# LISA

## A PROFESSIONAL ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM FOR APPLE COMPUTERS.

IF YOUR DISK SHOULD EVER FAIL FOR ANY REASON

Within 90 days of purchase send your Lisa disk for free replacement. Thereafter, please accompany the diskette with five dollars to cover shipping and handling.

CONTENTS

# SECTION 1.0 — INTRODUCTION

1) WHAT IS LISA?

2) WHAT DOES LISA REQUIRE?

3) WHAT DISK FACILITIES DOES LISA PROVIDE?

4) HOW DOES LISA INTERFACE WITH THE APPLE MONITOR AND DOS?

## SECTION 1.1 — WHAT IS LISA?

LISA (pronounced LI ZA, not LE SA) is an interactive 6502 assembler for the Apple II. It was carefully designed to suit the needs of beginners and advanced programmers alike. 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, as is the case with several other available assemblers. LISA is a symbolic assembler, the programmer does not have to keep track of addresses as with the built-in ROM mini-assembler. LISA has more built-in features than any other assembler available for the Apple II! More pseudo opcodes (which make the assembler easier to use), Sweet 16 mnemonics (which turn your Apple II into a 16-bit machine, requiring less code to perform a desired task), more extended mnemonics (a great memory aid), 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.

## SECTION 1.2 — WHAT DOES LISA REQUIRE?

LISA is a disk based product. Minimum requirements include at least one disk drive and 48K bytes of RAM. LISA 2.5 64K requires a language card for proper operation. Since the majority of LISA 2.5 owners have a RAM card of some type this documentation will be directed primarily at those individuals who have a 64K Apple. There are no syntactical differences between

LISA 48K and LISA 64K, only addresses and the amount of available memory are changed between the two versions. The appendicies contain a special section pointing out the differences. Apple owners with only 48K should study this section carefully.

The Lazer MicroSystems Lower Case +Plus is optional, but comes highly recommended. Along with the Lazer MicroSystems' Lower Case +Plus you will probably want the Keyboard +Plus as well. It's built-in type-ahead buffer is extremely useful when editing large programs. A printer (80 columns optimal) is also optional but comes highly recommended.

## SECTION 1.3 — WHAT DISK FACILITIES DOES LISA PROVIDE?

LISA provides the use of several disk options. The user 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 incompatable with the rest of the world. Text files may be read in by Apple PIE or your BASIC programs but are much slower than "LISA" type files for simple loading and saving. In addition a LISA source file on diskette 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 onto the disk, so codefiles of almost any length may be generated.

## SECTION 1.4 — HOW DOES LISA INTERFACE WITH THE APPLE MONITOR AND 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 for 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 a filetype of "L". When running under an unmodified DOS these files will look like binary files, but they cannot be BLOADED or BRUN'd. LISA is provided on DOS 3.2 but may be converted to DOS 3.3 using the DOS 3.3 MUFFIN program.

## SECTION 2.0 — 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

## SECTION 2.1 — ASSEMBLY LANGUAGE SOURCE FORMAT

Source statements in LISA are entered in a "free format" fashion. Source statements in LISA 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, and 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).

## SECTION 2.2 — 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 execpt 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 compatability 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 then a 6502 mnemonic must appear after the label. If a colon terminates the line then 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.

## SECTION 2.3 — 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.

VALID MNEMONICS:

```
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:

```
BTR BFL BGE BLT XOR
```

SWEET-16 MNEMONICS:

```
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:

```
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 appear in the following sections.

## SECTION 2.4 — THE OPERAND FIELD

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

## SECTION 2.5 — 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 a line by themselves by placing a semicolon or an asterisk in column one.

SECTION 3.0 — ADDRESSING MODES



SECTION 3.1 — ADDRESS EXPRESSIONS

The operand field 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 "+", "-", "*", "/", "&", "|", "^", "=", 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 off) 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.

EXAMPLES 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 than 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 than $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 dependant. If it is found between two expressions, 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.

## SECTION 3.2 — 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 that "/" 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 dicussing 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 the 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 #$10
0FF  -SAME AS #$FF
```

WARNING: These special forms of the immediate addressing mode were included to provide compatability 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 compatability with older versions of LISA. DON'T USE THESE FORMS IN NEW PROGRAMS YOU WRITE, or someday...

## SECTION 3.3 — ACCUMULATOR ADDRESSING MODE

The accumulator addressing mode applies to the 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 compatable with the mini-assembler built into the Apple and as such the "A" in the operand field is not required.

EXAMPLES OF THE ACCUMULATOR ADDRESSING MODE:

```
ASL
ROL
LSR
ROR
```

## SECTION 3.4 — 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 descriptions).

EXAMPLES:

```
LDA LABEL      -SYMBOLIC LABEL USED
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
```

## SECTION 3.5 — 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
```

## SECTION 3.6 — 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
```

## SECTION 3.7 — RELATIVE ADDRESSING

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

EXAMPLES:

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

## SECTION 3.8 — 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
```

## SECTION 3.9 — 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
```

## SECTION 3.10 — INDEXED BY X, INDIRECT ADDRESSING

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

( <expression>,X )

<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)
```

## SECTION 3.11 -- INDIRECT ADDRESSING

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

( <expression> )

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

EXAMPLES:

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

## SECTION 3.12 — 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 textfile. 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.

SECTION 4.0 — USING LISA



SECTION 4.1 — GETTING LISA UP AND RUNNING

To run LISA simply boot the disk provided. When LISA is ready to execute a command you will be greeted with a "!" prompt (the same one used by the mini-assembler, in case you're wondering).

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 "warmstart" operation) use the "E003G" monitor command or control-C. See the section on "warnings and extraneous notes" for the warmstart procedure.

SECTION 4.2 — 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 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 parenthesis. As an example you only need type "LO" to perform the "LOAD" command, "I" to execute the "INSERT" command, etc.

## SECTION 4.3 — EXPLANATION OF COMMANDS

NOTE: Optional information is enclosed in parenthesis.
Optional parameters are enclosed in braces.

### 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 than 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 numbering scheme. The first line in the text file is line number one, the second line is line number two, the third line is line 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 when you insert text into LISA, boot your disk and get into the command interpreter. Once you're in the 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 into 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 verfy by noticing the "!" prompt.

Now type "L" followed by a return ("L" is for "LIST"). The following should be displayed on the CRT screen:

```
1 LBL    LDA 00
2        END
```

Now type "I 2" at the command level. LISA will respond with the line number two and will once again await your 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 spaces) followed by return. When "3" appears prompting you to enter a new line press control-E.

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 2-6
```

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 which greatly facilitate searching for a specified line. If, during listing, you press the space bar then 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 ENTIRE FILE
LIST 2     -LISTS LINE # 2
LIST 2,6   -LISTS LINES 2-6
```

LO(AD) filename

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

EXAMPLES:

```
LOAD LZR IOS  -LOADS LZR IOS FROM DISKETTE
```

NOTE: Although the command "LOAD" is begin used this does not mean that LISA uses the DOS LOAD command. Internally (and before DOS gets a chance to see it) "LOAD" is converted to "BLOAD".

SA(VE) filename

The file in memory is saved to diskette under the specified filename. Once again, 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 SA(VE) command are saved as special "L" type files.

EXAMPLES:

```
SAVE TEMP        -SAVES TEXT FILE TO DISKETTE
SAVE TEMP,S6,D2
```

AP(PEND) filename

Reads in a text file from diskette and appends it to the existing text file in memory.

EXAMPLES:

```
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, of course), LISA will jump to your particular routine. To return to the LISA 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 onto diskette as a TEXT type file. This allows you to manipulate your LISA text file with a BASIC or APPLESOFT program. In addition, TEXT type files may be read into Apple PIE (version 2.0 or greater) and you can modify your LISA 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 file you may use the DOS "EXEC" command to reload these TEXT type files back into LISA (See the control-D command for more info on this feature).

LE(NGTH)

Displays the current length of the LISA 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 turns on an input device.
^D INT does not put you into BASIC, but rather returns you to LISA.
^D EXEC filename - where filename 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 and enters the Apple monitor.

F(IND) label

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.

## SECTION 4.4 — LISA SCREEN EDITING FEATURES

LISA incorporates several nice screen editing commands. Users of the Apple II without the Autostart ROM will be able to appreciate the convenience associated with the LISA screen editing commands.

To move the cursor up press control-O. To move the cursor to the right press control-K. To move the cursor down press control-L. To move the cursor to the left press control-J. To copy the character under the cursor press the right arrow (control-U). To delete the previously entered character press the back arrow (baskspace/control-H).

## SECTION 4.5 — LISA LOWER CASE FACILITIES

It is possible to enter lower case (as well as several special characters) into LISA directly from the keyboard. The display of lower case requires the Lazer MicroSystems' Lower Case +Plus" which is available directly from Lazer MicroSystems. If you do not have the lower case adapter on your Apple, lower case letters will appear as garbage on the Apple video screen, however, they are still lower case in memory so if you dump a listing to a printer with lower case capabilities it will be printed in lower case.

Normally, when moving the cursor over a lower case letter junk will be displayed on the screen since there are no inverted or blinking lower case letters in the Apple's character set. In order to improve legibility, whenever you move the cursor over a lower case letter it will be displayed as blinking (or possibly inverted) upper case letter. You can use this facility to double check lower case entry if your Apple does not have the lower case

adapter.

Since the Apple's shift key does not function as a shift key for input, a software shift key has to be used. LISA uses the ESC key as shift key for input. LISA also has a "caps lock" mode, since you use upper case most of the time anyway. The caps lock mode is toggled by pressing control-S. While in the upper case mode the cursor will be the normal blinking cursor, while in the lower case mode the cursor will not be blinking but rather a static inverted character. Naturally, while in the caps lock mode the ESC key will do absolutely nothing for an alphabetic character.

When you purchase and install the lower case adapter you also get several special characters added to the basic Apple display capabilities. These characters include: "|", "~", "{", "}", and "^". With LISA these characters, as well as "[", "]", and "_" can be entered directly from the keyboard. To enter one of these special characters you must first press ESC and then one of following keys:

"|" by pressing "!" or "1"
"~" by pressing "^" or "N"
"^" by pressing "'" or "7"
"{" by pressing "(" or "8"
"}" by pressing ")" or "9"
"[" by pressing "<" or ","
"]" by pressing ">" or "."
"_" by pressing "-"
"\" by pressing "/"
DEL which prints a funny looking box on the screen (but not on the printer) by pressing "#" or "3".

NOTE: If you have installed a Lazer MicroSystems' Keyboard +Plus, check the manual for special details on entering these characters.

# SECTION 5.0 -- ASSEMBLER DIRECTIVES/PSEUDO OPCODES



## SECTION 5.1 -- THE AVAILABLE PSEUDO OPCODES

As much as the 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. LISA probably has the most powerful and flexible pseudo opcode set available on any 6502 assembler.

The pseudo opcodes available are:

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

.FI =

SECTION 5.2 — 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 may also be stored directly in memory thus allowing the source file to be assembled at full speed. LISA, except in certain cases, always stores the assembled program into RAM memory. The question is where? Well, 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, however, 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 in 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 (see ORG below) and store it in another. Another use of the OBJ pseudo opcode is to reuse memory in a limited memory environment. For instance, suppose you wish to assemble a text file 10K bytes long. Unfortunately LISA does not leave you 10K free for this use (LISA allows only 4K). What you can do is assemble the first 4K of code and then save this first portion of code to disk (see "DCM" below). Now, by using the OBJ pseudo opcode, 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 instance where one would use the pseudo opcode 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.

No problem! Simply use the OBJ pseudo opcode 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 pseudo opcode 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.

## SECTION 5.3 — ORG: PROGRAM ORIGIN

SYNTAX: ORG <expression>

When an ORG pseudo opcode is encountered LISA begins generating code for the address specified in the address expression. When you use an ORG pseudo opcode, you are making a promise to LISA 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 an ORG pseudo opcode 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 of $800 will be used.

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 pseudo opcodes that reserve memory, etc. There is no real need for more than one ORG statement within a normal program.

The ORG pseudo opcode evaluates the expression in the operand field and loads the calculated variable into the code counter (see OBJ above) 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 try to assemble your code outside of 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.

## SECTION 5.4 — 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 does not require you to do so, it is good practice to define all of your zero page locations used in your program before any code is generated. Zero page is used mainly to hold variables and pointers. However, before wildly using up locations in zero page 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 pseudo opcode.

The EPZ pseudo opcode only supports the addition and subtraction operators in address expressions.

WARNING: Future versions of LISA will require that zero page variables be declared before they are used, for compatabilities sake...

## SECTION 5.5 — 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 pseudo 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
       LDA #CTLD

      -or-

HIMEM  EQU $9600
       LDA #HIMEM
       PHA
       LDA /HIMEM

LETRA  = "A"
       LDA #LETRA
```

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

## SECTION 5.6 — ASC: ASCII STRING DEFINITION

SYNTAX: ASC 'any string'

-or-

ASC "any string"

The ASC pseudo opcode 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 occurance 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 string created is:

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

Note: ASC is more or less obsolete. It is included to make LISA 2.5 compatable with eariler versions of LISA and other assemblers. When writing new programs you should use the BYT and .DA pseudo opcodes.

## SECTION 5.7 — 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$, MID$, & LEFT$), comparisons, output, etc. are accomplished much easier when the actual length of the string is known.

Except by manually counting the characters up and explicitly prefacing a length byte to your string, the ASC pseudo opcode does not allow you use use this very flexible data type.

The STR pseudo opcode functions identically to the ASC pseudo 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
```

## SECTION 5.8 — 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 pseudo opcode 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 "$" (in fact it cannot begin with a "$"!), nor does it have to begin with a decimal digit if the first hex digit is in the range A-F.

EXAMPLES:

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

## SECTION 5.9 — LST: LISTING OPTION ON

LST activates the listing option. During pass three all source lines after the 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 (described below) 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.

## SECTION 5.10 — 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" pseudo opcode is encountered, will not be listed.

LST and 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 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

## SECTION 5.11 — ADR: ADDRESS STORAGE

SYNTAX: ADR <expression> {,<expression>}

The ADR pseudo opcode 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 by LISA.

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 pseudo opcode. This technique is very useful for translating BASIC programs to assembly language.

## SECTION 5.12 — 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.

## SECTION 5.13 — 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 opcode should be used when there is not enough memory available for the current text file.

