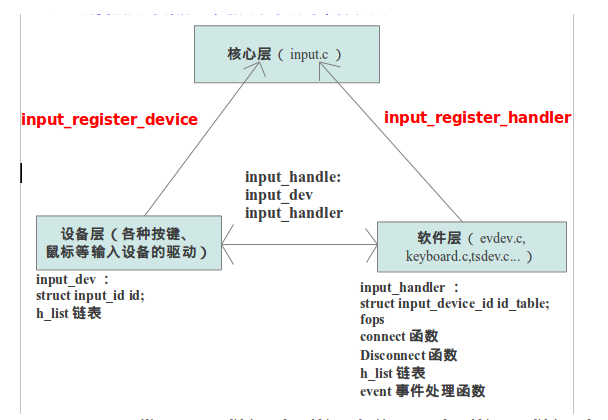
# 输入子系统

我们以前都是自己写驱动程序，可是我们需要嵌入系统之中，这时候我们就需要输入子系统

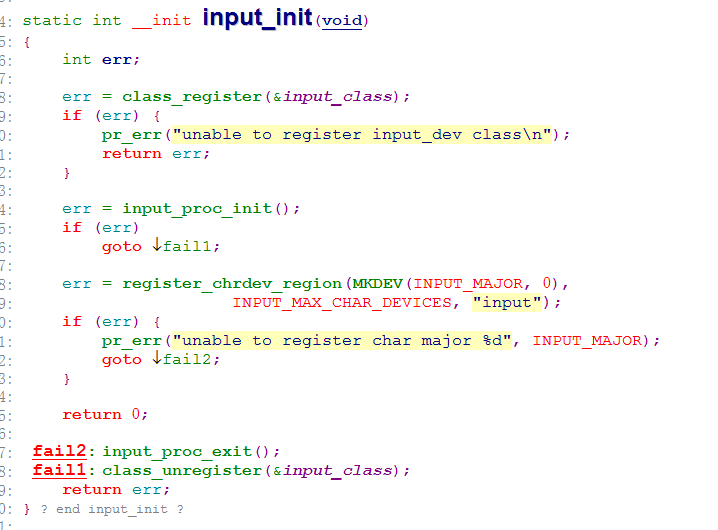
输入子系统就是内核的输入子系统是对分散的，多种不同类别的输入设备(如键盘，鼠标，跟踪球，操纵杆，触摸屏，加速计和手写板)等字符设备进行统一处理的一层抽象，就是在字符设备驱动上抽象出的一层。

我们的输入子系统就像下面的图一样



输入子系统一共有这么几个层面，首先它们都在input.c中开始的

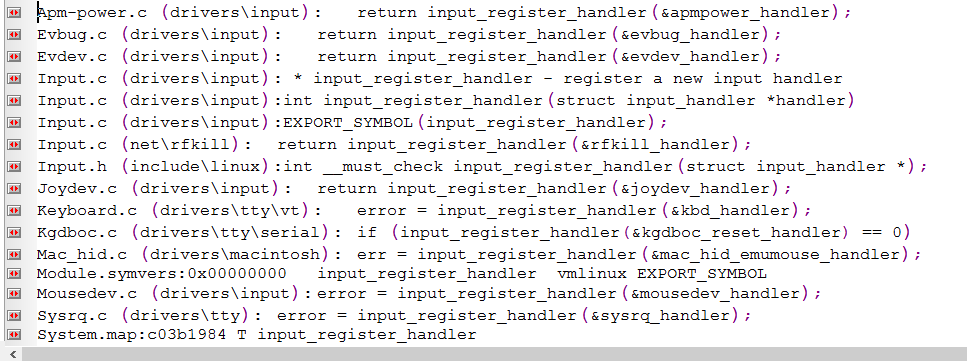
我们首先看一下input.c中的init函数



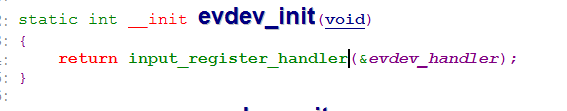
我们可以看到这个函数中有一个注册函数，register\_chrdev\_region，这跟我们的register\_chrdev有所不同，我们原来的register\_chrdev是将我们的一些工作都已经做好了，比如说将我们的fops之类的结构体已经帮我们注册到内核中去了，但是register\_chrdev\_region函数只是将我们的主次设备号以及设备节点名称注册，还得需要一个cdev\_init函数来将我们的函数注册进去，cdev\_init函数在我们驱动程序的connect中之后会有分析

这是我们的核心层面，input，然后我们还有另外两个层面，一个是硬件驱动dev层还有一个就是我们的处理handler层，它们都有它们自己的注册函数，我们的输入子系统也是从这里开始的，我们先看下面的这个函数

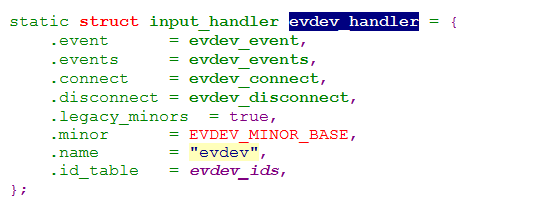
这个函数就是我们的handler层的注册函数，我们搜索一下看看哪个函数用到它了



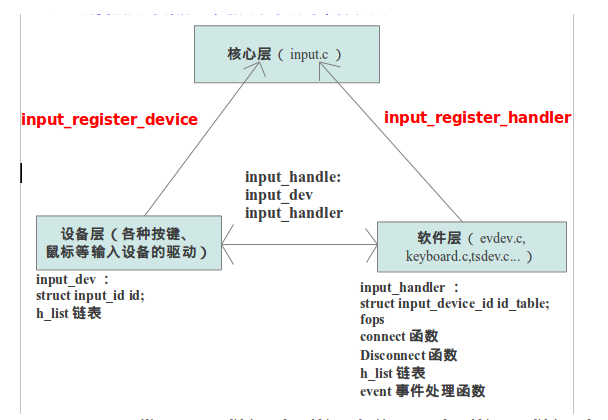
我们可以看到有很多驱动程序，比如说鼠标、键盘之类的，我们就打开evdev驱动程序看一下



我们看到是init函数调用的这个函数，这里面有个结构体evdev\_handler，我们先看看是什么



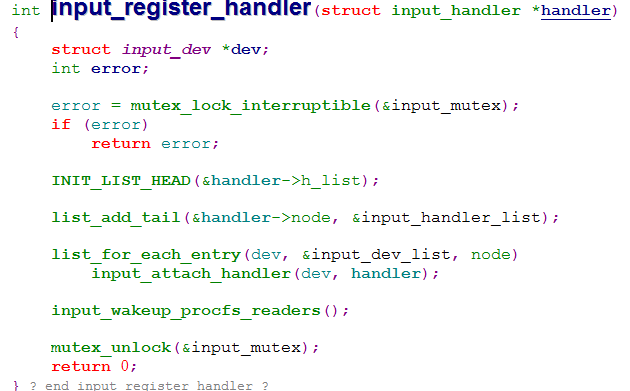
我们看到这里面有很多函数，我们看到最底下有个id\_table，这里我们就要研究一下input子系统中整个的体系了



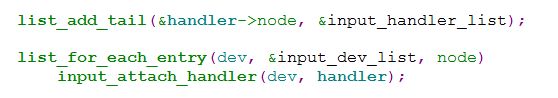
这里面有三个层，我们的软件层也是我们的处理层，是进行一些驱动的处理的，这样我们的驱动层只需要进行对硬件的处理提供现在的硬件信息就可以了，然后软件层进行处理

但是在那之前它们两个需要先进行比较连接，来看看这个handler层是否支持这个硬件dev层

这就需要前面的id\_table了，我们回到之前的handler注册函数中看一看



我们看到

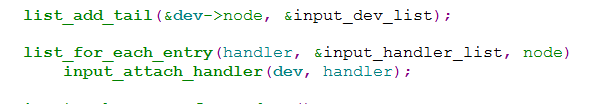


这里就是进行验证的函数

这里的第一个函数是先将handler链表插入，第二个函数对每个input\_dev都使用第三个函数

第三个函数是进行对dev和handler的比较id

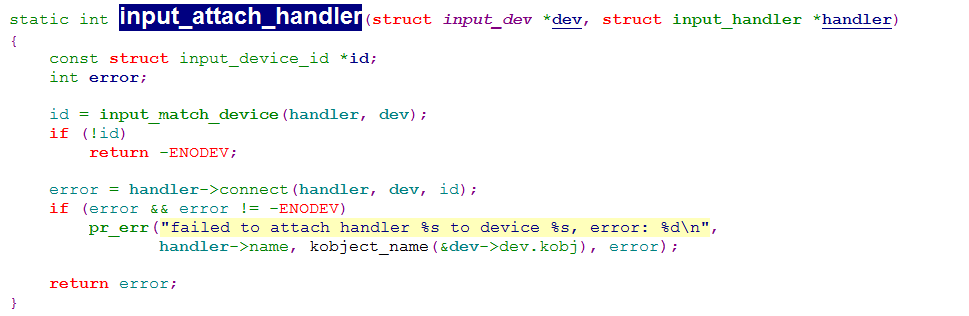
在这时我们也看一下dev的注册函数



这里也有一个类似的函数，意思也大致跟上面的相同，就是跟handler比较是否匹配

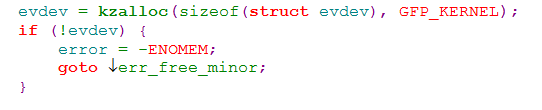
如果两个检测都匹配

我们进入input\_attach\_handler函数看一下

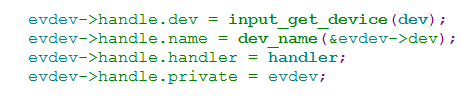


这里代表如果都匹配就调用handler->connect(handler, dev, id);函数也就是连接函数

我们回过头来看看这个连接函数

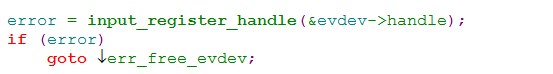


这个连接函数首先就会分配一个handle结构体来配置input\_dev和input\_handler



这里就是配置另外两层的结构体

然后最后需要在内核中注册一下



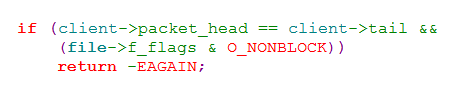


这里是将那些处理函数注册进去，这里就是和老版内核不同的地方

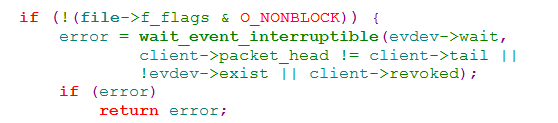
这里就是整个的输入子系统怎么初始化准备的，这样我们可以通过input\_dev找到handler或者也可以通过input\_handler找到dev

然后我们看一下那里的读函数

读函数就和我们自己写的读函数就差不多了，



这里是判断非阻塞方式，让我们重新检测的，跟我们写的是一样的



这里是如果是阻塞方式那就是让程序休眠，跟我们是一样的，我们再看看是谁来唤醒



我们看到在这个函数中有唤醒函数，然后我们看看这个函数是谁在调用它



结果我们看到是events函数在调用它，这也就说明了最终会调用这个处理函数

那么是谁来调用这个处理函数呢

应该是我们的驱动程序中调用了

我们有个历程是gpio\_keys.c这个是内核中的例子我们打开看看



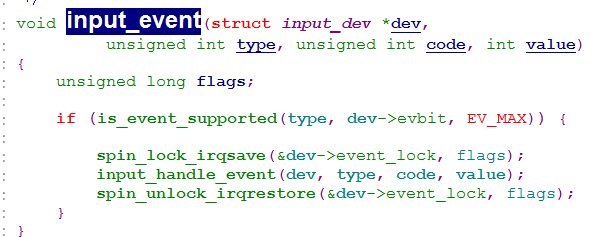
这里有个中断处理函数，我们仔细看一看，里面有



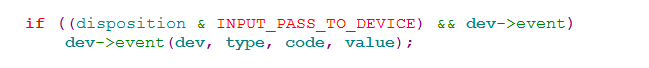
这两个函数，第一个函数就是上报核心层需要处理

第二个函数表明上报完毕

我们再进入input\_event函数看看



我们再进入input\_handle\_event看看



在这里我们就有调用event函数的了

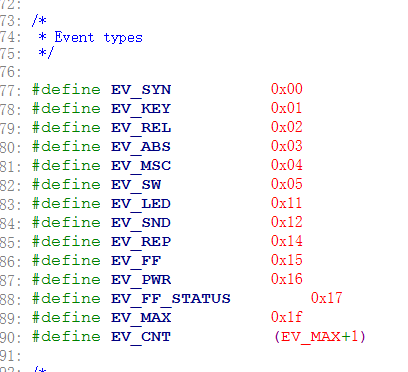
整个驱动框架就分析差不多了

我们总结一下步骤

1. 分配input\_handle
2. 配置
3. 注册

Linux的输入子系统不仅支持鼠标、键盘等常规的输入设备，而且还支持蜂鸣器、触摸屏等设备

输入子系统包括的设备在input.h中可以看到



#define EV\_SYN            0x00               //表示设备支持所有事件  
#define EV\_KEY            0x01               //键盘或者按键，表示一个键码  
#define EV\_REL            0x02                //鼠标设备，表示一个相对的光标位置结果  
#define EV\_ABS            0x03               //手写板产生的值，其是一个绝对整数值  
#define EV\_MSC            0x04              //其他类型  
#define EV\_SW             0x05                  
#define EV\_LED            0x11               //LED设备  
#define EV\_SND            0x12              //蜂鸣器，输入声音  
#define EV\_REP            0x14              //允许重复按键类型  
#define EV\_FF            0x15  
#define EV\_PWR            0x16             //电源管理事件  
#define EV\_FF\_STATUS        0x17  
#define EV\_MAX            0x1f  
#define EV\_CNT            (EV\_MAX+1)

## 编写输入子系统的按键驱动程序

我们现在已经大致了解输入子系统的框架了，我们现在来开始编写这个程序

其实整个驱动程序跟我们原来的差不多，只不过是输入子系统将其中部分抽离了，然后我们只需要完成dev层就可以了

我们首先来个初始化函数和退出函数



首先我们要按照上面的框架来写程序，我们用input\_allocate\_device函数来分配一个input\_dev的结构体的空间

然后我们需要配置这个结构体我们用set\_bit函数来配置这个结构体，我们用EV\_KEY来说明是按键类的输入，然后后面的EV\_REP是用重复按键，比如说我们一直按着键盘不放会一直输出很多个这个键子，然后我们我们设置按键输入的都是什么值

接下来就是注册这个结构体，用上面说到的input\_register\_device函数

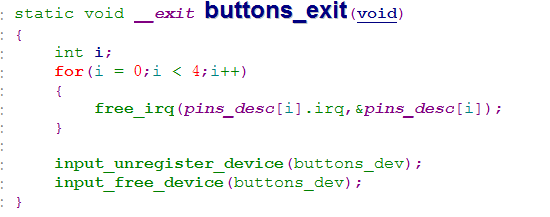
然后接下来我们要写的是硬件相关的函数

我们跟原来的一样，也就是说先启动定时器

跟之前写的驱动是一样的，然后注册中断函数，这里的pins\_desc数组是



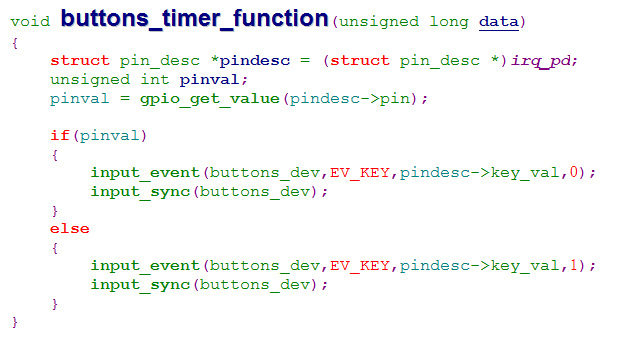
然后接下来是退出函数



很简单，就是将那些init函数中初始化的一些东西全都释放掉就可以了

然后接下来我们需要写buttons\_irq函数

这个函数跟之前的函数是一样的，在此就不说明了，然后是定时器中断处理函数



这里跟原来的不同是，原来是读出电平后然后就返回给上层函数就行了，然而这个是要将这个数据上报输入子系统

这两个一个是上报函数按下是0，松开是1

这样我们的驱动程序就写完了

然后我们放到板子上去运行，如果我们是运行的qt程序，我们就在它的记事本里新建一个文件就会出现效果

如果没有运行类似qt的程序我们可以cat /dev/tty0就好了，这个tty几是当时装载的几就是几

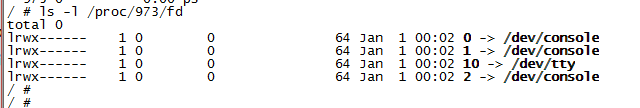


我这里是0所以我是tty0，然后按下之后回车就会有显示了

我们也可以把输入源从串口该到这个tty0，



这样我们就可以在命令行用那几个按键了，



这里是我们的shell程序的详细内容

0是标准输入、1是标准输出、2是标准错误

我们也可以用hexdump命令来进行16进制的查看按键值



后面的是这个驱动的位置

# 驱动的分层概念

驱动分层其实在上面的输入子系统已经接触到了分层的概念，就是将我们原来写的那种驱动程序分离开来了，将容易变动的代码放到一起，然后将比较稳定不动的代码放到一边

容易变动的代码我们暂时叫做device，稳定不动的叫driver，其实这就是两个结构体而已

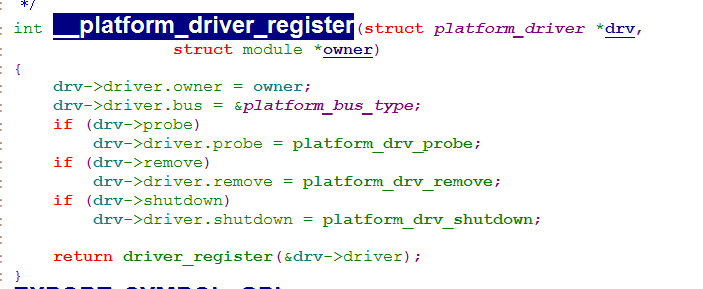
然后分别通过device\_add和platform\_driver\_register函数将这两个结构体放进驱动的链表里面，然后这两个结构体通过结构体中的.match函数进行名字的对比，如果相符就调用.probe函数

我们可以看一下代码

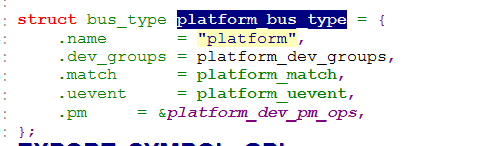
我们看一下gpio\_keys这个例程



这里是我们的platform\_driver\_register函数，我们可以点进去看看



这里首先有个platform\_bus\_tyoes结构体，我们点进去看看



这里就有我们之前说的match函数

我们点进去看看



在里面有这么一行，这行就是在对比driver与device的名字的函数，如果匹配就会调用probe函数

我们回过头来看看probe函数

这个函数其实就是我们可以自己干想干的事了

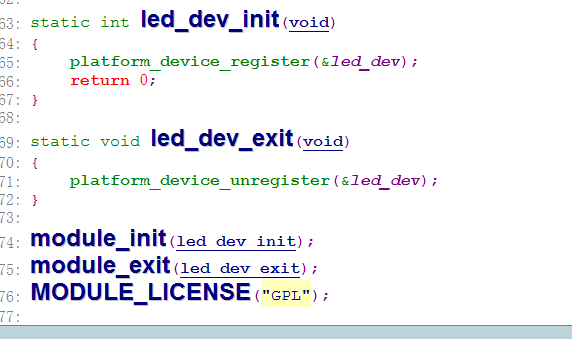
我们可以来写一个led驱动程序来看一下

我们需要创建两个文件，一个是led\_dev一个是led\_drv

然后我们分别对它们进行编辑

首先我们写dev，也就是需要经常改的那部分

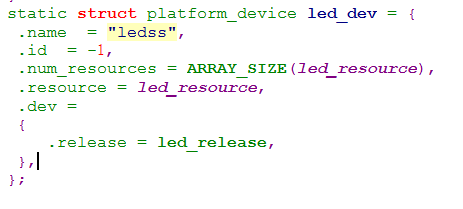
首先需要有一个init和一个exit函数



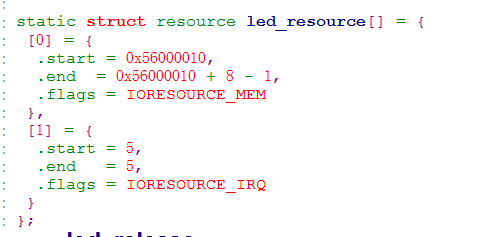
Init函数用platform\_device\_register函数注册led\_dev结构体，这里面就是调用了device\_add函数

然后我们再看这个led\_dev结构体怎么定义

我们搜索一下别的文件都怎么用的

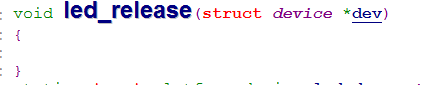


这里就是根据别的文件的结构体来改的，名字叫ledss，这里有个led\_resource结构体，叫做资源，我们看一下这个资源怎么定义



这里面写的是第一个元素是这个管脚的地址，第二个写的是第几个位置的灯亮，也就是我们要哪个灯亮就改这个数就好了，比如mini2440就是5 6 7 8这几个

然后led\_release函数什么也不写是个空函数

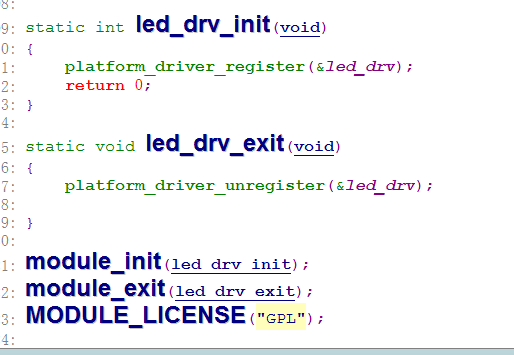


然后我们的led\_dev程序就写好了

接下来我们继续写led\_drv程序

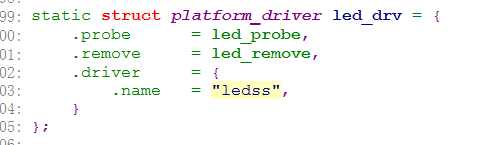
这部分程序是稳定不变的程序

首先我们一样要把init和exit函数写出来



这里跟dev是一样的，将结构体放到链表中

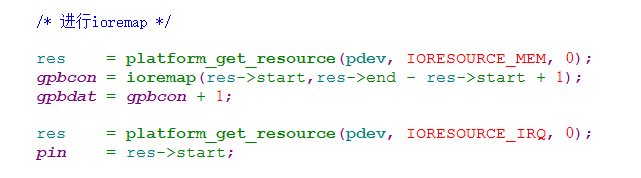
然后就要写led\_drv结构体



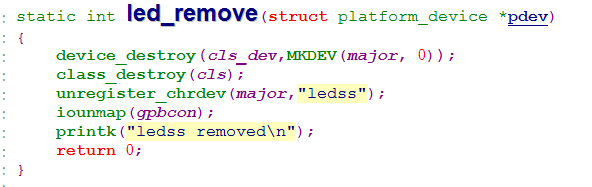
我们仿照gpio\_keys文件写一个，然后把这两个函数写出来



Led\_probe函数就相当于我们之前的驱动程序的init函数，注册设备节点之类的东西，这些东西都是稳定不变的程序

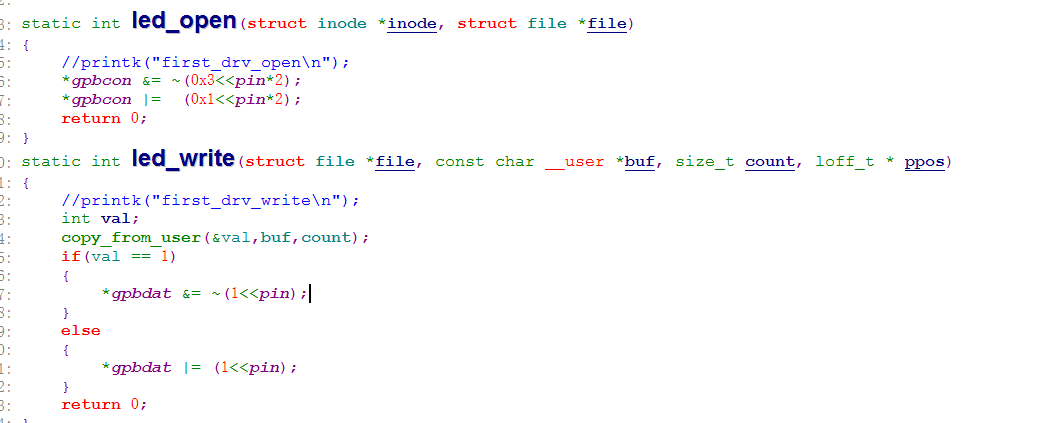
然后还有iormap

这里就是调用刚才在dev文件中写的资源结构体，



Remove就是将上面的函数创建的东西释放而已

然后我们继续写led\_fops结构体



这些就是跟我们原来写的东西一样

这样我们的驱动程序就写完了，然后测试程序跟原来测试led的程序是一样的

# LCD驱动

在我们的内核中有自带的LCD驱动，但是我们也可以自己写一个驱动程序

首先我们要了解LCD液晶屏幕

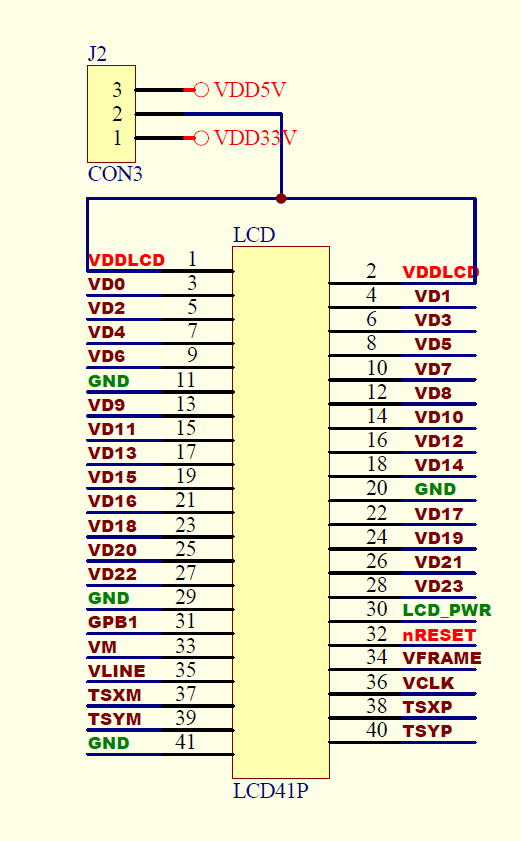
我们用的是mini2440的屏幕LCD\_X35,与山哥的jz2440是不一样的

我们首先要了解LCD是怎么工作的

LCD是由一个一个像素点构成的，我们如果要操作它就首先需要知道LCD内部有一个电子元件类似指针一样的东西，它会一个一个地在每个像素点控制颜色变化

我们操作的就是那个指针一样的东西，这个指针最开始是在最开始的位置，然后我们开始给出信号之后就会横向平移，然后平移到最后一个就会另起一行继续平移直到最后一个

我们是用一个信号引脚来控制这个指针平移的，VCLK这个引脚发出一个时钟信号后这个指针就开始移动，每发送一个时钟信号这个指针就会移动到下一个像素中去，然后如果到了这一行的最后一个像素的时候会有一个行信号来表示这个行已经显示完毕，然后我们就开始控制指针到下一行，然后到最后一个像素点的时候也会有个信号提示已经到最后一个像素点了，然后就控制这个指针就回到第一个像素点，然后往复这样VLINE就是这个行的信号，VFRAME是我们的列信号，表示到最后一个像素点的

