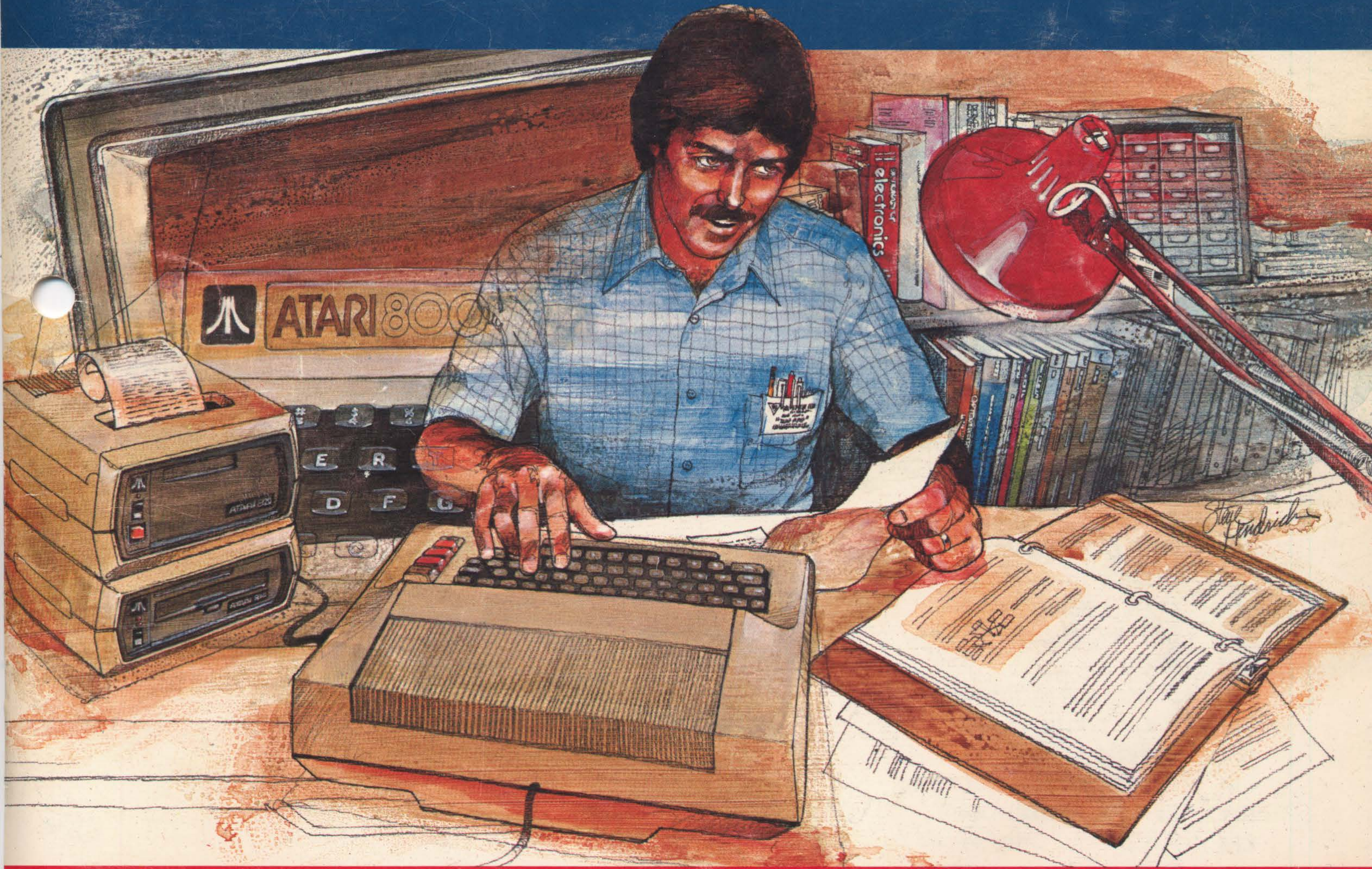
ATARI® 400/800™

ASSEMBLER EDITOR

USER'S MANUAL



ATARI®

A Warner Communications Company

Model CXL4003
Use with
ATARI® 400™ or ATARI 800™
PERSONAL COMPUTER SYSTEMS

ERROR CODES

ERROR CODE	ERROR CODE MESSAGE
2	Memory insufficient
3	Value error
4	Too many variables
5	String length error
6	Out of data error
7	Number greater than 32767
8	Input statement error
9	Array or string DIM error
10	Argument stack overflow
11	Floating point overflow/ underflow error
12	Line not found
13	No matching FOR statement
14	line too long error
15	GOSUB or FOR line deleted
16	RETURN error
17	Garbage error
18	Invalid string character

Note: The following are INPUT/OUTPUT errors that result during the use of disk drives, printers, or other accessory devices. Further information is provided with the auxiliary hardware.

ERROR CODE	ERROR CODE MESSAGE
19	LOAD program too long
20	Device number larger
21	LOAD file error
128	BREAK abort
129	IOCB
130	Nonexistent device
131	IOCB write only
132	Invalid command
133	Device or file not open
134	Bad IOCB number
135	IOCB read only error
136	EOF
137	Truncated record
138	Device timeout
139	Device NAK
140	Serial bus
141	Cursor out of range
142	Serial bus data frame overrun
143	Serial bus data frame checksum error
144	Device done error
145	Read after write compare error
146	Function not implemented
147	Insufficient RAM
160	Drive number error
161	Too many OPEN files
162	Disk full
163	Unrecoverable system data I/O error
164	File number mismatch
165	File name error
166	POINT data length error
167	File locked
168	Command invalid
169	Directory full
170	File not found
171	POINT invalid

For explanation of Error Messages see Appendix 1.

ASSEMBLER EDITOR MANUAL

ATARI®

A Warner Communications Company

PRINTED IN U.S.A.

PREFACE

This manual assumes the user has read an introductory book on assembly language. It is not intended to teach assembly language. Suggested references for assembly language beginners are *6502 Assembly Language Programming* by Lance Leventhal and *Programming the 6502* by Rodney Zaks (see Appendix 8).

The user should also know how to use the screen editing and control features of the ATARI® 400™ and ATARI 800™ Personal Computer Systems. These features are the same as used in ATARI BASIC. Review the ATARI BASIC Reference Manual if you are unsure of how to do screen editing.

This manual starts by showing the structure of statements in assembly language. The manual then illustrates the different types of 6502 operands. The Assembler Editor cartridge contains three separate programs:

- EDIT (Editor program) — Helps you put programming statements in a form the Assembler (ASM) program understands. The EDIT program lets you use a printer to print a listing of your program. Programs can also be stored and recalled using ENTER, LIST and SAVE, LOAD. The Assembler Editor allows automatic numbering, renumbering, delete, find and replace.

- ASM (Assembler program) — Takes the program statements you create in the EDIT step and converts to machine code.

- DEBUGGER — Helps you trace through the program steps by running the program a step at a time while displaying the contents of important internal 6502 registers. The DEBUGGER program also contains programming routines which allow you to display registers, change register contents, display memory, change memory contents, move memory, verify memory, list memory with disassembly, assemble one instruction into memory, go (execute program), exit. The disassembly routine is especially useful in reading and understanding machine language code.

The Assembler Editor cartridge allows you to talk in the computer's natural language — machine language. Assembly language programming offers you faster running programs and the ability to tailor programs to your exact needs.

CONTENTS

ILLUSTRATIONS

1

INTRODUCTION

To use the **ATARI® Assembler Editor** cartridge effectively, there are four kinds of information that you must have. First, you need some guidance about how to use the cartridge itself. Second, you need to know about the ATARI Personal Computer System you are using with the cartridge. Third, you need to know something about 6502 Assembly Language programming. And, fourth, the Assembler Editor Cartridge was designed to be used with the ATARI disk drives and DOS II.

ABOUT THIS MANUAL

This manual explains the operation of the ATARI Assembler Editor cartridge. It does not explain 6502 Assembly Language programming. If you are already familiar with 6502 Assembly Language, you will find this manual amply suited to your needs; otherwise, you should refer to one of the many books that explain 6502 Assembly Language programming; suitable books are listed in Appendix 8.

If you are familiar with ATARI BASIC and have written some programs on your ATARI 400™ or ATARI 800™ Personal Computer System, you will find no better way to learn assembly language than the combination of this manual, the ATARI Assembler Editor cartridge, and a 6502 programming book.

If you have had no experience with computers and no programming experience, then this manual is probably too advanced for you and you should start by writing some programs using ATARI BASIC and your ATARI Personal Computer System to become familiar with programming in general. Reading one of the books recommended in Appendix 8 will help you learn assembly language.

ATARI PERSONAL COMPUTER SYSTEMS

The ATARI Assembler Editor cartridge is installed in the cartridge slot of the ATARI 400 computer console and in the left cartridge slot of the ATARI 800 computer console. You must be familiar with the keyboard and all the screen-editing functions. That material is covered in the appropriate Operator's Manual supplied with your ATARI Personal Computer System. The special screen-editing keys are described in Section 6 of the Operator's Manual. You should read Section 6 and follow the instructions until you are completely familiar with the keyboard and the screen-editing functions.

You need not have any equipment except the ATARI Personal Computer System console, your television or a video monitor for display, and the ATARI Assembler Editor cartridge. However, without a permanent storage device you will have to enter your program on the keyboard each time you wish to use it. This can be tedious and time-consuming. An ATARI 410™ Program Recorder, ATARI 810™ Disk Drive, or ATARI 815™ Dual Disk Drive (double density) is a practical necessity.

The ATARI 410 Program Recorder is an accessory that functions with the ATARI 400 and the ATARI 800 Personal Computer Systems. The proper operation of your Program Recorder is explained in Section 8 of the ATARI 400 and ATARI 800 Operator's Manuals. Before using the Program Recorder with the Assembler Editor cartridge, be sure you know how to operate the Program Recorder. The disk drives are accessories that function with any ATARI Personal Computer System with at least 16K RAM. To use a disk drive you need a special program, the Disk Operating System (DOS). At least 16K of memory is required to accommodate DOS. Consequently, if you are using an ATARI 400 Personal Computer System, you must upgrade it from 8K to 16K (RAM). This can be done at any ATARI Service Center.

If you are using the ATARI 810 Disk Drive, you should refer to the instructions that come with it. You should also read the appropriate Disk Operating System Reference Manual. If you are currently using the 9/24/79 version of DOS (DOS I), you must use the program in Appendix 11 for the disk drive to be compatible with the Assembler Editor cartridge.

If you are using the ATARI 815 Dual Disk Drive, you should refer to the ATARI 815 Operator's Manual and the Disk Operating System II Reference Manual that come with it.

You can also add the ATARI 820™, the ATARI 825™ or the ATARI 822™ Printer to your Personal Computer System to give you "hard copy"—that is, a permanent record of your program written on paper.

HOW AN ASSEMBLER EDITOR IS USED

All assembly language programs are divided into two parts: a "source program," which is a human-readable version of the program, and the "object program," which is the computer-readable version of the program. These two versions of the program are distinct and must occupy different areas of RAM. As the programmer, you have three primary tasks:

- To enter your source program into the computer, edit it (make insertions, deletions, and corrections) and save it to or retrieve it from diskette or cassette.
- To translate your source code into object code.
- To monitor and debug the operation of your object program.

These three tasks are handled with three programs included in the ATARI Assembler Editor. The first program, called the Editor, provides many handy features for entering the program and making insertions, deletions, and corrections to it. It also allows you to save and retrieve your source code. The second program, called the Assembler, will translate your source program into an object program. While doing so, it will provide you with an "assembly listing," a useful listing in which your source program is lined up side by side with the resulting object program. The third program is called the Debugger; it helps you to monitor and debug your object program. The relationship between these three programs is depicted as follows:

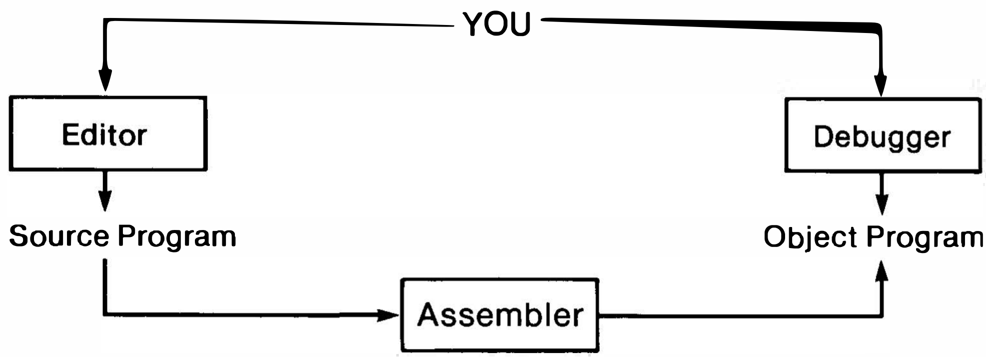


Figure 1. Relationship of various parts of Assembler Editor cartridge to you and your software.

In Section 3 we explain the Editor; in Section 4, the Assembler; and in Section 5, the Debugger. There are some fundamental ideas we must explain first.

NOTES:

2

GETTING STARTED

ALLOCATING MEMORY

The very first decision you must make when you sit down to write your source program involves the allocation of memory space.

All programs, regardless of language, occupy memory space. The computer has a limited amount of memory and must manage its memory carefully, allocating portions of memory for program, data, display space, and so forth. This is all done automatically in BASIC, so the BASIC user need not worry about where in memory his program and data are stored. Such is not quite the case with the Assembler Editor cartridge. You have the power to place your programs anywhere in memory that you desire. With this power comes the responsibility to allocate memory wisely.

The ATARI computer system uses low memory for its own internal needs. The amount it uses depends on whether or not DOS is loaded into RAM. In any event, the Assembler Editor cartridge will automatically place your source program into the chunk of memory starting with the first free memory location. As you type in more source code, the memory allocated to storing your source code (called the "Edit Text Buffer") grows. If you delete lines of source code, the edit text buffer shrinks. You can visualize the memory allocation with this figure, which is called a memory map:

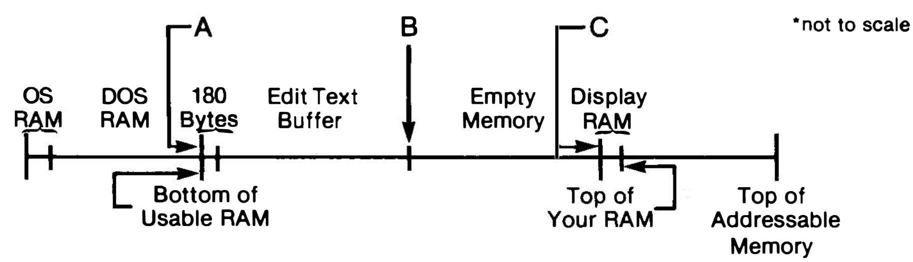


Figure 2. Memory map without use of LOMEN.

The edit text buffer always grows towards the right, into the "empty memory" area. The left side of the edit text buffer is fixed in place once you start entering code.

Your problem is to determine where to store the object code produced by the Assembler. If you put the object code into the regions marked OS RAM, DOS RAM, or display RAM, you will probably cause the computer to crash and all your typing will be lost. If you put it into the place called the edit text buffer, the object code will overwrite the source code, causing more chaos. The only safe place to put your object code is in the "empty memory" area.

You can find out where this empty memory area is by typing SIZE RETURN. Three hexadecimal numbers will be displayed, like so:

```
SIZE RETURN
0700  0880  5C1F
EDIT
```

The first number (0700 in this example) is the address of the bottom of usable RAM, the point labeled "A" on the memory map. The second number is the address of the top of the edit text buffer, labeled "B" on the memory map. The third number is the address of the top of empty memory, labeled "C" on the memory map. The difference between the second and third numbers (how good are you at hexadecimal subtraction?) is the amount of empty memory. You can use the SIZE command any time you desire to know how much empty memory remains.

Liberally estimate the amount of memory your object program will require, then subtract that amount from the third number. For extra insurance, round the result down. For example, if you thought that your object code might require 1.5K, you'd subtract 2K from $5C1F to get $541F and then for simplicity (and additional insurance) you would round all the way down to $5000. You would therefore store your object code at $5000, confident that it would not encroach on the display memory. More conservative estimates and greater care would be necessary if memory were in short supply.

