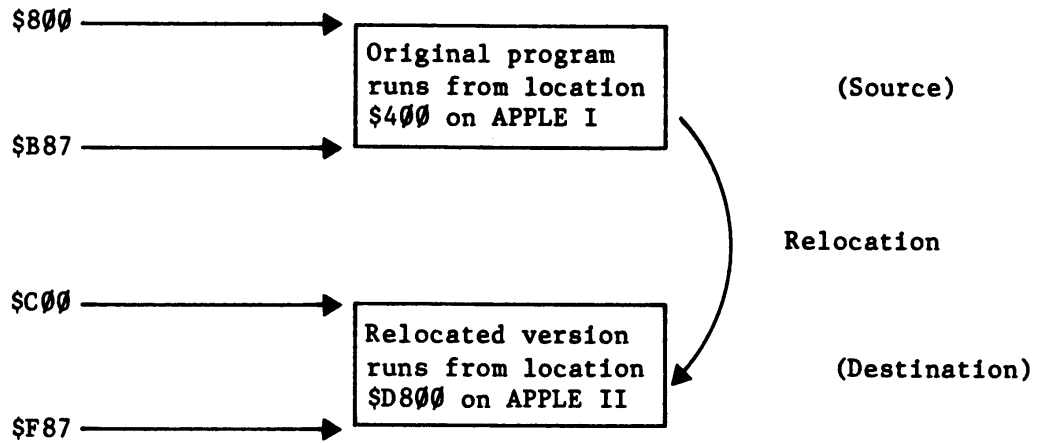
However, this model does not sufficiently describe situations where the "Source Block" and the "Destination Block" are the same locations in memory. For example, a program written to begin at location \$400 on an APPLE I (the \$ indicates a hexadecimal number) falls in the APPLE II screen-memory range. It must be loaded to some other area of memory in the APPLE II. But the program will not run properly in its new memory locations, because various absolute memory references, etc., are now wrong. This program can then be "relocated" right back into the same new memory locations, a process which modifies it to run properly in its new location.

A more versatile program model is as follows. A program or section of a program written to run in a memory range termed the "Source Block" actually resides currently in a range termed the "Source Segments". Thus a program written to run from location \$400 may currently reside beginning at location \$800. After relocation, the new version of the program must be written to run correctly in a range termed the "Destination Block" although it will actually reside currently in a range termed the "Destination Segments". Thus a program may be relocated such that it will run correctly from location \$D800 (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

**Segments:**  
Locations in APPLE II  
where Programs Reside  
During Relocation

**Blocks:**  
Locations where  
Programs Run



SOURCE BLOCK: \$400-\$787

DESTINATION BLOCK: \$D800-\$DB87

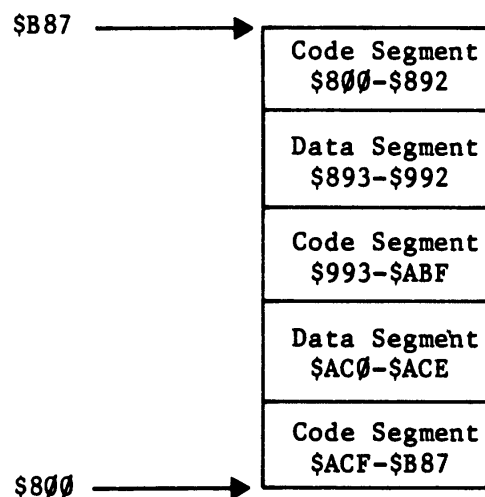
SOURCE SEGMENTS: \$800-\$B87

DESTINATION SEGMENTS: \$C00-\$F87

## DATA SEGMENTS

The problem with relocating a large program all at once is that blocks of data (tables, text, etc.) may be interspersed throughout the code. During relocation, this data may be treated as if it were code, causing the data to be changed or causing code to be altered incorrectly because of boundary uncertainties introduced when the data takes on the multi-byte attribute of code. This problem is circumvented by dividing the program into code segments and data segments, and then treating the two types of segment differently.

### CODE AND DATA SEGMENTS EXAMPLE



The Source Code Segments are relocated (using the 6502 Code-Relocation feature), while the Source Data Segments are moved (using the Monitor Move command).



## HOW TO USE THE CODE-RELOCATION FEATURE

1. To initialize the 6502 Code-Relocation feature, press the RESET key to invoke the Monitor, and then type

```
D4D5G return
```

The Monitor user function ctrl Y will now call the Code-Relocation feature as a subroutine at location \$3F8.

Note: To type "ctrl Y", type Y while holding down the CTRL key. To type "return", press the RETURN key. In the remainder of this discussion, all instructions are typed to the right of the Monitor prompt character ( \* ). The example instructions in this chapter often include spaces for easier reading; do not type these spaces.

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 Destination and Source Block parameters. Remember that a Block refers to locations from which the program will run, not the locations at which the Source and Destination Segments actually reside 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 . SOURCE BLOCK END ctrl Y * return
```

Notes: the syntax of this command closely resembles that of the Monitor Move command. Type "ctrl Y" by pressing the Y key while holding down the CTRL key. Then type an asterisk ( \* ); and finally, type "return" by pressing the RETURN key. Do not type any spaces within the command.

4. Move all Data Segments and relocate all Code Segments in sequential (increasing address) order. It is wise to prepare a list of segments, specifying beginning and ending addresses, and whether each segment is code or data.

If First Segment is Code:

DEST SEGMENT BEG < SOURCE SEGMENT BEG . SOURCE SEGMENT END ctrl Y return

If First Segment is Data:

DEST SEGMENT BEG < SOURCE SEGMENT BEG . SOURCE SEGMENT END M return

After the first segment has been either relocated (if Code) or Moved (if data), subsequent segments can be relocated or Moved using a shortened form of the command.

Subsequent Code Segments:

. SOURCE SEGMENT END ctrl Y return (Relocation)

Subsequent Data Segments:

. SOURCE SEGMENT END M return (Move)

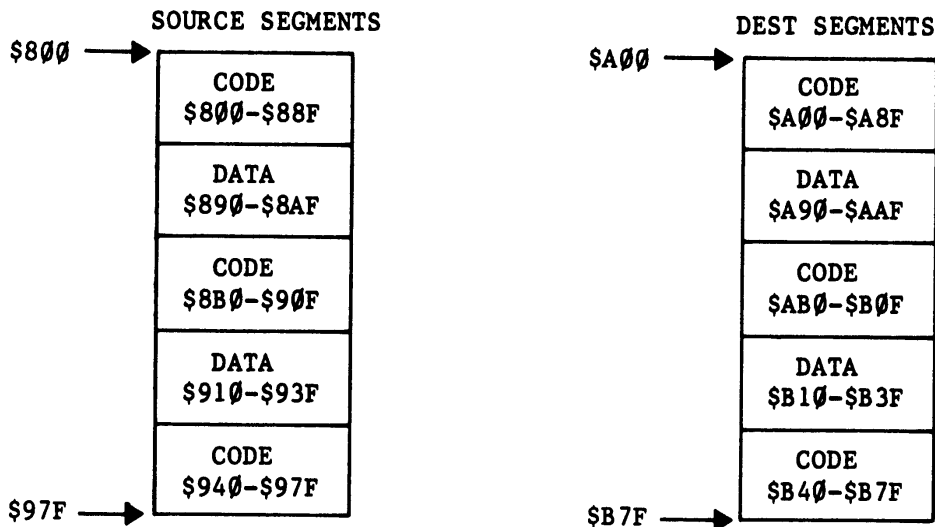
Note: the shortened form of the command can only be used if each "subsequent" segment is contiguous to the segment previously relocated or Moved. If a "subsequent" segment is in a part of memory that does not begin exactly where the previous segment ended, it must be Moved or relocated using the full "First Segment" format.

If the relocation is performed "in place" (SOURCE and DEST SEGMENTS reside in identical locations) then the SOURCE SEGMENT BEG parameter may be omitted from the First Segment relocate or Move command.

# PART B: CODE-RELOCATION EXAMPLES

## EXAMPLE 1. Straightforward Relocation

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



SOURCE BLOCK: \$800-\$97F  
SOURCE SEGMENTS: \$800-\$97F

DEST BLOCK: \$A00-\$B7F  
DEST SEGMENTS: \$A00-\$B7F

(a) Initialize Code-Relocation feature:

```
reset D4D5G return
```

(b) Specify Destination and Source Block parameters (locations from which the program will run):

```
A00 < 800 . 97F ctrl Y * return
```

(c) Relocate first segment (code):

```
A00 < 800 . 88F ctrl Y return
```

(d) Move subsequent Data Segments and relocate subsequent Code Segments, in ascending address sequence:

```
. 8AF M return          (data)
. 90F ctrl Y return     (code)
. 93F M return          (data)
. 97F ctrl Y return     (code)
```

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

```
A90 < 890 . 8AF M return      (data)
AB0 < 8B0 . 90F ctrl Y return  (code)
B10 < 910 . 93F M return      (data)
B40 < 940 . 97F ctrl Y return  (code)
```

## EXAMPLE 2. Index into Block

Suppose that the program of Example 1 uses an indexed reference into the Data Segment at \$890 as follows:

```
LDA 7B0,X
```

where the X-REG is presumed to contain a number in the range \$E0 to \$FF. Because address \$7B0 is outside the Source Block, it will not be relocated. This may be handled in one of two ways.

(a) You may fix the exception by hand; or

(b) You may begin the Block specifications one page lower than the addresses at which the original and relocated programs begin to use all such "early references." One lower page is enough, since FF (the number of bytes in one page) is the largest offset number that the X-REG can contain. In EXAMPLE 1, change step (b) to:

```
900 < 700 . 97F ctrl Y * return
```

Note: with this Block specification, all program references to the "prior page" (in this case the \$700 page) will be relocated.

### EXAMPLE 3. Immediate Address References

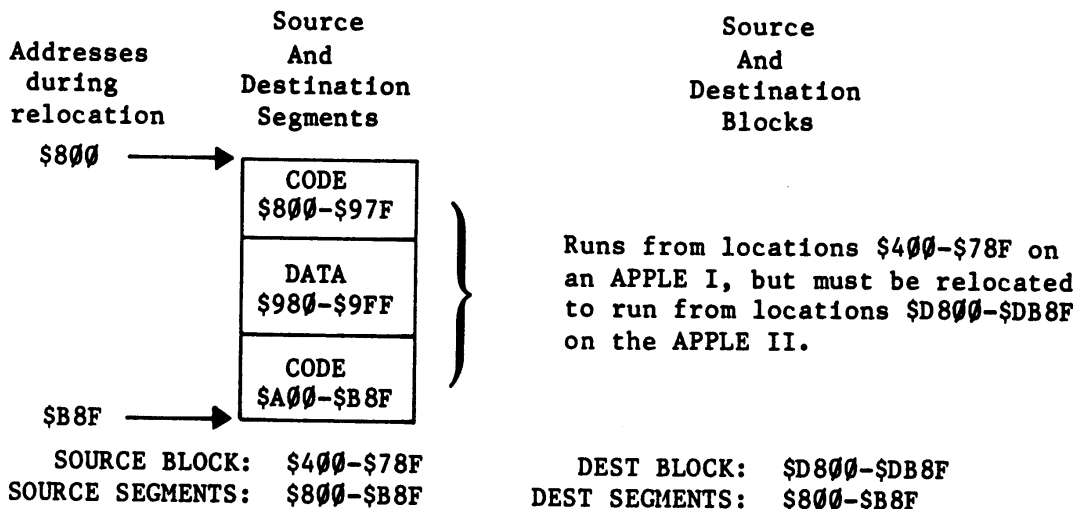
Suppose that the program of EXAMPLE 1 has an immediate reference which is an address. For example,

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

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

### EXAMPLE 4. Unusable Block Ranges

Suppose a program was written to run from locations \$400-\$78F on an APPLE I. A version which will run in ROM locations \$D800-\$DB8F must be generated. The Source (and Destination) Segments will reside in locations \$800-\$B8F on the APPLE II during relocation.



(a) Initialize the Code-Relocation feature:

```
reset D4D5G return
```

(b) Load original program into locations \$800-\$B8F (despite the fact that it doesn't run there):

```
800 . B8F R return
```

(c) Specify Destination and Source Block parameters (locations from which the original and relocated versions will run):

```
D800 < 400 . 78F ctrl Y return
```

(d) Move Data Segments and relocate Code Segments, in ascending address sequence:

```
800 < 800 . 97F ctrl Y return (first segment, code)
. 9FF M return (data)
. B8F ctrl Y return (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 ctrl Y return
```

## EXAMPLE 5. Changing the Page Zero Variable Allocation

Suppose the program of EXAMPLE 1 need not be relocated, but the page zero variable allocation is from \$20 to \$3F. Because these locations are reserved for the APPLE II system monitor, the allocation must be changed to locations \$80-\$9F. The Source and Destination Blocks are thus not the program but rather the variable area.

SOURCE BLOCK: \$20-\$3F	DEST BLOCK: \$80-\$9F
SOURCE SEGMENTS: \$800-\$97F	DEST SEGMENTS: \$800-\$97F

(a) Initialize the Code-Relocation feature:

```
reset D4D5G return
```

(b) Specify Destination and Source Blocks:

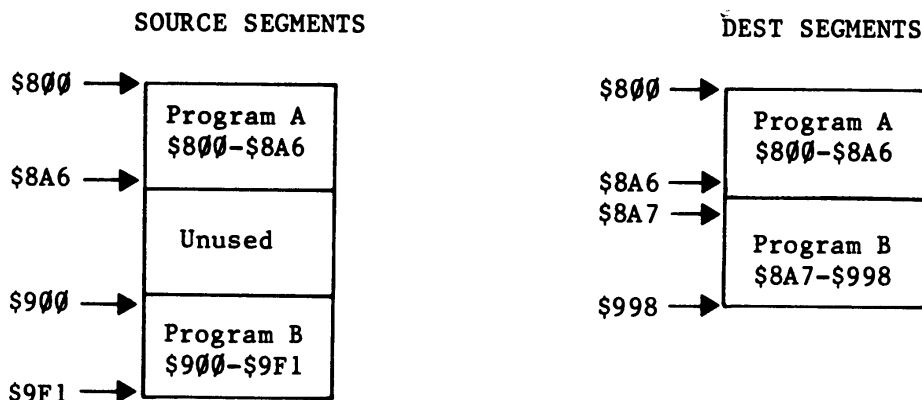
```
80 < 20 . 3F ctrl Y * return
```

(c) Relocate Code Segments and Move Data Segments, in place:

```
800 < . 88F ctrl Y return (first segment, code)
. 8AF M return (data)
. 90F ctrl Y return (code)
. 93F M return (data)
. 97F ctrl Y return (code)
```

## EXAMPLE 6. 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 references to memory locations within the other. It is assumed that the programs contain no Data Segments.



SOURCE BLOCK: \$900-\$9F1  
 SOURCE SEGMENTS: \$800-\$8A6 (A)  
                           \$900-\$9F1 (B)

DEST BLOCK: \$8A7-\$998  
 DEST SEGMENTS: \$800-\$8A6 (A)  
                           \$8A7-\$998 (B)

(a) Initialize the Code-Relocation feature:

```
D4D5G return
```

(b) Specify Destination and Source Blocks (Program B only):

```
8A7 < 900 . 9F1 ctrl Y * return
```

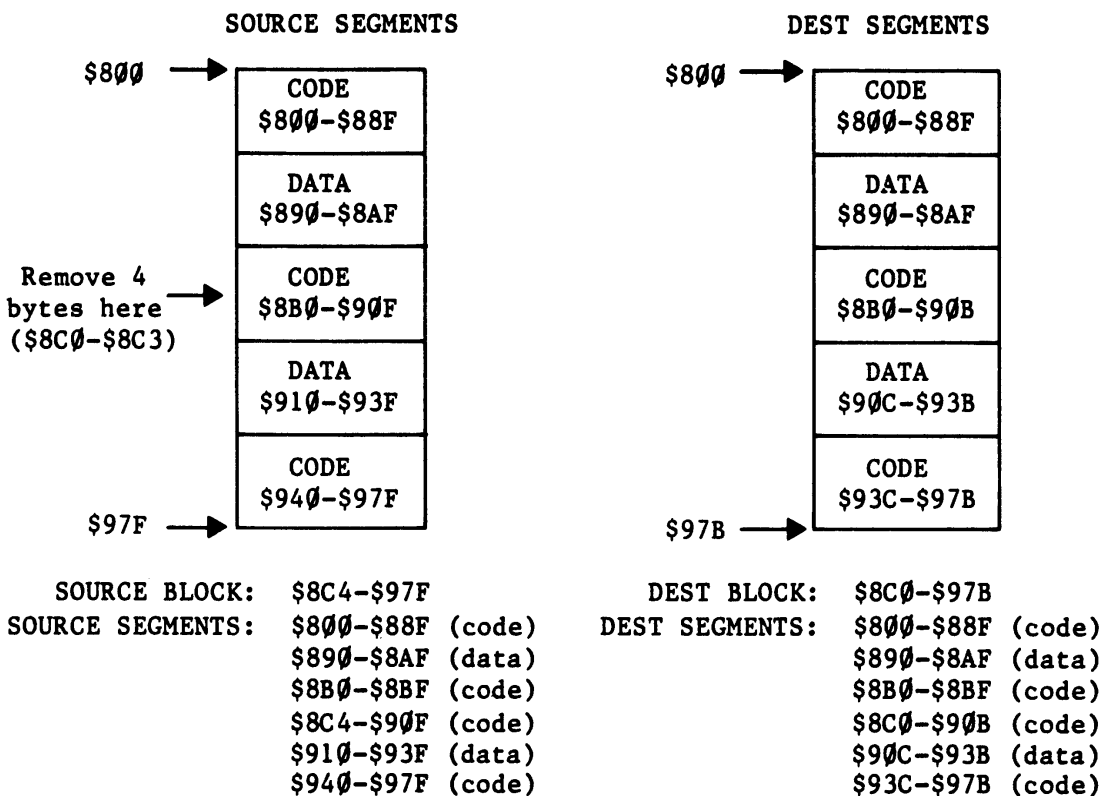
(c) Relocate each of the two programs individually. Program A must be relocated even though it does not move.

```
800 < . 8A6 ctrl Y return           (program A, "in place")
8A7 < 900 . 9F1 ctrl Y return       (program B, not "in place")
```

Note that any Data Segments within the two programs would necessitate additional relocation and Move commands.

## EXAMPLE 7. Code Deletion

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



(a) Initialize Code-Relocation feature:

```
reset D4D5G return
```

(b) Specify Destination and Source Blocks:

```
8C0 < 8C4 . 97F ctrl Y * return
```

(c) Relocate Code Segments and Move Data Segments, in ascending address sequence:

<pre>800 &lt;. 88F ctrl Y return . 8AF M return . 8BF ctrl Y return 8C0 &lt; 8C4 . 90F ctrl Y return . 93F M return . 97F ctrl Y return</pre>	<pre>(first segment, code, "in place") (data) (code) (first segment, code, not "in place") (data) (code)</pre>
---	--

(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.



## EXAMPLE 8. Relocating the APPLE II Monitor (\$F800–\$FFFF) to Run in RAM (\$800–\$FFF)

SOURCE BLOCK: \$F700–\$FFFF                      DEST BLOCK: \$700–\$FFF  
 (see EXAMPLE 2)

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

IMMEDIATE ADDRESS REFERENCES (see EXAMPLE 3):    \$FFBF  
    \$FEA8  
 (more if not relocating  
 to page boundary)

(a) Initialize the Code-Relocation feature:

```
reset D4D5G return
```

(b) Specify Destination and Source Block parameters:

```
700 < F700 . FFFF ctrl Y * return
```

(c) Relocate Code Segments and move Data Segments, in ascending address sequence:

```
800 < F800 . F961 ctrl Y return                      (first segment, code)
. FA42 M return                                      (data)
. FB18 ctrl Y return                                (code)
. FB1D M return                                      (data)
. FFCB ctrl Y return                                (code)
. FFFF M return                                      (data)
```

(d) Change immediate address references:

