PROGRAMMER’S AID #1

INSTALLATION AND OPERATING MANUAL

Apple Utility Programs
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
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)

10 Verifying a portion of memory saved on tape
10 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
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
  56 Assigning parameter values: SHAPE, SCALE and ROTation
  57 DRAWing shapes
  58 Linking shapes: DRAWl
  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**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>High—Resolution</td>
<td>$D000-$D3FF</td>
</tr>
<tr>
<td>76</td>
<td>Renumber</td>
<td>$D400-$D4BB</td>
</tr>
<tr>
<td>79</td>
<td>Append</td>
<td>$D4BC-$D4D4</td>
</tr>
<tr>
<td>80</td>
<td>Relocate</td>
<td>$D4DC-$D52D</td>
</tr>
<tr>
<td>82</td>
<td>Tape Verify (Basic)</td>
<td>$D535-$D553</td>
</tr>
<tr>
<td>83</td>
<td>Tape Verify (6502 Code &amp; Data)</td>
<td>$D554-$5AA</td>
</tr>
<tr>
<td>84</td>
<td>RAM Test</td>
<td>$D5BC-$D691</td>
</tr>
<tr>
<td>87</td>
<td>Music</td>
<td>$D717-$D7F8</td>
</tr>
</tbody>
</table>
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.

XI
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 he turned on.
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 commands

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:

\[
\begin{align*}
\text{START} & = 200 \\
\text{STEP} & = 20 
\end{align*}
\]

The next two variables specify the program portion’s range of line numbers before renumbering.

\[
\begin{align*}
\text{FROM} & = 300 \\
\text{TO} & = 500 
\end{align*}
\]

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

CALL —10521

If the program was previously numbered

\[
\begin{align*}
100 \\
120 \\
300 \\
310 \\
402 \\
500 \\
2000 \\
2022 
\end{align*}
\]

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

\[
\begin{align*}
300—>200 \\
310—>220 \\
402—>240 \\
500—>260 
\end{align*}
\]

and the new program line numbers will be

\[
\begin{align*}
100 \\
120 \\
200 \\
220 \\
240 \\
260 \\
2000 \\
2022 
\end{align*}
\]
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.
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.
Verifying a portion of memory SAVEd on tape
Comments
VERIFYING A PORTION OF MEMORY SAVED ON TAPE

Users of machine—language routine 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. Straiforward relocation
19 Example 2. Index into Block
20 Example 3. Immediate address reference
20 Example 4. Usable 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
LOCATING 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 snail 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 dissembler, 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. No 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 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 RON). In some cases, the Source and Destination Segments may overlap.
Segments: Locations in APPLE II where Programs Reside During Relocation

Blocks: Locations where Programs Run During Relocation

$800 \rightarrow \text{Original program runs from location $400 on APPLE I}$ (Source)

$B87 \rightarrow \text{Relocated version runs from location $D800 on APPLE II}$ (Destination)

$C00 \rightarrow \text{Source Block $400-787$: Destination Block: $D800-DB87$}

$F87 \rightarrow \text{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 date 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 N 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.
EXAMPLE 1. Straightforward Relocation

Program A resides and runs in locations $800—$97F. The relocated version will reside and run in locations $A00—$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 I 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 $730 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 1. 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.

<table>
<thead>
<tr>
<th>Addresses during relocation</th>
<th>Source And Destination Segments</th>
<th>Source And Destination Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800—-$97F</td>
<td>CODE $800—-$97F</td>
<td>Rums from locations $400-$78F on an APPLE 1, but must be relocated to run from locations $D800-$DB8F on the APPLE II.</td>
</tr>
<tr>
<td>$980—-$9FF</td>
<td>DATA $980—-$9FF</td>
<td></td>
</tr>
<tr>
<td>$B8F—-$A00</td>
<td>CODE $A00—-$B8F</td>
<td></td>
</tr>
</tbody>
</table>

(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):

0800 < 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
SOURCE SEGMENTS: $S00-$97F

DEST BLOCK: $80-$9F
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.

<table>
<thead>
<tr>
<th>SOURCE SEGMENTS</th>
<th>DEST SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800––&gt;</td>
<td>$800––&gt;</td>
</tr>
<tr>
<td>Program A</td>
<td>Program A</td>
</tr>
<tr>
<td>$800—$8A6</td>
<td>$800—$8A6</td>
</tr>
<tr>
<td>$8A6––&gt;</td>
<td>$8A6––&gt;</td>
</tr>
<tr>
<td>Unused</td>
<td>$8A7––&gt;</td>
</tr>
<tr>
<td>$900––&gt;</td>
<td>$998––&gt;</td>
</tr>
<tr>
<td>Program B</td>
<td>Program B</td>
</tr>
<tr>
<td>$900—$9F1</td>
<td>$8A7—$998</td>
</tr>
<tr>
<td>$9F1––&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SOURCE BLOCK: $900-$9F1</th>
<th>DEST BLOCKS: $8A7-$998</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE SEGMENTS: $800-$8A6 (A)</td>
<td>DEST SEGMENTS: $800-$8A6 (A)</td>
</tr>
<tr>
<td>$900-$9F1 (B)</td>
<td>$8A7-$998 (B)</td>
</tr>
</tbody>
</table>

(a) Initialize the Code-Relocation feature:

04B5G 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.

<table>
<thead>
<tr>
<th>SOURCE SEGMENTS</th>
<th>DEST SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800––&gt; CODE $800 -$88F</td>
<td>$800––&gt; CODE $800 -$88F</td>
</tr>
<tr>
<td>DATA $890 -$8AF</td>
<td>DATA $890 -$8AF</td>
</tr>
<tr>
<td>CODE $8B0 -$90F</td>
<td>CODE $830 -$90B</td>
</tr>
<tr>
<td>DATA $910 -$93F</td>
<td>DATA $90C -$933</td>
</tr>
<tr>
<td>CODE $940 -$97F</td>
<td>CODE $93C -$97B</td>
</tr>
</tbody>
</table>

Remove 4 bytes here ——> $8B0 -$8BF ($8C0 -$8C3) 

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

(a) Initialize Code-Relocation feature:
reset D4D5G return

(b) Specify Destination and Source Blocks:
8C0 < 8C4 . 97F ctrl Y* return

(e) Relocate Code Segments and Move Data Segments, in ascending address Sequence

800 < . 88F ctrl Y return (first segment, code, “in place”) . 8AF M return (data) . 8BF ctrl Y return (code)
8C0 < 8C4 . 90F ctrl Y return (first segment, code, not “in place”) . 93F M return (data) . 97F ctrl Y return (code)

(d) Relative branches crossing the deletion boundary will be incorrect, since the relocation process does not modify them (only zero -page and absolute memory references). The user must patch these by hand.
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) $A43—$A42 (data)
$FB19—$FB18 (code) $319—$B1D (data)
$FB1E—$FFCB (code) $B1E—$FCB (code)
$FFCC—$FFFF (data) $FCC—$FFF (data)

IMMEDIATE ADDRESS REFERENCES (see EXAMPLE 3) $FBF
$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 * 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)
. FFFC 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 Addrl 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:

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

3. If ctrl Y is not followed by *, then a segment relocation is initiated at RELOC2 ($3BB). Throughout, Addrl ($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 $2r)

<table>
<thead>
<tr>
<th>Instruction Type in A-REG</th>
<th>LENGTH (in $2F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>0</td>
</tr>
<tr>
<td>1 byte</td>
<td>0</td>
</tr>
<tr>
<td>2 byte</td>
<td>1</td>
</tr>
<tr>
<td>3 byte</td>
<td>2</td>
</tr>
</tbody>
</table>

25
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) them 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.
Each Move or relocation 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 APPLeS 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 labeled “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 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**

