MAC/65

TABLE OF CONTENTS

INTRODUCTION

This manual assumes the user is familiar with assembly language. It is not intended to teach assembly language. This manual is a reference for commands, statements, functions, and syntax conventions of MAC65. It is also assumed that the user is familiar with the screen editor of the Atari or Apple II computer, as appropriate. Consult Atari's or Apple's Reference Manuals if you are not familiar with the screen editor.

If you need a tutorial level manual, we would recommend that you ask your local dealer or bookstore for suggestions. Two books that have worked well for many of our customers are "Machine Language for Beginners" by Richard Mansfield from

COMPUTE! books and "Programming the 6502" by Rodney Zaks.

This manual is divided into two major sections; the first two chapters cover the Editor commands and syntax, source line entry, and executing source program assembly. The next three chapters then cover instruction format, assembler directives, functions and expressions, Macros, and conditional assembly.

MAC65 is a fast and powerful machine language development tool. Programs larger than memory can be assembled. MAC65 also contains directives specifically designed for screen format development. With MAC65's line entry syntax feature, less time is spent re-assembling programs due to assembly syntax errors, allowing more time for actual program development.

START UP

Power up the disk drive(s) and monitor, leave the computer off. Insert MAC65 disk in drive #1 and boot system by turning the computer on. This will load and execute DOS XL. Now enter MAC65 (return). This loads and executes MAC65, the Editor/Macro Assembler. Refer to the DOS XL Manual for other capabilities.

WARM START

The user can exit to DOSXL by entering the MAC65 command CP (return) or by pressing the System Reset key. To return to MAC65, the user can use the DOSXL command RUN (return). This "warm starts" MAC65 and does not clear out any source lines in memory.

BACK-UP COPY

Please do not work with your master disk! Make a back-up copy with DOSXL. Consult the DOSXL reference manual for specific instructions. Keep your master copy in a safe place.

SYNTAX

The following conventions are used in the syntax descriptions in this manual:

1. Capital letters designate commands, instruction, functions, etc., which must be entered exactly as shown (e.g. ENTER, .INCLUDE, .NOT).

2. Lower case letters specify items which may be used. The various types are as follows:

1no - Line number between 0-65535, inclusive.

hxnum - A hex number. It can be address or data. Hex numbers are treated as unsigned integers.

dcnum - A positive number. Decimal numbers are rounded to the nearest two byte unsigned integer; 3.5 to 3.9 is rounded to 4 and 100.1 to 100.4 is rounded to 100.

exp - An assembler expression.

string - A string of ASCII characters enclosed by double quotes (eg. "THIS IS A STRING").

strvar - A string representation. Can be a string as above, or a string variable within a Macro call (eg. %$1).

filespec OR file - A string of ASCII characters that refers to refers to a particular device. See device reference manual for more specific explanation.

3. Items in square brackets denote an optional part of syntax (eg. [,1no]). When an optional item is followed by (...) the item(s) may be repeated as many times as needed.

```
   Example:  .WORD exp [,exp ...]
```

4. Items in parentheses indicate that any one of the items may be used, eg. (.Q) (,A).

CHAPTER 1: THE EDITOR

The Editor allows the user to enter and edit MAC/65 source code or ordinary ASCII text files.

To the Editor, there is a real distinction between the two types of files; so much that there are actually two modes accessible to the user, EDIT mode and TEXTMODE. However, for either mode, source code/text must begin with a line number between 0 and 65535 inclusive, followed by one space.

```
     Examples:  10 LABEL LDA  #$32
                3020 This is valid in TEXT MODE
```

The first example would be valid in either EDIT or TEXTMODE, while the second example would only be valid in TEXTMODE.

The user chooses which mode he/she wishes to use for editing by selecting NEW (which allows general text entry). There is more discussion of the impact of these two modes below; but, first, there are several points in common to the two modes.

1.1 GENERAL EDITOR USAGE

The source file is manipulated by Editor commands. Since the Editor recognizes a command by the absence of a line number, a line beginning with a line number is assumed to be a valid source/text line. As such, it is merged with, added to, or inserted into the source/text lines already in memory in accordance with its line number. An entered line which has the same line number as one already in memory will replace the line in memory.

Also, as a special case of the above, a source line can be deleted from memory by entering its line number only. (And also see DEL command for deleting a group of lines.)

