ARM 汇编语言源程序格式

来源: MCU 嵌入式领域

常用 ARM 源程序文件类型

```
        CODE32                          ;32 位的 ARM 指令段
        AREA codesec, CODE, READONLY    ;代码段，名称为 codesec,属性为只读
main PROC                               ;函数 main
        STMFD       sp!,{lr}            ;保存必要的寄存器和返回地址到数据栈
        ADR         r0,strhello         ;取标签 strhello 代表的地址值
        BL          _printf             ;调用 C 运行时库的_printf 函数打印
                                        ; “Hello world!”字符串
        BL          welcomefun          ;调用子函数 welcomfun
        LDMFD       sp!,{pc}            ;恢复寄存器值
strhello                                ;strhello 代表本地字符串的地址
        DCB         "Hello world!\n\0"  ;定义一段字节空间
        ENDP                            ;函数 main 结束
```

汇编语言程序的结构 1

```
        CODE32                          ;32 位的 ARM 指令段
        AREA codesec, CODE, READONLY    ;代码段，名称为 codesec,属性为只读
main PROC                               ;函数 main
        STMFD       sp!,{lr}            ;保存必要的寄存器和返回地址到数据栈
        ADR         r0,strhello         ;取标签 strhello 代表的地址值
        BL          _printf             ;调用 C 运行时库的_printf 函数打印
                                        ; “Hello world!”字符串
        BL          welcomefun          ;调用子函数 welcomfun
        LDMFD       sp!,{pc}            ;恢复寄存器值
strhello                                ;strhello 代表本地字符串的地址
        DCB         "Hello world!\n\0"  ;定义一段字节空间
        ENDP                            ;函数 main 结束
```

汇编语言程序的结构 2

```
        EXPORT main
        ;引入三个 C 运行时库函数和 ARM 库
        IMPORT _main
        IMPORT __main
        IMPORT _printf
        IMPORT ||Lib$$Request$$armlib||, WEAK
```

汇编语言程序的结构 3

```
EXPORT main
;引入三个 C 运行时库函数和 ARM 库
IMPORT _main
IMPORT __main
IMPORT _printf
IMPORT ||Lib$$Request$$armlib||, WEAK
```

汇编语言程序的结构 4

ARM 的汇编语言程序一般由几个段组成，每个段均由 AREA 伪操作定义。

段可以分为多种，如代码段、数据段、通用段，每个段又有不同的属性，如代码段的默认属性为 READONLY，数据段的默认属性为 READWRITE。

本程序定义了两个段，第一个段为代码段 codesec，它在存储器中存放用于程序执行的代码以及 main 函数的本地字符串；第二个段为数据段 constdatasec，存放了全局的字符串，由于本程序没有对数据进行写操作，该数据段定义属性为 READONLY。

汇编语言的行构成 1

格式：

[标签] 指令/伪操作/伪指令 操作数 [;语句的注释]

所有的标签必须在一行的开头顶格写，前面不能留空格，后面也不能跟 C 语言中的标签一样加上“:”；

ARM 汇编器对标识符的大小写敏感，书写标号及指令时字母的大小写要一致；

注释使用“;”符号，注释的内容从“;”开始到该行的结尾结束

汇编语言的行构成 2

标签

标签是一个符号，可以代表指令的地址、变量、数据的地址和常量。

一般以字母开头，由字母、数字、下划线组成。

当符号代表地址时又称标号，可以以数字开头，其作用范围为当前段或者在下一个 ROUT 伪操作之前。

指令/伪操作

指令/伪操作是指令的助记符或者定义符，它告诉 ARM 的处理器应该执行什么样的操作或者告诉汇编程序伪指令语句的伪操作功能。

汇编语言的标号 1

标号代表地址。

标号分为段内标号和段外标号。段内标号的地址值在汇编时确定，段外编号的地址值在链接时确定 。

在程序段中，标号代表其所在位置与段首地址的偏移量。根据程序计数器（PC）和偏移量计算地址即程序相对寻址。

在映像中定义的标号代表标号到映像首地址的偏移量。映像的首地址通常被赋予一个寄存器，根据该寄存器值与偏移量计算地址即寄存器相对寻址。

例如：

```
loop SUBS r0,r0,#1 ;每次循环使 r0=r0-1
BNE loop ;跳转到 loop 标号去执行
```

汇编语言的标号 2

在宏中也可以使用局部符号。

局部标号是 0~99 的十进位数开始，可以重复定义。

局部标号引用格式：

%{F|B}{A|T} N{routname}

% ：局部标号引用操作。

F ：编译器只向前搜索。

B ：编译器只向后搜索。

A ：编译器搜索宏的所有嵌套层次。

T ：编译器搜索宏的当前层。

例如：

01 SUBS r0,r0,#1 ;每次循环使 r0=r0-1

BNE %B01 ;跳转到 01 标号去执行

汇编语言的常量

常量：其值在程序运行过程中不能被改变的量。

（1）数字常量：数字常量有 3 种表示方式：

十进制数，如 1、2、123

十六进制数，如 0x123,0xabc

n 进制数，形式为 n_XXX，n 的范围是 2 到 9，XXX 是具体数字

（2）字符常量：由单引号及中间的字符组成，包括 C 语言中的转义字符，如’a’,’\n’

