FTM8 C Programmer's Guide

Fortior Technology (Shenzhen) Co., Ltd.

Contents

1 Fortior C Programming Language

1.1 Characteristics of FTCC Programming Language

FTCC (*Fortior Technology C Compiler*) programming language basically conforms to ISO/IEC 9899:2011 (referred to as C11 standard), and is used to specify C11 for embedded domain-specific extensions.

1.1.1 Character Set

The character set most commonly used in FTCC programming language includes:

- The character set specified in C11 standard, including 52 Latin letters, 10 digits, space, horizontal tab, vertical tab, newline and the following 29 special characters.

Character	Name	Character	Name	Character	Name
!	Exclamation Point	+	Plus Sign	“	Double Quotation Mark
#	Octothorpe	=	Equal Sign	{	Open Brace
%	Percent Sign	~	Tilde	}	Close Brace
^	Circumflex	[	Open Bracket	,	Comma
&	Ampersand	]	Close Bracket	.	Period; Dot
*	Asterisk	‘	Apostrophe; Single Quotation Mark	<	Less-than Sign
(	Open Parenthesis	|	Vertical Bar	>	Greater-than Sign
_	Underscore	\	Back-slash	/	Slash
)	Close Parenthesis	;	Semicolon	?	Question Mark
-	Hyphen; Dash	:	Colon		

- Chinese Characters: Chinese characters are valid only in comment statements, character constants, string constants or file names. Otherwise, they are considered as invalid characters (Chinese characters are not recommended for FTCC).
- Absolute address symbol “@”;
- Format characters including backspace, carriage return and space. These characters are valid only if they appear in comment statements, character constants, string constants or file names. Otherwise, they are considered as a space.

1.2 Token

There are five categories of tokens in FTCC programming language: identifiers, keywords, operators, separators and constants.

1.2.1 Identifier

An identifier is mainly used for the naming purpose of variables, functions, labels, etc. in the program. The names of C standard library functions and CSCC arithmetic library functions are system-defined, and the rest are

user-defined. A string is considered as a valid identifier if it only contains letters (A~Z or a~z), numbers (0~9) or underscores (_) and starts with a letter or underscore, such as b, y, x6, Tom_1, sum1, etc. A valid identifier cannot start with a number or a dash, or contain other characters. 3x, -3x, n*h and cow-1, for example, are invalid identifiers.

To use identifiers, you need to notice:

- Identifiers are case-sensitive (lowercase and uppercase letters are distinct), and every character is significant. For example, SOME and some are two different identifiers.
- An identifier is a token that is used to identify the variables, constants, functions etc. It shall contain special meaning for easy understanding.

1.2.2 Keyword

Keywords are predefined words that have special meanings to the compiler, that is, the programmer is not allowed to use them as normal identifiers. In addition to some of the standard C keywords, there are some CSCC extended keywords, such as, bankx(0<=x<=15), sbit and so on.

float	sizeof	return	long	double	auto
signed	goto	char	break	static	short
case	switch	const	int	continue	typedef
if	unsigned	default	while	void	do
volatile	struct	else	register	union	extern
enum	for	asm	asm	sbit	bankx
interrupt					

The keywords in red are extended keywords for FTCC.

1.2.3 Operator

Operator	Name	Operator	Name	Operator	Name
!	Logical NOT	<=	Less than or equal to	{	Open Brace
%	Modulo	==	Equal to	|	Bitwise OR
&	Bitwise AND/Address-of	!=	Not equal to	}	Close Brace
++	Increment	>=	Greater than or equal to	~	Bitwise Complement
(	Open Parenthesis	>>	Shift right	<<=	Left Shift Assignment
)	Close Parenthesis	<<	Shift left	>>=	Right Shift Assignment
*	Multiplication/ Indirect Access	:	Colon	-=	Subtraction Assignment
+	Addition	;	Semicolon	|=	Bitwise OR Assignment
,	Comma	<	Less than	&=	Bitwise AND Assignment
-	Subtraction	=	Assignment	+=	Addition Assignment
.	Direct Member Selection	>	Greater than	/=	Division Assignment
/	Division	?	Conditional	%=	Modulo Assignment
--	Decrement	@	Absolute Address	^=	Bitwise Exclusive OR Assignment

Operator	Name	Operator	Name	Operator	Name
->	Indirect Member Selection	[	Open Subscript		
&&	Logical AND	]	Close Subscript		
||	Logical OR	^	Bitwise Exclusive OR		

1.2.4 Constant

There are two types of constants in programming language: literal constants and symbolic constants. Literal constants, such as 49 and 'a', are the constants that lexical analyzer mainly identify, and symbolic constants, such as enumeration constants, objects with unchanged values, are not the constants in lexical scoping. C11 standard uses the term constant for integer constants, floating-point constants, character constants and string constants.

1.3 Comment

FTCC supports multi-line comments that start with a forward slash and asterisk (/*) and end with an asterisk and forward slash (*/). Any text between /* and */ is treated as a comment. It also supports the single-line comments in C99 standard, which start with two forward slashes (//). Comments are completely ignored by the C compiler, and appear anywhere in the program.

1.4 Declaration

1.4.1 Declaration Specifier

A declaration is a C language construct that introduces one or more specifiers into the program and defines their meaning and properties. The declaration specifiers include storage-class specifiers, type specifiers and type qualifiers. It makes no sense to show normal users detailed differences among them, thus we focus more on their functions in this section.