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

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-I
```
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 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:

\[
\begin{align*}
\text{MUSIC} &= -10473 \\
\text{PITCH} &= 767 \\
\text{TIME} &= 766 \\
\text{TIMBRE} &= 765
\end{align*}
\]

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 TINE, 40: POKE TIMBRE, 32
30 FOR I = 1 TO 49
40 POKE PITCH, I
40 POKE PITCH, I
50 CALL MUSIC
60 NEXT I: END

Where K 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
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: SCALE AND ROTation
57   DRAWing shapes
58   Linking shapes: DRAW1
59   Collisions

60  Part F: Technical information
60   Locations of the High-Resolution subroutines
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 If 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 6000), 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 node 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 XO, but X or XCOORD could not.
The parameters SHAPE, ROT, and SCALE are used only by the subroutines that
draw shapes (DRAW and DRAWl, see PART E). These parameters may be omitted
from programs using only the PLOT and LINE features:

0 X0 = Y0 = 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 : BKGD = -11471
6 POSN = -11527 : PLOT = 11506 : LINE = -11500
7 DRAW = -11465 : DRAW1 = -11462
8 FIND = -11780 : SULOAD = -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.

10 BLACK = 0 : LET GREEN = 42 : VIOLET = 85
11 WHITE = 127 : ORANGE = 170 : BLUE = 213
12 BLACK2 = 128 : WHITE2 = 255

Any integer from 0 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 6000, 170 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 = YO = COLR = SHAPE = ROT = SCALE
5 INIT =- 12288k : 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 BKGNDE =- 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, BKCND, POS, 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 (S4000-5FFF), 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 corn 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 BRGND : 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 X, Y0 and COLR. X must in the range from 0 through 279, Y0 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 K-coordinate 35, Y-coordinate 55, and a violet dot at K-coordinate 85, Y-coordinate 90:

0   X0 = COLR : REM SET PARAMETERS
5   INIT = —12288 : PLOT =- 11506 : REM DEFINE SUBROUTINES
10  WHITE = 127 : VIOLET = 85 : REM DEFINE COLORS
20  CALL INIT :  REM INITIALIZE SUBROUTINES
30  COLR = WHITE :REM ASSIGN PARAMETER VALUES
40  X0 = 35 : Y0 = 55
50  CALL PLOT : REM PLOT WITH ASSIGNED PARAMETER VALUES
60  COLR = VIOLET : REM ASSIGN NEW PARAMETER VALUES
70  X0 =  85 : Y0 = 90
80  CALL PLOT  REM PLOT WITH NEW PARAMETER VALUES
90  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 X0, Y0, 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, X0 and Y0 (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 DRAWl (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:

<table>
<thead>
<tr>
<th>Section:</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Number:</td>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies:</td>
<td>D D P D D P D D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- = 01 move right  If P = 0 don’t plot
- = 10 move down =1 do plot
- = 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 DRAWl 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 II 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:

<table>
<thead>
<tr>
<th>Section</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte0</td>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>001 or 01</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>01</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>010 or 10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>011 or 11</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>01</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>000</td>
<td>000</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>110</td>
</tr>
</tbody>
</table>

This Vector Cannot Plot or Move up

Denotes End of Shape Definition

Move Only

Plot & Move

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.

<table>
<thead>
<tr>
<th>Bytes Recoded in Hex</th>
<th>Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0: 0 0 0 1 0 0 1 0 = 1 2</td>
<td>Binary: 0000 = 0</td>
</tr>
<tr>
<td>1 0 0 1 1 1 1 1 = 3 F</td>
<td>Hex: 0001 = 1</td>
</tr>
<tr>
<td>2 0 0 1 0 0 0 0 = 2 0</td>
<td></td>
</tr>
<tr>
<td>3 0 1 1 0 0 1 0 0 = 6 4</td>
<td></td>
</tr>
<tr>
<td>4 0 0 1 0 1 1 0 1 = 2 D</td>
<td></td>
</tr>
<tr>
<td>5 0 0 0 1 0 1 0 1 = 1 5</td>
<td></td>
</tr>
<tr>
<td>6 0 0 0 1 0 0 0 0 = 3 6</td>
<td></td>
</tr>
<tr>
<td>7 0 0 0 1 1 1 0 0 = 2 E</td>
<td></td>
</tr>
<tr>
<td>8 0 0 0 0 0 1 1 1 = 0 7</td>
<td></td>
</tr>
<tr>
<td>9 0 0 0 0 0 0 0 0 = 0 0 Denotes End</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2

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

Start byte $S+0$ to $n$ (0 to FF)

- Total Number of Shape Definitions
- $D1$: Index to First Byte of Shape Definition #1, Relative to $S$
- $D2$: Index to First Byte of Shape Definition #2, Relative to $S$
- $Dn$: Index to First Byte of Shape Definition #n, Relative to $S$

Index

- $D1$: Index to First Byte of Shape Definition #1
- $D2$: Index to First Byte of Shape Definition #2
- $Dn$: Index to First Byte of Shape Definition #n

Shape Definitions

- $S+D1$: First Byte
- Last Byte
- $S+D2$: First Byte
- Last Byte
- $S+Dn$: First Byte
- Last Byte+00

Figure 5

Start byte 0 01

- Number of Shapes
- Index to Shape Definition #1, Relative to Start
- First Byte
- Shape Definition #1

(Store this address in $328$ and $329$)

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>04</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3F</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>2D</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>A</td>
<td>36</td>
</tr>
<tr>
<td>B</td>
<td>1E</td>
</tr>
<tr>
<td>C</td>
<td>07</td>
</tr>
<tr>
<td>D</td>
<td>00</td>
</tr>
</tbody>
</table>

- Last Byte
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:

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 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:

and then type the numbers, separated by spaces:

0800:01 00 04 00 12 3F 20 64 2D 15 36 IE 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 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 cam 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 3 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 format

BSAVE filename. A$ startingaddress, L$ tablelength

For our example, you might type

BSAVE MYSHAPE1, AS 0800. LS 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 TAIL!

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 “AS$” 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 = COLE = SHAPE = ROT = SCALE REM SET PARAMETERS
20 INIT = -12288 : DRAW —11465 REM DEFINE SUBROUTINES
30 WRITE = 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 draw, 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 I: 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**

<table>
<thead>
<tr>
<th>ROT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(no rotation from shape definition)</td>
</tr>
<tr>
<td>16</td>
<td>(90 degrees clockwise rotation)</td>
</tr>
<tr>
<td>32</td>
<td>(180 degrees clockwise rotation)</td>
</tr>
<tr>
<td>48</td>
<td>(270 degrees clockwise rotation)</td>
</tr>
</tbody>
</table>

**DRAWING SHAPES**

The following example program DRAWs shape definition number three. It's 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 DRAM SHAPE 3, DOUBLE SIZE, TURNED 135 DEGREES
60 END
LINKING SHAPES

DRAW1 is identical to DRAW, except that the last point previously DRAWn, PLOTted or POSNed determines the color and the starting point for the new shape. X0, TO. and COLE, need not be specified, as they will have no effect on DRAW1. However, some point must have been plotted before CALLing DRAW1, or this CALL will have no effect.

The following example program draws “squiggles” by DRAWing a small shape whose orientation is given by game control #0, 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.

```
10 X0 = Y0 = COLR = SHAPE = ROT = SCALE REM SET PARAMETERS
20 INIT = -12288 DRAW = -11465 DRAW1 = -11462
22 CLEAR = -12274 UNITE = 127 REM NAME SUBROUTINES AND COLOR
30 FULLSCREEN = -16302 BUTN = -16287 REM NAME LOCATIONS
40 CALL INIT REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
50 POKE FULLSCREEN, 0 REM SET FULL-SCREEN GRAPHICS
60 COLR = WHITE : SHAPE = 1 : SCALE = 5
70 X0 = 140 Y0 = 80 : REM ASSIGN PARAMETER VALUES
80 CALL CLEAR : ROT = PDL(0) : CALL DRAW : REM DRAW FIRST SHAPE
90 IF PEEK(BUTN) > 127 THEN GOTO 80 : REM PRESS BUTTON TO CLEAR SCREEN
100 R = PDL(0) : IF (R < ROT+2) AND (R > ROT+2) THEN GOTO 90 :
      REM WAIT FOR CHANGE IN GAME CONTROL
110 ROT = R : CALL DRAW1 : REM ADD TO "SQUIGGLE"
120 GOTO 90 : REM LOOK FOR ANOTHER CHANCE
```

After DRAWing 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 CALLing 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 DRAWL 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
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 II 11gb—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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Locations beyond LOMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>$05, $06</td>
</tr>
<tr>
<td>Y0</td>
<td>$0C, $0D</td>
</tr>
<tr>
<td>COLR</td>
<td>$15, $16</td>
</tr>
<tr>
<td>SHAPE</td>
<td>$1F, $20</td>
</tr>
<tr>
<td>ROT</td>
<td>$27, $28</td>
</tr>
<tr>
<td>SCALE</td>
<td>$31, $32</td>
</tr>
</tbody>
</table>
## VARIABLES USED WITHIN THE HIGH-RESOLUTION SUBROUTINES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Hexadecimal Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAPEL, SHAPER</td>
<td>1A, 1B</td>
<td>On-the-fly shape pointer</td>
</tr>
<tr>
<td>HCOLOR1</td>
<td>1C</td>
<td>On-the-fly color byte</td>
</tr>
<tr>
<td>COUNTH</td>
<td>1D</td>
<td>High—order byte of step count count for LINE.</td>
</tr>
<tr>
<td>HBASL, HBASH</td>
<td>26, 27</td>
<td>On-the-fly BASE ADDRESS</td>
</tr>
<tr>
<td>HMASK</td>
<td>30</td>
<td>On-the-fly BIT MASK</td>
</tr>
<tr>
<td>QDRNTE</td>
<td>53</td>
<td>2 LSB’s are rotation quadrant for DRAW.</td>
</tr>
<tr>
<td>X0L, X0R</td>
<td>320, 321</td>
<td>Most recent X-coordinate. Used for initial endpoint of LINE. Updated by PLOT.</td>
</tr>
<tr>
<td>Y0</td>
<td>322</td>
<td>Most recent y-coordinate (see X0L).</td>
</tr>
<tr>
<td>BXSAV</td>
<td>323</td>
<td>Saves 6502 K-register during high-resolution CALLs from BASIC.</td>
</tr>
<tr>
<td>BCOLOR</td>
<td>324</td>
<td>Color specification for PLOT. POSN.</td>
</tr>
<tr>
<td>HNDX</td>
<td>325</td>
<td>On-the-fly byte index from BASES ADDRESS</td>
</tr>
<tr>
<td>HPAG</td>
<td>326</td>
<td>Memory page for plotting graphics. Normally ~20 for plotting in Page 1 of high—resolution display memory ($2000—$3FFF)</td>
</tr>
<tr>
<td>SCALE</td>
<td>327</td>
<td>On-the-fly scale factor for DRAW</td>
</tr>
<tr>
<td>SHAPXL, SHAPXH</td>
<td>328, 329</td>
<td>Start of Shape Table pointer.</td>
</tr>
<tr>
<td>COLLSN</td>
<td>32A</td>
<td>Collision Count from DRAW, DRAWl.</td>
</tr>
</tbody>
</table>
SHAPE TABLE INFORMATION

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

### SHAPE TABLE EXAMPLE

<table>
<thead>
<tr>
<th>Start of Table</th>
<th>0-255</th>
<th>Number of Shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Address Stored in $328—$329)</td>
<td>Unused</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>First Byte</td>
</tr>
<tr>
<td>Last Byte=0</td>
<td>First Byte</td>
<td>Shape #1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shape #2</td>
</tr>
</tbody>
</table>

LOMEM $\rightarrow$ BASIC Variables $\rightarrow$ (if Table SHLOADed) $\rightarrow$ BASIC Variables

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 MYSIIAPES2 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:

