Editor Assembler Version 4.x
Reference Manual

MISOSYS, Inc.
P. O. Box 239
Sterling, Virginia 22170-0239
703-450-4181

*** * * N O T I C E * * ***

*** * * L I M I T E D W A R R A N T Y * * ***

MISOSYS shall have no liability or responsibility to the purchaser or any other person, company, or entity with respect to any liability, loss, or damage caused or alleged to have been caused by this product, including but not limited to any interruption of service, loss of business and anticipatory profits, or consequential damages resulting from the operation or use of this program.

Should this program recording or recording media prove to be defective in manufacture, labeling, or packaging, MISOSYS will replace the program upon return of the program package to MISOSYS within 90 days of the date of purchase. Except for this replacement policy, the sale or subsequent use of this program material is without warranty or liability.

*** * * W A R N I N G * * ***

MISOSYS, Inc.
P. O. Box 239
Sterling, Virginia 22170-0239
703-450-4181

Reference Manual

This manual documents both the Model I/III assembler package entitled "EDAS" and the TRSDOS 6 compatible package entitled "PRO-CREATE". The package which you have acquired is denoted by the label affixed to the master diskette.

Distribution Disk - Model I/III

The Model I/III Editor Assembler Version 4.x and each of its utilities, are single programs that work on both the Model I and III under LDOS 5.x, DOSPLUS 3.5, TRSDOS 2.3, and TRSDOS 1.3. The package includes EDAS/CMD, MED/CMD, MAS/CMD, ADDCTLZ/TXT, XREF/CMD, TTD/CMD, SAID/CMD, and SAIDINS/CMD. It is distributed on a 35 track single density data diskette. Model III TRSDOS 1.3 users will need to use their CONVERT utility and a two-drive system to transfer the files from the master disk to a working system disk. Model I TRSDOS 2.3 users need to first modify their TRSDOS system via a one-byte patch prior to transferring the files from the master disk to a working system disk (see "Model I TRSDOS 2.3 Patch"). The master disk is readable by LDOS and DOSPLUS. Use under DOSPLUS 3.5 requires patches to EDAS, MED, MAS, and SAID! The patch files are DPEDAS/PAT, DPMED/PAT, DPMAS/PAT, and DPSAID/PAT.

Distribution Disk - TRSDOS 6.x

The TRSDOS 6.x Editor Assembler Version 4.x is distributed on a 40 track double density data diskette. The package includes EDAS/CMD, MED/CMD, MAS/CMD, ADDCTLZ/TXT, XREF/CMD, SAID/CMD, and SAIDINS/CMD.

Making BACKUPs

It is strongly recommended that before using your new Editor Assembler, you should make a BACKUP copy to use in a working environment and retain the release diskette as your MASTER copy. This "master" should be backed up to produce a "working" copy and the "master" archived. The BACKUP utility procedures are found in your DOS Owner's Manual in the section entitled "UTILITY PROGRAMS". Don't forget that the release diskette does NOT contain a DOS; thus, your BACKUP procedure is for a data diskette. After creating a BACKUP copy of the Editor Assembler diskette, store the MASTER diskette in a safe place. Use only your "working" copy for production.

Model I TRSDOS 2.3 Patch

Model I TRSDOS users will find difficulty in reading the distribution disk due to the data address mark used for the directory. Therefore, before making a BACKUP or copying EDAS files from the diskette, you will need to change one byte of the TRSDOS 2.3 disk driver using one of the following three methods. This change will not affect the operation of your TRSDOS.

Method (1) directly modifies the system diskette with a patch. To prepare for this patch, obtain a fresh BACKUP of your TRSDOS 2.3 to use for this operation. Then enter the following BASIC program and RUN it. After you RUN the program, re-BOOT your TRSDOS diskette to correct the byte in memory.

```
10 OPEN"R",1,"SYS0/SYS.WKIA:0"
20 FIELD 1,171 AS R1$, 1 AS RS$, 84 AS R2$
30 GET 1,3: LSET RS$="<": PUT 1,3: CLOSE: END
```

Method (2) uses DEBUG to change the byte in memory. Use this if you do not want to patch your TRSDOS system diskette and are familiar with DEBUG.

1. **At TRSDOS Ready, type DEBUG followed by <ENTER>.**
2. **Depress the <BREAK> key to enter the DEBUGger.**
3. **Type M46B0 followed by the <SPACE> bar.**
4. **Type 3C followed by <ENTER>.**
5. **Type G402D followed by <ENTER>.**

Method (3) uses a POKE from BASIC to change the value directly in memory. This procedure is as follows:

1. **Enter BASIC (files = 0, protect no memory)**
2. **Type POKE &H46B0,60 followed by <ENTER>.**
3. **Type CMD"S followed by <ENTER>.**

Now, after using any one of the methods noted above, COPY the EDAS files from the master diskette to your TRSDOS system diskette.

The EDAS Facility

The MISOSYS Editor Assembler 4.x includes the following files:

```
ADDCTLZ/TXT - a text file with <ENTER> followed by CTL-Z
EDAS/CMD    - a combined line editor and macro assembler
MAS/CMD     - a stand-alone macro assembler
MED/CMD     - a stand-alone line editor
SAID/CMD    - a stand-alone full-screen text editor
SAIDINS/CMD - SAID installation program
TTD/CMD     - (Model I/III) a source cassette Tape-to-Disk converter
XREF/CMD    - a symbol cross-reference listing generator
```

The MISOSYS Editor Assembler is a RAM-resident text editor and RAM resident or direct disk assembler. The Editor Assembler was designed to provide the maximum in user interface and ease of use while providing capabilities powerful enough for the expert Z-80 assembly language programmer.

The text editing features of the Editor Assembler facilitate the manipulation of alphanumeric text files for both assembler source and compiler source languages. The most common use of the editing capability is in the creation and maintenance of assembly language source programs to be assembled by EDAS. Through full support of upper and lower case text entry, the Editor can serve as a line-oriented text processing tool.

The assembler portion of the Editor Assembler facilitates the translation of Z-80 symbolic language source code programs into machine executable code. This object code may then be executed directly from the DOS Ready prompt.

Although EDAS could be used as an entry-level assembler, the scope of the documentation assumes a previous knowledge of assembler language and the hexadecimal number system. This is not a "learning" manual - it details the use of EDAS Version 4.x but in no way attempts to teach you how to program in the Z-80 assembly language. You should have available a standard reference handbook on the Z-80 code. Many texts are available.

It is necessary that all source text to EDAS, MED, or MAS must have a Control-Z (1AH) as the last character of the text. This byte must immediately follow a CARRIAGE RETURN (0DH). If you are using an editor other than MED or SAID to prepare your source text, and that editor does not terminate the text file with a CONTROL-Z, you may have difficulty in using the file with the assembler. If such is the case, you can either (1) APPEND the file named ADDCTLZ/TXT to your file, or (2) load your file into SAID and resave it.

Notation Conventions

Braces "{}"

Braces enclose optional information. The braces are never input in Editor Assembler commands (Note: braces are used in C language source code).

Ellipses "..."

The ellipses represents repetition of a previous item.

Line number "line"

"line" represents a number arbitrarily assigned to a statement for the purpose of identifying it to the editor functions. "Line" can be any decimal number ranging from <1 - 65529>.

Period "."

A period may be used in place of any line number. It represents a pointer to the current line of source code being assembled, printed, or edited. It is termed the "current line pointer" throughout this documentation.

Top of Text "#" or "t"

The pound sign character, "#", or the letter "t", may be used in place of any line number during a line number reference. It represents the beginning or top of the text buffer.

Bottom of Text "*" or "b"

The asterisk character, "*", or the letter "b", may be used in place of any line number during a line number reference. It represents the bottom of the text buffer.

Line Increment "inc"

This is a number representing an increment between successive line numbers.

Lower Case Entry

Lower case is supported freely throughout EDAS for text and command entry. All Editor Assembler commands may be entered in lower case as well as upper case to facilitate its use as a general purpose text editor.

Assembler source code can be entered in upper case or lower case. For lower case entry, the Editor must be in the case converted mode (see the <S>witch case command). This mode automatically converts lower case entry to upper case except for text which is between single quotes (enabling lower case string constants) and for all text following a semicolon (permitting lower case comments).

Invoking EDAS

EDAS is a directly executable command file. It is accessed in response to the DOS command prompt simply by entering:

```
EDAS (ECM,JCL,ABORT,LC,EXT="ext",Pn=val)
EDAS *

ECM          Is used to specify the Extended Cursor Mode
             for LDOS 5.1.

JCL          is used when running from Job Control Language
             so that EDAS uses the @KEYIN routine for its
             keyboard input.

ABORT        if specified, EDAS will automatically abort
             after an assembly with errors. It will return
             to DOS Ready.

LC           is used when editing LC source files. It will
             set tabs to 4, default extension to "CCC", and
             invoke "lower case permitted".

EXT="ext"    provides a means by which the default source
             file extension can be altered to "ext".

Pn=val       can be used to pass symbol equates to the
             assembler from the command line. "n" can range
             from <1-4> permitting four symbol equates.

*            if specified, will reload EDAS and maintain
             the text buffer pointers.

Note: Return to DOS from EDAS via the <B>ranch command.
```

The optional parameters shown in parentheses are used to alter the behavior of EDAS and give it greater flexibility. These options are as follows:

ECM

The Extended Cursor Mode (ECM) facility of the LDOS keyboard driver permits the differentiation of the UP-ARROW key versus the left bracket "[" to allow the input of both codes. The LDOS user may specify the ECM parameter to use this option on the LDOS KI/DVR. The default is normal KI operation. The ECM parameter is for LDOS users only; not for DOSPLUS or TRSDOS.

JCL

The JCL parameter is applicable only to LDOS. EDAS uses an internal line input routine to enable the parsing of certain characters. This hinders the ability of commanding EDAS from within the Job Control Language (JCL) of DOS.

If you want to control the assembly process from JCL, use the JCL parameter in the EDAS command line. If you are going to <I>nsert text while in a JCL mode, then you must use the "%01" to simulate a <BREAK> in the JCL file. Don't forget, the "%01" can only be used if you are going to compile the JCL. For example, the following enters EDAS and inserts one line:

```
edas (jcl)
i
This is a test
%01
//stop
```

ABORT

This parameter will cause EDAS to abort and return to DOS upon an assembly or disk error, or one of the following errors: no text in buffer, line number too large, bad parameters, buffer full, no such line, *GET or *SEARCH error, *SEARCH file not a PaDS, PaDS member error. It is useful when running from a Job Control Language to inhibit erroneous jobs from continuing.

LC

This parameter is used when you are editing LC source files (C language). It will do three things for you: (1) change the source file default extension from "ASM" to "CCC" - "CCC" is used in the LC compiler, (2) change the tab stops from every eight columns to every four columns - more reasonable for LC source code, (3) invoke the <S>witch case command to "lower case permitted" as LC source code is entered primarily in lower case.

EXT="ext"

The "EXT=ext" parameter is valid only for LDOS/TRSDOS 6 users. This parameter is available for those using the EDAS editor to edit and maintain files other than EDAS assembler source files. For instance, the M-80 assembler uses "MAC" as the standard extension. FORTRAN uses "FOR". You may be using EDAS to create or edit JCL files. Use this parameter to change the default source file extension (that used with the <L>oad and <W>rite commands) to one of your choice. You must enter a full three characters if you use this parameter. For example:

```
EDAS (EXT="MAC")
```

specifies that "MAC" be used as the default extension (make sure the supplied extension is entered in UPPER CASE).

Note that the override of "CCC" if the LC parameter is used takes precedence. If LC is specified, the EXT= parameter is ignored.

Pn=val

This parameter provides the power of entering symbol table equates directly from the EDAS command line. "Pn" is actually four parameters as "n" can range from <1-4>. Thus, you specify the parameter as either P1, P2, P3, or P4. These parameters are EDAS entry symbol table additions. By passing parameter values with these on the EDAS command line, you can alter four

symbol table entries. Thus, you can use these to control EQUate options, pass values to symbols, etc. The format usable is:

Pn	**sets @@n to TRUE.**
Pn=ddd	**sets @@n to decimal value ddd.**
Pn=X'hhhh'	**sets @@n to hexadecimal value hhhh.**

TRSDOS 1.3 users need to enter parameter values in the following hexadecimal format: 0xxxx, where "x" stands for a hexadecimal digit 0-9, A-F.

The actual labels added to the symbol table as DEFLs are "@@n", where "n" is the same as the "n" of "Pn". This is depicted as follows:

P1 == @@1	**P2 == @@2**	**P3 == @@3**	**P4 == @@4**

The four symbols initially have a value of zero (logical FALSE). You can use these to externally set flags for use in conditional assembly (or whatever else your heart desires). For example, say you have a program that uses two conditional symbols, MOD1 and MOD3. If your program has the statements:

```
MOD1    EQU   @@1
MOD3    EQU   @@3
```

then an EDAS command line of EDAS (P1) will set "@@1" to TRUE and "@@3" defaults to FALSE. "MOD1" would be TRUE and "MOD3" would be FALSE since the two conditional symbols you are using are equated to the "@@n" parameters.

You will find this parameter support a great feature when running EDAS from JCL.

EDAS *

The "EDAS *" is used to re-enter EDAS keeping the source program and variables intact. This permits you to recover after a re-boot providing the Editor Assembler region is not disturbed or in case you inadvertently entered the <B>ranch command without saving your source file. Remember to hold the <ENTER> key depressed during the RESET operation if your SYSTEM diskette contains an AUTO function.

EDAS Command Mode

Once "EDAS" is entered, the following message will appear on the video display screen:

```
MISOSYS EDAS-n.n
```

The "n.n" is indicative of the current version number. This display is followed by a right caret ">" prompt. The prompting character is displayed whenever EDAS is ready to accept a command.

Invoking MAS

MAS can assemble a disk file directly from disk. Its syntax is:

```
MAS source/ASM [+L=listing/PRN +O=object/CMD +X=reference/REF
    +S=symbol/SYM +I=include/ASM ] [assembler switches]
    [(p1=value1,p2=value2,p3=value3,p4=value4,LINES=n)]

+L=listing/PRN   - send listing to spec in lieu of *DO.
                   Use -LP for printer (or +L=*PR if DOS
                   supported). Will inhibit -NL and -LP.

+O=object/CMD    - send object to spec in lieu of "source/CMD".
                   Will inhibit -NO.

+X=reference/REF - send cross reference data to spec in lieu
                   of "source/REF" if -XR switch invoked.
                   Will invoke -XR.

+S=symbol/SYM    - send symbol table to spec in lieu of *DO or
                   *PR depending on setting of -WS and -LP
                   switches. Will invoke -WS.

+I=include/ASM   - use spec for "*INCLUDE" assembler directive
                   which is similar to "*GET".

Assembler switches:
  -NL -WO -LP -WS -WE -NE -XR -NC -NM -CI -NH -SL -MF -NO

Parms:
P1-P4            - as in manual.
LINES=n          - set printed lines per page to n ( abbrev=L).
```

MAS is essentially the assembler portion of EDAS. It should react from the DOS command line just like the EDAS assembler would react from the EDAS command line. The DOS command line file redirection options permit you to easily control the routing of the various outputs. See the section on EDAS for the use of the Parameters (P1-P4). The Assembler Switches operate according to the documentation covering the "assemble" command.

Invoking MED

MED is the line editor portion of EDAS. It supports all EDAS commands except for the "A" command. It is invoked with the same syntax as EDAS; however, it does not support the "Pn" parameters.

Syntax

The basic format of an assembly language statement consists of up to four fields of information. These fields, in order, are:

{LABEL} {OPCODE} {OPERAND{S}} {;COMMENT}	
LABEL	**is a symbolic name assigned the address value of the first byte of the object instruction.**
OPCODE	**is the mnemonic of a specific Z-80 assembler instruction or pseudo-OPeration code.**
OPERANDS	**are arguments of the OPCODE.**
;COMMENT	**is an informative notation that is ignored by the assembler but aids in documenting the source code.**

As can be noted, none of the fields are required; however, each line should contain at least one field. If you want the comment field to occupy the entire line, start the line with a semi-colon in the first character position of the line - then, no other field is needed. A symbolic label can exist by itself on a line. There are some Z-80 operation codes that have no arguments; thus, an OPCODE could exist by itself on a line (in field 2). You will never have an argument by itself as an argument relates to an OPCODE.

The statement line is considered to be freely formatted; meaning there are no columnar restrictions. Fields are separated by one or more tabs or spaces. If a tab is used, it makes for neater listings. Tabs are retained as tabs and thus will keep source files smaller than using multiple spaces.

Symbolic Labels

A label is the symbolic name of a line of code. Labels are always optional. A label is a string of no greater than 15 characters. The first character must be a letter (A-Z) or one of the special characters, "$", "_", and "@". A label may contain within character positions 2-15, letters (A-Z), decimal digits (0-9), or certain special characters: "@", "_", "?", or "$". The "$" appearing by itself is reserved for the value of the program counter of the current instruction. It cannot be used as a single character symbol.

A symbol appearing by itself in the LABEL field of a line, will be interpreted as being equated to the current value of the program counter.

Certain labels are reserved by the assembler for use in referring to registers. Others are reserved for branching conditions (condition codes) and may not be used for labels. (these conditions apply to status flags). The following labels are reserved and may not be used for other purposes:

Reserved Labels
A, B, C, D, E, H, L, I, R, IX, IY, SP, AF, BC, DE, HL
C, NC, Z, NZ, M, P, PE, PO, ON, OFF

Opcodes

The OPCODES for the Editor Assembler Version 4.x correspond to those in the Z-80-ASSEMBLY LANGUAGE PROGRAMMING MANUAL, 3.0 D.S., REL 2.1, FEB 1977.

Operands

Operands are always one or two values separated by commas. Some instructions may have no operands at all.

A value in parentheses "()" specifies indirect addressing when used with registers, or "contents of" otherwise.

Constants are data declarations of fixed value. They are constructed as a sequence of one or more digits and an optional radix specification character. The digits must be valid for the radix used. The following table denotes aceptable constant composition:

Data Type	Radix Char	Digits	Examples
hexadecimal	H	<0-9,A-F>	1AH, 0ABH, 0FFH
decimal	D	<0-9>	107D, 107, 15384
octal	O or Q	<0-7>	166Q, 166O
binary	B	<0-1>	01101110B

Note: Decimal is assumed if the radix character is omitted

A constant not followed by one of the radix characters is assumed to be decimal. A constant must begin with a decimal digit. Thus "FFH" is not permitted, while "0FFH" is valid.

Operands may also be constructed as complicated expressions using the mathematical and logical operators. Due to the extent of the documentation, they are described in the section on "Expressions".

Comments

All comments must begin with a semicolon ";". If a source statement line starts with a semicolon in the first character position of the line, the entire line is a comment. If EDAS is in the lower case converted mode, comments will be retained in whatever case they are entered. It is suggested that comments be entered in lower case with punctuation as required. It will make your source code listings much easier to read. All entry of text following a semi-colon is maintained in its entered case.

Expressions

A value of an operand may be an expression consisting of multiple terms (labels and data constants) connected with mathematical operators. These expressions are evaluated in strictly LEFT to RIGHT order. No parentheses are allowed. EDAS does not support operator precedence. Most operators are binary, which means that they require two arguments. Both "+" and "-" have unary uses also. The following operators are supported:

Operator	Function	Example
+	Addition	value1 + value2
-	Subtraction	value1 - value2
*	Multiplication	value1 * value2
/	Division	value1 / value2
.MOD.	Modulo Division	value1 .MOD. value2
<	Shift Left or Right	value1 < -value2
.AND. or &	Logical Bitwise AND	value1 .AND. value2
.OR. or !	Logical Bitwise OR	value1 .OR. value2
.XOR.	Logical Exclusive OR	value1 .XOR. value2
.NOT.	Logical 1's Complement	FALSE EQU .NOT. TRUE
.NE.	Logical Binary Not Equal	value1 .NE. value2
.EQ.	Logical Binary Equal	value1 .EQ. value2
.GE.	greater than or equal to	value1 .GE. value2
.GT.	greater than	value1 .GT. value2
.LE.	less than or equal to	value1 .LE. value2
.LT.	less than	value1 .LT. value2
.SHL.	shift value1 left	value1 .SHL. value2
.SHR.	shift value1 right	value1 .SHR. value2
.HIGH.	obtain high order byte	.HIGH.value
.LOW.	obtain low order byte	.LOW.value
%	Length of MACRO	%#LABEL or %%
%&	MACRO label concatenation	#NAME%&L

Addition (+)

The addition operator will add two constants and/or symbolic values. When used as a unary operator, it simply echoes the value.

```
001E     CON30   EQU   30
0010     CON16   EQU   +10H
0003     CON3    EQU   3
002E     A2      EQU   CON30+CON16
```

Subtraction (-)

The minus operator will subtract two constants and/or symbolic values. Unary minus produces a 2's complement.

```
000E    A2      EQU   CON30-CON16
FFF2    A4      EQU   -A2
```

Multiplication (*)

The multiplication operator will perform an integer multiplication of a 16-bit multiplicand by an 16-bit multiplier. Overflow of the resulting 16-bit value is not flagged as an error.

```
01E0    A5     EQU     CON30*CON16
BF20    A6     EQU     60000*3   ;this overflows
```

Division (/)

The division operator will perform an integer division of a 16-bit dividend by an 8-bit divisor.

```
0002    A7     EQU     5/2
1B4D    A8     EQU     48928/7
```

Modulo (.MOD.)

