**Linux操作系统把所有设备都当做文件，都是以操作文件的方式来操作设备的**

**位宽：**当一个外设芯片连接主芯片的时候，它的地址线需要进行错位

比如说8位的芯片就可以把地址0连接到芯片上的地址0

16位的外设芯片就需要主芯片的地址一与该芯片的地址0相连一次向下错位

32位的外设就需要主芯片的地址2开始相连，地址0和1可以忽略

以16位的芯片为例，如果主芯片需要读取数据的时候，这个芯片是以2个字节来传送的，传送到主芯片中会有控制器来过滤数据，所以当我们需要访问最地址1和地址2的时候发送的地址发送1和2都是一样的也就是说地址0的0或者1发过去是没有意义的，所以我们就可以把地址0可以不接

32位也同样如此

32位的处理器不等于有32跟地址线例如三星2440是32位处理器但是它只有27条地址线如果需要发出32位地址线就需要2440里面的控制器来决定，我们可以看2440的数据手册，它一最多可以发出128M的数据，也就是说27条地址线，剩下的可以是片选之类的东西这个就是需要自己来决定了

多少位的外设意味着有多少条数据线，但是并不代表会有多少条地址线，地址线是不固定的

# 容易记错记混的地方

Char型占8个位，但是它的最高位是表示的是正负

100MHz是10纳秒

1MHz = 1000kHz = 1000000Hz

1秒 = 10e-9纳秒

整型 int 4字节 32位

长整型 long 4字节 32位

字符型 char 1字节 8位

单精度 float 4字节 32位

双精度 double 8字节 64位

长双精度 long double 8字节 64位

unsigned long long: 8 64

unsigned long: 4 16

unsigned int: 4 16

unsigned short: 2 8

unsigned char: 1 8

换算关系:

1GB=1024MB

1MB=1024KB

1KB=1024Byte

1Byte=8Bit

一个字节等于八个位

$@--目标文件，$^--所有的依赖文件，$<--第一个依赖文件。

Linux命令：grep，搜索 grep 要搜索的内容

# 平时学习嵌入式的积累

Makefile中的obj-y += xxx.o和obj-m += xxx.o

内核中obj-y表示后面跟着的文件将会被编入内核中，obj-m表示后面的这个文件将会被编译成可加载模块 .ko文件

***APCS***，ARM 过程调用标准(ARM Procedure Call Standard)，提供了紧凑的编写例程的一种机制，定义的例程可以与其他例程交织在一起。最显著的一点是对这些例程来自哪里没有明确的限制。它们可以编译自 C、 Pascal、也可以是用汇编语言写成的。

APCS 定义了:

* 对寄存器使用的限制。
* 使用栈的惯例。
* 在函数调用之间传递/返回参数。
* 可以被"回溯"的基于栈的结构的格式，用来提供从失败点到程序入口的函数(和给予的参数)的列表。

**表 3.2. ARM寄存器的别名**

| **寄存器名字** | | |
| --- | --- | --- |
| **Reg#** | **APCS** | **意义** |
| R0 | a1 | 工作寄存器 |
| R1 | a2 | " |
| R2 | a3 | " |
| R3 | a4 | " |
| R4 | v1 | 必须保护 |
| R5 | v2 | " |
| R6 | v3 | " |
| R7 | v4 | " |
| R8 | v5 | " |
| R9 | v6 | " |
| R10 | sl | 栈限制 |
| R11 | fp | 桢指针 |
| R12 | ip | 内部过程调用寄存器 |
| R13 | sp | 栈指针 |
| R14 | lr | 连接寄存器 |
| R15 | pc | 程序计数器 |

**为什么PC = PC + 8：**

**ARM7和ARM9的流水线相似，ARM7是三级流水线ARM9是五级流水线**

**它们的执行阶段都是在第三阶段**

**假设，Execute（执行）阶段处于流水线中的第E阶段，每条指令是T个字节，那么**

**PC**

**= PC + N\*T**

**= PC + (E - 1) \* T**

能用一个字节标识的数据，都可以认为是合法立即数

伪指令代表在写汇编的时候用到的一些指令，当写完汇编程序时，编译完编译器自动把这个指令用别的一个或几个指令来实现这个功能，伪指令其实是没有的。

sp代表stack pointer，堆栈指针；

ip代表instruction pointer，指令指针。

伪指令ldr：在ldr后面的立即数前面加上一个“=”就表明ldr是伪指令，说白了，这个语句的作用是传送地址用的，像一般写的设置栈指针就是运用伪指令来传送地址“ldr sp， =4096”（4096是一个地址）

