LISA 2.5

A PROFESSIONAL ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM FOR APPLE COMPUTERS

Introduction

What is LISA

LISA (pronounced LI ZA, not LE SA) is an interactive 6502 assembler for the Apple II. Due to its structure, code compression, interaction, and built-in features, LISA is easily the most powerful assembler available for the Apple II. With LISA, machine language programming becomes almost as easy as BASIC. LISA works with you instead of working against you. LISA is a symbolic assembler, the programmer does not have to keep track of addresses as with the built-in ROM mini-assembler. More pseudo opcodes, Sweet 16 mnemonics (which turns your Apple II into a 16-bit machine), more extended mnemonics and more commands which allow the flexible use of DOS 3.2.

LISA also works with the new Apple II PLUS as well as with Apple's Autostart ROM or the language system. If your Apple II has the Lazer MicroSystems Lower Case +Plus installed, you may enter and display the entire 96 upper/lower case ASCII character set and all characters may be entered directly from an unmodified Apple keyboard. Not only that, but should you desire to incorporate lower case input into your assembly language programs Lazer Systems has provided a source listing of the "LISA P2.L" routines (used by LISA) for your convenience.

Requirements

Requirements include at least one drive and 48K bytes of RAM. LISA 2.5 64K requires a language card for proper operation.

Disk Facilities Provided

LISA uses several disk options. You may save LISA text files to disk as either a text or "LISA" type file. "LISA" files are much faster and require less space on the disk, but are incompatible with the rest of the world. Text files may be read in by APPLE Pie or your BASIC programs but are much slower than the "LISA" type files for simple loading and saving. In addition a LISA source file on disk may be appended to the existing file in memory by using the "AP(PEND)" command. During assembly it is possible to "chain" source files in from the disk using the "ICL" pseudo opcode. This allows the user to assemble text files which are much larger than the available memory in the Apple II.

Likewise, by using the "DCM" pseudo opcode, it is possible to save generated code on to the disk, so code files of almost any length may be generated.

How does LISA interface with the Monitor & DOS?

LISA operates under DOS 3.2 for file maintenance and peripheral control. Any DOS command may be executed directly from LISA's command level. Since PR# & IN# are DOS commands, PR# & IN# are available for peripheral control. In addition, control-P is reserved for use with user defined routines. These routines may be printer drivers for use with I/O devices not utilizing an on-board ROM, or use with device drivers using the game I/O jack, or any user defined utility such as slow list, entry into BASIC, etc. LISA uses only standard routines in the Apple Monitor, so LISA will work with both the normal Apple monitor and the Autostart ROM. LISA modifies pointers in DOS 3.2, therefore, when your LISA disk is booted the DOS which is loaded into memory should not be used for BASIC, or TINY PASCAL programs.

LISA saves source files in a special "LISA" format. When you catalog the disk these files will have filetype of "L". When running under an unmodified DOS these files will look like binary files, but they cannot be BLOAD'ed or BRUN'ed. LISA is provided on DOS 3.2 but may be converted to DOS 3.3 using the DOS 3.3 MUFFIN program.

Program syntax

Important Concepts

1) Source format
2) Label field
3) Mnemonic field
    a) Standard mnemonics
    b) Extended mnemonics
    c) SWEET-16 mnemonics
    d) Pseudo opcodes
4) Operand field
5) Comment field

Assembly Language source format
Source statements in LISA are entered in a "free format" fashion. They use the following format:

```
LABEL    MNEMONIC  OPERAND   ;COMMENT
```

Each member of the source statement is called a "field". There is a LABEL field, a MNEMONIC field, an OPERAND field, and a COMMENT field.

These fields may or may not be optional depending upon the context of the statement. These fields must be separated by at least one blank, & interleaving blanks may not appear inside any of the fields. If an upper case alphabetic character appears in column one, then that character defines the beginning of the LABEL field. If column one is blank, then this is a signal to LISA that there will not be a label on the current line. If column one contains a semicolon (";") or an asterisk ("*"), then the rest of the line will be considered a comment and will be ignored. The appearance of any other character in column one constitutes an error and this error will be flagged at edit time (assuming that you're using LISA's built-in editor).

The LABEL field
The label field contains a one to eight character label whose first character begins in column one. If you attempt to place a label in any column except column one LISA will mistake the label for a 6502 mnemonic and will (more than likely) give you a syntax error. Valid characters in labels are the uppercase alphabetics, numerics, and the two special characters period (".") and underline ("-"). While LISA 2.5 will accept certain other characters within a filename, they should be avoided to insure upwards compatibility with upcoming versions of LISA. Lower case alphabetics will be converted to uppercase when processing labels, they may be used if convenient.

Labels are terminated with either a blank or a colon. If a blank terminates the label that a 6502 mnemonic must appear after the label. If a colon terminates the line than the remainder of the line is ignored and the label will appear on the line by itself.

A special type of label, local labels, will be discussed later in this manual.

The MNEMONIC field

This field, delimited by a blank, must contain the three character mnemonic. This may be any of the valid 6502 mnemonics, Sweet-16 mnemonics, or pseudo-opcodes.

6502 mnemonics (55):

```
ADC AND ASL BCC BCS BEQ BIT BMI BNE BPL
BRK BVC BVS CLC CLD CLI CMP CPX CPY DEC
DEX DEY EOR INC INX INY JMP JSR LDA LDX
LDY LSR NOP ORA PHA PHP PLA PLP ROL ROR
RTI RTS SBC SEC SED SEI STA STX STY TAX
TAY TSX TXA TXS TYA
```

Extended mnemonics (5):

```
BTR BFL BGE BLT XOR
```

SWEET-16 mnemonics (26):

```
SET LDR STO LDD STD POP STP ADD SUB PPD
CPR INR DCR RTN BRA BNC BIC BIP BIM BNZ
BM1 BNM BKS RSB BSB BNZ
```

Pseudo opcodes (also known as assembler directives) (33):

```
OBJ ORG EPZ EQU ASC STR HEX LST NLS DCM
ICL END ADR DCI INV BLK DFS PAG PAU BYT
HBY DBY LET TTL NOG GEN PHS DPH .DA .IF
.EL .FI USR
```

LISA mnemonics may be entered in either uppercase or lowercase, LISA will always convert the input mnemonic to upper case. A complete description of these appears in the following sections.

The OPERAND field

The operand field, again delimited by a blank, contains the address expression and any required addressing mode information.

The COMMENT field

Following the operand field comes the optional comment field. The comment field must begin with a semicolon (";") and must be separated from the operand field by at least one blank. The remainder of the line (up to return) will be ignored by LISA. If there is no operand field (e.g. implied or accumulator addressing mode) then the comment field may follow the mnemonic field. Comments may not appear on the same line as the "END", "LST", PAG, PAU and "NLS" pseudo opcodes. As previously mentioned, comments may appear on line by themselves by placing a semicolon or an asterisk in column one.

Addressing modes

1) Address expressions
2) Immediate addressing mode
   a) Standard syntax
   b) Low Order Byte Selection
   c) High Order Byte Selection
   d) Extended Modes
3) Accumulator addressing mode
4) Absolute/Zero page addressing
5) Indexed by X addressing
6) Indexed by Y addressing
7) Relative addressing
8) Implied addressing
9) Indirect, indexed by Y addressing
10) Indexed by X, indirect addressing
11) Indirect addressing
12) Local labels

Address expressions
The operand filed provides two pieces of information to LISA. It provides the addressing mode, which tells the computer how to get the data, and the address expression which tells the computer where the data is coming from. An address expression is simply an integer expression, much like the expressions found in Integer BASIC, whose result is a sixteen-bit unsigned integer in the range 0-65535.

Version 2.5 supports addition, subtraction, multiplication, division, logical-AND, logical-OR, logical-exclusive OR, equality, and inequality.

An address expression can be defined in the following terms:
1) An address expression is defined as a "term" optionally followed by an operator and another address expression.
2) An operator is either "+", "-", "*", "/", "&", "I", “^", "=", or "#".
3) A term is either a label (regular or local), a hex constant, a decimal constant, a binary constant, a character constant or the special symbol"*".
4) Hex constants may be in the range $0000-$FFFF and must begin with the symbol "$".
5) Decimal constants may be in the range 0 - 65535 and may begin with the symbol "!" (the "!" is optional). Note that decimal constants in the range 65536-99999 (i.e. overflow) will not be detected at edit time or assembly time, please be careful! Signed decimal constants (i.e. negative decimal values) must begin with the sequence "!-".
6) Binary constants may be in the range %0000000000000000 - %1111111111111111 and must begin with the special symbol "%".
7) Character constants come in two varieties. If you wish to use the standard ASCII representation (i.e. high order bit of) simply enter the character enclosed by two apostrophes (e.g. 'A'). To use the extended ASCII form (i.e. high order bit on) enclose the character in quotes (e.g. "A").
8) The special symbol "*" can be thought of as a function which returns the address of the beginning of the current source line.

Address expressions may not contain any interleaving blanks.

Example of Address Expressions:

```
LBL+$3
HERE-THERE
*+!10
“Z"+$1
$FF
!10
!-936
LABEL/2*X^$FFFF&$10FF|1
LBL-$FF+!10-%1010011
```

Address expressions are evaluated from RIGHT TO LEFT! This is very similar in operation to the expression analyzer used by the APL programming language. Parenthesis are not allowed.

Examples:

$5+$3 evaluates to $8
$5+$3-$2 " evaluates to $6
$5-$3+$2 " evaluates to $0 ($3+$2 = $5 which is subtracted from $5)

In 99% of the cases, the order of evaluation will not make any difference since address expressions seldom have more than two terms. The only time the right to left evaluation sequence will make a difference is when the address expression contains more that two terms and the subtraction operator is used. From this point on, whenever "<expression>" appears you may substitute any valid address expression. A very special type of address expression is the “zero page address expression". In a nutshell, a zero page address expression is one which results in a value less than or equal to $FF and does not contain any terms greater than $FF.

For example, although $FE+$1 is a valid zero page address expression, $100-$1 is not. This is because the expression contains a term greater than $FF ($100). Also, if during evaluation the expression ever evaluates to a value greater than $FF, the expression will not be a zero page expression. Naturally, if an expression evaluates to a value greater that $FF, even though its terms are all less than $FF, it will not be a zero page expression.

Multiplication, division, logical-AND, logical-inclusive OR, and logical-exclusive OR, equality, and inequality operations are also supported in LISA 2.5 address expressions. The symbols used for these operations are "*", "/", "&", "|", "^", and "#" respectively. Note that the "|" character is obtained by typing esc-1 and is displayed properly only if the user has installed a Lazer MicroSystems Lower Case +Plus. The use of the asterisk ("*") becomes very context dependent. If it is found between two expression, then the multiplication operation is assumed. If it is found in place of an expression, the current location counter value will be substituted in its place.

Immediate addressing mode

Immediate data (i.e. a constant) is preceded by a '#' or '/'. Since the 6502 is an eight bit processor, and address expressions are 16-bits long, there must be some method of choosing the high order byte or the low order byte.

#: When an address expression is preceded by a "#" this instructs LISA to use the low order byte of the 16-bit address which follows.
    SYNTAX: #<expression>

Examples:

```
#LABEL
#$FF
#!6253
#%1011001
#’A'
#"Z"+$1
```

/: When the address expression is preceded by a "/" this instructs LISA to use the high order byte of the 16-bit address which follows.

    SYNTAX: /<expression>

Examples:

```
/LABEL
/$FF
/!6253
/%101001100
/LBL+$4
/$F88F
```

Note: "/" is one of the exceptions to MOS syntax. MOS uses “#<" instead. We feel the "/" is easier to type into the system (it saves you having to type two shifted characters). Another reason for not using the ">" and "<" operators will become evident when discussing local labels.

In addition to the standard syntax, LISA provides the user with three very convenient extensions to the immediate addressing mode.

A single apostrophe followed by a single character will tell LISA to use the ASCII code (high order bit off) for that character as the immediate data. This is identical to "'<character>' except you do not have to type the "#" and closing apostrophe.

    SYNTAX: '<single character>

The quote can be used in a similar manner to the apostrophe, except the immediate data used will then be in the extended ASCII format (high order bit on).

    SYNTAX: "<single character>

Examples:
```
'A       same as #’A'
```

'B	same as #’B'
'%	same as #’%'
"C	same as #”C"
"D	same as #”D"
"#	same as #"#"

If you're wondering why you would want to use the #"A" version, remember that an address expression is allowed after the "#". This allows you to construct constants of the form #"Z"+$1 which is useful on occasion. Address expressions are not allowed after the " or ' in extended form.

The last extension to the immediate mode concerns hexadecimal constants. Since hex constants are used much more often than any other data type in the immediate mode, a special provision has been made for entering them. If the first character of the operand field is a DECIMAL digit ("0"-"9") then the computer will interpret the following digits as immediate HEXADECIMAL data. If you need to use a hexadecimal number in the range $A0-$FF you must precede the hexadecimal number with a decimal zero. This is required so that LISA will not mistake your hexadecimal number for a label.

Examples:

00 same as #$0
05 same as #$5
10 same as #$10
OFF same as #$FF

WARNING** These special forms of the immediate addressing mode were included to provide compatibility with an older assembler. Since LISA's introduction, the assembler using this special syntax has been removed from the marketplace. To help streamline future versions of LISA these syntax additions will not be present in future versions of LISA. They are included in LISA 2.5 only for purposes of compatibility with older versions of LISA.

DON'T USE THESE FORMS IN NEW PROGRAMS YOU WRITE, or someday...........

Accumulator addressing mode

The accumulator addressing mode applies to the following four instructions:

ASL
LSR
ROL
ROR

Standard MOS syntax dictates that for the accumulator addressing mode you must place an "A" in the operand field. LISA is compatible with the mini-assembler built into the Apple and as such the "A" in the operand filed is not required.

Examples of the Accumulator Addressing Mode:

```
          ASL A
LABEL     ROL
          LSR A
```

```
ROR
```

Absolute/Zero page addressing
To use the absolute/zero page addressing mode simply follow the instruction with an address expression in the operand field. LISA handles zero page addressing automatically for you (but see EQU/EPZ description).

Examples:

`LDA LABEL`	(Symbolic label)
`LDA LABEL+$1`	(Label plus offset)
`LDA $1`	(Non-symbolic zero page)
`LDA $800`	(Non-symbolic absolute)
`ASL LBL`	(Symbolic label)
`ROL %10110`	(Non-symbolic zero page)

Indexed by X addressing
LISA supports the standard "indexed by X" syntax. To use this addressing mode, your operand field should have the form:

<expression>,X

When LISA encounters an operand of this form, the indexed by X addressing mode will be used. If it is possible to use the zero page indexed by X addressing mode, LISA will do so.

Note: STY <expression>,X <expression> must be a zero page expression or an assembly time error will result.

