

ARM cortex-A8 S5PV210 学习

作者：向仔州

目录

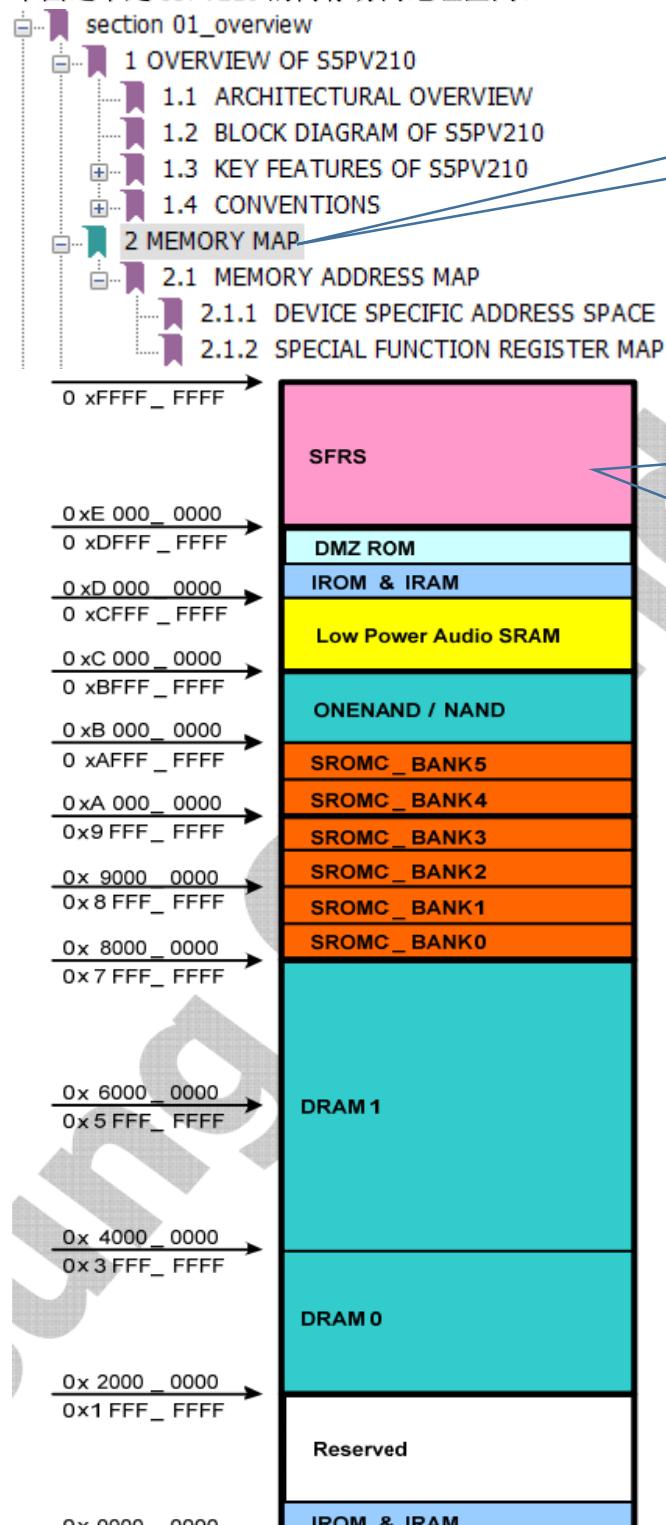
开发板基础调试连接.....	2
S5PV210 内存映射.....	3
SD 卡制作成启动介质的方法.....	12
DNW 软件下载 uboot.....	13
如果 Windows7 使用 dnw 下载 uboot 不行，换成 linux 使用 dnw.....	15
Fastboot 如何将 uboot, zimage, rootfs 烧录进 EMMC/iNAND/闪存.....	16
Linux 下安装软件的三种方法.....	20
Linux 文件系统目录管理.....	20
安装交叉编译工具链.....	20
汇编程序编写 LED 驱动.....	23
裸机下载汇编程序执行过程.....	25
汇编语言关闭看门狗.....	29
汇编启动代码启动 C 程序.....	30
裸机下 .s 汇编文件和.c C 文件混合编译成二进制文件过程.....	33
代码重定位.....	34
SDRAM 介绍及初始化,必须要有前面重定位知识.....	37
S5PV210 时钟系统.....	42
裸机汇编程序下载到什么地方？程序从内存哪行地址开始启动？.....	45
SD 卡启动程序(把程序烧录进 SD 卡启动).....	46
Nandflash 接口.....	54
iNAND 存储芯片使用，替代 NANDflash 的趋势.....	62
LCD 使用.....	68

开发板基础调试连接

根据 USB 转串口芯片安装，PL2303,CH340 任意一个驱动，开发板串口调试接口在 UART2

S5PV210 内存映射

下面这个是 S5PV210 的内存访问地址区间。



内存映射布局在
这一页

SFRS 这块区域里
面放的是
IIC,SPI,GPIO 等
等.....这种片上
外设的地址，你
可以向单片机访
问外设寄存器那
样去访问它

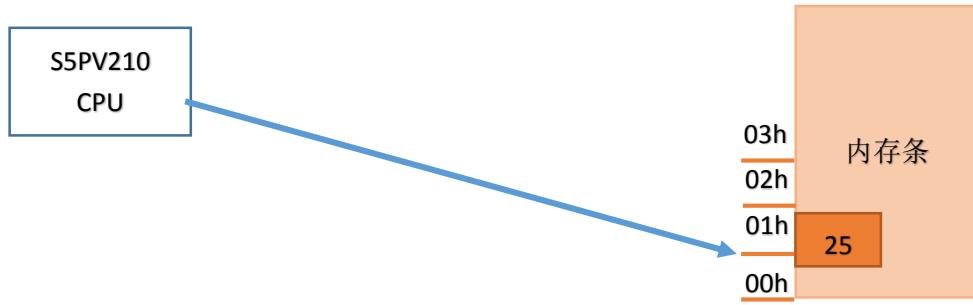
CPU 直接访问内存，说白了就
像 C 语言
`int *p = (int *)0x00000001, 然
后*p = 25`



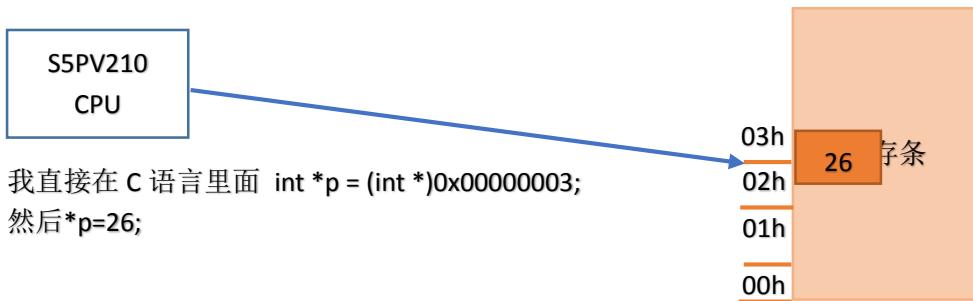
内存和外存区别

内存是可以直接和 CPU 进行访问的

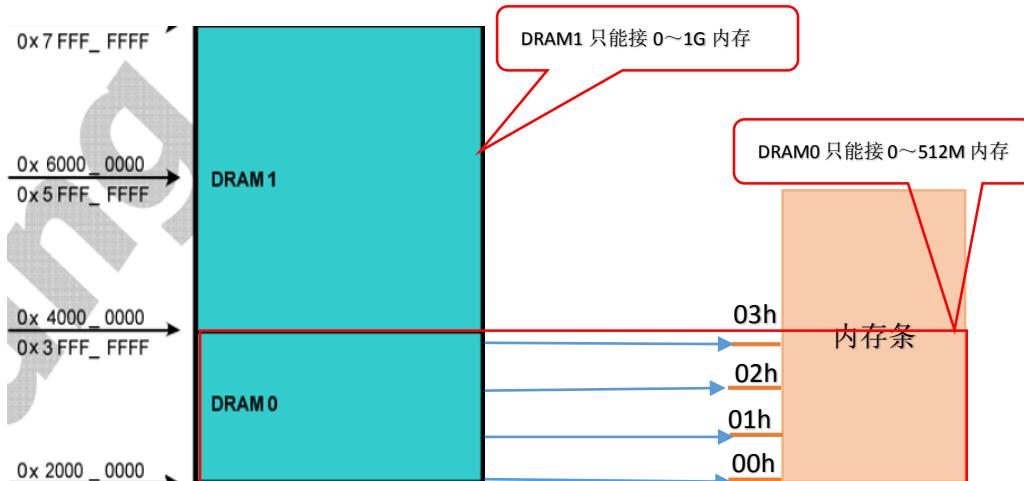
S5PV210
CPU



如果我想给内存地址 03h 写个值 26 的数据



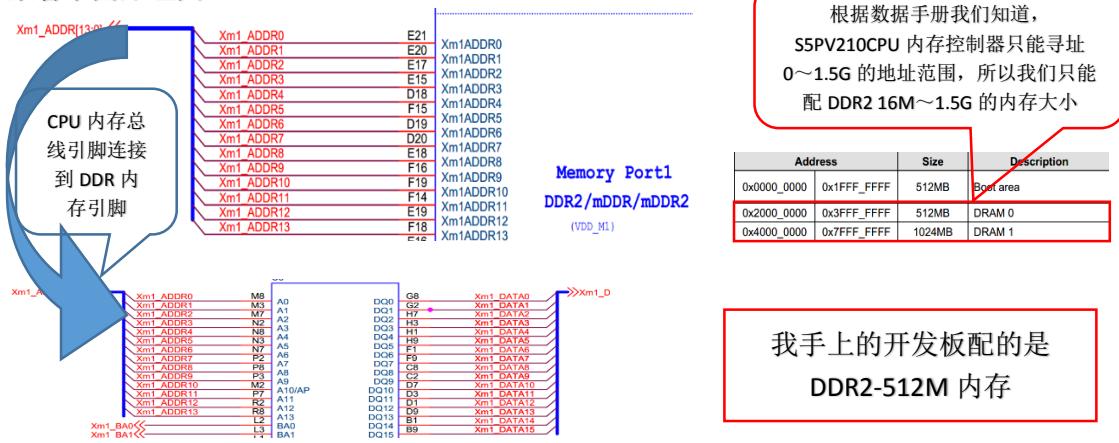
我直接在 C 语言里面 `int *p = (int *)0x00000003;`
然后`*p=26;`



我们 S5PV210 使用自己的内存控制器去控制内存的，所以我 C 语言写地址的方法是直接去写内存控制器的地址

比如我要写内存的 00h 这个地址，就是 `int *p = (int *)0x2000_0000;` 然后`*p=25;`

写 `0x2000_0000` 地址，然后 CPU 将 `0x2000_0000` 地址上的数据传给内存条 `00h` 地址。这就是内存映射，我们只需要指定 CPU 内存控制器的地址。至于内存条和 CPU 内存控制器的关系看下面原理图



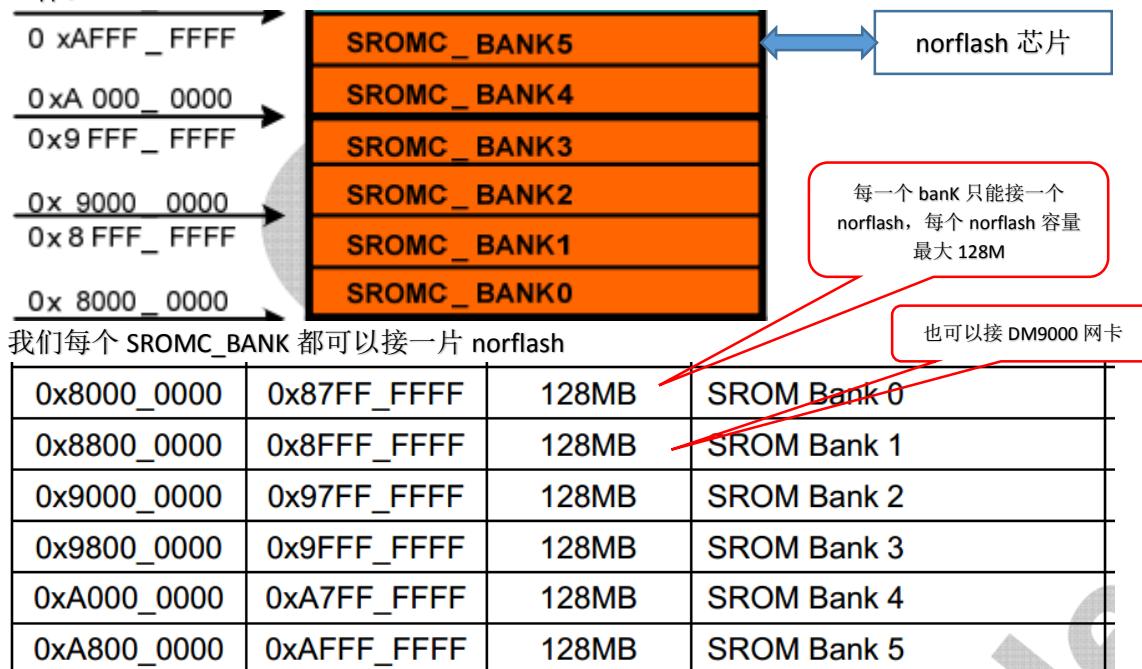
外存就不一样了

外存不像上面内存那样 CPU 内存控制器给多大的地址范围，你就只能访问这么大的地址范围，所以内存大小是受限制的。外存可以接无数大的存储器 0~1T 的存储器都可以接。

但是外存的访问速度慢，代码不能像上面 C 指针那样直接指定地址。比如 `flash` 就是外存，我要写时序一大堆代码才能访问外存。所以外存访问速度慢。但是就是存储空间做得大。

外存有以下系列：

nor flash：这个 norflash 类型存储器可以直接指定存储器的地址去访问，和上面 DDR 内存一样。



那么 norflash 最大存储量只能做 128M，我拿来干嘛？其实 norflash 16M 就很大了，norflash 很贵所以一般不用 norflash 做存储。norflash 的好处就是像 DDR 那样可以直接地址访问。虽然 norflash 很多是 SPI 总线的，但是 SPI 总线指定 norflash 地址，是直接选择任何一个地址，然后写数据。和 DDR 一个意思。

Nand flash:

以前的 CPU 很多都是直接从 norflash 启动的，但是 Nandflash 是不能直接启动代码的。所以三星公司在内存里面加了一块 IROM 区域



启动代码放在 IROM 里面，然后用 IROM 里面的代码去启动 Nand flash

Nand flash 分为两种，一种是 SLC，另一种是 MLC。

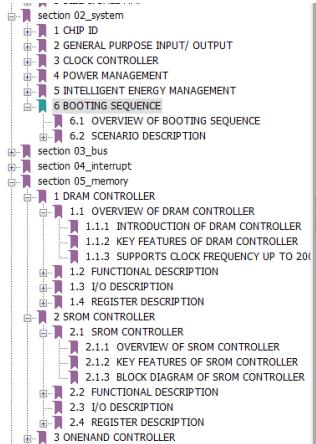
SLC 类型的 Nand flash 稳定性好，不容易坏块，容量小，你不做 ECC 校验都没有问题。如果你的存储空间只需要 128M, 256M, 512M，那么建议你用 SLC。

MLC 类型的 Nand flash 稳定性差容易坏块，容量大，你必须做 ECC 校验。如果你要做几个 G，几十个 G，成本要求很低，就用 MLC。

eMMC/iNand 也是 nandflash 的一种，这种东西里面自带一个小 CPU+Nandflash，芯片里面的 CPU 可以帮你对 Nandflash 进行坏块检查，如果能用这种，那么前面的 SLC 和 MLC 都不用考虑了，因为 eMMC 可以容量做到很大，价格便宜，不会坏块(其实会坏块，只是 emmc 内部 CPU 帮你处理了坏块问题)。

SD 卡也属于 nandflash 的一种。SD 卡里面有个 CPU+Nandflash。

S5PV510 能用那些介质启动呢?



6.1 OVERVIEW OF BOOTING SEQUENCE

S5PV210 consists of 64KB ROM and 96KB SRAM as internal memory. For internal 96KB SRAM regions can be used. S5PV210 boots from internal ROM ensures that the image cannot be altered by unauthorized users. To select S5PV210 should use e-fuse information. This information cannot be altered.

The booting device can be chosen from following list:

- General NAND Flash memory
- OneNAND memory
- SD/ MMC memory (such as MoviNAND and iNAND)
- eMMC memory
- eSSD memory
- UART and USB devices

支持 NAND flash 启动，支持 SD 卡启动支持 EMMC 启动支持 eSSD 启动，甚至支持 USB 启动

S5PV210 启动过程

内存:

SRAM 特点：容量小，价格高，不需要写时序去初始化，上电就可以直接用指针写地址数据。

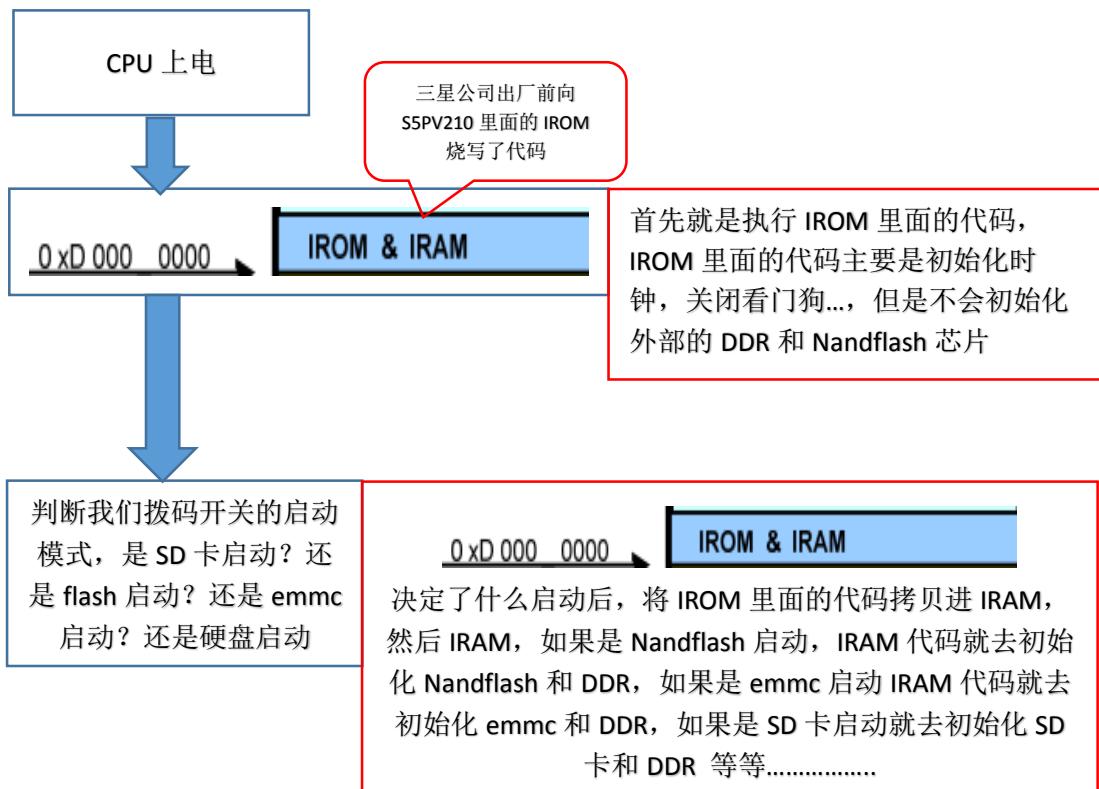
DRAM 特点：DDR2,DDR3 容量大，价格低，上电后不能直接用指针写地址数据，要用代码去初始化 DRAM 里面的寄存器。

外存:

Norflash

Nandflash

通用的系统启动过程是下面这样：



所以我们的 IRAM 里面装的是 Uboot 代码，Uboot 代码就是去初始化外存和 DDR 内存的。

但是一般的 Uboot 都是 190 多 K，起码 100K 以上的代码量。

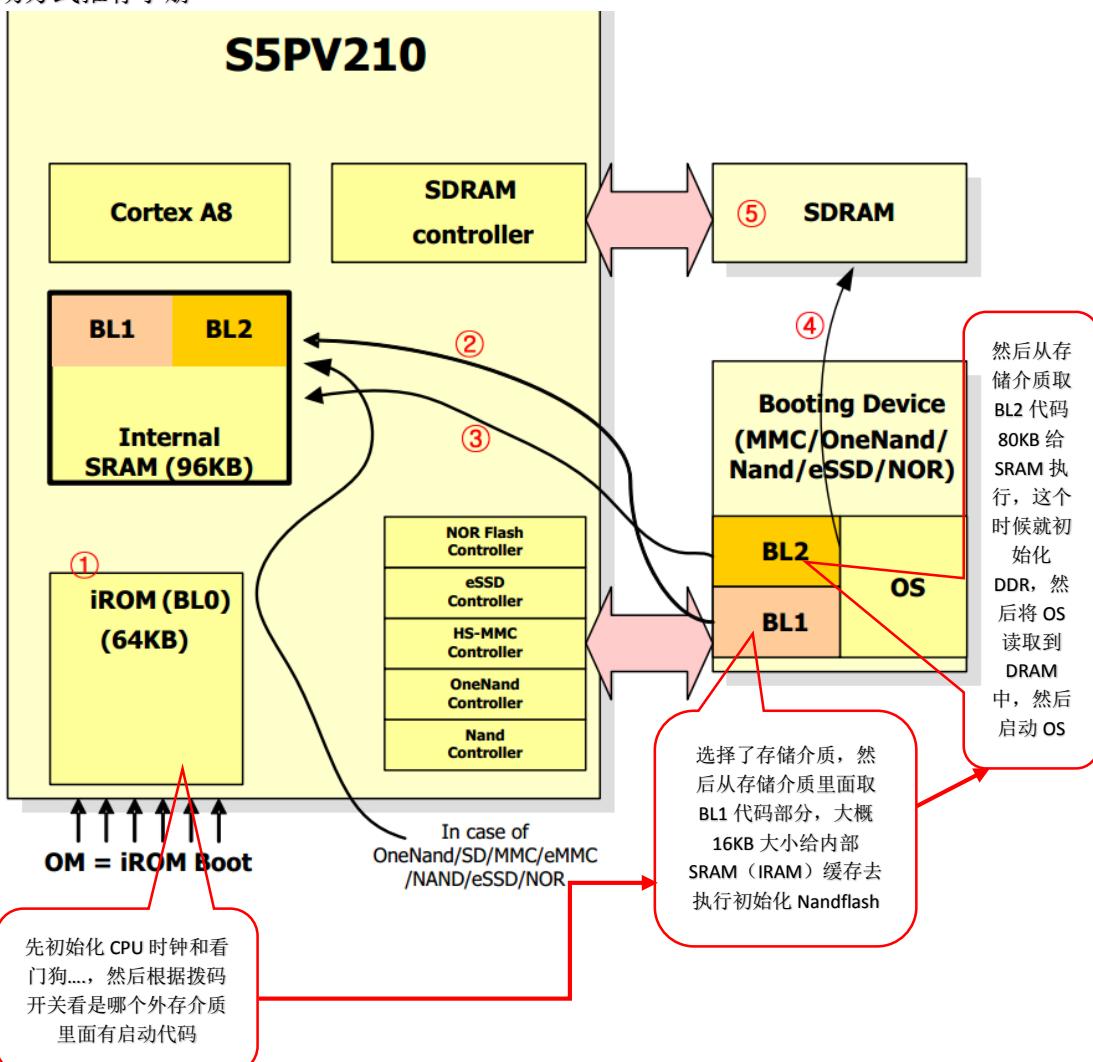
0xD000_0000	0xD000_FFFF	64KB	IROM
0xD001_0000	0xD001_FFFF	64KB	Reserved
0xD002_0000	0xD003_7FFF	96KB	IRAM

但是我们 S5PV210 里面内置的 IROM 64KB 和 IRAM 96KB，根本不够，而且不同的 Nandflash 和不同的 DDR 时序，初始化寄存器都不一样，三星怎么知道我们要初始化哪个型号？所以前面的初始化流程是对的，但是初始化代码放的位置没有对。

三星解决方法就是把 uboot 分成 2 半，一半是 BL1 一半是 BL2，两部分协同工作完成启动

[S5PV210_iROM_ApplicationNote_Preliminary_20091126.pdf](#)

这是三星 S5PV210 uboot 启动方式推荐手册

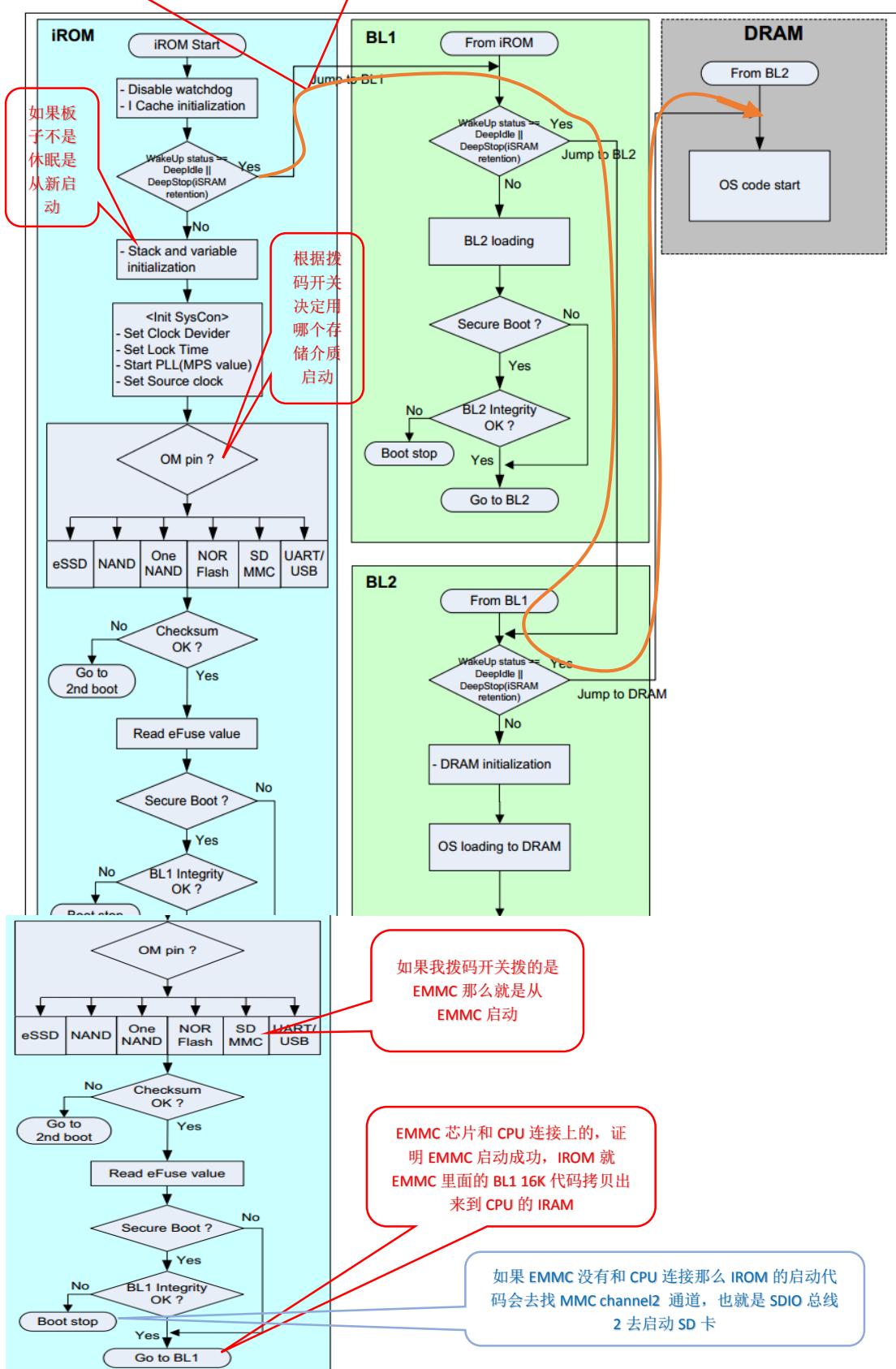


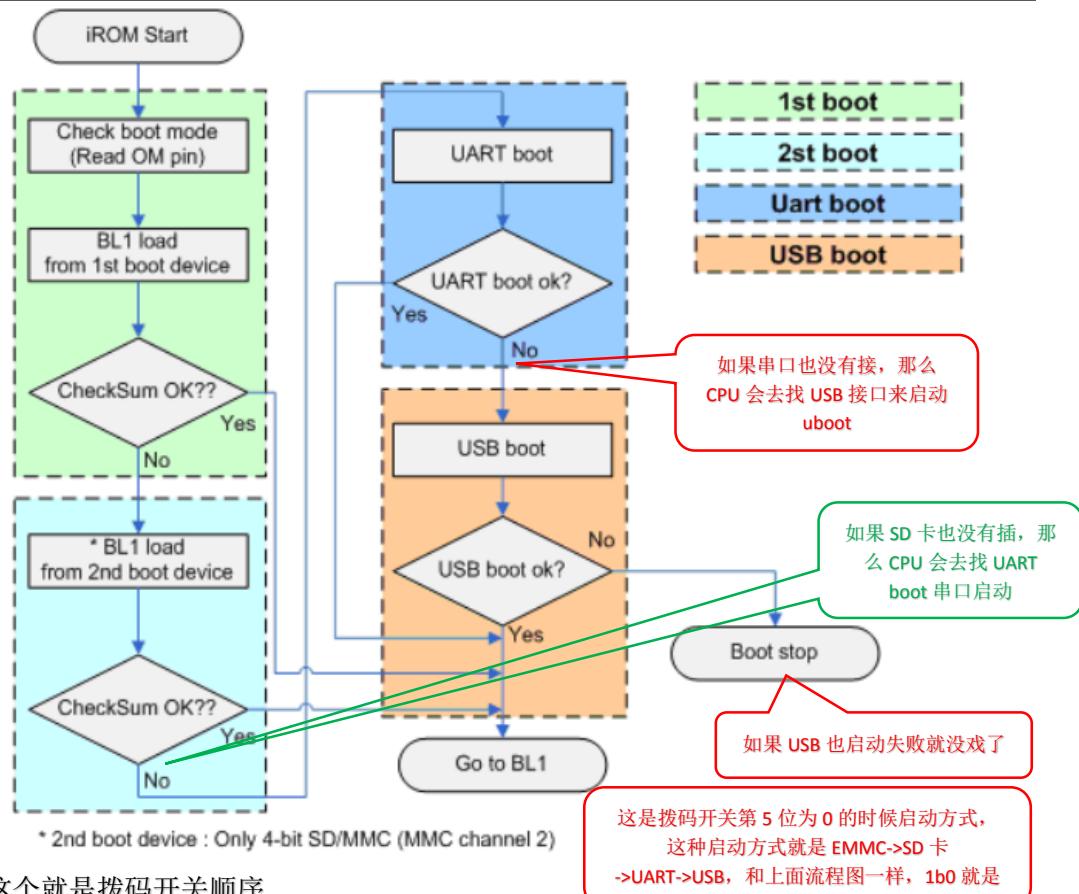
这是三星官方的推荐方法，但是并不是每个写 Uboot 的人都完全按照这个方法来，但是大部分还是按照官方来。

