



BOX 43, AUDUBON, PA 19407  
(215) 631 - 9052

ARESCO ASSEMBLER/TEXT EDITOR FOR THE  
APPLE II

The Editor.....	2
The Assembler.....	18
Appendix A.....	49
Appendix B.....	53

Entire contents copyright © 1979, by ARESCO, Norristown PA  
Reproduction prohibited.

## THE EDITOR

Table Of Contents

I	INTRODUCTION.....	3
II	SAMPLE USE OF THE ARESKO TEXT EDITOR	
	Loading The Editor.....	4
	Entering The Editor.....	4
	Entering The Text.....	5
	Listing The Text.....	6
	Changing The Text.....	7
	* Editor Command Summary.....	8
III	BASIC EDITOR COMMANDS	
	Entering The Text.....	9
	Correcting Text During Entry.....	9
	Listing The Text.....	11
	Adding A New Line To The Text.....	11
	Resequencing Lines.....	12
	Locating Specific Characters In The Text.....	12
	Exiting From The Editor.....	13
IV	OTHER EDITOR COMMANDS	
	The Status Command.....	14
	The Assemble Command.....	14
	Saving The Text On Audio Cassette Tape.....	15
	Reloading The Text From Audio Cassette.....	15
	The Query Command.....	16
	Changing The Prompt Character.....	16
	Using The Assembler/Editor with An APPLE Printer....	17

## INTRODUCTION

Congratulations on your purchase of the ARESKO Text Editor for the APPLE II microcomputer. The editor is an integral part of the ARESKO Assembler/Text Editor Package For The APPLE II microcomputer, and will allow you to easily input and modify text in your APPLE II system memory area.

Although primarily designed to work with the ARESKO APPLE II Assembler, any text material may be input and edited using the ARESKO Text Editor.

The ARESKO Text Editor features:

- \* Line numbered text entry and editing
- \* Single-letter mnemonic commands
- \* Automatic linkage to the ARESKO Assembler
- \* Complete compatibility with APPLE audio cassette or disk interface
- \* Memory independence. Text may be stored anywhere in memory, and multiple text files may reside in memory simultaneously.

To operate the ARESKO Text Editor as described in this manual, the following minimum APPLE system is required:

- \* APPLE microcomputer system
- \* 16K RAM
- \* Apple disk or audio cassette for program storage





## SAMPLE USE OF THE ARESCO TEXT EDITOR

### Loading The Editor

The ARESCO Assembler/Text Editor is loaded into APPLE memory from the ARESCO program tape, using the monitor load routines. Type 2000.3800R on the APPLE keyboard, press PLAY on the cassette unit, and then press the carriage return.

### Entering The Editor

To start the editor, type 3743G on the keyboard, and press the carriage return. The editor will respond with BASE= on the screen. You should reply by typing the hexadecimal start address at which you wish your block of source text to begin. The editor will then print N OR O? on the screen, asking you if this is to be a new file or an old one. If you wish to edit text already placed in memory (from an audio tape load, for example), you would respond by typing O. For the purposes of our example, enter an N, since you will be creating a new text file. The editor will respond by printing a carriage return and line feed, and wait for you to enter the first line of source text for your file. Example 1 shows this initial dialogue. Note that throughout this document, what you type in will be underlined for clarity. Naturally, this underlining does not occur when you actually use the editor.

### Example 1

#### STARTING THE EDITOR

<u>3743G</u>	Starting address of the editor
BASE= <u>4000</u>	Hex start address of text file
N OR O? <u>N</u>	Specify new text file

### Entering Text

Every line of text entered into the editor must have a decimal line number between 1 and 9999. If the same line number is used twice, the new line replaces the previous line with the same line number. Here is what our sample would look like as it is typed in. Note that increments of 10 are used between each line number. This is not necessary, but it makes it easier to enter additional text lines in the future.

#### Example 2

##### ENTERING TEXT

```
10 POINTL=$FA  
20 POINTH=$FB  
30 VAL1  
40 VAL2  
50 PROG CLC  
60      LDA VAL1  
70      ADC VAL2  
80      STA POINTL  
90      LAD #00  
100     STA POINTH  
110     JMP START
```

Remember that you must type a carriage return at the end of each line you type in. Note that it is not necessary to include spaces (that is, more than one space) between fields. The assembler will automatically insert them later. The spaces are included in this manual for readability.

Listing the Text

To list the text you have just typed in, type the character P for Print, a space, and the character A for All. The editor will then type back all of the text which you have typed in.

Example 3  
LISTING TEXT

```
P A  
0010 POINTL = $FA  
0020 POINTH = $FB  
0030 VAL1  
0040 VAL2  
0050 PROG CLC  
0060     LDA VAL1  
0070     ADC VAL2  
0080     STA POINTL  
0090     LAD #00  
0100     STA POINTH  
0110     JMP START  
*ET
```

Note that the \*ET at the end of a print command signifies the End of Text.

### Changing The Text

To make our sample program acceptable to the assembler, we will have to add and change several lines (the program was entered incorrectly to allow you the experience of editing it). To add a line between two existing lines, simply type a line number between the two existing line numbers, and enter the text. Notice the new lines 25, 26, and 120 in example 4.

To change an existing line, type the same line number and the new contents for that line. Notice how lines 30, 40, and 90 were changed, in example 4. Remember that a single character in a line cannot be corrected by itself; the entire line must be retyped.

#### Example 4

#### CORRECTING TEXT

<u>25</u> *=\$0000	This is a new line
<u>30</u> VAL1 *==+1	This is a change
<u>40</u> VAL2 *==+1	Another change
<u>90</u> LDA #00	Correct spelling
<u>120</u> .END	New line
<u>26</u> START=\$1000	New line
<u>P A</u>	Request printing of text
0010 POINTL=\$FA	
0020 POINTH=\$FB	
0025 *=\$0000	
0026 START=\$1000	
0030 VAL1 *==+1	
0040 VAL2 *==+1	
0050 PROG CLC	
0060 LDA VAL1	
0070 ADC VAL2	



```

0080      STA POINTL
0090      LDA #00
0100      STA POINTH
0110      JMP START
0120 .END
*ET

```

You may edit and re-edit the text until you're satisfied that it is correct. Notice that it isn't necessary to enter the lines in line number order, but they will be entered into memory and assembled in line number order. It is important to remember that you must have at least one space separating the fields in each line of text, that each line ends with a carriage return, and that each line must begin with a line number.

Now that you've had an opportunity to see the basic operation of the editor, we will examine each editor capability in detail. Table 1 shows the editor command summary, explaining briefly what each command does.

Table 1  
EDITOR COMMAND SUMMARY

P A	Print all stored text
P n	Print text beginning at line n
F xyz	Find and print all lines containing string xyz
S	Print status - hex origin, end of text, and decimal number of lines in the current text file
R	Resequence line numbers by 5's
Q	Query - return to BASE= query
E	Exit to APPLE monitor
?	Set "?" as prompt character
?x	Set x as prompt character
A	Go to Assembler with current text file
K <sup>c</sup>	Control K - exit to APPLE monitor
T <sup>c</sup>	Control T - Save current text file on audio tape

## BASIC EDITOR COMMANDS

### Entering Text

Text is entered into memory by typing in the line of text preceded by a line number. Line numbers may range from 1 to 9999. There is no necessity to type leading zeroes, the editor will insert them automatically. Note that any line typed to the editor which does not contain a line number will be interpreted as an editor command. If the editor cannot recognize the line as a command, the error message BAD COM (bad command) will be printed on the screen.

It is not necessary to "format" the text by entering spaces. One space must separate each field on a line from the rest of the fields on that line, but you don't need to use the number of spaces required to make the text "line up" as it will after it is assembled. The assembler does that for you.

### Correcting Text During Entry

If you type in a portion of a text line and realize you have made a mistake, there are two methods of correcting the error:

1. Depress the control key and hit the "X" key (control X). The line will be ignored and a carriage return/line feed will be issued by the editor.
2. Backspace by pressing the back arrow key (←), then retype the offending character. The editor will back up each time you press the back arrow key, moving one space to the left each time you press the key. When you have retyped the offending character, type the rest of the line - do not press the return key until you've reached the end of the entire line. All characters on the line will be deleted, beginning with the character

under the cursor and continuing to the end of the line, as soon as you press the return key. The deleted characters will not be removed from the screen, however. Example 5 shows the results of backspacing to correct an error.

