C和汇编如何互相调用？嵌入式工程师必须掌握

原创 土豆居士 一口Linux 2020-12-14 07:59

收录于合集
#所有原创 206 #从0学arm 27

ARM系列文章，请点击以下汇总链接：

《从0学arm合集》

一、gcc 内联汇编

内联汇编即在C中直接使用汇编语句进行编程，使程序可以在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)
2. 程序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

格式：

```
__asm__ __volatile__("asm code"
 :output
 :input
 :changed registers);
```

asm或__asm__开头，小括号+分号，括号内容写汇编指令。指令+\n\t 用双引号引上。

参数

「asm code」主要填写汇编代码：

```
"mov r0, r0\n\t"
"mov r1,r1\n\t"
"mov r2,r2"
```

「output(asm->C)」用于定义输出的参数，通常只能是变量：

```
:"constraint" (variable)
"constraint"用于定义variable的存放位置：
 r 表示使用任何可用的寄存器
 m 表示使用变量的内存地址
 + 可读可写
 = 只写
 & 表示该输出操作数不能使用输入部分使用过的寄存器，只能用"+&"或"=&"的方式使用
```

「input(C->asm)」用于定义输入的参数，可以是变量也可以是立即数：

```
:"constraint" (variable/immediate)
"constraint"用于定义variable的存放位置：
 r 表示使用任何可用的寄存器(立即数和变量都可以)
 m 表示使用变量的内存地址
 i 表示使用立即数
```

Note:

1. 使用__asm__和__volatile__表示编译器将不检查后面的内容，而是直接交给汇编器。
2. 如果希望编译器为你优化，__volatile__可以不加
3. 没有asm code也不能省略""
4. 没有前面的和中间的部分，不可以相应的省略:
5. 没有changed 部分，必须相应的省略:
6. 最后的;不能省略，对于C语言来说这是一条语句
7. 汇编代码必须放在一个字符串内，且字符串中间不能直接按回车换行，可以写成多个字符串，注意中间不能有任何符号，这样就会将两个字符串合并为一个
8. 指令之间必须要换行，还可以使用\t使指令在汇编中保持整齐

举例

例1：无参数，无返回值 这种情况，output和input可以省略：

```
asm
( //汇编指令
 "mrs r0,cpsr     \n\t"
 "bic r0,r0,#0x80 \n\t"
 "msr cpsr,r0     \n\t"
);
```

例2：有参数 ，有返回值 让内联汇编做加法运算，求a+b，结果存在c中

```
int a =100,  b =200,  c =0;
asm
(
 "add %0,%1,%2\n\t"
 : "=r"(c)
 : "r"(a),"r"(b)
 : "memory"
);
```

%0 对应变量c %1 对应变量a %2 对应变量b

例3：有参数 2 ，有返回值

让内联汇编做加法运算，求a+b，结果存在sum中，把a-b的存在d中

```
asm volatile
(
 "add %[op1],%[op2],%[op3]\n\t"
 "sub %[op4],%[op2],%[op3]\n\t"
 :[op1]"=r"(sum),[op4]"=r"(d)
 :[op2]"r"(a),[op3]"r"(b)
 :"memory"
);
```

%0 对应变量c %1 对应变量a %2 对应变量b

三、ATPCS规则：（ARM、thumber程序调用规范）

为了使单独编译的C语言程序和汇编程序之间能够相互调用,必须为子程序之间的调用规定一定的规则.ATPCS就是ARM程序和THUMB程序中子程序调用的基本规则。

基本ATPCS规定了在子程序调用时的一些基本规则，包括下面3方面的内容：

1. 各寄存器的使用规则及其相应的名称。
2. 数据栈的使用规则。
3. 参数传递的规则。

1. 寄存器的使用必须满足下面的规则：

- 1）子程序间通过寄存器R0—R3来传递参数，这时，寄存器R0～R3可以记作A1-A4。被调用的子程序在返回前无需恢复寄存器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．这时，寄存器 R4 ～ R11可以记作V1 ～ V8。如果在子程序中使用到了寄存器V1～V8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用作过程调用时的临时寄存器（用于保存SP，在函数返回时使用该寄存器出栈）， 记作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用作数据栈指针，记作sp。在子程序中寄存器R13不能用作其他用途。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，记作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则可以用作其他用途。
- 6）寄存器R15是程序计数器，记作pc。它不能用作其他用途。

