

十速

# TM57 C Compiler

## *User Manual*

## *Rev V1.4*

**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, 2011	New release
V1.1	Jan, 2013	<ol style="list-style-type: none"> <li>1. Add section: TM57 series C compiler special features</li> <li>2. Add RPLANE, FPLANE, bank1 keywords</li> <li>3. Bit operation only for F-Plane</li> <li>4. Add description of operation register</li> <li>5. Add description of union features</li> <li>6. Add bit variable addressing is across bank</li> <li>7. Return value is saved to op2</li> <li>8. Change "assembler" to "assembly"</li> <li>9. Add variable name will be changed after compiler</li> <li>10. Add interrupt protection function</li> <li>11. Add section 7</li> <li>12. Add section 8</li> <li>13. Add section 9</li> <li>14. Add section 10</li> </ol>
V1.2	Oct, 2013	<ol style="list-style-type: none"> <li>1. Add arrange the starting TABLE ROM address for global const variable and function</li> <li>2. Use pragma directive to specify the address of global const variable</li> <li>3. Specify the address of function in function definition section</li> <li>4. Use pseudo directive ".fixcode" in inline asm</li> <li>5. Add #pragma into the directive table</li> <li>6. Add section of "Pragma Directive (#pragma)"</li> <li>7. Add section of avoid using .org xx instruction in C and assemble hybrid programming</li> <li>8. Add the method of new a library project in IDE</li> </ol>
V1.3	Sep, 2021	<ol style="list-style-type: none"> <li>1. Add bit field struct comment.</li> <li>2. Add new asm format specifier %n and description.</li> </ol>
V1.4	Jun, 2022	<ol style="list-style-type: none"> <li>1. Add tableromdt description for Pragma Directive (#pragma) (support since 0.5.7 version)</li> </ol>

# CONTENTS

AMENDMENT HISTORY .....	2
CONTENTS.....	3
1. An Overview of TM57 C.....	7
TM57 Series C Compiler Special Features .....	7
Compiling C programs .....	8
Lexical Conventions .....	10
Source Program Character Set.....	10
Comments .....	11
Identifiers .....	11
Keywords .....	12
Constants.....	12
Numeric Constants.....	13
Character Constants .....	13
Enumeration Constants .....	13
Global Constants .....	14
String Literals.....	15
Operators .....	15
Punctuators .....	15
2. Meaning of Identifiers.....	16
Disambiguating names.....	16
Scope.....	16
Block Scope .....	16
Function Scope .....	16
Function Prototype Scope.....	17
File Scope (Global Scope) .....	17
Name Spaces of Identifiers .....	17
Linkage of Identifiers.....	18
Storage Duration .....	18
3. Declarations .....	19
Storage Class Specifiers .....	19
Type Specifiers .....	21

Fplane / Rplane Declarations .....	21
Structure and Union Declarations .....	24
Declaring and Using Bit Fields in Structures and Union .....	25
Bit Data Type.....	26
Bitwise Operator .....	28
Enumeration Declarations.....	29
Type Qualifiers .....	29
Declarators.....	30
Pointer Declarators .....	30
Array Declarators .....	31
Function Declarators and Prototypes.....	33
asm Declarators .....	34
Restrictions on Declarators .....	37
Typedef.....	39
Initialization.....	40
Initialization of Aggregates .....	40
4. Expressions and Operators .....	41
Operator Precedence and Associativity Rules in C .....	41
Primary Expressions .....	42
Postfix Expressions .....	42
Array Subscripting Operator.....	42
Structure and Union References .....	43
Indirect structure and Union References.....	43
Postfix ++ and Postfix – .....	43
Unary Operators .....	44
Address-of and Indirection Operators .....	44
Unary + and Unary – Operators.....	44
Logical Negation ! and Bitwise Negation ~ Operators.....	45
Prefix ++ and Prefix -- Operators.....	45
Sizeof Unary Operator.....	45
Multiplicative Operators .....	46
Additive Operators.....	46
Shift Operators .....	47

Relational Operators (< > <= >=) .....	47
Equality Operators (== !=) .....	48
Logical AND Operators (&&), Logical OR Operators (  ) .....	48
Conditional Operator.....	48
<b>5. Statements.....</b>	<b>50</b>
Expression Statements .....	50
Block Statement.....	50
Selection Statements.....	50
if Statement .....	51
switch Statement.....	52
Iteration Statements.....	52
while Statement .....	53
do Statement .....	53
for Statement .....	54
Jump Statements .....	55
goto Statement .....	55
continue Statement.....	55
break Statement .....	56
return Statement .....	56
Labeled Statement.....	56
Interrupt.....	57
(1) R-Plane .....	59
(2) F-Plane Bank 0 .....	59
(3) F-Plane Bank 1 .....	60
ISR_SaveData, ISR_RestoreData .....	61
ISR_SaveData_5, ISR_RestoreData_5, ISR_SaveData_10, ISR_RestoreData_10 .....	69
<b>6. Preprocessors.....</b>	<b>74</b>
Macro Definition .....	74
Non-parameter Macro Definition .....	74
Definition of Macro with Parameters.....	75
Files Include.....	75
Conditional Compilation .....	75
Pragma Directive (#pragma).....	76

<b>7. Mix of C and Assembly Code in C Project .....</b>	<b>78</b>
<b>Basic Concept .....</b>	<b>78</b>
<b>C Program Calls Assembly Function without Passing Parameter .....</b>	<b>79</b>
<b>C Program Calls Assembly Function with Passing Parameter .....</b>	<b>79</b>
<b>Assembly Code Calls C Function.....</b>	<b>80</b>
<b>C and Assembly Language Hybrid Programming Experiences .....</b>	<b>81</b>
(1) Using assembly instructions carefully.....	81
(2) Try to use embedded inline asm to replace .....	81
(3) Avoid using .org xx command while compiling C/ASM hybrid programming .....	81
<b>8. Create Function Library .....</b>	<b>82</b>
<b>Function Library .....</b>	<b>82</b>
<b>Use Function Library.....</b>	<b>82</b>
<b>Methods to Create the Library .....</b>	<b>82</b>
<b>How to Use Function Library .....</b>	<b>85</b>
<b>9. Memory Map .....</b>	<b>86</b>
<b>10. Appendix .....</b>	<b>87</b>
<b>Example 1.....</b>	<b>87</b>
<b>Example 2.....</b>	<b>91</b>
<b>Example 3.....</b>	<b>92</b>
<b>Example 4.....</b>	<b>94</b>

## 1. An Overview of TM57 C

### TM57 Series C Compiler Special Features

TM57 series C compiler complies with ANSI C standard (but, TM57 series C compiler does not support function pointer). Besides, to achieve optimal operating performance and control efficiency of tenx microcontroller, and to provide better programming support for C compiler programmer, the following features are added:

#### 1. Bit variable

- Declare variable of bit data type only in the global scope of program, please refer to [Bit Data Type](#) for the declaration syntax.
- Using bit field in structure, union, please refer to [Declaring and Using Bit Fields in Structures and union](#) for the declaration syntax.

#### 2. To allow C compiler programmer arranges global variable and function address more freely to meet actual needs. TM57 C compiler provides a feature to specify global variable address to which register (F-Plane or R-Plane) and specify starting TABLE ROM address for global const variable and function. In implementation, if F-Plane RAM has more than one bank, user can also determine global variable to be saved in specified bank of F-Plane.

- For declaration variable to specified register, please refer to [Fplane / Rplane Declarations](#) for the declaration syntax and notes.
- Arrange the starting TABLE ROM address for **global const variable**, user can specify [#pragma tableromaddr](#) to achieve this purpose.
- Specify the TABLE ROM address for function definition, please refer to [Function Declarators](#).

#### 3. Provide interrupt function and many interrupt protection features: when interrupt routing is triggered, operating register content may change and affect the result. Tenx provides not only auto-save chip, but also many kinds of interrupt protection features. These features allow user to decide which content of operating register will be saved efficiently based on different programming complexity.

- The corresponding relationship between programming complexity and operating register, please refer to [Operation Register](#).
- Operating register memory map, please refer to [Memory Map](#).
- Assembly subroutine code with interrupt protection feature, please refer to [Interrupt Protection](#).
- Precautions when interrupt protection feature is enabled, please refer to [Interrupt Restrictions](#).

#### 4. To meet single-chip's special command operating feature and real time control, assembly code relative to C programming language is more consistent with the needs of a single-chip. Therefore, C language and assembly code mixed programming is allowed in C project.

- Embed assembly code (asm , \_\_asm\_\_) directly in C program, please refer to [Asm Declarators](#) for the declaration syntax.

- To avoid variable, parameter, or function name in assembly code which may cause spelling and maintain problem, it is suggested to use **Format Symbol** to replace variable name, please refer to [format specifier](#).
- C code and assembly code mixed programming and example: (1) In C code calls assembly function, which is divided into with/without parameter, (2) In assembly code calls C function. Please refer to [Mix of C and Assembly Code in C Project](#).
- In some conditions, the most suitable C language and assembly code mixed programming experience sharing, please refer to [C and Assembly Language Hybrid Programming Experiences](#).

## 5. Support library

- C or assembly code can use TICE99 IDE tool to create function library, please refer to [Methods to Create the Library](#) for the detail steps to create library.
- Library reference, please refer to [How to Use Function Library](#).

## Compiling C programs

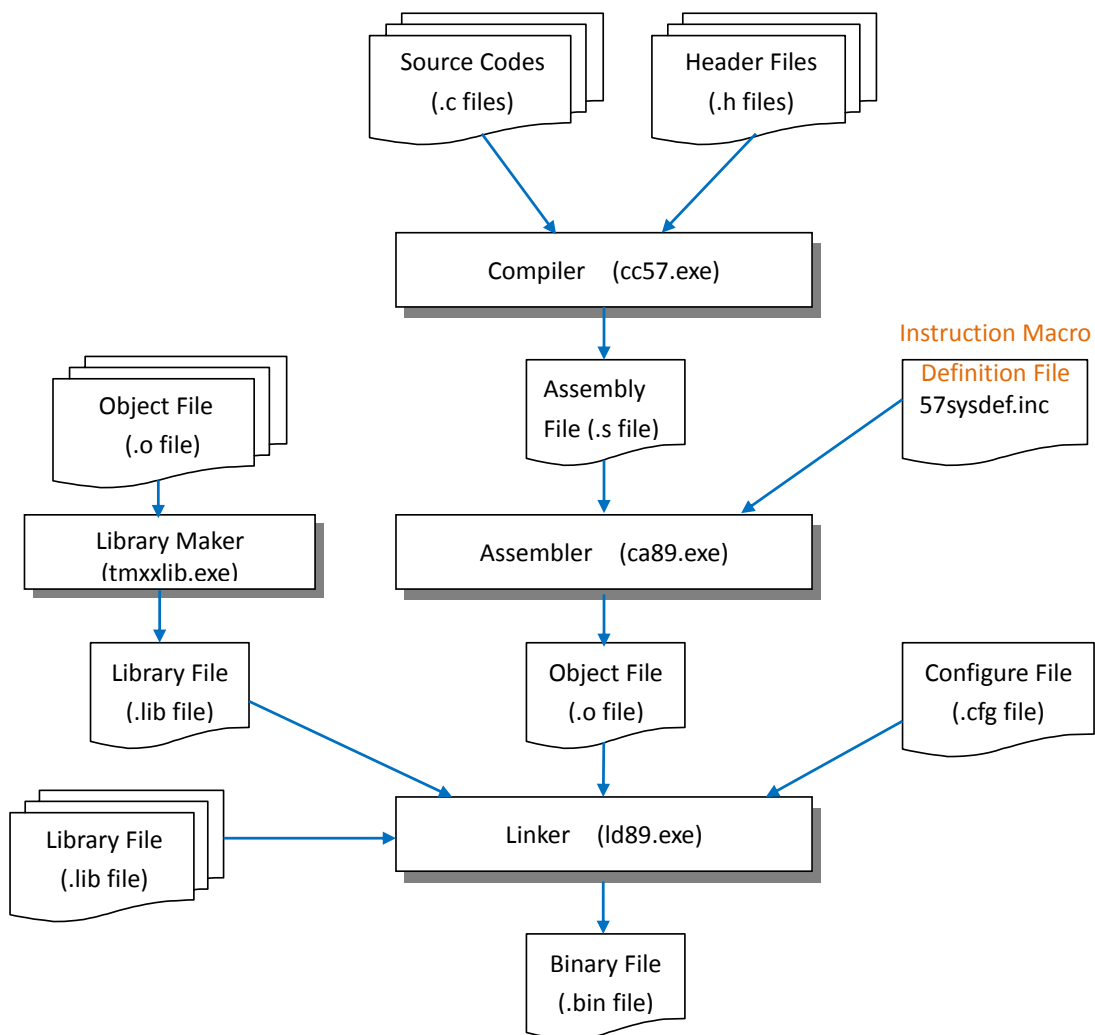
Microcontroller programs must fit in the available on-chip program memory, since it would be costly to provide a system with external, expandable memory. Compilers and assemblers are used to convert high-level language and assembly language codes into a compact machine code to be saved in the microcontroller's memory.

Compiling C programs requires you to work with five kinds of files:

1. **Regular source code files:** These files contain function definitions, and have names which end in ".c" by convention.
2. **Header files:** These files contain function declarations (also known as function prototypes) and various preprocessor statements. They are used to allow source code files to access externally-defined functions. Header files end in ".h" by convention.
3. **Object files:** These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in ".o" by convention.
4. **Chip relative files:** including runtime library files (runtime57\_XXX.lib files) and configure files (.cfg files). These files contain the information about memory relocation and instruction sets for each kind of chips.
5. **Binary executables:** These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables end in ".bin" by convention.

There are other kinds of files as well, assembly files (".s" files), and variable information files (".cfn" files), but you won't normally need to deal with them directly.





## Lexical Conventions

A lexical element refers to a character or groupings of characters that may legally appear in a source file. This section contains discussions of the C lexical conventions including tokens, character sets, comments, identifiers and constant literals.

A **token** is a series of contiguous characters that the C compiler treats it as a data unit. Blanks, tabs, newlines, and comments are collectively known as “white space.” White space is ignored by C compiler except as it serves to separate tokens. Some white space is required especially when it will separate otherwise adjacent identifiers, keywords, and constants. White space makes a program easier to read and maintain when it is used properly.

There are six different types of tokens:

- Identifiers
- Keywords
- Constant
- Literals
- Operators
- Punctuators

## Source Program Character Set

The following lists the basic source character sets that are available at both compile time and execution time:

- The uppercase and lowercase letters of the English alphabet

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- The decimal digits 0 through 9

0 1 2 3 4 5 6 7 8 9

- The underscore character (\_)

- The following punctuators (A punctuator is a character that has syntactic and semantic meaning)

!	#	&	(	{
"	%	'	)	}
,	-	.	/	
;	=	[	]	
<	>	\	_	
~	*	+	:	

- The space character
- Escape Sequences: some special and nongraphic characters are represented by the escape sequences.

The escape sequences and the characters they represent are:

Escape Sequence	Character Represented
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash

## Comments

A comment is text replaced during preprocessing by a single space character; the compiler therefore ignores all comments.

There are two kinds of comments:

- The /\* (slash, asterisk) characters introduce a comment, followed by any sequence of characters (including new lines), and the \*/ characters terminate a comment. This kind of comment is commonly called a C-style comment.
- The // (two slashes) characters followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. This kind of comment is commonly called a single-line comment.

**Note:** You cannot nest C-style comments inside other C-style comments. It means that each comment ends at the first occurrence of \*/. But You can nest single line comment within C-style comments

## Identifiers

An identifier, or name, consists of an arbitrary number of letters, digits, or the underscores (\_). The first character cannot be a digit. Uppercase and lowercase letters of identifier are distinct. The C compiler distinguishes between uppercase and lowercase letters in identifiers. For example, *TENX* and *tenx* represent different identifiers.

Identifiers provide names for the following language elements:

- Functions
- Objects
- Labels
- Function parameters
- Macros and macro parameters
- Typedefs
- Struct and union names

## Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is poor programming style. Only the exact spelling of keywords is reserved. To extend the ability of the C language to MCU features, some extra keywords are added into the TM57 C compiler. The following lists the reserved keywords in this compiler:

asm	enum	short	void
break	extern	signed	while
case	for	static	<b>__asm__</b>
char	goto	struct	<b>interrupt</b>
continue	if	switch	<b>rplane</b>
default	int	typedef	<b>bit</b>
Do	long	union	<b>FPLANE</b>
else	Return	unsigned	<b>RPLANE</b>
			<b>bank1</b>

### Note:

- **\_\_asm\_\_, interrupt, rplane, RPLANE, FPLANE, bank1 and bit** are the extra keywords for MCU.
- **float and double are NOT supported**

## Constants

A constant is non-addressable, it means its value is stored somewhere in memory, but we have no means of accessing that address. Every constant has a value and a data type.

The value of any constant does not change while the program runs and must be in the range of representable values for its type. The following are the available types of constant:

- Numeric constant
- Character constant
- Enumeration constant

## Numeric Constants

Numeric constant can be presented by decimal, hexadecimal and octal which depending on prefix modifier.

An octal constant consisting of a sequence of digits is considered octal if it begins with 0 (zero). An octal constant consists of the digits 0 through 7 only. A sequence of digits preceded by 0x or 0X is considered a hexadecimal integer. The hexadecimal digits include [aA] through [fF], which have values of 10 through 15. The suffix [L] or [l] traditionally indicate numeric constants of data type long. The suffix is allowed, but is superfluous, it makes program easier to be read.