```
FBF : E return                                      (was $FE)
EA8 : E return                                      (was $FE)
```

# PART C: PLOTTING POINTS AND LINES

## TECHNICAL INFORMATION

The following details illustrate special technical features of the APPLE II which are used by the Code-Relocation feature.

1. The APPLE II Monitor command

Addr4 < Addr1 . Addr2 ctrl Y return (Addr1, Addr2, and Addr4 are addresses)

vectors to location \$3F8 with the value Addr1 in locations \$3C (low) and \$3D (high), Addr2 in locations \$3E (low) and \$3F (high), and Addr4 in locations \$42 (low) and \$43 (high). Location \$34 (YSAV) holds an index to the next character of the command buffer (after the ctrl Y). The command buffer (IN) begins at \$200.

2. If ctrl Y is followed by \* , then the Block parameters are simply preserved as follows:

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

3. If ctrl Y is not followed by \* , then a segment relocation is initiated at RELOC2 (\$3BB). Throughout, Addr1 (\$3C, \$3D) is the Source Segment pointer and Addr4 (\$42, \$43) is the Destination Segment pointer.

4. INSDS2 is an APPLE II Monitor subroutine which determines the length of a 6502 instruction, given the opcode in the A-REG, and stores that opcode's instruction length in the variable LENGTH (location \$2F) .

<u>Instruction Type in A-REG</u>	<u>LENGTH (in \$2F)</u>
Invalid	0
1 byte	0
2 byte	1
3 byte	2

5. The code from XLATE to SW16RT (\$3D9-\$3E6) uses the APPLE II 16-bit interpretive machine, SWEET16. The target address of the 6502 instruction being relocated (locations \$C low and \$D high) occupies the SWEET16 register named ADR. If ADR is between FRMBEG and FRMEND (inclusive) then it is replaced by

ADR - FRMBEG + TOBEG

6. NXTA4 is an APPLE II Monitor subroutine which increments Addr1 (Source Segment index) and Addr4 (Destination Segment index). If Addr1 exceeds Addr2 (Source Segment end), then the carry is set; otherwise, it is cleared.

## ALGORITHM USED BY THE CODE-RELOCATION FEATURE

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 bytes).
4. If two-byte instruction with non-zero-page addressing mode (immediate or relative) then go to step 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 BEG + DEST BLOCK BEG
7. Move "length" bytes from INST area to Destination Segment (using DEST PTR). Update SOURCE and DEST PTR's by length.
8. If SOURCE PTR is less than or equal to SOURCE SEGMENT END then goto step 2., else done.

## COMMENTS:

Each Move or relocation is carried out sequentially, one byte at a time, beginning with the byte at the smallest source address. As each source byte is Moved or relocated, it overwrites any information that was in the destination location. This is usually acceptable in these kinds of Moves and relocations:

1. Source Segments and Destination Segments do not share any common locations (no source location is overwritten).
2. Source Segments are in locations identical to the locations of the Destination Segments (each source byte overwrites itself).
3. Source Segments are in locations whose addresses are larger than the addresses of the Destination Segments' locations (any overwritten source bytes have already been Moved or relocated). This is a move toward smaller addresses.

If, however, the Source Segments and the Destination Segments share some common locations, and the Source Segments occupy locations whose addresses are smaller than the addresses of the Destination Segments' locations, then the source bytes occupying the common locations will be overwritten before they are Moved or relocated. If you attempt such a relocation, you will lose your program and data in the memory area common to both Source Segments and Destination Segments. To accomplish a small Move or relocation toward larger addresses, you must Move or relocate to an area of memory well away from the Source Segments (no address in common); then Move the entire relocated program back to its final resting place.

Note: the example instructions in this chapter often include spaces for easier reading; do not type these spaces.



# CHAPTER 6

## RAM TEST

- 30 Testing APPLE's memory
- 31 Address ranges for standard memory configurations
- 32 Error messages
  - Type I - Simple error
  - Type II - Dynamic error
- 33 Testing for intermittent failure
- 34 Comments

## TESTING THE APPLE'S MEMORY

With this program, you can easily discover any problems in the RAM (for Random Access Memory) chips in your APPLE. This is especially useful when adding new memory. While a failure is a rare occurrence, memory chips are both quite complex and relatively expensive. This program will point out the exact memory chip or chips, if any, that have malfunctioned.

Memory chips are made in two types: one type can store 4K (4096) bits of information, the other can store 16K (16384) bits of information. Odd as it seems, the two types look alike, except for a code number printed on them.

The APPLE has provisions for inserting as many as 24 memory chips of either type into its main printed-circuit board, in three rows of eight sockets each. An eight-bit byte of information consists of one bit taken from each of the eight memory chips in a given row. For this reason, memory can be added only in units of eight identical memory chips at a time, filling an entire row. Eight 4K memory chips together in one row can store 4K bytes of information. Eight 16K memory chips in one row can store 16K bytes of information.

Inside the APPLE II, the three rows of sockets for memory chips are row "C", row "D" and row "E". The rows are lettered along the left edge of the printed-circuit board, as viewed from the front of the APPLE. The memory chips are installed in the third through the tenth sockets (counting from the left) of rows C, D and E. These sockets are labelled "RAM". Row C must be filled; and row E may be filled only if row D is filled. Depending on the configuration of your APPLE's memory, the eight RAM sockets in a given row of memory must be filled entirely with 4K memory chips, entirely with 16K memory chips, or all eight RAM sockets may be empty.

To test the memory chips in your computer, you must first initialize the RAM Test program. Press the RESET key to invoke the Monitor, and then type

D5BCG return

Next, specify the hexadecimal starting address for the portion of memory that you wish to test. You must also specify the hexadecimal number of "pages" of memory that you wish tested, beginning at the given starting address. A page of memory is 256 bytes (\$100 Hex). Representing the address by "a" and the number of pages by "p" (both in hexadecimal), start the RAM test by typing

a . p ctrl Y return

Note 1: to type "ctrl Y", type Y while holding down the CTRL key; ctrl Y is not displayed on the TV screen. Type "return" by pressing the RETURN key. The example instructions in this chapter often include spaces for easier reading; do not type these spaces.

Note 2: test length p\*100 must not be greater than starting address a.

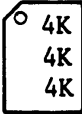

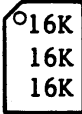
For example,

```
2000.10 ctrl Y return
```

tests hexadecimal 1000 bytes of memory (4096, or "4K" bytes, in decimal), starting at hexadecimal address 2000 (8192, or "8K", in decimal).

If the asterisk returns (after a delay that may be a half minute or so) without an error message (see ERROR MESSAGES discussion), then the specified portion of memory has tested successfully.

## TABLE OF ADDRESS RANGES FOR STANDARD RAM CONFIGURATIONS

If the 3 Memory Configuration Blocks Look like this:	Then Row of Memory	Contains this Range of Hexadecimal RAM Addresses	And the total System Memory, If this is last Row filled, is
	C	0000-0FFF	4K
	D	1000-1FFF	8K
	E	2000-2FFF	12K
	C	0000-3FFF	16K
	D	4000-4FFF	20K
	E	5000-5FFF	24K
	C	0000-3FFF	16K
	D	4000-7FFF	32K
	E	8000-BFFF	48K

A 4K RAM Row contains 10 Hex pages (hex 1000 bytes, or decimal 4096 bytes).  
 A 16K RAM Row contains 40 Hex pages (hex 4000 bytes, or decimal 16384 bytes).

A complete test for a 48K system would be as follows:

```
400.4 ctrl Y return } ← This tests the screen area of memory
800.8 ctrl Y return } ← These first four tests examine
1000.10 ctrl Y return } ← the first 16K row of memory (Row C)
2000.20 ctrl Y return }
4000.40 ctrl Y return } ← This tests the second 16K row of memory (Row D)
8000.40 ctrl Y return } ← This tests the third 16K row of memory (Row E)
```

Systems containing more than 16K of memory should also receive the following special test that looks for problems at the boundary between rows of memory:

```
3000.20 ctrl Y return
```

Systems containing more than 32K of memory should receive the previous special test, plus the following:

```
7000.20 ctrl Y return
```



Tests may be run separately or they may be combined into one instruction. For instance, for a 48K system you can type:

```
400.4 ctrl Y 800.8 ctrl Y 1000.10 ctrl Y 2000.20 ctrl Y 3000.20 ctrl Y
4000.40 ctrl Y 7000.20 ctrl Y 8000.40 ctrl Y return
```

Remember, ctrl Y will not print on the screen, but it must be typed. With the single exception noted in the section TESTING FOR INTERMITTENT FAILURE, spaces are shown for easier reading but should not be typed.

During a full test such as the one shown above, the computer will beep at the completion of each sub-test (each sub-test ends with a ctrl Y). At the end of the full test, if no errors have been found the APPLE will beep and the blinking cursor will return with the Monitor prompt character ( \* ). It takes approximately 50 seconds for the computer to test the RAM memory in a 16K system; larger systems will take proportionately longer.

## ERROR MESSAGES

### TYPE I - Simple Error

During testing, each memory address in the test range is checked by writing a particular number to it, then reading the number actually stored at that address and comparing the two.

A simple error occurs when the number written to a particular memory address differs from the number which is then read back from that same address. Simple errors are reported in the following format:

```
xxxx yy zz ERR r-c
```

where **xxxx** is the hexadecimal address at which the error was detected;  
**yy** is the hexadecimal data written to that address;  
**zz** is the hexadecimal data read back from that address; and  
**r-c** is the row and column where the defective memory chip was found. Count from the left, as viewed from the front of the APPLE: the leftmost memory chip is in column 3, the rightmost is in column 10.

Example:

```
201F 00 10 ERR D-7
```

**TYPE II - Dynamic Error**

This type of error occurs when the act of writing a number to one memory address causes the number read from a different address to change. If no simple error is detected at a tested address, all the addresses that differ from the tested address by one bit are read for changes indicating dynamic errors. Dynamic errors are reported in the following format:

```
xxxx yy zz vvvv qq ERR r-c
```

where **xxxx** is the hexadecimal address at which the error was detected;  
**yy** is the hexadecimal data written earlier to address **xxxx**;  
**zz** is the hexadecimal data now read back from address **xxxx**;  
**vvvv** is the current hexadecimal address to which data **qq** was successfully written;  
**qq** is the hexadecimal data successfully written to, and read back from, address **vvvv**; and  
**r-c** is the row and column where the defective memory chip was found. Count from the left, as viewed from the front of the APPLE: the leftmost memory chip is in column 3, the rightmost is in column 10. In this type of error, the indicated row (but not the column) may be incorrect.

This is similar to Type I, except that the appearance of **vvvv** and **qq** indicates an error was detected at address **xxxx** after data was successfully written at address **vvvv**.

**Example:**

```
5051 00 08 5451 00 ERR E-6
```

After a dynamic error, the indicated row (but not the column) may be incorrect. Determine exactly which tests check each row of chips (according to the range of memory addresses corresponding to each row), and run those tests by themselves. Confirm your diagnosis by replacing the suspected memory chip with a known good memory chip (you can use either a 4K or a 16K memory chip, for this replacement). Remember to turn off the APPLE's power switch and to discharge yourself before handling the memory chips.

## TESTING FOR INTERMITTENT FAILURE (Automatically Repeating Test)

This provides a way to test memory over and over again, indefinitely. You will type a complete series of tests, just as you did before, except that you will:

- a. precede the complete test with the letter N
- b. follow the complete test with 34:0
- c. type at least one space before pressing the RETURN key.

Here is the format:

N (memory test to be repeated) 34:0 (type one space) return

NOTE: You must type at least one space at the end of the line, prior to pressing the RETURN key. This is the only space that should be typed (all other spaces shown within instructions in this chapter are for easier reading only; they should not be typed).

Example (for a 48K system):

```
N 400.4 ctrl Y 800.8 ctrl Y 1000.10 ctrl Y 2000.20 ctrl Y 3000.20 ctrl Y
4000.40 ctrl Y 7000.20 ctrl Y 8000.40 ctrl Y 34:0 return
```

Run this test for at least one hour (preferably overnight) with the APPLE's lid in place. This allows the system and the memory chips to reach maximum operating temperature.

Only if a failure occurs will the APPLE display an error message and rapidly beep three times; otherwise, the APPLE will beep once at the successful end of each sub-test. To stop this repeating test, you must press the RESET key.

## COMMENTS:

1. You cannot test the APPLE's memory below the address of 400 (Hex), since various pointers and other system necessities are there. In any case, if that region of memory has problems, the APPLE won't function.
2. For any subtest, the number of pages tested cannot be greater than the starting address divided by 100 Hex. 2000.30 ctrl Y will not work, but 5000.30 ctrl Y will.
3. Before changing anything inside the APPLE, make sure the APPLE is plugged into a grounded, 3-wire power outlet, and that the power switch on the back of the computer is turned off. Always touch the outside metal bottom plate of the APPLE II, prior to handling any memory chips. This is done to remove any static charge that you may have acquired.

### EVEN A SMALL STATIC CHARGE CAN DESTROY MEMORY CHIPS

4. Besides the eight memory chips, some additions of memory require changing three other chip-like devices called Memory Configuration Blocks. The Memory Configuration Blocks tell the APPLE which type of memory chip (4K or 16K) is to be plugged into each row of memory. A complete package for adding memory to your computer, containing all necessary parts and detailed instructions, can be purchased from APPLE Computer Inc. To add 4K of memory, order the 4K Memory Expansion Module (P/N A2M0014). To add 16K of memory, order the 16K Memory Expansion Module (P/N A2M0016).

# CHAPTER 7 MUSIC

- 36 Generating musical tones
- 37 Comments

## GENERATING MUSICAL TONES

The Music feature is most easily used from within an Integer BASIC program. It greatly simplifies the task of making the APPLE II into a music-playing device.

There are three things the computer needs to know before playing a note: pitch (how high or low a note), duration (how long a time it is to sound), and timbre. Timbre is the quality of a sound that allows you to distinguish one instrument from another even if they are playing at the same pitch and loudness. This Music feature does not permit control of loudness.

It is convenient to set up a few constants early in the program:

```
MUSIC = -10473
PITCH = 767      NOTE      $80C
TIME = 766      LONG      $845
TIMBRE = 765    VOICE     $84A
```

There are 50 notes available, numbered from 1 to 50. The statement

```
POKE PITCH, 32
```

will set up the Music feature to produce (approximately) the note middle C. Increasing the pitch value by one increases the pitch by a semitone. Thus

```
POKE PITCH, 33
```

would set up the Music feature to produce the note C sharp. Just over four chromatic octaves are available. The note number 0 indicates a rest (a silence) rather than a pitch.

The duration of the note is set by

```
POKE TIME, t
```

Where t is a number from 1 to 255. The higher the number, the longer the note. A choice of t = 170 gives notes that are approximately one second long. To get notes at a metronome marking of MM, use a duration of 10200/MM. For example, to get 204 notes per minute (approximately) use the command

```
POKE TIME, 10200/204
```

There are five timbres, coded by the numbers 2, 8, 16, 32 and 64. They are not very different from one another. With certain timbres, a few of the extremely low or high notes do not give the correct pitch. Timbre 32 does not have this problem.

```
POKE TIMBRE, 32
```

When the pitch, time, and timbre have been set, the statement

```
CALL MUSIC
```

will cause the specified note to sound.

The following program plays a chromatic scale of four octaves:

```
10 MUSIC = -10473: PITCH = 767: TIME = 766: TIMBRE = 765
20 POKE TIME, 40: POKE TIMBRE, 32
30 FOR I = 1 TO 49
40 POKE PITCH, I
50 CALL MUSIC
60 NEXT I: END
```

Where X is a number from 51 through 255,

```
POKE PITCH, X
```

will specify various notes, in odd sequences. In the program above, change line 40 to

```
40 POKE PITCH, 86
```

for a demonstration.

## COMMENTS:

Some extremely high or low notes will come out at the wrong pitch with certain timbres.



# CHAPTER 8

# HIGH-RESOLUTION GRAPHICS

- 40 Part A: Setting up parameters, subroutines, and colors
  - 40 Positioning the High-Resolution parameters
  - 41 Defining subroutine names
  - 41 Defining color names
  - 42 Speeding up your program
  
- 43 Part B: Preparing the screen for graphics
  - 43 The INITIALization subroutine
  - 43 Changing the graphics screen
  - 44 CLEARing the screen to black
  - 44 Coloring the BACKGrouND
  
- 45 Part C: PLOTting points and LINEs
  
- 46 Part D: Creating, saving and loading shapes
  - 46 Introduction
  - 47 Creating a Shape Table
  - 53 Saving a Shape Table
  - 54 Loading a Shape Table
  - 55 First use of Shape Table
  
- 56 Part E: Drawing shapes from a prepared Shape Table
  - 56 Assigning parameter values: SHAPE, SCALE AND ROTation
  - 57 DRAWing shapes
  - 58 Linking shapes: DRAW1
  - 59 Collisions
  
- 60 Part F: Technical information
  - 60 Locations of the High-Resolution parameters
  - 61 Variables used within the High-Resolution subroutines
  - 62 Shape Table information
  - 63 Integer BASIC memory map for graphics
  
- 64 Part G: Comments



## PART A: SETTING UP PARAMETERS, SUBROUTINES, AND COLORS

Programmer's Aid #1 provides your APPLE with the ability to do high-resolution color graphics from Integer BASIC. You may plot dots, lines and shapes in a wide variety of detailed forms, in 6 different colors (4 colors on systems below S/N 60000), displayed from two different "pages" of memory. The standard low-resolution graphics allowed you to plot 40 squares across the screen by 47 squares from top to bottom of the screen. This high-resolution graphics display mode lets you plot in much smaller dots, 280 horizontally by 192 vertically. Because 8K bytes of memory (in locations from 8K to 16K, for Page 1) are dedicated solely to maintaining the high-resolution display, your APPLE must contain at least 16K bytes of memory. To use the Page 2 display (in locations from 16K to 24K), a system with at least 24K bytes of memory is needed. If your system is using the Disk Operating System (DOS), that occupies the top 10.5K of memory: you will need a minimum 32K system for Page 1, or 36K for Page 1 and Page 2. See the MEMORY MAP on page 63 for more details.

### POSITIONING THE HIGH-RESOLUTION PARAMETERS

The first statement of an Integer BASIC program intending to use the Programmer's Aid High-Resolution subroutines should be:

```
0 X0 = Y0 = COLR = SHAPE = ROT = SCALE
```

The purpose of this statement is simply to place the six BASIC variable names used by the High-Resolution feature (with space for their values) into APPLE's "variable table" in specific, known locations. When line 0 is executed, the six High-Resolution graphics parameters will be assigned storage space at the very beginning of the variable table, in the exact order specified in line 0. Your BASIC program then uses those parameter names to change the six parameter values in the variable table. However, the High-Resolution subroutines ignore the parameter names, and look for the parameter values in specific variable-table locations. That is why the program's first line must place the six High-Resolution graphics parameters in known variable-table locations. Different parameter names may be used, provided that they contain the same number of characters. Fixed parameter-name lengths are also necessary to insure that the parameter-value storage locations in the variable table do not change. For example, the name HI could be used in place of X0, but X or XCOORD could not.

The parameters SHAPE, ROT, and SCALE are used only by the subroutines that draw shapes (DRAW and DRAW1, see PART E). These parameters may be omitted from programs using only the PLOT and LINE features:

```
Ø XØ = YØ = COLR
```

Omitting unnecessary parameter definitions speeds up the program during execution. However, you can omit only those unused parameters to the right of the last parameter which is used. Each parameter that is used must be in its proper place, relative to the first parameter in the definition list.

## DEFINING SUBROUTINE NAMES

After the six parameters have been defined, the twelve High-Resolution subroutines should be given names, and these names should be assigned corresponding subroutine entry addresses as values. Once defined in this way, the various subroutines can be called by name each time they are used, rather than by numeric address. When subroutines are called by name, the program is easier to type, more likely to be error-free, and easier to follow and to debug.

```
5 INIT = -12288 : CLEAR = -12274 : BKGND = -11471
6 POSN = -11527 : PLOT = -115Ø6 : LINE = -115ØØ
7 DRAW = -11465 : DRAW1 = -11462
8 FIND = -1178Ø : SHLOAD = -11335
```

Any variable names of any length may be used to call these subroutines. If you want maximum speed, do not define names for subroutines that you will not use in your program.

## DEFINING COLOR NAMES

Colors may also be specified by name, if a defining statement is added to the program. Note that GREEN is preceded by LET to avoid a SYNTAX ERROR, due to conflict with the GR command.

```
1Ø BLACK = Ø : LET GREEN = 42 : VIOLET = 85
11 WHITE = 127 : ORANGE = 17Ø : BLUE = 213
12 BLACK2 = 128 : WHITE2 = 255
```

Any integer from Ø through 255 may be used to specify a color, but most of the numbers not named above give rather unsatisfactory "colors". On systems below S/N 6ØØØ, 17Ø will appear as green and 213 will appear as violet.

Once again, unnecessary variable definitions should be omitted, as they will slow some programs. Therefore, a program should not define VIOLET = 85 unless it uses the color VIOLET.

The following example illustrates condensed initialization for a program using only the INIT, PLOT, and DRAW subroutines, and the colors GREEN and WHITE.

