华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《从实践中学ARM体系结构与接口技术》

作者：华清远见

专业始于专注 卓识源于远见

第4章 ARM汇编语言程序设计

本章简介

在第2、3章中阐述的体系结构及指令集理论的基础上，本章主要介绍利用ARM汇编语言进行编程。ARM 编译器可以支持汇编语言、C/C++、汇编语言与 C/C++的混合编程等，本章将介绍汇编、C 相关的编程方法。本章主要内容：

- ARM 汇编器支持的伪操作
- ARM 汇编器支持的伪指令
- ARM 汇编器的使用
- 汇编语言与 C 的混合编程

4.1 ARM 汇编器支持的伪操作

4.1.1 伪操作概述

在 ARM 汇编语言程序中，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪操作标识符（directive），它们所完成的操作称为伪操作。伪操作在源程序中的作用是为了完成汇编程序做各种准备工作，这些伪操作仅在汇编过程中起作用，一旦汇编结束，伪操作的使命就完成了。

在 ARM 的汇编程序中，伪操作主要有符号定义伪操作、数据定义伪操作、汇编控制伪操作及其杂项伪操作等。

4.1.2 符号定义伪操作

符号定义伪操作用于定义 ARM 汇编程序中的变量、对变量赋值及定义寄存器的别名等操作。常见的符号定义伪操作有如下几种：

（1）用于定义全局变量的 GBLA、GBLL 和 GBLS。

（2）用于定义局部变量的 LCLA、LCLL 和 LCLS。

（3）用于对变量赋值的 SETA、SETL 和 SETS。

（4）为通用寄存器列表定义名称的 RLIST。

1．全局变量定义伪操作 GBLA、GBLL 和 GBLS

1）语法格式

GBLA、GBLL 和 GBLS 伪操作用于定义一个 ARM 程序中的全局变量并将其初始化。其中，

（1）GBLA 伪操作用于定义一个全局的数字变量并初始化为 0。

（2）GBLL 伪操作用于定义一个全局的逻辑变量并初始化为 F（假）。

（3）GBLS 伪操作用于定义一个全局的字符串变量并初始化为空。

由于以上 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唯一。

语法格式如下。

```
<gblx>   <variable>
```

参数说明如下。

- <gblx>：取值为 GBLA、GBLL、GBLS 三者之一。
- <variable>：定义的全局变量名，在其作用范围内必须唯一。全局变量的作用范围为包含该变量的源程序。

2）使用说明

如果用这些伪操作重新声明已经声明过的变量，变量的值将被初始化成后一次声明语句中的值。

3）示例

（1）使用伪操作声明全局变量。

```
GBLA       Test1                ;定义一个全局的数字变量，变量名为Test1
Test1  SETA    0xaa             ;将该变量赋值为0xaa
GBLL       Test2                ;定义一个全局的逻辑变量，变量名为Test2
Test2  SETL    {TRUE}           ;将该变量赋值为真
GBLS       Test3                ;定义一个全局的字符串变量，变量名为Test3
Test3  SETS     "Testing"       ;将该变量赋值为“Testing”
```

（2）声明变量 objectsize 并设置其值为 0xff，为“SPACE”操作做准备。

```
GBLA         objectsize
```

```
Objectsize      SETA     oxff

      SPACE      objectsize
```

（3）下面的例子显示如何使用汇编命令设置变量的值，具体做法是使用“–pd”选项。

```
    Armasm  -pd  "objectsize  SETA  oxff"  -o  objectfile  sourcefile
```

2．局部变量定义伪操作 LCLA、LCLL 和 LCLS

1）语法格式

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量并将其初始化。其中，

（1）LCLA 伪操作用于定义一个局部的数字变量并初始化为 0。

（2）LCLL 伪操作用于定义一个局部的逻辑变量并初始化为 F（假）。

（3）LCLS 伪操作用于定义一个局部的字符串变量并初始化为空。

以上 3 条伪操作用于声明局部变量，在其作用范围内变量名必须唯一。

语法格式如下。

```
<lclx>   <variable>
```

参数说明如下。

- <lclx>：取值为 LCLA、LCLL、LCLS 三者之一。
- <variable>：所定义的局部变量名，在其作用范围内必须唯一。局部变量作用范围为包含该局部变量的宏。

2）使用说明

如果用这些伪操作重新声明已经声明过的变量，则变量的值将被初始化成后一次声明语句中的值。

3）示例

（1）使用伪操作声明局部变量。

```
    LCLA       Test4               ;声明一个局部的数字变量，变量名为Test4
    Test3  SETA     0xaa           ;将该变量赋值为0xaa
    LCLL       Test5               ;声明一个局部的逻辑变量，变量名为Test5
    Test4  SETL     {TRUE}         ;将该变量赋值为真
    LCLS       Test6               ;定义一个局部的字符串变量，变量名为Test6
    Test6  SETS   "Testing"        ;将该变量赋值为“Testing”
```

（2）下面的例子定义一个宏，显示了局部变量的作用范围。

```
      MACRO                     ;声明一个宏
$label    message  $a           ;宏原型
      LCLS    err               ;声明局部字符串变量
$label
      INFO       0,"err":CC::STR:$a
      MEND                      ;宏结束，局部变量不再起作用
```

3．变量赋值伪操作 SETA、SETL 和 SETS

1）语法格式

伪指令 SETA、SETL 和 SETS 用于给一个已经定义的全局变量或局部变量赋值。其中：

（1）SETA 伪操作用于给一个数学变量赋值。

（2）SETL 伪操作用于给一个逻辑变量赋值。

（3）SETS 伪操作用于给一个字符串变量赋值。

语法格式如下：

```
Variable  <setx>  expr
```

参数说明如下。

- Variable：变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

- <setx>：取值为 SETA、SETL、SETS 三者之一。
- expr：数学、逻辑或字符串表达式，也就是将要赋予变量的值。

2）使用说明

在向变量赋值前必须先声明变量，也可以在汇编指令中预定义变量，如：

```
"Armasm --pd "objectsize SETA oxff" --o objectfile sourcefile"
```

3）示例

（1）为预先定义的变量赋值。

```
LCLA      Test3          ;声明一个局部的数字变量，变量名为 Test3
Test3  SETA  0xaa        ;将该变量赋值为 0xaa
LCLL      Test4          ;声明一个局部的逻辑变量，变量名为 Test4
Test4  SETL  {TRUE}      ;将该变量赋值为真
LCLS      Test6          ;定义一个局部的字符串变量，变量名为 Test6
Test6  SETS   "Testing"  ;将该变量赋值为“Testing”
```

（2）使用变量赋值伪操作，定义一些程序相关内容。

```
GBLA              versionNumber
VersionNumber    SETA    21

GBLL              Debug
Debug SETL  {TRUE}

GBLS  versionString
VersionString     SETS   "version 1.0"
```

4．通用寄存器列表定义伪操作 RLIST

1）语法格式

RLIST 伪操作可用于对一个通用寄存器列表定义名称，使用该伪操作定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器访问顺序根据寄存器的编号由低到高，与列表中的寄存器排列顺序无关。

语法格式如下。

```
Name  RLIST  {list-of-registers}
```

参数说明如下。

- Name：寄存器列表的名称。
- list-of-registers：通用寄存器列表。列表中的寄存器用“,”隔开，如果是编号连续的通用寄存器可以用“–”指定寄存器范围。具体用法参见程序示例。

2）使用说明

在使用 ARM 汇编编译器编译源文件时，可以使用“–checkreg”选项来指定汇编器进行寄存器检查。如果汇编器检测到寄存器列表中的寄存器编号非升序排列，将给出编译警告。

3）示例

（1）将寄存器列表名称定义为 RegList，可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表。

```
RegList  RLIST  {R0-R5,R8,R10}
```

（2）使用“–”在寄存器列表中指定寄存器范围。

```
Context  RLIST  {R0-R6,R8,R10-R12,R15}
```

4.1.3 数据定义（Data Definition）伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪操作有如下几种：

（1）DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。

（2）DCW（DCWU）用于分配一片连续的半字存储单元并用指定的数据初始化。
（3）DCD（DCDU）用于分配一片连续的字存储单元并用指定的数据初始化。
（4）DCFD（DCFDU）用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
（5）DCFS（DCFSU）用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
（6）DCQ（DCQU）用于分配一片以 8 字节为单位的连续的存储单元并用指定的数据初始化。
（7）SPACE 用于分配一片连续的存储单元。
（8）MAP 用于定义一个结构化的内存表首地址。
（9）FIELD 用于定义一个结构化的内存表的数据域。

1．DCB

1）语法格式

DCB 伪操作用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为数字或字符串。DCB 也可用“=”代替。

语法格式如下：

```
{label}  DCB  expr{,expr}
```

参数说明如下。

- {label}：程序标号。
- expr：可以是−128～255 之间的数字，也可以是字符串。

2）使用说明

在使用 DCB 伪操作时，其后常跟 ALIGN 伪操作以保证内存地址对齐。

3）示例

（1）分配一片连续的字节存储单元并初始化为指定字符串。

```
Str  DCB  "This is a test! "
```

（2）与 C 中的字符串不同，ARM 汇编中的字符串不以 null 结尾。下面的指令以 ARM 汇编形成一个 C 语言风格的字符串。

```
C_string  DCB  "C_string",0
```

2．DCW（或 DCWU）

1）语法格式

DCW（或 DCWU）伪操作用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。

用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

语法格式如下：

```
{label}  DCW  expr{,expr}…
```

参数说明如下。

- {label}：程序标号，可选。
- expr：数字表达式，取值范围为−32 768～65 525。

2）使用说明

DCW 可以在分配的内存单元前加 1 字节以保证内存半字对齐。当程序对内存对齐方式要求不严格时可以用 DCWU 伪操作。

3）示例

（1）分配一片连续的半字存储单元并初始化。

```
DataTest  DCW    1,2,3
```

（2）在指定内存单元初始值时可以使用已定义的变量。

华清远见

```
Data      DCW-255,2*number
          DCWU    number+4
```

3. DCD（或 DCDU）

1）语法格式

DCD（或 DCDU）伪操作用于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。其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。

用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

语法格式如下：

```
{label}  DCD{U}  expr{,expr}
```

参数说明如下。

- {label}：程序标号，可选。
- expr：expr 可以是数字表达式或程序相关表达式（program-relative expression）。

2）使用说明

DCD 可以在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以用 DCDU 伪操作。

3）示例

（1）分配一片连续的字存储单元并初始化。

```
DataTest  DCD  4,5,6
```

（2）用程序标号初始化内存单元。

```
DataTest  DCD  mem06+4
```

（3）在内存单元不能字对齐的情况下，使用 DCDU 伪操作。

```
        AREA  Mydata,  DATA,READWRITE
        DCB  255             ;字节定义使内存单元不能字对齐
Data3   DCDU  1,5,20
```

