FIRST AND FINEST

MAC/65

MAC/65

MAC/65

MAC/65

MAC/65

Systems Software for
Apple and Atari Computers

Optimized Systems Software, Inc.

a reference manual for

M A C / 6 5 .

a Macro Assembler and Editor program for
use with 6502-based computers built by
Apple Computer, Inc., and Atari, Inc.

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, of Chiselhurst, New Jersey. 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.

TRADEMARKS

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 "The Atari Assembler" by Don Inman and Kurt Inman and "Programming the 6502" by Rodney Zaks.

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

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

START UP

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

WARM START

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

BACK-UP COPY

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

SYNTAX

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

1. Capital letters designate commands, instructions, functions, ect., which must be entered exactly as shown (eg. ENTER, .INCLUDE, .NOT).

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

lno	- 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 is rounded to 4 and 100.1 is rounded to 100.
exp	- An assembler expression.
string	- A string of ASCII characters enclosed by double quotes (eg. "THIS IS A STRING").
strvar	- A string representation. Can be a string, as above, or a string variable within a Macro call (eg. %$1).
filespec OR file	- A string of ASCII characters that refers to a particular device. See device reference manual for more specific explaination.

3. Items in square brackets denote an optional part of syntax (eg. [,lno]). 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 parathesis indicate that any one of the items may be used , eg. (,Q) (,A).

---this page intentionally left blank--

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 so 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 chooses the MAC/65 EDIT mode) or TEXT (which allows general text entry). There is more discussion of the impact of these two modes below; but, first, there are several points in common to the two modes.

1.1 GENERAL EDITOR USAGE

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

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

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

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

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

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

```
WHAT?
```

1.2 TEXT MODE

The Editor supports a text mode. The text mode is entered with the command TEXT. This mode will NOT syntax check lines entered, allowing the user to enter and edit non-assembly language files. All Editor commands funtion 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 anywhere after the instruction, and comments may begin anywhere after the operand or instruction. Refer to Assembler Section for specific instruction syntax.

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

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

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

CHAPTER 2: EDITOR COMMANDS

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

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

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

Section 2.1

edit command: ASM

purpose : ASseMble MAC/65 source files

usage: ASM [#filespec1],[#filespec2],[#filespec3]

ASM will assemble the specified source file and will produce a listing and object code output. Filespec1 is the source device, filespec2 is the list device, and filespec3 is the object devive. Any or all of the three filespec's may be omitted, in which case MAC/65 assumes the following default filespec(s) are to be used:

filespec1 - user source memory.
filespec2 - screen editor.
filespec3 - memory (CAUTION: see below)

A "filespec" 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 D2:SOURCE, the assembler will list to D:LIST, and the object code will be written to D2: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:`

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.

Example: ASM #D:SOURCE , #P:

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

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

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

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

Section 2.2

edit command: BLOAD

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

usage: BLOAD #filespec

The BLOAD command will load a previously BSAVEd binary file, an assembled object file, or a binary file created with OS/A+ SAVe command.

```
Example:  BLOAD #D:OBJECT
```

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

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

Section 2.3

edit command: BSAVE

purpose: SAVE a Binary image of a portion of memory. Same as OS/A+ 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 OS/A+ SAVe command.

```
Example:  BSAVE #D:OBJECT<5000,5100
```

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

Section 2.4

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.

Section 2.5

edit command: C

purpose: Change memory contents

usage: C hxnum1 < (,)(hxnum) [(,)(,hxnum) ...]

Although MAC/65 does not include 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 5000<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.

Section 2.6

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.

Section 2.7

edit command: DEL

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

usage: DEL lno1 [,lno2]

DEL deletes source lines from memory. If only one lno is entered, only the line will be deleted. If two lnos are entered, all lines between and including lno1 and lno2 will be deleted.

Note: lno1 must be present in memory for DEL to execute.

Section 2.8

edit command: DOS [or, equivalently, CP]

purpose: exit from MAC/65 to the CP of OS/A+.

usage: DOS
or
CP

Either DOS or CP returns the user to OS/A+.

Section 2.9
----

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.

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 user memory before entering from the filespec.

Section 2.10
----

edit command: FIND

purpose: To FIND a string of characters somewhere in MAC/65's editor buffer.

usage: FIND /string/ [lno1 [,lno2]] [,A]

The FIND command will search all lines in memory or the specified line(s) (lno1 through lno2) 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.

Section 2.11

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,] [lno1 [,lno2]]

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

If only lno1 is specified, that line will be listed if it is in memory. If lno1 and lno2 are specified, all lines between and including lno1 and lno2 will be listed. When lno1 and lno2 are specified, neither one has to be in memory as LIST will search for the first line in memory greater than or equal to lno1, and will stop listing when the line in memory is greater than lno2.

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

EXAMPLE: LIST #D2:TEMP, 1030, 1800
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.

Section 2.12
----

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.

Section 2.13
----

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".

Section 2.14

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.

Section 2.15

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, 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-Z followed by RETURN.

Section 2.16

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,] [lno1 [,lno2]]

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.

Section 2.17

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 starting 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.

Section 2.18

edit command: REP

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

usage:
```
REP /old string/new string/ [lno1 [,lno2 ] ] [(,A)(,Q)]
```

The REP command will search the specified lines (all or lno1 through lno2) 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 occurrance.) 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/200,250,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.

Section 2.19

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
```

Section 2.20

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.

Section 2.21

edit command: TEXT

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

usage: TEXT

TEXT will CLEAR ALL user source from memory and put the Editor in the text mode. 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.

Section 2.22

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  =4608
          ?  8190   will print =$1FFE
```

---this page intentionally left blank--

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 OS/A+ 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 OS/A+ command line as the "MAC65" command. The formal usage is as follows:

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

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

And the options available 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 its 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 OS/A+. 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 all three filenames are given will the object code be directed to the file specified.

Note: if an assembly needs no listing but does need an object file, the user may specify E: as the listing file, thus sending the listing to the screen.

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.

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 form:

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 MOS Technology 6502 Programing Manual. We recommend that the user unfamiliar with 6502 assembly language programing should purchase:

"The Atari Assembler" by Don Inman & Kurt Inman
or
"Programing the 6502" by Rodney Zaks

3.2 INSTRUCTION FORMAT

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

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 anywhere 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.
```

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
```

3.5 OPERATORS

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

[ ]	pseudo 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) if 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).

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.

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.

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 & $00FF  is $0000
                           $03 ! $0A      is $000B
                           $003F ^ $011F  is $0120
```

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.

3.5.4 Operators: .OR .AND .NOT

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

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

3.5.5 Operator: - (unary)

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

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

3.5.6 Operators: < > (unary)

These UNARY operators are extremely useful when it is desired to extract just the high order or low order byte of an expression 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 immediate instruction.

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

3.5.7 Operator: .DEF

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

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

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

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

3.5.8 Operator: .REF

This unary operator tests whether the following label has been referenced by any instruction or directive in the assembly yet; and, in 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 cannot 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
                          .ENDIF
                          .IF .REF BLOOP
                        DOIT .= 1
                          .ENDIF
                          .IF DOIT
                          ...
```

3.5.9 Operator: [ ]

MAC/65 supports the use of the square brackets as "pseudo 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 anywhere in a MAC/65 expression to clarify or change the order of evaluation of the expression.

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

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

3.6 ASSEMBLER EXPRESSIONS

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

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

3.7 OPERATOR PRECEDENCE

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

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

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

3.8 STRINGS

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

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

CHAPTER 4: DIRECTIVES

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

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

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

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

Section 4.1

directive: *=

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

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

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

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

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

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

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

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

Section 4.2

directive: =

purpose: assigns a value to a label

usage: label = expression

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

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

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

Section 4.3

directive: .=

purpose: assign a possibly transitory value to a label

usage: label .= expression

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

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

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

Section 4.4
----

directive: .BYTE [and .SBYTE]

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

usage:

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

The .BYTE and .SBYTE directives allow the user to generate individ ial bytes of memory image in the output object. Expres ions must evaluate to an 8-bit arithmetic resu t. A strvar will generate as many bytes as the l ngth of the string. .BYTE simply assembles the ytes as entered, while .SBYTE will convert the byte to Atari screen codes (on the Atari) or to characters with their most significant bit on (on the Apple II).