```basic
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 PAEM4ETERS
```
APPLE II MEMORY MAP FOR USING HIGH-RESOLUTION GRAPHICS WITH INTEGER BASIC

Highest RAM
Memory address: >
This is 49151 ($BFFF) on a 48K system

Booting DOS
Sets HIMEM here

CALL SHLOAD
Sets LOMEM here

invoking BASIC
Sets HIMEM here

HIGH-RESOLUTION GRAPHICS PAGE 2
User’s BASIC program
Starts at HIMEM
and builds down

HIMEM’S VALUE IN
LOCATIONS 76–77
($4C–$4D)

DISK OPERATING SYSTEM
(BASIC VARIABLES)
Start at LOMEM
and build up

HIGH-RESOLUTION GRAPHICS PAGE 1

LOW-RESOLUTION GRAPHICS
AND TEXT SCREEN, ETC.

INTEGER BASIC SYSTEM USE
BASIC VARIABLES
Start
Shape Table
(if SHLOADed)

CALL SHLOAD
Sets LOMEM here

LOWEST RAM
Memory address: >
0000 ($0000)

Unfortunately, there is no convention for napping memory. This map shows the highest (largest) address at the top, lowest (smallest) address at the bottom. The naps of Shape Tables that appear on other pages show the starting address (lowest and smallest) at the top, the ending address (highest end 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 I

### Source Assembly Listings

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>High-Resolution Graphics</td>
<td>$D000-$D3FF</td>
</tr>
<tr>
<td>76</td>
<td>Renumber</td>
<td>$D400-$D4BB</td>
</tr>
<tr>
<td>79</td>
<td>Append</td>
<td>$D4BC-$D4D4</td>
</tr>
<tr>
<td>80</td>
<td>Relocate</td>
<td>$D4DC-$D52D</td>
</tr>
<tr>
<td>82</td>
<td>Tape Verify (BASIC)</td>
<td>$D535-$D553</td>
</tr>
<tr>
<td>83</td>
<td>Tape Verify (6502 Code &amp; Data)</td>
<td>$D554-$D5AA</td>
</tr>
<tr>
<td>84</td>
<td>RAM Test</td>
<td>$D5BC-$D691</td>
</tr>
<tr>
<td>87</td>
<td>Music</td>
<td>$D717-$D7F8</td>
</tr>
</tbody>
</table>
**APPLE-II HI-RESOLUTION GRAPHICS SUBROUTINES**

by WOZ 9/13/77

ALL RIGHTS RESERVED

* HI-RES EQUATES

SHAPEL EQU $1A POINTER TO SHAPE LIST
SHAPEH EQU $1B SHAPE LIST
HCOLOR1 EQU $1C RUNNING COLOR MASK
COUNTH EQU $1D
HBASL EQU $26 BASE ADR FOR CURRENT HI-RES PLOT LINE, A
HBASH EQU $27 HI-RES PLOT LINE, B
HMASK EQU $30
A1L EQU $3C MONITOR A1.
A1H EQU $3D
A2L EQU $3E MONITOR A2.
A2H EQU $3F
LOMEML EQU $4A BASIC ‘START CE VARS’.
LOMEMH EQU $4B
DXL EQU $50 DELTA-X FOR HI IN, SHAPE.
DXH EQU $51 SHAPE TEMP.
DY EQU $52 DELTA-Y FOR HLIN, SHAPE.
QDRNT EQU $53 ROT QUADRANT (SHAPE).
EL EQU $54 ERROR FOR HLIN.
EN EQU $55
PPL EQU $5A BASIC START OF PROG PTR.
PPI EQU $5B BASIC END OF VARS PTR.
PVL EQU $5C BASIC ACC.
PYH EQU $5D
ACL EQU $5E
ACH EQU $5F
X0L EQU $320 PRIOR X-COORD SAVE
X0H EQU $321 AFTER HLIN OR HPLT.
Y0 EQU $322 HLIN, HPLT Y-COORD SAVE.
BXSAV EQU $323 X-REG SAVE FOR SASIC.
HCOLOR EQU $324 COLOR FOR HPLT, HPOSN.
HNDX EQU $325 HORIZ OFFSET SAVE.
HPAG EQU $326 HI—RES PAGE ($20 NORMAL).
SCALE EQU $327 SCALE FOR SHAPE, MOVE.
SWAPXL EQU $328 START OF.
SHAPXH EQU $329 SHAPE TABLE.
COLLSN EQU $32A COLLISION COUNT
HIRES EQU $C057 SWITCH TO HI-RES VIDEO
MIXSET EQU $C053 SELECT TEXT/GRAFICS MIX
TXTCLR EQU $C050 SELECT GRAPHICS MODE.
MEMFUL EQU $E36B BASIC MEM FULL ERROR.
RINGERR EQU $EE68 BASIC RANGE ERROR.
ACADR EQU $E11E 2-BYTE TAPE READ SETUP.
RD2BIT EQU $FCFA TWO-EDGE TAPE SENSE
READ EQU $FF02 READ WITHOUT HEADER.

* HIGH RESOLUTION GRAPHICS INITS*

* RUM VERSION $D000 TO $D3FF*

ORG $D000
ODJ $A000

D000 A9 20
D002 8D 26 03

D000 A20 20
D002 8D 26 03

SETHL LDA #$20 INIT FOR $2000-3FFF
STA HPAG HI-RES SCREEN MEMORY.
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 IC     72 BKGNDOSTA 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 PACE, NORMALLY
D019 84 IA     76 STY SHAPEI. $2000-3FFF OR $4000-5FFF
D01B A5 1C     77 BKGND1 LDA HCOLOR1
D01D 85 IC     78 STA (SHAPEI) Y
D01F 20 A2 D0  79 JER 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 1B     83 LDA SHAPEH
D029 29 1F     84 AND #$1F TEST FOR DONE.
D02B D0 EE     85 BNE BKGND1
D02D 60         86 RTS

88 * HI—RES GRAPHICS POSITION AND PLOT SUBRS
D02E 8D 22 03  89 HPOSN STA Y0 ENTER WITH Y IN A-REQ
D031 8E 20 03  90 STX X0L XL IN X-REG.
D034 8C 21 03  91 STY X0H 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 04 A9     96 LSR :IN HBASL, HBASH FOR
D03E 05 26     97 ORA HBASL ACCESSING SCREEN MEM
D040 85 24     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 : HBASH = PPPFGHCD
D047 0A        103 ASL : HBASH =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 PA        107 ASL : PPP=010 FOR $4000—7FFF
D04E 66 26     108 ROR HBASL (GIVEN Y-COORD=ABCDEFGH)
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 00 05     115 BEG HPoSNS2 (QUOTIENT) AND BIT
D05E A0 23     116 LDY #$23 WITHIN SCREEN MEM BYTE
D060 69 04     117 AOC ##4 (MASK SPEC’D BY REMAINDER)
D062 C8        118 HPOSN1 INY
D063 E9 07     119 HPOSN2 SBC #$7 SUBTRACT OUT SEVENS.
D065 80 F8     120 BCS HPOSN1
D067 8C 25 03  121 STY MNDX WORKS FOR XO FROM
D06A AA        122 TAX 0 TO 279, LOW-ORDER
D06B BD EA D0  123 LDA MSKTB1-249, X BYTE IN X-REQ
D06E 85 30     124 STA HMASK HIGH IN Y-REQ 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 FDR 180 DEGREE SHIFT
D077 80 29     129 BCS CSHFT2 PRIOR TO COPYING TO HCOLOR1.
D079 60        130 RTS
D07A 20 2E D0  131 HPL0T JSR HPOSN
D07D A5 IC     132 HPL0T1 LOA HCOLOR1 CALC BIT POSN IN HBASL,H
D07F 51 24     133 EOR (HBASL),Y HNDX AND HMASK FROM
D081 04 30     134 AND HMASK Y-COORD IN A-REQ.
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 HMAS
D087 36 17     137 RTS SUBSTITUTE CORRESPONDING
D089 3E         138 BIT OF HCOLOR1.
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 B0 05 144 BCS LEFT1 7 BITS OF HMASK
D03F 49 C0 145 EOR #$C0 ONE BIT TO LSB
D091 85 30 146 LRI STA HMASK
D093 60 147 RTS
D094 88 148 LEFT1 DEY DEC R HORIZ INDEX.
D095 10 02 149 SPL LEFT2
D097 A0 27 150 LDY #$27 WRAP AROUND SCREEN
D099 A9 C0 151 LOA #$C0 NEW HMASK, RIGHTMOST
D09B 85 30 152 STA HMASK DOT OF BYTE
D0BD SC 25 03 153 STY HNDX UPDATE HORIZ INDEX
D0A0 A5 1C 154 CSHFT LDA HCOLOR1
D0A2 0A 155 CSHFT2 ASL ; ROTATE LOW-ORDER
D0A3 C9 C0 156 CMP #$C0 7 BITS OF HCOLDR1
D0A5 10 06 157 SPL RTSI ONE BIT POSN.
D0A7 A5 1C 158 LDA HCOLOR1
D0A9 49 7F 159 EOR #$7F ZXYXYXYX —> ZXYXYXY
D0A8 85 1C 160 LDA HCOLOR1
D0AD 60 161 RTS
D0AE A5 30 162 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 SIT TO MSB.