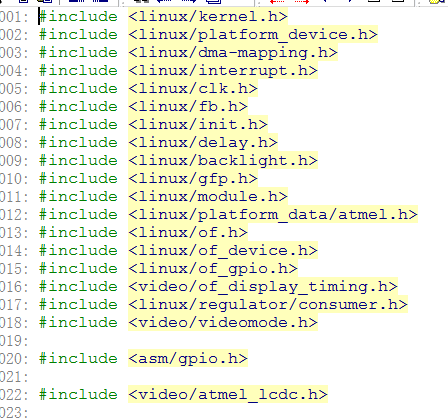


这个就是LCD的引脚图，VD0~VD23是控制这个指针的要显示的颜色的控制数据线

然而这个数据的来源就是叫一个显存的一个东西，但是2440还没有那么高级，我们是直接把它放到SDRAM中去，打一个像素点就传一个数据

硬件的操作就是这样，我们接下来可以看一下代码

首先头文件就可以参考别的文件的头文件



然后我们按照之前写的驱动程序一样首先要写init和exit以及他们的入口，和用GPL协议

然后我们首先要写的就是我们的init函数

我们首先需要定义一个结构体



然后我们要在init函数中需要干以下几点

给fb\_info分配一个空间

然后对它的设置

然后是硬件的相关操作，最后是注册

首先我们先来分配空间

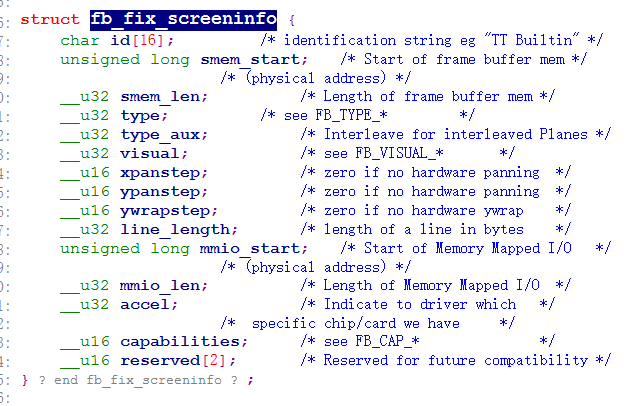


然后我们要对lcd\_info结构体进行设置，我们首先看看这个结构体都有什么成员



首先我们看一下这两个结构体，我们首先就需要设置这两个结构体，一个是固定的参数，一个是可变的参数

我们来看看fix（固定参数）里面都有什么



首先ID我们自己起就可以了

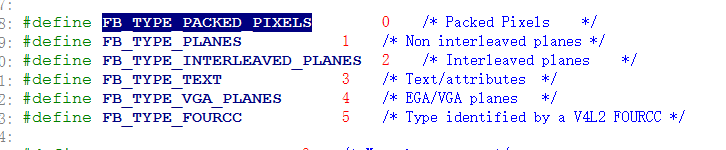


Smem\_start是我们的显示屏的物理地址，这个我们先放到后面，放到硬件相关的操作上面去

然后是smem\_len 这个是显存所占的字节数，在我们的友善之臂mini2440中我现在用的是LCD\_X35，这个LCD的像素是8\*8\*8的 真彩色所以我们是32/8



接下来是type，注释上说看FB\_BYTE那么我们就看看这个



这里有平板的还有别的，我们用第一个就可以了，默认



Type\_aux不用弄

Visual说让我们看看FB\_VIAUAL那我们就看看



我们的TFT显示屏是真彩色，所以我们用第三个



\_\_u16 xpanstep; /\* zero if no hardware panning \*/

\_\_u16 ypanstep; /\* zero if no hardware panning \*/

\_\_u16 ywrapstep; /\* zero if no hardware ywrap \*/

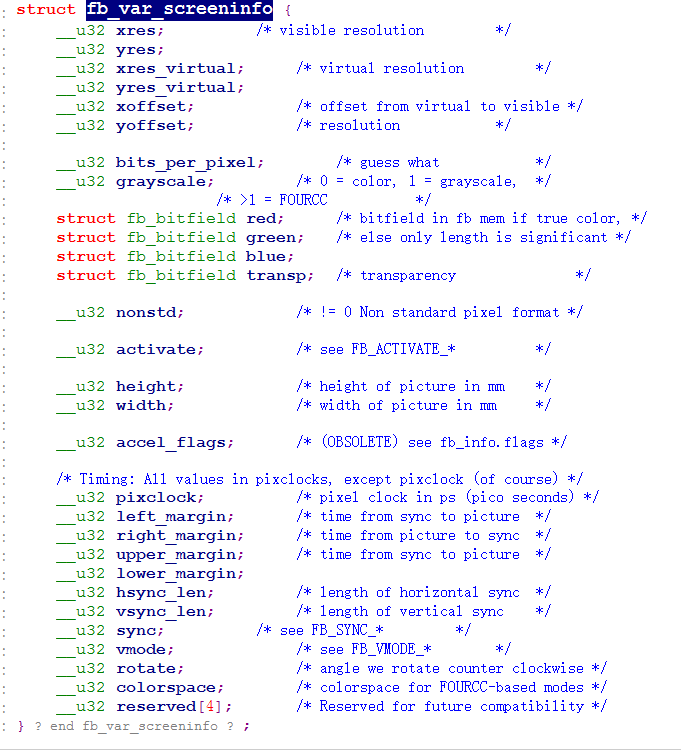
这几个我们看一下，我们不需要设置因为我们都没有注释上面说的东西，所以我们不用管，因为默认的就是0

然后到了line\_length，这个是一行的字节数，所以我们要写240\*4



其余的成员我们先不管

然后接下来我们配置一下var（可变参数）



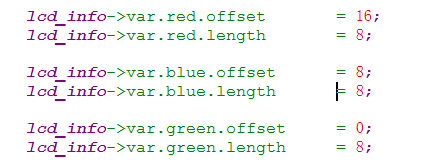
首先这个是它的成员

我们首先设置前几个



分别是x方向和y方向的像素数，和x与y的虚拟方向，这里虚拟方向就跟我们在电脑上设置屏幕的分辨率是一样的，可以设置比现在硬件上面小的分辨率

最后一个是红绿蓝三原色的位数，咱们的是888，所以就是24



这三个分别是三原色的偏移地址以及所占位数



这个我们设置成默认的就可以了

然后就是一些其它的设置了



这个是屏幕的所占的大小



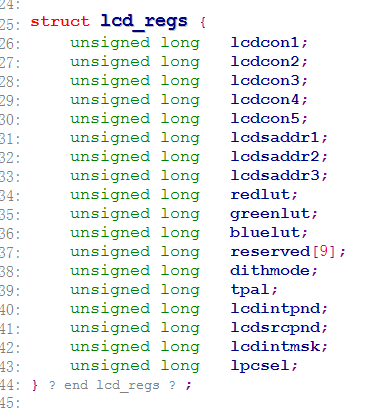
这里是注册

接下来就是对硬件的操作了

我们首先来看LCD控制器

我们首先要把这些LCD的控制器的地址全部映射出来

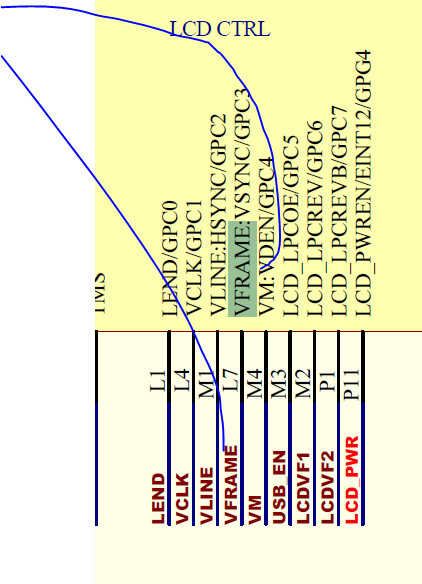
我们可以定义一个结构体然后对这个结构体进行映射

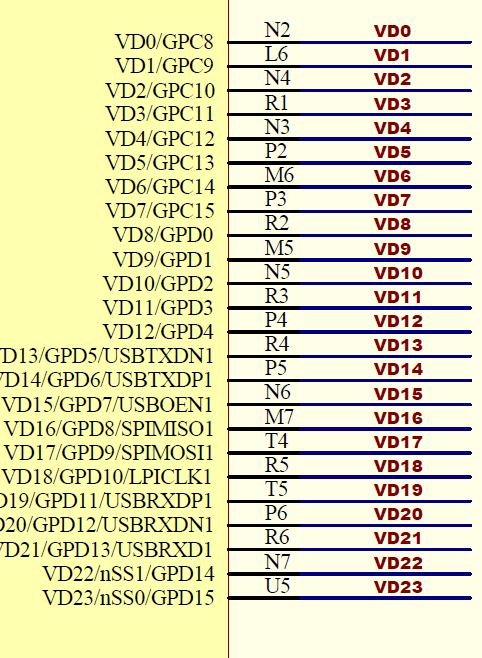


然后进行ioremap



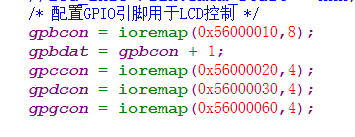
看看LCD都对应的都是哪些引脚





这里我们可以看到，LCD一共需要三类管脚，GPG GPC GPD

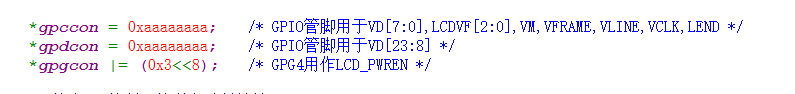
然后我们先ioremap一下



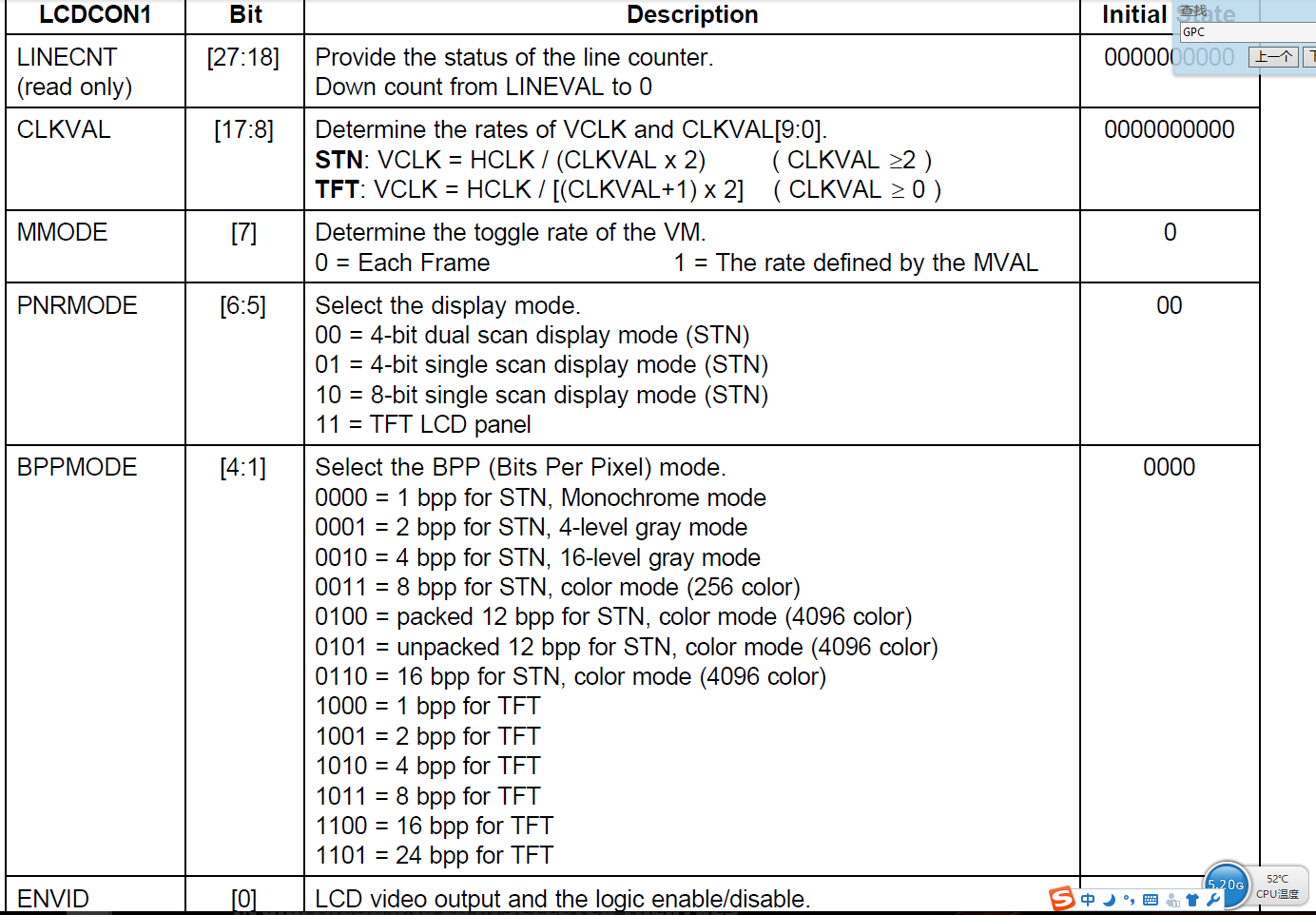
然后我们看数据手册，怎么设置他们才能成为对LCD有作用的管脚

我们看了一下就是将GPC和GPD所有管脚都设置为10，就好了

然后GPG因为就是一个，所以在那个管脚上赋值就可以了



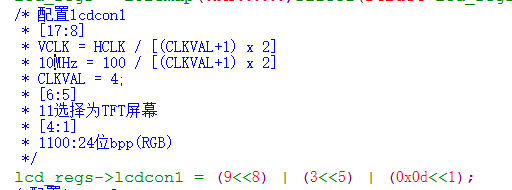
然后我们看一下lcdcon1寄存器



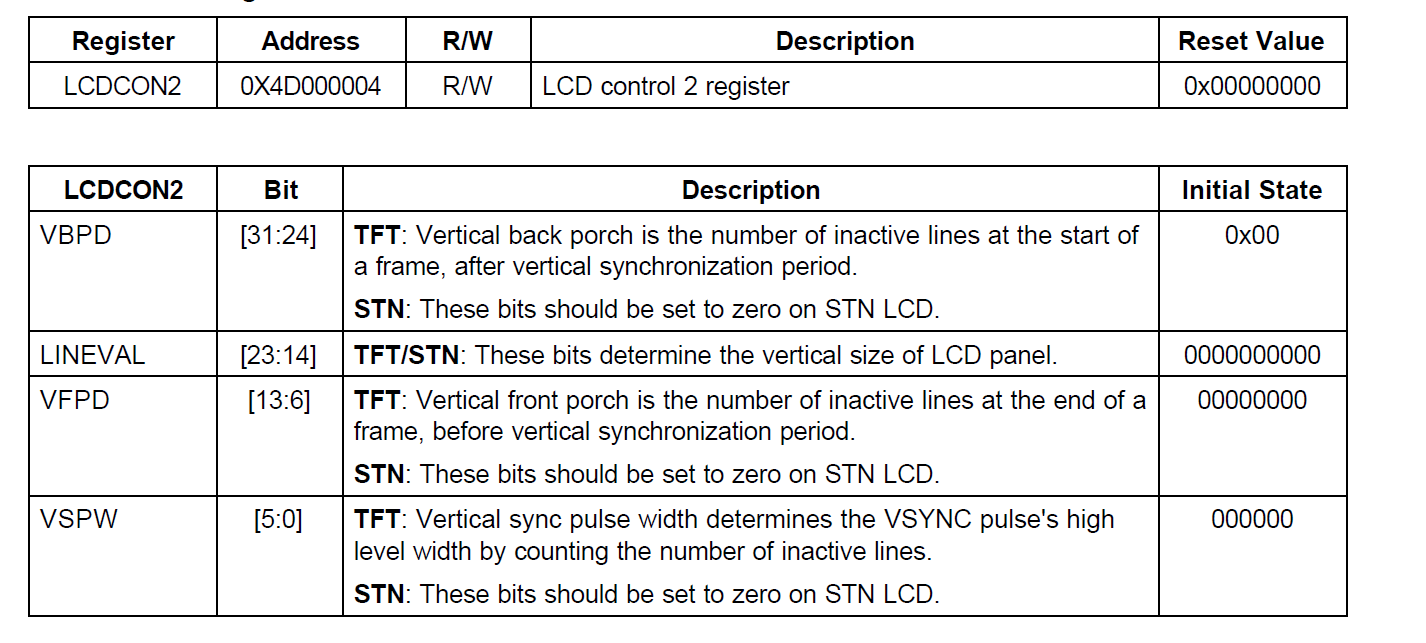
27:18这个位我们不用设置

17:8这个位置我们需要算一下，HCLK是100MHz，然后LCD的VCLK是183ns左右，所以算下来CLKVAL是9

然后是7位，0默认就可以了，6:5位设置11,4：1位设置1101就好了

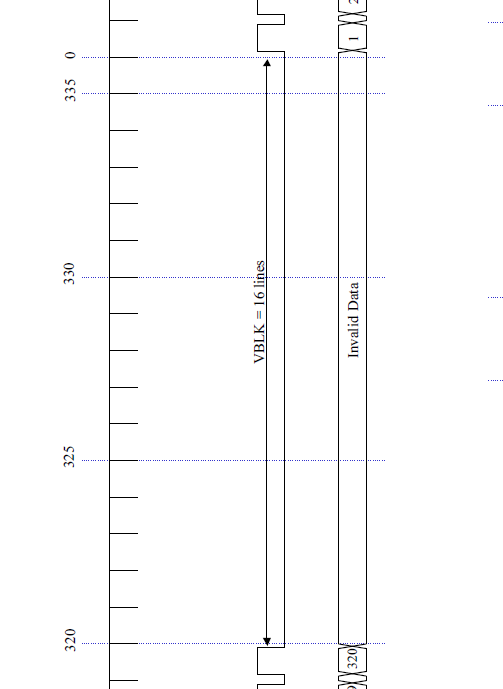


然后是lcdcon2

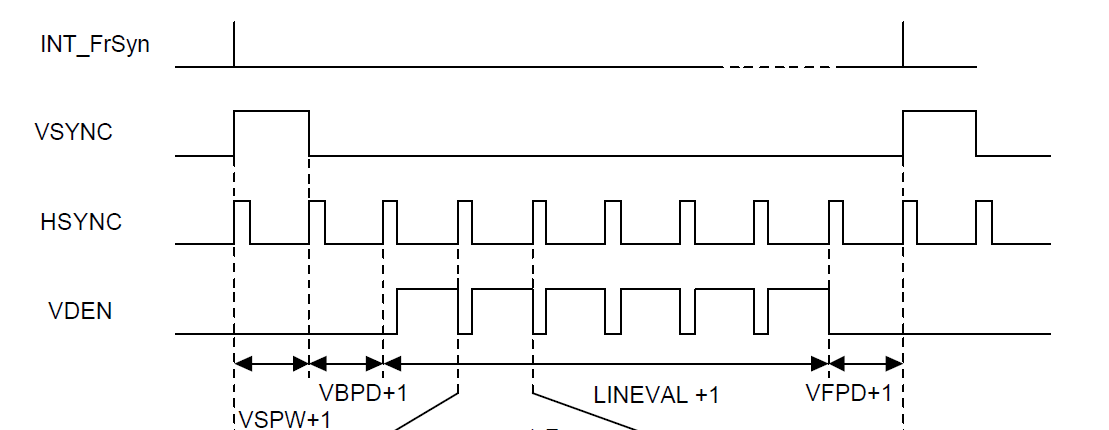


接下来就需要设置时序了，首先声明一下，LCD\_X35这个液晶屏的时序非常的坑，它没有HSYNC和VSYNC两条线，而且它的时序图画的非常的不清楚

这里就简单说一下



这是VSYNC线的时序图，我们看一下2440数据手册提供的参考的时序



我们再看一下LCD控制器

首先VBPD是31:24位，然后我们看一下VBPD对应我们的时序图的哪里

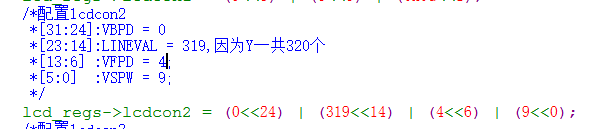


事实上对应的是这里，所以VBFD为0，那样那里的时序才能等于1

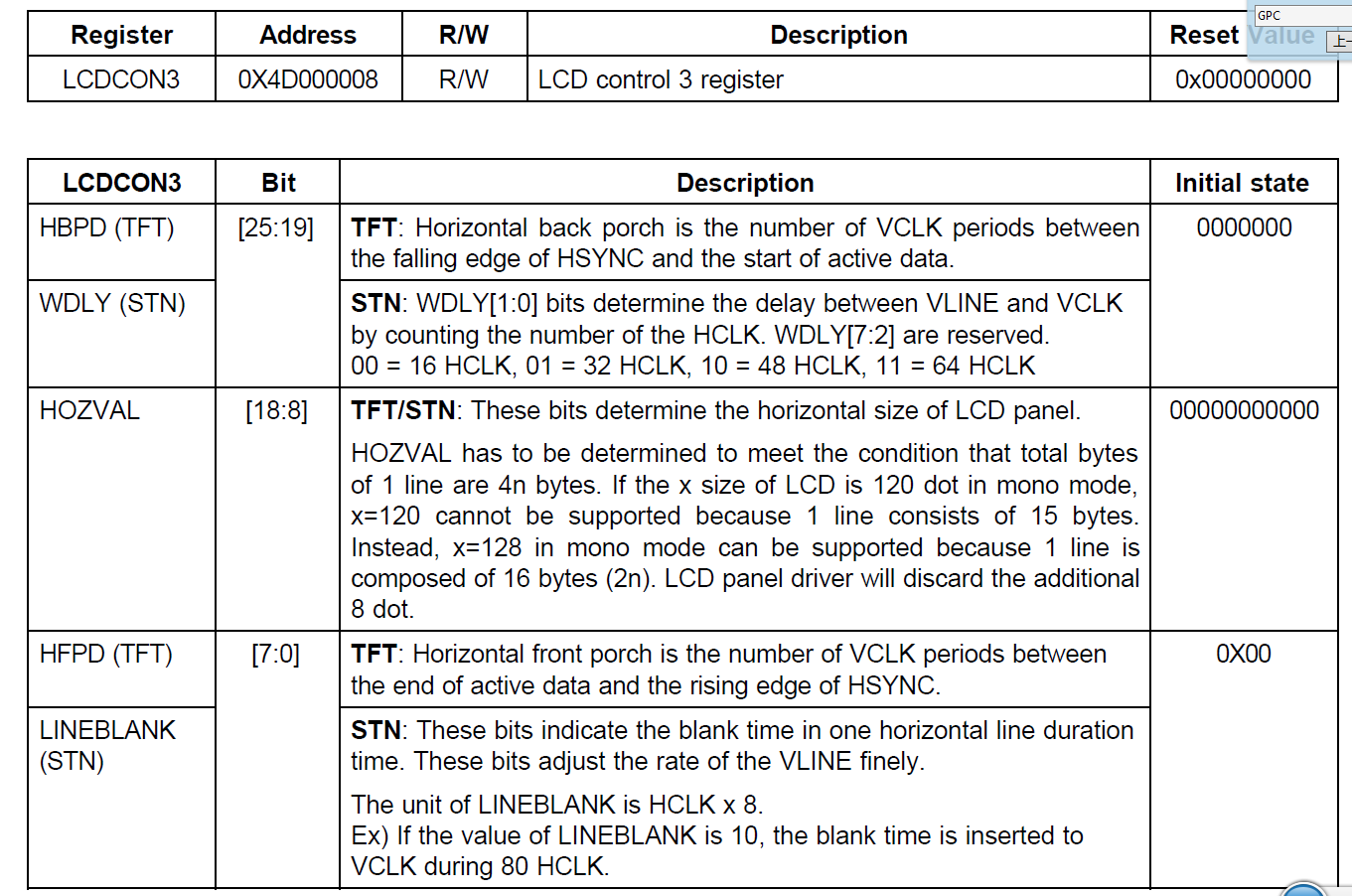
然后是LINEVAL是319，也就是320-1，没什么好说的

我们先看一下VSPW怎么弄，在我们的LCD里面是没有这个引脚的，我们就得按照MCK和PCI两条线来推断，具体是怎么推出来的本人也不太清楚，但是根据LCD的例程来看的话长度是10，然后我们应该取值是9，

