



KF8ASM 用户使用手册

V1.1

目 录

1	汇编器概述	4
1.1	绝对代码	4
1.2	可重定位代码	4
2	汇编源代码语法	4
2.1	标号	5
2.2	助记符、伪指令和宏	5
2.3	操作数	5
2.4	注释	6
3	汇编器接口	6
4	表达式语法和运算法则	7
4.1	文本字符串	7
4.2	转义字符	7
4.3	保留字和段名	8
4.4	数字常量	8
4.5	算数运算符和优先级	9
5	预处理	10
6	伪指令和宏	11
6.1	宏	11
6.2	伪指令	11
6.2.1	__CONFIG	11
6.2.2	BANKSEL	12
6.2.3	.CODE	12
6.2.4	.CONSTANT	12
6.2.5	.DA	13
6.2.6	.DATA	13
6.2.7	.DB	13
6.2.8	.DE	14

6.2.9	.DW	14
6.2.10	.DEFINE	14
6.2.11	.ELSE	14
6.2.12	.END	15
6.2.13	.ENDIF	15
6.2.14	.ENDM	15
6.2.15	.EQU	15
6.2.16	.EXTERN	15
6.2.17	.EXITM	16
6.2.18	.GLOBAL	16
6.2.19	.IDATA	17
6.2.20	.IF	18
6.2.21	.IFNDEF	18
6.2.22	.LOCAL	19
6.2.23	.MACRO	19
6.2.24	.ORG	20
6.2.25	.PROCESSOR	20
6.2.26	.RADIX	20
6.2.27	.RES	20
6.2.28	.SET	21
6.2.29	.UDATA	21
6.2.30	.VARIABLE	21

附录 A	汇编指令集[助记符中*不是语法部分，有说明事项]	22
------	--------------------------	----

1 汇编器概述

kf8asm 汇编器为 KF 系列的单片机提供了一个开发汇编语言源代码的平台。

汇编器可以有两种使用方式：

- 1、生成可以直接由单片机执行的**绝对代码**。
- 2、生成可以与其他独立汇编或编译的模块链接的**可重定位代码**。

1.1 绝对代码

在绝对代码模式, 汇编语言的源文件被汇编器直接装换为 HEX 文件, 这种模式是绝对的, 因为最终地址被硬编码入源文件。

1.2 可重定位代码

在可重定位模式, 源文件被分为几个模块, 每一个模块由汇编器独立编译为**目标文件**. 编译生成的目标文件可以被放置于单片机存储器的任何地方。链接器会解决符号应用, 分配地址以及为机器代码需要更新地址的地方写入最终的地址。

链接器的输出即是绝对代码。

2 汇编源代码语法

汇编语言是一种可以用来为应用程序编写源代码的编程语言。可以使用任何 ASCII 文本编辑器来创建源代码文件。

汇编语言是一种语法松散的编程语言。

源文件的每一行四种类型的信息：

- 1、**标号**
- 2、**助记符、伪指令和宏**
- 3、**操作数**
- 4、**注释**

这些信息的顺序和位置很重要。其中标号一定要顶格写及从第一列开始。助记符不能顶格写。操作数跟在助记符的后面。注释可以跟在操作数, 助记符或者标号的后面, 并且可以从任何一列开始。

必须用**空白**或者**冒号**将标号和助记符分开，必须使用**空白**将助记符和操作数分开。必须使用**逗号**将多个操作数分开。

空白是指一个或者多个空格或者制表符。空白用于将源代码行分段。使用空白的目的是代码清晰便于阅读。除非在字符常数的内部否则多个的空白的意义与一个空格相同。

2.1 标号

标号用来表示一行，一组代码或一个常数值。跳转指令需要它。标号应该从第一列开始。后面可以跟冒号(:)、空格、制表符或换行符。标号必须以一个字母字符或一个下划线(_)开头，可以包含字母数字字符、下划线。

标号限制：

- 1、不可以以两个前导下划线开头，例如：__config。
- 2、不可以以一个前导下划线和一个数字开头，例如：_2NDLOOP。
- 3、不能是汇编器的保留字（参见第 4.3 节 “保留字和段名”）。

如果在定义标号时使用冒号，冒号将被视作标号分隔符而非标号自身的一部分。

2.2 助记符、伪指令和宏

助记符告诉汇编器对哪些机器指令进行汇编。例如，加(add)、跳转(jmp)或移动(mov)。与您自己创建的标号不同，助记符由汇编器提供。助记符不区分大小写。

伪指令是出现在源代码中的汇编器命令，但是通常不被直接编译为操作码。它们用于控制汇编器的输入、输出和数据分配。伪指令不区分大小写。

宏是用户定义的一组指令和伪指令，每当调用宏时，这些指令和伪指令将嵌入汇编器源代码同一行执行。

汇编器指令助记符、伪指令和宏调用应该从第二列或以后的列开始。如果同一行有一个标号，必须用冒号或一个或多个空格或制表符将指令与该标号分开。

2.3 操作数

操作数给出有关指令的信息，包括应该使用的数据和该指令的存储单元。必须用一个或多个空格或制表符将操作数与助记符分开。必须用**逗号**将多个操作数分开。**立即数**须注意语法格式，即十六进制与十进制表达意义不同，如 MOV R0，

