Principles and Practices of Microcontroller (Embedded System Design I) -STM32 Processor II

Gang Chen (陈刚)

Associate Professor

Institute of Unmanned Systems

School of data and computer science

Sun Yat-Sen University

https://www.usilab.cn/team/chengang/

入群实名（班级-学号-姓名）

2019单片机课程群

扫一扫二维码，加入群聊。

ATPCS 概述

- **规定了子程序间调用的基本规则，分类**
  - 支持数据栈限制检查的ATPCS
  - 支持只读段位置无关（ROPI）的ATPCS
  - 支持可读写段位置无关（RWPI）的ATPCS
  - 支持ARM程序和Thumb程序混合使用的ATPCS
  - 处理浮点运算的ATPCS
- **规定了C语言+汇编的接口规范**

过程调用标准ATPCS与AAPCS

- 过程调用标准ATPCS（ARM-Thumb Produce Call Standard）规定了子程序间相互调用的基本规则，ATPCS规定子程序调用过程中寄存器的使用规则、数据栈的使用规则及参数的传递规则。
- 2007年，ARM公司推出了新的过程调用标准AAPCS（ARM Architecture Produce Call Standard）,它只是改进了原有的ATPCS的二进制代码的兼容性。

基本ATPCS

- 寄存器使用规则
- 数据栈使用规则
- 参数传递规则

寄存器使用规则

- （1）子程序间通过寄存器R0～R3传递参数，寄存器R0～R3可记作A0～A3。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，ARM状态下使用寄存器R4～R11来保存局部变量，寄存器R4～R11可记作V1～V8；Thumb状态下只能使用R4～R7来保存局部变量。
- （3）寄存器R12用作子程序间调用时临时保存栈指针，函数返回时使用该寄存器进行出栈，记作IP；在子程序间的链接代码中常有这种使用规则。
- （4）通用寄存器R13用作数据栈指针，记作SP。
- （5）通用寄存器R14用作链接寄存器，记作lr；
- （6）通用寄存器R15用作程序计数器，记作PC 。

数据栈使用规则

- 过程调用标准规定数据栈为FD类型，并且对数据栈的操作时要求8字节对齐的。

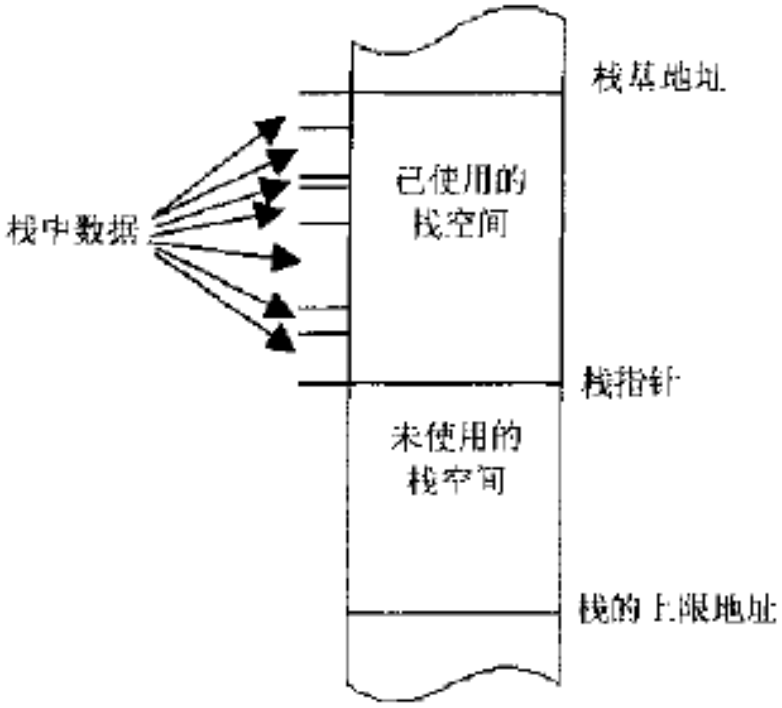


图 6.1　一个数据栈的示意图

参数传递规则

- 1．参数个数可变的子程序参数传递规则
- 对于参数个数可变的子程序，当参数个数不超过4个时，可以使用寄存器R0～R3来传递；当参数个数超过4个时，还可以使用数据栈进行参数传递。
- ２．参数个数固定的子程序参数传递规则
- 如果系统不包含浮点运算的硬件部件且没有浮点参数时，则依次将各参数传送到寄存器R0～R3中，如果参数个数多于4个，将剩余的字数据通过数据栈来传递；
- 如果包括浮点参数则要通过相应的规则将浮点参数转换为整数参数，然后依次将各参数传送到寄存器R0～R3中。如果参数多于4个，将剩余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。