D0B5 A9 81 166 LDA #$81
D0B7 C8 167 INY NEXT BYTE.
D0B8 C0 28 168 OPY #$28
D0B4 90 DF 169 BCC NEWNDX
D0BC A0 00 170 LDY #$0 WRAP AROUND SCREEN IF > 279
D0BE B0 DB 171 BCS NEWNDX ALWAYS TAKEN.

*D.R.U.D. SUBROUTINES.

D0C0 18 173
D0C1 A5 51 174 LRUDXI CLC NO 90 DEG ROT (X-OR).
D0C3 29 04 175 LRUDX2 LOA SHAPE
D0C5 F0 27 176 AND #4 IF B2=0 THEN NO PLOT.
D0C7 A9 7F 177 BEG LRUD4
D0CB 31 26 178 AND HMASK
D0CD 00 28 179 AND (HASL),Y SCREEN BIT SET?
D0CF SE 24 03 180 INC COLLSN
D0D2 A9 7F 181 LDA SHAPEX
D0D4 25 30 182 AND HMASK
D0D6 10 12 183 BPL LRUD3 ALWAYS TAKEN.
D0D8 18 184 LRUD1 CLC NO 90 DEG ROT.
D0DA A5 51 185 LRUD2 LOA SHAPE
D0DB 29 04 186 AND #4 IF B2=0 TNSN NO PLOT.
D0DD F0 0F 187 BEQ LRUD4
D0DF B1 26 188 LDA (HASL), V
D0F6 A5 26 189 EOR HCOLOR1 SET HI-RES SCREEN BIT
D0F8 25 *30 190 AND HMASK TO CORRESPONDING HCOLOR1
D0F9 A5 26 191
D0F0 30 30 192
D0F4 09 02 193
D0F6 6A 194
D0F7 B0 8F 195
D0F9 30 30 196
D0FE 2C EA 01 197
D101 00 22 198
D103 06 24 199

HI-RES GRAPHICS L, R, U, D SUBRS
336 * HI-RES GRAPHICS COORDINATE RESTORE SUSR
337 HFIND
338 LDA HBASL.
339 D1FC A5 26
340 D1FE 0A
341 D1FF A5 27
342 D201 29 03
343 D204 05 26
344 D207 0A
345 D208 0A
346 D209 8D 22 03
347 D20C A5 27
348 D20E 4A
349 D20F 4A
350 D210 29 07
351 D212 0D 22 03
352 D215 8D 22 03
353 D218 AD 25 03
354 D21B 0A
355 D21C 6D 25 03
356 D21F 0A
357 D220 AA
358 D221 CA
359 D222 A5 30
360 D224 29 7F
361 D226 E8
362 D227 4A
363 D228 D0 FC
364 D22A 8D 21 03
365 D22D 8A
366 D22E 18
367 D22F 6D 25 03
368 D232 90 03
369 D234 EE 21 03
370 D237 8D 20 03
371 D23A 60
372 D23B 86 1A
373 D23D 84 1B
374 D23F AA
375 D240 4A
376 D241 4A
377 D242 AA
378 D243 4A
379 D244 85 53
380 D246 8A
381 D247 29 0F
382 D249 AA
383 D24A BC EB DI
384 D24B 84 50
385 D24C 49 0F
386 D24D AA
387 D24E 84 52
388 D24F 49 0F
389 D250 85 53
390 D251 AA
391 D252 BC EB DI
392 D253 C8
393 D254 84 52
394 D255 85 53
395 D256 84 52
396 D257 AC 25 03
397 D258 8E 2A 03
398 D25B 8D 20 03
399 D25D A1 1A
400 D260 A1 1A

373*HI-RES GRAPHICS SHAPE DRAW SUBR
374*SHAPE DRAW
376*R = 0 TO 63
377*SCALE FACTOR USED (1=NORMAL)
378*SHAPE DRAW
379DRAW STX SHAPEL DRAW DEFINITION
380STY SHAPEH POINTER.
381DRAWI TAX
382D240 4A LSR ; ROT ($0-$3F)
383D241 4A LSR
384D942 4A LSR ; QDRNT 0=UP, 1=RT.
385D243 4A LSR ; 2=DWN, 3=LFT.
386D244 85 53 STA QDRNT
387D246 8A TXA
388D247 29 0F AND #$F
389D249 AA TAX
390D24A BC EB DI LDY COS, X SAVE COS AND SIN
391D24D 84 50 STY DXL VALS IN DXL AND DY
392D24F 49 0F E0R
393D251 AA TAX
394D252 BC EB DI LDY CDS+1.X
395D255 C8 I NY
396D256 84 52 STY DY
397D258 AC 25 03 DRAW2 LDY HNDX BYTE INDEX FROM
398D25B A2 00 LDX #$0 HI-RES BASEADR.
399D25D 8E 2A 03 STX COLLISON CLEAR COLLISION COUNT.
400D260 A1 1A LDA (SHAPEL,X) 1ST SHAPE DEF BYTE.
D262 85 51 401 DRAWS 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,I D VECTORS.
D26A AE 27 03 405 LDX SCALE SCALE FACTOR.
D26D A5 54 406 DRAW4 LDA EL
D26F 38 407 SEC IF FRAC 0VFL.
D270 65 50 408 ADC DXL THEN MOVE IN
D272 85 54 409 STA EH 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 0VFL
D27C 65 52 414 ADC DY THEN MOVE IN
D27E 85 55 415 STA EH SPECIFIED VECTOR
D280 90 03 416 SCC DRAW6 DIRECTION +90 DEG.
D282 20 09 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 DEFIN
D28C 4A 423 LSR
D28D 00 03 424 BNE DRAW3 NOT DONE THIS BYTE.
D28F E6 1A 425 INC SNAPEL.
D291 00 02 426 BNE DRAW3 NEXT BYTE OF
D293 56 12 427 INC SHAPE 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 * EX-OR SHAPE INTO SCREEN.
434 * ROT = 0 TO 3 (QUADRANT ONLY)
435 * SCALE IS USED
436 * SCALE IS USED

438 *

D29A 86 1A 439 XDRAW STX SHAPEX SHAPE DEFINITION
D29C 84 15 440 STY SHAPEX POINTER.
D29E AA 441 TXA LSR ; ROT (S0-S3F)
D29F 4A 442 LSR
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 #$F
D2A8 AA 449 TAX
D2A9 BCF 81 450 LDY COS. X SAVE COS AND SIN
D2AC 84 50 451 STY DXL VALS IN DXL AND DY.
D2AE 49 0F 452 EOR #$F
D2B0 AA 453 TAX
D2B1 JC 8C D1 454 LDY COS+1, X
D2B4 C8 455 I NY
D2B5 84 52 456 STY DY
D2B7 AC 25 03 457 XDRAW LDY HNDX INDEX FROM HI-RES
D2BA A2 00 458 LDX #$80 BADE ADR.
D2BC 8E 2A 03 459 STX COLLSN CLEAR COLLISION DETECT
D2BF A1 1A 460 LDA (SHAPEL, X) 1ST SHAPE DEF BYTE.
D2C1 05 51 461 XDRAW3 STA SHAPEX
D2C3 A2 80 462 LDX #$580
D2C5 96 54 463 STX EC EL,EH FOR FRACTIONAL
D2C7 85 55 464 STX EH L, R,U,D, VECTORS
D2C9 AE 27 03 465 LDX SCALE SCALE FACTOR
D2CC A5 54 466 LDA EI
D2CE 38 467 SEC IF FRAC COS OVFL
D2CF 65 50 468 A0C DXL THEN MOVE IN
D2D1 85 54 469 STA EL SPECIFIED VECTOR
D2D3 90 04 470 BCC XDRAWS DIRECTION
D2D5 20 C0 D0 471 JSR LRUDX 1
D2D8 18 472 CLC
D2D9 A5 55 473 XDRAW5 LDA EH IF FRAC SIN OVFL
D2DB 65 52 474 ADC DY THEN MOVE IN
D2DD 85 55 475 STA EH SPECIFIED VECTOR
D2DF 90 03 476 BCC XDRAW6 DIRECTION +90 DEC.
D2E1 20 D9 DO 477 JSR LRIJD2
D2E4 CA 478 X0RAW6 DEX LOOP ON SCALE
D2E5 D0 E5 479 BNE XDRAW4 FACTOR.
D2E7 A5 51 480 LDA SHAPEX
D2E9 4A 481 LSR ; NEXT 3-BIT VECTOR.
D2EA 4A 482 LSR ; OF SHAPE DEF
D2EB 4A 483 LSR
D2EC DO 03 484 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 Al 1A 488 XDARW7 LDX (SHAPEL, X)
D2F6 DO 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 GETY0 Y0 FROM 8ASIC.
D302 48 496 PHA
D303 20 9A D3 497 JSR GETX0 X0 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 02 502 BPLT JMP BPOSN PLOT CALL (BASIC).
D311 4C 7D D0 503 JMP HPL0T1
D314 AD 25 03 504 BLIN1 LDA HNDX
D317 4A 505 LSR ; SET HCOLORI FROM
D318 20 9D D3 506 JSR PCOLR BASIC VAR COLR.
D31B 20 75 D0 507 JSR HPOSN3
D31E 20 9A 03 508 BLINE JSR GETX0 LINE CALL, GET X0 FROM BASIC
D321 8A 509 TXA
D322 48 510 PHA
D323 98 511 TYA
D324 AA 512 TAX
D325 20 AF D3 513 JSR GETY0 Y0 FROM BASIC
D328 A8 514 TAY
D329 68 515 PLA
D32A 20 64 D1 516 JSR HL IN
D32D AE 23 03 517 LDX BXSAV
D330 60 518 RTS
D331 20 9D D3 519 BGN D JSR PCOLR BACKGROUND CALL
D334 4C 10 D0 520 JMP BKGND0
D337 20 F9 D2 523 DORAWI JSR BPOSN
D33A 20 51 D3 524 BDRAW TSR BDRAWX DRAW CALL FROM BASIC.
D33D 20 3B D2 525 JSR DRAW
D340 AE 23 D3 526 LDX DXSAV
D343 60 527 RTS
D344 20 F9 D2
D347 20 51 D3 528 BXDRAW1 JSR BPOSN
D34A 20 9A D2 529 JSR BDRAWX EX-OR DRAW
D34D AE 23 03 530 JSR XDRAW FROM BASIC.
D350 60 531 LDX BXSAV
D351 8E 23 03 532 BXSAV SAVE FROM BASIC
