4　嵌入式程序设计基础

- 基于ARM的编译器一般都支持汇编语言的程序设计、C/C++语言的程序设计及两者的混合编程。
- 本章介绍ARM的嵌入式程序的基础知识
  - 伪指令
  - 汇编语言的语句格式
  - 汇编语言程序
  - 汇编语言与C/C++语言的混合编程

4.1 伪指令

- **伪指令：有一些特殊指令助记符，与指令系统的助记符不同，没有相对应的操作码，他们所完成的操作称为伪操作。**
- **伪指令在源程序中的作用是既要把正常的程序用指令表达给计算机以外，又要把程序设计者的意图表达给编译器。**

  例如：要告诉编译器程序段的开始和结束，需要定义数据等。

4.1 伪指令

- 在ARM的汇编程序中，我们把伪指令分为三部分介绍:
  - 通用伪指令
  - 与ARM指令相关的伪指令
  - 与Thumb指令相关的伪指令

4.1.1 通用伪指令

通用伪指令包括:

- **符号定义伪指令**
- **数据定义伪指令**
- **汇编控制伪指令**
- **及其他一些常用伪指令等。**

4.1.1 通用伪指令

1.符号定义伪指令

声明ARM汇编程序中的变量、对变量赋值以及定义寄存器的名称等操作。

常见的符号定义伪指令有如下几种：

（1）GBLA、GBLL和GBLS 声明全局变量的伪指令

语法格式：　　GBLA（GBLL或GBLS）　　全局变量名

定义一个ARM程序中的全局变量，并将其初始化。

其中：

- **GBLA用于声明一个全局的数字变量，并初始化为0；**
- **GBLL伪指令用于声明一个全局的逻辑变量，并初始化为F（假）；**
- **GBLS伪指令用于声明一个全局的字符串变量，并初始化为空；**
- **对于全局变量来说，变量名在源程序中必须是唯一的。**

4.1.1 通用伪指令

指令示例：

```
    GBLA   DATE1
;   声明一个全局数字变量DATE1
    GBLL   DATE2
;   声明一个全局逻辑变量DATE2
    GBLS   DATA3
;   声明一个全局的字符串变量DATE3
    DATE3  SETS“Testing”
;   将该变量赋值为“Testing”
```

4.1.1 通用伪指令

(2) LCLA、LCLL和LCLS 声明局部变量伪指令

语法格式：

LCLA（LCLL或LCLS） 局部变量名

定义一个ARM程序中的局部变量，并将其初始化。

其中：

- **LCLA用于声明一个局部的数字变量，并初始化为0；**
- **LCLL用于声明一个局部的逻辑变量，并初始化为F（假）；**
- **LCLS用于声明一个局部的字符串变量，并初始化为空。**
- **对于局部变量来说，变量名在使用的范围内必须是唯一的，范围限制在定义这个变量的宏指令程序段内。**

4.1.1 通用伪指令

指令示例：

```
    LCLA    DATE4
; 声明一个局部数字变量DATE4
    LCLL    DATE5
; 声明一个局部的逻辑变量DATE5
    DATA4   SETL 0x10
; 为变量DATE4赋值为0x10
    LCLS    DATA6
; 声明一个局部的字符串变量DATA6
```

4.1.1 通用伪指令

（3）SETA、SETL和SETS 变量赋值伪指令

语法格式：

变量名 SETA（SETL或SETS）表达式

给一个已经定义的全局变量或局部变量赋值。

其中：

- **SETA用于给一个数学变量赋值；**
- **SETL用于给一个逻辑变量赋值；**
- **SETS用于给一个字符串变量赋值；**

4.1.1 通用伪指令

指令示例：

```
    GBLA  EXAMP1
;  先声明一个全局数字变量EXAMP1
    EXAMP1  SETA    0xaa
;  将变量EXAMP1赋值为0xaa
    LCLL  EXAMP2
;  声明一个局部的逻辑变量EXAMP2
    EXAMP1      SETL      {TRUE}
;  将变量EXAMP1赋值为TRUE
    GBLA     EXAMP3
;  先声明一个全局字符串变量EXAMP3
     EXAMP3    SETS       “string”
;  将变量EXAMP3赋值为string
```

4.1.1 通用伪指令

(4) RLIST 定义通用寄存列表伪指令

语法格式： 名称 RLIST {寄存器列表}

- 应用在堆栈操作或多寄存器传送中，即使用该伪指令定义的名称可在ARM指令LDM/STM中使用。
- 在LDM/STM指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。

RegList RLIST {R0-R5，R8 }；定义寄存器列表为RegList

在程序中使用：

```
STMFD   SP!, RegList    ；存储列表到堆栈
LDMIA   R5, RegList     ；加载列表
```

4.1.1 通用伪指令

2. 数据定义伪指令

数据定义伪指令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。

(1) DCB 字节分配内存单元伪指令

语法格式： 标号 DCB 表达式

用来分配一片连续的字节存储单元并用伪指令中指定的数值或字符初始化。数值范围为0～255，DCB也可用“=”代替。

```
String  DCB  “This is a test！”    ；分配一片连续的字节存储单元并初始化。
DATA2 DCB  15, 25, 62, 00     ；为数字常量15，25，62，00分片内存单元
```

