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
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
   55  Assigning parameter values: SHAPE, SCALE and ROTation
   57  DRAWing shapes
   58  Linking shapes: DRAW1
   59  Collisions

60  Part F: Technical information
   60  Locations of the High-Resolution parameters
   61  Variables used within the High-Resolution subroutines
   62  Shape Table information
   63  Integer BASIC memory map

64  Part G: Comments
### SOURCE ASSEMBLY LISTINGS

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Bytes</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>
APPENDIX

SUMMARY OF PROGRAMMER’S AID COMMANDS

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

FEATURES OF PROGRAMMER’S AID #1

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

Chapter 1. Renumber an entire Integer BASIC program, or a portion of the program.

Chapter 2. Load an Integer BASIC program from tape without erasing the Integer BASIC program that was already in memory, in order to combine the two programs.

Chapter 3. Verify that an Integer BASIC program has been saved correctly on tape, before the program is deleted from APPLE’s memory.

Chapter 4. Verify that a machine-language program or data area has been saved correctly on tape from the Monitor.

Chapter 5. Relocate 6502 machine-language programs.

Chapter 6. Test the memory of the APPLE.

Chapter 7. Generate musical notes of variable duration over four chromatic octaves, in five (slightly) different timbres, from Integer BASIC.

Chapter 8. Do convenient High-Resolution graphics from Integer BASIC.

Note: if your APPLE has the firmware APPLESOFT card installed, its switch must be down (in the Integer BASIC position) for Programmer’s Aid #1 to operate.
HOW TO INSTALL THE PROGRAMMER’S AID ROM

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

1. Turn off the power switch on the back of the APPLE II. This is important to prevent damage to the computer.

2. Remove the cover from the APPLE II. This is done by pulling up on the cover at the rear edge until the two corner fasteners pop apart. Do not continue to lift the rear edge, but slide cover backward until it comes free.

3. Inside the APPLE, toward the right center of the main printed-circuit board, locate the large empty socket in Row F, marked "ROM-D0".

4. Make sure that the Programmer's Aid ROM IC is oriented correctly. The small semicircular notch should be toward the keyboard. The Programmer's Aid ROM IC must match the orientation of the other ROM ICs that are already installed in that row.

5. Align all the pins on the Programmer's Aid ROM IC with the holes in socket D0, and gently press the IC into place. If a pin bends, remove the IC from its socket using an "IC puller" (or, less optimally, by prying up gently with a screwdriver). Do not attempt to pull the socket off the board. Straighten any bent pins with a needlenose pliers, and press the IC into its socket again, even more carefully.

6. Replace the cover of the APPLE, remembering to start by sliding the front edge of the cover into position. Press down on the two rear corners until they pop into place.

7. Programmer's Aid #1 is installed; the APPLE II may now be turned on.
2  Renumbering an entire BASIC program
2  Renumbering a portion of a BASIC program
4  Comments
RENUMBERING AN ENTIRE BASIC PROGRAM

After loading your program into the APPLE, type the

CLR

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

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

START = 1000

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

STEP = 10

Finally, type the this command:

CALL -10531

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

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

RENUMBERING PORTIONS OF A PROGRAM

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

CLR

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

START = 200
STEP = 20

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

FROM = 300
TO = 500

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

CALL -10521

If the program was previously numbered

100
120
300
310
402
500
2000
2022

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

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

and the new program line numbers will be

100
120
200
220
240
260
2000
2022
You cannot renumber in such a way that the renumbered lines would replace, be inserted between or be intermixed with un-renumbered lines. Thus, you cannot change the order of the program lines. If you try, the message

*** RANGE ERR

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

**COMMENTS:**

1. If you do not CLR before renumbering, unexpected line numbers may result. It may or may not be possible to renumber the program again and save your work.

2. If you omit the START or STEP values, the computer will choose them unpredictably. This may result in loss of the program.

3. If an arithmetic expression or variable is used in a GOTO or GOSUB, that GOTO or GOSUB will generally not be renumbered correctly. For example, GOTO TEST or GOSUB 10+20 will not be renumbered correctly.

4. Nonsense values for STEP, such as 0 or a negative number, can render your program unusable. A negative START value can renumber your program with line numbers above 32767, for what it's worth. Such line numbers are difficult to deal with. For example, an attempt to LIST one of them will result in a >32767 error. Line numbers greater than 32767 can be corrected by renumbering the entire program to lower line numbers.

5. The display of line number changes can appear correct even though the line numbers themselves have not been changed correctly. After the *** RANGE ERR message, for instance, the line numbers are left with their original numbering. LIST your program and check it before using it.

6. The Renumber feature applies only to Integer BASIC programs.

7. Occasionally, what seems to be a "reasonable" renumbering does not work. Try the renumbering again, with a different START and STEP value.
CHAPTER 2

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 SAVEEd on tape and the time the tape is verified, will cause the verification to fail.
Verifying a portion of memory saved on tape

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

`address1 . address2 W return`

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

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

`D52EG return`

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

`address1 . address2 ctrl Y return`

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

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

**COMMENTS:**

Any change in the specified memory area, however slight, between the time the program is saved on tape and the time the tape is verified, will cause the verification to fail.
CHAPTER 5
RELOCATE

12 Part A: Theory of operation
12 Relocating machine-language code
13 Program model
14 Blocks and Segments
15 Code and Data Segments
16 How to use the Code-Relocation feature

18 Part B: Examples
18 Example 1. Straightforward relocation
19 Example 2. Index into Block
20 Example 3. Immediate address reference
20 Example 4. Unusable Block ranges
21 Example 5. Changing the page zero variable allocation
22 Example 6. Split Blocks with cross-referencing
23 Example 7. Code deletion
24 Example 8. Relocating the APPLE II Monitor ($P800-$FFFF) to run in RAM ($8000-$FFFF)

25 Part C: Further details
25 Technical information
26 Algorithm used by the Code-Relocation feature
27 Comments
PART A: THEORY OF OPERATION

RELOCATING MACHINE-LANGUAGE CODE

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

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

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

The following situations call for program relocation:

1. Two different programs, which were originally written to run in identical memory locations, must now reside and run in memory concurrently.

2. A program currently runs from ROM. In order to modify its operation experimentally, a version must be generated which runs from a different set of addresses in RAM.

3. A program currently running in RAM must be converted to run from EPROM or ROM addresses.

4. A program currently running on a 16K machine must be relocated in order to run on a 4K machine. Furthermore, the relocation may have to be performed on the smaller machine.

5. Because of memory-mapping differences, a program that ran on an APPLE I (or other 6502-based computer) falls into unusable address space on an APPLE II.

6. Because different operating systems assign variables differently, either page-zero or non-page-zero variable allocation for a specific program may have to modified when moving the program from one make of computer to another.
7. A program, which exists as several chunks strewn about memory, must be combined in a single, contiguous block.

8. A program has outgrown the available memory space and must be relocated to a larger, "free" memory space.

9. A program insertion or deletion requires a portion of the program to move a few bytes up or down.

10. On a whim, the user wishes to move a program.

**PROGRAM MODEL**

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

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

A more versatile program model is as follows. A program or section of a program written to run in a memory range termed the "Source Block" actually resides currently in a range termed the "Source Segments". Thus a program written to run from location $400$ may currently reside beginning at location $800$. After relocation, the new version of the program must be written to run correctly in a range termed the "Destination Block" although it will actually reside currently in a range termed the "Destination Segments". Thus a program may be relocated such that it will run correctly from location $D800$ (a ROM address) yet reside beginning at location $C00$ prior to being saved on tape or used to burn EPROMs (obviously, the relocated program cannot immediately reside at locations reserved for ROM). In some cases, the Source and Destination Segments may overlap.
**BLOCKS AND SEGMENTS EXAMPLE**

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

- $800$
- $B87$
- $C00$
- $F87$

**Blocks:**
Locations where Programs Run

- Original program runs from location $400$ on APPLE I
- Relocated version runs from location $D800$ on APPLE II

**SOURCE BLOCK:** $400-787$
**DESTINATION BLOCK:** $D800-DB87$

**SOURCE SEGMENTS:** $800-B87$
**DESTINATION SEGMENTS:** $C00-F87$

14
DATA SEGMENTS

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

CODE AND DATA SEGMENTS EXAMPLE

The Source Code Segments are relocated (using the 6502 Code-Relocation feature), while the Source Data Segments are moved (using the Monitor Move command).
HOW TO USE THE CODE-RELOCATION FEATURE

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

D4D5G return

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

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

2. Load the source program into the "Source Segments" area of memory (if it is not already there). Note that this need not be where the program normally runs.

3. Specify the Destination and Source Block parameters. Remember that a Block refers to locations from which the program will run, not the locations at which the Source and Destination Segments actually reside during the relocation. If only a portion of a program is to be relocated, then that portion alone is specified as the Block.

DEST BLOCK BEG < SOURCE BLOCK BEG . SOURCE BLOCK END ctrl Y * return

Notes: the syntax of this command closely resembles that of the Monitor Move command. Type "ctrl Y" by pressing the Y key while holding down the CTRL key. Then type an asterisk ( * ); and finally, type "return" by pressing the RETURN key. Do not type any spaces within the command.
4. Move all Data Segments and relocate all Code Segments in sequential (increasing address) order. It is wise to prepare a list of segments, specifying beginning and ending addresses, and whether each segment is code or data.

**If First Segment is Code:**

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

**If First Segment is Data:**

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

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

**Subsequent Code Segments:**

. SOURCE SEGMENT END ctrl Y return (Relocation)

**Subsequent Data Segments:**

. SOURCE SEGMENT END M return (Move)

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

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

EXAMPLE 1. Straightforward Relocation

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

(a) Initialize Code-Relocation feature:

reset D4D5G return

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

A00 < 800 . 97F ctrl Y * return

(c) Relocate first segment (code):

A00 < 800 . 88F ctrl Y return
(d) Move subsequent Data Segments and relocate subsequent Code Segments, in ascending address sequence:

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

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

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

**EXAMPLE 2. Index into Block**

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

```
LDA 7B0,X
```

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

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

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

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

Note: with this Block specification, all program references to the "prior page" (in this case the $700 page) will be relocated.
EXAMPLE 3. Immediate Address References

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

LDA #$3F
STA LOCØ
LDA #$08
STA LOCI
JMP (LOCØ)

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

EXAMPLE 4. Unusable Block Ranges

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

<table>
<thead>
<tr>
<th>Source And Destination Segments</th>
<th>Source And Destination Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs from locations $400-$78F on an APPLE I, but must be relocated to run from locations $D800-$DB8F on the APPLE II.</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):

D800 < 400 . 78F ctrl Y return

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

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

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

800 < . 97F ctrl Y return

**EXAMPLE 5. Changing the Page Zero Variable Allocation**

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

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

(a) Initialize the Code-Relocation feature:

reset D4D5G return

(b) Specify Destination and Source Blocks:

80 < 20 . 3F ctrl Y * return

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

800 < . 88F ctrl Y return (first segment, code)
. 8AF M return (data)
. 90F ctrl Y return (code)
. 93F M return (data)
. 97F ctrl Y return (code)
EXAMPLE 6. Split Blocks with Cross-Referencing

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

(a) Initialize the Code-Relocation feature:

D4D5G return

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

8A7 < 900. 9F1 ctrl Y * return

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

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

Note that any Data Segments within the two programs would necessitate additional relocation and Move commands.
EXAMPLE 7. Code Deletion

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

**SOURCE SEGMENTS**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800-$88F</td>
<td>CODE</td>
</tr>
<tr>
<td>$890-$8AF</td>
<td>DATA</td>
</tr>
<tr>
<td>$8B0-$90F</td>
<td>CODE</td>
</tr>
<tr>
<td>$8C4-$90F</td>
<td>CODE</td>
</tr>
<tr>
<td>$910-$93F</td>
<td>DATA</td>
</tr>
<tr>
<td>$940-$97F</td>
<td>CODE</td>
</tr>
</tbody>
</table>

Remove 4 bytes here
($8C0-$8C3)