4. DCFS（或 DCFSU）

1）语法格式

DCFS（或 DCFSU）伪指令用于为单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。每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

语法格式如下：

```
{label}  DCFS{U}  fpliteral{,fpliteral}
```

参数说明如下。

- {label}：程序标号，可选。
- fpliteral：单精度浮点数。

2）使用说明

DCFS 可以在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以用 DCFSU 伪操作。

此伪操作使用的单精度浮点数的范围为：1.175 494 35e－38～3.402 823 47e＋38。

3）示例

（1）分配一片连续的字存储单元并初始化为指定的单精度浮点数。

```
FDataTest  DCFS  2E5,-5E-7
```

（2）分配一片连续的字存储单元并初始化为单精度浮点数，但不严格要求字对齐。

```
DCFSU  1.0,-0.1,3.1e6
```

5. DCFD（或 DCFDU）

1）语法格式

DCFD（或 DCFDU）伪指令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。每个双精度的浮点数占据两个字单元。

用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

语法格式如下：

```
{label} DCFD{U} fpliteral{,fpliteral}
```

参数说明如下。

- {label}：程序标号，可选。
- fpliteral：双精度浮点数。

2）使用说明

DCFD 可以在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以用 DCFDU 伪操作。

当程序中的浮点数要由 ARM 处理器进行操作时，用户选择的浮点处理器结构会自动完成字节顺序的转换。当编译时使用了编译选项–fpunone 时，伪操作 DCFD（DCFDU）不可使用。

此伪操作使用的单精度浮点数的范围为：2.225 073 858 507 201 38e－308～1.797 693 134 862 315 71e＋308。

3）示例

（1）分配一片连续的字存储单元并初始化为指定的双精度浮点数。

```
FDataTest    DCFD   2E115,-5E7
```

（2）分配一片连续的字存储单元并初始化为双精度浮点数，但不严格要求字对齐。

```
DCFDU  1.0,-0.1,3.1e6
```

6. DCQ（或 DCQU）

1）语法格式

DCQ（或 DCQU）伪指令用于分配一片以 8 字节为单位的连续存储区域并用伪指令中指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

语法格式如下：

```
{label} DCQ{U} {-}literal{,{-}literal}
```

参数说明如下。

- {label}：程序标号，可选。
- literal：用于初始化内存的数字必须是可数的数字表达式，其取值范围为 0～$2^{64}-1$。

可以在数字表达式前加负号来表示用负数初始化内存单元，但此时数字表达式的取值范围为-2^{63}～1。

2）使用说明

DCQ 可以在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以用 DCQU 伪操作。

3）示例

（1）分配一片连续的存储单元并初始化为指定的值。

```
DataTest DCQ   100
```

（2）使用标号定义内存单元。

```
ECQU  number+4
```

7. SPACE

华清远见

1）语法格式

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用“%”代替。

语法格式如下：

```
{label}  SPACE  expr
```

参数说明如下。

- {label}：程序标号，可选。
- expr：分配的字节数。

2）使用说明

SPACE 伪操作常和 ALIGN 一起使用，详见 ALIGN 伪操作。

3）示例

（1）分配连续 100 字节的存储单元并初始化为 0。

```
DataSpace SPACE    100
```

（2）在 Mydata 段的开始可以是 255 个初始化为 0 的字节单元。

```
         AREA    Mydata,DATA,READWRITE
data1 SPACE   255
```

8. MAP

1）语法格式

MAP 伪操作用于定义一个结构化的内存表的首地址。MAP 也可用“ ^ ”代替。

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。

语法格式如下：

```
MAP  expr{,base-register}
```

参数说明如下。

- expr：如果基地址寄存器（base-register）没有指定，expr 表达式存储到结构化内存表首地址；如果表达式 expr 是“程序相关的（program-relative）”，则程序标号在使用前必须定义。
- base-register：指定一个寄存器。当指令中包含这一项时，结构化内存表的首地址为 expr 和 base-register 寄存器值的和。

2）使用说明

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

当基地址寄存器（base-register）一旦被指定，下面所有的 FIELD 伪操作全部以基地址为基础增加偏移量。

3）示例

（1）定义结构化内存表首地址的值为 0x100+R0。

```
MAP   0x100,R0
```

（2）不存在基地址寄存器，结构化内存表的首地址直接由表达式定义。

```
MAP    0
```

9. FILED

1）语法格式

华清远见

FIELD 伪操作用于定义一个结构化内存表中的数据域。FILED 也可用“#”代替。

表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪操作常与 MAP 伪操作配合使用来定义结构化的内存表。MAP 伪操作定义内存表的首地址；FIELD 伪操作定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的操作引用。

注意 MAP 和 FIELD 伪操作仅用于定义数据结构，并不实际分配存储单元。

语法格式如下：

```
{label} FIELD  expr
```

参数说明如下。

- {label}：程序标号，可选。当指令中存在这一项时，label 的值为当前内存表的位置计数器{VAR}的值。汇编器处理完这条 FIELD 指令后，内存表计数器的值将加上 expr 的值。
- expr：FIELD 指定的域所占的内存单元字节数。

2）使用说明

MAP 伪操作中的基地址寄存器（base-register）一旦被指定，将被其后的所有 FIELD 伪操作定义的数据域默认使用，直至遇到下一个包含基地址寄存器（base-register）的 MAP 指令。另外，在操作中定义的标号也可以被 Load/Store 指令直接引用。

3）示例

（1）下面的例子定义了一个内存表，其首地址为固定地址 0x100。该结构化内存表包含 3 个域：A 的长度为 16 字节，位置为 0x100；B 的长度为 32 字节，位置为 0x110；S 的长度为 256 字节，位置为 0x130。

```
MAP    0x100                 ;定义结构化内存表首地址的值为 0x100
       A      FIELD    16    ;定义 A 的长度为 16 字节，位置为 0x100
       B      FIELD    32    ;定义 B 的长度为 32 字节，位置为 0x110
       S      FIELD    256   ;定义 S 的长度为 256 字节，位置为 0x130
```

（2）下面的例子显示了一个寄存器相关的首地址定义结构化内存表。

```
    MAP 0,R9                 ;将结构化内存表的首地址设为 R9 的值
    FIELD   4
LAB FIELD   4
    LDR r0,LAB
```

最后的 LDR 指令相当于：

```
LDR R0,[R9,#4]
```

4.1.4 汇编控制伪操作

汇编控制伪操作用于控制汇编程序的执行流程。常用的汇编控制伪操作包括以下几条：

（1）IF、ELSE、ENDIF。

（2）WHILE、WEND。

（3）MACRO、MEND。

（4）MEXIT。

1．IF、ELSE、ENDIF

1）语法格式

IF、ELSE、ENDIF 伪操作能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真时，则执行 IF 后的指令序列；否则执行 ELSE 后的指令序列。其中，ELSE 及其后指令序列可以没有，此时，当 IF 后面的逻辑表达式为真时，则执行指令序列；否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

语法格式如下：

```
IF  logical-expressing
…
{ELSE
…}
ENDIF
```

其中，logical-expression 参数用于决定指令执行流程的逻辑表达式。

2）使用说明

当程序中有一段指令需要在满足一定条件时执行，使用该指令。
该操作还有另一种形式：

```
IF  logical-expression
    Instruction
ELIF  logical-expression2
    Instructions
ELIF  logical-expression3
    Instructions
ENDIF
```

ELIF 形式避免了 IF…ELSE 形式的嵌套，使程序结构更加清晰、易读。

3）示例

```
IF  {CONFIG}=16
    BNE_rt_udiv_1     ;
    LDR R0,=_rt_div0   ;
    BX  R0             ;
    ELSE
    BEQ_rt_div()       ;
ENDIF
```

2．WHILE、WEND

1）语法格式

WHILE、WEND 伪操作能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真时，则执行指令序列。该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。

语法格式如下：

```
WHILE  logical-expression
code
WEND
```

其中，logical-expression 参数用于决定指令执行流程的逻辑表达式。

2）使用说明

WHILE、WEND 指令形式在进入循环之前判断执行条件，如果在第一次进入循环时，逻辑表达式即为“假”，则循环体可以不执行。

3）示例

下面的例子用 Count 来控制循环体执行次数。

```
Count    SETA    1          ;
         WHILE   count<5     ;
Count    SETA    count+1     ;
…
…
WEND
```

3．MACRO、MEND

1）语法格式

MACRO、MEND 伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，会被替换为用户定义的符号。

宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。

宏操作的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度，但在使用子程序结构时需要保护现场，从而增加了系统的开销。因此，在代码较短且需要传递的参数较多时，可以使用宏操作代替子程序。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。

MACRO、MEND 伪操作可以嵌套使用。

语法格式如下：

```
  MACRO
{$label} macroname {$parameter{,$parameter}…}
  ;code
  MEND
```

参数说明如下。

- {$label}：$标号在宏指令被展开时，会被替换为用户定义的符号。通常，在一个符号前使用“$”表示该符号被汇编器编译时，使用相应的值代替该符号。
- macroname：所定义的宏的名称。
- $parameter：宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的参数。

2）使用说明

在子程序代码比较短，而需要传递的参数比较多的情况下可以使用宏汇编技术。

首先通过 MACRO 和 MEND 伪操作定义宏，包括宏定义体代码。在 MACRO 伪操作之后的第一行声明宏的原型，其中包含该宏定义的名称及需要的参数。在汇编中可以通过该宏定义的名称来调用它。当源程序被编译时，汇编器将展开每个宏调用，用宏定义体代替源程序中宏定义的名称，并用实际参数值代替宏定义时的形式参数。

3）示例

（1）没有参数的宏定义如下：

```
MACRO
CSI_SETB                ;宏名为 CSI_SETB，无参数
LDR  R0,=rPDATG         ;读取 GPGO 口的值
LDR  R1,[r0]
LDR  R1,R1,#0x01        ;CSI 置位
STR    R1,[R0]          ;输出控制
MEND
```

（2）带参数的宏定义如下：

```
MACRO
$IRQ_Label       HANDLER    $IRQ_Exception
                 EXPORT     $IRQ_Label
                 IMPORT     $IRQ_Exception
$IRQ_Label
         SUB      LR,LR,#4
         SEMFD     SP!,{R0-R3,R12,LR}
         MRS  R3,STSR
         STMFD     SP!.{R3}
…
MEND
```

（3）下面的程序显示了一个完整的宏定义和调用过程。

```
;宏定义
        MACRO                           ;开始宏定义
```

华清远见

```
$label  mymacro  $p1,$p2
;code
$label.loop1                   ;代码段
; code
BGE $label.loop1
$label.loop2                   ;代码段
BL       $p1
BGT $label.loop2
;代码段
ADR $p2
;代码段
MEND

;程序汇编后，宏展开结果
abc         mymacro    subr1,de   ;使用宏
          ;代码段
abcloop1  ;代码段
          ;代码段
          BGE abcloop1
Abcloop2  ;代码段
          BL    subr1
          BGT abcloop2
          ;代码段
          ADR de
          ;代码段
```