这样我们的VFPD位也出来的是4

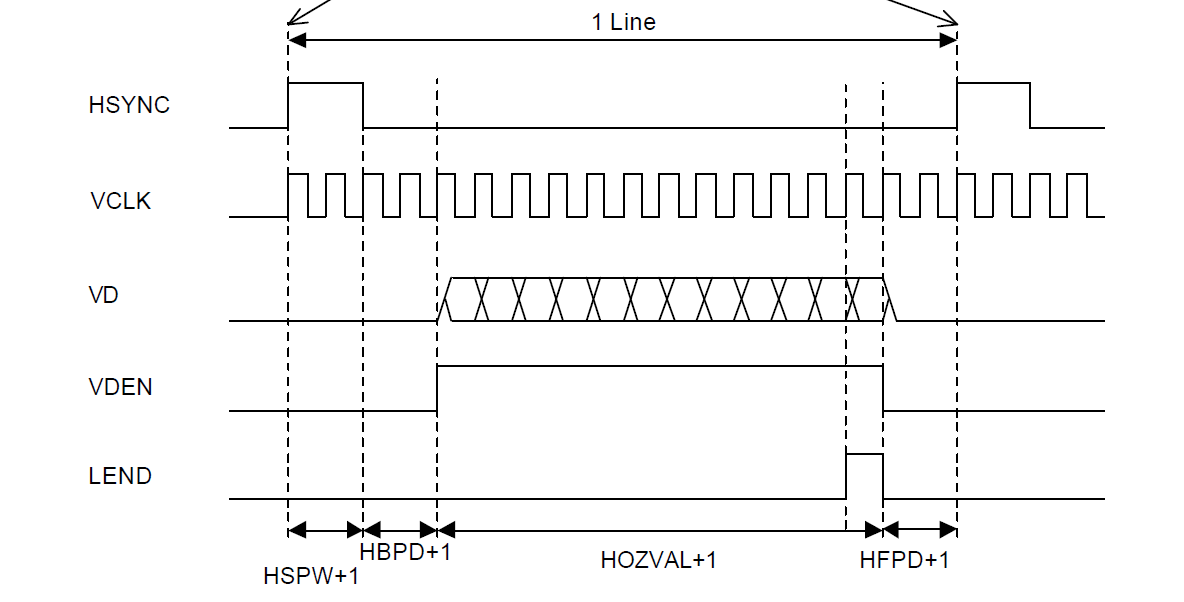


然后是lcdcon3

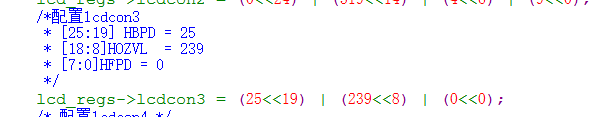


这个是行信号

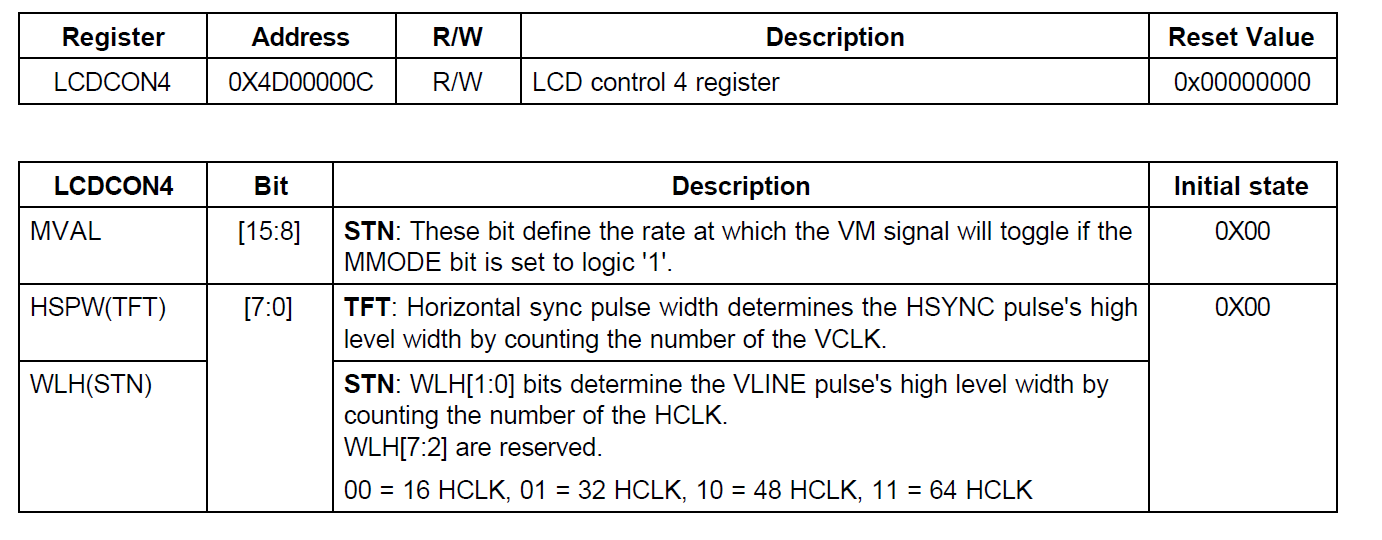
我们先看一下2440手册上的时序图以及LCD手册上的关于行的时序图



我们先看HBPD，是VCLK的数量，这样的话我们还是需要知道HSPW的数值是多少，是4，也就是说HSYNC的宽度是5，然后就能推算出来接下来的了



接下来是lcdcon4



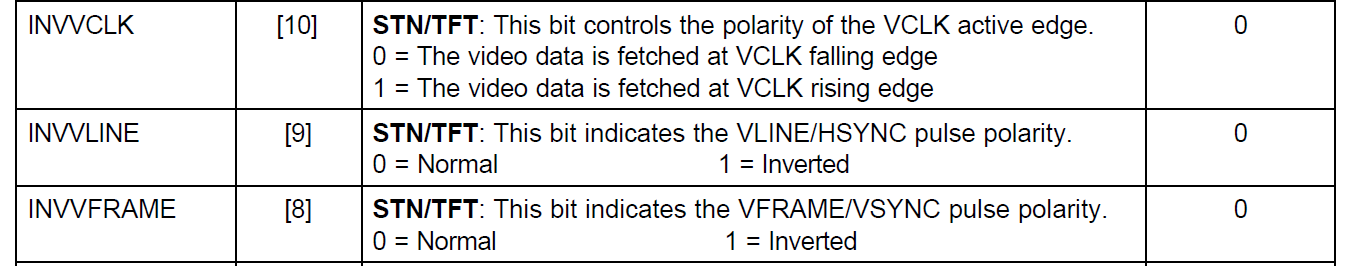
我们看一下，就需要设置HSPW，等于四就好了

这里还有个（13<<8）目前还不是很理解，但是如果不加上的话就会段错误

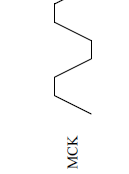
然后是lcdcon5

这个寄存器的每个位都有一些功能，我们需要设置的是12 10 9 8 6 1 0这几个位

我们来看看

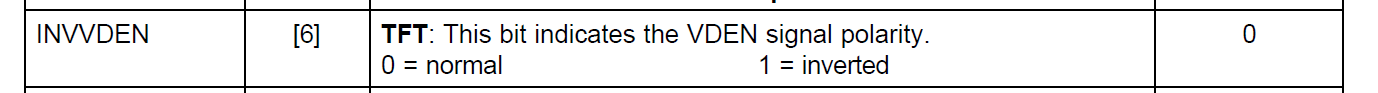


我们的VCLK是上升沿的所以我们10位置1



然后我们的HSYNC和VSYNC是没有的但是我们需要将它翻转

然后我们看6位



我们的VDEN是需要翻转的，我们可以看一下2440手册上的VDEN

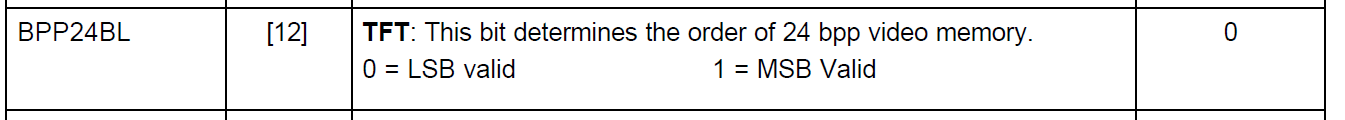


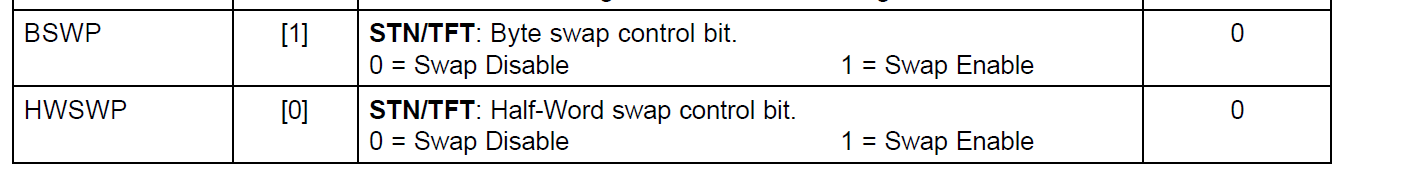
然后我们看一看LCD的VDEN



所以这个信号线需要反转

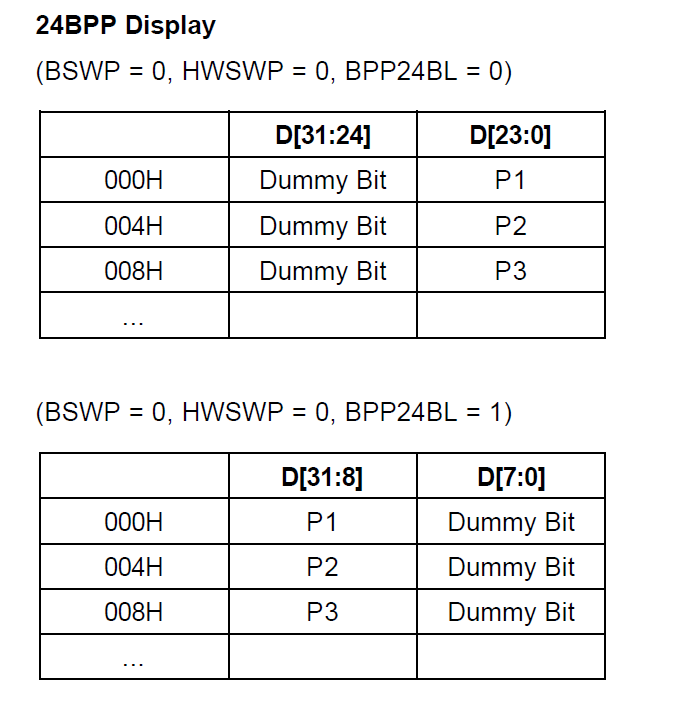
然后是1和0和12



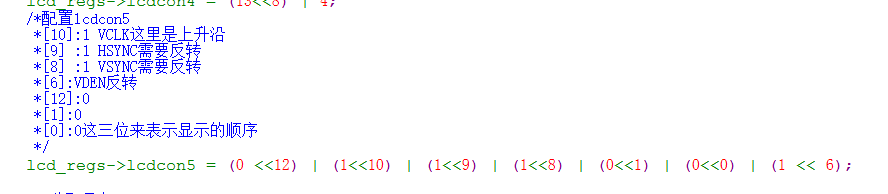


这是显存的存储格式的设置

我们看一下数据手册

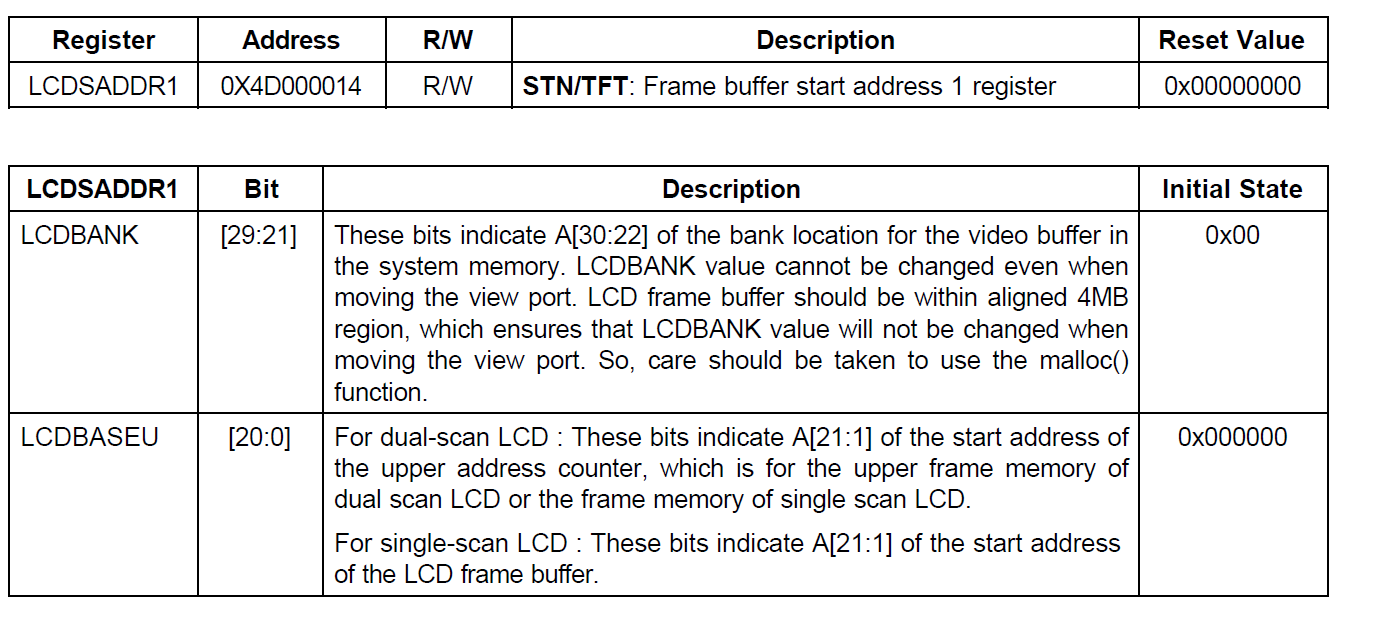


我们选择低位存储，这样是顺序存储，不用再做处理



然后是分配显存

在我们的LCD控制器中也有关于显存的寄存器

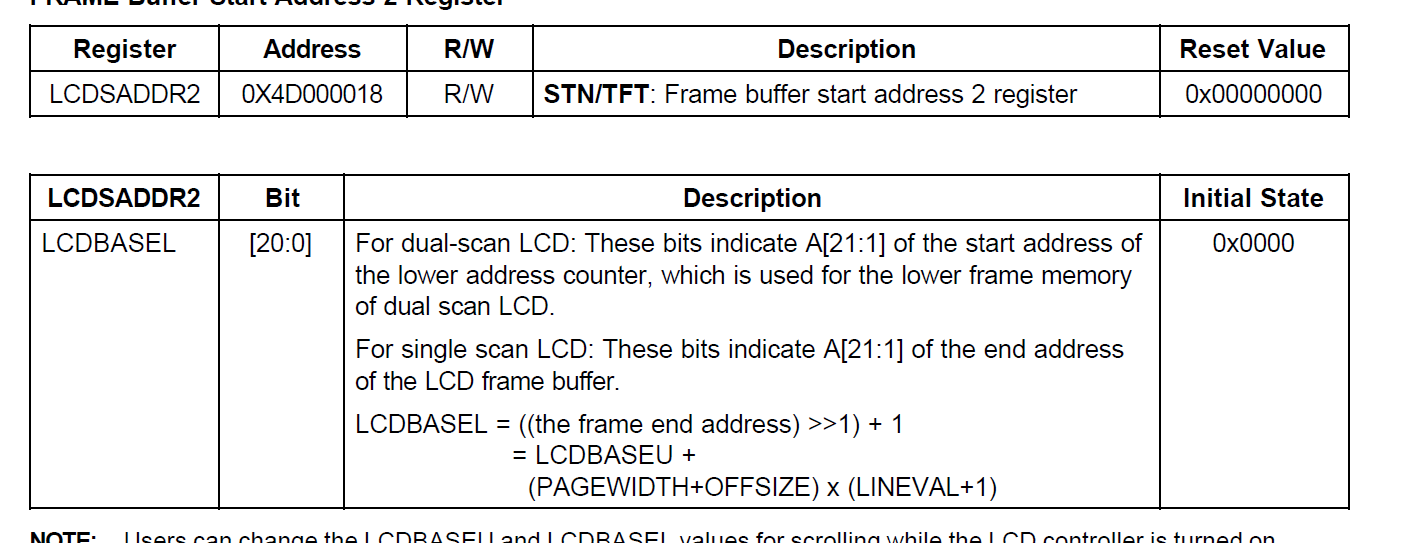


这个寄存器的描述的是显存用的是[30:1]而寄存器是[29:0]所以我们需要将显存的初始地址向右移动1位

因为我们不用前几位，所以我们可以将前几位清零

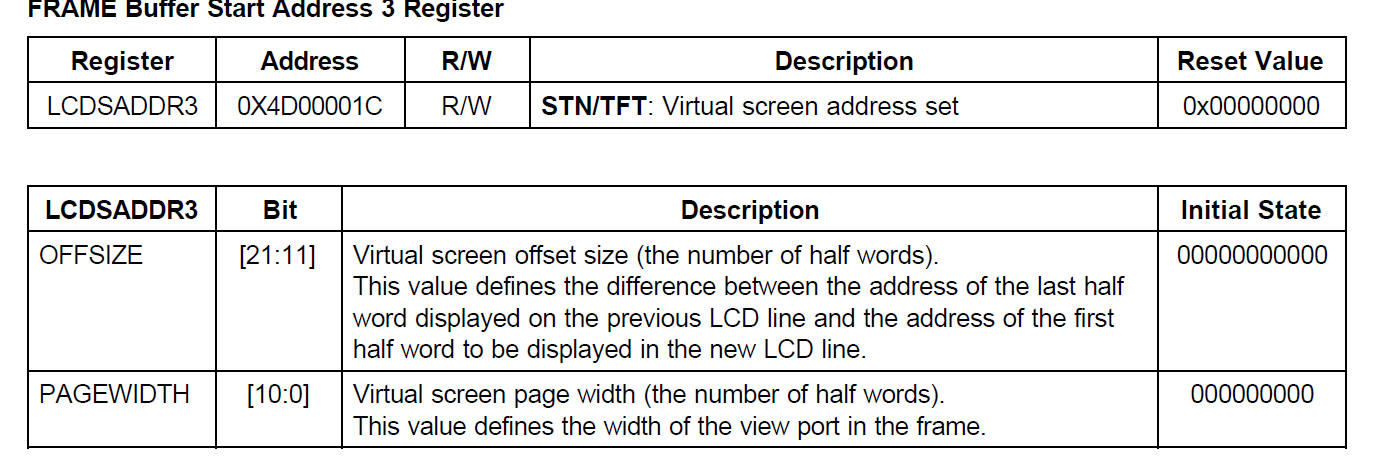


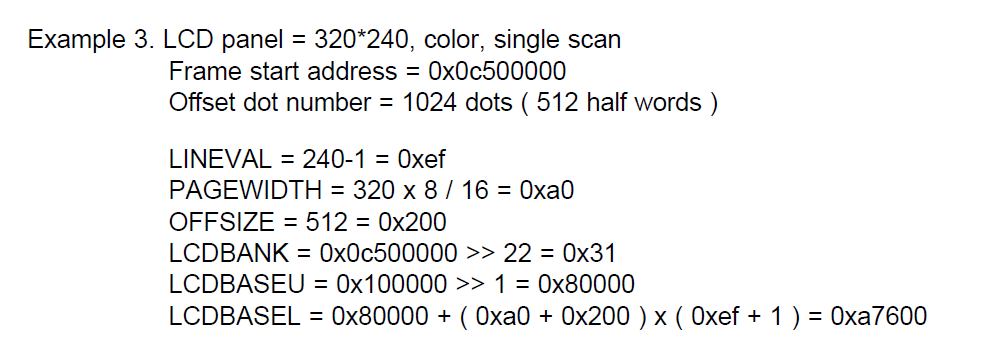
然后是



根据底下的公式就可以知道怎么做了







所以我们也知道怎么弄了

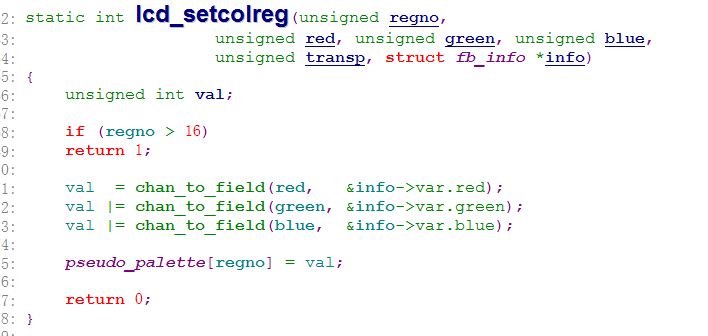


然后我们是需要设置调色板，这个调色板可以节省我们的内存空间，因为我们可以将原来的24位的像素可以弄成8位或者16位的，然后我们把真的颜色信息放在调色板里面，之后通过调用调色板来控制LCD的颜色

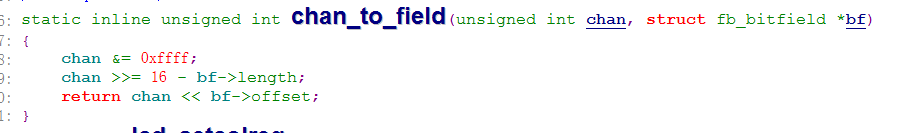




我们来看一下是怎么实现的

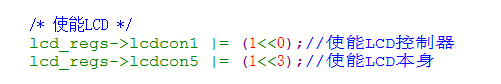


然后我们看chan\_to\_field是怎么实现的

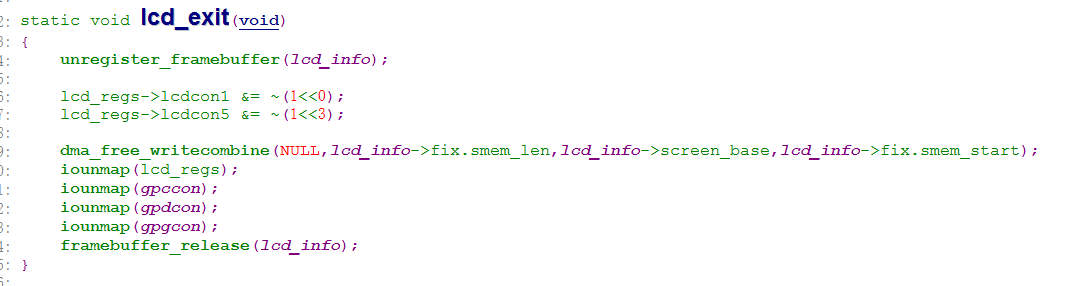


我们通过这个函数来算出颜色的位置，算出之后存入到调色板数组中，然后以后我们就可以调用这个数组就好了

最后我们要使能LCD，使能LCD需要两个地方，一个是使能LCD控制器还有一个是使能LCD本身



这样我们再写一个exit函数就可以了



然后我们的驱动程序就写完了，之后我们编译成.ko文件，这里我要说明的是我在3.18.26中并没有实现这个功能只是在2.6.22.6里面实现了，目前为什么在新内核中不好使还在研究，研究出来了会在移植内核中写到

我们先在老内核中make menuconfig

-> Device Drivers |

| -> Graphics support

中的2410LCD改成M模块加载，然后make uImage

之后再make modules

然后把video下的三个.ko文件复制到我们的文件系统中去

然后我们用这个编译好的内核使用，然后insmod三个ko文件，最后在insmod lcd.ko文件

这样我们就可以看到LCD亮了

然后echo hello > /dev/tty1就可以看到LCD上显示hello了

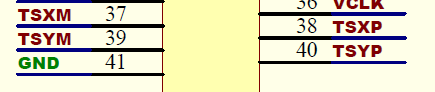
这样我们的LCD驱动就算写完了

# 触摸屏驱动

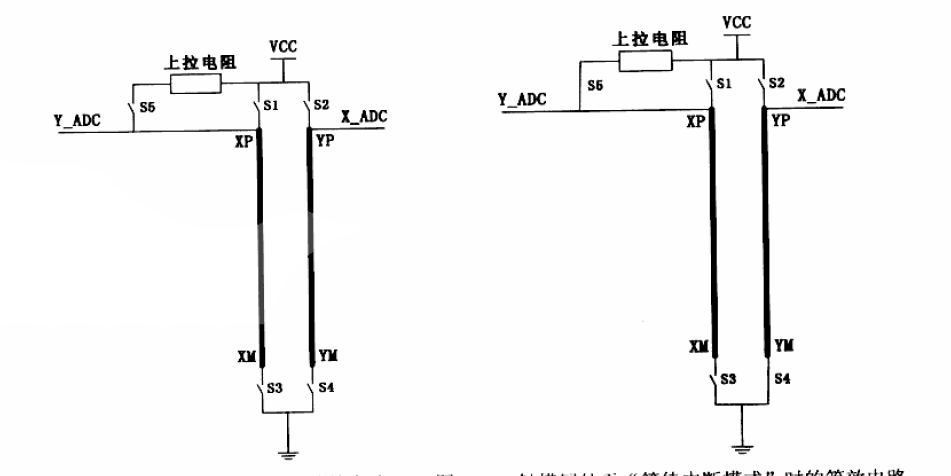
首先我们应该知道的是LCD和触摸屏并不是一个东西

它只是和LCD一样大，并且放到LCD上面而已

它的原理是有两层膜，一个是X膜一个是Y膜，分别通过ADC转换成电压值来获取绝对坐标



在我们的原理图中这四个引脚是触摸屏的引脚，一般就是两层膜，一个两头有XP XM另外一个有YP YM，



当有东西按下触摸屏时，将会触发中断，然后XP接高电平，XM接低电平，然后通过ADC测出X的电压值，同样地测出Y的电压，这两个值通过一种算法就可以得出触摸屏的坐标值了

但是我们还需要定时器来辅助，因为当我们长按或者滑动触摸屏的时候我们光是用中断唤醒然后开启ADC转换是不行的，因为这样我们滑动或者长按的时候只是检测一遍，那么我们可以利用定时器检测一下，也就是说我们检测到按下的时候先读取一下坐标，然后利用定时器延时一下然后再检测一下是否按下

我们先写代码

在我们的内核中也有触摸屏的驱动名字叫s3c2410\_ts.c，内核已经给我们写好了

但是我们可以自己写一个熟悉一下怎么写，并且可以参考一下这个代码

我们首先借鉴一下这个代码的头文件，然后首先写init和exit函数，

这个触摸屏是属于输入子系统的input

所以我们要定义一个input\_dev结构体



然后按照之前写的驱动程序

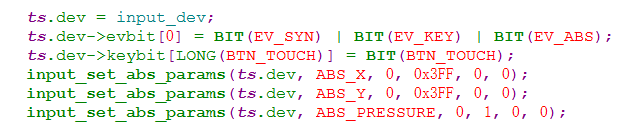
1. 分配空间
2. 设置结构体
3. 注册
4. 硬件相关的代码

首先分配input\_dev结构体空间

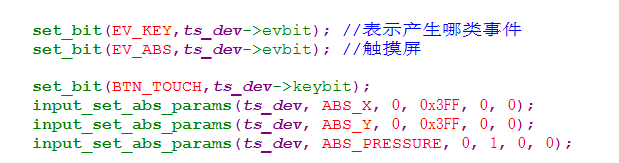


然后是设置这个结构体

我们可以参考一下s3c2410\_ts文件



我们就可以按照它设置

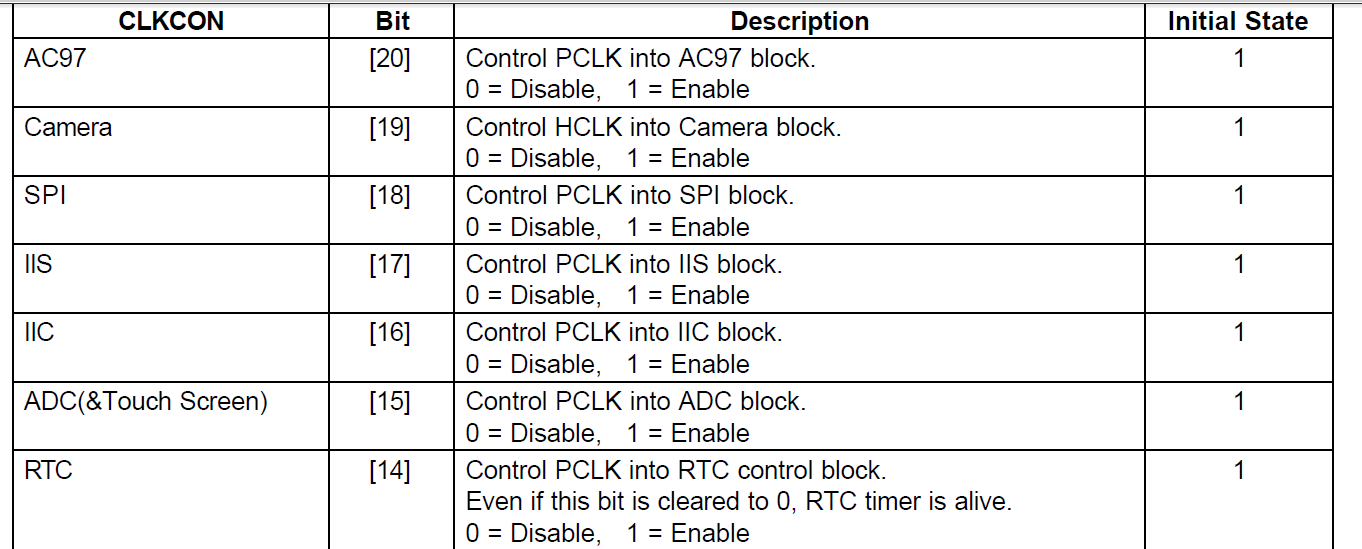


然后注册



然后接下来是硬件相关的操作

首先我们要使能ADC时钟并把其它时钟关掉，我们ADC时钟在CLKCON寄存器中



我们将ADC这个位使能就好了，我们参考一下s3c2410\_ts.c上面怎么写的



这里有个adc，我们搜索一下adc

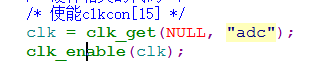
我们可以看到在clock.c中有这么一个结构体，我们点进去



这里面有个宏，就是将clkcon15置1的宏



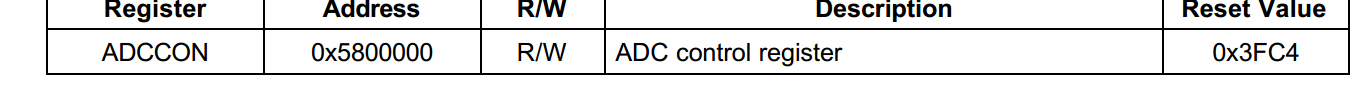
然后我们就可以按照它的写



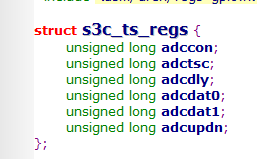
然后需要重新定义LCD寄存器

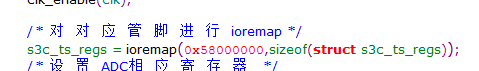
我们首先需要remap一下LCD寄存器

我们需要先看一下2440的数据手册



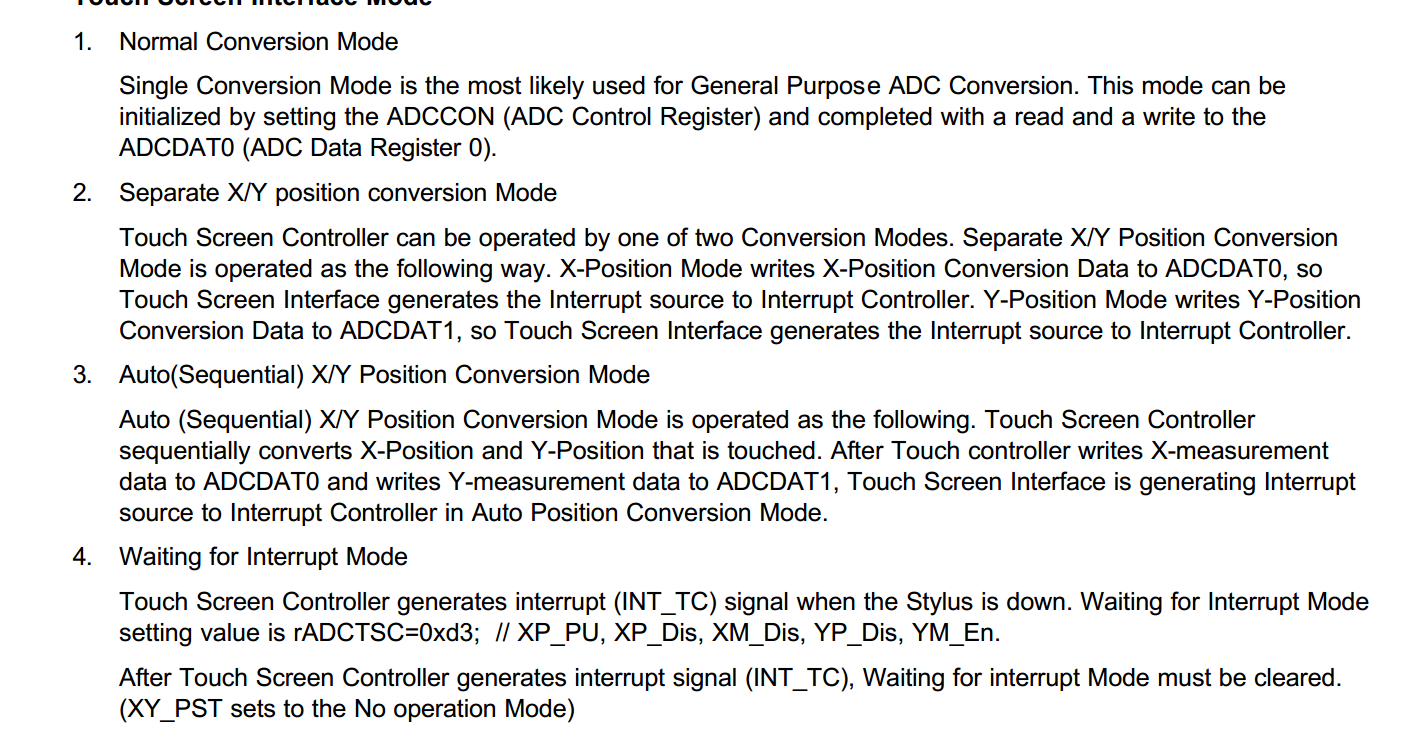
这是我们的第一个LCD寄存器的地址，我们跟以前一样，定义一个结构体然后仅仅需要remap第一个地址就好了