（3）字符串常量：由一对双引号及中间的字符串表示，中间也可以使用 C 语言中的转义字符，比如：“abcdef\0xa\r\n”

（4）逻辑常量：{TRUE},{FALSE},注意带大括号

汇编程序的变量代换 1

这里所说的变量，是相对于汇编程序的“变量”，是用于汇编程序进行处理的，但一旦编译到程序中，则不会改变，成为常量。

在字符串变量的前面有一个$字符,在汇编时编译器将用该字符串变量的内容代替该串变量。

在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并用该十六进制的字符串代换“$”后的数字变量。

需要将“$”字符加入到字符串中，可以用“$$”代替，此时编译器将不再进行变量代换，而是把“$$”看作一个“$”。

在两个“|”之间的“$”并不进行变量的代换，但如果“|”在双引号内，则将进行变量代换。

使用“.”来表示字符串中变量名的结束。

汇编程序的变量代换 2

```
        AREA ||.text||, CODE, READONLY ;代码段，名称为||.text||,属性为只读
        GBLS        str1                ;声明 str1 为全局字符串
        GBLS        str2                ;声明 str2 为全局字符串
        GBLL        l1                  ;声明 l1 为全局逻辑变量
        GBLA        num1                ;声明 num1 为全局数字变量
l1      SETL        {TRUE}
num1    SETA        0x4f
str1        SETS        "bbb"
str2        SETS        "aaa str1:$str1. l1:$l1,a1:$num1.ccc"   ;str2 包含
                                                                ;了多个变量
main PROC                           ;函数 main
        STMFD       sp!,{lr}        ;保存必要的寄存器和返回地址到数据栈
        ADR         r0,strhello
        BL          _printf         ;调用 C 运行时库的_printf 函数打印字符串
        LDMFD       sp!,{pc}        ;恢复寄存器值
strhello                            ;strhello 代表本地字符串的地址
        DCB         "$str2\n\0"     ;定义一段字节空间
        ENDP                        ;函数 main 结束

        EXPORT main                 ;导出 main 函数供外部调用
    ;引入三个 C 运行时库函数和 ARM 库
        IMPORT _main
        IMPORT __main
        IMPORT _printf
        IMPORT ||Lib$$Request$$armlib||, WEAK
        END
```

字符串“aaa str1$str1. l1$l1,a1$num1.ccc”中的 3 个变量将在编译时被替换。
程序运行后看到下面结果：
aaa str1:bbb l1:T,a1:0000004Fccc

伪指令

在 ARM 汇编语言源程序中有些特殊助记符，它们没有相对应的操作码或者机器码，通常称为伪指令，它们所完成的操作称为伪操作。
伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，由汇编程序在源程序的汇编期间进行处理，仅在汇编过程中起作用。
在 ARM 的汇编程序中，有如下几种伪指令：
符号定义伪指令
数据定义伪指令
汇编控制伪指令
信息报告伪指令
宏指令以及其他伪指令

符号定义伪指令

作用：用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等。
符号定义有如下几种伪指令：
用于定义局部变量的 LCLA、LCLL 和 LCLS。
用于定义全局变量的 GBLA、GBLL 和 GBLS。
用于对变量赋值的 SETA、SETL 和 SETS。
为通用寄存器列表定义名称的 RLIST。

符号定义伪指令 1-1

（1）LCLA、LCLL 和 LCLS
格式：
LCLA/LCLL/LCLS 局部变量名
说明：LCLA、LCLL 和 LCLS 伪指令用于定义一个汇编程序中的局部变量并初始化。
其中：
LCLA 定义一个局部的数字变量，初始化为 0。
LCLL 定义一个局部的逻辑变量，初始化为 F。
LCLS 定义一个局部的字符串变量，初始化为空串。
这 3 条伪指令用于声明局部变量，在其局部作用范围内变量名必须惟一，例如在宏内。

符号定义伪指令 1-2

```
例如：
MACRO TEST
LCLA    num1  ;定义一个局部的数字变量，变量名为
              ; num1
LCLL    l2    ;定义一个局部的逻辑变量，变量名为l2
LCLS    str3  ;定义一个局部的字符串变量，变量名
              ; 为str3
num1    SETA  0xabcd      ;将该变量赋值为0xabcd
l2      SETL  {FALSE}     ;将该变量赋值为真
str3    SETS  "Hello!"    ;将该变量赋值为“Hello!”
...
MEND
```

符号定义伪指令 2-1

2）GBLA、GBLL 和 GBLS
格式：
GBLA/GBLL/GBLS 变量名
说明：GBLA、GBLL 和 GBLS 伪操作定义一个汇编程序中的全局变量并初始化。
其中：
GBLA 定义一个全局数字变量，并初始化为 0。
GBLL 定义一个全局逻辑变量，并初始化为 F。
GBLS 定义一个全局字符串变量，并初始化为空串。
这 3 条伪指令用于定义全局变量，因此在整个程序范围内变量名必须惟一。

