国嵌

www.enjoylinux.cn

ARM程序设计

Contents

Contents

- ARM指令集
- ARM伪指令
- C/C++混合编程
- ADS集成开发环境

Contents

ARM

什么是ARM?

ARM（Advanced RISC Machines）

既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

ARM

1990年ARM公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用ARM技术知识产权（IP）核的微处理器，即我们通常所说的ARM微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于ARM技术的微处理器应用约占据了32位RISC微处理器75％以上的市场份额，ARM技术正在逐步渗入到我们生活的各个方面。

ARM

ARM公司是专门从事基于RISC技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从ARM公司购买其设计的ARM微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的ARM微处理器芯片进入市场。

ARM

据最新统计，全球有103家巨型IT公司在采用ARM技术，20家最大的半导体厂商中有19家是ARM的用户，包括德州仪器，意法半导体，Philips，Intel等。ARM系列芯片已经被广泛的应用于移动电话、手持式计算机以及各种各样的嵌入式应用领域，成为世界上销量最大的32位微处理器。

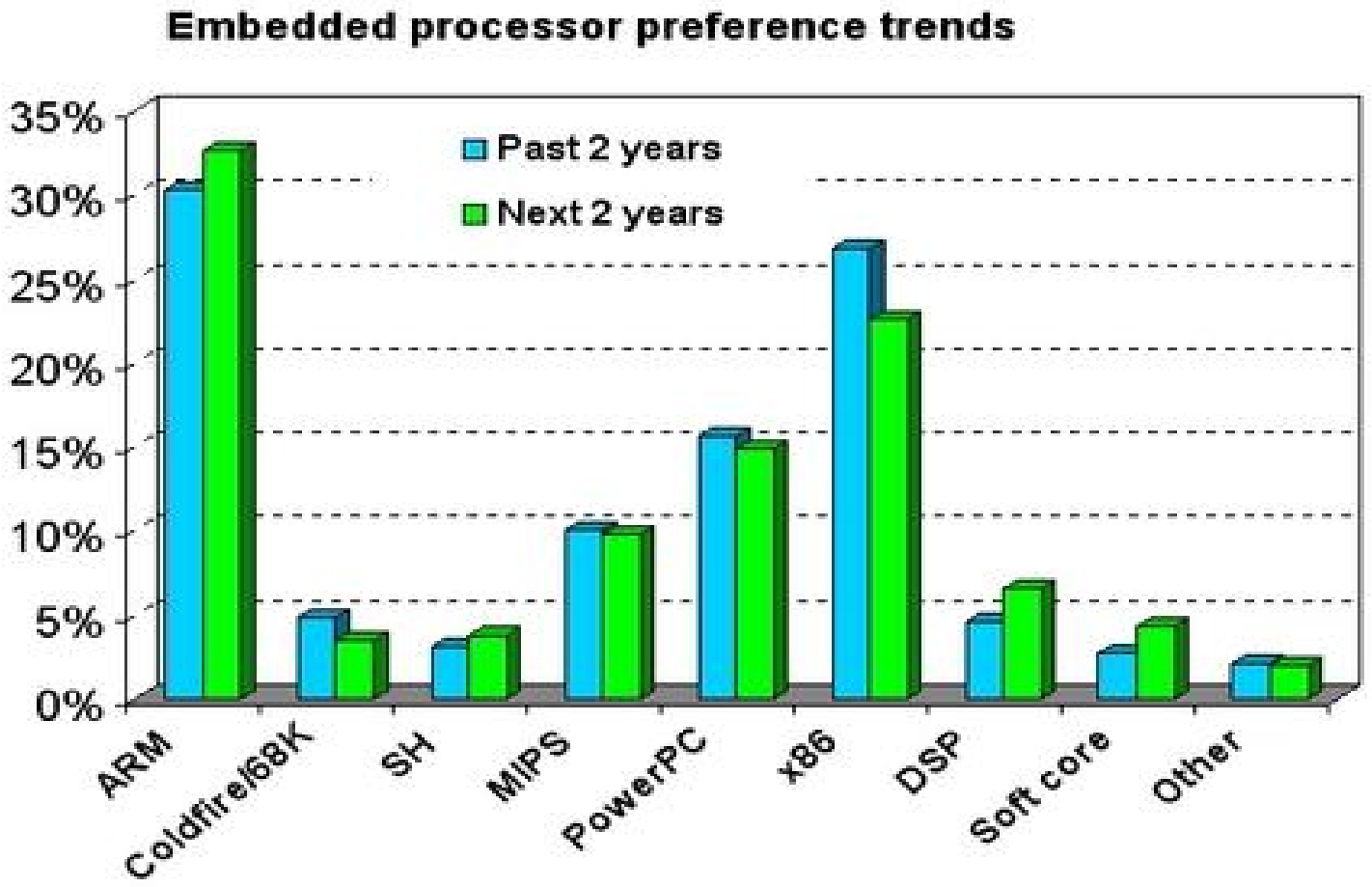
ARM

www.enjoylinux.cn

嵌入式Linux技术咨询QQ号：550491596

嵌入式Linux学习交流QQ群：65212116

ARM



发展历程

- 第一片ARM处理器是1983年10月到1985年4月间在位于英国剑桥的Acorn Computer公司开发 。
- 1990年，为广泛推广ARM技术而成立了独立的公司 。
- 20世纪90年代， ARM快速进入世界市场 。

发展历程

- 在ARM的发展历程中，从ARM7开始，ARM核被普遍认可和广泛使用 。
- 1995年 StrongARM 问世 。
- XScale是下一代StrongARM芯片的发展基础 。
- ARM10TDMI是ARM处理器核中的高端产品 。
- ARM11是ARM家族中性能最强的一个系列 。

发展历程

- 最近10多年来ARM技术的突出成果表现在:
- ARM9、ARM10、Strong-ARM和ARM11等系列处理器的开发，显著地提高了ARM的性能，使得ARM技术在面向高端数字音、视频处理等多媒体产品的应用中更加广泛;
- 更好的软件开发和调试环境，加快用户产品开发;
- 更为广泛的产业联盟使得基于ARM的嵌入式应用领域更加广阔;

发展历程

www.enjoylinux.cn

England
Cambridge, Maidenhead, Sheffield, Blackburn
Germany
Munich
France
Paris, Sophia Antipolis
Korea
Seoul

US
Seattle, Los Gatos, Walnut Creek, Austin, Boston, San Diego
China
Taiwan and Shanghai
Japan
Shin-Yokohama (Tokyo)

全球雇员
800+

应用领域

Diamond Multimedia Rio 600

Samsung ML5100A

JVC "Pixstar" GC-X1

Alba Bush
Internet TV

Nintendo
Gameboy
Advance

Iomega HipZip

Ericsson
R380

HP CapShare

Lexmark Z52 Color Jetprinter

HP Jornado 820

Nokia 8810

Nokia Mediamaster

应用领域

1、工业控制领域:

基于ARM核的微控制器不但占据了高端微控制器市场的大部分份额,同时也逐渐向低端微控制器应用领域扩展，ARM微控制器的低功耗、高性价比，向传统的8位/16位微控制器（单片机）提出了挑战。

应用领域

2、无线通讯领域:

目前已有超过85%的无线通讯设备采用了ARM技术, ARM以其高性能和低成本, 在该领域的地位日益巩固。

应用领域

3、网络应用：

随着宽带技术的推广,采用ARM技术的ADSL芯片正逐步获得竞争优势。此外，ARM在语音及视频处理上行了优化,并获得广泛支持，也对DSP的应用领域提出了挑战。

应用领域

4、消费类电子产品:

ARM技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到广泛采用。

应用领域

5、成像和安全产品:

现在流行的数码相机和打印机中绝大部分采用ARM技术。

除此以外，ARM微处理器及技术还应用到许多不同的领域，并会在将来取得更加广泛的应用。

特点

1、体积小、低功耗、低成本、高性能

2、支持Thumb（16位）/ARM（32位）

双指令集,能很好的兼容8位/16位器件

3、大量使用寄存器,指令执行速度更快

4、寻址方式灵活简单,执行效率高

系列

- ARM7系列
- ARM9系列
- ARM9E系列
- ARM10E系列
- ARM11系列
- SecurCore系列
- Inter的Xscale
- Inter的StrongARM