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

This example wil produce the following output bytes:

4 42 43 03 FF.

Note that the ne ative expression was truncated to a single byte valu .

```
Exa ple: 50     .SBYTE "Hello!"
```

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

2 65 6C 6C 6F 01.

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

C E5 EC EC EF A1.

SPECIAL NOTE: oth .BYTE and .SBYTE allow an additive Modifier. A Mod fier is an expression which will be added to all of bytes assembled. The assembler recognizes the M difer expression by the presence of the "+" charac er. The Modifier expression will not itself be genera ed 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, inclcuding CR, TAB, etc.

Section 4.5

```
directive:       .DBYTE           [ see also .WORD ]

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

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

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

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

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

Section 4.6

```
directive:       .ELSE

purpose:         SEE description of .IF for purpose, etc.
```

Section 4.7

```
directive:       .END

purpose:         terminate an in-memory assembly
```

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

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

Section 4.8

directive: .ENDIF

purpose: terminate a conditional assembly block

SEE description of .IF for usage and details.

Section 4.9

directive: .ERROR

purpose: force an assembler error and message

usage: [label] .ERROR [string]

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

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

Section 4.10

directive:	.IF
purpose:	chooses 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.

The .IF, .ELSE, and .ENDIF directives control conditional assembly.

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      LDX  # >ADDRESS        ; these two lines will
12      LDY  # <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 begining of each pass. Missing .ENDIF(s) will NOT be flagged.

Section 4.11

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.

Section 4.12

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 "?" 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.

Section 4.13
----
directive: .OPT

purpose: selects various assembly control OPTions

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

usage notes: the valid options are as follows:

LIST	ERR	EJECT	OBJ
MLIST	CLIST	NUM	WAIT

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.

The following are the descriptions of the individual options:

LIST controls 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.

WAIT will cause assembly to halt before printing the page number and title string. Assembly will resume when the console START key is pressed (or RETURN key on Apple II).
WAIT is generally useful only with printers not capable of handling continuous paper.

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

LIST	listing IS produced
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
NO WAIT	use continuous paper, etc.
NO OBJ	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.

Section 4.14

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.

Section 4.15

directive:	.SBYTE
purpose:	produces "screen" bytes in output object
usage:	see .BYTE description, section 4.4

Section 4.16

directive:	.SET
purpose:	controls various assembler functions
usage:	.SET dcnuml , dcnum2

The .SET directive allows the user to change specific variable parameters of the assembler. The dcnuml specifys 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.

dcnuml	dcnum2	function
0	(4) 1-4	sets the .BYTE and .SBYTE listing format. 1 to 4 bytes can be printed in the object field 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.

Section 4.17

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 field respectively. The defaults are 8,12,20.

```
        Example: 200     .TAB 16,32,50
                 ...
                 1200    .TAB 8,12,20 ; restores defaults
```

Section 4.18

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.

Section 4.19
----

directive: .WORD [see also .DBYTE]

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
```

---this page intentionally left blank--

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 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 requires 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.

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 for further information.

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 will 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 available 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.

An example follows, on the next page.

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. (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.

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 simply 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.

5.4 MACRO PARAMETERS

Macro parameters can be of two types: expressions 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.
```

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 as 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,INCR-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
```

Of course, you can also do silly things, which will no doubt produce some pretty horrible (and hard to debug) code:

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

5.6 MACRO STRINGS

String parameters are represented in a macro definition by the characters "%$". 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 orginal
-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
```

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

5.7 SOME MACRO HINTS

Each person will soon develop his/her own style of writing macros, but there 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.

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
        ASLA
        ASLA
        ASLA
        ASLA     ; times 16
        TAX
        .ELSE
        LDX #%1*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,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 so complex. But MAC/65 provides the tools to do many things if you need them.

---this page intentionally left blank--

CHAPTER 6: COMPATIBILITY

There are many different 65Ø2 assemblers available, and it seems that each has a few foibles, bugs, or whatever that are uniquely its own (and, of 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 65Ø2 assembler to MAC/65. We will restrict ourselves to such things as directives and operators. We will NOT go into a discussion of how to convert the actual 65Ø2 opcodes (equivalently: instructions, mnemonics, etc.). We consider it mandatory that any good 65Ø2 assembler will follow the MOS Technology standard in this regard.

Example: We know of some antique 65Ø2 assemblers that specify the various addressing modes via special opcodes. Thus the conventional "LDA #3" becomes "LDAIMM 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 changed. The next section of this chapter, then, enumerates these changes.

6.1 ATARI'S ASSEMBLER/EDITOR CARTRIDGE

This section presents all known functional differences between the Atari cartidge 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).

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.

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
```

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.

CHAPTER 7: 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 - CONDITIONALS 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.

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 foward 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 boundrys. 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.

2Ø - 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 in a .SET specified a non-existant Assembler system parameter.

3Ø - 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 the operating system. Refer to the OS/A+ manual for detailed descriptions of such errors and their causes.

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 the 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 OS/A+ 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, whichever 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 A+, of length bytes into the given buffer address.

BPUT chan,buffer,length
Binary write of length 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.

a reference manual for

B U G / 6 5

an Assebly Language Debugging program for
use with 65Ø2-based computers built by
Apple Computer, Inc., and Atari, Inc.

PREFACE

BUG/65 is an interactive debugging tool for use in the development of assembly language programs for the ATARI 800 or ATARI 400 personal computers. It's designed to take as much of the drudgery out of assembly language debugging as possible. The design philosophy behind BUG/65 is that the computer should serve as a tool in the debugging process as opposed to a hindrance. One result of this philosophy is that BUG/65 requires a relatively large amount of memory when compared to simpler debug monitors. This is the result of a tradeoff between memory and functionality, with function winning out.

BUG/65 is a RAM loaded machine language program occupying 8K of memory; it is self relocatable as shipped and requires a full 48K bytes of memory. BUG/65 is also designed to be floppy disk based - it isn't intended to be used in cassette-only systems. BUG/65 was designed for use by an experienced assembly language programmer.

BUG/65 is an original product of the McStuff Company, which developed the product under the name "McBUG", which name is their trademark.

For use on the ATARI 800 or 400 computer with a minimum of 48K of RAM and one floppy disk drive.

TRADEMARKS

TABLE OF CONTENTS

(continued)

TABLE OF CONTENTS (continued)

SUMMARY OF MAJOR FEATURES OF BUG/65

* A full set of debugging commands - change memory, display memory, goto user program with break points, etc.

* Binary file read and write, including appended write.

* A disassembler.

* An instant assembler providing labeling capability.

* Expanded command addressing capability: hex or decimal addresses, + and - operators supported, relocated addresses supported.

* Read or write disk sector(s).

* Multiple commands permitted in a command line. Command lines can be repeated with a single keystroke or repeated forever with the special slash operator.

* Support for relocatable assemblers - the base of a module can be specified and then used to reference addresses in that module.

* BUG/65 commands can be executed from a command file, and there is a command to create command files.

* Hex to decimal and decimal to hex conversions provided.

* Memory protection of BUG/65's code and data. BUG/65 won't allow you to use a BUG/65 command that will destroy any part of BUG/65 itself. For example, you can't use the Fill command to overwrite BUG/65's code.

* Page zero sharing. BUG/65 shares page zero with a user program by keeping two copies of the shared page zero locations - one for the user and one for BUG/65 itself.

SECTION 1 : COMMAND SUMMARY

This section is intended to be a handy reference guide and will probably prove indispensable after the user has thoroughly read through the rest of this manual. For the experienced debug user, might we suggest at least a quick perusal of Sections 2 through 6 and Sections 8 and 9.

The following table is simply a syntax summary of the available commands. Excepting for the first three commands (which are described in Section 8), all the commands are described in alphabetical order in Section 7.

