

# 《嵌入式系统》

## （第四讲）

厦门大学信息学院软件工程系 曾文华

2024年10月8日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：文件系统
- 第9章：设备驱动程序设计基础
- 第10章：字符设备和驱动程序设计
- 第11章：块设备和驱动程序设计
- 第12章：网络设备驱动程序开发
- 第13章：嵌入式GUI及应用程序设计
- 第14章：Android操作系统（增加）



# 第4章 嵌入式软件编程技术

- 4.1 嵌入式编程基础
- 4.2 嵌入式汇编编程技术
- 4.3 嵌入式高级编程技术
- 4.4 高级语言与汇编语言混合编程

# 4.1 嵌入式编程基础

4.1.1 嵌入式汇编语言基础

4.1.2 嵌入式高级编程知识

4.1.3 嵌入式开发工程

## • 4.1.1 嵌入式汇编语言基础

### – ARM处理器的寄存器（37个32位的寄存器）

- 31个通用寄存器

– R0、R1、R2、R3、R4、R5、R6、R7	8个
– R8、R8_fiq	2个
– R9、R9_fiq	2个
– R10、R10_fiq	2个
– R11、R11_fiq	2个
– R12、R12_fiq	2个
– R13、R13_svc、R13_abt、R13_und、R13_irq、R13_fiq	6个
– R14、R14_svc、R14_abt、R14_und、R14_irq、R14_fiq	6个
– R15 (PC)	1个

- 6个专用状态寄存器

– CPSR	1个
– SPSR_svc、SPSR_abt、SPSR_und、SPSR_irq、SPSR_fiq	5个

- R13: SP, 堆栈指针
- R14: LR, 链接寄存器
- R15: PC, 程序计数器

CPSR: 当前程序状态寄存器

SPSR: 备份程序状态寄存器, 用于处理器进入异常模式时保存CPSR的内容, 当从异常模式退出时, 用SPSR来恢复CPSR的值

## – ARM处理器的7种运行模式：

- ① 用户模式（**USR**）
- ② 快速中断模式（**FIQ**）
- ③ 外部中断模式（**IRQ**）
- ④ 管理模式（**SVC**）
- ⑤ 数据访问中止模式（**ABT**）
- ⑥ 未定义指令中止模式（**UND**，未定义模式）
- ⑦ 系统模式（**SYS**）

## – 特权模式（除用户模式外）：

- ① 快速中断模式（**FIQ**）
- ② 外部中断模式（**IRQ**）
- ③ 管理模式（**SVC**）
- ④ 数据访问中止模式（**ABT**）
- ⑤ 未定义指令中止模式（**UND**，未定义模式）
- ⑥ 系统模式（**SYS**）

## – 异常模式（除用户模式、系统模式外）：

- ① 快速中断模式（**FIQ**）
- ② 外部中断模式（**IRQ**）
- ③ 管理模式（**SVC**）
- ④ 数据访问中止模式（**ABT**）
- ⑤ 未定义指令中止模式（**UND**，未定义模式）

## • 4.1.2 嵌入式高级编程知识

### – 1、可重入函数概念

- **可重入函数**：可以由多个任务并发使用，而不必担心数据出错。可以被中断的函数。
- **不可重入函数**：不能由多个任务所共享，除非能确保函数的互斥。不可以被中断的函数。

**可重入函数（Reentrant）**主要用于多任务环境中，一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误。

而**不可重入的函数（Non-reentrant）**由于使用了一些系统资源，比如全局变量区、中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。

# 可重入函数和不可重入函数

例子1:

```
static int tmp;
```

不可重入函数

```
void swap(int *x, int *y){
```

```
    tmp = *x;
```

```
    *x = *y;
```

```
    *y = tmp;
```

```
}
```

例子2:

```
void swap2(int *x, int *y){
```

可重入函数

```
    int tmp;
```

```
    tmp = *x;
```

```
    *x = *y;
```

```
    *y = tmp;
```

```
}
```

swap是不可重入的

有静态变量: tmp

swap2是可重入的

没有静态变量

## – 2、中断及处理概念

- 中断：

- 硬中断：计算机外设引起的事件。

- 软中断：SWI指令（软中断指令）引起的中断。

- 异常：CPU在运行过程中由其本身引起的事件（如被0除、遇到未定义的指令等）。



- **中断处理过程**包括硬件和软件两部分：

- 由**硬件**完成的工作：

- ① 复制CPSR到SPSR\_<MODE>（如复制CPSR到SPSR\_irq）。
- ② 设置正确的CPSR位。
- ③ 切换到<MODE>。
- ④ 保存返回地址到LR\_<MODE>（如LR\_irq，即R14\_irq）。
- ⑤ 设置PC跳转到相应的异常向量表入口。

