国嵌的视频：

http://pan.baidu.com/s/1qWuKriG

http://pan.baidu.com/s/1jHnwBRo?third=0&num=20&dir=%2F国嵌学院%2F专业体系2-精通嵌入式&page=1

国嵌嵌入式收费版全套课程(珍贵)

http://pan.baidu.com/share/link?shareid=869039712&uk=3928540337#path=%252F

概览:

\*交叉工具链+连接器脚本

\*指令集，cpsr，cp15

\*启动方式和流程

\*uboot第一阶段，cpu相关

\*uboot第二阶段，外围设备相关，start\_armboot

\*混合汇编

\*栈结构和任务切换栈

\* MMU，中断，nand，uart，网口

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

裸机开发以后主要用于:

\*bootloader,类似于bios

\*linux驱动程序

-驱动接口规范和系统接口

-操作寄存器

安装工具链:

tar xvzf arm-linux-gcc-4.3.2.tgz -C /

修改环境变量，使得哪里都可以访问到arm-gcc，把交叉工具链添加到环境变量中:

在/etc/profile文件最后一行添加

或在/root/.bashrc文件最后一行添加

export PATH=$PATH:/usr/local/arm/4.3.2/bin

然后运行source /etc/profile 就可以更新环境变量了

那再运行arm-linux-gcc就不会找不到了

profile: 这个文件是每个用户登录时都会运行的环境变量设置，当用户第一次登录时，该文件被执行。

并从etc/profile.d目录的配置文件中搜集shell的设置

arm-linux-gcc -g -c led.S

arm-linux-ld -Tled.lds -o led.elf led.o

arm-linux-objcopy -O binary led.elf led.bin;-O后面紧跟着输出格式

用到的: 交叉工具链，链接脚本，makefile

下载led.bin：

\*烧写mmc\_256到 SD卡

\*从SD卡启动后擦除flash

\*然后选择3从usb下载bin

\*在linux上连接usb

\*安装dnw驱动dnw\_usb.ko, insmod dnw\_usb.ko

\*./dnw ../led.bin 50008000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

交叉工具链，这个链就是一个集合

file hello-arm 查看文件在哪个平台使用

交叉开发模型: 程序在宿主机产生,运行环境在目标机，宿主机和目标机平台是不一样的

常用交叉工具

arm-linux-gcc

arm-linux-ld

arm-linux-readelf

arm-linux-objdump ;反汇编器

arm-linux-objcopy ;文件格式转换工具

arm-linux-gcc --help

gcc 从 /user/include 寻找头文件

arm-linux-gcc 从 /user/local/arm目录下找

arm-linux-ld -Tled.lds -o led.elf led.o main.o

arm-linux-readelf -a led.elf

arm-linux-readelf -d hello ; 可以查看这个hello用到了哪些库函数, 这些库函数开发板上是否存在

arm-linux-objdump -D -S hello > dump; 反汇编器, 如果程序在编译的时候带-g, 那么反汇编里面会带C语言的信息

arm-linux-objcopy：直接编译生成的elf格式, 不能直接在arm上进行执行,能运行的是二进制的文件

linux系统上能运行elf文件,是因为linux系统上有elf解析器。

arm-linux-objcopy -O binary led.elf led.bin

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

连接器脚本

作用

-将若干输入文件(.o文件)根据一定规则合并为一个输出文件(例如elf文件)

脚本构成：

基本构成：段

起始链接地址

对齐

变量

代码段首文件

OUTPUT\_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")

OUTPUT\_ARCH(arm) //arm-linux-objdump -f led.elf可以获得machine architecture

ENTRY(\_start)

SECTIONS {

. = 0x30008000; //.表示当前位置

. = ALIGN(4); //=号左右都要空格, 此语句要以;结束

.text : //段名之后有空格有冒号

{

start.o(.text) //start.o 的代码段放这个位置

\*(.text) ;\*代表所有文件

}

.data :

{

\*(.data)

}

bss\_start = . ;//变量

.bss :

{

\*(.bss)

}

bss\_end = . ;

}

设置起始链接地址:

.=0x30008000 //.表示当前位置

对齐:

.LIGN(4);

变量：

bss\_start = .;

设置代码段首文件:

.text : //段名之后有冒号

{

start.o(.text) //start.o 的代码段放这个位置

\*(.text) ;\*代表所有文件

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

加载地址==存储地址

运行地址==链接地址，程序运行的时候在这个地址上跑

SECTIONS

{

.text\_rom\_exec1 : /\* No copy to INTRAM \*/

{

\_vector\_start = .;

\*/init.o(.Vector)

\*/init.o(.Boot)

} > flash //运行地址和加载地址都在flash

.text :

{

\*/init.o(.Irq)

\*/mbx\_drv.o(.text .rodata\*) /\* DON'T REMOVE \*/

\*/fw\_update.o(.text .rodata\*) /\* DON'T REMOVE \*/

} > intram AT > flash //运行地址在ram，加载地址在flash

\_itext\_start = LOADADDR(.text); //获取加载地址

\_text\_start = ADDR(.text); //获取运行地址

\_text\_end = \_text\_start + SIZEOF(.text);

}



Effio的启动流程:

-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Eclipse集成开发环境

-可以用于在线调试，另外可以把编译链接的功能集成进来

软件内容：

\*安装arm-linux-gcc,arm-linux-gdb

-./build-all

-添加gcc和gdb的路径/root/.bashrc

\*eclipse

\*Jlink软件

\*JLinkGDBServer

使用：

\*eclipse开发环境调用makefile, 调用arm-linux-gcc来进行编译

\*启动 JLinkGDBServer,eclipse->arm-linux-gdb->JLinkGDBServer->Jlink软件来对开发板进行调试

-前提是要从nand flash启动, 并且要先从sd卡启动格式化nand flash

-Jlink接线要正确

格式化nand flash

连接板子

-Jlink

-串口

-nand启动

\*arm-linux-gcc,arm-linux-gdb的安装

./build-all ;编译然后再安装

/root/.bashrc

一定要先export新安装的gdb路径, 先定义先找

\*Jlink软件的安装

Jlink\_linux\_v434a.tgz

cp -d libjlinkarm.so\* /usr/lib -f

cp 45-jlink.rules /etc/udev/rules.d

\*./JLinkGDBServer 可以查看到开发板连接信息, 另外打开eclipse之前, 一定要打开JLinkGDBServer

\*安装Eclipse

which eclipse ;查看目前安装的eclipse,这个版本不用

解压缩即可

安装一个插件

-help->install new software->http://opensource.zylin.com/zylincdt

eclipse其实是调用我们自己的makefile来编译我们的程序,

工具自动会在环境变量中找到。

smb共享时只看到目录看不到文件

-原因是selinux的问题, 可以/etc/selinux/config文件中的SELINUX="" 为 disabled ，然后重启。

\*create makefile with existing code

\*debug config->zylin 创建调试配置

调试的是elf文件

底层和框架层最有前途

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM家族

芯片：210， 2440, 6410

ARM核: A8, ARM9, ARM11

指令结构: ARMv7, ARMv4, ARMv6

ARM核的变迁过程

ARM7,ARM9,ARM11

Cortex-M(工控), Cortex-R(实时), Cortex-A(多媒体)

芯片对比: 2440 6410 210

处理速度 500M 533M 800M

缓存 16K 16K 32K

内存接口 SDRAM DDR DDR2

支持的OS LINUX LINUX/ANDROID LINUX/ANDROID

其他

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM的工作模式

ARM Architecture Reference Manual.pdf 网上的知识太零碎,其实所有的知识都来源于ARM公司的文档

汇编程序

哪些代码需要用到汇编:

\*bootloader,启动代码,建立C语言环境之前

\*内核,对效率有特殊要求的地方, 混合编程

ARM汇编的分类:

\*ARM标准汇编: 适合ARM公司的汇编器,适用于windowss平台

\*GNU汇编: 适用于GNU交叉汇编工具,适合linux开发平台

-以后在工作中会用到GNU汇编

汇编程序框架:

.section .data

<初始化的数据>

.section .bss

<未初始化的数据>

.section .text

.global \_start

\_start:

<汇编代码>

实际工作中.section也取消了

makefile

all:start.o

arm-linux-ld -Ttext 0x50000000 -o start.elf $^

%.o:%.S 把所有的[.S]文件都编译成[.o]文件.

arm-linux-gcc -g -o $@ $^ -c

clean:

rm \*.o \*.elf

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7种寻址

-操作数在哪，就是什么寻址方式

\*立即数寻址, 操作数直接给出

ADD R0,R0,#0x3f; 0x也可以用&,代表16进制

\*寄存器寻址, 操作数存放在寄存器

ADD R0,R1,R2;

\*寄存器间接寻址, 操作数的地址存放在寄存器

LDR R0,[R1]

\*基址变址寻址, 操作数的地址=寄存器基址 + 偏移地址

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, 操作数涉及多个寄存器

LDMIA R0,{R1,R2,R3,R4}; R1<-[R0]

; R2<-[R0+4]

; R3<-[R0+8]

; R4<-[R0+12]

\*相对寻址，操作数在堆栈里。以PC为基址，加上相对地址进行跳转

BL NEXT

NEXT:

...

\*堆栈寻址

stmfd sp, {r1-r3};满堆栈递减

学习指令的时候，都思考一下每条指令属于什么寻址方式

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CPSR标志位：

N:运算结果b31位的值。

Z:运算结果为0，则Z=1; 运算结果为非0, 则Z=0

C:加法的时候, 第31bit往前进位, 则C=1,否则为0. 减法时, 第31bit往前有借位则C=0, 否则C=1;对于包含移位的非加减操作，C为移出的位。

V:仅当从0x80000000跨越到0x7fffffff会产生V=1, 0X80000000 - 4, V=1.其他情况不会

指令分类

\*跳转指令

-B

-BL

-BX

-BLX

-MOV PC,LR

\*数据传输指令

如果有2个操作数，第一个操作数必须为寄存器，第二个操作数可以是寄存器，立即数，寄存器移位

mov r1,#3

mov r1,r0

mov r1,r0,lsl #3

mvn r0,#0xff;和mov类似，不过是立即数取反后传输

cmp r1,r0;进行一次减法运算，结果不保存，只会更新CPSR标志位

cmp r1, #100

tst r1,#0b101; %也代表二进制，按位与，会影响Z位

add r0,r1,r2

add r0,r1,#256

add r0,r2,r3,lsl #1; r0=r2+(r3 << 1)

sub r0,r1,r2

sub r0,r1,#256

sub r0,r2,r3,lsl #1

and r0,r0,#3;清某些位，位与操作

orr r0,r0,#3;设置某些位，位或操作

bic r0,r0,#%1011;清除r0中的013bit，并把结果存储到r0中

mul r0,r1,r2

muls r0,r1,r2

\*程序状态寄存器访问指令

mrs r0,cpsr

mrs r0,spsr

msr cpsr\_c,r0;[31:24]位cpsr\_f;[23:16]位cpsr\_s;[15:8]位cpsr\_x;[7:0]位cpsr\_c;

\*加载和存储指令,操作内存单元

ldr r0,[r1]

ldr r0,[r1,r2]

ldr r0,[r1,#8]

ldr r0,[r1,r2]!;r1=r1+r2

ldr r0,[r1,#8]!