4.1.1 通用伪指令

(2) DCW（或DCWU） 半字分配内存单元

语法格式： 标号 DCW（或DCWU）表达式

- 表达式可以为程序标号或数字表达式。
- 伪指令DCW用于为半字分配一段半字对准的内存单元，并用指定的数据初始化；
- 伪指令DCWU用于为半字分配一段可以非半字对准的内存单元，并用指定的数据初始化。

DATA1 DCW 1, 2, 3；分配一片连续的半字存储单元并初始化为1，2，3。

DATA2 DCWU 45, 0x2a*0x2a；分配一片非半字对准存储单元并初始化。

4.1.1 通用伪指令

(3) DCD（或DCDU） 为字分配内存单元伪指令

语法格式：　　标号　DCD（或DCDU）　　表达式

- 表达式可以为程序标号或数字表达式。DCD也可用“&”代替。
- 伪指令DCD用来为字分配一段对准的内存单元，并用指定的数值或标号初始化；
- 伪指令DCDU用来为字分配一段可以非对准的内存单元，并用指定的数值或标号初始化。

DATA1 DCD 4，5，6；分配一片连续的字存储单元并初始化。

DATA2 DCDU LOOP；为LOOP标号的地址值分配一个内存单元。

4.1.1 通用伪指令

（4）DCFD（或DCFDU）和DCFS（或DCFSU）

语法格式： 标号 伪指令 表达式

- DCFD（或DCFDU）和DCFS（或DCFSU）都是为浮点数分配内存单元的伪指令。
- DCFD用于为双精度的浮点数分配一段字对准的内存单元，并用指定的数据初始化，每个双精度的浮点数占两个字单元；
- DCFDU用于为双精度的浮点数分配一段非字对准的内存单元，并用指定的数据初始化，每个双精度的浮点数占两个字单元；
- DCFS用于为单精度的浮点数分配一段字对准的内存单元，并用指定的数据初始化，每个单精度的浮点数占一个字单元；
- DCFSU用于为单精度的浮点数分配一段非字对准的内存单元，并用指定的数据初始化，每个单精度的浮点数占一个字单元。

4.1.1 通用伪指令

指令示例：

```
FLO1  DCFD   2E115，-5E7
；分配一段字对准存储单元并初始化为指定的双精度数为
  2E115，-5E7。
FLO2  DCFDU   22，1E2
；分配一段非字对准存储单元并初始化为指定的双精度数为
  22，1E2。
FLO3  DCFS   2E5，-5E - 7
；分配一段非字对准存储单元并初始化为指定的单精度数为
  2E5，-5E-7。
```

4.1.1 通用伪指令

（5）DCQ(或DCQU) 为双字分配内存单元的伪指令

语法格式： 标号 DCQ（或DCQU）表达式

- 伪指令DCQ用于为双字分配一段字对准的内存单元，并用指定的数据初始化；
- 伪指令DCQU用于为双字分配一段可以非字对准的内存单元，并用指定的数据初始化。

指令示例：

```
DATA1   DCQ     100
；分配一片连续的存储单元并初始化为指定的值。
```

4.1.1 通用伪指令

(6) MAP和FILED 内存表定义伪指令

语法格式： MAP 表达式，{基址寄存器}

标号 FIELD 表明数据字节数的数值

- 伪指令MAP用于定义一个结构化的内存表的首地址，MAP也可用“^”代替；
- 伪指令FIELD用于定义内存表中的数据的长度。FILED也可用“#”代替。

4.1.1 通用伪指令

MAP 表达式，{基址寄存器}

- 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，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。
- 注意：MAP和FIELD伪指令仅用于定义数据结构，并不实际分配存储单元。

```
MAP   0x10，R1  ；定义内存表首地址的值为[R1]+0x10。
DATA1   FIELD   4     ；为数据DATA1定义4字节长度
DATA2   FIELD   16    ；为数据DATA1定义16字节长度
```

4.1.1 通用伪指令

- FIELD +MAP 配合使用来定义结构化的内存表。
- MAP 伪指令定义内存表的首地址，
- FIELD 伪指令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指令引用。

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

4.1.1 通用伪指令

(7) SPACE 内存单元分配伪指令

语法格式： 标号 SPACE 分配的内存单元字节数

用于分配一片连续的存储区域并初始化为0，SPACE也可用“%”代替。

指令示例：

```
DATASPA  SPACE   100
                ；为DATASPA分配100个存储单元
                ；并初始化为0
```

