MAC/65 6502 (HTML conversion by SysOp Fox-1, November 1999)

Note from Fox-1:
This manual is based on the DISK-version of the MAC/65 assembler. Text printed on a YELLOW background is additional information taken from the CARTRIDGE-version (Rev 1.2) of the manual of the MAC/65, and is typed up by an anonymous user who feels that this info is useful (like I do) for the ATARI community.

Programming Info Index To Main-Index

TABLE OF CONTENTS

Chapter 3 -- The Macro Assembler

Chapter 4 -- Directives

Chapter 9 -- Error Descriptions

Appendix -- A

Programming Info Index Top of Page

PREFACE

MAC/65 is a logical upgrade from the OSS product EASMD (Edit/ASseMble/Debug) which was itself an outgrowth of the Atari Assembler/Editor cartridge. Users of either of these latter two products will find that MAC/65 has a very familiar "feel". Those who have never experienced previous OSS products in this line should nevertheless find MAC/65 to be an easy-to-use, powerful and adaptable programming environment. While speed was not necessarily the primary goal in the production of this product, we nevertheless feel that the user will be hard pressed to find a faster assembler system in any home computer market. MAC/65 is an excellent match for the size and features of the machines it is intended for.

MAC/65 was conceived by and completely executed by Stephen D. Lawrow. The current version of MAC/65 is only the latest in a series of increasingly more complex and faster assemblers written by Mr. Lawrow following the lead and style of EASMD. As a measure of our confidence in this assembler, it is entrusted with assembling itself, probably a more difficult task than that to which most users will put it.

Table of Contents

TRADEMARKS

The following trademarked names are used in various places with this manual, and credit is hereby given:

DOS XL, BASIC XL, MAC/65, and C/65 are trademarks of Optimized Systems Software, Inc.

Atari, Atari 400, Atari 800, Atari Home Computers, and Atari 850 Interface Module are trademarks of Atari, Inc., Sunnyvale, CA.

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.

Table of Contents

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.

introduction Table of Contents

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.

startup Table of Contents

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.

warmstart Table of Contents

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).

MAC/65 in EDIT mode is NOT case sensitive.
Inverse video characters are uninverted.
Lower case letters are converted to upper case.

EXCEPTIONS: characters between double quotes, following a single quote, or in the comment field of a MAC/65 source line will remain unchanged. Text entered in TEXT mode, though, will not be changed.

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).

fspec
or
filespec - A string of ASCII characters
that refers to OR refers to a
particular device. See device
file reference manual for more
specific explanation.
Might be #D1:SOURCE.M65 or #E: or #P:

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).

back-up copy Table of Contents

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.