4. MEXIT

1）语法格式

MEXIT 用于从宏定义中跳转出去。
语法格式如下：

```
MEXIT
```

2）示例

```
    MACRO
$abc    macro    abc    $param1,$param2
;code
       WHILE    condition1
       ;code
       IF    condition2
            ;代码段
            MEXIT
       ELSE
       ;代码段
       ENDIF
       WEND
       ;代段段
       MEND
```

5. 关于伪操作的嵌套

下面的伪操作在使用时可以嵌套，但嵌套的深度不能超过 256。
（1）MACRO 宏定义。
（2）WHILE…END 循环。
（3）IF…ELSE…ENDIF 条件语句。
（4）INCLUDE 指定头文件。

4.1.5 杂项伪操作

ARM 汇编中还有一些其他的伪操作经常会被使用，包括以下几条：
（1）AREA 用于定义一个代码段或数据段。
（2）ALIGN 用于使程序当前位置满足一定的对齐方式。

（3）ENTRY 用于指定程序入口点。
（4）END 用于指示源程序结束。
（5）EQU 用于定义字符名称。
（6）EXPORT（或 GLOBAL）用于声明符号可以被其他文件引用。
（7）EXPORTAS 用于向目标文件引入符号。
（8）IMPORT 用于通知编译器当前符号不在本文件中。
（9）EXTERN 用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用。
（10）GET（或 INCLUDE）用于将一个文件包含到当前源文件。
（11）INCBIN 用于将一个文件包含到当前源文件。

1．ALIGN

1）语法格式

ALIGN 伪操作可通过添加填充字节的方式，使当前位置满足一定的对齐方式。
语法格式如下：

```
ALIGN{expr{,offset{,pad}}}
```

参数说明如下。

- expr：对齐表达式。表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。
- offset：偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：n*expr+偏移量。

注意

编时变量，由编译器根据内存对齐方式决定其值。

- pad：用做填充的字节。如果没有指定 pad，用零填充。

2）使用说明

ALIGN 伪操作使程序代码和数据保持正确的内存对齐方式。在下面的情况中，要求特定的地址对齐方式：

（1）Thumb 伪指令 ADR 要求加载的地址是字对齐的，但 Thumb 代码中的标号不一定是字对齐的，这就要使用伪操作 ALIGN4 来确保程序中 Thumb 代码的地址标号是字对齐的。

（2）可以使用伪操作 ALIGN 来更有效地使用 Cache。比如，在 ARM940T 体系结构中，Cache 是 16 字节对齐的，这时使用 ALIGN4 指定 16 字节的内存对齐方式可以充分发挥 Cache 的性能优势。

（3）Ldrd 和 Strd 双字传送指令要求内存 8 字节对齐，这样在 Ldrd/Strd 指令所有访问的内存单元前使用 ALIGN3 实现 8 字节对齐方式。

注意

在伪操作 AREA 后使用的 ALIGN 和直接使用伪操作 ALIGN 有所不同，详见 AREA 伪操作。

3）示例

（1）通过 ALIGN 伪操作使程序中的地址标号字对齐。

```
    AREA    Example,CODE,READONLY    ;声明一个名为 Example 的代码段
START  LDR R0,=Sdfjk
…
    MOV    PC,LR
Sdfjk DCB 0x58                       ;定义一个字节存储空间，字对齐方式被破坏
    ALIGN                            ;声明字对齐
SUBIMOV    R1,R3                     ;其他代码
…
        MOV    PC,LR
```

（2）将一个可能被 Cache 的功能段入口定义在 16 字节边界上。

```
        AREA    Cacheable, CODE, ALIGN=4
Rout1                                ;名称为 Cacheable 的代码段在 16 字节边界上对齐
```

```
        ;代码段
        MOV    pc,lr                 ;字边界上对齐
        ALIGN 16                     ;16 字节边界对齐
Rout2   ;代码段
…
```

（3）下面的 ALIGN 伪操作使用了 offset 偏移量。

```
AREA    OffsetExample,CODE
DCB 1
ALIGN  4,3
DCB 1
```

2. AREA

1）语法格式

AREA 伪指令用于定义一个代码段或数据段。

ARM 程序采用分段式设计，一个 ARM 源程序至少需要一个代码段，大的程序可以包含多个代码段和数据段。关于“段”更详细的描述，可以参考相关文档。

语法格式如下：

```
AREA  sectionname{,attr}{,attr}
```

参数说明如下。

- sectionname：指定所定义段的段名。段名若以数字开头，则该段名需用“|”括起来，如：|1_test|。

注意 一些代码段具有约定的名称，如|.text|表示 C 语言编译器产生的代码段或者与 C 语言库相关的代码段。

- attr：指定代码段或数据段的属性。

在 AREA 伪操作中，各属性之间用逗号隔开。表 4-1 所示为各段属性及相关说明。

表 4-1 段属性及说明

段 属 性	说 明
ALIGN = expr	默认情况下，ELF 的代码段和数据段是 4 字节对齐的，expr 可以取 0～31 的数值，相应的对齐方式为 2^{expr} 字节对齐。如 expr=10，表示代码段为 1k 边界对齐。expr 不能为 0 或 1
ASSOC = section	指定与本段相关的 ELF 段，任何时候连接 section 段必须包含 sectionname 段
CODE	指示该段为代码段。READONLY 为默认属性
COMDEF	定义一个通用的段，该段可以包含代码或者数据。在多个源文件中同名的 COMDEF 段必须相同。如果同名的 COMDEF 段不同，连接器会报错
COMMON	定义一个通用的数据段。该段不包括任何用户代码和数据，它被连接器自动初始化为 0。相同名称的 COMMON 段使用相同的内存单元，每个 COMMON 段的大小不必相同，连接器为其分配最大尺寸的内存
DATA	定义数据段，默认属性为 READWRITE
NOALLOC	指定该段为虚段，并不为其在目标系统上分配内存
NOINIT	指定本数据段不被初始化或仅初始化为 0。该操作仅为 SPACE/DCB/DCD/DCDU/DCQ/DCQ/ DCW/DCWU 伪操作保留了内存单元
READONLY	指定该段不可写，为程序代码段
READWRITE	指定可读可写段。数据段的默认属性

2）使用说明

编程时使用 AREA 伪操作将程序分成多个 ELF 格式的段，段名称可以相同，这时同名的段被放在同一个 ELF 段中。ELF 段的属性根据第一个出现的 AREA 伪操作的属性设定。

一般情况下，数据段和代码段是分离的。大的程序应该被分成多个不同的代码段和数据段。一个汇编程序至少包含一个段。

华清远见

3）示例

下面的伪操作定义了一个代码段，段名为 Init，属性为只读。

```
AREA    Init, CODE, READONLY
;code
```

3. END

1）语法格式

END 伪操作用于通知编译器已经到了源程序的结尾。
语法格式如下：

```
END
```

2）使用说明

每一个汇编源文件必须以 END 结束。
如果汇编文件通过伪操作 GET 指定了一个“父文件（parent file）”，当汇编器遇到 END 伪操作时将返回到“父文件”继续汇编。

3）示例

使用 END 伪操作指定应用程序的结尾。

```
AREA    Init, CODE, READONLY
…
END
```

4. ENTRY

1）语法格式

ENTRY 伪操作用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY（也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件中最多只能有一个 ENTRY（可以没有）。
语法格式如下：

```
ENTRY
```

2）使用说明

在一个完整的汇编程序中至少要有一个 ENTRY，如果在程序连接时没有发现 ENTRY 伪操作，连接器将产生警告信息。
在一个源文件里最多只能有一个 ENTRY，如果多个 ENTRY 同时出现在源文件中，汇编时将产生错误信息。

3）示例

使用伪操作 ENTRY 指定程序入口点。

```
AREA    Init, CODE, READONLY
ENTRY                    ;指定应用程序的入口点
…
```

5. EQU

1）语法格式

EQU 伪操作用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的#define。其中 EQU 可用“*”代替。
语法格式如下：

```
name  EQU  expr{,type}
```

参数说明如下。

- name：EQU 伪指令定义的字符名称。

华清远见

- expr：32 位表达式，其值为基于寄存器的地址值、程序中的标号、32 位的地址常量或 32 位的常量。
- type：指定数据类型，为一个可选项。

当表达式 expr 为 32 位的常量时，可以指定表达式的数据类型。可以有以下几种类型： CODE16、CODE32、ARM、THUMB 和 DATA。

当定义的名称（name）被声明为可被其他文件引用（exported）时，在目标文件的符号表中将包含名称（name）的数据类型。这些信息将会被连接器使用。

2）使用说明

EQU 类似于 C 语言中的#define 操作。

3）示例

定义标号 Test 的值为 50，定义 Addr 的值为 0x55。

```
Test  EQU  50              ;定义标号 Test 的值为 50
Addr  EQU  0x55,CODE32     ;定义 Addr 的值为 0x55，且该处为 32 位的 ARM 指令
```

6．EXPORT（或 GLOBAL）

1）语法格式

EXPORT 伪操作用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写。

语法格式如下：

```
EXPORT{symbol}{[WEAK,attr]}
```

参数说明如下。

- symbol：被声明的符号名称，名称区分大小写。如果 symbol 被忽略，所有符号被定义为可以被其他文件引用属性。
- [WEAK]：该选项声明其他的同名标号优先于该标号被引用。
- [attr]：符号属性，用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局（global）的符号对其他文件是“可见的”，也就是说可以被其他文件引用；而定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

attr 可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能被其他文件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

2）使用说明

EXPORT 声明的变量可以被其他文件访问。

3）示例

声明一个可全局引用的标号 Stest。

```
AREA    Init,CODE,READONLY
EXPORT        Stest              ;声明一个可全局引用的标号 Stest
…
END
```

7．EXPORTAS

1）语法格式

EXPORTAS 用于修改已被编译的目标文件中的符号。

语法格式如下：

```
EXPORTAS  symbol1,symbol2
```

参数说明如下。

- symbol1：源文件中的符号名。symbol1 必须在源文件中已被定义。它可以是段名、标号或常量。
- symbol2：目标文件中的符号名。它将取代目标文件中的 symbol1 符号。该符号名称区分大小写。

2）使用说明

用于修改目标文件中的符号定义。

3）示例

```
AREA data1, DATA            ;定义新的数据段 data1
AREA data2, DATA            ;定义新的数据段 data2
 EXPORTAS data2, data1      ;data2 中定义的符号将会出现在 data1 的符号表中
one EQU  2
     EXPORTAS one, two
     EXPORT one             ;符号 two 将在目标文件中以“2”的形式出现
```

