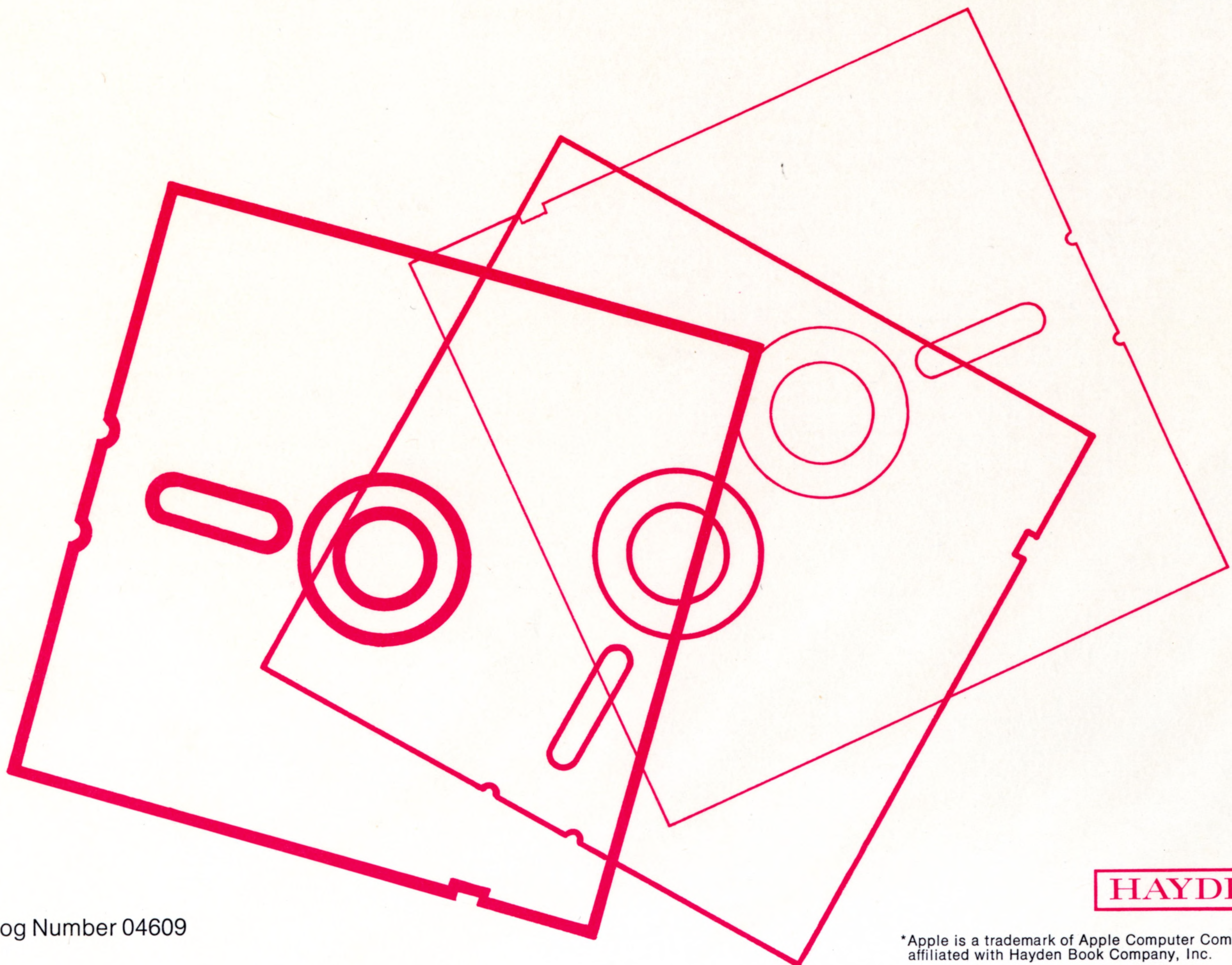
Apple II 24K

HAYDEN MICROCOMPUTER PROGRAM DISKS

Apple™ Assembly Language Development System: An Assembler/Editor/Formatter

Lets you write and modify machine language programs quickly and easily.

Paul Lutus



Catalog Number 04609

HAYDEN

*Apple is a trademark of Apple Computer Company, Inc., and is not affiliated with Hayden Book Company, Inc.

