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。

《嵌入式系统技术与设计》

作者：华清远见

专业始于专注 卓识源于远见

第 3 章 ARM 汇编语言程序设计

本章目标

在第 2 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，本章将介绍相关的编程方法。

本章主要内容：

- ARM/Thumb 混合编程
- 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/C++的混合编程

专业始于专注 卓识源于远见

3.1 ARM/Thumb 混合编程

3.1.1 Thumb 指令的特点及实现

Thumb 指令集把 32 位 ARM 指令集的一个子集编码为一个 16 位的指令集。在 16 位外部数据总线宽度下，ARM 处理器上使用 Thumb 指令的性能要比使用 ARM 指令的性能更好；而在 32 位外部数据总线宽度下，使用 Thumb 指令的性能要比使用 ARM 指令的性能差。因此，Thumb 指令多用于存储器受限的一些系统中。Thumb 指令集并没有改变 ARM 系统底层的程序设计模型，只是在该模型上增加了一些限制条件。Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令寻址地址也是 32 位的。

代码密度高是 Thumb 指令集的一个主要优势。对于同一个程序而言，使用 Thumb 指令实现所需的存储空间，要比等效的 ARM 指令实现少 30%左右。例 3-1 和例 3-2，介绍的是使用 ARM 指令和 Thumb 指令实现相同的除法操作。从例子中可以看出，虽然 Thumb 指令的实现使用了更多的指令，但是它占用的总的存储空间却比较小。

【例 3-1】 使用 ARM 指令实现除法运算。

```
        MOV  R3，＃0
loop
        SUB  R0，R0，R1
        ADDGE  R3，R3，＃1
        BGE  loop
        ADD  R2，R0，R1
```

R0 存放被除数，R1 存放除数，R2 和 R3 分别存放余数和商。完成整个除法运算使用了 5 条指令，每一条指令所占的字节数为 4，所以实现一个除法运算，ARM 指令所占有的字节数为 20。

【例 3-2】 使用 Thumb 指令实现除法运算。

```
        MOV  R3，＃0
loop
        ADD  R3，＃1
        SUB  R0，R1
        BGE  loop
        SUB  R3，＃1
        ADD  R2，R0，R1
```

例 3-2 使用 Thumb 指令完成了和例 3-1 完全相同的功能。Thumb 指令虽然使用了 6 条指令，但其每条指令占用 2 个字节，所以总的字节数为 6×2=12，小于 ARM 指令所占用的 20 个字节。

Thumb 指令是 ARM 指令的一个受限子集。在 Thumb 状态下，不能直接访问所有的处理器寄存器，只有 R0～R7 是可以被任意访问的。在 Thumb 状态下，使用该 8 个寄存器和在 ARM 状态下使用没有区别。寄存器 R8～R12 只能通过 MOV、ADD 或 CMP 指令访问。CMP 指令和所有操作 R0～R7 的数据处理指令都会影响 CPSR 中的条件标志位。一些 Thumb 指令还使用到了程序计数器 PC（R15）、链接地址寄存器 LR（R14）和堆栈指针寄存器 SP（R13）。在 Thumb 状态下，读取 R15 寄存器时，bit[0]值为 0，bits [31:1]包含了 PC 的值。当对 R15 进行写入时，bit[0]被忽略，bits[31:1]被设置成当前程序计数器的值。

表 3-1 列出了在 Thumb 状态下，各个寄存器的使用情况。

表 3-1 Thumb 寄存器的使用

寄 存 器	访 问

R0～R7	完全访问
R8～R12	只能通过 MOV、ADD 及 CMP 访问
R13	限制访问
R14	限制访问
R15	限制访问
CPSR	间接访问
SPSR	不能访问

从表 3-1 可以看出，在 Thumb 状态下不能直接访问 CPSR 和 SPSR。也就是没有和 MSR 和 MRS 等价的指令。为了改变 CPSR 和 SPSR 的值，必须使处理器状态切换到 ARM 状态，再使用指令 MSR 和 MRS 来实现。同样，在 Thumb 状态下也没有协处理器访问指令，要访问协处理器寄存器来配置 Cache 和进行内存管理，也必须使处理器切换到 ARM 状态。

3.1.2 ARM/Thumb 交互工作基础

Thumb 以其较高的代码密度和在窄存储器上的性能，使得它在很多系统中得到广泛应用。但在很多情况下，必须使用 ARM 指令，主要是由于下列原因。

① ARM 代码比 Thumb 代码有更快的执行速度。

② ARM 处理器的一些特定功能必须由 ARM 指令实现，例如，PSR 指令、协处理器指令。

③ 异常发生时，处理器自动进入 ARM 状态，如果异常处理程序需要使用 Thumb 指令也必须通用一个 ARM 程序头（ARM assembler header）。