#3 对比 MOV R0, #0x03, 前面语句识别为寄存器 3 赋值到 R0, 后面为立即数 3 赋值到 R0, 若按十进制书写立即数传递, 应修改为 MOV R0, ##3。

2.4 注释

注释是解释一行或数行代码的操作的文本。汇编器将分号后的任何文本视作注释。分号后直至行尾的所有字符均被忽略。

包含分号的字符串常数是允许的, 且不会与注释混淆。

注意: 上面提到的所有的标点符号(冒号、逗号、分号)必须都是相应英文标点符号。

3 汇编器接口

如果是以命令行运行汇编器, 一般的调用语法是:

kf8asm [options] asm-file

可用选项(options)如下:

选项	意义
-a<format>	选择生成的 HEX 文件的格式, format 可以是下面的四个之一: inhx8m, inhx8s, inhx16, inhx32(默认值)。
-c	生成可重定位的文件
-d	输出调试信息
-Dsymbol[=value]	与.define <symbol> <value>等效
-e[ON OFF]	在 list 文件中扩展宏
-o <file>	指定输出 HEX 文件的名字
-h	显示帮助信息
-i	忽略汇编源文件的大小写
-I <directory>	指定一个头文件目录
-l	列出支持的处理器
-M	输出依赖文件
-p <processor>	指定目标处理器
-q	退出
-r <radix>	设置进制值, 即汇编器解析无格式数字。<radix>可以是下列之一: oct、dec、hex。仍须注意操作数语法限定, #[0-9]+将整体被解析为 10 进制立即数。
-u	使用绝对路径
-v	输出汇编器的版本信息然后退出
-w[0 1 2]	设置信息级别

除非显式指定，汇编器将移除源文件名的后缀名“.asm”，然后给这个名字添加“.lst”或者“.hex”后缀作为输出的list文件和HEX文件的文件名。在同一目录下不要命名只依靠文件名中字符大小写来区分的文件名，现代的大多数操作系统的文件系统上文件名是大小写敏感的(Windows系统中，NTFS是否大小写敏感是可选的，FAT32是大小写不敏感的。依靠大小写来区分文件会造成很多难以觉察的问题)。

汇编器总会生成一个lst文件，如果没有错误，也会生成一个“.hex”文件和一个“.o”文件。

开发工具集成了参数信息，一般使用默认工具配置选项即可。

4 表达式语法和运算法则

4.1 文本字符串

“字符串”是用双引号引出的任何有效的ASCII字符（为0到127之间的十进制数）序列。可能包括引号或空字符。

在字符串中添加特殊字符的方法是在这些特殊字符前用反斜杠‘\’进行转义。应用于字符串的转义序列同样也适用于字符。

在对字符串的处理中如果找到相应的后引号，则表明字符串结束。如果在行末前仍未找到相应的后引号，字符串将在行尾处结束。虽然不可以将字符串直接延伸到第二行，但通常可以在下一行再次使用dw伪指令来达到这一目的。

dw伪指令将把整个字符串存储到连续的字单元中。如果字符串字符（字节）数为奇数，dw和data伪指令将在字符串尾填充一个字节的0(00)。

如果字符串用作立即操作数，它必须为一个字符长，否则将产生错误。需要注意的是字符串的操作方法往往以\0作为结束，除非自定义的基于给定长度的处理方法。

4.2 转义字符

汇编器接受ANSI C转义序列以表示某些特殊的控制字符：

ANSI C 转义序列

转义字符	说明	十六进制值
\a	报警字符	07
\b	退格符	08
\f	换页符	0C
\n	换行符	0A
\r	回车符	0D

\t	水平制表符	09
\v	垂直制表符	0B
\\	反斜杠	5C
\?	问号	3F
\'	单引号	27
\"	双引号	22
\000	八进制数	
\xHH	十六进制数	

4.3 保留字和段名

不能用下面的词作为标号、常数或变量名：

- 1, 伪指令（参见第 6.2 “伪指令”）。
- 2, 指令（参见附录 A “指令集”）。

此外，汇编器还保留了以下段名：

段名	用途
.code	.code 伪指令的默认段名
.idata	.idata 伪指令的默认段名
.udata	.udata 伪指令的默认段名

4.4 数字常量

汇编器支持以下常数基数格式：十六进制、十进制、八进制、二进制和 ASCII。默认的基数是十六进制；当没有用基本描述符明确指定基数时，默认的基数将决定要给目标文件中的常数赋什么样的值。

