Apple 65Ø2 Assembler/Editor

TABLE OF CONTENTS

APPENDICES 51

INDEX 65

INTRODUCTION

SYSTEM DESCRIPTION

The Apple II Assembler/Editor system is designed for editing and assembling small to very large programs in 65Ø2 assembly language. The system comprises three programs: the Editor; the Assembler; and the Command Interpreter, which calls the Editor and Assembler. The Command Interpreter is always in memory, and the Editor is in memory any time the colon prompt, : , is on the screen, which is whenever an assembly is not taking place. When the ASM command is issued, the Assembler is loaded in, replacing the Editor and erasing any edit file in memory. As soon as the assembly is complete, the Editor is loaded, replacing the Assembler.

The Editor is used to create source programs in the format required by the Assembler. It may also be used to create EXEC files and to edit or create BASIC programs. The Editor can handle both disk files, using DOS, and cassette tape files.

The Assembler is used to translate a source program into 6502 machine code. It reads source files from the disk and writes each output module directly to disk. By chaining multiple source files together, it can assemble very large programs. This Assembler only keeps the symbol table, and if needed, the relocation dictionary, in memory; thus the size of an assembly is not limited by the size of a source file.

Full use of all system features requires a 32K Apple II or Apple II Plus with at least one disk drive and controller with proms P5A and P6A. Very large programs require a 48K system and additional disk drives for the source files. The Editor will edit a source file up to 3ØK in size, or 1ØØØ to 15ØØ fully commented lines of source code.

The Assembler/Editor is designed to operate under 16-sector DOS. The system diskette contains a copy of DOS, so the system will always run in the proper environment. The system's DOS file structure is compatible with Applesoft and Integer BASIC. It is NOT compatible with the Apple Pascal system, which has its own Editor and Assembler.

BEFORE YOU START

Before trying to use the Editor or Assembler, you should make a backup copy of your Applesoft Tool Kit system diskette. To do this, first make sure your original is write-protected, then boot the DOS from your DOS System Master diskette and RUN the COPY program.

This manual assumes that you already have some experience in 65Ø2 Assembly Language programming, as well as some familiarity with the Apple Disk Operating System. If you are not an experienced disk user, you should read the DOS Manual before attempting to use this system.

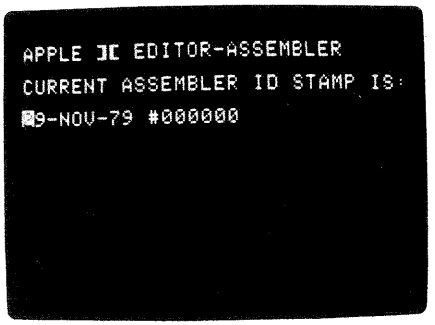
If you have a one-drive system, you can can make a backup copy using the FID (FIle Duplicator) program on the Apple Utilities Diskette.

STARTING THE SYSTEM

To run the Assembler/Editor, boot your Applesoft Tool Kit diskette, and type
RUN EDASM
If the system responds with LANGUAGE NOT AVAILABLE, type
RUN INTEDASM

Note: the Tool Kit diskette must be in drive 1 so that the Command Interpreter can load the Editor and Assembler modules properly.

The Assembler ID Stamp will now be displayed on the screen:



Update the information displayed on the screen by moving the cursor with the arrow keys -> and <-, and typing new characters over the old ones you wish to change, the same way you edit a line in a BASIC program. When you are satisfied, press RETURN. The system's name and a copyright message will be displayed on the screen, and the system prompt character, a colon, will appear at the bottom of the screen, followed by the cursor, a blinking underline.

NOTE: If you are already running and DOS is booted, you can run the Assembler/Editor without rebooting DOS. If you don't want to update the I.D. Stamp, insert the Assembler/Editor system diskette, type
BRUN EDASM.OBJ
and press RETURN.

At this point all the Editor commands may be used by simply typing the command and then pressing RETURN. Command mode is always indicated by the presence of the colon prompt at the beginnning of the line.

THE EDITOR

DESCRIPTION OF THE EDITOR

The Command Interpreter of the Assembler/Editor system indicates that it is ready to accept commands by displaying a colon , : , at the beginning of each line. Commands may be entered by simply typing the command and then pressing RETURN. The Command Interpreter ignores leading, trailing, or embedded spaces in reading commands. Parameters following a command may not begin with any of the optional characters in the command's name. Since most of the parameters that follow commands are numeric, this restriction seldom applies. Note also that one or more of the optional characters may be omitted from a command name, as long as the ones used occur in proper order. For example, the List command will be invoked by entering L, LT, LS, or LIT.

The Editor does not maintain line numbers within the edited text file, nor will it search the file for numbers that match the line numbers used in your commands. The Editor defines a "line" as all text between two RETURNs. The Editor creates the line numbers that are shown for each line by counting lines from the beginning of the file. Thus when a line or lines are Deleted, the numbers of all subsequent lines are automatically lowered. Likewise, when new lines are Inserted, the numbers of all subsequent lines are automatically raised. Thus the line numbers used by the Editor are for directing the Editor only and are not part of the file.

This relative-line-number approach has numerous implications in using the Editor, which are discussed in the relevant command descriptions. The Editor can only edit a file that will fit into the available edit buffer, which is about 29K for a 48K system and 13K for a 32K system. A typical 6502 source code line with a comment is about 40 characters, so a 48K system can edit a program of at least 700 lines and probably up to 1400 lines, since many lines are very short or don't include a comment.

The Editor provides 16 adjustable screen-position tab settings to format the text for easy reading; these are described in the section on the Tabs command. The Editor only displays the text file using the 40 column Apple video screen and expects input via the Apple keyboard. An external terminal can be used, but the output routine will wrap the lines at 40 columns. The Edit command is designed to operate only with the Apple display and keyboard: it will output the edited line to an external terminal each time a key is pressed.

This Editor is not intended to be a word-processing editor: it is intended primarily for program development. It is possible to edit Applesoft or Integer BASIC programs by converting them into text files using the procedure described in the chapter on Capturing Programs in a Text File in the DOS Manual. After this is done the text file may be edited using this Editor. This type of editing is described in more detail in the appendix Editing BASIC Programs.

ENTERING COMMANDS

There is a Help command in the Editor. To see the Help display, start the system, then type

?

and press RETURN. You will see the syntax of the various commands on the screen. The Help display fills two screens: press RETURN again to see the second screen.

When you are entering Editor commands, you may abbreviate the commands so as to use a minimum of characters to specify the command. For example, the List command can be invoked by typing L,LI,LIS or LIST. You will note in the display that the required characters of each command are shown in normal display mode and the optional characters are shown in inverse (black on white) mode.

This optional character facility only applies to alphabetic characters and the comma. The commas in the ASM command may be omitted only if all subsequent parameters are also omitted. This is discussed in more detail under the ASM and COpy commands. In the Help display the syntax of the parameter(s) after the command name only show the most common form. For many of the commands, more complex parameter lists are valid and and will be discussed later under each command.

COMMAND MODE FACILITIES

When you are in command mode (that is, when the colon prompt is showing), several facilities make the system easier to use. These are described in this section.

MULTIPLE COMMAND ENTRY

This facility lets you type more than one Editor command as a single line. The Command Interpreter will save the second and subsequent commands and, after executing each command, it will fetch the next command from the queue. Multiple commands are separated by the Command Delimiter. Until you change it, the Command Delimiter is set for the colon. Errors in any command will clear the queue of any remaining commands. Failure to find a string sought by the Find, Change, or Edit commands is not considered an error by the system.

If the ASM command is entered into the queue, it will be the last command executed. Any command following it will be flushed when the Assembler is loaded.

Some care must be used in queuing commands, as some of the line numbers used when keying in a queue of commands may no longer apply if one of the commands changes the line numbers of lines in the file. The Add, Insert, COpy, Delete, and Replace commands all could, depending on what is done with each, change the line numbers of the lines in the file.

The contents of the queue are not accessed by the Input mode or the Edit mode at any time, so it is possible to follow one of the commands that enters one of these modes with another command, such as the List command or the List Recall command (described below). This lets you do multiple Inserts or Adds interspersed with other commands. This facility becomes more powerful with practice.

COMMAND DELIMITER SET

The Command Delimiter setting capability allows you to change the command delimiter character from the default character, the colon. This is necessary because the Command Delimiter may not be used as a character in a search string for the Find, Change or Edit commands. The colon was chosen as the default since this character is not commonly used in 65Ø2 Assembly language source programs.

To change the Command Delimiter, press the ESC key, then the colon key followed by the desired delimiter character. If you press the spacebar after the colon the system will beep and expect you to type some other character. If you press RETURN, the system will just print the current Command Delimiter and return to command mode. Any other character will be accepted as the Delimiter to use for command separation. The period should not be used for a command character, as this is the Direct DOS Command escape character.

REPEAT LAST LIST COMMAND

This is a simple facility that saves the List command as it is executed and provides a one-character recall of that command with all its parameters. To Relist, type CTRL-R, then press RETURN or type the Command Delimiter and more commands. Relist may be actually be entered in the command queue like any other command. Remember that it always re-executes the last List command regardless of when that last List command was used. The CTRL-R must be the first character of the command.

DIRECT DOS COMMANDS

This facility lets you execute most DOS commands without leaving command mode. It adds the file-management capabilities of DOS to the Editor's scope, while keeping the Editor program small so that the Apple's memory can handle large files. It is designed to invoke the following commands only: RENAME, LOCK, UNLOCK, MON, and NOMON.

Direct DOS commands are invoked by simply typing a period as the first character of the command line, followed by the DOS command exactly as it would be typed when using Applesoft or Integer BASIC.

For example, the command

```
:.RENAME XX,YY
```

will RENAME a file on the most recently used disk drive.

The Assembler/Editor system is not protected from abuse by a novice user: it is designed to provide maximum power to the expert programmer. The Direct DOS Command facility provides unlimited access to the DOS commands normally available to an executing BASIC program. This facility must be used with care, as it is easy to execute a DOS command that will destroy any text in memory.

If you use any of the Access Commands, any of the Machine-Language File Commands, or any of the BASIC commands, such as LOAD, SAVE, RUN, or MAXFILES, unpredictable results will probably occur, but most likely you will clobber the Assembler/Editor if you misuse this facility. If you issue the INIT command via this facility, your INITialized diskette will lack a HELLO program, as there was none in memory for INIT to use, and it will not boot. If you INIT the system master, well...you did make a backup, didn't you?

COMMAND SYNTAX HELP

This is the ? command or the Help command. When you type

```
?
```

as the command, a special command syntax table will be displayed. Two screens of abbreviated command syntax are shown: to see the second screen, press any key. This facility provides a built-in "reference card" right at your fingertips. You may also enter a command after the ? and only that command's syntax will be displayed. If you type in an unknown command, the entire display will be provided as if no command had followed the ?.

SYNTAX OF PARAMETER LISTS

In the following discussions of the commands, the terms defined here are sometimes used to specify the syntax of a command's parameter list. Some commands have no parameters; others have required parameters, optional parameters, or both. The following table describes each type of parameter, but a command may use various combinations of them. The order shown in each command description (and in the HELP display) is the order in which the parameters must be entered. Optional ones can be omitted if not needed.

String parameters require the use of a delimiter (denoted as <Delim> in syntax diagrams), which is a character that does not occur in the string to be delimited. It cannot be a comma, hyphen, digit, blank, or RETURN. Recommended delimiters include single and double quotes, periods, question marks, exclamation points, and semicolons. If you use parentheses or brackets of any kind, make sure you use the same one for each of the delimiters in a command: two left parentheses are a valid pair of delimiters; a left parenthesis and a right parenthesis are not a valid pair.

In the following descriptions and examples, the horizontal line should be read as "or". The expression 1 | 33333 | Ø | 6 may be read as 1 or 33333 or Ø or 6.

Parameter name	Description and Examples
Rnumber	Required decimal number with a value range dependent on the command. 1 | 33333 | Ø | 6 are valid but not .3 | -4 | $CØØØ
Onumber	Optional decimal number with a value range dependent on the command. 1 | 33333 | Ø | 6 are valid but not .3 | -4 | $CØØØ
Rlinenum	Required decimal line number ranging from 1 to 65519. 1 | 33333 | 99 | 6 are valid but not .3 | Ø | -555

Olinenum

Optional decimal line number ranging from 1 to 65519.

1 | 33333 | 99 | 6 are valid

but not .3 | Ø | -555

Range

Optional line number range, consisting of one or two Olinenum's. If there are two Olinenums, they must be separated by a hyphen, - . If only a single Olinenum is present, it is the beginning and the end of the range. The second Olinenum may be omitted with the hyphen still present, implying the end of the file. If the second Olinenum is less than the first and itself less than 1ØØ, then this is taken to mean a count of the number of lines in the range, starting at and including the first line, so that 1ØØ-99 would be equivalent to 1ØØ-199.

