

KF32 系列

ChipON IDE C 语言开发手册 V1.1

2024 年 1 月

目录

目录	2
1. 工具链基础知识	6
1.1 简介	6
1.2 文件命名约定	6
1.3 数据存储	6
1.3.1 字节序	6
1.3.2 整型类型与范围	7
1.3.3 默认字节 <i>char</i> 变量的符号说明	7
1.3.4 浮点类型与长度	7
1.3.5 指针类型与长度	7
1.4 属性	7
1.4.1 <i>Interrupt</i> 中断函数	7
1.4.2 <i>section</i> (“name”) 指定类型段	8
1.4.3 <i>packed</i> 对齐	8
1.4.4 常见属性	8
1.5 命令行格式	9
1.5.1 设置环境变量	9
1.5.2 命令行选项	9
2 语言与库说明	10
2.1 简介	10
2.2 实现定义的限制<LIMITS.H>	11
2.3 C 常规精简库方法	11
2.3.1 <i>stdio</i> 串口输出库函数	11
2.3.2 <i>math</i> 数学运算库函数	12
2.3.3 <i>string</i> 字符串处理库函数	13
2.3.4 类型判断库函数	14
2.3.5 系统标准库函数	15
2.4 浮点库函数及 64 位整数库函数	18
2.4.1 浮点类型加减法	18
2.4.2 浮点类型乘除法	18
2.4.3 浮点类型比较运算	19
2.4.4 浮点类型转换	20
2.4.5 整型 64 位库函数	21
2.5 可变参数列表<STDARG.H>	22
2.5.1 <i>va_arg</i>	23
2.5.2 <i>va_end</i>	23
2.5.3 <i>va_list</i>	23

2.5.4	<i>va_start</i>	23
3	中断函数	24
3.1	简介	24
3.2	中断处理函数	24
3.2.1	中断函数现场保护	24
3.2.2	中断处理程序	24
3.2.3	中断向量配置	25
3.2.4	中断使用的变量	25
4	特殊功能寄存器的操作	25
4.1	简介	26
4.2	特殊功能寄存器的操作	26
5	编译器运行时约束	27
5.1	寄存器约定	27
5.2	堆栈使用	28
5.3	函数调用	29
5.4	混合语言编程	29
5.5	特殊代码时间消耗控制	29
6	语言特殊开发与说明	31
6.1	简介	31
6.2	C 语言代码	31
6.3	RAM 函数	31
6.4	链接脚本与修改	32
6.5	嵌汇编	34
6.5.1	普通字符串形式	35
6.5.2	GNU 格式的特殊汇编，支持传递参数	35
6.5.3	嵌汇编函数	36
6.5.4	嵌汇编注意事项	36
6.6	汇编与嵌汇编控制与优化	37
6.6.1	指定代码段属性保存位置	37
6.6.2	指定程序在 RAM 中运行	38
6.6.3	特殊伪指令 EQU 定义变量	38
6.6.4	特殊伪指令 LD Ra, #label	38
6.6.5	特殊伪指令 MOV Ra, #data	39
6.6.6	相对跳转 SJMP/LJMP/JMP	39
7	ROM 固有函数库与 BOOTLOADER 支持方法	40
7.1	函数声明与调用	40
7.2	IAP 可实现功能介绍	41
7.2.1	将 RAM 内容复制到 Flash	42
7.2.2	擦除扇区	43

7.2.3 扇区查空 <i>Flash</i>	43
7.2.4 比较 <i>Flash</i> 和 <i>RAM</i>	44
7.2.5 读 <i>FLASH</i> 信息区	44
7.2.6 将 <i>RAM</i> 内容复制到 <i>Flash</i> 配置区	44
7.3 API 应用接口函数介绍	45
8 芯片中断向量表实现与说明	45
8.1 C 语言格式框架示例摘录:	46
8.2 汇编语言格式示例:	47
9 更新记录	48

在使用本手册前，请您认真阅读以下使用许可协议。只有在同意以下使用许可协议的情况下，方能使用本手册中介绍的产品。

版权公告

未经上海芯旺微电子科技有限公司书面允许，任何公司、个人不得以任何形式复制本使用手册的全部或部分内容。

重要声明

上海芯旺微电子科技有限公司努力使本手册中提供的信息准确和适用，然而，产品及手册可能包括技术或印刷上的错误。上海芯旺微电子科技有限公司保留在不事先通知的情况下改变本使用手册全部或部分内容的权力。

1. 工具链基础知识

1.1 简介

C 编译器（kf32-gcc）对 C 语言模块文件进行编译；汇编器（kf32-as）对汇编语言模块文件进行汇编；链接器（kf32-ld）对目标文件及库文件进行链接输出程序映像 elf 文件等，可选项生成 HEX 文件。kf32-objcopy 实现程序目标文件 hex、bin 或 s19 格式的补充输出。kf32-objdump 实现程序的反汇编 lst 文件的输出，其提供参数接口可扩展反汇编的段配置。[\[该文档中描述仅适合 Chip0N 精简指令集编译器\]](#)。

1.2 文件命名约定

KF32 编译工具链识别如下文件扩展名：

表 2-1 文件名

扩展名	定义
*.c	C 语言源文件。
*.asm	汇编语言源文件。
*.h	C 语言头文件。
*.inc	汇编头文件。
*.i	已经过预处理的 C 源文件。
*.s	C 编译生成的汇编文件。
*.o	目标文件。
*.elf	ELF 文件（可执行可链接文件）。
*.map	MAP 文件。
*.lst	Objdump 工具基于 elf 文件输出的反汇编文件。
*.hex	Objcopy 工具基于 elf 文件输出 Intel HEX 程序文件。
*.s19	Objcopy 工具基于 elf 文件输出摩托罗拉格式程序文件。
*.bin	Objcopy 工具基于 elf 文件输出二进制程序文件。
*.cpp *.C	C++语言源文件。
其他	适配工具的文件，如 C++头文件、链接器脚本 ld 文件，如 mk 的 makefile 等。

1.3 数据存储

1.3.1 字节序

KF32 编译器以小端字节序格式存储数据。最低有效字节存储在最低地址。

例如，32 位数值 0x12345678 在地址 0x10000000 处开始，按如下格式存储：

地址	0x10000000	0x10000001	0x10000002	0x10000003
----	------------	------------	------------	------------

数值	0x78	0x56	0x34	0x12
----	------	------	------	------

1.3.2 整型类型与范围

KF32 编译器中的整型范围如下：

表 2-2 整型类型

类型	位	字节	最小值	最大值
signed char	8	1	-128	127
char,unsigned char	8	1	0	255
short,signed short	16	2	-32768	32767
unsigned short	16	2	0	65535
int,signed int	32	4	-2^{31}	$2^{31}-1$
unsigned int	32	4	0	$2^{32}-1$
long,signed long	32	4	-2^{31}	$2^{31}-1$
unsigned long	32	4	0	$2^{32}-1$
long long,signed long long	64	8	-2^{63}	$2^{63}-1$
unsigned long long	64	8	0	$2^{64}-1$
enum	32	4	-2^{31}	$2^{31}-1$

支持编译器选项-fshort-enums 将枚举类型处理位压缩下的 16bit 类型。

1.3.3 默认字节 char 变量的符号说明

默认情况下,不带修饰符的 char 类型的值被 KF32 编译器定义为无符号值。同时,可以使用命令行选项-fsigned-char 将默认类型设置为有符号,-funsigned-char 将默认类型设置为无符号。

1.3.4 浮点类型与长度

KF32 编译器使用 IEEE-754 浮点格式。

表 2-3 浮点类型

类型	位	字节
float	32	4
double	64	8
long double	64	8

1.3.5 指针类型与长度

KF32 编译器中指针长度为 32 位整数。

1.4 属性

通过属性扩展变量或函数的意义,如中断函数。格式为 `__attribute__((attribute-list))` 其支持多个属性,属性之间用“,”间隔。也支持该格式单个属性同格式的多次书写表达多属性。

1.4.1 Interrupt 中断函数

用作中断处理程序的函数生成序言和尾声代码。如：使用 `__attribute__((interrupt))` 修饰函数,则该函数即被编译器视为中断函数处理。详情参见[中断处理程序](#)。针对中断向量表入口函数,其为必须声明的属性。

1.4.2 section (“name”) 指定类型段

将函数或变量放入由 “name” 指定的段。

例如，`void __attribute__((section(“new_sect1”))) foo()`
`{return;}`

函数 `foo` 将被放入 `.new_sect1` 段。

`unsigned int var __attribute__((section(“new_sect2”)))`

变量 `var` 将被放入 `.new_sect2` 段。

当声明该属性后 `-ffunction-sections`、`-fdata-sections` 命令行选项对该属性定义的函数不起作用。

常见的段定义为 `.text`(flash 空间)|`.data`(RAM 空间)|`.bss`(RAM 空间)。同时根据需要将空间进行顺序使用。针对 `flash` 空间，先存放 `text` 段的 `vector` 文件中代码，即中断向量表；紧接着存放默认的函数代码段 `.text`；跟随存放常量属性的 `.rodata`(readonly flash 空间)和 `.rodata`(readonly flash 空间) 代码信息；工具控制将 `ram` 的初始化值跟随结尾存放。针对 `ram` 空间，设计 `ram` 函数的存放的定义 `.indata` 段名，定义初始化的数据 `.data` 段名，定义无初始化的数据 `COMMON` 或 `bss` 段名，RAM 的空间默认设计顺序：`.indata .date .bss .comm`，剩余空间为堆栈空间。

1.4.3 packed 对齐

具有该属性的变量或结构成员将具有所可能的最小对齐值。即，将不为声明分配任何对齐填充存储空间。与 `aligned` 属性联合使用时，`packed` 可以用于设置任意的对齐限制，即大于或小于变量或结构成员的类型所具有的默认对齐值。

需要注意的是，即使声明了该对齐模式，但针对结构体、联合体，其起始仍将按照对齐存放。

如 `struct name1{`
 `char order1;`
 `struct name2{`
 `char order2;`
 `} __attribute__((packed)) names;`
 `} __attribute__((packed)) name ;` //其 `order2` 偏移地址是 4。

注：对齐通常控制不同数据类型的联合。同时应考虑芯片对齐属性的要求进行数据类型的合理使用。如连续 `char` 数据的操作不应通过 `int` 型的指针操作。

注：针对结构体下的位定义，应该使用 `packed` 修饰该结构体，否则其代码联合优化将错误的选用对齐的指令操作，从而编译结果与代码期望不符。

1.4.4 常见属性

变量属性支持

`aligned(alignment)` 定义数据地址对齐配置

`unused` 定义参数未使用的不给予警告

`used` 即使看起来未使用，也对其进行编译

`packed` 一般语言结构体或联合，明确内部连续地址分派。

`section("section-name")`

函数属性支持

`section("section-name")`
`interrupt` 定义该函数为中断服务函数，**中断函数必须该属性修饰**
`noinline` 强制函数不内联，定义的 `inline` 及默认由编译器确定是否内联
`always_inline` 强制函数内联
`weak` 常定义中断函数，即 即使未发现调用关系仍保留该函数,但值地址为 0。
`alias("fun-name")` 函数为别名,可实现未定义时关联到默认函数
`unused` 未使用也不输出警告信息
`optimize("-Ox") x:0 1 2 3 s`, 指定函数自身的编译优化等级
`long_call` 声明该函数被调用时使用基于寄存器的相对调用

1.5 命令行格式

1.5.1 设置环境变量

在使用命令行环境之前，先设置环境变量。

在命令行下输入如下命令添加编译命令至环境变量(以 IDE 默认安装目录为例):

```
PATH=XXXX;%PATH%
```

在命令行下输入如下命令关闭 DOS 路径警告信息 “MS-DOS style path detected” :

```
set CYGWIN=nodosfilewarning
```

ChipON IDE 环境自动集成了环境的自动配置，命令行的工具使用参加命令的详细说明文档。

1.5.2 命令行选项

KF32 编译工具链提供了许多控制编译、链接的选项，**区分大小写**。

表 2-4 C 编译器常用命令行选项

命令	说明
-O0	不使用优化编译
-O2	开启时间优化编译
-Os	开启空间优化编译
-I dir	添加头文件搜索路径
-o file	生成输出文件 file
-gstabs+ -gdwarf-3	编译输出调试信息与格式选项，推荐 dwarf 调试格式
-c	编译、汇编，但不链接
-save-temps=obj	保留中间文件,无参输出在构建目录，参数 obj 保持文件相对路径结构
-Wno-packed-bitfield-compat	不警告位域 packed 属性
-funsigned-char	设置 char 类型为无符号类型
-fsigned-char	设置 char 类型为有符号类型
-funsigned-bitfields	设置位域为无符号类型