ldr r0,[r1],r2;r1=r1+r2

ldr r0,[r1,r2,lsl #2]!;将存储器地址为r1+r2\*4的数据读入r0，并更新r1=r1+r2\*4

ldr r0,[r1],r2,lsl #2;将存储器地址为r1的数据读入r0,并更新r1=r1+r2\*4

ldrb r0,[r1]; 只是把内存单元的低8位存入r0,高24bit清0

ldrb r0,[r1,#8]

ldrh r0,[r1]; 只是把内存单元的低16位存入r0,高16bit清0

ldr r1,=0x50000000; 这是一个伪指令，相当于mov

str r0,[r1],#8

str r0,[r1,#8]

ldr r0, =0x50008000

数据块传送:ia, increase after; ib, increase before

ldmia r0, {r1-r3};每次传输后r0+4, 数据先传送给r1

ldmib r0, {r1-r3};每次传输前r0+4, 数据先传送给r1

ldmda r0, {r1-r3};每次传输后r0-4, 数据先传送给r3

ldmdb r0, {r1-r3};每次传输前r0-4, 数据先传送给r3

stmia r0, {r1-r3};每次传输后r0+4, r1数据先传送

stmib r0, {r1-r3};每次传输前r0+4, r1数据先传送

stmda r0, {r1-r3};每次传输后r0-4, r3数据先传送

stmdb r0, {r1-r3};每次传输前r0-4, r3数据先传送

堆栈操作

stmfd sp, {r1-r3};满堆栈递减

stmfa sp, {r1-r3};满堆栈递增

stmed sp, {r1-r3};空堆栈递减

stmea sp, {r1-r3};空堆栈递增

{!}, 最终更新基址寄存器

{^}, LDM当寄存器列表中包含r15,还要将SPSR复制给CPSR

\*数据交换指令，存储器与寄存器之间交换数据

swp r0,r1,[r2];[r2]->r0, 同时r1->[r2]

swp r0,r0,[r1]

swpb r2,r3,[r1];r3的低8位写入到[r1]低8bit,[r1]其他bit不受影响, 但是[r1]的低8bit赋值给r2

\*移位指令

\*lsl,逻辑左移,低位用0填充,1左移出会置C标志

\*asl,算术左移，低位用0填充,1左移出会置C标志

\*lsr,逻辑右移,高位用0填充，1右移出会置C标志

\*asr,算术右移，高位用符号位填充，1右移出会置C标志

\*ror,循环右移, 左端用右移出的位来填充，1移出也会置C

mov r0,r1,lsl #2

mov r0,r1,ror #2

\*异常产生指令

swi 0x02;用户请求系统api调用，api号为0x20, 立即数只能是24bit

bkpt, 貌似我们这款CPU不支持bkpt命令

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM伪指令，就像C语言连定义数据结构，结构体，变量，一些不是程序执行的部分

-符号定义

-数据定义

-汇编控制

-宏指令

-其他

\*符号定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

指令分类:

\*算术和逻辑指令

-mov r1,#4

-mvn r1,#0b101

-cmp r1,r2

-tst r1,#0b101; %也代表二进制

-sub r1,r2

-add r3,r2,#4

-and r1,r1,#0b101

-bic r1,r1,#0b101

\*比较指令

-cmp r1,r2; 比较结果影响N Z

-tst r1, #ob101; 按位与

\*跳转指令

-b label;相当于goto

-bl ;相当于子程序调用, 虽然自动保存下一条指令地址在lr，但是函数执行完之后要手动mov pc,lr

\*移位指令

-mov r1,r1,lsl#1

-mov r1,r1,ror#1

\*程序状态字访问指令

-msr cpsr,r0

-mrs r0,cpsr

\*存储器访问指令

-ldr

-str

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM伪指令

-ARM机器码

-定义类伪指令

-操作类伪指令

\*ARM机器码

-汇编程序->(汇编器)->机器码->(运行)->CPU

分析机器码:

看到机器码0000->EQ 汇编指令, 看到汇编指令 EQ->0000 操作码

立即数是由12位组成,一个是immed\_8,immed\_rote 4bit用于左移右移

\*定义类的伪指令

-伪指令不会产生机器码

-在编译的时候起作用

-或转化为其他的指令来运行, 例如nop,转化为mov指令来完成空操作

.global \_start; 定义全局标号

.data ;定义数据段

.oscii; 定义字符串

.byte ;定义字节数据

.word ;定义一个字

.equ DA,0x89

.align 4;控制从什么对齐的地方开始

arm-linux-readelf -a start.elf 可以查看各个数据段和全局变量,相当于查看map文件

\*操作类伪指令

-nop; 实际上是 mov r0,r0; 作用是延时

-ldr r0, =0x1ff; 实际上是ldr r0, [pc+#164], 把0x1ff存储到内存中, 然后从内存中取到r0

伪指令可以通过arm-linux-objdump -D -S start.elf来查看伪指令被翻译成了什么指令

mov r0,#0x1ff; build的时候出错, 会显示无效的常量, 不符合立即数要求, 立即数必须可以由8位的数据移位偶数位得到

ldr r0,#0x1ff; 也出错了, 因为格式不对

ldr r0,=0x1ff; 这个是伪指令,立即数由=号指定, 立即数可以超过8位

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

协处理器访问指令

-什么是协处理器

-协处理器访问

\*什么是协处理器

-ARM可支持多达16个协处理器,其中CP15是最重要的一个.

CP15的作用

-系统控制协处理器,提供寄存器配置和控制caches,MMU,protected system,时钟模式,大小端

如何访问CP15:

-CP15定义了16个寄存器.

-使用MCR,MRC来访问这16组寄存器

要记住CP15的16组寄存器的意思

MCR{cond} P15,<Opcode\_1>,<Rd>,<CRn>,<CRm>,<Opcode\_2>, register ->co-cpu15

MRC{cond} P15,<Opcode\_1>,<Rd>,<CRn>,<CRm>,<Opcode\_2>, co-cpu15 ->register

-对于<Opcode\_1>,<CRd>,<CRm>,<Opcode\_2>在手册的表里面都给出来了

mcr p15,0,r0,c0,c0,0 ;访问MAIN ID

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

不想做设计师的程序员不是好的程序员,做一个架构设计师

Bootloader设计蓝图

-Bootloader的作用

-Bootloader的设计方法

-U-Boot快速体验

-U-Boot代码工程建立

Bootloader + Bootloader parameters, 就是一个助推器的作用,完成(启动内核)助推作用之后,就交给系统了

Bootloader-助推器

Kernel-航天飞机

设计方法

-模仿,90%的设计都是从模仿开始

-Uboot是老大,都模仿它

U-Boot

-用于多种CPU(ARM, MIPS, PPC)

-支持多种操作系统(LINUX,VxWORKS)

uboot自动启动linux-自主模式

uboot启动过程中按键进入命令模式-开发模式

-可以通过tftp下载内核到ram当中,来做试验

U-Boot代码工程建立

-uboot.tar.gz必须拷贝到linux下解压缩,因为linux区分大小写,而windows不区分,解压缩后会出问题

网络映射到本地Z盘

如何在source insight上添加\*.s,\*.S

-Options->Document Options->Document Type:C Source file, File filter: \*.c,\*.h;\*.s;\*.S

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM处理器启动流程:ARM处理器要求先做什么后做什么,所以uboot才设计成先设置什么后设置什么

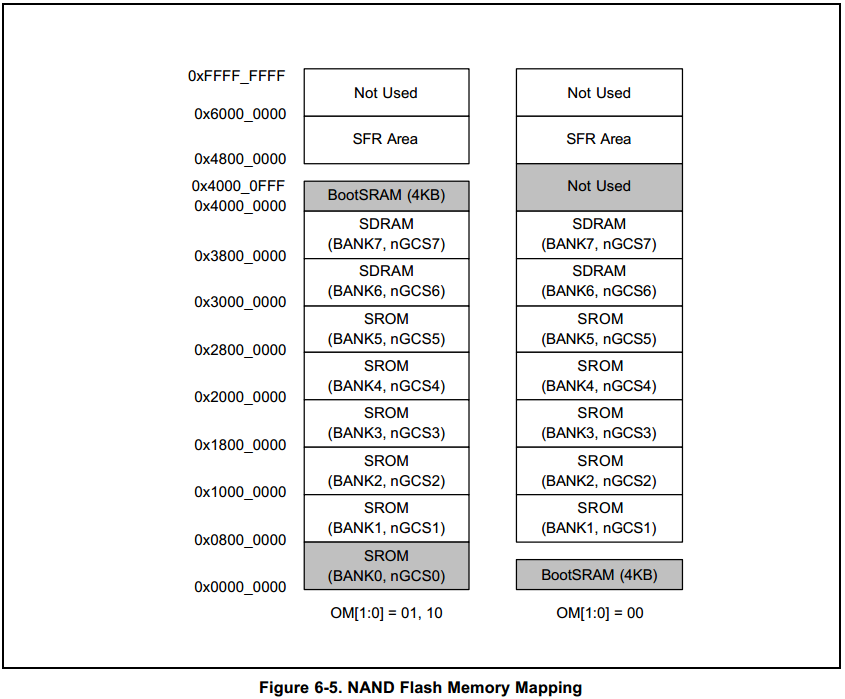
-启动方式

-地址布局

-启动流程

2240:

\*启动方式, 搜索mapping



-Nor falsh一般2M

-Nand falsh一般256M, 不能直接读取,要表读表写,一般有4K的bootsram(stepping stone)

SDRAM, NOR FALSH统一编址,nand falsh不能参与统一编址，因为不能直接读取，要用寄存器读取

nor flash：可以跑程序

nand flash：不能跑程序

one nand：可以跑程序，接口和nor一样，但是接口后内部和nand一样

\*启动流程

如果设置从nand falsh启动

-当系统上电,nand的前4k会被irom程序加载到stepping stone 4k(从0地址开始)内部ram中,

-执行stepping stone的程序完成硬件初始化,把剩余的nand falsh bootloader内容加载到sdram中

-跳转到sdram中剩余的bootloader进行执行

启动第一条指令从0地址开始取指

关于nand flash的启动流程，可以参考第6章

irom: 其实就是芯片出厂的时候芯片厂商固化在rom中的一段程序，用于拷贝存储设备的bootloader到stepping stone。.

-irom初始化系统时钟，初始化设备控制器

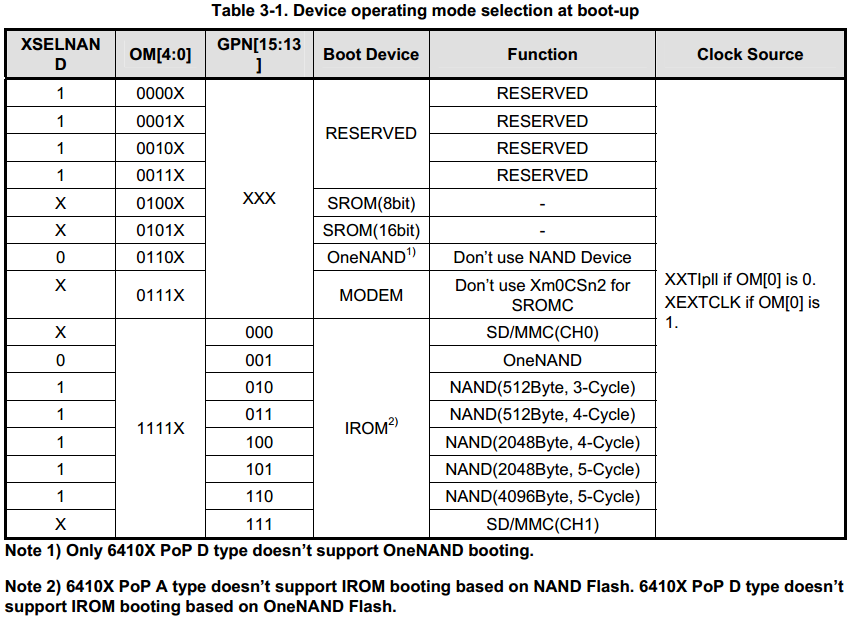
-irom拷贝nand flash中的bootloader到stepping stone，并校验这部分程序

-irom跳转到stepping stone执行bootloader

6410:

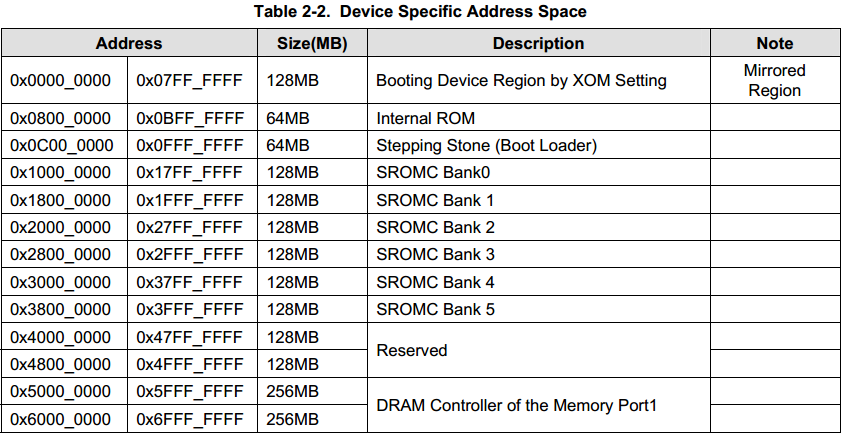
\*启动方式

搜索booting

SROM: NOR FALSH

IROM: 芯片内部rom启动. 地址是非0

\*地址布局

0地址处的功能: Mirror region, 这里只起到映射的作用.

