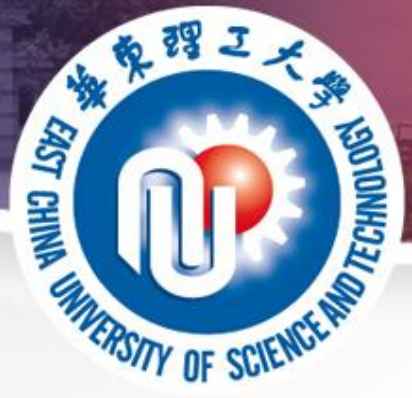


第四章 嵌入式软件编程技术

华东理工大学计算机系
罗飞

Content

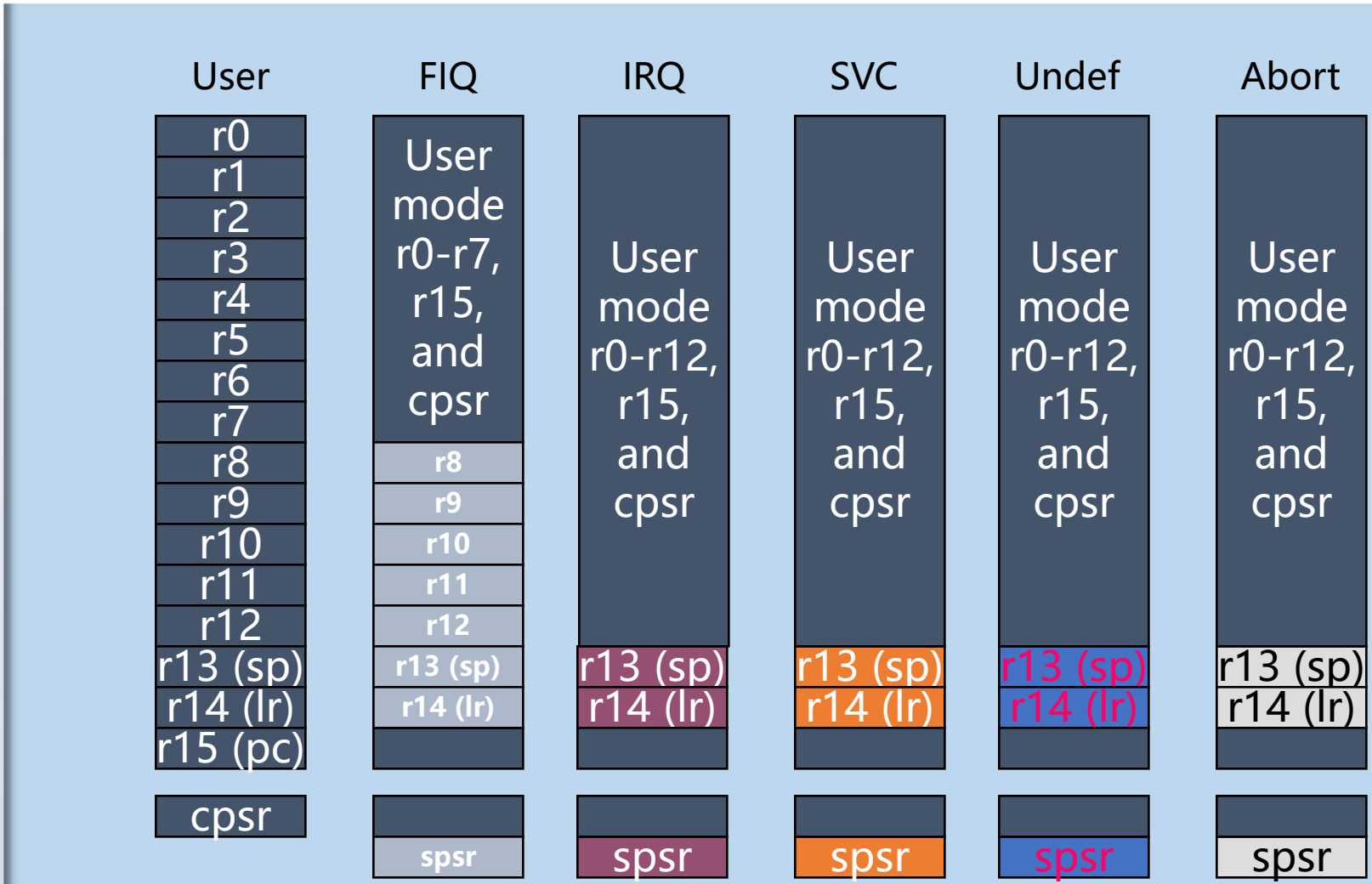
嵌入式编程基础

- 嵌入式汇编语言基础
- 嵌入式高级编程知识
- 嵌入式开发工程

汇编语言基础

- ARM寄存器组织
- ARM指令格式
- ARM寻址方式

ARM寄存器组织



注意: System模式使用user模式寄存器集

指令格式——基本格式

<opcode>{<cond>}{S} <Rd>,<Rn> {, <shift_op2>}

opcode	操作码，即指令助记符
cond	条件码，描述指令执行的条件
S	自动更新CPSR条件码标志位
Rd	目标操作数
Rn	第1操作数寄存器
opcode 2	第2操作数寄存器/立即数/ 位移运算的寄存器和立即数

高级语言程序设计概念-可重入函数

◆可重入性（reentrant)

如果某个函数可被多个任务并发调用而不会造成数据错误，则称该函数具有可重入性

可重入函数可在任意时刻被中断，稍后继续运行时不会造成错误

◆不可重入性（non-reentrant)

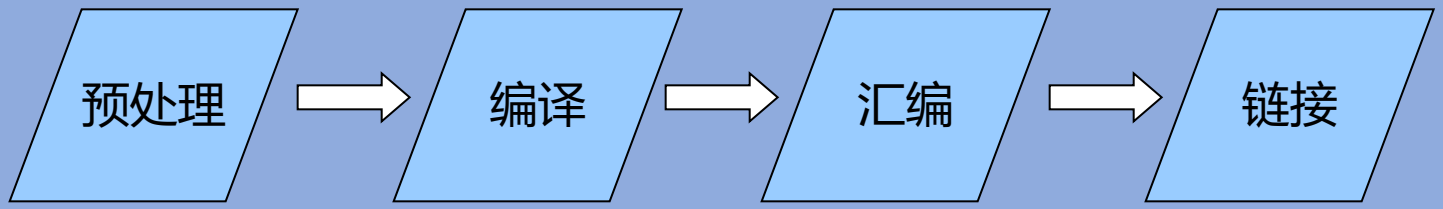
不可重入函数不能被多个任务共享，除非采用信号量等机制确保函数的互斥调用，或者在代码的关键部分禁止中断

高级语言程序设计概念-中断及处理

- 嵌入式系统开发中，经常会用到中断：硬中断、软中断、异常等。
- 标准C中不包含中断服务的自动处理。
- 许多编译器厂商增加关键字对中断服务程序的支持。

嵌入式开发工程概念-编译过程

- 在使用 GCC 编译程序时，编译过程可以被细分为四个阶段：
  - 预处理（Pre-Processing）；
  - 编译（Compiling）；
  - 汇编（Assembling）；
  - 链接（Linking）。



- Linux 程序员可以根据自己的需要让 GCC 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息
- GCC 提供了 30 多条警告信息和三个警告级别，使用它们有助于增强程序的稳定性和可移植性。

Makefile概述

- make 命令对于构建具有多个源文件的程序有很大的帮助
- 只有 make 命令还是不够的，还必须用 makefile 告诉它要做什么以及怎么做
- make 命令和 Makefile 配合使用，能给项目管理带来极大便利
- 一个 makefile 由依赖关系和规则两部分内容组成

