



十速科技股份有限公司
tenx technology inc.

**Advance
Information**

TM89 series

cross assembler

User Manual

**Tenx reserves the right to change or
discontinue this product without notice.**

tenx technology inc.

INDEX

I. Features:	3
II. Limits:	3
1. Name:	3
2. Case-sensitive:	3
3. Symbols and Labels:	3
III. Numeral Systems:	4
1. Expressions:	4
2. Four arithmetic operations:	4
IV. Types of Pseudo Instructions:	7
1. Instructions of Region Assignment:	7
(1) .CODE:	7
(2) .RODATA:	7
(3) .RAM & .ENDRAM:	7
(4) .END:	7
2. Commands of Chip Types:	7
(1) .CPU:	7
3. Other Instructions:	8
(1) .ADDR:	8
(2) .AUTOIMPORT:	8
(3) .BYT & .BYTE :	8
(4) .CASE:	9
(5) .DB:	9
(6) .DBYT:	9
(7) .DEFINE:	9
(8) .DEF & .DEFINED:	10
(9) .DN:	10
(10) .DWORD:	11
(11) .ELSE:	11
(12) .ELSEIF:	11
(13) .ENDIF:	11
(14) .ENDMAC & .ENDMACRO:	11
(15) .ENDPROC:	11
(16) .EQU:	11
(17) .ERROR:	12
(18) .EXITMAC & .EXITMACRO:	12
(19) .EXPORT:	12
(20) .GLOBAL:	13
(21) .IF:	13
(22) .IFBLANK:	13
(23) .IFDEF:	14
(24) .IFNDEF:	14
(25) .IMPORT:	14
(26) .INCLUDE:	15

(27)	.LOCAL:.....	15
(28)	.MAC & .MACRO:.....	15
(29)	._main:.....	16
(30)	.ORG:.....	16
(31)	.PARAMCOUNT:.....	17
(32)	.PROC:.....	18
(33)	.RELOC :	18
(34)	.SEGMENT:.....	19
(35)	.WORD:.....	20
V.	Error Messages:.....	20

I. Features:

1. OPRAND can be defined as a constant and used in four arithmetic operations.
2. It can use Marco function. A Code segment which is frequently used can be transformed into Macro for invocation.
3. It can compiler and link several source file. (Please see handbook of IDE)
4. It can set included library path of each project.(Please see handbook of IDE)

II. Limits:

1. Name:

Variables, constants and string labels only can use the following text symbols:

0 ~ 9, a ~ z, A ~ Z, “_”, and cannot begin with numeric character (0 ~ 9).

The length of string is unlimited.

2. Case-sensitive:

User can setting Case-sensitive or not while compiling LABEL and Macro name. (The default is Case-sensitive. Please see [.CASE](#))

3. Symbols and Labels:

(1). Numeric constants:

Numeric constants are defined using the equal (=) sign, for instance,
two = 2

the symbol "two" can be used everywhere in the program where it is evaluated to the value 2.

<example > : four = two * two

(2). Standard labels:

The usage of a label is to define the label name in front of each line, (before any instruction mnemonic, macro or pseudo directive), followed by a colon.

(3). Local labels and symbols:

[.PROC](#) command can be used to create a code segment where the local labels and symbols are declared in the segment. Those labels and symbols are not recognized outside this segment and cannot be accessed.

(4). Use macros to define labels and constants:

There are some drawbacks in this approach; however, it is handy in another situation. [.DEFINE](#) instruction is used to define symbols or constants for that may be used elsewhere. Since the macro function can be executed without

limitations in a low-end operation. String constants can be defined in this form. (the other symbol types are not fit).

<Example>:

```
.DEFINE two    2
.DEFINE version "SOS V2.3"

four = two * two           ; Ok
.byte version             ; Ok

.PROC                    ; Initiates local scope
two = 3                   ; "two = 3" two is global constant
.ENDPROC
```

III. Numeral Systems:

1. Expressions:

- (1). Binary: Use 'B' as an identifier with Case-insensitive, or use '%' in front of the value.

