MISOSYS Relocating Macro Assembler Development System

Table of Contents

The MISOSYS Relocating Macro Assembler product is published by: MISOSYS, Inc., P. O. Box 239, Sterling, Virginia 22170-0239 [703-450-4181]

MISOSYS Relocating Macro Assembler Development System

Important Note

Certain documentation pertaining to this package may be available after the user manual has gone to press. Consult the file entitled README/TXT for details on additional support material and errata.

General Information

The MISOSYS Relocating Macro Assembler (MRAS) is a disk assembler which generates a relocatable (REL) module from one or more source files. The REL module generated by MRAS is a bit-stream compatible with Microsoft (TM) M-80 generated files. Multiple REL modules are then linked via the MISOSYS linker (MLINK) to produce an executable object code file (CMD). The assembler is also capable of directly generating a CMD file when the source file(s) contain no references to relocatable segments. Source files may be created and edited with the full screen text editor (SAID) provided. Libraries of relocatable modules are organized with the librarian (MLIB).

MRAS was designed to provide the maximum in assembly power. As such, it is an advanced tool which is not recommended for the novice Z-80 assembly language programmer. This user manual is not a "learning" manual - it details the use of MRAS and its companion utilities - and 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.

The MISOSYS Relocatable Macro Assembler Development System includes:

FIXUP/CMD - a utility to convert from/to line-numbered source files
MLIB/CMD - a relocatable module librarian
MLINK/CMD - a relocatable module linker
MRAS/CMD - a macro assembler generating relocatable modules
OVERLAY/REL - a module which supports overlay handling
README/TXT - a LISTable text file containing errata
SAID/CMD - a full-screen text editor for source code preparation
SAIDINS/CMD - SAID installation program
XREF/CMD - a symbol cross-reference listing generator

All source text to MRAS 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 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, load the file into SAID using the ASM parameter and resave it (after ensuring that the last character in the file is a carriage return).

Source files commonly used with other assemblers take one of three forms; a pure ASCII text file, a line-numbered text file, or a line numbered file which includes a header. MRAS will automatically accept any of the three types for its input provided all files included in one source stream use the same convention. On the other hand, headered and numbered source files would be found unworkable with the SAID text editor. Thus, a utility called FIXUP has been provided. FIXUP allows you to change from one form to any of the

other forms. FIXUP requires a properly terminated file. Its syntax is:

FIXUP filespec {(*/strip/header/number}

where the parameter "strip" is used to eliminate headers and line numbers, "number is used to add line numbers, and header is used to add both a header and line numbers. The '*' is used to rewrite the file left in the FIXUP buffer. FIXUP defaults to "strip"; reads its input from and writes its output to the file identified as "filespec".

Distribution Disks

The TRSDOS 6.x MRAS Development System is distributed on a 40 track double density data diskette.

The Model I/III MRAS Development system works on both the Model I and Model III under LDOS 5.x, DOSPLUS 3.5, TRSDOS 2.3, and TRSDOS 1.3. It is released on a 35 track single density data diskette. TRSDOS 1.3 users must use the 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 below). The master disk is readable by LDOS and DOSPLUS. Model I or III use under a DOS other than LDOS may require patches to one or more of the supplied programs.

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 MRAS files from the diskette, you will need to change one byte of the TRSDOS 2.3 disk driver using either of the following two methods. This change should 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 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 either method noted above, COPY the MRAS files from the master diskette to your TRSDOS system diskette.

The MISOSYS Relocatable Macro Assembler Development System

Invoking MRAS

MRAS is a macro assembler used to assemble a source disk file(s) into a relocatable object code module. MRAS provides a command line rich in features. The syntax:

```
MRAS 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.
+L=:d              Use -LP for printer (or +L=*PR if DOS
                   supported). Will inhibit -NL and -LP.

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

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

+S=symbol/SYM    - send symbol table to spec in lieu of *DO or
+S=:d              *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".

Switches:          -CI -FE -GC -LP -MF -NC -NE -NH
                   -NL -NM -NO -SL -WE -WS -XR     (see text)

Parms:
Pn               - Set internal symbols            (see text)
Lines=n          - set printed lines per page to n ( abbrev=L).

Note: Default file extensions are shown capitalized in the
      file option filespecs.
```

File options

File options are denoted with a plus sign prefix and are used to re-direct one or more output streams of the assembler. They can accept a syntax of "+s=filename:d" or "+s=:d". The "s" refers to any of the file switches: O, I, L, S, X. The latter will re-use the source filename for the file being switched and the extension appropriate for the switch. File switches must precede the assembler switches.

Assembler switches

These switches control various aspects of the assembler. They are always prefixed with a minus sign.

Switch -CI

The "-CI" switch is used to generate a "core-image" object code file. Executable command files in the 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. When the "-CI" switch is specified, a number of changes take place in MRAS. First, the object code file default extension is changed to "/CIM". 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 MRAS reserves only storage locations when assembling the DS/DEFS pseudo-OPs, DS instructions will automatically be converted to their corresponding "DC" statements with a zero value for operand2. The "-GC" switch will also be turned on.

Switch -FE

The normal operation of MRAS will suppress the output of linkage data for symbols declared EXTRN but never referenced within the source code stream. This switch forces chain external linkage data to be generated for external symbols declared via EXTRN where no reference is made in the module. This can be used to force a loading of the extrn'd module from a library even though no other reference is made in the module with the EXTRN. Its use is generally associated with the inclusion of desired library modules into the ROOT segment of an overlayed program. If -FE is not specified, any symbol listed in the argument of the EXTRN statement which has no other reference in the module will not generate a "chain external" and will not be searched for in a library search.

Switch -GC

This switch tells the assembler to directly output a CMD file. The normal object file output is a relocatable module (/REL). Do not specify this switch if your source contains any CSEGs, DSEGs, or COMMONs. The -GC switch is automatically turned on by switch -CI. The assembler will default to ASEG if this switch is specified.

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. MRAS 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 prop-

erly 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 LINES parameter to alter the paging parameters to suit the specifications of your printer. Note that MRAS does not count characters per line!

Switch -MF

The assembler normally searches the OP code table prior to the macro table. If you want to redefine the code generation of Z-80 OP code mnemonics, you can specify the -MF switch. It causes the assembler to search the macro table before the OP code table.

Switch -NC

Conditional assembly (see the section on ASSEMBLER PSEUDO-OPS) 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 logically "false". This -NC switch is provided to have no "false" conditionals appear in your listings. If a conditional is suppressed, neither the "IF" statement nor the "ENDIF" statement of the "false" clause will be listed.

Switch -NE

Various 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-OP statements. If you want to inhibit the expansion from the listing only (the code will still be expanded for assembly of object code), then 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). MRAS automatically generates this record when writing an object code CMD file. The DOS loader has no problem with this record. If you would like your object code files to contain this record, then do absolutely nothing. 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 generates 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. If you are interested in listing statements containing errors, then you must not suppress the second phase. Note that the lines containing only 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.

Switch -NM

The macro model code is repeated whenever you invoke the 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

MRAS will generate an object code output file unless you tell it to suppress this generation via the -NO switch.

Switch -SL

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

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 continue without further interruption - even though additional errors may be detected. The listing may also be paused at any time by depressing the <SHIFT-@> key, momentarily.

Switch -WS

A complete symbol table cross-reference listing of a single assembly stream is available via the -XR switch and subsequent processing by the XREF/CMD program. An abbreviated printout that contains only a sorted listing of symbols and their value is available at assembly time by invoking the -WS switch. 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. The symbol table can also be invoked via the "+ S=filespec" file option.

Switch -XR

This is the switch option to use if you want to generate a complete symbolic cross reference listing of the assembly stream. Switch -XR will invoke the generation of a reference data file used by the XREF/CMD utility (see the chapter on CROSS REFERENCE UTILITY). The reference data file is generated during the listing pass (phase two). If the XREF filespec is entered via "+X=filespec", this switch is assumed to have been entered. If the XREF filespec is not entered via "+X=", the filespec of the reference file will be generated from the source filename.

Parameters: Pn=val

This parameter provides the power of entering symbol table equates directly from the MRAS 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 entered in MRAS as absolute DEFL values added to the symbol table. By passing parameter values with these on the MRAS 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 dependent on that supported by your DOS and may include:

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

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. 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 MRAS command line including **(P1)** will set "@@1" to TRUE, "@@3" was defaulted to FALSE, and thus "MOD1" would be TRUE and "MOD3" would be FALSE since the two conditional symbols you are using are equated to the "@@n" parameters.

Assembler listing

During the first pass, the name of each file included or searched will be displayed as an informative message. During the listing pass, MRAS keeps track of each statement's logical line number within its source file and the logical line number of the assembly output stream. Stream 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]. Lines containing errors will be prefixed with the name of the file containing the line, the line number within the file, and the error message. The statement itself will display the stream line number.

The "+" indicator denoting a macro expansion will appear after the stream line number. The address will be suffixed with a mode indicator which indicates the current mode of the assembly source. The 16-bit operand will be suffixed with a mode indicator which indicates the mode of the operand. The symbol table will include a mode indicator following the value of each symbol. The indicators are as follows:

```
blank - absolute
   '  - code relative
   "  - data relative
   !  - common relative
   C  - named common
   *  - extern symbol
```

At the conclusion of the listing pass, the free space remaining in the buffer pool will be displayed as, "ddddd Free space". This can be used as an indicator of how dangerously huge your program is getting.

Error totals

At the conclusion of pass three, the total number of errors will be listed. An "Unclosed conditional" error is also included in the ERROR TOTALS count. This error total will be displayed after the conclusion of phase two if object code is not generated. If you place a "*LIST OFF" pseudo-OP at the beginning of your code, lines containing errors will be listed.

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.