Table of Contents

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?
```

chapter 1 Table of Contents

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.

chapter 1 Table of Contents

Section 2.1 (ASM)

edit command: ASM

purpose: ASseMble MAC/65 source files

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

usage: ASM [#fspec1],[#fspec2],[#fspec3],[#fspec4]

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.

Example: ASM,#-

This produces what is probably the fastest possible MAC/65 assembly. Source code is read from memory and no listing is produced (because of the "#-"). If your program does not contain an ".OPT OBJ" line, this becomes what is essentially simply an error checking assembly. (Though even if you ARE producing object code, the assembly speed is extremely fast.)

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.

NOTE: You may use #C: as a device for the listing or object files. You may NOT use #C: for the source or cross-reference files (you will not get a cross-reference unless you use a disk drive). HOWEVER, we do not recommend using the cassette as the object file device, since you may get an excessively long leader tone (which will be difficult to re-BLOAD later). Instead, we suggest using BSAVE (after assembling directly to memory) whenever practicable.

Section 2.2 (BLOAD)

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 assembled 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.

chapter 2 Table of Contents

Section 2.3 (BSAVE)

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<5000,5100

This example will save the memory addresses from $5000 through $5100 to the file "OBJECT".

chapter 2 Table of Contents

Section 2.4 (BYE)

edit command: BYE

purpose: exit to system monitor level

usage: BYE

BYE will put the user to the Atari Memo Pad or Apple II monitor, as appropriate.

chapter 2 Table of Contents

Section 2.5 (Change Memory)

edit command: C

purpose: Change memory contents

usage: c hxnum1 < (,)(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<20,00,D8,,5

The example will change the memory addresses as follows: 5000 to 20, 5001 to 00, 5002 to D8, skip 5003, and change 5004 to 5.

chapter 2 Table of Contents

Section 2.6 (Display Memory)

edit command: D

purpose: Display contents of memory location(s)

usage: D hxnum1 [,hxnum2]

D allows the user to examine memory. If hxnum2 is specified, the memory locations between hxnum1 and hxnum2 will be displayed, else only hxnum1 through hxnum1 +8 will be displayed.

chapter 2 Table of Contents

Section 2.x (DDT)

edit command: DDT

purpose: enter the DDT debug package, which
is part of the MAC/65 cartridge.

usage: DDT

Once you have entered this command, DDT is entered and has control of the system. However, DDT saves enough of MAC/65's vital memory that, if you follow certain simple rules, you may return to MAC/65 from DDT with your source program still intact. The DDT manual gives more information on this subject, but as a general guide you must avoid locations $80 through $AF (in zero page) and the memory location located within the bounds displayed by the SIZE command. See the DDT manual (which is bound with but after this MAC/65 manual) for many, many more details.

chapter 2 Table of Contents

Section 2.7 (DEL)

edit command: DEL

purpose: DELetes a line or group of lines from
the source/text in memory.

usage: DEL 1nol [,1no2]

DEL deletes source lines from memory. If only one 1no is entered, only the line will be deleted. If two 1nos are entered, all lines between and including 1no1 and 1no2 will be

deleted.

Note: 1no1 must be present in memory for DEL to execute.

chapter 2 Table of Contents

Section 2.8 (DOS)

edit command: DOS [or, equivalently, CP]

purpose: exit from MAC/65 to the CP of DOS XL.

usage: DOS
or
CP

Either DOS or CP returns the user to DOSXL.

chapter 2 Table of Contents

Section 2.9 (ENTER)

edit command: ENTER

purpose: allow entry of ASCII (or ATASCII)
text files into MAC/65 editor memory

usage: ENTER #filespec [(,M) (,A)]

ENTER will cause the Editor to get ASCII text from the specified device. ENTER will clear the text area before entering from the filespec. That is any user program is memory at the time the ENTER command is given will be erased.

The parameter "M" (MERGE) will cause MAC/65 to NOT clear the text area before entering from the file, text entered will be merged with the text in memory. If a line is entered which has the same line number of a line in memory, the line from the device will overwrite the line in memory.

The parameter "A" allows the user to enter un-numbered text from the specified device. The Editor will number the incoming text starting at line 10, in increments of 10.

CAUTION: The "A" option will always clear the text area before entering from the filespec.

chapter 2 Table of Contents

Section 2.10 (FIND)

edit command: FIND

purpose: to FIND a string of characters some where in MAC/65's editor buffer.

usage: FIND / string/ [1no1 [,1no2]] [,A]

The FIND command will search all lines in memory or the specified line(s) (1no1 through 1no2) for the "string" given between the matching delimiter. The delimiter may be any character except a space. If a match is found, the line containing the match will be listed to the screen.

Note: do NOT enclose a string in double quotes.

Example: FIND/LDX/

This example will search for the first occurance of "LDX".

Example: FIND\Label\25,80

This example will search for the first occurance of "Label" in lines 25 through 80.

If the option "A" is specified, all matches within the specified line range will be listed to the screen. Remember, if no line numbers are given, the range is the entire program.

chapter 2 Table of Contents

Section 2.11 (LIST)

edit command: LIST

purpose: to LIST the contents of all or part of MAC/65's editor buffer in ASCII (ATASCII) form to a disk or device.

usage: LIST [#filespec,] [1no1 [,1no2]]

LIST lists the source file to the screen, or device when "#filespec" is specified. If no 1nos are specified, listing will begin at the first line in memory and end with the last line in memory.

If only 1no1 is specified, that line will be listed if it is in memory. If 1no1 and 1no2 are specified, all lines between and including 1no1 and 1no2 will be listed. When 1no1 and 1no2 are specified, neither one has to be in memory as LIST will search for the first line in memory greater than or equal to 1no1, and will stop listing when the line in memory is greater than 1no2.

EXAMPLE: LIST #P:
will list the current contents
of the editor memory to the P:
(printer) device.

EXAMPLE: LIST #D2:TEMP, 1030, 1000
lists only those lines lying
in the line number range from
1030 to 1800, inclusive, to the
disk file named "TEMP" on disk
drive 2.

NOTE: The second example points out a method of moving or duplicating large portions of text or source via the use of temporary disk files. By suitably RENumbering the in-memory text before and after the LIST, and by then using ENTER with the Merge option, quite complex movements are possible.

chapter 2 Table of Contents

Section 2.12 (LOAD)

edit command: LOAD

purpose: to reLOAD a previously SAVEd MAC/65 token
file from disk to editor memory.

usage: LOAD #filespec [,A]

LOAD will reload a previously SAVEd tokenized file into memory. LOAD will clear the user memory before loading from the specified device unless the ",A" parameter is

appended.

The parameter "A" (for APPEND) causes the Editor to NOT clear the text area before loading from the file. Instead, the load file will be appended with the current file in memory.

Note: The Append option will NOT renumber the file after loading. It is possible to have DUPLICATE LINE NUMBERS. Use the REN command if there are duplicate line numbers.

chapter 2 Table of Contents

Section 2.13 (LOMEM)

edit command: LOMEM

purpose: change the lower bound of editor memory
usable by MAC/65.

usage: LOMEM hxnum

LOMEM allows the user to select the address where the source program begins. Executing LOMEM clears out any source currently in memory; as if the user had typed "NEW".

chapter 2 Table of Contents

Section 2.14 (NEW)

edit command: NEW

purpose: clears out all editor memory, sets
syntax checking mode.

usage: NEW

NEW will clear all user source code from memory and reset the Editor to syntax mode. The "EDIT" prompt appears, reminding the user that syntax checking is now active. If the user needs to defeat the syntax checking, he/she must use the TEXT command.

chapter 2 Table of Contents

Section 2.15 (NUM)

edit command: NUM

purpose: initiates automatic line NUMbering mode

usage: NUM [dcnum1 [,dcnum2]]

NUM will cause the Editor to auto-number the incoming text from the Screen Editor (E:). A space is automatically printed after the line number. If no dcnums are specified, NUM will start at the last line number plus 10. NUM dcnum1 will start at the last line number plus "dcnum1" in increments of "dcnum1". NUM dcnum1". NUM dcnum1, dcnum2 will start at "dcnum1" in increments of "dcnum2".

EXAMPLE: NUM 1000,20
will cause the Editor to prompt the user with the number "1000" followed by a space. When the user has entered a line, the next prompt will be "1020", etc.

The NUM mode will terminate if the line number which would be next in sequence is present in memory.

The user may terminate NUM mode on the Atari by pressing the BREAK key or by typing a CONTROL-3. On the Apple, the user may terminate the NUM mode by pressing CONTROL-C followed by RETURN.

chapter 2 Table of Contents

Section 2.16 (PRINT)

edit command: PRINT

purpose: to PRINT all or part of the Editor text or source to a disk file or a device.

usage: PRINT [#filespec,] [1no1 [,1no2]]

Print is exactly like LIST except that the line numbers are not listed. If a file is PRINTed to a disk, it may be reENTERed into the MAC/65 memory using the ENTER

command with the Append line number option.

chapter 2 Table of Contents

Section 2.17 (REN)

edit command: REN

purpose: RENumber all lines in Editor memory.

usage: REN [dcnum1 [,dcnum2]]

REN renumbers the source lines in memory. If no dcnums are specified, REN will renumber the program stating at line 10 in increments of 10. REN dcnum1 will renumber the lines starting at line 10 in increments of dcnum1. REN dcnum1, dcnum2 will renumber starting at dcnum1 in increments of dcnum2.

chapter 2 Table of Contents

Section 2.18 (REP)

edit command: REP

purpose: REPlaces occurrence(s) of a given string
with another given string.

usage:
REP/old string/new string/ [1no1 [,1no2]] [(,A)(,Q)]

The REP command will search the specified lines (all or 1no1 through 1no2) for the "old string".

The "A" option will cause all occurrences of "old string" to be replaced with "new string". The "Q" option will list the line containing the match and prompt the user for the change (Y followed by RETURN for change, RETURN for skip this occurance.) If neither "A" or "Q" is specified, only the first occurrence of "old String" will be replaced with "new string". Each time a change is made, the line is listed.

Example: REP/LDY/LDA/100,250,Q

This example will search for the string "LDY" between the lines 200 and 250,

inclusive, and prompt the user at each occurrence to change or skip.

Note: Hitting BREAK (ESCape on Apple Ii) will terminate the REP mode and return to the Editor.

Note: If a change causes a syntax error in the line, the REP mode will be terminated and control will return to the Editor. Of course, if TEXTMODE is selected, there can be no syntax errors.

chapter 2 Table of Contents

Section 2.19 (SAVE)

edit command: SAVE

purpose: SAVEs the internal (tokenized) form
of the user's in-memory text/source
to a disk file.

usage: SAVE #filespec

SAVE will save the tokenized user source file to the specified device. The format of a tokenized file is as follows:

File Header
Two byte number (LSB,MSB) specifies the
size of the file in bytes.

For each line in the file:
Two byte line number (LSB,MSB)
followed by
One byte length of line (actually offset to next line)
followed by
The tokenized line

chapter 2 Table of Contents

Section 2.20 (SIZE)

edit command: SIZE

purpose: determines and displays the SIZE of
various portions of memory used by
the MAC/65 Editor.

usage: SIZE

SIZE will print the user LOMEM address, the highest used memory address, and the highest usable memory address, in that order, using hexadecimal notation for the addresses.

chapter 2 Table of Contents

Section 2.21 (TEXT)

edit command: TEXT

purpose: allow entry of arbitrary ASCII (ATASCII)
text without syntax checking.

usage: TEXT

TEXT will clear all user source code from memory and put the Editor in the textmode. After this command is used, the Editor will prompt the user for new commands and text with the word "TEXTMODE" (instead of "EDIT"), indicating that no syntax checking is taking place.

TEXTMODE may be terminated by the NEW command. CAUTION: there is no way to go back and forth between syntax (EDIT) mode and TEXTMODE without clearing the Editor's memory each time.

chapter 2 Table of Contents

Section 2.22 (HEX/DEC Convert)

edit command: ?

purpose: makes hexadecimal/decimal conversions

usage: ? ($hxnum) (dcnum)

? is the resident hex/decimal decimal/hex converter. Numbers in the range 0 - 65535 decimal (0000 to FFFF hex) may be converted.

Example: ? $1200 will print =4600
? 8190 will print =$1FFE

chapter 2 Table of Contents

CHAPTER 3: THE MACRO ASSEMBLER

Usually, the Assembler is entered from MAC/65 with the command ASM. For ASM command syntax, refer to section 2.1 (in the Editor commands). Assembly may be terminated by hitting the BREAK key (ESCape key on the Apple II).
However, MAC/65 also offers the DOSXL command line level an optional ability to bypass the Editor phase entirely. This is especially useful when doing assemblies during the processing of an EXeCution file. To invoke the assembler directly, simply include one or more file names on the same DOSXL command line as the "MAC65" command. The formal usage is as follows:

MAC65 [file1 [file2 [file3 [file4]]] [-A][-D]]

where "file1", "file2", "file3" and "file4" are legal DOSXL file or device names and "-A" and "-D" are option specifiers. Thus the arguments are an optional set of one to four filenames, construed to be the source, listing, object, and cross-reference files (respectively) of a MAC/65 assembly.

And the options avaliable are:

-A source file is Ascii
-D assembly must be Disk-to-Disk

Remember, if no filenames are given, MAC/65 will be invoked in it's interactive (Editor) mode. But, if one or more files are specified, MAC/65 will be invoked in its "batch" mode. That is, it will perform a single assembly and then return to DOSXL. Generally, this command line will perform the assembly in a manner equivalent to giving the "ASM" command from the MAC/65 Editor. That is, if only one filename is given, it is assumed to be the source file, implying that the listing will go to the screen and the object code will be placed in memory (but only if requested by the .OPT OBJ directive). If a second filename is given, it is assumed to be the name of the listing file. Only if three or four filenames are given will the object code be directed to the file specified. And, finally, if the fourth filename is given it must be a disk filename and will be used as a temporary file for the cross reference listing.

Note: if an assembly needs no listing but does need an object file, the user may specify

"-" as the listing file.

And some notes on the options:

The -A option is used to specify that the source file is not a standard MAC/65 SAVEd file but is instead an Ascii (or Atascii) file. This is equivalent to using the interactive Editor mode of MAC/65 to use the sequence of commands "ENTER #D..." and "ASM ,...".

The -D option is used to specify that the assembly MUST proceed from disk to disk. If this option is not given, the source file is LOADed (or ENTERed) before the assembly, and then the assembly proceeds with the source in memory (generally producing improved speed of assembly). If, however, the source file is too large to be assembled in memory, the user may use this option to allow assembly of even very large programs. (And remember, even if the source fits, the macro and symbol tables must reside in memory during assembly also.)

NOTE: the -D option can NOT be used in conjunction with the -A option. The source file assembled under the -D option MUST be a properly SAVEd (tokenized) file.

EXAMPLES:

```
MAC65 JUNK.M65 - JUNK.COM
  will assemble JUNK.M65, producing no
  listing but sending the object code
  to the file JUNK.COM

