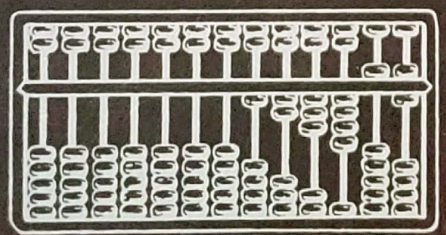
EDIT 6502©



COMPUTER BASED SOFTWARE

LJK T.M.

ENTERPRISES INC.

Saint Louis, Missouri

EDIT 6502

MONITOR COMMANDS

=	HEX RESULT (16 BIT)	@	DECIMAL RESULT
G	GO	L	LIST (DISASSEMBLE)
-	HEX SEARCH	:	HEX STORE
Z	FILL MEMORY	X	RETURN TO CCP
W	WRITE SECTOR	R	READ SECTOR
N	TURN OFF PRINTER	S	STEP
!	DOS	M	MOVE
U	USER	,	PARAMETER SEPARATOR
		?	DISPLAY (HEX AND ASC)
		"	ASC SEARCH
		>	ASC STORE
		#	DISPLAY REGISTERS
		P	TURN ON PRINTER
		T	TRACE
		V	VERIFY
		<	PARAMETER SEPARATOR

LJK ENTERPRISES INC.
P.O. Box 10827
St. Louis, MO 63129
(314) 846-6124

EDIT 6502

REFERENCE CARD

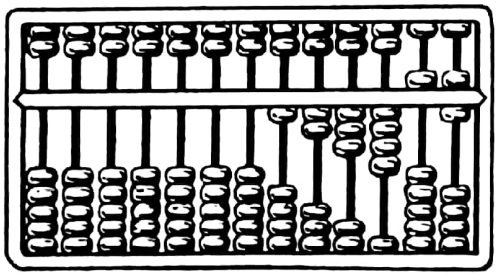
```
LOAD "D:F.E"                MERGE "D:F.E"          SAVE "D:F.E"          BLOAD [A1] "D:F.E"
BRUN [A1] "D:F.E"           DIR [D]                DOS NONE              BSAVE A1,A2 "D:F.E"
XIO A1,A2 "DEV:

NEW NONE                    CLR NONE               LEN NONE              LOM [A1]
HIM [A1]                    SYMB [A1]              DATE NONE             TABS [L1] [,L2] [,L3]
HELP NONE

ADD NONE                    DEL L1 [,L2]           CHNG [L1] [,L2]       COPY L1,L2 TO L3
INS L1                      LIST [L1] [,L2]        TABLE NONE            SRCH [L1] [,L2] "$1"$2"
EDIT L1 [,L2]               FIND [L1] [,L2] "$1"

ASM [PLTSWA"]               MON NONE               DIS A1,A2 "$1"
```

EDIT 6502™



COMPUTER BASED SOFTWARE

LJK T.M.

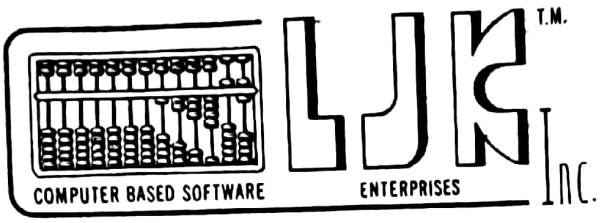
ENTERPRISES INC.

Saint Louis, Missouri

EDIT 6502 ATARI

PROGRAM REGISTRATION FORM

SEND FORM TO:



P.O. Box 10827 St. Louis, Mo. 63129

PROGRAM NUMBER 1-1074 DATE OF PURCHASE_____

STORE NAME_____

ADDRESS_____

CITY, STATE, ZIP_____

PURCHASER_____

ADDRESS_____

CITY, STATE, ZIP_____

PLEASE ANSWER THE FOLLOWING SO THAT WE MAY BETTER ANSWER YOUR QUESTIONS SHOULD YOU HAVE PROBLEMS.

PRINTER TYPE_____

INTERFACE TYPE_____

MODEM (Y/N)_____ATARI 400 OR 800_____

MEMORY SIZE_____

OTHER HARDWARE OR MODIFICATIONS TO YOUR ATARI:_____

_____

_____

_____

_____

_____

(MAIL WITHIN 10 DAYS OF PURCHASE)

THE OPPOSITE SIDE MUST BE SIGNED

(over)

EDIT 6502 ATARI

THIS SIDE MUST BE SIGNED TO REGISTER YOUR PROGRAM

SOFTWARE LICENSE AGREEMENT

LJK Enterprises Inc. (hereafter referred to as LJK) has sold or will sell to the undersigned ("purchaser") magnetic diskettes containing original data and programs expressed and arranged in electronically coded and magnetically stored form. LJK has sold or agreed to sell the diskettes only upon the condition that the purchaser acknowledge and agree to the following. By signing the Agreement, the purchaser so acknowledges and agrees.

1. The purchaser understands and acknowledges that the data and programs recorded on the diskettes are copyrighted original works of authorship and are unique proprietary material of LJK.

2. The purchaser will not copy or reproduce any portion of the data or programs recorded on the diskettes, nor allow others to do so.

3. The purchaser will not create or prepare any derivative materials from the data and programs recorded on the diskettes, nor will the purchaser translate such data and programs into another computer system or create any combination which includes any portion thereof, nor allow others to do so.

4. The purchaser will not sell, transfer, distribute, or display to the public any copy or reproduction of the data or programs recorded on the diskettes or any materials derived therefrom or translation or adaptation thereof to any person or organization, nor allow others to do so.

5. The purchaser acknowledges and agrees that the purchaser will be liable for damages to LJK for any violation of this Agreement.

6. The purchaser acknowledges that LJK makes no warranty or representation whatsoever with respect to the data and programs recorded on the diskettes, and that neither LJK nor any author thereof will be responsible for error or omissions therein or erroneous or inaccurate application of any results from the use thereof.

LJK ENTERPRISES, INC.

PURCHASER_____

DATE_____

E D I T 6 5 0 2 <TM>

ASSEMBLER, EDITOR, SYMBOLIC DISASSEMBLER,
AND MACHINE LANGUAGE MONITOR FOR THE
ATARI 800 AND ATARI 400

LJK ENTERPRISES, INC.
P. O. BOX 10827
ST. LOUIS, MO. 63129
(314) 846-6124

VERSION 1.0

E D I T 6 5 0 2

AUTHOR'S PREFACE

EDITORIAL -- ELEMENTS OF GOOD PROGRAMMING

When asked to lecture on assembly language programming two of the most often asked questions I get is "What are some good techniques in programming the 6502?" and "Most assembler manuals expect you to know too much before you start, why don't they have tutorials?" In response to the second question, I feel that there are a number of books in the marketplace that explain, in great detail, what is required for good assembly language programming. The problem in the beginner getting the most out of it leads back to the first question, where my overwhelming response is to be a good programmer the first requirement is to **KNOW THE INSTRUCTION SET OF THE PROCESSOR YOU ARE WORKING WITH.** And by knowing the instruction set, I mean, knowing exactly what each instruction does for you. I can pick up almost any magazine and see articles with assembly listings that demonstrate my point (and these people must have some proficiency or else they wouldn't be published). A typical example might be:

```
LDA  FLAG
CMP  #0
BEQ  OK
```

This code will always work, but if the programmer completely understood the instruction set of the 6502 he would **KNOW** that the LDA FLAG instruction did, indeed, either set or clear the 0 flag in the 6502's status register and that the CMP #0 instruction is redundant. This may seem a trivial matter to some of you; but, compounding this type of thing several times in code and not taking advantages of other 'features' of the

microprocessor by not understanding exactly what happens is what generally is the cause of good, concise code to be turned into slow and long code. I could demonstrate countless examples just like the one above on this subject.