Note: Fields are separated by a tab or spaces.
```

As can be noted from the format box, 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. That means that 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 also retained as tabs and thus will keep source files smaller than using multiple spaces. A statement line must not exceed 128 characters in length; thus, if a carriage return is not detected by the 129th character, a "Load file format error" will be generated.

Symbolic names

A label is a symbolic name of a line of code. Labels are always optional. A label is a string of characters of any length; however, only the first 15 characters will be significant. A symbol is defined as:

```
name{{:}:}                Defines "name"
```

A terminating single colon is optional. A double colon defines "name" as PUBLIC. If "name" is used as a reference suffixed with "##", then "name" is declared extern. Labels designated as PUBLIC or EXTRN which exceed seven

characters in length will be automatically truncated to seven during the generation of the /REL file without warning. If two or more labels with identical first seven characters are so truncated, the linker will flag a multiple definition error. The first character must be a letter (A-Z) or one of the special characters: the underline, "_"; the dollar sign, "$"; or the at sign, "@". It is recommended that you reserve use of "$" as the first character of "local" labels since they can be suppressed from a symbol table output via the "-SL" assemble switch

A label may contain, within character positions 2-15, letters (A-Z), decimal digits (0-9), or certain special characters: the at sign, "@"; the underline, "_"; the question mark, "?"; or the dollar sign, "$". The dollar sign "$", appearing by itself, is reserved for the value of the reference 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. Thus, the following two LABEL examples are completely equivalent:

```
ALLALONE
ALLALONE   EQU $
```

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, ON
C, NC, Z, NZ, M, P, PE, PO, OFF

Examples of labels:

```
ENTRY       @OPEN       BUFFER$      BYTE_POINTER     WHAT?
SELECT_CODE $$CORE      @            CARRIAGE_RETURN  @EXIT
```

A special symbol is "$MEMRY". If this symbol name is declared PUBLIC, the linker will store the address of the first available memory location which follows your program into the word defined as "$MEMRY". Thus, if you choose to use this capability, $MEMRY must be defined via a DW statement. or equivalent.

Opcodes

The OPCODES for the MRAS assembler 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 unless *RADIX is used to change the default radix.

A constant not followed by one of the radix characters is assumed to be decimal. This assumption can be changed via the *RADIX assembler directive. 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. These 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.

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. MRAS 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 a 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 a 16-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 equal to or larger then 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 larger than 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 smaller than or equal to 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 smaller than 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 section on MACRO PROCESSING.

Evaluation of expressions - limitations of mode/class

Symbols have both a mode and a class. The modes are absolute, code relative, data relative, and common relative (which is common specific, i.e. coupled to common relative is the specific common which the symbol is a part of). The class is either extern or not extern. The following rules apply to all expressions:

A. Addition:
   1. One term must be absolute.
   2. The resulting mode is: absolute + <mode> = <mode>
   3. Either term may be extern but not both.
   4. If one term is of class extern, the other must be absolute.

B. Subtraction:
   1. <mode> - absolute = <mode>
   2. <mode> - <mode> = absolute; both modes must be the same.
   3. <extern> - absolute = extern; the result is extern
   4. The second term cannot be of class extern.

C. All other binary operators require absolute terms. All unary operators except unary minus require an absolute term. Unary minus follows the rules of subtraction with the left term assumed to be absolute.

Additionally, all expressions which resolve to a byte value (in contrast to 16-bit word value) must resolve to absolute mode, class not-extern.

Pseudo-OPs

There are many pseudo-OPs which MRAS will recognize. These assembler operations, although written much like processor instructions, interface to the assembler instead of the Z-80 processor. They direct the assembler to perform specific tasks during the assembly process but have no meaning to the Z-80 processor. Some of these pseudo-OPs generate data values used by your program and are called "data declaration" pseudo-OPs. Others control paging operations and may be best termed, "listing" pseudo-OPs. A broad range of operations to invoke the assembly of code clauses based on conditional evaluations are supported through many "conditional" pseudo-OPs. These 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

ASEG Sets the program counter to the absolute segment

COMMON Sets the program counter to a common relative segment.

CSEG Sets the program counter to the code relative segment. This is the default mode of the assembler.

DEFL Establishes a value for a label which can be altered during the assembly.

DSEG Sets the program counter to the data relative segment.

END Signifies the end of a *GET, *INCLUDE, or *SEARCH member. Supplies the execution transfer address.

ENTRY Synonomous with GLOBAL.

EQU Estalishes a constant value for a label.

EXT Synonomous with EXTRN.

EXTRN Specifies the symbols in the name list as external.

GLOBAL Specifies the symbols in the name list as entries.

LORG Establishes a load origin for executable object code files. LORG is unusable for /REL output.

NAME Specifies the module's name for the /REL file. This defaults to the object code filename.

ORG Establishes an origin for executable object code files or relative code segments.

PUBLIC Synonomous with GLOBAL.

Note: An ORG can follow ASEG, CSEG, DSEG, or COMMON //; but not a named common. A maximum of seven named commons are permitted in one module. The "name" of a common cannot be the same as any symbol.

Conditionals

ELSE	Alternate clause to be assembled if the prior clause has evaluated FALSE.
ENDIF	Signifies the end of a conditional block.
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.
IFABS	Logically TRUE if "name" is absolute.
IFDEF	Logically TRUE if "name" has been defined prior to this statement or if "name" is extern, else FALSE.
IFEQ	Logically TRUE if expression1 = expression2.
IFEQ$	Logically TRUE if string1 = string2.
IFEXT	Logically TRUE if "name" is extern.
IFLT	Logically TRUE if expression1 < expression2.
IFLT$	Logically TRUE if string1 < string2.
IFGT	Logically TRUE if expression1 > expression2.
IFGT$	Logically TRUE if string1 > string2.
IFNDEF	Logically TRUE if "name" has not been defined prior to the statement or if "name" is not extern, else FALSE.
IFNEXT	Logically TRUE if "name" is not extern.
IFNE	Logically TRUE if expression1 <> expression2.
IFNE$	Logically TRUE if string1 <> string2.
IFREF	Logically TRUE if "label" has been referenced but not defined prior to the statement, else FALSE.
IFREL	Logically TRUE if "name" is relative.

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

Miscellaneous

COM	Generates a CMD 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 permissable assembler switches from within the source code during the first pass of the assembler.
PAGE	Outputs 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 section 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 MRAS provides the easier specification.

The following are valid declaration statements:

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

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. "DS" adds the quantity of storage locations reserved to the current program counter (PC) to calculate a new PC value. When generating a CMD object code file, this action will cause the next assembled byte to create a new load record. When generating a REL object code file, this action will generate a Set Location Counter special link item.

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, MRAS automatically converts all "DS" declarations into equivalent "DC" declarations using a value equal to zero. The previous 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 symbolic label.
```

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 NAME     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$     NAME    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 ASEG

This pseudo-OP is used to set the program counter to the absolute segment. The syntax of "ASEG" is:

```
ASEG
ORG  expression                (optional)

expression  When evaluated, "expression" will be the
            origin of the segment. Expression must
            evaluate to an absolute value.
```

It is not necessary for an ORG to follow an ASEG. An ASEG will set the absolute program counter to the last encountered ASEG value, or to zero if no previous ASEG had been specified.

Pseudo-OP COMMON

This pseudo-OP is used to set the program counter to a common relative segment. The syntax of "COMMON" is:

```
COMMON  /{name}/

name        An optional name which specifies a name
            for the common segment.
```

This pseudo-OP sets the PC to a common relative segment: the slashes are required. If "name" is omitted, blank common is assumed. A maximum of seven named commons are permitted in any one module. The "name" of a common cannot be the same as any symbol.

It would be unusual for an ORG to follow a COMMON. An ORG cannot follow a named common. A COMMON will set the specified common relative program counter to the beginning of the common segment (i.e. to zero relative).

Pseudo-OP CSEG

This pseudo-OP is used to set the program counter to the code relative segment. The syntax of "CSEG" is:

```
CSEG
ORG  expression                (optional)

expression  When evaluated, "expression" will be the
            origin of the segment. Expression must
            evaluate to an absolute value.
```

It is not necessary for an ORG to follow a CSEG. A CSEG will set the code relative program counter to the last encountered CSEG value, or to zero if no previous CSEG had been specified.

The assembler defaults to CSEG if no other segment pseudo-OP is specified; however, if MRAS is invoked with the -GC switch, it will default to ASEG.

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.

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 DSEG

This pseudo-OP is used to set the program counter to the data relative segment. The syntax of "DSEG" is:

```
        DSEG
        ORG  expression                (optional)

  expression  When evaluated, "expression" will be the
              origin of the segment. Expression must
              evaluate to an absolute value.
```

It is not necessary for an ORG to follow a DSEG. A DSEG will set the code relative program counter to the last encountered DSEG value, or to zero if no previous DSEG had been specified.

Pseudo-OP END

The "END" pseudo is used to denote the exit of a *GET, *INCLUDE, or *SEARCH process. If the END statement of the source file has a non-zero operand, it will denote the transfer address of the module. Its syntax is:

```
        END {nn}
        END {label}

  nn          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, processing stops when the end of the file is reached. The END statement can specify a transfer address (i.e. END LABEL or END 6000H). Only one transfer address should be specified per assembly stream; however, if more than one non-absolute-zero END operand is detected, only the first one will be retained. 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).

Pseudo-OPs ENTRY, GLOBAL, and PUBLIC

Any one of these may be used to specify that the names of the symbols in the name list are symbols global to linkage when REL modules are linked by the linker (MLINK). The syntax is:

```
ENTRY   name{,name}...
GLOBAL  name{,name}...
PUBLIC  name{,name}...

name        A symbol to be defined global.
```

MRAS treats GLOBAL, ENTRY, and PUBLIC totally equivalent in order to provide compatibility with other relocating assemblers. A symbol classified as such is known to other separately assembled modules which specify "name" as EXTRN. All symbols not specified as GLOBAL, PUBLIC, or ENTRY are known only to the module currently being assembled.

A symbol can also be implicitly declared PUBLIC by appending two colons to the "name" where the symbol is defined. Thus, the following two methods both declare the symbol, TRUST, as PUBLIC:

```
--------method 1-------              --------method 2-------
        PUBLIC TRUST
TRUST   LD     HL,VALUE              TRUST:: LD     HL,VALUE
```

Symbols declared PUBLIC in one module that need to be referenced by another module must be declared EXTRN in all modules other than the module where the symbol is defined.

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 EXTRN

The EXTRN pseudo-OP is used to declare a PUBLIC symbol which is defined in some other module. EXT is synonomous with EXTRN. The syntax of "EXTRN" is:

```
        EXTRN  name{,name}...
        EXT    name{,name}...

  name        A symbol defined external to the current
              module.
```

When your program is made up of more than one REL module linked together by the linker, symbols which are referenced in a module but defined in another must be declared EXTRN in all modules which reference the symbol other than the module which defines it.

Pseudo-OP LORG

The "LORG" pseudo-OP is used to establish a CMD 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           Turn on LORG
        LORG  expression   Turn on LORG
        LORG  $            Turn off LORG

  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. Such is the case when using MRAS to generate a PROMable module to be used on an external processor

origined at zero. For example:

```
ASEG
ORG   0000H
LORG  7000H
```

will assemble code so that absolute address references and the execution addresses are referenced from X'0000'; 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.

LORG is usable only when generating CMD files directly from the assembler via the -GC switch. LORG cannot be used when generating REL output.

Pseudo-OP NAME

This is used to specify the module name of the generated REL file. The syntax of "NAME" is:

```
NAME modname
NAME ('modname')              (equivalent)
```

modname Specifies the module name for the REL file.

If NAME is not specified in the source stream of an assembly which generates a REL object code file, a special link item module name record will be generated using as a default, the first seven non-blank characters of the REL file's name. The second format is supported for compatability with M-80.

Pseudo-OP ORG

The "ORG" pseudo-OP is used to establish an address for the program counter so that the address references within a program are designated to origin from other than address 0000H. 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.

An ORG can follow an ASEG, CSEG, DSEG, or COMMON //; but not a named common. When ORG follows a relative segment specification, the program counter will be set relative to the beginning of the segment, an amount equal to the operand of the ORG. The operand of the ORG must evaluate to an absolute value.

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 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 that 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 as follows:

```
IFxx  operand_of_IF
.
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 argument 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 the evaluation is 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 "0" for a logical FALSE so that, "FALSE EQU .NOT.TRUE" is a valid statment. The values can be set in program as follows:

```
TRUE     EQU     -1
FALSE    EQU      0
MOD1     EQU      TRUE
MOD2     EQU      FALSE
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 may be used. It is coded as:

```
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 alternative clauses that can be based on one switch.

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", "GT", or "NE" meaning less than, equal to, greater than, or not equal to respectively when comparing "exp1" to "exp2".
[$]	The "$" is specified in macro comparisons with the expressions treated as strings (see the section on USING MACROS).
yy	Can be "DEF", "NDEF", or "REF" representing whether <name> has been defined, undefined, or referenced but undefined; or ABS, REL, EXT, or NEXT representing a test of the mode or class of the symbol.

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 clause 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 or if name has been declared EXTRN. "IFNDEF name" will evaluate TRUE if "name" has NOT been defined prior to the evaluation of the IFNDEF on each assembler pass nor has it been declared EXTRN. "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 "IFEXT name" pseudo-OP will evaluate TRUE if "name" has been declared EXTRN. "IFNEXT name" will evaluate TRUE if "name" is not declared extern. "IFABS name" will evaluate TRUE if "name" is defined in an absolute segment whereas "IFREL name" will evaluate TRUE if "name" is defined in one of the relative segment types (code, data, common).

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 clause, 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
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 segment 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 of a directly generated CMD file. Its syntax is:

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

An object deck comment block 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 a part of the comment record.

The COM pseudo-OP will generate a comment block 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) 1985 by MISOSYS, Inc.>'
```

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. Its syntax is:

OPTION {-/+}switch{,-/+switch},...
-/+ An optional prefix to turn the switch OFF or ON
switch Any of the permissable 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 symbol(s) 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, functions, 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 a 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 sub-title to a heading is permitted with the "SUBTTL" pseudo-OP. The subtitle string length can be from zero (0) to 80 characters in length. A zero length indicates that sub-titling is disengaged. The SUBTTL string does not need to be enclosed in angle brackets; they are optional. SUBTTL also automatically invokes a PAGE.

Lower case strings can be maintained by the use of single quotes which surround the angle brackets. You may change the subtitle by using additional SUBTTL pseudo-OPs throughout the text. Subtitles will appear on the first page following the SUBTTL pseudo-op. If the SUBTTL text string is null (of zero length), 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. Where many *GETs are being used, you may want to establish a sub-title for each to provide a visual indication on the listing.

The TITLE pseudo-OP automatically invokes a page heading and adds the title to the headings of assembler listings. The title string is limited to 28 characters and only one TITLE is accepted. The 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 MRAS 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]. For this reason, if you use a title, it is advisable to set DATE and TIME prior to executing the assembler. 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 the angle brackets with single quotes as in:

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

The first TITLE pseudo-OP found in the text will be used for titling. All other TITLE pseudo-ops will be ignored.

Assembler Directives

MRAS supports seven assembler directives. 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 do not generate any machine language code by themselves, they merely act as "commands" to the assembler. Each "command" must start in column one of a source statement line, and must start with either an asterisk "*" or a period ".". The entire directive word may be entered, or it may be abbreviated to its minimum unique character string. The assembler directives are:

*Get file	Causes the assembler to begin reading source code from the "file".
*Inc file	Causes the assembler to begin reading source code from the file identified on the command line via "+I=filespec". Treated as *GET if no "+I=filespec" was specified.
*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.
*RAdix exp	Changes the default radix to expression which must evaluate to the range <1-16>.
*REquest	Generates a Special Link Item to request a search by the linker of the library file identified in the *REQUEST directive.
*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/ASM
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 MRAS will automatically detect its type and adjust accordingly; however, all source files must be similarly structured. When the end-of-file is reached, or an assembly language "END" statement is read, assembly resumes from the next statement following the statement which invoked the "*GET".

"*GETs" can be nested to four (4) levels. That is, a statement can GET a file which GETs a file which GETs a file 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 a series of source files as one assembly stream.

***INCLUDE filespec**

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

*Include filespec/ASM
filespec Causes the assembler to begin reading source code from the file identified on the command line via "+I=include"

This directive tells the assembler to temporarily switch its source assembly to the file identified on the MRAS command line via the "+I=include" file switch. If no "+I=" file switch was entered, the *Include is treated exactly as if it were a "*Get filespec."

LIST ON/OFF

This 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. All statements which follow a "*LIST OFF" will be suppressed during the listing pass. The "*LIST ON" will resume 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.

***MOD**

This directive increments a character substitution string to simulate local labels in blocks within one module. 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 statement. 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 occuring. 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 MOD string value cycles from A-Z, then from AA-AZ, BA-BZ, ..., ZA-ZZ, then from AAA-AAZ, BAA-BAZ, ..., ZZZ.

This will allow for a simulation of "local" labels. Remember, the "?" substitutions will only be made if *MOD was specified.

***RADIX expression**

This directive sets the default radix for all numeric terms except for "*RADIX expressions" which always default to 10. Its syntax is:

*RAdix expression
expression Is evaluated and becomes the new default radix for all numeric terms. The value of expression must be in the range <1-16>.

Note that in the evaluation of the expression for the *RADIX directive, the assembler will always use a radix default of 10. The assembler defaults to a radix of 10 unless overridden by a *RADIX directive.

***REQUEST lib1{,lib2},...**

This is used to convey information to the linker. It will generate a "Request Library Search" special link item for use by MLINK. The syntax is:

*REquest lib1{,lib2},...
libn The 1-7 character name of the REL library to be searched by the linker.

*REQUEST will generate the link item to the linker for each library name identifed in the argument list.

***SEARCH filespec**

This directive is used to invoke an automatic search of a Partitioned Data Set (PaDS) source library, "filename/LIB", for all members that will resolve undefined references in the source stream. This provides a source library structure. *SEARCH will require two (2) levels of "*GET" nesting. Also, a *SEARCH member cannot use a *GET directive or another *SEARCH directive. The default file extension for searched files is "LIB". The syntax of *SEARCH filespec is:

*Search filespec/LIB
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.

The PaDS source library constitutes members composed of one or more routines. Each routine 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.

MRAS 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 member will be accessed and read just as if it were the target of a *GET. MRAS 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 defined symbol in the member that has the same name as the member name, MRAS will abort the assembly and advise of a library error by displaying the message:

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

After the member's source code is read, MRAS will continue to search the PaDS library until it exhausts all 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.

Where more than one routine is in a member, each should be surrounded by IFREF/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.

```
; 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 PaDS library, a *SEARCH of the library will access the member and assemble only the SHIFT routine.

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 MRAS. 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 MRAS, 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, MRAS 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 MRAS. 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.