-fsigned-bitfields	设置位域为有符号类型
-ffunction-sections	函数独立为 section
-fdata-sections	全局变量独立为 section
-Wa,<options>	传递逗号分隔的命令给汇编器
-Wl,<options>	传递逗号分隔的命令给链接器

表 汇编器常用命令行选项

命令	说明
-I dir	添加库头文件搜索路径
-o file	生成输出文件 file

表 链接器常用命令行选项

命令	说明
-L dir	添加库文件搜索路径
-l libname	链接库文件
-T file	获取链接脚本
--gc-sections	删除无效段(section)
-o file	生成输出文件 file
-Map file	生成输出 map 文件
--start-group archives --end-group	库资源的搜索默认仅向下, 该参数将包括库作为一个组支持循环搜索库变量或函数。

一般情况下, 不需要修改属性, ChipON IDE 集成了常规的语言属性, 比较常见的修改属性包括添加库或引用路径, 是否输出调试信息, 编译优化等级等选项。

2 语言与库说明

2.1 简介

将一些常规方法到库, 方便用户程序开发与功能集成。

KF32 工具精简库位于 C 编译器安装目录的 lib 子目录中, 头文件定义在工具链外部的 include/Sys 路径, 可以通过 KF32 链接器将这些库直接链接到应用程序中。如下方法库分布在链接选项对应的 -lmath -lio -lstring -lstdlib -lctype -lcrtv1(2|3...) 库文件中, 具体型号的内核库 libcrtvx.a 选择可参见 ide 环境下的配置变量 -l\${CHIP_KERNEL_LIB} 的结果, 可通过构建目录下 makefile 查询获取。其中 libcrtvx.a 差异提供芯片的特殊 api, 如芯片信息获取以及最小串口打印 putchar 和 fputchar, 但不使用或源码中重实现这些方法是, 任一 libcrtvx.a 的一个系列库均可实现正常的编译构建工作。

KF32 工具提供基于 newlibs 输出的标准适配 c 库和数学库, 同时输出裁剪的 c++ 库以及基于编译器的完整适配运行时库, 这些头文件和库定义在工具链 include/Std_Sys 和精简一致的 /lib 路径下。如下方法库分布在链接选项对应的 -lgcc -lc -lm -lstdc++ -lsupc++ 库文件中。

2 套标准头文件与库文件系统针对格式化打印的支持具有差异，精简库直接对应 Usart 外设，通用库对应基于文件系统的框架，因此参考头文件定义并需要 malloc 和 free 方法的支持。

2.2 实现定义的限制<LIMITS.H>

头文件 limits.h 中包含了一些宏，这些宏定义了整型数的最小值和最大值。

宏	值	描述
MB_LEN_MAX	1	多字节字符的最大字节数
CHAR_BIT	8	char 类型位数
SCHAR_MIN	-128	signed char 型的最小值
SCHAR_MAX	127	signed char 型的最大值
UCHAR_MAX	255	unsigned char 型的最大值
CHAR_MIN	默认为 0，如果指定 -fsigned-char 选项则为 -128。	char 型的最小值
CHAR_MAX	默认为 255，如果指定 -fsigned-char 选项则为 127。	char 型的最大值
SHRT_MIN	-32768	short int 型的最小值
SHRT_MAX	32767	short int 型的最大值
USHRT_MAX	65535	unsigned short int 型的最大值
INT_MAX	2147483647	int 型的最小值
INT_MIN	-2147483648	int 型的最大值
UINT_MAX	4294967295	unsigned int 型的最大值
LONG_MAX	2147483647	long 型的最小值
LONG_MIN	-2147483648	long 型的最大值
ULONG_MAX	4294967295	unsigned long 型的最大值
LLONG_MAX	9223372036854775807	long long 型的最小值
LLONG_MIN	-9223372036854775808	long long 型的最大值
ULLONG_MAX	18446744073709551615	unsigned long long 型的最大值

2.3 C 常规精简库方法

KF32 编译工具链实现部分 C 标准库，包括字符操作、字符串操作、数学库及相应的常量、类型等定义。用户可通过编译工具头文件路径中的相应头文件进行查看。

2.3.1 stdio 串口输出库函数

```
#include "stdio.h"
```

```
char getchar(void)
```

串口 0 基于收标志的返回接收数据

```
void puts(const char *str)
```

串口 0 字符串输出函数，附加换行符 ‘\n’ 输出。

```
void putchar(int c)
```

串口 0 字符输出函数

```
char fgetchar (STREAM *stream)
```

指定串口基于收标志的返回接收数据

```
void fputs(const char *str, STREAM *stream)
```

指定串口字符串输出函数。

```
void fputchar(int c, STREAM *stream)
```

指定串口字符输出函数

```
int printf (const char *fmt, ...)
```

串口 0 的可变参数的格式化打印输出方法。

```
int fprintf (STREAM *stream, const char *fmt, ...)
```

指定串口的可变参数的格式化打印输出方法。

```
int sprintf (char *ebuf, const char *fmt, ...)
```

指定字符缓存区的可变参数的格式化链接生成方法。

```
int snprintf (char *str, unsigned int n, const char *format, ...)
```

指定字符缓存区的可变参数的格式化链接生成方法。

该方法额外定义局部 512 字节空间用于结果缓存输出。

注：历史方法基于 msp 与脚本全局 ram 分配尾部变量做堆栈溢出判断，该方法不适合服务于 OS 操作系统下的 psp 进程。

串口指定参数为：USART0_STREAM ~ USART7_STREAM，使用示例如下：

```
printf(          "文件%s:%d\r\n", __FUNCTION__, __LINE__ );
fprintf(USART0_STREAM,          "文件%s:%d\r\n", __FUNCTION__, __LINE__ );
sprintf(USART_Array_Tansmit, "F:%s:%d\r\n", __FUNCTION__, __LINE__ );
printf(          "abcdefghijklmnpqrst1234567890\r\nC1CC\r\n");
fprintf(USART0_STREAM,
        "abcdefghijklmnpqrst1234567890\r\nC1CC\r\n");
puts("ABBB\t\r\n");
fputs("ABBB\t\r\n",  USART0_STREAM);
putchar('A');
fputchar('\n',      USART0_STREAM);
```

```
Receive=getchar();
```

```
Receive=fgetchar(USART0_STREAM);
```

2.3.2 math 数学运算库函数

```
#include "math.h"
```

如：

```
float      sinf      ( float );
double     sin       ( double );
float      expf      ( float );
```

```
float    sqrtf      ( float )
float    powf       ( float, float );
float    fabsf      ( float );
float    j0f        ( float );
```

2.3.3 string 字符串处理库函数

```
#include "string.h"
```

```
extern int    ffs (int);
```

返回第一个 bit 位上位 1 的 bit 位

```
extern int    ffs1 (long);
```

返回第一个 bit 位上位 1 的 bit 位

```
extern int    ffs11 (long long);
```

返回第一个 bit 位上位 1 的 bit 位

```
extern void * memchr(const void *, int, size_t);
```

扫描指定字节长度是否包含指定的字符。相同返回匹配字符为开始的字符串。

```
extern int    memcmp(const void *, const void *, size_t);
```

大小写敏感比较指定字节数的字符串是否相同。相同返回 0。

```
extern void * memcpy(void *, const void *, size_t);
```

拷贝指定字符长度的字符串。

```
extern void * memmove(void *, const void *, size_t);
```

块区域源字符串按指定字符数量进行转移处理到目标区域。

```
extern void * memset(void *, int, size_t);
```

指定缓存区长度的赋指定数值

```
extern int    strcasecmp (const char *, const char *);
```

忽略大小写的字符串比较，一致返回 0。否则为字符小写差值。

```
extern char * strcat(char *, const char *);
```

字符串合并。

```
extern char * strchr(const char *, int);
```

返回指定字符起始的字符串，也可能为 NULL

```
extern int    strcmp(const char *, const char *);
```

大小写敏感的字符串比较，相同返回 0，否则返回不同值的差值。

```
extern int    strcoll(const char *, const char *);
```

同 strcmp 的大小写字符串比较。

```
extern char * strcpy(char *, const char *);
```

字符串拷贝赋值。

```
extern size_t strcspn(const char *, const char *);
```

字符串任意字符内容检索，返回不匹配前缀偏移量。

```
extern size_t strlen(const char *);
```

字符串长度计算。

```
extern int    strncasecmp (const char *, const char *, size_t);
```

指定长度的忽略大小写字符串比较函数，相等返回 0

```
extern char * strncat(char *, const char *, size_t);
```

指定长度的字符串合并函数。

```
extern int      strcmp(const char *, const char *, size_t);
```

指定长度的大小写敏感字符串比较函数。

```
extern char *   strncpy(char *, const char *, size_t);
```

指定长度的字符串赋值拷贝函数。

```
extern char *   strpbrk(const char *, const char *);
```

检测是否任意字符包含，返回起始位置，否则返回 NULL。

```
extern char *   strrchr(const char *, int);
```

字符串尾部开始搜集检索包含的字符字符串，否则返回 NULL。

```
extern size_t   strspn(const char *, const char *);
```

字符串任意字符匹配识别，返回偏移位置。

```
extern char *   strstr(const char *, const char *);
```

检索是否包含字符串序列，成功返回开始包含的字符串，否则返回 NULL。

```
extern char *   strtok(char *, const char *);
```

分解字符串位一组字符串。

```
extern size_t   strxfrm(char *, const char *, size_t);
```

根据长度拷贝字符串到目标字符串中，并返回原字符串的长度。

2.3.4 类型判断库函数

```
#include "ctype.h"
```

```
int isalpha(int c);
```

是否为 A-Z , a-z 的字符，是返回自身，否则返回 0

```
int isupper(int c);
```

是否为 A-Z，是返回自身，否则返回 0

```
int islower(int c);
```

是否为 a-z，是返回自身，否则返回 0

```
int isdigit(int c);
```

是否为 '0' - '9'，是返回自身，否则返回 0

```
int isxdigit(int c);
```

是否为 16 进制有效字符，是返回自身，否则返回 0. 【0-9 A-F a-f】

```
int isspace(int c);
```

是否为空字符，是返回自身，否则返回 0. 【\t \n \v \f \r \0x7F】

```
int ispunct(int c);
```

是否为不包括空格的有效字符，如【“，{。}%&等】

```
int isalnum(int c);
```

是否常规表达字符，是返回自身，否则返回 0. 【0-9 A-Z a-z】

```
int isprint(int c);
```

是否为可打印字符，是返回自身，否则返回 0. 【\0x20~0x7E】

```
int isgraph(int c);
```

是否图标符号，是返回自身，否则返回 0，类似 isprint，但不包括空格。

```
int iscntrl(int c);
```

是否为控制符号，是返回自身，否则返回 0. 【\0x00~0x1F】 【\0x7F】

```
int isascii(int c);
```

是否为 ascii 编码数值，是返回自身，否则返回 0. 【\0x00~0x7F】

```
int tolower(int c);
```

返回字符对于的小写，或自身。即 A-Z 返回 a-z，其他字符保持自身返回。

```
int toupper(int c);
```

返回字符对于的大写，或自身。即 a-z 返回 A-Z，其他字符保持自身返回。

2.3.5 系统标准库函数

```
#include "stdint.h"
```

定义常规数据类型别名，如：

```
typedef signed char      int8_t;
typedef unsigned char    uint8_t;
typedef short            int16_t;
typedef unsigned short   uint16_t;
typedef int               int32_t;
typedef unsigned int      uint32_t;
typedef long long         int64_t;
typedef unsigned long long uint64_t;
typedef int intptr_t;
typedef unsigned int uintptr_t;
typedef unsigned int size_t;
#define NULL ((void *)0)
typedef unsigned short wchar_t;
```

```
#include "stdlib.h"
```

```
typedef struct {
    int quot;
    int rem;
} div_t;
typedef struct {
    long quot;
    long rem;
} ldiv_t;
typedef struct {
    long long quot;
    long long rem;
} lldiv_t;
int abs (int i); long labs (long )
```