With the memory size of systems increasing this point **DOES NOT GO AWAY!** The philosophy should be the majority of the memory (as much of it as possible) should go to the user for **USER DATA** rather than to expand the program. There are only two viable exceptions to this, those being 1 - when there is no data and 2 - when shrinking the program costs the user something. The program is written for the **USER**, not the programmer.

Most of the code I see is not geared this way. It generally takes more time and organization to write user oriented code, but that is what it is all about. We call it designing a good user interface. Others call it human engineering. To me, it is a good practice to write code, even that is just for yourself, to be geared toward someone who is not familiar with the machine.

To observe this phenomenon, just take any program that you've written and place someone in front of the machine who has no knowledge of the machine, tell then to run the program and **WALK AWAY FROM THE MACHINE.** If the program is designed with the user interface foremost in mind, they should be able to <u>run the program without assistance</u>! This is a very hard thing for the programmer to do. He or she will want to help them along to warn them of pitfalls in the system; but that is what the test is all about.

I highly recommend that people flowchart their programs and spend a great deal of time in organization of the program before they ever sit down in front of a machine. This organizing time should be though of in terms of how the user will be perceiving and using the program and the possible things that they might do that the program never intended them to do (this is the hard part, you can never think of all the ways). Nothing is more frustrating to the user than a program that crashes for no reason that they can discern. In this organizational time methods of actual coding should be thought out so that by the time you sit down at the machine to write the code, it is merely an exercise in typing.

All of the aforementioned material deals with programming in general. To get specific to the 6502 and Atari, one of the key things to understand is that the status register controls all. All of your decisions are based on it. Pay particularly close attention as to how each instruction may affect the status. Lay out your entire memory map of the system before you start out coding including temporaries, zero page allocation and user data space.

You may notice that this particular program is less user oriented than other LJK programs which may seem a contradiction in what we have been discussing. It is not, however. As this program is geared toward a **USER** who would find menus and the like a retardant doing things in an efficient manner. The program is geared to a more knowledgeable user.

FOR THE BEGINNER

Most beginners are familiar with another programming language such as basic. The key thing they seem to forget is that they did not learn that language overnight. Remember the first day you brought your Atari home and sat down and said "What does PRINT mean?" Computers are excellent educational tools in that you can try something on the machine without worrying about destroying the hardware. If you totally crash the system you can just turn off the machine and reboot it. Just as you did when first learning, write small pieces of code and **TRY THEM.** The logic of assembly language is very similar to other programming languages, it is only a matter of learning new nomenclature and, in most cases, learning that in assembly language, you must provide the memory allocation. My recommendation on early routines is to write something that will **VISUALLY** show you its results. It is one of the most rewarding ways to experiment and learn.

FOR THE INTERMEDIATE PROGRAMMER

The best way to learn new techniques is to steal them. Before I get in trouble, let me say that you shouldn't steal peoples code, but examine the techniques they have used to perform tasks. There is nothing wrong with taking any piece of source listing that you can find and read through and understand exactly what the author was doing. This is an excellent way of finding new and better ways of performing tasks. Store any new technique you may see a way for that future time when it just happens to fit the bill.

E D I T 6 5 0 2

FOR THE ADVANCED PROGRAMMER

Before you read this, you should read the other two sections! The key thing to being a good programmer is to always look to improve. Ego can oftentimes get in the way of this. The very best of programmers don't claim to know it all and are always looking for new and better ways of doing things. Another key point is to not come in looking at things with a preconceived notion. The single biggest stifler of creativity is preconceived notions. Let your imagination run with ideas and keep it free of constraints that you <u>think</u> might be there.

Ken Leonhardi
LJK Enterprises, Inc.

TABLE OF CONTENTS

E D I T 6 5 0 2

KEYBOARD FUNCTIONS

INPUT CONTROLS

^A	Exit entry of line at current cursor position.
^B	Go to the beginning of the line.
^E	Go to the end of the line.
^F	Find character after cursor location.
<-	Move the cursor back 1 space (non-destructive backspace).
->	Advance the cursor by copying over the character from the screen and adding it to the input buffer.
|	Move the cursor up. This will not change your position in the input buffer, but will only move the cursor up.
|	Move the cursor down. Again, this will not change your position in the input buffer, but will only move the cursor down.
CLEAR	Clear entire screen
INSERT	Insert one character (space) at cursor location.
^DELETE	Delete character at cursor location.
sDELETE	Cancel entire input line and start over. The input buffer will be cleared and you will be placed on a new line (with prompting) for new input.
BACK S	Delete previous character typed into input buffer.
TAB SET	Clear screen from cursor location to end of line.
TAB CLR	Clear screen from cursor location to end of screen.
TAB	Tab cursor over to next multiple of 8 column. This will copy the screen contents into input buffer over the range of characters it copies
<CR>	Exit entry of input line by accepting entire line.
CAPS LOWR	Set lower case enter mode.
logo key	Toggle input inverse flag off and on.
ESC	Esc allows for entering and viewing any control character into the input buffer.
BREAK	Abort operation and return to outer level command.

NOTE: ^ = CONTROL
s = SHIFT

E D I T 6 5 0 2

COMMANDS SHEET

INPUT/OUTPUT COMMANDS

```
LOAD "DEV:FNAME.EXT"                    MERGE "DEV:FNAME.EXT"
BLOAD [ADR1] "DEV:FNAME.EXT"            SAVE "DEV:FNAME.EXT"
BRUN [ADR1] "DEV:FNAME.EXT"             DIR [D]
BSAVE ADR1,ADR2 "DEV:FNAME.EXT"         DOS NONE
XIO ADR1,ADR2 "DEV:
```

SYSTEM MANAGEMENT

```
NEW NONE                                CLR NONE
LEN NONE                                LOM [ADR1]
HIM [ADR1]                              SYMB [ADR1]
TABS [LNO1][,LNO2][,LNO3]               DATE NONE
HELP NONE
```

EDITTING COMMANDS

```
ADD NONE                                EDIT LNO1 [,LNO2]
INS LNO1                                DEL LNO1 [,LNO2]
FIND [LNO1] [,LNO2] "STRING1"           LIST [LNO1][,LNO2]
CHNG [LNO1][,LNO2] "STRING1"STRING2"    PLIST [LNO1][,LNO2]
SRCH [LNO1][,LNO2] "STRING1"STRING2"    COPY LNO1,LNO2 TO LNO3
TABLE NONE
```

