ZASM ASSEMBLER USER INFORMATION

FOREWORD

This documentation assumes that the reader has some familiarity with assembly language for the Z-80: it is not a primer. The best Z-80 assembly language book that I have found is Rodnay Zaks' "How to Program the Z-80". Another excellent book to supplement it is "Z80 Assembly Language Subroutines" by Lance A. Leventhal and Winthrop Saville.

There are also occasional references herein to the standard CP/M assembler ASM. For example the DL pseudo of ZASM is compared to the SET statement of ASM. Many of the pseudo instructions are the same or similar in the two assemblers, although ZASM is far more powerful.

Finally two notes on the examples in this file:

* They have NOT all been tested and I do make mistakes.

* They have been written in uppercase only to make them stand out. There seems to be a covenant among assembler language programmers to do everything, except comments, in uppercase. The fact is that, as with any other programming language, lowercase is easier to read and comprehend. ZASM will accept either case.

GENERAL

ZASM is a Macro Assembler for the Z-80 instruction set. It recognizes Zilog Z-80 instruction mnemonics and produces either .HEX or .REL (Microsoft standard) output. In short, it is a very powerful and very cheap (free) assembler. Found on a CP/M bulletin board without documentation, these notes are based on a disassembly as well as experimentation. There is no guarantee that they are entirely accurate or comprehensive.

Some features of note in ZASM are:

* The assembler builds a symbol table in pass one. In pass two the listing and code are generated. Because of its two pass nature, forward references in EQU and DL statements are permitted although they may result in 'Error in Pass 1' messages.

* Conditional assembly (IF/ELSE/ENDIF) nested to eight levels is permitted.

* Macros including REPT, IRP and IRPC. Nesting to eight levels is permitted. Macro libraries are supported.

* Include file nesting to four levels. Files may be included on pass one only if desired.

* Up to 15 named common blocks are allowed.

* Library search directives are allowed.

* Identifiers may include the special characters $ @ _ ? . which are regarded as significant. Names must begin with a letter or special character and may contain digits thereafter.

* Names are considered significant to eight characters BUT only seven characters are written to a .REL file and hence passed to the

linker. Hence only seven characters can be trusted if you are linking several modules. Furthermore, I am told that L80 (a Microsoft product only uses six characters so that identifiers differing only in the seventh character will look the same to it.

COMMAND LINE

The source file name must have the .Z80 extension. The command line is then of the form:

```
      ZASM PROG.sol ARG1 ARG2 ...
```

where the allowable arguments are given below and the letters 'sol' are:

s - Source device. This must be a disk drive in the range A-H.

o - Output device. This must be:

Z	no output
A-H	disk for output (.REL or .HEX)

l - List device. This must be:

X	console
Y	printer
Z	no listing
A-H	disk for .PRN file

An exception to the above rules is that if any of 'sol' are blank, the default disk drive is assumed. Hence:

```
      B>A:ZASM PROG           =     A:ZASM.BBB
      B>A:ZASM PROG.A         =     A:ZASM.ABB
      B>A:ZASM PROG.BZ  =     A:ZASM.BZB
```

Command Line Arguments

RANGE		Mark instances where JR could be used
PARITY		Mark all PE/PO/V/NV conditions (POTENTIAL 8080/Z80 conflict)
XREF		Produce a symbol cross reference in listing
NOXREF		Don't
SYMB		Print a symbol table
PAGE	= decno	Set page size for listing
TOP	= decno	Set top of form margin
WIDTH	= decno	Set page width
TRUNC	= decno	Set page width and truncate
MACRO	= name	Specify a macro library to load
COND		List conditionals
NOCOND		Don't
GEN		List macro generated code
NOGEN		Don't
TEXT		Show all bytes generated by instruction
NOTEXT		Show at most four bytes per instruction
LISTON		List source code
LISTOFF		Don't
OPCODE		Produce an opcode cross reference in listing
DEBUG		Does nothing but set an unused bit
HEX		Produce HEX output rather than REL
HEX	= hexno	Hex load address (sets HEX also)
DATE	= mmddyy	Set date for listing
TIME	= hhmmss	Set time for listing