```

0 X0 = Y0 = COLR = SHAPE = ROT = SCALE
5 INIT = -12288 : PLOT = -11506 : DRAW = -11465
10 LET GREEN = 42 : WHITE = 127

```

(Body of program would go here)

## SPEEDING UP YOUR PROGRAM

Where maximum speed of execution is necessary, any of the following techniques will help:

1. Omit the name definitions of colors and subroutines, and refer to colors and subroutines by numeric value, not by name.
2. Define the most frequently used program variable names before defining the subroutine and color names (lines 5 through 12 in the previous examples). The example below illustrates how to speed up a program that makes very frequent use of program variables I, J, and K:

```

0 X0 = Y0 = COLR = SHAPE = ROT = SCALE
2 I = J = K
5 INIT = -12288 : CLEAR = -12274
6 BKGND = -11471 : POSN = -11527
10 BLACK = 0 : VIOLET = 85

```

3. Use the High-Resolution graphics parameter names as program variables when possible. Because they are defined first, these parameters are the BASIC variables which your program can find fastest.

## PART B: PREPARING THE SCREEN FOR GRAPHICS

### THE INITIALIZATION SUBROUTINE

In order to use CLEAR, BKGND, POSN, PLOT, or any of the other High-Resolution subroutine CALLs, the INITIALIZATION subroutine itself must first be CALLED:

```
CALL INIT
```

The INITIALIZATION subroutine turns on the high-resolution display and clears the high-resolution screen to black. INIT also sets up certain variables necessary for using the other High-Resolution subroutines. The display consists of a graphics area that is 280 x-positions wide (X0=0 through X0=279) by 160 y-positions high (Y0=0 through Y0=159), with an area for four lines of text at the bottom of the screen. Y0 values from 0 through 191 may be used, but values greater than 159 will not be displayed on the screen. The graphics origin (X0=0, Y0=0) is at the top left corner of the screen.

### CHANGING THE GRAPHICS SCREEN

If you wish to devote the entire display to graphics (280 x-positions wide by 192 y-positions high), use

```
POKE -16302, 0
```

The split graphics-plus-text mode may be restored at any time with

```
POKE -16301, 0
```

or another

```
CALL INIT
```

When the High-Resolution subroutines are first initialized, all graphics are done in Page 1 of memory (\$2000-3FFF), and only that page of memory is displayed. If you wish to use memory Page 2 (\$4000-5FFF), two POKES allow you to do so:

```
POKE 806, 64
```

causes subsequent graphics instructions to be executed in Page 2, unless those instructions attempt to continue an instruction from Page 1 (for instance, a LINE is always drawn on the same memory page where the last previous point was plotted). After this POKE, the display will still show memory Page 1.

To see what you are plotting on Page 2,

```
POKE -16299, 0
```

will cause Page 2 to be displayed on the screen. You can switch the screen display back to memory Page 1 at any time, with

```
POKE -16300, 0
```

while

```
POKE 806, 32
```

will return you to Page 1 plotting. This last POKE is executed automatically by INIT.

## CLEARING THE SCREEN

If at any time during your program you wish to clear the current plotting page to black, use

```
CALL CLEAR
```

This immediately erases anything plotted on the current plotting page. INIT first resets the current plotting page to memory Page 1, and then clears Page 1 to black.

The entire current plotting page can be set to any solid background color with the BKGND subroutine. After you have INITIALIZED the High-Resolution subroutines, set COLR to the background color you desire, and then

```
CALL BKGND
```

The following program turns the entire display violet:

```
0 X0 = Y0 = COLR : REM SET PARAMETERS
5 INIT = -12288 : BKGND = -11471 : REM DEFINE SUBROUTINES
10 VIOLET = 85 : REM DEFINE COLOR
20 CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
30 COLR = VIOLET : REM ASSIGN COLOR VALUE
40 CALL BKGND : REM MAKE ALL OF DISPLAY VIOLET
50 END
```

## PART C: PLOTTING POINTS AND LINES

Points can be plotted anywhere on the high-resolution display, in any valid color, with the use of the PLOT subroutine. The PLOT subroutine can only be used after a CALL INIT has been executed, and after you have assigned appropriate values to the parameters XØ, YØ and COLR. XØ must be in the range from 0 through 279, YØ must be in the range from 0 through 191, and COLR must be in the range from 0 through 255, or a

\*\*\* RANGE ERR

message will be displayed and the program will halt.

The program below plots a white dot at X-coordinate 35, Y-coordinate 55, and a violet dot at X-coordinate 85, Y-coordinate 90:

```

Ø XØ = YØ = COLR : REM SET PARAMETERS
5 INIT = -12288 : PLOT = -11506 : REM DEFINE SUBROUTINES
1Ø WHITE = 127 : VIOLET = 85 : REM DEFINE COLORS
2Ø CALL INIT : REM INITIALIZE SUBROUTINES
3Ø COLR = WHITE : REM ASSIGN PARAMETER VALUES
4Ø XØ = 35 : YØ = 55
5Ø CALL PLOT : REM PLOT WITH ASSIGNED PARAMETER VALUES
6Ø COLR = VIOLET : REM ASSIGN NEW PARAMETER VALUES
7Ø XØ = 85 : YØ = 90
8Ø CALL PLOT : REM PLOT WITH NEW PARAMETER VALUES
9Ø END

```

The subroutine POSN is exactly like PLOT, except that nothing is placed on the screen. COLR must be specified, however, and a subsequent DRAW1 (see PART E) will take its color from the color used by POSN. This subroutine is often used when establishing the origin-point for a LINE.

Connecting any two points with a straight line is done with the LINE subroutine. As with the PLOT subroutine, a CALL INIT must be executed, and XØ, YØ, and COLR must be specified. In addition, before the LINE subroutine can be CALLED, the line's point of origin must have been plotted with a CALL PLOT or as the end point of a previous line or shape. Do not attempt to use CALL LINE without first plotting a point for the line's origin, or the line may be drawn in random memory locations, not necessarily restricted to the current memory page. Once again, XØ and YØ (the coordinates of the termination point for the line), and COLR must be assigned legitimate values, or an error may occur.

The following program draws a grid of green lines vertically and violet lines horizontally, on a white background:

```

0 X0 = Y0 = COLR : REM SET PARAMETERS, THEN DEFINE SUBROUTINES
5 INIT = -12288 : BKGND = -11471 : PLOT = -11506 : LINE = -11500
10 LET GREEN = 42 : VIOLET = 85 : WHITE = 127 : REM DEFINE COLORS
20 CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
30 POKE -16302, 0 : REM SET FULL-SCREEN GRAPHICS
40 COLR = WHITE : CALL BKGND : REM MAKE THE DISPLAY ALL WHITE
50 COLR = GREEN : REM ASSIGN PARAMETER VALUES
60 FOR X0 = 0 TO 270 STEP 10
70 Y0 = 0 : CALL PLOT : REM PLOT A STARTING-POINT AT TOP OF SCREEN
80 Y0 = 190 : CALL LINE : REM DRAW A VERTICAL LINE TO BOTTOM OF SCREEN
90 NEXT X0 : REM MOVE RIGHT AND DO IT AGAIN
100 COLR = VIOLET : REM ASSIGN NEW PARAMETER VALUES
110 FOR Y0 = 0 TO 190 STEP 10
120 X0 = 0 : CALL PLOT : REM PLOT A STARTING-POINT AT LEFT EDGE OF SCREEN
130 X0 = 270 : CALL LINE : REM PLOT A HORIZONTAL LINE TO RIGHT EDGE
140 NEXT Y0 : REM MOVE DOWN AND DO IT AGAIN
150 END

```

## PART D: CREATING, SAVING AND LOADING SHAPES

### INTRODUCTION

The High-Resolution feature's subroutines provide the ability to do a wide range of high-resolution graphics "shape" drawing. A "shape" is considered to be any figure or drawing (such as an outline of a rocket ship) that the user wishes to draw on the display many times, perhaps in different sizes, locations and orientations. Up to 255 different shapes may be created, used, and saved in a "Shape Table", through the use of the High-Resolution subroutines DRAW, DRAW1 and SHLOAD, in conjunction with parameters SHAPE, ROT and SCALE.

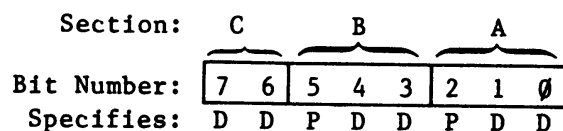
In this section, PART D, you will be shown how to create, save and load a Shape Table. The following section, PART E, demonstrates the use of the shape-drawing subroutines with a predefined Shape Table.

## HOW TO CREATE A SHAPE TABLE

Before the High-Resolution shape-drawing subroutines can be used, a shape must be defined by a "shape definition." This shape definition consists of a sequence of plotting vectors that are stored in a series of bytes in APPLE's memory. One or more such shape definitions, with their index, make up a "Shape Table" that can be created from the keyboard and saved on disk or cassette tape for future use.

Each byte in a shape definition is divided into three sections, and each section can specify a "plotting vector": whether or not to plot a point, and also a direction to move (up, down, left, or right). The shape-drawing subroutines DRAW and DRAW1 (see PART E) step through each byte in the shape definition section by section, from the definition's first byte through its last byte. When a byte that contains all zeros is reached, the shape definition is complete.

This is how the three sections A, B and C are arranged within one of the bytes that make up a shape definition:



Each bit pair DD specifies a direction to move, and each bit P specifies whether or not to plot a point before moving, as follows:

If DD = 00 move up	If P = 0 don't plot
= 01 move right	= 1 do plot
= 10 move down	
= 11 move left	

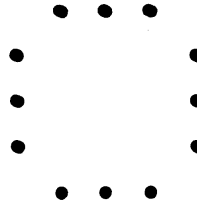
Notice that the last section, C (the two most significant bits), does not have a P field (by default, P=0), so section C can only specify a move without plotting.

Each byte can represent up to three plotting vectors, one in section A, one in section B, and a third (a move only) in section C.

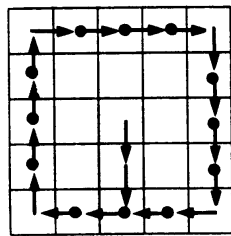
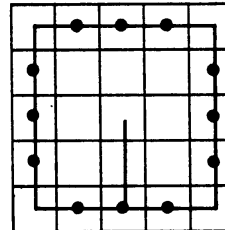
DRAW and DRAW1 process the sections from right to left (least significant bit to most significant bit: section A, then B, then C). At any section in the byte, IF ALL THE REMAINING SECTIONS OF THE BYTE CONTAIN ONLY ZEROS, THEN THOSE SECTIONS ARE IGNORED. Thus, the byte cannot end with a move in section C of 00 (a move up, without plotting) because that section, containing only zeros, will be ignored. Similarly, if section C is 00 (ignored), then section B cannot be a move of 000 as that will also be ignored. And a move of 000 in section A will end your shape definition unless there is a 1-bit somewhere in section B or C.



Suppose you want to draw a shape like this:



First, draw it on graph paper, one dot per square. Then decide where to start drawing the shape. Let's start this one at the center. Next, draw a path through each point in the shape, using only 90 degree angles on the turns:



Next, re-draw the shape as a series of plotting vectors, each one moving one place up, down, right, or left, and distinguish the vectors that plot a point before moving (a dot marks vectors that plot points).

Now "unwrap" those vectors and write them in a straight line:



Next draw a table like the one in Figure 1, below:

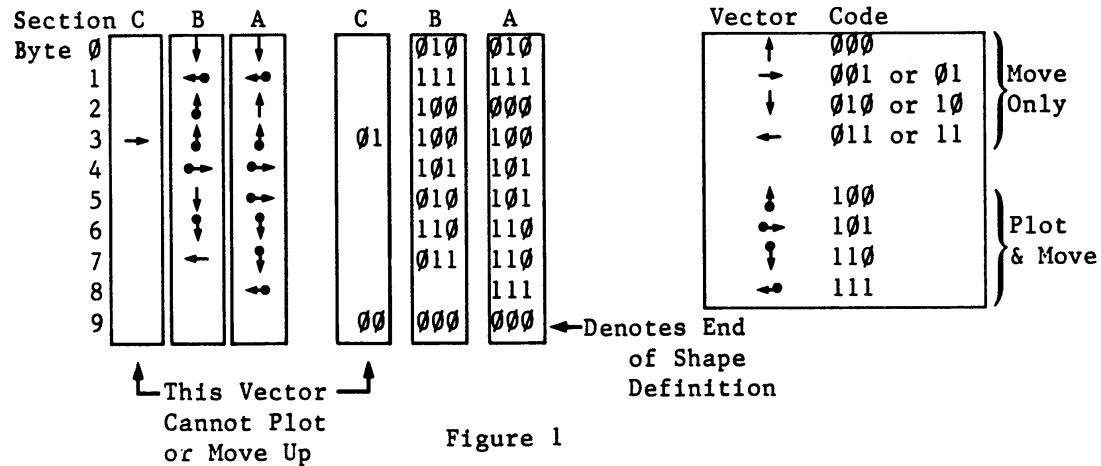


Figure 1

For each vector in the line, determine the bit code and place it in the next available section in the table. If the code will not fit (for example, the vector in section C can't plot a point), or is a 00 (or 000) at the end of a byte, then skip that section and go on to the next. When you have finished coding all your vectors, check your work to make sure it is accurate.

Now make another table, as shown in Figure 2, below, and re-copy the vector codes from the first table. Recode the vector information into a series of hexadecimal bytes, using the hexadecimal codes from Figure 3.

Section:	C	B	A	Bytes Recoded in Hex	Codes
Byte 0	0 0 0 1	0 0 1 0	0 0 1 0	= 1 2	0000 = 0
1	0 0 1 1	1 1 1 1		= 3 F	0001 = 1
2	0 0 1 0	0 0 0 0		= 2 0	0010 = 2
3	0 1 1 0	0 1 0 0		= 6 4	0011 = 3
4	0 0 1 0	1 1 0 1		= 2 D	0100 = 4
5	0 0 0 1	0 1 0 1		= 1 5	0101 = 5
6	0 0 1 1	0 1 1 0		= 3 6	0110 = 6
7	0 0 0 1	1 1 1 0		= 1 E	0111 = 7
8	0 0 0 0	0 1 1 1		= 0 7	1000 = 8
9	0 0 0 0	0 0 0 0		= 0 0 ← Denotes End of Shape Definition	1001 = 9
Hex:	Digit 1	Digit 2			1010 = A
					1011 = B
					1100 = C
					1101 = D
					1110 = E
					1111 = F

Figure 2

Figure 3

The series of hexadecimal bytes that you arrived at in Figure 2 is the shape definition. There is still a little more information you need to provide before you have a complete Shape Table. The form of the Shape Table, complete with its index, is shown in Figure 4 on the next page.

For this example, your index is easy: there is only one shape definition. The Shape Table's starting location, whose address we have called S, must contain the number of shape definitions (between 0 and 255) in hexadecimal. In this case, that number is just one. We will place our shape definition immediately below the index, for simplicity. That means, in this case, the shape definition will start in byte S+4: the address of shape definition #1, relative to S, is 4 (00 04, in hexadecimal). Therefore, index byte S+2 must contain the value 04 and index byte S+3 must contain the value 00. The completed Shape Table for this example is shown in Figure 5 on the next page.

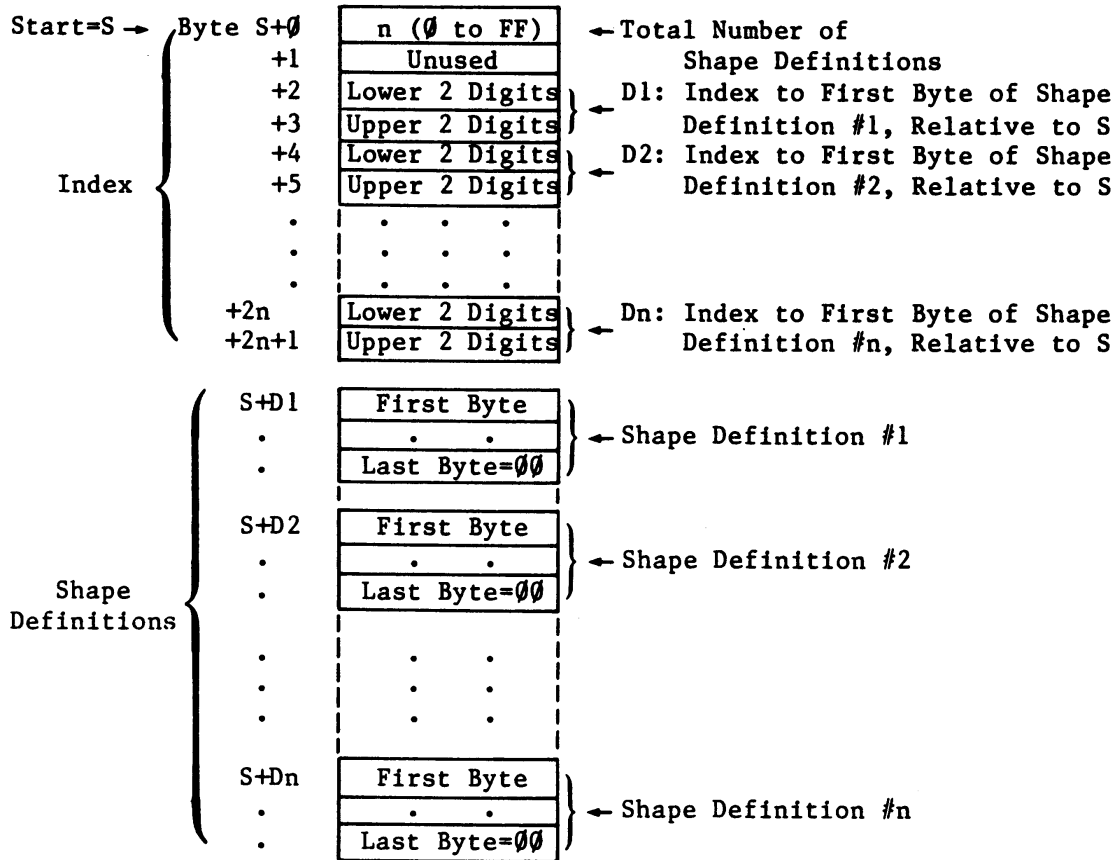


Figure 4

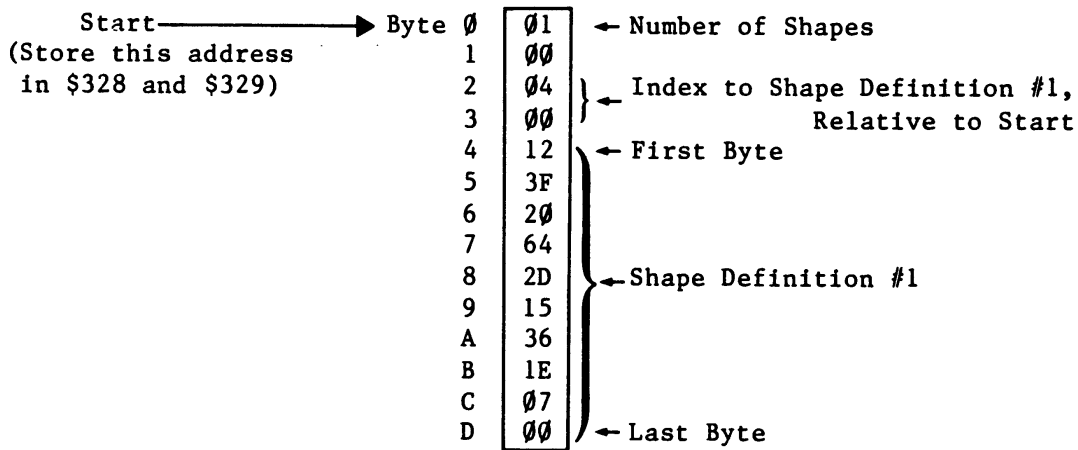


Figure 5

You are now ready to type the Shape Table into APPLE's memory. First, choose a starting address. For this example, we'll use hexadecimal address 0800.

Note: this address must be less than the highest memory address available in your system (HIMEM), and not in an area that will be cleared when you use memory Page 1 (hexadecimal locations \$2000 to \$4000) or Page 2 (hexadecimal locations \$4000 to \$6000) for high-resolution graphics. Furthermore, it must not be in an area of memory used by your BASIC program. Hexadecimal 0800 (2048, in decimal) is the lowest memory address normally available to a BASIC program. This lowest address is called LOMEM. Later on, we will move the LOMEM pointer higher, to the end of our Shape Table, in order to protect our table from BASIC program variables.

Press the RESET key to enter the Monitor program, and type the Starting address for your Shape Table:

0800

If you press the RETURN key now, APPLE will show you the address and the contents of that address. That is how you examine an address to see if you have a put the correct number there. If instead you type a colon ( : ) followed by a two-digit hexadecimal number, that number will be stored at the specified address when you press the RETURN key. Try this:

0800 return

(type "return" by pressing the RETURN key). What does APPLE say the contents of location 0800 are? Now try this:

0800:01 return  
 0800 return  
 0800- 01

The APPLE now says that the value 01 (hexadecimal) is stored in the location whose address is 0800. To store more two-digit hexadecimal numbers in successive bytes in memory, just open the first address:

0800:

and then type the numbers, separated by spaces:

0800:01 00 04 00 12 3F 20 64 2D 15 36 1E 07 00 return

You have just typed your first complete Shape Table...not so bad, was it? To check the information in your Shape Table, you can examine each byte separately or simply press the RETURN key repeatedly until all the bytes of interest (and a few extra, probably) have been displayed:

```

0800 return
0800- 01
return
00 04 00 12 3F 20 64
return
0808- 2D 15 36 1E 07 00 FF FF

```

If your Shape Table looks correct, all that remains is to store the starting address of the Shape Table where the shape-drawing subroutines can find it (this is done automatically when you use the SHLOAD subroutine to get a table from cassette tape). Your APPLE looks for the four hexadecimal digits of the table's starting address in hexadecimal locations 328 (lower two digits) and 329 (upper two digits). For our table's starting address of 08 00, this would do the trick:

```
328:00 08
```

To protect this Shape Table from being erased by the variables in your BASIC program, you must also set LOMEM (the lowest memory address available to your program) to the address that is one byte beyond the Shape Table's last, or largest, address.

It is best to set LOMEM from BASIC, as an immediate-execution command issued before the BASIC program is RUN. LOMEM is automatically set when you invoke BASIC (reset ctrl B return) to decimal 2048 (0800, in hexadecimal). You must then change LOMEM to 2048 plus the number of bytes in your Shape Table plus one. Our Shape Table was decimal 14 bytes long, so our immediate-execution BASIC command would be:

```
LOMEM: 2048 + 15
```

Fortunately, all of this (entering the Shape Table at LOMEM, resetting LOMEM to protect the table, and putting the table's starting address in \$328-\$329) is taken care of automatically when you use the High-Resolution feature's SHLOAD subroutine to get the table from cassette tape.

## SAVING A SHAPE TABLE

### Saving on Cassette Tape

To save your Shape Table on tape, you must be in the Monitor and you must know three hexadecimal numbers:

- 1) Starting Address of the table (0800, in our example)
- 2) Last Address of the table (080D, in our example)
- 3) Difference between 2) and 1) (000D, in our example)

Item 3, the difference between the last address and the first address of the table, must be stored in hexadecimal locations 0 (lower two digits) and 1 (upper two digits):

```
0:0D 00 return
```

Now you can "Write" (store on cassette) first the table length that is stored in locations 0 and 1, and then the Shape Table itself that is stored in locations Starting Address through Last Address:

```
0.1W 0800.080DW
```

Don't press the RETURN key until you have put a cassette in your tape recorder, rewound it, and started it recording (press PLAY and RECORD simultaneously). Now press the computer's RETURN key.

### Saving on Disk

To save your Shape Table on disk, use a command of this form:

```
B$AVE filename, A$ startingaddress, L$ tablelength
```

For our example, you might type

```
B$AVE MYSHAPE1, A$ 0800, L$ 000D
```

**Note:** the Disk Operating System (DOS) occupies the top 10.5K of memory (10752 bytes decimal, or \$2A00 hex); make sure your Shape Table is not in that portion of memory when you "boot" the disk system.

## LOADING A SHAPE TABLE

### Loading from Cassette Tape

To load a Shape Table from cassette tape, rewind the tape, start it playing (press PLAY), and (in BASIC, now) type

```
CALL -11335 return
```

or (if you have previously assigned the value -11335 to the variable SHLOAD)

```
CALL SHLOAD return
```

You should hear one "beep" when the table's length has been read successfully, and another "beep" when the table itself has been read. When loaded this way, your Shape Table will load into memory, beginning at hexadecimal address 0800. LOMEM is automatically changed to the address of the location immediately following the last Shape-Table byte. Hexadecimal locations 328 and 329 are automatically set to contain the starting address of the Shape Table.

### Loading from Disk

To load a Shape Table from disk, use a command of the form

```
BLOAD filename
```

From our previously-saved example, you would type

```
BLOAD MYSHAPE1
```

This will load your Shape Table into memory, beginning at the address you specified after "A\$" when you BSAVED the Shape Table earlier. In our example, MYSHAPE1 would BLOAD beginning at address 0800. You must store the Shape Table's starting address in hexadecimal locations 328 and 329, yourself, from the Monitor:

```
328:00 08 return
```

If your Shape Table is in an area of memory that may be used by your BASIC program (as our example is), you must protect the Shape Table from your program. Our example lies at the low end of memory, so we can protect it by raising LOMEM to just above the last byte of the Shape Table. This must be done after invoking BASIC (reset ctrl B return) and before RUNNING our BASIC program. We could do this with the immediate-execution BASIC command

```
LOMEM: 2048 + 15
```

## FIRST USE OF A SHAPE TABLE

You are now ready to write a BASIC program using Shape-Table subroutines such as DRAW and DRAW1. For a full discussion of these High-Resolution subroutines, see the following section, PART E.

Remember that Page 1 graphics uses memory locations 8192 through 16383 (8K to 16K), and Page 2 graphics uses memory locations 16384 through 24575 (16K to 24K). Integer BASIC puts your program right at the top of available memory; so if your APPLE contains less than 32K of memory, you should protect your program by setting HIMEM to 8192. This must be done after you invoke BASIC (reset ctrl B return) and before RUNNING your program, with the immediate-execution command

```
HIMEM: 8192
```

Here's a sample program that assumes our Shape Table has already been loaded from tape, using CALL SHLOAD. This program will print our defined shape, rotate it 5.6 degrees if that rotation is recognized (see ROT discussion, next section) and then repeat, each repetition larger than the one before.