Having decided to store the object program starting at address $5000, your next task is to declare this to the computer. This is done with *= directive. The very first statement of the source code would read:

```
10  *=$5000
```

This directive tells the Assembler to put all subsequent object code into memory starting at address $5000. Although it is not absolutely necessary, it is always wise practice to make the *= directive the very first line of your source program.

You have two other strategies for allocating memory space for your object program. The first and simplest strategy is to place your object code on page 6 of memory. The 256 locations on page 6 have been set aside for your use. If your object program and its data will all fit into 256 bytes, then you can put it there with the directive:

```
10  *=$0600
```

This is a good safe way to start when you are still learning assembly language programming and are writing only very short programs. As your programs grow larger, you will want to move them off page 6 and use page 6 for data and tables.

The second strategy is to bump the edit text buffer (your source program) upward in memory, leaving some empty memory space below it. You can then place your object code into this empty space. Figure 3 shows the adjustment of the memory map.

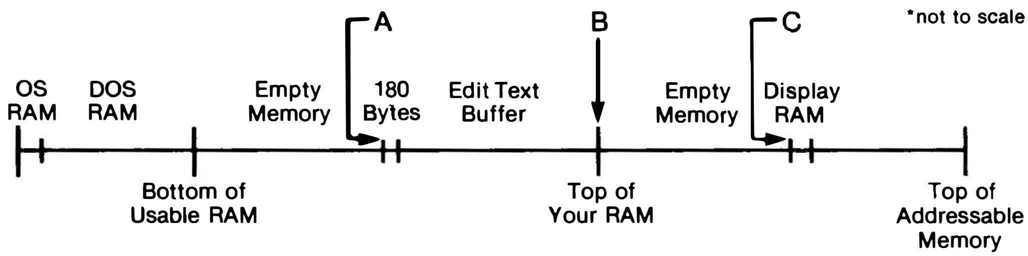


Figure 3. Memory map with use of LOMEM.

This bumping is accomplished with a special command called LOMEM. The command is special because it must be the very first command you enter after turning on the computer. Its form is simple:

LOMEM XXXX RETURN

where XXXX is the hexadecimal address of the new bottom edge of the edit text buffer (point A in the memory map). You must not set LOMEM to a smaller value than it normally is, or you will overwrite OS data or DOS and crash the system. Furthermore, if you set LOMEM too high, you will have too little room for your source program. You must estimate how much memory your object code will require, and bump the edit text buffer upward by that much plus some more for insurance. Then your first program instruction becomes:

```
10  *=$YYYY
```

where YYYY is the old value of A given by the SIZE command before you turned off the computer, turned it back on, and used the LOMEM command.

You might wonder why anybody would want to use the LOMEM command and store the object program in front of the source program instead of behind it. The primary reason this command is provided comes from the fact that the Assembler program, as it translates your source program into an object program, uses some additional memory (called a symbol table) just above the edit text buffer. If you really wanted to, you could figure just how much memory the symbol table uses; it is three bytes for each distinct label plus one byte for each character in each label. Most programmers who don't enjoy figuring out how big this symbol table is use the LOMEM command so they won't have to worry about it. (Only the label itself counts, not the number of times it appears in the program.)

Allocating memory can be a confusing task for the beginner. Only two instructions (LOMEM and *=) are used, but if they are misused you can crash the system and lose your work. Fortunately, if you restrict yourself to small programs initially you'll have plenty of empty memory space and fewer allocation problems.

The *= directive will be followed by your source program. The source program is composed of statements. The statements must be written according to a rigorous format. The rules for writing statements are given in the next section.

PROGRAM FORMAT—HOW TO WRITE A STATEMENT

A source program consists of statements. Each statement is terminated with RETURN. A statement may be 1-106 characters long, or almost three lines on the screen. A statement is also called a line. The distinction is made between a physical line (a line on the screen) and a logical line (the string of characters, up to three physical lines between RETURNs).

A statement can have up to five parts or "fields": the statement number, a label, the operation code mnemonic or directive, an operand, and a comment. These five fields occupy successive positions in the statement, with the statement number coming first and the comment coming last. Fields are separated ("delimited") by single spaces.

Statement Number

Every statement must start with a number from 0 to 65535. It is customary to number statements in increments of 10, 20, 30, etc. The Editor automatically puts the statements in numerical order for you. Numbering by tens allows you to insert new statements at a later date between existing statements. To assist you, the Editor has several convenient commands for automatically numbering statements (see NUM, REN).

Label

A label, if used, occupies the second field in the statement. You must leave exactly one space (not a tab) after the statement number. The label must start with a letter and contain only letters and numbers. It can be as short as one character and as long as the limitation of statement length permits (106 less the number of characters in the statement number). Most programmers use labels three to six characters long.

You are not forced to have a label. To go on to the next field, enter another space (or a tab). The Assembler will interpret the entries after a tab as an operation code mnemonic.

Operation Code Mnemonic

The operation code (or op code) mnemonic must be one of those given in Appendix 2. It must be entered in the field that starts at least two spaces after the statement number, or one space after a label. An operation code mnemonic in the wrong field will not be identified as an error in the Edit mode, but will be flagged when you assemble the program (Error 6).

Operand

The field of the operand starts at least one space (or a tab) after an operation code mnemonic. Some operation code mnemonics do not require an operand. The Assembler will expect an operand if the op code mnemonic requires one. Each different way of writing an operand is given in the section called HOW TO WRITE OPERANDS.

Comment

A comment appears on the listing of a program, but does not in any way affect the assembled object code. Programmers use comments to explain to others (and to themselves) how a section of code works.

There are two ways to have the Assembler interpret entries as comments. One way is to make the entries in the comment field, which occupies the remainder of the line after the instruction field(s). At least one space must separate the instruction fields from the comment field. There may not be enough space in the comment field for the comment you wish to write there. In that case it is best to use one or more lines as comment lines dedicated only to making comments and containing no code. To do so, you enter one space and a semicolon followed by any comment or explanatory markings you desire. Everything between the initial semicolon and the RETURN is ignored by the Assembler, but will be printed in the listing of the program.

A sample programming form for assembly language is reproduced as Figure 4. The form shows examples of how to enter line number, label, op code, operand and comments. These classes of entry are lined up vertically on the programming form. Most variation occurs in the method of entering a comment. Therefore, Figure 4 includes examples of the various ways to enter comments.

Sample, Reproducible
ATARI Programming Form

PROGRAM SAMPLE.ASM	PAGE 1 OF 1 — DATE 12/31/80
	PROGRAMMER JOHN DOE

LINE NO.	LABEL	OP CODE	OPERAND	COMMENT
10	LABL	LDX	ABS	COMMENT IN COMMENT FIELD
20		TXA		COMMENT IN COMMENT FIELD
30		PHA		COMMENT IN THIS LINE AUTOMA
				TICALLY CONTINUES ON THIS LINE
240		CLC		COMMENT ON THIS LINE CONT
250				;INUES ON THIS NUMBERED LINE
260		ADC	#$B2	
270				;COMMENT ON ITS OWN LINE
280	ANYTHINGMAKESALABEL			
290				;
300				;PREVIOUS LINE (290) CONTAINS
				ONLY BLANKS, (SPACE BAR). LINE
				280 CONTAINS ONLY THE LABEL
				"ANYTHINGMAKESALABEL"

Figure 4. Example of how to write Line No., Label, Op Code, Operand, and Label on the Atari programming form.

The spacing on the programming form is not the same as the spacing to be used on the screen, controlled by keyboard entry. On the screen the classes of entry (the fields) are not lined up vertically. The screen has 38 positions (you can change it to a maximum of 40), fewer than the programming form, and that is the main reason not to use many spaces between fields. Another difference between the programming form and screen is the 'wraparound' on the screen—automatic continuation of characters onto the next line.

Figure 5 shows the entries in Figure 4 as they should appear on the screen when entered on the keyboard with the recommended spacing. In general, the spacing recommended in this manual is the minimum spacing that will be correctly interpreted by the Assembler Editor. If you prefer to have more vertical alignment of fields, use TAB, rather than the single spacing between fields that we recommend. The statements below show various examples of comments correctly positioned in the statement. Each comment in the examples starts with "COMMENT" or semicolon(;).

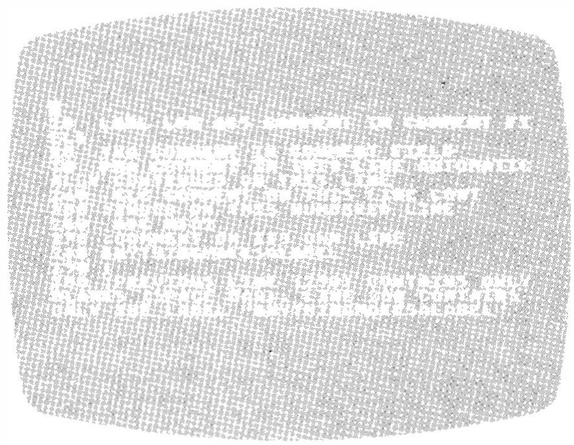


Figure 5. Statements as they would appear on the screen when entered on the keyboard with the recommended spacing. The various ways to enter comments are illustrated. Compare with Figure 4.

HOW TO WRITE OPERANDS

This section shows how to write operands. The examples use statement number XXXX (also called line number XXXX). An instruction entered without a statement number is not allowed by the Editor.

The examples use BY (for byte) and ABS (for absolute) as a one-byte and a two-byte number, respectively. This use implies that the program includes definitions of BY and ABS as, for example:

```
0100  BY = 155
0200  ABS = 567
```

Please refer to the description of the LABL = directive for an explanation of the definitions of lines 100 and 200.

Hexadecimal Operands

A number is interpreted as a decimal number unless it is preceded by $, in which case it is interpreted as a hexadecimal number.

Examples:

```
30  STA  $9325
80  ASL  $15
```

Immediate Operands

An immediate operand is an operand that contains the data of the instruction. The pound sign (#) must be present to indicate an immediate operand.

Examples:

```
  40  LDA  #12
  70  ORA  #$3C
1000  CPY  #BY
```

Page Zero Operands

When an operand is a number less than 255 decimal, (FF hex) and is not immediate, the number is interpreted as a page zero address.

Examples:

```
150  LDX  $12
250  ROR  33
500  DEC  BY
```

Absolute Operands

Absolute operands are evaluated as 16-bit numbers.

Examples:

```
  20  LDX  $1212
  40  CPY  2345
 990  DEC  579
2350  BIT  ABS
```

Absolute Indexed Operands

An absolute indexed operand uses register X or Y. The operand is written ________,X or ________,Y

Examples:

```
  10  AND  $3C26,X
 110  EOR  20955,Y
1110  STA  ABS,Y
```

Non-Indexed Indirect Operands

In general, an indirect operand is written with parentheses. The address within the parentheses is an intermediate address which itself contains the effective address. The only instruction with a non-indexed indirect operand is Jump Indirect. The operand is a number enclosed in parentheses. The parentheses in the operand enclose a number or an expression that is interpreted as an intermediate address.

Examples:

```
JMP  ($6000)
JMP  (ABS)
JMP  (7430)
JMP  (ABS+256*BY)
```

Indexed Indirect Operands

An indexed indirect instruction uses register X. The operand is written (—,X)

Examples:

```
10  INC  ($99,X)
```

Indirect Indexed Operands

An indirect indexed instruction uses register Y. The operand is written (—),Y

Examples:

```
  10  LDA  ($2B),Y
 110  CMP  ($E5),Y
1110  ORA  (BY),Y
```

Indexed Page Zero Operands

A zero page indexed operand is written —,X or —,Y

Examples:

```
  10  INC  $34,X
 110  STX  $AB,Y
1110  LDX  BY,Y
```

String Operands

Operands or parts of operands enclosed in double quotation marks are translated into the ATASCII codes of the characters between the quotation marks. The use of such operands must of course be appropriate to the type of instruction or directive to which they are appended.

Examples:

```
10  ADDR  .BYTE  "9+1 =s TEN"
```

Execution of this directive causes the ATASCII numbers corresponding to "9", "+", etc., to be stored at successive locations starting at ADDR. Note that the ATASCII representation of any character except the quotation mark (") can be stored with the .BYTE directive having a string operand.

Exhibit I

Sample, Reproducible ATARI Programming Form

PROGRAM	PAGE OF	DATE
	PROGRAMMER	

LINE NO.	LABEL	OP CODE	OPERAND	COMMENT

NOTES:

3

USING THE EDITOR

Now that we have explained how to get started writing a program, it is up to you to actually write the program. This manual contains very little information on assembly language programming techniques. We assume that you are already familiar with assembly language. The remainder of the section describes how to use the Assembler Editor cartridge.

COMMANDS TO EDIT A PROGRAM

A command is not the same thing as an instruction. An instruction has a line number; a command has no line number and is executed immediately.

NEW Command

This command clears the edit text buffer. After this command you cannot restore your source program; it has been destroyed.

Some programmers have the habit of giving the NEW command (or its equivalent with other assemblers) when they start a programming session. The reason is to remove any "garbage" that may be in memory by mistake. Since the ATARI Personal Computer System clears its memory when it is turned on, such routine use of NEW would be a needless precaution. Because NEW destroys your entire source program, it is more important to develop a habit of NOT using it routinely. You should, rather, use NEW in a very deliberate fashion only when you want to remove a source program from RAM.

DEL Command

This command deletes statements from your source program.

DELxx	RETURN	deletes statement number xx.
DEL xx,yy	RETURN	deletes statement numbers xx through yy.

NUM Command

This command assigns statement numbers automatically.

NUM	RETURN	increments statement number by 10 after each RETURN. The new statement number, followed by a space, is automatically displayed.
NUMnn	RETURN	has the same effect as NUM, but the increment is nn instead of 10.
NUMmm,nn	RETURN	forces the next statement number to be mm and the increment to be nn.
RETURN	RETURN	cancels the NUM command.

The effect of the NUM command stops automatically when a statement number that already exists is reached. For example:

```
10     LDX #$EF
20     CMP MEMORY
NUM    15,5
15
```

After statement number 15, the next statement number would be 20, which already exists, so the NUM command is cancelled. The automatic numbering of statements will continue until the next number is exactly equal to an existing number. A slight change from the above example illustrates this:

```
10     LDX #$EF
20     CMP MEMORY
NUM    15,6
15     TAX
21
```

Caution: You cannot use the special keyboard editing keys to change other statements while the NUM command is in effect. You will succeed in changing what appears on the screen, but, in an exception to the general rule, the contents of the edit text buffer will not be changed.

REN Command

This command renumbers statements in your source program.

REN	RETURN	renumbers all the statements in increments of 10, starting with 10.
RENnn	RETURN	renumbers all the statements in increments of nn, starting with 10.
RENmm,nn	RETURN	renumbers all the statements in increments of nn, starting with mm.

FIND Command

This command finds a specified string. The ways to write the command are shown below.

FIND/SOUGHT/	RETURN	finds the first occurrence of the string SOUGHT. The statement that contains the string is displayed.
FIND/SOUGHT/,A	RETURN	finds all occurrences of the string SOUGHT. All statements containing such occurrences are displayed.
FIND/SOUGHT/xx	RETURN	finds the string SOUGHT if it occurs in statement number xx. Statement xx is displayed if it contains the string.
FIND/SOUGHT/xx,yy,A		finds all occurrences of the string SOUGHT between statement number xx and yy. All the statements that contain the string are displayed.

In these examples, the string SOUGHT is delimited (marked off) by the character /. Actually, any character except space, tab and RETURN can be used as the delimiter. For example, the command

FIND DAD

finds the first occurrence of the character A. The delimiter is the character D. The delimiter is defined as the first character (not counting space or tab) after the keyword FIND. This feature is perplexing to beginners; its purpose is to allow you to search for strings that contain slashes (/) or, for that matter, any special characters.

The general form of the command is

FIND delimiter string delimiter [lineno, lineno] [,A]

In the general form, symbols within a pair of brackets are optional qualifiers of the command.

REP Command

This command replaces a specified string in your source program with a different specified string.