#### Example 5

##### USING ← TO CORRECT ERRORS

step 1     10 THIS IS A MESPILLING

now the cursor is positioned immediately following the G

step 2     Press the ← key 8 times, and type ISPILLING

now the line of test looks like this:

10 THIS IS A MEISPILLING

and the cursor is still positioned after the G

step 3     Press the ← key 10 times and type ISPELLING

now the line looks like this:

10 THIS IS A MISPELLINGG

and the cursor is positioned over the last G on the line.

Press the return key at this point. The offending G is not removed from the screen.

step 4     PA

The editor prints the line on the screen

0010 THIS IS A MISPELLING

Now that you have pressed the return key, the only way to correct the line is to retype it:

10 THIS IS A MISSPELLING

While you may well forget to continue typing the line after backspacing to correct an error (at least at first), it won't take long to get to the point where you do it automatically, since it works very much like the APPLE's own editor. Note that you cannot forward space using the ARESCO Text Editor, however.

### Listing The Text

You've seen how to list the entire text file by typing P A. If you wish to only list a portion of the text, type P, a space, and a line number. The editor will then print all of the text, beginning with that line number, and continuing through to the end of the file.

If you wish to only list a few lines of text, type P, space, and a line number. The editor will begin printing out the text, beginning with that line number. When you have seen all the text you wish to see, press any key. This will signal the editor that you wish it to stop listing, and it will stop printing the text and wait for another command to be entered.

Note that depressing any key while the assembler is running will also stop the assembler listing. However, you will be returned to the editor, not to the assembler.

### Adding A New Line To The Text

Each line you type will automatically be inserted in the text file in line number sequence. Thus, if you wish to add a new line between old lines 20 and 30, simply give the new line a number between 21 and 29. If you wish to insert a new line between two existing lines with adjacent numbers, you must first resequence the line numbers (see Resequencing Lines).

There is no restriction on the number of text lines or the size of the text file (except that line numbers must be in the range 1 - 9999). The only restriction, then, is the amount of memory available for such storage.



### Resequencing Lines

The assembler ignores the line numbers in your file when doing an assembly. Thus, the line numbers are a convenience for you when you are editing text. If you wish to resequence the line numbers, simply type an R and a carriage return. The editor will automatically resequence all the line numbers in your file. The first line will be the line number 5 and each line number which follows will be incremented by 5.

#### Example 5

#### RESEQUENCING LINES

```
1 LINE 1  
2 LINE 2  
3 LINE 3  
R  
P A  
0005 LINE 1  
0010 LINE 2  
0015 LINE 3  
*ET
```

### Locating Specific Characters In The Text

In a lengthy text file it is often desirable to be able to find all lines in the text which contain certain characters or groups of characters. This is done with the Find command. Example 6 illustrates the use of the Find command. First the entire text file is listed using the P A command, then all lines which contain references to POINTH are printed by issuing the

command F POINTH. The second portion of the example shows finding all lines which contain an asterisk by typing F \*. You must always type a space after the F.

### Example 6

#### FINDING SPECIFIED TEXT

```

P A
0010 POINTH = $FA
0020 POINTH = $FB
0025 *=$0000
0030 VAL1 = *+1
0040 VAL2 = *+1
0050 PROG CLC
0060     LDA VAL1
0070     ADC VAL2
0080     STA POINTL
0090     LDA #00
0100     STA POINTH
0110     JMP START
0120 .END
*ET

F POINTH
0020 POINTH = $FB
0100     STA POINTH
*ET

F *
0025 *=$0000
0030 VAL1 = *+1
0040 VAL2 = *+1
*ET

```

#### Exiting From The Editor

The user may return to the APPLE monitor by typing an E (Exit) command. Never exit the Assembler/Editor by pressing the RESET key. If your system uses the Apple Disk, and you press RESET while using the Assembler/Editor, you will have to reboot the DOS. You may return to the monitor using K<sup>C</sup> (control K) if you wish.

## OTHER COMMANDS

### The Status Command

Typing an S command to the editor will result in the editor printing out three numbers. The first number is the hexadecimal starting address of the current text file. The second number is the hexadecimal address of the end of the current text file. The third number is the decimal number of lines contained in the current text file. If you are entering a text file which you suspect may approach the capacity of your available memory, you can check the amount of memory being used from time to time as you enter the text. You should note, however, that the S command will return incorrect information if no text has been entered into your current text file.

### The Assemble Command

Typing an A command to the editor will terminate editor function and transfer control to the assembler program. The editor automatically configures the assembler for a memory-to-memory assembly of the current text file. You should note that the transfer is made to the assembler cold start entry point so this command may not be used in multiple file assembly. For the second and succeeding files of a multiple file assembly, you must exit from the editor to the monitor and then enter the resident assembler at its warm start point. See the assembler documentation for further details.

### Saving the Text on Audio Cassette Tape

To transfer your edited text to audio cassette for storage, first insert a cassette in your recorder and assure that the cassette recorder is properly connected to your system. To begin transfer of the text file to the audio cassette, hold down the control button (marked CTRL), and strike the character "T". Before striking the carriage return, put your cassette recorder in record mode and start the tape. Then strike the carriage return. Before recording, you should use the S command to find the beginning and end of your text area. Make a note of it on the tape. The editor will then transfer control to the audio cassette routines. When the tape has been properly recorded, control will return to the Editor.

### Reloading the Text from Audio Cassette

When you wish to place the text you previously stored on cassette back into memory for further editing or assembly, first prepare your cassette system for playback. Reload the cassette using the monitor tape load routine and entering the starting and ending address noted when you recorded the tape. When the tape has been read in, start the editor, give the starting address of the text in response to the BASE= query and answer "0" to the N OR 0? query. The text file is now ready for further processing.



### The Query Command

Typing a Q command to the editor returns you to the BASE= query of the editor. You may then specify a new origin for further text entry of other text files. Previous text files will not be disturbed unless the files overlap in memory.

### Changing The Prompt Character

It is sometimes convenient to have the editor type a prompt character at the beginning of each line when it is ready for input from the user. When the editor is initially entered, this prompt character is set to a null. If you wish a prompt character you simply type a question mark and carriage return and the editor will begin each line of input with a question mark as a prompt. If you wish to use a prompt character other than the question mark, type a question mark, the character you wish to use as a prompt, and a carriage return. For instance, typing a question mark, asterisk, carriage return will set the prompt character to an asterisk. To "turn off" the prompt character, simply type a question mark followed by a null (generated by depressing the control and shift buttons on your keyboard and striking the P key).

Using The ARESCO Assembler/Editor With An APPLE Printer

If you have a printer connected to your Apple II through an Apple Parallel Printer interface card, you may use your printer to print editor or assembler output. Before entering the editor type:

1P<sup>c</sup>        (turn on the printer board)  
I<sup>c</sup>40N      (initialize output)  
I<sup>c</sup>I        (turn screen back on)

Then type 3743G to start the Editor. Type P A when you are in command mode to list your source program or A to print the assembled output. When you have returned to monitor mode, type OP<sup>c</sup> to turn the printer off.

## THE ASSEMBLER

## Table Of Contents

I	INTRODUCTION.....	19
	* 6502 Instruction Set.....	21
II	ASSEMBLER/EDITOR OPERATION	
	Memory Space Requirements.....	22
	Reserving Space For The Symbol Table.....	22
	Reserved Memory Locations.....	23
	Assembling Large Source Programs From Disk Or Cassette Tape.....	23
III	INSTRUCTION FORMAT	
	General Information.....	25
	Constants.....	28
	Symbols.....	29
IV	ADDRESSING MODES	
	Symbolic.....	30
	Absolute.....	31
	Immediate.....	31
	Relative.....	31
	Implied.....	32
	Indexed.....	32
	Indexed Indirect.....	33
	Indirect Indexed.....	34
	* Instruction Addressing Modes.....	35
V	ASSEMBLER DIRECTIVES	
	.BYTE.....	36
	.WORD.....	36
	= (EQUATES).....	37
	.OPT.....	37
	.END.....	39
VI	ERROR MESSAGES.....	40

## INTRODUCTION

The process of translating a mnemonic or symbolic form of a computer program to actual machine code is called an assembly, and a program which performs the translation is an assembler. The symbols used and rules of association for those symbols are the assembly language. In general, one assembly language statement will translate into one machine instruction. This distinguishes an assembler from a compiler, which may produce many machine instructions from a single statement.