MODE CHANGING

```
ASM [PLTSWA"]                           MON NONE
DIS ADR1,ADR2 "STRING1"
```

E D I T 6 5 0 2

MONITOR COMMANDS

=	HEX RESULT (16 BIT)	@	DECIMAL RESULT
?	DISPLAY (HEX AND ASC)	G	GO
L	LIST (DISASSEMBLE)	"	ASC SEARCH
'	HEX SEARCH	:	HEX STORE
>	ASC STORE	Z	FILL MEMORY
X	RETURN TO CCP	#	DISPLAY REGISTERS
W	WRITE SECTOR	R	READ SECTOR
P	TURN ON PRINTER	N	TURN OFF PRINTER
S	STEP	T	TRACE
!	DOS	M	MOVE
V	VERIFY	U	USER
,	PARAMETER SEPARATOR	<	PARAMETER SEPARATOR

E D I T 6 5 0 2

I. INTRODUCTION

EDIT 6502 consists of a co-resident assembler, symbolic disassembler, text editor, and machine language monitor for the Atari 400 and Atari 800 computers. Assemblies and disassemblies can be linked. You are in no way limited in the length, or memory size of the program you are working on. The machine language monitor is very comprehensive in allowing step, trace, read and write a sector to disk, ASCII dumps of memory, and many other commands. The disassembler allows predefinition of labels and differentiation between 6502 code, ASCII strings, hex data, word data, and stack data. EDIT 6502 allows you to disassemble 6502 machine language programs, edit, assemble, execute, debug, reedit and go on without ever having to reload a program. This package works well for anyone from the neophyte to the professional programmer.

The command structure is set up very similarly to the immediate mode in Basic to allow the beginner to make the transition from basic to assembly language programming very comfortably.

This manual is not intended to be an instruction course on assembly language programming. Novice assembly language programmers are encouraged to consult one of the many books in the marketplace on the subject.

II. LINE INPUTS

Inputting a line of text is different in EDIT 6502 than in standard Atari operation. Additional editing key functions are always at your command. It is possible to go to the end or back to the beginning of a line of the program. Inserting a character, deleting a character, or searching for a character may also be done. You have the option of cancelling a line of the program, going to the end of the line or exiting the line. The additional line input changes are only one of the added editing capabilities of EDIT 6502. You also have the advantage of being able to pause the operation by hitting the space bar. While repeatative hitting of the space bar will cause the program to scroll a line at a time. To abort an operation do a [BREAK]. Following is a list of the useful editing functions that will make your use of EDIT 6502 far easier than any other assembler program available today.

TABLE I

KEYBOARD FUNCTIONS

1. [CTRL A] will end the line input at that point in the input line.
2. [CTRL B] will move the cursor to the beginning of the input line.
3. [CTRL E] will move the cursor to the end of the input line.
4. [CTRL F] (char) will find char on a forward (only) search. If the character is not found, the cursor will go to the end of the input line. The search can be called recursively (going over and over forward for the same character.
5. [<-] will backspace the cursor one position on the input line.
6. [->] will copy one character from the video screen, place it in the input buffer, and advance the cursor. If in lower case mode, the character copied over will be converted to lower case if it is alpha.
7. [|] will move the cursor up one line. This will not change

your position in the input buffer, but will only move the cursor up one line.

8. [↓] will move the cursor down one line. Again, this will not change your position in the input buffer, but will only move the cursor down one line.
9. [INSERT] will insert one space at the cursor location on the screen that can later be replaced with another character.
10. [CTRL DELETE] will delete one character from the input line at the current cursor location.
11. [SHIFT DELETE] will delete the entire line and provide a new line for input.
12. [BACK S] delete previous character typed into the input buffer.
13. [CR] will end the line of input at the current end of line. All input after the cursor that is on the current line will be accepted.
14. [TAB] will act as a tab to the next multiple of 8 column location. If there are characters on the screen, they will be copied over with a screen read.
15. [TAB SET] Clear screen from cursor location to the end of the line.
16. [TAB CLR] Clear screen from cursor location to the end of the screen.
17. [BREAK] will return to outer level command (break).
18. [logo key] will toggle on and off an input inverse flag. This will allow the most significant bit of the character to be set if the flag is set. This will allow the input of low bit ASCII directly into the text file. Please notice that this is not true inverse video basis a 40 column Atari in that inverse on the Atari is actually the control representation on Alpha characters. Under Edit 6502, all inverse characters will appear as inverse video.
19. [CAPS LOWR] will set the Atari into upper/lower case input.
20. [ESC] will allow the entry and viewing of any into the input buffer.

III. CONTROL OF OUTPUT

When text is being output (either to the video screen or the printer), the keyboard is being polled for a keystroke. If a key is pressed, the output is paused until another key is pressed. If the key is the space bar, the output will stop at the end of the next line, just as if a key had been pressed. In this way, listings a line at a time by hitting the space bar every time a new line is desired. Pressing [BREAK] during output will return you to the outer command level. There are places where no output is going on (as in pass 1 of an assembly). In these cases, the keyboard is polled for the [BREAK] only.

IV. INSTALLING YOUR EDIT 6502 CARTRIDGE

The first thing you must do in order to install your EDIT 6502 cartridge is to open the compartment where they are placed. Pull the lever that holds the lid down, this will release the lid. If you have an Atari 800 you will see two cartridge wells. IF you have an Atari 400 you will see only one cartridge well. If there are any cartridges in the machine now, take them out. Before handling the EDIT 6502 cartridge, it is wise to touch the metal part of the cartridge well. This will discharge any static electricity you may have on you during handling.

Hold the EDIT 6502 cartridge so that the "<- LEFT" is pointing to the left and the title EDIT 6502 is facing you. Press the cartridge gently into the cartridge well until it will go no further. Close the lid to the cartridge well(s) until it snaps shut. There you have it, your Atari is now ready to run the EDIT 6502 system.

V. STARTING THE SYSTEM

If you will be disassembling, you should boot from the disk provided with the package; otherwise, you can boot from any disk containing the operating system. The disk with the package will load the disassembler in memory from $1D00 to $2100 and will adjust the memory parameters of the system accordingly. Since the disassembler resides where it does, it is not possible to have the disassembler and the serial driver in the system simultaneously.

Upon a successful disk boot (or on power up if you have no disk), you will be prompted to input the current date. It is important to enter this for **YOUR** reference as to when you were working on a particular file. The format of the input is not a concern to the system as long as you enter something, the program will not allow you to just hit return. The date will be truncated off after 16 characters if you were to enter more than that. If you accidentally enter the wrong date, it can be changed from the command control processor at any time.

VI. THE COMMAND CONTROL PROCESSOR

The Command Control Processor is very similar in structure to the immediate mode in Basic. You will be prompted with a "C" (for Command) followed by a ">". The syntax of the Command form at this point is such that it may or may not be followed by parameters. The use of incorrect commands or parameters will be signaled with error messages. There are 32 EDIT 6502 commands in addition to 22 monitor commands. Commands are broken down into four different types, which are: Input/Output, System management, Editing, and Mode changing. Before going through each of the commands and command types, some syntactical ground rules for the nomenclature used in this manual to represent them is necessary:

1. The syntax for expressing commands in a line is represented as COMMAND parameter, COMMAND [parameter], or COMMAND none. When the word "parameter" is expressed without brackets, it is a required parameter. When it is expressed with brackets it is an optional parameter that may be entered but is not required. The required parameter of NONE indicates that no parameter may be entered.
2. There can be any number of spaces between the COMMAND and the parameter as they are ignored by the system. The spaces may be placed according to your discretion in terms of readability.
3. All commands must be entered in upper case with no spaces between any of the letters in the command.