The modulo operator calculates the remainder of the above integer division.

```
0001     A9     EQU     5.MOD.2
0005     A10    EQU     48928.MOD.7
```

Shift (<)

This operator can be used to shift a value left or right. The form is:

```
VALUE     <    {-}AMOUNT
```

If AMOUNT is positive, VALUE is shifted left. If AMOUNT is negative, VALUE is shifted right. The magnitude of the shift is determined from the numeric value of AMOUNT.

```
0057     HIORD  EQU    5739H<-8
C000     A1     EQU    3C00H<4
03C0     A2     EQU    3C00H<-4
BBFF     A3     EQU    3CBBH<8+255
03C0     A3     EQU    15+3C00H<-4
```

Logical AND (.AND. or &)

The logical AND operator bitwise ANDS two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" only if both arguments have a "1" in the corresponding position, or a "0" if either argument has a "0".

```
3C00     A1     EQU    3C00H&0FFH
0000     A2     EQU    0&15
0000     A3     EQU    0AAAAH.AND.5555H
```

Logical OR (.OR. or !)

The logical OR operator bitwise "ORS" two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" if either argument has a "1" in the corresponding position, or a "0" if neither argument has a "1".

```
3CFF     A1     EQU    3C00H!0FFH
000F     A2     EQU    0.OR.15
FFFF     A3     EQU    0AAAAH.OR.5555H
```

Logical XOR (.XOR.)

The logical XOR operator performs a bitwise exclusive OR on two constants and/or symbolic values. Each bit position of the 16-bit resultant value is a "1" only if both arguments have reversed bits in the corresponding position (i.e. one must have a "1" while the other must have a "0"). The XOR operation is considered a modulo two addition.

```
3CF8     A1     EQU    3C07H.XOR.0FFH
0007     A2     EQU    8.XOR.15
FFFF     A3     EQU    0AAAAH.XOR.5555H
```

Logical NOT (.NOT.)

This is a unary operator. It performs a one's complement on the term it precedes. Observe the following examples:

```
FFFE     T1     EQU    .NOT.1
FFFF     T2     EQU    .NOT.0
0000     T3     EQU    .NOT.-1
```

Logical NOT-EQUAL (.NE.)

This operator is a binary operator that compares two adjacent terms. The resultant value is TRUE if the terms are not equal. A FALSE result is returned if the two terms are equal. Observe the following examples:

```
0000    T1    EQU    1000.NE.1000
FFFF    T2    EQU    1000.NE.10
FFFF    T3    EQU    1.NE.-1
0000    T4    EQU    .NOT.0.NE.-1
```

Logical EQUAL (.EQ.)

This operator is a binary operator that compares two adjacent terms. The resultant value is TRUE if the terms are equal. A FALSE result is returned if the two terms are not equal. Observe the following examples:

```
FFFF    T1    EQU    1000.EQ.1000
0000    T2    EQU    1000.EQ.10
0000    T3    EQU    1.EQ.-1
FFFF    T4    EQU    .NOT.0.EQ.-1
```

Logical GREATER-THAN-OR-EQUAL-TO (.GE.)

This is a binary operator that compares two adjacent terms. The resultant value is TRUE if the left term is >= the right term.

```
0000    T1    EQU    1.GE.2
FFFF    T2    EQU    2.GE.2
```

Logical GREATER-THAN (.GT.)

This is a binary operator that compares two adjacent terms. The resultant value is TRUE if the left term is > the right term.

```
0000    T1    EQU    1.GT.2
0000    T2    EQU    2.GT.2
```

Logical LESS-THAN-OR-EQUAL-TO (.LE.)

This is a binary operator that compares two adjacent terms. The resultant value is TRUE if the left term is <= the right term.

```
FFFF    T1    EQU    1.LE.2
FFFF    T2    EQU    2.LE.2
```

Logical LESS-THAN (.LT.)

This is a binary operator that compares two adjacent terms. The resultant value is TRUE if the left term is < the right term.

```
FFFF    T1    EQU    1.LT.2
0000    T2    EQU    2.LT.2
```

Logical SHIFT LEFT (.SHL.)

This is a binary operator that shifts the left term a number of bits left according to the right term. It is the same as "value1<value2".

```
2340    T1    EQU    1234H.SHL.4
```

Logical SHIFT RIGHT (.SHR.)

This is a binary operator that shifts the left term a number of bits right according to the right term. It is the same as "value1<-value2".

```
0123    T1    EQU    1234H.SHR.4
```

Obtain HIGH-ORDER byte (.HIGH.)

This is a unary operator that provides a low-order result which is equal to the high order value. It is the same as "value.SHR.8".

```
0012    T1    EQU    .HIGH.1234H
```

Obtain LOW-ORDER byte (.LOW.)

This is a unary operator that provides a low-order result which is equal to the low order value. It is the same as "value.AND.0FFH".

```
0034    T1    EQU    .LOW.1234H
```

Macro Length Operator (%)

The length operator is applicable only with MACRO usage. Therefore, its use will be discussed in the chapter on MACRO PROCESSING.

Z-80 Status Indicators (FLAGS)

The flag registers (F and F') supply information regarding the status of the Z-80 at any given time. The bit positions for each flag are:

7	6	5	4	3	2	1	0
S	Z	X	H	X	P/V	N	C

C	is the Carry flag.	Z	is the Zero flag.
N	is the Add/Subtract flag.	S	is the Sign flag.
P/V	is the Parity/Overflow flag.	X	is not used.
H	is the Half-carry flag.		

Each of the two flag registers contain six (6) bits of status which are set or reset by CPU operations. Four of these bits are testable (C, P/V, Z, and S) for use with conditional jump, call, or return instructions. Two flags (H, N) are not directly testable and are used internally to handle Binary Coded Decimal (BCD) arithmetic. Two flag register bits (3, 5) are unused.

In the Z-80 instruction set, the "CALL", "JP", and "JR" instructions can contain a "condition code" which is part of the argument of the OPCODE. The branching determination is performed according to the result of the flag register status. The mnemonics for these condition codes are as follows:

Flag	Condition Set	Condition NOT Set
Carry	C	NC
Zero	Z	NZ
Sign	M (minus)	P (plus)
Parity	PE (even)	PO (odd)

Carry Flag (C)

The carry flag is set or reset depending on the operation being performed. For "ADD" instructions that generate a carry and "SUBTRACT" instructions that generate a borrow, the carry flag will be set. The carry flag is reset by an "ADD" that does not generate a carry and a "SUBTRACT" that generates no borrow. The "DAA" instruction will set the carry flag if the conditions for making the decimal adjustment are met.

For instructions RLA, RRA, RLS, and RRS, the carry bit is used as a link between the least significant bit (LSB) and most significant bit (MSB) for any register or memory location. During instructions RLCA, RLC s and SLA s, the carry contains the last value shifted out of Bit 7 of any register or memory location. During RRCA, RRC s, SRA s, and SRL s, the carry contains the last value shifted out of Bit 0 of any register or memory location.

For the logical instructions AND s, OR s, and XOR s, the carry flag will be reset. The carry flag can also be set (SCF) or complemented (CCF).

Add/Subtract Flag (N)

This flag is used by the decimal adjust accumulator instruction (DAA) to distinguish between "ADD" and "SUBTRACT" instructions. For all "ADD" instructions, "N" will be set to a "zero". For all "SUBTRACT" instructions, "N" will be set to a "one".

Parity/Overflow Flag (P/V)

This flag's state depends on the operation being performed. For arithmetic operations, P/V indicates an overflow condition when the Accumulator result is greater than the maximum possible number (+127) or is less than the minimum possible number (-128). The overflow condition is determined by examining the sign bits of the operands.

For addition, operands with different signs will never cause overflow. When adding operands with like signs and the result has a different sign, the overflow flag is set. For example:

```
 +120  =  0111 1000     ADDEND
 +105  =  0110 1001     AUGEND
--------------------------------
 +225  =  1110 0001    (-95) SUM
```

Adding two numbers together would result in a number that exceeds +127; thus the added positive operands result in an incorrect negative number (-95). The overflow flag is therefore set.

For subtraction, overflow can occur for operands of unlike signs. Operands of like sign will never cause overflow. For example:

```
  +127  =  0111 1111     MINUEND
(-)-64  =  1100 0000     SUBTRAHEND
----------------------------------
  +191  =  1011 1111     DIFFERENCE
```

The minuend sign has changed from positive to negative giving an incorrect difference. The overflow flag is therefore set.

P/V is used with logical operations and rotate instructions to indicate the parity of the result. The number of "one" bits in a byte are counted. If the total is odd, "ODD" parity (P=0) is flagged. If the total is even, "EVEN" parity is flagged (P=1). When inputting a byte from an I/O device "IN r,(C)", the flag will indicate the parity of the data.

During search instructions (CPI, CPIR, CPD, and CPDR) and block transfer instructions (LDI, LDIR, LDD, and LDDR), the P/V flag monitors the state of the byte count register (BC), When decrementing the byte counter results in a zero value, the flag is reset to zero, otherwise the flag is a one.

During "LD A,I" and "LD A,R" instructions, the P/V flag will be set with the contents of the interrupt enable flip-flop (IFF2) for storage or testing.

The Half Carry Flag (H)

The half carry flag's state depends on the carry and borrow status between bits 3 and 4 of an 8-bit arithmetic operation. It is used by the decimal adjust accumulator instruction (DAA) to correct the result of a packed BCD add or subtract operation. The "H" flag's state is according to the following table:

H	ADD	SUBTRACT
1	There is a carry from Bit 3 to Bit 4	There is no borrow from Bit 4
0	There is no carry from Bit 3 to Bit 4	There is a borrow from Bit 4

The Zero Flag (Z)

The Zero flag (Z) is set or reset if the result generated by the execution of a certain instruction is a zero. For 8-bit arithmetic and logical operations, the "Z" flag will be set to a "one" if the resulting byte in the Accumulator is zero.

For compare (search) instructions, the "Z" flag will be set to a "one" if a comparison is found between the value in the Accumulator and the memory location pointed to by the contents of the register pair HL.

When testing a bit in a register or memory location, the "Z" flag will contain the state of the indicated bit.

When a byte is transferred between a memory location and an I/O device (INI, IND, OUTI, or OUTD), if the result of register B minus one (1) is zero, the Z flag is set, otherwise it is reset. Also for byte inputs from I/O devices using "IN r,(C)", the Z flag is set to indicate a zero byte input.

The Sign Flag (S)

The Sign flag (S) stores the state of the most significant bit of the accumulator. When the arithmetic operations are performed on signed numbers, binary two's complement notation is used. A positive number is identified by a "zero" in bit 7; a negative number by a "one". The binary equivalent of a positive number's magnitude is stored in bits 0 to 6 giving a range of 0 to 127. A negative number is represented by the two's complement of the equivalent positive number. The range for negative numbers is -1 to -128.

When inputting a byte from an I/O device to a register, "IN r,(C)", the "S" flag will indicate either positive (S=0) or negative (S=1) data.

Pseudo-OPs

These assembler operations, although written much like processor instructions, direct the assembler to perform specific tasks during the assembly process but have no meaning to the Z-80 processor. Some generate data values and are called "data declaration" pseudo-OPs. Others that control paging operations are termed, "listing" pseudo-OPs. Operations to invoke the assembly of blocks of code based on conditional evaluations are supported through many "conditional" pseudo-OPs. The assembler pseudo-OPs are:

Constant Declarations	
DATE	Assembles system date as 8-character string, MM/DD/YY.
DB	Specifies a data byte or string of bytes. Also equivalent to DEFB, DEFM, and DM.
DC	Specifies a multiple of byte constants.
DS	Reserves a region of storage for program use. Equivalent to DEFS.
DSYM	Assembles "label" as an n-character string. (Similar to the construct, DB '&#label', in a macro.
DW	Specifies a word (16-bit data value) or a sequence of words. Also equivalent to DEFW.
DX	Assembles "expression" as a 4-hexadecimal digit string.
TIME	Assembles system time as 8-character string, HH:MM:SS.

Origins and Values	
DEFL	Establishes a value for a label which can be altered during the assembly.
END	Signifies the end of a *GET or *SEARCH member. Will indicate the end of the assembly when detected in the text buffer. Supplies the transfer address.
ENTRY	Uses the result of "expression" as the transfer address. This value overrides any expression on the final "END" statement.
EQU	Establishes a constant value for a label.
LORG	Establishes a load origin for executable files.
ORG	Establishes an execution origin for executable object code files or in-memory assemblies.

Conditionals

IF	**Conditional evaluation of expression.**
IF1	**Logically TRUE if the assembler is on the first pass.**
IF2	**Logically TRUE if the assembler is on the second pass.**
IF3	**Logically TRUE if the assembler is on the third pass.**
IFEQ	**Logically TRUE if expression1 = expression2.**
IFEQ$	**Logically TRUE if string1 = string2.**
IFLT	**Logically TRUE if expression1 < expression2.**
IFLT$	**Logically TRUE if string1 < string2.**
IFGT	**Logically TRUE if expression1 > expression2.**
IFGT$	**Logically TRUE if string1 > string2.**
IFNE	**Logically TRUE if expression1 <> expression2.**
IFNE$	**Logically TRUE if string1 <> string2.**
IFDEF	**Logically TRUE if "label" has been defined prior to this statement, else FALSE.**
IFNDEF	**Logically TRUE if "label" has not been defined prior to the statement, else FALSE.**
IFREF	**Logically TRUE if "label" has been referenced but not defined prior to the statement, else FALSE.**
ELSE	**Alternate clause to be assembled if the prior clause has evaluated FALSE.**
ENDIF	**Signifies the end of a conditional clause.**

Note: "IFxx$" denotes alternate macro string comparison.

Miscellaneous

COM	**Generates an object code file comment record.**
ENDM	**Designates the end of a MACRO model.**
ERR	**Forces an assembly error.**
EXITM	**Can be used to prematurely exit from a MACRO expansion. This is normally used within a conditional. [**]**
IRP	**The statements within IRP-ENDM are repeated for as many items are in the argument list with "dummy" being replaced by each argument in turn. [**]**
IRPC	**The statements within IRPC-ENDM are repeated for each character in the character-list while the "identifier" is replaced, in turn, from the identifier list. [**]**
MACRO	**Designates the prototype of a MACRO model.**
OPTION	**This permits the altering of any of the permissible assembler switches from within the source code.**
PAGE	**Transmits a form feed during a listing.**
REF	**Forces a reference to the symbols identified in the argument list.**
REPT	**The statements within REPT-ENDM are repeated according to the result of "expression". [**]**
SPACE	**Generates extra line feeds during a listing.**
SUBTTL	**Invokes a heading sub-title for listings.**
TITLE	**Invokes a heading title for listings.**
[**]	**Details are in the section on USING MACROS**

Pseudo-OP DB

The "DB" pseudo-OP is used to define a data byte or series of bytes. Its syntax is:

DB	**n{,n}{,'c'}{,s}{,expression}**
n	**Defines the contents of a byte at the current reference counter to be "n".**
'c'	**Defines the content of one byte of memory to be the ASCII representation of character "c".**
's'	**Defines the contents of n bytes of memory to be the ASCII representation of string "s", where "n" is the length of "s".**
expression	**Is a mathematical expression which evaluates to a number in the range <0-255>.**

The constant declaration "DB" permits the concatenation of its data arguments using the comma "," as an argument separator. Data values are denoted according to the specifications in the chapter on ASSEMBLY LANGUAGE INFORMATION.

The pseudo-OPs DM, DEFB, and DEFM can be used in lieu of "DB" and are completely equivalent.

"DB" string arguments permit two connected single-quotes to indicate a single-quote value PROVIDED that two or more characters precede the 2-quote appearance in the string. For example:

```
DB    'AB''C'
```

will produce the character string: 41 42 27 43. This may have been coded as a complex declaration such as, "'AB',27H,'C'", but the extensive declaration support in EDAS provides the easier specification.

The following are valid declaration statements:

```
DB    'This',' ','is',' ','a',' ','test'
DB    1,2,'buckle your shoe',3,4,'close the door'
DB    'This is a tes','t'!80H
```

The hexadecimal expansions of the constant will appear in listings as rows of eight bytes per row. The expansions may be suppressed from your listings by using the assembler switch, -NE.

Pseudo-OP DC

This pseudo-OP defines a repetitive constant. Its syntax is:

DC quantity,value	
quantity	**Specifies how many times that "value" is to be repeated as a data byte. It can be defined as any other data definition: n, expression, 'c'.**
value	**Is the constant to be repeated. As in a "DB" data declaration, the value can be specified as a character, 'c', a numeric value, n, or an expression evaluated to a number in the range <0-255>.**

The pseudo-OP, "DC", will define a repetitive constant and eliminate the necessity of defining a series of identical data values by long DB specifications. For example, the following two statements are equivalent:

```
DB 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DC 16,0
```

The latter is much shorter, easier to enter as text, more readable, and takes up less space in its source form.

The "quantity" must range from 1 to 65535 (a zero value will result in 65536). The "value" must be less than 256. With this pseudo-OP, you can generate repetitions of a single constant. For example, say you want to set 100 storage locations to a zero value during the assembly. Insert the statement,

```
DC 100,0
```

and it will be done. A character constant can also be used for "value" as illustrated in the following example:

```
DC 256,'A'
```

which will set the next 256 storage locations to the letter, "A".

The expansions of the constant will appear in listings just as they do in the DB expansion. The expansions may be suppressed from your listings by using the assembler switch, -NE.

Pseudo-OP DS

This pseudo-OP is used to reserve a quantity of storage locations for use by your program. Its syntax is:

DS nn	
nn	**Reserves "nn" bytes of memory starting at the current value of the reference counter.**

The DS pseudo-OP can also be entered as "DEFS".

The quantity, "nn", can be a data value or an expression. Note that "DS" does not define data values. The "DS" pseudo-OP adds the quantity of storage locations reserved to the current program counter (PC) to calculate a new PC value. When generating an object code file, this action will cause the next assembled byte to create a new load record. The following examples depict various "DS" declarations.

The statement,

```
FCB   DS   32
```

will define a 32-byte region for later use as a File Control Block. Its origin can then be referenced as "FCB". The statement,

```
TABLE DS   TABLE_LENGTH * TABLE_WIDTH
```

will reserve a quantity of storage locations equal to the result of multiplying the two terms, TABLE_LENGTH and TABLE_WIDTH.

If your source code is being assembled with the "-CI" switch, EDAS automatically converts all "DS" declarations into equivalent "DC" declarations using a value equal to zero. The above two examples would therefore be translated to the following:

```
FCB   DC   32,0
TABLE DC   TABLE_LENGTH * TABLE_WIDTH,0
```

Pseudo-OP DW

This declaration specifies a 16-bit data value. Its syntax is:

DW nn{,'cc'}{,nn}
nn Defines the contents of a 2-byte word to be the value, "nn".
'cc' Defines the contents of a 2-byte word to be the characters, 'cc'

The DW pseudo-OP can also be entered as "DEFW".

In the expansion of the data word, its least significant byte is located at the current program reference counter while the most significant byte is located at the reference counter plus one. The data word can be a numeric constant, an expression that evaluates to a 16-bit value, or a character constant of one or two characters. The following examples illustrate various forms of "DW" data declarations.

```
DW    10000,1000,100,10,1
DW    'ab'
DW    'R','o','y'
```

Note that if a single character is defined as a character constant word, the low-order byte of the word will contain the character value and the high-order byte of the word will be set to zero.