Normally, digital computers use the binary number system for representation of data and instructions. Computers understand only ones and zeroes, corresponding to an "on" or "off" state. Human users, on the other hand, find it difficult to work with the binary number system and hence use a more convenient representation such as octal (base 8), decimal (base 10), or hexadecimal (base 16). Two representations of the 6502 operation to "load" information into an "accumulator" are shown here:

10101001	(binary)
A9	(hexadecimal)

An instruction to move the value 21 (decimal) into the accumulator is:

A9 15	(hexadecimal)
-------	---------------

Users still find numeric representations of instructions tedious to work with, and hence have developed symbolic representations. For example, the preceding instruction might be written in assembly language as:

LDA #21	(assembly language)
---------	---------------------

In this case, LDA is a symbol for A9, Load the Accumulator. A computer program used to translate the symbolic form LDA to the numeric form A9 is called an assembler. The symbolic program is referred to as source code and the numeric program is the

object code. Only object code can be executed on the processor.

Each machine instruction to be executed has a symbolic name referred to as an opcode (operation code). The opcode for "store the contents of the accumulator" is STA. The opcode for "transfer the contents of the accumulator to index X" is TAX. There are 55 opcodes for the MOS 6502 processor (listed in Table 2). A machine instruction in assembly language consists of an opcode and (perhaps) operands, which specify the data on which the operation is to be performed.

Instructions may be labelled for reference by other instructions as shown in:

```
L2  LDA  #21
```

The label is L2, the opcode is LDA, and the operand is #21. At least one blank must separate the three parts (fields) of the instruction. Additional blanks may be inserted for ease of reading. Instructions for the ARESCO assembler have at most one operand, and many instructions have none. In these cases, the operation to be performed is completely specified by the opcode, as in CLC (clear the Carry bit).

Programming in assembly language requires learning the instruction set (opcodes), addressing conventions for referencing data, the data structures within the processor, as well as the structure of assembly language programs.

TABLE 2

## 6502 Instruction Set - Opcodes

ADC	Add with Carry to Accumulator	LDA	Transfer Memory to Accumulator
AND	"AND" to Accumulator	LDX	Transfer Memory to Index X
ASL	Shift Left One Bit (Memory or Accumulator)	LDY	Transfer Memory to Index Y
BCC	Branch on Carry Clear	LSR	Shift One Bit Right (Memory or Accumulator)
BCS	Branch on Carry Set	NØP	Do Nothing - No Operation
BEQ	Branch on Zero Result	ØRA	"OR" Memory with Accumulator
BIT	Test Bits in Memory with Accumulator	PHA	Push Accumulator on Stack
BMI	Branch on Results Minus	PHP	Push Processor Status on Stack
BNE	Branch on Result not Zero	PLA	Pull Accumulator from Stack
BPL	Branch on Result Plus	PLP	Pull Processor Status from Stack
BRK	Force an Interrupt or Break	RØL	Rotate One Bit Left (Memory or Accumulator)
BVC	Branch on Overflow Clear	RØR	Rotate One Bit Right (Memory or Accumulator)
BVS	Branch on Overflow Set	RTI	Return From Interrupt
CLC	Clear Carry Flag	RTS	Return From Subroutine
CLD	Clear Decimal Mode	SBC	Subtract Memory and Carry from Accumulator
CLI	Clear Interrupt Disable Bit	SEC	Set Carry Flag
CLV	Clear Overflow Flag	SED	Set Decimal Mode
CMP	Compare Memory and Accumulator	SEI	Set Interrupt Disable Status
CPX	Compare Memory and Index X	STA	Store Accumulator in Memory
CPY	Compare Memory and Index Y	STX	Store Index X in Memory
DEC	Decrement Memory by One	STY	Store Index Y in Memory
DEX	Decrement Index X by One	TAX	Transfer Accumulator to Index X
DEY	Decrement Index Y by One	TAY	Transfer Accumulator to Index Y
EØR	Exclusive-or Memory with Accumulator	TSX	Transfer Stack Register to Index X
INC	Increment Memory by One	TXA	Transfer Index X to Accumulator
INX	Increment X by One	TXS	Transfer Index X to Stack Register
INY	Increment Y by One	TYA	Transfer Index Y to Accumulator
JMP	Jump to New Location		
JSR	Jump to New Location Saving Return Address		



## ASSEMBLER/EDITOR OPERATION

### Memory Space Requirements

The Assembler/Editor resides in memory in locations 2000 - 3800. In addition, the programs use locations 0300 - 0308 and 1E00 - 1FFF for workspace. Memory space must be allocated by the user for storage of the symbol table.

### Reserving Space For The Symbol Table

The assembler will generate object code and store it at the locations specified in the assembly language program. RAM must exist in your APPLE at these locations. Since your source program is also memory-resident, your object program should not be written to the same RAM area wherein the source code resides.

In addition to space for source code and object code, you must reserve space in memory for the assembler to store its symbol table while it is constructing your object program. Each symbol you define in your source program will take 8 bytes of memory in the symbol table. For example, if you expect to use about 50 symbols in your source program, you should reserve at least 400 bytes for the symbol table. If the assembler runs out of room in the symbol table, the program will terminate with an error message.

You define the RAM area you wish to reserve for the symbol table by entering the upper and lower address limits of the area in two pairs of locations. At location 1FDF and 1FE0, you enter the low order and high order bytes, respectively, of the starting address of your symbol table. At locations 1FE1 and 1FE2, enter the location of the ending address of your symbol table. For example, to reserve locations 3802 through 3B00 for your symbol table, enter the monitor and type (when the APPLE monitor's asterisk prompt appears)

```
1FDF: 02 38 00 3B
```

If you specify these locations before entering the editor to type in your source program, you will be able to go directly to the assembler (with the A command) with the symbol table already defined.

If you elect to specify addresses for the symbol table after entering your source program, you must define the symbol table addresses before using the A command to call the assembler.

### Reserved Memory Locations

The ARESCO Assembler/Text Editor uses all page zero memory locations. Object code should not be assembled into those locations. You may use page zero locations for data storage, but remember that the contents of those locations will be altered by the assembler and editor during their operations.

### Assembling Large Source Programs From Disk Or Cassette Tape

If you wish to assemble a source program which will not fit all together in your available memory, you may enter it in segments, save the source code on disk or tape, and even assemble it in segments. (provided, of course, that the fully assembled object code will fit in your APPLE's available memory.)

Enter the first portion of your source program into the text editor and store it on audio cassette or disk (see the section on Saving Text On Audio Cassette Tape for details). Then enter the next and succeeding portions in the same source code memory area, designating each segment as a new file to the editor. Save each portion on audio tape. Only the last segment should contain an .END directive.

Now play the first segment back into the same source code memory area, using normal audio tape or disk loading procedures. Enter the editor, giving the base address of the text file and declaring it to be an old file. Assemble the first segment,

using the A command. (Remember to define your symbol table area first.) The assembler will return to the monitor when it comes to the end of the file. There is no need to save this assembled segment on tape or disk.

Now load and enter the second segment of your source code file into the same source code memory area, using the audio tape or disk loading procedure. Enter the editor, give the same base address as you used for the first segment, and declare it to be an old file. Do not use the A command to call the assembler. Exit to the APPLE monitor, and enter the assembler via location 3782 (type 3782G). The assembler will continue to assemble your program - right where it left off with the previous file - without clearing the previously generated symbol table or destroying the previously assembled program segment.

Continue to enter successive segments of text, entering the editor to give the base address each time. Exit to the monitor and re-enter the assembler at 3782. When the assembler encounters the .END directive, it will print the symbol table and terminate the assembly process. This is the time to save the assembled object code on APPLE disk or cassette.



## INSTRUCTION FORMAT

Assembler instructions for the ARESCO Assembler are of two basic types, according to function:

1. Machine Instructions
2. Assembler Directives

Machine instructions correspond to the 55 operations implemented on the MOS 6502. The instruction format is

(label)      opcode      (operands)      (comments)

Fields are in parentheses to show that they are optional. Labels and comments are always optional and many operation codes (opcodes) such as RTS (Return from subroutine) do not require operands. A typical instruction using all four fields is:

LOOP            LDA            BETA,X            FETCH BETA INDEXED BY X

A field is defined as a string of characters separated by a blank space or tab character or characters. The list of opcodes for the ARESCO Assembler is shown in Table 1.