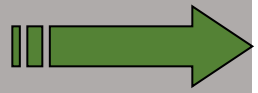
规则用来说明怎样使用所依赖的文件来建立目标文件

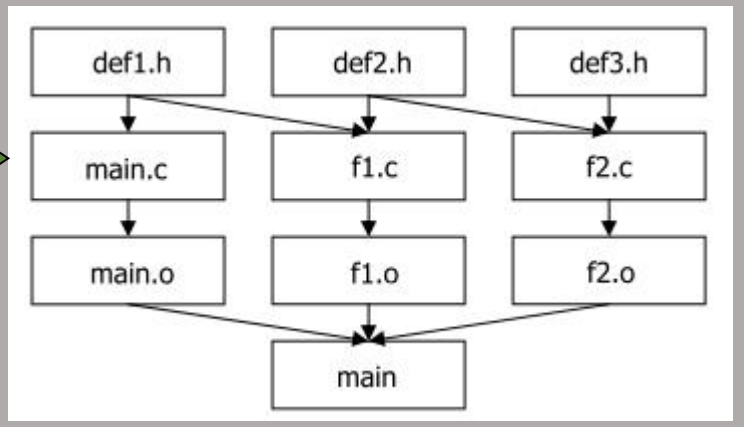
依赖关系由一个目标和一组该目标所依赖的源文件组成

makefile 中的依赖关系

- make 程序自动生成和维护通常是可执行模块或应用程序的目标，目标的状态取决于它所依赖的那些模块的状态。
- Make 的思想是为每一块模块都设置一个时间标记，然后根据时间标记和依赖关系来决定哪一些文件需要更新。一旦依赖模块的状态改变了，make 就会根据时间标记的新旧执行预先定义的一组命令来生成新的目标。

依赖关系规定了最终得到的应用程序跟生成它的各个源文件之间的关系





makefile 中的规则

- makefile 规定相应的规则来描述如何生成目标，或者说使用哪些命令来根据依赖模块产生目标。
- makefile 是以相关行为基本单位的，相关行用来描述目标、模块及规则三者之间的关系。一个相关行格式通常为：冒号左边是目标名；冒号右边是目标所依赖的模块名；紧跟着的规则是由依赖模块产生目标所使用的命令。
- 相关行的格式为：

目标：[依赖模块][;命令]

习惯上写成多行形式

目标：[依赖模块]
命令
命令

makefile

```
main.o: main.c
    gcc -c main.c
f1.o: f1.c
    gcc -c f1.c
f2.o: f2.c
    gcc -c f2.c

main: main.o f1.o f2.o
    gcc main.o f1.o f2.o –o main
```

Content

嵌入式汇编编程技术

- 基本语法
- 程序编译
- 程序例子

汇编语言程序-基本语法（1）

- ARM汇编语句格式
  - {label:} {instruction}　　{@comment}
  - {label:} {directive}{@comment}
  - {label:} {pseudo-instruction}