下面的表格给出了各种基数数值的规范：

类型	语法	示例
二进制	B' 二进制数字'	B' 01010101'
八进制	O' 八进制数字'	O' 777'
十进制	D' 十进制数字' . 数字	D' 100' . 100
十六进制	H' 十六进制数字 十六进制数字'	H' 9f' 0x9f
ASCII	A' 字符' '字符'	A' C' 'C'

注 1: 二进制整数为 ‘b’ 或 ‘B’ 后跟一个或多个用单引号括起的二进制数字 ‘01’。

2: 八进制整数为 ‘o’ 或 ‘O’ 后跟一个或多个用单引号括起的八进制数字 ‘01234567’。

3: 十进制整数为 ‘d’ 或 ‘D’ 后跟一个或多个用单引号括起的十进制数字 ‘0123456789’。或者，十进制整数为 ‘.’ 后跟一个或多个十进制数字 ‘0123456789’。

4: 十六进制整数为 ‘h’ 或 ‘H’ 后跟一个或多个用单引号括起的十六进制数字 ‘0123456789abcdefABCDEF’。或者，十六进制整为 ‘0x’ 后跟一个或多个十六进制数字 ‘0123456789abcdefABCDEF’。

5: ASCII 字符为 ‘a’ 或 ‘A’ 后跟一个用单引号括起的字符。或者，ASCII 字符是一个单引号括起的字符。

4.5 算数运算符和优先级

算术运算符可与下表中规定的伪指令及其变量一起使用。

注意：这些运算符不能与程序变量一起使用，它们仅可以与伪指令一起使用目的是用于控制条件编译等。

表中运算符的次序与其优先级相对应，其中第一个运算符优先级最高，最后一个运算符优先级最低。优先级指的是运算符在代码语句中的执行顺序。

算数运算符（按优先级排列）

运算符		优先级值	示例
\$	当前程序计数器	1	goto \$ + 3
!	非	2	if !(a == b)
-	取反（2 的补码）	2	-1 * length
~	取补	2	flags = ~flags
*	乘	3	a = b * c
/	除	3	a = b / c
%	取余	3	a = b % c
+	加	4	length = len * 8 + 3
-	减	4	length = (len - 1) * 2
<<	左移	5	flags = flags << 1
>>	右移	5	flags = flags >> 1
>=	大于等于	6	if id >= root
>	大于	6	if id > root
<	小于	6	if id < root
<=	小于等于	6	if id <= root

==	等于	6	if id == root
!=	不等于	6	if id != root
&	位与	7	flas = flags & ERR_BIT
^	位异或	7	flas = flags ^ ERR_BIT
	位或	7	flas = flags ERR_BIT
&&	逻辑与	8	if (len == 51) && b
	逻辑或	8	if (len == 62) b
=	赋值	9	index = 1
+=	加赋值	9	index += 1
-=	减赋值	9	index -= 1
*=	乘赋值	9	index *= 1
/=	除赋值	9	index /= 1
%=	取余赋值	9	index %= 1
>>=	右移赋值	9	index >>= 1
<<=	左移赋值	9	index <<= 1
&=	与赋值	9	index &= 1
=	或赋值	9	index = 1
^=	异或赋值	9	index ^= 1
++	加加	9	i ++
--	减减	9	i --

++和--运算符只能在一行中单独使用，不能嵌套在其他表达式求值中。

5 预处理

在下面的代码中：

```
.INCLUDE foo.inc
```

汇编器将会读取 foo.inc 文件直到这个文件的结尾，然后汇编器再去处理 .INCLUDE foo.inc 行之后的代码行。

.DEFINE 伪指令将由汇编器特别处理。

只要汇编器处理到类似如下的代码行：

```
.DEFINE X Y
```

每一个后续遇到的 X 都将被替换为 Y。直到源文件结束或者遇到代码行：

```
.UNDEFINE X
```

使用 .INCLUDE 指令包含合适的现有文件在你的代码中，汇编器将会自动搜索默认目录寻找指定名字的头文件。

6 伪指令和宏

在绝对代码模式，使用 .ORG 伪指令指示汇编器分配汇编代码的起始存储地址。如果没有使用 .ORG 指定，汇编器将会默认从 0x0000 存储地址生成代码。

6.1 宏

kf8asm 支持简单的宏模式，可以像下面这样定义使用宏。

定义：

```
SetDate      .macro  parm1,parm2
               mov  r0,#parm1
               MOV  R1,#parm2
               ST   [R0],R1
               .endm
```

