# uboot代码分析

## uboot常用机制

### 1.1配置文件

include/configs/<board\_name>.h，开发板的配置文件，修改该文件可以对uboot进行裁剪。

### 1.2nandflash使用分布

#define MTDIDS\_DEFAULT "nand0=nandflash0"

#define MTDPARTS\_DEFAULT "mtdparts=nandflash0:256k@0(bootloader)," \

"128k(params)," \

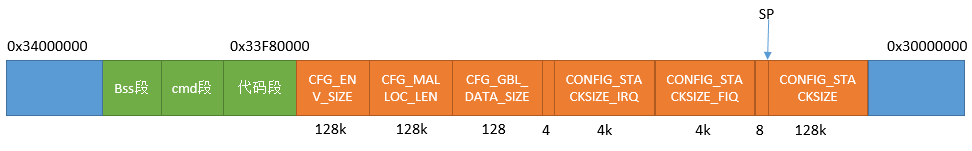
"2m(kernel)," \

"-(root)"

从配置文件中可以获取到nandflash的使用情况，其中128k的params用于存储uboot的环境变量。

### 1.3内存使用分布

JZ2440中内存大小为64MB，地址范围为0x3000000~0x34000000。



从代码段往后的内存块用途分别为：存储环境变量，MALLOC内存段，全局变量数据段，空余的4字节，IRQ堆栈，FIQ堆栈，空余的8字节，正常模式堆栈。

### 1.4环境变量工作原理

u-boot的环境变量用来存储一些经常使用的参数变量，uboot希望将环境变量存储在静态存储器中（如nand nor eeprom mmc）。

常用的环境变量如下：

bootdelay 执行自动启动的等候秒数

baudrate 串口控制台的波特率

netmask 以太网接口的掩码

ethaddr 以太网卡的网卡物理地址

bootfile 缺省的下载文件

bootargs 传递给内核的启动参数

bootcmd 自动启动时执行的命令

serverip 服务器端的ip地址

ipaddr 本地ip 地址

stdin 标准输入设备

stdout 标准输出设备

stderr 标准出错设备

uboot环境变量的设计逻辑是在启动过程中将env从静态存储器中读出放到RAM中，之后在uboot下对env的操作（如printenv editenv setenv）都是对RAM中env的操作，只有在执行saveenv时才会将RAM中的env重新写入静态存储器中。

#### 1.4.1 env数据结构体

**typedef** struct environment\_s **{**

unsigned long crc**;** /\* CRC32 over data bytes \*/

#ifdef CFG\_REDUNDAND\_ENVIRONMENT

unsigned char flags**;** /\* active/obsolete flags \*/

#endif

unsigned char data**[**ENV\_SIZE**];** /\* Environment data \*/

**}** env\_t**;**

// CONFIG\_ENV\_SIZE是我们需要在配置文件中配置的环境变量的总长度。

// 这里我们使用的nand作为静态存储器，nand的一个block是128K，因此选用一个block来存储env，CONFIG\_ENV\_SIZE为128K。

// Env\_t结构体头4个bytes是对data的crc校验码，没有定义CONFIG\_SYS\_REDUNDAND\_ENVIRONMENT，所以后面紧跟data数组，数组大小是ENV\_SIZE.

// ENV\_SIZE是CONFIG\_ENV\_SIZE减掉ENV\_HEADER\_SIZE，也就是4bytes

// 所以env\_t这个结构体就包含了整个我们规定的长度为CONFIG\_ENV\_SIZE的存储区域。

// 头4bytes是crc校验码，后面剩余的空间全部用来存储环境变量。

// 需要说明的一点，crc校验码是uboot中在saveenv时计算出来，然后写入nand，所以在第一次启动uboot时crc校验会出错，

// 因为uboot从nand上读入的一个block数据是随机的，没有意义的，执行saveenv后重启uboot，crc校验就正确了。

// data字段保存实际的环境变量。u-boot的env按 name=value”\0”的方式存储，在所有env的最后以”\0\0”表示整个 env的结束。

// 新的name=value对总是被添加到env数据块的末尾，当删除一个 name=value 对时，后面的环境变量将前移，对一个已经存在的环境变量的修改实际上先删除再插入。

u-boot 把env\_t的数据指针还保存在了另外一个地方,这就 是 gd\_t结构（不同平台有不同的 gd\_t 结构 ）,这里以ARM为例仅列出和 env 相关的部分 。

**typedef** struct global\_data

**{**

…

unsigned long env\_off**;** /\* Relocation Offset \*/

unsigned long env\_addr**;** /\* Address of Environment struct ??? \*/

unsigned long env\_valid /\* Checksum of Environment valid \*/

…

**}** gd\_t**;**

#### 1.4.2 env的初始化和迁移

首先在board\_init\_f中调用init\_sequence的env\_init，这个函数是不同存储器实现的函数，nand中的实现如下：

int env\_init**(**void**)**

**{**

gd**->**env\_addr **=** **(**ulong**)&**default\_environment**[**0**];**

gd**->**env\_valid **=** 1**;**

**return** 0**;**

**}**

//env\_init要早于静态存储器的初始化，所以无法进行env的读写，所以在这里将env设置为默认的环境变量

完成nandflash的初始化后，需要进行环境变量的搬移，从nandflash中读出存储的环境变量并搬移到内存中。

void env\_relocate **(**void**)**

**{**

env\_ptr **=** **(**env\_t **\*)**malloc **(**CFG\_ENV\_SIZE**);**

//从malloc段中分配128k的空间用于存储环境变量

env\_get\_char **=** env\_get\_char\_memory**;**

//获取环境变量内容的函数更换为env\_get\_char\_memory

env\_relocate\_spec **();**

//将存储在nandflash中环境变量拷贝到内存中

gd**->**env\_addr **=** **(**ulong**)&(**env\_ptr**->**data**);**

//重新设置gd全局变量，更改环境变量的内容指针从default\_environment到env\_ptr->data

**}**

uchar env\_get\_char\_memory **(**int index**)**

**{**

**if** **(**gd**->**env\_valid**)** **{**

**return** **(** **\*((**uchar **\*)(**gd**->**env\_addr **+** index**))** **);**

**}** **else** **{**

**return** **(** default\_environment**[**index**]** **);**

**}**

**}**

void env\_relocate\_spec **(**void**)**

**{**

#if !defined(ENV\_IS\_EMBEDDED)

ulong total **=** CFG\_ENV\_SIZE**;**

int ret**;**

ret **=** nand\_read**(&**nand\_info**[**0**],** CFG\_ENV\_OFFSET**,** **&**total**,** **(**u\_char**\*)**env\_ptr**);**

**if** **(**ret **||** total **!=** CFG\_ENV\_SIZE**)**

**return** use\_default**();**

//如果出现异常将default\_environment的内容读到内存中

**if** **(**crc32**(**0**,** env\_ptr**->**data**,** ENV\_SIZE**)** **!=** env\_ptr**->**crc**)**

**return** use\_default**();**

#endif /\* ! ENV\_IS\_EMBEDDED \*/

#### 1.4.3 printenv操作

int do\_printenv **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

int i**,** j**,** k**,** nxt**;**

int rcode **=** 0**;**

**if** **(**argc **==** 1**)** **{** /\* Print all env variables \*/

**for** **(**i**=**0**;** env\_get\_char**(**i**)** **!=** '\0'**;** i**=**nxt**+**1**)** **{**

**for** **(**nxt**=**i**;** env\_get\_char**(**nxt**)** **!=** '\0'**;** **++**nxt**)**

**;**

**for** **(**k**=**i**;** k**<**nxt**;** **++**k**)**

putc**(**env\_get\_char**(**k**));**

putc **(**'\n'**);**

**if** **(**ctrlc**())** **{**

puts **(**"\n \*\* Abort\n"**);**

**return** 1**;**

**}**

**}**

printf**(**"\nEnvironment size: %d/%d bytes\n"**,** i**,** ENV\_SIZE**);**

**return** 0**;**

**}**

//当printenv后不带任何参数的时候，会打印出所有的环境变量

**for** **(**i**=**1**;** i**<**argc**;** **++**i**)** **{** /\* print single env variables \*/

char **\***name **=** argv**[**i**];**

k **=** **-**1**;**

**for** **(**j**=**0**;** env\_get\_char**(**j**)** **!=** '\0'**;** j**=**nxt**+**1**)** **{**

**for** **(**nxt**=**j**;** env\_get\_char**(**nxt**)** **!=** '\0'**;** **++**nxt**)**

**;**

k **=** envmatch**((**uchar **\*)**name**,** j**);**

**if** **(**k **<** 0**)** **{**

**continue;**

**}**

puts **(**name**);**

putc **(**'='**);**

**while** **(**k **<** nxt**)**

putc**(**env\_get\_char**(**k**++));**

putc **(**'\n'**);**

**break;**

**}**

**if** **(**k **<** 0**)** **{**

printf **(**"## Error: \"%s\" not defined\n"**,** name**);**

rcode **++;**

**}**

**}**

**return** rcode**;**

//当printenv带参数的时，会将参数的内容打印出来

**}**

#### 1.4.4 setenv操作

setenv的操作步骤大致为，由于该部分代码较多，这里只截出一部分代码。

int \_do\_setenv **(**int flag**,** int argc**,** char **\***argv**[])**

**{**

//1.查找该环境变量参数名是否存在

oldval **=** **-**1**;**

**for** **(**env**=**env\_data**;** **\***env**;** env**=**nxt**+**1**)** **{**

**for** **(**nxt**=**env**;** **\***nxt**;** **++**nxt**)**

**;**

**if** **((**oldval **=** envmatch**((**uchar **\*)**name**,** env**-**env\_data**))** **>=** 0**)**

**break;**

**}**

//2.假如该环境变量参数存在，删除现有的内容并修改相关的全局变量

**if** **(**oldval **>=** 0**)** **{**

//如果修改的参数和console相关，修改完后需要重新assign devcie name

**if** **(**strcmp**(**name**,**"stdin"**)** **==** 0**)** **{**

console **=** stdin**;**

**}** **else** **if** **(**strcmp**(**name**,**"stdout"**)** **==** 0**)** **{**

console **=** stdout**;**

**}** **else** **if** **(**strcmp**(**name**,**"stderr"**)** **==** 0**)** **{**

console **=** stderr**;**

**}**

**if** **(**console **!=** **-**1**)** **{**

**if** **(**argc **<** 3**)** **{** /\* Cannot delete it! \*/

printf**(**"Can't delete \"%s\"\n"**,** name**);**

**return** 1**;**

**}**

/\* Try assigning specified device \*/

**if** **(**console\_assign **(**console**,** argv**[**2**])** **<** 0**)**

**return** 1**;**

**}**

//如果修改的参数和波特率有关，需要重新设置全局变量，并重新初始化串口

**if** **(**strcmp**(**argv**[**1**],**"baudrate"**)** **==** 0**)** **{**

int baudrate **=** simple\_strtoul**(**argv**[**2**],** **NULL,** 10**);**

int i**;**

**for** **(**i**=**0**;** i**<**N\_BAUDRATES**;** **++**i**)** **{**

**if** **(**baudrate **==** baudrate\_table**[**i**])**

**break;**

**}**

**if** **(**i **==** N\_BAUDRATES**)** **{**

printf **(**"## Baudrate %d bps not supported\n"**,**

baudrate**);**

**return** 1**;**

**}**

printf **(**"## Switch baudrate to %d bps and press ENTER ...\n"**,**

baudrate**);**

udelay**(**50000**);**

gd**->**baudrate **=** baudrate**;**

serial\_setbrg **();**

udelay**(**50000**);**

**for** **(;;)** **{**

**if** **(**getc**()** **==** '\r'**)**

**break;**

**}**

**}**

//将后面的内容往前搬移，覆盖掉原来的环境变量

**if** **(\*++**nxt **==** '\0'**)** **{** //如果是尾部最后一条，则需将env置\0

**if** **(**env **>** env\_data**)** **{**

env**--;**

**}** **else** **{**

**\***env **=** '\0'**;**

**}**

**}** **else** **{**

**for** **(;;)** **{**

**\***env **=** **\***nxt**++;**

**if** **((\***env **==** '\0'**)** **&&** **(\***nxt **==** '\0'**))** //连续遇到两个\0\0,到达尾部

**break;**

**++**env**;**

**}**

**}**

**\*++**env **=** '\0'**;**

**}**

//3.如果setenv命令操作没有带属性参数，则直接更新crc

**if** **((**argc **<** 3**)** **||** argv**[**2**]** **==** **NULL)** **{**

env\_crc\_update **();**

**return** 0**;**

**}**

//4.将新的环境变量插入到尾部

**for** **(**env**=**env\_data**;** **\***env **||** **\*(**env**+**1**);** **++**env**)** //当连续出现两个\0\0则到达尾部

**;**

**if** **(**env **>** env\_data**)**

**++**env**;**

//需要插入的总长度为环境变量长度+两个'\0'+'='+参数长度

len **=** strlen**(**name**)** **+** 2**;**

**for** **(**i**=**2**;** i**<**argc**;** **++**i**)** **{**

len **+=** strlen**(**argv**[**i**])** **+** 1**;**

**}**

//判断是否超过环境变量内存块

**if** **(**len **>** **(&**env\_data**[**ENV\_SIZE**]-**env**))** **{**

printf **(**"## Error: environment overflow, \"%s\" deleted\n"**,** name**);**

**return** 1**;**

**}**

//插入name

**while** **((\***env **=** **\***name**++)** **!=** '\0'**)**

env**++;**

//插入=和val

**for** **(**i**=**2**;** i**<**argc**;** **++**i**)** **{**

char **\***val **=** argv**[**i**];**

**\***env **=** **(**i**==**2**)** **?** '=' **:** ' '**;**

**while** **((\*++**env **=** **\***val**++)** **!=** '\0'**)**

**;**

**}**

/\* end is marked with double '\0' \*/

**\*++**env **=** '\0'**;**

/\* Update CRC \*/

env\_crc\_update **();**

**}**

#### 1.4.5 savenv操作

int saveenv**(**void**)**

**{**

//擦除nandflash存储环境变量的128k空间

ulong total**;**

int ret **=** 0**;**

puts **(**"Erasing Nand..."**);**

**if** **(**nand\_erase**(&**nand\_info**[**0**],** CFG\_ENV\_OFFSET**,** CFG\_ENV\_SIZE**))**

**return** 1**;**

//重新将内存的环境变量写入到nandflash中

puts **(**"Writing to Nand... "**);**

total **=** CFG\_ENV\_SIZE**;**

ret **=** nand\_write**(&**nand\_info**[**0**],** CFG\_ENV\_OFFSET**,** **&**total**,** **(**u\_char**\*)**env\_ptr**);**

**if** **(**ret **||** total **!=** CFG\_ENV\_SIZE**)**

**return** 1**;**

puts **(**"done\n"**);**

**return** ret**;**

**}**

### 1.5命令机制工作原理

#### 1.5.1 U\_BOOT\_CMD宏

在UBOOT当中，各个命令是通过U\_BOOT\_CMD这个宏来定义出来的。

#define Struct\_Section \_\_attribute\_\_ ((unused,section (".u\_boot\_cmd")))

//由此可见，被U\_BOOTD\_CMD定义过的结构体，最终回被放到一个u\_boot\_cmd段中。

#define U\_BOOT\_CMD(name,maxargs,rep,cmd,usage,help) \

cmd\_tbl\_t \_\_u\_boot\_cmd\_##name Struct\_Section = {#name, maxargs, rep, cmd, usage, help}

//这样一来，凡通过U\_BOOT\_CMD定义的cmd\_tbl\_t变量会全部被放在.u\_boot\_cmd段当中。

//例如：定义一个命令boot U\_BOOT\_CMD(boot, 0, 0, fun, "boot xxx"); 展开以后会变成：

//cmd\_tbl\_t \_\_u\_boot\_cmd\_boot \_\_attribute\_\_\_((unused, section(".u\_boot\_cmd"))) = {"boot", 0, 0, fun, "boot xxx"}

添加一个新命令的方法，先自定义命令执行函数。然后再调用U\_BOOT\_CMD添加

int do\_xxx **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[** **])**

**{**

……

……

**}**

#### 1.5.2 命令执行的流程

进入到uboot的命令模式后，在终端敲入命令，会执行run\_command函数去执行对应的指令。

int run\_command **(**const char **\***cmd**,** int flag**)**

**{**

cmd\_tbl\_t **\***cmdtp**;**

char cmdbuf**[**CFG\_CBSIZE**];** /\* working copy of cmd \*/

char **\***token**;** /\* start of token in cmdbuf \*/

char **\***sep**;** /\* end of token (separator) in cmdbuf \*/

char finaltoken**[**CFG\_CBSIZE**];**

char **\***str **=** cmdbuf**;**

char **\***argv**[**CFG\_MAXARGS **+** 1**];** /\* NULL terminated \*/

int argc**,** inquotes**;**

int repeatable **=** 1**;**

int rc **=** 0**;**

clear\_ctrlc**();** /\* forget any previous Control C \*/

//先是对命令的有效性进行检测

**if** **(!**cmd **||** **!\***cmd**)** **{**

**return** **-**1**;** /\* empty command \*/

**}**

**if** **(**strlen**(**cmd**)** **>=** CFG\_CBSIZE**)** **{**

puts **(**"## Command too long!\n"**);**

**return** **-**1**;**

**}**

//备份command

strcpy **(**cmdbuf**,** cmd**);**

/\* Process separators and check for invalid

\* repeatable commands

\*/

**while** **(\***str**)** **{**

//寻找分隔符和命令尾部，相邻的句子是用;分隔的。每次处理一句命令

**for** **(**inquotes **=** 0**,** sep **=** str**;** **\***sep**;** sep**++)** **{**

**if** **((\***sep**==**'\''**)** **&&**

**(\*(**sep**-**1**)** **!=** '\\'**))**

inquotes**=!**inquotes**;**

**if** **(!**inquotes **&&**

**(\***sep **==** ';'**)** **&&** /\* separator \*/

**(** sep **!=** str**)** **&&** /\* past string start \*/

**(\*(**sep**-**1**)** **!=** '\\'**))** /\* and NOT escaped \*/

**break;**

**}**

token **=** str**;** //token指向命令的开头

**if** **(\***sep**)** **{**

str **=** sep **+** 1**;** //str指向下一句开头

**\***sep **=** '\0'**;** //如果是分隔符的话，将分隔符替换为空字符

**}**

**else**

str **=** sep**;** //没有其他命令了，str指向命令尾部

//把token指向的命令中的宏替换掉，如把$(kernelsize)替换成内核的大小

process\_macros **(**token**,** finaltoken**);**

//把每个参数用'/0'隔开，argv中的每一个指针指向一个参数的起始地址，返回值为参数的个数

**if** **((**argc **=** parse\_line **(**finaltoken**,** argv**))** **==** 0**)** **{**

rc **=** **-**1**;** /\* no command at all \*/

**continue;**

**}**

//第一个参数就是要运行的命令，首先在命令表中找到它的命令结构体的指针

**if** **((**cmdtp **=** find\_cmd**(**argv**[**0**]))** **==** **NULL)** **{**

printf **(**"Unknown command '%s' - try 'help'\n"**,** argv**[**0**]);**

rc **=** **-**1**;** /\* give up after bad command \*/

**continue;**

**}**

//检查参数个数是否过多

**if** **(**argc **>** cmdtp**->**maxargs**)** **{**

printf **(**"Usage:\n%s\n"**,** cmdtp**->**usage**);**

rc **=** **-**1**;**

**continue;**

**}**

//调用命令执行函数

**if** **((**cmdtp**->**cmd**)** **(**cmdtp**,** flag**,** argc**,** argv**)** **!=** 0**)** **{**

rc **=** **-**1**;**

**}**

repeatable **&=** cmdtp**->**repeatable**;**

//置命令自动重复执行标志，也就是按下enter键是否可以执行最近执行的命令

//检查是否有ctrl+c按键按下，如果有，则不允许进行repeatable。

//本函数并没有从中断接收字符，接收ctrl+c的是一些执行命令的函数

**if** **(**had\_ctrlc **())**

**return** 0**;** /\* if stopped then not repeatable \*/

**}**

**return** rc **?** rc **:** repeatable**;**

**}**

### 1.6 uboot和内核交互机制

uboot和内核的交互是单向的，uboot将各类参数传递给内核。uboot将参数放在某个约定的地方后，再启动内核，内核启动后从这个地方获得参数。

除了约定好参数存放的地址外，还要规定参数的结构,该数据结构为tag。

//tag的类型和长度，是表示内存还是表示命令行参数等。

struct tag\_header **{**

u32 size**;**

u32 tag**;**

**};**

struct tag **{**

struct tag\_header hdr**;**

//不同的标记有不同结构体表示

union **{**

struct tag\_core core**;**

struct tag\_mem32 mem**;**

struct tag\_videotext videotext**;**

struct tag\_ramdisk ramdisk**;**

struct tag\_initrd initrd**;**

struct tag\_serialnr serialnr**;**

struct tag\_revision revision**;**

struct tag\_videolfb videolfb**;**

struct tag\_cmdline cmdline**;**

/\*

\* Acorn specific

\*/

struct tag\_acorn acorn**;**

/\*

\* DC21285 specific

\*/

struct tag\_memclk memclk**;**

**}** u**;**

**};**

#### 1.6.1 ATAG\_CORE标记

标记列表以ATAG\_CORE开始，uboot和内核约定的参数存放在地址0x30000100处。

static void setup\_start\_tag **(**bd\_t **\***bd**)**

**{**

params **=** **(**struct tag **\*)** bd**->**bi\_boot\_params**;**

params**->**hdr**.**tag **=** ATAG\_CORE**;**

params**->**hdr**.**size **=** tag\_size **(**tag\_core**);**

params**->**u**.**core**.**flags **=** 0**;**

params**->**u**.**core**.**pagesize **=** 0**;**

params**->**u**.**core**.**rootdev **=** 0**;**

params **=** tag\_next **(**params**);**

**}**

其中tag\_next指向当前标记的末尾。

#define tag\_next(t) ((struct tag \*)((u32 \*)(t) + (t)->hdr.size))

#### 1.6.2 内存标记

设置开发板使用的内存起始地址和大小，这里分别为0x30000000和0x4000000。

#define PHYS\_SDRAM\_1 0x30000000 /\* SDRAM Bank #1 \*/

#define PHYS\_SDRAM\_1\_SIZE 0x04000000 /\* 64 MB \*/

static void setup\_memory\_tags **(**bd\_t **\***bd**)**

**{**

int i**;**

**for** **(**i **=** 0**;** i **<** CONFIG\_NR\_DRAM\_BANKS**;** i**++)** **{**

params**->**hdr**.**tag **=** ATAG\_MEM**;**

params**->**hdr**.**size **=** tag\_size **(**tag\_mem32**);**

params**->**u**.**mem**.**start **=** bd**->**bi\_dram**[**i**].**start**;**

params**->**u**.**mem**.**size **=** bd**->**bi\_dram**[**i**].**size**;**

params **=** tag\_next **(**params**);**

**}**

**}**

#### 1.6.3 命令行标记

命令行是一个字符串，用于控制内核的一些行为。

#define CONFIG\_BOOTARGS "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0"

//root=/dev/mtdblock3 根文件系统挂在MTD3分区上

//init=/linuxrc 系统启动后执行的第一个程序为/linuxrc

//console=ttySAC0" 控制台为ttySAC0第一个串口

static void setup\_commandline\_tag **(**bd\_t **\***bd**,** char **\***commandline**)**

**{**

char **\***p**;**

**if** **(!**commandline**)**

**return;**

/\* eat leading white space \*/

**for** **(**p **=** commandline**;** **\***p **==** ' '**;** p**++);**

/\* skip non-existent command lines so the kernel will still

\* use its default command line.

\*/

**if** **(\***p **==** '\0'**)**

**return;**

params**->**hdr**.**tag **=** ATAG\_CMDLINE**;**

params**->**hdr**.**size **=**

**(sizeof** **(**struct tag\_header**)** **+** strlen **(**p**)** **+** 1 **+** 4**)** **>>** 2**;**

strcpy **(**params**->**u**.**cmdline**.**cmdline**,** p**);**

params **=** tag\_next **(**params**);**

**}**

#### 1.6.4 ATAG\_NONE标记

标记列表以ATAG\_NONE结束。

static void setup\_end\_tag **(**bd\_t **\***bd**)**

**{**

params**->**hdr**.**tag **=** ATAG\_NONE**;**

params**->**hdr**.**size **=** 0**;**

**}**

### 1.7 mkimage命令

uboot源代码的tools/目录下有mkimage工具，这个工具可以用来制作不压缩或者压缩的多种可启动映象文件。

mkimage在制作映象文件的时候，是在原来的可执行映象文件的前面加上一个0x40字节的头，记录参数所指定的信息，这样uboot才能识别这个映象是针对哪个CPU体系结构的，哪个OS的，哪种类型，加载内存中的哪个位置，入口点在内存的那个位置以及映象名是什么。

arm**-**linux**-**objcopy **-**O binary **-**R **.**note **-**R **.**comment **-**S vmlinux linux**.**bin

gzip **-**9 linux**.**bin

mkimage **-**A arm **-**O linux **-**T kernel **-**C gzip **-**a 0x30008000 **-**e 0x30008000 **-**n "Linux Kernel Image" **-**d linux**.**bin**.**gz uImage

//-A 指定CPU的体系结构

//-O 指定操作系统类型

//-C 指定映象压缩方式

//-a 指定映象在内存中的加载地址

//-e 指定映象运行的入口点地址

//-n 指定映象名

//-d 指定制作映象的源文件

mkimage部分代码流程分析如下：

int main **(**int argc**,** char **\*\***argv**)**

**{**

**while** **(--**argc **>** 0 **&&** **\*\*++**argv **==** '-'**)** **{**

**while** **(\*++\***argv**)** **{**

**case** 'A'**:**

**if** **((--**argc **<=** 0**)** **||**

**(**opt\_arch **=** get\_arch**(\*++**argv**))** **<** 0**)**

usage **();**

**goto** NXTARG**;**

//获取CPU体系名称，这里为#define IH\_CPU\_ARM 2 /\* ARM \*/

**case** 'C'**:**

**if** **((--**argc **<=** 0**)** **||**

**(**opt\_comp **=** get\_comp**(\*++**argv**))** **<** 0**)**

usage **();**

**goto** NXTARG**;**

//获取压缩方式#define IH\_COMP\_GZIP 1 /\* gzip Compression Used \*/

**case** 'O'**:**

**if** **((--**argc **<=** 0**)** **||**

**(**opt\_os **=** get\_os**(\*++**argv**))** **<** 0**)**

usage **();**

**goto** NXTARG**;**

//获取操作系统类型#define IH\_OS\_LINUX 5 /\* Linux \*/

**case** 'T'**:**

**if** **((--**argc **<=** 0**)** **||**

**(**opt\_type **=** get\_type**(\*++**argv**))** **<** 0**)**

usage **();**

**goto** NXTARG**;**

//获取image类型#define IH\_TYPE\_KERNEL 2 /\* OS Kernel Image \*/

**case** 'a'**:**

**if** **(--**argc **<=** 0**)**

usage **();**

addr **=** strtoul **(\*++**argv**,** **(**char **\*\*)&**ptr**,** 16**);**

**if** **(\***ptr**)** **{**

fprintf **(**stderr**,**

"%s: invalid load address %s\n"**,**

cmdname**,** **\***argv**);**

exit **(**EXIT\_FAILURE**);**

**}**

**goto** NXTARG**;**

//获取load address地址

**case** 'd'**:**

**if** **(--**argc **<=** 0**)**

usage **();**

datafile **=** **\*++**argv**;**

dflag **=** 1**;**

**goto** NXTARG**;**

//获取需要打包文件的名字

**case** 'e'**:**

**if** **(--**argc **<=** 0**)**

usage **();**

ep **=** strtoul **(\*++**argv**,** **(**char **\*\*)&**ptr**,** 16**);**

**if** **(\***ptr**)** **{**

fprintf **(**stderr**,**

"%s: invalid entry point %s\n"**,**

cmdname**,** **\***argv**);**

exit **(**EXIT\_FAILURE**);**

**}**

eflag **=** 1**;**

**goto** NXTARG**;**

//获取entry point地址

**case** 'n'**:**

**if** **(--**argc **<=** 0**)**

usage **();**

name **=** **\*++**argv**;**

**goto** NXTARG**;**

//获取image名字这里为Linux image kernel

**}**

NXTARG**:** **;**

**}**

imagefile **=** **\***argv**;**

//获取最终打包文件名

ifd **=** open**(**imagefile**,** O\_RDWR**|**O\_CREAT**|**O\_TRUNC**|**O\_BINARY**,** 0666**);**

//创建最终打包文件

memset **(**hdr**,** 0**,** **sizeof(**image\_header\_t**));**

write**(**ifd**,** hdr**,** **sizeof(**image\_header\_t**));**

//清空image头部数据并将head数据写入到最终打包的文件中

//image\_header\_t header;

//image\_header\_t \*hdr = &header;

copy\_file **(**ifd**,** datafile**,** 0**);**

//将需要打包的文件写入到最终打包的文件中

fstat**(**ifd**,** **&**sbuf**);**

//重新获取最终打包文件的文件状态

ptr **=** **(**unsigned char **\*)**mmap**(**0**,** sbuf**.**st\_size**,**

PROT\_READ**|**PROT\_WRITE**,** MAP\_SHARED**,** ifd**,** 0**);**

//将最终打包文件进行内存映射

hdr **=** **(**image\_header\_t **\*)**ptr**;**

//获取头部指针

checksum **=** crc32 **(**0**,**

**(**const char **\*)(**ptr **+** **sizeof(**image\_header\_t**)),**

sbuf**.**st\_size **-** **sizeof(**image\_header\_t**)**

**);**

//计算打包文件crc

hdr**->**ih\_magic **=** htonl**(**IH\_MAGIC**);**

hdr**->**ih\_time **=** htonl**(**sbuf**.**st\_mtime**);**

hdr**->**ih\_size **=** htonl**(**sbuf**.**st\_size **-** **sizeof(**image\_header\_t**));**

hdr**->**ih\_load **=** htonl**(**addr**);**

hdr**->**ih\_ep **=** htonl**(**ep**);**

hdr**->**ih\_dcrc **=** htonl**(**checksum**);**

hdr**->**ih\_os **=** opt\_os**;**

hdr**->**ih\_arch **=** opt\_arch**;**

hdr**->**ih\_type **=** opt\_type**;**

hdr**->**ih\_comp **=** opt\_comp**;**

strncpy**((**char **\*)**hdr**->**ih\_name**,** name**,** IH\_NMLEN**);**

checksum **=** crc32**(**0**,(**const char **\*)**hdr**,sizeof(**image\_header\_t**));**

hdr**->**ih\_hcrc **=** htonl**(**checksum**);**

//重新构建头部数据，头部数据如下：

//typedef struct image\_header {

//uint32\_t ih\_magic; /\* Image Header Magic Number \*/

//uint32\_t ih\_hcrc; /\* Image Header CRC Checksum \*/

//uint32\_t ih\_time; /\* Image Creation Timestamp \*/

//uint32\_t ih\_size; /\* Image Data Size \*/

//uint32\_t ih\_load; /\* Data Load Address \*/

//uint32\_t ih\_ep; /\* Entry Point Address \*/

//uint32\_t ih\_dcrc; /\* Image Data CRC Checksum \*/

//uint8\_t ih\_os; /\* Operating System \*/

//uint8\_t ih\_arch; /\* CPU architecture \*/

//uint8\_t ih\_type; /\* Image Type \*/

//uint8\_t ih\_comp; /\* Compression Type \*/

//uint8\_t ih\_name[IH\_NMLEN]; /\* Image Name \*/

//} image\_header\_t;

print\_header **(**hdr**);**

//打印头部数据信息

**}**

### 1.8 linux系统引导机制bootm

bootm 用于将内核镜像加载到内存的指定地址处，如果有需要还要解压镜像，然后根据操作系统和体系结构的不同给内核传递不同的启动参数，最后启动内核。

#### 1.8.1 arm架构处理器对 linux 内核启动之前环境的五点需求

1、cpu 寄存器设置

**\*** R0 **=** 0

**\*** R1 **=** 板级 id

**\*** R2 **=** 启动参数在内存中的起始地址

2、cpu 模式

**\*** 禁止所有中断

**\*** 必须为SVC（超级用户）模式

3、缓存、MMU

**\*** 关闭 MMU

**\*** 指令缓存可以开启或者关闭

**\*** 数据缓存必须关闭并且不能包含任何脏数据

4、设备

**\*** DMA 设备应当停止工作

5、boot loader 需要跳转到内核镜像的第一条指令处

#### 1.8.2 bootm代码分析

引导linux系统的启动命令如下：

#define CONFIG\_BOOTCOMMAND "nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0"