TABLE IV
PARAMETER SYNTAX

- **ADR1, ADR2** - Hexadecimal addresses (no $ required). Input may be in decimal if the number is preceded by a period (.) or binary if the number is preceded by a percentage symbol (%).

- **LNO1, LNO2, LNO3** - Decimal number, usually line numbers.

- **DEV:FNAME.EXT** - A valid device specification (as in standard Atari operation) and a filename with an extension. If the device is a cassette or other non-named device, no filename is needed.

- **STRING1, STRING2** - ASCII strings of text.

- **PLTSWA"** - Assembly time options (see Assembly section for details).

- " (delimiter) - delimiting character. Can usually be any character other than space. Only when mentioned will it have to be a " or a '.

- **VALUE** - An assembler value. Default radix is decimal, hex supported when preceded by a $, and binary supported when preceded by a %. The program counter may be used by using an asterisk *, and ASCII data can be used by using a " or a '. Non-spaced separation of +-*/ will perform non-precedented math functions. If the value is called for as a zero page value (<256), an automatic mod 256 will be performed on the result. Using the greater than symbol (>) will perform a divide by 256 on the final result.

VII. INPUT/OUTPUT COMMANDS

The syntax and entry order of parameters for DOS type commands follows the syntax of Atari DOS 2.0S.

1. **LOAD** "DEV:FNAME.EXT" - Load the source file FNAME.EXT into memory at the beginning of text area in memory. This command will erase any source file currently in memory.

2. **MERGE** "DEV:FNAME.EXT" - Load a source file from disk, but append it to the end of the file currently in memory.

3. **SAVE** "DEV:FNAME.EXT" - Save the file currently in memory to disk under the name FNAME.EXT.

4. **DIR** [D] - Read the disk directory and display it on the video screen. There will be 2 entries per line.

5. **BLOAD** [ADR1] "DEV:FNAME.EXT" - Load binary (or object) file to memory. If no address is given, the file loads in to memory where it was saved. If the address is given, the file loads at that address.

6. **BRUN** [ADR1] "DEV:FNAME.EXT" - Load and execute a binary file. All parameters of bload are in effect here.

7. **BSAVE** ADR1,ADR2 "DEV:FNAME.EXT" - Save binary file to disk. Starting address is ADR1 and the ending address is ADR2.

8. **DOS** - Load the Atari DOS utilities package.

9. **XIO** ADR1,ADR2 "DEV: - This command will allow you to enter almost any dos command straight from the Atari keyboard without ever having to leave EDIT 6502. ADR1 is the command requested. ADR2 should be entered as a zero unless otherwise requested. The XIO command is very simular to the basic XIO command.

   There are 5 sub-commands used in this command. They are:

   1. 20 or .32 is rename
      XIO 20, 0 "D:FNAME1.EXT,FNAME2.EXT"
   2. 21 or .33 is delete file XIO .33, 0 "D:FNAME.EXT"
   3. 23 or .35 is lock XIO 23,.0 "D:FNAME.EXT"
   4. 24 or .36 is unlock XIO .36,.0 "D:FNAME.EXT"

VIII. SYSTEM MANAGEMENT

The methods of system management are very flexible in allowing the user to be able to relocate source, object, and the symbol table. With this flexibility, comes the responsibility of the user to set the addresses so that there are no conflicts in addresses. Failure to follow this procedure could result in having the source overwrite the symbol table or some other undesirable result.

1. **NEW** NONE - Clears current source file in memory and symbol table. Will ask for verification of OK TO CLEAR (Y/N)? typing Y will clear everything; any other character will take no action.

2. **CLR** NONE - Clears the symbol table.

3. **LEN** NONE - Returns starting address (in hex only) and length of the source file, free space left, and symbol table in memory in both hex and decimal. The starting address is prefixed with a A$. The hex length is prefixed with a $.

4. **LOM** [ADR1] - Resets the pointer for the beginning of text. This command will do an implicit NEW after receiving verification from the user. When no address is specified, the lomem reverts to the default address of $1000 above APMLO is used. If DOS and the disassembler are present, this defaults to $3100. If no DOS is present the value is $1700. If you have the disassembler present and place the value below $2100, the disassembler will no longer be usable. If the user decides to not verify in response to that question, the lomem value will not change.

5. **HIM** [ADR1] - Sets the top of working memory in which the SOURCE file can be manipulated. When no address is given, the value of $A01 bytes below screen memory is used (for the default object to reside above it). When text is buffered (as in an insert, edit, or copy) the text is moved up to from himem down. It is important that you reserve space above himem if you plan to place the object or the symbol table there.

6. **SYMB** [ADR1] - Sets the starting location of the symbol table in memory. This command sequence does an implicit CLR. When no address is given, the symbol table is defaulted to start at the value given in APMLO ($2100 with the disassembler present).

7. **TABS** [LNO1][,LNO2][,LNO3] - Sets the tabs for listing and assembly. Tab field one (LNO1) is the opcode, field two is the operand and field three is the comment field. Any or all parameters can be passed to this command. If the parameters are 0's, the listing will come out unformatted. If no parameters are given, the default values of 10, 15 and 28 are used. Please notice that tab field one actually contains the length of the label field. When editing basic programs, use the command TABS 0,0,0 before listing the file.

8. **DATE** NONE - Changes system date in case of a typographical error upon entering the date originally.

9. **HELP** NONE - displays on the video screen all of the commands available from the command control processor.

IX. EDITING COMMANDS

The format of data entry is essentially free form, however there are some exceptions. The first character of a line should either be a space for no label, an '*', or a ';' for a line that is an entire comment line, or an alpha character for a label. Spaces are used to separate the fields, and you can insert one or more of them between fields, it doesn't matter to the system. Insert only one space between fields with the exception of the placing of 2 spaces between the opcode and operand fields on opcodes with implied operands, such as INX, PHA etc, to insure the good appearance of listing formats. When used with implied operands, the shift opcodes ASL, LSR, ROL and ROR, require the addition of a ; between the opcode and the comment field to tell the assembler that the addressing mode is implied. Labels can be any length, but it is a good idea to keep them under 10 characters for the listing formats to look proper. Table V shows some typical input examples.