Any line that does not start with a line number is assumed to be command line. The Editor will examine the line to determine what function is to be performed. If the line is a valid command, the Editor will execute the command. The Editor will prompt the user each time a command has been executed or terminated by printing:

```
     EDIT     for syntax (MAC/65 source) mode
     TEXTMODE for text mode
```

The cursor will appear on the following line. Since some commands may take a while to execute, the prompt signals the user that more input is allowed. The user can terminate a command before completion by hitting the break key (escape key on Apple II).

And one last point: If the line is neither a source line or a valid command. The Editor will print:

What?

1.2 TEXT MODE

The Editor supports a text mode. The text mode is entered with the command TEXT. This mode will NOT syntax check lines entered, allowing the user to enter and edit non-assembly language files. All Editor commands function in text mode.

Remember, though, that all text lines must begin with a line number; and, even in TEXTMODE, the space following the line number is necessary.

1.3 EDIT MODE

MAC/65 is nearly unique among assembler/editor systems in that it allows the assembly language user to enter source code and have it IMMEDIATELY checked for syntax validity. Of course, since assembly language syntax is fairly flexible (especially when macros are allowable, as they are with MAC/65), syntax checking will by no means catch all errors in user source code. For example, the existence of and validity of labels and/or zero page locations is not and can not be checked until assembly time. However, we still feel that this syntax checking will be a boon to the beginner and experienced programmer alike.

Again, remember that source lines must begin with a line number which must, in turn, be followed by one space. Then, the second space after the line number is the label column. The label must start in this column. The third space after the line number is the instruction column. Instructions may either start in at least the third column after the line number or at least one space after the label. The operand may begin any where after the instruction, and comments may begin any where after the operand or instruction. Refer to Assembler Section for specific instruction syntax.

As noted, the Editor syntax checks each source line at entry. If the syntax of a line is in error, the Editor will list the line with a cursor turned on (i.e., by using an inverse or blinking character) at the point of error.

The source lines are tokenized and stored in memory, starting at an address in low memory and building towards high memory. The resultant tokenized file is 60% to 80% smaller than its ASCII counterpart, thus allowing larger programs to be entered and edited in memory.

SPECIAL NOTE: If, upon entry, a source line contains a syntax error and is so flagged by the Editor, the line is entered into Editor memory anyway. This feature allows raw ASCII text files (possibly from other assemblers and possibly containing one or several syntax errors as far as MAC/65 is concerned) to be ENTERed into the Editor without losing any lines. The user can note the lines with errors and then edit them later.

CHAPTER 2: EDITOR COMMANDS

This chapter lists all the valid Editor-level commands, in alphabetical order, along with a short description of the purpose and function of each.

Again, remember that when the "TEXTMODE" or "EDIT" prompt is present any input line not preceded by a line number is presumed to be an Editor command.

If in the process of executing a command any error is encountered, the Editor will abort execution and return to the user, displaying the error number and descriptive message of the error before re-prompting the user. Refer to Appendix for possible causes of errors.

Section 2.1

edit command: ASM

purpose: ASseMble MAC/65 source files

usage: ASM [#file1],[#file2],[#file3],[#file4]

ASM will assemble the specified source file and will produce a listing and object code output; the listing may include a full cross reference of all non-local labels. File1 is the source device, file2 is the list device, file3 is the object device, and file4 is a temporary file used to help generate the cross reference listing.

Any or all of the four filespec's may be omitted, in which case MAC/65 assumes the following default filespec(s) are to be used:

file1 - user source memory
file2 - screen editor.
file3 - memory (CAUTION: see below)
file4 - none, therefore no cross reference

A filespec (#file1, #file3, etc.) can be omitted by substituting a comma in which case the respective default will be used.

```
     Example:  ASM #D2:SOURCE,#D:LIST,#D2:OBJECT
```

In this example, the source will come from D":SOURCE, the assembler will list to D:LIST, and the object code will be written to D":OBJECT.

```
     Example:  ASM #D:SOURCE ,,#D:OBJECT
```

In this example, the source will be read from D:SOURCE and the object will be written to D:OBJECT. The assembly listing will be written to the screen.

```
     Example:  ASM , #P: ,,#D:TEMP
```

In this example, the source will be read from memory, the object will be written to memory (but ONLY if the ".OPT OBJ" directive is in the source), and the assembly listing will be written to the printer along with the complete label cross reference. The file TEMP on disk drive 1 will be created and used as a temporary file for the cross reference.

```
     Example:  ASM #D:SOURCE .#P:
```

In this example, the source will be read from D:SOURCE and the assembly listing will be written to the printer. If the ".OPT OBJ" directive has been selected in the source, the object code will be placed in memory.