```
10 X0 = Y0 = COLR = SHAPE = ROT = SCALE : REM SET PARAMETERS
20 INIT = -12288 : DRAW = -11465 : REM DEFINE SUBROUTINES
30 WHITE = 127 : BLACK = 0 : REM DEFINE COLORS
40 CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
50 SHAPE = 1
60 X0 = 139 : Y0 = 79 : REM ASSIGN PARAMETER VALUES
70 FOR R = 1 TO 48
80 ROT = R
90 SCALE = R
100 COLR = WHITE
110 CALL DRAW : REM DRAW SHAPE 1 WITH ABOVE PARAMETERS
120 NEXT R : REM NEW PARAMETERS
130 END
```

To pause, and then erase each square after it is drawn, add these lines:

```
114 FOR PAUSE = 1 TO 200 : NEXT PAUSE
116 COLR = BLACK : REM CHANGE COLOR
118 CALL DRAW : REM RE-DRAW SAME SHAPE, IN NEW COLOR
```



## PART E: DRAWING SHAPES FROM A PREPARED SHAPE TABLE

Before either of the two shape-drawing subroutines DRAW or DRAW1 can be used, a "Shape Table" must be defined and stored in memory (see PART E: CREATING A SHAPE TABLE), the Shape Table's starting address must be specified in hexadecimal locations 328 and 329 (808 and 809, in decimal), and the High-Resolution subroutines themselves must have been initialized by a CALL INIT.

### ASSIGNING PARAMETER VALUES

The DRAW subroutine is used to display any of the shapes defined in the current Shape Table. The origin or 'beginning point' for DRAWing the shape is specified by the values assigned to X0 and Y0, and the rest of the shape continues from that point. The color of the shape to be DRAWn is specified by the value of COLR.

The shape number (the Shape Table's particular shape definition that you wish to have DRAWn) is specified by the value of SHAPE. For example,

```
SHAPE = 3
```

specifies that the next shape-drawing command will use the third shape definition in the Shape Table. SHAPE may be assigned any value (from 1 through 255) that corresponds to one of the shape definitions in the current Shape Table. An attempt to DRAW a shape that does not exist (by executing a shape-drawing command after setting SHAPE = 4, when there are only two shape definitions in your Shape Table, for instance) will result in a \*\*\* RANGE ERR message being displayed, and the program will halt.

The relative size of the shape to be DRAWn is specified by the value assigned to SCALE. For example,

```
SCALE = 4
```

specifies that the next shape DRAWn will be four times the size that is described by the appropriate shape definition. That is, each "plotting vector" (either a plot and a move, or just a move) will be repeated four times. SCALE may be assigned any value from 0 through 255, but SCALE = 0 is interpreted as SCALE = 256, the largest size for a given shape definition.

You can also specify the orientation or angle of the shape to be DRAWn, by assigning the proper value to ROT. For example,

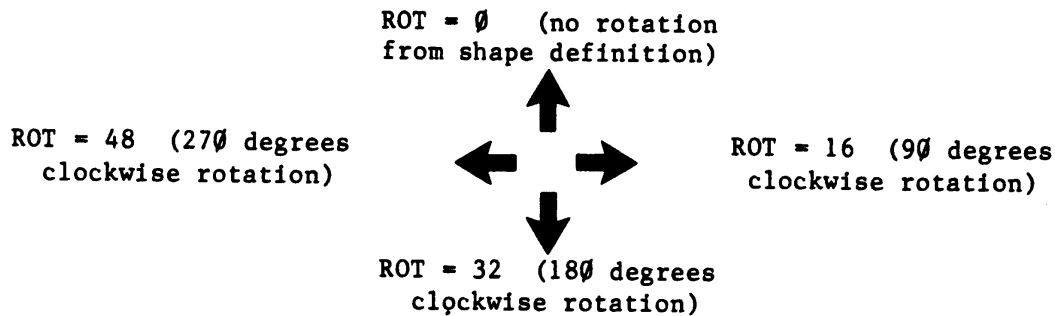
ROT = 0

will cause the next shape to be DRAWn oriented just as it was defined, while

ROT = 16

will cause the next shape to be DRAWn rotated 90 degrees clockwise. The value assigned to ROT must be within the range 0 to 255 (although ROT=64, specifying a rotation of 360 degrees clockwise, is the equivalent of ROT=0). For SCALE=1, only four of the 63 different rotations are recognized (0,16,32,48); for SCALE=2, eight different rotations are recognized; etc. ROT values specifying unrecognized rotations will usually cause the shape to be DRAWn with the next smaller recognized rotation.

## ORIENTATIONS OF SHAPE DEFINITION



## DRAWING SHAPES

The following example program DRAWs shape definition number three, in white, at a 135 degree clockwise rotation. Its starting point, or origin, is at (140,80).

```

0 X0 = Y0 = COLR = SHAPE = ROT = SCALE : REM SET PARAMETERS
5 INIT = -12288 : DRAW = -11465 : REM DEFINE SUBROUTINES
10 WHITE = 127 : REM DEFINE COLOR
20 CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
30 X0 = 140 : Y0 = 80 : COLR = WHITE : REM ASSIGN PARAMETER VALUES
40 SHAPE = 3 : ROT = 24 : SCALE = 2
50 CALL DRAW : REM DRAW SHAPE 3, DOUBLE SIZE, TURNED 135 DEGREES
60 END
    
```

## LINKING SHAPES

`DRAW1` is identical to `DRAW`, except that the last point previously `DRAWn`, `PLOT`ted or `POS`Ned determines the color and the starting point for the new shape. `XØ`, `YØ`, and `COLR`, need not be specified, as they will have no effect on `DRAW1`. However, some point must have been plotted before `CALL`ing `DRAW1`, or this `CALL` will have no effect.

The following example program draws "squiggles" by `DRAW`ing a small shape whose orientation is given by game control #Ø, then linking a new shape to the old one, each time the game control gives a new orientation. To clear the screen of "squiggles," press the game-control button.

```

1Ø XØ = YØ = COLR = SHAPE = ROT = SCALE : REM SET PARAMETERS
2Ø INIT = -12288 : DRAW = -11465 : DRAW1 = -11462
22 CLEAR = -12274 : WHITE = 127 : REM NAME SUBROUTINES AND COLOR
3Ø FULLSCREEN = -163Ø2 : BUTN = -16287 : REM NAME LOCATIONS
4Ø CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
5Ø POKE FULLSCREEN, Ø : REM SET FULL-SCREEN GRAPHICS
6Ø COLR = WHITE : SHAPE = 1 : SCALE = 5
7Ø XØ = 14Ø : YØ = 8Ø : REM ASSIGN PARAMETER VALUES
8Ø CALL CLEAR : ROT = PDL(Ø) : CALL DRAW : REM DRAW FIRST SHAPE
9Ø IF PEEK(BUTN) > 127 THEN GOTO 8Ø : REM PRESS BUTTON TO CLEAR SCREEN
1ØØ R = PDL(Ø) : IF (R < ROT+2) AND (R > ROT-2) THEN GOTO 9Ø :
    REM WAIT FOR CHANGE IN GAME CONTROL
11Ø ROT = R : CALL DRAW1 : REM ADD TO "SQUIGGLE"
12Ø GOTO 9Ø : REM LOOK FOR ANOTHER CHANGE

```

After `DRAW`ing a shape, you may wish to draw a `LINE` from the last plotted point of the shape to another fixed point on the screen. To do this, once the shape is `DRAWn`, you must first use

```
CALL FIND
```

prior to `CALL`ing `LINE`. The `FIND` subroutine determines the X and Y coordinates of the final point in the shape that was `DRAWn`, and uses it as the beginning point for the subsequent `CALL LINE`.

The following example **DRAWs** a white shape, and then draws a violet **LINE** from the final plot position of the shape to the point (10, 25).

```

0 X0 = Y0 = COLR = SHAPE = ROT = SCALE : REM SET PARAMETERS
5 INIT = -12288 : LINE = -11500 : DRAW = -11402 : FIND = -11780
10 VIOLET = 85 : WHITE = 127 : REM DEFINE SUBROUTINES AND COLORS
20 X0 = 140 : Y0 = 80 : COLR = WHITE : REM ASSIGN PARAMETER VALUES
30 SHAPE = 3 : ROT = 0 : SCALE = 2
40 CALL DRAW : REM DRAW SHAPE WITH ABOVE PARAMETERS
50 CALL FIND : REM FIND COORDINATES OF LAST SHAPE POINT
60 X0 = 10 : Y0 = 25 : COLR = VIOLET : REM NEW PARAMETER VALUES, FOR LINE
70 CALL LINE : REM DRAW LINE WITH ABOVE PARAMETERS
80 END

```

## **COLLISIONS**

Any time two or more shapes intersect or overlap, the new shape has points in common with the previous shapes. These common points are called points of "collision."

The **DRAW** and **DRAW1** subroutines return a "collision count" in the hexadecimal memory location \$32A (810, in decimal). The collision count will be constant for a fixed shape, rotation, scale, and background, provided that no collisions with other shapes are detected. The difference between the "standard" collision value and the value encountered while **DRAWing** a shape is a true collision counter. For example, the collision counter is useful for determining whether or not two constantly moving shapes ever touch each other.

```

110 CALL DRAW : REM DRAW THE SHAPE
120 COUNT = PEEK(810) : REM FIND THE COLLISION COUNT

```

## PART F: TECHNICAL INFORMATION

### LOCATIONS OF THE HIGH-RESOLUTION PARAMETERS

When the high-resolution parameters are entered (line 0, say), they are stored -- with space for their values -- in the BASIC variable table, just above LOMEM (the LOWest MEMory location used for BASIC variable storage). These parameters appear in the variable table in the exact order of their first mention in the BASIC program. That order must be as shown below, because the High-Resolution subroutines look for the parameter values by location only. Each parameter value is two bytes in length. The low-order byte is stored in the lesser of the two locations assigned.

### VARIABLE-TABLE PARAMETER LOCATIONS

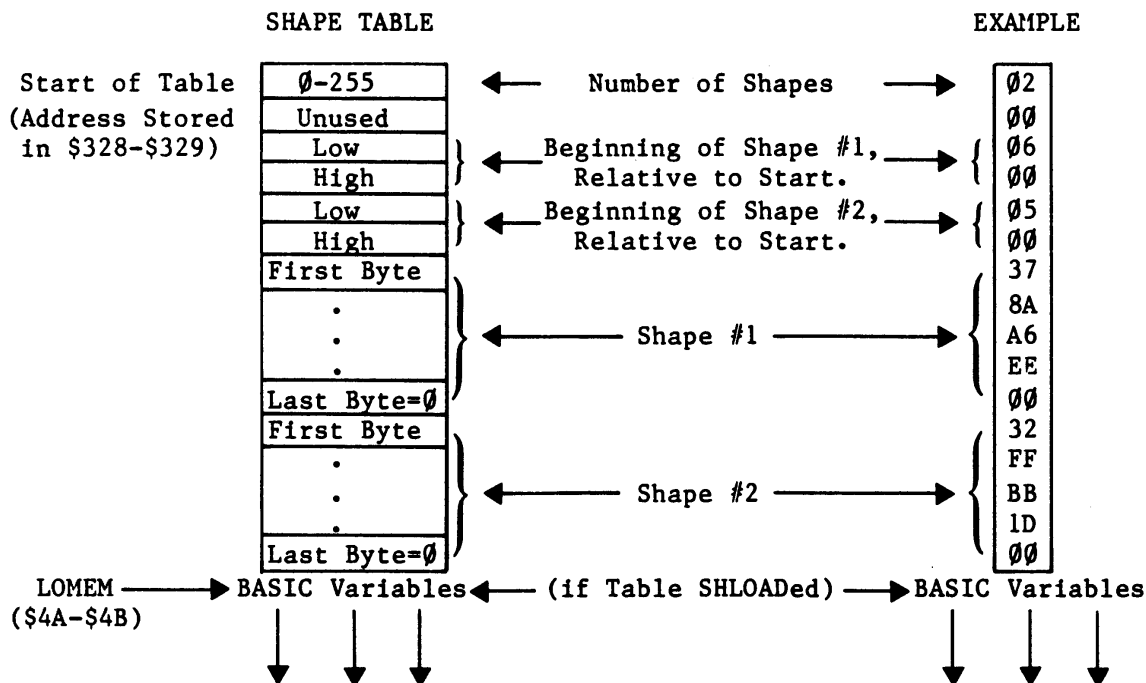
<u>Parameter</u>	<u>Locations beyond LOMEM</u>
X0	\$05, \$06
Y0	\$0C, \$0D
COLR	\$15, \$16
SHAPE	\$1F, \$20
ROT	\$27, \$28
SCALE	\$31, \$32

## VARIABLES USED WITHIN THE HIGH-RESOLUTION SUBROUTINES

<u>Variable Name</u>	<u>Hexadecimal Location</u>	<u>Description</u>
SHAPEL, SHAPEH	1A, 1B	On-the-fly shape pointer.
HCOLOR1	1C	On-the-fly color byte.
COUNTH	1D	High-order byte of step count for LINE.
HBASL, HBASH	26, 27	On-the-fly BASE ADDRESS
HMASK	30	On-the-fly BIT MASK
QDRNT	53	2 LSB's are rotation quadrant for DRAW.
XOL, XOH	320, 321	Most recent X-coordinate. Used for initial endpoint of LINE. Updated by PLOT, POSN, LINE and FIND, not DRAW.
YO	322	Most recent Y-coordinate (see XOL, XOH).
BXSAV	323	Saves 6502 X-register during high-resolution CALLs from BASIC.
HCOLOR	324	Color specification for PLOT, POSN.
HNDX	325	On-the-fly byte index from BASE ADDRESS.
HPAG	326	Memory page for plotting graphics. Normally \$20 for plotting in Page 1 of high-resolution display memory (\$2000-\$3FFF).
SCALE	327	On-the-fly scale factor for DRAW.
SHAPXL, SHAPXH	328, 329	Start of Shape Table pointer.
COLLSN	32A	Collision count from DRAW, DRAW1.

## SHAPE TABLE INFORMATION

<u>Shape Tape</u>	<u>Description</u>
Record #1	A two-byte-long record that contains the length of record #2, Low-order first.
Record Gap	Minumum of .7 seconds in length.
Record #2	The Shape Table (see below).



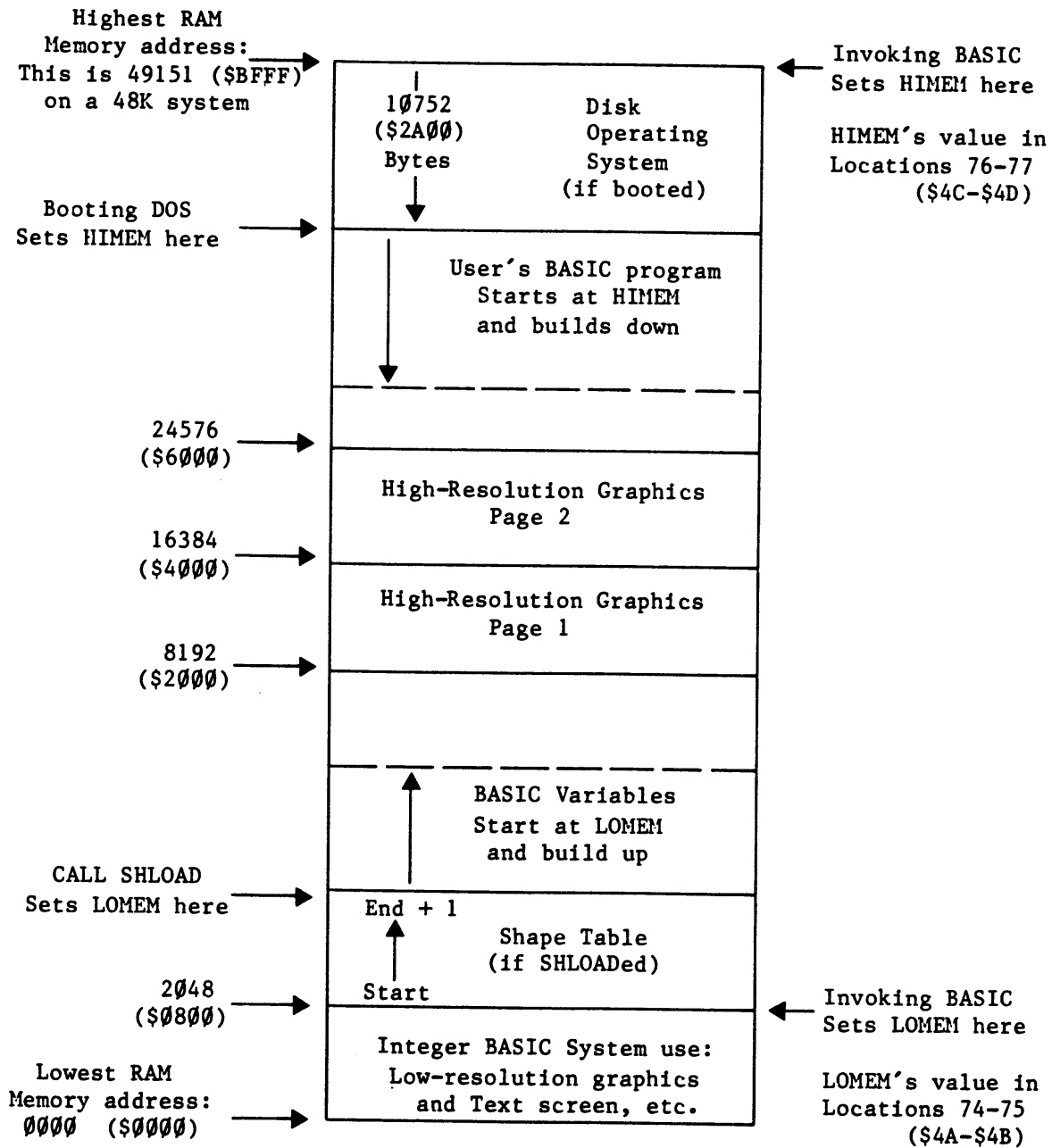
The address of the Shape Table's Start should be stored in locations \$328 and \$329. If the SHLOAD subroutine is used to load the table, Start will be set to LOMEM (normally this is at \$0800) and then LOMEM will be moved to one byte after the end of the Shape Table, automatically.

If you wish to load a Shape Table named MYSHAPES2 from disk, beginning at decimal location 2048 (0800 hex), and ending at decimal location 2048 plus decimal 15 bytes (as in the example above), you may wish to begin your BASIC program as follows:

```

0 D$ = "" : REM QUOTES CONTAIN CTRL D (D$ WILL BE ERASED BY SHAPE TABLE)
1 PRINT D$; "BLOAD MYSHAPES2 , A 2048" : REM LOADS SHAPE TABLE
2 POKE 808, 2048 MOD 256 : POKE 809, 2048 / 256 : REM SETS TABLE START
3 POKE 74, (2048 + 15 + 1) MOD 256 : POKE 75, (2048 + 15 + 1) / 256
4 POKE 204, PEEK(74) : POKE 205, PEEK(75) : REM SETS LOMEM TO TABLE END+1
5 X0 = Y0 = COLR = SHAPE = ROT = SCALE : REM SETS PARAMETERS
    