- Syntax:
  - Decimal: Default
  - Hexadecimal constant: Digit prefix with “0x”
  - Octal constant: Digit prefix with “0”
- Example:

12, 34	// Decimal constant
0x5A, 0xB2	// Hexadecimal constant
014	// Octal constant
3452L	// Numeric constant of type long

**Note:** Binary constant is not supported.

## Character Constants

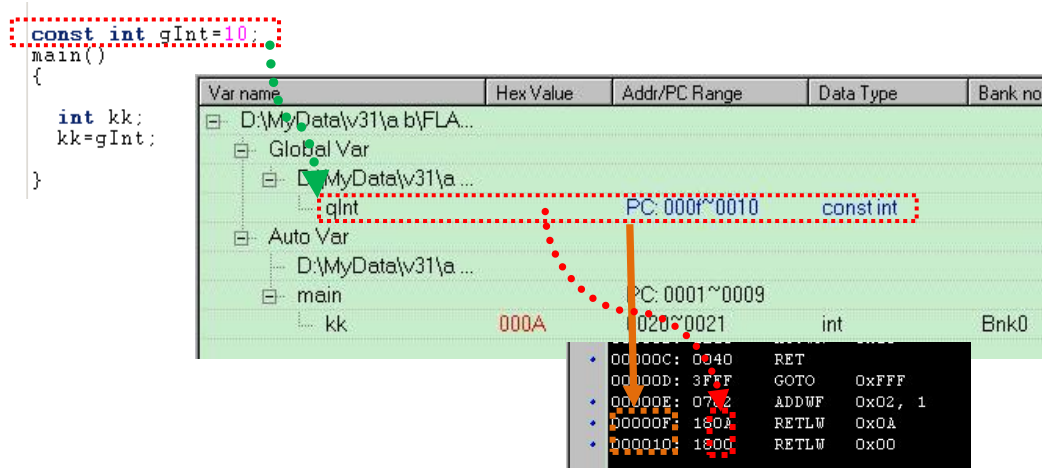
A character constant is a character enclosed in single quotation marks, such as 'x'. The value of a character constant is the numerical value of the character in the machine's character set. The type of a character constant is int.

## Enumeration Constants

In ANSI C, enumeration constants are simply integer constants that may be used anywhere. It means names declared as enumerators have type int. Similarly, ANSI C allows the assignment of other integer variables to variables of enumeration type, with no error.

## Global Constants

In TM57 C, global constant variables are stored in TABLE ROM (program memory). For example, declare a global constant: `const int gInt=10;` compiler address variable in 0009~000a, and data are combined by the last two bytes of the length of data type. In this example, the variable data is 0x000A = 10.



The screenshot shows a C program snippet on the left and its memory layout on the right.

```

const int gInt=10;
main()
{
    int kk;
    kk=gInt;
}
    
```

The memory layout table is as follows:

Var name	Hex Value	Addr/PC Range	Data Type	Bank no.
D:\MyData\w31\A b\FLA...				
Global Var				
D:\MyData\w31\A...				
gInt		PC: 000f~0010	const int	
Auto Var				
D:\MyData\w31\A...				
main		PC: 0001~0009		
kk	000A	0020~0021	int	Bnk0

Below the table, assembly instructions are shown:

```

00000C: 0040 RET
00000D: 3FFF GOTO 0xFFFF
00000E: 0702 ADDWF 0x02, 1
00000F: 180A RETLW 0x0A
000010: 1800 RETLW 0x00
    
```

User can also use preprocessor directive: `#pragma tableromaddr` to define starting address of global variable, for example, global variable array[6] is addressed at 0x0100; global variable gVar1 does not need to define address and it is addressed by C compiler. C program is shown as below figure:

```

3 #pragma tableromaddr (0x0100)
4 const unsigned char array[6] = {2,3,4,5,6,7};
5 #pragma tableromaddr (off)
6 const gVar1 = 23;
    
```

Var name	Dec Value	Addr/PC Range	Data Type	Bank no.
D:\bcb_test\fla80_cost...				
Global Var				
D:\bcb_test\fla8...				
array		PC: 0101~0106	const BYTE[6]	
[0]	2	PC: 0101~0101	const DATA_TYPE_BYTE	
[1]	3	PC: 0102~0102	const DATA_TYPE_BYTE	
[2]	4	PC: 0103~0103	const DATA_TYPE_BYTE	
[3]	5	PC: 0104~0104	const DATA_TYPE_BYTE	
[4]	6	PC: 0105~0105	const DATA_TYPE_BYTE	
[5]	7	PC: 0106~0106	const DATA_TYPE_BYTE	
gVar1	23	PC: 0008~0009	const int	

**Note:** Please note the arrangement of global variable, to avoid address conflicted error.

## String Literals

A string literal is a sequence of characters surrounded by double quotation marks, such as "abc". A string literal has type array of char and is initialized with the given characters. TM57 C compiler will place a null byte (\0) at the end of each string literal so that programs that scan the string literal can find its end. In addition, the escapes as those described for character constants can be used (See [Source Program Character Set](#) for a list of escapes).

## Operators

An operator specifies an operation to be performed. The operators [ ] and ( ) must occur in pairs, possibly separated by expressions. Operator can be one of the following:

[ ] ( ) . ->  
++ -- & \* + - ~ ! sizeof  
/ % << >> < > <= >= == != ^ | && ||  
?:  
\*= /= %= += -= <<= >>= &= ^= |=

## Punctuators

A punctuator is a symbol that has semantic significance but does not specify an operation to be performed. The punctuators [ ], ( ), and { } must occur in pairs, possibly separated by expressions, declarations, or statements. Punctuator is one of the following:

[ ] ( ) { } \* , : = ; ... #

Some operators, determined by context, are also punctuators. For example, the array index indicator [ ] is a punctuator in a declaration.

## 2. Meaning of Identifiers

An ANSI C identifier is disambiguated by four characteristics: its *scope*, *name space*, *linkage*, and *storage duration*. Storage-class specifiers are discussed in this chapter only in terms of their effect on an object's storage duration and linkage.

### Disambiguating names

This section discusses the ways of C disambiguates names: *scope*, *name space*, *linkage*, and *storage class*.

### Scope

The largest region of a program in which a given identifier is visible and can potentially be used to refer to its object is called the ***scope of identifier***. Compiler uses the rules of scope and name resolution to determine whether a reference to an identifier is legal at a given point in a file.

### Block Scope

Block scope is the scope of automatic variables which is declared within a function or a block. No conflict occurs if the same identifier is declared in two different blocks. When one block is nested inside another, the identifier from the outer block is usually visible in the nested block. The other one in the nested block hides until the end of the enclosed block is reached. The outer declaration is restored when program control returns to the outer block. This is called ***block visibility***.

### Function Scope

Only ***labels*** have function scope. A label is implicitly declared by its appearance in the program text and is visible until the end of the current function that declares it.

A label can be used in a [goto Statement](#) before the actual label is seen.



## Function Prototype Scope

If an identifier appears within the list of parameter declarations in a function prototype that is not part of a function definition, it has function prototype scope. Function prototype scope terminates at the end of the prototype.

Consider the following example:

```
char * getEnvName (const char * name);  
int name;
```

The `int` variable `name` does not conflict with the parameter name because the parameter went out of scope at the end of the prototype. However, the prototype is still in scope.

## File Scope (Global Scope)

File scope (or global scope) applies to identifiers appearing outside of any block, function, or function prototype. An identifier with global scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

An identifier with global scope is also accessible for the initialization of global variables. If that identifier is declared ***extern***, it is also visible at link time in all object files being linked.

## Name Spaces of Identifiers

The C compiler sets up name spaces to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different name spaces do not interfere with each other, even if they are in the same scope. You can redefine identifiers in the same name space but within enclosed program blocks.

ANSI C recognizes the following four distinct name spaces:

- **Tags:** *struct*, *union*, and *enum* tags have a single name space.
- **Labels:** labels are in their own name space.
- **Members:** each *struct* or *union* has its own name space for its members.
- **Ordinary identifiers:** the following identifiers must be unique within a single scope.
  - C function names
  - Variable names
  - Names of function parameters
  - Enumeration constants
  - typedef names

## Linkage of Identifiers

Linkage refers to the use or availability of an identifier that across multiple translation units or within a single one. The term translation unit refers to a source code file plus all the header and other source files that are included after preprocessing with the `#include` directive, minus any source lines skipped because of conditional preprocessing directives (because there are some conditional compilation of preprocessing instructions, `#if...#else...#endif`). Linkage allows the correct association of each instance of an identifier with one particular object or function.

## Storage Duration

The scope of an identifier is interrelated with the storage duration of the identified object, which is the length of time that an object remains in an identified region of storage. The lifetime of the object is influenced by its storage duration, which in turn is affected by the scope of the object identifier.

### 3. Declarations

Declarations determine the interrelated attributes of an object: storage class, type, scope, visibility, storage duration, and linkage.

Declarations have the following form:

<i>declaration:</i>	<i>declaration-specifiers</i> [ <i>init-declarator-list</i> ]
---------------------	---

The ***declaration specifiers*** consist of a sequence of specifiers that determine the linkage, storage duration, and part of the type of the identifiers indicated by the declarator.

<i>declaration-specifiers:</i>	<i>storage-class-specifier</i> [ <i>declaration-specifiers</i> ] <i>type-specifier</i> [ <i>declaration-specifiers</i> ] <i>type-qualifier</i> [ <i>declaration-specifiers</i> ]
--------------------------------	--

The ***init-declarator-list*** is a comma-separated sequence of declarators and it is optional, each of which can have an initializer.

<i>init-declarator-list:</i>	<i>init-declarator</i> <i>init-declarator-list</i> , <i>init-declarator</i>
<i>init-declarator:</i>	<i>Declarator</i> <i>declarator</i> = <i>initializer</i>

Declarations determine the following properties of data objects and their identifiers:

- ***Scope***, the region of program text in which an identifier can be used to access its object.
- ***Visibility***, the region of program text from which legal access can be made to the identifier's object.
- ***Duration***, the period during while the identifiers have real, physical objects allocated in memory.
- ***Linkage***, the correct association of an identifier to one particular object.
- ***Type***, which determines how much memory is allocated to an object and how the bit patterns found in the storage allocation of that object should be interpreted by the program.

#### Storage Class Specifiers

The storage class specifier indicates linkage and storage duration. Storage class specifiers have the following form:

<i>storage-class-specifier:</i>	static extern typedef rplane
---------------------------------	---------------------------------------

The typedef specifier does not reserve storage and is called a storage-class specifier only for syntactic convenience.

An rplane declaration enables users to address a global variable in the appointed rplane RAM.

Example:

```
rplane int _gx; // Allocation R plane variable _gx at global area

void main(void)
{
    int i,j;
    _gx = i+j; // save i+j to R-plane RAM _gx
}
```

**Note:** even some of the TM57 series chips provide write operation mode in R-plane RAM, but it is not allowed to read or write R-plane RAM data. Please see the relative specification document of current TM57 chip for detail.

```
#define _PWRDOWN 0x03
unsigned char _powerdown @_PWRDOWN:RPLANE ;
rplane int _gx;
void main(void)
{
    unsigned char uc;
    rplane int ri; // Error, must allocation at global area

    uc=_gx; // it will show Error when compiling,
            // if R-plane RAM is write only ( Ex: TM57PA40 chip )
    uc=_powerdown; // it will show Error when compiling/assembling,
                  // if R-plane RAM is write only ( Ex: TM57PA40 chip )
}
```

The following rules apply to the usage of storage class specifiers:

- A declaration can have at most one storage class specifier. If the storage class specifier is missing from a declaration, then this storage is maintained only during the execution of the block where the object is defined (auto variable).
- Identifiers declared within a function with the storage class **extern** must have external linkage, which means that it can be called from other translation units.
- Identifiers declared with the storage class **static** have static storage duration, and either internal linkage (if declared outside a function) or no linkage (if declared inside a function). If the identifiers are initialized, the initialization is performed once before the beginning of execution. If no explicit initialization is performed, static objects are implicitly initialized to zero.

## Type Specifiers

Type specifiers are listed below. The syntax is as follows:

```

type-specifier:      struct-or-union-specifier
                    typedef-name
                    enum-specifier
                    char
                    short
                    int
                    long
                    signed
                    unsigned
                    void
                    bit

```

**Note:** data type *float* and *double* are **NOT** supported.

## Fplane / Rplane Declarations

In TM57 chip series, the register is divided into two memory blocks: F-Plane and R-Plane. It gives a more convenience way to enable users to address a global variable in the appointed R-plane or F-plane RAM. RAM storage declaration has the following form:

```

declaration:      type-specifier Var_name @ Address [@bit]:(RPLANE|FPLANE)

```

Address format without "@" modifier is a decimal address, otherwise is a hexadecimal address.

Example:

C code	ASM code
<pre> /* Specify the address of var: F_Addr to 0x30 in F-plane*/  // declare global variable unsigned char F_Addr@0x30:FPLANE;  main() {     F_Addr=0x20; /* initial value = 0x20 */ } </pre>	<pre> 000000: 3001 GOTO    0x1 000001: 2005 CALL    0x5 000002: 1920 MOVLW   0x20 000003: 00B0 MOVWF   0x30 000004: 0040 RET 000005: 0040 RET </pre>

TM57 C provides a bit operation for F-Plane in order to maintain bit type of IC register. The address range for IC control address which can be address bit field, please see IC specification for detail. Bit operation example as below:

```

/* Specify the address of var: Addr to 0x30 in F-Plane*/

// declare global variable to access the bit field 0 of address 0x30 in F-Plane

unsigned bit Addr@0x30@0:FPLANE;

main()
{
    Addr=0x20; /* initial value = 0x20 */
}

```

**Note:**

- In R-plane, the instructions that are allowed to do memory access are MOVWR, MOVRW, and we do not recommend doing bit-wise operation because the data unit is BYTE.
- F-plane and R-plane declarations only enable users to address a global variable.

In linking process, linker will address the operation register from 0x20 to 0x37 (24 bytes) in F-Plane RAM, but the actual register usage depends on the complexity of operation, and whether there is bit variable declaration in program. The maximum register usage is as follows:

Address	Size (bytes)	Operation Register	Address Range
0x20~0x2F	16	op1 ~ op4	Floating
-	1	tmp1	Floating
-	4	stkptr	Floating

Determining whether to save operating register (op1~op4, tmp1, stkptr) and allocate the memory to save the content of register depend on the programming computational complexity (refer to [registers in expression](#)). This means that with the increase of the computational complexity, there are more operating register involved. Besides, consider efficient usage of memory and interrupt protection storage register, linker uses contiguous allocation memory to save operating registers which are used in computational procedure, and sets accurately the memory size needed. Therefore, the range of operating register address is not fixed. Furthermore, the common bit variable address range decided by linker is 0x20 ~ 0x3F, the actual bit variable address after saving operating register is described as following example:

Assume in computing process, needs to save op1: 2 bytes, op3: 4 bytes, and stkptr: 1 byte, at the same time there are two bit variables declared in the program. Therefore, 0x20 ~ 0x26 is assigned to store op1, op3 and stkptr register, and 0x27 is assigned to store two bit variables.

0x20	0x22	0x26	0x27
op1	op3	stkptr	Bit variable

**Note:** Bit variable declaration described above means **common bit variable declaration** (not specific bank1 and F-Plane address of bit variable declaration), which means the address of bit variable which is decided by linker, for example: bit bVal;.

Address range mentioned above is reserved for runtime library computing, use address range to store computational result. It is suggested that user do not use this address range to avoid unexpected error

occurs, especially when there are C code and ASM code occur at the same time in a project. Furthermore, it is strongly recommended that user using a simple expression to reduce computational complexity.

Below table shows registers may be used in all arithmetic expressions.

- Common rules:

	Multiplication 8	Multiplication 16	Multiplication 32	Division 8	Division 16	Division 32
op1	☑	☑	☑	☑	☑	☑
op2	☑	☑	☑	☑	☑	☑
op3	-	☑	☑	-	☑	☑
op4	-	-	☑	-	-	☑
tmp1	-	-	-	-	-	☑

- Addition – subtraction expression: depends on the complexity of expression, there will be different registers adjustment, actual register usage condition, please check the interpreter result in \*.s file.

	Subtraction 8	Subtraction 16	Subtraction 32	Addition 16	Addition 32
op1	-	☑	☑	☑	☑
op2	-	☑	☑	☑	☑
op3	-	-	-	-	-
op4	-	-	-	-	-
tmp1	-	-	-	-	-

	Memory content copy	Pointer (read/write) computational
op1	☑	☑
op2	☑	☑
op3	☑	☑
op4	-	-
tmp1	-	-

Note: Below three examples will trigger memory content copy computational:

- String initialization
- Table ROM copy
- String expression

## Structure and Union Declarations

A **structure** is an object consisting of an ordered group of data members. Unlike the elements of an array, each member within a structure can have various data type. A **union** is an object that can, at a given time, contain any one of several members.

Structure and union specifiers have the same form. The syntax is as follows:

<i>struct-or-union-specifier:</i>	<i>struct-or-union {struct-decl-list}</i> <i>struct-or-union identifier {struct-decl-list}</i> <i>struct-or-union identifier</i>
<i>struct-or-union:</i>	<b>struct</b> <b>union</b>

The *struct-decl-list* is a sequence of declarations for the members of the structure or union.