<Example-1>: 1000B, 10000100B, 1011b

<Example-2>: %1000, %10000100

- (2). Decimal: No need to add Identifier.

<Example>: 20

- (3). Hexadecimal:

(3-1). Use 'H' as identifier with Case-insensitive.

<Example>: 8H, 0FFH, 0fh

(3-2). Use '\$' in front of the value.

<Example>: \$9A, \$FD (Please use the '\$' before the number to avoid confusion in the definition of .EQU and .DN.)

(3-3). Use '0x' in front of the value and Case-insensitive.

<Example>: 0x8, 0X0FF

- (4). Constant or Address: Use ".EQU" as an identifier and Case-insensitive, or use '=' after name.

<Example-1>: addr .EQU 4

<Example-2>: addr = 4

2. Four arithmetic operations:

In source codes, constants can be performed with four arithmetic operations, which follow the operator precedence (e.g., multiplication and division are evaluated first, and then addition and subtraction are evaluated second). The followings are operators:

" + ": addition operator

" - ": subtraction operator

" * ": multiplication operator

" / ": division operator

" % ", ".MOD ": modulo operator

<Example>
 addr .equ 4
 lda addr%3 ; addr =1
 lda addr .mod 4 ; addr =0

"<<": shift left operator

<Example>
 addr .equ 2
 lda addr<<2 ; addr=8

">>": shift right operator

<Example>
 addr .equ 8
 lda addr>>2 ; addr=2

" | ", ".BITOR": Binary OR operator

<Example>
 addr .equ 2
 lda addr | 4 ; addr=6

" & ", ".BITAND": Binary AND operator

<Example>
 addr .equ 6
 lda addr & 4 ; addr=4

" ~ ", ".BITNOT": Binary NOT operator

<Example>
 addr .equ 11110000B
 lda ~addr ; addr=00001111B

" ^ ", ".BITXOR": Binary XOR operator

<Example>
 addr .equ \$F0
 lda addr ^ \$FF ; addr=\$0F

" (" , ") ": Parentheses operator

<Example> 2 nested layers of parentheses computation

HIGHT .EQU ((10+5)*2)/3

- " = ": Compare operator (equal)
- " <> ": Compare operator (not equal)
- " < ": Compare operator (less than)
- " > ": Compare operator (greater than)
- " <= ": Compare operator (less than or equal to)
- " >= ": Compare operator (greater than or equal to)
- " && ", ".AND": Boolean AND operator
<Example>
addr. equ 6
lda addr && 4 ; lda 1
- " || ", ".OR": Boolean OR operator
<Example>
addr. equ 1
.define zpaddr 1
lda (!addr) && zpaddr ; lda 0
- ".XOR": Boolean XOR operator
<Example>
addr. equ 6
lda addr .XOR 4 ; lda 0
- " ! ", ".NOT": Boolean NOT operator
<Example>
addr. equ 1
lda (!addr) && 1 ; lda 0

IV. Types of Pseudo Instructions:

1. Instructions of Region Assignment:

The segments described below do not have a certain precedence order, but each segment has to be separated in the integrity by its KEY WORD. The non-used segments do not have to be defined.

(1) .CODE:

Switch to the code segment with Case-insensitive. The source code of the program is defined in this segment. This is a abbreviation of 「.segment "CODE"」.

(2) .RODATA:

Switch to the Table ROM segment with Case-insensitive. Labels can be used while define the content of Table ROM.

(3) .RAM & .ENDRAM:

In RAM segment, the two instructions must be declared in a pair with Case-insensitive, in where variable or constant of data RAM address are declared. The variable DN of data RAM address can only be defined in this segment.

(4) .END:

It can only be defined once. Assembly program will be terminated when execute this instruction.