使用： SetDate 33,25

实际结果：

9821	M	mov r0,#parm1
9919	M	MOV R1,#parm2
F748	M	ST [R0],R1

需要注意的是宏定义的名字必须顶格写，可以不带参数，默认数据为 10 进制。不支持代码中“#”后面带“0x”更改为十六进制模式，立即数需要用“#”修饰，否则作为寄存器地址意义存在。

6.2 伪指令

6.2.1 __CONFIG

__CONFIG <expression>

注意：前面有两个下划线。

该伪指令作为配置字值的表达方法，如 “__config 0x2007,0x3ffc”。其中前面数值为地址，后面数值为赋值结果。地址信息应与芯片信息一致。其他地址数据不作用，另外前面应该有空格或制表符，即不能定格写。

赋值结果可以使用表达式，如 0x3ffc&0x0328|0x415A。

6.2.2 BANKSEL

BANKSEL <label>

此伪指令指示汇编器和链接器生成存储区选择代码，以将存储区设置为包含指定 label 的存储区。应只指定一个 label。必须已经在前面定义了此标号。

链接器将生成相应的存储区选择代码及生成针对 PSW 寄存器中 RP_x 位的相应置位/清零指令，或 BANK 寄存器赋值。如果器件仅包含一个 RAM 区的存储区，将不会生成任何指令。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。
简单示例：

```
banksel var      ;为 var 选择正确的 bank
add var, r0      ;操作 var
```

该伪指令作为不显式指定变量所在区域，或指定区域但编写代码不特别考虑区所在下具有特殊意义。

6.2.3 .CODE

<label> .CODE <expression>

此伪指令声明开始一段程序代码。如果未指定 label，该段被命名为.code，但一个项目中最多存在一个未命名的段。起始地址被初始化为 expression 指定的地址，如果没有指定地址，则在链接时分配起始地址。

此伪指令仅可用在可重定位模式中。

没有“end code”伪指令。当定义了另一个代码段或数据段或者到达文件的末尾时，一个段的代码自动结束。

针对汇编项目的编写，如果存在多个源文件时应该在代码的开始部分添加该伪指令，地址可以不指定。

6.2.4 .CONSTANT

.CONSTANT <label> = <expression> [, <label> = <expression>]*

创建在汇编器表达式中使用的符号。一旦常数被初始化之后，就不可以被再次改变了，而且赋值时表达式必须完全可解析及不可以包含需要重定位的标号。

这是声明为常数和那些声明为变量或者由 .set 伪指令创建的符号之间的主要区别。除此之外，常数和变量可以在绝对代码表达式中互换使用。

此伪指令使用情况：绝对代码或可重定位代码。

6.2.5 .DA

```
<label> .DA <expression> [, <expression>]*
```

在程序存储器中存储压缩字符串。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。

6.2.6 .DATA

```
.DATA <expression> [, <expression>]*
```

用数据初始化一个或多个程序存储器字。数据的形式可以是：常数、可重定位或外部标号，或是上述任何一种的表达式。

注意：这条伪指令不是定义数据段的伪指令，定义数据段的伪指令参见 .idata 和 .udata。

6.2.7 .DB

```
<label> .DB <expression> [, <expression>]*
```

用 8 位值保留程序存储器字。多个表达式继续连续地填充字节，直到表达式结束为止。如果有奇数个表达式，最后一个字节将是零。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。
简单示例：

```
.db 0x0f, 't', 0x0f, 'e', 0x0f, 's', 0x0f, 't', '\n'
ASCII: 0x0F74 0x0F65 0x0F73 0x0F74 0x0a00
一般可以配合 idata 伪指令使用，定义初始的 RAM 信息。示例：
ID_debug_touch_0 .idata
    _TOUCH_CH_TRS_EN ;// 数组名
    .de 0x01
```

```
.db 0x00  
.db 0x00  
.dw 0x1234
```

6.2.8 .DE

```
<label> .DE <expression> [, <expression>]*
```

定义 EEPROM 数据，每一个字符串中的字符将存储在一个单独的字中。

当前不支持该伪指令。

6.2.9 .DW

```
<label> .DW <expression> [, <expression>]*
```

定义字数据，即字长 16bit，占用一个程序地址空间。

如 “.DW 0x0000”，即编译顺序地址下放入数值 0x0000，这是一条 NOP 指令。

6.2.10 .DEFINE

```
.DEFINE name <string>
```

定义文本替换符号。此伪指令定义一个文本替换字符串。在汇编代码中的任何位置遇到 name 时，它将会被 string 替换。使用不带 string 的伪指令将引起 name 的定义在内部被标记，并可以再 ifdef 伪指令中用于条件检测。