开发板根据不同存储介质的启动过程，有 SD 卡启动，EMMC/Nandflash 启动还有 USB 启动

如果板子是休眠之后唤醒那么 CPU 就不用再去从 Nandflash/EMMC/SD 卡拷贝 uboot 代码到 IROM，因为板子是启动好了的，只是唤醒而已，程序已经在 DRAM 里面放着了

2.3 VZ1U boot-up diagram





下面这个就是拨码开关顺序

OM[5]	OM[4]	OM[3]	OM[2]	OM[1]	OM[0]	OM[5]	OM[4]	OM[3]	OM[2]	OM[1]	OM[0]
I-ROM						Boot Mode					
1'b0			1'b0	1'b0	1'b0			eSSD		X-TAL	
				1'b1	1'b1				X-TAL(USB)		
			1'b1	1'b0	1'b0			Nand 2KB, 5cycle (Nand 8bit ECC)	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b0			Nand 4KB, 5cycle (Nand 8bit ECC)	X-TAL		
				1'b1	1'b0				X-TAL(USB)		
			1'b1	1'b1	1'b1			Nand 4KB, 5cycle (Nand 16bit ECC)	X-TAL		
				1'b0	1'b0				X-TAL(USB)		
			1'b0	1'b1	1'b1			OnenandMux(Audi)	X-TAL		
				1'b1	1'b0				X-TAL(USB)		
			1'b1	1'b1	1'b1			OnenandDemux(Audi)	X-TAL		
				1'b0	1'b0				X-TAL(USB)		
			1'b1	1'b0	1'b1			SD/MMC	X-TAL		
				1'b1	1'b0				X-TAL(USB)		
			1'b0	1'b0	1'b0			eMMC(4-bit)	X-TAL		
				1'b1	1'b0				X-TAL(USB)		
			1'b1	1'b1	1'b0			Nand 2KB, 5cycle (16-bit bus, 4-bit ECC)	X-TAL		
				1'b0	1'b1				X-TAL(USB)		
			1'b1	1'b0	1'b1			Nand 2KB, 4cycle (Nand 8bit ECC)	X-TAL		
				1'b1	1'b0				X-TAL(USB)		
			1'b0	1'b0	1'b0			iROM NOR boot	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b1	1'b1	1'b1			eMMC(8-bit)	X-TAL		
				1'b0	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b0			eSSD	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b0			Nand 2KB, 5cycle	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b1	1'b0	1'b0			Nand 4KB, 5cycle	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b0			Nand 16bit ECC (Nand 4KB, 5cycle)	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b1	1'b0	1'b1			OnenandMux(Audi)	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b1			OnenandDemux(Audi)	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b1	1'b0	1'b0			SD/MMC	X-TAL		
				1'b1	1'b1				X-TAL(USB)		
			1'b0	1'b0	1'b0			eMMC(4-bit)	X-TAL		
				1'b1	1'b1				X-TAL(USB)		

这是拨码开关第5位为1的时候启动方式,
这种启动方式就是先从UART启动->USB启动

->EMMC->SD卡, 这个启动顺序和上面流程图的启动顺序就不一样了

OM[5]	OM[4]	OM[3]	OM[2]	OM[1]	OM[0]	OM[5]	OM[4]	OM[3]	OM[2]	OM[1]	OM[0]
0	0	0	0	0	1'b0			eSSD		X-TAL	
					1'b1					X-TAL(USB)	
0	0	0	0	1'b1	1'b0			Nand 2KB, 5cycle (Nand 8bit ECC)		X-TAL	
					1'b1					X-TAL(USB)	
是接的 sh 页大小 个时钟地 线 8 位, 就这样拨	1'b0	1'b1	1'b0	1'b0	1'b0	I-ROM	Boot Mode	Nand 4KB, 5cycle (Nand 8bit ECC)		X-TAL	
					1'b1					X-TAL(USB)	
			1'b1	1'b0	1'b0			Nand 4KB, 5cycle (Nand 16bit ECC)		X-TAL	
					1'b1					X-TAL(USB)	
		1'b1	1'b1	1'b0	1'b0			OnenandMux(Audi)		X-TAL	
					1'b1					X-TAL(USB)	
			1'b1	1'b1	1'b0			OnenandDemux(Audi)		X-TAL	
					1'b1					X-TAL(USB)	
	1'b1	1'b0	1'b1	1'b0	1'b0	I-ROM	First boot UART >USB	SD/MMC		X-TAL	
					1'b1					X-TAL(USB)	
			1'b1	1'b0	1'b0			eMMC(4-bit)		X-TAL	
					1'b1					X-TAL(USB)	
		1'b1	1'b1	1'b0	1'b0			Nand 2KB, 5cycle (16-bit bus, 4-bit ECC)		X-TAL	
					1'b1					X-TAL(USB)	
			1'b1	1'b0	1'b0			Nand 2KB, 4cycle (Nand 8bit ECC)		X-TAL	
					1'b1					X-TAL(USB)	
	1'b1	1'b0	1'b0	1'b1	1'b0	I-ROM	First boot UART >USB	iROM NOR boot		X-TAL	
					1'b1					X-TAL(USB)	
				1'b1	1'b0			eMMC(8-bit)		X-TAL	
					1'b1					X-TAL(USB)	
		1'b1	1'b1	1'b0	1'b0			eSSD		X-TAL	
					1'b1					X-TAL(USB)	
				1'b1	1'b0			Nand 2KB, 5cycle		X-TAL	
					1'b1					X-TAL(USB)	
		1'b1	1'b1	1'b0	1'b0	I-ROM	First boot UART >USB	Nand 4KB, 5cycle		X-TAL	
					1'b1					X-TAL(USB)	
				1'b1	1'b0			Nand 16bit ECC (Nand 4KB, 5cycle)		X-TAL	
					1'b1					X-TAL(USB)	
			1'b1	1'b0	1'b0			OnenandMux(Audi)		X-TAL	
					1'b1					X-TAL(USB)	
				1'b1	1'b0			OnenandDemux(Audi)		X-TAL	
					1'b1					X-TAL(USB)	
		1'b1	1'b1	1'b0	1'b0	I-ROM	First boot UART >USB	SD/MMC		X-TAL	
					1'b1					X-TAL(USB)	
				1'b1	1'b0			eMMC(4-bit)		X-TAL	
					1'b1					X-TAL(USB)	

如何设置 SD 卡启动 S5PV210

因为我们开发板是设置的 iNAND 存储器启动，根据前面的启动过程我们知道只有破坏了 iNAND 存储器里面的 uboot 系统，导致 CPU 找不到 iNAND 里面的 BL1，那么 CPU 就会去找 SDIO2 总线上的存储器启动，SD 卡就是接在 SDIO2 总线上的。

因为我们买的开发板已经烧录 Uboot 和安卓系统在 iNAND 上了，所以我们来破坏它

linux dd 命令

将一个文件的内容拷贝给另外一个文件，不是 `cp` 哦，`cp` 是一个文件覆盖另外一个文件，`dd` 是将一个文件的内容拷贝给另外一个文件，但是不是覆盖。

比如我要把 text 文件内容拷贝给 text2

```
root@ubuntu:/home/xiang/textdu# ls
```

text text2

1 aaa

text 文件

text2 文件

```
root@ubuntu:/home/xiang/textdu# dd if=text of=text2  
0+1 records in  
0+1 records out  
124 bytes (124 B) copied, 0.0123808 s, 10.0 kB/s  
root@ubuntu:/home/xiang/textdu#
```

If 是要读出哪个文件的内容

Of 是将 if 读出的内容写入
哪个文件

这就是 text2 文件的内容，和 text 文件内容

一样，记住 dd 命令不是追加，是直接将 text 文件内容，完全拷贝给 text2，覆盖 text2 文件里面的内容。

我们没有给 dd 命令加更多的后缀，那么就是 text 文件里面的内容完全拷贝给 text2.

如果你想把 `text` 文件里面的内容，按照固定大小拷贝给 `text2` 的话，就要在 `dd` 后面加后缀

```
root@ubuntu:/home/xiang/textdu# dd if=text of=text2 count=1 bs=10
1+0 records in
1+0 records out
10 bytes (10 B) copied, 0.000410615 s, 24.4 kB/s
root@ubuntu:/home/xiang/textdu# vim text2
root@ubuntu:/home/xiang/textdu#
```

count 指定拷贝多少个块，这里 count=1 表示拷贝一个块，如果不加 count 这个后面 bs 就无效
这个块里面拷贝多少字节给 text2

1 1111111111

你看 text2 的文件就只接受了 text 文件的前面 10 个字符。这就是指定 text 拷贝多少个字节给 text2

从第 1 个扇区开始，要写多少个扇区，我们这里就写一个扇区

`dd`最有用的是可以对`/dev`目录下面的文件进行覆盖，我们来破坏`iNAND`就是用`dd`

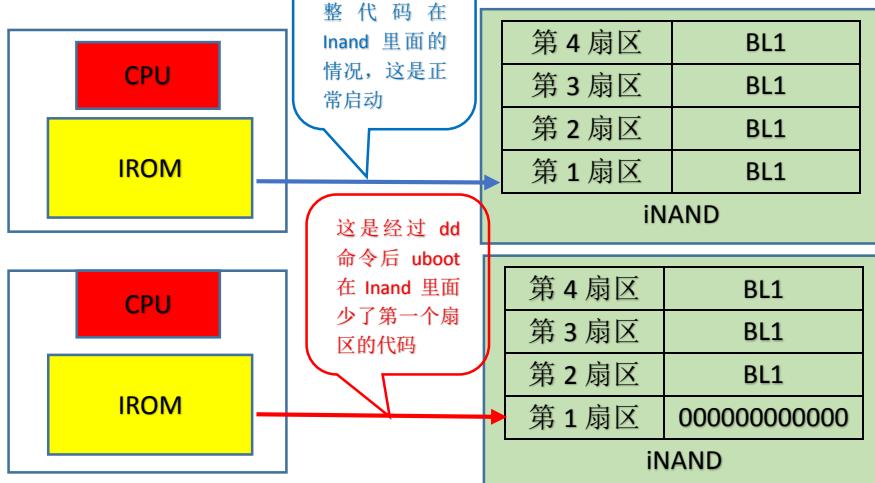
```
/ # busybox dd if=/dev/zero of=/dev/block/mmcblk0 count=1 seek=1 bs=512  
1+0 records in  
1+0 records out  
#+  
# iNAND 扇区这么多我们到底要写哪个扇区,  
# 这里... 1 < 表示我们就写第一个扇区  
#+  
# 因为 Inand 的块大小是 512,  
# 所以我们就堵住 512 字节
```

执行这条 busybox 命令后，就将 /dev/zero 文件里面的全 0，写给了 /dev/block/mmcblk0，因为 mmcblk0 就是 iNAND 存储器的设备节点。

```
/ # sync  
然后再输入 sync，就可以重启开发板了。  
SD checksum Error  
  
SD Init Error  
  
Uart negotiation Error  
  
Insert an OTG cable
```

你看我们关机后再次启动就启动不起来了，因为 Uboot 的第一个扇区 BL1 读入 IROM 后，IROM 计算校验和发现 Inand 第 1 个扇区不是 uboot

原理就是下面这个框图



所以板子启动后 IROM 读取到的 iNAND 里面，第一个扇区里面数据全部是 0，就认为没有 uboot，然后会跳转到 SDIO2 总线去读 SD 卡，看 SD 卡里面有没有 Uboot 启动代码。

SD 卡制作成启动介质的方法

现在我们将 SD 卡做成启动卡，要把 SD 卡做成支持 Uboot 格式的存储介质。

第一种用 windows 里面的刷卡工具制作。

第二种用 linux dd 命令来制作。

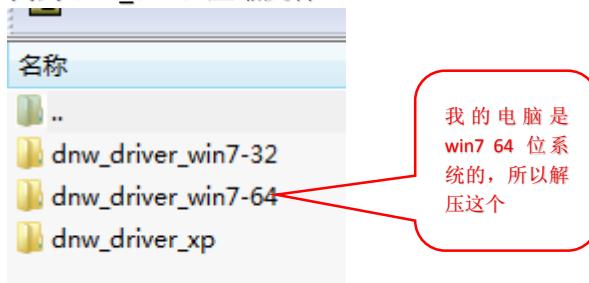
Windows 刷卡工具制作流程。

现在制作 SD 卡有点问题.....后面处理

DNW 软件下载 uboot

dnw_driver.rar

找到 dnw_driver 压缩文件



解压之后千万不要马上安装 dnw 驱动，先看看安装说明，要先破解微软的数字签名。

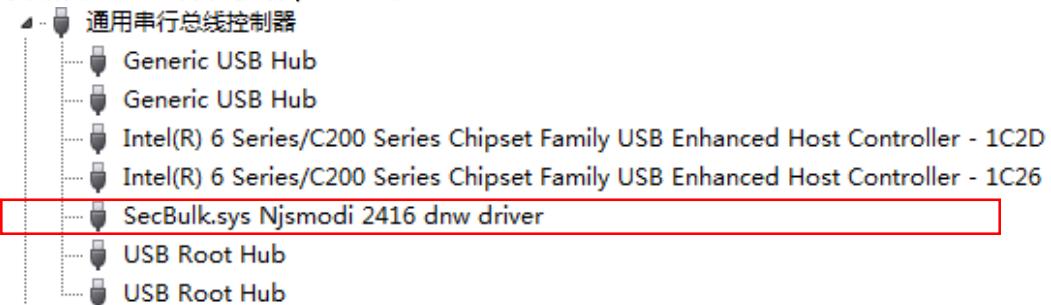
4. 重启电脑

5. 打开设备管理器 更新 SEC S5PC110 Test B/D.的驱动程序

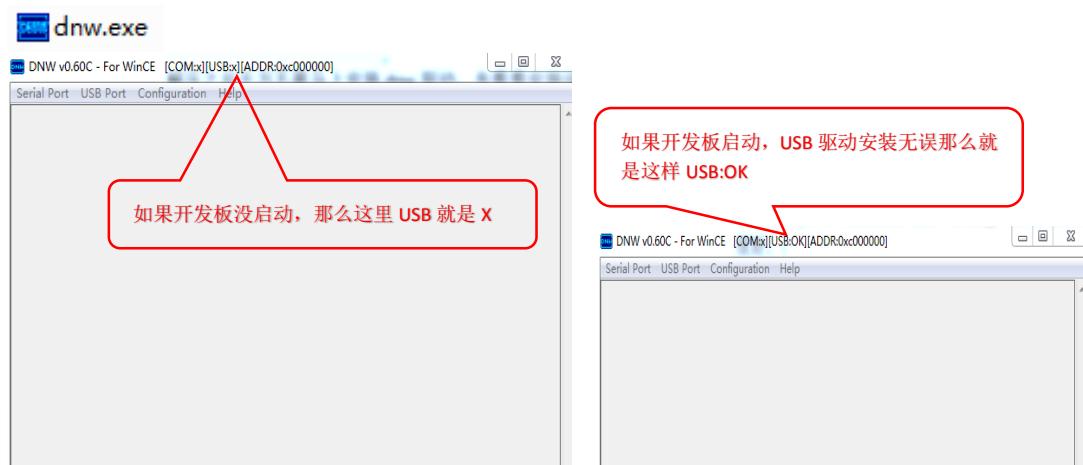
我们打开设备管理器之后没有看到 SEC S5PC110 Test B/D.的驱动程序，这是因为我们开发板是要按照 power 键才能启动的，如果不一直按着是不行的。所以这个九鼎开发板做的不是很好。

接着 POWER 键不放就会出现一个有感叹号的东西，然后按照下面第 6 步更新就是

6. 选择 D:\9tripodDownloads\DNWforwin7\win7-64 dnw usb driver\inf64 目录下的.inf 文件
更新完成后，再次按着 power 键

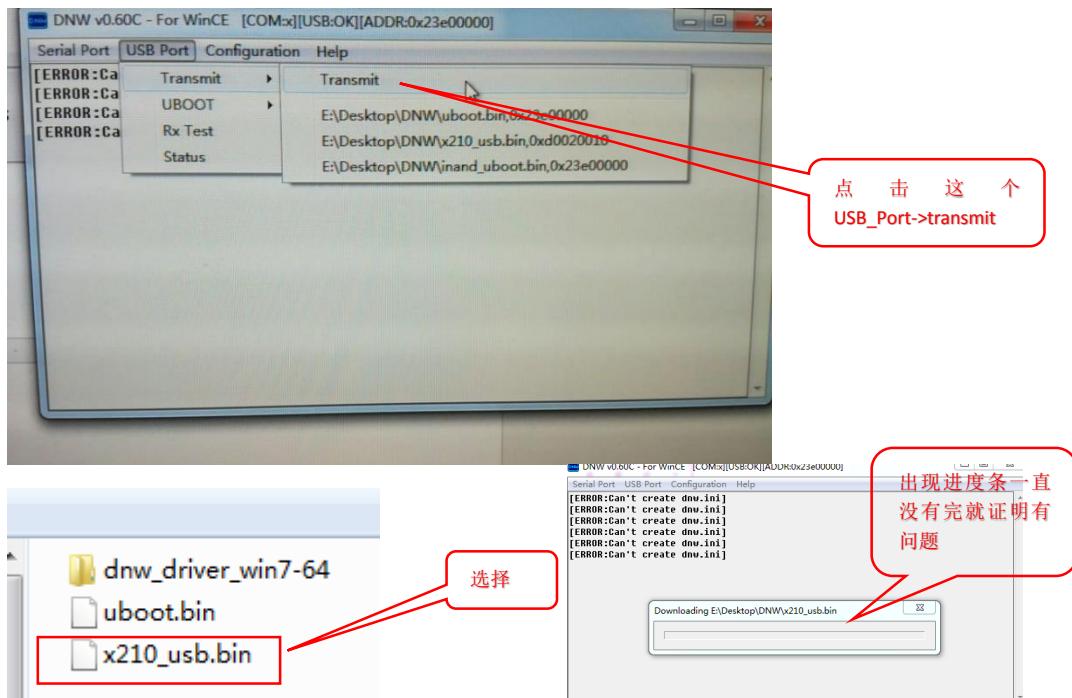


这就表示驱动安装成功了。然后执行 dnw.exe 软件

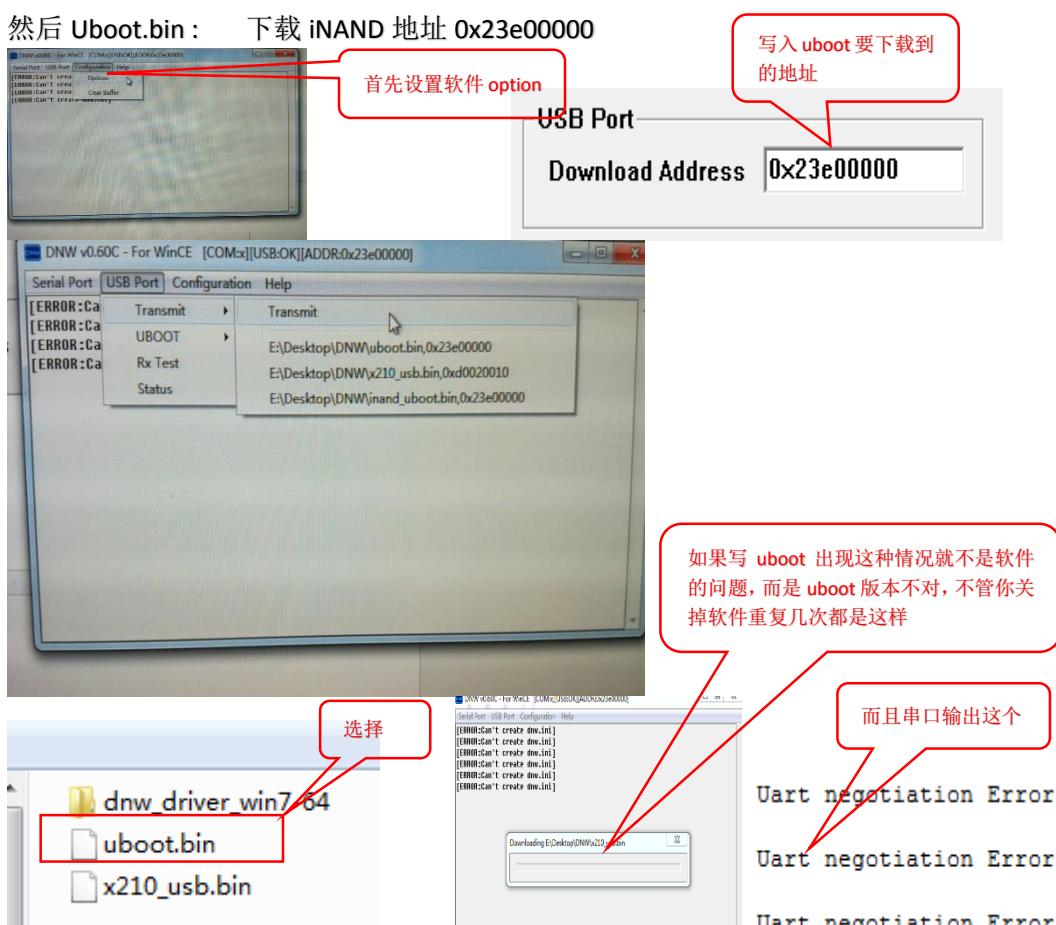


X210_usb.bin：下载 iNAND 地址 0xd0020010





这种情况就是重启 dnw 软件，然后再次下载，一般多来几次就正常了。
正常情况下就是进度条闪一下就结束了。



这个 uboot 版本不对把我折腾了一下午，后来在不同开发板目录里面 X210V3S 目录里面找到了正确的 uboot 版本，正确的 uboot 版本下载后进度条应该也是一闪而过。

如果 Windows7 使用 dnw 下载 uboot 不行，换成 linux 使用 dnw

首先确保 windows7 安装了 dnw 的 usb 驱动，安装驱动的方法看上面。

找到 dnw 的 linux 压缩包

dnw-linux-x210.tar.bz2

解压

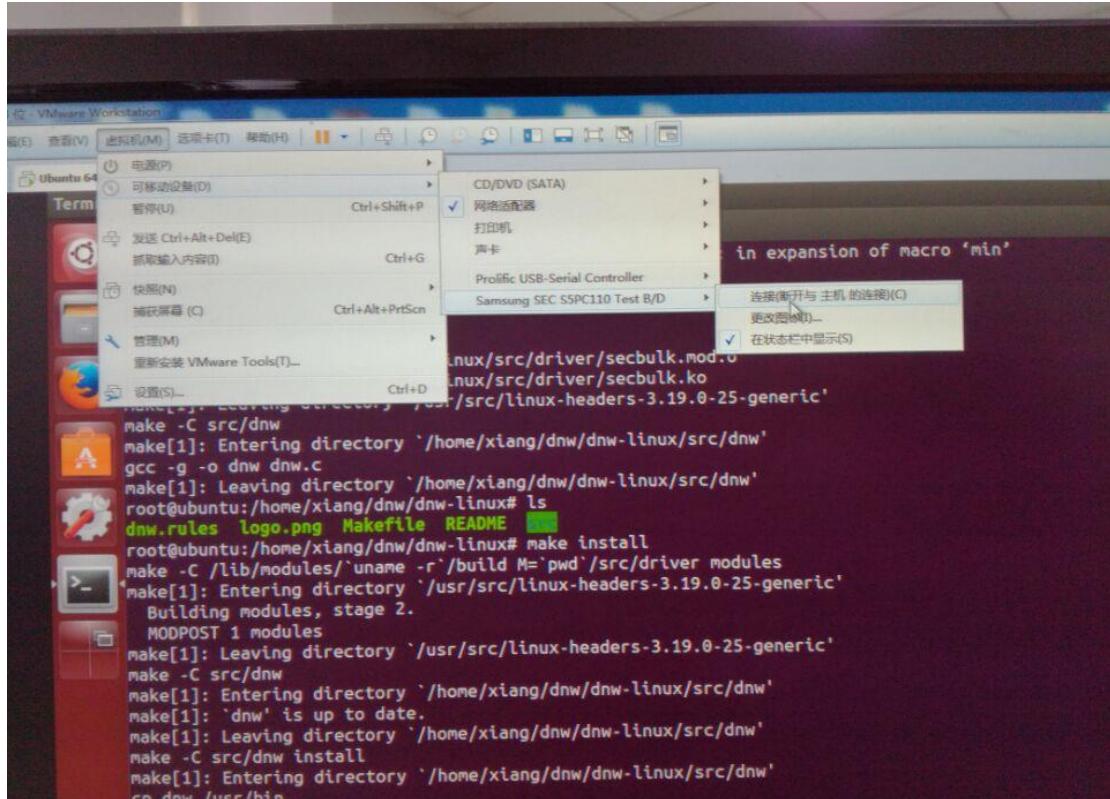
```
root@ubuntu:/home/xiang/dnw/dnw-linux# ls  
dnw.rules logo.png Makefile README src
```

进入 dnw-linux 目录

Make

Make install

经过上面两个命令，dnw 的 linux 驱动已经安装完成了。



因为我们的虚拟机，所以要把 windows 的 DNW 驱动转接过来。

然后给开发板插上 USB，因为开发板本身设计原因，所以要长按 power 按键

ls /dev/

在 /dev 目录下如果出现这个设备节点 **secbulk0**

secbulk0

就证明开发板 USB 是和 linux 虚拟机连接的。如果没有 secbulk0 节点，那么开发板就是和 windows7 连接的。

然后将 uboot.bin 和 X210_usb.bin 复制到 /dnw-linux/src/dnw 目录下

```
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# ls  
dnw dnw.c Makefile uboot.bin x210_usb.bin
```

然后执行 X210_usb.bin 和 uboot.bin 烧写

```
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# dnw -a 0xd0020010 x210_usb.bin  
optarg = 0xd0020010.  
load address: 0xD0020010  
Writing data...  
100%    0x00002BAA bytes (10 K)  
speed: 2.665043M/S  
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# dnw -a 0x23e00000 uboot.bin  
optarg = 0x23e00000.  
load address: 0x23E00000  
Writing data...  
100%    0x00006000A bytes (384 K)  
speed: 2.005399M/S  
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# ls
```

这就是烧写 X210_usb，指定 iNAND 地址为 0xd0020010，地址一定不要写错

这就是烧写 uboot，指定 iNAND 地址为 0x23e00000，地址一定不要写错

烧写成功后会有 speed:输出

下面这个是烧写失败的输出

```
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# dnw -a 0xd0020010 x210_usb.bin
optarg = 0xd0020010.
Load address: 0xD0020010
Writing data...
100%    0x00002BAA bytes (10 K)
speed: 2.665043M/S
root@ubuntu:/home/xiang/dnw/dnw-linux/src/dnw# dnw -a 0x23e00000 uboot.bin
optarg = 0x23e00000.
load address: 0x23E00000
Writing data...
write failed: Bad address
```

这就是 X210_USB 烧写成功，uboot 烧写失败。根据我一下午的测试发现烧写失败的原因是 uboot 版本不对

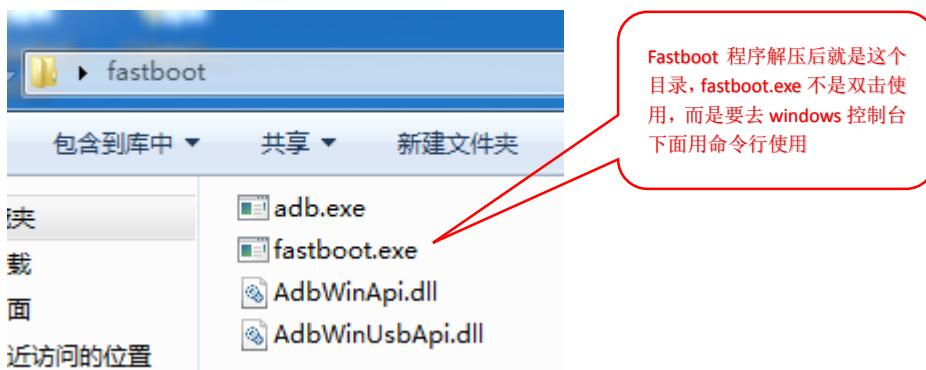
以上的 dnw 是将 uboot 烧入 DDR, 而不是烧入 EMMC/iNAND, 你关机之后然后重启，内存的 uboot 都消失了，所以还是启动不起来。dnw 是为后面 fastboot 烧录做准备。fastboot 烧录才是把 uboot, zimage, 和 rootfs 文件系统全部烧入 EMMC/iNAND

Fastboot 如何将 uboot, zimage, rootfs 烧录进 EMMC/iNAND/闪存

首先还是把 USB 连接电脑和开发板的 USB_OTG

 fastboot.rar 安装 fastboot 驱动程序，

 .. 解压 fastboot 应用程序



然后让开发板处于 DDR 的 uboot 状态，所以这个时候 DNW 就很重要了。

```
x210 # fastboot
[Partition table on MoviNAND]
ptn 0 name='bootloader' start=0x0 len=N/A (use hard-coded info. (cmd: movi))
ptn 1 name='kernel' start=N/A len=N/A (use hard-coded info. (cmd: movi))
ptn 2 name='ramdisk' start=N/A len=0x300000 (~3072KB) (use hard-coded info. (cmd: movi))
ptn 3 name='config' start=0xAEC00 len=0x1028DC00 (~264759KB)
ptn 4 name='system' start=0x10D7A800 len=0x1028DC00 (~264759KB)
ptn 5 name='cache' start=0x21008400 len=0x65F7000 (~104412KB)
ptn 6 name='userdata' start=0x275FF400 len=0xC906FC00 (~3158463KB)
Fastboot disconnect detected
Fastboot disconnect detected
Insert a OTG cable into the connector!
Received 17 bytes: download:00060000
Starting download of 393216 bytes
```