这样我们就把所有的LCD控制器都remap了

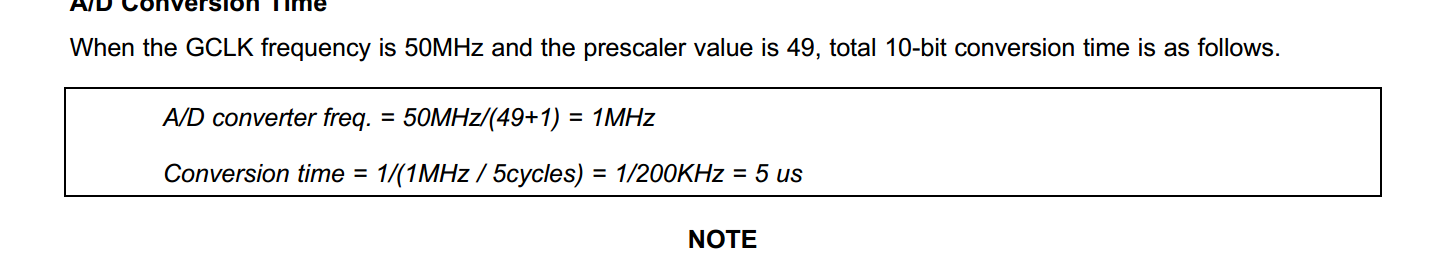
然后我们需要开始设置寄存器了，在设置寄存器之前我们需要先看一下2440的数据手册



在2440的数据手册上面一共有对触摸屏的4种设置模式，我们可以看出来触摸屏ad转换出来的值，x存放在ADCDAT0中y存放在ADCDAT1中



我们首先看一下15，它是只读位所以我们不用管它，然后是14，我们需要将它使能，然后是13：6这个是设置始终频率的，我们看下面说ADC的频率应该是要低于PCLK 5倍

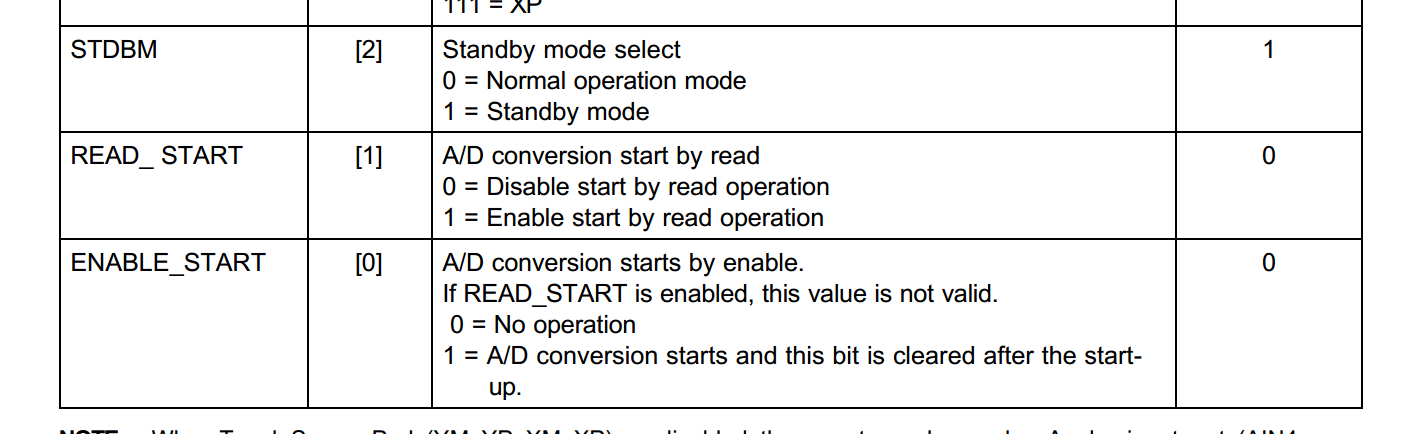


所以我们可以设置成49，因为我们2440的PCLK是50MHz所以我们这么算的话

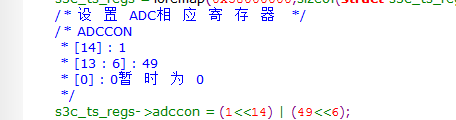
是1MHz最大只能是10MHz

所以这几个位的值是49

然后看下面的，因为我们是用比较简单的自动转换所以我们5：3不用设置



这几个位我们都不要设置，最后一个0位是使能ADC转换器的，我们暂时将它设置为0



然后我们的ADC基本设置已经设置好了，然后我们应该让它暂时只是实现一种功能，就是只要按下了就有打印

我们不能像按键一样一直循环检测是否按下，我们应该将它设置为中断等待队列

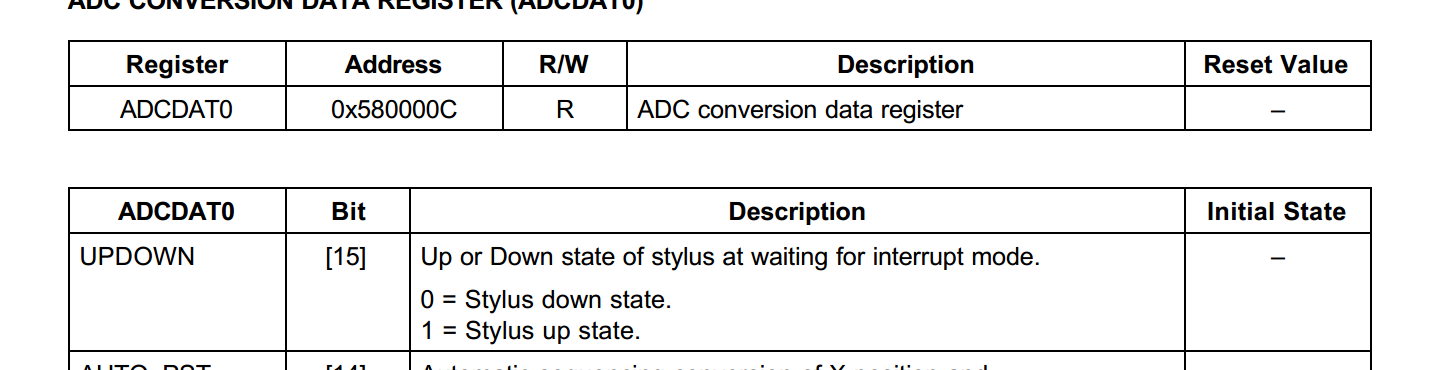
也就是说我们需要向内核中注册一个中断



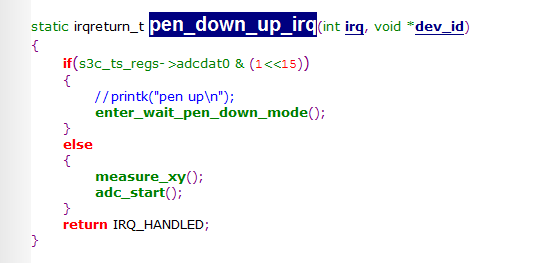
这个IRQ\_TC是触摸屏的类别

然后我们需要实现一下这个中断处理函数

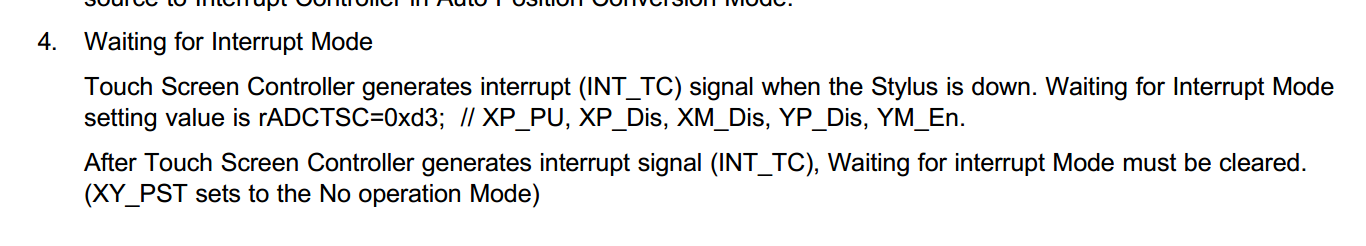
在ADCDAT0和1中的第15位都有一个判断按下和松开的位置



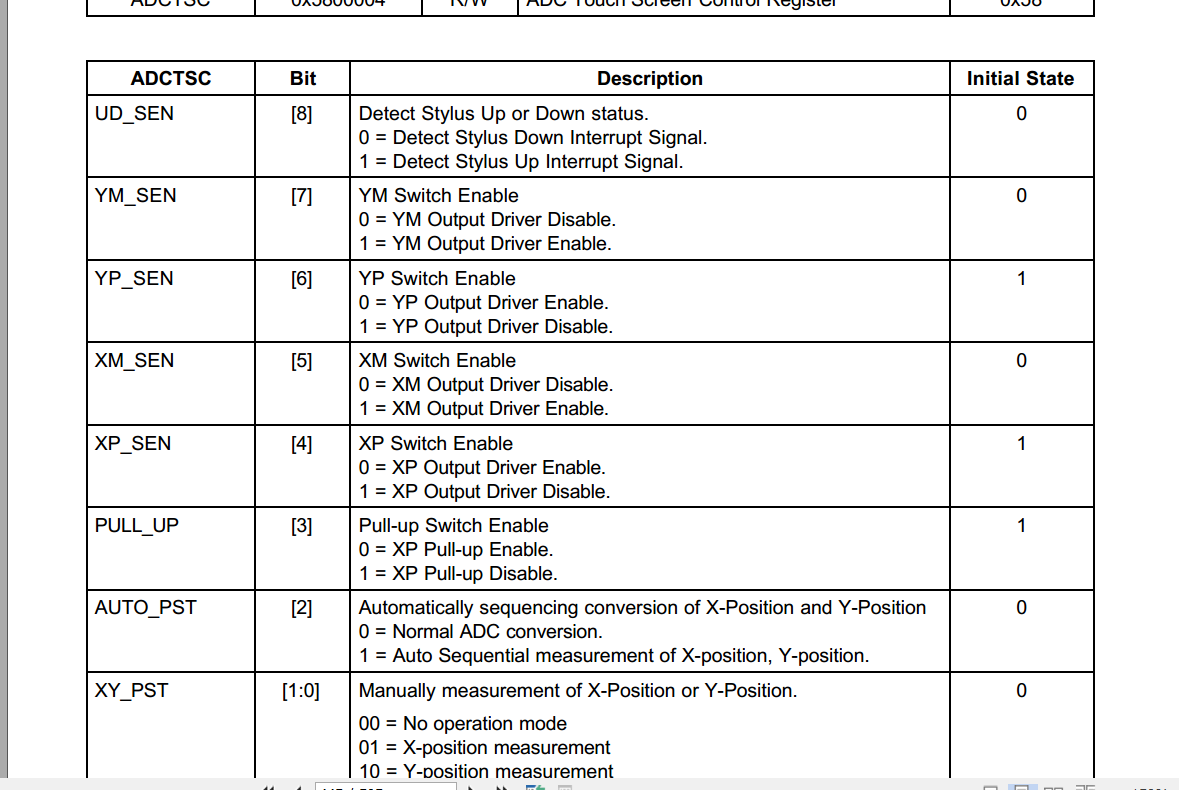
我们就可以利用这一点来编写中断处理程序



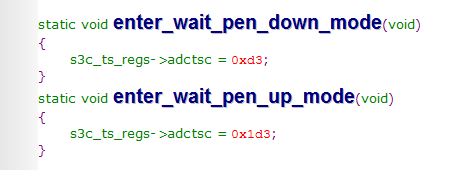
这样我们就利用上了这个寄存器，然后我们之前的模式4等待中断模式上面说的



将ADCTSC设置为0xd3我们可以先看一下这个寄存器



这里面的第8位是设置松开时等待和按下时等待的，所以我们需要有两个函数



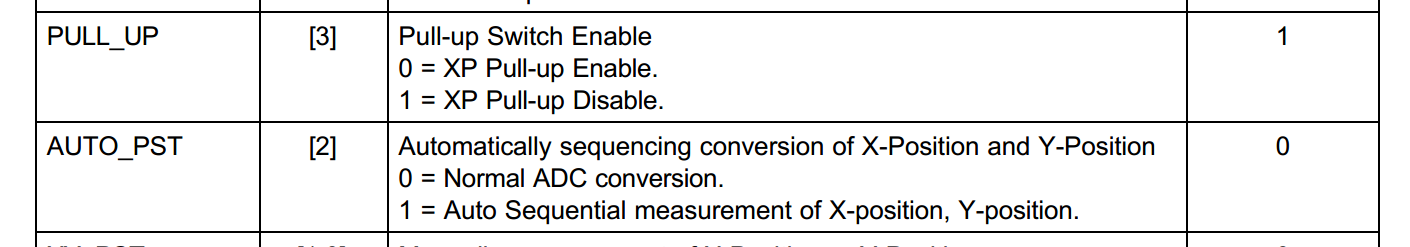
最后我们在init函数中加入首先要让它进入等待按下模式enter\_wait\_pen\_down\_mode函数

这样我们就可以放到服务器上面去编译了

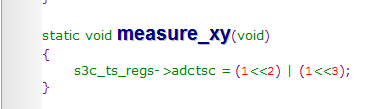
然后我们装载上按下屏幕就会有打印了

首先我们驱动起来了触摸屏，然后我们应该让它显示电压值

我们看一下ADCTSC寄存器的2 3 位



一个是上拉电阻使能另外一个是自动测量xy，我们需要将这两个位使能



测量出来的xy的电压值是存到了ADCDAT0和1中的低10位，然后我们还要使能ADC转换器，也就是上面说的ADCCON中的第0位需要置1

然后我们需要在init函数中注册ADC的中断函数



然后我们写一下adc\_irq

我们就暂时什么都不处理然后将adcdat0和1的低10位取出来之后打印就好了



然后我们rmmod之前的驱动然后我们重新insmod新的驱动，然后按下就会有xy的坐标值

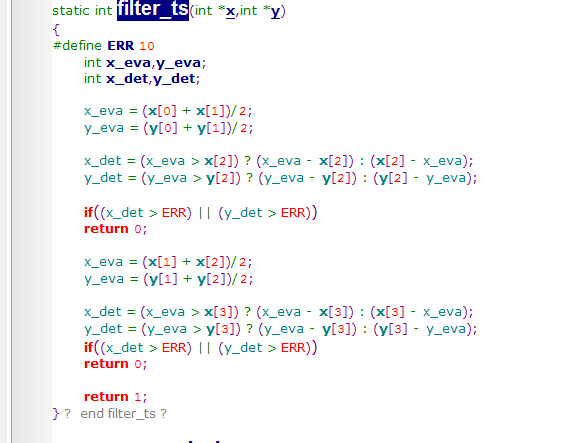
但是我们可以发现这个电压的误差值太大了，我们需要对其进行数据处理

首先我们定义两个数组x，y



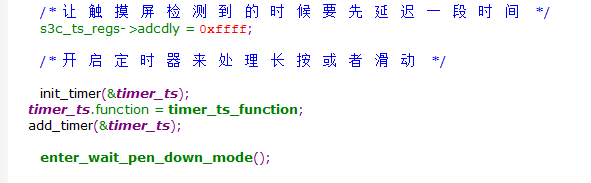
这两个数组是数据缓冲区，然后我们可以接收4次然后取平均值，但是光光这样还是不够的，我们还需要写一个软件过滤，也就是对x，y两个数组中的数据进行再一次的过滤，比如说首先前两个数据进行取平均值，然后这个平均值和第三个数据进行做差，如果差值小于一个标准，那么就可以放入到缓冲区中，如果大于这个标准那么就重新转换

以此类推





如果说在我们ADC转换完成之前松开手了，那么我们就重新检测，但是这样是不能检测到我们的滑动动作和长按动作的，那么我们需要定时器来辅助，我们最开始就说到定时器了，它是按下之后ad转换之后就开始进行延时然后如果还是按下那就继续检测就好了

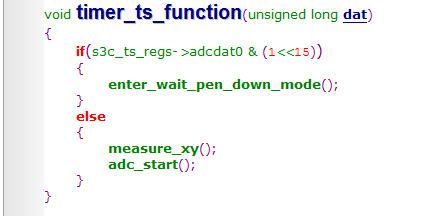


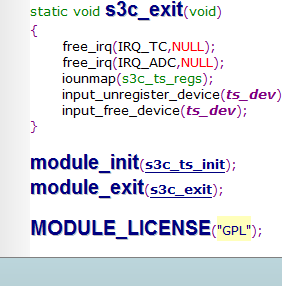
这是我们在init函数中加入的代码

第一个是一个寄存器，它是如果检测到有按下会延时一会然后再转换，这样会更精准，然后下面就是初始化定时器，我们之前需要定义一个timer\_list结构体指针



然后将定时器中断函数放到里面去，我们再看一下定时器中断函数



这跟正常的中断处理函数是一样的，这样我们就已经写好了一个完整的触摸屏驱动了

我们来重新说一遍

首先如果触摸屏有按下之后就会产生一个中断然后我们进入pen\_down\_up\_irq

然后如果按下了就进入测量模式并且打开ADC转换器然后就会进入ADC中断

Adc\_irq函数，在这里我们进行一些数据处理，并且给定时器一个时间初值，如果时间到了进入定时器中断函数然后同样进行检测

我们可以安装tslib进行测试，这里安装过程就不说名了，网上一大堆的教程

# USB驱动

USB支持四种基本的数据传输模式：控制传输，等时传输，中断传输及数据块传输。每种传输模式应用到具有相同名字的终端，则具有不同的性质。

**控制传输类型**

支持外设与主机之间的控制，状态，配置等信息的传输，为外设与主机之间提供一个控制通道。每种外设都支持控制传输类型，这样主机与外设之间就可以传送配置和命令/状态信息。

**等时（lsochronous)传输类型**

支持有周期性，有限的时延和带宽且数据传输速率不变的外设与主机间的数据传输。该类型无差错校验，故不能保证正确的数据传输，支持像计算机－电话集成系统（CTI）和音频系统与主机的数据传输。

**中断传输类型**

支持像游戏手柄，鼠标和键盘等输入设备，这些设备与主机间数据传输量小，无周期性，但对响应时间敏感，要求马上响应。

**数据块（Bulk)传输类型**

支持打印机，扫描仪，数码相机等外设，这些外设与主机间传输的数据量大，USB在满足带宽的情况下才进行该类型的数据传输。

USB采用分块带宽分配方案，若外设超过当前带宽分配或潜在的要求，则不能进入该设备。同步和中断传输类型的终端保留带宽，并保证数据按一定的速率传送。集中和控制终端按可用的最佳带宽来传输传输数据。

PC机的USB口的内部的数据传输的两个引脚D+ D-都接有上拉电阻和下拉电阻，平时默认是低电平，当我们插入USB设备的时候上拉电阻D+或者D-那个电平被拉高了这样通过硬件来通知PC机来告诉是否有USB设备来插入

每个USB设备都至少会有一个端点0，它用于控制传输，可以双向传输，但是除了端点0以外其它的端点都只能单向输入输出

我们现在说的输入输出都是以PC为主语的

USB设备都只能被动被读取，它没有主动读取的功能，比如说我们的鼠标和键盘就是每次产生一个数据之后只能等待主机来读数据，**但是注意一点，鼠标键盘都是属于中断传输类型**

所有USB在未分配地址前的地址全都是0

USB的数据需要由它的USB设备驱动程序来解析，然后传到USB总线驱动程序上才行

## USB总线过程

要写USB驱动我们需要先了解USB在系统中是怎么处理的，当我们用电脑的时候我们插入一个USB设备的时候，我们就会看到电脑会先检测USB设备，然后安装驱动程序，然后才能使用

我们先来看一下流程

1. 插入设备让系统进入中断
2. 然后系统读出USB设备的信息
3. 如果找到对应的驱动程序就自动安装驱动程序，如果没有就不安装，等待手动安装

USB主控制器分三种

UHCI （Intel）

OHCI （Microsoft）

EHCI

OHCI主要为非PC系统上以及带有SiShe ALi芯片组的PC主板上的USB芯片

UHCI大多为Intel和Via主板上的USB控制器芯片。UHCI的硬件电路比OHCI简单，成本第，但驱动复杂。但他们都是由USB1.1规格的。

EHCI是有Intel等几个厂商研发，兼容OHCI UHCI 遵循USB2.0规范。

我们2440在内核中用的是OHCI，我们在运行2440跑系统的情况下插入任意一个USB设备，我们看看会有什么反应

usb 1-1: new full speed USB device using s3c2410-ohci and address 2

usb 1-1: configuration #1 chosen from 1 choice

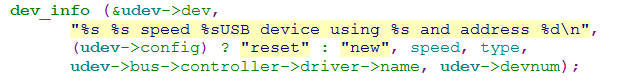
scsi0 : SCSI emulation for USB Mass Storage devices

首先会打印这几句话

我们看一下第一句话

USB device using，我们在内核中搜索一下这句话看看在哪里

我们搜索一下结果在drivers/usb/core/hub.c:2186:中



这句话是在hub\_port\_init函数中，我们再看hub\_port\_init函数被谁调用

hub\_port\_connect\_change，在这个函数里

hub\_port\_connect\_change在hub\_events中

hub\_events在hub\_thread中

hub\_thread在usb\_hub\_init中

这里是thread中的一个需要休眠的结构体，然后我们看一下是谁唤醒了这个结构体

kick\_khubd这个函数唤醒了，然后是hub\_irq调用

USB总线驱动程序主要是

识别USB设备

分配地址

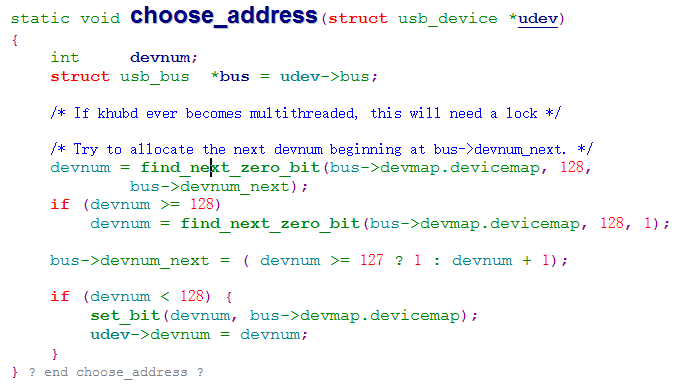
将分配好的地址告诉USB设备

发出命令描述符

我们从hub\_port\_connect\_change开始看

我们现在首先已经识别了USB设备了，然后我们需要分配地址，我们看一下hub\_port\_connect\_change函数中有这么一句话





我们看一下这个函数中的内容，它这里的意思就是首先要寻找下一位的0位，如果没找到就返回到第一个地址继续寻找，我们从这里就可以看出可以插入的USB设备最多是127个，地址是1到127

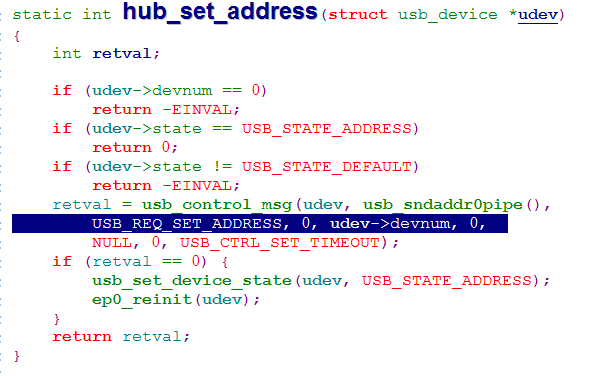
然后我们再看hub\_port\_connect\_change里面的hub\_port\_init函数，我们现在分配完地址了，但是我们还是没有看到把地址告诉USB设备的代码，我们看一下init函数，中有这么一句话



