专题10-C语言环境初始化

一、栈初始化

1.1、概念解析

首先，需要明白ARM-THUMB子程序调用规则ATPCS：主要包括 寄存器使用规则、数据栈使用规则、参数传递规则。

ATPCS中各寄存器的使用规则及其名称		
r0	a1	参数/结果/scratch寄存器1
r1	a2	参数/结果/scratch寄存器2
r2	a3	参数/结果/scratch寄存器3
r3	a4	参数/结果/scratch寄存器4
r4	v1	ARM状态局部变量寄存器1
r5	v2	ARM状态局部变量寄存器2
r6	v3	ARM状态局部变量寄存器3
r7	v4、wr	ARM状态局部变量寄存器4、 Thumb状态工作寄存器
r8	v5	ARM状态局部变量寄存器5
r9	v6、sb	ARM状态局部变量寄存器6、在支持RWPI的ATPCS中为静态基址寄存器
r10	v7、s1	ARM状态局部变量寄存器7、在支持数据栈检查的ATPCS中为数据栈限制指针
r11	v8	ARM状态局部变量寄存器8
r12	ip	子程序内部调用的scratch寄存器
r13	sp	数据栈指针
r14	lr	连接寄存器
r15	pc	程序计数器

寄存器使用规则总结如下：

1. 子程序间通过寄存器r0 r3来传递参数，这时，寄存器r0r3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用作子程序间scratch寄存器（用于保存SP，在函数返回时使用该寄存器出栈），记作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。

数据栈使用规则：

数据栈有两个增长方向：向内存地址减小的方向增长时，称为 DESCENDING栈 ；向内存地址增加的方向增长时，称为 ASCENDING栈 。

所谓数据栈的增长就是移动栈指针。当栈指针指向栈顶元素（最后一个入栈的数据）时，称为 FULL栈 ；当栈指针指向栈顶元素（最后一个入栈的数据）相邻的一个空的数据单元时，称为 EMPTY栈 。

则数据栈可以分为4种：

FD：Full Descending　　满递减

ED：Empty Descending　空递减

FA ：Full Ascending　　满递增

EA：Empty Ascending　　空递增

ATPCS规定数据栈为FD类型，并且对数据栈的操作是8字节对齐的。使用 stmdb / ldmia 批量内存访问指令来操作FD数据栈。

使用stmdb命令往数据栈中保存内容时，先递减sp指针，再保存数据，使用ldmia命令从数据栈中恢复数据时，先获得数据，再递增sp指针，sp指针总是指向栈顶元素，这刚好是FD栈的定义。

参数传递规则

一般地，当参数个数不超过 4 个时，使用 r0 ~ r3 这4个寄存器来传递参数；如果参数个数超过 4 个，剩余的参数通过数据栈来传递。

对于一般的返回结果，通常使用 r0 ~ r3 来传递。

1.1.1、栈

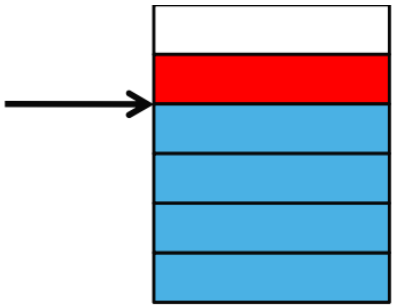
栈是一种具有后进先出性质 的数据组织方式，也就是说后存放的先取出，先存放的后取出。栈底是第一个进栈 的数据所处的位置，栈顶是最后一个进栈的数据所处的位置。