**MODE：** ARM处理器的运行模式

- 由**软件**（中断服务程序）完成的工作：

- ① 把SPSR和LR压栈。
- ② 把中断服务程序的寄存器压栈。
- ③ 开中断，允许嵌套中断。
- ④ 中断服务程序执行完毕后，恢复寄存器。
- ⑤ 弹出SPSR和PC，恢复执行。

## • 4.1.3 嵌入式开发工程

- GNU **make**: 编译工具（编译命令）

- **Makefile**文件: **make**工具（命令）读取的文件

- 1、**make**工作过程

- **Makefile**文件的基本规则:

**target : prereg1 prereg2 prereg3**  
**command**

- **target**: 规则的目标

- **prereg1、prereg2、prereg3**: 规则的依赖

- **command**: 规则执行的命令，该命令行必须通过**Tab**键进行缩进

## – 2、Makefile文件示例

规则的目标

伪目标

**#Makefile Example For Math**

注释

规则的依赖

**math**:main.o display.o plus.o minus.o multi.o divide.o mod.o

gcc -o math main.o display.o plus.o minus.o multi.o divide.o mod.o

main.o: main.c def.h display.h

gcc -c main.c

display.o:display.c def.h display.h

gcc -c display.c

plus.o:plus.c def.h

gcc -c plus.c

minus.o:minus.c def.h

gcc -c minus.c

multi.o:multi.c def.h

gcc -c multi.c

divide.o:divide.c def.h

gcc -c divide.c

mod.o:mod.c def.h

gcc -c mod.c

**.PHONY**:clean

clean:

-rm main.o display.o plus.o minus.o multi.o divide.o mod.o

规则执行的命令

- ① “#Makefile Example For Math” : 以#开头的为注释行
- ② “math” : 规则的目标
- ③ “main.o display.o plus.o minus.o multi.o divide.o mod.o” : 规则的依赖
- ④ “gcc -o math main.o display.o plus.o minus.o multi.o divide.o mod.o” : 规则执行的命令
- ⑤ “main.o : main.c def.h display.h  
gcc -c main.c” : 某一个依赖的规则
- ⑥ “.PHONY : clean” : 声明clean是一个伪目标
- ⑦ “-” : 表示忽略命令的执行错误继续执行下面的语句 (-rm)

## – 3、变量定义

- 定义变量:

– **OBJS** = main.o display.o plus.o minus.o multi.o divide.o mod.o

- 则以下语句:

– math:main.o display.o plus.o minus.o multi.o divide.o mod.o

gcc -o math main.o display.o plus.o minus.o multi.o divide.o mod.o

- 可以代替为:

– math : **\$(OBJS)**

gcc -o math **\$(OBJS)**

**\$(OBJS)**



main.o display.o plus.o minus.o multi.o divide.o mod.o

- make的内置变量:

- \$\*: 没有扩展名的当前目标文件
- \$@: 当前目标文件
- \$<: 规则的第一个依赖文件名
- \$?: 比目标文件更新的依赖文件列表
- \$^: 规则的所有依赖文件列表

- 定义:

**OBJS** = main.o display.o plus.o minus.o multi.o divide.o mod.o

- 则:

**math:main.o display.o plus.o minus.o multi.o divide.o mod.o**

**gcc main.o display.o plus.o minus.o multi.o divide.o mod.o -o math**

- 可以替换为:

**math : \$(OBJS)**

**gcc \$^ -o \$@**

规则的所有依赖文件列表

当前目标文件

main.o display.o plus.o minus.o multi.o divide.o mod.o

math

# Makefile 中常见预定义变量

命令格式

含 义

AR	库文件维护程序的名称,默认值为 ar
AS	汇编程序的名称,默认值为 as
CC	C 编译器的名称,默认值为 cc
CPP	C 预编译器的名称,默认值为 \$(CC) -E
CXX	C++编译器的名称,默认值为 g++
FC	FORTTRAN 编译器的名称,默认值为 f77
RM	文件删除程序的名称,默认值为 rm -f
ARFLAGS	库文件维护程序的选项,无默认值
ASFLAGS	汇编程序的选项,无默认值
CFLAGS	C 编译器的选项,无默认值
CPPFLAGS	C 预编译的选项,无默认值
CXXFLAGS	C++编译器的选项,无默认值
FFLAGS	FORTTRAN 编译器的选项,无默认值

## – 4、Makefile规则

① 显式规则：前面介绍的规则都属于显式规则

② 隐含规则：不需要在Makefile给出的规则

– 例如，编译.c文件为.o文件的隐含规则所执行的命令：

» \$(CC) -c \$(CFLAGS)