The declaration syntax between struct and union is similar, but from the viewpoint of the memory, each member of struct has its own memory space. The space occupied by a struct is the sum of the memory space occupied by each struct and the boundary alignment.

Unlike struct, union does not configure the memory space for each member, but all union data members share the same memory space. The memory size is the largest data type among the members. Therefore, when any data content in a field changes, it will relatively affect the content of other fields. This means, at the same time, only the data of one member can be saved. Therefore, a union only configured a large enough space to store the largest data member.

Use structures to group logically related objects. In the following example, line `int street_no;` through to `char *postal_code;` declare the structure tag address:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};

struct address perm_address;
struct address temp_address;
struct address *p_perm_address = &perm_address;

/*
The variables perm_address and temp_address are instances of the structure data type address.
Both contain the members described in the declaration of address. The pointer p_perm_address
points to a structure of address and is initialized to point to perm_address.
*/
```



## Declaring and Using Bit Fields in Structures and Union

ANSI C allows integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called **bit fields**, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

The default type of field members is **int**, but whether it is signed or unsigned int is defined by the implementation. Therefore, you should specify the signedness of bit fields when they are declared.

```
struct on_off {
    unsigned light : 1; // Low Bytes
    unsigned toaster : 1;
    unsigned ac : 4;
    unsigned clock : 1;
    unsigned flag: 1;//High Bytes
} kitchen ;
```

In the above example, the structure kitchen contains five members totaling 1 byte. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
Light	1 bit
toaster	1 bit
ac	4 bits
clock	1 bit
flag	1 bit

In the previous section, it is mentioned that when a field data in union changes, it will relatively affect the content of other fields. This union feature provides an efficient storage control in hardware programming. Below is an example:

Assume a variable **flag** which occupies a byte is defined in programming code, in implementation, if each bit is to be set individually at the same time, union definition can be used to operate Byte and bit data members which occupy the same memory space.

```
union test{
    unsigned char flag;
    bit flag_bit0;//Low Bytes
    bit flag_bit1;
    bit flag_bit2;
    bit flag_bit3;
    bit flag_bit4;
    bit flag_bit5;
    bit flag_bit6;
    bit flag_bit7;//High Bytes
}TEST;
```

From TEST example, we can operate a byte data (ex: `TEST.flag=10;`), and at the same time can operate each bit of data (ex: `TEST.flag_bit=1;`). In this way, no need to use OR, AND or SHIFT operation, to achieve the same effects. In 8-bit single-chip programming, it provides intuitive and convenience of C code, and the flexibility of assembly language.

Note:

1. The data type of each struct member is unlimited, therefore, when bit-field and other data type member are declared at the same struct, it is suggested to use **continuous declaration of bit-field** to save the memory space. In this way, linker will centralize the configuration address according to byte units.
2. Struct and union can declare specific address. Below is the syntax to specify the address:

```
declaration:      struct-or-union identifier Var_name @ Address :(RPLANE|FPLANE)
```

## Bit Data Type

Different from ANSI C, TM57 C has built-in support for bit fields. Bit datatype is used to store boolean information, i.e. 1 or 0 (true or false). Use the bit datatype to represent true/false or yes/no types of data, such as status flag, led status.....

In TM57 single chip series, the basic data unit of control register is the binary digit, or bit. Values with more than two states require multiple bits. The first half of F-Plane is bit-addressable, while the second half of F-Plane is not bit-addressable, and please see specification for the detail. The format is as below:

<i>bit-specifier:</i>	<b>bit identifier</b> <b>bit identifier@address@bit:plane_type</b> <b>bit identifier:bank1</b>
<i>plane_type</i>	<b>FPLANE</b>

**Note:** bit type variable can only be declared in **global area**. And also, when it is necessary to specify bit variable RAM storage address, it is allowed only in FPLANE.

If a local variable is declared as a bitfield, a read/write may access the entire storage unit (that is byte) containing the field. Below is the example

```
const int gInt = 10;
bit bit1;
bit bit2:bank1;
main()
{
    int kk;
    bit1 = 1;
    bit2 = 0;
    kk = gInt;
}
```

Var name	Hex Value	Addr/PC Range	Data Type	Bank no.	Plane
Global Var					
D:\bcb_test\pa20_0720\DE...					
gInt	000A				
bit1	0	0020.0	bit	Bank0	FPlane
bit2	0	0028.0	bit	Bank1	FPlane
Auto Var					
D:\bcb_test\pa20_0720\DE...					
main		PC: 0001~0009			
kk	531F	0021~0022	int	Bank0	FPlane

Declare a global bit variable and address in F-PLANE.

Example:

```
bit gBit_1 @0x20@0:FPLANE;
bit gBit_2 @0x20@1:FPLANE;
bit gBit_3 @0x20@2:FPLANE;

main()
{
    gBit_1 = 0;
    gBit_2 = 1;
    gBit_3 = 0;
}
```

Address: 0x0020 in FPLANE

0	1	0				
---	---	---	--	--	--	--

Bit 2 : gBit\_3  
Bit 1 : gBit\_2  
Bit 0 : gBit\_1

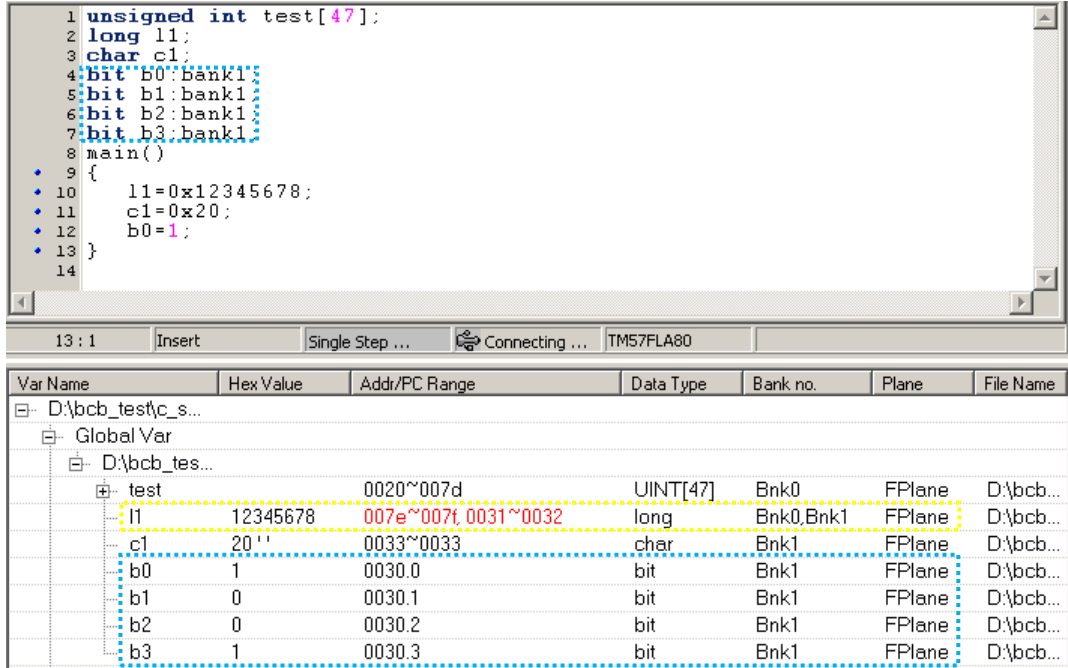
Var Name	Hex Value	Addr/PC Ran...	Data Type	Bank no.	Plane	File ...
Global Var						
D:\bcb_test\c...						
gBit_1	0	0020.0	bit	Bank0	FPlane	D:\...
gBit_2	1	0020.1	bit	Bank0	FPlane	D:\...
gBit_3	0	0020.2	bit	Bank0	FPlane	D:\...
Auto Var						
D:\bcb_test\c...						

C Variables   ASM F\_Plane 1   ASM F\_Plane 2   ASM R\_Plane

Bit variable according to actual needs can assign address at bank1 of F-Plane (when it is unsigned, linker default setting address is at bank0, therefore no need to assign address at bank0). Linker sets bit address at bank1 of F-Plane as the starting address. Below figure using TM57FLA80 as example, bank1

addressable starting address is 0x30, therefore the bit variable b0, b1, b2, b3 assigned address at bank1 is the first 0~3 bits of 0x30.

Please note that the length of the integer variable in example is 11, the storage address is across bank0 and bank1, linker will configure variable l1 address to 0x7e~0x7f of bank0 and 0x31~0x32 of bank1, to avoid the configured address 0x30. This means, when there is bit variable with assigned address in bank1 and variable in program which across bank0 and bank1, the across bank variable configuration will be readjusted by linker, to avoid repeat addressing.



```

1 unsigned int test[47];
2 long l1;
3 char c1;
4 bit b0:bank1;
5 bit b1:bank1;
6 bit b2:bank1;
7 bit b3:bank1;
8 main()
9 {
10     l1=0x12345678;
11     c1=0x20;
12     b0=1;
13 }
14

```

Var Name	Hex Value	Addr/PC Range	Data Type	Bank no.	Plane	File Name
D:\bcb_test\c_s...						
Global Var						
D:\bcb_tes...						
test		0020~007d	UINT[47]	Bnk0	FPlane	D:\bcb...
l1	12345678	007e~007f, 0031~0032	long	Bnk0,Bnk1	FPlane	D:\bcb...
c1	20	0033~0033	char	Bnk1	FPlane	D:\bcb...
b0	1	0030.0	bit	Bnk1	FPlane	D:\bcb...
b1	0	0030.1	bit	Bnk1	FPlane	D:\bcb...
b2	0	0030.2	bit	Bnk1	FPlane	D:\bcb...
b3	1	0030.3	bit	Bnk1	FPlane	D:\bcb...

**Suggestion:** when declare bit variable, it is suggested to use continuous declaration, in this way, linker will centralize the configuration address, to save address space. Otherwise, if it is uncontinuous declaration bit address, linker will not continuously configure the address, this will cause address space wasted.

## Bitwise Operator

For bit data type, TM57 C supports the following operator:

- Mathematic operator : + - \* /
- Bitwise operator : & | ~

**Note:** shift operator (<< >>) is not supported for bit variable

## Enumeration Declarations

An enumeration is a data type which consists of a set of values that are named integral constants. The syntax is as follows:

<i>enum-specifier:</i>	<b>enum</b> { <i>enum-list</i> } <b>enum</b> { <i>identifier enum-list</i> } <b>enum</b> <i>identifier</i>
<i>enum-list:</i>	<i>enumerator</i> <i>enum-list</i> , <i>enumerator</i>
<i>enumerator:</i>	<i>identifier</i> <i>identifier</i> = <i>constant-expression</i>

When you define an enumeration data type, you specify a set of identifiers. The identifiers are declared as *int constants* and can appear wherever such constants are allowed. Each identifier in this set is an *enumeration constant*.

The value of the enumeration constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

Example:

```
enum grain { oats, wheat, barley, corn, rice };  
/* 0 1 2 3 4 */  
enum grain { oats=1, wheat, barley, corn, rice };  
/* 1 2 3 4 5 */  
enum grain { oats, wheat=10, barley, corn=20, rice };  
/* 0 10 11 20 21 */
```

## Type Qualifiers

Type qualifiers have the following syntax:

<i>type-qualifier:</i>	<b>const</b>
------------------------	--------------

The **const** qualifier explicitly declares a data object as a data item that cannot be changed any more. You cannot use const data objects in expressions requiring a modifiable lvalue. For example, a const data object cannot appear on the lefthand side of an assignment statement. Although a const variable cannot be modified, it can be initialized following the same rules as the initialization of any other object.

In TM57 C compiler, global constant variable will be addressed to the memory area of program ROM, in order to save RAM memory space. Local constant variable is still addressed to the memory of RAM.

Example:

```
const char _szmydata[] = "hello";
const unsigned char _szdata[]={0x10,0x20,0x30,0x40,0x50,0x60};
const int _idata=0x55AA;
```

**Note:** const pointer declaration is not supported.

## Declarators

A **declarator** designates a data object or a function with the scope, storage duration, and type indicated by the declaration. Each declarator contains exactly one identifier; it is this identifier that is declared. In the declaration “**T D1**,” **D1** is simply an identifier, then the type of the identifier is **T**.

In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can also perform initialization in a declarator. The following table illustrates some examples of declarators:

Example	Description
int year	year is an <b>int</b> data object.
int *node	node is a pointer to an <b>int</b> data object.
int name[12]	name is an array of 12 <b>int</b> elements.
int *move()	move is a function returning a pointer to an int
extern const int sys_clock	sys_clock is a constant integer and external linkage

TM57 C compiler is for 8-bit single chip, and the definition and length of supported data types are list as below.

Data type	Size (bytes)	Scope
char	1	-128 ~ 127
unsigned char	1	0 ~ 255
short	2	-32768 ~ 32767
unsigned short	2	0 ~ 65535
int	2	-32768 ~ 32767
unsigned int	2	0 ~ 65535
long	4	-2147483648 ~ 2147483647
unsigned long	4	0 ~ 4294967295
pointer	1	0 ~ 255
bit	1	0 ~ 1

## Pointer Declarators

A pointer type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type except to a reference. A pointer is classified as a scalar type, means that it can hold only one value at a time. Pointer declarators have the form:

```
pointer:                * type-qualifier-listopt  
                        * type-qualifier-listopt pointer
```

When you use pointers in an assignment operation, you must make sure that the types of the pointers in the operation are compatible. It means two pointers types with the same type qualifiers are compatible if they point to objects of compatible types.

Example:

```
int section[80];  
int *student = section;
```

Some common uses for pointers are:

- To access elements of an array or members of a structure.
- To access an array of characters as a string.
- To pass the address of a variable to a function. By referencing a variable through its address, a function can change the contents of that variable.

## Array Declarators

An array is a collection of objects which have the same data type. Individual objects in an array, called elements, are accessed by their position in the array. The subscripting operator ([]) provides the mechanics for creating an index to array elements.

```
array:                Type identifier [constant-expression]
```

The initializer for an array is a comma-separated list of constant expressions enclosed in braces ({ }). The initializer is preceded by an equal sign (=).

Example:

```
int number[3] = { 5, 7, 2 };
```

The array number contains the following values: number[0] is 5, number[1] is 7 and number[2] is 2.

Example:

```
int item[ ] = { 1, 2, 3, 4, 5 };
```

The TM57 C compiler gives item of the five initialized elements, since no size is specified and there are five initializers.

Initializing a string constant places the null character (\0) at the end of the string if there is room or if the array dimensions are not specified. The following definitions show character array initializations:

```
static char name1[ ] = { 'J', 'o', 'y' };
static char name2[ ] = { "Joy" };
static char name3[4] = "Joy";
static char name4[4]= "Joys";//Error,because string contains \0 at the end ,so you can put a
//maximum of 3 characters.
```

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
{
    { 1, 2},
    { 3, 4},
    { 5, 6}
};
```

Element	Value	Element	Value	Element	Value
matrix[0][0]	1	matrix[1][0]	3	matrix[2][0]	5
matrix[0][1]	2	matrix[1][1]	4	matrix[2][1]	6
matrix[0][2]	0	matrix[1][2]	0	matrix[2][2]	0
matrix[0][3]	0	matrix[1][3]	0	matrix[2][3]	0

The following rules apply to array declarations:

- If the array is a fixed length array, the expression is enclosed in square brackets. If it exists, it must be an integral constant expression whose value is greater than zero.
- When several “array of” specifications are adjacent, it means a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays can be missing only for the first member of the sequence.
- The absence of the first array dimension is allowed if the array is external and the actual definition (which allocates storage) is given elsewhere, or if the declarator is followed by initialization. In the latter case, the size is calculated from the number of elements supplied.
- The array type is “fixed length array” if the size expression is an integer constant expression, and the element type has a fixed size.
- In order for two array types to be compatible, their element types must be compatible. In addition, if both of their size specifications are present and are integer constant expressions, they must have the same value.



## Function Declarators and Prototypes

When a function is invoked for which a function prototype is in scope, an attempt is made to convert each actual parameter to the type of the corresponding formal parameter specified in the function prototype, superseding the default argument promotions. The number of parameters appearing in the parameter list at the point of call must agree in number with those in the function prototype.

The following are two examples of function prototypes:

```
long foo(int *first, int second);
int *fip(int a, long l, int b);
```

It is recommended to use *the ANSI C function prototype*. In traditional C, however, the implementation of prototypes was incomplete. In one case, a significant difference still exists between the ANSI C and the traditional C implementations of prototypes.

ANSI C	Traditional C
void adjust_xy (short x, short y) {.....}	void adjust_xy (x, y) short x; short y; {.....}

**Note:** Recursive function is not supported.

The return value of C function is saved in F-Plane RAM, i.e. saved in the operation register: op2. A function prototype should be declared before the function can be called. Besides, the same with const variable, it allows user to define function in TABLE ROM (program memory) address. Below example shows function f1 specifies ROM address 0x0100, but function f2 does not specify ROM address. **Please note, ROM address can only be specified in definition section of function.**

```
char f1(char, char);           // Cannot specify address in function declaration
char f2(char, char);

void main()
{
    char a = 5;
    char b = 2;
    char c = 0;
    c = f1(a, b);
    c = f2(a, b);
}
char f1(char i, char j)@0x0100 //In function declaration specifies ROM address 0x0100,
{
    if(i < j)
        return (j - i);
    else
        return (i - j);
}
char f2(char i, char j)
{
    return (i + j);
}
```

## asm Declarators

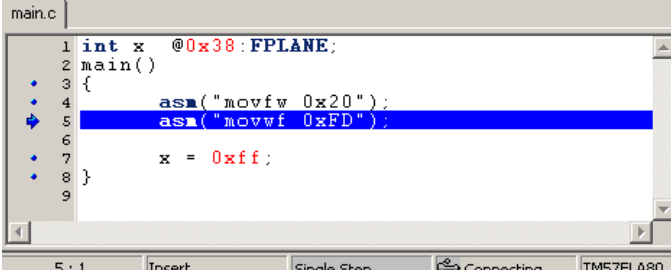
The keyword **asm** stands for assembly code. In this implementation, the TM57 C compiler recognizes and ignores the keyword asm in a declaration. The syntax is

```
asm (<string literal>[, optional parameters]) ;
or
__asm__ (<string literal>[, optional parameters]) ;
```

The asm statement may be used inside a function (but don't use it on global file level). An inline assembly statement is a primary expression, so it may also be used as part of an expression. The contents of the string literal are preprocessed by the compiler and inserted into the generated assembly output, so that it can be further processed by the backend and especially the optimizer. For this reason, the compiler does only allow regular TM57XX opcodes to be used with the inline assembly.