MRAS 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 uinique 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. MRAS 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

MRAS 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
```

General

MRAS recognizes two types of errors. These are:

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

Disk I/O errors can be received during an assembly. When an I/O error occurs, the assembly will be aborted and control will be returned to DOS.

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.

Following is a list of all error messages and an explanation of each.

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. If an I/O error is detected during an assembly, the long form of the error message will be displayed. This provides an observance as to which file was affected by the I/O error.

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

If you attempt to assemble a file that is not a valid source code file, the message, "Bad parameter(s)" may be displayed.

Terminal Errors:

Symbol table overflow

There is not enough memory for the assembler to generate your program's symbol table or macro storage. You have one option: divide your program into two or more relocatable modules and assemble each separately then link with the linker.

***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 SYMBOLIC NAMES.

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.

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.

Invoking MLINK

MLINK is used to link one or more relocatable modules to generate an executable program file. MLINK is easily invoked interactively or from a Job Control Language (JCL) by the syntax:

MLINK {filespec}{,filespec}... {switch}{switch}...

filespec	- One or more relocatable modules to load.
switch	- any valid MLINK switch as follows:
-A={y/N}	- Specify that MLINK is to abort or not abort upon an error in the command processing. Default=N.
-C=address	- Specify the hexadecimal origin of the common blocks for succeeding modules loaded.
-D=address	- Specify the hexadecimal origin of the data segment for succeeding modules loaded.
-E	- Exit to DOS; Save CMD file if modules loaded.
-E=symbol	- Same as -E but use symbol as entry point.
-H={y/N}	- Generate 05 record header from 1st module's name. Default=N.
-I={y/n}	- Generate a core-image '/CIM' file. Default=N.
-L=address	- Specify the hexadecimal link origin of all blocks not switched by -P, -D, or -C
-M	- List all defined symbols (3 per line).
-M=filespec	- List defined symbols to filespec (1 per line).
-N=filespec	- Specify /CMD file generation.
-N=:d	- Specify /CMD file generation. Use default filename for filespec.
-O	- Handling for overlays - See overlay section.
-P=address	- Specify the hexadecimal origin of the program segment for succeeding modules loaded.
-Q={wxyz}	- Specify generation order of segment classes in the /CMD file generated [A,P,D,C].
-Q	- Reset segment classes to link origin.
-R	- Reset and clear all tables.

-S=filespec	- Search referenced library to resolve any undefined globals.
-U	- List undefined symbols.
-V=filespec	- Use virtual memory file for stream buffering.
-V=:d	- ditto; filename generated is MLINK/VMF:d.
-X	- Equivalent to -E
-Y=<text>	- Used to specify the text for a 1F load module record (a comment record).
-Z={y/N}	- Generate hex zeroes for DS regions in DSEGs and COMMONs. This switch defaults to N.

Command input

Filespecs and switches can be intermixed on a single line by separating filespecs with either a switch prefix, a comma, or a space. A plus sign '+' or a minus sign '-' can be used as the switch prefix. Any character string not starting with a switch prefix will be interpreted as a file which contains a relocatable module or modules to load into the linker. The /REL extension will be automatically assumed. If a file contains more than one module, each module will be processed and loaded. If any symbol is detected as being defined at more than one address during the loading of a module, you will be notified of the symbol name and processing will continue. This may occur when one or more symbols of more than seven characters have been truncated to a length of seven and are thus no longer unique.

At the conclusion of processing a complete command input line, you will be prompted via '?' to enter another command line. A <BREAK> is an immediate exit to DOS. Command line entry uses the KEYIN handler and is thus usable from Job Control Language.

If any error is detected in an input, it will result in an appropriate error message and any remaining input fragment(s) from that entry will be ignored. File I/O errors may result in MLINK aborting.

Status messages

At the initiation of the '?' prompt for more input, a status line will list the number of free bytes remaining in the buffer area in the format:

ddddd free space (ddddd in decimal)

The segment origin address, end address, and length (all in hexadecimal) will also be listed for any segment chains loaded. Segment chains are either Pro-

gram (P), Data (D), or Common (C) for segments loaded after a -P, -D, or -C switch respectively. The "Program" segment results from a CSEG or code segment. "P" is used in the linker so as to be able to differentiate from the single letter "C" used to designate the COMMON segment. All other segments loaded without a respective switch will be stored classified as being in the Link Origin chain and will be designated by the letter, "L". The format of the status will be:

a <ssss-eeee llll>

where "a" = L, P, D, or C; "ssss" = the chain's lowest origin; "eeee" = the chain's highest load address; "llll" = the chain's length calculated from "llll=eeee-ssss+1".

The '-E' switch will also display this status prior to generation of any specified output file.

MLINK switches:

MLINK switches control various aspects of the linker. Some of the switches include a parameter which may be optional. A switch parameter is connected to its switch by either the '=' or ':' separator.

Switch -A={y,N}

Upon detecting an error, MLINK will normally recycle to the '?' command prompt to await a new command. If you wish an immediate exit on a detected error, then this switch can be used to tell MLINK to abort or not abort upon an error in the command processing. Its use is recommended for JCL invocation of the linker. The switch defaults to NO.

Switch -C=address

MLINK provides the capability of separating the code, data, and common segments to origins of your choosing. Any not switched are origined according to the link origin (see the -L switch). Use this -C switch to specify the hexadecimal origin of the common blocks for succeeding modules loaded. Note that the switch does not take effect on any module already loaded.

Switch -D=address

MLINK provides the capability of separating the code, data, and common segments to origins of your choosing. Any not switched are origined according to the link origin (see the -L switch). Use this -D switch to specify the hexadecimal origin of the data segment for succeeding modules loaded. Note that the switch does not take effect on any module already loaded.

Switch -E{=symbol}

This switch is used to generate the executable program file (/CMD) if any modules had been loaded then exit to DOS. If any symbol is left undefined (or some other error is detected), MLINK will terminate the -E switch processing and recycle to the '?' prompt. You may exit via <BREAK> or "-R-E". If you wish to designate one of the PUBLIC symbols as the entry point to the program, use the '-E=symbol' syntax. This will override any entry point which was designated via an END statement.

Switch -H={y/N}

This switch is used to have MLINK generate a 05 object file record header when the executable program is output. The header name is derived from the first loaded module's name. Switch -H defaults to "N".

Switch -I={y/N}

This switch is used to specify that MLINK should generate a core-image '/CIM' object file in lieu of a load format program file. -I defaults to N.

Switch -L=address

MLINK provides the capability of separating the code, data, and common segments to origins of your choosing. Any not switched are origined according to the link origin. The link origin defaults to 3000H for TRSDOS 6.x and 5200H for Model I/III operation. If you wish to designate another link origin for all blocks not switched by -P, -D, or -C, specify the hexadecimal link origin via this -L switch. Note that the switch does not take effect on any module already loaded.

Switch -M{=filespec}

This switch is used to list all defined symbols at three per line. Any symbol defined at more than one address (multiply defined) will be indicated by the appendage of an asterisk. If you enter a filespec parameter, the list of defined symbols will be written to the designated file at one per line.

Switch -N=filespec {-N=:d}

This switch is used to request the /CMD file generation. It may be entered anytime during the entry of link commands. If you use the syntax of "-N=:d", the generated /CMD file will use the default filename for filespec which will be the same name as the first REL module loaded. This syntax requires that at least one REL module is loaded prior to entry of the - N=:d.

Switch -O

This switch schedules the series of modules loaded since the last -O switch for generation as an overlay file. The first series of modules loaded will be the root program. See the section on overlays for more information.

Switch -P=address

MLINK provides the capability of separating the code, data, and common segments to origins of your choosing. Any not switched are origined according to the link origin. Use this switch to specify the hexadecimal origin of the program segment for succeeding modules loaded. Note that the switch does not take effect on any module already loaded.

Switch -Q={wxyz}

During the generation of the output object file, the byte stream is output by module segments in the order of the modules loaded. If you have switched any of the segment origins via the -P, -D, or -C switches, you may want to redesignate the specific order of the segments being output. Specify the generation order of segment classes in the output file generated via this -Q switch. Denote the segment class via: A, P, D, or C for absolute, program, data, and common respectively. Any segment class not sequenced by your entry will automatically be output in the sequence APDC. Note that a maximum of four entries are permitted and no two may be duplicated.

The action of the -Q switch may be restored to the default mode of module segment order by entering the -Q switch without a parameter [-Q].

Switch -R

This switch is used to reset and clear all tables MLINK had internally developed. Any open VM file will be deleted. Use -R when you wish to reconstruct a set of modules using different origins. If you wish to terminate the link session after you have loaded modules, you can reset the linker with -R then issue the -E switch..

Switch -S=filespec

You can manually request the linker to search a library of modules created with the MLIB librarian so as to resolve undefined globals. Do NOT enter a file extension with the filespec. The library search facility of MLINK is capable of searching either a /REL library or an /IRL library. MLINK will first assume a RELocatable structured (REL) library. If a REL library is not found, then an Indexed ReLocatable (IRL) library will be assumed. If an IRL library is not found, the "File not found" error will be issued. The -S switch processing must assign its own extension to be able to properly select REL searching or IRL indexed searching!

Switch -U

This switch will cause the listing of all symbols currently undefined. Unlike the -M switch, the list of undefined symbols cannot be redirected to a disk file.

Switch -V=filespec {-V=:d}

As relocatable modules are loaded, their code, data, and common segments are stored in memory (the available buffer pool starts immediately after the linker and extends to the memory address noted as HIGH$ by the DOS). The linker also maintains a set of linkage tables which contain information on how to interconnect the modules which make up the output file. A symbol table is also maintained in memory. In order to be able to handle the linkage of programs which are larger than the available memory, MLINK provides a virtual memory file facility which can be used to provide buffering for the segments in each module. The linkage tables and the symbol table are still maintained in memory. Operation of the linker via virtual memory will certainly degrade the speed performance of MLINK; however, it will enable you to link large programs. It is beneficial if the VM file is on a RAMdisk.

If you specify the -V=filespec switch, all modules subsequently loaded will be buffered through the virtual memory file specified. If you enter the switch with only a drivespec as the parameter (as in -V=:d), the filename generated will be MLINK/VMF:d. Note that the VM file is deleted during processing of the -E switch.

If you are using virtual memory buffering and still get a "Symbol table overflow" error, attempt to reclaim any high memory used by filters, drivers, or resident programs so as to increase the size of the buffer are available to the linker. Once you have exhausted that procedure, examine the memory usage requirements of the linker and then attempt to "compact" some of the modules making up your program.

Switch -X

This switch is equivalent in operation to switch -E.

Switch -Y=<text>

This switch is used to specify the text for a 1F load module record. Text can be continued onto as many additional lines as needed. A maximum of 255 characters may be entered. The '>' character terminates the text entry.

Switch -Z={y/N}

The normal operation of the linker is to suppress the generation of object code for data regions which have been skipped over by the DS/DEFS pseudo-OP. If you wish to generate byte zeroes (00H) for all such space in

DSEGs and COMMONs, Specify "-Z=Y". All space skipped over by DS/DEFS in CSEGs will be set to zero in the output object file.

The action of the -Z switch doesn't take effect until the generation of output but will pertain to all modules. To turn off the -Z switch and revert to normal MLINK operation, specify "-Z=N". This switch defaults to -Z=N.

Command file generation:

The default origin of the REL stream will be the link origin (3000H for TRSDOS 6; 5200H for Model I/III operation). This default value may be permanently patched at the link origin of the linker. You can also override the default link origin at MLINK's execution by using the -L switch. All segments will be loaded starting at this origin. You can alter the loading of the program, data, and or common segments via the -P, -D, and -C switches. Once a switch is in effect, all segments of the switched class will be origined relative to the address entered. All unswitched segment classes continue to utilize the link origin. The command file will be generated in the order of the modules loaded unless you specify a different order via the -Q switch. The linker will NOT advise you of any segments which overlap!

An output file is only generated when the -E switch is invoked if the -N switch has been specified. The -E command will first check for any multiply defined symbols and abort the operation if at least one symbol is defined at more than one address. Any "Request library search" requests will be first satisfied by searching the referenced libraries. If any symbols are left undefined, they will be displayed and the request will be aborted. The linker will recycle to request another command. Otherwise, next, the chain external requests will be handled. Finally, the extern+offset requests will be handled. If $MEMRY is defined, it will be loaded with the address of the first free byte. The loaded modules are now ready for object code generation.

If you specified "-H=y", a 05 header record will be generated using the module name of the first file loaded. After the header record, a record type 1F will be generated if you specified the "-Y=<text>" switch.

Note: MLINK handles the module bit stream on a segment record basis which incurs a great deal of overhead in the linker compared to a memory-fixed linker such as L-80. The method was chosen for MLINK so as to minimize the size of the resulting /CMD file. This is important for TRS-80 systems.

Linker memory overhead free space requirements:

Each segment (code, data, common, absolute) requires 9 bytes plus the segment length. Data and COMMON segments require an additional storage equal to the segment length. Code, data, and common segments require no storage for the segment length if the -V virtual memory switch has been specified.

Each chain external requires 8 bytes of storage.

Each extern+offset requires 8 bytes of storage.

Each entry symbol requires 5 bytes of storage plus the length of the symbol.

Each request library search takes 3 bytes plus the name length.

Each -O switch takes 20 bytes.

Overlay processing

MLINK supports the handling of programs which desire to use overlays. The method is straightforward. A program may be partitioned into a root program and one or more overlay programs. No address of an overlay is known to the root program. The entry point of the overlay is its link origin established by the linker. Proper entry is performed when the OVERLAY subroutine is called by your root program. No address of an overlay is known to any other overlay. All addresses of the root program are known to all overlays. The root program and all overlays are generated in a single link session.

When the -O switch is entered, MLINK will perform a fixup of the current chain of modules loaded. The fixup process will:

1) Search all libraries specified via a REQUEST assembler instruction,
2) Fix all chain externals,
3) Fix all extern+offset,
4) Prepare MLINK for a new program chain. This resets the -Y switch for the next chain (the -H switch is global). This also establishes the transfer address (entry point) for all overlays to be the next available LINK address after the root. That LINK address will be used for the overlay's entry and not a -P switched origin. It is therefore the proper procedure to not use the -P, -D, and -C switches if you want to use the overlay handling scheme of MLINK.

All modules loaded after an -O switch will be partitioned into a new chain. PUBLIC symbols in one overlay chain of modules do NOT interfere with PUBLIC symbols in other overlay chains; however, they must be distinct from the ROOT. The terminating -E switch will fixup the last chain loaded and prepare for the object file generation, if requested. Prior to its generation, the linker will store the address of the first available byte following all modules into the address of $MEMRY, if defined. The file specification of the overlays will also be added to $OVLNAM if defined (this is part of the OVERLAY/REL subroutine which has been provided).

MLINK provides support for up to 35 overlays. Each overlay is automatically assigned a file specification which consists of the filename assigned by the -N switch and an extension of /OVx: x = 1,2,...,9,A,B,...,Z. MLINK provides a subroutine to automatically invoke a desired overlay. The protocol for its use is as follows:

1) Load reg_HL with any root program data,
2) Load reg_DE with a pointer to a File Control Block (This is normally a 32-byte space under all systems except TRSDOS 1.3),
3) Load reg_A with the desired overlay in ASCII (i,e, LD A,'1'),
4) Invoke the OVERLAY handler via a CALL OVERLAY assembler instruction.

Note that return is not made from this overlay handler routine unless an error is detected in loading the requested overlay! However, the RETurn address to OVERLAY is in register DE at the time control is passed to the overlay. You therefore may find it useful to CALL a routine which CALLs OVERLAY. This is illustrated in the following set of programs coded for operation under TRSDOS 6:

```
;TESTOVER/ASM
        CSEG
BEGIN   LD      HL,START$
        LD      A,10            ;@DSPLY line handler
        RST     40
        LD      A,'1'
        CALL    GETOVER
        LD      A,'2'
        CALL    GETOVER
        LD      A,'3'
        CALL    GETOVER
        LD      HL,FIN$
        LD      A,10            ;@DSPLY line handler
        RST     40
        RET
GETOVER LD      DE,FCB
        CALL    OVERLAY##       ;OVERLAY doesn't return unless error
        OR      80H             ;Set ABORT bit
        LD      C,A
        LD      A,26            ;@ERROR handler in DOS
        RST     40
START$  DB      'This is the root executing',13
FIN$    DB      'End of processing',13
        DSEG
FCB     DS      32
        END     BEGIN

;TESTO1/ASM
        CSEG
        LD      HL,MESS$
        LD      A,10
        RST     40
        RET
MESS$   DB      'This is overlay 1',13
        END

;TESTO2/ASM
```

```
        CSEG
        LD      HL,MESS$
        LD      A,10
        RST     40
        RET
MESS$   DB      'This is overlay 2',13
        END

;TESTO3/ASM
        CSEG
        LD      HL,MESS$
        LD      A,10
        RST     40
        RET
MESS$   DB      'This is overlay 3',13
        END
```

The following MLINK session will create the root and overlay /CMD files as DOOVER/CMD, DOOVER/OV1, DOOVER/OV2, and DOOVER/OV3 respectively:

```
mlink
testover:1,overlay:2
-o,testo1:1
-o,testo2:1
-o,testo3:1
-n=doover:1 -e
```

Error and Status messages:

Any of the following error messages may be displayed during processing of the link session.

2nd COMMON larger /symbol/

The first reference to a common must be the largest.

Bad parameter(s)

The parameter of a switch was invalid for the switch.

Break detected

You depressed the <BREAK> key.

File not found

The referenced file was not found.

Filespec required

The switch required a filespec.

Internal Error

This error should not appear with properly assembled REL modules. If this message does appear, save all of the REL modules and assembler source modules loaded and contact MISOSYS.

Invalid command

The command character is not acceptable.

Invalid link item

Something is wrong in the REL bit stream.

Invalid switch

The switch entry is not supported.

Multiply defined symbol

A symbol has been defined at more than one address. When this occurs while a module is being loaded, the message will be prefixed with the name of the symbol. The -M switch appends an asterisk to the names of multiply defined symbols. Symbols are generally in this category if two or more modules declare them as PUBLIC, or if symbol names greater than seven characters in length are truncated to seven characters by the assembler and thus are no longer unique.

Non-contiguous image file

File output generated with the -I switch must present a contiguous sequence of load addresses. Thus, if you use the -P, -D, or -C switches, make sure that the address ranges of each segment connect. Use the -R switch and start over.

Record 1F too long

The 1F record text exceeds 255 characters.

Symbol table overflow

There is no more buffer room. Use the virtual memory switch. If you already are, you're probably out of luck. Try combining lots of little modules into one big module. Read the section on the -V switch.

Unknown COMMON referenced

A named common has been referenced without it being defined. Most likely a REL bit stream error.

Operating MLIB in interactive mode

MLIB functions in two different modes, interactive and batch. In the interactive mode you can enter commands one at a time. MLIB will process the command, and the results will be displayed when the command is completed. In batch mode, MLIB will get its command input from an existing file containing the sequence of commands. Batch mode is functional only under those operating systems which support the "DO" command.

From "DOS Ready", the librarian is invoked simply by entering:

```
MLIB (JCL,PAGE=nn)
MLIB *

JCL     - is used to specify BATCH mode.

PAGE=nn - specifies page length for listings and can
          range from 30-255 [make entry in decimal].
          If PAGE is omitted, it will default to 66.

*       - indicates a re-entry to MLIB (see text)

Abbreviations: J=JCL, P=PAGE
```

Upon successful loading and execution of MLIB, a command menu similar to the following is displayed:

```
 Ri|cl|in|TM
          MLIB 3.1L - Relocatable Object File Librarian
          (C) Copyright 1983 Riclin Computer Products

     X Load library              Save Library
       Add module                Module map
       Purge module              DOS Command
       Replace module            Clear buffer
       Extract module            eXit program
       Insert before module