符号定义伪指令 2-2

例如：

```
GBLA num1 ;定义一个全局的数字变
; 量，变量名为 num1
num1 SETA 0xabcd;将该变量赋值为 0xabcd
GBLL l2 ;定义一个全局的逻辑变
; 量，变量名为 l2
l2 SETL {FALSE} ;将该变量赋值为假
GBLS str3 ;定义一个全局的字符串变
; 量，变量名为 str3
str3 SETS "Hello!" ;将该变量赋值为“Hello!”
```

符号定义伪指令 3-1

（3）SETA、SETL 和 SETS

格式：

变量名 SETA/SETL/SETS 表达式

说明：

SETA：给一个数字变量赋值。

SETL：给一个逻辑变量赋值。

SETS：给一个字符串变量赋值。

格式中的变量名必须为已经定义过的全局或局部变量，表达式为将要赋给变量的值。

符号定义伪指令 3-2

例如：

```
LCLA num1 ;定义一个局部的数字
; 变量，变量名为 num1
num1 SETA 0x1234 ;将该变量赋值
; 为 0x1234
LCLS str3 ;定义一个局部的字符串变
; 量，变量名为 str3
str3 SETS “Hello!” ;将该变量赋值为
; “Hello!”
```

符号定义伪指令 4

4）RLIST

格式：

名称 RLIST {寄存器列表}

说明：RLIST 可用于对一个通用寄存器列表定义名称，该名称可在 ARM 指令 LDM/ STM 中使用。在 LDM/STM 指令中，列表中的寄存器为根据寄存器的编号由低到高访问次序，与列表中的寄存器排列次序无关。

例如：

```
pblock RLIST {R0-R3，R7,R5，R9}
;将寄存器列表名称定义为 pblock，可在 ARM 指令
```

;LDM/STM 中通过该名称访问寄存器列表

数据定义伪指令

作用：为数据分配存储单元，同时初始化。

有如下几种：

DCB 字节分配

DCW/DCWU 半字（2 字节）分配

DCD/DCDU 字（4 字节）分配

DCQ/DCQU 8 个字节分配

DCFS/DCFSU 单精度浮点数分配

DCFD/DCFDU 双精度浮点数分配

SPACE 分配一块连续的存储单元

FIELD 定义一个结构化的内存表的数据域

MAP 定义一个结构化的内存表首地址

数据定义伪指令 1

（1）DCB

格式：

标号 DCB 表达式

说明：分配一块字节单元并用伪指令中指定的表达式进行初始化。

表达式可以为使用双引号的字符串或 0~255 的数字。

DCB 可用“=”代替。

例如：

```
Array1 DCB 1,2,3,4,5 ;数组
str1 DCB "Your are welcome! "
;构造字符串并分配空间
```

数据定义伪指令 2

（2）DCW/DCWU

格式：

标号 DCW/DCWU 表达式

说明：DCW 分配一段半字存储单元并用表达式值初始化，它定义的存储空间是半字对齐的。

DCWU 功能与 DCW 类似，只是分配的字存储单元不严格半字对齐。

例如：

```
Arrayw1 DCW 0xa,-0xb,0xc,-0xd
;构造固定数组并分配半字存储单元
```

数据定义伪指令 3

（3）DCD/DCDU

格式：

标号 DCD/DCDU 表达式

说明：DCD 伪指令用于分配一块字存储单元并用伪指令中指定的表达式初始化，它定义的存储空间是字对齐的。

DCD 也可用“&”代替。

DCDU 功能与 DCD 类似，只是分配的存储单元不严格字对齐。
例如：
Arrayd1 DCD 1334，234，345435
;构造固定数组并分配字为单元的存储单元
Label DCD str1;该字单元存放 str1 的地址

数据定义伪指令 4
（4）DCQ/DCQU
格式：
标号 DCQ/DCQU 表达式
说明：DCQ 用于分配一块以 8 个字节为单位的存储区域并用伪指令中指定的表达式初始化，它定义的存储空间是字对齐的。
DCQU 功能与 DCQ 类似，只是分配的存储单元不严格字对齐。
例如：
Arrayd1 DCQ 234234，98765541
;构造固定数组并分配字为单元的存储空间。
；注意：DCQ 不能给字符串分配空间

数据定义伪指令 5
（5）DCFD/DCFDU
格式：
标号 DCFD/DCFDU 表达式
说明：DCFD 用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化，它定义的存储空间是字对齐的。
每个双精度的浮点数占据两个字单元。
DCFDU 功能与 DCFD 类似，只是分配的存储单元不严格字对齐。
例如：
Arrayf1 DCFD 6E2
Arrayf2 DCFD 1.23,1.45

数据定义伪指令 6
（6）DCFS/DCFSU
格式：
标号 DCFS/DCFSU 表达式
说明：DCFS 用于为单精度的浮点数分配一片连续的字存储单元并用表达式初始化，它定义的存储空间是字对齐的。
每个单精度浮点数使用一个字单元。
DCFSU 功能与 DCFS 类似，只是分配的存储单元不严格字对齐。
例如：
Arrayf1 DCFS 6E2 ，-9E-2，-.3
Arrayf2 DCFSU 1.23,6.8E9