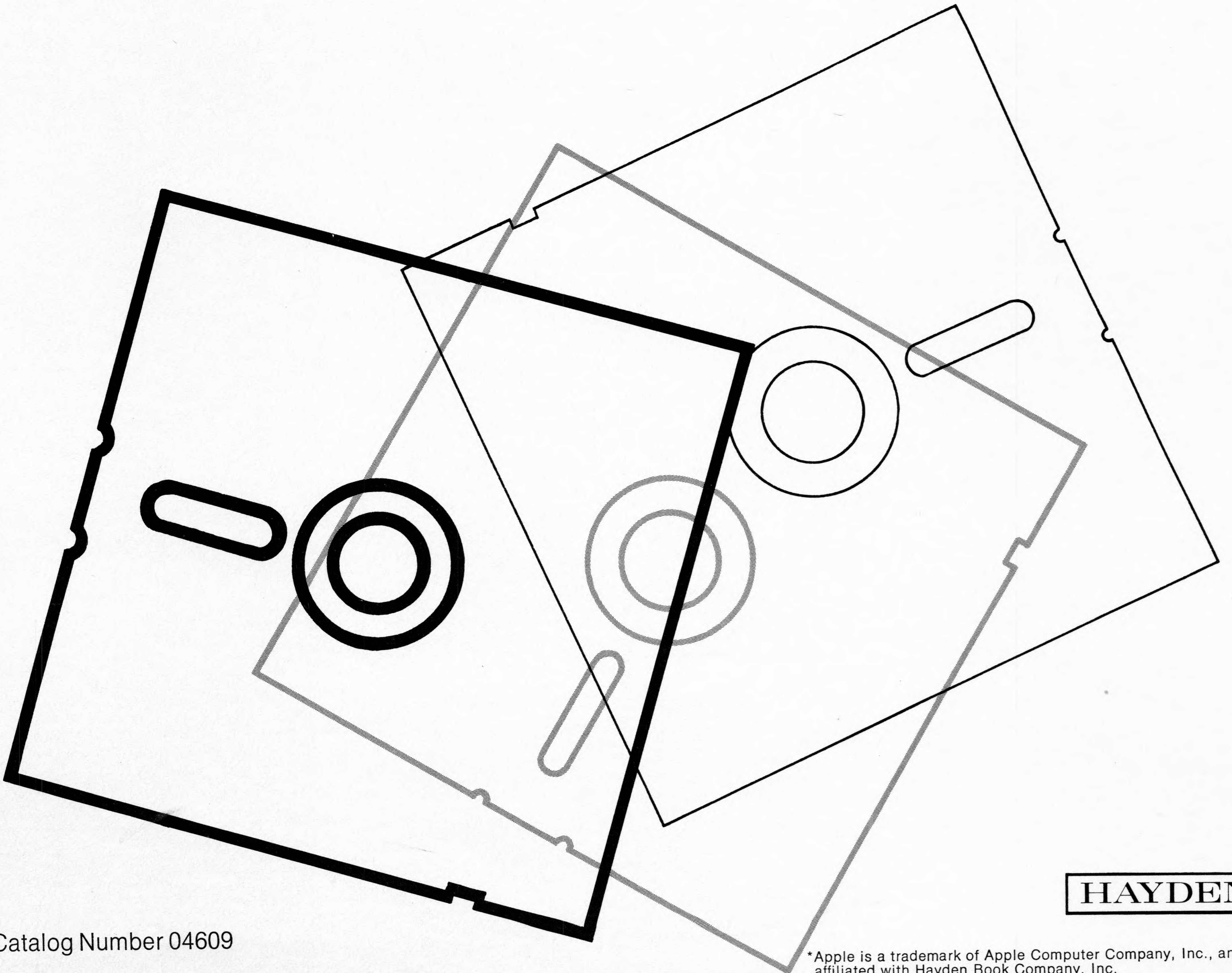
Apple II 24K

HAYDEN MICROCOMPUTER PROGRAM DISKS

Apple™ Assembly Language Development System: An Assembler/Editor/Formatter

Lets you write and modify machine language programs quickly and easily.

Paul Lutus



HAYDEN

Catalog Number 04609

*Apple is a trademark of Apple Computer Company, Inc., and is not affiliated with Hayden Book Company, Inc.

Copyright © 1980 by HAYDEN BOOK COMPANY, INC. All rights reserved.

Contents

Part 1. Introduction 1

What Do You Say to a Naked Computer? *1*

Machine language – High-level languages – Interpretation – Algorithms – Source file – Object file – Assembly – Listing

Hardware Requirements *2*

Part 2. Use of the Editor Program 3

Getting Started *3*

Editor Entry Format *3*

Label field – Operator field – Operand field – Comment field – Space bar tabulation – Deletions

Cursor Control Mode *4*

Controling the cursor – Fast – Free memory – Saving the file to disk

Advanced Editing Features 7

String trace and replacement – ConTRoL S – ConTRoL R – File segment to disk – ConTRoL K – Disk-based macroinstructions – ConTRoL I

Global and Local Labels *9*

Special Entry Types *10*

String – Hexadecimal data – ASCII mode – Alphabetical variable – Offset addressing – Label definition

Part 3. Assembling 13

Origin for assembly – Offset assembly – Process – Error diagnosis – Labels and numbers – Saving object file to disk

Part 4. Creating a Formatted List of the Program 15

Slot identification – Printer preparation – Format of print – Synchronization of source and object files

Part 5. General Characteristics 16

Relative density of files – Source file preservation

Appendix A. Serial Interface 17

PART 1

Introduction

(If you are familiar with the terminology of assembly language, you may pass up this first section.)

WHAT DO YOU SAY TO A NAKED COMPUTER?

In the early days of computing, programmers were required to use very rudimentary instructions in the writing of programs. These instructions were immediately understandable by the machine. There was no need to translate or otherwise modify them before they were carried out. These instructions were called *machine language instructions*.

Since that time, languages have been developed that are easier for humans to work with. These are called *high-level languages*. Such languages usually use more words and fewer numbers than are used in machine language. In general, a person reading a high-level language listing understands the meaning of the instructions much more readily than the equivalent machine language listing. Also, in high-level languages many operations may be combined into a single instruction. By comparison, machine language requires the use of many instructions, each very simple, to do something.