子程序结果返回规则

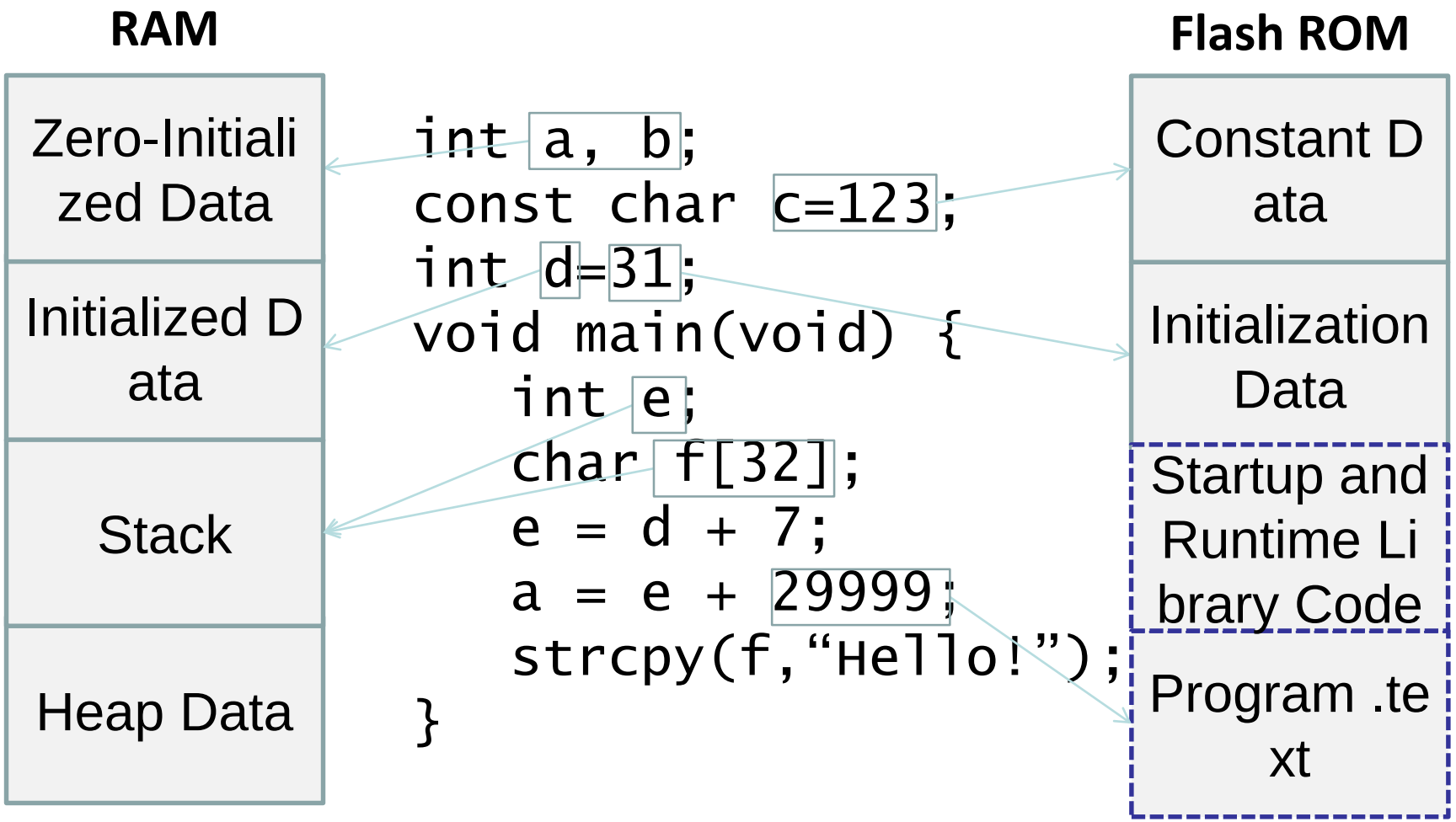
- （1）结果为一个32位的整数时，通过寄存器R0返回；结果为一个64位整数时，通过寄存器R0，R1返回。
- （2）结果为一个浮点数时，可以通过浮点运算部件的寄存器F0、D0或者S0来返回；结果为复合型的浮点数（如复数）时，可以通过寄存器F0～Fn或者Ｄ0～Ｄn来返回。
- （3）对于位数更多的结果，需要通过内存来传递。