数据定义伪指令 7
7）SPACE

格式：
标号 SPACE 表达式
说明：SPACE 用于分配一片连续的存储区域并初始化为 0，表达式为要分配的字节数。
SPACE 也可用“%”代替。
例如：
freespace SPACE 1000
;分配 1000 字节的存储空间

数据定义伪指令 8

（8）MAP
格式：
MAP 表达式 [，基址寄存器]
说明：MAP 定义一个结构化的内存表的首地址。此时，内存表的位置计数器{VAR}（汇编器的内置变量）设置成该地址值。
“ ^ ”可以用来代替 MAP。
表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。
MAP 可以与 FIELD 伪操作配合使用来定义结构化的内存表。
例如：
MAP 0x130,R2 ;内存表首地址为 0x130＋R2

数据定义伪指令 9

（9）FIELD
格式： 标号 FIELD 字节数
说明：FIELD 用于定义一个结构化内存表中的数据域。
“#”可用来代替 FIELD。
FIELD 常与 MAP 配合使用来定义结构化的内存表:FIELD 伪指令定义内存表中的各个数据域，MAP 则定义内存表的首地址，并为每个数据域指定一个标号以供其他的指令引用。
需要注意的是 MAP 和 FIELD 伪指令仅用于定义数据结构，并不分配存储单元。
例如：
MAP 0xF10000
;定义结构化内存表首地址为 0xF10000
count FIELD 4
;定义 count 的长度为 4 字节，位置为 0xF1000+0
x FIELD 4
;定义 x 的长度为 4 字节，位置为 0xF1004
y FIELD 4
;定义 y 的长度为 4 字节，位置为 0xF1008

汇编控制伪指令

作用：指引汇编程序的执行流程。
常用的伪操作包括：
（1）MACRO 和 MEND：宏定义的开始与结束。

（2）IF、ELSE 和 ENDIF：根据逻辑表达式的成立与否决定是否在编译时加入某个指令序列。
（3）WHILE 和 WEND：根据逻辑表达式的成立与否决定是否循环执行这个代码段。
（4）MEXIT：从宏中退出。

MACRO 和 MEND

格式

MACRO

[$标号] 宏名 [$参数 1，$参数 2，……]

指令序列

MEND

其中，$标号在宏指令被展开时，标号可被替换成相应的符号（在一个符号前使用$，表示程序在汇编时将使用相应的值来替代$后的符号）， $参数 1 为宏指令的参数，当宏指令被展开时将被替换成相应的值，类似于函数中的形式参数。

宏指令可以重复使用，与子程序有些类似，子程序可以节省存储空间，提供模块化的程序设计。但是使用子程序结构时需要保存/恢复现场，从而增加了系统的开销。

使用说明：在子程序比较短而需要传递的参数比较多的情况下，可使用宏汇编技术。

宏定义伪指令

例子

```
MACRO ;宏定义开始
$label jump $a1,$a2 ;宏的名称为 jump，有 2 个参数 a1 和 a2
$label.loop1 ; $label.loop1 为宏体的内部标号
…
BGE $label.loop1
$label.loop2
BL $a1 ;参数$a1 为一个子程序的名称
BGT $label.loop2
…
ADR $a2
…
MEND ;宏定义结束
```

宏定义伪指令

在程序中调用该宏

```
exam jump sub,det ;调用宏 jump,宏的标号为 exam, 参数 1 为 sub,参数 2 为 det
```

程序被汇编后，宏的展开结果：

```
…
examloop1
…
BGE examloop1
examloop2
BL sub
BGT examloop2
ADR det
```

IF、ELSE 和 ENDIF
格式：
IF 逻辑表达式
代码段 1
ELSE
代码段 2
ENDIF
说明：能根据逻辑表达式的成立与否决定是否在编译时加入某个指令序列。 IF、ELSE 和 ENDIF 分别可以用“[”，“|”，“]”代替。如果 IF 后面的逻辑表达式为真，则编译代码段 1，否则编译代码段 2。ELSE 及代码段 2 也可以没有，这时，当 IF 后面的逻辑表达式为真时，则代码段 1，否则继续编译后面的指令。

WHILE 和 WEND
格式：
WHILE 逻辑表达式
代码段
WEND
说明： WHILE 和 WEND 伪指令能根据逻辑表达式的成立与否决定是否循环执行这个代码段。当 WHILE 后面的逻辑表达式为真时，则执行代码段，该代码段执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。
例如：

```
GBLA num ;声明全局的数字变量 num
num SETA 9 ;由 num 控制循环次数
…
WHILE num>0
sub r0,r0,1
add r1,r1,1
WEND
```

其他伪指令
在汇编程序中经常会使用一些其他的伪指令，包括以下 18 条：
ASSERT AREA
ALIGN CODE16/CODE32
ENTRY END
EQU IMPORT
EXPORT/GLOBAL EXTERN
INCBIN GET/INCLUDE
RN ROUT
ADR ADRL
LDR NOP

其他伪指令 1
（1）ASSERT

格式：
ASSERT 逻辑表达式
说明：ASSERT 用来表示程序的编译必须满足一定的条件，如果逻辑表达式不满足，则编译器会报错，并终止汇编。
例如：
ASSERT ver>7 ;保证 ver>7