```

# APPLE II MEMORY MAP FOR USING HIGH-RESOLUTION GRAPHICS WITH INTEGER BASIC



Unfortunately, there is no convention for mapping memory. This map shows the highest (largest) address at the top, lowest (smallest) address at the bottom. The maps of Shape Tables that appear on other pages show the Starting address (lowest and smallest) at the top, the Ending address (highest and largest) at the bottom.



## PART G: COMMENTS

1. Using memory Page 1 for high-resolution graphics erases everything in memory from location 8192 (\$2000 hex) to location 16383 (\$3FFF). If the top of your system's memory is in this range (as it will be, if you have a 16K system), Integer BASIC will normally put your BASIC program exactly where it will be erased by INIT. You must protect your program by setting HIMEM below memory Page 1, after invoking BASIC (reset ctrl B return) and before RUNNING your program: use this immediate-execution command:

```
HIMEM: 8192 return
```

2. Using memory Page 2 for high-resolution graphics erases memory from location 16384 (\$4000) to location 24575 (\$5FFF). If yours is a 24K system, this will erase your BASIC program unless you do one of the following:

- a) never use Page 2 for graphics; or
- b) change HIMEM to 8192, as described above.

3. The picture is further confused if you are also using an APPLE disk with your system. The Disk Operating System (DOS), when booted, occupies the highest 10.5K (\$2A00) bytes of memory. HIMEM is moved to just below the DOS. Therefore, if your system contains less than 32K of memory, the DOS will occupy memory Page 1 and Page 2. In that case, you cannot use the High-Resolution graphics with the DOS intact. An attempt to do so will erase all or part of the DOS. A 32K system can use only Page 1 for graphics without destroying the DOS, but HIMEM must be moved to location 8192 as described above. 48K systems can usually use the DOS and both high-resolution memory pages without problems.

4. If you loaded your Shape Table starting at LOMEM in location 2048 (\$0800), from disk or from tape without using SHLOAD, Integer BASIC will erase the Shape Table when it stores the program variables. To protect your Shape Table, you must move LOMEM to one byte beyond the last byte of the Shape Table, after invoking BASIC and before using any variables. SHLOAD does this automatically, but you can use this immediate-execution command:

```
LOMEM: 2048 + tablelength + 1
```

where tablelength must be a number, not a variable name. Some programmers load their Shape Tables beginning in location 3048 (\$0BE8). That leaves a safe margin of 1000 bytes for variables below the Shape Table, and at least 5000 bytes (if HIMEM:8192) above the table for their BASIC program.

5. CALLing an undefined or accidentally misspelled variable name is usually a CALL to location zero (the default value of any undefined variable). This CALL may cause unpredictable and unwelcome results, depending on the contents of location zero. However, after you execute this BASIC command:

```
POKE 0, 96
```

an accidental CALL to location zero will cause a simple jump back to your BASIC program, with no damage.

# APPENDIX

# SOURCE ASSEMBLY LISTINGS

66	High-Resolution Graphics	\$D000-\$D3FF
76	Renumber	\$D400-\$D4BB
79	Append	\$D4BC-\$D4D4
80	Relocate	\$D4DC-\$D52D
82	Tape Verify (BASIC)	\$D535-\$D553
83	Tape Verify (6502 Code & Data)	\$D554-\$D5AA
84	RAM Test	\$D5BC-\$D691
87	Music	\$D717-\$D7F8

```

1 *****
2 *
3 * APPLE-II HI-RESOLUTION *
4 * GRAPHICS SUBROUTINES *
5 *
6 * BY WOZ 9/13/77 *
7 *
8 * ALL RIGHTS RESERVED *
9 *
10 *****
    
```

```

12 * HI-RES EQUATES
13 SHAPEL EQU $1A POINTER TO
14 SHAPEH EQU $1B SHAPE LIST
15 HCOLOR1 EQU $1C RUNNING COLOR MASK
16 COUNTH EQU $1D
17 HBASL EQU $26 BASE ADR FOR CURRENT
18 HBASH EQU $27 HI-RES PLOT LINE. A
19 HMASK EQU $30
20 A1L EQU $3C MONITOR A1.
21 A1H EQU $3D
22 A2L EQU $3E MONITOR A2.
23 A2H EQU $3F
24 LOMEML EQU $4A BASIC 'START OF VARS'.
25 LOMEMH EQU $4B
26 DXL EQU $50 DELTA-X FOR HLIN, SHAPE.
27 DXH EQU $51
28 SHAPEX EQU $51 SHAPE TEMP.
29 DY EQU $52 DELTA-Y FOR HLIN, SHAPE.
30 GDRNT EQU $53 ROT QUADRANT (SHAPE).
31 EL EQU $54 ERROR FOR HLIN.
32 EH EQU $55
33 PPL EQU $CA BASIC START OF PROG PTR.
34 PPH EQU $CB
35 PVL EQU $CC BASIC END OF VARS PTR.
36 PVH EQU $CD
37 ACL EQU $CE BASIC ACC.
38 ACH EQU $CF
39 XOL EQU $320 PRIOR X-COORD SAVE
40 XOH EQU $321 AFTER HLIN OR HPLOT.
41 YO EQU $322 HLIN, HPLOT Y-COORD SAVE.
42 BXSAY EQU $323 X-REQ SAVE FOR BASIC.
43 HCOLOR EQU $324 COLOR FOR HPLOT, HPOSN
44 HNDX EQU $325 HORIZ OFFSET SAVE.
45 HPAQ EQU $326 HI-RES PAGE ($20 NORMAL)
46 SCALE EQU $327 SCALE FOR SHAPE, MOVE.
47 SHAPXL EQU $328 START OF
48 SHAPXH EQU $329 SHAPE TABLE.
49 COLLSN EQU $32A COLLISION COUNT.
50 HIRES EQU $C057 SWITCH TO HI-RES VIDEO
51 MIXSET EQU $C053 SELECT TEXT/GRAPHICS MIX
52 TXTCLR EQU $C050 SELECT GRAPHICS MODE.
53 MEMFUL EQU $E36B BASIC MEM FULL ERROR.
54 RNGERR EQU $E46B BASIC RANGE ERROR.
55 ACADR EQU $F11E 2-BYTE TAPE READ SETUP.
56 RD2BIT EQU $FCFA TWO-EDGE TAPE SENSE.
57 READ EQU $FEFD TAPE READ (A1, A2).
58 READX1 EQU $FF02 READ WITHOUT HEADER.
    
```

```

60 * HIGH RESOLUTION GRAPHICS INITS
    
```

```

61 *
62 * ROM VERSION $D000 TO $D3FF
63 *
64 ORG $D000
65 OBJ $A000
66 SETHRL LDA $20 INIT FOR $2000-3FFF
67 STA HPAQ HI-RES SCREEN MEMORY.
    
```

```

D000 A9 20
D002 8D 26 03
    
```

```

D005 AD 57 C0      68      LDA      HIRES SET HIRES DISPLAY MODE
D008 AD 53 C0      69      LDA      MIXSET WITH TEXT AT BOTTOM.
D00B AD 50 C0      70      LDA      TXTCLR SET GRAPHICS DISPLAY MODE
D00E A9 00         71 HCLR   LDA      #*0
D010 85 1C         72 BKGND0 STA     HCOLOR1 SET FOR BLACK BKGND.
D012 AD 26 03      73 BKGND LDA      HPAG
D015 85 18         74      STA      SHAPEH INIT HI-RES SCREEN MEM
D017 A0 00         75      LDY      #*0 FOR CURRENT PAGE, NORMALLY
D019 84 1A         76      STY      SHAPEI. #2000-3FFF OR #4000-5FFF
D01B A5 1C         77 BKGND1 LDA     HCOLOR1
D01D 91 1A         78      STA      (SHAPEI), Y
D01F 20 A2 D0      79      JSR      CSHFT2 (SHAPEI, H) WILL SPECIFY
D022 C8            80      INY      32 SEPARATE PAGES.
D023 D0 F6         81      BNF      BKGND1 THROUGHOUT THE INIT.
D025 E6 18         82      INC      SHAPEH
D027 A5 18         83      LDA      SHAPEH
D029 29 1F         84      AND      #*1F TEST FOR DONE.
D02B D0 EE         85      BNF      BKGND1
D02D 60            86      RTS

88 * HI-RES GRAPHICS POSITION AND PLOT SUBRS
D02E 8D 22 03      89 HPOSN STA     YO ENTER WITH Y IN A-REG,
D031 8E 20 03      90      STX     XOL XL IN X-REG,
D034 8C 21 03      91      STY     XOH AND XH IN Y-REG.
D037 48            92      PHA
D038 29 C0         93      AND     #*C0
D03A 85 26         94      STA     HBASL FOR Y-COORD = 00ABCDEF.
D03C 4A            95      LSR     ; CALCULATES BASE ADDRESS
D03D 4A            96      LSR     ; IN HBASL. HBASH FOR
D03E 05 26         97      ORA     HBASL ACCESSING SCREEN MEM
D040 85 26         98      STA     HBASL VIA (HBASL), Y ADDRESSING MODE
D042 68            99      PLA
D043 85 27         100     STA     HBASH
D045 0A            101     ASL     ; CALCULATES
D046 0A            102     ASL     ; HRASH = PPPFQHCD,
D047 0A            103     ASL     ; HRASL = EABAB000
D048 26 27         104     ROL     HBASH
D04A 0A            105     ASL     ; WHERE PPP=001 FOR #2000-3FFF
D04B 26 27         106     ROL     HBASH SCREEN MEM RANGE AND
D04D 0A            107     ASL     ; PPP=010 FOR #4000-7FFF
D04E 66 26         108     ROR     HBASL (GIVEN Y-COORD=ABCDEFQH)
D050 A5 27         109     LDA     HBASH
D052 29 1F         110     AND     #*1F
D054 0D 26 03      111     ORA     HPAG
D057 85 27         112     STA     HBASH
D059 8A            113     TXA     DIVIDE X0 BY 7 FOR
D05A C0 00         114     CPY     #*0 INDEX FROM BASE ADR
D05C F0 05         115     BEQ     HPOSN2 (QUOTIENT) AND BIT
D05E A0 23         116     LDY     #*23 WITHIN SCREEN MEM BYTE
D060 69 04         117     ADC     #*4 (MASK SPEC'D BY REMAINDER)
D062 C8            118 HPOSN1 INY
D063 E9 07         119 HPOSN2 SBC     #*7 SUBTRACT OUT SEVENS.
D065 B0 FB         120     BCS     HPOSN1
D067 8C 25 03      121     STY     ;INDX WORKS FOR X0 FROM
D06A AA            122     TAX     0 TO 279, LOW-ORDER
D06B 8D EA D0      123     LDA     MSKTBL-249, X BYTE IN X-REG,
D06E 85 30         124     STA     HMASK HIGH IN Y-REG ON ENTRY
D070 98            125     TYA
D071 4A            126     LSR     ; IF ON ODD BYTE (CARRY SET)
D072 AD 24 03      127     LDA     HCOLOR THEN ROTATE HCOLOR ONE
D075 85 1C         128 HPOSN3 STA     HCOLOR1 BIT FOR 180 DEGREE SHIFT
D077 B0 29         129     BCS     CSHFT2 PRIOR TO COPYING TO HCOLOR1.
D079 60            130     RTS
D07A 20 2E D0      131 HPLOT JSR     HPOSN
D07D A5 1C         132 HPLOT1 LDA     HCOLOR1 CALC BIT POSN IN HBASL, H
D07F 51 26         133     EOR     (HBASL), Y HNDX, AND HMASK FROM
D081 25 30         134     AND     HMASK Y-COORD IN A-REG,
D083 51 26         135     EOR     (HBASL), Y X-COORD IN X, Y-REGS.
D085 91 26         136     STA     (HBASL), Y FOR ANY 'L' BITS OF HMASK
D087 60            137     RTS     SUBSTITUTE CORRESPONDING
138 *              138     BIT OF HCOLOR1.

```

```

140 * HI-RES GRAPHICS L,R,U,D SUBRS
D038 10 24 141 LFTRT BPL RIGHT USE SIGN FOR LFT/RT SELECT
D03A A5 30 142 LEFT LDA HMASK
D03C 4A 143 LSR ; SHIFT LOW-ORDER
D03D 80 05 144 BCS LEFT1 7 BITS OF HMASK
D03F 49 C0 145 EOR **C0 ONE BIT TO LSB.
D071 85 30 146 LR1 STA HMASK
D073 60 147 RTS
D074 88 148 LEFT1 DEY DECR HORIZ INDEX.
D075 10 02 149 BPL LEFT2
D077 A0 27 150 LDY **27 WRAP AROUND SCREEN.
D079 A9 C0 151 LEFT2 LDA **C0 NEW HMASK, RIGHTMOST
D07B 85 30 152 NEWNDX STA HMASK DOT OF BYTE.
D07D 8C 25 03 153 STY HNDX UPDATE HORIZ INDEX.
D0A0 A5 1C 154 CSHIFT LDA HCOLOR1
D0A2 0A 155 CSHFT2 ASL ; ROTATE LOW-ORDER
D0A3 C9 C0 156 CMP **C0 7 BITS OF HCOLOR1
D0A5 10 06 157 BPL RTS1 ONE BIT POSN.
D0A7 A5 1C 158 LDA HCOLOR1
D0A9 49 7F 159 EOR **7F ZXYXYXYX -> ZYXYXYXY
D0AB 85 1C 160 STA HCOLOR1
D0AD 60 161 RTS1 RTS
D0AE A5 30 162 RIGHT LDA HMASK
D0B0 0A 163 ASL ; SHIFT LOW-ORDER
D0B1 49 80 164 EOR **80 7 BITS OF HMASK
D0B3 30 DC 165 BMI LR1 ONE BIT TO MSB.
D0B5 A9 81 166 LDA **81
D0B7 C8 167 INY NEXT BYTE.
D0B8 C0 28 168 CPY **28
D0BA 90 DF 169 BCC NEWNDX
D0BC A0 00 170 LDY **0 WRAP AROUND SCREEN IF >279
D0BE 80 DB 171 BCS NEWNDX ALWAYS TAKEN.

173 * L,R,U,D, SUBROUTINES.
DOC0 18 174 LRUDX1 CLC NO 90 DEG ROT (X-OR).
DOC1 A5 51 175 LRUDX2 LDA SHAPEX
DOC3 29 04 176 AND **4 IF B2=0 THEN NO PLOT.
DOC5 F0 27 177 BEQ LRUD4
DOC7 A9 7F 178 LDA **7F FOR EX-OR IN'D SCREEN MEM
DOC9 25 30 179 AND HMASK
DOCB 31 26 180 AND (HBASL),Y SCREEN BIT SET?
D0CD D0 18 181 BNE LRUD3
D0CF EE 2A 03 182 INC COLLSN
D0D2 A9 7F 183 LDA **7F
D0D4 25 30 184 AND HMASK
D0D6 10 12 185 BPL LRUD3 ALWAYS TAKEN.
D0D8 18 186 LRUD1 CLC NO 90 DEG ROT.
D0D9 A5 51 187 LRUD2 LDA SHAPEX
D0DB 29 04 188 AND **4 IF B2=0 THEN NO PLOT.
D0DD F0 0F 189 BEQ LRUD4
D0DF B1 26 190 LDA (HBASL),Y
D0E1 45 1C 191 EOR HCOLOR1 SET HI-RES SCREEN BIT
D0E3 25 30 192 AND HMASK TO CORRESPONDING HCOLOR1
D0E5 D0 03 193 BNE LRUD3 IF BIT OF SCREEN CHANGES
D0E7 EE 2A 03 194 INC COLLSN THEN INCR COLLSN DETECT
D0EA 51 26 195 LRUD3 EOR (HBASL),Y
D0EC 91 26 196 STA (HBASL),Y
D0EE A5 51 197 LRUD4 LDA SHAPEX ADD GDRNT TO
D0F0 65 53 198 ADC QDRNT SPECIFIED VECTOR
D0F2 29 03 199 AND **3 AND MOVE LFT, RT,
200 EQ3 EQU *-1 UP, OR DWN BASED
D0F4 C9 02 201 CMP **2 ON SIGN AND CARRY.
D0F6 6A 202 ROR
D0F7 B0 8F 203 LRUD BCS LFTRT
D0F9 30 30 204 UPDWN BMI DOWN4 SIGN FOR UP/DWN SELECT
D0FB 18 205 UP CLC
D0FC A5 27 206 LDA HBASH CALC BASE ADDRESS
D0FE 2C EA D1 207 BII EG1C (ADR OF LEFTMOST BYTE)
D101 D0 22 208 BNE UP4 FOR NEXT LINE UP
D103 06 26 209 ASL HBASL IN (HBASL, HBASH)

```

```

D105 B0 1A      210      BCS    UP2 WITH 192-LINE WRAPAROUND
D107 2C F3 D0  211      BIT    EQ3
D10A F0 05      212      BEQ    UP1
D10C 69 1F      213      ADC    #$1F **** BIT MAP ****
D10E 38         214      SEC
D10F B0 12      215      BCS    UP3 FOR ROW = ABCDEFGH,
D111 69 23      216 UP1    ADC    ##23
D113 48         217      PHA
D114 A5 26      218      LDA    HBASL HBASL = EABAB000
D116 69 B0      219      ADC    ##B0 HBASH = PPPFGHCD
D118 B0 02      220      BCS    UP5
D11A 69 F0      221      ADC    ##F0 WHERE PPP=001 FOR PRIMARY
D11C 85 26      222 UP5    STA    HBASL HI-RES PAGE ($2000-$3FFF)
D11E 68         223      PLA
D11F B0 02      224      BCS    UP3
D121 69 1F      225 UP2    ADC    #$1F
D123 66 26      226 UP3    ROR    HBASL
D125 69 FC      227 UP4    ADC    ##FC
D127 85 27      228 UPDOWN1 STA    HBASH
D129 60         229      RTS
D12A 18         230 DOWN   CLC
D12B A5 27      231 DOWN4  LDA    HBASH
D12D 69 04      232      ADC    ##4 CALC BASE ADR FOR NEXT LINE
D12F 2C EA D1   233 EQ4    EQU    *-1 DOWN TO (HBASL, HBASH)
D132 D0 F3      234      BIT    EQ1C
D132 D0 F3      235      BNE    UPDOWN1

D134 06 26      236      ABL    HBASL WITH 192-LINE WRAPAROUND
D136 90 19      237      BCC    DOWN1
D138 69 E0      238      ADC    ##E0
D13A 18         239      CLC
D13B 2C 2E D1   240      BIT    EQ4
D13E F0 13      241      BEQ    DOWN2
D140 A5 26      242      LDA    HBASL
D142 69 50      243      ADC    ##50
D144 49 F0      244      EOR    ##F0
D146 F0 02      245      BEQ    DOWN3
D148 49 F0      246      EOR    ##F0
D14A 85 26      247 DOWN3  STA    HBASL
D14C AD 26 03   248      LDA    HPAQ
D14F 90 02      249      BCC    DOWN2
D151 69 E0      250 DOWN1  ADC    ##E0
D153 66 26      251 DOWN2  ROR    HBASL
D155 90 D0      252      BCC    UPDOWN1

254 * HI-RES GRAPHICS LINE DRAW SUBRS
D157 48         255 HLINRL PHA
D158 A9 00      256      LDA    ##0 SET (XOL, XOH) AND
D15A 8D 20 03   257      STA    XOL YO TO ZERO FOR
D15D 8D 21 03   258      STA    XOH REL LINE DRAW
D160 8D 22 03   259      STA    YO (DX, DY).
D163 68         260      PLA
D164 48         261 HLIN   PHA    ON ENTRY
D165 38         262      SEC    XL: A-REG
D166 ED 20 03   263      SBC    XOL XH; X-REG
D169 48         264      PHA    Y: Y-REG
D16A 8A         265      TXA
D16B ED 21 03   266      SBC    XOH
D16E 85 53      267      STA    QDRNT CALC ABS(X-XO)
D170 B0 0A      268      BCS    HLIN2 IN (DXL, DXH).

```

D172 68	269	PLA	
D173 49 FF	270	EOR	##FF X DIR TO SIGN BIT
D175 69 01	271	ADC	##1 OF GDRNT.
D177 48	272	PHA	0=RIGHT (DX POS)
D178 A9 00	273	LDA	##0 1=LEFT (DX NEG)
D17A E5 53	274	SBC	QDRNT
D17C 85 51	275 HLIN2	STA	DXH
D17E 85 55	276	STA	EH INIT (EL,EH) TO
D180 68	277	PLA	ARS(X-XO)
D181 85 50	278	STA	DXL
D183 85 54	279	STA	EL
D185 68	280	PLA	
D186 8D 20 03	281	STA	XOL
D189 8E 21 03	282	STX	XOH
D18C 98	283	TYA	
D18D 18	284	CLC	
D18E ED 22 03	285	SBC	YO CALC -ABS(Y-0)-1
D191 90 04	286	BCC	HLIN3 IN DY.
D193 49 FF	287	EOR	##FF
D195 69 FE	288	ADC	##FE
D197 85 52	289 HLIN3	STA	DY ROTATE Y DIR INTO
D199 8C 22 03	290	STY	YO GDRNT SIGN BIT
D19C 66 53	291	ROR	QDRNT (0=UP, 1=DOWN)
D19E 38	292	SEC	
D19F E5 50	293	SBC	DXL INIT (COUNTL, COUNTH).
D1A1 AA	294	TAX	TO -(DELTX+DELTY+1)
D1A2 A9 FF	295	LDA	##FF
D1A4 E5 51	296	SBC	DXH
D1A6 85 1D	297	STA	COUNTH
D1A8 AC 25 03	298	LDY	HNDX HORIZ INDEX
D1AB 80 05	299	BCS	MOVEX2 ALWAYS TAKEN.
D1AD 0A	300 MOVEX	ASL	; MOVE IN X-DIR. USE
D1AE 20 88 D0	301	JSR	LFTRT GDRNT B6 FOR LFT/RT SELECT
D1B1 38	302	SEC	
D1B2 A5 54	303 MOVEX2	LDA	EL ASSUME CARRY SET.
D1B4 65 52	304	ADC	DY (EL,EH)-DELTY TO (EL,EH)
D1B6 85 54	305	STA	EL NOTE: DY IS (-DELTY)-1
D1B8 A5 55	306	LDA	EH CARRY CLR IF (EL,EH)
D1BA E9 00	307	SBC	##0 GOES NEG.
D1BC 85 55	308 HCOUNT	STA	EH
D1BE 81 26	309	LDA	(HRASL),Y SCREEN BYTE.
D1C0 45 1C	310	EOR	HCOLOR1 PLOT DOT OF HCOLOR1.
D1C2 25 30	311	AND	HMASK CURRENT BIT MASK.
D1C4 51 26	312	EOR	(HRASL),Y
D1C6 91 26	313	STA	(HBSL),Y
D1C8 E8	314	INX	DONE (DEI.TX+DELTY)
D1C9 D0 04	315	BNH	HLIN4 DOTS?
D1CB E6 1D	316	INC	COUNTH
D1CD F0 68	317	BEG	RTS2 YES. RETURN.
D1CF A5 53	318 HLIN4	LDA	QDRNT FOR DIRECTION TEST
D1D1 80 DA	319	BCS	MOVEX IF CAR SET, (EL, EH) POS
D1D3 20 F9 D0	320	JSR	UPDWN IF CLR, NEG, MOVE YDIR
D1D6 18	321	CLC	
D1D7 A5 54	322	LDA	EL (EL,EH)+DELTX
D1D9 65 50	323	ADC	DXL TO (EL,EH).
D1DB 85 54	324	STA	EL
D1DD A5 55	325	LDA	EH CAR SET IF (EL,EH) GOES POS
D1DF 65 51	326	ADC	DXH
D1E1 50 D9	327	BVC	HCOUNT ALWAYS TAKEN.
D1E3 81	328 MSKTBL	HEX	81 LEFTMOST BIT OF BYTE.
D1E4 82 84 88	329	HEX	82, 84, 88
D1E7 90 A0	330	HEX	90, A0
D1E9 C0	331	HEX	C0 RIGHTMOST BIT OF BYTE.
D1EA 1C	332 EQ1C	HEX	1C
D1EB FF FE FA	333 COS	HEX	FF, FE, FA, F4, EC, E1, D4, C5, B4
D1F4 A1 8D 78	334	HEX	A1, 8D, 78, 61, 49, 31, 18, FF

