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:

1. OPRAND can be defined as constant and used for four arithmetic operations.
2. Marco function - frequently used codes can be compiled into Macro and can be called by other functions.
3. It provides compiler function and can be linked with multiple source codes. (Please refer to the handbook of IDE)
4. Each project can set its own library path.(Please refer to the handbook of IDE)

II. Limits:

1. Name:

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

0 ~ 9, a ~ z, A ~ Z, "_", and can not start with numerals of 0 ~ 9.

The length of string is unlimited.

2. Case-sensitive:

User can apply Case-sensitivity 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 by 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 function of the label is to define the label name b 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 macro to define label and constant:
There are some drawbacks in this approach; however, it is handy in another situation. .DEFINE instruction is used to define symbols or constants likely to 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. (other symbol types are not included).

<Example>:

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

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

.PROC                   ; Initiate the local scope of a code segment
two = 3                 ; "two = 3" two is a global constant
.ENDPROC
```

III. Numeral Systems:

1. Expressions:

(1). Binary: Use 'B' as an identifier without Case-sensitivity, or use '%' before 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 an identifier without Case-sensitivity.

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

(3-2). Use '$' before the value.

<Example>: $9A, $FD

(Please start with the symbol '$' to avoid the confusion with the definition of .EQU and .DN.)

(3-3). Please start with the '0x' before the value without Case-sensitivity.

<Example>: 0x8, 0X0FF

(4). Constant or Address: Use ".EQU" as an identifier without Case-sensitivity, or use '=' after the name.

<Example-1>: addr .EQU 4

<Example-2>: addr = 4

2. Four arithmetic operations:

In source codes, constant can be evaluated with four arithmetic operations, following the operator precedence (e.g., firstly, multiplication and division are evaluated; and then addition and subtraction are evaluated). The followings specify the 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 for Segment Assignment:

The code segments in a program are not in a certain precedence order, but each segment has to be separated in the integrity by its KEY WORD. The non-used segments need not to be defined.

(1) .CODE:

It will switch to the designated code segment without Case-sensitivity. The source code is defined in this segment. It is an abbreviation of 「.segment "CODE"」.

(2) .RODATA:

It will switch to the Table ROM segment without Case-sensitivity. Labels can be used while defining the content of Table ROM.

(3) .RAM & .ENDRAM:

In RAM segment, the two instructions must be declared in a pair without Case-sensitivity, in which 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 executing 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 double data bytes. It is an alias of .WORD for easier readability, especially when the data represents the value of a address. This instruction must be followed by a sequence of expressions (The data content can be constants or Identifiers).

<Example>

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

(2) .AUTOIMPORT:

This instruction can be followed by a plus (+) or a minus (-) character. When enabling this function (using '+'), the undefined symbols will be marked automatically with import instead of errors. When it switched off (default), those symbols will show up with error messages due to that those symbols are not be compiled by the assembly program.
However, if this function is enabled, then the assembly program will not generate error messages for the undefined symbols until Link program has been completed.

<Example>

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

(3) .BYT & .BYTE :

Define one data byte. 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:

It will switch the case-sensitivity for identifiers. Default is off mode (namely, identifiers are Case insensitive). This instruction must be followed by a '+' or '-' character to switch on or off the option.

<Example>

```
.case   -            ; Identifiers are Case insensitive
Or
.case   on           ; Identifiers are Case sensitive
```

(5) .DB:

Define the data bytes.

<Example>

```
.RODATA
.db '2','3','4', '5'            ; the data bytes will be generated as $32 $33 $34 $35
.org 10h
.byte $12, $34, $56, $78, $9a                ; Setup data starts at address 10h
.org 20h
.db $11, $22, $33, $44                       ; Setup data starts at address 20h
.db 'A', 'B', 'C', 'D'
```

(6) .DBYT:

Define the double data bytes where its higher and lower bytes will be swapped. (.WORD can be used to create word data in TM89xx format). This instruction must be followed by a sequence of (word ranged) expressions.

<Example>

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

The new generated bytes will be written into the current segment as 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 the parenthesis. Please see .MACRO.

(8) .DEF & .DEFINED:

This instruction must be followed by an argument within the parentheses. This argument will be checked if it has been defined somewhere before the reference to the current code. If it has been defined, the function value will return "true", otherwise the function value will return "false". The following is an example that .IFDEF statement can be replaced by .if .defined(a).

<Example>

```
if .defined(a)
```

(9) .DN:

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

```
Variable    .DN    Nibble
```

Variable: Variable name

Nibble: It defines the numbers where how many nibbles occupied by a variable in the data RAM, these numbers cannot exceed the maximum of data RAM address. This instrument must be used with ORG instrument in order to define the start address of data RAM.

<Example>

```
.RAM
    .ORG  40H          ; Define the start 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 data type to 4 bytes, this instruction must be followed by a sequence of expressions.

<Example>

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

(11) .ELSE:

Conditional instruction.

(12) .ELSEIF:

Conditional instruction. Check another conditional statement.