REP/OLD/NEW	RETURN	replaces the first occurrence of the string OLD with the string NEW.
REP/OLD/NEW/xx,yy	RETURN	replaces the first occurrence of the string OLD between statements number xx to yy with the string NEW.
REP/OLD/NEW/,A	RETURN	replaces all the occurrences of the string OLD with the string NEW.
REP/OLD/NEW/xx,yy,A	RETURN	replaces all the occurrences of the string OLD between statements xx to yy with the string NEW.
REP/OLD/NEW/xx,yy,Q	RETURN	displays, in turn, each occurrence of the string OLD between statements xx and yy. Q stands for "query." To replace the displayed OLD with NEW, type Y, then RETURN. To retain the displayed OLD, press RETURN.

In these examples, the strings OLD and NEW are delimited by the character "/". As with the FIND command, any character except space, tab and RETURN, can be used as the delimiter. For example, the command

REP + RTS + BRK + ,A

replaces all occurrences of RTS with BRK. The delimiter is the character " + ".

The general form of this command is

REP delimiter OLD delimiter NEW delimiter [lineno, lineno] $\begin{bmatrix} Q \\ ,A \end{bmatrix}$

In the general form, symbols within a pair of brackets are optional qualifiers of the command and the symbols within braces (A and Q) are alternatives.

Sample Program

Let us assume you have written a program on an ATARI Programming Form as shown in Figure 6:

Exhibit I

Sample, Reproducible
ATARI Programming Form

PROGRAM SAM2.ASM	PAGE 1 OF 1	DATE 12/31/80
	PROGRAMMER JOHN DOE	

LINE NO.	LABEL	OP CODE	OPERAND	COMMENT
10		*=	$3000	
20		LDY	#$00	
30	REP	LDX	ABSX,Y	
40		BNE	XEQ	SAME PAGE
50		INY		TALLY
60		JMP	REP	
70	ABSX	=	$3744	
80	XEQ	=	*+$60	
90		.END		

Figure 6. Sample Program as you write it on the ATARI programming form

Then when you type it in it would appear on the screen as shown in Figure 7:

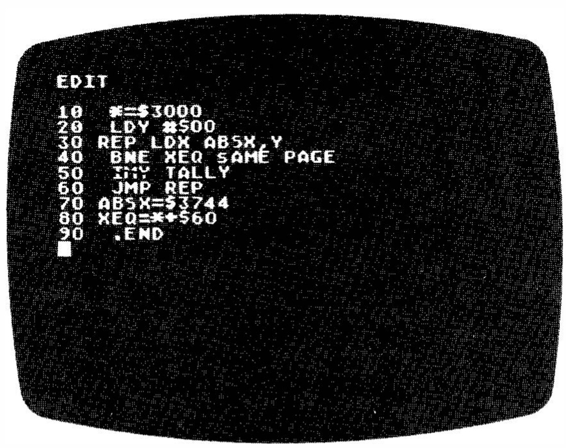


Figure 7. Appearance of the screen as your program is entered on the keyboard.

COMMANDS TO SAVE (OR DISPLAY) AND RETRIEVE PROGRAMS

The commands to save (or display) and retrieve programs are:

LIST	saves or displays a source program
PRINT	is the same as LIST, but omits line numbers
ENTER	retrieves a source program
SAVE	saves an object program
LOAD	retrieves an object program

With each of these commands there is a parameter that specifies the device that is the source or destination of the program that is to be saved, displayed or retrieved. The possible devices are different for different commands, and the default device is also different. Some of the commands have optional parameters that limit the application of the command to specified parts of the program.

The parameter that specifies the device that is the source or destination of the program is written as follows:

#E:	is the screen editor
#P:	is the printer
#C:	is the Program Recorder
#D[n]:FILENAME	is a disk drive. n is 1, 2, 3 or 4. D: is interpreted as D1:. A program saved on or retrieved from a diskette must be named (FILENAME).

LIST Command

Format: LIST# { device: / filespec } [,xx,yy]

Examples: LIST#E:

LIST#D:MYFILE

This command is used to display or save a source program. The device where the source program is to be displayed or saved is given in the command. If no device is specified, the screen is assumed by default. Other possible devices are the printer (#P:), Program Recorder (#C:) and disk drive (#D1: through #D8: or #D:, which defaults to #D1:). The commands to transfer a program (LIST it) to these various devices are:

LIST#E:	(LIST#E: is the same as LIST)
LIST#P:	
LIST#C:	(Use cassette-handling procedures described in your Program Recorder Operator's Manual.)
LIST#D:filename	where filename is an arbitrary name you give to the program. Filename must start with a letter and have no more than eight characters, consisting of letters and numbers only. It may also have an extension of up to three characters. For example, NAME3, ST5, and JOHN.23 are all legal names.

The forms of the commands to transfer only particular lines (lines xx to yy) to a device are:

LIST#E:,xx,yy	(LIST#E:,xx,yy is the same as LIST,xx,yy)
LIST#P:,xx,yy	
LIST#C:,xx,yy	(Use cassette-handling procedures described in the Program Recorder Operator's Manual.)
LIST#D:NAME,xx,yy	where "NAME" is an arbitrary name you give to the program. See the description above.

A single line may be displayed or saved with the command:

LISTlineno   where lineno is the line number.

Caution: The DOS makes sure that every file has a unique name by deleting old files if necessary. Therefore, do not name a file you are listing to diskette with the name of a file that is already stored on the diskette, unless you wish to replace the existing file with the one you are listing.

The LIST command is illustrated below. No device is specified, so the display device is the screen, by default. The small sample program, written in the previous section, is used for illustration.

```
EDIT
LIST RETURN
10  * = $3000
20  LDY #00
30 · REP LDX, ABSX, Y
40  BNE XEQ SAME PAGE
50  INY TALLY
60  JMP REP
70  ABSX = $3744
80  XEQ = * + $60
90  .END

EDIT
LIST30  RETURN

30 REP LDX ABSX, Y

EDIT
LIST 60,80  RETURN

60  JMP REP
70  ABSX = $3744
80  XEQ = * + $60

EDIT
▯
```

The examples above show the appearance of the screen, since that is the default device. The program or the particular lines in the examples could be displayed on the printer or saved on cassette or diskette by using the forms of the LIST command described above. Note that the commands tolerate a certain amount of variation in the insertion of blanks.

PRINT Command

This command is the same as LIST, except that it prints statements without statement numbers.

Example:

```
EDIT
PRINT  RETURN

* = $3000
LDY #00
REP LDX ABSX, Y
BNE XEQ SAME PAGE
INY TALLY
JMP REP
ABSX = $3744
XEQ = * + $60
.END

EDIT
PRINT30  RETURN

REP LDX ABSX, Y

RETURN
EDIT
PRINT 60,80  RETURN

JMP REP
ABSX = $3744
XEQ = * + $60

RETURN
EDIT
▯
```

After using a PRINT command, no further command can be entered until you press RETURN, which causes the EDIT message and cursor to be displayed.

ENTER Command

Format: ENTER# { device: / filespec }

Examples: ENTER#C:
ENTER#D:MYFILE

The command ENTER is used to retrieve a source program. As with the command LIST, a device has to be specified, in this case the device where the program is stored. There is only one device, the disk drive, on which a named program is stored in a retrievable form. To retrieve a source program from a diskette in a disk drive, the command is:

```
ENTER#D:NAME
```

where “NAME” is the arbitrary name you gave to the program when you listed it on the diskette. This command clears the edit text buffer before transferring data from the diskette.

To retrieve a source program from cassette, the command is:

ENTER#C: (Follow the CLOAD procedure given in your 410 Program Recorder Operator’s Manual.) Note that ENTER #C: clears the edit textbuffer before retrieving the source program.

To merge a source program on cassette with the source program in the edit text buffer, the command is:

ENTER#C:,M

In the above command, where a statement number is used twice (in the edit text buffer and on tape), the statement on cassette prevails.

Commands for saving and retrieving an object program are SAVE and LOAD. They correspond to LIST and ENTER, respectively.

SAVE Command

Format: SAVE# $\left\{\begin{array}{l}\text{device:}\\ \text{filespec}\end{array}\right\}$ < address1,address2

Examples: SAVE#C: < 1235,1736
SAVE#D2:MYFILE < 1235,1736

To save an object program residing in hex address1 to address2 on cassette or diskette, the commands are:

SAVE#C: < address1,address2

CAUTION: Use the CSAVE procedure illustrated in your 410 Program Recorder Operator’s Manual.

SAVE#D:FILENAME < address1,address2

where FILENAME is an arbitrary name you give to the block of memory that you are saving (where your object program is stored).

LOAD Command

Format: LOAD# $\left\{\begin{array}{l}\text{device:}\\ \text{filespec}\end{array}\right\}$

Examples: LOAD#C:
LOAD#D:MYFILE

To retrieve an object program that had previously been SAVED and which had previously been called NAME, the command is:

LOAD#D:NAME	where NAME is the arbitrary name that you gave to the object program when you saved it on diskette.
LOAD#C:	(Use the CLOAD procedure described in your 410 Program Recorder Operator's Manual.)

These commands will reload the memory locations address1 to address2 with the contents that were previously saved. The numbers address1 and address2 are those that were given in the original SAVE command.

NOTES:

4

USING THE ASSEMBLER

THE ASM COMMAND

The general form of the ASSEMBLE command is

```
                                  ,{#P:}
                                  ,{#E:}
ASM [#D[n]:PROGNAME[.SRC]]        ,{#D[n]:LISTING[.LST]}        [,#D[n]:SEMBLED[.OBJ]]
```

Where source program is located

Where assembly listing is to be stored or displayed

Where object program is to be stored

The default values of the three parameters of the ASM command are the edit text buffer for the source program, the television screen for the assembly listing, and computer RAM for the object program (the assembled program). To assemble a program using default values of ASM, type

ASM RETURN

On receiving this command, the Assembler translates the source program in the edit text buffer into object code and writes the object code into the memory locations specified in the source program. When this process is completed, the assembled program is displayed on the screen. For an example of assembly with default parameter values, we use the small sample program that we wrote. Figure 8 shows the appearance of the screen after the ASM command.

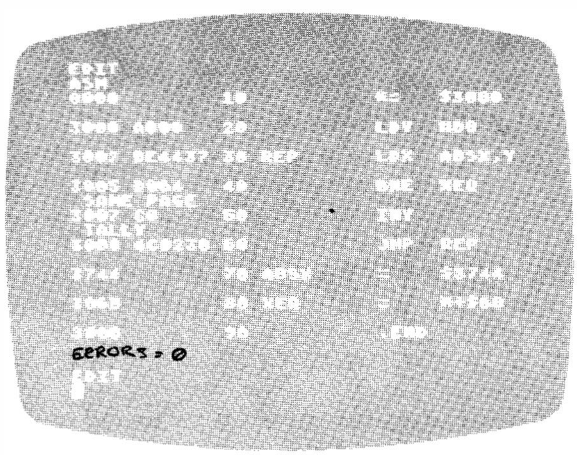


Figure 8. Appearance of the screen as your sample program is assembled.

Using statements 30 and 40 as examples, the format of the assembled program is shown below. Note, however, that some of the spacing can be changed by the TAB directive.

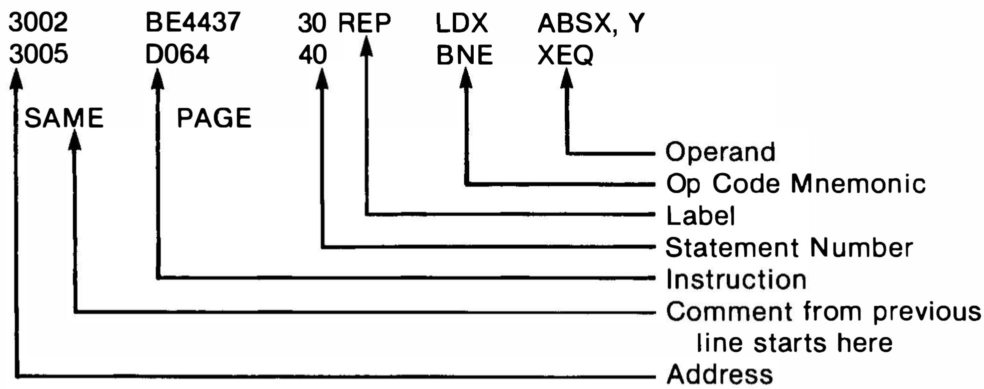


Figure 9. Normal (default) format of assembly listing as it appears on the screen.

The general form of the command shown at the beginning of the section shows how to override the default values of the parameters of the command. These override selections are explained below.

Location of Source Program

You may specify the location of the source program as a named program on diskette. You must have previously stored the source program under that name, using the LIST command. In the general form of the ASM command, the source program on diskette has been given the extension .SRC. Extensions are optional.

Where Assembly Listing Is To Be Stored

The default value is the screen (#E:). The other possibilities are the printer (#P:), the Program Recorder (#C:), and the disk drive (#D[n]:NAME [.LST]).

Where Object Program Is To Be Stored

You may specify that the assembled program is to be stored directly on diskette, using any name (subject to the restrictions of DOS). In the general form of the ASM command, the assembled program has been given the extension .OBJ. Extensions are optional.

It is easy to become confused by names of programs when a program may exist in several related forms. To reduce the chance of confusion, we recommend using names that include identifying extensions, such as .SRC, .LST and .OBJ for a source program, an assembly listing and an object program, respectively.

Note that in the ASM command the source program must be in the edit text buffer or on a diskette in the disk drive. It can not be on a cassette in the Program Recorder. The primary reason for this restriction is that the Assembler requires two passes of the source program and the Program Recorder is not controllable to permit two passes. However, you can assemble a source program recorded with your Program Recorder. First transfer the program from Program Recorder to the edit text buffer with the command:

ENTER#C: RETURN (Follow the cassette-handling instructions in your Program Recorder Operator's Manual.)

The ASM command with no default parameters is illustrated in the example below:

ASM#D:SOURCE,#P:,#D2:SEMBLED.OBJ RETURN

The above command takes the source program that you had previously stored on diskette and called SOURCE, assembles it, lists the assembled form on the printer, and records on the diskette the machine code translation of the program (the object program). The object program is given the name "SEMBLED.OBJ." Note that commands of this form store the machine code on diskette, not in computer RAM.

To make a default selection, enter a comma, as in the following useful command:

ASM,#P: RETURN

The above command takes the source program from the default edit text buffer, assembles and lists it on the printer as before, and stores the machine code object program directly into computer RAM.

DIRECTIVES (PSEUDO OPERATIONS)

Directives are instructions to the Assembler. Directives do not, in general, produce any assembled code, but they affect the way the Assembler assembles other instructions during the assembly process. Directives are also called pseudo operations or pseudo ops.

Directives are identified by the Assembler by the "." at the beginning. The only exceptions are the LABEL= directive and the *= directive.

A directive must have a line number, which it follows by at least two spaces. The directive LABEL= is an exception—there must be only one space before the label.

OPT Directive

This directive specifies an option. There are four sets of options. These are:

. OPT NOLIST
. OPT LIST (this is the default condition)

. OPT NOOBJ
. OPT OBJ (this is the default condition)

. OPT NOERR
. OPT ERR (this is the default condition)

. OPT NOEJECT
. OPT EJECT (this is the default condition)

The second listed of each pair represents the standard or default condition.

100 . OPT NOLIST (part of source program) 200 . OPT LIST	The effect of these directives is to omit from the listed form of the assembled program the lines between lines 100 and 200. (These line numbers are arbitrary.)
100 . OPT NOOBJ (part of source program) 200 . OPT OBJ	Assembly is suppressed between lines 100 and 200. The effect of these directives is to omit from the object program code corresponding to the lines between lines 100 and 200. Memory corresponding to these lines is skipped over, leaving a region of untouched bytes in the object program. (These line numbers are arbitrary.)
100 . OPT NOERR (part of source program) 200 . OPT ERR	The effect of these directives is to omit error messages for the assembled program lines between lines 100 and 200.
100 . OPT NOEJECT (part of source program) 200 . OPT EJECT	The effect of these directives is to suppress, between lines 100 and 200, the 4-line page spacing that is normally inserted after every 56 lines of the listed form of the assembled program.

More than one option may appear on a line. Options are then separated by a comma, as follows:

```
1000   . OPT NOLIST,NOOBJ
```

Title and Page Directives

```
10   . TITLE "name"
20   . PAGE "optional message"
```

We explain these directives together because they are intended to be used together to provide easily read information about the assembled program.