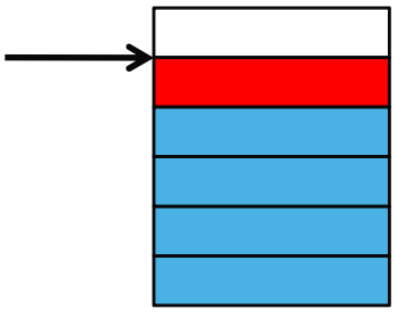
1.1.2、空/满栈

根据SP指针指向的位置，栈可以分为满栈和空栈。

1. 满栈：当堆栈指针SP总是指向最后压入堆栈的数据
2. 空栈：当堆栈指针SP总是指向下一个将要放入数据的空位置

ARM采用满栈！



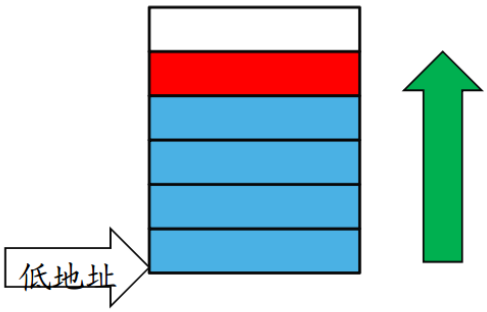


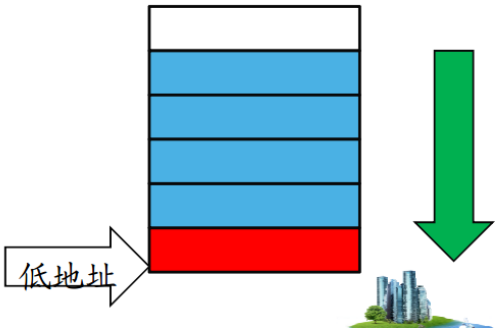
1.1.3、升/降栈

根据SP指针移动的方向，栈可以分为升栈和降栈。

1. 升栈：随着数据的入栈，SP指针从低地址->高地址移动
2. 降栈：随着数据的入栈，SP指针从高地址->低地址移动

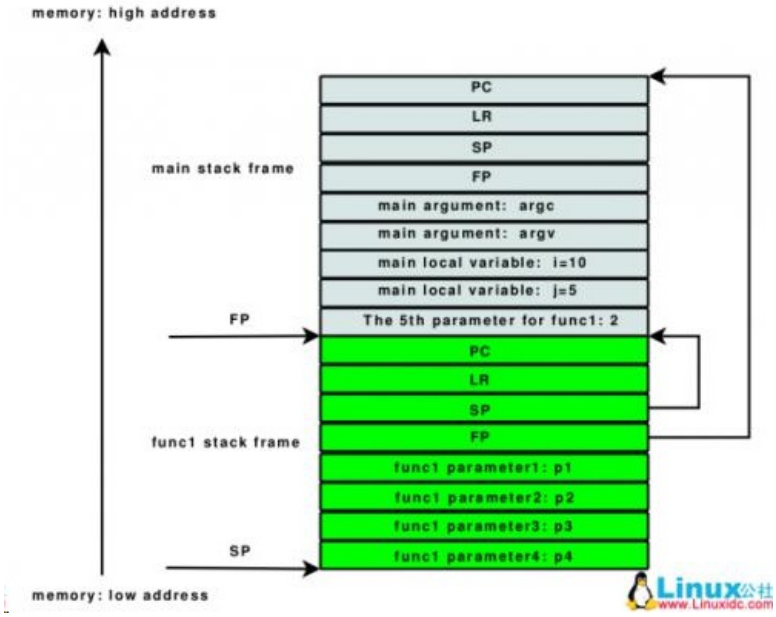
ARM采用降栈！





1.1.4、栈帧

简单的讲,栈帧(stack frame) 就是一个函数所使用的缃陪部分栈，所有函数的栈帧串起来就组成了一个完整的栈。栈帧的两个边界分别由fp(r11)和sp(r13)来限定。



1.2、栈作用

1.2.1、保存局部变量

1.2.2、参数传递

1.2.3、保存寄存器值

1.2.4、代码分析

str fp, [sp, #-4]!)

作用：将 fp 写入 sp-4（因为寄存器为4字节）之后指向的地址，并且将sp-4赋值给sp（！的作用）。

1.3、栈初始化编程

```
190 stack_init:
191         ldr sp, =0x54000000
192         mov pc, lr
```

二、BSS段初始化

2.1、BSS段的作用

初始化的全局变量： 数据段

局部变量： 栈

malloc： 堆

未初始化的全局变量： BSS段

堆与栈的比较：

1.申请方式

(1)栈（satck）:由系统自动分配。

(2)堆（heap）:需程序员自己申请（c：调用malloc,realloc,calloc申请 free 来释放）,并指明大小，并由程序员进行释放。容易产生内存泄漏.

2.申请大小的限制

（1）栈：在windows下栈是向底地址扩展的数据结构，是一块连续的内存区域(它的生长方向与内存的生长方向相反)。栈的大小是固定的。如果申请的空间超过栈的剩余空间时，将提示overflow。

（2）堆：堆是高地址扩展的数据结构（它的生长方向与内存的生长方向相同），是不连续的内存区域。这是由于系统使用链表来存储空闲内存地址的，自然是不连续的，而链表的遍历方向是由底地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。

2.2、为什么要初始化BSS段

为了避免没有初始化的全局变量被使用，造成未知的问题，所以系统工程师需要对BSS段清0的操作。

```
199 clear_bss:
200         ldr r0, =bss_start
201         ldr r1, =bss_end
202         mov r2, #0x00000000
```

```
203         cmp r0, r1
204         moveq pc, lr
205
206 clbss_1:
207         str r2, [r0]
208         add r0, r0, #4
209         cmp r0, r1
210         bne clbss_1
211
212         mov pc, lr
```