ARM汇编语言与嵌入式C混合编程

- 在嵌入式程序设计中，有些场合（如对具体的硬件资源进行访问）必须用汇编语言来实现，可以采用在嵌入式C语言程序中嵌入汇编语言或嵌入式C语言调用汇编语言来实现。

嵌入式C编写与编译

- **Program Memory Use**



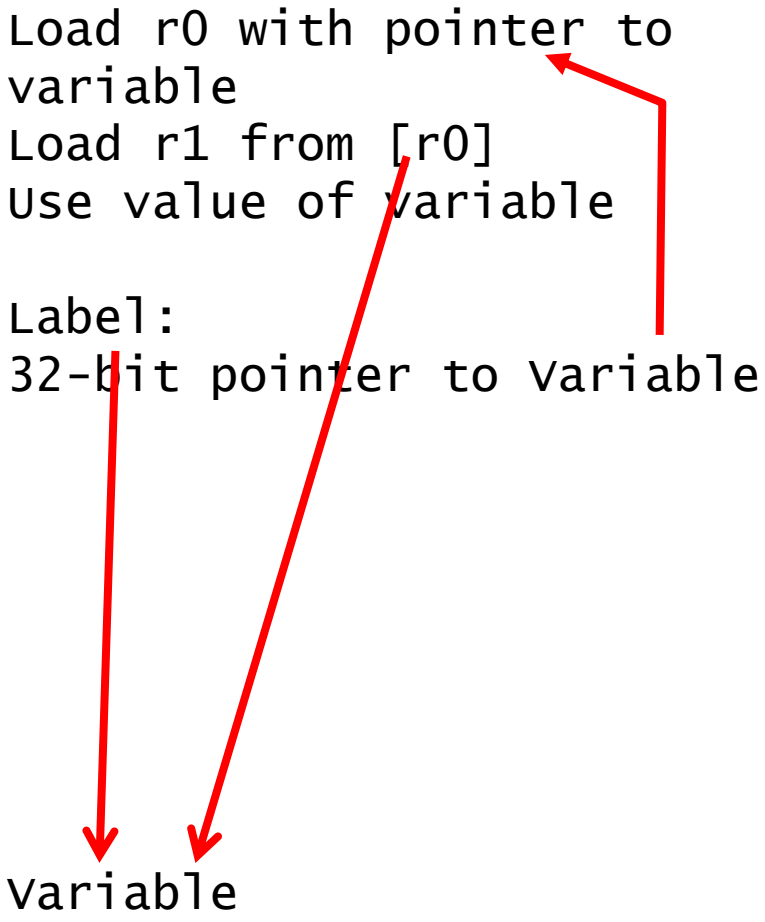
变量

- 全局变量\静态变量:因为这些变量在编译时存在ROM中,代码存在RAM中。编译时自动在代码附近开数据空间存长跳转地址。对变量的操作通过2次取地址完成
- 函数中变量、动态变量：存在该函数堆栈中。通过SP指针+偏移量完成

- **寄存器**
  - R0-R3，LR，SP一般不要用。传参数时占用
- **指针就是地址，比较简单**

Static Variables

- Static var can be located anywhere in 32-bit memory space, so need a 32-bit pointer
- Can't fit a 32-bit pointer into a 16-bit instruction (or a 32-bit instruction), so save the pointer separate from instruction, but nearby so we can access it with a short PC-relative offset
- Load the pointer into a register (r0)
- Can now load variable's value into a register (r1) from memory using that pointer in r0
- Similarly can store a new value to the variable in memory



Automatic Variables

Address	Contents
SP	aiG
SP+4	aiF
SP+8	aiE
SP+0xC	aiB
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

- **Initialize aiE**
- **Initialize aiF**
- **Initialize aiG**
- **Store value for aiB**

```
;;;14     void
static_auto_local( void ) {
000000  b50f PUSH {r0-r3,lr}
;;;15   int aiB;
;;;16   static int siC=3;
;;;17   int * apD;
;;;18   int aiE=4, aiF=5, aiG=6;
000002  2104 MOVS  r1,#4
000004  9102 STR   r1,[sp,#8]
000006  2105 MOVS  r1,#5
000008  9101 STR   r1,[sp,#4]
00000a  2106 MOVS  r1,#6
00000c  9100 STR   r1,[sp,#0]
…
;;;21         aiB = siC + siA;
…
00001c  9103 STR   r1,[sp,#0xc]
```

Using Pointers to Automatics

- C Pointer: a variable which holds the data's address
- aiB is on stack at SP+0xc
- Compute r0 with variable's address from stack pointer and offset (0xc)
- Load r1 with variable's value from memory
- Operate on r1, save back to variable's address

```
;;;22                    apD = & aiB;
00001e   a803  ADD   r0,sp,#0xc
;;;23                    (*apD)++;
000020   6801  LDR   r1,[r0,#0]
000022   1c49  ADDS  r1,r1,#1
000024   6001  STR   r1,[r0,#0]
```

Using Pointers to Statics

- Load r0 with variable's address from address of copy of variable's address
- Load r1 with variable's value from memory
- Operate on r1, save back to variable's address

```
;;;24          apD = &siC;
000026  4833  LDR   r0,|L1.244|
;;;25          (*apD) += 9;
000028  6801  LDR   r1,[r0,#0]
00002a  3109  ADDS  r1,r1,#9
00002c  6001  STR   r1,[r0,#0]
|L1.244|
                DCD       ||siC||
        AREA ||.data||, DATA,
ALIGN=2
||siC||
                DCD       0x00000003
```

数组

- 数组通过内存基地址+偏移量来访问
- 内存基地址需要在代码段附近开空间存储。

Accessing 1-D Array Elements

- Need to calculate element address: sum of…
  - array start address
  - offset: index * element size
- buff2 is array of unsigned characters

- Move n (argument) from r0 into r2
- Load r3 with pointer to buff2
- Load (byte) r3 with first element of buff2
- Load r4 with pointer to buff2
- Load (byte) r4 with element at address buff2+r2
  - r2 holds argument n
- Add r3 and r4 to form sum