LISA provides you with enough memory for approxamately 1500 to 2000 lines of text. Should you try to exceed this limitation a "memory full" error will result. When this happens you should delete the last 10 lines or so (to give you some working space) and, as the last line of your text file, use the "ICL" pseudo opcode to link in the next file.

Once the "ICL" pseudo opcode 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 to disk under the name specified in the "ICL" pseudo opcode. Now load in the original file 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. Yes, it would be nice if LISA brought the original text file back into memory, but unfortunately this takes time. And since most people want to run their program immediately after assembly the amount of time required to reload the original text file off of the disk is not justifiable for those few instances where you want the original text file back.

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, as well as the line number within the file where the infraction occured. As before, you have 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, resave the text file to the disk 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 on the number of files that you can chain together using the ICL pseudo opcode.

## SECTION 5.14 — DCM: DISK COMMAND

SYNTAX: DCM "dos command"

During passes one and two the DCM pseudo opcode 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 pseudo opcode 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 thereby freeing memory space, it can be used to create a disk text file listing of the assembled program. DCM may also 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 you 4K bytes for your object code (from $800 to $1800) you have to BSAVE your object files to disk when this 4K is used up. Once the file is BSAVED to disk you can use the OBJ pseudo opcode 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. Remember, 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 to top of each other!

To get around this you should use the "A$" option when BLOADing a program to load the module into its correct memory location. In fact, when BSAVEing a program with the DCM pseudo opcode it is a good idea to make the loading address part of the file name (for example: OBJ.1/$1800).

EXAMPLES:

```
                .
                .
                .
<FIRST 4K BYTES OF PROGRAM>
                .
```

```
                .
                .
DCM "BSAVE OBJECT/$800,A$800,L$1000
OBJ $800
                .
                .
                .
<NEXT 4K BYTES OF PROGRAM>
                .
                .
                .
DCM "BSAVE OBJECT/A$1800,A$800,L$1000"
OBJ $800
                .
                .
                .
ETC.
```

The symbol table listing may be suppressed by using the disk command "INT". This should be entered in your program immediately before the "END" pseudo opcode. Assembly automatically terminates when the DCM "INT" pseudo opcode is encountered and you are returned to LISA's command level.

To create a disk text 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 symbol table listing as part of the text file, then omit the DCM "CLOSE" and explicitly CLOSE the file with an immediate CLOSE command when you are returned to LISA's command level.

WARNING! Due to memory management techniques used by LISA 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 the file is closed you can execute the DOS command. After the DOS command 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. Since ICL uses the DOS care must be taken when writing such files out to disk.

## SECTION 5.15 — PAU: PAUSE/FORCE ERROR

PAU is ignored during passes one and two. During pass three however, this pseudo opcode 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 a some particular section of code. PAU is also useful in determining where the 4K cutoff is when you are saving object files to disk.

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.

## SECTION 5.16 — 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.

## SECTION 5.17 — DCI: DEFINE CHARACTERS IMMEDIATE

SYNTAX: DCI "any string"

-or-

DCI 'any string'

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

EXAMPLES:

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

## SECTION 5.18 — INV: INVERTED CHARACTERS

SYNTAX: INV "any string"

-or-

INV 'any string'

INV takes the string which follows and outputs the characters as Apple inverted 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 "[", "\", and "_" have inverted counterparts. The lower case letters and several special characters do not have coresponding 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
```

## SECTION 5.19 — BLK: BLINKING CHARACTERS

SYNTAX: BLK "any string"

-or-

BLK 'any string'

BLK is the counterpart 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 reasons).

## SECTION 5.20 — 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 (described below) it is 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 OF LBL
HBY "A"    -ANY ASCII DATA ALWAYS GENERATES $00

HBY LBL1,LBL2,LBL3
```

## SECTION 5.21 — 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 benifical 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 occurance 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
```

## SECTION 5.22 — DFS: DEFINE STORAGE

SYNTAX: DFS <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 opcode. 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
```

## SECTION 5.23 — 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
```

## SECTION 5.24 — 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, counters, 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 opcode and then subsequently redefine it with a LET pseudo opcode, 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 pseudo opcode 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 the LET pseudo opcode in place of the EQU in the original definition. If the original definition was a statement label then substitute "LABEL LET *" instead.

## SECTION 5.25 — TTL: TITLE

SYNTAX: TTL "STRING"

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

## SECTION 5.26 — .IF: CONDITIONAL ASSEMBLY

SYNTAX: .IF <expresion>

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 as HASPRNTR, HAS64K, HASMODEM, LCPLUS, KBPLUS, etc. can be set 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', and '.FI'. '.IF' begins a conditional assembly sequence. '.IF' is followed by an address expression. If it evaluates to true (non-zero), then the code between the '.IF' pseudo opcode and its corresponding '.EL' or '.FI' pseudo opcode is assembled. If the address expression evaluates to false, then the code immediately after the '.IF' pseudo-op will not get assembled (see the '.EL' description).

## SECTION 5.27 — .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.

## SECTION 5.28 — .FI: END IF

SYNTAX: .FI

'.FI' terminates the conditional assembly language sequence. It must be present whether or not there is a '.EL' pseudo-op present. All code after a '.FI' pseudo opcode 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, LISA will give you a nasty error at assembly time. All IF's must be terminated before an END or ICL pseudo op is encountered or LISA will terminate assembly.

To see an example of conditional assembly, look at the 'LISA P1.L' file on the LISA master disk.

## SECTON 5.29 — PHS: PHASE

SYNTAX: PHS <expression>

The PHS pseudo opcode 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 shprt 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 it performs an ORG without the OBJ. The DPH (described below) must be used to terminate the PHS code sequence.

## SECTION 5.30 — DPH: DEPHASE

SYNTAX: DPH

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

SECTION 5.31 — .DA: DATA

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

- where <special expression> is -

#<expression>
/<expression>
<expression>
"string"
'string'

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

EXAMPLES:

```
LBL1 .DA #CR,RETURN
LBL2 .DA 'C',#LBL1,/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.

SECTION 5.32 — GEN: GENERATE CODE LISTING

SYNTAX: GEN

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

## SECTION 5.33 — 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 listing containing strings and multiple addresses.

## SECTION 5.34 — USR: USER DEFINED PSEUDO OPCODE

SYNTAX: USR <anything>

For more info on the USR pseudo opcode check the appendicies.

SECTION 6.0 — ADDITIONAL FEATURES/RANDOM NOTES

1) EXTENDED MNEMONICS

2) SWEET-16 MNEMONICS

3) EXTRANEOUS NOTES

SECTION 6.1 — EXTENDED MNEMONICS

The word "mnemonic" means memory aid. "LDA #$FF" is certainly easier read as "load the accumulator with the constant $FF" than is A9FF. Nevertheless, 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 user, 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. These new extended mnemonics make life just a little easier for the programmer.

The extended mnemonics included in LISA 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
```

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

SECTION 6.2 — SWEET-16 MNEMONICS

LISA incorporates a Sweet-16 assembler for use with the Sweet-16 pseudo machine interpreter in the Apple ROMs (see BYTE, Nov 1977 or Applesauce, Nov 1979 for details). For the most part LISA uses standard WOZ mnemonics except where Steve Wozniak used two and four character mnemonics. Since this tends to disrupt the nice assembly listing, all two and four character mnemonics where converted to three character mnemonics to improve the listing format.

SWEET-16 MNEMONIC CONVERSIONS:

| WOZ'S | LISA'S |
|---|---|
| SET | SET |
| LD | LDR |
| ST | STO |
| LDD | LDD |
| STD | STD |
| ADD | ADD |
| SUB | SUB |
| POPD | PPD |
| CPR | CPR |
| INR | INR |
| DCR | DCR |
| RTN | RTN |
| BR | BRA |
| BNC | BNC |
| BC | BIC |
| BP | BIP |
| BM | BIM |
| BZ | BIZ |
| BM1 | BM1 |
| BNM1 | BNM |
| BK | BKS |
| RS | RSB |
| BS | BSB |
| BNZ | BNZ |

SECTION 6.3 — WARNINGS AND OTHER EXTRANEOUS NOTES

The RESET key on the Apple II always 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 the RESET key absoulutely 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 and 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 use E003G or CRTL-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 (hopefully) is the last line you typed in.

To help prevent an accidental depression of the RESET key you might remove it from the Apple keyboard. Using a small screwdriver it should be fairly easy to pop off (but if you meet great resistance, as though it were glued on, use common sense and STOP).

Coldstart is E000G (control-B) - this clears the text file, reinitializes pointers, etc. Never set MAXFILES to anything other than 1 while using LISA 48K. If you do, part of LISA will be destroyed.

SECTION 7.0 — ADVANCED TOPICS



SECTION 7.1 — MEMORY ALLOCATION

LISA, upon coldstart, initializes several parameters. 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 the symbol table. These numbers were not picked out of the air! After considerable research and experience with LISA version 1.3 Lazer Systems has determined these values to be optimal for CORESIDENT operation. Since 80% of all assembly language programs are less than 2000 lines long, contain less than 512 symbols, and the object code produced is less than 4K bytes our coldstart settings should prove sufficient for most applications.

We have not forgotten the other 20% of the time, however. 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, since you have t

anyway, 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.

To make a sound judgement as to how you should adjust these values you should realize that each entry in the symbol table requires 8 bytes. Also, the average line of text in LISA for a well documented program requires about 10 bytes. Object code is generated at the approximate ratio of two bytes per line of source code.

As an example, LISA represents almost 6000 lines of source code. It was reassembled (using LISA) in 5 sections with STXT set to $2000 (although $1800 would have sufficed) and HMEM set to $5800 (allowing 768 symbols).

To change these parameters in LISA load LISA and then get into the Apple monitor (i.e. the "BRK" command). Enter E000L into the Apple monitor. 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 coldstart (7000G/E000G) you will set STXT to $2000. The low order byte of STXT is always set to $00.

Immeditately 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 coldstart.

WARNING: If you create a large text file with LISA set up for the coresident 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 file 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 not be modified without careful consideration.

## SECTION 7.2 — LISA'S MNEMONIC TABLE

LISA MNEMONIC table format can be seen by looking at the P1.L file provided on the LISA diskette.

## SECTION 7.3 — LISA'S INTERNAL FORMAT

The detokenization routine for LISA is provided in the P1.L file. You can check this routine to see how LISA arranges things internally.

## SECTION 7.4 — WHY LISA IS SO FAST

Most of LISA's speed is due to the fact that the mnemonics are tokenized and the input line is prescanned for syntax errors. LISA's 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.

## SECTION 7.5 — USING A DIFFEREND EDITOR WITH LISA

If you have access to a text editor that outputs text type files to diskette you may be able to use it will 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 LISA's 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 (since it is interactive) 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.

## SECTION 7.6 — CONVERTING FILES FROM LISA 1.5 TO LISA 2.5

To convert a text file created using LISA 1.5 for use with LISA 2.5, follow this simple procedure: 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. That's all there is to it.

## SECTION 7.7 — 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 2.5 saves source files on the disk as LISA "L" files. LISA's LO)ad command only loads "L" files and will generate a FILE TYPE MISMATCH ERROR if you attempt to load a LISA 2.x binary type file.

To alleviate this problem, the LOB command (for LOad Binary) has been included in LISA 2.5's command set. LOB can be used to load older LISA textfiles into LISA 2.5, which may then be saved to disk as an "L" file from LISA 2.5

## SECTION 7.8 — WHERE TO ADDRESS ADDITIONAL QUESTIONS

Naturally, no matter how much documentation comes with a product, there will always be questions. Before doing anything else read this manual from cover to cover, most questions are already answered in this manual, you simply missed what you were looking for the first time through. Answering questions takes time, which costs money, and will eventually be reflected in the cost of this product and/or its updates. So please, read the manual first.

As mentioned, no matter how much documentation is provided questions will still go unanswered. If this should occur please write Lazer Systems at:

Lazer Systems
Bx 55518
Riverside, CA. 92517

For speedy response please send SASE as well as your phone number so that Lazer Systems may provide you with a reply.

SECTION 8.0 — "TRICKS" ETC. WHEN USING LISA

1) EFFECTIVELY USING LISA'S BUILT-IN EDITOR

2) USING THE AP(PEND) COMMAND

3) USING THE DOS "EXEC" COMMAND

4) A DESCRIPTION OF WHAT HAPPENS WHEN YOU ASSEMBLE A FILE

5) RUNNING A LISA PROGRAM: FROM START TO FINISH

SECTION 8.1 — EFFECTIVELY USING LISA'S BUILT-IN EDITOR

LISA's built-in editor is very easy to use once the user becomes accustomed to the few differences which exist between LISA's editor and the editor used in BASIC.

The first, and most important, thing to remember is that all modifications to a LISA text file affect the line numbering scheme. LISA's line numbers are dynamic (always changing) as opposed to the static line numbering system in BASIC (static means fixed). Although the line editor in LISA 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 or 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 LISA editing commands, form the basis of a very powerful screen oriented editor. For 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 accomplished by M(ODIFY)ing line 33 (M 33), pressing 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 accomplished 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

and then hit return. After you hit return press control-E to get you out of the insert mode. Now L(IST) 33. You will see "LABEL LDA 00". As you can 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 (perhaps change "LABEL" to "LABL"?) simply use the control-K cursor control to skip over the undesired characters.

To replace characters within a line simply type the new characters over the top of the undesirable characters.

It is also possible to use LISA's 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 than 20 lines do it in several stages, copying 20 lines each time (but remember to always 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 discussed above. Once the copy is complete, delete the original lines using the LISA D(ELETE) command.

## SECTION 8.2 — USING THE AP(PEND) COMMAND

The LISA AP(PEND) command will take a source file from diskette or tape and append it to the end of the existing text file in memory. This feature is very 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 very useful when you need to copy a section of code several times in your program and the copy procedure mentioned above turns out to be too much work.

You should be very careful when AP(PEND)ing text 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.

## SECTION 8.3 — USING THE DOS "EXEC" COMMAND

The Apple DOS EXEC command can be used in LISA 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
 NLS
<CONTROL E>
ASM
DEL 1
```

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

The EXEC command is also useful for performing intrafile merging. By W(RITE)ing a text file out to disk and then modifying the first line so that it reads "INS <linenum>" 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!