基于以上原因，即使程序需要由 Thumb 代码实现，也必须通过 ARM-Thumb 互交（ARM-Thumb interworking）进入 Thumb 状态。

ARM-Thumb 互交是指对汇编语言和 C/C++语言的 ARM 和 Thumb 代码进行连接的方法，它进行两种状态(ARM 和 Thumb 状态)间的切换。在进行这种切换时，有时需使用额外的代码，这些代码被称为 Veneer。AAPCS 定义了 ARM 和 Thumb 过程调用的标准。

从一个 ARM 例程调用一个 Thumb 例程，内核必须进行状态切换。状态的变化由 CPSR 的 T 位来显示。跳转到一个例程时，BX 指令可用于 ARM 和 Thumb 状态切换，具体用法如下。

在 Thumb 状态调用 ARM 例程时，采用：

```
BX  Rn
```

在 ARM 状态调用 Thumb 例程时，采用：

```
BX{cond}  Rn
```

其中，Rn 可以是 R0～R15 中的任意寄存器。

这种带状态切换的跳转指令 BX，将寄存器 Rn 的内容复制到程序计数寄存器 PC 中，因此可以实现 4G 空间的跳转。指令根据寄存器 Rn 的 bit[0]来决定处理器是否进行状态切换，详细内容参见 ARM 指令一节。

下面是一段 ARM 程序，该程序调用虚拟的 SWI_writeC 子程序从存储器的固定地址取出字符串“hello world”并输出。

```
        AREA    Hello,CODE,READONLY
SWI_WriteC       EQU     &0          ;软中断调用参数
SWI_Exit         EQU     &11         ;程序退出软中断调用参数
        ENTRY
START    ADR    R1,TEXT                  ;取字符串地址
LOOP    LDRB    R0,[R1],#1               ;取下一字节内容
        CMP  R0,#0                       ;判断是否为字符串尾
        SWINE  SWI_WriteC               ;软中断调用打印字符
        BEN  LOOP                       ;循环
        SWI  SWI_Exit                   ;软中断调用退出程序执行
```

```
TEXT     =      “Hello World” ,&0a,&0d,0
         END
```

下面的代码将上面的 ARM 代码转换成等价的 Thumb 代码。

```
         AREA  HelloW_Thumb,CODE,READONLY
SWI_WriteC  EQU    &0                  ;软中断调用参数
SWI_Exit  EQU    &11                   ;程序退出软中断调用参数
         ENTRY                         ;程序入口点
         CODE32                        ; 进入 ARM 状态
         ADR R0, START+1               ;取得 Thumb 代码入口地址
         BX R0                         ;进入 Thumb 代码
         CODE16                        ;Thumb 代码入口点
START    ADR  R1, TEXT                     ;R1 -> "Hello World"
LOOP     LDRB  R0, [R1]                ;取下一字节内容
         ADD  R1, R1, #1               ;地址指针加 1  **T
         CMP R0, #0                 ;判断是否为字符串尾
         BEQ DONE                   ;完成? **T
         SWI SWI_WriteC             ;如果不是字符串尾
         B LOOP                     ;继续循环
DONE     SWI  SWI_Exit                 ;程序退出
         ALIGN                      ;字对齐
TEXT     DATA
              "Hello World",&0a,&0d,&00
         END
```

上例中，ARM 代码到 Thumb 代码转换过程中新增加的指令用“**T”标注。

在实现 ARM 代码和 Thumb 代码转换时，大部分的 ARM 指令有等价的 Thumb 指令，只有少数指令没有。如加载字节指令（LDR）不支持自动变址，软中断指令不能条件执行。

在编写 Thumb 代码时要注意以下几点。

① 汇编器需要知道什么时候产生 ARM 代码，什么时候产生 Thumb 代码，程序中使用 CODE32 和 CODE16 伪操作提供给编译器这些信息。

② 由于处理器上的执行是在 ARM 状态下完成的，所以，要使用 Thumb 指令必须由 ARM 指令调用 Thumb 指令，这一过程是通过“BX LR”指令来实现的。需要注意的是，在使用“BX LR”指令之前，要对寄存器 LR 做正确的初始化。

③ 在 ARM 和 Thumb 混合编程时，常使用 ALIGN 伪操作保证内存地址对齐。

3.1.3 ARM/Thumb 交互子程序

编写 ARM/Thumb 互交代码时，需要注意下面两点。

① 对于 C/C++子程序而言，只要在编译时指定--apcs/interwork 选项，汇编器会生成合适的返回代码，使得程序返回到和调用程序相同的状态。

② 在汇编语言子程序中，用户必须自己编写相应的返回代码，使得程序返回到和调用程序相同的状态。

如果目标代码包含以下内容，应该在编译或汇编时使用--apcs/interwork 选项，使处理器能够在 ARM 和 Thumb 代码间进行正确的切换，这种情况包含以下 4 种。