2. Commands of Chip Types:

(1) .CPU:

Case-insensitive. It can only be defined once and placed in any location outside the segment. The declaration is as follows:

```
.CPU chip_species
chip_species: Chip number
```

<Example>

```
.CPU TM8959 <---- 8959 chip
.CODE
.....
.....
```

```

ADD    10H
ADD    21H
.....
.....
.END

```

3. Other Instructions:

(1) .ADDR:

Define two bytes of data. This is an alias for [.WORD](#) for easy readability, especially when the data is a value of address. This instruction must be followed by a sequence of expressions (The data content can be constants or Identifier).

<Example>

```
.addr $0D00, $AF13, _Clear
```

(2) .AUTOIMPORT:

This can be followed by a plus (+) or a minus (-) character. When enable this function (using '+'), those undefined symbols will be marked automatically with import instead of errors. When switched off (default), those symbols will show up error message because those symbols are not be judged by the assembly program.

However, when this function is enabled, the assembly source code will not generate error messages for those undefined symbols until Link program has completed.

<Example>

```
.autoimport + ; Switch on auto import
```

Or

```
.autoimport on ; Switch on auto import
```

(3) .BYT & .BYTE :

Define one byte of data. This instruction must be followed by a sequence of (byte ranged) expressions or strings.

<Example>

```
.byte 'l', 'i', 'n', 'k'
```

```
.byt 'f', 'i', 'l', 'e', $0D, $00
```

(4) .CASE:

Switch on or off case sensitivity on identifiers. Default is off (which means, identifiers are Case insensitive). The instruction must be followed by a '+' or '-' character to switch the option on or off.

<Example>

```
.case - ; Identifiers are Case insensitive
```

Or

```
.case on ; Identifiers are Case sensitive
```

(5) .DB:

Data bytes definition.

<Example>

```
.RODATA
```

```
.db '2','3','4', '5' ; this will generate the bytes $32 $33 $34 $35
```

```
.org 10h
```

```
.byte $12, $34, $56, $78, $9a ; Setting data from address 10h
```

```
.org 20h
```

```
.db $11, $22, $33, $44 ; Setting data from address 20h
```

```
.db 'A', 'B', 'C', 'D'
```

(6) .DBYT:

Define two bytes data with the “hi” and “lo” bytes swapped (use [.WORD](#) to create word data in TM89xx format). This instruction must be followed by a sequence of (word ranged) expressions.

<Example>

```
.dbyt $1234, $4512
```

Then these bytes will be written into the current segment in the following order:

```
$12 $34 $45 $12
```

(7) .DEFINE:

This instruction must be followed by an identifier (Macro name) and a sequence of arguments inside of the parenthesis. See [.MACRO](#).

(8) .DEF & .DEFINED:

This instruction must be followed by an argument within the parentheses. And the argument will be checked to see if it has been defined somewhere else before the current code reference. If it has been defined, then the function value "true" will be returned, otherwise the function value "false" will be returned. For example, [.IFDEF](#) statement can be replaced by `.if .defined(a)`

(9) .DN:

Defines a variable to replace the address of data RAM as well as defines the numbers of nibbles that occupy in the data RAM. In program, this variable can replace the data RAM address. Such an instruction only can be used in RAM segment and constant segment. The declaration is as follows:

Variable .DN Nibble

Variable: Variable name

Nibble: It defines the numbers of nibbles where a variable occupy in the data RAM, those numbers cannot exceed the maximum of data RAM address. This instrument must be used with ORG instrument in order to define the initial address of data RAM.

