

KF8CC User Attention Manual

V1.3

Contents

CONTENTS.....	2
1 OVERVIEW.....	4
2 KEYWORDS AND RESERVED WORDS.....	4
3 BIT TYPE.....	5
4 DATA TYPE.....	6
5 CHARACTER ARRAY LIMIT.....	7
6 GLOBAL POINTER LIMIT.....	8
7 ARRAY ACCESS.....	8
8 GLOBAL VARIABLE AND ADDRESS ASSIGNMENT.....	8
9 VOLATILE KEYWORD.....	9
10 STATIC KEYWORD [NOT SUPPORT IN DEBUG MODE].....	9
11 RECURSIVE FUNCTION [NOT SUPPORTED].....	9
12 FUNCTIONS ARE NOT REENTRANT.....	10
13 INLINE FUNCTIONS ARE NOT SUPPORTED.....	11
14 FUNCTION POINTERS ARE NOT SUPPORTED.....	11
15 FUNCTION PARAMETERS AND LIMITATIONS.....	11

16 ASSEMBLY NESTING.....	12
17 INTERRUPT FUNCTION.....	14
18 TYPE CONVERSION.....	14
19 HEADER FILE LIMITS [VARIABLE DEFINITION LIMITS].....	14
20 DEBUGGING AND COMPILING.....	15

1 Overview

The kf8cc compiler provides a C language program development platform for KF8 series single-chip computers. This document briefly explains the matters needing attention in the use of kf8cc, so that users can quickly master how to use kf8cc to develop programs.

2 Keywords and Reserved Words

Standard C reserved word

Table 2-1

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

Kf8cc reserved word

Table 2-2

at	_at	__at	code	_code
__code	critical	__critical	__critical	idata
_idata	__idata	data	_data	__data
interrupt	_interrupt	__interrupt	sfr	_sfr
__sfr	_using	__using	__using	

Kf8asm reserved word

Table 2-3

high	low			
------	-----	--	--	--

The kf8cc reserved word is used as a function extension, and the user-defined identifier cannot have the same name as it.

The C development does not support the symbols high and low to locate the high status of data when establishing structures and consortia (debugging does not support it). Such as

```
typedef struct{
    unsigned char high;
    unsigned int low;
}test;
```

This definition release passed compilation, but debug compilation failed, that is, syntax .def high, recognized as an internal function, high is normal in assembly code, For example: MOV R0,#high(a) MOV R0,low(a)

3 Bit Type

The current version of kf8cc does not implement the bit type, but kf8cc can use the bit field of standard C to implement the bit operation.

For example:

```
typedef struct
{
    unsigned char b0:1;
    unsigned char b1:1;
    unsigned char b2:1;
    unsigned char b3:1;
    unsigned char b4:1;
    unsigned char b5:1;
    unsigned char b6:1;
    unsigned char b7:1;
}BitStruct;
BitStruct FlagColumn;
#define      flag0      FlagColumn.b0
#define      flag1      FlagColumn.b1
#define      flag2      FlagColumn.b2
#define      flag3      FlagColumn.b3
#define      flag4      FlagColumn.b4
#define      flag5      FlagColumn.b5
#define      flag6      FlagColumn.b6
#define      flag7      FlagColumn.b7
```

This defines 8 bits. You can also use type definitions in the header file system.

```
typedef struct
{
    unsigned char _0:1;
    unsigned char _1:1;
    unsigned char _2:1;
    unsigned char _3:1;
    unsigned char _4:1;
    unsigned char _5:1;
    unsigned char _6:1;
    unsigned char _7:1;
} sfr_bits;
```

It is suggested that interrupts need to be separated from the status marks in the main program, that is, defined in different byte objects.

4 Data Type

Data Type	Bit length
char	8
unsigned char	8
short	16
unsigned short	16
int	16
unsigned int	16
long	32
unsigned long	32
float	32

The immediate data display type expression:

```
// [unsigned long]T2_cnt =PWM5L1<<8u; // right
// [unsigned long]T2_cnt +=PWM5L0;

// [unsigned long]T2_cnt =PWM5L1<<8; // Error
// [unsigned long]T2_cnt +=PWM5L0;
```

That is, immediate numbers support type modification: U is unsigned, F is float, and L is long. UL is unsigned long.