Address	Contents
buff2	buff2[0]
buff2 + 1	buff2[1]
buff2 + 2	buff2[2]

```
00009e   4602  MOV   r2,r0
;;;76     i = buff2[0] + buff2[n];
0000a0   4b1b  LDR   r3,|L1.272|
0000a2   781b  LDRB  r3,[r3,#0];buff2
0000a4   4c1a  LDR   r4,|L1.272|
0000a6   5ca4  LDRB  r4,[r4,r2]
0000a8   1918  ADDS  r0,r3,r4
|L1.272|
                     DCD   buff2
```

Accessing 2-D Array Elements

short int buff3[5][7]

Address	Contents
buff3	buff3[0][0]
buff3+1	
buff3+2	buff3[0][1]
buff3+3	
(etc.)	
buff3+10	buff3[0][5]
buff3+11	
buff3+12	buff3[0][6]
buff3+13	
buff3+14	buff3[1][0]
buff3+15	
buff3+16	buff3[1][1]
buff3+17	
buff3+18	buff3[1][2]
buff3+19	
(etc.)	
buff3+68	buff3[4][6]
buff3+69	

- var[rows][columns]
- Sizes
  - Element: 2 bytes
  - Row: 7*2 bytes = 14 bytes (0xe)
- Offset based on row index and column index
  - column offset = column index * element size
  - row offset = row index * row size

函数

- **函数都是有堆栈的**
  - 进入堆栈需要压栈
  - 退出函数需要恢复SP指针，相当于释放堆栈空间

Function Prolog and Epilog

- Save r4 (preserved register) and link register (return address)
- Allocate 32 (0x20) bytes on stack for array x by subtracting from SP
- Compute return value, placing in return register r0
- Deallocate 32 bytes from stack
- Pop r4 (preserved register) and PC (return address)

```
 fun4 PROC
;;;102   int fun4(char a, int
b, char c) {
;;;103     volatile int x[8];
00010a  b510  PUSH {r4,lr}

00010c  b088  SUB   sp,sp,#0x20
...

;;;106         return a+b+c;
00011c  1858  ADDS  r0,r3,r1
00011e  1880  ADDS  r0,r0,r2
;;;107     }
000120  b008  ADD   sp,sp,#0x20

000122  bd10  POP   {r4,pc}
              ENDP
```

Activation Record Creation by Prolog

Smaller address	space for x[0]	Array x	<- 3. SP after sub sp,sp,#0x20
	space for x[1]		
	space for x[2]		
	space for x[3]		
	space for x[4]		
	space for x[5]		
	space for x[6]		
	space for x[7]		
	lr	Return address	<- 2. SP after push {r4,lr}
	r4	Preserved register	
Larger address		Caller's stack frame	<- 1. SP on entry to function, before push {r4,lr}

Activation Record Destruction by Epilog

Smaller address	space for x[0]	Array x	<- 1. SP before add sp,sp,#0x20
	space for x[1]		
	space for x[2]		
	space for x[3]		
	space for x[4]		
	space for x[5]		
	space for x[6]		
	space for x[7]		
	lr	Return address	<- 2. SP after add sp,sp,#20
	r4	Preserved register	
Larger address		Caller's stack frame	<- 1. SP after pop {r4,pc}

注意

- 函数调用时需要保护好参数
- 有重要的数据需要用堆栈保护和传递

Call Example

```
int fun2(int arg2_1, int arg2_2) {
  int i;
  arg2_2 += fun3(arg2_1, 4, 5, 6);
  …
}
```

- Argument 4 into r3
- Argument 3 into r2
- Argument 2 into r1
- Argument 0 into r0
- Call fun3 with BL instruction
- Result was returned in r0, so add to r4 (arg2_2 += result)

```
fun2 PROC
;;;85     int fun2(int arg2_1,
int arg2_2) {
...
0000e0  2306 MOVS  r3,#6
0000e2  2205 MOVS  r2,#5
0000e4  2104 MOVS  r1,#4
0000e6  4630 MOV   r0,r6

0000e8  f7fffffe   BL    fun3

0000ec  1904 ADDS  r4,r0,r4
```

Call and Return Example

```
int fun3(int arg3_1, int arg3_2,
  int arg3_3, int arg3_4) {
  return   arg3_1*arg3_2*
           arg3_3*arg3_4;
}
```