Instruction Formats

The assembler differentiates between the label and the opcode fields by assuming that the instruction is of one of the following forms:

```
LABEL:      OPCODE      PARAMS            ;COMMENT
LABEL OPCODE      PARAMS            ;COMMENT
      OPCODE      PARAMS            ;COMMENT
      LABEL:      OPCODE      PARAMS      ;COMMENT
```

The thing to note is that an identifier that begins in the first column is assumed to be a label even though it may lack a colon. Hence opcodes, whether instructions or pseudo-instructions, should be tabbed over at least one. For example, the statements:

```
TITLE Test Program
      CR    EQU   13
```

will be flagged as errors. They should have been written:

```
      TITLE Test Program
CR    EQU   13
```

Assembler Pseudo Instructions

The following pseudo instructions require no special prefix characters as in some assemblers. There are four pseudo instructions, discussed later, that do require a special asterisk prefix.

In determining how to interpret the contents of the opcode field, ZASM searches in the following order:

- macro (hence macros may replace builtin opcodes)
- builtin opcodes

Most of these pseudos may not have a label - those that can are shown explicitly.

	ABS		Absolute program segment
	COM	name	Common block
	CONMSG	text	Displayed during pass 2 of assembly
	DATA		Data segment
[lbl]	DB	bytes	Define bytes
[lbl]	DEFB	bytes	Same
lbl	DL	value	Define label - same as SET in ASM
lbl	DEFL	value	Same
[lbl]	DM	bytes	Same as DEFB but sets bit 7 of last byte high
[lbl]	DEFM	bytes	Same
[lbl]	DS	size	Reserves size bytes of storage
[lbl]	DEFS	size	Same
[lbl]	DW	words	Define words (16 bit)
[lbl]	DEFW	words	Same
	EJECT		Page eject on listing
	ELSE		Used with IF and ENDIF
[lbl]	END	[lbl]	End statement (optional entry point)
	ENDIF		Used with IF ELSE
	ENDM		End of macro (includes REPT IRP IRPC)
	ENTRY	labels	Entry label - same as PUBLIC in RMAC
lbl	EQU	value	Equates label to fixed value (DL for variable)
	EXITM		Exit from macro before ENDM statement
	EXT	labels	Same as EXTRN
	EXTRN	labels	External labels

```
      FORM              Page eject (same as EJECT)
      GLOBAL      labels            Either EXTRN or ENTRY - assembler will
decide
      IF    expr        Used with ELSE ENDIF
      IRP   #a,b,c,           Indefinite repeat
      IRPC  #a,'abc'    Indefinite repeat by character
[lbl] JSYS  value       Strange - RST 1 followed by byte value
      LIST  params            Introduces list options
macnam      MACRO #a,#b,...   Macro definition
      MEND              Macro end (same as ENDM)
      MEXIT             Macro exit (same as EXITM)
      NAME  name        Name module - else same as name of source file
macnam      OMACRO      #a,#b,...   Like MACRO but name shows up in opcode
listing
      ORG   value       Origin - set program counter instruction
      REL               Relocatable code segment
      REM   text        Remarks statement (semicolon just as good)
      REPT  value       Repeat statements
      STRUCT      value       Define data structure
      SUBTTL                  Subtitle for listing
      TITLE             Title for listing
      TITLE2                  Same as SUBTTL
```

Standard Z-80 Mnemonics

The following are the Zilog mnemonics for the Z-80. The reader is referred to Rodnay Zaks' "How to Program the Z-80" for their syntax.

One special note is that index register offsets are calculated to 16 bits before testing to ensure that they are in the range (-128 to +127). Hence:

```
      LD    A,(IX + 0FFH)           is illegal

      LD    A,(IX + 0FFFFH)         is legal, as is
      LD    A,(IX - 1)
```

Another note is that conditional jumps and calls have a somewhat expanded syntax, illustrated below.

```
      JP    C,    =     JP    LT,
      JP    NC,   =     JP    GE,
      JP    Z,    =     JP    EQ,
      JP    NZ,   =     JP    NE,
      JP    PE,   =     JP    V,
      JP    PO,   =     JP    NV,
      JP    M,    =     no other
      JP    P,    =     no other
```