These directives are most useful when the assembled program is listed on the printer.

TITLE and PAGE allow you to divide your program listing into segments that bear messages written for your own convenience, much as you might add short explanatory notes to any complex material.

The PAGE directive causes the printer to put out six blank lines (printers so equipped will execute a TOP OF FORM), followed by the messages you have given for TITLE and PAGE. This causes the messages to stand out somewhat from the rest of the assembled program listing.

Usually there is only one TITLE directive, giving the program name and date, and different PAGE directives for giving different page messages. Then on listing the assembled program, the same TITLE message on every page would be followed by a different PAGE message.

The blank lines that the PAGE directive produces on the 40-column ATARI 820 Printer can be used to break up a long program into segments that can be mounted in a notebook.

To remove a title, use the following form:

```
1000  . TITLE  " "
```

The above directive removes titles after line 1000.

The PAGE directive on its own causes a page break—the printer simply puts out a number of blank lines.

Tab Directive

```
10  . TAB n1,n2,n3
```

The TAB directive sets the fields of the statement as they appear when assembled and listed on the screen or the printer. Let us use the specific example of Statement 40 of the small sample program we previously used for illustration. It was written as follows:

```
30 ...
40 BEQ XEQ SAME PAGE
50 ...
```

Note that one space, rather than a tab, is used between each field. Using spaces rather than tabs lets you write longer programs, since the edit text buffer will not be filled up with the extra spaces that tabs would require.

Compressing the program in this way makes it less easily readable than we might wish, but we can use the TAB directive to give us a more readable assembled version. The form of the directive is

lineno . TAB 10,15,20
or, more generally,
lineno . TAB number1,number2,number3

The previous example has a source program that was compressed in the above fashion. Note the difference between the spacing of the source listing and the assembled program. This is an example of the default TAB spacing.

The effect of the TAB directive of line xxx is confined to the appearance of lines following xxx when they are assembled and listed on the printer or screen.

In the case of line 40, the appearance on the printer would be as shown below:

```
3005  D064  40    BNE  XEQ  SAME PAGE
            |-10-|    |    |
            |-----15--|    |
            |---------20---|
```

If the TAB directive is not used, then the appearance of the assembler line on the printer will be as shown below in the default mode:

```
3005  D064    40    BNE  XEQ  S
AME PAGE
              |-12-|    |    |
              |-----17--|    |
              |---------27---|
```

That is, the default setting corresponds to . TAB 12,17,27.

The appearance of this line on the screen will be different only because the screen has 38 characters positions, while the printer has 40.

BYTE, DBYTE and WORD Directives

100 . BYTE a,b,... ,n
200 . ~~BYTE "A,B,...N"~~
300 . DBYTE a,b,... ,n
400 .WORD a,b,... ,n

These directives are similar in that they are used to insert data rather than instructions into the proper places in the program. Each directive is slightly different in the manner in which it inserts data.

BYTE Directive

The BYTE directive reserves a location (at least one) in memory. The directive increments the program counter to leave space in memory to be filled by information required by the program. The operand supplies the data to go into that space.

Examples:

```
10 .....
20 . BYTE 34
30 .....
```

Here, the Assembler assembles into successive locations the instruction of line 10, then the decimal number 34, then the instruction of line 30.

```
10 .....
20 . BYTE 34, 56,78
30 .....
```

Here, the Assembler assembles into successive locations the instruction of line 10, then the decimal numbers 34, 56 and 78, then the instruction of line 30. The operand may be an expression more complex than the numbers used in the examples. The rules for writing and evaluating an expression are given in Appendix ~~D:~~ 5

```
10 .....
20 . BYTE "ATARI"
30 .....
```

Here, the Assembler assembles into successive locations the instruction of line 10, then the (ATASCII code) hex numbers 41, 54, 41, 52 and 49, then the instruction of line 30.

DBYTE Directive

The DBYTE directive reserves two locations for each expression in the operand. The value of the expression is assembled with the high-order byte first (in the lower number location). For example:

```
10 * = $4000
20 . DBYTE   ABS + $3000
```

When line 20 is assembled and the value of ABS + $3000 is determined to be (say) $5123, $51 is put in location $4000 and $23 is put in location $4001.

WORD Directive

The WORD directive is the same as the DBYTE directive except that the value of the expression is stored with the low-order byte first.

For example:

```
10  * = $4000
20  .WORD   ABS + $3000
```

When line 20 is assembled and the value of ABS + $3000 is determined, as before, to be $5123, $23 is put in location $4000 and $51 is put in location $4001.

The WORD directive simplifies some programming since addresses in machine code are always given in the order low byte followed by high byte. Therefore, the WORD directive is useful, for example, in constructing a table of addresses.

LABEL = DIRECTIVE

100 LABEL = expression

The LABEL = directive is used to give a value to a label. Two examples appear in the sample program we used before. Statements 80 and 70 give values to ABSX and XEQ, as follows:

70 ~~60~~ ABSX = $3744
80 ~~70~~ XEQ= * + $60

Since the symbol that is given a value is a label, there must be only one space after the statement number. The expression on the right cannot have a value greater than FFFF (hex). The rules for writing and evaluating an expression are given in Appendix 5.

When the LABEL = directive is used to give a value to a label, the label can be used in an operand, by itself, as in statements 30 and 40 in the sample program.

A defined label may also appear as part of an expression. Our sample program does not contain an example, so we give one below in line 240.

```
100   TAB1 = $3000
...............
...............
...............
240   TAB2 = TAB1 + $20
```

When the program is assembled, TAB2 will be given the value $3020.

You should note that defining a label in this way gives the label a specific address; it does not define the contents of the address. In the example, above, TAB1 and TAB2 might be the location of two tables that contained the values of variables that your program required.

*** = Directive**

100   * =   expression

We encountered the * = directive in the "getting started" commands, where it is used to set the starting location of the assembled program. When the Assembler encounters the * = expression, it sets the program counter to the value of the expression.

You write *= without the initial "." that the other directives have (except LABEL=). Also, note that you write *= without any spaces between * and =.

You should not confuse the *= directive with the LABEL= directive. The * in *= is not a label. Note, however, that the *= directive itself may have a label, as follows:

```
200  GO   *=expression
500  JMP  GO
```

The Assembler will assemble statement 500 as a jump to the value the program counter had BEFORE it was changed by line 200.

The *= directive is useful for setting aside space needed by your program. For example, you will frequently want space reserved starting at a particular location. Use the following form:

```
720  TABLE35   *=*+$24
740  ...
```

The effect of the directive is to reserve ~~24~~ locations immediately after TABLE35. Other parts of your code will not be assembled into these locations (unless you take pains to do so). Your program can use TABLE35 as an operand and TABLE35 can be used as an element in an expression that your instructions evaluate in accessing the table.

IF Directive

```
900. IF expression @LABEL
.
.
.
990  LABEL  End of conditional assembly
```

The IF directive permits conditional assembly of blocks of code. In the illustration above, all the code between lines 900 and 990 will be assembled if and only if the expression is equal to zero. If the expression is not equal to zero, ~~the~~ IF directive has ~~no effect on assembl~~y.

The example given below shows how different parts of a source program may be omitted from assembly according to the value assigned to the LABEL in the IF directive. The source program is assembled with Z=0 in one case and Z=1 in another. With Z=0, the instruction TAX is assembled, and with Z=1 the instruction ASL A is assembled. Obviously, this kind of selective assembly can be extended indefinitely.

SOURCE CODE

```
0100  ;CONDITIONAL ASSEMBLY EXAMPLE
0120  Z=0
0130  *=$5000
0140  LDA=$45
0150  . IF Z@ZNOTEQUAL0
0160  TAX ;THIS CODE ASSEMBLED IFF Z=0
0170  ZNOTEQUAL0
0180  . IF Z-1@ZNOTEQUAL1
0190  ASL A   ;THIS CODE ASSEMBLED IFF Z=1
0200  ZNOTEQUAL1
0210  INX ;THIS CODE ALWAYS ASSEMBLED
```

ASSEMBLY LISTING (40-col. format)

```
    0100 ;CONDITIONAL ASSEMBLY E
XAMPLE
0000  0120 Z  =  0
0000  0130  *=  $5000
5000 A945  0140  LDA  #$45
5002  0150  .IF  Z@ZNOTEQUA
L0
5002 AA  0160  TAX  ;
THIS CODE ASSEMBLED IFF Z=0
    0170  ZNOTEQUAL0
5003  0180  .IF  Z-1@ZNOTEQ
UAL1
    0190  ASL A
    0200  ZNOTEQUAL1
5003 E8  0210  INX  ;
THIS CODE ALWAYS ASSEMBLED

0100  ;CONDITIONAL ASSEMBLY E
XAMPLE
0001  0120 Z  =  1
0000  0130  *=  $5000
5000 A945  0140  LDA  #$45
5002  0150  .IF  Z@ZNOTEQUA
L0
    0160  TAX  ;THIS CODE ASSEMBL
ED IFF Z=0
    0170 ZNOTEQUAL0
5002  0180  .IF  Z-1@ZNOTEQ
UAL1
5002 OA  0190  ASL A
0200 ZNOTEQUAL1
5003 E8  0210  INX  ;
THIS CODE ALWAYS ASSEMBLED
```

END Directive

```
1000  .END
```

Every program should have one and only one END directive. It tells the Assembler to stop assembling. It should come at the very end of your source program. Later, if you decide to add more statements to your program, you should remove the old .END directive and place a new one at the new end of your source program. Failure to do so will result in your added source code not being assembled. This mistake is particularly easy to make when you make your additions with the NUM command. It is not always essential to have an .END directive, but it is good practice.

NOTES:

5

DEBUGGING

PURPOSE OF DEBUGGER

The Debugger allows you to follow the operation of an object program in detail and to make minor changes in it.

A knowledge of machine language is helpful when you use the debugger, but it is not essential. The Debugger is able to convert machine code into assembly language (disassemble), so you can make code alterations at particular memory locations. All numbers used by the Debugger, both in input and output, are hexadecimal.

CALLING THE DEBUGGER

The Debugger is called from the Editor by typing:

BUG RETURN

This produces on the screen:

DEBUG
[ ]

The command to return to the Writer/Editor is:

X RETURN

DEBUG COMMANDS

The debug commands are listed below. In the list, "mmmm" indicates that the form of the command may include memory address(es). The actual methods of specifying the memory address(es) and the default addresses are shown in the examples given later in this section. If you use the commands with no address(es), the Debugger assigns a default address(es):

DR	Display Registers
CR	Change Registers
D or Dmmmm	Display Memory
C or Cmmmm	Change Memory
Mmmmm	Move Memory
Vmmmm	Verify Memory
L or Lmmmm	List Memory With Disassembly
A	Assemble One Instruction Into Memory
T or Tmmmm	Trace Operation
S or Smmmm	Single-Step Operation
Gmmmm	Go (Execute Program)
X	Return to EDITOR
BREAK	Pressing the BREAK key halts certain operations.

We now give several examples showing how to use the commands. In the examples, the lines ending with RETURN are entered on the keyboard. The other lines show the response of the system, as displayed on the screen.

DR Display Registers

Example:

```
EDIT
BUG RETURN

DEBUG
DR RETURN
      A=BA   X=12   Y=34   P=B0   S=DF
DEBUG
[]
```

The registers and contents are displayed as shown. A is the Accumulator, X and Y are the Index Registers, P is the Processor Status Register, and S is the Stack Pointer.

CR Change Registers

Example:

```
EDIT
BUG RETURN

DEBUG
CR<1,2,3,4,5 RETURN

DEBUG
[]
```

The effect of the command above is to set the contents of the registers A, X, Y, P, and S to 1, 2, 3, 4 and 5.

You can skip registers by using commas after the <. For example,

```
CR<,,,,E2 RETURN
```

sets the Stack Pointer to E2 and leaves the other registers unchanged. Registers are changed in order up to RETURN. For example,

```
CR<,34 RETURN
```

sets the X Register to 34 and leaves the other registers unchanged.

D or Dmmmm Display Memory

Dmmmm, yyyy where vvvv is less than or equal to mmmm shows the contents ~~of~~ address mmmm.

Example:

```
DEBUG
D5000,0 RETURN

5000  A9
DEBUG
[]
```

~~This~~ shows that address 5000 ~~contains the number~~ A9.

If the second address (yyyy) is omitted, the contents of eight successive locations are shown. The process can be continued by typing D RETURN.

Example:

```
DEBUG
D5000 RETURN

5000  A9  03  18  E5  F0  4C  23  91
DEBUG
D RETURN

5008  18  41  54  41  52  49  20  20
DEBUG
[ ]
```

Dmmmm,yyyy where yyyy is greater than mmmm, shows the contents of addresses mmmm to yyyy.

Example:

```
DEBUG
D5000,500F    RETURN

5000  A9  03  18  E5  F0  4C  23  91
5008  18  41  54  41  52  49  20  20
DEBUG
[ ]
```

The display can be stopped by pressing the BREAK key.

C or Cmmmm Change Memory

Cmmmm < yy changes the contents of address mmmm to yy.

Example:

```
DEBUG
C5001 <23 RETURN

DEBUG
[ ]
```

The effect of the command is to put the number 23 in location 5001. A comma increments the location to be changed.

Example:

```
DEBUG
C500B <21,EF RETURN

DEBUG
C700B <31,,,87 RETURN

DEBUG
[ ]
```

The first command puts 21 and EF in locations 500B and 500C, respectively.

The second command puts ~~34~~ 31 and 87 in locations 700B and 700E respectively.

You can conveniently use the C command in conjunction with the Display Memory command, and you need not enter the address a second time with the C command. The C command will default to the last specified address.

Example:

```
D5000 RETURN

5000 A0 03 18 E5 F0 4C 23 91
C<AA,14 RETURN

D5000 RETURN

5000 AA 14 18 E5 F0 4C 23 91

DEBUG
[ ]
```

Mmmmm Move Memory

Mmmmm<yyyy,zzzz copies memory from yyyy to zzzz to memory starting at mmmm. Address mmmm must be less than yyyy or greater than zzzz. If the origin and destination blocks overlap, results may not be correct.

Example:

```
DEBUG
M1230<5000,500F RETURN

DEBUG
[ ]
```

The command copies the data in location 5000-500F to location 1230-123F.

Vmmmm Verify Memory

Vmmmm<yyyy,zzzz compares memory yyyy to zzzz with memory starting at mmmm, and shows mismatches.

Example:

```
DEBUG
V7000<7100,7123 RETURN

DEBUG
[ ]
```

The command compared the contents of 7100-7123 with the contents of 7000-7023. There were no mismatches.

Mismatches would be shown as follows:

```
7101  00  7001  22
7105  18  7005  10
```

L or Lmmmm List Memory With Disassembly

This command allows you to look at any block of memory in disassembled form.

Examples:

L7000	RETURN	List a screen page (20 lines of code) starting at memory location 7000. Pressing the BREAK key during listing halts the listing.
L	RETURN	This form of the command lists a screen page starting at the instruction last shown, plus 1.
L7000, 0 L7000, 7000 L7000, 6000	RETURN RETURN RETURN	These forms list the instructions at address 7000 only.
L345, 567	RETURN	This form lists address 345 through 567. Only the last 20 instructions will actually be visible at the completion of the response of the system.

The command Lmmmm differs from Dmmmm in that Lmmmm disassembles the contents of memory.

Example:

```
EDIT
BUG  RETURN

DEBUG
L5000, 0  RETURN
5000      A9  03     LDA      #$03

DEBUG
[]
```

This example shows that the Debugger examined the contents of memory address 5000 and disassembled A9 to LDA. Since A9 must have a one-byte operand, the Debugger made the next byte (the contents of address 5001) the operand. Therefore, although the debugger was only "asked" for the content of location 5000, it showed a certain amount of intelligence and replied by showing the instruction that started at address 5000.

To illustrate this further, the number 03 corresponds to no machine code instruction, so the Debugger would interpret 03 as an illegal instruction, and alert you to a possible error, as shown below.