返回绝对值

```
double atof(char * s);
```

double 型字符串传送值还原。支持符号，小数点，科学计数法 E。

```
int atoi( const char * str );
```

int 型字符串传送值还原。

```
long atol( const char * str );
```

long 型字符串传送值还原。

```
long long atoll( const char * str );
```

longlong 型字符串传送值还原。

```
void _Exit(int status); void exit(int status);
```

异常死循环函数。

```
div_t div( int numer, int denom );
ldiv_t ldiv(long numer, long denom)
lldiv_t lldiv(long long numer, long long denom);
```

计算商和余数。

```
void ftoa (float value, char *str, unsigned char prec)
```

浮点转换为字符串。

```
void uitoa (unsigned int value, char *str, unsigned char radix);
void itoa (int value, char *str, unsigned char radix);
void ultoa(unsigned long value, char* str, unsigned char radix);
void ltoa(long value, char* str, unsigned char radix);
```

整数转对应进制的字符串。Radix=2, 8, 10, 16

```
void qsort( void *base, size_t np, size_t sizep, int (*cmp)(const void *, const void *) );
```

快速排序函数。实现对数组的排序。

```
double strtod(const char *st, char **endptr);
float strtof(const char *st, char **endptr);
```

将字符串转换为浮点数,遇到不合法内容可通过 endptr 传出。

```
long strtol(const char *st, char **endptr, int base);
long long strtoll(const char *st, char **endptr, int base);
unsigned long strtoul(const char *st, char **endptr, int base);
unsigned long long strtoull(const char *st, char **endptr, int base);
```

base 范围为 0 或 2 至 36。将字符串转换为整数数据。遇到不合法内容可通过 endptr 传出。即 base 为指定字符串的进制模式。0 默认是 10 进制，但遇到前缀会切换，如 0x 对于 16 进制。0 对于 8 进制。

```
int rand( void ) void srand( unsigned int seed )
```

设置随机数种子和获取一个随机数。

```
void * bsearch( const void *key, const void *base, size_t n, size_t size,
                int (*cmp)( const void *keyval, const void *datum ) )
```

对对象数组执行二分查找，数组的内容根据传入的比较函数进行升序排序。

```
#include "time.h"
```

```
typedef long    time_t;      /* for representing times in seconds */
typedef long    clock_t;

struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
```

```
};
```

```
extern unsigned char RtcReadTime(struct tm *timeptr);
```

```
extern unsigned char RtcWriteTime(struct tm *timeptr);
```

应该在外设库，或自行编写和芯片交互的实时时间获取与设定函数。如 RTC 应用。

```
#define clock()      time(NULL);
```

```
extern time_t      time(time_t *);
```

获取自 00:00:00 Jan 1 1970 后的时间整数值。

```
extern void timeTm(struct tm *);
```

获取当前时间到时间结构体对象中。

```
extern char stime(struct tm *);
```

```
extern char stime2(time_t *);
```

基于数值参数或时间结构体对象值设置当前系统中时间。

```
#define difftime(t1, t0) ((long double)((time_t)(t1)-(time_t)(t0)))
```

返回 2 次时间的时间整数值差，即单位秒。

```
extern time_t      mktime(struct tm *);
```

基于时间结构体转换为自 00:00:00 Jan 1 1970 后的时间整数值。

```
extern char *      asctime (struct tm *);
```

```
extern char *      ctime   (time_t *      );
```

传递时间并转换为“星期 月 日 时:分:秒 年”格式的字符串时间内容。其中星期显示为：

“Sun”, “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”。

月显示为

“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, “Dec”

```
extern struct tm * localtime(time_t *); /* local time */
```

```
extern struct tm * gmtime( time_t *); /* Universal time */
```

将时间整数转换到时间结构体对象中。

```
extern void gmtimeTM( time_t *timep, struct tm *inTmTime );
```

基于参数的将时间整数转换到时间结构体对象中。

```
#include “malloc.h”
```

```
extern void *calloc(size_t /*number*/, size_t /*size*/);
```

申请多个基本对象数量大小的字节空间。

```
extern void *malloc(size_t /*size*/);
```

申请指定字节数量的字节空间。

```
extern void *realloc(void *ptr_par, size_t n);
```

当前对象重新调整目标字节需求申请。

```
extern void free(void * /*p*/);
```

申请的字节空间释放。

注：该资源空间有脚本或向量表中数组实现大小，默认脚本配置仅 256 字节。ram 的划分：全局初始化(包括 ram 函数) 全局未初始化 堆空间(heap) 栈空间(stack)，堆地址起始更随变量之后，剩余空间为满栈减模式下动态使用。

2.4 浮点库函数及 64 位整数库函数

浮点库包括单精度、双精度浮点操作及 64 位整型变量运算的一系列函数组成。这些函数由 C 语言编译器编译生成调用过程，以辅助实现相应功能，用户无需控制。其中浮点格式采用 IEEE754 标准。嵌汇编或汇编仍可调用使用库，进而实现功能。

当 C 语言定义 float 类型并进行计算时，编译器会自动调用浮点库完成运算。当使用汇编编写代码时，根据参数传递规则调用对于的函数即可调用库实现浮点运算。

2.4.1 浮点类型加减法

1、__addsf3

描述：单精度浮点数加法。

函数原型：float __addsf3 (float a, float b);

返回值：a + b。

2、__adddf3

描述：双精度浮点数加法。

函数原型：double __adddf3 (double a, double b);

返回值：a + b。

3、__subsf3

描述：单精度浮点数减法。

函数原型：float __subsf3 (float a, float b);

返回值：a - b。

4、__subdf3

描述：双精度浮点数减法。

函数原型：double __subdf3 (double a, double b);

返回值：a - b。

2.4.2 浮点类型乘除法

1、__mulsf3

描述：单精度浮点数乘法。

函数原型：float __mulsf3 (float a, float b);

返回值：a * b。

2、__muldf3

描述：双精度浮点数乘法。

函数原型：double __muldf3 (double a, double b);

返回值：a * b。

3、__divsf3

描述：单精度浮点数除法。

函数原型：float __divsf3 (float a, float b);

返回值：a / b。

4、__divdf3

描述：双精度浮点数除法。

函数原型：double __divdf3 (double a, double b);

返回值： a / b 。

2.4.3 浮点类型比较运算

1、__eqsf2

描述： 单精度浮点数等于比较。

函数原型： `signed int __eqsf2 (float a, float b);`

返回值： a 与 b 相等返回 0，不等返回 1。

2、__eqdf2

描述： 双精度浮点数等于比较。

函数原型： `signed int __eqdf2 (double a, double b);`

返回值： a 与 b 相等返回 0，不等返回 1。

3、__nesf2

描述： 单精度浮点数不等比较。

函数原型： `signed int __nesf2 (float a, float b);`

返回值： a 与 b 相等返回 0，不等返回 1。

4、__nedf2

描述： 双精度浮点数不等比较。

函数原型： `signed int __nedf2 (double a, double b);`

返回值： a 与 b 相等返回 0，不等返回 1。

5、__lesf2

描述： 单精度浮点数小于等于比较。

函数原型： `signed int __lesf2 (float a, float b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

6、__ltsf2

描述： 单精度浮点数小于等于比较。

函数原型： `signed int __ltsf2 (float a, float b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

7、__ledf2

描述： 双精度浮点数小于等于比较。

函数原型： `signed int __ledf2 (double a, double b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

8、__ltdf2

描述： 双精度浮点数小于比较。

函数原型： `signed int __ltdf2 (double a, double b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

9、__gesf2

描述： 单精度浮点数大于等于比较。

函数原型： `signed int __gesf2 (float a, float b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

10、__gtsf2

描述： 单精度浮点数大于比较。

函数原型： `signed int __gtsf2 (float a, float b);`

返回值： a 等于 b 返回 0， a 大于 b 返回 1， a 小于 b 返回 -1。

11、__gedf2

描述：双精度浮点数大于等于比较。

函数原型：signed int __gedf2 (double a, double b);

返回值：a 等于 b 返回 0，a 大于 b 返回 1，a 小于 b 返回 -1。

12、__gtddf2

描述：双精度浮点数大于比较。

函数原型：signed int __gtddf2 (double a, double b);

返回值：a 等于 b 返回 0，a 大于 b 返回 1，a 小于 b 返回 -1。

2.4.4 浮点类型转换

1、__truncdfsf2

描述：转换 IEEE 双精度浮点数为 IEEE 单精度浮点数。

函数原型：float __truncdfsf2 (double a);

返回值：转换后的单精度浮点数。

2、__extendsfdf2

描述：转换 IEEE 单精度浮点数为 IEEE 双精度浮点数。

函数原型：double __extendsfdf2 (float a);

返回值：转换后的双精度浮点数。

3、__floatunsisf

描述：转换 32 位无符号整型为 IEEE 单精度浮点数。

函数原型：float __floatunsisf (unsigned int i);

返回值：转换后的单精度浮点数。

4、__floatunsidf

描述：转换 32 位无符号整型为 IEEE 双精度浮点数。

函数原型：double __floatunsidf (unsigned int i);

返回值：转换后的双精度浮点数。

5、__floatundisf

描述：转换 64 位无符号整型为 IEEE 单精度浮点数。

函数原型：float __floatundisf (unsigned long long i);

返回值：转换后的单精度浮点数。

6、__floatundidf

描述：转换 64 位无符号整型为 IEEE 双精度浮点数。

函数原型：double __floatundidf (unsigned long long i);

返回值：转换后的双精度浮点数。

7、__floatsisf

描述：转换 32 位有符号整型为 IEEE 单精度浮点数。

函数原型：float __floatsisf (int i);

返回值：转换后的单精度浮点数。

8、__floatsidf

描述：转换 32 位有符号整型为 IEEE 双精度浮点数。

函数原型：double __floatsidf (int i);

返回值：转换后的双精度浮点数。

9、__floatdisf

描述：转换 64 位有符号整型为 IEEE 单精度浮点数。

函数原型：float __floatdisf (long long i);

返回值：转换后的单精度浮点数。

10、__floatdidf

描述：转换 64 位有符号整型为 IEEE 双精度浮点数。

函数原型：double __floatdidf (long long i);

返回值：转换后的双精度浮点数。

11、__fixunssfsi

描述：转换 IEEE 单精度浮点数为 32 位无符号整型。

函数原型：unsigned int __fixunssfsi (float a);

返回值：转换后的 32 位无符号整型数。

12、__fixunssfdi

描述：转换 IEEE 单精度浮点数为 64 位无符号整型。

函数原型：unsigned long long __fixunssfdi (float a);

返回值：转换后的 64 位无符号整型数。

13、__fixunsdfsi

描述：转换 IEEE 双精度浮点数为 32 位无符号整型。

函数原型：unsigned int __fixunsdfsi (double a);

返回值：转换后的 32 位无符号整型数。

14、__fixunsdfdi

描述：转换 IEEE 双精度浮点数为 64 位无符号整型。

函数原型：unsigned long long __fixunsdfdi (double a);

返回值：转换后的 64 位无符号整型数。

15、__fixsfsi

描述：转换 IEEE 单精度浮点数为 32 位有符号整型。

函数原型：int __fixsfsi (float a);

返回值：转换后的 32 位有符号整型数。

16、__fixsfdi

描述：转换 IEEE 单精度浮点数为 64 位有符号整型。

函数原型：long long __fixsfdi (float a);

返回值：转换后的 64 位有符号整型数。

17、__fixdfsi

描述：转换 IEEE 双精度浮点数为 32 位有符号整型。

函数原型：int __fixdfsi (double a);

返回值：转换后的 32 位有符号整型数。

18、__fixdfdi

描述：转换 IEEE 双精度浮点数为 64 位有符号整型。

函数原型：long long __fixdfdi (double a);

返回值：转换后的 64 位有符号整型数。

2.4.5 整型 64 位库函数

1、__muldi3

描述：64 位整型乘法运算。

函数原型：long long __muldi3 (long long a, long long b);

返回值：a * b 的低 64 位。

2、__divdi3

描述：64 位有符号整型除法运算。

函数原型：long long __divdi3 (long long a, long long b);

返回值：a / b。

3、__moddi3

描述：64 位有符号整型取模运算。

函数原型：long long __moddi3 (long long a, long long b);

返回值：a % b。

4、__udivdi3

描述：64 位无符号整型除法运算。

函数原型：unsigned long long __udivdi3 (unsigned long long a, unsigned long long b);

返回值：a / b。

5、__umoddi3

描述：64 位无符号整型取模运算。

函数原型：unsigned long long __umoddi3 (unsigned long long a, unsigned long long b);