//先从nandflash中读取kernel分区内容到内存0x30007FC0处，bootm从该内存地址中开始启动。

//（0x40为uImage头部长度，所以bootm地址为0x30008000-0x40=0x30007FC0）。

//嵌入式linux内核在内存中运行地址0x30008000到内存起始运行地址0x30000000中的（0x8000=32k）怎么回事？

//0~0x4000存放Bootloader传递到Kernel的参数，0x4000~0x8000存放主页表的PGD（arm页表项4byte，

//映射1M地址空间，4G空间大小需要4096个页表项，所以16k = 4byte \* 4096）

do\_bootm部分代码分析如下：

int do\_bootm **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

**if** **(**argc **<** 2**)** **{**

addr **=** load\_addr**;**

**}** **else** **{**

addr **=** simple\_strtoul**(**argv**[**1**],** **NULL,** 16**);**

**}**

//这里addr为0x30007FC0，bootm 0x30007FC0指令传进来的参数

memmove **(&**header**,** **(**char **\*)**addr**,** **sizeof(**image\_header\_t**));**

//将0x30007FC0内存中的前0x40字节内容提取到header结构体中

**if** **(**ntohl**(**hdr**->**ih\_magic**)** **!=** IH\_MAGIC**){**

puts **(**"Bad Magic Number\n"**);**

SHOW\_BOOT\_PROGRESS **(-**1**);**

**return** 1**;**

**}**

//判断header中的magic number是否正确

data **=** **(**ulong**)&**header**;**

len **=** **sizeof(**image\_header\_t**);**

//data为header内存地址

//len为header长度

checksum **=** ntohl**(**hdr**->**ih\_hcrc**);**

hdr**->**ih\_hcrc **=** 0**;**

**if** **(**crc32 **(**0**,** **(**uchar **\*)**data**,** len**)** **!=** checksum**)** **{**

puts **(**"Bad Header Checksum\n"**);**

SHOW\_BOOT\_PROGRESS **(-**2**);**

**return** 1**;**

**}**

//计算header checksum并比较是否相同

print\_image\_hdr **((**image\_header\_t **\*)**addr**);**

//打印header相关信息

data **=** addr **+** **sizeof(**image\_header\_t**);**

len **=** ntohl**(**hdr**->**ih\_size**);**

len\_ptr **=** **(**ulong **\*)**data**;**

//data为linux kernel image内存所在地址 这里为0x30008000

//len为linux kernel image大小

**if** **(**hdr**->**ih\_arch **!=** IH\_CPU\_ARM**){**

printf **(**"Unsupported Architecture 0x%x\n"**,** hdr**->**ih\_arch**);**

SHOW\_BOOT\_PROGRESS **(-**4**);**

**return** 1**;**

**}**

//判断header中的系统架构是否ARM

**switch** **(**hdr**->**ih\_type**){**

**case** IH\_TYPE\_KERNEL**:**

name **=** "Kernel Image"**;**

**break;**

**}**

//如果image类型为kernel，则设置name为Kernel Image

**switch** **(**hdr**->**ih\_comp**)** **{**

**case** IH\_COMP\_GZIP**:**

printf **(**" Uncompressing %s ... "**,** name**);**

**if** **(**gunzip **((**void **\*)**ntohl**(**hdr**->**ih\_load**),** unc\_len**,**

**(**uchar **\*)**data**,** **&**len**)** **!=** 0**)** **{**

puts **(**"GUNZIP ERROR - must RESET board to recover\n"**);**

SHOW\_BOOT\_PROGRESS **(-**6**);**

do\_reset **(**cmdtp**,** flag**,** argc**,** argv**);**

**}**

**break;**

**}**

//如果压缩方式是gzip，则先解压

//gunzip((void\*)ntohl(hdr->ih\_load),unc\_len,(uchar \*)data,(int\*)&len)；

//1.hdr->ih\_load 为输出数据地址0x30008000

//2.unc\_len 为gunzip解压输出数据上限值－8M,do\_bootm函数前有定义#define CFG\_BOOTM\_LEN 0x800000，uint unc\_len = CFG\_BOOTM\_LEN;

//3.data 为输入数据地址data=0x300007FC0 +sizeof(image\_header\_t);

//4.Len 为输入数据长度len = ntohl(hdr->ih\_size);

//在这里load addr与image addr是可以重叠的

**switch** **(**hdr**->**ih\_os**)** **{**

**case** IH\_OS\_LINUX**:**

do\_bootm\_linux **(**cmdtp**,** flag**,** argc**,** argv**,**

addr**,** len\_ptr**,** verify**);**

**break;**

**}**

//如果是linux os，则调用do\_bootm\_linux

**}**

启动linux系统的最终的函数为do\_bootm\_linux，继续来分析。

void do\_bootm\_linux **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[],**

ulong addr**,** ulong **\***len\_ptr**,** int verify**)**

**{**

#ifdef CONFIG\_CMDLINE\_TAG

char **\***commandline **=** getenv **(**"bootargs"**);**

#endif

//获取bootargs noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0

void **(\***theKernel**)(**int zero**,** int arch**,** uint params**);**

theKernel **=** **(**void **(\*)(**int**,** int**,** uint**))**ntohl**(**hdr**->**ih\_ep**);**

//theKernel是一函数指针，它指向kernel的启动地址，对于uImage格式的内核，kernel=image\_header\_t结构体的ih\_ep值

**if** **(**argc **>=** 3**)** **{**

**...**

//这里检查bootm指令是否有两个以上地址的情况，例如bootm 0x300007CF0 0xxxxxxxx

//这种情况表示文件系统是ram文件系统

//ram文件系统也是使用mkimage打包而成，所以也需要读取ram文件系统image的header，

//将image搬移到header中指定的内存地址处

**}**

**else{**

len **=** data **=** 0**;**

//没有ram文件系统的情况

**}**

**if** **(**data**)** **{**

initrd\_start **=** data**;**

initrd\_end **=** initrd\_start **+** len**;**

**}** **else** **{**

initrd\_start **=** 0**;**

initrd\_end **=** 0**;**

**}**

setup\_initrd\_tag **(**bd**,** initrd\_start**,** initrd\_end**);**

//如果使用了ram文件系统，则需要设置initrd tag，这里并未使用ram文件系统

setup\_start\_tag **(**bd**);**

setup\_memory\_tags **(**bd**);**

setup\_commandline\_tag **(**bd**,** commandline**);**

setup\_end\_tag **(**bd**);**

//设置tag

cleanup\_before\_linux**();**

//关闭中断

//关闭I/D cache

//清空I/D cache

theKernel **(**0**,** bd**->**bi\_arch\_number**,** bd**->**bi\_boot\_params**);**

//启动linux系统的要求

//r0 = 0

//r1 = machine type number

//r2 = tag list物理地址

**}**

## uboot启动流程分析

uboot启动分为两个阶段

第一阶段的功能

Ø 硬件设备初始化

Ø 加载U**-**Boot第二阶段代码到RAM空间

Ø 设置好栈

Ø 跳转到第二阶段代码入口

第二阶段的功能

Ø 初始化本阶段使用的硬件设备

Ø 检测系统内存映射

Ø 将内核从Flash读取到RAM中

Ø 为内核设置启动参数

Ø 调用内核

### 2.1 第一阶段

#include <config.h>

#include <version.h>

// 等效为#include <configs/100ask24x0.h>

// 等效为#include "version\_autogenerated.h"

**.**globl \_start

// 相当于C语言中的Extern，声明此变量，并且告诉链接器此变量是全局的，外部可以访问

// 在u-boot.lds中有用到该变量：

// ENTRY(\_start)

// 即指定入口为\_start,而由下面的\_start的含义可以得知，\_start就是整个start.S的最开始，即整个uboot的代码的开始。

\_start**:** b reset

// \_start后面加上一个冒号’:’，表示其是一个标号Label，类似于C语言goto后面的标号。

// 而同时，\_start的地址值，也就是这个代码的位置了，此处即为代码的最开始，相对的0的位置。

// 如果是从NorFlash或者NandFlash启动，那么其地址是0

// 如果是sdram启动，那么其地址为0x33F80000

// 需要注意的是：\_start本身的值不是0x33F80000或者0，而是\_start的地址是0x33F80000或者0

// 如果需要引用\_start的地址，则需要\_armboot\_start: .word \_start

ldr pc**,** \_undefined\_instruction

ldr pc**,** \_software\_interrupt

ldr pc**,** \_prefetch\_abort

ldr pc**,** \_data\_abort

ldr pc**,** \_not\_used

ldr pc**,** \_irq

ldr pc**,** \_fiq

\_undefined\_instruction**:** **.**word undefined\_instruction

\_software\_interrupt**:** **.**word software\_interrupt

\_prefetch\_abort**:** **.**word prefetch\_abort

\_data\_abort**:** **.**word data\_abort

\_not\_used**:** **.**word not\_used

\_irq**:** **.**word irq

\_fiq**:** **.**word fiq

// ldr的作用：

// 以第一个\_undefined\_instruction为例，就是将地址为\_undefined\_instruction中的一个word的值，赋值给pc。

// .word的作用：

// 以第一个\_undefined\_instruction为例，用C语言来表达就是：

// \_undefined\_instruction = &undefined\_instruction

**.**balignl 16**,**0xdeadbeef

// 所以意思就是，接下来的代码，都要16字节对齐，不足之处，用0xdeadbeef填充。

// 前面的代码段一共是60个字节。

\_TEXT\_BASE**:**

**.**word TEXT\_BASE

// \_TEXT\_BASE是一个标号地址，此地址中是一个word类型的变量，变量名是TEXT\_BASE

// 在config.mk中

// CPPFLAGS := $(DBGFLAGS) $(OPTFLAGS) $(RELFLAGS) \

-D\_\_KERNEL\_\_ -DTEXT\_BASE=$(TEXT\_BASE) \

// 所以\_TEXT\_BASE为0x33F80000

**.**globl \_armboot\_start

\_armboot\_start**:**

**.**word \_start

//\_armboot\_start设置为\_start地址,也就是33f80000 T \_start(System.map)

**.**globl \_bss\_start

\_bss\_start**:**

**.**word \_\_bss\_start

**.**globl \_bss\_end

\_bss\_end**:**

**.**word \_end

// 关于\_bss\_start和\_bss\_end都只是两个标号，对应着此处的地址。

// 而两个地址里面分别存放的值是\_\_bss\_start和\_end,这两个的值，根据注释所说，是定义在开发板相关的链接脚本里面的，定义如下：

//. = ALIGN(4);

// \_\_bss\_start = .;

// .bss : { \*(.bss) }

// \_end = .;

**.**globl FREE\_RAM\_END

FREE\_RAM\_END**:**

**.**word 0x0badc0de

**.**globl FREE\_RAM\_SIZE

FREE\_RAM\_SIZE**:**

**.**word 0x0badc0de

// 关于FREE\_RAM\_END和FREE\_RAM\_SIZE，这里只是两个标号，之所以也是声明为全局变量，是因为uboot的源码中会用到这两个变量。

// 但是这里有点特别的是，这两个变量，将在本源码start.S中的后面要用到，而在后面用到这两个变量之前，uboot的C源码中，会先去修改这两个值

**.**globl PreLoadedONRAM

PreLoadedONRAM**:**

**.**word 0

// 这个变量用来在通过ViVi 或OpenOCD等工具将U-boot直接下到内存运行后，能在U-boot运行出现菜单和引导kernel前将U-boot停下来。

// 这个变量在这里定义为 0，如果将U-boot直接下到内存中运行，这个值会在后面被修改为1，在 Main.c 文件的 main\_loop()函数中，

// 如果检测到这个值为1，就会打印一段向导，然后跳出这个函数，进行死循环。

#ifdef CONFIG\_USE\_IRQ

/\* IRQ stack memory (calculated at run-time) \*/

**.**globl IRQ\_STACK\_START

IRQ\_STACK\_START**:**

**.**word 0x0badc0de

/\* IRQ stack memory (calculated at run-time) \*/

**.**globl FIQ\_STACK\_START

FIQ\_STACK\_START**:**

**.**word 0x0badc0de

#endif

// IRQ\_STACK\_START和FIQ\_STACK\_START，也是在cpu\_init中用到了。

// 不过此处，是只有当定义了宏CONFIG\_USE\_IRQ的时候，才用到这两个变量，其含义也很明显，

// 只有用到了中断IRQ，才会用到中断的堆栈，才有中端堆栈的起始地址。

// 快速中断FIQ，同理。

reset**:**

mrs r0**,**cpsr

bic r0**,**r0**,**#0x1f

orr r0**,**r0**,**#0xd3

msr cpsr**,**r0

// bic:清除r0的bit[4:0]位

// orr:r0与0xd3算数或运算，然后将结果给r0,即把r0的bit[7:6]和bit[4]和bit[2:0]置为1。

// CPSR=0xD3的位域及含义:

// CPSR位域：7 6 5 4 3 2 1

// 位域含义： I F M4 M3 M2 M1

// 0xD3 ：1 1 0

// 对应含义： 关闭中断IRQ 关闭快速中断FIQ 设置cpu为svc模式

# define pWTCON 0x53000000

# define INTMOD 0X4A000004

# define INTMSK 0x4A000008 /\* Interupt-Controller base addresses \*/

# define INTSUBMSK 0x4A00001C

# define CLKDIVN 0x4C000014 /\* clock divisor register \*/

ldr r0**,** **=**pWTCON

mov r1**,** #0x0

str r1**,** **[**r0**]**

mov r1**,** #0xffffffff

ldr r0**,** **=**INTMSK

str r1**,** **[**r0**]**

ldr r1**,** **=**0x3ff

ldr r0**,** **=**INTSUBMSK

str r1**,** **[**r0**]**

// 关闭看门狗，屏蔽所有中断和子中断

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

adr r0**,** \_start /\* r0 <- current position of code \*/

ldr r1**,** \_TEXT\_BASE /\* test if we run from flash or RAM \*/

cmp r0**,** r1 /\* don not reloc during debug \*/

blne cpu\_init\_crit

#endif

// 通过adr指令得到当前代码的地址信息，

// 如果U-boot是从RAM开始运行，r0=0x33F80000

// 如果U-boot从Flash开始运行，即从处理器对应的地址运行，则r0=0x0000

// 然后比较r0和\_TEXT\_BASE，如果不相等，则需要执行cpu\_init\_crit

// 在sdram中运行就不需要重新做很多初始化动作了，例如sdram的初始化

stack\_setup**:**

ldr r0**,** \_TEXT\_BASE /\* upper 128 KiB: relocated uboot \*/

sub r0**,** r0**,** #CFG\_MALLOC\_LEN /\* malloc area\*/

sub r0**,** r0**,** #CFG\_GBL\_DATA\_SIZE /\* bdinfo\*/

#ifdef CONFIG\_USE\_IRQ

sub r0**,** r0**,** #**(**CONFIG\_STACKSIZE\_IRQ**+**CONFIG\_STACKSIZE\_FIQ**)**

#endif

sub sp**,** r0**,** #12 /\* leave 3 words for abort-stack \*/

// 设置堆栈地址，

// 0x33F80000-CFG\_MALLOC\_LEN-CFG\_GBL\_DATA\_SIZE-CONFIG\_STACKSIZE\_IR-CONFIG\_STACKSIZE\_FIQ-12

// 其中12个字节大小为终止异常所用到的堆栈大小

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

bl clock\_init

#endif

// 设置系统时钟

#ifndef CONFIG\_SKIP\_RELOCATE\_UBOOT

relocate**:** /\* relocate U-Boot to RAM \*/

adr r0**,** \_start /\* r0 <- current position of code \*/

ldr r1**,** \_TEXT\_BASE /\* test if we run from flash or RAM \*/

cmp r0**,** r1 /\* do not reloc during debug \*/

beq clear\_bss

ldr r2**,** \_armboot\_start

ldr r3**,** \_bss\_start

sub r2**,** r3**,** r2 /\* r2 <- size of armboot \*/

bl CopyCode2Ram /\* r0: source, r1: dest, r2: size \*/

#endif /\* CONFIG\_SKIP\_RELOCATE\_UBOOT \*/

// 如果是在sdram中运行，直接跳转到clear\_bss

// 如果从nandflash或norflash运行，则需要将flash中的uboot代码搬移到ram中。

// 此时r0的值为0x0，r1为0x33F80000

// \_armboot\_start = 0，\_bss\_start为text+rodata+data的结束位置，即代码段。

// 所以将flash 0x0处的代码搬移r2个size到x33F80000

clear\_bss**:**

ldr r0**,** \_bss\_start /\* find start of bss segment \*/

ldr r1**,** \_bss\_end /\* stop here \*/

mov r2**,** #0x00000000 /\* clear \*/

clbss\_l**:**str r2**,** **[**r0**]** /\* clear loop... \*/

add r0**,** r0**,** #4

cmp r0**,** r1

ble clbss\_l

// 清空bss段，bss段包含未初始化的全局变量和静态变量，需要清0

SetLoadFlag**:**

/\* Set a global flag, PreLoadedONRAM \*/

adr r0**,** \_start /\* r0 <- current position of code \*/

ldr r1**,** \_TEXT\_BASE /\* test if we run from flash or RAM \*/

cmp r0**,** r1 /\* do not reloc during debug \*/

ldr r2**,** **=**PreLoadedONRAM

mov r3**,** #1

streq r3**,** **[**r2**]**

ldr pc**,** \_start\_armboot

\_start\_armboot**:** **.**word start\_armboot

// 如果是从sdram中运行，设置PreLoadedONRAM=1

// 接下来跳转到 start\_armboot，这个时候已经跳转到sdram中运行了

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

cpu\_init\_crit**:**

mov r0**,** #0

mcr p15**,** 0**,** r0**,** c7**,** c7**,** 0 /\* flush v3/v4 cache \*/

mcr p15**,** 0**,** r0**,** c8**,** c7**,** 0 /\* flush v4 TLB \*/

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

bic r0**,** r0**,** #0x00002300 @ clear bits 13**,** 9**:**8 **(--**V**-** **--**RS**)**

bic r0**,** r0**,** #0x00000087 @ clear bits 7**,** 2**:**0 **(**B**---** **-**CAM**)**

orr r0**,** r0**,** #0x00000002 @ set bit 2 **(**A**)** Align

orr r0**,** r0**,** #0x00001000 @ set bit 12 **(**I**)** I**-**Cache

mcr p15**,** 0**,** r0**,** c1**,** c0**,** 0

mov ip**,** lr

bl lowlevel\_init

mov lr**,** ip

mov pc**,** lr

#endif /\* CONFIG\_SKIP\_LOWLEVEL\_INIT \*/

// 第一段代码表示清理v3或v4的缓存和清空清空指令和数据的TLB

// 第二段代码表示关闭mmu，并清空和设置对应的位，功能如下表：

// Register bits Function Value

// 13 中断向量表位置 0 = Low address = 0x00000000

// 1 = High address =0xFFFF0000

// 12 I-Cache enable 0 = disable 1 = enable

// 9 rom protection

// 8 system protection

// 7 大小端 0 = 小端 1 = 大端

// 2 D-Cache enable 0 = disable 1 = enable

// 1 Align fault enable 0 = disable 1 = enable

// 0 MMU enable 0 = disable 1 = enable

// 第三段首先将返回地址暂存到ip寄存器，然后在调用lowlevel\_init，lowlevel\_init对sdram进行初始化操作。

#define S\_FRAME\_SIZE 72

#define S\_OLD\_R0 68

#define S\_PSR 64

#define S\_PC 60

#define S\_LR 56

#define S\_SP 52

#define S\_IP 48

#define S\_FP 44

#define S\_R10 40

#define S\_R9 36

#define S\_R8 32

#define S\_R7 28

#define S\_R6 24

#define S\_R5 20

#define S\_R4 16

#define S\_R3 12

#define S\_R2 8

#define S\_R1 4

#define S\_R0 0

#define MODE\_SVC 0x13

#define I\_BIT 0x80

// 先定义一些变量，具体使用见后面

**.**macro get\_bad\_stack

ldr r13**,** \_armboot\_start @ setup our mode stack

sub r13**,** r13**,** #**(**CONFIG\_STACKSIZE**+**CFG\_MALLOC\_LEN**)**

sub r13**,** r13**,** #**(**CFG\_GBL\_DATA\_SIZE**+**8**)** @ reserved a couple spots in abort stack

str lr**,** **[**r13**]** @ save caller lr **/** spsr

mrs lr**,** spsr

str lr**,** **[**r13**,** #4**]**

mov r13**,** #MODE\_SVC @ prepare SVC**-**Mode

msr spsr**,** r13

mov lr**,** pc

movs pc**,** lr

**.**endm

// 发生异常中断后，切换到svc模式下，以便获取到发生错误前的stack

// 当发生异常中断时，例如undefined\_instruction时，先将lr spsr的值保存到

// \_armboot\_start-CONFIG\_STACKSIZE-CFG\_MALLOC\_LEN-CFG\_GBL\_DATA\_SIZE-8处

// 当发生异常中断时，pc和cpsr的值会自动保存到lr spsr寄存器中，

// 也就是需要保存发生错误之前正常程序的pc和cpsr值

// 随后通过修改spsr的值为svc模式

// 将pc值暂存在lr中，然后将lr重新赋给pc，并同时更新cpsr，回到svc模式

// movs区别于mov的是：movs在将LR赋给PC的同时，将SPSR\_mode赋给CPSR

**.**macro bad\_save\_user\_regs

sub sp**,** sp**,** #S\_FRAME\_SIZE

stmia sp**,** **{**r0 **-** r12**}** @ Calling r0**-**r12

ldr r2**,** \_armboot\_start

sub r2**,** r2**,** #**(**CONFIG\_STACKSIZE**+**CFG\_MALLOC\_LEN**)**

sub r2**,** r2**,** #**(**CFG\_GBL\_DATA\_SIZE**+**8**)** @ set base 2 words into abort stack

ldmia r2**,** **{**r2 **-** r3**}** @ get pc**,** cpsr

add r0**,** sp**,** #S\_FRAME\_SIZE @ restore sp\_SVC

add r5**,** sp**,** #S\_SP

mov r1**,** lr

stmia r5**,** **{**r0 **-** r3**}** @ save sp\_SVC**,** lr\_SVC**,** pc**,** cpsr

mov r0**,** sp

**.**endm

// use bad\_save\_user\_regs for abort/prefetch/undef/swi…，该函数主要用来发生错误时保存用户寄存器的值

// 流程如下：

// 先令sp指针自减S\_FRAME\_SIZE(72)，这个时候的sp已经是svc模式下的sp

// 把r0至r12保存到SP中，ia表示，sp的值increase after save

// 使用get\_bad\_stack保存的pc、cpsr寄存器加载到r2,r3

// r0 = 原来的sp值

// 设置r5 = sp+52

// 保存sp\_svc lr\_svc pc cpsr的值

// “原来SP + 52”的地方 存入 r0，即“原来的SP值”，就是sp\_SVC，即被中断时的SP值

// “原来SP + 56”的地方 存入 r1，即lr，就是lr\_SVC，即被中断时的lr值

// “原来SP + 60”的地方 存入 r2，即pc，就是“使用get\_bad\_stack保存的pc”，即被中断时的pc值

// “原来SP + 64”的地方 存入 r3，即cpsr，就是“使用get\_bad\_stack保存的cpsr”，即被中断时的cpsr值

// 也就是从sp-72开始处，依次存储了r0~r12, sp\_SVC, lr\_SVC, pc, cpsr 17个寄存器

// 最后将sp-72的值赋给r0,后面函数调用会直接获取到sp-72地址处的值，并打印这些保存的寄存器值，方便定位错误

**.**align 5

undefined\_instruction**:**

get\_bad\_stack

bad\_save\_user\_regs

bl do\_undefined\_instruction

**.**align 5

software\_interrupt**:**

get\_bad\_stack

bad\_save\_user\_regs

bl do\_software\_interrupt

**.**align 5

prefetch\_abort**:**

get\_bad\_stack

bad\_save\_user\_regs

bl do\_prefetch\_abort

**.**align 5

data\_abort**:**

get\_bad\_stack

bad\_save\_user\_regs

bl do\_data\_abort

**.**align 5

not\_used**:**

get\_bad\_stack

bad\_save\_user\_regs

bl do\_not\_used

// 异常处理，先切换到svc模式，获取出错时的寄存器值，并打印

**.**align 5

irq**:**

/\* add by www.100ask.net to use IRQ for USB and DMA \*/

sub lr**,** lr**,** #4 @ the **return** address

ldr sp**,** IRQ\_STACK\_START @ the stack **for** irq

stmdb sp**!,** **{** r0**-**r12**,**lr **}** @ save registers

ldr lr**,** **=**int\_return @ set the **return** addr

ldr pc**,** **=**IRQ\_Handle @ call the isr

int\_return**:**

ldmia sp**!,** **{** r0**-**r12**,**pc **}^** @ **return** from interrupt

// irq中断处理

**.**align 5

fiq**:**

get\_fiq\_stack

/\* someone ought to write a more effiction fiq\_save\_user\_regs \*/

irq\_save\_user\_regs

bl do\_fiq

irq\_restore\_user\_regs

// fiq中断处理

### 2.2 第二阶段

第二阶段的代码从start\_armboot函数开始，并进入main\_loop循环，代码分析如下：

void start\_armboot **(**void**)**

**{**

init\_fnc\_t **\*\***init\_fnc\_ptr**;**

char **\***s**;**

//定义局部变量，init\_fnc\_ptr指向函数指针数组，其中typedef int (init\_fnc\_t) (void);

gd **=** **(**gd\_t**\*)(**\_armboot\_start **-** CFG\_MALLOC\_LEN **-** **sizeof(**gd\_t**));**

\_\_asm\_\_ \_\_volatile\_\_**(**""**:** **:** **:**"memory"**);**

memset **((**void**\*)**gd**,** 0**,** **sizeof** **(**gd\_t**));**

gd**->**bd **=** **(**bd\_t**\*)((**char**\*)**gd **-** **sizeof(**bd\_t**));**

memset **(**gd**->**bd**,** 0**,** **sizeof** **(**bd\_t**));**

//gd全局变量解释：

//#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm ("r8")

//声明gd这么一个全局的指针，这个指针指向gd\_t结构体类型，并且这个gd指针是保存在ARM的r8这个寄存器里面的。

//gd\_t : global data数据结构定义，其成员主要是一些全局的系统初始化参数。

//bd\_t : board info数据结构定义，要是保存开发板的相关参数。

//最后gd指向的内存位置为\_armboot\_start - CFG\_MALLOC\_LEN - sizeof(gd\_t)

//gd->bd = (bd\_t\*)((char\*)gd - sizeof(bd\_t))：

//bd指向的内存位置为gd - sizeof(bd\_t），也就是gd内存块的下方

monitor\_flash\_len **=** \_bss\_start **-** \_armboot\_start**;**

//代码段的长度，这部分内存段需要保护，而不能随意更改。

**for** **(**init\_fnc\_ptr **=** init\_sequence**;** **\***init\_fnc\_ptr**;** **++**init\_fnc\_ptr**)**

**{**

**if** **((\***init\_fnc\_ptr**)()** **!=** 0**)**

**{**

hang **();**

**}**

**}**

//系统初始化，详见2.2.1

#ifndef CFG\_NO\_FLASH

/\* configure available FLASH banks \*/

size **=** flash\_init **();**

display\_flash\_config **(**size**);**

#endif /\* CFG\_NO\_FLASH \*/

//norflash初始化，详见2.2.2

mem\_malloc\_init **(**\_armboot\_start **-** CFG\_MALLOC\_LEN**);**

//初始化malloc区域，该内存区位于0x33F80000下的256k，其中128k用于环境变量的存储，128k用于malloc

#if (CONFIG\_COMMANDS & CFG\_CMD\_NAND)

puts **(**"NAND: "**);**

nand\_init**();** /\* go init the NAND \*/

#endif

//nandflash初始化，详见2.2.3

env\_relocate **();**

//环境变量重定位，将存在flash中的128k环境变量值搬移到sdram中的malloc内存区，详见1.4.2

gd**->**bd**->**bi\_ip\_addr **=** getenv\_IPaddr **(**"ipaddr"**);**

//获取环境变量中的ip地址，详见2.2.4

**{**

int i**;**

ulong reg**;**

char **\***s**,** **\***e**;**

char tmp**[**64**];**

i **=** getenv\_r **(**"ethaddr"**,** tmp**,** **sizeof** **(**tmp**));**

s **=** **(**i **>** 0**)** **?** tmp **:** **NULL;**

**for** **(**reg **=** 0**;** reg **<** 6**;** **++**reg**)** **{**

gd**->**bd**->**bi\_enetaddr**[**reg**]** **=** s **?** simple\_strtoul **(**s**,** **&**e**,** 16**)** **:** 0**;**

**if** **(**s**)**

s **=** **(\***e**)** **?** e **+** 1 **:** e**;**

**}**

**}**

//获取mac地址

devices\_init **();**

//创建设备列表并将串口设备添加到列表中，详细见2.2.5

jumptable\_init **();**

//初始化常用函数跳转表，详细见2.2.6

console\_init\_r**();**

//初始化控制台，即标准输入、标准输出和标准错误，在这里都是串口,详细见2.2.7

Port\_Init**();**

//对应端口初始化

//如果不在ram启动，则打开中断并启动usb dnw功能

**if** **(!**PreLoadedONRAM**)** **{**

enable\_interrupts **();**

usb\_init**();**

**}**

//设置默认的下载内存地址

/\* 如果存在则从环境变量中读取装载地址，其默认为 ulong load\_addr = 0x30000000; \*/

**if** **((**s **=** getenv **(**"loadaddr"**))** **!=** **NULL)** **{**

load\_addr **=** simple\_strtoul **(**s**,** **NULL,** 16**);**

**}**

/\*设置默认的网络下载文件名\*/

#if (CONFIG\_COMMANDS & CFG\_CMD\_NET)

**if** **((**s **=** getenv **(**"bootfile"**))** **!=** **NULL)** **{**

copy\_filename **(**BootFile**,** s**,** **sizeof** **(**BootFile**));**

**}**

#endif /\* CFG\_CMD\_NET \*/

**for** **(;;)** **{**

//进入主循环

main\_loop **();**

**}**

**}**

void main\_loop **(**void**)**

**{**

static char lastcommand**[**CFG\_CBSIZE**]** **=** **{** 0**,** **};**

int len**;**

int rc **=** 1**;**

int flag**;**

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

char **\***s**;**

int bootdelay**;**

#endif

//mtd初始化，详细见2.2.8

#ifdef CONFIG\_JFFS2\_CMDLINE

extern int mtdparts\_init**(**void**);**

**if** **(!**getenv**(**"mtdparts"**))**

**{**

run\_command**(**"mtdparts default"**,** 0**);**

**}**

**else**

**{**

mtdparts\_init**();**

**}**

#endif

//获取到计时参数

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

s **=** getenv **(**"bootdelay"**);**

bootdelay **=** s **?** **(**int**)**simple\_strtol**(**s**,** **NULL,** 10**)** **:** CONFIG\_BOOTDELAY**;**