显式的 RAM 使用可以使用该伪指令定义。不区分大小写，如：

```
“.DEFINE aaa 0x183 ”
```

代码中的 aaa 作为变量意义存在，地址在 1 区的 0x83 地址。

6.2.11 .ELSE

```
.ELSE
```

与 .if 伪指令一起使用，以在 .if 求值为 FALSE 时提供汇编代码的备用路径。

6.2.12 .END

`.END`

在任何汇编程序中需要至少一条 `.end` 伪指令来表示编译的结束。在单个汇编文件程序中，必须且只能使用一条 `.end` 伪指令。

需要注意不要包含含有 `.end` 的文件，因为这些文件会使汇编过早地停止。

6.2.13 .ENDIF

`.ENDIF`

表示条件汇编的结束。

每使用一条 `.if` 伪指令，都必须有一条对应的 `.endif`。`.if` 和 `.endif` 不是指令，仅用于条件汇编代码。

6.2.14 .ENDM

`.ENDM`

终止由 `.macro` 开始的宏定义。

每使用一条 `.macro` 伪指令，都必须有一条对应的 `.endm`。

6.2.15 .EQU

`<label>.EQU <expression>`

定义一个汇编器常数。`expression` 的值被赋值给 `label`。`label` 的值不可以再改变。

Defing 伪指令外常用的定义变量的表达方式。如：

“bbb .EQU 0x183”，但需要注意的时，这里必须定格写。

6.2.16 .EXTERN

`.EXTERN <symbol> [, <symbol>]*`

声明一个外部定义的标号。此伪指令只可以用于可重定位的代码中。

此伪指令声明可在当前模块中使用但在另一个模块中被定义为全局标号的符号名称。必须先包含 `.extern` 语句，才能使用该 `label`。在这行上必须至少指定一个标号。

只要项目中有一个以上的文件，就可以使用此伪指令。当某文件使用一个标号（通常是变量）时，则该文件将使用 `.extern`。`.global` 将在另一个文件中使用从而使该标号可被其他文件看到。必须按照规定使用这两条伪指令，否则该标号在其他文件中将是不可见的。这里可以是变量的声明，也可以是函数的声明。

简单示例：

```
.extern func
...
call func
```

6.2.17 .EXITM

```
.EXITM
```

在汇编时，强制立即从宏扩展返回。作用与遇到 `.endm` 伪指令时相同。

6.2.18 .GLOBAL

```
.GLOBAL <symbol> [, <symbol>]*
```

此伪指令声明当前模块中定义的并且对于其他模块可用的符号名。在这行上必须至少指定一个标号。

此伪指令在以下类型的代码中使用：可重定位代码。

当项目使用多个文件时，需要生成可链接目标代码。当发生这种情况时，可以使用 `.global` 和 `.extern` 伪指令。

`.global` 用于让标号对于其他文件可见。`.extern` 必须在使用标号的文件中使用以便让标号在该文件中可见。

简单示例：


```
.global Var1, Var2
.global AddThree
.ldata
    Var1 .res 1
    Var2 .res 1
.code
AddThree:
    add r0, 3
```

6.2.19 .IDATA

<label> .IDATA <expression>

此伪指令声明开始一段已初始化的数据。如果未指定 label，此段被命名为 .idata。起始地址被初始化为指定的地址，如果没有指定地址，则在链接时分配起始地址。存储空间将会被分配，初始化数据将被放置在 ROM 中，用户必须提供代码加载初始化数据到已经分配的存储空间中。

此伪指令在以下类型的代码中使用：可重定位代码。

使用此伪指令初始化变量，或者使用 .ldata 伪指令，然后用代码中的值初始化变量。建议总是初始化变量。

简单示例：

```
.idata
    limitL .dw 0
    limitH .dw D'300'
    gain .dw D'5'
    flags .db 0
    string .db 'Hi there!'
```

使用该宏声明的内容为在 RAM 中需要的初始化赋予的值。因此需要代码来实现。如果存在多个 idate 段，应该分别给予 <label> 的命名区分。

在代码的起始部分需要调用初始化函数，即：

```
PAGESEL    _cinit
CALL       _cinit      ;//所需 做变量初始值处理 idate段
PAGESEL$
```

6.2.20 .IF

`.IF <expression>`

开始执行条件汇编的代码块:如果 `expression` 的值为 TRUE, 将汇编紧跟在 `.if` 后的代码。否则, 将跳过后续的代码直到遇到 `.else` 或 `.endif` 伪指令。值为 0 的表达式视为逻辑 FALSE。值为任何其他值的表达式视为逻辑 TRUE。`.if` 伪指令根据表达式的逻辑值来运行。

此伪指令在以下类型的代码中使用: 绝对代码或可重定位代码。

此伪指令不是指令, 只是用于控制哪些代码块将被汇编。

简单示例:

```
.if version == 100      ;检查当前版本
    mov r0, 0x0a
    mov io_1, r0
.else
    mov r0, 0x01a
    mov io_2, r0
.endif
```

```
.IFDEF
    .IFDEF <symbol>
```