返回值：a % b。

6、__cmpdi2

描述：64 位有符号整型比较。

函数原型：signed int __cmpdi2 (long long a, long long b);

返回值：a 小于 b 返回 0，a 等于 b 返回 1，a 大于 b 返回 2。

7、__ucmpdi2

描述：64 位无符号整型比较。

函数原型：signed int __ucmpdi2 (unsigned long long a, unsigned long long b);

返回值：a 小于 b 返回 0，a 等于 b 返回 1，a 大于 b 返回 2。

2.5 可变参数列表<STDARG.H>

头文件 `stdarg.h` 支持带有可变参数列表的函数。这允许函数带有没有相应声明的参数。参数列表中必须至少包含一个指定的参数。**可变参数用省略号 (...) 表示，当宏定义传递时，可以在宏定义带入代码中使用 __VA_ARGS__ 传递。**必须在函数中声明一个 `va_list` 类型的对象用以保存这些参数。`va_start` 将变量初始化为一个参数列表，`va_arg` 用来访问这个参数列表，`va_end` 用来终止参数的使用。示例如下：

```
int foo (int a, ...)
{
    va_list va;
    int i, res;

    va_start (va, a);
    for (i = 0; i < 4; ++i)
```

```
(void) va_arg (va, int);
res = va_arg (va, int);
va_end (va);
return res;
}
int main ()
{
if (foo (5, 4, 3, 2, 1, 0))
    return -1;
    return 0;
}
```

其中，foo 为可变参函数，可变参数在 `int a` 之后，其功能为返回第 5 个可变参数，如上例返回 0。

2.5.1 va_arg

描述： 获取当前参数。

头文件： `<stdarg.h>`

函数原型： `T va_arg(va_list ap, T)`

参数： `ap` 指向参数列表的指针

`T` 要获取的参数的类型

返回值： 返回 `T` 类型的当前参数

说明： `va_start` 必须在 `va_arg` 之前调用。

2.5.2 va_end

描述： 结束使用 `ap`。

头文件： `<stdarg.h>`

函数原型： `void va_end(va_list ap)`

参数： `ap` 指向参数列表的指针

说明： 在调用 `va_end` 之后，参数列表指针 `ap` 被视为是无效的。在遇到下一个 `va_start` 之前，不应该再调用 `va_arg`。

2.5.3 va_list

描述： 类型 `va_list` 声明一个变量，以引用可变长度参数列表中的每个参数。

头文件： `<stdarg.h>`

2.5.4 va_start

描述： 设置参数指针 `ap` 指向可变长度参数列表中的第一个可选参数。

头文件： `<stdarg.h>`

函数原型： `void va_start(va_list ap, last_arg)`

参数： `ap` 指向参数列表的指针

`last_arg` 可选参数（省略号代表的参数）之前的最后一个指定的参数。

3 中断函数

3.1 简介

中断处理用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用专门的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

KF32 器件支持多个内部和外部中断源。另外，允许高优先级中断中断任何正在处理的低优先级中断。

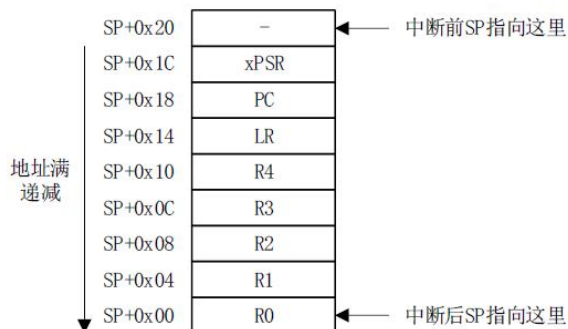
KF32 编译器完全支持在 C 或汇编代码中进行中断处理。

3.2 中断处理函数

3.2.1 中断函数现场保护

中断处理函数用于实现现场保护和恢复，以确保从中断返回时，程序现场恢复进入中断前的状态。

响应中断时，KF32 硬件约定将 R0、R1、R2、R3、R4、R13（LR）、返回地址（当前 PC 值）和程序状态寄存器（xPSR）压入当前激活的堆栈空间中，因此不需要主动保存它们。而其他被中断代码使用到的寄存器会被编译器进行入栈保护。



3.2.2 中断处理程序

函数通过 `interrupt` 属性标记为中断处理程序函数。

默认下，KF32 编译器已将中断向量表转为中断向量配置文件(`vector.c` 或 `vector.asm`)。可根据喜好修改文件中对应位置的符号为函数名，或修改对应符号为自定义中断处理函数名。示例如下。

如 T1 中断的入口地址被配置为“`.long _T1_exception`”，则其中断处理程序写为：

```
void __attribute__((interrupt)) _T1_exception (void)
{
}
```

或用户修改中断向量配置文件将 `_T1_exception` 修改为自定义函数名。

KF32 中断处理函数必须遵守下列规定：

- 1、返回的数据类型必须是 `void`;
- 2、不能有参数;
- 3、必须使用 `__attribute__((interrupt))` 声明;
- 4、函数地址必须与中断向量一致，即函数名称必须与 `vector.c` 文件中指定位置的向量名称一致;
- 5、既可以被中断处理程序访问也可以被其他函数访问的全局变量应定义为 `volatile`;
- 6、中断入口会自动保存寄存器(R0、R1、R2、R3、R4、R13)，并在中断返回时恢复;
- 7、使用的中断必须建立该中断函数，否则芯片将运行错误。

3.2.3 中断向量配置

KF32 系列支持多个内核中断和多个外设中断，256 个向量的中断向量表参见 KF32 芯片数据手册。默认情况下使用的中断向量表的起始单元位于存储器的 `0x0000_0000` 地址处。当响应中断时，处理器自动从向量表中的相应位置加载中断向量入口地址。所以中断处理函数的地址必须为中断向量中对应的地址。

中断向量配置文件中的代码将被链接器分配到芯片起始地址 `0x0` 处，其中每一中断类型分别对应各自地址。若用户程序未使用或未全部使用中断处理，则可修改中断向量配置文件，以节省内存空间。

中断向量配置文件的修改需遵循以下两点：

- 1、`0x0` 地址和 `0x4` 地址的向量不可删除，即初时 SP 和复位向量不可删除;
- 2、删除向量的操作需从后向前开始，不可跳过，否则无法正确进入中断服务程序；【如果应用中重映射中断向量入口，推荐不能删除，必须满 512 地址空间占用，否则空闲地址存在的代码因向量表切换会获取错误的内容并执行错乱。】

中断向量配置文件修改示例说明：

用户程序使用定时器 1 全局中断(T1，向量地址 `0x0000_0064`)和定时器 5 全局中断(T5，向量地址 `0x0000_0074`)，而其他中断向量均未使用。则用户可将中断向量配置文件中 “`.long _T5_exception`” 和 “`.weak _T5_exception`” 之后的几行信息删除，而之前的中断向量不可修改，包括未使用的中断向量。

3.2.4 中断使用的变量

中断和用户代码同时访问与修改的变量推荐使用 `volatile` 声明，如：

`volatile int var`; 否则因为函数的调用关系未存在关联，独自使用范围的优化可能操作参数无效的现象。

4 特殊功能寄存器的操作

由于位域处理的效率较低，建议用户对 SFR 进行 32 位操作，减少位域操作。用户如下需要可自行定义，本章节仅供参考，推荐使用 ChipON 的外设库，包括 HLI、MCAL 等。

4.1 简介

本章主要介绍 C 语言和汇编语言中如何访问特殊功能寄存器。

4.2 特殊功能寄存器的操作

特定于处理器的头文件是一些包含了在 C 或汇编语言中使用的特殊功能寄存器(Special Function Register, SFR)的外部声明的文件。依照约定, 每个 SFR 都使用数据手册中的相同名称进行命名, 如 OSC_CTL1 代表振荡器控制寄存器 1。定义结构以实现寄存器的位声明, 如 _HFCLKCAL。同时约定前面添加前缀 “_”。

举例, 地址为 0x40000004 的特殊功能寄存器 OSC_CTL1, 在 C 头文件的声明如下:

```
union OSC_CTL1_REG{
    unsigned int reg;
    struct OSC_CTL1_BITS{
        unsigned char _HFCLKTRIM:5;
        unsigned char _rsvd1:3;
        unsigned char _HFCLKCAL:8;
        unsigned char _FSCM:1;
        unsigned short _rsvd2:11;
        unsigned char _SCLKOE:1;
        unsigned char _SCLKOUTDIV:3;
    }__attribute__((packed)) bits;
    struct all_bit bit;
};

#define OSC_CTL1 (*(volatile union OSC_CTL1_REG *)0x40000004)
```

KF32 芯片特殊功能寄存器为芯片固定地址, 且芯片支持 32 位地址空间访问, 故 SFR 仅需在头文件中将其地址声明。如上所述, KF32 芯片的每一个 SFR 都被定义为一个结构(联合体), 其中 reg 为 32 位数据访问, bits 可访问一组位, bit 可访问每一个位。

特殊功能寄存器 OSC_CTL1 在汇编头文件的声明如下:

```
.EQU OSC_CTL1, 0x40000004
.EQU _SCLKOUTDIV2, 31
.EQU _SCLKOUTDIV1, 30
.EQU _SCLKOUTDIV0, 29
.EQU _SCLKOE, 28
.EQU _FSCM, 16
.EQU _HFCLKCAL7, 15
.EQU _HFCLKCAL6, 14
.EQU _HFCLKCAL5, 13
.EQU _HFCLKCAL4, 12
.EQU _HFCLKCAL3, 11
.EQU _HFCLKCAL2, 10
.EQU _HFCLKCAL1, 9
.EQU _HFCLKCAL0, 8
.EQU _HFCLKTRIM4, 4
.EQU _HFCLKTRIM3, 3
.EQU _HFCLKTRIM2, 2
```

```
.EQU _HFCLKTRIM1, 1
.EQU _HFCLKTRIM0, 0
```

在应用程序中使用 SFR 时，需要执行 2 个步骤。

1. 需包含芯片头文件：

1.1 C 语言需包含相应芯片的头文件，格式如下：

```
#include <KF32XXXX.h>
```

1.2 C 语言中嵌汇编使用 SFR 时，需要在 C 文件中包含汇编头文件，如下：

```
asm(".include \"KF32XXXX.inc\"");
```

1.3 汇编项目需包含汇编头文件：

```
.include "KF32XXXX.inc"
```

注：使用该头文件的应用用来开发寄存器版直接操作的程序，当使用外设库方案进行程序开发时，不应添加该头文件，即外设库具有独立的命令和结构定义，具体查看外设库使用说明。

2. C 语言直接联合体变量操作 SFR；汇编项目通过地址和位号直接操作。

2.1 如，设置 SCLK 输出时钟 1/32 分频：

C 语言：OSC_CTL1.bits._SCLKOUTDIV = 5;

```
汇编：LD R5, #OSC_CTL1
      SET [R5], #_SCLKOUTDIV0
      CLR [R5], #_SCLKOUTDIV1
      SET [R5], #_SCLKOUTDIV2
```

2.2 如，设置 OSC_CTL1 第 30 位为 1：

C 语言：OSC_CTL1.bit._b30 = 1;

```
汇编：LD R5, #OSC_CTL1
      SET [R5], #30
```

2.3 如，将 OSC_CTL1 所有位置 1，即寄存器整体赋值：

C 语言：OSC_CTL1.reg = 0xffffffff;

```
汇编：LD R5, #OSC_CTL1
      MOV R0, #0
      NOT R0, R0
      ST.w [R5], R0
```

5 编译器运行时约束

5.1 寄存器约定

表 6-1 寄存器约定

寄存器名称	用途	跨函数调用时是否保存
R0	gp/传参/返回值	不保存
R1	gp/传参/返回值	不保存
R2 - R4	gp/传参	不保存
R5	gp/scratch	不保存
R6	gp/fp	保存
R7 - R12	gp	保存
R13	lr/gp/函数返回地址	保存

R14	sp/fixed	否
R15	pc/fixed	否

在表 6-1 中，**gp** 表示该寄存为通用寄存器，可参与任何操作，**fixed** 表示该寄存器为特殊寄存器，具有固定的用途，寄存器分配时，不参与分配。**R6-R13** 用于保存生命周期跨函数调用的变量。**R6** 在特殊情况下可替代实现 **FP** 帧指针的功能，如临时栈空间开辟。**R13** 别名为 **LR** 链接寄存器，**R14** 别名为 **SP** 栈寄存器，**R15** 别名为 **PC**。