Note: If assembling from a "filespec", the source MUST have been a SAVEd file.

Note: Refer to the .OPT directive for specific information on assembler listing and object output.

Note: The object code file will have the format of compound files created by the DOSXL SAVE command. See the DOSXL manual for a discussion of LOAD and SAVE file formats.

Section 2.2

edit command: BLOAD

purpose: allows user to LOAD Binary (memory image) files from disk into memory

usage: BLOAD #filespec

The BLOAD command will load a previously BSAVEd binary file, an assembled object file, or a binary file created with DOSXL SAVe command.

     Example: BLOAD #D:OBJECT

This example will load the binary file "OBJECT" to memory at the address where it was previously saved from or assembler for.

CAUTION: it is suggested that the user only BLOAD files which were assembled into MAC/65's free area (as shown by the SIZE command) or which will load into known safe areas of memory.

Section 2.3

edit command: BSAVE

purpose: SAVE a Binary image of a portion of memory. Same as DOSXL SAVE command

usage: BSAVE #filespec < hxnum1 ,hxnum2

The BSAVE command will save the memory addresses from hxnum1 through hxnum2 to the specified device. The binary file created is compatible with the DOSXL SAVe command.

     Example: BSAVE #D:OBJECT< (,)(hxnum) [(,)(,hxnum) ...]

Although MAC/65 does not included a debug capability, there are a few machine level commands included for the convenience of the user who would, for example, like to change system registers and the like (screen color, margins, etc.). The C command is provided for this purpose.

C allows the user to modify memory. Hxnum1 is the change start address. The remaining hxnum(s) are the change bytes. The comma will skip an address.

     Example: C 50000" and "<" have quite different meanings when used as unary operators.

3.5.4 Operators: .OR .AND .NOT

These operators also perform logical operations and should not be confused with their bitwise companions. Remember, these operators always return only TRUE or FALSE.

```
EXAMPLES:      3 .OR 0        returns 1
               3 .AND 2       returns 1
               6 .AND 0       returns 0
               .NOT 7  returns 0
```

3.5.5 Operator:- (unary)

The minus sign may be used as a unary operator. Its effect is the same as if a minus sign had been used in a binary operation where the first operator is zero.

```
EXAMPLE:       -2 is $FFFE (same as 0-2)
```

3.5.6 Operators: < > (unary)

These UNARY operators are extremely useful when it is desired to extract just the high order or low order byte of an expression label. Probably their most common use will be that of supplying the high and low order bytes of an address to be used in a "LDA #" or similar immediate instruction.

```
EXAMPLE:       FLEEP = $3456
                LDA #FLEEP (same as LDA #$34)
```

3.5.7 Operator:.DEF

This unary operator tests whether the following label has been defined yet, returning TRUE or FALSE as appropriate.

CAUTION: Defining a label AFTER the use of a .DEF which references it can be dangerous, particularly if the .DEF is used in a .IF directive.

```
EXAMPLE:         .IF .DEF ZILK
                 .BYTE "generate some bytes"
                 .ENDIF
               ZILK = $3000
```

In this example, the .BYTE string will NOT be generated in the first pass but WILL be generated in the second pass. Thus, any following code will almost undoubtedly generate a PHASE ERROR.

3.5.8 Operator:.REF

This unary operator tests whether the following label has been referenced by any instruction or directive in the assembly yet; and, in conjuction with the .IF directive, produces the effect of returning a TRUE or FALSE value.

Obviously, the same cautions about .DEF being used before the label definition apply to .REF also, but here we can obtain some advantage from the situation.

```
EXAMPLE:        .IF .REF PRINTMSG
               PRINTMSG
               ...(code to implement the PRINTMSG routine)
             .ENDIF
```

In this example, the code implementing PRINTMSG will ONLY be assembled if something preceding this point in the assembly has referred to the label PRINTMSG! This is a very powerful way to build an assembly language library and assemble only the needed routines. Of course, this implies that the library must be .INCLUDEd as the last part of the assembly, but this seems like a not too onerous restriction. In fact, OSS has used this technique in writing the libraries for the C/65 compiler.