MAC65 TEST.LIS P: TEST.OBJ TEST.XRF
  will assemble TEST,LIS, which is an
  ASCII file, sending the listing to
  the printer (P:) and the object to
  the file TEST.OBJ. A cross reference
  of all labels will be appended to the
  printer listing, and the file TEST.XRF
  will be used by MAC/65 as a temporary
  file for this purpose.
```

chapter 2 Table of Contents

Section 3.1 (Assembler Input)

The Assembler will get a line at a time from the specified device or from memory. If assembling from a device, the file must have been previously SAVEd by the Editor. All

discussions of source lines and syntax will be at the Editor line entry level. The tokenized (SAVEd) form is discussed in general terms under the SAVE command, section 2.19

Source lines are in the forms:

line number + mandatory space + source statement

The source statement may be in one of the following forms:

[label] [(6502 instruction) (directive)] [comment]

The following examples are valid source lines:

```
100 LABEL
120 ;Comment line
140 LDA #5 and then any comment at all
150 DEY
160 ASL A double number in accumulator
170 GETNUM LDA (ADDRESS),Y
180 .PAGE "directives are legal, too"
```

In general, the format is as specified in the MOS Technology 6502 Programming Manual. We recommend that the user unfamiliar with 6502 assembly language programming should purchase:

"Machine Language for Beginners" by R. Mansfield
or
"Programming the 6502" by Rodney Zake
or
any other book which seems compatible with the
users current knowledge of assembly language.

Special Note:
The assembler of MAC/65 understands only upper case labels, op codes, etc. HOWEVER, the editor (see especially section 1.3) will convert all lower case to upper case (except in comments and quoted strings), so the user may feel free to type and edit in which ever case he/she feels most comfortable with.

chapter 3 Table of Contents

Section 3.2 (Instruction Format)

A) Instruction mnemonics are as described in the MOS Technology Programming Manual.

B) Immediate operands begin with "#".

C) "(operand,X)" and "(operand),Y" designate indirect addressing.

D) "operand,X" and "operand,Y" designate indexed addressing.

E) Zero page operands cannot be forward referenced. Attempting to do so will usually result in a "PHASE ERROR" message.

F) Forward equates are evaluated within the limits of a two pass assembler.

G) "*" designates the current location counter.

H) Comment lines may begin with ";" or "*".

I) Hex constants begin with "$".

J) The "A" operand is reserved for accumulator addressing.

K) A semicolon ";" anywhere in a line indicates the beginning of the comment field for that line.

L) The addressing formats available are extended to allow the new addressing modes available with the NCR 65C02 microprocessor. See Chapter 7 for the descriptions of 65C02 instructions not included in the standard 6502 set. The extensions include:

1: "(operand)", indicating indirect addressing, is now legal with ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA. The operand must be in zero page.

2: "(operand,X)" is now legal when used with JMP. The operand here may be any absolute address.

3: The BIT instruction is allowed the addressing mode "operand,X". The operand may be either a zero page or absolute address.

4: The mnemonics BRA, DEA, INA, PHX, PHY, PLX, PLY, STZ, TRB, and TSB are now recognized.

chapter 3 Table of Contents

Section 3.3 (LABELS)

Labels must begin with an Alpha character, "@". or "?". The remaining characters may be as the first or may be "0" to "9" or ".". The characters must be uppercase and cannot be broken by a space. The maximum number of characters in a label is 127, and ALL are significant.

Labels beginning with a question mark ("?") are assumed to be "LOCAL" labels. Such labels are "visible" only to code encountered within the current local region. Local regions are delimited by successive occurrences of the .LOCAL directive, with the first region assumed to start at the beginning of the assembly source, whether or not a .LOCAL is coded there or not. There are a maximum of 62 local regions in any one assembly. Of course, if a .LOCAL is not encountered any where in the assembly, then all labels are accessible at all times. In any case, labels beginning with a question mark will NOT be listed in the symbol table.

The following are examples of valid labels:

```
TEST1 @.INC LOCATION LOC22A WHAT?
ADDRESS1.1 EXP.. SINE45TAB.
```

chapter 3 Table of Contents

Section 3.4 (OPERANDS)

An operand can be a label, a Macro parameter, a numeric constant, the current program counter (*), "A" for accumulator addressing, an expression, or an ASCII character. The following are examples of the various types of operands:

```
10 LDA #VALUE    ; label
15 ROR A         ; accumulator addressing
20 .BYTE 123,$45 ; numeric constants
25 .IF %0        ; Macro parameter
30 CMP #'A       ; ASCII character
35 THISLOC = *   ; current PC
40 .WORD PMBASE+[PLNO+4]*256 ; expression
```

chapter 3 Table of Contents

Section 3.5 (OPERATORS)

The following are the operators currently supported by MAC/65:

[ ] psuedo parentheses
+ addition
- subtraction
/ division
* multiplication
& binary AND
! binary OR
^ binary EOR
= equality, logical
> greater than, logical
< less than, logical
< > inequality, logical
>= greater or equal, logical
<= less or equal, logical
.OR logical OR
.AND logical AND
- unary minus
.NOT unary logical. Returns true (1) if expression is zero. Returns false (0) it expression is non-zero.
.DEF unary logical label definition. Returns true if label is defined.
.REF unary logical label reference. Returns true if label has been referenced.
> unary. Returns the high byte of the expression.
< unary. Returns the low byte of the expression.

Logical operators will always return either TRUE (1) OR FALSE (0). However, any non-zero value is considered true when making a conditional test.

Some of these operators perhaps need some explanation as to their usage and purpose. The operators are thus described in groups in the following subsections.

chapter 3 Table of Contents

Section 3.5.1 (Operators: + - * /)

These are the familiar arithmetic operators. Remember, though, that they perform 16-bit signed arithmetic and ignore any overflows. Thus, for example, the value of $FF00+4096 is $0F00, and no error is generated.

section 3.5 Table of Contents

Section 3.5.2 (Operators: & ! ^)

These are the binary or "bitwise" operators. They operate on values as 16 bit words, performing bit-by-bit ANDs, ORs, or EXCLUSIVE ORs. They are 16 bit equivalents of the 6502 opcodes AND, ORA, and EOR.

EXAMPLES: $FF00 & $0FF is $0000
$03 ! $0A is $000B
$003F ^ $011F is $0120

section 3.5 Table of Contents

Section 3.5.3 (Operators: = > < <> >= <=)

These are the familiar comparison operators. They perform 16 bit unsigned compares on pairs of operands and return a TRUE (1) or FALSE (0) value.

EXAMPLES: 3 < 5 returns 1
5 < 5 returns 0
5 <= 5 returns 1

CAUTION: Remember, these operators always work on PAIRS of operands. The operators ">" and "<" have quite different meanings when used as unary operators.

section 3.5 Table of Contents

Section 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

section 3.5 Table of Contents

Section 3.5.5 (Operator: - (unary))

The minus sign may be used as an unary operator. It's 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)

section 3.5 Table of Contents

Section 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 or 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 instruction.

EXAMPLE: FLEEP = $3456
LDA #<FLEEP (same as LDA #$56)
LDA #>FLEEP (same as LDA #$34)

section 3.5 Table of Contents

Section 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 "generates some bytes"

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.

section 3.5 Table of Contents

Section 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 conjunction 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
  ...
```

section 3.5 Table of Contents

Section 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.

section 3.5 Table of Contents

Section 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
```

chapter 3 Table of Contents

Section 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.

chapter 3 Table of Contents

Section 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

chapter 3 Table of Contents

Section 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 3 Table of Contents

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.

chapter 3 Table of Contents

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.

chapter 4 Table of Contents

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.

chapter 4 Table of Contents

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 ".=".

chapter 4 Table of Contents

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.

chapter 4 Table of Contents

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
     ...
```

chapter 4 Table of Contents

Section 4.6 (directive: .DBYTE)

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

chapter 4 Table of Contents

Section 4.x (directive: .DS)

purpose: reserves space for data without initializing the space to any particular value(s).

usage: [label] .DS expression

Using ".DS expression" is exactly equivalent of using "*= *+expression". That is, the label (if it is given) is set equal to the current value of the location counter. Then the value of the expression is added to the location counter.

```
Example:  BUFFERLEN .DS 1  ;reserve a single byte
          BUFFER    .DS 256  ;reserve 256 bytes
```

chapter 4 Table of Contents

Section 4.7 (directive: .ELSE)

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

chapter 4 Table of Contents

Section 4.8 (directive: .END)

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,

chapter 4 Table of Contents

Section 4.9 (directive: .ENDIF)

purpose: terminate a conditional assembly block

SEE description of .IF for usage and details.

chapter 4 Table of Contents

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!"

chapter 4 Table of Contents

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, in 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.