5.2 堆栈使用

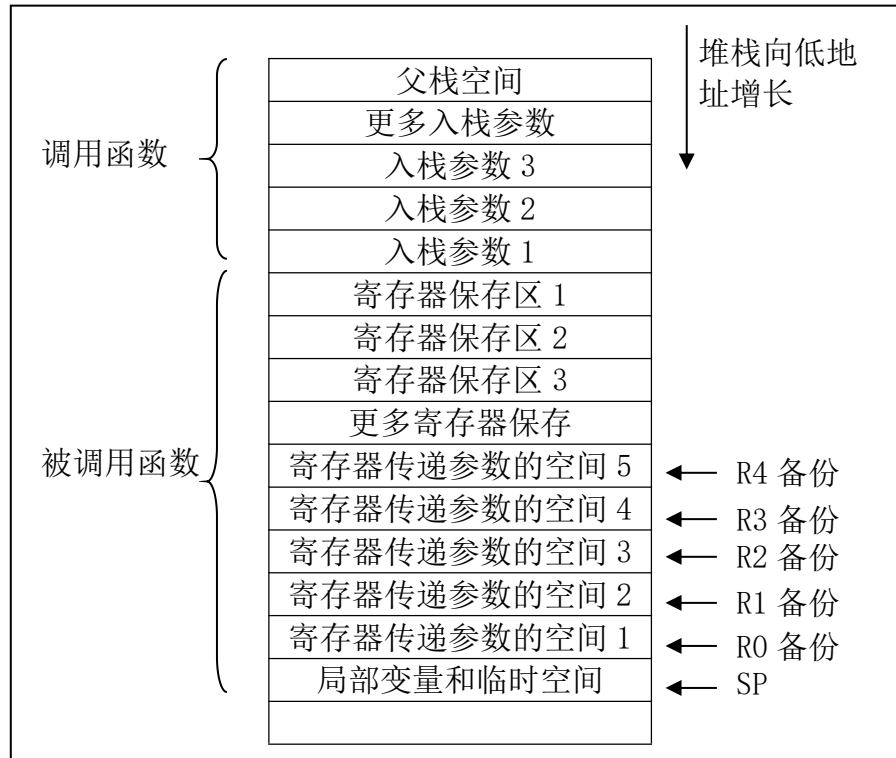


图 6-1 堆栈帧

KF32 编译器支持在示例向量表文件中与定义或脚本调整的预留堆空间[该实施在分配 **ram** 空间时确认具有该大小的空间用于局部变量使用，自身不限制函数的堆栈能力，即函数基于代码动态压栈或 **sp** 运算的申请满足其运行的空间]。堆访问使用通用寄存器 **SP** 作为栈指针，栈指针设计为满减的方式，即从高地址到低地址向下增长，**SP** 指向栈顶最后入栈的数据地址。

KF32 芯片从 **0x0** 地址获取栈指针的初始化数值。编译工具默认将 **SP** 设置在 **RAM** 末地址+1 位置，该入口地址通用通过向量表和链接脚本的变量 **__initial_sp** 配置实现。

在 **kf32** 体系结构中，没有提供固定的寄存器作为帧指针。但是对于某些特定的应用场景，在栈帧结构中，必须要维持一个帧指针，如函数中存在动态数组或者调用了 **__builtin_alloca** 函数等动态修改 **sp** 指针的情景。对于这种情况，其用 **r6** 作为 **fp**，用于访问栈空间的临时变量，以及函数退出时，用于更新 **sp**。**R6** 作为普通寄存器使用，不需要额外设置。

注：局部变量空间由 SP 在栈上临时申请与释放，因此局部变量应先写入值后使用，否则其初始值为其他无关值，同时未被写入过的区域将触发芯片 ECC

规则检查异常事件。

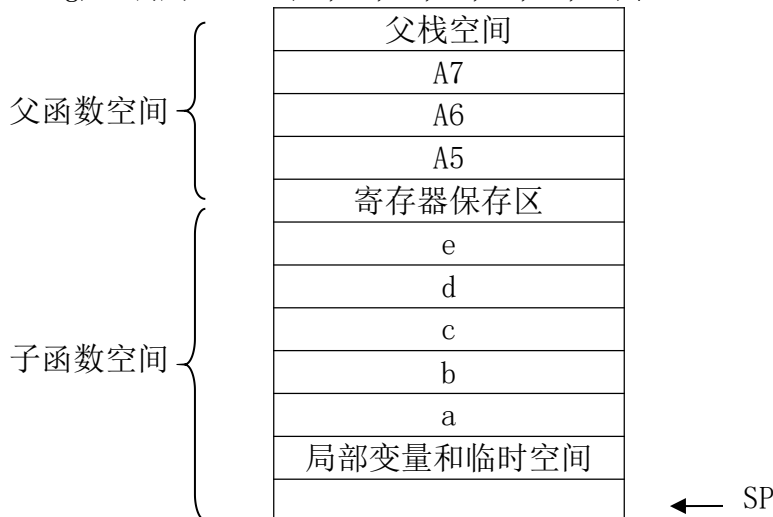
5.3 函数调用

KF32 编译器始终将函数地址分配在 2 字节边界对齐。

调用函数时：

- ✧ 寄存器 R0-R4 用于向函数传递参数，函数参数从左向右一次存入 R0 至 R4，多出部分的参数将按序存入栈中，如图 6-2。KF32 编译器支持某一参数被分隔在寄存器和栈中存储。如函数 `foo (int a,int b,int c,int d,long long e)`，其中参数 e 低位将存于 R4，高位存入栈。
- ✧ 函数的栈空间是四字节边界对齐。如函数仅一个 `char` 类型局部变量，KF32 编译器将在栈中申请四字节空间，并在其低地址存储局部变量(若局部变量未被优化)。
- ✧ 寄存器 LR 保存函数返回地址。
- ✧ 寄存器 R0、R1 为函数返回值。若返回值超出 64 位(如结构体)，编译器将在函数调用前，在父函数的栈申请空间，并将空间入口传入子函数处理。

示例：int func2(int a, int b, int c, int d, int e, int f, int g)，调用 func2(A1,A2,A3,A4,A5,A6,A7)；



5.4 混合语言编程

当前工具支持多种文件格式的混合编译，即同时具有 c 文件的 C 语言文件和汇编语言的 asm 或 s 文件。也可以 C 语言项目 c 源码文件内部内嵌汇编完成高效的代码编排或特殊芯片操作。详见语言开发与说明中嵌汇编说明。

注：存在内嵌汇编的函数需要考虑函数声明指定优化等级与强制不内联，即可根据参数机制充分有效使用寄存器实现合理使用，也避免不开优化模式下编译具有函数返回的自动压栈与结果的不匹配错误。推荐函数全部内嵌汇编的使用，若局部内嵌需请使用高级嵌汇编描述其使用的寄存器，要注意过程堆栈 sp 的变化与影响。

5.5 特殊代码时间消耗控制

```
void delay(unsigned short i)
{
```

```
while(i--);
```

} 代码逻辑没有错误，实际上调用该函数并不能完成延时的作用。同样不能完成延时作用的样例有：

```
type funname(parameter)
{
    unsigned int I;
    Code1...

    I=2324;
    While(I--);

    Code2...
}
```

原因：代码中使用的要么传参的 R，要么是堆栈的临时变量，该功能函数结束后对象消亡，对系统似乎未起到作用，编译器的优化处理上，被视为无效代码，即优化下不输出。实际情况下代码功能需要时间的开销或功能逻辑的时间停顿。因此代码中的参数或局部变量须通过修饰符修饰。修改如下：

```
void delay(volatile unsigned short i)
{
    while(i--);
}
type funname(parameter)
{
    volatile unsigned int I;
    Code1...

    I=2324;
    While(I--);

    Code2...
}
```

注：修改 volatile 修饰的结果下，如临时变量将强制分配在堆栈空间，因此其需要加载、修改、保存的指令完成代码意图，因此代码量增加，同时 LD、ST 为双周期指令。同比其他芯片相同的参数下的延时将具有可预见性差异，因此应根据情况选择合适的参数。

注 2：也可函数添加优化等级属性 optimize("-O1")。并将内部的对象定义为 register 的关键字修饰，此时函数在基于寄存器控制次数的代码输出。

6 语言特殊开发与说明

6.1 简介

可指定变量、函数在内存中的存储地址。KF32 编译器先将指定代码分配至新的 **section** 中，再通过链接脚本将新的 **section** 分配至指定的存储地址，以此来实现固定代码的存储地址。因此每次指定存储地址时都需修改链接脚本。

工具链采用 **elf** 作为最终链接对象，其使用对应工具输出下载所需的 **hex** 或 **s19** 文件，使用工具输出问题分析的反汇编 **lst** 文件。编译组织：代码的段落使用 **.text** 段落保存代码到 **Flash** 中，**.data** 段落保存变量到 **RAM** 中。段落下布局采用文件匹配与段名匹配的组织链接地址分配顺序。其中普通代码段名为 **.text** 前缀，其常量使用 **.rdata** 前缀或 **.rodata** 前缀（局部常量和全局常量）。扩展 **ram** 函数的保存段名为 **.indata** 前缀。全局初始化变量使用 **.data** 前缀，全局未初始化变量使用 **.bss** 前缀。其中便于死段优化的选项下的函数与变量段名唯一化，其脚本中段名需要通配符*的表达，如函数名 **.text** 段名在脚本中表达为 **.text***。全局初始化变量以表内容存在记录在程序的尾部，通过脚本中变量控制或脚本语法控制，程序运行时加载初始化函数 **start** 完成 **ram** 初始化值和清零值。

6.2 C 语言代码

支持标准 C99 和 GNU 扩展，可指定选择标准 C89、C90。

支持变量和函数的属性。如指定变量 **num**、**a[3]** 及函数 **foo()** 存储到 **.const_data1**、**.const_data2** 和 **.const_text1** 的位置，此时须修改链接脚本指定 **.const_data1**、**.const_data2** 和 **.const_text1** 的地址（参考[链接脚本修改](#)章节）。如果只是配置属性定义 **RAM** 函数，不需要调整链接脚本，见 **RAM** 函数章节介绍。

```
const int num __attribute__(( section(".const_data1") )) = 1;
const int a[3] __attribute__(( section(".const_data2") )) =
{1,2,3};
//方法 1 声明
unsigned int foo(int) __attribute__(( section(".const_text1") ));
// 方法 2 定义
__attribute__(( section(".const_text1") )) unsigned int foo (int)
{
}
```

脚本的修改段入口地址意味着这种应用下，可实现函数的固定地址存放成为可能。多个函数在同一段时，会根据处理的先后顺序进行空间的顺序放置。

6.3 RAM 函数

若将函数定位到 **RAM** 中，以提高性能或特殊功能实现。工具链接脚本中默认已将段 **.indata** 存储到 **RAM** 中并设定在起始位置，用户只需将代码定义至 **.indata** 段中即可。程序启动代码“**startup 函数**”会将指定为 **RAM** 函数的函

数指令(包括变量全局初始化值)复制到 RAM 中。(注：在项目的 config 文件下的 startup 文件中定义该函数，并在 vector 文件中引导实现函数先调用再转入 main 函数)。

在 C 代码中“__attribute__((section(".indata")))”说明符用于指定函数分配至.indata 段，举例：

```
void __attribute__((section(".indata"))) foo()
{
    return;
}
```

汇编代码中“.section .indata”语句用于分配后续段代码至.indata 段，见嵌汇编部分介绍。

注意：由于 FLASH 与 RAM 之间的距离较远，所以两者间的跳转需要使用绝对跳转指令。在 C 语言中，工具会默认会选择基于标签的汇编指令完成跳转，因此 ram 函数同时需要添加属性__attribute__((long_call))，即示例参考

```
void __attribute__((section(".indata"))) __attribute__((long_call))
或
void __attribute__((long_call,section(".indata")))
```

6.4 链接脚本与修改

芯片启动机制需要固定向量表的程序地址，因此脚本添加 KEEP (*vector.o(.text))实现基于文件规则和段名称规则的优先分配。若项目中向量表多个，或者该文件通用名冲突时应该解决冲突，如更多信息确认顺序或正确的编译输出文件后缀，如 vector.c.obj。

基于嵌入式不输出不使用的对象特性，编译器输出段名基于选项 -ffunction-sections -fdata-sections。段名跟随基于对象自身名称的后缀，如.text\$fun1。在编辑脚本段落规则时，应该添加基于通配符*的后缀。