BASIC is such a high-level language that the computer does not inherently understand it. BASIC instructions must be converted, by one means or another, into the equivalent machine language instructions (or sets of instructions) before they are carried out. In the Apple II and other small computers this conversion is called *interpretation.* Interpretation involves taking a BASIC instruction and finding the equivalent machine language instructions in a large set of stored *algorithms*. After the correct algorithm is found, it is carried out. Then the next instruction is fetched from the BASIC program and the process is repeated.

Interpretation is quite slow when compared with writing a machine language program to accomplish the purposes of the BASIC program. This is because the time the interpreter takes to find the correct machine language equivalent is frequently greater than the time required to carry out the instruction. As a result, BASIC programs are from 10 to 50 times slower in operation than the equivalent machine language programs.

If a program is used frequently, is well-defined, does not need to be modified very much or very often, then it is probable that it should be written in machine language rather than BASIC.

The Assembly Language Development System described here makes possible the creation of such machine language programs. But this does NOT mean that the user must use cryptic numbers in writing the programs. This is because there is an intermediate language between machine language and BASIC. This is called *assembly language*.

Courses of instruction in assembly language are offered in human and book form, just as they are for BASIC. It is necessary to obtain such instruction before attempting to write assembly language programs.

In the Assembly Language Development System being described, the user may:

1. Create a *source file* of assembly language instructions to accomplish the desired task(s). The source file is created using the *Editor program.*
2. Assemble an *object file* of machine language instructions using the source file as a guide. This process, called *assembly*, uses the *Assembler program.*
3. Test and evaluate the machine language program in the object file, and, if necessary, repeat steps 1 and 2 to make modifications or corrections.

4. Finally, when the program is working correctly and lots of comments have been added to the source file, print a formatted listing of the source and object files using the *Formprint program.* This step is a very good idea, but is optional.

   Instructions for the use of each of these programs are provided here.

HARDWARE REQUIREMENTS

One Apple II computer with a least 24K of memory, one Apple II disk drive, and one television set or video monitor.

PART 2

Use of the Editor Program

GETTING STARTED

After connecting all the equipment listed above, turn on your Apple II. You will get the power-on display (which could be anything) and the machine language prompt (*). Insert your Assembly Language Development System disk in the disk drive. Now type the slot number into which you have plugged your disk drive (slot number **6** is standard). Then hold down the button marked **ConTRoL** on the keyboard and, while holding it down, type, at your option, either **P** or **K**. Release both buttons and hit the button marked **RETURN**. The disk drive will whirr and click a few times and the red IN USE light will come on. This automatically runs the Editor program. A menu will be presented.

Option (E) permits one to continue editing a source file in memory. This option is exercised while one is going back and forth between the Editor and the Assembler in the course of debugging and modifying a program.

Option (N) erases memory contents in preparation for the creation of a new source file, option (L) permits loading, and option (S) is for saving a source file to disk. If the user has loaded a source file, the next option to be exercised is (E), not (N), so that the just-loaded file will not be erased. Also, if hours are consumed creating an assembly language source file, it is a good idea to frequently save the file to disk (or disks), thereby backing up the contents of memory against loss.

As the file is saved to disk, the program automatically adds the prefix "SRC." to the chosen file name. This is so that the source file and the later developed object file may be saved using the same name. The user need not enter this prefix because it is done automatically.

Option (A) runs the Assembler program after the source file has been developed and remains in memory.

EDITOR ENTRY FORMAT

For this example, choose option (**N**) INITIALIZE. After this option has been exercised, any memory contents will be erased and there will be a blank screen with a cursor in the upper left. At this point, assembly language entries may be made.

Assembly language entries to the source file are divided into four fields:

(1)	(2)	(3)	(4)
LABEL	LDA	$1234	;GET NEXT VALUE

The fields shown are (1) LABEL, (2) OPERATOR, (3) OPERAND, and (4) COMMENT. A label is used if this line must be gotten to from elsewhere in the program. If there is no label needed, the user presses the space bar once, which automatically tabulates to the Operator field. In the Editor, the space bar is really a tabulation command. With each press of the space bar, the cursor moves to the next location on the Apple II display that is a multiple of 8 spaces from the left. This is a memory-saving measure because it takes fewer characters in memory to format a line of the source file.

It is important to remember the use of the space bar. If the line being entered requires no label, press the space bar before entering the Operator. If there is to be a label, enter the label and *then* press the space bar.

NOTE: The label may have up to 7 characters. Use of more characters will result in incorrect assembly. A global label (discussed below) may have 6 characters plus its special delimiter.

The pressing of the space bar brings the cursor to the Operator field, and the operator may be entered. After this, press the space bar once again. This time, the cursor moves one space, to the Operand field. Enter the operand, if there is one.

The final and optional field is the Comment, although some would disagree that it is optional. After entering the operand, press the space bar once again and the cursor will move to the beginning of the Comment field.

There is a special character which must begin the Comment field. This character is the semicolon (;). Among other things, the semicolon prevents the tabulation feature from working, so that words in the Comment field will be separated by only one space. It is permitted to type comments starting either at the Label or the Operator fields if desired, which makes it possible to put in large comments. These comments also must begin with the semicolon. On such a line, there is no need for any assembly language entries.

At the end of your entries for the line press the **RETURN** key, and the cursor will place itself at the beginning of a new line. Each line must end with a carriage return. It is not permitted to space down to the next label entry point. It is important to realize that entry errors won't be detected until the source file is assembled by the Assembler program.