//如果是ram启动，直接进入命令模式

**if** **(**PreLoadedONRAM**)** **{**

printf**(**"Use these steps to program the image to flash:\n"**);**

printf**(**"1. In OpenOCD\n"**);**

printf**(**" Run the 'halt' command to halt u-boot\n"**);**

printf**(**" Run the 'load\_image <file> <address>' command to load file to SDRAM\n"**);**

printf**(**" Run the 'resume' command to resume u-boot\n"**);**

printf**(**"2. In u-boot, use the flash commands to program the image to flash\n"**);**

printf**(**"Or, use the tftp or nfs command to download file, and then program the flash.\n"**);**

**goto** PROMPT**;**

**}**

//获取启动命令

s **=** getenv **(**"bootcmd"**);**

//如果计时完成并且没有按键打断，则直接启动linux内核

**if** **(**bootdelay **>=** 0 **&&** s **&&** **!**abortboot **(**bootdelay**))**

**{**

printf**(**"Booting Linux ...\n"**);**

//启动命令nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0启动linux

run\_command **(**s**,** 0**);**

**}**

#endif /\* CONFIG\_BOOTDELAY \*/

//启动定制化菜单，如果退出定制化菜单，则进入命令行模式

run\_command**(**"menu"**,** 0**);**

//命令行模式

PROMPT**:**

**for** **(;;)** **{**

//读取一行

len **=** readline **(**CFG\_PROMPT**);**

flag **=** 0**;** /\* assume no special flags for now \*/

//如果输入数据>0，保存当前指令

**if** **(**len **>** 0**)**

strcpy **(**lastcommand**,** console\_buffer**);**

//如果只是按了回车，判断是否需要重新执行上一条指令

**else** **if** **(**len **==** 0**)**

flag **|=** CMD\_FLAG\_REPEAT**;**

//如果按了ctrl+c，则不处理，否则运行输入命令

**if** **(**len **==** **-**1**)**

puts **(**"<INTERRUPT>\n"**);**

**else**

rc **=** run\_command **(**lastcommand**,** flag**);**

//如果rc<=0,则不要重复上一条指令

**if** **(**rc **<=** 0**)** **{**

/\* invalid command or not repeatable, forget it \*/

lastcommand**[**0**]** **=** 0**;**

**}**

**}**

**}**

#### 2.2.1系统初始化代码分析

依次执行如下列表的函数。

init\_fnc\_t **\***init\_sequence**[]** **=** **{**

cpu\_init**,** /\* basic cpu dependent setup \*/

board\_init**,** /\* basic board dependent setup \*/

interrupt\_init**,** /\* set up exceptions \*/

env\_init**,** /\* initialize environment \*/

init\_baudrate**,** /\* initialze baudrate settings \*/

serial\_init**,** /\* serial communications setup \*/

console\_init\_f**,** /\* stage 1 init of console \*/

display\_banner**,** /\* say that we are here \*/

#if defined(CONFIG\_DISPLAY\_CPUINFO)

print\_cpuinfo**,** /\* display cpu info (and speed) \*/

#endif

#if defined(CONFIG\_DISPLAY\_BOARDINFO)

checkboard**,** /\* display board info \*/

#endif

dram\_init**,** /\* configure available RAM banks \*/

display\_dram\_config**,**

**NULL,**

**};**

##### 2.2.1.1 cpu\_init代码分析

//定义了IRQ堆栈起始地址，FIQ堆栈起始地址，空闲RAM的结束地址，空闲RAM的总长度

int cpu\_init **(**void**)**

**{**

#ifdef CONFIG\_USE\_IRQ

IRQ\_STACK\_START **=** \_armboot\_start **-** CFG\_MALLOC\_LEN **-** CFG\_GBL\_DATA\_SIZE **-** 4**;**

FIQ\_STACK\_START **=** IRQ\_STACK\_START **-** CONFIG\_STACKSIZE\_IRQ**;**

FREE\_RAM\_END **=** FIQ\_STACK\_START **-** CONFIG\_STACKSIZE\_FIQ **-** CONFIG\_STACKSIZE**;**

FREE\_RAM\_SIZE **=** FREE\_RAM\_END **-** PHYS\_SDRAM\_1**;**

#else

FREE\_RAM\_END **=** \_armboot\_start **-** CFG\_MALLOC\_LEN **-** CFG\_GBL\_DATA\_SIZE **-** 4 **-** CONFIG\_STACKSIZE**;**

FREE\_RAM\_SIZE **=** FREE\_RAM\_END **-** PHYS\_SDRAM\_1**;**

#endif

**return** 0**;**

**}**

##### 2.2.1.2 board\_init代码分析

//设置GPIO口的初始值，并且设置机器号和TAG列表的存放地址，该两个参数主要用于启动linux

//theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);

int board\_init **(**void**)**

**{**

S3C24X0\_CLOCK\_POWER **\*** const clk\_power **=** S3C24X0\_GetBase\_CLOCK\_POWER**();**

S3C24X0\_GPIO **\*** const gpio **=** S3C24X0\_GetBase\_GPIO**();**

/\* set up the I/O ports \*/

gpio**->**GPACON **=** 0x007FFFFF**;**

gpio**->**GPBCON **=** 0x00044555**;**

gpio**->**GPBUP **=** 0x000007FF**;**

gpio**->**GPCCON **=** 0xAAAAAAAA**;**

gpio**->**GPCUP **=** 0x0000FFFF**;**

gpio**->**GPDCON **=** 0xAAAAAAAA**;**

gpio**->**GPDUP **=** 0x0000FFFF**;**

gpio**->**GPECON **=** 0xAAAAAAAA**;**

gpio**->**GPEUP **=** 0x0000FFFF**;**

gpio**->**GPFCON **=** 0x000055AA**;**

gpio**->**GPFUP **=** 0x000000FF**;**

gpio**->**GPGCON **=** 0xFF95FFBA**;**

gpio**->**GPGUP **=** 0x0000FFFF**;**

gpio**->**GPHCON **=** 0x002AFAAA**;**

gpio**->**GPHUP **=** 0x000007FF**;**

**if** **(**isS3C2410**)**

**{**

/\* arch number of SMDK2410-Board \*/

gd**->**bd**->**bi\_arch\_number **=** MACH\_TYPE\_SMDK2410**;**

**}**

**else**

**{**

/\* arch number of SMDK2440-Board \*/

gd**->**bd**->**bi\_arch\_number **=** MACH\_TYPE\_S3C2440**;**

**}**

/\* adress of boot parameters \*/

gd**->**bd**->**bi\_boot\_params **=** 0x30000100**;**

**return** 0**;**

**}**

##### 2.2.1.3 interrupt\_init代码分析

//初始化定时器。为bootdelay的倒计时打下基础

int interrupt\_init **(**void**)**

**{**

S3C24X0\_TIMERS **\*** const timers **=** S3C24X0\_GetBase\_TIMERS**();**

/\* use PWM Timer 4 because it has no output \*/

/\* prescaler for Timer 4 is 16 \*/

timers**->**TCFG0 **=** 0x0f00**;**

**if** **(**timer\_load\_val **==** 0**)**

**{**

/\*

\* for 10 ms clock period @ PCLK with 4 bit divider = 1/2

\* (default) and prescaler = 16. Should be 10390

\* @33.25MHz and 15625 @ 50 MHz

\*/

timer\_load\_val **=** get\_PCLK**()/(**2 **\*** 16 **\*** 100**);**

**}**

/\* load value for 10 ms timeout \*/

lastdec **=** timers**->**TCNTB4 **=** timer\_load\_val**;**

/\* auto load, manual update of Timer 4 \*/

timers**->**TCON **=** **(**timers**->**TCON **&** **~**0x0700000**)** **|** 0x600000**;**

/\* auto load, start Timer 4 \*/

timers**->**TCON **=** **(**timers**->**TCON **&** **~**0x0700000**)** **|** 0x500000**;**

timestamp **=** 0**;**

**return** **(**0**);**

**}**

##### 2.2.1.4 env\_init代码分析

//将默认的环境变量赋值给gd对应的变量

//因为env\_init要早于静态存储器的初始化，所以无法进行env的读写，这里将gd中的env相关变量进行配置，

//默认设置env为valid。方便后面env\_relocate函数进行真正的env从nand到ram的relocate。

int env\_init**(**void**)**

**{**

gd**->**env\_addr **=** **(**ulong**)&**default\_environment**[**0**];**

gd**->**env\_valid **=** 1**;**

**return** **(**0**);**

**}**

##### 2.2.1.5 init\_baudrate代码分析

//从env\_addr对应的位置获取baudrate对应的参数

static int init\_baudrate **(**void**)**

**{**

char tmp**[**64**];** /\* long enough for environment variables \*/

int i **=** getenv\_r **(**"baudrate"**,** tmp**,** **sizeof** **(**tmp**));**

gd**->**bd**->**bi\_baudrate **=** gd**->**baudrate **=** **(**i **>** 0**)**

**?** **(**int**)** simple\_strtoul **(**tmp**,** **NULL,** 10**)**

**:** CONFIG\_BAUDRATE**;**

**return** **(**0**);**

**}**

##### 2.2.1.6 serial\_init代码分析

//串口初始化

int serial\_init **(**void**)**

**{**

serial\_setbrg **();**

**return** **(**0**);**

**}**

void serial\_setbrg **(**void**)**

**{**

S3C24X0\_UART **\*** const uart **=** S3C24X0\_GetBase\_UART**(**UART\_NR**);**

int i**;**

unsigned int reg **=** 0**;**

/\* value is calculated so : (int)(PCLK/16./baudrate) -1 \*/

reg **=** get\_PCLK**()** **/** **(**16 **\*** gd**->**baudrate**)** **-** 1**;**

/\* FIFO enable, Tx/Rx FIFO clear \*/

uart**->**UFCON **=** 0x07**;**

uart**->**UMCON **=** 0x0**;**

/\* Normal,No parity,1 stop,8 bit \*/

uart**->**ULCON **=** 0x3**;**

/\*

\* tx=level,rx=edge,disable timeout int.,enable rx error int.,

\* normal,interrupt or polling

\*/

uart**->**UCON **=** 0x245**;**

uart**->**UBRDIV **=** reg**;**

#ifdef CONFIG\_HWFLOW

uart**->**UMCON **=** 0x1**;** /\* RTS up \*/

#endif

**for** **(**i **=** 0**;** i **<** 100**;** i**++);**

**}**

##### 2.2.1.7 console\_init\_f代码分析

//控制台初始化，填充gd里面的have\_console ：gd->have\_console = 1。

int console\_init\_f **(**void**)**

**{**

gd**->**have\_console **=** 1**;**

**return** **(**0**);**

**}**

##### 2.2.1.8 display\_banner代码分析

//显示uboot的信息

static int display\_banner **(**void**)**

**{**

printf **(**"\n\n%s\n\n"**,** version\_string**);**

debug **(**"U-Boot code: %08lX -> %08lX BSS: -> %08lX\n"**,**

\_armboot\_start**,** \_bss\_start**,** \_bss\_end**);**

#ifdef CONFIG\_USE\_IRQ

debug **(**"IRQ Stack: %08lx\n"**,** IRQ\_STACK\_START**);**

debug **(**"FIQ Stack: %08lx\n"**,** FIQ\_STACK\_START**);**

#endif

**return** **(**0**);**

**}**

##### 2.2.1.9 dram\_init代码分析

//设置sdram内存地址和大小

int dram\_init **(**void**)**

**{**

gd**->**bd**->**bi\_dram**[**0**].**start **=** PHYS\_SDRAM\_1**;**

gd**->**bd**->**bi\_dram**[**0**].**size **=** PHYS\_SDRAM\_1\_SIZE**;**

**return** 0**;**

**}**

##### 2.2.1.10 display\_dram\_config代码分析

//显示DRAM大小

static int display\_dram\_config **(**void**)**

**{**

int i**;**

#ifdef DEBUG

puts **(**"RAM Configuration:\n"**);**

**for(**i**=**0**;** i**<**CONFIG\_NR\_DRAM\_BANKS**;** i**++)** **{**

printf **(**"Bank #%d: %08lx "**,** i**,** gd**->**bd**->**bi\_dram**[**i**].**start**);**

print\_size **(**gd**->**bd**->**bi\_dram**[**i**].**size**,** "\n"**);**

**}**

#else

ulong size **=** 0**;**

**for** **(**i**=**0**;** i**<**CONFIG\_NR\_DRAM\_BANKS**;** i**++)** **{**

size **+=** gd**->**bd**->**bi\_dram**[**i**].**size**;**

**}**

puts**(**"DRAM: "**);**

print\_size**(**size**,** "\n"**);**

#endif

**return** **(**0**);**

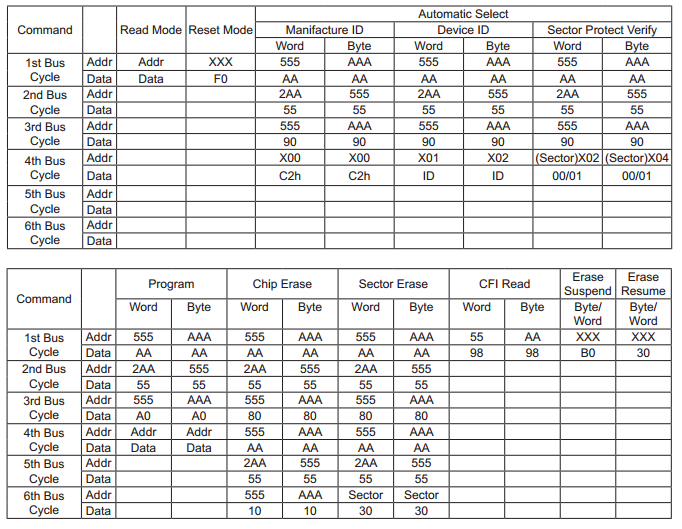
**}**

#### 2.2.2 norflash初始化

Nor Flash 具有像内存一样的接口，它可以像内存一样读，却不可以像内存一样写，Nor Flash 的写、擦除都需要发出特定的命令。

一种是jedec 探测，就是在内核里面事先定义一个数组，该数组里面放有不同厂家各个芯片的一些参数，探测的时候将 flash 的 ID 和数组里面的 ID一一比较，如果发现相同的，就使用该数组的参数。另一种是 cfi 探测，就是直接发各种命令来读取芯片的信息，比如 ID、容量等。jedec 探测的优点就是简单，缺点是如果内核要支持的 flash 种类很多，这个数组就会很庞大。这里使用jedec探测。

norflash操作指令如下图：



//norflash 结构体

**typedef** struct **{**

ulong size**;** /\* total bank size in bytes \*/

ushort sector\_count**;** /\* number of erase units \*/

ulong flash\_id**;** /\* combined device & manufacturer code \*/

ulong start**[**CFG\_MAX\_FLASH\_SECT**];** /\* physical sector start addresses \*/

uchar protect**[**CFG\_MAX\_FLASH\_SECT**];** /\* sector protection status \*/

#ifdef CFG\_FLASH\_CFI

uchar portwidth**;** /\* the width of the port \*/

uchar chipwidth**;** /\* the width of the chip \*/

ushort buffer\_size**;** /\* # of bytes in write buffer \*/

ulong erase\_blk\_tout**;** /\* maximum block erase timeout \*/

ulong write\_tout**;** /\* maximum write timeout \*/

ulong buffer\_write\_tout**;** /\* maximum buffer write timeout \*/

ushort vendor**;** /\* the primary vendor id \*/

ushort cmd\_reset**;** /\* vendor specific reset command \*/

ushort interface**;** /\* used for x8/x16 adjustments \*/

ushort legacy\_unlock**;** /\* support Intel legacy (un)locking \*/

uchar manufacturer\_id**;** /\* manufacturer id \*/

ushort device\_id**;** /\* device id \*/

ushort device\_id2**;** /\* extended device id \*/

ushort ext\_addr**;** /\* extended query table address \*/

ushort cfi\_version**;** /\* cfi version \*/

ushort cfi\_offset**;** /\* offset for cfi query \*/

ulong addr\_unlock1**;** /\* unlock address 1 for AMD flash roms \*/

ulong addr\_unlock2**;** /\* unlock address 2 for AMD flash roms \*/

const char **\***name**;** /\* human-readable name \*/

#endif

**}** flash\_info\_t**;**

unsigned long flash\_init **(**void**)**

**{**

unsigned long size **=** 0**;**

int i**;**

#define BANK\_BASE(i) (((unsigned long [CFI\_MAXFLASH\_BANKS])CFG\_FLASH\_BANKS\_LIST)[i])

//BANK\_BASE(i)就是第i片falsh的地址。

//(1) (unsigned long [CFI\_MAX\_FLASH\_BANKS])是一个类型，它是一个数组类型；

//(2) CONFIG\_FLASH\_BANKS\_LIST这个宏展开就是"{ CONFIG\_FLASH\_BASE}",这里只有一个

//所以(((unsigned long [CFI\_MAX\_FLASH\_BANKS])CFG\_FLASH\_BANKS\_LIST)[i])就是将(2)中大括号的内容转换成(1)所描述的数组类型

**for** **(**i **=** 0**;** i **<** CFG\_MAX\_FLASH\_BANKS**;** **++**i**)** **{**

flash\_info**[**i**].**flash\_id **=** FLASH\_UNKNOWN**;**

**if** **(!**flash\_detect\_legacy **(**BANK\_BASE**(**i**),** i**))**

flash\_get\_size **(**BANK\_BASE**(**i**),** i**);**

size **+=** flash\_info**[**i**].**size**;**

**}**

//保护norflash整个uboot代码段

#if (CFG\_MONITOR\_BASE >= CFG\_FLASH\_BASE)

flash\_protect **(**FLAG\_PROTECT\_SET**,**

CFG\_MONITOR\_BASE**,**

CFG\_MONITOR\_BASE **+** monitor\_flash\_len **-** 1**,**

flash\_get\_info**(**CFG\_MONITOR\_BASE**));**

#endif

**return** **(**size**);**

**}**

//使用legacy方法去检测norflash

//Legacy检测方式是通过AMD和Intel的标准指令进行nor flash的ID读取，

//随后将读取的ID跟jedec\_table定义的器件ID进行对比，从而获取器件信息，以及该器件对应的读写控制指令集。

//JZ2440上使用的norflash为MBM29LV160BE，属于amd标准指令。

static int flash\_detect\_legacy**(**ulong base**,** int banknum**)**

**{**

flash\_info\_t **\***info **=** **&**flash\_info**[**banknum**];**

**if** **(**board\_flash\_get\_legacy**(**base**,** banknum**,** info**))** **{**

//设置norflash芯片操作为word模式

**if** **(!**info**->**vendor**)** **{** //此时vendor为0

int modes**[]** **=** **{**

CFI\_CMDSET\_AMD\_STANDARD**,**

CFI\_CMDSET\_INTEL\_STANDARD

**};**

int i**;**

**for** **(**i **=** 0**;** i **<** **sizeof(**modes**)** **/** **sizeof(**modes**[**0**]);** i**++)** **{** //amd和intel两种模式分别尝试去获取芯片信息

int ret **=** 0**;**

info**->**vendor **=** modes**[**i**];**

//设置vendor信息为amd或intel其中的一种

info**->**start**[**0**]** **=** base**;**

//设置物理起始地址为0x000000000

**if** **(**info**->**vendor **==** CFI\_CMDSET\_INTEL\_STANDARD**)** **{** //使用intel指令集去尝试获取norflash信息，我们这里会失败

flash\_read\_jedec\_ids**(**info**);**

debug**(**"JEDEC PROBE: ID %x %x %x\n"**,**info**->**manufacturer\_id**,**info**->**device\_id**,**info**->**device\_id2**);**

ret **=** jedec\_flash\_match**(**info**,** base**);**

**}** **else** **{**

//使用amd指令集去尝试获取norflash信息

ret **=** jedec\_flash\_probe**(**info**,** base**);**

**}**

**if** **(**ret**)**

**break;**

**}**

**}**

**switch(**info**->**vendor**)** **{**

**case** CFI\_CMDSET\_INTEL\_PROG\_REGIONS**:**

**case** CFI\_CMDSET\_INTEL\_STANDARD**:**

**case** CFI\_CMDSET\_INTEL\_EXTENDED**:**

info**->**cmd\_reset **=** FLASH\_CMD\_RESET**;**

**break;**

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

**case** CFI\_CMDSET\_AMD\_EXTENDED**:**

**case** CFI\_CMDSET\_AMD\_LEGACY**:**

info**->**cmd\_reset **=** AMD\_CMD\_RESET**;** //amd芯片的复位指令为0xF0

**break;**

**}**

info**->**flash\_id **=** FLASH\_MAN\_CFI**;** //设置flash\_id为FLASH\_MAN\_CFI

**return** 1**;**

**}**

**return** 0**;** /\* use CFI \*/

**}**

//设置norflash芯片操作为word模式，word模式和byte模式的操作指令是不一样的，详细见datasheet

ulong board\_flash\_get\_legacy**(**ulong base**,** int banknum**,** flash\_info\_t **\***info**)**

**{**

info**->**portwidth **=** CFG\_FLASH\_CFI\_WIDTH**;**

info**->**chipwidth **=** CFG\_FLASH\_CFI\_WIDTH**;**

info**->**interface **=** FLASH\_CFI\_X16**;**

**return** 1**;**

**}**

//amd指令集探测norflash信息，其中MTD\_UADDR\_0x0555\_0x02AA表示如果想获取norflash的信息，

//第一条指令为往0x555地址写入一个特殊的值 第二条指令为往0x2AA地址写入一个特殊的值

//也就是要获取norflash信息前需要解锁的地址

//norflash MBM29LV160BE属于MTD\_UADDR\_0x0555\_0x02AA

int jedec\_flash\_probe**(**flash\_info\_t **\***info**,** ulong base**)**

**{**

int u**;**

**for** **(**u **=** MTD\_UADDR\_0x0555\_0x02AA**;** u **<=** MTD\_UADDR\_UNNECESSARY**;** u**++)** **{**

info**->**addr\_unlock1 **=** unlock\_addrs**[**u**].**addr1**;**

info**->**addr\_unlock2 **=** unlock\_addrs**[**u**].**addr2**;**

flash\_read\_jedec\_ids**(**info**);**

debug**(**"JEDEC PROBE: ID %x %x %x\n"**,**

info**->**manufacturer\_id**,**

info**->**device\_id**,**

info**->**device\_id2**);**

**if** **(**jedec\_flash\_match**(**info**,** base**))**

**return** 1**;**

**}**

**return** 0**;**

**}**

//读取芯片信息

void flash\_read\_jedec\_ids **(**flash\_info\_t **\*** info**)**

**{**

info**->**manufacturer\_id **=** 0**;**

info**->**device\_id **=** 0**;**

info**->**device\_id2 **=** 0**;**

**switch** **(**info**->**vendor**)** **{**

**case** CFI\_CMDSET\_INTEL\_PROG\_REGIONS**:**

**case** CFI\_CMDSET\_INTEL\_STANDARD**:**

**case** CFI\_CMDSET\_INTEL\_EXTENDED**:**

cmdset\_intel\_read\_jedec\_ids**(**info**);**

**break;**

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

**case** CFI\_CMDSET\_AMD\_EXTENDED**:**

cmdset\_amd\_read\_jedec\_ids**(**info**);**

**break;**

**default:**

**break;**

**}**

**}**

static void cmdset\_amd\_read\_jedec\_ids**(**flash\_info\_t **\***info**)**

**{**

flash\_write\_cmd**(**info**,** 0**,** 0**,** AMD\_CMD\_RESET**);**

//往0x0地址写入0xF0，芯片复位

flash\_unlock\_seq**(**info**,** 0**);**

//往0x555地址写入0xAA 往0x2AA写入0x55 解锁命令

flash\_write\_cmd**(**info**,** 0**,** info**->**addr\_unlock1**,** FLASH\_CMD\_READ\_ID**);**//往0x555地址写入0x90 读取MANUFACTURER\_ID命令

udelay**(**1000**);** /\* some flash are slow to respond \*/

info**->**manufacturer\_id **=** flash\_read\_uchar **(**info**,**

FLASH\_OFFSET\_MANUFACTURER\_ID**);**

//从地址0x00处读取MANUFACTURER\_ID

**switch** **(**info**->**chipwidth**){**

**case** FLASH\_CFI\_8BIT**:**

info**->**device\_id **=** flash\_read\_uchar **(**info**,**

FLASH\_OFFSET\_DEVICE\_ID**);**

**if** **(**info**->**device\_id **==** 0x7E**)** **{**

/\* AMD 3-byte (expanded) device ids \*/

info**->**device\_id2 **=** flash\_read\_uchar **(**info**,**

FLASH\_OFFSET\_DEVICE\_ID2**);**

info**->**device\_id2 **<<=** 8**;**

info**->**device\_id2 **|=** flash\_read\_uchar **(**info**,**

FLASH\_OFFSET\_DEVICE\_ID3**);**

**}**

**break;**

**case** FLASH\_CFI\_16BIT**:**

info**->**device\_id **=** flash\_read\_word **(**info**,**

FLASH\_OFFSET\_DEVICE\_ID**);**

//从0x01地址处读取DEVICE\_ID

**break;**

**default:**

**break;**

**}**

flash\_write\_cmd**(**info**,** 0**,** 0**,** AMD\_CMD\_RESET**);** //复位

**}**

//前面已经获取了nor flash的相关信息，和列表中的信息进行对比

//这里获取到的nor flash信息为

/\*

.mfr\_id = MANUFACTURER\_AMD,

.dev\_id = AM29LV160DT,

.name = "AMD AM29LV160DT",

.uaddr = {

[0] = MTD\_UADDR\_0x0AAA\_0x0555, //x8

[1] = MTD\_UADDR\_0x0555\_0x02AA //x16

},

.DevSize = SIZE\_2MiB,

.CmdSet = P\_ID\_AMD\_STD,

.NumEraseRegions= 4,

.regions = {

ERASEINFO(0x10000,31),

ERASEINFO(0x08000,1),

ERASEINFO(0x02000,2),

ERASEINFO(0x04000,1)

\*/

//有了上述信息后，需要将这些相关信息填充到flash\_info\_t结构体当中

int jedec\_flash\_match**(**flash\_info\_t **\***info**,** ulong base**)**

**{**

int ret **=** 0**;**

int i**;**

ulong mask **=** 0xFFFF**;**

**if** **(**info**->**chipwidth **==** 1**)**

mask **=** 0xFF**;**

**for** **(**i **=** 0**;** i **<** ARRAY\_SIZE**(**jedec\_table**);** i**++)** **{**

**if** **((**jedec\_table**[**i**].**mfr\_id **&** mask**)** **==** **(**info**->**manufacturer\_id **&** mask**)** **&&**

**(**jedec\_table**[**i**].**dev\_id **&** mask**)** **==** **(**info**->**device\_id **&** mask**))** **{**

fill\_info**(**info**,** **&**jedec\_table**[**i**],** base**);**

ret **=** 1**;**

**break;**

**}**

**}**

**return** ret**;**

**}**

//将jedec\_table中获取到的norflash信息填充到flash\_info\_t结构体当中

static inline void fill\_info**(**flash\_info\_t **\***info**,** const struct amd\_flash\_info **\***jedec\_entry**,** ulong base**)**

**{**

int i**,**j**;**

int sect\_cnt**;**

int size\_ratio**;**

int total\_size**;**

enum uaddr uaddr\_idx**;**

size\_ratio **=** info**->**portwidth **/** info**->**chipwidth**;**

//填充相关信息

info**->**vendor **=** jedec\_entry**->**CmdSet**;**

info**->**erase\_blk\_tout **=** 30000**;**

info**->**buffer\_write\_tout **=** 1000**;**

info**->**write\_tout **=** 100**;**

info**->**name **=** jedec\_entry**->**name**;**

**switch(**info**->**chipwidth**)** **{**

**case** FLASH\_CFI\_8BIT**:**

uaddr\_idx **=** jedec\_entry**->**uaddr**[**0**];**

**break;**

**case** FLASH\_CFI\_16BIT**:**

uaddr\_idx **=** jedec\_entry**->**uaddr**[**1**];** //填充解锁地址类型

**break;**

**case** FLASH\_CFI\_32BIT**:**

uaddr\_idx **=** jedec\_entry**->**uaddr**[**2**];**

**break;**

**default:**

uaddr\_idx **=** MTD\_UADDR\_NOT\_SUPPORTED**;**

**break;**

**}**

info**->**addr\_unlock1 **=** unlock\_addrs**[**uaddr\_idx**].**addr1**;** //填充解锁地址1

info**->**addr\_unlock2 **=** unlock\_addrs**[**uaddr\_idx**].**addr2**;** //填充解锁地址2

//填充每个section的起始物理地址

sect\_cnt **=** 0**;**

total\_size **=** 0**;**

**for** **(**i **=** 0**;** i **<** jedec\_entry**->**NumEraseRegions**;** i**++)** **{**

ulong erase\_region\_size **=** jedec\_entry**->**regions**[**i**]** **>>** 8**;**

ulong erase\_region\_count **=** **(**jedec\_entry**->**regions**[**i**]** **&** 0xff**)** **+** 1**;**

total\_size **+=** erase\_region\_size **\*** erase\_region\_count**;**

debug **(**"erase\_region\_count = %d erase\_region\_size = %d\n"**,**

erase\_region\_count**,** erase\_region\_size**);**

**for** **(**j **=** 0**;** j **<** erase\_region\_count**;** j**++)** **{**

**if** **(**sect\_cnt **>=** CFG\_MAX\_FLASH\_SECT**)** **{**

printf**(**"ERROR: too many flash sectors\n"**);**

**break;**

**}**

info**->**start**[**sect\_cnt**]** **=** base**;**

base **+=** **(**erase\_region\_size **\*** size\_ratio**);**

sect\_cnt**++;**

**}**

**}**

info**->**sector\_count **=** sect\_cnt**;** //填充section数

info**->**size **=** total\_size **\*** size\_ratio**;** //填充容量大小

**}**

//写保护开关函数，from为起始地址 to为结束地址，flag决定是设置还是清空写保护

void flash\_protect **(**int flag**,** ulong from**,** ulong to**,** flash\_info\_t **\***info**)**

**{**

ulong b\_end **=** info**->**start**[**0**]** **+** info**->**size **-** 1**;** /\* bank end address \*/

short s\_end **=** info**->**sector\_count **-** 1**;** /\* index of last sector \*/

int i**;**

debug **(**"flash\_protect %s: from 0x%08lX to 0x%08lX\n"**,**

**(**flag **&** FLAG\_PROTECT\_SET**)** **?** "ON" **:**

**(**flag **&** FLAG\_PROTECT\_CLEAR**)** **?** "OFF" **:** "???"**,**

from**,** to**);**

/\* Do nothing if input data is bad. \*/

**if** **(**info**->**sector\_count **==** 0 **||** info**->**size **==** 0 **||** to **<** from**)** **{**

**return;**

**}**

/\* There is nothing to do if we have no data about the flash

\* or the protect range and flash range don't overlap.

\*/

**if** **(**info**->**flash\_id **==** FLASH\_UNKNOWN **||**

to **<** info**->**start**[**0**]** **||** from **>** b\_end**)** **{**

**return;**

**}**

**for** **(**i**=**0**;** i**<**info**->**sector\_count**;** **++**i**)** **{**

ulong end**;** /\* last address in current sect \*/

end **=** **(**i **==** s\_end**)** **?** b\_end **:** info**->**start**[**i **+** 1**]** **-** 1**;**

/\* Update protection if any part of the sector

\* is in the specified range.