③ 模式规则

– 使用模式符号：%

– 例如，把所有的.c文件都编译成.o文件的模式规则：

» %.o : %.c

\$(CC) \$(CFLAGS) \$< -o \$@

规则的第一个依赖文件名（%.c）

当前目标文件（%.o）



# module\_test程序的Makefile

```
Makefile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
ifndef $(KERNELRELEASE),)

KERNELDIR ?= /home/linux/workdir/fs3399/system/kernel

all: modules

modules:
    $(MAKE) -C $(KERNELDIR) M=$$PWD modules

clean:
    rm -rf *.o *.ko *.cmd modules.order Module.symvers .tmp_versions *.a *.mod*



else

    obj-m := module_test.o

endif
```

make命令  
的入口

make  
clean命令  
的入口

Local Disk (C:) > FS3399M4 > Linux > Linux驱动与应用实验源码 (发给学生) > module_test		
<input type="checkbox"/> 名称	修改日期	类型
 module_test.c	2023/8/31 11:45	C 文件
 Makefile	2024/9/20 13:01	文件

# module\_test程序的Makefile

执行make命令

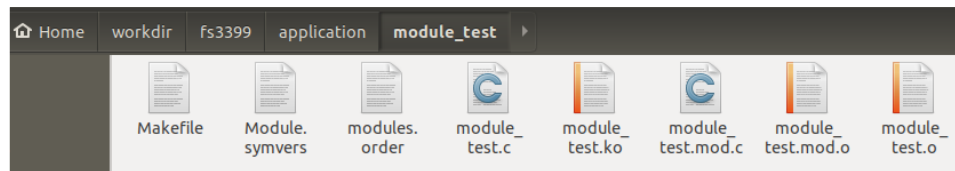
```
linux@linux-pc:~/workdir/fs3399/application/module_test$ make
make -C /home/linux/workdir/fs3399/kernel M=$PWD modules
make[1]: Entering directory '/home/linux/workdir/fs3399/kernel'
  CC [M] /home/linux/workdir/fs3399/application/module_test/module_test.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/linux/workdir/fs3399/application/module_test/module_test.mod.o
  LD [M] /home/linux/workdir/fs3399/application/module_test/module_test.ko
make[1]: Leaving directory '/home/linux/workdir/fs3399/kernel'
linux@linux-pc:~/workdir/fs3399/application/module_test$
```

得到：.ko文件

执行make clean命令

```
linux@linux-pc:~/workdir/fs3399/application/module_test$ make clean
rm -rf *.o *.ko *.cmd modules.order Module.symvers .tmp_versions *.a *.mod*
linux@linux-pc:~/workdir/fs3399/application/module_test$ ls
Makefile  module_test.c
linux@linux-pc:~/workdir/fs3399/application/module_test$
```

Local Disk (C:) > FS3399M4 > Linux > Linux驱动与应用实验源码 > module_test		
<input type="checkbox"/> 名称	修改日期	类型
Makefile	2024/8/27 16:57	文件
module_test.c	2023/8/31 11:45	C 文件



# character程序的Makefile

make命令  
的入口

make app  
命令的入口

make  
clean命令  
的入口

```
Makefile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
ifeq ($(KERNELRELEASE),)

KERNELDIR ?= /home/linux/workdir/fs3399/system/kernel

all: modules




modules:
    $(MAKE) -C $(KERNELDIR) M=$$PWD modules

app:
    aarch64-linux-gnu-gcc test.c -o test

clean:
    rm -rf *.o *.ko *.cmd modules.order Module.symvers .tmp_versions *.a *.mod* test

else
    obj-m := character.o

endif
endif
```

Local Disk (C:) > FS3399M4 > Linux > Linux驱动与应用实验源码 (发给学生) > character		
<input type="checkbox"/> 名称	修改日期	类型
 character.c	2024/9/20 13:11	C 文件
 test.c	2024/9/20 13:11	C 文件
 Makefile	2024/9/20 13:11	文件

# character程序的Makefile

执行make命令

```
linux@linux-pc:~/workdir/fs3399/application/character$ make
make -C /home/linux/workdir/fs3399/kernel M=$PWD modules
make[1]: Entering directory '/home/linux/workdir/fs3399/kernel'
CC [M] /home/linux/workdir/fs3399/application/character/character.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/linux/workdir/fs3399/application/character/character.mod.o
LD [M] /home/linux/workdir/fs3399/application/character/character.ko
make[1]: Leaving directory '/home/linux/workdir/fs3399/kernel'
linux@linux-pc:~/workdir/fs3399/application/character$
```

得到: .ko文件




执行make app命令

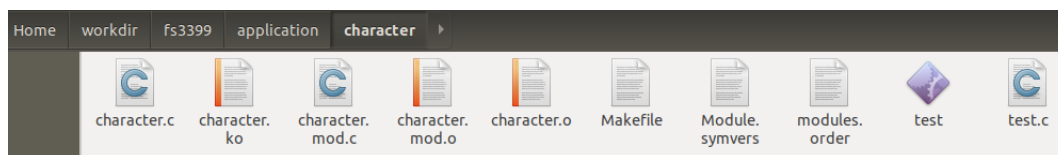
```
linux@linux-pc:~/workdir/fs3399/application/character$ make app
aarch64-linux-gnu-gcc test.c -o test
linux@linux-pc:~/workdir/fs3399/application/character$
```