4.1.1 通用伪指令

3. 汇编控制伪指令

用于控制汇编程序的执行流程

（1）MACRO、MEND和MEXIT

语法格式：

MACRO

定义一个宏语句段的开始；

$标号 宏名 $参数1，$参数2，……

语句段

MEXIT

从宏程序段的跳出

语句段

MEND

定义宏语句段的结束；

MACRO、MEND和MEXIT都是宏定义指令。

宏指令可以使用一个或多个参数，当宏指令被展开时，这些参数被相应的值替换。MACRO、MEND伪指令可以嵌套使用。

4.1.1 通用伪指令

- 宏是一段功能完整的程序，能够实现一个特定的功能，在使用中可以把它视为一个子程序。在其他程序中，可以调用宏完成某个功能。
- 调用宏是通过调用宏的名称来实现的。
- 宏指令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，可以使用宏指令代替子程序。
- 调用宏的好处是不占用传送参数的寄存器，不用保护现场。

4.1.1 通用伪指令

指令示例：

```
MACRO                    ；定义宏
$DATA1   MAX  $N1, $N2
                ；宏名称是MAX，主标号是$DATA1，两个参数
语句段                   ；语句段
$DATA1. MAY1     ；非主标号，由主标号构成
语句段                   ；语句段
……
$DATA1. MAY2     ；非主标号，由主标号构成
……
MEND                     ；宏结束
```

4.1.1 通用伪指令

(2) IF、ELSE、ENDIF 条件分支伪指令

语法格式：

IF 逻辑表达式

语句段1

ELSE

语句段2

ENDIF

- 当IF后面的逻辑表达式为真，则执行语句段1，否则执行语句段2。
- ELSE及语句段2可以没有，此时，当IF后面的逻辑表达式为真，则执行指令序列1，否则继续执行后面的指令。

IF、ELSE、ENDIF伪指令可以嵌套使用。

```
IF      R0=0x10      ；判断R0中的内容是否是0x10
ADD  R0, R1, R2   ；如果R0= 0x10，则执行R0= R1+ R2
ELSE
ADD  R0,R1,R3     ；如果R0≠ 0x10，则执行R0= R1+ R3
ENDIF
```

4.1.1 通用伪指令

(3) WHILE、WEND 条件循环伪指令

语法格式：

```
WHILE  逻辑表达式
       语句段
WEND
```

- 伪指令WHILE对条件进行判断，满足条件循环，不满足条件结束循环；伪指令WEND定义循环体结束。
- 当WHILE后面的逻辑表达式为真，则执行语句段，该语句段执行完毕后，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。

4.1.1 通用伪指令

指令示例：

```
GBLA    Cou1          ；声明一个全局的数学变量，变量名为Cou1
Cou1  SETA    1       ；为Cou1赋值1
WHILE  Cou1< 10       ；判断WHILE  Counter < 10进入循环
    ADD R1, R2, R3    ；循环执行语句
Cou1  SETA Cou1+1     ；每次循环Cou1加1
WEND
；执行ADD R1, R2, R3语句10次后，结束循环。
```

在应用WHILE、WEND伪指令时要注意：用来进行条件判断的逻辑表达式必须是编译程序能够判断的语句，一般应该是伪指令语句。

4.1.1 通用伪指令

4.其他杂类伪指令

(1) ALIGN 地址对准伪指令

语法格式：

ALIGN {表达式{，偏移量}}

- 可通过插入字节使存储区满足所要求的地址对准。
- 表达式的值用于指定对准方式，可能的取值为2n，0≤n≤31，如果不选表达式，则默认字对准；
- 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：2n + 偏移量。

4.1.1 通用伪指令

指令示例：

```
B  START
ADD  R0, R1, R2          ；正常语句
DATA1  DCB  “Ertai”
                ；由于插入5个字节的存储区，地址不对准
ALIGN 4
                ；指定后面的指令为 4 字节对齐
START  LDR  R5, [R6]
                ；否则此标号不对准
    ……
```

4.1.1 通用伪指令

(2) AREA 段指示伪指令

语法格式： AREA 段名 属性1，属性2，……

- **用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括起来，如|1_test|。**
- **属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。**

属性	含义	默认
CODE	定义代码段	READONLY
DATA	定义数据段	READWRITE
READONLY	指定本段为只读	READONLY
READWRITE	指定本段为可读可写	READWRITE

```
AREA    Example1，CODE，READONLY
```

；定义了一个代码段，段名为Example1，属性为只读。

4.1.1 通用伪指令

(3) CODE16、CODE32 代码长度定义伪指令

语法格式：

CODE16

CODE32

若在汇编源程序中同时包含ARM指令和Thumb指令时，可用CODE16伪指令定义后面的代码编译成16位的Thumb指令，CODE32伪指令定义后面的代码编译成32位的ARM指令。

4.1.1 通用伪指令

指令示例：

```
AREA      Example1，CODE，READONLY
……
CODE32                    ；定义后面的指令为32位的ARM指令
LDR  R0，=NEXT+1 ；将跳转地址放入寄存器R0
BX    R0
；程序跳转到新的位置执行，并将处理器切换到Thumb工作状态
……
CODE16              ；定义后面的指令为16位的Thumb指令
NEXT      LDR  R3，=0x3FF
……
END                       ；程序结束
```