The built-in inline assembly is not a replacement for the full-blown macro assembly which comes with the compiler. For example, the instruction parameter address in embedded assembly code, when exceeds bank0 area of its IC, C Compiler will automatically insert bank instruction within instructions. In TM57FLA80 as example, in instruction `asm("movwf 0xFD")`, parameter address 0xFD exceeds 0x80 (the size of bank0), therefore C Compiler will convert `asm("movwf 0xFD")` into 3 instructions as below

```
BSF 0x03, 5      // Change to bank1
MOVWF 0x7D       // 0xFD - 0x80 = 0x7D of Bank1
..
BCF 0x03, 5      // Change to bank0
```

Original Instruction	After Compiling
	<ul style="list-style-type: none"> <li>• 000000: 3001 GOTO 0x01</li> <li>• 000001: 200B CALL 0x0B</li> <li>• 000002: 11C3 BCF 0x03, 7</li> <li>• 000003: 0820 MOVFW 0x20</li> <li>• 000004: 1343 BSF 0x03, 5</li> <li>• 000005: 00FD MOVWF 0x7D</li> <li>• 000006: 19FF MOVLW 0xFF</li> <li>• 000007: 1143 BCF 0x03, 5</li> <li>• 000008: 00B8 MOVWF 0x38</li> <li>• 000009: 01B9 CLRF 0x39</li> <li>• 00000A: 0040 RET</li> <li>• 00000B: 1910 MOVLW 0x10</li> <li>• 00000C: 1E10 MOVWR 0x10</li> <li>• 00000D: 0040 RET</li> </ul>

In some actual implementation, if it needs to control bank switching manually, not controlled by C Compiler, user can use assembly pseudo code `.fixcode` to block RAM Bank auto function.

Original Instruction	After Compiling
<pre> main.c 1 int x @0x38:FPLANE; 2 main() 3 { 4     asm(".fixcode +"); 5     asm("movlw 0x20"); 6     asm("bsf 0x3,5"); 7     asm("movwf 0x7D"); 8     asm(".fixcode -"); 9     x = 0xff; 10 } 11 </pre>	<pre> 000000: 3001 GOTO 0x01 000001: 200B CALL 0x0B 000002: 11C3 BCF 0x03, 7 000003: 0820 MOVFW 0x20 000004: 1343 BSF 0x03, 5 000005: 00FD MOVWF 0x7D 000006: 19FF MOVLW 0xFF 000007: 1143 BCF 0x03, 5 000008: 00B8 MOVWF 0x38 000009: 01B9 CLRF 0x39 00000A: 0040 RET 00000B: 1910 MOVLW 0x10 00000C: 1E10 MOVWR 0x10 00000D: 0040 RET </pre>

**Note:** Inline assembly statements are subject to all optimizations done by the compiler. There is currently no way to protect an inline assembly statement from being moved or removed completely by the optimizer. If in doubt, check the generated assembly output, or disable optimizations.

The string literal may contain format specifiers from the following list. For each format specifier, an argument is expected which is inserted instead of the format specifier before passing the assembly code line to the backend.

Format specifier	Description
%b	Numerical 8-bit value
%w	Numerical 16-bit value
%l	Numerical 32-bit value
%v	Assembly name of a (global) variable or function
%o	Stack offset of a (local) variable
%%	The % sign itself Specifically for bsf, bcf, btfsc, btfss instructions, if it's the above instruction, it will determine which bit the variable is in and add it to the instruction. ※ Only applicable to bit type variables
%n	Ex: bit bb;//Assume 0x20, bit 0 asm("bsf %n",bb); //  Result: bsf 0x20,0

Using these format specifiers, you can access C #defines, variables or similar stuff from the inline assembly. For example, to load the value of a C #define into the W accumulator, one would use

```

#define OFFS 23
__asm__ ("MOVLW %b", OFFS);

```

Or,

to access a struct member of a static variable:

```
#define offsetof(type, member) (unsigned) (&((type*) 0)->member)
typedef struct {
    unsigned char x;
    unsigned char y;
    unsigned char color;
} pixel_t;
static pixel_t pixel;
__asm__ ("MOVLW %v+%b", pixel, offsetof(pixel_t, color));
or
__asm__ ("MOVLW %v+%b", pixel, &pixel.color); // no need to define #define offsetof
```

**Note:** Do not embed the assembly labels that are used as names of global variables or functions into your asm statements.

Code like this:

```
int foo;
int bar () { return 1; }
__asm__ ("MOVFW _foo"); /* DON'T DO THAT! */
...
__asm__ ("call _bar"); /* DON'T DO THAT EITHER! */
```

The original global variable and function name in the .s file after C compiler compiles will be added with '-' as prefix in the name of variable or function. As above example, the global variable declaration: `int foo=0;`, in the corresponding output .s file will become as below:

```
MOVLW $00 ;1,n=1,2
MOVWF _foo+0
MOVWF _foo+1
```

Variable name `foo` is converted into `_foo`; please note that for local variable and parameter name do not follow this conversion rule. Therefore, avoid spelling and maintaining problem in variable, parameter or function name in asm expression. In C program, when using inline asm expression, it is suggested to use **format symbol** to replace variable name, the usage such as `__asm__ ("MOVFW %v", foo);`. Similarly, when function without input parameter is called, such as `__asm__ ("call _bar");`, it is suggested to use `bar();` to replace directly.

Example1: Coding as full assembly code in C function:

```
char *strcpy(char *tar, char *src)
{
    // Return tar value from op2 (0x24)
    asm("movfw %o", tar);
    asm("movwf op2"); // return tar pointer in op2 address (0x24)
    asm("_strcpy_LOOP:"); // generate label name

    // Read from source
    asm("movfw %o", src); // Set offset of LOCAL name src
```

```
asm("call runtime_Ind_Read"); // call indirect read
asm("movwf op3");           // op3 to write to target

// Save to target
asm("movfw %o",tar);        // Set offset of LOCAL name tar (strcpy_LOCAL+1)
asm("call runtime_Ind_Write"); // call indirect write

// Check end
asm("testz op3");
asm("btfsc STATUS, ZERO_FLAG");
asm("ret");

// Next
asm("incf %o,1",src);
asm("incf %o,1",tar);
asm("goto _strcpy_LOOP");
}
```

Example2: in \*.C file call ASM

```
Main(void)
{
    Asm("CALL asmLabelDelay"); //in *.ASM file need ".export asmLabelDelay"
}
```

In [Example 1](#) of appendix, listed series related expression function implementation using inline asm.

## Restrictions on Declarators

Not all the possibilities allowed by the syntax of declarators are permitted. The following restrictions are applied:

- Functions cannot return arrays or functions although they can return pointers. We suggest using the following statements for example to return array in a function.

```
int* subFoo(int x);
void main(void)
{
    .....
    int C[10],*p;
    p = subFoo(value1);
    for(i=0; i<10; ++i)
    {
        C[i] = *p;
        ++p;
    }
    ....
}
```

```
// subFoo: assign array elements
int* subFoo(int x)
{
    int B[10];
    int i;
    for(i=0; i<10; ++i)
        B[i] = 10;
    return B;
}
```

- No arrays of functions exist although arrays of pointers to functions can exist.
- A structure or union cannot contain a function. The below example of structure declaration is not allowed:

```
struct ERROR_STRUCT
{
    int i;
    int y;

    int foo(int var)
    {
        ....
        return var;
    };
};
```

## Typedef

A typedef declaration lets you define your own identifiers that can be used in place of type specifiers such as int, long, struct, and pointer. Declarations with the storage class specifier **typedef** do not reserve storage. The names you define using **typedef** are not new data types, but synonyms for the data types or combinations of data types they represent.

The following statements declare TLENGTH as a synonym for int and then use this **typedef** to declare length, width, and height as integer variables:

```
typedef int TLENGTH;
TLENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
Int length, width, height;
```

Similarly, typedef can be used to define object type such as structure, union. For example:

```
typedef struct {
    int scruples;
    int drams;
    int grains;
} WEIGHT;
```

The structure WEIGHT can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

## Initialization

A declaration of an object or of an array of unknown size can specify an initial value for the identifier being declared. The initializer is preceded by ‘=’ and consists of an expression or a list of values enclosed in nested braces:

<i>initializer:</i>	<i>assignment-expression</i> <i>{initializer-list}</i>
<i>initializer-list:</i>	<i>initializer</i> <i>initializer-list , initializer</i>

## Initialization of Aggregates

In TM57 C, objects that are **struct** or **union** types can be initialized, even if they have automatic storage duration. **union** are initialized using the type of the first named element in their declaration. When the declared variable is a **struct** or **array**, the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order.

Example:

```
union dc_u {  
    int d;  
    char *cptr;  
};  
union dc_u dc0 = { 4 };
```

A final abbreviation allows a char array to be initialized by a string literal. In this case, successive characters of the string literal initialize the members of the array.

```
char msg[] = "Syntax error on line %s\n";
```



## 4. Expressions and Operators

This chapter introduces the various expressions and operators available in C. The sections describing expressions and operators are presented roughly in order of precedence

### Operator Precedence and Associativity Rules in C

Operators in C have rules of precedence and associativity that determine how operands group with operators and expressions are evaluated. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first.

The following table lists the C language operators in order of precedence and shows the direction of associativity for each operator (L-R means left-to-right, R-L means right-to-left):

Tokens (From High to Low Priority)	Operators	Class	Associativity
Identifiers, constants, string literal, parenthesized expression	Primary expression	Primary	
() [] -> .	Function calls, subscripting, in direct selection, direct selection	Postfix	L-R
++ --	Increment, decrement (postfix)	Postfix	L-R
++ --	increment, decrement (prefix)	Prefix	R-L
! ~ + - & sizeof *	Logical and bitwise NOT, unary plus and minus, address, size, indirection	Unary	R-L
( type )	Cast	Unary	R-L
* / %	Multiplicative	Binary	L-R
+ -	Additive	Binary	L-R
<< >>	Left shift, right shift	Binary	L-R
< <= > >=	Relational comparisons	Binary	L-R
== !=	Equality comparisons	Binary	L-R
&	Bitwise and	Binary	L-R
^	Bitwise exclusive or	Binary	L-R
	Bitwise inclusive or	Binary	L-R
&&	Logical and	Binary	L-R
	Logical or	Binary	L-R
? :	conditional	Ternary	R-L
= += -= *= /= %= ^= &=  =	Assignment	Binary	R-L
<<= >>=			
,	Comma	Binary	L-R

## Primary Expressions

The following are all considered as “primary expressions:”

Identifiers	An identifier refers to an object is an lvalue. An identifier refers to a function is a function designator.
Constants	A constant’s type is determined by its form and value
String literals	A string literal’s type is “array of char,” subject to Modification.
Parenthesized expressions	A parenthesized expression’s type and value are identical to those of the unparenthesized expression. The presence of parentheses does not affect whether the expression is an lvalue, rvalue, or function designator.

## Postfix Expressions

Postfix operators are operators that appear after their operands. A postfix expression is a primary expression, or a primary expression that contains a postfix operator. The following summarizes the available postfix operators:

Operator Function	Usage	Example
member selection	object . member	Table.Color
member selection	pointer -> member	Table->Color
subscripting	pointer [ expr ]	ArrayOne[2]
function call	expr ( expr_list )	Foo(a,b,c)
value construction	type ( expr_list )	Long(intOne)
postfix increment	lvalue ++	Lindex--
postfix decrement	lvalue --	Lindex++

## Array Subscripting Operator

A postfix expression followed by an expression in [ ] (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a subscript. The first element of an array has the subscript zero. The expression code[10] refers to the 11th element of the array code.

In a multidimensional array, you can refer each element (in the order of increasing storage locations) by incrementing the right-most subscript most frequently. For example, the following statement gives the value 100 to each element in the array code[4][3][6]:

```

Int first, second, third;

for (first = 0; first < 4; ++first)
{
    for (second = 0; second < 3; ++second)
    {
        for (third = 0; third < 6; ++third)
        {
            code[first][second][third] = 100;
        }
    }
}

```

## Structure and Union References

A postfix expression followed by a dot followed by an identifier denotes a structure or union reference. The syntax is as follows:

```
postfix-expression.identifier
```

The *postfix expression* must be a **structure** or a **union**, and the *identifier* must name a member of the structure or union. The value is the value of the named member of the structure or union, and is an lvalue if the first expression is an lvalue. The result has the type of the indicated member and the qualifiers of the structure or union.

## Indirect structure and Union References

The -> (arrow) operator is used to access structure or union members using a pointer. A postfix-expression followed by an arrow (built from `-` and `>`) followed by an identifier is an indirect structure or union reference. The syntax is as follows:

```
postfix-expression-> identifier
```

The *postfix expression* must be a pointer to a **structure** or a **union**, and the *identifier* must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the postfix expression points. The result has the type of the selected member, and the qualifiers of the structure or union to which the postfix expression points. Thus, the expression `E1->MOS` is the same as `(*E1).MOS`.

## Postfix ++ and Postfix --

The syntax of postfix ++ and postfix -- is as follows:

```
postfix-expression ++  
postfix-expression --
```

When postfix ++ is applied to a modifiable lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented by 1 (one). The type of the result is the same as the type of the lvalue expression. The result is not an lvalue.

When postfix -- is applied to a modifiable lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented by 1 (one). The type of the result is the same as the type of the lvalue expression. The result is not an lvalue.

For both postfix ++ and postfix -- operators, updating the stored value of the operand may be delayed until the next sequence point.

## Unary Operators

A unary expression contains one operand and a unary operator. All unary operators have the same precedence and have *right- to-left associativity*. A unary expression is therefore a postfix expression. As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most unary expressions. The following table summarizes the operators for unary expressions:

Operator Function	Usage
size of object in bytes	<b>sizeof</b> (expr)
size of type in bytes	<b>sizeof</b> type
prefix increment	<b>++</b> lvalue
prefix decrement	<b>--</b> lvalue
complement	<b>~</b> expr
not	<b>!</b> expr
unary minus	<b>-</b> expr
unary plus	<b>+</b> expr
address of	<b>&amp;</b> lvalue
indirection or dereference	<b>*</b> expr

## Address-of and Indirection Operators

The unary **\*** operator means "indirection"; the cast expression must be a pointer, and the result is either an lvalue referring to the object to which the expression points, or a function designator. The operand of the unary **&** operator can be either a function designator or an lvalue that designates an object. If it is an lvalue, the object it designates cannot be a bit field, and it cannot be declared with the storage class register. The result of the unary **&** operator is a pointer to the object or function referred to by the lvalue or function designator.

## Unary + and Unary – Operators

The result of the unary **-** operator is the negative of its operand. The integral promotions are performed on the operand, and the result has the promoted type and the value of the negative of the operand.

The **+** (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type or pointer type. The result is not an lvalue.

## Logical Negation ! and Bitwise Negation ~ Operators

The ! (logical negation) operator determines whether the operand evaluates to 0 (false) or nonzero (true). The result of the logical negation operator ! is 1 if the value of its operand is zero, and 0 if the value of its operand is nonzero.

The following two expressions are equivalent:

```
!right;  
right == 0;
```

The ~ (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.

Example:

Suppose x represents the decimal value 5. The 8-bit binary representation of x is: 00000101. The expression ~x yields the following result: 11111010

## Prefix ++ and Prefix -- Operators

The prefix operators ++ and -- increment and decrement their operands. Their syntax is as follows:

```
++unary-expression  
--unary-expression
```

The object referred to by the modifiable lvalue operand of prefix ++ is incremented. The expression value is the new value of the operand but is not an lvalue. The expression ++x is equivalent to x += 1. The prefix -- decrements its lvalue operand in the same way that prefix ++ increments it.

## Sizeof Unary Operator

The sizeof operator yields the size in bytes of its operand, which can be an expression or the parenthesized name of a type.

In TM57 C compiler, the size of a char is 1 (one), the size of an int is 2, and the size of a long is 4. Its major use is in communication with routines such as storage allocators and I/O systems. The syntax of the sizeof operator is as follows:

```
sizeof unary-expression  
sizeof (type-name)
```

The sizeof operator may not be applied to:

- A bit field
- A function type
- An undefined structure or class
- An incomplete type (such as void)

## Multiplicative Operators

The multiplicative operators `*`, `/`, and `%` group from left to right. The usual arithmetic conversions are performed. The following is the syntax for the multiplicative operators:

```
multiplicative expression:      cast-expression  
                                multiplicative-expression * cast-expression  
                                multiplicative-expression / cast-expression  
                                multiplicative-expression % cast-expression
```

Operands of `*` and `/` must have arithmetic type. Operands of `%` must have integral type. The binary `*` operator indicates multiplication, and its result is the product of the operands. The binary `/` operator indicates division of the first operator (dividend) by the second (divisor). Integral division results in the integer quotient whose magnitude is less than or equal to that of the true quotient, and with the same sign.