<Example>

```
.RAM
    .ORG 40H          ; Define the initial address of variable in data RAM.
    DISPLAY .DN 1    ; Declare the address 40H of data RAM as
                    ; DISPLAY.
    SPEED .DN 2      ; Declare the address 41H and 42H of data RAM as
                    ; SPEED.
                    ; SPEED replaces address 41H in data RAM.
                    ; SPEED+1 replaces address 42H in data RAM.
    CHAR .DN 1       ; Declare the address 43H of data RAM as CHAR.
    ORG 70H
    LCD1 .DN 1       ; Declare the address 70H of data RAM as LCD1.
.ENDRAM
.CODE
```

```

.....
      ADC SPEED      ; SPEED replaces address 41H in data RAM ADC
                      SPEED+1
                      ; SPEED+1 replaces address 42H in data RAM.
.....
.END

```

(10) .DWORD:

Define 4 bytes data type, this instruction must be followed by a sequence of expressions.

<Example>

```
.dword $12344512, $12FA489
```

(11) .ELSE:

Conditional Expressions instruction.

(12) .ELSEIF:

Conditional Expressions instruction. Check another conditional statement.

(13) .ENDIF:

Conditional Expressions instruction. Terminate an [.IF](#) or [.ELSE](#) branch.

(14) .ENDMAC & .ENDMACRO:

Terminate macro definition (see section [.MACRO](#)).

(15) .ENDPROC:

Terminate local program block (see [.PROC](#)).

(16) .EQU:

Define a constant with Case-sensitive. This instruction can not be defined in RAM segment. The declarations are as follows:

```
Constant .EQU data
```

Constant: constant name

data: Content value of the constant

<Example 1>

```
VALUE1 .EQU 10H
```

<Example 2>

```
.RODATA
    AH .EQU 0H
    BH .EQU 0H

.CODE
    ADD AH ; AH is equal to 0H.
    ADC BH ; BH is equal to 0H.

.END
```

(17) **.ERROR:**

The interpreter will output a warning through “User-Defined” error message, therefore, there are no object files will be generated. This instruction is used to check the initial conditions for interpreting the source files.

<Example 1.>

```
.if          foo = 1
.....

.elseif     bar = 1
.....

.else

.error      "Must define foo or bar!"

.endif
```

<Example 2.>

```
.if DEBUG=1
    .error "No support in Debug mode"
.endif
```

(18) **.EXITMAC & EXITMACRO:**

Skip macro immediately. This command is used in recursive macros frequently. See [.MACRO](#).

(19) **.EXPORT:**

Mapping the declared symbols to the other source code (*.asm ,*.c). This instruction must be separated by a comma. (see [.AUTOIMPORT](#)).

<Example>

```
.export msg, start
.case -
.RODATA ; For table Rom use
.word $1234
.db '2','3','4'
.word 't','7'
.dword 12345678h,8765432h
msg:
.addr $0D00, $AF13;, _Clear
.code
Start:
    szrx
    setdat $00
    sta 12
    sta 12 .mod 11
```

(20) .GLOBAL:

Declare symbols as global symbols. This instruction must be separated by a comma. The symbols from the list, that are defined somewhere in the source code, are exported. When using global symbols, user has to use [.IMPORT](#) instruction to import. In addition, [.IMPORT](#) or [.EXPORT](#) instructions for using the same symbol are allowed.

<Example>

```
.global foo, bar
```

(21) .IF:

Evaluate an expression value to determine whether interpreting and output. This expression must be a constant expression, which means, all operands must be defined as constants. The expression value of zero is evaluated to FALSE, any other value is evaluated to TRUE.

(22) .IFBLANK:

Conditional Expressions instruction: Check if an argument is stringed in this line. If it is evaluated to FALSE, then its following statement code won't be executed until it fetches [.ELSE](#) or [.ELSEIF](#) or [.ENDIF](#) . This instruction is often used to check if a macro argument is stringed

in. If there is no macro argument is stringed in, it will be evaluated to TRUE, and so, contrariwise.