**DEST SEGMENTS**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800-$88F</td>
<td>CODE</td>
</tr>
<tr>
<td>$890-$8AF</td>
<td>DATA</td>
</tr>
<tr>
<td>$8B0-$90B</td>
<td>CODE</td>
</tr>
<tr>
<td>$90C-$93B</td>
<td>DATA</td>
</tr>
<tr>
<td>$93C-$97B</td>
<td>CODE</td>
</tr>
</tbody>
</table>

**SOURCE BLOCK:** $8C4-$97F
**DEST BLOCK:** $8C0-$97B

**SOURCE SEGMENTS:** $800-$88F (code) $890-$8AF (data) $8B0-$8BF (code) $8C4-$90F (code) $910-$93F (data) $940-$97F (code)

**DEST SEGMENTS:** $800-$88F (code) $890-$8AF (data) $8B0-$8BF (code) $8C0-$90B (code) $90C-$93B (data) $93C-$97B (code)

(a) Initialize Code-Relocation feature:

reset D4D5G return

(b) Specify Destination and Source Blocks:

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

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

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)

$F962-F8A2$ (data) $962-A2$ (data)

$FA43-FB18$ (code) $A43-B18$ (code)

$FB19-FB1D$ (data) $B19-B1D$ (data)

$FB1E-FFCB$ (code) $B1E-FCB$ (code)

$FFCC-FFFF$ (data) $FCC-FFF$ (data)

IMMEDIATE ADDRESS REFERENCES (see EXAMPLE 3): $FFBF$

$FEA8$

(more if not relocating to page boundary)

(a) Initialize the Code-Relocation feature:

reset D4D5G return

(b) Specify Destination and Source Block parameters:

$700 < F700 . FFFF ctrl Y * return$

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

$800 < F800 . F961 ctrl Y return$ (first segment, code)

. $FA42 M return$ (data)

. $FB18 ctrl Y return$ (code)

. $FB1D M return$ (data)

. $FFCB ctrl Y return$ (code)

. $FFFF M return$ (data)

(d) Change immediate address references:

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

$EA8 : E return$ (was $FE$)
PART C: PLOTTING POINTS AND LINES

TECHNICAL INFORMATION

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

1. The APPLE II Monitor command

\texttt{Addr4 < Addr1 \ . \ Addr2 \ ctrl \ Y \ return} \quad \text{\texttt{(Addr1, Addr2, and Addr4 are addresses)}}

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

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

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

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

4. \texttt{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 \texttt{LENGTH} (location $2F$).

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

\[
ADR = FRMBEG + TOBEG
\]

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

**ALGORITHM USED BY THE CODE-RELOCATION FEATURE**

1. Set SOURCE PTR to beginning of Source Segment and DEST PTR to beginning of Destination Segment.
2. Copy 3 bytes from Source Segment (using SOURCE PTR) to temp INST area.
3. Determine instruction length from opcode (1, 2 or 3 bytes).
4. If two-byte instruction with non-zero-page addressing mode (immediate or relative) then go to step 7.
5. If two-byte instruction then clear 3rd byte so address field is 0-255 (zero page).
6. If address field (2nd and 3rd bytes of INST area) falls within Source Block, then substitute

\[
ADR = SOURCE BLOCK BEG + DEST BLOCK BEG
\]
7. Move "length" bytes from INST area to Destination Segment (using DEST PTR). Update SOURCE and DEST PTR's by length.
8. If SOURCE PTR is less than or equal to SOURCE SEGMENT END then goto step 2., else done.
COMMENTS:

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

1. Source Segments and Destination Segments do not share any common locations (no source location is overwritten).

2. Source Segments are in locations identical to the locations of the Destination Segments (each source byte overwrites itself).

3. Source Segments are in locations whose addresses are larger than the addresses of the Destination Segments' locations (any overwritten source bytes have already been Moved or relocated). This is a move toward smaller addresses.

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

Note: the example instructions in this chapter often include spaces for easier reading; do not type these spaces.
30  Testing APPLE's memory

31  Address ranges for standard memory configurations

32  Error messages
   Type I - Simple error
   Type II - Dynamic error

33  Testing for intermittent failure

34  Comments
TESTING THE APPLE'S MEMORY

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

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

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

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

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

D5BCG return

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

a . p ctrl Y return

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

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

\[ \text{2000.10 ctrl Y return} \]

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>Configuration</th>
<th>Memory Row of</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>Blocks Look like this:</td>
<td>Then</td>
<td></td>
<td></td>
</tr>
<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:

\[ \text{400.4 ctrl Y return} \]

This tests the screen area of memory

\[ \text{800.8 ctrl Y return} \]

These first four tests examine the first 16K row of memory (Row C)

\[ \text{1000.10 ctrl Y return} \]

This tests the second 16K row of memory (Row D)

\[ \text{2000.20 ctrl Y return} \]

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:

\[ \text{3000.20 ctrl Y return} \]

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

\[ \text{7000.20 ctrl Y return} \]
Tests may be run separately or they may be combined into one instruction. For instance, for a 48K system you can type:

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

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

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

ERROR MESSAGES

TYPE I - Simple Error

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

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

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

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

Example:

```
201F 00 10 ERR D-7
```
TYPE II - Dynamic Error

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

xxxx yy zz vvvv qq ERR r-c

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

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

Example:

5051 00 08 5451 00 ERR E-6

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

TESTING FOR INTERMITTENT FAILURE
(Automatically Repeating Test)

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

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

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

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

Example (for a 48K system):

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

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

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

COMMENTS:

1. You cannot test the APPLE’s memory below the address of 400 (Hex), since various pointers and other system necessities are there. In any case, if that region of memory has problems, the APPLE won’t function.

2. For any subtest, the number of pages tested cannot be greater than the starting address divided by 100 Hex. 2000.30 ctrl Y will not work, but 5000.30 ctrl Y will.

3. Before changing anything inside the APPLE, make sure the APPLE is plugged into a grounded, 3-wire power outlet, and that the power switch on the back of the computer is turned off. Always touch the outside metal bottom plate of the APPLE II, prior to handling any memory chips. This is done to remove any static charge that you may have acquired.

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

4. Besides the eight memory chips, some additions of memory require changing three other chip-like devices called Memory Configuration Blocks. The Memory Configuration Blocks tell the APPLE which type of memory chip (4K or 16K) is to be plugged into each row of memory. A complete package for adding memory to your computer, containing all necessary parts and detailed instructions, can be purchased from APPLE Computer Inc. To add 4K of memory, order the 4K Memory Expansion Module (P/N A2M0014). To add 16K of memory, order the 16K Memory Expansion Module (P/N A2M0016).
CHAPTER 7
MUSIC

36 Generating musical tones
37 Comments
GENERATING MUSICAL TONES

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

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

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

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

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 \( \text{MM} \), use a duration of \( 10200/\text{MM} \). For example, to get 204 notes per minute (approximately) use the command

POKE TIME, 10200/204
There are five timbres, coded by the numbers 2, 8, 16, 32 and 64. They are not very different from one another. With certain timbres, a few of the extremely low or high notes do not give the correct pitch. Timbre 32 does not have this problem.

POKE TIMBRE, 32

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

CALL MUSIC

will cause the specified note to sound.

The following program plays a chromatic scale of four octaves:

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

Where X is a number from 51 through 255,

POKE PITCH, X

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

40 POKE PITCH, 86

for a demonstration.

COMMENTS:

Some extremely high or low notes will come out at the wrong pitch with certain timbres.
CHAPTER 8

HIGH-RESOLUTION GRAPHICS

40 Part A: Setting up parameters, subroutines, and colors
40 Positioning the High-Resolution parameters
41 Defining subroutine names
41 Defining color names
42 Speeding up your program

43 Part B: Preparing the screen for graphics
43 The INITialization subroutine
43 Changing the graphics screen
44 CLEARing the screen to black
44 Coloring the BacKGrouND

45 Part C: PLOTting points and LINES

46 Part D: Creating, saving and loading shapes
46 Introduction
47 Creating a Shape Table
53 Saving a Shape Table
54 Loading a Shape Table
55 First use of Shape Table

56 Part E: Drawing shapes from a prepared Shape Table
56 Assigning parameter values: SHAPE, SCALE AND ROTation
57 DRAWing shapes
58 Linking shapes: DRAW1
59 Collisions

60 Part F: Technical information
60 Locations of the High-Resolution parameters
61 Variables used within the High-Resolution subroutines
62 Shape Table information
63 Integer BASIC memory map for graphics

64 Part G: Comments

39
PART A: SETTING UP PARAMETERS, SUBROUTINES, AND COLORS

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

POSITIONING THE HIGH-RESOLUTION PARAMETERS

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

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

The purpose of this statement is simply to place the six BASIC variable names used by the High-Resolution feature (with space for their values) into APPLE's "variable table" in specific, known locations. When line 0 is executed, the six High-Resolution graphics parameters will be assigned storage space at the very beginning of the variable table, in the exact order specified in line 0. Your BASIC program then uses those parameter names to change the six parameter values in the variable table. However, the High-Resolution subroutines ignore the parameter names, and look for the parameter values in specific variable-table locations. That is why the program's first line must place the six High-Resolution graphics parameters in known variable-table locations. Different parameter names may be used, provided that they contain the same number of characters. Fixed parameter-name lengths are also necessary to insure that the parameter-value storage locations in the variable table do not change. For example, the name HI could be used in place of X0, but X or XCOORD could not.
The parameters SHAPE, ROT, and SCALE are used only by the subroutines that draw shapes (DRAW and DRAW1, see PART E). These parameters may be omitted from programs using only the PLOT and LINE features:

\[ \emptyset \ X\emptyset = Y\emptyset = \text{COLR} \]

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 \[ \text{INIT} = -12288 \] : \[ \text{CLEAR} = -12274 \] : \[ \text{BKGND} = -11471 \]
6 \[ \text{POSN} = -11527 \] : \[ \text{PLOT} = -11506 \] : \[ \text{LINE} = -11500 \]
7 \[ \text{DRAW} = -11465 \] : \[ \text{DRAW1} = -11462 \]
8 \[ \text{FIND} = -11780 \] : \[ \text{SHLOAD} = -11335 \]

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

**DEFINING COLOR NAMES**

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

10 \[ \text{BLACK} = 0 \] : \[ \text{LET GREEN} = 42 \] : \[ \text{VIOLET} = 85 \]
11 \[ \text{WHITE} = 127 \] : \[ \text{ORANGE} = 170 \] : \[ \text{BLUE} = 213 \]
12 \[ \text{BLACK2} = 128 \] : \[ \text{WHITE2} = 255 \]

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.

\[
\begin{align*}
\text{\$} & \quad X_0 = Y_0 = \text{COLR} = \text{SHAPE} = \text{ROT} = \text{SCALE} \\
5 & \quad \text{INIT} = -12288 : \text{PLOT} = -11506 : \text{DRAW} = -11465 \\
10 & \quad \text{LET} \; \text{GREEN} = 42 : \text{WHITE} = 127
\end{align*}
\]

(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:

\[
\begin{align*}
\text{\$} & \quad X_0 = Y_0 = \text{COLR} = \text{SHAPE} = \text{ROT} = \text{SCALE} \\
2 & \quad I = J = K \\
5 & \quad \text{INIT} = -12288 : \text{CLEAR} = -12274 \\
6 & \quad \text{BKGND} = -11471 : \text{POSN} = -11527 \\
10 & \quad \text{BLACK} = \text{\$} : \text{VIOLET} = 85
\end{align*}
\]

3. Use the High-Resolution graphics parameter names as program variables when possible. Because they are defined first, these parameters are the BASIC variables which your program can find fastest.
PART B: PREPARING THE SCREEN FOR GRAPHICS

THE INITIALIZATION SUBROUTINE

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

CALL INIT

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

CHANGING THE GRAPHICS SCREEN

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

POKE -16302, 0

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

POKE -16301, 0

or another

CALL INIT

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

POKE 806, 64

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

POKE -16299, Ø

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, Ø

while

POKE 806, 32

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

CLEARING THE SCREEN

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

CALL CLEAR

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

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

CALL BKGND

The following program turns the entire display violet:

Ø XØ = YØ = COLR : REM SET PARAMETERS
5 INIT = -12288 : BKGND = -11471 : REM DEFINE SUBROUTINES
1Ø VIOLET = 85 : REM DEFINE COLOR
2Ø CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
3Ø COLR = VIOLET : REM ASSIGN COLOR VALUE
4Ø CALL BKGND : REM MAKE ALL OF DISPLAY VIOLET
5Ø 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 X0, Y0 and COLR. X0 must be 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 X-coordinate 35, Y-coordinate 55, and a violet dot at X-coordinate 85, Y-coordinate 90:

```
0 X0 = Y0 = 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 XV = 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 DRAW (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  XO = YO = 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 XO = 0 TO 270 STEP 10
70  YO = 0 : CALL PLOT : REM PLOT A STARTING-POINT AT TOP OF SCREEN
80  X0 = 190 : CALL LINE : REM DRAW A VERTICAL LINE TO BOTTOM OF SCREEN
90  NEXT XO : 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 DRAWI (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
- = 10 move down
- = 11 move left
- If P = 0 don't plot
- = 1 do plot

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 DRAWI 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 0000 as that will also be ignored. And a move of 0000 in section A will end your shape definition unless there is a 1-bit somewhere in section B or C.
Suppose you want to draw a shape like this:

![Shape Diagram]

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:

![Path Diagram]

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

![Vector Sequence]

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).

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>Byte 0</th>
<th>0 0 1 1 1 1 1 1</th>
<th>=</th>
<th>1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 1 1 1 1 1 1</td>
<td>=</td>
<td>3 F</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 1 1 1 0 0</td>
<td>=</td>
<td>2 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1 1 1 1 0</td>
<td>=</td>
<td>6 4</td>
</tr>
<tr>
<td>4</td>
<td>0 0 1 1 1 1 1 1</td>
<td>=</td>
<td>2 D</td>
</tr>
<tr>
<td>5</td>
<td>0 0 1 1 1 1 0 0</td>
<td>=</td>
<td>1 5</td>
</tr>
<tr>
<td>6</td>
<td>0 0 1 1 1 1 1 0</td>
<td>=</td>
<td>3 6</td>
</tr>
<tr>
<td>7</td>
<td>0 0 1 1 1 1 1 1</td>
<td>=</td>
<td>1 E</td>
</tr>
<tr>
<td>8</td>
<td>0 0 1 1 1 1 1 0</td>
<td>=</td>
<td>0 7</td>
</tr>
</tbody>
</table>
| 9     | 0 0 1 1 1 1 1 1 | = | 0 0 | ←Denotes End of Shape

Hex: Digit 1 Digit 2 Definition

<table>
<thead>
<tr>
<th>Section: C</th>
<th>B</th>
<th>A</th>
<th>Bytes Recoded in Hex</th>
<th>Codes Binary Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000 = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001 = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010 = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011 = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0100 = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0101 = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0110 = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0111 = 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000 = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1001 = 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1010 = A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1011 = B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1100 = C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101 = D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1110 = E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111 = F</td>
</tr>
</tbody>
</table>

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.
### Shape Definitions

<table>
<thead>
<tr>
<th>Index</th>
<th>Byte $S + \varnothing$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>Unused</td>
</tr>
<tr>
<td>+2</td>
<td>Lower 2 Digits</td>
</tr>
<tr>
<td>+3</td>
<td>Upper 2 Digits</td>
</tr>
<tr>
<td>+4</td>
<td>Lower 2 Digits</td>
</tr>
<tr>
<td>+5</td>
<td>Upper 2 Digits</td>
</tr>
<tr>
<td>+2n</td>
<td>Lower 2 Digits</td>
</tr>
<tr>
<td>+2n+1</td>
<td>Upper 2 Digits</td>
</tr>
</tbody>
</table>

- Total Number of Shape Definitions
- $D_1$: Index to First Byte of Shape Definition #1, Relative to $S$
- $D_2$: Index to First Byte of Shape Definition #2, Relative to $S$
- $D_n$: Index to First Byte of Shape Definition #n, Relative to $S$

#### Figure 4

- Shape Definition #1
- Shape Definition #2
- Shape Definition #n

### Start $= S$ → Byte $S + \varnothing$ → $n$ ($\varnothing$ to FF)

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

#### Figure 5

- Last Byte
- Shape Definition #1
- 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:

0800

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

0800 return

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

0800:01 return
0800 return
0800- 01

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

0800:

and then type the numbers, separated by spaces:

0800:01 00 04 00 12 3F 2D 64 2D 15 36 1E 07 00 return
You have just typed your first complete Shape Table...not so bad, was it?
To check the information in your Shape Table, you can examine each byte separately or simply press the RETURN key repeatedly until all the bytes of interest (and a few extra, probably) have been displayed:

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

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

```
328:00 08
```

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

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

```
LOMEM: 2048 + 15
```

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

Saving on Cassette Tape

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

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

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

0:0D 00 return

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

0.1W 0800.080DW

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

Saving on Disk

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

BSAVE filename, A$ startingaddress, L$ tablelength

For our example, you might type

BSAVE MYSHAPE1, A$ 0800, L$ 000D

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

Loading from Cassette Tape

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

CALL -11335 return

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

CALL SHLOAD return

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

Loading from Disk

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

BLOAD filename

From our previously-saved example, you would type

BLOAD MYSHAPE1

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

328:00 08 return

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

LOMEM: 2048 + 15
FIRST USE OF A SHAPE TABLE

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

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

HIMEM: 8192

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

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

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

```
114 FOR PAUSE = 1 TO 200 : NEXT PAUSE
116 COLR = BLACK : REM CHANGE COLOR
118 CALL DRAW : REM RE-DRAW SAME SHAPE, IN NEW COLOR
```
PART E: DRAWING SHAPES FROM A PREPARED SHAPE TABLE

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

ASSIGNING PARAMETER VALUES

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

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

SHAPE = 3

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

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

SCALE = 4

specifies that the next shape DRAWn will be four times the size that is described by the appropriate shape definition. That is, each "plotting vector" (either a plot and a move, or just a move) will be repeated four times. SCALE may be assigned any value from 0 through 255, but SCALE = 0 is interpreted as SCALE = 256, the largest size for a given shape definition.
You can also specify the orientation or angle of the shape to be DRAWn, by assigning the proper value to ROT. For example,

\[ \text{ROT} = \emptyset \]

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

\[ \text{ROT} = 16 \]

will cause the next shape to be DRAWn rotated 90 degrees clockwise. The value assigned to ROT must be within the range \( \emptyset \) to \( 255 \) (although \( \text{ROT}=64 \), specifying a rotation of 360 degrees clockwise, is the equivalent of \( \text{ROT}=\emptyset \)). For \( \text{SCALE}=1 \), only four of the 63 different rotations are recognized (\( \emptyset,16,32,48 \)); for \( \text{SCALE}=2 \), 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>( \emptyset )</td>
<td>(no rotation from shape definition)</td>
</tr>
<tr>
<td>48</td>
<td>(270 degrees clockwise rotation)</td>
</tr>
<tr>
<td>16</td>
<td>(90 degrees clockwise rotation)</td>
</tr>
<tr>
<td>32</td>
<td>(180 degrees clockwise rotation)</td>
</tr>
</tbody>
</table>

**DRAWING SHAPES**

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

\[ \emptyset \]

X0 = Y0 = COLR = SHAPE = ROT = SCALE : REM SET PARAMETERS
5 INIT = -12288 : DRAW = -11465 : REM DEFINE SUBROUTINES
1\( \emptyset \) WHITE = 127 : REM DEFINE COLOR
2\( \emptyset \) CALL INIT : REM INITIALIZE HIGH-RESOLUTION SUBROUTINES
3\( \emptyset \) X0 = 140 : Y0 = 80 : COLR = WHITE : REM ASSIGN PARAMETER VALUES
4\( \emptyset \) SHAPE = 3 : ROT = 24 : SCALE = 2
5\( \emptyset \) CALL DRAW : REM DRAW SHAPE 3, DOUBLE SIZE, TURNED 135 DEGREES
6\( \emptyset \) 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, Y0, and COLR, 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 : WHITE = 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 CHANGE
```

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).

\( X_0 = Y_0 = \text{COLR} = \text{SHAPE} = \text{ROT} = \text{SCALE} : \text{REM SET PARAMETERS} \)

\( \text{INIT} = -12288 : \text{LINE} = -11500 : \text{DRAW} = -11402 : \text{FIND} = -11780 \)

\( \text{VIOLET} = 85 : \text{WHITE} = 127 : \text{REM DEFINE SUBROUTINES AND COLORS} \)

\( \text{SHAPE} = 3 : \text{ROT} = 0 : \text{SCALE} = 2 \)

\( \text{CALL DRAW} : \text{REM DRAW SHAPE WITH ABOVE PARAMETERS} \)

\( \text{CALL FIND} : \text{REM FIND COORDINATES OF LAST SHAPE POINT} \)

\( \text{X}_0 = 10 : \text{Y}_0 = 25 : \text{COLR} = \text{VIOLET} : \text{REM NEW PARAMETER VALUES, FOR LINE} \)

\( \text{CALL LINE} : \text{REM DRAW LINE WITH ABOVE PARAMETERS} \)

\( \text{END} \)

**COLLISIONS**

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

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

\( \text{CALL DRAW} : \text{REM DRAW THE SHAPE} \)

\( \text{COUNT} = \text{PEEK}(810) : \text{REM FIND THE COLLISION COUNT} \)
PART F: TECHNICAL INFORMATION

LOCATIONS OF THE HIGH-RESOLUTION PARAMETERS

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

VARIABLE-TABLE PARAMETER LOCATIONS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Locations beyond LOMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>XØ</td>
<td>$05, $06</td>
</tr>
<tr>
<td>YØ</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>
<tr>
<td>Variable Name</td>
<td>Hexadecimal Location</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>SHAPEL, SHAPEH</td>
<td>1A, 1B</td>
</tr>
<tr>
<td>HCOLOR1</td>
<td>1C</td>
</tr>
<tr>
<td>COUNTH</td>
<td>1D</td>
</tr>
<tr>
<td>HBASL, HBASH</td>
<td>26, 27</td>
</tr>
<tr>
<td>HMASK</td>
<td>30</td>
</tr>
<tr>
<td>QDRNT</td>
<td>53</td>
</tr>
<tr>
<td>XOL, XOH</td>
<td>320, 321</td>
</tr>
<tr>
<td>YO</td>
<td>322</td>
</tr>
<tr>
<td>BXSAV</td>
<td>323</td>
</tr>
<tr>
<td>HCOLOR</td>
<td>324</td>
</tr>
<tr>
<td>HNDX</td>
<td>325</td>
</tr>
<tr>
<td>HPAG</td>
<td>326</td>
</tr>
<tr>
<td>SCALE</td>
<td>327</td>
</tr>
<tr>
<td>SHAPXL, SHAPXH</td>
<td>328, 329</td>
</tr>
<tr>
<td>COLLNSN</td>
<td>32A</td>
</tr>
</tbody>
</table>
SHAPE TABLE INFORMATION

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

<table>
<thead>
<tr>
<th>SHAPE TABLE</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of Table (Address Stored in $328-$329)</td>
<td>Number of Shapes →</td>
</tr>
<tr>
<td>0-255</td>
<td>→</td>
</tr>
<tr>
<td>Unused</td>
<td>02</td>
</tr>
<tr>
<td>Low</td>
<td>00</td>
</tr>
<tr>
<td>High</td>
<td>06</td>
</tr>
<tr>
<td>Low</td>
<td>00</td>
</tr>
<tr>
<td>High</td>
<td>00</td>
</tr>
<tr>
<td>First Byte</td>
<td>{</td>
</tr>
<tr>
<td>Last Byte=0</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>Shape #1</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>8A</td>
</tr>
<tr>
<td></td>
<td>A6</td>
</tr>
<tr>
<td></td>
<td>EE</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>BB</td>
</tr>
<tr>
<td></td>
<td>1D</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LOMEM → BASIC Variables (if Table SHLOADed) → BASIC Variables</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

62
APPLE II MEMORY MAP FOR USING HIGH-RESOLUTION GRAPHICS WITH INTEGER BASIC

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

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

   HIMEM: 8192 return

2. Using memory Page 2 for high-resolution graphics erases memory from location 16384 ($4000) to location 24575 ($5FFF). If yours is a 24K system, this will erase your BASIC program unless you do one of the following:
   a) never use Page 2 for graphics; or
   b) change HIMEM to 8192, as described above.