不是伪指令的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

mov指令后面的立即数是有限制的，这个立即数，能够**必须由一个8位的二进制数，即属于0x00-0xFF内的某个值，经过偶数次右移后得到，这样才是合法数据**，而ldr伪指令没有这个限制

cache：缓存

SoC : system on chip 芯片系统 它和CPU不同，SoC包含CPU也同时包含很多控制器(比如UART,NAND等等)，而CPU就相当于内核，是在SoC内部的

MCR 指令的格式为：

MCR{条件} 协处理器编码，协处理器操作码1，源寄存器，目的寄存器1，目的寄存器2，协处

理器操作码2。

MCR 指令用于将ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，源寄存器为ARM 处理器的寄存器，目的寄存器1 和目的寄存器2 均为协处理器的寄存器。

指令示例：

MCR P3 ， 3 ， R0 ， C4 ， C5 ， 6 ；该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中。

MRC 指令的格式为：

MRC{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理

器操作码2。

MRC 指令用于将协处理器寄存器中的数据传送到ARM 处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器为ARM 处理器的寄存器，源寄存器1 和源寄存器2 均为协处理器的寄存器。

指令示例：

**协处理器将执行的操作的操作码。对于CP15协处理器来说，<opcode\_1>永远为0b000，当<opcode\_1>不为0b000时，该指令操作结果不可预知。**

MRC P3 ， 3 ， R0 ， C4 ， C5 ， 6 ；该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器中.

**MCR指令和MRC指令只能在处理器模式为系统模式时执行，在用户模式下执行MCR指令和MRC指令将会触发未定义指令的异常中断。**

CP15中的寄存器介绍（系统控制协处理器）通过协处理器指令MCR和MRC提供具体的寄存器来配置和控制caches、MMU、保护系统、配置时钟模式（在bootloader时钟初始化用到）

**CP15的寄存器只能被MRC和MCR（Move to Coprocessor from ARM Register ）指令访问**

|  |  |  |  |
| --- | --- | --- | --- |
| Register（寄存器） | Read | Write |  |
| C0 | ID Code (1) | Unpredictable |  |
| C0 | Catch type(1) | Unpredictable |  |
| **C1** | **Control** | **Control** |  |
| **C2** | **Translation table base** | **Translation table base** |  |
| **C3** | **Domain access control** | **Domain access control** |  |
| C4 | Unpredictable | Unpredictable |  |
| C5 | Fault status(2) | Fault status (2) |  |
| C6 | Fault address | Fault address |  |
| **C7** | **Unpredictable** | **Cache operations** |  |
| **C8** | **Unpredictable** | **TLB operations** |  |
| C9 | Cache lockdown(2) | Cache lockdown (2) |  |
| C10 | TLB lock down(2) | TLB lock down(2) |  |
| C11 | Unpredictable | Unpredictable |  |
| C12 | Unpredictable | Unpredictable |  |
| C13 | Process ID | Process ID |  |
| C14 | Unpredictable | Unpredictable |  |
| C15 | Test configuration | Test configuration |  |

TLB：缓冲

**时钟频率计算公式（MHz）：**

**MPLL Control Register**  
Mpll(FCLK) = (2 \* m \* Fin) / (p \* 2S)  
m = (MDIV + 8), p = (PDIV + 2), s = SDIV  
**UPLL Control Register**  
Upll = (m \* Fin) / (p \* 2S)  
m = (MDIV + 8), p = (PDIV + 2), s = SDIV

**栈指针**

由于ARM在各种执行模式下都需要设置各自的栈指针，所以"ldr sp,=xxxx"操作较多。根据ARM的ATPCS规则，对栈的操作属于FD（满递减），即栈指针一直指向栈顶元素，是按地址减小的方向增长的，所以一般将SP设置在地址的最高处。

 ldr sp, =4\*1024，  ldr sp, =4\*1024， ldr sp, =0x34000000 ，这几个值的确定与硬件关系很大：

1.在reset中：

    ldr sp, =4\*1024：在ARM9（S3C2440）中，SRAM有效地址范围为0~4K，所以可以把栈初始指针设置在SRAM的有效地址的最高地址处，当然，如果空间够用，也可以设置小点儿，比如：ldr sp , = 2\*1024 。

2.在on\_sdram中：  
    ldr sp, =4096           @ 设置中断模式栈指针  
    ldr sp, =0x34000000     @ 设置系统模式栈指针，  
这是分别设定中断模式和系统模式下的堆栈指针到4096（SRAM的有效地址的最高地址）和0x34000000（从0x30000000开始的64M的SDRAM的最高地址处，此时SDRAM已经初始化，可以使用了）。