Example:

```
DEBUG
L5001, 0  RETURN
5001, 03    ???

DEBUG
```

However, if the first instruction you wrote was LDA $8A, then you would have obtained the following, apparently inconsistent, results while debugging:

Example:

```
DEBUG
L5000, 00   A9   8A   LDA #$8A

DEBUG
L5001, 0  8A   TXA
```

Because the disassembler starts disassembling from the first address you specify, you have to take care that the first address contains the first byte of a "real" instruction.

A Assemble One Instruction Into Memory

The DEBUGGER has a mini-assembler, that can assemble one assembly language instruction at a time. To enter the Assemble mode, type:

A RETURN

Once in the Assemble mode, you stay there until you wish to return to DEBUGGER, which you may do by pressing RETURN (on an empty line).

To assemble an instruction, first enter the address at which you wish to have the machine code inserted. The number that you enter will be interpreted as a hex address. Now type "<" followed by at least one space, then the instruction. You may omit an address if assembly is to be in successive locations.

Example:

```
EDIT
BUG  RETURN

DEBUG
A RETURN
5001<LDY $1234 RETURN
5001  AC3412              Computer Responds.
<INY RETURN
5004  CB                  Computer Responds.
[ ] RETURN

DEBUG
[ ]
```

Since the mini-assembler assembles only one instruction at a time, it cannot refer to another instruction. Therefore, it cannot interpret a label. Consequently, labels are not legal in the mini-assembler.

You can use the directives BYTE, DBYTE, and WORD.

Gmmmm Go (Execute Program)

This command executes instructions starting at mmmm. For example:

G7B00 RETURN Executes instructions starting at location 7B00. Execution continues indefinitely. Execution is stopped by pressing the BREAK key (unless the program at 7B00 tricks or crashes the operating system).

Tmmmm Trace Operation

This command has the same effect as Gmmmm, except that after execution of each instruction the screen shows the instruction address, the instruction in machine code, the instruction in assembly language (disassembled by the debugger—not necessarily the same as you wrote it in assembly language) and the values of Registers A, X, Y, P and S.

The execution stops at a BRK instruction (machine code 00) or when you press the BREAK key on the keyboard.

Example:

```
DEBUG
T5000 RETURN
5000        A9      03              LDA       #$03
   A=03     X=02      Y=03    P=34       S=05
5002        18                      CLC
   A=03     X=02      Y=03    P=34       S=05
5003        E5      F0              SBC       $F0
   A=03     X=02      Y=03    P=34       S=05
5005        4C      23      71      JMP       $7123
   A=03     X=02      Y=03    P=34       S=05
7123        00                      BRK
   A=03     X=02      Y=03    P=34       S=05
DEBUG
```

S or Smmmm Step Operation

This command has the same effect as T or Tmmmm, except that only one instruction is executed. To step through a program, type S RETURN repeatedly after the first command of Smmmm RETURN

X Exit

To return to the Editor type:

X RETURN

NOTES:

APPENDIX 1

ERRORS

When an error occurs, the console speaker gives a short "beep" and the error number is displayed.

Errors numbered less than 100 refer to the Assembler Editor cartridge, as follows:

ERROR NUMBER

1.	The memory available is insufficient for the program to be assembled.
2.	For the command "DEL xx,yy" the number xx cannot be found.
3.	There is an error in specifying an address (mini-assembler).
4.	The file named cannot be loaded.
5.	Undefined label reference.
6.	Error in syntax of a statement.
7.	Label defined more than once.
8.	Buffer overflow.
9.	There is no label or * before "=".
10.	The value of an expression is greater than 255 where only one byte was required.
11.	A null string has been used where invalid.
12.	The address or address type specified is incorrect.
13.	Phase error. An inconsistent result has been found from Pass 1 to Pass 2.
14.	Undefined forward reference.
15.	Line is too large.
16.	Assembler does not recognize the source statement.
17.	Line number is too large.
18.	LOMEM command was attempted after other command(s) or instruction(s). LOMEM, if used, must be the first command.
19.	There is no starting address.

Errors

Errors numbered more than 100 refer to the Operating System and the Disk Operating System. For a complete list of DOS errors, refer to the DOS manual.

128	BREAK key pressed during an I/O operation.
130	A nonexistent device specified for I/O.
132	The command is invalid for the device.
136	EOF. End of file read has been reached. This error may occur when reading from cassette.
137	A record was longer than 256 characters.
138	The device specified in the command does not respond. Make sure the device is connected to the console and powered.
139	The device specified in the command does not return an Acknowledge signal.

140	Serial bus input framing error.
142	Serial bus data frame overrun.
143	Serial data checksum error.
144	Device done error.
145	Diskette error: Read-after-write comparison failed.
146	Function not implemented.
162	Disk full.
165	File name error.

APPENDIX 2

ASSEMBLER MNEMONICS (Alphabetic List)

ADC	Add Memory to Accumulator with Carry
AND	AND Accumulator with Memory
ASL	Shift Left (Accumulator or Memory)
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BEQ	Branch if Result = Zero
BIT	Test Memory Against Accumulator
BMI	Branch if Minus Result
BNE	Branch if Result ≠ Zero
BPL	Branch on Plus Result
BRK	Break
BVC	Branch if V Flag Clear
BVS	Branch if V Flag Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode Flag
CLI	Clear Interrupt Disable Flag (Enable Interrupt)
CLV	Clear V Flag
CMP	Compare Accumulator and Memory
CPX	Compare Register X and Memory
CPY	Compare Register Y and Memory
DEC	Decrement Memory
DEX	Decrement Register X
DEY	Decrement Register Y
EOR	Exclusive-OR Accumulator with Memory
INC	Increment Memory
INX	Increment Register X
INY	Increment Register Y
JMP	Jump to New Location
JSR	Jump to Subroutine
LDA	Load Accumulator
LDX	Load Register X
LDY	Load Register Y
LSR	Shift Right (Accumulator or Memory)
NOP	No Operation
ORA	OR Accumulator with Memory
PHA	Push Accumulator on Stack
PHP	Push Processor Status Register (P) onto Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status Register (P) from Stack
ROL	Rotate Left (Accumulator or Memory)
ROR	Rotate Right (Accumulator or Memory)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode Flag
SEI	Set Interrupt Disable Flag (Disable Interrupt)

STA	Store Accumulator
STX	Store Register X
STY	Store Register Y
TAX	Transfer Accumulator to Register X
TAY	Transfer Accumulator to Register Y
TSX	Transfer Register SP to Register X
TXA	Transfer Register X to Accumulator
TXS	Transfer Register X to Register SP
TYA	Transfer Register Y to Accumulator

APPENDIX 3
SPECIAL SYMBOLS

Below we give a list of special symbols that have a restricted meaning to the Assembler. You should avoid using these symbols as a matter of course. Most attempts to use these symbols in any but their special sense will result in errors. They may be used, without their special meaning, in comments and in the operands of memory reservation directives.

; The semicolon is used to indicate the start of a comment. Everything between the semicolon and RETURN appears in the listed form of the program and is ignored by the Assembler. When comments take more than one line, start each new line with a semicolon.

The # sign is used as the first symbol of an immediate operand, as in LDX #24.

$ The $ sign is used before numbers to signify that they are to be interpreted as hex numbers. For example, LDX #$34.

***** The asterisk is used to signify the value of the current location counter. For example, the instruction in line 50 gives the symbol HERE a value equal to 5 or more than the number in the current location counter:

```
50  HERE=*+5
```

Example:

```
18  *=$911
19  RTS
20  *=*+$F
21  TAX
```

When this example is assembled, line 18 causes the location counter to be $0911, RTS is placed in location $0911, line 20 causes the location counter to be increased from $0912 to $0921, and TAX is placed in $0921. This leaves 15 empty bytes between the RTS and the TAX.

The asterisk also signifies multiplication (see Appendix 6). The Assembler uses the syntax of the statement to distinguish the two meanings of the asterisk.

Register names:

A Accumulator
X X Register
Y Y Register
S Stack Pointer
P Processor Status Register

NOTES:

TABLE OF HEX DIGITS WITH CORRESPONDING OP CODE MNEMONICS AND OPERANDS

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	LSD / MSD
0	BRK	ORA-IND, X				ORA-Z, Page	ASL-Z, Page		PHP	ORA-IMM	ASL-A			ORA-ABS			0
1	BPL	ORA-IND, Y				ORA-Z, Page, X	ASL-Z, Page, X		CLC	ORA-ABS, Y				ORA-ABS, X	ASL-ABS, X		1
2	JSR	AND-IND, X			BIT-Z, Page	AND-Z, Page	ROL-Z, Page		PLP	AND-IMM	ROL-A		BIT-ABS	AND-ABS	ROL-ABS		2
3	BMI	AND-IND, Y				AND-Z, Page	ROL-Z, Page, X		SEC	AND-ABS, Y				AND-ABS, X	ROL-ABS, X		3
4	RTI	EOR-IND, X				EOR-Z, Page	LSR-Z, Page		PHA	EOR-IMM	LSR-A		JMP-ABS	EOR-ABS	LSR-ABS		4
5	BVC	EOR-IND, Y				EOR-Z, Page, X	LSR-Z, Page, X		CLI	EOR-ABS, Y				EOR-ABS, X	LSR-ABS, X		5
6	RTS	ADC-IND, X				ADC-Z, Page	ROR-Z, Page		PLA	ADC-IMM	ROR-A		JMP-IND	ADC-ABS	ROR-ABS		6
7	BVS	ADC-IND, Y				ADC-Z, Page, X			SEI	ADC-ABS, Y				ADC-ABS, X			7
8		STA-IND, X			STY-Z, Page	STA-Z, Page	STX-Z, Page		DEY		TXA		STY-ABS	STA-ABS	STX-ABS		8
9	BCC	STA-IND, Y			STY-Z, Page, X	STA-Z, Page, X	STX-Z, Page, Y		TYA	STA-ABS, Y	TXS			STA-ABS, X			9
A	LDY-IMM	LDA-IND, X	LDX-IMM		LDY-Z, Page	LDA-Z, Page	LDX-Z, Page		TAY	LDA-IMM	TAX		LDY-ABS	LDA-ABS			A
B	BCS	LDA-IND, Y			LDY-Z, Page, X	LDA-Z, Page, X	LDX-Z, Page, Y		CLV	LDA-ABS, Y	TSK		LDY-ABS, X	LDA-ABS, X	LDX-ABS, Y		B
C	CPY-IMM	CMP-IND, X			CPY-Z, Page	CMP-Z, Page	DEC-Z, Page		INV	CMP-IMM	DEX		CPY-ABS	CMP-ABS	DEC-ABS		C
D	BNE	CMP-IND, Y				CMP-Z, Page, X	DEC-Z, Page, X		CLD	CMP-ABS, Y				CMP-ABS, X	DEC,ABS, X		D
E	CPX-IMM	SBC-IND, X			CPX-Z, Page	SBC-Z, Page	INC-Z, Page		INX	SBC-IMM	NOP		CPX-ABS	SBC-ABS	INC-ABS		E
F	BEQ	SBC-IND, Y			SBC-Z, Page, X	INC-Z, Page, X			SED	SBC-ABS, Y				SBC-ABS, X	INC-ABS, X		F

IMM - IMMEDIATE ADDRESSING — THE OPERAND IS CONTAINED IN THE SECOND BYTE OF THE INSTRUCTION.

ABS - ABSOLUTE ADDRESSING — THE SECOND BYTE OF THE INSTRUCTION CONTAINS THE 8 LOW ORDER BITS OF THE EFFECTIVE ADDRESS. THE THIRD BYTE CONTAINS THE 8 HIGH ORDER BITS OF THE EFFECTIVE ADDRESS (EA).

Z, PAGE - ZERO PAGE ADDRESSING — SECOND BYTE CONTAINS THE 8 LOW ORDER BITS OF THE EFFECTIVE ADDRESS. THE 8 HIGH ORDER BITS ARE ZERO.

A, - ACCUMULATOR — ONE BYTE INSTRUCTION OPERATING ON THE ACCUMULATOR.

Z, PAGE X - Z, PAGE, Y - ZERO PAGE INDEXED — THE SECOND BYTE OF THE INSTRUCTION IS ADDED TO THE INDEX (CARRY IS DROPPED) TO FORM THE LOW ORDER BYTE OF THE EA. THE HIGH ORDER BYTE OF THE EA IS ZEROS.

ABS, X¬ABS, Y ABSOLUTE INDEXED — THE EFFECTIVE ADDRESS IS FORMED BY ADDING THE INDEX TO THE SECOND AND THIRD BYTE OF THE INSTRUCTION.

(IND, X) - INDEXED INDIRECT — THE SECOND BYTE OF THE INSTRUCTION IS ADDED TO THE X INDEX, DISCARDING THE CARRY, THE RESULTS POINTS TO A LOCATION ON PAGE ZERO WHICH CONTAINS THE 8 LOW ORDER BITS OF THE EA. THE NEXT BYTE CONTAINS THE 8 HIGH ORDER BITS.

(IND), Y - INDIRECT INDEXED — THE SECOND BYTE OF THE INSTRUCTION POINTS TO A LOCATION IN PAGE ZERO. THE CONTENTS OF THIS MEMORY LOCATION ARE ADDED TO THE Y INDEX, THE RESULT BEING THE LOW ORDER 8 BITS OF THE EA. THE CARRY FROM THIS OPERATION IS ADDED TO THE CONTENTS OF THE NEXT PAGE ZERO LOCATION, THE RESULTS BEING THE 8 HIGH ORDER BITS OF THE EA.

APPENDIX 4

NOTES:

APPENDIX 5

EXPRESSIONS

When an instruction or directive calls for a number in the operand, the number may be given as an "expression," the number used being the value of the expression. An expression is really just a formula.

Expressions are made up of operators and terms. Terms are either numbers or labels which stand for numbers. An expression containing a label term that does not have a numeric value will be flagged as an error.

There are five operators; four are arithmetic, and one is logical.

Addition is signified by the sign	+
Subtraction is signified by the sign	–
Multiplication is signified by	*
Division is signified by	/
Logical AND is signified by	&

Expressions must not contain parentheses.

Expressions are evaluated from left to right.

Examples:

100 * = $90 + 1007	
200 JMP 3 + 2*25*4/5 – 3	These instructions are equivalent.
300 JMP 0097	
400 JMP $0061	
100 LDA #NUM1 + NUM2	NUM1 and NUM2 must be defined somewhere in the program. The instruction loads the Accumulator with the sum of the numbers in the locations NUM1 and NUM2.
600 LDA #LABEL & $00FF	This yields the low order byte of the value
610 STA $CC	of LABEL.
620 LDA #LABEL/256	This yields the high order of byte of the
630 STA $CD	value of LABEL.

NOTES:

APPENDIX 6

DIRECTIVES

. OPT Operand	specifies an option. Operand can be LIST or NOLIST, OBJ or NOOBJ, ERRORS or NOERRORS, EJECT or NOEJECT. (Default options are LIST, OBJECT, ERRORS, and EJECT.)
. TITLE "NAME"	causes NAME to be printed at the top of each page.
. PAGE 'MESSAGE"	produces a blank space in the listing, then causes MESSAGE to be printed after NAME.
. TAB n1,n2,n3	controls the spacing of the fields of Op Code Mnemonic, Operand, and Comment as they appear in the listing.
. BYTE a,b...n	places in successive memory locations the values of the expressions a, b, ..., n (one byte for each value).
. BYTE "AB...N"	places in successive memory locations the ATASCII values of A, B, ..., N.
. DBYTE a, b,...n	places in successive pairs of memory locations the values of the expressions a, b, ..., n (two bytes for each value, high byte first).
. WORD a, b,...,n	places in successive pairs of memory locations the values of the expressions a, b, ..., n (two bytes for each value, low byte first).
AB = Expression	makes the Label AB equal to the value of the expression (up to FFFF hex).
* = Expression	makes the Program Counter equal to the value of the expression (up to FFFF hex).
. IF Expression . LABEL	 assembles following code, up to . LABEL, if and only if expression evaluates to zero.
. END	indicates the end of the program to be assembled.