CAUTION: note that in the description above it was implied that .REF only worked properly with a .IF directive. Not only is this restriction imposed, but attempts to use.REF in any other way can produce bizarre results. ALSO, .REF can not effectively be used in combination with any other operators. Thus, for example,

```
     .IF .REF ZAM .OR .REF BLOOP is ILLEGAL!
```

The only operator which can legally combined with .REF is .NOT, as in .IF .NOT .REF LABEL.

Note that the illegal line above could be simulated thus:

```
     EXAMPLE:  DOIT . = 0
                 .IF .REF ZAM
               DOIT . = 1
                 .IF .REF BLOOP
               DOIT . = 1
                 .ENDIF
                 .IF DOIT
                 ...
```

3.5.9 Operator: [ ]

MAC/65 supports the use of the square brackets as "psuedo parentheses". Ordinary round parentheses may NOT be used for grouping expressions, etc., as they must retain their special meanings with regards to the various addressing modes. In general, the square brackets may be used any where in a MAC/65 expression to clarify or change the order of evaluation of the expression.

```
     EXAMPLES:
          LDA GEORGE+5*3        ;This is legal, but
                                 it multiplies 3*5
                                 and adds the 15 to
                                 GEORGE...probably
                                 not what you wanted.
          LDA (GEORGE+5)*3      ;Syntax Error!!!
          LDA [GEORGE+5]*3      ;OK...the addition
                                 is performed before
                                 the multiplication
          LDA ( [GEORGE+5]*3),Y ;See the need
                                 for both kinds of
                                 "parentheses"?
```

REMEMBER: Operators in MAC/65 expressions follow precedence rules. The square brackets may be used to override these rules.

3.6 ASSEMBLER EXPRESSIONS

An expression is any valid combination of operands and operators which the assembler will evaluate to a 16-bit unsigned number with any overflow ignored. Expressions can be arithmetic or logical. The following are examples of valid expressions:

```
10  .WORD  TABLEBASE+LINE+COLUMN
55  .IF .DEF INTEGER .AND [ VER=1 .OR VER >=3 ]
200 .BYTE >EXPLOT-1, >EXDRAW-1, >EXFILL-1
300 LDA  # < ADDRESS^-1 ] +1
305 CMP  # -1
400 CPX  # 'A
440 INC  #1+1
```

3.7 OPERATOR PRECEDENCE

The following are the precedence levels (high to low) used in evaluating assembler expressions:

```
[ ] (psuedo parenthesis)
 > (high byte), < (low byte), .DEF, .REF, - (unary)
.NOT
*, /
+, -
&, !,  ^
=, >. <=, >=, <> (comparison operators)
.AND
.OR
```

Operators grouped on the same line have equal precedence and will be executed in left-to-right order unless higher precedence operator(s) intervene.

3.8 NUMERIC CONSTANTS

MAC/65 accepts three types of numeric constants: decimal, hexadecimal, and characters.

A decimal constant is simply a decimal number in the range 0 through 65535; an attempt to use a decimal number beyond these bounds may or may not work and will certainly produce unexpected and undesired results.

```
     EXAMPLES:  1  234  65200  32767
     (as used:) .BYTE 2,4,8,16,32,64
                LDA #1
```

A hexadecimal constant consists of a dollar sign followed by one to four legal hexadecimal digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F). Again, usage of more than four digits may produce unwanted results.

```
     EXAMPLES:  $1 $EA $FF00 $7FFF
     (as used:) .WORD $100,$200,$400,$800,$1000
                AND #$7F
```

A character constant is an apostrophe followed by any printable or displayable character. The value of a character constant is the ASCII (or ATASCII) value of the character following the apostrophe.

```
     EXAMPLES:  'A '* '" '=
     (as used:) CMP #'=
                CMP #'Z+1 ; same as #$5B
```

3.9 STRINGS

Strings are of two types. String literals (example: "This is a string literal"), and string variables for Macros (example: %$5).

```
     Example:  10  .BYTE "A STRING OF CHARACTERS"
                    or
     Example:  20  .SBYTE %$1
```

CHAPTER 4: DIRECTIVES

As noted in Section 3.1, the instruction field of an assembled line may contain an assembler directive (instead of a valid 6502 instruction). This chapter will list and describe, in roughly alphabetical order, all the directives legal under MAC/65 (excepting directives specific to macros, which will be discussed separately in Chapter 5).