得到: 可执行文件

执行make clean命令

```
linux@linux-pc:~/workdir/fs3399/application/character$ make clean
rm -rf *.o *.ko *.cmd modules.order Module.symvers .tmp_versions *.a *.mod* test
linux@linux-pc:~/workdir/fs3399/application/character$ ls
character.c Makefile test.c
linux@linux-pc:~/workdir/fs3399/application/character$
```

Local Disk (C:) > FS3399M4 > Linux > Linux驱动与应用实验源码 > character		
<input type="checkbox"/> 名称	修改日期	类型
 character.c	2024/8/27 16:44	C 文件
 Makefile	2024/8/27 16:57	文件
 test.c	2024/8/27 16:48	C 文件



# 4.2 嵌入式汇编编程技术

4.2.1 基本语法

4.2.2 汇编语言程序设计案例

- 4.2.1 基本语法

- 1、GNU汇编语言语句格式

- [**<label>:**][**<instruction or directive or pseudo-instruction>**] **@comment**

- |   |                           |       |
|---|---------------------------|-------|
| ① | <b>&lt;label&gt;:</b>     | 标号    |
| ② | <b>instruction</b>        | 指令    |
| ③ | <b>directive</b>          | 伪操作   |
| ④ | <b>pseudo-instruction</b> | 伪指令   |
| ⑤ | <b>@comment</b>           | 语句的注释 |

# ARM的指令格式

- 指令格式: `<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift_op2>}`
  - `<>`内的项是必须的, `{ }`内的项是可选的
  - **opcode**: 指令助记符 (**操作码**), 如LDR, STR 等
  - **cond**: 执行条件 (**条件码**), 如EQ, NE 等
  - **S**: 可选**后缀**, 加S时影响CPSR中的条件码标志位, 不加S时则不影响
  - **Rd**: **目标寄存器**
  - **Rn**: **第1个源操作数的寄存器**
  - **op2**: **第2个源操作数**
  - **shift**: **位移操作**

- 下面定义一个"add"的函数，最终返回两个参数的和：

定义一个段

代码段

执行段（表示这个代码段是可执行的）

全局  
变量

**.section .text, "x"**

**.global add**

**add:**

**ADD r0, r0, r1**

**MOV pc, lr**

**@ end of program**

**@ give the symbol "add" external linkage**

**@ add input arguments**

**r0 + r1 -> r0**

**@ return from subroutine**

**子程序返回**

注释

**lr: 链接寄存器**

**pc: 程序计数器**

## – 2、GNU汇编程序中的标号symbol（或label）

- 标号只能由a~z, A~Z, 0~9, “.”, \_等（由点、字母、数字、下划线等组成，除局部标号外，不能以数字开头）字符组成。
- symbol的本质：代表它所在的地址，因此也可以当作变量或者函数来使用。

• [**<label>**:[<instruction or directive or pseudo-instruction>] @comment

- 
- |   |                           |       |
|---|---------------------------|-------|
| ① | <b>&lt;label&gt;:</b>     | 标号    |
| ② | <b>instruction</b>        | 指令    |
| ③ | <b>directive</b>          | 伪操作   |
| ④ | <b>pseudo-instruction</b> | 伪指令   |
| ⑤ | <b>@comment</b>           | 语句的注释 |



## – 3、GNU汇编程序中的分段

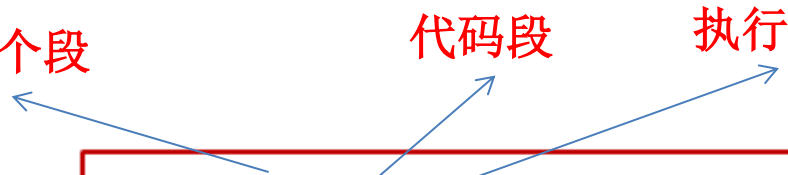
- 用户可以通过“**.section**”伪操作来自定义一个段，格式如下：

– `.section section_name [, "flags" [, %type[, flag_specific_arguments]]]`

– **flags**的含义：

- ① **a**: 允许段
- ② **w**: 可写段
- ③ **x**: 执行段

定义一个段      代码段      执行段



```
.section .text, "x"  
.global add @ give the symbol "add" external linkage  
add:  
    ADD r0, r0, r1 @ add input arguments  
    MOV pc, lr @ return from subroutine  
@ end of program
```

- 例如，定义一个“段”：

自定义  
数据段

```
.section .mysection      @自定义数据段，段名为“.mysection”  
  
.align 2  
strtemp:  
  
    .ascii "Temp string \n\0"  
  
@ 将"Temp string \n\0"这个字符串存储在以标号strtemp为起始地  
址的一段内存空间里
```