ASSEMBLY INPUT EXAMPLES

Note: the underscore (_) has been used in place of a space for reader clarity.

```
1.)  TEST_LDA_FLAG_check_to_see_if_flag_set
2.)  _ASL__;times 2
3.)  TEST
4.)  _LDY_#STRING1-STRING-1 print_string_in_inverse
5.)  LOOP_LDA_TABLE,X_get_next_value
6.)  _CMP_(CHAR),Y
7.)  _DSS UP,LEFT,RIGHT,RTS1,DOWN cursor move_stack
8.)  CH EQU $24_cursor horizontal
9.)  TEST_INP_"MEMORY IN K -> "
```

1. **ADD** NONE - Appends source file in memory from keyboard entry. EDIT 6502 will keep accepting input until a CR (carriage return) is entered as the first character of the line. The system will continually prompt with line numbers for each line. If a line is aborted (SHIFT DELETE), the prompt will now be 'A>'.

2. **EDIT** [LNO1][,LNO2] - Edits one or more lines of text. Each line will be displayed with the cursor pointing at the first character past the line number. To abort the edit mode without altering that line, press CTRL-A at that point. Alternate prompt in this mode is 'E>'.

3. **INS** [LNO1] - Inserts text in source file before LNO1. System will keep accepting input up until a CR is entered as the first character of the line. User will be prompted with line numbers. All text, after the insertions, will have their line numbers raised accordingly. Alternate prompt in this mode is 'I>'.

4. **DEL** [LNO1][,LNO2] - Deletes from the source file either a single line (LNO1) or all lines inclusive from LNO1 to LNO2. If no line numbers are given, all lines of text will be deleted.

5. **FIND** [LNO1][,LNO2] "STRING1" - Allows you to find all occurrences of the ASCII string STRING1 in the current source file from the ranges LNO1 to LNO2. If no line number is given, FIND will search the entire file.

6. **CHNG** [LNO1][,LNO2] "STRING1"STRING2" - Changes all occurrences of STRING1 to STRING2 in the range of line numbers given. Note the requirement of only three delimiters for the string separations.

7. **SRCH** [LNO1][,LNO2] "STRING1"STRING2" - Similar to change except that the user is prompted when each occurrence of STRING1 is found. To change STRING1 into STRING2 press any alpha character; to not change, press any control character. ESC is especially handy for this. This command is a SELECTIVE change.

8. **LIST** [LNO1][,LNO2] - Lists source file on the video screen in the ranges of LNO1 to LNO2. Listing will be formatted according to tab settings. If no line numbers are given, the entire file will be listed. If only one number is given, only that line will be listed.

9. **PLIST** [LNO1][,LNO2]- Same as list, but the output is sent to the printer.

10. **COPY** LNO1,LNO2 TO LNO3 - Allows for the movement of text. It will copy the text in the range of LNO1 to LNO2 and insert it in the lines preceding LNO3. This function uses the symbol table as a buffer, so it does an implicit CLR. The original text will remain in LNO1 to LNO2, although the line numbers may be changed from the insertion, as well as being copied.

11. **TABLE** NONE - Prints current symbol table in memory to the video screen. 2 entries per line will be printed.

X. MODE CHANGERS

1. **MON** NONE - Enter machine language monitor. See the monitor section for a detail of the commands and their usage. Command prompt for this mode is 'M>'.

2. **DIS** ADR1,ADR2 "STRING1" - Enters 2 pass disassembly mode. ADR1 and ADR2 represent the starting and ending addresses of the entire disassembly for the purposes of determining if labels should have equates applied to them. These addresses do not need to correspond with the actual area in memory that is being disassembled. This is useful if the program has to be loaded in the wrong area of memory. String1 is just a reference header that will be placed in the file. Command prompt for this mode is 'D>'.

3. **ASM** [PLTSWA"] - Assembles file in memory. If links are used in the file, then more than just what is in memory can be assembled. PLTSWA are assembly time toggle switches that can be set at your convenience. The switches and their default settings are:

    P. Printer. Default is off. Listing normally goes to the video screen; using the P option will send the listing to the printer.
    L. Listing. Default is on. Listing normally occurs. Using the L option you turn the listing off. This switch can be changed in the assembler as well.
    T. Table. Default is off. The symbol table is normally not printed. This option allows the symbol table to be printed only on a successful assembly.
    S. Store. Default is on. The object code is normally stored. By toggling this switch, the object file will not be generated unless the " option is used which sends the object to the disk.
    W. Wait. Default is off. This is a wait on errors flag. User will be signaled errors with a message and a bell. If the wait flag is set, he will be prompted to hit a key before resuming the assembly.
    A. Abort. Default is off. This is an abort on errors switch. If set, all assemblies will abort when an assembly time error is encountered.
    ". The other option of " must be used as the last option and is the start of the object file name to

save the object to disk under. No other delimiter to start the name other than the quote (") may be used here.

4. **GO** ADR1 - Executes directly the program in memory starting at address ADR1.

XI. MACHINE LANGUAGE MONITOR

The machine language monitor can be entered by typing MON <CR> from the command control processor. When the monitor is entered, the prompt now becomes 'M>'. The monitor is an outer level command module in that hitting the [BREAK] to abort from in the monitor will return you to the monitor. Entry of commands to the monitor is different from the command control processor in that the parameters (usually addresses) are entered before the command, all commands are single letters and multiple commands can be input on a single line.

All parameters are 16 bit unsigned numbers with wrap-around. On inputs of larger than 65535 decimal, the most significant digit is thrown out. All numbers entered as parameters can be expressions. Mathematical operators supported are: +-*/ for addition, subtraction, multiplication, and division respectively. No precedence is taken with expressions, just simple left to right evaluation. Three parameter entry is 3<2,1. Two parameter entry is 2,1. Single parameter entry is 1. Entry of extra parameters is just ignored. Default radix (number base) is hexadecimal with decimal numbers supported by preceding them with a period (.) and binary numbers supported by preceding them with a percent symbol (%).

For the following discussion, parameters will be discussed as 3<2,1. The syntax of the description will be command letter followed by the number of parameters in brackets and a description of the command. If no parameter is entered, the last used value will be used.

XII. COMMAND SET FOR MONITOR

= <1> Hex result. Prints the result of parameter 1 (expression) as a 16 bit hex number.

@ <1> Decimal result. Prints the result of parameter 1 as a 16 bit decimal number (0-65535).

? <2> Display. Display in both hex and ASCII the memory from parameter 2 up to and including parameter 1. If only parameter is entered, only that memory location will be displayed.

G <1> Go. Jump to routine at parameter 1. If the routine ends in an RTS instruction, control will return to the monitor. The 6502 meta registers are restored to the 6502 registers before going to the routine. If no address is given, the last used PC address will be used.

L <1> List. Disassembly (1 pass) the next 20 instruction and list the disassembly on the current output device.

" <2> ASCII Search. Search from parameter 2 to parameter 1 for the ASCII string following the quote and ending in (but not including) the carriage return at the end of the input line. With this command multiple commands can not be entered on the input line unless they precede the search command.

' <2> Hex Search. Search for the hex string following the single quote. All hex entries are separated by commas (,). Same rules as to multiplicity of commands applies as in an ASCII search.

: <1> Hex store. Change the contents of memory starting at parameter 1 by inserting the bytes entered in the input line following the colon. Again the carriage return is the delimiter and all entries are separated by commas. No commands can be made after the store.

> <1> ASCII store. Store the ASCII string in memory starting at parameter 1. Ending delimiter is the carriage return at the end of the input line.