Specifier	Description
extern	It is used to declare external functions or variables and mostly appears in the header file or inside the code block. The functions or variables declared with the extern keyword will be linked externally (external linkage). The programmer can declare a function or variable as a global function or variable, so that they can be accessed from anywhere in the program. The name and location of the external object is managed by a linker. Linker error occurs if a name that is not defined in the project is used.
const	It creates a read-only reference to a value (l-value).
typedef	It defines a new name for a data type, also known as a "type alias."
volatile	It prevents the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler. Objects declared as volatile are omitted from optimization.
static	It is used to declare functions or variables. In a data declaration, it creates some local or global static objects that must be allocated within a static region.
bankx(0~15)	It defines a variable must be located in the bankx.

Specifier	Description
register	It is compatible with that in ANSI C and has no practical function.

1.4.2 Type Specification

Type	Classification of Type		
sbit	Integer Types	Arithmetic Types	Scalar Type
enum			
unsigned char, signed char			
unsigned int, signed int			
unsigned long, signed long			
float	Floating-point Types		
double			
T*	Pointer Types		
T[…]	Array Types		Aggregate Types
struct{…}	Structure Types		
union{…}	Union Types		
void	Void Types		

1.4.3 Type Size

Type	Size (bit)	Range of Values	Description
sbit	1	0~1	Bit Type
enum	8	-127~127	Enumerated Type
signed char	8	-127~127	Signed Character
unsigned char	8	0~255	Unsigned Character
signed int	16	-32767 ~ 32767	Signed Integer
unsigned int	16	0 ~ 65535	Unsigned Integer
signed long	32	-2147483647 ~ 2147483647	Long Signed Integer
unsigned long	32	0 ~ 4294967295	Long Unsigned Integer
float	32		Single Precision
double	32		Double Precision

1.4.4 Pointer Type

A pointer is declared using the following syntax:< DataType > * < Name of Pointer Variable>. For example, if a pointer is declared as `char*p;//p`, its data type is char.

Note: The pointer is stored in RAM, but it points to either RAM or ROM, which is defined by the programmer. If the pointer has been assigned to RAM, it cannot point to ROM, and vice versa.

For example:

```
char aVar[10];
char *p = "abccde"; // an error occurs because "abccde" is the ROM address but char *p is assigned to RAM (The compile is successful after the const keyword is added in the definition)
```

```
p = aVar;  // p points to RAM, and aVar is an array of RAM, so the assignment is correct.
```

1.4.5 Array Type

There are two types of arrays in FTCC: array stored in ROM (constant array) and array stored in RAM. If a const is defined, the array is stored in ROM, instead of RAM. Like ANSI C, an array name in FTCC can also be used to represent the first address of the array.

Note: Global const arrays are stored in ROM, and local const arrays in RAM. A pointer declared with const points to ROM, whether it is a local pointer or a global pointer. If it is assigned to RAM, the compiler fails with error message.

1.4.6 Structure Type and Union Type

The structure types and union types in FTCC are basically compatible with those in ANSI C, and the related syntax is not described in details in this guide. However, the following shall be noted:

- The structure member cannot be sbit;
- The structure member ignores bankx specifier.

For example:

```
struct tagT {
    bank2 int var;      //The bank is ignored as the structure is stored in RAM as a whole.
    sbit Var;    //sbit is not allowed. An error is reported.
};
```

1.4.7 Void Type

The void type applied in FTCC is basically the same as that in ANSI C. It falls into the following categories:

- The void return type is used when a function does not return anything;
- A void pointer (with a * symbol after the void keyword) points to objects of any data type;
- If an argument list is replaced with the keyword void, it indicates a function that takes no arguments.

1.4.8 Consistency in Declaration

According to standards for the C language, declarations of a same object can be repeated, but all these declarations must be compatible, otherwise, it may cause fatal error.

For example:

```
int Var;
char Var;
```

Compile error occurs if the two command lines are in a same source file. But if the command lines are in two separate source files, the C compiler generates the target program normally while runtime error occurs. In most cases, two or more incompatible declarations results in memory access conflicts. Incompatible declarations are

also common in function declarations.

For example:

```
void Func(int P1, int P2);
void Func(char P1, int P2);
```

The C compiler also cannot detect the error if the command lines are in two separate source files. For function symbols, type compatibility means that return types must be compatible with parameter types. Inconsistent declarations are common errors in the C programming language, but the compiler cannot detect them in many cases, resulting in fatal runtime errors. Programmers must be careful to such errors.

1.5 Expression

Operators in FTCC are fully compatible with those in ANSI C.

1.5.1 Arithmetic Expression

When ternary operators (: and ?), binary operators [+ (plus), - (minus), * (multiplication), / (division) and % (modulo)] and unary operators [+ (positive), - (negative), + + (increment) and -- (decrement)] are used in the arithmetic expression, you shall pay attention to the following:

- The data type of the result of a division operation depends on the data types of the operands. If both operands are of type int, the result is of type int, and the fractional part is automatically dropped. If one of the operands is of type float, the result is of type float.
- The operand type and result type of modulo operation must be of type int.
- The operand of both increment or decrement operations must be a modifiable l-value.

