TICE99IDE

Cross Assembler

User Manual

Rev V1.3

tenx reserves the right to change or discontinue the manual and online documentation to this product herein to improve reliability, function or design without further notice. **Tenx** does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. **Tenx** products are not designed, intended, or authorized for use in life support appliances, devices, or systems. If Buyer purchases or uses tenx products for any such unintended or unauthorized application, Buyer shall indemnify and hold tenx and its officers, employees, subsidiaries, affiliates and distributors harmless against all claims, cost, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use even if such claim alleges that tenx was negligent regarding the design or manufacture of the part.

AMENDMENT HISTORY

Version	Date	Description
V1.0	Aug, 2012	New release
V1.1	Jan, 2013	1. Modify Features section. 2. Replace "number system" with "value system". 3. Add "==" and "!=" in Logical Operations section. 4. Add .else if. 5. Add comment. 6. Add "_main". 7. Add reference between .EXPORT , .IMPORT and .AUTOIMPORT . 8. Add default value for ".DEFINEOLDSTYLE"
V1.2	Mar, 2014	1. modify the .autoimport function description 2. High line import instruction: import, export, autoimport
V1.3	Oct, 2019	1. Add ***(Suspend support, please use ".DT".)*** message.

CONTENTS

AMENDMENT HISTORY **2**
Features **5**
1. Limitations **5**
1.1 Naming Rule **5**
1.2 Case Sensitive **5**
1.3 Symbols and Labels **6**
2. Value System **7**
2.1 Representation **7**
2.2 Arithmetic, Bitwise and Logical Operations **7**
3. Pseudo Instructions **11**
3.1 Local Specified Instructions **11**
(1) ".CODE" 11
(2) ".RODATA" 11
(3) ".END" 11
3.2 Chip Type Instruction **11**
(1) ".CPU" 11
3.3 Other Instructions **12**
(1) ".ADDR" 12
(2) ".AUTOIMPORT" 12
(3) ".BYT", ".BYTE" 12
(4) ".CASE" 12
(5) ".DB" 13
(6) ".DBYT" 13
(7) ".DEFINE" 13
(8) ".DEF", ".DEFINED" 13
(9) ".DEFINEOLDSTYLE" 14
(10) ".DEFSTR" 14
(11) ".DN" 15
(12) ".DWORD" 16
(13) ".ELSE" 16
(14) ".ELSEIF", ".ELSE IF" 16
(15) ".ENDIF" 16
(16) ".ENDMAC",".ENDMACRO" 16
(17) ".ENDPROC" 16
(18) ".EQU" 17

(19) ".ERROR" 17
(20) ".EXITMAC",".EXITMACRO" 18
(21) ".EXPORT" 18
(22) “.FIXCODE” 18
(23) “.FPLANE0” 19
(24) “.FPLANE1” 20
(25) ".GLOBAL" 20
(26) ".IF" 21
(27) ".IFBLANK" 21
(28) ".IFDEF" 22
(29) ".IFNDEF" 22
(30) ".IMPORT" 23
(31) ".INCLUDE" 23
(32) ".LOCAL" 23
(33) ".MAC",".MACRO" 24
(34) ".MATCH" 25
(35) "_main" 26
(36) ".ORG" 26
(37) ".PARAMCOUNT" 27
(38) ".PROC" 27
(39) ".RELOC" 28
(40) “.RPLANE” 28
(41) ".SEGMENT" 29
(42) ".WORD" 29
(43) ".XMATCH" 30

4. Error Messages 32

Features

1. Operand can be defined as constant, and the constant can perform arithmetic, bitwise and logical operations.
2. Macro function. Frequently used program can be written as Macro which can be called by other programs.
3. Can compile and link multiple source files (Please refer to TICE99 IDE User Manual).
4. Each project can set library path for itself (Please refer to TICE99 IDE User Manual).
5. Provide multi-line comment which is similar with C language (/* -- This is comment -- */) and conditional compiler pseudo instruction (.if, .elseif, .endif).

1. Limitations

1.1 Naming Rule

The string of variable, constant and label can only use below text symbols:

0 ~ 9, a ~ z, A ~ Z, "_", however cannot be started with the number 0 ~ 9. Length of name is unlimited.

1.2 Case Sensitive

When assembling label name and macro name, user can decide whether to set as case sensitive (default setting is case sensitive, please refer to .CASE).

1.3 Symbols and Labels

(1) Numeric constants

Numeric constants are defined using equal sign (‘=’), for example:

two = 2

Then user can use constant “two” at anywhere in the program, and the content is 2.

<*Example*>: four = two * two

(2) Standard labels

1. The usage of labels is to define a label name at the beginning of each line, and the label name is then followed by a colon.
2. The statement without operand and it does not conflict with instruction name, and then assembler will treat it as label.

(3) Local labels and symbols

Use .PROC instruction, it can setup a program segment. Labels and symbols which are declared inside the segment are regarded as having local property. Outside this segment, these labels and symbols are treated as unknown and cannot be accessed.

(4) Using macros to define labels and constants

Sometimes, using macros to define labels and constants is not so convenient, but it may be convenient in some case. For example, use .DEFINE instruction to define symbols or other constants which are used in other place. Basically, using macro is not restricted; it can be used in low level operation or calculation. In addition, you can also use macro to define string constant (the other data types of symbols are not allowed).