8．EXTERN

1）语法格式

EXTERN 伪操作用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。标号在程序中区分大小写。

语法格式如下：

```
EXTERN  symbol{[WEAK,attr]}
```

参数说明如下。

- symbol：要引用的符号名称。该名称区分大小写。
- [WEAK]：该选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息。在多数情况下将该标号置为 0；若该标号被 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。
- [attr]：符号属性，用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局（global）的符号对其他文件是“可见的”，也就是说，可以被其他文件引用；而定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

attr 可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能被其他文件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

2）使用说明

当源文件的符号用 EXTERN 声明后，该符号在连接时被解析。

3）示例

（1）通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义。

```
AREA    Init, CODE, READONLY
EXTERN       Main ;通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义
…
END
```

（2）下面的程序用于检测 C++库是否被连接，并根据检测结果，执行指令跳转。

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]            ;如果 C++ 库被连接
                                          ;得到__CPP_INITIALIZE 函数的入口地址
 LDR     R0,=__CPP_INITIALIZE             ;如果没有被连接，地址为 0
 CMP     R0,#0                            ;如果为 0
 BEQ     nocplusplus                      ;跳转到相应函数
```

9．GET（或 INCLUDE）

1）语法格式

GET 伪操作用于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进行汇编处理。可以使用 INCLUDE 代替 GET。

语法格式如下：

```
GET  filename
```

其中，filename 是被包含的文件名称。ARM 汇编器接受的路径名称可以是 UNIX 或 MS-DOS 的路径格式。

2）使用说明

汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。其使用方法与 C 语言中的“include”相似。

GET 伪操作只能用于包含源文件，包含目标文件需要使用 INCBIN 伪操作。

3）示例

通知编译器当前源文件包含源文件 a1.s 和源文件 C:\ a2.s。

```
AREA    Init, CODE, READONLY
GET  a1.s      ;通知编译器当前源文件包含 a1.s
GE  T    C:\a2.s    ;通知编译器当前源文件包含 C:\a2.s
…
END
```

10. IMPORT

1）语法格式

IMPORT 伪操作用于通知编译器要使用的标号在其他的源文件中定义。标号在程序中区分大小写。

注 意

IMPORT 和 EXTERN 用法相似，IMPORT 声明的符号无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。而 EXTERN 声明的符号，如果当前源文件实际并未引用该标号，则该标号就不会被加入到当前源文件的符号表中。

语法格式如下：

```
IMPORT  symbol{[WEAK, attr]}
```

参数说明如下。

- symbol：被声明的符号名称。名称区分大小写。如果 symbol 被忽略，所有符号被定义为可以被其他文件引用的属性。
- [WEAK]：该选项表示当所有的源文件都没有定义这样一个标号时，编译器不给出错误信息。在多数情况下将该标号置为 0；若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。
- [attr]：符号属性，用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局（global）的符号对其他文件是“可见的”，也就是说，可以被其他文件引用；定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

attr 可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能被其他文件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

2）使用说明

当源文件的符号用 IMPORT 声明后，该符号在连接时被解析。

3）示例

参见 EXTERN 伪操作。

华清远见

11. INCBIN

1）语法格式

INCBIN 伪操作用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不做任何变动地存放在当前文件中，编译器从其后开始继续处理。

语法格式如下：

```
INCBIN  filename
```

其中，filename 指定将要包含进当前源文件的文件名。汇编器接受的路径名称可以是 UNIX 或 MS-DOS 的路径格式。

2）使用说明

使用 INCBIN 可以包含任何格式的文件，如二进制文件、字符文件等。汇编器对此文件内容不做任何修改。

3）示例

通知编译器当前源文件包含文件 a1.dat 和 C:\a2.txt。

```
AREA    Init,CODE,READONLY
INCBIN       a1.dat    ;通知编译器当前源文件包含文件 a1.dat
INCBIN  C:\a2.txt      ;通知编译器当前源文件包含文件 C:\a2.txt
…
END
```

4.2 ARM 汇编器支持的伪指令

ARM 汇编器支持 ARM 伪指令，这些伪指令在汇编阶段被翻译成 ARM 或者 Thumb（或 Thumb-2）指令（或指令序列）。ARM 伪指令包含 ADR、ADRL、LDR 等。

1. ADR 伪指令

1）语法格式

ADR 伪指令为小范围地址读取伪指令。ADR 伪指令将基于 PC 相对偏移地址或基于寄存器相对偏移地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−255～255B；当地址值是字对齐时，取值范围为−1 020～1 020。当地址值是 16 字节对齐时其取值范围更大。

语法格式如下：

```
ADR{cond}{.W}  register,label
```

参数说明如下。

- cond：可选的指令执行条件。
- W：可选项。指定指令宽度（Thumb-2 指令集支持）。
- register：目标寄存器。
- label：基于 PC 或具有寄存器的表达式。

2）使用说明

ADR 伪指令被汇编器编译成一条指令。汇编器通常使用 ADD 指令或 SUB 指令来实现伪操作的地址装载功能。如果不能用一条指令来实现 ADR 伪指令的功能，汇编器将报告错误。

3）示例

```
        LDR     R4,=data+4*n      ;n 是汇编时产生的变量
        ;code
        MOV     pc,lr
data    DCD     value0
        DCD     valuen            ;n-1 条 DCD 伪操作
                                  ;所要装载入 R4 的值
                                  ;更多 DCD 伪操作
```

2．ADRL 伪指令

1）语法格式

ADRL 伪指令为中等范围地址读取伪指令。ADRL 伪指令将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−64～64KB；当地址值是字对齐时，取值范围为−256～256KB；当地址值是 16 字节对齐时，其取值范围更大。在 32 位的 Thumb-2 指令中，地址取值范围到达−1～1MB。

语法格式如下：

```
ADRL{cond}  register,label
```

参数说明如下。

- cond：可选的指令执行条件。
- register：目标寄存器。
- label：基于 PC 或具体寄存器的表达式。

2）使用说明

ADRL 伪指令与 ADR 伪指令相似，都是用于将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中。有所不同的是，ADRL 伪指令比 ADR 伪指令可以读取更大范围的地址。这是因为在编译阶段，ADRL 伪指令被编译器换成两条指令。即使一条指令可以完成该操作，编译器也将产生两条指令，其中一条为多余指令。如果汇编器不能在两条指令内完成操作，将报告错误，中止编译。

3．LDR 伪指令

1）语法格式

LDR 伪指令装载一个 32 位的常数和一个地址到寄存器。

语法格式如下：

```
LDR{cond}{.W}  register,=[expr|label-expr]
```

参数说明如下。

- cond：可选的指令执行条件。
- .W：可选项。指定指令宽度（Thumb-2 指令集支持）。
- register：目标寄存器。
- expr：32 位常量表达式。汇编器根据 expr 的取值情况，对 LDR 伪指令做如下处理。
  - 当 expr 表示的地址值没有超过 MOV 指令或 MVN 指令的地址取值范围时，汇编器用一对 MOV 和 MVN 指令代替 LDR 指令。
  - 当 expr 表示的指令地址值超过了 MOV 指令或 MVN 指令的地址范围时，汇编器将常数放入数据缓存池，同时用一条基于 PC 的 LDR 指令读取该常数。
- label-expr：一个程序相关或声明为外部的表达式。汇编器将 label-expr 表达式的值放入数据缓存池，使用一条程序相关 LDR 指令将该值取出放入寄存器。

当 label-expr 被声明为外部的表达式时，汇编器将在目标文件中插入链接重定位伪操作，由链接器在连接时生成该地址。

2）使用说明

当要装载的常量超出了 MOV 指令或 MVN 指令的范围时，使用 LDR 指令。

由 LDR 指令装载的地址是绝对地址，即 PC 相关地址。

当要装载的数据不能由 MOV 指令或 MVN 指令直接装载时，该值要先放入数据缓存池，此时 LDR 伪指令处的 PC 值到数据缓存池中目标数据所在地址的偏移量有一定限制。ARM 或 32 位的 Thumb-2 指令中该范围是−4～4KB，Thumb 或 16 位的 Thumb-2 指令中该范围是 0～1KB。

3）示例

（1）将常数 0xff0 读到 R1 中。

华清远见

```
LDR R3,=0xff0 ;
```

相当于下面的 ARM 指令：

```
MOV R3,#0xff0
```

（2）将常数 0xfff 读到 R1 中。

```
LDR R1,=0xfff ;
```

相当于下面的 ARM 指令：

```
LDR R1,[pc,offset_to_litpool]
…
litpool DCD 0xfff
```

（3）将 place 标号地址读入 R1 中。

```
LDR R2,=place ;
```

相当于下面的 ARM 指令：

```
LDR R2,[pc,offset_to_litpool]
…
litpool DCD place
```

4.3 ARM 汇编语言的语句格式

4.3.1 ARM 汇编语言中的符号

在汇编语言程序设计中，经常使用各种符号代替地址（addresses）、变量（variables）和常量（constants）等，以增加程序的灵活性和可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下约定：

（1）符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。

（2）符号在其作用范围内必须唯一。

（3）自定义的符号名不能与系统的保留字相同，其中保留字包括系统内部变量（built in variable）和系统预定义符号（predefined symbol）。

（4）符号名不应与指令或伪指令同名。如果要使用和指令或伪指令同名的符号，要用双斜杠“||”将其括起来，如||ASSERT||。

注意

虽然符号被双斜杠括起来，但双斜杠并非符号名的一部分。

（5）局部标号以数字开头，其他的符号都不能以数字开头。

1．变量（variable）

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有 3 种：

（1）数字变量（numeric）。数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。

（2）逻辑变量（logical）。逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真（{TURE}）和假（{FALSE}）。

（3）字符串变量（string）。字符串变量用于在程序的运行中保存一个字符串，注意字符串的长度不应超出字符串变量所能表示的范围。

在 ARM（Thumb）汇编语言程序设计中，可使用 GBLA、GBLL、GBLS 伪指令声明全局变量，使用 LCLA、LCLL、LCLS 伪指令声明局部变量，使用 SETA、SETL 和 SETS 对其进行初始化。

2．常量（constants）

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

（1）数字常量一般为 32 位的整数，当作为无符号数时，其取值范围为 $0\sim2^{32}-1$；当作为有符号数时，其取值范围为$-2^{31}\sim2^{31}-1$。汇编器认为$-n$ 和 $2^{32}-n$ 是相等的。对于关系操作，如比较两个数的大小，汇编器将其操作数看成无符号的数，也就是说，“0>−1”对汇编器来说取值为“假（{FLASE}）”。

（2）逻辑常量只有两种取值情况：真或假。

（3）字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

3．程序中的变量代换

汇编语言中的变量可以作为一整行出现在汇编程序中，也可以作为行的一部分使用。

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

（2）如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。

（3）如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量。

（4）如果程序中需要字符“$”，则可以用“$$”来表示，汇编器将不进行变量替换，而是将“$$”作为“$”。

下面的两个例子说明了变量替换的过程。