\*/

**if** **(**from **<=** end **&&** to **>=** info**->**start**[**i**])** **{**

**if** **(**flag **&** FLAG\_PROTECT\_CLEAR**)** **{**

#if defined(CFG\_FLASH\_PROTECTION)

flash\_real\_protect**(**info**,** i**,** 0**);**

#else

info**->**protect**[**i**]** **=** 0**;**

//这里的写保护操作仅仅是用一个标志位来判断

#endif /\* CFG\_FLASH\_PROTECTION \*/

debug **(**"protect off %d\n"**,** i**);**

**}**

**else** **if** **(**flag **&** FLAG\_PROTECT\_SET**)** **{**

#if defined(CFG\_FLASH\_PROTECTION)

flash\_real\_protect**(**info**,** i**,** 1**);**

#else

info**->**protect**[**i**]** **=** 1**;**

#endif /\* CFG\_FLASH\_PROTECTION \*/

debug **(**"protect on %d\n"**,** i**);**

**}**

**}**

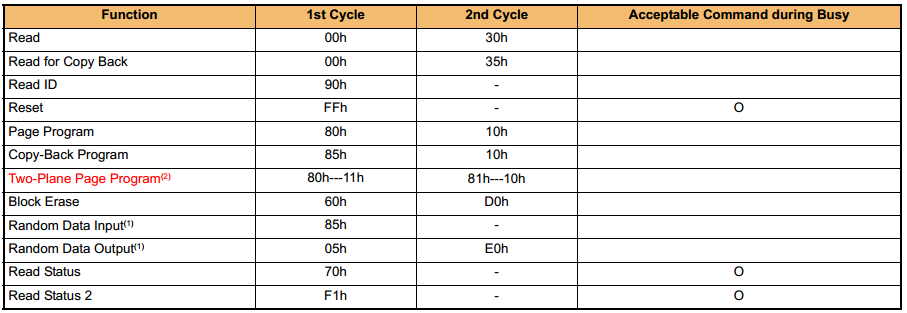
**}**

**}**

#### 2.2.3 nandflash初始化

nandflash的操作和norflash不太一样，nandflash不是ram器件，不能像内存一样直接操作，需要通过nandflash控制器进行操作。一般nandflash操作顺序为写指令，写地址，读写数据。

nandflash操作指令集如下：



nand初始化完成

void nand\_init**(**void**)**

**{**

int i**;**

unsigned int size **=** 0**;**

**for** **(**i **=** 0**;** i **<** CFG\_MAX\_NAND\_DEVICE**;** i**++)** **{**

nand\_init\_chip**(&**nand\_info**[**i**],** **&**nand\_chip**[**i**],** base\_address**[**i**]);**

size **+=** nand\_info**[**i**].**size**;**

**if** **(**nand\_curr\_device **==** **-**1**)**

nand\_curr\_device **=** i**;**

**}**

printf**(**"%lu MiB\n"**,** size **/** **(**1024 **\*** 1024**));**

**}**

//nandflash mtd初始化

//mtd\_info mtd结构体 定义了memroy操作的框架

//nand\_chip结构体 定义了nandflash芯片本身的一些操作

static void nand\_init\_chip**(**struct mtd\_info **\***mtd**,** struct nand\_chip **\***nand**,**ulong base\_addr**)**

**{**

mtd**->**priv **=** nand**;**

nand**->**IO\_ADDR\_R **=** nand**->**IO\_ADDR\_W **=** **(**void \_\_iomem **\*)**base\_addr**;**

board\_nand\_init**(**nand**);**

**if** **(**nand\_scan**(**mtd**,** 1**)** **==** 0**)** **{**

**if** **(!**mtd**->**name**)**

mtd**->**name **=** **(**char **\*)**default\_nand\_name**;**

**}** **else**

mtd**->**name **=** **NULL;**

**}**

//初始化nand\_chip结构体

void board\_nand\_init**(**struct nand\_chip **\***chip**)**

**{**

S3C2440\_NAND **\*** const s3c2440nand **=** S3C2440\_GetBase\_NAND**();**

s3c24x0\_nand\_inithw**();** //nandflash初始化

//设置读数据寄存器

chip**->**IO\_ADDR\_R **=** **(**void **\*)&**s3c2440nand**->**NFDATA**;**

//设置写数据寄存器

chip**->**IO\_ADDR\_W **=** **(**void **\*)&**s3c2440nand**->**NFDATA**;**

//切换IO\_ADDR\_W指向的寄存器地址

chip**->**hwcontrol **=** s3c2440\_nand\_hwcontrol**;**

//判断nandflash是否处于就绪状态

chip**->**dev\_ready **=** s3c2440\_nand\_devready**;**

//是否使能该芯片

chip**->**select\_chip **=** s3c2440\_nand\_select\_chip**;**

chip**->**options **=** 0**;**

chip**->**eccmode **=** NAND\_ECC\_SOFT**;** //设置ecc模式为软件模式

**}**

//nandflash初始化，设置时序，使能nandflash控制器，初始化ecc，使能片选

static void s3c24x0\_nand\_inithw**(**void**)**

**{**

S3C2440\_NAND **\*** const s3c2440nand **=** S3C2440\_GetBase\_NAND**();**

#define TACLS 0

#define TWRPH0 4

#define TWRPH1 2

/\* Set flash memory timing \*/

s3c2440nand**->**NFCONF **=** **(**TACLS**<<**12**)|(**TWRPH0**<<**8**)|(**TWRPH1**<<**4**);**

/\* Initialize ECC, enable chip select, NAND flash controller enable \*/

s3c2440nand**->**NFCONT **=** **(**1**<<**4**)|(**0**<<**1**)|(**1**<<**0**);**

**}**

//切换当前操作的寄存器是操作CMD,ADDR还是DATA

static void s3c2440\_nand\_hwcontrol**(**struct mtd\_info **\***mtd**,** int cmd**)**

**{**

S3C2440\_NAND **\*** const s3c2440nand **=** S3C2440\_GetBase\_NAND**();**

struct nand\_chip **\***chip **=** mtd**->**priv**;** //设置mtd的私有结构体指向nand\_chip结构体

**switch** **(**cmd**)** **{**

**case** NAND\_CTL\_SETNCE**:**

**case** NAND\_CTL\_CLRNCE**:**

printf**(**"%s: called for NCE\n"**,** \_\_FUNCTION\_\_**);**

**break;**

**case** NAND\_CTL\_SETCLE**:**

chip**->**IO\_ADDR\_W **=** **(**void **\*)&**s3c2440nand**->**NFCMD**;**

**break;**

**case** NAND\_CTL\_SETALE**:**

chip**->**IO\_ADDR\_W **=** **(**void **\*)&**s3c2440nand**->**NFADDR**;**

**break;**

/\* NAND\_CTL\_CLRCLE: \*/

/\* NAND\_CTL\_CLRALE: \*/

**default:**

chip**->**IO\_ADDR\_W **=** **(**void **\*)&**s3c2440nand**->**NFDATA**;**

**break;**

**}**

**}**

//判断nandflash操作是否就绪

static int s3c2440\_nand\_devready**(**struct mtd\_info **\***mtd**)**

**{**

S3C2440\_NAND **\*** const s3c2440nand **=** S3C2440\_GetBase\_NAND**();**

**return** **(**s3c2440nand**->**NFSTAT **&** S3C2440\_NFSTAT\_READY**);**

**}**

//是否使能该nandflash

static void s3c2440\_nand\_select\_chip**(**struct mtd\_info **\***mtd**,** int chip**)**

**{**

S3C2440\_NAND **\*** const s3c2440nand **=** S3C2440\_GetBase\_NAND**();**

**if** **(**chip **==** **-**1**)** **{**

s3c2440nand**->**NFCONT **|=** S3C2440\_NFCONT\_nFCE**;**

**}** **else** **{**

s3c2440nand**->**NFCONT **&=** **~**S3C2440\_NFCONT\_nFCE**;**

**}**

**}**

int nand\_scan **(**struct mtd\_info **\***mtd**,** int maxchips**)**

**{**

int i**,** j**,** nand\_maf\_id**,** nand\_dev\_id**,** busw**;**

//获取mtd私有数据

struct nand\_chip **\***this **=** mtd**->**priv**;**

//我们这里的BUS宽度为8bit，也就是DATA线的条数

busw **=** this**->**options **&** NAND\_BUSWIDTH\_16**;**

**if** **(!**this**->**chip\_delay**)**

this**->**chip\_delay **=** 20**;**

//设置nandflash命令操作函数，我们使用的nand芯片是2048一页

//后面改函数会替换成nand\_command\_lp

**if** **(**this**->**cmdfunc **==** **NULL)**

this**->**cmdfunc **=** nand\_command**;**

//nand相关操作后的等待函数，例如擦除，写等

**if** **(**this**->**waitfunc **==** **NULL)**

this**->**waitfunc **=** nand\_wait**;**

//在board\_nand\_init已经定义了select\_chip

**if** **(!**this**->**select\_chip**)**

this**->**select\_chip **=** nand\_select\_chip**;**

//定义nand写一个字节操作，其实就是操作IO\_ADDR\_W寄存器

**if** **(!**this**->**write\_byte**)**

this**->**write\_byte **=** busw **?** nand\_write\_byte16 **:** nand\_write\_byte**;**

//定义nand读一个字节操作

**if** **(!**this**->**read\_byte**)**

this**->**read\_byte **=** busw **?** nand\_read\_byte16 **:** nand\_read\_byte**;**

//定义nand写两个字节操作

**if** **(!**this**->**write\_word**)**

this**->**write\_word **=** nand\_write\_word**;**

//定义nand读两个字节操作

**if** **(!**this**->**read\_word**)**

this**->**read\_word **=** nand\_read\_word**;**

//判断nand对应的page是否是坏块

**if** **(!**this**->**block\_bad**)**

this**->**block\_bad **=** nand\_block\_bad**;**

//nand坏块标记函数

**if** **(!**this**->**block\_markbad**)**

this**->**block\_markbad **=** nand\_default\_block\_markbad**;**

//连续写字节操作函数

**if** **(!**this**->**write\_buf**)**

this**->**write\_buf **=** busw **?** nand\_write\_buf16 **:** nand\_write\_buf**;**

//连续读字节操作函数

**if** **(!**this**->**read\_buf**)**

this**->**read\_buf **=** busw **?** nand\_read\_buf16 **:** nand\_read\_buf**;**

//读出的字节和buf进行比较

**if** **(!**this**->**verify\_buf**)**

this**->**verify\_buf **=** busw **?** nand\_verify\_buf16 **:** nand\_verify\_buf**;**

//这里未使用bbt，暂不分析

**if** **(!**this**->**scan\_bbt**)**

this**->**scan\_bbt **=** nand\_default\_bbt**;**

//选中nand芯片

this**->**select\_chip**(**mtd**,** 0**);**

//发送NAND\_CMD\_READID指令

//整个read id操作为，写命令0x90 写地址0x00 读maker code，读device code，后面还有3个字节可以读

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READID**,** 0x00**,** **-**1**);**

//maker code为0xEC

nand\_maf\_id **=** this**->**read\_byte**(**mtd**);**

//device code为0xDA

nand\_dev\_id **=** this**->**read\_byte**(**mtd**);**

//从nand列表中寻找到device code为0xDA的设备

// {"NAND 256MiB 3,3V 8-bit", 0xDA, 0, 256, 0, NAND\_SAMSUNG\_LP\_OPTIONS | NAND\_NO\_AUTOINCR},

**for** **(**i **=** 0**;** nand\_flash\_ids**[**i**].**name **!=** **NULL;** i**++)** **{**

**if** **(**nand\_dev\_id **!=** nand\_flash\_ids**[**i**].**id**)**

**continue;**

//设置mtd名字

**if** **(!**mtd**->**name**)** mtd**->**name **=** nand\_flash\_ids**[**i**].**name**;**

//256\*1M，1<<20=1MB

this**->**chipsize **=** nand\_flash\_ids**[**i**].**chipsize **<<** 20**;**

//如果列表中的pagesize为空，则继续读后面的2个字节，确定pagesize的大小

**if** **(!**nand\_flash\_ids**[**i**].**pagesize**)** **{**

int extid**;**

//读取第三个字节，Internal Chip Number, Cell Type, Number of Simultaneously Programmed Pages, Etc

extid **=** this**->**read\_byte**(**mtd**);**

//读取第四个字节 Page Size, Block Size,Redundant Area Size, Organization, Serial Access Minimum

extid **=** this**->**read\_byte**(**mtd**);**

//第四个字节的bit0 bit1代表page size，这里为2048

mtd**->**oobblock **=** 1024 **<<** **(**extid **&** 0x3**);**

//右移两位

extid **>>=** 2**;**

//第四个字节的bit3代表每512个字节对应多少个字节的oob数据，这里为16个字节，所以2048个字节需要有64个字节的oob数据

mtd**->**oobsize **=** **(**8 **<<** **(**extid **&** 0x03**))** **\*** **(**mtd**->**oobblock **/** 512**);**

//右移两位

extid **>>=** 2**;**

//第四个字节的bit4~5代表block size大小，这里为128k

mtd**->**erasesize **=** **(**64 **\*** 1024**)** **<<** **(**extid **&** 0x03**);**

//右移两位

extid **>>=** 2**;**

//第四个字节bit6代表bus width，这里为8

busw **=** **(**extid **&** 0x01**)** **?** NAND\_BUSWIDTH\_16 **:** 0**;**

**}** **else** **{**

mtd**->**erasesize **=** nand\_flash\_ids**[**i**].**erasesize**;**

mtd**->**oobblock **=** nand\_flash\_ids**[**i**].**pagesize**;**

mtd**->**oobsize **=** mtd**->**oobblock **/** 32**;**

busw **=** nand\_flash\_ids**[**i**].**options **&** NAND\_BUSWIDTH\_16**;**

**}**

//检查bus的宽度是否和设置的一致

**if** **(**busw **!=** **(**this**->**options **&** NAND\_BUSWIDTH\_16**))** **{**

printk **(**KERN\_INFO "NAND device: Manufacturer ID:"

" 0x%02x, Chip ID: 0x%02x (%s %s)\n"**,** nand\_maf\_id**,** nand\_dev\_id**,**

nand\_manuf\_ids**[**i**].**name **,** mtd**->**name**);**

printk **(**KERN\_WARNING

"NAND bus width %d instead %d bit\n"**,**

**(**this**->**options **&** NAND\_BUSWIDTH\_16**)** **?** 16 **:** 8**,**

busw **?** 16 **:** 8**);**

this**->**select\_chip**(**mtd**,** **-**1**);**

**return** 1**;**

**}**

//为了避免使用除法，所以采用位移的方式加快计算

//ffs(2048)-1 = 0xb = 11 2048

this**->**page\_shift **=** ffs**(**mtd**->**oobblock**)** **-** 1**;**

//ffs(131072) -1 = 0x11 = 17 128K

this**->**bbt\_erase\_shift **=** this**->**phys\_erase\_shift **=** ffs**(**mtd**->**erasesize**)** **-** 1**;**

//ffs(2684354566)-1 = 0x1C = 28 256M

this**->**chip\_shift **=** ffs**(**this**->**chipsize**)** **-** 1**;**

//设置坏块标志在oob处的位置，这里为0

this**->**badblockpos **=** mtd**->**oobblock **>** 512 **?**

NAND\_LARGE\_BADBLOCK\_POS **:** NAND\_SMALL\_BADBLOCK\_POS**;**

//保留bit0，bit1~bit15清0

this**->**options **&=** **~**NAND\_CHIPOPTIONS\_MSK**;**

//设置options，bit0保持不变

this**->**options **|=** nand\_flash\_ids**[**i**].**options **&** NAND\_CHIPOPTIONS\_MSK**;**

//默认设置bit0为1

this**->**options **|=** NAND\_NO\_AUTOINCR**;**

//不满足下面的条件

**if** **(**nand\_maf\_id **!=** NAND\_MFR\_SAMSUNG **&&** **!**nand\_flash\_ids**[**i**].**pagesize**)**

this**->**options **&=** **~**NAND\_SAMSUNG\_LP\_OPTIONS**;**

**if** **(**this**->**options **&** NAND\_4PAGE\_ARRAY**)**

this**->**erase\_cmd **=** multi\_erase\_cmd**;**

**else**

//设置erase命令操作函数

this**->**erase\_cmd **=** single\_erase\_cmd**;**

//由于一个page的大小为2048，需要切换命令操作函数为nand\_command\_lp

**if** **(**mtd**->**oobblock **>** 512 **&&** this**->**cmdfunc **==** nand\_command**)**

this**->**cmdfunc **=** nand\_command\_lp**;**

/\* Try to identify manufacturer \*/

**for** **(**j **=** 0**;** nand\_manuf\_ids**[**j**].**id **!=** 0x0**;** j**++)** **{**

**if** **(**nand\_manuf\_ids**[**j**].**id **==** nand\_maf\_id**)**

**break;**

**}**

**break;**

**}**

//如果没有nand name，出错返回

**if** **(!**nand\_flash\_ids**[**i**].**name**)** **{**

printk **(**KERN\_WARNING "No NAND device found!!!\n"**);**

this**->**select\_chip**(**mtd**,** **-**1**);**

**return** 1**;**

**}**

//maxchips = 1，下面不会运行

**for** **(**i**=**1**;** i **<** maxchips**;** i**++)** **{**

this**->**select\_chip**(**mtd**,** i**);**

/\* Send the command for reading device ID \*/

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READID**,** 0x00**,** **-**1**);**

/\* Read manufacturer and device IDs \*/

**if** **(**nand\_maf\_id **!=** this**->**read\_byte**(**mtd**)** **||**

nand\_dev\_id **!=** this**->**read\_byte**(**mtd**))**

**break;**

**}**

**if** **(**i **>** 1**)**

printk**(**KERN\_INFO "%d NAND chips detected\n"**,** i**);**

//为每个block分配oob\_buf

**if** **(!**this**->**oob\_buf**)** **{**

size\_t len**;**

//计算每个block需要oob\_buf的空间，一个block大小是128k，一个page大小为2k，也就是需要64\*64个字节

len **=** mtd**->**oobsize **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**);**

this**->**oob\_buf **=** kmalloc **(**len**,** GFP\_KERNEL**);**

**if** **(!**this**->**oob\_buf**)** **{**

printk **(**KERN\_ERR "nand\_scan(): Cannot allocate oob\_buf\n"**);**

**return** **-**ENOMEM**;**

**}**

this**->**options **|=** NAND\_OOBBUF\_ALLOC**;**

**}**

//为每个page分配data\_buf

**if** **(!**this**->**data\_buf**)** **{**

size\_t len**;**

//大小为2048+64

len **=** mtd**->**oobblock **+** mtd**->**oobsize**;**

this**->**data\_buf **=** kmalloc **(**len**,** GFP\_KERNEL**);**

**if** **(!**this**->**data\_buf**)** **{**

**if** **(**this**->**options **&** NAND\_OOBBUF\_ALLOC**)**

kfree **(**this**->**oob\_buf**);**

printk **(**KERN\_ERR "nand\_scan(): Cannot allocate data\_buf\n"**);**

**return** **-**ENOMEM**;**

**}**

this**->**options **|=** NAND\_DATABUF\_ALLOC**;**

**}**

//设置芯片个数为1

this**->**numchips **=** i**;**

//设置芯片总大小为256MB

mtd**->**size **=** i **\*** this**->**chipsize**;**

//设置pagemask为该芯片总page数目-1

this**->**pagemask **=** **(**this**->**chipsize **>>** this**->**page\_shift**)** **-** 1**;**

//将oob\_buf区域的值设置为0xff

memset**(**this**->**oob\_buf**,** 0xff**,** mtd**->**oobsize **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**));**

//设置oob区域属性

/\*

NAND Flash出错的时候一般不会造成整个Block或是Page不能读取或是全部出错，

而是整个Page（例如512Bytes）中只有一个或几个bit出错。

一般使用一种比较专用的校验——ECC。ECC能纠正单比特错误和检测双比特错误，而且计算速度很快，

但对1比特以上的错误无法纠正，对2比特以上的错误不保证能检测。

ECC一般每256字节原始数据生成3字节ECC校验数据，所以2048个字节需要3\*8=24字节来纠错

static struct nand\_oobinfo nand\_oob\_64 = {

.useecc = MTD\_NANDECC\_AUTOPLACE,

.eccbytes = 24, //ecc纠错所需要的空间

.eccpos = {

40, 41, 42, 43, 44, 45, 46, 47, //ecc纠错数据存放的位置bit40~63用于存储

48, 49, 50, 51, 52, 53, 54, 55,

56, 57, 58, 59, 60, 61, 62, 63},

//偏移两位，最开始的两个字节用于存储坏块标志，还剩余38个字节可以供其他使用

//64-24-2 = 38个字节

.oobfree = { {2, 38} }

};

\*/

**if** **(!**this**->**autooob**)** **{**

**switch** **(**mtd**->**oobsize**)** **{**

**case** 8**:**

this**->**autooob **=** **&**nand\_oob\_8**;**

**break;**

**case** 16**:**

this**->**autooob **=** **&**nand\_oob\_16**;**

**break;**

**case** 64**:**

this**->**autooob **=** **&**nand\_oob\_64**;**

**break;**

**default:**

printk **(**KERN\_WARNING "No oob scheme defined for oobsize %d\n"**,**

mtd**->**oobsize**);**

/\* BUG(); \*/

**}**

**}**

**if** **(**this**->**options **&** NAND\_BUSWIDTH\_16**)** **{**

mtd**->**oobavail **=** mtd**->**oobsize **-** **(**this**->**autooob**->**eccbytes **+** 2**);**

**if** **(**this**->**autooob**->**eccbytes **&** 0x01**)**

mtd**->**oobavail**--;**

**}** **else**

//oob区域能被fs使用的个数oobavail=64-(24+1)=39

mtd**->**oobavail **=** mtd**->**oobsize **-** **(**this**->**autooob**->**eccbytes **+** 1**);**

//每256个字节ecc纠错就3个字节

this**->**eccsize **=** 256**;**

this**->**eccbytes **=** 3**;**

//设置ecc操作函数

**switch** **(**this**->**eccmode**)** **{**

**case** NAND\_ECC\_SOFT**:**

//ecc计算函数

this**->**calculate\_ecc **=** nand\_calculate\_ecc**;**

//ecc纠错函数

this**->**correct\_data **=** nand\_correct\_data**;**

**break;**

**default:**

printk **(**KERN\_WARNING "Invalid NAND\_ECC\_MODE %d\n"**,** this**->**eccmode**);**

/\* BUG(); \*/

**}**

mtd**->**eccsize **=** this**->**eccsize**;** //256bytes

/\* Set the number of read / write steps for one page to ensure ECC generation \*/

**switch** **(**this**->**eccmode**)** **{**

**case** NAND\_ECC\_SOFT**:**

this**->**eccsteps **=** mtd**->**oobblock **/** 256**;** //2048/256=8，也就是计算2048个字节的ecc需要进行8次

**break;**

**}**

//取消芯片选中

this**->**select\_chip**(**mtd**,** **-**1**);**

//清空pagebuf指向，此时pagebuf不指向任何page

this**->**pagebuf **=** **-**1**;**

mtd**->**type **=** MTD\_NANDFLASH**;**

mtd**->**flags **=** MTD\_CAP\_NANDFLASH **|** MTD\_ECC**;**

mtd**->**ecctype **=** MTD\_ECC\_SW**;**

//设置nand块擦除函数

mtd**->**erase **=** nand\_erase**;**

mtd**->**point **=** **NULL;**

mtd**->**unpoint **=** **NULL;**

//nand\_read = nand\_read\_ecc

mtd**->**read **=** nand\_read**;**

//nand\_write = nand\_write\_ecc

mtd**->**write **=** nand\_write**;**

//nand读操作

mtd**->**read\_ecc **=** nand\_read\_ecc**;**

//nand写操作

mtd**->**write\_ecc **=** nand\_write\_ecc**;**

//

mtd**->**read\_oob **=** nand\_read\_oob**;**

mtd**->**write\_oob **=** nand\_write\_oob**;**

mtd**->**sync **=** nand\_sync**;**

mtd**->**block\_isbad **=** nand\_block\_isbad**;**

mtd**->**block\_markbad **=** nand\_block\_markbad**;**

/\* and make the autooob the default one \*/

memcpy**(&**mtd**->**oobinfo**,** this**->**autooob**,** **sizeof(**mtd**->**oobinfo**));**

/\* Build bad block table \*/

**return** 0**;** //this->scan\_bbt (mtd); // cancelled by www.100ask.net, scan\_bbt uses more time

**}**

//判断nand ofs位置对应的page是否是坏块

static int nand\_block\_bad**(**struct mtd\_info **\***mtd**,** loff\_t ofs**,** int getchip**)**

**{**

int page**,** chipnr**,** res **=** 0**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

u16 bad**;**

page **=** **(**int**)(**ofs **>>** this**->**page\_shift**);**

chipnr **=** **(**int**)(**ofs **>>** this**->**chip\_shift**);**

**if** **(**getchip**)** **{**

nand\_get\_device **(**this**,** mtd**,** FL\_READING**);** //无用函数

this**->**select\_chip**(**mtd**,** chipnr**);** //选中芯片

**}**

**if** **(**this**->**options **&** NAND\_BUSWIDTH\_16**)** **{**

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READOOB**,** this**->**badblockpos **&** 0xFE**,** page **&** this**->**pagemask**);**

bad **=** cpu\_to\_le16**(**this**->**read\_word**(**mtd**));**

**if** **(**this**->**badblockpos **&** 0x1**)**

bad **>>=** 1**;**

**if** **((**bad **&** 0xFF**)** **!=** 0xff**)**

res **=** 1**;**

**}** **else** **{**

//读取对应page的oob数据，这里nand一页大小为2048，oob大小为64个字节

//对于big page的NAND芯片，坏块信息存储在每个block的第一个page的oob的第1和第2个字节中

//读取指定page的oob数据，其中 this->badblockpos=0，表示oob的坏块数据存储在第1~2字节中

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READOOB**,** this**->**badblockpos**,** page **&** this**->**pagemask**);**

//读取oob第一个字节数据，如果是0xff则不是坏块

**if** **(**this**->**read\_byte**(**mtd**)** **!=** 0xff**)**

res **=** 1**;**

// Apply delay or wait for ready/busy pin

// add by www.100ask.net, if not, the erase will be failed

**if** **(!**this**->**dev\_ready**)**

udelay **(**this**->**chip\_delay**);**

**else**

**while** **(!**this**->**dev\_ready**(**mtd**));**

**}**

**if** **(**getchip**)** **{**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);** //释放芯片

**}**

**return** res**;**

**}**

//标记ofs位置对应的page为坏块

static int nand\_default\_block\_markbad**(**struct mtd\_info **\***mtd**,** loff\_t ofs**)**

**{**

struct nand\_chip **\***this **=** mtd**->**priv**;**

u\_char buf**[**2**]** **=** **{**0**,** 0**};**

size\_t retlen**;**

int block**;**

/\* Get block number \*/

block **=** **((**int**)** ofs**)** **>>** this**->**bbt\_erase\_shift**;**

this**->**bbt**[**block **>>** 2**]** **|=** 0x01 **<<** **((**block **&** 0x03**)** **<<** 1**);**

//这里我们没有使用bbt

**if** **(**this**->**options **&** NAND\_USE\_FLASH\_BBT**)**

**return** nand\_update\_bbt **(**mtd**,** ofs**);**

//往ofs对应page的oob区域前两个字节写入0，0，表示为坏块

ofs **+=** mtd**->**oobsize **+** **(**this**->**badblockpos **&** **~**0x01**);**

**return** nand\_write\_oob **(**mtd**,** ofs **,** 2**,** **&**retlen**,** buf**);**

**}**

//page erase操作函数

static void single\_erase\_cmd **(**struct mtd\_info **\***mtd**,** int page**)**

**{**

//操作流程：写命令0x60，写地址page 写命令0xD0

struct nand\_chip **\***this **=** mtd**->**priv**;**

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_ERASE1**,** **-**1**,** page**);**

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_ERASE2**,** **-**1**,** **-**1**);**

**}**

static void nand\_command\_lp **(**struct mtd\_info **\***mtd**,** unsigned command**,** int column**,** int page\_addr**)**

**{**

register struct nand\_chip **\***this **=** mtd**->**priv**;**

//如果指令时READOOB，则需要将column的值增加2048，2048后才是oob存放的位置

//并将指令设置为READ，没有READOOB指令，还是通过普通的READ来完成

**if** **(**command **==** NAND\_CMD\_READOOB**)** **{**

column **+=** mtd**->**oobblock**;**

command **=** NAND\_CMD\_READ0**;**

**}**

//切换命令操作模式

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_SETCLE**);**

//发送对应的命令

this**->**write\_byte**(**mtd**,** command**);**

//切换到数据操作模式

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_CLRCLE**);**

//如果column或page不为-1，需要写入对应的地址

**if** **(**column **!=** **-**1 **||** page\_addr **!=** **-**1**)** **{**

//切换到地址操作模式

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_SETALE**);**

**if** **(**column **!=** **-**1**)** **{**

this**->**write\_byte**(**mtd**,** column **&** 0xff**);** //写入colum的低字节

this**->**write\_byte**(**mtd**,** column **>>** 8**);** //写入colum的高字节

**}**

**if** **(**page\_addr **!=** **-**1**)** **{**

//当chip size大于128M，则需要写入三次page对应的字节

this**->**write\_byte**(**mtd**,** **(**unsigned char**)** **(**page\_addr **&** 0xff**));**

this**->**write\_byte**(**mtd**,** **(**unsigned char**)** **((**page\_addr **>>** 8**)** **&** 0xff**));**

**if** **(**this**->**chipsize **>** **(**128 **<<** 20**))**

this**->**write\_byte**(**mtd**,** **(**unsigned char**)** **((**page\_addr **>>** 16**)** **&** 0xff**));**

**}**

/\* Latch in address \*/

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_CLRALE**);**

**}**

/\*

\* program and erase have their own busy handlers

\* status and sequential in needs no delay

\*/

**switch** **(**command**)** **{**

**case** NAND\_CMD\_CACHEDPROG**:**

**case** NAND\_CMD\_PAGEPROG**:**

**case** NAND\_CMD\_ERASE1**:**

**case** NAND\_CMD\_ERASE2**:**

**case** NAND\_CMD\_SEQIN**:**

**case** NAND\_CMD\_STATUS**:**

**return;**

//如果是reset指令，则等待操作完成

**case** NAND\_CMD\_RESET**:**

**if** **(**this**->**dev\_ready**)**

**break;**

udelay**(**this**->**chip\_delay**);**

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_SETCLE**);**

this**->**write\_byte**(**mtd**,** NAND\_CMD\_STATUS**);**

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_CLRCLE**);**

**while** **(** **!(**this**->**read\_byte**(**mtd**)** **&** 0x40**));**

**return;**

//如果是read指令，还需要发送0x30指令

**case** NAND\_CMD\_READ0**:**

/\* Begin command latch cycle \*/

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_SETCLE**);**

/\* Write out the start read command \*/

this**->**write\_byte**(**mtd**,** NAND\_CMD\_READSTART**);**

/\* End command latch cycle \*/

this**->**hwcontrol**(**mtd**,** NAND\_CTL\_CLRCLE**);**

/\* Fall through into ready check \*/

/\* This applies to read commands \*/

**default:**

/\*

\* If we don't have access to the busy pin, we apply the given

\* command delay

\*/

**if** **(!**this**->**dev\_ready**)** **{**

udelay **(**this**->**chip\_delay**);**

**return;**

**}**

**}**

/\* Apply this short delay always to ensure that we do wait tWB in

\* any case on any machine. \*/

ndelay **(**100**);**

/\* wait until command is processed \*/

**while** **(!**this**->**dev\_ready**(**mtd**));**

**}**

//nand块擦除函数 start:instr->addr,len:instr->len

int nand\_erase\_nand **(**struct mtd\_info **\***mtd**,** struct erase\_info **\***instr**,** int allowbbt**)**

**{**

int page**,** len**,** status**,** pages\_per\_block**,** ret**,** chipnr**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

//擦除的起始地址，必须以block大小128k对齐

**if** **(**instr**->**addr **&** **((**1 **<<** this**->**phys\_erase\_shift**)** **-** 1**))** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_erase: Unaligned address\n"**);**

**return** **-**EINVAL**;**

**}**

//擦除的长度也必须以128k对齐

**if** **(**instr**->**len **&** **((**1 **<<** this**->**phys\_erase\_shift**)** **-** 1**))** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_erase: Length not block aligned\n"**);**