<*Example*>:

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

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

.PROC                    ; Start local scope
 two = 3                 ; "two = 3", two is local constant
.ENDPROC                 ; end of local scope
```

2. Value System

2.1 Representation

(1) Binary

(1-1). Use “B” as identifier (“B” is case-insensitive), or use “%” at the beginning.

<*Example*>: 1000B, 10000100B, 1011b, b’10001000’

(1-2). Use “%” at the beginning.

<Example>: %1000, %10000100

(1-3). Use “B” at the beginning, use single quotes to mark the beginning and end of the value.

<Example>: B'1000', B'10000100'

(2) Decimal: no need to add any identifier

<*Example*>: 20, .20

Decimal (10, .10) syntax support

(3) Hexadecimal

(3-1). Use “H” as identifier (“H” is case-insensitive)

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

(3-2). Use “$” at the beginning.

<Example>: $9A, $FD

(It is suggested to use $ at the beginning, to avoid confusion in .EQU and .DN name definition)

(3-3). Use “0x” at the beginning (“x” is case-insensitive)

<*Example*>: 0x8, 0X0FF

(4) Constant or address: Use “.EQU” as identifier (“EQU” is case-insensitive). Or an equal sign “=” is followed behind the constant name.

<Example 1>: addr .EQU 4

<Example 2>: addr = 4

2.2 Arithmetic, Bitwise and Logical

Operations

In source code, there are some operations for the constants, including arithmetic operations, bitwise operations, and logical operations. Arithmetic operations follow the order of precedence of four types operations which is multiplication and division first, then addition and subtraction. The operators are as below:

"+": addition operator

"-": subtraction operator

"*": multiplication operator

"/": division operator

"%", ".MOD": modular operator

<Example>

```
addr .equ 4
movlw addr%3                ; movlw 1
movlw addr .mod 4           ; movlw 0
```

"<<": left shift operator

<Example>

```
adde .equ 2
movlw adde<<2               ; movlw 8
```

">>": right shift operator

<Example>

```
adde .equ  8
movlw adde>>2               ; movlw 2
```

"|", ".BITOR": bit-wise OR operator

<Example>

```
adde    .equ   2
movlw  addr | 4             ; movlw 6
movlw  addr .BITOR 2        ; movlw 2
```

"&", ".BITAND": bit-wise AND operator

<Example>

```
adde .equ    6
movlw adde & 4                    ; movlw 4
movlw adde .BITAND & 2            ; movlw 2
```

"~", ".BITNOT": bit-wise NOT operator

<Example>

Correct:

```
adde    .equ    2
molvw   ~adde & $FF               ; movlw $FD
movlw   .BITNOT adde & $FF  ; movlw $FD
```

Wrong:

```
adde    .equ   2
movlw  ~adde
movlw  .BITNOT adde
```

"^", ".BITXOR": bit-wise XOR operator

<Example>

```
adde .equ $F0
movlw adde ^ $FF                  ; movlw $0F
movlw adde .BITXOR $FF            ; movlw $0F
```

"(", ")": parentheses

<Example> Two-tier parentheses calculation

```
HIGHT   .equ   ((10+5)*2)/3
```

"=" or "==": assign operator (equal to)

"<>" or "!=": comparison operator (not equal to)

"<": comparison operator (less than)

">": comparison operator (greater than)

"<=": comparison operator (less than or equal to)

">=": comparison operator (greater than or equal to)

"&&", ".AND": Boolean AND operator

<Example>

```
adde .equ 0
.IF (adde>1) && 1
movlw adde                ; Will not execute
.ENDIF
```

"||", ".OR": Boolean OR operator

<Example>

```
adde .equ 1
.IF (adde>1) || 1
movlw  adde+$02           ; movlw $03
.ENDIF
```

".XOR": Boolean XOR operator

<Example>

```
adde .equ  20
.IF  (adde >20) .xor 0
movlw $20
.ENDIF
```

"!", ".NOT": Boolean NOT operator

<Example>

```
adde .equ 1
.IF  !(adde >20)
movlw $20
.ENDIF
```

3. Pseudo Instructions

3.1 Local Specified Instructions

Below segments have no precedence order; however, each segment must use KEYWORD to separate each other. No need to define unused segment.

(1) ".CODE"

Switch to code segment, case-insensitive. Define the program source code in this segment. It is abbreviation of ".segment "CODE"".

(2) ".RODATA"

Switch to data segment, case-insensitive. It is used to define Table ROM contents; in addition, it can use labels

(3) ".END"

Case-insensitive, it can be declared only once. Assembler will be forced to terminate if this instruction is encountered. Assembler will stop here, although the end of instruction is read from include file.

3.2 Chip Type Instruction

(1) ".CPU"

Case-insensitive, it can be declared only once, and can be declared anyplace outside of the segment (it is suggested to declare ".CPU" on the beginning of the file). The declaration is as below:

.CPU chip_species ;chip_species: chip code

<Example>