① 需要返回到 ARM 状态的 Thumb 子程序。

② 需要返回到 Thumb 状态的 ARM 子程序。

③ 间接调用 ARM 子程序的 Thumb 子程序。

④ 间接调用 Thumb 子程序的 ARM 子程序。

如果在程序连接阶段，连接器发现 ARM 子程序和 Thumb 子程序间存在相互调用，而源文件在编译时没有使用--apcs/interwork 选项，则连接器将报告以下错误。

```
Error: L6239E: Cannot call ARM symbol 'arm_function' in non-interworking object
armsub.o from THUMB code in thumbmain.o(.text)
```

其中，“arm_function”是需要进行状态切换的子程序名。

在这种情况下，用户必须使用--apcs/interwork 选项重新对源文件进行编译。但在下面两种情况下，不必指定--apcs/interwork 选项。

① 在 Thumb 状态下，发生异常中断时，处理器自动切换到 ARM 状态，这时不需要添加状态切换代码。

② 当异常发生在 Thumb 状态时，从异常返回不需要添加状态切换的 Veneer 代码。

➢ 使用汇编语言实现互交

对于汇编程序来说，可以有两种方法来实现程序状态的切换。第一种方法是利用连接器提供的交互子程序 Veneer 来实现程序状态的切换，这时用户可以使用指令 BL 来调用子程序；第二种方法是用户自己编写状态切换的程序，本节主要介绍第二种方法。

在 ARMv4 版本及其以前的版本中，可以使用 BX 指令实现程序状态的切换。

从 ARMv5 版本开始，下面的指令也可以用来实现程序的状态切换。

- BX（Branch and eXchange）
- BLX、LDR、LDM 和 POP

下面简单介绍用于状态切换的指令和伪操作。

（1）BX 指令

ARM 状态下的 BX 指令，使程序跳转到指令中指定的参数 Rm 所指定的地址执行程序，Rm 的第 0 位拷贝到 CPSR 中的 T 位，bits[31：1]移入 PC。若 Rm 的 bit[0]为 1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位 bit[0]为 0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。

指令的语法格式如下：

```
BX{<cond>}  <Rm>
```

① <cond>

cond 为指令编码中的条件域。它指示指令在什么条件下执行。当 cond 忽略时，指令为无条件执行（cond = AL（Alway））。

② <Rm>

Rm 包含跳转指令的目标地址。如果 Rm 的 bit[0] = 0，目标地址处指令为 ARM 指令；如果 Rm 的 bit[0] = 1，目标地址处指令为 Thumb 指令。

Thumb 状态下的 BX 指令，也用于 ARM 和 Thumb 代码间的相互调用。

指令的语法格式如下：

```
BX  <Rm>
```

其中，<Rm>为目标地址寄存器，包含程序的跳转地址。BX 指令的目标地址寄存器可以是 R0～R15 中的任意寄存器。

ARM 指令集中的 BX 指令和 Thumb 指令集中的 BX 指令相差较大，它们分别为不同方向的跳转。当 R15 作为目的寄存器使用时，要特别注意该指令在两个指令集中的区别。

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令。

下面通过一个实例，说明 ARM 和 Thumb 之间的状态切换过程。

（2）编程实例

```
        PRESERVE8
        AREA    AddReg,CODE,READONLY        ;段名为 AddReg，属性为 READONLY
        ENTRY                               ;程序入口
; SECTION 1
main
        ADR R0, ThumbProg + 1               ;确定跳转地址
                                            ;并将 bit[0]置 1
                                            ;使程序切换到 Thumb 状态
        BX  R0                              ;程序跳转并执行状态切换
; SECTION 2
        CODE16                              ;Thumb 代码指示伪操作
ThumbProg
        MOV R2, #2                          ;R2 = 2
        MOV R3, #3                          ;R2 = 3
        ADD R2, R2, R3                      ;R2 = R2 + R3
        ADR R0, ARMProg
        BX  R0                              ;程序跳转并执行状态切换
; SECTION 3
        CODE32                              ;ARM 代码指示伪操作
ARMProg
    MOV R4, #4
    MOV R5, #5
    ADD R4, R4, R5
; SECTION 4
stop MOV R0, #0x18                          ;设置 semihosting 软中断号
        LDR R1, =0x20026                        ;ADP_Stopped_ApplicationExit
        SWI 0x123456                            ;ARM semihosting SWI 软中断调用
        END                                 ;文件结束
```