DELETING A CHARACTER

If you make a mistake in entry and want to delete the last entered *character*, press the **LEFT ARROW** key. This may be done repeatedly. If you make a mistake *correcting* a mistake, and want your character back, press the **RIGHT ARROW** key. This also may be done repeatedly.

DELETING A WORD

If you want to delete the last entered *word* (word = something preceded by a space), press **ConTRoL W** (press the **ConTRoL** key, and, while holding it down, press the **W** key. All **ConTRoL** keys are gotten this way). The cursor will move back to the last space, delete it, and stop. If you want this word back, press the **RIGHT ARROW** key repeatedly.

DELETING A LINE

If you want to delete the last entered *line* (line = something preceded by a carriage return), press **ConTRoL X**. The cursor will move back to the end of the previous line. To recover the deleted line, press the **RIGHT ARROW** key repeatedly. If you delete more than 256 characters without recovering any, the oldest entries will be overwritten sequentially by the new deletions.

MOVING THE CURSOR WITHOUT ENTRY

If you want to move the cursor around the screen without making any entries, press the **ESCape** key. When you do this, the cursor will change from a flashing blank square to a

flashing "+", which indicates the Cursor Control Mode. Now, to move the cursor around, use the following keys:

KEY	MOVES CURSOR
J	Left one character
K	Right one character
I	Up one line
M	Down one line

These keys may be used repeatedly. They may also be used with the **REPeaT** key for speed. One use for these keys is to review entries which have scrolled off the top of the screen. This is because the cursor is *always* somewhere on the screen, so if the cursor is moved up far enough, the screen display will scroll down to deep it on the screen. Another use for these keys is to add new lines between previous entries. Example (□ marks the cursor position):

You have entered:

```
LABEL     LDA  $0     ;GET FIRST
          ADC  #3     ;ADD 3
          STA  $1     ;SAVE RESULT
          RTS  □
```

You have forgotten to clear carry before performing the addition and a line needs to be added between the present first and second lines. Use the Cursor Control Mode to move the cursor as shown below (cursor = +):

```
LABEL     LDA  $0     ;GET FIRST+
          ADC  #3     ;ADD 3
          STA  $1     ;SAVE RESULT
          RTS
```

The cursor is shown as a + because the Cursor Control Mode is enabled. To disable this mode, press any character *other* than the four cursor control characters (**I**, **J**, **K**, **M**). When this is done the cursor will become blank, indicating that normal entries may be made. Now press **RETURN** to begin a new line. This will push the other lines down the screen. Now press the space bar once (because no label is needed) and type as shown:

```
LABEL     LDA  $0     ;GET FIRST
          CLC         ;CLEAR CARRY □
          ADC  #3     ;ADD 3
          STA  $1     ;SAVE RESULT
          RTS
```

This example shows that any new information may be added anywhere in the file simply by inserting it. Another use for this editing capability is to move words and lines around the screen. Example, you have entered:

```
LABEL     LDA  $0     ;GET FIRST
          STA  $1     ;SAVE RESULT
          CLC         ;CLEAR CARRY
          ADC  #3     ;ADD 3
          RTS
```

The second line should be the fourth. Position the cursor as shown below using the Cursor Control Mode (the cursor is +):

```
LABEL     LDA  $0     ;GET FIRST
          STA  $1     ;SAVE RESULT+
          CLC         ;CLEAR CARRY
          ADC  #3     ;ADD 3
          RTS
```

Now press the space bar to disable the Cursor Control Mode and press **ConTRoL X** (delete line). Result:

```
LABEL     LDA  $0    ;GET FIRST □
          CLC        ;CLEAR CARRY
          ADC  #3    ;ADD 3
          RTS
```

The line has been deleted. It is now stored for reuse in the *copy buffer*. Contents of the copy buffer are gotten back using the **RIGHT ARROW** key.

Now move the cursor to the position shown below:

```
LABEL     LDA  $0    ;GET FIRST
          CLC        ;CLEAR CARRY
          ADC  #3    ;ADD 3+
          RTS
```

Again, disable the Cursor Control Mode, and press the **RIGHT ARROW** key repeatedly to bring the deleted line from the copy buffer back onto the screen:

```
LABEL     LDA  $0    ;GET FIRST
          CLC        ;CLEAR CARRY
          ADC  #3    ;ADD 3
          STA  $1    ;SAVE RESULT □
          RTS
```

MOVING THE CURSOR FAST

If it is desired to move the cursor (and therefore the viewed segment) faster than is possible with the cursor controls, use the following method: **ConTRoL T** moves cursor *up* 12 lines and **ConTRoL V** moves it *down* 12 lines. It is not necessary to be in the Cursor Control Mode to use these keys.

If it is desired to jump to the beginning or the end of the file from the present cursor position, use **ConTRoL B** to move the cursor to the file *beginning*, and **ConTRoL E** moves it to the file *end*.

DETERMINING HOW MUCH MEMORY REMAINS

The source file memory is 25,088 ($6200) bytes long. To get remaining memory, press **ConTRoL F**. A display will be printed at the bottom of the screen:

<FREE MEMORY = $[number of remaining bytes]>

It is a good idea to stop making entries before running entirely out of memory. This will make possible small additions and changes in the future.