Directives may be classified into three types: (1) those which produce object code for use by the assembled program (e.g., .BYTE, .WORD, etc.); (2) those which direct the assembler to perform some task, such as changing where in memory the object code should go or giving a value to a label (e.g., *=, =, etc.); and (3) those which are provided for the convenience of the programmer, giving him/her control over listing format, location of source, etc. (e.g., .TITLE, .OPT, .INCLUDE).

Obviously, we could in theory do without the type 3 directives; but, as you read the descriptions that follow, you will soon discover that in practice these directives are most useful in helping your 6502 assembly language production. Incidentally, all the macro-specific directives could presumably be classified as type 3.

Three of the directives which follow (.PAGE, .TITLE, and .ERROR) allow the user to specify a string (enclosed in quotes) which will be printed out. For these three directives, the user is limited to a maximum string length of 70 characters. Strings longer than 70 characters will be truncated.

Section 4.1

directive: *=

purpose: change current origin of the assembler's location counter

usage: [label] *= expression

The *= directive will assign the value of the expression to the location counter. The expression cannot be forward referenced. *= must be written with no intervening spaces.

```
     Example:  50 *= $1234 ;sets the location
                             counter to $1234
```

Another common usage of *= is to reserve space for data to be filled in or used at run time. Since the single character "*" may be treated as a label referencing the current location counter value, the form "*= *+exp" is thus the most common way to reserve "exp" bytes for later use.

```
     Example:  70 LOC *= *+1 ;assigns the current
                              value of the location
                              counter to LOC and
                              then advances the
                              counter by one.
```

(Thus LOC may be thought of as a one byte reserved memory cell.)

CAUTION: Because any label associated with this directive is assigned
the
value of the location counter BEFORE the directive is executed, it is
NOT
advisable to give a label to "*=" unless, indeed, it is being used as
in the
second example (i.e., as a memory reserver).

NOTE: Some assemblers use "ORG" instead of "*=" and may also have a
separate
directive (such as "DS" or "RMB") for "defining storage" or "reserving
memory
bytes". Use caution when converting from and to such assemblers; pay
special
attention to label usage. When in doubt, move the label to the next
preceding
or next following line, as appropriate.

Section 4.2

directive: =

purpose: assigns a value to a label

usage: label = expression

The = directive will equate "label" with the value of the expression.
A
"label" can be equated via "=" only once within a program.

```
     Example:  10 PLAYER0 = PMBASE + $200
```

Note: If a "label" is equated more than once, "label" will contain the
value
of the most recent equate. This process, however, will result in an
assembly
error.

Section 4.3

directive: .=

purpose: assign a possibly transitory value to a label

usage: label .= expression

The .= directive will SET "label" with the value of the expression. Using this directive, a "label" may be set to one or more values as many times as needed in the same program.

```
     EXAMPLE:

     10 LBL  .= 5
     20     LDA #LBL  ;same as LDA #5
     30 LBL  .= 3+'A
     40     LDA #LBL  ;same as LDA #68
```

CAUTION: A label which has been equated (via the "=" directive) or assigned a value through usage as an instruction label may not then be set to another value by ".=".

Section 4.4

```
directive:  .BYTE   [and .SBYTE]

purpose:    specifies the contents of individual
            bytes in the output object

usage:
[label] .BYTE  [+exp,] (exp)(strvar)[,(exp)(strvar) ...]
[label] .SBYTE [+exp,](exp)(strvar)[,(exp)(strvar) ...]
```

The .BYTE and .SBYTE directives allow the user to generate individual bytes of memory image in the output object. Expressions must evaluate to an 8-bit arithmetic result. A strvar will generate as many bytes as the length of the string. .BYTE simply assembles the bytes as entered, while .SBYTE will convert the bytes to Atari screen codes (on the Atari) or to characters with their most significant bit on (on the Apple II).

```
     Example:  100  .BYTE "ABC" , 3, -1
```

This example will produce the following output bytes:

```
     41 42 43 03 FF.
```

Note that the negative expression was truncated to a single byte value.

```
     Example:  50   .SBYTE "Hello!"
```

