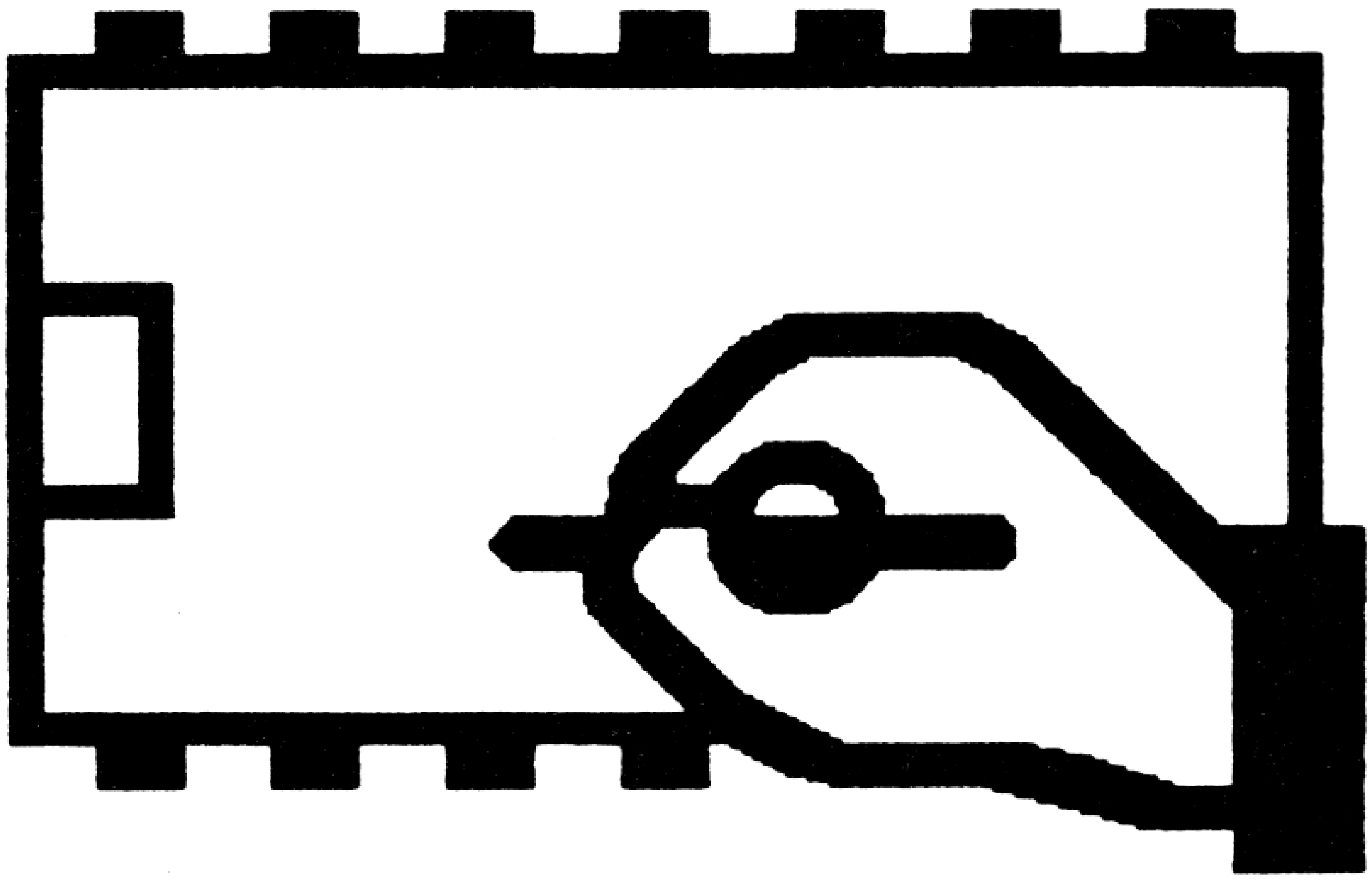

MacASM™

Macro Assembler

for the

Macintosh

MAINSTAY

Yves (805) 373-6218

MacASM

A co-resident editor/macroassembler
for the
Macintosh computer.

User and Reference Manual

Copyright 1984, Mainstay
~~28611B Canwood St~~. 5311-B DERRY AVE.
Agoura Hills, CA 91301
(818)-991-6540

Printed in the U.S.A.

MAINSTAY CUSTOMER LICENSE AGREEMENT

IMPORTANT: The enclosed MacASM program is licensed by Mainstay to customers for their use only according to the terms and conditions set forth below. Opening this package indicates acceptance of these terms.

LICENSE

Mainstay agrees to grant you a non-exclusive license to use the enclosed Mainstay program (the "Program") subject to the terms and conditions of this license agreement.

COPYRIGHT

The Program and its User and Reference Manual are copyrighted. You may not copy or otherwise reproduce any part of the Program and its User and Reference manual except for personal archival copies.

RESTRICTIONS on USE and TRANSFER

The original and any archival copies of the Program and its User and Reference manual are to be used in connection with a single computer. You may physically transfer the Program from one computer to another, provided that the Program is used in connection with only one computer at a time. You may not transfer the Program electronically from one computer to another over a network. You may not distribute copies of the Program or the user and reference manual to other parties. You may not use, copy, modify or transfer the Program in whole or in part, except as expressly provided for in this license agreement.

NO PERFORMANCE WARRANTY

Mainstay can not and does not warrant the performance or results that may be obtained by using the program. Accordingly, the Program and its User and Reference Manual are licensed "as is" without warranty as to their performance, merchantability or fitness for any particular purpose. The entire risk as to the results and performance of the program is assumed by you. Should the Program prove defective, you and not Mainstay (or its dealer) shall assume the entire cost of all necessary service, repair or correction.

LIMITED WARRANTY FOR DISKS

Mainstay warrants, to the original licensee only, the magnetic disk on which the Program is recorded to be free from defects in materials and faulty workmanship under normal use for a period of 180 days from the date of purchase. Mainstay will replace the disk without charge provided that you have returned the enclosed registration card. If the disk has been damaged due to accident, misuse, or misapplication, then Mainstay shall have no responsibility to replace the disk under the terms of this limited warranty. This limited warranty grants you specific legal rights, and you may also have other rights which vary from state to state.

DISK REPLACEMENT POLICY

If after 180 days of purchase, a defect in the disk should occur, the disk may be returned to Mainstay along with $10.00. Mainstay will replace the disk provided that you have previously returned your product registration card to Mainstay.

LIMITATION of LIABILITY

Neither Mainstay nor anyone else who has been involved in the creation, production, or distribution of this program shall be liable for any direct, incidental, or consequential damages, such as, but not limited to, loss of anticipated profits or benefits, resulting from the use of the Program or as the result of a breach of any warranty. Some states do not allow the exclusion or limitation of direct, incidental, or consequential damages, so the above limitation may not apply to you.

TERM

This license is effective until terminated. You may terminate it at any time by destroying the Program and the User and Reference manual, together with all copies, modifications, and merged portions in any form. The license also terminates if you fail to comply with any term or condition of this license agreement. You agree upon such termination to destroy the Program and User and Reference manual with all copies, modifications, and merged portions in any form.

Your use of this program indicates that you have read this customer license agreement and agree to its terms. You further agree that the license agreement is the complete and sole statement of the agreement between us and supersedes any oral or written proposal or prior agreement and any other communications between us concerning the subject of this agreement.

DISK BACK-UP POLICY

This program has not been copy protected in the interest of providing you with a more useful product. The first action you should take is to MAKE A BACK UP COPY OF THE DISK and put it in a safe place. Use the Macintosh Disk Copy utility which is included on the System Disk.

SUPPORT POLICY

Mainstay intends to provide quality support to bona fide licensees of MacASM who respect our license agreement. Although MacASM is straight foreward, we realize that this is not the case for 68000 assembly language programming and the Macintosh interface. Our goal is to provide quality support to help you do whatever you want with the Macintosh and MacASM.

Each copy of MacASM possesses a unique serial number which is displayed on the screen upon command. This number defines your support account and is activated by the receipt by Mainstay of your completed registration card.

ABSOLUTELY NO SUPPORT WILL BE PROVIDED UNLESS WE RECEIVE THE COMPLETED REGISTRATION CARD.

Your purchase of MacASM credits your support account with five (5) support calls. Mainstay will, however, respond to any caller who provides a valid serial number. This means that if you allow others to make copies of MacASM, they may use the support for which you have paid. Your serial number is only known to you; the serial number is auto-verifying and of a form that cannot possibly be guessed.

When your support account shows no support calls left, you will be notified and given the opportunity to purchase additional support. Again, if you allow others to copy MacASM, they will use the support for which you have paid.

Future support will also be offered via an electronic bulletin board.

Program written by Y. Lempereur

Manual by T. Nalevanko

Assembler design consultation
by B. Sander-Cederlof
of S-C Software

TABLE OF CONTENTS

INTRODUCTION

MacASM is an assembly language development system for the Macintosh computer. MacASM features a co-resident editor/macroassembler which allows rapid assembly language program development directly on the Macintosh. In addition, MacASM provides an integrated resource compiler to facilitate independent applications which run under their own icon. MacASM provides a new level of power and performance to Macintosh programmers -- for beginners as well as those who are more experienced.

MacASM will operate on standard 128K and 512K Macintosh computers and also the Lisa computer running under MacWorks. MacASM also supports the ImageWriter printer and hard disk drives. MacASM only occupies 22K of memory allowing fairly large source programs to be edited and assembled entirely within RAM. Much larger source programs, up to the limit of on-line disk storage, can be edited, saved to disk and then assembled using the source file "INCLUDE" capability.

Programs can be edited, assembled, and tested entirely within the framework of MacASM. The editor and assembler are both resident in memory at the same time, allowing rapid cycles of source program modification, re-assembly and check-out. Since this edit-assemble-debug-edit cycle can be performed without leaving the MacASM environment, programmer productivity is dramatically improved.

Design Philosophy

MacASM was designed primarily for ease of use. Simple one-word commands and a full-screen work environment result in a simple but effective interface.

Learning MacASM is also quite easy since it is a single environment which resembles that of other microcomputers with which you are probably already acquainted. As with all worthwhile exercises however, some effort is required. This is particularly true for the editing commands which do not use the Macintosh mouse and associated editing structure. The decision was made to employ a more conventional line based type of editing using a full screen environment to provide a faster and more flexible product. We feel that your investment of time to learn MacASM will be but a small fraction of the time MacASM saves you by its fast, clean environment. However, should you prefer editing under mouse control, please note that this can be done by using MacWrite, subject to the MacWrite memory limitation.

Getting to know MacASM

The easiest way to get to know MacASM is to try it out. Follow the instructions to load, assemble, execute, modify and re-execute an example program which affects the normal Macintosh screen presentation.

1. Turn your Macintosh "on" and wait for the disk icon with the "question mark" to appear.

2. Insert the MacASM disk and wait for the icon screen to appear.

3. Double click the icon for the "Examples" folder. Then wait a moment and double click "Exampl.Asm". MacASM will be automatically loaded along with the example source program.

4. Type "LIS" "RETURN" to view the source program example.

5. Type "ASM" "RETURN" to assemble the example program.

6. Type "RUN START" "RETURN" to execute the program. Note the screen changes and wait for the cursor to appear. You have now assembled and executed a program using MacASM.

7. Type "LIS" "RETURN" to review the source program.

8. Now you'll modify the program. Move the cursor to line 00100 by holding down the "COMMAND" key and pressing the "K" key successively. If you go too far, use "COMMAND" and "L" to go down. Next, while holding down "COMMAND", press the ">" key successively to move the cursor to "FFFFFFFF". If you go too far, use "COMMAND" and "<" to go left. Type a "0" (zero) for every other "F", like this "0F0F0F0F", and press "RETURN".

9. While holding down "COMMAND", press the "0" (zero) key to clear the screen.

10. Redo steps 4 thru 6 of this demo to view your modified program.

If all went well you will have seen the screen flash when the original program was "RUN" and have vertical bands of grey when the modified version was "RUN". If this did not happen, type "EJE1" "RETURN" to eject the disk and start over from the beginning, paying more careful attention to the directions.