```
;直接的变量替换
        GBLS    add4ff
        ;
add4ff  SETS    "ADD  r4,r4,#0xFF"   ;给变量 add4ff 赋值
        $add4ff.00 ;引用变量
        ; codes
        ADD  r4,r4,#0xFF00
    ;有特殊符号的变量替换
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ;s1 =a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|" ;fixup= |xyabcz|
|C$$code|   MOV     r4,#16  ;label= C$$code
```

4．程序标号（label）

在 ARM 汇编中，标号代表一个地址，段内标号的地址在汇编时确定，而段外标号地址值在连接时确定。根据标号的生成方式，程序标号分为以下 3 种：

（1）程序相关标号（Program-relative labels）。程序相关标号指位于目标指令前的标号或程序中的数据定义伪操作前的标号。这种标号在汇编时将被处理成 PC 值加上或减去一个数字常量。它常用于表示跳转指令的目标地址或代码段中所嵌入的少量数据。

（2）寄存器相关地址（Register-relative labels）。这种标号在汇编时将被处理成寄存器的值加上或减去一个数字常量。它常被用于访问数据段中的数据。这种基于寄存器的标号通常用 MAP 和 FIELD 伪操作定义，也可以用 EQU 伪操作定义。

（3）绝对地址（Absolute address）。绝对地址是一个 32 位的数字量，使用它可以直接寻址整个内存空间。

5．局部标号

局部标号是一个 0～99 之间的十进制数字，可重复定义。局部标号后面可以紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围为当前段，也可以用伪操作 ROUT 来定义局部标号的作用范围。

局部标号在子程序或程序循环中常被用到，也可以配合宏定义伪操作（MACRO 和 MEND）来使程序结构更加合理。

在同一个段中，可以使用相同的数字命名不同的局部变量。默认情况下，汇编器会寻址最近的变量。也可以通过汇编器命令选项来改变搜索顺序。

局部变量命名语法如下：

```
n{routname}
```

局部变量引用的语法格式如下：

```
%{F|B}{A|T}n{routname}
```

其中，routname 为变量作用范围名称；%表示引用操作；F 指示汇编器只向前搜索；B 指示汇编器只向后搜索；A 指示汇编器搜索所有宏的嵌套；T 指示汇编器只搜索宏的当前层。

如果在引用过程中没有指定 F 和 B，则汇编器先向后搜索，再向前搜索。

如果 A 和 T 没有指定，汇编器搜索所有从当前层次到宏最高层次，比当前层次低的层次不再搜索。

如果指定了 routname，汇编器向前搜索最近的 ROUT 操作；若 routname 与该 ROUT 伪操作定义的名称不匹配，汇编器报告错误并结束汇编。

4.3.2 ARM 汇编语言中的表达式和运算符

在汇编语言程序设计中经常使用各种表达式，表达式一般由变量、常量、运算符和括号构成。常用的表达式有数字表达式、逻辑表达式和字符串表达式。下面分别介绍表达式中的各元素。

1．字符串表达式

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。字符串由包含在双引号内的一系列字符组成。编译器所支持的字符串最大长度为 512 字节。

当在字符串中包含“$”或引号时，可以用“$$”表示“$”，用两个双引号表示一个双引号。例如：

```
abc     SETS    "one "" double quote"
def     SETS    "one $$ dollar symbol"
```

上面的例子分别将字符串 abc 和 def 赋值为“one " "double quote”和“one $ dollar symbol”。

字符串可以通过 SETA、SETL、SETS 伪操作对其赋值。

常用的与字符串表达式相关的运算符如下。

（1）LEN：计算字符串长度运算符。

（2）CHR：ASCII 码转换运算符。

（3）STR：字符串转换运算符。

（4）LEFT：字符串取左运算符。

（5）RIGHT：字符串取右运算符。

（6）CC：字符串连接运算符。

下面的例子说明了如何使用字符串操作符给字符串变量赋值。

```
improb  SETS    "literal":CC:(abc:LEFT:4)
```

这个例子将字符串赋值为“literaluote”。

2．整数表达式

整数表达式一般由数字常量、数字变量、数字运算符和括号构成。

整数表示式可以包含寄存器相关（register-relative）或程序相关（program-relative）表达式，这些表达式在编译时被汇编器翻译为地址无关数字常量。

整数表达式一般被计算为 32 位的整数，当此整数被定义为无符号数时，其取值范围为 $0 \sim 2^{32}-1$；当被定义为有符号数时，其取值范围为$-2^{31} \sim 2^{31}-1$。汇编器认为-n 和 $2^{32}-n$ 是相等的。对于关系操作，如比

华清远见

较两个数的大小，汇编器将其操作数看成无符号的数，也就是说，“0>-1”对汇编器来说取值为“假({FLASE})”。

下面的例子说明了在程序中如何对整数表达式进行操作。

```
a   SETA    256*256          ;将数字变量赋值为 256*256
    MOV     r1,#(a*22)       ;将数字表达式(a*22) 的值放入 r1
```

在汇编语言中，整数数字量有以下几种形式：

（1）十进制数（decimal-digis）。

（2）“0x” + 十六进制数（0xhexadecimal-digits）。

（3）“&” + 十六进制数（&hexadecimal-digits）。

（4）n 进制数（n_base-n-digits）。

（5）字符（character）。

其中，十进制数（decimal-digis）可以是“0”到“9”数字的任意组合；十六进制数（hexadecimal-digits）可以是“0”到“9”数字和字母“A”到“F”的任意组合；“n_”可以取 2~9，“base-n-digits”是在 n 进制下合法的任意数值；字符（character）可以是除单引号以外的所有字符。

下面的例子说明了整数表达式的基本用法。

```
a      SETA    34906
addr   DCD     0xA10E
       LDR     r4,=&1000000F
       DCD     2_11001010
c3     SETA    8_74007
       DCQ     0x0123456789abcdef
       LDR     r1,='A'      ;ARM 伪指令将整数 65（A 的 ASCII 码）存入寄存器
       ADD     r3,r2,#'\''  ;将整数 39（字符“/”的 ASCII 码）加到 r2，结果存入 r3
```

3．浮点数字量表达式

浮点数字量有以下几种形式：

（1）{-}digitsE{-}digits。

（2）{-}{digits}.digits{E{-}digits}。

（3）0xhexdigits。

（4）&hexdigits。

其中，digits 为十进制数，要在其后加上字母 E（大写或小写）来表示其指数；hexdigits 为十六进制数。

单精度浮点数的表示范围为 1.175 494 35e−38～3.402 823 47e+38；双精度浮点数的表示范围为 2.225 073 858 507 201 38e−308～1.797 693 134 862 315 71e+308。

下面的例子说明了浮点数据量的基本用法。

```
DCFD    1E308,-4E-100
DCFS    1.0
DCFD    3.725e15
LDFS    0x7FC00000           ;
LDFD    &FFF0000000000000    ;
```

4．逻辑表达式

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符有“=”、“>”、“<”、“>=”、“<= ”、“/=”、“ <>” 运算符和“LAND”、“LOR”、“LNOT”、“LEOR”运算符。

5．程序或寄存器相关表达式

寄存器相关表达式的值等于指定寄存器的值加上或减去一个数字表达式。

程序相关表达式的值等于程序计数器 PC 的值加上或减去一个数字表达式的值。此种表达式通常由程序中的标号与一个数字表达式组成。

下面的例子说明了程序或寄存器相关表达式的基本使用方法。

```
       LDR     r4,=data+4*n      ;n 是汇编时取值变量
```

```
        ; code
        MOV     pc,lr
data    DCD     value0
        ; n-1 个 DCD 伪操作
        DCD     valuen             ;data+4*n 指向此
        ;更多 DCD 伪操作
```

6. 汇编中的操作符

1）操作符的优先级

在汇编语言程序设计中，表达式包含一个扩展的操作符集，这些操作符和高级语言中的运算符十分接近。其运算次序遵循如下的优先级：

（1）优先级相同的双目运算符的运算顺序为从左到右。

（2）相邻的单目运算符的运算顺序为从右到左，单目运算符的优先级高于其他运算符。

（3）括号运算符的优先级最高。

汇编语法的操作符优先级和 C 语言中的不完全相同。例如在汇编中，汇编语言(1+2:SHR:3)相当于(1+(2:SHR:3))，而在 C 语言中，运算则变为((1+2)>>3)=0。类似于这样的操作，在使用时要特别注意。

注意 保证表达式运算结果的正确，建议使用“()”来避免异议。

表 4-2 列出了汇编操作符的优先级及对应的 C 语言运算符。

表 4-2　汇编操作符优先级

汇编操作符	C 语言运算符
单目运算	单目运算
* / :MOD:	* / %
字符串操作	n/a
:SHL::SHR::ROR::ROL:	<< >>
+ - :AND: :OR: :EOR:	+ — $ |
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&& ||

说明 表 4-2 是按操作符的优先级从上到下排列的。

C 语言运算符优先级从高到低排列如下：

- 单目运算。
- * / %。
- + - (as binary operators)。
- << >>。
- < <= > >=。
- == !=。
- &。
- ^。
- |。
- &&。
- ||。

2）单目运算

最高优先级的单目运算在表达式中最先被计算。单目操作符写在操作数的前面。运算顺序为从右到左。表 4-3 列出了汇编中单目运算操作符及其返回值。

华清远见

表 4-3 汇编中单目运算操作符及其返回值

操 作 符	使 用	描 述
:CHR:	:CHR:A	返回字母 A 的 ASCII 码
:LOWERCASE	:LOWERCASE:string	将给定字符串中的所有大写字母变成小写
REVERSE_CC	:REVERSE_CC:cond_code	对条件码取反
:STR:	:STR:A	将一个数字量或逻辑表达式转换成串
:UPPERCASE:	:UPPERCASE:string	将给定字符串中的所有小写字母变成大写
?	? A	返回定义符号 A 的代码行所产生代码行的字节数
+和–	+A 和-A	单目加和单目减，操作数为数学或程序相关表达式
:BASE:	:BASE:A	如果 A 是程序或寄存器相关表达式，:BASE:返回基址寄存器的编号
:CC_ENCODING:	:CC_ENCODING:cond_code	返回条件码中的数字值
:DEF:	:DEF:A	判断 A 是否被定义，如果被定义返回{TRUE}；如果没有被定义返回{FALSE}
:INDEX:	:INDEX:A	如果 A 是寄存器相关表达式，:INDEX:返回 A 相对于寄存器的偏移量。常用在宏操作中
:LEN:	:LEN:A	字符串 A 的长
:LNOT:	:LNOT:A	逻辑表达式 A 的值取反
:NOT:	:NOT:A ~A	A 的值按位取反
:RCONT:	:RCONT:Rn	返回寄存器编号，0～15 对应寄存器 r0～r15

3）双目运算

ARM 汇编中将双目运算符放在两个操作数中间。一般情况下，双目运算的优先级低于单目运算。下面将以操作符的优先级为序分别介绍各操作符。

注意

操作符的优先级与 C 语言中操作符的优先级顺序略有不同，详见“单目运算”一节。

表 4-4 列出了乘法相关操作符。

表 4-4 乘法操作符

操 作 符	别 名	使 用	说 明
*		A*B	乘法操作
/		A/B	除法操作
:MOD:	%	A:MOD:B	以 B 为除数对 A 取模

乘法相关操作符包括乘、除、取模运算，在双目运算中具有最高优先级。这些运算的操作数只能是数字表达式。

表 4-5 列出了字符串相关操作符。

表 4-5 字符串操作符

操 作 符	使 用	说 明

:CC:	A:CC:B	连接两个字符串
:LEFT:	A:LEFT:B	返回字符串 A 最左端 B 长度的字符，操作数 A 必须为字符串，B 必须为整数表达式
:RIGHT:	A:RIGHT:B	返回字符串 A 最右端 B 长度的字符，操作数 A 必须为字符串，B 必须为整数表达式

表 4-6 列出了移位操作符。移位操作中两个操作数均为数字表达式。

表 4-6　移位操作符

操　作　符	别　　名	使　　用	说　　明
:ROL:		A:ROL:B	A 循环左移 B 位
:ROR:		A:ROR:B	A 循环右移 B 位
:SHL:	<<	A:SHL:B	A 左移 B 位
:SHR:	>>	A:SHR:B	A 右移 B 位

注意

SHR是逻辑右移，不影响符号位。

表 4-7 列出了所有加、减、逻辑操作符。

表 4-7　加、减、逻辑操作符

操　作　符	别　　名	使　　用	说　　明
+		A+B	A 加上 B
−		A−B	从 B 中减去 A
:AND:	&&	A:AND:B	A 和 B 按位与
:EOR:	^	A:EOR:B	A 和 B 按位异或
:OR:	||	A:OR:B	A 和 B 按位或

加、减运算的操作数均为数字表达式。逻辑运算的表达式为数字表达式，此运算按位操作产生结果。

表 4-8 列出了 ARM 汇编中的关系操作符。关系操作符用于表示两个同类表达式之间的关系。关系符的两个操作数必须为同种类型的操作数。操作数可以是数字变量、程序相关表达式、寄存器相关表达式或字符串。

表 4-8　关系操作符

操　作　符	别　　名	使　　用	说　　明
=	==	A=B	判断 A 是否等于 B
>		A>B	判断 A 是否大于 B
>=		A>=B	判断 A 是否大于等于 B
<		A<B	判断 A 是否小于 B
<=		A<=B	判断 A 是否小于等于 B
/=	<> !=	A/=B	判断 A 是否不等于 B

表 4-9 列出了汇编语言中的逻辑操作符。逻辑操作符进行两个逻辑表达式之间的基本逻辑操作，操作的结果为{FALSE}或{TURE}。

表 4-9　逻辑操作符

操　作　符	使　　用	说　　明
:LAND:	A:LAND:B	A 和 B 做逻辑与
:LEOR:	A:LEOR:B	A 和 B 做逻辑异或
:LOR:	A:LOR:B	A 和 B 做逻辑或

4.3.3　ARM 汇编语言内置的变量

华清远见

ARM 汇编器中定义了一些内置变量，这些内置变量不能使用伪指令设置（如 SETA、SETL、SETS 等），一般用于程序的条件汇编控制。

下面的例子显示了如何使用内置变量控制程序的执行流程。