A label is an alphanumeric string of from one to six characters, the first of which must be alphabetic. A label may not be any of the 55 opcodes and may not be any of the special single characters A, S, P, X, or Y. These special characters are used by the assembler to reference the Accumulator (A), the Stack pointer (S), the processor status (P), and index registers X and Y respectively. A label may begin in any column, provided it is the first field of an instruction. Labels are used on instructions such as branch targets and on data elements for reference in operands.

The operands portion of an instruction specifies either an address or a value. An address may be computed by expression evaluation, and the assembler allows considerable flexibility in expression formation. An assembly language expression consists of a string of names and constants separated by operators +, -, \*, and / (add, subtract, multiply, and divide). Expressions are evaluated left to right, with no operator precedence and no parenthetical grouping. Expressions are evaluated at assembly time and not at execution time.

Any string of characters following the operands field is considered to be comments, and is listed but not processed further. If the first non-blank character on a line is a semi-colon (;), the line is processed as a comment. On instructions which require no operand, comments may follow the opcode. A semi-colon need not precede the comment if the comment is on the same line as an instruction. At least one space must separate the fields of an instruction.

There are five assembler directives used to reserve storage and direct information to the assembler. Four have symbolic names with a period as the first character. The fifth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. A list of the directives is given here, but their use is explained later in the section on Assembler Directives.

```
.BYTE
.WORD
.OPT
.END
=
```

Labels and symbols other than directives may not begin with a period.

When using the assembler, remember that if you press any key during the assembly, the assembly and listing will stop, and you will be returned to the editor.

A typical assembler program segment is shown on the following page to illustrate the form of the information provided by the assembler. The formatting of text is done automatically unless you specifically select the NOTAB option (see Assembler Directives).

Note the semi-colons preceding comments which occupy separate lines (that is, do not occupy lines containing opcodes).

Example 1  
A Typical Assembly Language  
Program

```

213 076A 20 60 09 ALPHA JSR GETINS FIND START OF NEXT INSTR
214 076D A9 00 LDA #0
215 076F 85 ID STA EFLAG
216 0771 85 1E STA DFLAG NO DATA OR EFFECTIVE ADDR YET
217 ; PICK UP THE OPCODE AND BREAK IT INTO ITS PARTS
218 0773 A5 14 LDA OPCODE
219 0775 29 03 AND #%11
220 0777 85 13 STA GROUP BITS 1,0 = GROUP CODE
221 0779 A5 14 LDA OPCODE
222 077B 29 FC AND #%11111100
223 077D 4A LSR A
224 077E 85 10 STA B72
225 0781 AA TAX
227 0782 29 07 AND #%111
228 0784 85 12 STA B42
229 0786 8A TXA
230 0787 4A LSR A
231 0788 4A LSR A
232 0789 4A LSR A
233 078A 85 11 STA B75
234 078C 20 79 09 JSR SETUP GET DATA FROM IT
235 ; SEE IF WE HAVE A LABEL TO PRINT
236 078F AF 15 LDA IADR
237 0791 85 27 STA NUMBER
238 0793 AF 16 LDA IADR+1
239 0795 85 28 STA NUMBER+1
240 0797 20 IC 08 BETA JSR NUM PRINT CURRENT P.C.

```

label  
operand  
opcode  
comments

object code  
address  
memory  
number  
line number

### Constants

Constant values in assembly language can take several forms, as needed by the programmer. If a constant is other than decimal, a prefix character is used to specify type.

\$ (Dollar sign)	specifies hexadecimal
@ (Commercial "at")	specifies octal
% (Percent sign)	specifies binary
' (Apostrophe)	specifies an ASCII literal in immediate mode instructions
# (Pounds sign)	Specifies immediate mode

The absence of a prefix symbol indicates decimal value. In the statement

```
LDA BETA+5
```

the decimal number 5 is added to BETA to compute the address. Similarly, in the statement

```
LDA BETA+$5F
```

the hexadecimal value 5F is to be added to BETA for address computation. If you wish to load an ASCII character (for example, the letter "G") into the accumulator, you would use the apostrophe (single quote), as in the statement

```
LDA #'G
```

It isn't necessary to use a closing quote unless you have embedded quotes in a string of characters. To embed quotes, type the apostrophe twice (') rather than use the double quotes ("), at both the beginning and the end of the string.

Note that constant values can be used in address expressions and as values in immediate mode addressing. They can also be used to initialize locations, as explained later in a section on assembler directives.



Symbols

Symbols may be used to refer to addresses or to data values. Symbols must begin with an alphabetic character, and must be no more than six characters in length. The letters A, S, P, X, and Y may not be used, as previously explained.

## ADDRESSING MODES

Symbolic

Perhaps the most common operand addressing mode is the symbolic form, as in

```
LDA BETA PUT BETA VALUE IN THE ACCUMULATOR
```

In the example, BETA refers to a byte in memory that is to be loaded into the accumulator. BETA is an address at which a value is located. Similarly, in the instruction

```
LDA ALPHA+BETA
```

the address ALPHA+BETA is computed by the assembler, and the value at the computed address is loaded into the accumulator. Both ALPHA and BETA must have been previously defined.

Memory associated with the 6502 processor is segmented into pages of 256 bytes each. The first page, page zero, is treated differently by the assembler and by the processor, for optimization of memory storage space. Many of the 6502 instructions have alternate operation codes if the operand address is in page zero memory. In such cases, the address requires only one byte of storage rather than the normal two bytes. For example, if BETA is located at byte 4B in page zero memory, the code generated for the instruction

```
LDA BETA
```

is A5 4B

This is called "page zero addressing". If BETA is at 01 3C in page one memory, the code generated is

```
AD 3C 10
```

This is an example of "absolute addressing". Thus, to optimize storage space and execution time, a programmer should design with data areas in page zero memory whenever possible. Note that the assembler makes decisions on which form of the operation code to use based upon operand address computation.

### Absolute Mode

In absolute mode, a specific address is given from which data is to be fetched or to which a branch will be made. Any two-byte address may be used, in decimal, hexadecimal, binary, or octal, provided the appropriate prefix character is employed.

### Immediate Mode

It is often useful to be able to reference the address of a label as immediate mode data. The assembler recognizes the characters < and > for this purpose. For instance,

```
LDA  #< HERE
```

will load the accumulator with the low order eight bits of the address of the byte labeled HERE, and

```
LDA  #> HERE
```

will load the accumulator with the high order byte of the address of HERE.

Immediate mode addressing always generates two bytes of machine code, the opcode and the value to be used as operand. Note that constant values can be used in address expressions and as values in immediate mode addressing.

### Relative Mode

There are 8 conditional branch instructions available to the user. An example is

```
BEQ      START      IF EQUAL BRANCH TO START
```

which might typically follow a compare instruction. If the values compared are equal, a transfer to the instruction labelled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during execution. At the time the addition is made the program counter is pointing to the next instruction beyond the branch instruction. A branch address must be within 129 bytes forward or 125 bytes backward from the conditional branch instruction. An error will be flagged at assembly time if a branch target falls outside the bounds for relative addressing. Relative addressing is used only for branch instructions.

#### Implied Mode

Twenty-five instructions such as TAX (Transfer contents of Accumulator to Index X) require no operand and hence are single byte instructions. Thus, the operand addresses are implied by the operation code.

Four instructions (ASL,LSR,ROR, and ROL) are special, in that the accumulator, A, can be used as an operand. In this special case these four instructions are treated as implied mode addressing and only an operation code is generated.

#### Indexed Mode

Operands may be indexed with values in registers X and Y. Indexing is indicated by a comma and appropriate letter following the operand. For example

```
LDA BETA,Y
```

The value in register Y is added to BETA to form the address of the operand. Not all instructions can be indexed and on some,

indexing may be permitted with one register, but not the other. Refer to Table 2 for allowable addressing modes.

### Indexed Indirect Mode

In this mode the operand address is a location in page zero memory which contains the address to be used as an operand.