The above error analysis: PWM5L0 unsigned char operates signed int 8, that is, the immediate number is identified as signed, so the result type of PWM5L0 << 8 is promoted to signed int. When converting to the unsigned long of the result, the data symbol is identified according to the high order of the result, that is, bit15, and the high 2 bytes of unsigned long are assigned to 0 or 0xFF according to the symbol, that is, when PWM5L1 is greater than or equal to 128, the T2_cnt value is wrong, increasing 0xFFFF0000.

kf8cc supports structures and complexes, and only structures support initial values. Reference examples are as follows:

```
typedef struct //Custom protocol structure{
    uint8 id;
    uint8 type;
    uint8 product[6];
    uint8 index[5];
    uint8 reservation[DATAFLASH_BUFFER_BYTE_SIZE-13]; //Use byte number
    parameter, such as 13 here
}s_Dataflash_ConstBuffer;
const s_Dataflash_ConstBuffer __at(0x1800) appUserData={
    .id=8,
    .type=1,
    .product={1,2,4,5,6,7,8,9,10},
    .index="abce\0",
```

```
    // .index={'a','b','c','e'},  
    .reservation={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,}  
}; // It should be noted that constants can only store 1 byte in per flash  
space word, therefore it cannot be transferred directly to RAM variable  
space equivalently.
```

If you need to define multiple structures of the union, you can define the content structure separately, that is, assign initial values in the sense of the content structure object. The union object or parameter is responsible for writing the difference code. The following union is defined:

```
typedef union  
{  
    unsigned char bytes[DATAFLASH_BUFFER_BYTE_SIZE]; // Pure array size  
    cache without protocol structure  
    struct // Custom protocol structure service const object{  
        uint8 id;  
        uint8 type;  
        uint8 product[6];  
        uint8 index[5];  
        uint8 reservation[DATAFLASH_BUFFER_BYTE_SIZE-13]; // Use  
        byte number parameter, such as 13 here  
    };  
} s_Dataeeprom_RAMBuffer; // read-write sample support bound to const,  
number type prompt, still assign value by byte, leaving the high bit blank  
to realize 0xB0 placeholder of the high bit CRET of the original read Flash
```

5 Character Array Limit

kf8cc already supports string arrays.

The current version of kf8cc uses the following format for array initialization:

```
char ch[]={ '0' , '1' , '2' , '3'  };
```

```
Or char ch[]={ "0123" };
```

It should be noted that 0 will be placed at the end of the string, but the string does not support the interval multi-expression of \0, such as `char ch[]={ "01\023" }`; At this time, the failure string is only the initial 0x30 0x31 0x00 of length 3.

When the string is located in the structure data type, it should strictly meet its defined length at the beginning. That is, it can't be more and less, otherwise there is an error offset in the initialization of the whole structure data.

6 Global Pointer Limit

Kf8cc supports pointers, but does not support global pointer initialization.

```
unsigned char *p=&var; Error
```

The compilation result discards the initial value, that is, the exception that the address of p is 0x000000. The correct write global definition is `unsigned char * p;`. In the startup initialization function `p = & var.`

7 Array Access

Considering the bank characteristics of kf8 series single chip microcomputer, any non-const type array shall not occupy more than one bank, such as char type array, its subscript value shall not exceed 128, int type data, its subscript value shall not exceed 64.

When using arrays, use unsigned char type arrays and unsigned char type subscripts whenever possible.

When evaluating array subscript through expression, the user needs to confirm whether the value of the expression exceeds the limit value of the expression type, otherwise it will cause overflow error.

kf8cc supports one-dimensional and two-dimensional values. The efficiency of two-dimensional is low than one-dimensional. It is recommended to use a one-dimensional array. For two-dimensional needs, it can be realized by displaying expressions, such as `AA[i][j] = 0x05`, which can be written as `AA[i*Maxj+j]`. For continuous code operations of arrays, subscript positioning can be written in front as an independent variable. E.g. `u8b = i*Maxj+j; AA [b] = AA[b+1];` It is more efficient than `AA[i*Maxj+j] = AA[i*Maxj+j+1]`.

8 Global Variable and Address Assignment

kf8cc supports constant and non-constant global variables and initialization upon declaration.

Constant variables can only be declared globally, that is, constant variables cannot be created inside functions. Otherwise, it will lead to wrong results.

The declaration of variables will automatically allocate RAM space in C language. If necessary, you can specify the address by adding it after

the type and before the name. E.g. `unsigned char __at (0x83) a=4;` That is, the C variable is established, and the RAM address of a is 0x083. If a is required to point to zone 3 of the RAM partition, it can be written as `unsigned char __at (0x383) a = 4;` If it is a constant variable, it can be written as `const unsigned char __at (0xF70) a=4;.`

The variable can only be created in C type file for kf8cc. If other files need to be used, it can be declared in its corresponding H file or total H file, that is, `extern unsigned char __at (0x83) a;.`

9 volatile Keyword

In general, volatile modification is not required. If the variable is used in both interrupt and main loop, the keyword modification needs to be added, that is the change relationship of two independent branches is uncertain, and the code expression involving the variable should not use the optimized recognition result.