**栈的作用：**

1.保存现场/上下文  
现场/上下文，意思就相当于案发现场，总有一些现场的情况，要记录下来的，否则被别人破坏掉之后，你就无法恢复现场了。而此处说的现场，就是指CPU运行的时候，用到了一些寄存器，比如r0,r1等等，对于这些寄存器的值，如果你不保存而直接跳转到子函数中去执行，那么很可能就被其破坏了，因为其函数执行也要用到这些寄存器。因此，在函数调用之前，应该将这些寄存器等现场，暂时保持起来(入栈push)，等调用函数执行完毕返回后(出栈pop)，再恢复现场。这样CPU就可以正确的继续执行了。  
保存寄存器的值，一般用的是push指令，将对应的某些寄存器的值，一个个放到栈中，把对应的值压入到栈里面，即所谓的压栈。然后待被调用的子函数执行完毕的时候，再调用pop，把栈中的一个个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从栈中弹出去，即所谓的出栈。  
其中保存的寄存器中，也包括lr的值（因为用bl指令进行跳转的话，那么之前的pc的值是存在lr中的），然后在子程序执行完毕的时候，再把栈中的lr的值pop出来，赋值给pc，这样就实现了子函数的正确的返回。

2.传递参数  
C语言进行函数调用的时候，常常会传递给被调用的函数一些参数，对于这些C语言级别的参数，被编译器翻译成汇编语言的时候，就要找个地方存放一下，并且让被调用的函数能够访问，否则就没发实现传递参数了。对于找个地方放一下，分两种情况。一种情况是，本身传递的参数不多于4个，就可以通过寄存器传送参数。因为在前面的保存现场的动作中，已经保存好了对应的寄存器的值，那么此时，这些寄存器就是空闲的，可以供我们使用的了，那就可以放参数。另一种情况是，参数多于4个时，寄存器不够用，就得用栈了。

3.临时变量保存在栈中

包括函数的非静态局部变量以及编译器自动生成的其他临时变量。

位置无关代码码与位置相关代码

    位置无关与位置相关代码是关于arm程序在跳转时的寻址方式的两种，一般情况下两种方法都能达到跳转到目的地址的目标，但是在某些特定的环境下，两种跳转方法得到的结果相关较大。

位置无关代码的跳转可通过“B或BL 标号”命令执行，当执行B或BL命令实现跳转时，实际的二进制代码在跳转时是执行：计算可执行代码中目标地址到当前PC值处的距离，然后把该距离值加上当前的PC值，然后赋值给PC寄存器实现跳转，该跳转方法由于指令的地址域只有26位，所以它只能向前/向后寻址32M位地址，如果目标地址到当前PC值的距离大于该值时，可使用伪指令“ADR和ADRL PC,=标号地址”实现跳转，实际执行过程是先计算目标址址到当前PC的距离，然后把当前PC值加上距离值赋值到目标寄存器PC中，实现跳转，ADR与B或BL功能类似，但ADRL可以实现全部范围内的跳转，其通过把目标地址与当前PC的距离值存储到一个缓存字中，然后把缓存字中的值加载到寄存器PC中。

位置相关跳转也可以叫绝对位置跳转，一般使用伪指令LDR PC,=label实现跳转到label标号的链接地址中，绝对地址也就是链接地址。这里位置相关及位置无关本质上的区别其实是链接地址与运行地址的区别。

在无操作系统中的裸机代码中，如果链接的偏移基址为0，则绝对位置跳转与相对位置跳转的效果是一样的，而对于一些把可执行代码拷贝到ram中再运行的程序中，链接地址与起始运行地址是不相同的，一般地链接地址是一个ram逻辑地址内的一个数值，而运行地址则是代码烧写的flash等存储器件的逻辑地址，一般是0地址。链接的偏移地址可以通过在编译时通过在链接脚本或Makefile中指定。

在完成可执行代码从flash到ram的拷贝后，通过LDR PC,=label指令跳转到内存中执行程序，**而在这之前执行的代码跳转必须是位置无关的，否则会出现跑飞的异常。**

ldr r0, \_start  
\* 从内存地址 \_start 的地方把值读入。执行这个后，r0 = 0xe1a00000  
\*  
\* adr r0, \_start  
\* 取得 \_start 的地址到 r0，但是请看反编译的结果，它是与位置无关的。