ATPCS下ARM寄存器的命名：

寄存器	别名	功能
R0	a1	工作寄存器

寄存器	别名	功能
R1	a2	工作寄存器
R2	a3	工作寄存器
R3	a4	工作寄存器
R4	v1	必须保护;局部变量寄存器
R5	v2	必须保护;局部变量寄存器
R6	v3	必须保护;局部变量寄存器
R7	v4	必须保护;局部变量寄存器
R8	v5	必须保护;局部变量寄存器
R9	v6	必须保护;局部变量寄存器
R10	sl	栈限制
R11	fp	帧指针
R12	ip	指令指针
R13	sp	栈指针
R14	lr	连接寄存器

2、堆栈使用规则:

ATPCS规定堆栈为FD类型，即满递减堆栈。并且堆栈的操作是8字节对齐。

而对于汇编程序来说,如果目标文件中包含了外部调用,则必须满足以下条件:

1. 外部接口的数据栈一定是8位对齐的，也就是要保证在进入该汇编代码后,直到该汇编程序调用外部代码之间,数据栈的栈指针变化为偶数个字;

2. 在汇编程序中使用PRESERVE8伪操作告诉连接器,本汇编程序是8字节对齐的.

3、参数的传递规则:

根据参数个数是否固定,可以将子程序分为参数个数固定的子程序和参数个数可变的子程序.这两种子程序的参数传递规则是不同的.

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,当参数不超过4个时,可以使用寄存器R0~R3来进行参数传递,当参数超过4个时,还可以使用数据栈来传递参数.

在参数传递时,将所有参数看做是存放在连续的内存单元中的字数据。然后,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.

按照上面的规则,一个浮点数参数可以通过寄存器传递,也可以通过数据栈传递,也可能一半通过寄存器传递, 另一半通过数据栈传递。

举例:

```
void func(a,b,c,d,e)
   a -- r0
   b -- r1
   c -- r2
   d -- r3
   e -- 栈
```

2.参数个数固定的子程序参数传递规则

对于参数个数固定的子程序,参数传递与参数个数可变的子程序参数传递规则不同,如果系统包含浮点运算的硬件部件。

浮点参数将按照下面的规则传递: (1) 各个浮点参数按顺序处理; (2) 为每个浮点参数分配FP寄存器;

分配的方法是,满足该浮点参数需要的且编号最小的一组连续的FP寄存器.第一个整数参数通过寄存器R0~R3来传递,其他参数通过数据栈传递.

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.

举例：
使用r0 接收返回值

```
int func1(int m, int n)
 m  -- r0
 n  -- r1
 返回值给 r0
```

「为什么有的编程规范要求自定义函数的参数不要超过4个？」答：因为参数超过4个就需要压栈退栈，而压栈退栈需要增加很多指令周期。对于参数比较多的情况，我们可以把数据封装到结构体中，然后传递结构体变量的地址。

四、C语言和汇编相互调用

C和汇编相互调用要特别注意遵守相应的ATPCS规则。

1. C调用汇编

例1：c调用汇编文件中函数带返回值 简化代码如下，代码架构可以参考《7. 从0开始学ARM-GNU伪指令、代码编译，lds使用》。

```
;.asm
add:
 add r2,r0,r1
 mov r0,r2
    MOV pc, lr
```

main.c

```
extern int add(int a,int b);

printf("%d \n",add(2,3));
```

1. a->r0,b->r1
2. 返回值通过r0返回计算结果给c代码

例2，用汇编实现一个strcopy函数

```
;.asm
.global strcopy
strcopy:      ;R0指向目的字符串 ;R1指向源字符串
 LDRB R2, [R1], #1   ;加载字字符并更新源字符串指针地址
 STRB R2, [R0], #1   ;存储字符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继续循环
 MOV pc, lr   ;程序返回
```

```
//.c
#include <stdio.h>
extern void strcopy(char* des, const char* src);
int main(){
 const char* srcstr = "yikoulinux";
 char desstr[]="test";
 strcopy(desstr, srcstr);
 return 0;
}
```