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

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

   LOMEM: 2048 + tablelength + 1

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

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

   POKE 0, 96

an accidental CALL to location zero will cause a simple jump back to your BASIC program, with no damage.
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>RAM Address</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 (6582 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>
1 ****************************
2 *
3 * APPLE-II HI-RESOLUTION  *
4 * GRAPHICS SUBROUTINES   *
5 *
6 * BY NOZ  9/13/77         *
7 *
8 * ALL RIGHTS RESERVED     *
9 *
10 ****************************

12 * HI-RES EQUATES
13 SHAPE EQU $1A POINTER TO
14 SHAPEH EQU $1B SHAPE LIST
15 HCOLOR EQU $1C RUNNING COLOR MASK
16 COUNT EQU $1D
17 HBASL EQU $26 BASE ADR FOR CURRENT
18 HBASH EQU $27 HI-RES PLOT LINE. A
19 HMASK EQU $30
20 AIL EQU $3C MONITOR A1.
21 AIM EQU $3D
22 AIL EQU $3E MONITOR A2.
23 AIL EQU $3F
24 LOMEM EQU $4A BASIC ‘START OF VARS’.
25 LOMEMH EQU $4B
26 DXL EQU $50 DELTA-X FOR HI, SHAPE.
27 DXL EQU $51
28 SHAPE EQU $51 SHAPE TEMP.
29 DY EQU $52 DELTA-Y FOR HLIN, SHAPE.
30 QDRT EQU $53 ROT QUADRANT (SHAPE).
31 LE EQU $54 ERROR FOR HLIN.
32 EL EQU $55
33 PPL EQU $CA BASIC START OF PROG PTR.
34 PPH EQU $C8
35 PHL EQU $CD
36 PHV EQU $CE BASIC ACC.
37 ACL EQU $CE BASIC ACC.
38 ACH EQU $CF
39 XOX EQU $320 PRIOR X-COORD SAVE
40 XOH EQU $321 AFTER HLIN OR HPLT.
41 YO EQU $322 HLIN, HPLT Y-COORD SAVE.
42 RXS EQU $323 X-REG SAVE FOR BASIC.
43 HCOLOR EQU $324 COLOR FOR HLIN, HPOSN.
44 HCOLOR EQU $325 HORIZ OFFSET SAVE.
45 HPAQ EQU $326 HI-RES PAGE ($20 NORMAL).
46 SCALE EQU $327 SCALE FOR SHAPE, MOVE.
47 SHAPE EQU $328 START OF
48 SHAPX EQU $329 SHAPE TABLE.
49 COL2 EQU $32A COLLISION COUNT.
50 HIRASH EQU $32B SWITCH TO HI-RES VIDEO
51 MEXEC EQU $32C SELECT TEXT/GRAPHICS MIX.
52 TXTCLR EQU $32D SELECT GRAPHICS MODE.
53 MEMFUL EQU $32E BASIC MEM FULL ERROR.
54 RNGERR EQU $32F BASIC RANGE ERROR.
55 GCADN EQU $F11E 2-BYTE TAPE READ SETUP.
56 D2B EQU $F11F TWO-EDGE TAPE SENSE.
57 READ EQU $FFD TAPE READ (A1, A2).
58 READX EQU $FF02 READ WITHOUT HEADER.

60 * HIGH RESOLUTION GRAPHICS INITS
61 *
62 * ROM VERSION $D000 TO $D3FF
63 *
64 ORG $D000
65 OBJ $4000

D000 A9 20
D002 BD 26 03
D001 A9 20
D002 BD 26 03

66 SETHRL LDA #$20 INIT FOR $2000-3FFF
67 STA HPAQ HI-RES SCREEN MEMORY.
DO05 AD 57 CO 68 LDA HIRES SET HIRES DISPLAY MODE
DO0B AD 53 CO 69 LDA MIXSET WITH TEXT AT BOTTOM.
DO0B AD 50 CO 70 LDA TXTCLR SET GRAPHICS DISPLAY MODE
DO0E A9 00 71 HCLR LDA $$0
DO10 B5 1C 72 BKGND0 STA HCOLOR1 SET FOR BLACK BKGND.
DO12 AD 26 03 73 BKGND LDA UPAG
DO15 B5 1B 74 STA SHAPE INIT HI-RES SCREEN MEM
DO17 AO 00 75 LDY $4100 FOR CURRENT PAGE, NORMALLY
DO19 B4 1A 76 STY SHAPE. $2000-3FFF OR $4000-5FFF
DO1B A5 1C 77 BKGND1 LDA HCOLOR1
DO1D 91 1A 78 STA (SHAPEL),Y
DO1F 20 A2 DO 79 JSR CBHPT2 (SHAPEL,H) WILL SPECIFY
DO22 C8 80 INY 32 SEPARATE PAGES.
DO23 D0 F6 81 BNF BKGND1 THROUGHOUT THE INIT.
DO25 E6 1B 82 INC SHAPEH
DO27 A5 18 83 STA (SHAPEL),Y
DO2F 20 A2 DO 84 JSR CSHFT2 (SHAPEL,H) WILL SPECIFY
DO32 CB 80 INY 32 SEPARATE PAGES.
DO34 DO F6 81 BNF BKGND1 THROUGHOUT THE INIT.
DO36 88 * HI-RES GRAPHICS POSITION AND PLOT SUBRS
DO31 BE 20 03 90 STX XL IN X-REG;
DO34 BC 21 03 91 STY XH IN X-REG.
DO37 4B 92 PHA
DO39 29 C0 93 AND $$CO
DO3A B5 26 94 STA HBASL FOR Y-COORD = 00ABCDEF.
DO3C 4A 95 LSR ;CALCULATES BASE ADDRESS
DO3D 4A 96 LSR ;IN HBASL, HBASH FOR
DO3E 05 26 97 STA HBASL ACCESSING SCREEN MEM
DO40 B5 26 98 STA HBASL VIA (HBASL),Y ADDRESSING MODE
DO42 6B 99 PLA
DO43 B5 27 100 STA HBASH
DO45 0A 101 ASL ;CALCULATES
DO46 0A 102 ASL ;HASH = $$PFPFHCD.
DO47 0A 103 ASL ;$HASH = EABAB000
DO48 26 27 104 ROL HBASH
DO4A 0A 105 ASL ;WHERE $$P=001 FOR $2000-3FFF
DO4B 26 27 106 ROL HBASH SCREEN MEM RANGE AND
DO4D 0A 107 ASL ;PPP=010 FOR $4000-7FFF
DO4E 66 26 108 ROR HBASH (GIVEN Y-COORD=ABCDEFH)
DO50 A5 27 109 LDA HBASH
DO52 29 1F 110 AND $$IF
DO54 OD 26 03 111 ORA HPAG
DO57 B5 27 112 STA HBASH
DO59 BA 1A 113 TXA DIVIDE XO BY 7 FOR
DO5A CO 00 114 CPY $$0 INDEX FROM BASE ADR
DO5C F0 05 115 BEQ HPOSN2 (QUOTIENT) AND BIT
DO5E AO 23 116 LDY $$23 WITHIN SCREEN MEM BYTE
DO5F 69 04 117 ADC *$1 (MASK SPEC'D BY REMAINDER)
DO61 CB 0B 118 HPOSN1 INC
DO63 E9 07 119 HPOSN2 SBC #7 SUBTRACT OUT SEVENS.
DO65 BO FB 120 BCS HPOSN1
DO67 BC 25 03 121 STY $NDX WORKS FOR XO FROM
DO6A AA 122 TAX 0 TO 279, LOW-ORDER
DO6B BD EA DO 123 LDA MSKTB1=249,X BYTE IN X-REG.
DO6E B5 30 124 STA $MASK HIGH IN Y-REG ON ENTRY
DO70 9B 125 TYA
DO71 4A 126 LSR ; IF ON ODD BYTE (CARRY SET)
DO72 AD 24 03 127 LDA HCOLOR THEN ROTATE HCOLOR ONE
DO75 B5 1C 128 HPOSN3 STA HCOLOR1 BIT FOR 180 DEGREE SHIFT
DO77 B9 29 129 BCS CBHPT2 PRIOR TO COPYING TO HCOLOR1.
DO79 60 130 RTS
DO7A 20 2E DO 131 HPLT JSR HPOSN
DO7D A5 1C 132 HPLT1 LDA HCOLOR1 CALC BIT POSN IN HBASL,H
DO7F 51 26 133 EOR (HBASL),Y HNDX, AND $MASK FROM
DO81 25 30 134 AND $MASK Y-COORD IN A-REG,
DO83 51 26 135 EOR (HBASL),Y X-COORD IN X,Y-REGS.
DO85 91 26 136 STA (HBASL),Y FOR ANY 'L' BITS OF $MASK
DO87 60 137 RTS SUBSTITUTE CORRESPONDING
DO89 30 138 * BIT OF HCOLOR1.

67
140 * HI-RES GRAPHICS L.R.U.D SUBRS

D23B 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 **CO ONE BIT TO LSB.
D031 B5 30 146 LRI STA HMASK
D033 60 147 RTS
D039 88 148 LFT1 DEY DECR HORIZ INDEX.
D03F 10 02 149 BPL LEFT2
D077 A0 27 150 LDY **27 WRAP AROUND SCREEN.
D079 A9 C0 151 LEFT2 LDA **CO NEW HMASK, RIGHTMOST
D07B 85 30 152 NEWNDX STA HMASK DOT OF BYTE.
D07E BC 25 03 153 STY HNHX UPDATE HORIZ INDEX.
D0A0 A5 1C 154 CSHIFT LDA HCOLOR1
D0A2 GA 155 CSHIFT2 ASL ; ROTATE LOW-ORDER
D0A3 C0 C0 156 CMP **CO 7 BITS OF HCOLOR1
D0A5 10 06 157 BPL RTS1 ONE BIT POSN.
D0A7 A5 1C 158 LDA HCOLOR1
D0A9 49 7F 159 EOR **7F ZYXYXYXY -> ZYXYXYXY
D0AB 85 1C 159 LR1 STA HCOLOR1
D0AD 60 161 RTS1 RTS
D0AE A0 30 162 RIGHT LDA HMASK
D0B0 0A 163 ABL ; SHIFT LOW-ORDER
D0B1 49 80 164 EOR **80 7 BITS OF HMASK
D0B3 30 DC 165 DM1 LR1 ONE BIT TO MSB.
D0B5 A9 81 166 LDA **81
D0B7 CB 167 INY NEXT BYTE.
D0B9 C0 2B 168 CPY **2B
D0BA 90 DF 169 BCS NEWNDX
D0BC A0 00 170 LDY **0 WRAP AROUND SCREEN IF >279
D0BE B0 01 171 BCB NEWNDX ALWAYS TAKEN.

173 * L.R.U.D. SUBROUTINES.