NOTE: to leave the edit mode and return to the menu, press **ESCape** followed by **ConTRoL Q**.

SAVING THE FILE TO DISK

It is a good practice to save the file to disk frequently so that there is a backup to the file in memory. If option ⟨**S**⟩ SAVE TO DISK is chosen, the program will ask for a file name. If a file name had been entered previously, this name will be presented. To use this name, press the **RIGHT ARROW** key repeatedly, moving the cursor over the name. This captures the displayed name for reuse. If a different file name is desired, type this name over the one being displayed.

NOTE: If it is desired to access a disk drive or slot other than that last accessed, include this information after the file name, as shown below:

[file name] ,D [drive number] ,S [slot number]

If there is a disk error, such as a mistyping of the file name, simply restart the editor program. DOS entry errors are the most frequent cause for unscheduled program exits, and the file in memory is well defended against loss in this case.

ADVANCED EDITING FEATURES/STRING TRACE AND REPLACEMENT

You may use this feature to view each example of an entered word or phrase (called a *string*), and selectively replace one such string with another. This feature may also be used to jump about in the file to selected locations without changing anything.

To enable the feature, press **ConTRoL S**. A bell will ring and you will be presented with:

```
STRING SEARCH & REPLACE
SEARCH ONLY, ENTER  /AS IS/
SEARCH & REPLACE, ENTER  /AS IS/TO BE/
```

The diagonals (/) are used to separate the search string (on the left) from the replacement string, if any (on the right): /The way it is now/The way you want it/

It is permitted to use the null string (//) as the replacement string. This simply deletes the target string and replaces it with nothing.

Any character may be used for the delimiter (/). The character used must not appear in either string. If you simply want to trace without making any changes, just enter one string: /Show me each of these/

If your entry includes a replacement string, the next display will be:

```
REPLACE (A) AUTOMATIC, OR (M) MANUAL
:
```

If you choose (**A**) AUTOMATIC, the program will seek out each case of the search string *following the present cursor position* and replace it. It will then return to the original cursor position. If you had chosen (**M**) MANUAL instead, this message would be presented:

```
TO REPLACE, PRESS CTRL R
TO CONTINUE TRACE PRESS RETURN
TO EXIT, TYPE ANYTHING ELSE
```

After pressing return, the search will begin, unless an entry error has been made. The most frequent errors are:

1. Making an input for trace with only one delimiter: /test
2. Making an input for replacement without trace with less than three delimiters, meaning no replacement string: /test/ or /test/replacement
3. Misplacing the delimiters in the entry: //test

If all is well, the program will begin searching for, and, in (M) MANUAL mode, displaying cases of, the target string.

NOTE: This function searches forward through the file for the larget string. It won't notice any cases above the present cursor position. In many cases it is a good idea to jump to the file beginning to use this feature.

If you want to replace a displayed string with the entered replacement, press **ConTRoL R**. If you want to go on to the next without replacing the displayed case, press **RETURN**. If you want to stop at the present position and leave the string mode, type anything other

than **ConTRoL R** or **RETURN**. This will put you in the normal editing mode.

When the end of the file has been reached, a bell will ring and a label will be printed at the bottom of the display:

<DONE>

Or, if the target string wasn't found at all,

<STRING NOT FOUND>
<DONE>

After one of these labels has been displayed the program will return to the cursor position at which the search began.

SAVING A FILE SEGMENT TO DISK

Let us say you want to save a segment of the entered file to disk for use elsewhere or reuse within the same file. Place the cursor at the end of the segment as shown:

```
LABEL    LDA  $0    ;GET FIRST
         CLC        ;CLEAR CARRY
         ADC  #3    ;ADD 3
         STA  $1    ;SAVE RESULT
         RTS  □
```

Now press **ConTRoL K** (for Keep). The program will ask for a marker which identifies the beginning of the segment. Remember: there must be no identical marker between the cursor and the desired marker or the program will only save to there. For example, if we had chosen the dollar sign as the marker, the saved file would be:

```
              $1    ;SAVE RESULT
         RTS
```

In some difficult cases, it may be necessary to create special markers which are later deleted. If the specified marker cannot be found by searching backwards through memory, a bell will sound and the following label will be printed:

<STRING NOT FOUND>

If the marker *is* found, the program will ask for a file name by which the file is to be saved.

DISK-BASED MACROINSTRUCTIONS

Use of the segment-saving feature just described, and the insertion feature about to be described, become the basis for the development of disk-based macroinstructions. Rather than writing frequently-used routines over and over again, the user writes them once and saves them to disk. Then, when the function is needed later in the program or in another program, this insertion method is used. For further information on this use, see the section below on "Global and Local Labels."

INSERTING A FILE, OR FILE SEGMENT, AT THE PRESENT CURSOR POSITION

It is desired to insert a disk file at the present cursor position. Because a disk file almost never begins with a carriage return, it is best to press **RETURN** once before using this feature. Now press **ConTRoL I** (for Insert). The program will ask for the desired file name. The file will be inserted at the present cursor position and, when the file is inserted, the cursor will appear at the end of the inserted segment.

WARNING: If the file being inserted is larger than remaining free memory, the file in memory will be destroyed. Always keep a backup file when using this procedure on large files where this outcome is possible.