4.1.1 通用伪指令

(4) ENTRY 程序入口伪指令

语法格式： ENTRY

- 在一个完整的汇编程序中至少要有一个ENTRY，编译程序在编译连接时依据程序入口进行连接。
- 在只有一个入口时，编译程序会把这个入口的地址定义为系统复位后的程序起始点。
- 在一个源文件里最多只能有一个ENTRY。

指令示例：

```
AREA  Example1，CODE，READONLY
ENTRY ；程序的入口处
……
```

4.1.1 通用伪指令

(5) END 编译结束伪指令

语法格式： END

- 用于通知编译器已经到了源程序的结尾，每个汇编语言的源程序都必须有一个END伪指令定义源程序结尾。
- 编译程序检测到这个伪指令后，不再对后面的程序编译。

指令示例：

```
AREA  Example1，CODE，READONLY
……
END                          ；程序结束
```

4.1.1 通用伪指令

(6) EQU 赋值伪指令

语法格式： 名称 EQU 表达式{，类型}

- 用于为程序中的常量、标号等定义一个等效的字符名称。
- 当表达式为32位的常量时，可以指定表达式的数据类型：CODE16、CODE32和DATA。

```
Test   EQU   50            ；定义标号Test的值为50
DATA1  EQU  0x55，CODE32    ；定义DATA1的值为0x55且该处为
   32位的ARM指令
abcd EQU label+16    ;定义abcd符号的值为(label+16)
```

4.1.1 通用伪指令

(7) GET和INCBIN 文件引用伪指令

语法格式：

GET 文件名

INCBIN 文件名

- 伪指令GET声明包含另一个源文件，并将被包含的源文件在当前位置进行汇编处理；
- 伪指令INCBIN声明包含另一个源文件，在INCBIN处引用这个文件但不汇编。

4.1.1 通用伪指令

指令示例：

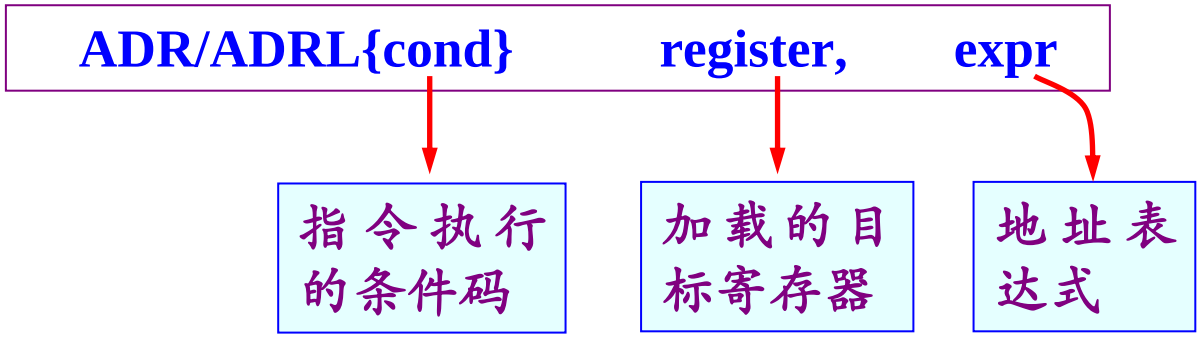
```
AREA  Example1，CODE，READONLY
GET   File1.s
              ；包含文件File1.s，并编译
INCBIN  File2.dat
              ；包含文件File2.s，不编译
……
GET   F:\EX\ File3.s
              ；包含文件File3.s，并编译
……
END
```

4.1.2 与ARM指令相关的伪指令

- 与ARM指令相关的伪指令共有4条。这4条伪指令和通用伪指令不同，在程序编译过程中，编译程序会为这4条指令产生代码，但这些代码不是它们自己的代码，所以尽管它们可以产生代码，但还是伪指令。
  - ADR伪指令
  - ADRL伪指令
  - LDR伪指令
  - NOP

4.1.2 与ARM指令相关的伪指令

ADR/ADRL伪指令格式 ——小/中等范围的地址读取



➢**功能：** 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读取到寄存器中。

✓**ADR：** 当地址值是字节对齐时，expr的取值范围为：-255~255；字对齐时，取值范围为-1020～1020；

✓**ADRL：** 当地址值是字节对齐时，expr的取值范围为：-64K~64K；字对齐时，取值范围为-256K～256K。

4.1.2 与ARM指令相关的伪指令

❖ ADR举例：

应用示例（源程序）：

```
...
ADR   R0,Delay
...
Delay
  MOV   R0,r14
...
```

使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

```
...
0x20  ADD   r0,pc,#0x3c
...
...
0x64  MOV   r0,r14
...
```