其他伪指令 2
2）AREA
格式： AREA 段名 属性，……
说明：AREA 用于定义一个代码段、数据段或者特定属性的段。如果段名以数字开头，那么该段名需用“|”字符括起来,如|7wolf|,用 C 的编译器产生的代码一般也用“|”括 起来。属性部分表示该代码段/数据段的相关属性，多个属性可以用“，”分隔。
常见属性如下：
① DATA：定义数据段，默认属性是 READWRITE。
② CODE：定义代码段，默认属性是 READONLY 。
③ READONLY：表示本段为只读。
④ READWRITE：表示本段可读写。
⑤ ALIGN=表达式，表示段的对齐方式为 2 的表达式次方，例如：表达式=3，则对齐方式为 8 字节对齐。表达式的取值范围为 0~31。
⑥ COMMON 属性：定义一个通用段，这个段不包含用户代码和数据。

其他伪指令 3
（3）ALIGN
格式：
ALIGN [表达式[，偏移量]]
说明：ALIGN 伪操作可以通过填充字节使当前的位置满足一定的对齐方式。
表达式的值为 2 的幂，如 1、2、4、8、16 等，用于指定对齐方式。
如果伪操作中没有指定表达式，则编译器会将当前位置对齐到下一个字的位置。偏移量也是个数字表达式，如果存在偏移量，则当前位置自动对齐到 2 的表达式值次方＋偏移量。
例如：
AREA ||.data||,DATA,READWRITE,ALIGN=2

其他伪指令 4
（4）CODE16/CODE32
格式： CODE16/CODE32
说明：CODE16 伪操作指示编译器后面的代码为 16 位的 Thumb 指令。CODE32 伪操作指示编译器后面的代码为 32 位的 ARM 指令。
如果在汇编源代码中同时包含 Thumb 和 ARM 指令时,可以用“CODE32”通知编译器后的指令序列为 32 位的 ARM 指令，用“CODE16”伪指令通知编译器后的指令序列为 16 位的 Thumb 指令。
CODE16/CODE32 不能对处理器进行状态的切换。
例如：
CODE32 ; 32 位的 ARM 指令

```
AREA ||.text||,CODE，READONLY
…
LDR R0,＝0x8500;
BX R0 ;程序跳转，并将处理器切换到 Thumb 状态
…
CODE16 ;16 位的 Thumb 指令
ADD R3,R3,1
END ;源文件结束
```

其他伪指令 5-1

（5）ENTRY

格式：

ENTRY

说明：ENTRY 用于指定汇编程序的入口。

在一个完整的汇编程序中至少要有一个 ENTRY，程序中也可以有多个，此时，程序的真正入口点可在链接时指定，但在一个源文件里最多只能有一个 ENTRY 或者没有 ENTRY。

其他伪指令 5-2

下面的代码使用了ENTRY:

```
    AREA subrout, CODE, READONLY ; name this block of code mar k
    ENTRY                        ; first instruction to execute
start
    MOV   r0, #10                ; Set up parameters
    MOV   r1, #3
    BL    doadd                  ; Call subroutine
stop
    MOV   r0, #0x18              ; angel_SWIreason_ReportException
    LDR   r1, =0x20026           ; ADP_Stopped_ApplicationExit
    SWI   0x123456               ; ARM semihosting SWI
doadd
    ADD   r0, r0, r1             ; Subroutine code
    MOV   pc, lr                 ; Return from subroutine
    END                          ; Mark end of file
```

其他伪指令 6

（6）END

格式：

END

说明：END 告诉编译器已经到了源程序的结尾。

例如：

```
AREA constdata，DATA，READONLY
…
END ;结尾
```

其他伪指令 7

（7）EQU

格式： 名称 EQU 表达式 [，类型]

说明：EQU 用于将程序中的数字常量、标号、基于寄存器的值赋予一个等效的名称，这一点类似于 C 语言中的＃define.

可用“*”代替 EQU。

如果表达式为 32 位的常量，我们可以指定表达式的数据类型，类型域可以有以下 3 种：

CODE16/CODE32/DATA

例如：

```
num1 EQU 1234 ;定义 num1 为 1234
addr5 EQU str1+0x50
d1 EQU 0x2400，CODE32 ;定义 d1 的为 0x2400，且该处为 32 位的 ARM 指令
```

其他伪指令 8

（8）EXPORT/GLOBAL

格式：

EXPORT/GLOBAL 标号 [,WEAK]

说明：EXPORT 在程序中声明一个全局标号，其他文件中的代码可以被该标号引用。用户也可以用 GLOBAL 代替 EXPORT。

[,WEAK]可选项声明其他文件有同名的标号，则该同名标号优先于该标号被引用。

例如：

```
AREA ||.text||,CODE,READONLY
main PROC
…
ENDP
EXPORT main ;声明一个可全局引用的函数 main
…
END
```

其他伪指令 9

（9）IMPORT

格式：

IMPORT 标号 [,WEAK]