在 uboot 模式下输入 fastboot

然后终端输出 fastboot 能
烧写哪些区域？比如
bootloader 区域

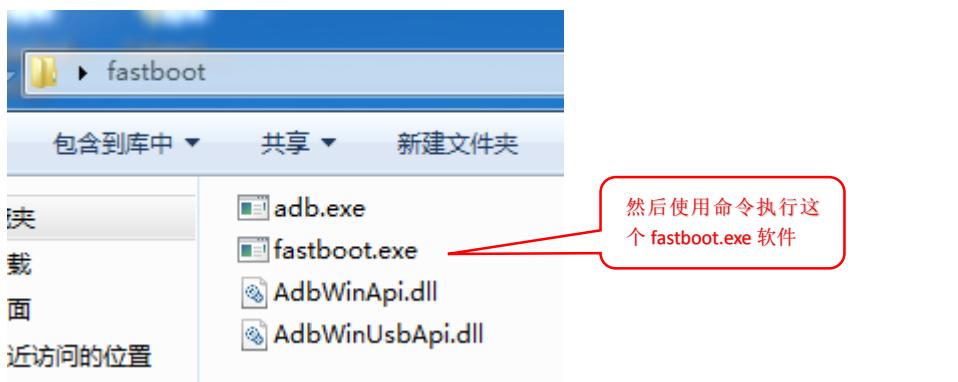
比如内核区域

比如文件系统区域

然后我们看到电脑自动识别了 USB 驱动，然后电脑自动帮你装 USB 驱动。



但是我运气好，电脑自动帮我装了 fastboot 驱动



这个软件我解压在 E 盘桌面的，进入命令控制台

```
C:\Users\xiang>E:  
E:\Desktop\fastboot>
```

在 E 后面加冒号就是切换到 E 盘

然后用 dir(类似 linux 的 ls)查看 E 盘的文件

```
E:\Desktop>cd fastboot  
E:\Desktop\fastboot>
```

用 cd 进入 E 盘桌面的 fastboot 目录

```
E:\Desktop\fastboot>dir  
驱动器 E 中的卷是 软件  
卷的序列号是 0009-4A2B  
E:\Desktop\fastboot 的目录  
  
2012/04/05 18:04 <DIR> .  
2012/04/05 18:04 <DIR> ..  
2010/06/10 20:13 578,611 adb.exe  
2010/06/10 20:13 96,256 AdbWinApi.dll  
2010/06/10 20:13 60,928 AdbWinUsbApi.dll  
2012/03/20 11:03 992,202 fastboot.exe  
4 个文件 1,727,997 字节  
2 个目录 156,185,014,272 可用字节  
E:\Desktop\fastboot>
```

用 dir 查看 fastboot 里面的软件

软件在这里

```
E:\Desktop\fastboot>fastboot devices
```

```
E:\Desktop\fastboot>
```

然后执行 fastboot devices 看看开发板是不是链接上电脑了，这里发现是空的，看来没有连接上。有可能是因为 uboot 没有启动。所以要用 dnm 把 uboot 下载进去。然后启动开发板停在 uboot 界面下，执行 fastboot，看有没有设备节点，如果还没有，先不慌关开发板。

拔掉 USB 线再插上看看。

```
E:\Desktop\fastboot>fastboot devices  
SMDKG110-01      fastboot
```

```
E:\Desktop\fastboot>
```

这不是就有设备节点了吗

```
E:\Desktop\fastboot>fastboot flash bootloader ..\DNW\uboot.bin  
sending 'bootloader' <384 KB>... OKAY  
writing 'bootloader'... OKAY
```

我的 uboot 放在 DNW 目录下的

执行 fastboot 下载 uboot 命令

这就是显示下载成功

```
x210 # fastboot  
[Partition table on MoviNAND]  
ptn 0 name='bootloader' start=0x0 len=N/A (use hard-coded info. (cmd: movi))  
ptn 1 name='kernel' start=N/A len=N/A (use hard-coded info. (cmd: movi))  
ptn 2 name='ramdisk' start=N/A len=0x300000 (~3072KB) (use hard-coded info)  
ptn 3 name='config' start=0xAECC00 len=0x1028DC00 (~264759KB)  
ptn 4 name='system' start=0x10D7A800 len=0x1028DC00 (~264759KB)  
ptn 5 name='cache' start=0x21008400 len=0x65F7000 (~104412KB)  
ptn 6 name='userdata' start=0x275FF400 len=0xC0C6FC00 (~3158463KB)  
Fastboot disconnect detected  
Fastboot disconnect detected  
Insert a OTG cable into the connector!  
Received 17 bytes: download:00060000  
Starting download of 393216 bytes
```

开发板这边也会显示向 EMMC 写入了 uboot

```
downloading of 393216 bytes finished  
Received 16 bytes: flash:bootloader  
flashing 'bootloader'  
Writing BL1 to sector 1 (16 sectors)... checksum : 0xed75e  
writing bootloader.. 49, 1024  
MMC write: dev # 0, block # 49, count 1024 ... 1024 blocks written: OK  
completed  
partition 'bootloader' flashed
```

这样 uboot 就完全烧写在 EMMC/iNAND 里面了，你关了开发板重启，uboot 都还在。

然后我们用 fastboot 下载内核 zimage

```
E:\Desktop\fastboot>fastboot flash kernel ..\DNW\android\zImage-android  
sending 'kernel' <3800 KB>... OKAY  
writing 'kernel'... OKAY
```

我们烧写的是 android 内核

```
downloading of 3891200 bytes finished  
Received 12 bytes: flash:kernel  
flashing 'kernel'  
writing kernel.. 1073, 8192  
MMC write: dev # 0, block # 1073, count 8192 ... 8192 blocks written: OK  
completed  
partition 'kernel' flashed
```

开发板串口接收到了 android 内核烧写

然后我们用 fastboot 下载文件系统

```
E:\Desktop\fastboot>fastboot flash system ../DNW/android/x210.img  
sending 'system' <262144 KB>...
```

然后你看一直卡在 `sending` 这里，那是因为文件系统太大了，你看串口的....在走



```
E:\Desktop\fastboot>fastboot flash system ..\DNN\android\x210.img  
sending 'system' <262144 KB>... OKAY
```

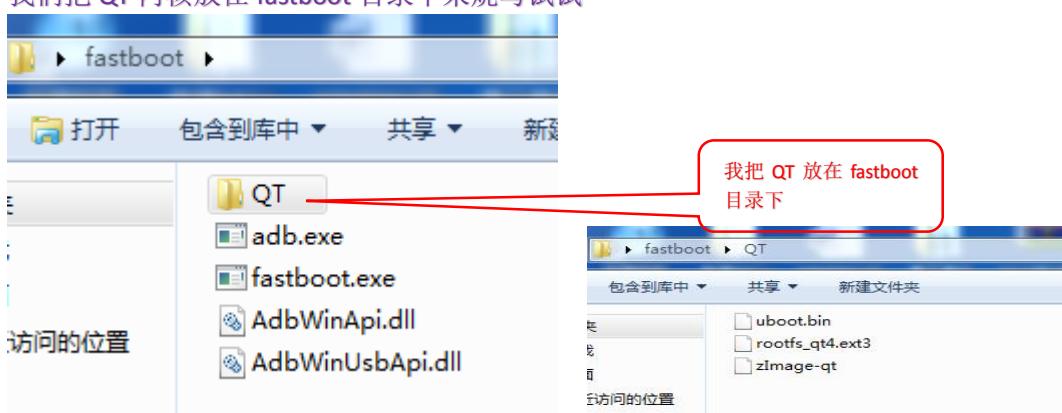
writing 'system'... OKAY 其实这样这样写很麻烦。内核文件应该放在 fastboot 目录下，这样就不需要手工寻找路径了。

你看最后 cmd 显示 OKAY 表示烧写成功

```
Starting download of 268435456 bytes
.....
.....
.....
.....
download of 268435456 bytes finished
Received 12 bytes: flash:system
flashing 'system'

MMC write: dev # 0, block # 551892, count 529518 ... 529518 blocks written: OK
串口也是这样显示OK
```

我们把 QT 内核放在 fastboot 目录下来烧写试试



```
E:\Desktop\fastboot>fastboot devices
SMDKC110-01      fastboot

E:\Desktop\fastboot>fastboot flash bootloader QT\uboot.bin
sending 'bootloader' <384 KB>... OKAY
writing 'bootloader'... OKAY

E:\Desktop\fastboot>fastboot flash kernel QT\zImage-qt
sending 'kernel' <3566 KB>... OKAY
writing 'kernel'... OKAY

E:\Desktop\fastboot>fastboot flash system QT\rootfs_qt4.ext3
sending 'system' <262144 KB>... OKAY
writing 'system'... OKAY

E:\Desktop\fastboot>
```

这样 QT 系统就已经烧写好了。

Linux 下安装软件的三种方法

1. `apt-git install vim` 这就是按照 vim 软件，这种直接在线安装是最好的，最不容易出问题。
2. linux 版本的某个软件压缩包拷贝进 ubuntu，然后解压安装，这种就有可能会出现你这个压缩包里面的软件和你 ubuntu 版本不兼容的问题，当然也有软件兼容，看你自己。像前面的 `apt-git` 这种方式是根据 ubuntu 里面 `apt-git` 表栏里面写的网站去寻找软件源，然后软件源服务器根据你本地的 ubuntu 版本来选择兼容的软件给你安装。
3. 直接去 `github` 之类的网站拷贝源代码到 ubuntu，然后用 `gcc` 编译。这种也是一种比较好的安装软件方式，比如我们前面的 linux 版本 `dnw` 就是这种方式。

Linux 文件系统目录管理

```
root@ubuntu:/bin#
```

/bin 目录是用来存放 ubuntu 系统自带的一些常用命令

```
cat          echo  
chacl        ed  
chgrp        egrep  
chmod        false  
chown        fgconsole
```

```
root@ubuntu:/bin#
```

```
root@ubuntu:/sbin#
```

/sbin 是用来放一些 ubuntu 系统自带的特殊命令

```
fdisk  
mkfs.ext2  
mkfs.ext3  
mkfs.ext4
```

```
mkfs.fat
```

比如经常使用的 echo,
ls, cat, chmod 这些

比如 fdisk 格式化磁盘，mkfs.ext3
把磁盘格式化成 ext3 格式
mkfs.fat 把磁盘格式成 fat32 格式

```
root@ubuntu:/usr# ls  
arm-linux-gnueabi  arm-linux-gnueabihf  bin  games  include  lib  lib32  libx32  local  sbin  share  src  x86_64-linux-gnu  
root@ubuntu:/usr#
```

/usr 目录下的 bin 和 sbin 是我们自己另外给 ubuntu 安装的软件，所以我们编译的软件最好装在/usr/bin, /usr/sbin 下面，我们编译的库也是放在/usr/lib 下面。不要去掺和系统的目录

安装交叉编译工具链

我们安装交叉编译起用的就是上面的第 2 种方式。



这是交叉编译器版本

我们交叉编译器压缩包，按照上面文件系统目录管理的要求，最好放在/usr/local 目录下，但是我已经习惯放在/opt 目录下了，所以我解压到/opt 目录下

```
root@ubuntu:/opt# ls  
arm-2009q3  arm-2009q3.tar.bz2  
root@ubuntu:/opt#
```

解压后进入 arm-2009q3 目录

```
root@ubuntu:/opt/arm-2009q3# ls  
arm-none-linux-gnueabi  bin  lib  libexec  share  
root@ubuntu:/opt/arm-2009q3#
```

进入 bin

```
root@ubuntu:/opt/arm-2009q3/bin# ls  
arm-none-linux-gnueabi-addr2line  arm-none-linux-gnueabi-g++      arm-none-linux-gnueabi-gprof    arm-none-linux-gnueabi-readelf  
arm-none-linux-gnueabi-ar       arm-none-linux-gnueabi-gcc      arm-none-linux-gnueabi-ld      arm-none-linux-gnueabi-size  
arm-none-linux-gnueabi-as       arm-none-linux-gnueabi-gcov     arm-none-linux-gnueabi-nm      arm-none-linux-gnueabi-sprite  
arm-none-linux-gnueabi-c++      arm-none-linux-gnueabi-gcov     arm-none-linux-gnueabi-objcopy   arm-none-linux-gnueabi-strings  
arm-none-linux-gnueabi-c++filt  arm-none-linux-gnueabi-gdb      arm-none-linux-gnueabi-objdump   arm-none-linux-gnueabi-strip  
arm-none-linux-gnueabi-cpp     arm-none-linux-gnueabi-gdbtui   arm-none-linux-gnueabi-ranlib
```

你看交叉编译器 `arm-linux-gcc`, `g++`, 什么的都在这里。

解压完了我们这个程序就装完了。你看这些绿色的软件都是可以执行的。

我们在当前目录下执行 `./arm-none-linux-gnueabi-gcc -v` 来确定编译器是否可以用

```
root@ubuntu:/opt/arm-2009q3/bin# ./arm-none-linux-gnueabi-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabi
Configured with: /scratch/julian/2009q3-respin-linux-lite/src/gcc-4.4/configure --build=i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi --enable-threads --disable-libmudflap --disable-libssp --disable-libstdcxx-pch --enable-extra-spxxelite-multilibs --with-arch=armv5te --with-gnu-as --with-gnu-ld --with-specs='%{funwind-tables|fno-unwind-tables|mabi=*|ffreestanding|nostdlib:::funwind-tables}%{02%:{!fno-remove-local-statics: fremove-local-statics}}%{0*%{0|00|01|02|0s:::!fno-remove-local-statics: fremove-local-statics}}' --enable-languages=c,c++ --enable-shared --disable-lto --enable-symvers-gnu --enable_cxa_atexit --with-pkgversion='Sourcery G++ Lite 2009q3-67' --with-bugurl=https://support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery --with-sysroot=/opt/codesourcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/scratch/julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/lib --with-gmp=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-ppi=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-mprf=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-cloog=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --enable-poison-system-directories --with-build-time-tools=/scratch/julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/bin --with-build-time-tools=/scratch/julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/bin
Thread model: posix
gcc version 4.4.4 (Sourcery G++ Lite 2009q3-67)
root@ubuntu:/opt/arm-2009q3/bin#
```

如果出现 `gcc version 4.4.4` 就证明这个编译器可以使用没有问题。有些编译器可能版本不一样，反正 `gcc version` 后面跟的版本可以随便变化，但是只要能打印出这个版本就没有问题

```
1 #include<stdio.h>
2
3 int main()
4 {
5     printf("welcome s5pv210\n");
6     return 0;
7 }
```

我用 `arm-none-linux-gnueabi-gcc` 编译 `s5pv210.c` 看行不行

```
root@ubuntu:/home/xiang/S5PV210# ls
s5pv210.c
```

```
root@ubuntu:/home/xiang/S5PV210# ./arm-none-linux-gnueabi-gcc s5pv210 s5pv210.c
bash: ./arm-none-linux-gnueabi-gcc: No such file or directory
root@ubuntu:/home/xiang/S5PV210#
```

很明显是不行的，要不然我把该程序放到 `arm-2009` 的 `bin` 目录下去编译？这样当然可以但是很麻烦啊。

```
root@ubuntu:/home/xiang/S5PV210# /opt/arm-2009q3/bin/arm-none-linux-gnueabi-gcc -o s5pv210 s5pv210.c
root@ubuntu:/home/xiang/S5PV210# ls
s5pv210 s5pv210.c
```

我们把交叉编译器全路径加上就可以了。但是这样每次都要去打很多字也很烦。

我们用环境变量来把编译器路径导入进去。

Linux 里面环境变量很多，每个环境变量的功能都不一样，我们这里要用 `PATH` 这个变量

```
root@ubuntu:/home/xiang/S5PV210# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/games:/opt/freescale/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin:/opt/poky/1.8/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi
```

`PATH` 变量就是为了在你执行任何命令的时候，程序会去 `PATH` 指定的多个路径里面找有没有你这个命令，有就执行，没有你就执行不了。

用 `export` 给 `PATH` 变量增加新的路径

```
root@ubuntu:/home/xiang/S5PV210# export PATH=/opt/arm-2009q3/bin/:$PATH
root@ubuntu:/home/xiang/S5PV210# echo $PATH
/opt/arm-2009q3/bin/:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/games:/opt/freescale/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin:/opt/poky/1.8/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi
```

我们看到 `PATH` 就多了 `arm-2009q3` 的路径，因为 `arm-none-linux-gnueabi-gcc` 在这个路径下，所以 `PATH` 就要把这个路径加进来

```
root@ubuntu:/home/xiang/S5PV210# arm-none-linux-gnueabi-gcc -o s5pv210 s5pv210.c
root@ubuntu:/home/xiang/S5PV210# ls
s5pv210 s5pv210.c
```

你看我直接输入 `arm-none` 然后按 `tab` 就能找到我这个编译器，然后编译出执行程序

但是还是有一个问题，我现在把终端关了再次打开终端就会出现再次找不着编译器的问题

所以我们要把交叉编译路径放在.bashrc

```
root@ubuntu:~# ls -a
. .bash_history bin .con
.. .bashrc .cache .git
root@ubuntu:~#
```

.bashrc 在用户目录下

.bashrc 的功能是每次你去打开命令行终端，.bashrc 都会附带着去执行一次，所以你可以把你想要的任何环境变量或者命令都写进.bashrc

```
8 # See /usr/share/doc/bash-doc/examples in the bash-doc package.
9
10 if [ -f ~/.bash_aliases ]; then
11     . ~/.bash_aliases
12 fi
13 export PATH=$PATH:/opt/freescale/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin
14
15 export PATH=$PATH:/opt/poky/1.8/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi
16
17 export PATH=/opt/arm-2009q3/bin:$PATH
18 #export PATH=$PATH:/opt/poky1.7/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi
19 # enable programmable completion features (you don't need to enable
20 # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
21 # sources /etc/bash.bashrc).
22 #if [ -f /etc/bash_completion ] && ! shopt -o posix; then
23 #fi
```

我在.bashrc 里面加上了我的 S5PV210 交叉编译器路径

其实你如果是关闭终端再次打开，那么你不用对.bashrc 执行 source，当然执行以下最好。

```
root@ubuntu:~# source /etc/bash.bashrc
```

如果你觉得每次编译代码写这个

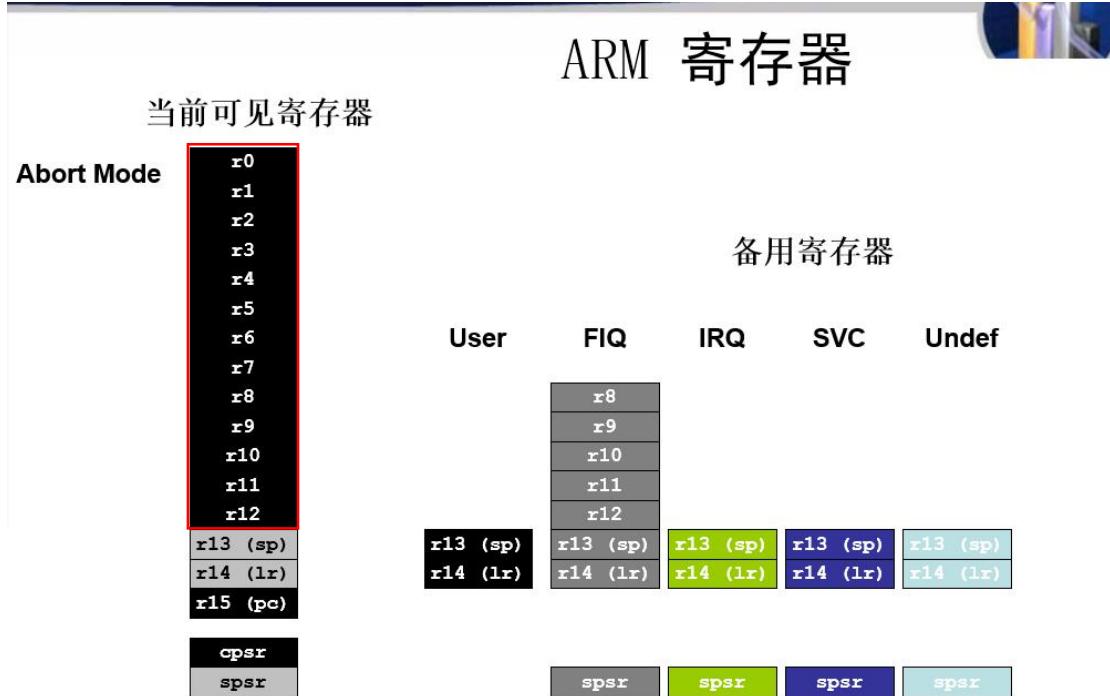
还是太长了的话，可以给这个编译器做一个快捷方式

```
root@ubuntu:/opt/arm-2009q3/bin# ln arm-none-linux-gnueabi-gcc -s arm-linux-gcc
root@ubuntu:/opt/arm-2009q3/bin# ls
arm-linux-gcc          arm-none-linux-gnueabi-cpp      arm-none-linu
arm-none-linux-gnueabi-addr2line  arm-none-linux-gnueabi-g++  arm-none-linu
```

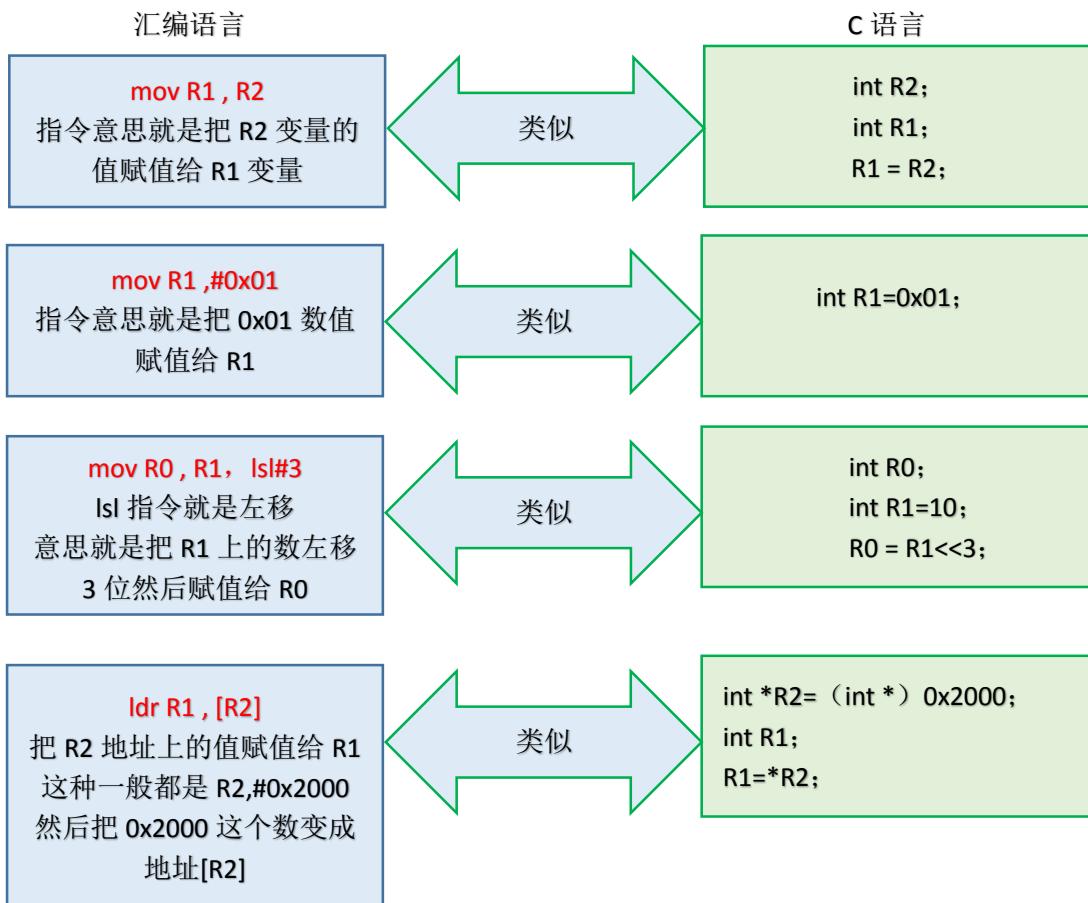
用 ln 创建个符号链接，也就是快捷方式。这样你就可以直接用 arm-linux-gcc，但是你一定要记住这个 arm-linux-gcc 链接的是哪个平台的交叉编译器哦！！，我这里是 S5PV210 平台

汇编程序编写 LED 驱动

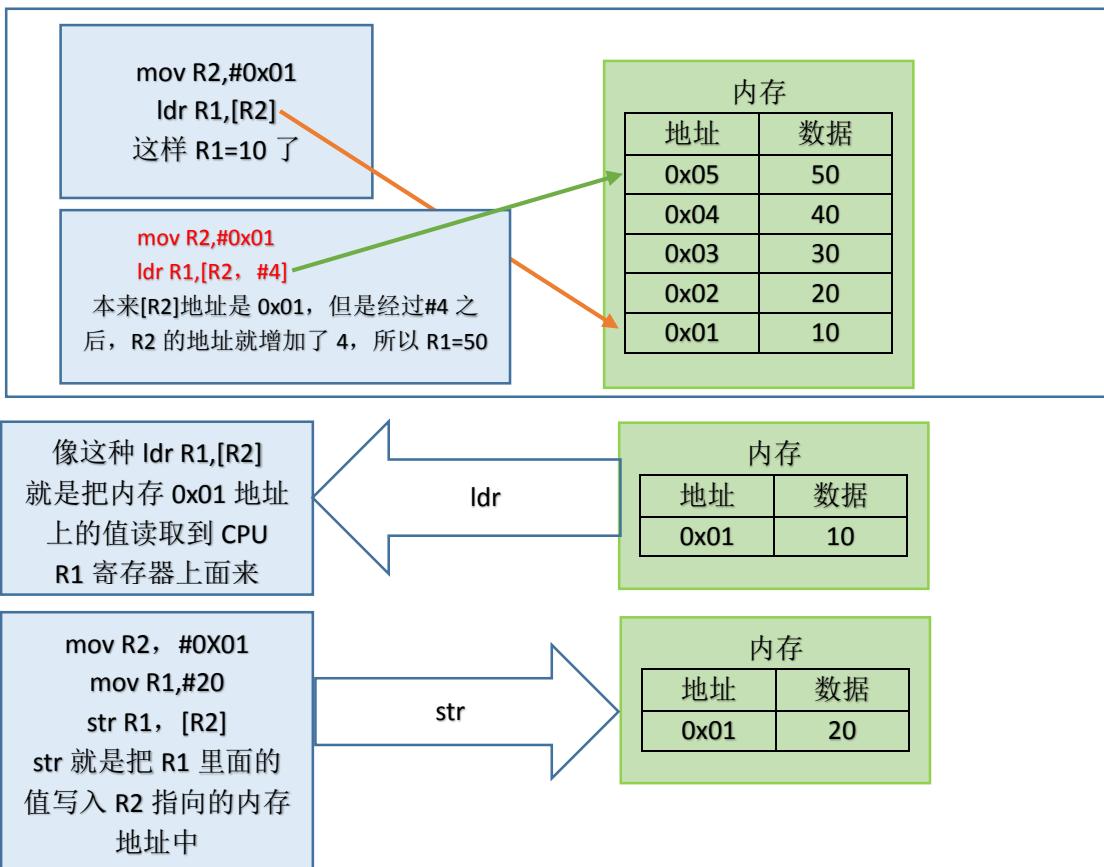
ARM 有 37 个寄存器，其中 R0~R12 是可以用来当变量使用的。



如果我们要给变量赋值，就直接给 R0 赋值，或者 R1。汇编指令只能操作这 37 个寄存器，所以 ARM 用汇编代码做计算的时候只能用 R0~R12 来做变量，不像 C 语言可以随便定义无数个变量。



ARM 汇编指令集: ldr, str



伪指令:

伪指令是给编译器看的，不会生成二进制机器码。指令经过编译会生成二进制机器码。

伪指令一般都是 . 点开头

.global //这个伪指令表示全局的意思

```
17 _xxxx:
18     mov R1,R2
19 bb:
20     mov R3,R4
```

在前面随便写名字然后冒号结束，意思是我不管程序执行到哪里了，如果我想执行 `mov R1,R2`，就执行指定 `_xxxx:` 程序就能找到这个 `_xxxx` 地址下的指令

```
2 flag:
3     b flag
```

类似 C 语言死循环

While(1)

b 就是跳转命令，跳转到指定的伪指令哪里去，执行它下面的指令

裸机下载汇编程序执行过程

平时我们编译 c 程序就是直接用 `gcc -o hello hello.c` 然后执行 `./hello` (这种就是编译和链接一步完成)

```
/*led.s ass noOS*/\n\n_start:\n    ldr r0, =0x11111111\n    ldr r1, =0xe0200240\n    str r0,[r1]\n\n    ldr r0, =0x0\n    ldr r1, =0xe0200244\n    str r0,[r1]\n\nflag:\n    b flag
```

首先将汇编程序用 `gcc` 编译成.o 中间文件，编译成中间文件用 `gcc -c` 格式

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-linux-gcc-4.6.2 -c led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.o led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf led.o  
arm-none-linux-gnueabi-ld: warning: cannot find entry symbol _start; defaulting to 00000000  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.elf led.o led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-none-linux-gnueabi-objcopy -O binary led.elf led.bin  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.bin led.elf led.o led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# █
```

这个 elf 格式的文件可以用 tftp 下载在板子上的操作系统上执行，但是我们板子是裸机，没有操作系统，所以要用烧录的方式，那么烧录软件只认识 bin 格式的文件

用 objcopy 转换一下 led.elf 格式为 bin 格式

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-none-linux-gnueabi-objdump -D led.elf >led_elf.dis
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls
led.bin led.elf led_elf.dis led.o leds

Disassembly of section .text:
00000000 <start>:
 0: e59f0014        ldr    r0, [pc, #20] ; 1c <flag+0x4>
 4: e59f1014        ldr    r1, [pc, #20] ; 20 <flag+0x8>
 8: e5810000        str    r0, [r1]
 c: e3a00000        mov    r0, #0 ; 0x0
10: e59f100c        ldr    r1, [pc, #12] ; 24 <flag+0xc>
14: e5810000        str    r0, [r1]

00000018 <flag>:
18: eaffffff        b     18 <flag>
1c: 11111111        .word 0x11111111
20: e0200240        .word 0xe0200240
24: e0200244        .word 0xe0200244
Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
 0: 00001541        andeq r1, r0, r1, asr #10
 4: 61656100        cmvns r5, r0, lsl #2
 8: 01006962        tsteq r0, r2, ror #18
 c: 0000000b        andeq r0, r0, fp
10: 01080106        tsteq r8, r6, lsl #2
14: Address 0x00000014 is out of bounds.
```

在汇编里面标号就是一个地址

这就是每条汇编语句
对应生成的机械码

这里就是汇编的 `ldr r0, =0x11111111`
因为这个`=0x11111111`是伪指令，所以它是存放在 `1c` 这里的。`[pc, #20]` 这里的 `#20= 20+8=28` 将 `28` 转换成 `16` 进制就是 `1c`。所以 `pc` 就去 `1c` 这里取 `11111111` 赋值给 `r0`