-可以映射到从IROM区域启动

\*irom can be mapped to boot image area when internal ROM booting is selected. irom被连接到boot image area (0x00000000)

\*Address range of boot image area is from 0x0000\_0000 to 0x07FF\_FFFF, but there is no real mapped-memory.没有真正的地址

0x0000\_0000 ~ 0x07FF\_FFFF: 128M, Booting Device Region by XOM Setting, Mirrored Region

0x0800\_0000 ~ 0x0BFF\_FFFF: 64MB, Internal ROM, 地址有64M，实际只有32K，可以被映射到0地址

0x0C00\_0000 ~ 0x0FFF\_FFFF: 64MB, Stepping Stone (Boot Loader), 实际上只有4K

0x6000\_0000 ~ 0x6FFF\_FFFF: 64MB, DDR, 0x6000\_0000是规划分配给DDR的，这个地址送入到逻辑，逻辑就会映射到相应的总线上，从而选择了DDR，然后根据DDR控制器的寄存器值来对DDR进行时序访问

假如我们映射到从IROM区域启动,且各种引脚设置成nand flash4G 启动

那么:

从0地址开始->IROM->拷贝NAND到IRAM(stepping stone).

\*启动流程(假设选择IROM->NAND启动)

IROM: 芯片厂商固化的一段程序,也称为BL0

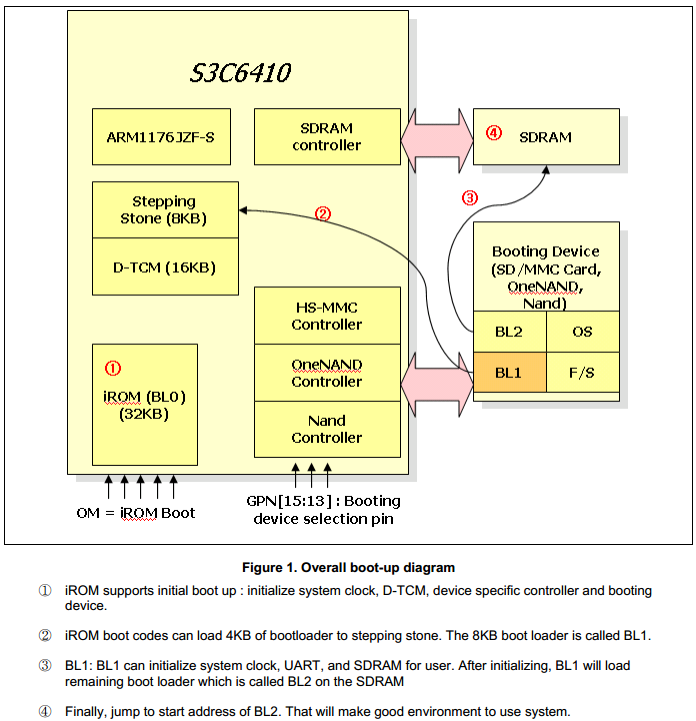
-因为0地址映射到IROM, 所以执行IROM里面的内容

-IROM程序做硬件初始化,同时把nand flash的BL1(8K)拷贝到stepping stone(8K)

-跳转到stepping stone(8K)

-BL1运行后会把BL2拷贝到SDRAM,然后跳转过去运行

S3C6410\_Internal\_ROM\_Booting.pdf



\*从IROM的NAND启动还是从IROM的SD卡启动，由XSELNAND, OM[4:0], GP[15：13]决定

\*芯片有两个P-BANK, 对于第一个P-BANK，可以由NAND/NOR/MODEM/SROM 内存控制器接管, 选择哪个控制器接管如何设置？

-Booting mode

-MP0\_CS\_CFG regsiter residing in System Controller

-XSELNAND input port value

这样就定出了由什么内存控制器接管这个P-BANK，并制定了哪些pin脚来作为CS信号来选通芯片。

\*对于第二个P-BANK，只能由DRAM控制器来控制，所以不用像第一个P-BANK那样，要配置给哪个内存控制器使用。

210:

\*启动流程

-IROM执行,拷贝BL1到stepping stone, BL1必须小于16K

-BL1运行的时候把BL2拷贝到stepping stone(BL2<=80K)或SDRM(BL2>80K),BL2要完整的拷贝,要么都拷贝到stepping stone,要么都拷贝到SDRAM

-跳转到BL2

bootdevice: 可以是nand,nor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

U-Boot工作流程分析

-程序入口

-第一阶段程序分析: BL1

-第二阶段程序分析: BL2

2440:

\*程序入口

-一定要找到

-从makefile入手, 用的是smdk2440

-Z:\s3-arm\part4\uboot\board\samsung\smdk2440\u-boot.lds

ENTRY(\_start)

cpu/s3c24xx/start.o (.text)

Z:\s3-arm\part4\uboot\cpu\s3c24xx\start.S //这里就是BL1代码

在BL1代码中只关心做了什么,而不关心怎么做的. 把Uboot要做的事情(注释有说明),用思维导图记录下来

第一阶段程序分析:

-设置中断向量表

-把处理器设置成svc的模式

-刷新I/D caches

-关闭MMU和Cache

-关闭看门狗 //lowlevel\_init

-关闭中断

-初始化系统时钟

-初始化串口

-对nand进行简单初始化

-进行内存的初始化

-执行nand\_copy,复制nand falsh中的BL到内存中

-设置堆栈

-清除bss段

-ldr pc, \_start\_arm\_boot;直接跳转到sdram继续执行BL2, \_start\_arm\_boot在

编译uboot:

make smdk2440\_config

make

从u-boot.elf文件中看到\_start的地址是 0x30000000, 不是说在stepping stone(0地址)中运行吗?

\_start\_arm\_boot: 0x30001100,这个地址是SDRAM的地址, 0x30001100 这个地址是怎么来的?

0x30000000,0x30001100连接地址是怎么来的?

-由链接脚本指定, 在uboot/config.mk, 这个是通用的,有:

LDFLAGS += -Bstatic -T$(LDSCRIPT) -Ttext $(TEXT\_BASE) $(PLATFORM\_LDFLAGS)

-那这里有两个代码段基地址,一个是基于连接器脚本0000开始,一个是基于TEXT\_BASE,会以TEXT\_BASE为准

TEXT\_BASE在哪里定义呢?

Z:\s3-arm\part4\uboot\board\samsung\smdk2440\config.mk //这个是本芯片特殊的地方,定制化的

TEXT\_BASE = 0x30000000

不是说在stepping stone(0地址)中运行吗?

在第一阶段: pc指针的指<4k

当跳转到sdram后,pc的值就是0x30000000以上了

bl lowlevel\_init;lowlevel\_init的地址也是0x30000000以上的,为什么这里不会跳转到0x30000000以上呢?

-bl指令跳转的是相对跳转,而不是绝对跳转

通过elf可以看到链接地址:

30008000: bl lowlevel\_init; //实际执行的时候,假如此时pc=100,则跳转到100+(30008010-30008000)

...

30008010: lowlevel\_init

总结:

bl lowlevel\_init //相对跳转

ldr pc, \_start\_arm\_boot //绝对跳转

刚开始的时候pc=0,那么BL1中执行的都是相对跳转,所以一直在stepping stone中运行,等到执行ldr pc, \_start\_arm\_boot绝对跳转,才跳转到sdram中执行程序

start\_armboot: 做硬件初始化和软件初始化，主要是和板子相关

第一阶段:和CPU有关

第二阶段程序分析: 和CPU外围设备有关

\*\_start\_arm\_boot函数开始,这里已经是C语言代码了

-执行init\_sequence里面的初始化,这里只关心硬件

-串口初始化

-lcd初始化

-网卡初始化

-初始化led

-main\_loop: 执行用户输入的命令,开发模式

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6410:

\*程序入口

-从makefile入手, 用的是smdk6410

-Z:\s3-arm\part4\uboot\board\samsung\smdk6410\u-boot.lds

ENTRY(\_start)

cpu/s3c64xx/start.o (.text)

Z:\s3-arm\part4\uboot\cpu\s3c64xx\start.S //这里就是BL1代码

第一阶段程序分析:

-设置中断向量表

-把处理器设置成svc的模式

-刷新I/D caches

-关闭MMU和Cache

-外设基地址初始化 //6410

-点亮led //lowlevel\_init

-关闭看门狗

-关闭所有中断

-初始化系统时钟

-初始化串口

-对nand进行简单初始化

-进行内存的初始化

-执行nand\_copy,复制nand falsh中的BL到内存中

-设置TTL表，使能MMU

-设置堆栈

-清除bss段

-ldr pc, \_start\_arm\_boot;直接跳转到sdram继续执行BL2, \_start\_arm\_boot在

如何编译:

make smdk6410\_config, 在uboot目录下执行这条命令后，结果是为后面的编译做配置

make，在uboot目录下执行这条命令后，开始编译用于smdk6410的uboot

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

210:用高版本的uboot

2440和6410, 只有一个uboot, 而210有2个bin, BL1(24K), BL2(24K开始以后的空间)

iram == stepping stone

\*函数入口

/uboot-210/spl/u-boot-spl.lds 可以看到

arch/arm/cpu/armv7/start.o(.text) 这里就是它的BL1入口

第一阶段程序分析:

-设置中断向量表

-把处理器设置成svc的模式

-invalidate L1 I/D cache //

-关闭MMU和Cache

-检查reset状态 //lowlevel\_init

-IO引脚初始化,恢复默认值

-关闭看门狗

-SRAM,SROM初始化 //210

-关闭所有中断

-初始化系统时钟

-进行内存的初始化

-初始化串口

-bl tzpc,取消存储保护区 //210

-对nand进行简单初始化

-disable ABB

-设置堆栈

-nand\_boot\_210:复制nand flash中的BL2到SDRAM

-(void \*)config\_sys\_text\_base执行 跳转到BL2

第二阶段程序分析: BL2

arch/arm/cpu/armv7/u-boot-spl.lds 可以看到

arch/arm/cpu/armv7/start.o(.text) 这里就是它的BL2入口 和 BL1一样

-判断是否在内存当中,则跳转到board\_init\_ram->board\_init\_f 这才是BL2真正的入口

save\_boot\_params

\_\_attribute\_\_((weak, alias("save\_boot\_params\_default")))

-没有函数体

-weak, 弱函数,定义同名函数时,这个函数自动失效

-alias, 别名.如果在另外的地方没有重新定义save\_boot\_params, 则用save\_boot\_params\_default代替

-清除bss段

-ldr pc, \_start\_arm\_boot;直接跳转到sdram继续执行BL2, \_start\_arm\_boot在

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bootloader架构设计

G-Boot架构设计

-第一阶段程序设计

-第二阶段程序设计

第一阶段程序设计

\*核心初始化

-设置中断向量表

-把处理器设置成svc的模式

-关闭看门狗

-关闭中断

-关闭MMU和Cache

-外设基地址初始化 //6410

\*C语言编程环境设置

-设置堆栈

-清除bss段

\*Led初始化

\*初始化系统时钟

\*进行内存的初始化

-内存初始化

-取消存储保护区 //210

-irom/iram初始化 //210

\*执行nand\_copy,复制nand falsh中的BL到内存中

-对nand进行简单初始化

-复制代码到sdram

-ldr pc, \_start\_arm\_boot;直接跳转到sdram继续执行BL2

第二阶段程序设计

\*MMU初始化

\*中断初始化

-中断初始化

-按键初始化

\*串口初始化

-串口初始化

-移植printf函数

\*网卡的初始化

\*lcd初始化

-触摸板初始化

-LCD初始化

\*解析执行用户命令

-移植tftp命令

-移植bootm命令

第2季-裸奔吧-ARM, 上期其实就是第一阶段程序设计,下期其实就是第二阶段程序设计.