2. 汇编调用C

```
//.c
int fcn(int a, int b , int c, int d, int e)
{
 return a+b+c+d+e;
}
```

```
;.asm ;
.text .global _start
_start:
 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 fcn ;调用C程序
ADD sp, sp, #4 ;从堆栈中删除第五个参数
.end
```

假设程序进入f时，R0中的值为i ；

```
int f(int i){
 return fcn(i, 2*i, 3*i, 4*i, 5*i);
}
```

五、内核实例

为了让读者有个更加深刻的理解， 以内核中的例子为例：

arch/arm/kernel/setup.c

```
void notrace cpu_init(void)
{
    unsigned int cpu = smp_processor_id();－－－－获取CPU ID
    struct stack *stk = &stacks[cpu];－－－－获取该CPU对于的irq abt和und的stack指针
……
#ifdef CONFIG_THUMB2_KERNEL
#define PLC    "r"－－－－Thumb-2下，msr指令不允许使用立即数，只能使用寄存器。
#else
#define PLC    "I"
#endif    __asm__ (
    "msr    cpsr_c, %1\n\t"－－－－让CPU进入IRQ mode
    "add    r14, %0, %2\n\t"－－－－r14寄存器保存stk->irq
    "mov    sp, r14\n\t"－－－－设定IRQ mode的stack为stk->irq
    "msr    cpsr_c, %3\n\t"
    "add    r14, %0, %4\n\t"
    "mov    sp, r14\n\t"－－－－设定abt mode的stack为stk->abt
    "msr    cpsr_c, %5\n\t"
    "add    r14, %0, %6\n\t"
    "mov    sp, r14\n\t"－－－－设定und mode的stack为stk->und
    "msr    cpsr_c, %7"－－－回到SVC mode
        :－－－－上面是code，下面的output部分是空的
```

```
        : "r" (stk),－－－－对应上面代码中的%0
          PLC (PSR_F_BIT | PSR_I_BIT | IRQ_MODE),－－－－对应上面代码中的%1
          "I" (offsetof(struct stack, irq[0])),－－－－对应上面代码中的%2
          PLC (PSR_F_BIT | PSR_I_BIT | ABT_MODE),－－－－以此类推，下面不赘述
          "I" (offsetof(struct stack, abt[0])),
          PLC (PSR_F_BIT | PSR_I_BIT | UND_MODE),
          "I" (offsetof(struct stack, und[0])),
          PLC (PSR_F_BIT | PSR_I_BIT | SVC_MODE)
        : "r14");－－－－上面是input操作数列表，r14是要clobbered register列表
}
```

推荐阅读

【1】嵌入式工程师到底要不要学习ARM汇编指令？**必读**

【2】7. 从0学ARM-汇编伪指令、lds详解

【3】IP协议入门**必读**

【4】【从0学ARM】你不了解的ARM处理异常之道

【5】4. 从0开始学ARM-ARM汇编指令其实很简单

【6】【典藏】大佬们都在用的结构体进阶小技巧

【7】[粉丝问答6]子进程进程的父进程关系

进群，请加一口君个人微信，带你嵌入式入门进阶。

上一篇
7. 从0学ARM-汇编伪指令、lds详解

下一篇
散装 vs 批发谁效率高？变量访问被ARM架构安排的明明白白

阅读原文

喜欢此内容的人还喜欢

Konva实现图片自适应裁剪
A逐梦博客

面试连环问--操作系统
阿Q正砖

pinia
睡不着所以学编程