An example is:

LDA (BETA,X)

The parentheses around the operand indicate it is indirect mode. In the above example the value in index register X is added to BETA. That sum must reference a location in page zero memory. During execution the high order byte of the address is ignored, thus forcing a page zero address. The two bytes starting at that location in page zero memory are taken as the address of the operand. For purposes of illustration, assume the following:

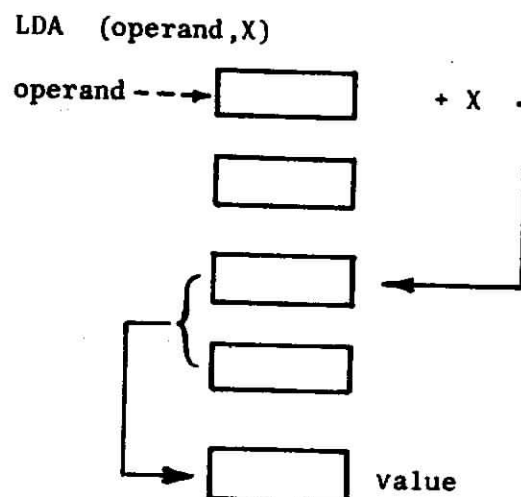
BETA is 12

X contains 4

Locations 0017 and 0016 are 01 and 25

Location 0125 contains 37

Then  $BETA + X$  is 16, the address at location 16 is 0125. The value at 0125 is 37 and hence the instruction LDA (BETA,X) loads the value 37 into the accumulator. This form of addressing is shown in the illustration below.



### Indirect Indexed Mode

Another mode of indirect addressing uses index register Y and is illustrated by:

LDA (GAMMA),Y

In this case GAMMA references a page zero location at which an address is to be found. The value in index Y is added to that address to compute the actual address of the operand. Suppose for example that:

GAMMA is 38 (hexadecimal)

Y contains 7

Locations 0039 and 0038 are 00 and 54

Location 005B contains 126

Then the address at 38 is 0054 and 7 is added to this, giving an effective address 005B. The value at 005B is 126 which is loaded into the accumulator.

In indexed indirect the index X is added to the operand prior to the indirection. In indirect indexed the indirection is done and then the index Y is added to compute the effective address. Indirect mode is always indexed except for a JMP instruction which allows an absolute indirect address as exemplified by JMP (DELTA) which causes a branch to the address at location DELTA. The indexed indirect mode of addressing is shown in the illustration below.

LDA (operand),Y

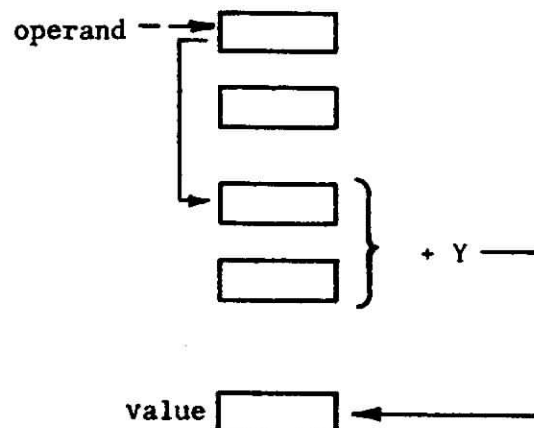


Table 2

Instruction Addressing Modes

	Immediate	Page zero	Absolute	Page zero indexed by X	Absolute indexed by X	Absolute indexed by Y	Indirect indexed
ACD	X	X	X	X	X	X	X
AND	X	X	X	X	X	X	X
ASL (1)		X	X	X	X		
BIT		X	X				
CMP	X	X	X	X	X	X	X
CPY	X	X	X				
CPX (2)	X	X	X				
DEC		X	X	X	X		
EOR	X	X	X	X	X	X	X
INC		X	X	X	X		
JMP (3)			X				X
JSR			X				
LDA	X	X	X	X	X	X	X
LDX (2)	X	X	X	X	X		
LDY	X	X	X	X	X		
LSR (1)		X	X	X	X		
ORA	X	X	X	X	X	X	X
ROL (1)		X	X	X	X		
ROR (1)		X	X	X	X		
SBC	X	X	X	X	X	X	X
STA		X	X	X	X	X	X
STX		X	X	X			
STY		X	X	X			

(1) Accumulator A can also be an operand

(2) Indexing with Y

(3) Indirect is absolute indirect and not indexed



## ASSEMBLER DIRECTIVES

There are five directives which are used to control the assembly process, define values or initialize memory locations. Assembler directives always appear in the opcode field of an instruction and thus might be considered as assembly time opcodes instead of execution time opcodes. The directives are: .BYTE, .WORD, .OPT, .END and equates (which is denoted by the equals sign =). All directives which are preceded by the period may be abbreviated to the period and three characters if desired (eg., .BYT).

.BYTE is used to reserve one byte of memory and load it with a value. The directive may contain multiple operands which will store values in consecutive bytes. ASCII strings may also be generated by enclosing the string with quotes.

```
HERE      .BYTE  2
THERE     .BYTE  1, $F, @3, %101, 7
ASCII     .BYTE  'ABCDEFH'
```

Note that numbers may be represented in the most convenient form. In general, any valid MCS650X expression which can be resolved to eight bits may be used in this directive. If it is desired to include a quote in an ASCII string, this may be done by putting two quotes in the string;

```
.BYTE 'JIM''S CYCLE'
```

could be used to print:

```
JIM'S CYCLE
```

.WORD is used to reserve and load two bytes of data at a time. Any valid expression, except for ASCII strings, may be used in the operand field.

```
HERE     .WORD  2
THERE    .WORD  1, $FF03, @3
WHERE    .WORD  HERE, THERE
```

The most common use for .WORD is to generate addresses, as shown in the above example. "WHERE" stores the 16 bit addresses of "HERE" and "THERE". Addresses in the 6502 are fetched from memory in the order low-byte, high-byte, and therefore .WORD generates the values in this order. The hexadecimal portion of the second example above (\$FF03) would be stored as 03 FF.

= is the EQUATE directive, and is used to reserve memory locations, reset the program counter (\*), or assign a value to a symbol.

HERE **+1	reserve one byte
WHERE **+2	reserve two bytes
**\$200	set program counter
NB=8	assign value
MN=NB+%101	assign value

Note that expressions must not contain forward references or they will be flagged as an error. For example,

$$Q=C+D-E*F$$

is legal only if C, D, E, and F are all defined. It is illegal if any of the variables is a forward reference, to be defined later in the program. Forward references in expressions may be used in the modified two-pass version of the ARESCO Assembler, but not in the one-pass version.

Note also that expressions are evaluated in strict left to right order.

.OPT is the most powerful directive available, and is used to control generation of output fields, listings, and expansion of ASCII strings in .BYTE directives. The options available are:

.OPT	ERRORS, LIST, SYMBOLS, GENERATE, TAB
.OPT	NOERRORS, NOSYMBOLS, NOLIST, NOGENERATE, NOTAB

The operand fields in the .OPT directive are only scanned for the first three characters. The individual .OPT operands are

1. SYM is used to control the printing of the symbol table at the end of the listing. The symbol table is not sorted. If NOSYM is selected, the symbol table is not printed.
2. ERR is used to control creation of error listing. To view only the errors on the assembly, use

```
.OPT ERR,NOLIST
```

When all the errors have been corrected, make another run using

```
.OPT NOERR,LIST
```

3. LIST is used to control the generation of the listing which contains source input, errors & warnings, and and code generated. NOLIST suppresses the listing.
4. GEN is used to control printing of ASCII strings in the .BYTE directive. The first two characters only are printed if NOGEN is used. Further characters (normally two bytes per line) are printed if GEN is used.
5. TAB is used to control automatic spacing of labels, opcodes, and comments. Use of this option will save memory space in storing the source code, since only a single space is needed between labels, opcodes, and comments. The assembler output will be neatly formatted. With narrow-carriage terminals (for example, 32 characters/line CRTs), the spacing may be objectionable. Use of NOTAB suppresses the spacing.

Default settings for the .OPT directive are

```
.OPT      SYM, LIST, ERR, TAB
```

.END must be the last statement in a program and is used to signal the physical end of the text file. If no .END is used, the assembler returns you to the monitor. This is useful when doing multiple-file assembly. In this case, only the last file assembled must contain the .END directive.

## ERROR MESSAGES

Error #1                    Not Used

Error #2                    FATAL - Label Previously Defined

The first field on the line is not an opcode, so it is interpreted as a label. If the current line is the first line in which that symbol appears as a label (or on the left side of an equals sign), it is put in the symbol table and tagged as defined in that line. However, if the symbol has appeared as a label, or on the left side of an equate, prior to the current line, the assembler finds the label already in the symbol table. The assembler does not allow redefinitions of symbols and will, in this case, print the error message.