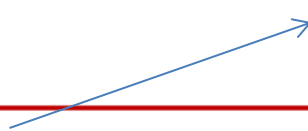
**.align:** 对齐方式伪操作

- 汇编系统预定义的段名：

- ① **.text** @代码段
- ② **.data** @初始化数据段
- ③ **.bss** @未初始化数据段
- ④ **.sdata** @短数据的初始化数据段
- ⑤ **.sbss** @短数据的未初始化数据段

– 注意：源程序中**.bss**段应该在**.text**段之前。

代码段



```
.section .text, "x"  
.global add                @ give the symbol "add" external linkage  
add:  
    ADD r0, r0, r1          @ add input arguments  
    MOV pc, lr              @ return from subroutine  
@ end of program
```

## – 4、GNU汇编语言定义入口点

- 汇编程序的缺省入口是 **`__start`** 标号，用户也可以在连接脚本文件中用 **ENTRY** 标志指明其它入口点。
- 例如，定义入口点：

```
.section .data
    < initialized data here      初始化数据段>

.section .bss
    < uninitialized data here    未初始化数据段>

.section .text      代码段

.globl __start
__start:
    <instruction code goes here  代码段>
```

## – 5、GNU汇编程序中的常数

- ① 十进制数以非0数字开头，如：**123**和**9876**；
- ② 二进制数以**0b**开头，其中字母也可以为大写；
- ③ 八进制数以**0**开始，如：**0456**，**0123**；
- ④ 十六进制数以**0x**开头，如：**0xabcd**，**0X123f**；
- ⑤ 字符串常量需要用引号括起来，中间也可以使用转义字符，如：“**You are welcome!\n**”；
- ⑥ 当前地址以“**.**”表示，在GNU汇编程序中可以使用这个符号代表当前指令的地址；
- ⑦ 表达式：在汇编程序中的表达式可以使用常数或者数值，“**-**”表示取负数，“**~**”表示取补，“**<>**”表示不相等，其他的符号如：**+**、**-**、**\***、**/**、**%**、**<**、**<<**、**>**、**>>**、**|**、**&**、**^**、**!**、**==**、**>=**、**<=**、**&&**、**||**，跟C语言中的用法相似。

## — 6、GNU ARM汇编的常用伪操作

- 数据定义伪操作:

- ① `.byte`
- ② `.short`
- ③ `.long`
- ④ `.quad`
- ⑤ `.float`
- ⑥ `.string/.asciz/.ascii`

`.byte`: 定义一个字节, 并为之分配空间

`.short`: 定义一个短整型, 并为之分配空间

`.long`: 定义一个长整型, 并为之分配空间

`.quad`: 定义一个1~8字节的长整数

`.float`: 分配一段字内存单元, 并用32位IEEE单精度浮点数`expr`初始化内存单元

`.string`: 将字符串拷贝到目标文件中, 串以0结尾

`.asciz`: 定义一个或多个字符串并为之分配空间, 字符串后自动加0结尾

`.ascii`: 定义一个或多个字符串并为之分配空间, 字符串后不自动加0结尾

- 函数的定义伪操作

- 其他常用伪操作:

- ① `.align`: 对齐方式伪操作
- ② `.end`: 源文件结束伪操作
- ③ `.include`: 可以将指定的文件在使用`.include`的地方展开, 一般是头文件
- ④ `.incbin`: 将原封不动的一个二进制文件编译到当前文件中
- ⑤ `.global/.globl`: 用来定义一个全局的符号
- ⑥ `.type`: 用来指定一个符号的类型是函数类型或者是对象类型

## • 4.2.2 汇编语言程序设计案例

### – 例1: 20的阶乘

- $20 \times 19 \times 18 \times \dots \times 2 \times 1 = 2,432,902,008,176,640,000$
- 用ARM汇编语言设计程序实现求 20!（20的阶乘），并将其64位的结果放在[R9:R8]中（R9中放置高32位，R8中放置低32位）

$$2^{64} = 18,446,744,073,709,551,616$$

$$-9,223,372,036,854,775,808 \sim +9,223,372,036,854,775,807$$

- 程序设计思路：64位结果的乘法指令通过两个32位的寄存器相乘，可以得到64位的结果，在每次循环相乘中，我们可以将存放64位结果两个32位寄存器分别与递增量相乘，最后将得到的高32位结果相加。

• 程序代码如下：

伪操作

**.global \_start**

@声明全局变量 “\_start”