ARM7

ARM7系列微处理器的主要应用领域为：工业控制、Internet设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM7系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中,ARM7TMDI是目前使用最广泛的32位嵌入式RISC处理器,属低端ARM处理器核。

ARM9

ARM9系列微处理器主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数字照相机和数字摄像机等。ARM9系列微处理器包含ARM920T、ARM922T和ARM940T三种类型。

ARM9E

ARM9E系列微处理器为综合处理器,提供了增强的DSP处理能力,很适合于那些需要同时使用DSP和微控制器的应用场合，如下一代无线设备、数字消费品、成像设备、工业控制、存储设备和网络设备等领域。ARM9E系列微处理器包含ARM926EJ-S、ARM946E-S和ARM966E-S三种类型。

ARM10E

www.enjoylinux.cn

ARM10E系列微处理器具有高性能、低功耗的特点,由于采用了新的体系结构，与同等的ARM9器件相比较，在同样的时钟频率下，性能提高了近50％，同时ARM10E系列微处理器采用了先进的节能方式，使其功耗极低。ARM10E系列微处理器主要应用于下一代无线设备、数字消费品、成像设备、工业控制、通信和信息系统等领域。ARM10E系列微处理器包含ARM1020E、ARM1022E和ARM1026EJ-S三种类型。

ARM11

ARM11

ARM11系列微处理器是ARM公司近年推出的新一代RISC处理器，它在性能上展示了巨大的提升，首先推出350M-500MHz时钟频率的内核，在未来将上升到1GHz时钟频率。 ARM11处理器在提供高性能的同时，也允许在性能和功耗间做权衡以满足某些特殊应用，通过动态调整时钟频率和供应电压，开发者完全可以控制这两者的平衡。ARM11系列主要有ARM1136J，ARM1156T2和ARM1176JZ（iphone）三个型号。

Xscale

Xscale 处理器是基于ARMv5TE体系结构的解决方案，是一款全性能、高性价比、低功耗的处理器。它已使用在数字移动电话、个人数字助理和网络产品等场合。Xscale 处理器是Inter目前主要推广的一款ARM微处理器。

StrongARM

Inter StrongARM 处理器是采用ARM体系结构高度集成的32位RISC微处理器。它融合了Inter公司的设计和处理技术以及ARM体系结构的电源效率，采用在软件上兼容ARMv4体系结构、同时采用具有Intel技术优点的体系结构。

Intel StrongARM处理器是便携式通讯产品和消费类电子产品的理想选择，已成功应用于多家公司的掌上电脑系列产品。

Contents

mini2440

www.enjoylinux.cn

mini2440是国内开发板第一品牌友善之臂旗下的一款性价比很高的ARM开发板，它采用Samsung S3C2440 为微处理器，并采用专业稳定的CPU 内核电源芯片和复位芯片来保证系统运行时的稳定性。mini2440 的PCB 采用沉金工艺的四层板设计，专业等长布线，保证关键信号线的信号完整性，生产采用机器贴片，批量生产。

mini2440

www.enjoylinux.cn

嵌入式Linux技术咨询QQ号: 550491596

嵌入式Linux学习交流QQ群: 65212116

硬件资源

1. CPU 处理器

- Samsung S3C2440A，主频400MHz，最高533Mhz

2. SDRAM 内存

- 在板64M SDRAM
- 32bit 数据总线
- SDRAM 时钟频率高达100MHz

硬件资源

3. FLASH 存储

- 128M Nand Flash, 掉电非易失
- 2M Nor Flash，掉电非易失

4. LCD 显示

- 标准配置为NEC 256K 色240x320/3.5 英寸TFT 真彩液晶屏，带触摸屏;
- 板上引出一个12V 电源接口，可以为大尺寸TFT 液晶的12V CCFL 背光模块(Inverting) 供电。

硬件资源

5. 接口

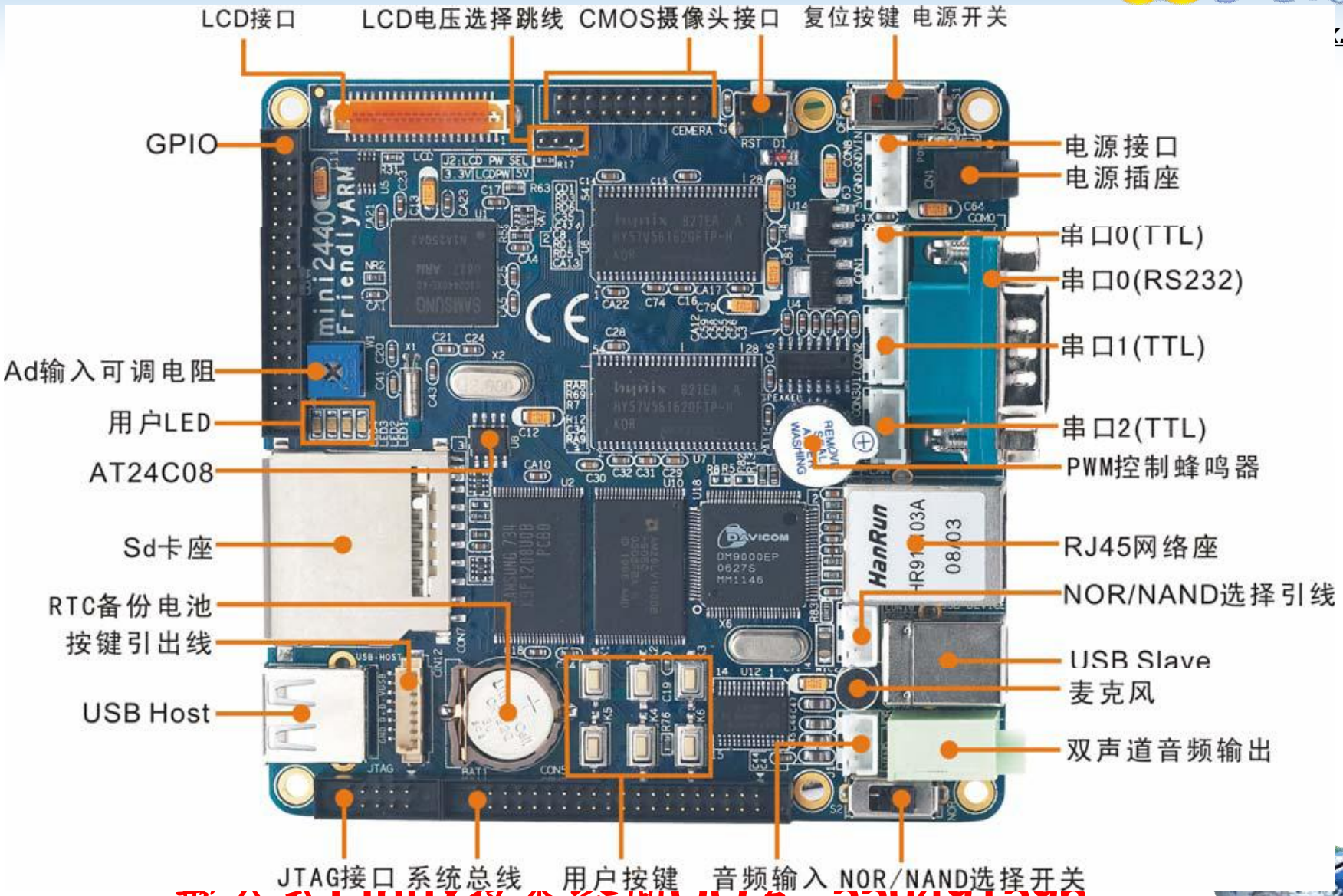
- 1 个100M 以太网RJ-45 接口(采用DM9000 网络芯片)
- 3 个串行口
- 1 个USB Host
- 1 个USB Slave 接口
- 1 个SD 卡存储接口
- 1 路立体声音频输出接口，一路麦克风接口
- 1 个2.0mm 间距10 针JTAG 接口
- 4 USER Leds

硬件资源

5. 接口

- 6 USER buttons
- 1个PWM 控制蜂鸣器
- 1个可调电阻，用于AD 模数转换测试
- 1个I2C 总线AT24C08 芯片，用于I2C 总线测试
- 1个2.0 mm 间距20pin 摄像头接口
- 板载实时时钟电池
- 电源接口(5V)，带电源开关和指示灯