Examples:

```
LDA LBL,X
LDA LBL+$1,X
LDA $100,X
LDA $1010,X
```

Indexed by Y addressing
LISA supports the standard "indexed by Y" syntax. To use this addressing mode your operand should be of the form:

<expression>,Y

When LISA encounters an operand of this form, the indexed by Y addressing mode will be used. If it is possible to use the zero page addressing mode (only with LDX & STX) then the zero page version will be used.

Note: STX <expression>,Y <expression> must be a zero page expression or an assembly time error will result.

Examples:

```
LDA LBL,Y
STA LBL+$80,Y
LDX $0,Y
```

Relative addressing
Relative addressing is used solely by the branch instructions. Relative addressing is syntactically identical to the absolute/zero page addressing mode.

Examples:

```
BNE LBL
BCS LBL+$3
BVC *+$5
BMI $900
BEQ LBL-$3
```

Implied addressing
Several mnemonics do not require any operands. When one of these instructions is used, simply leave the operand field blank.

Examples:

```
CLC
SED
PHA
PLP
```

Indirect, indexed by Y addressing
Indirect, indexed by Y addressing has the following syntax:

(<expression>),Y <expression> must be a zero page expression or an assembly time error will result.

Examples:

```
LDA (LBL),Y
LDA (LBL+$2),Y
LDA ($2),Y
LDA (!10+%101),Y
```

Indexed by X, indirect addressing
The indexed by X, indirect addressing mode has the format:

(<expression>,X) where <expression> must be a zero page expression or an assembly time error will result.

Examples:

```
                LDA (LBL,X)
                ADC (LBL+$3,X)
                STA (LABEL-!2,X)
                AND ($00,X)
```

Indirect addressing
The indirect addressing mode can only be used with the JMP instruction. The indirect addressing mode uses the following syntax:

(<expression>) where <expression> may be any valid 16-bit quantity.

Examples:

```
                JMP (LBL)
                JMP (LBL+$3)
                JMP ($800)
```

Local labels

LISA 2.5 supports a special type of label known as the local label. A local label definition consists of the up-arrow ("^") in column one followed by a digit in the range 0-9.

Examples:

```
     ^0         LDA #0
     ^9         STA LBL
     ^7         BIT $C010
```

Local labels' main attribute is that they may be repeated throughout the text file. That is, the local label '^1' may appear in several places within the text file. To reference a local label, simply use the greater than sign ('>') or the less than sign ('<') followed by the digit of the local label you wish to access. If the less than sign is used, then LISA will use the appropriate local label found while searching backwards in the text file. If the greater than sign is used then LISA will use the first appropriate local label found searching forward in the text file.

Examples:

Incrementing a 16-bit value:

```
   INC16        INC ZPGVAR
                BNE >1
                INC ZPGVAR+1
   ^1           RTS
```

A Loop:

```
                LDX #0
   ^8           LDA #0
```

```
           STA LBL,X
           INX
           BNE <8
```

Local labels may not be equated using the EQU, "=", or EPZ pseudo opcodes. They are only allowed to appear as a statement label.

Using LISA

Getting LISA up and running

To run LISA simply boot the disk. When LISA is ready to execute a command you will be greeted with a “!" prompt (the same one used by the mini:-assembler). You can also run LISA by issuing the DOS command "BRUN MXFLS". If LISA is already in memory, you can enter LISA by issuing the Apple monitor command "E000G" or control-B. This enters LISA and clears the text file in memory. If you wish to enter LISA without clearing the existing text file memory space (a “warm start" operation) use the "E003G" monitor command or control-C. Note: See warning and extraneous notes for the “warm start” procedure.

The commands

After you successfully enter LISA, the computer will be under the control of the command interpreter. This is usually referred to as the command level. When you are at the command level a "!" prompt will be displayed and the computer will be waiting (with a blinking cursor) for a command. When at the command level you have several commands available to you. They are:

N(EW)
LO(AD)
SA(VE)
W(RITE)
^D(control-D)
L(IST)
I(NSERT)
D(ELETE)
M(ODIFY)
^P(control-P)
A(SM)
AP(PEND)
LE(NGTH)
BRK
F(IND)

The optional information is enclosed in "()". As an example you only need type "LO" to perform the "LOAD" command, "I" to execute “INSERT" command, etc.

Explanation of each of the LISA commands

I(NSERT) {line#}

Insert command, will allow user to insert assembly language source code into the source file. This command accepts text from the keyboard and inserts it before line number "line#". If "line#" is not specified, text is inserted after the last line in the text file. If the current text file is empty, then insert will begin entering text into a new text file. If a line number is specified which is larger that the number of lines in the file, text will be inserted after the last line in the text file. To terminate the insert mode type control-E as the first character of a new line.

LISA uses a logical line number scheme. The first line in the text file is line number one, the second line is line number two, line three is number three & etc. Whenever you perform an insertion between two lines the line numbers are more or less "renumbered".

As an example of what happens, boot LISA disk and get into the command interpreter. Once in command mode, type "I" followed by a return.

LISA will respond with a line number of one and will wait for text to be entered in the system. At this point type "LBL LDA 00" followed by return.

LISA will print a "2" on the video screen and await the entry of line number two. Now type " END" (note the space before the END) followed by return.

LISA will respond by printing "3" on the video screen. Now press control-E followed by return to terminate text entry. LISA will return to the command level which you will verify by noticing the "!"prompt.

Now type "I 2" at the command level. LISA will respond with the line number two and will once again await you text entry.

DO NOT WORRY ABOUT DELETING THE PREVIOUSLY ENTERED LINE #2.

Each time you enter a line LISA "pushes" the existing lines down into memory. To prove this to yourself enter " STA $00" (note the blank space before the instruction) followed by return.

When "3" appears prompting you to enter a new line press control-E to exit the Insert mode.

Now type "L" and the Apple will display:

```
1 LBL  LDA 00
2      STA $00
3      END
```

Notice that "END" which was previously at line #2 has become line #3 after the insertion. Since the line numbers change every time an insertion is performed it's a good idea to list a section of your source every time you perform an operation on it because the line number you decide to use may have been modified by previous editing.

D(ELETE) line#1{,line#2}

Deletes the lines in the range specified. If only one line number is specified then only that line is deleted. If two line numbers, separated by a comma, are specified then all the lines in that particular range are deleted.

Examples:

```
DELETE 2 deletes line #2
DELETE 2,6 deletes lines 2 to 6
```

Note that again, as with insert, the lines are renumbered after the command to reflect their position relative to the first line.

L(IST) {line#1{,line#2}}

Lists the lines in the specified range. If you need to scan a section of the text file there are two options that facilitate searching for a specified line. If, during the list, you press the space bar the listing will stop until the space bar is pressed again. If the space bar is repressed the listing will continue from where it left off. If instead of pressing the space bar you press control-C then you will be returned to the command level.

Examples:

```
LIST         lists the entire file
LIST 2       lists only line #2
LIST 2,6     lists lines 2 to 6
```

L(OAD) filename

The specified LISA files will be loaded in. All valid DOS binary file options except ",A" may be suffixed to the name. LISA files are usually loaded in at location $1800.

Example:

```
LOAD LZRIOS            loads the file LZRIOS from disk
```

Note: Although the command "LOAD" is begin used this does not mean the LISA used the DOS LOAD command. Internally (and before DOS has a chance to see it), "LOAD" is converted to “BLOAD".

S(AVE) filename

The file in memory is saved to disk under the specified name. SAVE is internally converted to "BSAVE", so all conventions, restrictions, etc., which apply to "BSAVE" may be used (You cannot, however, specify a starting address and length as LISA does this automatically and will override your specs). Files saved using the LISA S(AVE) command are saved as special "L" type files.

Example:

```
SAVE TEMP              saves the source file TEMP to disk
SAVE TEMP,S6,D2        saves the source file TEMP to disk in a specified drive.
```

AP(PEND) filename

Reads in a source file from disk and appends it to the existing source file in memory.

Example:

```
APPEND TEMP
APPEND TEMP,D2
```

^P (control-P)

When control-P is pressed LISA will jump to location $E009 and begin execution there.

Currently at location $E009 is a JMP to the command processor used by LISA. You may replace this JMP with a jump to a location where your particular routine begins. Then by pressing control-P (followed by return), LISA will jump to your particular routine. To return to the command level you may either execute an RTS instruction, or JMP $E003. Space has been provided for your user routine in the area $9480-$95FF.

WARNING**** Use only a JMP instruction at location $E009 as LISA system jumps appear both before and after the JMP $E009.

A(SM)

Assembles the current text file in memory. LISA currently allows up to 512 labels in the symbol table, to change this see the appropriate appendix. During assembly if any errors are encountered LISA will respond with:

```
A(BORT) OR C(ONTINUE)?
```

as well as the error message. Should you wish to continue the assembly (possibly to see if any further errors exist), write down the line number of the current infraction and press "C" followed by return. If you wish to immediately exit the assembly mode to correct the error, press "A" then return.

W(RITE) filename

Writes the current text file to disk as a TEXT file type. This allows you to manipulate your text file with a BASIC or APPLESOFT program. In addition, TEXT type files may be read into APPLE PIE (VER. 2.0 or greater), and you can modify your text files using this very powerful text editor. The first line output when using the W(RITE) command is "INS". With "INS" as the first line in the text files you may use the DOS "EXEC" command to reload these TEXT type files back in LISA (See control-D for more)

L(ENGTH)

Displays the current length of the text file in memory.

^D (control-D)

Allows you to execute one DOS command from LISA.

^D PR#n -turns on an output device
^D IN#n - " " " input "
^D INT -does not put you into BASIC, but rather returns you to LISA
^D EXEC filename -where file is a TEXT type file previously created by the W(RITE) command, loads into LISA the desired text file.
~D (any other DOS command) -executes that command

M(ODIFY) line#1{,line#2}

Performs the sequence:

```
L(IST) line#1{,line#2}
D(ELETE) line#1{,line#2}
I(NSERT) line#1
```

which allows you to effectively replace a single line or many lines. If you do not specify a line number then the entire file will be listed. You will get an ILLEGAL NUMBER error, and you will be placed in the insertion mode with the inserted text being inserted after the last line of the text file.

N(EW)

Clears the existing text file. You are prompted before the clear takes place.

BRK

Exits from LISA, enters Apple monitor

F(IND)

Searches for the label specified after the find command. FIND will print the line number of all lines containing the specified label in the label field.

Screen editing

To move the cursor, use the following key combinations:

up	Control-O
down	Control-L

right	Control-K
left	Control-J

Other cursor commands:

Right arrow (Ctrl-U)	Copies character under cursor
Left arrow (Ctrl-H)	Deletes character under cursor

Using lower case characters
Unless you have Lazer MicroSystems' Lower Case +Plus lower case letters will appear as garbage on the screen. They are lower case in memory, hence dumping to the printer with lower case capabilities you will have lower case printed. When moving the cursor over the lower case letter junk will seen on the screen, you will see a blinking or inverted upper case letter. You can use this facility to double check lower case entry if you do not have the adapter. Since the shift key does not function for input, the ESC is used as a shift key when a software shift has to be used. "Caps lock" mode, is toggled by pressing- Control-S. In upper case mode the cursor will blink, in lower case mode it will be a static inverted character. While the caps lock mode is on the ESC will not work.

LOWER CASE ADAPTED SPECIAL KEYS:

Character	Press Key
|	"!" or "1"
~	""" or "7"
{	"(" or "8"
}	")" or "9"
[	"<" or ","
]	">" or "."
_	"-"
\	"/"````
DEL	Prints a funny looking box on the screen (but not on the printer) by pressing "#" or "3"

Pseudo Opcodes

As opcodes tell the 6502 what to do, pseudo opcodes tell LISA what to do. With pseudo opcodes you may reserve data, define symbolic addresses, instruct LISA as to where the code is to be stored, access the disk, etc. The pseudo opcodes are:

OBJ: OBJECT CODE ADDRESS
SYNTAX: OBJ <expression>

An assembler takes a source file which you create and generates an "object code" file. This file has to be stored somewhere. It is possible to store the object file to disk, however this causes assembly to proceed very slowly, because the disk is very slow compared to the computer. The object file many also be stored directly in memory thus allowing the source file to be assembled at full speed. Except in certain cases, LISA always stores the assembled program into RAM memory.

Under normal circumstances (meaning you have not told LISA otherwise), programs are stored in RAM beginning at location $800 and grow towards high memory. Often, the user needs to be able to specify where the code will be stored in memory. The OBJ pseudo opcode would be used in this instance. When an OBJ pseudo opcode is encountered in the source file, LISA will begin storing the object code generated at the specified address. This allows you to assemble code at one address and store it in another. Another use of the OBJ pseudo opcode is to reuse memory in a limited memory environment.

Suppose you wish to assemble a text file 10K bytes long. Unfortunately LISA does not leave you 10K free for this use (only 4K). What you do is to assemble the first 4K of code and then save this first portion of code to disk. Now, by using the OBJ you can instruct LISA to assemble the next 4K of code on top of the old code which was saved to disk. This allows a very flexible management of memory resources. Another example, is when you wish to assemble your code for an address outside the $800-$1800 range. Since LISA uses almost every byte outside of this range for one thing or another you must assemble your code within this area. Unfortunately, not all users want to be restricted to this area. Many users might wish to assemble an I/O driver into page 3 or possibly up in high memory.

Regardless of where you wish the program to run, the object code generated by LISA must be stored within the range $800-$1800. Simply use the OBJ to store your code beginning at location $800 and remember to move it to it's final location (using the monitor "move" command or the DOS ",A$" option) before running it. LISA contains a special variable called the code counter. This variable points to the memory location where the next byte of object code will be stored. The OBJ will load the value contained in its operand field into the code counter (in fact that's the only operation OBJ performs). Other pseudo opcodes affect the code counter as well, they will be pointed out as they are discussed.

ORG:PROGRAM ORIGIN

SYNTAX: ORG <expression>

When ORG is encountered LISA begins generating code for the address specified in the address expression. When you use ORG you are making a promise that you will run your program at the address specified. If you set the program origin to $300, then you must move the program to location $300 before running it.

Whenever ORG is executed it automatically performs an OBJ operation as well. Thus, if you do not want the code to be stored where you have ORG'd it, you must immediately follow the ORG statement with an OBJ statement. If you do not specify a program origin in your program, the default will be $800.

Multiple ORG statements may appear in your program. Their use, however, should be avoided as they tend to cause problems during the modification of a program (e.g.if you re-

ORG the program at some later date those embedded ORG statements can kill you). LISA supports several opcodes that reserve memory, so there is no real need for more than one ORG statement within a normal program.

ORG evaluates the expression in the operand field and loads the calculated variable into the code counter and the LISA location counter variable. It is important to remember that ORG affects both the location counter and code counter.