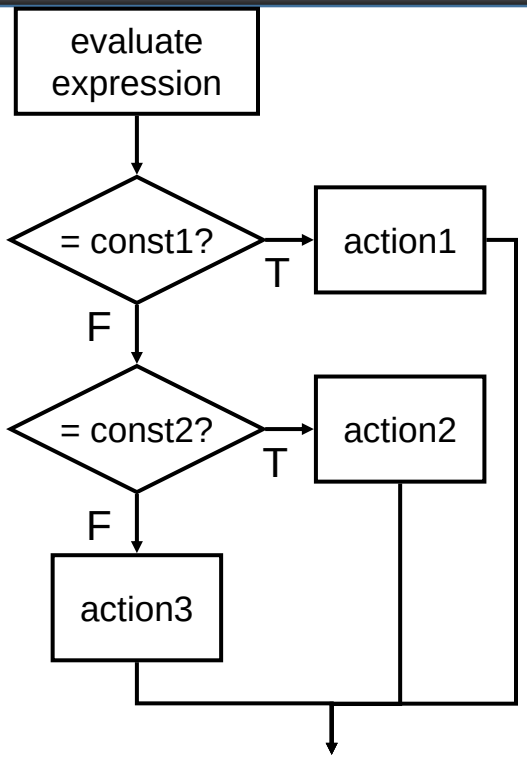
- Save r4 and Link Register on stack
- r0 = arg3_1*arg3_2
- r0 *= arg3_3
- r0 *= arg3_4
- Restore r4 and return from subroutine
- Return value is in r0

```
 fun3 PROC
;;;81     int fun3(int arg3_1,
int arg3_2, int arg3_3, int
arg3_4) {

0000ba  b510 PUSH {r4,lr}


0000c0  4348 MULS r0,r1,r0
0000c2  4350 MULS r0,r2,r0
0000c4  4358 MULS r0,r3,r0

0000c6  bd10 POP  {r4,pc}
```

控制流

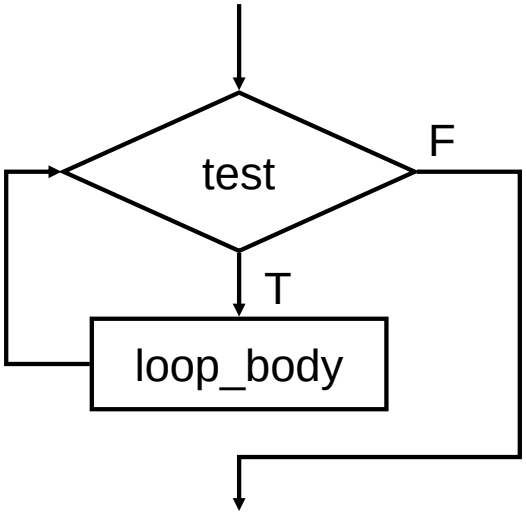
- If –else
- While
- For
- Do-while

Control Flow: Switch



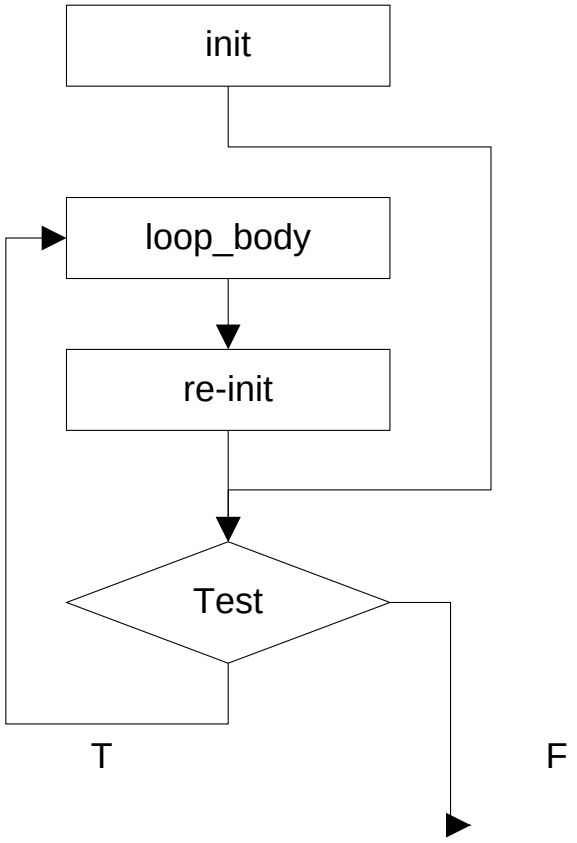

```
;;;45        switch (x) {
000060  2901    CMP   r1,#1
000062  d002    BEQ   |L1.106|
000064  291f    CMP   r1,#0x1f
000066  d104    BNE   |L1.114|
000068  e001    B     |L1.110|
    |L1.106|
;;;46          case 1:
;;;47            y += 3;
00006a  1cd2    ADDS  r2,r2,#3
;;;48            break;
00006c  e003    B     |L1.118|
|L1.110|
;;;49          case 31:
;;;50            y -= 5;
00006e  1f52    SUBS  r2,r2,#5
;;;51            break;
000070  e001    B     |L1.118|
|L1.114|
;;;52          default:
;;;53            y--;
000072  1e52    SUBS  r2,r2,#1
;;;54            break;
000074  bf00    NOP
|L1.118|
000076  bf00    NOP
;;;55        }
```