```

336 * HI-RES GRAPHICS COORDINATE RESTORE SUBR
D1FC A5 26 337 HFIND LDA HBASL
D1FE 0A 338 ASL ; CONVERTS BASE ADR
D1FF A5 27 339 LDA HBASH TO Y-COORD.
D201 29 03 340 AND ##3
D203 2A 341 ROL ; FOR HBASL = EABAB00D
D204 05 26 342 ORA HBASL HBASH = PPPFGHCD
D206 0A 343 ASL
D207 0A 344 ASL ; GENERATE
D208 0A 345 ASL ; Y-COORD = ABCDEFGH
D209 8D 22 03 346 STA YO
D20C A5 27 347 LDA HBASH (PPP=SCREEN PAGE,
D20E 4A 348 LSR ; NORMALLY 001 FOR
D20F 4A 349 LSR ; $2000-$3FFF
D210 29 07 350 AND ##7 HI-RES SCREEN)
D212 0D 22 03 351 ORA YO
D215 8D 22 03 352 STA YO CONVERTS HNDX (INDEX
D218 AD 25 03 353 LDA HNDX FROM BASE ADR)
D21B 0A 354 ASL ; AND HMASK (BIT
D21C 6D 25 03 355 ADC HNDX MASK) TO X-COORD
D21F 0A 356 ASL ; IN (XOL,XOH)
D220 AA 357 TAX (RANGE $0-$133)
D221 CA 358 DEX
D222 A5 30 359 LDA HMASK
D224 29 7F 360 AND ##7F
D226 EB 361 HFIND1 INX
D227 4A 362 LSR
D228 D0 FC 363 DNE HFIND1
D22A 8D 21 03 364 STA XOH
D22D 8A 365 TXA
D22E 18 366 CLC CALC HNDX*7 +
D22F 6D 25 03 367 ADC HNDX LOG (BASE 2) HMASK.
D232 90 03 368 BCC HFIND2
D234 EE 21 03 369 INC XOH
D237 8D 20 03 370 HFIND2 STA XOL
D23A 60 371 RTS2 RTB

```

```

373 * HI-RES GRAPHICS SHAPE DRAW SUBR
374 *
375 * SHAPE DRAW
376 * R = 0 TO 63
377 * SCALE FACTOR USED (1=NORMAL)
378 *
D23B 86 1A 379 DRAW STX SHAPE1. DRAW DEFINITION
D23D 84 1B 380 STY SHAPEH POINTER.
D23F AA 381 DRAW1 TAX
D240 4A 382 LSR ; ROT ($0-$3F)
D241 4A 383 LSR
D242 4A 384 LSR ; QDRNT 0=UP, 1=RT,
D243 4A 385 LSR ; 2=DWN, 3=LFT.
D244 85 53 386 STA QDRNT
D246 8A 387 TXA
D247 29 0F 388 AND ##F
D249 AA 389 TAX
D24A BC EB D1 390 LDY COS, X SAVE COS AND SIN
D24D 84 50 391 STY DXL VALS IN DXL AND DY.
D24F 49 0F 392 EOR ##F
D251 AA 393 TAX
D252 BC EC D1 394 LDY COS+1, X
D255 CB 395 INY
D256 84 52 396 STY DY
D258 AC 25 03 397 DRAW2 LDY HNDX BYTE INDEX FROM
D25B A2 00 398 LDX ##0 HI-RES BASE ADR.
D25D BE 2A 03 399 STX COLLN CLEAR COLLISION COUNT.
D260 A1 1A 400 LDA (SHAPE1, X) 1ST SHAPE DEF BYTE.

```



```

D262 85 51      401 DRAW3 STA  SHAPEX
D264 A2 80      402      LDX  ##80
D266 86 54      403      STX  EI, EL, EH FOR FRACTIONAL
D268 86 55      404      STX  EH L, R, U, D VECTORS.
D26A AE 27 03   405      LDX  SCALE SCALE FACTOR.
D26D A5 54      406 DRAW4 LDA  EL
D26F 38         407      SEC  IF FRAC COS OVFL
D270 65 50      408      ADC  DXL THEN MOVE IN
D272 85 54      409      STA  EI SPECIFIED VECTOR
D274 90 04      410      BCC  DRAW5 DIRECTION.
D276 20 D8 D0   411      JSR  LRUD1
D279 18         412      CLC
D27A A5 55      413 DRAW5 LDA  EH IF FRAC SIN OVFL,
D27C 65 52      414      ADC  DY THEN MOVE IN
D27E 85 55      415      STA  EH SPECIFIED VECTOR
D280 90 03      416      BCC  DRAW6 DIRECTION +90 DEG.
D282 20 D9 D0   417      JSR  LRUD2
D285 CA         418 DRAW6 DEX  LOOP ON SCALE
D286 D0 E5      419      BNE  DRAW4 FACTOR.
D288 A5 51      420      LDA  SHAPEX
D28A 4A         421      LSR  ; NEXT 3-BIT VECTOR
D28B 4A         422      LSR  ; OF SHAPE DEF.
D28C 4A         423      LSR
D28D D0 D3      424      BNE  DRAW3 NOT DONE THIS BYTE.
D28F E6 1A      425      INC  SHAPEL
D291 D0 02      426      BNE  DRAW7 NEXT BYTE OF
D293 E6 1B      427      INC  SHAPEH SHAPE DEFINITION.
D295 A1 1A      428 DRAW7 LDA  (SHAPEL, X)
D297 D0 C9      429      BNE  DRAW3 DONE IF ZERO.
D299 60         430      RTS

432 * HI-RES GRAPHICS SHAPE EX-OR SUBR
433 *
434 * EX-OR SHAPE INTO SCREEN.
435 *
436 * ROT = 0 TO 3 (QUADRANT ONLY)
437 * SCALE IS USED
438 *
D29A 86 1A      439 XDRAW STX  SHAPEL SHAPE DEFINITION
D29C 84 1B      440      STY  SHAPEH POINTER.
D29E AA         441 XDRAW1 TAX
D29F 4A         442      LSR  ; ROT ($0-$3F)
D2A0 4A         443      LSR
D2A1 4A         444      LSR  ; QDRNT 0=UP, 1=RT,
D2A2 4A         445      LSR  ; 2=DWN, 3=LFT.
D2A3 85 53      446      STA  QDRNT
D2A5 8A         447      TXA
D2A6 29 0F      448      AND  ##F
D2A8 AA         449      TAX
D2A9 8C EB D1   450      LDY  COS, X SAVE COS AND SIN
D2AC 84 50      451      STY  DXL VALS IN DXL AND DY,
D2AE 49 0F      452      EOR  ##F
D2B0 AA         453      TAX
D2B1 8C EC D1   454      LDY  COS+1, X
D2B4 C8         455      INY
D2B5 84 52      456      STY  DY
D2B7 AC 25 03   457 XDRAW2 LDY  HNDX INDEX FROM HI-RES
D2BA A2 00      458      LDX  ##0 BASE ADR.
D2BC 8E 2A 03   459      STX  COLLSN CLEAR COLLISION DETECT
D2BF A1 1A      460      LDA  (SHAPEL, X) 1ST SHAPE DEF BYTE.

```

```

D2C1 85 31      461 XDRAW3 STA  SHAPEX
D2C3 A2 80      462      LDX  ##80
D2C5 86 34      463      STX  EI EL,EH FOR FRACTIONAL
D2C7 86 55      464      STX  EH L,R,U,D, VECTORS.
D2C9 AE 27 03   465      LDX  SCALE SCALE FACTOR.
D2CC A5 34      466 XDRAW4 LDA  EI
D2CE 38         467      SEC  IF FRAC COS OVFL
D2CF 65 30      468      ADC  DXL THEN MOVE IN
D2D1 85 54      469      STA  EI SPECIFIED VECTOR
D2D3 90 04      470      BCC  XDRAW5 DIRECTION
D2D5 20 C0 D0   471      JSR  LRUDX1
D2D8 18         472      CLC
D2D9 A5 55      473 XDRAW5 LDA  EH IF FRAC SIN OVFL
D2DB 65 32      474      ADC  DY THEN MOVE IN
D2DD 85 55      475      STA  EH SPECIFIED VECTOR
D2DF 90 03      476      BCC  XDRAW6 DIRECTION +90 DEG.
D2E1 20 D9 D0   477      JSR  LRUD2
D2E4 CA         478 XDRAW6 DEX  LOOP ON SCALE
D2E5 D0 E5      479      BNE  XDRAW4 FACTOR.
D2E7 A5 31      480      LDA  SHAPEX
D2E9 4A         481      LSR  ; NEXT 3-BIT VECTOR
D2EA 4A         482      LSR  ; OF SHAPE DEF.
D2EB 4A         483      LSR
D2EC D0 D3      484      BNE  XDRAW3
D2EE E6 1A      485      INC  SHAPEL
D2F0 D0 02      486      BNE  XDRAW7 NEXT BYTE OF
D2F2 E6 1B      487      INC  SHAPEH SHAPE DEF.
D2F4 A1 1A      488 XDRAW7 LDA  (SHAPEL,X)
D2F6 D0 C9      489      BNE  XDRAW3 DONE IF ZERO.
D2F8 60         490      RTS

492 * ENTRY POINTS FROM APPLE-II BASIC
D2F9 20 90 D3   493 BPOSN JSR  PCOLR POSN CALL, COLR FROM BASIC
D2FC 8D 24 03   494      STA  HCOLOR
D2FF 20 AF D3   495      JSR  GETYO YO FROM BASIC.
D302 48         496      PHA
D303 20 9A D3   497      JSR  GETXO XO FROM BASIC.
D306 68         498      PLA
D307 20 2E D0   499      JSR  HPOSN
D30A AE 23 03   500      LDX  BXSAV
D30D 60         501      RTS
D30E 20 F9 D2   502 BPLOT JSR  BPOSN PLOT CALL (BASIC).
D311 4C 7D D0   503      JMP  HPLOT1
D314 AD 25 03   504 BLIN1 LDA  HNDX
D317 4A         505      LSR  ; SET HCOLOR1 FROM
D318 20 90 D3   506      JSR  PCOLR BASIC VAR COLR.
D31B 20 75 D0   507      JSR  HPOSN3
D31E 20 9A D3   508 BLINE JSR  GEYXO LINE CALL, GET XO FROM BASIC
D321 8A         509      TXA
D322 48         510      PHA
D323 98         511      TYA
D324 AA         512      TAX
D325 20 AF D3   513      JSR  GETYO YO FROM BASIC
D328 AB         514      TAY
D329 68         515      PLA
D32A 20 64 D1   516      JSR  HLIN
D32D AE 23 03   517      LDX  BXSAV
D330 60         518      RTS
D331 20 90 D3   519 BOND JSR  PCOLR BACKGROUND CALL
D334 4C 10 D0   520      JMP  BKONDO

```

```

522 * DRAW ROUTINES
D337 20 F9 D2 523 BDRAW1 JSR BPOSN
D33A 20 51 D3 524 BDRAW JSR BDRAWX DRAW CALL FROM BASIC.
D33D 20 3B D2 525 JSR DRAW
D340 AE 23 03 526 LDX BXSAB
D343 60 527 RTS
D344 20 F9 D2 528 BXDRW1 JSR BPOSN
D347 20 51 D3 529 BXDRAW JSR BDRAWX EX-OR DRAW
D34A 20 9A D2 530 JSR XDRAW FROM BASIC.
D34D AE 23 03 531 LDX BXSAB
D350 60 532 RTS
D351 8E 23 03 533 BDRAWX STX BXSAB SAVE FOR BASIC.
D354 A0 32 534 LDY ##32
D356 20 92 D3 535 JSR PBYTE SCALE FROM BASIC.
D359 8D 27 03 536 STA SCALE
D35C A0 28 537 LDY ##28
D35E 20 92 D3 538 JSR PBYTE ROT FROM BASIC.
D361 48 539 PHA SAVE ON STACK.
D362 AD 28 03 540 LDA SHAPXL
D365 85 1A 541 STA SHAPEL START OF
D367 AD 29 03 542 LDA SHAPXH SHAPE TABLE.
D36A 85 1B 543 STA SHAPXH
D36C A0 20 544 LDY ##20
D36E 20 92 D3 545 JSR PBYTE SHAPE FROM BASIC.
D371 F0 39 546 BEQ RERR1
D373 A2 00 547 LDX ##0
D375 C1 1A 548 CMP (SHAPEL, X) > NUM OF SHAPES?
D377 F0 02 549 BEQ BDRWX1
D379 80 31 550 BCS RERR1 YES, RANGE ERR.
D37B 0A 551 BDRWX1 ASL
D37C 90 03 552 BCC BDRWX2
D37E E6 1B 553 INC SHAPEH
D380 18 554 CLC
D381 A8 555 BDRWX2 TAY SHAPE NO. * 2.
D382 B1 1A 556 LDA (SHAPEL), Y
D384 65 1A 557 ADC SHAPEL
D386 AA 558 TAX ADD 2-BYTE INDEX
D387 C8 559 INY TO SHAPE TABLE
D388 B1 1A 560 LDA (SHAPEL), Y START ADR
D38A 6D 29 03 561 ADC SHAPXH (X LOW, Y HI).
D38D A8 562 TAY
D38E 68 563 PLA ROT FROM STACK.
D38F 60 564 RTS

566 * BASIC PARAM FEICH SUBR'S
D390 A0 16 567 PCOLR LDY ##16
D392 B1 4A 568 PBYTE LDA (LOMEML), Y
D394 D0 16 569 BNE RERR1 GET BASIC PARAM.
D396 88 570 DEY (ERR IF >255)
D397 B1 4A 571 LDA (LOMEML), Y
D399 60 572 RTSB RTS
D39A 8E 23 03 573 GETXO STX BXSAB SAVE FOR BASIC.
D39D A0 05 574 LDY ##5
D39F B1 4A 575 LDA (LOMEML), Y XO LOW-ORDER BYTE.
D3A1 AA 576 TAX
D3A2 C8 577 INY
D3A3 B1 4A 578 LDA (LOMEML), Y HI-ORDER BYTE.
D3A5 A8 579 TAY
D3A6 E0 18 580 CPX ##18
D3A8 E9 01 581 SBC ##1 RANGE ERR IF >279
D3AA 90 ED 582 BCC RTSB
D3AC 4C 68 EE 583 RERR1 JMP RNGERR
D3AF A0 0D 584 GETYO LDY ##D OFFSET TO YO FROM LOMEM
D3B1 20 92 D3 585 JSR PBYTE GET BASIC PARAM YO.
D3B4 C9 C0 586 CMP ##C0 (ERR IF >191)
D3B6 80 F4 587 BCS RERR1
D3B8 60 588 RTS

```

```

590 * SHAPE TAPE LOAD SUBROUTINE
D3B9 8E 23 03 591 SHLOAD STX  BXS AV SAVE FOR BASIC.
D3BC 20 1E F1 592          JSR  ACADR READ 2-BYTE LENGTH INTO
D3BF 20 FD FE 593          JSR  READ BASIC ACC
D3C2 A9 00 594          LDA  #00 ; START OF SHAPE TABLE IS #0800
D3C4 85 3C 595          STA  A1L
D3C6 8D 28 03 596          STA  SHAPXL
D3C9 18 597          CLC
D3CA 65 CE 598          ADC  ACL
D3CC A8 599          TAY
D3CD A9 08 600          LDA  #08 ; HIGH BYTE OF SHAPE TABLE POINTER.
D3CF 85 3D 601          STA  A1H
D3D1 8D 29 03 602          STA  SHAPXH
D3D4 65 CF 603          ADC  ACH
D3D6 B0 25 604          BCS  MFULL1 NOT ENOUGH MEMORY.
D3D8 C4 CA 605          CPY  PPL
D3DA 48 606          PHA
D3DB E5 CB 607          SBC  PPH
D3DD 68 608          PLA
D3DE B0 1D 609          BCS  MFULL1
D3E0 84 3E 610          STY  A2L
D3E2 85 3F 611          STA  A2H
D3E4 C8 612          INY
D3E5 D0 02 613          BNE  SH1 OD1
D3E7 69 01 614          ADC  #01
D3E9 84 4A 615 SHLOD1 STY  LOMEML
D3EB 85 4B 616          STA  LOMEMH
D3ED 84 CC 617          STY  PVL
D3EF 85 CD 618          STA  PVH
D3F1 20 FA FC 619          JSR  RD2BIT
D3F4 A9 03 620          LDA  #03 . 5 SECOND HEADER.
D3F6 20 02 FF 621          JSR  READX1
D3F9 AE 23 03 622          LDX  BXS AV
D3FC 60 623          RTS
D3FD 4C 6B E3 624 MFULL1 JMP  MEMFUL

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00

```

1 *****
2 *
3 * APPLE-1C BASIC RENUMBER / APPEND SUBROUTINES *
4 *
5 *          VERSION TWO *
6 *          RENUMBER *
7 *          >CLR *
8 *          >START= *
9 *          >STEP= *
10 *          >CALL --10531 *
11 *
12 *          OPTIONAL *
13 *          >FROM= *
14 *          >TO= *
15 *          >CALL -10521 *
16 *
17 *          USE RENX ENTRY *
18 *          FOR RENUMBER ALL *
19 *
20 *          NOZ APRIL 12, 1978 *
21 *          APPLE COMPUTER INC. *
22 *****

```

```

24 *
25 *
26 *          6502 EQUATES
27 *
28 ROL EQU $0          LOW-ORDER SW16 RO BYTE.
29 ROH EQU $1          HI-ORDER.
30 ONE EQU $01
31 R11L EQU $16        LOW-ORDER SW16 R11 BYTE.
32 R11H EQU $17        HI-ORDER.
33 HIMEM EQU $4C       BASIC HIMEM POINTER.
34 PPL EQU $CA         BASIC PRUG POINTER.
35 PVL EQU $CC         BASIC VAR POINTER.
36 MEMFULL EQU $E36B   BASIC MEM FULL ERROR.
37 PRDEC EQU $E51B     BASIC DECIMAL PRINT SUBR.
38 RANGERR EQU $EE6B   BASIC RANGE ERROR.
39 LOAD EQU $F0DF      BASIC LOAD SUBR.
40 SW16 EQU $F689      SWFET 16 ENTRY.
41 CROUT EQU $FD8E     CAR RET SUBR.
42 COUT EQU $FDED      CHAR OUT SUBR.

```

```

44 *
45 *          SWEET 16 EQUATES
46 *
47 ACC EQU $0          SWEET 16 ACCUMULATOR.
48 NEWLOW EQU $1       NEW INITIAL LNO.
49 NEWINCR EQU $2      NEW LNO INCR.
50 LNLOW EQU $3        LOW LNO OF RENUM RANGE.
51 LNHI EQU $4         HI LNO OF RENUM RANGE.
52 TBLSTRT EQU $5      LNO TABLE START.
53 TBLNDX1 EQU $6      PASS 1 LNO TBL INDEX.
54 TBLIM EQU $7        LNO TABLE LIMIT.
55 SCR8 EQU $8         SCRATCH REG.
56 HMEM EQU $8         HIMEM (END OF PRGM).
57 SCR9 EQU $9         SCRATCH REG.
58 PRGNDX EQU $9       PASS 1 PRUG INDEX.
59 PRGNDX1 EQU $A      ALSO PRUG INDEX.
60 NEWLN EQU $B        NEXT "NEW LNO".
61 NEWLN1 EQU $C       PRIOR "NEW LNO" ASSIGN.
62 TBLND EQU $6        PASS 2 LNO TABLE END.
63 PRGNDX2 EQU $7      PASS 2 PRUG INDEX.
64 CHRO EQU $9         ASCII "O".
65 CHRA EQU $A         ASCII "A".