For example:

```
      LD    A,SAM
      CP    20
      JP    GE,DEST
```

means jump if SAM GE 20 - i.e. if A >= 20.

```
ADC   ADD   AND   BIT   CALL  CCF   CP    CPD
CPDR  CPI   CPIR  CPL   DAA   DEC   DI    DJNZ
EI    EX    EXX   HALT  IM    IM0   IM1   IM2
IN    INC   IND   INDR  INI   INIR  JP    JR
LD    LDD   LDDR  LDI   LDIR  NEG   NOP   OR
```

OTDR OTIR OUT OUTD OUTI POP PUSH RES
RET RETI RETN RL RLA RLC RLCA RLD
RR RRA RRC RRCA RRD RST SBC SCF
SET SLA SRA SRL SUB XOR

Expressions

Expressions are evaluated to 16 bits and may use the following operators as well as parentheses or brackets - i.e. () or []. Mnemonics that require parentheses such as LD A,(HL) or ADD A,(IX+3) may not use the [] form.

The priorities shown below are such that lower numbers connote higher priorities. For equal priorities evaluation is left to right.

Unary operators

OP	PRI	COMMENTS
+	1	unary plus
-	1	unary minus
^	1	2 ^ power (i.e. ^11 is a 16 bit word with bit 11 set)
~	4	not (ones complement)
NOT	4	not (ones complement)
LOW	8	low byte (high byte set to zero)
HIGH	8	high byte (swap bytes and set new high byte to zero)

Binary operators

+	3	add
-	3	subtract
*	2	multiply
/	2	divide
%	2	modulus (e.g. 13 % 5 = 3, 15 % 5 = 0)
&	5	and (bitwise)
|	6	or (bitwise)
>>	2	shift right (e.g. 80h >> 2 = 20h)
<<	2	shift left (e.g. 7h << 8 = 700h)
>=	7	ge
<=	7	le
<>	7	not equal
>	7	gt
<	7	lt
=	7	equal
MOD	2	mod - same as %
SHL	2	shift left - same as >>
SHR	2	shift right - same as <<
AND	5	and - same as &
OR	6	or - same as |
XOR	6	exclusive or - exclusive or (bitwise)
LT	7	lt - same as <
GT	7	gt - same as >
EQ	7	equal - same as =
NE	7	not equal - same as <>
LE	7	le - same as <=
GE	7	ge - same as >=

Expressions may use the $ symbol to refer to the value of the program counter. In such usage, $ = the PC at the beginning of the line in question. For example:

```
      START DL    $
```

```
        DS    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  NCHRS DL    $ - START   ;number of characters
```

Furthermore, relational operators may be used with character strings contained in double quotes. E.g.

```
  IF "SAM" < "GEORGE"
```

is meaningful (and false).

The mixing of types - e.g. absolute, relocatable, external etc in expressions is usually expressed in a number of complicated rules. A little common sense will normally suffice to see what will work. For example, the difference of two internal relocatable references is OK and will produce an absolute quantity (example - NCHRS above) since they will be given the same offset at link time. The sum of two such quantities is a no-no as is multiplication or division involving non-absolute quantities.

Listing Options

These can be embedded in the source file in the form:

```
        LIST  NOGEN,NOCOND,TEXT
```

When so embedded, they are overridden by contrary listing options contained in the command line.

GEN	NOGEN	Macro generated code
ON	OFF	Source code
COND	NOCOND	Conditionals (IF/ELSE/ENDIF)
TEXT	NOTEXT	Complete byte listing (else max 4 per instruction)

Detailed Discussion of Instruction Mnemonics

ABS

This specifies that the following code or data is to be placed into the absolute program section. The syntax is simply:

```
        ABS
```

COM

This specifies that the following code or data is to be placed into the common section of the given name. Presumably, common blocks of the same name declared in different modules will be assigned the same start address by the linker. The syntax is:

```
        COM   BLKNAME
```

where BLKNAME is a legal identifier of at most seven characters.