代码位置设计格式信息：

精简程序段示例：

```
.text :
{
    . = 0x0000;                /*偏移值,0 可忽略写*/
    KEEP (*vector.o(.text))    /* 中断向量表芯片要求的始地址 */
    . = (. + 3) & (-4);        /*4 字节对齐, 等效 . = ALIGN(4);*/
    __vec_end__ = .;
    *(.text*)
    *(.rdata*)
    *(.rodata*)
    . = (. + 3) & (-4);
    __text_end__ = .;          /*脚本变量服务 ram 初始化值实现*/
} > flash
```

修改链接脚本(ld 文件)SECTIONS 中.text 内容，相关代码说明：

```
.text :
{
    . = 0x0000;
```

```
KEEP (*vector.o(.text)) /* 中断向量表 */
__vec_end__ = (. + 3) & (-4);
. = 0x2000; /*段.usr1 起始地址*/
*(.usr1) /*段.usr1*/
. = 0x2100; /*段.usr2 起始地址*/
*(.usr2) /*段.usr2*/
*(.text*)
*(.rdata*)
. = 0x6100; /*段.usr3 起始地址*/
*(.usr3) /*段.usr3*/
*(.rodata*)
__text_end__ = (. + 3) & (-4);
} > flash
```

“KEEP (*vector.o(.text))” 语句用来链接中断向量表，不能删除，且该语句前需添加位置指定语句且必须与 VECTOFF 寄存器的内容一致，如 “. = 0x0000;”。

“__vec_end__ = (. + 3) & (-4);” 可用于调试功能，必须在中断向量表信息之后。

“*(.usr1)” “*(.usr2)” “*(.usr3)” 均为用户自定义的段信息，其前面指定的地址为该段所在的位置。用户可将类似的段，放置于 .text 中任意位置 (.text 位于 FLASH, .data 位于 RAM)。由于链接器分配是顺序执行的，用户需自己保证各段长度不会溢出，如上述中 .usr1 不能超过 0x2100、.rdata* 不会使用 0x6100 之后空间。

“. = 0x2100;” 表示后续内容从 0x2100 偏移地址开始分配。0x2100 表示偏移地址，其基址为 flash 空间的起始地址。“> flash” 表示将 .text 分配至 flash 空间中，flash 由 MEMORY 定义。

“__text_end__ = (. + 3) & (-4);” 必须在 .text 段最后。

精简数据区示例：

```
.data :
{
. = (. + 3) & (-4);
__data_start__ = .; /*要求为段的起始，该区域起始地址 4 字节对齐*/
KEEP (*(.indata*)) /* 在 RAM 中运行的段，即 ram 函数*/
*(.data*)
. = (. + 3) & (-4); /*4 字节对齐，等效 . = ALIGN(4);*/
__data_end__ = .;
. = (. + 3) & (-4);
__bss_start__ = .; /*推荐值同__data_end__，避免中间插入*/
*(COMMON*)
*(.bss*)
. = (. + 3) & (-4);
__bss_end__ = .;
__initial_sp = LENGTH(ram); /*可__initial_sp = 0x00010000;需要该起始加此偏
移到芯片的恰当位置，如 ram 尾部*/
} > ram
```

.data 中相应的标号，如 __data_start__、__bss_end__，其位置也不能改动和缺少。否则无法进行 RAM 空间初始化。即这里控制了传递符号信息到启动函数 startup 中。

默认空间对齐或设计空间保留的填充值为 0, 若需要其他添加可以在对应段后面添加脚本语法控制, 如 } > flash = 0x12345678

更多脚本语法见 [1d 官方文档](#)或 [ChipON 工具下的相关文档](#)。

6.5 嵌汇编

在 C 语言的函数中嵌入一段汇编语言程序段是编译器提供的“asm”功能, 形式看起来陌生, 它不是标准 C 所定义的形式, 而是编译器对 C 语言的扩充, 支持传递变量, 比单纯的汇编语言代码要复杂, 涉及到怎么分配和使用寄存器, 以及 C 代码中的变量应该存放在哪个寄存器中。

嵌汇编的一般形式为:

```
__asm__ __volatile__ ( "<asm routine>" :output:input:modify);
```

其中 `__asm__` 表示汇编代码的开始, `__volatile__` 是可选项, 作用为避免 asm 指令被删除、移动或组合。<asm routine>为汇编指令部分, 支持数字前加前缀“%”, 如%0, %1, %2 等表示使用寄存器的样板操作数。指令中有几个操作数, 就说明有几个变量需要与寄存器结合, 由编译器根据后面输出部分和输入部分的约束条件进行相应的处理。

输出部分: 用于规定输出变量如何与寄存器结合的约束, 可以有多个约束, 互相以逗号分开。每个约束以“=”开头, 接着用一个字母表示操作数类型, 然后是变量的结合约束, 如: “=r” (__dummy), r 表示目标操作数可以使用任一个通用寄存器, __dummy 为定义的 C 变量。如果为: “=m” (__dummy) 表示目标操作数是存放在内存单元 __dummy 中。

输入部分: 与输出部分相似, 但没有“=”。如果输入与输出部分约束使用同一个寄存器, 将对于的操作数编号 (如“0”, “1”, “2”等) 放在约束条件中。存在多个输入使用逗号间隔。

修改部分: 通常以“memory”作为约束条件, 表示操作完成后内存中的内容已有改变。如果原来某个寄存器的内容来自内存, 那么现在内存中这个单元的内容已经改变。当输入“r0”, “r1”时, 即当前语句分配寄存器不使用 R0 和 R1, 针对语句中使用固定寄存器时有作用。

注意: 指令部分为必选项, 其他为可选项, 当输入部分存在, 而输出部分不存在时, 分号: 要保留, 当仅修改部分“memory”存在时, 3 个分号都要保留。因为宏定义的关系, `__asm__` 可以只写 `asm`, `__volatile__` 只写 `volatile`。

主要约束字母及含义如下:

字母	含义
m	表示内存单元
r	表示任何通用寄存器
l、h	表示直接操作数
E、F	表示浮点数
g	表示“任意”
I	表示常数 (0~31)

6.5.1 普通字符串形式

```
1、asm(  
    asm_instruct1;\br/>    asm_instruct2;\br/>    asm_instruct3;\br/>);
```

编译结果所有汇编指令为一行，每条指令需用分号分隔。该形式通过连接符告知功能代码在 1 行。如果需要占用的不同的行参加下面示例。

```
2、asm(  
    "asm_instruct1"  
    "\n\t"  
    "asm_instruct2"  
    "\n\t"  
    "asm_instruct3"  
);
```

编译结果每条汇编指令为一行，\n\t 用于每一指令的分隔，也可为如下格式：

```
asm(  
    "asm_instruct1\n\t"  
    "asm_instruct2\n\t"  
    "asm_instruct3\n\t"  
);
```

6.5.2 GNU 格式的特殊汇编，支持传递参数

指令部分参见普通字符串形式，但添加输出、输出和修改的约束，格式为：

```
asm("instruction":output:input:modify);
```

如：asm("ST.w %0,%1":"=m"(a):"r"(num[3]):"memory");

该汇编执行目的为将 num[3] 赋值给 a，示例生成代码如下：

```
LD r5,#num  
LD.w r8,[r5+#3]  
LD r5,#a  
ST.w [r5],r8
```

前三条指令为变量的取值，由编译器生成并进行寄存器分配，选择可用寄存器，避免破坏寄存器中的数据；此时由于样例使用 R8，所以编译器在此函数中自动添加 PUSH R8/POP R8 的保护与还原指令。

对应 modify 信息也可以配置寄存器说明，如 "r0", "r1"。表示语句占用 r0 和 r1。

错误特殊举例说明：

```
typedef unsigned short SFRAC16;  
SFRAC16 FracMpy(SFRAC16 mul_1, SFRAC16 mul_2)  
{  
    return mul_1 * mul_2;  
}
```

根据函数的参数传递规则，参数分别使用 R0 和 R1。因此嵌汇编

`MULS R2,R0,R1` 似乎可完成乘功能，实际上该样例会存在错误情形。如果该函数添加 `inline` 属性，此时参数将直接使用调用它的函数处的直接符号。即使不内联修饰，但编译器语言优化时考虑该函数指令少，会自动内嵌代码实现更少空间或时间消耗。这 2 种情况下参数均不再一定是 `R0` 和 `R1`，因此应使用 GNU 的特殊汇编实现，示例如下：

```
asm("    Muls R2,%0,%1 \n\t": : "r"(mul 1), "r"(mul 2): "memory");
```

注：这里通过传递参数实现原始的 mul_1 和 mul_2 作为参数，在该高级嵌汇编表达下，目标语言不一定仅当前单句实现，会根据参数类型进行中转。如果函数确实为函数的调用，参数为 R0 和 R1，这里约束寄存器满足，直接参数替换为 MULS R2, R0, R1。但内嵌调用时，当原函数的待传递参数如果存放在 R4 和 R5 中时，直接参数替换为 MULS R2, R4, R5。如果参数 1 是原型的符号，会先

MOV R4, #Symba1

LD.W R4,[R4] 加载,即参数从内存对象 m 转换为寄存器对象 r。

这个样例因输出不明确，这里如果编写固定使用 R2，但也可能影响优化情况下使用 R2 缓存的目标需要重新获取。该示例建议编写为：

```
inline SFRAC16 FracMpy(SFRAC16 mul_1, SFRAC16 mul_2)
{
    SFRAC16 limit_var;
    asm volatile("MULS %0,%1,%2 \n\t"
        : "=r"(limit_var)
        : "r"(mul_1), "r"(mul_2)
        : "memory");
    return limit_var;
}
```

6.5.3 嵌汇编函数

普通字符串形式支持传递段属性信息，故编译器支持纯嵌汇编的函数。示例如下：

```
SFRAC16  FracMpy(SFRAC16 mul 1, SFRAC16 mul 2);
```

```
asm (" .section .text$FracMpy");  
asm (" .align 1");  
asm (" .exportFracMpy");  
asm (" .type FracMpy, @function");  
asm ("FracMpy:");  
  
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<  
asm (" MULS R0,R0,R1");  
  
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
  
asm (" JMP LR");  
  
asm (" .size FracMpy,.-FracMpy");
```

6.5.4 嵌汇编注意事项

1) 用户在使用 R6-R13 等前，必须对其进行入栈处理，结束时再出栈，以保护其中数据不丢失(确认工具默认已保护的可忽略)，使用时用户可参考使用 **PUSH list/POP list** 指令。**POP list** 不能使 LR(即 R13)出栈，应先使用 **POP list** 将 LR 之外的寄存器出栈，再 **POP LR**(若 LR 被入栈)。

2)KF32 编译器中在 O2 等级优化下(默认为 O2 等级), R0-R5 中可能存有中间数据,为防止数据丢失,建议将嵌汇编内容放置在一个新的子函数中,因此子函数内 R0-R5 可自由使用, R6-R13 等根据需要使用前需保护,否则 R0-R5 也可能需要保护,否则会存在运行错误。

3) 嵌汇编中可通过 C 语言的变量名、函数名直接获取变量地址、函数地址。汇编文件访问 C 文件中的全局变量、函数时,需要先在汇编文件中使用 `.extern` 声明外部变量,而后可直接使用变量名、函数名作为其地址。

4)汇编文件中未初始化的全局变量定义为 `.common a,1`,其中数值 1 表示一个字节长度。C 文件可通过 `extern` 声明汇编文件的变量为外部变量后使用。

5)汇编文件中函数需使用 `.export` 声明,如: `.export function`, C 文件可通过 `extern` 声明外部变量后使用。

6)C 文件中的宏定义与汇编中 EQU 无法通用,需分别定义。

7)若在嵌汇编中使用芯片特殊功能寄存器,则需引用汇编头文件,嵌汇编语句可在函数外使用,示例如: `asm(".include \"KF32XXX.inc\"");// 汇编包含头文件`

包含时需要增加选项传递头文件路径,否则需要全路径 `include`。工具选项 `-Wa,-I\"XXXXX/include\"` 负责 C 项目将编译选项传递头文件路径给汇编器。在使用 ChipON IDE 的开发环境是,默认已经添加了路径传递。如果使用外设库开发模式,不建议使用该头文件中定义,根据需要可仿照该头文件进行内容的声明,否则符号定义可能与外设库冲突。

8)嵌汇编中使用全局变量时,先使用 `LD R5,#global` 获取变量的内存地址,再使用 `LD/ST.[whb]` 指令进行加载/存储;