Pseudo-OP DATE

The DATE pseudo-OP is used to assemble the system date as an 8-character string, MM/DD/YY. It's syntax is:

DATE

This actual date is established when you power up your computer and respond to the DOS's date entry query or by using the DOS's DATE library command. The date string can be useful to place an ASCII date stamp in your object program for the purpose of identification as to when it was assembled. See example 1 for an illustration of DATE.

Pseudo-OP DSYM

DSYM is usually used within a macro to assemble the "symbol" argument as if it were a DB character string. It's syntax is:

```
label  DSYM  symbol

label       An optional statement label.

symbol      A defined symbol.
```

When used in a macro environment, "symbol" will have the "#" indicator prefixed to designate the symbol as a macro dummy argument name. An alternative method is to use the ampersand escape function within a standard quoted character string such as "DB '&#symbol'" which also assembles to the same thing in a macro. See example 1 for an illustration of DSYM.

Pseudo-OP DX expression

DX assembles "expression" as a 4-hexadecimal digit character string. Its syntax is:

```
label  DX    expression

label       An optional statement label.

expression  An expression operand.
```

The expression can be a simple symbol or a complicated collection of terms. The expression is evaluated to a 16-bit value and output as four hexadecimal digits. See example 1 for an illustration of DX.

PSEUDO-OP TIME

The TIME pseudo is used to assemble the system time as an 8-character string, HH:MM:SS. It's syntax is:

```
TIME
```

This actual time is established when you power up your computer and respond to the DOS's time entry query or by using the DOS's TIME library command. The TIME string can be useful to place an ASCII TIME stamp in your object program for the purpose of identification as to when it was assembled. See example 1 for an illustration of TIME.

Example 1

```
3000            00001           ORG     3000H
3000            00002 LBLNAM    MACRO   #SYM
3000            00003           DSYM    #SYM
3000            00004           DX      #SYM
3000            00005           ENDM
3000            00006           ENTRY   BEGIN
3000 210730     00007 BEGIN     LD      HL,MSG$
3003 3E0A       00008           LD      A,10
3005 EF         00009           RST     40
3006 C9         00010           RET
3007            00011 MSG$      LBLNAM  BEGIN
3007+42         00012           DSYM    BEGIN
     45 47 49 4E
300C+33         00013           DX      BEGIN
     30 30 30
3010 0D         00014           DB      13
3011 31         00015           DATE
     32 2F 33 31 2F 38 34
3019 30         00016           TIME
     39 3A 31 31 3A 33 36
0000            00017           END
```

Pseudo-OP DEFL

The "DEFL" pseudo-OP assigns a value to a label. The value is permitted to be changed during the assembly. The "DEFL" syntax is:

```
label  DEFL  nn
label  DEFL  expression

nn          Sets the value of "label" to the quantity "nn"

expression  Sets the value of "label" to the evaluated
            result of "expression".
```

This declaration is similar to the "EQU" declaration except that the label value is permitted to change during the course of the assembly without producing phase errors (which are generally observed as numerous MULTIPLY DEFINED SYMBOL errors). If the value of "label" is declared by a "DEFL", the declaration can be repeated in the program with different values for the same label. One useful purpose to support this method of coding would be to simulate the maintenance of two program reference counters. Observe the following sequence of code:

```
            ... some code
PROG$       DEFL    $           ; Save current program counter
            ORG     DATA$       ; Set PC to data counter
MSG1        DB      'This is a test message',CR
DATA$       DEFL    $           ; Save current data counter
            ORG     PROG$       ; Reset PC to program counter
            ... more code
PROG$       DEFL    $           ; Save current program counter
            ORG     DATA$       ; Now set PC to the data counter
MSG2        DB      'Another message',LF,CR
DATA$       DEFL    $           ; Save new current data counter
            ORG     PROG$       ;  then re-establish PC
            ... continuation of program code
```

The program maintains two address counters. One is utilized as a counter to keep track of the code portion of the program (PROG$), while the other is used to keep track of the data portion of the program (DATA$). This technique can be used to keep the data fields associated with routines in close proximity to their associated routine in the source code, while the object code location of the data is collected into some other region.

Labels defined as "DEFL" will be carried as "DEFL" in the EQUate file generation of the Cross-Reference utility. They will also be notated in the cross-reference listing by a plus sign, "+", prefix to the label name.

Pseudo-OP END

The "END" pseudo is used to denote the exit of a *GET or *SEARCH process, or when used in the memory text buffer, it will denote the end of the source code. Its syntax is:

END {expression}	
END {label}	
	Signifies the end of the source program (see text for handling during *GET and *SEARCH).
expression	Specifies an execution transfer address branch that will be used by the system loader.
label	Specifies an execution transfer address branch to be the value of "label".

The "END" statement is used to indicate to the assembler, when the last source code statement is reached so that any following statements are ignored. If no "END" statement is found, a warning is produced. The END statement can specify a transfer address (i.e. END LABEL or END 6000H). The transfer address is used by the DOS program execution to transfer control to the address specified in the END statement. Note that the END statement cannot have a label in the label field of the statement).

If an "END" statement is detected in a file read via *GET, the "END" is treated as if the end-of-file was reached and EDAS will switch back to assemble from what ever invoked the *GET. A similar process takes place with *SEARCH, except that EDAS continues the normal searching process.

PSEUDO-OP ENTRY

The ENTRY pseudo-OP is used to establish the object program's entry point when invoked as a CMD program. Its syntax is:

ENTRY expression	
expression	Specifies an execution transfer address branch that will be used by the system loader.

ENTRY uses the result of "expression" as the transfer address. The use of ENTRY will override any expression argument on the END statement.

Pseudo-OP EQU

This pseudo-OP assigns a constant value to a label. Its syntax is:

label EQU nn	
label EQU expression	
nn	Sets the value of label to nn.
expression	Sets the value of label to the calculated value of "expression"

The "EQU" (equate) pseudo-OP is the generally accepted way to define constant values for use in your program. This declaration serves a different purpose than the data declarations such as DB, DC, and DW. Data declarations specify storage locations that contain the values declared. The "EQU" assigns the value to the label; thus, anywhere the label is used, the assigned value is utilized. Your programs will be more readable, and easier to maintain if the values need to be altered in a program revision.

An "EQU" can occur only once for any label. A multiple "EQU" with different values will result in the MULTIPLY DEFINED SYMBOL error.

Pseudo-OP LORG

The "LORG" pseudo-OP is used to establish an object code file (or part of one) that loads at an address different from where it will execute. The syntax of "LORG" is:

LORG nn	
LORG expression	
nn	Is the address to start loading the object file (or part of the file).
expression	When evaluated, "expression" will be treated the same as "nn".

A load-origin assembler directive, "LORG", is provided to cause the load addresses of the object file to be based on the LORG operand while the execution code address references will still be based on the "ORG" operand. This is useful to construct a module (or part of a module) that will load at an address different from its execution address. For example:

```
ORG   5200H
LORG  7000H
```

will assemble code so that absolute address references and the execution addresses are referenced from X'5200'; however, the object code file will start

loading at X'7000'. Any subsequent "ORG" will maintain the offset difference established at the previous "ORG" until another "LORG" is detected. If you want to switch off the offsetting operation of LORG, add the statement

```
LORG  $
```

to follow the last statement of the offset block of code. The assembler will specifically test for the case, LORG $, so that it forces a new load block where one is required.

Pseudo-OP ORG

The "ORG" pseudo-OP is used to establish an address for the program counter so that the absolute address references within a program are designated. The syntax of "ORG" is:

```
ORG  nn
ORG  expression

nn          Sets the address reference counter to the
            value "nn".

expression  When evaluated, "expression" will be treated
            the same as "nn". Terms of "expression" must
            be defined prior to the "ORG" statement.
```

The "ORG" statement is used to tell the assembler at what address to begin generating the object code for statements which follow. The assembler will generate object code starting at the address specified by "nn" or "expression", automatically advancing the program counter by the length of each instruction or data declaration assembled. The "DS" data declaration advances the program counter by the amount of storage locations reserved.

A program can have more than one "ORG" statement. If multiple "ORGs" are used, and one or more inadvertently will cause the overwrite of a previously assembled module of code, no warning message of any kind will be issued. It is left up to the programmer, to protect against such events by use of conditional tests (using conditional pseudo-OPs) and the "ERR" pseudo-OP.

The ORG pseudo-OP causes no code generation itself but just prepares the assembly process to start a new object deck record with the generation of subsequent object code (note that if the evaluated address is one greater than the current PC, a new object file record will not be started).

Conditional Pseudo-OPs

The "conditional" pseudo-OPs provide a powerful way to maintain a program that is slightly different when assembled to run on different machine configurations. Instead of having to maintain multiple copies of a program, with each copy having some routines and modifications to make a "custom" version of the program, by using the conditional pseudo-OPs, you can maintain one set of source code that has conditional clauses which perform the "customization". It is very easy to specify which clauses are to be assembled during a particular assembly. The structure of a conditional clause is:

```
IFxx  argument_of_IF
code clause
ENDIF

THE OPERAND OF THE CONDITIONAL MUST BE DEFINED
PRIOR TO THE EVALUATION OF THE "IF" STATEMENT!
```

The operand of the "IF" takes on different formats depending on the particular "IF" pseudo-OP. It can be an expression, a label, or two expressions separated by commas. If the operand evaluates to a non-zero value, it is interpreted as a logical TRUE condition. If the operand evaluates to a zero value, it is interpreted as a logical FALSE condition. When the condition is TRUE, the conditional clause between the "IF" and the "ENDIF" is assembled. If "expression" evaluates to a zero value then the conditional clause is not assembled. For the sake of uniformity, use the value of "-1" for a logical TRUE and a "0" for a logical false so that, "FALSE EQU .NOT.TRUE" is a valid statement. The values can be set in programs as follows:

```
TRUE     EQU     -1
FALSE    EQU      0
MOD1     EQU      TRUE
MOD3     EQU      FALSE
```

Conditional clauses can also be nested, in case complicated logical constructs are needed or in case a conditional clause itself has a conditional sub-clause. For example:

```
IF    expression1
      IF   expression2
      ENDIF
ENDIF
```

is a two-level conditional. Conditional clauses can be nested to sixteen (16) levels although you will rarely find a need for more than three.

The conditional construct of IF-ELSE-ENDIF is coded as follows:

```
IF    expression
clause_1
ELSE
clause_2
ENDIF
```

which implies that if expression is TRUE, clause_1 assembles. If expression is FALSE, then clause_2 will be assembled. The ELSE construct is not required in a conditional but may be used where you have an alternative clause that can be based on one expression.

As mentioned earlier, the IF argument can take one of three forms. The conditional structures of these are as follows:

```
---Type I---      -----Type II------        --Type III--
IF[x] exp         IFxx[$]  exp1,exp2        IFyy    name
.                 .                         .
clause            clause                    clause
.                 .                         .
ENDIF             ENDIF                     ENDIF
```

[x]	Optional entry of 1, 2, or 3 to evaluate based on the assembler phase during the assembly.
xx	Can be "LT", "EQ", or "GT" representing less than, equal to, or greater than conditions respectively when comparing "exp1" to "exp2".
[$]	The "$" is specified in macro comparisons with the expressions treated as strings (see the section on MACRO PROCCESSING).
yy	Can be "DEF", "NDEF", or "REF" representing whether "label" has been defined, undefined, or referenced but undefined.

Pseudo-OPs IFx - Type I

The IF1, IF2, and IF3 conditional pseudo-OPs evaluate TRUE when the assembler is on pass 1, 2, and 3 respectively. Pass 1 is the first pass used to evaluate the value of all symbols. Pass 2 generates the listing and cross reference data file. Pass 2 will be omitted if -NL is TRUE and -XR is FALSE. Pass 3 generates the object code. Macros must be read in on each pass. EQUates must be read in on each pass if they are the object of an IFDEF pseudo-OP, otherwise, they can be read in on the first pass only. In the latter case, surround the *GET which gets the equate file with an IF1-ENDIF.

Pseudo-OPs IFxx - Type II

Among the Type II constructs, using "IFLT", if the value of expression_1 is less than the value of expression_2, then the conditional clause will be assembled. Using "IFEQ", the conditional clause will be assembled only if expression_1 and expression_2 have equal values. The "IFGT" pseudo-OP will assemble the conditional clause (i.e. result in a TRUE condition) only if expression_1 has a value exceeding that of expression_2. The last possibility is "IFNE", which will cause the assembly of the conditional segment if the expressions are not of equal value.

If, for instance, you want to ensure that a program does not assemble code past a particular address, then the ERR pseudo-op could be used in conjunction with IFGT to force an assembly error as follows:

```
IFGT  $,MAXADDRESS
ERR    Program is too long!
ENDIF
```

which compares the current value of the program counter (PC) to some previously specified maximum address. Once the PC exceeds this maximum value, the condition evaluates TRUE resulting in an assembly of the segment. The "ERR" pseudo-OP is used to force an assembly error.

Pseudo-OPs IFyy - Type III

Among the Type III constructs, "IFDEF name" will evaluate TRUE if "name" has been defined prior to the evaluation of the IFDEF on each assembler pass. "IFNDEF name" will evaluate TRUE if "name" has NOT been defined prior to the evaluation of the IFNDEF on each assembler pass. "IFREF name" will evaluate TRUE if "name" has been referenced but NOT defined prior to the evaluation of the IFREF on each assembler pass.

The Type III constructs will find greater use when working with source libraries of code. For instance, if a clause is a routine that is surrounded with an IFREF-ENDIF conditional, the routine will only be assembled if prior to the segment, the "name" has been referenced but not yet defined. If "name" is the entry point symbol to the routine, then the routine will be assembled if it is needed. Similarly, you may have a library routine that is always to be placed in your program unless its "name" has already been defined in some alternate routine. Surrounding it with the IFDEF-ENDIF conditional will inhibit its assembly if your program has defined that name.

Suppressing FALSE Conditionals

If during the listing pass, you want to suppress the listing of certain conditional clauses that are not assembled (i.e. they are evaluated as FALSE), use the following sequence of operators:

```
*LIST OFF
IF    expression
*LIST ON
code clause
*LIST OFF
ENDIF
*LIST ON
```

With this sequence, the "IF" and "ENDIF" lines will always be suppressed. The conditional clause will only be listed if the condition being evaluated is logically TRUE. If no FALSE conditional clause is to be listed, then you may use the assembler "-NC" switch which inhibits the listing of all FALSE conditionals - including the IF-ENDIF statements.

Pseudo-OP ENDIF

Each "IF" statement must be matched up with a corresponding "ENDIF". The "ENDIF" is needed to define the scope of the conditional clause.

Pseudo-OP COM

This pseudo-OP is used to generate a comment record in the object code file. Its syntax is:

COM <string>
<string> Is the information to be placed as a comment.

An object file comment record can be generated within the executable object code file directly by using the "COM" pseudo-OP. The comment string must have a length less than 128 characters. As can be noted, the comment string must be enclosed in angle brackets. The closing bracket may be omitted. If lower case characters are desired, then single quotes must surround the angle brackets. Neither the quotes nor the angle brackets will be part of the comment record.

The "COM" pseudo-OP will generate a comment record in the object file of the format X'1F' followed by the string length, followed by the string itself. A typical use would be to place a non-loading copyright statement in an executable object code file. For example:

```
COM   '<Copyright (c) 1982 by Roy Soltoff>'
```

will produce the comment record which would be viewed if the file were listed.

The generation of the "COM" object code record will be inhibited if the assembly is performed using the "-CI" switch. A binary core-image file can not have a non-loadable record.

Pseudo-OP ERR

The "ERR" pseudo-OP is used to force an assembly error. Its syntax is:

ERR {message}
message is an optional message to inform what is wrong.

This pseudo-OP forces an immediate warning error and displays the optional message. It is commonly used in a conditional clause for error trapping.

Pseudo-OP OPTION

This pseudo-OP is used to alter the state of any of the assembler switches entered on the command line invoking the assembly (either via MAS or the "A" command of EDAS). Its syntax is:

OPTION {-/+}switch{,-/+switch},...
-/+ An optional prefix to turn the switch OFF or ON.
switch Any of the permissible assembler switches.

Prefix each switch with "-" to turn OFF, or "+" to turn ON (i.e. +NL suppresses the listing - sets the NO LISTING switch to TRUE). If "+" is omitted, it is assumed. The COMMA separator is mandatory if you omit the "+". OPTION switches over-ride command line switches.

The OPTION pseudo-OP is only processed during the first pass; therefore, you cannot use it to dynamically switch options ON and OFF during an assembly. It is used to conveniently set options specific to a source stream to eliminate the need for their entry on the assembler command line.

Pseudo-OP REF

REF may be used to force a reference to the symbols identified in the argument list. Its syntax is:

REF symbol1{,symbol2},...
symboln A "name" to be force-referenced.

This function may be useful to force references to macros so that they may be loaded via a '*SEARCH' operation.

Listing Pseudo-OPS

Four pseudo-OPs are available to control the assembler listings. These are: PAGE, SPACE, SUBTTL, and TITLE. Their syntax is:

```
PAGE

SPACE   n

SUBTTL  {<string>}

TITLE   <string>

n          Specifies how many line feeds to generate.

<string>   Is the title or sub-title string to appear in
           the listing headings.
```

A new page can be forced to provide separation of routines, modules, etc. by using the "PAGE" pseudo-op. This pseudo-OP will be ignored if it appears between *LIST OFF and *LIST ON. PAGE statements are automatically suppressed from the listing. PAGE will output the form feed character only during the listing pass.

"SPACE n" performs line spacing whenever the "SPACE" pseudo-OP is used. When assembled, "n" is the number of lines to space and is interpreted as modulo 256. The line containing the SPACE pseudo-op is not displayed. This pseudo-op also will be ignored if it appears between *LIST OFF and *LIST ON.

A heading sub-title is requested with the "SUBTTL" pseudo-OP. The sub-title string can be up to 80 characters long. A NULL length indicates that sub-titling is disengaged. The SUBTTL string does not need to be enclosed in angle brackets; they are optional. SUBTTL automatically invokes a PAGE. Lower case strings can be maintained by surrounding the brackets with single quotes. You may change the subtitle by using "SUBTTL" pseudo-OPs throughout the text. If the "SUBTTL" text string is null, then subtitling will cease on the subsequent page. A line will also be skipped between the subtitle and first printed text line on the page.

The "TITLE" pseudo-OP automatically invokes a page heading and adds the title to the headings of assembler listings. The first "TITLE" pseudo-OP found in the text will be used for titling. Any other "TITLE" pseudo-OPs will be ignored. The title string is limited to 28 characters. The left and right carets (angle brackets) must be entered but are not output in the listing - they serve only to delimit your title string. The title line will include the EDAS version, the date and time retrieved from the system, your title string, and a page number [page number is limited to the range <1-255> and will wrap around to zero if more than 255 pages are printed]. A line will be skipped between the title and start of printed text (or subtitle if used). Lower case titles will be maintained by surrounding carets with single quotes as in:

```
TITLE   '<This is an UC/lc title>'
```

Assembler Directives

The MISOSYS Editor Assembler Version 4.x, supports five assembler commands. In contrast to source statements which are translated to machine language, these directives are "conversation" to the assembler. Each directs the assembler to behave in a particular manner or perform a specific function. The directives, by themselves, do not generate any machine language code - they merely act as "commands" to the assembler. Each "command" must start in column one of a source statement line, and must start with an asterisk (*). Only the first character of each directive is significant. The entire directive "word" may be entered, or the directive may be abbreviated to its first character. The assembler directives are:

*GET file	Causes the assembler to begin reading source code from the "file".
*INCLUDE	Same as "*GET" for EDAS; works in concert with "+I=filespec" for MAS.
*LIST OFF	Causes the assembler listing to be suspended, starting with the next line.
*LIST ON	Causes assembler listing to resume, starting with this line.
*MOD exp	Advances the "module" character substitution string and optionally sets/resets the prefix.
*SEARCH lib	Invokes an automatic search of the Partitioned Data Set (PaDS) "lib" to resolve any undefined references capable of being resolved by PaDS assembler source member modules.

*GET filespec

This directive invokes assembly from a source disk file. Its syntax is:

*GET filespec	
filespec	Causes the assembler to begin reading source code from the file, "filespec".

This directive tells the assembler to temporarily switch its source assembly to the file identified as "filespec", and use it to continue the assembly. A default file extension of "/ASM" will be used if none is provided in the directive statement. The file itself can be headered and/or numbered, as the assembler will automatically detect its type and adjust accordingly; however, all nested *GETs must be similarly configured. When the end-of-file

is reached, or an assembly language "END" statement is read, assembly automatically resumes from the next statement following the statement which invoked the "*GET". Any "END" statement read during the *GET process will be ignored as the program end. The only "END" accepted will be that in the text buffer (or the source file identified on the command line in the case of a MAS assembly).