**.text**

@代码段

**\_start:**

**MOV R8, #20**

@低32位初始化为20

**MOV R9, #0**

@高32位初始化为0 [R9:R8]=20

**SUB R0, R8, #1**

@初始化计数器 R0=19

**Loop:**

**MOV R1, R9**

@暂存高位值

**UMULL R8, R9, R0, R8**

@[R9:R8]=R0\*R8 乘法指令

**MLA R9, R1, R0, R9**

@R9=R1\*R0+R9 乘加指令

**SUBS R0, R0, #1**

@计数器递减

**BNE Loop**

@计数器不为0，继续循环

**Stop:**

**B Stop**

ARM指令

伪操作

**.end**

@源文件结束（程序结束）



- 例2：设计一段程序完成**数据块的复制**，数据从源数据区复制到目标数据区，要求以4个字为单位进行复制，最后所剩不到4个字的数据，以字为单位进行复制。

- 程序代码如下：

伪操作

```
.global _start           @声明全局变量 “_start”  
equ NUM, 18             @设置要拷贝的字数  
.text                   @代码段  
.arm                    @ARM程序  
_start:
```

**LDR R0, =SRC**

@R0指向源数据区起始地址

**LDR R1, =DST**

@R1指向目的数据区起始地址

**MOV R2, #NUM**

@ R2存放待复制数据量大小，  
以字为单位

**MOV SP, #0X9000**

@堆栈指针指向0X9000，  
堆栈增长模式由装载指令的  
类型域确定

**MOV R3, R2, LSR #2**

@ 将R2中值除以4后的结果存放在R3，  
R3中值表示NUM中有多少个4字单元

**BEQ COPY\_WORDS**

@ 若Z=1(R3=0，数据少于1个4字单元)，  
则跳转到COPY\_WORDS处，  
运行少于4字单元数据处理程序

**STMFD SP!, {R5-R8}**

@保存R5-R8的内容到堆栈，并更新栈指针，

批量存数指令

FD:满递减堆栈，由此可知堆栈长向

## COPY\_4WORD:

LDMIA R0!, {R5-R8}

批量取数指令

STMIA R1!, {R5-R8}

批量存数指令

SUBS R3, R3, #1

BNE COPY\_4WORD

LDMFD SP!, {R5-R8}

批量取数指令

@从R0所指的源数据区装载4个字数据到R5-R8中，  
每次装载1个字后R0中地址加1，  
最后更新R0中地址

@将R5-R8的4个字数据存入R1所指的目的地数据区，  
每次装载1个字后R1中地址加1，  
最后更新R1中地址

@每复制一次，则R3=R3-1，  
表示已经复制了1个4字单元，结果影响CPSR

@若CPSR的Z=0(即运算结果R3不等于0)，  
跳转到COPY\_4WORD，  
继续复制下一个4字单元数据

@将堆栈内容恢复到R5-R8中，并更新堆栈指针，  
此时整4字单元数据已经复制完成，  
且出栈模式应和入栈模式一样

### **COPY\_WORDS:**

**ANDS R2,R2, #3**

**BEQ STOP**

### **COPY\_WORD:**

**LDR R3, [R0], #4**

**STR R3, [R1], #4**

**SUBS R2, R2, #1**

**BNE COPY\_WORD**

@得到NUM除以4后余数，  
即未满4字单元数据的字数(1个字=4个字节)  
@若R2=0(NUM有整数个4字单元)，  
则停止复制

@将R0所指源数据区的4个字节(1个字)  
数据装载至R3，然后R0=R0+4

@将R3中4个字节(1个字)数据存到  
R1所指目的数据区，然后R1=R1+4

@数据传输控制计数器减1(其总是小于4)，  
成功复制一个字数据

@若R2不等于0，则转到WORDCOPY，  
继续复制下一个字数据

第一次复制: 1,2,3,4

第二次复制: 5,6,7,8

第三次复制: 9, 0xa,0xb,0xc

第四次复制: 0xd,0xe,0xf,0x10

第五次复制: 0x11

第六次复制: 0x12

**STOP:**

**B STOP**

伪操作

**.ltorg**

@声明一个数据缓冲池的开始，  
一般在代码的最后面

**SRC:**

**.long 1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,  
0xd,0xe,0xf,0x10,0x11,0x12**

@18个源数据

**DST:**

**.long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0**

@18个目的数据

伪操作

**.end**

@程序结束

# 4.3 嵌入式高级编程技术

4.3.1 函数可重入

4.3.2 中断处理过程

## • 4.3.1 函数可重入

– 针对函数可重入问题，有以下几种优化解决方案：

- 1、将全局变量或静态变量改成局部变量

– 优化前：

```
static int tmp;  
void swap(int *x, int *y){  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

不可重入函数

– 优化后：

```
void swap(int *x, int *y){  
    int tmp;  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

可重入函数

- 2、采用信号量进行临界资源保护

– 优化前:

```
static int tmp;  
void swap(int *x, int *y){  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

不可重入函数

– 优化后:

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

可重入函数

### • 3、禁止中断

– 优化前:

不可重入函数

```
static int tmp;  
void swap(int *x, int *y){  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

– 优化后:

可重入函数

```
static int tmp;  
void swap(int *x, int *y){  
    Disable_IRQ();  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
    Enable_IRQ();  
}
```