如果前面已经定义了 `label` (通常通过执行 `.define` 伪指令)。汇编将继续进行直到遇到匹配的 `.else` 或 `.endif` 伪指令为止。

此伪指令在以下类型的代码中使用: 绝对代码或可重定位代码。

此伪指令不是指令, 只是用于控制哪些代码块将被汇编。在调试过程中使用此伪指令删除或添加代码, 而无需对大块的代码进行注释。

6.2.21 .IFNDEF

`.IFNDEF <symbol>`

如果前面没有定义 `label` 或已经通过执行 `.undefine` 伪指令解除定义, 则伪指令后的代码将被汇编。汇编将被使能或禁止直到遇到下一条匹配的 `.else` 或 `.endif`

伪指令。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。

此伪指令不是指令，只是用于控制哪些代码块将被汇编。在调试过程中使用此伪指令删除或添加代码，无需对大块的代码进行注释。

6.2.22 .LOCAL

```
.LOCAL <symbol> [[ =<expression>], [<symbol> [=<expression>]]*]
```

声明将指定的数据元素看作宏的局部元素。label 可以与在宏定义之外声明的其他标号相同，两个标号之间不会有冲突。

如果递归调用宏，每个调用都会有自己的局部复制。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。

6.2.23 .MACRO

```
<label> .MACRO [<symbol> [, <symbol>]*]
```

宏是一系列指令，可以使用一个宏调用将这一系列指令插入到汇编源代码中。必须首先定义宏，然后才能在后续的源代码中引用它。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。

简单示例：

；定义宏

```
Read .macro device, buffer, count
    mov r0, device
    mov ram_20, r0
    mov r0, buffer      ;buffer 地址
    mov ram_21, r0
    mov r0, count        ;计数
    call sys_21          ;调用子例程
.endm
```

；使用上面定义的宏

```
Read 0x0, 0x55, 0x05
```

6.2.24 .ORG

`.ORG <expression>`

将后续代码的程序起始处设置在由 expression 定义的地址。如果未指定.org 伪指令，将从地址 0 开始生成代码。

6.2.25 .PROCESSOR

`.PROCESSOR<symbol>`

选择目标处理器。**暂无作用。**

6.2.26 .RADIX

`.RADIX <symbol>`

设置汇编器的默认基数值。symbol 从 oct、dec、hex 中选取，分别代表八进制、十进制、十六进制。汇编器将会使用这里指定的基数值去解释没有显式指定基数值的数值。

`MOV R0, #100` 默认为 10 进制的 100, 16 进制的 0x64。但如果该宏声明默认按照 16 进制下的立即数参数结果为 0x100。

6.2.27 .RES

`.RES <expr>`

使存储器单元指针从当前的单元向前移在 expr 中指定的值。被用于保留数据存储空间。

`.res` 最常用的用途是在可重定位代码中存储数据。需要与 `udata` 联合使用。

示例：

```
_MAIN_RAM_2    .udata
    Touch_Flag    .res    1
    Touch_Flag1    .res    1
```

即在 MAIN_RAM_2 表示的 RAM 段下，第一个地址下变量为 Touch_Flag。第二地址下为变量 Touch_Flag1。并且 Touch_Flag 和 Touch_Flag1 一定在同一个 RAM 分区中。

6.2.28 .SET

```
<label> .SET <expression>
```

将由 expression 指定的有效汇编器表达式的值赋给 label。set 伪指令的功能与 equ 伪指令相似，不同之处在于设置的值后来会被其他 set 伪指令更改。

此伪指令在以下类型的代码中使用：绝对代码或可重定位代码。

6.2.29 .UDATA

```
<label> .UDATA <expression>
```

此伪指令声明开始一段未初始化的数据。如果未指定标号，此段被命名为 .udata。起始地址被初始化为指定的地址，如果没有指定地址，则在链接时分配起始地址。此段不会生成代码。应该使用 .res 伪指令来为数据保留空间。

此伪指令在以下类型的代码中使用：可重定位代码。

6.2.30 .VARIABLE

```
.VARIABLE <label> [= <expression>, <label> [= <expression>]]*
```

声明名字为 label 的变量，label 可以被重新赋值。label 的值不一定要在声明时指定。

附录 A 汇编指令集[助记符中*不是语法部分，有说明事项]