**return** **-**EINVAL**;**

**}**

//擦除的长度不要超过nand总大小

**if** **((**instr**->**len **+** instr**->**addr**)** **>** mtd**->**size**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_erase: Erase past end of device\n"**);**

**return** **-**EINVAL**;**

**}**

instr**->**fail\_addr **=** 0xffffffff**;**

//空函数，无用

nand\_get\_device **(**this**,** mtd**,** FL\_ERASING**);**

//计算擦除的地址所处于的page

page **=** **(**int**)** **(**instr**->**addr **>>** this**->**page\_shift**);**

chipnr **=** **(**int**)** **(**instr**->**addr **>>** this**->**chip\_shift**);**

//计算一个block有多少个page

pages\_per\_block **=** 1 **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**);**

//选中nand

this**->**select\_chip**(**mtd**,** chipnr**);**

//检查nand是否被写保护了

**if** **(**nand\_check\_wp**(**mtd**))** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_erase: Device is write protected!!!\n"**);**

instr**->**state **=** MTD\_ERASE\_FAILED**;**

**goto** erase\_exit**;**

**}**

//暂存要擦除的长度

len **=** instr**->**len**;**

//更改状态为正在擦除

instr**->**state **=** MTD\_ERASING**;**

**while** **(**len**)** **{**

//检查当前page是否是坏块，出现坏块，该block不进行擦除

**if** **(**nand\_block\_checkbad**(**mtd**,** **((**loff\_t**)** page**)** **<<** this**->**page\_shift**,** 0**,** allowbbt**))** **{**

printk **(**KERN\_WARNING "nand\_erase: attempt to erase a bad block at page 0x%08x\n"**,** page**);**

instr**->**state **=** MTD\_ERASE\_FAILED**;**

**goto** erase\_exit**;**

**}**

//假如pagebuf指向的page在当前要擦除的block之间，则需要让当前的page缓存无效

**if** **(**page **<=** this**->**pagebuf **&&** this**->**pagebuf **<** **(**page **+** pages\_per\_block**))**

this**->**pagebuf **=** **-**1**;**

//擦除该block

this**->**erase\_cmd **(**mtd**,** page **&** this**->**pagemask**);**

//等待擦除结束

status **=** this**->**waitfunc **(**mtd**,** this**,** FL\_ERASING**);**

//判断是否擦除成功

**if** **(**status **&** 0x01**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_erase: " "Failed erase, page 0x%08x\n"**,** page**);**

instr**->**state **=** MTD\_ERASE\_FAILED**;**

instr**->**fail\_addr **=** **(**page **<<** this**->**page\_shift**);**

**goto** erase\_exit**;**

**}**

//len长度减去一个block的长度

len **-=** **(**1 **<<** this**->**phys\_erase\_shift**);** //clear one block

//page+=64

page **+=** pages\_per\_block**;**

//检查page是否超过该chip的最大page数

**if** **(**len **&&** **!(**page **&** this**->**pagemask**))** **{**

chipnr**++;**

this**->**select\_chip**(**mtd**,** **-**1**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

**}**

**}**

instr**->**state **=** MTD\_ERASE\_DONE**;**

erase\_exit**:**

ret **=** instr**->**state **==** MTD\_ERASE\_DONE **?** 0 **:** **-**EIO**;**

/\* Do call back function \*/

**if** **(!**ret**)**

mtd\_erase\_callback**(**instr**);**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);**

/\* Return more or less happy \*/

**return** ret**;**

**}**

//读取nand是否被写保护了，读出的字节的最高位是写保护标志位，0代表写保护

static int nand\_check\_wp **(**struct mtd\_info **\***mtd**)**

**{**

struct nand\_chip **\***this **=** mtd**->**priv**;**

/\* Check the WP bit \*/

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_STATUS**,** **-**1**,** **-**1**);**

**return** **(**this**->**read\_byte**(**mtd**)** **&** 0x80**)** **?** 0 **:** 1**;**

**}**

//nand读取函数，从from地址开始读取长度len的数据，存储在buf里面，读取的长度存放在retlen里面

//读取的过程中使用ecc纠正数据

static int nand\_read\_ecc **(**struct mtd\_info **\***mtd**,** loff\_t from**,** size\_t len**,**

size\_t **\*** retlen**,** u\_char **\*** buf**,** u\_char **\*** oob\_buf**,** struct nand\_oobinfo **\***oobsel**)**

**{**

int i**,** j**,** col**,** realpage**,** page**,** end**,** ecc**,** chipnr**,** sndcmd **=** 1**;**

int read **=** 0**,** oob **=** 0**,** ecc\_status **=** 0**,** ecc\_failed **=** 0**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

u\_char **\***data\_poi**,** **\***oob\_data **=** oob\_buf**;**

u\_char ecc\_calc**[**32**];**

u\_char ecc\_code**[**32**];**

int eccmode**,** eccsteps**;**

unsigned **\***oob\_config**;**

int datidx**;**

int blockcheck **=** **(**1 **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**))** **-** 1**;**

int eccbytes**;**

int compareecc **=** 1**;**

int oobreadlen**;**

//读取的区域不要超过nand的最大范围

**if** **((**from **+** len**)** **>** mtd**->**size**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_read\_ecc: Attempt read beyond end of device\n"**);**

**\***retlen **=** 0**;**

**return** **-**EINVAL**;**

**}**

//无用函数

nand\_get\_device **(**this**,** mtd **,**FL\_READING**);**

//使用nand\_oob\_64

**if** **(**oobsel **==** **NULL)**

oobsel **=** **&**mtd**->**oobinfo**;**

//使用nand\_oob\_64

**if** **(**oobsel**->**useecc **==** MTD\_NANDECC\_AUTOPLACE**)**

oobsel **=** this**->**autooob**;**

//是否使用ecc

eccmode **=** oobsel**->**useecc **?** this**->**eccmode **:** NAND\_ECC\_NONE**;**

//保存ecc存放的位置数组

oob\_config **=** oobsel**->**eccpos**;**

//选中nand

chipnr **=** **(**int**)(**from **>>** this**->**chip\_shift**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

//计算from地址处于的page

realpage **=** **(**int**)** **(**from **>>** this**->**page\_shift**);**

//确保该page不超过最大的page数

page **=** realpage **&** this**->**pagemask**;**

//获取columu数

col **=** from **&** **(**mtd**->**oobblock **-** 1**);**

//end=2048

end **=** mtd**->**oobblock**;**

//ecc=256

ecc **=** this**->**eccsize**;**

//eccbytes=3

eccbytes **=** this**->**eccbytes**;**

//下面的条件不成立

**if** **((**eccmode **==** NAND\_ECC\_NONE**)** **||** **(**this**->**options **&** NAND\_HWECC\_SYNDROME**))**

compareecc **=** 0**;**

//oobreadlen=64

oobreadlen **=** mtd**->**oobsize**;**

//下面条件不成立

**if** **(**this**->**options **&** NAND\_HWECC\_SYNDROME**)**

oobreadlen **-=** oobsel**->**eccbytes**;**

**while** **(**read **<** len**)** **{**

//判断要读的colum是否为0，判断要读的数量是否大于2048

int aligned **=** **(!**col **&&** **(**len **-** read**)** **>=** end**);**

/\*

\* 如果要读的位置不是页对齐都话，那么只要先把整页读出来，

\* 取出所需要读取的数据，然后修改读位置，那么以后的读操作都是页对齐的了。

\*/

**if** **(**aligned**)**//如果是对齐的，并且要读的数量>2048，就先放到buf里面

data\_poi **=** **&**buf**[**read**];**

**else**

data\_poi **=** this**->**data\_buf**;**//如果是不对齐的，或者读取的数量少于2048，就先读到暂存区中(page+oob)

/\*

\* 如果我们所需要的数据还存在于缓冲中都话：

\* 1 如果读位置页对齐，我们只要把缓冲中的数据直接拷贝到data\_poi(buf[read])中即可(因为数据存在与缓存中，所以也无需要考虑ecc问题)

\* 2 如果读位置不是页对齐，什么读不要作，让其继续留在缓存(data\_buf)中，以后会从data\_poi(指向缓存data\_buf)中提取所需要的数据。

\*/

**if** **(**realpage **==** this**->**pagebuf **&&** **!**oob\_buf**)** **{**

**if** **(**aligned**)**

memcpy **(**data\_poi**,** this**->**data\_buf**,** end**);**

**goto** readdata**;**

**}**

//发送读page指令

**if** **(**sndcmd**)** **{**

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READ0**,** 0x00**,** page**);**

sndcmd **=** 0**;**

**}**

//oob\_data指向data\_buf[2048]处

**if** **(!**oob\_buf **||** oobsel**->**useecc **==** MTD\_NANDECC\_AUTOPLACE **||**

oobsel**->**useecc **==** MTD\_NANDECC\_AUTOPL\_USR**)**

oob\_data **=** **&**this**->**data\_buf**[**end**];**

//eccsteps=8

eccsteps **=** this**->**eccsteps**;**

**switch** **(**eccmode**)** **{**

**case** NAND\_ECC\_SOFT**:** /\* Software ECC 3/256: Read in a page + oob data \*/

//读取该页的2048个字节

this**->**read\_buf**(**mtd**,** data\_poi**,** end**);**

//分别8次，计算2048个字节的ecc值

**for** **(**i **=** 0**,** datidx **=** 0**;** eccsteps**;** eccsteps**--,** i**+=**3**,** datidx **+=** ecc**)**

this**->**calculate\_ecc**(**mtd**,** **&**data\_poi**[**datidx**],** **&**ecc\_calc**[**i**]);**

**break;**

**}**

//继续读取64字节的oob数据到data\_buf[2048]

//mtd->oobsize - oobreadlen=0, data\_buf[2048]=&oob\_data[0]

this**->**read\_buf**(**mtd**,** **&**oob\_data**[**mtd**->**oobsize **-** oobreadlen**],** oobreadlen**);**

//是否需要跳过ecc校验，这里不跳过

**if** **(!**compareecc**)**

**goto** readoob**;**

//取出刚才读出的ecc值

**for** **(**j **=** 0**;** j **<** oobsel**->**eccbytes**;** j**++)**

ecc\_code**[**j**]** **=** oob\_data**[**oob\_config**[**j**]];**

//纠正数据，传入的参数为data，根据data计算出的ecc值，保存在oob中的ecc值

**for** **(**i **=** 0**,** j **=** 0**,** datidx **=** 0**;** i **<** this**->**eccsteps**;** i**++,** datidx **+=** ecc**)** **{**

ecc\_status **=** this**->**correct\_data**(**mtd**,** **&**data\_poi**[**datidx**],** **&**ecc\_code**[**j**],** **&**ecc\_calc**[**j**]);**

//继续跳入下一个循环纠正数据

j **+=** eccbytes**;**

**if** **(**ecc\_status **==** **-**1**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_read\_ecc: " "Failed ECC read, page 0x%08x\n"**,** page**);**

ecc\_failed**++;**

**}**

**}**

//如果函数参数提供了oob\_buf指针，则需要将oob数据读取到oob\_buf中

//这里跳过

readoob**:**

**if** **(**oob\_buf**)** **{**

**switch(**oobsel**->**useecc**)** **{**

**case** MTD\_NANDECC\_AUTOPLACE**:**

**case** MTD\_NANDECC\_AUTOPL\_USR**:**

**for** **(**i **=** 0**,** j **=** 0**;** j **<** mtd**->**oobavail**;** i**++)** **{**

int from **=** oobsel**->**oobfree**[**i**][**0**];**

int num **=** oobsel**->**oobfree**[**i**][**1**];**

memcpy**(&**oob\_buf**[**oob**],** **&**oob\_data**[**from**],** num**);**

j**+=** num**;**

**}**

oob **+=** mtd**->**oobavail**;**

**break;**

**}**

**}**

readdata**:**

//如果columu不为0，并且读的数据少于2048，则需要读一部分出来放到buf中

//并且由于部分读会将数据先读到缓冲区中，所以需要更新此时的缓冲区pagebuf的指向

**if** **(!**aligned**)** **{**

**for** **(**j **=** col**;** j **<** end **&&** read **<** len**;** j**++)**

buf**[**read**++]** **=** data\_poi**[**j**];**

this**->**pagebuf **=** realpage**;**

**}** **else**

read **+=** mtd**->**oobblock**;**

//等待操作结束

**if** **(!**this**->**dev\_ready**)**

udelay **(**this**->**chip\_delay**);**

**else**

**while** **(!**this**->**dev\_ready**(**mtd**));**

//读取完毕，退出循环

**if** **(**read **==** len**)**

**break;**

//还没有读取完，继续读取，更新column和page值

col **=** 0**;**

realpage**++;**

page **=** realpage **&** this**->**pagemask**;**

**if** **(!**page**)** **{**

chipnr**++;**

this**->**select\_chip**(**mtd**,** **-**1**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

**}**

/\* Check, if the chip supports auto page increment

\* or if we have hit a block boundary.

\*/

**if** **(!**NAND\_CANAUTOINCR**(**this**)** **||** **!(**page **&** blockcheck**))**

sndcmd **=** 1**;**

**}**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);**

/\*

\* Return success, if no ECC failures, else -EBADMSG

\* fs driver will take care of that, because

\* retlen == desired len and result == -EBADMSG

\*/

**\***retlen **=** read**;**

**return** ecc\_failed **?** **-**EBADMSG **:** 0**;**

**}**

//nand写函数，写入数据buf到地址to，长度为len

static int nand\_write\_ecc **(**struct mtd\_info **\***mtd**,** loff\_t to**,** size\_t len**,**

size\_t **\*** retlen**,** const u\_char **\*** buf**,** u\_char **\*** eccbuf**,** struct nand\_oobinfo **\***oobsel**)**

**{**

int startpage**,** page**,** ret **=** **-**EIO**,** oob **=** 0**,** written **=** 0**,** chipnr**;**

int autoplace **=** 0**,** numpages**,** totalpages**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

u\_char **\***oobbuf**,** **\***bufstart**;**

int ppblock **=** **(**1 **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**));** //17-11=6 2的6次方 64 一个块中有64个页

**\***retlen **=** 0**;**

//写入范围不要超过nand总大小

**if** **((**to **+** len**)** **>** mtd**->**size**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_ecc: Attempt to write past end of page\n"**);**

**return** **-**EINVAL**;**

**}**

//to和len必须是2048一页为单位

**if** **(**NOTALIGNED **(**to**)** **||** NOTALIGNED**(**len**))** **{**

printk **(**KERN\_NOTICE "nand\_write\_ecc: Attempt to write not page aligned data\n"**);**

**return** **-**EINVAL**;**

**}**

//无用函数

nand\_get\_device **(**this**,** mtd**,** FL\_WRITING**);**

chipnr **=** **(**int**)(**to **>>** this**->**chip\_shift**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

//判断nand是否写保护

**if** **(**nand\_check\_wp**(**mtd**))**

**goto** out**;**

//使用nand\_oob\_64

**if** **(**oobsel **==** **NULL)**

oobsel **=** **&**mtd**->**oobinfo**;**

**if** **(**oobsel**->**useecc **==** MTD\_NANDECC\_AUTOPLACE**)** **{**

oobsel **=** this**->**autooob**;**

autoplace **=** 1**;**

**}**

**if** **(**oobsel**->**useecc **==** MTD\_NANDECC\_AUTOPL\_USR**)**

autoplace **=** 1**;**

//计算len所占用的page数

totalpages **=** len **>>** this**->**page\_shift**;**

//计算起始page

page **=** **(**int**)** **(**to **>>** this**->**page\_shift**);**

/\* 如果缓存保存的数据在我们要写数据的范围内，把缓存里的数据设置为不可用\*/

**if** **(**page **<=** this**->**pagebuf **&&** this**->**pagebuf **<** **(**page **+** totalpages**))**

this**->**pagebuf **=** **-**1**;**

page **&=** this**->**pagemask**;**

startpage **=** page**;**

//计算出本block中允许被写的页数

//如果起始page为65，则numpages = min(64-1,totalpages),

//也就是如果要写入的总数大于63，则需要在另外一个block里面操作

numpages **=** min **(**ppblock **-** **(**startpage **&** **(**ppblock **-** 1**)),** totalpages**);**

//返回初始化时候创建的oob\_buf地址

oobbuf **=** nand\_prepare\_oobbuf **(**mtd**,** eccbuf**,** oobsel**,** autoplace**,** numpages**);**

//暂存要写数据的起始地址

bufstart **=** **(**u\_char **\*)**buf**;**

**while** **(**written **<** len**)** **{**

//记录当前写入起始地址

this**->**data\_poi **=** **(**u\_char**\*)** **&**buf**[**written**];**

//写入一页

ret **=** nand\_write\_page **(**mtd**,** this**,** page**,** **&**oobbuf**[**oob**],** oobsel**,** **(--**numpages **>** 0**));**

**if** **(**ret**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_ecc: write\_page failed %d\n"**,** ret**);**

**goto** out**;**

**}**

//oob+=64

oob **+=** mtd**->**oobsize**;**

//written+=2048

written **+=** mtd**->**oobblock**;**

//判断是否写完

**if** **(**written **==** len**)**

**goto** cmp**;**

//更新page

page**++;**

//假如page处于block的边缘，也就是63，126这种page号

**if** **(!(**page **&** **(**ppblock **-** 1**))){**

int ofs**;**

//如果要检查数据是否正确，这里是有bug的，需要改成this->data\_poi = bufstart + (startpage-ostartpage)\*mtd->oobblock;

this**->**data\_poi **=** bufstart**;**

//写完一个block后，需要检查写入的数据是否正确，这里nand\_verify\_pages为空，所以不检查。

ret **=** nand\_verify\_pages **(**mtd**,** this**,** startpage**,**

page **-** startpage**,**

oobbuf**,** oobsel**,** chipnr**,** **(**eccbuf **!=** **NULL));**

**if** **(**ret**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_ecc: verify\_pages failed %d\n"**,** ret**);**

**goto** out**;**

**}**

//更新写入的字节数

**\***retlen **=** written**;**

ofs **=** autoplace **?** mtd**->**oobavail **:** mtd**->**oobsize**;**

//eccbuf为空，跳过

**if** **(**eccbuf**)**

eccbuf **+=** **(**page **-** startpage**)** **\*** ofs**;**

//计算还剩余的需要写入的pages

totalpages **-=** page **-** startpage**;**

//是否超过64个block

numpages **=** min **(**totalpages**,** ppblock**);**

//判断page数是否会超过flash总page的大小

page **&=** this**->**pagemask**;**

startpage **=** page**;**

oob **=** 0**;**

this**->**oobdirty **=** 1**;**

//重新提供oobbuf

oobbuf **=** nand\_prepare\_oobbuf **(**mtd**,** eccbuf**,** oobsel**,**

autoplace**,** numpages**);**

/\* Check, if we cross a chip boundary \*/

**if** **(!**page**)** **{**

chipnr**++;**

this**->**select\_chip**(**mtd**,** **-**1**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

**}**

**}**

**}**

//这里nand\_verify\_pages为空，所以不检查。

cmp**:**

this**->**data\_poi **=** bufstart**;**

ret **=** nand\_verify\_pages **(**mtd**,** this**,** startpage**,** totalpages**,**

oobbuf**,** oobsel**,** chipnr**,** **(**eccbuf **!=** **NULL));**

**if** **(!**ret**)**

**\***retlen **=** written**;**

**else**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_ecc: verify\_pages failed %d\n"**,** ret**);**

out**:**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);**

**return** ret**;**

**}**

//准备oobbuf

static u\_char **\*** nand\_prepare\_oobbuf **(**struct mtd\_info **\***mtd**,** u\_char **\***fsbuf**,** struct nand\_oobinfo **\***oobsel**,**

int autoplace**,** int numpages**)**

**{**

struct nand\_chip **\***this **=** mtd**->**priv**;**

int i**,** len**,** ofs**;**

//autoplace=1,条件不成立

**if** **(**fsbuf **&&** **!**autoplace**)**

**return** fsbuf**;**

//如果oob\_buf被修改了，需要重新将oob\_buf置成0xff

**if** **(**this**->**oobdirty**)** **{**

memset **(**this**->**oob\_buf**,** 0xff**,**

mtd**->**oobsize **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**));**

//将oob\_buf的64bytes\*64pages个字节置成0xff

this**->**oobdirty **=** 0**;**

**}**

//由于fsbuf=null，所以返回oob\_buf

**if** **(!**autoplace **||** **!**fsbuf**)** **{**

this**->**oobdirty **=** 1**;** // by thisway.diy, www.100ask.net

**return** this**->**oob\_buf**;**

**}**

**}**

//nand页写操作函数

static int nand\_write\_page **(**struct mtd\_info **\***mtd**,** struct nand\_chip **\***this**,** int page**,**

u\_char **\***oob\_buf**,** struct nand\_oobinfo **\***oobsel**,** int cached**)**

**{**

int i**,** status**;**

u\_char ecc\_code**[**32**];**

int eccmode **=** oobsel**->**useecc **?** this**->**eccmode **:** NAND\_ECC\_NONE**;**

uint **\***oob\_config **=** oobsel**->**eccpos**;**

int datidx **=** 0**,** eccidx **=** 0**,** eccsteps **=** this**->**eccsteps**;**

int eccbytes **=** 0**;**

//这里传进来的cached值没有什么用，固定为0

cached **=** 0**;**

//发出写page命令

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_SEQIN**,** 0x00**,** page**);**

**switch** **(**eccmode**)** **{**

**case** NAND\_ECC\_SOFT**:**

//计算ecc的值

**for** **(;** eccsteps**;** eccsteps**--)** **{**

this**->**calculate\_ecc**(**mtd**,** **&**this**->**data\_poi**[**datidx**],** ecc\_code**);**

**for** **(**i **=** 0**;** i **<** 3**;** i**++,** eccidx**++)**

oob\_buf**[**oob\_config**[**eccidx**]]** **=** ecc\_code**[**i**];**

datidx **+=** this**->**eccsize**;**

**}**

//写入数据到对应的页

this**->**write\_buf**(**mtd**,** this**->**data\_poi**,** mtd**->**oobblock**);**

**break;**

**}**

**if** **(**this**->**options **&** NAND\_HWECC\_SYNDROME**)**

this**->**write\_buf**(**mtd**,** **&**oob\_buf**[**oobsel**->**eccbytes**],** mtd**->**oobsize **-** oobsel**->**eccbytes**);**

**else** **if** **(**eccmode **!=** NAND\_ECC\_NONE**)**

//写入oob数据 0xff表示不改变 保持原样

this**->**write\_buf**(**mtd**,** oob\_buf**,** mtd**->**oobsize**);**

//写入命令0x10

this**->**cmdfunc **(**mtd**,** cached **?** NAND\_CMD\_CACHEDPROG **:** NAND\_CMD\_PAGEPROG**,** **-**1**,** **-**1**);**

//等待写结束

**if** **(!**cached**)** **{**

/\* call wait ready function \*/

status **=** this**->**waitfunc **(**mtd**,** this**,** FL\_WRITING**);**

/\* See if device thinks it succeeded \*/

**if** **(**status **&** 0x01**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "%s: " "Failed write, page 0x%08x, "**,** \_\_FUNCTION\_\_**,** page**);**

**return** **-**EIO**;**

**}**

**}** **else** **{**

/\* FIXME: Implement cached programming ! \*/

/\* wait until cache is ready\*/

/\* status = this->waitfunc (mtd, this, FL\_CACHEDRPG); \*/

**}**

**return** 0**;**

**}**

//nand\_read\_oob 读取对应page的oob数据，该函数用于yaffs文件系统的读操作，yaffs文件系统本身包含了oob数据

//len一般大小为64

//这个函数非常奇怪，如果给定的地址为0x40002类似的，会先读取62个字节的oob，另外两个字节的oob在下一个页中。

static int nand\_read\_oob **(**struct mtd\_info **\***mtd**,** loff\_t from**,** size\_t len**,** size\_t **\*** retlen**,** u\_char **\*** buf**)**

**{**

int i**,** col**,** page**,** chipnr**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

int blockcheck **=** **(**1 **<<** **(**this**->**phys\_erase\_shift **-** this**->**page\_shift**))** **-** 1**;** //64

//计算from所在的起始page

page **=** **(**int**)(**from **>>** this**->**page\_shift**);**

chipnr **=** **(**int**)(**from **>>** this**->**chip\_shift**);**

//计算column值，这里的column只保留了低6位

//举个例子，例如要读取地址为0x40002，则col=2

col **=** from **&** **(**mtd**->**oobsize **-** 1**);** //oobsize = 63 低6位为1，也就保留低6位

**\***retlen **=** 0**;**

//不要超过nand最大的size

**if** **((**from **+** len**)** **>** mtd**->**size**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_read\_oob: Attempt read beyond end of device\n"**);**

**\***retlen **=** 0**;**

**return** **-**EINVAL**;**

**}**

//无效函数

nand\_get\_device **(**this**,** mtd **,** FL\_READING**);**

//选中nand芯片

this**->**select\_chip**(**mtd**,** chipnr**);**

//读取对应的page，col+2048处的内容

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READOOB**,** col**,** page **&** this**->**pagemask**);**

/\*

\* Read the data, if we read more than one page

\* oob data, let the device transfer the data !

\*/

i **=** 0**;**

**while** **(**i **<** len**)** **{**

int thislen **=** mtd**->**oobsize **-** col**;**

thislen **=** min\_t**(**int**,** thislen**,** len**);**

this**->**read\_buf**(**mtd**,** **&**buf**[**i**],** thislen**);**

i **+=** thislen**;**

//继续上述的假设，这里thislen=62,所以先读取62个字节的oob，64个字节的后62个字节

//等待操作结束

**if** **(!**this**->**dev\_ready**)**

udelay **(**this**->**chip\_delay**);**

**else**

**while** **(!**this**->**dev\_ready**(**mtd**));**

//假如没有完成len个字节的读取，读取下一页的oob

**if** **(**i **<** len**)** **{**

page**++;**

col **=** 0**;**

/\* Check, if we cross a chip boundary \*/

**if** **(!(**page **&** this**->**pagemask**))** **{**

chipnr**++;**

this**->**select\_chip**(**mtd**,** **-**1**);**

this**->**select\_chip**(**mtd**,** chipnr**);**

**}**

/\* Check, if the chip supports auto page increment

\* or if we have hit a block boundary.

\*/

**if** **(!**NAND\_CANAUTOINCR**(**this**)** **||** **!(**page **&** blockcheck**))** **{**

/\* For subsequent page reads set offset to 0 \*/

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READOOB**,** 0x0**,** page **&** this**->**pagemask**);**

**}**

**}**

**}**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);**

/\* Return happy \*/

**\***retlen **=** len**;**

**return** 0**;**

**}**

//

static int nand\_write\_oob **(**struct mtd\_info **\***mtd**,** loff\_t to**,** size\_t len**,** size\_t **\*** retlen**,** const u\_char **\*** buf**)**

**{**

int column**,** page**,** status**,** ret **=** **-**EIO**,** chipnr**;**

struct nand\_chip **\***this **=** mtd**->**priv**;**

//计算所需写入的page

page **=** **(**int**)** **(**to **>>** this**->**page\_shift**);**

chipnr **=** **(**int**)** **(**to **>>** this**->**chip\_shift**);**

//保存低6位

column **=** to **&** **(**mtd**->**oobsize **-** 1**);**

**\***retlen **=** 0**;**

//不要写到另外一个页去

**if** **((**column **+** len**)** **>** mtd**->**oobsize**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_oob: Attempt to write past end of page\n"**);**

**return** **-**EINVAL**;**

**}**

//无用函数

nand\_get\_device **(**this**,** mtd**,** FL\_WRITING**);**

//选中芯片

this**->**select\_chip**(**mtd**,** chipnr**);**

//复位

this**->**cmdfunc**(**mtd**,** NAND\_CMD\_RESET**,** **-**1**,** **-**1**);**

//是否有写保护

**if** **(**nand\_check\_wp**(**mtd**))**

**goto** out**;**

//如果写入的oob

**if** **(**page **==** this**->**pagebuf**)**

this**->**pagebuf **=** **-**1**;**

//往该页的2048个字节后写入oob数据

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_SEQIN**,** mtd**->**oobblock **+** column**,** page **&** this**->**pagemask**);**

this**->**write\_buf**(**mtd**,** buf**,** len**);**

//写入0x10命令开始写入

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_PAGEPROG**,** **-**1**,** **-**1**);**

//等待写入完成

status **=** this**->**waitfunc **(**mtd**,** this**,** FL\_WRITING**);**

/\* See if device thinks it succeeded \*/

**if** **(**status **&** 0x01**)** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_oob: " "Failed write, page 0x%08x\n"**,** page**);**

ret **=** **-**EIO**;**

**goto** out**;**

**}**

/\* Return happy \*/

**\***retlen **=** len**;**

#ifdef CONFIG\_MTD\_NAND\_VERIFY\_WRITE

/\* Send command to read back the data \*/

this**->**cmdfunc **(**mtd**,** NAND\_CMD\_READOOB**,** column**,** page **&** this**->**pagemask**);**

**if** **(**this**->**verify\_buf**(**mtd**,** buf**,** len**))** **{**

DEBUG **(**MTD\_DEBUG\_LEVEL0**,** "nand\_write\_oob: " "Failed write verify, page 0x%08x\n"**,** page**);**

ret **=** **-**EIO**;**

**goto** out**;**

**}**

#endif

ret **=** 0**;**

out**:**

/\* Deselect and wake up anyone waiting on the device \*/

nand\_release\_device**(**mtd**);**

**return** ret**;**

**}**

#### 2.2.4 ip地址获取

IPaddr\_t getenv\_IPaddr **(**char **\***var**)**

**{**

**return** **(**string\_to\_ip**(**getenv**(**var**)));**

**}**

//getenv 函数的参数是字符指针变量name，实参就是"ipaddr"

char **\***getenv **(**char **\***name**)**

**{**

int i**,** nxt**;**

WATCHDOG\_RESET**();**

**for** **(**i**=**0**;** env\_get\_char**(**i**)** **!=** '\0'**;** i**=**nxt**+**1**)** **{** //下一条环境变量

int val**;**

**for** **(**nxt**=**i**;** env\_get\_char**(**nxt**)** **!=** '\0'**;** **++**nxt**)** **{**

**if** **(**nxt **>=** CFG\_ENV\_SIZE**)** **{**

**return** **(NULL);**

**}**

**}**//获取一条环境变量，以'\0'结束，并计算该条环境变量的长度

**if** **((**val**=**envmatch**((**uchar **\*)**name**,** i**))** **<** 0**)**

//比较环境变量name是否匹配

**continue;**

**return** **((**char **\*)**env\_get\_addr**(**val**));**//如果匹配成功，返回参数地址

**}**

**return** **(NULL);**

**}**

//经过env\_relocate函数，将env\_get\_char=env\_get\_char\_memory

//获取环境变量内存区对应index的字符

uchar env\_get\_char\_memory **(**int index**)**

**{**

**if** **(**gd**->**env\_valid**)** **{**

**return** **(** **\*((**uchar **\*)(**gd**->**env\_addr **+** index**))** **);**

**}** **else** **{**

**return** **(** default\_environment**[**index**]** **);**

**}**

**}**

//比较“ipaddr"和"ipaddr=192.168.1.230”。

//while (\*s1 == env\_get\_char(i2++)) 就是从default\_environment字符串的指定位置开始比较，

//指定位置由env\_get\_char返回值所定，比较不一致时函数返回-1。

//当比较到default\_environment字符串的IPaddr时，开始有相同的字符出现了，这时while值为1，

//依次比较i,p,a,d,d,r字符，因为每比较一次，S1（“ipaddr"）和i2的值都加1，即指向下一个字符。

//当比较完"r"后，指针指向了下一个字符即"=", "="之后的字符串正是我们需要的，所以return(i2)，i2是赋给val的，

//下一步就是env\_get\_addr(val)。

static int envmatch **(**uchar **\***s1**,** int i2**)**

**{**

**while** **(\***s1 **==** env\_get\_char**(**i2**++))**

**if** **(\***s1**++** **==** '='**)**

**return(**i2**);**

**if** **(\***s1 **==** '\0' **&&** env\_get\_char**(**i2**-**1**)** **==** '='**)**

**return(**i2**);**

**return(-**1**);**

**}**

//获取环境变量内存区对应index的指针

//找到了val的值，也就是ipaddr在default\_environment中所处的位置，

//即可以返回ipaddr字符串的地址。前面说为什么会返回字符型指针？

//这里给出了答案。注意：这里返回的是指针，是192.168.1.230的地址

uchar **\***env\_get\_addr **(**int index**)**

**{**

**if** **(**gd**->**env\_valid**)** **{**

**return** **(** **((**uchar **\*)(**gd**->**env\_addr **+** index**))** **);**

**}** **else** **{**

**return** **(&**default\_environment**[**index**]);**

**}**

**}**

// string\_to\_ip函数将字符串192.168.1.230转化成整数，

//关键的地方是for (addr=0, i=0; i<4; ++i)，这里可以看出进行了四次转化，分别是四个段。

//但是函数simple\_strtoul是如何区分"."而分成四个段？

//这是靠simple\_strtoul中的while (isxdigit(\*cp) && (value =

isdigit(\*cp) ? \*cp-'0' : (islower(\*cp) ? toupper(\*cp) : \*cp)-'A'+10) < base)实现的,