D354 AO 32 533 JSR PBYTE SCALE FROM BASIC
D359 8D 27 03 534 STA SCALE
D35C A0 28 03
D35E 20 92 03
D361 48 535 PHA SAVE ON STACK.
D362 AD 28 03 540 LDA SHAPEXL
D365 85 1A 541 STA SHAPES START OF SHAPE TABLE.
D367 AD 29 03 542 LDA SHAPEH SHAPE TABLE.
D36A 85 18 543 STA SHAPEH
D36C AG 20 544 LDY #$20
D36E 20 92 03 545 JSR PBYTE SHAPE FROM BASIC.
D371 F0 39 546 BEQ RERR1
D373 A2 00 547 LDY #$0
cmp (SHAPEL, X) > NUM OF SHAPES?
D375 C1 1A 548 cmp (SHAPEL, X) > NUM OF SHAPES?
D377 F0 02 549 BEQ BDRWX1
D379 B0 31 550 BCS RERR1 YES, RANGE ERR.
D37B 0A 551 ASL
D37C 90 03 552 BCC BDRWX2
D37E E6 1B 553 INC SHAPEH
D380 18 554 CLC
D381 AB 555 BDRWX2 TAY SHAPE NO. * 2.
D382 B1 1A 556 LDA (SHAPEL, Y)
D384 65 1A 557 ADC SHAPE
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 60 29 03 561 ADC SHAPXH (X LOW, Y HI)
D38B A8 562 TAY
D38E 68 563 PLA ROT FROM STACK.
D38F 60 564 RTS

D390 A0 16 566 PCOLR LDY #$16
D392 B1 4A 567 PBYTE LDA (LOMEML, Y)
D394 D0 16 568 PBYTE LDA (LOMEML, Y)
D396 88 569 BNE RERRI GET BASIC PARAM.
D397 B1 4A 570 DEY (ERR IF >255)
D399 60 571 LDA (LOMEML, Y)
D39A 8E 23 03 572 RTSB. RTS
D39D A0 05 573 GETYO STX BXSAV SAVE FOR BASIC.
D39F B1 4A 574 LDY #5
D3A1 AA 575 LDA (LOMEML), Y X0 LOW-ORDER BYTE.
D3A2 C8 576 TAX
D3A3 B1 4A 577 INY
D3A5 A8 578 LDA (LOMEML), Y—HI—ORDER BYTE
D3A6 E0 18 579 TAY
D3A8 E9 01 580 CPX #$18
D3A9 90 ED 581 SBC #51 RANGE ERR IF >279
D34C 4C 68 EE 582 BCC RTSB
D3AF A0 0D 583 RERR1 JMP RNGERR
D3B1 20 92 D3 584 GETYO LDY #5D OFFSET TO Y0 FROM LOMEM
D3B4 C9 C0 585 JSR PBYTE GET BASIC PARAM Y0
D3B6 80 F4 586 CMP #$C0 (ERR IF >191)
D3B8 60 587 BCS RERR1
D3B9 588 RTS
SHAPE TAPE LOAD SUBROUTINE

SHLOAD STX SXSAV SAVE FOR SASIC.
JSR ACAOR READ 2-BYTE LENGTH INTO
LDA #$00 START OF SHAPE TABLE IS $0800
STA SHAPXL
CLC ADC ACL
ADC ACL TAY
LDA #$08 HIGH BYTE OF SHAPE TABLE POINTER
STA A1H
STA SHAPXL
ADC ACH
BCS MFULL1 NOT ENOUGH MEMORY.
CPY PPL PHA
SEC PPH
PLA BNE SHLOD1
ADC #$1 SHLOD1
STY LOMEML
STA LOMENH
JSR RD2BIT
LDA #$3 . 5 SECOND HEADER
JSR READX1
LDX BXSAV
RTS
MFULL1 JMP MEM FUL

TOTAL ERRORS: 00
<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>**************************************************</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>* APPLE-JI BASIC RENUMBER/ APPEND SUBROUTINES *</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>* VERSION TWO *</td>
</tr>
<tr>
<td>6</td>
<td>* RENUMBER *</td>
</tr>
<tr>
<td>7</td>
<td>* &gt;CLR *</td>
</tr>
<tr>
<td>8</td>
<td>* &gt;START= *</td>
</tr>
<tr>
<td>9</td>
<td>* &gt;STEP= *</td>
</tr>
<tr>
<td>10</td>
<td>* &gt;CALL-10531 *</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td>* OPTIONAL *</td>
</tr>
<tr>
<td>13</td>
<td>* &gt;FROM= *</td>
</tr>
<tr>
<td>14</td>
<td>* &gt;T0= *</td>
</tr>
<tr>
<td>15</td>
<td>* &gt;CALL-10521 *</td>
</tr>
<tr>
<td>16</td>
<td>*</td>
</tr>
<tr>
<td>17</td>
<td>* USE RENX ENTRY *</td>
</tr>
<tr>
<td>18</td>
<td>* FOR RENUMER ALL *</td>
</tr>
<tr>
<td>19</td>
<td>*</td>
</tr>
<tr>
<td>20</td>
<td>* WOZ APRIL 12, 1978 *</td>
</tr>
<tr>
<td>21</td>
<td>* APPLE COMPUTER INC. *</td>
</tr>
<tr>
<td>22</td>
<td>**************************************************</td>
</tr>
<tr>
<td>24</td>
<td>6502 EQUATES</td>
</tr>
<tr>
<td>26</td>
<td>* 6502 EQUATES</td>
</tr>
<tr>
<td>27</td>
<td>*</td>
</tr>
<tr>
<td>28</td>
<td>28 ROL EQU $0 LOW-ORDER SW16 RO BYTE.</td>
</tr>
<tr>
<td>29</td>
<td>29 ROH EQU $1 HI-ORDER</td>
</tr>
<tr>
<td>30</td>
<td>30 ONE EQU $01</td>
</tr>
<tr>
<td>31</td>
<td>31 R11L EQU $16 LOW-ORDER SW16 R11 BYTE.</td>
</tr>
<tr>
<td>32</td>
<td>32 R11H EQU $17 HI-ORDER.</td>
</tr>
<tr>
<td>33</td>
<td>33 HIMEM EQU $4C BASIC HIMEM POINTER.</td>
</tr>
<tr>
<td>34</td>
<td>34 PPL EQU $CA BASIC PROG POINTER</td>
</tr>
<tr>
<td>35</td>
<td>35 PVL EQU $CC BASIC VAR POINTER</td>
</tr>
<tr>
<td>36</td>
<td>36 MEMEULL EQU $E36B BASIC MEM FULL ERROR</td>
</tr>
<tr>
<td>37</td>
<td>37 PRDEC EQU $E51B BASIC DECIMAL PRINT SUBR.</td>
</tr>
<tr>
<td>38</td>
<td>38 RANGERR EQU $EE68 BASIC RANGE ERROR</td>
</tr>
<tr>
<td>39</td>
<td>39 LOAD EQU $F0DF BASIC LOAD SUBR</td>
</tr>
<tr>
<td>40</td>
<td>40 SW16 EQU $F689 SWEET 16 ENTRY</td>
</tr>
<tr>
<td>41</td>
<td>41 CROUT EQU $FDE8 CAR RET SUBR.</td>
</tr>
<tr>
<td>42</td>
<td>42 COUT EQU $FDEED CHAR OUT SUBR.</td>
</tr>
<tr>
<td>44</td>
<td>SWEET 16 EQUATES</td>
</tr>
<tr>
<td>45</td>
<td>* SWEET 16 EQUATES</td>
</tr>
<tr>
<td>46</td>
<td>*</td>
</tr>
<tr>
<td>47</td>
<td>47 ACC. EQU $0 SWEET 16 ACCUMULATOR.</td>
</tr>
<tr>
<td>48</td>
<td>48 NEULOW EQU $1 NEW INITIAL LNO.</td>
</tr>
<tr>
<td>49</td>
<td>49 NEWINCR EQU $2 NEW LNO INOR.</td>
</tr>
<tr>
<td>50</td>
<td>50 LNLOW EQU $3 LOW LNO OF RENUM RANGE.</td>
</tr>
<tr>
<td>51</td>
<td>51 LNHI EQU $4 HI LNO OF RENUM RANGE</td>
</tr>
<tr>
<td>52</td>
<td>52 TBLSTRT EQU $5 LNO TABLE START.</td>
</tr>
<tr>
<td>53</td>
<td>53 TBLNDX1 EQU $6 PASS 1 LNO TBL INDEX.</td>
</tr>
<tr>
<td>54</td>
<td>54 TBLIM EQU $7 LNO TABLE LIMIT.</td>
</tr>
<tr>
<td>55</td>
<td>55 SCRIB EQU $8 SCRATCH REG.</td>
</tr>
<tr>
<td>56</td>
<td>56 HMEM EQU $8 HIMEM (END OF PRGM).</td>
</tr>
<tr>
<td>57</td>
<td>57 SCR9 EQU $9 SCRATCH REQ.</td>
</tr>
<tr>
<td>58</td>
<td>58 PRGNDX EQU $9 PASS 1 PROC INDEX.</td>
</tr>
<tr>
<td>59</td>
<td>59 PRONDXI EQU $A ALSO PROC INDEX.</td>
</tr>
<tr>
<td>60</td>
<td>60 NEWLN EQU $B NEXT “NEW UND”.</td>
</tr>
<tr>
<td>61</td>
<td>61 NEWLN1 EQU $C PRIOR “NEW LNO” ASSIGN.</td>
</tr>
<tr>
<td>62</td>
<td>62 TBLND EQU $6 PASS 2 LNO TABLE END.</td>
</tr>
<tr>
<td>63</td>
<td>63 PRGNDX2 EQU $7 PASS 2 PROC INDEX.</td>
</tr>
<tr>
<td>64</td>
<td>64 CHRO EQU $9 ASCII “0”</td>
</tr>
<tr>
<td>65</td>
<td>65 CHRA EQU $A ASCII “A”.</td>
</tr>
</tbody>
</table>
APPLE - 11 BASIC RENUMBER SUBROUTINE - PASS 1

ORG $D400
OBJ $A400

EQU $77 CONST/LNO MODE.
EQU $78 LNO. TBL IDX FOR UPDATE
EQU $79 OLD LNO F03 UPDATE.
EQU $7A BASIC STR CON TOKEN.
EQU $7B BASIC REM TOKEN.
EQU $7C SWEET 16 REG 13 (CPR NEC).
EQU $7D BASIC THEN TOKEN
EQU $7E BASIC LIST TOKEN
EQU $7F SCRATCH REQ FOR APPEND.

APPEND - 11 BASIC RENUMBER SUBROUTINE - PASS 1

ORG $D400
OBJ $A400

,JSR SW16 OPTIONAL RANGE ENTRY.
SUB ACC
ST LNLOW SET LNLOW=0, LNHI=0.
ST LNH I