布局

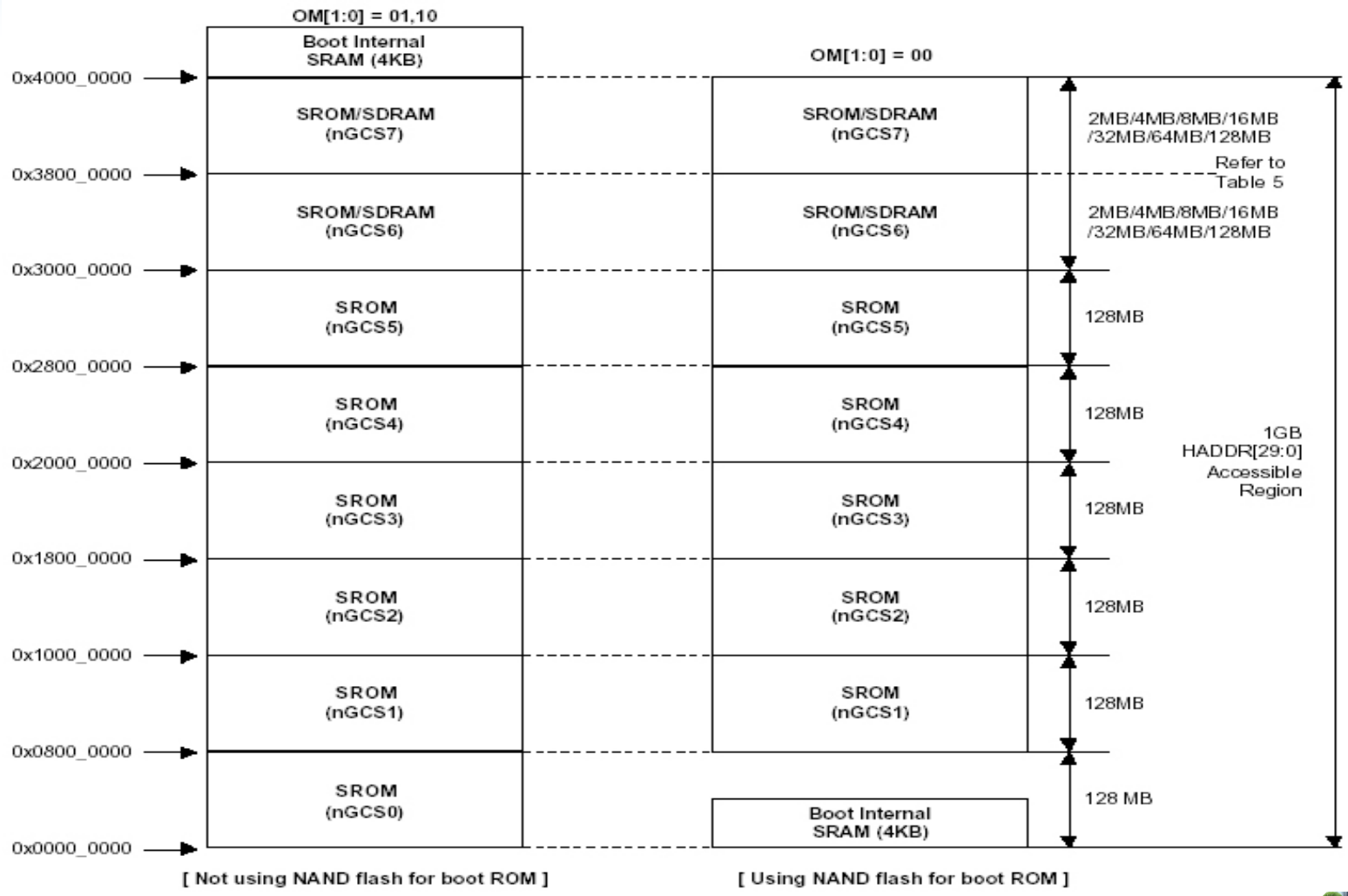


启动与片选

S3C2440 支持两种启动模式：一种是从Nand Flash 启动；一种是从Nor Flash 启动。在此两种启动模式下，各个片选的存储空间分配是不同的：

- 在 NAND Flash 启动模式下，内部的4K Bytes BootSram 被映射到nGCS0 片选的空间
- 在 Nor Flash 启动模式下，与nGCS0 相连的Nor Flash被映射到nGCS0 片选的空间

启动与片选



启动与片选

mini2440的启动模式选择，是通过拨动开关 S2 来决定的，根据提示:

- S2 接到Nor Flash 标识一侧时， 系统将从 Nor Flash 启动;
- S2 接到Nand Flash 标识一侧时， 系统将从Nand Flash 启动。

启动与片选

mini2440 使用了两片外接的32M bytes 总共64M bytes 的SDRAM 芯片(型号为:HY57V561620FTP)，一般称之为内存，它们并接在一起形成32-bit 的总线数据宽度，这样可以增加访问的速度；因为是并接，故它们都使用了nGCS6 作为片选，根据CPU 手册5-2 中的介绍可知，这就决定了它们的物理起始地址为0x30000000。

LED

LED 是开发中最常用的状态指示设备，mini2440 板具有4 个用户可编程LED，它们直接与CPU 的GPIO 相连接，低电平有效(点亮)。

资源对应表:

LED1	LED2	LED3	LED4
GPB5	GPB6	GPB7	GPB8

按键

mini2440共有6 个用户按键，它们均从CPU中断引脚直接引出，属于低电平触发。

K1	K2	K3	K4	K5	K6
EINT8	EINT11	EINT13	EINT14	EINT15	EINT19
GPG0	GPG3	GPG5	GPG6	GPG7	GPG11

JTAG

当开发板从贴片厂下线时，里面没有任何程序，这时一般通过JTAG接口烧写第一个程序，就是bootloader，借助bootloader可以使用USB口下载更加复杂的系统程序等。除此之外，JTAG 接口在开发中最常见的用途是单步调试，不管是市面上常见的JLINK还是ULINK，以及其他的仿真调试器，最终都是通过JTAG 接口连接的。

Contents

工作状态

从编程的角度看，ARM微处理器的工作状态一般有两种，并可在两种状态之间切换:

- 第一种为ARM状态，此时处理器执行32位的字对齐的ARM指令。
- 第二种为Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

工作状态

当ARM微处理器执行32位的ARM指令集时，工作在ARM状态；当ARM微处理器执行16位的Thumb指令集时，工作在Thumb状态。在程序的执行过程中，微处理器可以随时在两种工作状态之间切换，并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。

存储器格式

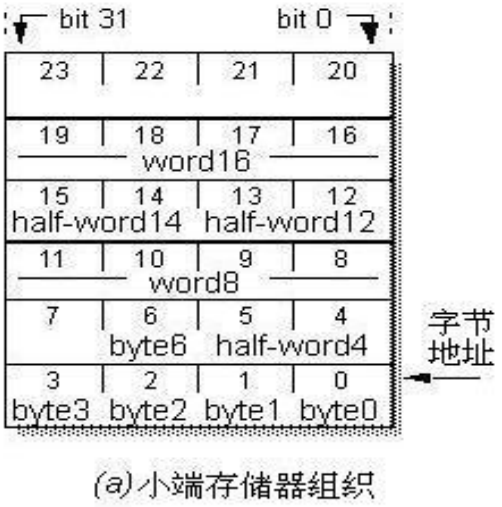
ARM体系结构将存储器看作是从零地址开始的字节的线性组合。从零字节到三字节放置第一个存储的字（32位）数据，从第四个字节到第七个字节放置第二个存储的字数据，依次排列。作为32位的微处理器，ARM体系结构所支持的最大寻址空间为4GB。

存储器格式

ARM体系结构可以用两种方法存储字数据，称之为大端格式和小端格式。

- 大端格式：字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中。
- 小端格式：与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。

存储器格式



指令长度

ARM微处理器的指令长度可以是32位（在ARM状态下），也可以为16位（在Thumb状态下）。ARM微处理器中支持字节（8位）、半字（16位）、字（32位）三种数据类型，其中，字需要4字节对齐（地址的低两位为0）、半字需要2字节对齐（地址的最低位为0）。

工作模式

ARM微处理器支持7种工作模式，分别为：

1、用户模式（Usr）

用于正常执行程序

2、快速中断模式(FIQ)

用于高速数据传输

3、外部中断模式(IRQ)

用于通常的中断处理

工作模式

4. 管理模式（svc）

操作系统使用的保护模式

5. 数据访问终止模式(abt)