1-2 | 1ØØØ- | 5ØØ-666 | 1ØØ-99 | 8ØØ-1 | 77-77

Rangelist

This is a set of independent Ranges, separated by commas; , . Each Range is processed independently of the others. There is no restriction on the order of the line numbers in each Range within the list.

1-2 | 1,2-1ØØ,3 | 1ØØØ-,222-33,4,55-6,77-77

Dstring

A Delimited string consists of a delimiter, zero or more alphanumeric characters, and the same delimiter.

.what i want to see. | ;; | 'a good example'

Chgstring

A Change string is similar to a Dstring. It consists of a delimiter, as above, zero or more characters, the same delimiters, zero or more characters, and the the same delimiter again. See further information under the Change command.

?oldstring?newstring? | %old%replacement%

Rfilename

A Required filename is 1 to 3Ø alphanumeric characters not including the comma, naming a DOS TEXT file.

MYFILE | MYBASICPROGRAM | MYEXEC | ETC

Ofilename

An Optional filename is Ø to 3Ø alphanumeric characters not including the comma, naming a DOS TEXT file.

MYFILE | MYBASICPROGRAM | MYEXEC | ETC

OObjfilename

An Optional Object filename is the same as an Ofilename, except that it specifices the name of a Binary or Relocatable object file generated by the assembler.

MYDRIVER.OBJ | THE LONG NAME OF MY OBJECT

DevCtlstring

A string of characters designed to initialize an APPLE peripheral card for use as the assembler's output device.

CTL-I12ØN | CTL-AF

EDITING COMMANDS

This section describes the commands that manipulate the contents of the edit file. The next section describes the commands that move the contents of the buffer to and from diskette. The third section describes the commands that control the overall operation of the Editor.

Some notes on syntax: Letters in a command name which must be typed in order to implement that command are shown here in capital letters. Parameters enclosed in square brackets are optional.

ADD

Add [Olinenum]

The Add command is normally used to add new lines to the end of the edit file. When you type A, AD, or ADD to the command prompt, the system will count the lines in the file and display the next line number to be added to the edit file. You will now have entered Input mode. You may now type text into the edit file, terminating each line by pressing RETURN. After each RETURN is pressed, the Editor

will prompt with the line number of the next line to be added.

When you have completed your entry of text lines, which might be an assembler source program, type CTRL-D or CTRL-Q immediately after the line number displayed on the screen, then press RETURN. This will terminate Input mode and return you to the Editor Command mode by displaying the colon on a new line.

You may also use the Add command with a line number. When you do this, you will enter Input mode as described above, but your new lines will be added after the line with the given number, rather than the last line in the file. The Insert command works similarly, but inserts the new lines before the line with the given number.

When you are in Input mode, the normal Apple II input editing functions are all available. If your Apple has the Autostart ROM, the additional ESC features for cursor movement are also available. The Editor's Input mode extends the normal Apple II input routines. When you enter control characters into the input line and then backspace over them with the left arrow key, the cursor correctly avoids backing up on the screen.

Although this mode allows you to put control characters into your file, you should be careful to keep them out of assembly-language programs, as they are not valid in assembly language.

COPY

COpy Rlinenuml [-Olinenum2] TO Rlinenum3

This command will copy one or more lines TO just before linenum3 in the file. When the optional linenum2 is used the range of lines from linenuml to linenum2 is copied. If linenum2 is omitted then only linenuml is copied. The COpy command is the only command that requires an embedded keyword, TO, inside the parameter list. As is indicated by the CAPITAL letters in the command syntax above, the two letters CO are required and the letters py are optional.

Note that the Editor has no 'Move' command. Lines may be moved by COpying to the new location, then Deleting them from the old location. Be aware that if you do a COpy with linenum3 less than linenuml that the line numbers of the original lines will be changed when the Copied lines are inserted before them. So if you then want to Delete the original lines, you must determine their new numbers before you can Delete them! You will not have this problem if you are copying forward in the file to a linenum3 higher than linenuml, as the original lines will not move.

When linenum2 is used it must always be greater than or equal to linenuml.

CHANGE

Change [Rangelist] Chgstring

The Change command lets you substitute a new string for some or all occurrences of an old string in the entire edit file or within the lines specified by the optional Rangelist. If you omit the Rangelist, the Editor will search the entire file for the old string and replace all occurrences of it with the new string.

The syntax for the Chgstring is

<delim> [string] <delim> [string] [<delim>]

where the first string is the old string, and the second string is the new string. The syntax diagram shows that at least two delimiters (<delim>) are required. The old string cannot be null, and the new string may be null. The trailing delimiter may be omitted, as the RETURN can serve in its place. <delim> may not be the command character.

The Change command will always prompt with the following line:

ALL OR SOME? (A/S)

You may then choose whether to let the Editor change all occurrences in the range of lines given, or prompt you to verify or reject each possible change. If you respond with the letter S or any key with a higher ASCII code, the Editor will stop after showing each Change on the screen, then flash the cursor, asking for verification of that specific change.

If you press the ESC key or any other control key except CTRL-C, the Editor will not make the change, and will search for the next occurrence of the old string. CTRL-C cancels all changes and returns you to the command mode. If you press the space bar or any non-control key (except RESET) the Editor WILL replace the old string with the new string in the file and go on searching. When you reject a specific case, the Editor just keeps searching: it does not change the line on the screen back to its original form, even though the line in the file was left unchanged.

The Change command does not rescan the current line from the beginning after each change is made to the line, nor will the new characters just added by the change command be scanned. This means that if you have, for example, 1Ø '*'s in a line, and you do a change of /**/*/ the line will have 5 '*'s remaining after changing ALL occurrences of /**/ to /*/.

The old-string part of the Chgstring may be used in a special way by inclusion of the wild-card character, CTRL-A. It is used to indicate that any character, except a carriage return, will be matched for each CTRL-A found in the old-string portion of the Chgstring. This provides a way of changing a set or class of strings in the file all

to the same new string. You may NOT use CTRL-A as the wild-card character in the new-string portion of the Chgstring; it can only be used to insert a CTRL-A in that position.

DELETE

Delete [Rangelist]

The Delete command deletes the lines specified by the Rangelist. If multiple ranges are to be included in the range list they must be in reverse order (i.e., highest range first, descending to lowest range last. If this is not done when using multiple ranges with Delete, the lower range will change all the line numbers of the later lines and the wrong lines will then be deleted by the higher linenumbered range! Remember that each Delete will reduce the line numbers associated with all following lines in the file.

The Delete command may NOT be used without a Rangelist: attempting to do so results in an ERR: SYNTAX message. Rangelist may not, however, be linenum-; that is, you must specify both tha beginning and the end of the Rangelist.

EDIT

Edit [Rangelist] [Dstring]

The Edit command is the most versatile text-manipulation command in the editor. It puts you into Edit mode, which gives you complete character-at-a-time editing facilities. Edit mode, like Input mode, is separate from Command mode. Edit mode applies to one line at a time and is one-dimensional: that is, you may only move forward using the right-arrow key , or backward using the left-arrow key, within the line being edited.

The Edit command puts the system into Edit mode and specifies the lines to be edited. If both the optional Rangelist and the optional delimited edit-string are omitted, then ALL the lines in the file are presented, one after the other, within Edit mode. If both the Rangelist and the edit string are used, then only those lines listed in the Rangelist and containing the edit string will be selected and presented for Editing. If the edit string is omitted, all lines in the Rangelist will be selected.

Thus you can see that if only the edit string is used then all lines in the file containing the edit string will be selected for editing. The edit string may contain the wild-card character to allow matching on any character (see further under the Change command).

When you are in Edit mode, you can move the cursor forward and backward, and replace, insert, and delete characters. The line on the

screen will show all your changes immediately: you will never be kept in suspense.

All of the Edit mode facilities are accomplished by using control characters to instruct Edit mode what to do. Typing an Edit mode control character causes its respective command to be executed; any other control character will be rejected with a beep. Typed characters other than control characters are taken as one-for-one replacements of the characters in the line. These may be the same characters that are already there, since sometimes it is faster to retype a few characters than to use the cursor-motion facilities. The length of the edited line is not limited by the length of the original line and may end up longer or shorter than the original. Here's how to use the Edit mode features:

First, get into Edit mode, using the Edit command to specify the range of lines to be edited. The first line in the range will be displayed, and the cursor will move to the first character position of that line.

Now move the cursor to the first character or space you wish to change, using the arrow keys. Don't worry if the characters after the cursor jump back and forth: this happens because the space is used as the tab character (unless you changed the tab character). Don't use the space bar to move the cursor: this will replace the characters passed over with blanks.

To replace the blinking character, simply type the new character. It will replace the blinking character, without affecting any other characters, and the cursor will move to the next character. As you keep typing, more characters will be replaced.

To insert a character before the blinking character, type CTRL-I, then the character to be inserted. It will appear before the blinking character, which will move right one space, taking the rest of the line with it. As you continue to type, more characters will be inserted. To finish the insertion, press either arrow key, or type any control character except CTRL-V.

To delete the blinking character, type CTRL-D. The blinking character will disappear, and the one to its right will move under the cursor, taking the rest of the line with it. To erase more characters, type CTRL-D again, or use the REPT key with CTRL D.

To restore a line on the screen to its original form, type CTRL-R. The line on the screen will be replaced by a copy of the original, unchanged, line in the file. This feature is useful if you have made hash of a line and want to start over. It will only work if you have not pressed RETURN, which puts the new line into the file.

To find the next occurrence of a character after the cursor, type CTRL-F, then the character to be found. The cursor will jump to that character if it can be found in the current line. Otherwise, the cursor stays in its current position.

Control characters can be troublesome: they are always popping up when you least expect them, and disappearing when you need them most. The Editor has two facilities that help you control control characters: Popout and Verbatim.

Popout displays the control characters in a line in blinking, 'popout', mode. Normally, control characters are invisible on the screen, but this facility lets you examine them, one at a time. As you move the cursor back and forth on a line, in Edit mode, any control characters will pop out at you, pushing the rest of the line aside. If you are not sure whether a character is a control character (because the cursor blinks, too), simply move the cursor off it. If the character goes away, it is a control character.

Verbatim, or CTRL-V, lets you put control characters in your file, either by replacement or by insertion. (Normally, Edit mode rejects control characters with a beep.) To replace a character with a control character, move the cursor to that character. (If you are already there, and just inserted a character before it, move the cursor back and forth with the arrow keys, to disable insertion.) Now type CTRL-V and the character you wish to enter. If you wish to replace more than one character with control characters, type CTRL-V before each.

To insert a control character before the cursor, type CTRL-I, then CTRL-V, then the control character. If you wish to insert several control characters, type CTRL-V before each. You can stay in Insert mode, or return to Replace mode, as described above.

If you have made all the changes you want, or if you have decided not to make any, you can leave Edit mode in any of three ways. The first is to type a CTRL-X anywhere in the line. This completely cancels Edit mode, even if all lines in the Rangelist have not come up, and returns to you to Command mode, leaving the current line in its original form. Even if some changes have been made on the screen, the line in the file is not changed unless one of the other two exit modes is used.

To put the changed line into your file and leave Edit mode, press RETURN. This puts the line as it appears on the screen into the file in place of the original line. The line in the file in not changed until this is done. When RETURN is used, the entire line is put in the file regardless of where the cursor is currently positioned in the line.

To put the first part of your line into the file, position the cursor on the last character you want to keep, then type CTRL-T. This erases all characters to the right of the cursor and puts the truncated line into the file.

Once Edit mode has been terminated by a RETURN or CTRL-T, then if another line can be selected for editing, that line will be presented, with the cursor at the first character. If another line is not selected, Edit mode is automatically terminated and you will be returned to Command mode, having satisfied the conditions of the Edit command.

Edit Mode Control Character	Function Description
RETURN	Accepts the line as it appears on the screen and replaces the old line in the file.
CTRL-X	Cancels Edit mode entirely WITHOUT replacing the current line in the file.
CTRL-T	Truncates the line at the current cursor position and replaces the old line in the file.
CTRL-R	Replaces the not-yet-accepted edit line with the original line still in the file.
CTRL-D	Deletes the character at the current cursor position, shortening the line by 1 character.
CTRL-I	Enters character Insert mode. Characters are Inserted before current cursor position.
right-arrow key	Moves the cursor forward one position in the line.
left-arrow key	Moves the cursor backward one position in the line.
CTRL-V character	Puts the character verbatim into the line, either inserting or replacing depending on the current character mode. Do not type a space before the character to be inserted.
CTRL-F character	Finds the next occurrence of the character to the right of the current cursor position. Do not type a space before the character to be found.

FIND

Find [Rangelist] [Dstring]

The Find command locates and lists all lines containing an occurrence of the Dstring within the lines of the Rangelist. If no Rangelist is specified then the entire edit file is searched, if no Dstring is given then all lines in the Rangelist are listed and if neither are specified then all lines in the file are listed. The wild-card character (CTRL-A) may be used in the Dstring of the Find command: refer to the description of the Change command for details.