DCR LNH I
RTN
JSR SW16
SET HMEM, HIMEM
LDD @HMEM
ST HMEM
SET SCR9, PVL+2
POP D @SCR9 BASIC VAR PNT TO
ST TBLSTRT TBLSTRT AND TBLNDX1
ST TBLNDX1
LD NEWLOW COPY NEWLOW (INITIAL)
ST NEWLN TO NEWLN,
ST NEWLN1
POPD @SCR9 BASIC PROG PNTR
ST TBLIM  TO TBLIM AND PRGNDX.
ST PRGNDX
LD PRGNDX
CPR HMEM IF PRGNDX > =HMEM
BC PASS2  THEN DONE PASS 1.
ST PRGNDX1
LD TBLNDX1
INR ACC IF < TWO BYTES AVAIL IN
CPR TBLIM  LNO TABLE THEN RETURN
BC MERR  WITH "MEM FULL" MESSAGE
LD @PRGNDX1
ADD LENTH BYTE TO PROG INDEX.
ST PRGNDX
LDD @PRGNDX1 LINE 'lUMBER.
CPR LNLOW  IF < LNLOW THEN OOTO P1B
BNC P1B
CPR LNHI IF > LNHI THEN GOTO P1C
BNZ P1C
STD @TBLNDX1 ADD TO LNO TABLE.
RTN
LDA R0H **** 6502 CODE ****
LDX R0L
JSR PRDEC PRINT OLD LNO "—>" NEW LNO
LDA #$AD  (R0 R11) IN DECIMAL.
JSR COUT
LDA R11H
LDX R11L
BC P1A
BNZ P1C
ADD TO LNO TABLE.
RTN
LDA R0H **** 6502 CODE ****
LDX R0L
JSR PRDEC PRINT OLD LNO "—>" NEW LNO
LDA #$AD  (R0 R11) IN DECIMAL.
LD NEWLN
ST NEWLN1 COPY NEWLN to NEWLNI AND INCR
ADD NEWINCR UEWLN by NEWINOR
ST NEWLN
HEX .00 'NUL' (WELL SKIP NEXT INSTRUCTION)
CPR NEWLOW IF LOW LNO< NEW LOW THEN RANGE ERR
BNC PASS1
RTN PRINT "RANGE ERR" MESSAGE AND RETURN.
JMP RANGERR
RTN PRINT "MEM FULL" MESSAGE AND RETURN
JMP MEMFULL
INR NEWLN1 IF HI LNO <= MOST RECENT HEWLN THEN
CRR NEWLN1 RANGE ERROR.
BNC RERR

APPLE || BASIC RENUMBER / APPEND SUBROUTINE -- PASS 2

SET CHRO, $00B0 ASCII "0"
SET CHRA, $00C0 ASCII "A"
LD PRGNDX2
CPR HMEM IP PROG INDEX = HIMEN THEN DONE PASS 2.
BC DONE
INRPRONDX2 SKIP LENIN BYTE
LDD @PRGNDX2 LINE NUMBER
ST OLDLN SAVE OLD LUD.
LD TBLSTRT
ST TBLNDX2 INIT LNO TABLE INDEX
LD NEWLOW INIT NEWLN TO NEWLOW
HEX1C (WILL SKIP NEXT INSTR)
LD NEWLN1
AD0 NEWINCR ADD INCR TO NEWLN1.
ST NFWLN1
LD TBLNDX2 IF LNO TBL lDX = TBLND THEN DONE
SUB TELND SCANNING LNO TABLE
BC UD3
LDD @TBLNDX2NEXT LNO FROM TABLE.
BCR PASS1
LD@PRGNDX2BASIC TOKEN.2
CPR CHRO BNC CHKTOK CHECK TOKEN FOR SPECIAL.
CPR CHRA IF >= "0" AND < "A" THEN UPDATE.
DCR MODE IF MODE = 0 THEN UPDATE LNO REF.
BM SKPASC OR NAME.
DCR PRGNDX2
LDD @PRGNDX2 REPLACE OLD LNO WITH CORRESPONDING
SUB TELND SCANNING LNO TABLE
LD NEWLN1 NEW LNO.
LD NEWLN1 NEW LINE.
STD @PRGNDX2
SET STRCON, #$028 STR CON TOKEN.
HEX IC (SKIP-S NEXT TWO INSTRUCTIONS)
LDD @TBLNDX2NEXT LNO FROM TABLE.
SUB OLDLN LOOP TO UD2 IF NOT SAME AS OLDLN.
BNZ UD2
POPD @PRGNDX2 REPLACE OLD LNO WITH CORRESPONDING
LD NEWLN1 NEW LINE.
ST @PRGNDX2BASIC TOKEN.2
CPR CHRO
BNC CHKTOK CHECK TOKEN FOR SPECIAL.
CPR CHRA IF >= "0" AND < "A" THEN SKIP CONST
BNC GOTTCON OR UPDATE.
LD @PRGNDX2 REPLACE ALL. NEG. BYTES OF STR CON, REM,
D49A DB 191 CHKTOK CPR STRCON SW CON TOKEN?
D49B 06 F7 192 BZ SKPASC YES, SKIP SUBSEQUENT BYTES.
D49D 1C 5D 00 193 SET REM, $0050
D4A0 DC 194 CPR REM REM TOKEN?
D4A1 06F1 195 BZ SKPASC YES, SKIP SUBSEQUENT LINE
D4A3 08 13 196 BMI CONTST GOSUB, LOOK FOR LINE NUMBER.
D4A5 FD 197 DCR R13
D4A6 FD 198 DCR R13 (TOKEN $5F IS GOTO)
D4A7 06 0F 199 BZ CONTST
D4A9 10 24 00 200 SET THEN, $0024
04AC DD 201 CR9 THEN
D4AD 06 09 202 BZ CONTST 'THEN' LNO, LOOK FOR LNO.
D4AF F0 203 DCR ACC
D4B0 06 116 204 BZ P2A E0L (TOKEN 01)?
D4B2 10 74 00 205 SET LIST, $0074
D4B5 20 206 SUB LIST SET MODEIF LIST OR LIST COMMA.
D4B6 09 01 207 BNM1 CONTS2 (TOKENS $74, $75)
D4B8 20 208 CONTST SUB ACC CLEAR MODE FOR LNO
D4B9 3C 209 CONTS2 ST MODE UPDATE CHECK
D4BA 01 210 BR ITEM

212 *
213 *
214 * APPLE | BASIC APPEND SUBROUTINE
215 *
D4BC 20 89 F6 216 APPEND JSR SW1 6
D4BF 1C 4E 00 217 SET SCRC, HIMEM+2
D4C2 CC 218 POPD @SCRC SAVE HIMEM.
D4C0 88 219 ST HMEM
D4C4 19 CA00 220 SET SCR9, PPL
D4C7 69 221 LJ0D @SCR9
D4C8 7C 222 ST D @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
D4.00CC 226 POPOD @SCRC RESTORE HIMEM TO SHOW BOTH PROGRAMS
D402 28 227 LD HMEM (OLD AND NEW)
D402 7C 228 STD RETURN
D403 00 229 DONE RTN
D404 60 230 RTH

--- END ASSEMBLY ---

TOTAL ERRORS: 00
6502 RELOCATION SUBROUTINE

1. DEFINE BLOCKS

1. DEFINE BLOCKS

2. FIRST SEGMENT

3. SUBSEQUENT SEGMENTS

RELOCATION SUBROUTINE EQUATES

- ROL.EQU $02 SWEET 16 REG 1.
- INST EQU $0B 3-BYTE INST FIELD.
- LENGTH EQU $2F LENGTH CODE
- YSAV EQU $34 CMND BUF POINTER
- A1L EQU $3C APPLE-II MON PARAM AREA.
- A4L EQU $42 APPLE-II MON PARAM REG 4
- IN EQU $0200
- SW16 EQU $F689 ;SWEET 16 ENTRY
- INSDS2 EQU $F88E DISASSEMBLER ENTRY
- NXTA4 EQU $FCB4 POINTER INCR SUBR
- FRMBEG EQU $01 SOURCE BLOCK BEGIN
- FRMEND EQU $02 SOURCE BLOCK END
- TOBEG EQU $04 DEST BLOCK BEGIN
- ADR EQU $06 ADR PART OF INST.
ORG $D4DC

RELOC LDY YSAV CMND BUF POINTER

LDA IN, Y NEXT CMD CHAR

CMP #$AA '4'?

BNE RELOC2 NO, RELOC CODE SEQ.

INC YSAV ADVANCE POINTER

LDX .#$07

INIT LDA A1L, X MOVE BLOCK PARAMS

DEX AREA TO SW16 AREA

BPL INIT R1=SOURCE BEG, R2=

RTS SOURCE END, R4=DEST BEG.

RELOC2 LDY #$02

GETINS LDA (A1L), Y COPY 3 BYTES TO

STA INST, Y SW16 AREA

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.

DEY

BPL GETINS

JSR INSDS2 CALCULATE LENGTH OF

LDX LENGTH INST FROM OPCODE.

INSDS2 MOVE BLOCK PARAMS

STA A1L, X FROM APPLE-II MON

FROM AREA TO SW16 AREA

RTS SOURCE END, R4=DEST BEG.
TAPE VERIFY EGUATES

CHKSOM EQU $2E
A1 EQU $3C
HIMEM EQU $4C ; BASIC HIMEM POINTER
PP EQU $CA ; BASIC BEGIN OF PROGRAM
PRLEN EQU $CE ; BASIC PROGRAM LENGTH
XSAVE EQU $D8 ; PRESERVE X-REG FOR BASIC
HDRSET EQU $F11E ; SETS TAPE POINTERS TO $CE.CF
PRGSET EQU $F12C ; SETS TAPE POINTERS FOR PROGRAM
NXTA1 EQU $FCBA ; INCREMENTS (A1) AND COMPARES TO (A2)
HEADR EQU $FCC9
RDBYTE EQU $FCEC
RD2BIT EQU $FCFA
RDBIT EQU $FCFD
PRA1 EQU $F092 ; PRINT (A1)-
PRBYTE EQU $FDDA
COUT EQU $FDED
FINISH EQU $FF26 ; CHECK CHECKSUM, RING SELL
PRERR EQU $FF2D

TAPE VERIFY ROUTINE

ORG $D535
OBJ $A535

0R G $D535
0BJ $A535

STX XSAVE ; PRESERVE X-REG FOR BASIC

LDX #$FF
GET LEN LDA HIMEM+1 CALCULATE PROGRAM LENGTH
SBC PP+1,X INTO PRLEN
INX
BEQ GETLEN

JSR HDRSET ; SET UP POINTERS
JSR TAPEVFY ; DO A VERIFY ON HEADER
LDX #$01 ; PREPARE FOR PRGSET
JSR PRGSET SET POINTERS FOR PROGRAM VERIFY
LDX XSAVE ; RESTORE X-REG
RTS
53 *
54 * TAPE VERIFY RAM IMAGE (A1,A2)
55 *
D554 20 FA FC 56 TAPEVPY JBR 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 VRPY2 LDY #$24
D563 20 PD PC 62 JSR RDBIT
D566 20 P9 63 BCS VRPY2 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 VRPY3 JSR RDBYTE READ A BYTE