The example program is listed here with some comments to make it easier to understand.

```
00010 ;SAVE"Exampl.Asm"
00020 *-----------------------------------------
00030 SCREEN EQU    $07A700      Define Screen
00040 *-----------------------------------------
00050        ORG    $15000       Program Origin
00060 *-----------------------------------------
00070 START  MOVE.W #100-1,D2      Invert Screen
00080 .0     LEA    SCREEN,A0      100 Times
00090        MOVE.W #5472-1,D1
00100        MOVE.L #$FFFFFFFF,D0
00110 .1     EOR.L  D0,(A0)+       Invert Each Pixel
00120        DBRA   D1,.1          All Pixels Done?
00130        DBRA   D2,.0          Done 100 Times?
00140        RTS
00150 *-----------------------------------------
00160        END
```

Now that you have "hands-on" experience with MacASM you'll want to read the manual further, trying out MacASM features as you go along.

Loading MacASM

MacASM is loaded into Macintosh memory by inserting the disk, waiting for the icon page to appear and double clicking MacASM's characteristic "chip" icon. If you wish to load a source file that was generated by MacASM, simply double click the file icon. MacASM will automatically be loaded along with the source file.

Assembly Language

Assembly language allows you to interact directly with all facilities of the Macintosh. The primary advantages are that carefully written programs will be very fast and efficient.

The program that the Macintosh will execute however must be a machine language program. Machine language is entirely numeric and not too easily understood by humans. Assembly language programs are symbolic and must be converted to machine language. This is done by the MacASM assembler which processes your source program or code into the object program or code. The object code, when loaded into the computer's memory and started, will be executed directly by the 68000 microprocessor.

The assembly language source program is either entered from the keyboard or loaded from a previously created source file that has been stored on disk.

The source program consists of source statements, one to a line. Each line of the source program generally produces one machine language instruction, although in certain cases, only data is generated or some assembler control action takes place. This is explained in more detail in the section on Source Programs.

Full Screen Environment

MacASM commands, source program entries and their modifications are facilitated by a full screen work environment. This allows you to create or make changes to the source program or to issue or reissue MacASM commands anywhere on the screen. This is unlike line editors which limit you to calling up and changing one line at a time, as in standard BASICs. MacASM also employs a safety feature which prevents inadvertent and unwanted changes in your program. Although you or the computer may make changes to one or more lines on the screen, these changes are only entered in memory if the "RETURN" or "ENTER" key is pressed while the cursor is somewhere in a line which was modified.

The MacASM full screen environment operates in a character replace mode. Any character that you type will replace the character currently under the cursor.

The screen display should be considered a "snapshot" of your program as it existed when the source statements were displayed on the screen. Your source program may be changed by modifications to source statement lines made and re-entered on the screen after their initial display. You may also make changes to the program with commands which do not automatically update the screen; an example is the RENUMBER command. For this reason, it is a good idea to clear the screen and redisplay the relevant part(s) of your program after having issued commands. Otherwise "what you see on the screen" may not be "what you have" in memory.

The screen display and the cursor positioning are controlled with the following keys:

"RETURN" or "ENTER"

Enters or re-enters the line where the cursor is currently positioned. **** NOTE **** "RETURN" or "ENTER" MUST BE PRESSED TO ENTER ANY MODIFICATIONS MADE TO AN EXISTING LINE.

"COMMAND" and "0" (zero)

Clears the screen.

"COMMAND" and "<"

Moves the cursor one space to the left.

"COMMAND" and ">"

Moves the cursor one space to the right.

"COMMAND" and "K"

Moves the cursor one line up.

"COMMAND" and "L"

Moves the cursor one line down.

"COMMAND" and "+"

Inserts a space at the current position of the cursor.

"COMMAND" and "-"

Deletes the character at the current position of the cursor.

"COMMAND" and "SHIFT" and "+"

Inserts a blank line at the current position of the cursor. This blank line exists only on the screen and not in the source code. A line number and at least one non-blank character must be entered for a line to exist in the source code.

"COMMAND" and "SHIFT" and "-"

Deletes the line at the current position of the cursor. This line is only deleted from the screen and not in the source code. Line number n may be deleted by typing DEL n or by typing the line number n followed by "RETURN" or "ENTER".

"COMMAND" and "*"

Creates a comment separator line of dashes useful for separating sections of source code. This function works only when the cursor is in the first column of the label field; typically when the line number has been generated by the TAB key. If you type the line number yourself you must type the entire line number, i.e. five digits, followed by a blank space.

"TAB"

Displays the next line number when the cursor is in the first column of the screen; otherwise the cursor will be moved to the next tab stop.

The Macintosh keypad is supported for cursor controls using the arrow keys. The keypad "CLEAR" key will clear the screen.

SOURCE PROGRAMS

A source program is a logical sequence of assembly language statements designed to perform a certain task. Source programs are assembled by MacASM to produce object code which can be executed directly by the Macintosh's 68000 microprocessor.

A source statement must at least contain a label, an assembly language instruction mnemonic, an assembler directive or a comment. Source programs are entered one line at a time, with a line number identifying each line. The line numbers may range from 00000 to 65535. Source programs are kept sorted in line number order with the numbers used for editing purposes, just as in BASIC.

A blank space must always follow the line number. After the blank there may be up to four fields of information. These are: the LABEL, the instruction MNEMONIC or directive, the OPERAND, and COMMENT fields. Adjacent fields must be separated by at least one blank. Lines may be up to 80 characters long and may include both upper and lower case characters.

A sample source statement line follows:

```
line   label    mnemonic  operand        comment
 no.

00010 MAINPROG   BSR       GO        THIS STARTS IT ALL!
```

LINE NUMBERING

You may either type a line number or have it semi-automatically generated for you by MacASM.

If you type the number yourself, you should type a five digit number, e.g. 00010. Otherwise your source presentation will shift with larger line numbers as MacASM will always display the source with a five digit line number.

MacASM also includes a semi-automatic line number generator. If you press the TAB key while the cursor is at the left edge of the screen, the next line number will be generated. If the cursor is not at the beginning of a line, the TAB key operates normally.

The "next line number" is always the value of the last line number which was entered by you plus the current line number "increment". The standard increment is 10 but it can be changed to any reasonable value with the INCREMENT command. After inserting or editing lines in your program, it is a good practice to move the cursor to the line with the highest number and press "ENTER". This will now be the previous line entered when you press "TAB" and will cause the expected continuing line number to be automatically generated.

BUILT-IN TAB STOPS:

The standard tab stops allow label and instruction mnemonic fields of eight characters each. An array of additional tab stops are given for the operand and comment fields. Tab stops are in columns 6, 15, 23, 28, 39, 47, 55, 63 and 71, where the first column is designated as column "zero".

LABEL FIELD:

The label field may be left blank or may contain a label. Three types of labels can be used. These are: normal labels, local labels and private labels. The first character of the label must be in the second column after the line number as in the following examples.

```
00010 HERE.WE.GO     (normal label)
01050 .23            (local label)
00010 :12            (private label)
```

NORMAL LABELS: Are used to name places in your program to which branches are made, as well as to name constants and variables. Normal labels may be up to 32 characters long. If more characters are used, these are ignored by MacASM during assembly and exist only in the source code presentation. The first character of a normal label must be a letter. Subsequent characters may be letters, digits, the period or the underline character. The period or underline character are both useful for making long labels readable. For example a subroutine to extract the next character from a buffer might be named:

```
01060 GET.NEXT.CHARACTER.FROM.BUFFER
```

The standard tab stops assume your labels to be eight or less characters long. However, since the assembler allows a relatively free format, you may type a label of any length followed by a blank and the opcode, operand and comment fields.

You may also type a long label on a line all by itself. In this form the label is assigned the current value of the program location counter. If your source line contains only a normal label, it cannot have a comment.

An example with normal labels follows:

```
00010 *  SAMPLE PROGRAM WITH NORMAL LABELS
00020 *
00030 SOURCE.LINE.POINTER EQU $10000   NORMAL LABEL
00040 CHAR.POINTER        EQU $10004   NORMAL LABEL
00050 * THE FOLLOWING LINE IS A NORMAL LABEL.
00060 READ.NEXT.CHARACTER.FROM.LINE
00070                     MOVE.L CHAR.POINTER,A0
00080                     MOVE.B (A0)+,D0
00090                     MOVE.L A0,CHAR.POINTER
00100                     RTS
```

LOCAL LABELS: Are used to name branch points within a section of program between two normal labels or between a normal label and the end of the source program. The main purpose of local labels is to make programs more readable by reducing the number of label names that must be invented. The use of local labels also encourages structured programming habits.

Local labels have a period as the first character, followed by one to four digits. Any label from ".0" through ".9999" may be used. Please note that the local label names are integer numbers, not decimal fractions. The period only acts to designate a local label. Consequently, the label ".1" is treated as exactly equivalent to the label ".0001".

A local label is defined, internal to MacASM, relative to the normal label which precedes it in the source program. Use of a local label without a preceding normal label will result in a "No normal label error.".

Since each set of local labels is associated with a particular normal label, you may re-use the same local labels as often as you wish -- as long as they are associated with different normal labels.

Here is an example of three short routines in the same source program using normal and local labels:

```
00010 *-----------------------------------
00020 PRINT.MESSAGE
00030 .0       MOVE.B    (A0)+,D0     LOCAL LABEL HERE
00040          BEQ.S     .1
00050          BSR       PRINT.CHARACTER
00060          BRA.S     .0
00070 .1       RTS                    LOCAL LABEL HERE
00080 *-----------------------------------
00090 GET.NEXT.CHAR
```

```
00100          MOVE.L    CHAR.POINTER,A0
00110          MOVE.B    (A0),D0
00120          CMP.B     #RETURN,D0
00130          BEQ.S     .0
00140          ADDQ.L    #1,CHAR.POINTER
00150 .0       RTS                   LOCAL LABEL HERE
00160 *-----------------------------------
00170 GET.NEXT.NONBLANK.CHAR
00180 .0       BSR       GET.NEXT.CHAR  LOCAL LABEL HERE
00190          CMP.B     #' ',D0
00200          BEQ.S     .0
00210          RTS
00220 *-----------------------------------
```

Private Labels: Are used within macros as branch points. These are discussed in detail in the section on MACROS.

MNEMONIC FIELD:

The mnemonic field (sometimes called op-code) contains a 68000 instruction mnemonic, a macro name, or an assembler directive (sometimes called a pseudo-op). If you are using the tab stops, the mnemonic field normally starts in column 15. However, instruction mnemonics may begin in any column after at least one space from a label or at least two spaces from a line number.

MacASM uses the standard 68000 instruction mnemonics as defined by Motorola Microsystems. The 68000 mnemonics, assembler directives and macros are discussed later in this manual.

OPERAND FIELD:

The operand field usually contains an expression. Some of the 68000 instructions such as NOP, RTS, and RTE have no written operand expression. In these cases a comment may directly follow the mnemonic with only a blank character as a separator.

COMMENT FIELD:

Comments are separated from the mnemonic or operand field by at least one blank.

COMMENT LINES:

Full lines of comments may be entered by typing an asterisk (*) or a semi-colon (;) in the first column of the label field. This type of comment is useful in separating various routines from each other and labeling their contents. It is analogous to the REM statement in BASIC. A comment separator line of dashes can be automatically created by typing "COMMAND" and "*" when the cursor is in the first column of the label field.

SOURCE PROGRAM LOCATION

The Macintosh employs dynamic memory allocation. Programs are placed in memory where they best fit. For this reason it is not possible to give a memory map of the location of MacASM and the beginning of the source file. These locations may change as a function of the type and state of your Macintosh when MacASM was loaded.

The Macintosh DOS, buffers and screens take up about 59K of RAM in a 128K Macintosh. MacASM uses approximately 22K. This leaves approximately 47K for source code, the symbol table and "Free Memory" which may be used to execute your program. The 512K Macintosh will have approximately 410K available for source code, the symbol table and "Free Memory".

EXPRESSIONS

MacASM offers a sophisticated expression evaluation capability. Expressions are used as the operand of an instruction mnemonic or a directive or as the argument of a command. An expression may be as simple as a number or have a complexity limited only by one's needs, creativity and courage.

Expressions may be used to automatically generate addresses and other data such as table or string lengths. Expressions are written using elements, algebraic and logical operators and parentheses. Elements may be decimal, hexadecimal, octal or binary numbers, labels, literal ASCII characters or the program location counter (PC). Operators are algebraic or logical and are given later in a table.