Iteration: While



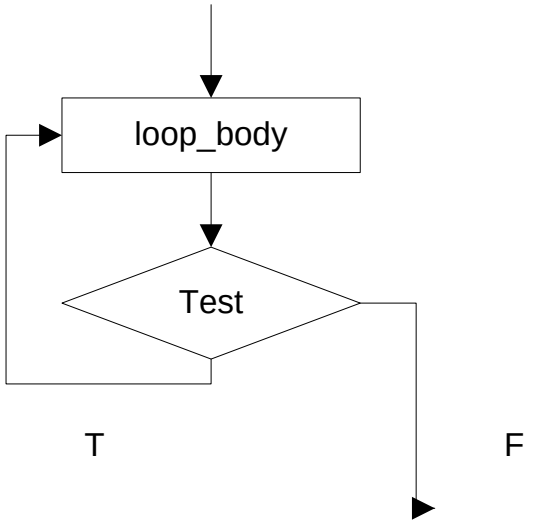

```
;;;57          while (x<10) {
000078  e000 B      |L1.124|
                |L1.122|
;;;58           x = x + 1;
00007a  1c49 ADDS  r1,r1,#1
                |L1.124|
00007c  290a CMP   r1,#0xa
;57
00007e  d3fc BCC   |L1.122|
;;;59        }
```

Iteration: For




```
;;;61        for (i = 0; i <
10; i++){
000080  2300 MOVS  r3,#0
000082  e001 B     |L1.136|

                |L1.132|
;;;62           x += i;
000084  18c9 ADDS  r1,r1,r3
000086  1c5b ADDS  r3,r3,#1
;61

                |L1.136|
000088  2b0a CMP   r3,#0xa
;61
00008a  d3fb BCC   |L1.132|
;;;63        }
```

Iteration: Do/While




```
;;;65        do {
00008c  bf00  NOP

                |L1.142|
;;;66          x += 2;
00008e  1c89  ADDS  r1,r1,#2
;;;67        } while (x < 20);
000090  2914  CMP   r1,#0x14
000092  d3fc  BCC   |L1.142|
```

ARM汇编与C语言混合编程

- 内嵌汇编
- 汇编程序中访问C全局变量
- 汇编与C的相互调用

ARM汇编与C语言混合编程

在需要 C 与汇编混合编程时,若汇编代码较结,则可使用直接内嵌汇编的方法混合编程;否则,可以将汇编文件以文件的形式加入项目中,通过 ATPCS 规定与 C 程序相互调用及访问.

在 C 程序嵌入汇编程序,可以实现一些高级语言没有的功能,提高程序执行效率. armcc 编译器的内嵌汇编器支持,ARM 指令集,tcc 编译器的内嵌汇编支持 Thumb 指令集.

内嵌汇编的语法:

```
__asm
{
    指令[;指令]    /*注释*/
    ...
   [指令]
}
```

ARM汇编与C语言混合编程

嵌入汇编程序的例子如下所示，其中 enable_IRQ 函数为使能 IRQ 中断，而 disable_IRQ 函数为关闭 IRQ 中断.

使能/禁能 IRQ 中断：

```
__inline  void  enable_IRQ(void)
{
   int  tmp
   _asm                          //嵌入汇编代码
   {
     MRS  tmp,CPSR               //读取 CPSR 的值
     BIC  tmp,tmp,#0x80          //将 IRQ 中断禁止位 I 清零,即允许 IRQ 中断
     MSR
     CPSR_c,tmp                  //设置 CPSR 的值
   }
}
```

关于 **inline**内联函数

普通函数在调用时会出栈入栈，频繁的出栈入栈会量消耗栈空间，在系统下，栈空间是有限的，如果频繁大量的使用就会造成因栈空间不足所造成的程序出错的问题。

inline是内联函数，即为函数的调用用函数的实体替换，节省了函数调用的成本,**inline** 适合于函数体内代码简单的函数，不能包含复杂的结构控制语句例如**while**、**switch**，并且内联函数本身不能直接调用递归函数。

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

ARM汇编与C语言混合编程

内嵌汇编的指令用法

操作数. 内嵌的汇编指令中作为操作数的寄存器和常量可以是表达式. 这些表达式可以是 char, short 或 int 类型, 而且这些表达式都是作为无符号数进行操作. 若需要带符号数, 用户需要自己处理与符号有关的操作. 编译器将会计算这些表达式的值, 并为其分配寄存器.

物理寄存器. 内嵌汇编中使用物理寄存器有以下限制;

- 不能直接向 PC 寄存器赋值, 程序跳转只能使用 B 或 BL 指令实现
- 使用物理寄存器的指令中, 不要使用过于复杂的C表达式. 因为表达式过于复杂时, 将会需要较多的物理寄存器. 这些寄存器可能与指令中的物理寄存器使用冲突.
- 编译器可能会使用 R12 或 R13 存放编译的中间结果, 在计算表达式的值时可能会将寄存器 R0~R3, R12 和 R14 用于子程序调用. 因此在内嵌的汇编指令中, 不要将这些寄存器同时指定为指令中的物理寄存器.
- 通常内嵌的汇编指令中不要指定物理寄存器, 因为这可能会影响编译器分配寄

内嵌汇编的指令用法

存器,进而影响代码的效率.

常量.在内嵌汇编指令中,常量前面的”#”可以省略.

指令展开.内嵌汇编指令中,如果包含常量操作数,该指令有可能被内嵌汇编器展开成几条指令.

标号.C程序中的标号可以被内嵌的汇编指令使用,但是只有指令B可以使用C程序中的标号,而指令BL则不能使用.