Z <3> Fill memory. Fill memory with the low 8 bits of parameter 3 in the range of parameter 2 to parameter 1.

X <0> Return to Edit 6502. Returns user to Edit 6502 command control processor.

<0> Display 6502 Meta registers. Displays the 6502 stored registers (psuedo registers that are restored to the real registers every time a go is executed). Registers may be modified with hex or asc store commands without an address parameter being entered. But, to modify the Y register, you must enter the preceding values for A and X.

W <2> Write sector. Writes sector parameter 2 (high byte is sector, low byte is track; ie, to write track $11, sector $F from $800 you type 'F11,800W') from data buffer at parameter 1 to the disk in current slot and drive.

R <2> Read sector. Reads sector parameter 2 into data buffer starting at parameter 1 from the disk in current slot and drive. All sector buffers are 256 ($100) bytes long.

P <0> Printer. Sets the printer as the current output device. All output to the printer will be paginated with a heading, the date, and the page number. This mode will remain in effect until the N command is issued.

N <0> Normal video. Return output device as video screen.

S <1> Step. Execute one instruction at parameter 1 and display the 6502 registers.

T <1> Trace. Continually steps through code. After each instruction the 6502 registers will be displayed. Execution will return to the monitor upon receiving the BRK instruction (00). Outputs can be paused with the space bar and halted (return to the monitor) by the break key ([CTRL C]).

M <3> Move. Move the memory from parameter 2 to parameter 1 and place it in memory starting at parameter 3.

V <3> Verify. Verify that the memory from parameter 2 to parameter 1 is the same as the memory starting at parameter 3. All non-identical locations will be printed with the corresponding data.

U <0-3> User. User extension of monitor commands. This command will jump to the user location $3F8 (where you can place the jump command to your routine). Parameters will be stored (low byte, high byte) as follows: parameter 3 -- $88,$89; parameter 2 -- $84,$85 and parameter 1 -- $86,$87. The user location is not initialized with anything on bootstrap or reset, it is the user's responsibility to fill the location with a jump before using the U command from the monitor.

! <0> Dos. Executes the DOS utilities package.

XIII. ASSEMBLY

The assembly section of the program will set the toggle switches according to the parameters passed and will assemble the source file in memory. It will begin by doing an implicit CLR of the symbol table. EDIT 6502 supports all addressing modes of the 6502 and includes 20 psuedo opcodes (or assembler directives). This facilitates making the writing of assembly language programs easier.

The default number base is decimal. Hexadecimal is supported when the number is prefixed by a '$', and binary is supported by prefixing the number with a '%'. The program counter can be used by using the symbol '*'. ASC characters can be used by prefixing them with a " or a '. The double quote (") sets the high bit, and the single quote (') clears the high bit. All operands can also be made up of combinations of the modes. Non-spaced separation of values by the characters +-*/ will perform the functions of addition, subtraction, multiplication, and division respectively. There is no precedence for operators; the field is read from left to right. For example, if we have the label HOME assigned to $FC58 and our operand looked like #HOME/256+$FB*%10-$100, EDIT 6502 would generate $FC (HOME/256), $1F7 ($FC+$FB), $3EE ($1F7*2) and finally $2EE ($3EE-$100). If the opcode were an LDA, the accumulator would be loaded with $EE in the immediate mode (#).

The structure of addressing modes is basically standard with a couple of exceptions. Namely, the use of the implied mode on a shift opcode (ASL, LSR, ROL and ROR) does not require an A as an operand, but rather a blank field. The greater than symbol (>) will generate the high half of the value for an entire math type

operation. It will perform a divide by 256 of the final value. This is invaluable in loading the page of a software stack. For example, to get the page value of the label HOME, do a LDA #>HOME.

The syntactic structure of the operand field for each addressing mode should look like table VI.

A label can be the only thing present on an input line. In that case, the program counter will be assigned to the label and the assembler will go on to the next line. This can be a very handy feature when you are constantly changing the entry to the label. You can isolate it on a line by itself and not have to edit the label line.

The assembly is done in two passes. The first pass collects all of the labels and places them in the symbol table. The second pass generates the actual code. Some of the assembler directives are accomplished on pass 1, and some are accomplished on pass 2. At the end of pass 1, the symbol table is complete. If you are assembling just to view that, you can abort the assembly [BREAK] at that point and use the TABLE command to view the symbol table.

TABLE VI

OPERANDS FOR ASSEMBLY

1. **IMPLIED** - No operand (should use two spaces before comment for tabbing to remain formatted).
2. **IMMEDIATE** - #VALUE where value can be any combination of labels and numbers as described above.
3. **ZERO-PAGE** - VALUE where value represents a value less than 256. For value to represent a label only, it must have been previously equated to zero-page (earlier in the assembly) to take on the zero-page addressing mode.
4. **ABSOLUTE** - VALUE where value represents a value greater than 255.
5. **ZERO-PAGE INDEXED BY X** - VALUE,X where value is <256.
6. **ZERO-PAGE INDEXED BY Y** - VALUE,Y where value is <256.
7. **ABSOLUTE INDEXED BY X** - VALUE,X where value evaluates to >255.
8. **ABSOLUTE INDEXED BY Y** - VALUE,Y where value evaluates to >255.
9. **INDIRECT PRE INDEXED BY X** - (VALUE,X) value must evaluate to <256 or an error message will occur.
10. **INDIRECT POST INDEXED BY Y** - (VALUE),Y where value must evaluate to <256.
11. **INDIRECT** - (VALUE) the only instruction using this mode is the indirect jump.
12. **RELATIVE** - VALUE where the difference from value to the program counter (*) must be between -128 and +127.
13. **ACCUMULATOR** - NONE. Remember to place two spaces in the listing for formatting tabbing and to include a ; before the comment listing.

XIV. PSUEDO OPCODES

There are 20 psuedo opcodes to EDIT 6502. Nine of them are genuine psuedo opcodes in that they generate code. The other 11 are assembler directives which furnish the assembler information to assist in the easy generation of code. The 20 psuedo opcodes and their syntax are:

1. **EQU** VALUE - Equates the label to the value VALUE. For value to contain a label as part of it's make-up, the label must have been previously defined.

2. **ORG** VALUE - Sets the origin of the assembly. This is where the object code will reside when it is to be run. If no ORG is given, the default value is $A00 below screen memory. On assemblies to disk, only one ORG directive can be given.

3. **OBJ** VALUE - Sets the object location of where the code will be assembled in memory. OBJ and ORG work totally independently for greater flexibility. This is the location where the code will be going right now, even if it is not meant to run there. This allows for resolving conflicts of addresses by not having the origin space available at time of assembly. If no OBJ is given, the default value is $800 below screen memory.

4. **PAG** or **PAGE** none - This directive will generate a form feed to the printer on pass 2 of the assembly.

5. **LST** ON or OFF - This directive will turn the listing option on or off depending on the parameter in the operand field.

6. **INP** ["STRING1"] - This directive, during pass 1 of the assembly, will prompt the user with STRING1 for input to be equated to the corresponding label on the same line. Input radix is defaulted to hex (decimal prefixed with . and binary with %), and no math is allowed on value. This procedure is very convenient when asking during assembly time for the parameters of a conditional assembly. The delimiters on the prompt string must be quotes (").