<Example>

```
.macro ADD2 v1,v2,sum
  lda v1
  .ifblank v2
    add sum      ; if v2 has no argument stringed in.
  .else
    add v2       ; if v2 has an argument stringed in.
  .endif
  sta sum
.endmacro

.code
lda 1
ADD2 1, , sum   ; If there is no argument stringed in, use “,” to
                ; replace it.

sta 2
.endcode
```

(23) **.IFDEF:**

Conditional Expressions instruction: Check if a symbol is defined. And it must be followed by a symbol name. The condition TRUE means the symbol is already define, and false otherwise.

(24) **.IFNDEF:**

Conditional Expressions instruction: Check if a symbol is defined. And it must be followed by a symbol name. The condition TRUE means the symbol is not define, and false otherwise.

(25) **.IMPORT:**

Import a symbol from another module. The command is followed by a sequence of symbols separated by commas.

<Example>

```
.import foo, bar
```

(26) .INCLUDE:

Include another file. Include files may be nested up to a depth of 16.

<Example>

```
.include    "subs.inc"
```

(27) .LOCAL:

The Label names declared as local are used in Macro only. It implicates to avoid "duplicate symbol" problem to the macro expansion. Using local labels outside the Macro will generate error messages when interpreting the source codes.

<Example>

```
.macro ADD1 v1,v2,sum
.local L1    ; L1 is defined as a label ADD1 in the macro, it
              ; cannot be referenced outside the macro.

        lda v1
        add v2
        jmp L1
        lda v2
L1:     sta sum
.endmacro
```

(28) .MAC & .MACRO:

Initiate a macro definition. The command is followed by an identifier (the macro name) and the macro arguments can be separated by commas.

<Example>

```
.CODE
.macro foo  arg1, arg2, arg3
    .if arg3 >0
        .define sum 123
    .endif
```

```
.if    .paramcount < 3          ; if the return value of Macro parameter is less than 3?
```

```
.error "Too few parameters for macro foo" ; Output user defined error message.
.endif
```

```
.if .paramcount > 3 ; if the return value of Macro parameter is greater than 3 ?
.error "Too many parameters for macro foo" ; Output user defined error message.
.exitmacro
.endif
```

```
lds 0x10,arg1
lda 0x10
lds 0x11,arg2
.ifdef sum ; Addition
adc 0x11
.exitmacro
.endif
sbc 0x11 ; Subtraction
.endmacro
```

```
start :
    foo 5,9,
    foo 5,9,1
    jmp start
.END
```

(29) **._main:**

Define the entry point of the of *.ASM files, just like void main() used in Link program for settings of the PC value.

It is recommended to use when a project includes one above *.ASM files or a project with mixed files types.

(30) **.ORG:**

Define the initial address for the follow-up program or for the variable declarations. This command can be used in a same segment many times with case-insensitive.

in Data segment (.RODATA), this command is not fit for the mixed type of project with *.C & *.ASM files. It is suggested to use only in the project with *.ASM files. RELOC command can be used together to avoid overlap condition.

The variable declaration is as follows:

```
.ORG Setting_addr
```

Setting_addr: the addresses of program or data RAM or Table ROM.

If `.ORG` command is used in Program segment, the valid address for the next valid source code can be assigned directly in the program.

However, label name cannot be followed by `.ORG` in a same line.

<Example>

```
.code
.....
        jb1 30H
        jb2 40H
        jb3 50H
        .org 30H
        lda 1H      ; PC address is 30H.
.....
        .org 40H    ; PC address is 40H.
        lda 2H
.....
        .org 50H    ; PC address is 50H.
        lda 3H
.....
.end
```

直接定義(下一個“DN”指令所定義的)data RAM 變數 所代表的 data RAM 位址

If `.ORG` command is used in RAM segment, the next address of the data RAM variable defined by the `.DN` can be defined directly in the program.

(31) **.PARAMCOUNT:**

Evaluate the number of arguments that are passed to a Macro.