Error #3                    FATAL - Illegal Or Missing Opcode

The assembler searches a line until it finds the first non-blank character string. If this string is not one of the 55 valid opcodes, it assumes the string is a label and places it in the symbol table. It then continues parsing for the next non-blank string. If none is found, the next line will be read in and the assembly will continue. However, if a second field is found, it is assumed to be an opcode (since only one label is allowed per line). If this character string is not a valid opcode, the error message is printed. This error can occur if opcodes are misspelled, in which case, the assembler will interpret the opcode as a label (if no label appears on the line). It will then try to assemble the next field as the opcode. If there is another field, this error will be printed. Check for more than one label on a line or a misspelled opcode.

**Error #4**                    FATAL - Address Not Valid

An address referred to in an instruction or in one of the assembler directives (.BYTE and .WORD) is invalid. In the case of an instruction, the operand that is generated by the assembler must be greater than or equal to zero and less than or equal to  $FFFF_{16}$  (2 bytes long). This excludes relative branches which are limited to  $\pm 127$  bytes from the next instruction. If the operand generates more than two bytes of code, or is less than zero, this error message will be printed. For a .BYTE, each operand is limited to one byte, and for a .WORD, each operand is limited to two bytes. All operands must be greater than or equal to zero. This validity is checked after the operand is evaluated. Check for values of symbols used in the operand field (see the symbol table for this information).

**Error #5**                    FATAL - Accumulator Mode Not Allowed

Following a legal opcode and one or more spaces is the letter A, followed by one or more spaces. The assembler is trying to use the accumulator (A means "accumulator mode") as the operand. However, the opcode in the statement is one which does not allow reference to the accumulator. Check for a statement labelled A (an illegal label) to which this statement is referring. If you were trying to reference the accumulator, look up the valid operands for the opcode used.

**Error #6**                    FATAL - Forward Reference In .BYTE or .WORD**Error #7**                    FATAL - Ran Off End Of Line

This error message will occur if the assembler is looking for a needed field and runs off the end of the line before the field is found. The following should be checked for:

1. A valid opcode field without an operand field on the same line.

2. An opcode that was thought to take an implied operand which in fact required an operand.
3. An ASCII string that is missing the closing quote (be sure any embedded quotes are doubled - to have a quote at the end, there must be three quotes: 2 for the embedded quote and 1 to close off the string).
4. A comma at the end of the operand field indicates that there are more operands to come. If there aren't any other operands, the assembler will run off the line looking for them.

## Error #8

FATAL - Label Doesn't Begin With An Alphabetic Character.

The first non-blank field is not a valid opcode. Therefore the assembler tried to interpret it as a label. However, the first character of the field does not begin with an alphabetic character and the error message is printed. Check for an unlabelled statement with only an operand field that does start with a special character. Also check for illegal labels.

## Error #9

FATAL - Label Greater Than Six Characters

All symbols are limited to six characters in length. When parsing, the assembler looks for one of the separating characters to find the end of a label or string. If other than one of these separators is used, the error message will be printed - providing the illegal separator causes the symbol to extend beyond six characters in length. Check for no spacing between labels and opcodes. Also check for a comment line with a long first word that doesn't begin with a semi-colon. In this case, the assembler is trying to interpret part of the comment as a label.



## Error #10

FATAL - Label Or Opcode Contains Non-Alphanumeric Character

Labels are made up of from one to six alphanumeric digits. The label field must be separated from the opcode field by one or more blanks. If a special character or other separator is between the label and the opcode, this error message might be printed.

The 55 valid opcodes are each three alphabetic characters. They must be separated from the operand field (if one is necessary) by one or more blanks. If the opcode ends with a special character (such as a comma), this error message will be printed.

In the case of a lone label or an opcode that needs no operand, they can be followed directly by a semicolon to denote the rest of the line as a comment.

## Error #11

FATAL - Forward Reference In Equate Or Org

The expression on the right side of an equals sign contains a symbol that hasn't been defined previously. One of the operations of the assembler is to evaluate expressions or labels and assign addresses or values to them. The assembler processes the input values sequentially which means that all of the symbolic values that are encountered fall into two classes--already defined values and not previously encountered values. The assembler assigns defined values and builds a table of undefined values. When a previously used value is discovered, it is substituted into the table. A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value.

To allow for conformity of evaluating expressions, this assembler allows for one level of forward reference so that the following code is allowed:

## Error #11 (Continued)

<u>Line Number</u>	<u>Label</u>	<u>Opcode</u>	<u>Operand</u>
100		BNE	New One
200	New One	LDA	#5

but the following is not allowed:

<u>Line Number</u>	<u>Label</u>	<u>Opcode</u>	<u>Operand</u>
100		BNE	New One
200	New One		Next + 5
300	Next	LDA	#5

This feature should not disturb the normal use of labels.

The cure for this error

<u>Line Number</u>	<u>Label</u>	<u>Opcode</u>	<u>Operand</u>
100		BNE	New One
300	Next	LDA	#5
301	New One		Next + 5

is very simple and always solves the problem.

This error may also mean that the value on the right side of the = is not defined at all in the program in which case the cure is the same as for undefined values.

Due to the sequential processing of the assembler and the dependency of the value of the program counter on symbols, throughout the rest of the program, the assembler cannot process a forward reference in this type of statement. All expressions with symbols that appear on the right side of any equals sign must refer only to previously defined symbols for the equate to be processed.

## Error #12

FATAL - Invalid Index. Index Must Be X Or Y

After finding a valid opcode, the assembler looks for the operand. In this case, the first character in the operand field is a left paren. The assembler interprets the next field as an indirect address which, with the exception of the jump statement, must be indexed by one of the index registers, X or Y. In the erroneous case, the character the assembler was trying to interpret as an index register

Error #12 (Continued) was not X or Y and the error was printed. Check for the operand field starting with a left paren. If it is supposed to be an indirect operand, recheck the format for the two types available. If the format was wrong (missing right paren or index register), this error will be printed. Also check for missing or wrong index registers in an indexed operand (form: expression, index register).

Error #13 FATAL - Invalid Expression In Operand

In evaluating an expression, the assembler found a character it couldn't interpret as being part of a valid expression. This can happen if the field following an opcode contains special characters not valid within expressions (e.g. parentheses). Check the operand field and make sure only valid special characters are within a field (between commas).

Error #14 FATAL - Undefined Assembler Directive

All assembler directives begin with a period. If a period is the first character in a non-blank field the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed. Check for a misspelled directive, or a period at the beginning of a field that is not a directive.

Error #15 FATAL - Invalid Operand For Page Zero Mode

Error #16 FATAL - Invalid Operand For Absolute Mode

Error #17

FATAL - Relative Branch Out Of Range

All of the branch instructions (excluding the two jumps) are assembled into 2 bytes of code. One byte is for the opcode and the other for the address to branch to. To allow a forward or backward branch, this branch is taken relative to the beginning of the next instruction, according to the address byte. If the value of the byte is 0-127 the branch is forward; if the value is 128-255 the branch is backward. (A negative branch is in 2's complement form). Therefore, a branch instruction can only branch forward or backward 127 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, the error message will be printed.

Error #18

FATAL - Illegal Operand Type For This Instruction

After finding an opcode that does not have an implied operand, the assembler parses the operand field (the next non-blank field following the opcode) and determines what type of operand it is (indexed, absolute, etc.). If the type of operand found is not valid for the opcode, this error message will be printed. Check to see what types of operands are allowed for the opcode and make sure the form of the operand type is correct (see the section on addressing modes).

Error #19

FATAL - Out Of Bounds On Indirect Addressing

An indirect address is recognized by the assembler by the parentheses that surround it. If the field following an opcode has parens around it, the assembler will try to assemble

Error #19 (continued)      it as an indirect address. Since indirects work only in page zero memory, if the address in the operand field is larger than 256 (one byte), this error message will be printed.

This error will only occur if the operand field is in correct form (i.e. an index register following the address), and the address field is out of page zero. To correct this, the address field must refer to page zero memory.

Error #20                    FATAL - A, S, P, X, and Y Are Reserved Labels

A label on a statement is one of the five reserved names (A, X, Y, S AND P). They have special meaning to the assembler and therefore cannot be used as labels. Use of one of these names will cause the above error message to be printed and no code to be generated for the statement. The label does not get defined and will appear in the symbol table as an undefined variable. Reference to such a label elsewhere in the program will cause error messages to be printed as if the label were never declared.