The Find command will only list a line once regardless of how many occurrences of Dstring occur with that line. Find simply returns to Command mode if no occurrences of Dstring can be found or no lines occur within the Rangelist.

INSERT

Insert Rlinenum

The Insert Command enters Input mode, described earlier under the Add command, and inserts all the new lines before the given line number. Insert 1 will insert lines before the first line in the file. Input mode is terminated by typing either CTRL-D or CTRL-Q. Inserting a range of lines raises the numbers of all lines after them, so check the new numbers before doing anything to these lines.

LIST

List [Rangelist]

The List command lists the lines of the Rangelist on the screen, showing before each line the current 'editor' line number of that line. If no Rangelist is given, the entire file is listed. You may halt the listing at any point by typing CTRL-C. This will return you to Command mode. You may also suspend the listing by pressing the SPACE BAR and restart it again by pressing any other key, except CTRL-C. If you press the SPACE BAR again, the List command will display one more line and stop again, thus providing a single-step mode for examining the lines one at a time.

The List command will just return to Command mode if no lines fall within the Rangelist. When multiple ranges are used, the List command puts one extra blank line between the lines of each range.

PRINT

Print [Rangelist]

The Print command is just like the List command in all respects except that the line numbers are not printed and the first character of the line is in column 1.

REPLACE

Replace Range

The Replace command is just two other commands linked together for convenience of use. You may enter a Rangelist, but only the first range of the list will be used. Range may not be linenum in this case. Replace Deletes the first range and then puts you into Input mode, Inserting at the first line number of the range. Refer to the description of Input mode in the Add command for more details. Range may not be linenum-; that is, you must specify both the start and the end of the line.

DISK AND TAPE COMMANDS

The Assembler/Editor has three commands that allow you to transfer text files to and from diskettes. The Editor does this disk input and output in the same way that a BASIC program would as described in the DOS Manual. The text files on the diskette are read and written sequentially. Two additional commands allow you to transfer files to and from tape cassettes. Note that the line numbers used by the editor are not part of the file and are never written to the diskette or cassette.

The Editor's disk commands work with any DOS sequential text file, but not random-access text files. Thus the Editor can be used to create and modify EXEC files, and examine and modify sequential data files written by BASIC programs, if those files will fit in the available size of the edit file.

All of the disk commands described in this section use two parameters: current disk slot and current disk drive. These two parameters are not supplied in the disk commands themselves, but are initialized to the boot slot and to drive 1, respectively, when the Assembler/Editor system is invoked. These two parameters are changed by the SLot and DRive commands described under Operating Commands, in the next section.

When any disk command is used, a number of the usual DOS errors can occur, such as DISK FULL ERROR, or I/O ERROR, or NO BUFFERS AVAILABLE, or SYNTAX ERROR. This manual assumes you are familiar with DOS and understand these messages. The appendix DOS Errors with the Editor discusses the most common errors that occur on this system, their probable causes, and and their remedies. The other DOS error messages are explained in the DOS Manual.

LOAD

LOAD Rfilename

The LOAD command will read a file from the disk drive specified by the current SLot and DRive settings. The LOAD command assumes that the filename you have used is the name of a file already on the diskette that will be accessed. If this is true, the current contents of the edit buffer will be completely replaced by the contents of that file. The LOAD command will beep and type END OF DATA when it reaches the end of the file, and then return to Command mode. You may now use any of the Editing Commands to manipulate the text.

If the filename is NOT the name of an existing file before the LOAD command is used, it will be one afterwards, if the diskette is not write protected. If you thus create a null file on the diskette, the edit file will be cleared and nothing will be LOADed. This is not usually what you want, so you must issue a .DELETE filename command to DOS so the null file will be removed from the diskette's catalog. This characteristic of the LOAD command lets you set the current filename for a new file so that you can use the SAVE command without specifying a name. It also creates a catalog entry for that name.

WARNING: Do not issue a DOS .LOAD command, as this means to DOS "Load the named BASIC program from the diskette": When DOS tries to load your file as a BASIC program, it will write over the Editor.

SAVE

SAVE [Rangelist] [Ofilename]

The SAVE command will save all or part of the edit file to the diskette in the current SLot and DRive. If that diskette is write protected, the SAVE command will fail with the usual DOS error message, and then return to Command mode. The SAVE command first tries to do a DOS DELETE command on the file before actually doing the save, so as to free any previously used space in that file. This is necessary if the old file was longer than the new one.

It is recommended that the DOS LOCK and UNLOCK commands be used to protect all the other text files on a given diskette when using the Assembler/Editor. If you do this and only UNLOCK the file you are editing, the SAVE command will not be able to accidently wipe out a file with a name similar to the one you meant to type! This happens particularly often when you are working with a large assembly with multiple source files, whose names may differ only in the last few characters.

WARNING: Do not issue a DOS .SAVE command, as this means to DOS "Save the contents of the current BASIC program": since there is no such program, the pointers used by DOS to determine the program's size contain garbage. This command will therefore have unexpected results.

If you include a Rangelist in the SAVE command, the Editor will save only the first range to the disk file, and will ignore any other ranges. This provides an easy way to create specific subsets of a file as other files.

The Optional filename may be omitted if you have previously done a LOAD or a SAVE with a filename. When a LOAD or a SAVE with a name is done, the filename is saved and this is used as the default filename when no filename is given. You can find out the default filename at any time by using the FILE command. If you have not previously done a LOAD or a SAVE with a filename, then doing a SAVE without a filename will get a DOS SYNTAX ERROR and return you to Command mode.

Whenever a DOS error message occurs and SAVE aborts, there may be a partial file on the diskette that you may want to .DELETE to remove it. The current edit file is always still intact in memory after a SAVE, even when DOS gets an error of some kind. When this happens it is usually best to obtain another initialized diskette, and insert it into your disk drive, in place of the current one, and try the SAVE command again.

Back up your diskettes. You should not type more than about twenty-minutes-worth of input or editing before you stop and do a SAVE command to get the edit file on the diskette. A one-second loss of power to your computer can wipe out many hours of editing, so don't get so busy editing that you forget to back up your file.

APPEND

APPEND [Olinenum] Rfilename

The APPEND command is normally used to load a specified file at the end of the current edit file, thus appending the new file onto the previously LOADed one. When this is done the current SAVE filename is not changed to the APPENDed filename. The APPEND command will work even if the current edit file is empty. In this case APPEND works like LOAD except that the SAVE filename is not set from the APPEND command filename.

When the Optional linenumber is given with the APPEND command, it indicates that the APPEND file is to be written over the edit file after the given line. This is effectively a truncate-and-load facility that is useful for moving around pieces of files.

TLOAD

TLOAD [Olinenum]

The TLOAD (Tape LOAD) command loads an Assembler/Editor tape file, previously written by an Assembler/Editor TSAVE command. All the usual problems with reading tape will be encountered when using this command, so it is assumed that you know how to adjust your tape recorder to get it to LOAD. You should refer to the Apple II BASIC Programming Manual or one of the other Apple manuals providing instructions on loading tapes if you need any help. This command is included to provide a cheaper medium for exchanging files than the diskette, and also to provide a way of archiving files on a less expensive medium. The format of the Assembler/Editor tape files is described in the appendix Editor Tape Formats.

When the Optional linenumber is specified, TLOAD will insert the tape text file before the given line. This is the only way to directly insert text into the middle of an existing edit file, without typing it in. The APPEND command, followed by a COpy command and a Delete command, can provide the insert function for Disk files.

TSAVE

TSAVE [Rangelist]

The TSAVE command writes an Assembler/Editor format tape file containing the entire edit file if no Rangelist is used. Only those lines in the Rangelist are put on the tape if this option is used. If no text is present in the edit file the TSAVE command will still write its header on the tape before returning to Command mode.

Note that files written by the Assembler/Editor system may not be LOADed on tape in the same way as BASIC programs, since they are not in the same format as BASIC programs; tapes are written by the TSAVE command in the following format:

(1Ø-second leader)
(16-bit file length)
(1Ø-second header)
(file data)

This format is written using the monitor WRITE routine in normal APPLE II cassette recording format.

Because of the difficulty many users have with cassette tape input/output, this is not always a practical means of storing programs, but it does provide a cheaper means of storing old programs, or text files. The assembler can not be used without a disk system.

OPERATING COMMANDS

These commands control various aspects of the system: some let you adjust parameters to other commands; others give you access to useful information.

SLOT

SLot Rnumber

The SLot command sets the SLot of the disk controller card on which all disk operations will occur. This is initialized to the 'boot' SLot at system startup. This command seldom needs to be used, but is provided for those who use different machine configurations or have more than 1 disk drive controller in their Apple. The Rnumber must be a valid SLot number for a Disk drive controller and should contain a disk card. Valid SLot numbers are 1 thru 7. The system does not check to verify that the SLot does indeed have a controller, but the next time you do a disk command you will get a DOS 'I/O ERROR' if it does not.

DRIVE

DRive 1

or

DRive 2

The DRive command sets the current DRive on which all Assembler/Editor system disk operations will occur. The only valid values are 1 and 2, any other value will give you an error message. The DRive command returns you to Command mode after setting the DRive parameter in memory. This parameter is not saved on the disk at any time and always is initialized to DRive 1 when the system is started. To find out the current value of the DRive parameter, use the CAT command.

CATALOG

CATalog

This is just an easier way to perform a DOS CATALOG command, which can be done by using the direct DOS command

.CATALOG,Dd,Ss

The main differences between these two are (a) that you can abbreviate to the shorter 'CAT' and (b) that you can't specify a SLot or DRive, since these are added by the system from the current SLot and DRive

settings. This is very useful when using a two-drive environment, with the Assembler/Editor system diskette in drive 1 and your text file diskette in drive 2. You must issue a DRive 2 command when operating this way, but after that, the CATalog command will always refer to that drive, even if it was not the last drive accessed.

FILE

FILE

The FILE command displays the current SAVE filename along with the information provided by the LENgth command and returns to Command mode. If no SAVE filename has been 'set' by a LOAD or a SAVE, nothing is shown above the LENgth display.

HIMEM=

HImem= Rnumber

This command sets the highest available address (HIMEM) for the edit file in memory. It is initialized to the largest possible address at system startup time, and it may be reduced if you want to limit the size of your edit files to less than 29K bytes of text. This command may be used when a file is in memory, but this is risky, since if you reduce HIMEM too far you will lose the tail end of your edit file.

The Rnumber represents a decimal address to set HIMEM to and should be a valid address for this purpose. The HImem= command does not check what value you give it, and bad values will destory the system's operation very quickly. There is generally no important reason to use HImem=, unless you want to recieve the usual 'OUT OF MEMORY' error before it would normally occur.

HIMEM will be set to 384ØØ for a 48K system, 26112 for a 36K system and to 22Ø16 for a 32K system. To determine what address to use with the HImem= command, subtract from these values the amount you want to to reduce the edit file by and enter that address in decimal. HIMEM can be calculated, if LOMEM has not been changed, by adding the items shown by the LENgth command to 8192, the default LOMEM value.

LOMEM=

LOmem= Rnumber

This sets the lowest address (LOMEM) of the edit file. The editor uses the byte below this address as well. LOMEM is normally set to 8192 at system startup and may not safely be set to less than this value. If you do set it lower, you will destroy the Assembler/Editor system, since the Editor will very soon write on top of itself.

LENGTH

LENgth

The LENgth command displays the amount of memory, in bytes or characters, that is currently used by the edit file and how much memory space is remaining unused. The sum of the two is the total space available to the Editor.

MON

MON

This command is intended to provide easy access to the monitor and easy return access to the editor. Normally this command is not used, except when you want to to some unusual things. This exit sets up the monitor CTRL-Y command so that when you type CTRL-Y, you will return to Command mode. What you do from the Monitor is up to you, but the memory areas used by the Assembler/Editor system must not be disturbed. These areas are described in the appendix Editor Memory Usage. This command is a tool for the expert Apple user and programmer and would be better left unused by the beginner. Note that entering either kind of Basic from the Monitor will probably destroy the memory areas used by the Assembler/Editor system. You may also return to Command mode via a monitor '3DØG' command.

NEW

NEW

This command causes the Editor to clear its current edit file. This does not actually do anything to the contents of the buffer: it simply sets the end-of-text pointer equal to the beginning-of-text pointer, otherwise known as LOMEM. It is possible, via the Monitor commands, to restore the end-of-text pointer to recover an 'NEWed' edit file. This requires direct examination of the edit file and the pointers to determine the prior end-of-text address and put that into the end-of-text pointer.

PR#

PR#p[, DevCtlstrg]

The PR# command tells the Editor that assembly listings from the current session will be sent to some device other than the Apple video screen. This is normally a printer connected to a printer interface card, but it could be an 8Ø column terminal or some other such output device. The number p, an integer from Ø to 7, is the slot number of the interface card for the output device, unless p is Ø, which represents the screen. The Assembler will output to this device or the special 8Ø column video output routine, but not both.