<Example>

```
.macro foo arg1, arg2, arg3
.if .paramcount <> 3
```

```
.error "Too few parameters for macro foo"
.endif
.....
.endmacro
```

(32) .PROC:

Start a nested lexical level. All new symbols from now on are in the local lexical level and are not accessible from outside. Symbols defined outside this local level may be accessed as long as their names are not used for new symbols inside the level. Symbols names in other lexical levels do not clash, so you may use the same names for identifiers. The lexical level ends when the [.ENDPROC](#) command is read. Lexical levels may be nested up to a depth of 16. The command may be followed by an identifier, in this case the identifier is declared in the outer level as a label having the value of the program counter at the start of the lexical level. Note: Macro names are always in the global level and in a separate name space. There is no special reason for this, it's just that I've never had any need for local macro definitions.

<Example>

```
.proc Clear      ; Declare Clear as a subroutine, Initiate a new
                  declaration level.

lds HOUR1,5
Clear_all :      ; L1 is local and does not cause a
                  ; duplicate symbol error if used in other
                  ; places

dec* HOUR1
jac 0
setdat Return
jmp Clear_all
Return: rts
.endproc        ; Leave lexical level
```

(33) .RELOC :

To interrupt .ORG and reallocate a PC address with Link program.
(.ORG command can be used together.)

<Example>

`.RODATA` ; Declares the TABLE ROM segment.

```
.org 00h
.db 03fh,0f3h
.db 00fh,0f0h
.db 0cfh,0fch
.org 20h
.db 0f1h,00fh
.db 0f0h,00fh
.db 0f8h,08fh
```

`.CODE`

`.RELOC` ; To Interrupt `.ORG` in the code of `.RODATA` and reallocate a PC address with Link program.

```
LCD_clear:
SHLX
SETDAT $0080
LDS8# @HL,$00
LDS8# @HL,$00
...
```

(34) **.SEGMENT:**

Switch to another segment. `CODE` and `RODATA` will output to their own segment, that is, what is called a segment of data, a named data segment. The default segment is "CODE" segment. There are up to 254 different segments per object file (and up to 65534 segments per .executable file).

The `CODE` segment and `RODATA` segment are most used for declaration. This command must be followed by a segment name which has been defined by the user. (There are some constraints on the name – it is suggested to use those names that are corresponded with the valid variable rules).

<Example 1>

```
.segment "RODATA" ; Switch to RODATA segment
.segment "CODE" ; Switch to CODE segment
```

<範例 2>

```
.segment "RODATA" ; Switch to DATA segment
INT_Counter:
.db 'I','N','T',' ','C'
.db 'o','u','n','t','e'
.db 'r',00
```

```

.segment "CODE"           ; Switch to CODE segment
IntCounterMode:
    fast
    lds    Mode,DeadLoopMode
    call  ClearLcd
    lds    Row_Pixel,00
    lds    Column_Pixel+0,00
    lds    Column_Pixel+1,00
    lds    StringTableAddress+0,INT_Counter & 0fh
    lds    StringTableAddress+1,INT_Counter>>4&0fh
    lds    StringTableAddress+2,INT_Counter>>8
    lds    StringTableAddress+3,INT_Counter>>12
    call  DisplayString
        :
        :

```

(35) .WORD:

Define two bytes word data. This instruction must be followed by a sequence of (word ranged, but not necessarily constant) expressions.

<Example>

```
.word $0D00, $AF13
```

V. Error Messages:

1. 1."Command/operation not implemented"
2. "Cannot open include file"
Please check whether the file exists.
3. "Cannot read from include file"
Please check whether the file exists.
4. "Include nesting too deep"
The depth of included files can not exceed 16 layers.
5. "Invalid input character: "

6. "Hex digit expected"

Please see hexadecimal expressions of [Value Systems](#).

<Example>:

MRW %01, \$%10 correct is: MRW %01, \$10

7. "Digit expected"

Please see decimal expressions of [Numeral Systems](#).

8. "'0' or `1' expected"

Please see binary expressions of [Numeral Systems](#).