On the Atari, this example will produce the following screen codes:

```
     28 65 6C 6C 6F 01.
```

On the Apple II, the same example would produce the following bytes:

```
     C8 E5 EC EC DF A1.
```

SPECIAL NOTE: Both .BYTE and .SBYTE allow an additive Modifier. A Modifier is an expression which will be added to all of bytes assembled. The assembler recognizes the Modifier expression by the presence of the "+" character. The Modifier expression will not itself be generated as part of the output.

```
     Example:  5  .BYTE +$80 , "ABC" , -1
```

This example will produce the following bytes:

```
     C1 C2 C3 7F
```

```
     Example:  100  .BYTE +$80,"DEF",'G+$80
```

This example will produce: C4 C5 C6 47.

(Note especially the effect of adding $80 via the modifier and also adding it to the particular byte. The result is an unchanged byte, since we have added a total of 256 ($100), which does not change the lower byte of a 16 bit result.)

```
     Example:  55  .SBYTE +$40 , "A12"
```

This example will produce:

```
     61 51 52 Atari
     01 F1 F2 Apple II.
```

```
     Example:  80  .SBYTE +$C0,"G-$C0,"REEN"
```

This example will produce:

```
     27 F2 E5 E5 EE Atari
     C7 92 85 85 8E Apple II.
```

Note: .SBYTE performs its conversions according to a numerical algorithm and does NOT special case any control characters, including BELL, TAB, etc.--these characters ARE converted.

Section 4.5

directive: .CBYTE

```
purpose:    same as .BYTE except that the most
            significant bit of the last byte of a
            string argument is inverted
```

usage:

```
[label] .CBYTE [+exp,](exp)(strvar) [,(exp)(strvar)...]
```

The .CBYTE directive may often be used to advantage when building tables of strings, etc., where it is desirable to indicate the end of a string by some method other than, for example, storing a following zero byte. By inverting the sense of the upper bit of that last character of the string, a routine reading the strings from the table could easily do a BMI or BPL as it reads each character.

```
     Example:  ERRORS .CBYTE 1,"SYSTEM"
```

The line shown would produce these object bytes:

```
     01 53 59 53 54 45 CE
```

And a subroutine might access the characters thus:

```
          LDY #1
LOOP      LDA ERRORS,Y
          BMI ENDOFSTRING
          INY
          BNE LOOP
          ...
ENDOFSTRING
          ...
```

Section 4.6

Directive: DBYTE [see also .WORD]

purpose: specifies Dual BYTE values to be
            placed in the output object.

usage: [label] .DBYTE exp [,exp ...]

Both the .WORD and .DBYTE directives will put the value of each expression into the object code as two bytes. However, while .WORD will assemble the expression(s) in 6502 address order (least significant byte, most significant byte), .DBYTE will assemble the expression(s) in the reverse order (i.e., most significant byte, least significant byte).

.DBYTE has limited usage in a 6502 environment, and it would most probably be used in building tables where its reversed order might be more desirable.

```
     EXAMPLE:  .DBYTE $1234,1,-1
               produces:  12 34 00 01 FF FF
               .WORD $1234,1,-1
               produces:  34 12 01 00 FF FF
```

Section 4.7

directive: .ELSE

purpose: SEE description of .IF for purpose nd usage.

Section 4.8

directive: .END

purpose: terminate an in-memory assembly

usage: [label] .END

The .END directive will terminate the assembly ONLY if the source is being read from memory. Otherwise, .END will have no effect on assembly.

This "no effect" is handy in that you may thus .INCLUDE file(s) without•having to edit out any .END statements they might contain. In truth, .END is generally not needed at all with MAC/65,

Section 4.9

directive: .ENDIF

purpose: terminate a conditional assembly block

SEE description of .IF for usage and details.

Section 4.10

directive: .ERROR

purpose: force an assembler error and message

usage: [label] .ERROR [string]

The .ERROR directive allows the user to generate a pseudo error. The string specified by .ERROR will be sent to the screen as if it were an assembler-generated error. The error will be included in the count of errors given at the end of the assembly.

```
     Example:  100  .ERROR "MISSING PARAMETER!"
```

Section 4.11

directive: .FLOAT

purpose: specifies floating point constant values
            to be placed in the output object.

usage:

[label] .FLOAT floating-constant [,flotation-constant...]