禁止中断

允许中断



## • 4.3.2 中断处理过程

- **\_\_irq**: 中断处理函数声明关键字

✓ irq: 外部中断模式

- 中断处理函数的C语言代码:

```
__irq void IRQHandler(void)
{
    volatile unsigned int *source = (unsigned int*) 0x80000000;
    if(*source == 1)    判断地址为0x80000000端口的值是不是为1
        int_hander_1();    //中断服务程序
    *source = 0;    将地址为0x80000000端口的值置为0
}
```

- 中断处理函数的汇编代码:

```
STMFD sp!, {r0-r4, r12, lr}
```

```
MOV r4, #0x80000000
```

```
LDR r0, [r4, #0]
```

判断地址为0x80000000端口的值是不是为1

```
CMP r0, #1
```

```
BLEQ int_hander_1
```

@转中断服务程序入口

```
MOV r0, #0
```

```
STR r0, [r4, #0]
```

将地址为0x80000000端口的值置为0

```
LDMFD sp!, {r0-r4, r12, lr}
```

```
SUBS pc, lr, #4
```

# 4.4 高级语言与汇编语言混合编程

4.4.1 高级语言与汇编语言混合编程概述

4.4.2 汇编程序调用C程序

4.4.3 C程序调用汇编程序

## • 4.4.1 高级语言与汇编语言混合编程概述

- 大部分程序采用嵌入式**C语言**编写；对硬件相关的操作、中断处理、对性能要求比较高的模块，需要用**汇编语言**编写程序。
- C语言调用汇编语言；汇编语言调用C语言。
- **ATPCS**：过程调用标准（ARM-THUMB Procedure Call Standard）。
- **AAPCS**：新的过程调用标准（ARM Architecture Produce Call Standard）。
- C语言与汇编语言之间的调用需遵循过程调用标准。

- **ATPCS（过程调用标准）中寄存器的使用规则：**
  - ✓ 子程序之间通过寄存器**R0-R3**来传递参数，当参数个数多于**4**个时，使用堆栈来传递参数；此时**R0-R3**可记作**A1-A4**。
  - ✓ 在子程序中，使用寄存器**R4-R11**保存局部变量；因此当进行子程序调用时要注意对这些寄存器的保存和恢复；此时**R4-R11**可记作**V1-V8**。
  
- **ATPCS（过程调用标准）中数据栈的使用规则：**
  - ✓ **FD（Full Decrease）**型堆栈，即满递减堆栈。
  
- **ATPCS（过程调用标准）中参数传递的规则：**
  - ✓ 整数参数的前**4**个使用寄存器**R0~R3**来传递参数，其他参数使用数据栈来传递参数。
  - ✓ 浮点参数使用编号最小且能满足需要的一组连续的**FP**寄存器传递。
  
- **ATPCS（过程调用标准）中子程序返回结果的规则：**
  - ✓ 如果结果为一个**32**位的整数，可以通过寄存器**R0**返回。
  - ✓ 如果结果为一个**64**位整数，可以通过寄存器**R0**和**R1**返回。
  - ✓ 如果结果为一个浮点数，可以通过浮点运算的寄存器**f0**、**d0**或**s0**返回。

## • 4.4.2 汇编程序调用C程序

- 汇编程序的设计要遵守**ATPCS**（过程调用标准），保证程序调用时参数的正确传递。在汇编程序中使用**IMPORT伪操作**声明将要调用的C程序，然后通过**BL指令**调用C函数。
- **IMPORT伪操作**告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号，而且不论本源文件是否实际引用该符号，该符号都将被加入到本源文件的符号表中。

– 例子:

C  
语  
言  
程  
序

```
int add(int x, int y)
{
    return(x+y);
}
```

**IMPORT伪操作**告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的

**IMPORT add**

@声明要调用的C函数

**MOV r0, 1**

**MOV r1, 2**

@通过r0、r1传递参数（参数传递规则）

**BL add**

@调用C函数add；返回结果由r0带回  
（子程序返回结果规则）

程序执行完后， $r0 = 1 + 2 = 3$

汇  
编  
语  
言  
调  
用  
C  
语  
言  
程  
序

# 第1次实验的汇编程序调用C程序

按Reset键后，从这里开始执行

调用C语言的main函数

```
startup_stm32f407xx.s - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

; Reset handler
Reset_Handler PROC
    EXPORT Reset_Handler            [WEAK]
    IMPORT SystemInit
    → IMPORT __main

    LDR    R0, =SystemInit
    BLX    R0
    → LDR    R0, =__main
    BX     R0
ENDP
```

Local Disk (C:) > FS3399M4 > STM32 > STM32实验源码 > 1\_Led > MDK-ARM

名称	修改日期	类型
1_Marquee	2024/8/27 15:01	文件夹
DebugConfig	2024/8/27 15:01	文件夹
RTE	2024/8/27 15:01	文件夹
1_Marquee.uvguix.lily2002	2024/8/27 15:07	LILY2002 文件
1_Marquee.uvoptx	2024/8/27 15:07	UVOPTX 文件
1_Marquee.uvprojx	2024/8/27 15:01	礦ision5 Project
startup_stm32f407xx.lst	2024/8/27 15:01	LST 文件
→ startup_stm32f407xx.s	2024/8/27 17:43	S 文件