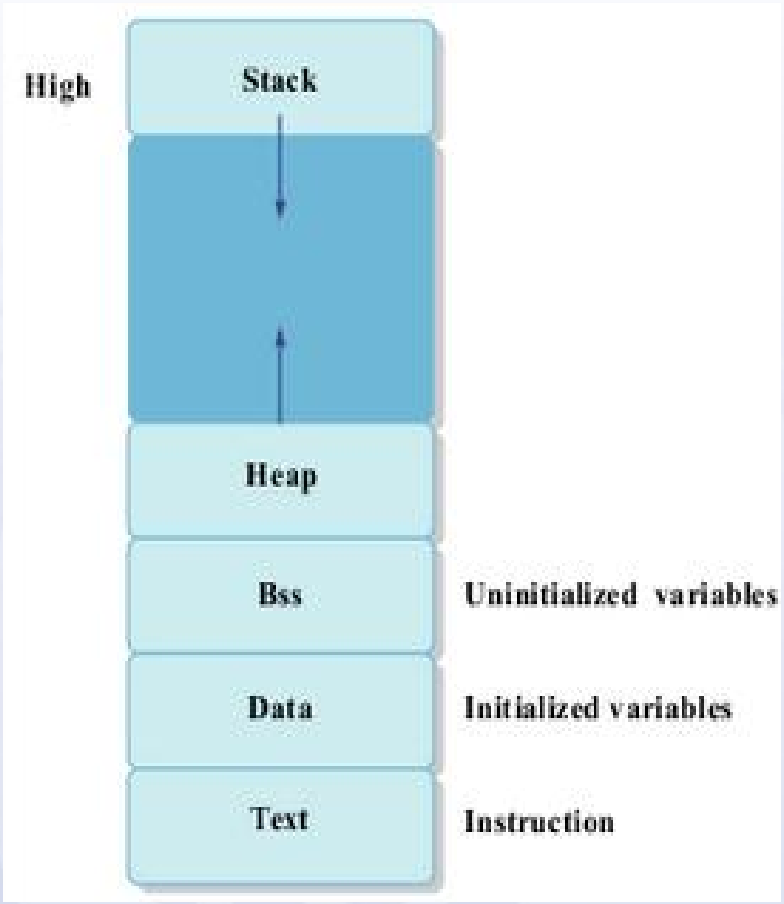
    {@comment}

汇编语言程序-基本语法（2）

- **常用的预定义寄存器名称**
- R0-R15，通用寄存器
- A1-A4，入口参数、处理结果、暂存，同R0-R3
- V1-V8，变量寄存器，同R4-R11
- IP，保存栈指针SP，同R12；SP，栈指针，同R13
- LR，链接寄存器，同R14；PC，同R15
- CPSR，当前程序状态寄存器
- SPSR，程序状态备份寄存器
- F0-F7，浮点运算加速寄存器
- S0-S31，单精度浮点寄存器
- D0-D15，双精度浮点寄存器

汇编语言程序-基本语法（3）

- 汇编程序段
  - text section
  - 代码段，只读
  - data section
  - 数据段，存放已初始化的全局变量、静态变量和常量等
  - bss section
  - block started by symbol，
  - 存放未初始化全局和静态变量



汇编语言程序-基本语法（4）

- 汇编程序设计规范
  - 符号命名规则
  - 注释
  - 其他规则

汇编语言程序-编译

- 预处理
- 汇编器
- 链接器

汇编语言程序-例子

```
.global _start
.text
_start:
        Mov R8, #20     @低32位初始化为20
        Mov R9,#0       @高32位初始化为0
        Sub R0,R8,#1    @初始化计数器
Loop:
        MOV R1,R9       @暂存高位值
        UMULL R8,R9,R0,R8       @[R9:R8]=R0*R8
        MLA R9,R1,R0,R9         @R9=R1*R0+R9
        SUBS R0，R0，#1         @计数器递减
        BNE Loop        @计数器不为0时继续循环
.Stop:
        B Stop
.end                    @文件结束
```

Content

函数可重入问题

```
static int tmp;
void swap(int *a, int *b)
{
    tmp=*a;
    *a = *b;
    *b = tmp;
}
```

```
void task1(void)
{
    …
    swap(a, b);
}

void task2(void)
{
    …
    swap(c,d);
}
```

函数可重入问题解决（1）

```
void swap(int *a, int *b)
{
    int  tmp;
    tmp=*a;
    *a = *b;
    *b = tmp;
}
```

全局变量 -> 局部变量

```
void task1(void)
{
    ...
    swap(a, b);
}

void task2(void)
{
    ...
    swap(c,d);
}
```

函数可重入问题解决（2）

```
static int tmp;
void swap(int *a, int *b)
{
    [申请信号量操作]
    tmp=*a;
    *a = *b;
    *b = tmp;
    [释放信号量操作]
}
```

```
void task1(void)
{
    …
    swap(a, b);
}

void task2(void)
{
    …
    swap(c,d);
}
```

p 函数可重入问题解决（3）

```
static int tmp;
void swap(int *a, int *b)
{
    Disalble_IRQ();
    tmp=*a;
    *a = *b;
    *b = tmp;
    Enable_IRQ()
}
```

```
void task1(void)
{
    …
    swap(a, b);
}

void task2(void)
{
    …
    swap(c,d);
}
```