COMMAND CODE	SYNTAX	PURPOSE
{RETURN}		Repeat last command line
/		When appended to a command line: repeat line forever.
=		Display last command line
A	A <addr>b̸	Ascii mode memory change
B	B <addr>	Base address for relocation
C	C <startaddrl> <endaddrl> <startaddr2>	Compare memory blocks
D	D <startaddr> [<endaddr>]	Display memory
E	E #filespec	Execute a command file
F	F <startaddr> <endaddr> [<value>]	Fill memory block with value
G	G [<startaddr>] [@<breakpoint> [Rn=<value>] [I=<count>]	Go at address, set optional breakpoint, with optional Regist value breakpoint and pass Counte
H	H <numberl> <number2>	Hexadecimal arithmetic result
I	I	disk Inventory (directory listin
J	J #filespec,string	create command file
K	K <number>	convert hex to decimal

L	L <startaddr> <endaddr> <byte1> [<byteN> ...]	Locate byte string in memory block
M	M <startaddr> <endaddr> <toaddr>	Move memory block
P	P [S] [P]	Print output on Screen and/or Printer
Q	Q	Quit...go to OS/A+
R	R [<offset>] #filespec	Read a binary file to memory with optional offset
R%	R% [<sectornumber> [<bufferaddr> [<numsectors>]]]	Read sector(s) from disk to memory buffer
S	S <addr>b̸	Substitute memory, numeric mode
T	T [S] [<count>]	Trace, with optional Skip over subroutine calls, for (optional) count intstructions.
U	U <addr> [<param>]	call User routine at given address and pass optional parameter in X,Y registers
V	V	View user registers
W	W [:A] <startaddr> <endaddr> #filespec	Write a block of memory to a binary image file, optionally appending instead of creating new file.
W%	W% [<sectornumber> [<bufferaddr> [<numsectors>]]]	Write sectors from memory buffer to disk
X	XA or XX or XY or XS or XP or XF	change user register value
Y	Y <startaddr> [<endaddr>]	dissasemble memory block
Z	Z <addr>b̸	instant assembler (at address)

SECTION 2: Notations Used In This Manual

The following notations are used in this manual:

<...>	Is used to indicate a numerical address parameter. The address expression between the two characters "<" and ">" may be any valid address as described in Section 3. For example, <START> means that you can enter any valid address expression to specify the START parameter.
ȸ	Is used to indicate one and only one blank. In most cases, blanks are insignificant and any number of them may be entered between commands and parameters. However, in certain cases, one and only one blank must be entered - this blank is indicated by the "ȸ" character.
[...]	Is used to specify an optional parameter. For example, [<VALUE>] would indicate that VALUE is an optional address parameter. You'll find that many parameters are optional, and in such cases logical default values will be supplied by BUG/65.
or	Is used to delimit a list of choices. In such a list, one and only one choice may be used. For example, "+ or -" indicates that you may enter a plus sign or a minus sign, but not both.
filespec	Is used to indicate a standard OS/A+ filespec. This consists of the device name followed by a colon and the filename. For example, "D:DATAFILE" is a valid filespec for a file named DATAFILE on disk drive one.

SECTION 3: Address Parameters

BUG/65 allows numerical addresses to be specified in a variety of ways. You can use hexadecimal or decimal notation, add and subtract terms, or add a relocation factor to any address. The following Backus-Naur definitions describe the various address types:

```
<ADDR>     :=   + or - <TERM>  [ + or - <ADDR> ]

<TERM>     :=   <NUMBER>  or  X<NUMBER>

<NUMBER>   :=   <DECNUM>  or  <HEXNUM>

<DECNUM>   :=   .<DECIMAL DIGITS>

<HEXNUM>   :=   <HEXADECIMAL DIGITS>
```

In the above, the only item not literally defined is the "X" item in the definition of a TERM. This is used to indicate that the following NUMBER is to be relocated by adding the value of the current relocation base to the value of NUMBER. The current relocation base is set by the "B" command.

All address parameters are interpreted as 16-bit positive numbers in the range of Ø to 65535. Overflow isn't detected or reported as an error.

Some examples will help (all of these are valid address expressions):

1FA1	a hexadecimal number.
.1ØØ	a decimal number (one hundred).
1ØØØ+.2Ø	a hexadecimal number plus a decimal number. This evaluates to 1Ø14 hex (4116 decimal).
1+2-3+4	a long expression. Evaluates to 4.
X1234	a relocated address. If the current relocation base has the value $1ØØØ, then this expression will evaluate to $2234.

3.1 Spaces as Parameter Delimiters

BUG/65 uses spaces as parameter delimiters. This makes for easier and quicker entry of commands. However, it does introduce some conventions regarding the use of spaces that you must be aware of:

* Spaces may not be embedded in a number. For example, "12 34" is interpreted as two parameters ($12 and $34) and not as the single parameter $1234.

* Spaces aren't allowed between the "X" relocation specifier and it's associated relocated address. For example, "X 1234" is interpreted as two parameters. The first will have the value of the current relocation base and the second is $1234.

* Any number of spaces may be used to separate two parameters. For example, "1234   5678" is a perfectly valid way of entering the two parameters $1234 and $5678.

SECTION 4: Loading and Running BUG/65

BUG/65 is shipped on your master diskette as a relocatable COMmand file, named "BUG65.COM". Therefore, BUG/65 functions just as does any OS/A+ extrinsic command: simply type "BUG65" when OS/A+ prompts with D1: (or Dn: if you have changed default drives...see the OS/A+ manual for more details) and BUG/65 will load into memory and relocate itself to just above the current value of LOMEM (contents of $2E7-$2E8).

4.1 Specifying BUG/65's Load Address

If you need BUG/65 to load at some location other than LOMEM (which is typically around $2ØØØ with OS/A+ version 2 and around $2CØØ with version 4), you may also enter a load address on the OS/A+ command line. The address must be in hex, must be at or below $9AØØ, and should be above LOMEM. Remember, BUG/65 occupies 8K bytes, which means it will occupy memory starting at the address you give and ending $2ØØØ bytes higher.

EXAMPLE:

```
[D1:]BUG65 8ØØØ
```

This usage will load BUG/65 at $8ØØØ, set its restart point at $82ØØ, and occupy memory from $8ØØØ through $9FFF.

4.2 Creating a Non-Relocatable Version

In order to allow itself to be relocated virtually anywhere in memory, BUG/65 as shipped includes a relocation bit map and a relocation program. In addition, relocatable BUG/65 always loads in at locations $9800 through $BC00. If these addresses are "poison" to you (e.g., if you want to use BUG/65 with a cartridge plugged in), you may wish to produce a non-relocatable version designed to run within an address range you pick.

If so, USING A 48K SYSTEM, simply specify the loadpoint, as shown in the preceding section (e.g, via "BUG65 7000") and allow BUG/65 to load and relocate. Then exit to OS/A+ (via Quit) and use the OS/A+ intrinsic command SAVE to save a non-relocatable version. The address range to be SAVEd may be calculated as follows:

```
SAVE filename.COM loadpoint+$200 loadpoint+$2000
```

Thus, if you had specfied "BUG65 7000", you could save the non-relocatable version via

```
SAVE BUG7000.COM 7200 9000
```

thus also giving it a name which will later remind you where it will load at. To execute this non-relocatable version, simply type in its name (BUG7000 in the example shown).

SECTION 5: Command Entry

When you see BUG/65's input prompt (the ">" character) in the left-hand column of the screen, then you're in command entry mode. Any data typed at that point will be entered into the command line buffer - the command line isn't executed until you type RETURN. You can enter as many commands in one command line as will fit in the command line buffer (1ØØ characters). As soon as you type the RETURN, you'll leave command entry mode and BUG/65 will begin executing the command(s) in the command line.

You can tell the difference between command entry mode and command execution mode. In command entry mode, the cursor is displayed. When a command is executing, the cursor is blanked.

If you try to enter more than 1ØØ characters in the command line, BUG/65 will beep the bell and not allow any more characters to be input. At that point, you may either hit RETURN to execute what's in the command line so far, or edit some characters out of the command line with the BACKSPACE key.

5.1 Command Line Editing

When entering commands, you may edit mistakes with the BACKSPACE key. The BACKSPACE will move the cursor one column to the left and delete whatever character was in that column. Unfortunately, the normal system editing facilities aren't supported. This is because of the manner in which BUG/65 does keyboard input.