当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。

6. 系统模式（sys）

运行具有特权的操作系统任务。

7. 未定义指令中止模式（und）

当未定义的指令执行时进入该模式，可用于支持硬件

工作模式

ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

工作模式

除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

Contents

寄存器

ARM微处理器共有37个32位寄存器，其中31个为通用寄存器，6个为状态寄存器。但是这些寄存器不能被同时访问，具体哪些寄存器是可以访问的，取决ARM处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器R14～R0、程序计数器PC、一个状态寄存器都是可访问的。

寄存器(ARM状态)

在ARM工作状态下，任一时刻可以访问16个通用寄存器和一到两个状态寄存器。在非用户模式（特权模式）下，则可访问到特定模式分组寄存器， 具体见下页图:

寄存器(ARM状态)

www.enjoylinux.cn

ARM状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM状态下的程序状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= 分组寄存器

寄存器(Thumb状态)

Thumb状态下的寄存器集是ARM状态下寄存器集的一个子集，程序可以直接访问8个通用寄存器（R7～R0）、程序计数器（PC）、堆栈指针（SP）、连接寄存器（LR）和CPSR。 具体见下页图:

寄存器(Thumb状态)

www.enjoylinux.cn

Thumb状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	SP_fiq	SP_svg	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
PC	PC	PC	PC	PC	PC

Thumb状态下的程序状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

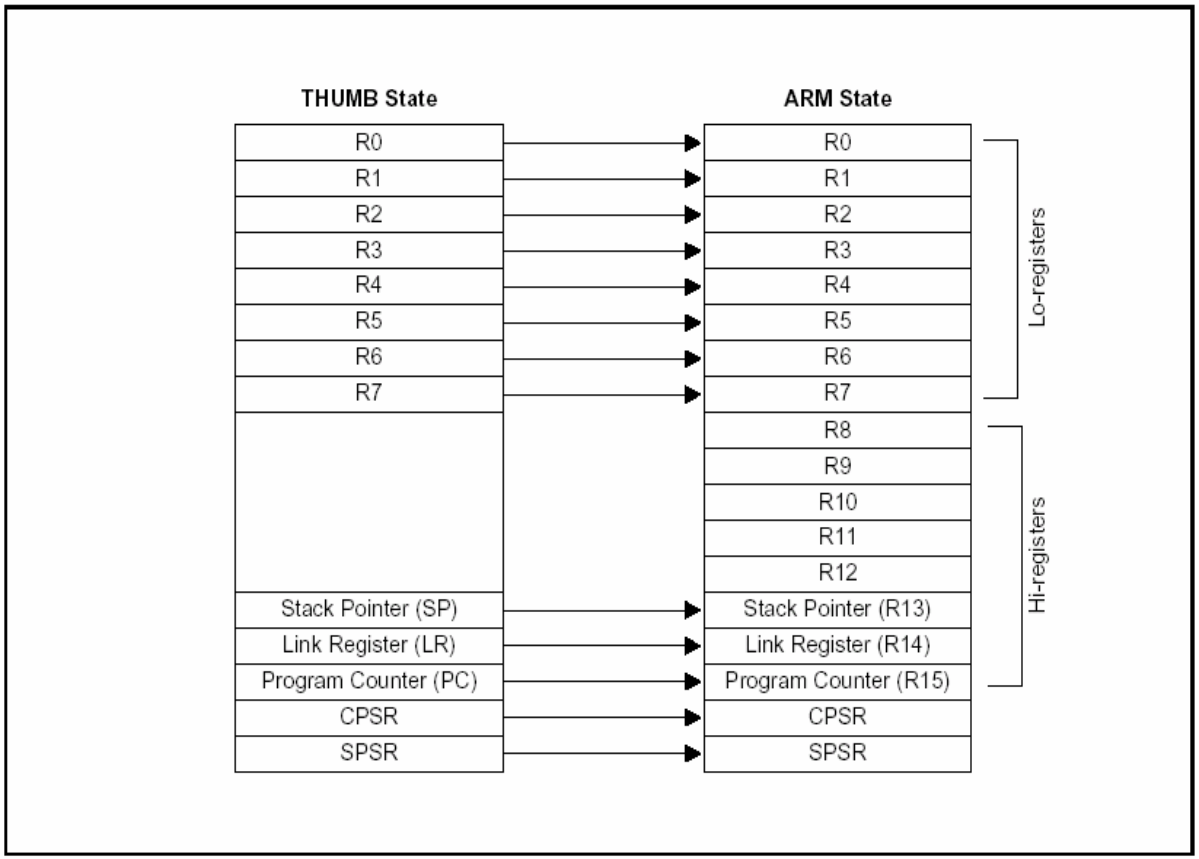
= 分組寄存器

寄存器对应

Thumb状态下的寄存器组织与ARM状态下的寄存器组织的关系：

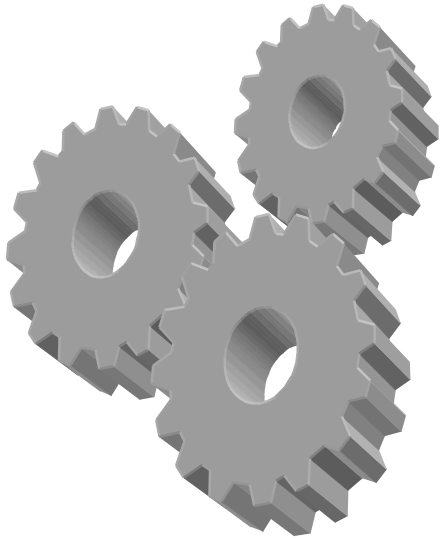
- Thumb状态下和ARM状态下的R0～R7是相同的。
- Thumb状态下和ARM状态下的CPSR和SPSR是相同的。
- Thumb状态下的SP对应于ARM状态下的R13。
- Thumb状态下的LR对应于ARM状态下的R14。
- Thumb状态下的程序计数器PC对应于ARM状态下R15。

寄存器对应



通用寄存器

- R0 ~ R15
- R13_svc、R14_svc
- R13_abt、R14_abt
- R13_und、R14_und
- R13_irq、R14_irq
- R8_fiq ~ R14_fiq



通用寄存器

- 不分组寄存器（The unbanked registers）

  R0-R7

- 分组寄存器（The banked registers）

  R8-R14

- 程序计数器：R15（PC）

不分组通用寄存器

R0-R7是不分组寄存器。这意味着在所有处理器模式下，访问的都是同一个物理寄存器。未分组寄存器没有被系统用于特别的用途，任何可采用通用寄存器的应用场合都可以使用未分组寄存器。

分组通用寄存器

- 分组寄存器R8-R12
  1. FIQ模式分组寄存器R8-R12
  2. FIQ以外的分组寄存器R8-R12
- 分组寄存器R13、R14
  1. 寄存器R13通常用做堆栈指针SP
  2. 寄存器R14用作子程序链接寄存器（Link Register－LR），也称为LR，指向函数的返回地址