WARNING** Locations $800-$1800 are reserved for code storage. If you assemble outside this range possible conflicts with LISA, the source file, the symbol table, or I/O buffer areas may arise. If you need to assemble your code at an address other than in the range $800-$1800 be sure to use the OBJ pseudo opcode to prevent conflicts.

EPZ: EQUATE TO PAGE ZERO

SYNTAX: LABEL EPZ <expression>

The label is assigned the value of the expression and entered into the symbol table. If <expression> evaluates to a value greater than $FF then an assembly time error occurs. If any symbolic references appear in the expression then they must have been previously defined with an EPZ pseudo opcode, or an error will result. Although LISA doesn't require you to do so, it is a good practice to define all of you zero page locations used in your program before any code is generated.

Zero page is used mainly to hold variables and pointers. Before wildly using up locations in zero pages, it's wise to consult your Apple manuals to make sure that there are no zero page conflicts between your program and the monitor or whatever language you are using.

When a variable is defined using the EPZ pseudo opcode then zero page addressing will be used if at all possible. The label is not optional for the EPZ opcode. The EPZ opcode only supports the addition and subtraction operators in address expressions.

EQU: EQUATE

SYNTAX: LABEL EQU <expression>
or LABEL = <expression>

The 16-bit value of <expression> will be used as the address for LABEL, and it will be entered into the symbol table. Absolute addressing will always be used when using the EQU opcode, even if the expression is less than $100. <expression> may contain symbolic references (i.e. labels), but they must have been previously defined in either an EQU statement, an EPZ statement, or as a statement label. EQU may also be used to create symbolic constants. For instance:

```
 CTLD  EQU $84      or   HIMEM   EQU $9600    or  LETRA  = "A"
       LDA #CTLD               LDA #HIMEM             LDA #LETRA
```

The use of symbolic constants in your program helps improve the readability considerably.

ASC: ASCII STRING DEFINITION

SYNTAX: ASC 'any string'

or   ASC "any string"

The ASC pseudo code instructs LISA to store the following text directly in memory beginning at the current location. If the apostrophe is used, then the text is stored in normal ASCII format (i.e. high order bit off). If the quotes are used, then the character string is stored in memory in an extended ASCII format (i.e. high order bit on). Since the APPLE II computer uses the extended ASCII format, you will probably use the latter version most of the time. If the apostrophe begins the string, then the apostrophe must be used to terminate the string. Quotes may appear anywhere inside such a string with no consequence. If the quotes are used to delimit the string, then an apostrophe may be placed anywhere inside the string with no problems whatsoever. In this case the quote must be used to terminate the string.

Examples:

```
   ASC 'THIS "STRING" IS OK!'
   ASC "SO IS THIS 'STRING'"
   ASC 'THIS IS 'NOT' ALLOWED'
```

The last example is illegal because the first occurrence of the apostrophe terminates the string, leaving an illegal operand delimiter (NOT) in the operand field. Should you ever need to place an apostrophe or a quote within a string delimited by the respective character it can be accomplished by typing two of these characters together in the string.

Examples:

```
               ASC "THIS IS ""HOW"" YOU DO IT!"
               ASC 'THIS ''WAY'' WORKS FINE ALSO'
               ASC '''THIS LOOKS WEIRD,
   BUT IT WORKS'''
```

In the last example the created string is:

```
           'THIS LOOKS WEIRD,
BUT IT WORKS'
```

Note: ASC is more or less obsolete. However, it was decided that LISA 2.5 had to be compatible with earlier versions of the product and that is why it is still included. That said, when writing new programs you should use the BYT and .DA pseudo opcodes.

STR: CHARACTER STRING DEFINITION

SYNTAX: STR 'any string'
or STR "any string"

Most high level languages define a character string as a length byte followed by 0 to 255 characters. The actual number of characters following the length byte is specified in the length byte. Strings stored this way are very easy to manipulate in memory. Functions such as concatenation, substring (RIGHT$, MID$, and LEFT$ in BASIC), comparisons, output, etc., are accomplished much easier when the actual length of the string is known.

Except by manually counting the characters up & explicitly prefacing a length byte to your string,the ASC opcode does not allow you to use this very flexible data type. The STR opcode functions identically to the ASC opcode with one minor difference, before the characters are output to memory, a length byte is output. This allows you to create strings which can be manipulated in a manner identical to that utilized in high level languages.

Examples:

```
STR 'HI'        outputs   02 48 49
STR "HELLO"     outputs   05 C8 C5 CC CC CF
```

HEX: HEXADECIMAL STRING DEFINITION

The HEX pseudo opcode allows you to define hexadecimal data and/or constants for use in your program. HEX may be used for setting up data tables, initializing arrays, etc. The string of characters following the HEX are assumed to be a string of hex digits. Each pair of digits is converted to one byte and stored in the next available memory location pointed at by the location counter Since exactly two digits are required to make one byte, you must enter an even number of hexadecimal digits after the HEX pseudo opcode, or an error will result. As such, leading zeros are required in hex strings. The hex string does not have to begin with a "$" (in fact it cannot begin with a "$"!).

Examples:

```
HEX FF003425
HEX AAAA8845
HEX 00
```

LST: LISTING OPTION ON

LST activates the listing option. During pass three all source lines after LST will be listed onto the output device (usually the video screen). Listing will continue until the end of the program or until an NLS pseudo opcode is encountered. Note that there is an implicit "LST" at the beginning of your program, so unless otherwise specified your program will be listed from the beginning.

NLS: NO LISTING/LISTING OPTION OFF

NLS deactivates the listing option. When encountered in the source file, all further text until the end of the program or until an “LST" is encountered, will not be listed. LST & NLS can be used together to list small portions of a program during assembly. By placing an "NLS" at the beginning of your program, then a "LST" before the section of code you want to printed, and then an "NLS" after the text you want printed you can selectively print a portion of the text file during assembly. Neither "LST" nor "NLS" allow an operand.

ADR: ADDRESS STORAGE

SYNTAX: ADR <expression> [,<expression>]

The ADR lets you store, in two successive bytes, the address specified in the operand field. The address is stored in the standard low order/high order format. ADR can be used to set up "jump tables" or for storing 16-bit data. ADR is particularly useful for storing decimal and binary constants since conversion to hex is performed automatically. Multiple address expressions may appear in the operand field. If additional address expressions are present, they must be separated from each other with commas.

Examples:

```
   ADR LABEL           ADR LABEL-$1        ADR LABEL+$3
   ADR LBL1,LBL2,LBL3
 *- ADR !10050        *- ADR %10011011000111
```

* Note in particular the last two examples which demonstrate how you can store decimal and binary constants in memory using the ADR. This technique is very useful for translating BASIC programs to assembly language.

END: END OF ASSEMBLY

END tells LISA that the end of the source file has been encountered. During passes one and two LISA will start at the beginning of the text file and continue with the next pass. At the end of pass three control will be returned to LISA's command level. If the END is not present in the source file then a "MISSING END" error will occur at the end of pass one.

ICL: INCLUDE TEXT FILE
SYNTAX: ICL "filename"

ICL is a very powerful and advanced pseudo opcode. It allows you to "chain" in another text file. This pseudo should be used when there is not enough memory available for the current text file. LISA provides you with enough memory for approximately 1500 to 2000 lines of text. Should you try to exceed this limit a "memory full" error will result. When this happens, delete the last 10 lines or so (to give you working space) and, as the last line of your text file use ICL to link in the next file.

Once the ICL has been entered, save the text file to disk. Now use the N(EW) command to clear the text file workspace and then enter the rest of your assembly language text file, continuing from where you left off. Once you have finished entering the text, save the text file disk under the name specified in the ICL. Now load the original files and assemble it.

During assembly LISA will automatically bring in the second file from disk and continue assembly at that point.

Note** You shouldn't use "ICL" unless you really have to. The use of ICL slows down assembly from 20,000 lines per minute to about 500-1000 lines per minute due to considerable disk access.

Since LISA is a three pass assembler the original text file in memory must be saved to disk. It is saved under the name "TEMP." so you should be careful not to use that filename. After assembly the resident text file in memory will be the last text file chained in.

The original text file is not brought back into memory. During assembly, if an error occurs in a section of code which was ICL'd off of the disk, the error message will give you the name of the file, the line number within the file where the infraction occurred, as well as the option of continuing or aborting. If you abort you will find the text file with the error currently in memory. You may fix the error, save the text file to disk again under its original name, then reload "TEMP." and reassemble the text file. ICL is similar to END in that it must be the last statement in the text file. Any additional lines after the ICL will be ignored. There is no limit to the number of files that you can chain together using ICL.

DCM: DISK COMMAND

SYNTAX: DCM "dos command"

During pass one and two the DCM pseudo is ignored. During pass three, however, whatever string is placed between the quotes gets executed as an Apple DOS command. A control-D is not required at the beginning of the DOS command. The DCM has several uses, you may use it to selectively turn on and off input and output devices during assembly (using PR# & IN#), it can be used to save generated code to disk, thus freeing memory space. It can be used to create a disk text file listing of the assembled program, also it can be used to prevent the symbol table listing from being printed, and for loading load modules off of the disk after an assembly. Since LISA only allows 4K bytes for your obj code (from $800-$1800), you have to BSAVE your obj file to disk when this 4K is used up. Once the file is BSAVEd to disk you can use the OBJ pseudo to begin storing your object code beginning at location $800 once again. When the second 4K is used up you must once again use the DCM/OBJ sequence to make room for the new object code.

Once these "load modules" have been saved to disk, you can reload them in sequence and then run the finished product. However, you cannot simply BLOAD each of the object modules and expect your program to run. The BLOAD command loads the program in from where it was saved, since all load modules were saved beginning at location $800, the BLOAD command will load them in on top of each other! To get around this, use the "A$" option when BLOADing a program to load the module into its correct memory location.

In fact, when BSAVEing a program with DCM its a good idea to make the loading address part of the filename (for example: OBJ.1/$1800)

Examples:

```
DCM "BSAVE OBJECT/$800,A$800,L$1000
DCM "BSAVE OBJECT/A$1800,A$800,L$1000"
```

The symbol table listing may be suppressed by using the disk command "INT". This should be entered in your program immediately before the "END". Assembly automatically terminates when the DCM "INT" pseudo is encountered and you are returned to the command level. To create a disk file listing of the assembly text file use the DCM command sequence:

```
DCM "OPEN <filename>"
DCM "WRITE <filename>"
```

Once this has been accomplished all further text normally written onto the screen will be sent to the disk under the name "<filename>". The last statement before the END (or DCM "INT" if present) should be:

```
     DCM "CLOSE".
```

This will close the file, restore buffers, etc. Since the CLOSE will be executed before the symbol table is printed, the symbol table will not be included in your text file listing. If you need to include the text, then omit the DCM "CLOSE" and explicitly CLOSE the file with an immediately CLOSE command when you are returned to the command level.

Warning** Due to the memory management techniques used (48K MAXFILES) is always set to one. This implies that several problems can develop if your program contains other disk commands sandwiched between the OPEN & CLOSE commands. Should you need to execute a disk command while writing the assembled source to disk you must first CLOSE the file. Once closed, you can execute the DOS command. After the command (DOS) is executed you may continue writing the assembly listing by APPENDing (instead of OPENing) and then WRITEing to the file.

*NOTE ** Remember, any DOS command terminates the WRITE command. So, if you issue any DOS commands when writing a text file out to disk you must reissue the WRITE command immediately after the DOS command. ICL uses the DOS, so care must be take when writing files to disk.

PAU: PAUSE/FORCE ERROR

PAU is ignored during passes one and two. During pass three however, this pseudo will automatically cause an error ("**ERROR: PAUSE ENCOUNTERED") to occur. At this point the programmer may A(BORT) the assembly or C(ONTINUE) the assembly.

PAU is very useful for debugging purposes as you don't have to watch the screen with your finger on the space bar should you desire to stop the assembly listing at some particular section of code. PAU is also useful in determining where the 4K cutoff is when you are saving obj files.

Although an error message is generated, this has no effect on the assembly. If the pause error is the only error encountered, then the assembly can be considered successful.

PAG: PAGE EJECT

PAG will print a control-L to the listing device when encountered during pass three. If you are sending the listing to a printer with forms control, your printer should skip to top-of-form. PAG allows you to format your listings nicely, breaking up subroutines so that they begin on different pages.

DCI: DEFINE CHARACTERS IMMEDIATE

```
    SYNTAX: DCI "any string"
    or      DCI 'any string'
```

DCI is a special hybrid pseudo. In function its identical to ASC with one exception: The last character in the string will have an order bit which is opposite the value for the rest of the string. That is, if you are storing the string in memory with high order bit on, then the last character in the string will be stored with its high order bit equal to zero. If the string is being stored memory with the high order bit off, then the last character in the string will be stored in memory with the high order bit one.

Examples:

```
DCI "ABCDE"      generates      C1 C2 C3 C4 45
DCI 'ABCDE'      generates      41 42 43 44 C5
```

INV: INVERTED CHARACTERS
SYNTAX: INV "any string"
or   INV 'any string'

INV takes the string which follows and outputs the characters as APPLE inverted (inverse) characters. The high order bit is always off: so whether you use the apostrophe or quote to delimit the string is of no consequence. You should realize that only the characters directly available from the Apple keyboard plus "[", "\", "_" have inverted counterparts. The lower case letters and several special characters do not have corresponding inverted counterparts, and should they appear within the INV string, garbage will be created.

Examples:

```
INV "ABCDE"      generates      01 02 03 04 05
INV 'ABCDE'      generates      01 02 03 04 05
```

BLK: BLINKING CHARACTERS
SYNTAX: BLK "any string"
or   BLK 'any string'

BLK is the counter to INV. Instead of generating the code for inverted characters, BLK generates code for blinking characters. All restrictions mentioned for INV apply as well to BLK (for the same reason).

HBY: HIGH BYTE DATA
SYNTAX: HBY <expression> {,<expression>}

HBY is similar to ADR except only the high order byte of the following address expression is stored in memory. Combined with BYT its possible to break up address tables into two groups of one byte data apiece instead of the two-byte data generated by ADR. This allows a very convenient method of loading addresses when using the index registers.