我们前面用数字表示寄存器地址太难记住了

```
/*led.s ass noOS*/\n\n_start:\n    ldr r0, =0x11111111\n    ldr r1, =0xe0200240\n    str r0,[r1]\n\n    ldr r0, =0x0\n    ldr r1, =0xe0200244\n    str r0,[r1]\n\nflag:\n    b flag\n\n    #.global _start\n    _start:\n        ldr r0, =0x11111111\n        ldr r1, =0xe0200240\n        str r0,[r1]\n\n        ldr r0, =0x0\n        ldr r1, =0xe0200244\n        str r0,[r1]\n        b .\n\n        /*汇编语言 .点 代表当前指令的地址*/
```

```
root@ubuntu:/home/xiang/SSPV210/noOS_driver/s_driver# make  
arm-none-linux-gnueabi-gcc-4.6.2 -c led.s  
arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf led.o  
arm-none-linux-gnueabi-ld: warning: cannot find entry symbol _start; defaulting to 00000000  
arm-none-linux-gnueabi-objcopy -O binary led.elf led.bin
```

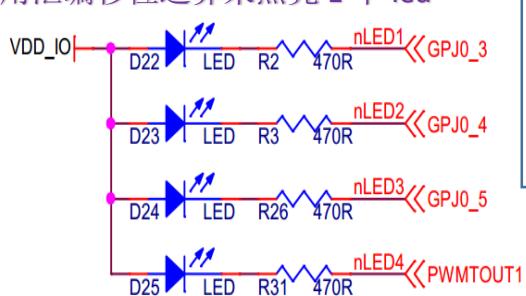
这个问题是程序连接不到 start，它找不到 start

```
.global _start
_start:
    ldr r0, =0x11111111
    ldr r1, =0xe0200240
    str r0, [r1]

    ldr r0, =0x0
    ldr r1, =0xe0200244
    str r0, [r1]
b .      /*汇编语言 .点 代表当前指令的地址*/
```

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# make  
arm-none-linux-gnueabi-gcc-4.6.2 -c led.s  
arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf led.o  
arm-none-linux-gnueabi-objcopy -O binary led.elf led.bin
```

你看这下编译就没有问题了。.global 就是把_start 改成外部这样其它文件就能看到_start 了
用汇编移位运算来点亮 1 个 led



7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1

GPJ0DAT

我们点亮 GPJ0_4

```

2 /*led.s ass noOS*/
3
4
5 .global _start
6 _start:
7     ldr r0, =0x11111111
8     ldr r1, =0xe0200240 /*#GPJ0CON*/
9     str r0, [r1]
10
11    ldr r0, =0xef /*GPJ0DAT 11101111*/
12    ldr r1, =0xe0200244 /*GPJ0DAT*/
13    str r0, [r1]
14    b . /*汇编语言 .点 代表当前指令的地址*/
15
16
17

```

这样就点亮了一个 GPJ0_4 的 LED，但是我们用的不是移位运算

```

.global _start
_start:
    ldr r0, =0x11111111
    ldr r1, =0xe0200240 /*#GPJ0CON*/
    str r0, [r1]
    ldr r0, =(1<<4) /*GPJ0DAT 00010000*/
    ldr r1, =0xe0200244 /*GPJ0DAT*/
    str r0, [r1]
    b . /*汇编语言 .点 代表当前指令的地址*/

```

GPJ0_3, GPJ0_5 等亮，GPJ0_4 灯灭，为什么呢？(1<<4)就是把 GPJ0_4 设置为高电平，所以用移位可以确定某个 io 口状态。移位运算和赋值运算用哪个？根据实际情况而定
给 LED 汇编程序加延时

cmp 指令：语法格式 cmp r2, r3

cmp 指令就是($r2 - r3 = 0$)那么 Z=1，如果($r2 - r3 \neq 0$)那么 Z=0，

sub 指令：语法格式 sub ax, 9

sub 指令就是，给 ax 减 9，之后的结果赋值给 ax

bne 指令：前面的结果不等于 0($r2 - r3 \neq 0$)就跳转到指定位置

从 b 行跳转
到 flag 行，继
续向下执行

```

        str r0,[r1]
        ldr r0, =0
        ldr r1, =0
        str r0,[r1]
        b flag

```

程序跑到 b 指令，执行
跳转，跳转到哪里呢？
flag 就是跳转的位置

bne 就是比 b 多了个条件判断功能，如果上
面 $cmp r2 - r3 \neq 0$ ，那么执行 bne 跳
转，跳转到 delay_loop 行执行。如果 $r2 - r3 = 0$
那么不跳转继续向下执行

```

delay_loop:
    sub r2, r2, #1
    cmp r2, r3
    bne delay_loop

```

beq 指令:

该指令和 bne 是反过来的, 如果 cmp r0, r1 (r0-r1=0), 那么就跳转指定地址

```
5 .global _start
6 _start:
7     ldr r0, =0x11111111
8     ldr r1, =0xe0200240 /*#GPJ0CON*/
9     str r0, [r1]
0 strr:
1     ldr r0, =((1<<4) | (1<<3) | (1<<5)) /*GPJ0DAT 00111000*/
2     ldr r1, =0xe0200244 /*GPJ0DAT*/
3     str r0, [r1]
4     bl delay /*调用我下面写好的延时函数*/
5
6     ldr r0, =((0<<4) | (0<<3) | (0<<5)) /*GPJ0DAT 00000000*/
7     ldr r1, =0xe0200244 /*GPJ0DAT*/
8     str r0, [r1]
9     bl delay /*调用我下面写好的延时函数*/
0     b strr /*跳转到strr类似while死循环*/
1
2
3 /*延时程序 函数名叫delay*/
4 delay:
5     ldr r2, =9000000 /*给R2变量赋值9000000*/
6     ldr r3, =0 /*给R3变量赋值0*/
7 delay_loop:
8     sub r2, r2, #1 /*r2 = r2-1 这个#1就是r2要减多少值*/
9     cmp r2, r3 /*如果r2等于r3, 下一句eq就会成立*/
0     bne delay_loop
1     mov pc, lr
```

bl 是跳转指令,意思是下面的指令不执行,跳转到指定的程序段执行,这里的指定程序段就是 delay,也就是 C 语言的主函数跳转到子函数执行

Pc 跳转到 lr 存放的地址执行,
lr 是自动存储前面程序断点的地址

下面我们给 mov pc, lr 做个比喻

0x08	ldr r0
0x07	ldr r1
0x06	bl delay
0x05	Str:r0, [r1]
0x04	
0x03	
0x02	delay:
0x01	
地址	

lr = 0x06

汇编程序是按照地址顺序来执行的,但是这里被 bl 打断了,要求跳过下面的 0x05 去先执行 0x02 地址上面的程序,那么 0x02 地址上面的程序执行完了,我就要直接跳出汇编程序结束

0x08	ldr r0
0x07	ldr r1
0x06	bl delay
0x05	Str:r0, [r1]
0x04	
0x03	
0x02	delay:
0x01	mov pc, lr
地址	

lr = 0x06

因为指令是根据 pc 指针指向的地址挨着顺序一个地址一个地址的执行,我这里加 mov pc, lr 就是想在 0x02 地址上 delay 程序执行完了,将 pc 指针指向 lr 存放的地址,也就是前面的 0x05 地址,然后接着执行 0x05 地址上的程序,这样汇编就不会像左边那样没有 mov pc, lr 指令从新指向新的地址,而导致程序结束。

这样 led 灯循环延时闪烁实验成功

汇编语言关闭看门狗

因为 S5PV210cpu 上电是启动了看门狗的(WTGD), 所以我们要先关掉看门狗

3.4.1.1 Watchdog Timer Control Register (WTCON, R/W, Address = 0xE270 0000)

The WTCON register allows you to enable/ disable the watchdog timer, select the clock signal from four different sources, enable/ disable interrupts, and enable/ disable the watchdog timer output.

The Watchdog timer is used to restart the S5PV210 to recover from mal-function; if controller restart is not desired, the Watchdog timer should be disabled.

If you want to use the normal timer provided by the Watchdog timer, enable the interrupt and disable the Watchdog timer.

WTCON	Bit	Description		Initial State
section 07_timer		Reserved	[7:6]	Reserved. These two bits must be 00 in normal operation.
1 PULSE WIDTH MODULATION TIMER		Watchdog timer	[5]	Enables or disables Watchdog timer bit. 0 = Disables 1 = Enables
2 SYSTEM TIMER		Clock select	[4:3]	Determines the clock division factor. 00 = 1K
3 WATCHDOG TIMER				
3.1. OVERVIEW OF WATCHDOG TIMER				
3.2. KEY FEATURES OF WATCHDOG TIMER				
3.3. FUNCTIONAL DESCRIPTION OF WATCHDOG TIMER				
3.4. REGISTER DESCRIPTION				
3.4.1. REGISTER MAP				
3.4.1.1. Watchdog Timer Control Register (WTCON, R/W, Add:				
3.4.1.2. Watchdog Timer Data Register (WTDAT, R/W, Addr:				

```
.global _start
_start:
    /*第1步关看门狗*/
    ldr r0, =0xe2700000 /*WTCON*/
    ldr r1, =0x0           /*将 0 写入 r1 变量*/
    str r1, [r0]           /*将 r1 变量里面的 0 全部写入看门狗控制寄存器地址*/

    ldr r0, =0x11111111
    ldr r1, =0xe0200240 /*#GPJ0CON*/
    str r0, [r1]

strr:
    ldr r0, =((1<<4) | (1<<3) | (1<<5)) /*GPJ0DAT 00111000*/
    ldr r1, =0xe0200244 /*GPJ0DAT*/
    str r0, [r1]
    bl delay /*调用我下面写好的延时函数*/

    ldr r0, =((0<<4) | (0<<3) | (0<<5)) /*GPJ0DAT 00000000*/
    ldr r1, =0xe0200244 /*GPJ0DAT*/
    str r0, [r1]
    bl delay /*调用我下面写好的延时函数*/
    b strr     /*跳转到strr类似while死循环*/

/*延时程序 函数名叫delay*/
delay:
    ldr r2, =9000000 /*给R2变量赋值9000000*/
    ldr r3, =0 /*给R3变量赋值0*/
delay_loop:
    sub r2, r2, #1 /*r2 = r2-1 这个#1就是r2要减多少值*/
    cmp r2, r3      /*如果r2等于r3，下一句eq就会成立*/
    bne delay_loop
    mov pc, lr
```

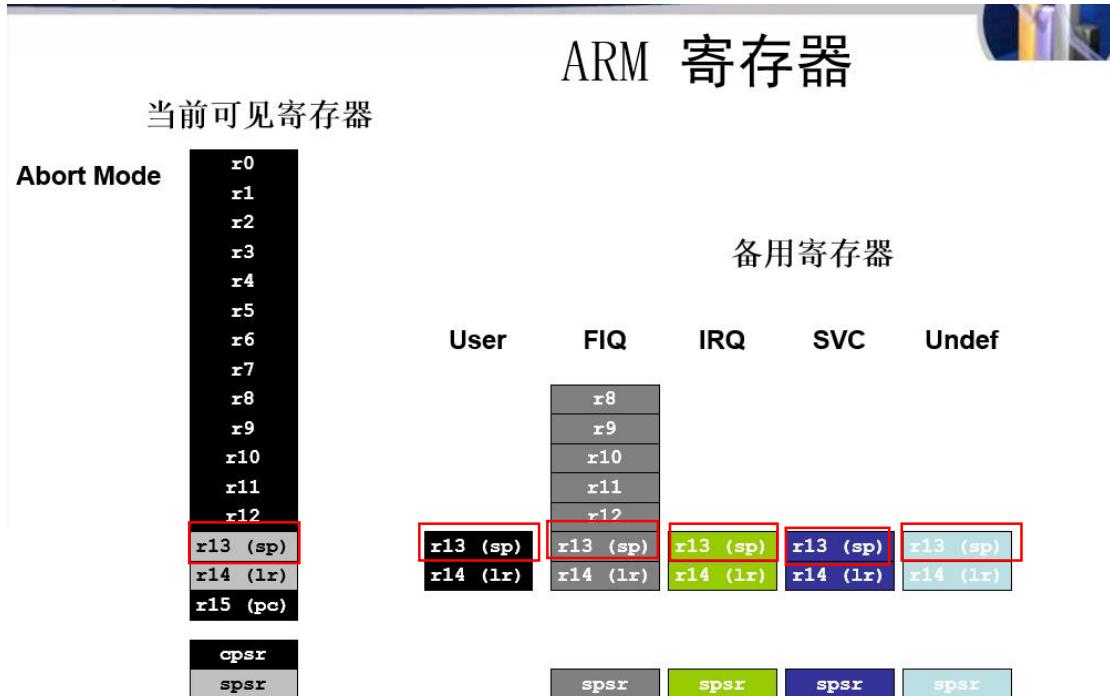
关闭看门狗后 led 闪烁实验成功

汇编启动代码启动 C 程序

要运行 C 语言，必须先用汇编设置栈，因为 C 语言的局部变量都要栈来保存。

像单片机，STM32 之流用的是 IDE，IDE 帮我们做好栈空间了。但是我们 ARM linux 要自己去设置 IDE 实现的栈。

我们用 sp 指令去设置栈



但是为什么有这么多 sp，一个 sp 不行吗？如果一个 sp，那么整个程序就只有一个栈。

一个栈里面有操作系统程序的变量，QQ 应用程序的变量，微信应用程序的变量，如果其中一个(QQ)程序变量太多导致内存溢出问题，其他程序就跟着遭殃。

所以必须做得和我们电脑一样，qq 报错了，但是其他程序还能正常运行。这就需要多个栈空间。User 模式下一个栈，FIQ 模式下一个栈，IRQ 模式下一个栈，SVC 模式下一个栈..

```
.global _start
_start:
    /*第1步关看门狗*/
    ldr r0, =0xe2700000 /*WTCON*/
    ldr r1, =0x0
    str r1, [r0]
    ldr sp, =0xd0037d80 /*设置svc栈指针开始地址*/
    bl led_blink
    b .
```

系统在复位之后是处于 svc 模式的，所以这里执行 sp 是设置 svc 模式下的栈

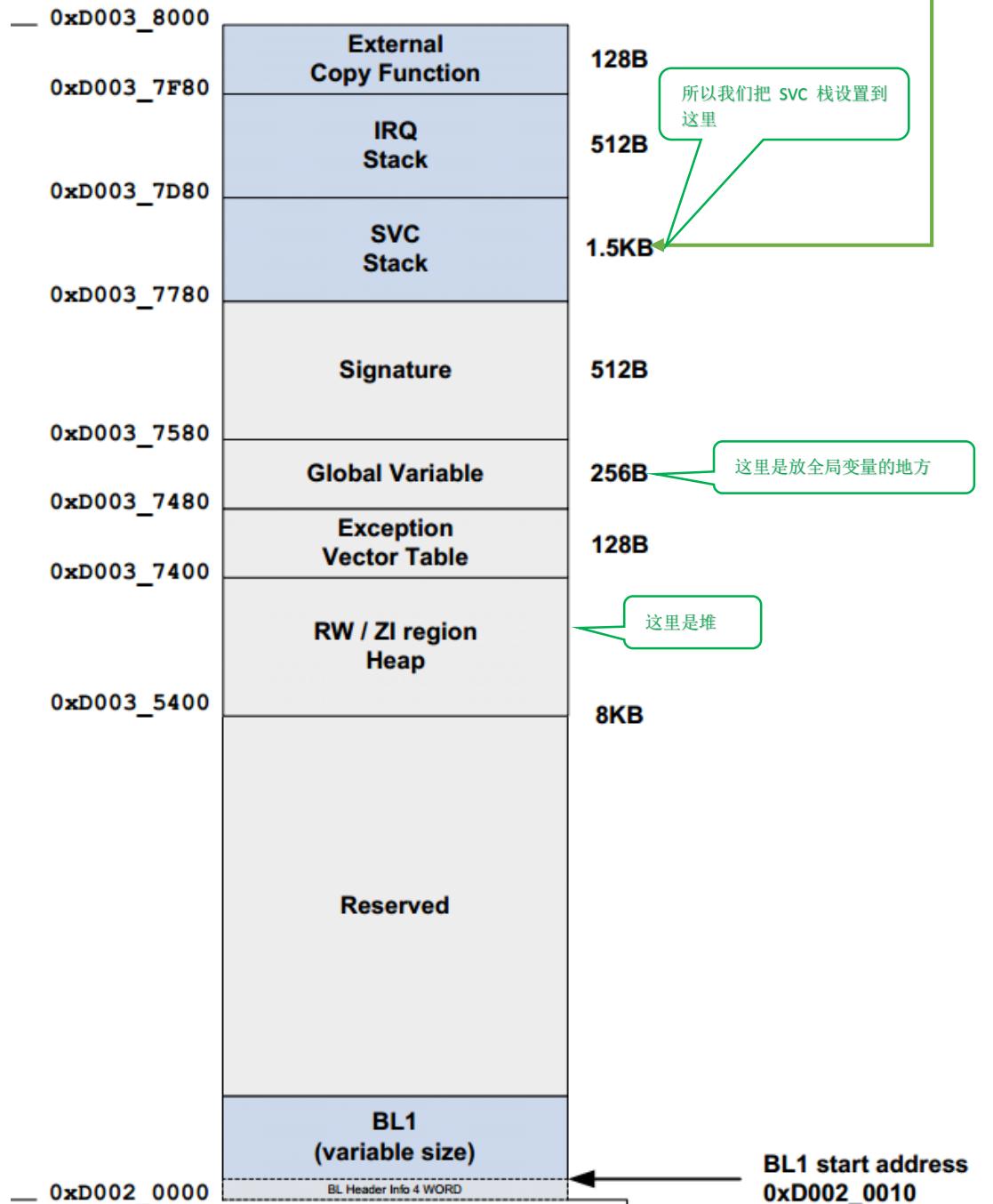
这是跳转去执行C文件里面的某个函数

这里死循环要开

汇编程序启动代码来设置栈。

这个栈地址只能找内部的 SRAM 来设置，因为外部的 DDR 还没有初始化，内部的 SRAM 是可以直接找地址随意读写的。

但是三星已经规定了 SRAM 空间的分配区域，看下面图。



```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver/start# ls
led.c Makefile start.s
```

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver/start#
```

一个 start.s 文件，上面已经讲过了，用来启动 C 文件 led_blink 函数的。

一个 led.c 文件，里面有 led_blink 函数

一个 Makefile 编译文件。

```

#define GPJ0CON 0xe0200240
#define GPJ0DAT 0xe0200244

void delay(unsigned int num);

void led_blink(void)
{
    unsigned int *gpio_con = (unsigned int *)GPJ0CON;
    unsigned int *gpio_dat = (unsigned int *)GPJ0DAT;
    *gpio_con = 0x11111111; //设置gpio管教功能 , GPJ0_3 , GPJ0_4 , GPJ0_5

    while(1)
    {
        *gpio_dat = ((0<<3)|(0<<4)|(0<<5));
        delay(500000);
        *gpio_dat = ((1<<3)|(1<<4)|(1<<5));
        delay(500000);
    }
}

void delay(volatile unsigned int num)
{
    while(num--);
}

```

```

BULIDD=led
ELF=start.o led.o

$(BULIDD).bin:$(BULIDD).elf
    $(shell arm-none-linux-gnueabi-objcopy -O binary $^ $@)

$(BULIDD).elf : $(ELF)
    arm-none-linux-gnueabi-ld -Ttext 0x0 -o $@ $^

%.o : %.s
    arm-none-linux-gnueabi-gcc-4.6.2 -o $@ $^ -c -nostdlib
%.o : %.c
    arm-none-linux-gnueabi-gcc-4.6.2 -o $@ $^ -c -nostdlib

clean:
    rm -rf *.bin *.elf *.o

```

```

root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver/start# make
arm-none-linux-gnueabi-gcc-4.6.2 -o start.o start.s -c -nostdlib
arm-none-linux-gnueabi-gcc-4.6.2 -o led.o led.c -c -nostdlib
arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf start.o led.o
arm-none-linux-gnueabi-ld: Warning: led.o: Unknown EABI object attribute 44
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver/start#

```

编译成功，用 dnw 烧写 bin 文件进开发板运行就可以了。

裸机下 .s 汇编文件和.c C 文件混合编译成二进制文件过程

```
BULIDD=led
ELF=start.o led.o

$(BULIDD).bin:$(BULIDD).elf
    $(shell arm-none-linux-gnueabi-objcopy -O binary $^ $@)

$(BULIDD).elf : $(ELF)
    arm-none-linux-gnueabi-ld -Ttext 0x0 -o $@ $^

%.o : %.s
    arm-none-linux-gnueabi-gcc-4.6.2 -o $@ $^ -c -nostdlib

%.o : %.c
    arm-none-linux-gnueabi-gcc-4.6.2 -o $@ $^ -c -nostdlib

clean:
    rm -rf *.bin *.elf *.o
```

? arm-none-linux-gnueabi-ld

用 `ld` 链接将汇编 `s` 和 `c` 文件的中间件联合起来，`-Ttext` 指定运行地址，编译成最终可执行的 `elf` 可执行文件

1. **arm-none-linux-gnueabi-gcc**
这里用 gcc 交叉编译器将 s 文件编译成.o，将 c 文件编译成.o
.o 就是中间文件

.elf 格式就是可执行程序，你可以在 X86 上编译后试试。但是因为我们是要烧录进 ARM 开发板的，所以我要把 elf 文件做成二进制文件。才能烧录

-3 所以有了这一步 **arm-none-linux-gnueabi-objcopy**

程序烧录进开发板之后，先执行二进制文件里面的汇编内容

led.elf

[mo/xi.ape/](#)这就是我们编译好的文件，你可以反编译它

前面这段是汇编代码

这就是 elf 文件

这用就是跳转到 .c 文件的代码

代码重定位

C 语言代码在内存存放位置，分为代码段(.text)，数据段(.data)，.bss 段

代码段：放 C 语言函数

数据段：放初始化的全局变量，一定是全局变量初始化的不是 0

.bss 段：如果全局变量初始化为 0，就放在 bss 段

```
lode@lode-OptiPlex-5090:~/Desktop$ cat link.lds
SECTIONS
{
    . = 0xd0024000;

    .text : {
        start.o
        * (.text)
    }

    .data : {
        * (.data)
    }

    bss_start = .;
    .bss : {
        * (.bss)
    }

    bss_end = .;
}
```

这就是链接脚本文件

- . 点符号在链接脚本中代表当前位置
- = 号就是赋值，意思就是在当前位置，赋值了一个 0xd0024000 地址
- 代码段就放在.text 这个大括号下
- 这个意思就是我们写的 start.s 生成的 start.o 启动汇编程序放在最前面
- *(.text) 这个*号就是匹配所有的代码段，也就是其余 C 语言生成的.o 放在这里
- 数据段不关心代码前后，全局变量数据放在这里就是

这些代码都是有地址偏移的

```
lode@lode-OptiPlex-5090:~/Desktop$ cat link.lds
SECTIONS
{
    . = 0xd0024000;
    .text : {
        start.o
        * (.text)
    }
    .data : {
        * (.data)
    }
    bss_start = .;
    .bss : {
        * (.bss)
    }
    bss_end = .;
}
```

比如我的地址是从 0xd0024000 开始

经过 start.o, text 段, data 段之后，执行到了 bss_start，因为 start.o text data 这些段都是要占用地址空间的，比如占用了 1000 个地址

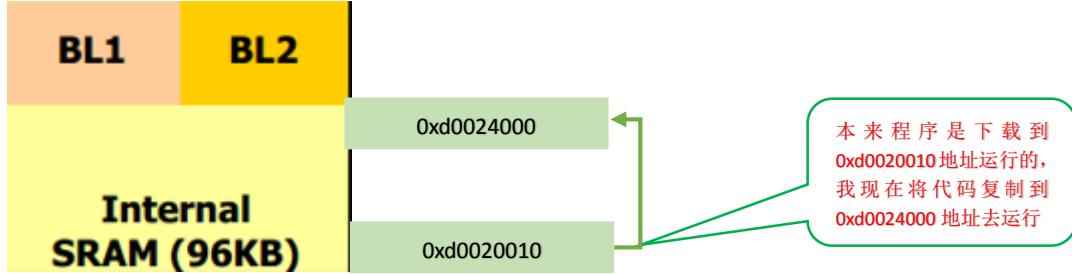
我们知道，. 点表示当前地址，那么这里的地址就是 0xd0025000.= 就是把 0xd0025000 赋值给 bss_start 变量

.bss 段也占用了 1000 个地址

那这里就是把 0xd0026000 地址赋值给 bss_end

bss_start 和 bss_end 的值可以让汇编或者 C 程序调用。

在 S5PV210 内部的 SRAM，将放在 0xd0020010 的代码重定位到 0xd0024000



```

#define WTC0N      0xE2700000
#define SVC_STACK  0xd0037d80

.global _start          // 把_start链接属性改为外部，这样其他文件就可以看见_start了
_start:
    // 第1步：关看门狗（向WTC0N的bit5写入0即可）
    ldr r0, =WTC0N
    ldr r1, =0x0
    str r1, [r0]

    // 第2步：设置SVC栈
    ldr sp, =SVC_STACK

    // 第3步：开/关icache
    mrc p15, 0, r0, c1, c0, 0;           // 读出cp15的c1到r0中
    //bic r0, r0, #(1<<12)           // bit12 置0 关icache
    orr r0, r0, #(1<<12)             // bit12 置1 开icache
    mcr p15, 0, r0, c1, c0, 0;

    // 第4步：重定位
    // adr指令用于加载_start当前运行地址
    adr r0, _start                  // adr加载时就叫短加载
    // ldr指令用于加载_start的链接地址:0xd0024000
    ldr r1, =_start // ldr加载时如果目标寄存器是pc就叫长跳转，如果目标寄存器是r1等就叫长加载
    // bss段的起始地址
    ldr r2, =bss_start // 就是我们重定位代码的结束地址，重定位只需重定位代码段和数据段即可
    cmp r0, r1           // 比较_start的运行时地址和链接地址是否相等
    beq clean_bss        // 如果相等说明不需要重定位，所以跳过copy_loop，直接到clean_bss
    //如果不相等说明需要重定位，那么直接执行下面的copy_loop进行重定位
    // 重定位完成后继续执行clean_bss。
    ...

    clean_bss:
        ldr r0, =bss_start
        ldr r1, =bss_end
        cmp r0, r1           // 如果r0等于r1，说明bss段为空，直接下去
        beq run_on_dram       // 清除bss完之后的地址
        mov r2, #0

    // 第4步：重定位
    // adr指令用于加载_start当前运行地址
    adr r0, _start                  // adr加载时就叫短加载
    // ldr指令用于加载_start的链接地址:0xd0024000
    ldr r1, =_start
    // bss段的起始地址
    ldr r2, =bss_start
    cmp r0, r1           // 如果r0, r1地址不相等，那么就不执行beq跳转，继续执行下面的copy loop段
    beq clean_bss

```

这部分就是关闭看门狗，设置栈，打开高速 cache 缓存

adr 指令是加载运行地址给 r0 寄存器变量。
ldr 是加载链接地址给 r1 寄存器变量

我们来判断 r0 的运行地址和 r1 的链接地址是不是同一个地址，如果是同一个地址那么就不用执行从定位，直接跳转到 clean_bss 位置

.data : { * (.data) }
bss_start = .; bss_end = . 把链接脚本的 bss_start 的值 0xd0025000 赋值给 r2

```
// 用汇编来实现的一个while循环
copy_loop:
    ldr r3, [r0], #4      // 源
    str r3, [r1], #4      // 目的  这两句代码就完成了4个字节内容的拷贝
    cmp r1, r2            // r1和r2都是用ldr加载的，都是链接地址，所以r1不断+4总能等于r2
    bne copy_loop

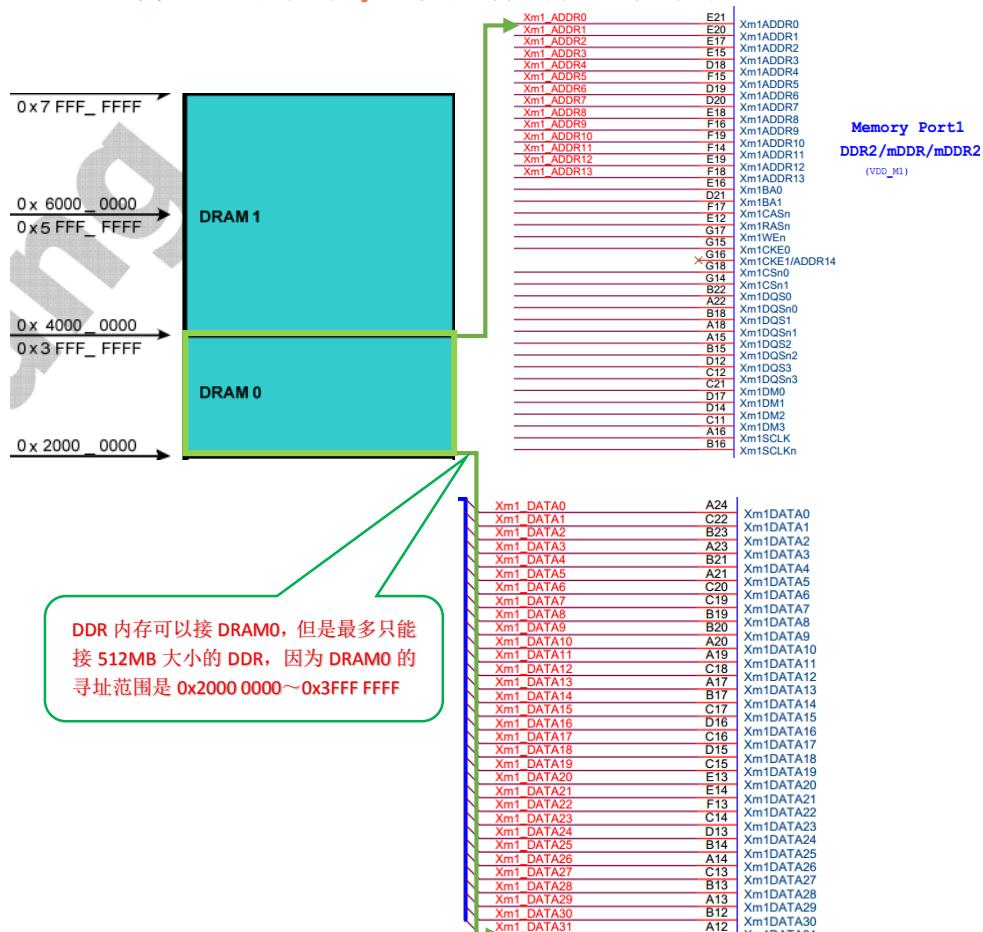
// 清bss段，其实就是在链接地址处把bss段全部清零
clean_bss:
    ldr r0, =bss_start
    ldr r1, =bss_end
    cmp r0, r1            // 如果r0等于r1，说明bss段为空，直接下去
    beq run_on_dram        // 清除bss完之后的地址
    mov r2, #0
clear_loop:
    str r2, [r0], #4        // 先将r2中的值放入r0所指向的内存地址（r0中的值作为内存地址），
    cmp r0, r1            // 然后r0 = r0 + 4
    bne clear_loop

run_on_dram:
    // 长跳转到led_blink开始第二阶段
    ldr pc, =led_blink      // ldr指令实现长跳转

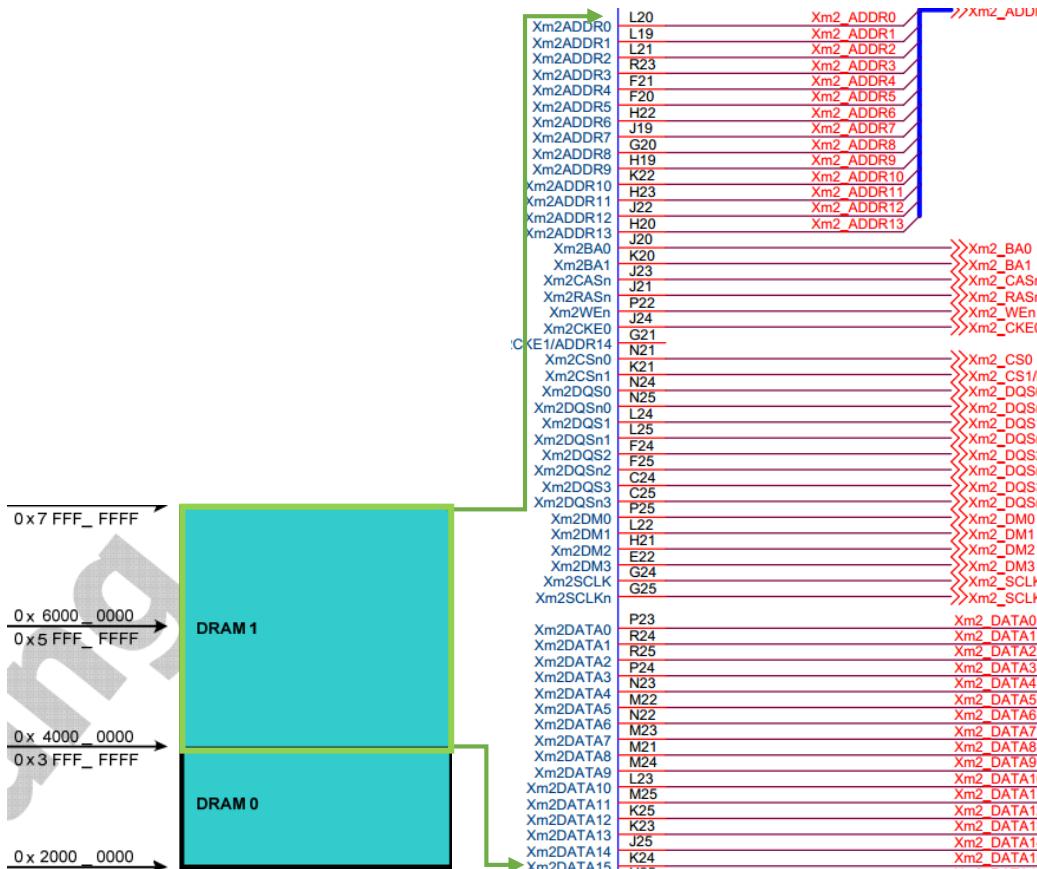
    // 从这里之后就可以开始调用C程序了
    //bl led_blink           // bl指令实现短跳转

// 汇编最后的这个死循环不能丢
b.
```