程序计数器

寄存器R15被用作程序计数器，也称为PC 。其值等于当前正在执行的指令的地址+8(因为在取址和执行之间多了一个译码的阶段)。

状态寄存器

- CPSR
- SPSR_svc
- SPSR_abt
- SPSR_und
- SPSR_irq
- SPSR_fiq

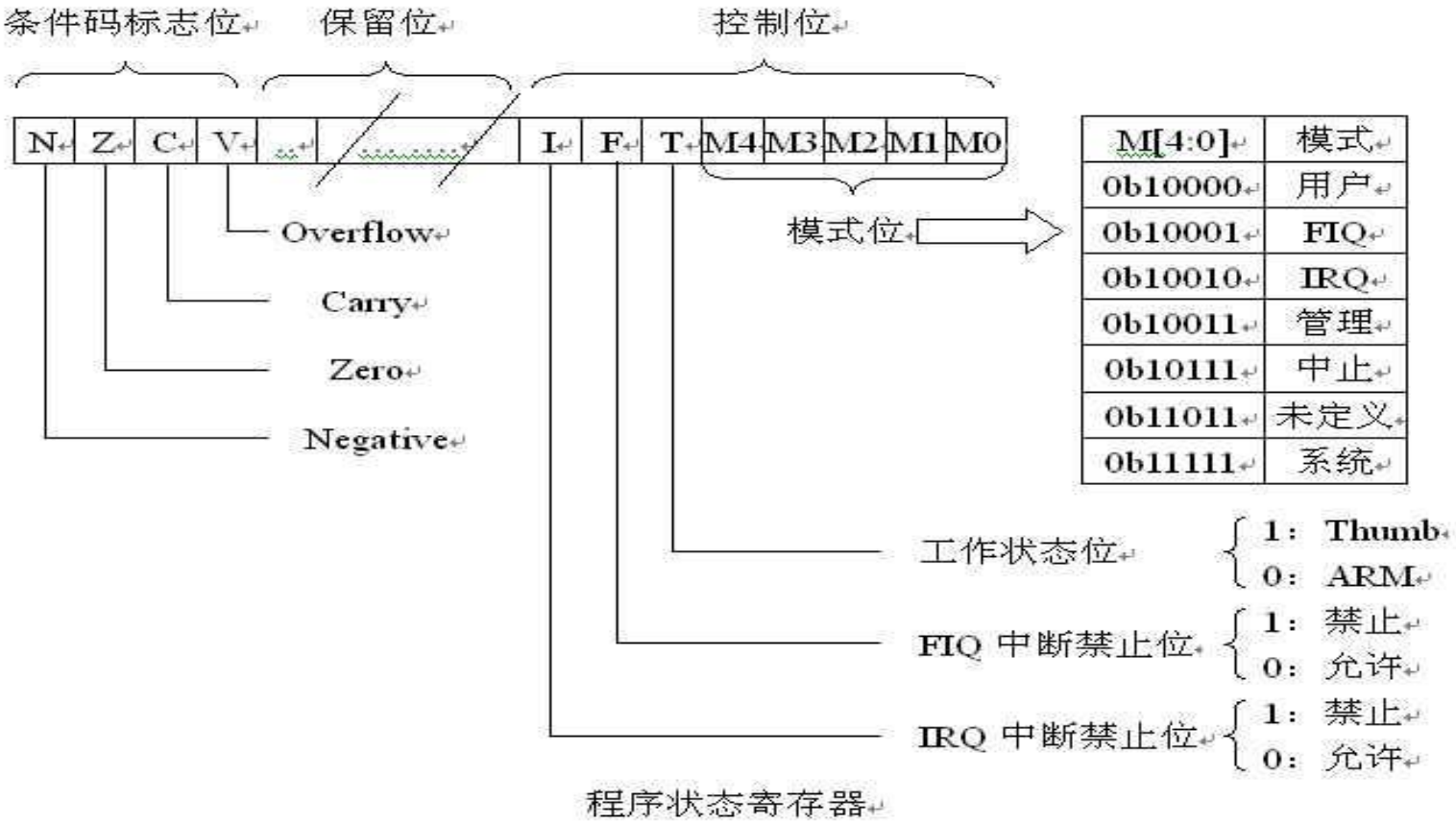
状态寄存器

ARM所有工作模式下都可以访问程序状态寄存器CPSR。CPSR包含条件码标志、中断禁止位、当前处理器模式以及其它状态和控制信息。

状态寄存器

CPSR在每种异常模式下都有一个对应的物理寄存器——程序状态保存寄存器SPSR。当异常出现时，SPSR用于保存CPSR的值，以便异常返回后恢复异常发生时的工作状态。

CPSR/SPSR



M[4:0]	模式
0b10000	用户
0b10001	FIQ
0b10010	IRQ
0b10011	管理
0b10111	中止
0b11011	未定义
0b11111	系统

CPSR/SPSR

www.enjoylinux.cn

CPSR[4:0]	模式	用途	可访问的寄存器
10000	用户	正常用户模式，程序正常执行模式	PC，R14~R0，CPSR
10001	FIQ	处理快速中断，支持高速数据传送或通道处理	PC，R14_fiq~R8_fiq，R7~R0，CPSR，SPSR_fiq
10010	IRQ	处理普通中断	PC，R14_irq~R13_fiq，R12~R0，CPSR，SPSR_irq
10011	SVC	操作系统保护模式，处理软件中断（SWI）	PC，R14_svc~R13_svc，R12~R0，CPSR，SPSR_svc
10111	中止	处理存储器故障、实现虚拟存储器和存储器保护	PC，R14_abt~R13_abt，R12~R0，CPSR，SPSR_abt
11011	未定义	处理未定义的指令陷阱，支持硬件协处理器的软件仿真	PC，R14_und~R13_und，R12~R0，CPSR，SPSR_und
11111	系统	运行特权操作系统任务	PC，R14~R0，CPSR

Contents

寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。

立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。例如以下指令:

- ADD R0，R0，＃1 ；R0←R0＋1
- ADD R0，R0，＃0x3f ；R0←R0＋0x3f

在以上两条指令中，第二个源操作数即为立即数，要求以“＃”为前缀，对于以十六进制表示的立即数，还要求在“＃”后加上“0x”或“&”。

寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。

ADD　　R0，R1，R2　　；R0←R1＋R2

该指令的执行效果是将寄存器R1和R2的内容相加，其结果存放在寄存器R0中。

寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如以下指令：

- ADD R0，R1，[R2]　　；R0←R1＋[R2]
- LDR　　R0，[R1]　　；R0←[R1]

在第一条指令中，以寄存器R2的值作为操作数的地址，在存储器中取得一个操作数后与R1相加，结果存入寄存器R0中。第二条指令将以R1的值为地址的存储器中的数据传送到R0中。

基址变址寻址

基址变址寻址就是将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址：

- LDR R0，[R1，＃4]　　；R0←[R1＋4]
- LDR R0，[R1，＃4]!　　；R0←[R1＋4]、R1←R1＋4
- LDR R0，[R1]，＃4　　；R0←[R1]、R1←R1＋4
- LDR R0，[R1，R2]　　；R0←[R1＋R2]

多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这寻址方式可以用一条指令完成传送最多16个通用寄存器的值。以下指令:

- LDMIA R0，{R1，R2，R3，R4} ； R1←[R0]
- ； R2←[R0 + 4]
- ； R3←[R0 + 8]
- ； R4←[R0 + 12]

该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1～R4。

相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器PC的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令BL采用了相对寻址方式：

```
BL      NEXT            ；跳转到子程序NEXT处执行
……
NEXT
……
MOV   PC，LR        ；从子程序返回
```


堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

递增堆栈：向高地址方向生长

递减堆栈：向低地址方向生长

满堆栈：堆栈指针指向最后压入堆栈的有效数据项

空堆栈：堆栈指针指向下一个要放入数据的空位置

Contents

- ARM指令集
- ARM伪指令
- C/C++混合编程
- ADS集成开发环境

跳转指令

跳转指令用于实现程序流程的跳转，在ARM程序中有两种方法可以实现程序流程的跳转：

- 使用专门的跳转指令。
- 直接向程序计数器PC写入跳转地址值，通过向程序计数器PC写入跳转地址值，可以实现在4GB的地址空间中的任意跳转，在跳转之前结合使用

```
MOV        LR，PC
```

等类似指令，可以保存将来的返回地址值，从而实现在4GB连续的线性地址空间的子程序调用。

跳转指令

ARM指令集中的跳转指令可以完成从当前指令向前或向后的32MB的地址空间的跳转，包括以下4条指令:

- B 跳转指令
- BL 带返回的跳转指令
- BLX 带返回和状态切换的跳转指令
- BX 带状态切换的跳转指令

跳转指令

B 指令的格式为：

B{条件}　　目标地址

B指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。

- B Label

  程序无条件跳转到标号Label处执行

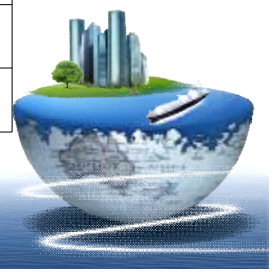
```
CMP  R1，  # 0
BEQ  Label
```

当CPSR寄存器中的Z条件码置位时，程序跳转到标号Label处执行。

指令条件

www.enjoylinux.cn

指令条件	标 志 位	含 义
EQ	Z置位	相等
NE	Z清零	不相等
CS	C置位	无符号数大于或等于
CC	C清零	无符号数小于
MI	N置位	负数
PL	N清零	正数或零
VS	V置位	溢出
VC	V清零	未溢出
HI	C置位Z清零	无符号数大于
LS	C清零Z置位	无符号数小于或等于
GE	N等于V	带符号数大于或等于
LT	N不等于V	带符号数小于
GT	Z清零且（N等于V）	带符号数大于
LE	Z置位或（N不等于V）	带符号数小于或等于
AL	忽略	无条件执行