GLOBAL AND LOCAL LABELS

The Assembly Language Development System incorporates a powerful feature that makes it possible to create a functional module which is independent of those around it. Such modules may be relocated anywhere in the file, or in another file, without conflicts caused by multiple use of label names. Consider the following example:

```
ADD     LDY  #8
        CLC
LOOP    LDA  $0,Y
        ADC  $10,Y
        STA  $20,Y
        DEY
        BPL  LOOP
        RTS
SUB     LDY  #8
        SEC
LOOP    LDA  $0,Y
        SBC  $10,Y
        STA  $20,Y
        DEY
        BPL  LOOP
        RTS
```

One of the tasks of the Assembler is to find addresses or numeric equivalents for each label used, so that this value may be used when the label is called from an Operand field. In the example shown, there are two uses of the label "LOOP". This will result in incorrect assembly. More exotic examples could be created in which 2000-line programs must be laboriously designed so that no two labels are alike, and any additions or modifications to the program must be done with new, unused labels. To avoid this problem and to make it possible to have portable program modules, the concept of global and local labels is introduced. Example:

```
*ADD    LDY  #8
        CLC
LOOP    LDA  $0,Y
        ADC  $10,Y
        STA  $20,Y
        DEY
        BPL  LOOP
        RTS
*SUB    LDY  #8
        SEC
LOOP    LDA  $0,Y
        SBC  $10,Y
        STA  $20,Y
        DEY
        BPL  LOOP
        RTS
```

The *functional modules* "ADD" and "SUB" are now separate. The two labels which start the modules are preceded by the asterisk (*). The asterisk identifies a *global* label.

A *global* (*) label may be called from *anywhere* in the program. If a global label is called in a line's Operand field, the Assembler searches through all global labels for the corresponding address or numeric value. All global labels begin with (*). Each global label must be unique in the program.

A *local* label may be sought only between the two nearest global labels. This means that common names like LOOP and NEXT may be used over and over again in the program, each used between two global labels. Therefore, the previous example will assemble correctly, because the two LOOP labels are separated by global labels and therefore cannot be confused with one another. The following example will *not* work:

```
*ADD      LDY  #8
          CLC
LOOP      LDA  $0,Y
          ADC  $10,Y
*SAVE     STA  $20,Y
          DEY
          BPL  LOOP
          RTS
*SUB      LDY  #8
          SEC
LOOP      LDA  $0,Y
          SBC  $10,Y
*STORE    STA  $20,Y
          DEY
          BPL  LOOP
          RTS
```

This example will not work because the Operand field calls for the LOOP label are separated from their respective local labels by global labels. Therefore, the Assembler will not find the LOOP label at all, and assembly will halt.

This feature makes possible the development of portable functional modules, which, because of the segregation ability of global labels, may be relocated within the same program or in another program (using the segment save and load to disk features discussed previously). These functional modules may be called by name from disk while writing assembly language programs.

Such a functional module is called a disk-based macroinstruction because it is possible to put together a program very quickly by calling the modules from disk as required, using the INSERT feature.

SPECIAL ENTRY TYPES

The Assembler will accept the following special entry types and convert them as specified:

1. String entry. The Assembler will accept the following:

```
STRING     "THIS IS A TEST STRING"
"ANOTHER TEST STRING LOCATION"
```

As shown, the string may begin at the Label or the Operator field. Strings may also be entered as long, single entries. Such long entries may not exceed 6 lines (or 256 characters) without a close quote and a carriage return. Very long strings may be entered by breaking the entry up with quotes and carriage returns each 6 lines of entry.

Upon assembly, the equivalent ASCII numeric values are gotten for each character and inserted in the object file at the string location.

NOTE: The special character "] " (**Shift M**) is replaced with a carriage return during assembly. This makes it possible to embed carriage returns in long strings which need format control.

An example program follows:

```
*PRINT    LDY  #0           ;CLEAR INDEX
LOOP      LDA  STRING, Y    ;GET NEXT CHAR
          JSR  $FDED        ;MONITOR PRINT
          INY               ;INDEX TO NEXT
          CMP  '?           ;END OF STRING?
          BNE  LOOP         ;GET MORE CHARS
          RTS               ;GO BACK
STRING    "]] ENTER YOUR NAME?"
```

The use of "CMP '?" in the example will be explained below under ASCII mode. The demonstration program will generate the two embedded carriage returns (]]), then print the rest of the string including the question mark.

2. Hexadecimal data entry:

```
DATA      .0 1 2 3 4 5 6 7 8 9
.A B C D E F 10 11 12 13 14 15
```

The identifier for this mode of entry is the period. The hexadecimal numbers are inserted by the Assembler at the location of the label data. As was the case with strings, either of the starting points shown may be used, and long sets of such data may be entered.

3. ASCII mode:

```
TEST      LDA  $0           ;GET TEST CHAR
          CMP  'A           ;IS IT "A"?
          BEQ  TRUE         ;YES
          BNE  FALSE        ;NO
```

This mode replaces a single character with the equivalent ASCII number during assembly. The identifier is the apostrophe, which must precede the character to be converted. The character is always treated as a number, not an address. As in string mode, if the character "]" is used, it will be converted to a carriage return on assembly.