5.2 Normal and Immediate Type Commands

BUG/65 has two types of commands - normal and immediate. Normal commands are those that don't require interaction with the operator for their execution. Immediate commands do require operator interaction. Normally, you'll never be aware of the distinction between the two types - command entry "flows" without any consideration of the command type required. The only difference is that an immediate command must be the first command entered in a command line. Once an immediate command is entered, BUG/65 will begin interacting with the operator for further input. Since this interaction is required for completion of the command, it doesn't make sense to allow immediate commands to be "stacked" in the middle of a command line for execution between other commands. If you try to enter an immediate command in the middle of a command line, you'll get an "IMMEDIATE ERROR" error message and find yourself back in the command entry mode.

The immediate commands are the "A" command (ASCII memory change), the "S" command (hex memory change), the "X" command (change user registers), and the "Z" command (instant assembler).

5.3 Command Execution

For a normal type command, BUG/65 will begin command execution as soon as you type RETURN. For immediate type commands, BUG/65 will begin command execution as soon as you type the command character (provided that character is the first character in the command line).

5.4 Multiple Commands on a Line

Multiple commands may be entered on the same command line. Normally, successive commands in the command line don't require command separators between them other than at least one space character. The exceptions to this are commands for which an optional parameter is being defaulted. For example, the display memory command ("D") may have an optional parameter specified as the end of the area of memory to be displayed. If that ending parameter isn't specified, BUG/65 will default the end to the start plus eight bytes. If you wanted to enter two successive display commands in the command line without defaulting the end parameters, you could type

```
D 1000 1010 D 2000 2010
```

and no command separators would be required because BUG/65 knows that the "D" command only has two parameters and will interpret further characters in the command line as the beginning of a new command. However, if you wanted to default the ending address of the first display command, then you'd have to insert a command separator so that BUG/65 knows that the first display command is finished. If you didn't do this, then the second display command "D" would be interpreted as the second parameter of the first display command (the end address would be interpreted as $ØD. The command separator is a comma, so in this case you would enter the commands as follows:

```
D 1000, D 2000 2010
```

SECTION 6: Command Termination

This section describes the many ways that a command will stop.

6.1 Normal Termination

Once a command line is given to BUG/65 for execution, BUG/65 will execute all of the commands in the line to conclusion before returning to command entry mode. It's possible to instruct BUG/65 to execute a command line "forever" (see Section 8.2), in which case BUG/65 will never come back to command entry mode until you manually intervene (with ESC or BREAK - see Section 6.4)

6.2 Error Termination

If an error occurs in command execution, BUG/65 will beep the bell and display a short error message in English indicating the cause of the error. Command execution will stop and you'll enter the command entry mode. Any commands in the command line after the command which caused the error won't be executed. (You should also be aware that BUG/65 will close any file that has been opened using IOCB number one when any error occurs.) (A complete list of error messages is in Section 14.)

6.3 Command Suspension

Once BUG/65 begins executing a command line, you may temporarily suspend command execution by hitting the space bar. This will put BUG/65 in a "hold" condition, at which point you have two alternatives: you can restart the command by hitting the space bar again, or you can abort the command with ESC or BREAK.

6.4 Command Abort

You can abort any command that is executing (except for the read and write disk commands) by hitting the ESC or BREAK keys. BUG/65 will stop executing the command and you'll enter command entry mode.

6.5 The RESET Key

BUG/65 traps the RESET key so that hitting RESET will bring you back to BUG/65. RESET will stop any command that is executing. You'll see the BUG/65 version and copyright prompt, and you'll be in command entry mode. RESET will reset all of BUG/65's internal stuff except for any user defined or modified parameters. For example, the user's registers, the current relocation base, etc., aren't cleared on a RESET - they'll retain whatever values they had before the RESET. (All of this depends, however, on the fact that the reset vectors haven't been modified by the user - either by using a BUG/65 command or by a user program. If you've modified the reset vectors, then the action of the RESET key is your responsibility.)

6.6 Manual Restart

Since BUG/65 is relocatable, the manual restart point (coldstart) depends upon where it has been relocated to. If you specified an address to load BUG/65 when you gave the OS/A+ command line (e.g., BUG65 4000), then the coldstart point is $200 greater than the address specified, and you may use 'RUN address' from OS/A+ if desired (e.g, RUN 4200 if the original command was BUG65 4000). In any case, you may inspect location $000C (via the BUG/65 command 'D C') to determine the coldstart point. The 6502 word address in locations $0C and $0D (LSB, MSB order) points to BUG/65's restart point. The result of a manual restart is the same as if the default RESET key processing occurred (see section 6.5).

SECTION 7: Command Descriptions

Throughout the descriptions of the commands, comments are sometimes presented in the command line examples. These are denoted by the characters "*/". Anything appearing on a line after these characters is a comment and is NOT part of the command line being exemplified.

The commands are presented in alphabetical order.

7.1 A - Change Memory, ASCII mode

```
A <ADDR>b̸
```

The A command allows you to replace the contents of memory bytes beginning at location <ADDR> with ASCII characters. As soon as you type the required space character after the address, BUG/65 will prompt you with the current contents of the memory location at <ADDR>. Those contents will be displayed as an ASCII character. At that point, you have the following options:

1. Typing a SPACE will cause the current memory location to be skipped and the contents of the next memory location to be displayed.

2. Typing an UNDERLINE will cause the current address to be decremented by one. The new address is then displayed on the next line of the screen followed by the contents of the new memory location.

3. Typing a RETURN will cause the address of the current memory location to be displayed on the next line of the screen followed by the contents of the current location.

4. Typing ESC will get you out of the command and back into command entry mode.

5. Typing any character other than "@" will cause the ATASCII value of that character to be entered into memory at the current address. The address is then incremented by one and the contents of the new memory location are displayed.

6. Typing the character "@" causes the next character typed to be entered into the current memory location as its pure ATASCII value without any of its control character significance. For example, typing "@ ESC" will insert the ATASCII value for ESC into memory. The address is then incremented by one and operation continues as in 5. above.

After you exercise any option except option 4., BUG/65 will again prompt you with the contents of the current location and you may then choose from any option again.

7.2 B - Set Relocation Base

```
B <ADDR>
```

The B command will set the value of the relocation base to ADDR. The relocation base is intended for use with relocating assemblers. In a relocatable environment, listings typically are addressed from location zero. When a module to be debugged is subsequently loaded into memory, it will have a relocation offset added to the addresses in the listing. The B command allows you to set the relocation base to the load address of the module you're working on and then to reference addresses within the module by simply prefixing each address expression with the relocator symbol "X".

For example, suppose that a relocatable module is loaded at location $5380 in memory. Suppose further that we want to display the contents of a memory location which is $230 from the beginning of the module. The following commands would do the job:

```
B 5380, D X230
```

The world isn't overrun with relocating assemblers for the ATARI. However, until it is, the B command has other useful applications. These take advantage of the fact that the relocation base value is a variable which can be modified during command execution. For example, suppose you know that the string of characters "ABCD" is stored somewhere on a diskette and you want to find the sector that contains it. The following commands will do the trick:

```
B 1

D X, R% X 4000 1, L 4000 407F 41 42 43 44, B X+1/
```

This uses some commands not introduced yet, but this is what happens: First, X is set to 1 with one command line. Then a second command line will display memory at the location X (so you'll know where you're at as you step through), read sector number X into memory locations $4000-$407F, locate the string "ABCD" in that sector buffer, then bump X by one for the next sector. The slash at the end of the command line means that the command line will execute forever. What will happen is that BUG/65 will continuously read diskette sectors. For every sector read, you'll see at least a memory display of eight bytes beginning at address X (which is the sector number). If the Locate instruction finds the string "ABCD" in the sector buffer, it will display the location of the string. At that point, just hit ESC to stop the command, and display the value of X ("D X RETURN"). The sector containing the string will either be the value of X or one before it, depending on how fast your ESC was.

7.3 C - Compare Memory Blocks

```
C <STARTBLOCK1> <ENDBLOCK1> <STARTBLOCK2>
```

Compare is used to compare the contents of two blocks of memory. The block of memory beginning at STARTBLOCK1 and ending with ENDBLOCK1 is compared to the same size block beginning at STARTBLOCK2. If both blocks are the same, then there will be no output. If any bytes in the blocks differ, then BUG/65 will display a line of data in the following format for every byte that is different:

```
AAAA = BB CCCC = DD
```

where AAAA = the hex address of the differing location in the first block, BB = the hex contents of location AAAA, CCCC = the hex address of the differing location in the second block, and DD = the hex contents of location CCCC.