CONMSG

This is used to generate a message to the console during the second pass of the assembler. It does not generate any code and can hence not be used to generate run-time messages. For example:

```
  CPM   EQU   1
  ...
  IF    CPM = 1
  CONMSG      Assembly is for CP/M
  ELSE
  CONMSG      Assembly is for non-CP/M system
```

```
      ENDIF
      ...
```

DATA

This specifies that the following data is to be placed in the DATA area by the linker. It is normally used to separate dynamic data from program code. Such is essential when generating code to run in ROM. The syntax is simply:

```
      DATA
```

DB and DEFB

These are synonyms used to initialize data to given byte values. Note that if the initial value is meaningless, the DS statement is more appropriate, simply defining the number of bytes to be reserved. Examples are:

```
      DB    'THIS IS A STRING WITH A NULL BYTE',0
      DB    1,2,3,'SAM',4FH
```

DL and DEFL

This is like the SET statement in ASM. In several cases the writer of ZASM seems to have been relieving himself in a windward direction in his choice of names. It is used when the same symbol must be assigned more than one value at assembly time. This is of the greatest importance in writing macros but is useful elsewhere also. For example:

```
BASE  DL    0
      IF    NOT STANDARD
BASE  DL    4100H
      ENDIF
```

Of course in this case, an IF..ELSE..ENDIF construct would have allowed the use of EQU rather than DL.

In writing macros it is often necessary to have the value of a label vary. Two examples follow:

```
COUNT?      MACRO #STRING
N?    DL    0
      IRPC  #Z,#STRING
          IF      '#Z' = '?'
N?          DL    N? + 1
          ENDIF
      ENDM
      ENDM
```

invoked as, say:

```
      COUNT?      'A?BB???C'
      IF    N?
      CONMSG      Found some queries.
      ENDIF
```

would set N? to the number of ? characters in the argument string. In the macro expansion, N? occurs L + 1 times as a label where L is the length of the string. Futhermore, COUNT? may be invoked at more than one place. The example seems trivial because anyone can see that there are several ?'s in the string. The string might itself, however, be a macro parameter which sometimes contains queries and sometimes does not.

A final example is shown below. The LENGTH macro sets the symbol LEN to the length of the parameter. For example, after the statement:

```
      LENGTH      ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

the symbol LEN will have the value 26. This macro, like the last, generates no code - it simply sets a value which may then be used in expressions in the following code.

```
LENGTH      MACRO #NAME             ;length of a string
LEN   DL    0                 ;initialize to zero
      IRPC  #Z,'#NAME'        ;for each character in string
LEN   DL    LEN + 1                 ;...bump len
      ENDM                    ;end of irpc
      ENDM                    ;end of macro
```

DM and DEFM

This is just like DB except that the last byte specified, in a line, has bit 7 set to 1. The chief usage of this is providing character strings for use by routines that detect 'end of string' by checking bit 7 rather than using the more traditional method of following strings by null bytes. For example, using DM we can write:

```
      DM    'THE LAST BYTE HAS BIT 7 HIGH'

            rather than

      DB    'THE LAST BYTE HAS BIT 7 HIG','H'+80H
```

This method of indicating the end of character strings was extensively used by the writer of ZASM. It is also used in a number of text editors - for example, in its swap file Perfect Writer sets bit 7 on line feed characters. Wordstar does a lot with bit 7, even on the end of words.

DEFS and DS

These are used to specify only the amount of storage needed without providing initial values. It would be meaningless, for example, to initialize the contents of a disk input buffer. Areas specified by the DS statement may or may not be included in the .COM file by the linker. The syntax is as below:

```
      SAM:  DS    128
```

This assigns the label SAM to the start of a block of 128 bytes. No assumptions may be made about the initial value of locations specified by a DS directive. Some linkers will initialize such areas to zero and other linkers will simply leave garbage in them. Indeed such areas will not necessarily even be included in the final .COM file.

In this assembler, DS seems to have another function which will be discussed later under STRUCT.

DEFW and DW

This is used to reserve and initialize one or more words (16-bit) of storage. The syntax is:

```
      DW    32,SAM,'AB'
```

Note that this initializes the second word to contain a pointer to label

SAM.

EJECT and FORM

These both cause a page eject on the listing.

IF/ELSE/ENDIF

These are used in conditional constructs which may be nested to a depth of eight. It is used to control which portions of the program will actually be assembled. A typical use might be:

```
KAYPRO      EQU   1     ;set your computer to 1, the rest to zero
OSBORNE     EQU   0
IBM   EQU   0

WORTH:      IF    KAYPRO
            CONMSG      This is for a Kaypro - Hurrah !
      ELSE
            IF    OSBORNE
                  CONMSG      This for an Osborne - Why ?
            ELSE
                  IF    IBM
                        CONMSG      This is for an IBM - What's that ?
                  ELSE
                        CONMSG    Improper Computer Definition !
                  ENDIF
            ENDIF
      ENDIF
```

Note that all of that code will only generate one byte of code - in fact none at all if the CONMSG portion is reached. Note also that the indents are not necessary - indeed they seem to be uncommon in assembly programming.

END

This is used to mark the end of the source file, but ZASM is not very picky - if you leave it out it will be assumed. Note that only one source module is permitted in a .Z80 source file. You cannot stack a number of modules and separate them with END's. The syntax is:

```
      END         or

      END   ZORK
```

where ZORK is a label at which you want execution to start. The usage of this option in CP/M is unclear to me since the CCP always transfers control to the start of the TPA when a program is run. In other systems - e.g. RT11 in PDP machines - the start address of a program is not necessarily its load address.

ENDM / MEND

These two statements are identical in function. They are used to end the scope of a MACRO, OMACRO, REPT, IRP, IRPC or STRUCT block of code. The syntax is simply:

```
      ENDM
```

ENTRY

This is identical to the PUBLIC statement in RMAC. It is used to inform the linker which labels in the module are to be made accessible to other

modules. See also the GLOBAL and EXTERNAL statements. The syntax is:

```
      ENTRY LABEL1,LABEL2,...
```

EQU

This is like the DL statement except that once a label is equated to a value it can not be changed to another value with another EQU or DL. The EQU statement should be used for values that are truly constant in the module. The syntax illustrated below:

```
      BDOS  EQU   5                 or

      BUFLEN      EQU   BUFEND - BUFBEG         or

      BUFLEN      EQU   $ - BUFHED
```

Note that if the labels BUFEND or BUFBEG are defined below the EQU statement an 'Error in Pass 1' message will occur. It will disappear in pass 2.

EXITM

This is used to exit from a macro definition before the closing ENDM (which must still exist). This can be used to simplify logical structures and perhaps speed processing. For example:

```
      SPCLCH      MACRO #CHAR
            IRPC  #TEST,'._$?@'
            IF    '#TEST' = '#CHAR'
            CONMSG      Special character detected.
            ENDIF
            EXITM
            ENDM
```

EXT / EXTRN

These synonyms are used to declare labels that are defined external to the current module. They are used to tell the linker which labels must be found elsewhere and to ensure the assembler that a value will be provided for the label at link time. This is the opposite of the ENTRY statement. For each label shared between modules there should be one ENTRY statement, in the module defining it, and an EXTRN statement in each module referencing it. The syntax is:

```
      EXTRN LBL1,LBL2,...
```

See also the ENTRY and GLOBAL statements.

FORM

Same as EJECT.

GLOBAL

The GLOBAL statement is provided for sloppy programmers. It can replace either the ENTRY or EXTRN declaration. If a label is declared GLOBAL and the assembler detects it's definition in the current module, it is assumed to be an ENTRY (public) label. If the assembler does not find it defined in the current module, it assumes the label to be of EXTRN type. Using GLOBAL thus short-circuits any chance the assembler might have had of detecting a typo. Furthermore it makes it very hard for the programmer to find where a shared label is defined: he has to scan the code of every module in which the label is declared GLOBAL. If the

ENTRY and EXTRN declarations are used instead, it is only necessary to scan the ENTRY statements to see which module contains the definition.

IRP

This is a special predefined macro and hence takes up one nesting level and requires an ENDM statement. The IRP stands for 'indefinite repeat'. The usage is illustrated below as is the ability to compare strings:

```
POPEM MACRO #R1,#R2,#R3,#R4,#R5,#R6
      IRP   #REG,#R1,#R2,#R3,#R4,#R5,#R6
      IF    "#REG" NE ""
      PUSH  #REG
      ELSE
      EXITM
      ENDIF
      ENDM
      ENDM
```

An invocation POPEM HL,BC,IX will expand to