The binary `%` operator yields the remainder from the division of the first expression (dividend) by the second (divisor). The operands must be integral.

## Additive Operators

The additive operators `+` and `-` associate from left to right. The usual arithmetic conversions are performed. The syntax for the additive operators is as follows:

```
additive-expression:      multiplicative-expression  
                           additive-expression + multiplicative-expression  
                           additive-expression - multiplicative-expression
```

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount of the integral value treated as a subscript. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. The compiler does not provide boundary checking on the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];  
int *ptr;  
ptr = array + 2;
```

## Shift Operators

The bitwise shift operators move the bit values of a binary object. The bitwise shift operators << and >> associate from left to right. Each operand must be an integral type. The integral promotions are performed on each operand. The syntax is as follows:

```

shift-expression:      additive-expression
                      shift-expression << additive-expression
                      shift-expression >> additive-expression
    
```

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

For example, if left\_op has the value 4019, the bit pattern (in 16-bit format) of left\_op is:

0000111110110011

The expression left\_op << 3 yields:

0111110110011000

## Relational Operators (< > <= >=)

The relational operators compare two operands and determine the validity of a relationship. The type of the result is int and has the values 1 if the specified relationship is true, and 0 if false. The result is not an lvalue.

Operator	Usage
<	Indicates whether the value of the left operand is less than the value of the right operand.
>	Indicates whether the value of the left operand is greater than the value of the right operand.
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined. If two pointers refer to the same object, they are considered equal.

## Equality Operators (== !=)

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. The type of the result is int and has the values 1 if the specified relationship is true, and 0 if false.

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

## Logical AND Operators (&&), Logical OR Operators (||)

The && (logical AND) operator indicates whether both operands are true. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is int. Both operands must have an arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

Expression	Result
1 && 0	0
1 && 6	1
0 && 0	0

The || (logical OR) operator indicates whether either operand is true. If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is int. Both operands must have an arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

Expression	Result
1    0	1
1    6	1
0    0	0

## Conditional Operator

A conditional expression is a compound expression that contains a condition implicitly converted to bool (operand<sub>1</sub>), an expression to be evaluated if the condition evaluates to true (operand<sub>2</sub>), and an expression to be evaluated if the condition has the value false (operand<sub>3</sub>).

```
( operand1 ? operand2 : operand3 )
```

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is true, the second operand is evaluated.
- If the value is false, the third operand is evaluated.



The result is the value of the second or third operand.

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

$$x = (y > z) ? y : z;$$

The following is an equivalent statement:

```
if (y > z)
    x = y ;
else
    x = z;
```

## 5. Statements

A statement is a complete instruction to the computer, it is the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence.

### Expression Statements

Usually expression statements are expressions evaluated for their side effects, such as assignments or function calls. A special case is the null statement, which consists of only a semicolon.

Examples of Expressions:

```
printf("Account Number: \n");      /* call to the printf */
marks = dollars * exch_rate;       /* assignment to marks */
(difference < 0) ? ++losses : ++gain; /* conditional increment */
```

### Block Statement

A block statement, or compound statement, lets you group any number of data definitions, declarations, and statements into one statement. Declarations within compound statements have block scope. If any of the identifiers in the declaration list were previously declared, the outer declaration is hidden for the duration of the block, after which it resumes its force.

### Selection Statements

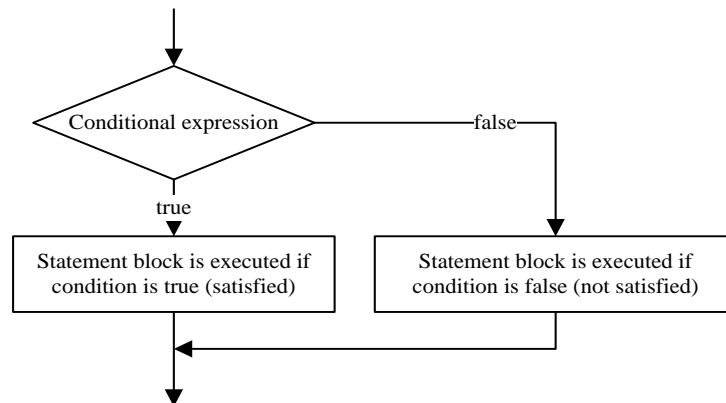
Selection statements include if and switch statements. Selection statements choose one of a set of statements to execute based on the evaluation of the expression. The expression is referred to as the controlling expression.

```
selection-statement:      if (expression) statement
                           if (expression) statement else statement
                           switch (expression) statement
```

## if Statement

An if statement is a selection statement that allows more than one possible flow of control. You can optionally specify an else clause on the if statement. If the test expression evaluates to a zero value and an else clause exists, the statement associated with the else clause runs. If the test expression evaluates to a non-zero value, the statement following the expression runs and the else clause is ignored.

When if statements are nested and else clauses are present, a given else is associated with the closest preceding if statement within the same block.



## switch Statement

A switch statement is a selection statement that lets you transfer control to different statements within the switch body depending on the value of the switch expression. The switch expression must evaluate to an integral or enumeration value. The body of the switch statement contains case clauses that consist of:

- A case label
- An optional **default** label
- A **case** expression
- A list of statements.

If the switch expression matches a case expression, the statements following the case expression are processed until a break statement is encountered or the end of the switch body is reached.

### Example:

```
char key;
.....
switch (key)
{
    case '+':
        add();
        break;
    case '-':
        subtract();
        break;
    case '*':
        multiply();
        break;
    case '/':
        divide();
        break;
    default:
        break;
}
```

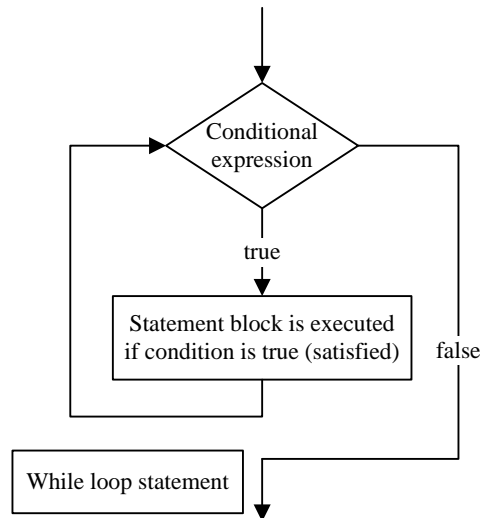
## Iteration Statements

Iteration statements execute the attached statement (called the body) repeatedly until the controlling expression evaluates to zero. In the **for** statement, the second expression is the controlling expression. The format is as follows:

<i>iteration-statement:</i>	<b>while</b> ( <i>expression</i> ) <i>statement</i> <b>do</b> <i>statement</i> <b>while</b> ( <i>expression</i> ) ; <b>for</b> ([ <i>expression</i> ] ; [ <i>expression</i> ] ; [ <i>expression</i> ]) <i>statement</i>
-----------------------------	---

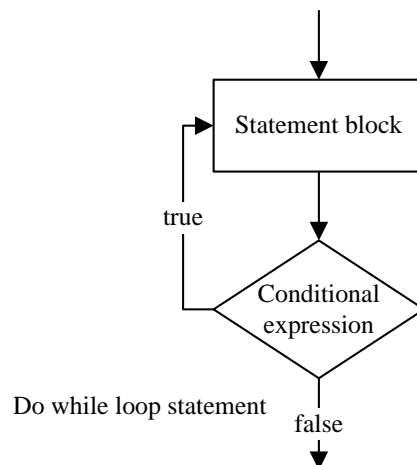
## while Statement

A while statement repeatedly runs the body of a loop until the controlling expression evaluates to a zero value (0). A break, return, or goto statement can cause a while statement to end, even when the condition does not evaluate to 0.



## do Statement

Unlike the **while** statement, the controlling expression of a **do-while** statement is evaluated after each execution of the body. Because of the order of processing, the statement is run at least once.



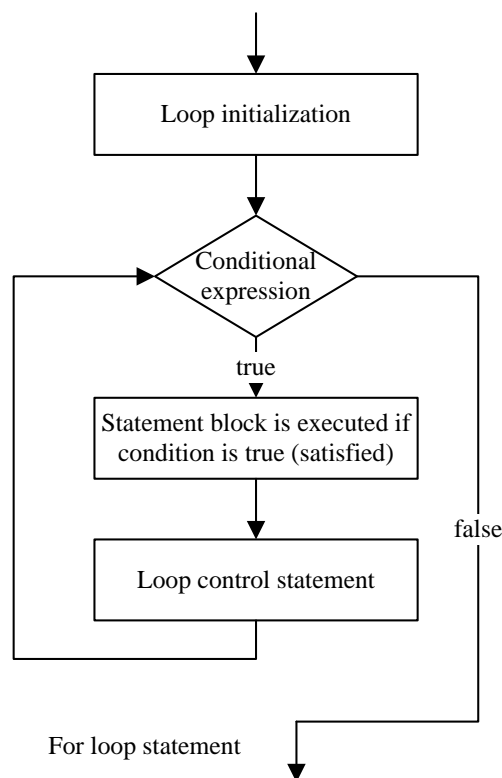
## for Statement

A for statement lets you do the following:

- Evaluate an expression before the first iteration of the statement (initialization)
- Specify an expression to determine whether or not the statement should be processed (the condition)
- Evaluate an expression after each iteration of the statement (often used to increment for each iteration)
- Repeatedly process the statement if the controlling part does not evaluate to 0.

A **break**, **return**, or **goto** statement can cause a for statement to end, even when the second expression does not evaluate to 0. If you omit expression<sub>2</sub>, you must use a **break**, **return**, or **goto** statement to end the for statement.

The first expression specifies initialization for the loop. The second expression is the controlling expression, which is evaluated before each iteration. The third expression often specifies incrementation. It is evaluated after each iteration.



## Jump Statements

Jump statements cause unconditional transfer of control. The syntax is as follows:

```
jump-statement:      goto identifier;  
                      continue;  
                      break;  
                      return [expression]
```

### goto Statement

A **goto** statement causes your program to be transferred unconditionally to the statement associated with the label specified on the goto statement.

```
goto identifier;
```

The identifier must name a label located in the enclosing function. If the label has not yet appeared, it is implicitly declared.

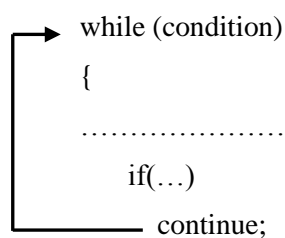
Because the goto statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a break statement, a continue statement, or a function call can eliminate the need for a goto statement.

Example:

```
int i=0;  
Label:  
if( i < 10 )  
    goto Label;
```

### continue Statement


The continue statement can appear only in the body of an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is, to the end of the loop.



## break Statement

A **break** statement lets you end an iterative (**do**, **for**, or **while**) statement or a **switch** statement and exit from it at any point other than the logical end. A break may only appear on one of these statements. In an iterative statement, the break statement ends the loop and moves control to the next statement outside the loop.

```
while (condition)
{
.....
.....
    break;
.....
.....
}
```



## return Statement

A **return** statement ends the processing of the current function and returns control to the caller of the function. The **return** statement cannot have an expression if the type of the current function is void. If the end of a function is reached before the execution of an explicit return, an implicit return (with no expression) is executed.

## Labeled Statement

There are three kinds of labels: **identifier**, **case**, and **default**. Labeled statements have the following syntax:

<i>Labeled-statement:</i>	<b>identifier</b> : statement <b>case</b> constant-expression : statement <b>default</b> : statement
---------------------------	--

Any statement can have a label attached as a simple identifier. The scope of such a label is the current function. Thus, labels must be unique within a function.



## Interrupt

In contrast to general-purpose computers, microcontrollers used in embedded systems often seek to optimize interrupt latency over instruction throughput. Issues include both reducing the latency, and making it be more predictable (to support real-time control). tenx microcontrollers provide a real time (predictable, though not necessarily fast) response to events in the embedded system they are controlling. The interruption implementation will produce a directly change to the timers and the counters. In TM57 C compiler, the interrupt format is:

Interrupt service routine :	<code>void interrupt &lt;function(void)&gt; @ &lt;interrupt vector address&gt;</code>
-----------------------------	---

The <interrupt vector address> means if there are many interrupt vectors in MCU, for example in TM57FLA80 chip, there has following interrupts: Pin interrupts, Timer interrupt, PWM/CMP/ADC and Wakeup Timer/USI interrupt. We give the sequence 0x01(TMR0) , 0x02(TMR1) , 0x03(TMR2) , 0x04(PWM0) , 0x05(WKT) , 0x06(XINTA) , 0x07(XINTB) , 0x08(UART) , 0x09(SPI) to match IC interrupt vector.

For TM57FLA80, the declaration of interrupt as below:

```
void interrupt TMR0_Intrerrupt(void) @ 0x01 {....}
void interrupt TMR1_Intrerrupt(void) @ 0x02 {....}
void interrupt TMR2_Intrerrupt(void) @ 0x03 {....}
void interrupt PWM0_Intrerrupt(void) @ 0x04 {....}
void interrupt WKT_Intrerrupt(void) @ 0x05 {....}
void interrupt XINTA_Intrerrupt(void) @ 0x06 {....}
void interrupt XINTB_Intrerrupt(void) @ 0x07 {....}
void interrupt UART_Intrerrupt(void) @ 0x08 {....}
void interrupt SPI_Intrerrupt(void) @ 0x09 {....}
```

Interrupt Service Routine must not have any parameter; otherwise the compiler will generate error.

Example:

```
//RPLANE
char OPTION @0x02:RPLANE;

//FPLANE
char INTE1 @0x08:FPLANE;
char INTF1 @0x09:FPLANE;
char TIMER0 @0x01:FPLANE;
char TM1CTRL @0x0D:FPLANE;

int temp_b=0;
unsigned char Timer_Buf=0xF0;
main()
```

```
{
    char c=0;
    int a=0;

    TIMER0=Timer_Buf;
    OPTION=0;
    INTE1=0x10; // Enable Timer0 Interrupt

    for(c=0;;c++)
        ++a;
}

void interrupt TM0_Interrupt(void)@0x01
{
    TIMER0=Timer_Buf;
    INTF1=0;
    ++temp_b;
}
```

In the above example, after setting related setting for TMR0 Interrupt, then use an infinite loop to let TMR0 overflow. It will execute interrupt service.

**Note:** Different from the usage of general function, the interruption implementation will execute automatically when the interrupt condition has been matched.

In actual application, after interrupt is triggered and before executing interrupt service routine, operation related Data RAM must be stored in advance (especially when this interrupt service routine will change the operation of related data). In this way, after the interrupt service routine, the operation data can be restored. After interrupt is executed, control process can still execute correctly and continuously, means to ensure that the operation result will not be affected by the interrupt execution and cause errors.

Please note that `ISR_SaveData` and `ISR_RestoreData` assembly subroutines have been implemented in all TM57 series single chip runtime library, user can call the two subroutines mentioned above according to the actual application. Besides, to avoid the execution time of the two subroutine mentioned above is too long, we provided `ISR_SaveData_5`, `ISR_RestoreData_5`, and `ISR_SaveData_10`, `ISR_RestoreData_10` for another choice. Where `ISR_SaveData_10` and `ISR_RestoreData_10` are used for R-Plane data register. The differences among `ISR_SaveData`, `ISR_SaveData_5` and `ISR_SaveData_10` is mainly in the different type of operation data saved (later will be explained in detail). According to the actual computational complexity, user can use interrupt protection subroutine to save Data RAM more efficiently.

In order not to affect current saved memory data content, when interrupt occurs, `ISR_SaveData`, `ISR_SaveData_5`, and `ISR_SaveData_10` will store the operation data beforehand in the end of the memory address. According to the whole project and function routine which are included, linker will calculate all operation register or stack pointer register used in the program. And according to this amount, calculated from the last address of the address space needed, stored orderly the operation register, status register, op1~op4 or stkptr data contents, and stored in the end of the memory.

According to the single chip characteristics, memory location of data register can be divided into three types: (1) R-Plane, (2) F-Plane Bank0, (3) F-Plane Bank1. According to the TM57 series chip features, the detail implementation will be described below.

### (1) R-Plane

Status value and expression data is stored in the end of the R-Plane RAM memory address. For example, in TM57FLA80, assume the expression data to be stored include: working register, status register, and op1, then the data stored address is as follows:

Storage Address	
Working register	0xFA
Status register	0xFB
op1	0xFC~0xFF

TM57 series chips which store operation data in R-Plane are as below, with the common features as follows

- R-Plane memory is readable and writable

R-Plane final address	
TM57FE80	0xFF
TM57FLA80	0xFF
TM57ML40	0xFF

### (2) F-Plane Bank 0

Status value and expression data is stored in the end of the F-Plane Bank0 memory address. For example, in TM57PE11, assume the expression data to be stored include: working register, status register, and op1, then the stored address is as follows:

Storage Address	
Working register	0x4A
Status register	0x4B
op1	0x4C~0x4F

TM57 series chips which store operation data in F-Plane Bank0 are as below, with the common features as follows

- F-Plane contains only one Bank
- R-Plane only allows write but not read

F-Plane bank0 final address	
TM57ME20	0x7F
TM57P11	0x4F
TM57P12	0x4F
TM57PA10	0x5F
TM57PA10A	0x5F
TM57PE10	0x4F
TM57PE11	0x4F
TM57PE11A	0x4F
TM57PE12	0x4F
TM57PE12A	0x4F
TM57RE12	0x4F

### (3) F-Plane Bank 1

Status value and expression data is stored in the end of the F-Plane Bank1 memory address. For example, in TM57PA40, assume the expression data to be stored include: working register, status register, and op1, then the data stored address is as follows:

	Storage Address
Working register	0x7A
Status register	0x7B
op1	0x7C~0x7F

TM57 series chips which store operation data in F-Plane Bank1 are as below, with the common features as follows

- F-Plane contain two Banks
- R-Plane only allows write but not read

F-Plane bank1 final address	
TM56FA40	0x7F
TM57FA40	0x7F
TM57PA40	0x7F
TM57PA20	0x7F
TM57PA20A	0x7F
TM57PE40	0x7F
TMU3130	0x7F
TMU3131	0x7F
TMU3132	0x7F

Note: Although the R-Plane memory of TMU3130, TMU3131 and TMU3132 is readable and writable, but it is conflicted with USB read-write address. Therefore, the expression data storage address is changed to F-Plane Bank1.

## ISR\_SaveData, ISR\_RestoreData

Register content saved in `ISR_SaveData`, is the max register content needed in all program computation (not the used register after interrupt is triggered). Therefore, for more efficient usage of the limited memory space, user can design store/restore function according to actual interrupt application, or using `ISR_SaveData_5` or `ISR_SaveData_10` function to store important register content to response the real register usage condition.

Assume only `op1~op2` are used in program when interrupt is triggered. But all project program can use up to `op1~op4`. Executing runtime library `ISR_SaveData()` will store working register, status register and `op1~op4`, user can define function which is similar to `ISR_SaveData()` (or using `ISR_SaveData_5` or `ISR_SaveData_10`) to store working register, status register and `op1~op2` to save memory space. When user define store and restore function, it is necessary to remind user to pay attention to the following rule when coding:

- Please follow single chip characteristic and refer to every chip applicable condition to decide the storage memory location, for example, TM57FA40 is supposed to store in Bank1 of F-Plane; while TM57FLA80 is in R-Plane.
- When status and computation data are saved in R-Plane and Bank0 of F-Plane, it is not necessary to consider switching and data copy problem between Bank when store and restore data. The only thing is when it is stored in Bank1 of F-Plane, user needs to pay attention about those problems (because status and computation register is stored in Bank0 of F-Plane as default setting).

Below example is for user's reference about C code of interrupt trigger and interrupt service routine, using `ISR_SaveData()` and `ISR_RestoreData()` to store and restore related Data RAM computational (Note: Calling `ISR_SaveData_5`, `ISR_RestoreData_5`, `ISR_SaveData_10` and `ISR_RestoreData_10` in C code is using the same way).

**C code:** `counter_Function()` is the function which will be executed in interrupt service routine.