1.5.2 Assignment Expression

When binary operators [= (basic assignment), += (addition assignment), -= (substruction assignment), *= (multiplication assignment), /= (division assignment, <<= (bitwise left shift assignment), >>= (Bitwise right shift assignment), &= (Bitwise AND assignment), ^= (Bitwise XOR assignment) and |= (Bitwise OR assignment)] are used in the arithmetic expression, you shall pay attention to the following

- In an assignment statement, the operand on the left-hand side should be a modifiable l-value.
- The assignment operation always takes place from right to left, and the result is always an r-value.

For example:

```
a = b = c = 6; //same as a = (b = (c = 6)) and assignment operators are right-associative.
(a = b) = 6;    //invalid because the result of a = b is an r-value.
```

1.5.3 Comma Expression

Comma expression is formally treated as a single expression < expression 1>, < expression 2>, ..., < expression N>. Two expressions separated by a comma are evaluated left to right, and the value of the entire

expression is the value of the last expression. For example:

```
int a;
int b;
a = (b++, 10, 20); // a is equal to 20
```

When comma expressions are used in the arithmetic expression, you shall pay attention to the following:

- The result of a comma expression is a r-value. For example, `(a,b) = 6` is invalid;
- In C programming language, a comma as a separator is used to separate multiple arguments in a function call, so the comma operator and its operands must be enclosed in parentheses. If not, the compiler treats comma expressions as two arguments. For example, the expression `aa(a+6,b+6)` is regarded as two arguments, rather than a comma expression.

1.5.4 Type Conversion

There are two types of type conversion in C programming language: explicit type conversion and implicit type conversion. Explicit type conversion is also known as type casting and is user initiated. It specifies what data type to treat a variable as in a given expression with general syntax (<type name >) (<expression>). In some cases, the complier won't raise an error even if the operation isn't type-safe. The result of an explicit type conversion is always a r-value. Implicit type conversion, also called automatic type conversion, is performed automatically by the compiler when one data type is required. Generally, this type conversions are defined by C language standards and compiler designers. In principle, implicit type conversions are performed in type-safe manner. The standard ANSI C provides detailed descriptions on implicit type conversions, but a normal user does not have to know all details. You can also read section 6 of *C: A Reference Manual* to learn the type conversions.

1.5.5 Function Call

A function call has the form: < function name >([argument expression 1, argument expression 2,... argument expression n]). According to function prototype, the argument list can be empty. The arguments in a function call are separated by commas. To call a function, you need to notice:

- A temporary variable allows arguments where there is an implicit conversion from the argument type to the parameter type. Otherwise, the complier reports errors.
- The number of arguments must match the number of parameters. FTCC does not support the variable argument list.
- Order of evaluation of any part of any expression, including order of evaluation of function arguments is unspecified in the C language. The compiler can evaluate operands and other subexpressions in any order. FTCC uses a left-to-right evaluation order, so expressions used as arguments shall not contain side effects.

For example:

```
void Func(int a, int b, int c, int d)
{
    return;
}
int main()
{
    int i = 2;
    Func(++i, --i, i++,  i--); return 0;
}
```

The arguments are evaluated from left to right, and four arguments of Func are 3,2,2,3.

1.6 Statement

The statements applied to FTCC are fully compatible with the standard ANSI C.

Statement	Syntax	Example
Expression Statements	<Expression>	`Var = i + 8;`
Labels	Label:	`endFunc1:`

Compound Statements	{
	[Statement List]
	}
		`{`
		`  Var += 1;`
		`}`

if Statements	if (<Expression>) <Statement>	`if (Var <= 8)` `  Var = 8;`

if-else Statements	if (<Expression>)
	<Statement>
	else
	<Statement>
		`if (Var <= 8)`
		`  Var = 8;`
		`else`
		`  Var = 9;`

while Statements	while (<Expression>) <Statement>	`while (Var <= 8)` `  Var++;`

do Statements	do
	[Statement List]
	while (<Expression>)
		`do {`
		`  Var++;`
		`}`
		`while(Var <= 8)`

for Statements	for (<Expression>;<Expression>;<Expression>)	`for(i=1; i <=8; i++)` `  Var++;`

Statement	Syntax	Example

switch Statements	switch(<Expression>)
	{
	[Statement List]
	}
		switch (Var)
		{
		case 1: gV += 1;
		}

break Statements	break	break;
continue Statements	continue	continue;
return Statements	return <Expression>	return ++Var;
goto Statements	goto	goto endFunc1;
Null Statements		;

1.7 Domain-specific Extensions

1.7.1 Bit Type

FTCC supports declaration of the user-defined function and bit-type variables. The bit-type variables can be declared as global variables, static variables and function return types, but cannot be declared as local automatic variables or function parameters. In FTCC, bit-type variables cannot be initialized, and their initial values are assigned by an assignment statement. The compiler automatically assigns the bit-type variables without specified absolute address, and FTCC Linker groups eight bit-type variables in the same bank together to make one byte at a fixed address. For bit-type variables with specified absolute address, the absolute address refers to the bit address, instead of the byte address. For example:

```
sbit   FT_AT(0x50+2) sFTVar1;
```

sFTVar1 is stored at the byte address 0x2A (0x20+0x52/8) with its byte offset 0x2 (0xA2%8). To use a bit type, you need to notice:

- Bit-type variables start at 0x20, so absolute addresses are defined from the address with internal offset.
- It is not specified in standard ANSI C that how a variable is converted to a bit-type variable. FTCC assigns the lowest bit of a variable to the bit-type variable.

1.7.2 Function Definitions and Calls

1.7.2.1 Syntax Is Not Supported

FTCC is a special C compiler for embedded development applications, so it must be compatible with target hardware device characteristics. In this case, some features of ANSI C language that are rarely used in embedded development are removed. As the chip architecture upgrades, FTCC will support these syntax features in future versions.