Examples:

```
HBY $1234              generates      $12
HBY $F3                generates      $00
HBY LBL              generates      H.O. BYTE OF THE ADDRESS LBL
HBY "A"              any ASCII data always generates $00
HBY LBL1,LBL2,LBL3
```

BYT: LOW BYTE DATA
SYNTAX BYT <expression> {,<expression>}

BYT works in a manner similar to HBY except it stores the low order address byte into memory at the current location. BYT is also useful for introducing symbolic values into your programs. For instance, $00 is often used as the "end-of-string" token. You can define a constant "EOS" (for"end-of-string") and then use BYT to store the value for EOS in memory for you. This has two beneficial effects on your program. First, it makes your program easier to read since "BYT EOS" states exactly what the value is for whereas "HEX 00" is somewhat ambiguous. The second beneficial feature is the fact that should you decide to change the EOS value from zero to say ETX (ASCII end-of-text) you only need change one line (the EQU statement which defines EOS) instead of having to go through your program and change each occurrence of "HEX 00" to "HEX 03"

Examples:

```
BYT $1234  generates     $34
BYT $F3    generates     $F3
BYT "A"    generates     $C1 (EXTENDED ASCII FOR "A")
BYT LBL    generates     CODE CORRESPONDING TO LBL'S LOW ORDER ADDRESS
```

DFS: DEFINE STORAGE
SYNTAX: DFX <expression> {,<expression>}

DFS reserves memory storage for variables. DFS takes the first address expression found in the operand field and adds this value to both the location counter and the code counter. This leaves a wide gap of memory open for use by arrays, variables, etc. If the second operand is not specified, then the memory space reserved is not initialized and contains garbage. The second operand in the address expression, if specified, determines the value to which memory will be initialized. The low-order byte of the second address expression will be stuffed into each byte of the storage reserved by the DFS pseudo.

** Note** This initialization is optional. If it is not explicitly required it should not be used as it slows assembly speed down considerably. If more than two expressions are specified, the remainder are ignored.

Examples:

```
LBL   DFS  $1        reserves one byte at location "LBL"
LBL1  DFS  $100      reserves 256 bytes at location "LBL1"
LBL2  DFS  300,0     reserves 300 bytes and inits them to zero
```

DBY: DOUBLE BYTE DATA
SYNTAX: DBY <expression> {,<expression>}

DBY is used in a manner identical to ADR except that the address data generated is stored in high order (H.O.) byte/low order (L.O.) byte order instead of the normal L.O./H.O. order.

Examples:

```
DBY   $1020     generates      $10 $20
DBY   $1234     generates       $12 $34
DBY   LABEL     generates      (H.O. BYTE) (L.O.BYTE)
DBY   LBL1,LBL2,LBL3
```

LET: LABEL REASSIGNMENT

SYNTAX: LABEL LET <expression>

LET allows the programmer to redefine a previously defined (non-zero page) label. This is useful for defining local labels, counter, etc.

One note of caution: LET is active on passes two and three. EQU and statement label declarations are noted only during pass two. If you declare a label during pass two as a statement label or with the EQU pseudo and then subsequently redefine it with a LET pseudo, the address used during pass three is the value defined in the LET statement regardless of the EQU or statement label definition. This is due to the fact that a label defined using the LET retains that value until another LET redefinition (with the same label) comes along. Since EQU is not active during pass three and statement label values are only noted during pass two, the label will never be set to its original value.

These problems are easily overcome, simply use LET in place of EQU in the original definition. If the original definition was a statement label then substitute "LABEL LET*" instead.

TTL:TITLE

SYNTAX: TTL “STRING"

The TTL pseudo causes an immediate page eject (via control-L/form feed character) and then prints the title specified at the to of the page. Every 65 lines a page eject is issued and the title is printed at the top of the new page.

.IF: CONDITIONAL ASSEMBLY

SYNTAX: .IF <expression>

Conditional assembly under LISA lets you selectively assemble code for different operating environments. For example, you could have a couple of equates at the beginning of a program which specify the target Apple system.

Labels such HASPRNTR, HAS64K, HASMODEM, LCPLUS, KBPLUS, etc., can be set to true or false, depending upon the hardware involved.

For example, LISA 48K and 64K are the same file with just one equate changed.

Conditional assembly handles all the minor details. Conditional assembly uses three pseudo opcodes:

‘.IF'
'.EL'
'.FI'.

'.IF' begins a conditional assembly sequence. '.IF' is followed by an address expression. If it evaluates to true (non-zero), the the code between the '.IF' pseudo and its corresponding '.EL' or '.FI' pseudo is assembled. If the address expression evaluates to false, then the code immediately after the '.IF' pseudo-op will not get assembled

.EL: ELSE
SYNTAX: .EL

'.EL' terminates the '.IF' code sequence and begins the alternate code sequence. The alternate code sequence is assembled only if the address expression in the operand field of the '.IF' pseudo-op evaluates to false (zero). '.EL' (and its corresponding code section) is optional and need not be present in the conditional assembly language sequence.

.FI: CONDITIONAL ASSEMBLY BLOCK END
SYNTAX: .FI

'.FI' terminates the conditional assembly language sequences. It must be present whether or not there is a '.EL' pseudo-op present. All code after a '.FI' will be assembled regardless of the value in the ‘.IF' operand field.

Note LISA does not support nested IF's. If a nested IF is present, you will be given a nasty error at assembly time. All IF's must be terminated before an END or ICL ps-op is encountered or assembly will terminate. To see any example of conditional assembly, look at the "LISA P1.L' file on the LISA master disk.

PHS:PHASE
SYNTAX: PHS <expression>

The PHS ps-op lets you assemble a section of code for a different address, yet include the code within the body of a program running at a different address. This feature lets you include a short driver that runs at location $300, for example, within a program that normally runs up at $1000. It is the responsibility of the program at $1000 to move the program down to location $300. Technically, PHS loads the location counter with the address specified in the address expression, but it does not affect the code counter at all.

In essence, PHS performs an ORG without the OBJ. DPH must be used to terminate the PHS code sequence.

DPH: DEPHASE

SYNTAX: DPH

DPH is used to terminate the section of code following the PHS ps-op. It loads the code counter into the location counter, restoring the damage done by the PHS ps-op.

.DA

SYNTAX: .DA <special expression> {,<special expression>}

'.DA' is another hybrid ps-op. It is a combination of the ADR, BYT and HBY ps-op. It is particularly useful with the SPEED/ASM package's CASE statement and similar routines.

Examples:

```
LBL1 .DA   #CR,RETURN
LBL2 .DA   'C',#LBL,/LBL2,LBL2
LBL3 .DA   "HELLO THERE",#0,STRADR
```

If an address expression is prefaced with the pound sign ("#") then the lower order byte will be used. If an address expression is prefaced with the slash ("/") then the high order byte will be used.

If neither a pound sign or a slash is specified, then the two bytes of the address (in low/high format) will be stored in memory.

GEN: GENERATE CODE LISTING

SYNTAX: GEN

GEN (and NOG) control the output during assembly. If GEN is in effect (the default) all object code output is sent to the display device.

NOG: NO GENERATE

SYNTAX: NOG

NOG will cause only the first three bytes to be listed to the output device during an assembly. This dramatically shortens program listings containing strings and multiple address.

USR: USER DEFINED PSEUDO OPCODE

SYNTAX: USR <anything>

Additional features/Random notes

Extended mnemonics

The word mnemonics means memory aid. "LDA #$FF" is certainly easier read as "load the accumulator with the constant $FF" than is A9FF. There are times when even the mnemonic doesn't make much sense. For instance BCC, Branch if Carry Clear, does not register in most people's minds as meaning the same as branch if less than. Several 6502 instructions can be used, or recognized by the use as a different function. The BCC instruction is but one example.

In order to make 6502 assembly language programming easier to use by the programmer, LISA incorporates several "extended" mnemonics. These extended mnemonics are simply redefinitions of existing mnemonics.

The extended mnemonics are:

BLT	Branch if Less Than, same as BCC
BGE	Branch if Greater or Equal, same as BCS
BTR	Branch if TRue, same as BNE
BFL	Branch if FaLse, same as BEQ
XOR	eXclusive OR, same as EOR

FALSE is defined as $00 and TRUE is defined as anything else.
Note that these extended mnemonics are included IN ADDITION to the existing mnemonics.

SWEET-16 Mnemonics

Sweet 16 is a meta processor or "pseudo microprocessor" originally implemented in 6502 assembly language written by Steve Wozniak and used in the Apple II. Sweet 16 is a kind of virtual machine that gives the 6502 programmer a 16 bit extension to the CPU. Sweet 16 creates sixteen 16-bit registers/pointers (in zero page) and new opcodes to use this registers. Although Sweet 16 is not as fast as standard 6502 code, it can reduce the code size of your programs and ease programming.

LISA incorporates a Sweet-16 assembler for use with the Sweet-16 pseudo machine interpreter in the Apple ROMs. For the most part LISA uses standard WOZ mnemonics except where Wozniak used two or four character mnemonics. Since this tends to disrupt the nice assembly listing, all two & four character long mnemonics were converted to three characters in order to improve the listing format.

Sweet-16 mnemonic conversion

Woz	LISA	Woz	LISA
SET	SET	BR	BRA
LD	LDR	BNC	BNC

Woz	LISA	Woz	LISA
ST	STO	BC	BIC
LDD	LDD	BP	BIP
STD	STD	BM	BIM
ADD	ADD	BZ	BIZ
SUB	SUB	BM1	BM1
POPD	PPD	BNM1	BNM
CPR	CPR	BK	BKS
INR	INR	RS	RSB
DCR	DCR	BS	BSB
RTN	RTN	BNZ	BNZ
POP	POP	STP	STP

Warnings and other extraneous notes

The RESET key has been and always will be a major pain. When pressed at the wrong time it can cause all kinds of trouble. There are two times when RESET absolutely cannot be pressed. Whenever anything is being written out to disk, should the RESET key be pressed you will, at least, destroy the file being written. At worst you could destroy the whole disk. Users of the Autostart ROM cannot hit RESET while in the LISA I(NSERT) mode. In this case, you will be returned to the command level & part of your text file may be lost. Other than the above cases there are fixes, should you accidentally hit RESET. If you were in the command processor when RESET was pressed, simply type E003G or CTRL-C. This will return you to LISA without erasing the existing text file, or changing pointers, etc. If you were in the I(NSERT) mode when the RESET key was pressed, you CANNOT use E003G to restart LISA. If you do so, your text file will be partially destroyed.

If you do hit RESET while in the I(NSERT) mode, simply type E006G from the monitor and you will be returned to the insert mode. The only data lost is the last line you typed in (hopefully).

To prevent depression of the RESET key you might think of removing it. A small screwdriver may pop it off easily. “Cold start” is E000G (control-B). This clears the text file, reinitializes pointers. Never set MAXFILES to anything other than 1 while using LISA 48K. If you do set MAXFILES to anything else, LISA will be destroyed.

Advanced Topics

Memory allocation

Upon “cold start”, several parameters are initialized, in particular it sets the start of text pointer (STXT) to $1800 and the upper text file limit to $8000 (HMEM). This gives you 4K for object code, 26K for text file use, and 5K for symbol table. Since 80% of assembly language programs are less than 2000 lines long, contain less than 512 symbols, and produce object code that is less than 4K bytes long, the “cold start” setting should prove sufficient.

By changing two bytes at the beginning of LISA, it is possible to change these memory "fences" anywhere in memory. For disk based operation, you may want to adjust HMEM so that more than 512 labels are allowed. Likewise, you may want to decrease the size of the allowable text file and make more room for object code by setting STXT to $2000 or even $2800.

Sound judgment as to how to adjust these values. You should realize that each entry in the symbol table requires 8 bytes. Also, the average line of text is about 10 bytes. Object code is generated at the approximate ratio of two bytes per line of source code. To change the parameters follow these instructions:

Load LISA and then get into the monitor (i.e. the "BRK" command). Enter E000L. A series of jumps will be displayed and then, at location $E01B, the hex value $18 will appear. This particular byte is used to set the high order byte of STXT. By changing location $E01B to $20 and then executing a “cold start” (7000G/E000G) you will set STXT to $2000. The low order byte of STXT is always set to $00. Immediately following the STXT parameter is the HMEM parameter. Currently (at location $E01C) the value $80 is stored here. This is the high order byte for HMEM. As with STXT this value can be modified and will take place on the next “cold start”.

Warning: If you create a large text file set up for the co-resident mode, and then attempt to load this text file into LISA after modifications to STXT and/or HMEM have been made, problems may develop. If the large text files takes up more memory than is allowed by HMEM, no load error will result. Upon assembly, however, part of your text file may be destroyed by the symbol table. The aforementioned parameters should therefore not be modified without careful consideration.

LISA's file (and internal) format

The LISA assembler stores source code in a binary format. LISA source files are very similar to standard Apple II binary files, although they use a different file type, known as new B ($40) to differentiate it from the more common B file type ($04). Just like regular binary files, LISA source files start with the load address (2 bytes) and the length of the file (2 bytes). The load address should generally be $1800.

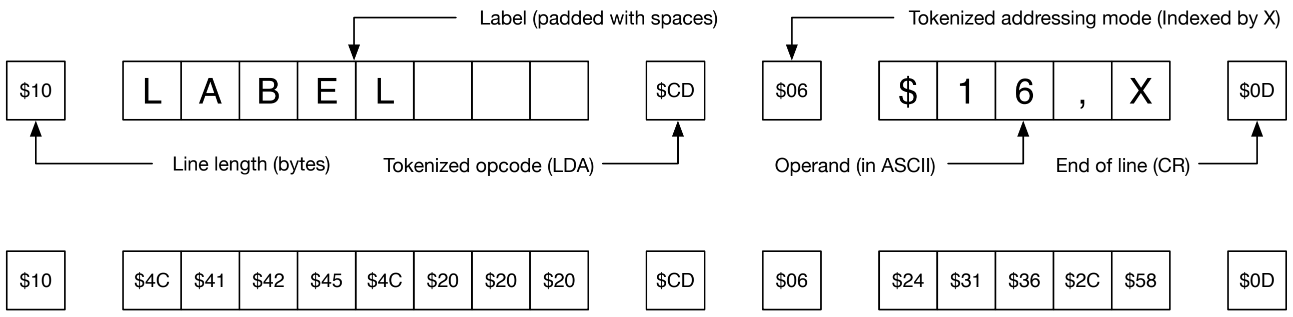
In order to deliver blazing fast performance, every time the programmer enters a new program line, the data is immediately tokenized. That means that the ASM command has less work to do as the program’s syntax has already been validated and the 6502 mnemonics, extended mnemonics, Sweet-16 mnemonics and LISA pseudo-opcodes have already been tokenized. In addition, LISA also tokenizes the addressing mode, which allows for further performance gains.

In a LISA file, the 4 byte long header is followed by the source code.