7. **LNK** "DEV:FNAME.EXT" - This directive allows you to assemble source files that are too large to fit in the Atari's memory at any one time. The original file in memory at the start of assembly will be saved under the name of 'D:TEMP' so be sure to not have a file of value by that name on your disk. Any number of source files can be linked on any one assembly. This directive can only be done with a disk system.

8. **RES** VALUE - Reserve storage for value bytes of memory. The assembler will not place anything in this space, but will skip over it. On assembly to disk, nulls will be used in the area. The maximum value for RES is $FF bytes.

9. **DFB** VALUE [,VALUE] etc. - Define byte. Will take the low order byte of each value and store it in memory. Multiple arguments are supported when separated by a comma.

10. **DFW** VALUE [,VALUE] etc. - Define word. Will store a 16 bit address in standard 6502 low byte, high byte format. Multiple arguments are supported when separated by commas.

11. **DFS** VALUE [,VALUE] etc. - Define a multiple page software stack. This directive is similar to DFW with the exception that 1 is subtracted from each argument for software stacking. Multiple arguments are supported when separated by commas.

12. **DSS** VALUE [,VALUE] etc. - Define a single page software stack. Similar to DFB with the exception that 1 is subtracted from the low byte of the argument. Multiple arguments are supported when separated by commas.

13. **ASC** "STRING" or 'STRING' - Store ASC string in memory. For listing formats to look proper, you should use the double quote as often possible. If single quotes are used, any space in the operand will force the rest of the operand over to the comment field on listing.

14. **DCI** "STRING" or 'STRING' - Same as ASC with the exception that the last character stored has the high order bit inverted.

15. **DTZ** "STRING" or 'STRING' - Define text zero. This is an ASC command with a null byte appended to the end of the string.

16. **INV** "STRING" or 'STRING' - Similar to ASC with the exception that the ASC string is stored in memory in the inverse order of how it is listed in the operand.

17. **TLE** "STRING1" - Place string1 as the page header for assembly output. The header is placed during pass 1 of the assembly; so, if you have more than 1 tle command, the last one will be printed on all pages of the assembly listing. Page headers on printouts will also include the date and the page number.

18. **IFC** VALUE - If condition. If value equates to any non-zero value, the assembly will continue until an ELS directive is encountered. If value equates to 0, then no code will be generated until an els or an end command is reached. Conditional assemblies cannot be nested. So be certain to enter an END directive before issuing another IFC directive.

19. **ELS** - Else condition. If the conditional flag was set to false, it is now set to true and vice versa. If an ELS directive is encountered before an IFC directive, it will turn off code generation up until an IFC or an END is encountered.

20. **END** - End condition. Force code to be generated from now until another IFC is encountered.

XV. DISASSEMBLY

Before ever entering the disassembly mode, we must first determine what portions of the data to be disassembled are code, data, words, and stacks. This is accomplished by using memory dumps and single pass disassembly in the monitor. A straight shot at using all code in a two pass disassembly will generally work, but will usually generate some meaningless labels that may not work if the code is relocated elsewhere when reassembling. Separating ASCII text from hex data is merely a matter of determining that the section in question is indeed data and doing an ASCII dump to determine the portions, if any, that are ASCII data.

The disassembler never clears the text or symbol table areas of memory. In this way, you can disassemble with predefined labels. Long disassemblies that are larger in length than the source area can be accomplished by disassembling a section, returning to the CCP, and typing 'DEL 1, <CR>'. This will clear the text but will leave the symbol table intact. Reentry to the disassembly mode does not clear the symbol table to allow the freedom of disassembling large bodies of code.

to enter the disassembly mode, type DIS ADR1,ADR2 "STRING1". Where ADR1 and ADR2 are delimiting values for where equates are placed in the file. They do not have to correspond to the actual area of memory being disassembled (they generally do unless the code is not in its usual location). The STRING1 is a name identity that you may select for your ease of understanding of what the file is about. You will now be rewarded with a prompt of 'D>'. Commas (,) is used as an address delimiter and spaces are ignored. There are five parameters that can be

passed to the disassembler. They are: L-6502 list (disassembly of standard 6502 code); T - ASCII data (character strings); H - Hex data (DFB opcodes); W - word data (DFW opcodes) and S - Stack data (DFS opcodes). The format of a line looks like:

D>ADDR1,ADDR2 (param),ADDR3 (param),ADDR4 (param) etc.

Where ADDR1, ADDR2, ADDR3 and ADDR4 stand defined on page 9 and (param) stands for L, T, H, W or S. For example, if there was code from $800-$900 that we wished to disassemble and we had already determined that from 800 to 832 was 6502 code, 833 to 858 was ASCII text, 859 to 86A was hex data, 86B to 8C1 was 6502 code, 8C2 to 8F3 was Stacks, and 8F4 to 900 was 6502 code, our entry line would look like this:

D>800,832L,858T,86AH,8C1L,8F3S,900L

The following is a scenario of a long disassembly.

First we did a NEW and cleared the entire text. Then we had the option of entering and assembling (we must get those labels to the table) a file made up of equates for the disassembly file. Then we entered disassembly mode by typing DIS and its options. We disassembled a large portion of code and found that we were nearing the memory limits of the machine. We exited disassembly by pressing the return key as the first character of a line, got back to the editor and saved our first portion of the text. We then did a DEL<CR> to delete all text (but kept our symbol table intact), reentered disassembly mode (DIS) and were able to continue without refilling our text with redundant equ's (that also cause assembly errors). We could then go on our merry way continuing the disassembly where we left off before.

When we hit the return key at the end, we would see two passes of the code flashing before our eyes and the file from 800 to 900 would be disassembled. We would then be shown the length of our file (to determine if memory is getting full) and be returned to the disassembly mode (prompt D>). To exit the disassembly mode at this point, we just press the return key. We can then go back and edit, list or assemble our newly created source file from the disassembly.

Not all disassemblies will reassemble directly. The use of labels in the middle of instructions and data is a common occurrence. For example, the loading of a multiple page software stack is accomplished by:

```
GO      ASL             ;times 2
        TAX             for indexing
        LDA MSTK+1,X    page byte
        PHA
        LDA MSTK,X      low order byte
        PHA
        RTS             to sub via rts
```

The loading of A from MSTK+1,X will almost always generate an assembly error because the data in MSTK will be done on an eight wide basis.

Simple editing of the loading labels can correct this problem before reassembling the text.

Another potential assembly problem of disassembly listing is the use of the BIT opcode ($24 and $2C) or other branch never type instructions to get the program to skip over a load or store for the purposes of saving code. An example of this would be the setting or resetting of a flag depending on conditions met. To illustrate:

```
TEST    CMP EQUAL     the test condition
        BEQ GOOD      passed test!
        LDA #$FF      set flag, failed
        DFB $2C       skip over next LDA
GOOD    LDA #0        signal good test
        STA FLAG
```

In either instance, the value in A was stored in the label FLAG. If the test passed, the value stored was a 0, if it failed, the value was $FF.