Currently, FTCC does not support the following syntax features:

- Function pointer;
- Structures or unions as arguments cannot be passed by value to functions, or returned by value by functions. Otherwise, compile error may occur.

- The function does not support variable parameters, such as `int func(int a,...);`

1.7.2.2 Restrictions on Function Call Hierarchy

The FTCC-based MCU uses the hardware stack to store the return address of the function when it calls a function, which is different from the PC compiler (implemented by software stack). Therefore, the call depth of a function is restricted by the capacity of the hardware stack. Currently, hardware stack for FTCC-based MCUs contains only four nested layers. If the main function is added, at most five layers can be nested. The programmer must strictly restrict the maximum nesting depth as every nested layer of ISR takes up two layers of hardware stack. It is noted that such restriction is dynamic, that is, the function call depth cannot exceeds maximum nesting depth of the hardware stack at any time during operation. Otherwise, the program runs incorrectly and debugging is impossible. But the emulator reports such errors to the IDE. Similarly, due to the limited hardware stack, FTCC does not support recursive function calls, and will not report errors for user-defined recursive functions. But the Linker reports such errors to the IDE

1.7.2.3 ISR Function

FTCC-based MCUs integrate interrupt service routine (ISR) with the interrupt vector located at 0x0003. You need to configure the flag bit and enable bit for each interrupt source in the software to implement the requested interrupt service. In addition, FTCC also provides priority bit for these interrupts.

ISR functions have the form:

```
  void <ISR function name> (void) interrupt   N   //N refers to the interrupt priority and ranges from 0 to 15
{
  [Statement List]
}
```

To use an interrupt function, you need to notice:

- The interrupt function must take no parameters and return nothing (with void keyword). It must be called directly from MCU when an interrupt is generated, and cannot be called from other programs.
- A project supports up to 16 interrupt functions, which are assigned to MCU's interrupt entry address without specified absolute address;
- Interrupt protection codes and recovery codes are automatically generated by the compiler. The compiler ignores user-defined variables.
- ISR functions shall not contain complex or time-consuming operations.
- A function can be called from an ISR function, but it cannot be used in the main function.

1.7.2.4 Absolute Address

Code segments are explicitly located using the linker command line interface. FTCC allows programmers to specify absolute addresses for code segments, known as "absolute addressing". The compiler does not guarantee that the absolute address is unique in the function, i.e., when two or more functions with absolute addresses exist in a

project, the compiler will not detect whether these absolute addresses are overlapped or partial overlapped. The above operations are performed by the Linker. If overlapped or partial overlapped absolute addresses are detected, the Linker reports an error.

An absolute address of a function has the form:

```
<return type> <function name> ([argument list]) @<absolute address>
{
  [Statement List]
}
```

The "absolute address" must be an integer constant expression, that is, the compiler generates the corresponding code segment based on this expression value. Linker assigns the code segment to the corresponding ROM. The declaration and definition for an absolute address in a function shall stay consistent. Otherwise, the one in the function declaration determines the location in ROM.

For example:

```
void Func(char Val) @ 0x300; //Absolute address in function declaration.
void Func (char Val)         //Func is assigned to 0x300 in ROM.
{
    return;
}
```

1.7.3 Variable Definition

RAM in Fortior's MCU is composed of one or more pages and each page has two banks, corresponding to the first 256 bytes and the last 256 bytes of the page respectively. During compilation, the compiler maps valid variables in the program to the memory areas (i.e. IRAM & XRAM).

You can define a variable by unspecified MemMode, specified MemMode without offset and specified MemMode with offset.

1.7.3.1 Unspecified MemMode

If MemMode is not specified, it defaults to Large and the variable is assigned to XRAM by the Linker. For example,

```
int Var; //same as bank0 int Var, indicating that Var is stored in bank0 and the offset is randomly assigned by the Linker
```

1.7.3.2 Specified MemMode without Offset

After the bank number is specified, the variable will be allocated in the bank. For example,

```
data int i32Var;     //indicating that i32Var is stored in data area and the offset is randomly assigned by the Linker.
```

1.7.3.3 Specified MemMode with Offset

When defining a variable, you can specify both MemMode and the offset in the section. For example,

```
xdata FT_AT(0x2ffe) int i32Var; //indicating that i32Var is stored in XRAM with the offset 0x2ffe (byte).
```

Notes:

- If Mem does not have enough space for the variable, an error will be reported and the Linker will not store the variable in another data area.
- Any variable without xdata keyword is stored in XRAM by default.
- A variable can be assigned to a variable in different data area. The complier implemented the switch between the data area.
- The variables assigned to data area can be global variables and static variables. Otherwise, an error occurs.
- An error occurs if the pointer in data area is converted to a general pointer. For example,

```
int  i32Var = 0   //XRAM is defaulted if MemMode keyword is not written.
xdata int *pBankPt = &i32Var;
```

1.7.4 Hybrid Programming

Hybrid programming is a programming approach that groups related sets of functions together into a module to exchange data and combine different programming paradigms with the C program. Such development mode is very necessary in embedded applications. FTCC supports the following three forms of hybrid programming:

- Assembly code is embedded within a C program, for example, `__asm__ ("inline_asm_code")` or `__asm ... __endasm`
- Separated assembly files
- Relocatable library files, which are compiled by FTCC toolchain's CSLibmaker.