```
.CPU  TM57FLA80
.CODE
…….
movlw   10H
…….
.END
```

3.3 Other Instructions

(1) ".ADDR"

Define the data of 2-byte size. This is an alias name for ".WORD" instruction, and it is more readable when using ".ADDR" especially the data content is address. This instruction must be followed by a series of expressions (not necessarily have to be constant value, can also be an identifier).

<Example>

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

(2) ".AUTOIMPORT"

Auto import symbol definition feature. Using .autoimport on or .autoimport + to enable auto import symbol definition feature; or using .autoimport off or .autoimport - to disable auto import symbol definition feature (this is default value). The main difference between enable/disable auto import symbol definition features is: when the feature is enabled, undefined symbol during compilation stage will be ignored automatically and will not generate error messages, until linking stage, after linking all routines, when the symbol is not defined in all routines, the error messages of undefined symbol will be generated. Otherwise, when this feature is disabled and .import directive is not used to import the symbol definition, the undefined symbol during compilation stage will show undefined symbol error. (Please refer to .EXPORT, .IMPORT for the usage).

<Example>

```
        .autoimport     +       ; Enable auto import symbol feature
or
        .autoimport     on      ; Enable auto import symbol feature
```

(3) ".BYT", ".BYTE"

Define the data of 1-byte size. This instruction must be followed by a series of consecutive expressions or string. ***(Suspend support, please use ".DT".)***

<Example>

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

(4) ".CASE"

Enable or disable the case-sensitivity function during assembling time, default is disabled (means the identifier is case-insensitive by default). This instruction must be followed by a "+" or "-" sign to decide whether to enable or disable.

<Example>

```
        .case   -       ; Case-insensitive (reserved keyword or variable name)
or
        .case   on      ; Case-sensitive (reserved keyword or variable name)
```

(5) ".DB"

Define the data of byte size. ***(Suspend support, please use ".DT".)***

<Example>

```
.RODATA
.db '2','3','4', '5'                ; The byte results are $32 $33 $34 $35
.org 10h
.byte $12, $34, $56, $78, $9a       ; Data are set starting from TABLE
                                    ; ROM address 10h
.org 20h
.db $11, $22, $33, $44    ; Data are set starting from TABLE
                                    ; ROM address 20h
.db 'A', 'B', 'C', 'D'
```

(6) ".DBYT"

Define the data of 2-byte and high/low byte can be exchanged. This instruction must be followed by a series of word ranged. ***(Suspend support, please use ".DT".)***

<Example>

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

The byte results are $12、$34、$45、$12, and will be written into current segment according to this order.

(7) ".DEFINE"

This instruction must be followed by an identifier (macro name); in addition it can be followed by a series of variable in parentheses. After the parentheses, can be followed by a series of variable, please refer to .MACRO.

(8) ".DEF", ".DEFINED"

This instruction must contain a defined parameter inside the parentheses. This parameter will be checked, if this parameter has been defined somewhere else before current program position, the function will return TRUE; otherwise, return FALSE. Below *<Example>* can replace .IFDEF instruction:

<Example>

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

(9) ".DEFINEOLDSTYLE"

Declare identifier name can use special character, such as ( ) [ ], etc…, default value is: allow identifier name using special character.

<Example>

```
.defineoldstyle on                ; enabled

T_0.5S .equ 3DH

.define  LCDBrightness_(12Div19)   b'00000001'
.define  LCDpin_use_com[0~3]andseg[20~23]   b'00010000'
.define  LCD_ON  0x23

movlw  LCDBrightness_(12Div19)
movlw  LCDpin_use_com[0~3]andseg[20~23]
movlw  LCD_ON

movlw  LCDBrightness_(12Div19)
movlw  LCD_ON
movlw  T_0.5S

.defineoldstyle off               ; disabled
```

(10) ".DEFSTR"

.defstr (.defstring) :

<Example>

```
SET_BANK0  .defstr  BSF 03H,5

SET_BANK0 ; The same as BSF 03H,5

The same as using .define instruction to define:

.define   SET_BANK0   BSF 3,5
```

(11) ".DN"

Declare a variable to replace the address of data RAM, and define the Byte amount of data RAM which is occupied by this variable. In the program, this variable can directly replace the address of data RAM address. This instruction can only be declared inside RAM segment and constant segment. The declaration is as follows:

```
Variable   .DN        Value
```

where

Variable: variable name

Value: declare byte count of data RAM which is occupied by this variable, but it does not exceed max address of the data RAM. This instruction must be used together with ORG instruction, in order to define the starting address of the data RAM.

<Example>

```
.RAM
.ORG        20H          ; Define the start address of data RAM variable
DISPLAY .DN  1           ; Declare the data RAM 20H address as DISPLAY
SPEED    .DN  2          ; Declare the data RAM 21H and 22H address
                         ; as SPEED
                         ; SPEED replaces the 21H address of data
                         ; RAM
                         ; SPEED+1 replaces the 22H address of data
                         ; RAM
CHAR     .DN  1          ; Declare the data RAM 23H address as CHAR
ORG         30H
LCD1     .DN  1          ; Declare the data RAM 30H address as LCD1
.ENDRAM
.CODE
……
movwf  SPEED             ; SPEED replaces data RAM 21H address
                         ; SPEED+1 replaces data RAM 22H address
……
.END
```

(12) ".DWORD"

Define 4-byte data type; this instruction must be followed by a series of consecutive expressions. ***(Suspend support, please use ".DT".)***

<Example>

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

(13) ".ELSE"

Conditional instruction, which is used to reverse the condition expression (Please refer to .ERROR).

(14) ".ELSEIF", ".ELSE IF"

Conditional instruction, which is used to reverse current condition expression and to check other conditional expressions (Please refer to .ERROR).

".ELSEIF" is compatible with ".ELSE IF" instruction.

(15) ".ENDIF"

Conditional instruction which ends an .IF or .ELSE statement (Please refer to .ERROR).

(16) ".ENDMAC",".ENDMACRO"

End of macro definition (Please refer to .MACRO).

(17) ".ENDPROC"

End of part of program segment (Please refer to .PROC).

(18) ".EQU"

This instruction is used to define constant and it is case-sensitive. Declaration is as follows:

```
Constant        .EQU        data
```

where

Constant: constant name

Data: constant content

<Example *1*>

```
VALUE1      .EQU   10H
```

<Example *2*>

```
.RODATA
    AH      .EQU    00H
    BH      .EQU    01H
.CODE
    MOVLW   AH  ; AH is equal to 00H.
    MOVLW   BH  ; BH is equal to 01H.
.END
```

(19) ".ERROR"

Assembly error warning. Assembler will output a user-defined error message, and therefore object file will not be created. This instruction is used to check the conditions which must be meet by assembler.

<Example *1*>

```
.if       foo = 1
……
.elseif   bar = 1          ; same as .else if  bar == 1, or .elseif  bar == 1
……
.else
.error    "Must define foo or bar!"
.endif
```

<Example *2*>

```
.if DEBUG!=1            ; same as .if DEBUG <> 1
.error "No support in Debug mode"
.endif
```

(20) ".EXITMAC",".EXITMACRO"

Jump out from the macro immediately. This instruction is frequently used in recursive macro (Please refer to .MACRO).

(21) ".EXPORT"

Let the other programming files (*.asm, *.c) can also linked to current declared symbols. In this instruction, user must use commas to separate a list of symbols. Below <Example> uses two programming files for illustration, those are main.asm and labels.asm (Please refer to .AUTOIMPORT , .IMPORT).

<Example>

Main.asm

```
.include "$_texn_TM57FLA80_$.inc"
.autoimport on
call   lab
.end
```

labels.asm

```
.export  lab
lab:
movlw      02H
movwf      20H
```

(22) ".FIXCODE"

A switch to open or close "Freeze auto-switch RAM Bank" function, the default is closed (which means the auto-switch RAM Bank is taken over by compiler). This instruction must be followed by a "+" or "-" sign to decide whether to open or close the function.

<Example>