The expression result may be specified as one or two bytes by immediately preceding the expression with a "#" or a "/" character respectively. The default length is four bytes. If your expression evalutes to a value larger than your specification, the high order bytes will be truncated. The expression result length in bytes is specified by:

```
one byte    (lowest order) : #expression

two bytes   (lowest order) : /expression

four bytes                 : expression
```

ELEMENTS

-Decimal Numbers: Any integer number in the range from -2147483648 to 2147483647. The number must not contain commas. For example:

```
                    00010           ORG     $10000
                    00020 ;
00010000: 103C 00C8 00030           MOVE.B  #200,D0
00010004: 323C FFF6 00040           MOVE.W  #-10,D1
00010008: 8B6B      00050           DATA    /35691
FFFFFFFF:           00060 FLAG      EQU     -1


    --- Symbol Table ---

FFFFFFFF: FLAG

0000 Errors in Assembly
```

-Hexadecimal Numbers: Any number in the range from $0 through $FFFFFFFF. Hexadecimal numbers are indicated by a preceding dollar sign, and may have from one to eight digits. For example:

```
                         00010                ORG        $10000
                         00020 ;
00010000: 103C 002F 00030                MOVE.B     #$2F,D0
00010004: 13C0 0000
00010008: FFFF      00040                MOVE.B     D0,$FFFF
0001000A: 6633      00050                BNE.S      $1003F
0001000C: 4EB9 0000
00010010: E02A      00060                JSR        $E02A
000000AB:           00070 VALL           EQU        $AB
00001278:           00080 NUM            EQU        $1278
00001278:           00090 DATA           EQU        $1278

     --- Symbol Table ---

00001278: DATA
00001278: NUM
000000AB: VALL

0000 Errors in Assembly
```

If you leave out the dollar sign, the assembler will consider your hexadecimal number as decimal.

-Octal Numbers: Any number in the range from 0 to 37777777777. Octal numbers are preceded by an "@" symbol and may have from one to 11 digits. For example:

```
VAL @147
$00000067      103
```

-Binary Numbers: Any number in the range from 0 to 1111111111111111111111111111111 (32 bits). Binary numbers must be preceded by a per-cent sign "%" and may have from one to 32 digits. The syntax is %10101010111 (up to 32 bits or numbers).

For example:

```
                         00010  ORG             $10000
                         00020 ;
00010000: 103C 00C6 00030  MOVE.B          #%11000110,D0
00010004: 0200 0060 00040  ANDI.B          #%01100000,D0
00010008: 8001      00050  DATA            /%1000000000000001
0001000A: 1234      00060  DATA            /%0001001000110100

     --- Symbol Table ---

0000 Errors in Assembly
```

-Labels: There are three types of labels in MacASM. Normal labels are from 1 to 32 characters long. The first character must be a letter; following characters may be letters, digits, the underline character or periods. Local labels are written as a period followed by one to four digits and follow an associated normal label. Private labels are written as a colon followed by one or two digits.

Labels must be defined prior to their use in an expression. For example, labels used in operand expressions with ORG, TADDR, DEFS and EQU directives must be previously defined. Labels are defined by being written in the label field of a valid instruction or directive line.

-Literal ASCII Characters: Literal characters are written as an apostrophe followed by the character. The value is the ASCII code of the character (a value from $00 through $7F).

```
                     00010          ORG      $10000
                     00020 ;
00000041:            00030 LETTERA  EQU      'A'
00010000: 5800 0000
00010004: 4143       00040          DATA     #'X','A',#'C'
00010006: 0C00 00DA  00050          CMP.B    #"Z",D0


    --- Symbol Table ---

0000041: LETTERA
```

ASCII literals with the high-bit set are signified with the quotation mark: "A" generates $000000C1. Note that a trailing quotation mark is optional just as is a trailing apostrophe with previously given ASCII literals.

-Asterisk(*): Stands for the value of the program location counter. This is useful for storing the length of a string as a constant in a program. It can also be used to fill memory to the end of a page to assure that the following code begins at an even page boundary.

For example:

```
                        00010          ORG      $10000
                        00020 ;
00010000: 0B            00030 QT       DATA     #QTSIZE
00010001: 414E 5920
00010005: 4D45 5353
00010009: 4147 45       00040          ASC      /ANY MESSAGE/
0000000B:               00050 QTSIZE   EQU      *-QT-1
0001000C: 0000 0000 00060 VAR      DATA     *-*
00010010:               00070 FILLER   DEFS     $10100-*
00010100: 00            00080          DATA     #0


    --- Symbol Table ---

00010010: FILLER
00010000: QT
0000000B: QTSIZE
0001000C: VAR

0000 Errors in Assembly
```

Note that QTSIZE is 11 ($0B), the length of the "ANY MESSAGE" string.

OPERATORS

Operators' notation, description, and precedence level are given in the following table.

OPERATOR NOTATION	OPERATION DESCRIPTION	PRECEDENCE LEVEL
)	right parenthesis	highest - 8
.NOT.	NOT (1's complement)	7
& or .AND.	logical AND	6
.XOR. or .EOR.	logical exclusive OR	5
! or .OR.	logical OR	5
.MOD.	modulo	4
/ or .DIV.	division	4
* or .MULT.	multiplication	4
- or .SUB.	subtraction	3
+ or .ADD.	addition	3
> or .GT.	greater than	2
< or .LT.	less than	2
<>, ><, or .NEQ.	not equals	2
=, or .EQ.	equals	2
=>, >=, or .GE.	greater than or equal	2
<=, =<, or .LE.	less than or equal	2
(	left parenthesis	lowest - 1

Expressions within parentheses are always evaluated first. Next, operators with the highest precedence level are evaluated. Operators with the same precedence level are evaluated from left to right. Do not include spaces in your expressions.

Expressions are used with the BSAVE, RUN, SETLABEL and VALUE commands. These are explained in the section on commands.

The VALUE command allows you to evaluate your expressions in "real-time". This is an excellent tool for familiarizing yourself with expressions as well as for general programming.

Some examples follow:

```
Ok.
VALUE 10+5*4
$0000001E       30

Ok.
VALUE (10+5)*4
$0000003C       60

Ok.
VAL .NOT.-1
$00000000       0

Ok.
```

COMMANDS

MacASM is ready to work for you whenever "Ok." or the cursor are displayed. You directly control MacASM by typing simple, one-word commands.

Commands may be typed whenever the cursor is at the left edge of the screen (column 0). The command will be executed whenever you press the "RETURN" or "ENTER" key after typing the command. Any commands not starting in column 0 will be ignored. The full screen environment applies to commands also. For example, if you previously typed a command and you wish to use it again, it is not necessary to retype the command. Simply move the cursor anywhere in the line containing the command and press the "RETURN" or "ENTER" key; the command will then be re-executed.

All MacASM commands may be abbreviated to their first three letters. However, if you type additional letters, these will also be checked for correct spelling.

MacASM commands can be conveniently grouped into source, editing, object, and DOS commands. Source commands are used to manipulate entire source files. Editing commands are used to modify and manipulate lines or groups of lines of source text. Object commands allow you to specify labels, check memory and generate, store and retrieve object code. DOS commands allow you to do a directory of a disk, eject a disk, determine the disk serial number or quit MacASM and return to Macintosh Finder control. MacASM commands are given in the following table where complementary commands are included within parentheses.

GROUP	COMMANDS
SOURCE	- (APPEND, SAVE and LOAD), NEW, (TYPE and ENTER)
EDITING	- COPY, DELETE, FIND, INCREMENT, LIST, MOVE RENUMBER, REPLACE
OBJECT	- ASM, (BLOAD and BSAVE), MEMORY, RUN, SETLABEL SYMBOLS, VALUE
DOS	- DIRECTORY, DOS, EJECT, SERIAL, QUIT

SOURCE COMMANDS

Source commands act on the source program ensemble. They may be used to append a source program file on disk to the source program in memory, save and load a source program to/from disk, erase the current source program from memory, or merge a source program from disk with a source program in memory. A description of each source command follows:

(APPEND, SAVE and LOAD),

APPEND

Appends a source file on disk to the source program presently in memory. Please note that the file on disk which is to be appended must have been stored on the disk using the SAVE command. Also, APPEND does not modify the line numbers or text of either source programs and thus should be followed by a RENUMBER command.

The format for the APPEND command is:

```
APPEND"diskname:filename.Asm"
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Asm is suggested but not required.

Example:

```
APPEND"MYDISK:TEST.Asm"
```

This appends the file "TEST.Asm" from the disk named "MYDISK" to the source program currently in memory.

SAVE

Stores your source file on disk using a MacASM unique compressed format.

The format for the SAVE command is:

```
SAVE"diskname:filename.Asm"
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Asm is suggested but not required.

Example:

```
SAVE"MYDISK:TEMP.Asm"
```

This saves the source code in memory to the disk named "MYDISK" under the "TEMP.Asm" name.

A "time-saver" and safety feature is to employ the SAVE (or TYPE) command within a comment in the first line of your program. For example:

```
00010 ; SAVE"MYDISK:TEMP.Asm"
```

To save the program, type "LIS10" "RETURN" to list the line. Delete the line number, spaces and semi-colon and space characters so that that SAVE starts in column zero. Press "RETURN" to execute the SAVE command. This method has the added advantage that you will never save a source program to the wrong file.

MacASM stores the source program in memory in a compressed format. Blank spaces and repetitive characters are compressed to conserve memory. These characters are only expanded on the screen display.

When the source program is stored on the disk with SAVE, the compressed format is used. Files which are intended to be APPENDed to source code in memory must be stored on the disk using the SAVE command since the APPEND command expects a compressed format.

LOAD

Retrieves source files which were saved on disk with the SAVE command.

The format for the LOAD command is:

```
LOAD"diskname:filename.Asm"
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Asm is suggested but not required.

Example:

```
LOAD"MYDISK:PROGRAM1.Asm"
```

NEW

Deletes the current source program from memory and restarts the assembler. The screen is cleared, MacASM, the version number and the copyright notice are displayed on the the screen, "Ok." and the cursor appear, indicating that MacASM is ready for your command.

(TYPE and ENTER)

TYPE

Stores your source file on disk using a standard ASCII format as if the program were being typed on the keyboard and transfered directly to disk. Files saved with the TYPE command can be edited with MacWrite, subject to MacWrite memory limitations. Additional information on MacWrite compatibility is given in Appendix A.

The format for the TYPE command is:

```
TYPE"diskname:filename.Txt"#
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Txt is suggested but not required. The "#" symbol is optional and indicates that source text line numbers will not be stored on the disk. This economizes space in MacWrite.

Example:

```
TYPE"MYDISK:MYFILE.Txt"#
```

This saves the full ASCII source under the MYFILE.Txt name to the disk named MYDISK without the source text line numbers.

ENTER

Loads a source program file from disk which has been saved with the TYPE command. This command enters the program from the disk on a line by line basis, as if the program were being entered via the keyboard. The effect is that the program is line merged with the current source program in memory. In the event of common line numbers, the file being ENTERed has precedence and will overwrite those lines currently in memory.

If you have created or modified a source file with MacWrite, the file can be loaded into MacASM with the ENTER command. Additional information on MacWrite compatibility is given in Appendix A.

The format for the ENTER command is:

```
ENTER"diskname:filename.Txt"
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Txt is suggested but not required.

Example:

```
ENTER"MYDISK:MYFILE.Txt"
```

This enters the source file named MYFILE.Txt into memory from the disk named MYDISK.

EDITING COMMANDS

Editing commands are used to modify and manipulate lines or groups of lines of source text. These commands may employ line range, string and option parameters to make their operation more selective.

-Line Range Parameters: Are used to operate in a limited area of the source program. A range parameter may be written with one or two line numbers separated by a comma. If there is only one line number, it can stand alone or with a preceding or following comma. Each of these arrangements has the following meaning:

(no number) - specifies the entire source program.

, - specifies the entire source program.

n - specifies line number n.

,n - specifies lines from the beginning of the source program through line number n.

n, - specifies lines from line number n through the end of the source program.

n1,n2 - specifies lines from n1 through n2.

In addition, symbols for the start "#" and end "*" of your program can be used instead of the actual line numbers.

A specified number of lines can also be defined starting at line n by using the "!" operator followed by the number of lines desired.

A forward offset can also be applied to a line number n by using the "+" operator followed by the offset, in terms of number of lines. A backward offset can be applied in the same manner by using the "-" operator.

Examples that clarify the use of these operators are given in the sections which explain the COPY, DELETE, FIND and LIST commands.

-String Parameters: Are used with the LIST, FIND, and REPLACE commands to operate only on lines containing a given character string. The search string is of the form dstringd where "d" is a delimiter of your choice. The delimiter can be any printing character that does not occur in your search string, except for the comma ",", period ".", the pound "#", the asterisk "*" or a digit (0-9). You can use a WILDCARD feature in search strings to operate on all lines containing partial matches with your search string. The wildcard feature is selected by using a single question mark "?" for each occurence. Examples are given in the section which explains the LIST and FIND commands.

-Options: May be selected by appending the letters "A","U" or "P" to the command line. The letter "A" on the end of the command line causes automatic operation of the REPLACE command without any prompting. The letter "U" will cause any differences between upper and lower case letters to be ignored in any of the commands. The letter "P" will cause the result of any of the commands to also be printed on a properly connected Imagewriter printer.

An explanation of each editing command follows:

COPY

COPY range, target

Copies a range of lines from one place in the program to another. A copy of all the lines in the specified range is placed just before the target line.

If the target line does not exist, the range will be copied where the target line would have been. A range of code cannot be copied into itself; if attempted, a "Range error." message will be displayed.

COPY does not delete the original section nor renumber the copy. This gives you a chance to look at what you have done -- before it is too late. You should follow the COPY command by the RENUMBER command as in the following example.