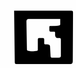
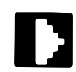
NOTES:

APPENDIX 7

ATASCII CHARACTER SET AND HEXADECIMAL TO DECIMAL CONVERSION

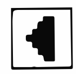
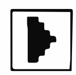
DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
0	0	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
10	A	
11	B	
12	C	
13	D	
14	E	
15	F	
16	10	
17	11	
18	12	
19	13	
20	14	
21	15	
22	16	
23	17	
24	18	
25	19	
26	1A	
27	1B	
28	1C	
29	1D	
30	1E	
31	1F	
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
39	27	'	55	37	7	71	47	G
40	28	(	56	38	8	72	48	H
41	29	)	57	39	9	73	49	I
42	2A	*	58	3A	:	74	4A	J
43	2B	+	59	3B	;	75	4B	K
44	2C	,	60	3C	<	76	4C	L
45	2D	-	61	3D	=	77	4D	M
46	2E	.	62	3E	>	78	4E	N
47	2F	/	63	3F	?	79	4F	O
48	30	0	64	40	@	80	50	P
49	31	1	65	41	A	81	51	Q
50	32	2	66	42	B	82	52	R
51	33	3	67	43	C	83	53	S
52	34	4	68	44	D	84	54	T
53	35	5	69	45	E	85	55	U
54	36	6	70	46	F	86	56	V

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
87	57	W	103	67	g	119	77	w
88	58	X	104	68	h	120	78	x
89	59	Y	105	69	i	121	79	y
90	5A	Z	106	6A	j	122	7A	z
91	5B	[	107	6B	k	123	7B	
92	5C	\	108	6C	l	124	7C	|
93	5D	]	109	6D	m	125	7D	
94	5E	^	110	6E	n	126	7E	
95	5F	_	111	6F	o	127	7F	
96	60		112	70	p	128	80	*(inverse characters begin)*
97	61	a	113	71	q	129	81	
98	62	b	114	72	r	130	82	
99	63	c	115	73	s	131	83	
100	64	d	116	74	t	132	84	
101	65	e	117	75	u	133	85	
102	66	f	118	76	v	134	86	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
135	87		151	97		167	A7	
136	88		152	98		168	A8	
137	89		153	99		169	A9	
138	8A		154	9A		170	AA	
139	8B		155	9B	(EOL) RETURN	171	AB	
140	8C		156	9C	↑	172	AC	
141	8D		157	9D	↓	173	AD	
142	8E		158	9E	←	174	AE	
143	8F		159	9F	→	175	AF	
144	90		160	A0		176	B0	
145	91		161	A1		177	B1	
146	92		162	A2		178	B2	
147	93		163	A3		179	B3	
148	94		164	A4		180	B4	
149	95		165	A5		181	B5	
150	96		166	A6		182	B6	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
183	B7		199	C7		215	D7	
184	B8		200	C8		216	D8	
185	B9		201	C9		217	D9	
186	BA		202	CA		218	DA	
187	BB		203	CB		219	DB	
188	BC		204	CC		220	DC	
189	BD		205	CD		221	DD	
190	BE		206	CE		222	DE	
191	BF		207	CF		223	DF	
192	C0		208	D0		224	E0	
193	C1		209	D1		225	E1	
194	C2		210	D2		226	E2	
195	C3		211	D3		227	E3	
196	C4		212	D4		228	E4	
197	C5		213	D5		229	E5	
198	C6		214	D6		230	E6	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
231	E7	
232	E8	
233	E9	
234	EA	
235	EB	
236	EC	
237	ED	
238	EE	
239	EF	
240	F0	
241	F1	
242	F2	
243	F3	
244	F4	
245	F5	
246	F6	
247	F7	
248	F8	
249	F9	
250	FA	
251	FB	
252	FC	
253	FD	 *(Buzzer)*
254	FE	 *(Delete character)*
255	FF	 *(Insert character)*

Notes:

1. ATASCII stands for ATARI ASCII. Letters and numbers have the same values as those in ASCII, but some of the special characters are different.
2. Except as shown, characters from 128-255 are reverse colors of 1 to 127.
3. Add 32 to upper case code to get lower case code for same letter.
4. To get ATASCII code, tell computer (direct mode) to PRINT ASC ("______") Fill blank with letter, character, or number of code. Must use the quotes!

APPENDIX 8

REFERENCES

ATARI PUBLICATIONS

Obtainable from your ATARI dealer, or ATARI Consumer Division, Customer Support, 1195 Borregas Avenue, Sunnyvale, CA 94086

ATARI 400™ Operator's Manual	CO14768
ATARI 800™ Operator's Manual	CO14769
ATARI 810™ Operator's Manual	CO14760
ATARI 815™ Operator's Manual	CO16377
ATARI Disk Operating System II Reference Manual	
ATARI 410™ Operator's Manual	CO14810

OTHER PUBLICATIONS

6502 Programming Manual
SYNERTEK, 3050 Coronado Drive, Santa Clara, CA 95051 or
MOS Technology, 950 Rittenhouse Road, Norristown, PA 19401

6502 Assembly Language Programming by Lance Leventhal
Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, CA 94710

Programming the 6502 by Rodney Zaks
Sybex, 2020 Milvia Street, Berkeley, CA 94704

NOTES:

APPENDIX 9

USING THE ASSEMBLER CARTRIDGE TO BEST ADVANTAGE

The Assembler Editor cartridge is designed to support intermediate-level assembly language software development. It is good enough in this function to be used by ATARI's own programmers for some software development.

The Assembler is powerful and it can do a great deal, but it is not a professional software development system. It is not well suited for development of large assembly language programs. A good rule of thumb is: take the amount of RAM you have in your system and divide by ten to find the largest amount of object code you can comfortably develop with this cartridge. Thus, an ATARI Personal Computer System with 16K of RAM can be used to develop object code programs up to about 1600 bytes in size. Of course, you can stretch your memory by eliminating all explanatory comments and using very short labels. This will allow you to fit in much more code, but it will make your program difficult to revise at a later time.

Our recommendation is that this cartridge is best used to develop machine language subroutines that enhance the speed and power of BASIC programs. Assembly language complements BASIC very well; the combination of BASIC and machine language is extremely powerful. You can unleash almost all of the power inside your ATARI Personal Computer System with this combination. You should use BASIC for most of your programming; it is easy to write and debug. You should use assembly language only when necessary. There are five applications of machine language that are particularly appropriate:

- To provide certain special logical operations not readily available from BASIC
- To generate special sound effects that BASIC is too slow to generate
- To generate high-speed graphics and animation
- To utilize the interrupt capabilities of the machine
- To accomplish high-speed complex logical calculations and manipulations

Most of these applications are situations that call for high speed; in general, the primary advantage of machine language is its higher speed. Machine language averages about ten times faster than BASIC and in special cases, can be up to a thousand times faster. We do not recommend using machine language for floating point arithmetic or for I/O to and from peripherals (except the screen). In general, one should use machine language only when its speed advantages outweigh the difficulties of programming in assembly language.

Extensive use of assembly language requires a thorough knowledge of the layout and operating system of the host machine. Unfortunately, the ATARI Personal Computer System is far too complex to cover adequately in a brief appendix. We therefore provide four commented sample programs which show how to execute some of the most commonly used functions. These programs are meant only for demonstration purposes; they certainly do not exercise the full power of the machine. You may wish to enhance them, adding whatever features you desire. In this way you will learn more about the ATARI Personal Computer System.

All four programs have been written to reside on page 6 of memory. These 256 bytes have been reserved for your use. On page zero, only 7 bytes have been reserved for your use by the BASIC cartridge; they are locations $CB through $D1 (203 through 209). Locations $D4 and $D5 (212 and 213) are also usable; they are used to return parameters from machine language routines to BASIC through the USR function. Furthermore, locations $D6 through $F1 are used only by the floating point package; you may use them from BASIC USR calls if you do not mind having them altered every time an arithmetic operation is performed. If your program runs only with the Assembler Editor cartridge and not the BASIC cartridge you may use zero page locations $B0 through $CF. You will have to be very sparing in your use of page zero locations, as nine safe locations will not take you far. It is not wise to usurp other locations on page zero, as they are used by the operating system and BASIC; there is no way to know if you thereby sabotage some vital function and crash the system until it is too late. For the moment, we recommend that you limit yourself to programs that fit onto page 6 and use less than 9 bytes of page zero. The four sample programs meet that restriction; later we will show you how to make larger programs with BASIC.

Our first sample program is very simple: it takes two 16-bit numbers, exclusive OR's them together, and returns the resulting 16-bit number to BASIC. It is only 17 bytes long and uses only 4 bytes of page zero. We will use it as a vehicle to show you the rudiments of interfacing machine language to BASIC. Here's how: First, type in the program with the Assembler Editor cartridge in place. Make sure that you have typed it in properly by assembling the program (the command ASM) and verifying that no errors are flagged to you. If it squawks unpleasantly, you have offended its delicate sensitivities; note the line number where the error occurred (CONTROL-1 is a handy way to stop the listing so you can see what happened). Then list the offending line and correct the typo. You may rest assured that the program as we list it will indeed assemble without errors; if you type it in exactly as listed it will work fine. Now assemble the program with the object code going to your nonvolatile storage medium (either diskette or cassette). If you have a disk drive, type in:

```
ASM,,#D:EXCLOR.OBJ
```

If you have a Program Recorder, type in:

```
ASM,,#C:
```

Follow normal procedures for using these devices. After the object code is stored to your diskette or cassette, open the cartridge slot cover and replace the Assembler Editor cartridge with the BASIC cartridge. Close the cover and when you see the READY prompt, load the program from diskette or cassette tape into RAM.

If you have a diskette, type DOS to call DOS, then type L to load a binary file. When it asks what file to load, respond with:

EXCLOR.OBJ RETURN

When it returns the SELECT ITEM prompt, type B RETURN to return to BASIC. If you have a cassette, type in CLOAD and follow the normal procedures for loading from cassette tape. When the machine language program is fully loaded and you are back in BASIC's READY mode, you are ready to use your program. Your program begins at address $0600, or 1536 decimal. Confirm this by the command:

?PEEK(1536) RETURN

The computer should respond with the value 104, which, if you care to cipher it out, is the opcode for the PLA instruction at the beginning of the program. If it doesn't, you blew it; start backtracking to figure out where you went wrong. If it comes up correct, then try this command:

A = USR(1536, 1, 3): ?A RETURN

The computer should respond by printing the value 2, because 1 exclusive OR'ed with 3 equals 2. If you are not familiar with the exclusive OR operator, look it up in any beginning book on assembly language programming. You now have a new function to use. The first parameter is the address of the machine language routine. The second and third parameters are the two numbers to be exclusive OR'ed together. They must be integers between 0 and 65535.

Our second sample program generates notes with controllable attack and decay properties. You may have toyed with the SOUND command in BASIC; if so, perhaps you have noticed that the sounds you can produce with BASIC are somewhat primitive. With assembly language it is easier to produce higher quality sounds. With this routine you can come much closer to the ideal by directly specifying the attack and decay characteristics of each note. It only controls one sound channel, and its algorithm is very simple, so there is plenty of opportunity for improvement. Four parameters are used: the frequency, the attack time, the peak plateau time, and the decay time. All three times are specified in units of 1.6 milliseconds. Using the same procedure as before, enter the program with the Assembler Editor cartridge, assemble it to the diskette or cassette, save it, switch to BASIC, and load the machine language code. Then run the program with:

A = USR(1536, 50, 10, 50, 200) RETURN

Make sure the volume on your television set is turned up and you will hear a note with a very short attack, a short plateau, and a long decay. Experiment with different values of the last four parameters to see what effects can be generated with this technique. Of course, do not change the first parameter (1536) or the program will almost surely crash.

Our third sample program is a much longer affair which generates a pleasing animated pattern on the screen. If you study it carefully you will learn a great deal about the display system of your ATARI Personal Computer System. This program only scratches the surface. There is much more to the ATARI display system than is evident here. Follow the same procedure to set up the program as before; you activate the program with:

GR. 19: A = USR(1536) RETURN

There is no termination point in the program; you must press the S RESET key to terminate the program. After you press the S RESET key, the program will still be intact and usable.

The last sample program demonstrates a very useful capability of the ATARI Personal Computer System—the display list interrupt. Perhaps you have been itching to have more than five colors on the screen. With display list interrupts you can have up to 128 colors. Here's the idea behind it: the ATARI display system uses a display list and display memory. The display list is a sequence of instructions that tell the computer what graphics format to use in putting information onto the screen. The display memory is the information going onto the screen. The address of the beginning of the display list can be found in locations 560 and 561 (decimal).The address of the beginning of the display memory can be found in locations 88 and 89 (decimal). Wondrous things can be done by changing the display list; this program demonstrates only one of the capabilities of the display list system. If bit 7 of a display list instruction is set (equal to 1), then the computer will generate a non-maskable interrupt for the 6502 when it encounters that display list instruction.

If we place an interrupt routine which changes the color values in the color registers, the color on the screen will be changed each time a display list interrupt is encountered. This program consists of two parts: an initializing routine which sets up the display list interrupt vectors, sets all of the display list instructions to generate display list interrupts, and lastly, enables the display list interrupts. The second routine actually services the display list interrupts by changing the color value in the color registers every time it is called. This routine is designed to operate in GRAPHICS 5 mode; it will put all 128 colors onto the screen. (Is that enough for you?) To see it in action, follow the familiar procedure for entering, assembling, saving, and loading the program. Then key in the following BASIC immediate instruction:

```
GR. 5: FOR I= 0  TO 3: COLOR I: FOR J=20*I TO 20*I+19: PLOT J, 3:
DRAWTO J, 39: NEXT J: NEXT I:  A=USR(1536)
```

We hope that these four sample programs have given you a clearer idea of how your ATARI Assembler Editor cartridge might be useful. There are some more advanced techniques for getting even more use out of your cartridge. The first problem many programmers encounter arises when they attempt to write a program larger than 256 bytes long. It will no longer fit onto page 6 and the programmer must find a new place to put the program. The problem is made worse by the fact that the Operating System and BASIC use memory all over the address space. The programmer will have a hard time finding a safe place in memory where the machine language routine will not be wiped out by BASIC or the Operating System. There are a number of solutions to this problem; each solution has advantages and disadvantages. We recommend the following approach, which is difficult to understand but is also the most useful and safest route. What we are going to do is store the machine language program inside a BASIC program and then touch it up so that it will run from anywhere in memory.

We begin by writing an assembly language program with the Assembler Editor cartridge. Originate the program near the top of your available memory. For example, if you have 2K of object code and a 16K machine, originate the program at the 12K boundary with the directive '* =$3000'. This leaves 4K of space—2K for your program, 1K for a GRAPHICS mode 0 display, and 1K of extra space for good measure. Now go through the procedure of assembling the object code to diskette or cassette, changing the cartridges, and loading the object code into memory. Calculate the decimal addresses of the beginning and end of your object code. Let us say that your program is 2179 bytes long. It begins at $3000 or 12288 decimal, so the last byte is at 14466. Print PEEK(12288) and PEEK(14466) to verify that these addresses really do contain the first and last bytes of your program. Remember, the computer will print the results in decimal, not hexadecimal, so you will have to convert in your head or with the computer.

Now start writing a BASIC program, begin with:

```
2 DIM E$(2179)
```

Then add this subroutine (which you can delete later):

```
25000 A = 90*J + 1:B = A + 89: IF B > LIMIT THEN B = LIMIT:?"LAST LINE"
25010?J + 5;"E$(";A;",",B,") = ";CHR$(34);
25020 FOR I = A TO B:?"ESC ESC";CHR$(PEEK)I + C));:NEXT I
25030 ?CHR$(34);J = J+1:RETURN
```

Here the ESC ESC symbol means that you press the escape key twice. Now type in the following direct commands:

```
J = 0 RETURN
C = 12287 RETURN
LIMIT = 2179 RETURN
```

The value of C is the address of the byte before the first byte of your program. The value of LIMIT is the length of your object program. Now type GOSUB 25000 RETURN.

The computer will print a BASIC line onto the screen. It will look very strange—all sorts of strange characters inside a string. They are the screen representation of your object code. To make this line part of your BASIC prgram simply move the cursor up to the line and press RETURN. You might reassure yourself that you were successful by entering:

```
LIST 5 RETURN
```

and verifying that line 5 really did go in. Now type GOSUB 25000 RETURN again and another line will be printed. Enter this one the same way as before. Continue this process of printing and entering lines until the entire object program has been encoded inside BASIC statements. You will know you have reached this point when the computer prints "LAST LINE" onto the screen.