**BSS段和data段，text段**

一般C语言的编译后执行语句都编译成机器代码，保存在.text段；已初始化的全局变量和局部静态变量都保存在. data段；未初始化的全局变量和局部静态变量一般放在一个叫.“bss”的段里。 我们知道未初始化的全局变量和局部静态变量默认值都为0，本来它们也可以被放在.data段的，但是因为它们都是0，所以为它们在.data段分配空间并且存放数据0是没有必要的。 程序运行的时候它们的确是要占内存空间的，并且可执行文件 必须记录所有未初始化的全局变量和局部静态变量的大小总和，记为.bss段。 所以.bss段只是为未初始化的全局变量和局部静态变量预留位置而已，它并没有内容，所以它在文件中也不占据空间。

# STM32自己知识的漏洞

周期时间=((1+TIM\_Prescaler )/72M)\*(1+TIM\_Period )

72M/（1+TIM\_Prescaler（预分频值））

死区的作用：

通俗点：设计带死区的pwm波可以防止上下两个器件同时导通。也就是说，当一个器件导通后关闭，再经过一段死区，这时才能让另一个导通。

# 使用最新Ubuntu下遇到的问题以及解决方案

如果想一直进入root权限模式的话那就输入sudo su就可以了，退出就用su 用户名

韦东山给的Ubuntu版本是9.10太老了，所以我就去Ubuntu官网上下载了最新版本的Ubuntu14.04（在写这篇文章时候的最新版本）的64位系统，然后一步一步地在虚拟机安装上了然后接下来的问题就层出不穷，搞了一两天才走出阴影

首先winscp和CRT连不上，这是因为ssh服务没有打开，我们就需要安装openssh-server但是在安装之前我们需要更新一下apt-get

所以运行sudo apt-get update（这里有个小插曲，在用14.04版本的时候我当时不懂就下载了一个14.10版本的，结果怎么换更新源都没有用，后来查了一下Ubuntu已经不维护14.10版本了所以不行了）

接下来执行sudo apt-get install openssh-server

sudo ps -e |grep ssh看一下如果有sshd就表示启动成功如果没有输入sudo service ssh start命令就可以了，这样就可以远程登录Ubuntu了

然后又发现了一个问题，就是没有arm-linux-gcc交叉编译工具链

所以我又在网上下载了一个4.3.2版本的编译器，然后接下来安装该编译器，安装过程网上一搜一大堆大部分都是正确的所以在此就不罗嗦了

然后接下来就会有个蛋疼的问题了，那就是当执行arm-linux-gcc –v的时候会出现

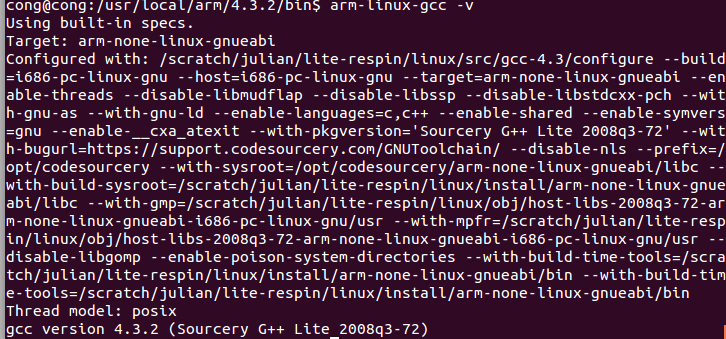


这个问题纠结了半天后来在网上看到有人说这个编译器是32位的而系统是64位的不兼容缺少32位库

接下来执行sudo apt-get install ia32-libs这句话但是发现没有，是因为这个版本已经不能用这个了，就又在网上搜最后找到了答案，输入如下两个命令就可以了就已经包括大部分的32库了

sudo apt-get install lib32z1  
sudo apt-get install lib32stdc++6

接下来再运行arm-linux-gcc –v就可以了



安装编译器的方法：

下载好安装包

然后执行 tar zxvf arm-linux-gcc-4.4.3.tar.gz -C/或者也可以tar zxvf arm-linux-gcc-4.4.3.tar.gz /目标目录 -C/ 之后再看编译器是否符合要求

然后找到这个编译器的版本号的文件夹比如4.4.3之类的然后在usr/local/下创建目录arm然后把4.4.3复制到usr/local/arm/下

接下来改写环境变量

先echo $PATH查看一下环境变量

然后export PATH=你的编译器路径：/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games就可以了

# Nandflash的ECC问题

我们的nandflash空间会很大，但是nandflash有一个缺陷，就是不一定会有某个位出现位反转，导致数据的错误，首先我们应该了解一下nandflash的结构，nandflash的存储是按页存储，存是整页整页地存，读也是整页整页地读，写也是整页整页写，nandflash也分大页和小页，我们现在用的mini2440的大小是256m的它的一页大小是2048b+64b，一共有2048个块，一块里面有64个页