助记符、操作数	指令格式	指令说明	周期	影响标志
NOP	0000_0000_0000_0000	空操作指令	1	
NOPZ	1111_1111_1111_1111	空操作指令	1	
CRET	0000_0000_0000_1000	子程序返回指令	2	
RRET Rn, #data *	1011_0rrr_kkkk_kkkk	立即数送到 Rn 中返回	2	
IRET	0000_0000_0000_1001	中断返回指令	2	
CWDT	0000_0000_0110_0100	WDT 清 0	1	
IDLE	0000_0000_0110_0011	进入休眠模式	1	
数据传送指令				
MOV dir	0000_1111_ffff_ffff	$dir \leftarrow (dir)$	1	Z
MOV Rn, dir	0101_rrr0_ffff_ffff	$Rn \leftarrow (dir)$	1	
MOV dir, Rn	0101_rrr1_ffff_ffff	$dir \leftarrow (Rn)$	1	
MOV Rn, #data *	1001_lrrr_kkkk_kkkk	$Rn \leftarrow data$	1	
MOV Rn, Rs	1111_1000_1lss_srrr	$Rn \leftarrow (Rs)$	1	
LD Rn, [Rs]	1111_0111_00ss_srrr	$Rn \leftarrow ((Rs))$	1	
ST [Rn], Rs	1111_0111_0lss_srrr	$(Rn) \leftarrow (Rs)$	1	
SWAPR Rn, dir	0100_rrr0_ffff_ffff	$Rn<7:4>=dir<3:0>$ $Rn<3:0>=dir<7:4>$	1	
SWAP dir	0100_rrr1_ffff_ffff	$dir<7:4>=dir<3:0>$ $dir<3:0>=dir<7:4>$	1	
MOVB #data *	1110_0001_kkkk_kkkk	$BANK \leftarrow data$	1	
MOV P #data *	1110_0000_kkkk_kkkk	$PCH \leftarrow data$	1	
算术运算指令				
ADD Rm, dir	0010_0rr0_ffff_ffff	$Rm \leftarrow (Rm) + (dir)$	1	CY、DC、Z
ADD dir, Rm	0010_0rr1_ffff_ffff	$dir \leftarrow (Rm) + (dir)$	1	CY、DC、Z
ADD Rn, #data *	1000_0rrr_kkkk_kkkk	$Rn \leftarrow (Rn) + data$	1	CY、DC、Z
ADD Rn, Rs	1111_1000_00ss_srrr	$Rn \leftarrow (Rn) + (Rs)$	1	CY、DC、Z
SUB Rm, dir	0011_lrr0_ffff_ffff	$Rm \leftarrow (dir) - (Rm)$	1	CY、DC、Z
SUB dir, Rm	0011_lrr1_ffff_ffff	$dir \leftarrow (dir) - (Rm)$	1	CY、DC、Z
SUB Rn, #data *	1010_0rrr_kkkk_kkkk	$Rn \leftarrow data - (Rn)$	1	CY、DC、Z
SUB Rn, Rs	1111_1000_0lss_srrr	$Rn \leftarrow (Rs) - (Rn)$	1	CY、DC、Z
CMP Rn, #data *	1111_0010_1kkk_krrr	-	1	CY、DC、Z
CMP Rn, Rs	1111_0001_10ss_srrr	-	1	CY、DC、Z
INC dir	0000_1011_ffff_ffff	$dir \leftarrow (dir) + 1$	1	Z
INCR dir	0000_1010_ffff_ffff	$R0 \leftarrow (dir) + 1$	1	Z
INC Rn	1111_1111_0001_0rrr	$Rn \leftarrow (Rn) + 1$	1	Z
DEC dir	0000_0111_ffff_ffff	$dir \leftarrow (dir) - 1$	1	Z
DECR dir	0000_0110_ffff_ffff	$R0 \leftarrow (dir) - 1$	1	Z
DEC Rn	1111_1111_0000_1rrr	$Rn \leftarrow (Rn) - 1$	1	Z
逻辑运算指令				
AND Rm, dir	0010_lrr0_ffff_ffff	$Rm \leftarrow (Rm) \wedge (dir)$	1	Z
AND dir, Rm	0010_lrr1_ffff_ffff	$dir \leftarrow (dir) \wedge (Rm)$	1	Z
AND Rn, #data *	1000_lrrr_kkkk_kkkk	$Rn \leftarrow (Rn) \wedge data$	1	Z