跳转指令

BL指令的格式为：

BL{条件}　　目标地址

BL 是另一个跳转指令，但跳转之前，会在寄存器R14中保存PC当前值，因此，可以通过将R14 的内容重新加载到PC中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。

- BL　Label　　；当程序无条件跳转到标号Label处执行时，同时将当前的PC值保存到R14中

跳转指令

BLX指令的格式为：

BLX 目标地址

BLX指令从ARM指令集跳转到指令中所指定的目标地址，并将处理器的工作状态有ARM状态切换到Thumb状态，该指令同时将PC的当前内容保存到寄存器R14中。因此，当子程序使用Thumb指令集，而调用者使用ARM指令集时，可以通过BLX指令实现子程序的调用和处理器工作状态的切换。同时，子程序的返回可以通过将寄存器R14值复制到PC中来完成。

跳转指令

BX指令的格式为:

BX{条件} 目标地址

BX指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是ARM指令，也可以是Thumb指令。

数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。数据传送指令用于在寄存器和存储器之间进行数据的双向传输。

算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新CPSR中的相应条件标志位。

MOV指令

MOV指令的格式为:

MOV{条件}{S}　目的寄存器，源操作数

MOV指令完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中S选项决定指令的操作是否影响CPSR中条件标志位的值，当没有S时指令不更新CPSR中条件标志位的值。

指令示例:

- MOV　R1，R0　；将寄存器R0的值传送到寄存器R1
- MOV　PC，R14　；将寄存器R14的值传送到PC，常用于子程序返回
- MOV　R1，R0，LSL＃3　；将寄存器R0的值左移3位后传送到R1

MVN指令

MVN指令的格式为:

MVN{条件}{S}　目的寄存器，源操作数

MVN指令可完成从另一个寄存器、被移位的寄存器、或将一个立即数加载到目的寄存器。与MOV指令不同之处是在传送之前按位被取反了，即把一个被取反的值传送到目的寄存器中。其中S决定指令的操作是否影响CPSR中条件标志位的值，当没有S时指令不更新CPSR中条件标志位的值。

- MVN　R0，#0xff　；R0<=0xffffff00

CMP指令

CMP{条件} 操作数1，操作数2

CMP指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行比较，同时更新CPSR中条件标志位的值。该指令进行一次减法运算，但不存储结果，只更改条件标志位。标志位表示的是操作数1与操作数2的关系(大、小、相等)，例如，当操作数1大于操作操作数2，则此后的有GT 后缀的指令将可以执行。

指令示例:

- CMP R1，R0 ；将寄存器R1的值与寄存器R0的值相减，并根据结果设置CPSR的标志位
- CMP R1，＃100 ；将寄存器R1的值与立即数100相减，并根据结果设置CPSR的标志位

TST指令

TST{条件} 操作数1，操作数2

TST指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的与运算，并根据运算结果更新CPSR中条件标志位的值。操作数1是要测试的数据，而操作数2是一个位掩码，根据测试结果设置相应标志位。

当位与结果为0时，EQ位被设置。

指令示例:

- TST R1，#%1 ；用于测试在寄存器R1中是否设置了最低位（%表示二进制数）。

ADD 指令

ADD{条件}{S} 目的寄存器，操作数1，操作数2

ADD指令用于把两个操作数相加，并将结果存放到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。

指令示例:

- ADD R0，R1，R2 ； R0 = R1 + R2
- ADD R0，R1，#256 ； R0 = R1 + 256
- ADD R0，R2，R3，LSL#1 ； R0 = R2 + (R3 << 1)

SUB指令

SUB{条件}{S} 目的寄存器，操作数1，操作数2

SUB指令用于把操作数1减去操作数2，并将结果存放到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。

- SUB　　R0，R1，R2　　；　R0 = R1 - R2
- SUB　　R0，R1，#256　　；　R0 = R1 - 256
- SUB　　R0，R2，R3，LSL#1　；　R0 = R2 - (R3 << 1)

AND指令

AND{条件}{S} 目的寄存器，操作数1，操作数2

AND指令用于在两个操作数上进行逻辑与运算，并把结果放置到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于屏蔽操作数1的某些位。

- AND　R0，R0，＃3　　；该指令保持R0的0、1位，其余位清零。

ORR 指令

ORR{条件}{S} 目的寄存器，操作数1，操作数2

ORR指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操作数1的某些位。

- ORR　　R0，R0，＃3　　；该指令设置R0的0、1位，其余位保持不变。

BIC指令

BIC{cond}{S} Rd,Rn,operand2

BIC指令用于清除Rn 中的某些位，并把结果存放在Rd中，操作数operand2 为32位的掩码，如果掩码中设置了某一位为1，则清除这一位。

例：BIC R0,R0,#%1011；将R0的0，1，3位清零，其余位不变。

MUL指令

MUL{条件}{S}　　目的寄存器，操作数1，操作数2

MUL指令完成将操作数1与操作数2的乘法运算，并把结果放置到目的寄存器中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中，操作数1和操作数2均为32位的有符号数或无符号数。

- MUL　　R0，R1，R2　　；R0 = R1 × R2
- MULS　　R0，R1，R2　　；R0 = R1 × R2，同时设置CPSR中的相关条件标志位

程序状态寄存器访问指令

ARM微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据。

MRS指令

MRS{条件} 通用寄存器，程序状态寄存器（CPSR或SPSR）

MRS指令用于将程序状态寄存器的内容传送到通用寄存器中。

该指令一般用在以下几种情况:

- 当需要改变程序状态寄存器的内容时，可用MRS将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。
- 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。

```
MRS   R0，CPSR    ；传送CPSR的内容到R0
MRS   R0，SPSR    ；传送SPSR的内容到R0
```


MSR指令

MSR{条件} 程序状态寄存器（CPSR或SPSR）_<域>，操作数

MSR指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，32位的程序状态寄存器可分为4个域:

位[31：24]为条件标志位域，用f表示；位[23：16]为状态位域，用s表示；

位[15：8]为扩展位域，用x表示；　　位[7：0]为控制位域，用c表示；

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在MSR指令中指明将要操作的域。

指令示例:

- MSR CPSR，R0　　；传送R0的内容到CPSR
- MSR SPSR，R0　　；传送R0的内容到SPSR
- MSR CPSR_c，R0 ；传送R0的内容到SPSR，但仅仅修改CPSR中的控制位域

加载/存储指令

ARM微处理器支持加载/存储指令用于在寄存器和存储器之间传送数据，加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。

LDR指令

LDR指令的格式为：

LDR{条件} 目的寄存器，<存储器地址>

LDR指令用于从存储器中将一个32位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取32位的字数据到通用寄存器，然后对数据进行处理。

LDR指令

- LDR R0，[R1]

  将存储器地址为R1的字数据读入寄存器R0。

- LDR R0，[R1，R2]

  将存储器地址为R1+R2的字数据读入寄存器R0。

- LDR R0，[R1，＃8]

  将存储器地址为R1+8的字数据读入寄存器R0。

- LDR R0，[R1，R2]！

  将存储器地址为R1+R2的字数据读入寄存器R0，并将新地址R1 + R2写入R1。

LDR指令

- LDR R0，[R1，＃8]！

  将存储器地址为R1+8的字数据读入寄存器R0，并将新地址R1＋8写入R1。

- LDR R0，[R1]，R2

  将存储器地址为R1的字数据读入寄存器R0，并将新地址R1＋R2写入R1。

- LDR R0，[R1，R2，LSL＃2]!