10 static Keyword [Not Support in Debug Mode]

kf8cc supports static keyword.

static has the following uses:

- 1) Modifies a function to indicate that the function is inaccessible outside the C file in which it is defined.
- 2) Modifies a global variable to indicate that it is inaccessible outside the C file in which it is defined.
- 3) Modifies a local variable to indicate that the variable is located in static memory. This variable still exists after the function exits.
- 4) When static modifies local variables, global variables can be used instead. It is not recommended to use static to modify the function. This qualification is for the information exception of debugging compilation, which will be recognized by compilation. If only file function are not declared externally.

11 Recursive Function [Not Supported]

kf8cc does not support recursive functions because local variables are unique. KF8 series chip PC pointers cannot be nested beyond the hardware stack size through the hardware stack saving program. The

hardware stack size is divided into 8 levels and 16 levels.

Therefore, the function cannot be called in both high and low interrupt code. Nor can it be called at the same time as the main program in high-level interrupt or low-level interrupt. The default conflict kf8cc will report an error and give a text prompt. When the linker uses -z option, it will give a warning prompt for possible function conflicts.

12 Functions are not Reentrant

The code generated by the kf8cc compiler is not reentrant.

Because the function is not reentrant, the following operations will cause problems:

- 1、 Call function external function in interrupt.
- 2、 Perform *, \, %, and pointer operations in interrupts.
- 3、 Parameter Passing of Interrupt Function.

This is not recommended if you want to call a function in an interrupt. Refer to the Function Usage Restrictions section.

- 1、 Call a function without parameters (parameters can be passed through global variables)
- 2、 Manually save the pass-through stack

Note: This function cannot be called outside the interrupt. When calling outside, it needs to enter the critical section (close the interrupt).

Example:

- 1、 Call the function with parameters (interrupt the external call)

```
void ISR() __interrupt
{
    if(T0IF)
    {
        PUSH_STK();//Save the pass-through stack

        //Interrupt content
        //You can call functions with arguments, but functions cannot be called
outside interrupts
        fun(arg0,arg1,.....);

        POP_STK();//Restore the pass-through stack
    }
}
```

The protection stack function should be defined by itself, the number of

function parameters shall prevail, and R0, STK00, STK01... shall be used for parameter passing in turn.

2、Call function with parameters (interrupt external call)

```
void ISR() __interrupt
{
    if(T0IF)
    {
        PUSH_STK();//Save the pass-through stack

        //Interrupt content
        //You can call functions with parameters, but you need to turn off the
interrupt when interrupting external calls
        fun(arg0,arg1,.....);

        POP_STK();//Restore the pass-through stack
    }
}
```

```
.....
SAIE=AIE;          //Save interrupt master control bit
AIE=0;             //Close interrupt
fun(arg0,arg1,.....);
AIE=SAIE;          //Resume interrupt master control bit
.....
```

This method is also suitable for parametric functions. It is suggested that for functions used both externally and interruptedly internally, only the functions should be written twice and different contents should be named, that is they should be used independently.

13 Inline Functions are not Supported

14 Function Pointers are not Supported

15 Function Parameters and Limitations

The length of kf8cc function parameters is limited, and the length

of all function parameters cannot exceed 13 bytes. Parameters use R0, STK00, STK01, STK02, STK03 ... STK11 in turn according to the number of bytes used. The interrupt stack does not protect the STKxx variable. The matters needing attention please refer to the non-reentrant section of the function for precautions.

When there is only one parameter of type char or uchar, only R0 is used to pass the parameter, so there is no need to consider the interrupt call. If it is int, R0 and STK00 are used, where R0 stores the high bit of the parameter. If it is long type or two int types, R0, STK00, STK01 and STK02 are used in sequence. It should be noted that the low byte uses large encoded variables. For example, int type uses STK04 and STK05, and STK04 stores the high byte content of variables.

The return of the function also uses this parameter, and the rule is the same. That is the result can be considered according to the parameter. According to the number of bytes to be occupied, the content is cached to the corresponding variable.

kf8cc does not support automatic backup establishment of functions under simultaneous use of interrupt and external, that is this usage will generate errors (when the process variable is used, the process variable is interrupted by interrupt, the process variable is reused in the interrupt operation, the process variable of the external code at the end of the interrupt has been changed, and subsequent code execution will be wrong).