```

```

66 MODE EQU *C CONST/LNO MODE.
67 TBLNDX2 EQU *B LNO TBL IDX FOR UPDATE.
68 OLDLN EQU *D OLD LNO FOR UPDATE.
69 STRCON EQU *B BASIC STR CON TOKEN.
70 REM EQU *C BASIC REM TOKEN.
71 R13 EQU *D SWEET 16 REG 13 (CPR REG).
72 THEN EQU *D BASIC THEN TOKEN.
73 LIST EQU *D BASIC LIST TOKEN.
74 DEL EQU *D
75 SCRC EQU *C SCRATCH REG FOR APPEND.

77 *
78 * APPLE-11 BASIC RENUMBER SUBROUTINE - PASS 1
79 ORG $D400
80 OBJ $A400
D400 20 89 F6 81 RENX JSR SW16 OPTIONAL RANGE ENTRY.
D403 80 82 SUB ACC
D404 33 83 ST LNLOW SET LNLOW=0, LNHI=0.
D405 34 84 ST LNHI
D406 F4 85 DCR LNHI
D407 00 86 RTN
D408 20 89 F6 87 RENUM JSR SW16
D40B 18 4C 00 88 SET HMEM,HIMEM
D40E 68 89 LDD @HMEM
D40F 38 90 ST HMEM
D410 19 CE 00 91 RNUM3 SET SCR9,PVL+2
D413 C9 92 POPD @SCR9 BASIC VAR PNT TO
D414 35 93 ST TBLSTRT TBLSTRT AND TBLNDX1.
D415 36 94 ST TBLNDX1
D416 21 95 LD NEWLOW COPY NEWLOW (INITIAL)
D417 38 96 ST NEWLN TO NEWLN.
D418 3C 97 ST NEWLN1
D419 C9 98 POPD @SCR9 BASIC PROG PNTR
D41A 37 99 ST TBLIM TO TBLIM AND PRGNDX.
D41B 39 100 ST PRGNDX
D41C 29 101 PASS1 LD PRGNDX
D41D DB 102 CPR HMEM IF PRGNDX >= HMEM
D41E 03 46 103 BC PASS2 THEN DONE PASS 1.
D420 3A 104 ST PRGNDX1
D421 26 105 LD TBLNDX1
D422 E0 106 INR ACC IF < TWO BYTES AVAIL IN
D423 D7 107 CPR TBLIM LNO TABLE THEN RETURN
D424 03 38 108 BC MERR WITH "MEM FULL" MESSAGE.
D426 4A 109 LD @PRGNDX1
D427 A9 110 ADD PRGNDX ADD LENTH BYTE TO PROG INDEX.
D428 39 111 ST PRGNDX
D429 6A 112 LDD @PRGNDX1 LINE NUMBER.
D42A D3 113 CPR LNLOW IF < LNLOW THEN GOTO P1B.
D42B 02 2A 114 BNC P1B
D42D D4 115 CPR LNHI IF > LNHI THEN GOTO P1C.
D42E 02 02 116 BNC P1A
D430 07 30 117 BNZ P1C
D432 76 118 P1A STD @TBLNDX1 ADD TO LNO TABLE.
D433 00 119 RTN
D434 A5 01 120 LDA ROH **** 6502 CODE ****
D436 A6 00 121 LDX ROL
D438 20 1B E5 122 JSR PRDEC PRINT OLD LNO "->" NEW LNO
D43B A9 AD 123 LDA **AD (RO,R11) IN DECIMAL.
D43D 20 ED FD 124 JSR COUT
D440 A9 BE 125 LDA **BE
D442 20 ED FD 126 JSR COUT
D445 A5 17 127 LDA R11H
D447 A6 16 128 LDX R11L
D449 20 1B E5 129 JSR PRDEC
D44C 20 BE FD 130 JSR CROUT
131 *
D44F 20 BC F6 132 JSR SW16+3 **** END 6502 CODE ****

```

```

133 *
D452 2B      134      LD      NEWLN
D453 3C      135      ST      NEWLN1      COPY NEWLN TO NEWLN1 AND INCR
D454 A2      136      ADD     NEWINCR      NEWLN BY NEWINCR.
D455 3B      137      ST      NEWLN
D456 0D      138      HEX     OD          'NUL' (WILL SKIP NEXT INSTRUCTION)
D457 D1      139 P1B    CPR     NEWLOW      IF LOW LNO < NEW LOW THEN RANGE ERR.
D458 02 C2   140      BNC     PASS1
D45A 00      141 RERR    RTN     PRINT "RANGE ERR" MESSAGE AND RETURN.
D45B 4C 6B EE 142      JMP     RANGERR
D45E 00      143 MERR    RTN     PRINT "MEM FULL" MESSAGE AND RETURN.
D45F 4C 6B E3 144      JMP     MEMFULL
D462 EC      145 P1C    INR     NEWLN1      IF HI LNO <= MOST RECENT NEWLN THEN
D463 DC      146      CPR     NEWLN1      RANGE ERROR.
D464 02 F4   147      BNC     RERR

147 *
150 *          APPLE II BASIC RENUMBER / APPEND SUBROUTINE - PASS 2
151 *
D466 19 30 00 152 PASS2 SET     CHRO,$00B0      ASCII "0".
D469 1A C0 00 153      SET     CHRA,$00C0      ASCII "A".
D46C 27      154 P2A    LD      PRGNDX2
D46D 09      155      CPR     HMEM          IF PRG INDEX = HIMEM THEN DONE PASS 2.
D46E 03 63   156      BC      DONE
D470 E7      157      INR     PRGNDX2      SKIP LENGTH BYTE.
D471 67      158      LDD     @PRGNDX2     LINE NUMBER.
D472 3D      159 UPDATE ST     OLDLN          SAVE OLD LNO.
D473 25      160      LD      TBLSTR
D474 3B      161      ST      TBLNDX2      INIT LNO TABLE INDEX.
D475 21      162      LD      NEWLOW      INIT NEWLN TO NEWLOW.
D476 1C      163      HEX     1C          (WILL SKIP NEXT INSTR)
D477 2C      164 UD2    LD      NEWLN1
D478 A2      165      ADD     NEWINCR      ADD INCR TO NEWLN1.
D479 3C      166      ST      NEWLN1
D47A 2B      167      LD      TBLNDX2      IF LNO TBL IDX = TBLND THEN DONE
D47B B6      168      SUB     TBLND          SCANNING LNO TABLE
D47C 03 07   169      BC      UD3
D47E 6B      170      LDD     @TBLNDX2     NEXT LNO FROM TABLE.
D47F 3D      171      SUB     OLDLN          LOOP TO UD2 IF NOT SAME AS OLDLN.
D480 07 F5   172      BNZ     UD2
D482 C7      173 PCPD   @PRGNDX2     REPLACE OLD LNO WITH CORRESPONDING
D483 2C      174      LD      NEWLN1      NEW LINE.
D484 77      175      STD     @PRGNDX2
D485 1B 2B 00 176 UD3   SET     STRCON,$002B     STR CON TOKEN.
D488 1C      177      HEX     1C          (SKIPS NEXT TWO INSTRUCTIONS)
D489 67      178 GOTCON LDD     @PRGNDX2
D48A FC      179      DCR     MODE          IF MODE = 0 THEN UPDATE LNO REF.
D48B 0B E5   180      BMI     UPDATE
D48D 47      181 ITEM   LD      @PRGNDX2      BASIC TOKEN.
D48E 09      182      CPR     CHRO
D48F 02 09   183      BNC     CHKTKO      CHECK TOKEN FOR SPECIAL.
D491 DA      184      CPR     CHRA          IF >= "0" AND < "A" THEN SKIP CONST
D492 02 F5   185      BNC     GOTCON      OR UPDATE.
D494 F7      186 SKPASC DCR     PRGNDX2
D495 67      187      LDD     @PRGNDX2     SKIP ALL NEG. BYTES OF STR CON, REM,
D496 05 FC   188      BM      SKPASC      OR NAME.
D498 F7      189      DCR     PRGNDX2
D499 47      190      LD      @PRGNDX2

```

```

D49A DB      191 CHKTKO CPR   STRCON   STR CON TOKEN?
D49B 06 F7   192          BZ   SKPASC   YES, SKIP SUBSEQUENT BYTES.
D49D 1C 5D 00 193          SET   REM, $005D
D4A0 DC      194          CPR   REM     REM TOKEN?
D4A1 06 F1   195          BZ   SKPASC   YES, SKIP SUBSEQUENT LINE.
D4A3 08 13   196          BM1  CONST   GOSUB, LOOK FOR LINE NUMBER.
D4A5 FD      197          DCR   R13
D4A6 FD      198          DCR   R13   (TOKEN $5F IS GOTO)
D4A7 06 0F   199          BZ   CONST
D4A9 1D 24 00 200          SET   THEN, $0024
D4AC DD      201          CPR   THEN
D4AD 06 09   202          BZ   CONST   'THEN' LNO, LOOK FOR LNO.
D4AF F0      203          DCR   ACC
D4B0 06 BA   204          BZ   P2A   EOL (TOKEN 01)?
D4B2 1D 74 00 205          SET   LIST, $0074
D4B5 BD      206          SUB   LIST   SET MODEIF LIST OR LIST COMMA.
D4B6 09 01   207          BN#11 CONST2 (TOKENS $74, $75)
D4B8 30      208 CONSTS SUB   ACC     CLEAR MODE FOR LNO
D4B9 3C      209 CONSTS2 ST   MODE   UPDATE CHECK.
D4BA 01 D1   210          BR    ITEM

212 *
213 *
214 *          APPLE II BASIC APPEND SUBROUTINE
215 *
D4BC 20 89 F6 216 APPEND JSR   SW16
D4BF 1C 4E 00 217          SET   SCRC, HIMEM+2
D4C2 CC      218          POPD  @SCRC   SAVE HIMEM
D4C3 38      219          ST    HMEM
D4C4 19 CA 00 220          SET   SCR9, PPL
D4C7 69      221          LOD   @SCR9
D4C8 7C      222          STD   @SCRC   SET HIMEM TO PRESERVE PROGRAM.
D4C9 00      223          RTN
D4CA 20 DF F0 224          JSR   LOAD   LOAD FROM TAPE.
D4CD 20 89 F6 225          JSR   SW16
D4D0 CC      226          POPD  @SCRC   RESTORE HIMEM TO SHOW BOTH PROGRAMS.
D4D1 28      227          LD    HMEM   (OLD AND NEW)
D4D2 7C      228          STD   @SCRC
D4D3 00      229 DONE  RTN   RETURN.
D4D4 60      230          RTS

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00



: ASM

```

1 *****
2 *
3 *      6502 RELOCATION      *
4 *      SUBROUTINE        *
5 *
6 *      1. DEFINE BLOCKS   *
7 *      *A4CA1. A2 ^Y     *
8 *      (^Y IS CTRL-Y)    *
9 *
10 *     2. FIRST SEGMENT   *
11 *     *A4CA1. A2 ^Y     *
12 *     (IF CODE)         *
13 *
14 *     *A4CA1. A2M       *
15 *     (IF MOVE)         *
16 *
17 *     3. SUBSEQUENT SEGMENTS *
18 *     *. A2 ^Y OR *. A2M *
19 *
20 *     W0Z 11-10-77      *
21 *     APPLE COMPUTER INC. *
22 *
23 *****

25 *
26 *     RELOCATION SUBROUTINE EQUATES
27 *
28 R1L EQU $02 SWEET 16 REQ 1.
29 INST EQU $0B 3-BYTE INST FIELD.
30 LENGTH EQU $2F LENGTH CODE
31 YSAV EQU $34 CMND BUF POINTER
32 A1L EQU $3C APPLE-II MON PARAM AREA.
33 A4L EQU $42 APPLE-II MON PARAM REQ 4
34 IN EQU $0200
35 SW16 EQU $F6B9 ; SWEET 16 ENTRY
36 INSD82 EQU $FB8E ; DISASSEMBLER ENTRY
37 NXTA4 EQU $FCB4 POINTER INCR SUBR
38 FRMBEQ EQU $01 SOURCE BLOCK BEGIN
39 FRMEND EQU $02 SOURCE BLOCK END
40 TOBEQ EQU $04 DEST BLOCK BEGIN
41 ADR EQU $06 ADR PART OF INST.
    
```

```

43 *
44 *   6502 RELOCATION SUBROUTINE
45 *
46     ORG   $D4DC
47     OBJ   $A4DC
D4DC A4 34   48 RELOC LDY   YSAV CMND BUF POINTER
D4DE B9 00 02 49     LDA   IN,Y NEXT CMD CHAR
D4E1 C9 AA   50     CMP   ##AA '*'?
D4E3 D0 0C   51     BNE   RELOC2 NO, RELOC CODE SEG.
D4E5 E6 34   52     INC   YSAV ADVANCE POINTER.
D4E7 A2 07   53     LDX   #$07
D4E9 B5 3C   54 INIT   LDA   A1L,X MOVE BLOCK PARAMS
D4EB 95 02   55     STA   R1L,X FROM APPLE-II MON
D4ED CA      56     DEX   AREA TO SW16 AREA
D4EE 10 F9   57     BPL   INIT R1=SOURCE BEG, R2=
D4F0 60      58     RTS   SOURCE END, R4=DEST BEG.
D4F1 A0 02   59 RELOC2 LDY  #$02
D4F3 B1 3C   60 GETINS LDA  (A1L),Y COPY 3 BYTES TO
D4F5 99 0B 00 61     STA   INST,Y SW16 AREA
D4FB 8B      62     DEY
D4F9 10 FB   63     BPL   GETINS
D4FB 20 8E FB 64     JSR   INSDS2 CALCULATE LENGTH OF
D4FE A6 2F   65     LDX   LENGTH INST FROM OPCODE.
D500 CA      66     DEX   0=1 BYTE, 1=2 BYTES,
D501 D0 0C   67     BNE   XLATE 2=3 BYTES.
D503 A5 0B   68     LDA   INST
D505 29 0D   69     AND   #$0D WEED OUT NON-ZERO-PAGE
D507 F0 14   70     BEQ   STINST 2 BYTE INSTS (IMM).
D509 29 0B   71     AND   #$0B IF ZERO PAGE ADR
D50B D0 10   72     BNE   STINST THEN CLEAR HIGH BYTE
D50D B5 0D   73     STA   INST+2
D50F 20 89 F6 74 XLATE JSR  SW16 IF ADR OF ZERO PAGE
D512 22      75     LD   FRMEND OR ABS IS IN SOURCE
D513 D6      76     CPR   ADR (FRM) BLOCK THEN
D514 02 06   77     BNC   SW16RT SUBSTITUTE
D516 26      78     LD   ADR ADR-SOURCE BEG+DEST BEG
D517 B1      79     SUB   FRMBEG
D518 02 02   80     BNC   SW16RT
D51A A4      81     ADD   TOBEG
D51B 36      82     ST   ADR
D51C 00      83 SW16RT RTN
D51D A2 00   84 STINST LDX  #$00
D51F B5 0B   85 STINS2 LDA  INST,X
D521 91 42   86     STA   (A4L),Y COPY LENGTH BYTES
D523 EB      87     INX   OF INST FROM SW16 AREA TO
D524 20 B4 FC 88     JSR   NXTA4
D527 C6 2F   89     DEC   LENGTH DEST SEGMENT. UPDATE
D529 10 F4   90     BPL   STINS2 SOURCE, DEST SEGMENT
D52B 90 C4   91     BCC   RELOC2 POINTERS. LOOP IF NOT
D52D 60      92     RTS   BEYOND SOURCE SEG END.

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00

```

1 *****
2 *
3 *      TAPE VERIFY      *
4 *
5 *      JAN 78          *
6 *      BY W0Z          *
7 *
8 *
9 *****

11 *
12 *      TAPE VERIFY EQUATES
13 *
14 CHKSUM EQU    $2E
15 A1    EQU    $3C
16 HIMEM EQU    $4C ; BASIC HIMEM POINTER
17 PP    EQU    $CA ; BASIC BEGIN OF PROGRAM
18 PRLEN EQU    $CE ; BASIC PROGRAM LENGTH
19 XSAVE EQU    $DB ; PRESERVE X-REG FOR BASIC
20 HDRSET EQU   $F11E ; SETS TAPE POINTERS TO $CE.CF
21 PRQSET EQU   $F12C ; SETS TAPE POINTERS FOR PROGRAM
22 NXTA1 EQU    $FCBA ; INCREMENTS (A1) AND COMPARES TO (A2)
23 HEADR EQU    $FCC9
24 RDBYTE EQU    $FCEC
25 RD2BIT EQU    $FCFA
26 RDBIT EQU    $FCFD
27 PRA1 EQU    $FD92 ; PRINT (A1)-
28 PRBYTE EQU    $FDDA
29 COUT EQU    $FDED
30 FINISH EQU   $FF26 ; CHECK CHECKSUM, RING BELL
31 PRERR EQU    $FF2D

33 *
34 *      TAPE VERIFY ROUTINE
35 *
36          ORG    $D535
37          OBJ    $A535
D535 B6 D8   38 VFYBSC STX    XSAVE ; PRESERVE X-REG FOR BASIC
D537 38     39          SEC
D538 A2 FF   40          LDX    #$FF
D53A A5 4D   41 GETLEN LDA    HIMEM+1 ; CALCULATE PROGRAM LENGTH
D53C F5 CB   42          SBC    PP+1, X ; INTO PRLEN
D53E 95 CF   43          STA    PRLEN+1, X
D540 E8     44          INX
D541 F0 F7   45          BEQ    GETLEN
D543 20 1E F1 46          JSR    HDRSET ; SET UP POINTERS
D546 20 54 D5 47          JSR    TAPEVFY ; DO A VERIFY ON HEADER
D549 A2 01   48          LDX    #$01 ; PREPARE FOR PRQSET
D54B 20 2C F1 49          JSR    PRQSET ; SET POINTERS FOR PROGRAM VERIFY
D54E 20 54 D5 50          JSR    TAPEVFY
D551 A6 D8   51          LDX    XSAVE ; RESTORE X-REG
D553 60     52          RTS

```

```

53 *
54 * TAPE VERIFY RAM IMAGE (A1.A2)
55 *
D554 20 FA FC 56 TAPEVFY JSR RD2BIT
D557 A9 16 57 LDA ##16
D559 20 C9 FC 58 JSR HEADR ; SYNCHRONIZE ON HEADER
D55C 85 2E 59 STA CHKSUM ; INITIALIZE CHKSUM
D55E 20 FA FC 60 JSR RD2BIT
D561 A0 24 61 VRFY2 LDY ##24
D563 20 FD FC 62 JSR RDBIT
D566 B0 F9 63 BCS VRFY2 ; CARRY SET IF READ A '1' BIT
D568 20 FD FC 64 JSR RDBIT
D56B A0 3B 65 LDY ##3B
D56D 20 EC FC 66 VRFY3 JSR RDBYTE ; READ A BYTE
D570 F0 0E 67 BEQ EXTDEL ; ALWAYS TAKEN
D572 45 2E 68 VFYLOOP EOR CHKSUM ; UPDATE CHECKSUM
D574 85 2E 69 STA CHKSUM
D576 20 BA FC 70 JSR NXTA1 ; INCREMENT A1, SET CARRY IF A1>A2
D579 A0 34 71 LDY ##34 ; ONE LESS THAN USED IN READ FOR EXTRA 12
D57B 90 F0 72 BCC VRFY3 ; LOOP UNTIL A1>A2
D57D 4C 26 FF 73 JMP FINISH ; VERIFY CHECKSUM&RING BELL
D580 EA 74 EXTDEL NOP ; EXTRA DELAY TO EQUALIZE TIMING
D581 EA 75 NOP ; (+12 USEC)
D582 EA 76 NOP
D583 C1 3C 77 CMP (A1,X) ; BYTE THE SAME?
D585 F0 EB 78 BEQ VFYLOOP ; IT MATCHES, LOOP BACK
D587 48 79 PHA ; SAVE WRONG BYTE FROM TAPE
D588 20 2D FF 80 JSR PRERR ; PRINT "ERR"
D58B 20 92 FD 81 JSR PRA1 ; OUTPUT (A1)"-"
D58E B1 3C 82 LDA (A1),Y
D590 20 DA FD 83 JSR PRBYTE ; OUTPUT CONTENTS OF A1
D593 A9 A0 84 LDA ##A0 ; PRINT A BLANK
D595 20 ED FD 85 JSR COUT
D598 A9 AB 86 LDA ##AB ; '('
D59A 20 ED FD 87 JSR COUT
D59D 68 88 PLA ; OUTPUT BAD BYTE FROM TAPE
D59E 20 DA FD 89 JSR PRBYTE
D5A1 A9 A9 90 LDA ##A9 ; ')'
D5A3 20 ED FD 91 JSR COUT
D5A6 A9 8D 92 LDA ##8D ; CARRIAGE RETURN, AND RETURN TO CALLER
D5A8 4C ED FD 93 JMP COUT

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00

: ASM

```

1 *****
2 *
3 *      RAMTEST:
4 *
5 *      BY WOZ
6 *      6/77
7 *
8 *      COPYRIGHT 1978 BY:
9 *      APPLE COMPUTER INC
10 *
11 *****