Each line in the source code starts with a length byte and ends with a CR ($0D). Let's consider the following line:

```
LABEL     LDA $16,X
```

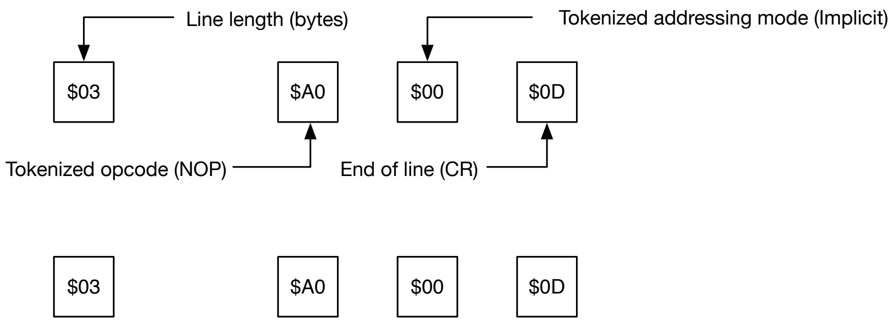
This is how it will be stored on disk:



When a source code line doesn't include a label, LISA 2.5 saves some disk space by not including the first eight bytes. That situation is very easy to detect, as ASCII characters have values < 128, while opcodes have values >= 128. Here is an example of a source code line that does not include a label:

```
          NOP
```

This is how LISA 2.5 stores this very simple line on disk:



If a line starts with either ";" ($3B) or "*" ($2A), it is a comment and the line contains no tokens.

Example:

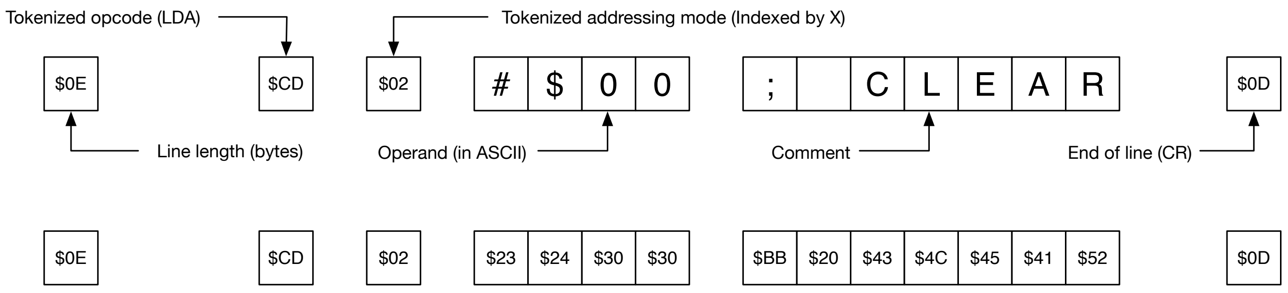
```
* This line is a long comment *
```

If a line that contains code includes a comment after the operand, the comment will be stored just after the operand, with no separation. In this case, the ASCII code for ";" has his high bit set ($BB).

Example:

```
        LDA  #$00      ; CLEAR
```

This is how LISA 2.5 represents this line internally:



Finally, all LISA source files end with a $FF byte.

Tokenized opcodes

Token	Mnemonic	Type	Token	Mnemonic	Type
$80	BGE	Extended mnemonic	$C0	ADC	6502 mnemonic
$81	BLT	Extended mnemonic	$C1	AND	6502 mnemonic
$82	BMI	6502 mnemonic	$C2	ORA	6502 mnemonic
$83	BCC	6502 mnemonic	$C3	BIT	6502 mnemonic
$84	BCS	6502 mnemonic	$C4	CMP	6502 mnemonic
$85	BPL	6502 mnemonic	$C5	CPX	6502 mnemonic
$86	BNE	6502 mnemonic	$C6	CPY	6502 mnemonic
$87	BEQ	6502 mnemonic	$C7	DEC	6502 mnemonic
$88	BVS	6502 mnemonic	$C8	EOR	6502 mnemonic
$89	BVC	6502 mnemonic	$C9	INC	6502 mnemonic
$8A	BSB	Sweet-16	$CA	JMP	6502 mnemonic
$8B	BNM	Sweet-16	$CB	JSR	6502 mnemonic
$8C	BM1	Sweet-16	$CC		Unused
$8D	BNZ	Sweet-16	$CD	LDA	6502 mnemonic
$8E	BIZ	Sweet-16	$CE	LDX	6502 mnemonic
$8F	BIM	Sweet-16	$CF	LDY	6502 mnemonic
$90	BIP	Sweet-16	$D0	STA	6502 mnemonic
$91	BIC	Sweet-16	$D1	STX	6502 mnemonic
$92	BNC	Sweet-16	$D2	STY	6502 mnemonic
$93	BRA	Sweet-16	$D3	XOR	Extended mnemonic

Token	Mnemonic	Type	Token	Mnemonic	Type
$94	BTR	Extended mnemonic	$D4	LSR	6502 mnemonic
$95	BFL	Extended mnemonic	$D5	ROR	6502 mnemonic
$96	BRK	6502 mnemonic	$D6	ROL	6502 mnemonic
$97	BKS	Sweet-16	$D7	ASL	6502 mnemonic
$98		Unused	$D8	ADR	Pseudo opcode
$99	CLC	6502 mnemonic	$D9	EQU	Pseudo opcode
$9A	CLD	6502 mnemonic	$DA	ORG	Pseudo opcode
$9B	CLI	6502 mnemonic	$DB	OBJ	Pseudo opcode
$9C	DEX	6502 mnemonic	$DC	EPZ	Pseudo opcode
$9D	DEY	6502 mnemonic	$DD	STR	Pseudo opcode
$9E	INX	6502 mnemonic	$DE	DCM	Pseudo opcode
$9F	INY	6502 mnemonic	$DF	ASC	Pseudo opcode
$A0	NOP	6502 mnemonic	$E0	ICL	Pseudo opcode
$A1	PHA	6502 mnemonic	$E1	END	Pseudo opcode
$A2	PLA	6502 mnemonic	$E2	LST	Pseudo opcode
$A3	PHP	6502 mnemonic	$E3	NLS	Pseudo opcode
$A4	PLP	6502 mnemonic	$E4	HEX	Pseudo opcode
$A5	RTS	6502 mnemonic	$E5	BYT	Pseudo opcode
$A6	RTI	6502 mnemonic	$E6	HBY	Pseudo opcode
$A7	RSB	Sweet-16	$E7	PAU	Pseudo opcode
$A8	RTN	Sweet-16	$E8	DFS	Pseudo opcode
$A9	SEC	6502 mnemonic	$E9	DCI	Pseudo opcode
$AA	SEI	6502 mnemonic	$EA		Unused
$AB	SED	6502 mnemonic	$EB	PAG	Pseudo opcode
$AC	TAX	6502 mnemonic	$EC	INV	Pseudo opcode
$AD	TAY	6502 mnemonic	$ED	BLK	Pseudo opcode
$AE	TSX	6502 mnemonic	$EE	DBY	Pseudo opcode
$AF	TXA	6502 mnemonic	$EF	TTL	Pseudo opcode
$B0	TXS	6502 mnemonic	$F0	SBC	6502 mnemonic
$B1	TYA	6502 mnemonic	$F1		Unused
$B2	ADD	Sweet-16	$F2	LET	Pseudo opcode
$B3	CPR	Sweet-16	$F3	.IF	Pseudo opcode
$B4	DCR	Sweet-16	$F4	.EL	Pseudo opcode
$B5	INR	Sweet-16	$F5	.FI	Pseudo opcode
$B6	SUB	Sweet-16	$F6		Unused
$B7	LDD	Sweet-16	$F7	PHS	Pseudo opcode
$B8	POP	Sweet-16	$F8	DPH	Pseudo opcode
$B9	PPD	Sweet-16	$F9	.DA	Pseudo opcode

Token	Mnemonic	Type	Token	Mnemonic	Type
$BA	STD	Sweet-16	$FA	GEN	Pseudo opcode
$BB	STP	Sweet-16	$FB	NOG	Pseudo opcode
$BC	LDR	Sweet-16	$FC	USR	Pseudo opcode
$BD	STO	Sweet-16	$FD		Unused
$BE	SET	Sweet-16	$FE		Unused
$BF		Unused	$FF		Unused

Tokenized Addressing modes

Code	Addressing mode
$00	Implicit
$01	Absolute/Zero page or relative
$02	Immediate
$03	Indirect
$04	Indirect, pre indexed by X
$05	Indirect, pre indexed by Y
$06	Indexed by X
$07	Indexed by Y
$08	Used by Sweet-16. Examples: LDR R5 or STO R5
$09	Used by Sweet-16. Example: LDR @R5
$0A	Used by Sweet-16. Examples SET R1,$A034

LISA's speed

Due to the fact that the mnemonics are tokenized and the input line is pre-scanned for syntax errors. Speed could probably be doubled by improving the symbol table search routines, but the need (for greater speed) is yet to justify the additional work.

Using a different text editor with LISA

If you have access to a text editor that outputs text type files to disk you may be able to use it with LISA files. To do so W)rite the LISA file to disk as a text type file and load it into your text editor. To convert the file back into LISA format, type control-D EXEC, followed by the filename from the command level. When doing so, always make sure the first line in the file contains "INS" so that the file is loaded in properly. You should not use an external text file while creating a file, LISA's editor is better suited for that purpose. An external editor should be used when modifying large files, such as those created by the DISASM/65 module.

Converting files from LISA 1.5 to LISA 2.5

To convert: LO(AD) the file into version 1.5 and W(RITE) the file back to disk as a TEXT type file, then EXEC the file into LISA 2.5.

Loading LISA 2.X files into LISA 2.5

LISA 2.0, 2.1, 2.2, and 2,3 saved source files on the disk as binary files. LISA's LO)AD command only loads "L" files and will generates a FILE TYPE MISMATCH ERROR if you attempt to load a LISA 2.x binary type file.

To load these files, you have to use the LOB command (for LOad Binary) included in LISA 2.5's command set. LOB can be used to load older LISA source files into version 2.5, which may then be saved to disk as an “L" file from LISA 2.5

”Tips & Tricks” when using LISA

Effectively using LISA’s built-in editor

The first thing to remember is that all modifications to a text file affect the line numbering scheme. Line numbers are dynamic (always changing) as opposed to the static line numbering system in BASIC (static means fixed). Although the line editor is very simple, the screen editing features make it very powerful.

The screen editing commands allow the user to move the cursor up (Control-O), down (Control-L), right (Control-K), and left (Control-J). Note that the mentioned control characters are NOT entered into the line buffer, nor is the character under the cursor before or after the movement takes place. The cursor control characters simply move the cursor around on the video screen. Control-H (backspace/left arrow) erases the previously entered character from the line buffer. If you try to erase past the beginning of the currently entered line the backspace will be ignored. Control-U (right arrow) copies the character currently under the cursor into the line buffer. The cursor is also moved one location to the right. These cursor commands, when combined with the normal editing commands, form the basis of a very powerful screen oriented editor.

Example:
If line 33 in your text file contains: "LABL LDA 00" and you wish to change it to: "LABEL LDA 00” this is easily done.

M(ODIFY) line 33 (M 33). Press control-O to move the cursor up one line (and on top of the "L" in “LABL").

Now press control-U (right arrow) three times to copy "LAB". When this is done press control-J to move the cursor back one character, next press “E".

Now use the right arrow (control-U) to copy the rest of the line & hit return.

After you hit return press control-E to get out of the insert mode. Now L(IST) 33. You will see your correction.

As you see, insertions into the current line are performed by copying up to the desired location, moving the cursor back one character entering the insertion, and finally the right arrow is used to copy the rest of the line. To delete characters from a line, simply use control-K cursor control to skip over the undesired characters.

It's also possible to use the text editor to copy or move several lines of text. To copy a block of text elsewhere in memory first L(IST) the lines you wish to copy (up to 20 lines). Next, use the IN(SERT) command to insert text where you want to copy the block of text. Now use the control-O to move the cursor over the first character of the lines you previously listed out. By using the right arrow (and of course return at the end of each line), you can copy these lines into the new memory location. If you need to copy more that 20 lines do it in several stages, copying 20 lines each time (but remember to check the line numbers in case they may have been changed by the I(NSERT) command. To move a block of text, follow the procedure for the copy as above. Once copy is complete, delete the original lines using the D(ELETE) command.

Using the AP(PEND) command
The AP(PEND) command will take a source file from disk or tape and append it to the end of the existing text file in memory. This is useful if you need to copy a set of declarations, a copyright notice, a set of I/O routines, etc., to each of your text files. It is also useful when you need to copy a section of code several times in your program & the copy procedure mentioned above turns out to be too much work. You should be careful when AP(PEND)ing files onto a text file in memory which is very large. AP(PEND) does not check to see if the memory is full or not, so always use the LE(NGTH) command to make sure that there is enough room available.

Using the DOS "EXEC" command
The Apple DOS EXEC command can be used to create several "shells" or procedures. For instance, if you create a text file on disk (using APPLE PIE or equivalent) which contains:

```
INS 1
ASM
NLS DEL 1
<CONTROL E>
```

When you EXEC this file, it will assemble the current text file without listing the program.

The EXEC command is also useful for performing intra-file merging. By W(RITE)ing a text file out to disk and then modifying the first line so that it reads "INS <linemnum>” instead of just "INS" you can create a text file which when EXEC’d will insert the following text in front of line <linenum>. Besides allowing you to insert text within a file, instead of just at the end of the file as with AP(PEND), this version will catch a memory full condition should it arise. In fact if you think you might run out of memory when using the AP(PEND) command, it might be a good idea to use the EXEC method of appending text files, just in case.

What happens when you assemble a LISA source file

The 6502 microprocessor does not understand "assembly language”. What is does understand is "6502 machine language". Assembly language & machine language are one and the same, but in fact, they are not.

Assembly language consists of the labels, mnemonics, expressions, and comments which have been previously described in this manual. Machine language, on the other hand, is an unreadable collection of binary data. An assembler's job is to convert assembly language to machine language so that the 6502 will understand what's going on. It is fairly easy to convert assembly language to machine language. At some point during an assembly, stop the assembly by pressing the space bar. In addition to the source code, which is displayed on the right hand side of the screen (possibly with warp-around) you will see several HEX digits on the left side of the screen. The first four HEX digits correspond to the address where the program will reside when run. Following these four HEX digits, separated by one space, come zero to six HEX digits. These digits correspond to the matching code generated by LISA. If you look up these code on the 6502 program card you will see that these operation codes correspond to the actual assembly language instructions (plus any required data). The next four digits correspond to the text file line number followed by the actual source code. Normally the object code (machine code) produced is stored in memory at the address specified by the address listed at the far left (but see the OBJ opcode). When you run the program the code must be stored at this location in memory. If the program is not stored at this location in memory, it will not work properly when executed. Some are an exception to this rule called “relocatable” programs.