SDRAM 介绍及初始化,必须要有前面重定位知识



DRAM0 最大接 512MB 的 DDR，最小可以接 64M 的 DDR，所以 DRAM0 支持 64M、128M、256M、512M



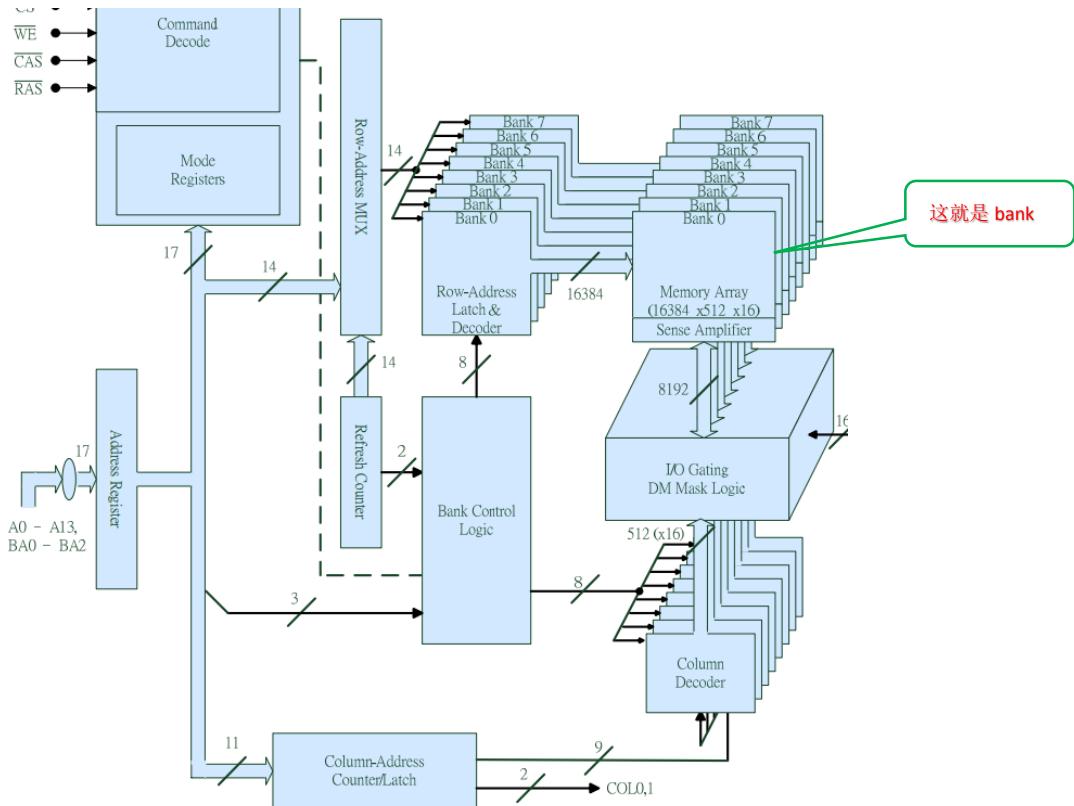
DRAM1 内存寻址范围是 0x4000_0000 ~ 0x7FFFFFFF 可以接 1024M (1Gb) 的 DDR 内存
所以我们 S5PV210 型号的 CPU 最大支持 1.5G 的内存，如果要用更大的内存只要换 CPU

DDR 内部分布局

Density	Banks	Part Number	Package & Power, Temp. (-C/-L) & Speed	Org.
128Mb O-die	4Banks	K4T28163QO	HCF8/E7/F7/E6	8M x 16
256Mb I-die	4Banks	K4T56163QI	Z*1C(L)E7/F7/E6/D5/CC	16M x 16
256Mb N-die	4Banks	K4T56163QN	HCF8/E7/F7/E6	16M x 16
512Mb G-die	4Banks	K4T51083QG	HC(L)F8/E7/F7/E6	64M x 8
		K4T51163QG	HC(L)F8/E7/F7/E6	32M x 16
512Mb I-die	4Banks	K4T51043QI	HC(L)E7/F7/E6	128M x 4
		K4T51083QI	HC(L)E7/F7/E6	64M x 8
		K4T51163QI	HC(L)F8/E7/F7/E6	32M x 16
1Gb E-die	8Banks	K4T1G084QE	HC(L)F8/E7/F7/E6	128M x 8
		K4T1G164QE	HC(L)F8/E7/F7/E6	64M x 16

我使用的是这个 DDR2 DDR 总大小是 1Gb (实际字节大小就是 128MB)
所以我这个 DDR 只有 128M 大小

8Banks, 意思就是这个 DDR 有 8 个 Bank(块), 每块是 128Mb 位, 所以 8 个加起来是 1024Mb 位



汇编程序初始化 DDR

v210.h sram_init.S link.lds

```

SECTIONS
{
    . = 0x20000000;
    .text : {
        start.o
        sram_init.o
        * (.text)
    }

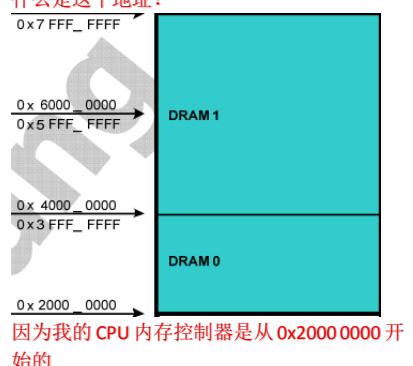
    .data : {
        * (.data)
    }

    bss_start = .;
    .bss : {
        * (.bss)
    }

    bss_end = .;
}

```

链接脚本的重定位地址改成 0x20000000, 为什么是这个地址?



因为我的 CPU 内存控制器是从 0x2000 0000 开始的

下面是启动过程分析

210.h sram_init.S link.lds start.S

```

#define WTCN 0xE2700000
#define SVC_STACK 0xd0037d80

.global _start // 把_start链接属性改为外部，这样其他文件就可以看见_start了
_start:
    // 第1步：关看门狗（向WTCN的bit5写入0即可）
    ldr r0, =WTCN
    ldr r1, =0x0
    str r1, [r0] 1.这里没变，先关闭看门狗

    // 第2步：设置SVC栈
    ldr sp, =SVC_STACK 2.这里没变，设置栈

    // 第3步：开/关icache
    mrc p15, 0, r0, c1, c0, 0; // 读出cp15的c1到r0中
    //bic r0, r0, #(1<<12) // bit12 置0 关icache
    orr r0, r0, #(1<<12) // bit12 置1 开icache 3.这里没变，开启 cache

    mcr p15, 0, r0, c1, c0, 0;

    // 第4步：初始化ddr
    bl sram_asm_init 4.这里变了，先初始化 DDR，bl 就是调用 DDR 的汇编函数，这个汇编函数在哪里呢？

    // 第5步：重定位
    // adr指令用于加载_start当前运行地址
    adr r0, _start // adr加载时就叫短加载 5.然后再重新定位
    // ldr指令用于加载_start的链接地址:0xd0024000
    ldr r1, =_start // ldr加载时如果目标寄存器是pc就叫长跳转，如果目标寄存器是r1等就叫长加载
    // bss段的起始地址
    ldr r2, =bss_start // 就是我们重定位代码的结束地址，重定位只需重定位代码段和数据段即可
    cmp r0, r1 // 比较_start的运行时地址和链接地址是否相等
    beq clean_bss // 如果相等说明不需要重定位，所以跳过copy_loop，直接到clean_bss
    // 如果不相等说明需要重定位，那么直接执行下面的copy_loop进行重定位
    // 重定位完成后继续执行copy_loop

```

s5pv210.h sram_init.S link.lds

```

.global sram_asm_init
sram_asm_init:
    ldr r0, =0xf1e00000
    ldr r1, =0x0
    str r1, [r0, #0x0]

```

这个 sram_asm_init 汇编函数在 sram_init.s 文件下，这里的.global 就是指定该函数可以被其他汇编文件调用

s5pv210.h sram_init.S link.lds

```

#include "s5pv210.h"

#if 1
#define DMCO_MEMCONTROL
#define DMCO_MEMCONFIG 0

```

而且 sram_init.s 文件里面还包含了 s5pv210.h

s5pv210.h sram_init.S link.lds

```

/* S5PC110 device base addresses */
#define ELFIN_DMA_BASE 0xE0900000
#define ELFIN_LCD_BASE 0xF8000000
#define ELFIN_USB_HOST_BASE 0xEC200000
#define ELFIN_I2C_BASE 0xE1800000
#define ELFIN_I2S_BASE 0xE2100000
#define ELFIN_ADC_BASE 0xE1700000
#define ELFIN_SPI_BASE 0xE1300000

```

S5pv210.h 就是定义寄存器地址的

现在我们来具体分析 `sram_init.s`

```
.global sram_asm_init

sram_asm_init:
    ldr r0, =0xf1e00000
    ldr r1, =0x0
    str r1, [r0, #0x0]

    /* DMC0 Drive Strength (Setting 2X) */
    ldr r0, =ELFIN_GPIO_BASE
    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_0DRV_SR_OFFSET] // 寄存器中对应0b10，就是2X

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_1DRV_SR_OFFSET]

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_2DRV_SR_OFFSET]

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_3DRV_SR_OFFSET]

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_4DRV_SR_OFFSET]

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_5DRV_SR_OFFSET]

    ldr r1, =0x0000AAAA
    str r1, [r0, #MP1_6DRV_SR_OFFSET]
```

MP1_0DRV 0xE020_03CC

芯片要求设置成 2x

MP1_0DRV	Bit	Description	Initial State
MP1_0DRV[n]	[2n+1:2n] n=0~7	00 = 1x 10 = 2x 01 = 3x 11 = 4x	0xAAAA

所以我设置寄存器为 0x0000 AAAA

```
/* DMC1 Drive Strength (Setting 2X) */

ldr r0, =ELFIN_GPIO_BASE

ldr r1, =0x0000AAAA
str r1, [r0, #MP2_0DRV_SR_OFFSET]

ldr r1, =0x0000AAAA
str r1, [r0, #MP2_1DRV_SR_OFFSET]
str r1, [r0, #MP2_2DRV_SR_OFFSET]
str r1, [r0, #MP2_3DRV_SR_OFFSET]
str r1, [r0, #MP2_4DRV_SR_OFFSET]
str r1, [r0, #MP2_5DRV_SR_OFFSET]
str r1, [r0, #MP2_6DRV_SR_OFFSET]
```

DMC0 就是 DRAM0, DMC1 就是 DRAM1

这些都是设置 DRAM1 引脚的驱动强度，和 DRAM0 操作一样

```

/* DMC0 initialization at single Type*/
ldr r0, =APB_DMC_0_BASE
ldr r1, =0x00101000          @PhyControl0 DLL parameter setting, manual 0x00101000
str r1, [r0, #DMC_PHYCONTROL0]

ldr r1, =0x00000086          @PhyControl1 DLL parameter setting, LPDDR/LPDDR2 Case
str r1, [r0, #DMC_PHYCONTROL1]

ldr r1, =0x00101002          @PhyControl0 DLL on
str r1, [r0, #DMC_PHYCONTROL0]

ldr r1, =0x00101003          @PhyControl0 DLL start
str r1, [r0, #DMC_PHYCONTROL0]

find_lock_val:
    ldr r1, [r0, #DMC_PHYSTATUS]      @Load Phystatus register value
    and r2, r1, #0x7
    cmp r2, #0x7                      @Loop until DLL is locked
    bne find_lock_val

    and r1, #0x3fc0
    mov r2, r1, LSL #18
    orr r2, r2, #0x100000
    orr r2, r2, #0x1000

    orr r1, r2, #0x3
    str r1, [r0, #DMC_PHYCONTROL0]

```

中间经过了一大堆汇编 DDR 初始化，最后运行到 `mov pc, lr`

```

ldr r1, =DMC1_MEMCONTROL
str r1, [r0, #DMC_MEMCONTROL]
// 函数返回
mov pc, lr

```

这句话必须加，意思就是 `sdram_init.s` 汇编函数调用结束，返回到 `start.s` 继续执行下面的重定位函数

```

run_on_dram:
    // 长跳转到led_blink开始第二阶段
    ldr pc, =led_blink           // ldr指令实现长跳转

    // 从这里之后就可以开始调用C程序了
    // bl led_blink               // bl指令实现短跳转
    // 汇编最后的这个死循环不能丢
    b .

```

重定位完成后调到 c 文件去执行 led_link 主函数

所以 C 文件主函数的 `main` 名字我们是可以修改的

```

root@ubuntu:/home/xiang/S5PV210/noOS_driver/7.sdram_init# ls
led.c  link.lds  Makefile  mkv210_image.c  s5pv210.h  sdram_init.S  start.S  write2sd
root@ubuntu:/home/xiang/S5PV210/noOS_driver/7.sdram_init#

```

准备好你的初始化文件 `start.s`, `sdram_init.s` 和连接文件 `link.lds`, 然后准备跳转后的 c 文件 `led.c`

```

led.bin: start.o led.o sdram_init.o
        arm-linux-ld -Tlink.lds -o led.elf $^
        arm-linux-objcopy -O binary led.elf led.bin
        arm-linux-objdump -D led.elf > led_elf.dis
        gcc mkv210_image.c -o mkx210
        ./mkx210 led.bin 210.bin

%.o : %.S
|     arm-linux-gcc -o $@ $< -c -nostdlib

%.o : %.c
|     arm-linux-gcc -o $@ $< -c -nostdlib

clean:
|     rm *.o *.elf *.bin *.dis mkx210 -f

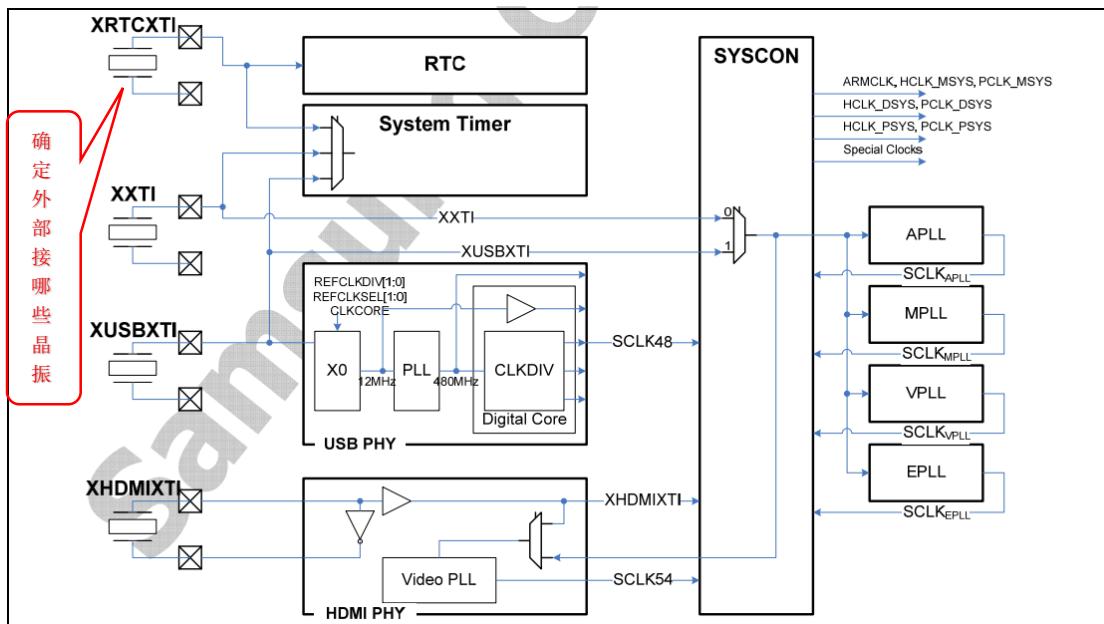
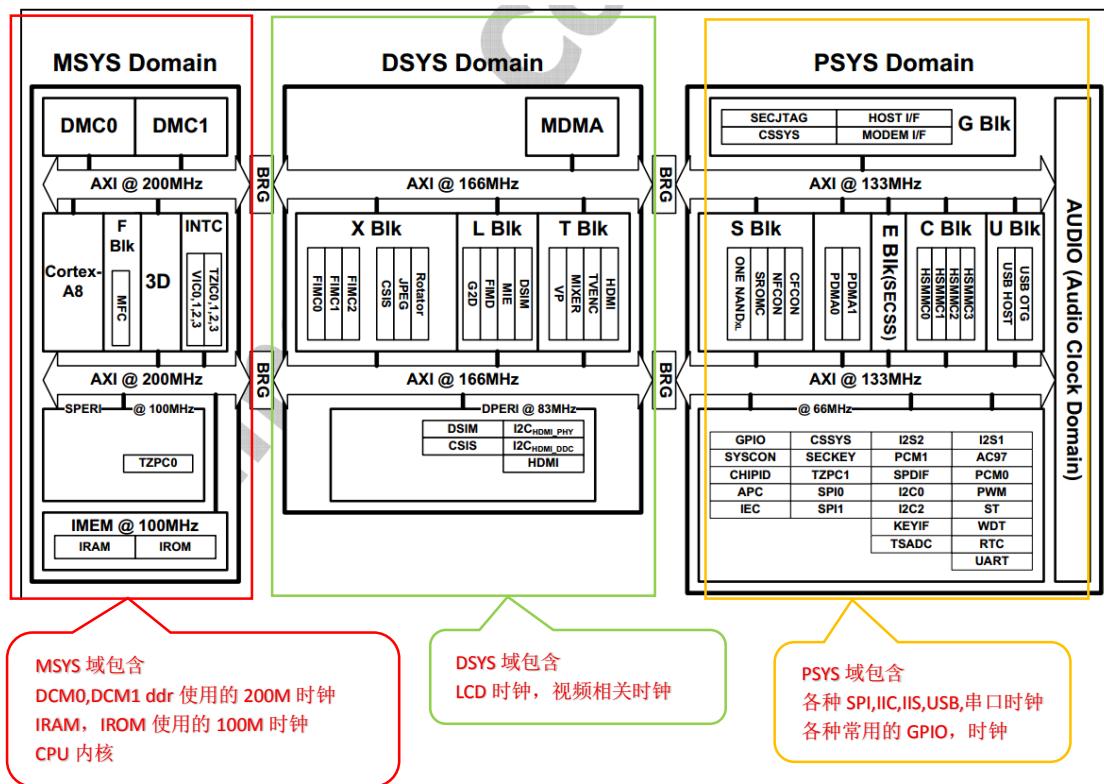
```

2.根据 link.lds 的链接地址，把程序放在指定的内存区域去执行，我这里的 link.lds 是 0x2000 0000

1.先把 s 文件编译成.o 把 c 文件编译成.o

生成 bin 文件下载进板子

S5PV210 时钟系统



时钟初始化

```
clock.S      start.S  X
/*
 * 文件名: led.s
 * 作者: 朱老师
 * 描述: 演示汇编开关icache
 */

#define WTCON      0xE2700000
#define SVC_STACK   0xd0037d80

.global _start          // 把_start链接属性改为外部, 这样其他文件就可以看见_start了
_start:
    // 第1步: 关看门狗(向WTCON的bit5写入0即可)
    ldr r0, =WTCON
    ldr r1, =0x0
    str r1, [r0]

    // 第2步: 初始化时钟
    bl clock_init           // 然后马上设置时钟, 跳转到时钟汇编函数

    // 第3步: 设置SVC栈
    ldr sp, =SVC_STACK

    // 第4步: 开/关icache
    mrc p15, 0, r0, c1, c0, 0;      // 读出cp15的c1到r0中
    //bic r0, r0, #(1<<12)        // bit12 置0 关icache
    orr r0, r0, #(1<<12)         // bit12 置1 开icache
    mcr p15, 0, r0, c1, c0, 0;

    // 从这里之后就可以开始调用C程序了
    bl led_blink             // led_blink是C语言实现的一个函数

    // 汇编最后的这个死循环不能丢
    b .

.global clock_init
clock_init:
    ldr r0, =ELFIN_CLOCK_POWER_BASE

    // 1 设置各种时钟开关, 暂时不使用PLL
    ldr r1, =0x0
    // 芯片手册P378 寄存器CLK_SRC: Select clock source 0 (Main)
    str r1, [r0, #CLK_SRC0_OFFSET]

    // 2 设置锁定时间, 使用默认值即可
    // 设置PLL后, 时钟从Fin提升到目标频率时, 需要一定的时间, 即锁定时间
    ldr r1, =0x0000FFFF
    str r1, [r0, #APLL_LOCK_OFFSET]
    str r1, [r0, #MPPLL_LOCK_OFFSET]

    // 3 设置分频
    // 清bit[0~31]
    ldr r1, [r0, #CLK_DIV0_OFFSET]
    ldr r2, =0x10001111
    orr r1, r1, r2
    str r1, [r0, #CLK_SRC0_OFFSET]

    mov pc, lr
```

先关闭看门狗

然后马上设置时钟, 跳转到时钟汇编函数

Clock_init 函数执行完后返回到 start.s
执行重定位, 最后跳到 C 去执行 led_link 函数

汇编设置时钟太麻烦，不好理解，那么我们用 C 语言来设置时钟

The diagram illustrates the flow of control from assembly code in `start.S` to C code in `clock.c` and `led.c`.

start.S:

```
* 描述： 演示汇编开关icache
*/
#define WTCON      0xE2700000
#define SVC_STACK   0xd0037d80
.global _start          // 把_start链接属性改为外部，这样其他文件就可以直接调用_start了
_start:
    // 第1步：关看门狗（向WTCON的bit5写入0即可）
    ldr r0, =WTCON
    ldr r1, =0x0
    str r1, [r0]

    // 第2步：初始化时钟
    bl clock_init          // 时钟初始化由汇编文件改成C文件的clock.c

    // 第3步：设置SVC栈
    ldr sp, =SVC_STACK

    // 第4步：开/关icache
    mrc p15, 0, r0, c1, c0, 0;           // 读出cp15的c1到r0中
    //bic r0, r0, #(1<<12)           // bit12 置0 关icache
    orr r0, r0, #(1<<12)             // bit12 置1 开icache
    mcr p15, 0, r0, c1, c0, 0;

    // 从这里之后就可以开始调用C程序了
    bl led_blink             // led_blink是C语言实现的一个函数

// 汇编最后的这个死循环不能丢
    b .
```

clock.c:

```
#define REG_CLK_SRC0  (ELFIN_CLOCK_POWER_BASE + CLK_DIV0_OFFSET)
#define REG_APLL_CON0 (ELFIN_CLOCK_POWER_BASE + APLL_CON0_OFFSET)
#define REG_MPPLL_CON (ELFIN_CLOCK_POWER_BASE + MPPLL_CON_OFFSET)

#define rREG_CLK_SRC0  (*(volatile unsigned int *)REG_CLK_SRC0)
#define rREG_APLL_LOCK (*(volatile unsigned int *)REG_APLL_LOCK)
#define rREG_MPPLL_LOCK (*(volatile unsigned int *)REG_MPPLL_LOCK)
#define rREG_CLK_DIV0  (*(volatile unsigned int *)REG_CLK_DIV0)
#define rREG_APLL_CON0 (*(volatile unsigned int *)REG_APLL_CON0)
#define rREG_MPPLL_CON (*(volatile unsigned int *)REG_MPPLL_CON)

void clock_init(void)
{
    // 1 设置各种时钟开关，暂时不使用PLL
    rREG_CLK_SRC0 = 0x0;

    // 2 设置锁定时间，使能默认值即可
    // 设置PLL后，时钟从Fin提升到目标频率时，需要一定的时间，即锁定时间
    rREG_APLL_LOCK = 0x0000ffff;
    rREG_MPPLL_LOCK = 0x0000ffff;

    // 3 设置分频
    // 清bit[0~31]
    rREG_CLK_DIV0 = 0x14131440;

    // 4 设置PLL
    // FOUT = MDIV*FIN/(PDIV*2^(SDIV-1))=0x7d*24/(0x3*2^(1-1))=1000 MHz
    rREG_APLL_CON0 = APLL_VAL;
    // FOUT = MDIV*FIN/(PDIV*2^SDIV)=0x29b*24/(0xc*2^1)= 667 MHz
    rREG_MPPLL_CON = MPPLL_VAL;

    // 5 设置各种时钟开关，使用PLL
    rREG_CLK_SRC0 = 0x10001111;
}
```

led.c:

```
void led_blink(void)
{
    // led初始化，也就是把GPJ0CON中设置为输出模式
    //volatile unsigned int *p = (unsigned int *)GPJ0CON;
    //volatile unsigned int *p1 = (unsigned int *)GPJ0DAT;
    rGPJ0CON = 0x11111111;

    while (1)
    {
        // led亮
        rGPJ0DAT = ((0<<3) | (0<<4) | (0<<5));
        // 延时
        delay();
        // led灭
        rGPJ0DAT = ((1<<3) | (1<<4) | (1<<5));
    }
}
```

Annotations in the diagram:

- A callout points to the `bl clock_init` instruction in `start.S` with the text: "时钟初始化由汇编文件改成 C 文件的 clock.c".
- A callout points to the `bl led_blink` instruction in `start.S` with the text: "时钟初始化完成后调用 C 语言执行 led 文件".