It is possible that the selected output device may echo what it is receiving back to the normal Apple 4Ø column video output routine, but this is up to you and how you use your device. The Apple printer interface cards can not simultaneously echo to the Apple screen and print a listing that is wider that 4Ø columns.

The 'DevCtlstrg' (Device Control string) is any string of characters that should be sent to the printer device to initialize it for listing the Assembler's output. It may not exceed 32 characters. The first two parts of the Device Control string are optional: the Logical and Physical page lengths. The Logical page length is the number of lines to be printed on a page: it is specified by typing the letter L followed by a two-digit number. The Physical page length is the number of lines from one top-of-form to the next: it is specified by typing the letter P followed by a two-digit number.

The default logical page length is 6Ø. If no physical page length is specified, a form feed is given after each page, if a SBTL is used.

The last part of the Device Control string is required: for Apple printer interface cards it will normally be the typical CTRL-I8ØN sequence, or something similar, that turns off the Apple screen and sets the printer width for the paper in use. (NOTE: This sequence is typed by pressing the I key while the CTRL key is held down, then pressing the 8, Ø, and N keys, without spaces anywhere. The CTRL-I will not show on the screen when it is typed.)

Here is an example of this command, as it is typed at the keyboard

PR#1,L54P66CTRL-I8ØN

and as it appears on the screen

PR#1,L54P668ØN

This command will send this session's assemblies to the printer in slot 1, which will be set to print 54 lines on a page 66 lines long, each line having up to 8Ø characters, as specified by the string CTRL- I8ØN.

This command stores its information in RAM, but not on disk, so it must be used again if you start a new session.

TRUNCATE

TRuncON

or

TRuncOFf

These two commands provide a means of TRuncating the displayed lines of the edit file. TRuncation is set to OFf at system startup and can be turned on via the TRuncON command. The purpose of this mode of display is to shorten the text lines so they fit within the Apple's 4Ø column display. This command in no way changes the contents of the edit file itself: it simply changes what comes out via the List and Print commands.

Lines are TRuncated at the first occurence of a semicolon preceded by a space. This is how comments in 65Ø2 source programs are begun, when they are included at the end of a statement. When the comments are TRuncated, it is much easier to read the source programs because, without TRuncation, the comments usually wrap to a second line. This mode of display is most useful when writing assembly-language programs, especially if no printer is available and all output is being sent to the screen.

TRuncation mode is automatically suspended when using Edit mode, so you needn't fear losing your comments because you forgot to set TruncOFF.

TABS

Tabs [Tablist] [Dstring]

The Tabs command lets you use the space bar (or some other key) like the tab key of a typewriter to format assembly-language statements as you type them in. When you start up the system, the tabs are set to the standard column positions for 65Ø2 assembly language.

Tabs only operate within the first 4Ø columns of the Editor output line. Any Tabs set beyond column 39 will be ignored by the Editor, but not by the Assembler. Tabbing will move a portion of an assembler operand over into what is supposed to be the comment field if it contains a tab character: this usually happens when using an operand with embedded blanks.

Up to 16 Tabs may be set via the optional Tablist, which is just a list of absolute display line positions separated by commas. The Tabs

must be in increasing order if they are to function properly. If no parameters are supplied, the Tab table is cleared to all zeros, effectively turning Tabbing off. The Dstring sets the Tab character which can be any character except RETURN. The space is the default Tab character, because this is the proper one for the format of assembler source programs. The Editor will use any tab character you like, but the Assembler will only accept the space.

Tabbing occurs when the Tab character is encountered while Listing a line. This implies that the content of the edit file controls execution of a tab during line display. The Tabs are output line positions and they cause different results for List, which prints line numbers, than for Print, which does not print them. Using the same Tab settings for Print as for List will, under normal conditions, cause 6 extra spaces to appear between the first two fields on the print output line.

WHERE

Where Rlinenum

This command is a tool for manipulating the edit file from the Apple Monitor. It provides the HEX address of the beginning of any line in the file. It also provides a means of finding out the current HEX value of the LOMEM pointer. To find LOMEM, type

Where 1

and press RETURN. This will normally display '=$2ØØØ', which is the same as 8192 decimal.

END

END

This command is used to exit from the Assembler/Editor system and return to the BASIC language. This is a complete exit from the Assembler/Editor and is provided for ease of use. If you use this command by accident, you can usually recover from Integer Basic by immediately typing

CALL 3Ø75

and pressing RETURN. If you have done this, examine your edit file completely to be sure it is intact before SAVEing it. By doing this, you can be sure you won't replace the intact old version of your file with a scrambled new version.

THE ASSEMBLER

DESCRIPTION OF THE ASSEMBLER

The Assembler is designed to facilitate assembly-language programming on the Apple II. The Assembler requires standard 65Ø2 address-mode syntax and has an extensive set of assembler directives. The source programs must first be created using the editor, following the format described below. All source programs must reside on diskette, as the Assembler is not designed for co-resident assembly.

The Assembler is invoked by the ASM command, as described below. The Assembler translates the source statements from the source file into 65Ø2 machine instructions, and writes them into the object, or output, file. This is a two-pass process. During pass one, the Assembler reads the source files and generates a symbol table, assigning the values to all the symbolic labels defined by the user.

In addition, the length of all instructions is determined during this pass, and any forward references to labels not yet defined are forced to be absolute, rather than zero-page, labels. This requires that all zero-page, or one-byte, labels be defined before they are referenced in your source programs. The second pass then rereads the source files and generates the actual machine instructions, using the symbol table to fill in the address portion of those instructions. As pass two proceeds, the Assembler writes the object files one sector (256 bytes) at a time into the Binary or Relocatable (see below) output file.

The Assembler will generate error messages during both passes, and it will output the assembly listing during pass two. The Assembler will send the listing to either the Apple video screen or another user specificed output device. When using the Apple video screen, a special 8Ø column display routine is used. The first 4Ø columns of the listing will be shown on the screen and the second 4Ø, or rightmost 4Ø, will be available for viewing via keyboard commands. This has been done to put the comment field off the screen, to make using the Assembler without a printer as practical as possible.

The Assembler may be directed to generate a Relocatable object file. This type of DOS file provides the additional information necessary to relocate the machine instructions so they may be executed at an address other than the one for which the object file was assembled. This is done by a Relocating Loader, without repeating the assembly. The additional information forms a Relocation Dictionary, or RLD for short, which is added to the end of the machine instructions. The format of the RLD is described in a later section.

The Assembler prints, during pass one and sometimes during pass two, the names of any chained source files as it begins that file. The Assembler also prints a dot on the screen during pass one, and sometimes during pass two, every 1ØØ lines of assembled source code. The Assembler only prints the dots and the file names during pass two if you have turned the listing output off.

BEFORE YOU START

The Assembler is called by using the ASM command, described in the next section. The Assembler is not in memory at the same time as the Editor, so it must be BLOADed from disk before starting the assembly process. This is done automatically by the command interpreter after setting up the parameter list you enter with the ASM command. If any errors are made in entering the parameter list, the system returns an error message instead of BLOADing the Assembler. Not all kinds of errors are found by the system before it BLOADs and calls the Assembler.

For example, if you use a source filename for which there is no file, this error is not discovered until the Assembler gets a DOS error when trying to open that file. The Assembler does not print nice error messages: it just says

OOPS DOS ERROR CODE = ??

and returns to the command interpreter. The Assembler only does this for FATAL errors related to disk I/O or due to insufficient memory for the Assembly process. These error codes are explained in Appendix D.

The normal Assembler error messages about your source program are shown on the output device, which may be either the screen or a printer device, but not both.

The Assembler has a special video output routine that produces 8Ø-column video output by using the second video page. The output, seemingly truncated at 4Ø columns, can actually be examined in full, as revealed in the section Commands Available During Assembly.

The Assembler is not designed to do assemblies with the source text or the object output in memory: it only reads from and writes to the disk. The Assembler also keeps the disk drive spinning during the entire assembly, even when it is not actually reading, so that it does not have to wait for the drive motor to start and stop.

NOTE: You must SAVE your source program on the current diskette before you use the ASM command (see below). If you fail to SAVE your file, and do the ASM command with the edit file still in memory, the command interpreter will BLOAD the Assembler on top of your edit file, destroying it, and will then try to read your old source file from the disk.

THE ASM COMMAND

ASM Rfilename [,OObjfilename [,[S]s [,[D]d]]]

The ASM command is used to call the Assembler. This section will explain the parameter list for the ASM command and tell how to use it. Two parameters not in the above list are also passed to the Assembler: the Editor SLot and DRive parameters. They are associated with the name of the beginning source file, shown above as Rfilename. This file must be online and should be a 65Ø2 source program in the format described in the Assembler portion of this manual. This will normally be the file you have just finished editing and SAVEd on diskette.

The Optional Object filename is the name you want the Assembler to use for the output object (65Ø2 machine code) file. If you don't provide a name, a default name is constructed by adding the characters '.OBJØ' to the source file name. The default name is discussed in more detail in the Assembler section of this manual. If you want to use the default name and no other parameters, the source filename is all that is required.

The Ss and Dd parameters are only used if you want to direct the object file to a disk drive other than the one that contains the source file. They are the slot and drive numbers defined in the DOS Manual: s is an integer from 1 to 7; d is either 1 or 2.

Note that the optional parameters are positional and must occur in the order shown. If you want to use the default values of some parameters but specify one of the later ones, just put in a comma in place of each defaulted parameter. For example, if you want to specify only the drive of the object file and use the default object name and slot, then you should enter the ASM command as shown below.

```
:ASM PROGRAM,,,D2
```

The two defaulted parameters are indicated by the fact that nothing is between their respective delimiting commas. The labels 'S' and 'D' are optional--the numbers themselves will suffice--but they make it easier to remember the purpose of each numeric parameter.

A few more typical ASM commands and what they mean:

```
:ASM MYFILE
```

This command assembles MYFILE from the current Editor SLot and DRive, putting the output object file, MYFILE.OBJØ, on that same diskette and displays the output listing on the Apple screen, using the '8Ø column' display mode.

```
:ASM MYFILE,MYPROGRAM
```

This ASM command has the same meaning as above, except that the output filename is to be MYPROGRAM instead of the default. It will not be

named MYPROGRAM.OBJØ unless you explicitly call it that.

:ASM BIGFILE,BIG.OBJ,,D2

This command will assemble BIGFILE into the object file named BIG.OBJ on (presumably) the other disk drive, Drive 2, of the current Editor disk controller SLot.

Before starting to Assemble, a message PRESS ANY KEY TO CONTINUE will appear, giving you the chance to change disks. You can press any key except reset, and the Assembly will commence. When the Assembly is finished, a similar prompt will appear, to let you change disks again before the editor is reloaded.

ASSEMBLY MODE COMMANDS

Several mode commands have been incorporated into the Assembler to let you control the assembly process as it takes place. These commands are described below:

ABORT ASSEMBLY

CTRL-C

The Apple looks at the keyboard during both passes of the assembly process, but only the Abort command affects pass one. The assembly may be Aborted at any time during either pass by typing a CTRL-C on the keyboard. Doing this will cause the Assembler to 'clean up' before returning to the Command Interpreter. This clean-up consists of CLOSING all the open files and freeing up the DOS buffers that are in use. It does not include removal of the output files that may have been generated on the output diskette. This may be done by issuing a DOS '.DELETE' command from command mode.

SUSPEND OR SINGLE-STEP LISTING

SPACE BAR

During pass two the listing may be stopped by pressing the SPACE BAR on the Apple keyboard. The listing may be restarted by typing any other character that is not a mode character to the Assembler. In additon, you may 'single step' the listing a line at a time by pressing the SPACE BAR once for each line. Note that this may leave the source disk drive motor running and the assembly process is suspended until the AC power goes away or you resume it as described above.

LIST PART OF PROGRAM

LST ON

or

LST OFF

The Assembler has a directive, LST ON/OFF, that may be put into your source programs to control what parts of the listing are output. This same facility may be controlled from the keyboard during pass two. Doing this from the keyboard overrides the current state of the LST option until the next LST directive in your source program or your next command from the keyboard, whichever occurs first. Thus you may examine sections of the listing you turned off from within the source program, or you may turn off sections you have turned on. This is usually only used when working on large programs, where a small area of the program has been changed and you don't want to see all the rest.

The Listing NO command is CTRL-N; the Listing ON command is CTRL-O. Remember that the Assembler could encounter a LST directive that counteracts your most recent CTRL-N or CTRL-O from within the source program. To correct this, just reissue your command from the keyboard.

SOURCE PROGRAM FORMAT

The Assembler's input source files are normal DOS text files made up of logical records. A logical record is a string of ASCII characters terminated by an ASCII carriage return ($8D). All ASCII characters must have their most significant bit (MSB) set to a one. Each of these source records is divided into four fields: the label, the operation code or opcode, the operand and the comment. These fields are normally separated by one space, since the editor tab function will format the display, during editing, using the space for its tab character. Make sure you use the space as the tab character in all assembly-language source files: although the Editor allows you to use other characters for this purpose, the Assembler does not.