1.7.5 C Program Embedded with Assembly Language

In this programming mode, a line or a piece of assembler instructions are embedded in the C code. These embedded assembly codes are inserted into C code for a particular architecture. FTCC supports two types of inline assembly, `__asm__("")` and `__asm ... __endasm;`

Example 1:

```
1)  __asm__("mov a, r2");
2)  __asm__("nop\n nop");   //Multiline inline assembly uses \n for line wrapping
```

Example 2:

```
void TestFunc()
{
    __asm__("mov r2, _mcState"); __asm__("nop\nmov a, #mcCurOffset
```

```
");
    __asm      //Multi-line assembly
    movx dptr,  #_mcCtrlMode
    mov  r6, @dptr
    add a, r6
    __endasm;
}
```

1） "_" is required to add to the variable name in order to access a variable defined in C (mcState) in an embedded assembly.

2） "_" is not required to add to the variable name in order to access a variable defined in C (mcCurOffset) in an embedded assembly.

1.7.6 Separated Assembly Files

In this programming mode, one or more assembler files is (or are) added to a C program project. With corresponding naming conventions, an assembly function can be called from a C program to exchange the data, and vice versa. Since assembly names are user-defined, what you need to pay attention to is C programming symbols within an assembly after compilation: with a "_" before the symbol name. For example, int “Var” in the assembly is "_Var". This rule applies to all global symbols (including function names) in the C program, and realizes data exchange and calling assembly from C or C from assembly.

The following shows how a C program accesses assembly symbols and how an assembler accesses C symbols.

Assembly files:

```
.GLOABL ._gv1; declare as a global symbol
.GLOABL _gv2  ; declare as a global symbol
 VAR_SEC_RAM ..section data area
 _gVar1 .ds 0x2 ;  occupies 2 bytes
 _gVar2 .ds 0x4  ; occupies 4 bytes
 .ends
```

C program files:

```
extern xdata int gVar1; // _gVar1 variable declared in the assembly file
extern idata long gVar2; // _gVar2 variable declared in the assembly file
int main()
 {
     gVar1 = 5;
     gVar2 = 10; gVar2 += gVar1;
 }
```

Note: You shall pay special attention to the switch of data area when accessing global variables defined in C programs within an assembly.

```
.extern _gTest1;  declare as an external symbol (defined in the C program file as xdata char gTest1)
…
Mov r3, 0x20
banksel _gTest1;  the bank is selected by using the pseudo instruction bankse, or if it is accessed indirectly, mov r2, _gTest1 of the pseudo instruction bankisel is used
```

1.7.6.1 Relocatable Library Files

This programming mode is quite similar to that applied to separated assembly files. The C file or assembly file is compiled into a relocatable object file (*.obj file) by the C compiler or assembler, and then produced into a library file (*.lib file) by FTLibMaker.exe. The FTLibMaker.exe is stored in the IDE installation directory. The relocatable library file is added to the project for compilation. Operations: run IDE, open the project, and then select Project Setting->Linker->Libraries to link to add the *.lib file to the ListBox. Since relocatable library files are in binary form, they are very suitable for some projects with high requirements on source code security. As for the access mode, it is exactly the same as that for separated assembly files.

FTLibMaker.exe is operated as follows:

```
USAGE   : FTLibMaker.exe [Options] OPTIONS:
-lib <lib file>  output FTCC library file
-o <obj file> input FTCC object file
-help   Display available options
```

If the two *.obj files (a.obj file and b.obj file) in disk D are packaged as c.lib, the following command is used:

```
FTLibMaker.exe   -lib   d:/c.lib  -o d:/a.obj  -o d:/b.obj
```

2 FTCC High Efficiency Programming Instructions

Based on compiler design, this guide provides some programming suggestions for improving your target program.

2.1 Register

You can embed assembly code into C code because FTCC supports hybrid programming. In this case, you have to configure corresponding registers, including PSW, TIM0, TIM2, DPH, DPL. But you shall not directly use these registers as they implement specific functions when internal codes are generated. Otherwise, it may affect program operating results. If you have to use these registers, you shall backup and restore their data after they interrupted.

2.2 SRAM

Since startup code in FTCC automatically clears SRAM and initializes the stack used in the compiler and global variables, you shall not clear SRAM again during programming. Otherwise, the initialized global variables become uninitialized, causing program exceptions. You shall comply with the C language standard construct the program.

2.3 Conditional Judgment Statement

Conditional judgment statements are commonly used in the C programming. Programming conditional statements in a proper manner saves more ROM space and makes the generated assembly code more efficient.

- Write the conditional statements as simple as possible and avoid some expression operations.
- Compare global variables with global variables or with constants when compare variables on the left and right sides of the statement. For example,

```
void main()
{
    for (int i = 0; i < 100; i++)
        asm("nop");
}
//不如以下代码高效
int i;
void main()
{
    for (i = 0; i < 100; i++)
        asm("nop");
}
```