(13) .ENDIF:

Conditional instruction. Terminate an .IF or .ELSE branch.

(14) .ENDMAC & .ENDMACRO:

Terminate macro definition (please see section .MACRO).

(15) .ENDPROC:

Terminate the local block of the program (please see .PROC).

(16) .EQU:

Define a constant with Case-sensitivity. 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, 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 frequently used in recursive macros. Please see .MACRO.

(19) .EXPORT:

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

<Example>

```
.export msg, start
.case -
.RODATA                    ; Define the start area of Table Rom
.word $1234
.db '2','3','4'
.word 't','7'
.dword 12345678h,8765432h
msg:
.addr   $0D00, $AF13;
.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, 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, meaning all the operands must be defined as constants. The expression value of zero is evaluated to FALSE while any other value is evaluated to TRUE.

(22) .IFBLANK:

Conditional instruction: Test if some parameters in the macro are passed in. If the condition fails, the instructions that follow will not be compiled until .ELSE or .ELSEIF or .ENDIF are fetched. This instruction is frequently used to check if a parameter in the macro is passed in. If no parameters in the macro are passed in, the function value will return "True", and vice versa.

<Example>

```
.macro ADD2 v1,v2,sum
 lda v1
.ifblank   v2
 add sum        ; If no parameters are passed in  V2.
.else
 add    v2      ; if parameters are passed in V2.
.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, 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 defined, 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. Included 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 the prevention of the macro expansion from the "duplicate symbol" problem. 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 must be 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          ; Test whether the parameter in the Macro passed in is
                                   less than 3.
.error  "Too few parameters for macro foo"     ; Output the “user defined” error
                                                 message.
.endif

.if     .paramcount > 3          ; Test whether the parameter in the Macro passed in is
                                   greater than 3.
.error  "Too many parameters for macro foo"  ; Output the “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 as 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 the same segment many times without Case-sensitivity.

In the Data segment (.RODATA), this command does not fit for the hybrid 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:

Count the number of parameters passed in the Macro.

<Example>

```
.macro  foo     arg1, arg2, arg3
.if     .paramcount <> 3
.error  "Too few parameters for macro foo"
.endif
……
.endmacro
```

(32) .PROC:

The use of .PROC indicates the vocabulary declaration level. All the new symbols defined after the instruction will only exit within the local declaration block and can not be accessed from external. Symbols that are defined outside the block can be accessed if they are not redefined within the block. The naming of the symbol will not conflict with symbols within other declaration blocks. Therefore, it is fine to use the same name to declare a variable. When encountering .ENDPROC instruction, the function of the vocabulary declaration block will end. A maximum of 16 vocabulary declaration levels are allowed. The Instruction can be followed by a variable name which is a label defined outside the vocabulary and its value is just the value of the PC (program counter) at the start of this declaration level. Please note that the name of a macro must be declared in the TOP level and exit in another naming space.

<Example>

```
.proc Clear       ; Declare Clear as a subroutine and Initiate a
                    new declaration level.
lds HOUR1,5
Clear_all :       ; Clear_all is local. If it is used in other place, it
                    will not generate error for the same symbol.
```

```
dec* HOUR1
jac 0
setdat Return
jmp Clear_all
Return: rts
.endproc          ; The function of  vocabulary block ends.
```

(33) .RELOC：

To interrupt .ORG and reallocate the 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       ; This instruction is used to Interrupt .ORG and reallocates
                   the PC address with Link program.

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

(34) .SEGMENT:

It will switch to another segment. The CODE and RODATA will output into their own segment, which 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 mostly 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 corresponding 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 data type to double bytes. 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 if the file exists.

4. "Include nesting too deep"
The maximum depth of the included files cannot 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/+/-)"

18. ***Reserve***

19. ***Reserve***

20. "Integer constant expected"

21. "String constant expected"
<Example>:
.include a.asm Correct : .include "a.asm"

22. "Character constant expected"

23. "Constant expression expected"
The constant name is not defined.

24. "Identifier expected"
Variable, constant and string label can only be used as the following text symbols: '0' ~ '9', 'a' ~ 'z', 'A' ~ 'Z','_' , . The above identifiers cannot be initiated as numeric 0 ~ 9.

25. "`.ENDMACRO' expected"
“ENDMACRO" instruction must be used along with ".MACRO".

26. "Option key expected"

27. "`=' expected"

28. ***Reserve***

29. "User error:"
Error messages defined by User-defined.

30. 30."String constant too long"
The maximum size of string constant is 255.

31. "Newline in string constant"
The string constant must be in the same line.

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 symbol name 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 reduplicate prior to ".ENDIF".

60. "Conditional assembly branch was never closed"
".ENDIF" is missing to end the test branch for condition ".IF" or condition ".ELSE".

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 other commands of Types of pseudo Commands.

75. "Illegal data"

76. "Oprand error"
The numbers of instruction OPRAND do not match.
<Example>:
lda src<<1, 1 ; Correct : lda src <<1

77. "Cannot open COE file"
Please check if the COE file exists.

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.