Every source record or statement must contain an opcode, unless it is purely a comment. Blank or null source statements are invalid. The Assembler uses spaces as tab characters, so you should put spaces between fields and not within a field. To skip over a field, use two spaces. The Assembler does not allow multiple statements in a record, or allow line numbers in source statements (except as comments).

THE LABEL

label

or

label:

The label is an optional field, except where noted in a particular statement description. Any source statement may be identified with a symbolic label, except a pure comment. A label must begin with a letter, A-Z, and must contain only letters, digits, and periods. A label may contain 1 to 25Ø characters, but 16 is a practical limit. The label may be terminated by a colon or a space (or by any other character that is not a letter, digit, period, or RETURN), and it will be ignored by the Assembler. Labels may not contain imbedded blanks.

The label must always begin in the first character position of the record. All labels or symbols must be unique: that is, any symbol must be defined only once in a given program, even a multifile program. The Assembler will flag a 'DUPLICATE SYMBOL' error if you attempt to create two identical symbols. All characters of a symbol are significant in determining uniqueness. When symbols are used on a source statement, except for the EQU directive, the value assigned to the symbol will be the current address as calculated by the Assembler for that line.

If a symbol is referenced in an operand field, the Assembler will substitute the value assigned to that symbol for that symbolic reference. All absolute symbols, or 16-bit symbols, may be defined anywhere in the program, but symbols referring to 'page zero' addresses, those less than 256 decimal, must normally be defined before they are referenced so the Assembler can determine the number of bytes required for the instruction on the first pass. If you fail to do this the Assembler is forced to assume an absolute address mode instruction.

It is good programming practice to define all data symbols, as opposed to program symbols, at the beginning of your programs. This is facilitated by the DSECT and DEND directives in the Assembler.

THE OPERATION CODE

The second field of the source statement is the opcode field. It must always be separated from the label field by at least one space. If no label is present, the opcode must be preceded by at least one space. Opcodes consists of two or more letters terminated by a space. These mnemonics are the same as the MOS Technology mnemonics, with a few added synonyms for the branch instructions. The assembler directives used are not the standard MOS Technology ones. A complete table of opcodes and directives is found in the appendix on Object File Formats.

Assembler directives, or pseudo-operation-codes (pseudo-ops), are entered in the opcode field and used just like opcodes. Pseudo-ops are instructions to the Assembler and direct the course of the Assembly, but do not appear in the object code.

THE OPERAND FIELD

The operand field of the statement is required for some opcodes and not for others. It generally contains an expression formed out of constants, labels, and arithmetic operators. These operations have no precedence other than left-to-right occurrence. The Assembler recognizes the standard 65Ø2 MOS Technology address mode syntax. A summary of this is given on page 49, but if you are not familiar with 65Ø2 programming, it is suggested that you read one of the many 65Ø2 Programming Manuals before you start coding.

THE COMMENT FIELD

The comment field, always optional, is used to document what the program is doing. The Assembler ignores the comment field, and just prints it with the rest of the statement when the program is listed. The comment may contain any arbitrary set of ASCII characters and should be separated from the operand field by a space and a semicolon (;). When the Assembler can unambiguously determine where the previous opcode or operand field ends, it does not need the space and semicolon; but the editor's TRuncation facility does require these two characters.

The Assembler also recognizes statements that begin with an asterisk, * , in the first character. These statements are treated entirely as comments, and are listed and ignored by the Assembler. The semicolon may also be used as the first character to indicate a comment.

FORMING THE OPERAND FIELD

Before describing the assembler directives and, briefly, the operation codes, it is necessary first to present the form of the operand field or expression. Expressions consist of simple operands--such as labels, constants, and reserved words--combined into expressions with arithmetic operators. The Assembler performs the expression evaluation during pass two, after all symbolic labels have been defined.

In determining whether an expression is to return a 16-bit or an 8-bit expression, and thus change the length of an instruction, the Assembler only looks at the first simple operand in an expression

during pass one. Thus it is possible to force a long, three-byte, instruction to be generated with a zero-page address. This is done by subtracting an absolute symbol from itself and then adding the desired zero-page sub-expression.

LABELS

Labels are, in general, symbolic names for a 16-bit value of some kind. For the 65Ø2, there are two kinds: values whose most significant 8 bits are zero, and values whose most significant 8 bits are not zero. Those of the first kind are called zero-page labels. The value assigned to a label is determined by the type of statement on which it occurs. If a label occurs on a statement that generates machine code or reserves memory space, the label is given the value of the program address counter that it labels. This will always be a 16-bit value, of one type or the other. If a label occurs on an EQUate pseudo-operation, the label is given the value of the operand expression evaluated.

When a label appears in the operand field, the Assembler substitutes the value for the label in calculating the value of the operand.

CONSTANTS

The Assembler will recognize four types of constants in the operand expression: string constants and three types of numeric constants. Constants are used to represent actual data items, such as ASCII characters, or tables of data, as well as address offsets, or absolute addresses of hardware devices, or locations fixed by the 65Ø2 microprocessor design. A constant has a fixed value that is evident from looking at the constant itself, whereas a label has a value that may change if the program is modified or located elsewhere in memory.

Decimal Constants

Decimal constants represent numbers in base 1Ø. A decimal constant is a positive integer between Ø and 65535, composed of a sequence of decimal digits from Ø to 9. Any numeric constant is assumed by the Assembler to be decimal unless it is preceded by the hexadecimal or octal radix character. If a numeric constant evaluates to a number larger than a 16 bit binary number, the Assembler generates a numeric overflow error message for that expression.

Hexadecimal Constants

Hexadecimal, or Hex, constants represent numbers in base 16. The characters used to represent Hex constants are the decimal digits Ø to 9, plus the letters A to F for the decimal values 1Ø thru 15. Hex

constants must be preceded by the Hex radix character, the dollar sign, $. The $ may not be used for any other purpose in numeric constants, but is allowed as a character in a string constant. The Assembler terminates what it considers a numeric constant of any kind when it finds a character that is not legal for that type of constant. In addition, if such a character terminates the last simple operand of an expression, that character will be treated as the beginning of the comment field.

Octal Constants

Octal constants represent numbers in base 8. They are not commonly used with micro-computers, but many programmers with long years of experience learned on them, so they are included as a convenience for those who like to use them. The digits used to represent Octal constants are the decimal digits Ø to 7. Octal constants must be preceeded by the Octal radix character, the 'at' sign, @. The @ may not be used for any other purpose in numeric constants, but is allowed in a string constant. Octal constant are terminated in the same manner as Hex constants, except that the digits 8 and 9 are not valid octal characters.

String Constants

String constants represent sequences of ASCII characters, and are represented by enclosing the characters between single quotes, '. A string may not cross statement boundaries, and may be up to 24Ø characters in length. When a string constant is used as the operand of an immediate-mode expression, it need not have the trailing quote, since such a string may only be one character in length. The value of a character is its corresponding ASCII code plus the most significant bit (bit 8 in ASCII terms). The value assigned to the MSB is determined by the MSB assembler directive, unless it is used with the DFI pseudo, which has a standard pattern associated with the MSB.

RESERVED WORDS

Normally, a 65Ø2 assembler is supposed to prevent the use of the one-character reserved words, A, X, Y, P, and S, as labels in source programs. This assembler will allow you to use any of these as a label on a statement, but will not allow the use of the character A as a label in an operand expression. It is suggested that you NOT use these or any other one-character labels, to keep your source programs compatible with the standard 65Ø2 syntax.

ARITHMETIC OPERATORS

The Assembler supports the four arithmetic operators +, -, *, and / for use in creating simple linear address expressions. In performing these arithmetic operations, the Assembler does not check for numeric overflow of the results: it just retains the 16-bit results, thus allowing wrap-around address calculations. When the Assembler has been directed to generate a Relocatable output module, it will not allow the multiplication and division operators to be applied to a relative symbol, as this would degenerate to a non-relocatable result.

ADDRESS EXPRESSIONS

The Assembler provides for simple linear address expressions consisting of simple operands and the above arithmetic operators. The valid syntax for an expression, in Backus-Naur Form (BNF), is as follows:

```
Sopand := Symbolic Label | Constant

Aop := + | - | * | /

Expression := Sopand [Aop Sopand [Aop Sopand]...]
```

This syntax definition says 'An expression is a simple operand optionally followed by one or more arithmetic-operator-and-simple-operand sequences'. The expression is evaluated from left to right.

The syntax definition also indicates that an Aop may not be first in an expression, nor may an Aop be last. The address expression must not contain blanks: if it contains a blank, the part after the blank will be read as a comment. Failure to follow the above rules in forming expressions will usually result in a BAD EXPRESSION error for the offending source line, during pass two of the assembly.

ASSEMBLER DIRECTIVES

The directives in this section are used to direct the overall operation of the assembly process, and to identify to the Assembler those labels that are fixed addresses or have special meanings. Each directive is used like a normal operation code in the source line. Some of these pseudo-operation-codes require labels, since they are used to establish the value of a label or some special characteristic of a label. Any of these directives may be preceded by a label, followed by a comment, or both.

ORG

ORG expression

The ORG directive establishes the origin of the object code. The label is optional, and the symbolic labels used in the expression must have been defined previous to the ORG statement. The presence of the ORG statement is required to cause the generation of an output file from the Assembler. If no ORG statement occurs the Assembler will produce a listing without producing an object file. The Assembler recognizes two kinds of ORG statements, absolute and relative. A relative ORG is one which contains relocatable symbols, and thus just updates the position in the current object file. A few examples of this are:

```
ORG *+5
ORG SYM+1Ø
ORG *-2
```

These relative ORGs cause the Assembler to position to a new position in the output file for the generation of the output code. The Assembler will position both forward and backward.

An absolute ORG is one which contains an absolute expression, which normally means a constant. Each absolute ORG is used by the Assembler to define the address to assign to code generating statements. It is also used to control the address placed in the output file, which will be used by DOS when the object file is later BLOADed or BRUN. The Assembler generates a new output file for each absolute ORG it encounters during an assembly.

Normally only one such ORG will occur in an assembly, but when a second one is encountered, the current one is closed and a new output file is started. The last character of the object filename is incremented by one, each time this occurs, to form a unique filename. It is your responsibility to make proper use of these multiple output files or to combine them later for execution if that is required. When RELocatable object files are being generated, the RLD is cleared each time an absolute ORG is encountered, so that each segment of such an assembly will have its own separate RLD.

OBJ

The OBJ directive has been included for compatibility with a previous version of this Assembler, in which it was used to specify the memory address of the output object file, which now can only be written on the output diskette. This directive is reserved for possible future enhancement of the Assembler/Editor system.

EQU

label EQU expression

The EQUate directive is used to assign a value to a symbolic label, where the label must be present, and must not be used for a label on another statement. The Assembler evaluates the expression, during pass one, and assigns this value to the symbolic label.

The purpose of using symbolic labels is to create a source program that means something to people, rather than just to the computer. When a program is written, extensive use of this directive will create a program that is easy to change and understand a year after it is written. The EQUate statement provides an easy way to name even an ASCII character that might serve as a special delimiter in a program. Using this directive, rather than just using a string constant in many places, makes it possible to change how a program functions without having to edit many lines of the program to make such a change.

MSB

MSB ON

or

MSB OFF

The MSB directive provides a means of controlling the value of the most significant bit or MSB of the ASCII characters as they are generated by the Assembler. The MSB pseudo may be changed as many times as needed during an assembly program. The ASCII characters affected by MSB are those generated as immediate string constants, and the string operand of the ASC pseudo, but NOT the DCI pseudo.

The Assembler defaults to MSB ON because the APPLE II expects ASCII characters this way for normal display.

DSECT

DSECT

The DSECT directive is used to define an area of memory, such as a data table, or command control block, without actually generating any output object code. The DSECT directive is used to mark the beginning of a block or group of statements that define the values of the labels that will be used to reference such a memory area. The most common use of the DSECT is to define the labels of data items, and pointers, etc., that occur in the 65Ø2 Page Zero area of memory.

The DSECT will cause an implicit ORG to address zero and will temporarily suspend object code output. The DSECT may contain most of the Assembler statements, including the ORG statement. The name DSECT comes from Dummy SECTion, so called because it generates no output image. Once a DSECT has been started, it remains in force until the occurance of the DEND directive. If an ORG statement occurs within a DSECT, it is used only to control the addresses assigned to labels within the DSECT, and does not function as a normal ORG as described above, nor does it change the current address, or program counter, which has been saved for the duration of the DSECT.

It is not valid to try to nest DSECTs, by having a DSECT inside another DSECT.

DEND

DEND

The DEND directive is used to signal the end of the current DSECT, and the resumption of code generation at the saved program counter address.

If a DSECT is started and never ended by the DEND directive, the remainder of your program, following the DSECT statement, will be listed just like a normal assembly, but no output will be generated. This missing DEND problem can be the source of mysterious loss of proper object code in your output files.