```
.fixcode +          ; Freeze auto-switch RAM Bank
.fixcode -          ; Auto-switch RAM Bank is done by compiler
```

In some conditions, if user wants to decide when to switch RAM Bank, user can use pseudo instruction .fixcode + for this purpose. Compiler will temporarily close auto-switch RAM Bank function, until .fixcode - is found, and then it will be reopened.

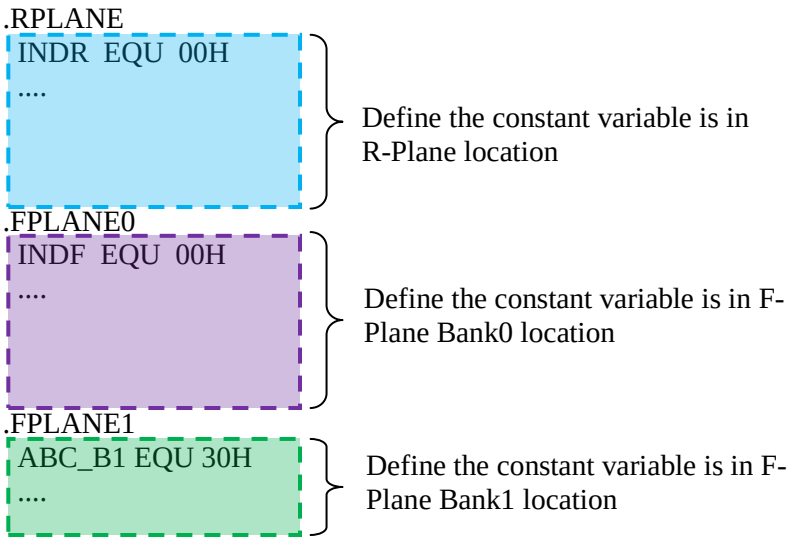
It is suggested that .fixcode + and .fixcode – should be used in pair, and when switch RAM Bank by user is not necessary, then switch RAM Bank should be done by compiler, to avoid unexpected error.

(23) ".FPLANE0"

In inc or asm file, user can define particular register as a significant constant variable name (refer to .EQU). Before a variable declaration is defined, RAM segment pseudo instructions (.FPLANE0, .FPLANE1, or .RPLANE) can be optionally used to clearly indicate the memory location of the variable (F-Plane bank0, F-Plane bank1 or R-Plane).

In using .FPLANE0, .FPLANE1, or .RPLANE pseudo instructions, it is suggested to declare .RPLANE at first then declare .FPLANE0 or .FPLANE1.

Example,

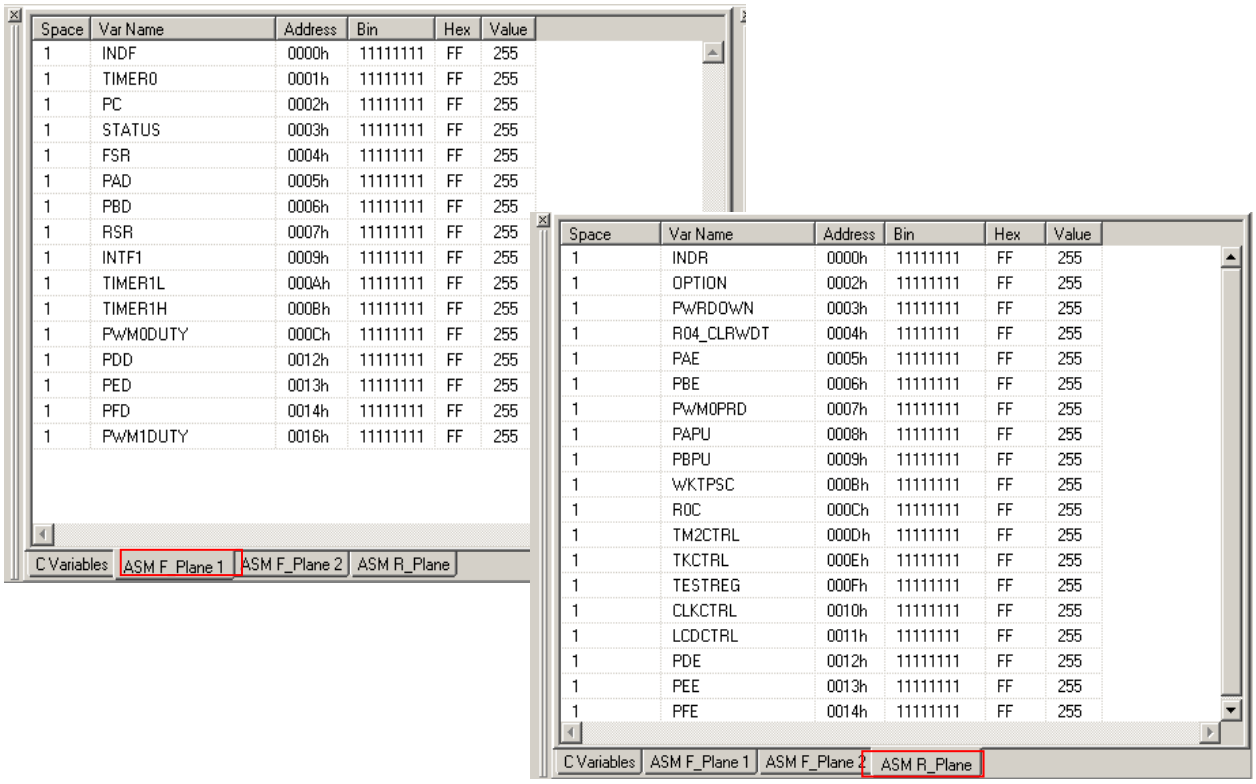


where

1. Constant variable INDR and the other variables which are defined before .FPLANE0 pseudo instruction are all defined in R-Plane location.
2. Constant variable INDF and the other variables which are defined before .FPLANE1 pseudo instruction are all defined in F-Plane bank0 location.
3. Constant variable ABC_B1 and variables defined later are all defined in F-Plane bank1 location.

Note:

1. Without declare pseudo instruction of RAM block, the variable are defined in F-Plane bank0 location by default.
2. If there is bank1 in operating memory, and declare a constant variable in F-Plane bank1, user still needs to control bank switch during instruction operation (i.e, by using instruction: bsf 0x3,5 to switch to bank1).
3. The purpose of .FPLANE0, .FPLANE1, or .RPLANE pseudo instructions is to classify the defined constant variable according to RAM block, and it can be shown according to the class in TICE99IDE assembler variable window. It is shown as below figure:



Space	Var Name	Address	Bin	Hex	Value
1	INDF	0000h	11111111	FF	255
1	TIMER0	0001h	11111111	FF	255
1	PC	0002h	11111111	FF	255
1	STATUS	0003h	11111111	FF	255
1	FSR	0004h	11111111	FF	255
1	PAD	0005h	11111111	FF	255
1	PBD	0006h	11111111	FF	255
1	RSR	0007h	11111111	FF	255
1	INTF1	0009h	11111111	FF	255
1	TIMER1L	000Ah	11111111	FF	255
1	TIMER1H	000Bh	11111111	FF	255
1	PWM0DUTY	000Ch	11111111	FF	255
1	PDD	0012h	11111111	FF	255
1	PED	0013h	11111111	FF	255
1	PFD	0014h	11111111	FF	255
1	PWM1DUTY	0016h	11111111	FF	255

C Variables | ASM F_Plane 1 | ASM F_Plane 2 | ASM R_Plane

Space	Var Name	Address	Bin	Hex	Value
1	INDR	0000h	11111111	FF	255
1	OPTION	0002h	11111111	FF	255
1	PWRDOWN	0003h	11111111	FF	255
1	R04_CLRWDT	0004h	11111111	FF	255
1	PAE	0005h	11111111	FF	255
1	PBE	0006h	11111111	FF	255
1	PWM0PRD	0007h	11111111	FF	255
1	PAPU	0008h	11111111	FF	255
1	PBPU	0009h	11111111	FF	255
1	WKTPSC	000Bh	11111111	FF	255
1	R0C	000Ch	11111111	FF	255
1	TM2CTRL	000Dh	11111111	FF	255
1	TKCTRL	000Eh	11111111	FF	255
1	TESTREG	000Fh	11111111	FF	255
1	CLKCTRL	0010h	11111111	FF	255
1	LCDCTRL	0011h	11111111	FF	255
1	PDE	0012h	11111111	FF	255
1	PEE	0013h	11111111	FF	255
1	PFE	0014h	11111111	FF	255

C Variables | ASM F_Plane 1 | ASM F_Plane 2 | ASM R_Plane

(24) ".FPLANE1"

The same as .FPLANE0, define the declared variable is in F-Plane Bank1 location.

(25) ".GLOBAL"

Declare global symbols. This instruction declaration must use commas to separate a list of symbols. The list of symbols are defined somewhere in source code, and they are exported. Import must be applied when other program needs to use these symbols. Besides, the same symbol is allowed to be used both in.IMPORT or .EXPORT instructions.

<Example>