在这句话底下有一个宏，看样子是设置什么东西，我们再看一下这个宏附近都有什么



它的上面就是设置地址，我们搜索一下这个宏



结果我们搜索到时再这里被调用了

我们再看这个函数是被谁调用的

我们发现它也是在hub\_port\_init函数中被调用的

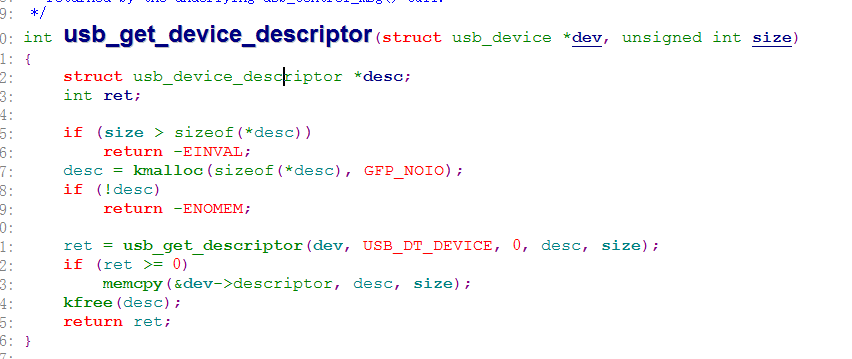
所以这个函数就是把地址告诉USB设备的语句

然后我们再往下看

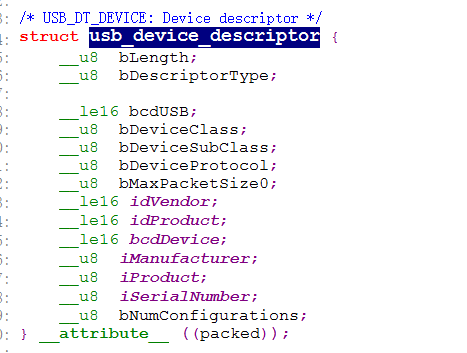


这个接下来就是获得设备描述符了

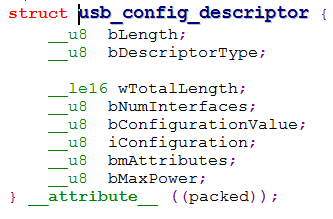
我们点开设备描述符函数看一下



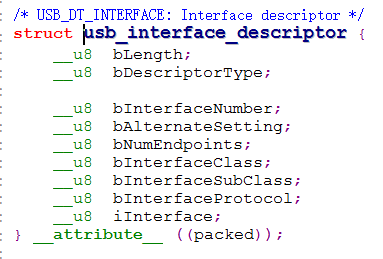
然后这里有定义设备描述符的，我们点开看一下



这里是定义设备描述符的地方，这里面bNumConfigurations是代表配置描述符的个数，然后我们看看配置描述符

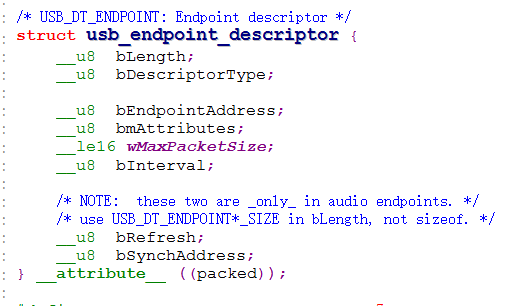


然后这里面有接口描述符



我们USB设备一个设备中可能有很多逻辑设备比如说一个音频设备里面有录音功能和播放功能，所以这就是有两个逻辑设备，我们的接口描述符里面有记录这些设备的个数，这里还有端点描述符的个数

我们USB设备的收发数据最根本的其实就是跟对应端点收发的，每个USB设备都是要至少有一个端点0的，只有这个端点是可以双向收发的，其它的端点只能是单向的



端点描述符包括能发送多少数据，传输类型，编号之类的信息

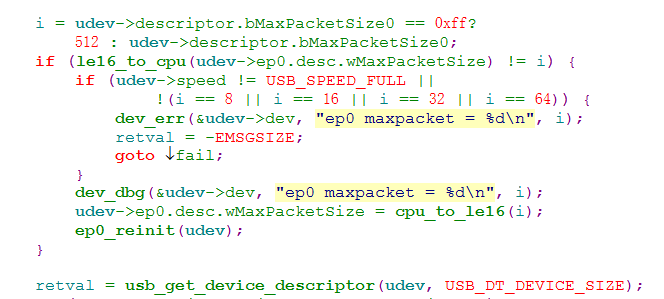
我们再回来



这里只是读了8个数据，因为我们现在还不知道这个USB设备的信息有多少，所以我们要首先读取8个字节将它们的信息都读出来



然后第一次读取之后获得信息之后，然后设置成员，接下来再次读取一遍



我们再回过来看看hub\_port\_connect\_change函数，我们之前说到了分配地址和设置设备描述符，然后我们继续往下看



有设备描述符了有地址了然后我们就可以创建新的设备了跟我们原来总线设备驱动是一样的

我们点开usb\_new\_device函数



这个函数是把所有的描述符都读出来

我们回到usb\_new\_device函数中



这个函数的功能就是把udev结构体放入到总线的dev链表中，然后从总线的driver链表中取出driver，一一比较，如果能匹配就调用这个driver的probe函数

到这里就跟我们的之前那一套就一样了

到这里总线的过程就是这样

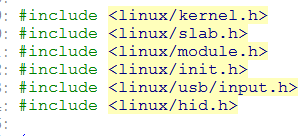
## 编写USB驱动

我们现在就可以编写一个USB驱动了，由于鼠标就是一个USB设备所以我们就先拿鼠标的驱动为例来写一个驱动

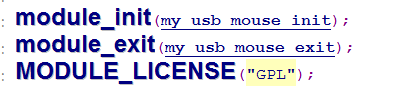
首先内核中本身就有一个鼠标的驱动，如果我们自己写的话就需要把内核中的鼠标驱动make menuconfig中去掉鼠标驱动，鼠标驱动就是HID设备，并且我们也可以按照这个程序作为参考来写

我们首先看一下那个例子

我们首先需要用它的头文件

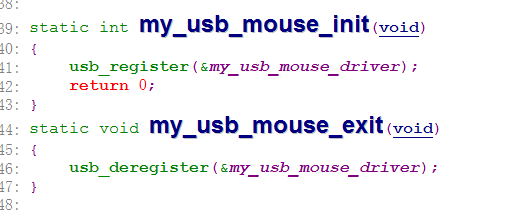


然后写它的入口函数和出口函数，以及修饰



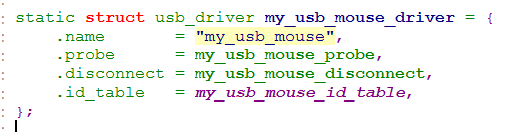




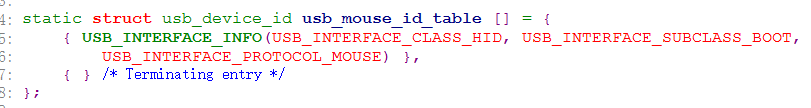


入口函数用上注册，出口函数用注销

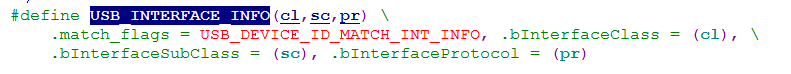
然后我们看一下，这个东西和我们原来的驱动分层的概念是一样的，首先要对比一下id，然后如果匹配的话调用probe函数，所以我们需要写一个usb\_driver的结构体



然后我们一个一个地写，我们首先要写id\_table，我们可以参考原来的内核的例子



，我们就可以直接复制过来，我们也可以点进去看看是什么



这里就是一些比较函数

然后我们回到代码，我们开始写另外两个函数，另外两个函数一个是处理函数，另外一个是不连接函数，也就是说当我们拔出USB的时候或者有什么突发状况把USB弄掉了的时候就会调用这个代码

我们首先就是要把它的功能写的简单一点，就是打印一点东西，我们注意一下这两个函数的参数

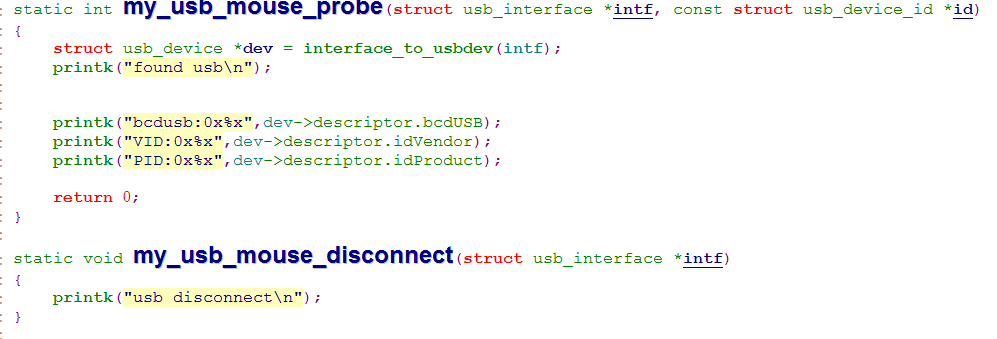




我们可以看到，除了内核中自带的usb\_device结构体，然后传进probe和disconnect函数的结构体是上面介绍的三个结构体的接口结构体，因为接口结构体包含逻辑结构体，所以如果这个设备有两个逻辑设备的话我们就要写两个处理函数，我们可以看一下参考程序的probe函数，然后我们看一下



这个就是获得设备描述符的东西，然后我们我们就可以用



然后我们可以编译加载模块然后我们插入就有反应了

然后我们现在继续完善USB驱动程序

我们现在的probe函数只是让它打印东西而已，我们现在来让它识别USB设备的数据，我们的鼠标是我们的输入子系统，那么就离不开那几步

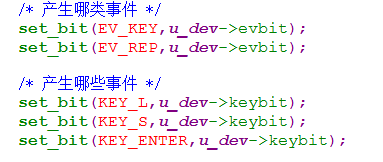
1. 分配input\_dev结构体
2. 设置
3. 注册
4. 硬件相关代码

我们先把分配和注册都写完





然后是设置input\_dev结构体



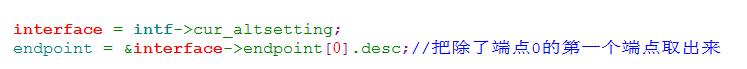
然后就是我们的硬件相关的代码了

USB数据传输有三要素：源，目的，长度

源我们可以参考内核自带代码



这里是定义了一个宏里面有USB传输的类型、以及输入输出的方式还有被分配到的地址的编号以及是第几个端点，我们在之前还得加上一个设置端点的代码



这个放在probe的最开始就好了

然后我们继续写，我们的长度



长度就是端点里面的最大数据包

以及我们的目的，目的其实就是虚拟地址

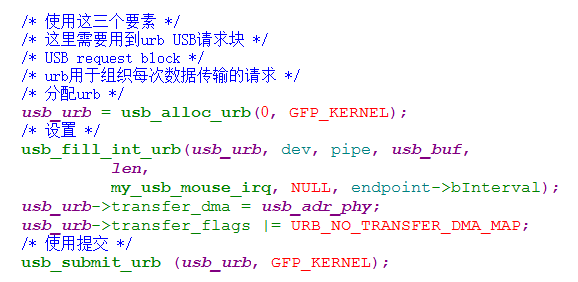


这个函数也顺带帮我们设置了物理地址所以我们还要一个物理地址的变量

然后这三个要素设置完了我们就得需要开始使用这三个要素了，USB传输还得需要一个数据请求包，urb（usb request block）

它用于组织每次数据传输的请求

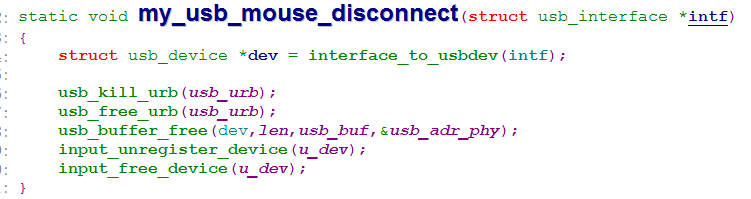
我们可以参考内核的代码



然后这里面还有一个中断处理函数，然后我们开始写一下中断处理函数



然后我们还剩下最后一个退出函数



我们的驱动程序就写完了

# 块设备驱动程序

我们首先需要了解一下块设备跟字符设备有什么区别，像我们的硬盘，内存等等都是块设备，首先它的硬件结构是一个小盘，然后有一个磁头然后用磁头来读写，一个块设备不会只有一个小盘，它会有很多个小盘，如果我们按照普通的字符设备来读写的话，首先它的读写方式是以字符为单位的，而我们的块设备是以扇区为单位的如果我们要读写的话就要对其进行优化

然后它的扇区顺序不是对齐的我们还需要进行优化

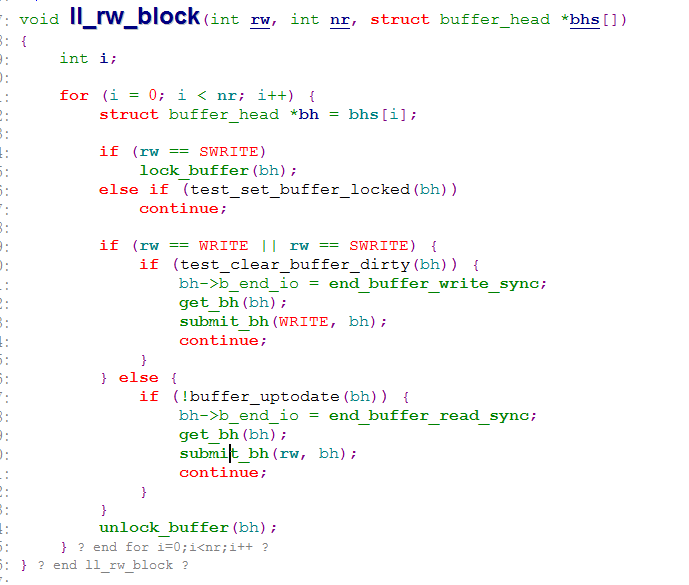
而且如果顺序不一样的话会对硬件造成损坏，所以我们如果要写块设备的驱动程序需要对这种读写方式进行优化

## 块设备驱动程序的框架

我们再写一下整个块设备驱动程序的框架

首先应用程序层要读写一个文件需要用到open read write这几个函数，然后应用程序通过文件系统将文件的读写转换成为了扇区的读写

然后扇区的读写需要用到一个ll\_rw\_block函数



我们首先来看一下它的参数，rw是代表读还是写，nr代表的是数量，后面的buffer\_head是装的数据传输三要素：源、目的、长度

然后我们可以看到这是一个for循环，也就是将这些数量的buffer\_head放入到bh结构体中去

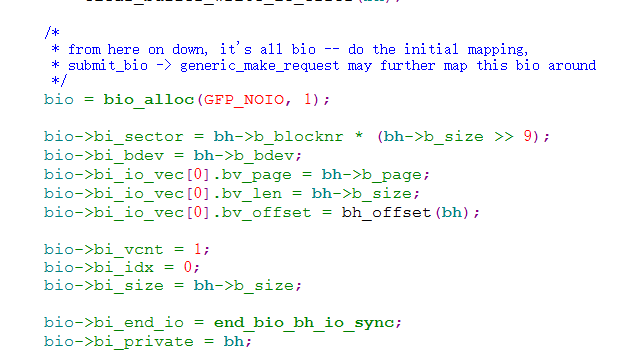
最后用submit\_bh提交一下

我们点进去submit函数看一下

在这里面有一个bio( block input/output )



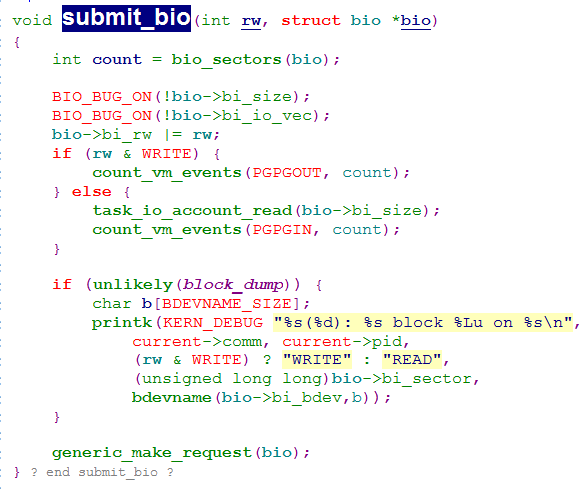
这个函数主要就是利用buffer\_head构造一个bio结构体



最后再提交bio



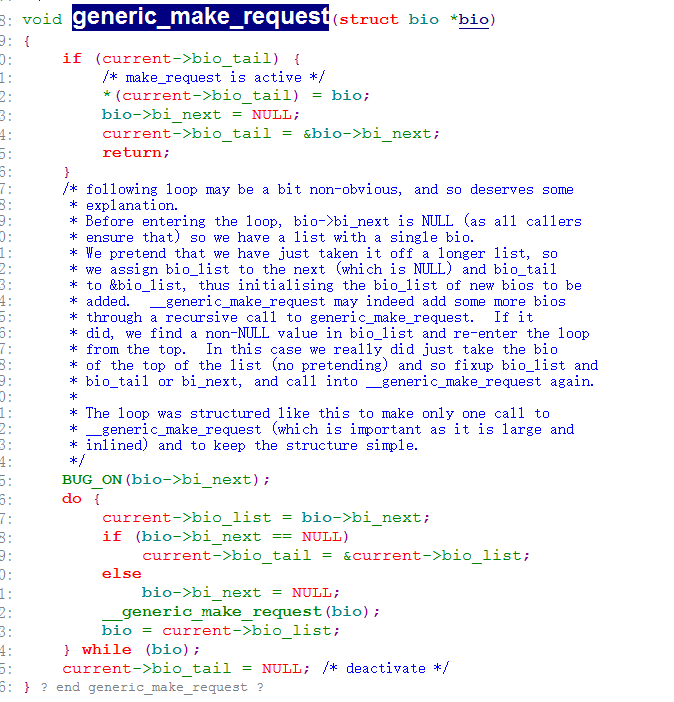
我们点开这个bio



在这个函数的最后一句上面写到

generic\_make\_request(bio);

通用的构造请求，我们点进去看看



这个函数的本体应该是\_\_generic\_make\_request，我们点进去看一下



这里有这句话，找到请求队列

然后在这个函数中有一句话是这么说的

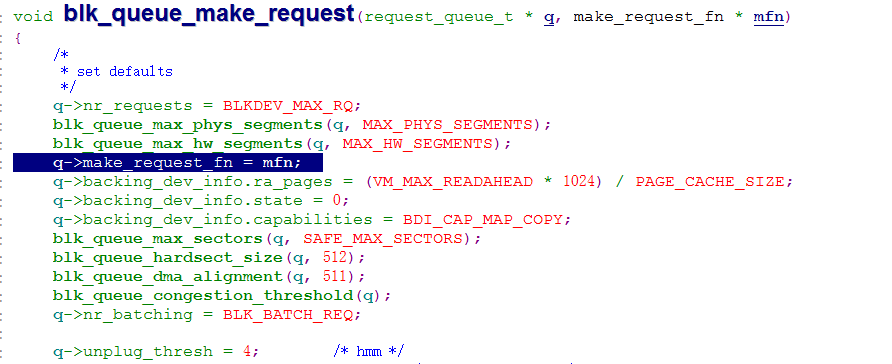


这句话是调用请求队列里面的一个make\_request\_fn函数，我们来搜索一下这个函数在哪里被设置了

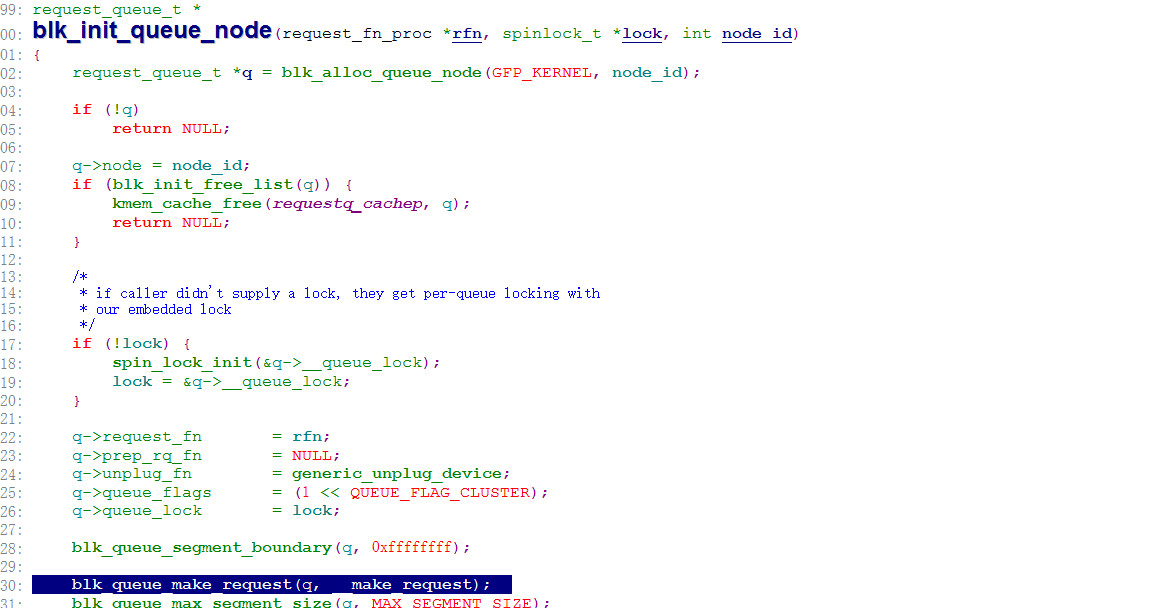
我们搜索到



这个函数是在这里被设置的，我们点开它看看

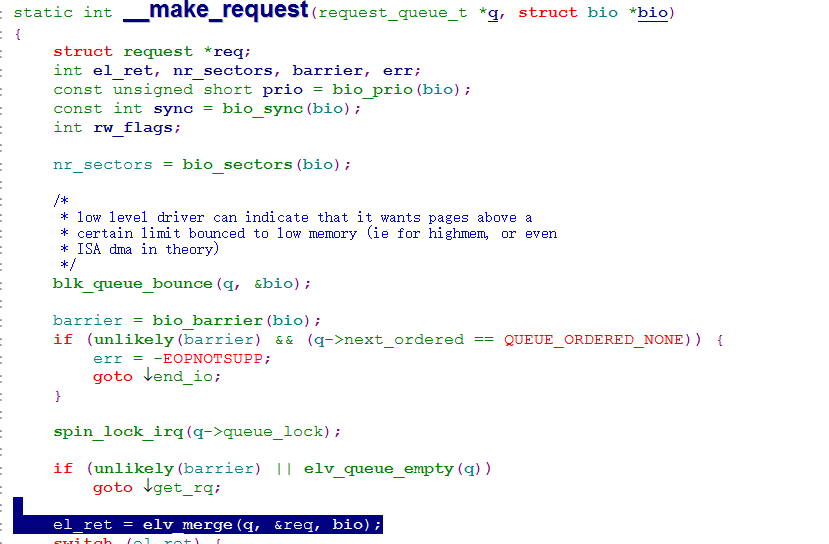


发现是被这个函数调用的，我们再搜索这个函数是哪里被调用的



我们搜索到是这份函数调用的它

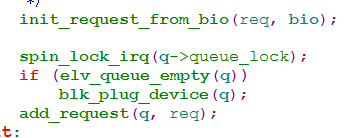
这个的参数是\_\_make\_request，我们看一下这个函数，这个函数是默认的制造请求函数



我们看一下这个elv\_merge函数，是电梯合并算法

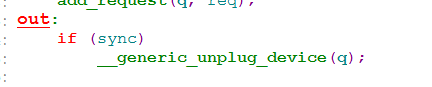
这个原理跟我们的电梯的原理是很像的，也就是说我们这里有好几个楼层的人在各自的楼层里面，这里面有上楼的也有下楼的，比如说上楼的有1 4 6层，其中5层的人是要下楼的，电梯就会首先从1楼开始上，然后经过5楼的时候电梯是不会停的，直到电梯把所有上楼的人都送完了然后才会去下楼接5楼的人，这就电梯算法，因为我们之前说过块设备是有好几个磁盘的，这样就是先把所有需要读的都读出来，然后再把写的都写进去，这样就跟电梯是一样的，因为先把上楼的都送上去再都去送下楼的