```
LIST
01000 *  COPY EXAMPLE
01005 SAMPLE MOVEQ #5,D2
01006        NOP
01010        RTS

COPY 1005,1006,9999
LIST
01000 *  COPY EXAMPLE
01005 SAMPLE MOVEQ #5,D2
01006        NOP
01010        RTS
01005 SAMPLE MOVEQ #5,D2
01006        NOP

RENUMBER
LIS
```

```
00010 *  COPY EXAMPLE
00020 SAMPLE MOVEQ #5,D2
00030        NOP
00040        RTS
00050 SAMPLE MOVEQ #5,D2
00060        NOP
```

```
COPY 30,50,20
REN
LIST
```

```
00010 *  COPY EXAMPLE
00020        NOP
00030        RTS
00040 SAMPLE MOVEQ #5,D2
00050 SAMPLE MOVEQ #5,D2
00060        NOP
00070        RTS
00080 SAMPLE MOVEQ #5,D2
00090        NOP
```

DELETE

DELETE range

Deletes a line or range of lines from the source program. Another way to delete a single line is to type its line number followed immediately by "RETURN" or "ENTER".

DELETE must be followed by a range parameter and cannot have a search string parameter.

```
DEL            (doesn't work -- for safety's sake)
*** Syntax error.

DEL,           (deletes all lines)

DEL 1230       (deletes only line 1230)

DEL 1230,2890  (deletes lines 1230 through 2890)

DEL 1230,      (deletes all lines from 1230 through end)

DEL ,1230      (deletes all lines from beginning through
                1230)
```

FIND and LIST

```
FIND
FIND range
FIND dstringd
FIND dstringd range
FIND dstringd range options

LIST
LIST range
LIST dstringd
LIST dstringd range
LIST dstringd range options
```

LIST is a synonym for FIND. Many users find it more natural to use LIST with line number ranges and FIND with a search string, but either command will work with either parameter (or all parameters!).

Both FIND and LIST list a single line, a range of lines, or an entire source program. If you specify a search string, only those lines which match the string will be listed. The search string length must be less than 40 characters long.

While a program or range of lines is listing, you can momentarily pause the listing by pressing the space bar. Pressing the space bar again will restart the listing. You can abort the listing by pressing the "RETURN" or ENTER" keys.

Here are some specific examples using range and string parameters that assume a program starting at line 10 with a line spacing of 10.

LIST	(lists entire program)
LIST 1230	(lists only line 1230)
LIS 1230,2890	(lists line 1230 through 2890)
LIST 1230,	(lists all lines from 1230 through the program end)
FIND /ASCII/	(finds all lines containing the string "ASCII")
FIND "BI",1200	(finds all lines up through 1200 that contain the string "ASCII".)
FIND ,2000	(finds all lines from the beginning of the program through line 2000)

```
LIST #,*            (lists all lines)

LIST #!5            (lists first five lines of the program)

LIST 20!3           (lists three lines starting with line 20)

LIST #+1            (lists second line of the program)

LIST 10+1           (lists line 20 of program)

LIST 20-1           (lists line 10 of program)

LIST/COUT/          (lists all lines containing the string
                       "COUT")

FIN"MAIN" P         (finds all lines containing the string
                     "MAIN" and also lists these to the
                     printer)

LIST P              (lists all lines to printer)


FIND/AS?TA/         (this demonstrates the wildcard feature)

001100           BSR AS.DATA
001120           BSR AS.DATA
001200           JMP BASKETA
```

INCREMENT

INCREMENT n

Sets the increment "n" used between line numbers for semi-automatic line number generation with the TAB key. The increment value is normally 10, but it may be set to any value between 1 and 9999.

Example

```
00010 ; FIRST PROGRAM LINE
TAB                 (Tab key pressed)
00020 START         (00020 was generated by MacASM)
                    (You typed the label START
                     followed by "RETURN")

INC 50
TAB                 (Tab key pressed)
00070
```

Note that INCREMENT does not affect the RENUMBER command line spacing.

MOVE

MOVE range, target

Moves a range of lines from one place in the program to another. The lines in the specified range are placed just before the target line.

If the target line does not exist, the range will be moved where the target line would have been. A range cannot be moved into itself; if attempted, a "Range error." will result.

MOVE deletes the original section but does not renumber the copy. This gives you a chance to look at what you have done -- before it is too late. You should follow the MOVE command by the RENUMBER command as in the following example.

```
00010 LAB1 =     1
00020 LAB2 =     2
00030 LAB3 =     3
00040 LAB4 =     4
MOVE 20,30,50

Ok.
LIST

00010 LAB1 =     1
00040 LAB4 =     4
00020 LAB2 =     2
00030 LAB3 =     3
Ok.
REN

Ok.
LIST

00010 LAB1 =     1
00020 LAB4 =     4
00030 LAB2 =     2
00040 LAB3 =     3
Ok.
```

RENUMBER

```
RENUMBER
RENUMBER base
RENUMBER base, spacing
RENUMBER base, spacing, start
```

Renumbers all or part of the lines in your source program with the specified starting line number and spacing. Note that RENUMBER only affects the source code in memory and does not automatically update the source code on the screen. It is a good habit to precede and follow a RENUMBER by CLEAR and/or LIST.

There are three optional parameters for specifying the renumbered lines. These are the line number assigned to the first renumbered line (called the base), the spacing, and the place in your program to begin renumbering (called the start). The default values for the base and spacing are both 10. The default value for the start is 0. There are four possible forms of the command:

REN	Renumbers the entire source program: BASE=00010, SPACING=10, START=0
REN n	Renumbers the entire source program: BASE=n, SPACING=10, START=0
REN n1,n2	Renumbers the entire source program: BASE=n1, SPACING=n2, START=0
REN n1,n2,n3	Renumbers from line n3 to the end of the source program: BASE=n1, SPACING=n2, START=n3

Note that the SPACING and the argument of the INCREMENT command are entirely independent.

The last form is useful for opening up a "hole" in the line numbers for entering a new section of code.

For example:

```
LIST

001000 *  RENUMBER EXAMPLE
001005 SAMPLE MOVEQ #5,D2
001006        NOP
001010        RTS

RENUMBER
```

```
LIST

00010 *  RENUMBER EXAMPLE
00020 SAMPLE MOVEQ #5,D2
00030        NOP
00040        RTS

REN 100
LIST
00100 *  RENUMBER EXAMPLE
00110 SAMPLE MOVEQ #5,D2
00120        NOP
00130        RTS

RENUMBER 2000,4
LIS
02000 *  RENUMBER EXAMPLE
02004 SAMPLE MOVEQ #5,D2
02008        NOP
02012        RTS

RENUMBER 3000,10,2008
LIST
02000 *  RENUMBER EXAMPLE
02004 SAMPLE MOVEQ #5,D2    Note "hole".
03000        NOP
03010        RTS
```

REPLACE

```
REPLACE dstringldstring2d
REPLACE dstringldstring2d range
REPLACE dstringldstring2d options
REPLACE dstringldstring2d range options
```

These commands search for the first indicated character string, called the search string and replace it with the second indicated character string, called the replace string. REPLACE operates on all fields, from the first character in the label field through the end of each line. It can be global, i.e. search the entire program, or it can be made local by using line range parameters to restrict those lines that are searched. The search string and the replace string must each be less than 40 characters. In addition, the line length after the replace action must not exceed 80 characters -- the maximum length of a source program line.

When REPLACE finds your search string in a line, that line will be printed, with the matching string shown in inverse video. The program will then ask "Replace?", and wait for you to type "Y", "N" or some other character. If you type "Y", the corrected line will be listed, then the search will continue. If you type "N" the process will continue. If you type some other character, REPLACE will terminate.

If you append the "A" option, the REPLACE actions will proceed automatically without any prompting.

It is possible to replace more than one matching string in the same source line.

An example using REPLACE follows:

```
REP/CONT/GO.ON/  (change a label name from CONT to GO.ON)

001130        BSR CONT (The underlined characters
REPLACE? Y              appear in inverse video.)
001130        BSR GO.ON
001210 * NOW WE CAN CONTINUE
REPLACE? N
001360 CONT   BSR WORDIN
REPLACE? Y
001360 GO.ON  BSR WORDIN
```

You may use the WILDCARD feature, a single question mark for each occurence, in the search string. The entire matching string will be replaced with the replacement string. Do not put any wildcard characters in the replacement string, If you do, they will lose their wildcard effect.

OBJECT COMMANDS

Object commands allow you to specify labels, check memory and generate, store and retrieve object code.

ASM

```
ASM
ASM P
```

Initiates assembly of your source program which resides either solely in memory or which INCLUDEs source files from disk. MacASM features a two-pass assembler. During the first pass through the source code, it builds a symbol table with references for every label used in your program. During the second pass the assembler replaces symbol references with actual definitions, stores object code into memory (or writes it to a disk file) and will produce an assembly listing on the screen or the printer unless directed otherwise. At the end of the second pass, all the labels and their values are listed in alphabetical order.

The listing may be momentarily paused and restarted by successively pressing the SPACE bar. The assembly may be aborted by pressing the RETURN key. The content of the assembly listing may be controlled with the LIST directive; this allows you to list (or print) any part of it or none at all.

The assembly process normally continues after an error, so that you can catch (and later correct) as many errors as possible in the first pass. Certain errors, however, are fatal and cause the assembly to be aborted. The fatal line and an error message will be displayed.

If any errors are detected during pass one, pass two is not attempted. At the end of an assembly with no fatal errors, the total number of errors is printed. If any errors are detected in either pass, these are printed along with a copy of the offending line. Error messages are listed in Appendix B.

MacASM features a flexible assembler which can assemble source code from memory or disk and send the object code to memory and/or disk.

When the ASM command is entered, the source code in memory will be assembled. The source code may be totally in memory or, at minimum, the source code in memory may be limited to INCLUDE statements which include source program files from disk into the assembly process. Assembly of included source program files is slower than assembly of a source program that resides in memory because of disk access time.

You can specify that object code be assembled to disk and/or memory. The object code is sent to disk via a TFILE directive. If any part of the object code is assembled to memory, care must be taken to assure that it is sent to "Free Memory" as defined by MacASM. This can be determined with the MEMORY command.

Use the MEMORY command before assembly to determine the location of the source program. During assembly a symbol table will be created and will directly follow the source program in memory. A good rule of thumb for the size of a normal symbol table is that it should not exceed 5% of the size of the source program. However if you include the system equates from the "Library.Asm" file, this may be considerably larger. Free Memory begins after the symbol table. Estimate the beginning of Free Memory and set the program origin to at least this value with the ORG directive.

It is generally safe to start with a program origin of $10000 for a 128K Macintosh for most programs and reset the program origin after the first assembly since the symbol table length and the address of free memory will then be known. Use the MEMORY command to determine these.

(BSAVE and BLOAD)

BSAVE

Used to save binary object files to disk. The format is:

```
BSAVE"diskname:filename.Bin",start expression,end expression
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Bin is suggested but not required.

Example:

```
BSAVE"MYDISK:MYDREAM.Bin",$10000,$10800
```

This saves the object code from HEX locations $10000 to $10800 in memory to a disk called MYDISK in the binary file called MYDREAM.Bin.

Please note that a $ sign must precede a HEX number, otherwise the assembler will assume that a number is decimal.

BLOAD

Used to load binary object files from disk. The format is:

```
BLOAD"diskname:filename.Bin"
```

The "diskname:" is optional; if omitted, the default drive will be used. This is the last drive into which a disk has been inserted. The extension .Bin is suggested but not required.

Example:

```
BLOAD"MYDISK:PROGRAM.Bin"
```

Loads the binary file called PROGRAM.Bin from the disk named MYDISK into memory at the address where it was saved.