4. Alphabetic variable mode:

```
MOVE      LDA  %A           ;MOVE THE DATA
          STA  %C
          LDA  %B
          STA  %D
          RTS
```

This mode takes advantage of the fact that there are 26 unused locations in Apple II page zero (hex $0 – $19), which are assigned labels and used as easily identified alphabetic variables, just as in BASIC. The identifier for this mode is the percent sign (%). Only single letters may be used.

5. Offset addressing:

```
SET       LDA  ADDR
          STA  DEST+$1
          STA  DEST-$3
          LDA  ADDR-$4
          STA  DEST+$1F
```

This mode is used to compute offset addresses during assembly rather than requiring run-time routines to obtain the addresses. Addition or subtraction are the permitted modes, the numeric entry is hexadecimal. The following is a special case which returns the *numeric value*, rather than the contents, of an address:

```
FIND        LDA  #ADDR
            STA  ADRLOW
            LDA  #ADDR+$1
            STA  ADRHIGH
```

In this immediate mode, 2-byte addresses may be gotten. The two displayed cases, no modifier and "+$1", are the only permitted cases because addresses are no longer than 2 bytes.

6. Label definition mode:

```
PRINT       =$FDED
BUFFER      =$300
LINEIN      =$FD6A
CTRLD       =#84
BELL        =#87
LOCO        =00
```

This mode is used to define routines or memory locations that lie outside the program or constants which are to be labeled for clarity. The immediate mode of the last two entries will be carried over during assembly. The entries are accomplished by entering the label in the Label field, then pressing the space bar. Then an equals sign (=) is entered, followed by the value. Hexadecimal entries must have at least two digits, as shown in the last example.

PART 3

Assembling

After the source file has been developed, the Edit mode may be exited using the **ESCape—ConTRoL Q** method. Then, with the source file still in memory, option (**A**) RUN ASSEMBLER may be chosen.

The Assembler program presents another menu. To assemble the source file, choose option (**A**) ASSEMBLE FILE IN MEMORY. You will be presented with the option to choose an address at which the machine language program will run.

The object file is always assembled in the hex $803 – $1600 address range. It will frequently be desirable to run the program at some other address. Therefore, the program will be assembled in the specified range of addresses, but a *destination*, or operating location, may be chosen for the program. When the program has been assembled it may be saved to disk and later relocated to this destination address for use. If no entry is made for the destination address, the origin address of $803 is assumed.

Assembly requires two passes because a line may call for a label which is forward in the program, thus still undefined on the first pass. Also, the program's actual physical length is not known until one pass is complete, so that address references will change during the first pass and all references to that address will be updated on the second pass.

ERROR DIAGNOSIS

Assembly will begin after the destination address entry has been made. If an error is detected, the assembly will pause to allow examination of the offending line and the intermediate numeric form for that line. Example:

```
BEGINNING PASS 2
..................□

ERROR:      REL  BRCH
LABEL            BNE   LOOP        ;NOT READY
$900:            BNE   $87E
(RETURN)         = MORE, (Q) = QUIT ? □
```

This particular error comes about because relative branches may not exceed +127 or −128 bytes from the origin. The other errors are:

ERROR	EXAMPLE	
BAD FORMAT	LDA (TEST,Y	[ONLY ONE PARENTHESIS]
UNRECOGNIZED	LDQ $1234	
ILLEGAL SYMBOL	STA ?0	
INCOMPLETE	JMP	[NO DESTINATION]

If a line is displayed, and the Operand field has the label on display in both the source and intermediate numeric lines, it means that the Assembler was unable to find the label and assumed that it was a number. If the second assumption turns out to be false, the error diagnostic routine will be called. Example:

```
BEGINNING PASS 2
..................□

ERROR:      REL BRCH
LABEL            BNE   FACE        ;EYEBROWS
$927:            BNE   FACE
```

Is "FACE" a label or the hexadecimal equivalent of 64,206? To avoid problems like this, *always precede numbers with the dollar sign* (**$**).

There is a particularly difficult problem to diagnose, if this is not done, in which a page zero number lacking a dollar sign is temporarily given a 2-byte number during the first pass, which creates an acceptable assembly in the absence of all label definitions. Then, in the second pass, an equivalent label is still lacking, so the numeric assumption is made. When the number is defined, the step containing it is a byte shorter than it was on the first pass, which destroys all branches around this step. This is a run-time error and assembly will complete without an error indication.

As experience is gained in the operation of the system these errors will become less frequent. The location of the offending line in the source file may be located in difficult cases using the string search feature.

After the errors have been removed and assembly is successful, the Assembler will ring a bell and print the source and destination address ranges. After this display the user may choose option (**S**) SAVE from the menu. The title under which the file is saved is the name provided by the user plus the automatically added prefix "OBJ.". This is so that a source file and an object file may be saved under the same name without confusing the Disk Operating System.

PART 4

Creating a Formatted List of the Program

Assembler option (P) PRINT FORMATTED LIST is now discussed. This program is called after successful assembly, while the source and object files are still in memory.

Upon running the Formprint program, a menu will be displayed. In the normal sequence for printing, the user chooses option (**E**) ENTER PRINT ADDRESS first, and enters the address for the printing peripheral card. Example:

> If an Apple II SERIAL interface card is in use, and it is in peripheral slot 1, enter **C100** for the address. If an Apple II PARALLEL interface card is in use in the same slot, it must first be tricked into believing that everything is all right (see user manual), then an address of **C102** is used. (The other printer interface option, (S) ENABLE INTERNAL SERIAL INTERFACE, is discussed in Appendix A to these instructions.)