```
      POP   HL
      POP   BC
      POP   IX
```

You may prefer the one liner, but remember to pop in reverse order to pushing. A second example follows:

```
      IRP   #ADDR,BUF1,BUF2,BUF3,BUF4
#ADDRP:     DW    #ADDR
      DS    128
      ENDM
```

This will be expanded into:

```
BUF1P:      DW    BUF1
BUF1: DS    128
BUF2P:      DW    BUF2
BUF2: DS    128
BUF3P:      DW    BUF3
BUF3: DS    128
BUF4P:      DW    BUF4
BUF4: DS    128
```

The #ADDR above is a dummy parameter which will take on the macro equivalents BUF1, BUF2, etc once each.

This example also illustrates a very important consideration with respect to macro parameter matching. The assembler matched the first characters of #ADDRP with the parameter #ADDR and assumed that the rest must be a literal character - i.e. one to be left as is.

IRPC

This is very similar to IRP except that the dummy parameter takes on only on character on each repeat. The syntax is illustrated below in which the example shown for IRP above is done by an IRPC instead:

```
      IRPC  #N,'1234'
BUF#NP:     DW    BUF#N
BUF#N:      DS    128
      ENDM
```

The characters in the substitution string need not be digits.

JSYS

This seems a bit strange. I suspect it is a hold over from ZASM used with some other operating system. It seems as if the syntax:

```
      JSYS  BYTVAL
```

generates the machine instructions:

```
      RST   1
      DB    BYTVAL
```

This would be meaningful if location 0008H contained a jump to some code which would retrieve the BYTVAL, do something with it, and then return control to the address following BYTVAL (or maybe perform an error exit).

LIST

This is used to switch listing options on and off at assembly time. The listing options have been given above. Note that listing options specified in the command line override those specified in the source code. Hence you cannot 'hide' a piece of code - that is, prevent it from showing up in a listing.

MACRO

The MACRO keyword is used to specify a macro - surprise. Formal parameters in ZASM begin with a # symbol rather than the ? symbol used in RMAC. Note that in ZASM, the query is a legal identifier character instead. This is not a treatise on macros but a few examples below will demonstrate some simple uses. The macro definition is given only once, and may even be hidden away in a file called a macro library. It can then be invoked as often as desired with different actual parameters. Note that the 'values' of the parameters are the actual character strings themselves. Note also that macros are assembly time phenomena - parameters can not depend on values generated at run time.

```
;     This macro moves a 16-bit value from one address to
;     another without (seemingly) any effect on registers

      MOVE  MACRO #SOURCE,#DESTN
      PUSH  HL
      LD    HL,(#SOURCE)
      LD    (#DESTN),HL
      POP   HL
      ENDM
```

If invoked as MOVE SAM,GEORGE it would expand to:

```
            PUSH  HL
            LD    HL,(SAM)
            LD    (GEORGE),HL
            POP   HL
```

Another example of a form frequently used is shown below. It is used to invoke CP/M functions in a neat form. As written, the first parameter is the CP/M function number, the second the contents of the DE register pair if required. This example also demonstrates the use of the special symbol #SYM in macros. #SYM expands to the number of parameters actually specified in the invocation.

```
      CPM   MACRO #FN,#PARAM
            PUSH  BC
            PUSH  DE
            LD    C,#FN
            IF    #SYM > 1
                  LD    DE,#FN
            ENDIF
            CALL  5
            POP   DE
            POP   BC
            ENDM
```

Then parts of the program code could be written as:

```
      CONIN EQU   1
      CONOUT      EQU   2
      SETDMA      EQU   26

            ...

            CPM   CONOUT, '?' ;display a ? on screen
            CPM   CONIN       ;get a keyboard character
            ...