There is one possible hitch with this process. If you have a hex code of $22 (decimal value 34) anywhere in your code it will be put onto the screen as a double quotation mark. This will confuse the BASIC interpreter, which will give you a syntax error when you try to enter the line. If this happens, carefully count which byte is the offender and write down the index of the array which should contain the double quotation mark. Then go back and replace the offending quotation mark with a blank space; that will keep the BASIC interpreter happy. Make note of all such occurrences. When you are done entering the characters, type in a few more lines like:

```
30 E$(292, 292) = CHR$(34)
```

This line puts the double quotation mark into the 292nd array element by brute force. It should come immediately after the lines that declare the string. You should have a line similar to this for each instance of the double quotation mark. Make sure that you have counted properly and put the double quotation marks into the right places.

Now your object program is a part of the BASIC program. You can SAVE and LOAD the BASIC program and the object program will be saved and loaded along with it—a great convenience. You can run the object program by running the BASIC program and then executing the command:

A = USR(ADR(E$))

But there is still another possible hitch. The 6502 machine language code is not fully relocatable; any absolute memory references to the program are certain to fail. For example, suppose your program has a jump-to-subroutine (JSR) instruction that refers to a subroutine within the object code. This instruction would tell it to jump to a specific address. Unfortunately, your program has no way of knowing at what specific address that subroutine is stored and thus will almost certainly jump to the wrong address. The problem arises from the fact that BASIC might move your object program almost anywhere in memory.

There are several solutions to this problem. The simplest solution is to write fully relocatable code; that is, code with no JMP's, no JSR's and no data tables enclosed within it. Put all data tables and subroutines onto page 6. If you still need more space, put very large data tables into the BASIC string and point to them indirectly. Replace long JMP's with a bucket brigade of branch instructions. These techniques should allow you to write large machine language programs.

Example 1.

```
                10      ;
                20      ; ROUTINE EXCLOR
                30      ; PERFORMS EXCLUSIVE OR OPERATION ON
                40      ; TWO BYTES PASSED THROUGH THE STACK
                50      ; PASSES RESULTS DIRECTLY THROUGH USR FUNCTION
                60      ;
                70      ;
0000            80                  *=      $0600
00CC            90      TEMPL       =       $CC       TEMPORARY HOLDING LOCATION
00CD            0100    TEMPH       =       $CD       TEMPORARY HOLDING LOCATION
00D4            0110    RESLTL      =       $D4       ADDRESS FOR PASSING RESULTS
00D5            0120    RESLTH      =       $D5       ADDRESS FOR PASSING HIGH RESULT
0600    68      0130    EXCLOR      PLA
0601    68      0140                PLA
0602    85CD    0150                STA     TEMPH     SAVE HIGH BYTE
0604    68      0160                PLA
0605    85CC    0170                STA     TEMPL     SAVE LOW BYTE
0607    68      0180                PLA
0608    45CD    0190                EOR     TEMPH     PERFORM HIGH EXCLUSIVE OR
060A    85D5    0200                STA     RESLTH    STORE RESULT
060C    68      0210                PLA
060D    45CC    0220                EOR     TEMPL     PERFORM LOW EXCLUSIVE OR
060F    85D4    0230                STA     RESLTL    STORE RESULT
0611    60      0240                RTS               WHAT COULD BE SIMPLER?
0612            0250                .END
```

Example 2.

```
                10      ;
                20      ; ROUTINE NOTE
                30      ; GENERATES NOTES WITH CONTROLLABLE ATTACK AND DECAY
                40      ; TIMES
                50      ; CALL FROM BASIC WITH COMMAND:
                60      ; A = USR(1536, F, A, P, D)
```

```
                  70      ; WHERE
                  80      ; F IS THE FREQUENCY
                  90      ; A IS THE ATTACK TIME
                  0100    ; P IS THE PEAK TIME
                  0110    ; D IS THE DECAY TIME
                  0120    ;
                  0130    ; ALL TIMES GIVEN IN UNITS OF 1.6 MILLISECONDS
0000              0140              * =   $0600
D200              0150    AUDF1     =     $D200        AUDIO FREQUENCY REGISTER
D201              0160    AUDC1     =     $D201        AUDIO CONTROL REGISTER
00CC              0170    ATTACK    =     $CC          ATTACK TIME
00CD              0180    PEAK      =     $CD          PEAK OR PLATEAU TIME
00CE              0190    DECAY     =     $CE          DECAY TIME
0600    68        0200    NOTE      PLA
0601    68        0210              PLA
0602    68        0220              PLA
0603    8D00D2    0230              STA   AUDF1        SET FREQUENCY
0606    68        0240              PLA
0607    85CC      0250              STA   ATTACK       SET ATTACK TIME
0608    68        0260              PLA
060A    68        0270              PLA
060B    68        0280              PLA
060C    85CD      0290              STA   PEAK         SET PEAK TIME
060E    68        0300              PLA
060F    68        0310              PLA
0610    85CE      0320              STA   DECAY        SET DECAY TIME
                  0330    ;
                  0340    ; ATTACK LOOP
                  0350    ;
0612    A9A0      0360              LDA   #$A0         START WITH ZERO VOLUME
0614    8D01D2    0370    ATLOOP    STA   AUDC1
0617    A6CC      0380              LDX   ATTACK
0619    204106    0390              JSR   DELAY
061C    18        0400              CLC
061D    6901      0410              ADC   #$01
061F    C9B0      0420              CMP   #$BØ
0621    D0F1      0430              BNE   ATLOOP
                  0440    ;
                  0450    ; PEAK LOOP
                  0460    ;
0623    A90E      0470              LDA   #$ØE
0625    A6CD      0480    PKLOOP    LDX   PEAK
0627    204106    0490              JSR   DELAY
062A    38        0500              SEC
062B    E901      0510              SBC   #$01
062D    D0F6      0520              BNE   PKLOOP
                  0530    ;
                  0540    ; DECAY LOOP
                  0550    ;
062F    A9AF      0560              LDA   #$AF
0631    8D01D2    0570    DCLOOP    STA   AUDC1
0634    A6CE      0580              LDX   DECAY
0636    204106    0590              JSR   DELAY
```

```
0639    38        0600              SEC
063A    E901      0610              SBC     #$01
063C    C99F      0620              CMP     #$9F
063E    D0F1      0630              BNE     DCLOOP
0640    60        0640              RTS
                  0650    ;
0641    A013      0660    DELAY     LDY     #$13
0643    88        0670    DELAY2    DEY
0644    D0FD      0680              BNE     DELAY2
0646    CA        0690              DEX
0647    D0F8      0700              BNE     DELAY
0649    60        0710              RTS
064A              0720              .END
```

Example 3.

```
                  10      ;
                  20      ;
                  30      ; ROUTINE SPLAY
                  40      ; PUTS A PRETTY DISPLAY ONTO THE SCREEN
                  50      ; CALL FROM BASIC WITH THE FOLLOWING COMMANDS
                  60      ; GR. 19: A=USR(1536)
                  70      ; EXIT PROGRAM WITH S RESET
                  80      ;
                  90      ;
                  0100              *=      $0600
00CC              0110    TEMP      =       $CC      TEMPORARY LOCATION
00CD              0120    XLOC      =       $CD      HORIZONTAL POSITION OF PIXEL
00CE              0130    YLOC      =       $CE      VERTICAL POSITION OF PIXEL
00CF              0140    DIST      =       $CF      DIST. OF PIXEL FROM SCREEN CENTER
00D0              0150    PHASE     =       $D0      COLOR PHASE
00D1              0160    COLOR     =       $D1      COLOR CHOICE
0058              0170    SAVMSC    =       $58      POINTER TO BEG. OF DISPLAY MEMORY
02C4              0180    COLORØ    =       $02C4    LOCATION OF COLOR REGISTERS
D20A              0190    RANDOM    =       $D20A    HARDWARE RANDOM NUMBER LOCATION
0600    68        0200    SPLAY     PLA              POP A ZERO FROM STACK
0601    85D0      0210              STA     PHASE    STORE IT IN PHASE
0603    AA        0220              TAX              SET COUNTER
                  0230    ;
                  0240    ; THIS IS THE MAIN PROGRAM LOOP
                  0250    ; FIRST WE RANDOMLY CHOOSE THE SCREEN LOC. TO MODIFY
                  0260    ; SCREEN IS 40 PIXELS HORIZONTALLY BY 24 PIXELS VERTICALLY
                  0270    ; WITH 4 HORIZONTALLY ADJACENT PIXELS PER BYTE
                  0280    ; HENCE THERE ARE 10 BYTES PER HORIZONTAL ROW
                  0290    ; AND 24 ROWS FOR A TOTAL OF 240 BYTES
                  0300    ; TO REPRESENT THE SCREEN
                  0310    ;
                  0320    ;
                  0330    ;
0604    AD0AD2    0340    BEGIN     LDA     RANDOM   GET A RANDOM NUMBER
0607    290F      0350              AND     #$0F     MASK OFF LOWER NYBBLE
0609    C90A      0360              CMP     #$0A     MUST BE SMALLER THAN 10
060B    B0F7      0370              BCS     BEGIN    IF NOT, TRY AGAIN
```

```
060D  85CD    0380          STA  XLOC    STORE THE RESULT
060F  38      0390          SEC
0610  E905    0400          SBC  #$05    GET X-DISTANCE FROM CENTER
0612  1005    0410          BPL  XA      IS IT POSITIVE OR NEGATIVE?
0614  49FF    0420          EOR  #$FF    IF NEGATIVE, MAKE IT POSITIVE
0616  18      0430          CLC
0617  6901    0440          ADC  #$01
0619  85CF    0450 XA       STA  DIST    SAVE THE ABSOLUTE VALUE
061B  AD0AD2  0460 TRYAGN   LDA  RANDOM GET ANOTHER RANDOM NUMBER
061E  291F    0470          AND  #$1F    MASK OFF LOWER 5 BITS
0620  C918    0480          CMP  #$18    MUST BE SMALLER THAN 24
0622  B0F7    0490          BCS  TRYAGN (BECAUSE THERE ARE ONLY 24 ROWS)
0624  85CE    0500          STA  YLOC    STORE THE RESULT
0626  38      0510          SEC
0627  E90C    0520          SBC  #$0C    GET Y-DIST FROM CENTER OF SCREEN
0629  1005    0530          BPL  XB      IS IT POSITIVE OR NEGATIVE?
062B  49FF    0540          EOR  #$FF    IF NEGATIVE, MAKE IT POSITIVE
062D  18      0550          CLC
062E  6901    0560          ADC  #$01
              0570 ;
              0580 ; NOW CALCULATE THE COLOR TO PUT INTO THIS POSITION
              0590 ;
0630  18      0600 XB       CLC
0631  65CF    0610          ADC  DIST    TOTAL DIST FROM CENTER OF SCREEN
0633  65D0    0620          ADC  PHASE   COLOR PHASE OFFSET
              0630 ;
              0640 ; BITS 3 AND 4 NOW GIVE THE COLOR TO USE
              0650 ;
0635  291F    0660          AND  #$1F    MASK OUT BITS 5, 6, AND 7
0637  4A      0670          LSR  A
0638  4A      0680          LSR  A
0639  4A      0690          LSR  A       SHIFT OFF BITS 0, 1, AND 2
063A  85D1    0700          STA  COLOR   STORE RIGHT-JUSTIFIED RESULT
              0710 ;
              0720 ; NOW WE MUST DETERMINE WHICH OF THE 4 PIXELS
              0730 ; IN THE BYTE GET THE COLOR
              0740 ;
063C  AD0AD2  0750          LDA  RANDOM
063F  2903    0760          AND  #$03    GET A RANDOM NO. BETWEEN 0 AND 3
0641  A8      0770          TAY          USE IT AS A COUNTER
0642  F007    0780          BEQ  NOSHFT  SKIP AHEAD IF IT IS 0
              0790 ;
              0800 ; SHIFT OVER TWICE FOR EACH STEP IN Y
              0810 ;
0644  06D1    0820 SHFTLP   ASL  COLOR
0646  06D1    0830          ASL  COLOR
0648  88      0840          DEY
0649  D0F9    0850          BNE  SHFTLP
              0860 ;
              0870 ; NOW WE MUST CALCULATE WHERE IN MEMORY TO PUT OUR
              0880 ; SQUARE
064B  A5CE    0890 NOSHFT   LDA  YLOC    GET VERTICAL POSITION
064D  0A      0900          ASL  A       YLOC*2
```

```
064E  85CC    0910           STA   TEMP      SAVE IT FOR A FEW MICROSECONDS
0650  0A      0920           ASL   A
0651  0A      0930           ASL   A         YLOC*8
0652  65CC    0940           ADC   TEMP      ADD IN YLOC*2
              0950  ;
              0960  ; RESULT IN ACCUMULATOR IS YLOC*10
              0970  ; REMEMBER, THERE ARE TEN BYTES PER SCREEN ROW
              0980  ;
0654  65CD    0990           ADC   XLOC
              1000  ;
              1010  ; RESULT IS MEMORY LOCATION OF DESIRED PIXEL GROUP
0656  A8      1020           TAY
0657  A5D1    1030           LDA   COLOR     GET COLOR BYTE
0659  9158    1040           STA   (SAVMSC),Y PUT IT ONTO THE SCREEN
065B  CA      1050           DEX             WE SHALL PUT 254 MORE SQUARES
065C  D0A6    1060           BNE   BEGIN     ONTO THE SCREEN
              1070  ;
              1080  ; END OF MAIN INNER LOOP
              1090  ;
065E  C6D0    1100           DEC   PHASE     STEP COLOR PHASE FOR EXPLOSION
0660  A5D0    1110           LDA   PHASE
0662  291F    1120           AND   #$1F      EVERY 32 PHASE STEPS
0664  D09E    1130           BNE   BEGIN     WE CHANGE COLOR REGISTERS
              1140  ; THIS SECTION USES BITS 5 AND 6 OF PHASE
              1150  ; TO CHOOSE WHICH COLOR REGISTER TO MODIFY
              1160  ;
0666  A5D0    1170           LDA   PHASE
0668  4A      1180           LSR   A
0669  4A      1190           LSR   A
066A  4A      1200           LSR   A
066B  4A      1210           LSR   A
066C  4A      1220           LSR   A
066D  2903    1230           AND   #$03
066F  AA      1240           TAX
              1250  ;
0670  AD0AD2  1260           LDA   RANDOM CHOOSE A RANDOM COLOR
0673  9DC402  1270           STA   COLOR0,X PUT NEW COLOR INTO COLOR REG.
0676  4C0406  1280           JMP   BEGIN     START ALL OVER
0679          1290           .END
```