裸机汇编程序下载到什么地方？程序从内存哪行地址开始启动？

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-linux-gcc-4.6.2 -c led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.o led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf led.o  
arm-none-linux-gnueabi-ld: warning: cannot find entry symbol _start; defaulting to 00000000  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.elf led.o led.s  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# arm-none-linux-gnueabi-objcopy -O binary led.elf led.bin  
root@ubuntu:/home/xiang/S5PV210/noOS_driver/s_driver# ls  
led.bin led.elf led.o led.s
```

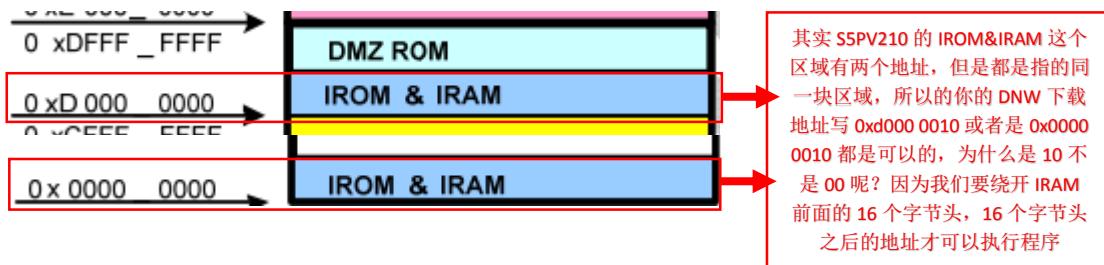
汇编程序下载到内存 0x0 地址，然后从 0 地址开始运行 -Ttext 就是指定程序启动地址

但是实际程序下载的地址是由我们 DNW 软件设置的

- USB Port

Download Address **0xd0020010**

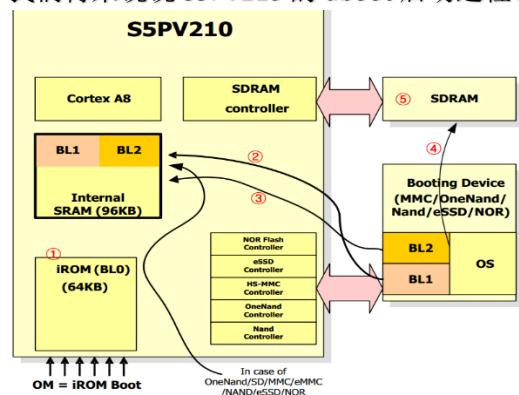
所以程序下载的位置是在 S5PV210 里面的 SRAM 里面，地址为 0xd0020010



所以说在编译的时候-Ttext 0x0 是指定理想的地址运行，但是实际是要根据 DNW 设定的地址来确定程序启动位置，这个地址必须是下载到 0x0000 0010 或者 0xd000 0010，下载到其它地址程序运行不起来，因为芯片厂家规定板子上电后，IROM 的 BL0 加载 BL1 的地址就在 0x0000 0010 开始读取程序运行

因为我们前面写的 LED 汇编代码比较简单，所以我们不关心汇编指令里面有没有位置有关码，但是在程序复杂的汇编指令里面，绝对有指令是位置有关码，位置有关码就是不能随便指定地址执行，必须根据指令来确定地址位置。

我们再来说说 S5PV210 的 uboot 启动过程。



这个图是三星官方要求你的 uboot 大小小于 96KB，板子上电后 IROM 先读取 EMMC 里面 BL1 的 uboot 代码 16KB 到 SRAM，然后再读取 80KB 的 uboot 代码到 BL2，然后去用 BL2 的 Uboot 代码去初始化 SDRAM，然后加载 OS 到 SDRAM。最后 BL1, BL2 里面的 uboot 执行长跳转指令，把运行程序指针跳转到 SDRAM 上。这样就启动完成了。

但是我们的 uboot 都是大于 96KB 的，怎么办呢？一般都是 200 多 KB

解决方法：板子上电后先加载 EMMC 里 BL1 的前 16KB 大小 uboot 代码到 SRAM，在 SRAM 中用 BL1 代码初始化 SDRAM(DDR)，然后将 BL1 和 BL2 的 Uboot 整个搬运到我们的 SDRAM 中，然后 SRAM 中的 BL1 用一句长跳转指令，跳转到 SDRAM 继续执行 uboot，然后加载 OS

SD 卡启动程序(把程序烧录进 SD 卡启动)

Nandflash: 读写时序复杂，没有坏块处理，不同型号接线关系和时序都不一致

SLC 的 Nandflash 可靠性高，不容易坏块，存储容量最大 512M

MLC 的 Nandflash 容易坏块，容量大可以上 1G，价格便宜

为了解决 MLC 的问题，开发了 SD 卡，tf 卡和 MMC 卡。卡内自带主控，解决接线不一致和坏块问题。读写时序一致，引脚一致。

为了解决插卡的热插拔不方便问题，现在出了 iNand，essd，emmc 来解决外置插卡问题，而且时序和接口与 SD 卡标准一致。

图二 SD卡和Micro SD (TF) 卡的管脚定义

引脚号	SD卡	TF卡 (SD模式)	TF卡 (SPI模式)
1	Data3	Data2	Rsv
2	Cmd	Data3	Cs
3	Vss	Cmd	Di
4	Vdd	Vdd	Vdd
5	Clk	Clk	Sclk
6	Vss	Vss	Vss
7	Data0	Data0	Do
8	Data1	Data1	Rsv
9	Data2	--	--

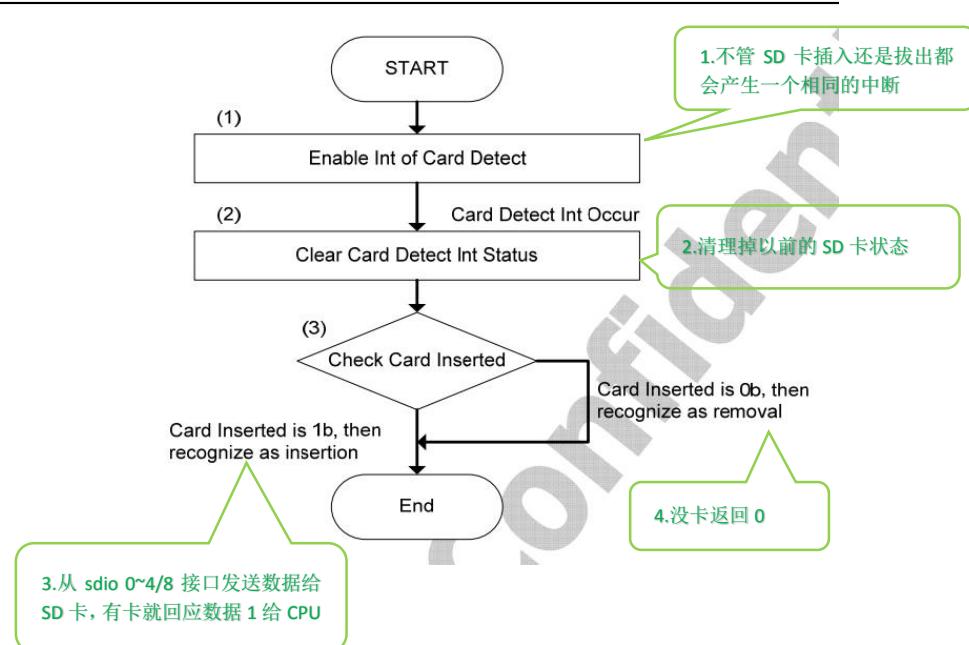
SRAM/DDR/SROM 可以直接用 C 语言指针操作地址来访问

SD 卡/NANDflash 不能直接指针操作地址来访问，必须写驱动

as clock rate is 52-MHz and access 8-bit data pin simultaneously.

SDIO 卡支持 52M 的传输速度，SDIO 接口是 8 个引脚

SD CARD DETECTION SEQUENCE



如何使用 SD 卡启动系统

2.6 Global Variable

If the MMC device is used to boot up, the information of MMC card must be saved in the special area. Refer to table 2 and Figure 3.

Address	Name	Usage
0xD0037480	globalBlockSize	Total block count of the SD/MMC device.
0xD0037484	globalSDHCInfoBit	globalSDHCInfoBit[31:16] : RCA(Relative Card Address) Data globalSDHCInfoBit[2] : SD Card globalSDHCInfoBit[1] : MMC Card globalSDHCInfoBit[0] : High Capacity Enable
0xD0037488	V210_SDMMC_BASE	Current boot channel. CPU 的 SDIO 通道选择, 根据你 SD 卡接入的 SDIO 通道, 来选择 SDIO 通道

如果使用 SD 卡启动, 那么你使用的 SD 卡的一些信息要保存在这个区域

这个地址要写入使用 SD 卡的块大小是多大

SD 卡选这个

高容量 SD 卡选这个(SDHC)

CPU 如何读取 SD 卡

2.7 Device Copy Function

The S5PV210 internally has a ROM code of block copy function for boot-u device. Therefore, developer may not needs to implements device copy functions. These internal functions can copy any data from memory devices to SDRAM. User can use these function after ending up the internal ROM boot process completely.

Address	Name	Usage
0xD0037F90	NF8_ReadPage_Adv	This internal function is advanced NF8_ReadPage function. (8-Bit ECC Check) Note. 2048, 4096 Page 8bits-bus Nand Only.
0xD0037F94	NF16_ReadPage_Adv	This internal function is advanced NF16_ReadPage function. (4-Bit ECC Check) Note. 2048 page size, 5 cycle address, 16bits-bus Nand Only.
0xD0037F98	CopySDMMCtoMem	This internal function can copy any data from SD/MMC device to SDRAM. User can use this function after the IROM boot process completely.
0xD0037F9C	CopyMMC4_3toMem	This internal function can copy any data from eMMC device to SDRAM. User can use this function after the IROM boot process completely.
0xD0037FA0	CopyOND_ReadMultiPages	This internal function can copy any data from OneNand device to SDRAM. User can use this function after the IROM boot process completely. (normal speed copy)
0xD0037FA4	CopyOND_ReadMultiPages_Adv	This internal function can copy any data from OneNand device to SDRAM. User can use this function after the IROM boot process completely. (fast speed copy)

读 sd 卡选这个

读取 8 位接口的 NANDflash, 支持 NAND 的页大小为 2048, 4096

读 16 位接口的 NANDflash, 支持 NAND 的页大小为 2048

● SD/MMC Copy Function Address

External source clock parameter is used to fit EPLL source clock at 20MHz.

```
/*
 * This Function copy MMC(MoviNAND/iNand) Card Data to memory.
 * Always use EPLL source clock.
 * This function works at 20Mhz.
 * @param u32 StartBlkAddress : Source card(MoviNAND/iNand MMC) Address.(It must block address.)
 * @param u16 blockSize : Number of blocks to copy.
 * @param u32* memoryPtr : Buffer to copy from.
 * @param bool with_init : determined card initialization.
 * @return bool(u8) - Success or failure.
 */
#define CopySDMMCtoMem(z,a,b,c,e)((bool(*)(int, unsigned int, unsigned short, unsigned int*, bool))(*((unsigned int *)0xD0037F98)))(z,a,b,c,e))

    
```

1 个扇区 = 512 字节，也可以 1 个扇区数 1024, 2048 字节。

但是为了兼容的以前操作系统，1 个扇区大家都默认为 512 字节。

扇区 = 块，扇区就是块(block)，叫法不同而已。

块设备有：硬盘，软盘，DVD，flash 设备(u 盘，SSD 硬盘，SD 卡，NANDflash, eMMC, iNAND)。

Linux 的 MTD 驱动是用来管理块设备。

块设备就是用一块一块来读写的，就类似于内存是一个字节一个字节读写。

一块一块，可以是 512, 1024 或者 2048 字节，

● SD/MMC Copy Function Address

External source clock parameter is used to fit EPLL source clock at 20MHz.

```
/*
 * This Function copy MMC(MoviNAND/iNand) Card Data to memory.
 * Always use EPLL source clock.
 * This function works at 20Mhz.
 * @param u32 StartBlkAddress : Source card(MoviNAND/iNand MMC) Address.(It must block address.)
 * @param u16 blockSize : Number of blocks to copy.
 * @param u32* memoryPtr : Buffer to copy from.
 * @param bool with_init : determined card initialization.
 * @return bool(u8) - Success or failure.
 */
#define CopySDMMCtoMem(z,a,b,c,e)((bool(*)(int, unsigned int, unsigned short, unsigned int*, bool))(*((unsigned int *)0xD0037F98)))(z,a,b,c,e))

/*第1种方法宏定义*/
#define CopySDMMCtoMem(z, a, b, c, e) ((bool(*)(int, unsigned int, unsigned short, unsigned int*, bool))(*((unsigned int *)0xD0037F98)))(z, a, b, c, e)
CopySDMMCtoMem(x, x, x, x, x);

/*第2种方法，用函数指针*/
typedef unsigned int bool;
typedef bool (*CopySDMMCtoMem)(int, unsigned int, unsigned short, unsigned int*, bool);

CopySDMMCtoMem p1 = (CopySDMMCtoMem)0xD0037F98;
p1(x, x, x, x, x); //第1种函数调用写入的5个参数

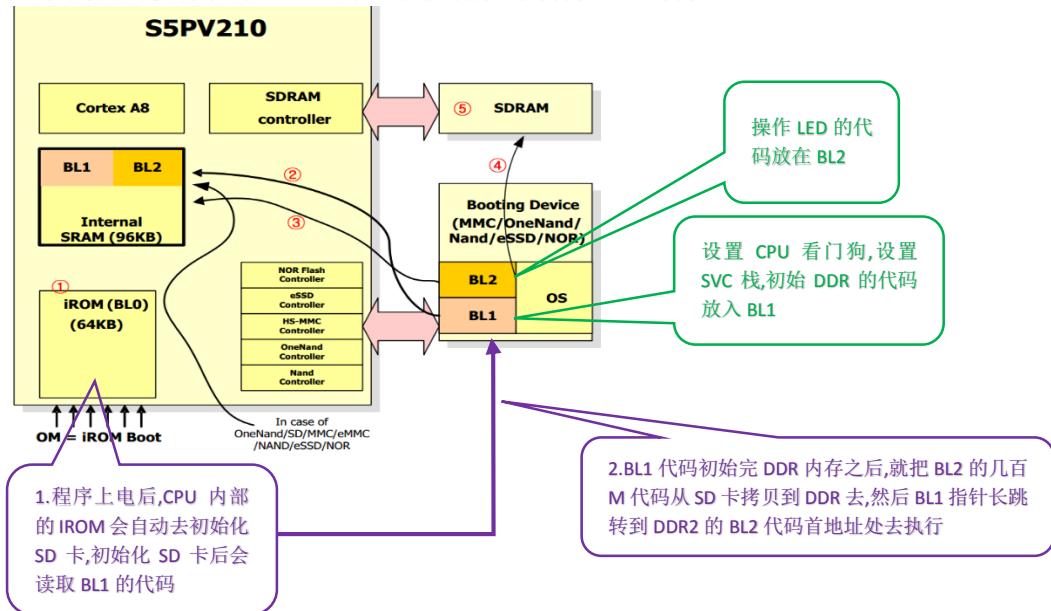
(*p1)(x, x, x, x, x); //第2种函数调用写入的5个参数
    
```

如何实现手册要求的代码

SD 卡启动代码实现

我们的执行程序<16 KB 可以直接烧录进开发板使用

如果我们的执行程序>16KB 那么就要把程序拆分成 2 部分



```
root@ubunt:~#
BL1 BL2
```

我们将 BL1 和 BL2 分别建立一个文件夹来存放各自的 BL1 和 BL2 代码

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/SDcardboot/BL1# ls
link.lds  Makefile  mkv210_image.c  s5pv210.h  sdram_init.S  sd_relocate.c  start.S
```

BL1 的代码第 1 执行 start.S, 然后现在 start.S 里面去调用 sdram_init.S 和 SD_relocate.c

Start.S 代码

```
#define WTC0N 0xE2700000
#define SVC_STACK 0xd0037d80

.global _start// 把_start链接属性改为外部，这样其他文件就可以看见_start了
_start:
    // 第1步：关看门狗（向WTC0的bits写入0即可）
    ldr r0, =WTC0N
    ldr r1, =0x0
    str r1, [r0]

    ldr sp, =SVC_STACK // 第2步：设置SVC栈

    /*第3步：开/关icache*/
    mrc p15,0,r0,c1,c0,0; // 读出cp15的c1到r0中
    orr r0, r0, #(1<<12) // bit12 置1 开icache
    mcr p15,0,r0,c1,c0,0;

    bl sdram_asm_init // 第4步：初始化ddr
    bl copy_b12_2_ddr // 第5步：重定位，从SD卡第45扇区开始，复制32个扇区内容到DDR的0x23E00000
    b .
```

sdram_init.S 代码初始化 DDR,一般官方写

```
.global sdram_asm_init
sdram_asm_init:
    ldr r0, =0xf1e00000
    ldr r1, =0x0
    str r1, [r0, #0x0]

    /* DMCO Drive Strength (Setting 2X) */
    ldr r0, =ELFIN_GPIO_BASE
    ldr r1, =0x0000AAAA
```

汇编程序执行完 sdram_asm_init 之后会接着执行 copy_b12_2_ddr. copy_b12.....函数在下面 sd_relocate.c 里面

所以 sdram_init.s 执行完后 DDR 就可以使用了

SD_relocate.c 代码

```
#define SD_START_BLOCK 45
#define SD_BLOCK_CNT 32
#define DDR_START_ADDR 0x23E00000

typedef unsigned int bool;
typedef bool(*pCopySDMMC2Mem)(int, unsigned int, unsigned short, unsigned int*, bool);
typedef void (*pBL2Type)(void);

void copy_bl2_2_ddr(void)
{
    pCopySDMMC2Mem p1 = (pCopySDMMC2Mem)(*(unsigned int *)0xD0037F98); // 第一步，读取SD卡扇区到DDR中
    p1(2, SD_START_BLOCK, SD_BLOCK_CNT, (unsigned int *)DDR_START_ADDR, 0); // 读取SD卡到DDR中
    pBL2Type p2 = (pBL2Type)DDR_START_ADDR; // 第二步，跳转到DDR中的BL2去执行
    p2();
}
```

link.lds 链接脚本

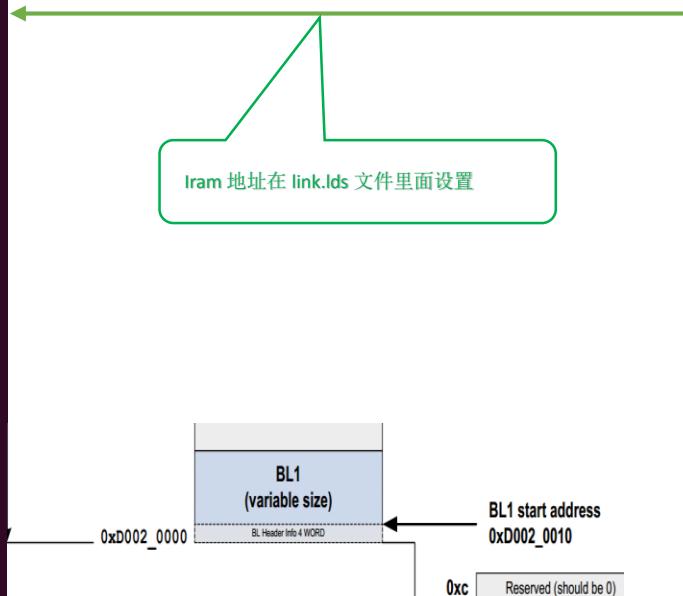
```
SECTIONS
{
    . = 0xD0020010;

    .text : {
        start.o
        sram_init.o
        * (.text)
    }

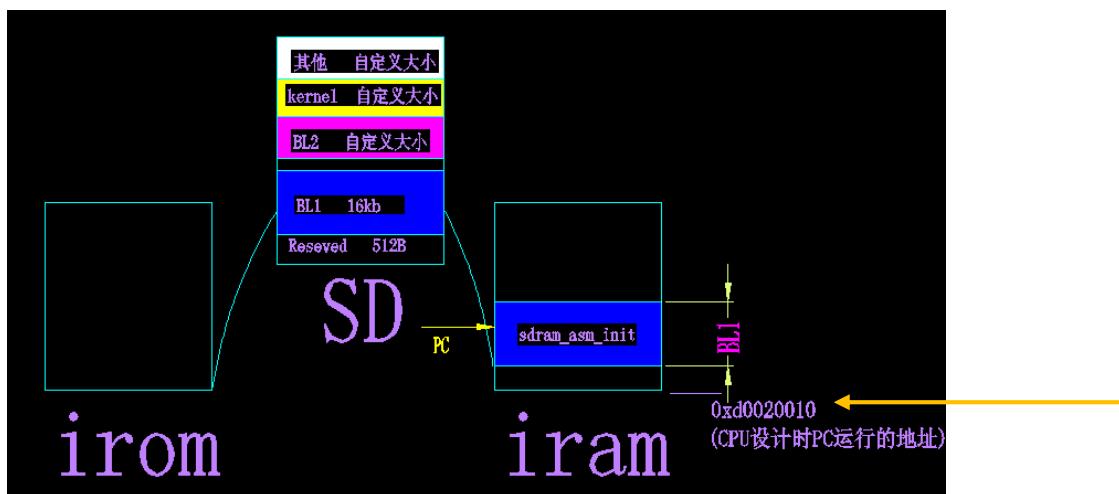
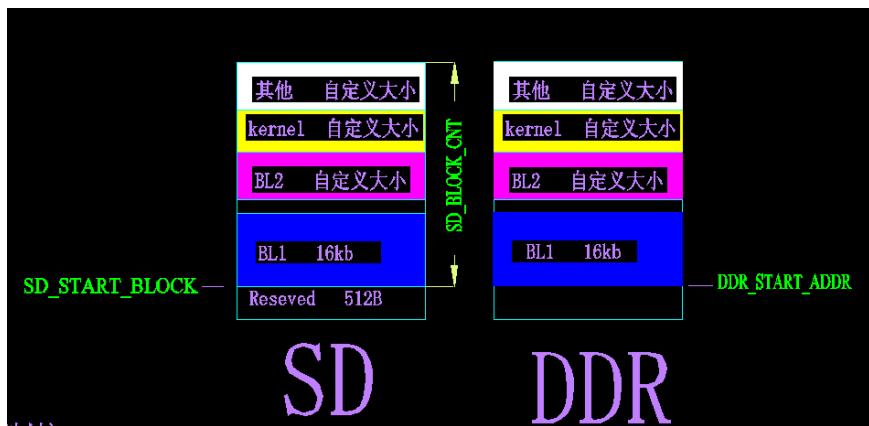
    .data : {
        * (.data)
    }

    bss_start = .;
    .bss : {
        * (.bss)
    }

    bss_end = .;
}
```



```
bootloader1.bin: start.o sram_init.o sd_relocate.o
arm-linux-ld -Tlink.lds -o bootloader1.elf $^
arm-linux-objcopy -O binary bootloader1.elf bootloader1.bin
arm-linux-objdump -P bootloader1.elf > bootloader1_elf.dis
gcc mkv210_image.c -o mkv210
./mkv210 bootloader1.bin BL1.bin
3.将链接成一体的代码编译成二进制文件烧写进开发板
%.o : %.S
    arm-linux-gcc -o $@ $< -c -nostdlib
%.o : %.c
    arm-linux-gcc -o $@ $< -c -nostdlib
clean:
    rm *.o *.elf *.bin *.dis mkv210 -f
1. 现将 start.S, sram_init.S, SD relocate.c 代码编译成.o
2. 用链接脚本 link.lds 将的代码链接成一体,放在 0xD002 0010 地址处, 这样iram 读的时候就将 BL1 代码读取到 iram 的 0xD002 0010 处
```



这是 BL1 就在 CPU 内部 iram 里面执行 start.S, sram_asm_init.S, SD_relocate.c 的代码
执行 SD_relocate.c 文件里 copy_b12_2ddr 函数的时候，就是把 SD 卡后面扇区的 BL2 代码拷贝到 DDR

```
#define SD_START_BLOCK 45
#define SD_BLOCK_CNT 32
#define DDR_START_ADDR 0x23E00000

typedef unsigned int bool;
typedef bool (*pCopySDMMC2Mem)(int, unsigned int, unsigned short, unsigned int*, bool);
typedef void (*pBL2Type)(void);

void copy_b12_2_ddr(void)
{
    pCopySDMMC2Mem p1 = (pCopySDMMC2Mem)(*(unsigned int *)0xD0037F98); // 第一步，读取SD卡扇区到DDR中
    p1(2, SD_START_BLOCK, SD_BLOCK_CNT, (unsigned int *)DDR_START_ADDR, 0); // 读取SD卡到DDR中
    pBL2Type p2 = (pBL2Type)DDR_START_ADDR; // 第二步，跳转到DDR中的BL2去执行
    p2(); // P2 函数去调用内存 BL2 地址的 start.c 代码
}
```

SD_START_BLOCK: 从 SD 卡 45 扇区开始拷贝，将 45 扇区后的 BL2 代码拷贝到 DDR 的 DDR_START_ADDR 地址

SD_BLOCK_CNT: 读取 32 个扇区到 DDR，这 32 个扇区就是从 45 扇区开始读，读到 77 扇区结束

DDR_START_ADDR: 就是 SD_BLOCK_CNT 读取的 32 个扇区的第一个扇区放在 DDR 的 0x23E00000 这个位置，也就是 BL2 代码在 DDR 的首地址

2.8.1 SD/MMC/eSSD Device Boot Block Assignment

<1Block=512B> (SD/MMC/eSSD)			
Block0	Block1 ~ (N-1)	BlockN ~ (M-1)	BlockM ~ EB(End of Block)
Mandatory	Recommendation		
Reserved (512B)	BL1	BL2	Kernel User File System

其实 BL1 大小 16K 只占用了 32 个扇区，为什么要从第 45 个扇区开始拷贝 BL2，这是为了安全起见将两份代码用扇区隔离下

```
@UBUNTU
BL2
Ubuntu root@ubuntu:/home/xiang/S5PV210/noOS_driver/SDcardboot/BL2# ls
led.c link.lds Makefile start.S
```

BL2 里面的代码

Start.s 代码

```
1
2 #define WTC0N 0xE2700000
3 #define SVC_STACK 0xd0037d80
4
5 .global _start // 把_start链接属性改为外部，这样其他文件就可以看见_start了
6 _start:
7     ldr pc, =main// ldr指令实现长跳转
8     b .
```

led.c 代码(这里就是我们主程序代码了)

```
void main(void)
{
    rGPJ0CON = 0x11111111;

    while (1)
    {
        rGPJ0DAT = ((0<<3) | (0<<4) | (0<<5));

        delay();

        rGPJ0DAT = ((1<<3) | (1<<4) | (1<<5));
    }
}
```

link.lds 链接脚本，这里的 BL2 文件的代码我们是要跳转到 DDR2 的 0x23E0 0000 地址上执行

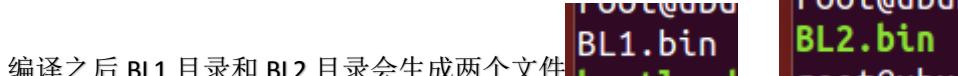
```
SECTIONS
{
    . = 0x23E00000;
    .text : {
        start.o
        * (.text)
    }
    .data : {
        * (.data)
    }
    bss_start = .;
    .bss : {
        * (.bss)
    }
    bss_end = .;
}
```

为什么会 start.c 代码的首执行地址是在 0x23e0 0000 呢，因为我们在 IRAM 用 BL1 拷贝 BL2 代码到 DDR 的时候就指定了地址

```
BL2.bin: start.o led.o
| arm-linux-ld -Tlink.lds -o BL2.elf $^
| arm-linux-objcopy -O binary BL2.elf BL2.bin
| arm-linux-objdump -D BL2.elf > BL2_elf.dis
%
%.o : %.S
| arm-linux-gcc -o $@ $< -c -nostdlib
%
%.o : %.c
| arm-linux-gcc -o $@ $< -c -nostdlib
clean:
| rm *.o *.elf *.bin *.dis mks210 -f
```

Makefile 一样





编译之后 BL1 目录和 BL2 目录会生成两个文件
那么如何将这两个程序放在 SD 卡里面去执行呢？

write2sd.sh 这时候就需要写个 shell

```
#!/bin/sh
sudo dd iflag=dsync oflag=dsync if=./BL1/BL1.bin of=/dev/sdb seek=1
sudo dd iflag=dsync oflag=dsync if=./BL2/BL2.bin of=/dev/sdb seek=45
```

将指定目录 BL1 的二进制文件

拷贝进 sd 卡
，这个 sdb 设备节点 X86 上必须是插入 sd 卡才有

BL1 二进制文件拷贝到 SD 卡第 1 个扇区，为什么是第 1 个扇区，而不是第 0 个呢

2.8.1 SD/MMC/eSSD Device Boot Block Assignment

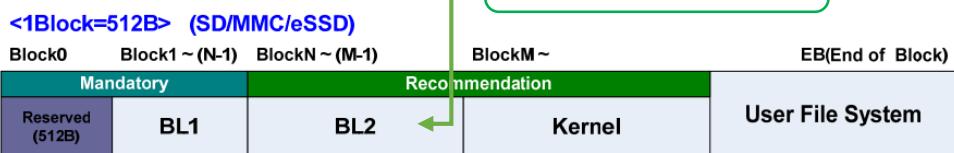


因为 CPU 的 IROM 不识别第 0 个扇区，而是从 SD 卡第 1 个扇区开始读，所以我们就将就 CPU 咯

```
#!/bin/sh
sudo dd iflag=dsync oflag=dsync if=./BL1/BL1.bin of=/dev/sdb seek=1
sudo dd iflag=dsync oflag=dsync if=./BL2/BL2.bin of=/dev/sdb seek=45
```

然后将 BL2 二进制文件拷贝到 SD 卡第 45 扇区开始

2.8.1 SD/MMC/eSSD Device Boot Block Assignment

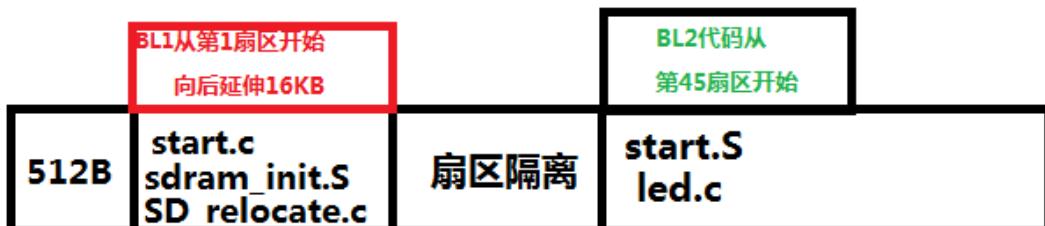


为什么 BL2 是第 45 个扇区，我们说了两个代码要做扇区隔离

烧录完之后将 SD 卡取出 Ubuntu，插入开发板指定的 SDIO 通道，选择 SD 卡启动就行

最后无限循环的是 BL2 的 led.c 代码，这样就对了

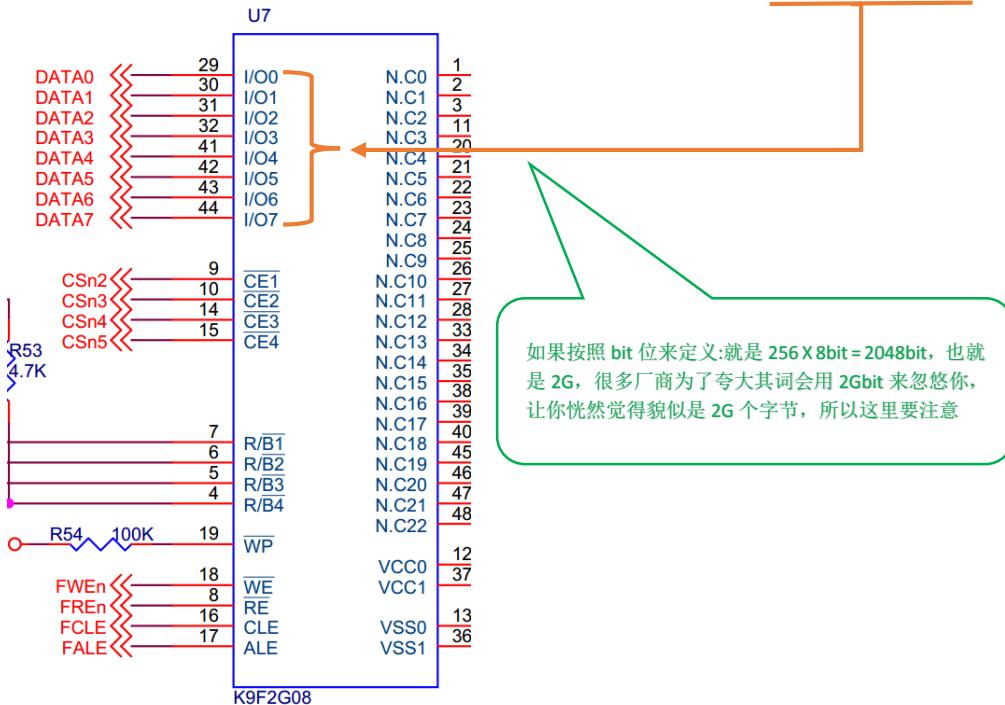
SD 卡内部布局



Nandflash 接口

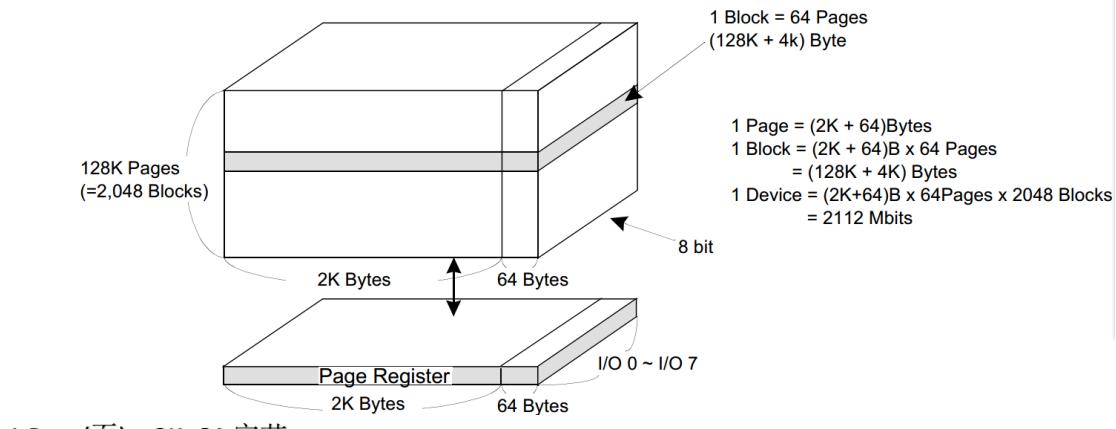
256M x 8 Bit / 128M x 16 Bit NAND Flash Memory