Error #21                    FATAL - Program Counter Negative! Reset To 0

An assembled program is loaded into core from position 0 to 64K (65536). This is the extent of the machine. Instructions can only refer to up to 2 bytes of information.

Because there is not such a thing as negative memory, an attempt to reference a negative position will cause this error and the program counter (or pointer to the current memory location) will be reset to 0.

When this error occurs, the assembler continues assembling the code with the new value of the program counter. This

Error #20 (continued) could cause multiple bytes to be assembled into the same locations. Therefore, care is to be taken to keep the program counter within the proper limits.

Error #22                    FATAL - Invalid Character. Expecting "="  
                             For Org

Other error messages are mnemonic, such as BAD COM, for Bad Command, in the Editor. They are self-explanatory, and hence are not discussed herein.

## APPENDIX A

Modifying The ARESCO Assembler For Two Pass Operation

Normal operation of the ARESCO Assembler/Text Editor allows the assembler portion of the package to assemble the source file into object code during a single reading of the file. The assembly is therefore fairly fast. The original version of the ARESCO Assembler was designed for the KIM-1, and it was an advantage to be able to assemble lengthy programs read from punched paper tape, without having to read the paper tape source code twice. In the version of the assembler designed for the APPLE II, this is no longer a benefit.

In order to assemble in a single pass, the assembler must build a symbol table and generate object code simultaneously. This works fine, except when the assembler is required to resolve forward references. (A forward reference is the use of an operand not previously defined, with the intention of defining the operand later in the program.) Consider this example:

```
10      LDY #04
20      CMP $43
30      BNE DONE
40      CLC
50 DONE JMP $1C00
60 .END
```

When the assembler encounters the label DONE in line 30, it cannot calculate the relative branch offset, because it does not yet know the address of the label DONE. It handles this situation by reserving two bytes for the branch offset, marking the symbol table entry for DONE as an undefined reference. The notation "\*\*\*" is printed in the listing of the assembled code. When DONE is defined in line 50, the assembler makes the cor-



rect entry in the symbol table, goes back and calculates the correct branch values for previous references to DONE, and inserts these values into the object code. The second byte reserved for the branch offset is set to EA (a NOP instruction).

If you assemble our example program, you will see that the branch address is listed as "\*\*\*". If you examine the object code generated by the assembler, you will see that the correct value for the relative branch was automatically inserted into the object code, and is followed by an EA (NOP) instruction.

For most quick assemblies, this single-pass operation is simple and quite sufficient. For lengthy programs, however, a two-pass assembler may be needed, and it is easy to modify the ARESCO Assembler to provide the two-pass operation.

#### Modifications To The Assembler

Modify the following locations using the APPLE monitor:

```
257A: 4C F0 30
30F0: B1 52 A0 03 29 1F C9 10
30F8: D0 01 88 A9 01 4C 7D 25
3782: 20 11 37 4C 11 20
```

and save this modified version on disk or tape (2000.3800W).

#### Using The Modified (Two-Pass) Assembler

To use this new two-pass assembler, follow these simple steps:

1. Set the symbol pointers (1FDF-1FE2), as usual.
2. Enter the editor at 3743, as usual.
3. Input and correct the source code, as usual.
4. Assemble the source code, using the A command - and ignore any resulting error messages.
5. When the assembler exits to the APPLE monitor, restart the assembler by entering 3782G (3782 is the assembler "warm start" address). This will produce a correct listing and a proper object code. There will be no asterisks in the listing, and forward branches will be a single byte.

Note that when the one-pass assembler is in operation, the program counter is set to \$200 if no "\*" statement is found. When using this modified, two-pass version of the assembler, the first line of the source code MUST be a definition of the program counter, such as

```
5 *=$200
```

The two-pass assembler will not work correctly without such a statement as the first line of code in the program.

#### Additional Modifications To The ARESCO Assembler/Text Editor

The ARESCO Assembler/Editor was originally designed to run on a KIM/TIM 6502 microcomputer. When the package was redesigned for the APPLE, massive page zero conflicts occurred. This problem was circumvented by keeping two copies of page zero in memory, one for use by the assembler/editor and one for use by the monitor I/O and disk routines. The assembler's page zero is stored in page 1F while the I/O and disk routines are being used, and the monitor's page zero is stored in page 1E while the assembler/editor is being operated. Locations 0300 through 0306 are used as temporary storage locations to preserve the values of A, X, Y, and P during I/O calls from the assembler/editor.

The MONASM routine located at #3711 saves the monitor's page zero and loads the assembler's page zero. The ASMMON routine at location \$36DF saves the assembler's page zero and loads the monitor's page zero. All assembler/editor I/O routines begin with a call to ASMMON and end with a call to MONASM. Only the contents of register A are passed between the monitor and the assembler.

The following I/O routines are included in the assembler and may be disassembled, examined, and modified by the knowledgeable user:

Assembler/Editor cold start	3743
Assembler warm start	3782
Input a character	36BD
Output a character	36D3
Carriage return	36D1
Bell	376C
Exit to APPLE monitor	377C

The last memory location actually used by the ARESCO Assembler/Text Editor is \$37A2. Within the assembler/editor, all character data is in true ASCII code, not the quasi-ASCII normally used for APPLE I/O.

#### Source Text Format

Source Text is stored in APPLE memory in standard ASCII. Each line of text stored in memory begins with the two byte line number in BCD, and is stored in high-order, low-order byte sequence; not as low, high. Each line is terminated with a 0D (carriage return). The end-of-file mark is 1F following the final 0D.

#### For More Information

A complete, commented listing of the original ARESCO Assembler/Text Editor for 6502 based microsystems is available from ARESCO for \$20.00. The original version does not contain some of the minor patches which have been added to the package over the past three years.

ARESCO also publishes the Rainbow, a newsletter devoted exclusively to APPLE II owners. See the last page of this document for subscription information.

APPENDIX B  
Sample Runs

A disassembled listing of a subroutine to "click" the APPLE's speakers was published in the ARESCO Rainbow in January, 1979. (See Basic Music & Sound Effects, Issue 1, Volume 1, page 17.)

A greater segment of the program from which that subroutine was taken is used here to illustrate the operation of the ARESCO Assembler/Text Editor Package For The APPLE II.

First, the Editor portion of the package is entered and the program is listed. The assembler is called, and the program is listed during assembly. The asterisks in lines 110, 120, and 130 indicate the space reserved for forward references. In the later disassembly, the addresses referenced are present, followed by an EA (NOP) instruction.

After the modifications for the two-pass version of the assembler have been made, the editor is again entered, and the assembler is called. The asterisks are still present, and spurious errors have been generated.

The errors are ignored, and the assembler is entered at \$3782 for the second pass. Now the forward references are properly calculated and the asterisks replaced. Again, the resulting machine code is shown, then disassembled using the APPLE's resident disassembler. Notice the absence of the EA instruction in the branch instructions in the two-pass assembler version.

Sample Editor Run

\*3743G Start the Editor  
 BPSE=4000 Text starts at \$4000  
 N OR 0? Specify old file  
 4000 4147 0018 Status  
 P.B Print out the file

```

0010 ;SUBROUTINE TO GENERATE TONES
0020 *=4000
0030 PITCH =++1
0040 LONGL =++1
0050 CLICK = $C030 ; CLICK SPEAKER
0060 WAIT = $FC08 ; DELAY SUB.
0070 ENTRY LDA CLICK ; MAKE CLICK
0080 LDA PITCH
0090 JSR WAIT
0100 LDA LONGL
0110 BNE LOOP
0120 DEC LONGL
0130 BEQ END
0140 LOOP DEC LONGL
0150 JMP ENTRY
0160 END RTS
0170 LONGL . BYTE 00 ; DURATION HI BYTE
0180 .END
*ET

```

A

Ask for assembly  
Single-pass version

BRESO

LINE # LOC CODE LINE

```

0010 0200      ;SUBROUTINE TO GENERATE TONES
0020 0200      #=0000      Set origin
0030 0000      PITCH #=#+1      Reserve space
0040 0001      LONGL #=#+1
0050 0002      CLICK = $C030      ; CLICK SPEAKER
0060 0002      WAIT = $FC0B      ; DELAY SUB.      Define subroutines
0070 0002 AD 30 C0 ENTRY LDA CLICK      ; MAKE CLICK
0080 0005 AD 00 03      LDA PITCH
0090 0008 20 0B FC      JSR WAIT
0100 000B AD 01 03      LDA LONGL
0110 000E D0 ** **      BNE LOOP
0120 0011 CE ** **      DEC LONGH      Forward references
0130 0014 F0 ** **      BEQ END
0140 0017 CE 01 03 LOOP DEC LONGL
0150 001A 4C 02 03      JMP ENTRY
0160 001D 00      END RTS
0170 001E 00      LONGH .BYTE 00      ; DURATION HI BYTE
0180 001F      .END