注意：需要考虑没有为初始化的全局变量这一情况（即BSS段大小为0，bss_start=bss_end）

三、进入C语言环境

3.1、采用什么方式进入C语言环境

3.1.1、相对跳转(+/- 32 M)。
B、BL

3.1.2、绝对跳转
LDR PC, =address

采用绝对跳转的方式到内存中运行。

3.2、如何检验是否成功进入C语言环境
通过在C语言中点亮LED来检验C语言程序是不是成功运行。

```
1 #define GPMCON ((volatile unsigned long *) 0x7f008820)
2 #define GPMDAT ((volatile unsigned long *) 0x7f008824)
3 #define GPMPUD ((volatile unsigned long *) 0x7f008828)
4
5 int uboot_main()
6 {
7         *(GPMCON) = 0x00001111;
8         *(GPMDAT) = 0b1000;
9
10         return 0;
11 }
```

四、C与汇编混合编程

4.1、汇编语言和C语言的优缺点

4.1.1、汇编语言
执行效率高；编写繁琐。

4.1.2、C语言
可读性强，移植性好，调试方便。

4.2、混合编程的意义
1、执行效率
2、能够更直接地控制处理器

4.3、混合编程类型

4.3.1、汇编调用C函数

.S文件中

```
54        ldr pc, _uboot_main
55 @       bl led_on
56
57 _uboot_main:
58        .word uboot_main
```

.c文件中

```
5 int uboot_main()
6 {
7         *(GPMCON) = 0x00001111;
```

```
 8       *(GPMDAT) = 0b1000;
 9
10       return 0;
11 }
```

4.3.2、C调用汇编函数

.c文件中

```
 5 int uboot_main()
 6 {
 7 //      *(GPMCON) = 0x00001111;
 8 //      *(GPMDAT) = 0b1000;
 9
10       led_on();
11
12       return 0;
13 }
```

.S文件中

```
215 #define GPMCON 0x7f008820
216 #define GPMDAT 0x7f008824
217 #define GPMPUD 0x7f008828
218 .global led_on
219 led_on:
220       ldr r0, =GPMCON
221       ldr r1, =0x00001111
222       str r1, [r0]
223
224       ldr r0, =GPMDAT
225       ldr r1, =0b001010
226       str r1, [r0]
227
228       mov pc, lr
```

注意：在C中调用汇编程序函数，被调用的函数必须声明是全局函数。

4.3.3、C内嵌汇编

4.3.3.1、格式

```
__asm__(
    汇编语句部分
    :输出部分
    :输入部分
   : 破坏描述部分
);
```

C内嵌汇编以关键字”__asm__”或”asm”开始，下辖四个部分，各部分之间使用":"分开, 第一部分是必须写的，后面三部分是可以省略，但是分号:不能省略！

1.汇编语句部分：汇编语句的集合，可以包含多条汇编 语句，每条语句之间需要使用换行符 “\n”隔开或使用分号“;”隔开。

2.输出部分：在汇编中被修改的C变量列表

3.输入部分: 作为参数输入到汇编中的变量列表

4.破坏描述部分: 执行汇编指令会破坏的寄存器描述

4.3.3.2、实例

```
void write_p15_c1 (unsigned long value)
{
    __asm__(
        “mcr p15, 0, %0, c1, c0, 0\n”
       :
      : “r” (value) @编译器选择一个R*寄存器
       : "memory"
   );
}
```

%0　表示 0 号参数；
"r" (value)　中 "r" 表示汇编输入部分为寄存器，有编译器制定一个R*寄存器，而寄存器中的值有括号里面的变量赋值。

```
unsigned long read_p15_c1 (void)
{
    unsigned long value;
    __asm__(
         “mrc p15, 0, %0, c1, c0, 0\n”
        : “=r”  (value) @"="表示只写操作数，用于输出部
        :
        : "memory"
    );
    return value;
}
```

%0　表示 0 号参数；
"="　表示只写操作数，用于输出部。

4.3.3.3、优化
使用volatile来告诉编译器，不要对接下来的这部分代码进行优化。

```
unsigned long old;
unsigned long temp;
__asm__ volatile(
    "mrs %0, cpsr \n"
    "orr %1, %0, #128 \n  “
    "msr cpsr_c, %1\n"
    : "=r  “ (old), "=r  “ (temp)
    :
    : "memory"
);
```

4.3.3.4、C内嵌汇编实现点亮LED

```
13 #define GPMCON 0x7f008820
14 #define GPMDAT 0x7f008824
15 #define GPMPUD 0x7f008828
16
17 int uboot_main()
18 {
19         __asm__ volatile (
20                 "ldr r1, =0x00001111\n"
21                 "str r1, [%0]\n"
22
23                 "ldr r1, =0b001010\n"
24                 "str r1, [%1]\n"
25
26                 :
27                 :"r"(GPMCON),"r"(GPMDAT)
28                 :"r1"
29
30         );
31
32         return 0;
33 }
```