```
If  {CONFIG}=16            ;若为 Thumb 代码，则执行 If 后的语句
;codes
else
;codes
endif
b                          ;程序结束
```

下面介绍由 ARM 汇编器预定义的内置变量。

（1）{ARCHITECTURE}：选定的 ARM 体系结构的值，如 3、3M、4、4T。

（2）{AREANAME}：当前段名。

（3）{ARMASM_VERSION}：ARM 编译器 ARMASM 的变量号。

（4）|ads$version|：ARM 编译器 ARMASM 的变量号，同{ARMASM_VERSION}。

（5）{CODESIZE}：如果当前指令为 ARM 指令，该内置变量取值为 32；如果当前指令为 Thumb 指令，该内置变量取值为 16，同{CONFIG}。

（6）{COMMANDLINE}：当前命令行内容。

（7）{CONFIG}：如果当前指令为 ARM 指令，该内置变量取值为 32；如果当前指令为 Thumb 指令，该内置变量取值为 16，同{CODESIZE}。

（8）{CPU}：所使用的 CPU 名称，默认为 ARM7TDMI。如果在编译命令行中使用“-CPU”选项确定 CPU 类型，则该值为“Generic ARM”。

（9）{ENDIAN}：如果编译器在大端模式下，则其值为“big”；如果在小端模式下，则其值为“little”。

（10）{FPIC}：默认为{FALSE}。如果设置了“/fpic”选项，其值为{TRUE}。

（11）{FPU}：所选 FPU 协处理器的名字，默认为“softVFP”。

（12）{INPUTFILE}：当前源文件名。

（13）{INTER}：默认为{FALSE}。如果设置了“/inter”选项，其值为{TRUE}。

（14）{LINENUM}：目前源文件行号。

（15）{NOSWST}：默认为{FALSE}。如果设置了“/noswst”选项，其值为{TRUE}。

（16）{OPT}：保存当前设置的列表选项。伪操作 OPT 用来保存当前列表选项，改变选项值，或恢复原始值。

（17）{PC} 或“.”：当前程序地址值。

（18）{PCSTOREOFFSET}：指令 STR pc,[...]和 STM Rb,{..., pc}与存储的 PC 值之间的偏移量。

（19）{ROPI}：默认为{FALSE}。如果设置了“/ropi”选项，其值为{TRUE}。

（20）{RWPI}：默认为{FALSE}。如果设置了“/rwpi”选项，其值为{TRUE}。

（21）{SWST}：默认为{FALSE}。如果设置了“/swst”选项，其值为{TRUE}。

（22）{VAR}或@：存储区位置寄存器的当前值。

4.4 ARM 汇编语言的程序结构

4.4.1 汇编语言的程序格式

在 ARM（Thumb）汇编语言程序中以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段（Code Section）和数据段（Data Section），代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

华清远见

可执行映像文件通常由以下几部分构成：

（1）一个或多个代码段，代码段的属性为只读。

（2）零个或多个数据段，数据段的属性为可读写。数据段可是被初始化的数据段或没有被初始化的数据段（Zero Initialized，ZI）。

链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置，因此源程序中段之间的相对位置与可执行的映像文件中段的相对位置一般不会相同。

以下是一个汇编语言源程序的基本结构。

```
        AREA    Init,CODE,READONLY
        ENTRY
Start
        LDR     R0,=0x3FF5000
        LDR     R1,0xFF
        STR     R1,[R0]
        LDR     R0,=0x3FF5008
        LDR     R1,0x01
        STR     R1,[R0]
        …
        END
```

在汇编语言程序中，用 AREA 伪操作定义一个段，并说明所定义段的相关属性。本例定义一个名为 Init 的代码段，属性为只读。ENTRY 伪操作标识程序的入口点。接下来为指令序列，程序的末尾为 END 伪指令，该伪操作告诉编译器源文件的结束。每一个汇编程序段都必须有一条 END 伪操作，指示代码段的结束。

4.4.2 汇编语言子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用指令“BL 子程序名”即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器 R0～R3 完成。

注意

不同编译器编译的代码间的相互调用要遵循 AAPCS（ARM Architecture），详见 ARM 编译工具手册。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
AREA     Init,CODE,READONLY
ENTRY
Start
LDR      R0,=0x3FF5000
LDR      R1,0xFF
STR      R1,[R0]
LDR      R0,=0x3FF5008
LDR      R1,0x01
STR      R1,[R0]
BL       PRINT_TEXT
…
PRINT_TEXT
…
      MOV          PC,BL
…
END
```

4.4.3 过程调用标准 AAPCS

为了使不同编译器编译的程序之间能够相互调用，必须为子程序间的调用规定一定的规则。AAPCS 就是这样一个标准。所谓 AAPCS，其英文全称为 Procedure Call Standard for the ARM Architecture，即 ARM 体系结构过程调用标准，它是 ABI（Application Binary Interface）标准的一部分。

可以使用“--apcs”选项告诉编译器将源代码编译成符号 AAPCS 调用标准的目标代码。

注意 “--apcs”选项并不影响代码的产生，编译器只是在各段中放置相应的属性，标识用户选定的 AAPCS 属性。

1．AAPCS 相关的编译/汇编选项

（1）none：指定输入文件不使用 AAPCS 规则。

（2）/interwork：指定输入文件符合 ARM/Thumb 交互标准。

（3）/nointerwork：指定输入文件不能使用 ARM/Thumb 交互。这是编译器默认选项。

（4）/ropi：指定输入文件是位置无关只读文件。

（5）/noropi：指定输入文件是非位置无关只读文件。这是编译器默认选项。

（6）/pic：同/ropi。

（7）/nopic：同/noropi。

（8）/rwpi：指定输入文件是位置无关可读可写文件。

（9）/norwpi：指定输入文件是非位置无关可读可写文件。

（10）/pid：同/rwpi。

（11）/nopid：同/norwpi。

（12）/fpic：指定输入文件编译成位置无关只读代码。代码中地址是 FPIC 地址。

（13）/swstackcheck：编译过程中对输入文件使用堆栈检测。

（14）/noswstackcheck：编译过程中对输入文件不使用堆栈检测。这是编译器默认选项。

（15）/swstna：如果汇编程序对于是否进行数据栈检查无所谓，而与该汇编程序连接的其他程序指定了选项/swst 或选项/noswst，这时该汇编程序使用选项/swstna。

2．ARM 寄存器使用规则

AAPCS 中定义了 ARM 寄存器使用规则如下：

（1）子程序间通过寄存器 R0、R1、R2、 R3 来传递参数。如果参数多于 4 个，则多出的部分用堆栈传递。被调用的子程序在返回前无须恢复寄存器 R0~R3 的内容。

（2）在子程序中，使用寄存器 R4~R11 来保存局部变量，如果在子程序中使用到了寄存器 R4~R11 中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；对于子程序中没有用到的寄存器则不必进行这些操作。在 Thumb 程序中，通常只能使用寄存器 R4~R7 来保存局部变量。

（3）寄存器 R12 用做子程序间 scratch 寄存器（用于保存 SP，在函数返回时使用该寄存器出栈），记为 ip。在子程序间的连接代码段中常有这种使用规则。

（4）寄存器 R13 用做数据栈指针，记为 sp。在子程序中寄存器 R13 不能用做其他用途。寄存器 sp 在进入子程序时的值和退出子程序时的值必须相等。

（5）寄存器 R14 称为连接寄存器，记为 lr，它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器 R14 则可以用做其他用途。