Example 4.

```
10    ;
20    ; KATHY'S COLOR PALETTE
30    ; PUTS ALL 128 COLORS ONTO THE SCREEN
40    ; CALL FROM BASIC WITH THE FOLLOWING COMMANDS:
50    ; GR. 5
60    ; FORI=0 TO 3: COLOR I: FOR J=20*I TO 20*I+19: PLOT J, 3:
65    ; DRAWTO J, 39: NEXT J: NEXT I
70    ; A=USR(1536)
80    ; BASIC IS STILL USABLE
90    ; EXIT WITH SYSTEM RESET KEY
0100  ;
```

```
                ;
0000                            *=      $0600
00CC            POINTA          =       $CC         POINTER TO DISPLAY LIST
00CE            COLCNT          =       $CE         KEEPS TRACK OF COLOR WE ARE ON
00CF            DECK            =       $CF         BIT 0 KEEPS TRACK OF WHICH DECK
0230            DSLSTL          =       $0230       O. S. DISPLAY LIST ADDRESS
D40E            NMIEN           =       $D40E       NON-MASKABLE INTERRUPT ENABLE
D40F            NMIRES          =       $D40F       NON-MASKABLE INTERRUPT RESET
D40F            NMIST           =       $D40F       NON-MASKABLE INTERRUPT STATUS
0200            VDSLST          =       $0200       DISPLAY LIST INTERRUPT VECTOR
D01A            COLBAK          =       $D01A       BACKGROUND COLOR REGISTER
D016            COLPF0          =       $D016       COLOR REGISTER #0
D017            COLPF1          =       $D017       COLOR REGISTER #1
D018            COLPF2          =       $D018       COLOR REGISTER #2
D40A            WSYNC           =       $D40A       WAIT FOR HORIZONTAL SYNC
0600  68        SETUP           PLA                 CLEAN STACK
                ;
                ; SET UP POINTER ON PAGE ZERO
                ;
0601  AD3002                    LDA     DSLSTL
0604  85CC                      STA     POINTA
0606  AD3102                    LDA     DSLSTL+1
0609  85CD                      STA     POINTA+1
                ;
060B  A007                      LDY     #$07        POINT TO 3RD MODE BYTE
060D  A98A                      LDA     #$8A        NEW MODE BYTE
                ;
                ; STORE 16 DISPLAY LIST INTERRUPT MODE BYTES
                ;
060F  91CC      LOOP 1          STA     (POINTA), Y
0611  C8                        INY
0612  C017                      CPY     #$17
0614  D0F9                      BNE     LOOP1
                ;
                ; SKIP FOUR BLANK LINES
                ;
0616  C8                        INY
0617  C8                        INY
0618  C8                        INY
0619  C8                        INY
                ;
                ; STORE 16 MORE DISPLAY LIST INTERRUPT MODE BYTES
                ;
061A  91CC      LOOP2           STA     (POINTA), Y
061C  C8                        INY
061D  C02B                      CPY     #$2B
061F  D0F9                      BNE     LOOP2
                ;
                ; SET UP DISPLAY LIST INTERRUPT VECTOR
                ;
0621  A950                      LDA     #$50
0623  8D0002                    STA     VDSLST
0626  A906                      LDA     #$06
```

```
0628   8D102            STA   VDSLST+1
                  ;
062B   A900             LDA   #$00
062D   85CE             STA   COLCNT  INITIALIZE COLOR COUNTER
062F   85CF             STA   DECK    INITIALIZE DECK COUNTER
0631   8D0FD4           STA   NMIRES  RESET INTRPT. STATUS REGISTER
0634   AD0FD4   WAIT    LDA   NMIST   GET INTERRUPT STATUS REGISTER
0637   2940             AND   #$40    HAS VERTICAL BLANK OCCURRED?
0639   F0F9             BEQ   WAIT    NO, KEEP CHECKING
063B   AD0ED4           LDA   NMIEN   YES, ENABLE DISPLAY LIST
063E   0980             ORA   #$80
0640   8D0ED4           STA   NMIEN   THIS ENABLES DLI
0643   60               RTS           ALL DONE
                  ;
                  ; DISPLAY LIST INTERRUPT SERVICE ROUTINE
                  ;
0644                    *=    $0650
0650   48       DLISRV  PHA           SAVE ACCUMULATOR
0651   A5CE             LDA   COLCNT  GET CURRENT COLOR
0653   18               CLC
0654   6910             ADC   #$10    NEXT COLOR
0656   85CE             STA   COLCNT  SAVE IT
0658   D013             BNE   OVER    END OF DECK?
                  ;
                  ; END OF DECK, BLACKEN SCREEN
                  ;
065A   8D0AD4           STA   WSYNC   WAIT FOR NEXT SCAN LINE
065D   8D0AD0           STA   COLBAK  BLACKEN ALL REGISTERS
0660   8D16D0           STA   COLPF0
0663   8D17D0           STA   COLPF1
0669   E6CF             STA   COLPF2
066B   68               INC   DECK    NEXT DECK
066C   40               PLA           RESTORE ACCUMULATOR
                        RTI           DONE
                  ;
                  ; PUT OUT NEXT COLOR, WITH FOUR LUMINOSITIES
                  ;
066D   A5CF     OVER    LDA   DECK    UPPER OR LOWER DECK?
066F   2901             AND   #$01    MASK OFF RELEVANT BIT
0671   0A               ASL   A       SHIFT INTO HIGH LUMINOSITY
0672   0A               ASL   A
0673   0A               ASL   A
0674   05CE             ORA   COLCNT  MERGE WITH COLOR NYBBLE
0676   8D0AD4           STA   WSYNC   WAIT FOR NEXT SCAN LINE
0679   8D1AD0           STA   COLBAK  STORE COLOR
067C   6902             ADC   #$02    NEXT HIGHER LUMINOSITY
067E   8D16D0           STA   COLPF0  STORE COLOR
0681   6902             ADC   #$02    NEXT HIGHER LUMINOSITY
0683   8D17D0           STA   COLPF1  STORE COLOR
0686   6902             ADC   #$02    NEXT HIGHER LUMINOSITY
0688   8D18D0           STA   COLPF2  STORE COLOR
068B   68               PLA           RESTORE ACCUMULATOR
068C   40               RTI           DONE
```

APPENDIX 10

QUICK REFERENCE: COMMANDS RECOGNIZED BY THE ASSEMBLER EDITOR

The following list includes all commands and directives recognized by the Assembler Editor cartridge. However, not all options, parameters, or defaults are presented. In most cases only the most useful or interesting version is presented.

EDITOR		**Reference Page No.**
NUMxx, yy	provides auto line numbering starting at xx in increments of yy	15
RENxx, yy	renumbers all statements in increments of yy, starting with xx	16
DELxx, yy	deletes statement numbers xx through yy	15
NEW	wipes out source program	15
FIND/SOUGHT/xx, yy, A	finds and displays all occurrences of the string SOUGHT between xx and yy	16
REP/OLD/NEW/xx, yy, A	replaces all occurrences between lines xx and yy of the string OLD with the string NEW	17
LIST #P:	lists source program to printer	19
PRINT #P:	prints source program on printer	21
ENTER #D: NAME	retrieves source program from diskette	21
SAVE #C:<xxxx, yyyy	saves data in addresses xxxx through yyyy to cassette	22
LOAD #C:	retrieves data from cassette	22

ASSEMBLER

ASM#D: NAME. SRC, #P:, #D: NAME. OBJ	retrieves source file called NAME. SRC on diskette, lists assembly listing to printer, and saves object program to diskette under filename NAME. OBJ	25

DEBUGGER

DR	displays 6502 registers A, X, Y, P, and S.	36
CR<,,x	puts an x into the Y-register.	36
Dxxxx, yyyy	displays contents of addresses xxxx through yyyy	36

Cxxxx<yy	puts yy into address xxxx.	37
Mxxxx<yyyy, zzzz	copies memory block yyyy through zzzz into block starting at xxxx.	38
Vxxxx<yyyy, zzzz	compares memory block yyyy through zzzz with block starting at xxxx, displaying mismatches.	38
Lxxxx	disassembles memory starting at address xxxx.	38
A	activates mini-assembler (no labels, one line at a time).	40
Gxxxx	runs object program at xxxx.	40
Txxxx	trace; displays 6502 registers while running object program at address xxxx at readable speed.	40
Sxxxx	single-steps object program at xxxx, displaying registers.	41
X	return to EDIT mode	41

APPENDIX 11

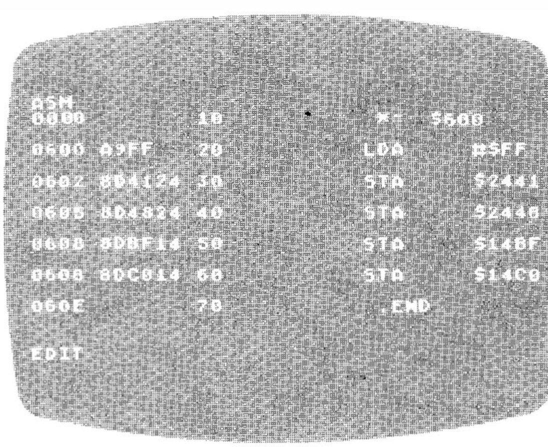
MODIFYING DOS I TO MAKE BINARY HEADERS COMPATIBLE WITH ASSEMBLER EDITOR CARTRIDGE

The following assembly language program modifies four memory locations in DOS I to make binary file headers compatible with the Assembler Editor cartridge. There are two headers (each two bytes long)—one for SAVE and one for LOAD. To change the header bytes from hex 8409 to hex FFFF for full compatibility, type the following modification program.

```
EDIT

10              *=600
20      LDA     #$FF
30      STA     $2441
40      STA     $2448
50      STA     $14BF
60      STA     $14C0
70                END
```

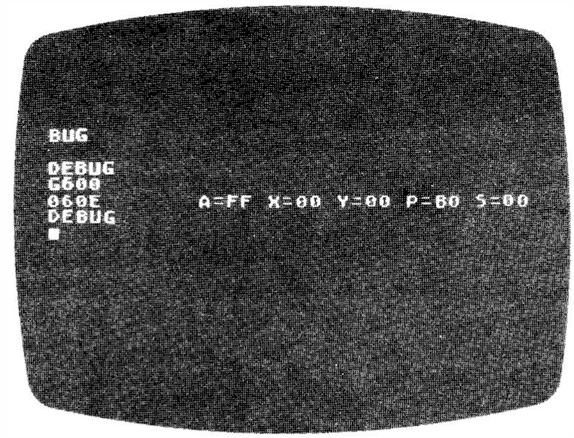
To assemble the modification program, type ASM and press RETURN.



To run this program, you must be in DEBUG mode so, type the following.

- Type BUG and press RETURN.
- Type G600 and press RETURN.

The screen will display:



DOS I will now have header bytes that are fully compatible with the Assembler Editor cartridge.

To change DOS I permanently on your diskette:

1. Run the Modification Program.
2. Type X RETURN to get out of BUG.
3. Type DOS RETURN to enter DOS.
4. Type H RETURN to write a fully compatible DOS on diskette.

CHANGES AND LOCATIONS

LOCATION		**PRESENT CONTENTS**		**CHANGE TO**	
DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX
9281	2441	132	84	255	FF
9288	2448	9	09	255	FF —LOAD
5311	14BF	132	84	255	FF
5312	14CO	9	09	255	FF —SAVE

Instead of using the Modification Program, you could use BASIC to POKE decimal 255 into memory locations 9281, 9288, 5311, and 5312. After making the pokes, type DOS RETURN to display the DOS Menu. Type H RETURN to write the DOS modification onto diskette.

NOTES:

INSTRUCTION SET (OPERATION CODES)

TYPE OF INSTRUCTION	MNEMONIC	OPERATION	TYPE OF ADDRESSING: NON-INDEXED, INDIRECT	NON-INDEXED, DIRECT			INDEXED, DIRECT				INDEXED, INDIRECT		MISCELLANEOUS		CONDITION FLAGS (Affected- ●)							
			Indirect	Immed.	Page 0	Absolute	Abs ,X	Abs. Y	Indexed, X Page 0	Indexed, Y Page 0	Indexed Indirect	Indirect Indexed	Register (A,X,Y,P, PC or SP) Flag (F) Relative (R) Stack (S) None (N)		7	6	5	4	3	2	1	0
		Operand ►	($hhhh) (ABS)	#$hh #BY	$hh BY	$hhhh ABS	$hhhh,X ABS,X	$hhhh,Y BY,Y	$hh,X BY,X	$hh,Y BY,Y	($hh,X) (BY,X)	($hh),Y (BY),Y										
		Bytes ►	3	2	2	3	3	3	2	2	2	2	1	2	N	V		B	D	I	Z	C
LOAD & STORE	LDA	M→A Note 1		A9	A5	AD	BD	B9	B5		A1	B1			●						●	
	STA	A→M			85	8D	9D	99	95		81	91										
	LDX	M→X		A2	A6	AE		BE		B6					●						●	
	STX	X→M			86	8E				96												
	LDY	A→Y		A0	A4	AC	BC		B4						●						●	
	STY	Y→M			84	8C			94													
	Machine cycles			2	3	4	4	4	4	4	6	5										
ARITHMETIC & LOGICAL	AND	A ∧ M→A Note 1		29	25	2D	3D	39	35		21	31			●						●	
	BIT	A ∧ M			24	2C									M7	M6					●	
	CMP	A−M		C9	C5	CD	DD	D9	D5		C1	D1			●						●	4
	CPX	X−M		E0	E4	EC									●						●	4
	CPY	Y−M		C0	C4	CC									●						●	4
	ADC	A+M+C→A Note 1,3		69	65	6D	7D	79	75		61	71			●	●					●	●
	SBC	A−M−C→A Note 1.3		E9	E5	ED	FD	F9	F5		E1	F1			●	●					●	●
	ORA	A∨M→A		09	05	0D	1D	19	15		01	11			●						●	
	EOR	A∀M→A		49	45	4D	5D	59	55		41	51			●						●	
	INC	M+1→M			E6(5)	EE(6)	FE(7)		F6(6)						●						●	
	DEC	M−1→M			C6(5)	CE(6)	DE(7)		D6(6)						●						●	
	INX	X+1→X											E8(X)		●						●	
	DEX	X−1→X											CA(X)		●						●	
	INY	Y+1→Y											C8(Y)		●						●	
	DEY	Y−1→Y											88(Y)		●						●	
	Machine cycles			2	3	4	4	4	4		6	5	2									
SHIFT & ROTATE	ASL	C←[7 0]←0			06	0E	1E		16				0A(A)		●						●	●
	ROL	←[7 0]←[C]←			26	2E	3E		36				2A(A)		●						●	●
	LSR	0→[7 0]→C			46	4E	5E		56				4A(A)		0						●	●
	ROR	→[C]→[7 0]→			66	6E	7E		76				6A(A)		●						●	●
	Machine cycles				5	6	7		6				2									
REGISTER TRANSFER	TAX	A→X											AA(A)		●						●	
	TXA	X→A											8A(X)		●						●	
	TAY	A→Y											A8(A)		●						●	
	TYA	Y→A											98(Y)		●						●	
	TSX	SP→X											BA(SP)		●						●	
	TXS	X→SP											9A(X)									
	Machine cycles												2									
SET & CLEAR FLAGS	CLV	0→V											B8(F)			0						
	CLD	0→D											D8(F)						0			
	SED	1→D											F8(F)						1			
	CLI	0→I											58(F)							0		
	SEI	1→I											78(F)							1		
	CLC	0→C											18(F)									0
	SEC	1→C											38(F)									1
	Machine cycles												2									
BRANCH	BPL	Branch if N=0 Note 2												10(R)								
	BMI	Branch if N=1 Note 2												30(R)								
	BVC	Branch if V=0 Note 2												50(R)								
	BVS	Branch if V=1 Note 2												70(R)								
	BCC	Branch if C=0 Note 2												90(R)								
	BCS	Branch if C=1 Note 2												B0(R)								
	BNE	Branch if Z=0 Note 2												D0(R)								
	BEQ	Branch if Z=1 Note 2												F0(R)								
	JMP	Jump	6C			4C(3)																
	JSR	Jump to Subrout.				20(6)																
	RTS	Return fr. Subrout.												60(S)(6)								
	BRK	Break (Interrupt)												00(P,PC)(7)						1		
	RTI	Return fr. Interrupt												40(S)(6)	From Stack							
	Machine cycles		5											2								
STACK	PHP	P→S;SP−1→SP												08(P)(3)								
	PLP	SP+1→S;S→P												28(S)(4)	From Stack							
	PHA	A→S; SP−1→SP												48(A)(3)								
	PLA	SP+1→SP;S→A												68(S)(4)	●						●	
NO.OP	NOP	No operation												EA(N)(2)								

A — Accumulator, or contents
X, Y, P — Registers X, Y, P, or contents
SP — Stack Pointer, or contents
S — Stack
M — Memory location (effective address), or contents
M7 — Bit 7 of M
∧ V, ∀ — Logical AND, OR, XOR
P→Q — P is copied to Q; P unchanged

NOTES: At the head of each column, under TYPE OF ADDRESSING, the correct way to write an Operand is given, in Hex, where 'h' represents a hex digit, and symbolically, where 'BY' and 'ABS' represent numbers of one and two bytes, respectively The number at the head of each column is the number of bytes of that type of instruction.

The circled number at the foot of a column is the number of machine cycles for the instructions in that block, exceptions are indicated by the circled numbers after the Op Code.

1. If the page boundary is crossed, the number of machine cycles is one more than shown.
2. If the condition is true and the branch is taken, the number of machine cycles is one more than shown when the branch is to the same page and two more than shown when the branch is to a different page.
3. Effects of ADC and SBC may be confusing if the D Flag is set. Check results carefully.
4. C=0 when A or X or Y < M; C=1 when A or X or Y ≥ M.

ATARI®

A Warner Communications Company

C014189-03 REV. 2