## SECTION 8.4 — A SHORT DESCRIPTION OF WHAT HAPPENS WHEN YOU ASSEMBLE A LISA SOURCE FILE

The 6502 microprocessor does not understand "assembly language" despite what you may have heard. What the 6502 does understand is "6502 machine language". Although you may have been led to believe that assembly language and machine language are one and the same, 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. Fortunately, it is fairly easy to convert assembly language to machine language (at least easy compared to converting say BASIC, FORTRAN, or Pascal to 6502 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 wrap-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 machine code generated by LISA. If you look up these codes on the provided 6502 programming card you will see that these operation codes correspond to the actual

assembly language instructions (plus any required data). The next one to four digits correspond to the LISA 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 pseudo opcode). When you run the program the code must be stored at these locations in memory. If the program is not stored at this location in memory, it will not work properly when executed. Some programs are an exception to this rule and are called "relocatable" programs.

Once it is assembled, running an assembly language program is very easy. First get into the monitor by issuing the LISA "BRK" command. Once you are in the monitor you may run your program by using the monitor "GO" command. This is very similar in practice to the BASIC "RUN" command. The only difference is that you must specify a run address. 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.

You may terminate your machine language program in one of several ways. You can use the BRK mnemonic which will stop the program, print the current contents of the 6502 registers and return to the monitor. If you terminate your program with a RTS instruction you will be returned to the calling routine (in this case the monitor). You can also use a JMP instruction to enter BASIC, LISA, Pascal, or some other program directly.

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

## SECTION 8.5 — RUNNING A LISA PROGRAM: FROM START TO FINISH

First, decide exactly what program you wish to write. In our example we will simply print the two hex digits "FF" onto the Apple video screen and then return to the monitor.

Once the program has been decided upon, the next step is to decide how the program is to be written. In this case we will simply load the accumulator with $FF and then JSR to a monitor routine which prints the contents of the accumulator as two hex digits.

Now the program has to be entered into LISA's text editor. To do this type "INS" when you get the "!" prompt. After hitting return your screen should look something like this:

```
!INS
```

```
1
```

and a blinking cursor will prompt you for text entry. As with all 6502 programs not utilizing decimal arithemetic, the first instruction should be a clear decimal flag instruction or "CLD". To enter this instruction type a space (remember, column one is reserved for labels) and the the sequence "CLD" followed by return. The screen should now look like this:

```
!INS
   1  CLD
   2
```

Once again the blinking cursor tells you that LISA is waiting for text entry. Now we can begin the main portion of our program. The first thing which 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 0FF if you prefer). To enter this instruction type a space (remember column one...) followed by "LDA". Now type another space and then "#$FF" followed by return. The screen should now look like this:

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

Again, the blinking cursor will prompt you to enter more text. Now we must enter the instruction which prints the contents of the accumulator as two hex digits. This happens to be "JSR $FDDA". To enter this program type a space, followed by "JSR", followed by a space, followed by "$FDDA" followed by return. The screen should now look like this:

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

Now we must enter the command to return control to the Apple monitor. One way to do this is by using the "RTS" instruction. So let's enter it on the next line. Type a space followed by "RTS", followed by return. The screen should now look like this:

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

```
5
```

As far as the 6502 is concerned, our program is complete. However, we still haven't told LISA that the end of the program has been reached. To do this type the pseudo opcode "END" onto line 5 (don't forget the space first!). Upon hitting return the screen should look like this:

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

Note that LISA still wants more text! To get out of the text entry mode type control-E as the first character of line 6. Upon hitting return you will be returned to the command level.

Since there are no labels in our program, printing the symbol table will prove to be a big waste. Why don't we issue the pseudo 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 place you in the insert mode with all text being inserted before line number 5. The screen should now look like this:

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

!INS 5
   5
```

and the blinking cursor reminds you that text entry is required.

Now type in a space, followed by DCM, followed by "INT", followed by return The screen should now look like this:

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

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

Since this is all the text we wish to enter, type control-E as the first character of line 6.

Our program is now complete (and this can be verified by using the L(IST) command). We must now assemble the text file before it can be run. To do this simply type ASM at the command level, followed by return. Your screen should now look like the following:

```
!ASM

*** END OF PASS ONE

*** END OF PASS TWO


0800 D8        1        CLD
0801 A9FF      2        LDA #$FF
0803 20DAFD    3        JSR $FDDA
0806 60        4        RTS
0807           5        DCM "INT"
INT!
```

The first column is the hexadecimal address where the current instruction/code resides. 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 our program was assembled at location $800, to run the program we must issue either a "CALL 2048" from BASIC or 800G from the monitor ($800 = 2048 decimal). Just as an example let's run our program from the Apple monitor. To do so we must first get into the Apple monitor. This is accomplished by using the LISA command "BRK". Simply type "BRK" while in the command mode and you will be placed into the Apple monitor. When you get the "*" prompt type 800G followed by return. The screen should now contain:

```
!BRK
*800G
FF
*
```

The FF was the printed result of our program and we were returned to the Apple monitor. If you wish to return to LISA you may do so by typing "E000G" (coldstart) or "E003G" (warmstart).

Although you don't have to, most 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 very easy to determine the starting address of the program.

If you plan to go back and forth between your program and LISA, make sure that you do not utilize the zero page locations used by LISA. Unpredictable things may happen should you do so. A list of the zero page variables used by LISA is provided in the appendices.

Before running your program, ALWAYS make sure that you have saved your latest version to disk or tape. It's too easy for a bug in your program to wipe out everything in memory.

## SECTION 9.0 — SOFTWARE PROVIDED WITH LISA

1) LZR IOS ROUTINES (LISA P2.L)
2) RANDY'S HIRES ROUTINES
3) RWTS
4) SINGLE DRIVE COPY
5) SOFTWARE TOOLS
6) SORT
7) MPTOLISA

## SECTION 9.1 — THE LZR IOS ROUTINES (LISA P2.L)

The Lazer System's Input/Output System (source provided) is a collection of various input/output routines and several utility routines. These routines, in fact, are used for I/O handling inside LISA. To get an idea how these routines are used, check out the LISA P1.L file.

## SECTION 9.2 — RANDY'S HIRES ROUTINES

These routines are some handy hires graphic routines intended especially for the assembly language programmer. They allow flexible display production directly from machine language programs.

These routines were intended to set in the range $1000 - $2000. But you may change this by re-ORGing the code elsewhere. Since these routines contain a considerable amount of internal documentation there was not enough room to hold the entire text file in memory all at once (comments require a considerable amount of memory). As a result "HIRES.2" gets chained in during assembly. All you have to do is LO(AD) RANDY'S HIRES ROUTINES and then A(SM) the text file. An "ICL" pseudo opcode at the end of RANDY'S HIRES ROUTINES automatically does the rest for you.

Rather than describe each of the routines provided in the hires package, a reprint of the October 1979 Applesauce describing these routines is provided in the appendices.

## SECTION 9.3 — RWTS

Due to the space limitations on the disk, RWTS is no longer provided on the LISA diskette.

SECTION 9.4 - SINGLE DRIVE COPY.

SDC is no longer provided either.

SECTION 9.5 -- USING "SOFTWARE TOOLS"

Software tools is a collection of routines loosely modeled after the routines found in the book "SOFTWARE TOOLS" by Kernighan and Plauger. Check out the listing for details on how the routines are used.

# APPENDIX A -- MEMORY USAGE

PAGE 0 MEMORY LOCATIONS $50-$AF; $E0-$FF

PAGE 2 IS A RESERVED I/O BUFFER

PAGE 3 IS OPEN FOR USER SUBROUTINES FROM $360-$3FF

$0800-$1800 IS RESERVED FOR USER GENERATED CODE

$1800-$7FFF IS RESERVED FOR THE TEXT FILE

$8000-$94FF IS RESERVED FOR THE SYMBOL TABLE

$9500-$95FF IS A RESERVED I/O BUFFER

$D000-$F7FF IS RESERVED FOR LISA

LISA modifies the Apple DOS so if you want to return to BASIC or APPLESOFT to do some other programming you should reboot a different disk containing the desired language.

LISA requires that DOS 3.2 be used. LISA MAY BE CONVERTED TO DOS 3.3 BY USING APPLE COMPUTER'S "MUFFIN" PROGRAM.

ADDRESSES PERTAINENT TO LISA 2.5:

| ADDR | 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<br>0=NO TTL PSEUDO OP<br>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. Also remember that control-C and control-B do not work for LISA 48K systems. You must type "$6000G" or "$6003G" to cold or warm start LISA 48K.

# APPENDIX B -- SUMMARY OF COMMANDS

NOTE: Optional information is enclosed in parentheses.
Optional parameters are enclosed in brackets.

I(NSERT): Inserts text after the last line in the text file.
I(NSERT) {line#}: Inserts text before line number "line#".

D(ELETE) line#: Deletes line number "line#".
D(ELETE) line#{,line#}: Deletes lines in range specified.

L(IST): Lists the entire file.
L(IST) {line#}: Lists a single line.
L(IST) {line#,line#}: Lists lines in range specified.

LO(AD): Loads text file from cassette.
LO(AD) {filename}: Loads file from diskette.

SA(VE): Saves file to cassette tape.
SA(VE) {filename}: Saves file to diskette.

W(RITE) filename: Writes file to diskette as a TEXT type file.

AP(PEND): Loads a file from cassette and appends it to text file in memory.
AP(PEND) {filename}: Loads a file from diskette and appends it to file in memory.

Control-D: Allows user to execute a 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 ERRORS

## COMMAND LEVEL ERRORS

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 ERRORS

ILLEGAL SYMBOL IN LABEL:
An illegal symbol was detected in the label field. Possibly a control character. For more info see the section on labels.

LABEL TOO LONG:
A label appeared in the label field which contained more than 6 characters. If you accidentally enter a 7th character (or longer) label in the operand field you will get an "ILLEGAL OPERAND" message.

ILLEGAL MNEMONIC:
The symbol in the mnemonic field is not a valid LISA mnemonic. Probable causes: beginning the mnemonic in column one, beginning a 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:
Programmer 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. Remember, it takes 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 field 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 ERRORS

PASS ONE:

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 pseudo opcode is invalid.

MISSING 'END':
The end of the text file was encountered but no END pseudo 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.

PASS TWO:

DUPLICATE LABEL:

User attempted to redefine a previously defined label.

UNDEFINED SYMBOL/ILLEGAL ADDRESS:
Occurs when a symbol is encountered 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 pseudo 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 illegal 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 those OBJ's and ORG's).

PASS THREE:

UNDEFINED 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. The particular symbol, due to its useage, must be a zero page variable.

ADDRESSING MODE REQUIRES ZPAGE VARIABLE:
User attempted to use an absoulute location where a zero page location is required.

LISA 2.5 supports a special, user-definable, pseudo opcode. This pseudo-op is a no holds barred, always syntactically correct, user assignable pseudo opcode.

Whenever the USR pseudo opcode is encountered, a JSR to location $E029 is performed. Normally there is an RTS instruction and two NOP instructions at location $E029. You can, however, 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 on the disk, LISA P1.L, LISA P2.L, and LISA P6.TXT contain sources for portions of LISA; you may take a look at these files and call any routines within them.

Several additional routines useful to to programmers utilizing the USR pseudo opcode, as well as certain memory locations you can mess around with, are:

ERRR- prints an error message and gives the user the chance to abort assembly. ERRR's location is $EBE6, 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- 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 and 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, a zero page address expression. BIT #6 of AFND is set if all 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. Calling address: $EBE9

SYMLUP- 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 table. Calling address: $EBEC.

Useful variables:

PNTR: ($EA) points at the beginning of the current line

LNUM: ($F2) holds 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 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. (see CODSAV below)

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) $2C0-$2DF 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.

ENTRY CONDITIONS.

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

## XREF/65

XREF/65 is a general purpose cross reference generator for LISA 1.5F and LISA 2.x. It lists each symbol defined in a program as well as the line number of the line where it was defined and the line number of each occurence of that label within the file. A cross referencer program is useful when debugging a program since a variable's usage can be followed. A cross referencer is also invaluable when documenting a program since all variables and labels used within a program are conveniently listed for you. To use XREF/65 you must have an Applesoft ROM card or a 16K RAM card with DOS 3.3 or a BASICS disk. Also, you will need a printer with a printer interface card installed in slot one (this can be changed within the Applesoft program). It is assumed that you have a 48K (or larger) Apple.

Before generating a cross reference listing there are two things you must do. First, make sure that LISA will assemble your program without any errors. If the program is not quite complete you will have to insert dummy labels so that all labels are defined within your program. THIS IS VERY IMPORTANT. If your program assembles correctly there should be no problems. The second thing you must do before creating a cross reference is to generate a TEXT type file of your LISA program. This is accomplished by using LISA's W)rite command. For example, if you wish to create a cross reference listing of a file named "BLETCH" you would load it into memory using the LISA L)oad command. Next, create a TEXT type file using the LISA command "WRITE BLETCH.TXT". With all this preparation out of the way you can now boot the XREF/65 disk and type "RUN XREF/65" to begin program execution. XREF/65 will prompt for a filename at which point you should enter the desired filename (in this case "BLETCH.TXT"). A little while later a cross reference listing will be printed on your printer.

## SCTOLISA

SCTOLISA is a program which will help SC ASSEMBLER II version 3.2 owners convert their existing software to a LISA format. To use SCTOLISA type "BRUN SCTOLISA" while in BASIC. The program will ask you to enter an SC ASSEMBLER II filename. Upon hitting return the program will read the file from disk and proceed to convert it to a LISA format. The file is written to the disk as a TEXT type file. The filename is "SC-CONVERTED" which may be EXEC'd into LISA.

SCTOLISA is a semi-automatic conversion program. That is to say some manual conversion will be required. In particular, not all Sweet-16 mnemonics are properly converted. These, however, are easy to fix as LISA will catch such problems while the file is being EXEC'd in (in fact, if you have LISA 2.2 you will get a chance to correct the problem when it is detected).

One other major problem with the SCTOLISA program concerns address expressions. In a nutshell SCTOLISA does not handle certain types very well. There are two types in particular that SCTOLISA has problems with. First, the SC assembler allows any expression which evaluates to a value between 0 and 15 to be used as a Sweet-16 register operand. LISA only allows operands of the type R0, R1, R2, ..., RF. So any Sweet-16 registers operands not conforming to LISA's syntax must be manually converted. The second problem concerns zero page equates. If SCTOLISA encounters any symbolic values in the operand field of an ".EQ" pseudo opcode it automatically assumes that the "EQU" should be used. This usually leads to several "non-zero page value" errors when attempting to assemble the file. The fix is quite simple- just change the "EQU" to an "EPZ" for the infracting label.

## SORT 2.0

Sort 2.0 is a symbol table sort routine provided for LISA 2.2 users. It should be "BRUN" immediately after an assembly in order to print a sorted (numeric and alphabetic) symbol table listing.

## Support (Or the lack thereof)

Due to the incredible low cost of this package, Lazer Systems cannot provide support for the routines supplied herein. Since support is not provided, the source listings (LISA 2.2 compatable) are provided in the event a user may wish to add improvements to a program provided here. The source listings are provided for personal use only, commercial rights are reserved.

# THE LAZER SYSTEMS' DISASSEMBLER

## FORWARD

This program was written against my free will. I swore up and down that a good disassembler program could not be written. But pressure from numerous LISA owners finally convinced me that this program had to be written. A disassembler program can never be "perfect". While an assembler will always generate the same code for a given source file, a disassembler can produce radically different code during each disassembly of a given section of machine code. Although I am still convinced that a good disassember cannot be written, I hereby give you DISASM/65. This program is dedicated to all those people who said: "So what if it isn't perfect, it's better than nothing!"

## AN INTRODUCTION TO DISASSEMBLERS

An assembler, such as LISA, takes an assembly language source textfile and converts it to the proper machine language. A disassembler should do just the opposite. That is, it should take machine language and produce assembly language source code.

So why should you buy this product when the Apple monitor provides you with a disassembler which nicely performs this function? For the same reason you bought LISA instead of using the Apple's built-in mini-assembler: The Apple mini-disassembler is non-symbolic whereas DISASM/65 produces a symbolic disassembly listing. The disassembly produced by DISASM/65 can be reassembled using the LISA interactive assembler. This allows you to relocate programs, see how other programs are written, patch up programs, etc. DISASM/65 produces LISA-compatable textfiles complete with labels instead of non-symbolic addresses. This feature alone makes this disassembler invaluable since programs with labels are much easier to follow than disassembled programs without symbolic labels. DISASM/65 also allows the user to specify hexadecimal, ASCII, address, and pushed address data types. With the addition of the DISASM/65 "ignor" mode DISASM/65 becomes easily the most powerful disassembler for the Apple II.

## PROBLEMS WITH DISASSEMBLERS

Most disassemblers disassemble code between two boundry locations input by the user. This concept would work fine, except that machine code does not consist of pure instruction code! Consider the following assembly language program:

```
        LDA HERE
        STA THERE
        BRK
HERE    HEX AD
        HEX 0A08
THERE   HEX 00
        END
```

If you were to assemble this program you would get the machine code:

AD,07,08,8D,0A,08,00,AD,0A,08,00

at location $800. If you were to disassemble this (using the Apple mini-disassembler or something similar) you would get the instruction sequence:

```
800-    LDA $807
803-    STA $80A
806-    BRK
807-    LDA $80A
80A-    ???
```

The variable 'HERE' at location $806 was interpreted as an instruction. And that is the major problem with disassemblers. They cannot distinguish between code and data. Because of this problem, a disassembler program cannot be an automatic one. The user will have to tell the disassembler which portions of memory to treat as instruction code and which portions to treat as data.

If hexadecimal data was the only type of data used in machine language programs there wouldn't be that much of a problem. Unfortunately data comes in many types. Hex data, ASCII data, addresses, and return addresses are data types which are often specified in a machine language program. While DISASM/65 allows for all these data types (and then some) the user must tell DISASM/65 which addresses are to be code, which are to be data, and what type of data.

## USING DISASM/65

To run DISASM/65 simply boot the disk provided and type "BRUN DISASM/65". Within a few seconds DISASM/65 will greet you with the message:

"ENTER RANGE OF DISASSEMBLY:"

At this point you must give the beginning and ending addresses of the section of code you wish disassembled. These addresses should be given in the standard monitor "start.end" format. The address range should be terminated with a return. The ending address provided must be greater than the starting address or you will be prompted to reenter the disassembly range.

This disassembly range is used to determine whether statement labels or equates are to be used when an address is encountered in an operand field. For example, if you specify an address range of "8000.BFFF" then all address references within the range $8000 to $BFFF will be disassembled as statement labels (i.e, labels within the source program). References to memory outside this range will be listed in the disassembled source in an EQU or EPZ format. So if the range 8000.9000 was specified then references to location $88C3 will reference a label within the program. If the machine code being disassembled references location $FDED then an EQU instruction at the beginning of the program will be generate to link the disassembled program to outside routines (in this case the COUT routine in the Apple monitor).

It should be pointed out that the range specified by the user does not necessarily mean that the entire specified range is to be disassembled. The range input by the user is used only as a bounds for the disassembly (i.e, you can't specify code outside this range be disassembled) and as a means of determining which labels will be statement labels and which labels will need equates.

WARNING! If you specify an abnormally large range and labels fall within this range, but outside of the section of code you specify for disassembly, neither a statement label nor an equate will be generated for that label. For example, if you specify a range of $8000 to $FFFF and only disassemble code in the range $8000 to $9000, none of the references to locations within the Apple monitor will be listed within your program.

Once the range has been entered DISASM/65 will prompt the user to enter a starting address. This address will be used as the starting address of the disassembly. It must fall within the range specified in the previous entry. If the starting address is not the same as the initial address specified in the range then all references to data and/or instructions between the two addresses will be ignored. Usually the starting addresses will be the same.

Once the starting address is entered DISASM/65 will prompt the user to enter the disassembly command. The disassembly command consists of a string of disassembly directives which specify how the code is to be disassembled. What these directives are, and how they are specified will be considered momentarily.

## LABELS

Whenever DISASM/65 encounters a reference to an address, an entry is made in the DISASM/65 symbol table. During pass two of the disassembly, DISASM/65 ejects a label whenever the program counter is equal to a label contained in the symbol table. DISASM/65 always emits labels of the form:

Lnnnn

where "nnnn" is the absolute address (during disassembly) of the specified label. For example, the label corresponding to location $FDED in the Apple monitor will be emited as "LFDED". The single

exception to this rule is a zero page equate. Whenever the zero page addressing mode is encountered a label of the form:

Lnn

is emitted. If an instruction references a zero page location using the absolute addressing mode then a label of the form:

L00nn

is generated but no equate for this label will be emitted. During reassembly (with LISA) an undefined symbol error message will be generated. This is not a "bug" in the program. This "feature" was left in so that you can easily determine if zero page locations are being referenced using the absolute addressing mode. This usually occurs when timing loops are being used, or when the JMP indirect or LDA ABS,Y addressing mode is being used. In any case, the error message generated by LISA will draw special attention to the instruction so that you can take special action if required. To remidy the error, simply insert the statement

L00nn EQU $nn

somewhere within your program. NOTE: to insure compatability between your disassembled program and the original source program, do NOT use the EPZ pseudo opcode.

## FILE FORMAT

During pass two of the disassembly DISASM/65 writes the disassembled program to the disk. The disassembled listing is written as a standard Apple TEXT type file. The file is stored on the disk under the name "DISASSEMBLY". The first line within this file is the LISA command "INS". This allows you to EXEC the file into LISA once the disassembly is complete.

Since a text type file is created, DISASSEMBLY can also be loaded into a text editor which handles text type files such as APPLE PIE 2.0 from Programma International. With this type of editor the user can go in and change the "Lnnnn" type labels to more meaningful, mnemonic labels.

## DATA TYPES AND OTHER PROBLEMS

The major problem with any disassembler program is the fact that machine code consists of several data types besides just instruction code. DISASM/65 allows the user to specify instructions, hexadecimal data, ASCII data, address data, and return address data. While addtional, and more exotic, data types can be found these data types should prove to be sufficient for almost all applications.

Instructions:

This data type, of course, is the actual disassembled 6502 instruction code. This data type can be one, two, or three bytes

long depending upon the length of the 6502 instruction being disassembled. A full instruction is always disassembled, even if it means that the range specified for the instruction disassembly will be exceeded. If an invalid instruction is encountered, then a hex byte is emitted in place of the opcode. Additional hex bytes are emitted until a valid 6502 instruction is encountered at which point the disassembly process continues.

HEX DATA:

Tabular data is usually represented as strings of hex digits. DISASM/65 allows the user to specify that a section of memory be disassembled as hex data. The LISA "HEX" pseudo opcode is output followed by the hexadecimal data. If more than 20 hexadecimal values are output a carriage return followed by another "HEX" pseudo opcode is output and then the hexadecimal output continues.

ASCII DATA:

Messages and other text are often stored in ASCII format. DISASM/65 uses the LISA "ASC" pseudo opcode to output ASCII data. If the high order bit of the data is set then the quotes (") are used to delimit the data. If the high order bit of the data is clear then the apostrophe (') is used to delimit the data. If the sense of the high order bit changes then the current string is terminated and a new string is begun on the next line. If control characters are encountered then the data is output using the "HEX" pseudo opcode.

ADDRESS DATA:

DISASM/65 contains a feature which allows the user to specify that certain portions of the program contain address information. Address tables are often used by programmers to set up jump tables, arrays indices, etc. The address data is output with LISA's "ADR" pseudo opcode. The operand field contains a label of the form:

Lnnnn

and the address "nnnn" is entered into the DISASM/65 symbol table.

RETURN ADDRESS DATA:

A programming trick used by several programmers is to push an address onto the 6502 stack and then execute a RTS instruction to simulate an indirect jump. Since the 6502 return address is not actually the return address, but rather the return address minus one, tables containing return address for such an instruction sequence do not contain true address data. Instead, these tables are usually of the form:

```
LBL     ADR     ADDRS0-1
        ADR     ADDRS1-1
        ADR     ADDRS2-1
        ETC.
```

If this data was treated as normal address data the corresponding

equate (or statement label) would be displaced by one byte indicating a false position for the label. Since DISASM/65 can handle return address data this problem is taken care of.

## SPECIFYING DISASSEMBLY COMMANDS

Once the disassembly range and starting address are entered into DISASM/65 the program prompts the user for a command. Disassembler commands are of the form:

<address><command>

All commands for the disassembly are entered on the same line and are separated by commas. An example of a simple disassembly might be:

8040L,8060H,8080L,8120S,9000L

This particular sequence tells DISASM/65 to disassemble (list) the instructions from the starting location through 8040. After that, hex data is to be output up to location 8060. From location 8061 to 8080 memory is to be treated as instruction code. Then until location 8120 string (ASCII) data is assumed. Finally, from location 8121 to 9000 instruction code is expected.

After hitting return, DISASM/65 first checks all the values input to insure that they are within the disassembly range. If not the line will be printed up to the offending address and you will be prompted to reenter the disassembly instruction. All successive addresses in a disassembly instruction must be greater than the previous addresses specified and less than the final address specified in the range.

If all the addresses present in the disassembly command are valid then the disassembly process begins. DISASM/65 lists out portions of the program during pass one. This is your indication that the computer is actually doing something. During pass two, the disassembled listing is written to disk. This will appear on the screen if the DOS NOMON O,I,C option is set. After the disassembly is complete DISASM/65 stops and leaves you in the monitor.

## THE DISASSEMBLY INSTRUCTIONS

All disassembly instructions consist of a one or two letter sequence prefaced with an ending address. The instruction begins its operation at either the starting address or wherever the previous instruction left off. The starting address is used by the first instruction executed in the command stream.

The valid instructions are:

L (list):

The list instruction informs DISASM/65 that the incoming data is to be treated as instruction code. The list command uses the syntax:

<address>L

The address specified gives the last address where valid instruction code is found. The list command terminates whenever the disassembly program counter is greater than the address specified. Note that the disassembler will finish disassembling an instruction even if it means that additional data beyond the specifed address is required. On some occassions as much as two bytes beyond the specified terminating address will be used to complete the last instruction.

H (hexadecimal data):

The HEX command informs DISASM/65 that all code up to the specified terminating address is to be translated to a hex string. The HEX command has the format:

<ending address>H

As with the list instruction DISASM/65 will treat all data from the current location through to the ending address as hex data. All hex strings are entered into the disassembled file using LISA's HEX pseudo opcode.

W (word, or address)

The word instruction causes DISASM/65 to treat furthur data as address data. The word instruction uses the format:

<ending address>W

This causes DISASM/65 to get two consecutive bytes at a time and display them using LISA's ADR pseudo opcode. The first byte retrieved is considered to be the low order byte and the second byte is treated as the high order byte. This address is printed and placed in DISASM/65's symbol table. During pass two, DISASM/65 prints a label at the address corresponding to the address determined by the 'W' command. For this reason, ADR should not be used to define non-relocatable data such as two byte decimal constants etc (The H command should be used instead).

P (pushed data)

The 'P' command used used in a manner identical to the 'W' command. As with the word command the 'P' command grabs two bytes, constructs an address from them, and then outputs this address using LISA's ADR pseudo opcode. The difference between the 'W' and 'P' commands is that the 'W' command simply outputs the address found in the next two bytes. The 'P' command increments the address by one and then outputs the address using the format:

ADR <address>-1

The incremented address (not <address>-1) is stored in the symbol table so that a label is output whenever the program counter equals the incremented address during pass two.

The Pushed data command allows DISASM/65 users to easily create return address tables. These tables are used in code sequences such as:

```
SUBR:   ASL
        TAX
        LDA     RTNTBL,X
        PHA
        LDA     RTNTBL+1,X
        PHA
        RTS
;
RTNTBL  ADR     JUMP1-1
        ADR     JUMP2-1
        ADR     JUMP3-1
```

S (string data)

The 'S' command instructs DISASM/65 to treat following data as ASCII string data. The string command uses the format:

<ending address>S

The 'S' command generates string data using LISA's ASC pseudo opcode, it does NOT use LISA's STR pseudo opcode. DISASM/65 automatically outputs the quotes or apostrohpe depending upon the high order bit setting of the ASCII data. This allows the 'S' command to be used to disassemble code assembled with LISA's DCI pseudo opcode.

O (program Origin)

The 'O' command allows the DISASM/65 user to move the disassembly program counter to an absolute location. The 'O' command uses the format:

<orgin address>O

The origin address has the same restrictions as the ending addresses specified in the previous commands (i.e, it must be greater than all previous addresses and it must fall within the specified disassembly range). The O command will cause DISASM/65 to issue a LISA ORG pseudo opcode. As such, it should be used with care as this can lead to problems should you desire to relocate the disassembled program.

I (ignor)

The ignor command (not to be confused with the ignor mode to be described later) instructs DISASM/65 to ignor all data upto the specified ending address. The ignor command has the syntax:

<ending address>I

DISASM/65 issues a LISA DFS pseudo opcode with a length to cause all data up to the ending address to be skipped. Any symbols in the DISASM/65 symbol table which should be output within the address range covered by the ignor command will not be output.

The ignor command can be used to break up a disassembly into two or more sections. For more information see the section on the ignor mode.

V (variables)

The 'V' command performs the same function as the 'I' command except that DISASM/65 will print any labels occuring within the range of the 'V' command. Each label will be output followed by a 'DFS' statement giving the number of bytes reserved for that particular label. This value is assumed to be the number of bytes until the next label is encountered or the ending address is encountered. The 'V' command uses the syntax:

<ending address>V

## THE IGNOR MODE

One of the more powerful features of the DISASM/65 disassembler is the 'ignor' mode. The ignor mode allows you to treat a section of memory as though it were instruction code, hex data, string data, address data, etc BUT NOT INCLUDE SUCH CODE IN YOUR DISASSEMBLY LISTING.

The ignor mode allows you to disassemble code in portions. It also gives you the ability to capture portions of a program (such as DOS or the Apple monitor) for inclusion in your program leaving out unwanted sections in the middle of the code. The syntax for the ignor mode is:

<ending address><DISASM/65 CMD>I

Where <DISASM/65 cmd> is any of 'L', 'H', 'S', 'O', 'V', 'W', 'P', or 'I' (obviously using the ignor mode in conjunction with the 'I' command is meaningless). An example of the ignor mode is: include:

CMD:8080L,80C5LI,8100L,9000H,9100WI

This command will cause DISASM/65 to disassemble and write to disk the instructions in the range 8000 to 8080 (assuming a starting location of $8000). From location $8081 (or wherever the instruction at location $8080 ends) through location $80C5 DISASM/65 will disassemble the code but not write it to the disk file. Even though this section of code is not being written to disk symbol table entries are still made for all variable references found within this section of code. From location $80C5 to $8100 DISASM/65 will disassembly and write to the disk file instruction code. The data from location $8100 to $9000 is treated as HEX data and is written to disk. From location $9000 to $9100 the instruction code is treated as address information, but since the ignor mode is in effect this address information will not be written to disk. The addresses found in the range $9000 to $9100 will be stored in the symbol table so that equates or statement labels will be generated for these addresses during pass two.

## USING DISASM/65: THE PROBLEMS

Since a disassembler program cannot be an automatic one the use of such a program requires a certain amount of operator expertise. DISASM/65 was intended to be used by two types of people: The beginner who wishes to analyze programs written by others, and the advanced programmer who wishes to use portions of existing programs. The beginning programmer should simply experiment with DISASM/65 and observe the various disassembly sequences possible. The remainder of this discussion is aimed towards the advance 6502 assembly language programmer as a preparation of the problems which will be faced while disassembling programs.

Whenever a program is to be disassembled the DISASM/65 user must first analyze the program and determine where the code and data sections lie within the program. The data segments must be furthur divided into hex, string, address, and return address sections. Once this is accomplished DISASM/65 may be used to create a LISA compatable textfile of the program. Determining which sections are code and which sections are data is a long and tedious process. The technique of determining what is what is gained only through experience. Nevertheless a few guidelines and hints will be presented here.

Determining whether you are in a code segment or not is fairly simple. Disassemble a section of code (using the Apple minidisassembler) and look at it. If it seems like reasonable code it probably is. If it contains lots of ridiculous instruction sequences and several "???" opcodes chances are you are looking at data. Very seldom does an honest to God data section look like good code. Data sections typically follow an "RTS", "JMP", or branch instruction. Occassionally a data section may follow a "JSR" instruction. These simple rules are useful for determining where a data section begins, especially

when parts of the data section look like valid code.

Once you have determined which sections are code and which sections are data the hardest task still remains: determining what type of data is present in the data section. This problem is especially compounded when a data section contains more than one type of data. Hex and string data can be completely interchanged without any problems. Address and return address data cannot be treated as hex data or vice versa however. A two byte address stored within a program can change it's value if the program is reassembled in a different location other than where it was disassembled at. This holds true for return address data as well. Hex (or string) data remains constant no matter where the program is assembled at. consider the program sequence:

```
TEST    LDA     RTNADR+1
        PHA
        LDA     RTNADR
        PHA
        RTS
;
RTNADR  ADR     TEST-1
;
;
```

This program sequence pushes the return address pointing at "TEST" onto the 6502 stack. The following RTS instruction will cause the program to continue execution at location "TEST" forming an infinite loop. If assembled at location $800 this program would generate the code:

$800: AD 0A 08 48 AD 09 08 48 60 FF 07

If this program were assembled at location $900 then the code generated would be:

$900: AD 0A 09 48 AD 09 09 48 60 FF 08

There are three differences between the first assembly and the second. At locations $902, $906, and $90A the values in the second assembly are one greater since the code was assembled $100 bytes apart. The first two locations present no problem to DISASM/65 since they are portions of a 6502 instruction. The remaining discrepancy, since it is not an instruction code, does cause some problems for DISASM/66. If the last two bytes are treated as hex data instead of return address data problems arise. For example, if the code assembled at location $900 were disassembled (treating locations $909 and $90A as hex data) you would get the following text file:

```
L0900:
        LDA     L090A
        PHA
        LDA     L0909
        PHA
        RTS
L0909:
```

```
        HEX     FF
L090A:
        HEX     08
```

If this program were reassembled at location $800 the code generated would be:

```
$800: AD 0A 08 48 AD 09 08 48 60 FF 08
```

Note that the value contained in location $80A differs from the original value contained in location $80A. This is due to the fact that HEX data types remain constant whereas address and return address data types can change depending upon the location of the program in memory. This short example is but one of the problems encountered when using DISASM/65.

Luckily, DISASM/65 contains the "W" and "P" directives for disassembling address data and return (pushed) address data. There are instances, however, where even DISASM/65 fails to properly disassemble a code sequence. Consider the following short program:

```
TEST    LDA     /TEST1-1
        PHA
        LDA     #TEST1-1
        PHA
        RTS
;
TEST1   LDX     #0
        LDA     TBLLOW,X
        STA     ADDR
        LDA     TBLHI,X
        STA     ADDR+1
        JMP     (ADDR)
;
TBLLOW  BYT     TEST
        BYT     TEST1
;
ADDR    DFS     2
;
TBLHI   HBY     TEST
        HBY     TEST1
        END
;
;
```

This program demonstrates two different occurances of the same problem: having a two-byte address stored in two non-contiguous locations. In the first case a two byte address is referenced in two separate LDA immediate instructions. Since DISASM/65 always treats immediate data as hex data reassembly of this program at a location other than were it was originally assembled will cause bad code to be generated. In the second case in the previous example, the addresses in the jump table are split and stored in two separate tables. DISASM/65 does not allow the capability of accessing split tables such as this. In either of these two cases, the final disassembly must be accomplished by hand.

## USING DISASM/65: AN EXAMPLE

The use of DISASM/65 will be explained via an example. Since the Apple monitor is a complex program, and the source listing is available for it, it shall be used in the disassembly example (note: the non-auto start version of the monitor ROM will be used in the following example).

The Apple Monitor begins at location $F800 and ends at location $FFFF. So when DISASM/65 asks for a disassembly range specify F800.FFFF followed by return. DISASM/65 will ask for a starting address, enter F800 (the beginning address of the Apple monitor). At this point DISASM/65 will ask for a disassembly command. Before this command can be entered the monitor must be dissected to determine where instructions and where the data lie. Luckily we have a source listing (in the Apple reference manual). By scanning the listing you can see that there is code from $F800 to $F961. From $F962 to $F9B3 hex data is present in the Apple monitor. From $F9B4 to $F9BF string data is present. From $F9C0 to $FA42 hex data appears again. Starting at location $FA43 there is instruction code again. This instruction code goes through to location $FB19 where there is some more hex data. The hex data goes from $FB19 to $FB1D. This is followed by instruction code which goes from $FB1E to $FFCB. From location $FFCC to $FFFF we have a mixture of address and hex data. Unfortunately the address data is present only as low order byte data. The high order byte is implied by the use of the monitor program. This is one of those cases which must be handled manually after the disassembly takes place.

With this knowledge in hand we can give DISASM/65 the requested command, it is:

F961L,F9B3H,F9C2S,FA42H,FB16L,FB1DH,FFCBL,FFFEH

Upon hitting return DISASM/65 will make two passes of the program and write the disassembled version to disk under the name "DISASSEMBLY". Once DISASM/65 is done you can run LISA and load the disassembled file into LISA by typing control-D EXEC DISASSEMBLY followed by return. The file can now be assembled using LISA and compared to the original object file.

# APPENDIX D — ASCII CHARACTER SET

The letters in parenthesis under the heading KEY below have the following meanings: c = Control, s = Shift. These must be held down while striking the other key. In addition, most terminals provide separate keys for commonly used control characters, e.g. Carriage Return, Line Feed, etc.

| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 000 | 000 | NUL | P (c s) | NULL |
| 001 | 001 | SOH | A (c) | Start of Heading |
| 002 | 002 | STX | B (c) | Start of Text |
| 003 | 003 | ETX | C (c) | End of Text |
| 004 | 004 | EOT | D (c) | End of Transmission |
| 005 | 005 | ENQ | E (c) | Enquiry |
| 006 | 006 | ACK | F (c) | Acknowledge |
| 007 | 007 | BEL | G (c) | BELL |
| 008 | 008 | BS | H (c) | Backspace |
| 009 | 009 | HT | I (c) | Horizontal Tab |
| 010 | 00A | LF | J (c) | Line Feed (New Line) |
| 011 | 00B | VT | K (c) | Vertical Tab |
| 012 | 00C | FF | L (c) | Form Feed (Top of Form) |
| 013 | 00D | CR | M (c) | Carriage Return (Return) |

| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 014 | 00E | SO | N (c) | Shift Out |
| 015 | 00F | SI | O (c) | Shift In |
| 016 | 010 | DLE | P (c) | Data Link Escape |
| 017 | 011 | DC1 | Q (c) | Device Control 1 |
| 018 | 012 | DC2 | R (c) | Device Control 2 |
| 019 | 013 | DC3 | S (c) | Device Control 3 |
| 020 | 014 | DC4 | T (c) | Device Control 4 |
| 021 | 015 | NAK | U (c) | Negative Acknowledge |
| 022 | 016 | SYN | V (c) | Synchronous Idle |
| 023 | 017 | ETB | W (c) | End of Transmission Block |
| 024 | 018 | CAN | X (c) | Cancel |
| 025 | 019 | EM | Y (c) | End of Medium |
| 026 | 01A | SUB | Z (c) | Substitute |
| 027 | 01B | ESC | K (c s) | Escape |
| 028 | 01C | FS | L (c s) | File Separator |
| 029 | 01D | GS | M (c s) | Group Separator |
| 030 | 01E | RS | N (c s) | Record Separator |
| 031 | 01F | US | O (c s) | Unit Separator |
| 032 | 020 | SP | Space | Space (Blank) |

| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 033 | 021 | ! | ! | Exclamation Point |
| 034 | 022 | " | " | Quotation Mark (Double) |
| 035 | 023 | # | # | Number Sign |
| 036 | 024 | $ | $ | Dollar Sign |
| 037 | 025 | % | % | Percent Sign |
| 038 | 026 | & | & | Ampersand |
| 039 | 027 | ' | ' | Apostrophe (Single) |
| 040 | 028 | ( | ( | Opening Parenthesis |
| 041 | C29 | ) | ) | Closing Parenthesis |
| 042 | 02A | * | * | Asterisk |
| 043 | 02B | + | + | Plus |
| 044 | 02C | , | , | Comma |
| 045 | 02D | - | - | Hyphen (Minus) |
| 046 | 02E | . | . | Period (Decimal Point) |
| 047 | 02F | / | / | Slant |
| 048 | 030 | 0 | 0 | 0 (Zero) |
| 049 | 031 | 1 | 1 | 1 (One) |
| 050 | 032 | 2 | 2 | 2 |
| 051 | 033 | 3 | 3 | 3 |

| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 052 | 034 | 4 | 4 | 4 |
| 053 | 035 | 5 | 5 | 5 |
| 054 | 036 | 6 | 6 | 6 |
| 055 | 037 | 7 | 7 | 7 |
| 056 | 038 | 8 | 8 | 8 |
| 057 | 039 | 9 | 9 | 9 |
| 058 | 03A | : | : | Colon |
| 059 | 03B | ; | ; | Semicolon |
| 060 | 03C | < | < | Less Than |
| 061 | 03D | = | = | Equals |
| 062 | 03E | > | > | Greater Than |
| 063 | 03F | ? | ? | Question Mark |
| 064 | 040 | @ | @ | Commercial At |
| 065 | 041 | A | A | A |
| 066 | 042 | B | B | B |
| 067 | 043 | C | C | C |
| 068 | 044 | D | D | D |
| 069 | 045 | E | E | E |
| 070 | 046 | F | F | F |

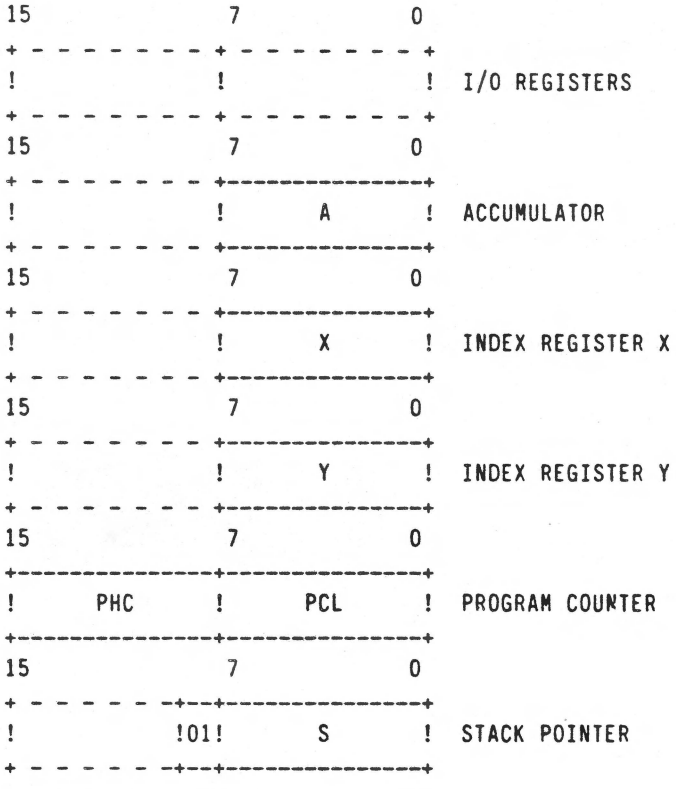
| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 071 | 047 | G | G | G |
| 072 | 048 | H | H | H |
| 073 | 049 | I | I | I |
| 074 | 04A | J | J | J |
| 075 | 04B | K | K | K |
| 076 | 04C | L | L | L |
| 077 | 04D | M | M | M |
| 078 | 04E | N | N | N |
| 079 | 04F | O | O | O |
| 080 | 050 | P | P | P |
| 081 | 051 | Q | Q | Q |
| 082 | 052 | R | R | R |
| 083 | 053 | S | S | S |
| 084 | 054 | T | T | T |
| 085 | 055 | U | U | U |
| 086 | 056 | V | V | V |
| 087 | 057 | W | W | W |
| 088 | 058 | X | X | X |
| 089 | 059 | Y | Y | Y |

| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 090 | 05A | Z | Z | Z |
| 091 | 05B | [ | [ | Opening Bracket |
| 092 | 05C | \ | \ | Reverse Slant |
| 093 | 05D | ] | ] | Closing Bracket |
| 094 | 05E | ↑ | ↑ | Circumflex (Up Arrow) |
| 095 | 05F | _ | _ | Underline (Backarrow) |
| 096 | 060 | ' | ' | Grave Accent |
| 097 | 061 | a | a | a |
| 098 | 062 | b | b | b |
| 099 | 063 | c | c | c |
| 100 | 064 | d | d | d |
| 101 | 065 | e | e | e |
| 102 | 066 | f | f | f |
| 103 | 067 | g | g | g |
| 104 | 068 | h | h | h |
| 105 | 069 | i | i | i |
| 106 | 06A | j | j | j |
| 107 | 06B | k | k | k |
| 108 | 06C | l | l | l |

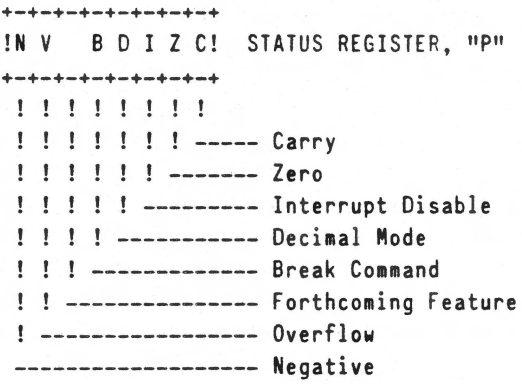
| DEC | HEX | SYMBOL | KEY | MEANING |
|---|---|---|---|---|
| 109 | 06D | m | m | m |
| 110 | 06E | n | n | n |
| 111 | 06F | o | o | o |
| 113 | 070 | p | p | p |
| 113 | 071 | q | q | q |
| 114 | 072 | r | r | r |
| 115 | 073 | s | s | s |
| 116 | 074 | t | t | t |
| 117 | 075 | u | u | u |
| 118 | 076 | v | v | v |
| 119 | 077 | w | w | w |
| 120 | 078 | x | x | x |
| 121 | 079 | y | y | y |
| 122 | 07A | z | z | z |
| 123 | 07B | [ | [ | Opening Brace |
| 124 | 07C | \| | \| | Vertical Line |
| 125 | 07D | ] | ] | Closing Brace |
| 126 | 07E | ~ | ~ | Overline (Tilde) |
| 127 | 07F | | DELETE | Delete (Rubout) |

| | Inverse | | | | Flashing | | | | Normal (Control) | | | | | | Normal (Lowercase) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal | 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 128 | 144 | 160 | 176 | 192 | 208 | 224 | 240 |
| Hex | $00 | $10 | $20 | $30 | $40 | $50 | $60 | $70 | $80 | $90 | $A0 | $B0 | $C0 | $D0 | $E0 | $F0 |
| 0 $0 | @ | P | | 0 | @ | P | | 0 | @ | P | | 0 | @ | P | | 0 |
| 1 $1 | A | Q | ! | 1 | A | Q | ! | 1 | A | Q | ! | 1 | A | Q | ! | 1 |
| 2 $2 | B | R | " | 2 | B | R | " | 2 | B | R | " | 2 | B | R | " | 2 |
| 3 $3 | C | S | # | 3 | C | S | # | 3 | C | S | # | 3 | C | S | # | 3 |
| 4 $4 | D | T | $ | 4 | D | T | $ | 4 | D | T | $ | 4 | D | T | $ | 4 |
| 5 $5 | E | U | % | 5 | E | U | % | 5 | E | U | % | 5 | E | U | % | 5 |
| 6 $6 | F | V | & | 6 | F | V | & | 6 | F | V | & | 6 | F | V | & | 6 |
| 7 $7 | G | W | ' | 7 | G | W | ' | 7 | G | W | ' | 7 | G | W | ' | 7 |
| 8 $8 | H | X | ( | 8 | H | X | ( | 8 | H | X | ( | 8 | H | X | ( | 8 |
| 9 $9 | I | Y | ) | 9 | I | Y | ) | 9 | I | Y | ) | 9 | I | Y | ) | 9 |
| 10 $A | J | Z | * | : | J | Z | * | : | J | Z | * | : | J | Z | * | : |
| 11 $B | K | [ | + | ; | K | [ | + | ; | K | [ | + | ; | K | [ | + | ; |
| 12 $C | L | \ | , | < | L | \ | , | < | L | \ | , | < | L | \ | , | < |
| 13 $D | M | ] | - | = | M | ] | - | = | M | ] | - | = | M | ] | - | = |
| 14 $E | N | ^ | . | > | N | ^ | . | > | N | ^ | . | > | N | ^ | . | > |
| 15 $F | O | _ | / | ? | O | _ | / | ? | O | _ | / | ? | O | _ | / | ? |

# APPENDIX F — PROGRAMMING MODEL MCS650X




Solid line indicates currently available features.
Dashed line indicates forthcoming members of family.


+-+-+-+-+-+-+-+-+
!N V B D I Z C! STATUS REGISTER, "P"
+-+-+-+-+-+-+-+-+
! ! ! ! ! ! ! !
! ! ! ! ! ! ! ----- Carry
! ! ! ! ! ! ------- Zero
! ! ! ! ! --------- Interrupt Disable
! ! ! ! ----------- Decimal Mode
! ! ! ------------- Break Command
! ! --------------- Forthcoming Feature
! ----------------- Overflow
------------------- Negative

## APPENDIX G — ALPHABETICAL LIST OF MNEMONICS

| MNEMONIC | DESCRIPTION |
|---|---|
| ADC | Add Memory to Accumulator with Carry |
| AND | AND Memory with Accumulator |
| ASL | Shift Left One Bit (Memory or Accumulator) |
| BCC | Branch on Carry Clear (less than) |
| BCS | Branch on Carry Set (greater than or zero) |
| BEQ | Branch on Result Zero |
| BIT | Test Bits in Memory with Accumulator |
| BMI | Branch on Result Minus |
| BNE | Branch on Result Not Zero |
| BPL | Branch on Result Plus |
| BRK | Force Break |
| BVC | Branch on Overflow Clear |
| BVS | Branch on Overflow Set |
| CLC | Clear Carry Flag |
| CLD | Clear Decimal Mode |
| CLI | Clear Interrupt Disable Bit |
| CLV | Clear Overflow Flag |
| CMP | Compare Memory and Accumulator |

| MNEMONIC | DESCRIPTION |
|---|---|
| CPX | Compare Memory and Index X |
| CPY | Compare Memory and Index Y |
| DEC | Decrement Memory by One |
| DEX | Decrement Index X by One |
| DEY | Decrement Index Y by One |
| EOR | Exclusive-OR Memory with Accumulator |
| INC | Increment Memory by One |
| INX | Increment Index X by One |
| INY | Increment Index Y by One |
| JMP | Jump to New Location |
| JSR | Jump to New Location Saving Return Address |
| LDA | Load Accumulator with Memory |
| LDX | Load Index X with Memory |
| LDY | Load Index Y with Memory |
| LSR | Shift Right One Bit (Memory or Accumulator) |
| NOP | No Operation |
| ORA | OR Memory with Accumulator |
| PHA | Push Accumulator on Stack |
| PHP | Push Processor Status on Stack |

| MNEMONIC | DESCRIPTION |
|---|---|
| PLA | Pull Accumulator from Stack |
| PLP | Pull Processor Status from Stack |
| ROL | Rotate One Bit Left (Memory or Accumulator) |
| ROR | Rotate One Bit Right (Memory or Accumulator) |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| SBC | Subtract Memory from Accumulator with Borrow |
| SEC | Set Carry Flag |
| SED | Set Decimal Mode |
| SEI | Set Interrupt Disable Status |
| STA | Store Accumulator in Memory |
| STX | Store Index X in Memory |
| STY | Store Index Y in Memory |
| TAX | Transfer Accumulator to Index X |
| TAY | Transfer Accumulator to Index Y |
| TSX | Transfer Stack Pointer to Index X |
| TXA | Transfer Index X to Accumulator |
| TXS | Transfer Index X to Stack Pointer |
| TYA | Transfer Index Y to Accumulator |

## APPENIDX H — LIST OF 6502 OPCODES

| | | | |
|---|---|---|---|
| 00 | BRK | 12 | RESERVED |
| 01 | ORA - (Indirect,X) | 13 | RESERVED |
| 02 | RESERVED | 14 | RESERVED |
| 03 | RESERVED | 15 | ORA - Zero Page,X |
| 04 | RESERVED | 16 | ASL - Zero Page,X |
| 05 | ORA - Zero Page | 17 | RESERVED |
| 06 | ASL - Zero Page | 18 | CLC |
| 07 | RESERVED | 19 | ORA - Absolute,Y |
| 08 | PHP | 1A | RESERVED |
| 09 | ORA - Immediate | 1B | RESERVED |
| 0A | ASL - Accumulator | 1C | RESERVED |
| 0B | RESERVED | 1D | ORA - Absolute,X |
| 0C | RESERVED | 1E | ASL - Absolute,X |
| 0D | ORA - Absolute | 1F | RESERVED |
| 0E | ASL - Absolute | 20 | JSR |
| 0F | RESERVED | 21 | AND - (Indirect,X) |
| 10 | BPL | 22 | RESERVED |
| 11 | ORA - (Indirect,Y) | 23 | RESERVED |

24 BIT - Zero Page

25 AND - Zero Page

26 ROL - Zero Page

27 RESERVED

28 PLP

29 AND - Immediate

2A ROL - Accumulator

2B RESERVED

2C BIT - Absolute

2D AND - Absolute

2E ROL - Absolute

2F RESERVED

30 BMI

31 AND - (Indirect),Y

32 RESERVED

33 RESERVED

34 RESERVED

35 AND - Zero Page,X

36 ROL - Zero Page,X

37 RESERVED

38 SEC

39 AND - Absolute,Y

3A RESERVED

3B RESERVED

3C RESERVED

3D AND - Absolute,X

3E ROL - Absolute,X

3F RESERVED

40 RTI

41 EOR - (Indirect,X)

42 RESERVED

43 RESERVED

44 RESERVED

45 EOR - Zero Page

46 LSR - Zero Page

47 RESERVED

48 PHA

49 EOR - Immediate

4A LSR - Accumulator

4B RESERVED

4C JMP - Absolute

4D EOR - Absolute

4E LSR - Absolute

4F RESERVED

50 BVC

51 EOR - (Indirect),Y

52 RESERVED

53 RESERVED

54 RESERVED

55 EOR - Zero Page,X

56 LSR - Zero Page,X

57 RESERVED

58 CLI

59 EOR - Absolute,Y

5A RESERVED

5B RESERVED

5C RESERVED

5D EOR - Absolute,X

5E LSR - Absolute,X

5F RESERVED

60 RTS

61 ADC - (Indirect,X)

62 RESERVED

63 RESERVED

64 RESERVED

65 ADC - Zero Page

66 ROR - Zero Page

67 RESERVED

68 PLA

69 ADC - Immediate

6A ROR - Accumulator

6B RESERVED

6C JMP - Indirect

6D ADC - Absolute

6E ROR - Absolute

6F RESERVED

70 BVS

71 ADC - (Indirect),Y

72 RESERVED

73 RESERVED

74 RESERVED

75 ADC - Zero Page,X

76 ROR - Zero Page,X

77 RESERVED

78 SEI

79 ADC - Absolute,Y

7A RESERVED

7B RESERVED

7C RESERVED

7D ADC - Absolute,X

7E ROR - Absolute,X

7F RESERVED

80 RESERVED

81 STA - (Indirect,X)

82 RESERVED

83 RESERVED

84 STY - Zero Page

85 STA - Zero Page

86 STX - Zero Page

87 RESERVED

88 DEY

89 RESERVED

8A TXA

8B RESERVED

8C STY - Absolute

8D STA - Absolute

8E STX - Absolute

8F RESERVED

90 BCC

91 STA - (Indirect),Y

92 RESERVED

93 RESERVED

94 STY - Zero Page,X

95 STA - Zero Page,X

96 STX - Zero Page,Y

97 RESERVED

98 TYA

99 STA - Absolute,Y

9A TXS

9B RESERVED

| | | | |
|---|---|---|---|
| 9C | RESERVED | B0 | BCS |
| 9D | STA - Absolute,X | B1 | LDA - (Indirect),Y |
| 9E | RESERVED | B2 | RESERVED |
| 9F | RESERVED | B3 | RESERVED |
| A0 | LDY - Immediate | B4 | LDY - Zero Page,X |
| A1 | LDA - (Indirect,X) | B5 | LDA - Zero Page,X |
| A2 | LDX - Immediate | B6 | LDX - Zero Page,Y |
| A3 | RESERVED | B7 | RESERVED |
| A4 | LDY - Zero Page | B8 | CLV |
| A5 | LDA - Zero Page | B9 | LDA - Absolute,Y |
| A6 | LDX - Zero Page | BA | TSX |
| A7 | RESERVED | BB | RESERVED |
| A8 | TAY | BC | LDY - Absolute,X |
| A9 | LDA - Immediate | BD | LDA - Absolute,X |
| AA | TAX | BE | LDX - Absolute,Y |
| AB | RESERVED | BF | RESERVED |
| AC | LDY - Absolute | C0 | CPY - Immediate |
| AD | LDA - Absolute | C1 | CMP - (Indirect,X) |
| AE | LDX - Absolute | C2 | RESERVED |
| AF | RESERVED | C3 | RESERVED |

C4 CPY - Zero Page

C5 CMP - Zero Page

C6 DEC - Zero Page

C7 RESERVED

C8 INY

C9 CPY - Immediate

CA DEX

CB RESERVED

CC CPY - Absolute

CD CMP - Absolute

CE DEC - Absolute

CF RESERVED

D0 BNE

D1 CMP - (Indirect,Y)

D2 RESERVED

D3 RESERVED

D4 RESERVED

D5 CMP - Zero Page,X

D6 DEC - Zero Page,X

D7 RESERVED

D8 CLD

D9 CMP - Absolute,Y

DA RESERVED

DB RESERVED

DC RESERVED

DD CMP - Absolute,X

DE DEC - Absolute,X

DF RESERVED

E0 CPX - Immediate

E1 SBC - (Indirect,X)

E2 RESERVED

E3 RESERVED

E4 CPX - Zero Page

E5 SBC - Zero Page

E6 INC - Zero Page

E7 RESERVED

E8 INX

E9 SBC - Immediate

EA NOP

EB RESERVED

EC CPX - Absolute

ED SBC - Absolute

EE INC - Absolute

EF RESERVED

F0 BEQ

F1 SBC - (Indirect),Y

F2 RESERVED

F3 RESERVED

F4 RESERVED

F5 SBC - Zero Page,X

F6 INC - Zero Page,X

F7 RESERVED

F8 SED

F9 SBC - Absolute,Y

FA RESERVED

FB RESERVED

FC RESERVED

FD SBC - Absolute,X

FE INC - Absolute,X

FF RESERVED

# APPENDIX I — AN INTRODUCTION TO SWEET-16

Deep inside the Integer BASIC ROMs lives a mysterious program known as "Sweet-16." Sweet-16 is a meta processor which is implemented interpreter style. Its virtues include a bunch of 16-bit instructions, most of which are implemented with one-byte opcodes. Since performing 16-bit operations with normal 6502 code requires several two- and three-byte instructions, Sweet-16 code is very compact. In this section we will explore the possibilities of the Sweet-16 interpreter, its advantages and disadvantages.

First, just exactly what is a "meta processor" and what does an interpreted implementation imply? A meta processor is simply a fantasy machine, one which does not exist as a physical machine, but simply as a design tool. A meta processor has the capability of taking on almost any instruction set. Since there are only a few pieces of hardware actually capable of performing this task (and the 6502 is not such a piece of hardware), a meta processor implementation must be handled in a somewhat different way on the 6502. An interpreter must be written, with a single subroutine for each instruction code to be implemented. A small control program picks up the Sweet-16 opcodes from memory, decodes the instruction, and then passes control to the appropriate subroutine. Once the desired subroutine is finished execution, the code control is returned to the control program which accesses the n byte of Sweet-16 code and continues the process.

So far everything sounds wonderful. But what are the disadvantages of Sweet-16 code? First, and probably most important, Sweet-16 programs run much slower than the same algorithm coded entirely in 6502 assembly language, five to seven times slower in fact. Another mark against Sweet-16 code is that the Sweet-16 interpreter exists only in the Integer BASIC ROMs (which is no big deal if you have an APPLE II computer, a language card, or an Integer BASIC card), but, if you only have an APPLE II Plus computer without Integer BASIC, or you wish to sell your programs to others who may not have the Integer BASIC language, you will either have to forget about Sweet-16 altogether or inject the code for the Sweet-16 interpreter into your program. Since the Sweet-16 interpreter is about 400 bytes long, you would have to write more than one kilobyte of code in Sweet-16 before it would pay to include the interpreter within your programs. Because of this problem, Sweet-16 should only be used where the Integer BASIC language is available. The interpreter is already provided ther for you (free-of-charge even!).

What does Sweet-16 look like? Sweet-16 is a 16-bit computer complete with sixteen 16-bit registers. These registers are used to hold addresses and intermediate values for use in address calculations. These registers are numbered R0 to RF (hex) for reference purposes. Several of these registers are special purpose. They include R0, RC, RE, and RF. R0 is the Sweet-16 accumulator. Sweet-16 can only perform the addition, subtraction, and comparison operations, and these must all be routed through the Sweet-16 accumulator. RC is the Sweet-16 stack pointer used when Sweet-16 processor status data and RF is the Sweet-16 program counter. Except for these four registers which are for special use only, all the Sweet-16 registers are general purpose address registers.

Before discussing how the Sweet-16 instruction set is used, entering and exiting the Sweet-16 mode must be covered. A program toggles back and forth between Sweet-16 code and 6502 code in much the same manner as you would toggle between the decimal mode and binary mode. A program enters the Sweet-16 mode with a JSR SW16 instruction. SW16 is

located at address $F689. Once this is accomplished, all further code is assumed to be Sweet-16 code. To terminate the Sweet-16 mode of operation, the Sweet-16 instruction "RTN" (for ReTurN to 6502 mode) must be executed immediately after the RTN instruction, valid 6502 instructions are expected. A quick excursion into Sweet-16 with an immediate return to 6502 mode would consist of the code sequence:

```
SW16    EQU $F689

        JSR SW16
        RTN

        RTS
        END
```

If this short program were executed, the JSR SW16 instruction would cause a transfer to the Sweet-16 mode to take place. All further instructions are assumed to be Sweet-16 instructions. The next instruction is the Sweet-16 RTN instruction which causes a transfer back to the 6502 mode. All instructions following the RTN instruction are assumed to be valid 6502 instructions. The next instruction is the familiar 6502 RTS instruction which causes a return to the Apple monitor. This simple sequence of instructions, although trivial and producing no noticeable results, demonstrates how to enter and terminate the Sweet-16 mode. Normally, several Sweet-16 instructions would be sandwiched between the JSR SW16 and the RTN instructions.

The Sweet-16 processor status word holds several conditions. A carry flag, zero flag, and negative flag are implemented. A test for minus one ($FFFF) is also implemented.

The Sweet-16 SET instruction allows the programmer to set the contents of any Sweet-16 register to a desired value. Its 6502 equivalent is the load immediate instruction. The SET instruction has the syntax:

SET Rn, ◄16-BIT VALUE►

The 16-bit value can be any valid LISA address expression. 'n' is simply a hex value in the range $0-$F and denotes which register is to be loaded with the declared value. Examples of the SET instruction:

```
LABEL   SET RO,LABEL        ;LOADS THE CURRENT ADDRESS
                            ;INTO RO
        SET R1,$25          ;LOADS $0025 INTO R1
        SET R5,$800         ;LOADS $0800 INTO R5
```

The SET instruction is three bytes long: one byte for the SET opcode and two bytes for the 16-bit value that is to be loaded into the specified register. SET RF,◄VALUE► is a very special case. Since RF is the Sweet-16 program counter, loading immediate data into register $F is the same as performing an absolute jump instruction. RC and RE must be treated carefully as well since they are used to hold the Sweet-16 stack pointer and status register. If zero is loaded into the specified register, the Sweet-16 zero flag is set; otherwise it is cleared. If minus one flag is set; otherwise the minus one flag is cleared. The Sweet-16 carry flag is always cleared after a SET instruction is executed.

The next instruction in the Sweet-16 instruction set is the load register or LDR instruction. This instrucion loads the Sweet-16 accumulator (R0) from the register specified in the

operand field. The term 'load' is somewhat misleading as this instruction really a register transfer instruction not unlike the 6502 TYA and TXA instructions. The LDR instruction has the syntax:

LDR Rn

Where n is the Sweet-16 register number in the range $0-$F Note that LDR is perfectly allowable and performs the operation of making a copy of R0 into R0, a somewhat useless instruction (except, possibly, for comparison purposes) but nevertheless valid. The LDR instruction is a one-byte instruction and will cause 16 bits to be transfered to the Sweet-16 accumulator. If zero is transferred between the registers, then the Sweet-16 zero flag is set; otherwise the zero flag is cleared. If minus one is transfered to the accumulator, the minus one flag is cleared. The negative flag is set according to the data transferred to the Sweet-16 accumulator. The negative flag always reflects the contents of the sixteenth bit, not the eighth bit as in the 6502 status register. The Sweet-16 carry flag is always cleared.

STO (store register) is the inverse operation to LDR. STO stores the contents of the Sweet-16 accumulator into the specified Sweet-16 register. This is similar to the 6502 instructions TAY & TAX. The Sweet-16 status bits are affected in the same manner as with the LDR instruction, and the STO instruction is one byte long, just like the LDR instruction.

You will note that there is no direct way to transfer the data from one register to another without going through the Sweet-16 accumulator. For example, to transfer the data from R5 to R6 you must execute the code sequence:

LDR R5
STO R6

As you can see, the Sweet-16 accumulator is destroyed during such transfers. For this very reason, the Sweet-16 accumulator should not be used to hold important data. It should be used totally as a transient register used only for calculations.

The Sweet-16 interpreter allows two types of arithmetic. 16-bit addition and subtraction. Addition is performed with the Sweet-16 ADD instruction. It takes a single register as its operand. This register is added to the Sweet-16 accumulator and the result is left in the accumulator. The syntax for the ADD instruction is:

ADD Rn

Where n is a hex value in the range $0-$F. Note that the instruction 'ADD R0' is very useful; it doubles the value in the Sweet-16 accumulator. If there is a carry out of the 17th bit during the addition, the carry is noted in the Sweet-16 carry flag. An add with carry instruction is not possible, so the carry flag is useful only for detecting overflow. All the other condition modes are set accroding to the outcome of the additional operation. The Sweet-16 ADD instruction is a one-byte instruction.

Subtraction is performed using the Sweet-16 SUB instruction. The register specified in the operand field is subtracted from the accumulator with the results being left in the accumulator. The SUB instruction can be used as a compare instruction in a manner similar to the SBC instruction on the 6502. If the value in the accumulator (prior to the SUB instruction) is greater than or equal to the value in the specified register, the carry flag will be set after the SUB instruction occurs. If the value in the accumulator is less than the value in the specified register, the carry flag will be clear after the SUB instruction is executed. If the two registers are equal, then the zero flag is set; if they are not equal, the zero flag is reset. Note that the SUB R0 instruction can be used as a one-byte clear accumulator instruction. It performs the same function as SET R0, yet requires only one third the memory.

Comparisons can also be performed using the CPR (compare register) instruction. CPR performs the same function as the SUB instruction, except that the results are placed in RD instead of the ACC. Any tests following the CPR instruction will test the value in RD instead of the accumulator. Register RD can be thought of as an auxillary processor status register. As such, its use should also be avoided.

Conditions in the Sweet-16 processor status register are tested in a manner very similar to the 6502 microprocessor. That is, branch instructions are used to test conditions. Branches on the Sweet-16 processor use relative addressing, just like their 6502 counterparts. The branch instructions include: BRA (branch always, an uncontitional branch), BNC (branch if no carry), BIC (branch if carry), BIP (branch if positive), BIM (branch if minus), BIZ (branch if zero or branch if equal), BNZ (branch if not zero or not equal), and BSB (branch to Sweet-16 subroutine). All Sweet-16 branches are two bytes long.

The branch to subroutine (BSB) instruction really needs some additional explanation. When a Sweet-16 subroutine is called, the return address is pushed onto the Sweet-16 return address stack. The pointer is RC. Wherever RC happens to be pointing when the BSB instruction is executed, the return address will be stored. If you have not initialized the Sweet-16 stack pointer (RC), it could be pointing anywhere in memory, which means that a BSB instruction could potentially wipe out valuable program and data storage.

The cure for these ailments is always to initialize the Sweet-16 stack pointer prior to using Sweet-16 subroutines. This is accomplished quite easily by using the SET instruction and loading RC with an intial stack pointer value (this is similar to using the 6502 sequence: LDX #VALUE, TXS). Unlike the 6502 stack pointer which is an 8-bit register that wraps around, the Sweet-16 stack pointer is a 16-bit register which can take on any 16-bit value. This means that if you're not very careful, it is possible to have the stack go wild and wipe out everything in memory. Typically, you will not have to even use Sweet-16 subroutines, but should the need arise, be very careful.

To return from a Sweet-16 subroutine you must use the RSB (return from subroutine) instruction. The RSB instruction is a single byte instruction.

Register increments and decrements are performed by the INR and DCR instructions. INR increments the register specified in the operand field by one; DCR decrements the specified register by one. All branch conditions are set to reflect the final results in the specified register. The INR and DCR instructions are both one byte long.

So far, only a discussion of the arithmetic and conditional testing capabilities of the Sweet-16 processor have been presented. Although these instructions are useful, they do not really present anything new that was not already available in the 6502 microprocessor instruction set. Sweet-16's real power comes from its pointer and data movement capabilities. Several powerful load and store instructions are available which allow the programmer to perform certain actions in one byte that would take eight to sixteen bytes on the 6502. These instructions revolve around the idea of loading the Sweet-16 accumulator indirectly through a specified register.

The first instruction in this family of instructions is the load indirect instruction. It uses the syntax:

LDR@Rn

Note that the mnemonic is the same as the normal load register instruction, but that the '@' character appears in the operand field immediately before the register specifier. This instruction is an 8-bit load instruction. It loads the low-order eight bits of the Sweet-16 accumulator from the memory location pointed to by the specified register. The high-order byte of the Sweet-16 accumulator is cleared. After the accumulator is loaded with the data from the address pointed to by Rn, Rn is incremented by one. This has the effect of causing the pointer register to point to the next available byte immediately after the LDR instruction is executed. This type of instruction (where the register is automatically incremented for you) is called an "auto-increment" instruction. The LDR indirect instruction is very useful for memory movements and searches. Consider the following code:

```
START   JSR SW16
        SET R1,$8000
        SET R3,1FF
LOOP    LDR @Rn

        CPR R3          ;CHECK FOR $FF
        BNZ LOOP        ;LOOP IF NOT FOUND
        RTN             ;QUIT SWEET-16, DATA FOUND
                        ;ADDRESS LEFT IN R1
```

This routine starts at location $8000 and searches diligently intil a $FF is encountered. To load two bytes into the accumulator one would use the LDD (load double indirect) instrtuction. It uses the syntax:

LDD @Rn

It loads the low order accumulator byte from the location pointed at by Rn; then Rn is incremented by one. After the increment is performed, the high order accumulator byte is loaded indirectly through the new value in Rn. Once this is accomplished, Rn is again incremented. The net result is that the Sweet-16 accumulator is loaded indirectly from the locations pointed at by Rn and Rn+1. Afterwards Rn is incremented twice. The branch conditions will reflect the final accumulator contents and the carry will be cleared.

Data can also be stored indirectly through one of the registers. The store indirect instruction is the inverse of the load indirect instruction. It has the syntax:

STO @Rn

This instruction stores the contents of the low-order byte of the Sweet-16 accumulator at the location in memory pointed to by the Rn register. After the store operation is performed, Rn is incremented by one. The branch conditions reflect the Sweet-16 accumulator contents. The store indirect instruction can be used rather well with the load indirect instruction for memory movement routines. The following routine moves data from $8000 through $9000 to the area $3000 through $4000:

```
START   JSR SW16
MOVE    SET R1,$8000        ;SET UP POINTER REG #1
        SET R2,$9000        ;SET UP FINAL VALUE REG
        SET R3,$3000        ;SET UP POINTER REG #2

LOOP    LDR @R1             ;GET DATA @R1

        STO @R3             ;STORE @R3
```

```
LDR R1
CPR R2          ;DONE YET?
BNC LOOP        ;IF NO CARRY (I.E. LESS THAN)
BIZ LOOP        ;IF EQUAM
RTN
BRK
END
```

Compare this to the amount of code required to perform the same operation in 6502 machine code!

To store both halves of the Sweet-16 accumulator into memory, you must use the STD (store double indirect) instruction. This instruction stores the low-order byte of the Sweet-16 accumulator at the location pointed to by Rn. Rn is then incremented by one, and the high-order byte of the accumulator is then stored at the new location pointed to by Rn, after which Rn is again incremented by one.

The last three Sweet-16 instructions are POP (pop indirect), STP (store pop indirect), and PPD (pop double indirect). POP loads the low-order accumulator byte from the location pointed to by Rn AFTER Rn is decremented by one. POP has the syntax:

POP @Rn

User-defined stacks may be implemented using the POP Rn and STO Rn instructions (where Rn is the stack pointer). POP is also useful in implementing the "move right" routine presented elsewhere in this book.

STP is the inverse of POP. This operation causes the low-order byte of the Sweet-16 accumulator to be stored at the address pointed to by Rn after Rn is decremented by one. Single byte user-defined stacks may also be implemented using the STP Rn and LDR Rn instructions (where Rn is the user-defined stack pointer).

PPD (pop double indirect) is the 2-byte equivalent of POP. PPD performs the following action: Rn is decremented by one and the high-order accumulator byte is loaded from the location pointed to by Rn. Rn is then again decremented by one and the low-order accumulator byte is loaded from the address pointed to by Rn. PPD has the syntax:

PPD @Rn

Double byte stacks may be implemented using the PPD and STD instructions. The POP, STP, and PPD instructions are all one byte long. The carry is always cleared after one of these operations is performed. POP always results in a positive value which is never minus one. PPD and STP affect the status bits depending upon the final accumulator contents.

## SWEET-16 HARDWARE REQUIREMENTS

All of the Sweet-16 registers are implemented as zero page memory locations (in fact, the first 32 bytes of zero page are used for the Sweet-16 registers). For this reason, care must be exercised when using zero page memory in a program in which Sweet-16 is also used. R0 corresponds to memory locations $0 and $1; R1 corresponds to memory locations $2 and $3; and so on for the other registers. Since they are implemented in zero page memory, it is a simple matter for 6502 programs to pass data to a Sweet-16 routine simply by shoving data into the respective registers. Likewise, Sweet-16 can return data to the 6502 program in the Sweet-16 registers. A Sweet-16 call is transparent to the 6502 program. All registers, including the processor status register, are preserved and then restored before returning to the

6502 mode. Another important fact to remember is that the 6502 must be in the binary (as opposed to decimal) mode before entering the Sweet-16 mode. Strange things happen if this is not the case.

# APPENDIX J — USING THE HIRES ROUTINES SUPPLIED WITH LISA

Included with LISA Version 2.0 is a very flexible set of hires subroutines. These routines contain considerable internal documentation. It must be realized that these routines are are intended for assembly language programmers only. For the most part these routines cannot be called from INTEGER BASIC, APPLESOFT, or TINY PASCAL.

In addition to being easy to use, the routines are very fast. The package itself is 3 1/2 to 4K bytes long. Much of the code has been replicated in several parts of the program to save time (remember, it takes 12 microseconds to service a subroutine call). 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. In addition a SYMBOL routine allows you to draw any of the 96 printable ASCII characters onto the screen with descenders on the lower case and a screen format of 46 x 30.

When you load the HIRES routines into the text file buffer, the first thing you will notice is a series of JMP statements. These jumps form a vector table. Each jump corresponds to one of the aforementioned routines.

Normally your assembly language routines will JSR to one of these JMP's instead of JSRing to the actual routine. The purpose behind this is two fold. First, it allows you to modify any of the internal procedures without having to reassemble any existing software (perhaps to speed up a particular routine). Second, you only need to know one address (the address of the beginning of these routines) in your assembly language programs. This way, should you desire to re-ORG your entire program you need only change one line instead of 16.

Immediately following the JMP's come several tables. Ignore these for now.

Most of the routines in this HIRES package work on the principle of an NXM picture. A picture is nothing more than a dot matrix. For example, the ASCII characters are composed of 5 x 7 pictures. PROGRAMMA's Hires Character generator, and others, all use a 5 x 7 picture. The HIRES routines provided with LISA allow you to define a picture of any size up to 256 x 192. Furthermore, unlike the hires character generators, a picture may be drawn at any valid location on the screen. This allows you to write letters diagonally on the screen dropping down only one bit instead of a whole row with each character.

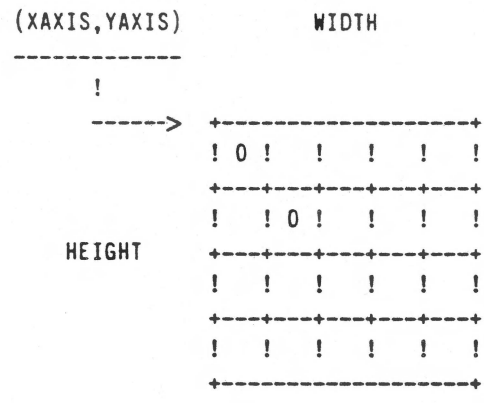
Pictures are positioned on the screen at the X,Y location contained in the bytes XAXIS,XAXIS+$1, and YAXIS. Pictures are positioned with the upper leftmost bit at (XAXIS,YAXIS). An explanation of pictures, and how to draw them, will follow shortly.

The first usable routine after the data tables is the RADAR function. RADAR is 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
```

where width is the number of dots in the X-direction (from XAXIS) and height is the number of dots in the Y-direction (from YAXIS). Height and width, together with (XAXIS,YAXIS) form a matrix or rectangle on the CRT:




MATRIX FORMED BY
HEIGHT, WIDTH, XAXIS, YAXIS

When called, RADAR checks this rectangle for any ON dots. If any dots were found to be ON, RADAR will return with the overflow flag set. If no dots where found in the range specified, then RADAR returns with the overflow flag cleared. RADAR is very useful in testing for "collisions."

DRAWLN and ERSLN draw and erase lines on the hires screen. The line is drawn from (XAXIS,YAXIS) to (DESTX,DESTY). You simply store the desired values in the proper locations and then JSR DRAWLN or JSR ERSLN.

Two very useful subroutines which you will want to use are XY= and DXY=. XY= gives you an easy method of initializing XAXIS and YAXIS to a particular value. XY= is called in the following fashion:

```
JSR  XY=
ADR  XVALUE
HEX  YVALUE
```

where XVALUE is a 16-bit quantity in the range 0-279 and YVALUE is an eight bit quantity in the range 0-191. This data is stored in (XAXIS,YAXIS).

EXAMPLE:

```
JSR  XY=
ADR  $FF    ;PUT $00FF IN XAXIS
HEX  26     ;PUT $26 IN YAXIS
```

DXY= allows the programmer to initialize (DESTX,DESTY) and (XAXIS,YAXIS) in a manner similar to XY=. DXY= is called in the following manner:

```
JSR  DXY=
ADR  DESTX VALUE
HEX  DESTY VALUE
ADR  XAXIS VALUE
HEX  YAXIS VALUE
```

This routine is particularly useful with DRAWLN and ERSLN.

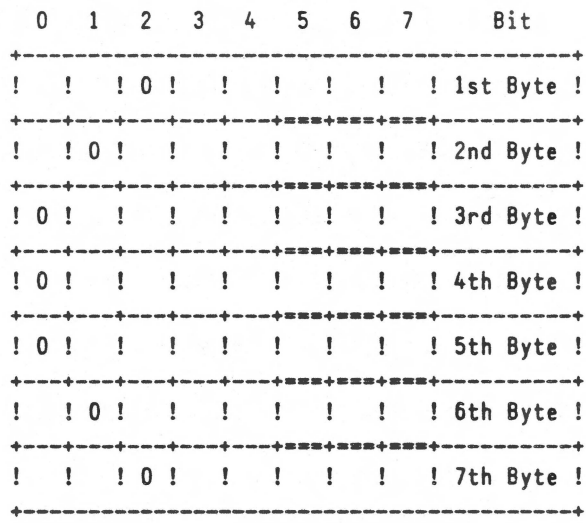
The subroutine SYMBOL allows the user to display ASCII characters on the hires screen. The characters are user definable by modifying the character table at the beginning of the HIRES routines (more on modifying these characters later). To use SYMBOL you must first load (XAXIS,YAXIS) with the (X,Y) coordinate of where you wish the text to appear on the screen. This (X,Y) coordinate points to the upper left hand dot of the first character in the string. Once the correct value is placed in (XAXIS,YAXIS), you may call SYMBOL as follows:

```
JSR  SYMBOL
ASC  "ANY STRING"
HEX  00
```

Note that the string must be terminated with a HEX 00. Control characters are not allowed in the string.

Immediately following the JMP vectors in the HIRES routines comes the character definition table. Each character in the table is made up of 7 bytes. In each byte, 5 bits are utilized which gives a 5 x 7 dot matrix.

Organization of 1 character:




The character "(" is shown. It has the character table value $04, $02, $01, $01, $01, $02, $04.

You may go in and modify any of the ASCII characters you so desire. This feature allows you to "customize" your character sets (to allow Greek letters for instance).

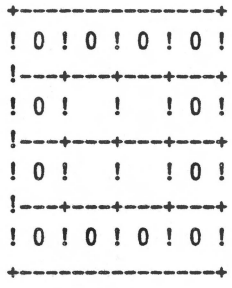
If you want to animate figures on the screen, the HIRES package provides four very useful subroutines for doing just that. These routines display a picture on the hires screen.

A picture is simply a dot matrix which you wish to have appear on the video screen. A picture has two attributes, a width and a height. Width is the number of bits which are required in the XAXIS, height is the number of bits required in the YAXIS. When defining a picture you must first specify the width and height.

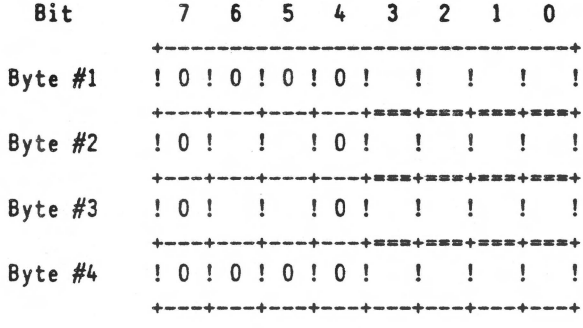
As an example, suppose we want to define a 4 x 4 box. We would start off with the assembly language statement:

```
BOX    HEX  04 04
```

This tells the HIRES routines that our box will be 4 dots wide and 4 dots high. The first byte in the HEX string ("04") is the width, the second byte is the height. Now, to draw a box we must get out a piece of graph paper and draw out the outline of our box:



The HIRES picture routines require that all new rows in a picture begin on a byte boundary. Since only four bits are used in each row this means that 4 bits will be wasted. The box is coded into the picture table in the following manner:



```
BOX     HEX  04 04 ;WIDTH & HEIGHT
        HEX  F0    ;BYTE #1
        HEX  90    ;BYTE #2
        HEX  90    ;BYTE #3
        HEX  F0    ;BYTE #4
```

Other than for legibility there is no reason why all these HEX digits didn't appear on the same line:

```
BOX     HEX  04 04 F0 90 90 F0
```

Please note that unlike the character table pictures, the 8th bit (bit No. 7) corresponds to the first dot which will be plotted on the screen. Of course a picture may be more than 8 dots wide, just remember that new rows must begin on byte boundaries.

The HIRES routines provided with LISA also include a SHAPE subroutine. Basically, the SHAPE subroutine allows you to draw the desired picture onto the screen and then SHAPE converts it to a picture table. SHAPE is called in the following manner:

```
        JSR  SHAPE
        ADR  PICTURE
        ADR  PICADR
```

PICTURE points to the ASCII representation of the picture (described later), and PICADR points to a block of memory where the resulting picture table may be stored.

At least ((WIDTH/8)+1)*LENGTH+2 bytes should be reserved at PICADR for the picture table.

PICTURE is the ASCII representation of the dot matrix you wish to create. As usual, the first two bytes at PICTURE contain the width and the height of the picture. The next (width * height) bytes contain blanks or non-blank character specifying whether or not a bit is to be set.

EXAMPLE:

```
BOXPIC HEX  04 04
       ASC  "*****"
       ASC  "*   *"
       ASC  "*****"

       JSR  SHAPE
       ADR  BOXPIC
       ADR  BOX
```

When used as shown, BOX will contain the correct picture table to draw the desired shape.

The last (and most useful) routines in the HIRES package allow you to draw and erase pictures to and from the screen. PIX simply draws a picture onto the screen. As usual (XAXIS,YAXIS) contains the coordinate of the upper left most dot of the picture.

A call is made to PIX with the address of the picture table immediately following the JSR:

```
       JSR  PIX
       ADR  BOX
```

PIX draws the picture verbatim onto the hires screen, with each "1" becoming a white dot and each "0" becoming a black dot.

ORPIX is very similar to PIX except the picture is OR'd onto the screen instead of drawn verbatim.

ANDDIX is used to erase a picture on the screen. Each occurrance of a "1" in a picture will correspond to a black dot on the hires screen.

XORPIX exclusive-OR's the picture to the screen allowing some interesting effects.