7.4 D - Display Memory

```
D <START> [ <END> ]
```

The D command displays the contents of the memory block beginning at START and ending at END. If END isn't specified, then the default value of START+7 is used. The memory block is displayed in the following format:

```
AAAA = BB BB BB BB BB BB BB BB   CCCCCCCC
```

where AAAA = the hex address of the first byte in this line, BB = the hex contents of successive memory locations beginning at location AAAA, and C = the ASCII character interpretation of the positionally corresponding BB value of the byte.

7.5 E - Execute a Command File

```
E #filespec
```

The E command is used to execute a command line from a command file. The file specified by filespec must consist of a line of BUG/65 commands and parameters and must be ended with an ATASCII EOL character ($9B). BUG/65 will only execute one command line from a command file and then it will stop reading the file. Command files can be chained however, so that the last command in one file can execute another command file. An E command should be the last command in a command line because any commands after the E in the line won't be executed.

7.6 F - Fill a Memory Block with a Value

```
F <START> <END> [ <VALUE> ]
```

The F command will fill the block of memory beginning with START and ending with END with VALUE. If VALUE isn't specified, then zero will be used. Note that VALUE is a byte value - the least significant byte of the 16-bit VALUE will be used for the fill.

7.7 G - Goto a User Program

```
G  [<START>] [@<BRKPOINT> [RN=<VALUE>] [I=<COUNT>] ]
```

The G command will execute a user program beginning at START. If START isn't specified, then execution begins at the current value of the user's PC register. BRKPOINT is an optional breakpoint. If the user's program trys to execute the instruction at BRKPOINT, the program will break back to BUG/65 and BUG/65 will display the contents of the user's registers at that point. Examples:

```
 G  1ØØØ   /* go at location $1ØØØ, no breakpoint

 G  @43ØØ  /* go from wherever our PC was and break
           /* at location $43ØØ
```

A breakpoint may be conditionally qualified by a required value in a specified register. "RN=<VALUE>" will tell BUG/65 to break at that point only if the value of user register "N" equals VALUE. If that condition isn't met, then the user's program is allowed to continue executing at the location of the breakpoint. (The instruction that was at the breakpoint location WILL be executed.) The mnemonic names of the registers that may be specified for "N" are: A, X, Y, S, and F, which stand for the user's A, X, Y, Stack, and Status (flags) registers respectively. (Note that only the least significant byte of VALUE is used for this qualification.)

Example:

```
 G  1ØØØ  @1422  RX=33  /* go from location $1ØØØ and
                        /* break at location $1422
                        /* only if register X equals
                        /* $33
```

A breakpoint may also be qualified with an iteration counter. "I=<COUNT>" tells BUG/65 to allow the execution of the instruction at the breakpoint COUNT times before breaking.

Example:

```
 G   1ØØØ  @23ØØ  I=2  /* go from location $1ØØØ and
                       /* break the second time we hit
                       /* the instruction at $23ØØ
```

The register and iteration qualifications may be used together. In this case, the register condition must be met before the iteration counter is decremented. As in the following example:

```
G  1000  @1234  RA=50  I=3   /* go from location $1000
                             /* and break the third time
                             /* the instruction at loc-
                             /* ation $1234 is executed
                             /* with register A equal
                             /* to $50
```

All of this flexibility isn't without its price, however. Because BUG/65 has to do quite a bit of evaluation at every breakpoint before deciding if the break condition has been met, don't expect to be able to conditionally pass through breakpoint instructions at real-time speed. As long as you never execute the instruction at the breakpoint, you're OK, but as soon as BUG/65 gets the break, expect several hundred instructions to be executed before your program is given back control after the break isn't met.

Also, BUG/65 was NOT designed to allow breakpoints in PROM resident code. If you attempt to set such a break point, or if you try to set a breakpoint at a non-existent memory location, you'll get a "BREAKPOINT ERROR".

One other thing. BUG/65 will automatically remove breakpoints from your program after a break occurs. Breakpoints aren't left set after the break is performed.

7.8 H - Hexadecimal Arithmetic

```
H  <NUMBER1>  <NUMBER2>
```

The H command will calculate the sum NUMBER1 + NUMBER2 and the difference NUMBER1 - NUMBER2 and display the results on the next line of the screen as two hex words. The sum is the first word displayed, the difference is the second.

7.9 I - Display Disk Directory

```
I
```

The I command will display the directory of the diskette in drive one. The display can be suspended or halted with the SPACE or ESCAPE keys respectively.

7.10 J - Create a Command File

```
J #filespec, string
```

The J command allows you to create command files for execution by the E command. The string in the command is any string of valid BUG/65 commands. The string will be written to the file specified by filespec in the format expected by the E command. Please note the comma after the filespec - it's required, else BUG/65 won't know where your filespec stops and your command string starts. Also note that the J command doesn't allow multiple commands in the command line to be executed after the J command - everything in the line after the filespec and up to the RETURN is written to the file instead of being executed.

7.11 K - Convert Hex to Decimal

```
K <NUMBER>
```

The K command will convert NUMBER to a decimal number and display the result on the next line of the screen. NUMBER can be any valid address expression.

To convert decimal to hex, just display memory at the decimal location of the number you want to convert. The hex equivalent of the decimal location appears in the display output as the hex word on the beginning of the line. For example, to convert 1000 decimal to hex, just execute the command "D .1000". You'll see the hex conversion of 1000 as the first hex word on the next line.

7.12 L - Locate a Hex String

```
  L  <START>  <END>  <BYTE1> <BYTE2> ... <BYTEn>
```

The L command will search the block of memory beginning at START and ending at END for a hex string. The hex string is defined by BYTE1...BYTEn, which are interpreted as the hex bytes of the pattern string. (Only the least significant bytes of the address values are used for each byte in the string.) Wildcard bytes which will match any byte in memory may be specified by the character "*" in the string. BUG/65 will output the addresses of every occurrence of the string found in the block. For examples:

```
L 1000 10FF  41  42  43 /* will locate any occur-
                        /* rences of the string "ABC"
                        /* in the memory block
                        /* $1000 to $10FF

L 1000 2000  10  *  20  /* will locate any occur-
                        /* rences of a three-character
                        /* string which begins with
                        /* $10 and ends with $20 in
                        /* the memory block $1000
                        /* to $2000
```

7.13 M - Move a Memory Block

```
  M  <START>  <END>  <TO>
```

The M command will move the block of memory beginning at START and ending at END to TO. BUG/65 will take care to handle overlapping moves correctly, either for moves up or down.

7.14 P - Select Output Devices

```
P [S] [P]
```

The P command is used to select output to either the screen ("S") or the printer ("P") or to both ("SP"). For example:

```
P  S       /* turns screen output on, printer output off
P  P       /* turns printer output on, screen output off
P  S  P    /* turns both screen and printer output on
P          /* turns both outputs off - commands will
           /* still be accepted and executed, you just
           /* won't see their entry or output anywhere.
```

In addition to allowing you to list BUG/65 results to the printer, this command was designed to allow you to debug the generation of intricate screen displays without having the outputs of BUG/65 commands scroll your display off the screen. It is a little crude, and might have a few problems depending on what your program has done to OS, but is handy to have in emergencies. (The LFFLAG and NULFLG bytes in the Configuration Table can help you here - see section 11.)

7.15 Quit to OS/A+ command

```
Q
```

The Q command will coldstart DOS. The results are essentially the same as when you power-up the machine.

7.16 Read Commands

7.16.1 R - Read a File

```
  R  [ <OFFSET> ]  #filespec
```

The R command is used to load binary files. If OFFSET is specified, then OFFSET is added to the load address(es) specified in the file, and the data will be loaded at the loading point(s) plus OFFSET. This allows you to load a file into a different memory location than where it is origined at. After the file is loaded, the load starting point specified in the file is placed into the user's PC register.

BUG/65 supports concatenated binary file sections as described in the DOS 2.ØS manual. If such a file is loaded using the OFFSET option, however, ALL file sections will be loaded starting at the load addresses specified in the file plus OFFSET. In addition, the user's PC register will contain the value of the load point of the last file section loaded (not plus OFFSET).