（6）寄存器 R15 是程序计数器，记为 pc。它不能用做其他用途。

（7）ARM 寄存器在函数调用过程中的保护规则，如图 4-1 所示。

华清远见

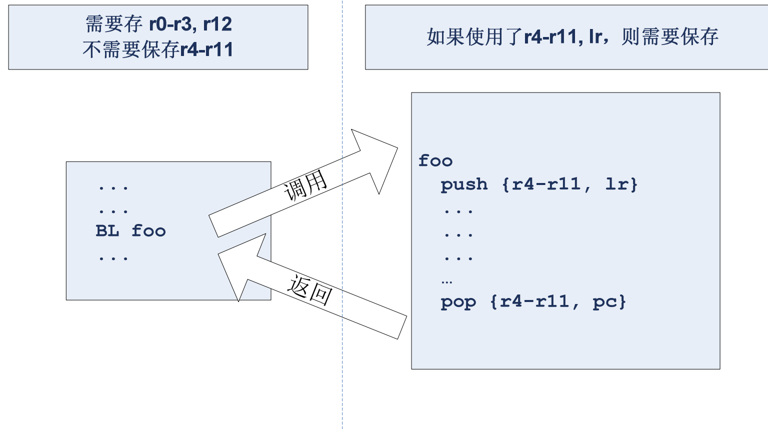

图 4-1 ARM 寄存器在函数调用中的保护规则

4.4.4 scatter 文件的使用

根据镜像文件地址映射的复杂程度，通常有两种方法告知编译器相关的地址信息：（1）对于镜像文件中地址关系比较简单的情况可以使用命令行选项，或在 ARM 开发环境 ADS 或 RealView 中通过图形的方式来设置相应的值；（2）对于镜像文件中地址关系比较复杂的情况，可以使用一个 scatter 文件来指定相关的地址映射关系。

下面通过两个例子来介绍 scatter 文件的编写方法。

（1）要实现如图 4-2 描述的镜像加载运行关系，需要编写的 scatter 文件内容如下：

```
LOAD_ROM 0x0000 0x4000  //定义加载地址，0x0000 是开始地址，0x4000 是空间长度
{
//定义属性为 RO 域的运行地址，0x0000 是开始地址，0x4000 是空间长度
EXEC_ROM 0x0000 0x4000
{
    * (+RO)
}
//定义属性为 RW、ZI 域的运行地址，0x0000 是开始地址，0x4000 是空间长度
RAM 0x10000 0x8000
{
    * (+RW,+ZI)
}
}
```

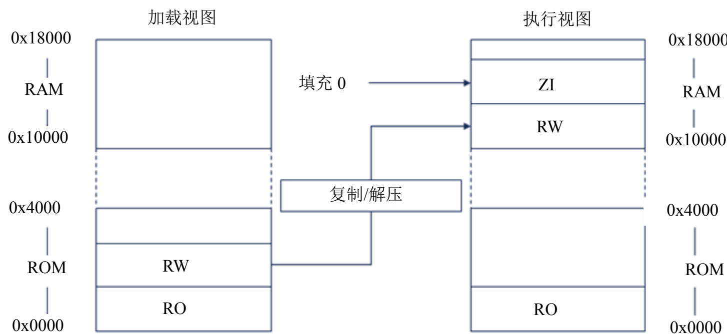

图 4-2 镜像加载运行内存分布图

（2）针对本书介绍的软件开发环境 RealView MDK 和基于 S3C2410 的硬件开发环境举例说明如何编写 scatter 文件。硬件开发平台的 NOR Flash 及 sdram 的空间如图 4-3 所示。

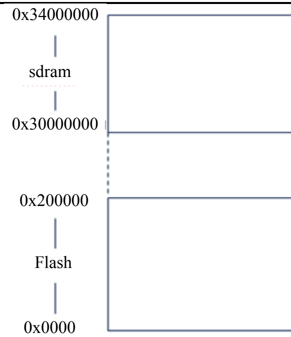

图 4-3 硬件开发平台内存分布图

Flash 存储空间是 0~2MB，sdram 的存储空间是 0x30000000~0x34000000。本书后面的接口实验会使用到两个 scatter 文件，一个是针对用仿真器调试时使用 RuninRAM.sct，另一个是针对将程序下载到 Flash 运行的 RuninFlash.sct 文件。

① RuninRAM.sct 的内容如下：

```
LR_ROM1 0x30000000
{    ; load region
     ER1   0x30000000 0x0300000
{  ; load address = execution address
          S3C2410A.o (RESET, +First)
               * (+RO)
     }
     RAM 0x30300000 0x3D00000
{  ; RW data
          * (+RW +ZI)
     }
}
```

② RuninFlash.sct 的内容如下：

```
LR1 0x00000000
{    ; load region
  ER1 0x00000000 0x200000
  {  ; load address = execution address
          S3C2410A.o (RESET, +FIRST)
          * (+RO)
  }
  RAM 0x30000000 0x4000000
  {  ; RW data
          * (+RW +ZI)
  }
}
```

如果读者想了解关于连接脚本文件更细节的知识，可以参考 RealView MDK 帮助文件中相关部分的内容。

4.4.5 汇编语言程序设计举例

1．条件执行的例子

通过组合使用条件执行和条件标志设置，可是简单地实现分支语句，不需要任何分支指令。这样不仅可以改善性能，因为分支指令会占用较多的周期数；同时也可以减小代码尺寸，提高代码密度。

下面是一段 C 语言程序，该程序实现了著名的 Euclid 最大公约数算法。

```
int gcd(int a, int b)
{
      while (a != b)
        {
           if (a > b)
                   a = a - b;
           else
```

```
              b = b - a;
        }
     return a;
}
```

用 ARM 汇编语言来重写这个例子。

【程序 1】

```
gcd      CMP      r0, r1
         BEQ      end
         BLT      less
         SUB      r0, r0, r1
         B        gcd
less
         SUB      r1, r1, r0
         B        gcd
End
```

充分利用条件执行修改上面的例子，得到【程序 2】。

【程序 2】

```
gcd
        CMP      r0, r1
        SUBGT    r0, r0, r1
        SUBLT    r1, r1, r0
        BNE      gcd
```

【程序 1】仅使用了分支指令，【程序 2】充分利用了 ARM 指令条件执行的特点，仅使用 4 条指令就完成了全部算法，这对提高程序的代码密度和执行速度十分有帮助。

事实上，分支指令十分影响处理器的速度。每次执行分支指令，处理器都会排空流水线，重新装载指令。

2．使用 LDM 和 STM 指令实现块复制

当程序中有大量数据需要复制或搬移时，常用到 LDM 和 STM 指令。虽然使用单寄存器数据传送指令 LDR 和 STR 也能实现同样的功能，但代码密度和执行效率要低于使用多寄存器传送指令。

下面通过两个例子说明使用单寄存器数据传送指令和使用多寄存器数据传送指令的区别。

```
    AREA Word, CODE, READONLY  ;给代码段起名为 Word
num EQU 20  ;num=20 表示将要复制的字的个数
    ENTRY             ;程序入口
call
start
    LDR r0, =src ;r0 = 源操作块起始地址
    LDR r1, =dst ;r1 = 目的操作块起始地址
    MOV r2, #num ;r2 = 将要复制的字的个数
wordcopy LDR r3, [r0], #4 ;从源操作块装载一个字
    STR r3, [r1], #4 ;存储到目的操作块
    SUBS r2, r2, #1   ;指针移到下一个要复制的字单元
    BNE wordcopy      ;循环复制
stop MOV r0, #0x18    ;angel_SWIreason_ReportException 为软中断调用准备参数
    LDR r1, =0x20026 ;调用 Semihosting 的 ADP_Stopped_ApplicationExit 功能
    SWI 0x123456      ;ARM semihosting SWI 调用
    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
```

下面是使用多寄存器数据复制命令 LDM 和 STM 重写的代码。

```
    AREA Block, CODE, READONLY      ;给代码段起名为 Block
num EQU 20  ;num=20 表示将要复制的字的个数
    ENTRY          ;程序入口
call
start
    LDR r0, =src ;r0 = 源操作块起始地址
    LDR r1, =dst ;r1 = 目的操作块起始地址
    MOV r2, #num ;r2 = 将要复制的字的个数
```

```
        MOV  sp, #0x400              ;设置堆栈指针 (r13)
    blockcopy     MOVS r3,r2, LSR #3;判断要移动的字的个数是 8 的几倍
        BEQ  copywords               ;如果移动的字的个数小于 8 则跳转到 copywords 子函数
        PUSH {r4-r11}                ;将用到的工作寄存器压栈保存
    octcopy  LDM  r0!, {r4-r11}      ;从源地址复制 8 个字
        STM  r1!, {r4-r11}           ;存到目的地址
        SUBS r3, r3, #1              ;计数器减 1
        BNE  octcopy                 ;如果计数器不等于零，继续复制
        POP  {r4-r11}                ;如果计数器等于零，恢复工作寄存器
    copywords     ANDS r2, r2, #7    ;判断要复制的剩余字个数
        BEQ  stop                    ;判断剩余字数是否为零
    wordcopy LDR  r3, [r0], #4       ;从源地址加载一个字
        STR  r3, [r1], #4            ;到目的地址
        SUBS r2, r2, #1              ;计算器减 1
        BNE  wordcopy                ;如果计数器不等于零，继续复制
    stop MOV  r0, #0x18              ;为软中断调用准备参数
        LDR  r1, =0x20026            ;调用 Semihosting 的 ADP_Stopped_ApplicationExit 功能
        SWI  0x123456                ;调用 Semihosting 软中断
        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
```

为了增加数据复制的效率，程序中使用了“MOVS r3,r2, LSR #3”指令。该指令将为下面以 8 字节为单位进行数据复制做准备。将数据以 8 个字为单位复制，是因为 ARM 体系结构中所能使用的寄存器为 8 个，即 r4～r11（根据 AAPCS 标准，r0～r3 在子程序调用过程中将被使用）。

4.5 汇编语言与 C 语言的混合编程

在 C 代码中实现汇编语言的方法有内联汇编和内嵌汇编两种，使用它们可以在 C 程序中实现 C 语言不能完成的一些工作。例如，在下面几种情况中必须使用内联汇编或嵌入型汇编。

（1）程序中使用饱和算术运算（Saturating Arithmetic），如 SSAT16 和 USAT16 指令。

（2）程序中需要对协处理器进行操作。

（3）在 C 程序中完成对程序状态寄存器的操作。

4.5.1 内联汇编

1. 内联汇编语法

内联汇编使用“asm”关键字声明，语法格式如下：

```
• __asm("instruction[;instruction]");   //必须为单条指令
  __asm{instruction[;instruction]}
• __asm{
  …
  instruction
  …
}
• asm("instruction[;instruction]");     //必须为单条指令
  asm{instruction[;instruction]}
• asm{
   …
   instruction
   …
}
```