```
.global foo, bar
```

(26) ".IF"

Evaluate an expression and based on the result value of the expression to determine whether assembler proceeds to output. This expression must be a constant expression, which means all operators must be a defined constant. If the expression value is 0, it means FALSE; otherwise, it means TRUE (Please refer to .ERROR).

(27) ".IFBLANK"

Conditional instruction, determine whether the parameters of macro are imported or not. If the condition does not hold, the next program will not be assembled until meet the .ELSE or .ELSEIF or .ENDIF instruction. This instruction is generally used to determine whether the parameters of macro are imported, if macro parameters are not imported, then it is TRUE, otherwise, is FALSE.

<Example>

```
.macro ADD2 v1,v2,sum
        movlw v1
.ifblank   v2
        addlw sum          ; If v2 IS NOT imported
.else
        addlw    v2        ; If v2 IS imported
.endif
        movwf sum
.endmacro
.code
        ADD2 1, , sum      ; If no parameters are imported, must be replaced
                           ; by ","
.endcode
```

(28) ".IFDEF"

Conditional instruction, test whether symbol is defined. This instruction must be followed by a symbol name. If the condition is hold (TRUE), means that symbol is defined, otherwise, is FALSE (Please refer to.MACRO).

<Example>

```
v1 .equ $10
      .ifdef v2            ; If v2 IS defined
v3  .equ   (v1>>3)
      movlw    v2
      movwf    0x20
      nop
      .else                ; If v2 IS NOT defined
v3   .equ   (v1>>2)
      movlw    value2
      movwf    0x20
      nop
.endif
```

(29) ".IFNDEF"

Conditional instruction, test whether symbol is defined. This instruction must be followed by a symbol name. If the condition is hold (TRUE), means that symbol is not defined, otherwise, is FALSE.

<Example>

```
v1 .equ $10
      .ifndef v2           ; If v2 IS NOT defined
v3 .equ (v1>>3)
      movlw    v2
      movwf    0x20
      nop
      .else                ; If v2 IS defined
v3 .equ (v1>>2)
      movlw    value2
      movwf    0x20
      nop
```

```
.endif
```

(30) ".IMPORT"

Import a symbol from the other module, this instruction is followed by a string of comma-separated imported symbol (Please refer to .EXPORT, .AUTOIMPORT for the usage).

<Example>

```
.import     foo, bar
```

(31) ".INCLUDE"

Include another file; the depth of nested including file is up to 16 levels.

Syntax:

```
#include "filename"
```

If filename consists of file path (Ex: #include "\tm57fla80\test\tm57.inc"), it will directly open include file, however, if the filename does not consist of file path (Ex: #include "tm57.ic"), the search priority will be as follows:

(1). At first, search the files in the project path

(2). If not found in (1), then search the installed path of TICE99IDE

(3). Search other include path which user specified

<Example>

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

(32) ".LOCAL"

Declare local label name, it can be used only in the macro; it cannot be used outside the macro. The purpose of using local declared label is to avoid problem especially when macro is expanded many times, the label will be used repeatedly. If local label is used outside the macro, error message will occur during assembling time.

<Example>

```
.macro ADD1 v1,v2,sum
.local L1               ; L1 is defined as label in ADD1 macro,
                        ; program outside macro cannot refer this label
    movlw   v1
    movwf   v2
    call    L1
    movwf   v2
L1:
    movlw sum
    ret
```

```
.endmacro
```

(33) ".MAC",".MACRO"

Start a macro definition, this instruction must be followed by an identifier (macro name) and can use comma to separate the identifier of the macro parameters.

<Example>

```
.CODE
.macro  foo     arg1, arg2, arg3
.if arg3 >0
        .define sum 123
.endif
.if     .paramcount < 3             ; Determine whether the imported
                                    ; variable of macro is less than 3
.error  "Too few parameters for macro foo" ; Output user-defined
                                    ; error message
.endif

.if     .paramcount > 3             ; Determine whether the imported
                                    ; variable of macro is larger than 3
.error  "Too many parameters for macro foo" ; Output user-defined
                                    ; error message
.exitmacro
.endif


movlw  0x01
movwf  0x20
movwf  0x21
.ifdef sum                          ; Determine whether sum is defined
addlw   0x04
.exitmacro
.endif
subwf   0x20,0                      ; Subtract the content of W with content
                                    ; in RAM 20H
.endmacro

start :
        foo 5,9,
        foo 5,9,1
        goto start
```

.END

(34) ".MATCH"

Built-in function. Matches two token lists against each other. This is most useful within macros, since macros are not stored as strings, but as lists of tokens. The syntax is

.MATCH(<token list #1>, <token list #2>)

Generally, macro parameter can be used as token list. Both token list may contain arbitrary tokens with the exception of the terminator token (comma resp. right parenthesis) and

- end-of-line
- end-of-file

Please note that the function only compares tokens, not token attributes. So any number is equal to any other number, regardless of the actual value. The same is true for strings. If you need to compare tokens and token attributes, use the .XMATCH function.

<Example>