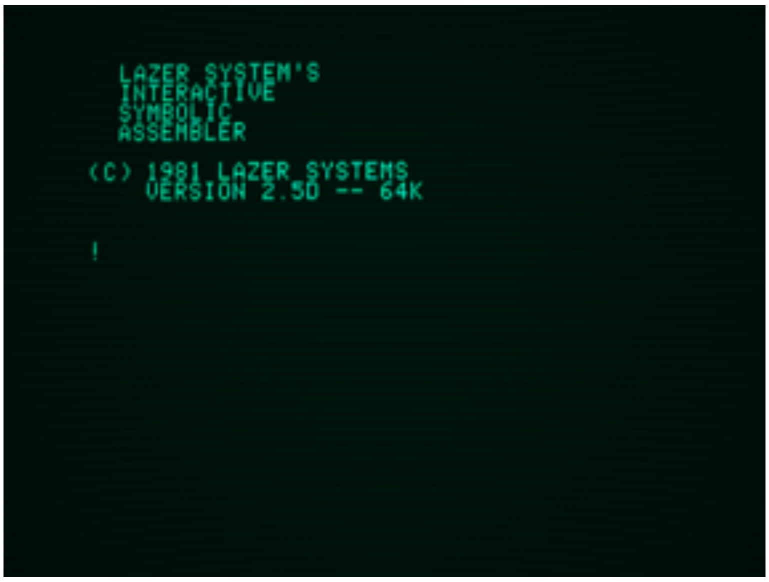
Once assembled, running an assembly language program is easy. First get into the monitor by issuing the LISA "BRK" command. Once in monitor you may run your program by using the monitor “GO" command (similar to BASIC "RUN"). If your program begins at address $800 you must issue the monitor command "800G" to run your program. This command tells the monitor to begin running the machine language program located at location $800. To terminate your machine language program, there are several ways. You can issue a BRK mnemonic which will stop the program, print the current contents of the 6502 registers & return to the monitor. If you terminate with a RTS instruction you will be returned to the calling routine (in this case the monitor). You also can use JMP instruction to enter BASIC, LISA, Pascal or some other program directly.

NOTE ** If you plan to CALL your routine from BASIC or Pascal, you should always end you assembly language program with a RTS instruction so that control will be returned to the calling program.

Running a LISA program: From Start to Finish

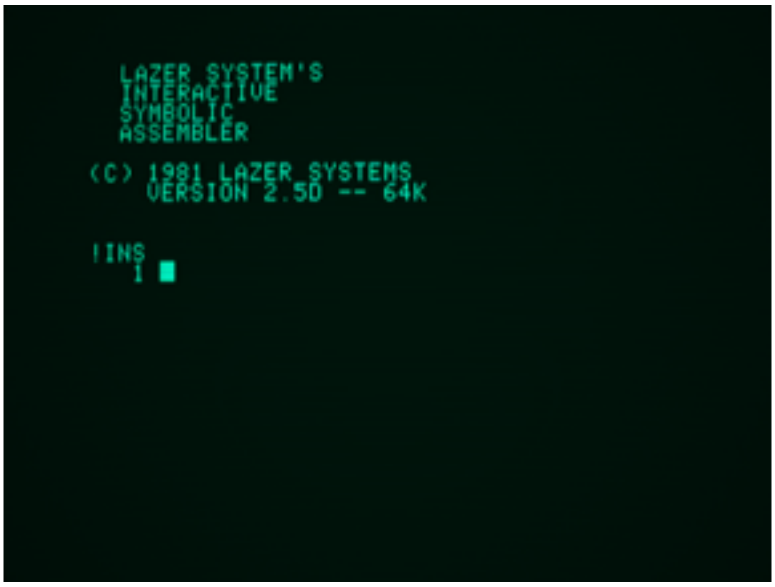
Decide exactly what program you wish to write. Once that first crucial decision has been taken, the next step is to decide how the program is to be written. Then the program has to be entered into LISA's text editor.

Boot your Apple II computer with the LISA 5.25" floppy disk inserted in your boot drive. After a couple of seconds you will be greeted by LISA's home screen.



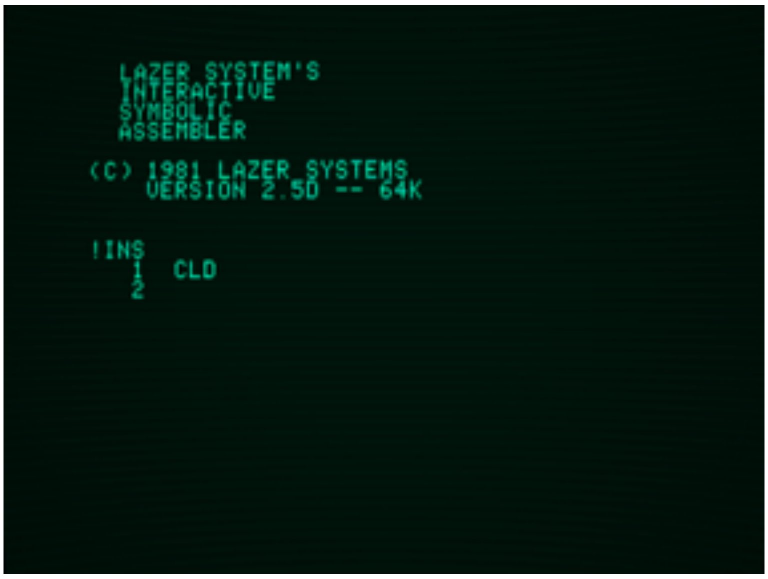
```
    LAZER SYSTEM'S
    INTERACTIVE
    SYMBOLIC
    ASSEMBLER

(C) 1981 LAZER SYSTEMS
     VERSION 2.5D -- 64K


!
```

You are now ready to start programming. To do this, type "INS" at the "!" prompt. After hitting return your screen should look something like this:



```
    LAZER SYSTEM'S
    INTERACTIVE
    SYMBOLIC
    ASSEMBLER

(C) 1981 LAZER SYSTEMS
     VERSION 2.5D -- 64K


!INS
   1
```

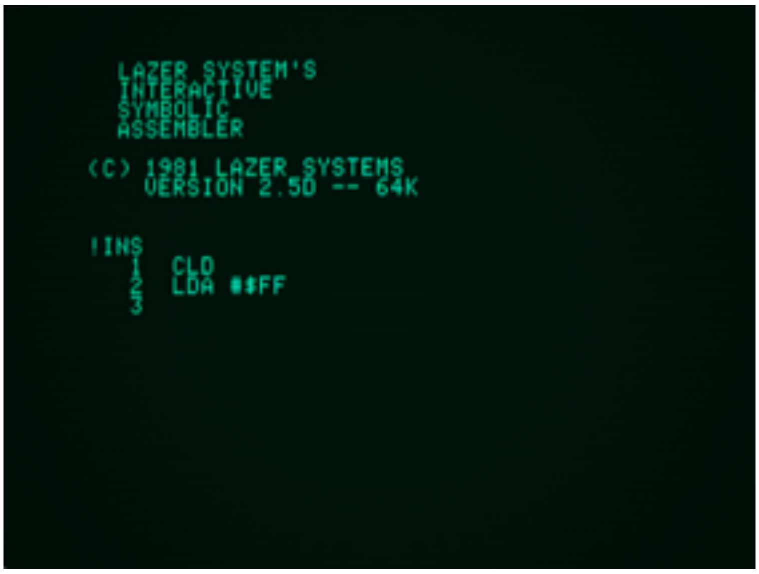
A blinking cursor will prompt you for text entry. As will all 6502 programs not utilizing decimal arithmetic, the first instruction should be a clear decimal flag instruction or "CLD".



```
LAZER SYSTEM'S
INTERACTIVE
SYMBOLIC
ASSEMBLER

(C) 1981 LAZER SYSTEMS
   VERSION 2.50 -- 64K

!INS
   1  CLD
   2
```

To enter this instruction type a space (remember, column one is reserved for labels) and the character sequence “CLD" followed by return.

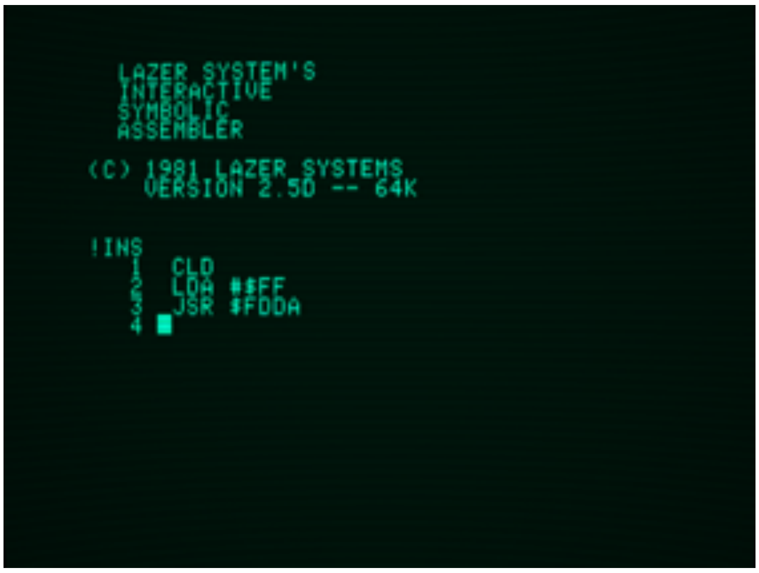
Once again the blinking cursor tells you that its waiting for text entry. The first thing we want to do is load the accumulator with the constant $FF. To do this we must use the assembly   language instruction "LDA #$FF" (or LDA OFF if you like). To enter this instruction type a space (remember column one....) followed by "LDA".



```
LAZER SYSTEM'S
INTERACTIVE
SYMBOLIC
ASSEMBLER

(C) 1981 LAZER SYSTEMS
   VERSION 2.50 -- 64K

!INS
   1  CLD
   2  LDA #$FF
   3
```

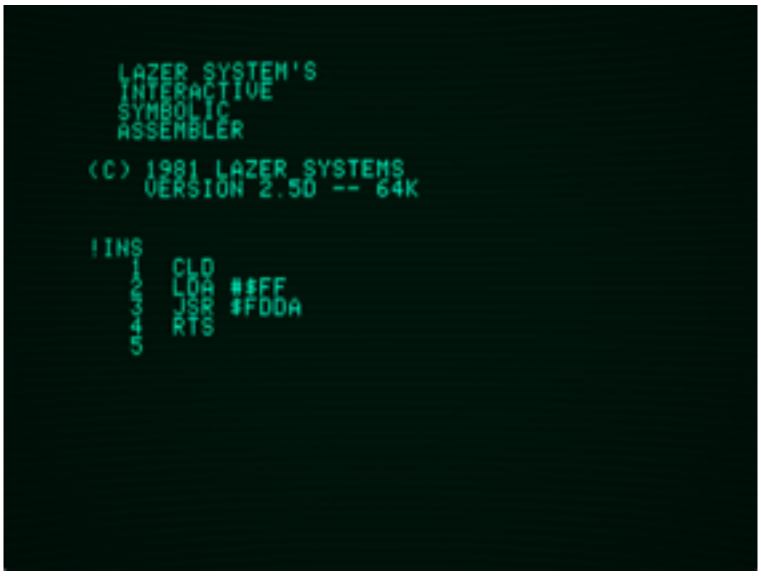
Now another space then "#$FF" followed by return.
Again the blinking cursor. Now we must enter the instruction which prints the contents of the accumulator as two hex digits. This happens to be "JSR" $FDDA”.

To enter, type space, followed by “JSR, followed by space, Followed by $FDAA", followed by return.



```
  LAZER SYSTEM'S
  INTERACTIVE
  SYMBOLIC
  ASSEMBLER

(C) 1981 LAZER SYSTEMS
    VERSION 2.5D -- 64K

!INS
   1   CLD
   2   LDA #$FF
   3   JSR $FDDA
   4 
```

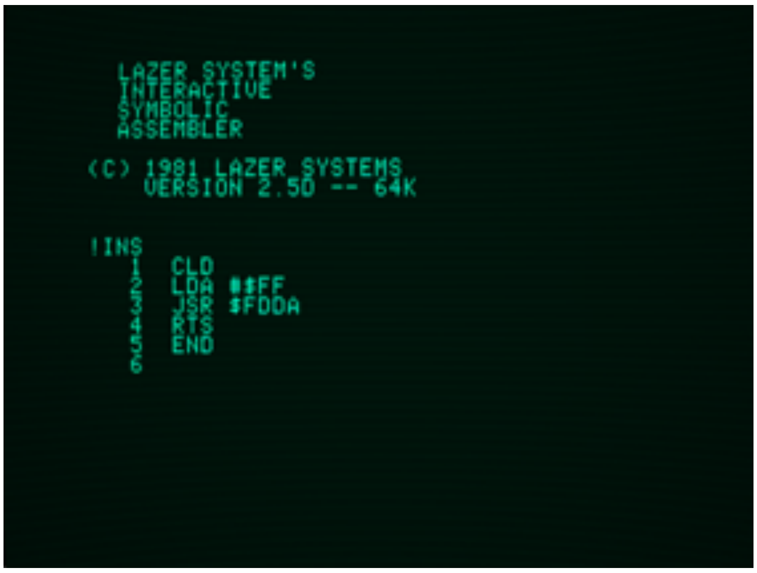
Now the command to return control to the monitor. Type a space followed "RTS", followed by return.



```
  LAZER SYSTEM'S
  INTERACTIVE
  SYMBOLIC
  ASSEMBLER

(C) 1981 LAZER SYSTEMS
    VERSION 2.5D -- 64K

!INS
   1   CLD
   2   LDA #$FF
   3   JSR $FDDA
   4   RTS
   5
```

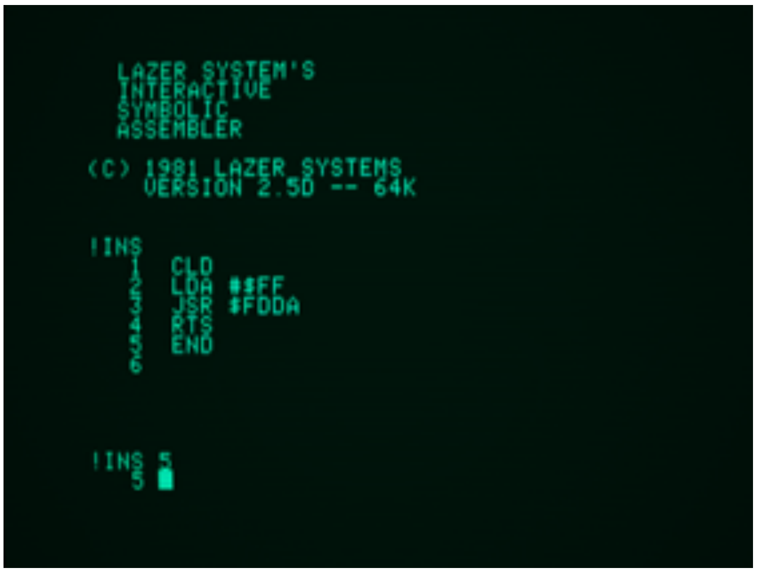
As far as the 6502 processor is concerned, our program is complete and can be run, but we still have to tell the LISA assembler that the end of the program has been reached.