chapter 4 Table of Contents

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 # <ADDRESS ; not be assembled
 13 .IF 1
 14 .ERROR   "can't get here"
 15   ;likewise, this can't be assembled because it
 16   ;is "nested" within the .IF 0 structure
 17   ;
 18 .ELSE
 19   ;
 20 LDX # <ADDRESS ; these lines will
 21 LDA # >ADDRESS ; be assembled
 22 .ENDIF
 23 JSR PRINTSTRING ; go print the string
```

Note: The assembler resets the conditional stack at the beginning of each pass. Missing .ENDIF(s) will NOT be flagged.

chapter 4 Table of Contents

Section 4.13 (directive: .INCLUDE)

purpose: allows one assembly language program to request that
another program be included and assembled in-line

usage: .INCLUDE #filespec

usage note: this directive should NOT have a label

The .INCLUDE directive causes the assembler to begin reading source lines from the specified "filespec". When the end of "filespec" is reached, the assembler will resume reading source from the previous file (or memory).

CAUTION: The .INCLUDEd file MUST be a properly SAVEd MAC/65 tokenized program. It can NOT be an ASCII file.

Note: A .INCLUDED file cannot itself contain a .INCLUDE directive.

EXAMPLE .INCLUDE #D:SYSEQU.M65

This example line will include the system equates file supplied by OSS.

chapter 4 Table of Contents

Section 4.14 (directive: .LOCAL)

purpose: delimits a local label region

usage: .LOCAL

usage note: this directive should not be associated with a label.

This directive serves to end the previous local region and begin a new local region. It is assumed that the first local region begins at the beginning of the assembly, and the last local region ends at the end of the assembly.

Within each local region, any label beginning with a question mark ("?") is assumed to be a "local label". As such, it is invisible to code, equates, references, etc., outside of its own local region.

This feature is especially handy when using automatic code generators or when several people are working on a single project. In both these cases, the coder may use labels beginning with "7" and be sure that there will be no duplicate label errors produced.

EXAMPLE:

```
10 *= $4000
11 LDX #3     ; establish a counter
12 ?LOOP
13 LDA FROM,X ; get a byte
14 STA TO,X   ; put a byte
15 DEX        ; more to do?
16 BPL ?LOOP  ; goes to label on line 12
17 ;
18 .LOCAL     ; another local region!
```

```
19 ;
20 ?LOOP=6
21 ;
22 LDY #?LOOP ; same as LDY #6
23 (etc.)
```

FEATURE: Local labels MAY be forward referenced, just like any other label.
NOTE: Local labels do not appear in the symbol table listing.

chapter 4 Table of Contents

Section 4.15 (directive: .OPT)

purpose: selects various assembly control OPTions

usage: .OPT option
(or)
.OPT NO option

usage: .OPT option [,[NO] option...]
.OPT NO option [,[NO]option...]

usage notes: the valid options are as follows:
LIST ERR EJECT OBJ
MLIST CLIST NUM XREF

The .OPT directive allows the user to control certain functions of the assembly. Generally, coding ".OPT option" will invoke a feature or option, while ".OPT NO option" will "turn off" that same feature.

You may use any number of options (or NO options) on a single source line. The following line is therefore legal.

Example: .OPT NO LIST,NO XREF, OBJ,ERR

The following are the descriptions of the individual options:

LIST control the entire assembly listing.
NO LIST turns off all listing except error lines.

ERR will determine if errors are returned to the user in the listing and/or the screen.
NO ERR is thus dangerous.

EJECT controls the title and page listing.
NO EJECT only turns off the automatic page generation; it has no effect on .PAGE requests.

OBJ determines if the object code is written to the device/memory.
NO OBJ is useful during trial assemblies. OBJ is NECESSARY when the object code is to placed in memory.

NUM will auto number the assembly listing instead of using the user line numbers. NUM will begin at 100 and increment by 1. NUM is generally not useful except for final, "pretty" assemblies.

MLIST controls the listing of Macro expansions.
NO MLIST will list only the lines within a Macro expansion which generate object code. MLIST will expand the entire Macro. NO MLIST is extraordinarly useful in producing readable listings.

CLIST controls the listing of conditional assembly.
NO CLIST will not list source lines which are not assembled. CLIST will list all lines within the conditional construct.

XRef allows the user, when cross reference has been specified in the ASM comland line, to control which portions of the source program will be cross referenced during the assembly.
Any lines of source code between an .OPT NO XREF and the next succeeding .OPT XREF will not be cross-referenced. By combining NO XREF and NO LIST, you can list and cross reference even extremely large programs in pieces. Or you might use NO XREF to avoid indexing entries out of an INCLUDEd file. XREF and NO XREF are useless and inoperative (but do not generate errors) if you have not specified a cross-reference file name in the ASM command line.

NOTE: unless specified otherwise by the user, all of the options will assume their default settings. The default settings for ..OPT are:

ERR errors are reported
EJECT pages are numbered and ejected
NO NUM use programmer's line numbers
MLIST all macro lines are listed
CLIST all failed conditionals list
LIST listing IS produced
XREF continuous cross reference
NO OBJ l SEE CAUTION !!!!!

CAUTION: The OBJ option is handled in a special way:
IF assembling to memory the object default is NO OBJ.
IF assembling to a device the object option is OBJ.

NOTE: Macro expansions with the NO NUM option will not be listed with line numbers.

chapter 4 Table of Contents

Section 4.16 (directive: .PAGE)

purpose: provides page headings and/or moves
to top of next page of listing

usage: .PAGE [string]

usage note: no label should be used with .PAGE

The .PAGE directive allows the user to specify a page heading. The page heading will be printed below the page number and title heading.

.PAGE will eject the next page, and prints the most recent title and page headings.

Example: 300 .PAGE "EXECUTE LABEL SEARCH"

Note: The assembler will automatically eject and print the current title and page headings after 61 lines have been listed.

chapter 4 Table of Contents

Section 4.17 (directive: .SBYTE)

purpose: produces "screen" bytes in output object

usage: See .BYTE description, section 4.4

chapter 4 Table of Contents

Section 4.18 (directive: .SET)

purpose: controls various assembler functions

usage: .SET dcnum1 , dcnum2

The .SET directive allows the user to change specific variable parameters of the assembler. The dcnum1 specifies the parameter to change, and dcnum2 is the changed value. The following table summarizes the various .SET parameters. Defaults for each parameter are given in parentheses, followed by the allowable range of values.

dcnum1 dcnum2 function

0 (4) 1-4 sets the .BYTE and .SBYTE listing format. 1 to 4 bytes can be printed in the object code filed of the listing.

1 (0) 0-31 sets the assembly listing left margin. The specified number is the number of spaces which will be printed before the assembled source line.

2 (80) 40-132 set width for listing, adjust for your printer.

3 (12) 0,12 form feed select. 0 implies no form feed on printer--use multiple line feeds. Any other used as form feed char.

4 (66) any number of lines per page for listing.

5 (0) 0-255 number of spaces from semi-colon in comment field to where remainder of comment is printed.

6 (0) 0-$FFFF an offset, which is added to the location counter when an object byte is stored or written to disk. You can thus assemble code to one address but it is written to run at another address. See Chapter 8 for a complete discussion of .SET 6 capabilities

Section 4.19 (directive: .TAB)

purpose: sets listing "tab stops" for readability

usage: .TAB dcnum1 .dcnum2 .dcnum3

The .TAB directive allows the user to specify the starting column for the listing of the instruction field, the operand field, and the comment filed respectively. The defaults are 8,12,20.

Example: 200 .TAB 16,32,50

```
...
1200  .TAB 8,12,20 ;restores defaults
```

chapter 4 Table of Contents

Section 4.20 (directive: .TITLE)

purpose: specify assembly listing heading

usage: .TITLE string

The .TITLE directive allows the user to specify a assembly title heading. The title string will be printed at the top of every page following the page number.

chapter 4 Table of Contents

Section 4.21 (directive: .WORD)

(See also .DBYTE section 4.6)

purpose: place 16 bit word values in output object

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

The .WORD and .DBYTE directives both put the value of each following expression into the object code as two bytes. But where .WORD will assemble the expression(s) in 6502 address order (least significant byte, most significant byte). .DBYTE will assemble the expression(s) in reverse order (most significant byte, least significant byte).

Generally, for 6502 programs, .WORD is the more useful of the two, and is more compatible with the code produced by assembled 6502 instructions.

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

chapter 4 Table of Contents

CHAPTER 5: MACRO FACILITY

A MACRO DEFINITION is a series of source lines grouped together, given a name, and stored in memory. When the assembler encounters the corresponding name in the instruction (opcode, directive) column, the saved lines will be substituted for the Macro name and assembled. Effectively, this allows the user to define and then use new assembler instructions. Depending upon the code stored in its definition, a macro might be thought of as either an "extra" directive or a "new" opcode.

The process of finding of a macro in the table when its name is used, and then assembling the code it was defined with, is called a MACRO EXPANSION. The unique facility of Macro Expansions is that they may have PARAMETERS passed to them. These parameters will be substituted for the "formal parameters" during the expansion of the Macro.

The use (expansion) of a Macro in a program required that the Macro first be defined. To the set of directives already discussed in chapter 4 then, must be added two new directives used for defining new macros:

```
.MACRO
.ENDM
```

This chapter will first discuss these two directives, show how to invoke a macro (cause its expansion) and then examine the use of formal and calling parameters, including string parameters.

chapter 4 Table of Contents

Section 5.1 (directive: .ENDM)

purpose: end the definition of a macro

usage: .ENDM

usage note: generally, the .ENDM directive should not be labelled.

This directive is used solely to terminate the definition of a macro. When invoking a macro, do NOT use this directive. Basically, the concept of macros requires that all source lines between the .MACRO directive and the .ENDM directive be stored in a special section of memory (the macro table). Thus, encountering an improperly paired .ENDM directive is considered a severe assembly error. See the description of .MACRO (section 5.2) for further information.

chapter 5 Table of Contents

Section 5.2 (directive: .MACRO)

purpose: initiates a macro definition

usage: .MACRO macroname

usage note: "macroname" may be any valid MAC/65 label. It MAY be the same name as a program label (without conflict).

The .MACRO directive all cause the lines following to be read and stored under the Macro name of "macroname". The definition is terminated with the .ENDM directive.

All instructions except another .MACRO directive are valid Macro source lines. A Macro definition can NOT contain another Macro definition.

A simple example of a MACRO DEFINITION:

```
10  .MACRO PUSHXY ; The name of this Macro is "PUSHXY"
11     ; When this Macro is used (expanded), the following
12     ; instructions will be substituted for "PUSHXY"
13     ; and then assembled.
14   TXA
15   PHA
16   TYA
18   PHA
19  .ENDM   ; The terminator for "PUSHXY"
```

SPECIAL NOTE: ALL labels used within a macro are assumed to be local to that macro. MAC/65 accomplishes this by performing a "third pass" of the assembly during macro expansions. Thus, a label defined within a macro expansion is ava liable to code which follows the macro; but another expansion of the same macro with the same label will reset the labels value. The action is similar to the ".=" directive, except that forward references to internal macro labels ARE legal.

EXAMPLE:

```
20 .MACRO MOVE6
21 LDX #5
22 LOOP
23 LDA FROM,X
24 STA TO,X
25 DEX
26 BPL LOOP
27 .ENDM
```

The label "LOOP" is local to this macro usage, and yet it may (if needed) be referenced outside the macro expansion (although not in another macro expansion). (note that if a macro label is only defined once by a single macro usage, the effect is the same as if the label were defined outside any macro.) Although the .LOCAL-produced local regions may be used by and with macros, the user is limited to a maximum of 62 local regions. No such restriction applies to the number of possible local usages of a label in a macro expansion.

chapter 5 Table of Contents

Section 5.3 (Macro Expansion, Part 1)

As stated above, a macro is expanded when it is used. And the "use" of a macro is simplicity itself.

To invoke (use, expand--all equivalent words) a macro, simply place its name in the opcode/directive field of an assembler line. Remember, though, that macros MUST be defined before they can be used.

For example, to invoke the two macros defined in examples in the previous section 5.2 , one could simple type them in as shown and then enter and assemble:

EXAMPLE:

```
2000 ALABEL PUSHXY
```

```
2010 ; and pushxy generates the code
2020 ;   TXA   PHA   TYA   PHA
2030 ;
2040 MOVE6
2050 ; similarly, MOVE6 is used
2060 JMP LOOP
2070 ; and LOOP refers to the label
2080 ; defined in the MOVE6 macro
...
```

Note that the use of a label on the macro invocation is optional. The label is assigned the current value of the location counter and is not dependent upon the contents of the macro at all.

There are many more "tricks" and features usable with macros, but we will continue this discussion after an examination of macro parameters as used in a macro definition.

chapter 5 Table of Contents

Section 5.4 (Macro Parameters)

Macro parameters can be of two types: expressions (which are evaluated as 16 bit words) or strings. The parameters are passed via the macro expansion (invocation, use, etc.) and are stacked in memory in the order of occurance. A maximum of 63 parameters can be stacked by a macro expansion, including expansions within expansions.

However, before a parameter can be used in an expansion, there must be a way of accessing it in the MACRO DEFINITION. Parameters are referenced in a macro definition by the character "%" for expressions and the characters "%$" for strings. The value following the character refers to the actual parameter number.

SPECIAL NOTE: The parameter number can be represented by a decimal number (e.g., %2) or may be a label enclosed by parentheses (e.g., %$(LABEL)). Of course, strings may be similarly referenced, as in %$(INDEX) or %$1.

Examples:

```
10 LDA #>%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 # <%2 ; low byte of the increment
 29  STA %1     ; low byte of result
 30  LDA %1 +1 ; high byte of location
 31  ADC # >%2 ; add in high byte of increment
 32  STA %1 +1 ; and store rest of result
```

```
33 ;
34 .ELSE
35 ; this is assembled if only one parameter given
36  INC %1  ; just increment by 1.
37  BNE SKIPHI ; implicitly local label
38  INC %1 +1 ; must also increment high byte
39 SKIPHI
40 .ENDIF   ; matches the .IF %0>1 (line 24)
41 .ENDIF   ; matches the .IF of line 18
42 .ENDM    ; terminator.
```

chapter 5 Table of Contents

Section 5.5 (Macro Expansion, Part 2)

We have shown how macro definitions may include specifications of particular parameters (the specifications might also be called "formal parameters"). This section will show how to pass actual parameters (equivalently "value parameters", "calling parameters", etc.) to the definition.

The concept is simple: on the same line as the macro invocation (by use of its name, of course) and following the macro's name, the user may place expressions (or strings, see section 5.6). MAC/65 simply assigns each of these values a number, from 1 to 63, and then, during the macro expansion, replaces the formal parameters (%1, %2, %(label), etc.) with the corresponding values.

Does that sound too complicated? Internally, it is. Externally, it is an easy as this:

EXAMPLE:

Assume that the BUMP macro has been defined (as above, section 5.4), then the user may invoke it as needed, thus:

```
100 ALABEL BUMP A.LOCATION
110 INCR .= 7
120  BUMP A.LOCATION,3
130  BUMP A.LOCATION-2
140  BUMP
150  BUMP A.LOCATION,INCR,7
160 A.LOCATION .WORD 0
```

note: lines 140 and 150 will each cause the BUMP error to be invoked and printed

```
170  BUMP INCR,A.LOCATION
  will try to increment address 7 by something
180  BUMP PORT5
  assuming the PORT5 is some hardware port,
  strange and wonderful things could happen
```

chapter 5 Table of Contents

Section 5.6 (Macro Strings)

String parameters are represented in a macro definition by the character "%$". All numeric parameters have a string counterpart, not all of which are useful. All string parameters have a numeric counterpart (their length).

As a special case, %$0 always returns the macro NAME]

The following table shows the various string and numeric values returned for a given parameter:

As appears in Macro call:	string returned (in quotes):	numeric value returned:
"A String 1 2 3"	"A String 1 2 3"	length of string
NUMERICSYMBOL	"NUMERICSYMBOL"	value of label
SYMBOL+1	"SYMBOL"	value of expr
%$4	the string of parameter 4 value of original (above would be used by a macro calling another macro)	
-LABEL	"LABEL"	value of expr
GEORGE*HARRY+PETE	undefined	value of expr
.DEF CIO	"CIO"	value of expr
2 + 2 * 65	undefined	value of expr

A Macro string example:

```
10  .MACRO PRINT
11 ;
12 ; This Macro will print the specified string.
13 ; parameter 1, but if no parameter string is
14 ; passed, only an EOL will be printed.
15 ;
```

```
16 ; The calling format is: PRINT [ string ]
17 ;
18 .IF %0 = 1 ; is there a string to print?
19  JMP PASTSTR ; yes, jump over string storage
20 STRING .BYTE %$1,EOL ; put string here.
21 ;
22 PASTSTR
23  LDX #>STRING ; get string address into X&Y
24  LDY #<string ; for JSR to 'print string'
25  JSR STRINGOUT
26 .ELSE
27 ; no string...just print an EOL
28  LDA #EOL
29  JSR CHAROUT
30 ;
31 .ENDIF
32 .ENDM   ; terminator.
```

To invoke this macro, then, the following calls would be appropriate:

```
100 PRINT "this is a string"
110 PRINT
120 PRINT message
...
999 message .BYTE "another string", EOL
```

note that, in line 120, only a single word (label, actually) is allowed.

chapter 5 Table of Contents

Section 5.7 (Some Macro Hints)

Each person will soon develop his/her own style of writing macros, but these are certain common sense rules that we all should heed.

A. When a macro is defined, its entire definition must be stored in memory (in a macro table). Since memory space is obviously finite, it is a good idea to keep macros as short as possible. One way to do this is to avoid putting comments (remarks) within the body of the macro. If you do document your macros (and we hope you do), place the comments in the file BEFORE the .MACRO directive. The assembler will then do nothing at all with them and they will occupy no additional space.

B. Don't use a caller's macro parameter unless you are sure that it is there. Using a parameter that the caller left out will produce a MACRO PARAMETER error.

Depending upon the macro definition, this may or may not also produce undesired results. An example of unsafe coding:

```
.If %0>1 .OR %2=0
.WORD %1
.ENDIF
```

The danger here occurs if the caller invokes the macro with only one parameter. Since %2 is non-existent (and hence undefined), the sub-expression "%2=0" is indeed true and the effect of "%0>1" is nullified. Of course, the lack of parameter 2 will produce a "PARAMETER ERROR", but it will already be too late. A better coding of the above would be:

```
.IF %0>1
 .IF %2<>0
  .WORD %1
 .ENDIF
.ENDIF
```

C. Even though labels defined within macros are local to each invocation, they are still "visible" outside the macro(s). Thus, it might be a good idea to have a special form for labels defined in macros and avoid that form outside macros. The macro library supplied with MAC/65 uses labels beginning with "@" as local labels to macros.

chapter 5 Table of Contents

Section 5.8 (A Complex Macro Example)

The following set of macros is designed to demonstrate several of the points made in the preceding sections. Aside from that, though, it is a good, usable macro set. Study it carefully, please. (The line numbers are omitted for the sake of brevity. Any numbers will do, of course.)