```
Match_Value .MACRO V1, V2,

        .IF (.MATCH(V1,V2))        ; Compare if the type of V1 and V2 is the same
            MOVLW V1+V2            ; If yes, MOV V1+V2 TO W
            MOVWF rs               ; MOV W TO RSR
            NOP
        .ELSE                      ; If not, execute NOP
            NOP
        .ENDIF
        .IF (.XMATCH(V1,V2))       ; Compare if the value and type of V1 and V2
                                   ; are the same
            MOVLW V1+V2
            MOVWF pbd
            NOP
        .ELSE                      ; If value or type is different, then execute this..
            MOVLW V2-V1
            MOVWF pbd
            NOP
        .ENDIF

        .ENDM

.DEFINE TEST1 'A'                  ; Define TEST1=' A', ASCII code is 041H
.DEFINE TEST2 41h                  ; Define TEST2 = $41
.DEFINE TEST3 'C'                  ; Define TEST3 = 'C', ASCII code is 043H
```

Match_Value TEST1, TEST2

Match_Value TEST1, TEST3

(35) "_main"

Define the entry point of program in ASM file, same as void main() in C program to let Link program set the PC value, which is suggested when there are more than one *.ASM in one project or it is mixed project.

(36) ".ORG"

Define a starting address, for the follow-up program or the use of variable declarations. It can be defined multiple times in the same segment. If it is used in table segment (.RODATA), it is not suitable for C and ASM hybrid project. It is suggested for user to use in pure ASM project, can use with .RELOC instruction to avoid mistaken overlap condition. This instruction is case-insensitive, and the declaration is as follows:

.ORG Setting_addr

Setting addr: program, RAM or Table ROM address

When using ORG instruction in code segment, user can directly define next effective source address in the program. But, behind the label name, cannot be followed by ".ORG".

<Example>

```
.code
…..…
        btfss    0x0a, 1
        goto     30H
        btfss    0x0b, 0
        goto     40H
        btfss    0x0c, 1
        goto     50H
        .org     30H          ; Program PC address is 30H
        movlw    1H
……
        .org     40H          ; Program PC address is 40H
        movlw    2H
……
        .org     50H          ; Program PC address is 50H
        movlw    3H
……
.end
```

When using ORG instruction in RAM segment, user can directly define next data RAM address which is represented by data RAM variable which is defined by “DN” instruction.

(37) ".PARAMCOUNT"

Used in macro, to decide the number of parameters which are used in the macro (Please refer to .MACRO).

(38) ".PROC"

Using .PROC command will enter lexical declaration of function. All symbols defined after .PROC will only exist in this local declaration block, and cannot be accessed from outside. The symbols defined outside the declaration layer can be accessed as long as they are not redefined by local words. The symbols in other declaration layer will not cause naming conflicts, therefore, user can use same name in different declaration block to declare variables. When .ENDPROC instruction is found, the lexical declaration layer will be ended.

There are at most 8 layers of lexical declaration layer (for example, TM57FLA80 has 8 layers). .PROC instruction must be followed by a function name or label, to be called by this program function or other function. To allow other program to call, .export instruction must be used at first to export the program name. In this way, when it is in link process, linked program can use “.import” or “.autoimport on” pseudo instruction to check every symbol name which is imported, whether it is exported in other function by using .export pseudo instruction, if not, then the link error message will occur.

<Example>

```
                .export Clear       ; export program name, to be called by other program
                .proc Clear         ; Declare portion of Clear program, start a new
                                    ; declaration level
                movlw   $01
Clear_all :                         ; Clear_all is for local; if it is used in other place,
                                    ; it will not cause error in the same symbol
                movlw   $02
                movlw   $03
.endproc                            ; End of this declaration layer
Clear_all:                          ; Will not conflict with Clear_all mentioned above
                movlw   $04
```

(39) ".RELOC"

Interrupt ".ORG" command; let linker reorganize PC address (use with .ORG command).

<Example>

```
.RODATA    ; Declare 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      ; Interrupt the .org which is set in .RODATA, let link program to
            ; organize the PC address
LCD_clear:
MOVLW $00
        MOVWR $20
        MOVWR $21
        MOVWR $22
```

(40) ".RPLANE"

The same as .FPLANE0, define the declared variable is in R-Plane location.

(41) ".SEGMENT"

Command to switch to another segment. CODE and RODATA regularly output to their segments. The so-called “segment” is a kind of named data block; the default segment is program segment. There may be up to 254 different segments per object file (and up to 65534 per executable). CODE and RODATA are the two most frequently commands which are used to declare segment. Segment declaration command is followed by user defined segment name (there are some restrictions on the naming, it is suggested to use segment name which meets variable naming rule).

<Example>

```
.segment  "RODATA"          ; Switch to data segment
INT_Counter:
.db   set1        $02
.db   set2        $03
.segment  "CODE"            ; Switch to program segment
IntCounterMode:
movlw         $20/set1
movlw         $30/set2
```

(42) ".WORD"

Define 2 bytes data type; this command must be followed by a string of consecutive expressions (word ranged, not necessarily be a constant value).

<Example>

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

(43) ".XMATCH"

Built-in function. Matches two token lists against each other. This is most useful within macros, since macros are not stored as strings, but as lists of tokens

The syntax is

.XMATCH(<token list #1>, <token list #2>)

Often a macro parameter is used for any of the token lists. Both token list may contain arbitrary tokens with the exception of the terminator token (comma resp. right parenthesis) and

- end-of-line
- end-of-file

The function compares tokens and token values. If you need a function that just compares the type of tokens, have a look at the .MATCH function.

<Example>