13 *
14 *      EQUATES:
15 *
16 DATA EQU $0 TEST DATA $00 OR $FF
17 NDATA EQU $1 INVERSE TEST DATA.
18 TESTD EQU $2 GALLOP DATA.
19 R3L EQU $6 AUX ADR POINTER.
20 R3H EQU $7
21 R4L EQU $8 AUX ADR POINTER.
22 R4H EQU $9
23 R5L EQU $A AUX ADR POINTER.
24 R5H EQU $B
25 R6L EQU $C GALLOP BIT MASK.
26 R6H EQU $D ($0001 TO 2^N)
27 YSAV EQU $34 MONITOR SCAN INDEX.
28 A1H EQU $3D BEGIN TEST BLOCK ADR.
29 A2L EQU $3E LEN (PAGES) FROM MON.
30 SETCTLY EQU $D5B0 ; SET UP CNTRL-Y LOCATION
31 PRBYTE EQU $FD0A BYTE PRINT SUBR.
32 COUT EQU $FDED CHAR OUT SUBR.
33 PRERR EQU $FF2D PRINTS 'ERR-BELL'
34 BELL EQU $FF3A
    
```

```

36 *
37 *           RAMTEST:
38 *
39           ORG   #D5BC
40           OBJ   #A5BC
D5B8 A9 C3   41 SETUP LDA   #C3 ,SET UP CNTRL-Y LOCATION
D5BE A0 D5   42         LDY   #D5
D5C0 4C B0 D5 43         JMP   SETCTLY
D5C3 A9 00   44 RAMTST LDA   #00 TEST FOR #00,
D5C5 20 D0 D5 45         JSR   TEST
D5C8 A9 FF   46         LDA   #FF THEN #FF.
D5CA 20 D0 D5 47         JSR   TEST
D5CD 4C 3A FF 48         JMP   BELL
D5D0 B5 00   49 TEST  STA   DATA
D5D2 49 FF   50         EOR   #FF
D5D4 B5 01   51         STA   NDATA
D5D6 A5 3D   52         LDA   A1H
D5D8 B5 07   53         STA   R3H INIT (R3L,R3H),
D5DA B5 09   54         STA   R4H  (R4L,R4H), (R5L,R5H)
D5DC B5 0B   55         STA   R5H  TO TEST BLOCK BEGIN
D5DE A0 00   56         LDY   #00 ADDRESS.
D5E0 B4 06   57         STY   R3L
D5E2 B4 08   58         STY   R4L
D5E4 B4 0A   59         STY   R5L
D5E6 A6 3E   60         LDX   A2L LENGTH (PAGES).
D5E8 A5 00   61         LDA   DATA
D5EA 91 08   62 TEST01 STA  (R4L), Y SET ENTIRE TEST
D5EC C8      63         INY   BLOCK TO DATA.
D5ED D0 FB   64         BNE   TEST01
D5EF E6 09   65         INC   R4H
D5F1 CA      66         DEX
D5F2 D0 F6   67         BNE   TEST01
D5F4 A6 3E   68         LDX   A2L
D5F6 B1 06   69 TEST02 LDA  (R3L), Y VERIFY ENTIRE
D5F8 C5 00   70         CMP   DATA TEST BLOCK.
D5FA F0 13   71         BEQ   TEST03
D5FC 48      72         PHA   PRESERVE BAD DATA.
D5FD A5 07   73         LDA   R3H
D5FF 20 DA FD 74         JSR   PRBYTE PRINT ADDRESS,
D602 98      75         TYA
D603 20 8A D6 76         JSR   PRBYSP
D606 A5 00   77         LDA   DATA THEN EXPECTED DATA,
D608 20 8A D6 78         JSR   PRBYSP
D60B 68      79         PLA   THEN BAD DATA,
D60C 20 7F D6 80         JSR   PRBYCR THEN 'ERR-BELL'.
D60F C8      81 TEST03 INY
D610 D0 E4   82         BNE   TEST02
D612 E6 07   83         INC   R3H
D614 CA      84         DEX
D615 D0 DF   85         BNE   TEST02
D617 A6 3E   86         LDX   A2L LENGTH.
D619 A5 01   87 TEST04 LDA  NDATA
D61B 91 0A   88         STA  (R5L), Y SET TEST CELL TO
D61D B4 0D   89         STY  R6H  NDATA AND R6
D61F B4 0C   90         STY  R6L  (GALLOP BIT MASK)
D621 E6 0C   91         INC  R6L  TO #0001.
D623 A5 01   92 TEST05 LDA  NDATA
D625 20 45 D6 93         JSR  TEST6 GALLOP WITH NDATA.
D628 A5 00   94         LDA  DATA
D62A 20 45 D6 95         JSR  TEST6 THEN WITH DATA.
D62D 06 0C   96         ASL  R6L
D62F 26 0D   97         ROL  R6H SHIFT GALLOP BIT
D531 A5 0D   98         LDA  R6H  MASK FOR NEXT

```

```

D633 C5 3E      99      CMP      A2L  NEIGHBOR.  DONE
D635 90 EC      100     BCC      TEST05  IF > LENGTH.
D637 A5 00      101     LDA      DATA
D639 91 0A      102     STA      (R5L),Y RESTORE TEST CELL.
D63B E6 0A      103     INC      R5L
D63D D0 DA      104     BNE      TEST04
D63F E6 0B      105     INC      R5H INCR TEST CELL
D641 CA         106     DEX      POINTER AND DECR
D642 D0 D5      107     BNE      TEST04  LENGTH COUNT.
D644 60         108     RTS1    RTS
D645 85 02      109     TEST6  STA      TESTD SAVE GALLOP DATA.
D647 A5 0A      110     LDA      R5L
D649 45 0C      111     EOR      R6L SET R4 TO R5
D64B 85 0B      112     STA      R4L  EX-OR R6
D64D A5 0B      113     LDA      R5H  FOR NEIGHBOR
D64F 45 0D      114     EOR      R6H  ADDRESS (1 BIT
D651 85 09      115     STA      R4H  DIFFERENCE).
D653 A5 02      116     LDA      TESTD
D655 91 0B      117     STA      (R4L),Y GALLOP TEST DATA.
D657 B1 0A      118     LDA      (R5L),Y CHECK TEST CELL
D659 85 01      119     CMP      NDATA  FOR CHANGE.
D65B F0 E7      120     BEQ      RTS1 (OK).
D65D 48         121     PHA      PRESERVE FAIL DATA.
D65E A5 0B      122     LDA      R5H
D660 20 DA FD   123     JSR      PRBYTE PRINT TEST CELL
D663 A5 0A      124     LDA      R5L  ADDRESS,
D665 20 8A D6   125     JSR      PRBYSP
D668 A5 01      126     LDA      NDATA
D66A 91 0A      127     STA      (R5L),Y (REPLACE CORRECT DATA)
D66C 20 8A D6   128     JSR      PRBYSP  THEN TEST DATA BYTE.
D66F 68         129     PLA
D670 20 8A D6   130     JSR      PRBYSP  THEN FAIL DATA.
D673 A5 09      131     LDA      R4H
D675 20 DA FD   132     JSR      PRBYTE
D678 A5 0B      133     LDA      R4L  THEN NEIGHBOR ADR.
D67A 20 8A D6   134     JSR      PRBYSP
D67D A5 02      135     LDA      TESTD  THEN GALLOP DATA.
D67F 20 8A D6   136     PRBYCR JSR      PRBYSP OUTPUT BYTE, SPACE.
D682 20 2D FF   137     JSR      PRERR  THEN 'ERR-BELL'.
D685 A9 8D      138     LDA      #8D ASCII CAR. RETURN.
D687 4C ED FD   139     JMP      COUT
D68A 20 DA FD   140     PRBYSP JSR      PRBYTE
D68D A9 A0      141     LDA      #A0 OUTPUT BYTE, THEN
D68F 4C ED FD   142     JMP      COUT  SPACE.
                143     ORQ      $3FB
03FB 4C C3 D5   144     USRLOC JMP      RAMTST ENTRY FROM MON (CTRL-Y)

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00

```

*****
4 *
5 * MUSIC SUBROUTINE
6 *
7 * GARY J. SHANNON
8 *
*****
10      ORG      $D717
11 *
12 * ZERO PAGE WORK AREAS
13 * PARAMETER PASSING AREAS
14 *
15 DOWNTIME EQU    $0
16 UPTIME EQU     $1
17 LENGTH EQU     $2
18 VOICE EQU      $2FD
19 LONG EQU       $2FE
20 NOTE EQU       $2FF
21 SPEAKER EQU    $C030
D717 4C 4E D7    22 ENTRY JMP     LOOKUP
23 *
24 * PLAY ONE NOTE
25 *
26 * DUTY CYCLE DATA IN 'UPTIME' AND
27 * 'DOWNTIME', DURATION IN 'LENGTH'
28 *
29 *
30 * CYCLE IS DIVIDED INTO 'UP' HALF
31 * AND 'DOWN' HALF
32 *
D71A A4 01      33 PLAY  LDY     UPTIME ; GET POSITIVE PULSE WIDTH
D71C AD 30 C0   34      LDA     SPEAKER ; TOGGLE SPEAKER
D71F E6 02      35 PLAY2 INC     LENGTH ; DURATION
D721 D0 05      36      BNE     PATH1 ; NOT EXPIRED
D723 E6 03      37      INC     LENGTH+1
D725 D0 05      38      BNE     PATH2
D727 60         39      RTS     ; DURATION EXPIRED
D728 EA         40 PATH1 NOP     ; DUMMY
D729 4C 2C D7   41      JMP     PATH2 ; TIME ADJUSTMENTS
D72C 88         42 PATH2 DEY     ; DECREMENT WIDTH
D72D F0 05      43      BEQ     DOWN  ; WIDTH EXPIRED
D72F 4C 32 D7   44      JMP     PATH3 ; IF NOT, USE UP
45 *
46 * DOWN HALF OF CYCLE
47 *
D732 D0 EB      48 PATH3 BNE     PLAY2 ; SAME # CYCLES
D734 A4 00      49 DOWN  LDY     DOWNTIME ; GET NEGATIVE PULSE WIDTH
D736 AD 30 C0   50      LDA     SPEAKER ; TOGGLE SPEAKER
D739 E6 02      51 PLAY3 INC     LENGTH ; DURATION
D73B D0 05      52      BNE     PATH4 ; NOT EXPIRED
D73D E6 03      53      INC     LENGTH+1
D73F D0 05      54      BNE     PATH5
D741 60         55      RTS     ; DURATION EXPIRED
D742 EA         56 PATH4 NOP     ; DUMMY
D743 4C 46 D7   57      JMP     PATH5 ; TIME ADJUSTMENTS
D746 88         58 PATH5 DEY     ; DECREMENT WIDTH
D747 F0 D1      59      BEQ     PLAY  ; BACK TO UP-SIDE
D749 4C 4C D7   60      JMP     PATH6 ; USE UP SOME CYCLES
D74C D0 EB      61 PATH6 BNE     PLAY3 ; REPEAT

```



```

62 *
63 * NOTE TABLE LOOKUP SUBROUTINE
64 *
65 * GIVEN NOTE NUMBER IN 'NOTE'
66 * DURATION COUNT IN 'LONG'
67 * FIND 'UPTIME' AND 'DOWNTIME'
68 * ACCORDING TO DUTY CYCLE CALLED
69 * FOR BY 'VOICE'.
70 *
D74E AD FF 02 71 LOOKUP LDA NOTE ; GET NOTE NUMBER
D751 0A 72 ASL ; DOUBLE IT
D752 A8 73 TAY
D753 B9 96 D7 74 LDA NOTES,Y ; GET UPTIME
D756 85 00 75 STA DOWNTIME ; SAVE IT
D758 AD FD 02 76 LDA VOICE ; GET DUTY CYCLE
D758 4A 77 SHIFT LSR
D75C F0 04 78 BEQ DONE ; SHIFT WIDTH COUNT
D75E 46 00 79 LSR DOWNTIME ; ACCORDING TO VOICE
D760 D0 F9 80 BNE SHIFT
D762 B9 96 D7 81 DONE LDA NOTES,Y ; GET ORIGINAL
D765 38 82 SEC
D766 E5 00 83 SBC DOWNTIME ; COMPUTE DIFFERENCE
D768 85 01 84 STA UPTIME ; SAVE IT
D76A C8 85 INY ; NEXT ENTRY
D76B B9 96 D7 86 LDA NOTES,Y ; GET DOWNTIME
D76E 65 00 87 ADC DOWNTIME ; ADD DIFFERENCE
D770 85 00 88 STA DOWNTIME
D772 A9 00 89 LDA #0
D774 38 90 SEC
D775 ED FE 02 91 SBC LONG ; GET COMPLIMENT OF DURATION
D778 85 03 92 STA LENGTH+1 MOST SIGNIFICANT BYTE
D77A A9 00 93 LDA #0
D77C 85 02 94 STA LENGTH
D77E A5 01 95 LDA UPTIME
D780 D0 98 96 BNE PLAY ; IF NOT NOTE #0, PLAY IT
97 *
98 * 'REST' SUBROUTINE' PLAYS NOTE #0
99 * SILENTLY, FOR SAME DURATION AS
100 * A REGULAR NOTE.
101 *
D782 EA 102 REST NOP ; DUMMY
D783 EA 103 NOP ; CYCLE USERS
D784 4C 87 D7 104 JMP REST2 ; TO ADJUST TIME
D787 E6 02 105 REST2 INC LENGTH
D789 D0 05 106 BNE REST3
D78B E6 03 107 INC LENGTH+1
D78D D0 05 108 BNE REST4
D78F 60 109 RTS ; IF DURATION EXPIRED
D790 EA 110 REST3 NOP ; USE UP 'INC' CYCLES
D791 4C 94 D7 111 JMP REST4
D794 D0 EC 112 REST4 BNE REST ; ALWAYS TAKEN

```

```

113 *
114 * NOTE TABLES
115 *
D796 00 00 F6 116 NOTES HEX 00, 00, F6, F6, E8, E8, DB, DB
D79E CF CF C3 117 HEX CF, CF, C3, C3, B8, B8, AE, AE
D7A6 A4 A4 9B 118 HEX A4, A4, 9B, 9B, 92, 92, 8A, 8A
D7AE 82 82 7B 119 HEX 82, 82, 7B, 7B, 74, 74, 6D, 6E
D7B6 67 68 61 120 HEX 67, 68, 61, 62, 5C, 5C, 57, 57
D7BE 52 52 4D 121 HEX 52, 52, 4D, 4E, 49, 49, 45, 45
D7C6 41 41 3D 122 HEX 41, 41, 3D, 3E, 3A, 3A, 36, 37
D7CE 33 34 30 123 HEX 33, 34, 30, 31, 2E, 2E, 2B, 2C
D7D6 29 29 26 124 HEX 29, 29, 26, 27, 24, 25, 22, 23
D7DE 20 21 1E 125 HEX 20, 21, 1E, 1F, 1D, 1D, 1B, 1C
D7E6 1A 1A 18 126 HEX 1A, 1A, 1B, 19, 17, 17, 15, 16
D7EE 14 15 13 127 HEX 14, 15, 13, 14, 12, 12, 11, 11
D7F6 10 10 0F 128 HEX 10, 10, 0F, 10, 0E, 0F

```

--- END ASSEMBLY ---

TOTAL ERRORS: 00



# APPENDIX SUMMARY OF PROGRAMMER'S AID COMMANDS

92	Renumber
92	Append
92	Tape Verify (BASIC)
93	Tape Verify (Machine Code and Data)
93	Relocate (Machine Code and Data)
94	RAM Test
94	Music
95	High-Resolution Graphics
96	Quick Reference to High-Resolution Graphics Information

## Chapter 1: RENUMBER

- (a) To renumber an entire BASIC program:

```
CLR
START = 1000
STEP = 10
CALL -10531
```

- (b) To renumber a program portion:

```
CLR
START = 200
STEP = 20

FROM = 300          (program portion
TO = 500           to be renumbered)

CALL -10521
```

## Chapter 2: APPEND

- (a) Load the second BASIC program, with high line numbers:

```
LOAD
```

- (b) Load and append the first BASIC program, with low line numbers:

```
CALL -11076
```

## Chapter 3: TAPE VERIFY (BASIC)

- (a) Save current BASIC program on tape:

```
SAVE
```

- (b) Replay the tape, after:

```
CALL -10955
```

## Chapter 4: TAPE VERIFY (Machine Code and Data)

- (a) From the Monitor, save the portion of memory on tape:

address1 . address2 W return

- (b) Initialize Tape Verify feature:

D52EG return

- (c) Replay the tape, after:

address1 . address2 ctrl Y return

Note: spaces shown within the above commands are for easier reading only; they should not be typed.

## Chapter 5: RELOCATE (Machine Code and Data)

- (a) From the Monitor, initialize Code-Relocation feature:

D4D5G return

- (b) Blocks are memory locations from which program runs. Specify Destination and Source Block parameters:

Dest Blk Beg < Source Blk Beg . Source Blk End ctrl Y \* return

- (c) Segments are memory locations where parts of program reside. If first program Segment is code, Relocate:

Dest Seg Beg < Source Seg Beg . Source Seg End ctrl Y return

If first program Segment is data, Move:

Dest Seg Beg < Source Seg Beg . Source Seg End return

- (d) In order of increasing address, Move subsequent contiguous data Segments:

. Source Segment End ctrl Y return

and Relocate subsequent contiguous code Segments:

. Source Segment End M return

Note: spaces shown within the above commands are for easier reading only; they should not be typed.

## Chapter 6: RAM TEST

- (a) From the Monitor, initialize RAM Test program:

D5BCG return

- (b) To test a portion of memory:

address . pages ctrl Y return (test begins at address,  
continues for length pages.)

Note: test length, pages\*100, must not be greater than  
starting address. One page = 256 bytes (\$100 bytes, in Hex).

- (c) To test more memory, do individual tests or concatenate:

addr1.pages1 ctrl Y addr2.pages2 ctrl Y addr3.pages3 ctrl Y return

Example, for a 48K system:

400.4 ctrl Y 800.8 ctrl Y 1000.10 ctrl Y 2000.20 ctrl Y  
3000.20 ctrl Y 4000.40 ctrl Y 7000.20 ctrl Y 8000.40  
ctrl Y return

- (d) To repeat test indefinitely:

N complete test 34:0 type one space return

Note: except where specified in step (d), spaces shown within the above  
commands are for easier reading only; they should not be typed.

## Chapter 7: MUSIC

- (a) Assign appropriate variable names to CALL  
and POKE locations (optional):

MUSIC = -10473  
PITCH = 767  
TIME = 766  
TIMBRE = 765

- (b) Set parameters for next note:

POKE PITCH, p	(p = 1 to 50; 32 = middle C)
POKE TIME, m	(m = 1 to 255; 170 = 1 second)
POKE TIMBRE, t	(t = 2, 8, 16, 32 or 64)

- (c) Sound the note:

CALL MUSIC

## Chapter 8: HIGH-RESOLUTION GRAPHICS

(a) Set order of parameters (first lines of program):

```
1 XØ = YØ = COLR
2 SHAPE = ROT = SCALE           (if shapes are used)
```

(b) Assign appropriate variable names to subroutine calling addresses (optional; omit any subroutines not used in program):

```
1Ø INIT = -12288 : CLEAR = -12274 : BKGND = -11471
11 POSN = -11527 : PLOT = -115Ø6 : LINE = -115ØØ
12 DRAW = -11465 : DRAW1 = -11462
13 FIND = -1178Ø : SHLOAD = -11335
```

(c) Assign appropriate variable names to color values (optional; omit any colors not used in program):

```
2Ø BLACK = Ø : LET GREEN = 42 : VIOLET = 85
21 WHITE = 127 : ORANGE = 17Ø : BLUE = 213
22 BLACK2 = 128 : WHITE2 = 255
```

(d) Initialize:

```
3Ø CALL INIT
```

(e) Change screen conditions, if desired. Set appropriate parameter values, and CALL desired subroutines by name.

Example:

```
4Ø COLR = VIOLET : CALL BKGND : REM TURN BACKGROUND VIOLET
5Ø FOR I = Ø TO 279 STEP 5
6Ø XØ = 14Ø : YØ = 15Ø : COLR = WHITE : REM SET PARAMETERS
7Ø CALL POSN : REM MARK THE "CENTER"
8Ø XØ = I : YØ = Ø : REM SET NEW PARAMETERS
9Ø CALL LINE : REM DRAW LINE TO EDGE
1ØØ NEXT I : END
```



## QUICK REFERENCE TO HIGH-RESOLUTION INFORMATION

<u>Subroutine Name</u>	<u>CALLing Address</u>	<u>Parameters Needed</u>
INIT	-12288	
CLEAR	-12274	
BKGND	-11471	COLR
POSN	-11527	XØ, YØ, COLR
PLOT	-115Ø6	XØ, YØ, COLR
LINE	-115ØØ	XØ, YØ, COLR
DRAW	-11465	XØ, YØ, COLR, SHAPE, ROT, SCALE
DRAW1	-11462	SHAPE, ROT, SCALE
FIND	-1178Ø	
SHLOAD	-11335	

<u>Color Name</u>	<u>COLR Value</u>	<u>Color Name</u>	<u>COLR Value</u>
BLACK	Ø	BLACK2	128
GREEN	42	ORANGE	17Ø
VIOLET	85	BLUE	213
WHITE	127	WHITE2	255

(Note: on systems below S/N 6ØØØ, colors in the second column appear identical to those in the first column)

### CHANGING THE HIGH-RESOLUTION GRAPHICS DISPLAY

Full-Screen Graphics	POKE -163Ø2, Ø
Mixed Graphics-Plus-Text (Default)	POKE -163Ø1, Ø
Page 2 Display	POKE -16299, Ø
Page 1 Display (Normal)	POKE -163ØØ, Ø
Page 2 Plotting	POKE 8Ø6, 64
Page 1 Plotting (Default)	POKE 8Ø6, 32

(Note: CALL INIT sets mixed graphics-plus-text, and Page 1 plotting, but does not reset to Page 1 display.)

Collision Count for Shapes                      PEEK (81Ø)

(Note: the change in PEEKed value indicates collision.)



10260 Bandley Drive  
Cupertino, California 95014  
(408) 996-1010