             Bytes used:  0, free: 29736
```

To execute a command, enter the first letter of the command (e.g. to select "<L>oad library", enter the letter <L>). The only exception to this is the "e<X>it program" command, which is selected by typing the letter <X>. Alternatively, you may find it simpler to move the blinking cursor until it is next to the desired command, and then depress <ENTER> to execute that function. This is accomplished by using either the <DOWN-ARROW> and

<UP-ARROW> keys, or equivalently, the <SPACE> and <BACKSPACE> keys. The blinking cursor will reappear as a solid block next to the selected command, as a visual indication of which command is executing.

MLIB will accept either uppercase or lowercase for all keyboard input. Whenever MLIB asks you to enter a file specification, you do not normally have to enter an extension; it will default to either "/IRL" or "/REL", depending on whether you request Indexed ReLocatable modules or RELocatable modules at the "IRL or REL ?" prompt.

Certain options require you to enter either a file specification or a module name. MLIB will fully buffer your keystrokes. Several special function keys are available:

Key entry	**Key function**
<LEFT-ARROW>	non-destructive backspace
<RIGHT-ARROW>	non-destructive forward space
<SHIFT-CLEAR>	erase from cursor position to end of line
<SHIFT-LEFT-ARROW>	restart from scratch
<SHIFT-RIGHT-ARROW>	move cursor to end of line
<BREAK>	cancel this command
<ENTER>	invoke your entry.

If an error has occurred, MLIB will "beep" the console speaker if your machine is so equipped (on a Model I/III computer, a short beep tone will be directed to the cassette port which can be audible if you have an audio amplifier and speaker hooked up). An error message will appear at the bottom of the screen, and will remain there until you enter any keystroke, indicating that you have acknowledged the error.

Re-entering MLIB

If you have exited MLIB accidentally without saving the buffer to disk, or your system has rebooted itself, or you were forced to reboot due to your system crashing, MLIB may be re-entered by the command "MLIB *". All command line parameter values will be retained as per the previous invocation of MLIB. In nearly all cases your buffer will be intact. However, it is wise to immediately request a detailed module map of what is now in the buffer. If strange things appear in the map, then you have lost your data. If things look normal, then you should save the buffer to disk right away. Prudent practice dictates that you keep backups of your work, and if your system has hardware problems, save the buffer to disk frequently to prevent loss of data. If a reboot occurs or is necessary, don't forget to hold down the <ENTER> key to prevent any AUTO command from being executed and possibly overwriting the MLIB data.

MLIB commands

<A>dd module

This command adds a new module to the library in memory from a disk file. The new module will be loaded following what is already in memory. Use of this command is the primary method of building a new library or adding to an existing one. It will abort on out of memory, symbol table overflow, or disk error.

<C>lear buffer

This command resets all internal pointers, thus effectively destroying what is currently in the memory buffer. In interactive mode, MLIB gives you a chance to cancel this command if you decide not to clear the buffer.

<D>OS command

This allows you to enter any DOS library command. You will be prompted to enter the command with the message, "Enter DOS command:".

<E>xtract module

This command extracts a single module from the library in memory and writes it to disk as a stand-alone /REL or /IRL file. It will abort on module not found, or disk error.

<I>nsert before module

This command inserts a new module from disk into the library in memory. Answer the first prompt with the name of the module BEFORE which you wish to insert the new module. This command will abort on module not found, disk error, out of memory, or symbol table overflow.

<L>oad library

This command loads an existing library into memory from a disk file. It will abort on disk error, out of memory, symbol table overflow, or data already in memory.

<M>odule map

This command maps the attributes of the library which is currently in memory. This listing can be in either detailed or summary format. When you request the MODULE MAP command, you will be asked if you want a printer listing with the question:

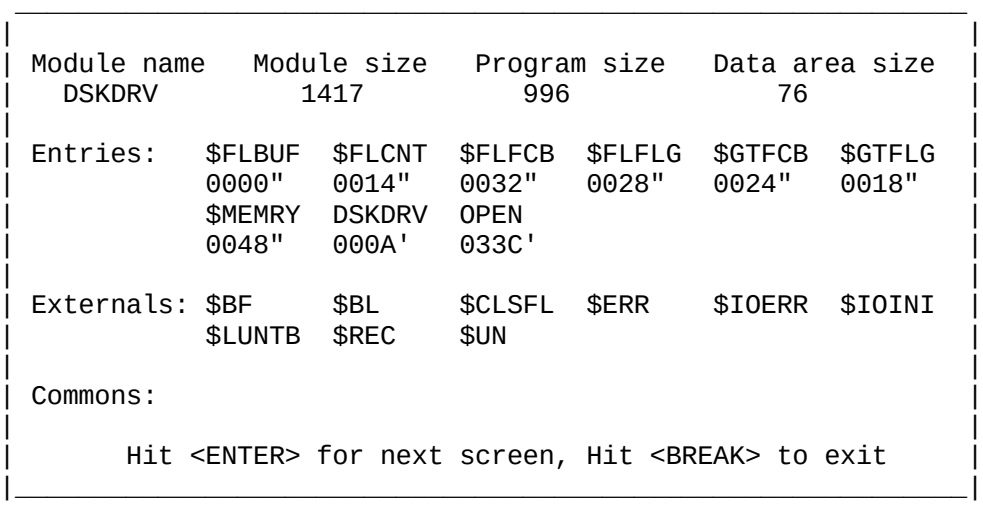
Listing to printer (Y,N) ?

You can print a module map, if desired, by responding <Y>. Printed output is paginated at a page length of 66 lines unless changed by use of the PAGE command line parameter. In either interactive or batch mode, <BREAK> will cancel a printout at any time.

In interactive mode, with certain exceptions, MLIB will prompt you to set your printer to the top of a page - do so and then depress any keystroke to continue. If you are running MLIB under LDOS, you can use the system spooler, or you can have printer output routed to a file. In either case, output will begin with a form feed, and there will be no prompt for top-of-form.

In batch mode, the module map will always go to the printer, and will begin with an automatic form feed and no prompt for top-of-form.

The detailed format option is available in both interactive and batch modes. The attributes of each module are displayed as shown in the following illustration:



Module name

This is shown if it exists; otherwise "------" is displayed. If the module is written with MRAS, the name is defined by use of the NAME pseudo-op or will default to the REL file name.

Module size

This is the absolute length, in decimal, of the module, including all segments, symbol definitions, etc. A stand-alone disk file containing the module would have the same length plus one.

Program size

This is the length of the module's program segment, in decimal.

Data area size

This is the length of the module's data segment, in decimal.

Entries

Each entry point is defined by a symbol and its location within the module. A module may have more than one entry point. MLINK entry symbols may be up to seven characters in length. Locations are expressed in hexadecimal, relative from the starting point of the segment in which the entry point is contained (X'0000'). The entry point type is also displayed as follows:

<space>	absolute segment
'	program relative (code) segment
"	data relative segment
!	COMMON area

Externals

Any symbol which is not defined in the module is called an external symbol, and is listed here. These are actually references to entry points defined in other modules.

Commons

Any COMMON areas defined in the module are listed. The lengths of the COMMON areas are also shown, in hexadecimal.

The summary format option lists all modules, by name only, in the order they exist in memory (from left to right, top to bottom, on the CRT screen). Summary format is available only in interactive mode and would look like:

```
 _____________________________________________________________________
|                                                                     |
|  RAN     INT4    DSQRT   DMOD    DSIGN   DABS    DATAN2    |
|  DATAN   DLOG10  DCOS    DSIN    DBLEXP  DEXP    DLOG      |
|  DMIN1   DMAX1   DCOMP   DBLDIV  DBLFLR  DBLPLY  DBLUTL    |
|  DMLDV   DSHR    DBLCON  MFM     AMIN0   MIN0    AMAX0     |
|  TANH    SQRT    MIN1    MAX1    MAX0    IFIX    FLOAT     |
|  EXP     DIM     COSIN   ATAN2   ATAN    AMOD    AMIN1     |
|  AMAX1   AINT    ABS     ALOG10  ALOG    MOD     RIEXP     |
|  NEGF    RREXP   NEG      NEG     EXPB    LOG2    POLY      |
|  FADD    IIEXP   FDIV    STOP    FMUL    ADE     FLR       |
|  FCP     FLT     CMPGTO  IDIV    SIGN    RAT     DAT       |
|  ISIGN   IABS    NEGATE  XAR     IDIM    FAT2    IMUL      |
|  POKE    UNPACK  PSPLAC  ICP     FORMIO  NORM    SHIFT     |
|  RNDOVF  ZAC     IOINIT  LUNTB   IAT2    LODSTR  SAF       |
|                                                             |
|       Hit <ENTER> for next screen, Hit <BREAK> to exit      |
|_____________________________________________________________|
```

<P>urge module

This command purges a module from the library in memory. If the module has multiple entry points, all will be deleted. In response to the prompt, enter a module name. The module will then be purged, and the library in memory will be compressed to recover freed space. This command will abort if the module is not found in the symbol table.

<R>eplace module

This command replaces an existing module in memory with a new version from a disk file. All aliases will be replaced or deleted. The new module may be longer, shorter, or the same length as the old one - MLIB will compress, overlay in place, or expand the library in memory to fit the new length. This command will abort on module not found, out of memory, symbol table overflow, or disk error.

<S>ave library

This command saves the entire library in memory to a disk file. MLIB will prompt you, if the file already exists, as to whether you wish to over-write or not. This command will abort on memory empty, or disk error.

e<X>it program

This command exits MLIB and returns you to "DOS Ready". In interactive mode, if there is data in the buffer which has not yet been saved to disk, you have a chance to recover (i.e abort the eXit). If you choose to exit anyway, your data will be lost, unless you re-enter MLIB immediately with the "MLIB *" command, as explained in OPERATING MLIB IN INTERACTIVE MODE. If the data has been saved, MLIB will exit immediately without a prompt.

Operating MLIB in batch mode

The command which executes MLIB in batch mode is as follows:

```
MLIB (JCL,PAGE=nn)

JCL      -  specifies BATCH mode.

PAGE=nn  - as shown earlier.