"*GETs" can be nested to five (5) levels. That is, a source statement can GET a file which GETs a file which GETs a file which GETs a file which GETs a file. This assembler directive is extremely powerful. It can be used to provide the capability of assembling large programs which are stored on disk in modules, since more than one *GET may be in the text buffer or "gotten" file.

The text buffer can be composed of nothing but *GET statements (and one END statement) which will provide maximum space in the text buffer for generation of the symbol table. For example, the following could represent the source linkage needed to assemble a program called "PARMDIR/CMD":

```
; PARMDIR/ASM - 04/07/82
;*=*=*
;     Linkage to assemble PARMDIR
;*=*=*
*GET  PARMDIR1
*GET  PARMDIR2
*GET  PARMDIR3
      END  PARMDIR
```

***INCLUDE filespec**

This directive is used to insert a subordinate source file into the input stream. Its syntax is:

*INCLUDE {filespec}	
	Same as "*GET" for EDAS; works in concert with "+I=filespec" for MAS.
filespec	Causes the assembler to begin reading source code from the file, "filespec".

In the EDAS macro assembler, "*INCLUDE" operates exactly like "*GET". It is totally equivalent in operation.

When used with the MAS macro assembler, "*INCLUDE" is used to "get" the file identified on the MAS command line with the "+I=filespec" redirection specification. With MAS, this is useful to effect the inclusion of a specified file in the input stream based on your command line option. Note that for use with MAS, there should be only one "*INCLUDE" within your input file stream.

LIST ON/OFF

This directive is used to suppress the listing of blocks of code. Its syntax is:

***LIST off/on**
OFF — **Causes the assembler listing to be suspended, starting with the next statement.**
ON — **Causes assembler listing to resume, starting with this statement.**

The pair of directives, "*LIST OFF" and "LIST ON", can be used to suppress the listing of a block of code. Once the "*LIST OFF" is invoked, all statements following will not be listed to the display or the line printer (if assembler switch -LP is specified). The directive "*LIST ON" re- establishes standard listing. An exception to the suppression is that any assembler source statement containing an assembly error will be listed along with its appropriate error message. In this manner, you can use an "*LIST OFF" directive at the beginning of your assembly source (to suppress all listing) and lines containing errors will be forced to be displayed by EDAS.

Some examples illustrating use of the *LIST directive are:

```
*LIST OFF
      DB  'This line will not be displayed!'
*LIST ON

*LIST OFF
      DB  'Only the next line will be displayed!'
      LD  (M,100
*LIST ON
```

***MOD expression**

This directive is used to increment a character substitution string for the purpose of simulating local labels. Its syntax is:

***MOD**
Advances the "module" character substitution string.

The "*MOD" directive will increment a string replacement variable each time the directive is executed. The string will replace the question mark, "?", character in labels and label references found in any line assembled

from a *GET or *SEARCH file. Its use is essentially applicable to subroutine libraries where duplication of labels could occur. By specifying the "*MOD" directive as the first statement of each module of code and by using a question mark in labels, you can construct source subroutine libraries for use in your programs without having to worry about duplicate labels occurring. Unless at least one "*MOD" statement is specified, the question mark will not be translated.

Labels such as $?001 will have the "?" replaced with the current "MOD" string value. Thus, a "*MOD" directive preceding each module will force $?001 labels in each module to be distinctly named by having the question mark replaced with the substitution string.

The label substitution automatically generates up to a three-character string: A...Z, AA...ZZ, AAA...ZZZ. The "MOD" string value cycles from A-Z, then from AA-AZ, BA-BZ, ..., ZA-ZZ, then from AAA-AAZ, ABA-ABZ, ... etc. This will allow for a simulation of "local" labels. Remember, the "?" substitutions will only be made to those source lines fetched from a *GET or *SEARCH file, not from statements resident in memory! It really was designed that way folk's, it's not just a limitation.

If you need more than the 18278 unique string values generated by a single/dual/triple alphabetic string (26*26*26+26*26+26), you will probably have to move to a machine with more memory, as there wouldn't be enough room in the symbol table to adequately make use of the current maximum.

***SEARCH filespec**

This directive is used to invoke an automatic search of a Partitioned Data Set (PaDS) source library. Its syntax is:

***SEARCH filespec**	
filespec	**Invokes an automatic search of the PaDS "filespec/LIB" to resolve any undefined references capable of being resolved by PaDS assembler source member modules.**

This assembler "*SEARCH filespec" directive is a very powerful feature. It will invoke, a directory search of the Partitioned Data Set "filename/LIB" for all members that will resolve undefined references in the source assembly. This provides a source library structure for the assembler. "*SEARCH" will require two (2) levels of "*GET" nesting. Also, restrictions prevent a "*SEARCH" member from using a "*GET" directive or another "*SEARCH" directive. The library members must be lowest level. The default file extension for searched files is "LIB".

The PaDS source library constitutes members composed of one or more routines. Each routine that needs to be automatically fetched should have its routine name (the label field entry) in the PaDS member directory. This is accomplished by naming the source file to be appended to the library the same

name as the routine or by appending using a MAP. Details on constructing and using Partitioned Data Sets is included with PaDS documentation. The PaDS utility is available separately.

The assembler will search the PaDS library and locate a member name that matches up with a symbol table entry. If that symbol is currently undefined, the source member will be accessed and read just as if it were the target of a "GET". The assembler will verify that the member just accessed did in fact define the symbol invoking its access. If a member is accessed and there exists no symbolic label in the member that has the same name as the member name, the assembler will abort the assembly and advise of a library error by displaying the message:

```
Member definition error: filespec(member)
```

At the conclusion of the member's source code, the assembler will continue to search the PaDS library until it exhausts all PaDS members. There are no restrictions on the order of members. Routines in one member can reference other members with complete disregard as to any ordering of entries in the PaDS. The assembler will correctly access all members required.

Where more than one routine is in a member, each should be surrounded by IFREF's/ENDIF and each should have an entry in the member directory (you must use the MAP option of PaDS to provide multiple entries to a member). This will benefit by not having needless routines appear in your object code output. For example, the following depicts two routines stored as one member in a PaDS.

```
; Entry for routine entitled "MOVE"
        IFREF  MOVE
MOVE    .           ;Routine of code
        .
        .
        ENDIF
; Entry for routine entitled "SHIFT"
        IFREF  SHIFT
SHIFT   .           ;Routine of code
        .
        .
        ENDIF
```

If your source code references "SHIFT" but not "MOVE", as long as both "SHIFT" and "MOVE" are member entries in the library PaDS directory, a "*SEARCH" of the library will access the member and assemble only the "SHIFT" routine. You should read the section on the "IFREF" conditional in the chapter on ASSEMBLER PSEUDO-OPS to understand the evaluation of the "IFREF".

Finally, it should be obvious that the "*SEARCH filespec" directive should appear in your source code near the end of the source (i.e. after all references to modules in the PaDS library have occurred).

What is a MACRO?

In virtually all programs, you find particular sequences of code that are repeated. These sequences may be termed routines. They could be so short that the overhead needed to set them up as CALLable routines is ineffective. Or, they could be longer routines that just cannot be constructed as CALLable segments. You may even want a code sequence to be an in-line assembly in contrast to a CALLable routine for the purpose of fast execution. The most useful function is to be able to have parameterized routines - algorithms that operate on different values each time the algorithm is invoked.

There are a few ways to deal with routines that are repeated in a program. You could block copy it from the first appearance to wherever you needed the routine. Or you could establish the routine as a macro. The first method could take up more source storage than is desirable. Also, if you decide to change the routine's algorithm, having many copies in a program can be cumbersome to update.

The second method mentioned is the use of macros. Consider the following commonplace sequence of code:

```
        LD      HL,VALUE
        LD      (MEMORY),HL
```

How many times is this little sequence repeated in your programs? Five? Ten? If we set up a macro near the beginning of our program that looked something like this:

```
STOR MACRO  #VAL,#MEM     ;Macro to store "VAL" into memory
     LD     HL,#VAL       ;Get value into HL
     LD     (#MEM),HL     ;Load value into memory
     ENDM                 ;End of the macro
```

we could perform the above two statements with one macro call as follows:

```
        STOR   VALUE,MEMORY  ;Invoke the macro
```

The first part of the example, defines a macro called "STOR". This is done exactly once per program! If we save our macros in a macro source file, each of our programs could "*GET MACROS"; thus, we would not have to even manually enter the macro into each program.

We invoke the statements defined in the macro by specifying the macro name AS IF IT WERE AN OPCODE. Using the macro invocation method, we can save storage space and introduce structured techniques to our coding. Notice that we have used some fictitious names when the STOR macro was defined. These names are called "dummy" parameters. They serve to provide a means to pass actual parameters when the macro is invoked. Through the dummy parameters, the real power of the macro is utilized. During the macro invocation, the model statements are expanded with substitutions for the dummy parameters that are provided in the macro call.

MACRO Definition

The format for a macro definition is illustrated as:

```
MOVE    MACRO   #parm1,#parm2=dflt2,#parm3
        LD      HL,#parm1
        LD      DE,#parm2
        LD      BC,#parm3
        LDIR
        ENDM
```

The macro definition consists of three parts: a macro prototype, a macro model, and the ENDM statement. The prototype is used to specify the macro name and the dummy parameter names used in the model. Default substitutions may be specified in the prototype to be used if the corresponding parameter is not passed in the macro invocation. The macro model contains all of the assembler statements to be generated when the macro is invoked. The model is sometimes called the macro skeleton or template. The dummy parameter names occupy the positions where the actual parameters will be placed by the macro processor in EDAS. The third part, the ENDM statement, is used to indicate the end of the macro model.

When a macro is defined, it is not assembled into your program. The macro prototype is parsed and analyzed. The macro definition is then stored in a compressed format within the macro storage area. Comments appearing with the macro definition are not stored if the comment starts with a double semi-colon in lieu of a single one. Comments with a single semi-colon are thus carried through a macro expansion to the listing.

Macro definitions may be nested. The inner macro will not become defined until the outer macro is expanded during an invocation. However, since macros cannot be redefined, the outer macro should be invoked only once!

Macro Prototype

Macros are named just like symbolic labels. The same rules apply. The number sign "#" is used to denote a parameter in the macro prototype; however, its use is optional. It is still required in the macro model to indicate the start of a parameter name. The length of macro names can range from <1-15>. Special characters <@, $, _> may be used in the name construct. Do not use the question mark in macro names as it would conflict with the symbol substitution string use made of "?".

The MACRO pseudo-OP is used to define the prototype of a macro model. Its syntax is:

```
mname  MACRO  {#parm1}{=dflt1}{,#parm2{=dflt2}}{,...}

mname       is the macro name used to invoke the macro.

#parmn      are dummy parameters of the macro which will
            be replaced by actual parameters during the
            macro invocation. "#" is an optional prefix.

dfltn       are optional default strings to be used for
            the dummy parameters when a parameter is not
            provided in the macro invocation.
```

The upper limit on the number of macro parameters is 127; however, you can not exceed the length of a standard assembler source statement. Thus, the statement length becomes the limiting factor. As is the case with macro names, the rules for naming dummy parameters are identical to the rules for labels. If a macro parameter is enclosed in angle brackets, the entire string which is enclosed within brackets will be treated as one parameter - even if it contains separator characters. Neither the macro names nor the "dummy" names are included in the symbol table generated by EDAS, thus there is no restriction on reusing the same name as a "dummy" for a label; however, to avoid confusion, it is recommended that you avoid using dummy names as symbolic label names.

Default strings can contain any character except the comma, ",". The comma is used as a field delimiter. There is no limit to the length of a default string other than the limiting factor of the statement length.

Macros must be defined prior to use but can be defined in a separate disk file accessed via a "*GET filespec".

MACRO parameters are acceptable within a quoted string if prefixed by an ampersand. i.e. TEST DB '&#NAME'. See the following example.

```
5200          00002 FEED    MACRO   #STRING
5200          00003 $?1     JR      $?2
5200          00004 LABEL?  IRPC    XX,#STRING
5200          00005 LABXX   DB      '&XX'
5200          00006         IFGT    $-LABEL?,3
5200          00007         EXITM
5200          00008         ENDIF
5200          00009         ENDM
5200          00010 $?2     LD      HL,LABEL?
5200          00011         ENDM
5200          00012         FEED    012345
5200+1806     00013 $A1     JR      $A2
              00014 LABELA  IRPC    XX,012345
5202+         00015 LABXX   DB      '&XX'
5202+         00016         IFGT    $-LABELA,2
```

```
5202+          00017          EXITM
5202+          00018          ENDIF
5202+          00019          ENDM
5202+30        00020 LAB0     DB      '0'
               00021          IFGT    $-LABELA,2
               00022          EXITM
               00023          ENDIF
5203+31        00024 LAB1     DB      '1'
               00025          IFGT    $-LABELA,2
               00026          EXITM
               00027          ENDIF
5204+32        00028 LAB2     DB      '2'
               00029          IFGT    $-LABELA,2
               00030          EXITM
               00031          ENDIF
5205+210252    00036 $A2      LD      HL,LABELA
0000           00037          END
```

Macro Model

Any valid Z-80 statement, EDAS pseudo-OP, or assembler directive (except *GET or *SEARCH) is valid in the macro model.

ENDM pseudo-OP

This pseudo-OP is used to specify the scope of a macro model. It is used much like ENDIF. Its syntax is:

```
mname  MACRO  parms
       model statements
       ENDM
```

The ENDM pseudo-OP must be used to let the macro processor know what is the last macro model statement. If macros are nested, each must have an ENDM.

EXITM Pseudo-OP

This pseudo-OP can be used to prematurely exit from a MACRO expansion. This is normally used within a conditional clause. One level of conditional nesting will be removed (if any are present). See the example for IRP.

Macro Definition Examples

This macro will move a block of memory from one location to another. If the "length" parameter is omitted, then a value of "255" will be used:

```
MOVBLK  MACRO  #FM,#TO,#LEN=255
        LD     HL,#FM
        LD     DE,#TO
        LD     BC,#LEN
        LDIR
        ENDM
```

This is a macro to clear a region of memory (i.e. set to 0). This macro will invoke the MOVBLK macro in a nested invocation:

```
CLRMEM  MACRO  #BUF,#LEN=255
        LD     HL,#BUF
        LD     (HL),0
        MOVBLK #BUF,#BUF+1,#LEN
        ENDM
```

This macro will add the 8-bit register "A" to 16-bit register pair "HL":

```
ADDHLA  MACRO
        ADD    A,L
        LD     L,A
        ADC    A,H
        SUB    L
        LD     H,A
        ENDM
```

A macro is not required to contain dummy parameters as is evidenced by the last example.

Incorporating Conditionals

Conditional pseudo-OPs can be specified in macro models. For instance, say you want the MOVBLK macro to be able to perform a non-destructive move (a destructive move would be where the destination is an address between "from" and "from+length-1"). You can insert conditional pseudo-OPs to test the parameters during the assembly of the expansion. Don't forget that the actual labels substituted for parameters must be defined prior to invoking the MACRO! Then, only certain segments of the macro will be assembled according to the result of the evaluation. Analyze the following example:

```
MOVBLK MACRO   #FM,#TO,#LEN=255
       IFNE    #FM,#TO          ;Don't expand if #FM=#TO
       LD      BC,#LEN          ;Establish the length
       IFGT    #FM,#TO          ;Do we LDIR or LDDR?
       LD      HL,#FM           ;#FM > #TO => LDIR
       LD      DE,#TO
       LDIR
       ELSE
       LD      HL,#FM+#LEN-1    ;#TO > #FM => LDDR
```

```
        LD      DE,#TO+#LEN-1
        LDDR
        ENDIF
        ENDIF
        ENDM
```

MACRO Nesting

The CLRMEM example depicts a macro that nests a macro invocation. Macros may be nested to seven (7) levels. That is, at any time, macro expansions for 7 macros called in a chain can be pending. For example:

```
ABC   MACRO    #PARMS,...
      (model statements)
      MOVE     parm,parm   ;call macro "MOVE"
      (model statements)
      ENDM
MOVE  MACRO    #parm1,#parm2,#parm3
      (model statements)
      ENDM
```

is perfectly legal. The expansion of the "MOVE" macro is not performed during the definition of the "ABC" macro but rather during the invocation of "ABC".

Macro definitions also may be nested. The inner macro will not be defined until the outer macro is expanded. For instance:

```
ABC   MACRO     #PARM
      (model statements)
XYZ   MACRO     #PARMs,...
      (model statements)
      ENDM
      ENDM
```

is a legal macro definition. The inner macro (XYZ) will not be defined until the outer macro (ABC) is invoked. Note the two ENDM statements.

If macro A "calls" another macro, say B, any dummy parameter in the macro call of B that matches a dummy in macro A, will be considered part of macro A and the parameter substitution will be invoked by the parameter passed when the user calls macro A.

MACRO Invocation

The invocation of a macro is termed a macro "call". The macro processor then proceeds to replace the call with the model statements specified when the macro was defined. The replacement of the macro call by the macro model statements is termed the macro "expansion".

During the expansion, the "actual" parameters passed in the call statement are substituted for the "dummy" parameters which appear in the macro model and which are designated in the prototype of the macro. Note that the actual parameter values are character strings and can be labels, expressions,

or data constants. An actual parameter can even be a quoted string data declaration if its use is designed into the macro model.

The entire expanded macro model is listed during the listing pass (phase two). Macro expansions in the listing will be so noted by the appendage of a plus sign immediately following the line number displayed. You may find that you don't really want to see these expansions since the macro definition contains the entire illustration of the macro. An assembler switch, "-NM" is provided to suppress listing of macro expansions. In the case of nested macro calls (i.e. a macro is defined which calls another macro which was separately defined), only the primary macro call will be listed if the "suppress" switch is invoked.

The substitution of the actual character string parameters for the dummys occurs during the macro expansion when the macro is called. Since a macro can have more than one parameter, it is necessary to have a procedure that specifies which actual parameter corresponds to each dummy parameter. There are two methods supported in EDAS. Parameters can be passed to the macro expansion when calling by either position or keyword.

Positional Parameters

"Positional" parameters are correlated by the position they appear in the macro call. For example, if the "MOVBLK" macro was called with:

```
MOVBLK  VIDEO,CRT_BUFFER,CRT_SIZE
```

then the substitution string "VIDEO" would replace every appearance of "#FM", the string "CRT_BUFFER" would replace every appearance of "#TO", and "CRT_SIZE" would replace the dummy parameter, "#LEN". Note that actual strings are positionally correlated with the positions of the dummy parameters in the macro prototype.

If you wish to omit an actual parameter in a macro call, then you must supply the comma to denote its place. For instance:

```
SHIFT  4200H,,100H
```

omits the middle of three parameters. Generally, a default would have been provided in the macro definition.

Keyword Parameters

If the number of parameters is large, it is sometimes burdensome to remember the order of the parameters, or to provide the correct number of commas if a series of parameters are omitted. These drawbacks are remedied by the use of "keyword" parameters. The macro call parameter list can identify the actual parameters by using the name of the dummy parameter as well. The keyword syntax is:

```
|                                                                |
|                  #dummy=actual parameter                       |
|                                                                |
|          mname  #parm2=actual2,#parm3=actual3                  |
|________________________________________________________________|
```

If the previous macro call was invoked by keyword parameter specification, it could look something like this:

```
        SHIFT  #LEN=100H,#FM=4200H
```

Mixing Positional and Keyword Parameters

A single macro invocation can intermix both positional and keyword parameters. The point that needs clarification, is what positions are actually denoted in the parameter list. It is simply treated. In a mixed parameter list, keyword parameters are ignored when considering place positions. For example, in the following macro call:

```
        SHIFT  #LEN=100,BLOCK,BUF_START
```

even though the length parameter appeared first in the parameter list, since it was designated as a keyword, it is ignored from the positional count and "BLOCK" is the first parameter with "BUF_START" second. In a similar manner:

```
        COMP   PARM1,#P6=2,,PARM3,#P8=38,PARM4
```

"PARM1" is in position one, the second parameter is omitted (the double comma), "PARM3" and PARM4" are in the third and fourth positions respectively. The sixth and eighth parameters have been entered by keyword.

Note that the parameter list contains five parameters. Thus if you were to use the "%%" operator which returns the number of parameters passed in a macro call ("%%" is described later), it would return a value of five.