ADR伪指令被汇编成一条指令

4.1.2 与ARM指令相关的伪指令

❖ ADRL举例：

应用示例（源程序）：

```
    ...
    ADRL    R0,Delay
    ...
Delay
    MOV     R0,r14
    ...
```

使用伪指令将程序标号Delay的地址存入R0

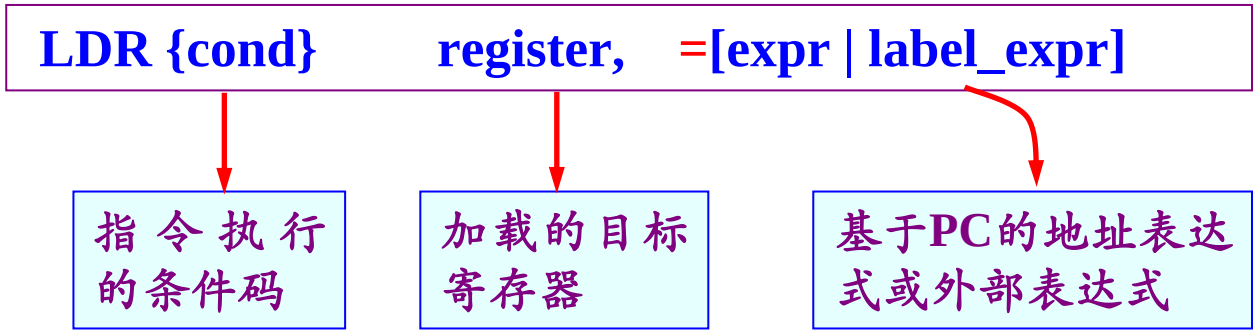
编译后的反汇编代码：

```
    ...
0x20  ADD     r0,pc,#40
0x24  ADD     r0,r0,#FF00
    ...
0xFF68 MOV    r0,r14
    ...
```

ADRL伪指令被汇编成两条指令

4.1.2 与ARM指令相关的伪指令

- LDR伪指令格式——加载32位立即数，或一个地址值到指定寄存器



注意：

- 从指令位置到文字池的偏移量必须小于4KB；
- 与ARM指令的LDR相比，伪指令的LDR的参数有=号。

4.1.2 与ARM指令相关的伪指令

❖ LDR伪指令举例：

```
LDR R2, =0xFF0              ;MOV R2, #0xFF0
LDR R1, =0xFFFFFFFE         ;MVN R1, #0x00000001
```

```
...
LDR    R1,=InitStack
...
InitStack
MOV    R0, LR
...
```

使用伪指令将程序标号InitStack的地址存入R1

4.1.2 与ARM指令相关的伪指令

- NOP伪指令格式——空操作

```
        MOV    R1,#0x1234
Delay
        NOP                    ;空操作，编译时被替换为类似
        NOP                    ; MOV R0, R0这样的无用指令
        NOP
        SUBS   R1,R1,#1        ;循环次数减一
        BNE    Delay           ;如果循环没结束，跳转Delay继续
        MOV    PC,LR           ;子程序返回
```

4.1.3 与Thumb指令相关的伪指令

与Thumb指令相关的伪指令共有3条，这些伪指令必须出现在Thumb程序段。

- ADR伪指令
- LDR伪指令
- NOP

4.1.3　与Thumb指令相关的伪指令

1. ADR伪指令　地址加载到低端寄存器中的伪指令

语法格式：

ADR{cond}　Rd，语句标号+数值表达式

；其中Rd是目标寄存器，为R0～R7。

- **这个地址必须是基于PC相对偏移的地址值。**
- **偏移必须向前偏移，偏移量不大于1KB，且该指令只可以加载字对准的地址。**

4.1.3 与Thumb指令相关的伪指令

指令示例：

```
        ADR R0, LOOP                ；把LOOP处绝对地址
                                    ;加载给R0
        ADR R1, LOOP+0x40*2         ；把LOOP+0x40*2处
                                    ;绝对地址加载给R1
        ……
    ALIGN
LOOP ADD R2, R0, R1
……
```

4.1.3 与Thumb指令相关的伪指令

2. LDR伪指令 把一个数字常量或一个地址加载到低端寄存器伪指令

语法格式：

LDR{cond} Rd, =数值表达式 ；加载数字常量

LDR{cond} Rd, =语句标号+数值表达式 ；加载地址

- **加载的是一个32位的数字常量，则编译程序就可以把这条语句编译成一条MOV指令；**
- **如果不能用MOV指令来表达，则编译成一条LDR指令。**
- **加载的是地址的话，编译程序会把这条语句编译成LDR指令。**

4.1.3 与Thumb指令相关的伪指令

在使用LDR指令替代伪指令时，编译程序先把数据（或地址）存放在数据缓冲区内，在执行LDR指令时，从缓冲区读出这个数据加载到寄存器中去。**是为程序创建数据缓冲区。**