9. "Numeric overflow"

The Integer is too large, it must be less than or equal to \$FFFFFFFF.

10. "Control statement expected"

<Example>:

.INCLUDE a.asm Correct: .INCLUDE "a.asm"

11. "Too many characters"**12. "':' expected"**

Missing a colon in label definition.

13. "'(' expected"

Missing a right parenthesis in arithmetical operation.

14. "')' expected"

Missing a left parenthesis in arithmetical operation.

15. "*Reserve***"****16. "',' expected"**

<Example>:

MRW div/value,0fh,12 Correct: MRW div/value,0fh

17. "Boolean switch value expected (on/off/+/-)"

32. "Illegal character constant"

<Example>:

.BYTE 'c1','2'

Correct : .BYTE 'c','2'

33. "Illegal addressing mode"

34. "Illegal character to start local symbols"

35. "Illegal use of local symbol"

36. "Illegal segment name"

37. "Illegal segment attribute"

38. "Illegal macro package name"

39. "Illegal emulation feature"

40. "Illegal scope specify"

41. "Syntax error"

<Example>:

LDS value#-\$1, \$1

Correct : LDS value -\$1, \$1

42. "Symbol is already defined"

Symbol is redefined.

43. "Undefined symbol"

Please use ".AUTOIMPORT ON" if the symbol name is defined in another source file.

44. "Symbol is already marked as import"

45. "Symbol is already marked as export"

46. "Exported symbol is undefined"

The name of symbol is not defined.

47. ***Reserve***

48. "Unexpected end of file"

49. "Unexpected end of line"

<Example>:

.BYTE 'c','2',

Correct : .BYTE 'c','2'

50. ***Reserve***

51. "Division by zero"

<Example>:

.RODATA

div = 0

.CODE

MRW value/div, 1

Correct: div = 10

52. "Modulo operation with zero"

<Example>:

MRW 08%,\$10

Correct: MRW 08%value,\$10

53. "Range error"

The size of Integer or constant exceeds the definition of its size of data type.

<Example>:

.BYTE \$1234

Correct : .BYTE \$12

54. "Too many macro parameters"

Too many parameters are passed to macro.

55. "Macro parameter expected"

56. "Circular reference in symbol definition"

57. "Symbol re-declaration mismatch"

- 58. "Alignment value must be a power of 2"
- 59. "Duplicate `.ELSE`"
The `.ELSE` is reduplicated used prior to `.ENDIF`.
- 60. "Conditional assembly branch was never closed"
Missing a `.ENDIF` to end the `.IF` or `.ELSE` branch.
- 61. "Lexical level was not terminated correctly"
- 62. "No open lexical level"
- 63. "Segment attribute mismatch"
- 64. "Segment stack overflow"
- 65. "Segment stack is empty"
- 66. "Segment stack is not empty at end of assembly"
- 67. `***Reserve***`
- 68. "Counter underflow"
- 69. `***Reserve***`
- 70. `***Reserve***`
- 71. "File name ``%s'` not found in file table"
- 72. `"'.DN'` must define in `' .RAM'` segment"
The `.DN` must be defined in the RAM segment.
- 73. `"'.ENDRAM'` expected"
`.RAM` and `.ENDRAM` instructions must exist in a pair.

74. ".DN' expected"

The .DN must be used in RAM segment.

Please see others command of [Types of pseudo Commands](#).

75. "Illegal data"**76.** "Operand error"

The number of instruction OPRAND do not match.

<Example>:

lda src<<1, 1 ;

Correct : lda src <<1

77. "Cannot open COE file"

Please check the COE file whether exist.

78. "***Reserve***"**79.** "Program ROM (XXXXH) out of range (YYYYH)"

The maximum address of program ROM is YYYYH.

XXXXH exceeds the range of YYYYH.

80. "Table ROM (XXXXH) out of range (YYYYH)"

The maximum address of table ROM is YYYYH.

XXXXH exceeds the range of YYYYH.