我们继续看，这里是首先尝试合并，如果合并不成功那就执行下面的函数

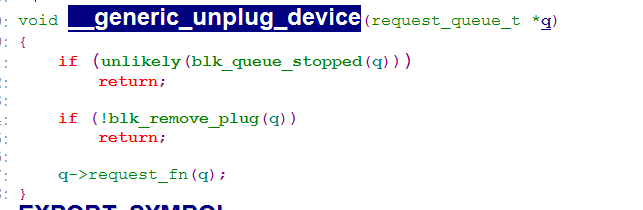


这里就是使用bio直接构造请求，然后将请求写入队列中去，在这里就已经将处理进行了优化了

然后接下来就是执行这个队列



这里面这个函数并不一定是一定要执行它，只是其中的一个入口，我们点进去这个函数看看



这里面有个request\_fn这个函数就是执行函数

这就是我们块设备驱动的框架

## 编写一个块设备驱动程序

我们编写一个块设备驱动程序需要有以下几步

1. 分配一个gendisk
2. 设置

2.1、分配/设置一个队列 request\_queue\_t 这个队列给它读写的能力

blk\_init\_queue

2.2、设置gendisk的其它信息

1. 注册

我们知道了怎么写之后我们就可以开始写块设备驱动程序了

这个程序是我们当做练习来写的，我们把内存当做块设备存储

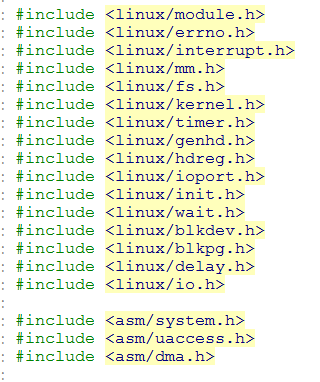
首先我们需要在内核中搜索blk\_init\_queue函数

我们看到



我们就可以参照它来写程序

我们可以参照这个文件所包含的头文件



然后分别写init函数和exit函数以及它们的修饰符







然后我们按照我们之前列出来的几点写程序

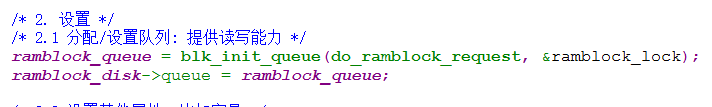
首先就是分配一个gendisk结构体



alloc\_disk函数的参数是分配之后的次设备号的个数

分配完结构体之后就是设置结构体

首先是分配设置队列，提供读写能力

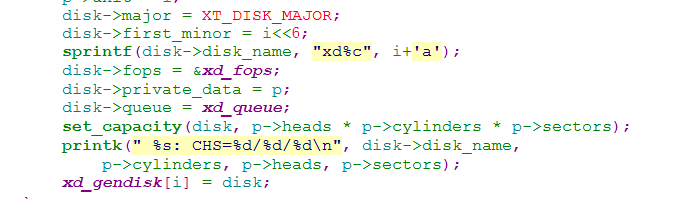


这里我们按照示例程序来写参数，这里面有个do\_ramblock\_request函数，这个函数是我们自己写的处理读写函数我们先把它放在一边一会再写

后面还有一个ramblock\_lock这个我们按照参考程序来定义

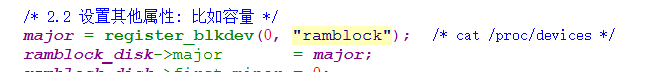


接下来是设置gendisk的一些其它信息，这里我们按照刚才搜索到的程序来写



我们可以把这段复制下来我们自己来修改

首先是主设备号，我们让系统自动给我们分配



然后是第一个次设备号，我们写成0



然后接下来是fops，我们也按照示例程序写，我们可以不再fops中不放东西，但是不能不定义，如果不定义是不好使的



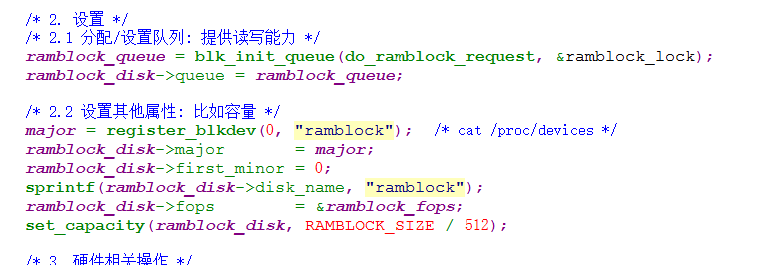


最后是容量，在内核中都是以512字节为单位的

所以我们这么写



这些就是第二步设置

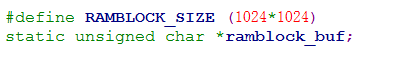


然后是第三步，硬件相关的操作

这次的硬件相关操作很少



我们需要来一个宏定义



然后是最后一步的注册



然后接下来是写处理函数

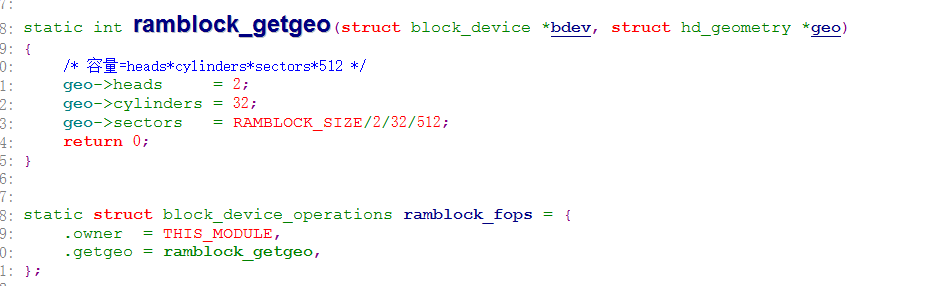
这里我们也仿照内核中的代码来写



我们为了兼容老的分区工具我们需要再写一个函数，来仿造一个信息，也就是说要有磁头扇区

我们需要在fops中加上getgeo函数

这个函数也在内核中的代码有



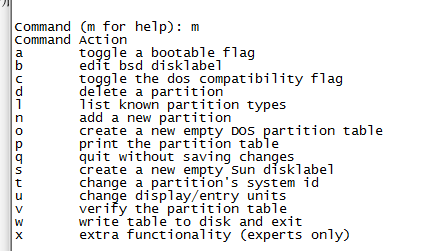
这里需要注意的是前两个磁头以及柱头随便写的话扇区就要计算一下了

总容量 = 磁头数量 \* 扇区数量 \* 扇区大小 \* 512

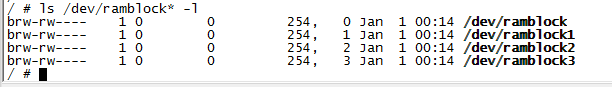
然后我们就可以生成ko文件了然后我们可以用fdisk命令来给设备分区



我们输入m就可以看到帮助



分完区后我们再看/dev目录下就会多出来几个



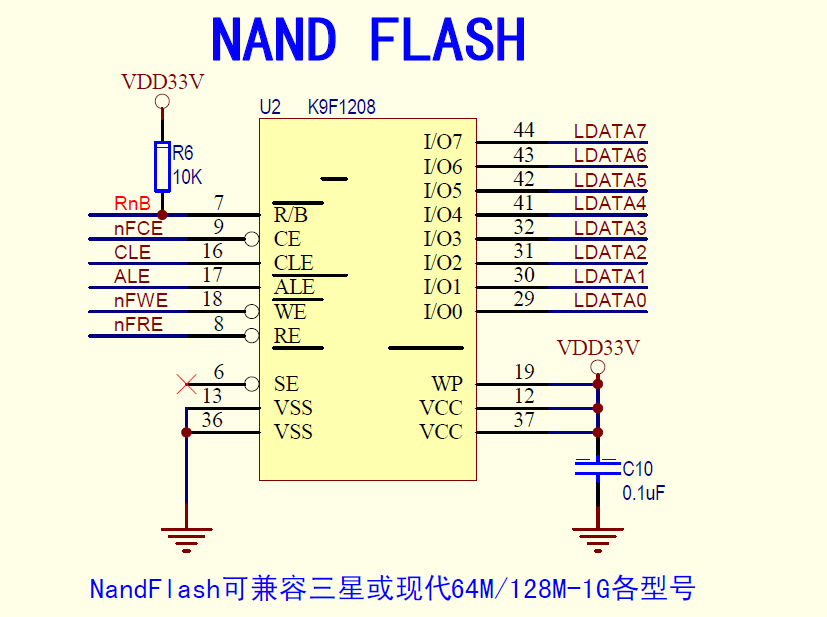
然后我们可以用mkdosfs命令来格式化它们，然后接着用mount命令挂接到任意目录下进行操作就可以了

格式化: mkdosfs /dev/ramblock

挂接: mount /dev/ramblock /tmp/

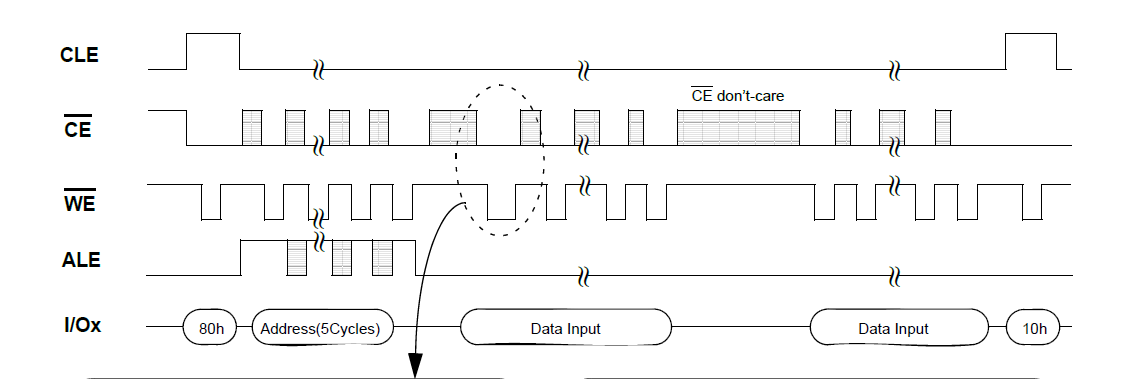
# NAND FLASH

Nandflash是一个真正的块设备，我们首先看一下nandflash的原理图



我们看到在左边有很多信号线，右边是数据线，而我们可以看到在别的地方也会有数据线，所以我们需要借用左边信号线中的nfce引脚来启动片选，这样8个data位就是只给nandflash用了，但是就8条数据线怎么知道发过来的是地址还是命令还是数据呢，当ALE为高电平时就是发送地址，当CLE为高电平时发送的就是命令，当两个都是低电平的时候就是数据

我们打开我们自己的nandflash的芯片手册我们可以看一下时序图



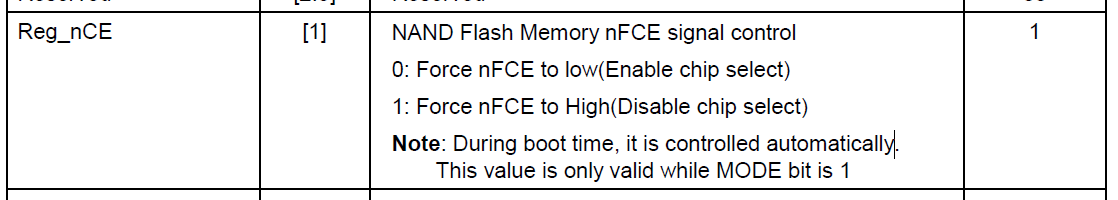
这个是我们的nandflash的其中的一个读数据的时序图

首先我们看CE位是需要一直拉低的，因为我们需要选中芯片，然后我们再看CLE引脚，这个引脚是之前说的命令控制引脚，我们看IO位首先发出的是0x80，这时候我们看CLE是置于高电平的，也就是说这个0x80是一个命令，然后接下来我们看到后面发的是address这时候ALE引脚是高电平，然后我们看一下在刚才写的时候WE引脚是发生脉冲的，同理，当我们读数据的时候RE也是需要发送脉冲的

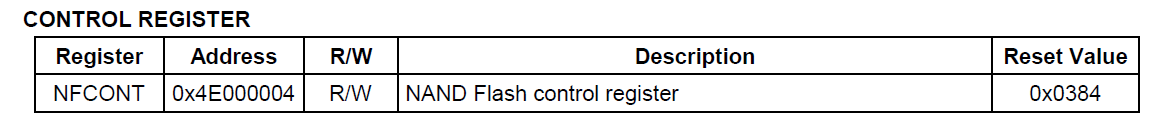
RnB是状态引脚

我们把所有引脚都看完了，然后我们看2440的nandflash控制器

我们首先看这里

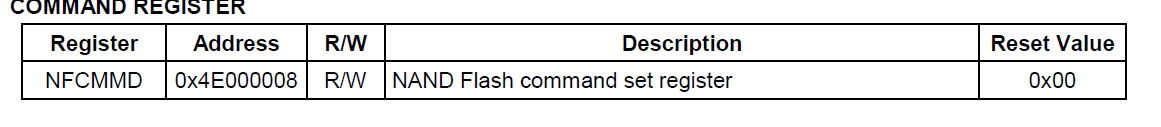


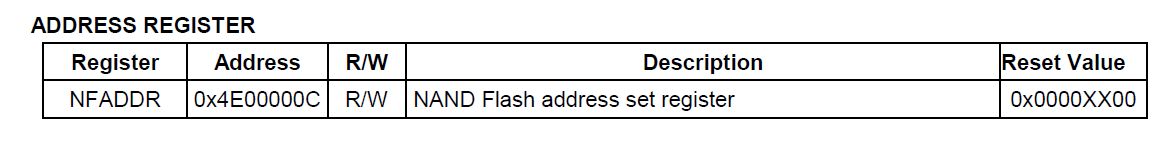
这个是

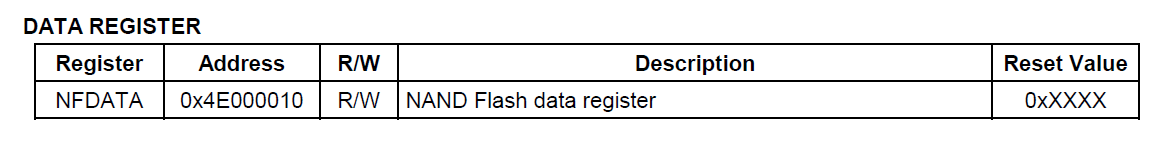


NFCONT控制器我们需要将它的1位置1

然后我们需要以下寄存器

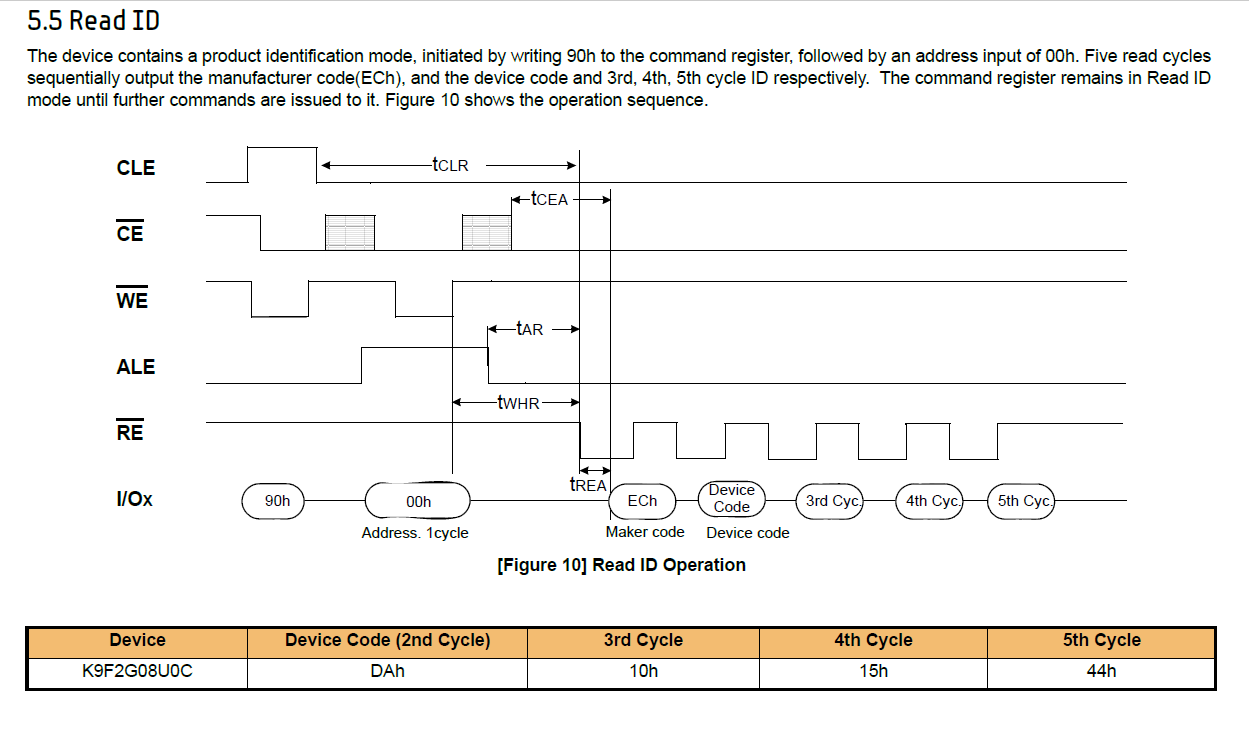






然后我们可以用uboot来体验以下nandflash的读写

首先我们可以来读ID



我们首先需要使能芯片，也就是在NFCONF这个寄存器的1位写成1

Uboot里面有个命令mw和md，分别是memory write和memory display，我们首先要看一下nandflash的1位是什么



L表示4个字节长度，后面的1是1个4字节的长度

我们看最后一个字节是3我们需要写成1



然后我们按照时序图上面说的写0x90



接着写地址0x00

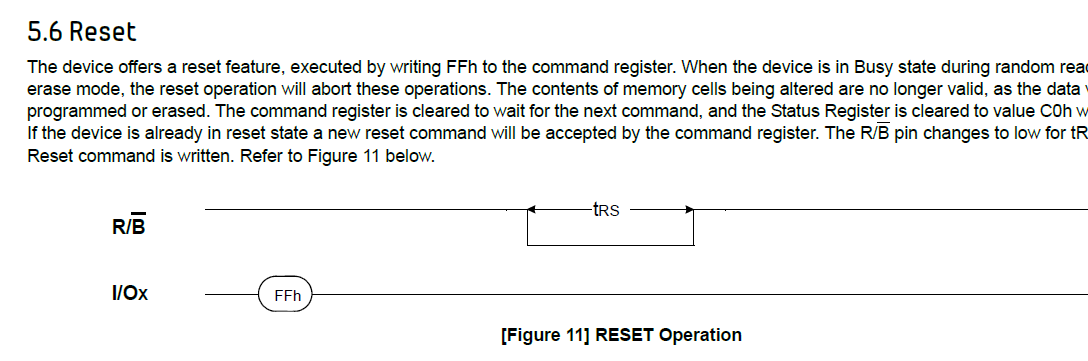
然后就是读了，第一个数据应该就是时序图上面写的应该是ec



我们继续读

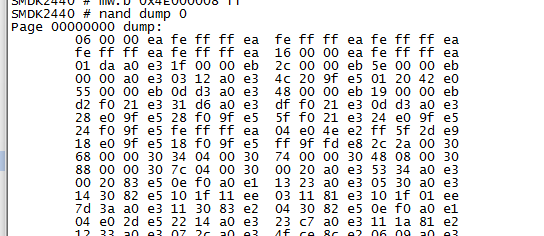


跟我们手册上面写的是一样的

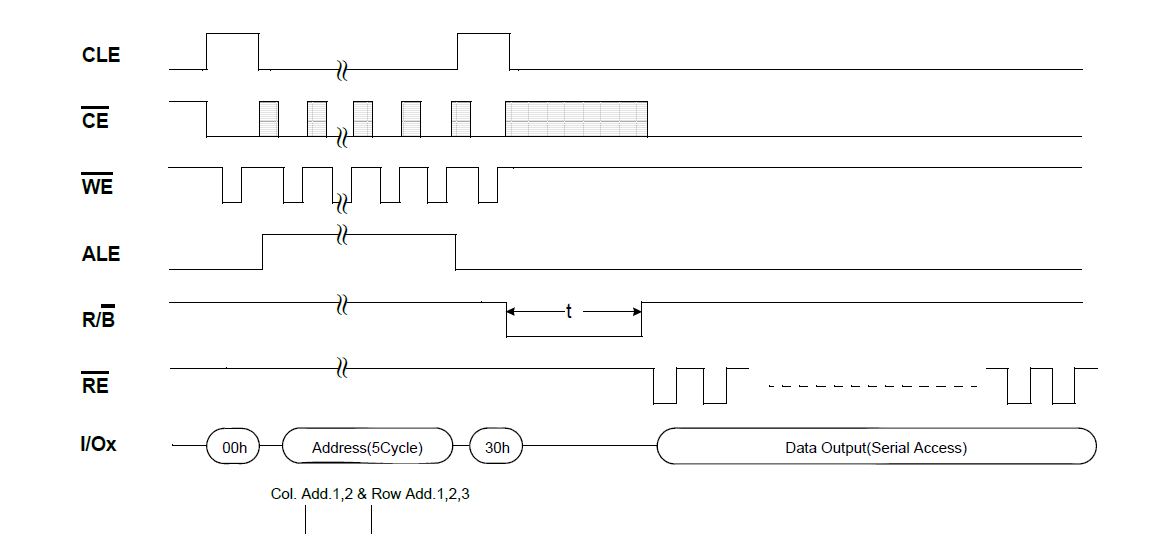
然后我们看怎么结束

手册上面写着是要发送ff就可以了

然后我们看怎么读取数据，首先我们看第0个块存储的是什么，我们用nand dump 0来查看



这里有很多我们看时序



我们看一下，这里地址需要发送5次，因为我们这个nandflash是256M的

1M是2的20次方，256是2的8次方，所以应该是要2的28次方的字节数，也就是说如果真要发送的话也是需要发送4回，但是因为需要兼容更多的设备所以变成了5次



因为我们是访问第0个地址，所以发送5次0就好了

我们就按照时序图的来

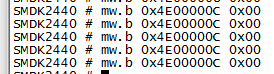
首先应该选中芯片



首先发送0x00命令



然后发送五次地址



然后再发命令0x30



然后我们就可以跟之前我们看到的数据是否相同了



确实是一样的，然后再取消选择

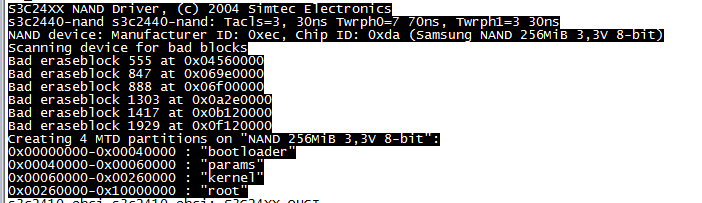


整个流程就是

1. 选择芯片
2. 发送命令
3. 发送地址
4. 发送数据
5. 读出数据
6. Reset

## Nandflash的框架

我们现在来启动内核来看看内核的打印信息

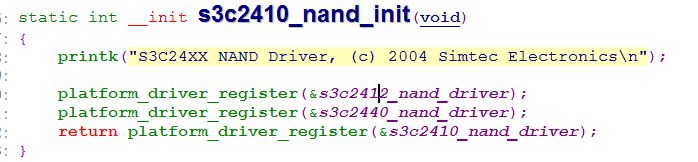


我们来搜索S3C24XX NAND Driver

搜索结果在

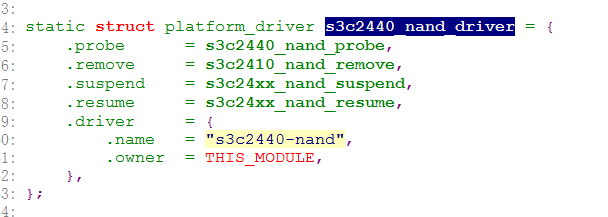


我们点开这个文件看看



这里是一堆平台注册函数

我们看看s3c2440\_nand\_driver、



只要有设备的名字符合s3c2440-nand的设备就会调用probe函数

我们来点开看看那个probe函数

点开之后应该是一个很长的一个函数，我们略过那些看到里面应该有



这个函数是硬件初始化

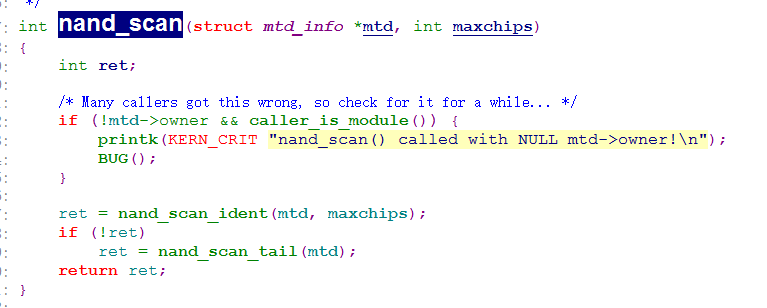


芯片初始化



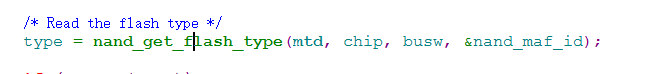
这里nand scan函数是用来构造mtd\_info的

我们点进去看看



这里面有个nand\_scan\_ident函数，我们点开来看看

这个函数里面有一个

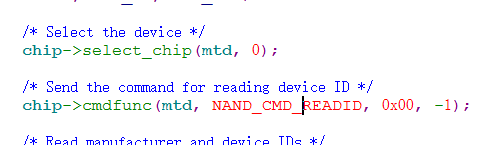


这个函数就是获得nandflash的型号的函数，我们点开看看

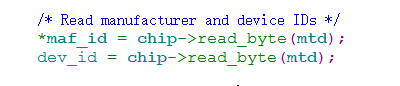
这个函数中有个参数是我们之后要写的一个结构体



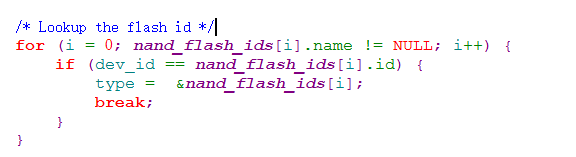
这个nand\_chip结构体就是要



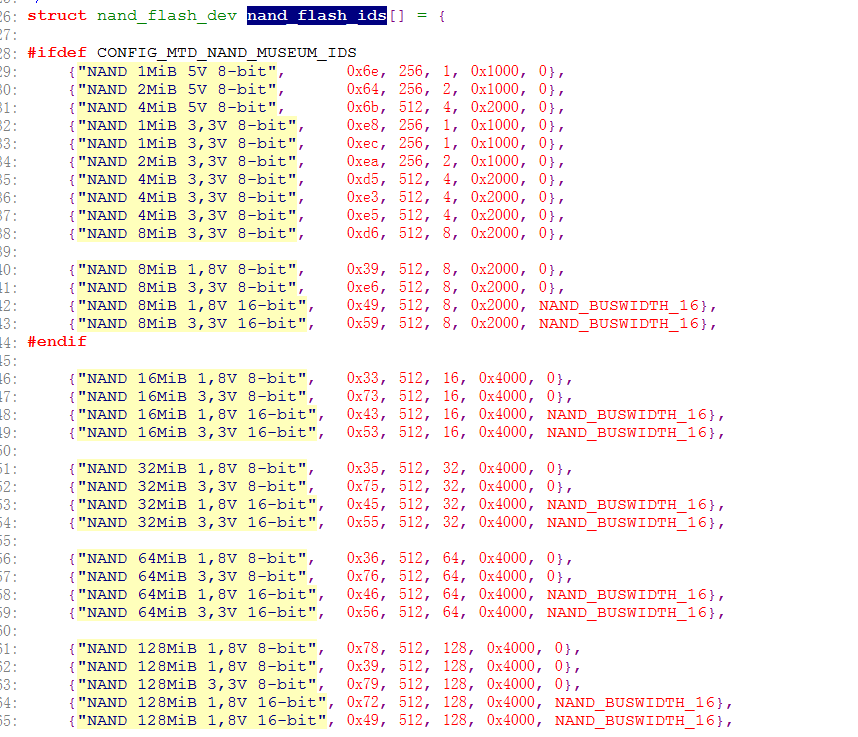
首先就是选中芯片，然后发出命令来读ID，我们可以看到，NAND\_CMD\_READID这个宏定义是0x90，也就是说跟我们之前用过的发送命令是一样的，然后发送0x00