//因为当while遇到"."时，判定条件即为0，就不再判定而往下走了。

//四个段都操作完成，就出现了addr = (((((192 << 8) | 168) << 8) | 1) << 8 ) | 230，

//unsigned long long simple\_strtoull(const char \*cp, char \*\*endp, unsigned int base)

//cp指向字符串的开始，endp指向分析的字符串末尾的位置，base为要用的基数（进制数）

IPaddr\_t string\_to\_ip**(**char **\***s**)**

**{**

IPaddr\_t addr**;**

char **\***e**;**

int i**;**

**if** **(**s **==** **NULL)**

**return(**0**);**

**for** **(**addr**=**0**,** i**=**0**;** i**<**4**;** **++**i**)** **{**

ulong val **=** s **?** simple\_strtoul**(**s**,** **&**e**,** 10**)** **:** 0**;**

addr **<<=** 8**;**

addr **|=** **(**val **&** 0xFF**);**

**if** **(**s**)** **{**

s **=** **(\***e**)** **?** e**+**1 **:** e**;**

**}**

**}**

**return** **(**htonl**(**addr**));**

**}**

#### 2.2.5 设备列表初始化

int devices\_init **(**void**)**

**{**

//创建设备列表

devlist **=** ListCreate **(sizeof** **(**device\_t**));**

**if** **(**devlist **==** **NULL)** **{**

eputs **(**"Cannot initialize the list of devices!\n"**);**

**return** **-**1**;**

**}**

drv\_system\_init **();**

**return** **(**0**);**

**}**

//创建串口设备并添加到设备列表中

static void drv\_system\_init **(**void**)**

**{**

device\_t dev**;**

memset **(&**dev**,** 0**,** **sizeof** **(**dev**));**

strcpy **(**dev**.**name**,** "serial"**);**

dev**.**flags **=** DEV\_FLAGS\_OUTPUT **|** DEV\_FLAGS\_INPUT **|** DEV\_FLAGS\_SYSTEM**;**

dev**.**putc **=** serial\_putc**;**

dev**.**puts **=** serial\_puts**;**

dev**.**getc **=** serial\_getc**;**

dev**.**tstc **=** serial\_tstc**;**

device\_register **(&**dev**);**

**}**

#### 2.2.6 函数跳转表初始化

//初始化跳转表gd->jt，该跳转表是一个函数指针数组，它定义了U-Boot中基本的常用函数库。

void jumptable\_init **(**void**)**

**{**

int i**;**

gd**->**jt **=** **(**void **\*\*)** malloc **(**XF\_MAX **\*** **sizeof** **(**void **\*));**

**for** **(**i **=** 0**;** i **<** XF\_MAX**;** i**++)**

gd**->**jt**[**i**]** **=** **(**void **\*)** dummy**;**

//XF\_get\_version=0,下面一次类推，

gd**->**jt**[**XF\_get\_version**]** **=** **(**void **\*)** get\_version**;**

gd**->**jt**[**XF\_malloc**]** **=** **(**void **\*)** malloc**;**

gd**->**jt**[**XF\_free**]** **=** **(**void **\*)** free**;**

gd**->**jt**[**XF\_getenv**]** **=** **(**void **\*)** getenv**;**

gd**->**jt**[**XF\_setenv**]** **=** **(**void **\*)** setenv**;**

gd**->**jt**[**XF\_get\_timer**]** **=** **(**void **\*)** get\_timer**;**

gd**->**jt**[**XF\_simple\_strtoul**]** **=** **(**void **\*)** simple\_strtoul**;**

gd**->**jt**[**XF\_udelay**]** **=** **(**void **\*)** udelay**;**

**}**

enum **{**

#define EXPORT\_FUNC(x) XF\_ ## x ,

#include <\_exports.h>

#undef EXPORT\_FUNC

XF\_MAX

**};**

//上面的枚举等效为

enum **{**

#define EXPORT\_FUNC(x) XF\_ ## x ,

EXPORT\_FUNC**(**get\_version**)**

EXPORT\_FUNC**(**getc**)**

EXPORT\_FUNC**(**tstc**)**

EXPORT\_FUNC**(**putc**)**

EXPORT\_FUNC**(**puts**)**

EXPORT\_FUNC**(**printf**)**

EXPORT\_FUNC**(**install\_hdlr**)**

EXPORT\_FUNC**(**free\_hdlr**)**

EXPORT\_FUNC**(**malloc**)**

EXPORT\_FUNC**(**free**)**

EXPORT\_FUNC**(**udelay**)**

EXPORT\_FUNC**(**get\_timer**)**

EXPORT\_FUNC**(**vprintf**)**

EXPORT\_FUNC**(**do\_reset**)**

EXPORT\_FUNC**(**getenv**)**

EXPORT\_FUNC**(**setenv**)**

EXPORT\_FUNC**(**simple\_strtoul**)**

#undef EXPORT\_FUNC

XF\_MAX

**};**

//再次展开等效为

enum **{**

XF\_get\_version

XF\_getc

XF\_tstc

XF\_putc

XF\_puts

XF\_printf

XF\_install\_hdlr

XF\_free\_hdlr

XF\_malloc

XF\_free

XF\_udelay

XF\_get\_timer

**......**

XF\_MAX

**}**

#### 2.2.7 控制台初始化

int console\_init\_r **(**void**)**

**{**

device\_t **\***inputdev **=** **NULL,** **\***outputdev **=** **NULL;**

int i**,** items **=** ListNumItems **(**devlist**);**

//扫描设备寻找输入和输出设备

**for** **(**i **=** 1**;**

**(**i **<=** items**)** **&&** **((**inputdev **==** **NULL)** **||** **(**outputdev **==** **NULL));**

i**++**

**)** **{**

device\_t **\***dev **=** ListGetPtrToItem **(**devlist**,** i**);**

//这里串口设备赋给inputdev

**if** **((**dev**->**flags **&** DEV\_FLAGS\_INPUT**)** **&&** **(**inputdev **==** **NULL))** **{**

inputdev **=** dev**;**

**}**

//这里串口设备赋给outputdev

**if** **((**dev**->**flags **&** DEV\_FLAGS\_OUTPUT**)** **&&** **(**outputdev **==** **NULL))** **{**

outputdev **=** dev**;**

**}**

**}**

//设置stdout stdout对应的设备

**if** **(**outputdev **!=** **NULL)** **{**

console\_setfile **(**stdout**,** outputdev**);**

console\_setfile **(**stdout**,** outputdev**);**

**}**

//设置stdin对应的设备

**if** **(**inputdev **!=** **NULL)** **{**

console\_setfile **(**stdin**,** inputdev**);**

**}**

gd**->**flags **|=** GD\_FLG\_DEVINIT**;** /\* device initialization completed \*/

//打印输入输出设备相关信息

#ifndef CFG\_CONSOLE\_INFO\_QUIET

/\* Print information \*/

puts **(**"In: "**);**

**if** **(**stdio\_devices**[**stdin**]** **==** **NULL)** **{**

puts **(**"No input devices available!\n"**);**

**}** **else** **{**

printf **(**"%s\n"**,** stdio\_devices**[**stdin**]->**name**);**

**}**

puts **(**"Out: "**);**

**if** **(**stdio\_devices**[**stdout**]** **==** **NULL)** **{**

puts **(**"No output devices available!\n"**);**

**}** **else** **{**

printf **(**"%s\n"**,** stdio\_devices**[**stdout**]->**name**);**

**}**

puts **(**"Err: "**);**

**if** **(**stdio\_devices**[**stderr**]** **==** **NULL)** **{**

puts **(**"No error devices available!\n"**);**

**}** **else** **{**

printf **(**"%s\n"**,** stdio\_devices**[**stderr**]->**name**);**

**}**

#endif /\* CFG\_CONSOLE\_INFO\_QUIET \*/

/\* Setting environment variables \*/

**for** **(**i **=** 0**;** i **<** 3**;** i**++)** **{**

setenv **(**stdio\_names**[**i**],** stdio\_devices**[**i**]->**name**);**

**}**

**return** **(**0**);**

**}**

static int console\_setfile **(**int file**,** device\_t **\*** dev**)**

**{**

int error **=** 0**;**

**if** **(**dev **==** **NULL)**

**return** **-**1**;**

**switch** **(**file**)** **{**

**case** stdin**:**

**case** stdout**:**

**case** stderr**:**

//启动设备，串口设备没有启动函数

**if** **(**dev**->**start**)** **{**

error **=** dev**->**start **();**

**if** **(**error **<** 0**)**

**break;**

**}**

//设置stdio对应的设备

stdio\_devices**[**file**]** **=** dev**;**

//更新函数跳转表中对应的函数

**switch** **(**file**)** **{**

**case** stdin**:**

gd**->**jt**[**XF\_getc**]** **=** dev**->**getc**;**

gd**->**jt**[**XF\_tstc**]** **=** dev**->**tstc**;**

**break;**

**case** stdout**:**

gd**->**jt**[**XF\_putc**]** **=** dev**->**putc**;**

gd**->**jt**[**XF\_puts**]** **=** dev**->**puts**;**

gd**->**jt**[**XF\_printf**]** **=** printf**;**

**break;**

**}**

**break;**

**default:** /\* Invalid file ID \*/

error **=** **-**1**;**

**}**

**return** error**;**

**}**

#### 2.2.8 mtd初始化

为了更好的管理nand，采用了mtd，例如启动时候执行的nand read.jffs2 0x30007FC0 kernel命令，其中kernel代表了nand的一些信息，这些信息包含了存放kernel的nand的偏移地址和长度。

#define MTDIDS\_DEFAULT "nand0=nandflash0"

#define MTDPARTS\_DEFAULT "mtdparts=nandflash0:256k@0(bootloader)," \

"128k(params)," \

"2m(kernel)," \

"-(root)"

int mtdparts\_init**(**void**)**

**{**

static int initialized **=** 0**;**

const char **\***ids**,** **\***parts**;**

const char **\***current\_partition**;**

int ids\_changed**;**

char tmp\_ep**[**PARTITION\_MAXLEN**];**

**if** **(!**initialized**)** **{**

//mtdids链表初始化

INIT\_LIST\_HEAD**(&**mtdids**);**

//device链表初始化

INIT\_LIST\_HEAD**(&**devices**);**

memset**(**last\_ids**,** 0**,** MTDIDS\_MAXLEN**);**

memset**(**last\_parts**,** 0**,** MTDPARTS\_MAXLEN**);**

memset**(**last\_partition**,** 0**,** PARTITION\_MAXLEN**);**

initialized **=** 1**;**

**}**

//ids = "nand0=nandflash0"

ids **=** getenv**(**"mtdids"**);**

//parts = "nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)"

parts **=** getenv**(**"mtdparts"**);**

//current\_partitionw为空

current\_partition **=** getenv**(**"partition"**);**

tmp\_ep**[**0**]** **=** '\0'**;**

**if** **(**current\_partition**)**

strncpy**(**tmp\_ep**,** current\_partition**,** PARTITION\_MAXLEN**);**

**if** **(!**ids**)** **{**

**if** **(**mtdids\_default**)** **{**

ids **=** mtdids\_default**;**

setenv**(**"mtdids"**,** **(**char **\*)**ids**);**

**}** **else** **{**

printf**(**"mtdids not defined, no default present\n"**);**

**return** 1**;**

**}**

**}**

**if** **(**strlen**(**ids**)** **>** MTDIDS\_MAXLEN **-** 1**)** **{**

printf**(**"mtdids too long (> %d)\n"**,** MTDIDS\_MAXLEN**);**

**return** 1**;**

**}**

**if** **(!**parts**)**

printf**(**"mtdparts variable not set, see 'help mtdparts'\n"**);**

**if** **(**parts **&&** **(**strlen**(**parts**)** **>** MTDPARTS\_MAXLEN **-** 1**))** **{**

printf**(**"mtdparts too long (> %d)\n"**,** MTDPARTS\_MAXLEN**);**

**return** 1**;**

**}**

/\* check if we have already parsed those mtdids \*/

**if** **((**last\_ids**[**0**]** **!=** '\0'**)** **&&** **(**strcmp**(**last\_ids**,** ids**)** **==** 0**))** **{**

ids\_changed **=** 0**;**

**}**

**else**

**{**

ids\_changed **=** 1**;**

//分析nand0=nandflash0，并将相关信息添加到mtdids链表

**if** **(**parse\_mtdids**(**ids**)** **!=** 0**)** **{**

devices\_init**();**

**return** 1**;**

**}**

strncpy**(**last\_ids**,** ids**,** MTDIDS\_MAXLEN**);**

**}**

//分析"nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)"

**if** **(**parts **&&** **((**last\_parts**[**0**]** **==** '\0'**)** **||** **((**strcmp**(**last\_parts**,** parts**)** **!=** 0**))** **||** ids\_changed**))** **{**

**if** **(**parse\_mtdparts**(**parts**)** **!=** 0**)**

**return** 1**;**

**if** **(**list\_empty**(&**devices**))** **{**

printf**(**"mtdparts\_init: no valid partitions\n"**);**

**return** 1**;**

**}**

strncpy**(**last\_parts**,** parts**,** MTDPARTS\_MAXLEN**);**

/\* reset first partition from first dev from the list as current \*/

current\_dev **=** list\_entry**(**devices**.**next**,** struct mtd\_device**,** link**);**

current\_partnum **=** 0**;**

current\_save**();**

DEBUGF**(**"mtdparts\_init: current\_dev = %s%d, current\_partnum = %d\n"**,**

MTD\_DEV\_TYPE**(**current\_dev**->**id**->**type**),**

current\_dev**->**id**->**num**,** current\_partnum**);**

**}**

/\* mtdparts variable was reset to NULL, delete all devices/partitions \*/

**if** **(!**parts **&&** **(**last\_parts**[**0**]** **!=** '\0'**))**

**return** devices\_init**();**

/\* do not process current partition if mtdparts variable is null \*/

**if** **(!**parts**)**

**return** 0**;**

/\* is current partition set in environment? if so, use it \*/

**if** **((**tmp\_ep**[**0**]** **!=** '\0'**)** **&&** **(**strcmp**(**tmp\_ep**,** last\_partition**)** **!=** 0**))** **{**

struct part\_info **\***p**;**

struct mtd\_device **\***cdev**;**

u8 pnum**;**

DEBUGF**(**"--- getting current partition: %s\n"**,** tmp\_ep**);**

**if** **(**find\_dev\_and\_part**(**tmp\_ep**,** **&**cdev**,** **&**pnum**,** **&**p**)** **==** 0**)** **{**

current\_dev **=** cdev**;**

current\_partnum **=** pnum**;**

current\_save**();**

**}**

**}** **else** **if** **(**getenv**(**"partition"**)** **==** **NULL)** **{**

DEBUGF**(**"no partition variable set, setting...\n"**);**

current\_save**();**

**}**

**return** 0**;**

**}**

static int parse\_mtdids**(**const char **\***const ids**)**

**{**

const char **\***p **=** ids**;**

const char **\***mtd\_id**;**

int mtd\_id\_len**;**

struct mtdids **\***id**;**

struct list\_head **\***entry**,** **\***n**;**

struct mtdids **\***id\_tmp**;**

u8 type**,** num**;**

u32 size**;**

int ret **=** 1**;**

//删除mtdids链表中的内容

list\_for\_each\_safe**(**entry**,** n**,** **&**mtdids**)** **{**

id\_tmp **=** list\_entry**(**entry**,** struct mtdids**,** link**);**

list\_del**(**entry**);**

free**(**id\_tmp**);**

**}**

last\_ids**[**0**]** **=** '\0'**;**

//重新初始化mtdids链表

INIT\_LIST\_HEAD**(&**mtdids**);**

//分析字符串nand0=nandflash0

**while(**p **&&** **(\***p **!=** '\0'**))** **{**

ret **=** 1**;**

//分析存储设备类型是nor还是nand，设备号

**if** **(**id\_parse**(**p**,** **&**p**,** **&**type**,** **&**num**)** **!=** 0**)**

**break;**

//判断是否有=

**if** **(\***p **!=** '='**)** **{**

printf**(**"mtdids: incorrect <dev-num>\n"**);**

**break;**

**}**

p**++;**

//从nand\_info全局变量中获取对应num号的nand的size

**if** **(**device\_validate**(**type**,** num**,** **&**size**)** **!=** 0**)**

**return** 1**;**

//mtd\_id = "nandflash0"

mtd\_id **=** p**;**

**if** **((**p **=** strchr**(**mtd\_id**,** ','**))** **!=** **NULL)** **{**

mtd\_id\_len **=** p **-** mtd\_id **+** 1**;**

p**++;**

**}** **else** **{**

//分支运行到这

mtd\_id\_len **=** strlen**(**mtd\_id**)** **+** 1**;**

**}**

**if** **(**mtd\_id\_len **==** 0**)** **{**

printf**(**"mtdids: no <mtd-id> identifier\n"**);**

**break;**

**}**

//轮询列表检查是否存在type和num一样的设备信息

int double\_entry **=** 0**;**

list\_for\_each**(**entry**,** **&**mtdids**)** **{**

//list\_entry从一个结构的成员指针找到其容器的指针。

id\_tmp **=** list\_entry**(**entry**,** struct mtdids**,** link**);**

**if** **((**id\_tmp**->**type **==** type**)** **&&** **(**id\_tmp**->**num **==** num**))** **{**

double\_entry **=** 1**;**

**break;**

**}**

**}**

**if** **(**double\_entry**)** **{**

printf**(**"device id %s%d redefined, please correct mtdids variable\n"**,**

MTD\_DEV\_TYPE**(**type**),** num**);**

**break;**

**}**

//如果链表中没有重复设备，将该设备添加到mtdids链表

**if** **(!(**id **=** **(**struct mtdids **\*)**malloc**(sizeof(**struct mtdids**)** **+** mtd\_id\_len**)))** **{**

printf**(**"out of memory\n"**);**

**break;**

**}**

memset**(**id**,** 0**,** **sizeof(**struct mtdids**)** **+** mtd\_id\_len**);**

id**->**num **=** num**;**

id**->**type **=** type**;**

id**->**size **=** size**;**

id**->**mtd\_id **=** **(**char **\*)(**id **+** 1**);**

strncpy**(**id**->**mtd\_id**,** mtd\_id**,** mtd\_id\_len **-** 1**);**

id**->**mtd\_id**[**mtd\_id\_len **-** 1**]** **=** '\0'**;**

INIT\_LIST\_HEAD**(&**id**->**link**);**

list\_add\_tail**(&**id**->**link**,** **&**mtdids**);**

ret **=** 0**;**

**}**

**if** **(**ret **==** 1**)** **{**

/\* clean mtdids list and free allocated memory \*/

list\_for\_each\_safe**(**entry**,** n**,** **&**mtdids**)** **{**

id\_tmp **=** list\_entry**(**entry**,** struct mtdids**,** link**);**

list\_del**(**entry**);**

free**(**id\_tmp**);**

**}**

**return** 1**;**

**}**

**return** 0**;**

**}**

static int parse\_mtdparts**(**const char **\***const mtdparts**)**

**{**

const char **\***p **=** mtdparts**;**

struct mtd\_device **\***dev**;**

int err **=** 1**;**

//清空devices链表的所有内容

**if** **(**devices\_init**()** **!=** 0**)** **{**

printf**(**"could not initialise device list\n"**);**

**return** err**;**

**}**

///分析"mtdparts=nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)"

p **=** getenv**(**"mtdparts"**);**

**if** **(**strncmp**(**p**,** "mtdparts="**,** 9**)** **!=** 0**)** **{**

printf**(**"mtdparts variable doesn't start with 'mtdparts='\n"**);**

**return** err**;**

**}**

p **+=** 9**;**

//分析"nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)"

**while** **(**p **&&** **(\***p **!=** '\0'**))** **{**

err **=** 1**;**

//分析device，并建立分区信息

**if** **((**device\_parse**(**p**,** **&**p**,** **&**dev**)** **!=** 0**)** **||** **(!**dev**))**

**break;**

DEBUGF**(**"+ device: %s\t%d\t%s\n"**,** MTD\_DEV\_TYPE**(**dev**->**id**->**type**),**

dev**->**id**->**num**,** dev**->**id**->**mtd\_id**);**

//检查该device是否存在devices链表中

**if** **(**device\_find**(**dev**->**id**->**type**,** dev**->**id**->**num**)** **!=** **NULL)** **{**

printf**(**"device %s%d redefined, please correct mtdparts variable\n"**,**

MTD\_DEV\_TYPE**(**dev**->**id**->**type**),** dev**->**id**->**num**);**

**break;**

**}**

//添加到devices链表末尾

list\_add\_tail**(&**dev**->**link**,** **&**devices**);**

err **=** 0**;**

**}**

**if** **(**err **==** 1**)** **{**

device\_delall**(&**devices**);**

**return** 1**;**

**}**

**return** 0**;**

**}**

static int device\_parse**(**const char **\***const mtd\_dev**,** const char **\*\***ret**,** struct mtd\_device **\*\***retdev**)**

**{**

struct mtd\_device **\***dev**;**

struct part\_info **\***part**;**

struct mtdids **\***id**;**

const char **\***mtd\_id**;**

unsigned int mtd\_id\_len**;**

const char **\***p**,** **\***pend**;**

LIST\_HEAD**(**tmp\_list**);**

struct list\_head **\***entry**,** **\***n**;**

u16 num\_parts**;**

u32 offset**;**

int err **=** 1**;**

p **=** mtd\_dev**;**

**\***retdev **=** **NULL;**

**\***ret **=** **NULL;**

//mtd\_id="nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)"

mtd\_id **=** p**;**

**if** **(!(**p **=** strchr**(**mtd\_id**,** ':'**)))** **{**

printf**(**"no <mtd-id> identifier\n"**);**

**return** 1**;**

**}**

//计算nandflash0长度

mtd\_id\_len **=** p **-** mtd\_id **+** 1**;**

p**++;**

//从mtdids链表中查找是否有nandflash0存在

**if** **((**id **=** id\_find\_by\_mtd\_id**(**mtd\_id**,** mtd\_id\_len **-** 1**))** **==** **NULL)** **{**

printf**(**"invalid mtd device '%.\*s'\n"**,** mtd\_id\_len **-** 1**,** mtd\_id**);**

**return** 1**;**

**}**

//判断是否有";"

pend **=** strchr**(**p**,** ';'**);**

DEBUGF**(**"parsing partitions %.\*s\n"**,** **(**pend **?** pend **-** p **:** strlen**(**p**)),** p**);**

num\_parts **=** 0**;**

offset **=** 0**;**

//从devices链表查找是否存在对应的设备信息

**if** **((**dev **=** device\_find**(**id**->**type**,** id**->**num**))** **!=** **NULL)** **{**

//获取parts链表最后一个part的信息

part **=** list\_entry**(**dev**->**parts**.**prev**,** struct part\_info**,** link**);**

//计算出当前device的最大长度

offset **=** part**->**offset **+** part**->**size**;**

**}**

**while** **(**p **&&** **(\***p **!=** '\0'**)** **&&** **(\***p **!=** ';'**))** **{**

err **=** 1**;**

//分析分区信息，并将分区信息存放到part变量内

**if** **((**part\_parse**(**p**,** **&**p**,** **&**part**)** **!=** 0**)** **||** **(!**part**))**

**break;**

/\* calculate offset when not specified \*/

**if** **(**part**->**offset **==** OFFSET\_NOT\_SPECIFIED**)**

part**->**offset **=** offset**;**

**else**

offset **=** part**->**offset**;**

//检查分区地址是否块对齐

**if** **(**part\_validate**(**id**,** part**)** **!=** 0**)**

**break;**

offset **+=** part**->**size**;**

//将该分区的part变量添加到tmp\_list链表

list\_add\_tail**(&**part**->**link**,** **&**tmp\_list**);**

num\_parts**++;**

err **=** 0**;**

**}**

**if** **(**err **==** 1**)** **{**

part\_delall**(&**tmp\_list**);**

**return** 1**;**

**}**

**if** **(**num\_parts **==** 0**)** **{**

printf**(**"no partitions for device %s%d (%s)\n"**,**

MTD\_DEV\_TYPE**(**id**->**type**),** id**->**num**,** id**->**mtd\_id**);**

**return** 1**;**

**}**

DEBUGF**(**"\ntotal partitions: %d\n"**,** num\_parts**);**

//检查是否有下一个device的存在

**if** **(**p**)** **{**

**if** **(\***p **==** ';'**)** **{**

**\***ret **=** **++**p**;**

**}** **else** **if** **(\***p **==** '\0'**)** **{**

**\***ret **=** p**;**

**}** **else** **{**

printf**(**"unexpected character '%c' at the end of device\n"**,** **\***p**);**

**\***ret **=** **NULL;**

**return** 1**;**

**}**

**}**

//创建mtd\_device结构体

**if** **((**dev **=** **(**struct mtd\_device **\*)**malloc**(sizeof(**struct mtd\_device**)))** **==** **NULL)** **{**

printf**(**"out of memory\n"**);**

**return** 1**;**

**}**

memset**(**dev**,** 0**,** **sizeof(**struct mtd\_device**));**

dev**->**id **=** id**;**

dev**->**num\_parts **=** 0**;** /\* part\_sort\_add increments num\_parts \*/

INIT\_LIST\_HEAD**(&**dev**->**parts**);**

INIT\_LIST\_HEAD**(&**dev**->**link**);**

//轮询tmp\_list链表，取出part\_info，并将其添加到新创建的mtd\_device链表中

list\_for\_each\_safe**(**entry**,** n**,** **&**tmp\_list**)** **{**

part **=** list\_entry**(**entry**,** struct part\_info**,** link**);**

list\_del**(**entry**);**

//按offset升序添加

**if** **(**part\_sort\_add**(**dev**,** part**)** **!=** 0**)** **{**

device\_del**(**dev**);**

**return** 1**;**

**}**

**}**

**\***retdev **=** dev**;**

DEBUGF**(**"===\n\n"**);**

**return** 0**;**

**}**

//分析256k@0(bootloader),128k(params),2m(kernel),-(root)

static int part\_parse**(**const char **\***const partdef**,** const char **\*\***ret**,** struct part\_info **\*\***retpart**)**

**{**

struct part\_info **\***part**;**

unsigned long size**;**

unsigned long offset**;**

const char **\***name**;**

int name\_len**;**

unsigned int mask\_flags**;**

const char **\***p**;**

p **=** partdef**;**

**\***retpart **=** **NULL;**

**\***ret **=** **NULL;**

/\* fetch the partition size \*/

**if** **(\***p **==** '-'**)** **{**

//-代表最后一个分区，大小为nand的剩余大小

DEBUGF**(**"'-': remaining size assigned\n"**);**

size **=** SIZE\_REMAINING**;**

p**++;**

**}** **else** **{**

//否则提取出256k，128k，2m等数据

size **=** memsize\_parse**(**p**,** **&**p**);**

**if** **(**size **<** MIN\_PART\_SIZE**)** **{**

printf**(**"partition size too small (%lx)\n"**,** size**);**

**return** 1**;**

**}**

**}**

offset **=** OFFSET\_NOT\_SPECIFIED**;**

//加入看到@标志，表示后面的数值为起始地址

**if** **(\***p **==** '@'**)** **{**

p**++;**

offset **=** memsize\_parse**(**p**,** **&**p**);**

**}**

//获取分区名字

**if** **(\***p **==** '('**)** **{**

name **=** **++**p**;**

**if** **((**p **=** strchr**(**name**,** ')'**))** **==** **NULL)** **{**

printf**(**"no closing ) found in partition name\n"**);**

**return** 1**;**

**}**

name\_len **=** p **-** name **+** 1**;**

**if** **((**name\_len **-** 1**)** **==** 0**)** **{**

printf**(**"empty partition name\n"**);**

**return** 1**;**

**}**

p**++;**

**}** **else** **{**

/\* 0x00000000@0x00000000 \*/

name\_len **=** 22**;**

name **=** **NULL;**

**}**

/\* test for options \*/

mask\_flags **=** 0**;**

**if** **(**strncmp**(**p**,** "ro"**,** 2**)** **==** 0**)** **{**

mask\_flags **|=** MTD\_WRITEABLE\_CMD**;**

p **+=** 2**;**

**}**

//如果出现类似-(root)分区后面还有其他分区，分析结束，否则继续分析

**if** **(\***p **==** ','**)** **{**

**if** **(**size **==** SIZE\_REMAINING**)** **{**

**\***ret **=** **NULL;**

printf**(**"no partitions allowed after a fill-up partition\n"**);**

**return** 1**;**

**}**

**\***ret **=** **++**p**;**

**}** **else** **if** **((\***p **==** ';'**)** **||** **(\***p **==** '\0'**))** **{**

**\***ret **=** p**;**

**}** **else** **{**

printf**(**"unexpected character '%c' at the end of partition\n"**,** **\***p**);**

**\***ret **=** **NULL;**

**return** 1**;**

**}**

//申请part\_info结构体内存，并填充该分区相关信息

part **=** **(**struct part\_info **\*)**malloc**(sizeof(**struct part\_info**)** **+** name\_len**);**