要学会用思维导图来设计软件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARM工作模式

http://blog.csdn.net/tankai19880619/article/details/8085184

异常向量表

-概念解析

a.异常定义

b.异常类型

c.异常入口

d.向量表

-代码编写

异常定义:

因为内部或外部的一些事件,导致处理器停下正在处理的工作,转而去处理这些发生的事件.

异常的类型:7种

名字 固定地址(异常向量) 模式

reset 0x00000000 svc

undefined instructions 0x00000004 und

software interrupt(SWI) 0x00000008 svc

Prefetch Abort(instruction fetch memory abort) 0x0000000c abort

Data Abort(data access memory abort) 0x00000010 abort

IRQ 0x00000018 irq

FIQ 0x0000001c fiq

上表在ARM Architecture Reference Manual.pdf的54页有定义。

也可以选择High vector address

异常向量:(固定地址)

当一种异常发生的时候,ARM处理器会跳转到对应该异常的固定地址去执行异常处理程序,

而这个固定的地址,就称之为异常向量

异常向量表:

由7个异常向量及其处理函数跳转关系组成的表即异常向量表

向量 处理指令

0x00000000: b reset

0x00000004: ldr pc,\_underfined\_instruction

0x00000008: ldr pc,\_software\_interrupt

0x0000000c: ldr pc,\_prefetch\_abort

0x00000010: ldr pc,\_data\_abort

0x00000014: ldr pc,\_not\_used

0x00000018: ldr pc,\_irq

0x0000001c: ldr pc,\_fiq

手把手教你写代码

-start.S

-gboot.lds

-makefile

问题: 起始地址是0x50008000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

设置svc模式 0b10011

mrs r0,cpsr

bic r0,r0,#0x1f

orr r0,r0,#0xd3

msr cpsr,r0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

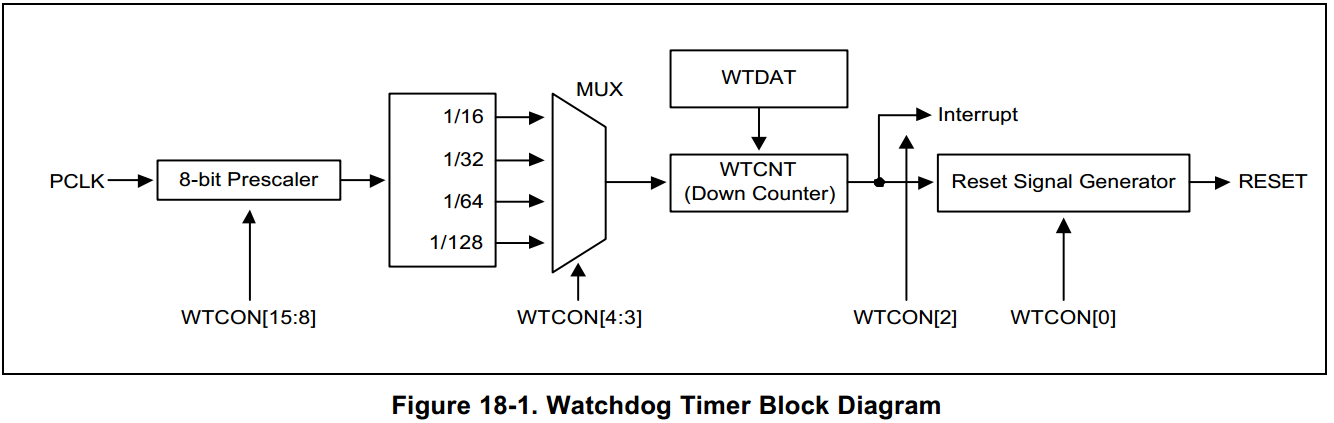
设置看门狗

嵌入式系统死机的时候,看门狗能感受到这种情况,对系统进行自动重启.

再重启之前可以配置是否送出中断.

如果设置看门狗的超时时间为5分钟,那么程序在5分钟之内就要喂一次狗.

Figure 34-1. Watchdog Timer Block Diagram



看门狗其实就是一个硬件定时器，也可以配置成超时产生中断，在中断里，可以记录临终遗言等。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

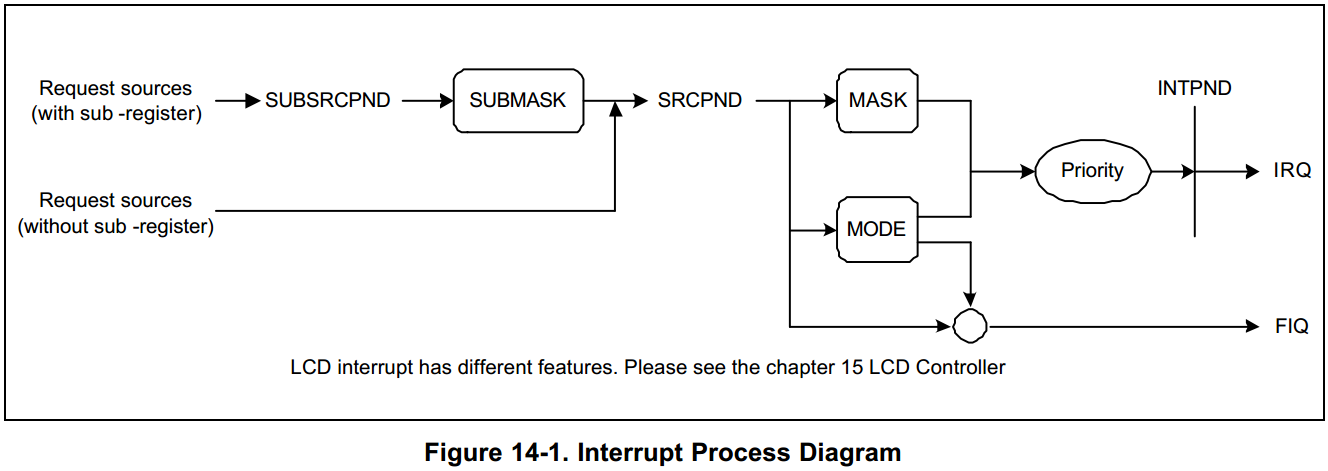
关闭中断

-CPSR中关闭I和F位

-中断屏蔽寄存器

中断屏蔽寄存器

2440 Figure 14-1 Interrupt Process Diagram



**F-bit and I-bit of Program Status Register (PSR)**

If the F-bit of PSR in ARM920T CPU is set to 1, the CPU does not accept the Fast Interrupt Request (FIQ) from the

interrupt controller. Likewise, If I-bit of the PSR is set to 1, the CPU does not accept the Interrupt Request (IRQ) from

the interrupt controller. So, the interrupt controller can receive interrupts by clearing F-bit or I-bit of the PSR to 0 and

setting the corresponding bit of INTMSK to 0.

**Interrupt Pending Register**

The S3C2440A has two interrupt pending registers: source pending register (SRCPND) and interrupt pending register

(INTPND). These pending registers indicate whether an interrupt request is pending or not. When the interrupt

sources request interrupt the service, the corresponding bits of SRCPND register are set to 1, and at the same time,

**only one bit of the INTPND register is set to 1 automatically after arbitration procedure**. If interrupts are masked, then

the corresponding bits of the SRCPND register are set to 1. This does not cause the bit of INTPND register changed.

**When a pending bit of INTPND register is set, the interrupt service routine will start whenever the I-flag or F-flag is**

**cleared to 0**. The SRCPND and INTPND registers can be read and written, so the service routine must clear the

pending condition by writing a 1 to the corresponding bit in the SRCPND register first and then clear the pending

condition in the INTPND registers by using the same method.

在中断处理的最后，要清除SUBSRCPND,SRCPND,INTPND的相应位要清除，因为要为接受下一次

中断的到来做准备。如果没有清，那下一次中断来，就会丢中断了。接收中断的那一段涉及的

记录寄存器都要清除。

INTMASK

SRCPND->INTMSK->PRIORITY->INTPND->ISR

|->INTMOD(irq,fiq)

\*6410

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

关闭mmu和cache

ARM存储体系:快到慢，近到远

-处理器内部寄存器

-TCM:紧耦合存储器(cache,主存储器)

-辅助存储器(flash,sd)

为什么要cache:

不用cache的话,处理器直接访问内存,cpu一直等着内存

cache是一种容量小,但是存取速度非常快的存储器,,它保存最近用到的存储器中的数据的拷贝.

-I-Cache

-D-Cache

write buffer:

-把要写到内存的数据先写到write buffer中，等CPU空闲的时候，数据才会慢慢地被搬移到内存中

ARM1176 sheet PAGE 146

什么是虚拟地址:

-几个应用程序相同的虚拟地址可以映射到不同的物理地址

要正确配置MMU和Cache之后，才能打开

MMU和Cache都是通过CP15来控制的，在ARM1176 手册中

disable\_mmu:

mcr p15,0,r0,c7,c7,0 ;invalidate I cache and D cache

mrc p15,0,r0,c1,c0,0

bic r0, r0, #0x00000007

mcr p15,0,r0,c1,c0,0 ;disable I cache, disable D cache, disable MMU

mov pc, lr

流程:

\*使I cache和D cache失效

\*关闭D cache, 关闭MMU，I cache可关可不关

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

系统时钟

时钟分两种:

-晶振, 石英经过精密切割,接上电极,同上电,会产生机械震荡,稳定,但是频率低

-PLL, 由晶振或外部时钟输入,经过PLL可以倍频分频,输出多个频率

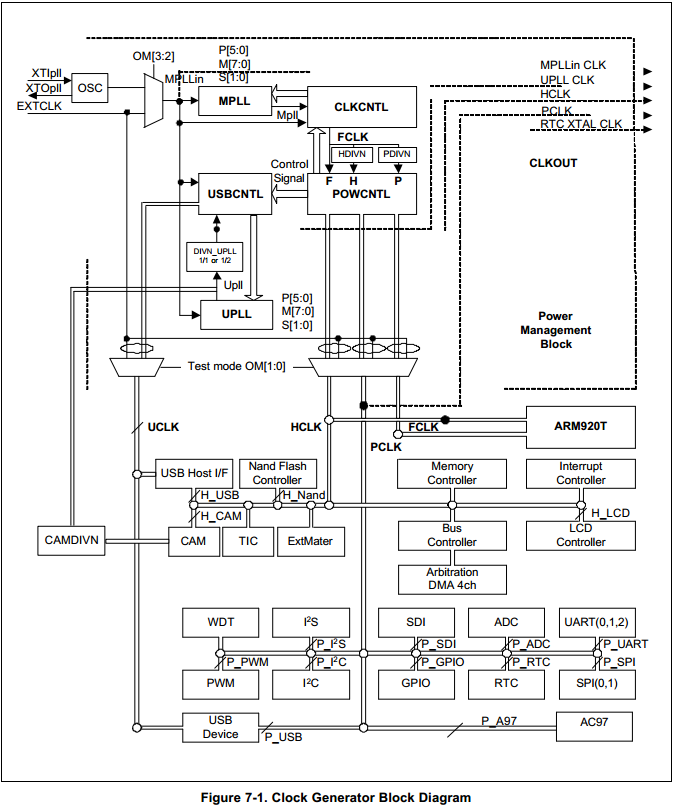
配置系统的时钟要点:

-外部晶振要求多少MHz

-时钟模块有几个PLL

-每个PLL输出什么时钟信号

-每个时钟信号控制什么设备



另外,所有的其他时钟信号,都是FCLK乘以一个系数得到

时钟脉冲信号

从以下4个点来学习时钟体系:

\*了解晶振频率

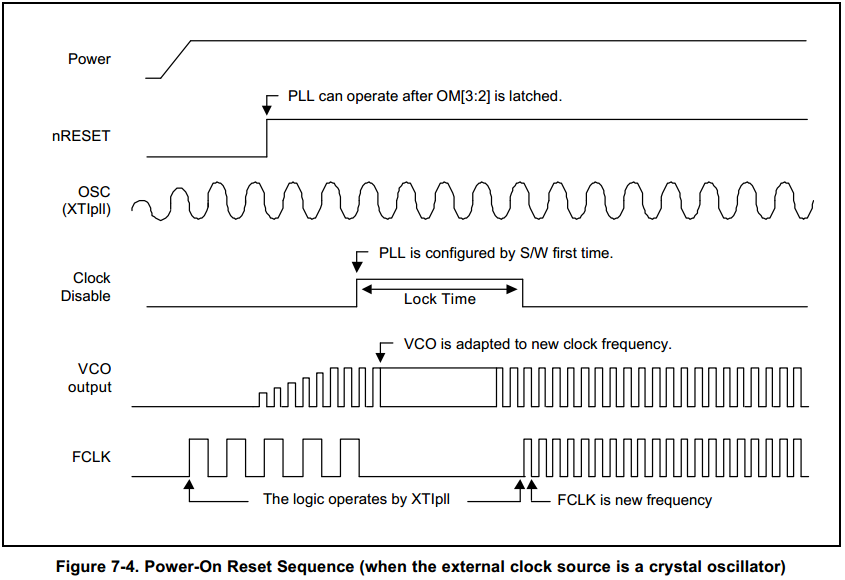
\*有多少个PLL

\*每个PLL分别产生哪些时钟

\*这些时钟是用来做什么的

完成PLL的配置之后，进入lock time，lock time过程中PLL不输出任何频率

lock time过后，PLL就按照配置的频率输出.



配置时钟流程:

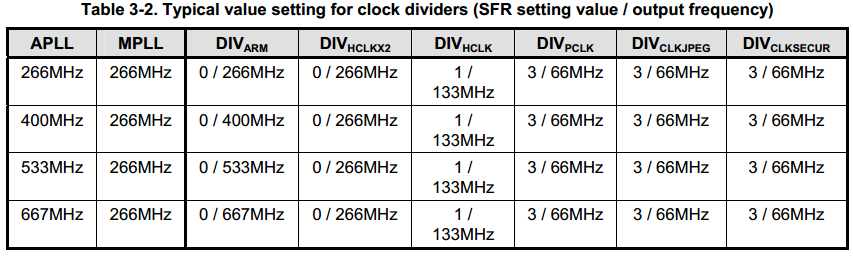
\*配置lock time

\*给PLL外部设置分频系数, 有经典设置表 Table 3-2. Typical value setting for clock dividers

\*把CPU设置成异步工作模式还是同步工作模式

\*PLL刚开始输出和晶振一样的频率，这里要配置PLL内部系数，一般有参考表

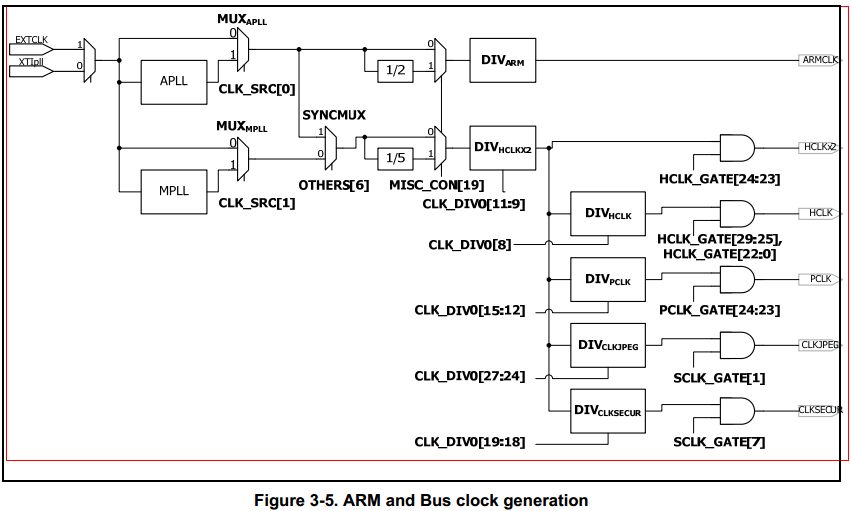
\*如果有选择Gate的地方要设置



另外，配置PLL外部分频系数的值，可以参考uboot。

如果FCLK不等于HCLK(不是来自同一个源，没有倍数同步关系)，要把CPU设置成异步工作模式，不能设置成快速、同步工作模式

\*6410 P125,P142



晶振->APLL->时钟->分频->ARM CLK

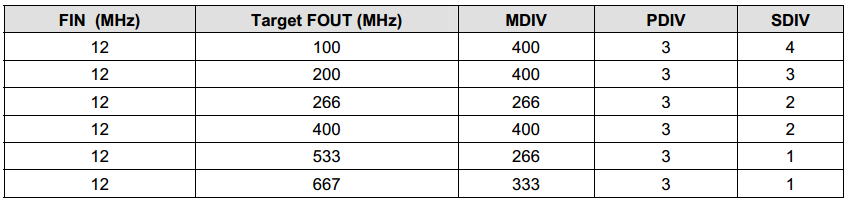
晶振->MPLL->时钟->分频->HCLKX2->分频->HCLK

|->PCLK

最重要其实就是PLL的输出频率和各个时钟的分频系数

具体设置成什么值，参考uboot lowlevel\_init.s

P142, 设置APLL,MPLL的输出时钟



有2种方法读手册

-整个篇章全看

-以某一个信号或寄存器位为入口，搜索

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

我从内部看内存

内存分类

\*SRAM, 不用电容充电, 接近CPU的那部分高速内存

\*DRAM，需要给小电容充电(刷新)，随机访问哪个地址

-SDRAM,同步动态随机存储器，需要一个同步时钟

-DDR,Double Data Rate SDRAM, 上升沿和下降沿都传输数据，所以速率是SDRAM的2倍

-DDR2, 速率是DDR的2倍

2240: SDRAM

6410: DDR

210: DDR2

内存内部结构

\*表结构,内存内部结构如一张表格，由行地址和列地址找到存储单元格

\*L-Bank，一块内存芯片一般由多个L-BANK组成，logical bank

\*寻址信息

-L-BANK选择信号

-行地址

-列地址

推导内存容量:

=4 \* L-BANK容量

=4 \* (单元格数量 \* 每个单元格容量)

每个单元格的容量，芯片手册上有说，同时这个也是数据宽度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2440内存芯片初始化

-2440地址空间

-内存芯片硬件连接

-存储控制器寄存器

-手把手写代码

地址空间

-地址线

++27根，访问范围是2^27=128M, 但是如果增加片选信号，范围就加大了

-片选信号

++提供nGCS0~nGCS7 8个片选信号，范围是8\*128=1G

-外设空间

-存储控制器(最主要初始化这个)

++ARM核->存储控制器(将地址分解成l-bank选择信号，行列地址)->内存,norflash,网卡芯片

内存芯片硬件连接

-选取的内存芯片是4\*4M\*16bit=32M,数据宽度是16bit，但是2440的数据线是32根

-所以2440的低16根数据线接到第1块32M内存，高16根数据线接到第2块32M内存，就实现了一次访问32bit数据了

除了数据线和QDM以外，其他控制线从CPU出来送给2块内存是一样的。

-这就实现了2块32M内存芯片(2块HY57V561620)并联构成了64M的容量,接到Bank6的接口

设置2440的存储控制器

-直接看寄存器一个个设置，从寄存器入手，看都没看内存芯片一眼

#define mem\_contrl 0x48000000

init\_sdram:

ldr r0, =mem\_contrl

add r3, r0, #4\*13

adrl r1, mem\_data

0:

ldr r2, [r1], #4

str r2, [r0], #4

cmp r0, r3

bne 0b

mov pc, lr

mem\_data:

.long 0x22000000

.long 0x00000700

.long 0x00000700

.long 0x00000700

.long 0x00000700

.long 0x00000700

.long 0x00000700

.long 0x00018001

.long 0x00018001

.long 0x008c04f5 //行有效时间为7个CLK？芯片手册写的最大6个CLK

.long 0x000000b1

.long 0x00000030

.long 0x00000030

重点域解析:

ST7:控制QDM用的是什么信号线，在原理图上搜看是否用了UB/LB，如果没有则用的是nWBE[3:0]

DW7:数据宽度是32bit

Trcd:从RAS到CAS的延迟，看图是2个时钟

SCAN:有多少根列地址线，从SDRAM芯片手册中搜索coloum就可以知道是9根

Trp:行准备充电时间，看图是2个时钟

Tsrc:行有效时间，在这个有效时间内，电容电量有效，超过则要充电

CL:仅针对读有意义，从列地址出现到有数据出现在DQ的时钟个数

Refresh Count: 64ms要刷8192行，所以每行的刷新周期Refresh period为7.8us,带入公式算count

P-BANK:一组内存芯片的集合，可能是2块16bit芯片合成32bit

L-BANK:一个内存芯片内部有若干个L-BANK

1bit:1个存储体，但是数据交换是以一个单元格为单位的

SDRAM基础知识，搜索百度文库: SDRAM原理

http://wenku.baidu.com/view/adf56e81e53a580216fcfeec.html?from=search

通过CS,RAS,CAS,WE,BA,ADDR来实现各种命令操作

命令，由连接CPU的那些芯片数据线的高低状态组合来实现的。

-行激活,激活某个存储体的某行，使之进入准备读写状态。

-预充电，用于对已激活的行进行预充电，即结束活动状态。猝发写操作必须保证在写入预充电命令

前操作已经完成，并使用DQM禁止继续写入数据。预充电结束后，回到空闲状态，也可以再次被激活。

空闲->激活->读写->预充电->空闲->激活

-自动预充电，A10/AP 位置为1，在猝发读写操作完成后会自动附加一次预充电，然后回到空闲状态。

-猝发读，允许某个体中的一行被激活后，连续读出若干个数据。第一个数据在经过CAS延时节拍后呈现在数据

线上，以后每个时钟节拍都有一个新的数据。猝发读的行内初始列地址可以任意，不要求对界，

但是地址变化范围被限制在4或8个字节内，猝发读结束后，数据总线呈高阻态，可以在适当的节拍上再次发起

猝发读命令，使数据线上保持没有空隙。

-猝发写，猝发写命令与猝发读命令类似，允许某个体中的一行激活后，连续写入若干数据。

第一个写数据与猝发写命令同时在数据线上给出，以后每个时钟节拍给出一个新的数据，输入缓冲在

猝发数据量满足要求后停止接受数据。

-自动刷新，由于动态存储单元存在漏电现象，必须保证在64ms内对所有的存储单元刷新一次。

一个自动刷新周期刷新一行(所有体)，然后刷新行地址加1.只有所有体都空闲，并且未处于低功耗模式

时才能启动自动刷新操作，刷新操作完后进入空闲。该期间7.8us(8192行)刷新一行，也可以在64ms内的某个时间段对所有单元集中刷新一遍。

-自刷新，通常用于低功耗模式下保持SDRAM的数据。自刷新下，禁止了所有的内部时钟和输入缓存，为了降低功耗，刷新地址和刷新

时间全部由器件内部产生。一旦进入自刷新方式只有通过CKE置低才能激活。如果在自动刷新方式下是集中式刷新，那么在退出自刷新模式后

也会执行一次集中式的自动刷新操作。

-时钟和时钟屏蔽，时钟信号是所有操作的同步信号，上升沿有效。时钟屏蔽信号CKE决定是否把时钟输入施加到内部电路。

CKE置高有效。CKE变低后的下一个节拍冻结输出状态和猝发地址，知道CKE变为高为止。在所有体都处于空闲状态时，CKE变低后的下一个节拍SDRAM

进入低功耗模式并保持到CKE变高为止。

-DQM，用于屏蔽输入输出操作。对于读操作DQM延迟2个时钟周期开始起作用，对于写操作则当拍有效。

所有信号线，控制信号，地址信号，数据信号:

CLK:芯片时钟输入

CKE:片内时钟信号使能

/CS:片选，如果为高，则禁止CLK,CKE,DQM以外的所有输入信号

BA0，BA1: 可以用于片内4个L-BANK的选择

A11~A0: 行地地址:A11~A0，列地址:A7~A0。A10是自动预充电标志

/RAS:行地址锁存

/CAS:列地址锁存

/WE:写使能

LDQM,UDQM:读模式下控制输出缓存；写模式下屏蔽输入数据

DQ15，DQ0:数据输入输出引脚

VDD/VSS:电源/地

NC:

关键要知道:

\*内存控制接口是怎么样通过信号完成读写SDRAM的

\*各种重要信号线及其功能

\*有哪些重要的命令

\*\*\*\*\*\*\*\*\*\*

6410内存初始化

-地址空间

-硬件连接

-芯片手册导读

地址空间

-保留区:0x80000000~0xFFFFFFFF,2G,目前没用

-外设区:0x70000000~0x7FFFFFFF,256M

-主存储区:0x0000000~0x6FFFFFFF,1972M

主存储区

-0x50000000~0x6FFFFFFF, 512M,Dynamic memory, DDR内存放在这里了

-0x40000000~0x4FFFFFFF, Reserved,256M

-0x10000000~0x3FFFFFFF, Static memory, 768M

-0x0C000000~0x0FFFFFFF, I\_SRAM, 8K/64M

-0x08000000~0x0BFFFFFF, I\_ROM,32K/64M

-0x00000000~0x07FFFFFF, Boot Image, 128M

2\*128M = 256M，2个DDR内存芯片，一个使用高16bit数据线，一个使用低16bit的数据线

参考uboot是如何初始化的

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

代码搬移不可少

由于垫脚石尺寸有限，所以BL1要把剩下的代码从nand flash拷贝到内存

-起点

++6410:0x0c000000

-终点

++链接地址: lds 里面的地址, 0x50008000(\_start)

-搬移方式

链接地址的作用

-如果调用reset(), 实际上就是pc赋值成reset()的链接地址

-ldr pc, =reset; pc直接被赋值成reset()的链接地址

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

跃进C大门

使用绝对跳转, ldr pc, =gboot\_main; 这个是伪指令

汇编中调用函数:

ldr pc, =main

C语言调用汇编函数

.global light\_led

light\_led();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

混合编程

-C中嵌入汇编指令

-汇编中访问C的全局变量，当做立即数来使用

-C中调用汇编函数

-汇编中调用C函数

\*C内嵌汇编

-格式,使用"\_\_asm\_\_"或"asm"

\_\_asm\_\_(

汇编语句部分,多条语句使用\n或;隔开

:输出部分

:输入部分

:破坏描述部分, "memory" or "r1"

);

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" @如果没有破坏部，这一行删除掉

);

}

void read\_p15\_c1()

{

unsigned long value

\_\_asm\_\_(

"mrc p15,0,%0,c1,c0,0\n"

:"=r"(value) @‘=’表示只写操作数, 用于输出部分

:

:"memory"

);

}

int getvalue()

{

int c,d;

\_\_asm\_\_ volatile(

"mov r1,#3\n"

"mov %0,r1\n" @%0作为第一个输出部

"orr %1,%0,#4" @%1作为第二个输出部

:"=r"(c),"=r"(d)

:

:"memory"

);

}

int gboot\_main()

{

#define GPKCON 0x7f000000

#define GPKDAT 0x7f000004

\_\_asm\_\_ valatile(

"ldr r1,=0x11110000\n"

"str r1,[%0]\n"

"ldr r1,=0x11110000\n"

"str r1,[%1]\n"

:

:"r"(GPKCON),"r"(GPKDAT) @这是输入部

:"r1" @对r1寄存器有影响

);

}

volatile告诉编译器不要优化这段代码

如何定义输出部: 就是会改变你这个变量的

混合汇编，必须是

-输入是C语言数据，输出是汇编寄存器或反过来

-一条汇编语句中必须从%0开始

\*汇编中访问C的全局变量

-int geric = 0x88888888; C语言中

-ldr r0,=geric; //汇编中，这里r0的值存入的不是0x88888888，而是geric的地址

\*C中调用汇编函数

.global str\_cpy ;在汇编中声明这个是全局函数

str\_cpy:

ldrb r2,[r1],#1

cmp r2,#0

beq set\_peri\_port

strb r2,[r0],#1

b str\_cpy

void str\_cpy(char \*dst, char \*src); ;在c语言中声明这个函数

str\_cpy(buf,string); ;C语言中调用这个函数

\*汇编中调用C函数

-ldr pc,=gboot\_main

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

栈初始化

fp指向函数栈的第一个元素，sp(r13)指向函数的最后一个元素

堆栈开始的时候，sp没有指向任何栈元素

函数调用栈帧:

\_\_\_\_\_

lr0 <-fp1 高地址

fp0

...

...

\_\_\_\_\_

lr1 <-fp2

fp1

...

...

\_\_\_\_\_

lr2 <-fp3

fp2

...

...

\_\_\_\_\_

fp3 ->这个函数没有函数调用，所以不用压栈lr指针

...

...

\_\_\_\_\_ 低地址

分析调用栈的时候，可以通过fp3->fp2->fp1->fp0找出每个函数的栈帧，同时也找出了lr2->lr1->lr0, 加上异常的时候打出的lr，则lr->lr2->lr1->lr0, 这个就是调用关系

push {fp,lr}, lr 先入栈, fp后入栈, 并且sp-=8

pop {fp,lr}, fp 先出栈， lr后出栈,并且sp+=8

str fp, [sp, #-4]!; 意思是说把fp存储到[sp-4],然后跟新sp的值sp=sp-4

r0~r3, 最多传输4个函数参数，更多的函数参数需要利用栈来传递

当前的fp，意思是当前函数的栈帧开始位置

栈中的fp(r11)，上一级函数的栈帧开始位置，可以把栈帧连接起来

栈中的lr(r14)，把函数调用连接起来

/\*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; PERFORM A CONTEXT SWITCH (From task level)//任务切换的堆栈结构由操作系统决定(压榨所有寄存器)，函数自己的堆栈结构由编译器决定(压榨必要的寄存器)

; void OSCtxSw(void)

;

; Note(s): 1) Upon entry:

; OSTCBCur points to the OS\_TCB of the task to suspend

; OSTCBHighRdy points to the OS\_TCB of the task to resume

;

; 2) The stack frame of the task to suspend looks as follows:

;

; PC (High memory)

; LR(R14)

; R12

; R11

; R10

; R9

; R8

; R7

; R6

; R5

; R4

; R3

; R2

; R1

; R0

; OSTCBCur->OSTCBStkPtr ----> CPSR (Low memory)

;

;

; 3) The stack frame of the task to resume looks as follows:

;

; PC (High memory)

; LR(R14)

; R12

; R11

; R10

; R9

; R8

; R7

; R6

; R5

; R4

; R3

; R2

; R1

; R0

; OSTCBHighRdy->OSTCBStkPtr ----> CPSR (Low memory)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

瞻前顾后

第一阶段: 使用汇编初始化CPU

第二阶段: 使用C语言初始化板载硬件，如网口，串口

ARM 的ram包括静态ram，动态ram，TCM。 TCM是一个固定大小的RAM，紧密地耦合至处理器内核，提供与cache相当的性能，相比于cache的优点是，

程序代码可以精确地控制什么函数或代码放在那儿(RAM里)。当然TCM永远不会被踢出主存储器，因此，他会有一个被用户预设的性能，

而不是象cache那样是统计特性的性能提高。

TCM对于以下几种情况的代码是非常有用、也是需要的:可预见的实时处理(中断处理)、

时间可预见(加密算法)、避免cache分析(加密算法)、或者只是要求高性能的代码(编解码功能)。

如同缓存的哈佛结构，指令TCM和数据TCM是分开的。TCM有两种使用方式:作为快缓存使用，和作为本地内存使用。

本地内存

这时，TCM被用作更快速的内存，如同一般的RAM。因为指令段有时也是数据访问的对象，

指令TCM实际上是指令数据一体化TCM。对TCM写操作后和后续对此写操作的依赖指令之间必须跟一个阻塞操作。

快缓存(smartcache)

TCM可以配置成当作外部RAM的缓存使用，对应的外部RAM也要设置可缓存标志。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

下学期

专题1-世界一下变大了-MMU\第1课-MMU功能解析

对比单片机和ARM

MMU的作用：

-将虚拟地址转化为物理地址

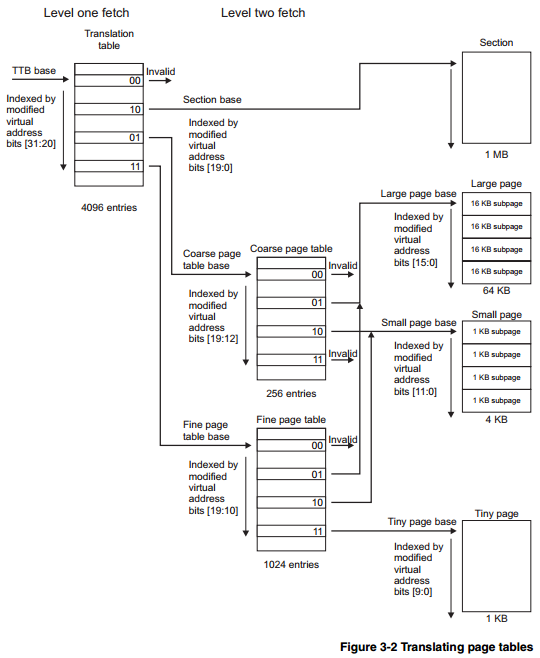
-访问权限管理

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

深入剖析地址转换

地址转化流程总体分析

Figure 3-2 Translating page tables



第一级转换表

表放在什么地方

-表放在ram

这张表是谁建立的

-工程师建立的

MMU如何知道这张表在哪里

-TTB，这张表的基地址通过CP15，c2寄存器提交给MMU

第二级转换表

-10,段转换 \*, 1M

-01,粗页转换

-11,细页转换 \*, 4K,1K,64, 物理页，页帧

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MMU配置与使用

以段为例：

-建立一级页表

-写TTB到cp15的c2寄存器

-打开MMU

这段代码对于所有板子都一样的

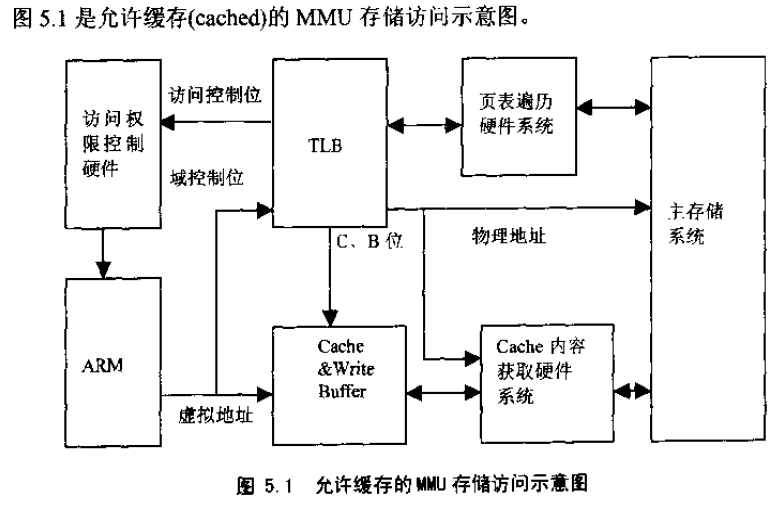
一级页表的起始地址:

440: 0x30000000

6410:0x50000000

210: 0x20000000

CPU核--cache--mmu--writebuf--内存



ap和domain和control的SR bit决定访问权限

-control的SRbit, 参考ARM11 Figure 3-26 Control Register format, P176

-AP, 参考ARM 9 PAGE 73

-domain, 参考ARM11， page 3-63, 当前entry指定的空间指定属于哪个域，对于没有TTB表的ARM，另外有寄存器设置每个domain的空间起始和大小

如果遇到一些伪指令不会, 查询一下GCC编译器手册就可以，查MPLAB编译器手册就可以。

TTB和TLB的关系:

-TLB其实就是把TTB放到cache里面

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第1课，中断处理流程深度剖析

-中断生命周期

-中断源

-中断过滤

-中断处理

中断生命周期

外设--中断控制器(过滤)--CPU核处理

2440：非向量处理方式

-进入中断程序总入口

-保存环境

-判断中断源

-调用对应中断源的处理程序

-恢复环境

6410/210：向量式处理方式

-直接跳转到对应的中断处理程序

-保存环境

-设备的中断处理

-恢复环境

软件需要做的事情:

-初始化中断源

-初始化中断控制器

-中断处理

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2240：

程序优化

-模块化

中断编程

-初始化按键

-初始化控制器

-中断处理

在中断处理的最后，要清除SUBSRCPND,SRCPND,INTPND的相应位要清除，因为要为接受下一次

中断的到来做准备。如果没有清，那下一次中断来，就会丢中断了。接收中断的那一段涉及的

记录寄存器都要清除。

要对各个模式下设置SP。svc,irq,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6410

基于向量的中断

-也就是说，每个中断源都有一个寄存器用于存储中断处理函数的地址，一旦中断来了，

就直接调用这个地址的函数进行处理

中断编程

-初始化按键,找到按键对应的GPIO，并配置成中断方式

-初始化控制器

--配置下降沿触发

--配置MASK寄存器

--使能中断，有按bit的使能

--使能中断啊，在CPSR中总的使能

--使能向量中断，并配置中断向量地址寄存器

-中断处理

--保存环境

--中断处理

--清除中断

--恢复环境

在12.2 FEATURES里面就有提到下面的内容

-有针对每个中断使能(这个有时候和MASK是一回事)

-有针对总体中断使能

-有RAW中断信号

-有最终输出中断信号

-RAW到最终,中间经过MASK

-可以选择每个中断设置成IRQ或FIQ

-中断优先级设置

-可以设置软中断模仿硬件中断产生,记录在RAW

-中断控制寄存器可以设置为只允许特权模式访问,user不能访问

-一个中断可连接到另外一个中断

-在中断ISR的末尾要清中断

上面部分是内部中断控制器,如果中断还涉及到外部通过GPIO进入的，需要考虑GPIO作为子中断，自中断的流程和上面差不多

-需要配置引脚成中断模式,这个需要在GPIO那里的寄存器设置

-中断的触发方式,电平还是边沿,这个需要在GPIO那里的寄存器设置

-有MASK,有PENDING寄存器

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

210中断故障分析

通过反汇编发现堆栈内容有问题，看栈初始化是否有问题

栈的初始化没有问题，但是栈是放在DDR2上的，如何检查DDR2初始化有没问题

DDR2初始化和uboot对比发现有地方不对。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NandFlash工作原理

物理结构区别来分类:

-SLC, single level cell, 单层式存储

-MLC, multi level cell, 多层式存储

SLC，一个存储格只存储一位数据

MLC, 一个存储格存放两位数据

MLC对比SLC

价格

-同容量的MLC价格要远低于SLC

访问速度

-SLC的访问速度一般要比MLC快3倍以上

使用寿命

-SLC能进行10万次的擦写，MLC能进行1万次

功耗

-MLC功耗比SLC高15%左右

\*访问方式

-编址方式

-地址结构

-信号引脚

编址方式

-内存统一编址，有地址线和数据线

-NandFlash独立编址，不在地址空间内。从0地址开始

CPU--NandFlash控制器--NandFlash

-传地址，NandFlash控制器有对应寄存器提供给CPU

-传命令，读写擦，NandFlash控制器有对应寄存器提供给CPU

-数据传输，NandFlash控制器有对应寄存器提供给CPU

地址结构

-k9f2g08x0a

-1 Page = (2K +64)Bytes, 2k存的是数据, 64字节存的是一些校验信息

1 block = 64 Pages

1 Device = 2048 Blocks

1st column addr

2nd column addr

3rd row addr

4th row addr

5th row addr

行地址: 其实就是page号

列地址: 一个page中的第几个字节

信号引脚:

CLE, Command Latch Enable, 命令锁存允许

ALE, Address Latch Enable, 地址锁存允许

CE, 芯片选择

RE, 读允许

WE, 写允许

WP, 在写或擦除期间，提供写保护

R/B, 读/忙

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

读取

-页读，读一个页的2K数据过来，提供行地址(页地址)

-随机读，2K数据中的某个偏移的数据，提供行地址和列地址

今天讲解页读

从nandflash芯片手册中，搜索operation, 找到read的时序图，

然后根据时序图来设置读的工作流程。关注IOpin脚的信号

void NF\_read(unsigned long addr, unsigned char \*buff)

{

///选中nandflash芯片

///清除R/#B信号

//发送命令0x00

//发送列地址

//发送行地址

//发送命令0x30

//等待R/#B信号变高，空闲

//从IO pin脚读取数据

///取消选中nandflash

}

使用之前，还要初始化nandflash

void nand\_init()

{

//初始化 NFCONF, 在nandflash芯片手册上指定了一些timing参数

//TACLS, TWRPH0，TWRPH1

//初始化 NFCONT

//复位

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

写

-按页写

-随机写

参照写时序图，看看要完成那些流程

任何写flash操作，都要先擦除，擦除是以block为单位的, 发送的页地址所在的block会被擦除

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口工作常识

-串口角色

-串口通信参数

-引脚连接

串口角色

-数据传输

-充当控制台

波特率: 1秒传输的bit数

串口通信参数

-起始位

-数据位

-奇偶校验位

-停止位

引脚连接

-我们通常使用RS232的9针串口，其中最重要的是2,3,5脚

-2,RXD, 接收数据

-3,TXD, 发送数据

-5,GND, 接地

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口驱动程序设计

-串口初始化

-数据发送

-数据接收

smart210\A\No OS(), 有很多裸机代码

串口初始化

-引脚设置，把引脚设置成串口

-帧格式设置, 数据位，奇偶校验位，停止位，fifo，流控。line control register,ULCON0

-工作模式设置, DMA，中断，轮训模式. control register, UCON0

-波特率设置, 根据串口时钟和要设置成的波特率值，计算出div值, UBRDIV0=(PCLK/BAUD\*16 - 1)

--整数填充到UBRDIV0中，小数部分填充到另外的寄存器UDIVSLOT0

数据发送

-检查发送buf是否空

-填充UTXH0即可

数据接收

-检查接收buf是否有数据

-读取URXH0即可

在PC的串口端回显数据

-在读取URXH0数据后，把数据填充给UTXH0发送给PC端串口进行回显示

0x0d和0x0a是回车换行。

刚开始串口没能正常工作，是因为PLL时钟配置错了，CPU仍然工作在晶振频率

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口控制台建立

-搭建控制台框架

-printf/scanf的实现

-程序结构优化

控制台类型

-菜单式的

-解析型的, 可以解析命令

printf实现

-将变参转换为字符串

-打印字符串到串口

unsigned char outbuf[1024];

int printf(const char \* fmt, ...)

{

va\_list args;

va\_start(args,fmt);//获得变参并放到args

vsprintf((char \*)outbuf, fmt, args);//将变参合fmt字符串打印到outbuf

va\_end();//结束一次变参转换

for (i = 0; i < strlen((char \*)outbuf); i++)