内联汇编支持大部分的 ARM 指令，但不支持带状态转移的跳转指令，如 BX 和 BLX 指令，详细内容参见 ARM 相关文档。

由于内联汇编嵌入在 C 或 C++程序中，所以在用法上有其自身的一些特点。

（1）如果同一行中包含多条指令，则用分号隔开。

（2）如果一条指令不能在一行中完成，使用“/”将其连接。

（3）内联汇编中的注释语句可以使用 C 或 C++风格。

（4）汇编语言中使用逗号“,”作为指令操作数的分隔符，所以如果在 C 语言中使用逗号必须用圆括号括起来。例如，__asm {ADD x, y, (f(), z)}。

（5）内联汇编语言中的寄存器名被编译器视为 C 或 C++语言中的变量，所以，内联汇编中出现的寄存器名不一定和同名的物理寄存器相对应。这些寄存器名在使用前必须声明，否则编译器将提示警告信息。

（6）内联汇编中的寄存器（除程序状态寄存器 CPSR 和 SPSR 外）在读取前必须先赋值，否则编译器将产生错误信息。

下面的例子显示了内联汇编和真正汇编的区别。

错误的内联汇编函数如下：

```
int f(int x)
{
      __asm
      {
          STMFD sp!, {R0}      //保存 R0 不合法，因为在读之前没有对寄存器写操作
          ADD R0, x, 1
          EOR x, R0, x
          LDMFD sp!, {R0}      //不需要恢复寄存器
      }
      return x;
}
```

将其进行改写，使它符合内联汇编的语法规则。

```
int f(int x)
{
      int R0;
      __asm
      {
          ADD R0, x, 1
          EOR x, R0, x
      }
      return x;
}
```

2．内联汇编示例

下面通过几个例子进一步了解内联汇编的语法。

1）字符串的复制

下面的例子使用一个循环完成了字符串的复制工作。

```
#include <stdio.h>
void my_strcpy(const char *src, char *dst)
{
     int ch;
     __asm
     {
     loop:
         LDRB    ch, [src], #1
         STRB    ch, [dst], #1
         CMP     ch, #0
         BNE     loop
     }
}
int main(void)
{
     const char *a = "Hello world!";
     char b[20];
     my_strcpy (a, b);
```

```
    printf("Original string: '%s'\n", a);
    printf("Copied string: '%s'\n", b);
    return 0;
}
```

2）中断使能

下面的例子通过读取程序状态寄存器（CPSR）并设置它的中断使能位 bit[7]来禁止/打开中断。需要注意的是，该例只能运行在系统模式下，因为用户模式是无权修改程序状态寄存器的。

```
__inline void enable_IRQ(void)
{
     int tmp;
     __asm
     {
         MRS tmp, CPSR
         BIC tmp, tmp, #0x80
         MSR CPSR_c, tmp
     }
}
__inline void disable_IRQ(void)
{
     int tmp;
     __asm
     {
         MRS tmp, CPSR
         ORR tmp, tmp, #0x80
         MSR CPSR_c, tmp
     }
}
int main(void)
{
     disable_IRQ();
     enable_IRQ();
}
```

4.5.2 嵌入型汇编

利用 ARM 编译器可将汇编代码包括到一个或多个 C 函数定义中。嵌入式汇编器提供对目标处理器不受限制的低级别访问。有关为 ARM 处理器编写汇编语言的详细信息，请参阅 ADS 或 RealView 编译工具的汇编程序指南。

1. 嵌入式汇编语言语法

嵌入式汇编函数定义由 __asm 函数限定符标记，用__asm 声明的函数可以有调用参数和返回类型。它们从 C 中调用的方式与普通 C 函数调用方式相同。嵌入式汇编函数的语法如下：

```
__asm return-type function-name(parameter-list)
{
    //ARM/Thumb/Thumb－2 assembler code
    instruction[; instruction]
    …
    [instruction]
}
```

嵌入式汇编的初始执行状态是在编译程序时由编译选项决定的，这些编译选项如下：

（1）如果初始状态为 ARM 状态，则内嵌汇编器使用--arm 选项。

（2）如果初始状态为 Thumb 状态，则内嵌汇编器使用--thumb 选项。

在参数列表中允许使用参数名，但不能用在嵌入式汇编函数体内。例如，以下函数在函数体内使用整数 i，但在汇编中无效。

```
__asm int f(int i) {
    ADD i, i, #1 //编译器报错
}
```

可以使用 R0 代替 i。

下面通过嵌入式汇编的例子，来进一步熟悉嵌入式汇编的使用。

下面的例子实现了字符串的复制，注意和 4.5.1 节中内联汇编中字符串复制的例子相比较，分析其中的区别。

```
#include <stdio.h>
__asm void my_strcpy(const char *src, const char *dst) {
loop
      LDRB  R3, [R0], #1
      STRB  R3, [R1], #1
      CMP   R3, #0
      BNE   loop
      MOV   pc, lr
}
void main()
{
     const char *a = "Hello world!";
     char b[20];
     my_strcpy (a, b);
     printf("Original string: '%s'\n", a);
     printf("Copied   string: '%s'\n", b);
}
```

2．嵌入式汇编程序表达式和 C 表达式之间的差异

嵌入式汇编表达式和 C 表达式之间存在以下差异。

（1）汇编程序表达式总是无符号的。相同的表达式在汇编程序和 C 中有不同值。例如：

```
MOV R0, #(-33554432 / 2)        //结果为 0x7f000000
MOV R0, #__cpp(-33554432 / 2)  //结果为 0xff000000，CPP 指明的部分表示访问的是 C 表达
                                //式，参考本节稍后部分的“__CPP”部分
```

（2）以 0 开头的汇编程序编码仍是十进制的。例如：

```
MOV R0, #0700                   //十进制 700
MOV R0, #__cpp(0700)            //八进制 0700 等于十进制 448
```

（3）汇编程序运算符优先顺序与 C 不同。例如：

```
MOV r0, #(0x23 :AND: 0xf + 1)     //((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)    //(0x23 & (0xf + 1)) => 0
```

（4）汇编程序字符串不是以空字符为终止标志的。

```
DCB "no trailing null"                //16 Bytes
DCB __cpp("I have a trailing null!!")  //24 Bytes
```

3．关键字__cpp

可用__cpp 关键字从汇编代码中访问 C 的编译时的常量表达式，其中包括含有外部链接的数据或函数地址。

编译时，编译器将使用__cpp(expr) 的地方用汇编程序可以使用的常量所取代。例如：

```
LDR R0, =__cpp(&some_variable)
LDR R1, =__cpp(some_function)
BL  __cpp(some_function)
MOV R0, #__cpp(some_constant_expr)
```

4.5.3 汇编代码访问 C 全局变量

在汇编代码中访问 C 全局变量，只能通过地址间接访问全局变量。要访问全局变量，必须在汇编中使用 IMPORT 伪操作输入全局变量，然后将地址载入寄存器。可以根据变量的类型使用载入和存储指令访问该变量。

对于无符号变量，可以使用以下指令。

（1）LDRB/STRB：用于 char 型。

（2）LDRH/STRH：用于 short 型（对于 ARM 体系结构 V3，使用两个 LDRB/STRB 指令）。

（3）LDR/STR：用于 int 型。

对于有符号变量，请使用等效的有符号数的 Load/Store 指令，如 LDRSB 和 LDRSH。

对于少于 8 个字的小结构体，可以用 LDM 和 STM 指令将其作为整体访问。同时也可以用适当类型的 Load/Store 指令访问结构的单个成员。为了访问成员，必须了解该成员地址相对于结构体开始处的偏移量。

下面的例子将整型全局变量 globvar 的地址载入 R1，将该地址中包含的值载入 R0，将它与 2 相加，然后将新值存回 globvar 中。

```
    PRESERVE8
    AREA     globals,CODE,READONLY
    EXPORT    asmsubroutine
    IMPORT    globvar
    asmsubroutine
    LDR  R1, =globvar   ;从内存池中读取 globvar 变量的地址，加载到 R1 中
    LDR  R0, [R1]
    ADD  R0, R0, #2
    STR  R0, [R1]
    MOV  pc, lr
    END
```

4.5.4　混合编程调用举例

汇编程序、C 程序相互调用时，要特别注意遵守相应的 AAPCS。下面一些例子具体说明了在这些混合调用中应注意遵守的 AAPCS 规则。

1. C 程序调用汇编语言

下面的程序显示如何在 C 程序中调用汇编语言子程序，该段代码实现了将一个字符串复制到另一个字符串。

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{   const char *srcstr = "First string - source ";
    char dststr[] = "Second string - destination ";
     /*下面将 dststr 作为数组进行操作*/
    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return(0);
}
```

下面为调用的汇编程序。

```
    PRESERVE8
    AREA    SCopy, CODE, READONLY
    EXPORT strcopy
Strcopy                         ;R0 指向目的字符串
                                ;R1 指向源字符串
    LDRB R2, [R1],#1            ;加载字节并更新源字符串指针地址
    STRB R2, [R0],#1            ;存储字节并更新目的字符串指针地址
    CMP R2, #0                  ;判断是否为字符串结尾
    BNE strcopy                 ;如果不是，程序跳转到 strcopy 继续复制
    MOV pc,lr                   ;程序返回
    END
```

2. 汇编语言调用 C 程序

下面的例子显示了如何从汇编语言调用 C 程序。

下面的子程序段定义了 C 语言函数。

```
int g(int a, int b, int c, int d, int e)
```

```
{
    return a + b + c + d + e;
}
```

下面的程序段显示了汇编语言调用。假设程序进入 f 时，R0 中的值为 i。

```
;int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }
PRESERVE8
EXPORT f
AREA f, CODE, READONLY
IMPORT g                //声明 C 程序 g()
STR lr, [sp, #-4]!      //保存返回地址 lr
ADD R1, R0, R0          //计算 2*i(第 2 个参数)
ADD R2, R1, R0          //计算 3*i(第 3 个参数)
ADD R3, R1, R2          //计算 5*i
STR R3, [sp, #-4]!      //第 5 个参数通过堆栈传递
ADD R3, R1, R1          //计算 4*i(第 4 个参数)
BL g                    //调用 C 程序
ADD sp, sp, #4          //从堆栈中删除第 5 个参数
LDR pc, [sp], #4        //返回
END
```

4.6 本章小结

本章介绍了 ARM 程序设计的过程与方法，包括汇编语言编程、伪指令的使用、汇编器的使用、汇编和 C 混合编程等内容。这些内容是嵌入式编程的基础，希望读者掌握。

4.7 本章习题

1．在 ARM 汇编中如何定义一个全局的数字变量？
2．ADR 和 LDR 的用法有什么区别？
3．AAPCS 中规定的 ARM 寄存器的使用规则有哪些？
4．什么是内联汇编？什么是嵌入型汇编？两者之间的区别是什么？
5．在汇编代码中如何调用 C 代码中定义的函数？

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见