指令示例：

```
LDR R1, =0xFFE          ；加载0xFFE到R1中
                        ；汇编器汇编成MOV R1,# 0xFFE
LDR R1, =START          ；加载START处的地址到R1中
```

4.1.3　与Thumb指令相关的伪指令

3. NOP

语法格式：

NOP

NOP伪指令是空操作指令，在汇编时将被编译成一条无效指令，如MOV　R0，R0，占用32位代码空间。

4.2 汇编语言的语句格式

汇编语言的源程序主要组成:

- **指令**
- **伪指令**
- **语句标号**
- **注释**

4.2.1 书写格式

ARM（Thumb）汇编语言的语句格式为：

{语句标号}{指令或伪指令}{；注释}

1.语句标号

可以大小写字母混合使用，可以使用数字和下划线。语句标号不能与指令助记符、寄存器、变量名同名。

2.指令和伪指令

可以大写，也可以小写，但不能大小写混合使用。

指令助记符和后面的操作数寄存器之间必须有空格，不可以在这之间使用逗号。

3. 注释

发现一个分号后，把后面的内容解释为注释，不予以编译。

4.2.1 书写格式

举例：

```
AREA  EXAMPLE1, CODE, READONLY

；EXAMPLE1程序段代码段，只读属性
```

4.2.2 数字表达式

数字表达式包括数字、数字常量、数字变量、数字运算符和括号构成。表达式的结果不能超过一个32位数的表达范围。

(1) 数字形式可以:

- 十进制
- 十六进制
- N进制
- ASCII

4.2.2 汇编语言中表达式和运算符

若是十进制，在表达的时候可以直接表达，

例如：1234、56789。

若是十六进制，有两种表达方法。

- 一种是在数值前加“0x”
- 另一种是在数值前加“&”。

例如：0x12A，&FF00。

4.2.2 汇编语言中表达式和运算符

- 若是N进制：N是一个2~9之间的整数。

表示方法是n_数值。

例如：

2_01101111 是一个二进制数字

8_54231067 是一个八进制数字。

- 若是ASCII表达：有些值可以使用ASCII表达。

例如： ‘A’表达A的ASCII码。

例如：MOV R1, #'A'等同于MOV R1, #0x41。

4.2.2 汇编语言中表达式和运算符

(2) 数字常量是一个32位的整数。

可以使用伪指令EQU定义一个数字常量，并且定义后不能改变。

(3) 数字变量是被定义变量的数字。

4. 2. 3　汇编程序基本结构

下面是一个汇编语言源程序的基本结构：

```
AREA  example, CODE, READONLY    ；定义代码块为example
ENTRY                            ；程序入口
Start
MOV R0,  #40                     ；R0=40
MOV R1, #16
ADD R2, R0, R1
MOV R0, #0x18
LDR R1, =0x20026
SWI 0x123456
END
```

4. 3 汇编语言与C/C++的混合编程

- ARM体系结构支持C/C + 以及与汇编语言的混合编程
- 在一个完整的程序设计的中，除了初始化部分用汇编语言完成以外，其主要的编程任务一般都用C/C++完成。

通常有以下几种方式：

- 在C/C + + 代码中嵌入汇编指令。
- 在汇编程序和C/C + + 的程序之间进行变量的互访。
- 汇编程序、C/C + + 程序间的相互调用。

4．4　汇编语言与C/C++的混合编程

遵守一定的调用规则：ATPCS规则

- **PCS即Procedure Call Standard（过程调用规范），ATPCS即ARM-THUMB procedure call standard。**
- **PCS规定了应用程序的函数可以如何分开地写，分开地编译，最后将它们连接在一起，所以它实际上定义了一套有关过程（函数）调用者与被调用者之间的协议。**

4．4　汇编语言与C/C++的混合编程

1.数据栈的使用规则

ATPCS规定数据栈为满递减类型。并对数据栈的操作是8字节对齐的。

（1）数据栈栈指针（stack pointer）：指向最后一个写入栈的数据的内存地址。

（2）数据栈的基地址（stack base）：数据栈的最高地址。

4. 4　汇编语言与C/C++的混合编程

2.参数的传递规则

- 参数不超过4个时，使用寄存器R0~R3来进行参数传递
- 参数超过4个时，还可以使用数据栈来传递参数。
- 依次将各名字数据传送到寄存器R0、R1、R2、R3；
- 如果参数多于4个，将剩余的字数据传送到数据栈中，入栈的顺序与参数顺序相反，即最后一个字数据先入栈。
- 局部变量用R4~R11保存。

4. 4　汇编语言与C/C++的混合编程

3.子程序结果返回规则

(1) 结果为一个32位的整数时,可以通过寄存器R0返回。

(2) 结果为一个64位整数时,可以通过R0和R1返回，依此类推。

(3) 对于位数更多的结果,需要通过调用内存来传递。

4. 4　汇编语言与C/C++的混合编程

在实际的编程应用：

- 程序的初始化部分用汇编语言完成，用C/C + + 完成主要的编程任务；
- 程序在执行时首先完成初始化过程，然后跳转到C/C + + 程序代码中，汇编程序和C/C + + 程序之间一般没有参数的传递，也没有频繁的相互调用；
- 整个程序的结构显得相对简单，容易理解。