```
;
; the first macro, "@CH", is designed to load an
; IOCB pointer into the X register. If passed a
; value from 0 to 7, it assumes it to be a constant
; (immediate) channel number. If passed any other
; value, it assumes it to be a memory location which
; contains the channel number.
;
; NOTE that these comments are outside the body of
; the macro, thus saving valuable table space.
```

```
;
 .MACRO @CH
 .IF %1>7
 LDA %1 ; channel # is in memory cell
 ASL A
 ASL A
 ASL A
 ASL A   ; times 16
 TAX
 .ELSE
 LDX #%*16
 .ENDIF
 .ENDM
;
; this next macro, "@CV", is designed to load a
; Constant or Value into the A register. If
; passed a value from 0 to 255, it assumes it
; to be a constant (immediate) value. If passed
; any other value, it assumes it to be a memory
; location (non-zero page).
;
 .MACRO @CV
 .IF %1<256
 LDA #%1
 .ELSE
 LDA %1
 .ENDIF
 .ENDM
;
; The third macro is "@FL", designed to establish
; a filespec. If passed a literal string, @FL
; will generate the string in line, jumping around
; it, and place its address in the IOCB pointed to
; by the X register. If passed a non-zero page
; label, @FL assumes it to be the label of a valid
; filespec string and uses it instead.
;
 .MACRO @FL
 .IF %1<256
 JMP *+%1+4
@F .BYTE %$1,0
 LDA #<@F
 STA ICBADR+1,X
 LDA #>@F
 STA ICBADR+1,X
 .ELSE
```

```
 LDA #<%1
 STA ICBADR,X
 LDA #>%1
 STA ICBADR+1,X
 .ENDIF
 .ENDM

;
; The main macro here is "XIO", a macro to
; implement a simulation of BASIC's XIO command.
; The general syntax of the usage of this macro is:
;  XIO command, channel [,aux1,aux2] [,filespec]
;
; where channel may be a constant from 0 to 7
;   or a memory location.
; where command, aux1, and aux2 may be a constant
;   from 0 to 255 or a non-zero page location
; where filespec may be a literal string or
;   a non-zero page location
; if aux1 and aux2 are omitted, they are assumed
;   to be zero (you may not omit aux2 only)
; if the filespec is omitted, it is assumed to
;   be "S:"
;
 .MACRO XIO
 .IF %0<2 .OR %0>5
 .ERROR "XIO: wrong number of parameters"
 .ELSE
 @CH %2
 @CV %1
 STA ICCOM,X ; command
 .IF %0>=4
 @CV %3
 STA ICAUX1,X
 @CV %4
 STA ICAUX2,X
 .ELSE
 LDA #0
 STA ICAUX1,X
 STA ICAUX2,X
 .ENDIF
 .IF %0=2 .OR %0=4
 @FL "S:"
 .ELSE
@FPTR .= %0
 @FL %$(@FPTR)
```

```
.ENDIF
JSR CIO
.ENDIF
.ENDM
```

Did you follow all that? The trick is that, the way "XIO" is specified, it is legal to pass it 2, 3, 4, or 5 arguments; but each of those numbers represents a unique combination of parameters, to wit:

```
XIO  command,channel
XIO  command,channel,filespec
XIO  command,channel,aux1,aux2
XIO  command,channel,aux1,aux2,filespec
```

This is not a trivial macro example. Perhaps you will not have occasion to write something to complex. But MAC/65 provides the tools to do many things if you need them.

chapter 5 Table of Contents

CHAPTER 6: COMPATIBILITY

There are many different 6502 assemblers available, and it seems that each has a few foibles, bug, or whatever that are uniquely its own (and, or course, they are called "features" by their promoters). Well, MAC/65 is no different.

This chapter is devoted to telling you of some of the things to watch out for when converting from another 6502 assembler to MAC/65. We restrict ourselves to such things as directives and operators. We will NOT go into a discussion of how to convert the actual 6502 opcodes (equivalently: instructions, mnemonics, etc.). We consider it mandatory that any good 6502 assembler will follow the MOS Technology standard in this regard.

Example: We know of some antique 6502 assemblers that specify the various addressing modes via special opcodes. Thus the conventional "LDA #3" becomes "LDAIMN 3" and "LDA (ZIP),Y" becomes "LDAIY ZIP". Unfortunately, there was never any standard established for such distortions, so we shall ignore them as antique and outmoded. In any case, unless you are entering a program out of an older magazine, you are unlikely to run into one of these strange beasts.

The rest of this chapter pays homage to our birthright. MAC/65 is a direct descendant of the Atari assembler/editor cartridge (via EASMD). As much as possible, we have tried to keep MAC/65 compatible with the cartridge. Unfortunately, in the interest of

providing a more powerful tool, a few things had to be enumerates these changes.

chapter 5 Table of Contents

Section 6.1 (ATARI'S ASSEMBLER/EDITOR CARTRIDGE)

This section presents all known functional differences between the Atari cartridge and MAC/65. Obviously, MAC/65 also has many more features not enumerated here, but they will not impact the transferrance of code originally designed for the cartridge (or, for that matter, EASMD).

chapter 5 Table of Contents

Section 6.1.1 (.OPT OBJ / NOOBJ)

By default, the Atari cartridge produces object code, even when the destination of the object is RAM memory. This is a dangerous practice, at best: it is too easy to make a mistake in a program and write over DOS, the user's source, the screen memory, or even (horror of horrors) some of the hardware registers.

MAC/65 makes a special case of object in memory: you don't get it unless you ask for it. You MUST have a ".OPT OBJ" directive before the code to be generated or the code will not be produced.

section 6.1 Table of Contents

Section 6.1.2 (OPERATOR PRECEDENCE)

The cartridge assigns no precedence to arithmetic operators. MAC/65 uses a precedence similar to BASIC's. Most of the time, this causes no problems; but watch out for mixed expressions.

Example: LDA #LABEL-3/256
seen as LDA #[LABEL-3] / 256 by the cartridge
seen as LDA #LABEL - [3/256] by MAC/65

section 6.1 Table of Contents

Section 6.1.3 (THE .IF DIRECTIVE)

The implementation of .IF in the cartridge is clumsy and unusable. MAC/65's implementation is more conventional and much more powerful. Rather than try to offer a long example here, we will simply refer you to the appropriate sections of the two manuals.

section 6.1 Table of Contents

CHAPTER 7: ADDED 65C02 INSTRUCTIONS

MAC/65 as originally produced, supported the "standard" 6502 instruction set as well as the directives and addressing mode designators recommended by MOS Technology (the originators of the 6502 chip).
This version of MAC/65 supports all features of the original version along with added support for one of the more popular enhanced versions of the 6502 chip. In particular, MAC/65 supports all new instructions and addressing modes available on the 65C02 chip as produced by NCR Corporation. We describe here the primary added addressing mode, the instruction with variants added, and the completely new instructions. But before we start, we should note that these instructions would only work properly on your computer if you have installed an NCR 65C02 in place of the 6502 which came in the machine as purchase. Also, remember that a program using these instructions may work great in your machine. It will not work properly in your friend's machine unless he/she also installs a 65C02.

chapter 6 Table of Contents

Section 7.1 (A Major Added Addressing Mode)

The standard 6502 chip supports two forms of indirect addressing for what might be considered its primary instructions. The forms appear in assembly listings as:

```
 lda (indirect,X)
   and
 lda (indirect,Y)
```

(where "lda" is only one of several valid mnemonics that can be used with these addressing modes).
The latter of these modes, often referred to as the "indirect-Y" mode, is perhaps the most useful and flexible of all 6502 addressing modes. And, yet, it suffers from one flaw: it ties up two registers (A and Y). And, as importantly, probably a full 50% or more of the time the Y-register is loaded with zero before instructions in this mode are executed.

The NCR 6502 instructions set as supported by MAC/65 provides a help here: You may code instructions allowing Indirect-Y addressing in "Indirect" mode as well. With Indirect mode, the assembler format is simply