7.16.2 R% - Read Sector(s)

```
  R%  [ <SECNO>  [ <BUFFER>  [ <NOSECS> ] ] ]
```

The R% command allows you to read a sector or a group of sectors from a diskette in disk drive number one. SECNO specifies the sector number to be read and defaults to one. BUFFER specifies the buffer the sector is to be read into and defaults to BUG/65's loadpoint plus $2ØØØ. NOSECS specifies the number of sectors to read and defaults to one. If more than one sector is specified, then consecutive sectors are read sequentially into memory beginning at BUFFER.

7.17 S - Change Memory, Numeric mode

S <ADDR>␢

The S command allows you to replace the contents of memory bytes beginning at location ADDR with numerical values. As soon as you type the required space character after the address, BUG/65 will prompt you with the current contents of the memory location at ADDR. Those contents will be displayed as a hexadecimal byte value. At that point, you have the following options:

1. Typing SPACE will cause the current memory location to be skipped and the contents of the next memory location to be displayed.

2. Typing an UNDERLINE will cause the current address to be decremented by one. The new address is then displayed on the next line of the screen followed by the contents of the new memory location.

3. Typing a RETURN will cause the address of the current memory location to be displayed on the next line of the screen followed by the contents of the current location.

4. Typing ESC will get you out of the command and put you back into command entry mode.

5. Typing an address value (any valid address expression) will cause that value to be entered into memory at the current address. The address is then incremented by one and the contents of the new memory location are displayed. (Only the least significant byte of the address value will be entered into memory.)

After you exercise any option except option 4., BUG/65 will again prompt you with the contents of the current memory address and you may select any of these options again.

7.18 T - Trace a User Program

```
  T  [S]  [ <COUNT> ]
```

The T command will single-step through user program instructions beginning with the instruction at the current user PC register. The number of instructions to be executed are specified by COUNT, which defaults to one. If "S" is specified, then all of the instructions in a subroutine are counted as one instruction for tracing purposes - the trace is turned off until return from the subroutine ("S" stands for "skip the subroutine"). After every instruction traced, BUG/65 will display the contents of the user's registers. Some examples:

```
T                /* will execute one instruction and then
                 /* display the register contents

T 5              /* will execute five instructions, displaying
                 /* registers after each instruction

TS  10           /* will execute 16 instructions. If any of
                 /* the instructions are JSR's, then the
                 /* trace will be turned off after the JSR
                 /* until the subroutine executes an RTS
```

The trace command can't be use to trace instruction execution through PROM resident code. Any attempt to do so, or to trace through non-existent memory, will result in a "BREAKPOINT ERROR".

7.19 U - Call a User Subroutine

```
  U  <ADDR>  [ <PARAM> ]
```

The U command is used to call a user subroutine at ADDR. The user routine is passed the optional parameter PARAM in the X register (low byte) and Y register (high byte). The user routine should return to BUG/65 via an RTS instruction. If PARAM isn't specified, then zero is used.

7.2Ø   V - Display User's Registers

```
V
```

The V command will display the contents of the user's registers in the following format:

```
 A  X  Y SP NV_BDIZC   PC  INSTR
HH HH HH HH BBBBBBBB HHHH  LDA  1ØØØ,X
```

This is interpreted as follows:

```
A   =  the hex value of the A reg
X   =  the hex value of the X reg
Y   =  the hex value of the Y reg
SP  =  the hex value of the stackpointer
N   =  the binary value of the negative flag
V   =  the binary value of the overflow flag
_   =  the binary value of an unused bit in the
         status reg
B   =  the binary value of the break flag
D   =  the binary value of the decimal flag
I   =  the binary value of the interrupt enable bit
Z   =  the binary value of the zero flag
C   =  the binary value of the carry flag
PC  =  the hex value of the PC reg (This is a
        pseudo register maintained by BUG/65.
        It contains the location of the next
        user program instruction to be executed.)
INSTR = the instruction at the current PC
```

7.21 Write Commands

7.21.1 W - Write a File

```
W [ :A ] <START> <END> #filespec
```

The W command is used to write a binary file. Memory from START to END is written to the file specified by filespec in the standard OS/A+ binary file format. If the ":A" option isn't specified, then the data written will replace the current contents of the file if the file already exists. If the ":A" option is specified, then the data is appended to any data already in the file. A load header consisting of a start and end address as described in the OS/A+ manual will precede the appended data.

7.21.2 W% - Write Sector(s)

```
W% [ <SECNO> [ <BUFFER> [ <NOSECS> ] ] ]
```

The W% command is used to write a sector or a group of sectors to a diskette. SECNO specifies the sector number to be written and defaults to one. BUFFER specifies the memory location of the sector data to be written and defaults to the BUG/65 loadpoint plus $2ØØØ. NOSECS specifies the number of sectors to be written and defaults to one. If more than one sector is specified, then consecutive sectors are written sequentially from memory beginning at BUFFER.

7.22 X - Change User's Registers

X REGNAME

The X command allows you to change the contents of user registers. REGNAME is a one-character register name mnemonic. The allowed register names and their meanings are:

A	=	A register
X	=	X register
Y	=	Y register
S	=	stackpointer register
P	=	program counter pseudo-register
F	=	status register (flags)

After you type in the name of the register to be changed, BUG/65 will prompt you with that name character followed by an equals sign. At that point you have the following options:

1. Enter the new value for the register. The new value may be any valid address expression. After the new value, typing RETURN will end the command. Or you can type SPACE which will prompt you with another register name for possible change. The next register name is determined by the order of the above list. For example, if you change register Y then hit a space after the new value, BUG/65 will prompt you for possible change of register S. This prompt list continues through register F and then wraps back to register A again.

2. Enter RETURN or ESC to end the command. BUG/65 will display the new contents of the registers and then put you back into command mode.

7.23 Y - Disassemble Memory Block

```
Y <START> <END>
```

The Y command will disassemble instructions in memory beginning at START and ending at END. The following conventions are used in the disassembly:

1. Standard MOS Technology mnemonics are used for opcodes.

2. Illegal opcodes are displayed as "***".

3. All numeric operands are displayed as hexadecimal numbers.

4. Zero page operands will display as two hex digits, all other non-immediate operands will display as four hex digits.

5. No operand is displayed for accumulator mode operands.

7.24 Z - Instant Assembler

Z <ADDR>b̸

The Z command allows you to assemble instructions to be stored in memory at ADDR. Immediately after typing the SPACE character (or RETURN, which is allowed as well), BUG/65 will prompt you with the current program counter value of the instant assembler (which initially will be ADDR). At that point you may type in a valid assembly language instruction. The format for an instruction line is:

[<LABEL>] <OPCODE> [<OPERAND>]

LABEL may be any label in the form "Ln", where "n" may be any digit from zero to nine. OPCODE may be any valid MOS Technology instruction mnemonic or one of two pseudo-ops (described below). OPERAND, if allowed by the addressing mode of the instruction, may be any valid address expression. At least one space must separate a label from an opcode or an opcode from an operand.

After typing your instruction, type RETURN and the instruction will be entered into memory at the current PC if it doesn't contain any errors. If there are any errors, then BUG/65 will display an error message and will reprompt you with the current (unchanged) PC. If there are no errors, then BUG/65 will display the object code created by the instruction to the right of the instruction on the screen and will prompt you with the PC of the next instruction on the next screen line. You may exit the instant assembler by typing ESC at any time, or by typing RETURN by itself in response to the PC address prompt.

The instant assembler provides you with two pseudo-ops. "/" followed by an address will change the PC to that address. It acts like an ORG ("*=") pseudo-op. For example, "/4000" will set the PC of the next instruction location to $4000. "+" followed by an address will insert the value of that address (least significant byte) at the current PC and bump the PC by one. It acts like a DB (.BYTE) pseudo-op. For example, "+34" will insert the hex byte 34 at the current PC.

The instant assembler provides a simple labeling capability. You may prefix an instruction with a two character label of the form "Ln", where "n" may be any digit from Ø-9. You may then use that label as an operand in an instruction, with the following three restrictions:

1.) Immediate type operands (#HH) can't be labels.

2.) Indirect type operands can't be labels.

3.) A label can't be combined with any of the standard address operators (+, -, X, etc.)