Abbreviations: J=JCL, P=PAGE
```

In batch mode, MLIB will take its commands from a batch file, whose extension is usually '/JCL'. Under some systems, the extension may be '/BLD'.

MLIB will request each command with the prompt "Enter option:". The valid responses to this prompt are: Add, Clear, Extract, Insert, Load, Map, Purge, Replace, Save, or eXit. Only the first letter of each command is significant; the remaining letters are ignored and may be left out of the command line. Command lines may also be commented. Each command line must be terminated with a carriage return.

Prompts may be given to, "Enter filespec:" which expects a JCL line containing a file specification; and "Enter module name:" which expects a JCL line containing the name of a module which is currently in the symbol table.

The "Module map" command will default to a detailed map which will be printed. The listing will start with a form feed.

Any command which writes out a disk file will automatically overwrite an existing file; you will NOT be given the chance to abort this command.

The "Clear buffer" and "eXit program" commands will do just that, with no opportunity to abort.

The occurrence of ANY error is fatal, and will terminate the batch processing via the DOS ABORT error return.

Here is an example of a typical series of commands for a batch operation. The comments may be included in the batch file.

Batch command	**Comment**
mlib (jcl)	execute MLIB in batch mode
load	load a library...
irl	designate an "IRL"
funcs2	this one is FUNCS2/REL
add	add a new module...
rel	designate a "REL"
shelsort	SHELSORT/REL
replace	replace a module...
index	module name is INDEX
rel	designate a "REL"
index	replace it with INDEX/REL
save	save the new version of...
irl	designate an "IRL"
funcs2	FUNCS2/REL
xit	and exit to DOS

Error messages

MLIB will recover from all non-fatal errors; you will not lose your data. Standard disk I/O errors may also occur, and MLIB will recover if at all possible. Fatal errors result in an immediate return to DOS without warning. In batch mode all errors become fatal, and will terminate execution.

Non-fatal errors:

Buffer empty

You cannot save the library, since memory is empty!

Buffer not empty

You cannot load a library if one is already in memory.

Invalid file format

You tried to read in a file which was not in the expected /REL format.

Module not found

A module name was not found; it was not in memory.

Out of memory

You cannot load a disk file because the file length would exceed available memory. When replacing or inserting a module, there must be enough free memory so that the new module can be read in before it is eventually relocated to its correct position in the buffer. If you do run out of memory, it's time to consider splitting the library into two smaller libraries.

Symbol table overflow

MLIB handles up to 200 individual modules; you have exceeded this limit.

Fatal errors:

Can't - buffer destroyed!

MLIB determined that part of the memory buffer was destroyed before MLIB was re-entered with the "*" parameter.

Invalid option

An invalid command was given to MLIB while in batch mode.

Parameter error!

An invalid DOS command line parameter was entered. The only acceptable command line parameters are "PAGE" which requires a decimal entry in the range 30 through 255, and "JCL", which can be either ON or OFF.

Unrecoverable error!

If a file has been read into memory which appears ON THE SURFACE to be a /REL-format file, but really isn't, there is a high probability that an unrecoverable error will occur during symbol table update.

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:

```
-------+-------+-------+-------+-------+-------+-------+-------+-------+-------
   Srch  Repl  Again  All   Rev   Hex  Quote Copy  Move   Cmd  Print Exit
   ==1== ==2== ==3== ==4== ==5== ==6== ==7== ==8== ==9== ==0== ==:== ==-==
    Ins  Line   Del  Word  Block Load  Save  Macro File   Meta  Pg Dn Pg Up
File:aaaaaaaaaaaaaa Len:bbbbb/ccccc Ln:ddddd Col:ee = x'ff' ggg%  Banks: hhhhhhh
Srch: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 MRAS. 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 MRAS (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 respecify 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.

Tips for programming relocatable modules

Make a module's /REL filename the same as its program NAME and its main ENTRY point. This will help you build libraries in an organized way.

Avoid writing modules with multiple entry points. Modules written in this way can often lead to confusion and inefficient programming.

Library sizes are absolutely limited by the size of MLIB's memory buffer. In practice, however, a smaller library size of about 10K bytes is most useful for three reasons: (a) MLINK library searches will take less time; (b) the library will fit into MLIB's buffer even if you have a lot of things in high memory; (c) you will have some room for library growth.

Build special purpose libraries (e.g. communications, graphics, string processing, disk I/O).

Use MRAS's CSEG (code segment), DSEG (data segment), and COMMON pseudo-OPs only! DO NOT use ASEG (absolute segment), as this produces object code which is not relocatable and thus not usable in a general purpose library.

Libraries of related routines must be constructed in a particular order. This order is determined by each module's external references. If, for instance, module A references module B, then module B MUST FOLLOW module A in the library. Otherwise, a backward reference will occur, and MLINK may force you to search the library TWICE in order to satisfy the reference. If the library is constructed as an IRL, MLINK will perform multiple searches automatically; however, minimum processing time will occur in searching an IRL when no external reference is to be resolved by a module which is stored before the module having the extern.

You may encounter a problem if you try to use MLIB on libraries not created with MLIB. This is because the end-of-file marker in these files may not match the EOF in the directory. There is a simple solution: create a text file using SAID which contains a single character of value 9EH; append this file to the problem file; load the /REL file into MLIB, and immediately re-save it. This caveat also applies to any /REL files constructed under CP/M which have been transferred over to your system.

Microsoft compatible 'REL' format

All Z80 assemblers work in a similar fashion, in that they convert a file containing SOURCE CODE, written in Z80 assembly language mnemonics, to OBJECT CODE in some binary format. In ABSOLUTE assemblers, this binary data is a faithful representation of the actual machine language (ones and zeros) that the Z80 will execute when you want your program to run. This object code can only load and execute at a FIXED address in the Z80's memory space. On the other hand, a RELOCATABLE assembler, such as MRAS, will generate object code which can be relocated to any address in the Z80's 64K memory space before the program is to be executed. MRAS and MLINK support a Microsoft compatible relocation format.

Let's look at an example of absolute assembly. The following program has been assembled at an ORIGIN of 0100H. Notice especially the values assigned to the memory addresses @DATE, @EXIT, @DSPLY, START, and BUFFER:

```
0100           00100          ORG     0100H
4470           00110 @DATE    EQU     4470H
402D           00120 @EXIT    EQU     402DH
4467           00130 @DSPLY   EQU     4467H
000D           00140 CR       EQU     0DH
0100 211401    00150 START:   LD      HL,BUFFER
0103 CD7044    00160          CALL    @DATE
0106 3E0D      00170          LD      A,CR
0108 321C01    00180          LD      (BUFFER+8),A
010B 211401    00190          LD      HL,BUFFER
010E CD6744    00200          CALL    @DSPLY
0111 C32D40    00210          JP      @EXIT
0114           00220 BUFFER:  DS      9
0100           00230          END     START

@DATE            4470 @DSPLY          4467 @EXIT          402D
BUFFER           0114 CR              000D START          0100
```

The program has been reassembled below at a new origin, 0200H. Some of the addresses for the above labels have changed, while some remain the same:

```
0200           00100          ORG     0200H
4470           00110 @DATE    EQU     4470H
402D           00120 @EXIT    EQU     402DH
4467           00130 @DSPLY   EQU     4467H
000D           00140 CR       EQU     0DH
0200 211402    00150 START:   LD      HL,BUFFER
0203 CD7044    00160          CALL    @DATE
0206 3E0D      00170          LD      A,CR
0208 321C02    00180          LD      (BUFFER+8),A
020B 211402    00190          LD      HL,BUFFER
020E CD6744    00200          CALL    @DSPLY
0211 C32D40    00210          JP      @EXIT
0214           00220 BUFFER:  DS      9
0200           00230          END     START

@DATE            4470  @DSPLY          4467  @EXIT          402D
BUFFER           0214  CR              000D  START          0200
```

To be specific, START and BUFFER have changed, while the others are unchanged. Both START and BUFFER have been relocated! START, instead of being at 0100H is now at 0200H, and BUFFER has moved from 0114H to 0214H. This offset of 0100H is due to the changed origin, 0100H versus 0200H. START and BUFFER are therefore internally relocatable values, while @DATE, for example, will always be 4470H, and is thus known as an absolute value.

The same program, as assembled using relocation looks like this:

```
4470                          @DATE   EQU     4470H
402D                          @EXIT   EQU     402DH
```

```
4467                            @DSPLY  EQU     4467H
000D                            CR      EQU     0DH
0000'   21 0014'                START:  LD      HL,BUFFER
0003'   CD 4470                         CALL    @DATE
0006'   3E 0D                           LD      A,CR
0008'   32 001C'                        LD      (BUFFER+8),A
000B'   21 0014'                        LD      HL,BUFFER
000E'   CD 4467                         CALL    @DSPLY
0011'   C3 402D                         JP      @EXIT
0014'                           BUFFER: DS      9
                                        END     START

@DATE           4470  @DSPLY          4467  @EXIT           402D
BUFFER          0014' CR               000D  START          0000'
```

All of the internal program addresses have been assembled as if the program had an origin of 0000H, and are noted with a following single-quote ('). This is relocation at work. The binary output of this assembly (a /REL file) cannot be executed by the Z80 until you choose an origin for the program; this is done by the MISOSYS linker, MLINK), and can be ANY address in the Z80 memory space. MLINK will determine, from the origin you have selected, where START and BUFFER really will be when the program is run. If you choose 0100H as the origin, then START will be located at 0100H, and BUFFER at 0114H. Other origins will produce similar results; START and BUFFER will be at different addresses, but the offset between them (0014H) will always be the same.

This characteristic of relocatable object files, that they can be LINKED at any origin, is extended by a further capability: relocatable object files may be linked TOGETHER to form a complete program from many smaller pieces. This allows you to write a very large program in lesser chunks which are easier to edit and to understand. In addition, you can develop libraries of standard and useful subroutines, each thoroughly tested and debugged, which any main program may call upon when necessary. The Microsoft FORTRAN library (FORLIB/REL), for example, thus contains many subroutines which can be used by any FORTRAN or Z80 assembler program.