MEMORY

```
MEMORY
```

Displays the beginning and ending memory addresses of the source program, the symbol table and "Free Memory". The following values are examples only and may change depending on the events that occur prior to loading MacASM:

```
MEM
Source program: $00A74E-00A74E
  Symbol Table: $00A74E-00A74E
   Free Memory: $00A74E-016545
```

Free memory can be used to store and execute your object program without any undesireable effects. The assembler only allows you to assemble code to "Free Memory". If you try to assemble to any other section of memory or you forget to specify the program origin, a "Memory protect error." will result. This is a fatal error which will abort an assembly.

RUN

```
RUN expression
```

Executes the object code starting at "expression". MacASM views an object program executed with the RUN command as a subroutine. If you wish to automatically return to the

MacASM environment after your program has completed execution, terminate your program execution activity with an RTS mnemonic. Note that the RTS must be prior to the program END directive.

Please note that while MacASM is resident in memory, there is a maximum of 5K free on the application heap. Each time you run your program you will decrease the free space on the heap. This may eventually lead to a "Memory full" condition. If this space is insufficient for repeated tests of your program, consider setting it up as an application. Alternatively, you can type the DOS command and reclick MacASM to reinitialize the environment. Save your source program to disk first, as entering DOS will clear all source code from memory.

If your program modifies the screen border, this will remain in a modified state when MacASM regains control. MacASM will not reset the screen border because this would decrease the application heap free space. MacASM will be intact, however, and function normally.

SETLABEL

SETLABEL expression

Sets a normal label to a given value.

For example:

```
SET ZERO=0
```

SYMBOLS

SYMBOLS

Displays a copy of the Symbol Table, just like the one that normally is printed at the end of pass two of an assembly. Local labels are listed just after their associated normal label. For example:

```
SYMBOLS

000010CC: CLN
.0000=000010D8,.0001=000010DA,.0002=000010E6,.0003=000010F0
0000A001: CLOSE
000007AE: CLOSE.TXT
```

VALUE

VALUE expression

Executes any legal operand expression, and prints the value in both hexadecimal and decimal notation. VALUE may also be used to quickly convert decimal, octal or binary numbers to hexadecimal, to determine the ASCII code for a character, or to find the value of a label from the last assembled program, as in the following examples:

```
VAL 12345
$00003039       12345

VAL 'X
$00000058       88

VAL LOOPA+3
$0000084E       2126

VAL 15+$0A
$00000019       25
```

DOS COMMANDS

DOS commands allow you to do a directory of a disk, eject a disk, determine the disk serial number or quit MacASM and return to Macintosh Finder control.

DIRECTORY

```
DIRECTORY
DIRECTORY n
DIRECTORY n P
```

Lists the files on disk in drive n or default drive. The default drive is that drive into which a disk has last been inserted.

For example:

```
DIR          (default drive)
DIR 1        (Macintosh resident drive)
DIR 2        (Macintosh disk 2)
DIR 3        (hard disk, etc.)
DIR 4

DIR P        (prints directory of default drive)
```

The directory gives the disk volume name, the number of files and the number of free blocks of storage available on the disk. Each file is listed along with a locked or unlocked indicator, the file type, the creator, data and resource lengths and the file size.

DOS

```
DOS
```

The DOS command transfers control from the assembler to the Macintosh Finder.

EJECT

```
EJECT
EJECT 1
EJECT 2
EJECT 3
EJECT 4
```

Ejects a disk from drive n or the default drive. The default drive is that drive into which a disk has been last inserted.

Example:

```
EJE 1
```

This ejects the disk in the internal drive of the Macintosh.

SERIAL

```
SERIAL
```

Displays MacASM disk serial number. This number is unique and is required for MacASM support.

QUIT

```
QUIT
```

Returns control to the Macintosh Finder.

ASSEMBLER

Instruction Mnemonics

MacASM uses the standard 68000 instruction mnemonics as defined by Motorola Microsystems. All legal addressing modes are supported.

Registers and pointers are designated as follows:

D0...D7	Data Registers 0 through 7
A0...A7	Address Registers 0 through 7
A7 or SP	Stack Pointer
CCR	Condition Code Register
SR	Status Register - 16 bits
PC	Program Counter
USP	User Stack Pointer

A group of data and address registers, used by the MOVEM instruction, is designated as follows:

Syntax	Meaning
D0-D2/A5	D0,D1,D2, and A5
D4-D5/A1-A2/D6	D4,D5,A1,A2, and D6

Any combination of individual data and address registers and ranges of data and address registers can be used.

Length parameters are appended to instruction mnemonics to specify an operand byte length. These are as follows:

.B	one byte
.W	two bytes
.L	four bytes

The Motorola instruction mnemonic definition permits a choice between two types of syntax for PC relative addressing. One type uses a RORG directive and associated symbols. The other uses a displacement term "d" relative to the PC. MacASM employs the latter syntax.

For example:

```
00010    MOVE DATA1(PC,D1),D0   Note that DATA1 is an address.
                                (one byte)

00050    MOVE DATA2(A0),D0      Note that DATA2 is an offset
                                value. (two bytes)
```

The syntax for all the addressing modes is given below. The notation Dn refers to data registers D0 through D7; An refers to address registers A0 through A7. Expressions are designated by expr.

Syntax	Addressing Mode
Dn or An	Register Direct
(An)	Register Indirect
(An)+	Postincrement Register Indirect
-(An)	Predecrement Register Indirect
expr(An)	Register Indirect with Offset
expr(An,An)	Indexed Register Indirect with Offset
expr(An,Dn)	Indexed Register Indirect with Offset
expr	Absolute or Relative
expr(PC)	Relative with Address
expr(PC,An)	Relative with Index and Address
expr(PC,Dn)	Relative with Index and Address

Note that programs which are intended to be Macintosh applications must be relocatable in memory. You should use the last three addressing modes, which are PC relative, to assure the relocatability of your program.

MacASM provides a feature to eliminate ambiguity in absolute addressing that occurs in two pass assemblers. In a case where operand data is defined prior to its use, the assembler will use a corresponding word length, 16 or 32 bits. If the operand data is undefined, a long word, 32 bits, will be used.

For example:

Example 1

```
00010 DATA = $1000
00220       MOVE DATA,D0   16 BITS USED
```

Example 2

```
00450       MOVE DATA,D0   32 BITS USED
00460 DATA = $1000
```

In an analagous manner, the assembler will choose between 8 and 16 bits for a branch .S or branch .L.

The following syntax allows you to specify the length used in order to eliminate any ambiguity:

```
00100       MOVE <DATA,D0  16 BITS USED

00100       MOVE >DATA,D0  32 BITS USED
```

Directives

MacASM provides the following directives to control the assembly process and to define data in your programs.

DIRECTIVE	FUNCTION
ADJST	Adjust Boundary
ASC	ASCII String
AST	ASCII String Terminated
DATA	Data Definition
DEFS	Define Storage Allocation
DO, ELSE, FIN	Conditional Assembly
END	End Source Program
ENDM	End Macro Definition
ENDR	End Resource Definition
EQU or =	Equate Label
HEX	Hex String Storage
INCLUDE	Include Source File
LIST	Listing Control
MACRO	Macro Definition Start
ORG	Origin of Program Definition
PAGE	Pagination Control
RFILE	Resource File - Start Definition
RSRC	Resource - Start Definition
SET	Set Labels
STR	String with Length - Storage
TADDR	Target Address Definition
TFILE	Target File Definition
TITLE	Title Control

An explanation of each directive follows:

label ADJST

Adjust Boundary

Adjusts immediately preceding code by adding $00 after last position if that position is an odd address. This assures that the following code will continue on a word boundary. Note that the adjustment is based on the address specified in the ORG directive and not the TADDR directive, if employed.

For example:

```
00010           ORG         $400
00020           TADDR       $10001
00030 START     DATA        #0
00040           ADJST
00050           END
```

ASM

```
                    00010           ORG         $400
                    00020           TADDR       $10001
00000400: 00        00030 START     DATA        #0
00000401: 00        00040           ADJST
                    00050           END

      --- Symbol Table ---

00000400: START

0000 Errors in Assembly
Ok.
```

label ASC daaaaa...aaaad

ASCII String Storage

Stores the binary form of the string of ASCII characters in sequential locations in memory beginning at the current location. The label is optional and if present is defined as the address where the first character is stored. The string may contain any number of printing characters. The beginning and end of the string is indicated by any delimiter "d" that you choose. The delimiter "d" must be a printing character.

ASCII character codes are seven bit values with their high bit equal to zero. Since some programmers or programs prefer ASCII with the high bit set to one, an option is included to handle this case. The Macintosh features other characters, however, which are eight bit values, i.e. their high bit is one. This option does not affect the Macintosh non-ASCII characters.

```
    ASC daaa...aad   (normal) - does not affect high bit
    ASC -daaa...aad  (option) - sets high bit to one
```

label AST daaa...d **ASCII String Terminated**

This works like the ASC directive, except that the high-order bit of the last byte in the string is set opposite from its normal state. This allows a message printing routine to easily find the end of a message when using ASCII characters.

The AST directive should not be used with Macintosh non-standard characters whose high bit is normally set to one.

Examples of both ASC AND AST follow:

```
                         00010          ORG  $10000
                         00020 ;
                         00030 ; Example with ASC
                         00040 ;
00010000: 5354 5249
00010004: 4E47           00050 NSTR     ASC  "STRING"
00010006: 2222 22        00060 QT       ASC  /"""/
00010009: C8D5 C8BF      00070 HUH      ASC  -QHUH?Q
                         00080 ;
                         00090 ; Example with AST
                         00100 ;
0001000D: 5354 5249
00010011: 4EC7           00110 TSTR     AST  "STRING"
00010013: 2222 A2        00120 TQT      AST  /"""/
00010016: C8D5 C83F      00130 THUH     AST  -QHUH?Q


      --- Symbol Table ---

00010009: HUH
00010000: NSTR
00010006: QT
00010016: THUH
00010013: TQT
0001000D: TSTR

0000 Errors in Assembly
```

label DATA exprlist

Data Definition

Creates constants or variables in your program. "Exprlist" is a list of one or more expressions separated by commas. See the section on expressions for a detailed definition of data types.

The expression result may be specified as one or two bytes by immediately preceding the expression with a "#" or a "/" character respectively. The default definition is four bytes. If the expression evaluates to a value greater than your specification, the higher order bytes will be truncated. The expression result length in bytes is specified as follows:

```
  one byte  (lowest order)  : #expression

  two bytes (lowest order)  : /expression

  four bytes                : expression
```

The value of the expression, as one, two or four bytes, is stored at the current location in a high-low manner. If a label is present, it is defined as the address where the first byte is stored. If you use DATA to define a variable, it is a good habit to use an expression like "*-*", where "*" represents the program location counter (PC). This expression has the value of zero. Some examples follow:

```
                         00010          ORG   $10000
00002000:                00020 IN       EQU   $2000
00010000: 0000 03E8      00030 TEN3     DATA  1000  ; 4 bytes
00010004: 41             00040 LETTERA  DATA  #'A'  ; 1 byte
00010005: 2000           00050 BFPG     DATA  /IN   ; 2 bytes


---- Symbol Table ---

00010005: BFPG
00002000: IN
00010004: LETTERA
00010000: TEN3
```

label DEFS expression

Define Storage Allocation

Reserves a block of bytes starting at the current location in the program. The expression specifies the number of bytes to reserve. The label is optional. If there is a label, it will be assigned the value at the beginning of the block. The address of the beginning of the block will be printed in the address column of the assembly listing.

If the object code is being stored directly into memory, no bytes are stored for the DEFS directive. However if object code is being written to a file on disk using the TFILE directive, the DEFS directive will write "expression" bytes to the file. All the bytes written will have the value of $00.

The DEFS directive will generate a "Range error." message if the number of bytes is negative, or greater than 65535.

An example follows:

```
                         00010          ORG    $10000
00010000: 4321           00020 X        DATA   /$4321
00010002:                00030 Y        DEFS   $6
                         00040 *----------------------------

00010008: 4E71           00050          NOP

    --- Symbol Table ---

00010000: X
00010002: Y

0000 Errors in Assembly
```

Conditional Assembly

DO expression
ELSE
FIN

These directives allow you to include or exclude a particular section of source code in the assembly, depending on a condition which was set earlier in the assembly. The expression is evaluated as a truth value, and must be defined before the DO directive. If the expression evaluates to zero (false), lines that follow will not be assembled; if the expression evaluates to a non-zero value (true), the lines will be assembled.