```
void predivider_initial(void)
void counter_Function(void);

// Function prototype of asm codes
void ISR_SaveData(void);
void ISR_RestoreData(void);

main()
{
    predivider_initial();
    while(1)
    {
        ..... // do something
    }
}

void interrupt counter_Interrupt(void) @ 0x1c
```

```

{
    ....
    ISR_SaveData();    // Save data
    counter_Function();
    ISR_RestoreData(); // Restore data
    ....
}
// Initial process
void predivider_initial(void)
{
    ..... // initialization process
}
void counter_Function(void)
{
    ..... // do something
}

```

Below is the implementation assembly code of `ISR_SaveData` and `ISR_RestoreData` in runtime library to store and restore computational related Data RAM, which is divided into three types: R-Plane, F-Plane Bank0 and F-Plane Bank1, described with examples below:

- **R-Plane:** Below is TM57FLA80 as example

```

.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

SAVEADDR = __RPLANE_ADDRESS_MAX__ - ( 1 + __OPXSTKPTRSIZE__ +
__STKPTRSAVESIZE__ )
SAVESTKPTRADDR = (__RPLANE_ADDRESS_MAX__ - __STKPTRSAVESIZE__) + 1

.proc _ISR_SaveData

    MOVFW FSR
    MOVWR SAVEADDR
    MOVFW RSR
    MOVWR SAVEADDR+1

    MOVFW op1
    MOVWR SAVEADDR+2

    MOVLW op1+1
    MOVWF FSR
    MOVLW SAVEADDR+3
    MOVWF RSR
    MOVLW (__OPXSTKPTRSIZE__ - 1)+__STKPTRSAVESIZE__

    movwf op1 ; counter

    addwf FSR,1
    addwf RSR,1

```

```

loop:
    decf RSR,1
    decf FSR,1
    movfw R0
    MOVWR R0
    decfsz op1
    goto loop

RET

.endproc

.proc _ISR_RestoreData

    MOVLW SAVEADDR+3
    MOVWF RSR
    MOVLW op1+1
    MOVWF FSR
    MOVLW (__OPXSTKPTRSIZE__ - 1)+__STKPTRSAVESIZE__

    movwf op1 ; counter
    addwf FSR,1
    addwf RSR,1

loop:
    decf FSR,1
    decf RSR,1

    MOVRW R0
    movwf R0

    decfsz op1
    goto loop

    MOVRW SAVEADDR+2
    MOVWF op1

    MOVRW SAVEADDR+1
    MOVWF RSR
    MOVRW SAVEADDR
    MOVWF FSR

RET

.endproc

```

- **F-Plane Bank0:** below is TM57PA10 as example

```

.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

#define _BANK_FLAG 5

```

```
SAVEADDR = __FPLANE_ADDRESS_MAX__ - ( 1 + 1 + __OPXSTKPTRSIZE__ +  
__STKPTRSAVESIZE__ )  
SAVESTKPTRADDR = __FPLANE_ADDRESS_MAX__ - ( __STKPTRSAVESIZE__ ) + 1
```

```
.proc _ISR_SaveData
```

```
MOVWF SAVEADDR  
MOVFW STATUS  
MOVWF SAVEADDR+1
```

```
MOVFW op1  
MOVWF SAVEADDR+2  
MOVFW op1+1  
MOVWF SAVEADDR+3
```

```
MOVFW FSR  
MOVWF SAVEADDR+4
```

```
MOVLW ( __OPXSTKPTRSIZE__ - 2 ) + __STKPTRSAVESIZE__  
movwf op1  
testz op1  
btfsc STATUS, ZERO_FLAG  
ret
```

```
loop:
```

```
movlw op1+1  
movwf FSR  
movfw op1  
addwf FSR  
movfw R0  
movwf op1+1
```

```
movlw SAVEADDR+4  
movwf FSR  
movfw op1  
addwf FSR  
movfw op1+1  
movwf R0
```

```
decfsz op1  
goto loop  
RET
```

```
.endproc
```

```
.proc _ISR_RestoreData
```

```
MOVLW ( __OPXSTKPTRSIZE__ - 2 ) + __STKPTRSAVESIZE__  
movwf op1  
testz op1  
btfsc STATUS, ZERO_FLAG  
goto L0
```



```

loop:
    movlw SAVEADDR+4
    movwf FSR
    movfw op1
    addwf FSR
    movfw R0
    movwf op1+1

    movlw op1+1
    movwf FSR
    movfw op1
    addwf FSR
    movfw op1+1
    movwf R0

    decfsz op1
    goto loop

L0:

    MOVFW SAVEADDR+4
    MOVWF FSR

    MOVFW SAVEADDR+2
    MOVWF op1
    MOVFW SAVEADDR+3
    MOVWF op1+1

    MOVFW SAVEADDR+1
    MOVWF STATUS
    MOVFW SAVEADDR
    RET

.endproc

```

- **F-Plane Bank1:** below is TM57PA40 as example

```

.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

#define _BANK_FLAG 5

SAVEADDR = (__FPLANE_ADDRESS_MAX__-0x80) - ( 2 + __OPXSTKPTRSIZE__ +
__STKPTRSAVESIZE__)

SAVESTKPTRADDR = (__FPLANE_ADDRESS_MAX__ -
( (__BANK1_OFFSET_VALUE__-0x80)+__STKPTRSAVESIZE__))+1

.fixcode +
.proc _ISR_SaveData

    BTFSC STATUS,_BANK_FLAG
    GOTO BANK1
    BSF STATUS,_BANK_FLAG
    MOVWF SAVEADDR

```

```
BCF STATUS,_BANK_FLAG  
goto B0
```

```
BANK1:  
MOVWF SAVEADDR
```

```
B0:  
MOVFW STATUS  
BSF STATUS,_BANK_FLAG  
MOVWF SAVEADDR+1
```

```
MOVFW op1  
MOVWF SAVEADDR+2  
MOVFW op1+1  
MOVWF SAVEADDR+3
```

```
MOVFW FSR  
MOVWF SAVEADDR+4
```

```
MOVLW (__OPXSTKPTRSIZE__ - 2)+__STKPTRSAVESIZE__  
movwf op1  
testz op1  
btfsc STATUS,ZERO_FLAG  
ret
```

```
loop:  
movlw op1+1  
BCF STATUS,_BANK_FLAG  
movwf FSR  
movfw op1  
addwf FSR  
movfw R0  
movwf op1+1
```

```
movlw SAVEADDR+4  
BSF STATUS,_BANK_FLAG  
movwf FSR  
movfw op1  
addwf FSR  
movfw op1+1  
movwf R0
```

```
decfsz op1  
goto loop  
RET
```

```
.endproc
```

```
.proc _ISR_RestoreData
```

```
MOVLW (__OPXSTKPTRSIZE__ - 2)+__STKPTRSAVESIZE__  
movwf op1
```

```
    testz op1
    btfsc STATUS,ZERO_FLAG
    goto SAVE_FSR

loop:
    movlw SAVEADDR+4
    movwf FSR
    movfw op1
    addwf FSR
    BSF STATUS,_BANK_FLAG
    movfw R0
    movwf op1+1

    movlw op1+1
    movwf FSR
    movfw op1
    addwf FSR
    movfw op1+1
    BCF STATUS,_BANK_FLAG
    movwf R0

    decfsz op1

    goto loop

SAVE_FSR:

    BSF STATUS,_BANK_FLAG
    MOVFW SAVEADDR+4
    MOVWF FSR

RES_OP1:
    MOVFW SAVEADDR+2
    MOVWF op1
    MOVFW SAVEADDR+3
    MOVWF op1+1

W_S:

    MOVFW SAVEADDR+1
    MOVWF STATUS
    BTFSS STATUS,_BANK_FLAG
    GOTO L0
    MOVFW SAVEADDR
    RET
L0:
    BSF STATUS,_BANK_FLAG
    MOVFW SAVEADDR
    BCF STATUS,_BANK_FLAG
    RET

.endproc

.fixcode –
```

Definitions of variables in the program:

Variables	Definition
__RPLANE_ADDRESS_MAX__	The final address of R-Plane memory.
__FPLANE_ADDRESS_MAX__	The final address of F-Plane memory.
__OPXSTKPTRSIZE__	The linker calculates the whole memory size which is occupied by the computation register or stack pointer register in project and including function library code.
__BANK1_OFFSET_VALUE__	F-Plane Bank1 relative to Bank0, offset value of Bank1 common data block.
__STKPTRSAVESIZE__	Compiler reserves memory storage stkptr size during compilation according to computational complexity, the default value is 0.

**Note:** Please be careful with below two restrictions in interrupt implementation

1. Nested interrupt routine call is not allowed (i.e. enter another interrupt routine before leaving an interrupt routine)
2. If interrupt routine needs to declare variable to do calculation, it is suggested not to declare as local variable, but **global variable**, in order to save stack space.

## ISR\_SaveData\_5, ISR\_RestoreData\_5, ISR\_SaveData\_10, ISR\_RestoreData\_10

Some tenx chips have built-in auto saving function, this function before and after interrupt routine execution will auto store and restore the content of working register and status register. User can simply enable below flags to trigger the auto saving function.

Chip Name	Flag address, bit
TM57ME20	RPLANE 0x0B,7
TM57FLA80	RPLANE 0x10,4
TM57ML40	RPLANE 0x10,5
TM57FE80	RPLANE 0x07,5
TMU3130	RPLANE 0x07,5
TMU3131	RPLANE 0x07,5
TM56FA40	RPLANE 0x0B,2

The register contents of auto saving function and interrupt protection program are repetitive (i.e. repetitive working register and status register), therefore interrupt protection program implements in above chip will be able to complete its job in a simpler way.

**Note:** before starting interrupt protection program in the chip with auto saving function, interrupt protection function will trigger auto saving function, then do the protection action. Therefore, **during interrupt protection program execution, do not disable auto saving function manually**, otherwise, it will cause incomplete storage protection action.

Below is implementation assembly code in runtime library : `ISR_SaveData_5`, `ISR_RestoreData_5`, `ISR_SaveData_10` and `ISR_RestoreData_10` store and restore operations Data RAM. We describe the implementation code which stores data into F-Plane or R-Plane according to whether the chips have auto saving function.

### ● R-Plane:

Chip which stores interrupt protection data into R-Plane mostly contains auto saving function. Therefore, according to the saved register content and different amount, interrupt protection assembly code is divided into two types.

Store op1, op1+1, op3+2	Store op1, op1+1, op1+2, op1+3, op2, op3, op3+1, op3+2
<pre>.autoimport on .export _ISR_SaveData_5, _ISR_RestoreData_5  SAVEADDR5 = __RPLANE_ADDRESS_MAX__ - ( 1 + 1 ) ;***** ; save op1, op1+1, op3+2 ;*****  .proc _ISR_SaveData_5      MOVFW op1     MOVWR SAVEADDR5</pre>	<pre>.autoimport on .export _ISR_SaveData_10, _ISR_RestoreData_10  SAVEADDR10 = __RPLANE_ADDRESS_MAX__ - ( 1 + 6 ) ;***** ; save op1, op1+1, op1+2, op1+3, op2, op3, op3+1, op3+2 ;*****  .proc _ISR_SaveData_10      MOVFW op1     MOVWR SAVEADDR10+2</pre>

Store op1, op1+1, op3+2	Store op1, op1+1, op1+2, op1+3, op2, op3, op3+1, op3+2
MOVFW op1+1 MOVWR SAVEADDR5+1 MOVFW op3+2 MOVWR SAVEADDR5+2  RET .endproc  .proc <b>_ISR_RestoreData_5</b>  MOVRW SAVEADDR5 MOVWF op1 MOVRW SAVEADDR5+1 MOVWF op1+1 MOVRW SAVEADDR5+2 MOVWF op3+2  RET .endproc	MOVFW op1+1 MOVWR SAVEADDR10+3 MOVFW op1+2 MOVWR SAVEADDR10+4 MOVFW op1+3 MOVWR SAVEADDR10+5  MOVFW op2 MOVWR SAVEADDR10+6  MOVFW op3 MOVWR SAVEADDR10+7 MOVFW op3+1 MOVWR SAVEADDR10+8 MOVFW op3+2 MOVWR SAVEADDR10+9  RET  .endproc  .proc <b>_ISR_RestoreData_10</b>  MOVRW SAVEADDR10+9 MOVWF op3+2 MOVRW SAVEADDR10+8 MOVWF op3+1 MOVRW SAVEADDR10+7 MOVWF op3  MOVRW SAVEADDR10+6 MOVWF op2  MOVRW SAVEADDR10+5 MOVWF op1+3 MOVRW SAVEADDR10+4 MOVWF op1+2 MOVRW SAVEADDR10+3 MOVWF op1+1 MOVRW SAVEADDR10+2 MOVWF op1  ; MOVRW SAVEADDR10+1 ; MOVWF STATUS ; MOVRW SAVEADDR10  RET  .endproc

# ● F-Plane Bank0:

With Auto Saving Function Store: op1, op1+1, op3+2	Without Auto Saving Function Store: W, Status, op1, op1+1, op3+2
<pre> .autoimport on .export _ISR_SaveData_5, _ISR_RestoreData_5  #define _BANK_FLAG 5  SAVEADDR5 = 0x7D .proc _ISR_SaveData_5  MOVFW op1 MOVWF SAVEADDR5+0 MOVFW op1+1 MOVWF SAVEADDR5+1 MOVFW op3+2 MOVWF SAVEADDR5+2 RET  .endproc  .proc _ISR_RestoreData_5  MOVFW SAVEADDR5 MOVWF op1 MOVFW SAVEADDR5+1 MOVWF op1+1 MOVFW SAVEADDR5+2 MOVWF op3+2 ; MOVFW SAVEADDR5+1 ; MOVWF STATUS ; MOVFW SAVEADDR5 RET  .endproc </pre>	<pre> .autoimport on .export _ISR_SaveData_5, _ISR_RestoreData_5  #define _BANK_FLAG 5  SAVEADDR5 = 0x5B .proc _ISR_SaveData_5  MOVWF SAVEADDR5 ; save W MOVFW STATUS MOVWF SAVEADDR5+1 MOVFW op1 MOVWF SAVEADDR5+2 MOVFW op1+1 MOVWF SAVEADDR5+3 MOVFW op3+2 MOVWF SAVEADDR5+4 RET  .endproc  .proc _ISR_RestoreData_5  MOVFW SAVEADDR5+2 MOVWF op1 MOVFW SAVEADDR5+3 MOVWF op1+1 MOVFW SAVEADDR5+4 MOVWF op3+2 MOVFW SAVEADDR5+1 MOVWF STATUS MOVFW SAVEADDR5 RET  .endproc </pre>

Note: Command `SAVEADDR5 = 0x##`, where the ## value changes according to actual RAM size of the chip.

**● F-Plane Bank1:**

With Auto Saving Function	Without Auto Saving Function
<pre> .autoimport on .export _ISR_SaveData_5,_ISR_RestoreData_5  #define _BANK_FLAG 5  SAVEADDR5 = 0x7D  .fixcode +  .proc <b>_ISR_SaveData_5</b>  BTFSC STATUS,_BANK_FLAG GOTO B1  BSF STATUS,_BANK_FLAG MOVFW op1 MOVWF SAVEADDR5+0 MOVFW op1+1 MOVWF SAVEADDR5+1 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+2 BCF STATUS,_BANK_FLAG ret  B1: MOVFW op1 MOVWF SAVEADDR5 MOVFW op1+1 MOVWF SAVEADDR5+1 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+2 RET  .endproc  .proc <b>_ISR_RestoreData_5</b>  BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5 MOVWF op1 MOVFW SAVEADDR5+1 MOVWF op1+1 MOVFW SAVEADDR5+2 BCF STATUS,_BANK_FLAG MOVWF op3+2  RET </pre>	<pre> .autoimport on .export _ISR_SaveData_5,_ISR_RestoreData_5  #define _BANK_FLAG 5  SAVEADDR5 = 0x7B  .fixcode +  .proc <b>_ISR_SaveData_5</b>  BTFSC STATUS,_BANK_FLAG GOTO B1  BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5 ; save W BCF STATUS,_BANK_FLAG MOVFW STATUS BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+1 MOVFW op1 MOVWF SAVEADDR5+2 MOVFW op1+1 MOVWF SAVEADDR5+3 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+4 BCF STATUS,_BANK_FLAG ret  B1: MOVWF SAVEADDR5 MOVFW STATUS MOVWF SAVEADDR5+1 MOVFW op1 MOVWF SAVEADDR5+2 MOVFW op1+1 MOVWF SAVEADDR5+3 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+4 RET  .endproc  .proc <b>_ISR_RestoreData_5</b>  BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5+2 </pre>



With Auto Saving Function	Without Auto Saving Function
.endproc  .fixcode -	MOVWF op1 MOVFW SAVEADDR5+3 MOVWF op1+1 MOVFW SAVEADDR5+4 BCF STATUS,_BANK_FLAG MOVWF op3+2 BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5+1 MOVWF STATUS BTFSS STATUS,_BANK_FLAG GOTO L0 MOVFW SAVEADDR5 RET  L0: BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5 BCF STATUS,_BANK_FLAG  RET  .endproc  .fixcode -

## 6. Preprocessors

The preprocessor is a program that is invoked by the compiler to process code before compilation. Commands for that program are lines of the source file beginning with the character #, which distinguishes them from lines of source program text. The preprocessed source code, an intermediate file, must be a valid C program, because it becomes the input to the compiler.

Preprocessor directives and the related subject of macro expansion are discussed in this section. After an overview of preprocessor directives, the topics covered include textual macros, file inclusion, conditional compilation directives, and pragmas.

The preprocessor is controlled by the following directives:

Directive	Description
#define	Defines a macro
#undef	Removes a preprocessor macro definition.
#include	Inserts text from another source file.
#if	Conditionally suppresses portions of source code, depending on the result of a constant expression.
#ifdef	Conditionally includes source text if a macro name is defined.
#ifndef	Conditionally includes source text if a macro name is not defined.
#else	Conditionally includes source text if the previous #if, #ifdef, #ifndef, or #elif test fails.
#elif	#ifndef, or #elif test fails, depending on the value of a constant expression.
#endif	Ends conditional text.
#pragma	Set compiler status or to order compiler to finish some certain actions.

### Macro Definition

A preprocessor define directive directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

### Non-parameter Macro Definition

The #define directive can contain an object-like definition or a function-like definition.

#define versus const\_value

The #define directive can be used to create a name for a numerical, character, or string constant, whereas a const object of any type can be declared.

## Definition of Macro with Parameters

A macro argument can be empty (consisting of zero preprocessing tokens).

For example,