助记符、操作数	指令格式	指令说明	周期	影响标志
AND Rn, Rs	1111_1000_10ss_srrr	$Rn \leftarrow (Rn) \wedge (Rs)$	1	Z
ORL Rm, dir	0011_0rr0_ffff_ffff	$Rm \leftarrow (Rm) \vee (dir)$	1	Z
ORL dir, Rm	0011_0rr1_ffff_ffff	$dir \leftarrow (dir) \vee (Rm)$	1	Z
ORL Rn, #data *	1001_0rrr_kkkk_kkkk	$Rn \leftarrow (Rn) \vee data$	1	Z
ORL Rn, Rs	1111_1001_00ss_srrr	$Rn \leftarrow (Rn) \vee (Rs)$	1	Z
XOR Rm, dir	0001_1rr0_ffff_ffff	$Rm \leftarrow (Rm) \oplus (dir)$	1	Z
XOR dir, Rm	0001_1rr1_ffff_ffff	$dir \leftarrow (dir) \oplus (Rm)$	1	Z
XOR Rn, #data *	1010_1rrr_kkkk_kkkk	$Rn \leftarrow (Rn) \oplus data$	1	Z
XOR Rn, Rs	1111_1001_01ss_srrr	$Rn \leftarrow (Rn) \oplus (Rs)$	1	Z
CLR Rn	0000_0010_xxxx_1rrr	$Rn = 0$	1	Z
CLR dir	0000_0011_ffff_ffff	$dir = 0$	1	Z
CPLR dir	0000_0100_ffff_ffff	$R0 \leftarrow \neg (dir)$	1	Z
CPL dir	0000_0101_ffff_ffff	$dir \leftarrow \neg (dir)$	1	Z
CPL Rn	1111_1111_0000_0rrr	$Rn \leftarrow \neg (Rn)$	1	Z
RRCR dir	0001_0000_ffff_ffff	$R0 \leftarrow (dir)$ 带进位 C 循环右移 1 位	1	CY
RRC dir	0001_0001_ffff_ffff	$dir \leftarrow (dir)$ 带进位 C 循环右移 1 位	1	CY
RRC Rn	1111_1111_0010_0rrr	$Rn \leftarrow (Rn)$ 带进位 C 循环右移 1 位	1	CY
RLCR dir	0001_0010_ffff_ffff	$R0 \leftarrow (dir)$ 带进位 C 循环左移 1 位	1	CY
RLC dir	0001_0011_ffff_ffff	$dir \leftarrow (dir)$ 带进位 C 循环左移 1 位	1	CY
RLC Rn	1111_1111_0001_1rrr	$Rn \leftarrow (Rn)$ 带进位 C 循环左移 1 位	1	CY
位操作指令				
CLR dir, b	0110_0bbb_ffff_ffff	将 dir 的 b 位清 0	1	
SET dir, b	0110_1bbb_ffff_ffff	将 dir 的 b 位置 1	1	
CLR Rn, b	1111_1110_00bb_brrr	将 Rn 的 b 位清 0	1	
SET Rn, b	1111_1110_01bb_brrr	将 Rn 的 b 位置 1	1	
转移指令				
DECJZ dir	0000_1000_ffff_ffff	$R0 \leftarrow (dir) - 1$, 为 0 跳过下一条指令	1/2	
DECJZ dir	0000_1001_ffff_ffff	$dir \leftarrow (dir) - 1$, 为 0 跳过下一条指令	1/2	
DECJZ Rn	1111_1111_0101_1rrr	$Rn \leftarrow (Rn) - 1$, 为 0 跳过下一条指令	1/2	
INCRJZ dir	0000_1100_ffff_ffff	$R0 \leftarrow (dir) + 1$, 为 0 跳过下一条指令	1/2	
INCRJZ dir	0000_1101_ffff_ffff	$dir \leftarrow (dir) + 1$, 为 0 跳过下一条指令	1/2	
INCRJZ Rn	1111_1111_0101_0rrr	$Rn \leftarrow (Rn) + 1$, 为 0 跳过下一条指令	1/2	
JNB dir, b	0111_0bbb_ffff_ffff	dir 的 b 位为 0 跳过下一条指令	1/2	
JB dir, b	0111_1bbb_ffff_ffff	dir 的 b 位为 1 跳过下一条指令	1/2	
JNB Rn, b	1111_0111_10bb_brrr	Rn 的 b 位为 0 跳过下一条指令	1/2	
JB Rn, b	1111_0111_11bb_brrr	Rn 的 b 位为 1 跳过下一条指令	1/2	
JMP #data12 *	1100_kkkk_kkkk_kkkk	无条件转移指令	2	
CALL #data12 *	1101_kkkk_kkkk_kkkk	子程序调用指令	2	

注：dir 为通用寄存器或特殊功能寄存器；Rn、Rs 表示 R0~R7；Rm 表示 R0~R3；#data 表示 8 位立即数，#data12 表示 12 位立即数，其中立即数应按十六进制书写，若为十进制应书写为##data 或##data12；b 表示寄存器的第 b 位；[Rn] 表示 Rn 中的数值指向的地址中数据；() 表示特殊功能寄存器、通用数据寄存器或寄存器组中的数据。

带*的指令 MOVb #data、MOVP #data 和 CMP Rn, #data 为部分型号存在，具体参照芯片的数据手册。