This directive would normally only be used by the programmer wishing to access the built-in floating point routines of the Atari Operating System ROM's (or similar routines as supplied with the BASIC XL package from OSS for Apple II or equivalent machines).

Each floating point constant following the .FLOAT directive will produce 6 bytes of bytes of output object code, in a format consistent with the above-mentioned floating point routines. In particular, the first byte contains the exponent portion of the number, in excess-64 notation representing power of 100. The upper bit of the exponent byte designates the sign of the mantissa portion. The following 5 bytes are the mantissa, inn packed BCD form, normalized on a byte boundary (consistant with the powers-of-100 exponent).

```
     EXAMPLES:
          .FLOAT 3.14156295,-2,718281828
```

The above example would produce the following bytes in the output object code:

```
     40 03 14 15 62 95
     C0 27 18 28 18 28
```

NOTE: Only floating point constants, NOT expressions, are legal as operands to .FLOAT. Generally, this is not a problem, since the user may perform any constant arithmetic on a calculator (or in BASIC) before placing the result in his/her MAC/65 program.

Section 4.12

directive: .IF

purpose: choose to perform or not perform some portion of an assembly based on the "truth" of an expression.

```
usage:      .IF  exp
            [.ELSE]
            .ENDIF
```

usage note: there may be any number of lines of assembly language code or directives between]IF and .ELSE or .ENDIF and similarly between .ELSE and .ENDIF.

When a .IF is encountered, the following expression is evaluated. If it is non-zero (TRUE), the source lines following .IF will be assembled, continuing until an .ELSE or .ENDIF is encountered. If an .ELSE is encountered before an .ENDIF, then all the source lines between the .ELSE and the corresponding .ENDIF will not be assembled. If the expression evaluates to zero (false), the source lines following .IF will not be assembled. Assembly will resume when a corresponding .ENDIF or an .ELSE is encountered.

The .IF-.ENDIF and .IF-.ELSE-.ENDIF constructs may be nested to a depth of 14 levels. When nested, the "search" for the "corresponding" .ELSE or .ENDIF skips over complete .IF-.ENDIF constructs if necessary.

Examples:

```
10  .IF 1       ; non-zero, therefore true
20  LDA # '?    ; these two lines will
30  JSR CHAROUT ; be assembled
40  .ENDIF
```

EXAMPLE:

```
10   .IF 0          ; expression is false
11   LDA # >ADDRESS ; these two lines will
12   LDX #
%1 ; get the high byte of parameter 1.
15 CMP (%11 ,X) ;yes, that really is number 11.
20 .BYTE %2-1  ;value of parameter 2 less 1.
```

NOTE: the above is NOT equivalent to using parameter %1. Parameter substitution has highest precedence!

```
25 SYMBOL .= SYMBOL + 1
30 LDX # -%(SYMBOL) ; see the power available?
40 .BYTE %$1,%$2,0 ; string parameters, ending 0.
```

Remember, in theory the parameters are numbered from 1 to 63. In reality, the TOTAL number of parameters in use by all active (nested) macro expansions cannot exceed 63. This does NOT mean that you can have only 63 parameter references in your macro DEFINITIONS. The limit only applies at invocation time, and even then only to nested (not sequential) macro usages.

SPECIAL NOTE: In addition to the "conventional" parameters, referred to by number, parameter zero (%0) has a special meaning to MAC/65. Parameter zero allows the user to access the actual NUMBER of real parameters passed to a macro EXPANSION.

This feature allows the user to set default parameters within the Macro expansion, or test for the proper number of parameters in an expansion, or more. The following example illustrates a possible use of %0 and shows usage of ordinary parameters as well.

EXAMPLE:

```
10 .MACRO BUMP
11 ;
12 ; This macro will increment the specified word
13 ;
14 ; The calling format is:
15 ;      BUMP address [ ,increment ].
16 ; If increment is not given, 1 is assumed
17 ;
18 .IF%0=0 .OR %0>2
19 .ERROR "BUMP": Wrong number of parameters"
20 .ELSE
21 ;
22 ; this is only done if 1 or 2 parameters
23 ;
24  .IF $0>1 ; did user specify "increment" ?
25 ; this is assembled if user gave two parameters
26 LDA %1    ; add "increment" to "address".
27 CLC
28 ADC #
```