The ELSE directive toggles the current truth value, thus permitting an if...then...else structure. There may be more than one ELSE directive within a DO - FIN block. Each time ELSE is encountered, the truth value is switched. FIN terminates the conditional section. ELSE is optional but FIN is required. DO - FIN blocks may be nested.

This directive set is often used to produce different specialized versions of a program from the same source code. For example, DO - FIN blocks may be used to exclude testing routines from a finished program or to add or delete extra variables.

An example follows:

```
00010            ORG       $10000
00020 ; Conditional Assembly Demo
00030 ;
00040 FLAG       EQU       0
00050            DO        FLAG
00060            BSR       SOME.PLACE
00070            ELSE
00080            BSR       ANOTHER.PLACE
00090            ELSE
00100            BSR       ONE.MORE.PLACE
00110            FIN
00120            RTS
00130 ;
00140 SOME.PLACE       RTS
00150 ANOTHER.PLACE    RTS
00160 ONE.MORE.PLACE   RTS
```

ASM

```
                    00010            ORG       $10000
                    00020 ; Conditional Assembly Demo
                    00030 ;
00000000:           00040 FLAG       EQU       0
                    00050            DO        FLAG
                    00070            ELSE
00010000: 6100 0006 00080            BSR       ANOTHER.PLACE
                    00090            ELSE
                    00110            FIN
00010004: 4E75      00120            RTS
                    00130 ;
00010006: 4E75      00140 SOME.PLACE       RTS
0001000A: 4E75      00150 ANOTHER.PLACE    RTS
0001000A: 4E75      00160 ONE.MORE.PLACE   RTS

    ---Symbol Table ---

00010008: ANOTHER.PLACE
00000000: FLAG
0001000A: ONE.MORE.PLACE
00010006: SOME.PLACE

0000 Errors in Assembly
```

(Note that lines 00060 and 00100 were not assembled.)

```
00040 FLAG       EQU       -1
```

ASM

```
                            00010            ORG        $10000
                            00020 ; Conditional Assembly Demo
                            00030 ;
FFFFFFFF:                   00040 FLAG       EQU        -1
                            00050            DO         FLAG
00010000: 6100 0008 00060            BSR        SOME.PLACE
                            00070            ELSE
                            00090            ELSE
00010004: 6100 0008 00100            BSR        ONE.MORE.PLACE
                            00110            FIN
00010008: 4E75              00120            RTS
                            00130 ;
0001000A: 4E75              00140 SOME.PLACE       RTS
0001000C: 4E75              00150 ANOTHER.PLACE    RTS
0001000E: 4E75              00160 ONE.MORE.PLACE   RTS


    --- Symbol Table ---

0001000C: ANOTHER PLACE
FFFFFFFF: FLAG
0001000E: ONE.MORE.PLACE
0001000A: SOME.PLACE

0000 Errors in Assembly
```

(Note that line 00080 was not assembled.)

label END

End Source Program

Ends source program. If not entered at the end of the source program, the assembler supplies this directive automatically.

ENDM

End Macro Definition

Ends macro definition. If not entered at the end of a macro, MacASM will consider all further code to be part of the macro until an ENDM is encountered. In the worst case this would be the entire program. Care should be taken to end every macro with an ENDM directive.

ENDR — End Resource Definition

Ends resource definition. If not entered, an error message will be printed.

label EQU expression
label = expression — Equate Label

Defines the label to have the value of the expression. The label cannot be redefined later in a program. If attempted, an "Extra definition error." message will occur. If the expression is not defined, an "Undefined label error." message is printed. If you forget to use a label with an equate directive, a "No label error." message is printed. One common use for this directive is to define logic values; another may be to define the location of the screen in memory.

For example:

```
00010 FALSE      EQU      0
00020 TEN4       =        10000
00030 SCREEN     EQU      $07A700
ASM

00000000:                    00010 FALSE      EQU      0
00002710:                    00020 TEN4       =        10000
0007A700:                    00030 SCREEN     EQU      $07A700


    --- Symbol Table ---


00000000: FALSE
0007A700: SCREEN
00002710: TEN4

0000 Errors in Assembly
Ok.
```

label HEX hhhh.....hh — Hex String Storage

Converts a string of hexadecimal digits to binary, two digits per byte and stores them at the starting location. For clarity, each two digits may be separated by a period or the underline character. The label is optional. However, if a label is present, it is defined as the address where the first byte is stored.

The HEX directive may be used to store BCD (binary coded decimal) data directly. For example the following statememt would produce the number 1984 in BCD.

```
BCD.DATA       HEX 1984
```

There must be an even number of hexadecimal digits; if not, the assembler prints out a "Bad address error." message.

NOTE: Unlike hexadecimal numbers used in operand expressions, a dollar sign "$" CANNOT be used within the HEX directive.

An example follows:

```
00000000:                 00010 FALSE  EQU   0
                          00010        ORG   $10000
                          00020 ;
00010000: 0123 4567       00030 TAB    HEX   01.23.45.67
00010004: FF0A FF         00040        HEX   FF_0A_FF

    --- Symbol Table ---

00010000: TAB

0000 Errors in Assembly
```

INCLUDE "diskname:filename.Asm"   Include Source File

Causes the contents of the specified source file on the specified disk to be included in the assembly at the position of the INCLUDE directive.

The program which is in memory when the ASM command is typed is called the "root program". Only the root program may have INCLUDE directives within it. If you put INCLUDE directives in a source file that has been included in a root program, a "Nested INCLUDE error." will occur.

The INCLUDE directive can be used to assemble very large programs whose source code cannot fit into memory all at once. It is also useful for connecting together a library of subroutines with a main program.

The INCLUDE directive can also be used to maximize the "Free Memory" to which your program may be directly assembled. The INCLUDE loads and asssembles the included source file one line at a time. The included file is not stored in memory thus allowing a maximum of "Free Memory" to which your object program may be directly assembled.

LIST optionlist — Listing Control

Controls the listing output of the assembler. "Optionlist" is a list of one or two of the following keywords, separated by a comma:

OFF	Listing off.
ON	Listing on.
MOFF	Macro expansion listing off.
MON	Macro expansion listing on.

If LIST OFF is put at the beginning of the source program, and no LIST ON occurs afterwards, no listing at all will be produced. The program will assemble much faster without a listing, as much of the time is consumed by putting characters on the screen and screen scrolling.

If you put LIST OFF at the beginning of your source program, and LIST ON at the end, only the alphabetized symbol table will print. The LIST ON must be placed just before the END directive.

You may also use this pair of directives to bracket any portion of the listing you wish to see or not to see.

The default condition is LIST ON.

With LIST MON in effect, the complete macro expansion will be listed. The call line will be printed with its line number, then the expansion lines, each with the same line number. LIST MOFF will turn off the macro expansion.

The default condition is LIST MON.

macroname MACRO — Macro Definition - Start

A macro definition must begin with the macroname followed by the MACRO directive and end with the ENDM directive. For detailed information, see the section on macros.

ORG expression — Origin of Program Definition

Sets the program origin and the target address to the value of the expression. The program origin is the address at which the object program will be executed. The target address is the memory address at which the program will be stored during assembly. The ORG directive sets these both to the same value, which is the usual way of operating.

If you do not use the ORG directive, the assembler will set both the program origin and the target address to $0. This will cause a "Memory protect error." upon assembly with subsequent assembly abort. An assembly must be made to "Free Memory" as defined by MacASM.

PAGE Pagination Control

Sends an ASCII form-feed character ($0C) to the printer. If the assembly listing is being printed on a printer which recognizes this character, a form feed will occur and the next listing line will appear at the top of the next page. The PAGE directive line itself is not listed.

RFILE Resource File - Start Definition

Defines a resource file that is necessary to create an application. See the section on applications for detailed information.

RSRC Resource - Start Definition

Starts the definition of a resource specification which makes up part of a resource file. See the section on applications for detailed information.

label SET expression Set Labels

This directive facilitates re-definable labels. Labels whose values are originally defined by the SET directive may be re-defined within the same assembly. Some examples are:

```
ASM

00000000:                  00010 X              SET         16
000000A0:                  00020 X              SET         X*$A
00000008:                  00030 Y              EQU         8
00000014:                  00040 X              SET         X/Y

    --- Symbol Table ---

00000014: X
00000008: Y

0000 Errors in Assembly
```

A label defined with an EQU statement cannot be redefined by using the SET directive.

STR "aaaaa...aaa" **String with Length - Storage**

Stores the string length and binary form of the string of ASCII characters in sequential locations in memory.

For example:

```
ASM
                          00010               ORG   $10000
00010000: 0554 6573
00010004: 7430            00020 A             STR   "Test0"
00010006: 0654 4553
0001000A: 542E 31         00030 B             STR   "TEST.1"

     --- Symbol Table ---

00010000: A
00010006: B

0000 Errors in Assembly
```

Note the string lengths of 05 and 06 respectively.

TADDR expression **Target Address Definition**

Sets the target address in memory at which the object code will begin to be stored during assembly. The target address must reside in "Free Memory". The target address is distinct from the program origin, which is set by the ORG directive or is implicitly set to $0. While the ORG directive sets both the origin and target address; the TADDR directive sets only the target address. Object code is produced ready-to-run at the program origin, but is stored starting at the target address. This directive is useful for cases where your program must later execute at an address normally protected by MacASM, i.e. any address not in "Free Memory".

TFILE"diskname:filename.Bin" **Target File Definition**

Causes the object code generated to be stored in a binary file on disk, rather than in memory. Only the code which follows the TFILE directive will be stored in the file. Code will be stored in the file until another TFILE directive is encountered. Once a TFILE directive is encountered in an assembly, all code will be sent to files on disk; no further code can be directed to memory during that assembly. The format is given in the following example:

```
01000     TFILE"MYDISK:MYNAME.Bin"
```

The extension .Bin is suggested but not required.

TITLE expression,title **Title Control**

When TITLE is in effect, the assembly listing will have a title line and page number printed at the top of each page. The expression specifies the maximum number of lines which will be printed on a page. This is the page length. The title will be printed starting at the left margin and can be up to 70 characters long. The title will be followed by a blank space. "Page n" will be printed immediately after the space. If there is no "title", the page number will be printed at the left margin. Centering of the title and page number can be adjusted by adding leading spaces to the title.

The assembler will issue an automatic form-feed when a page fills up. A page can be terminated early by using the PAGE directive. You can use more than one TITLE directive in a program if you like. The TITLE directive also issues a form-feed command.

The titling can be turned "off" by using a TITLE with a page length of zero.

Macros

A macro is a single instruction in your source code which, when assembled, is replaced by a defined series of instructions. Essentially it is a custom assembly language statement that stands for a predefined set of regular assembly language statements. If the code in your macro is used often, it saves a lot of typing. Moreover, your source program will be shorter and easier to read and understand. With macros, you can also define your own instructions for the 68000, and even rename or expand the 68000 mnemonics or the MacASM directives. Macros, however, are probably most useful for developing data tables.

While macros may look like subroutines, especially to beginning programmers, there is a difference. The macro itself operates only during the assembly process and produces the expanded source which is assembled into object code. The subroutine, however, operates during program execution only. The object code resulting from a macro after assembly is indistinguishable from that which could be produced from regular source code. While subroutines are great space savers for both source and object code, macros expand into normal assembly language statements and, in terms of their object code, save no space at all. The object code will be the same in either case.

Macros are defined using the MACRO directive. The definition format is:

macroname MACRO

A simple example follows to explain the definition:

```
                    00010 STR      MACRO
                    00020          DATA     #:2-:1
                    00030 :1       ASC      "]1"
                    00040 :2
                    00050          ENDM
                    00060  *-----------------------------
                    00070          ORG      $10000
                    00080 ;
00010000:           00090 X        STR      "Simple MACRO"
00010000: 0C        00090>          DATA     #:2-:1
00010001: 5369 6D70
00010005: 6C65 204D
00010009: 4143 524F 00090> :1       ASC      "Simple MACRO"
                    00090> :2

    --- Symbol Table ---

00010000: X

0000 Errors in Assembly
```

You may recognize this MACRO as performing the function of the STR directive. STR calculates the length of an ASCII string and then stores the string in memory, preceding it by the string length. In the example, the string length is 12 ($0C). The directive MACRO signals the start of a macro definition, and is preceded by a label which is the macro name. The example label name is STR. The operand "11" is a macro call parameter. In the example assembly, it will be replaced by the operand in the macro call line (in our example, Simple MACRO). The label ":1" is a private label used to name a branch point within the macro. The directive ENDM signals the end of a macro definition.