```
#define SUM(a,b,c) a + b + c
SUM(1,2,3)
```

// 1 is substituted for a, 2 is substituted for b, and 3 is substituted for c. \*/

## Files Include

A preprocessor include directive causes the preprocessor to replace the directive with the contents of the specified file. A preprocessor #include directive has the following format:

```
#include <file1.h>
Or
#include "file1.h"
```

For example:

```
#include <july.h>
```

## Conditional Compilation

A preprocessor conditional compilation directive causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- #if
- #ifdef
- #else
- #ifndef
- #elif
- #endif

## Pragma Directive (#pragma)

This directive is used to specify different options to the compiler. These options are specific for the platform and the compiler you use. If the compiler does not support a specific argument for #pragma, it ignores the #pragma directive without any error or warning message.

### ● #pragma tableromaddr

This pragma allows you to set the starting TABLE ROM address of global const variable which behind this pragma statement, and require you pass off to turn the option off. The specification is as follows:

```
declaration:      #pragma tableromaddr (const_variable_start_address)
                  #pragma tableromaddr (off)
```

### ● #pragma tableromdt

This pragma allows you to pointer value to global constant variable ,and use dt format access in ROM,if not enabled ,use retlw to access.(support since 0.5.7 version)

```
declaration:      #pragma tableromdt (on)
```

Const int value=0x3FFF; // declare a constant global variable, the following table use the difference between dt and retlw of constant global variables.

dt			Retlw		
000004:	3FFF	dt	000005:	18FF	RETLW 0xFF
			000006:	183F	RETLW 0x3F

advantage:

occupies the size of 1 ROM,which is conducive to the use of variables.

shortcoming:

the dt maximum value is 0x3FFF,only support int 、 unsigned int 、 short 、 unsigned short .

Int 、 short range:-8191~8191

unsigned int 、 unsigned short range:0~16383

supports specific IC use

advantage:

Constant global variable scope is the same as the local variable.

Char:-128~128

Unsigned char:0~255

Int 、 short:-32767~32767

Unaigned int 、 unsigned short:0~65534

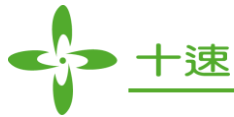
Long:- 2147483647~2147483647

Unsigned long:0~4294967295

shortcoming:

when using the int type ,will occupy the size of 2 ROMs

Long type will occupy the size of 4 ROMs,will take up more ROM space.



dt	Retlw
Support tableromdt IC:	Support Retlw introduction IC:All
TM57ME16	
TM57ME16AS	
TM57ME18	
TM57MA25	
TM57MA28	
TM57MA28B	
TM57MA29	
TM57MA29C	
TM57MA21B	
TM57MA15	
TM57MA16	
TM57MA1668	
TM57MA1672	
TM57M5526C	
TM57M5536C	
TM57MA17	
TM57MA18	
TM57P8620	
TM57P8625	
TM57P8640	
TM57P8645	
TM57M5620	
TM57M5625	
TM57M5640	
TM57M5645	
TM57M5610	
TM57M5615	
TM57ME15B	
TM57ME15CG	
TM57MA45	
TM57MA46	
TM57MA33	
TM57M5541	
TM57M5551	

## 7. Mix of C and Assembly Code in C Project

### Basic Concept

Generally, there have two conditions which will call assembly function in a C application program.

- (1) Some special instruction operations of the single chip, which cannot be described using the standard C language syntax.
- (2) To implement single chip system that emphasizes immediate controllability, assembly code must be referred when it is necessary to implement portion of the code to improve the efficiency of the program execution.

In this way, there will be C and assembly hybrid programming condition occurs in one C project. In this section, we will discuss the basic way of each hybrid programming and experience sharing, and please refer to “[Appendix](#)” section for the example to learn more about the actual application.

In TM57 C Compiler, the mixed mechanism between C code and assembly code is done in intuitive way. The descriptions are divided into three parts as shown below:

1. Embed inline assembly instruction directly in C program (please refer to “[asm Declarators](#)” section)
2. Call assembly function in C program
  - The C code in \*.c file uses function prototype to declare the function which is exported from an assembly code. The exceptional condition is when no parameter is needed in assembly function, user can optionally not declare assembly function prototype in C program.
  - Function which are exported using keyword **export** in assembly code in \*.asm file include two types:
    - (1) Export label
    - (2) Assembly function which is defined using **.proc** / **.endproc** keywords
3. Call C function in assembly code:
  - Declare and define function of C code in \*.c file
  - Call C function in \*.asm file using `call _FunctionName` assembly code

If assembly function returns value, the return value will be stored in register op2.

After C and assembly code are compiled, the related C code, assembly program file and related program library must be added into project manager tree, for project compiling. The function call between C code and assembly code is divided into with passing parameter or without passing parameter, which will be described in [Appendix](#) orderly with the example.

## C Program Calls Assembly Function without Passing Parameter

If no parameters need to be passed between C and Assembly programming call, please refer to [Example 2](#).

## C Program Calls Assembly Function with Passing Parameter

When C program calls assembly function with passing parameter, the order of the passing parameter in assembly function is “from right to left” order, and the addressing order of the local variable declaration in assembly function is “from bottom to top” order. Parameter and variable addressing in assembly language is relative. Please refer to below example for the description:

### C program caller:

```
int Sum (int, int*);
main()
{
    int a=255,b=20,c=0;
    c= Sum (a,&b);
}
```

### Assembly function callee:

```
.autoimport    on
.debuginfo     on

.export        _Sum
.declfunc      Sum(2,3)
.CODE
.proc _Sum

    MOVFW Sum_PARAM+1
    MOVWF Sum_LOCAL
    MOVFW Sum_PARAM+2
    MOVWF Sum_LOCAL +1
    MOVFW Sum_PARAM+0
    MOVWF FSR
    MOVFW $00
    ADDWF Sum_LOCAL,1
    BTFSC STATUS,0
    INCF Sum_LOCAL +1,1
    INCF FSR,1
    MOVFW $00
    ADDWF Sum_LOCAL +1,1
    MOVFW Sum_LOCAL
    MOVWF op2
    MOVFW Sum_LOCAL +1
    MOVWF op2+1
    RET

.endproc
```

In `.declfunc Sum (2,3)`, the first parameter is the memory size occupied by local variable; while the second parameter is the size of the passing parameter (in byte unit). From above example, according to the data type of passing parameter, calculate the memory size as  $3 = 2 (\text{int}) + 1 (\text{char}^*)$ . Local variable is used to store temporarily the input value of int, the memory size is 2 (please refer to Section [Declarations](#) for data type size ).

**Local variable** naming rule in assembly function

*FunctionName\_LOCAL*

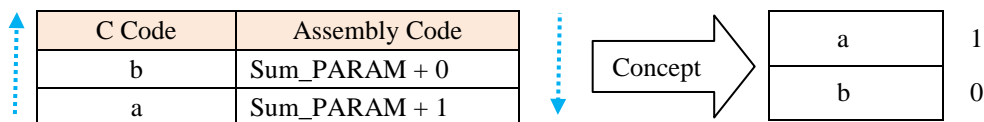
Assembly
Sum_LOCAL + 0

**Parameter** setting order is “from left to right” order; therefore, the starting address of the corresponding assembly function is as following table

*FunctionName\_PARAM*

Input parameter direction

Parameter addressing order



Please refer to Appendix [Example 3](#) .

## Assembly Code Calls C Function

Pure assembly file (\*.asm) may also calls function declared in C code, please refer to Appendix [Example 4](#).



## C and Assembly Language Hybrid Programming Experiences

In C project, mixed C and assembly language programming can improve the operating efficiency of the single chip applications, and get the best fit between software and hardware. Hereby, share some experiences in practical application

### (1) Using assembly instructions carefully

Relative to assembly language, C language programming has below advantages: improve the development efficiency, using natural language way to edit the commands and statements. The modulation is easy to manage and maintain and the program is portable in different platform. Therefore, it is strongly suggested in C language programming avoid embedded inline asm or using all assembly language commands to write the module program.

Viewpoint of the data storage space utilization, TM57 C compiler is certainly more efficient than manually setting variables and parameters address. This can also reduce repetitive addressing problem which cause hidden and identified errors. At the same time, C language provides complete function library, various and intuitive control and computing functions. Therefore, except some single chips which greatly emphasize timeliness or when C language cannot support the operation, user can consider using assembly instruction to implement, the other condition is still suggested should be written in C language.

### (2) Try to use embedded inline asm to replace

This is not contradicted with the statement “using assembly instructions carefully” mentioned above. In practical application, relative to the C language implementation, but using assembly language to realize part of the programming code, can indeed improve the operational efficiency. Of course, it is recommended to use embedded inline asm statement for implementation. However, we strongly recommend to avoid writing “pure assembly language file” (\*.asm file).

Programming code which is similar to pure assembly file may still be implemented in C language by using C standard syntax to define all variables and functions name (include formal parameter and local variables which need to be passed) and the parameter statement which needs to be returned, however the content of the instruction of the function is written with embedded inline asm instruction. In other words, using function syntax to wrap inline asm. In this way, the operation efficiency of the function will be almost the same with the pure assembly language coding; the different is each format of parameter passed is unified by the C language standard syntax. So, it can improve management and ease of maintenance.

### (3) Avoid using .org xx command while compiling C/ASM hybrid programming

Addressing mode in assembly program is set to reallocate address mode as default. Therefore, common pure assembly program adds `.org xx` command in front of the program, to switch to absolute address mode, for example