Label references may be forward or backward. BUG/65 will store unresolved references and resolve them when the label is later defined. You may reference undefined labels twenty times before BUG/65 runs out of room to store the unresolved locations - you'll then get an error message and the assembly will be aborted. The same label may be reused more than once. In such cases, BUG/65 will use the last defined address of the label when it is referenced.

If any labels have been referenced but not defined when you exit the instant assembler, BUG/65 will prompt you with a message and the label name followed by an equals sign. At that point you may either define the label by entering any valid address expression followed by a RETURN, or you may chose not to define it and simply hit RETURN. If you don't define the label, then the value of the label is defaulted according to the following two rules.

1.) If an instruction using the undefined label is a relative branch, then the value of the label for that instruction defaults to the location of the instruction plus two.

2.) For all other instructions, the value of the label defaults to the location of the instruction plus three.

These rules guarantee that all branching instructions using undefined labels are effectively turned into NOP'S. This offers some measure of protection against a program going into never-never land. (If you reference a label that isn't yet defined, the object code displayed to the right of the instruction on the screen will show addresses generated according to these rules. Don't worry, when the label is subsequently defined, BUG/65 goes back and fixes up all these references.)

SECTION 8: Special Command Modifiers

8.1 Repeat Last Command Line

{RETURN}

The last command line entered and executed may be repeated without typing the whole thing in again - just hit RETURN. BUG/65 remembers the last line entered for just this purpose.

8.2 Repeat Command Line Forever

/

Appending a slash to the end of a command line will cause BUG/65 to repeat the execution of that command line forever. The only way to stop such a repeat is to suspend or abort the command.

8.3 Display Last Command Line

=

If you want to see what your last command line was, possibly because you might want to repeat it, just type the "=" character as the first character of the new command line. BUG/65 will display the last line entered for you.

SECTION 9: BUG/65 Memory Protection

BUG/65 won't allow you to modify any portion of it's code or variable storage areas with a BUG/65 command. Any attempt to do so will result in a "PROTECTION ERROR". For example, if we assume that the BUG/65 was loaded via the command "BUG65 2000", the following command will cause an error because it attempts to move a memory block into BUG/65's area:

```
M 4000 40FF 2000
```

BUG/65 protects all memory from loadpoint to loadpoint+$1FFF in this manner, where loadpoint is that specified in the invoking OS/A+ command line (or LOMEM, if no loadpoint is specified). (The memory protection feature can be turned off by changing a byte in the Configuration Table.)

SECTION 10: BUG/65 Memory Usage

BUG/65 uses memory from $80 to $XX and loadpoint to loadpoint+$01FF for variable storage. You can determine the value of XX by looking at the LSTPG0 byte in the Configuration Table. It uses memory from loadpoint+$200 to loadpoint+$1FFF for code storage.

10.1 Page Zero Sharing

BUG/65 will share the page zero memory that it needs with a user program. It does this by keeping two copies of these page zero locations. When BUG/65 is running, the BUG/65 page zero locations contain BUG/65's stuff. When a Go is done to a user program, BUG/65 will save it's own page zero data and replace it with the user's data. If a user program breaks back to BUG/65, the reverse operation is performed.

In addition, BUG/65 will translate any command reference to these shared page zero locations so that the user may modify or inspect his own page zero data. It does this by translating any command reference to the user's page zero data to the location where the user's copy of the data is actually being stored. This is all transparent to the user. For example, you can fill memory from $80 to $FF with zeros without crashing BUG/65. If you then display $80 to $FF, you will see zeros. They aren't really in locations $80 to $FF of course, but they will be when you run your program. (This is the reason it may seem to take an extraordinarily long time to perform certain commands (Fills, for example). The reason is that every memory reference has to go through this translation process - both to translate zero page references if necessary and to check to make sure that BUG/65 isn't being overwritten.)

SECTION 11: Customization with the Configuration Table

There is a Configuration Table located near the beginning of the code segment of BUG/65. By changing this data, you can customize some BUG/65 stuff. In the table which follows, "+$xxx" means that the configuration value is located $xxx bytes above the loadpoint address, where loadpoint is the address specified in the invoking OS/A+ command line (or LOMEM, if loadpoint is not specified). Example: if the invoking command was "BUG65 6ØØØ", then DISPV will be located at $62Ø9.

NAME	LOCATION	FUNCTION/COMMENTS
DISPV	+$2Ø9	A JMP instruction to BUG/65's display a character routine. All chars displayed on the screen go through here. The char to be displayed is passed in reg A.
PRINTV	+$2ØC	A JMP instruction to BUG/65's print a character routine. All chars sent to the printer go through here. The char to be printed is passed in reg A.
GETKYV	+$2ØF	A JMP instruction to BUG/65's get a keyboard character routine. All keyboard reads go through here. The key read is returned in reg A.
TSTKYV	+$212	A JMP instruction to BUG/65's test for a key waiting routine. All tests for key waiting go through here. If no key is waiting, the equal flag is returned set. (The key is NOT returned by this routine - GETKYV will be called to read the key if there's one waiting.)
BEEPV	+$215	A JMP instruction to BUG/65's bell routine. All beeps are generated through here. To eliminate the beeps, just patch this out with an RTS.
CHRCLR	+$218	Character background color byte value.
CHRLUM	+$219	Character luminance byte value.
BRDCLR	+$21A	Border color byte value.
EOLBYT	+$21B	This is the byte sent to the printer at the end of a line. Normally set to ØDH or 9BH.

LFFLAG	+$21C	If nonzero, then a linefeed character is sent to the printer after every EOLBYT.
NULFLG	+$21D	If nonzero, then 40 nulls will be sent to the printer after every line. Used to flush the printer buffer maintained by the ATARI OS so that all lines will print immediately.
PROTFG	+$21E	If nonzero, then BUG/65 will not allow itself to be overwritten with a BUG/65 command. If zero, then BUG/65 will allow itself to be modified.
MCBEND	+$21F	High byte of end address of BUG/65's code. Normally set to high byte address of loadpoint+$2000 (e.g, $50 if the invoking OS/A+ command were BUG65 3000). You would change this if you added any user command handlers after BUG/65. The handlers would then be included in BUG/65's memory protection features.

To change anything in the Configuration Table, you must first disable memory protection by writing a small program to stuff a zero into PROTFG. For example, assuming that the loadpoint is $2000 (command line was BUG65 2000), then using the instant assembler, you could enter "LDA #0, STA 221E, RTS" at location $5000, and then run the program with the "U" command by entering "U5000 <RETURN>". This will disable memory protection. Then make your changes, reenable memory protection if you want by storing $FF into PROTFG, then dump the modified BUG/65 to diskette.

Be careful when changing any of the JMP instruction vectors. Since BUG/65 is constantly calling these locations, the instant you change them control will be passed to the new routine. Your replacement routines had better be in place and ready to run or it's ga-ga time. Actually, you will probably have to change all three bytes of a vector at once with a small user program.

Also, be careful about calling the vectors DISPV, PRINTV, GETKYV, TSTKYV, and BEEPV. Since they use BUG/65's page zero data to operate, they can't be called from a running user program without first calling the MCBGP0 routine defined in the User Program Interface section.

SECTION 12: User Command Interface

It's possible to add commands to BUG/65. The hooks to do so have been provided in a group of vectors located at loadpoint+$0220 called the User Command Interface Vectors. These vectors provide most of the interfaces to BUG/65 that you'll need to add commands.

The commands you add may be activated by any non-BUG/65 command char. For example, you could add the numeric commands "1" through "9". When BUG/65 recognizes a non-alphabetic command character, it will call the vector USERCMD. In it's initial state, USRCMD is just a 3-byte subroutine that returns the equal flag reset. BUG/65 assumes that the equal flag being reset means that a user command handler considers the command illegal. In this case, BUG/65 will report a "CMD ERROR". If USRCMD returns the equal flag set, then BUG/65 assumes that a user command handler processed the command. In this case, BUG/65 won't generate a command error, and will proceed to process the rest of the command line.

So, to add your own command handler, just patch a JMP to your handler at USRCMD. BUG/65 will pass you the command character that it considered illegal in reg A. On return, you must indicate the status of the command - equal set means you handled it, equal reset means you didn't like it either.