Local Labels

So far, all of the examples have shown macro models without labels. What would happen if we had a macro defined as follows:

```
    FILL MACRO  #CHAR,#NUM
         LD     B,#NUM
    FLP  LD     (HL),#CHAR
         INC    HL
         DJNZ   FLP
         ENDM
```

We would have a problem because every time the macro was called, the label, "FLP", would be used. If "FILL" was invoked more than once, the assembler would generate MULTIPLY DEFINED SYMBOL errors on each expansion. We have to be able to use labels, but we need to find a way to be able to make "unique" labels on each macro expansion.

EDAS provides a facility for doing this by keeping a substitution string which is changed each time a macro is expanded. The string replaces the question mark character, "?", during a macro expansion whenever it appears outside of single quotes in a macro model statement. Each time a macro is expanded, the string will be changed. The string starts with the single letter "A", changes to "B", ..., "Z", then increments to the two-letter strings, "AA", "AB", ..., "ZZ", then to three letter strings, AAA-ZZZ each time a macro call is made. By using the question mark as one of the characters in symbols of a macro model statement, it will uniquely identify labels local to a macro. You may want to standardize the way you create labels to ensure that uniqueness is maintained. For example, you may use macro labels of the form, "$$?1", "$$?2", ... You can repeat the use of "$$?1", "$$?2", ... in another macro since the substituted string will be unique for each macro expansion.

The substitution string will be different from the *MOD directive substitution but is similarly used. Macro expansion substitution of "?" takes precedence over *MOD substitution. In the case of nested macros, each nest level will have its own unique substitution.

By using the question mark string substitution specifier, the previous macro would be defined like this:

```
FILL MACRO  #CHAR,#NUM
     LD     B,#NUM
$$?1 LD     (HL),#CHAR
     INC    HL
     DJNZ   $$?1
     ENDM
```

String Comparisons

It is sometimes desirable to be able to test within a macro model, the exact string passed as a parameter. Four conditional pseudo-OPs have been added strictly for string comparisons within macro processing. These are:

IFLT$	string1,string2	TRUE if string1 < string2
IFEQ$	string1,string2	TRUE if string1 = string2
IFGT$	string1,string2	TRUE if string1 > string2
IFNE$	string1,string2	TRUE if string1 <> string2

These pseudo-OPs provide TRUE/FALSE evaluation in the comparison of string1 to string2 (like the non-"$" pseudo-OPs do with mathematical expressions). Obviously, hard encoding of both string1 and string2 would be nonsense! Aha, he said... If we use a macro dummy parameter, it will be substituted by the actual parameter string passed in the macro call expansion. This means that the macro itself can test the parameter string in a limited manner. For example:

```
IFNE$   #TO,(DE)
LD      DE,#TO
ENDIF
```

as part of a macro model, will have the "#TO" replaced during the expansion. The test becomes dynamic! The dummy parameter can be either string1 or string2 - it doesn't matter.

These string conditional pseudo-OPs can only be useful in macros. That's because the evaluation, to make sense, has to be dynamic.

Testing String Lengths

Another feature available in the macro processor is the per cent sign "%" operator. This operator is used to recover the length of the passed parameter string and the number of parameters passed in the macro call. Note that the limitation for the use of the "%" operator, is that it is acceptable only for parameters of the current macro expansion. That means that you can't test for lengths outside of the current macro if you are nesting macro calls (macros cannot be recursive!). The operator can be used like these examples:

```
LD   B,%#PARM          ;loads B with the length of #PARM

IFGT %#PARM1,6         ;Restricts parm1 to a length <1-6>
ERR  Parm too long!
ENDIF

IFLT  %%,4             ;This macro requires 4 actual parms
ERR   Missing required parameters!
ENDIF
```

The "%%" operator will return the number of parameters passed in the current Macro call. When a dummy parameter name (including the "#" prefix) follows the per cent operator, the length of the parameter string is returned.

These values can be tested arithmetically to produce a TRUE/FALSE result (as was just demonstrated), or they can be used directly to represent logic TRUE/FALSE conditions. Realizing that if a parameter was not passed in the parameter list of the macro call, its length would be zero. A zero is also a logical FALSE. EDAS will accept as TRUE, any non-zero value (in normal use of TRUE/FALSE specifications, "-1" is recommended for TRUE to maintain proper evaluation of the ".NOT." operation). Thus, the string lengths can be minimally used to test if the parameter was not passed (%#parm=0=FALSE) or the parameter was passed (%#parm<>0=TRUE).

Concatenating MACRO Labels

You can concatenate a string to a dummy parameter name by connecting it with the concatenation operator, "%&". For instance, the model statement:

```
        IFREF    #NAME%&L
```

will have the "#NAME" replaced by the MACRO call substitution string appended with the letter "L".

Special in-line MACROs

EDAS supports the standard INTEL macro operations of REPT, IRPC, and IRP. These macro operations immediately expand the model statements according to specifications in the macro prototype statement. They may also be an interior macro of a nested macro definition.

Macro REPT

The statements within REPT-ENDM are repeated according to the result of "expression". The syntax of this macro is:

```
        label  REPT <expression>
               statements
               ENDM
```

In the prototype statement, the angle brackets are not required. See the following example which generates values from 1 through n where "n" is controlled by the value passed as "#COUNT" in the DOIT invocation.

```
5200           00002 DOIT    MACRO   #COUNT
5200           00003 T       DEFL    0
5200           00004         REPT    #COUNT
5200           00005 T       DEFL    T+1
5200           00006         DB      T
5200           00007         ENDM
5200           00008         ENDM
5200           00009         DOIT    3
0000+          00010 T       DEFL    0
               00011         REPT    3
5200+          00012 T       DEFL    T+1
5200+          00013         DB      T
5200+          00014         ENDM
0001+          00015 T       DEFL    T+1
5200+01        00016         DB      T
0002+          00017 T       DEFL    T+1
5201+02        00018         DB      T
0003+          00019 T       DEFL    T+1
5202+03        00020         DB      T
```

Macro IRPC

The statements within IRPC-ENDM are repeated for each character in the character list while the "identifier" is replaced with each character in turn from the identifier list. The identifier can be a multi-character string which is not a reserved word. This macro's syntax is:

```
        label  IRPC identifier,character-list
               statements
               ENDM
```

See the following example which generates values from 1 to 3.

```
              00002          IRPC    X,123
5200          00003          DB      X
5200+         00004          ENDM
5200+01       00005          DB      1
5201+02       00006          DB      2
5202+03       00007          DB      3
```

Macro IRP

The statements within IRP-ENDM are repeated for as many items as are in the argument list with "dummy" being replaced by each argument in turn. The angle brackets surrounding the argument list are mandatory. Its syntax is:

```
        label  IRP <dummy>,<arg1,arg2,...,argn>
               statements
               ENDM
```

where label is an optional statement label. See the following example which generates values from 1 to 3 and makes use of the EXITM escape.

```
              00003 LABEL    IRP     XX,<1,2,3,4,5>
5200          00004 LABXX    DB      XX
5200          00005          IFGT    $-LABEL,3
5200          00006          EXITM
5200          00007          ENDIF
5200+         00008          ENDM
5200+01       00009 LAB1     DB      1
              00010          IFGT    $-LABEL,3
              00011          EXITM
              00012          ENDIF
5201+02       00013 LAB2     DB      2
              00014          IFGT    $-LABEL,3
              00015          EXITM
              00016          ENDIF
5202+03       00017 LAB3     DB      3
              00018          IFGT    $-LABEL,3
              00019          EXITM
              00020          ENDIF
```

Command Summary

The appearance of the prompt symbol, ">", indicates the "command mode" of the Editor Assembler. Editor Assembler commands may be typed after the prompt symbol. The following list summarizes the commands recognized by the Editor Assembler:

A <A>ssemble source currently in the text buffer.
B <B>ranch to a specified address or return to DOS.
C <C>hange string_1 to string_2.
C <C>opy a block of lines to another location.
D <D>elete specified line(s).
E <E>dit a specified line of text.
F <F>ind a specified string of characters.
H Provide a printout of a specified range of text buffer lines.
I <I>nsert source text.
L <L>oad a source text file from disk.
M <M>ove a block of text from one location to another.
N Re<N>umber source text lines in the text buffer.
P <P>rint source text to the display device.
Q <Q>uery a directory from the designated drive.
R <R>eplace lines currently in the text buffer.
S <S>witch the upper case/lower case conversion mode.
T <T>ype source text lines without line numbers to a printer.
U Display the memory <U>tilization.
V <V>iew a file without loading it into the text buffer.
W <W>rite the current text buffer to disk.
X e<X>tend the text buffer by eliminating the Assembler.
Z Command reserved for user.
1 Alter printed lines per page and page length.

UPARW Scroll up one source text line.
DNARW Scroll down one source text line.
LTARW BACKSPACE key
RTARW TAB key
SRARW Page forward one screen.
PAUSE Performs a functional pause of any operation: <SHIFT @ (Model I/III)>

<A>ssemble

The <A>ssemble command is used to invoke the assembly of your source stream from memory and optionally, disk files (when "*GET filespec" or "*SEARCH library" is used in the source stream). The <A>ssemble command is also used to create a cross reference data file for downstream processing by the XREF/CMD program which will create a complete symbol cross reference listing. The syntax of the <A>ssemble command is:

A {filespec1/CMD}{,filespec2/REF} {-switch {-switch}...}

filespec1	**is the filespec to be used for the object code file generation. If the file extension is omitted, "/CMD" will be used (see -CI).**
filespec2	**is the filespec to be used for the cross reference data file. If the file extension is omitted, "/REF" will be used.**

Switches:

-CI	**Generates a Core-Image object file.**
-IM	**Assembles the object code into memory.**
-LP	**Generates a Listing to the Printer.**
-MF	**Search macro table before OP code table.**
-NC	**Suppresses FALSE conditional clauses.**
-NE	**Suppresses data declaration expansions.**
-NH	**Suppresses the object file header record.**
-NL	**Suppresses the assembly listing pass.**
-NM	**Suppresses MACRO expansions.**
-NO	**Suppresses object code generation (MAS).**
-SL	**Suppresses symbol table local label listing.**
-WE	**Pauses the assembly listing on an error.**
-WO	**Generates an object code output.**
-WS	**Generates a symbol table listing.**
-XR	**Generates a cross reference data file.**

The <A>ssemble command can be used to generate object code into either an executable object code file (/CMD) or a binary core-image object code file

(/CIM). Your program can also be assembled directly into the unoccupied memory region when the memory locations to be occupied by your program are not in conflict with storage areas of the assembler, your resident source code, the MACRO storage area, or the symbol table.

The source text to be assembled can exist either in memory only, or a combination of memory and disk files. The in-memory source is considered to be in the "text-buffer". When your source program is too large to be contained solely in the text buffer, it needs to be segmented into a combination of a memory segment and one or more disk file segments. The disk file segments are accessed during the assembly process by use of the "*GET filespec" assembler directive (detailed instructions concerning the use of *GET, are contained in the section entitled "ASSEMBLER DIRECTIVES").

The following paragraphs describe the command line entries and switch options in detail. Please note that if the EDAS e<X>tend command has been invoked, the <A>ssemble command will be inoperative.

Filespec1

The first filespec on the command line, identified as "filespec1", is the filespec to be used for the object code file. Its entry is entirely optional. When an object code filespec is entered, its entry will automatically invoke the generation of the object code to the disk file. Another method can also be employed to invoke object code generation to a disk file by means of the "-WO" switch (see below). If your filespec entry omits the file extension, the default of "/CMD" will be used. This default is changed to "/CIM" if the "-CI" switch is specified. It is recommended that you let the assembler assign the file extension, automatically. It will help to keep your directories orderly, and there will be less danger of overwriting a source file with the object code file.

Filespec2

The second filespec on the command line, noted as "filespec2", identifies the filespec to be used when writing the cross-reference data. The cross-reference data generation is optional - it is required in order to run the XREF/CMD program. EDAS will assign a default file extension of "/REF" if you omit the extension from your filespec. As XREF/CMD will also use this extension when accepting the file specification, it is suggested that you let EDAS assign it. You can also invoke generation of cross-reference data by using the "-XR" switch (see below). EDAS requires the entry of the comma to recognize the cross-reference filespec as "filespec2". Therefore, if you want the cross-reference data file but not the object code file, then either start the command line with the comma separator or use the XR switch without entering the filespec with the command line.

Switch -CI

The "-CI" switch is used to generate a "core-image" object code file. Executable command files in DOS are constructed with address information that the system loader uses when loading and executing your command file. Also, a header record is usually found in a load module object code file. There are times when you would prefer an object code file without this "load" and "comment" data. For example, say you want to burn a Programmable Read Only

Memory (PROM) from a file. A core-image file is needed. When the "-CI" switch is specified, a number of changes take place in EDAS. First, the object code file default extension is changed to "/CIM" (note: you must still enter the filespec or the switch "-WO" to invoke object code generation). Next, the header record and the transfer address record are suppressed. Any COM pseudo-OP statement is, likewise, suppressed. A core-image file needs to contain contiguous address sequential code. Since EDAS reserves only storage locations when assembling the DS/DEFS pseudo-OPs, the DS instruction would cause your object code file to be non-contiguous. Invoking the "-CI" will automatically convert all "DS" statements to their corresponding "DC" statements with a zero value for operand2.

Switch -IM

This switch will invoke object code generation; however, instead of the code being written to a file, it is placed into memory starting at the address specified as the operand of the "ORG" pseudo-OP. The "-IM" switch will override the entry of the "-WO" switch or entry of "filespec1". That is, if both "-IM" and "-WO" (or filespec1) are entered, assembly into memory will occur and assembly to disk will NOT take place.

Your program will not be permitted to overwrite any region below the end of the text buffer (or macro storage area if macros are being used) nor will it be permitted to overwrite the symbol table stored in high memory. The error message,

Memory overlay aborted

will be displayed if your assembled program will violate these restrictions. The assembly will be immediately stopped and EDAS will return to the command ready prompt. Upon successful completion of the assembly to memory, the message,

Memory region loaded
XXXX is the transfer address

will be displayed. This does not mean that your program assembled without error - only that the object code generated did not interfere with the text buffer or tables created during the assembly process. The "XXXX" field in the second message will contain the transfer address of the program. It will be listed in hexadecimal.

Switch -LP

The "-LP" switch is used to send the assembler listing, error messages occurring during the assembly of your source code, and the symbol table listing (if specified by means of the "-WS" switch) to a line printer. EDAS assembler listings print 56 lines per page and send a form feed at the conclusion of the 56 lines. If you are generating a listing output and a properly paged display is desired, it is suggested that you set your paper to begin printing at the sixth line from the top of the page (which assumes paging parameters set at 56 print lines and 66 lines page length - the default). This will provide five blank lines for a top margin, and five blank lines for a bottom margin.

If you are using other than 11" form paper, use the EDAS command "<1>" to alter the paging parameters to suit the specifications of your printer.

All line numbers are output in a sequential order incremented by one for each line of logical output. Lines suppressed from display use up one line number for each line omitted [i.e. from *LIST OFF to *LIST ON; -NC statements; -NM statements].

Switch -MF

This switch is used to force the assembler to search the macro name table prior to searching the OP code table when checking the "string" contained in the OP code field. This allows you to name a macro the same as an OP code; thereby altering the code generation of a standard Z-80 mnemonic.

Switch -NC

Conditional assembly can greatly ease the maintenance of programs designed to work with multiple configurations of hardware. However, it is unnecessary to "see" the source statements within conditional clauses that are FALSE. This "-NC" switch is provided to suppress FALSE conditional clauses from appearing in your listings. If a conditional is suppressed, neither the "IF" statement nor the "ENDIF" statement of the FALSE clause will be listed except those statements in a macro definition.

Switch -NE

Data declaration pseudo-OPs create a structured format for the listing of code generated after the first byte of the statement. These are the DB/DEFB/DM/DEFM, DW/DEFW, and the DC pseudo-OPs. If you want to inhibit the expansion from the listing, specify the No Expansion, "-NE", switch.

Switch -NH

Object code files usually start off with a header record of X'05 06 xx xx xx xx xx xx'. The x's would be replaced with the first six characters of the object code filename (buffered with spaces). EDAS automatically generates this record when writing the object code file. If you do not want to have this header record generated, then specify the No Header, "-NH", switch.

Switch -NL

The second phase of the assembly process is used solely to generate the assembler listing. If you do not want to see a listing, then you may enter the No Listing, "-NL", switch. This will completely suppress phase two and shift the assembler to phase three, object code generation. If you are interested in listing statements containing errors, then you must not suppress the second phase. Note that only the lines containing assembly errors can be listed by specifying the "*LIST OFF" assembler directive. See the section on ASSEMBLER DIRECTIVES" for further details.

The cross-reference data file is written during phase two. In order to guarantee that the second phase is available, a cross-reference specification will automatically override any entry of the "-NL" switch. This could be useful during a job stream assembly (from Job Control Language) where

selected assemblies need the cross-reference data. Thus, your JCL could specify "-NL" for every assembly; whenever the XR option was invoked, phase two would not be suppressed.

Switch -NM

You may have realized that the macro model code is repeated whenever you invoke a macro. Once you become familiar with what the macro does, you really don't need to see its expansion in your listings every time the macro is invoked. Switch "-NM" has been provided to inhibit the listing of such expansions. If you specify No Macro expansions, only the statements invoking the macros will be listed - the listing of the expansions will be inhibited. In the case of a nested macro invocation, only the highest level macro call will be listed.

Switch -NO

The "-NO" switch is used by MAS to inhibit the generation of the object code file. Since EDAS does NOT generate object code unless you tell it to do so (by "filespec1", switch "-WO", or switch "-IM"), the "-NO" switch is ignored.

Switch -SL

If you specify "-SL", then any label starting with a dollar sign, "$", will be suppressed from the symbol table listing and from any cross-reference data file. Therefore, use of the "$" as the first character of local labels and specifying "-SL" will result in keeping your symbol table listings uncluttered with local labels - especially true with the LC compiler.

Switch -WE

In a long assembly, you may want the assembler to pause the listing if it detects an assembly error (you're bound to get some of them). The Wait on Error switch, "-WE", is available for that purpose. If specified, each time the assembler comes to an error during phase two, it will pause the listing. Any character entered from the keyboard will continue the assembly and listing. If you choose to enter the character "C" or "c", then the phase two process will "c"ontinue without further interruption - even though additional errors may be detected. The listing may also be paused at any time by depressing the <PAUSE> key, momentarily.

Switch -WO

As noted in a preceding paragraph, object code generation is specified when "filespec1" is entered. Assembled object code is also generated to disk if the With Object switch, "-WO" is specified. If "filespec1" has not been entered, the prompt message:

Obj filespec?

will be displayed. Enter the object code filespec that you want to use to save the assembled object code command file at this time. If you do not enter a file extension, the default "/CMD" will be assumed. EDAS will open the file if it is an existing file and display the message, Replaced, or create

the file if it is non-existent and display the message, New file.

If you enter "filespec1", it is not necessary to enter the "-WO" switch as entering the object code filespec will activate the "-WO" switch. If the switch, "-IM", is specified denoting an in-memory assembly, the "-WO" switch will be ignored.

Switch -WS

A complete symbol table cross-reference listing is available via the "-XR" switch and subsequent processing by the XREF/CMD program. Such a separate process is needed in order to be able to handle cross referencing of statements fetched from a *GET or *SEARCH file. An abbreviated printout that contains only a sorted listing of symbols and their value is available at assembly time by invoking the With Symbol switch, "-WS". The symbol table listing would normally be displayed on the video display. If the "-LP" switch was specified, the listing would be directed to the Line Printer.

Switch -XR

This is the switch option to use if you want to generate a complete symbolic cross reference listing. Switch "-XR" will invoke the generation of a reference data file used by the XREF/CMD utility. The reference data file is generated during the listing pass (phase two). If the XREF filespec is entered with the command line, this switch is assumed to have been entered. If the XREF filespec is not entered with the command line, the filespec of the reference file will be prompted for with the query,

XREF Filespec?

Respond with the filespec that you want to use to store the reference data. If you do not enter a file extension, the default "/REF" will be assumed. EDAS will open the file if it is an existing file and display the message, "Replaced" or create the file if it is non-existent and display the message, "New file".

Error totals

At the conclusion of phase three which generates object code, a listing of the total number of errors will appear. This error total will be displayed after the conclusion of phase two if object code is not generated. If you need to get a quick idea whether or not your source code contains errors, place an "*LIST OFF" pseudo-OP at the beginning of your code and omit any object code generation - but do not specify "-NL". Only lines containing errors will be listed. You could also specify switch "-WE" to pause when an error occurs.