It is not recommended to use static to modify functions, which is not supported by debugging and compilation.

Note: Multiplication, division, floating point and pointer are all implemented in the form of libraries, which are essentially functions, so these operations cannot be stored in interrupts, including functions called by interrupts. The division external main program is not used, but the stack variable protection passed by the external function call parameters needs to be considered, as explained in the function parameters section.

16 Assembly Nesting

The kf8cc assembly nesting syntax is:

```
__asm  
Assembly code  
__endasm;
```

When accessing registers and global variables in assembly code, it needs to underscore "_" before the variable name. Embedded assembly does

not support the use of local variables.

Example:

```
__asm
Label:
    MOV R0, _P0
    MOV _a, R0
    MOV R0, _P1
    MOV _P0, R0
    MOV R0, _a
    MOV _P1, R0
    JMP $-6 //Or JMP Label
__endasm;
```

Note:

- 1、Label can be used for embedded assembly, and ":" needs to be added after the label.
- 2、It is recommended that the embedding assembly use the R register using only R0 and R1, and R6 and R7 can be used if desired. R2~R5 are used by the compiler to push the stack out of the interrupt protection site. Unless the interrupt is turned off, R2~R5 cannot be used in embedded assembly code.
- 3、According to the embedded assembly expression of the defined global variables, the variables have partition information. The compiler will optimize the slice as required, but it must be optimized based on pseudo instructions. Examples are as follows:
(unsigned int) var++;

```
__asm
    BANKSEL _var
    INC _var
__endasm;
```

- 4、For types with MOVP instructions, the embedded assembly call function also needs to be explained by pseudo instructions. There is a compiler for the actual implementation of cutting code. Examples are as follows:

```
Fun(a, b);
__asm
    ; Pass parameters a and b to the parameter stack. See Function
    Parameters.
    PAGESEL _Fun
    CALL _Fun
    PAGESEL $
__endasm;
```

- 5、If the function has a return content, the use of the register type parameter in which the return result is explained in the section of Function Parameters.

17 Interrupt Function

The format of the kf8cc interrupt function is as follows:

```
void int_fun0() __interrupt (0)
void int_fun1() __interrupt (1)
```

The function name can be replaced by any legal identifier that users like.

Where zeros and ones in parentheses represent high-level interrupt entry 0x04 and low-level interrupt entry 0x14, respectively.

It should be noted that the C project code does not need to write its own stack pressing instruction, and the compiler will automatically process it. Push the stack will save the contents to R2~R5 and the established global variables. That is to say, except to the written content, the interrupt function code also has additional code for pushing and popping the stack. If the function is called, the parameters used need to be pushed or popped the stack by manual code.

18 Type Conversion

For example:

```
unsigned char Result = 0;
unsigned char FristNum ;
unsigned cahr SecondNum;
Result = FristNum * SecondNum;
```

If the multiplication result of two numbers is greater than 255, the multiplier and multiplicand types need to be converted to int or long, that is type conversion is required, and the modification adopts the expression form of the type in parentheses. For example:

```
unsigned int Result = 0;
unsigned char FristNum ;
unsigned cahr SecondNum;
Result = (unsigned int)FristNum * (unsigned int)SecondNum;
```

19 Header File Limits [Variable Definition Limits]

Header files can only have variable or function declarations. That is, variables can only be declared in the source file. If variables need to be used externally, only extern modification needs to be added to the header file for reference, and other files contain the header file.

20 Debugging and Compiling

KF8 series single chip microcomputer debugging uses software debugging, that is additional monitoring code needs to be added during debugging and compilation, and certain RAM resources also need to be occupied. Generally, debugging code occupies less than 255 sentences and RAM occupies less than 16 bytes.

For C project, RAM usage does not need to be considered, that is debug variables are equivalent to global variables, and there is no conflict in address allocation, except when variable addresses are specified.

When using assembly projects, pseudo-instruction writing method of applying variables can be adopted. More assembly development variables are defined as RAM addresses. When using debugging compilation, RAM occupied by debugging code should be considered, and code variables should not coincide with these addresses. Because of the existence of debug variables, debug compilation will prompt “warning: call _ cinit at startup to clean bss section.”, unless you use a pseudo instruction declaration that applies for variables and assigns initial values, you don’t need to consider calling. There are differences in debugging RAM resources used by different series of single chip computers. The details can be viewed through the compiled suffix map file. In the Symbols section, the variable is prefixed with “ICD” and used in the XXX_monitor.asm file.