```
 lda (indirect)
```

where, as with Indirect-Y, the indirect location must be in zero page.
Generally, the effect of using this instruction will be the same as coding the sequence:

```
 LDY #0
 lda (indirect),Y
```

EXCEPTING that the Y-register remains intact and untouched and may be used for other purposes. The following, then, are ALL of the 65C02 instructions which allow and support this new addressing mode:

```
 ADC (indirect)  ;ADd with Carry
 AND (indirect)  ;bit wise AND
 CMP (indirect)  ;compare with A-reg
 EOR (indirect)  ;Exclusive OR
 LDA (indirect)  ;LoaD the A-register
 ORA (indirect)  ;inclusive OR
 SBC (indirect)  ;SuBtract with Carry
 STA (indirect)  ;STore the A-register
```

REMINDER: while the "indirect" location may be any zero page location, you should probably restrict yourself to the available locations documented in the DDT manual.

chapter 7 Table of Contents

Section 7.2 (Minor Variations on 6502 Instructions)

The "BIT" instruction has added two new addressing modes, and "JMP" has added one new mode. They are described here individually:

Original allowed forms of 6502 BIT instructions were:
 BIT absolute
 BIT zeropage
New 65C02 forms available are:
 BIT absolute,X
 BIT zeropage,X
The ability to use the X register as an index with the BIT instruction greatly enhances its power for testing tables, etc. The "indexed-x" address modes function as they do for other 6502 instructions (eg LDA and CMP).

Original allowed forms of 6502 JMP instructions were:
 JMP absolute
 JMP (indirect)
New 65C02 form available is:
 JMP (indirect,X)

Note that the JMP instruction alone in both the 6502 and 65C02 instructions sets uses an absolute (ie 16 bit, 2 byte) address for its indirect value. The new "indirect-X" form is no different: the location specified as the indirect address may be anywhere in memory. The "indirect-X" address mode is unique and new. Its effect is as follows: add the contents of the X-register to the ADDRESS (not the contents) specified by the given indirect address; use the result as the address of the true operand for this instruction; JuMP to the address contained in the word-sized location accessed via the true operand.

Example: .WORD SUB1,SUB2,SUB3
```
 ...
 LDA value     ;assume that "value"
               ;contains 0, 1, or 2
 ASL A         ;double the value
 TAX           ;... to X-register
 JMP (TABLE,X) ;and go to SUB1,SUB2
               ;SUB3 depending of "value"
```

chapter 7 Table of Contents

Section 7.3 (ALL-NEW 65C02 Instructions)

We detail here, in what we hope are logical groupings, the 65C02 instructions which are truly "new" to the 6502 world.

chapter 7 Table of Contents

Section 7.3.1 (BRA)

Mnemonic: BRA
Read as: BRanch (Always)
Format: BRA addr
where addr must be in the range *-126 to *+129
(* is the current value of the location counter)

BRA joins the Branch family (BNE, BEQ, BMI, etc) and adds the powerful capability of ALWAYS branching. It thus becomes equivalent to a JMP instruction with the advantage that it occupies one less byte in memory and is inherently relocatable. Its address range is restricted in a fashion identical with the other members of the "branch" family.

Section 7.3 Table of Contents

Section 7.3.2 (DEA and INA)

Mnemonic:DEA
INA
Read as: DEcrement Accumulator
Increment Accumulator
Format: DEA
INA

These simple instructions add a capability long lacking in the 6502. Until now, if you wished to change the contents of the accumulator by one, you had to either use TAX/INX/TXA (or something similar) or CLC/ADC (or SEC/SBC), three byte substitutes for what should (and now is) a simple byte instruction. Processor status flags (ie N and Z), timing, etc, are all identical to the very similar INX/INY/DEX/DEY set of instructions.

Section 7.3 Table of Contents

Section 7.3.3 (PHX, PHY, PLX, PLY)

Mnemonic:PHX
PHY
PLX
PLY
Read as: PusH X onto CPU stack
PusH Y onto CPU stack
PulL X from CPU stack
PulL Y from CPU stack
format: PHX
PHY
PLX
PLY

Again, these instructions are provided as short cuts for the cumbersome sequences necessary on the standard 6502. As an example, PHX can replace a sequence of instructions as complex as this:

```
STA temp
TXA
PHA
LDA temp
```

By giving you direct access to the stack from the X and Y registers, it is possible and desirable to write more compact and more relocatable code. Processor status flag usage, timings, etc, are identical to the very similar PHA and PLA instructions.

Section 7.3 Table of Contents

Section 7.3.4 (STZ)

Mnemonic:STZ
Read As: STore Zero
Format: STZ absolute
STZ absolute,X
STZ zeropage
STZ zeropage,X

Yet another short cut, STZ simply replaces the sequence

```
LDA #0
STA address
```

with the difference that it does not affect the contents of the A register. In fact, to properly simulate this instruction on an ordinary 6502, the following code would be needed in the general case: PHA
 LDA #0
 STA address
 PLA

Section 7.3 Table of Contents

Section 7.3.5 (TRB and TSB)

Mnemonic:TRB
TSB
Read As: Test and Rest Bits
Test and Set Bits

Format: TRB absolute
TRB zeropage
TSB absolute
TSB zeropage

These instructions have many uses, not the least of which would be synchronization of background and foreground (interrupt-driven) routines. In Boolean terms, the instructions might be thought of thus: TRB: Memory := (Not A) and Memory
 TSB: Memory := A or Memory
In words, we might describe the operation of these instructions as follows:
For TRB: The complement of the contents of the Accumulator is bit-wise- AND-ed with the contents of the memory cell addressed by this instruction (either and absolute or zero-page location). The result of this AND-ing is placed back in the addressed memory cell.
For TSB: The contents of the Accumulator are bit-wise OR-ed with the contents of the memory cell addressed by this instruction. The result of this OR-ing is placed back in the addressed memory cell. If the result of the AND-ing or OR-ing is zero, the Zero processor status flag is set. The N and V flags are set to the contents of the bits 6 and 7 (similar to the usage and results of the BIT instruction) of the addressed memory cell as those contents were BEFORE the bit-wise operation took place.
Example: FLAG .BYTE 3
TEST .BYTE $FF
...
LDA #$FF
TRB FLAG ;resets all bits!
...
LDA #0
TSB TEST ;just tests value

CHAPTER 8: PROGRAMMING TECHNIQUES WITH MAC/65

Section 8.1 (Memory Usage by MAC/65 and DDT)

The following memory locations are used by MAC/65 and/or DDT for the purposes shown:

range of addresses	used by MAC/65	DDT	used for
$80-$AF	yes	yes	pointers and temporaries
$B0-$D3	yes	no	pointers and temporaries
$D4-$FF	yes	yes	floating point registers, etc
$100-$1FF	yes	yes	normal 6502 CPU stack
$3FD-$47F	no	yes	buffers and display area
$480-$57F	yes	yes	buffers and work area
$580-$67F	yes	no	input buffers, etc
"size"	yes	*	programs text, etc

Note that "size" refers to the memory area delineated by the lowest and middle number displayed when the "SIZE" command is used form the MAC/65 editor. The * in DDT column indicates that DDT saves MAC/65's zero page memory (and other, related, locations) in the area actually shown to be part of the "size" memory.
The worst implication of the memory map above (especially for Atari BASIC users) is that page 6 is NOT completely available to you. Since many magazine articles assume that page 6 is available, they will not run AS IS under MAC/65 and DDT. But see the next section for methods to use if you MUST use page 6.

Section 8.2 (Assembling With An Offset: .SET 6)

In Section 4.18 , we noted that the assembler directive ".SET 6,value" could be used to specify an additive offset for the storage address vis-a-vis the location counter address. In this section, we present a method for using this capability in a practical sense. Let us assume that we wish to assemble a small program which will reside in

page 6 ($600 thru $6FF). The program which we will assemble is presented here:

```
 10   *= $600
 20 COLOR4 = $2C8
 30 ;
 40 START
 50   PLA      ;remove count of parameters
 60   CMP #0   ;any parameters?
 70   BEQ *    ;if yes, loop forever
 80   LDA COLOR4 ;get current background color
 90   CLC
 100  ADC #$10 ;change to next hue
 110  STA COLOR4 ;...by changing shadow register
 120  INC COUNT ;and count the number of times
 130  RTS
 140 COUNT .BYTE 0 ;just a simple counter
 150  .END
```

If you assemble this routine, you should get an error free assembly. And those of you who are BASIC users will recognize this as a routine callable from Atari BASIC, thanks to the PLA and check on number of parameters at the beginning. But it is designed to reside in page 6. What can we do? Answer: simply add the following two lines to the listing:

```
 12 .OPT OBJ     ;we do want object code
 14 .SET 6,$3000 ;and we will offset
```

Now, if you assemble this code, you will notice that the addresses shown start at $3600. And, indeed, the assembly is placing the code in memory at the addresses shown. But look at line 120. Notice that the object code generated does NOT show that location $3612 is being incremented! Instead, location $0612 is affected. Also note that in the symbol table listing START is shown to be at location $0600 and COUNT at $0612. Now use the "DDT" command to enter DDT. From DDT, enter the command