A "No end statement" error is included in the ERROR TOTALS count. An "Unclosed conditional" error is also included in the ERROR TOTALS count. If the END statement is omitted, the ERROR TOTALS count figure will be correct. Note that error totals is omitted if pass 2 and pass 3 are suppressed.

<B>ranch

The <B>ranch command is used to exit EDAS. Since the <B>ranch command permits an address as an optional parameter, you can use it to jump to any address (the entry to an in-memory assembled program, for instance). The syntax of <B>ranch is:

```
B {address}

address     is the branch address entered in hexadecimal.
```

This command is used to exit the Editor Assembler or optionally branch to any user designated address. If a branch address is omitted, a return to the DOS Ready command mode is performed. If a branch address is provided, the top of the stack will contain a re-entry address to EDAS. This can benefit the testing of a program assembled into memory. A simple "RET" instruction in your program will return control to EDAS (provided your program maintained stack integrity and did not crash).

Examples of the <B>ranch command are:

```
B        "B" will cause an exit from EDAS and return to DOS

B 9000   This command will cause an exit from EDAS and branch
         to your program at X'9000'.

B 30     This will cause EDAS to enter DEBUG. The PC register
         displayed by DEBUG is the return address to EDAS.
```

<C>hange

The <C>hange command performs a global modification of a string of characters. Its syntax is:

```
C /string1/string2{/n1,n2}

string1     is the current string to change.

string2     is the replacement string for string1.

n1          is the line number of the line preceding the
            first change (FIND always starts at line+1).

n2          is the line number of the last line to change.

/           represents a string separator character. It
            can be any character except a digit <0-9>.
```

A string of characters can be changed throughout the text buffer by this one easy command. The global <C>hange command will change the appearances of "string1" to the sequence "string2". Because <C>hange uses the <F>ind command to locate strings and the <F>ind command always starts searching at "current line + 1", no changes can be performed on the first line of the text buffer - at least not with the <C>hange command. Also, only the first appearance of "string1" in each line that "string1" appears will be altered.

The first non-blank character following the "C" becomes the string delimiter (the slash character is shown above; any character except a digit <0-9> is permitted). Null strings are not permitted (i.e. the string must contain at least one character).

There is no requirement for "string2" to be the same length as "string1". It can be of lesser, equal, or greater length; however, no string can exceed 16 characters in length. If a change would result in a line exceeding the maximum line length (which is 128), the change will not be performed on that line and the message, "Field overflow" will be issued. The search for "string1" continues for the remaining lines.

A line which contains "string1" will be displayed as it exists both before and after the change. The <SHIFT-@> key may be used to pause the display. If you depress the <BREAK> key, it will stop further changing.

The entry of "n1" and "n2" is optional. If "n1" is entered, then "n2" must be entered. If neither "n1" nor "n2" is entered, then "n1" is assumed to be the beginning of the text buffer (# or t) and "n2" is assumed to be the end of the text buffer (* or b). Either "n1" or "n2" can be entered as the current line indicator (.). You can enter "n1" as (# or t) to indicate the beginning or top of the text buffer while "n2" can be entered as (* or b) to indicate the bottom of the text buffer. One additional restriction is that if you enter "n2" as "b" or "*", then no change will be made on the last line of the text.

When EDAS is set to the "lower-case converted" mode (see the information concerning the "<S>witch-case" command), both "string1" and "string2" will be converted to upper case characters prior to the search and replacement. If you need to change lower case characters as well, then you must switch EDAS to the "lower-case permitted" mode prior to issuing the <C>hange command.

The "tab" character is a perfectly acceptable character to be used within "string1" or "string2". This may be useful if you want to convert a contiguous sequence of spaces to a single tab.

Some examples of the <C>hange command are:

```
C /MODIFY/ALTER/    This command will change all appearances of
                    the string "MODIFY" to the string "ALTER".

C .DEFB.DB.90,1000  This command will change all appearances of
                    "DEFB" to "DB" from line 100 to line 1000.

C /DEFM/DB/90,b     This will translate all appearances of
                    "DEFM" to "DB" from line 100 to the end of the file.
```

<C>opy

The <C>opy command can be used to duplicate a line or block of lines from in the text buffer. Its syntax is:

C line1,line2,line3	
line1	**is the first line of the block to duplicate.**
line2	**is the last line of the block to duplicate.**
line3	**is the line number of the line that the copied block should follow.**

This command is useful to duplicate a line or block of lines. Note that the command letter is the same as the <C>hange command. EDAS will interpret the <C> as a <C>opy command if the first non-blank character following the <C> is a digit <0-9>. At the conclusion of the <C>opy operation, the entire text will be renumbered using the increment currently in effect. A few restrictions are in order. A <C>opy cannot be performed if "line3" is interior to the block "line1"-"line2". "Line1" must either precede "line2" or be equal to "line2" (where "line1" is equal to "line2", the block to be duplicated consists of the single line, "line1").

If insufficient space is remaining in the text buffer to duplicate the entire block, none of the block of lines will be copied and the message, "Text buffer full" will be displayed. The parameters (line numbers) must specify specific lines in the text buffer. If any of the line numbers cannot be found, the copy will not be performed and the message "No such line" will be displayed. The <C>opy command requires all three parameters entered and separated with the comma (,). If this syntax is not met, the message "Bad parameters" will be displayed.

Some examples of the <C>opy command are:

C 100,200,1000 This command will duplicate the block of lines numbered from 100 to 200 inclusive to also appear after line number 1000.

C t,50,50 This command will copy the block of lines from the top of the text through line number 50 so that it will also follow line number 50.

c 580,700,b This <C>opy command will duplicate the block of lines numbered from 580 to 700 so that they also appear after the current bottom of text.

<D>elete

The <D>elete command is used to remove a line or block of lines from the text buffer. Its syntax is:

```
D {line1{,line2}}

line1        is the first line to delete.

line2        is the last line to delete.
```

This command is used to delete the line or lines specified from the source text buffer. The characters "#" or "t" are used to indicate the beginning of the text buffer when used for "line1". The characters "*" or "b" are used to indicate the bottom of the text buffer when used for "line2". If the line parameters are omitted, the current line, "." is assumed.

To aid in you in observing what becomes the new current line after a line delete operation, the new current line will be displayed.

Some examples of line deletes are:

D 100,500	**This <D>elete will remove from the text buffer, lines 100 through 500 (inclusive).**
D T,B	**This command will remove the entire source text from the text buffer. "d t,b" and "d #,*" are equivalent forms of this delete.**
D or d	**This <D>elete command will remove the current source text line. A period, ".", may also be used to indicate the current line (i.e. "D.").**
D 105	**This command will delete the the single line numbered 105.**

<E>dit

The <E>dit command is used to invoke the line editor for purposes of making alterations to a single text line. Its syntax is:

```
E {line}

line        is the number of the line to edit.
```

This command permits the user to edit or modify any source text line. The syntax and function of all edit subcommands are similar to those implemented in the BASIC editor. If the optional line number is not entered, the

current line, ".", will be edited.

When using the line editor, it will always operate in the "lower-case permitted" mode. Therefore, you will need to pay attention to use of the <SHIFT> key when editing upper-case characters. However, once you complete your editing and exit the line editor, your line will be properly converted to upper-case as required if EDAS is in the "lower-case converted" mode.

The following table of Edit Subcommands are provided for a reminder of the common edit operations:

A	**Abort and restart the line edit.**
nC	**Change n characters.**
nD	**Delete n characters.**
E	**End editing and enter the changes.**
H	**Delete (hack) the remainder of the line and insert the following string. A line hacked to zero length will be automatically deleted when exiting the line editor.**
I	**Insert string.**
nKx	**Kill all characters up to the nth occurrence of x.**
L	**Print the rest of the line and go back to the starting position of the line.**
Q	**Quit and ignore all editing.**
nSx	**Search for the nth occurrence of x.**
<--	**Move edit pointer back one space.**
ENTER	**Enter the line in its presently edited form and exit the edit mode.**
ESCAPE	**Escape from any edit mode subcommand. The <SHIFT-UP-ARROW> key is the escape key on the Model I/III/4.**
SPACE	**Display the next character of the current line being edited.**

<F>ind

The <F>ind command is used to locate the next occurrence of a string of characters within a line. Its syntax is:

```
F {string}

string      is the character sequence to find.
```

The text buffer is searched starting at the current "line+1" for the first occurrence of "string". "String" can be from <1 to 16> characters in length. If more than 16 are entered, then any characters beyond the 16th will be ignored. If no string is specified, the search is the same as that of the last <F>ind command in which a string was specified (provided a global <C>hange command was not performed after the last <F>ind command). If the search string is found, the line containing it is displayed and the current line pointer, ".", is updated to point to the displayed line. If the string is not found, the message "String not found" is displayed and the current line pointer, ".", remains unchanged. A "P#" or "Pt" command can be used to position the line pointer to the top of the text buffer prior to use of the <F>ind command. Spaces and tabs are considered to be part of "string" and are thus acceptable for "finding".

Some examples of the <F>ind command are:

```
FWRITEWORD     This <F>ind command will locate the next
               appearance of the string "WRITEWORD".

F              This finds the next appearance of "WRITEWORD".
```

<H>ardcopy

This command lists a line or block of lines on a line printer to provide a "hard copy". Its syntax is:

```
H {line1{,line2}}

line1       is the line number of the first line to print.

line2       is the line number of the last line to print.
```

This command will print a line or a group of lines to a line printer. EDAS will print 56 lines to a page (see the discussion of the <1> command). If a properly paged display is desired, it is suggested that you set your paper to begin printing at the sixth line from the top of the page.

Some examples of the <H>ardcopy command are:

H #,* or H t,b **This command will print the entire text buffer.**

H 100,500 **This command will print lines numbered 100 through 500 inclusive.**

H. **This command will print the single line pointed to by the current line pointer, ".".**

H **This command will print 15 or 23 lines (depending on the DOS) starting with the current line.**

<I>nsert

This command is used to invoke the <I>nsert mode so lines can be input into the text buffer. <I>nsert's syntax is:

I {line#{,inc}}
line# **is the number of the line that the insert should follow.**
inc **changes the current increment to "inc".**
Note: use <BREAK> or <SHIFT-CLEAR> to exit

The Insert command is used to insert or add text lines into the text buffer. All lines of source text are entered with the use of the <I>nsert command. After using the <I>nsert command to specify where you wish to place new lines, the editor will generate the designated line number and allow the inserting of that numbered text line. After entering the first text line the editor will generate the next line number higher, as specified by your increment selection. Incremental line numbers will continue to be generated as long as there is room between lines or room left in the text buffer.

If a desired increment is not specified, the last specified increment is assumed. Period, ".", may be used for "line#" to indicate the current line or if "line#" is omitted, the current line will be assumed.

The <BREAK> key will allow you to leave the insert mode at any time. The <CLEAR> key also performs a functional BREAK. If you have entered the <BREAK> before depressing <ENTER> to complete the input of a line, that line will not get entered into the text buffer.

Some examples of the <I>nsert command are:

I 300,5 **This command will begin the text insertion to follow line numbered 300 and also change the increment to 5.**

IB **This appends new text to the end of the old text. It is the same as a "Pb" followed by an "I".**

<L>oad

This command is used to load a source file into the text buffer. Its syntax is:

L {filespec}
filespec is the filespec of the file to be loaded.

The <L>oad command will read the file denoted by the "filespec" into the text buffer. The text file will be concatenated to any text already in the text buffer. The file specification is composed of a FILENAME, optional EXTension, optional PASSWORD, and optional DRIVE reference as in:

FILENAME/EXT.PASSWORD:D

If you do not enter the "filespec", EDAS will prompt you for the file-spec. If you omit the file extension (EXT), a default extension of "/ASM" will be used thus saving keyboard input and at the same time providing for a standard file naming convention. If the "LC" parameter was specified in the EDAS command line, then "/CCC" will be used for the default. The EDAS parameter "EXT=ext" can be used to override the assigned default extension to that of "ext".

The <L>oad command will automatically handle a source file that is line-numbered and headered (EDAS Version III format), line-numbered and un-headered (EDTASM Series I format), or un-numbered and un-headered (EDAS format, text editor prepared files, or certain M-80 files). If the file being read is not line-numbered, EDAS will automatically number it as it loads. A line number counter is kept internally that advances by the current increment for each un-numbered line read. Thus, concatenation of source text via multiple loads of un-numbered source files will produce a sequentially numbered in-memory text. The line number counter is reset to its initial starting value by a warm-start or the <CLEAR> command function.

A line-numbered file is interpreted as one in which the first five characters of a line have the high-order bit (bit 7) set. The 5-character line number is also followed by a terminating character (usually a space but could be a tab with bit 7 set). A headered file is interpreted as one in which the first character of the file is a X'D3'.

"ASCII" files prepared by a word processor program (i.e. SCRIPSIT) are loadable by EDAS; however, they must be pure ASCII and must have line lengths not exceeding 128. The other requirement is that there must be an end-of-file (EOF) character as the last character of the text immediately following a carriage return. The EOF character can be either an X'1A' or a NULL, X'00'. EDAS can convert lower case to upper only during <I>nput or <E>diting so if you use an external word processor program, keep the Z-80 code in upper case.

Some examples of <L>oad commands are:

L myprog This command will search for a file named

"MYPROG/ASM" (assuming a default extension of "/ASM") and load it into the text buffer.

```
L theprog:1   This command will load the file named
              "THEPROG/ASM" from drive 1 into the text buffer.

Dt,b
L newprog:2   This sequence clears the text buffer then loads
              the file named "NEWPROG/ASM" from drive 2.
```

<M>ove

This command is used to <M>ove a line or block of lines from one text buffer location to another. Its syntax is:

```
M line1, line2, line3

line1       is the line number of the first line to move.

line2       is the line number of the last line to move.

line3       is the number of the line that the block
            should follow after the move.
```

This command is used to move a block of lines from one location in the text buffer to another. In the command syntax, "line1" and "line2" are the beginning and ending line numbers of the text block to be moved; they can reference the same line number when moving a single line. "Line3" is the line number of the line that the text block will follow after the move. The line number references must be offset by commas ",". Your line number parameters must specify existing lines in the text buffer. If any of the entered line numbers are non-existant, the message "No such line" will be displayed.

"Line3" is not permitted to equal "line1" or "line2" as that would represent an illogical move operation. "Line3" is not permitted to be a line interior to the range "line1" through "line2" as that would also be an illogical operation. The message "Bad parameter(s)" will be issued if your input violates any of these conditions.

The block of text to be moved is stored temporarily in the spare text region. If this region is not large enough to store the block, the message, "Text buffer full" will be issued. Try moving the block in smaller segments.

Upon completion of the move, all lines in the text buffer will be renumbered starting from 100 and incremented according to the line increment currently in effect. Renumbering is absolutely essential to perform proper operation of Editor Assembler commands and so it is done automatically.

An example of a <M>ove command is:

```
M 500,900,1510  You desire to move the block of text starting
```

at line 500 and ending at line 900 to follow line 1510. This command will perform the desired operation.

Re<N>umber

This command is used to re<N>umber the lines of text in the text buffer. Its syntax is:

N {line{,inc}}
line is the new first line number.
inc is the new increment.

The <N> command is used to renumber the lines in the text buffer. The first line in the buffer is assigned the number specified as "line". If "line" is not specified, it defaults to 00100. The remaining lines in the buffer are renumbered according to the increment "inc" or the previous increment in a re<N>umber, <R>eplace, or <I>nsert command if the increment was not specified. The current line pointer, ".", points to the same line as it did before the re<N>umber command was used, but the actual number of this line may be changed.

Some examples of line re<N>umbering are:

N This command will renumber the text to start with line number 100. The previous increment in effect will be used.

N5 This command will renumber the text to start with line number 5. It also uses the previous increment.

N10,5 This command will renumber the text to start with line number 10. The line increment is changed to 5.

<P>rint

The <P>rint command is used to display a line or block of lines to the video display. Its syntax is:

P {line1{,line2}}
line1 is the number of the first line to display.
line2 is the number of the last line to display.

The <P>rint command will display a line or a group of lines on the monitor screen. The current line pointer, ".", is updated to point to the last line displayed.

If "line1" is entered without entering "line2", then only "line1" will be displayed. If neither "line1" nor "line2" are entered, then the current line plus 14/22 additional lines for a total of 15/23 will be displayed (depending on the DOS).

Some examples of <P>rinting lines:

P #,*	**This command will display all lines in the text buffer. You may use the <PAUSE> function to temporarily halt the display from scrolling. "P t,b" is equivalent to "P #,*".**
P 100,500	**This command displays lines 100 through 500 inclusive.**
P .	**This command will display the line pointed to by the current line pointer. Only a single line will be displayed.**
P	**This command displays a full screen of lines starting with the current line.**

<Q>uery

This command can be used to invoke a DOS command. Its syntax is:

Q DOS-command
DOS-command can be any DOS library command.

<Q>uery is used to interface with the DOS while in the environment of the Editor Assembler. Any DOS library command can be accessed.

An example of a <Q>uery command is:

Q DIR	**This <Q>uery command will list the diskette directory to the display device.**

<R>eplace

This command can be used to replace a specified text line and automatically enter <I>nsert mode. Its syntax is:

```
R {line{,inc}}

line        is the number of the line to replace.

inc         is the new increment to be used.
```

The <R>eplace command only replaces the one line specified and then enters <I>nsert mode. If "line" is omitted, then the current line is assumed. If "line" exists, it is deleted and then <I>nsert mode is entered starting with that line number. If "line" doesn't exist, <I>nsert mode is entered just as if the <I>nsert command were invoked. If "inc" is not specified, the last increment specified by an <I>nsert, <R>eplace, or re<N>umber command is used. The current line pointer, ".", is always updated to the new current line.

If during subsequent INPUT of lines, the error message "No more room" is issued, it means that a line numbered "current" + "inc" already exists. It is suggested that you renumber the lines and continue your insertion after ascertaining the new line number assigned to the "current" line.

Some examples of <R>eplace commands are:

R — **This command will replace the current line.**

R 100,10 — **This <R>eplace command will start replacing lines beginning at line numbered 100 and enter <I>nsert mode with an increment of 10.**

R 100 — **This command will start replacing at line numbered 100 using the last specified increment.**

<S>witch Case Conversion Mode

This command is used to toggle the "case conversion mode" of EDAS. It will either permit the acceptance of both upper case and lower case, or invoke the automatic conversion of lower case to upper case where required. Its syntax is:

```
S

There are no parameters or options.
```

Command <S>witch will toggle the switch-case conversion of lower case to upper case. Two modes are available:

1. Lower case accepted: This permits entry in either lower case or upper case. Your input is preserved in whatever case it is entered. EDAS is suitable as a text editor in this mode. This is the mode used when entering LC C-language source text.

2. Lower case converted: This permits entry in either upper case or lower case. All lines are converted to upper case during <I>nput mode or when exiting the <E>dit mode. This mode should be used to input assembler source text. While in this mode, character strings within single quotes stay in their entered case. This will ensure that your string declarations are kept intact. Also, characters entered following a semi-colon are kept in their entered case. This permits the entry of comments in lower case which makes your source text much more "readable".

On entry to EDAS, the "lower case converted" mode is activated. Each entry of an "S" command will toggle the case mode and an appropriate message will be displayed.

Since the <I>nsert command mode converts to upper case, the <F>ind and <C>hange commands utilize the <I>nsert input and will also convert to upper case. You can <F> or <C> lower case by using the case switch toggled to "lower case permitted".

<T>ype

This command can be used to print a line or block of lines without the line numbers on a line printer. Its syntax is:

```
T {line1{,line2}}

line1       is the number of the first line to print.

line2       is the number of the last line to print.
```

The <T>ype command prints a line or block of lines onto the Line Printer. The current line pointer, ".", is updated to point to the last line printed. This command is much like the <H>ard copy command, except line numbers are not printed. Only the source text is printed.

For examples of the <T>ype command, see the <H>ard copy command. The two commands differ in that <T>ype omits the line numbers during the printing.

Memory <U>SAGE

This command is used to display certain statistics concerning the memory usage of your source text buffer. Its syntax is:

```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│  U                                                           │
│                                                              │
│  There are no parameters or options.                         │
└──────────────────────────────────────────────────────────────┘
```

This command will display the number of bytes of text buffer in use, the number of bytes spare and the first address available for assembly to memory (note that if macros are being used, the macro storage area extends from the address shown as the first address available for assembly and you will have to experimentally choose a higher address for an "in-memory" assembly).