内存单元的分配.所有的内存分配均由 C 编译器完成,分配的内存单元通过变量供内嵌汇编器使用.内嵌汇编器不支持内嵌汇编程序中用于内存分配的伪指令.

SWI 和 BL 指令.在内嵌的 SWI 和 BL 指令中,除了正常的操作数域外,还必须增加以下 3 个可选的寄存器列表:

- 第 1 个寄存器列表中的寄存器用于输入的参数.
- 第 2 个寄存器列表中的寄存器用于存储返回的结果
- 第 3 个寄存器列表中的寄存器的内容可能被被调用的子程序破坏,即这些寄存器是供被调用的子程序作为工作寄存器

内嵌汇编器与 armasm 汇编器的差异

内嵌汇编器不支持通过“.”指示符或 PC 获取当前指令地址;不支持 LDR Rn, =expr 伪指令,而使用 MOV Rn, expr 指令向寄存器赋值;不支持标号表达式;不支持 ADR 和 ADRL 伪指令;不支持 BX 指令;不能向 PC 赋值.

使用 0x 前缀代替”&”,表示十六进制数.使用 8 位移位常数导致 CPSR 的标志更新时,N、Z、C 和 V 标志中的 C 不具有真实意义.

内嵌汇编注意事项

- ✓ 必须小心使用物理寄存器,如 R0~R3, IP, LR 和 CPSR 中的 N, Z, C, V 标志位.因为计算汇编代码中的 C 表达式时,可能会使用这些物理寄存器,并会修改 N, Z, C, V 标志位.如:

内嵌汇编

```
__asm
  {  MOV   R0,x
     ADD   y,R0,x/y       //计算 x/y 时 R0 会被修改
  }
```

在计算 x/y 时 R0 会被修改,从而影响 R0+x/y 的结果.用一个 C 程序的变量代替 R0 就可以解决这个问题:

```
__asm
{
   MOV   var,x
   ADD   y,var,x/y
 }
```

内嵌汇编器探测到隐含的寄存器冲突就会报错.

✓ 不要使用寄存器代替变量. 尽管有时寄存器明显对应某个变量, 但也不能直接使用寄存器代替变量.

```
int  bad_f(int x)     //x 存放在 R0 中
{
    __asm
        {
            ADD  R0, R0, #1 //发生寄存器冲突, 实际上 x 的值没有变化
        }
   return(x);
}
```

尽管根据编译器的编译规则似乎可以确定 R0 对应 x, 但这样的代码会使内嵌汇编器认为发生了寄存器冲突. 用其他寄存器代替 R0 存放参数 x, 使得该函数将 x 原封不动地返回.

这段代码的正确写法如下：

```
int  bad_f(int x)
{    __asm
     {
        ADD   x,x,#1
     }
      return(x)
   }
```

内嵌汇编

✓ 使用内嵌式汇编无需保存和恢复寄存器. 事实上, 除了 CPSR 和 SPSR 寄存器, 对物理寄存器先读后写都会引起汇编器报错. 例如. :

```
int  f(int x)
{   __asm
    {
     STMFD   SP!{R0}   //保存 R0. 先读后写, 汇编出错
     ADD     R0, x, 1
     EOR     x, R0, x
     LDMFD   SP!, {R0}
     }
  returnt(x):
 }
```

LDM 和 STM 指令的寄存器列表中只允许使用物理寄存器. 内嵌汇编可以修改处理器模式, 协处理器模式和 FP, SL, SB 等 APCS 寄存器. 但是编译器在编译时并不了解这些变化, 所以必须保证在执行 C 代码前恢复相应被修改的处理器模式.

内嵌汇编

✓ 汇编语言中的”.”号作为操作数分隔符号.如果有C表达式作为操作数,若表达式包含有“.”必须使用“(”号和“)”号将其归纳为一个汇编操作数。例如：

```
_asm
{
    ADD x,y,(f(),z)    //“f(),z”为一个带有“.”的C表达式
 }
```

汇编程序中访问C全局变量

访问全局变量

使用 IMPORT 伪指令引入全局变量,并利用 LDR 和 STR 指令根据全局变量的地址访问它们,对于不同类型的变量,需要采用不同选项的 LDR 和 STR 指令:

unsigned char	LDRB/STRB
unsigned short	LDRH/STRH
unsingned int	LDR/STR
char	LDRSB/STRSB
short	LDRSH/STRSH

对于结构,如果知道各个数据项的偏移量,可以通过存储/加载指令访问.如果结构所占空间小于 8 个字,可以使用 LDM 和 STM 一次性读写.

汇编程序中访问C全局变量

下面例子是一个汇编代码的函数,它读取全局变量 globval,将其加 1 后写回.

访问 C 程序的全局变量：

```
                AREA    globats,CODE,READONLY]
                EXPORT  asmsubroutime
                IMPORt  glovbvar                ;声明外部变量 glovbvar
asmsubroutime
                LDR     R1,=glovbvar            ;装载变量地址
                LDR     R0,[R1]                 ;读出数据
                ADD     R0,R0,#1                ;加 1 操作
                STR     R0,[R1]                 ;保存变量值
                MOV     PC LR
                END
```

C 程序调用汇编程序

汇编程序的设置要遵循 ATPCS 规则，保证程序调用时参数的正确传递.