REL

REL

The RELocatable directive causes the Assembler to create a relocation dictionary for use by a relocating loader program. The RLD is only produced during the assembly process and written to the object file if the REL pseudo occurs in the source program. When REL is used, the object file is given a new type character, the letter R in the DOS catalog. This file has a new format defined in a later section, and can NOT be used by the DOS BLOAD and BRUN commands. The RLOAD program is discussed in Appendix C. The format of the REL pseudo is the same as the DEND pseudo except for the mnemonic.

The REL pseudo must occur at the very beginning of a source file, before any symbols are used, to operate correctly.

EXTRN

EXTRN label

The EXTeRNal directive is used to indicate a reference to an externally-defined symbol. These symbols are always treated as two-byte (or long-form) symbols, never as zero-page symbols. The Assembler generates zeros in the address portion of the instructions referencing symbols that are defined as EXTeRNal. If REL is used, and symbols are defined as EXTeRNal, the Assembler adds an External Symbol Directory after the RLD in the relocatable object file. This dictionary would be used by a Linking Loader program, which would resolve these external references from other modules, possibly in a library of modules.

The External Symbol Directory, or ESD, that could be added to a RELocatable file is cumulative, for all segments of multiple output file assembly, since this information resides in the symbol table, and not in the RLD table.

The optional label, in the label field of the EXTRN, serves only to define that label with the current value of the program address. The label after the EXTRN pseudo is the label defined as the EXTeRNal symbol. The EXTRN pseudo should not be used inside a DSECT.

ENTRY

ENTRY label

The ENTRY directive is the complement of the EXTRN pseudo, and is used to define what symbols of a program may be EXTRNed, or referenced, by another program. More than one of these may be used in a program, to define alternative entry points for the module. These symbols are then marked in the ESD along with the necessary information to link other modules to this entry point.

The label after the ENTRY pseudo is the label that is defined to be the entry point label. The value of this label is normally defined elsewhere, but it could be the optional label field of the ENTRY statement. If this is done, the entry point is defined as the current value of the program counter at the point of the ENTRY statement. The ENTRY statement should not be used inside a DSECT.

Both the ENTRY and EXTRN psuedos may be used even when REL is not used, and have been included in the Assembler for completeness even though no Linking Loader has been written to make use of the ESD. Using these directives provides a useful way of defining an address that is filled in at execution time, when self modifying code is being created. The ENTRY pseudo never creates anything in the machine code portion of the output file; the EXTRN pseudo allows an undefined symbol to remain undefined without generating an error message to that effect.

CHN

CHN sourcefilename[,slot expression [,drive expression]]

The CHaiN directive is used to connect together the segments of a large source program. The slot and drive expressions are optional; the filename is required. All statements following a CHN directive will be ignored; thus CHN would normally be the last statement of a source file. The Assembler will abort with an OOPS error 6 if the filename given does not exist on the current or specified slot and drive. The Assembler does not check to see whether such a slot and drive actually exist, so if you address a non-existent slot or drive, DOS will return an I/O error that will abort your assembly, with an OOPS error 8.

Slot expression and drive expression are checked for valid range, of 1 to 7 and 1 or 2 respectively.

LISTING DIRECTIVES

The listing directives, or pseudo-ops, are designed to provide control over the format and presentation of the Assembler's output listing. The use of these directives, is entirely optional, but using them does save space in the source files and improves the readability of your video and printed listings.

Most of these listing directives will accept a label within the label field itself. These embedded pseudos will be assigned the current program counter address, but many of these pseudos do something, and do not print the actual line on the listing. This can result in a label being defined but not printed, although it will be visible when using the editor. It is not recommended that labels be defined in this manner.

PAGE

PAGE

This directive causes a page eject to occur by sending an ASCII form-feed character to the output device. It also sends a blank line to the APPLE video screen at the same time. The PAGE directive itself does not print as a line on the listing, but its presence is shown by its action and the 'missing' line number in the listing. It may be found, when using the editor, by editing the missing line number.

LST

LST ON

or

LST OFF

The LiST directive provides a means of supressing part or all of the source listing. Turning the listing off, using this directive, can increase the speed of assembly; this will be most noticeable when a large assembly is being done to a printer. Any number of these directives may be used to selectively turn on and turn off various parts of the listing.

REP

REP expression

The REPeat directive is used to create a string of characters, starting in the first character of the source line part of the listing. This is commonly used to create a line of asterisks to set off comment headings at the beginning of subroutines or modules. This pseudo-op conserves source program file space by compressing the number of characters required to create a long line of the same thing to 6 or 7 characters. The default character that is repeated is the asterisk (*): it may be changed with the CHR directive described below. Any number of REP directives may be used, in the following format.

The expression is treated modulo 256: from 1 to 256 of the currently defined CHR will be printed. In other words, if the value of the expression is a number longer than 8 bits, only the least significant 8 bits will be used, the rest ignored.

CHR

CHR ?

The CHaRacter directive is used to change the character repeated by the REP directive.

The ? represents the character you want to see printed. Any number of these may be used to change the character around for different parts of your fancy listing.

SKP

SKP expression

The SKiP directive provides a means of inserting some number of blank lines in the listing, by sending ASCII carriage returns to the output device. The device must provide its own Line Feed on CR if that device requires a LF to advance a print line on the paper.

The expression is treated the same as the expression of the REP pseudo.

SBTL

SBTL Dstring

The SuBTitLe directive provides a title line (specified by the Dstring) at the top of each page of the listing file. Using SBTL is optional, but it does provide a useful means of identifing a specific listing. This pseudo causes the first line of each page to contain the current subtitle followed by the Assembler ID Stamp, which is the date followed by a six-digit integer. The ID Stamp is kept in a Binary file named ASMIDSTAMP. The system will work without this file, so you can DELETE it if you choose.

DATA DEFINITION DIRECTIVES

These directives are used to allocate or define data areas within the assembler program. Special directives are provided for address tables and messages, as these data structures are very common in assembly programs. These directives may be preceded by labels, followed by comments, or both.

ASC

ASC Dstring

The ASCII directive defines a string of 8 bit bytes in the output object file that are filled with the ASCII values of the characters in the string constant of the ASC directive. All the bytes generated by the ASC pseudo are printed on the output listing, with the source line being printed on the first line of this output. Three or fewer bytes are printed on each source line, without a line number until all the bytes are printed. If a label is present on the ASC pseudo, it is assigned the current value of the program counter, which will be the address of the first character of the string constant in memory.

A Dstring (delimited string) is begun with any character that is not to be in the string, and optionally terminated with the same delimiter. The terminating delimiter may be omitted if the comment is also omitted. The MSB pseudo controls whether the MSB bit of each character in the generated bytes is a one or a zero.

DCI

DCI Dstring

This directive functions just like the ASC directive, except that the MSB pseudo does not control the MSB of each byte. Instead, all bytes, except the last, of the DCI string have a zero MSB and the last has an MSB of one. The format of this command is identical to the ASC format.

DFB

DFB expr[,expr...]

The DeFine Byte directive is used to separately define one or more bytes. The assembler evaluates each expression and uses the resulting value, modulo 256, as the value for each byte. A label on the DFB pseudo will have the address of the first byte generated. If the bytes that are generated are calculated from a relocatable expression, an entry will be made in the RLD, for each such byte, so that its value can be relocated. It is suggested that the DFB directive be limited to 5 to 1Ø expressions, and using multiple DFBs rather than a large number of expressions on one DFB.

The comma is the only valid delimiter between expressions.

DW

DW expression

The Define Word directive is used to define a two byte 65Ø2 word. A 65Ø2 word is special in that the lowest 8 bits of the 16 bit expression are stored in the first of the two bytes, and the most significant or high 8 bits are stored in the second byte. This is the

order that the bytes must be in to be able to use the 16 bit address as an indirect address pointer, such as is used by the indirect indexed and jump indirect instructions of the 65Ø2 mircoprocessor.

The label is given the value of the program address, which is the address of the first, or low order, byte of the word.

DDB

DDB expression

The Define Double Byte directive is like the DW directive, except that the bytes are stored in reverse order, with the high-order byte first and the low-order byte second.

DS

DS expression

The Define Storage directive is used to reserve a group of bytes without having to define what is to be put in the bytes. The expression of the DS directive must not have any forward references. The amount of space reserved by the DS pseudo is included in the size of the output object module, by performing a file position command. This means that if you accidentally enter a DS with an expression that comes up with a value of, say 48K bytes, you will suddenly get a very large output file. The expression is most often used as a relatively small constant, for small data areas, since large buffers and work areas should not be part of an object program, (unless of course you have disk space burning a hole in your diskette).

The label will be assigned the address of the first byte of the reserved space that is allocated. When a DS is used inside a DSECT no space is actually reserved, but this is an easy way to define a data structure that can have insertions made, by adding more DS statements, without having to edit any of the other statements before re-assembly.

CONDITIONAL ASSEMBLY DIRECTIVES

A basic conditional Assembly feature is included in the assembler. This feature allows the programmer to conditionally select alternative sections of source code for inclusion or exclusion in the assembled object file. Conditional assembly is most often used to write a single "generic" program which includes a number of possible environments, such as production versus test, or 'machine configuration x' versus 'machine configuration y'. Three directives

are used for conditional assembly: DO, ELSE, and FINish. The DO and FIN directives must always be used as a pair, which mark the beginning and end, respectivly, of the section of conditional source statements. The ELSE directive is only allowed within the range of a DO-FIN block of statements. All the control of the conditional assembly is at the DO statement; the ELSE statement is used to invert the condition determined by the DO statement. These directives may be preceded by labels, followed by comments, or both.

DO

DO expression

The DO directive performs two tasks for the assembler. One is to mark the beginning of a conditional block of source statements. The other is to evaluate its expression to determine if the block of statements is or is not to be assembled. The expression of the DO statement is evaluated during pass 1 and so it may not contain any forward references to undefined or external labels. If the result of the expression evaluation is non-zero, the assembler continues to assemble the statements within the block. In other words, the condition for assembly is TRUE or ASSEMBLE ON.

If, on the other hand, the expression result is zero, the assembler starts skipping source statements, although they are listed and marked as skipped, until the FIN statement terminates conditional assembly. Thus the conditon for assembly is FALSE or ASSEMBLE OFF.

ELSE

ELSE

The ELSE directive may only occur inside the conditional assembly block delimited by the DO-FIN directives. The ELSE directive complements or inverts the state of the condition that is in effect as the result of the expression evaluation of the preceding DO statement. Thus if the state is OFF, it is changed to ON, and vice versa. In other words, if the source is being skipped until FIN, the ELSE resumes assembly until FIN. If assembly was proceeding, then ELSE begins source skipping until FIN. This is designed to allow for alternative code blocks to be selected by a single sequence of assembler directives:

```
DO condition
block 1
ELSE
block 2
FIN
```

FIN

FIN

The FIN directive is used to terminate the conditional assembly block of statements. After a FIN terminates such a DO-FIN group, then assembly returns to the unconditional or assemble ON state. A FIN may not ever occur by itself.

Note the order of first letters of the Do, Else, Fin is alphabetical and it is also the only valid order for these three directives. The use of the ELSE directive is entirely optional. In addition, there need not be any statements between the DO and the ELSE, thus allowing a DO NOT construct. This DO NOT construct is useful if it is necessary to separate complementry blocks of source code with some common section of code that must occur before one alternative and after the other. This also allows the same expression to be used in widely separate areas of the program that must assemble in opposite ways on a common condition.

ADDRESSING MODE SUMMARY

Note that all required syntax may be preceded by an optional label.