```
Match_Value .MACRO V1, V2,
    .IF (.MATCH(V1,V2))     ; Decide whether the TYPE of V1 and V2 is
                            ; the same
        MOVLW V1+V2         ; If yes, MOV V1+V2 TO W
        MOVWF rsr           ; MOV W TO RSR
        NOP
    .ELSE                   ; If not, execute NOP
        NOP
    .ENDIF

    .IF (.XMATCH(V1,V2))    ; Decide whether the TYPE and content of
                            ; V1 and V2 are similar
        MOVLW V1+V2
        MOVWF pbd
        NOP
    .ELSE                   ; If there is a different, perform the following code
        MOVLW V2-V1
        MOVWF pbd
        NOP
    .ENDIF
    .ENDM

.DEFINE TEST1 'A'           ; TEST1 is character ‘A’, ASCII code is 041H
```

```
.DEFINE TEST2 41h          ; TEST2 is $41
.DEFINE TEST3 'C'          ; TEST3 is character ‘C’, ASCII code is 043H

Match_Value TEST1, TEST2
```

4. Error Messages

1. "Command/operation not implemented"
2. "Cannot open include file"

   Please make sure whether the file exists.
3. "Cannot read from include file"

   Please make sure whether the file exists.
4. "Include nesting too deep"

   Include files cannot be more than 245 levels depth.
5. "Invalid input character: "
6. "Hex digit expected"

   Please refer to Value System– Hexadecimal representation.
7. "Digit expected"

   Please refer to Value System– Decimal representation.
8. "`0' or `1' expected"

   Please refer to Value System– Binary representation.
9. "Numerical overflow"

   The integer is too large, must be less than or equal to $FFFFFFFF.
10. "Control statement expected"

    <Example>

    .INCLUDE a.asm Correction: .INCLUDE "a.asm"
11. "Too many characters"
12. "`:' expected"

    A colon is missing in the Label definition.
13. "`(' expected"

    A left parenthesis is missing in the arithmetic operation.

14. "`)' expected"

A right parenthesis is missing in the arithmetic operation.

15. ***Reserved***

16. "`,' expected"

<Example>

Correction:

```
        BCF port
        BCF port,0
```

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

18. ***Reserved***

19. ***Reserved***

20. "Integer constant expected"

21. "String constant expected"

<Example>

.include a.asm Correction: .include "a.asm"

22. "Character constant expected"

23. "Constant expression expected"

Constant name is undefined.

24. "Identifier expected"

The character in variable, constant and label can only contain below text symbols: '0' ~ '9', 'a' ~ 'z', 'A' ~ 'Z', '_', however cannot be started with the number 0 ~ 9.

25. "`.ENDMACRO' expected"

'.ENDMACRO' instruction must be used together with '.MACRO'.

26. "Option key expected"

27. "`=' expected"

28. ***Reserved***

29. "User error:"

User defined error message.

30. "String constant too long"

The max size of string constant is up to 255 characters.

31. "Newline in string constant"

String constant must be in the same line.

32. "Illegal character constant"

<Example>

.BYTE 'c1','2' Correction: .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"

42. "Symbol is already defined"

Symbol is defined repeatedly.

43. "Undefined symbol"

If the symbol name is in other source file, please use '.AUTOIMPORT ON'.

44. "Symbol is already marked as import"

45. "Symbol is already marked as export"

46. "Exported symbol is undefined"

Symbol name is not defined.

47. ***Reserved***

48. "Unexpected end of file"

49. "Unexpected end of line"

<Example>

.BYTE 'c','2', Correction: .BYTE 'c','2'

50. ***Reserved***

51. "Division by zero"

52. "Modulo operation with zero"

53. "Range error"

The integer constant size is more than data type size.

<Example>

.BYTE $1234 *Correction: .BYTE $12*

54. "Too many macro parameters"

There are too many parameters for a macro.

55. "Macro parameter expected"

56. "Circular reference in symbol definition"

57. "Symbol redeclaration mismatch"

58. "Alignment value must be a power of 2"

59. "Duplicate `.ELSE'"

'.ELSE' is reused before '.ENDIF'.

60. "Conditional assembly branch was never closed"

'.ENDIF' is missing in the end of the '.IF' or '.ELSE' conditional assembly 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. ***Reserved***

68. "Counter underflow"

69. ***Reserved***

70. ***Reserved***

71. "File name `%s' not found in file table"

72. "'.DN' must define in '.RAM' segment"

'.DN' must be used in '.RAM' segment.

73. "`.ENDRAM' expected"

'.RAM' and '.ENDRAM' must be in pairs.

74. "`.DN' expected"

'.DN' must be used in '.RAM' segment (Please refer to Pseudo Instruction – Other Instructions).

75. "Illegal data"

76. "Operand error"

The number of the operand is error.

<Example>

movlw src<<1, 1 ; Correction: movlw src <<1

77. "Cannot open COE file"

Please make sure whether COE file exists.

78. ***Reserved***

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

The max address of Program ROM is YYYYH, XXXXH is out of range.

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

The max address of Table ROM is YYYYH, XXXXH is out of range.