Let’s add an END directive at the end of the program.



```
   LAZER SYSTEM'S
   INTERACTIVE
   SYMBOLIC
   ASSEMBLER

(C) 1981 LAZER SYSTEMS
  VERSION 2.5D -- 64K

!INS
   1   CLD
   2   LDA #$FF
   3   JSR $FDDA
   4   RTS
   5   END
   6
```

Note that more text is wanted (by line 6). To get out of text entry mode type control-E as the first character on line 6. Upon hitting return you will return to the command level.



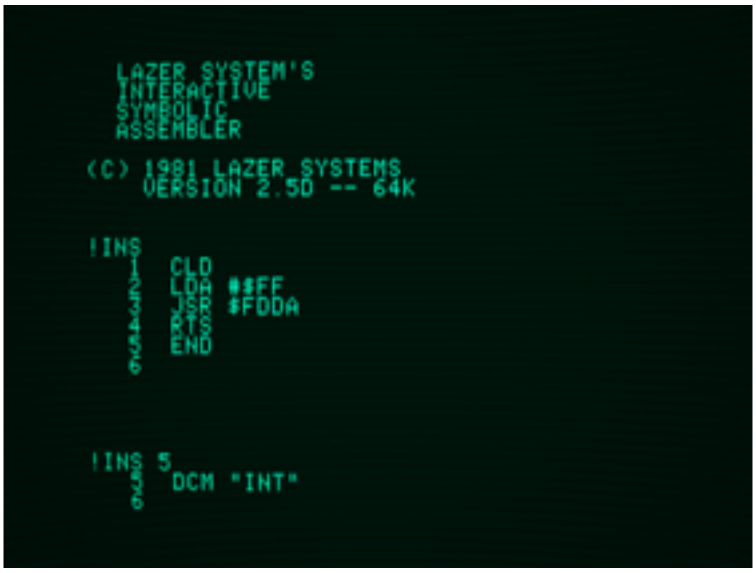
```
   LAZER SYSTEM'S
   INTERACTIVE
   SYMBOLIC
   ASSEMBLER

(C) 1981 LAZER SYSTEMS
  VERSION 2.5D -- 64K

!INS
   1   CLD
   2   LDA #$FF
   3   JSR $FDDA
   4   RTS
   5   END
   6


!INS 5
   5 █
```

At this point we are ready to assemble the program. However, since there are no labels in our program, printing the symbol table will prove to be just a big waste of time. Why don't we issue the opcode DCM "INT" just before the END so that the symbol table info will not be printed?

To do this type "INS 5" at the command level. This tells LISA to put you in "insert" mode with all text being inserted before line 5.

The blinking cursor reminds you that a text entry is required. Now type in a space, followed by DCM, followed by “INT", followed by return.

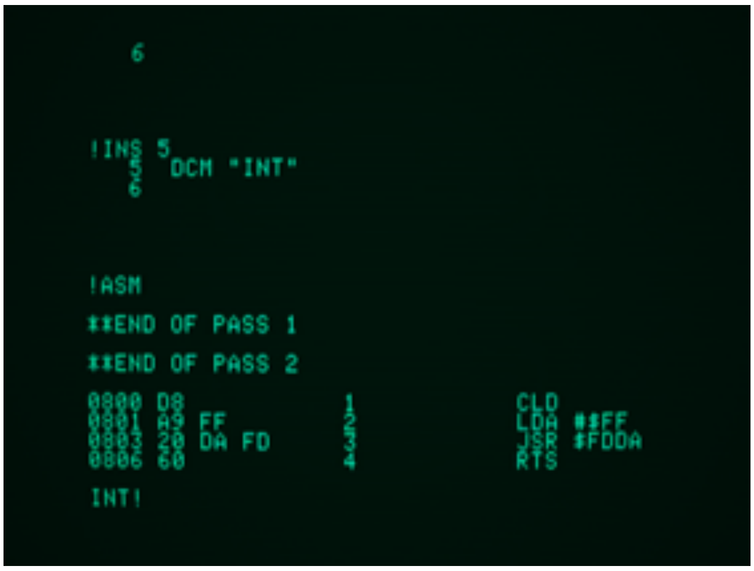
Type control-E on line 6 to end this correction. You have just completed your first assembly language program. Congratulations!



```
     LAZER SYSTEM'S
     INTERACTIVE
     SYMBOLIC
     ASSEMBLER

   (C) 1981 LAZER SYSTEMS
       VERSION 2.5D -- 64K

   !INS
      1   CLD
      2   LDA #$FF
      3   JSR $FDDA
      4   RTS
      5   END
      6


   !INS 5
      5   DCM "INT"
      6
```

You can use the L(IST) command to verify that your source code looks OK.

We must now assemble the text file before it can be run. To do this simply type ASM at the command level.

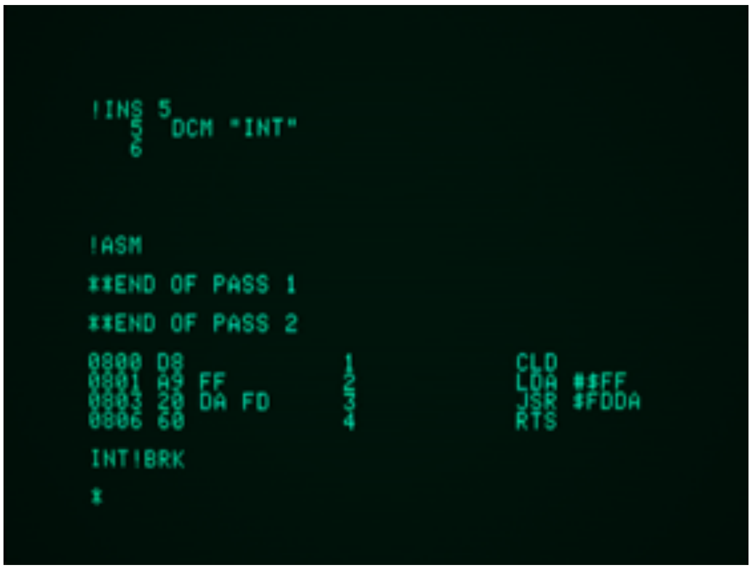


```
      6


   !INS 5
      5   DCM "INT"
      6


   !ASM
   **END OF PASS 1
   **END OF PASS 2
   0800 D8             1          CLD
   0801 A9 FF          2          LDA #$FF
   0803 20 DA FD       3          JSR $FDDA
   0806 60             4          RTS
   INT!
```

The first column is the hexadecimal address where the current instruction/code reside. The second field contains the opcodes produced by LISA. The third field is the line number followed by the source statement. Since we did not explicitly specify a program origin, LISA used the default of $800. Likewise, since we did not specify where the code was to be stored in memory, it was stored beginning at location $800.

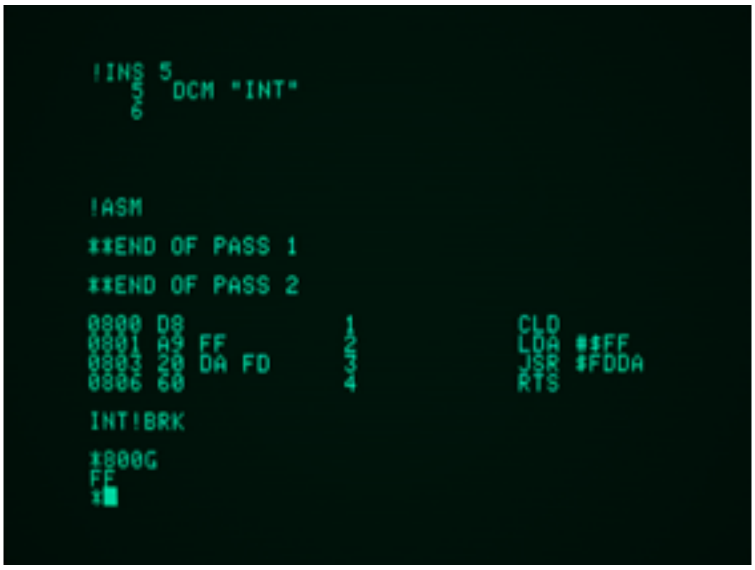
Since the program was assembled at $800, to run the program we must issue either a "CALL 2048" from BASIC or 800G from the monitor ($800= 2048 decimal).

To run the program you just wrote from the monitor, use the LISA command "BRK" (type BRK while in the command mode) and you will be placed in the monitor.



```
!INS 5
   5  DCM "INT"
   6


!ASM
**END OF PASS 1
**END OF PASS 2
0800 D8           1          CLD
0801 A9 FF        2          LDA #$FF
0803 20 DA FD     3          JSR $FDDA
0806 60           4          RTS
INT!BRK
*
```

When you get the "*" prompt (signaling that you entered the monitor), type 800G followed by return. The screen will look now like this:



```
!INS 5
   5  DCM "INT"
   6


!ASM
**END OF PASS 1
**END OF PASS 2
0800 D8           1          CLD
0801 A9 FF        2          LDA #$FF
0803 20 DA FD     3          JSR $FDDA
0806 60           4          RTS
INT!BRK
*800G
FF
*█
```

The FF was the printed result of our program and we were returned to the monitor ("*"). If you wish to return to LISA type, “E000G" (“cold start”) or "E003G" (“warm start”).

Although you don't have to, programmers make sure that the first line in their program is either the first line to be executed, or a JMP to the first line to be executed. This makes it easy to determine the starting address of the program. If you plan to go back and forth between your program & LISA, make sure that you do not utilize the zero page locations used by LISA. Unpredictable things may happen. Before running your program, ALWAYS make sure that you have saved your latest version to disk, its too easy for a bug in your program to wipe out everything in memory.

Software provided with LISA 2.5

The LZR IOS routines (LISA P2.L)
A collection of various input/output routines & several utility routines. Used for I/O handling inside LISA.

Randy's HIRES routines
Handy hires graphic routines intended for assembly language programing. Intended ranges is $1000-$2000. Since the routines contain a considerable amount of internal docs there was not enough room to hold the entire text file in memory all at once. As a result "HIRES.2" gets chained in during assembly. LO(AD) RANDY'S HIRES ROUTINES & then A(SM) to text file. An "ICL" pseudo opcode at the end of RANDY'S HIRES ROUTINES automatically does the rest.

Using "SOFTWARE TOOLS"
This is a collection of routines is loosely modeled after the routines found in the book "SOFTWARE TOOLS" by Kernighan & Plauger

XREF/65
A general purpose cross reference generator for LISA 1.5F and LISA 2.x. To use you must have Applesoft ROM or 16K RAM card, also a printer interface installed in slot one, & 48K or larger. Make sure program is without errors if not complete insert dummy labels.

Before cross reference generate a TEXT type file of your LISA program. L)oad then W)rite. Boot XREF/65 disk and “RUN XREF/65".

SCTOLISA
Converts SC ASSEMBLER II 3.2 to LISA. To execute this program, “BRUN SCTOLISA” and then enter the SC ASSEMBLER II filename. After pushing return, the file will be converted to the LISA format. The output file is written to disk as a TEXT type file named SC-CONVERTED which may be EXEC'd into LISA. Other problems… SCTOLISA does not handle certain types of address expressions very well.

SORT 2.0
A symbol table sort routine provided for 2.2 users. BRUN immediately after an assembly in order to print a sorted (numeric & alphabetic) symbol table listing.

DISASM/65
This produces a symbolic disassembly listing. You can relocate programs, see how other programs are written, patch up programs.DISAM/65 allows the user to specify hexadecimal, ASCII, address, and pushed address data types. To run- “BRUN DISASM/65". At the "ENTER RANGE OF DISASSEMBLY” prompt, answer the beginning & ending addresses of the section of code you wish to disassemble.

HIRES ROUTINES
These routines are intended for assembly language programmers only. For the most part, these cannot be called from INTEGER BASIC, APPLESOFT, OR TINY PASCAL.

Being easy to use the routines are very fast. The subroutine package includes routines which plot a point, erase a point, perform base calculations, draw a line, create a shape matrix, detect if any dots are ON in a specified range, clear the hires screen, turn on the graphics, draw a picture, "OR" a picture to the screen, erase a picture from the screen, "XOR" a picture to the screen, erase a line, and set parameters. A SYMBOL routine allows you to draw any of the 96 printable ASCII characters onto the screen, in a format of 46x30. Most of the routines in HIRES work on the principle of NXM picture.

Name	Subroutine description

RADAR	Used to determine whether or not any bits (i.e. dots) are ON within a specified range. The calling sequence for RADAR is:
	`JSR RADAR`
	`HEX WIDTH`
	`HEX HEIGHT`

DRAWLN	Draws a line on the HIRES screen
ERSLN	Ersases a line on the HIRES screen
DXY	Allows the programmer to initialize
SYMBOL	Allows the user to display ASCII characters
PICTURE	Points to the ASCII representation of the picture
PIX	Praws the picture verbatim onto the screen
ORPIX	Similar to PIX except the picture is OR’d onto the screen instead of drawn verbatim
ANDDIX	Used to erase a picture on the screen
XORPIX	Exclusive-OR's the picture to the screen allowing interesting effects

APPENDICES

Appendix A
LISA Memory usage

Memory range	Usage
$0050-$00AF	Page 0 memory locations
$00E0-$00FF	Page 0 memory locations
$0200-02FF	Reserved for I/O buffer
$0360-$03FF	Open for user subroutines
$0800-$1800	Space that can be used for user generated code
$1800-$7FFF	Reserved for the source code text file
$8000-$94FF	Reserved for the symbol table
$9500-$95FF	Reserved for I/O buffer
$D000-$F7FF	Reserved for LISA

LISA modifies the Apple DOS so if you want to return to BASIC or APPLESOFT, REBOOT a different disk containing the language. LISA requires that DOS 3.2 be used. LISA may be converted to DOS 3.3 by using the "MUFFIN" program.

Address	Description
$E000	COLDSTART (CONTROL-B)
$E003	WARMSTART (CONTROL-C)
$E006	INSERT ENTRY
$E009	USER COMMAND
$E01B	HIGH ORDER BYTE OF TEXT FILE STARTING ADDRESS
$E01C	HIGH ORDER BYTE OF SYMBOL TABLE STARTING ADDRESS
$E01D	MNEMONIC TABLE ADDRESS
$E01F	LINES/PAGE
$E020	TITLE BOOLEAN 0=NO TTL PSEUDO OP. 1=TTL PSEUDO OP
$E024	CLOCK SLOT#
$E025	CLOCK Cn05 VALUE
$E026	CLOCK Cn07 VALUE
$E027	END OF SYMBOL TABLE ADDRESS
$E029	USR PSEUDO-OP JSR'S HERE

NOTE ** for LISA 48K systems subtract $8000 from the above addresses. Control-C & Control-B do not work for 48K systems. You must type "$6000G" or "$6003G" to cold or warm start LISA 48K.