说明：告诉编译器，这个标号要在当前源文件中使用，但标号是在其他的源文件中定义的。

[,WEAK]：如果所有的源文件都没有找到这个标号的定义，编译器也不会提示错误信息，同时编译器也不会到当前没有被 INCLUDE 进来的库中去查找该符号。

例如：

```
AREA mycode,CODE,READONLY
IMPORT _ printf
;通知编译器当前文件要引用函数_ printf
…
END
```

IMPORT

使用说明
使用 IMPORT 为操作声明一个符号是在其他源文件中定义的。如果链接器在链接处理时不能解析该符号，而且 IMPORT 为操作中没有指定[WEAK]选项，则链接器将会报告错误。如果链接器在链接处理时不能解析该符号，而 IMPORT 伪操作中指定了[WEAK]选项，则链接器不会报告错误，而是进行下面的操作：
如果该符号被 B 或 BL 指令引用，则该符号被设置成下一条指令的地址，该 B 或者 BL 指令相当于一条 NOP 指令。例如“B sign ”，“sign”不能被解析，则该指令被忽略为 NOP 指令，继续执行下面的指令，也就是将 sign 理解为下一条指令的地址。
其他情况下该符号被设置为 0。

其他伪指令 10
10）EXTERN
格式：
EXTERN 标号 [,WEAK]
说明：告诉编译器，标号要在当前源文件中引用，但是该标号是在其他的源文件中定义的。与 IMPORT 不同的是，如果当前源文件实际上没有引用该标号，该标号就不会被加入到当前文件的符号表中。
[,WEAK]：即使所有的源文件都没有找到这个标号的定义，编译器也不给出错误信息。
例如：
```
AREA ||.text||,CODE,READONLY
…
EXTERN _ printf,WEAK ;告诉编译器当前文件要引用标号，如果找不到，则不提示错误
…
END
```

其他伪指令 11
（11）GET/INCLUDE
格式： GET 文件名
说明：GET 将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置展开进行汇编处理。
INCLUDE 和 GET 的作用是等效的。
使用方法：在某源文件中定义一些宏指令，用 MAP 和 FIELD 定义结构化的数据类型，用 EQU 定义常量的符号名称，然后用 GET/INCLUDE 将这个源文件包含到其他的源文件中。
使用方法与 C 语言中的“#include”相似。
GET/INCLUDE 只能用于包含源文件，包含其他文件则需要使用 INCBIN 伪指令。
例如：
```
AREA mycode,DATA,READONLY
GET E：\code\prog1.s ;通知编译器在当前源文件包含源文件 E：\code\ prog1.s
GET prog2.s ;通知编译器当前源文件包含可搜索目录下的 prog2.s
…
END
```

其他伪指令 12
（12）INCBIN

格式：
INCBIN 文件名
说明：INCBIN 将一个数据文件或者目标文件包含到当前的源文件中，编译时被包含的文件不作任何变动地存放在当前文件中，编译器从后面开始继续处理。
例如：

```
AREA constdata,DATA,READONLY
INCBIN data1.dat ;源文件包含文件 data1.dat
INCBIN E：\DATA\data2.bin
;源文件包含文件 E：\DATA\data2.bin
…
END
```

其他伪指令 13
（13）RN
格式：
名称 RN 表达式
说明：RN 用于给一个寄存器定义一个别名，以便程序员记忆该寄存器的功能。
名称为给寄存器定义的别名，表达式为寄存器的编码。
例如：

```
count RN R1 ;给 R1 定义一个别名 count
```

其他伪指令 14
（14）ROUT
格式：
[名称] ROUT
说明：ROUT 可以给一个局部变量定义作用范围。
在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作用范围为当前 ROUT 和下一个 ROUT 之间。
例如：

```
routine ROUT ;定义局部标号的有效范围
…
1 routine ； routine 内的局部标号 1
…
BEQ %1 routine ;若条件成立，则跳转到 routine 范围内的局部标号 1
…
Otherroutine ROUT ； 定义新的局部标号的有效范围
```

其他伪指令 14
（15）LTORG
说明：LTORG 用于声明一个数据缓冲池（literal pool）的开始。通常放在无条件跳转指令之后，或者子程序返回指令之后，以免处理器错误地将数据缓冲池中地数据作为指令来执行。
例如：

```
Func1
……
```

```
MOV PC, LR
LTORG
DATA SPACE 26;从 data 标号开始预留 256 字节地内存单元
END
```

其他伪指令 15

（16）ADR 小范围地址读取

格式：

ADR{<cond>} <Rd>,< expr>;

说明：将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值（expr 地址表达式）读取到目标寄存器 Rd 中。

当地址值是非字对齐时，取值范围在-255~255 字节之间；

当地址值是字对齐时，取值范围在-1 020～1 020 字节之间。

在汇编编译源程序时,ADR 伪指令被编译器替换成一条合适的指令。

通常,编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能。若不能用一条指令实现,则产生错误,编译失败。

对于基于 PC 相对偏移的地址值时,给定范围是相对当前指令地址后两个字处(因为 ARM7TDMI 为三级流水线)。

可以用 ADR 加载地址实现查表。

例如：

```
LOOP MOV R1，#0xF0
ADR R2，LOOP ；将 LOOP 的地址放入 R2，因为 PC 值为当前指令地址值加 8 字节，所以本 ADR 伪指令将被编译器换成“SUB R2,PC，0XC”
```