            CPM   SETDMA,DISKBUF    ;set dma address to buffer
```

Of course, macros may not always be the most code-efficient way of doing things. They get expanded fully at each invocation as opposed to a subroutine which exists only at one space. If a macro expansion is long or referenced frequently, consider a subroutine. On the other hand, for short macros the subroutine linkage (call/ret) may obviate any savings.

NAME

This statement tells the librarian that the module name to be inserted into the .REL file is not the default one - that is, it is not the name of the .Z80 file which was assembled. The syntax is:

```
      NAME  NEWNAME
```

OMACRO

This is a puzzle. It seems to behave just the same as the MACRO keyword with the exception that the macro name no longer shows up in a symbol cross reference listing - it does now show up in an opcode cross reference if such is requested. Nevertheless there is still one piece of code that has not been fully explored and may show some other difference between these two. Note that the same symbol may NOT be declared both as a MACRO and an OMACRO. Also note that defining a built-in opcode (e.g. LD) as either a MACRO or an OMACRO will replace the built-in definition.

ORG

This is used to force the assembler and linker to place program or data at specific memory locations. Most .COM files begin at 100H but that is assumed anyways if no ORG statement is given. The syntax is:

```
      ORG   value
```

REL

This signals that the following is to be placed into the relocatable code program section. The syntax is simply:

```
      REL
```

REM

This seems to be the equivalent of a line beginning with a semicolon - i.e. pure comment. Why it is included escapes me, unless some unweaned BASIC programmer insisted on it.

REPT

The repeat statement is a predefined macro, taking one nest level and requiring an ENDM. It repeats a block of statements a fixed number of times. For example:

```
      ROTLEFT     MACRO #REG,#TIMES
            LD    A,#REG
            REPT  #TIMES
            RLCA
            ENDM
            LD    #REG,A
            ENDM
```

Then the instruction:

```
            ROTLEFT     B,5
```

would expand into:

```
            LD    A,B
            RLCA
            RLCA
            RLCA
            RLCA
            RLCA
            LD    B,A
```

which is much more readable in the macro/repeat form above.

STRUCT

This is another rather strange thing but perhaps a rather nice one to. The structure block which this introduces generates no code whatsoever. Indeed any instruction that would generate code is disallowed. Hence even DB, DW and DM instructions are illegal. The purpose seems to be to define the structure of a block of code or data more neatly than in using ordinary artifices. Some tentative examples of usage follow:

```
      STRUCT      0
DRIVE:      DS    1
NAME: DS    8
TYPE: DS    3
EXTENT:     DS    1
S1:   DS    1
S2:   DS    1
RECCT:      DS    1
BLOKS:      DS    16
CURREC:     DS    1
RANREC:     DS    2
TOOBIG:     DS    1
      ENDM
```

Then one can write, assuming that FCB is the address of a file control block:

```
      LD    A,(FCB+CURREC)
```

Of course we could have said just as well:

```
DRIVE EQU   0
NAME  EQU   DRIVE + 1
TYPE  EQU   NAME  + 8
...
```

but the STRUCT is clearer.

SUBTTL / TITLE2

This is the second title that will show up on pages of the listing. The syntax is:

```
      SUBTTL      This is a subtitle.
```

TITLE

This is the main title that will show up on each page of the listing. The syntax is as for SUBTTL.

Library Commands

There are four more special commands allowed, each beginning with an asterisk:

```
      *INCLUDE    filenam
      *INCLUDE1   filenam
      *MACLIB           filenam
      *RELLIB           filenam
```

The asterisk must lie in column 1 of the source - no tabbing is permitted. There are no compulsory or default extensions for the filenames. The meaning of the commands is:

*INCLUDE

Include the file named into the assembly at this point. This file might typically contain constant definitions, entry and external declarations, some documentation detail, or simply more program code. The include file will be read on both assembler passes. INCLUDE's can be nested to four levels.

*INCLUDE1

The same as INCLUDE except that it happens only on pass 1. Since the listing is not generated until pass 2, INCLUDE1 should not be used for anything to be listed. Also code is not generated until pass 2 so INCLUDE1 can not be used for source statements that generate code. Perhaps constant definitions, entry and external declarations, or macro definitions can be included only in pass 1.

*MACLIB

This declares the named file to contain a number of macro definitions. I'm not sure just what happens but I think that the macro names only are loaded into memory together with their location on disk in case they are needed later.

*RELLIB

This causes a special record to be sent to the .REL file instructing the linker to automatically search the named file to resolve external references. This can save typing in the name of a standard library every time the link command is typed. Any other advantage escapes me.