Appendix B
Summary of LISA Commands

Optional info & parameters in brackets

Command	Description
I(NSERT)	Inserts text after the last line of the text file
I(NSERT) {ln#}	Inserts text before line number "ln#"
D(ELETE) ln#	Deletes line in range specified
D(ELETE) ln#{,ln#}	Deletes line in range specified
L(IST)	List the entire file
L(IST) {ln#)	Lists a single line
L(IST) {ln#,ln#}	Lists lines in range specified
LO(AD)	Loads text file from cassette
LO(AD) {filename}	Loads file from disk
SA(VE)	Saves file to cassette
SA(VE) {filename}	Saves file to disk
AP(PEND)	Loads a file from tape and appends it to text file in memory
AP(PEND) {filename}	Loads a file from disk and appends it to the file in memory
Control-D	Execute DOS command directly from LISA's command level
Control-P	Jump to user defined routine
LE(NGTH)	Prints the length (in hex) of the current text file
A(SM)	Assembles current text file
BRK	Breaks to Apple monitor
F(IND)	Searches for the specified label

Appendix C
Explanation of all errors that LISA can produce

Command Level

Message	Description
ILLEGAL COMMAND	User typed in an unrecognizable command at the command level (usually a typo).
ILLEGAL LINE #	A digit was expected but not found

Edit Time

Message	Description
ILLEGAL SYMBOL IN LABEL	An illegal symbol was detected in the label field. Possible a control character.
LABEL TO LONG	A label appeared in the label field which contained more than 6 characters. If 7th or longer in the operand field.
ILLEGAL MNEMONIC	Symbol in the mnemonic filed is not a valid LISA mnemonic. Causes-- beginning the mnemonic in column one, beginning the label in other than column one, the mnemonic is missing (perhaps you forgot to delimit a label with a ":"?), or a simple typo.
ILLEGAL ADDRESSING MODE	Attempted to use an addressing mode which is not possible for the intended instruction.
ILLEGAL OPERAND	General catch-all for syntax errors in the operand field.
NOT ENOUGH DIGITS	Occurs when a hex string is entered with an odd number of digits. It take two hex digits to equal one byte and leading zero's must be typed into the HEX string.
ILLEGAL HEX DIGIT	A hex digit was expected, but not found (remember, in hex strings you don't use "$").
ILLEGAL BLANK IN OPERAND FIELD	The first non-blank character in the operand filed after the first blank detected was not ";" or return.
ILLEGAL CHARACTER IN STRING	Occurs when user did not enclose the entire string in quotes, or attempted to use the " or ' characters without doubling them up.
STRING ERROR	Usually occurs when the string does not have closing quotes.

Assembly Time Pass One

Message	Description
VALUE EXCEEDS $FF	User attempted to define a zero page location using EPZ, but the value of the expression exceeded $FF.
ILLEGAL EXPRESSION	The address expression in the operand field of the EPZ opcode is invalid.
MISSING 'END'	The end of the text file was encountered but no END opcode was present. This error is an automatic abort.
DUPLICATE LABEL	Zero page variable was previously defined
ILLEGAL OPERAND IN ADDRESS FIELD	An illegal quantity was present in the address field ("*" is not allowed in EPZ statements).
SYMBOL WAS NOT PREVIOUSLY DEFINED IN AN EPZ STATEMENT	Occurs when a symbolic reference is made in the address expression, but the label was not previously defined.

Assembly Time Pass Two

Message	Description
DUPLICATE LABEL	User attempted to redefine a previously defined label.
UNDEFINED SYMBOL/ILLEGAL ADDRESS	Occurs when a symbol is found which is not defined in the program.
ILLEGAL FORWARD REFERENCE	Occurs when the user attempts to use a symbolic reference in the address expression of an EQU opcode which has not been previously defined.
EQU W/O LABEL	An equate was encountered, but no label was present.
STY ABS,X NOT ALLOWED	User attempted to use an illega addressing mode.
ABS,Y NOT ALLOWED	User attempted to use an illegal addressing mode, variable must be zero page.
** DAMAGE ** ILLEGAL CHARACTER IN OPERAND	This usually occurs when part of the text file has been destroyed (watch OBJ's & ORG).

Assembly Time Pass Three

Message	Description
UNDEFINDED SYMBOL	Label in expression field was not defined anywhere else in the program.
BRANCH OUT OF RANGE	Relative addressing only allows a range of -126 to +129:, this range was exceeded.
UNDEFINED SYMBOL - MUST BE ZPAGE	An undefined symbol was encountered. This particular symbol, due to its usage, must be a zero page variable.
ADDRESSING MODE REQUIRES ZPAGE VARIABLE	User attempted to use an absolute location where a zero page location is required.

Appendix D
USR Pseudo Opcode

LISA 2.5 supports a special, user-definable pseudo opcode. This is a no hold barred, always syntactically correct, user assignable pseudo opcode. Whenever USR opcode is encountered, a JSR to location $E029 is performed. Normally there is an RTS instruction & two NOP instructions at location $E029. You can, replace these three bytes with a jump to your pseudo opcode handler. In order to perform some operations you may need to call some of the routines within the LISA package. Three files in the disk,

LISA P1.L
LISA P2.L
LISA P6.TXT

contain sources for portions of LISA; you may take a look at these files and call any routines within them. Other useful routines are:

ERRR ($EBE6)
Prints an error message and gives the user the chance to abort assembly. The calling sequence is:

```
JSR  ERRR
ASC  'ERROR MESSAGE’
HEX  00
```

If the user wishes to continue, control will be returned to your program after the HEX 00 statement.

GETADR ($EBE9)
PNTR points at the beginning of the current line, the Y register contains an index into current line. GETADR converts the address expression found at the spot in the line pointed at by the Y register into a 16-bit value which is returned in location SADR & SADR+1. Carry is returned clear if there was an error, set if there was no error. Two locations are of interest after a call to GETADR:

EYET
If true, specifies an absolute address expression, if false, zero page address expression.

BIT #6 of AFND:
Is set if symbolic labels were pre-declared. If there are any forward references (during pass 2, at pass 3 they are undeclared symbols) then AFND has a zero in bit six.

SYMLUP ($EBEC)
Looks up the symbol pointed at by PNTR and the Y register. EYET should be set to zero before calling SYMLUP. On return, EYET will be set to one if the symbol is not zero page and the carry flag will be set if the symbol was found in the symbol tab.

Useful variables:

Name	Location	Description
PNTR	$EA	Points at the beginning of the current line
LNUM	$F2	Hold line number of current line
SADR	$91	Value of symbol or address expression returned here
EYET	$97	Returned with 0 if zero page value, one otherwise
LOCC	$99	Location counter.Contains the current program counter address
CODE	$9B	Code counter. Contains the address of where the next byte of object code is to be stored.
AFND	$9F	Returned with $FF if all symbols in address expression were defined
PRTR	$A1	Set to true (1) if LST option is in effect, false (0) if NLS option in effect
CDSP	$A2	Contains the number of bytes of object code output by this operation. Should be set to zero if no code is output
PASS	$BC	Contains 0 if this is pass two, contains 1 if this is pass three (USR address at $E029 is not called during pass one)
FNAME	$2C0	Contain the filename of the current assembling file
CODSAV	$2E0-$35F	Output code buffer, each byte of output object code is output to this buffer by successive calls to OUTC

Upon calling the routine at $E029 several conditions exist. First, if there was a symbol at the beginning of the line it was entered into the symbol table (during pass two, at pass three the symbol is ignored), PNTR points at the beginning of the line, and the Y register points at the USR mnemonic token. To point the Y register at the operand field simply increment it by two. Now you can parse the operand field and do whatever you please with it. The operand field is terminated with a carriage return ($0D). You must terminate your user routine with a BIT $C080 and an RTS instruction.

Appendix E
ASCII Character Set

DEC	HEX	Symbol	Key	Meaning	DEC	HEX	Symbol	Key	Meaning
000	000	NUL	P (c,s)	NULL	064	040	@	@	Commercial At
001	001	SOH	A (c)	Start of Heading	065	041	A	A	A
002	002	STX	B (c)	Start of Text	066	042	B	B	B
003	003	ETX	C (c)	End of Text	067	043	C	C	C
004	004	EOT	D (c)	End of Transmission	068	044	D	D	D
005	005	ENQ	E (c)	Enquiry	069	045	E	E	E
006	006	ACK	F (c)	Acknowledge	070	046	F	F	F
007	007	BRL	G (c)	BELL	071	047	G	G	G
008	008	BS	H (c)	Backspace	072	048	H	H	H
009	009	HT	I (c)	Horizontal Tab	073	049	I	I	I
010	00A	LF	J (c)	Line Feed (New Line)	074	04A	J	J	J
011	00B	VT	K (c)	Vertical Tab	075	04B	K	K	K
012	00C	FF	L (c)	Form Feed (Top of form)	076	04C	L	L	L
013	00D	CR	M (c)	Carriage Return (Return)	077	04D	M	M	M
014	00E	SO	N (c)	Shift Out	078	04E	N	N	N
015	00F	SI	O (c)	Shift In	079	04F	O	O	O
016	010	DLE	P (c)	Data Line Escape	080	050	P	P	P
017	011	DC1	Q (c)	Device Control 1	081	051	Q	Q	Q
018	012	DC2	R (c)	Device Control 2	082	052	R	R	R
019	013	DC3	S (c)	Device Control 3	083	053	S	S	S
020	014	DC4	T (c)	Device Control 4	084	054	T	T	T
021	015	NAK	U (c)	Negative Acknowledge	085	055	U	U	U
022	016	SYN	V (c)	Synchronous Idle	086	056	V	V	V
023	017	ETB	W (c)	End of Transmission Block	087	057	W	W	W
024	018	CAN	X (c)	Cancel	088	058	X	X	X
025	019	EM	Y (c)	End of Medium	089	059	Y	Y	Y
026	01A	SUB	Z (c)	Substitute	090	05A	Z	Z	Z
027	01B	ESC	K (c,s)	Escape	091	05B	[	[	Opening Bracket
028	01C	FS	L (c,s)	File Separator	092	05C	\	\	Reverse Slant
029	01D	GS	M (c,s)	Group Separator	093	05D	]	]	Closing Bracket
030	01E	RS	N (c,s)	Record Separator	094	05E	^	^	Up Arrow
031	01F	US	O (c,s)	Unit Separator	095	05F	_	_	Underline (back arrow)
032	020	SP	Space	Space (Blank)	096	060	‘	‘	Grave Accent
033	021	!	!	Exclamation Point	097	061	a	a	a
034	022	"	"	Quotation Mark (Double)	098	062	b	b	b
035	023	#	#	Number Sign	099	063	c	c	c
036	024	$	$	Dollar Sign	100	064	d	d	d
037	025	%	%	Percent Sign	101	065	e	e	e
038	026	&	&	Ampersand	102	066	f	f	f

DEC	HEX	Symbol	Key	Meaning	DEC	HEX	Symbol	Key	Meaning
039	027	‘	‘	Apostrophe (Single)	103	067	g	g	g
040	028	(	(	Opening Parenthesis	104	068	h	h	h
041	029	)	)	Closing Parenthesis	105	069	i	i	i
042	02A	*	*	Asterisk	106	06A	j	j	j
043	02B	+	+	Plus	107	06B	k	k	k
044	02C	,	,	Comma	108	06C	l	l	l
045	02D	-	-	Hyphen (Minus)	109	06D	m	m	m
046	02E	.	.	Period (Decimal Point)	110	06E	n	n	n
047	02F	/	/	Slant	111	06F	o	o	o
048	030	0	0	0 (Zero)	112	070	p	p	p
049	031	1	1	1 (One)	113	071	q	q	q
050	032	2	2	2	114	072	r	r	r
051	033	3	3	3	115	073	s	s	s
052	034	4	4	4	116	074	t	t	t
053	035	5	5	5	117	075	u	u	u
054	036	6	6	6	118	076	v	v	v
055	037	7	7	7	119	077	w	w	w
056	038	8	8	8	120	078	x	x	x
057	039	9	9	9	121	079	y	y	y
058	03A	:	:	Colon	122	07A	z	z	z
059	03B	;	;	Semicolon	123	07B	{	{	Opening Brace
060	03C	<	<	Less Than	124	07C	|	|	Vertical Line
061	03D	=	=	Equal	125	07D	}	}	Closing Brace
062	03E	>	>	Greater Than	126	07E	~	~	Overline (Tilde)
063	03F	?	?	Question Mark	127	07F		DELETE	Delete (Rubout)

c = Control key   s = Shift key

Appendix F
Mnemonics

Opcode	Description	Opcode	Description
ADC	ADd with Carry	JSR	Jump to SubRoutine
AND	AND Accumulator with	LDA	LoaD Accumulator
ASL	Arithmetic Shift Left	LDX	LoaD index X
BCC	Branch on Carry Clear (BLT)	LDY	LoaD index Y
BCS	Branch on Carry Set (BGE)	LSR	Logical Shift Right
BEQ	Branch on EQual	NOP	NO oPeration
BIT	Bit Test	ORA	OR Accumulator with
BMI	Branch on MInus	PHA	PusH Accumulator on stack
BNE	Branch on Not Equal	PHP	PusH Processor status bits on stack
BPL	Branch on PLus	PLA	PulL Accumulator from stack
BRK	BReaK	PLP	PulL Processor status bits from stack
BVC	Branch on oVerflow Clear	ROL	ROtate Left
BVS	Branch on oVerflow Set	ROR	ROtate Right
CLC	CLear Carry flag	RTI	ReTurn from Interrupt
CLD	CLear Decimal mode	RTS	ReTurn from Subroutine
CLI	CLear Interrupt disable flag	SBC	SuBtract from accumulator with Carry
CLV	CLear oVerflow flag	SEC	SEt Carry flag
CMP	CoMPare accumulator with	SEI	SEt Interrupt disable flag
CPX	CoMPare X with	STA	STore Accumulator in memory
CPY	CoMPare Y with	STX	STore index X in memory
DEC	DECrement memory	STY	STore index Y in memory
DEX	DECrement Index X	TAX	Transfer Accumulator to index X
DEY	DECrement Index Y	TAY	Transfer Accumulator to index Y
EOR	Exclusive OR accumulator with	TSX	Transfer Stack pointer to index X
INC	INCrement memory	TXA	Transfer index X to Accumulator
INX	INcrement index X	TXS	Transfer index X to Stack pointer
INY	INcrement index Y	TYA	Transfer index Y to Accumulator
JMP	JuMP to new location		