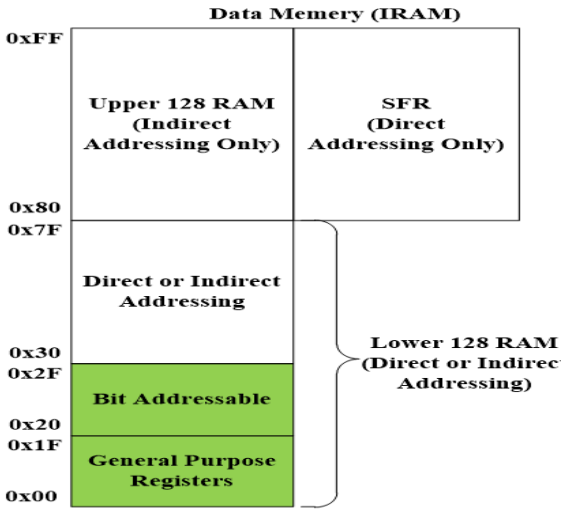
2.4 Global Variable

In FTCC, the layout of global variables is designed by programmers. You can specify bank or absolute address of global variables. In this case, the memory layout shall be properly designed to improve performances of the target program and reduce frequent code switching among data areas.

```
65 bank1 char Var;
66 bank2 char Sum;
67 void TestFunc()
68 {
69     if (Var)
70         Sum += Var;
71 }
72
73 //不如以下代码高效:
74 bank1 char Var;
75 bank1 char Sum;
76 void TestFunc()
77 {
78     if (Var)
79         Sum += Var;
80 }
```

If Var and Sum are not in the same PAGE, direct or indirect access to these variables trigger PAGE shifting, which greatly slows the target program. In this case, the adjacent codes shall not control different PAGE variables.

Fortior 8-bit IC Memory layout is shown as follows:



2.5 Function

Function is an important part of structured programming. Actually, function implementation slows down to a certain extent the object program, including parameter passing, function call, function return, etc., which are more obvious in embedded system. Therefore, function is not suggested in simple process and frequent function calls blur data stream and affect the optimizations. In addition, hardware structure of the chip restricts layers (8) of nested functions in the program. Since such kind of errors occur irregularly and are random, it is not possible to remove them completely.

2.6 Direct Addressing and Indirect Addressing

For Fortior MCUs, direct addressing shall be preferred, because it is more efficient to execute instructions and only involves statements like `mov a, Rn;`. Indirect addressing, however, involves the statements including `mov @ri, a; mov @ri, direct; mov @ri, #data;` How to distinguish direct addressing and indirect addressing in FTCC? FTCC uses direct addressing to access elements and structures of an array and

variable fields of an union. Therefore, you shall identify arrays and pointers and implement operations based on the array as much as possible, so as to optimize the target program as much as possible.

2.7 Parameter Passing

In structured programming, function arguments are commonly used. However, you shall not use too many arguments in an embedded system, otherwise it greatly affects the performance of the program. In principle, it is not necessary to use function arguments, or it is better to use no more than four of them. When four arguments or even more are indeed necessary, you shall decompose and reconstruct the function. If argument list of a function is too complicated and cannot be decomposed, you shall use a pointer to as structure as a parameter. For example,

```
15 void TestFunc1(int p1, long p2, unsigned char p3, unsigned long)
16 {
17 }
18
19 //更改为以下:
20
21 struct S_Param
22 {
23     int p1;
24     long p2;
25     unsigned char p3;
26     unsigned long p4;
27 };
28 void TestFunc2(struct S_Param * psParam)
29 {
30     ...
31 }
```

2.8 Return Value Passing

Return value passing is similar to parameter passing. Program's performance is not affected when return value is smaller than or equal to two bytes. The return value greater than two bytes shall be avoided. Besides, the compiler reports an error if structures and unions are returned from a function using the return value.

2.9 Data Type

Fortior MCU has an 8-bit CPU. It is very important to choose a reasonable data type for the variable, otherwise it may cause type promotion due to different data types with unnecessary overhead costs to the target program.

For example, when binary operation is performed for signed variables, the variables shall be of the same type. Otherwise, type promotion is required and the signed bit expansion causes an obvious increase in the code size. For example,

```
16  long Sum;
17  char Value;
18  void TestFunc1()
19  {
20      Sum += Value;
21  }
```

It is better to change Sum to int and even better to char, as long as the data does not overflow the boundary. The principle of "small code size first and unsigned number first" shall be followed.

2.10 Explicit Type Conversion

Explicit type conversion, like "`Value=(long)(int)Temp;`", shall be avoided. Although syntactically correct, it reduces efficiency of the object code and is therefore discouraged.

2.11 `if` Expression

Using bit variables for `if` expression evaluation is more efficient than other types of variables. For example,

```
23  sbit bJudge;
24  char nJudge;
25  char Value;
26  char Sum;
27  void TestFunc1()
28  {
29      if (bJudge)
30          Sum += Value;
31  }
32
33  //比以下代码更高效：
34
35  void TestFunc2()
36  {
37      if (nJudge)
38          Sum += Value;
39  }
```

One instruction is required for determining a bit variable within FTCC, while multiple statements and even function calls for other variables. It is strongly recommended to use a bit variable for if expression evaluation.

2.12 Constant

Constants in FTCC are defined as `const int C_VALUE = 10`. Since FTCC stores global constant arrays and string constants in ROM and other constants in RAM. However, whether in RAM or ROM, it requires space, so a predefined way is preferred. For example,

`#define C_VALUE 10。`

In this way, both RAM and ROM are not required.