上面的例子分为 4 部分，通过下面的步骤编译和运行。

① 使用文本编辑器，如 notepad，输入上面的代码，并保存成文件 addreg.s。

② 在命令行中键入汇编命令 armasm –g addreg.s。

③ 在命令行中键入链接命令 armlink addreg.o -o addreg。

④ 使用调试器（Debugger）（如 RealView Debuggeror AXD）运行映像文件。可以使用单步执行，观察代码在 Thumb 状态下的执行。

Thumb 代码的地址标号如果用伪操作 export 声明为“外部的”，则连接器会自动调整该地址标号使其 bit[0]等于 1；如果该地址标号没有被声明为“外部的”，则使用者必须手动地对标号进行调整，如上例中的 ThumProg+1。

（3）ARMv5 架构下的状态切换

在 ARMv5 体系结构的指令集中，增加了下面两条指令用于 ARM 代码和 Thumb 代码之间的互交。

① BLX address

该指令跳转到指令中指定的地址处执行程序并进行程序状态切换，该地址是“PC 相关的”，地址范围为−32～32MB（ARM 状态）或−4～4MB（Thumb 状态）。

② BLX register

在该格式的跳转指令中，寄存器 Rm 指定转移目标，Rm 的第 0 位拷贝到 CPSR 中的 T 位，bits[31:0]移入 PC。

使用上面两条指令，在执行程序跳转之前，处理器自动将返回连接寄存器 LR 的 bit[0]位更新为 CPSR 寄存器的 T 位，所以，无论处理器状态是否发生变化，程序都能正确返回。

当使用 LDR、LDM 及 POP 指令向 PC 寄存器中赋值时，寄存器 CPSR 中的 Thumb 位将被设置成 PC 寄存器的 bit[0]，这时就实现了程序状态的切换。这种方法在子程序的返回时非常有效，同样的指令可以根据需要返回到 ARM 状态或 Thumb 状态。

连接器在对目标代码进行链接时，将代码中的地址标号分为 3 类。

① ARM 指令地址标号。

② Thumb 指令地址标号。

③ 数据（Data）地址标号。

当连接器重定位 Thumb 代码中的地址标号时，地址标号的 bit[0]位将被自动设置为 1。这就意味着跳转指令（包括 BX、BLX 和 LDR）可以根据目标地址正确地进行状态切换。

上面提到的连接器自动设置目的地址的行为 ,只有在 ARMv5 及其以上版本中支持。

➢ 使用 C 和 C++语言实现互交

对于不同的 C 和 C++源程序，可能有些程序中包含 ARM 指令，有些程序中包含 Thumb 指令，这些程序可以相互调用，只是在编译这些程序时指定--apcs/interwork 选项。当使用了--apcs/interwork 选项，编译器会自动进行一些相应处理；连接器在检测到程序中存在互交工作时，会生成一些用于程序状态切换的代码。

（1）代码编译

可以使用下面的指令，将 C 或 C++程序编译为可以执行互交的目标代码。

```
armcc --c90 --thumb --apcs /interwork
armcc --c90 --arm --apcs /interwork
armcc --cpp --thumb --apcs /interwork
armcc --cpp --arm --apcs /interwork
```

使用--apcs/interwork 选项对文件进行编译时，编译器会进行如下处理。

① 对于叶子程序（Leaf Function，即程序中没有其他子程序调用的程序），编译器将程序中的“MOV PC，LR”指令替换成“BX　LR”指令，因为“MOV　PC”指令不能进行状态切换。

② 对于非叶子程序，要进行一系列的指令替换，如：

```
POP  {R4,R5,pc}
```

替换为：

```
  POP  {R4,R5}
  POP  {R3}
  BX   R3
```

下面的例子显示了一段带子程序调用的 C 语言程序，使用--apcs/interwork 选项进行编译时，对代码产生的影响。

C 语言源程序。

```
Void  func(void)
{
…
Sub()
…
}
```

使用 armcc --apcs/interwork 选项进行编译产生结果如下。

```
Func
STMFD  sp!,{R4-R7,lr}
…
BL  sub
…
LDMFD  sp!, {R4-R7,lr}
BX  lr
```

使用 tcc --apcs/interwork 选项进行编译产生结果如下。

```
PUSH  {R4-R7,lr}
…
BL  sub
…
POP  {R4-R7}
POP  {R3}
BX
```