Addressing Mode	Required Syntax
Implied (no address)	opc
Accumulator	opc A
Immediate	opc #expression
Low 8 bits of address	opc #>expression
High 8 bits of address	opc #<expression
Zero page	opc zpg-expression
Indexed X	opc zpg-expression,X
Indexed Y	opc zpg-expression,Y
Absolute	opc abs-expression
Indexed X	opc abs-expression,X
Indexed Y	opc abs-expression,Y
Indirect,Indexed X	opc (zpg-expression,X)
Indirect,Indexed Y	opc (zpg-expression),Y
Absolute Indirect	JMP (zpg-expression)

ASSEMBLER DIRECTIVE SUMMARY

ORG	ORiGin
OBJ	OBJect
EQU	EQUate
MSB	MSB
DSECT	DSECTion
DEND	DsectEND
REL	RELocatable
EXTRN	EXTeRNal
ENTRY	ENTRY
DO	DO if
ELSE	ELSE
FIN	FINish
PAGE	PAGE Eject
LST	LiSTing
REP	REPeat
CHR	CHaRacter
SKP	SKiP
SBTL	SuBTitLe
ASC	ASCii
DCI	DCI
DFB	DeFine Byte
DW	Define Word
DDB	Define Double Byte
DS	Define Storage

OPERATION CODE SUMMARY

```
ADC   A + M + C → A
AND   A and M   → A
ASL   C ← [7..Ø] ←Ø
BCC   Branch on C = Ø
BCS   Branch on C = 1
BEQ   Branch on Z = 1
BIT   A and M, M7 → N, M6 → V
BGE   Branch on C = 1
BLT   Branch on C = Ø
BMI   Branch on N = 1
BNE   Branch on Z = Ø
BPL   Branch on N = Ø
BRK   Force Break
BVC   Branch on V = Ø
BVS   Branch on V = 1

CLC   Ø → C
CLD   Ø → D
CLI   Ø → I
CLV   Ø → V
CMP   A - M → P
CPX   X - M → P
CPY   Y - M → P

DEC   M - 1 → M
DEX   X - 1 → X
DEY   Y - 1 → Y

EOR   A xor M → A

INC   M + 1 → M
INX   X + 1 → X
INY   Y + 1 → Y

JMP   Jump to New Location
JSR   Jump to Subroutine
LDA   M → A
LDX   M → X
LDY   M → Y
LSR   Ø → [7..Ø] → C

NOP   No Operation (PC=PC+1)

PHA   A → Ms     S-1 → S
PHP   P → Ms     S-1 → S
PLA   S+1 → S    Ms  → A
PLP   S+1 → S    Ms  → P

ROL   ┌← [7..Ø] ← C ←┐
ROR   └→ C → [7..Ø] →┘
RTI   Return from Interrupt
RTS   Return from Subroutine

SBC   A - M - C → A
SEC   1 → C
SED   1 → D
SEI   1 → I
STA   A → M
STX   X → M
STY   Y → M

TAX   A → X
TAY   A → Y
TSX   S → X
TXA   X → A
TXS   X → S
TYA   Y → A
```

SYMBOL TABLE LISTING

The Assembler has a third phase which produces an optional Symbol Table Listing. This listing is produced twice, first in alphabetical order, and then in address order. The Symbol Table Listing is suppressed if the assembly is aborted by typing a CTRL-C or if the Listing has been suppressed by the LST OFF pseudo or the CTRL-N command. Placing the LST OFF pseudo at the end of your source program will suppress the two Symbol Table Listings. Suppressing the entire Listing with a LST OFF pseudo at the beginning and putting a LST ON psuedo at the end will produce ONLY the Symbol Table Listings.

The Symbol Table Listing automatically adjusts the width of its output for the APPLE 4Ø Column Screen or a printer (which is assumed to be 8Ø columns or more). It will display the table in two columns on the screen and print it in four columns on the printer. The listing only

contains the first 14 characters of any label, and it only sorts up to that many as well.

The various special characteristics of labels can be determined from the format in which the addresses are printed and the special flag characters that print just before the address value. Below is a small example of the Symbol Table Listing. When a symbol address is printed with two leading spaces, this indicates that the assembler considered that symbol to be a zero-page address. If the two leading zeros are printed, it indicates that the assembler was forced to consider the address to be an absolute address, because of a forward reference, even though it could have been a zero- page address if it were defined before being used. This is usually corrected by moving a DSECT or a EQU up to the beginning of your program.

The ? before the address indicates a symbol that was defined but never referenced. The * indicates the symbol was referenced but never defined, thus causing one or more NO SUCH LABEL errors. The X before the address indicates the symbol is an EXTERNAL symbol and the N indicates the symbol is an ENTRY point symbol.

Here is a sample Symbol Table Listing:

```
SYMBOL TABLE       SORTED BY SYMBOL          27-JUL-79  #ØØØ9Ø PG 11

   6D ADPTR          6F ADTBLND        2D3E ALPHAS      24 CH
N2CØØ SYMDUMP       ØØ72 SYTYPE       2D6E SWAP     ?2DF4 SWIPEIT
 2E5F TABLEND      ?2E84 TESTLBL      ?  ØA TXTBEG      77 VAL
*2E99 XXXXXX       X  76 YYYAV
```

Due to limitations in the Assembler's syntax-checking, you may occasionally be able to enter an invalid line into a program, without having it rejected. If this happens, and you run the program, the system may hang, or the program may seem to run, but run incorrectly: when you look at the object code, the program counter may jump from $4ØØ to $2ØØØ, or a branch may go to tht wrong place. Two examples of incorrect lines that may be allowed into your program follow:

Case 1. A line should have a series of numbers, separated by commas:

```
n1,n2,n3
```

but instead you have a space after one number:

```
n1,n2, n3
```

When the assembler makes its first pass, it will get all the n's, but when it makes its second pass, it will take the space for the end of the line, and ignore n3.

Case 2. You type a blank line followed by a RETURN. The assembler, instead of flagging it, will reuse the last opcode, and will generate a spurious line of code.

These errors rarely occur--programmers have used this assembler for months without running into any--but if your program crashes for no apparent reason, check for this kind of error after ruling out the more likely sources of error.

APPENDICES

APPENDIX A:

EDITOR/ASSEMBLER MEMORY USAGE

The editor is written in a combination of 65Ø2 assembler language and the 'psuedo 16-bit machine' language Sweet 16. The Assembler does not itself generate Sweet 16 code. The Sweet 16 code interpreter is included in the Editor module (called EDITOR in the diskette directory) of the system. It has been reworked from the version in Apple II's Integer Basic ROM, to maximize the speed of the Editor. The Command Interpreter (EDASM) module is entirely in 65Ø2 assembly code; its main function is to parse the commands into parameter lists and to call the EDITOR. The Command Interpreter module also parses the ASM command into a parameter list and overlays the Assembler (ASSM) module and the EDITOR module from the diskette.

This memory map shows the areas used by the Command Interpreter, Editor, and Assembler modules.

Memory Address (in HEX)	COMMAND INTERPRETER/ EDITOR USAGE	ASSEMBLER USAGE
$ØØØØ to $ØØFF	SWEET 16 Registers, work variables, and flags for editing.	Work pointers and variables during assembly.
$Ø1ØØ to $Ø1FF	65Ø2 Hardware Stack	65Ø2 Hardware Stack
$Ø2ØØ to $Ø2FF	Editor Input Buffer for command entry and text entry.	Not used
$Ø3ØØ to $Ø3CF	Editor numeric input stack, permanent flags ASMIDSTAMP file	Input parameters and ASMIDSTAMP file
$Ø3DØ to $Ø3FF	Not used	DOS Interface JUMPS and SUBRS
$Ø4ØØ to $Ø7FF	Editor Screen Memory	Assembler Screen Memory

Memory Address (in HEX)	COMMAND INTERPRETER/ EDITOR USAGE	ASSEMBLER USAGE
$Ø8ØØ to $Ø8FF	Editor String Parameter Stack	Assembler Parameter List and Assembler Output Screen
$Ø9ØØ to $Ø9FF	Editor Command Stack Save Area	Assembler Output Screen
$ØAØØ to $ØAFF	Editor LIST Command Save Area	Assembler Output Screen
$ØBØØ to $ØBFF	Not used	Assembler Output Screen
$ØCØØ to $11FF	EDASM Module (65Ø2)	EDASM Module (65Ø2)
$12ØØ to $1D8Ø	EDITOR Module (65Ø2 & SW16) SWEET 16 Machine (65Ø2 code)	ASSM Module (65Ø2)
$2ØØØ to $318Ø	Edit File beginning (first line)	Approximate beginning of Symbol Table Area (first symbol) (last symbol) (last RLD entry)
HIMEM=	(last line)	(first RLD entry)
$96ØØ to $BFFF	DOS (48K)	DOS (48K)

This memory map is provided for reference only. Apple Computer Inc. reserves the right to change or expand the areas used at any time and without notice.

APPENDIX B:

DOS ERRORS WITH THE EDITOR

The Command Interpreter and Editor programs interface to DOS in the same manner that a BASIC program would. When the system is started up, it changes DOS so that the Command Interpreter is treated by DOS as its host language. This provides a means of trapping all the DOS error conditions and returning into the editor after they occur. In addition, reinitalizing the DOS via the usual *3DØG command will reenter the Command Interpreter module. Likewise, if you have the Autostart ROM, the RESET key will return you into the Command Interpreter, via DOS.

The system only uses the error-trapping aspect of this setup for detecting 'END OF DATA' or EOF conditions when reading input text files and when BLOADing the ASMIDSTAMP module. All error messages are displayed on the Apple screen before DOS returns into the system. Thus when a LOAD command is given, DOS responds with 'END OF DATA' when if reaches the end of the file and this shows on the screen. The DOS MON and NOMON commands may be given if you want to see exactly what the Assembler/Editor system is doing. The Direct DOS command facility is used to do this, and you can view a file being LOADed by typing

.MON I

or view the DOS commands by typing

.MON C

This table shows some of the ways in which DOS errors may happen. It is not the total list but it represents the most common errors.

DOS Error Message	Usual Command and Situation
WRITE PROTECTED	Attempting to SAVE to a diskette with a write-protect tab, or doing the ASM command with a write-protect tab on the Assembler/Editor diskette, which prevents the ASMIDSTAMP file from being updated with the new number.
END OF DATA	This normally occurs at the end of each LOAD command.
FILE NOT FOUND	This will occur immediately after the ASM command if you do not have a file named ASMIDSTAMP on your Assembler/Editor diskette. The system does not require that this file be present and the message should be ignored.

This can also occur if you try to issue a Direct DOS command, such as LOCK or UNLOCK, that cannot find a file.

I/O ERROR

This should not normally occur, but it could occur for LOAD, SAVE, or CATalog. If it occurs during a SAVE, your output file is probably bad and you should do a SAVE command onto another diskette. If it occurs during a LOAD, your file was only partially read in and may be permanently lost (backing up is wise). If it occurs for CATalog, you're in deep trouble and may have lost your diskette.

DISK FULL

This usually occurs during a SAVE: if you get this message, you should find another diskette to reSAVE your file onto, then clean up the full diskette if you want to put the file on it.

FILE LOCKED

This can occur if you are in the habit of LOCKing your files and try to SAVE to a file that is LOCKED. The LOCKed file remains intact, as does your current edit file. UNLOCK the file or change the SAVE name before trying to save the file again. It is excellent practice to lock all the files on your diskette except the one you are editing to avoid SAVEing with the wrong name.

SYNTAX ERROR

This is the result of issuing a Direct DOS command with incorrect syntax.

NO BUFFERS AVAILABLE

This can occur if you invoke the system after MAXFILES has been set to one and you somehow issue a DOS command that tries to use two files at once.

FILE TYPE MISMATCH

This will occur if you try to LOAD some kind of a file other than TEXT or try to SAVE using a name that is already in use by a another type of file, such as Integer or Binary.

PROGRAM TOO LARGE

If this occurs, you have tried to do a Direct DOS LOAD command and have clobbered the entire system, along with your edit file. Read the manual again for the warnings about this class of problems.

APPENDIX C:

THE RELOCATING LOADER

The Assembler can be used to generate relocatable object files that can be relocated after being loaded by a relocating loader. Note that a relocating loader is not a linking loader. The two programs on the Assembler/Editor system diskette, RBOOT and RLOAD, make up a Relocating Loader for ROM or Language Card Applesoft II.

These two programs provide a way to load one or more assembler modules from an Applesoft program and have the load address of each module returned after each module is loaded. The modules are loaded starting just below the current HIMEM setting and the new value of HIMEM is reduced by the length+1 of the object code portion of the REL object file. This simple method of memory management requires that no strings may be allocated before using the Relocating Loader, since no attempt is made to save any string data that might be allocated where the modules are placed in memory.

In addition to the above restriction of using the Relocating Loader prior to using strings variables, the programmer must not DIMENSION or allocate new numeric or string variables between using RBOOT and the last usage of RLOAD to pull in the desired modules. This is due to the way in which RBOOT and RLOAD occupy memory relative to Applesoft's variable tables.

RBOOT is a small program, very similar to the Applesoft CHAIN module, that is BLOADED into addresses $2Ø8 thru $3CF and invoked by doing a CALL 52Ø. Note the usual D$ for the DOS BLOAD command should be replaced with a CHR$(4)... remember NO strings!!. RBOOT is a small scanning relocator designed to load and relocate RLOAD into memory at least one page (256 bytes) above the end of Applesoft's variable table and set the Applesoft USR function jump address to point to the entry point of RLOAD. This means that RLOAD can be pulled in after variables have been dimensioned if desired, but before strings have been used in any way.

Now, to load a REL module, the programmer invokes RLOAD via the USR(Ø) function; RLOAD either returns the load point address or gives an ON ERR message that indicates the problem encountered during the attempted load. RLOAD and RBOOT both assume that the ON ERR statement is in effect prior to using RBOOT or RLOAD, and will not function correctly if it is not and an error is encountered.

The RBOOT function accepts no parameters; it assumes that RLOAD is on the diskette last accessed, which will normally be the same diskette from which RBOOT was BLOADED. The RLOAD function accepts three

parameters from the USR(Ø) statement. These parameters must be inside a quoted literal and may be separated from the USR function by a comma. The following example shows the syntax to be used:

```
1Ø ADRS = Ø : REM PRE ALLOCATE VARIABLE TABLE

2Ø PRINT CHR$(4);"BLOAD RBOOT":CALL 52Ø

3Ø ADRS=USR(Ø),"MYMODULE,S6,D1"
```

The slot and drive parameters are optional; the filename is required, and must be the name of a REL type file, not a BINARY type file. The slot and drive parameters may be in either order, and either or both may be omitted as needed. If the filename is omitted, the 'FILE NOT FOUND' error code is returned, instead of a 'SYNTAX ERROR' code.