其他伪指令 16

（17）ADRL 中等范围地址读取

格式：

ADRL{<cond>} <Rd>,< expr>;

说明：类似于 ADR， 但比 ADR 读取更大范围的地址。

当地址值是非字对齐时，取值范围在-64KB～64 KB 之间；

地址值是字对齐时，取值范围在-256KB~256 KB 之间。

在汇编编译源程序时，ADRL 伪指令被编译器替换成两条合适的指令。

若不能用两条指令实现 ADRL 伪指令功能，则产生错误，编译失败。

可以用 ADRL 加载地址，实现程序跳转。

例如：

```
ADRL R0，DATA_BUF
ADRL R1，DATA_BUF+80
DATA_BUF
SPACE 100 ；定义 100 字节缓冲区
```

其他伪指令 17

（18）LDR 大范围地址读取

格式：

LDR{<cond>} <Rd>,< =expr/label-expr >;

说明：加载 32 位的立即数或一个地址值到目标寄存器 Rd。
在汇编编译源程序时，LDR 伪指令被编译器替换成一条合适的指令。
若加载的常数未超出 MOV 或 MVN 的范围，则使用 MOV 或 MVN 指令代替该 LDR 伪指令；否则汇编器将常量放入文字池，并使用一条程序相对偏移的 LDR 指令从文字池读出常量。
LDR 用于加载芯片外围功能部件的寄存器地址(32 位立即数)，以实现各种控制操作。从 PC 到文字池的偏移量必须小于 4 KB。
与 ARM 指令的 LDR 相比，伪指令的 LDR 的参数有“＝”符号。
例如：

```
LDR R0，=0x12345678 ； 加载 32 位立即数 0x12345678
LDR R0，=DATA_BUF+60 ； 加载 DATA_BUF 地址+60
…
LTORG ； 声明文字池
```

其他伪指令 18
（19）NOP 空操作
格式：

```
NOP ；
```

说明：不产生任何有意义的操作，只是占用一个机器时间。
NOP 伪指令在汇编时将会被替代成 ARM 中的空操作，比如可能为“MOV R0,R0”指令等。

简单的 ARM 汇编程序设计（一）
查表和散转程序设计
当涉及到数据串或者跳转表格时，常常需要通过地址对他们进行访问，通常有两种方法装载地址：
通过 ADR 和 ADRL 伪指令直接装载地址；
通过伪指令 LDR Rd，＝label 从数据池中装载地址。
下面以程序 jump.s 为例，介绍通过 ADR 伪指令装载地址地散转程序地设计。

查表和散转程序设计
主程序中设置了 3 个参数，arithfunc 根据 3 个参数返回一个 R0 值。
当 R0＝0 时，R0:=R1+R2;
当 R0＝1 时，R0:=R1-R2;
简单的 ARM 汇编程序设计（一） 查表和散转程序设计

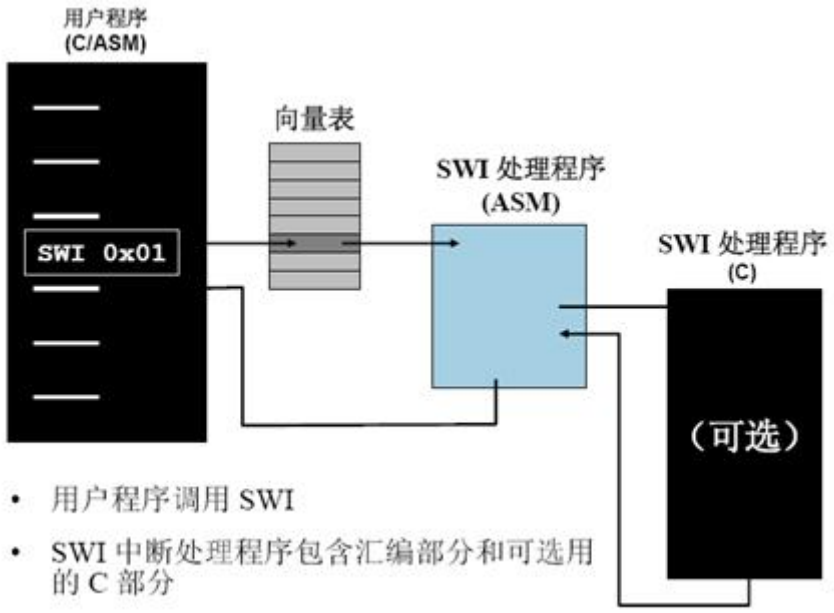
```
AREA Jump, CODE, READONLY ; name this block of code
CODE32 ; Following code is ARM code
num EQU 2 ; 跳转表的入口数目
ENTRY ; 程序入口
start
MOV r0, #0 ; 设置 3 个参数
MOV r1, #3
MOV r2, #2
BL arithfunc ; 调用子程序
Stop ; 执行中止
MOV r0, #0x18 ;软中断参数设置
```

```
LDR r1, =0x20026 ;软中断参数设置
SWI 0x123456 ; 将CPU的控制权交给调试器ARM ; semihosting SWI
```