A macro must be defined before it is called. It is a good practice to put all macro definitions at the beginning of your program. Once a macro is defined, it can be called as often as you like by typing the macro name in the mnemonic field. At assembly time, the assembler will insert the correct code from the macro definition and your call parameters.

Call Parameters

Macro call parameters are dummy variables used in the macro definition. When the macro is called from a program, these parameters are replaced by the expressions used in the macro call operand field. There can be up to nine call parameters,]1 through]9, indicated by a "right bracket" and a number. There is also a parameter named]#, which represents the number of operands passed to the macro.

Parameters are written in the operand field of the macro call line, separated by commas. If you want a parameter to include a comma or space, enclose the parameter in quotation marks. If you want to also include a quotation mark, use two quotation marks in a row to define one. For example:

```
     001200     DOIT NOW,$23FF,"ABC DEF","ABC, DEF,"" GHI"
```

The macro called is named DOIT.
]1 is NOW.
]2 is $23FF.
]3 is ABC DEF.
]4 is ABC, DEF," GHI.
]# is 4.

Private Labels

Private labels are used inside a macro definition to name branch points in the same way that normal labels are used in the main program. They are written as a colon ":" followed by one or two digits.

The assembler treats each use of a private label as though the label included a macro call number. This allows the same private label to be used each time that you call the macro in your program. Private labels do not interfere in any way with local labels. Here is an example using both private labels and local labels. The ZAP macro with a call parameter of $FFFFFFFF will invert all pixels of the screen.

```
                         00010 ZAP     MACRO
                         00020         LEA             $07A700,A0
                         00030         MOVE.W          #5472-1,D1
                         00040         MOVE.L          #]1,D0
                         00050 :0      EOR.L           D0,(A0)+
                         00060         DBRA            D1,:0
                         00070         ENDM
                         00080 *-------------------------------------
                         00090         ORG             $10000
00010000: 343C 0063 00100 START   MOVE.W          #100-1,D2
00010004:           00110 .0      ZAP             $FFFFFFFF
00010004: 41F9 0007
00010008: A700      00110>         LEA            $07A700,A0
0001000A: 323C 155F 00110>         MOVE.W         #5472-1,D1
0001000E: 203C FFFF
00010012: FFFF      00110>         MOVE.L        #$FFFFFFFF,D0
00010014: B198      00110> :0      EOR.L         D0,(A0)+
00010016: 51C9 FFFC 00110>         DBRA          D1,:0
0001001A: 51CA FFE8 00120         DBRA          D2,.0


    --- Symbol Table ---

00010000: START
.0000=00010004

0000 Errors in Assembly
```

Each private label takes up 13 bytes of storage during assembly; this makes no difference, however, in terms of object code.

Listing the Macro Expansions

There are two directives to control the appearance of macros in the assembly listing.

With LIST MON in effect, the complete macro expansion will show. The call line will be printed first, and then the assembled code on subsequent lines. The expansion lines will have line numbers of the calling line and be indented one space.

With LIST MOFF in effect, only the macro call line will be printed. This saves space and makes the logic of the program easier to follow. However, you lose the listing of the object code, showing exactly what is stored at each address.

Using Conditional Assembly in Macro Definitions

The DO, ELSE and FIN directives may be used within macro definitions. They will be executed during macro expansion, so that the same macro can be expanded in different ways, depending on call parameters.

The example in the next section shows how you might use conditional assembly inside a macro definition.

Nested and Recursive Macro Definitions

Macros may be called within other macro definitions. Macros may even call themselves. This is not recommended, as the consequences may be more than expected.

There may be a case where you may want to redefine and expand a Motorola instruction mnemonic or a MacASM directive and use the original mnemonic or directive in the new definition. In this case the prior definition of the mnemonic must be immediately preceded by a period.

Two examples follow:

Example 1

You want to modify the Motorola PEA instruction mnemonic to accept a string for the operand instead of an address.

The following macro does it in five lines.

```
00010 *-----------------------------
00020 * PEA String
00030 *-----------------------------
00040 PEA       MACRO
```

```
00050          .PEA       :1(PC)
00060          BRA.S      :2
00070 :1       STR        "]1"
00080          ADJST
00090 :2
00100          ENDM
00110 *-------------------------------
00120          ORG        $10000
00130 *-------------------------------
00140 START    NOP
00150          PEA        "Test X"
00160          NOP
00170 *-------------------------------
00180          END
```

ASM

```
                    00010 *-------------------------------
                    00020 * PEA String
                    00030 *-------------------------------
                    00040 PEA      MACRO
                    00050          .PEA       :1(PC)
                    00060          BRA.S      :2
                    00070 :1       STR        "]1"
                    00080          ADJST
                    00090 :2
                    00100          ENDM
                    00110 *-------------------------------
                    00120          ORG        $10000
                    00130 *-------------------------------
00010000: 4E71      00140 START    NOP
00010002:           00150          PEA        "Test X"
00010002: 487A 0004 00150>          .PEA       :1(PC)
00010006: 6008      00150>          BRA.S      :2
00010008: 0654 6573
0001000C: 7420 58   00150> :1       STR        "Test X"
0001000F: 00        00150>          ADJST
                    00150> :2
00010010: 4E71      00160          NOP
                    00170 *-------------------------------
                    00180          END

    --- Symbol Table ---

00010000: START

0000 Errors in Assembly
```

Example 2

Suppose you want to write a macro which can be used to call one or more subroutines from a single source line. For example, CALL SUB1 should translate to JSR SUB1. CALL SUB1,SUB2 should generate two JSRs and so on. This is easily done using conditional directives and nested macro definitions. Note that the following example employs a macro which is just four lines long.

Conditional directive usage is fairly direct. The]# parameter is tested to determine whether another parameter is present; if so, a JSR line is produced.

```
00010 ;SAVE"Examp2.Asm"
00020 *--------------------------------
00030 CALL      MACRO
00040           JSR     ]1
00050           DO      ]#>1
00060           CALL    ]2,]3,]4,]5,]6,]7,]8,]9
00070           FIN
00080           ENDM
00090 *--------------------------------
00100           ORG     $10000
00110 START     CALL    SUB1,SUB2,SUB3,SUB4,SUB5
00120           RTS
00130 SUB1      RTS
00140 SUB2      RTS
00150 SUB3      RTS
00160 SUB4      RTS
00170 SUB5      RTS
00180 *--------------------------------
00190           END
```

This example program is included on the MacASM disk as part of "Examp2.Asm". Load the example and use ASM to view the macro expansion. If you want to print it out to study it in more detail, use ASM P.

Possible Errors

If there are more parameters on a macro call line than the macro definition expects, the extra parameters are simply ignored. You can use the]# parameter with conditional assembly directives (DO, ELSE, FIN) to test for the correct number of parameters.

If you do not have enough parameters on the call line, those missing will be treated as null strings. Again, you may test for the correct number of parameters, using conditional assembly directives.

The assembler tests for three error conditions. If you call a macro that has not been defined earlier in the program, you will see an "Undefined MACRO error." message. If you use a MACRO directive with no name in the label field, you will get a "No MACRO name error." message. If you use the "]" in a macro "definition without a digit 1-9 or "#" character following, you will get a "Bad MACRO parameter error." message.

MACRO LIBRARY

A macro library titled "Library.Asm" is provided on the MacASM disk. This library is used for developing applications which run under their own icon. The macros in this library can also be used in your general programming. For example, the OST and TBX macros are useful for accessing Macintosh ROM routines. Macintosh ROM equates are also part of the "Library.Asm" file.

APPLICATIONS DEVELOPMENT

MacASM includes a resource compiler which facilitates the creation of an application. This application is "stand-alone" and can be executed by double clicking the application's icon.

The resource compiler's actions are automatic. A properly constructed binary file is transformed into an application simply by double clicking the binary file icon.

The binary file is created from a general source program which contains variable, resource and segment definitions and your application's specific source statements. One source program defines your whole application.

A skeleton example follows. References to line numbers are for this example.

1. The source program should begin with an INCLUDE of the file called "Library.Asm". This defines macros used in variable allocation and resource compilation. (line 00030)

2. Next it is a good practice to INCLUDE your own macros or to type them here. (line 00040)

3. The program equates should then follow. (lines 00060-00130)

4. A global variable area is then defined and variables are declared. Variables declared here are set up relative to A5, a register used as a general pointer in the Macintosh. These can later be referenced by name in your program.

   You must first supply two arguments to the macro GLOBAL. Both arguments must be even numbers.

   The first argument is a space allocation of the number of bytes for the user variables. You can estimate this value by adding the byte lengths for the user variables. MacASM, however, can be used to calculate the value. Assign an initial value of 0 for the argument and assemble your program. This will cause a fatal error due to an insufficient allocation but will allow you to determine a correct allocation number by using the VALUE command. Type "VAL VAR.RLN" to obtain the correct allocation number for the argument. Edit your program using the number obtained from the VAL operation and reassemble the program using ASM.

   The second argument is a space allocation for the system variables. A safe value for this argument is $200. Smaller values may be suitable for your particular application, however, extreme caution here is suggested.

Next each of your variables are defined with the DEFV macro. The first argument to this macro is the variable length which is of the form B, W, L or an expression, indicating the number of bytes allocated to the variable.

The global variable definition area is then ended with an ENDG macro. (lines 00150-000250).

5. The binary file to be compiled is created with a TFILE directive. The filename is "Buffer" and any "filename.Bin" can be used. (line 00270)

6. The resource file is created with an RFILE directive. The filename is "Test" and any "filename" can be used. The "type" is "APPL" and fixed. The "signature" is "GOOD" and can be changed to any four characters. The Fdflags are set by $0000. These flags must correlate with the resources you define later in your program. (line 00290)

7. Resource definitions follow. These include your signature, icons, file references, bundles, etc. Resources must begin with an RSRC directive and end with an ENDR directive. (lines 000310-00430)

8. Your application's specific source program statements follow the resource definitions. These will be in one or more segments. Multi-segment source programs are usually particularly large or are modular in terms of memory usage. Due to the nature of the Macintosh segment loader your program must be entirely relocatable. You can do this by using the "PC relative with Address" and the "PC relative with Index and Address" addressing modes. (lines 450-750)

   Segments must begin with a SEG macro with two parameters. The first parameter is the segment number. The second parameter is the resource attribute flag. Any jump to another segment must be made with a SJMP or a SJSR macro with the target segment label as the macro call parameter. A segment must end with an ENDR directive since the SEG macro defines a resource.

   Segments are defined sequentially.

9. After all segments have been defined, segment "zero" must be defined, again using the SEG macro. Segment zero's first two parameters are identical in definition to that of the previously defined segments. In addition, parameter three is the length of the application globals and parameter four is the application parameter area. (start line 00770)

   A label SEG0 must follow on the next line. (line 00780)

Jumps between segments are now defined using the JP macro. Segment jumps for segment n are defined beginning with a label SEG_n, followed by JP macros. The first JP must have the program entry point as its first parameter. The second parameter is the segment number. Additional JP macros are used to define labels in the segment that are targets of a jump from another segment. Here the first parameter is the label and the second parameter is the segment number. The segment jump definition ends with a label END_n. (lines 780-860)

A label END0 must follow on the next line. (line 00870)

Segment "zero" definition is ended with an ENDR directive. (end line 00880)

10. The application is ended with an END directive. (line 00900)

Skeleton Example

```
00010 ;SAVE"Example.Asm"
00020 *-----------------------------
00030         INCLUDE "Lib.asm"
00040         INCLUDE "Ur.own.macros"
00050 *-----------------------------
00060 ;       Equates
00070          .
00080          .
00090          .
00100 SCREEN  EQU      $7A700
00110          .
00120          .
00130          .
00140 *-----------------------------
00150         GLOBAL 50,$200    ;User,System
00160 ;                         ;Both must be even!
00170         DEFV L,WINDOW1
00180         DEFV L,WINDOW2
00190         DEFV 16,DSKNAME
00200         DEFV W,MODIFY
00210          .
00220          .
00230          .
00240 ;
00250         ENDG
00260 *-----------------------------
00270         TFILE    "Buffer.Bin"
00280 *-----------------------------
00290         RFILE    "Test",APPL,GOOD,$0000
00300 *-----------------------------
00310         RSRC     STR,0,0
00320         STR      "Blah.blah.blah"
00330         ENDR
00340         RSRC     ICON,128,0,"MyIcon"
00350          .
00360          .
00370          .
00380         ENDR
00390         RSRC     WIND,129
00400          .
00410          .
00420          .
00430         ENDR
00440 *-----------------------------
00450         SEG      1,52
00460 START    .
00470          .
00480          .
00490          .
00500 LAB1     .
```

```
00510              .
00520              .
00530             SJMP      LAB2
00540              .
00550              .
00560              .
00570             ENDR
00580 *---------------------------------
00590             SEG       2,52
00600              .
00610              .
00620              .
00630 LAB2         .
00640              .
00650              .
00660             SJMP      LAB1
00670              .
00680             SJSR      LAB3
00690              .
00700             ENDR
00710 *---------------------------------
00720             SEG       n,52
00730 LAB3         .
00740              .
00750             ENDR
00760 *---------------------------------
00770             SEG       0,32,VAR.LEN,$20
00780 SEG0
00790 SEG_1       JP        START,1        ;ENTRY POINT
00800             JP        LAB1,1
00810 END_1
00820 SEG_2       JP        LAB2,2
00830 END_2
00840 SEG_n       JP        LAB3,n
00850              .
00860 END_n        .
00870 END0
00880             ENDR
00890 *---------------------------------
00900             END
```

The ASM command will assemble the source program to the target file on disk creating the "Buffer.Bin" file and icon. Type DOS to return to the system Finder control. Simply double click the "Buffer.Bin" icon to produce an application. During compilation, resource types and IDs will be displayed as they are compiled. If an error occurs, the offending resource will remained displayed.