Local Disk (C:) > FS3399M4 > STM32 > STM32实验源码 > 1\_Led > Src

名称	修改日期
gpio.c	2024/8/10 11:34
→ main.c	2024/8/9 10:45
stm32f4xx_hal_msp.c	2024/7/22 10:32
stm32f4xx_it.c	2024/7/22 10:34
system_stm32f4xx.c	2024/8/21 9:04

# 第2次实验的汇编程序调用C程序

汇编程序的开始

```
start.S - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
#include "macro.h"

.globl _start
_start:
    b         reset
    .align 3
reset:
    #if 1
        switch_el x1, 3f, 2f, 1f
3:      msr     vbar_el3, x0
        mrs     x0, scr_el3
        orr     x0, x0, #0xf          /* SCR_EL3.NS||IRQ|FIQ|EA */
        msr     scr_el3, x0
        msr     cptr_el3, xzr        /* Enable FP/SIMD */
        ldr     x0, =240000000
        msr     cntfrq_el0, x0      /* Initialize CNTFRQ */
        b      0f
2:      msr     vbar_el2, x0
        mov     x0, #0x33ff
        msr     cptr_el2, x0        /* Enable FP/SIMD */
        b      0f
1:      msr     vbar_el1, x0
        mov     x0, #3 << 20
        msr     cpacr_el1, x0      /* Enable FP/SIMD */
0:      bl      lowlevel_init
        branch_if_master x0, x1, master_cpu
slave_cpu:
    wfe
    ldr     x1, =0x41c00000
    ldr     x0, [x1]
    cbz     x0, slave_cpu
    br      x0                      /* branch to the given address */
master_cpu:
    #endif
_main:
    mov     x0, #1;
    mov     x1, #0;
    bl      main
    b       _main
```

调用C语言的main函数

Local Disk (C:) > FS3399M4 > ARM > ARM裸机实验源码 > 02-fs\_pwm > start

名称

修改日期

2024/7/9 17:21

start.S

Local Disk (C:) > FS3399M4 > ARM > ARM裸机实验源码 > 02-fs\_pwm >

名称

common

include

output

source

start

main.c

Makefile

map.lds

rk3399.init



## • 4.4.3 C程序调用汇编程序

### – 两种形式：

- ① 嵌入式汇编
- ② 内联汇编

### – 1、嵌入式汇编

- 在汇编程序中使用**EXPORT伪指令**声明被调用的子程序，表明该子程序将在其他文件中被调用。
- 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。
- **EXPORT伪指令**用于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。
- **extern关键字**可置于变量或者函数前，以表示变量或者函数的定义在别的文件中。

# 嵌入式汇编

- 例子:

- 汇编语言:

汇编语言程序

```
EXPORT add      @声明add子程序将被外部函数调用
add:
    ADD r0,r0,r1    @r0+r1 -> r0
    MOV pc,lr       @子程序返回
```

- C语言:

C语言调用汇编语言程序

```
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);
}
```

## – 2、内联汇编

- 内联汇编：即将汇编语句直接写在C程序中，有两种形式
- 内联汇编的形式1：

C  
语言  
调用  
汇编  
语言  
程序

```
void enable_IRQ(void)
```

```
{
```

```
    int tmp;
```

```
    __asm
```

//声明内联汇编代码

```
{
```

```
    MRS tmp, CPSR
```

```
    BIC tmp, tmp, #0x80
```

```
    MSR CPSR_c, tmp
```

```
}
```

```
}
```

内联汇编形式1

- 内联汇编的形式2:

- 格式:

```
__asm(  
    code  
    : 输出操作数列表  
    : 输入操作数列表  
    : clobber列表（破坏列表）  
);
```

## 内联汇编形式2

- 例子:

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

输入操作数列表

code（代码）

输出操作数列表

– 调用例子:

C  
语言  
调用  
汇编  
语言  
程序

## 内联汇编形式2

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int result,value;
```

```
    value = 1;
```

result

```
    printf("old value is %x",value);
```

value

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

```
    printf("new value is %x\n",result);
```

```
    return 1;
```

```
}
```

mov result, value, ror #1

value(00000001H)循环右移1位, 结果为: 10000000H

# 小结

- 嵌入式汇编编程技术。
- 嵌入式高级编程：函数可重入、中断处理。
- 高级语言和汇编语言的混合编程技术。

# 进一步探索

- 了解在Windows操作系统交叉开发环境下ARM汇编和C语言的混合编程技术。
- 了解PowerPC或MIPS嵌入式处理器架构下，汇编编程技术和混合编程技术。

**Thanks**