**if** **(!**part**)** **{**

printf**(**"out of memory\n"**);**

**return** 1**;**

**}**

memset**(**part**,** 0**,** **sizeof(**struct part\_info**)** **+** name\_len**);**

part**->**size **=** size**;**

part**->**offset **=** offset**;**

part**->**mask\_flags **=** mask\_flags**;**

part**->**name **=** **(**char **\*)(**part **+** 1**);**

**if** **(**name**)** **{**

/\* copy user provided name \*/

strncpy**(**part**->**name**,** name**,** name\_len **-** 1**);**

part**->**auto\_name **=** 0**;**

**}** **else** **{**

/\* auto generated name in form of size@offset \*/

sprintf**(**part**->**name**,** "0x%08lx@0x%08lx"**,** size**,** offset**);**

part**->**auto\_name **=** 1**;**

**}**

part**->**name**[**name\_len **-** 1**]** **=** '\0'**;**

INIT\_LIST\_HEAD**(&**part**->**link**);**

**\***retpart **=** part**;**

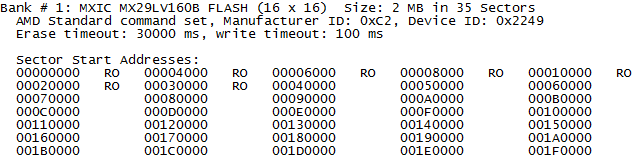
**return** 0**;**

**}**

## 3.uboot常用命令分析

### 3.1 norflash操作指令

#### 3.1.1 flinfo指令



do\_flinfo的代码比较简单，仅仅从flash\_info中读取对应信息并打印出来

int do\_flinfo **(** cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

ulong bank**;**

**if** **(**argc **==** 1**)** **{** /\* print info for all FLASH banks \*/

**for** **(**bank**=**0**;** bank **<**CFG\_MAX\_FLASH\_BANKS**;** **++**bank**)** **{**

printf **(**"\nBank # %ld: "**,** bank**+**1**);**

flash\_print\_info **(&**flash\_info**[**bank**]);**

**}**

**return** 0**;**

**}**

**return** 0**;**

**}**

void flash\_print\_info **(**flash\_info\_t **\*** info**)**

**{**

int i**;**

**if** **(**info**->**flash\_id **!=** FLASH\_MAN\_CFI**)** **{**

puts **(**"missing or unknown FLASH type\n"**);**

**return;**

**}**

printf **(**"%s FLASH (%d x %d)"**,**

info**->**name**,**

**(**info**->**portwidth **<<** 3**),** **(**info**->**chipwidth **<<** 3**));**

**if** **(**info**->**size **<** 1024**\***1024**)**

printf **(**" Size: %ld kB in %d Sectors\n"**,**

info**->**size **>>** 10**,** info**->**sector\_count**);**

**else**

printf **(**" Size: %ld MB in %d Sectors\n"**,**

info**->**size **>>** 20**,** info**->**sector\_count**);**

printf **(**" "**);**

**switch** **(**info**->**vendor**)** **{**

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

printf **(**"AMD Standard"**);**

**break;**

**}**

printf **(**" command set, Manufacturer ID: 0x%02X, Device ID: 0x%02X"**,**

info**->**manufacturer\_id**,** info**->**device\_id**);**

**if** **(**info**->**device\_id **==** 0x7E**)** **{**

printf**(**"%04X"**,** info**->**device\_id2**);**

**}**

printf **(**"\n Erase timeout: %ld ms, write timeout: %ld ms\n"**,**

info**->**erase\_blk\_tout**,**

info**->**write\_tout**);**

**if** **(**info**->**buffer\_size **>** 1**)** **{**

printf **(**" Buffer write timeout: %ld ms, "

"buffer size: %d bytes\n"**,**

info**->**buffer\_write\_tout**,**

info**->**buffer\_size**);**

**}**

puts **(**"\n Sector Start Addresses:"**);**

**for** **(**i **=** 0**;** i **<** info**->**sector\_count**;** **++**i**)** **{**

**if** **((**i **%** 5**)** **==** 0**)**

printf **(**"\n"**);**

printf **(**" %08lX %s "**,**

info**->**start**[**i**],**

info**->**protect**[**i**]** **?** "RO" **:** " "**);**

**}**

putc **(**'\n'**);**

**return;**

**}**

#### 3.1.2 erase指令

norflash的擦除动作都是以section为单位的。操作命令如下。

erase all

erase bank N

erase N**:**SF**[-**SL**]** //erase sectors SF-SL in FLASH bank # N

erase start **+**len

erase start end

int do\_flerase **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

flash\_info\_t **\***info**;**

ulong bank**,** addr\_first**,** addr\_last**;**

int n**,** sect\_first**,** sect\_last**;**

#if (CONFIG\_COMMANDS & CFG\_CMD\_JFFS2) && defined(CONFIG\_JFFS2\_CMDLINE)

struct mtd\_device **\***dev**;**

struct part\_info **\***part**;**

u8 dev\_type**,** dev\_num**,** pnum**;**

#endif

int rcode **=** 0**;**

**if** **(**argc **<** 2**)** **{**

printf **(**"Usage:\n%s\n"**,** cmdtp**->**usage**);**

**return** 1**;**

**}**

//擦除所有的section

**if** **(**strcmp**(**argv**[**1**],** "all"**)** **==** 0**)** **{**

**for** **(**bank**=**1**;** bank**<=**CFG\_MAX\_FLASH\_BANKS**;** **++**bank**)** **{**

printf **(**"Erase Flash Bank # %ld "**,** bank**);**

info **=** **&**flash\_info**[**bank**-**1**];**

rcode **=** flash\_erase **(**info**,** 0**,** info**->**sector\_count**-**1**);**

**}**

**return** rcode**;**

**}**

//分析类似erase N:SF[-SL]命令

**if** **((**n **=** abbrev\_spec**(**argv**[**1**],** **&**info**,** **&**sect\_first**,** **&**sect\_last**))** **!=** 0**)** **{**

**if** **(**n **<** 0**)** **{**

puts **(**"Bad sector specification\n"**);**

**return** 1**;**

**}**

//擦除指定section

rcode **=** flash\_erase**(**info**,** sect\_first**,** sect\_last**);**

**return** rcode**;**

**}**

**if** **(**argc **!=** 3**)** **{**

printf **(**"Usage:\n%s\n"**,** cmdtp**->**usage**);**

**return** 1**;**

**}**

//擦除bank N的所有分区

**if** **(**strcmp**(**argv**[**1**],** "bank"**)** **==** 0**)** **{**

bank **=** simple\_strtoul**(**argv**[**2**],** **NULL,** 16**);**

**if** **((**bank **<** 1**)** **||** **(**bank **>** CFG\_MAX\_FLASH\_BANKS**))** **{**

printf **(**"Only FLASH Banks # 1 ... # %d supported\n"**,**

CFG\_MAX\_FLASH\_BANKS**);**

**return** 1**;**

**}**

printf **(**"Erase Flash Bank # %ld "**,** bank**);**

info **=** **&**flash\_info**[**bank**-**1**];**

rcode **=** flash\_erase **(**info**,** 0**,** info**->**sector\_count**-**1**);**

**return** rcode**;**

**}**

//分析erase start end指令

**if** **(**addr\_spec**(**argv**[**1**],** argv**[**2**],** **&**addr\_first**,** **&**addr\_last**)** **<** 0**){**

printf **(**"Bad address format\n"**);**

**return** 1**;**

**}**

**if** **(**addr\_first **>=** addr\_last**)** **{**

printf **(**"Usage:\n%s\n"**,** cmdtp**->**usage**);**

**return** 1**;**

**}**

rcode **=** flash\_sect\_erase**(**addr\_first**,** addr\_last**);**

**return** rcode**;**

**}**

//norflash擦除函数，以section为单位

int flash\_erase **(**flash\_info\_t **\*** info**,** int s\_first**,** int s\_last**)**

**{**

int rcode **=** 0**;**

int prot**;**

flash\_sect\_t sect**;**

**if** **(**info**->**flash\_id **!=** FLASH\_MAN\_CFI**)** **{**

puts **(**"Can't erase unknown flash type - aborted\n"**);**

**return** 1**;**

**}**

**if** **((**s\_first **<** 0**)** **||** **(**s\_first **>** s\_last**))** **{**

puts **(**"- no sectors to erase\n"**);**

**return** 1**;**

**}**

prot **=** 0**;**

**for** **(**sect **=** s\_first**;** sect **<=** s\_last**;** **++**sect**)** **{**

//判断所擦除的块是否有写保护

**if** **(**info**->**protect**[**sect**])** **{**

prot**++;**

**}**

**}**

**if** **(**prot**)** **{**

printf **(**"- Warning: %d protected sectors will not be erased!\n"**,**

prot**);**

**}** **else** **{**

putc **(**'\n'**);**

**}**

**for** **(**sect **=** s\_first**;** sect **<=** s\_last**;** sect**++)** **{**

**if** **(**info**->**protect**[**sect**]** **==** 0**)** **{** /\* not protected \*/

**switch** **(**info**->**vendor**)** **{**

//发出section擦除指令，具体指令见2.2.2

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

**case** CFI\_CMDSET\_AMD\_EXTENDED**:**

flash\_unlock\_seq **(**info**,** 0**);**

flash\_write\_cmd **(**info**,** 0**,**

info**->**addr\_unlock1**,**

AMD\_CMD\_ERASE\_START**);**

flash\_unlock\_seq **(**info**,** 0**);**

flash\_write\_cmd **(**info**,** sect**,** 0**,**

AMD\_CMD\_ERASE\_SECTOR**);**

**break;**

**}**

//判断擦除是否完成

**if** **(**flash\_full\_status\_check

**(**info**,** sect**,** info**->**erase\_blk\_tout**,** "erase"**))** **{**

rcode **=** 1**;**

**}** **else**

putc **(**'.'**);**

**}**

**}**

puts **(**" done\n"**);**

**return** rcode**;**

**}**

static int flash\_full\_status\_check **(**flash\_info\_t **\*** info**,** flash\_sect\_t sector**,**

ulong tout**,** char **\***prompt**)**

**{**

int retcode**;**

retcode **=** flash\_status\_check **(**info**,** sector**,** tout**,** prompt**);**

**return** retcode**;**

**}**

//检查操作是否完成

static int flash\_status\_check **(**flash\_info\_t **\*** info**,** flash\_sect\_t sector**,**

ulong tout**,** char **\***prompt**)**

**{**

ulong start**;**

#if CFG\_HZ != 1000

tout **\*=** CFG\_HZ**/**1000**;**

#endif

/\* Wait for command completion \*/

start **=** get\_timer **(**0**);**

**while** **(**flash\_is\_busy **(**info**,** sector**))** **{**

**if** **(**get\_timer **(**start**)** **>** tout**)** **{**

flash\_write\_cmd **(**info**,** sector**,** 0**,** info**->**cmd\_reset**);**

**return** ERR\_TIMOUT**;**

**}**

udelay **(**1**);** /\* also triggers watchdog \*/

**}**

**return** ERR\_OK**;**

**}**

//判断操作是否完成

static int flash\_is\_busy **(**flash\_info\_t **\*** info**,** flash\_sect\_t sect**)**

**{**

int retval**;**

**switch** **(**info**->**vendor**)** **{**

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

**case** CFI\_CMDSET\_AMD\_EXTENDED**:**

#ifdef CONFIG\_FLASH\_CFI\_LEGACY

**case** CFI\_CMDSET\_AMD\_LEGACY**:**

#endif

retval **=** flash\_toggle **(**info**,** sect**,** 0**,** AMD\_STATUS\_TOGGLE**);**

**break;**

**default:**

retval **=** 0**;**

**}**

debug **(**"flash\_is\_busy: %d\n"**,** retval**);**

**return** retval**;**

**}**

//它的原理是连续两次读取数据总线上的数据，判断数据总线上的第6位数值（DQ6）是否翻转，

//如果没有翻转则正确，否则还要判断第5位（DQ5），以确定是否是因为超时而引起的翻转。

static int flash\_toggle **(**flash\_info\_t **\*** info**,** flash\_sect\_t sect**,**

uint offset**,** uchar cmd**)**

**{**

void **\***addr**;**

cfiword\_t cword**;**

int retval**;**

addr **=** flash\_map **(**info**,** sect**,** offset**);**

flash\_make\_cmd **(**info**,** cmd**,** **&**cword**);**

**switch** **(**info**->**portwidth**)** **{**

**case** FLASH\_CFI\_16BIT**:**

//连续读两次，看两次数据是否一致

retval **=** flash\_read16**(**addr**)** **!=** flash\_read16**(**addr**);**

**break;**

**}**

flash\_unmap**(**info**,** sect**,** offset**,** addr**);**

**return** retval**;**

**}**

//擦除指令范围的地址对应的section

int flash\_sect\_erase **(**ulong addr\_first**,** ulong addr\_last**)**

**{**

flash\_info\_t **\***info**;**

ulong bank**;**

int s\_first**[**CFG\_MAX\_FLASH\_BANKS**],** s\_last**[**CFG\_MAX\_FLASH\_BANKS**];**

int erased **=** 0**;**

int planned**;**

int rcode **=** 0**;**

rcode **=** flash\_fill\_sect\_ranges **(**addr\_first**,** addr\_last**,**

s\_first**,** s\_last**,** **&**planned **);**

**if** **(**planned **&&** **(**rcode **==** 0**))** **{**

**for** **(**bank**=**0**,**info**=&**flash\_info**[**0**];**

**(**bank **<** CFG\_MAX\_FLASH\_BANKS**)** **&&** **(**rcode **==** 0**);**

**++**bank**,** **++**info**)** **{**

**if** **(**s\_first**[**bank**]>=**0**)** **{**

erased **+=** s\_last**[**bank**]** **-** s\_first**[**bank**]** **+** 1**;**

debug **(**"Erase Flash from 0x%08lx to 0x%08lx "

"in Bank # %ld "**,**

info**->**start**[**s\_first**[**bank**]],**

**(**s\_last**[**bank**]** **==** info**->**sector\_count**)** **?**

info**->**start**[**0**]** **+** info**->**size **-** 1**:**

info**->**start**[**s\_last**[**bank**]+**1**]** **-** 1**,**

bank**+**1**);**

rcode **=** flash\_erase **(**info**,** s\_first**[**bank**],** s\_last**[**bank**]);**

**}**

**}**

printf **(**"Erased %d sectors\n"**,** erased**);**

**}** **else** **if** **(**rcode **==** 0**)** **{**

puts **(**"Error: start and/or end address"

" not on sector boundary\n"**);**

rcode **=** 1**;**

**}**

**return** rcode**;**

**}**

//分析类似 erase N:SF[-SL]\n - erase sectors SF-SL in FLASH bank # N\n"命令

static int abbrev\_spec **(**char **\***str**,** flash\_info\_t **\*\*** pinfo**,** int **\***psf**,** int **\***psl**)**

**{**

flash\_info\_t **\***fp**;**

int bank**,** first**,** last**;**

char **\***p**,** **\***ep**;**

**if** **((**p **=** strchr **(**str**,** ':'**))** **==** **NULL)**

**return** 0**;**

**\***p**++** **=** '\0'**;**

//取得bank号N

bank **=** simple\_strtoul **(**str**,** **&**ep**,** 10**);**

**if** **(**ep **==** str **||** **\***ep **!=** '\0' **||**

bank **<** 1 **||** bank **>** CFG\_MAX\_FLASH\_BANKS **||**

**(**fp **=** **&**flash\_info**[**bank **-** 1**])->**flash\_id **==** FLASH\_UNKNOWN**)**

**return** **-**1**;**

//获取-

str **=** p**;**

**if** **((**p **=** strchr **(**str**,** '-'**))** **!=** **NULL)**

**\***p**++** **=** '\0'**;**

//获取起始section

first **=** simple\_strtoul **(**str**,** **&**ep**,** 10**);**

**if** **(**ep **==** str **||** **\***ep **!=** '\0' **||** first **>=** fp**->**sector\_count**)**

**return** **-**1**;**

//获取结束section

**if** **(**p **!=** **NULL)** **{**

last **=** simple\_strtoul **(**p**,** **&**ep**,** 10**);**

**if** **(**ep **==** p **||** **\***ep **!=** '\0' **||**

last **<** first **||** last **>=** fp**->**sector\_count**)**

**return** **-**1**;**

**}** **else** **{**

last **=** first**;**

**}**

**\***pinfo **=** fp**;**

**\***psf **=** first**;**

**\***psl **=** last**;**

**return** 1**;**

**}**

#### 3.1.3 protect指令

protect指令主要对section的protect标志位写0或1，这里不进行分析。

#### 3.1.4 cp指令

由于norflash是内存类器件，但是有点区别，norflash可以直接读，但是不能直接写，下面来分析下norflash的cp指令是如何实现的。

//cp source target count

int do\_mem\_cp **(** cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

ulong addr**,** dest**,** count**;**

int size**;**

**if** **(**argc **!=** 4**)** **{**

printf **(**"Usage:\n%s\n"**,** cmdtp**->**usage**);**

**return** 1**;**

**}**

//判断cp的后缀，包含b w l 字节 字 双字，默认为4字节

**if** **((**size **=** cmd\_get\_data\_size**(**argv**[**0**],** 4**))** **<** 0**)**

**return** 1**;**

//获取起始地址

addr **=** simple\_strtoul**(**argv**[**1**],** **NULL,** 16**);**

addr **+=** base\_address**;**

//获取结束地址

dest **=** simple\_strtoul**(**argv**[**2**],** **NULL,** 16**);**

dest **+=** base\_address**;**

//获取复制长度

count **=** simple\_strtoul**(**argv**[**3**],** **NULL,** 16**);**

**if** **(**count **==** 0**)** **{**

puts **(**"Zero length ???\n"**);**

**return** 1**;**

**}**

#ifndef CFG\_NO\_FLASH

//判断是否要复制到norflash中

**if** **(** **(**addr2info**(**dest**)** **!=** **NULL))** **{**

int rc**;**

puts **(**"Copy to Flash... "**);**

rc **=** flash\_write **((**char **\*)**addr**,** dest**,** count**\***size**);**

**if** **(**rc **!=** 0**)** **{**

flash\_perror **(**rc**);**

**return** **(**1**);**

**}**

puts **(**"done\n"**);**

**return** 0**;**

**}**

#endif

//如果不是赋值到norflash则直接复制

**while** **(**count**--** **>** 0**)** **{**

**if** **(**size **==** 4**)**

**\*((**ulong **\*)**dest**)** **=** **\*((**ulong **\*)**addr**);**

**else** **if** **(**size **==** 2**)**

**\*((**ushort **\*)**dest**)** **=** **\*((**ushort **\*)**addr**);**

**else**

**\*((**u\_char **\*)**dest**)** **=** **\*((**u\_char **\*)**addr**);**

addr **+=** size**;**

dest **+=** size**;**

**}**

**return** 0**;**

**}**

int flash\_write **(**char **\***src**,** ulong addr**,** ulong cnt**)**

**{**

int i**;**

ulong end **=** addr **+** cnt **-** 1**;**

flash\_info\_t **\***info\_first **=** addr2info **(**addr**);**

flash\_info\_t **\***info\_last **=** addr2info **(**end **);**

flash\_info\_t **\***info**;**

**if** **(**cnt **==** 0**)** **{**

**return** **(**ERR\_OK**);**

**}**

**if** **(!**info\_first **||** **!**info\_last**)** **{**

**return** **(**ERR\_INVAL**);**

**}**

//检查所写的区域是否有写保护

**for** **(**info **=** info\_first**;** info **<=** info\_last**;** **++**info**)** **{**

ulong b\_end **=** info**->**start**[**0**]** **+** info**->**size**;** /\* bank end addr \*/

short s\_end **=** info**->**sector\_count **-** 1**;**

**for** **(**i**=**0**;** i**<**info**->**sector\_count**;** **++**i**)** **{**

ulong e\_addr **=** **(**i **==** s\_end**)** **?** b\_end **:** info**->**start**[**i **+** 1**];**

**if** **((**end **>=** info**->**start**[**i**])** **&&** **(**addr **<** e\_addr**)** **&&**

**(**info**->**protect**[**i**]** **!=** 0**)** **)** **{**

**return** **(**ERR\_PROTECTED**);**

**}**

**}**

**}**

//写入数据到addr

**for** **(**info **=** info\_first**;** info **<=** info\_last **&&** cnt**>**0**;** **++**info**)** **{**

ulong len**;**

len **=** info**->**start**[**0**]** **+** info**->**size **-** addr**;**

**if** **(**len **>** cnt**)**

len **=** cnt**;**

**if** **((**i **=** write\_buff**(**info**,** **(**uchar **\*)**src**,** addr**,** len**))** **!=** 0**)** **{**

**return** **(**i**);**

**}**

cnt **-=** len**;**

addr **+=** len**;**

src **+=** len**;**

**}**

**return** **(**ERR\_OK**);**

**}**

int write\_buff **(**flash\_info\_t **\*** info**,** uchar **\*** src**,** ulong addr**,** ulong cnt**)**

**{**

ulong wp**;**

uchar **\***p**;**

int aln**;**

cfiword\_t cword**;**

int i**,** rc**;**

int buffered\_size**;**

//portwidth = 0x02，也就是将bit0清0，也就是地址要以2为单位

wp **=** **(**addr **&** **~(**info**->**portwidth **-** 1**));**

//如果addr不等于wp表示地址不对齐

//这里aln只可能为1

**if** **((**aln **=** addr **-** wp**)** **!=** 0**)** **{**

cword**.**l **=** 0**;**

//获取wp的地址

p **=** map\_physmem**(**wp**,** info**->**portwidth**,** MAP\_NOCACHE**);**

//从norflash中读取对齐处地址

//例如要写入的的norflash地址为0x01，则需要将0x0处的字节先读出来

**for** **(**i **=** 0**;** i **<** aln**;** **++**i**)**

flash\_add\_byte **(**info**,** **&**cword**,** flash\_read8**(**p **+** i**));**

//将src处的第一个字节读出来，放到上面的cword中

**for** **(;** **(**i **<** info**->**portwidth**)** **&&** **(**cnt **>** 0**);** i**++)** **{**

flash\_add\_byte **(**info**,** **&**cword**,** **\***src**++);**

cnt**--;**

**}**

**for** **(;** **(**cnt **==** 0**)** **&&** **(**i **<** info**->**portwidth**);** **++**i**)**

flash\_add\_byte **(**info**,** **&**cword**,** flash\_read8**(**p **+** i**));**

//将组合好的字节写入到wp中

rc **=** flash\_write\_cfiword **(**info**,** wp**,** cword**);**

unmap\_physmem**(**p**,** info**->**portwidth**);**

**if** **(**rc **!=** 0**)**

**return** rc**;**

wp **+=** i**;**

FLASH\_SHOW\_PROGRESS**(**scale**,** dots**,** digit**,** i**);**

**}**

/\* handle the aligned part \*/

buffered\_size **=** **(**info**->**portwidth **/** info**->**chipwidth**);**

buffered\_size **\*=** info**->**buffer\_size**;**

**while** **(**cnt **>=** info**->**portwidth**)** **{**

//如果buffer size == 1，不要使用buufer write

**if** **(**info**->**buffer\_size **==** 1 **||** info**->**buffer\_size **==** 0**)** **{**

cword**.**l **=** 0**;**

**for** **(**i **=** 0**;** i **<** info**->**portwidth**;** i**++)**

flash\_add\_byte **(**info**,** **&**cword**,** **\***src**++);**

**if** **((**rc **=** flash\_write\_cfiword **(**info**,** wp**,** cword**))** **!=** 0**)**

**return** rc**;**

wp **+=** info**->**portwidth**;**

cnt **-=** info**->**portwidth**;**

**continue;**

**}**

//如果地址为buffer\_size的整数倍, 那么i就等于 buffer\_size.

//否则继续写到下一个buffered\_size对齐的位置

i **=** buffered\_size **-** **(**wp **%** buffered\_size**);**

**if** **(**i **>** cnt**)**

i **=** cnt**;**

**if** **((**rc **=** flash\_write\_cfibuffer **(**info**,** wp**,** src**,** i**))** **!=** ERR\_OK**)**

**return** rc**;**

i **-=** i **&** **(**info**->**portwidth **-** 1**);**

wp **+=** i**;**

src **+=** i**;**

cnt **-=** i**;**

FLASH\_SHOW\_PROGRESS**(**scale**,** dots**,** digit**,** i**);**

**}**

**if** **(**cnt **==** 0**)** **{**

**return** **(**0**);**

**}**

//如果还剩余一个字节，则先要读出后一个字节进行组合再写入

cword**.**l **=** 0**;**

p **=** map\_physmem**(**wp**,** info**->**portwidth**,** MAP\_NOCACHE**);**

**for** **(**i **=** 0**;** **(**i **<** info**->**portwidth**)** **&&** **(**cnt **>** 0**);** **++**i**)** **{**

flash\_add\_byte **(**info**,** **&**cword**,** **\***src**++);**

**--**cnt**;**

**}**

**for** **(;** i **<** info**->**portwidth**;** **++**i**)**

flash\_add\_byte **(**info**,** **&**cword**,** flash\_read8**(**p **+** i**));**

unmap\_physmem**(**p**,** info**->**portwidth**);**

**return** flash\_write\_cfiword **(**info**,** wp**,** cword**);**

**}**

//nor flash的buffer write

static int flash\_write\_cfibuffer **(**flash\_info\_t **\*** info**,** ulong dest**,** uchar **\*** cp**,**

int len**)**

**{**

flash\_sect\_t sector**;**

int cnt**;**

int retcode**;**

void **\***src **=** cp**;**

void **\***dst **=** map\_physmem**(**dest**,** len**,** MAP\_NOCACHE**);**

void **\***dst2 **=** dst**;**

int flag **=** 0**;**

uint offset **=** 0**;**

unsigned int shift**;**

uchar write\_cmd**;**

**switch** **(**info**->**portwidth**)** **{**

**case** FLASH\_CFI\_16BIT**:**

shift **=** 1**;**

**break;**

**}**

cnt **=** len **>>** shift**;**

//检查所写的区域是否erase过，否则退出

**while** **((**cnt**--** **>** 0**)** **&&** **(**flag **==** 0**))** **{**

**switch** **(**info**->**portwidth**)** **{**

**case** FLASH\_CFI\_8BIT**:**

flag **=** **((**flash\_read8**(**dst2**)** **&** flash\_read8**(**src**))** **==**

flash\_read8**(**src**));**

src **+=** 1**,** dst2 **+=** 1**;**

**break;**

**case** FLASH\_CFI\_16BIT**:**

flag **=** **((**flash\_read16**(**dst2**)** **&** flash\_read16**(**src**))** **==**

flash\_read16**(**src**));**

src **+=** 2**,** dst2 **+=** 2**;**

**break;**

**case** FLASH\_CFI\_32BIT**:**

flag **=** **((**flash\_read32**(**dst2**)** **&** flash\_read32**(**src**))** **==**

flash\_read32**(**src**));**

src **+=** 4**,** dst2 **+=** 4**;**

**break;**

**case** FLASH\_CFI\_64BIT**:**

flag **=** **((**flash\_read64**(**dst2**)** **&** flash\_read64**(**src**))** **==**

flash\_read64**(**src**));**

src **+=** 8**,** dst2 **+=** 8**;**

**break;**

**}**

**}**

**if** **(!**flag**)** **{**

retcode **=** ERR\_NOT\_ERASED**;**

**goto** out\_unmap**;**

**}**

src **=** cp**;**

//计算dest地址所处的section

sector **=** find\_sector **(**info**,** dest**);**

**switch** **(**info**->**vendor**)** **{**

**case** CFI\_CMDSET\_AMD\_STANDARD**:**

**case** CFI\_CMDSET\_AMD\_EXTENDED**:**

//解锁并写入对应的指令

flash\_unlock\_seq**(**info**,**0**);**

flash\_write\_cmd**(**info**,** sector**,** offset**,** AMD\_CMD\_WRITE\_TO\_BUFFER**);**

cnt **=** len **>>** shift**;**

flash\_write\_cmd**(**info**,** sector**,** offset**,** **(**uchar**)**cnt **-** 1**);**

**switch** **(**info**->**portwidth**)** **{**

**case** FLASH\_CFI\_16BIT**:**

**while** **(**cnt**--** **>** 0**)** **{**

//从src读数据写入到dst，2个字节为一个单位

flash\_write16**(**flash\_read16**(**src**),** dst**);**

src **+=** 2**,** dst **+=** 2**;**

**}**

**break;**

**}**

flash\_write\_cmd **(**info**,** sector**,** 0**,** AMD\_CMD\_WRITE\_BUFFER\_CONFIRM**);**

//等待操作完成

retcode **=** flash\_full\_status\_check **(**info**,** sector**,**

info**->**buffer\_write\_tout**,**

"buffer write"**);**

**break;**

**default:**

debug **(**"Unknown Command Set\n"**);**

retcode **=** ERR\_INVAL**;**

**break;**

**}**

out\_unmap**:**

unmap\_physmem**(**dst**,** len**);**

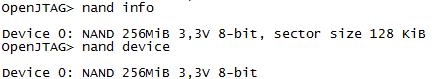
**return** retcode**;**

**}**

### 3.2 nandflash操作指令

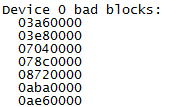
#### 3.2.1 nand info和nand device

从nand\_info全局变量读取nand基本信息，代码比较简单，这里不进行分析。



#### 3.2.2 nand bad

检查nand中的坏块。



//以页为单位，遍历整个nand，找出带有坏块标记的页

**if** **(**strcmp**(**cmd**,** "bad"**)** **==** 0**)** **{**

printf**(**"\nDevice %d bad blocks:\n"**,** nand\_curr\_device**);**

**for** **(**off **=** 0**;** off **<** nand**->**size**;** off **+=** nand**->**erasesize**)**

**if** **(**nand\_block\_isbad**(**nand**,** off**))**

printf**(**" %08x\n"**,** off**);**

**return** 0**;**

**}**

#### 3.2.3 nand erase和nand scrub

相关的操作指令和代码分析如下：

nand erase kernel

nand erase **[**clean**]** **[**off size**]**

nand scrub

**if** **(**strcmp**(**cmd**,** "erase"**)** **==** 0 **||** strcmp**(**cmd**,** "scrub"**)** **==** 0**)** **{**

nand\_erase\_options\_t opts**;**

//如果命令中的第二个参数为clean，则需要写CLEANMARKER，用于jffs2文件系统

//当一个擦写块被擦写完毕后，CLEANMARKER 节点会被写在 NOR flash 的开头，

//或 NAND flash 的 OOB(Out-Of-Band) 区域来表明这是一个干净，可写的擦写块。

int clean **=** argc **>** 2 **&&** **!**strcmp**(**"clean"**,** argv**[**2**]);**

int o **=** clean **?** 3 **:** 2**;**

//判断cmd是否是scrub

int scrub **=** **!**strcmp**(**cmd**,** "scrub"**);**

printf**(**"\nNAND %s: "**,** scrub **?** "scrub" **:** "erase"**);**

//跳过前2个或者3个参数，分析off和size，也分析nand erase kernel这种写法

**if** **(**arg\_off\_size**(**argc **-** o**,** argv **+** o**,** nand**,** **&**off**,** **&**size**)** **!=** 0**)**

**return** 1**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**offset **=** off**;**

opts**.**length **=** size**;**

opts**.**jffs2 **=** clean**;**

opts**.**quiet **=** quiet**;**

//如果是scrub命令

**if** **(**scrub**)** **{**

//由于scrub会擦除所有的坏块标记，所以会提示用户是否确定做这个操作

puts**(**"Warning: "

"scrub option will erase all factory set "

"bad blocks!\n"

" "

"There is no reliable way to recover them.\n"

" "

"Use this command only for testing purposes "

"if you\n"

" "

"are sure of what you are doing!\n"

"\nReally scrub this NAND flash? <y/N>\n"**);**

//y/n确认

**if** **(**getc**()** **==** 'y' **&&** getc**()** **==** '\r'**)** **{**

opts**.**scrub **=** 1**;**

**}** **else** **{**

puts**(**"scrub aborted\n"**);**

**return** **-**1**;**

**}**

**}**

printf**(**"nand\_erase\_opts\n"**);**

ret **=** nand\_erase\_opts**(**nand**,** **&**opts**);**

printf**(**"%s\n"**,** ret **?** "ERROR" **:** "OK"**);**

**return** ret **==** 0 **?** 0 **:** 1**;**

**}**

static int

arg\_off\_size**(**int argc**,** char **\***argv**[],** nand\_info\_t **\***nand**,** ulong **\***off**,** ulong **\***size**)**