2.13 Pointer

C language provides a very flexible and effective pointer mechanism, FTCC inherits this feature of ANSI C. There are two kinds of pointer in FTCC, one for RAM and the other for ROM. For example,

```
const char *p = "abcde"; //p is a pointer for ROM
char ary[10];
char *p2 = ary; //p2 is a pointer for RAM
p = p2;          //Compiler error. FTCC does not support assigning a pointer for RAM to a pointer for ROM.
```

Programmers shall be aware of this when using pointers. The compiler reports an error if a pointer for ROM is assigned to a pointer for RAM. In addition, using pointers generally involves indirect access, which sends several instructions more than direct access. It has a certain impact on the execution efficiency of the target program, so using pointers in large numbers is not preferred.

2.14 Judgement Statement

```
55  if(CntIn_Cur-CntIn_Pre<Cnt_Filter_Slower)
56  {
57      ....
58  }
59
60  //不如以下代码高效：
61  int Temp = CntIn_Cur-CntIn_Pre;
62  if(Temp < Cnt_Filter_Slower)
63  {
64      ....
65  }
```

The results of `CntIn_Cur-CntIn_Pre` are stored in the compiler's internal stack which are acquired via indirect access. If a temporary variable is used to hold the result of `CntIn_Cur-CntIn_Pre` and then compare it with `Cnt_Filter_Slower`, no indirect access is required and therefore it is more efficient. The more the judgement statements, the lower the efficiency. Therefore, simple judgement statements are preferred.

2.15 Local Array Variable

Local array variables shall not be defined and initialized, because they will double RAM size. If a local array with two elements is defined, four elements are required. The extra two elements are the internal private global variables used to initialize the local array.

2.16 Bitwise (&) and Modulus (%)

Generally, "&" operation is easier than "%" operation. In certain special conditions, replacing "%" with "&" reduces task overhead.

```
char i,j;
void main(void)
{
    j = i%8;
}
//不如以下代码高效：
char i,j;
void main(void)
{
    j = i&7;
}
```

2.17 Switch Statement and if else Statement

Convert `if else` statements to `switch` statements in nested functions to save ROM if possible.

```
unsigned char i, j;
void main(void)
{
    if(j == 0) i = 0;
    else if (j == 1) i = 1;
    else if (j == 2) i = 3;
    else if (j == 3) i = 5;
    else if (j == 4) i = 2;
    else if (j == 5) i = 9;
    else if(j == 6) i = 7;
    else if(j == 7) i = 4;
    else if(j == 8) i = 8;
}

//不如以下代码高效：
unsigned char i, j ;
void main(void)
{
    switch (j) {
    case 0: i = 0;break;
    case 1: i = 1;break;
    case 2: i = 3;break;
    case 3: i = 5;break;
    case 4: i = 2;break;
    case 5: i = 9;break;
    case 6: i = 7;break;
    case 7: i = 4;break;
    case 8: i = 8;break;
    }
}
```

2.18 Loops

When there are many repeated operations in the program, and they are regular, you can use a loop instead.

```
unsigned char show_data[6];
unsigned long value;
void main()
{
    show_data[5]= value%10;
    show_data[4]= value/10%10;
    show_data[3]= value/100%10;
    show_data[2]= value/1000%10;
    show_data[1]= value/10000%10;
    show_data[0]= value/100000%10;
}

//不如以下代码高效：
unsigned char show_data[6];
unsigned long value;
void main()
{
    unsigned long temp=value;
    unsigned char i;
    for(i=6;i>0;)
    {
        i--;
        show_data[i]=temp%10;
        temp/=10;
    }
}
```

2.19 Frequently Used Code

If there is a piece of code that is used more than once in a program, it shall be encapsulated as a separate function when possible to reduce instructions.

2.20 Function Body

Functions shall not be too complex. Generally, they shall not exceed 100 lines long. In this case, it not only enhances readability, but also facilitates the optimization of RAM.

2.21 Interrupt Service Routine

Generally, if two functions are not related to each other, their local variables can be assigned with the same address, but the interrupt server does not share the address of local variables with the main function. Therefore, in order to reduce RAM, interrupt service routine shall be as simple as possible, and not be too complex.

3 Common Errors and Solutions

3.1 Error<"sbit variable must be global or static.">;

- Example:

```
20 void main()
21 {
22     sbit WDTCON1;
23 }
```

- Compile error:

```
\main.c:22:6: error: sbit variable must be global or static.
```

- Note: The sbit variable must be global or static variables.

3.2 Error<"sbit variable can't be initialized.">;

- Example:

```
21 sbit s1=1;
22 void main()
23 {
24
25 }
```

- Compile error:

```
\main.c:21:8: error: sbit variable can't be initialized.
```

- Note: The sbit variable cannot be initialized.

3.3 Error<"invalid use of 'sbit'.">;

- Example:

```
20 sbit a[5];
21 void main()
22 {
23
24 }
```

- Compile error:

```
\main.c:20:9: error: invalid use of 'sbit'.
```

- Note: The sbit variable cannot be array variable.

3.4 Error<"auto variable '%0' should not be bank qualified.">;

- Example:

```
20 void main()
21 {
22     bank1 char c1=9;
23 }
```

- Compile error:

```
\main.c:22:13: error: auto variable 'c1' should not be bank qualified.
```

- Note: The local variable cannot be defined with `bankx`.

3.5 Error<"struct member don't allowed sbit type.">;

- Example:

```
21 struct s1
22 {
23     char c1;
24     sbit s1;
25 }
26 void main()
27 {
28
29 }
```

- Compile error:

```
\main.c:24:9: error: struct member don't allowed sbit type.
```

- Note: The structure member does not support the sbit type.