Now option (**P**) ENTER HEADING AND BEGIN PRINTING is chosen. The user is prompted for a page title. This title will appear at the top of each page along with a page number:

[title] Page [page number]

After entering the title, and before pressing **RETURN**, be sure that the printer is ready. This means:

1. The top of the first page must be aligned with the printing mechanism. Allow for one carriage return as printing gets underway. This first carriage return clears residual contents from the input buffer, a requirement on some printers. Standard 66-line page intervals are used, and 57 lines are actually printed on each page, in accordance with the format of standard fanfold computer printing paper. About 65 character columns are used.
2. If the printer has automatic form feed, disable it. The Formprint program takes care of this.
3. If the printer has the option of generating a line feed on a carriage return, enable this feature.

If the object file at the left gets out of synchronization with the source file at the center (incorrect machine language equivalent instructions), it may be that one or more comments in the source file exceed one line of Apple's display. This ought to be picked up by the Assembler as an error, but in rare cases it will not. Multiple-line string entries and hex data entries, of course, are permitted and are handled by the Formprint program.

The other common problem encountered in the use of this program is that the source and object files are no longer in memory, or have been destroyed by, for example, a DOS boot after assembly. Booting DOS usually wipes out the object file but spares the source file. A successful listing will be generated only if both source and object files are in memory and intact.

The addresses for the object file will be destination addresses so that the listing will be appropriate to the machine language program in its running location. Just remember that the listing is carried out with the object file in its original location, just as assembled.

PART 5

General Characteristics

RELATIVE DENSITY OF FILES

A statistical study of programs developed using the system reveals that the average entered line is 10 characters long, thus a theoretical maximum of 2500 lines may be entered. 2112 2-byte instructions will fit into the allocated object file space, thus the balance between the source and object files will depend upon the number of comments that are entered in the source file and the specific nature of the program being written. If many 3-byte instructions are used, fewer source lines will be assembled before overflow.

Each label entry is 11 characters long, 7 for the label and 4 for the numeric equivalent, including special identifying characters. Therefore the Label file can hold 361 labels, an average of one label for 6 instructions.

SOURCE FILE PRESERVATION

A sophisticated method is used to preserve the source file while it is in memory. It should be realized that the source file is split while editing is underway, and is recombined for assembly or saving to disk. This recombination will be done at the Assembler if the operator presses the **RESET** key instead of exiting in the normal fashion. Many anomalous conditions are defended against. However, if the disk is rebooted during editing, all will be lost.

APPENDIX A

Serial Interface

NOTE: The following is a technical discussion of the built-in serial interface. It has no direct bearing on the operation of the Assembly Language Development System.

Format printer program option (S) enables a software-implemented serial interface. The connection point for this interface is the Apple II Game I/O socket. To make connection, it is necessary to acquire a 16-pin I.C. plug, such as that on the game paddles.

All standard baud rates are available. Electrical interface is normal 5-volt TTL. A standard serial signal with one sync and one stop bit is generated, and appears at Game I/O pin 15. The printer's read ready signal is connected to pin 2. Ground connections are made to pin 8.

Because these interfaces are not buffered, great care must be taken in making the connections. Don't connect or disconnect the printer while either it or the Apple II are turned on. Make sure that the printer will electrically interface with the Apple II.

If your printer is not equipped with a read ready signal, connect Game I/O pin 2 to pin 1. This causes continual output without interruption, a state that can be dealt with by only the hardiest of printers.

Apple™ Assembly Language Development System

EDITOR / ASSEMBLER / FORMATTER

At last, a software system that allows assembly language programming — write, edit and modify — in easy style on the Apple II.

Designed for the serious assembly language programmer, yet simple enough for the beginner who wants to leave BASIC behind, this package brings sophistication to 6502 programming on the Apple II. Actually a set of three programs (Editor, Assembler, and Formatter), the ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM makes writing machine language programs relatively painless. The programmer enters program information in Source Code form through the character-oriented Editor and can then assemble this code into Object Code using the Assembler. A formatted listing of the program Source Code or Object Code can then be created using the Formatter. The Editor helps spot bugs in the program and provides a means of correcting them, while the Assembler evaluates the machine language program for obvious errors before it is permanently encoded.

Features

EDITOR

- Cursor control mode for fast on-screen editing.
- Available memory counter.
- String tracer for finding and modifying information.
- Entire file or segment save features.
- Disk based macroinstructions.
- Local and global labels for independent units within a program that provide easy insertion of subroutines without making label changes.
- Special entry types, including String entry, Hexadecimal entry, ASCII mode, Alphabetic variable mode, Offset addressing, and Label definition mode.
- 7 character global and local labels permitted.
- Character, Word, and Line deletes.

ASSEMBLER

- Two pass operation.
- Supports all standard 6502 op-codes.
- Error diagnosis.

FORMATTER

- Built-in serial interface using game I/O socket.
- Page titling and numbering.
- Complete Source and Object listings.

Specifications

System requirements: The ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM requires an Apple II or Apple II Plus using DOS 3.2 (or earlier), at least 24K RAM and one or more disk drives. ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM can also interface with any standard printer.

Apple is a trademark of Apple Computer Company, Inc., and is not affiliated with Hayden Book Company, Inc.