  将存储器地址为R1＋R2×4的字数据读入寄存器R0，并将新地址R1＋R2×4写入R1。

- LDR R0，[R1]，R2，LSL＃2

  将存储器地址为R1的字数据读入寄存器R0，并将新地址R1＋R2×4写入R1。

LDRB指令

LDRB指令的格式为：

LDR{条件}B 目的寄存器，<存储器地址>

LDRB指令用于从存储器中将一个8位的字节数据传送到目的寄存器中，同时将寄存器的高24位清零。该指令通常用于从存储器中读取8位的字节数据到通用寄存器，然后对数据进行处理。

指令示例：

- LDRB　　R0，[R1]　　；将存储器地址为R1的字节数据读入寄存器R0，并将R0的高24位清零。
- LDRB　　R0，[R1，＃8]　；将存储器地址为R1＋8的字节数据读入寄存器R0，并将R0的高24位清零。

LDRH指令

LDRH指令的格式为:

LDR{条件}H 目的寄存器，<存储器地址>

LDRH指令用于从存储器中将一个16位的半字数据传送到目的寄存器中，同时将寄存器的高16位清零。该指令通常用于从存储器中读取16位的半字数据到通用寄存器，然后对数据进行处理。

指令示例:

- LDRH R0，[R1] ；将存储器地址为R1的半字数据读入寄存器R0，并将R0的高16位清零。
- LDRH R0，[R1，R2] ；将存储器地址为R1＋R2的半字数据读入寄存器R0，并将R0的高16位清零。

STR指令

STR指令的格式为:

STR{条件} 源寄存器，<存储器地址>

STR指令用于从源寄存器中将一个32位的字数据传送到存储器中。

指令示例:

- STRR0，[R1]，＃8　　；将R0中的字数据写入以R1为地址的存储器中，并将新地址R1＋8写入R1。
- STRR0，[R1，＃8]　　；将R0中的字数据写入以R1＋8为地址的存储器中。

批量加载/存储指令

ARM微处理器所支持的批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下：

- LDM　　批量数据加载指令
- STM　　批量数据存储指令

LDM指令

LDM指令的格式为：

LDM {条件}{类型} 基址寄存器{！ }，寄存器列表{∧}

LDM（或STM）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。其中，{类型}为以下几种情况：

LDM指令

其中，{类型}为以下几种情况：

- IA　每次传送后地址加1;
- IB　每次传送前地址加1;
- DA　每次传送后地址减1;
- DB　每次传送前地址减1;
- FD　满递减堆栈;
- ED　空递减堆栈;
- FA　满递增堆栈;
- EA　空递增堆栈;

LDM指令

{!}为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变。基址寄存器不允许为R15，寄存器列表可以为R0～R15的任意组合。

{∧}为可选后缀，当指令为LDM且寄存器列表中包含R15，选用该后缀时表示：除了正常的数据传送之外，还将SPSR复制到CPSR。

LDM指令

指令示例:

- **STMFD R13!，{R0，R4-R12，LR}**

  将寄存器列表中的寄存器（R0，R4到R12，LR）存入堆栈。

- **LDMFD R13!，{R0，R4-R12，PC}**

  将堆栈内容恢复到寄存器（R0，R4到R12，LR）。

数据交换指令

ARM微处理器所支持数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条:

- SWP　字数据交换指令
- SWPB　字节数据交换指令

SWP指令

SWP指令的格式为：

SWP{条件} 目的寄存器，源寄存器1，[源寄存器2]

SWP指令用于将源寄存器2所指向的存储器中的字数据传送到目的寄存器中，同时将源寄存器1中的字数据传送到源寄存器2所指向的存储器中。显然，当源寄存器1和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

- SWP R0，R1，[R2] ；将R2所指向的存储器中的字数据传送到R0，同时将R1中的字数据传送到R2所指向的存储单元。
- SWP R0，R0，[R1] ；该指令完成将R1所指向的存储器中的字数据与R0中的字数据交换。

移位指令

ARM微处理器支持数据的移位操作，移位操作在ARM指令集中不作为单独的指令使用，它只能作为指令格式中是一个字段，在汇编语言中表示为指令中的选项。移位操作包括如下6种类型，ASL和LSL是等价的，可以自由互换:

- LSL 逻辑左移
- ASL 算术左移
- LSR 逻辑右移
- ASR 算术右移
- ROR 循环右移

LSL操作

LSL操作的格式为:

通用寄存器，LSL操作数

LSL可完成对通用寄存器中的内容进行逻辑左移操作，按操作数所指定的数量向左移位，低位用零来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例:

```
MOV    R0, R1, LSL#2
```

将R1中的内容左移两位后传送到R0中。

ROR操作

ROR操作的格式为:

通用寄存器，ROR 操作数

ROR可完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循环移位，左端用右端移出的位来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。显然，当进行32位的循环右移操作时，通用寄存器中的值不改变。

操作示例:

```
MOV    R0, R1, ROR#2
```

将R1中的内容循环右移两位后传送到R0中。

异常产生指令

ARM微处理器所支持的异常指令有如下两条:

- SWI　　软件中断指令
- BKPT　断点中断指令

SWI指令

SWI指令的格式为：

SWI{条件} 24位的立即数

SWI指令用于产生软件中断，以便用户程序能调用操作系统的系统API。操作系统在SWI的异常处理程序中提供相应的系统服务，指令中24位的立即数指定用户程序调用的API类型。

指令示例：

```
SWI    0x02
```

该指令调用操作系统编号位02的系统例程。

Contents

- ARM指令集
- ARM伪指令
- C/C++混合编程
- ADS集成开发环境

伪指令

在ARM汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，他们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成。

伪指令

在ARM的汇编程序中，有如下几种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

符号定义伪指令

符号定义伪指令用于定义ARM汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。

常见的符号定义伪指令有如下几种：

- 定义全局变量的GBLA、GBLL和GBLS
- 定义局部变量的LCLA、LCLL和LCLS
- 对变量赋值的SETA、SETL、SETS
- 为通用寄存器列表定义名称的RLIST

GBLA/GBLL/GBLS

语法格式:

- GBLA（GBLL或GBLS） 全局变量名
- GBLA、GBLL和GBLS伪指令用于定义一个ARM程序中的全局变量，并将其初始化。其中:
- GBLA: 定义一个全局的数字变量，并初始化为0;
- GBLL:定义一个全局的逻辑变量，并初始化为F（假）;
- GBLS:定义一个全局的字符串变量，并初始化为空;

以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

GBLA/GBLL/GBLS

使用示例:

- GBLA Test1

  定义一个全局的数字变量，变量名为Test1

- Test1 SETA 0xaa

  将该变量赋值为0xaa

- GBLL Test2

  定义一个全局的逻辑变量，变量名为Test2

- Test2 SETL {TRUE}

  将该变量赋值为真

- GBLS Test3

  定义一个全局的字符串变量，变量名为Test3

- Test3 SETS “Testing”

  将该变量赋值为“Testing”

RLIST指令

语法格式:

名称　RLIST　{寄存器列表}

RLIST伪指令用于对一个通用寄存器列表定义名称，使用该伪指令定义的列表名称可在ARM指令LDM/STM中使用。在LDM/STM指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。

使用示例:

- RegList　RLIST　{R0-R5，R8，R10}　；将寄存器列表名称定义为RegList，可在ARM指令LDM/STM中通过该名称访问寄存器列表。

数据定义伪指令

数据定义伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪指令有如下几种：

DCB　DCW　　DCD　DCFD

DCFS　DCQ　　SPACE　　MAP　　FIELD

DCB指令

语法格式:

标号　DCB　表达式

DCB伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为0～255的数字或字符串。DCB也可用“=”代替。

使用示例:

Str DCB “This is a test！”；分配一片连续的字节存储单元并初始化。

SPACE指令

语法格式:

标号　SPACE　　表达式

SPACE伪指令用于分配一片连续的存储区域并初始化为0。 其中，表达式为要分配的字节数。SPACE也可用“%”代替。

使用示例:

DataSpace　　SPACE　　100　；分配连续100字节的存储单元并初始化为0。

MAP指令

语法格式:

MAP　表达式{，基址寄存器}

MAP伪指令用于定义一个结构化的内存表的首地址。MAP也可用“^”代替。表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。MAP伪指令通常与FIELD伪指令配合使用来定义结构化的内存表。

使用示例:

MAP　0x100，R0

定义结构化内存表首地址的值为0x100 + R0。

FILED指令

语法格式:

标号　　FIELD　表达式

FIELD伪指令用于定义一个结构化内存表中的数据域。FILED也可用“#”代替。表达式的值为当前数据域在内存表中所占的字节数。FIELD伪指令常与MAP伪指令配合使用来定义结构化的内存表。MAP伪指令定义内存表的首地址，FIELD伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。

使用示例:

```
MAP     0x100   ；定义结构化内存表首地址的值为0x100。
A       FIELD   16      ；定义A的长度为16字节，位置为0x100
B       FIELD   32      ；定义B的长度为32字节，位置为0x110
S       FIELD   256     ；定义S的长度为256字节，位置为0x130
```