```

ERRORS = 0000

SYMBOL TABLE

```

PITCH 0000  LONGL 0001  CLICK 0030  WAIT  FC0B
ENTRY 0002  LOOP  0017  LONGH 001E  END   001D

```

END OF ASSEMBLY



\*000, 81F

Show results

```

0000- 00 00 AD 30 C0 AD 00 00
0005- 20 A8 FC AD 01 08 D0 07
0010- EA CE 1E 08 F0 07 EA CE
0015- 01 08 4C 02 08 60 00 A8

```

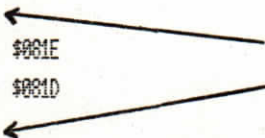
\*000, 81FLDisassemble the code  
with the APPLE monitor

```

0000- 00      BRK
0001- 00      BRK
0002- AD 30 C0 LDA  $C030
0005- AD 00 00 LDA  $0000
0008- 20 A8 FC JSR  $FA08
000B- AD 01 08 LDA  $0001
000E- D0 07    BNE  $0017
0010- EA      NOP
0011- CE 1E 08 DEC  $001E
0014- F0 07    BEQ  $001D
0016- EA      NOP
0017- CE 01 08 DEC  $0001
001A- 4C 02 08 JMP  $0002
001D- 60      RTS
001E- 00      BRK

```

Generated by forward references  
in branch instructions





Sample Run Using The Two-Pass Modification To The Assembler

\*37436

Enter The Editor

BASE=4000

N OR 0?0

4000 4147 0018

A

Assemble pass one

ARESCO

LINE #	LOC	CODE	LINE	
0010	0200			; SUBROUTINE TO GENERATE TONES
0020	0200			*=\$000
0030	0000			PITCH **++1
0040	0001			LONGL **++1
0050	0002			CLICK = \$C030 ; CLICK SPEAKER
0060	0002			WAIT = \$FC08 ; DELAY SUB.
0070	0002	AD 30 C0	ENTRY	LDA CLICK ; MAKE CLICK
0080	0005	AD 00 08		LDA PITCH
0090	0008	20 A8 FC		JSR WAIT
0100	0008	AD 01 08		LDA LONGL
0110	000E	** **		BNE LOOP
0120	0010	CE ** **		DEC LONGL
0130	0013	** **		BEQ END
				***ERROR # 04 PC = CEFF
0140	0015	CE 01 08	LOOP	DEC LONGL
0150	0018	4C 02 08		JMP ENTRY
				***ERROR # 04 PC = CEFF
0160	001B	00	END	RTS
0170	001C	00	LONGH	.BYTE 00 ; DURATION HI BYTE
0180	001D			.END

Ignore the errors

ERRORS = 0002

## SYMBOL TABLE

PITCH	0000	LONGL	0001	CLICK	0030	WAIT	FC00
ENTRY	0002	LOOP	0015	LONGH	001C	END	001B

END OF ASSEMBLY

\*3782G

Start pass two

PRESCO

LINE #	LOC	CODE	LINE
0010	001D		; SUBROUTINE TO GENERATE TONES
0020	001D		*=\$000
0030	0000	PITCH	*++1
0040	0001	LONGL	*++1
0050	0002	CLICK	=\$0030 ; CLICK SPEAKER
0060	0002	WAIT	=\$FC00 ; DELAY SUB.
0070	0002	AD 30 C0	ENTRY LDA CLICK ; MAKE CLICK
0080	0005	AD 00 00	LDA PITCH
0090	0005	20 A0 FC	JSR WAIT
0100	0006	AD 01 00	LDA LONGL
0110	000E	D0 05	BNE LOOP
0120	0010	CE 10 00	DEC LONGH
0130	0013	F0 06	BEQ END
0140	0015	CE 01 00	LOOP DEC LONGL
0150	0018	4C 02 00	JMP ENTRY
0160	001B	00	END RTS
0170	001C	00	LONGH .BYTE 00 ; DURATION HI BYTE
0180	001D		.END

Forward references OK

ERRORS = 0002

Note that error count is  
not reset after pass one

## SYMBOL TABLE

PITCH	0000	LONG1	0001	CLICK	0030	WRIT	FC08
ENTRY	0002	LOOP	0015	LONGH	001C	END	001B

END OF ASSEMBLY

\*000 81E

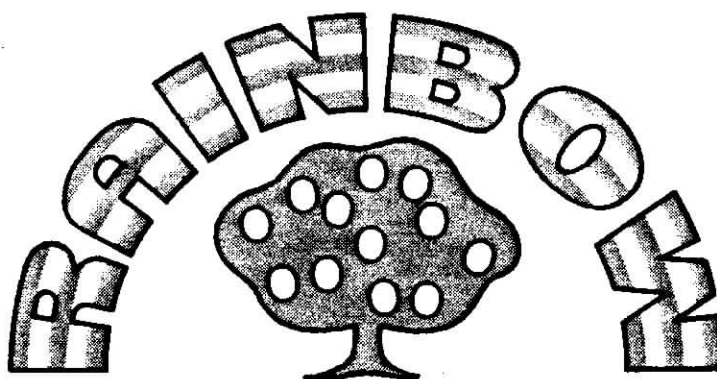
Show results

```
0000- 00 00 AD 30 C0 AD 00 00
0000- 20 A8 FC AD 01 03 D0 05
0010- CE 1C 08 F0 06 CE 01 00
0010- 4C 02 00 60 00 60 00
```

\*000 LDisassemble the code  
with the APPLE monitor

```
0000- 00      BRK
0001- 00      BRK
0002- AD 30 C0 LDA $C030
0005- AD 00 00 LDA $0000
0008- 20 A8 FC JSR $FC08
000B- AD 01 00 LDA $0001
000E- D0 05    BNE $0015
0010- CE 1C 08 DEC $001C
0013- F0 06    BEQ $001B
0015- CE 01 00 DEC $0001
0018- 4C 02 00 JMP $0002
001B- 60      RTS
001C- 00      BRK
```

Note that error count is  
not reset after pass one



**BOX 43, AUDUBON, PA 19407**

(215) 631-9052

The Rainbow is an independent, national newsletter dedicated to APPLE II owners. It's published ten times a year (every month except July and December) and a subscription includes all ten issues of the current volume.

The Rainbow is a user newsletter dedicated to the non-professional computerist. It's the only newsletter to acknowledge the fact that people have to start somewhere before they get to be "experts".

The Rainbow is intended to be an information exchange - you can learn about the projects for which other people use their APPLES - and share your own experiences and discoveries. Find other APPLE owners in your area - or communicate with APPLE owners in other countries. The whole world of APPLE users is available to you through the pages of the Rainbow!

Send us your name, address, and a check or money order for \$15.00 (the price includes all ten issues of the current volume). By return mail, you'll receive your copy of the Rainbow Questionnaire to fill out and return, so we'll know what you want to see in the newsletter. If you live outside the USA, and don't want to wait the three-to-four weeks required by surface (international) mail, include \$10.00 for airmail postage. Please send us a street address - if we're shipping back issues, we ship via UPS - or an additional \$0.40 per copy for domestic postage. No invoicing. Sorry.

.....  
 Please ship me all the back issues of this volume, and enter my subscription for the current volume of The Rainbow. I enclose payment in full, including postage if required.

NAME \_\_\_\_\_

ADDRESS \_\_\_\_\_

CITY, STATE, ZIP \_\_\_\_\_

CHECK \_\_\_\_\_/MC/VISA CARD # \_\_\_\_\_

If using Master Charge, we need the other 4 digits on the card

\_\_\_\_\_ EXP DATE \_\_\_\_\_ SIGN \_\_\_\_\_

.....  
 Mail to: THE RAINBOW \* P.O. BOX 43 \* AUDUBON \* PA \* 19407



BOX 447, ALBUQUERQUE, N.M.

The following information is being furnished to you for your information and is not to be used for any other purpose without the express written consent of the National Association of the Deaf.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.

The National Association of the Deaf is a non-profit organization which is organized for the purpose of promoting the welfare of the deaf and to provide for their education, training, and employment.