putc(outbuf[i]);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

210 的全局变量有问题

数据段的位置或者内容有问题

通过查bin文件，可以查看到全局变量的值

原来是bin文件加头之后，head\_bin文件被裁减了, 限制了16K、

把16K加大到32K，那么down到板子的时候，说checksum错误

原来BL0计算checksum的时候是以16K计算的，

那加头的程序也在计算checksum的时候是以16K计算，然后就可以了。

遇到问题的时候要用排除法。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DMA原理解析

RAM--CPU--外设

RAM--DMA--外设，这样就可以释放CPU

DMA控制器

-通道数

-请求源

--不同的外设对应不同的DMA通道

-基本时序

-工作模式

--Demand模式，请求模式，只要request有效，则触发新一次传输

--Handshake模式，握手模式，需要等待request变成无效，再变成有效才触发新一次传输

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DMA初始化

-源地址，处于哪条总线，是否递增

-目的地址，处于哪条总线，是否递增，是否产生中断

-DMA控制寄存器，比如设置是否中断通知，传送长度

DMA触发发送

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LCD工作原理

-LCD硬件体系结构

-LCD显示与信号

--LCD显示流程

--LCD时序信号

LCD控制器(ARM芯片内)--LCD驱动芯片--LCD液晶屏

LCD驱动芯片：给液晶屏的液晶分子提供偏转电压

VCLK: 像素点间

HSYNC：行间

VSYNC：帧间

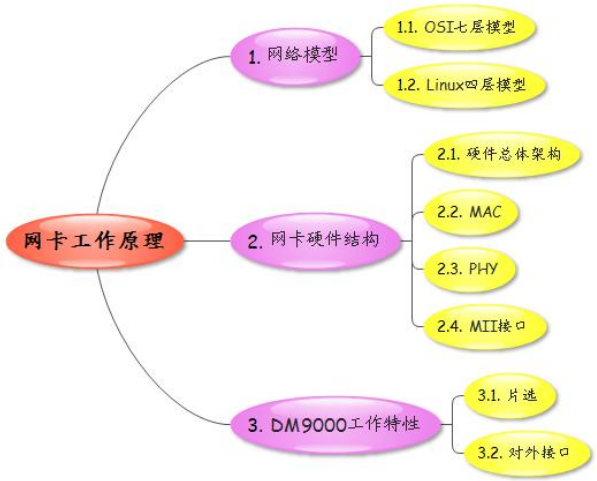
VD[23：0]:LCD像素数据输出端口

VDEN: 数据使能信号

LEND: 行结束信号

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

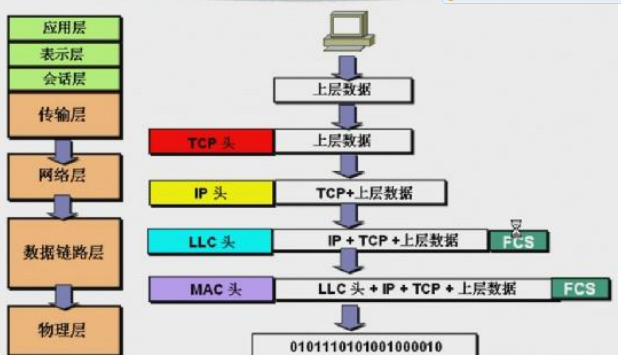
网卡工作原理

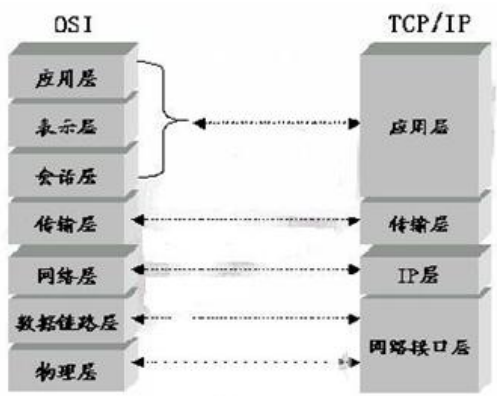


\*网卡模型

-OSI七层模型

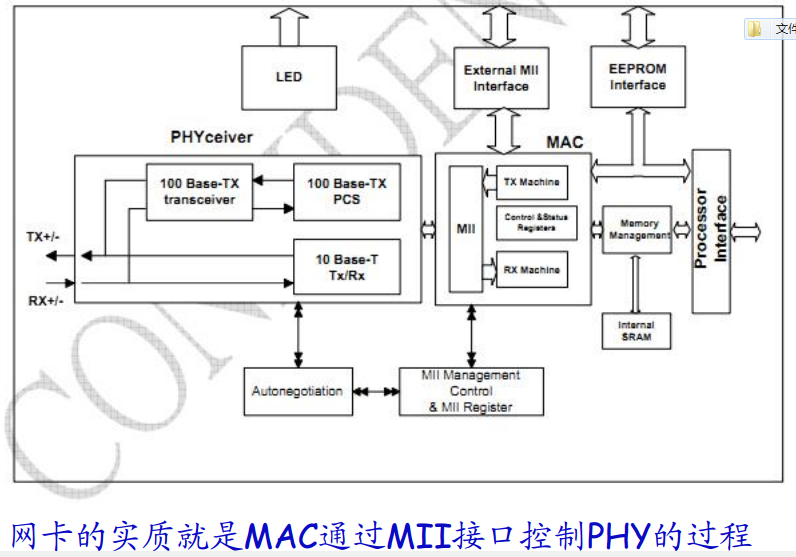
-Linux四层模型，应用层，传输层，IP层，网络接口层(物理层和数据链路层)

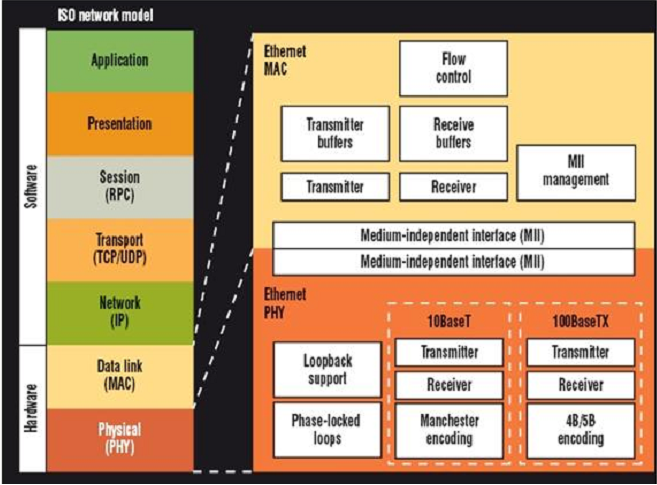




网卡硬件结构

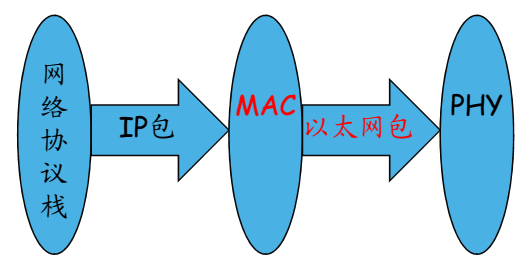
-硬件总体框架



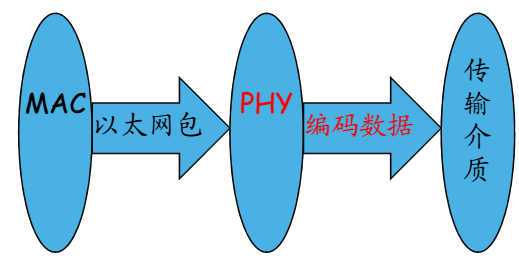


-MAC，属于数据链路层, MAC包提取IP分片提交给CPU，或把CPU传送过来的IP分片组成MAC包发送给PHY

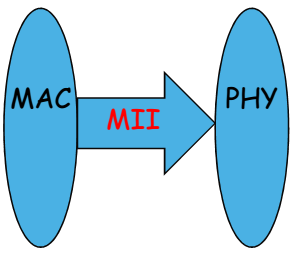
MAC主要负责数据帧的构建，数据差错检查，传送控制等。



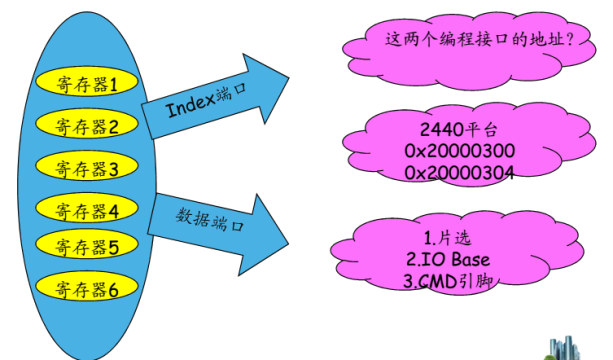
-PHY，是物理接口收发器，属于物理层,当它收到MAC过来的数据时，它会去加上校验码，然后按照物理层的规则进行数据编码，1bit1bit的再发送到传输介质（网线，光纤）上。接收过程则相反。



-MII接口, 媒体（介质）独立接口，是一个规范，MAC->MII->PHY，如果你换了一个PHY，对MAC没有任何影响，因为都遵循MII规范



DM9000工作特性，寄存器很多，不过就两个编程接口，共用数据线



-index端口, 选哪个寄存器，地址0x20000300

-数据端口, 寄存器读写, 地址0x20000304

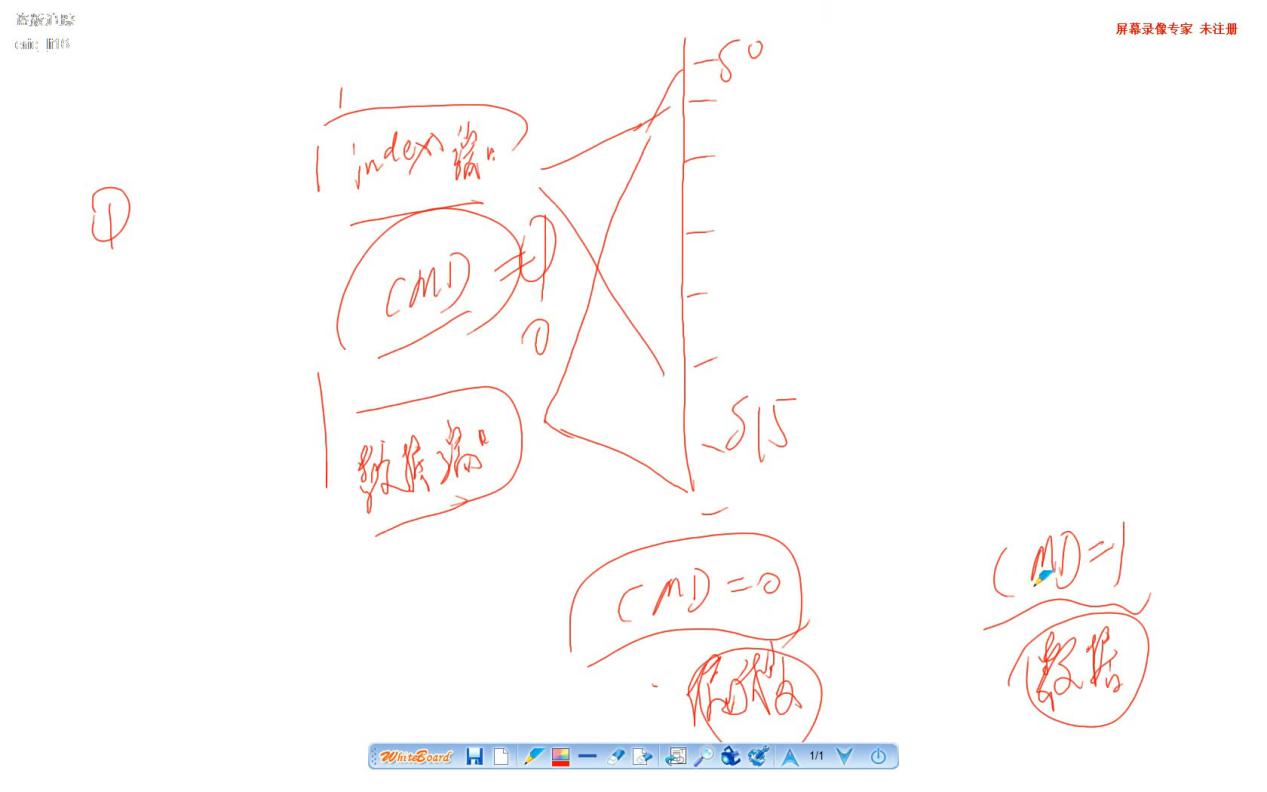
地址信息里面包含:(地址由原理图连接线决定)

-片选信号（AEN），0x20000000， 在2440 芯片手册和原理图中看到

-IO Base, 0x00000300， 在DM9000芯片手册中定义

-CMD引脚，0x00000004，为0则S0~S15为index端口使用，置高则S0~S15为数据端口所使用，这引脚是接到地址bit2的

-S0~S15：传输数据， 和CPU的DTAT0~DATA15数据总线相连



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DM9000驱动程序设计

这里采用的方法是移:基于已有的一套东西，分析修改挪到另外一个平台

从uboot中的dm9000.c 移植到 gboot dm9000.c

从0开始写代码的时间，会很少，大部分时候要做的是移植

我也做过comic驱动和bilinx协议栈的移植

\*参考那一份代码，uboot中的DM9000

\*分析，初始化，发送，接收

\*修改

设置时序的办法

-打开dm9000芯片手册找到相应的时序timing，然后计算填写寄存器值

-在uboot里面找参考

下面2440进行dm9000的初始化:

\*设置片选

-设置BANK4(BWSCON)数据宽度为16

-设置BANK4(BANKCON4)各种timing

--方法一，从dm9000芯片手册里面获得

--方法二，从uboot里面查看，在 \arch\arm\cpu\arm920t\s3c2440\lowlevel\_init.s里面定义

\*中断初始化

-把GPF7设置成EINT7中断

-设置成高电平触发

-由于这个是sub中断(外部中断)，所以要设置EINTMASK和INTMASK两个寄存器

-把SRCPND和INTPND相应位清掉

\*复位设备

\*捕获dm9000

\*MAC初始化

\*填充MAC地址

\*激活dm9000

发送函数

-禁止中断，避免发送过程中受到rx干扰

-写入待发送数据长度

-写入待发送的数据

-启动发送

-等待发送结束

-清除发送状态

-恢复中断使能，恢复接收中断

接收函数(其实是一个ISR)

-判断是否确实产生接收中断,并清除

-空读

-读取接收状态

-读取包长度

-读取包数据

参考代码 专题8-网卡搭建新通道\第2课-DM9000驱动程序设计\范例代码

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARP协议实现

ARP包的内容有本主机的MAC和IP地址，目的MAC和IP地址

板子发送以太网广播的ARP包给PC

PC会ARP包

在网络中，认为所有主机都是大端的，所以包要扔到网络上就得是大端的

TCP协议

-三次握手

-每个包有一个序号

-如果发送端在RTT时间内没有收到确认，那么对应的数据包就假设丢失并将被重传

-有校验和，发送和接收端都有校验和函数

-数据包的MSS长度通过计算数据链路层的MTU获得

-在可靠性上，有超时重传，捎带确认机制

-在流量控制上，采用滑动窗口协议

-在拥塞控制算法上，控制窗口结合控制算法实现