（2）C 语言的互交实例

下面的例子显示了一个 Thumb 状态下的代码通过互交调用 ARM 子程序，然后又在 ARM 子程序中调用 Thumb 指令集的库函数 printf()。

```
/*********************
*      thumbmain.c  *
**********************/
#include <stdio.h>
extern void arm_function(void);
int main(void)
{
      printf("Hello from Thumb World\n");
      arm_function();
      printf("And goodbye from Thumb World\n");
      return (0);
}
/*********************
*          armsub.c    *
**********************/
#include <stdio.h>
void arm_function(void)
{
      printf("Hello and Goodbye from ARM world\n");
}
```

使用下面的命令对程序进行编译连接。

① 编译生成带互交的 Thumb 代码。

```
armcc --thumb -c -g --apcs /interwork -o thumbmain.o thumbmain.c
```

② 编译生成带互交的 ARM 代码。

```
armcc -c -g --apcs /interwork -o armsub.o armsub.c
```

③ 连接目标文件。

```
armlink thumbmain.o armsub.o -o thumbtoarm.axf
```

另外，可以使用--info 选项使连接器输出由于互交所增加的代码大小。

```
armlink armsub.o thumbmain.o -o thumbtoarm.axf --info veneers
```

输出信息如下所示。

```
Adding Veneers to the image
    Adding TA veneer(4 bytes, Inline) for call to'arm_function'from thumbmain.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__0printf' from armsub.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__rt_lib_init' from kernel.o(.text).
    Adding AT veneer (12 bytes,Long) for call to'__rt_lib_shutdown'from kernel.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_memclr_w' from stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_raise' from stdio.o(.text).
    Adding TA veneer (8 bytes, Short) for call to '__rt_exit' from exit.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__user_libspace' from free.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_fp_init' from lib_init.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__heap_extend' from malloc.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__raise' from rt_raise.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_errno_addr' from ftell.o(.text).
12 Veneer(s) (total 72 bytes) added to the image.
```

（3）Thumb 状态下的功能指针

任何指向 Thumb 函数（由 Thumb 指令完成的功能函数并且其返回状态也为 Thumb 状态）的指针，其最低有效位（LSB）必为 1。

当重定位 Thumb 代码中的地址标号时，连接器将自动设置地址的最低有效位。如果在程序中使用绝对地址，连接器将无法完成该设置。因此，在 Thumb 代码中使用绝对地址时，必须手工设置为其地址加 1。

下面的例子显示了 Thumb 代码的功能指针的使用。

```
typedef int (*FN)();
myfunc() {
    FN fnptrs[] = {
        (FN)(0x8084 + 1),        // 有效的 Thumb 地址
        (FN)(0x8074)         // 无效的 Thumb 地址
    };
    FN* myfunctions = fnptrs;
    myfunctions[0]();            // 调用成功
    myfunctions[1]();            // 调用失败
}
```

3.2 ARM 汇编器支持的伪操作

3.2.1 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）[1]，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的，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。

[1] 大学有些文献中也称其为操作标识。

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。

3.2.2 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。

➢ 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>
```

① <gblx>

取值为 GBLA、GBLL、GBLS 三者中的之一。

② <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
```

➢ 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>
```

① <LClx>

取值为 LCLA、LCLL、LCLS 三者中的之一。

② <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
```

➢ 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。

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
```

① 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。

② <setx>

取值为 SETA、SETL、SETS 三者中的之一。

③ 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"
```

➢ 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次序根据寄存器的编号由低到高，与列表中的寄存器排列次序无关。

语法格式如下：

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

① Name

寄存器列表的名称。

② 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}
```

3.2.3 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}
```

① {label}

程序标号。

② expr

可以是−128～255 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（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}…
```

① {label}

程序标号，可选。

② expr

数字表达式，取值范围为−32768～65525。

（2）使用说明

DCW 可能在分配的内存单元前加一个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是 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。

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

3．DCD（DCWU）

（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 DCDU 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}
```

① {label}

程序标号，可选。

② 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 可能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是 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}
```

① {label}

程序标号，可选。

② fpliteral

单精度浮点数

（2）使用说明

DCFS 可能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是 DCFSU 伪操作。

此伪操作使用的单精度浮点数的范围为：1.17549435e－38～3.40282347e＋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}
```

① {label}

程序标号，可选。

② fpliteral

双精度浮点数。

（2）使用说明

DCFS 可能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是 DCFSU 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，伪操作 DCFS（DCFSU）不可使用。

此伪操作使用的单精度浮点数的范围为：2.22507385850720138e－308～1.79769313486231571e＋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 个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}
```

① {label}

程序标号，可选。

② 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 可能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是 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

（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
```