4.3.1 在C/C++程序中内嵌汇编指令的语法格

- **ARM C语言程序：使用关键字_asm来标识一段汇编指令程序**

```
__asm
{   instruction [; instruction] 汇编语言程序段以及注释
        …
    [instruction]
}
```

- 如果一行中有多个汇编指令，指令之间使用分号“；”隔开；
- 如果一条指令占多行，使用续行符号”\”表示接续；
- 在汇编指令段中可以使用C语言的注释语句。

- **ARM C/C++程序：使用关键词asm来内嵌一段汇编程序**

Asm (“instruction [; instruction]”);

其中，asm后面括号中必须是一条汇编语句，且其不能包含注释语句。

4.3.2 C/C++与汇编语言的混合编程应用

在C中内嵌汇编语言：

```
#include <stdio.h>
void my_strcpy(const char *src，  char *dest)    //声明一个函数
{
    char ch;                                    //声明一个字符型变量
    __asm                                       //调用关键词__asm
    {
    LOOP                                  ；循环入口
        LDRB CH，[SRC]，#1                ；ch←src+1.将无符
                                          ；号src地址的数+1送入ch
        STRB CH，[dest]，#1               ；  [dest+1] ←ch，
                                          ；将无符号CH数据送入[dest+1]存储
        CMP  CH，  #0                     ；比较CH是否为零，否则循环。
                                          ；总共循环256次
        BNE  LOOP                         ；B 指令跳转，NE为Z位清零不相等
    }
}
```

4.3.2 C/C++与汇编语言的混合编程应用

2．在汇编中使用C程序全局变量

内嵌汇编不用单独编辑汇编语言文件，比较简洁，但是有诸多限制，当汇编的代码较多时一般放在单独的汇编文件中。这时就需要在汇编和C之间进行一些数据的传递，最简便的办法就是使用全局变量。方法如下：

(1) 使用IMPORT伪操作声明该全局变量。

(2) 使用LDR指令读取该全局变量的内存地址，通常该全局变量的内存地址值存放在程序的数据缓冲池中（Literal pool）。

(3) 根据该数据的类型，使用相应的LDR/STR指令读取/修改该全局变量的值。

4.3.2 C/C++与汇编语言的混合编程应用

在汇编程序中访问C程序全局变量。

```
AREA  asmfile,CODE,READONLY      ；建立一个汇编程序段
EXPORT asmDouble          ；声明可以被调用的汇编函数asmDouble
IMPORT gVar_1             ；调用C语言中声明的全局变量
asmDouble                 ；汇编子函数入口
LDR R0,=gVar_1            ；将等于gVar_1地址的数据送入R0寄存器
LDR R1,[R0]               ；将R0中的值为地址的数据送给R1。
MOV R2, #10               ；将立即数2送给R2
ADD R3, R1, R2            ；R3=R1+R2，实现了gVar_1= gVar_1+10
STR R3,[R0]               ；将R3中的数据送给R0
MOV PC, LR                ；子程序返回
END
```

4.3.2 C/C++与汇编语言的混合编程应用

3．C程序中调用汇编的函数

- **在C中声明函数原型，并加extern关键字；**
- **在汇编中用EXPORT导出函数名，并用该函数名作为汇编代码段的标识，最后用MOV PC，LR返回。**

C程序调用汇编程序：

汇编程序strcpy实现字符串复制功能，C程序调用strcpy完成字符串复制的工作。

4.3.2 C/C++与汇编语言的混合编程应用

```
/* C程序*/
#include <stdio.h>
extern void asm_strcpy(const char *src，  char *dest);
//声明可以被调用的函数
int main()                                          //C语言主函数
{
   const char *s = "seasons in the sun";    //声明字符型指针变量
   char d[32];                                      //声明字符型数组
   asm_strcpy(s,d);                                 //调用汇编子函数
   printf("source: %s",s);                          //屏幕显示，S的值
   printf(" destination: %s",d);                    //屏幕显示，d的值。
   return 0;
}
```

4.3.2 C/C++与汇编语言的混合编程应用

```
; 汇编语言程序段
AREA  asmfile,CODE,READONLY    ； 声明汇编语言程序段
EXPORT asm_strcpy              ； 声明可被调用函数名称
asm_strcpy                     ； 函数入口地址
LOOP                           ； 循环标志条
LDRB R4， [R0]， #1            ； R0的地址加1后送给R4
CMP R4， #0                    ； 比较R4是否为零
BEQ OVER                       ； 为零跳转到结束
STRB R4， [R1]， #1            ； R4的值送入R1加1地址
B LOOP                         ； 跳转到循环位置
OVER                           ； 跳出标志位
MOV PC， LR                    ； 子函数返回
END
```