查表和散转程序设计——续

```
arithfunc ;
CMP r0, #num ; 比较参数
MOVHS pc, lr ; 若超出范围则程序返回
ADR r3, JumpTable ; 装载跳转表格标号地址
LDR pc, [r3,r0,LSL#2] ; 跳转到相应子程序入口地址处
JumpTable
DCD DoAdd
DCD DoSub
DoAdd
ADD r0, r1, r2 ; =0时的操作
MOV pc, lr ; 返回
DoSub
SUB r0, r1, r2 ; =1时的操作
MOV pc,lr ; 返回
END ; 程序结尾
```

软中断指令 SWI



Semihosting 在ADS的C语言函数库中，某些ANSIC的功能是由主机的调试环境来提供的，这套机制有一个专门术语叫Semihosting。

Semihosting通过一组软件中断(SWI)指令来实现。

当一个Semihosting软中断被执行时，调试系统先识别这个SWI请求，然后挂起正在运行的程序，调用Semihosting的服务，完成后再恢复原来的程序执行。

因此，主机执行的任务对于程序来说是透明的。

SWI 传递的功能号
(例如： semi-hosting, 使用 0x123456 (ARM) or 0xAB (Thumb)

续

在此例中，表格 jumptable 中存放地是子程序地入口地址，我们把这种表格称为跳转表格。注意指令 LDR PC，[R3,R0,LSL #2],执行地操作为 PC＝R3＋R0×4，因为表格中存放地地址为 4 字节地址，所以要将 R0 乘以 4 得出偏移量，再加上表格首地址，得出子程序地入口地址赋值给 PC。

字符串拷贝程序设计

下面的例子为用 ARM 指令编写的串拷贝的例子。
两个数据串都放在数据段中，且用 DCB 伪指令定义,DCB 为定义 1 字节或多字节内存空间，双引号中的字符串在内存中是顺序存放的，因此取数/存数时需要使用 LDRB 和 STRB 指令；若数据串是用 DCD 存放的，则应使用 LDR 和 STR 指令。
另外，例子中采用的 LDRB/STRB 指令是后索引寻址方式，即寻址完成后更新地址。

字符串拷贝程序设计（用 LDR 和 STR 实现）

```
AREA StrCopy, CODE, READONLY
ENTRY ; 程序入口
start
LDR r1, =srcstr ; 初始串的指针
LDR r0, =dststr ; 结果串的指针
BL strcopy ; 调用子程序执行复制
stop
MOV r0, #0x18 ; 执行中止
LDR r1, =0x20026 ;
SWI 0x123456 ;
```

字符串拷贝程序设计（用 LDR 和 STR 实现）——续

```
strcopy
LDRB r2, [r1],#1 ; 加载并且更新源串指针
STRB r2, [r0],#1 ; 存储且更新目的串指针;
CMP r2, #0 ; 是否为 0
BNE strcopy ;
MOV pc,lr ;
AREA Strings, DATA, READWRITE
srcstr DCB "First string - source",0
dststr DCB "Second string - destination",0
END
```

字符串拷贝程序设计

数据串拷贝时，若使用 LDM 和 STM 则可增加程序的效率。考虑到 ARM 的寄存器，一次采用 8 个寄存器进行传输比较合适，
通过指令： MOVS r3,r2, LSR #3 来计算需要几轮 8 位数据传送，剩余的数据个数可以通过

指令 ANDS r2, r2, #7 获得，再对其进行按字传输即可。

字符串拷贝程序设计（用 LDM 和 STM 实现）

```
AREA Block, CODE, READONLY ; 命名
num EQU 20 ; 设置被拷贝的字数
ENTRY ; 程序入口
start
LDR r0, =src ; r0 = 源串指针
LDR r1, =dst ; r1 = 目的串指针
MOV r2, #num ; r2 = 拷贝字数
MOV sp, #0x400 ; 设置堆栈指针 (r13)
blockcopy
MOVS r3,r2, LSR #3 ; 字数/8
BEQ copywords ; 少于 8 个字
STMFD sp!, {r4-r11} ; save some working registers
octcopy
LDMIA r0!, {r4-r11} ; 从源串加载 8 个字
STMIA r1!, {r4-r11} ; 放入目的串
SUBS r3, r3, #1 ; 控制变量减少
BNE octcopy ; ... 继续
```

字符串拷贝程序设计（用 LDM 和 STM 实现）——续

```
LDMFD sp!, {r4-r11} ;
copywords
ANDS r2, r2, #7 ; 奇数字被拷贝
BEQ stop ; No words left to copy ?
wordcopy
LDR r3, [r0], #4 ; 从源串加载一个字且指针自增
STR r3, [r1], #4 ; 存储到目的串
SUBS r2, r2, #1 ; 字控制变量减少
BNE wordcopy ; 继续
stop
MOV r0, #0x18 ; 执行中止
LDR r1, =0x20026 ;
SWI 0x123456 ;
AREA BlockData, DATA, READWRITE
src DCD 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst DCD 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
END
```