① {label}

程序标号，可选。

② 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。

MAP 伪操作通常与 FIELD 伪操作配合使用来定义结构化的内存表。

语法格式如下。

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

① 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。

② 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

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。

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
```

① {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。

② expr

FIELD 指定的域所占内存单元字节数。

（2）使用说明

MAP 伪操作中的基地址寄存器（base-register）一旦指定，将被其后的所有 FIELD 伪操作定义的数据域默认使用，指定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 个字节，位置为 0x100；B 的长度为 32 个字节，位置为 0x110；S 的长度为 256 个字节，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]
```

3.2.4 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，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，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
```

logical-expression：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，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
```

logical-expression：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”，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，标号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
```

① {$label}

$标号在宏指令被展开时，标号会被替换为用户定义的符号。通常，在一个符号前使用“$”表示该符号被汇编器编译时，使用相应的值代替该符号。

② macroname

所定义的宏的名称。

③ $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。首先要先用 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
$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    $paraml,$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，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。

当这些伪操作混合使用时，总的嵌套深度不能超过 256。

3.2.5 杂项伪操作

ARM 汇编中还有一些其他的伪操作，在汇编程序中经常会被使用，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}}}
```

① 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。
② 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。

n 为汇编时变量，由编译器根据内存对齐方式决定其值。

③ pad
用作填充的字节。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 来更有效的使用 Cache。比如，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。

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}
```

① 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|。

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。

② 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 10-3 为各段属性及相关说明。

表 10-3　　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。

（3）示例

下面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），但在一个源文件里最多只能有一个 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}
```

① name

EQU 伪指令定义的字符名称。

② 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。

③ 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]}
```

① 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。

② [WEAK]

[WEAK]选项声明其他的同名标号优先于该标号被引用。

③ [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，被定义为全局的（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：符号不能其他组件引用。
- 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
```

① 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。

② 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]}
```

① symbol

要引用的符号名称。该名称区分大小写。

② [WEAK]

[WEAK]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。

③ [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，被定义为全局的（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。

```
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。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。

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。EXTERN 声明的符号，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。

```
IMPORT  symbol{[WEAK，attr]}
```

① 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。

② [WEAK]

[WEAK]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。

③ [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，被定义为全局的（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：符号不能被其他组件引用。
- 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。

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，被包含的文件不作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。

华清远见

（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
```

3.3 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。

3.3.1 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−255～255，当地址值是字对齐时，取值范围为−1020～1020。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
```

① cond

可选的指令执行条件。

② .W

可选项。指定指令宽度（Thumb-2 指令集支持）。

③ register

目标寄存器。

④ 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
        ; n-1 条 DCD 伪操作
        DCD     valuen              ;所要装载入 R4 的值
        ;更多 DCD 伪操作
```

3.3.2 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
```

① cond

可选的指令执行条件。

② register

目标寄存器。

③ 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，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。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。

3.3.3 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]
```

① cond

可选的指令执行条件。

② .W

可选项。指定指令宽度（Thumb-2 指令集支持）。

③ register

目标寄存器。

④ 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。

a．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。

b．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。

⑤ 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 被声明为外部的表示式时，汇编器将在目标文件中插入链接重定位伪操作，由链接器在链接时生成该地址。

（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。

华清远见

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。

```
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
```

3.4 汇编语言与 C/C++的混合编程

在 C 或 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/C++程序中实现 C/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 或 C++程序中完成对程序状态寄存器的操作。

3.4.1 内联汇编

1. 内联汇编语法

内联汇编使用“_asm”（C++）和“asm”（C 和 C++）关键字声明，语法格式如下所示。

```
• __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，使用反斜杠“/”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。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所示。

```
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();
}
```

3.4.2 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 或 C++函数定义中去。嵌入式汇编器提供对目标处理器不受限制的低级别访问。

本小节将介绍以下内容：

（1）嵌入式汇编程序语法；

（2）嵌入式汇编程序表达式和 C 或 C++表达式之间的差异；

（3）嵌入式汇编函数的生成；

（4）__cpp 关键字；

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（C 和 C++）或 asm（C++）函数限定符标记，可用于：

（1）成员函数；

（2）非成员函数；

（3）模板函数；

（4）模板类成员函数。

用__asm 或 asm 声明的函数可以有调用参数和返回类型。它们从 C 和 C++中调用的方式与普通 C 和 C++函数调用方式相同。嵌入式汇编函数语法是：

```
__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，这些编译选项如下所示。

（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。

可以显示地使用 ARM、Thumb 和 Code16 伪操作改变嵌入式汇编的执行状态。关于 ARM 伪操作的详细信息请参见指令伪操作一节。如果使用的处理器支持 Thumb-2 指令，则可以在 Thumb 状态下，在嵌入式汇编中使用 Thumb-2 指令。

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，注意和上一节中内联汇编中字符串复制的例子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 或 C++表达式之间的差异

嵌入式汇编表达式和 C 或 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 或 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、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 和 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
```