D0C0 18 174 LRUDX1 CLC NO 90 DEG ROT (X-OR).
D0CA A5 51 175 LRUDX2 LDA SHAPEX
D0C3 29 04 176 AND NO IF 82=0 THEN NO PLOT.
D0C5 F0 27 177 BEQ LRUD4
D0C7 A9 7F 178 LDA **7F FOR EX-OR INO SCREEN MEM
D0C9 25 30 179 AND HMASK
D0CB 31 26 180 AND (HBASL),Y SCREEN BIT SET?
D0CD 0D 18 181 BNE LRUD3
D0CF EE 2A 03 182 INC COLLSN
D0D2 A9 7F 183 LDA **7F
D0D4 25 30 184 AND HMASK
D0D6 05 DC 185 BPL LRUD3 ALWAYS TAKEN.
D0DB 18 186 LRUD1 CLC NO 90 DEG ROT.
D0D9 A5 51 187 LRUD2 LDA SHAPEX
D0DB 29 04 188 AND **4 IF 82=0 THEN NO PLOT.
D0DD F0 0F 189 BCS LRUD4
D0DF B1 26 190 LDA (HBASL),Y
D0E1 45 1C 191 EOR HCOLOR1 SET HI-RES SCREEN BIT
D0E3 25 30 192 AND HCOLOR1 TO CORRESPONDING HCOLOR1
D0E5 D0 03 193 BNE LRUD3 IF BIT OF SCREEN CHANGES
D0E7 EE 2A 03 194 INC COLLSN THEN INCR COLLSN DETECT
D0EA 51 26 195 LRUD3 EOR (HBASL),Y
D0EB 91 24 196 STA (HBASL),Y
D0EE A5 51 197 LRUD4 LDA SHAPEX ADD GDANT TO
D0F0 65 53 198 ADC GDANT SPECIFIED VECTOR
D0F2 29 03 199 AND **3 AND MOVE LFT, RT,
D0F4 0E 03 200 EG3 EQU -*-1 UP, OR DWN BASED
D0F5 0A 201 CMP **2 ON SIGN AND CARRY.
D0F7 08 B8 202 ADD ROR
D0F9 30 30 203 LRUD BC3 LFTRT
D0FB 30 30 204 UPDOWN BCS DOWN4 SIGN FOR UP/DWN SELECT
D0FB 18 205 UP CLC
D0FC A5 27 206 LDA HBASH CALC BASE ADDRESS
D0FD 0C D1 207 BIT EQIC (ADR OF LEFTMOST BYTE)
D0F1 D0 22 208 BNE UP4 FOR NEXT LINE UP
D033 06 26 209 ASL HBASL IN (HBASL, HBASH)
D105 B0 1A 210 BCS UP2 WITH 192-LINE WRAPAROUND
D107 2C F3 D0 211 BIT EQ3
D10A F0 05 212 BEQ UP1
D10C 69 1F 213 ADC #1F **** BIT MAP ****
D10E 3B 214 SEC
D10F B0 12 215 BCS UP3 FOR ROW = ABCDEFGH.
D111 69 23 216 UP1 ADC #23
D113 4B 217 PHA
D114 A5 26 218 LDA HBASL HBASL = EABAB000
D116 69 B0 219 ADC #B0 HBASH = PPPP000CD
D118 B0 02 220 BCS UP5
D11A 69 F0 221 ADC #FO WHERE PPP=001 FOR PRIMARY
D11C B5 26 222 UP5 STA HBASL HI-RES PAGE ($2000-$3FFF)
D11E 6B 223 PLA
D11F 30 02 224 BCS UP3
D121 69 1F 225 UP2 ADC #1F
D123 66 26 226 UP3 ROR HBASL
D125 69 FC 227 UP4 ADC #FC
D127 B5 27 228 UPDWN1 STA HBASH
D129 60 229 RTB
D12A 18 230 DOWN DLG
D12B A5 27 231 DOWN4 LDA HBASH
D12D 69 04 232 ADC #4 CALC BASE ADR FOR NEXT LINE
D12F 2C EA D1 233 EQ4 EQU -1 DOWN TO (HBASL,HBASH)
D132 D0 03 234 BIT EQ1C
D133 06 26 235 BNE UPDWN1
D136 90 19 236 BCC DOWN1
D138 69 E0 237 ADC #EO
D13A 1B 238 CLG
D13B 2C B5 D1 239 BIT EQ4
D13E F0 13 240 BE9 DOWN2
D140 A5 26 241 LDA HBASL
D142 69 50 242 ADC #50
D144 4F F0 243 EOR #40
D146 F0 02 244 BE0 DOWN3
D148 49 F0 245 EOR #FO
D14A B5 26 246 DOWN3 STA HBASL
D14C AD 26 03 247 LDA HPAG
D14F 90 02 248 BCC DOWN2
D151 69 E0 249 DOWN1 ADC #EO
D153 66 26 250 DOWN2 ROR HBASL
D155 90 D0 251 BCC UPDWN1

D157 4B 252 HLINL PHA
D159 A9 00 253 LDA #0 SET (XOL,XOH) AND
D15A B0 20 03 254 STA XOL YO TO ZERO FOR
D15D BD 21 03 255 STA XOH REL LINE DRAW
D15D BD 22 03 256 STA YO (DX, DY).
D153 6B 260 PLA
D164 6B 261 HLin PHA ON ENTRY
D165 3B 262 SEC XL: A-REQ
D166 ED 20 03 263 SBC XOL XH: X-REQ
D169 4B 264 PHA Y: Y-REQ
D16A 8A 265 TXA
D16B ED 21 03 266 SBC XOH
D16B 83 267 STA QDRNT CALC ABS(X-XG)
D170 B0 0A 268 BCS HLin2 IN (DXL,DXH)
D172 69 269  PLA
D173 49 FF 270  EOR  **FF X DIR TO SIGN BIT
D175 49 01 271  ADC  **1 OF QDRNT.
D177 48 272  PHA  O=RIGHT (DX POS)
D178 A9 00 273  LDA  **0 =LEFT (DX NEG)
D17A E5 53 274  SBC  QDRNT
D17C 85 51 275  HLIN2 STA DXH
D17E 85 53 276  STA  EM INIT (EL, EH) TO
D180 68 277  PLA  ARS(X-XO)
D181 85 50 278  STA  DDL
D183 85 54 279  STA  EL
D185 68 280  PLA
D186 BD 20 03 281  STA  XOL
D188 BD 21 03 282  STX  XOH
D18C 98 283  TVA
D18D 18 284  CLC
D18E ED 22 03 285  SBC  YO CALC -ABS(Y-C)-1
D191 90 04 286  BCC  HLIN3 IN DY.
D193 49 FF 287  EDX  EOR **FF
D195 69 FE 288  ADC  EOR
D197 85 52 289  HLIN3 STA DY ROTATE Y DIR INTO
D199 BC 22 03 290  STY  YO QDRNT SIGN BIT
D19C 66 53 291  ROR QDRNT (O=UP, 1=DOWM)
D19E 39 292  SEC
D19F E5 50 293  SBC  DXL INIT (COUNTL, COUNTH).
D201 A4 FF 294  TAX  TO -(DELTX+DELTY+1)
D202 A9 FF 295  STA  *0
D204 E5 51 296  SBC  DXH
D206 85 1D 297  STA  COUNTH
D20B AC 25 03 298  LDY  MX INDEX INDEX
D20D 80 05 299  BCS  MOVEX2 ALWAYS TAKEN.
D20F 0A 300  MOVEX ABL : MOVE IN X-DIR. USE
D214 20 86 DO 301  JSR  LEFT RT QDRNT B6 FOR LFT/RT SELECT
D218 38 302  SEC
D21B 55 303  MOVEX2 LDA  EL Assumes CARRY SET.
D21D 55 304  SBC  DDY (EL, EH)-DELTY TO (EL, EH)
D21F 55 305  STA  EL NOTE: DY IS (-DELTY)-1
D221 55 306  LDA EM CARRY CLR IF (EL, EH)
D223 E9 00 307  SBC **0 GOES NEG.
D225 55 308  HCOUNT STA EH
D22E B1 26 309  LDA (HRASL), Y SCREEN BYTE.
D230 45 1C 310  EOR HCOLORI PLOT DOT OF HCOLORI
D232 AB 30 311  AND HMASK CURRENT BIT MABK.
D234 91 26 312  EOR (HRASL), Y
D236 91 26 313  STA (HBRASL), Y
D23E EB 314  INX DONE (DEI, TX+DELTY)
D243 E0 04 315  BNE- HLIN4 DOTS?
D247 E6 01 316  INC COUNTH
D24D F0 68 317  BEQ RTS2 YES, RETURN.
D24F 55 318  HLIN4 LDA QDRNT FOR DIRECTION TEST
D251 BO DA 319  BCS MOVEX IF CAR SET, (EL, EH) POS
D253 20 F9 DO 320  JSR UPRDW IF CLR, NEG. MOVE YDIR
D256 18 321  CLC
D25D A5 34 322  LDA EL (EL, EH)+DELTX
D25F 69 50 323  ADC DXL TO (EL, EH).
D263 55 324  STA EL
D265 A9 55 325  LDA EH CAR SET IF (EL, EH) GOES POS
D267 D3 51 326  ADC DXH
D26E 50 D9 327  BVC HCOUNT ALWAYS TAKEN.
D273 B1 328  MSKTLB HEX 81 LEFTMOST BIT OF BYTE.
D276 84 80 329  HEX 82, 84, 88
D278 90 AO 330  HEX 90, AO
D27A 2C 331  HEX CO RIGHTMOST BIT OF BYTE.
D280 1C 332  QEHC Hex 1C
D282 FE FA 333  CO HEX FF, FE, FA, F4, EC, E1, D4, C5, B4
D285 A1 BD 78 334  Hex A1, BD, 76, 61, 49, 31, 18, FF
336 * HI-RES GRAPHICS COORDINATE RESTORE SUBR

D1FC A5 26 337 HFIND LDA HBAVL
D1FE OA 338 ABL ; CONVERTS BASEADR
D1FF A5 27 339 LDA HBASH TO Y-COORD.
D201 29 03 340 AND #3
D203 2A 341 ROL ; FOR HBAVL = EABABCCD
D204 B5 26 342 DRA HBAVL HBASH = PPPF9HCD
D206 OA 343 ASL
D207 OA 344 ASL ; GENERATE
D208 OA 345 ASL ; Y-COORD = ABCDEFGH
D209 BD 22 03 346 STA YO
D20C AS 27 347 LDA HBAVL (PPP=SCREEN PAGE,
D210 29 07 348 LSR ; NORMALLY 001 FOR
D212 OD 22 03 349 ORA YO
D215 BD 22 03 350 STA YO CONVEPTS HNDX (INDEX
D218 AD 25 03 351 LDA HNDX FROM BASEADR)
D21A 0A 352 ASL ; AND HNDX MASK (BIT
D21C 6D 25 03 353 ADC HNDX TO X-COORD
D21E 0A 354 ASL ; IN (XOL,XOH)
D220 AA 355 TAX (RANGE $0-$133)
D221 CA 356 DEX
D222 A5 30 357 LDA HMAK.
D224 29 7F 358 AND #2FF
D226 EB 359 HFIND1 INX
D227 4A 360 LSR
D228 D0 FC 361 BNE HFIND1
D22A BD 21 03 362 STA XOH
D22D BA 363 TAX
D22E 18 364 CLC CALC HNDX*7 +
D22F 6D 25 03 365 ADC HNDX TO X-COORD
D232 90 03 366 BCC HFIND2
D234 EE 21 03 367 INC ION
D235 BD 20 03 368 HFIND2 STA XOL
D23A 60 369 RTS2 RTS

373 * HI-RES GRAPHICS SHAPE DRAW SUBR
374 * SHAPE DRAW
375 * R = D TO 63
376 * SCALF FACTOR USED (1=NORMAL)
377 *
D238 96 1A 378 DRAW STX SHAPEL. DRAW DEFINITION
D23D 8A 1B 379 STY SHAPEH POINTER.
D23F AA 380 DRAW1 TAX
D240 4A 381 LDR1 ; ROT (#0-#3F)
D241 4A 382 LSR
D242 4A 383 LSR ; GDRNT 0=UP, 1=RT,
D243 4A 384 LSR ; 2=DN, 3=LFT.
D244 8S 53 385 STA GDRNT
D246 BA 386 TAX
D247 29 1F 387 AND #1F
D249 AA 388 TAX
D24A AC EC D1 389 LDY COS*X SAVE COS AND SIN
D24D 8A 50 390 STY DXL VALS IN DXL AND DY.
D250 8F 9F 391 EOR #FF
D251 AA 392 TAX
D252 BC EC D1 393 LDY COS+1, X
D25B CB 394 INY
D25E 8A 52 395 STY DY
D25F AC 25 03 396 DRAW2 LDY HNDX BYTE INDEX FROM
D265 BD 02 00 397 LDX #00 HI-RES BASEADR
D266 8E 2A 03 398 STX COLLISION CLEAR COLLISION COUNT.
D26A A1 1A 399 LDA (SHAPEL,X) 1ST SHAPE DEF BYTE.
RS HIF—RES GRAPHICS SHAPE EX-OR SUBR

432 * HI-RES GRAPHICS SHAPE EX-OR SUBR
433 *
434 * EX-OR SHAPE INTO SCREEN.
435 *
436 * ROT = 0 TO 3 (QUADRANT ONLY)
437 * SCALE IS USED
438 *