然后分别有厂家ID和设备ID



下面的这个for循环是对数组里面存放的nandflash信息进行对比，我们可以看看这个数组



我们的设备是在

我们看内核的打印信息就是这条



Nandflash扫描完了我们再回过头来看一下刚才的nand\_scan函数的下面的一个函数

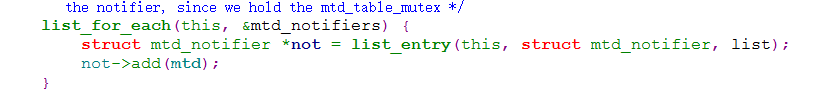


这个函数是创建分区我们点进去看看



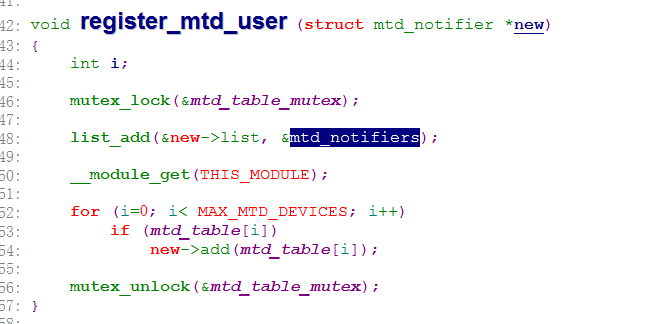
我们再进去这个函数来看看

这个函数里面有这么一句

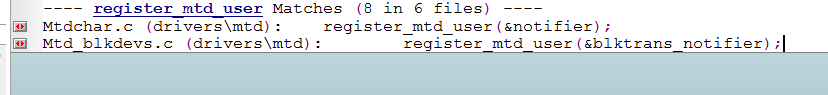


这里有一个链表，对这个链表的每一项都调用add函数

我们来看一下这个链表是放在哪里了

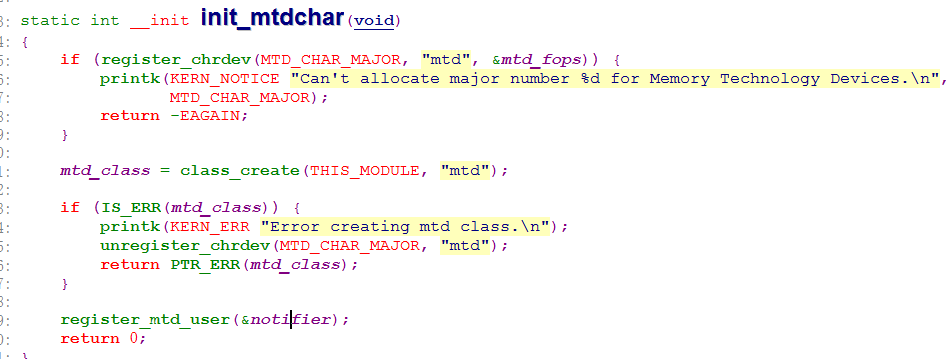


在这个函数中是写入这个链表的，我们来看看这个函数是谁在调用它

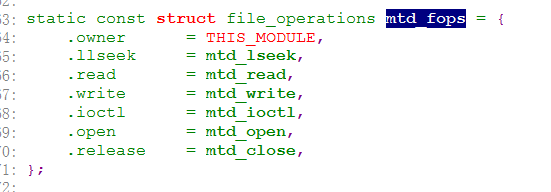


我们可以看到，其中有这两个函数在调用它，分别是字符设备和块设备，这也就是说明了nandflash不仅可以当做块设备使用，还可以当做字符设备来使用

我们先点进去字符设备的进去看看

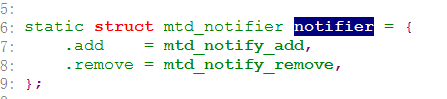


在这个函数的上面还有一个mtd\_fops这个就是之前我们写字符设备的时候弄的结构体

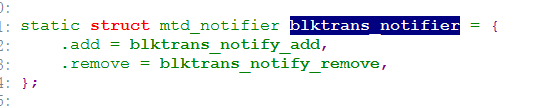


进行open read write

然后我们看register\_mtd\_user的这个参数是什么

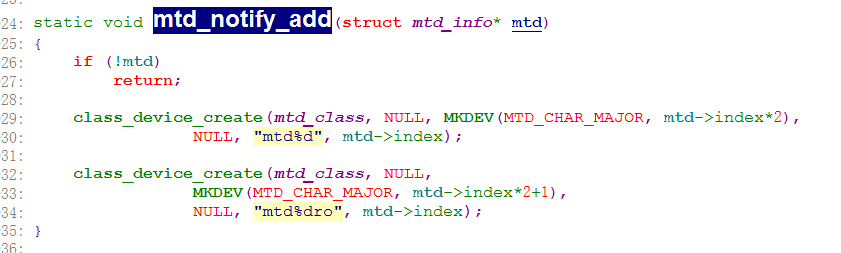


点进去看了一下，发现之前说list\_for\_each函数调用的add函数就是这个函数，在我们的block文件中也是有这个结构体的



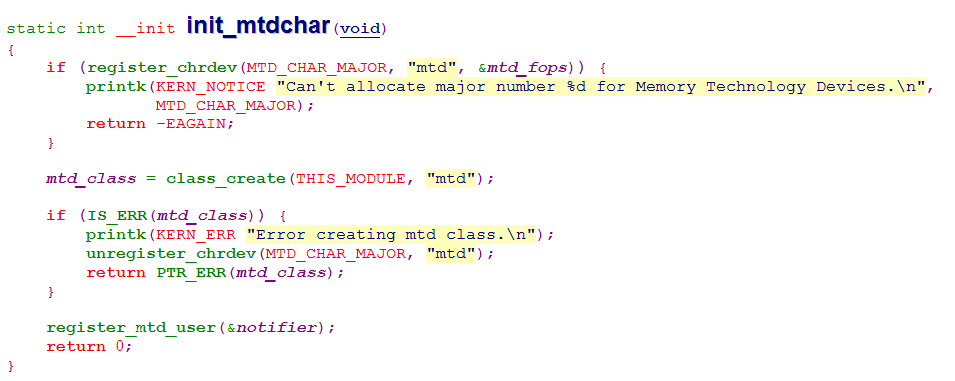
所以就是调用这两个add函数

我们先看字符设备的add函数



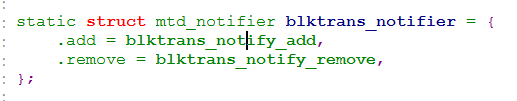
这个是两个create，一个是正常的另外一个是只读

我们再看一下这个文件的初始化

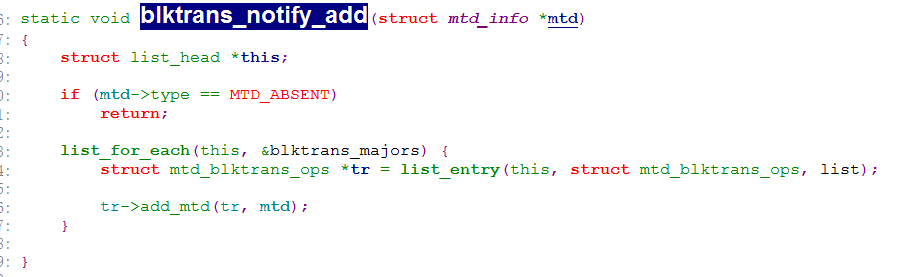


这个跟我们原来的是一样的，注册设备，创建一个类，但是没有创建设备节点，因为我们还没有识别出这个flash，所以等我们驱动程序识别出来了才会创建设备节点

我们再看一下块设备



我们点开add函数

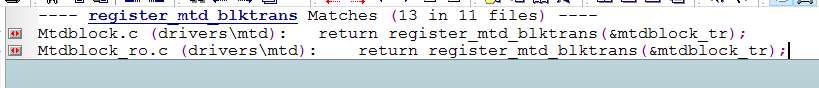


我们可以看到。这里面又出来了一个链表，list\_for\_each，那么我们看看这个链表是在哪里

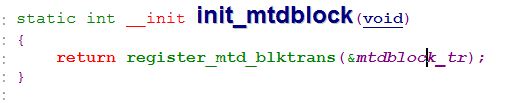
结果发现是在下面的这个函数里面



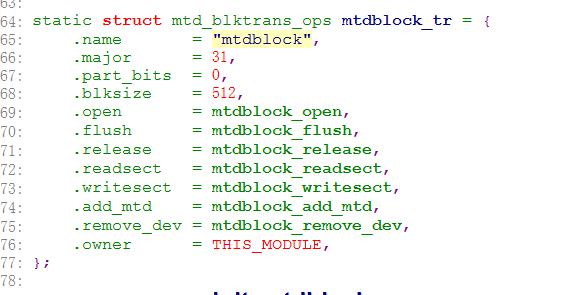
然后我们搜索一下，这个函数是在哪调用了



我们看到被一个正常的和一个只读的文件调用了，我们点进去那个正常的



这里面有个结构体我们点开看看



我们看到这里面有add函数，这个函数就是之前的add函数

我们点开add函数中的add\_mtd\_blktrans\_dev函数看看

我们再这个函数中可以看到有







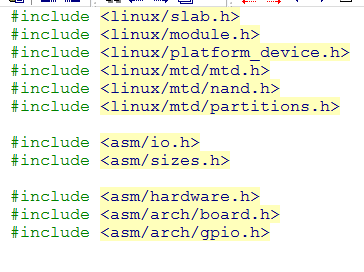


这就跟我们之前写内存驱动的过程是一样的了

## 编写nandflash驱动程序

我们现在来编写一个nandflash驱动程序，首先在内核里面有两个文件可以让我们来参考着写驱动程序

和这两个文件，我们先来参考一下它们的头文件



然后分别写出init和exit函数以及它们的入口

然后我们按照我们之前分析的来写这个驱动

1. 分配一个nand\_chip结构体
2. 设置这个结构体
3. 硬件相关的设置
4. 使用nand\_scan函数
5. 创建新的分区

然后我们就可以开始写了，首先是创建一个nand\_chip结构体

并分配它空间



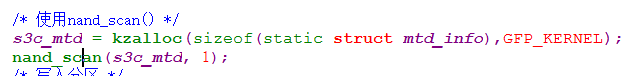


然后我们先看nandscan函数，我们调用它



但是我们看到，这个函数的参数还是需要两个形式参数，一个是mtd\_info结构体另外一个是最大芯片数，最大芯片数是1，然后我们再自己定义一个mtd\_info的结构体，并给它分配内存

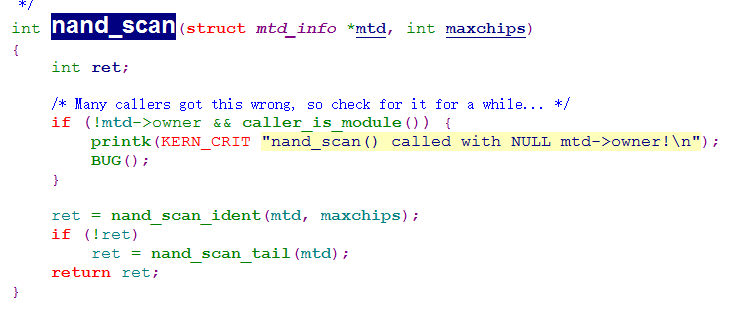




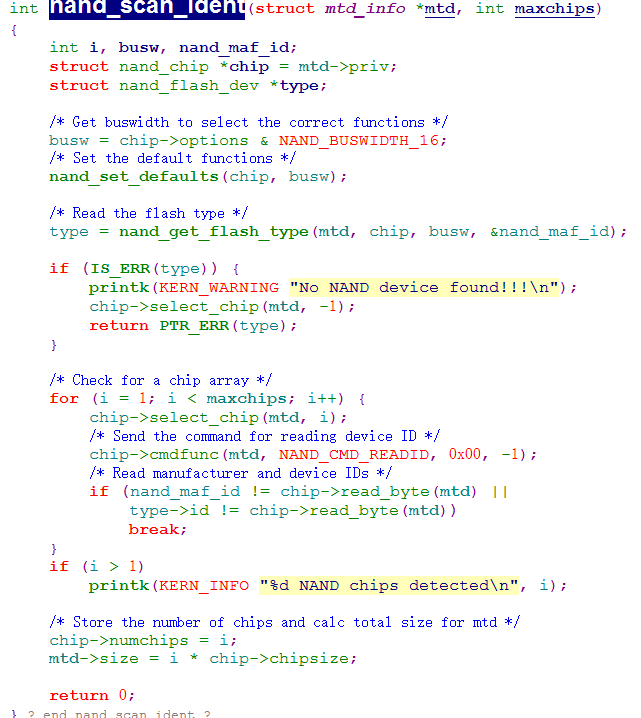
这样，然后我们看关键的nand\_chip结构体

但是我们nand\_chip结构体并不知道怎么去设置，但是这个结构体是给nand\_scan函数来用的，那么我们就可以看看nand\_scan函数都用到哪些nand\_chip中的成员了

我们先点进去看看

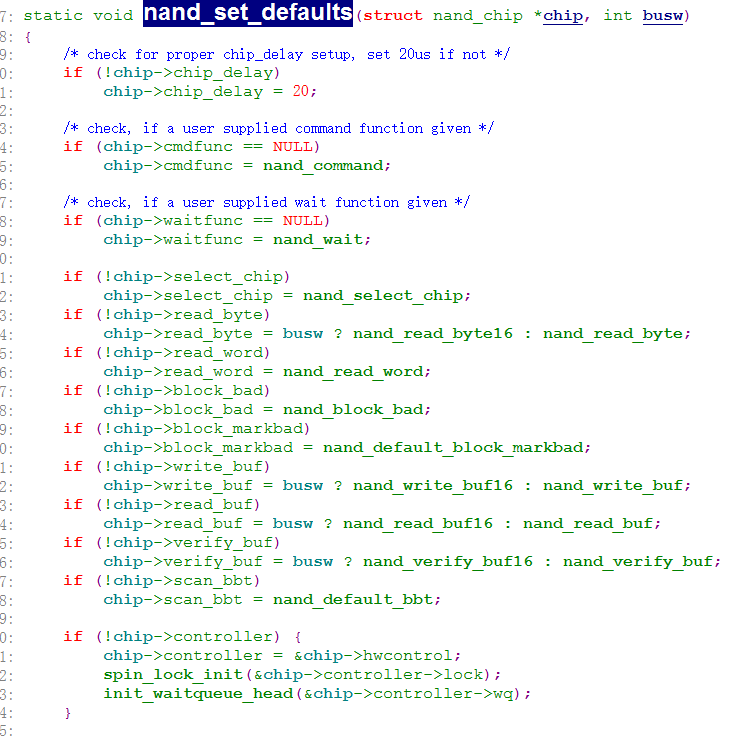


我们来看nand\_scan\_ident函数



我们看这里面有一个nand\_set\_defaults函数

我们点进去看看，这里是设置默认值的函数



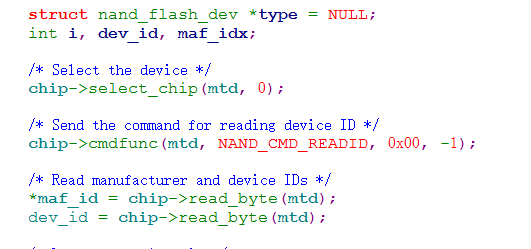
这里我们看到，如果nand\_chip结构体哪些没有设置的话，那么就设置默认的函数，但是我们需要使用哪些函数呢，因为我们构造nand\_chip结构体是给nandscan函数来用的，那么我们就需要看这个函数

我们继续往下面看

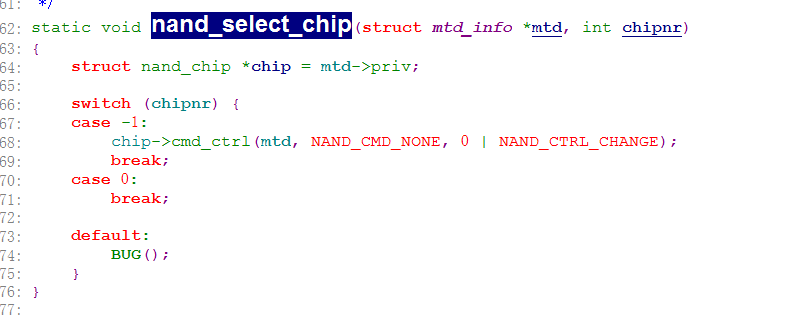


在设置默认函数的下面有一个这个函数我们点进去看看

在这个函数中我们之前也分析过了，这里面都用到了一些操作函数

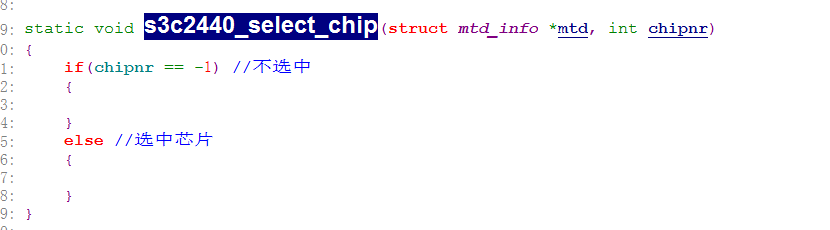


我们说nand\_chip是在最下面一层了，也就是说它要具备读地址、读数据、写数据、写命令、读状态这几个功能，那么我们看首先由选中芯片，我们看用的是select\_chip函数，我们回过头来看这个默认函数我们可不可以用

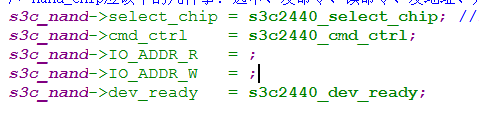


这个是我们的默认选中芯片的函数，我们貌似不可以用，因为-1是不干，到0就没有操作了，所以这个我们得自己写函数





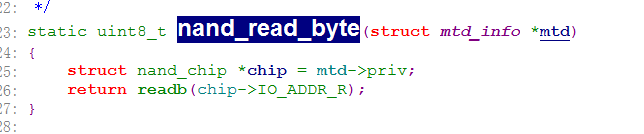
然后后面也是一样的

后面的都需要自己看

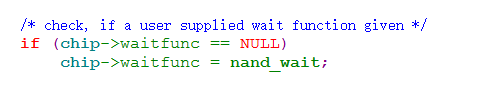
我们看一下后面三个是怎么找的，前面两个已经写完了，一个是选中芯片一个是发送命令，然后就是读地址、写地址、读状态



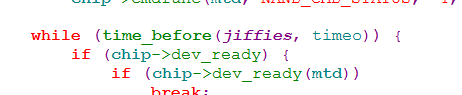
这个是我们默认函数给我们的读函数，因为我们的nandflash是8位的，所以我们就看后面的nand\_read\_byte就好了



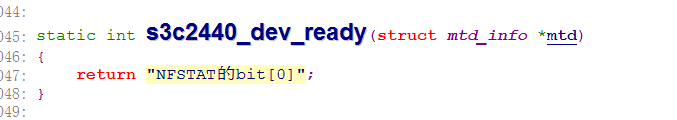
我们看到，这个函数里面就是一个return，也就是说我们本质上其实不是写nand\_read\_byte函数，而是设置它的return函数的返回的东西，那么读也是一样的，最后我们看读取状态怎么读



默认函数有一个等待函数，那么我们先看看这个函数



我们发现最后调用的是dev\_ready成员，所以我们设置这个成员就好了

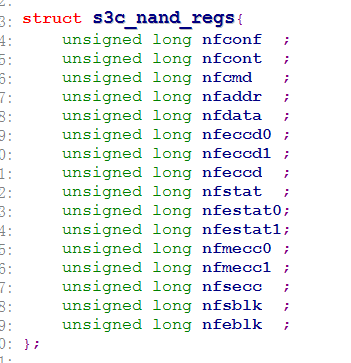


读状态我们需要设置的就是一个事返回NFSTAT的bit0

我们继续把程序完善了

首先我们应该把nand控制器全部的映射出来

我们可以写一个结构体，但是我们需要注意的是每个寄存器的长度都应该是等长的这样地址排列下来才可以是对的，否则就会出现错误的情况



然后我们再init函数中对它进行ioremap，让它们的地址可以映射到内存中去

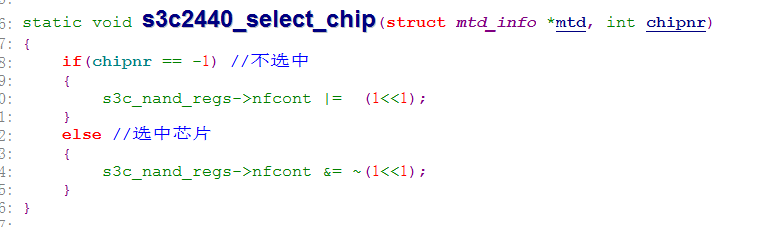


然后我们补全之前的读和写的虚拟地址

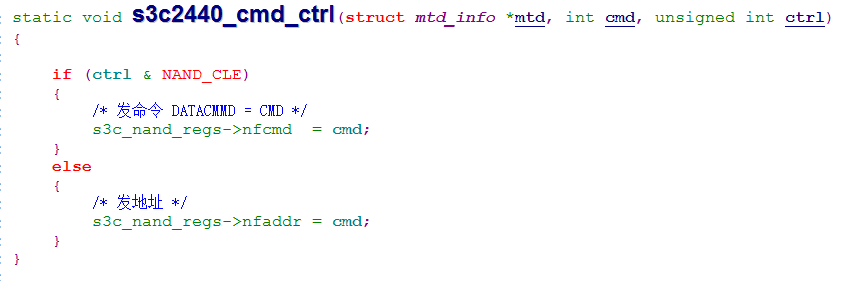


因为读写都是在data寄存器的，所以这两个都要写data寄存器

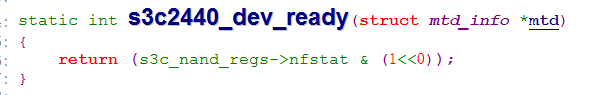
然后我们把之前的函数写出来，首先是选中芯片



然后是发送地址或命令

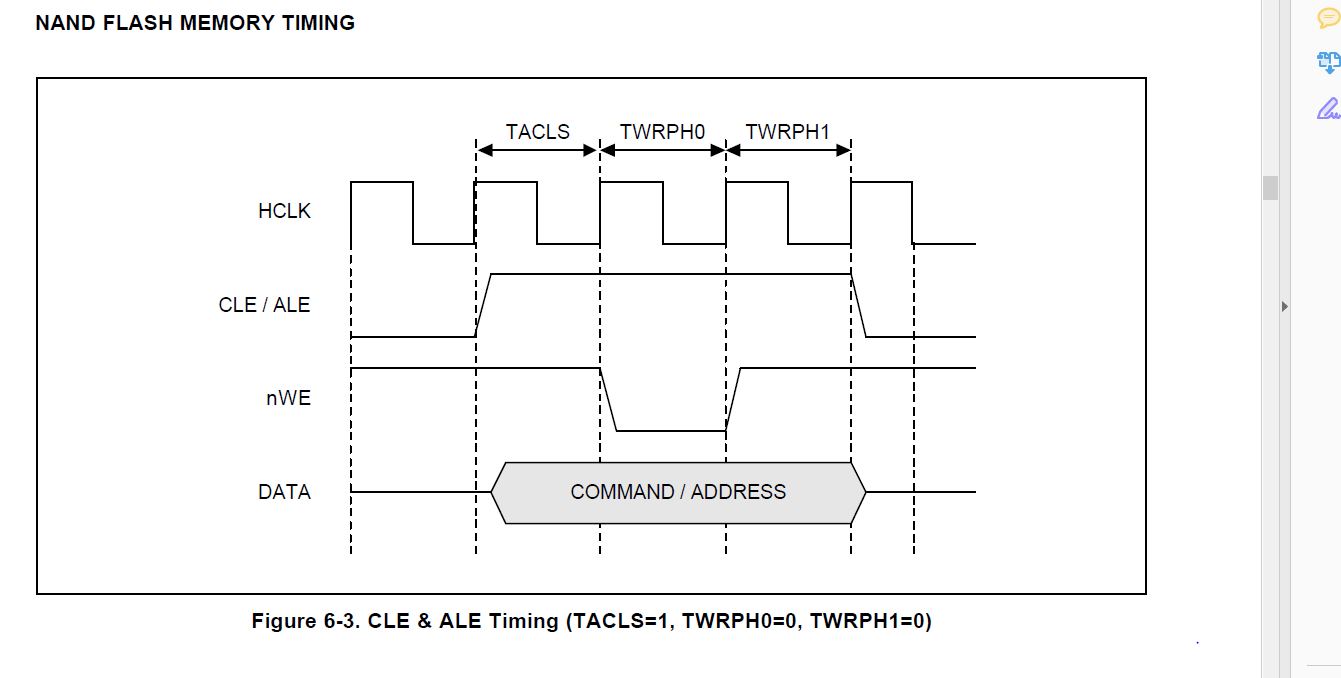


然后是检查状态



然后就是硬件相关的代码了

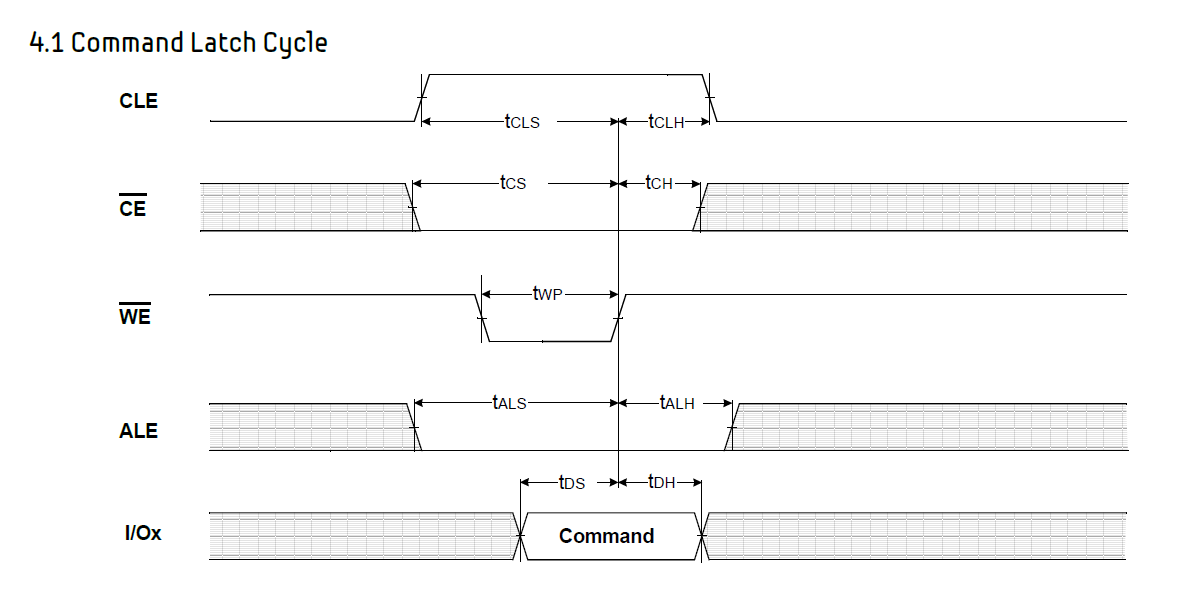
我们看一下2440的数据手册



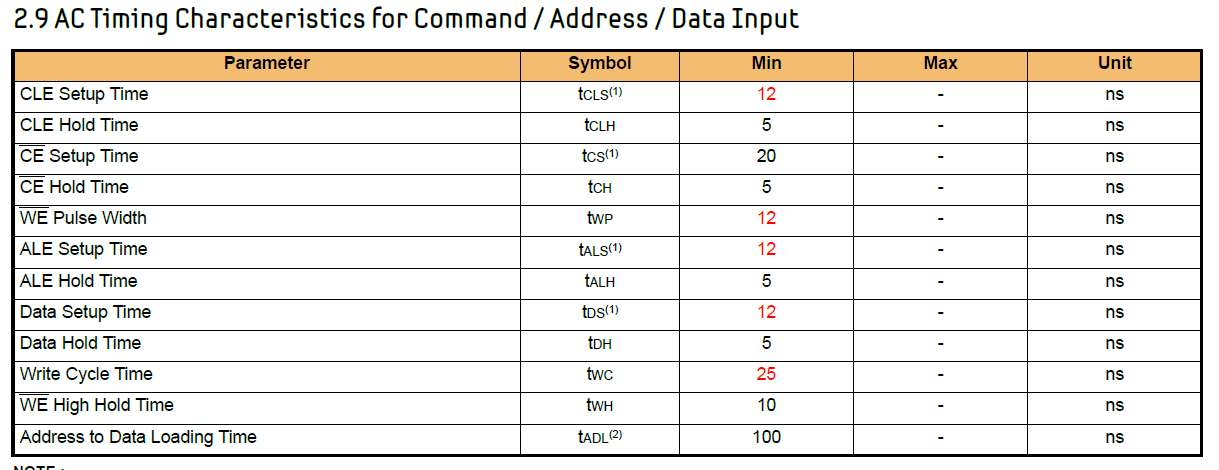
在数据手册中nand这一节中有这个时序图，也就是说想要控制nandflash需要把这几个值设置好