我们分析一下一页中的64k是什么，我们看一下整个nandflash的读写流程

我们为了防止出现位反转而导致的数据错误，nandflash里面有一个ECC校验，我们每一个页里面都会多出64字节的空余位置那里叫oob，那里就会放置ECC的校验码

首先我们写nandflash：

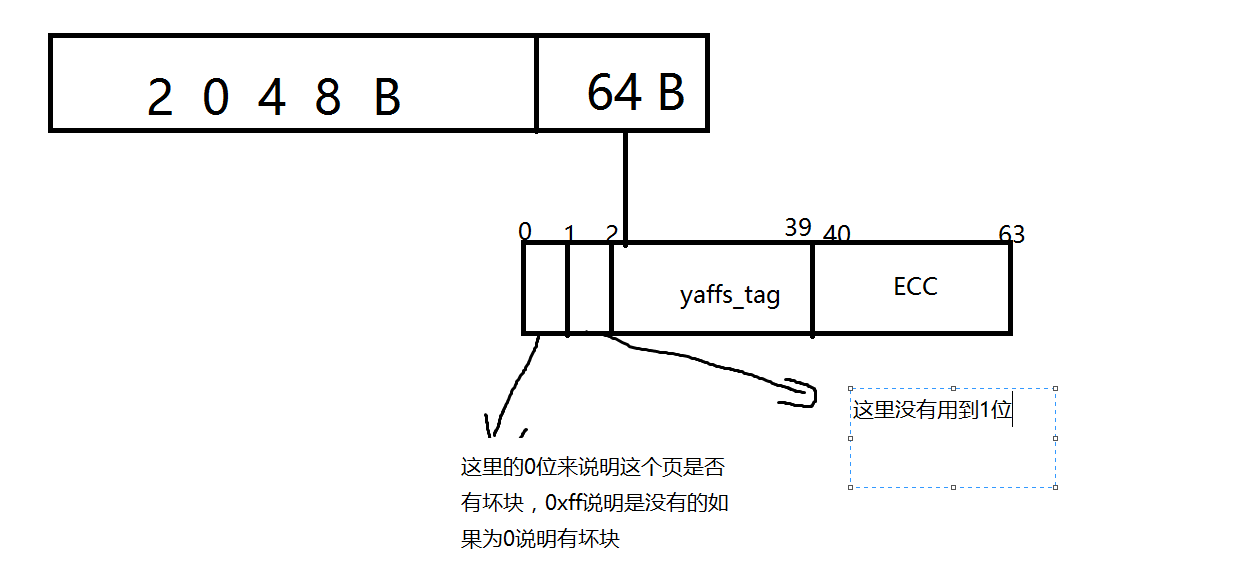
1. 写一页数据
2. 将写的内容转换成ECC校验码存放到oob当中去

然后我们读nandflash

1. 将nandflash的一页数据读出来
2. 再读出oob当中的ECC
3. 将读出的数据转换成ECC
4. 比较两个ECC然后找出哪个位反转了
5. 反转的位修正回去

**Yaffs工具特点**

我们已一张图来说明



这张图是nandflash中一个页的详细分析图，前面是存放一个页的数据后面是oob，我们在mkyaffs2image.c中可以看到yaffs是对oob怎么来划分的

我们的页中每256b对应ECC的3个字节当我们写数据的时候yaffs会生成一个结构体放到yaffs\_tag空间中，然后再生成ECC校验码

# 大端和小端的问题

首先我们应该了解一下什么是大端和小端，这两个都是处理器的一种数据存储形式。

大端：

把低地址放到高地址中去

比如说0x11223344这是一个32位的数据，也就是4个字节，低地址为44 高地址为11，比如说这里内存有

0x01 0x02 0x03 0x04，

11 22 33 44

这样就是大端的存放形式

小端：

其实就是跟大端是相反的，就是低地址存放到低地址中去也就是：

0x01 0x02 0x03 0x04，

44 33 22 11

我们的x86架构和ARM架构都是用的小端模式 51用的是大端模式，ARM是可是设置成大端和小端模式的，但是这并不影响我们的编程，该怎么配置寄存器就怎么配置寄存器，没有什么地址倒置问题

但是需要注意的是当我们用一个别的设备连接处理器的时候，我们需要看一下是大端模式还是小端模式，如果两者一个是大端一个是小端那就需要一些处理程序来处理它了