439 XDRAW STX SHAPE SHAPE DEFINITION
D9C B4 1B 440 STY SHAPE SHAPE DEFINITION
D9E AA 441 XDRAW1 TAX
D9F 4A 442 LSR ; ROT ($0—$3F)
D9A 4A 443 LSR
D91 4A 444 LSR ; QDRNT 0=UP, 1=RT,
D92 4A 445 LSR ; 2=DWN, 3=LFT.
D93 B5 53 446 STA QDRNT
D94 B8 447 TXA
D9A 2F 448 AND $FF
D9B AA 449 TAX
D9C BC E8 D1 450 LDY COS, X SAVE COS AND SIN
D9C B4 50 451 STY DXL VALS IN DXL AND DY.
D9E 4F 452 EOR $FF
D98 AA 453 TAX
D9B BC EC D1 454 LDY COS+1, X
D9B C8 455 INY
D9C 4B 456 STY DV
D97 AC 25 03 457 XDRAW2 LDY HNDX INDEX FROM HI-RES
D9A E2 00 458 LDX $00 BASE ADR.
D9B BE 2A 03 459 STX COLLSN CLEAR COLLISION DETECT
D9E A1 1A 460 LDA (SHAPE, X) 1ST SHAPE DEF BYTE.
DRAW ROUTINES

D337 20 F9 D2 523 BDRAW1 JSR BPOSN
D339 20 51 D3 524 BDRAW JSR BDRAWX DRAW CALL FROM BASIC.
D33B 20 3B D2 525 JSR DRAW
D33F AE 23 03 526 LDX BXSAV
D343 60 527 RTS
D344 20 F9 D2 528 BXDRW1 JSR BPOSN
D347 20 51 D3 529 BXDRAW JSR BDRAWX EX-OR DRAW
D34A 20 9A D2 530 JSR XDRAW FROM BASIC.
D34D AE 23 03 531 LDX BXSAV
D350 60 532 RTS
D351 8E 23 03 533 BDRAWX STX BXSAV SAVE FOR BASIC.
D354 A0 32 534 LDY #32
D356 20 92 D3 535 JSR BYTE SCALE FROM BASIC.
D359 B0 27 03 536 STA SCALE
D35C A0 28 537 LDY #28
D35E 20 92 D3 538 JSR BYTE ROT FROM BASIC.
D361 48 539 PLA SAVE ON STACK.
D362 AD 28 03 540 LDA SHAPEL
D365 B5 1A 541 STA SHAPE START OF SHAPE TABLE.
D369 A5 13 542 LDA SHAPE SAVE FOR BASIC
D36C A0 20 543 LDX BXSAV
D36E 20 92 D3 544 JSR EX-OR DRAW
D371 F0 39 545 BEQ RERR1
D373 D2 00 546 JSR XDRAW FROM BASIC.
D375 AE 23 03 547 LDX BXSAV
D378 A0 16 548 LDY 41116
D37B 81 4A 549 PBYTE LDA (LOMEMU),Y
D37C DO 16 550 BNB RERR1 GET BASIC PARAM.
D37E 88 570 (ERR IF >255)
D380 B1 28 551 LDA (LOMEMU),Y XO LOW-ORDER BYTE
D382 AA 552 TAX
D383 CS 553 INY
D384 81 4A 554 LDA (LOMEMU),Y HI-ORDER BYTE.
D386 AS 555 TAY
D387 ED 18 556 CPX #118
D389 E9 01 557 SBC $11 RANGE ERR IF >279
D38B 90 ED 558 RERR1 JMP RNGERR.
D38F 4C 68 EE 559 JMPE RERR1
D390 A0 16 560 LDY #16
D392 B1 4A 561 PBYTE LDA (LOMEMU),Y
D394 D0 16 562 BNE RERR1 GET BASIC PARAM.
D396 8B 570 563 DEY (ERR IF >255)
D398 B1 4A 564 LDA (LOMEMU),Y
D39B 60 565 RTSB RTS
D39D 8E 23 03 573 GETX0 STX BXSAV SAVE FOR BASIC.
D39F AO 05 574 LDY #5
D3A1 B1 4A 575 LDA (LOMEMU),Y XO LOW-ORDER BYTE.
D3A4 68 EE 576 TAX
D3A7 81 4A 577 LDY (LOMEMU),Y HI-ORDER BYTE.
D3A9 4B 578 TAY
D3AC 1E 18 580 CPX #18
D3AF E7 01 581 SBC #1 RANGE ERR IF >279
D3B3 A0 90 ED 582 BCS RTSB
D3B6 4C 68 EE 583 RERR1 JMPI RNGERR.
D3B9 A0 0D 584 GETYO LDY #0 OFFSET TO YO FROM LOMEM
D3BE 20 92 D3 585 JSR BYTE QE1 BASIC PARAM YO.
D3C4 C9 CO 586 CMP #101 (ERR IF >191)
D3CB 80 F4 587 BCS RERR1
D3D8 60 588 RTS
SHAPE TAPE LOAD SUBROUTINE

590 * SHAPE TAPE LOAD SUBROUTINE
591 SHLOAD STX BXSAV SAVE FOR BASIC.
592 JSR ACDADR READ 2-BYTE LENGTH INTO
593 JSR READ BASIC ACC
594 LDA #$00 ;START OF SHAPE TABLE IS #$0800
595 STA A1L
596 STA SHAPXL
597 CLC
598 ADC ACL
599 TAY
600 LDA #$08 ;HIGH BYTE OF SHAPE TABLE POINTER.
601 STA A1H
602 STA SHAPXH
603 ADC ACH
604 BCS MFULL1 NOT ENOUGH MEMORY.
605 CPY PPL
606 PHA
607 SBC PPH
608 PLA
609 BCS MFULL1
610 STY A2L
611 STA A2H
612 INY
613 BNE SHI OD1
614 ADC #$1
615 SHLOAD STY LOMEML
616 STA LOMEMH
617 STY PVL
618 STA PVH
619 JSR RD2BIT
620 LDA #$3 ;SECOND HEADER.
621 JSR READX1
622 LDX BXSAV
623 RTS
624 MFULL1 JMP MEMFUL

--- END ASSEMBLY ---

TOTAL ERRORS: 00
APPLE-II BASIC RENUMBER / APPEND SUBROUTINES

VERSION TWO

ренumber

-clear

-start=

-call -10531

optional

-from=

to=

-call -10521

use renx entry

for renumber all

MoZ 04 12, 7970

Apple Computer Inc.

wendung 6502 equates

rol equ $0

low-order sw16 ro byte.

r0h equ $1

hi-order.

one equ $01

r0l equ $16

low-order sw16 r11 byte.

r0h equ $17

hi-order.

himem equ $4c

basic himem pointer.

ppl equ $ca

basic prog pointer.

pvl equ $cc

basic var pointer.

memfull equ $e360

basic mem full error.

prdec equ $e51b

basic decimal print subr.

ranger equ $ee6b

basic range error.

load equ $f0df

basic load subr.

sw16 equ $f089

swift 16 entry.

crut equ $f08e

car ret subr.

cout equ $f0de

char out subr.

wendung sweet 16 equates

acc equ $0

sweet 16 accumulator.

newlow equ $1

new initial lno.

newincr equ $2

new lno incr.

lnlow equ $3

lno low of renum range.

lnhi equ $4

lno high of renum range.

tblstart equ $5

lno table start.

tblndx1 equ $6

pass 1 lno tbl index.

tblim equ $7

lno table limit.

scr7 equ $8

scratch reg.

himem equ $9

himem (end of prom).

scr9 equ $9

scratch reg.

prgndx equ $a

pass 1 prog index.

prgndx1 equ $a

also prog index.

newln equ $b

next "new lno".

newln1 equ $c

prior "new lno" assign.

tbland equ $6

pass 2 lno table end.

prgndx2 equ $7

pass 2 prog index.

chr0 equ $9

ascii "0".

chr1 equ $a

ascii "a".
66 MODE EQU $C  
67 TBLNDX2 EQU $B  
68 OLDLN EQU $D  
69 STRCON EQU $B  
70 REM EQU $C  
71 R13 EQU $D  
72 THEN EQU $D  
73 LIST EQU $D  
74 DEL EQU $D  
75 SCRC EQU $C  
76 *  
77 *  
77 MODE EQU CONST/LNO MODE.  
78 TBLNDX2 EQU LNO TBL IDX FOR UPDATE.  
79 OLDLN EQU OLD LNO FOR UPDATE.  
80 STRCON EQU BASIC STR CON TOKEN.  
81 REM EQU BASIC REM TOKEN.  
82 R13 EQU SWEET 16 REG 13 (CPR REG).  
83 THEN EQU BASIC THEN TOKEN.  
84 LIST EQU BASIC LIST TOKEN.  
85 DEL EQU SCRATCH REG FOR APPEND.  
86 *  
87 *  
87 APPLE-11 BASIC RENUMBER SUBROUTINE - PASS 1  
88 ORG $0400  
89 08J $4400  
90 D400 20 89 F6 81 RENX JSR SW16 OPTIONAL RANGE ENTRY.  
91 D403 B0 82 JSR SW16 SET LNLOW=0, LNHI=0.  
92 D404 33 83 ST LNLOW SET LNLOW=0, LNHI=0.  
93 D405 34 84 ST LNHI  
94 D406 F4 85 DCR LNHI  
95 D407 00 86 RTN  
96 D408 20 89 F6 87 RENUM JSR SW16  
97 D409 1B 4C 00 88 SET HMEM,HIMEM  
98 D40E 6B 89 LDD @HMEM  
99 D40F 38 90 ST HMEM  
100 D410 19 CE 00 91 RNUM3 SET SCR9,PVL+2  
101 D413 C9 92 POPD @SCR9 BASIC VAR PNT TO  
102 D414 35 93 ST TBLSTRT TBLSTRT AND TBLNDX1.  
103 D415 36 94 ST TBLNDX1  
104 D416 21 95 LD NEWLN  
105 D417 3B 96 ST NEWLN1  
106 D418 3C 97 ST NEWLN1  
107 D419 C9 98 POPD @SCR9 BASIC PROG PNTR  
108 D41A 39 99 ST TBLIM TO TBLIM AND PRGNDX  
109 D41B 3A 100 ST PRGNDX  
110 D41C 29 101 PASS1 LD PRGNDX  
111 D41D DB 102 CPR HMEM IF PRGNDX >= HMEM THEN PASS 1.  
112 D41E 03 46 103 BC PASS2 THEN DONE PASS 1.  
113 D420 3A 104 ST PRGNDX1  
114 D421 26 105 LD TBLNDX1  
115 D422 E0 106 INR ACC IF < TWO BYTES AVAIL IN  
116 D423 D7 107 CPR TBLIM LNO TABLE THEN RETURN.  
117 D424 03 3B 108 BC MEHR WITH "MEM FULL" MESSAGE.  
118 D425 4A 109 LD @PRGNDX1  
119 D427 A9 110 ADD PRGNDX ADD LENTH BYTE TO PROG INDEX.  
120 D42B 39 111 ST PRGNDX  
121 D429 6A 112 LDD @PRGNDX1 LINE NUMBER.  
122 D42A D3 113 CPR LNLOW IF < LNLOW THEN GOTO P1B.  
123 D42B 02 2A 114 BNC P1B  
124 D42D 44 115 CPR LNHI IF > LNHI THEN GOTO P1C.  
125 D42E 02 02 116 BNC P1C  
126 D430 07 30 117 BNZ P1C  
127 D432 76 118 P1A STD @TBLNDX1 ADD TO LNO TABLE.  
128 D433 00 119 RTN  
129 D434 A5 01 120 LDA ROH **** 6502 CODE ****  
130 D436 A6 00 121 LDX ROL  
131 D439 20 1B E5 122 JSR PRDEC PRINT OLD LNO "¬" NEW LNO (RO,.R11) IN DECIMAL  
132 D43B A9 AD 123 LDA #AB  
133 D43D 20 ED FD 124 JSR COUT  
134 D440 A9 BE 125 LDA #BE  
135 D442 20 ED FD 126 JSR COUT  
136 D445 A5 17 127 LDA R11H  
137 D447 A6 16 128 LDX R11L  
138 D449 20 1B E5 129 JSR PRDEC  
139 D44C 20 8E FD 130 JSR COUT  
140 D44F 20 8C F6 132 JSR SW16+3 **** END 6502 CODE ****
2.