```
.org
    goto start
start:
    ....
```

But while compiling C/ASM hybrid programming, it is strongly suggested **not** to use `asm(“.org xx”)`, to avoid unpredicted error.

## 8. Create Function Library

### Function Library

Different from executable file, function library is not an independent program code, but the code which provides service to other programs. Function library is composed by many relocatable object modules, the file extension is \*.lib. The function of a series of related operations can be gathered up and created to be a function library, which can be called by other program, or can directly import another library function provided by TM57 C. In this way, it benefits to achieve code reusable, and reduce the burden of repetitive coding.

### Use Function Library

If the implementation is a simple and small application, it is not suggested to refer to function library, because during compiling and linking process, the content of function library will be integrated into the executable file. This will spend more system resources and consume more time in loading to internal memory.

When large scale of application is developed using function library mechanism will provide below advantages:

- Gather up the related operation modules into a single function file; let the program management and maintenance easier.
- Reduce function repetitive development time, and more organized in the description of the document.
- Achieve program sharing purpose, efficient development system and shorten the development time.

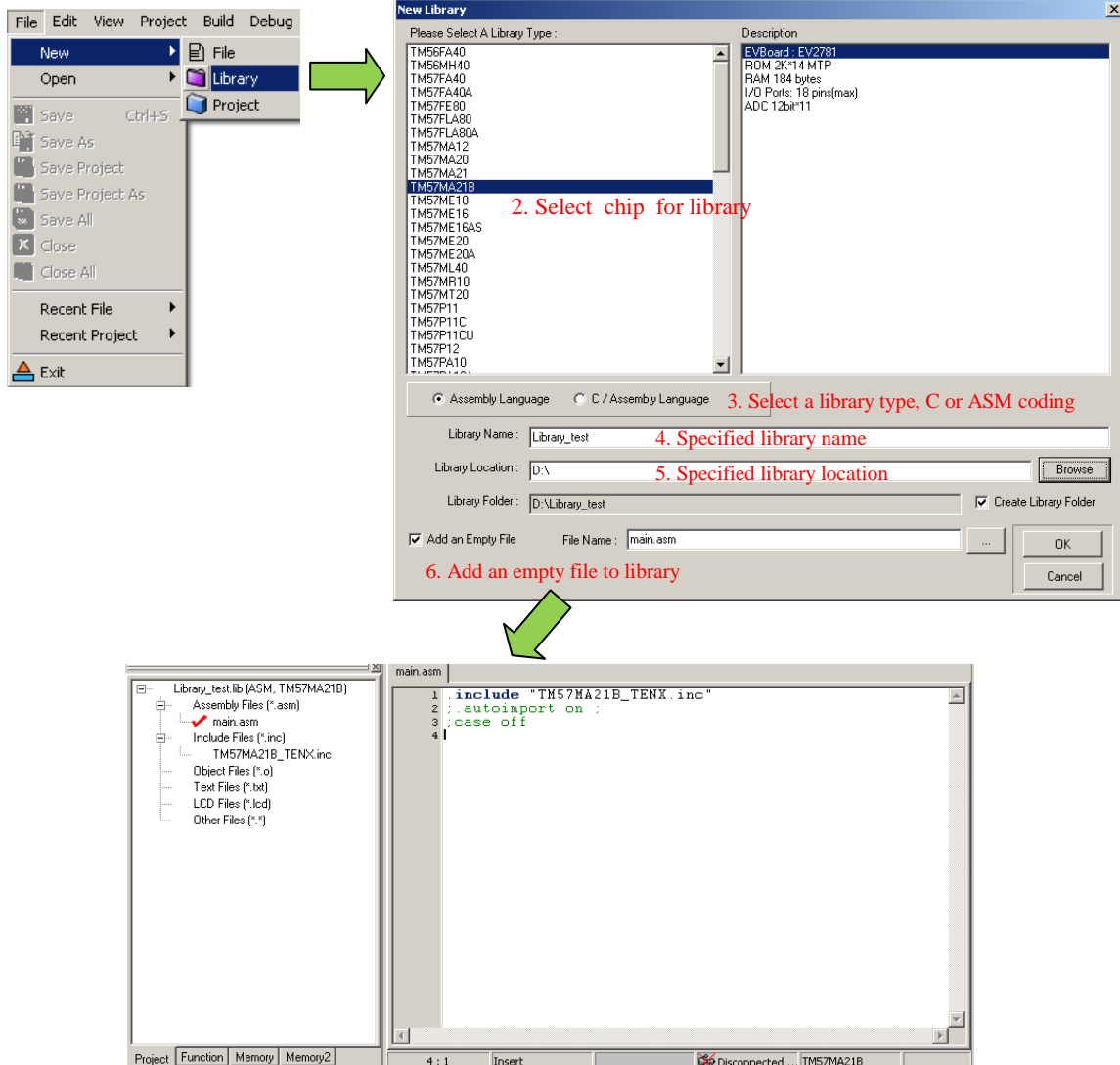
### Methods to Create the Library

C language or assembly language can also create function library through the tool provided by TICE99 IDE. There are two methods to create library and the concept shown as below figure:

Method 1: The same as creating new project, while creating a library, user can follow below steps

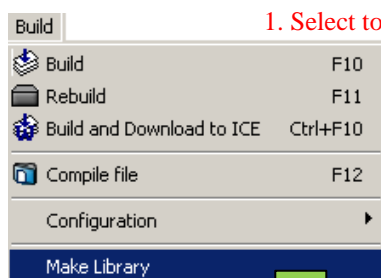
1. Select File|New|Library, in New Library window put in library information, includes ic type, C/assembly language, defined or created library list, etc...
2. After creating library, the same as common project, decide optionally c/asm file or object file (\*.o) added to library.
3. After editing the related files, user can directly compile the files to create \*.lib file.

## 1. Select File | New|Library

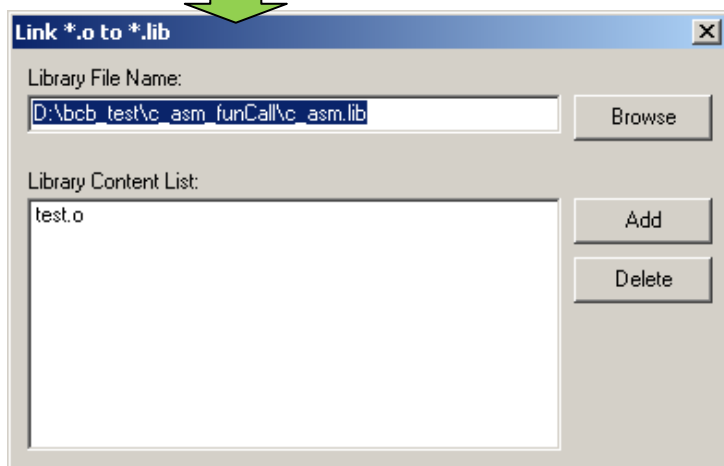


Method 2: Using “Making Library” tool to create library and the step is divided into two sections:

1. **Create object file**, single C language or assembly programming file which contain many function module is compiled and assembled to create an object file (\*.o).
2. **Create function library**, using tool provided by TICE99 IDE: library maker, decide selectively which object files to be gathered up to create a single function library (\*.lib)



1. Select toolbar Build | Make Library

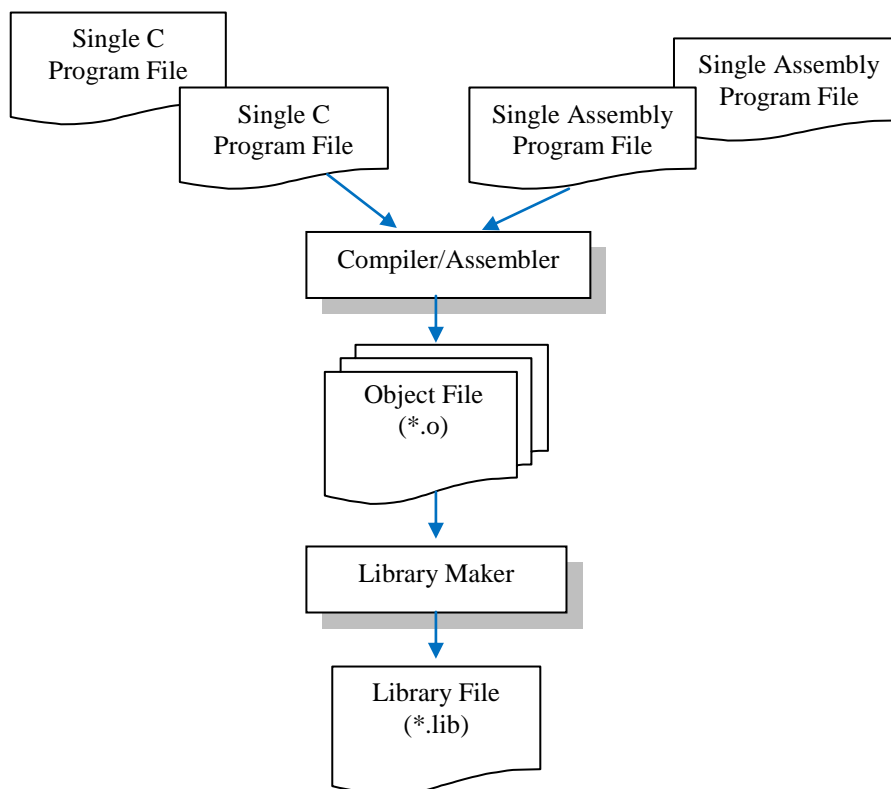


4. Browse the library name you want to create

2. Selectively add the object files (can be many files)

3. Selectively delete the object files

### Concept:



## How to Use Function Library

When C or assembly file in the same project refer to some function modules in function library, please follow below steps to import function library:

1. Select the library file item in Project Manager

2. Right click, select the current saved file from the open window

3. Select the function library file imported

4. Complete the selecting, press the “Open” button

5. Cancel the selecting by pressing “Cancel” button

6. Function library file name imported, appears in Project Manager

## 9. Memory Map

In this section, we will show the memory address which are used during C compiler is in operation or when interrupt is triggered and needs to store and restore data. The detail memory map of each single chip, please refer to each single chip's data sheet.

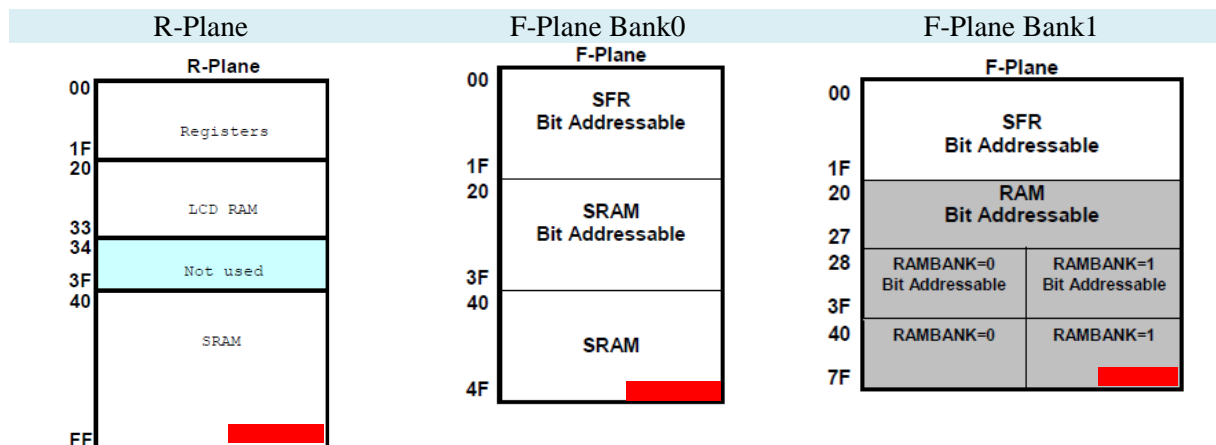
- Register address may be used during operation process: FSR, RSR, OP1~OP4, tmp1 and stkptr. Below figure shows the maximum possible usage:

	0	1	2	3	4	5	6	7	8	9	A	b	c	d	E	f
0					FSR			RSR								
10																
20	OP1				OP2				OP3				OP4			
30	tmp1	stkptr														
...																

### Note:

- When R-Plane in the operating MCU uses MOVWR and MOVW commands to do memory access, it will save RSR register content.
- The address range to store OP1~OP4, tmp1 and stkptr is changeable, it is positively correlated with program operation complexity, detail description please refer to [Fplane / Rplane Declarations](#).

- Register address needed when an interrupt is triggered and needed to store and restore data is shown below (as shown in red color block):



## 10. Appendix

### Example 1

- Calculate the string length

```
int strlen(char *tar)
{
    // Clear op2;
    asm("clrf op2");
    asm("clrf op2+1");

    asm("_strlen_LOOP:");
    // Read from source
    asm("movfw %o", tar);

    // Read value of tar address
    asm("call runtime_Ind_Read");
    asm("movwf op3");

    // Check end
    asm("testz op3");
    asm("btfsc STATUS, ZERO_FLAG");
    asm("ret");

    // Next
    asm("incf %o,1", tar);
    asm("incf op2,1");
    asm("goto _strlen_LOOP");
}
```

- Copy the source string (string pointed by pointer src) to destination string (string pointed by pointer tar)

```
char *strcpy(char *tar,char *src)
{
    // Return tar value from op2 ( 0x24 )
    asm("movfw %o",tar);
    // return tar pointer in op2 address ( 0x24)
    asm("movwf op2");
    asm("_strcpy_LOOP:"); // generate label name
    // Read from source
    asm("movfw %o",src); // Set offset of LOCAL name src
    asm("call runtime_Ind_Read"); // call indirect read
    asm("movwf op3"); // op3 to write to target
    // Save to target
    // Set offset of LOCAL name tar ( strcpy_LOCAL+1 )
    asm("movfw %o",tar);
    asm("call runtime_Ind_Write"); // call indirect write
    // Check end
    asm("testz op3");
    asm("btfsc STATUS, ZERO_FLAG");
    asm("ret");
    // Next
    asm("incf %o,1",src);
    asm("incf %o,1",tar);
    asm("goto _strcpy_LOOP");
}
```



- Compare whether two strings are the same

```
int *strcmp(char *tar,char *src) // Compare whether two strings are the same
{
    asm("_strcmp_LOOP:"); // generate label name

    // Get tar value from op3 ( 0x28 )
    asm("movfw %o",tar);
    asm("call runtime_Ind_Read"); // call indirect read
    asm("movwf op1");

    // Set offset of LOCAL name src
    asm("movfw %o",src);
    asm("call runtime_Ind_Read");
    asm("testz op1");
    asm("btfsc STATUS,2");
    asm("ret");

    asm("subwf op1,0");
    asm("btfsc STATUS,2");
    asm("goto LZ1");
    asm("goto LZ0");

    // When ZF is Zero
    asm("LZ0:");
    asm("movlw $FF");
    asm("btfsc STATUS,0");
    asm("xorlw $FE");
    asm("movwf op2");
    asm("btfss op2,7");
    asm("movlw $00");
    asm("movwf op2+1");
    asm("ret");

    // When ZF is one
    asm("LZ1:");
    asm("btfsc STATUS,0");
    asm("movlw $01");
    asm("xorlw $01");
    asm("movwf op2");
    asm("clrf op2+1");
    asm("incf %o,1",src);
    asm("incf %o,1",tar);
    asm("movfw %o",tar);
    asm("call runtime_Ind_Read");
    asm("movwf op1");
    asm("testz op1");
    asm("btfsc STATUS,2");
    asm("ret");
    asm("goto _strcmp_LOOP");
}
```

- Concatenate original string (string pointed by pointer src) to the end of the destination string (string pointed by pointer tar).

```
// Concatenate src string to the end of tar string
char *strcat(char *tar,char *src)
{
    // Return tar value from op2 (0x24)
    asm("movfw %o",tar);
    asm("movwf op2");

    // Check the char of tar string is '\0'
    asm("START:");
    asm("movfw %o",tar);
    asm("call runtime_Ind_Read");
    asm("movwf op1");
    asm("testz op1");
    asm("btfss STATUS,2");
    asm("goto tarnext_LOOP");

    asm("_strcat_LOOP:");
    asm("movfw %o", src);          // Set offset of LOCAL name src
    asm("call runtime_Ind_Read"); // call indirect read
    asm("movwf op3");             // op3 to write to target

    asm("testz op3");
    asm("btfsc STATUS, ZERO_FLAG");
    asm("goto add_null");
    asm("movfw %o", tar);
    asm("call runtime_Ind_Write"); // call indirect write
    asm("testz op3");
    asm("goto Next_LOOP");

    asm("Next_LOOP:");
    asm("incf %o", tar);
    asm("incf %o", src);
    asm("goto _strcat_LOOP");

    asm("tarnext_LOOP:");
    asm("incf %o", tar);
    asm("goto START");

    asm("add_null:");
    asm("movlw $00");
    asm("movwf op3");
    asm("movfw %o", tar);
    asm("call runtime_Ind_Write");
    asm("ret");
}
```

## Example 2

C program calls assembly function WRKeyData1Bytes which does not need to pass parameter and return value.

### C code: call assembly function directly

There is no passing parameter and return value needed during call process, therefore user can decide whether to declare assembly function prototype, and call the function directly.

```
main()
{
    WRKeyData1Bytes();
}
```

### Assembly language function:

```
;WRKeyData1Bytes
.autoimport on
.debuginfo on
.export _WRKeyData1Bytes ; leader char is _ ( at prefix )
; use ".declfunc" directive to allocate the size of local and parameter.
; format: .declfunc funname(local_size,param_size)
.declfunc WRKeyData1Bytes(0,0)

.CODE
.proc _WRKeyData1Bytes
    movlw    40h
    addwf    79h,0
    movwf    FSR
    bsf 03h,5 ;STATUS,RAMBANK
    movfw    7Fh
    movwf    00h ;INDF
    bcf 03h,5 ;STATUS,RAMBANK
    movlw    .1
    addwf    79h,1
    ret
.endproc
```

### Example 3

#### C code: declare function prototype, and call strcpy() which is exported from assembly code

C program calls strcpy function which is declared and defined by assembly language, this function needs two character pointers as input to do string copy, but without return value.

```
void strcpy(char*,char*);      // Function Prototype for asm function
main()
{
    char String1[12] = "I like TM57";
    char String2[12] = "I like TM89";
    strcpy(String1, String2);   // call asm function
}
```

#### Assembly language strcpy() function

Before function is defined, the keyword **export** in `.export _strcpy` is used to export function `_strcpy`, and the keyword **declfunc** is used to declare local variable and input parameter memory size (BYTE unit) of the function `strcpy`, the declaration expression is: `.declfunc strcpy(0,2)`.

```
; string copy function by asm code
;*****
; char* strcpy (char* dest, char* src)
;*****
    .autoimport    on
    .debuginfo     on
    .export        _strcpy ; leader char is _ ( at prefix )
    ; use ".declfunc" directive to allocate the size of local and parameter.
    ; format: .declfunc funname(local_size,param_size)
    ; parameter count= source(0)+target(2) = 2
    ; declared 2 bytes space to parameter
    .declfunc      strcpy(0,2)

.CODE

.proc _strcpy ;*** parameter address stack counter is from right to left

    movfw strcpy_PARAM+1 ; target
    movwf op2 ; return tar pointer in op2 address (0x24)

LOOP:
    ;*** Read from source
    movfw strcpy_PARAM+0 ; Set offset of LOCAL name src
    call runtime_Ind_Read ; call indirect read (call runtime library function)
    movwf op3            ; op3 to write to target

    ;*** Save to target
    movfw strcpy_PARAM+1 ; Set offset of LOCAL name tar
```

```
call runtime_Ind_Write ; call indirect write (call runtime library function)

;*** Check end ?
testz op3
btfsc STATUS, ZERO_FLAG
ret

;*** Next
incf strcpy_PARAM+0,1
incf strcpy_PARAM+1,1
goto LOOP

.endproc
```

### Example 4

In this example, assembly program calls C language strcpy() function, to do string copy.

#### C code: define strcpy() function to be used by assembly code

```
void strcpy(char* des,char* source)
{
    int i=0;
    for (i=0;source[i] != '\0'; ++i)
    {
        des[i] = source[i];
    }
    return;
}
```

#### Assembly code: Call C language strcpy() function

After the compiling of C Compiler, the original C function name will be added by prefix '\_' (that means, \_strcpy). Therefore, in this example, the syntax of assembly code to call C function strcpy becomes **call \_strcpy**.

In below program code, set original string variable to src\_str, destination string variable to tar\_str, and set these two variable addresses to strcpy\_PARAM+0 and strcpy\_PARAM+1 respectively, for C function strcpy computation. User can read variable tar\_str to check the computation result (i.e. result value of tar\_str is "ABCD").

```

;*****
;*** call strcpy function from asm code
;*****

.autoimport on

src_str = 40h
tar_str = 45h

movlw 'A'
movwf src_str
movlw 'B'
movwf src_str+1
movlw 'C'
movwf src_str+2
movlw 'D'
movwf src_str+3
movlw 0
movwf src_str+4

;*****
;*** pass parameter from right to left
```

```
*** pointer size is 1 byte
*****
movlw src_str
movwf strcpy_PARAM+0 ; store source address to PARAM+0
movlw tar_str
movwf strcpy_PARAM+1 ; store target address to PARAM+1
call _strcpy

loop:
  nop
  goto loop
```