D570 F0 0E 67 SEQ 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 #$34 ONE LESS THAN USED IN READ FOR EXTRA 12
D57B 90 F0 72 BCC VRPY3 ;LOOP UNTIL A1>A2
D57D 4C 26 FF 73 JMP FINISH ;VERIFY CHECKSUM SCRIRNG BELL
D580 EA 74 EXTDEL NOP ;EXTRA DELAY TO EQUALIZE TIMING
D581 EA 75 NOP ; (+12 USEC )
D582 EA 76 NOP
D583 C1 3C 77 CMP (ALX) BYTE THE SAME?
D585 F0 EB 78 BEQ VFYLOOP IT MATCHES, LOOP BACK
D587 48 79 PHA ;SAVE WRONG BYTE FROM TAPE
D598 20 2D FF 80 JSR PRERR ;PRINT “ERR”
D59B 20 92 FD 81 JSR PRA1 ;OUTPUT (A1)“-”
D59E B1 3C 82 LDA (A1),Y
D59F 20 DA FD 83 JSR PRBYTE OUTPUT CONTENTS OP A1
D59F A9 A0 84 LDA #$A0 PRINT A BLANK
D595 20 ED FD 85 JSR COUT
D598 A9 A8 86 LDA #$A8 ; ‘(’
D59A 20 ED FD 87 JSR COUT
D59D 6B 88 PLA ;OUTPUT BAD BYTE FROM TAPE
D59E 20 DA FD 89 JSR PRBYTE
D5A1 A9 A9 90 LDA #$A9 ; ‘)’
D5A3 20 ED FD 91 JSR COUT
D5A6 A9 FD 92 LDA #$8D;CARRIAGE RETURN, AND RETURN TO CALLER
D5A8 4C ED FD 93 JMP COUT

- - - END ASSEMBLY - - -

TOTAL ERRORS: 00

83
EQUATES:

DATA EQU $0  TEST DATA $00 OR $FF
NDATA EQU $1  INVERSE TEST DATA.
TESTD EQU $2  GALLOP DATA
R3L EQU $6  AUX ADR POINTER
R3H EQU $7  AUX ADR POINTER.
R4L EQU $8  AUX ADR POINTER.
R4H EQU $9  AUX ADR POINTER.
R5L EQU $A  AUX ADR POINTER.
R5H EQU $D  GALLOP BIT MASK.
R6L EQU $C  GALLOP BIT MASK.
R6H EQU $D  ($0001 TO 2^N)
YSAV EQU $34  MONITOR SCAN INDEX.
A1H EQU $3D  BEGIN TEST BLOCK ADR.
A2L EQU $3E  LEN (PAGES) FROM MON.
SETCTLY EQU $D5B0  SET UP CNTRL - Y LOCATION
PRBYTE EQU $FDDA  BYTE PRINT SUSR.
COOT EQU $FDED  CI-FAR OUT SUEBR
PRERR EQU $FF2D  PRINTS 'ERR - BELL'
BELL EQU $FF3A
36 * RAMTEST
37 *
38 *
39 ORG $D5BC
40 OBJ $A5BC

D5BC A9 C3 41 SETUP LDA #$C3 ;SET UP CNTRL-V LOCATION
D5BE A0 D5 42 LDY #$D5
D5C0 4C B0 D5 43 JMP SETCTRLY
D5C3 A9 00 44 RAMTST LDA 00 #$0 TEST FOR $00.
D5C5 20 D0 05 45 JSR TEST
D508 A9 FF 46 LDA #$FF THEN #$FF.
D50A 20 D0 D5 47 JSR TEST
D50C 4C 3A FF 48 JMP BELL
D50E 85 00 49 TEST STA DATA
D50E 85 FF 50 EOR #$FF
D50F 85 01 51 STA NDATA
D50F 85 03 52 LDA A1H
D50F 85 07 53 STA R3H INIT (R3L, R3H)
D50F 85 09 54 STA R4H (R4L, R4H), (R5L, R5H)
D50F 85 0B 55 STA A4H TO TEST BLOCK BEGIN
D50F 85 0D 56 LDA #50 ADDRESS.
D55C A0 00 57 STY R3L
D55E 84 06 58 STY R3L
D55E 84 08 59 STY R5L
D55E 84 0A 60 LDX A2L LENGTH (PAGES).
D55E 84 3E 61 LDA DATA
D55F 91 08 62 TEST01 STA (R4L), Y SET ENTIRE TEST
D55E C8 63 INY BLOCK TO DATA.
D560 D0 FB 64 BNE TEST01
D560 E6 09 65 INC R4H
D562 CA 66 DEX
D562 D0 F6 67 BNE TEST01
D563 A6 3E 68 LDX A2L
D563 B1 06 69 TEST02 LDA (R3L), Y VERIFY ENTIRE
D563 C5 00 70 CMP DATA TEST BLOCK.
D564 F0 13 71 BEQ TEST03
D565 48 72 PHA PRESERVE BAD DATA.
D566 A5 07 73 LDA R3H
D566 FD 74 JSR PRBYTE PRINT ADDRESS,
D568 98 75 TYA
D568 20 8A D6 76 JSR PRBYSP
D569 A5 00 77 LDA DATA THEN EXPECTED DATA,
D569 8A D6 78 JSR PRBYSP
D569 68 79 PLA THEN BAD DATA,
D56A 20 7F D6 80 JSR PRBYCR THEN ‘ERR-BELL’.
D56A C8 81 TEST03 INY
D56B D0 E4 82 BNE TEST02
D56C E6 07 83 INC R3H
D56D CA 84 DEX
D56E D0 DF 85 BNE TEST02
D56F A6 3E 86 LDX A2L LENGTH.
D571 A5 01 87 TEST04 LDA NDATA
D572 91 0A 88 STA (R5L), Y SET TEST CELL TO
D572 84 0D 89 STY R6H NDATA AND R6
D572 64 0C 90 STY R6L (GALLOP BIT MASK)
D572 E6 0C 91 INC R6L TO $0001.
D573 A5 01 92 TEST05 LDA NDATA
D575 20 45 D6 93 JSR TEST6 GALLOP WITH NDATA
D576 A5 00 94 LDA DATA
D577 20 45 D6 95 JSR TEST6 THEN WITH DATA.
D578 06 0C 96 ASL R6L
D579 26 0D 97 ROL R6H SHIFT GALLOP BIT
D57A 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  IPNC 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 RTSI RTS
D645 85 02  109 TEST 6 STA TESTD SAVE GALLOP DATA.
D647 A5 0A  110  LDA R5L
D649 45 0C  111  EOR R6L SETR4 TO R5
D64B 85 08  112  STA R4L EX - OR R6
D64D A5 0B  113  LDA R5N FOR NEIGHBOR
D64F 45 0D  114  EUR R6H ADDRESS (1 BIT
D651 85 09  115  STA R4H DIFFERENCE)
D653 A5 02  116  LDA TESTD
D655 91 08  117  STA (R4L) Y GALLOP TEST. DATA.
D657 B1 0A  118  LDA (R5L),Y CHECK TEST CELL.
D659 C5 01  119  CMP NDATA FOR CHANGE.
D65B F0 E7  120  BEG RTSI (OK).
D65D 48  121  PHA PRESERVE FAIL DATA.
D65E A5 0B  122  LDA R5N
D660 20 0A FD 123  JSR PRBYTE PRINT TEST CELL
D663 A5 0A  124  LDA R5L ADDRESS.
D665 20 8A D6 125  JSR PRBYS
D668 A5 01  126  LDA NDATA
D66A 91 0A  127  STA (R5L),Y (REPLACE CORRECT DATA)
D66C 20 8A D6 128  JSR PRBYS THEN TEST DATA BYTE.
D66F 68  129  PLA
D670 20 8A D6 130  JSR PRBYS THEN FAIL DATA.
D673 A5 09  131  LDA R4H
D675 20 DA FD 132  JSR PRBYTE
D678 A5 08  133  LDA R4L THEN NEIGHBOR ADR,
D67A 20 8A D6 134  JSR PRBYS
D67D A5 02  135  LDA TESTD THEN GALLOP DATA.
D67F 20 8A D6 136  JSR PRBYS 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
D69A 20 DA FD 140  JSR PRBYTE
D63D A9 A0  141  LDA #SA0 OUTPUT BYTE. THEN
D65F 4C ED FD 142  JMP COUT SPACE.
D657 84 143  ORG $3F8
03F8 4C C3 D5  144 USRLOC JMP RAMTST ENTRY PROM MON (CTRL—Y)

--- END ASSEMBLY ---

TOTAL ERRORS: 00

86
*********************************
4  * MUSIC SUBROUTINE
5  * GARY J. SHANNON
6  *
7  *********************************
8  ORG$D717
9  *
10  * ZERO PAGE WORK AREAS
11  * PARAMETER PASSING AREAS
12  *
13  DOWNTIME EQU $0
14  UPTIME    EQU   $1
15  LENGTH   EQU   $2
16  VOICE    EQU  $2FD
17  LONG    EQU  $2FE
18  NOTE    EQU  $2FF
19  SPEAKER    EQU   $C030
20  ENTRY   JMP     LOOKUP
21  *
22  * PLAY ONE NOTE
23  *
24  * DUTY CYCLE DATA IN ‘UPTIME’ AND
25  * ‘DOWNTIME’. DURATION IN LENGTH'
26  *
27  * CYCLE IS DIVIDED INTO ‘UP’ HALF
28  * AND ‘DOWN’ HALF
29  *
30  D717 4C 4E D7
31  *
32  33 PLAY    LDY UPTIME ; GET POSITIVE PULSE WIDTH
33  34       LDA SPEAKER ; TOGGLE SPEAKER
34  35 PLAY2 INC LENGTH ; DURATION
35  36       BNE PATH1 ; NOT EXPIRED
36  37       INC LENGTH=1
37  38       BNE PATH2
38  39       RTS ; DURATION EXPIRED
39  40 PATH1 NOP ; DUMMY
40  41       JMP PATH2 ; TIME ADJUSTMENTS
41  42 PATH2 DEY ; DECREMENT WIDTH
42  43       BEG DOWN ; WIDTH EXPIRED
43  44       JMP PATH3 ; IF NOT, USE UP
44  45 *
46  D7F D0 05
47  48 PATH3 BNE PLAY2 ; SAME # CYCLES
48  49 DOWN   LDY DOWNTIME ; GET NEGATIVE PULSE WIDTH
49  50       LDA SPEAKER ; TOGGLE SPEAKER
50  51 PLAY3 INC LENGTH ; DURATION
51  52       BNE PATH4 ; NOT EXPIRED
52  53       INC LENGTH+1