4.3.2 C/C++与汇编语言的混合编程应用

4．在汇编程序中调用C的函数

在汇编中使用伪指令IMPORT 声明将要调用的C函数。

汇编程序调用C程序的例子：

在汇编程序中设置好各参数的值，本例有5个参数，分别使用寄存器R0存放第1个参数，R1存放第2个参数，R2存放第3个参数。

4.3.2 C/C++与汇编语言的混合编程应用

```
EXPORT  asmfile                          ;可被调用的汇编段
AREA  asmfile,CODE,READONLY   ;声明汇编程序段
IMPORT  cFun                             ;声明调用C语言的cFun函数
ENTRY                                    ;主程序起始入口
MOV  R0，  #11                           ;将11放入R0
MOV  R1，  #22                           ;将22放入R1
MOV  R2，  #33                           ;将33放入R2
BL  cFun;调用C语言子函数
END
/*C 语言函数，  被汇编语言调用 */
int cFun(int a，  int b，  int c)          //声明一个函数
{
   return a + b + c;                     //返回a+b+c的值
}
```

4.3.2 C/C++与汇编语言的混合编程应用

5．C++嵌入式系统中应用

C++和C是可以互相调用的，并且可以灵活的进行汇编语言、C语言、C++语言的混合调用。

使用伪指令“extern “C”{….}”

extern “C”{include “cHeadfile.h”}。

extern "C"包含双重含义，其一：被它修饰的目标是“extern”的；其二：被它修饰的目标是“C”的。

（1）被extern “C”限定的函数或变量是extern类型的

extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在本模块或其他模块中使用。

4.3.2 C/C++与汇编语言的混合编程应用

- extern int a；此语句仅仅是在声明一个变量，并不是定义变量a，并未为a分配内存空间。变量a在所有模块中作为一种全局变量只能被定义一次，否则会出现连接错误。
- 在模块的头文件中对模块提供给其他模块引用的函数和全局变量以关键字extern声明。
- 如果模块B欲引用该模块A中定义的全局变量和函数时只需包含模块A的头文件即可。这样，模块B中调用模块A中的函数时，在编译阶段，模块B虽然找不到该函数，但是并不会报错，它会在连接阶段中从模块A编译生成的目标代码中找到此函数。

4.3.2　C/C++与汇编语言的混合编程应用

与extern对应的关键字是static，被它修饰的全局变量和函数只能在本模块中使用。

（2）被extern "C"修饰的变量和函数是按照C语言方式编译和连接的。

作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。函数被C++编译后在符号库中的名字与C语言的不同。

4.3.2 C/C++与汇编语言的混合编程应用

例如:某个函数的原型为： void foo(int x, int y);

- C编译器编译后在符号库中的名字为_foo，
- C++编译器: _foo_int_int之类的名字（不同的编译器可能产生的名字不同，但是都采用了相同的机制）。
- 包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。

例如:在C++中，函数void foo(int x, int y)/void foo(int x, float y)编译产生的符号是不相同的，后者为_foo_int_float。

4.3.2 C/C++与汇编语言的混合编程应用

（3）extern "C"的惯用法

①在C++中引用C语言中的函数和变量，在包含C语言头文件（假设为cExample.h）时，需进行下列处理：

```
extern "C"
{
#include "cExample.h"
}
```

在C语言的头文件中，对其外部函数只能指定为extern类型，C语言中不支持extern "C"声明，在.c文件中包含了extern "C"时会出现编译语法错误。

4.3.2 C/C++与汇编语言的混合编程应用

②在C中引用C++语言中的函数和变量时，C++的头文件需添加extern "C"，但是在C语言中不能直接引用声明了extern "C"的头文件，应该仅将C文件中将C++中定义的extern "C"函数声明为extern类型。

4.3.2 C/C++与汇编语言的混合编程应用

例如：

```
//C++头文件 cppExample.h
#ifndef CPP_EXAMPLE_H
#define CPP_EXAMPLE_H
extern "C" int add(int x, int y);
#endif
```

```
//C++实现文件 cppExample.cpp
#include "cppExample.h"
int add(int x, int y)
{
    return x+y;
}
```

4.3.2 C/C++与汇编语言的混合编程应用

```
/*C实现文件cFile.c
/*这样会编译出错：#include "cppExample.h"*/
extern int add(int x, int y);
int main(int argc, char *argv[])      //C 语言主程序入口
{
    add(2,3);
    return 0;
}
```

4.5 本章小结

- ARM汇编语言程序设计中常见的通用伪指令、与ARM指令相关的伪指令、与Thumb指令相关的伪指令
- 汇编语言的基本语句格式和基本结构等，
- C/C + + 和汇编语言的混合编程的语法格式及应用等。
- 通过本章的学习，要求基本掌握伪指令、表达式和运算符的含义及用法，能够编写出汇编语言程序及掌握与C/C++的混合编程方法。