中断处理过程

- 硬件部分
  - 复制CPSR到SPSR_<mode>
  - 设置正确的CPSR位
  - 切换到<mode>
  - 保存返回地址到LR_<mode>
  - 设置PC跳转到相应的异常向量表入口

中断处理过程

- 软件部分
  - 把SPSR和LR压栈
  - 把中断服务程序的寄存器压栈
  - 开中断，允许嵌套中断
  - 中断服务程序执行完后，恢复寄存器
  - 弹出SPSR和PC，恢复执行

中断处理程序和汇编

许多编译开发商在提供的标准C库中增加对中断的支持，提供新的关键字用于标识中断服务程序：自动为该函数增加中断服务程序所需要的中断现场入栈和出栈代码。

```
__irq void
IRQHandler(void)
{
    volatile unsigned int
*source = (unsigned
int*)0x80000000;
    if(*source ==1)
        int_hander_1();
    *source = 0
}
```

```
STMFD sp!, {r0-
r4,r12,lr}
MOV r4, #0x80000000
LDR r0, [r4,#0]
CMP r0, #1
BLEQ int_hander_1
MOV r0, #0
STR r0, [r4,#0]
LDMFD sp!,{r0-
r4,r12,lr}
```

Content

高级语言与低级语言混合编程

- 若汇编代码较为简洁，可使用直接采用内联汇编，即在C语言中内嵌汇编语句的方法

- 否则要将汇编程序以文件的形式加入到项目中，按照ATPCS(ARM/Thumb过程调用标准，ARM/Thumb Procedure Call Standard)的规定与C程序相互调用与访问

ATPCS基本规则

- 数据栈使用规则
  - 采用满递减类型(FD，Full Descending)，即栈通过减小存储器地址而向下增长
- 参数传递规则
  - 整数参数的前4个使用R0-R3传递，其他参数使用堆栈传递
  - 浮点参数使用编号最小且能够满足需要的一组连续的FP寄存器传递

汇编程序调用C程序的方法

- 汇编程序编写要遵循ATPCS规则，以保证程序调用时参数正确传递
- 首先在汇编程序中使用IMPORT伪指令事先声明将要调用的C语言函数
- 然后通过BL指令来调用C函数

例子

```
/* C程序函数定义 */
int add(int x,int y)
{
    return(x+y);
}
```

```
;汇编程序调用
;声明要调用的C函数
IMPORT add
……
MOV r0, 1
MOV r1, 2
;调用C函数add
BL add
```

注：使用r0和r1实现参数传递，返回结果由r0带回

C程序调用汇编程序

- 汇编程序编写也要遵循ATPCS规则，以保证程序调用时参数正确传递
- 首先在汇编程序中使用EXPORT伪指令声明被调用的子程序，表示该子程序将在其他文件中被调用
- 然后在C程序中使用extern关键字声明要调用的汇编子程序为外部函数

例子

```
  EXPORT add ;声明add子程序将被外部函数调用
  ……
  add ;求和子程序add
  ADD r0,r0,r1
  MOV pc,lr
extern int add (int x,int y); //声明add为外部函数
void main()
{
    int a=1,b=2,c;
    c=add(a,b); //调用add子程序
    ……
  }
```

内联汇编

- 可以实现一些高级语言不能实现或者不容易实现的功能
- 对于时间紧迫要求的功能也可以采用内联汇编来实现
- 支持大部分ARM指令和Thumb指令，但受操作系统限制，不支持一些底层功能的指令

例子

__asm __ (“instruction
…
instruction”); //Linux gcc中支持

asm(
汇编语句模板:
输出部分:
输入部分:
修改部分)

```
asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
```

THANKS!