256M X 8bit 256M 意思就是这个 Nand 存储大小 256MB 字节. X8 就是线宽为 8



128M X 16 bit 就是这个 Nand 存储是 128M 字节，但是因为是 X16，数据线相对于 X8 翻了一倍，所以也是 2Gbit 的，也就等于 256M 字节存储量。

Figure 2-1. K9F2G08X0M (X8) Array Organization



1 Page(页) = 2K+64 字节

1 Block(块) = 1 Page(页) X 64

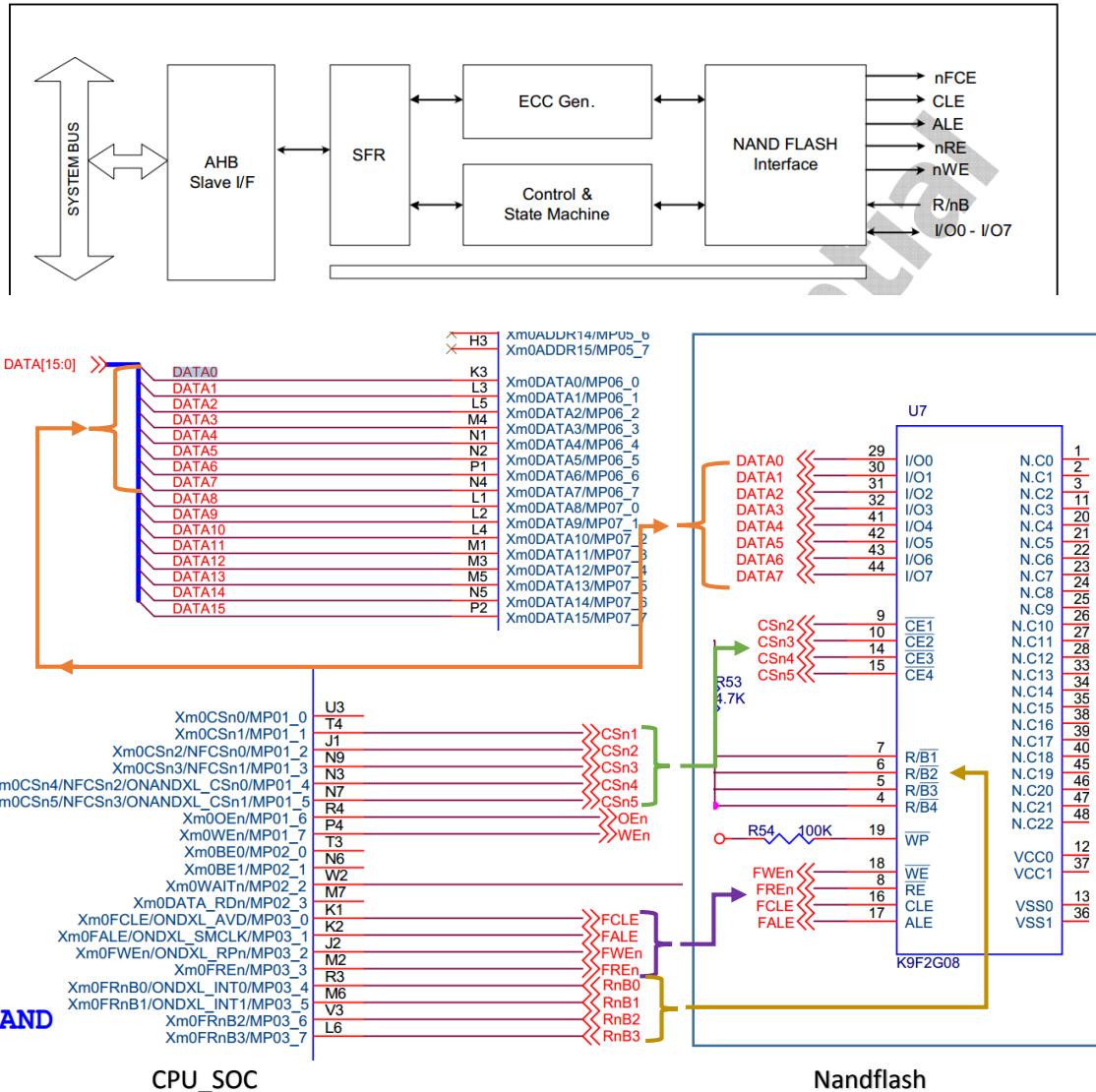
1 个 Nandflash = 2048 Block

我们读写 NANDflash 必须是 1 Page(页) (2k+64byte)，一页一页的读取，不能一个字节一个字节读取。

其实 Nandflash 的页就等同于磁盘，机械硬盘的扇区。因为早期的磁盘和机械硬盘 1 个扇区是 512Byte，但是根据技术的发展现在的硬盘也有 1024Byte，2048Byte。所以 Nandflash 根据型号的不同，也有 512Byte = 1Page，1024Byte = 1Page 2048Byte = 1Page 等等.....

1 个 Block 等于多少 Page 页，也是根据 Nand 型号不同而不同。

4.2.1 BLOCK DIAGRAM



S5PV210 操作 Nandflash 程序

```

#define rNFCONF      ( *((volatile unsigned long *) 0xB0E00000) )
#define rNFCONT      ( *((volatile unsigned long *) 0xB0E00004) )
#define rNFCMMD      ( *((volatile unsigned long *) 0xB0E00008) )
#define rNFADDR      ( *((volatile unsigned long *) 0xB0E0000C) )
#define rNFDATA      ( *((volatile unsigned long *) 0xB0E00010) )
#define rNFDATA8     ( *((volatile unsigned char *) 0xB0E00010) )
#define rNFSTAT      ( *((volatile unsigned long *) 0xB0E00028) )

#define rMP0_1CON    ( *((volatile unsigned long *) 0xE02002E0) )
#define rMP0_2CON    ( *((volatile unsigned long *) 0xE0200300) )
#define rMP0_3CON    ( *((volatile unsigned long *) 0xE0200320) )

```

这是要用到的 S5PV210 Nandflash 控制器地址

```

#define NAND_CMD_READ_1st          0x00      /* 命令 */
#define NAND_CMD_READ_2st          0x30

#define NAND_CMD_READ_CB_1st       0x00
#define NAND_CMD_READ_CB_2st       0x35

#define NAND_CMD_RANDOM_WRITE     0x85
#define NAND_CMD_RANDOM_READ_1st  0x05
#define NAND_CMD_RANDOM_READ_2st  0xe0

#define NAND_CMD_READ_ID          0x90
#define NAND_CMD_RESET             0xff
#define NAND_CMD_READ_STATUS       0x70

#define NAND_CMD_WRITE_PAGE_1st    0x80
#define NAND_CMD_WRITE_PAGE_2st    0x10

#define NAND_CMD_BLOCK_ERASE_1st   0x60
#define NAND_CMD_BLOCK_ERASE_2st   0xd0

```

这是 K9F2G08 Nandflash 需要的命令码，需要 S5PV210 的 nand 控制器发送

```

// nandflash 初始化
void nand_init(void)
{
    // 1. 配置nandflash
    rNFCONF = (TACLS<<12) | (TWRPH0<<8) | (TWRPH1<<4) | (0<<3) | (0<<2) | (1<<1) | (0<<0) ;
    rNFCONT = (0<<18) | (0<<17) | (0<<16) | (0<<10) | (0<<9) | (0<<8) | (0<<7) | (0<<6) | (0x3<<1) | (1<<0) ;

    // 2. 配置引脚
    rMP0_1CON = 0x22333322;
    rMP0_2CON = 0x00002222;
    rMP0_3CON = 0x22222222;

    // 3. 复位
    nand_reset();
}

// 复位
void nand_reset(void)
{
    nand_select_chip();
    nand_send_cmd(NAND_CMD_RESET);
    nand_wait_idle();
    nand_deselect_chip();
}

// 等待就绪
void nand_wait_idle(void)
{
    unsigned long i;
    while( !(rNFSTAT & (1<<4)) )
        for(i=0; i<10; i++);
}

```

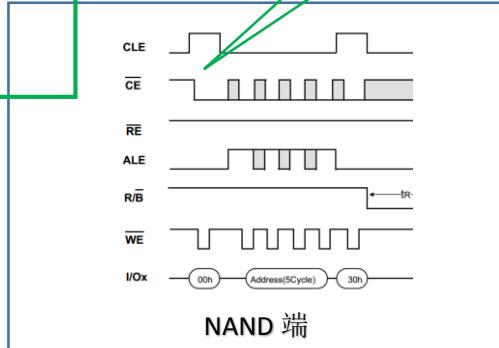
```
// 发片选
void nand_select_chip(void)
{
    unsigned long i;
    rNFCONT &= ~(1<<1);
    for(i=0; i<10; i++);
}
```

将 CE 片选拉低，这样才能选中 NAND 芯片

NAND 芯片 CE 变低才能接受 CPU 控制

Reg_nCE1	[2]	NAND Flash Memory nRCS[1] si
Reg_nCE0	[1]	NAND Flash Memory nRCS[0] si 0 = Force nRCS[0] to low (Enable) 1 = Force nRCS[0] to High (Disable) Note: The setting all nCE[3:0] zero one nCE can be asserted to enable memory. The lower bit has more nCE[3:0] zeros.
MODE	[0]	NAND Flash controller operating 0 = Disable NAND Flash Controller 1 = Enable NAND Flash Controller

SOC 端



```
// 取消片选
void nand_deselect_chip(void)
{
    unsigned long i = 0;
    rNFCONT |= (1<<1);
    for(i=0; i<10; i++);
}
```

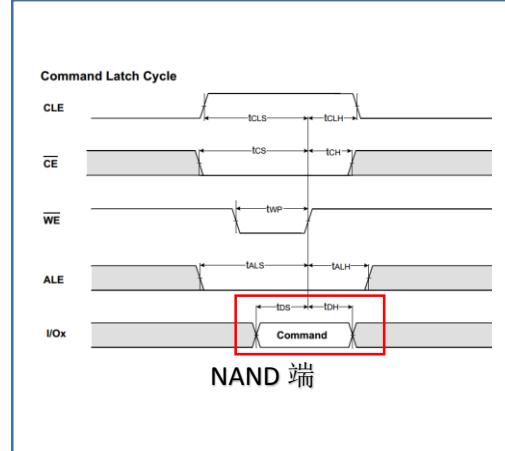
// 发命令

```
void nand_send_cmd(unsigned long cmd)
{
    unsigned long i = 0;

    rNFCMMD = cmd;
    for(i=0; i<10; i++);
}
```

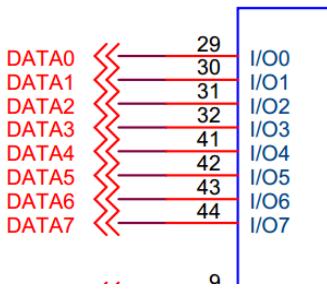
4.5.2.3 Command Register (NFCMMD, R/W, Address = 0xB0E0_000)		
NFCMMD	Bit	Description
Reserved	[31:8]	Reserved
REG_CMMD	[7:0]	NAND Flash memory command value

SOC 端



我们只向命令寄存器写数据就是，至于 WE, ALE 时序，SOC 会帮我们处理

为什么 cmd 是 8 位？



因为我们 Nandflash, IO 数据端口是 8 位的, 改端口根据时序要求, 既是数据线, 也是地址线, 而且还是命令值接受线。

```
// 擦除块, 参数为块号 (0 ~ MAX_NAND_BLOCK-1)
int nand_block_erase(unsigned long block_num)
{
    unsigned long i = 0;

    // 获得row地址, 即页地址
    unsigned long row = block_num * NAND_BLOCK_SIZE;

    // 1. 发出片选信号
    nand_select_chip();
    // 2. 擦除: 第一个周期发命令0x60, 第二个周期发块地址, 第三个周期发命令0xd0
    nand_send_cmd(NAND_CMD_BLOCK_ERASE_1st);
    for(i=0; i<10; i++);

    // Row Address A12~A19
    rNFADDR = row & 0xff;
    for(i=0; i<10; i++);
    // Row Address A20~A27
    rNFADDR = (row >> 8) & 0xff;
    for(i=0; i<10; i++);
    // Row Address A28~A30
    rNFADDR = (row >> 16) & 0xff;

    rNFSTAT |= (1<<4); // clear RnB bit

    nand_send_cmd(NAND_CMD_BLOCK_ERASE_2st);
    for(i=0; i<10; i++);
    // 3. 等待就绪
    nand_wait_idle();

    // 4. 读状态
    unsigned char status = nand_read_status();
    if (status & 1)
    {
        // status[0] = 1, 表示擦除失败, 详见NAND Flash数据手册中 READ STATUS一节的描述
        // 取消片选信号
        nand_deselect_chip();
        printf("masking bad block %d\r\n", block_num);
        return -1;
    }
    else
    {
        // status[0] = 0, 表示擦除成功, 返回0
        nand_deselect_chip();
        return 0;
    }
}
```

这里循环的向寄存器写地址, 寄存器不会被覆盖, 你写一次CPU内部会先移位处理一次

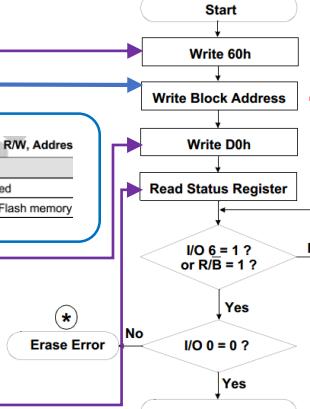
1page=2K, 1Block=64page

NAND_BLOCK_SIZE=64

4.5.2.4 Address Register (NFADDR, R/W, Address)		
NFADDR	Bit	R/W, Address
Reserved	[31:8]	Reserved
REG_ADDR	[7:0]	NAND Flash memory

发送 D0h 表示叫 NAND 擦除指定地址的块区域

Erase Flow Chart



NAND 端擦除流程

向 rNFADDR 写 3 次, 就将行地址发给 NAND 了, 因为擦除 NAND 不需要列, 只需要行地址

因为 NANDflash 擦除块成功后会给 SOC 的 RnB 引脚发送一个擦除成功的信号, 所以在接受这个成功信号之前我们先要把该寄存器位清空 0

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7
2nd Cycle	A8	A9	A10	A11	*L	*L	*L	*L
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27
5th Cycle	A28	*L						

Column Address

Column Address

Row Address

Row Address

Row Address

```

// 等待就绪
void nand_wait_idle(void)
{
    unsigned long i;
    while( !(rNFSTAT & (1<<4)) )
        for(i=0; i<10; i++);
}

unsigned char nand_read_status(void)
{
    unsigned char ch;
    int i;

    // 1. 发出片选信号
    nand_select_chip();

    // 2. 读状态
    nand_send_cmd(NAND_CMD_READ_STATUS);
    for(i=0; i<10; i++);
    ch = nand_read8(); // 接受来自 NAND 状态命令数据

    // 3. 取消片选
    nand_deselect_chip();
    return ch;
}

// 读一个字节的数据
unsigned char nand_read8(void)
{
    return rNFDATA8;
}

```

这就是擦除前我们清空的位，现在 NAND 擦除完了，看是否给 SOC 返回 1，证明擦除成功

RnB_TransDetect [4] When RnB[0] low to high transition or interrupt is issued if RnB_TransDetec write '1'.
0 = RnB transition is not detected
1 = RnB transition is detected

发送读 NAND 擦除成功的状态的命令

虽然写的是 32 位数据寄存器，但是我们 NAND 接的 CPU 总线 I/O 电路上是 8 位，所以这里只取 8 位数据，所以这里地址定义 char

4.5.2.5 Data Register (NFDATA, R/W, Address = 0xB0E0_0010)

NFDATA	Bit	Description
NFDATA	[31:0]	NAND Flash read/ program data value fo Note: For more information, refer to 4.3.1 Configuration in page 4-4 .

```
#define rNFDATA8 ((*((volatile unsigned char *)0xB0E00010))
```

```

int nand_page_read(unsigned int pgaddr, unsigned char *buf, unsigned int length)
{
    int i = 0;

    // 1 发出片选信号
    nand_select_chip();

    // 2 写页读命令1st
    nand_send_cmd(NAND_CMD_READ_1st);

    // 3 写入页地址
    rNFADDR = 0;
    rNFADDR = 0;
    rNFADDR = pgaddr&0xff;
    rNFADDR = (pgaddr>>8)&0xff;
    rNFADDR = (pgaddr>>16)&0xff;

    // 4 clear RnB
    rNFSTAT |= (1<<4);

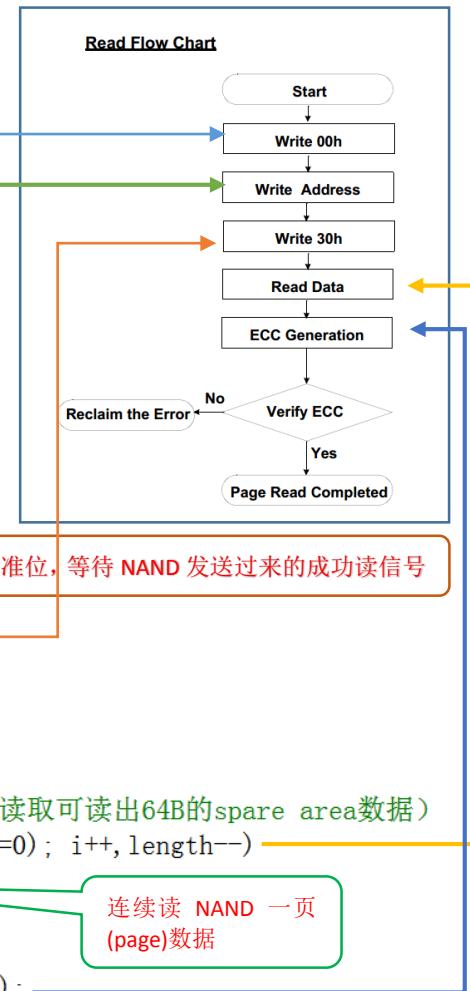
    // 5 写页读命令2st
    nand_send_cmd(NAND_CMD_READ_2st);

    // 6 等待空闲
    nand_wait_idle();

    // 7 连续读取2KB的Page main区数据 (继续读取可读出64B的spare area数据)
    for (i=0; (i<NAND_PAGE_SIZE) && (length!=0); i++, length--)
        *buf++ = nand_read8();

    // 8 读状态
    unsigned char status = nand_read_status();
    if (status & 1)
    {
        // 读出错, 取消片选信号, 返回错误码-1
        nand_deselect_chip();
        printf("nand random read fail\r\n");
        return -1;
    }
    else
    {
        // 读正确, 取消片选, 返回0
        nand_deselect_chip();
        return 0;
    }
}

```



清空标准位, 等待 NAND 发送过来的成功读信号

连续读 NAND 一页
(page)数据

```

]int nand_page_write(unsigned int pgaddr, const unsigned char *buf, unsigned int length)
{
    int i = 0;

    // 1 发出片选信号
    nand_select_chip();

    // 2 write cmd 1st
    nand_send_cmd(NAND_CMD_WRITE_PAGE_1st);

    // 3 write page addr
    rNFADDR = 0;
    rNFADDR = 0;
    rNFADDR = pgaddr&0xff;
    rNFADDR = (pgaddr>>8)&0xff;
    rNFADDR = (pgaddr>>16)&0xff;

    // 4 写入一页内容
    for(; i<NAND_PAGE_SIZE && length!=0; i++, length--)
        nand_write8(*buf++);

    // 5 clear RnB
    rNFSTAT = (rNFSTAT) | (1<<4);

    // 6 write cmd 2
    nand_send_cmd(NAND_CMD_WRITE_PAGE_2st);

    // 7 wait idle
    nand_wait_idle();

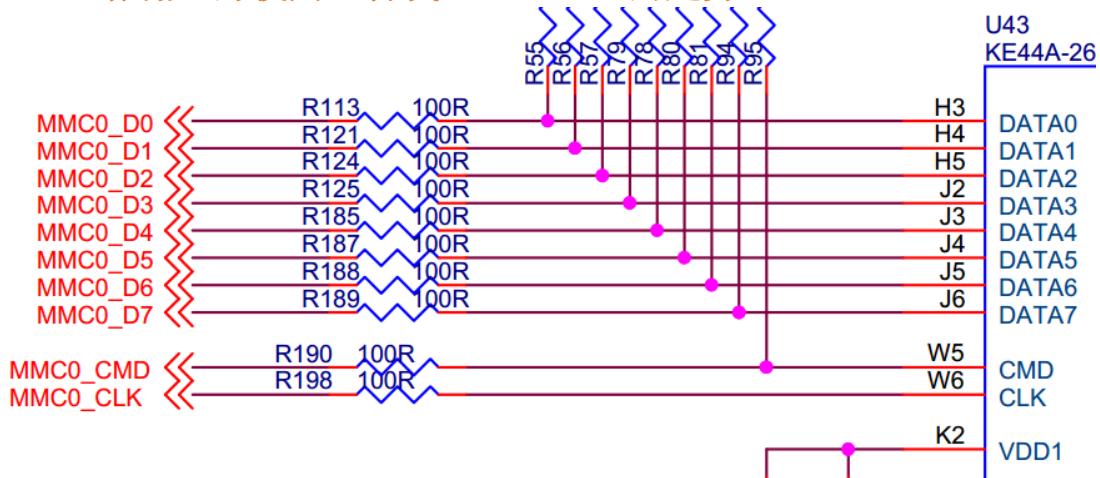
    // 8 读状态
    unsigned char status = nand_read_status();
    if (status & 1)
    {
        // 取消片选信号
        nand_deselect_chip();
        printf("nand random write fail\r\n");
        return -1;
    }
    else
    {
        nand_deselect_chip();
        return 0;
    }
}

```

写 NAND 一页和读 NAND 一页差不多



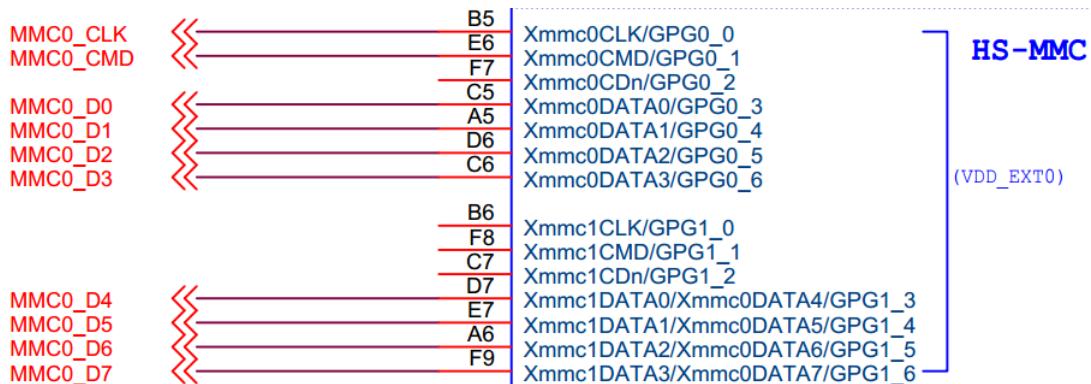
iNAND 存储芯片使用，替代 NANDflash 的趋势



iNAND 和 EMMC, SD 卡接口一样，都是 SDIO 接口(也叫做 MMC 卡接口)

SD 一般支持 4 位传输，所以是 D0~D3，4 根数据线

EMMC/iNAND 支持 8 位传输，所以是 D0~D7，8 根数据线



```

    #if (HSMMC_NUM == 0)
#define HSMMC_BASE (0xEB000000)
    #elif (HSMMC_NUM == 1)
#define HSMMC_BASE (0xEB100000)
    #elif (HSMMC_NUM == 2)

```

因为 Inand 接的是 SDIO0/MMC0 通道，所以该 SDIO0 控制器基地址是 EB000000

Register	Address	R/W	Description
SDMASYSAD0	0xEB00_0000	R/W	SDMA System Address register (Channel 0)
RI_KSI7E0	0xFB00_0001	R/W	Host DMA Buffer Boundary and Transfer

<pre> // 设置HSDIO的接口引脚配置 #if (HSMMC_NUM == 0) // channel 0, GPG0[0:6] = CLK, CMD, CDn, DAT[0:3] GPG0CON_REG = 0x2222222; // pull up enable GPG0PUD_REG = 0x2aaa; GPG0DRV_REG = 0x3fff; // channel 0 clock src = SCLK_EPLL = 96M CLK_SRC4_REG = (CLK_SRC4_REG & (~(0xf<<0))) (0x7<<0); // channel 0 clock = SCLK_EPLL/2 = 48M CLK_DIV4_REG = (CLK_DIV4_REG & (~(0xf<<0))) (0x1<<0); </pre>																													
<table border="1"> <tr> <td></td><td></td><td>0010 = SD_0_DATA[1] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[4]</td><td></td><td></td></tr> <tr> <td>GPG0CON[3]</td><td>[15:12]</td><td>0000 = Input 0001 = Output 0010 = SD_0_DATA[0] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[3]</td><td></td><td>0000</td></tr> <tr> <td>GPG0CON[2]</td><td>[11:8]</td><td>0000 = Input 0001 = Output 0010 = SD_0_CDn 0011 ~ 1110 = Reserved 1111 = GPG0_INT[2]</td><td></td><td>0000</td></tr> <tr> <td>GPG0CON[1]</td><td>[7:4]</td><td>0000 = Input 0001 = Output 0010 = SD_0_CMD 0011 ~ 1110 = Reserved 1111 = GPG0_INT[1]</td><td></td><td>0000</td></tr> <tr> <td>GPG0CON[0]</td><td>[3:0]</td><td>0000 = Input 0001 = Output 0010 = SD_0_CLK 0011 ~ 1110 = Reserved 1111 = GPG0_INT[0]</td><td></td><td>0000</td></tr> </table>						0010 = SD_0_DATA[1] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[4]			GPG0CON[3]	[15:12]	0000 = Input 0001 = Output 0010 = SD_0_DATA[0] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[3]		0000	GPG0CON[2]	[11:8]	0000 = Input 0001 = Output 0010 = SD_0_CDn 0011 ~ 1110 = Reserved 1111 = GPG0_INT[2]		0000	GPG0CON[1]	[7:4]	0000 = Input 0001 = Output 0010 = SD_0_CMD 0011 ~ 1110 = Reserved 1111 = GPG0_INT[1]		0000	GPG0CON[0]	[3:0]	0000 = Input 0001 = Output 0010 = SD_0_CLK 0011 ~ 1110 = Reserved 1111 = GPG0_INT[0]		0000	
		0010 = SD_0_DATA[1] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[4]																											
GPG0CON[3]	[15:12]	0000 = Input 0001 = Output 0010 = SD_0_DATA[0] 0011 ~ 1110 = Reserved 1111 = GPG0_INT[3]		0000																									
GPG0CON[2]	[11:8]	0000 = Input 0001 = Output 0010 = SD_0_CDn 0011 ~ 1110 = Reserved 1111 = GPG0_INT[2]		0000																									
GPG0CON[1]	[7:4]	0000 = Input 0001 = Output 0010 = SD_0_CMD 0011 ~ 1110 = Reserved 1111 = GPG0_INT[1]		0000																									
GPG0CON[0]	[3:0]	0000 = Input 0001 = Output 0010 = SD_0_CLK 0011 ~ 1110 = Reserved 1111 = GPG0_INT[0]		0000																									
2.2.15.3 Port Group GPG0 Control Register (GPG0PUD, R/W, Address)																													
<table border="1"> <thead> <tr> <th>GPG0PUD</th><th>Bit</th><th>Description</th><th></th></tr> </thead> <tbody> <tr> <td>GPG0PUD[n]</td><td>[2n+1:2n] n=0~6</td><td>00 = Pull-up/down disabled 01 = Pull-down enabled 10 = Pull-up enabled 11 = Reserved</td><td></td></tr> </tbody> </table>				GPG0PUD	Bit	Description		GPG0PUD[n]	[2n+1:2n] n=0~6	00 = Pull-up/down disabled 01 = Pull-down enabled 10 = Pull-up enabled 11 = Reserved																			
GPG0PUD	Bit	Description																											
GPG0PUD[n]	[2n+1:2n] n=0~6	00 = Pull-up/down disabled 01 = Pull-down enabled 10 = Pull-up enabled 11 = Reserved																											
2.2.15.4 Port Group GPG0 Control Register (GPG0DRV, R/W, Address)																													
<table border="1"> <thead> <tr> <th>GPG0DRV</th><th>Bit</th><th>Description</th><th></th></tr> </thead> <tbody> <tr> <td>GPG0DRV[n]</td><td>[2n+1:2n] n=0~6</td><td>00 = 1x 10 = 2x 01 = 3x 11 = 4x</td><td></td></tr> </tbody> </table>				GPG0DRV	Bit	Description		GPG0DRV[n]	[2n+1:2n] n=0~6	00 = 1x 10 = 2x 01 = 3x 11 = 4x																			
GPG0DRV	Bit	Description																											
GPG0DRV[n]	[2n+1:2n] n=0~6	00 = 1x 10 = 2x 01 = 3x 11 = 4x																											
<table border="1"> <tr> <td>MMC0_SEL</td><td>[3:0]</td><td>Control MUXMMC0, which is the source clock of MMC0 (0000: XXTI, 0001: XUSBXTI, 0010: SCLK_HDMI27M, 0011: SCLK_USBPHY0, 0100: SCLK_USBPHY1, 0101: SCLK_HDMIPHY, 0110: SCLKMPPLL, 0111: SCLK_EPLL, 1000: SCLKVPLL, OTHERS: reserved)</td><td>0x0</td></tr> </table>				MMC0_SEL	[3:0]	Control MUXMMC0, which is the source clock of MMC0 (0000: XXTI, 0001: XUSBXTI, 0010: SCLK_HDMI27M, 0011: SCLK_USBPHY0, 0100: SCLK_USBPHY1, 0101: SCLK_HDMIPHY, 0110: SCLKMPPLL, 0111: SCLK_EPLL, 1000: SCLKVPLL, OTHERS: reserved)	0x0																						
MMC0_SEL	[3:0]	Control MUXMMC0, which is the source clock of MMC0 (0000: XXTI, 0001: XUSBXTI, 0010: SCLK_HDMI27M, 0011: SCLK_USBPHY0, 0100: SCLK_USBPHY1, 0101: SCLK_HDMIPHY, 0110: SCLKMPPLL, 0111: SCLK_EPLL, 1000: SCLKVPLL, OTHERS: reserved)	0x0																										
<table border="1"> <tr> <td>MMC0_RATIO</td><td>[3:0]</td><td>DIVMMC0 clock divider ratio, SCLK_MMC0 = MOUTMMC0 / (MMC0_RATIO + 1)</td><td></td></tr> </table>				MMC0_RATIO	[3:0]	DIVMMC0 clock divider ratio, SCLK_MMC0 = MOUTMMC0 / (MMC0_RATIO + 1)																							
MMC0_RATIO	[3:0]	DIVMMC0 clock divider ratio, SCLK_MMC0 = MOUTMMC0 / (MMC0_RATIO + 1)																											
<p>这是将输入MMC0的96M时钟分频成48M给MMC控制器用</p>																													

```

// software reset for all 复位主机SoC控制器，而不是复位SD卡
__REGb(HSMMC_BASE+SWRST_OFFSET) = 0x1;
Timeout = 1000; // Wait max 10 ms
while (__REGb(HSMMC_BASE+SWRST_OFFSET) & (1<<0)) {
    if (Timeout == 0) {
        return -1; // reset timeout
    }
    Timeout--;
    Delay_us(10);
}