D152 2B  
D153 3C  
D154 A2  
D155 3B  
D156 OD  
D157 D1  
D158 1C  
D159 4C  
D15A 00  
D15B 4C  
D15C EC  
D15D EC  
D15E 02  
D15F 4C  
D160 4C  
D161 4C  
D162 02  
D163 47  
D164 19  
D165 1A  
D166 37  
D167 DD  
D168 03  
D169 67  
D16A 3D  
D16B 25  
D16C 04  
D16D 03  
D16E 15  
D16F 23  
D170 24  
D171 23  
D172 3D  
D173 35  
D174 3B  
D175 21  
D176 02  
D177 37  
D178 1B  
D179 00  
D17A 1C  
D17B 13  
D17C 00  
D17D 6D  
D17E 01  
D17F 4D  
D180 07  
D181 27  
D182 00  
D183 1B  
D184 1C  
D185 67  
D186 FC  
D187 08  
D188 47  
D189 D9  
D18A 05  
D18B F7  
D18C 47  

133 *  
D152 2B  
D153 3C  
D154 A2  
D155 3B  
D156 OD  
D157 D1  
D158 1C  
D159 4C  
D15A 00  
D15B 4C  
D15C EC  
D15D EC  
D15E 02  
D15F 4C  
D160 4C  
D161 4C  
D162 02  
D163 47  
D164 19  
D165 1A  
D166 37  
D167 DD  
D168 03  
D169 67  
D16A 3D  
D16B 35  
D16C 3B  
D16D 21  
D16E 02  
D16F 23  
D170 24  
D171 23  
D172 3D  
D173 35  
D174 3B  
D175 21  
D176 02  
D177 37  
D178 1B  
D179 00  
D17A 1C  
D17B 13  
D17C 00  
D17D 6D  
D17E 01  
D17F 4D  
D180 07  
D181 27  
D182 00  
D183 1B  
D184 1C  
D185 67  
D186 FC  
D187 08  
D188 47  
D189 D9  
D18A 05  
D18B F7  
D18C 47  

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

--- END ASSEMBLY ---

TOTAL ERRORS: 00
RELOCATION SUBROUTINE

1. DEFINE BLOCKS

^Y IS CTRL-Y

2. FIRST SEGMENT

3. SUBSEQUENT SEGMENTS

RELOCATION SUBROUTINE EQUATES

R1L EQU $02 SWEET 16 REG 1.
INST EQU $08 3-BYTE INST FIELD.
LENGTH EQU $2F LENGTH CODE
YSAV EQU $34 CMND BUF POINTER
AIL 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 $FB9E ;DISASSEMBLER ENTRY
NXTA4 EQU $FCB4 POINTER INCR SUBR
FRMEOQ EQU $01 SOURCE BLOCK BEGIN
FRMEND EQU $02 SOURCE BLOCK END
TOBEG EQU $04 DEST BLOCK BEGIN
ADR EQU $06 ADR PART OF INST.
$D4DC RELOCATION SUBROUTINE

ORG $A4DC

LDA IN.Y  NEXT CMD CHAR
CMP #$AA  '*'?
BNE RELOC2  NO, RELOC CODE SEG.

INC YSAV ADVANCE POINTER.

LDY YSAV CMND BUF POINTER
LDA IN,Y NEXT CMD CHAR
CMP #$AA  '*'?
BNE RELOC2  NO, RELOC CODE SEG.

INC YSAV ADVANCE POINTER.

LDA IN,Y NEXT CMD CHAR
CMP #$AA  '*'?
BNE RELOC2  NO, RELOC CODE SEG.

INC YSAV ADVANCE POINTER.

LDY #402
GETINS LDA (A4L),Y COPY 3 BYTES TO
STA INST,Y SW16 AREA

DEY
JMP GETINS

LDX #400
STINST LDX #400
STINS2 LDA INST,X
STA (A4L),Y COPY LENGTH BYTES
INX OF INST FROM SW16 AREA TO
JSR NXTA4

--- END ASSEMBLY ---

TOTAL ERRORS: 00
1 ***********************************************************
2 * TAPE VERIFY *
3 * *
4 * JAN 78 *
5 * BY WOZ *
6 * *
7 * *
8 * *
9 ***********************************************************
10 *
11 * TAPE VERIFY EQUATES *
12 *
13 *
14 CHKSUM EQU $2E
15 A1  EQU $3C
16 HIMEM EQU $4C ; BASIC HIMEM POINTER
17 PP   EQU $CA ; BASIC BEGIN OF PROGRAM
18 PRLN EQU $CE ; BASIC PROGRAM LENGTH
19 XSAVE EQU $DB ; PRESERVE X-REG FOR BASIC
20 HDRSET EQU $F11E ; SETS TAPE POINTERS TO $CE.CF
21 PRGSET EQU $F12C ; SETS TAPE POINTERS FOR PROGRAM
22 NXTA EQU $FCB ; INCREMENTS (A1) AND COMPARES TO (A2)
23 HEADR EQU $FFC9
24 RDBYTE EQU $FCEC
25 RD2BIT EQU $FCFA
26 RDBIT EQU $FCFD
27 PRA   EQU $FD92 ; PRINT (A1) -
28 PRBYTE EQU $FDAA
29 COUT  EQU $FDED
30 FINISH EQU $FF26 ; CHECK CHECKSUM; RING BELL
31 PRERR EQU $FF2D
32 *
33 * TAPE VERIFY ROUTINE *
34 *
35 *
36 ORG $D535
37 OBJ  $A535
38 VFYBSC STX %SAVE ; PRESERVE X-REG FOR BASIC
39 SEC
40 LDX $$FF
41 GETLEN LDA HIMEM+1 ; CALCULATE PROGRAM LENGTH
42 SBC PP+1,X ; INTO PRLN
43 STA PRLN+1,X
44 INX
45 BEQ GETLEN
46 HDRSET ; SET UP POINTERS
47 JSR TAPEVFY ; DO A VERIFY ON HEADER
48 LDX $01 ; PREPARE FOR PRGSET
49 JSR PRGSET ; SET POINTERS FOR PROGRAM VERIFY
50 JSR TAPEVFY
51 LDX XSAD ; RESTORE X-REG
52 RTS

82
TAPE VERIFY RAM IMAGE (A1, A2)

D54  20 FA FC 56 TAPEVFY JSR RD2BIT
D55  A9 16 57 LDA #$16
D55  20 C9 FC 58 JSR HEADR ;SYNCHRONIZE ON HEADER
D55  B5 2E 59 STA CHKSUM ;INITIALIZE CHKSUM
D55  20 FA FC 60 JSR RD2BIT
D56  A0 24 61 VRFY2 LDY #$24
D56  20 FD FC 62 JSR RDBIT ;CARRY SET IF READ A '1' BIT
D56  20 FD FC 63 JRFY3 JSR RDBYTE ;READ A BYTE
D56  20 FD FC 64 JRFY3 JSP RDBYTE ;READ A BYTE
D56  A0 3B 65 LDY 4003B
D56  20 EC FC 66 JSR RDBYTE ;READ A BYTE
D56  20 FD FC 67 JSR RDBYTE ;READ A BYTE
D56  AO 24 68 VRFY2 LDY #$24 ;ONE LESS THAN USED IN READ FOR EXTRA 12
D56  20 FD FC 69 JSR RDBYTE ;READ A BYTE
D56  BO F9 70 BCS VRFY2 ;CARRY SET IF READ A '1' BIT
D56  20 FD FC 71 JSR RDBYTE ;READ A BYTE
D56  AO 3B 72 LDY #034 ;ONE LESS THAN USED IN READ FOR EXTRA 12
D56  90 FO 73 BCC VRFY3 ;LOOP UNTIL A1>A2
D56  4C 26 FF 74 JMP FINISH ;VERIFY CHECKSUM&RING BELL
D56  EA 74 EXTDEL NOP ;EXTRA DELAY TO EQUALIZE TIMING
D56  EA 75 NOP ; (+12 USEC)
D56  EA 76 NOP
D56  C1 3C 77 CMP (A1,X) ;BYTE THE SAME?
D56  90 FO 78 BEQ VRFY2 ;IT MATCHES, LOOP BACK
D56  48 79 PHA ;SAVE WRONG BYTE FROM TAPE
D56  B0 2D FF 80 JSR PRERR ;PRINT "ERR"
D56  B0 92 FD 81 JSR PRBYTE ;OUTPUT (A1)"-
D56  B1 3C 82 LDA (A1),Y
D56  B0 92 FD 83 JSR PRBYTE ;OUTPUT CONTENTS OF A1
D56  B0 DA FD 84 LDA #$A0 ;PRINT A BLANK
D56  B0 ED FD 85 JSR COUT
D56  A9 A9 86 LDA #$A9 ;'('
D56  B0 ED FD 87 JSR COUT
D56  68 BB 88 PLA ;OUTPUT BAD BYTE FROM TAPE
D56  B0 DA FD 89 JSR PRBYTE
D56  A9 A9 90 LDA #$A9 ;')'
D56  B0 DA FD 91 JSR COUT
D56  4C ED FD 92 LDA #$A9 ;CARRIAGE RETURN, AND RETURN TO CALLER
D56  81 3C 93 LDA (A1),Y
D55  4C 26 FF 94 JMP COUT