**{**

int idx **=** nand\_curr\_device**;**

#if (CONFIG\_COMMANDS & CFG\_CMD\_JFFS2) && defined(CONFIG\_JFFS2\_CMDLINE)

struct mtd\_device **\***dev**;**

struct part\_info **\***part**;**

u8 pnum**;**

//处理nand erase kernel这种操作

**if** **(**argc **>=** 1 **&&** **!(**str2long**(**argv**[**0**],** off**)))** **{**

//查找kernel在mtd nand中的偏移

**if** **((**mtdparts\_init**()** **==** 0**)** **&&**

**(**find\_dev\_and\_part**(**argv**[**0**],** **&**dev**,** **&**pnum**,** **&**part**)** **==** 0**))** **{**

**if** **(**dev**->**id**->**type **!=** MTD\_DEV\_TYPE\_NAND**)** **{**

puts**(**"not a NAND device\n"**);**

**return** **-**1**;**

**}**

**\***off **=** part**->**offset**;**

//如果还有一个参数，便是长度，否则为整个分区的长度

**if** **(**argc **>=** 2**)** **{**

**if** **(!(**str2long**(**argv**[**1**],** size**)))** **{**

printf**(**"'%s' is not a number\n"**,** argv**[**1**]);**

**return** **-**1**;**

**}**

**if** **(\***size **>** part**->**size**)**

**\***size **=** part**->**size**;**

**}** **else** **{**

**\***size **=** part**->**size**;**

**}**

idx **=** dev**->**id**->**num**;**

**\***nand **=** nand\_info**[**idx**];**

**goto** out**;**

**}**

**}**

#endif

//分析偏移位置

**if** **(**argc **>=** 1**)** **{**

**if** **(!(**str2long**(**argv**[**0**],** off**)))** **{**

printf**(**"'%s' is not a number\n"**,** argv**[**0**]);**

**return** **-**1**;**

**}**

//如果argc=0，偏移为0

**}** **else** **{**

**\***off **=** 0**;**

**}**

//分析长度

**if** **(**argc **>=** 2**)** **{**

**if** **(!(**str2long**(**argv**[**1**],** size**)))** **{**

printf**(**"'%s' is not a number\n"**,** argv**[**1**]);**

**return** **-**1**;**

**}**

//如果argc=0，size为整个nand芯片的size

**}** **else** **{**

**\***size **=** nand**->**size **-** **\***off**;**

**}**

#if (CONFIG\_COMMANDS & CFG\_CMD\_JFFS2) && defined(CONFIG\_JFFS2\_CMDLINE)

out**:**

#endif

printf**(**"device %d "**,** idx**);**

**if** **(\***size **==** nand**->**size**)**

puts**(**"whole chip\n"**);**

**else**

printf**(**"offset 0x%x, size 0x%x\n"**,** **\***off**,** **\***size**);**

**return** 0**;**

**}**

int nand\_erase\_opts**(**nand\_info\_t **\***meminfo**,** const nand\_erase\_options\_t **\***opts**)**

**{**

struct jffs2\_unknown\_node cleanmarker**;**

int clmpos **=** 0**;**

int clmlen **=** 8**;**

erase\_info\_t erase**;**

ulong erase\_length**;**

int isNAND**;**

int bbtest **=** 1**;**

int result**;**

int percent\_complete **=** **-**1**;**

int **(\***nand\_block\_bad\_old**)(**struct mtd\_info **\*,** loff\_t**,** int**)** **=** **NULL;**

const char **\***mtd\_device **=** meminfo**->**name**;**

memset**(&**erase**,** 0**,** **sizeof(**erase**));**

erase**.**mtd **=** meminfo**;**

erase**.**len **=** meminfo**->**erasesize**;**

erase**.**addr **=** opts**->**offset**;**

erase\_length **=** opts**->**length**;**

isNAND **=** meminfo**->**type **==** MTD\_NANDFLASH **?** 1 **:** 0**;**

//如果是jffs2擦除，准备好cleanmarker结构体

**if** **(**opts**->**jffs2**)** **{**

cleanmarker**.**magic **=** cpu\_to\_je16 **(**JFFS2\_MAGIC\_BITMASK**);**

cleanmarker**.**nodetype **=** cpu\_to\_je16 **(**JFFS2\_NODETYPE\_CLEANMARKER**);**

**if** **(**isNAND**)** **{**

struct nand\_oobinfo **\***oobinfo **=** **&**meminfo**->**oobinfo**;**

/\* check for autoplacement \*/

**if** **(**oobinfo**->**useecc **==** MTD\_NANDECC\_AUTOPLACE**)** **{**

/\* get the position of the free bytes \*/

**if** **(!**oobinfo**->**oobfree**[**0**][**1**])** **{**

printf**(**" Eeep. Autoplacement selected "

"and no empty space in oob\n"**);**

**return** **-**1**;**

**}**

clmpos **=** oobinfo**->**oobfree**[**0**][**0**];**

clmlen **=** oobinfo**->**oobfree**[**0**][**1**];**

**if** **(**clmlen **>** 8**)**

clmlen **=** 8**;**

**}**

cleanmarker**.**totlen **=** cpu\_to\_je32**(**8**);**

**}**

cleanmarker**.**hdr\_crc **=** cpu\_to\_je32**(**

crc32\_no\_comp**(**0**,** **(**unsigned char **\*)** **&**cleanmarker**,**

**sizeof(**struct jffs2\_unknown\_node**)** **-** 4**));**

**}**

//如果是srcub操作

**if** **(**opts**->**scrub**)** **{**

struct nand\_chip **\***priv\_nand **=** meminfo**->**priv**;**

//暂时设置block bad为空，scrub操作不需要检查坏块

nand\_block\_bad\_old **=** priv\_nand**->**block\_bad**;**

priv\_nand**->**block\_bad **=** nand\_block\_bad\_scrub**;**

**if** **(**priv\_nand**->**bbt**)** **{**

kfree**(**priv\_nand**->**bbt**);**

**}**

priv\_nand**->**bbt **=** **NULL;**

**}**

//轮询并按block擦除

**for** **(;**

erase**.**addr **<** opts**->**offset **+** erase\_length**;**

erase**.**addr **+=** meminfo**->**erasesize**)** **{**

//如果不是如果是srcub操作操作

**if** **(!**opts**->**scrub **&&** bbtest**)** **{**

//检查是否是坏块

int ret **=** meminfo**->**block\_isbad**(**meminfo**,** erase**.**addr**);**

**if** **(**ret **>** 0**)** **{**

**if** **(!**opts**->**quiet**)**

printf**(**"\rSkipping bad block at "

"0x%08x "

" \n"**,**

erase**.**addr**);**

**continue;**

**}**

**}**

//擦除block

result **=** meminfo**->**erase**(**meminfo**,** **&**erase**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"\n%s: MTD Erase failure: %d\n"**,**

mtd\_device**,** result**);**

**continue;**

**}**

//如果是jffs2擦除，写入cleanmarker到erase.addr + clmpos位置，长度为clmlen

**if** **(**opts**->**jffs2**)** **{**

/\* write cleanmarker \*/

**if** **(**isNAND**)** **{**

size\_t written**;**

result **=** meminfo**->**write\_oob**(**meminfo**,**

erase**.**addr **+** clmpos**,**

clmlen**,**

**&**written**,**

**(**unsigned char **\*)**

**&**cleanmarker**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"\n%s: MTD writeoob failure: %d\n"**,**

mtd\_device**,** result**);**

**continue;**

**}**

**}**

**}**

**if** **(!**opts**->**quiet**)** **{**

int percent **=** **(**int**)**

**((**unsigned long long**)**

**(**erase**.**addr**+**meminfo**->**erasesize**-**opts**->**offset**)**

**\*** 100 **/** erase\_length**);**

/\* output progress message only at whole percent

\* steps to reduce the number of messages printed

\* on (slow) serial consoles

\*/

**if** **(**percent **!=** percent\_complete**)** **{**

percent\_complete **=** percent**;**

printf**(**"\rErasing at 0x%x -- %3d%% complete."**,**

erase**.**addr**,** percent**);**

**if** **(**opts**->**jffs2 **&&** result **==** 0**)**

printf**(**" Cleanmarker written at 0x%x."**,**

erase**.**addr**);**

**}**

**}**

**}**

**if** **(!**opts**->**quiet**)**

printf**(**"\n"**);**

//将block\_bad函数替换回去

**if** **(**nand\_block\_bad\_old**)** **{**

struct nand\_chip **\***priv\_nand **=** meminfo**->**priv**;**

priv\_nand**->**block\_bad **=** nand\_block\_bad\_old**;**

priv\_nand**->**scan\_bbt**(**meminfo**);**

**}**

**return** 0**;**

**}**

#### 3.2.4 nand dump

nand dump用于打印指定page的内容和oob内容。

nand dump**[.**oob**]** off **-** dump page

**if** **(**strncmp**(**cmd**,** "dump"**,** 4**)** **==** 0**)** **{**

**if** **(**argc **<** 3**)**

**goto** usage**;**

s **=** strchr**(**cmd**,** '.'**);**

off **=** **(**int**)**simple\_strtoul**(**argv**[**2**],** **NULL,** 16**);**

**if** **(**s **!=** **NULL** **&&** strcmp**(**s**,** ".oob"**)** **==** 0**)**

//该函数目前为空

ret **=** nand\_dump\_oob**(**nand**,** off**);**

**else**

ret **=** nand\_dump**(**nand**,** off**);**

**return** ret **==** 0 **?** 1 **:** 0**;**

**}**

static int nand\_dump**(**nand\_info\_t **\***nand**,** ulong off**)**

**{**

int i**;**

u\_char **\***buf**,** **\***p**;**

//2048+64

buf **=** malloc**(**nand**->**oobblock **+** nand**->**oobsize**);**

**if** **(!**buf**)** **{**

puts**(**"No memory for page buffer\n"**);**

**return** 1**;**

**}**

//取2048的整数倍

off **&=** **~(**nand**->**oobblock **-** 1**);**

//读取一个page和oob

i **=** nand\_read\_raw**(**nand**,** buf**,** off**,** nand**->**oobblock**,** nand**->**oobsize**);**

**if** **(**i **<** 0**)** **{**

printf**(**"Error (%d) reading page %08x\n"**,** i**,** off**);**

free**(**buf**);**

**return** 1**;**

**}**

//打印

printf**(**"Page %08x dump:\n"**,** off**);**

i **=** nand**->**oobblock **>>** 4**;** p **=** buf**;**

**while** **(**i**--)** **{**

printf**(** "\t%02x %02x %02x %02x %02x %02x %02x %02x"

" %02x %02x %02x %02x %02x %02x %02x %02x\n"**,**

p**[**0**],** p**[**1**],** p**[**2**],** p**[**3**],** p**[**4**],** p**[**5**],** p**[**6**],** p**[**7**],**

p**[**8**],** p**[**9**],** p**[**10**],** p**[**11**],** p**[**12**],** p**[**13**],** p**[**14**],** p**[**15**]);**

p **+=** 16**;**

**}**

puts**(**"OOB:\n"**);**

i **=** nand**->**oobsize **>>** 3**;**

**while** **(**i**--)** **{**

printf**(** "\t%02x %02x %02x %02x %02x %02x %02x %02x\n"**,**

p**[**0**],** p**[**1**],** p**[**2**],** p**[**3**],** p**[**4**],** p**[**5**],** p**[**6**],** p**[**7**]);**

p **+=** 8**;**

**}**

free**(**buf**);**

**return** 0**;**

**}**

#### 3.2.5 nand read和nand write

nand在这里的读写操作都是以页为单位的。

nand write和nand write.jffs2的区别在于jffs2操作写入的长度可以不是2048的整数倍。

nand write.yaffs和其他操作的区别在于ecc数据就存在文件系统中，

nand read**[.**jffs2**]** **-** addr off size

nand write**[.**jffs2**]** **-** addr off size

或者

nand read**.**jffs2 0x30007FC0 kernel

nand write**.**yaffs 0x30000000 root

//K9F2G08U0C除了提供了页读和页写功能外，还提供了页内地址随意读、写功能。

//页读和页写是从页的首地址开始读、写，而随意读、写实现了在一页范围内任意地址的读、写。

//随意读操作是在页读操作后输入随意读命令和页内列地址，这样就可以读取到列地址所指定地址的数据。

//随意写操作是在页写操作的第二个页写命令周期前，输入随意写命令和页内列地址，以及要写入的数据，

//这样就可以把数据写入到列地址所指定的地址内。

/\* read write \*/

**if** **(**strncmp**(**cmd**,** "read"**,** 4**)** **==** 0 **||** strncmp**(**cmd**,** "write"**,** 5**)** **==** 0**)** **{**

int read**;**

**if** **(**argc **<** 4**)**

**goto** usage**;**

addr **=** **(**ulong**)**simple\_strtoul**(**argv**[**2**],** **NULL,** 16**);**

read **=** strncmp**(**cmd**,** "read"**,** 4**)** **==** 0**;** /\* 1 = read, 0 = write \*/

printf**(**"\nNAND %s: "**,** read **?** "read" **:** "write"**);**

//分析操作nand的偏移地址和长度

**if** **(**arg\_off\_size**(**argc **-** 3**,** argv **+** 3**,** nand**,** **&**off**,** **&**size**)** **!=** 0**)**

**return** 1**;**

s **=** strchr**(**cmd**,** '.'**);**

//jffs2读写

**if** **(**s **!=** **NULL** **&&**

**(!**strcmp**(**s**,** ".jffs2"**)** **||** **!**strcmp**(**s**,** ".e"**)** **||** **!**strcmp**(**s**,** ".i"**)))** **{**

**if** **(**read**)** **{**

/\* read \*/

nand\_read\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

opts**.**quiet **=** quiet**;**

ret **=** nand\_read\_opts**(**nand**,** **&**opts**);**

**}** **else** **{**

/\* write \*/

nand\_write\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

/\* opts.forcejffs2 = 1; \*/

opts**.**pad **=** 1**;**

opts**.**blockalign **=** 1**;**

opts**.**quiet **=** quiet**;**

ret **=** nand\_write\_opts**(**nand**,** **&**opts**);**

**}**

//yaff2读写

**}else** **if** **(** s **!=** **NULL** **&&** **!**strcmp**(**s**,** ".yaffs"**)){**

**if** **(**read**)** **{**

/\* read \*/

nand\_read\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

opts**.**readoob **=** 1**;**

opts**.**quiet **=** quiet**;**

ret **=** nand\_read\_opts**(**nand**,** **&**opts**);**

**}** **else** **{**

/\* write \*/

nand\_write\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

/\* opts.forceyaffs = 1; \*/

opts**.**noecc **=** 1**;**

opts**.**writeoob **=** 1**;**

opts**.**blockalign **=** 1**;**

opts**.**quiet **=** quiet**;**

opts**.**skipfirstblk **=** 1**;**

ret **=** nand\_write\_opts**(**nand**,** **&**opts**);**

**}**

//raw读写，without ecc and oob

**}else** **if** **(** s **!=** **NULL** **&&** **!**strcmp**(**s**,** ".raw"**)){**

**if** **(**read**)** **{**

/\* read \*/

nand\_read\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

opts**.**readoob **=** 0**;**

opts**.**quiet **=** quiet**;**

opts**.**noecc **=** 1**;**

opts**.**nocheckbadblk **=** 1**;**

ret **=** nand\_read\_opts**(**nand**,** **&**opts**);**

**}** **else** **{**

/\* write \*/

nand\_write\_options\_t opts**;**

memset**(&**opts**,** 0**,** **sizeof(**opts**));**

opts**.**buffer **=** **(**u\_char**\*)** addr**;**

opts**.**length **=** size**;**

opts**.**offset **=** off**;**

/\* opts.forceyaffs = 1; \*/

opts**.**noecc **=** 1**;**

opts**.**writeoob **=** 0**;**

opts**.**blockalign **=** 1**;**

opts**.**quiet **=** quiet**;**

opts**.**skipfirstblk **=** 0**;**

opts**.**nocheckbadblk **=** 1**;**

ret **=** nand\_write\_opts**(**nand**,** **&**opts**);**

**}**

**}**

//普通读写

**else** **{**

**if** **(**read**)**

ret **=** nand\_read**(**nand**,** off**,** **&**size**,** **(**u\_char **\*)**addr**);**

**else**

ret **=** nand\_write**(**nand**,** off**,** **&**size**,** **(**u\_char **\*)**addr**);**

**}**

printf**(**" %d bytes %s: %s\n"**,** size**,**

read **?** "read" **:** "written"**,** ret **?** "ERROR" **:** "OK"**);**

**return** ret **==** 0 **?** 0 **:** 1**;**

**}**

int nand\_read\_opts**(**nand\_info\_t **\***meminfo**,** const nand\_read\_options\_t **\***opts**)**

**{**

int imglen **=** opts**->**length**;**

int pagelen**;**

int baderaseblock**;**

int blockstart **=** **-**1**;**

int percent\_complete **=** **-**1**;**

loff\_t offs**;**

size\_t readlen**;**

ulong mtdoffset **=** opts**->**offset**;**

u\_char **\***buffer **=** opts**->**buffer**;**

int result**;**

struct nand\_oobinfo old\_oobinfo**;**

int oobinfochanged **=** 0**;**

/\* make sure device page sizes are valid \*/

**if** **(!(**meminfo**->**oobsize **==** 16 **&&** meminfo**->**oobblock **==** 512**)**

**&&** **!(**meminfo**->**oobsize **==** 8 **&&** meminfo**->**oobblock **==** 256**)**

**&&** **!(**meminfo**->**oobsize **==** 64 **&&** meminfo**->**oobblock **==** 2048**))** **{**

printf**(**"Unknown flash (not normal NAND)\n"**);**

**return** **-**1**;**

**}**

//yaffs2才需要read oob，因为文件系统本身是包含oob的

pagelen **=** meminfo**->**oobblock

**+** **((**opts**->**readoob **!=** 0**)** **?** meminfo**->**oobsize **:** 0**);**

//读取的长度不要超过nand的总长度

**if** **(((**imglen **/** pagelen**)** **\*** meminfo**->**oobblock**)**

**>** **(**meminfo**->**size **-** opts**->**offset**))** **{**

printf**(**"Image %d bytes, NAND page %d bytes, "

"OOB area %u bytes, device size %u bytes\n"**,**

imglen**,** pagelen**,** meminfo**->**oobblock**,** meminfo**->**size**);**

printf**(**"Input block is larger than device\n"**);**

**return** **-**1**;**

**}**

**if** **(!**opts**->**quiet**)**

printf**(**"\n"**);**

//备份当前的oobinfo

memcpy**(&**old\_oobinfo**,** **&**meminfo**->**oobinfo**,** **sizeof(**old\_oobinfo**));**

//如果不需要ecc操作，将none\_oobinfo赋给当前的oobinfo

**if** **(**opts**->**noecc**)** **{**

memcpy**(&**meminfo**->**oobinfo**,** **&**none\_oobinfo**,**

**sizeof(**meminfo**->**oobinfo**));**

oobinfochanged **=** 1**;**

**}**

/\* get data from input and write to the device \*/

**while** **(**imglen **&&** **(**mtdoffset **<** meminfo**->**size**))** **{**

WATCHDOG\_RESET **();**

//在进行读操作之前，判断操作的区域是否是坏块，如果是坏块，则需要跳过该坏块区域

**while** **(**blockstart **!=** **(**mtdoffset **&** **(~**meminfo**->**erasesize**+**1**)))** **{**

blockstart **=** mtdoffset **&** **(~**meminfo**->**erasesize**+**1**);**

offs **=** blockstart**;**

baderaseblock **=** 0**;**

/\* check all the blocks in an erase block for

\* bad blocks \*/

**if** **(!**opts**->**nocheckbadblk**)** **{**

**do** **{**

int ret **=** meminfo**->**block\_isbad**(**meminfo**,** offs**);**

**if** **(**ret **<** 0**)** **{**

printf**(**"Bad block check failed\n"**);**

**goto** restoreoob**;**

**}**

**if** **(**ret **==** 1**)** **{**

baderaseblock **=** 1**;**

**if** **(!**opts**->**quiet**)**

printf**(**"\rBad block at 0x%lx "

"in erase block from "

"0x%x will be skipped\n"**,**

**(**long**)** offs**,**

blockstart**);**

**}**

**if** **(**baderaseblock**)** **{**

mtdoffset **=** blockstart

**+** meminfo**->**erasesize**;**

**}**

offs **+=** meminfo**->**erasesize**;**

**}** **while** **(**offs **<** blockstart **+** meminfo**->**erasesize**);**

**}**

**}**

//读nand偏移为mtdoffset处的内容，长度为2048

result **=** meminfo**->**read**(**meminfo**,**

mtdoffset**,**

meminfo**->**oobblock**,**

**&**readlen**,**

**(**unsigned char **\*)** **&**data\_buf**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"reading NAND page at offset 0x%lx failed\n"**,**

mtdoffset**);**

**goto** restoreoob**;**

**}**

//如果读出的数据大于剩余要读的数据，修改readlen的值

**if** **(**imglen **<** readlen**)** **{**

readlen **=** imglen**;**

**}**

//复制数据到缓冲并修改相关参数

memcpy**(**buffer**,** data\_buf**,** readlen**);**

buffer **+=** readlen**;**

imglen **-=** readlen**;**

//如果需要都oob，则将oob读出

**if** **(**opts**->**readoob**)** **{**

result **=** meminfo**->**read\_oob**(**meminfo**,**

mtdoffset**,**

meminfo**->**oobsize**,**

**&**readlen**,**

**(**unsigned char **\*)**

**&**oob\_buf**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"\nMTD readoob failure: %d\n"**,**

result**);**

**goto** restoreoob**;**

**}**

**if** **(**imglen **<** readlen**)** **{**

readlen **=** imglen**;**

**}**

//将obb拷贝到缓冲

memcpy**(**buffer**,** oob\_buf**,** readlen**);**

buffer **+=** readlen**;**

imglen **-=** readlen**;**

**}**

**if** **(!**opts**->**quiet**)** **{**

int percent **=** **(**int**)**

**((**unsigned long long**)**

**(**opts**->**length**-**imglen**)** **\*** 100

**/** opts**->**length**);**

**if** **(**percent **!=** percent\_complete**)** **{**

**if** **(!**opts**->**quiet**)**

printf**(**"\rReading data from 0x%x "

"-- %3d%% complete."**,**

mtdoffset**,** percent**);**

percent\_complete **=** percent**;**

**}**

**}**

mtdoffset **+=** meminfo**->**oobblock**;**

**}**

**if** **(!**opts**->**quiet**)**

printf**(**"\n"**);**

restoreoob**:**

//恢复oobinfo，将ecc操作还原

**if** **(**oobinfochanged**)** **{**

memcpy**(&**meminfo**->**oobinfo**,** **&**old\_oobinfo**,**

**sizeof(**meminfo**->**oobinfo**));**

**}**

**if** **(**imglen **>** 0**)** **{**

printf**(**"Could not read entire image due to bad blocks\n"**);**

**return** **-**1**;**

**}**

/\* return happy \*/

**return** 0**;**

**}**

int nand\_write\_opts**(**nand\_info\_t **\***meminfo**,** const nand\_write\_options\_t **\***opts**)**

**{**

int imglen **=** 0**;**

int pagelen**;**

int baderaseblock**;**

int blockstart **=** **-**1**;**

loff\_t offs**;**

int readlen**;**

int oobinfochanged **=** 0**;**

int percent\_complete **=** **-**1**;**

struct nand\_oobinfo old\_oobinfo**;**

ulong mtdoffset **=** opts**->**offset**;**

ulong erasesize\_blockalign**;**

u\_char **\***buffer **=** opts**->**buffer**;**

size\_t written**;**

int result**;**

int skipfirstblk **=** opts**->**skipfirstblk**;**

//jffs2 pad=1 writeoob=0

//yaffs2 pad=0 writeoob=1

**if** **(**opts**->**pad **&&** opts**->**writeoob**)** **{**

printf**(**"Can't pad when oob data is present.\n"**);**

**return** **-**1**;**

**}**

//jffs2 yaffs2 raw blockalign=1

**if** **(**opts**->**blockalign **==** 0**)** **{**

erasesize\_blockalign **=** meminfo**->**erasesize**;**

**}** **else** **{**

erasesize\_blockalign **=** meminfo**->**erasesize **\*** opts**->**blockalign**;**

**}**

/\* make sure device page sizes are valid \*/

**if** **(!(**meminfo**->**oobsize **==** 16 **&&** meminfo**->**oobblock **==** 512**)**

**&&** **!(**meminfo**->**oobsize **==** 8 **&&** meminfo**->**oobblock **==** 256**)**

**&&** **!(**meminfo**->**oobsize **==** 64 **&&** meminfo**->**oobblock **==** 2048**))** **{**

printf**(**"Unknown flash (not normal NAND)\n"**);**

**return** **-**1**;**

**}**

//备份当前的oobinfo

memcpy**(&**old\_oobinfo**,** **&**meminfo**->**oobinfo**,** **sizeof(**old\_oobinfo**));**

//是否需要写ecc到oob

//yaffs2 raw noecc=1，yaffs2文件系统自带ecc

**if** **(**opts**->**noecc**)** **{**

memcpy**(&**meminfo**->**oobinfo**,** **&**none\_oobinfo**,**

**sizeof(**meminfo**->**oobinfo**));**

oobinfochanged **=** 1**;**

**}**

//这里不会运行下面的条件

**if** **(**opts**->**autoplace **&&** **(**old\_oobinfo**.**useecc **!=** MTD\_NANDECC\_AUTOPLACE**))** **{**

memcpy**(&**meminfo**->**oobinfo**,** **&**autoplace\_oobinfo**,**

**sizeof(**meminfo**->**oobinfo**));**

oobinfochanged **=** 1**;**

**}**

//这里不会运行下面的条件

**if** **(**opts**->**forcejffs2 **||** opts**->**forceyaffs**)** **{**

struct nand\_oobinfo **\***oobsel **=**

opts**->**forcejffs2 **?** **&**jffs2\_oobinfo **:** **&**yaffs\_oobinfo**;**

**if** **(**meminfo**->**oobsize **==** 8**)** **{**

**if** **(**opts**->**forceyaffs**)** **{**

printf**(**"YAFSS cannot operate on "

"256 Byte page size\n"**);**

**goto** restoreoob**;**

**}**

/\* Adjust number of ecc bytes \*/

jffs2\_oobinfo**.**eccbytes **=** 3**;**

**}**

memcpy**(&**meminfo**->**oobinfo**,** oobsel**,** **sizeof(**meminfo**->**oobinfo**));**

oobinfochanged **=** 1**;** /\* add by thisway.diy, www.100ask.net \*/

**}**

//获取要写内容的长度

imglen **=** opts**->**length**;**

//获取要写页的大小，带oob的为2048+64，不带为2048

pagelen **=** meminfo**->**oobblock

**+** **((**opts**->**writeoob **!=** 0**)** **?** meminfo**->**oobsize **:** 0**);**

//如果是yaffs2操作，需要检查imglen和pagelen是否是整数倍关系

printf**(**"imglen=%d,pagelen=%d\n"**,**imglen**,**pagelen**);**

**if** **((!**opts**->**pad**)** **&&** **((**imglen **%** pagelen**)** **!=** 0**))** **{**

printf**(**"Input block length is not page aligned\n"**);**

**goto** restoreoob**;**

**}**

//检查要写入的长度是否超过nand边界

**if** **(((**imglen **/** pagelen**)** **\*** meminfo**->**oobblock**)**

**>** **(**meminfo**->**size **-** opts**->**offset**))** **{**

printf**(**"Image %d bytes, NAND page %d bytes, "

"OOB area %u bytes, device size %u bytes\n"**,**

imglen**,** pagelen**,** meminfo**->**oobblock**,** meminfo**->**size**);**

printf**(**"Input block does not fit into device\n"**);**

**goto** restoreoob**;**

**}**

**if** **(!**opts**->**quiet**)**

printf**(**"\n"**);**

/\* get data from input and write to the device \*/

**while** **(**imglen **&&** **(**mtdoffset **<** meminfo**->**size**))** **{**

WATCHDOG\_RESET **();**

//在进行读操作之前，判断操作的区域是否是坏块，如果是坏块，则需要跳过该坏块区域

**while** **(**blockstart **!=** **(**mtdoffset **&** **(~**erasesize\_blockalign**+**1**)))** **{**

blockstart **=** mtdoffset **&** **(~**erasesize\_blockalign**+**1**);**

offs **=** blockstart**;**

baderaseblock **=** 0**;**

/\* check all the blocks in an erase block for

\* bad blocks \*/

**if** **(!**opts**->**nocheckbadblk**)** **{**

**do** **{**

int ret **=** meminfo**->**block\_isbad**(**meminfo**,** offs**);**

**if** **(**ret **<** 0**)** **{**

printf**(**"Bad block check failed\n"**);**

**goto** restoreoob**;**

**}**

**if** **(**ret **==** 1**)** **{**

baderaseblock **=** 1**;**

**if** **(!**opts**->**quiet**)**

printf**(**"\rBad block at 0x%lx "

"in erase block from "

"0x%x will be skipped\n"**,**

**(**long**)** offs**,**

blockstart**);**

**}**

**if** **(**baderaseblock**)** **{**

mtdoffset **=** blockstart

**+** erasesize\_blockalign**;**

**}**

offs **+=** erasesize\_blockalign

**/** opts**->**blockalign**;**

**}** **while** **(**offs **<** blockstart **+** erasesize\_blockalign**);**

**}**

**}**

//如果是yaffs image需要跳过第一个好的块，why？

/\* skip the first good block when wirte yaffs image, by www.100ask.net \*/

**if** **(**skipfirstblk**)** **{**

mtdoffset **+=** erasesize\_blockalign**;**

skipfirstblk **=** 0**;**

**continue;**

**}**

//readlen=2048

readlen **=** meminfo**->**oobblock**;**

//如果是jffs2操作，如果要写入的长度小于2048，因为写是以页为单位

//将不需要操作的区域置为0xff，也就是不改变当前内容

**if** **(**opts**->**pad **&&** **(**imglen **<** readlen**))** **{**

readlen **=** imglen**;**

memset**(**data\_buf **+** readlen**,** 0xff**,**

meminfo**->**oobblock **-** readlen**);**

**}**

//从bufffer中读取数据

memcpy**(**data\_buf**,** buffer**,** readlen**);**

buffer **+=** readlen**;**

//如果是yaffs2系统需要将ecc写入到oob

**if** **(**opts**->**writeoob**)** **{**

//读取64字节的ecc数据

memcpy**(**oob\_buf**,** buffer**,** meminfo**->**oobsize**);**

buffer **+=** meminfo**->**oobsize**;**

//写oob

result **=** meminfo**->**write\_oob**(**meminfo**,**

mtdoffset**,**

meminfo**->**oobsize**,**

**&**written**,**

**(**unsigned char **\*)**

**&**oob\_buf**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"\nMTD writeoob failure: %d\n"**,**

result**);**

**goto** restoreoob**;**

**}**

imglen **-=** meminfo**->**oobsize**;**

**}**

//写数据

result **=** meminfo**->**write**(**meminfo**,**

mtdoffset**,**

meminfo**->**oobblock**,**

**&**written**,**

**(**unsigned char **\*)** **&**data\_buf**);**

**if** **(**result **!=** 0**)** **{**

printf**(**"writing NAND page at offset 0x%lx failed\n"**,**

mtdoffset**);**

**goto** restoreoob**;**

**}**

imglen **-=** readlen**;**

**if** **(!**opts**->**quiet**)** **{**

int percent **=** **(**int**)**

**((**unsigned long long**)**

**(**opts**->**length**-**imglen**)** **\*** 100

**/** opts**->**length**);**

/\* output progress message only at whole percent

\* steps to reduce the number of messages printed

\* on (slow) serial consoles

\*/

**if** **(**percent **!=** percent\_complete**)** **{**

printf**(**"\rWriting data at 0x%x "

"-- %3d%% complete."**,**

mtdoffset**,** percent**);**

percent\_complete **=** percent**;**

**}**

**}**

mtdoffset **+=** meminfo**->**oobblock**;**

**}**

**if** **(!**opts**->**quiet**)**

printf**(**"\n"**);**

restoreoob**:**

**if** **(**oobinfochanged**)** **{**

memcpy**(&**meminfo**->**oobinfo**,** **&**old\_oobinfo**,**

**sizeof(**meminfo**->**oobinfo**));**

**}**

**if** **(**imglen **>** 0**)** **{**

printf**(**"Data did not fit into device, due to bad blocks\n"**);**

**return** **-**1**;**

**}**

/\* return happy \*/

**return** 0**;**

**}**

### 3.3网络操作指令

## uboot自定义menu分析

增加命令读取uboot到ram并启动