3.6 Error<"invaild use of absolute address.">;

- Example:

```
21 void main()
22 {
23     char c1 @0x80;
24 }
```

- Compile error:

```
\main.c:23:7: error: invaild use of absolute address.
```

- Note: The local variable cannot be specified with an address.

3.7 Error<"0 is illegal bit-field size.">;

- Example:

```
21 struct s1
22 {
23     char c1:0;
24     int i1;
25 }
26 void main()
27 {
28
29 }
30
```

- Compile error:

```
main.c:23:11: error: 0 is illegal bit-field size.
```

- Note: A bit length of 0 is invalid.

3.8 Error<"only bank0 variable can be initialized.">;

- Example:

```
21  bank1 char b1=10;
22  void main()
23  {
24
25  }
```

- Compile error:

```
main.c:21:14: error: only bank0 variable can be initialized.
```

- Note: Only bank0 global variables can be initialized, and bank1~bankx global variables cannot be initialized.

3.9 Error<"invalid bank keyword.">;

- Example:

```
20  bank char c=0;
21  void main()
22  {
23
24  }
```

- Compile error:

```
\main.c:20:1: error: invalid bank keyword.
```

- Note: The keyword must be bank0, bank1, bank2,..... , bank13.

3.10 Error<"not support function point now.">;

- Example:

```
22  int func(int x); /* 声明一个函数 */
23  int (*f) (int x); /* 声明一个函数指针 */
24
25  void main()
26  {
27      f=func; /* 将func函数的首地址赋给指针f */
28  }
```

- Compile error:

```
main.c:23:17: error: not support function point now.
```

- Note: It doesn't support the function pointer.

3.11 Error<"structures or unions parameters can't passed by value.">;

- Example:

```
20 struct s1
21 {
22     char c1;
23     int i1;
24 };
25 union u1
26 {
27     char d1;
28     int f1;
29 };
30 void fun(struct s1 t1,union u1 t2)
31 {
32
33 }
34 void main()
35 {
36
37 }
```

- Compile error:

```
\main.c:30:20: error: structures or unions parameters can't passed by value.

\main.c:30:32: error: structures or unions parameters can't passed by value.
```

- Note: The argument cannot be structure or union, but can be replaced by the pointer of the structure or union.

3.12 Error<"structures or unions can't be a return value from a function.">;

- Example:

```
20 struct s1
21 {
22     char c1;
23     int i1;
24 };
25 union u1
26 {
27     char d1;
28     int f1;
29 };
30 struct s1 fun1()
31 {
32
33 }
34 union u1 fun2()
35 {
36
37 }
38 void main()
39 {
40
41 }
```

- Compile error:

```
\main.c:35:1: error: structures or unions can't be a return value from a function.
```

- Note: The structure or union cannot be a return value from a function, but can be replaced by the pointer of the structure or union.

3.13 Error<"variable locate can't greater than 0xff.">;

- Example:

```
21  char a1 @0x100;
22
23  void main()
24  {
25
26  }
```

- Compile error:

```
\main.c:21:15: error: variable locate can't greater than 0xff.
```

- Note: The address specified by the variable cannot exceed 0xff or IC limit.

3.14 Error<"the chip can't support global variable initialization, in addition to the constant array.">;

- Example:

```
21  char a1 = 0;
22
23  void main()
24  {
25
26  }
```

- Compile error:

```
)\main.c:21:10: error: the chip can't support global variable initialization, in addition to the constant array.
```

- Note: The chip does not support Movp instructions, and local array cannot be initialized.

3.15 Error<"the chip can't support local array variable initialize.">;

- Example:

```
23  void main()
24  {
25    char a1[3] ={1,2,3};
26  }
```

- Compile error:

```
\main.c:25:8: error: the chip can't support local array variable initialize.
```

- Note: The chip does not support Movp instructions, and local array cannot be initialized.

3.16 Error<"variable length array declaration not allowed at local scope.">;

- Example:

```
23 void main()
24  {
25  int a=9;
26  char a1[a];
27 }
```

- Compile error:

```
\main.c:26:7: error: variable length array declaration not allowed at local scope.
```

- Note: The length of an array variable cannot be defined with a local variable.

3.17 Error<"array byte sizes exceeds the maximum of 127 for this chip.">;

- Example:

```
22  char a1[128];
23 void main()
24  {
25
26 }
```

- Compile error:

```
\main.c:22:7: error: array byte sizes exceeds the maximum of 127 for this chip.
```

- Note: The length of the array cannot exceed 127 bytes.

4 Revision History

Rev.	Description	Date	Prepared By
V1.0.1	First release, translated from Chinese version V1.0.1.	2021/08/11	Eric Deng

Copyright Notice

Right to make changes — Fortior Technology (Shenzhen) Co., Ltd. reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. The information contained in this manual is provided for the general use by our customers. Our customers shall ensure that they take appropriate action so that their use of our products does not infringe upon any patents. It is the policy of Fortior Technology (Shenzhen) Co., Ltd. to respect the valid patent rights of third parties and not to infringe upon or assist others to infringe upon such rights.

If there are any differences between the Chinese and the English contents, please take the Chinese version as the standard.

Fortior Technology (Shenzhen) Co., Ltd.

Room 203, 2/F, Building No.11, Keji Central Road 2,
Software Park, High-Tech Industrial Park, Shenzhen, P.R. China 518057
Tel: 0755-26867710
Fax: 0755-26867715
URL: http://www.fortiortech.com

Contained herein