9)函数功能参数规则: R0-R4 (堆栈)用于传递参数, R0、R1 (堆栈)存放返回值, LR 自动跟踪函数返回地址。内嵌函数可根据情况自定义;

10)嵌汇编的 `sp` 操作(修改,如申请空间),若汇编嵌入在 c 函数中, c 函数的局部变量仍基于 `sp` 和编译时分配的偏移值进行操作。即嵌汇编 `sp` 的临时修改与申请须在 c 作用时还原。不支持使用伪指令 `MOV SP,#address` 的修改,拆分 `MOVL` 和 `MOVH` 实现赋值时,若中间被中断打断,中断的现场保护和中断下的函数的栈使用均会出现异常。即 SP 高位为 0 待更新的非法栈区。

6.6 汇编与嵌汇编控制与优化

6.6.1 指定代码段属性保存位置

指定常量及函数 `foo()` 存储到 `.const_data1`、`.const_data2` 和 `.const_text1` 的位置。

需修改链接脚本指定 `.const_data1`、`.const_data2` 和 `.const_text1` 的地址,链接脚本修改参考章节。

```
.section .const_data1
.long    0x1
.section .const_data2
.long    0x1,0x2,0x3
.section .const_text1
foo:
```

JMP LR

注：应遵循工具 FLASH/RAM 空间的分配原则和代码编写规范。该方法实现函数的入口地址。

6.6.2 指定程序在 RAM 中运行

在汇编程序段的代码前添加 “.section .indata”，如：

```
.section .indata
foo:
    MOV R0, R0
    .....
    .text
func:
    .....
```

如例，“.section .indata”表示后续代码存放在 RAM 中，故 foo 标号和 func 标号之间的代码均会存放在 RAM 中，即从脚本定义的 0x10000000 地址开始顺序排列存放。后序 “.text”表示新的段起始，即后续代码放在 FLASH 中。

6.6.3 特殊伪指令 EQU 定义变量

用途：将一个表达式赋予一个别名。

格式：.EQU new_name, old_name_or_const

.EQU 伪指令可以定义常数、符合，类似 C 语言中的宏定义。

.EQU 指令支持嵌套，即 old_name_or_const 可以是另一个 .EQU 的 new_name。old_name_or_const 可以是寄存器名称；old_name_or_const 支持简单的运算表达式。

.EQU 定义数组时，通过 .EQU 定义数组起始地址，用户代码保证一定长度内其他地址空间不被占用即可。

使用注意事项：

- 名为别名，不可与汇编关键词同名，否则汇编器将无法准确替换；
- 汇编器采用解析运算表达式的方式解析 EXPR，嵌套 EQU 或可运算部分将被处理，EXPR 最终被解析为最简格式；因此 EXPR 中的括号、“#”等符号可能会因运算而丢失；
- EQU 伪指令支持嵌套定义，最终解析需符合 KF32 指令集格式；

6.6.4 特殊伪指令 LD Ra,#label

用途：加载 32 位立即数（label 为 32 位立即数，符号或地址）

说明：该伪指令将 label 数值存入 flash 空间中常量表，并使用 LD Ra, [pc + #offset8] 加载，可以节约代码空间，如果有多个相同的常量还可以节约常量保存数量。

该伪指令执行 LD Ra, [pc + #offset8]，周期为 2，存储空间最小为 6 字节。

一般情况下，加载 32 位立即数时，使用该伪指令优于 MOVL/MOVH 指令。并且常量表输出在函数结尾，无需额外代码；函数代码较多，使常量表偏移超出 LD Rd, L 最大跳转范围 1024 字节时，常量表将插入在函数代码中，并通过添加

SJMP 指令连接函数代码。也就是说因为要生成常量表，如果在纯汇编语言中使用该伪指令，需要将汇编的代码按照函数段格式进行编写，即使用 `.text` 修饰新的起始，此时函数使用 LD 指令的后面代码应该小于当前指令的偏移可到底范围，纯汇编函数不支持函数中间插入常量表。

用法示例：

1、加载立即数

LD R5, #0X12345678 //将立即数 0x12345678 加载入 R5 中

2、加载符号

LD R5, #FUNCTION //将子函数入口地址 FUNCTION 加载至 R5

LJMP R5 //同调用子函数，并将子函数返回地址存入 LR

3、加载地址

LD R5, #OSC_CTL1 //将寄存器 OSC_CTL1 的地址 0x40000004 加载至 R5

LD.W R4, [R5] //读寄存器 OSC_CTL1

LD R3, #0X55AA55AA //准备向寄存器 OSC_CTL1 写入 0X55AA55AA

ST.W [R5], R3 //写寄存器 OSC_CTL1

SET [R5], #_FSCM //寄存器 OSC_CTL1 第 16 位(FSCM)置 1。

6.6.5 特殊伪指令 MOV Ra,#data

用途：可加载 32 位立即数（label 为 32 位立即数或符号）

说明：该伪指令将依据 Ra 的使用及 data 的数值大小，选择 MOV R, #data8、MOVL R, #data16 或 MOVL/MOVH 实现。所以当 data 小于 0x10000 时，Ra 为 R0-R12 或 LR 时，此伪指令效率优于 LD Ra, #label。

因此，16 位的 MOV 指令，在 data 的不同取值的情况下，会被编译为一条 32 位指令或两条 32 位指令。

纯汇编语言下需要注意该伪指令的指令使用为条数或周期不确定，附近代码的跳转表达不要使用绝对数的表示。如 JMP \$-8 JMP \$+8。一旦该间隔范围内有使用这条伪指令，但数据范围根据需要跳转的处理条数不确定，此时跳转位置将与预期不一致。不应使用该伪指令直接给 LR 和 SP 寄存器赋值，否则拆分实现过程被中断打断后，如 SP 值缺失高位中断压栈与中断下函数压栈均会出现异常。

6.6.6 相对跳转 SJMP/LJMP/JMP

相对跳转指令格式：

序号	格式	说明
1	SJMP/JMP/LJMP label	跳转至 label 处
2	SJMP/JMP/LJMP \$	表示原地跳转
3	SJMP/JMP/LJMP \$-M	表示向前跳转 M 条 16 位指令长度
4	SJMP/JMP/LJMP \$+M	表示向后跳转 M 条 16 位指令长度

JMP label 为 32 位指令，SJMP label 为 16 位指令，LJMP 分 16 位指令和 32 位指令。

对于序号 1，编译器将通过判断 label 的距离，优先选择 SJMP label 指令实现跳转功能。

后三种格式中，JMP 指令编码为 32 位，SJMP 编码为 16 位，不进行指令转换。LJMP 会执行超范围的提升。使用注意事项：

- 无法识别跳转范围内的 32 位指令，则会出现编译器无法发现的跳转错误。如下述例子中，MOVL 为 32 位指令，JMP 指令则会将 PC 跳转至 MOVL 指令编码的第 16 位，致使程序执行出错。

```
MOVL R5, #0x1234
JMP $-1
```

- 伪指令 LD R, #label 生成的常量表较长时，用户代码会被常量表截断，此时通过用户源码无法发现。所以可能跳入常量地址，以至出现编译器无法发现错误。

建议用户在使用这些跳转指令时，全部采用标号形式的跳转方式实现，如下：

```
JMP_BACK_ONE_INST:
    MOVL R5, #0x1234
    JMP JMP_BACK_ONE_INST //替代 JMP $-2 ,向上跳过一条指令
JMP_ENDLESS:
    JMP JMP_ENDLESS //替代 JMP $ ,原地跳转
```

7 ROM 固有函数库与 bootloader 支持方法

KF32 设计存在 ROM 启动引导程序，部分型号也存放着提供升级的 IAP 程序。这里示意定义与使用。**推荐基于源码的功能安全函数重实现。**

7.1 函数声明与调用

关于函数的使用已做集成功能函数，方法声明在工具头文件 ChipMessageApi.h 中，这里仅展示指针函数使用的原理方法。

函数原型

```
typedef void (*pRomFunction)(unsigned int src[], unsigned int dest[]);
```

声明函数入口地址

```
#define IAP_LOCATION 0xFFFFFFFF
```

定义函数配套参数或者联合的数组空间

```
typedef struct{
```

```
    unsigned int command[5];           // 输入参数缓存
```

```
}Rom_IAP_function_Data;
```

```
Rom_IAP_function_Data Iap_server;
```

```
unsigned int commanddata[1+1024]; // 输出状态和数据缓存,示例4K字节区
```

参数传递

```
Iap_server.command[0]=XX;
Iap_server.command[1]=0XXX;
Iap_server.command[2]=XX;
Iap_server.command[3]=commanddata;
Iap_server.command[4]=XX;
```

函数调用

```
pRomFunction Jump_To_Application =(pRomFunction) IAP_LOCATION;
Jump_To_Application((unsigned int*)&Iap_server, (unsigned
int*)&Iap_server);
```

参数与函数结果

参数 1 为配置功能函数命令编码和数据的结构指针，参数 2 为输出的结果参数指针，若无数据返回时可以与参数 1 相同，即首个内容从命令编码更新为函数执行结果。若为读取方法，参数 2 指向输出的数组区域，并数据偏移 4 字节返回，即首个内容为函数功能的执行结果。

7.2 IAP 可实现功能介绍

IAP 实现程序自升级时，应确保管理 IAP 升级代码在特定区域（可以在 Flash 中，也可以在 RAM 中），即升级时不能修改 IAP 固件程序，否则 IAP 功能执行后，IAP 升级代码自身可能被修改，从而不能正确工作。

当 IAP 管理程序定义为 RAM 程序（通讯接口初始或自身的运行 ram 程序），此时 Flash 可根据需要可全局修改。芯片重运行后程序更新，RAM 属性的 IAP 管理程序也可以被升级。

函数原型 void (IAPXXXX) (unsigned int [], unsigned int [])，参数 1 为指向输入区的缓存指针，参数 2 为指向输出的缓存指针，其首个内容为函数执行结果返回值，不同的参数具有不同的含义，具体见功能描述部分。

IAP 支持命令	命令代码
将 RAM 内容复制到 Flash	51 ₁₀
擦除扇区	52 ₁₀
扇区查空	53 ₁₀
读 FLASH 信息区	54 ₁₀
将 RAM 内容复制到 Flash 信息区	55 ₁₀
比较	56 ₁₀

返回值	意义符号	信息描述
0	CMD_SUCCESS	成功执行了命令
1	INVALID_COMMAND	无效命令
2	SRC_ADDR_ERROR	源地址没有以字为边界
3	DST_ADDR_ERROR	目标地址的边界错误
4	SRC_ADDR_NOT_MAPPED	源地址的映射不在存储器映射中。无法考证运算值适用之处
5	DST_ADDR_NOT_MAPPED	目标地址的映射不在存储器映射中。无法考证运算值适用之处
6	COUNT_ERROR	字节计数值不是 4 的倍数或是一个非法值
7	INVALID_SECTOR	扇区号无效或结束扇区号大于起始扇区号
8	SECTOR_NOT_BLANK	扇区非空
9	SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION	为写操作准备扇区的命令未执行
10	COMPARE_ERROR	源和目标数据不相等
11	BUSY	Flash 编程硬件接口忙
12	PARAM_ERROR	参数不足或无效参数
13	ADDR_ERROR	地址没有以字为边界
14	ADDR_NOT_MAPPED	地址的映射不在存储器映射中。无法考证运算值的适用之处
15	CMD_LOCKED	命令被锁定
16	INVALID_CODE	解锁代码无效
17	INVALID_BAUD_RATE	无效波特率设定
18	INVALID_STOP_BIT	无效停止位设定
19	CODE_READ_PROTECTION_ENABLED	代码读保护使能

7.2.1 将 RAM 内容复制到 Flash

应建立在当前区域 Flash 被擦除的情况下才可编程成功。

命令	将RAM内容复制到Flash
输入	命令代码：51 ₁₀ 参数1 (DST)：要写入数据字节的目标Flash 地址。目标地址的边界应当为1024字节 参数2 (SRC)：读取数据字节的源RAM 地址。该地址应当以字为边界 参数3：写入字节的数目。应当为1024 2048 3072 4096 参数4：写入方式（“S”为单字编程，其他为半页编程）

返回代码	CMD_SUCCESS SRC_ADDR_ERROR (地址不以字为边界) DST_ADDR_ERROR (地址边界错误) SRC_ADDR_NOT_MAPPED DST_ADDR_NOT_MAPPED COUNT_ERROR (字节计数值不是1024 2048 3072 4096) SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION BUSY
结果	无
描述	该命令用于编程Flash 存储器。在写FLASH之前应对对应的地址进行擦除操作。