汇编控制伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条:

- IF、ELSE、ENDIF
- WHILE、WEND
- MACRO、MEND
- MEXIT

IF指令

语法格式:

IF逻辑表达式

指令序列1

ELSE

指令序列2

ENDIF

IF、ELSE、ENDIF伪指令能根据条件的成立与否决定是否执行某个指令序列。当IF后面的逻辑表达式为真，则执行指令序列1，否则执行指令序列2。其中，ELSE及指令序列2可以没有。IF、ELSE、ENDIF伪指令可以嵌套使用。

IF指令

使用示例:

```
GBLL      Test   ；声明一个全局的逻辑变量，变量名为Test
……
IF        Test = TRUE
  指令序列1
ELSE
  指令序列2
ENDIF
```

WHILE指令

语法格式:

```
WHILE  逻辑表达式
  指令序列
WEND
```

WHILE、WEND伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当WHILE后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

WHILE、WEND伪指令可以嵌套使用。

WHILE指令

使用示例:

```
GBLA Counter    ; 声明一个全局的数学变量，变量名为Counter
Counter  SETA    3          ; 由变量Counter控制循环次数
……
WHILE   Counter < 10
        指令序列
WEND
```


其他伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下几条:

- AREA
- ALIGN
- CODE16、CODE32
- ENTRY
- END
- EQU
- EXPORT（或GLOBAL）
- IMPORT
- EXTERN
- GET（或INCLUDE）

AREA指令

语法格式：

AREA 段名 属性1，属性2，……

AREA伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括起来，如|1_test|。属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。常用的属性如下：

- CODE属性：用于定义代码段，默认为READONLY。
- DATA属性：用于定义数据段，默认为READWRITE。
- READONLY属性：指定本段为只读，代码段默认为READONLY。
- READWRITE属性：指定本段为可读可写，数据段的默认属性为READWRITE。

AREA指令

- ALIGN属性,使用方式为

  ALIGN 表达式

  在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的。

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

使用示例:

```
AREA    Init，CODE，READONLY
```

该伪指令定义了一个代码段，段名为Init，属性为只读

ALIGN指令

语法格式：

ALIGN {表达式{，偏移量}}

ALIGN伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式|。其中，表达式的值用于指定对齐方式，可能的取值为2的幂，如1、2、4、8、16等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2的表达式次幂＋偏移量。

使用示例：

AREA　Init，CODE，READONLY，ALIEN＝3

指定后面的指令为8字节对齐。

CODE16指令

语法格式:

CODE16（或CODE32）

CODE16伪指令通知编译器，其后的指令序列为16位的Thumb指令。CODE32伪指令通知编译器，其后的指令序列为32位的ARM指令。因此，在使用ARM指令和Thumb指令混合编程的代码里，可用这两条伪指令进行切换，但注意他们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

CODE16指令

使用示例:

```
AREA  Init，CODE，READONLY
……
CODE32 ；通知编译器其后的指令为32位的ARM指令
LDR    R0，= NEXT + 1        ；将跳转地址放入寄存器R0
BX     R0    ；程序跳转到新的位置执行，并将处理器切换到Thumb工作状态
……
CODE16 ；通知编译器其后的指令为16位的Thumb指令
NEXT   LDR          R3，= 0x3FF
……
END       ；程序结束
```

ENTRY指令

语法格式:

ENTRY

ENTRY伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个**ENTRY**（也可以有多个，当有多个**ENTRY**时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个**ENTRY**（可以没有）。

使用示例:

```
AREA    Init，CODE，READONLY
ENTRY   ；指定应用程序的入口点
……
```


EQU指令

语法格式:

名称 EQU 表达式{，类型}

EQU伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于C语言中的＃define。其中EQU可用“*”代替。

使用示例:

```
Test  EQU   50        ；定义标号Test的值为50
Addr EQU   0x55，CODE32   ；定义Addr的值为0x55，且该处为32位的ARM指令。
```


EXPORT指令

语法格式:

EXPORT 标号

EXPORT伪指令用于在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。EXPORT可用GLOBAL代替。标号在程序中区分大小写。

使用示例:

```
AREA    Init，CODE，READONLY
EXPORT      Stest
```

声明一个可全局引用的标号Stest

IMPORT指令

语法格式:

IMPORT　标号

IMPORT伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用。标号在程序中区分大小写。

使用示例:

```
AREA    Init，CODE，READONLY
IMPORT Main
```

通知编译器当前文件要引用标号Main，但Main在其他源文件中定义

END指令

语法格式:

END

END伪指令用于通知编译器已经到了源程序的结尾。

使用示例:

```
AREA Init，CODE，READONLY
……
END；指定应用程序的结尾
```

Sample.S分析

实验

汇编程序分析

阅读Sample.S，写出每行代码的注释

Contents

- ARM指令集
- ARM伪指令
- C/C++混合编程
- ADS集成开发环境

混合编程

在应用系统的程序设计中，若所有的编程任务均用汇编语言来完成，其工作量是可想而知的，同时，不利于系统升级或应用软件移植，事实上，ARM体系结构支持C/C + 以及与汇编语言的混合编程，在一个完整的程序设计的中，除了初始化部分用汇编语言完成以外，其主要的编程任务一般都用C/C++ 完成。汇编语言与C/C++\的混合编程通常有以下几种方式:

- 在C/C++代码中嵌入汇编指令。
- 在汇编程序和C/C++的程序之间进行变量的互访。
- 汇编程序、C/C++程序间的相互调用。

混合编程

在实际的编程中，使用较多的方式是：程序的初始化部分用汇编语言完成，然后用C/C++完成主要的编程任务，程序在执行时首先完成初始化过程，然后跳转到C/C++程序代码中，汇编程序和C/C++程序之间一般没有参数的传递，也没有频繁的相互调用，因此，整个程序的结构显得相对简单，容易理解。

混合编程

www.enjoylinux.cn

```
IMPORT Main        ;通知编译器该标号为一个外部标号
AREA    Init,CODE,READONLY        ；定义一个代码段
ENTRY                ；定义程序的入口点
LDR        R0,=0x3FF0000            ；初始化系统配置寄存器
LDR        R1,=0xE7FFFF80
STR        R1,[R0]
LDR        SP,=0x3FE1000            ；初始化用户堆栈
BL         Main     ；跳转到Main（）函数处的C/C++代码执行
END                  ；标识汇编程序的结束
```


混合编程


```
void Main(void)
{
        int i;
        *((volatile unsigned long *) 0x3ff5000) = 0x0000000f;
        while(1)
        {
           *((volatile unsigned long *) 0x3ff5008) = 0x00000001;
           for(i=0; i<0x7fFFF; i++);
           *((volatile unsigned long *) 0x3ff5008) = 0x00000002;
           for(i=0; i<0x7FFFF; i++);
        }
}
```


Contents

- ARM指令集
- ARM伪指令
- C/C++混合编程
- ADS集成开发环境

ADS

www.enjoylinux.cn

参考

友善之臂《mini2440用户手册》

实验一

www.enjoylinux.cn

编写汇编程序

1. 使用MOV指令访问ARM通用寄存器

2.使用 LDR/STR指令完成存储器的访问

实验二

www.enjoylinux.cn

使用ADD、SUB、AND、ORR、CMP、TST等

指令完成数据加减运算及逻辑运算

实验三

www.enjoylinux.cn

编写汇编程序，分别实现:

- 立即数寻址
- 寄存器寻址
- 寄存器间接寻址
- 基址变址寻址
- 多寄存器寻址

实验四

1. 使用ARM汇编指令实现if条件执行
2. 使用ARM汇编指令实现for循环结构
3. 使用ARM汇编指令实现while循环结构
4. 使用ARM汇编指令实现do…while循环结构
5. 使用ARM汇编指令实现switch开关结构

实验五

使用MRS/MSR指令切换工作模式，并初始化各种模式下堆栈指针；观察ARM处理器在各种模式下寄存器的区别。

实验六

编写一个汇编程序文件Startup.S和一个C程序文件Test.c。汇编程序的功能是初始化堆栈指针和初始化C程序的运行环境，然后调跳转到C程序运行，这就是一个简单的启动程序。C程序使用加法运算来计算1+2+3+...+(N-1)+N的值(N>0)。