4．嵌入式汇编函数的生成

由关键字__asm 声明的嵌入式汇编程序，在编译时将作为整个文件体传递给 ARM 汇编器。在传递过程中，__asm 函数的顺序保持不变（用模板实例生成的函数除外）。正是由于嵌入式汇编的这个特性，使得由一个__asm 标识的嵌入式汇编程序调用在同一文件中的另一个嵌入式汇编程序是可以实现的。

当使用编译器 armcc 时，局部链接器（Partial Link）将汇编程序产生的目标文件与编译 C 程序的目标文件相结合，产生单个目标文件。

编译程序为每个 __asm 函数生成 AREA 命令。如以下__asm 函数：

```
#include <cstddef>
```

```
struct X { int x,y; void addto_y(int); };
__asm void X::addto_y(int) {
    LDR     R2,[R0, #__cpp(offsetof(X, y))]
    ADD     R1,R2,R1
    STR     R1,[R0, #__cpp(offsetof(X, y))]
    BX      lr
}
```

对于此函数，编译器生成：

```
AREA ||.emb_text||, CODE, READONLY
EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
 LDR R2,[R0, #4]
 ADD R1,R2,R1
 STR R1,[R0, #4]
 BX lr
 ENDP
 END
```

由上面的例子可以看出，对于变量 offsetof 的使用必须加__cpp（）标识符才能引用，因为该变量是在 cstddef 头文件中定义的。

由__asm 声明的常规函数被放在名为.emb_text 的段（Section）中。这一点也是嵌入式汇编和内联汇编最大的不同。相反，隐式实例模板函数（Implicitly Instantiated Template Function）和内联汇编函数放在与函数名同名的区域（Area）内，并为该区域增加公共属性。这就确保了这类函数的特殊语义得以保持。

由于内联和模板函数的区域的特殊命名，所以这些函数不按照文件中定义的顺序排列，而是任意排序。因此，不能以__asm 函数在原文件中的排列顺序，来判断它们的执行顺序，也就是说，即使两个连续排列的__asm 函数，也不一定能顺序执行。

5．关键字__cpp

可用__cpp 关键字从汇编代码中访问 C 或 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。标识符__cpp 内的表达式必须是适合用作 C++静态初始化的常量表达式（请参阅 ISO/IEC 14882:2003 中的 3.6.2 非本地对象初始化一节和本书的常量表达式一节）。

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)
```

__cpp 表达式中的名称可在__asm 函数的 C++上下文中查阅。__cpp 表达式结果中的任何名称按照要求被损毁并自动为其生成 IMPORT 语句。

3.4.3 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，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
```

3.4.4 C++中使用 C 头文件

本节描述如何在 C++代码中使用 C 头文件。从 C++调用 C 头文件之前，C 头文件必须包含在 extern“C”命令中。本节包含以下两部分内容：

（1）在 C++中使用系统的 C 头文件；

（2）在 C++中使用自定义的 C 头文件。

1．在 C++中使用系统 C 头文件

要包括标准的系统 C 头文件，如 stdio.h，不必进行任何特殊操作，由编译器自动包含标准 C 头文件。例如：

```
#include <stdio.h>
int main()
{
    …    // C++ 代码
    return 0;
}
```

如果使用此语法包含头文件，则所有库名都放在全局命名空间中。

C++标准规定可以通过特定的 C++头文件获取 C 头文件。这些文件与标准 C 头文件一起安装在 install_directory\RVCT\Data\2.0\build_num \include \platform 目录下，可以用常规方法进行引用。例如：

```
#include <cstdio-h>
int main()
{
   …    // C++ 代码
```

```
    return 0;
}
```

2．在 C++中使用自定义的 C 头文件

要包含自己的C头文件，用户必须将#include 命令包在 extern "C"语句中。可以用以下方法完成此操作。

（1）在#include 文件之前使用 extern，如下例所示。

```
// C++ code
extern "C" {
#include "my-header1.h"
#include "my-header2.h"
}
int main()
{
    // …
    return 0;
}
```

（2）将 extern "C"语句添加到头文件，如下例所示。

```
/* C header file */
#ifdef __cplusplus    /* 加入到 extern C 结构的开始处 */
extern "C" {
#endif
/* Body of header file */
#ifdef __cplusplus  /* 加入到 extern C 结构的结束处 */
}
#endif            /* 此时包含在 C++头文件中的 C 头文件已经可以正常使用了*/
```