The value returned by the USR function is a signed REAL result that can later be used to enter the loaded module via the CALL statement. This obviously assumes that the module begins with an executable code segment. An effective means of providing multiple entry points is to put a table of jump instructions at the start of the module. This allows doing CALLs to the returned ADRS, ADRS+3, ADRS+6, ADRS+9, and so forth, as a means of entering the various sub-functions in a module. It also allows the contents of the module to grow or shrink later and not disturb the BASIC program's interface to the module. Additional jumps can be added to the table later for new functions and not disturb the existing interface to the original entry points.

The programmer may load as many modules as he chooses, up to the available memory space minus the size of RLOAD itself. RLOAD is about 1.5K in length and always loaded on a page boundary by RBOOT. RLOAD requires that there be at least 1 free file buffer available that it can borrow from DOS . If this is not true, a NO BUFFERS AVAILABLE error will occur. The reduction of HIMEM by RLOAD is not restored, (except by entering Integer BASIC) and the programmer should save the initial value of HIMEM, by PEEKing it out of page zero, and restore it at program termination.

Care must be taken when testing a program that uses this loading tool, since repeated RUNning will cause RLOAD to bring in a new copy of a module for each run, and allocate new space each time. This can eat up all of memory in short order if HIMEM is not restored to its normal value before each test. An easy way to do this is to issue an FP command to DOS, and then RUN the program from the disk. This may add extra SAVES to the testing process, but that seldom proves to be a disadvantage anyway. Note that an FP command erases the program currently in memory!

Note that this relocating loader does not provide any support for the EXTRN and ENTRY pseudo-ops of the Assembler, although the necessary data is generated in the ESD by the Assembler, to support a linkage editor-loader.

APPENDIX D:

ASSEMBLER "OOPS" DOS ERROR CODES

The Assembler uses DOS throughout the assembly process, and it is possible for the DOS to fail to accomplish some of the functions requested of it by the assembler. None of these errors are normal; all are fatal to the assembly process. When they occur the Assembler will stop the assembly and print the message:

OOPS! DOS ERROR! CODE=xx

while beeping three times. The error codes and their meanings are shown in the table below.

When this happens, the Assembler will then abort the assembly and attempt to close all open files. The attempt to close may also fail with another OOPS error, in which case the Assembler will give up and return to the Command Interpreter. Normally, it will be possible to close open files, but in some cases--say, if you open the disk drive door and remove the diskette--it will be impossible, and you will get an error message.

The error codes are the ones used by DOS with the 'ON ERR GOTO' statement: they are also explained in the DOS Messages chapter of the DOS Manual. Many of the comments about causes for these errors apply to this system, so it is suggested that this chapter be read as background material.

If any other OOPS error codes occur, most probably the Assembler/Editor system has been clobbered in memory, due either to a software bug or a hardware failure. Please refer any repeatable errors of this kind to Apple Computer Inc. in writing: if at all possible send a diskette that will reproduce the problem. Include all relevant information: your machine type, memory size, peripheral cards installed, number of disk drives and controllers in use, and any modifications that might have been made to any of the above.

You should always attempt to recreate any problem with a fresh copy of the Assembler/Editor system diskette before concluding that you have found a program bug.

OOPS Error Code	Meaning and Usual Cause
Ø4	WRITE PROTECTED DISKETTE You have a write-protect tab on the diskette to which the assembler was told to write the output object file. This will not occur until the beginning of PASS 2.
Ø6	FILE NOT FOUND This is usually due to using the ASM command with a source file name that is not in existence or not on the diskette in the current SOURCE DRive and SLot. It can also occur when a CHN command is used and the needed file is not present on the proper diskette.
Ø8	I/O ERROR This is usually a read error but it can also be a write error caused by bad diskette media. If it is a hard read error on an input file, the same problem would show up when you try to LOAD that file. If it was a write error on the object file, that file will show up in the CATALOG as being only 1 sector, or it will get an I/O error when you try to BLOAD the file into memory.
Ø9	DISK FULL ERROR This occurs when there is no space left on the output file diskette for the output file. This can occur at any time during PASS 2, so it is wise to know that there is enough space on the diskette for the output.
ØC (12)	NO BUFFERS AVAILABLE This error is not likely to occur unless you have entered the system with MAXFILES set to 1. A normal assembly requires that there be at least 2 buffers available. If the ASM had previously 'OOPSed' out of an assembly and was unable to close all the open files, this situation could arise. You should execute a Direct DOS .CLOSE after this occurs, if the system doesn't die because it can't reload the editor.
ØA (1Ø)	FILE LOCKED This occurs if you have locked the object file.

APPENDIX E:

OBJECT FILE FORMATS

The assembler generates two kinds of DOS object files: Binary memory-image files and Relocatable binary code files. The format of the Binary files generated by the assembler is identical to the DOS Binary format, and this type of file will be generated unless the REL pseudo occurs at the beginning of an assembly source program.

These files may be BLOADed and, if properly coded, BRUN from the normal BASIC/DOS environment (NOT from within the Assembler/Editor system). A Binary file must begin its code at the first byte of the file if it is to be BRUN, and may not have data areas at the beginning of the file. A common way to do this is to put a JMP to the actual program at the very beginning of the file.

The Relocatable binary file type is an extension of DOS: the table below defines the format of this file type. The symbol => may be read as "indicates" in this context.

RELOCATABLE FILE FORMAT

Sector	Byte (Hex)	Contents of byte
1	Ø	Starting RAM address, low byte
	1	Starting RAM address, high byte
	2	Length of RAM image, low byte
	3	Length of RAM image, high byte
	4	Length of code image, low byte
	5	Length of code image, high byte
1 to N	6 to cl+6	Binary code image, of length in bytes 4 and 5 above
	cl+7	Begin Relocation Dictionary, which consists of N 4 byte entries. N is variable(Ø to??)
	1	RLD flags bytes containing 4 flag bits as follows
	$8Ø bit	Size of relocatable field Clear => 1 byte, SET => 2 byte
	$4Ø bit	Upper/Lower 8 of a 16 bit value Clear => low 8, Set => high 8

Sector	Byte (Hex)	Contents of byte
	$2Ø bit	Normal/reversed 2 byte field Clr => low-hi, Set => hi-low (the DDB pseudo causes Set)
	$1Ø bit	Field is EXTRN 16 bit reference Clr => not ext, Set => is EXTRN
	$Ø1 bit	'NOT END OF RLD' flag bit ALWAYS SET ON for RLD entry Clear marks end of RLD
	2	Field offset in code, low byte
	3	Field offset in code, high byte
	4	Low 8 bits of 16 bit value for an 8 bit field containing upper 8 bits, Zero if $4Ø bit clear in RLD byte one. Or if the $1Ø bit is set, then this is the ESD symbol number.
	N*4+1	Binary ØØ marks end of RLD.
	N*4+2	Beginning of optional External Symbol Directory (ESD). This area will only contain bytes if an EXTRN and/or ENTRY pseudo occurs in the program.
	1 to s1	The ext/ent Symbolic name of length s1 bytes where all bytes have their $8Ø bit set except the last one.
	s1+1	Symbol type flag byte defines which type of symbol ext/ent
	$1Ø bit	Set => EXTRN symbol type
	$Ø8 bit	Set => ENTRY symbol type
	s1+2	Ext/ent symbol number refered to by an RLD entry with EXT bit set on.
	s1+3	High byte of offset for entry type symbol, (low is in s1+2) for the ENTRY type of symbol.
	End mark	Binary zero byte marks end of the ESD entries, of which there may be zero.

APPENDIX F:

SYMBOL TABLE FORMATS

The symbol table generated during pass 1 of the assembly process is described here, along with the table format as it remains after the symbol table has been modified by the symbol-table sort and print (pass 3) routine. The symbol table will be in its modified form and the RLD may be clobbered, if the symbol table sort and dump was allowed to execute and it overwrote the RLD with its sort index table.

The symbol table is a variable-length entry format table with flag bits to signal the end of the variable-length name character string.

The basic format is (Symbolicname)(Flagbyte)(Low value)(High value)

Symbolicname — consists of 1 to n characters each with their $8Ø bit set with the last character's $8Ø bit reset.

Flagbyte — contains the bits which define the characteristics of the symbol and its value and how it can be used to generate instructions.

$8Ø bit — Forward Reference bit Set

This means that the symbol was referenced but not defined. This flag is reset when a symbol is defined, and if it remains set at the end of pass 1 the symbol is undefined and will cause the 'no such label' error during pass 2. Symbols with this bit set are printed by pass 3 with an '*' next to the 'address' (which is meaningless: it is simply the the value of the program counter at the first reference).

$4Ø bit — Unreferenced Symbol bit Set

The symbol was defined but never used as the operand of any instruction in the program. This bit causes the '?' to print next to the address value for an unreferenced symbol in the dump.

$2Ø bit — Relative Symbol bit Set

The symbol's value is a relative symbol rather than an absolute address. Relative means relative to the beginning of the module. It is used internally by the assembler when generating the Relocatable type of output file to cause an RLD entry to be created for any references to the symbol.

$1Ø bit EXTeRNal Symbol bit Set

The symbol was defined as an external symbol via the EXTRN pseudo. This causes the symbol to be put into the ESD and prevents the symbol from being considered undefined, even though no value is assigned to the symbol. Using such a symbol will cause an RLD entry to be marked as EXT and cause the external symbol number to be put in the RLD entry in place of the relative offset. EXTeRNal symbols can only represent undefined 16-bit values (not-8 bit or zero-page values).

$Ø8 bit ENTRY Symbol bit Set

The symbol is an entry point into the module that can be referred to by an EXTRN in another module. This causes the symbol to be included in the ESD for resolution by a linkage editor (not yet implemented).

$Ø4 bit MACRO name bit Set

The symbol is really a macro file name: the value bytes hold drive and slot respectively. This is not yet implemented.

$Ø2 bit NO Such Label Error bit Set

The symbol has caused one or more NO SUCH LABEL errors. This is used to prevent a duplication of a single error in the error summary table during pass 2.

$Ø1 bit Absolute Address bit Set

A forward reference forced the symbol to be considered a 16-bit value. Zero-page labels print in the symbol dump with blanks for the first two bytes. They print with two zeros when this bit is set. If the definition is moved forward so that the symbol is defined before it is referred to, reassembing the program may generate shorter, zero-page, instructions.

When the Symbol Sort and Dump routine executes, it modifies the symbol table format to speed up the scanning of the table for its second phase. The last character of each symbol has its high-order bit set on and the Flagbyte is changed. If the Flagbyte has its $8Ø bit set its value is changed by ORing it with $7E to set all bits on but the $Ø1 bit, which is retained, and the $8Ø bit is set off to mark the end of the Symbolicname. Thus if pass 3 is run, all Flagbytes will have their $8Ø bits reset, and undefined symbols will have a Flagbyte of $7E or $7F.

APPENDIX G:

EDITING BASIC PROGRAMS

It is possible to list a BASIC program into a text file and edit it with the Editor. The section Capturing Programs in a Text File of the DOS Manual explains how to create the text file from BASIC. After LOADing that text file you may perform many useful functions on the text file using the Editor. The Find command can be used to locate all statements that refer to a given BASIC variable name, or line number. The global Change command can be used to change all occurrences of a variable name to a new name, or to change all occurrences of a GOSUB to some other line number, etc.

Care must be taken when changing line numbers of statements that are part of the edit file, because is is possible to change a statement's line number but not the references to it elsewhere in the program. It would be best to avoid changing any line numbers anywhere in a BASIC program and use the Applesoft/Integer BASIC RENUMBER programs to perform that kind of change. The Edit command is very useful in changing characters within a line or adding a statement in the middle of a line.

The Editor will show two line numbers for every line of the BASIC program. The first line number is always the Editor line number, and the second is the line number of the BASIC statement. You may not use the line numbers in the file as line number parameters for Editor commands. Changing the order of lines in the text file without changing the line numbers will not change their order after re-entering the text file back into BASIC. When you are done editing, you must SAVE the text file back onto your diskette and get back into BASIC, via the END command, before you can re-enter the edited text file back into BASIC. After you are back in the proper BASIC language, you must enter a DOS command similar to the one below:

```
EXEC myprogram
```

This command will cause the entire text file to be read back from the diskette into BASIC, just as if you were typing it from the keyboard. Thus each line must begin with a line number if it is to get into the program, and the order of the BASIC lines in the file is not important, since the line number determines where the line goes in the program. This example assumes that you SAVEd your edited file in the Assembler/Editor system with a 'myprogram' name of your own choosing.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N