This command is useful to ascertain requirements for storing the text buffer to disk. Note that a disk file, which is written in ASCII (un-numbered), will contain two (2) bytes less per text line. The 2 bytes represent the line number used in the storage format of text in memory versus text in an un-numbered ASCII file.

It also is useful when assembling into memory. Since the Assembler will not permit you to overwrite it or the text buffer, you will have to "ORG" your program in the free text buffer area. The first available address is output by this command (remember the note on macro storage).

An example of <U>sage output is:

```
30622 bytes spare
00000 bytes in use
8863H is the first free address
```

<V>iew

This command is used to list (display) a file on the video display device. Its syntax is:

```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│  V {filespec}                                                │
│                                                              │
│  filespec    is the filespec of the file to be displayed.    │
└──────────────────────────────────────────────────────────────┘
```

This command can be used to display any file without actually loading the file into the text buffer. No attempt is made to convert non-ASCII characters prior to displaying. Therefore, if the file is not an ASCII file, strange characters may be displayed. Use the <V>iew command primarily to display source files.

The output may be temporarily stopped by depressing the <PAUSE> key or may be interrupted and cancelled by depressing the <BREAK> key.

If you do not enter the filespec with the command line, it will be prompted for with the query:

filespec?

If you do not enter a file extension with the file specification, a default extension of "ASM" will be used unless the "LC" parameter was specified when entering EDAS. "LC" redefines the default specification to "CCC". Note that the default extension could also have been changed via the "EXT=ext" parameter.

<W>rite

The <W>rite command is used to save the contents of the text buffer into a disk file. Its syntax is:

W{+}{#}{$}{!hh} {filespec}
filespec — **is the filespec to be written.**
+ — **is an optional switch to write a source file created with a header record.**
# — **is an optional switch to write a source file with line numbers.**
$ — **is an optional switch to write a source file with line numbers terminated by X'89'.**
!hh — **is an optional switch to specify a end-of-file terminating byte of X'hh' other than X'1A'. Use "!!" to suppress the E-O-F byte.**

This command will write the text buffer to the file denoted by filespec. If no filespec is entered, you will be prompted for it in a manner identical to the <L>oad command. If you omit the file extension (EXT), a default extension of "ASM" will be used thus saving keyboard input and at the same time providing for a standard file naming convention. Remember, if you had specified "LC" or "EXT=ext" when you entered EDAS, the default source extension will be "CCC" or "ext" respectively.

The switches are used for compatibility in writing source files for use with other editors such as the M-80 editor, EDIT80, earlier versions of EDAS (3.4 and 3.5), and EDTASM. If more than one switch is used, the order is irrelevant. Use of the switch "+" will enable creating a file with a file header record (X'D3' followed by a 6-character filename).

If the source file is to contain line numbers, then the "#" switch should be used. This will write line numbers as five ASCII digits with the high order bit (bit-7) set. The line number is terminated with a space character (X'20'). The switch "$" generates a line numbered file the same as the "#" switch; however, the terminating character is written as a tab with bit-7 set (X'89'). Some versions of FORTRAN require the source file to be in this manner; thus, EDAS could be used to prepare source files for FORTRAN.

Finally, the "!hh" switch can be used to specify an end-of-file byte to be other than the standard X'1A' normally used by EDAS. For instance, specifying "!00" will change the E-O-F byte to X'00', the value used by SCRIPSIT. If instead of the two-character hexadecimal value, you enter a second exclamation point as in "!!", then no E-O-F byte will be written. Observe caution as EDAS can only properly load a file if the E-O-F byte is an X'1A' or an X'00'.

If the file denoted by "filespec" is non-existent, a file will be created and the message, "New File", will be issued. If the file denoted by "filespec" is an existing file, it will be replaced by the write operation and the message, "Replaced", will be issued.

Some examples of <W>rite commands are:

W parmdir1:3	**This command will write the current contents of the text buffer to the file, PARMDIR1/ASM:3**
W !00 doparm/jcl:0	**This <W>rite command will save the text buffer in the file, DOPARM/JCL:0. An E-O-F byte of X'00' would be used instead of X'1A'. Thus, EDAS was used to edit a Job Control Language file.**

e<X>tend

This command can be used to increase the area of the text buffer by eliminating the assembler. Its syntax is:

X
There are no parameters or options.

This command can be used to extend the text buffer area by moving the text over the Assembler portion of EDAS in memory. Approximately 8000 bytes are gained by this extend operation. It is useful if you are editing a large body of text or are dealing with a large assembly language source program. Since the capability of direct assembly from disk files is a function of the EDAS Editor Assembler, editing can be performed without the Assembler module of the program in memory. You, of course, will have to exit and reload the Editor Assembler for further assembling.

Another reason for the use of e<X>tend, is to handle those EDAS 3.5 files that now exceed the maximum text buffer size of EDAS version 4.x. It is

suggested that you keep your source files in smaller modules. The *GET capability provides great power in handling multiple source files in an assembly stream. You will thus find that a program made up of smaller modules of code is perhaps easier to maintain and just as fast to assemble.

Following the entry of the <X> command, the prompt, "Are you sure?", will be displayed. This is provided as a safeguard in case you inadvertently enter the <X> command. You must respond <Y> in order to complete the extension. Entry of any other character will abort the extend operation. A response to the query with a <Y> will move the current contents of the text buffer and reset all pointers to their proper value. Once the e<X>tend command is invoked, both it and the <A>ssemble command will be made inoperative.

<1> (ONE)

This command can be used to display or alter the current page formatting parameters of EDAS. Its syntax is:

1{n1{,n2}}
n1 is the number of lines to print per page.
n2 is the page length in lines.

For the Model I/III operation, this command can be used to alter the two paging parameters used by EDAS. The "n1" parameter specifies how many lines to print on a page before issuing a form feed. The "n2" parameter is set in the printer Device Control Block at X'4028' and represents the maximum printing lines on a page. EDAS initializes with "n1" set to 56 (57 on a Model III which starts counting from 1). Thus, 56 lines will be printed before sending a page eject. Either value can be changed with this command. If no parameter is entered, then the current values will be displayed.

Only the "n1" parameter can be used under TRSDOS 6.x. If FORMS is active, its LINES parameter must not be less than or equal to the value set by the "1" command. The default will be 56 printed lines per page. Page breaks generate a form feed character, X'0C'. For printers that do not support a form feed, use the FORMS filter under TRSDOS 6.x.

An example of the <1> command is:

```
1 46 51  This command will set the maximum page length to 51.
         The number of printed lines until a form feed is
         generated will be set to 46.
```

Scroll UP <UP-ARROW>

The "SCROLL UP" command displays the line preceding the current line and updates the current line pointer, ".", to point to the line displayed. If the

current line is the first line in the text buffer, it is displayed and period "." remains unchanged. "SCROLL UP" is an immediate command and must be the first character of a command line in order to be interpreted.

Scroll DOWN <DOWN-ARROW>

The "SCROLL DOWN" command displays the line following the current line and updates the current line pointer, ".", to point to the line displayed. If the current line is the last line in the text buffer, the last line is displayed and period "." remains unchanged. "SCROLL DOWN" is an immediate command and must be the first character of a command line to be interpreted.

Clear Screen

The <CLEAR> or <SHIFT><CLEAR> key (depending on DOS used) is used to perform a functional clear screen. "CLEAR" also performs a <BREAK> operation but cannot be used to interrupt output. "CLEAR" will reset automatic line numbering to its initial value of 100.

Pause <SHIFT-@>

The <PAUSE> key is used to pause the computer during a display, during any assembly, or Editor Assembler printing. When a pause is sensed, depression of any key except <PAUSE> or <SHIFT> will continue the operation paused. It is only necessary to momentarily depress the key as a pause function will be held pending as soon as the key is pressed.

BREAK

The <BREAK> key is used to terminate the <I>nsert mode. It is also used to abort an assembly in effect. It will also abort any disk I/O operation or display listing. <BREAK> will return EDAS to the command ready prompt, ">".

Page Forward

The <SHIFT-RT-ARROW> key is used to advance the display by 15/23 lines. This command is similar to the <P>rint command except that the display screen is cleared prior to displaying the 15/23 lines of source text.

User Patch Space - ZCMD

A 50-byte patch space is available for your use. A vector pointing to this space is located at X'5A09' (Model I/III), or X'2D09' (TRSDOS 6.x). If you place a routine in this space, it can be executed by entering a <Z> at command ready. <Z> currently has a RET instruction as the first byte.

Error Messages - General

The Editor Assembler recognizes three types of errors. These are:

Command	**This is an EDAS command syntax error. The error message is displayed and control is returned to command mode.**
DOS	**This is an operating system disk I/O error. The error message is displayed and control is returned to command mode.**
Assembler	**These errors may occur while executing an Assemble command. There are three types: terminal, fatal, and warning.**

DOS disk I/O errors can also be received during an assembly. When a disk I/O error occurs, the assembly will be aborted and control will be returned to EDAS command ready.

Three different types of assembler errors can occur. The types relate to the severity of the error. These types are:

Terminal	**Assembly is terminated and control is returned to command mode.**
Fatal	**Processing of the line containing the error is immediately stopped and no object code is generated for that line. Assembly proceeds with the next statement.**
Warning	**The error message is displayed and assembly of the line containing the warning continues. The resulting object code may not be what the programmer intended.**

Command Errors

Buffer full

There is no more room in the text buffer for adding text.

Bad parameter(s)

Any command line not entered according to the syntax appropriate for that command will generate this error message. Also, if you attempt to load a file that is not a valid source code file, this message may be displayed.

Illegal command

The first character of the command line entered does not specify a valid Editor Assembler command.

Line number too large

Renumbering with the specified starting line number and increment would cause line(s) to be assigned numbers greater than 65529. The renumbering is not performed. This message would also be displayed if you attempted to INSERT a line with a line number exceeding 65529.

No room between lines

The next line number generated by INSERT or REPLACE would be greater than or equal to the line number of the next line of text in the buffer. The increment must be decreased or the lines in the buffer renumbered.

No such line

The line specified by a command does not exist. The command is ignored.

No text in buffer

A command requiring text in the buffer was issued when the buffer was empty. The commands <L>oad, <I>nsert, <Q>uery, <S>witch, <B>ranch, <U>sage, <V>iew, e<X>tend, <.>, <Z>, and <1> can be invoked when the buffer is empty. All other commands require at least one line of text to be in the buffer.

String not found

The string being searched for by the <F>ind command could not be found between the current line and the end of the text buffer. This message will also be displayed at the completion of a global change command.

DOS Errors

The standard DOS error messages will be displayed if the DOS returns an error code after return from any disk operation. Consult your DOS operating manual for explanations of those errors. During most error handling, the abbreviated form of the error message will be displayed. If an I/O error is detected during an assembly, the long form of the error message will be displayed. This allows you to observe which file was affected by the error.

Any attempt to load or *GET a file that has a line longer than 128 characters will result in "Load file format error".

Terminal Errors

Memory overlay aborted

During an assembly to memory, a block of code was assembled that would load into a memory region other than the spare text buffer area. Your program

will not be permitted to load to an address below the end of the text buffer or above the symbol table. Use the Usage command to locate the first available memory address. If you are using MACROs, the first available memory address is indeterminate as the MACRO processor uses the memory area immediately following the text buffer for a MACRO model and string buffer storage area.

Symbol table overflow

There is not enough memory for the assembler to generate your program's symbol table. You have three options:

> **Remove comment lines and/or comments following Z-80 code operands. This may free up enough space to perform the assembly.**
>
> **Divide your program into two or more modules and assemble them using the *GET filespec directive.**
>
> **Extend the text buffer area, expand your source, then assemble it using the *GET filespec directive.**

***GET or *SEARCH error**

A "*GET filespec" or "*SEARCH library" assembler directive was found in a library member. A searched library cannot have "* GETs" or nested "*SEARCHes".

Member definition error: filespec(member)

This is a result of a fetched *SEARCH member not resolving the symbol reference invoking its fetch.

Fatal Errors

Bad label

The character string found in the label field of the source statement does not match the criteria specified under ASSEMBLY LANGUAGE INFO - LABELS.

Expression error

The operand field contains an ill-formed expression.

Illegal addressing mode

The operand field does not specify an addressing mode which is legal with the specified OPCODE.

Illegal opcode

The character string found in the opcode field of the source statement is not a recognized instruction mnemonic, assembler pseudo-op, or MACRO name.

Missing information

Information vital to the correct assembly of the source line was not provided. The OPCODE is missing or the operands are not completely specified.

Too many nested *GETs

***GET filespec nesting exceeds the number of levels supported. The *GET will be ignored.**

Unclosed conditional

The "END" statement or end of source was reached and an open "IF" conditional block was still pending. Your program is missing the closing "ENDIF".

ENDIF without IF

An "ENDIF" pseudo-op was detected without a corresponding conditional "IF" or "IFxx" in effect. The "ENDIF" will be ignored.

ELSE without IF

An "ELSE" statement was detected without a preceding "IF" conditional segment.

Filespec required

A *GET or *SEARCH directive was detected but the statement did not contain the required file specification. The *GET or *SEARCH will be ignored.

Bad parameter(s)

When output preceding a MACRO definition, it implies an error in the parameters of a MACRO.

Nested MACRO ignored

A macro definition statement was nested in the model of another macro.

Missing MACRO name

The name field of the macro definition statement did not contain the macro name. The macro will not be defined.

ENDM without MACRO

An ENDM pseudo-OP was detected while not in a macro definition phase. It will be ignored.

Too many parameters

In a macro call, the number of parameters passed exceeded the number defined for the macro. The macro call will not be expanded.

Too many nested MACROs

The number of pending nested macro calls exceeds the current nest level supported. The macro call will not be expanded.

MACRO forward reference

A macro call was detected prior to the definition of the macro. The macro call will not be expanded since gross phase errors would result.

Multiply defined MACRO

A macro definition statement was detected for a macro already defined. The subsequent definition will be ignored.

Warnings

Branch out of range

The destination of a relative jump instruction (JR or DJNZ) is not within the proper range for that instruction. The instruction is assembled as a branch to itself by forcing the offset to hex X'FE'.

Field overflow

A number or expression result specified in the operand field is too large for the specified instruction operand. The result is truncated to the largest allowable number of bits. This error would also be output during a global change if a resultant line would exceed 128 characters.

Multiply defined symbol

The operand field contains a reference to the symbol which has been defined in another line. The first definition of the symbol is used to assemble the line.

Multiple definition

The source line is attempting to illegally redefine a symbol. The original definition of the symbol is retained. Symbols may only be redefined by the DEFL pseudo-OP and only if they were originally defined by DEFL.

No END statement

The program END statement is missing. Note that if your program is missing the "END" statement, EDAS cannot detect an unclosed conditional. Also, be aware that if your program has a FALSE unclosed conditional, then the "END" statement will NOT be detectable - even if present.

Undefined symbol

The operand field contains a reference to a symbol which has not been defined. A value of zero is used for the undefined symbol.

Object File Format

The disk file object code format consists of a header record, an optional comment record, one or more load block records, and a transfer address record. The specific formats of these records are as follows:

Header Record

The file header record consists of the hex byte X'05' (record type) which indicates the header field of an object file. It is followed by the header length byte which indicates the length of the header data following. The length of the header data is constant in EDAS and is six bytes. The data is constructed as the first six bytes of the object code file name field and is filled out with spaces if the file name is less than six characters.

Comment Record

This record is optional. It is generated by the "COM" pseudo-OP. It consists of a record type byte of X'1F' followed by a length byte which is the length of the comment. The comment data, itself, follows.

Load Block Record

The load block record starts with a record type code of X'01' which indicates it is a load block. A 1-byte length is next. This indicates the length of the object code data plus the 2-byte block load address. The length is encoded as a modulo 256 value (object code length of 253 = X'FF', object code length of 254 = X'00', object code length of 256 will show as X'02').

The block length byte is followed by the 2-byte block load address which is the address that will be loaded with the first byte of the block.

Finally the object code block immediately follows for as many bytes as two less than the block length.

Transfer Address Record

The Transfer address record (TRAADR) starts with a record type of X'02'. An X'02' is written to indicate the length of the entry point address. This is then followed by the 2-byte entry point or transfer address generated from the label or constant in the operand field of the assembler source END statement. As is the case with all 16-bit data values, the TRAADR data has the low-order byte of the address followed by the high-order byte.

Source File Format

The source code file format used by EDAS has neither header nor line numbers. Headers and numbers are entirely optional and can be generated with appropriate switches in the <W>rite command. The formats are as follows:

Header Record

A header record as described under "Object file format" is optionally used for source files with the exception that the first byte is a hex X'D3'

(X'53' - with bit 7 set) to identify the file as source, immediately followed by a 6-character name (the name length byte is omitted). Files written with "W+" contain this header.

Text Lines

Text lines are written in ASCII each composed of an optional 5-character line number (bit 7 is set), a space, the text line, ending with an <ENTER> (X'0D'). Files written with the "W#" command incorporate both the 5-character line number and following space.

End-of-File Mark

The file's end is indicated by an end-of-file mark of X'1A' which would be in the first character position of a text line (or 1st byte of the line number if line numbered files are used).

Reference Data File Format

The reference data file is a compressed collection of data corresponding to each symbol definition and reference. The file contains a title record, and definition/ reference records. The format of these records is as follows:

Title Record

The title record is always present even though the assembler source file stream may or may not have supplied a TITLE pseudo-OP. The title record is 28 characters long. If the source files did not contain a TITLE pseudo-OP, the record will be filled with spaces.

Definition/Reference Records

These records contain the data for either a symbol definition or reference. It is composed of a filename field, a line number field, a type field, a value field (omitted for references), and a symbol name field. These fields are defined as follows:

Filename Field

This field will be either an eight character filename or a hex X'22'. If a hex X'22', then the filename reference is the same as the previous record.

Line Number Field

This field contains the line number of the definition or reference statement in low-order high-order form.

Type Field

The type field contains an X'00' for a reference, an X'01' for a definition, or an X'02' for a DEFL defined symbol.

Value Field

The value field contains the defined value of the symbol. This field is omitted for references (type field = 0).

Name Field

The name field contains the symbol name. It is terminated with a carriage return (X'0D'). If the symbol is the name of a macro, the first character of the name has the high-order bit set.

Linkage to Debugging (Model I/III or TRSDOS 6.x)

In order to facilitate the debugging of user generated programs, a number of features have been built into EDAS. It provides the option of assembling source code directly to memory. It provides a command to transfer control to a user-supplied address (via the <B>ranch command).

A re-entry address to the Editor Assembler has been provided. If at any time during the debugging phase, you want to return to the Editor Assembler without reinitializing it (which would have deleted the entire text buffer), and are under the control of a debugging utility that does not utilize memory from X'5400' (X'2800' under TRSDOS 6.x) to the protected HIGH$, issue a jump command to X'5A03' (X'2D03' under TRSDOS 6.x). Alternately, you can provide a "JP 5D03H" (or JP 2D03H under TRSDOS 6.x) in your program as an exit and return to EDAS. A return to the Editor Assembler will be performed and the text buffer pointers will be maintained. If your program has maintained the integrity of the stack pointer, a RET instruction will return to the EDAS command prompt as the top of the stack contains the prompt address when an exit is made via the "B"ranch command.

EDAS disables the automatic entry to DEBUG on <BREAK> to avoid inadvertently entering DEBUG by depressing <BREAK> to exit an <I>nsert or abort an assembly. In order to enter DEBUG directly from EDAS, perform a <B>ranch command to address X'30'.

Invoking the SAID Editor

SAID is a full-screen text editor that can be used to edit assembler source, C-language source, or other ASCII text files. When used under TRSDOS 6.x or equivalent, SAID provides you with up to seven editing buffers, depending on the availability of 32K memory banks, and the added capability of moving blocks of text from one buffer to another. Before SAID is first used, please install it into your system by running the SAIDINS installation program as noted at the end of this chapter. SAID is easily invoked via:

SAID [filespec] (parm1,parm2,...)

SAID *	**Used to re-enter SAID immediately after exiting so as to reclaim the text buffers.**
filespec	**The name of the file to edit. If filespec is not found, SAID prompts to create it. Command line filespec entry is optional.**
ASM	**Tabs default to 8. File extension defaults to "/ASM". X'1A' stripped from end of file on read and replaced on write.**
CCC	**Tabs default to 4. File extension defaults to "/CCC". X'1A' stripped from end of file on read and replaced on write.**
EXT=string	**Sets the default file extension.**
TAB=nn	**Set default tab width.**

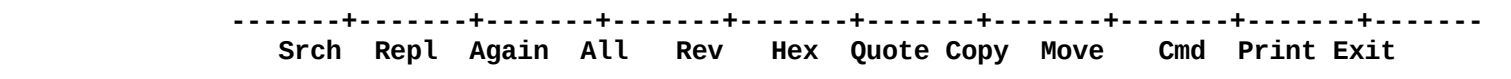
Abbreviations: A=ASM, C=CCC, E=EXT, T=TAB

Note: EXT parameter not usable under TRSDOS 1.3 and 2.3. TRSDOS 1.3 users must enter ASM and CCC parms as parm=0FFFF and TAB entry in hexadecimal, 0xx.