✓ 在汇编程序中使用 EXPORT 伪指令声明本子程序，使其它程序可以调用此子程序.

✓ 在 C 语言程序中使用 extern 关键字声明外部函数(声明要调用的汇编子程序)，即可调用此汇编子程序.

如以下程序所示，汇编子程序 strcopy 使用两个参数，一个表示目标字符串地址，一个表示源字符串的地址，参数分别存放 R0，R1 寄存器中.

调用汇编的 C 函数：

```
#include  <stdio.h>
extern    void  strcopy(char*d,const char*s) //声明外部函数,即要调用的汇编子程序
int  mian(void)
{
   const  char *srcstr="First string-source";     //定义字符串常量
   char   dstsrt[] ="Second string-destination"; //定义字符串变量
   printf("Before copying: \n");
   printf("' %s' \n  '%s\n," srcstr,dststr); //显示源字符串和目标字符串的内容
   strcopy(dststr,srcstr);        //调用汇编子程序,R0=dststr,R1=srcstr
   printf("After copying: \n")
   printf("' %s' \n  '%s\n," srcstr,dststr); //显示 strcopy 复制字符串结果
   return(0);
}
```

被调用汇编子程序:

```
        AREA      SCopy,CODE,READONLY
        EXPORT    strcopy   ;声明 strcopy,以便外部程序引用
strcopy
        ;R0 为目标字符串的地址
        ;R1 为源字符串的地址    ;
LDRB   R2,[R1],#1       ;读取字节数据,源地址加 1
STRB   R2,[R0],#1       ;保存读取的 1 字节数据,目标地址加 1
CMP    r2,#0            ;判断字符串是否复制完毕
BNE    strcopy          ;没有复制完毕,继续循环
MOV    pc,lr            ;返回
END
```

汇编程序调用 C 程序

汇编程序的设置要遵循 ATPCS 规则，保证程序调用时参数的正确传递.

在汇编程序中使用 IMPORT 伪指令声明将要调用的 C 程序函数.

在调用 C 程序时，要正确设置入口参数，然后使用 BL 调用.

以下程序清单所示，程序使用了 5 个参数，分别使用寄存器 R0 存储第 1 个参数，R1 存储第 2 个数，R2 存储第 3 个参数，R3 存储第 4 个参数，第 5 个参数利用堆栈传送. 由于利用了堆栈传递参数，在程序调用用结果后要调整堆栈指针

汇编调用 C 程序的 C 函数：

```
/*函数 sum5()返回 5 个整数的和*/
int sum5(int  a,lit  b,  int  c,int  d,int  e)
{
   return(a+b+c+d+e);     //返回 5 个变量的和
}
```

汇编调用 C 程序的汇编程序：

```
    EXPORT     CALLSUM5
    AREA    Example,  CODE,READONLY
    IMPORT   sum5              ;声明外部标号 sum5,即 C 函数 sum5()
CALLSUMS
    STMFD    SP!{LR}           ;LR 寄存器放栈
    ADD      R1,R0,R0          ;设置 sum5 函数入口参数,R0 为参数 a
    ADD      R2,R1,R0          ;R1 为参数 b,R2 为参数 c
    ADD      R3,R1,R2,
    STR       R3,[SP,#-4]!     ;参数 e 要通过堆栈传递
    ADD       R3,R1,R1         ;R3 为参数 d
    BL        sum5             ;调用 sum5(),结果保存在 R0
    ADD      SP,SP#4           ;修正 SP 指针
    LDMFD    SP,{PC            ;子程序返回
    END
```

C函数原型：

```
int  g(int a，  int b，  int c，  int d，  int e)
 {
return  a+b+c+d+e;
}
```

////汇编程序调用C程序g（）计算5个整数i， 2*i， 3*i， 4*i， 5*i 的和。

汇编源程序：

```
EXPORT  f
AREA  f，CODE，READONLY
IMPORT  g                    ；声明该变量函数g( )，i在R0中
```

```
STR LR，[SP，#-4]!                ；预先保存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(第5个参数)
STR R3，[SP，#-4]!                ；将第5个参数压人堆栈
ADD R3，R1，R1          ；计算4 * i（第4个参数）
BL g                              ；调用C程序g( )
ADD SP，SP，#4                ；调整数据栈指针，准备返回
LDR PC，［SP］，#4              ；从子程序返回
END
```

在C语言程序中调用汇编程序

汇编程序的设计要遵守ATPCS。在汇编程序中使用EXPORT伪操作来声明，使得本程序可以被其它程序调用。

在C程序调用该汇编程序之前使用extern关键词来声明该汇编程序。

在汇编程序中调用C语言程序

汇编程序的设计要遵守ATPCS。在汇编程序调用该C程序之前使用IMPORT伪操作来声明该C程序，通过BL指令来调用子程序。

在C程序中不需要使用任何关键字来声明将被汇编语言调用的C程序