7.2.2 擦除扇区

一个扇区的空间大小为 1K。主要用于擦除 Flash。

命令	擦除扇区
输入	命令代码：52 ₁₀ 参数0：起始页 参数1：结束页（应当大于或等于起始扇区号）
返回代码	CMD_SUCCESS BUSY SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION INVALID_SECTOR
结果	无
描述	该命令用于擦除片内Flash 存储器的一个或多个页。要擦除单个扇区可将“起始”和“结束”页设定为相同值

7.2.3 扇区查空 Flash

命令	扇区查空
输入	命令代码：53 ₁₀ 参数0：起始页 参数1：结束页（应当大于或等于起始页）
返回代码	CMD_SUCCESS SECTOR_NOT_BLANK INVALID_SECTOR
结果	结果0：状态代码为SECTOR_NOT_BLANK 时第一个非空字位置的偏移量 结果1：非空字位置的内容
描述	该命令用于对片内Flash存储器的一个或多个扇区进行查空。要查空单个扇区可将“起始”和“结束”扇区号设定为相同值

7.2.4 比较 Flash 和 RAM

配合从 RAM 数据编程进 Flash 空间后的编程结果验证。

命令	比较
输入	命令代码：56 ₁₀ 参数1（DST）：要被比较的数据字节的Flash 或RAM 起始地址。该地址应当以字为边界 参数2（SRC）：要被比较的数据字节的Flash 或RAM 起始地址。该地址应当以字为边界 参数3：待比较的字节数。计数值应当为4 的倍数
返回代码	CMD_SUCCESS COMPARE_ERROR COUNT_ERROR（字节数不是4 的倍数） ADDR_ERROR ADDR_NOT_MAPPED
结果	结果0：当状态代码为COMPARE_ERROR 时第一个不匹配字节的偏移地址
描述	该命令用来比较两个地址单元的存储器内容

7.2.5 读 FLASH 信息区

仅用于读取芯片数据空间读取。限定地址为 0x1C00~0x1FFC。

命令	读FLASH信息区
输入	命令代码：54 ₁₀ 参数1（SRT）：要读取的Flash信息区起始地址。该地址应当以字为边界 参数2：要读取的字节数，计数值应当为4的倍数 参数3：FLASH信息区标志号（0/1）
返回代码	CMD_SUCCESS COUNT_ERROR（字节数不是4 的倍数） ADDR_ERROR ADDR_NOT_MAPPED
结果	结果0：读取的FLASH信息区数据
描述	该命令用于读取FLASH信息区的数据，需要注意的是数据读出的RAM中，需要预留足够的 RAM空间进行存储(目前只支持度一个字的长度)

7.2.6 将 RAM 内容复制到 Flash 配置区

仅用于编程芯片数据空间。限定地址为 0x1C00~0x1FF8。

命令	将RAM内容复制到Flash配置区
----	-------------------

输入	命令代码：55 ₁₀ 参数1（DST）：要写入数据字节的目标Flash 地址。目标地址的边界应当为1024字节 参数2（SRC）：读取数据字节的源RAM 地址。该地址应当以字为边界 参数3：FLASH信息区标志号（0/1） 参数4：写入方式（“S”为单字编程，其他为半页编程）
返回代码	CMD_SUCCESS SRC_ADDR_ERROR（地址不以字为边界） DST_ADDR_ERROR（地址边界错误） SRC_ADDR_NOT_MAPPED DST_ADDR_NOT_MAPPED COUNT_ERROR（字节计数值不是256 512 1024 4096） SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION BUSY
结果	无
描述	该命令用于编程的Flash信息区，写且只写一页（1024byte）数据到Flash信息区。该命令不需要事先进行擦除操作

7.3 API 应用接口函数介绍

ChipON 提供基于 IAP 的函数功能应用 API，仅需包含头文件 ChipMessageApi.h，并组合调取对应的函数即可实现编程相关操作，参数应满足函数的要求。**推荐使用跟随 ChipON 样例与外设库等资源的安全功能函数。这里仅适合一般性功能实现。**

序号	函数	功能
1	unsigned int __getChipMessage(unsigned int address);	读取数据
2	unsigned int __getChipUniqueSerialNumber(unsigned int buffers[]);	获取芯片唯一 96bit 值
3	unsigned int __getChipMessages(unsigned int address,unsigned int len,unsigned int buffers[]);	读取一定长度数据
4	unsigned int __FLASH_Erase__(unsigned key,unsigned int address,unsigned int length);	Flash 擦除
5	unsigned int __FLASH_Program__(unsigned key,unsigned int address,unsigned int length,unsigned char buffers[]);	Flash 编程
6	unsigned char __FLASH_Read_One__(unsigned int address);	Flash 读取
7	unsigned int __FLASH_Read__(unsigned int address,unsigned int length,unsigned char buffers[]);	Flash 读取
8	unsigned int __FLASH_Check__(unsigned int address,unsigned int length,unsigned char buffers[]);	Flash 校验方法
9	unsigned int __FLASHCFGUSER_Program__(unsigned key,unsigned int address,unsigned int length,unsigned int buffers[]);	数据编程
10	unsigned int __FLASHCFGUSER_Read__(unsigned int address,unsigned int length,unsigned int buffers[]);	数据读取

注：关于函数的参数使用限制等具体请查看头文件内容及注释，对应函数根据需要支持设计在 Flash 中或 RAM 中，部分特殊函数固定在 RAM 中。

8 芯片中断向量表实现与说明

向量表是芯片运行的前提条件，即基于芯片设计首个 4 字节地址内容为复位下默认 MSP 地址，KungFu32 芯片为堆栈满减设计，因此该内容依赖工具变量初始值__initial_sp，一般为型号芯片 RAM 的最大地址+1，如 64K 对应 0x10000.首个 PUSH 的地址则为 0xFFFFC。次 4 字节为首个代码的函数地址入口，其他为特殊意义或中断函数的地址入口。

对应单个程序，其向量表必须存放在 0 地址，如 bootload 应用，app 应用建议存在其配置起始空间，即 bootload 可知 app 的入口而执行引导执行，脚本配置控制也简单。

向量表 7 为特殊应用，ChipON 应用于代码存在检查，即向量表书写时为 0，但输出时填入校验码，该条件作为 rom 检测代码存在并引导运行代码的前提。若 app 应用下不期望该向量表位置生成校验码，可以在链接器选项中给予关闭。

C 语言与汇编项目的语言格式存在差异，但实现意图一样。建议保留完整的 512 字节向量表空间，即向量表应用重映射时确保完整与一致。

ChipON 工具提供了默认的向量表与规则命名，向量表函数名为 weak 修饰，因此若无具体的该函数实现，向量表入口默认内容为 0，即不应该产生该中断，否则从 0 地址执行，而那里为类似函数指针的向量表而非代码空间，从而造成致命错误。

不同系列芯片的向量表可能存在不同的情况，具体根据芯片数据手册或根据 ChipON IDE 建立项目下自动生成的文件文件为准。默认该文件输出在项目的 _config 文件夹下的 vector.c 或 vector.asm 文件中，并链接脚本声明在控制在 Flash 配置空间的起始地址。如果应用代码使用向量表重映射，应该在其他文件中重新编写向量表，并运行到切换时关联到新的中断向量表地址即可。

```
.text :
{
    . = 0x0000;
    KEEP (*vector.o(.text*)) /* chip interrupt vector ,writed in
file named vector.s or vector.c */
}
```

8.1 C 语言格式框架示例摘录：

```
#define V2_0x00000008_VectorFunction _NMI_exception
#define V3_0x0000000C_VectorFunction _HardFault_exception
#define V4_0x00000010_VectorFunction _Soft4_exception
#define V5_0x00000014_VectorFunction _StackFault_exception
#define V6_0x00000018_VectorFunction _AriFault_exception

#define pFunc void (*Func) (void)
typedef struct
{
    pFunc;
}interruptVector;

typedef struct
{
    int *value; //auto variable by tool ,value is default sp
    interruptVector Reset_Enter; //Enter Function,design lead function and run to main

    interruptVector NMI_Enter;
    interruptVector HardFault_Enter;
    interruptVector Rev4_Enter;
    interruptVector StackFault_Enter;
    interruptVector AriFault_Enter;
    interruptVector Intended_Rev; // create characteristic by linker,code write 0 here

    interruptVector interrupt[120] ; // other
}VectorEnter;
void __attribute__((weak,interrupt,alias("_Default_NULL_exception"))) V2_0x00000008_VectorFunction (void);
void __attribute__((weak,interrupt,alias("_Default_NULL_exception"))) V3_0x0000000C_VectorFunction (void);
void __attribute__((weak,interrupt,alias("_Default_NULL_exception"))) V4_0x00000010_VectorFunction (void);
void __attribute__((weak,interrupt,alias("_Default_NULL_exception"))) V5_0x00000014_VectorFunction (void);
void __attribute__((weak,interrupt,alias("_Default_NULL_exception"))) V6_0x00000018_VectorFunction (void);
const VectorEnter _start __attribute__((section(".text"))) =
{
    &_initial_sp,
    startup,
    V2_0x00000008_VectorFunction,
    V3_0x0000000C_VectorFunction,
    V4_0x00000010_VectorFunction,
    V5_0x00000014_VectorFunction,
    V6_0x00000018_VectorFunction,
```

```

0,
void __attribute__((interrupt,section(".rodata"))
_Default_NULL_exception (void)
// 如上中断函数定义，实现编写函数优先，否则默认关联到该函数，其作为异常捕捉用，并避免未编写的中断函数入口地址默认 weak 下地址为 0 的执行错误。
{
    while(1){};
}

```

8.2 汇编语言格式示例：

```

.global      _start
.text
_start:
.long    __initial_sp
.long    startup
.long    _NMI_exception
.long    _HardFault_exception
.long    _Soft4_exception
.long    _StackFault_exception
.long    _AriFault_exception
.long    0
.long    _Soft8_exception
.long    _Soft9_exception
省略...

.section .rodata$_Default_NULL_exception
.align 1
.func _Default_NULL_exception
.export _Default_NULL_exception
.type _Default_NULL_exception, @function

_Default_NULL_exception:
.L2:
    JMP .L2
.size _Default_NULL_exception, .- _Default_NULL_exception
.endfunc    .weak    _NMI_exception

.set    _NMI_exception, _Default_NULL_exception
.weak   _HardFault_exception
.set    _HardFault_exception, _Default_NULL_exception
.weak   _Soft4_exception
.set    _Soft4_exception, _Default_NULL_exception
.weak   _StackFault_exception
.set    _StackFault_exception, _Default_NULL_exception
.weak   _AriFault_exception
.set    _AriFault_exception, _Default_NULL_exception
.weak   _Soft8_exception
.set    _Soft8_exception, _Default_NULL_exception
.weak   _Soft9_exception
.set    _Soft9_exception, _Default_NULL_exception
省略...

.end

```

9 更新记录

序号	日期	版本	更新内容	其他
1	2019-10	V1.0	初稿	
2	2020-3	V1.1	文档名称变更	
3	2022-2	V1.1.1	1、调整 IAP 章节的使用与间接 2、修正对齐属性拼写错误 <code>aligned</code> → <code>aligned</code> 3、脚本描述增加填充控制配置 4、更新向量表描述，c 项目实现更换为 c 语言的表达	调用函数使用
4	2022-11	V1.1.2	1、增加函数属性优化等级的描述 2、更新标准库 <code>malloc.h</code> 介绍，支持重新申请空间和空间量控制描述 3、修改描述，当前 ide 支持项目下 c 文件与汇编文件的混合编译，添加嵌汇编注意说明	
5	2023-1	V1.1.3	1、更新文件后缀的文档类型描述 2、修正如嵌汇编章节 <code>volatile modify</code> 的拼写错误 3、更新向量表示例，默认未定义中断服务函数关联到空执行体异常捕获函数 4、若干细节描述调整	
6	2024-1	V1.1.4	1、添加 <code>stdio</code> 类型下添加 <code>snprintf</code> 函数支持和限制说明 2、若干细节描述调整	
7	2026-6	V1.1.5	1、添加函数属性 <code>long_call</code> 支持 2、添加默认枚举类型的大小和参考控制 3、添加 2 中场景的库说明 4、若干细节描述调整	