Unless altered by SAIDINS, SAID command functions are invoked with the keyboard depressions as shown in the SAID menu. In the following text, multiple key depressions are shown as connected sequences of keys within angle brackets, i.e. <CLEAR><4> means simultaneously depress the <CLEAR> key and the <4> key.

Editing Status

SAID can display a great deal of status concerning the text contained in the editing buffer. You also can control the optional display of the SAID menu of command keys. This information will look like the following:



```
   ==1== ==2== ==3== ==4== ==5== ==6== ==7== ==8== ==9== ==0== ==:== ==-==
    Ins  Line   Del  Word  Block Load  Save  Macro File  Meta  Pg Dn Pg Up
ile:aaaaaaaaaaaaaa Len:bbbbb/ccccc Ln:ddddd Col:ee = x'ff' ggg%  Banks: hhhhhhh
rch:iiiiiiiiiiiiiiiii Repl:jjjjjjjjjjjjjjjjj Dir:kkk  Cnt:lll
Messages and prompts..."                                  SAID Version 1.1
```

The top line will be a rule of dashes with a plus sign "+" denoting each TAB stop as established by either the default tabbing for the file type [ASM=8, CCC=4] or that set via the TAB parameter.

The next three-line menu is optional. Its display mode is established when SAID is invoked by the MENU setting during the SAIDINS installation. The META command also permits you to toggle the display of this menu. It is recommended that you keep the menu displayed until you get proficient at using SAID's editing commands. The menu displays the command function activated when the key identified by the second line is pressed simultaneously with the <CLEAR> key. The first line designates shifted keys and the third line designates unshifted keys.

The next line contains a great deal of information. The file specification of the file currently being edited in the context buffer is identified by "aaaaaaaaaaaaaa". The current length of the text is shown as "bbbbb" while the total length of the editing buffer is shown as " ccccc". SAID keeps track of a logical line number for the text. For line numbering purposes, a line is considered to be all characters up to and including a carriage return. The number of the line to which the cursor is positioned to is shown as " ddddd". The video column number to which the cursor is positioned is shown as "ee". This value will range from column 00 to column 79 (63 in the case of a 64 column screen). The character which the cursor is positioned over has its value shown in hexadecimal as "ff". This value is useful for determining the text character for undisplayable character values. SAID also keeps a ratio of where the cursor is positioned relative to the end of the text. This is shown as a percentage by the "ggg" value. It is accurate only when the text exceeds 99 characters.

The last field of the status line shows the availability of editing buffers as "hhhhhhh". When SAID is invoked under those systems supporting banked switching, SAID scans for the availability of up to seven editing buffers. SAID will display a dash for each buffer that is available. These are selected by you with the assignments 1, 2, 3, ..., 7. Anytime that you have entered text into one of these buffers, its corresponding dash will be changed to a plus sign.

The next line of status shows the current search string: " iiiiii..."; the current replacement string: "jjjjjj..."; the search direction, "kkk": "For=forward", "Rev=reverse"; and the macro repeat count: " lll".

The last line will be used to display prompting messages or error messages on an as required basis.

Cursor Movement

Cursor movement relates to the re-positioning of the cursor on the screen. "Up" movements invoked while the cursor is on the top line of the screen cause the text to be scrolled downward. Conversely, "down" movements invoked while the cursor is on the bottom text line will cause the text to be scrolled upward.

Left one position	**<LEFT ARROW>**
Right one position	**<RIGHT ARROW>**
Up one line	**<UP ARROW>**
Down one line	**<DOWN ARROW>**
Beginning of next word	**<CLEAR><4>**
Next Screen Page	**<CLEAR><->**
Previous Screen Page	**<CLEAR><:>**
Start of line	**<SHIFT><LEFT ARROW>**
End of line	**<SHIFT><RIGHT ARROW>**
Start of file	**<CLEAR><UP ARROW>**
End of file	**<CLEAR><DOWN ARROW>**
Insert a tab	**<CLEAR><RIGHT ARROW>**

Modes

SAID operates in various modes. In normal operation, entered text over-types any text beneath the cursor. The cursor will appear as an underline unless it is positioned over an underline character at which point the cursor will be displayed as a full block (191D). When toggled into "insert" mode, all text to the right of the cursor will be pushed down one character as each character is entered. The cursor is also changed to a full block. Although a TAB character occupies one position in the text buffer, it is expanded on the screen via spacing to the next tab stop. In line insert mode, a line of spaces is inserted into the text at the cursor position. A new line will automatically be inserted when you attempt to type past the last position of the opened line. Hex insertion mode can be invoked regardless of the state of insert mode. The hex mode allows you to enter all 256 character values by the entry of two hexadecimal digits per character. In quote insert mode, all cursor movement functions are defeated and the character values used for the functions are entered into the text when a cursor movement key is pressed.

Insert/overtype mode	**<CLEAR><1>**
Line insert	**<CLEAR><2> - <Break> to cancel**
Hex insert	**<CLEAR><SHIFT><6> - <Break> to cancel**
Quote insert	**<CLEAR><SHIFT><7> - <Break> to cancel**

Deletions

This section relates to the various operations of deleting text. When you invoke a deletion, it is preserved by SAID and can be restored via the Reverse-DELETE function; however, only the LAST deletion performed is saved. Thus, if you delete a block via "delete-block", the character removed by the DELETE key is lost after the "block" key causes the removal of the block since the "block" delete is the last deletion. Fortunately, you only would have to manually enter one character.

Delete character	**<CLEAR><3>.**
Delete word	**<CLEAR><3> followed by <CLEAR><4> or just <CLEAR><4> if in delete mode.**
Delete line	**<CLEAR><3> followed by <CLEAR><2> or just <CLEAR><2> if in delete mode.**
Delete block	**Mark the block, position cursor inside, then <CLEAR><3> followed by <CLEAR><5>**
Delete to top	**<CLEAR><3> followed by <CLEAR><UP ARROW> or just <CLEAR><UP ARROW> if in delete mode.**
Delete to end	**<CLEAR><3> followed by <CLEAR><DOWN ARROW> or just <CLEAR><DNARW> if in delete mode.**
Delete all	**<CLEAR><3> followed by <CLEAR><SHIFT><4>. Deletes entire context except macro.**
Undelete [oops function]	**<CLEAR><SHIFT><5> followed by <CLEAR><3> which is "reverse" followed by "delete".**

Macro functions

SAID supports a macro key which can be soft programmed (or reprogrammed) by you throughout the operation of SAID. This key can store up to 64 key-strokes. Its use for capturing a series of key entries for repetitive entry will help in speeding up your editing and minimize key entry. Note that in-voking the macro will cause it to repeat according to the repeat rate count set with the Meta function. This repeat count is set to one when you store a series of keystrokes for the macro key.

Invoke current macro	**<CLEAR><8>**
Store a macro	**<CLEAR><6> followed by <CLEAR><8>; The Macro will be saved until the next <CLEAR><8> or until 64 characters are entered.**

I/O functions

This section relates to the loading or saving of text as well as printing out the text buffer or a portion of it. Note that when you are merging two or more files into the text buffer, SAID will load the file at the cursor location - be it the beginning of the text, the middle of the text, or the end of the text. When you invoke "exit", SAID will prompt you to save any buffer which contains text. Note that when you save a file or exit, you will be prompted for the filespec. Respond via <ENTER> to use the current filespec shown in the status line or enter a new filespec which will become the new current one. SAID will abort a printing request if a <BREAK> is detected.

Print a block	**Mark block, then <CLEAR><SHIFT><:>, followed by <CLEAR><5> followed by <0-9>**
Print a file [in memory]	**<CLEAR><SHIFT><:> followed by <CLEAR><9>**
Load file at cursor position	**<CLEAR><6> followed by <CLEAR><9> then the filespec in response to the prompt.**
Save file under current name	**<CLEAR><7> followed by <CLEAR><9>**
Save block	**Mark block, then <CLEAR><7> followed by <CLEAR><5> followed by filespec, then <0-9>.**
Exit	**<CLEAR><SHIFT><->**
Change filespec	**<CLEAR><9> followed by the filespec.**

Block functions

SAID allows you to designate up to ten distinctly labeled blocks. These are numbered from 0-9. You can have more than one block designated with the same number; however, in block copy or block move operations, the first such numbered block found in the text when searched from the beginning of the file will be used for the copy or move operation. Blocks are marked by indicating a BLOCK START and a BLOCK END (the end marker must appear in the text after the start marker).

Block start	**<CLEAR><5> followed by <0-9>**
Block end	**<CLEAR><5> followed by <CLEAR><DOWN ARROW> or <CLEAR><5> followed by <E>.**
Copy block	**Mark block, position to destination, then <CLEAR><SHIFT><8> followed by <0-9>. Note that this command duplicates the contents of the block at the new position. The marked block is retained in its marked position.**
Move block	**Mark block, position to destination, then <CLEAR><SHIFT><9> followed by <0-9>. Note that this command deletes the marked block**

	after inserting the block text into the designated position.
Unmark all blocks	**<CLEAR><SHIFT><5> followed by <CLEAR><5>. Use before saving assembler source.**

Search and replace

This section relates to the facility for finding character strings in the text and optionally replacing them with another string. The replacement string may be null. If any character in the search string is in uppercase then the search will be case sensitive (i.e. "A" and "a" are distinct), otherwise the search will be case insensitive (i.e. "A" and "a" are considered to be the same character). The search string may contain a wildcard character or characters which match all character values (the wildcard character is specified during the installation via SAIDINS). The replacement string may also contain a wildcard character or characters which indicates that the character in that position in the search string will be re-used in the replacement string. Both the search and replacement strings may contain hexadecimal values via entry of a per cent "%" character followed by two hexadecimal digits. "Again" finds the next matching string or replaces the next matching string. "All" invokes the search or replace on all matching strings. Note that the meta command provides an option to force a query before replace which is also installable with SAIDINS.

Search	**<CLEAR><SHIFT><1> followed by the string.**
Reverse search	**<CLEAR><SHIFT><5> followed by <CLEAR><SHIFT><1> followed by the search string.**
Replace	**Invoke a SEARCH, then <CLEAR><SHIFT><2> followed by the replacement string.**
Again	**<CLEAR><SHIFT><3>**
All	**<CLEAR><SHIFT><4>**

Miscellaneous

Invoke a DOS command **<CLEAR><SPACE>**

This allows you to enter any DOS command that is acceptable at DOS Ready with the exception of any command which alters HIGH$.

Invoke a Meta command <CLEAR><0> followed by [C,E,G,H,M,O,R,T,7,UP,DN]

This gives you access to an additional set of infrequently accessed commands. The meta command letter determines the function invoked.

C	**Calculator**
E	**External memory [TRSDOS 6.x version only]**
S	**swap memory bank and full context**
C	**copy a block from an external memory bank**
G	**Go to the start of a line via its line number**
H	**Toggle the help display**
M	**Set macro repeat count**
O	**Set SAID options**
A	**set ASM mode in current buffer**
C	**set CCC mode in current buffer**
T	**set default extension in current buffer**
R	**Replace options: query before replace**
T	**Set tabs position (i.e. every nth column)**
7	**Strip bit 7 off all text in buffer**
<UP ARROW>	**Uppercase next word**
<DOWN ARROW>	**Lowercase next word**

Calculator

SAID contains a built-in reverse polish notation calculator which supports the following three types of numbers:

xxxxB - Binary (i.e. 101101)
xxxxD - Decimal (default, i.e. 45)
xxxxH - Hexadecimal (i.e. 2d)

The following functions are supported:

*****	**Multiplication**
/	**Division**
+	**Addition**
-	**Subtraction (negation is not supported)**
&	**Logical AND**
|	**Logical OR**
^	**Logical XOR**
.	**Used to denote the previous result**

If you wish to output the answer in any base other than decimal then follow the '=' with a 'B' or an 'H' to specify binary or hexadecimal. Entering a period will cause the last result to be substituted. Note the following sample calculation which multiplies 22 base 16 by 1111 base 2, then adds 2 base 10 and outputs the result in decimal:

22h 1111b * 2 +<ENTER>

To output the same result in binary, specify ". =b".

Installing SAID [running SAIDINS]

SAID should be installed in your system by invoking the command,

SAIDINS filespec

where "filespec" should be SAID/CMD - the name of the screen editor - unless you renamed SAID/CMD to some other name. SAID is supplied to support all of the SAID functions mapped to the keyboard, with the exceptions of functions 34, 35, and 36. This mapping can be tailored to your specifications during the installation of SAID while SAIDINS/CMD is running. This installation program must be used first to establish certain DOS interfacing needed before SAID can be used. The following function codes are used during the installation of SAID. They designate the function numbers corresponding to the thirty-six separate command functions in SAID.

1 Cursor left	**19 Meta**
2 Cursor right	**20 Previous Page**
3 Cursor up	**21 Next Page**
4 Cursor down	**22 Find**
5 Beginning of line	**23 Replace**
6 End of line	**24 Again**
7 Top of file	**25 All**
8 End of file	**26 Unmark**
9 Insert a tab	**27 Hex**
10 Insert mode toggle	**28 Quote**
11 Line	**29 Copy block**
12 Delete	**30 Move block**
13 Word	**31 DOS command**
14 Block	**32 Print**
15 Load	**33 Exit**
16 Save	**34 Delete previous character**
17 Macro	**35 Swap buffer with external buffer # 1**
18 File	**36 Swap buffer with external buffer # 2**

The TRSDOS 6.x version of SAID uses the DOS keyboard driver and makes use of the type-ahead supported by the DOS. The Model I/III version of SAID contains a built in keyboard driver which supports type-ahead as well as a complete ASCII keyboard. The installation program can be used to override this built-in keyboard driver. For LDOS users who are using the DOS KI/DVR, you must either override the SAID driver or not use the LDOS KI/DVR (meaning that type-ahead must be off). The Model I/III keyboard driver uses various key combinations to produce the extra characters not available on the TRS-80 keyboard. These are as follows:

<CLEAR> plus		**<CLEAR><SHIFT> plus**	
<,>	**[(left bracket)**	**<,>**	**{ (left brace)**
</>	**\ (reverse slash)**	**</>**	**| (vertical bar)**
<.>	**] (right bracket)**	**<.>**	**} (right brace)**
<;>	**^ (carat)**	**<;>**	**(tilde)**
<ENTER>	**_ (underline)**	**<ENTER>**	**(delete)**

<SHIFT><DOWN ARROW> (control - use with A-Z)

Invoking XREF

The XREF utility is used to generate a cross reference listing of symbols used in your source code of a single assembly stream. Its syntax is:

```
XREF filespec/REF {(LEN=val,PAGE=val,LINES=val,EQU,LIMIT)}

filespec    is the specification of the reference data
            file generated by the -XR switch of EDAS. If
            the file extension is omitted, "REF" is used.

LEN         is the length of your print line (the default
            value is 80).

PAGE        is the maximum number of lines per page (the
            default is 66 for Mod I, 67 for Mod III).

LINES       is the number of lines to print on a page (the
            default is 56 for Mod I, 57 for Mod III).

EQU         is used to generate a file of EQUates instead
            of the cross reference listing.

LIMIT       is used to limit the file of EQUates to those
            symbols containing a special character.

Note: the format of "value" is PARM=ddd or PARM=X'hhhh'.

PAGE is not supported under TRSDOS 6.x

There are no parameter abbreviations.
```

The XREF/CMD utility generates a symbolic cross-reference listing which includes a sorted list of all defined labels, the file of origin of the definition, the line number of the definition, the value of the definition, and the line numbers of all statements referencing the label. XREF will also identify the filename of the file containing the references. XREF will not identify unresolved labels. Therefore, make sure that either all labels are resolved during the assembly that generates the XREF data file, or you do not need the line numbers of those unresolved references appearing in the cross reference listing.

XREF can also be used to generate an assembler source file of EQUates of all symbols used in the program being assembled or a subset of all symbols used. The LIMIT parameter is used to limit the EQUates to only those symbols having at least one special character in the symbol name.

XREF uses, as input, the reference data file which is optionally generated by the -XR switch during the LISTING pass of EDAS (phase 2). XREF cannot function without this data file. You need not enter the file extension, /REF, as it will be assumed if omitted.

The XREF command line parameters enclosed in parentheses are entirely optional. The may be used as follows:

LEN

This parameter controls the printed line length during the XREF listing. If omitted, a value of 80 is assumed to deal with 80-column line printers. If you are using a wide-carriage printer (typically 132 columns), then XREF can use the entire print line by specifying the parameter as:

```
XREF (LEN=132)
```

PAGE

This parameter controls the page size. A value of 66 lines per page (67 on the Model III due to its line counter starting from 1 instead of 0) is used. If your paper is shorter or longer, you can re-specify the page length from the command line. For instance:

```
XREF filespec (PAGE=51,LINES=41)
```

will set the page length to 51 lines per page and initialize to print 41 lines.

LINES

This parameter controls the quantity of lines printed on a page before a form feed is generated. LINES defaults to a value of 56.

EQU

This parameter controls the generation of the EQUate file. If specified, then the cross reference listing is suppressed and a source file of symbols equated to their value is generated. The filespec used to write the EQUate file will be constructed using the filename and drive specification of the "/REF" file. A file extension of "/EQU" will be used. Symbols defined by the "DEFL" pseudo-OP will be maintained as DEFL's in the EQUate file.

LIMIT

This parameter controls what symbols are written to the EQUate file. If entered in addition to the "EQU" parameter, then the EQUate file will be limited to those symbols that contain at least one special character (a character other than A-Z, 0-9).

Cross-Reference Listing

Three informative messages will be displayed prior to generating the listing. "Building symbols declared" will be displayed as XREF creates a table of all symbols declared. The message, "Sorting symbol table" will be displayed as the symbols are sorted. A second pass through the REF data file will be made while the message, "Building symbols referenced" is displayed. This pass is used to create a second table of all references to symbols.

The listing will contain a heading on each page composed of the system DATE and TIME, the TITLE pseudo-op text, and a page number. The heading needs a minimum of 74 columns. Thus, you should not specify a LEN parameter of less than 74. The reference columns will include:

Origin

The filename where the symbol was declared. The ORIGIN will list either the source filename or the filename of the "*GET"/"*SEARCH" directive.

Symbolic Label

This column contains the defined symbol name. If the symbol was defined by a "DEFL" pseudo-OP, a plus sign, "+", will precede the symbol name.

Value

This column contains the value of the symbol as determined during the assembly process. If the symbol shows a DEFL definition, the value will be the first defined value.

Line#

This column lists the line number of the statement defining the symbol.

Usage

This column contains the filename of the file containing a reference to the label. It will be the filename of the *GET/*SEARCH filespec.

Line# of References

This field will contain the line number of all source statements which reference the symbol. All of the references listed on a print line will be contained in the file identified under the usage column. Whenever the Usage file changes, it will cause a new line to be generated in the listing.

Statistics

The quantity of symbols defined is listed along with the quantity of references associated with those definitions.

TTD

The MISOSYS TTD utility is used for transferring to disk, a source cassette file that was created with the Radio Shack EDTASM, Microsoft EDTASM+, or other compatible editor assembler. TTD is not supported under TRSDOS 6.x.

To execute the TTD utility, at your DOS ready, simply use the syntax:

```
TTD {:d}
```

TTD is used to transfer a source cassette file to disk. The filespec will be constructed using the filename found on the cassette tape file and the file extension "/ASM". If the optional drive specification, ":d" (where "d" is the drive number of the drive receiving the disk file), is entered with the TTD command line, it will be used in the construction of the file specification.

TTD will prompt you to ready the cassette via the message:

Ready cassette and <ENTER> -> for a Model I
Ready cassette and enter <H,L> -> for a Model III

The <H,L> entry for Model III users will select either High speed cassette operation (1500 baud) or Low speed cassette operation (500 baud). Respond to the prompt by depressing the <ENTER> key if you are a Model I user, or the correct baud rate character if you are a Model III user.

The cassette source file will be transferred to disk. TTD will then return to DOS.