Three examples of applications are included on the MacASM disk.

"Examp3.Asm" shows a simple two segment application which flashes the Macintosh screen.

"Examp4.Asm" shows a more sophisticated application involving two windows.

"Examp5.Asm" shows a 3-D graphics demo titled "Tumbling Mac".

APPENDIX A

MacWrite Compatability

Source program files generated by MacASM can be edited using Macwrite and vice versa. The following guidelines must be followed for compatibility.

A MacASM generated source file must be saved with the TYPE command.

A MacWrite generated source file must be saved as "text only" and loaded into MacASM with the ENTER command. A source file generated using MacWrite should not have any line numbers. These will be generated automatically during entry into MacASM. For best results, use the Monaco font -- as this is the only mono-spaced font currently available. A proportional-spaced font will result in unevenly aligned source statement fields.

MacWrite usage is subject to its memory limitation. It may be necessary to break a large MacASM source program file into several smaller files if the memory requirements of MacWrite are being exceeded.

APPENDIX B

Error Messages

Error Message	Meaning
Bad address error.	Addressing mode is incorrect.
Bad MACRO parameter error.	Parameter not of form]"number" or]#
Bad mnemonic error.	Mnemonic format incorrect.
Bad label error.	Label format incorrect.
DO nest too deep error.	Nest greater than 65535 levels.
DOS error.	File system error.
Expression error.	Expression format incorrect.
Extra definition error.	Label defined more than once.
File type mismatch error.	File type incompatible.
MACRO nest too deep error.	Nest greater than 255 levels.
Memory full error.	Memory limit exceeded.
Memory protect error.	Assembly outside of Free Memory.
Missing DO error.	ELSE or FIN without DO.
Missing ENDR error.	RSRC without ENDR.
Missing RSRC error.	ENDR without RSRC.
Nested INCLUDE error.	INCLUDE not in root program.
Nested RSRC error.	RSRC without ENDR.
No label error.	EQU or SET without label.
No MACRO name error.	MACRO without label.
No normal label error.	Local label without normal label.

Error Message	Meaning
Opcode not on WORD boundary error.	Opcode at odd address.
Range error.	Range out of bounds.
Replace too long error.	Line too long after replace.
String too long error.	String greater than 39 characters.
Syntax error.	Syntax or format incorrect.
Undefined label error.	Label non-existant.
Undefined MACRO error.	Macro not found.
Value > 65535 error.	Range too large.

File System Errors

Error Message	Meaning
BadMDBErr	bad master directory block
BdNamErr	there may be no bad names in final system
DirFulErr	directory full
DskFulErr	disk full
DupFNErr	duplicate filename (rename)
EOFErr	end of file
ExtFSErr	volume in question belongs to an external fs
FBsyErr	file is busy (delete)
FLckdErr	file is locked
FNFErr	file not found
FNOpnErr	file not open
FSDSErr	file system deep s--t; during rename - old entry deleted but could not be restored
GFPErr	get file position error
IOErr	I/O errors
MFulErr	memory full (open) or file won't fit (load)
NoMacDskErr	not a mac diskette
NSDrvErr	no such drive
NSVErr	no such volume
OpWrErr	file already open with write permission
ParamErr	error in user parameter list
PermErr	permissions error (on file open)
PosErr	tried to position to before start of file (r/w)
RFNumErr	refnum error
TMFOErr	too many files open
VLckdErr	volume is locked
VolOffLinErr	volume not on line error (was ejected)
VolOnLinErr	drive volume already on-line at MountVol
WPrErr	diskette is write protected
WrPermErr	write permissions error

APPENDIX C

Suggested Reading

Inside Macintosh, Apple Computer Inc., 1983

Kane G. et al, 68000 Assembly Language Programming, Berkeley: Osborne/Mc-Graw Hill, 1981

M68000 16/32-Bit Microprocessor Programmer's Reference Manual, Motorola Microsystems

Scanlon L., The 68000: Principles and Programming, Indianapolis, Howard Sams, 1981

APPENDIX D

Macsbug Debuggers

The MacASM disk includes two debuggers which are provided with permission of Apple Computer, Inc. These are Macsbug debuggers and are excellent program development tools.

The two debuggers are included on the MacASM disk as xMacsbug", and "Maxbug" and are normally not active in the MacASM environment.

"xMacsBug" is an "eight line" version of MacsBug. This means that when MacsBug is activated, it provides eight lines of debugging information on the bottom of the Macintosh screen. "MaxBug" is a full screen display version that should only be used on a 512K Macintosh. MaxBug will not run under MacWorks on the Lisa.

Macsbug is normally not installed in the MacASM environment. To install a Macsbug, simply name one of the two files "Macsbug" and reset or restart the Macintosh. Macsbug will self-install during the boot process.

Macsbug consumes between 16K and 32K of memory that is otherwise part of your "Free Memory". Thus you should only install a Macsbug when needed.

Macsbug is a line-oriented single-Macintosh debugger. Since Macsbug shares information with the application being debugged, Macsbug may not fit in memory with very large applications.

Macsbug includes the following features:

- the ability to display and set memory and registers
- the capability to disassemble memory
- the ability to step and trace through both ROM and RAM
- the monitoring of system traps
- display and checking of system and application heaps

Macsbug takes control whenever certain 68000 exceptions occur. You can then use Macsbug to examine registers or memory, step and trace through applications, or set up break conditions and execute the application until those conditions occur.

Press the interrupt switch to activate an installed Macsbug.

MacsBug uses six types of simple commands: Memory, Register, Control, A-Trap, Heap Disassembly and miscellaneous.

Memory Commands

DM a n (Display Memory)

Displays "n" bytes at address "a". If "n" is omitted, the default value is 16 bytes. If both "a" and "n" are omitted, the display will start where it last left off. The "dot" symbol is set to the address "a" by this command. The "dot" can be used to designate the last item displayed.

SM a el .... en Set Memory

Sets the given values at address "a". The amount of memory set is determined by the width of each expression. For hex and decimal values, the width is the minimum number of bytes that holds the digits given. Decimal numbers do not encode very optimally and should be avoided. Text literals encode directly from one to four bytes and indirect operands are four bytes wide. An expression's width is the maximum of any sub-expression. Also, the dot symbol is set to the address "a" by this command.

Register Commands

Dx e Data Register

Displays or sets data register "x". If "e" is omitted, the register will be displayed. If not, the register is set to "e".

Ax e Address Register

Displays or sets address register "x". If "e" is omitted, the register will be displayed. If not, the register is set to "e".

PC e Program Counter

Displays or sets the program counter. If "e" is omitted, the program counter will be displayed. If not, the program counter will be set to "e".

SR e Status Register

Displays or sets the status. If "e" is omitted, the status register will be displayed. If not, the register is set to "e". Note that the SR is limited to a word.

TD Total Display

Dumps the entire set of registers.

Control Commands

BR a c Break

Sets a break point at address "a". As many as eight breakpoints can be used. The "c" parameter is an option which determines the number of times the breakpoint is reached before the program is stopped. If "a" is omitted, a list of the current breakpoints and counts is displayed.

CL a Clear

Clears the breakpoint at location "a". If "a" is omitted, all breakpoints are cleared.

GT a Go Till

Sets a one-time breakpoint at location "a" which is automatically cleared upon arrival at address "a".

T n Trace

Traces through "n" instructions. If "n" is omitted, only one instruction is traced. Note that A-Traps are considered as a single instruction by this command.

S n Step

Steps through "n" instructions. If "n" is omitted, one instruction is executed. Note that A-Traps are NOT considered a single instruction by this command.

MR n Magic Return

Looks n bytes down the stack and replaces the long-word there with a magic address in the debugger. The long-word must be a valid return address to any routine. A return is not normally made; instead, control is passed to the debugger which swaps back in the real return address. If "n" isn't specified, the default value is zero. This command is used to bypass certain routines by tracing up to the first instruction of the specified routine whose return address is at the top of the stack and then doing a MR. Note that this command isn't repeated by the carriage return and that a Trace command is substituted.

A-Trap Commands

Many A-Trap commands use the following parameter convention. There can be up to six parameters: t1, t2, a1, a2, d1, d2. If none are given, the command is invoked for all A-Traps. If only the first parameter, t1, is given, the command will occur on A-Traps numbered t1. If t1 and t2 are given, it works on any A-Trap in that range. If t1, t2 and a1 are given, it uses the range for the trap numbers but a1 for the PC check. If t1-a2 are given, it will be invoked on traps in the range called from the given address range. Finally d1 and d2 form a range for register D0 in the same manner.

AB t1, t2, a1, a2, d1, d2 A-Trap Break

Breaks into the debugger when the condition specified by the parameters is satisfied.

AT t1, t2, a1, a2, d1, d2 A-Trap Trace

Traces the A-Trap but does not break into the debugger when the condition specified by the parameters is satisfied.

AH t1, t2, a1, a2, d1, d2 A-Trap Heap Check

Checks the heap for consistency when the condition specified by the parameter is satisfied.

HS t1, t2, a1, a2, d1, d2 Heap Scramble

Scrambles the heap on any A-Trap in the given range. The heap is scrambled only on certain calls. It always scrambles on NewPtr, Newhandle and ReallocHandle calls and only on SetHandleSize and SetPtrSize if the new length is greater than the current.

AD a1 a2 A-Trap Data Check

Calculates a checksum for the memory range (a1, a2) and then checks it on every A-Trap thereafter. If the checksum fails it will break into the debugger.

AX A-Trap Clear

Clears the effects of any A-Trap command.

Heap Commands

HC Heap Check

Performs a one-shot heap check of the current heap.

HD Heap Dump

Dumps out each block in the current heap in the following form:

```
  block address type size (master pointer) *
```

The block address points to the memory manager block. The type is 0 for a free block, 4 for a pointer, and 8 for a relocatable block. The size is the length of the block (not the pointer or handle). For relocatables the master pointer is given which contains the locked bits, etc. The asterisk marks any immobile object. The dump is followed by a summary which contains the total number of relocatable, the total of those locked, the total of non-relocatable blocks, the total number of free blocks and the total amount of free space.

HT Heap Total

Prints summary line from heap dump.

HX Heap Exchange

Toggles current heap between system and application heap.

Disassembler Commands

ID a Immediate Disasemble

This does a one line disassembly at location "a". If "a" is omitted, the disassembly will start at the next logical location. This sets the dot symbol to the location "a".

IL a n Immediate List

Disassembles "n" lines at location "a". If "a" is omitted, disassembly will begin at the next logical location. If "n" is omitted, a screen's worth of lines will be disassembled. This command also sets the dot symbol to "a".

Miscellaneous Commands

F a c d m — Find

Searches "c" bytes from address "a" looking for data "d" after masking the target with "m". As soon as a match is found, the address and value are displayed, and the dot symbol is set to that address. The size of the target is determined by the width of "d" and is limited to 1, 2 or 4 bytes. The commands can operate successively, so incremental searches are possible.

WH x — Where

If "x" is less than 512, the address corresponding to the A-Trap will be printed. If "x" is greater than or equal to 512, the A-Trap "nearest" the address "x" will be printed. This is useful for finding out what trap was executing when an error occurred.

CV x — Convert

Takes the expression "x" and prints it out in unsigned hex, signed hex, decimal and text.

RX — Register Exchange

Toggles the display mode so that registers are or are not dumped during a trace command. The disassembly at the PC will always occur.

INDEX