--- END ASSEMBLY ---

TOTAL ERRORS: 00
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
R4L EQU $8 AUX ADR POINTER.
R4H EQU $9
R5L EQU $A AUX ADR POINTER.
R5H EQU $B
R6L EQU $C GALLOP BIT MASK.
R6H EQU $D ($0001 TO $000F)
YSAV EQU $34 MONITOR SCAN INDEX.
AIH EQU $3D BEGIN TEST BLOCK ADR.
A2L EQU $3E LEN (PAGES) FROM MON.
SETCLY EQU $D5B0 ;SET UP CNTRL-Y LOCATION
PRBYTE EQU $FDAD BYTE PRINT SUBR.
COUT EQU $FDED CHAR OUT SUBR.
PRERR EQU $FF2D PRINTS 'ERR-BELL'
BELL EQU $FF3A
36 * RAMTEST:
37 *
38 *
39 ORG $D5BC
40 OBJ $A5BC
41 SETUP LDA $$C3 ;SET UP CNTRL-Y LOCATION
42 LDY $$D5
43 JMP SETCTLY
44 RAMTST LDA $00 ;TEST FOR $00,
45 JSR TEST
46 LDA $FF THEN $FF.
47 JSR TEST
48 JMP BELL
49 TEST STA DATA
50 EDR $0FF
51 STA NDATA
52 LDA A1H
53 STA R3H INIT (R3L,R3H),
54 STA R4H (R4L,R4H), (R5L,R5H)
55 STA R5H TO TEST BLOCK BEGIN
56 LDY $00 ;ADDRESS.
57 STY R3L
58 STY R4L
59 STY R5L
60 LDX $2L LENGTH (PAGES).
61 LDA DATA
62 TESTO1 STA (R4L),Y SET ENTIRE TEST
63 INY BLOCK TO DATA.
64 JSR PRBYTE PRINT ADDRESS,
65 INC R4H
66 DEX
67 BNE TESTO1
68 LDX $2L
69 TESTO2 LDA (R3L),V VERIFY ENTIRE TEST BLOCK.
70 CMP DATA TEST BLOCK.
71 BEQ TESTO3
72 PHA PRESERVE BAD DATA.
73 LDA R3H
74 JSR PRBYTE PRINT ADDRESS,
75 TYA
76 JSR PRBYSP SET DATA THEN EXPECTED DATA,
77 JSR PRBYSP SET NDATA AND R6
78 PLA THEN BAD DATA.
79 JSR PRBYCR THEN 'ERR-BELL'.
80 JSR PRBYTE PRINT ADDRESS,
81 TESTO3 INY
82 BNE TESTO2
83 INC R3H
84 DEX
85 BNE TESTO2
86 LDX $2L LENGTH.
87 TESTO4 LDA NDATA
88 STA (R5L),Y SET TEST CELL TO
89 STY R6H NDATA AND R6
90 STY R6L (GALLOP BIT MASK)
91 INC R6L TO $0001.
92 TESTO5 LDA NDATA
93 JSR TEST6 GALLOP WITH NDATA.
94 LDA DATA
95 JSR TEST6 THEN WITH DATA.
96 ASL R6H SHIFT GALLOP BIT
97 JSR TEST6 MASK FOR NEXT
D633 C5 3E 99 CMP A2L NEIGHBOR. DONE
D639 90 EC 100 BCC TEST05 IF > LENGTH.
D639 A5 00 101 LDA DATA
D639 91 0A 102 STA (R5L),Y RESTORE TEST CELL.
D63B E6 0A 103 INC R5L
D63D 0D 60 104 BNE TEST04
D63F E6 0B 105 INC R5H INCR TEST CELL.
D641 CA 0A 106 DEX POINTER AND DECR
D642 D0 D5 107 BNE TEST04 LENGTH COUNT.
D644 60 108 RTS1 RTS
D645 80 05 109 TEST6 STA TESTD SAVE GALLOP DATA.
D647 A5 0A 110 LDA R5L
D649 45 0C 111 EOR R6L SET R4 TO R5
D64B 85 0B 112 STA R4L EX-OR R6
D64D A5 0B 113 LDA R5H FOR NEIGHBOR
D64F 45 0D 114 EOR R6H ADDRESS (1 BIT
D651 85 09 115 STA R4H DIFFERENCE).
D653 A5 02 116 LDA TESTD
D655 80 02 117 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 RTS1 RTS1 (OK).
D65D 45 F7 121 STA TESTD STA TESTD SAVE GALLOP DATA.
D65F 91 0A 122 LDA R5L
D661 45 OD 123 EOR R6H ADD R5L DIFFERENCE.
D663 85 08 124 STA R4H FOR NEIGHBOR
D665 91 08 125 STA (R4L),Y GALLOP TEST DATA.
D667 A5 01 126 LDA R5H
D669 20 DA FD 127 JSR PRBYTE PRINT TEST CELL
D66C A5 0F 128 LDA R5L
D66E A5 02 129 LDA R5H
D66F 20 DA FD 130 JSR PRBYTE PRINT TEST CELL
D671 A5 OA 131 LDA R4H
D673 A5 09 132 LDA R5L
D675 A5 08 133 LDA R4L THEN NEIGHBOR ADR.
D677 20 DA FD 134 JSR PRBYTE
D679 A5 02 135 LDA TESTD THEN GALLOP DATA.
D67B A5 DB 136 JSR PRBYTE OUTPUT BYTE, SPACE.
D67D 20 2D FF 137 JSR PRBYTE THEN 'ERR-BELL'
D67F A9 8D 138 LDA #BD ASCII CAR. RETURN.
D681 4C ED FD 139 JMP COUT
D683 A9 A0 140 JSR PRBYTE
D685 4C ED FD 141 JMP COUT SPACE.
D687 4C ED FD 142 JMP COUT SPACE.
D689 4C ED FD 143 JMP COUT SPACE.
D68B 4C ED FD 144 JSR PRBYTE
D68D 4C C3 D5 144 USRLC JMP RAMTST ENTRY FROM MON (CTRL-Y)

--- END ASSEMBLY ---

TOTAL ERRORS: 00

86
MUSIC SUBROUTINE

GARY J. SHANNON

ORG $D717

ZERO PAGE WORK AREAS
PARAMETER PASSING AREAS

DOWNTIME EGU $0
UPTIME EGU $1
LENGTH EGU $2
VOICE EGU $2FD
LONG EGU $3FE
NOTE EGU $3FF
SPEAKER EGU $C030

ENTRY JMP LOOKUP
PLAY ONE NOTE
DUTY CYCLE DATA IN 'UPTIME' AND 'DOWNTIME', DURATION IN 'LENGTH'
CYCLE IS DIVIDED INTO 'UP' HALF AND 'DOWN' HALF

PLAY LDY UPTIME ; GET POSITIVE PULSE WIDTH
LD A SPEAKER ; TOGGLE SPEAKER
INC LENGTH ; DURATION
INC LENGTH+1
INC UPTIME
RTS ; DURATION EXPIRED

PATH1 NOP ; DUMMY
DEY ; DECREMENT WIDTH
BNE DOWN ; WIDTH EXPIRED

PLAY2 LDY DOWNTIME ; GET NEGATIVE PULSE WIDTH
LDA SPEAKER ; TOGGLE SPEAKER
INC LENGTH ; DURATION
INC LENGTH+1
INC DOWNTIME
RTS ; DURATION EXPIRED

PATH2 BNE PLAYS ; REPEAT
PLAY ; BACK TO UP-SIDE

PATH3 BNE PLAY2 ; SAME # CYCLES

PATH4 NOP ; DUMMY
PLAY3 INC LENGTH ; DURATION
PATH5 BNE PLAY4 ; NOT EXPIRED
INC LENGTH+1
INC PATH5
RTS ; DURATION EXPIRED

PATH6 ; USE UP SOME CYCLES
PLAY3 ; REPEAT

87
NOTE TABLE LOOKUP SUBROUTINE

GIVEN NOTE NUMBER IN 'NOTE'
DURATION COUNT IN 'LONG'
FIND 'UPTIME' AND 'DOWNTIME'
ACCORDING TO DUTY CYCLE CALLED
FOR BY 'VOICE'.

D74E AD FF 02 71 LOOKUP LDA NOTE ; GET NOTE NUMBER
D751 0A 72 ASL ; DOUBLE IT
D752 AB 73 TAY
D753 B9 9E D7 74 LDA NOTES,Y / GET UPTIME
D756 B5 00 75 STA DOWNTIME ; SAVE IT
D758 AD FD 02 76 LDA VOICE ; GET DUTY CYCLE
D75A 4A 77 SHIFT LSR
D75C F0 04 78 BEQ DONE ; SHIFT WIDTH COUNT
D75E 46 00 79 LSR DOWNTIME ; ACCORDING TO VOICE
D760 D0 F9 80 BNE SHIFT
D762 B9 9E D7 81 DONE LDA NOTES,Y ; GET ORIGINAL
D765 38 82 SEC
D766 E5 00 83 SBC DOWNTIME ; COMPUTE DIFFERENCE
D768 B5 01 84 STA UPTIME ; SAVE IT
D76A C8 85 INY ; NEXT ENTRY
D76B B9 9E D7 86 LDA NOTES,Y ; GET DOWNTIME
D76E 65 00 87 ADC DOWNTIME ; ADD DIFFERENCE
D770 B5 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 B5 03 92 STA LENGTH+1 MOST SIGNIFICANT BYTE
D77A A9 00 93 LDA #0
D77C B5 02 94 STA LENGTH
D77E A5 01 95 LDA UPTIME
D780 D0 98 96 BNE PLAY ; IF NOT NOTE #0, PLAY IT
D782 EA 102 REST NQP ; DUMMY
D783 EA 103 NQP ; CYCLE USERS
D784 4C 87 D7 104 JMP REST2 ; TO ADJUST TIME
D787 E6 02 105 REST2 INC LENGTH
D789 D0 05 106 BNE REST3
D78B E6 03 107 INC LENGTH+1
D78D D0 05 108 BNE REST4
D78F 60 109 RTS ; IF DURATION EXPIRED
D790 EA 110 REST3 NQP ; USE UP 'INC' CYCLES
D791 4C 94 D7 111 JMP REST4
D794 4C EC 112 REST4 BNE REST ; ALWAYS TAKEN
<table>
<thead>
<tr>
<th>Hex</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00, 00, F6, F6, E8, E8, DB, DB</td>
</tr>
<tr>
<td>CF, CF, C3, BB, BB, AE, AE</td>
<td></td>
</tr>
<tr>
<td>A4, A4, 9B, 9B, 92, 92, 8A, 8A</td>
<td></td>
</tr>
<tr>
<td>82, 82, 7B, 7B, 74, 74, 6D, 6E</td>
<td></td>
</tr>
<tr>
<td>67, 6B, 61, 62, 5C, 5C, 57, 57</td>
<td></td>
</tr>
<tr>
<td>52, 52, 4D, 4E, 49, 49, 45, 45</td>
<td></td>
</tr>
<tr>
<td>41, 41, 3D, 3E, 3A, 3A, 36, 37</td>
<td></td>
</tr>
<tr>
<td>33, 34, 3D, 31, 2E, 2E, 2B, 2C</td>
<td></td>
</tr>
<tr>
<td>29, 29, 26, 27, 24, 25, 22, 23</td>
<td></td>
</tr>
<tr>
<td>20, 21, 1E, 1F, 1D, 1D, 1B, 1C</td>
<td></td>
</tr>
<tr>
<td>1A, 1A, 1B, 19, 17, 17, 13, 16</td>
<td></td>
</tr>
<tr>
<td>14, 15, 13, 14, 12, 12, 11, 11</td>
<td></td>
</tr>
<tr>
<td>10, 10, 0F, 10, 0E, 0F</td>
<td></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 and Data)
93 Relocate (Machine Code and Data)
94 RAM Test
94 Music
95 High-Resolution Graphics
96 Quick Reference to High-Resolution Graphics Information
Chapter 1: RENUMBER

(a) To renumber an entire BASIC program:

CLR
START = 1000
STEP = 10
CALL -10531

(b) To renumber a program portion:

CLR
START = 200
STEP = 20
FROM = 300
TO = 500
(program portion to be renumbered)

CALL -10521

Chapter 2: APPEND

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

LOAD

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

CALL -11076

Chapter 3: TAPE VERIFY (BASIC)

(a) Save current BASIC program on tape:

SAVE

(b) Replay the tape, after:

CALL -10955
Chapter 4: TAPE VERIFY (Machine Code and Data)

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

```
address1 . address2  W  return
```

(b) Initialize Tape Verify feature:

```
D52EG  return
```

(c) Replay the tape, after:

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

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

Chapter 5: RELOCATE (Machine Code and Data)

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

```
D4D5G  return
```

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

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

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

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

If first program Segment is data, Move:

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

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

```
. Source Segment End  ctrl Y  return
```

and Relocate subsequent contiguous code Segments:

```
. Source Segment End  M  return
```

Note: spaces shown within the above commands are for easier reading only; they should not be typed.
Chapter 6: RAM TEST

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

D5BCG return

(b) To test a portion of memory:

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

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

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

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

Example, for a 48K system:

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

(d) To repeat test indefinitely:

N complete test 34:0 type one space return

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

Chapter 7: MUSIC

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

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

(b) Set parameters for next note:

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

(c) Sound the note:

CALL MUSIC
Chapter 8: HIGH-RESOLUTION GRAPHICS

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

1  XO = YO = COLR
2  SHAPE = ROT = SCALE  (if shapes are used)

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

10 INIT = -12288 : CLEAR = -12274 : BKGND = -11471
11 POSN = -11527 : PLOT = -11506 : LINE = -11500
12 DRAW = -11465 : DRAW1 = -11462
13 FIND = -11780 : SHLOAD = -11335

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

20 BLACK = 0 : LET GREEN = 42 : VIOLET = 85
21 WHITE = 127 : ORANGE = 170 : BLUE = 213
22 BLACK2 = 128 : 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 COLR = VIOLET : CALL BKGND : REM TURN BACKGROUND VIOLET
50 FOR I = 0 TO 279 STEP 5
60 XO = 140 : YO = 150 : COLR = WHITE : REM SET PARAMETERS
70 CALL POSN : REM MARK THE "CENTER"
80 XO = I : YO = 0 : REM SET NEW PARAMETERS
90 CALL LINE : REM DRAW LINE TO EDGE
100 NEXT I : END
## Quick Reference to High-Resolution Information

<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>CALLing Address</th>
<th>Parameters Needed</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</td>
</tr>
<tr>
<td>PLOT</td>
<td>-11506</td>
<td>X₀, Y₀, COLR</td>
</tr>
<tr>
<td>LINE</td>
<td>-11500</td>
<td></td>
</tr>
<tr>
<td>DRAW</td>
<td>-11465</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**: POKE -16302, 0
- **Mixed Graphics-Plus-Text (Default)**: POKE -16301, 0
- **Page 2 Display**: POKE -16299, 0
- **Page 1 Display (Normal)**: POKE -16300, 0
- **Page 2 Plotting**: POKE 806, 64
- **Page 1 Plotting (Default)**: 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**: PEEK (810)

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

96