我们再看看我们nandflash的数据手册

我们随便在nandflash的数据手册上照一张时序图看看



在数据手册中还有一个表就是说明这些参数的表

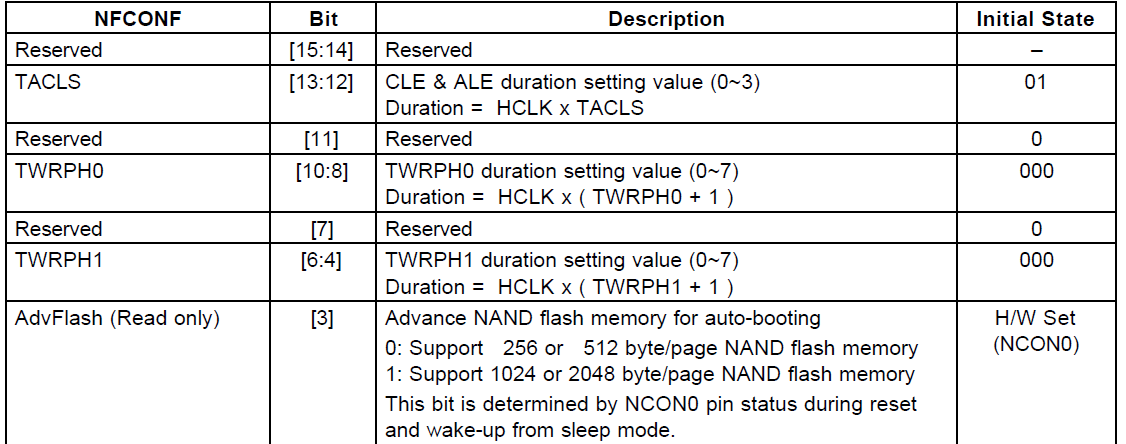


我们看到TACLS是CLE和WE之间的那么一段，也就是说tcls减去twp，所以是0，所以TACLS就是0

TWRPH0中间那段WE的时间也就是12ns

TWRPH1是nandflash时序图中的tch，也是说最少是5ns

我们再看看这三个数分别是在哪设置的



结果我们看到是在nfconf寄存器中设置的，我们看到每个值都会有一个公式，可以看到其实这些纳秒值都是算出来的

我们看到每个公式都会有个HCLK，这个值我们可以在内核的启动信息中可以知道



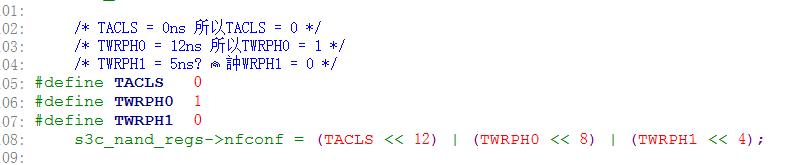
我们可以看到HCLK是100MHz

那么我们就可以算了，100MHz其实就是10ns

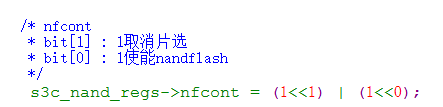
首先TACLS就为0

TWRPH0是要求的12ns，所以是我们只要让和HCLK乘的数大于1就可以了

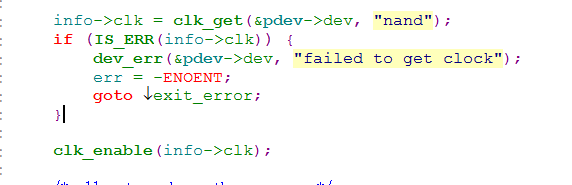
TWRPH1是只要大于5ns就可以了，也就是说HCLK就可以，所以让系数等于1就好，那么就让TWRPH1等于0就可以了



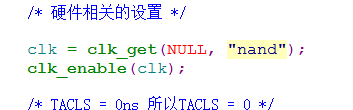
然后我们还需要使能nandflash以及先关闭片选



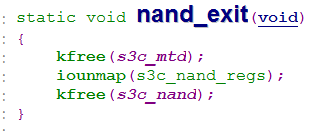
然后我们再看看例子里面还设置了什么



还有这个我们需要使能一下时钟



最后是exit函数



然后我们制作成ko文件，最后装载到内核中去但是这时候内核就会提示我们没有使用ECC校验

我们在设置nand\_chip结构体的时候加上

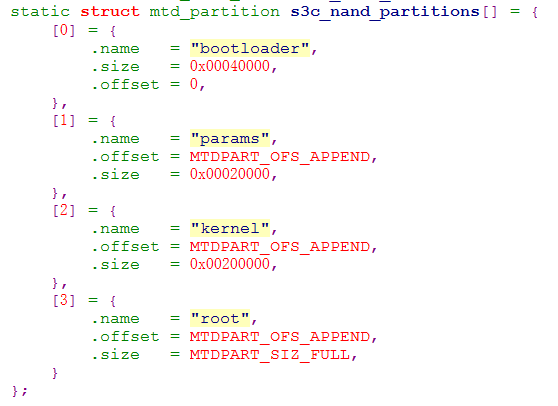


就会没有提示了

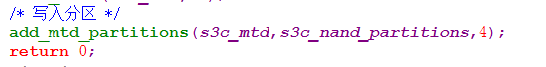
我们继续完善这个驱动程序

我们现在只是差最后一步分区了

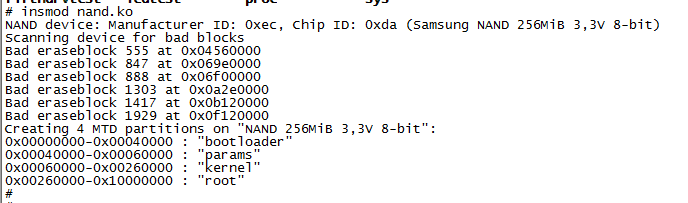
我们首先要把区分好



然后我们用add\_mtd\_parition函数就可以了



然后制作成ko文件我们装载之后就会看到分区



然后我们还有格式化工具我们解压那个压缩包



然后我们进入到util文件夹，然后修改makefile

将arm-linux前面的#去掉，之后make，make出来的flash\_erase 和 flash\_eraseall复制到文件系统的bin目录内

之后去掉内核的nandflash选项

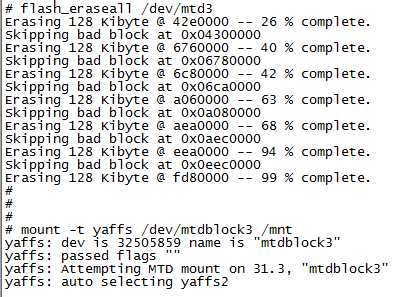
-> Device Drivers

-> Memory Technology Device (MTD) support

-> NAND Device Support

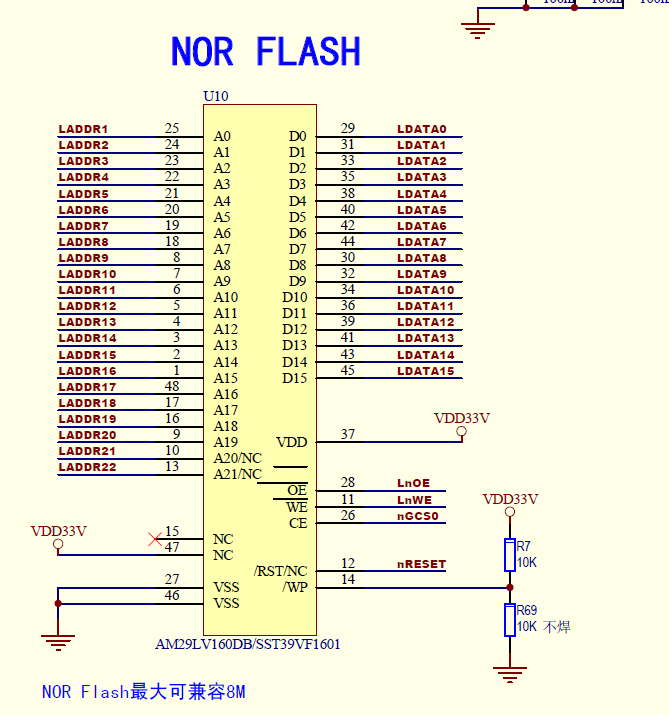
< > NAND Flash support for S3C2410/S3C2440 SoC

最后使用那个工具来格式化

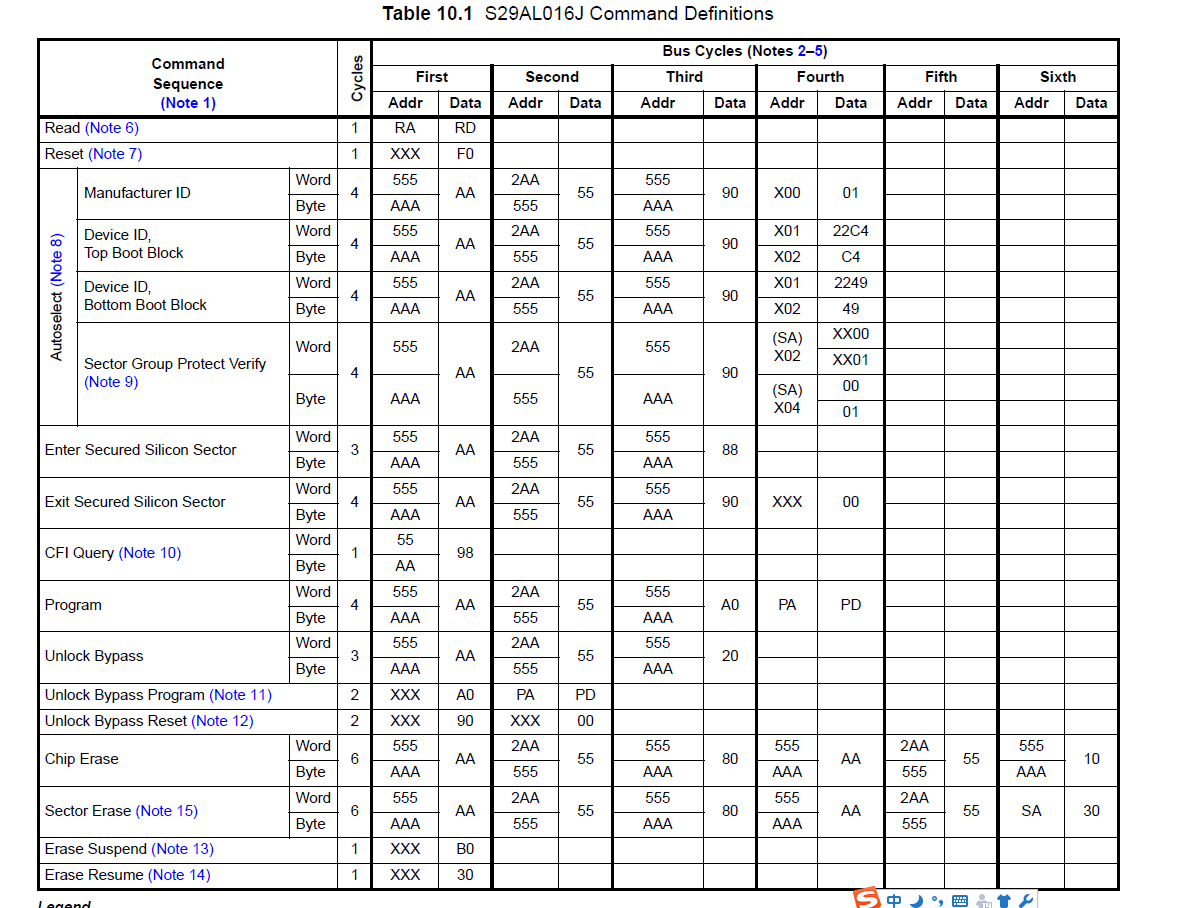


# NOR FLASH

Norflash也是一种存储工具我们先来了解一下norflash



这个就是norflash的原理图，我们可以看到是22条地址线16条数据线，这个相比nandflash要多出很多线，这个norflash跟内存的操作非常像，我们来看看norflash的使用手册



Norflash手册有一个命令表，我们可以按照这个命令先使用u-boot发命令体验一下，我们可以看到原理图上面说的，先用u-boot发命令

但是我们看到我们这款norflash是16位的，所以我们需要注意位宽的问题，16位的芯片是从地址1开始的，所以我们发命令的时候需要把地址左移一位

我们现在可以读取norflash的ID，我们按照手册上面说的

向555发送数据aa

向2aa发送数据55

向555发送数据90

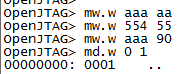
然后读取00地址

我们可以试一下因为我们需要地址都左移一位那么我们就需要把地址都重新算一遍

所以这几次的地址就是

aaa 554 aaa

所以我们就这么发送就好

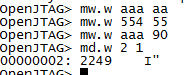


我们看到返回的数据是0001，这个跟我们数据手册上面的是一样的



返回的确实是01

然后我们可以读取一下设备ID



我们看到读出来的是2249这个也是没有出现错误



当我们这时候想读出来norflash里面的数据的时候是不行的，如果想读取正确的数据那就必须要退出这个读取ID的状态才可以



我们让它复位就好了

然而我们想往norflash写数据的时候现在是写不进去的



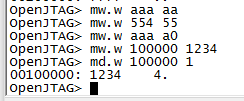
我们需要按照数据手册上面的来弄



aaa aa

554 55

aaa a0



我们这个时候再写就可以写了

记住一定要退出这个状态

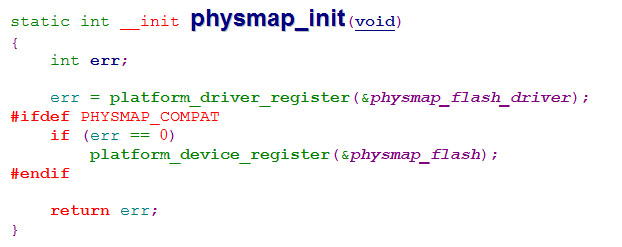
## Norflash的框架

我们进入内核看一下norflash的框架，norflash也是属于块设备，跟之前说的的块设备的框架是差不多的，我们先看一下内核中norflash的驱动程序

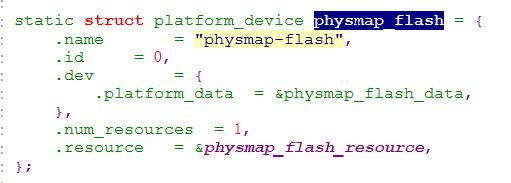


这个文件就是内核中norflash的驱动程序

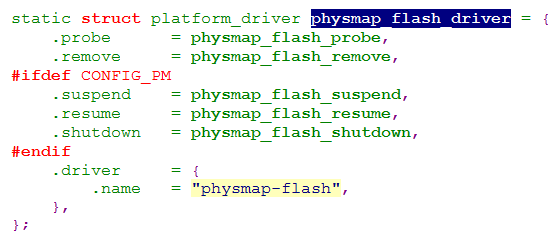
我们首先看init函数



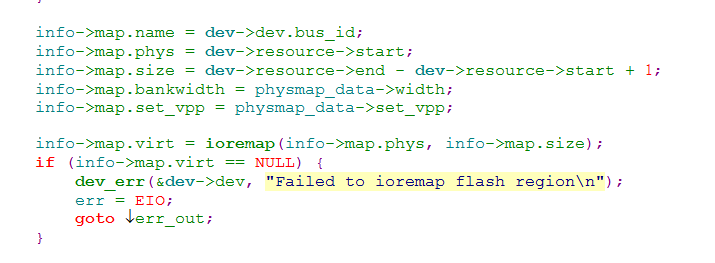
首先我们就看到有两个结构体我们看看



底下这个结构体主要是在写norflash的数据的，我们看一下上面的那个结构体



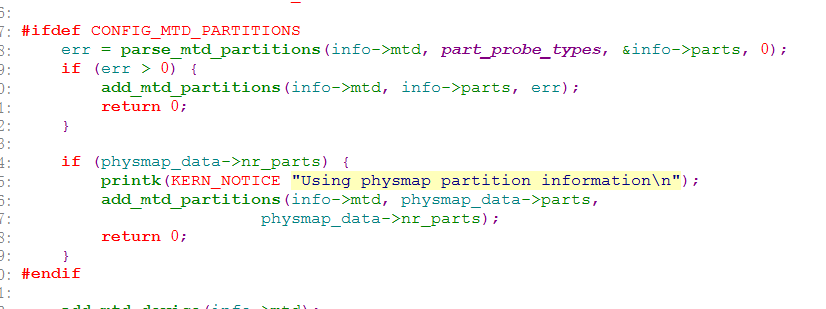
首先就是probe函数，我们看一下probe函数



我们看到probe函数先给分配了physmap\_flash\_info结构体然后.

设置这个结构体

然后是硬件相关的代码，最后是创建分区



我们配置一下内核来装载这个norflash驱动

-> Device Drivers

-> Memory Technology Device (MTD) support

-> Mapping drivers for chip access

<M> CFI Flash device in physical memory map

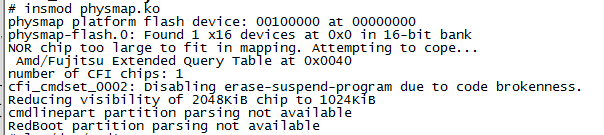
(0x0) Physical start address of flash mapping // 物理基地址

(0x1000000) Physical length of flash mapping // 长度

(2) Bank width in octets (NEW) // 位宽

我们需要照这个改动一下，然后make modules

之后我们把生成的对应ko文件放到文件系统中去，最后再启动开发板装载这个驱动

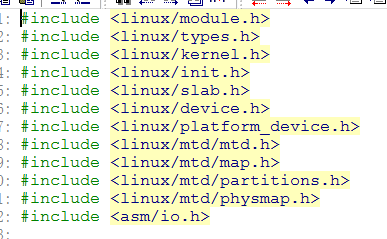


我们装载的norflash的信息就都显示了

## 编写一个norflash驱动程序

我们了解完框架之后就可以写一个真正的驱动程序了

首先我们可以参考内核中的驱动程序的头文件



然后再写它们的init和exit函数以及它们的修饰符

然后我们按照

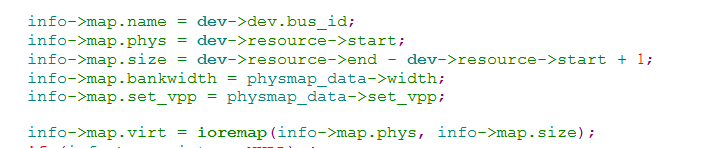
分配一个map\_info结构体

设置map\_info结构体

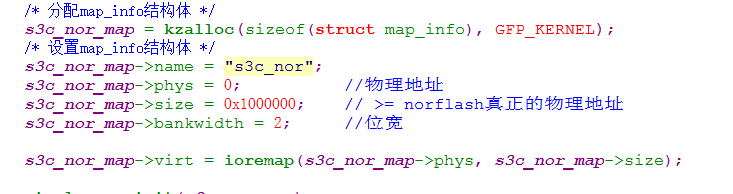
识别norflash

新建分区这几个步骤来写

我们可以参考一下内核的函数是怎么写的



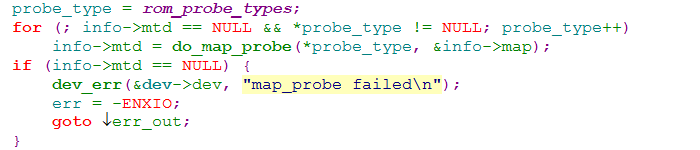
他们是这么设置这个结构体的，那么我们也仿照这个做



我们看到下面有一个简单初始化我们也跟着加上



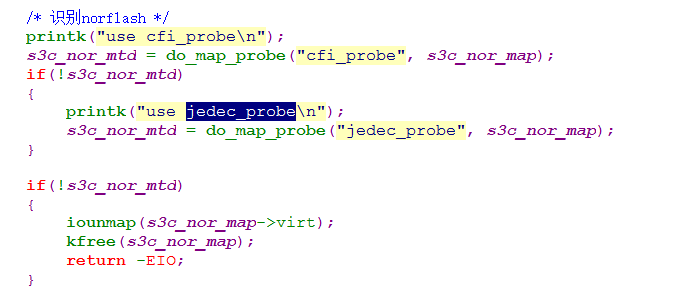
然后我们接着看内核中的驱动都干了什么



我们看到它下面调用了一个do\_map\_probe函数，这个函数就是识别出norflash的型号的

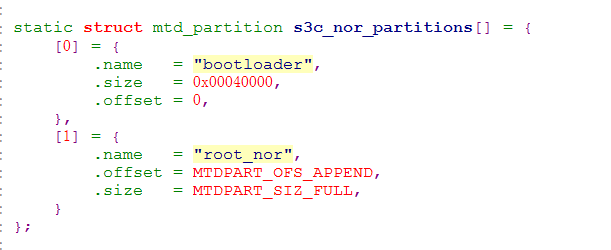
我们也按照它这么来

Norflash用两种协议一种是cfi\_probe一种是jedec\_probe，我们就先使用cfi，如果不行在用jedec最后不行就报错

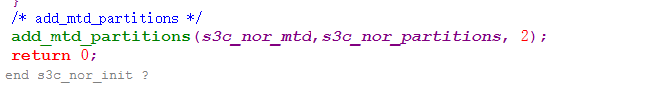


最后一步就是新建分区了

我们按照nandflash来新建分区就可以了

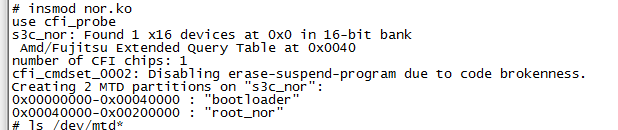


然后调用add\_mtd\_partitions



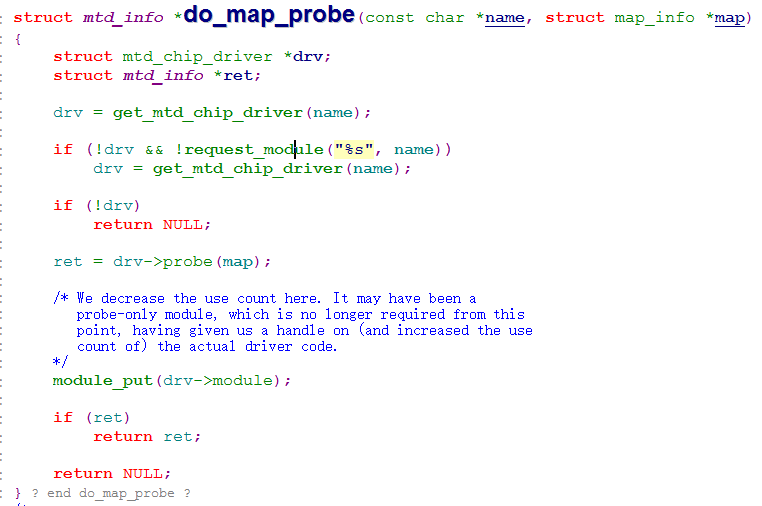


然后我们制作成ko文件装载到内核当中



我们设置的信息就都出来了

我们来深入地分析一下内核是怎么识别norflash的，识别norflash的函数是do\_map\_probe我们来点进去分析一下，norflash有两种协议来识别norflash，一个是cfi另外一个是jedec协议我们先进去do\_map\_probe函数看看

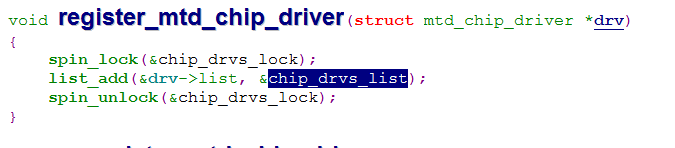


我们看到有函数，这个函数是按照两种协议来弄的

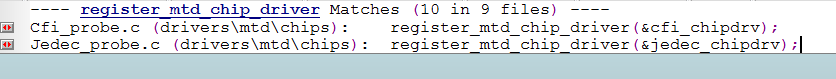
我们点进去看看



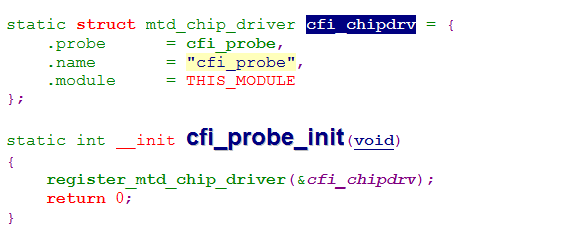
我们看到插入链表那么我们搜索一下这链表在哪调用了



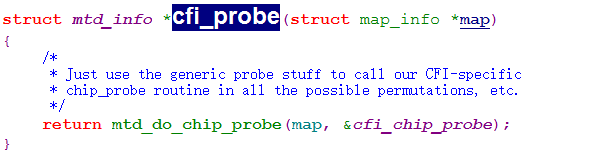
我们再查找一下这个函数是在哪调用的



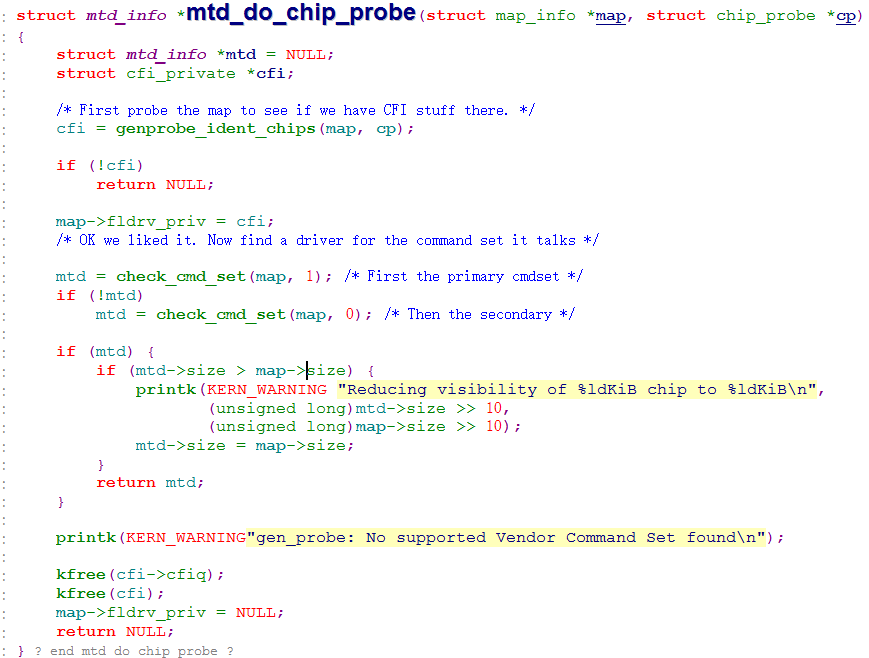
我们看到有这两个文件调用了这个函数一个是cfi一个是jedec，我们先看cfi



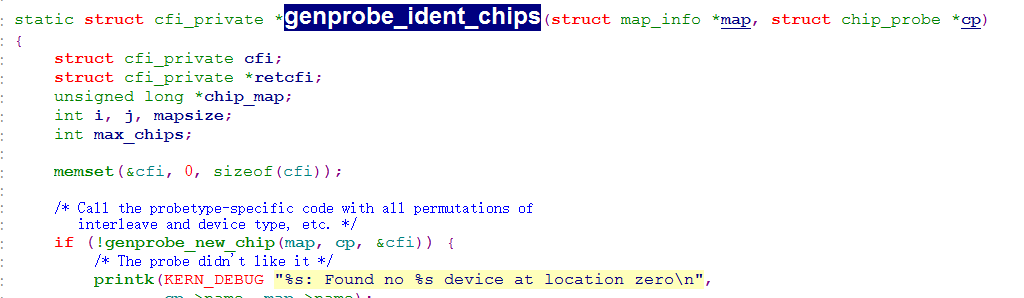
这里有一个cfi\_probe函数



我们看它调用的函数



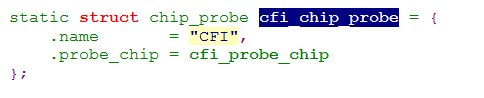
再往里看

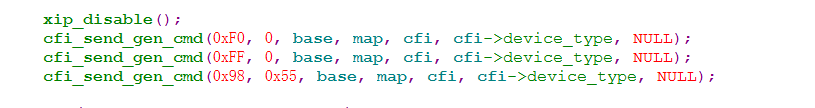




我们最终看到是调用的cp->probe我们就可以看这个函数了

其实cp实际上就是cfi\_chip\_probe我们看一下cfi\_chip\_probe这个结构体



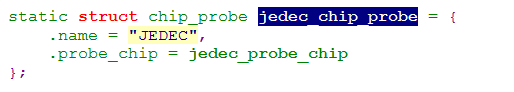


我们可以看到这个函数里面有发送命令，我们看到发送98是让norflash进入cfi模式

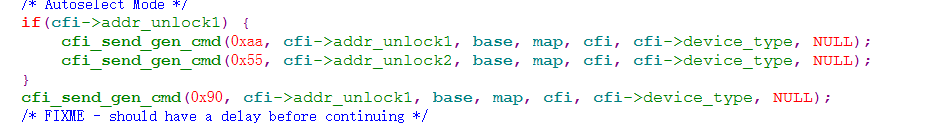
之后就是读qry

然后我们看jedec模式

Jedec模式前面跟cfi是一样的，我们直接看jedec\_probe.c就好了



直接看这个函数就好



这个函数里面有解锁命令读ID命令



得到厂家ID设备ID，之后和数组进行比较