The mechanism of program and subroutine linkage that is often used is implemented by the ENTRY and EXTERNAL attributes. A label which is declared ENTRY (or GLOBAL or PUBLIC) in one module can be accessed by another module in the following way:

```
;Module 1
        ENTRY   LABEL1          ;this is an entry point
LABEL1:
        <code follows>
        END                     ;end of module 1

;Module 2
        EXTRN   LABEL1          ;this is an EXTERNAL declaration
                                ;could also be EXT.
        CALL    LABEL1          ;and this is a reference to the
                                ;external
        END                     ;end of module 2
```

The relocatable format also allows you to do other things. In many applications, program code and data areas must be separated. This most often occurs when code must be placed in ROM, such as the BASIC interpreter in a TRS-80. However, the data areas cannot be in ROM; they must be in writeable memory (RAM), and thus must be separated from the code areas. This can be accomplished by use of the CSEG and DSEG commands to the MRAS assembler. A CSEG pseudo-operation signals the start of a code area, while a DSEG indicates the start of a data area. Code and data SEGMENTS may be intermixed in a program source file, and the assembler will automatically keep them separate by the use of two distinct program or location counters, one for each segment. When you link the program with MLINK, you may tell the linker at what address to place the code, and also where to place the data. Thus the two segments are separated. The above example is shown below using this technique:

```
4470                            @DATE   EQU     4470H
402D                            @EXIT   EQU     402DH
4467                            @DSPLY  EQU     4467H
000D                            CR      EQU     0DH
0000'                                   CSEG    ;code starts here
0000'   21 0000"                START:  LD      HL,BUFFER
0003'   CD 4470                        CALL    @DATE
0006'   3E 0D                          LD      A,CR
0008'   32 0008"                        LD      (BUFFER+8),A
000B'   21 0000"                        LD      HL,BUFFER
000E'   CD 4467                        CALL    @DSPLY
0011'   C3 402D                        JP      @EXIT
                                        DSEG    ;data starts here
0000"                           BUFFER: DS      9
                                        END     START

@DATE           4470  @DSPLY           4467  @EXIT           402D
BUFFER          0000" CR               000D  START           0000'
```

Notice how the label BUFFER is now located at 0000H, but in the data segment, as indicated by the double-quote (") following the address. Am MLINK session could then be as follows with user entries in BOLDFACE:

```
DOS Ready
MLINK
MLINK - Ver 1.0a Copyright 1985 MISOSYS, Inc., All rights reserved
?-p=100
?-d=1000
?test
27937 Free space
P <0100-0113 0014> D <1000-1008 0009>
*test-n-e
DOS Ready
```

The -p command to the linker established the program (or code) segment origin, while the -d command did the same for the data segment. After loading TEST/REL with the next command, the linker then tells us where the two segments are located and how long they are. The final command writes out an

executable command file (/CMD). If we were to disassemble TEST/CMD, we would find that START is located at 0100H and BUFFER at 1000H. Thus the program is separated into ROM and RAM sections.

MLINK has other capabilities, such as the use of COMMON blocks, which are explained in sections on MRAS and MLINK. MRAS can also generate absolute code, if you use the ASEG command.

Finally, we get to the actual format of a Microsoft relocatable object file. A /REL file is composed of a bit (not byte) stream. Each /REL file may contain a table of ENTRY points and EXTERNAL references. Each ENTRY point is identified by its name (1 to 7 ASCII characters) and its relative location within one of the module's code, data, or common segments. Each EXTERNAL reference is identified by its name, and also by a chain (or linked list) of pointers, each of which locates the relative address within the module where the external was used. The last pointer in the chain is zero. The /REL file also contains internal relocation data necessary for resolution of label references within the module. All external and internal relocatable references are changed to absolute values at link time, when the program's segment origins have been established. The remainder of the information in the /REL file consists of absolute code and data bytes which do not need relocation, and numerous other fields which describe common blocks, the module name, the module segment lengths, and the /REL file end (or EOF byte). A library file would contain many such modules, each separated by program end indicators, and terminated by an EOF byte.

Let's take one last look at our example, modified slightly, to see what the relocatable object file assembled from this source code would look like:

```
                                NAME    ('TEST')
4470                            @DATE   EQU     4470H
402D                            @EXIT   EQU     402DH
4467                            @DSPLY  EQU     4467H
000D                            CR      EQU     0DH
                                        CSEG    ;code starts here
                                        ENTRY   START
                                        EXT     MESSAGE
0000'   21 0000"                START:  LD      HL,BUFFER
0003'   CD 4470                        CALL    @DATE
0006'   3E 0D                          LD      A,CR
0008'   32 0008"                       LD      (BUFFER+8),A
000B'   21 0000"                       LD      HL,BUFFER
000E'   11 0000*                       LD      DE,MESSAGE
0011'   01 0009                        LD      BC,BUFFLEN
0014'   ED B0                          LDIR
0016'   21 0000*                       LD      HL,MESSAGE
0019'   CD 4467                        CALL    @DSPLY
001C'   C3 402D                        JP      @EXIT
                                        DSEG    ;data starts here
                                        ENTRY   BUFFER
0000"                           BUFFER: DS      9
0009                            BUFFLEN EQU     $-BUFFER
                                        END     START
```

```
@DATE           4470  @DSPLY          4467  @EXIT           402D
BUFFER          0000" BUFFLEN         0009  CR              000D
MESSAGE         0017* START           0000'
```

Notice how the external label, MESSAGE, is defined in the symbol table; the value 0017H represents the relative location of the LAST reference to MESSAGE in the assembled code, and the trailing asterisk (*) denotes an external symbol both in this table and in the assembled code listing.

The following is a tabular picture of the decoded /REL file. Each column represents:

(1) Absolute [0] or relocatable [1] item [1 bit]. If absolute, column (2) shows the value in hex [8 bits].
(2) Relocation type [0 = special link item; 1, 2, or 3 = segment relative] [2 bits]. See column (8).
(3) Special link item control field in decimal [4 bits]. See column (8).
(4) "A-field" address type, same as column (2) [2 bits].
(5) "A-field" value, displayed as high/low, but reversed in file [16 bits].
(6) "B-field" length [3 bits].
(7) "B-field" symbol in ASCII [8 bits each character].
(8) Description of the object file record as decoded.

```
(1) (2)  (3)  (4)    (5) (6)   (7)                  (8)
--- ---  ---  ---  ------ -- -------  ---------------------------------
 1    0    2              4  TEST     program name
 1    0    0              5  START    entry symbol for library search
 1    0    0              6  BUFFER   entry symbol for library search
 1    0   10    0   00 09             define data area size
 1    0   13    1   00 1F             define program size
 1    0   11    1   00 00             set loading location counter (code)
 0   21                               absolute (1st byte in code segment)
 1    2             00 00             data relative (ref. to BUFFER)
 0   CD                               absolute
 0   70                               absolute
 0   44                               absolute
 0   3E                               absolute
 0   0D                               absolute
 0   32                               absolute
 1    2             00 08             data relative (ref. to BUFFER+8)
 0   21                               absolute
 1    2             00 00             data relative (ref. to BUFFER)
 0   11                               absolute (ref. to MESSAGE follows)
 0   00                               absolute (this plus next byte are
 0   00                               absolute  end of external chain)
 0   01                               absolute
 0   09                               absolute
 0   00                               absolute
 0   ED                               absolute
 0   B0                               absolute
 0   21                               absolute (ref. to MESSAGE follows)
 1    1             00 0F             program relative (link in chain)
 0   CD                               absolute
 0   67                               absolute
```

```
0   44                                  absolute
0   C3                                  absolute
0   2D                                  absolute
0   40                                  absolute
1    0   11   2    00 00                set loading location counter (data)
1    0   11   2    00 09                set loading location counter (data)
1    0    7   2    00 00 6  BUFFER      define entry point (data)
1    0    6   1    00 17 7  MESSAGE     chain external (head of list)
1    0    7   1    00 00 5  START       define entry point (code)
1    0   14   1    00 00                end program (force to next byte)
1    0   15                            end file marker
```

Relocation Format of /REL files

The REL file is an encoded bit-stream containing relocatable object code information. It follows the format documented by Microsoft for the M-80 assembler and L-80 linker; however, only 16-bit externals are supported and the linker currently does not support chain address. The following text documents the bit stream supported by MLINK.

1) IF the next bit is a zero, THEN the following eight bits are loaded according to the value of the location counter currently in effect, THEN recycle to 1).

ELSE IF the next bit is a one, THEN the next two bits represent a code which is interpreted as follows:

01 - indicates a code relative value follows. The next 16 bits are loaded after being offset by the code segment origin, THEN recycle to 1).

10 - indicates a data relative value follows. The next 16 bits are loaded after being offset by the data segment origin, THEN recycle to 1).

11 - indicates a common relative value follows. The next 16 bits are loaded after being offset by the selected common segment origin, THEN recycle to 1).

00 - Indicates a Special Link item. The SL item consists of the following four bits which are interpreted as one of 16 different items described below; an optional VALUE field which consists of a 2-bit address type [00 = absolute, 01 = code relative, 10 = data relative, 11 = common relative] and a 16-bit address; and an optional NAME field that consists of a 3-bit name length followed by the name in 8-bit bytes. SLs 0000-0100 use only a NAME field; SLs 0101-1000 use both a VALUE field and a NAME field; SLs 1001-1110 use only a VALUE field; SL 1111 has neither a NAME nor a VALUE field. Unless otherwise specified, at the conclusion of processing a special link item, processing recycles to 1). The Special Link items are as follows:

0000 - indicates an entry symbol. This is used by the linker only when searching a library to see if the module is needed to satisfy an undefined extern.

0001 - Select Common Block. Used to specify the NAMEd Common Block for subsequent common relative references.

0010 - Module name. This is the name of the module. The first one encountered is saved by MLINK for use in generating the optional HEADER record of the /CMD file.

0011 - Request Library Search. The library designated by the NAME field will be searched to resolve undefined externals prior to any object code generation. An REL will be first assumed. If one is not found, an IRL will then be assumed.

0100 - This item is not supported by MLINK.

0101 - Define Common Size. This is used by MLINK to establish the size of the common block designated by the NAME field.

0110 - Chain External. The VALUE field contains a pointer to the head of a chain which ends with an absolute zero. Each 16-bit element of the chain will be replaced with the value of the external symbol described in the NAME field.

0111 - Define Entry Point. The VALUE field specifies the value of the symbol described by the NAME field.

1000 - This item is not supported by MLINK.

1001 - External plus Offset. This specifies that the VALUE field must be added to the following two bytes in the current segment after all chain externals have been processed.

1010 - Define Data Size. The VALUE field is used by MLINK to establish the size of the data segment of the current module.

1011 - Set Location Counter. The location counter is set to the value identified by the VALUE field.

1100 - This item is not supported by MLINK.

1101 - Define Code Size. The VALUE field is used by MLINK to establish the size of the code segment of the current module.

1110 - End of Module. The VALUE field defines the transfer address for the module if other than absolute zero. This item denotes the end of the module. The bit stream is also advanced to a byte boundary. Recycle to 1) if loading a module from other than a library search.

1111 - End of File. This is used to indicate the end of the file. It is used when searching libraries or when loading modules to detect the end of the file.