```
 M 360006000080 <RETURN>
```

which will move $80 (128) bytes from location $3600 to location $0600. Use the command

```
 * 0600 <RETURN>
```

to view the contents of locations $0600 and beyond. Use the up and down arrows (remember, WITHOUT pushing CTRL) to view the code. Lo and behold, your code has been successfully deposited where you wanted it, waiting for you to debug. Some final notes on this subject: MAC/65 will generate this "offset" kind of code either directly to memory (as we did here) or to an object file (on disk, presumably). When the file is reloaded (via MAC/65's BLOAD command or via some load command from the DOS you are using), it will be loaded at the address shown in the listing. It is your responsibility to then somehow move it to the desired location. The technique is not

necessarily easy, but using these methods you can overwrite DOS or even produce code designed to run in the cartridge space. In the latter case, you may wish to use a negative offset with .SET 6. This is perfectly legal and reasonable.

chapter 8 Table of Contents

Section 8.3 (Making MAC/65 Even Faster)

If you .INCLUDE a file consisting ONLY of equates and/or macro definitions (NOT macro calls!), there is a technique you can use which will speed up assembly somewhat. In particular, since equates need be made only once and macros need be only defined once, there is no reason to read such .INCLUDED files on pass two. The following code shows a workable technique:

```
  *= 0
PASS .= PASS+1   ;do this only once per assembly
  .IF PASS=1
  .INCLUDE #D:equatesfile
  .ENDIF
  *= beginning
```

Why this works: Normally, an undefined label has a value of zero. The ".=" directive, however, causes a mildly strange thing to happen: an undefined label used on the right side of ".=" takes on the current value of the location counter. Hence the need for the "*= 0" line at the beginning of the above example. In any case, thanks to this mechanism, the first time the second line is assembled (in pass 1); PASS takes on a value of 1 (of course, the line also generates an "undefined label" error, but such errors are not printed in pass 1). The next time it is assembled, PASS receives a value of 2. Simple and neat. Note that if the ".=" used in the second line above is placed ahead of any "*=" (".ORG") lines, then the first line shown is not needed, since the location counter is assumed to start at zero unless told otherwise.

chapter 8 Table of Contents

CHAPTER 9: ERROR DESCRIPTIONS

When an error occurs, the system will print

*** ERROR -

followed by the error number (unless the error was generated with the .ERROR assembler directive) and, for most errors, a descriptive message about the error.

Note: The assembler will print up to 3 errors per line.

The format used in the listing of descriptions which follows is simply ERROR NUMBER, ERROR MESSAGE, description and possible causes.

1 - MEMORY FULL

All user memory has been used. If issued by the Editor, no more source lines can be entered. If issued by the Assembler, no more labels or macros can be defined.
NOTE: If memory full occurs during assembly and the source code is located in memory, SAVE the source to disk, type NEW, and assemble from the disk instead. Now the assembler can use all of the space formerly occupied by your source for macro and symbol tables, etc.

2 - INVALID DELETE

Either the first line number is not present in memory, or the second line number is less than the first line number.

3 - BRANCH RANGE

A relative instruction references an address displacement greater than 129 or less than 126 from the current address.

4 - NOT Z-PAGE / IMMEDIATE MODE

An expression for indirect addressing or immediate addressing has resolved to a value greater than 255 ($FF).

5 - UNDEFINED

The Assembler has encountered a undefined label.

6 - EXPRESSION TOO COMPLEX

The Assembler's operator stack has overflowed. If you must use an expression as complex as the one which generated the error, try breaking it down using temporary SET labels (i.e., using ".=").

7 - DUPLICATE LABEL

The Assembler has encountered a label in the label column which has already been defined.

8 - BUFFER OVERFLOW

The Editor syntax buffer has overflowed. Shorten the input line.

9 - CONDITIONAL NESTING

The .IF-.ELSE-.ENDIF construct is not properly nested. Since MAC/65 cannot detect excess .ENDIFs, the problem must be an EXTRA .ELSE or .ENDIF instead.

10 - VALUE > 255

The result of an expression exceeded 255 when only one byte was needed and allowed.

11 - CONDITIONAL STACK

The .IF-.ELSE-.ENDIF nesting has gone past the number allowed. Conditionals may be nested a maximum of 14 levels.

12 - NESTED MACRO DEFINITION

The Assembler encountered a second .MACRO directive before the .ENDM directive. This error will abort assembly.

13 - OUT OF PHASE

The address generated in pass 2 for a label does not match the address generated in pass 1. A common cause of this error are forward referenced addresses. If using conditional assembly (with or without macros), this error can result from a .IF evaluating true during one pass and false during the other.

14 - *= EXPRESSION UNDEFINED

The program counter was forward referenced.

15 - SYNTAX OVERFLOW

The Editor is unable to syntax the source line. Simplify complex expressions or break the line into multiple lines.

16 - DUPLICATE MACRO NAME

An attempt was made to define more than one Macro with the same name. Only the first definition will be valid.

17 - LINE # >65535

The Editor cannot accept line numbers greater than 65535.

18 - MISSING .ENDM

In a Macro definition, an EOF was reached before the corresponding .ENDM terminator. Macro definitions cannot cross file boundaries. This error will abort assembly.

19 - NO ORIGIN

The *= directive is missing from the program.
Note: This error will only occur if the assembler is writing object code.

20 - NUM/REN OVERFLOW

On the REN or NUM command, the line number generated was greater than 65535. If REN issued the error, entering a valid REN will correct the problem. If NUM issued the error, the auto-numbering will be aborted.

21 - NESTED .INCLUDE

An included file cannot itself contain an .INCLUDE directive.

22 - LIST OVERFLOW

The list output buffer has exceeded 255 characters. Use smaller numbers in the .TAB directive.

23 - NOT SAVE FILE

An attempt was made to load or assemble a file not created with the SAVE command.

24 - LOAD TOO BIG

The load file cannot fit into memory.

25 - NOT BINARY SAVE

The file is not in a valid binary (memory image, assembler object, etc.) format.

27 - INVALID .SET

The first dcnum is a .SET specified a non-existant Assembler system parameter.

30 - UNDEFINED MACRO

The Assembler encountered a reference to a Macro which is not defined. Macros must first be defined before they can be expanded.

31 - MACRO NESTING

The maximum level of Macro nesting has exceeded 14 levels.

32 - BAD PARAMETER

In a Macro expansion, a reference was made to a nonexistent parameter, or the parameter number specified was greater than 63.

128 -255 [operating system errors]

Error numbers over 127 are generated in operating system. Refer to the DOSXL manual for detailed descriptions of such errors and their causes.

chapter 8 Table of Contents

APPENDIX A

Actually, the bulk of this appendix is contained on your master MAC/65 diskette in the form of a system macro file. This appendix is here simply to alert you to the existence of the file and to give a brief description of macros available. We would suggest that you use MAC/65 to LOAD and LIST (to a printer or the screen) the file IOMAC.LIB.

May we suggest that you adopt a naming convention for you MAC/65 files, both SAVEd and LISTed, that does not conflict with anything? We use the following extensions (though you are obviously free to rename our files to your own preferences):

.M65 MAC/65 SAVEd files
.ASM MAC/65 LISTed files
.LIB MAC/65 SAVEd libraries

(note that C/65 insists on its runtime library being named RUNTIME.LIB, hence this convention)

In any case, the macros available in IOMAC.LIB are:

OPEN chan,aux1,aux2,filename

Opens the given filename on the given channel using aux1 and aux2 as per DOSXL specifications.

PRINT chan [,buffer[,length]]

If no buffer given, prints just a CR on chan. If no length given, length assumed to be 255 or position of CR, which ever is smaller. Buffer may be literal string, in which case length is ignored if given.

INPUT chan,buffer [,length]

If no length given, defaults to 255 bytes.

BGET chan,buffer,length

Binary read, a la BASIC XL, of length number of bytes into the given buffer address.

BPUT chan,buffer,length

Binary write of length number of bytes from the given buffer address.

CLOSE chan

Closes the given file.

XIO command,chan [,aux1,aux2][,filename]

As described in chapter 5

NOTES:

"chan" may be a literal channel number (0 through 7) or a memory location containing a channel number (0 through 7).

"aux1", "aux2", "length" and "command" may all be either literal numbers (0 to 255) or memory locations.

"filename" may be either a literal string (e.g., "D:FILE1.DAT") or a memory location, the latter assumed to be the address of the start of the filename string.

Where memory locations are given instead of literals, they must be non-zero page locations which are defined BEFORE their usage in the macro(s). The following example will NOT work properly!! :

```
PRINT 3,MESSAGE1 ; WRONG!
...
MESSAGE1 .BYTE "This WON'T WORK !!! "
```

These macros are useful instruments, but they really are meant only as examples, to

show you what you can do with MAC/65. Please feel free to study them and change them as you need.

chapter 8 Top of Page