53  54       BNE PATH5
54  55       RTS ; DURATION EXPIRED
55  56 PATH4 NOP ; DUMMY
56  57       JMP PATH5 ; TIME ADJUSTMENTS
57  58 PATH5 DEY ; DECREMENT WIDTH
58  59       BEQ PLAY ; BACK TO UP-SIDE
59  60       JMP PATH6 ; USE UP SOME CYCLES
60  61 PATH6 BNE PLAY3 ; REPEAT
61
62 * 63 * NOTE TASLE L00~SUP SUDROUTINE
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 71LOOKUP LDA NOTE GET NOTE NUMOER
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
D752 4A 77SHIFT LSR
D75C F0 04 78 BEQ DONE ; SHIFT WIDTH COUNT
D75E 46 00 79 LSR DOWNTIME ; ACCORDING TO VOICE
D760 D0 P9 90 BNE SHIFT
D762 B9 96 D7 81DONE 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
D762 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’ SUDROUTINE’ PLAYS NOTE #0
99* SILENTLY, FOR SAME DURATION AS
100* A REGULAR NOTE
101*

D782 EA
D783 EA 102REST NOP ; DUMMY
D784 4C 87 07 103 NOP ; CYCLE USERS
D787 E6 02 104 JMP REST2 ; TO ADJUST TIME
D789 D0 05 105REST2 . INC LENGTH
D788 E6 03 106 BNE REST3
D780 D0 05 107 INC LENGTH+ 1
D78F 60 108 BNE REST4
D790 EA 109 RTS ; IF DURATION EXPIRED
D791 4C 94 D7 110RESTS NOP ; USE UP ‘INC’ CYCLES
D794 D0 EC 111 JMP REST4
112REST4 BNE REST ; ALWAYS TAKEN

88
<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>NOTE TABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>D796</td>
<td>00 00 F6</td>
<td>HEX 00, 00, F6, F6, E8, E8, DB, DB</td>
</tr>
<tr>
<td>D79E</td>
<td>CF CF C3</td>
<td>HEX CF, CF, C3, C3, B8, B8, AE, AE</td>
</tr>
<tr>
<td>D7A6</td>
<td>A4 A4 9B</td>
<td>HEX A4, A4, 9B, 9B, 92, 92, 8A, 8A</td>
</tr>
<tr>
<td>D7AE</td>
<td>82 82 7B</td>
<td>HEX 82, 82, 7B, 7B, 74, 74, 6D, 6E</td>
</tr>
<tr>
<td>D7B6</td>
<td>67 68 61</td>
<td>HEX 67, 68, 61, 62, 5C, 5C, 57, 57</td>
</tr>
<tr>
<td>D7BE</td>
<td>52 52 4D</td>
<td>HEX 52, 52, 4D, 4E, 49, 49, 45, 45</td>
</tr>
<tr>
<td>D7CE</td>
<td>33 34 30</td>
<td>HEX 33, 34, 30, 31, 2E, 2E, 2B, 2C</td>
</tr>
<tr>
<td>D7D6</td>
<td>29 29 26</td>
<td>HEX 29, 29, 26, 27, 24, 25, 22, 23</td>
</tr>
<tr>
<td>D7DE</td>
<td>20 21 1E</td>
<td>HEX 20, 21, 1E, 1F, 1D, 1D, 1B, 1C</td>
</tr>
<tr>
<td>D7E6</td>
<td>1A 1A 1B</td>
<td>HEX 1A, 1A, 1B, 1B, 1A, 18, 19, 17, 17, 15, 16</td>
</tr>
<tr>
<td>D7EE</td>
<td>14 15 13</td>
<td>HEX 14, 15, 13, 14, 12, 12, 11, 11</td>
</tr>
<tr>
<td>D7F6</td>
<td>10 10 0F</td>
<td>HEX 10, 10, 0F, 10, 0E, 0F</td>
</tr>
</tbody>
</table>

- - - END ASSEMBLY - - -

TOTAL ERRORS: 00
APPENDIX II

SUMMARY OF PROGRAMMER'S AID COMMANDS

92  Renumber
92  Append
92  Tape Verify (BASIC)
93  Tape Verify (Machine Code & Data)
93  Relocate (Machine Code & 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 show 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

(4) 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 show 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 \( X_0 = Y_0 = \text{COLR} \)
2 \( \text{SHAPE} = \text{ROT} = \text{SCALE} \) \hspace{1cm} \text{(if shapes are used)}

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

10 \( \text{INIT} = -12288 \) \hspace{0.5cm} \( \text{CLEAR} = -12274 \) \hspace{0.5cm} \( \text{BKGND} = -11471 \)
11 \( \text{POSN} = -11527 \) \hspace{0.5cm} \( \text{PLOT} = -11506 \) \hspace{0.5cm} \( \text{LINE} = -11500 \)
12 \( \text{DRAW} = -11465 \) \hspace{0.5cm} \( \text{DRAW1} = -11462 \)
13 \( \text{FIND} = -11780 \) \hspace{0.5cm} \( \text{SHLOAD} = -11335 \)

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

20 \( \text{BLACK} = 0 \) \hspace{0.5cm} \( \text{LET GREEN} = 42 \) \hspace{0.5cm} \( \text{VIOLET} = 85 \)
21 \( \text{WHITE} = 127 \) \hspace{0.5cm} \( \text{ORANGE} = 170 \) \hspace{0.5cm} \( \text{BLJJE} = 213 \)
22 \( \text{BLACK2} = 128 \) \hspace{0.5cm} \( \text{WHITE2} = 255 \)

(d) Initialize:

30 CALL INIT

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

Example:

40 \( \text{COLR} = \text{VIOLET} \) \hspace{0.5cm} \( \text{CALL BKCND} \) \hspace{0.5cm} \( \text{REM} \) \hspace{0.5cm} \text{TURN BACKGROUND VIOLET} \)
50 \( \text{FOR I = 0 TO 279 STEP 5} \)
60 \( \text{X0 = 140} \) \hspace{0.5cm} \( \text{Y0 = 150} \) \hspace{0.5cm} \( \text{COLR = WHITE} \) \hspace{0.5cm} \text{REM SET PARAMETERS} \)
70 \( \text{CALL POQN} \) \hspace{0.5cm} \text{REM MARK THE 'CENTER'} \)
80 \( \text{X0 = 1} \) \hspace{0.5cm} \( \text{Y0 = 0} \) \hspace{0.5cm} \text{REM SET NEW PARAMETERS} \)
90 \( \text{CALL LINE} \) \hspace{0.5cm} \text{REM DRAW LINE TO EDGE} \)
100 \( \text{NEXT I} \) \hspace{0.5cm} \text{END}
QUICK REFERENCE TO HIGH-RESOLUTION INFORMATION

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>CALLing Address</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>—12288</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>—12274</td>
<td></td>
</tr>
<tr>
<td>BKGND</td>
<td>—11471</td>
<td>COLR</td>
</tr>
<tr>
<td>POSN</td>
<td>—11527</td>
<td>X0, Y0, COLR X0,</td>
</tr>
<tr>
<td>PLOT</td>
<td>—11506</td>
<td>Y0, COLR</td>
</tr>
<tr>
<td>LINE</td>
<td>—11300</td>
<td>X0, Y0, COLR</td>
</tr>
<tr>
<td>DRAW</td>
<td>—11463</td>
<td>X0, Y0, COLR, SHAPE, ROT, SCALE</td>
</tr>
<tr>
<td>DRAW1</td>
<td>—11462</td>
<td>SHAPE, ROT, SCALE</td>
</tr>
<tr>
<td>FIND</td>
<td>—11780</td>
<td></td>
</tr>
<tr>
<td>SHLOAD</td>
<td>—11335</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color Name</th>
<th>COLR Value</th>
<th>Color Name</th>
<th>COLR Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK</td>
<td>0</td>
<td>BLACK2</td>
<td>128</td>
</tr>
<tr>
<td>GREEN</td>
<td>42</td>
<td>ORANGE</td>
<td>170</td>
</tr>
<tr>
<td>VIOLET</td>
<td>85</td>
<td>BLUE</td>
<td>213</td>
</tr>
<tr>
<td>WHITE</td>
<td>127</td>
<td>WHITE2</td>
<td>255</td>
</tr>
</tbody>
</table>

(Note: on systems below S/N 6000, colors in the second column appear identical to those in the first column)

CHANGING THE High-Resolution GRAPHICS DISPLAY

Full—Screen Graphics
Mixed Graphics—Plus—Text (Default)
Page 2 Display
Page 1 Display (Normal)
Page 2 Plotting
Page 1 Plotting (Default)

(POKE —16302, 0
POKE —16301, 0
POKE —16299, 0
POKE —16300, 0
POKE 806, 64
POKE 806, 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

(PEC (810))

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