To correct a problem like this, again simply editing the text will do it. If the information had been previously known, the disassembly could have supplied the byte before the label GOOD as being hex data as it was generated when disassembling originally.

APPENDIX A.

EDIT 6502 MEMORY MAP

MEMORY AREA	USAGE	USER SPACE
$0-$7F	SYSTEM POINTERS	NO
$80-$B6	EDIT 6502 POINTERS	NO
$B7-$FF	FREE	YES
$100-$1FF	6502 STACK	NO
$200-$47F	SYSTEM POINTERS	NO
$2D9-$2DA	BLOAD ADDRESS	NO
$2DB-$2DC	BLOAD LENGTH	NO
$480-$57F	INPUT BUFFER	YES
$580-APMLO	SYSTEM POINTERS	NO
APMLO-SVMSC	USER SPACE	YES
SVMSC-HIMEM	VIDEO SCREEN	NO
$A000-$BFFF	EDIT 6502	NO
$D000-$FFFF	OPERATING SYSTEM	NO

Zero page locations of particular consequence are: $2C9 text buffered flag, $78 current output stream (0 for video), $64-$67 video screen pointers, $9C-$9D system lomem, $9E-$9F system himem, $8A-$8B symbol table pointers, and $82-$83 end of text pointer.

Some useful entry points for trying out routines are:

```
COUT     EQU  $B86E     output character in A to current
                        output device
RDKEY    EQU  $B915     obtain character w/cursor
                        from current input device
GETLN    EQU  $B7C9     start with cr then prompt
GETLN0   EQU  GETLN+3   with prompt, but no cr
GETLN1   EQU  GETLN+$E  no prompt unless cancelled
BLANK    EQU  $B83F     output a space
CROUT    EQU  $B843     output a carriage return
PRBYTE   EQU  $B85B     hex print byte in A to current
                        output device
PRDEC    EQU  $A62A     output a 5 digit decimal to current
                        output device
CLS      EQU  $B6ED     clear screen and home cursor
PUTLN    EQU  $B95E     print text from A,Y to a CR or NULL
```

APPENDIX B.

EDIT 6502 ERROR MESSAGES

PAST END -- The line number or range of line numbers extended beyond the range of the file.

FORMAT -- Numbers were out of range or the format of an operand was not correct.

SYNTAX -- The command was unrecognized by the system. Either the command is not allowed, or there is a misspelling.

DUP LABEL -- There was more than one occurrence of a label definition for the same label.

INVALID OPCODE -- The opcode was unrecognized as being either 6502 or a psuedo opcode.

LABEL REQUIRED -- Either an equate or input psuedo op has been used without a label associated with them.

BRANCH OUT OF RANGE -- The branch address was not within -127 or +128 of the program counter.

ILLEGAL ADDRESS MODE -- The mode of operation for that particular opcode is not in the 6502 opcode set. This can also occur if a zero page label has not been equated to zero page and an indirect instruction is used.

UNDEFINED LABEL -- A label was referenced that was not defined by an equate or a program location.

OUT OF MEMORY -- The systems memory could not handle the last line entered. If the text had been buffered (as in an insertion or edit) all will be restored with the exception of the last line entered. The Len command may say that there are a few bytes left; but not enough for the entire line.

RANGE -- Either a himem parameter was too high, line numbers went from high to low (e.g. 100,50) or a disk option specified a drive outside the range allowed.

I/O CODE: -- An I/O operation was attempted and an error was received. The number following the message will be a standard Atari error code.

NOT A LOAD FILE -- File trying to be loaded is not an Atari object file.

APPENDIX C.

SOFTWARE STACKING

Software stacking is a term used to emulate the missing jump indirect indexed instruction. How this is accomplished requires the user to understand exactly how the 6502 microprocessor operates an RTS instruction. When an RTS is encountered, the 6502 will pull from its stack two bytes and place them in its program counter low byte and then high byte. Then the program counter is **INCREMENTED BY ONE** and the 6502 fetches the next instruction. By pushing onto the 6502 stack the value minus one of the place one wishes to go in memory, the user can then execute an RTS to jump to the place he wishes to go. These values are usually loaded off of a stack of addresses indexed by one of the index registers (X or Y); thus the term, software stacking.

This stacking can be done in one of two ways. The first way requires all routines that are used in the stack reside on **THE SAME PAGE** of memory. In this way, the stack only needs to contain the low order byte, minus one, of the addresses of the routines to be accessed. The page byte (common to all of these routines) is pushed on the 6502 stack first (so it can be pulled last) and the indexed byte is then pushed on the 6502 stack. The RTS is then executed, and the routine is carried out. In examining this procedure, one must use caution. Scrutinize the results, if the routine to use starts at the beginning of the page (i.e. byte 00). The stack will contain the byte minus 1 or $FF, and when pushed, will actually be on the stack as the end of the page. When pulled off of the stack, by the RTS, the entire word will be incremented and the routine will not execute

as expected but, instead, the program counter will be one page too high. Because of this, the range of starting addresses for single page stacks is from 01 to 00. Note that 00 is the address of the start of the next page, but this will work for the very same reason that the 00 byte of the current page will not.

The second type of software stack allows the routines to reside anywhere in memory. Here the index register is multiplied by 2 because there are 2 bytes per stack entry, and the high order byte, the plus 1 entry, is loaded by an LDA instruction similar to LDA STACK+1,X and pushed on the 6502 stack. Then the low order byte is loaded and pushed before the RTS is executed. The advantage of this are obvious in that the routines can reside anywhere in memory. The disadvantages lie in the fact that the stack must be twice as long for a similar number of routines, and the code to push the stack addresses on the 6502 stack must be longer.

Perhaps the user wonders if the results are worth all the constraints required to do this kind of manipulation. There are several reasons for the answer's being yes. One, the speed of execution is generally much faster than doing several compares and jumps. Two, the amount of code required for such an operation is usually much smaller than the compare/jump technique. And third, single page stacking is particularly effective for a short number of routines where comparisons do not have to be made based on the code being in order. For example, to process control characters with each having their own particular routine, no comparisons have to be made once the character is determined to be a control character before a particular control routine can be accessed! To show what the

code would be, we might write for a single page operation:

```
        CMP  #SP        sp or > is non control
        BCS  NOCTRL     not in the routine
        TAY             only minus values work
        BPL  NOCTRL     skip if plus
        JSR  GOSUB      do indirect sub by char in A
        JMP  NEXT       and continue
NOCTRL  ... ......      whatever we do to non controls
GOSUB   TAY             get character to index
        LDA #>PAGE      push page byte first
        PHA
        LDA STACK-$80,Y get byte from software stack
        PHA
        RTS             execute our routine for each control
```

Now 32 different routines have been accessed without ever having to see which control character we had. As is often the case with 32 routines, the starting addresses of each routine could hardly be expected to fit on one page. To change this routine to a multiple page software stack, merely change the gosub routine to as follows:

```
GOSUB   ASL             ;control *2 will also
                        ;remove msb, as being high
        TAY
        LDA  STACK+1,Y  ;get page byte
        PHA
        LDA  STACK,Y    ;and low order byte
        PHA
        RTS             ;and execute
```