There are a number of other vectors in the User Interface group which you may use to process the command. Here's the complete list (and, as in the previous section, the string "+$xxx" indicates a displacement from the loadpoint):

NAME	LOCATION	FUNCTION/COMMENTS
USRCMD	+$220	Subroutine called by BUG/65 on every non alpha comand char. Returns equal set if command handled by user, else equal reset.
GETCHR	+$223	User handler can call this to get the next char from the command line in reg A.
PUTCHR	+$226	User handler can call this to return the last char taken from the command line. The char itself doesn't have to be passed. This is used to put chars back that you've taken but don't want - like an EOL.
GET1HX	+$229	User handler can call this to collect a hex address from the command line. The address is returned in a word at $FE,$FF. If next command line chars are not a valid address, zero is returned.
GET2HX	+$22C	User handler can call this to collect two hex addresses from the command line. The first address is returned in a word at $FC,$FD, the second at $FE,$FF. Zero is returned for any invalid address.
GET3HX	+$22F	User handler can call this to collect three hex addresses from the command line. The first address is returned in a word at $FA,$FB, the second at $FC,$FD, and the third at $FE,$FF. Zero is returned for any invalid address.

ADRCHK	+$232	User handler can call this to perform the usual BUG/65 address checking and translation. The checking refers to not allowing BUG/65 to be overwritten. The translation refers to correcting user page zero addresses. The user handler passes the address to check in reg X (LO) and reg Y (HI). If the address points into BUG/65, a "PROT ERROR" will occur, and the user handler will not be returned to. If the address references a user page zero value that is being stored somewhere else by BUG/65, then the address of where the actual user page zero byte is located will be returned in reg X (LO) and reg Y (HI).
ERRPAR	+$235	The user handler can JMP to here to report a parameter error. There is no return back to the user handler. BUG/65 will abort command line processing.
DHXBYT	+$238	The user handler can call this to display a hex byte. The byte is passed in reg A.
DHXWRD	+$23B	The user handler can call this to display a hex word. The hex word is passed in reg X (LO) and reg Y (HI).
CTBPTR	+$23E	This is a pointer to BUG/65's jump table for the alphabetic comands. Every letter has a word entry in this table. The entry is the address of the handler for that command minus one. The first word in the table is the address minus one for the "A" command, the last is the same for the "Z" command. If you want, you can change this table to point to your own comand routines, thereby changing the BUG/65 command set.
LSTPGØ	+$24Ø	This is the address (byte value) of the last page zero location used by BUG/65. You can use this to locate free page zero memory for your own use. (See the example user command listing.).

**** SPECIAL NOTE ****

All of the above routines assume that BUG/65 data is in page zero. THEY WILL NOT WORK if called from a running user program for that reason, unless the user program manages page zero with the following two routines:

MCBGPØ	+$241	Assumes BUG/65 data is in page zero. Saves BUG/65 page zero and replaces with user page zero. Use this routine from a running user program before calling any of the above routines.
USERPØ	+$244	Assumes user data is in page zero. Saves user page zero and restores BUG/65 page zero. Use this routine from a running user program after calling any of the above routines to restore the running program's page zero data.

12.1 User Command Handler Example

Here is an assembly listing of an example user comand. This command will be command "1". It will calculate and display an exclusive-or checksum byte on a range of memory. The syntax of the command is:

```
   1  <START>  <END>
```

NOTE: It is highly recommended that user commands only be patched into a non-relocatable version of BUG/65. See Section 4.2 for instructions on making a non-relocatable version with a user specified loadpoint.

```
;***********************************************
;
;  EQUATES INTO BUG/65:

loadpoint =  ????              to be determined by user!!
lp = loadpoint                 just an abbreviation

MCBEND   =    lp+$21F          BUG/65 END CODE MSB
DISPV    =    lp+$209          DISPLAY CHAR
USRCMD   =    lp+$220          USER COMMAND VECTOR
GET2HX   =    lp+$22C          GET 2 HEX PARAMS
HEX1     =    $FC              HEX PARAM 1 RESULT
HEX2     =    $FE              HEX PARAM 2 RESULT
ERRPAR   =    lp+$235          REPORT PARAM ERROR
DHXBYT   =    lp+$238          DISPLAY HEX BYTE
LSTPGØ   =    lp+$240          LAST BUG/65 PØ BYTE USED
;
EOL      =    $9B              END OF LINE CHAR
```

```
;***********************************************************
            *=      USRCMD            PATCH US INTO BUG/65
            JMP     USERC1
;
            *=      1p+$2000         RIGHT AFTER BUG/65 CODE
USERC1      CMP     #'1              COMMAND "1" ?
            BEQ     CMDOK            YES
            RTS                      ELSE RTN EQUAL RESET - ERR
;
CMDOK       JSR     GET2HX           GET START, END
            LDA     HEX1             MAKE SURE BOTH SPECIFIED
            ORA     HEX1+1
            BEQ     PARMER           OR ELSE ERROR
            LDA     HEX2
            ORA     HEX2+1
            BNE     PARMOK
;
PARMER      JMP     ERRPAR           REPORT PARAM ERROR
;
PARMOK      LDX     LSTPGØ           LAST BUG/65 PØ BYTE
;                                      (WE'LL USE THE NEXT
;                                      FOR OUR ACCUMULATOR)
            LDA     #Ø                CLEAR ACCUMULATOR
            STA     1,X
            TAY                       INIT Y PTR INDEX
;
LOOP        LDA     HEX2+1            PAST END ADDRESS ?
            CMP     HEX1+1
            BCC     DONE              YES
            BNE     NXTEOR            NO
            LDA     HEX2
            CMP     HEX1
            BCC     DONE              YES
;
NXTEOR      LDA     (HEX1),Y          CALC EOR CHKSUM
            EOR     1,X               EOR WITH ACCUM
            STA     1,X               AND SAVE IN ACCUM
            INC     HEX1              BUMP PTR
            BNE     LOOP
            INC     HEX1+1
            JMP     LOOP
;
DONE        LDA     #EOL             TO NEXT SCREEN LINE
            JSR     DISPV
            LDX     LSTPGØ           RESTORE ACCUM ADDRESS
            LDA     1,X              DISPLAY HEX RESULT
            JSR     DHXBYT
            LDA     #Ø               RTN OK (EQUAL SET)
            RTS
;
            *=       MCBEND           CHANGE BUG/65 CODE
            .BYTE    >[*+$FF]         END BYTE TO INCLUDE
            .END                     THAT'S ALL FOLKS
```

SECTION 13: Error Messages

The following is a list of all of the error messages and a short explanation of each one:

COMMAND ERROR...an attempt to execute an illegal command. A letter or number that isn't a valid command mnemonic was interpreted as a command character. For example, trying to execute the command "N" will cause a command error.

IMMEDIATE CMD ERROR...an attempt to execute an immediate type command in the middle of a command line. An immediate command (A, S, X, or Z) must be the first command on a command line. See section 5.2.

PROTECTION ERROR...an attempt was made to modify BUG/65's code or variable memory areas with a BUG/65 command.

PARAM ERROR...caused by the usage of any invalid command parameter.

REGISTER ERROR...an invalid register name was specified in either the G or X command.

BREAKPOINT ERROR...an attempt was made to set a breakpoint in either PROM memory space or non-existent memory.

PRINTER ERROR...any printer error returned to BUG/65 by the operating system. (BUG/65 uses the ATARI OS to print characters. Any error returned by the OS on a print character call will cause this error.)

SYNTAX ERROR...caused by an error in the syntax of a command.

I/O ERROR - NNN...any disk I/O error returned to BUG/65 by the operating system. (BUG/65 uses the OS/A+ to do disk I/O. Any error returned by the OS/A+ call will cause this error.) NNN is the decimal error number returned by the OS. Refer to your OS/A+ manual for the meanings of these numbers.

*** ERROR - MNEMONIC ...in the instant assembler, an invalid opcode mnemonic was entered.

*** ERROR - OPERAND...in the instant assembler, an invalid instruction operand was entered.

*** ERROR - RANGE ...in the instant assembler, a branch out of range was attempted.

*** ERROR - TOO MANY LABEL REFS...in the instant assembler, too many references have been made to an undefined label. BUG/65 2.Ø allows twenty references to undefined labels before it's label buffer overflows.

*** ERROR - UNDEFINED - Ln ...in the instant assembler, a label has been referenced but not defined. "n" is the label number that needs definition.