3.4.5 混合编程调用举例

汇编程序、C 程序及 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）从 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 继续拷贝
    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]!    // 第五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
```

（3）从 C++程序调用 C++。

下面的例子显示了如何从 C++程序中调用 C 程序。

```
struct S {              // 本结构没有基类和虚函数

    S(int s):i(s) { }
    int i;
};
extern "C" void cfunc(S *);
                        // 被调用的 C 函数使用 extern “C” 声明
int f(){
     S s(2);                // 初始化 's'
```

```
    cfunc(&s);          // 调用 C 函数 'cfunc' 将改变 's'
    return si*3;
}
```

下面的程序段显示了被调用的 C 程序代码。

```
struct S {
    int i;
};
void cfunc(struct S *p) {
/*定义被调用的 C 功能 */
    p->i += 5;
}
```

（4）从 C++程序中调用汇编程序。

下面的例子显示了如何从 C++程序中调用汇编程序。

```
struct S {                  // 本结果没有基类和虚拟函数
                            //
    S(int s) : i(s) { }
    int i;
};
extern "C" void asmfunc(S *);    // 声明被调用的汇编函数

int f() {
    S s(2);                     // 初始化结构体 's'
    asmfunc(&s);            // 调用汇编子程序 'asmfunc'
    return s.i * 3;
}
```

下面是被调用的汇编程序。

```
    PRESERVE8
    AREA Asm, CODE
    EXPORT asmfunc
asmfunc             // 被调用的汇编程序定义
    LDR R1, [R0]
    ADD R1, R1, #5
    STR R1, [R0]
    MOV pc, lr
    END
```

（5）从 C 程序中调用 C++程序。

下面的例子显示了如何从 C 代码中调用 C++程序。

```
struct S {      // 本结构没有基类和虚拟函数
    S(int s) : i(s) { }
    int i;
};
extern "C" void cppfunc(S *p) {
// 定义被调用的 C++代码
// 连接了 C 功能
    p->i += 5;              //
}
```

下面是调用了 C++代码的 C 函数。

```
struct S {
     int i;
};
extern void cppfunc(struct S *p);
/* 声明将会被调用的 C++功能 */
int f(void) {
      struct S s;
      s.i = 2;              /* 初始化 S */
      cppfunc(&s);          /* 调用 cppfunc 函数，该函数可能改变 S 的值 */
      return s.i * 3;
}
```

（6）从汇编中调用 C++程序。

下面的代码显示了如何从汇编中调用 C++程序。

```
struct S {          // 本结构没有基类和虚拟函数
     S(int s) : i(s) { }
     int i;
};
extern "C" void cppfunc(S * p) {
// 定义被调用的 C++功能
// 功能函数体
     p->i += 5;
}
```

在汇编语言中，声明要调用的 C++功能，使用带连接的跳转指令调用 C++功能。

```
    AREA Asm, CODE
    IMPORT cppfunc        ;声明被调用的 C++ 函数名
    EXPORT   f
f
    STMFD  sp!,{lr}
    MOV    R0,#2
    STR    R0,[sp,#-4]!       ;初始化结构体
    MOV    R0,sp          ;调用参数为指向结构体的指针
    BL     cppfunc            ;调用 C++功能'cppfunc'
    LDR    R0, [sp], #4
    ADD    R0, R0, R0,LSL #1
    LDMFD  sp!,{pc}
    END
```

（7）在 C 和 C++函数间传递参数。

下面的例子显示了如何在 C 和 C++函数间传递参数。

```
extern "C" int cfunc(const int&);
// 声明被调用的 C 函数
extern "C" int cppfunc(const int& r) {
// 定义将被 C 调用的 C++函数
     return 7 * r;
}
int f() {
     int i = 3;
     return cfunc(i);    // 向 C 函数传参
```

```
}
```

下面为 C 函数。

```
extern int cppfunc(const int*);
/* 声明将被调用的 C++函数 */
int cfunc(const int *p) {
/*定义被 C++调用的 C 函数*/
    int k = *p + 4;
    return cppfunc(&k);
}
```

3.5 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/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。

3.6 思 考 题

1. 比较 ARM 指令和 Thumb 指令的不同。
2. 如何从 ARM 状态切换到 Thumb 状态？
3. 在 ARM 汇编中如何定义一个全局的数字变量？
4. ADR 和 LDR 的用法有什么区别？
5. AAPCS 过程调用标准的内容是什么？
6. 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？
7. 汇编代码中如何调用 C 代码中定义的函数？
8. C++代码中如何包含 C 头文件？

华清远见

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 250 号银海大厦 11 层 B 区，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：0755-25590506

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

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067

华清远见