```

再读一下复位寄存器看看是否设置成功

RSTALL	[0]	Software Reset For All This reset affects the entire Host Controller except for the card	0
--------	-----	---------------------------------------------------------------------------------------------	---

detection circuit. Register bits of type ROC, RW, RW1C, RWAC, HWInit are cleared to 0. During its initialization, the Host Driver sets this bit to 1 to reset the Host Controller. The Host Controller reset this bit to 0 if capabilities registers are valid and the Host Driver reads them. If this bit is set to 1, the SD card resets itself and must be reinitialized by the Host Driver. (RWAC)

1 = Reset
0 = Work

复位 soc 的 SD/MMC 卡控制器

因为我们不知道插入的 SD 卡是高速还是低速, 所以给 SD 卡设置 400K 时钟, 兼容所有 SD 卡, 然后与 SD 卡通信来探测 SD 卡是高速还是低速

```

// 上面设置的是SoC的SD控制器的时钟, 现在设置的是SD卡的时钟
Hsmmc_SetClock(400000); // 400k
__REGb(HSMMC_BASE+TIMEOUTCON_OFFSET) = 0xe; // 最大超时时间
__REGb(HSMMC_BASE+HOSTCTL_OFFSET) &= ~(1<<2); // 正常速度模式
// 清除正常中断状态标志
__REGw(HSMMC_BASE+NORINTSTS_OFFSET) = __REGw(HSMMC_BASE+NORINTSTS_OFFSET)
// 清除错误中断状态标志
__REGw(HSMMC_BASE+ERRINTSTS_OFFSET) = __REGw(HSMMC_BASE+ERRINTSTS_OFFSET)
__REGw(HSMMC_BASE+NORINTSTSEN_OFFSET) = 0xffff; // [14:0]中断状态使能
__REGw(HSMMC_BASE+ERRINTSTSEN_OFFSET) = 0x3ff; // [9:0]错误中断状态使能
__REGw(HSMMC_BASE+NORINTSIGEN_OFFSET) = 0xffff; // [14:0]中断信号使能
__REGw(HSMMC_BASE+ERRINTSIGEN_OFFSET) = 0x3ff; // [9:0]错误中断信号使能

```

static void Hsmmc_SetClock(uint32_t Clock)

```

{
    uint32_t Temp;
    uint32_t Timeout;
    uint32_t i;
    Hsmmc_ClockOn(0); // 关闭时钟
    Temp = __REG(HSMMC_BASE+CONTROL2_OFFSET);
    // Set SCLK_MMC(48M) from SYSCON as a clock source
    Temp |= (~(3<<4)) | (2<<4);
    Temp |= (1u<<31) | (1u<<30) | (1<<8);
    if (Clock <= 500000) {
        Temp &= ~(1<<14) | (1<<15);
        __REG(HSMMC_BASE+CONTROL3_OFFSET) = 0; // 如果 SD 卡时钟 <400K 选择这个
    } else {
        Temp |= ((1<<14) | (1<<15));
        __REG(HSMMC_BASE+CONTROL3_OFFSET) = (1u<<31) | (1<<23);
    }
    __REG(HSMMC_BASE+CONTROL2_OFFSET) = Temp;

    for (i=0; i<=8; i++) {
        if (Clock >= (4800000/(1<<i))) {
            break;
        }
    }
}
```

如果 SD 卡时钟 <400K 选择这个

最后将设置好的值写入 CONTROL2 寄存器

```

static void Hsmmc_ClockOn(uint8_t On)
{
    uint32_t Timeout;
    if (On) {
        __REGw(HSMMC_BASE+CLKCON_OFFSET) |= (1<<2); // sd时钟使能
        Timeout = 1000; // Wait max 10 ms
        while (!(__REGw(HSMMC_BASE+CLKCON_OFFSET) & (1<<3))) {
            // 等待SD输出时钟稳定
            if (Timeout == 0) {
                return;
            }
            Timeout--;
            Delay_us(10);
        }
    } else {
        __REGw(HSMMC_BASE+CLKCON_OFFSET) &= ~(1<<2); // sd时钟禁止
    }
}

```

查看时钟使能是否成功

SELBASE CLK	[5:4]	Base Clock Source Select 00 or 01 = HCLK 10 = SCLK_MMC0~3 (from SYSCON), 11 = Reserved
-------------	-------	-------------------------------------------------------------------------------------------------

ENSDCLK	[2]	SD Clock Enable The Host Controller stops SD SDCLK Frequency Select ch. Controller maintains the same stopped (Stop at SDCLK=0). State register is cleared, this 1 = Enables 0 = Disables
---------	-----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

Temp = ((1<<i) / 2) << 8; // clock div
Temp |= (1<<0); // Internal Clock Enable
__REGw(HSMMC_BASE+CLKCON_OFFSET) = Temp;
Timeout = 1000; // Wait max 10 ms
while (!(__REGw(HSMMC_BASE+CLKCON_OFFSET) & (1<<1))) {
    // 等待内部时钟振荡稳定
    if (Timeout == 0) {
        return;
    }
    Timeout--;
    Delay_us(10);
}

Hsmmc_ClockOn(1); // 使能时钟
}

```

```

// 从这里开始和SD卡通信，通信其实就是发命令然后收响应
Hsmmc_IssueCommand(CMD0, 0, 0, CMD_RESP_NONE); // 复位所有卡到空闲状态

CardType = UNUSABLE; // 卡类型初始化不可用
if (Hsmmc_IssueCommand(CMD8, 0x1aa, 0, CMD_RESP_R7)) { // 没有回复, MMC/sd v1.x/not card
    for (i=0; i<100; i++) {
        // CMD55 + CMD41 = ACMD41
        Hsmmc_IssueCommand(CMD55, 0, 0, CMD_RESP_R1);
        if (!Hsmmc_IssueCommand(CMD41, 0, 0, CMD_RESP_R3)) { // CMD41有回复说明为sd卡
            OCR = __REG(HSMMC_BASE+RSPREG0_OFFSET); // 获得回复的OCR(操作条件寄存器)值
            if (OCR & 0x80000000) { // 卡上电是否完成上电流程, 是否busy
                CardType = SD_V1; // 正确识别出sd v1.x卡
                Debug("SD card version 1.x is detected\r\n");
                break;
            }
        } else {
            // MMC卡识别
            Debug("MMC card is not supported\r\n");
            return -1;
        }
        Delay_us(1000);
    }
} else { // sd v2.0

```

```

static int32_t Hsmmc_IssueCommand(uint8_t Cmd, uint32_t Arg, uint8_t Data, uint8_t Response)
{
    uint32_t i;
    uint32_t Value;
    uint32_t ErrorState;
    // 检查CMD线是否准备好发送命令
    for (i=0; i<0x1000000; i++) {
        if (!__REG(HSMMC_BASE+PRNSTS_OFFSET) & (1<<0)) { 检查 CMD 线是否准备好
            break;
        }
    }
    if (i == 0x1000000) {
        Debug("CMD line time out, PRNSTS: %04x\r\n", __REG(HSMMC_BASE+PRNSTS_OFFSET));
        return -1; // 命令超时
    }

    // 检查DAT线是否准备好
    if (Response == CMD_RESP_R1B) { // R1b回复通过DAT0反馈忙信号
        for (i=0; i<0x1000000; i++) {
            if (!(__REG(HSMMC_BASE+PRNSTS_OFFSET) & (1<<1))) { 检查 dat 线是否准备好
                break;
            }
        }
        if (i == 0x1000000) {
            Debug("Data line time out, PRNSTS: %04x\r\n", __REG(HSMMC_BASE+PRNSTS_OFFSET));
            return -2;
        }
    }
}

```

CMDINHCMD	[0]	<p>Command Inhibit (CMD) (ROC)</p> <p>If this bit is 0, it indicates the CMD line is not in use and the Host Controller issues a SD Command using the CMD line.</p> <p>This bit is set immediately after the Command register (00Fh) is written. This bit is cleared if the command response is received.</p> <p>Even if the Command Inhibit (DAT) is set to 1, Commands using only the CMD line is issued if this bit is 0. Changing from 1 to 0 generates a Command</p> <p>Complete interrupt in the Normal Interrupt Status register. If the Host Controller cannot issue the command because of a command conflict error (Refer to Command CRC Error) or because of Command Not Issued By Auto CMD12 Error, this bit remains 1 and the Command Complete is not set. Status issuing Auto CMD12 is not read from this bit.</p> <p>1 = Cannot issue command 0 = Issues command using only CMD line</p>
-----------	-----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

__REG(HSMC_BASE+ARGUMENT_OFFSET) = Arg; // 写入命令参数

Value = (Cmd << 8); // command index
// CMD12可终止传输
if (Cmd == 0x12) {
    Value |= (0x3 << 6); // command type
}
if (Data) {
    Value |= (1 << 5); // 需使用DAT线作为传输等
}

switch (Response) {
case CMD_RESP_NONE:
    Value |= (0<<4) | (0<<3) | 0x0; // 没有回复, 不检查命令及CRC
    break;
case CMD_RESP_R1:
case CMD_RESP_R5:
case CMD_RESP_R6:
case CMD_RESP_R7:
    Value |= (1<<4) | (1<<3) | 0x2; // 检查回复中的命令, CRC
    break;
case CMD_RESP_R2:
    Value |= (0<<4) | (1<<3) | 0x1; // 回复长度为136位, 包含CRC
    break;
case CMD_RESP_R3:
case CMD_RESP_R4:
    Value |= (0<<4) | (0<<3) | 0x2; // 回复长度48位, 不包含命令及CRC
    break;
case CMD_RESP_R1B:
    Value |= (1<<4) | (1<<3) | 0x3; // 回复带忙信号, 会占用Data[0]线
    break;
default:
    break;
}

```

有些命令跟着参数，就放入这个寄存器

ARGUMENT	Bit	Description
ARGUMENT	[31:0]	Command Argument The SD Command Argument is specified as bit [39:8] of Command-Format in the SD Memory Card Physical Layer Specification.

经过运算得到一个 value 值，将 value 值放入 CMDREG0 寄存器

```
__REGw(HSMMC_BASE+CMDREG_OFFSET) = Value;  
  
ErrorState = Hsmmc_WaitForCommandDone();  
if (ErrorState) {  
    Debug("Command = %d\r\n", Cmd);  
}  
return ErrorState; // 命令发送出错  
}
```

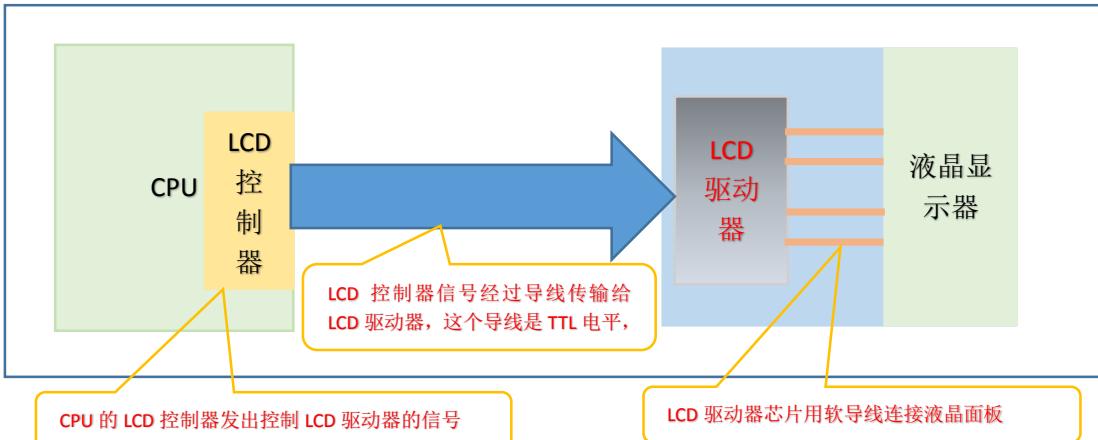
将我们上面说的 value 放入 CMDREG0 寄存器

等待 SD 卡完成它自己的程序，然后返回

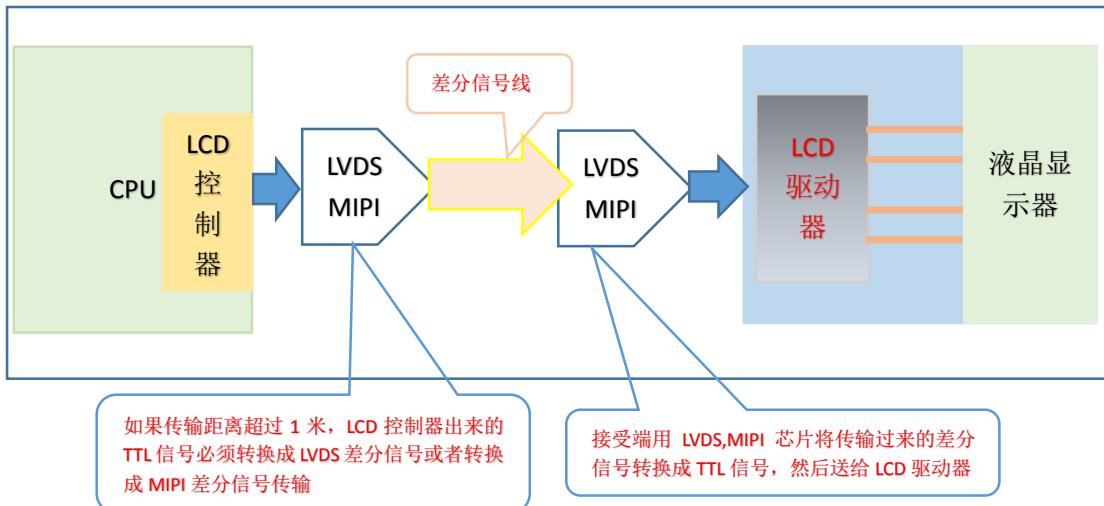
CMDREG0	0xEB00_000E	R/W	Command Register (Channel 0)
---------	-------------	-----	------------------------------

LCD 使用

CPU 里面的 SOC 如何控制 LCD

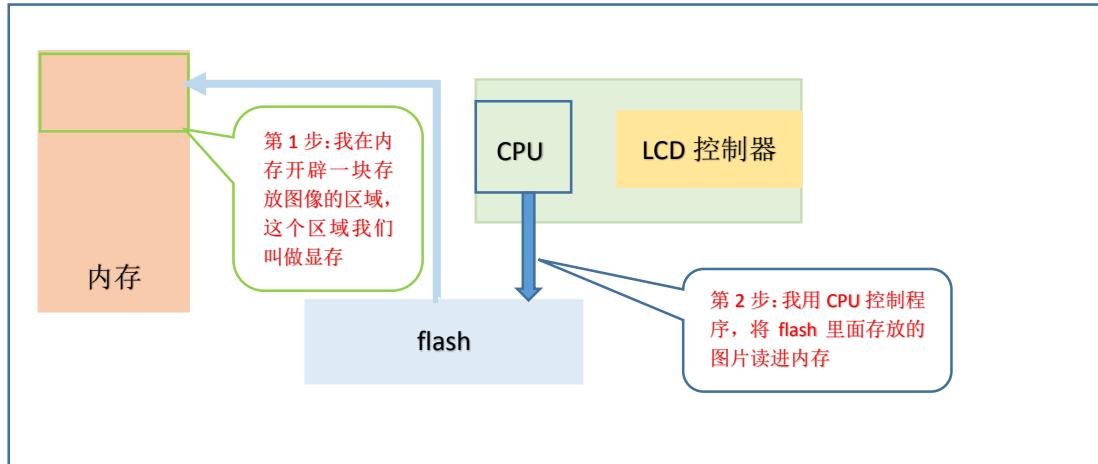


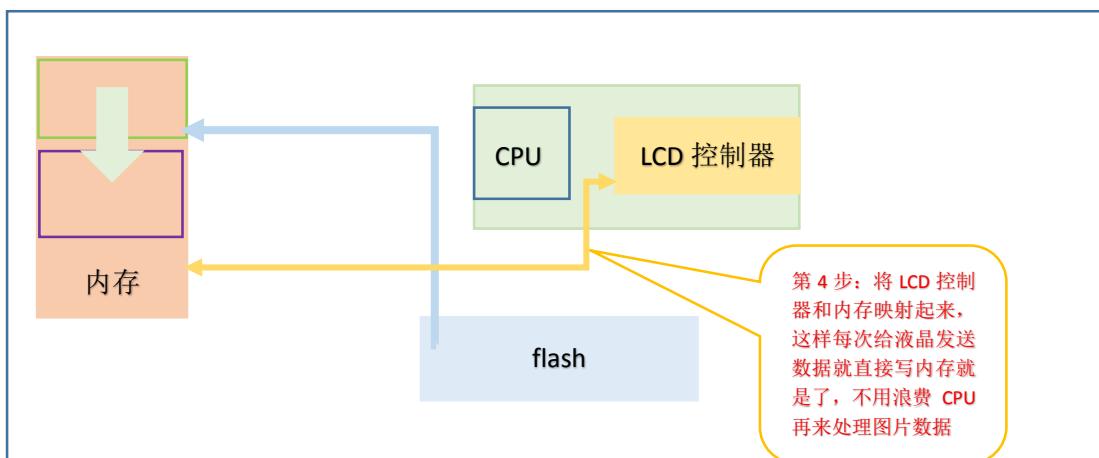
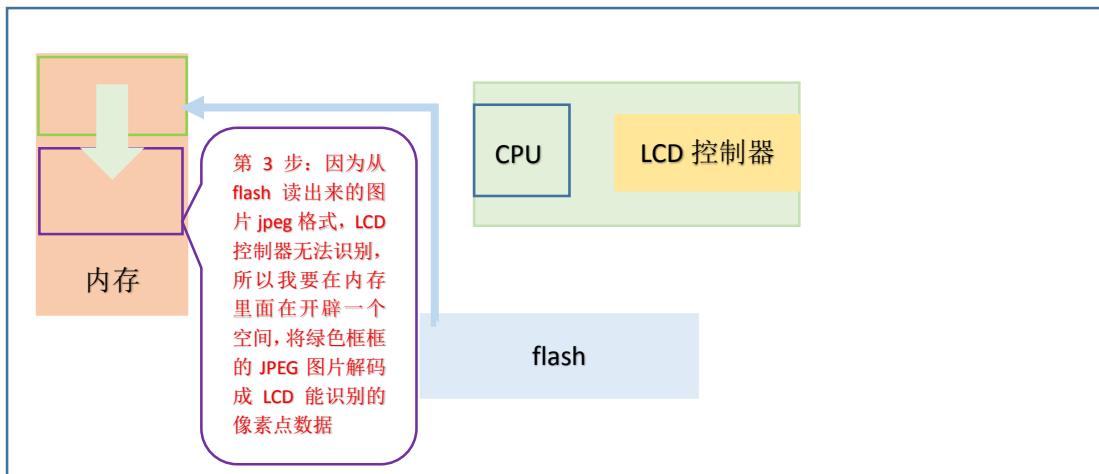
所以 LCD 的控制原理就是 CPU 的 LCD 控制器发出 TTL 控制信号给 LCD 驱动器，LCD 驱动器将接受到的 TTL 数字信号转换成模拟信号去触发液晶里面的每个像素点。



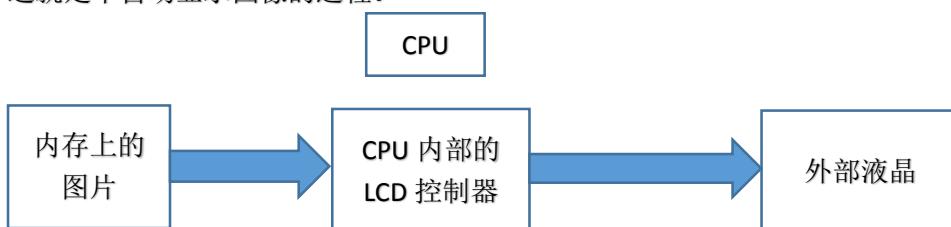
在长距离传输就必须外加芯片来解决 TTL 信号传输的问题，但是两边的信号使用本质还是 TTL

那么一幅图片是怎么进入 LCD 控制器的呢？





这就是个自动显示图像的过程。



这个自动图像显示就是将图像数据写入内存，然后内存自动发给 LCD 控制器，然后外部液晶显示，这个过程中 CPU 没有参与任何和图像有关的活动，类似 STM32 的 DMA 功能。这样 CPU 就可以去做其它的事情，不用在显示这里浪费时间

RGB 接口详解

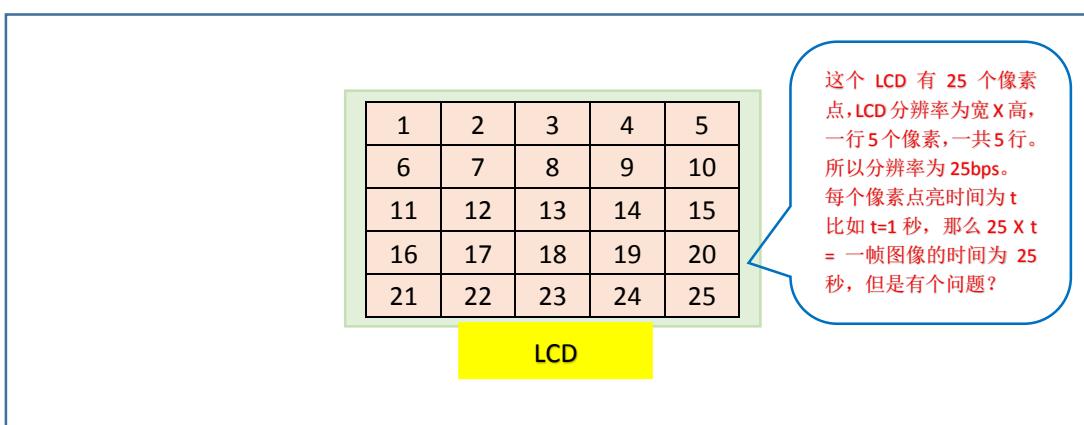
VD0~VD23 是 LCD 的 24 根数据线，用来传输图像数据的

AA9	XvVD0/SYSD0/VEND0/GPF0_4
AB9	XvVD1/SYSD1/VEND1/GPF0_5
AB8	XvVD2/SYSD2/VEND2/GPF0_6
AB7	XvVD3/SYSD3/VEND3/GPF0_7
Y9	XvVD4/SYSD4/VEND4/GPF1_0
AB6	XvVD5/SYSD5/VEND5/GPF1_1
AE7	XvVD6/SYSD6/VEND6/GPF1_2
AC9	XvVD7/SYSD7/VEND7/GPF1_3
AA8	XvVD8/SYSD8/V656D0/GPF1_4
W9	XvVD9/SYSD9/V656D1/GPF1_5
AE6	XvVD10/SYSD10/V656D2/GPF1_6
AC8	XvVD11/SYSD11/V656D3/GPF1_7
Y8	XvVD12/SYSD12/V656D4/GPF2_0
AC7	XvVD13/SYSD13/V656D5/GPF2_1
AD6	XvVD14/SYSD14/V656D6/GPF2_2
AE5	XvVD15/SYSD15/V656D7/GPF2_3
AD7	XvVD16/SYSD16/GPF2_4
AA7	XvVD17/SYSD17/GPF2_5
AD5	XvVD18/SYSD18/GPF2_6
AA6	XvVD19/SYSD19/GPF2_7
AB5	XvVD20/SYSD20/GPF3_0
AC5	XvVD21/SYSD21/GPF3_1
AC6	XvVD22/SYSD22/GPF3_2
Y7	XvVD23/SYSD23/V656_CLK/GPF3_3
W8	

帧概念

帧就是 1 秒钟在显示器显示几幅图像，一般是 1 秒钟 24 帧图像，也就是显示 24 副图像，显示器看起人物就是流畅运动。

显示器显示一帧图像的原理



这里 $t=1$ 秒是我的假设，其实 LCD 像素间切换的时间是按 us 或者 ns 进行的。

所以像素切换时间就决定了你的显示器显示画面的流畅度

我 LCD 是一个点一个点的点亮，应该说每个点时间是一样的，一幅图应该是 25 秒，但是这里问题是出现在，每一行里面一行里面的像素点是按照 1 秒间隔来点亮

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

LCD

但是换行的时候不是 1 秒，比如我点亮 5 号像素点了，突然换到第二行第 6 个像素点的时候不是 1 秒，所以行的转换不是同步的。所以 25 秒显示一幅图是不成立的

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

LCD

现实情况是下面这样

每一行像素显示都有 HBPD 时间

HBPD

和 HFPD 时间

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

LCD

HBPD 时间就是让 LCD 控制器准备时间，然后一次性将一行的像素点全部点亮

HBPD

HFPD

一行像素点点亮后，换到第二行，也要先休息 HBPD 个时间，然后才能一次性将第二行点亮

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

HFPD

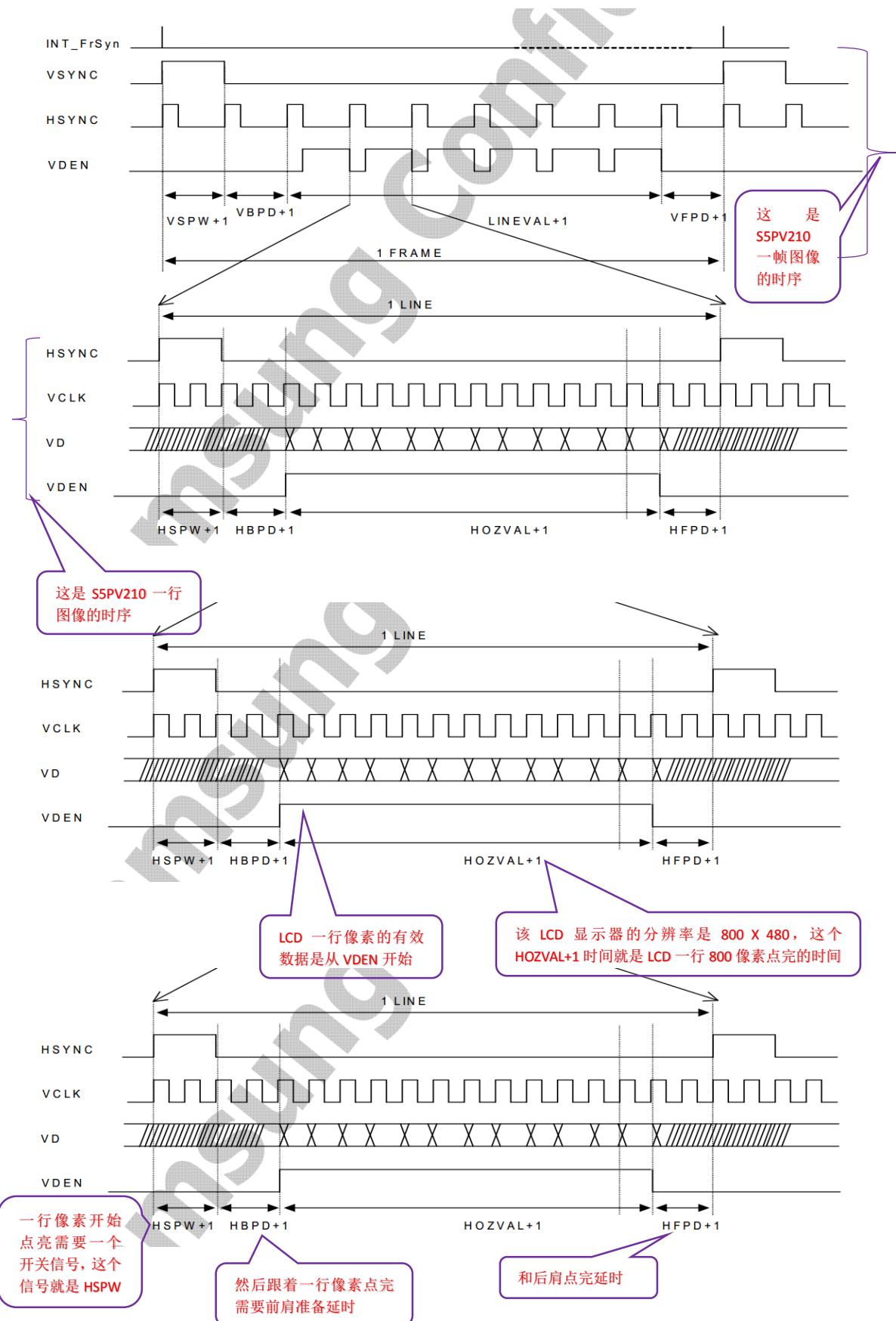
一行像素点显示完后有一个 LCD 控制器休息时间，这个时间就是 HFPD 时间

第二行点完后也要休息 HFPD 时间，后面几行也是一样的

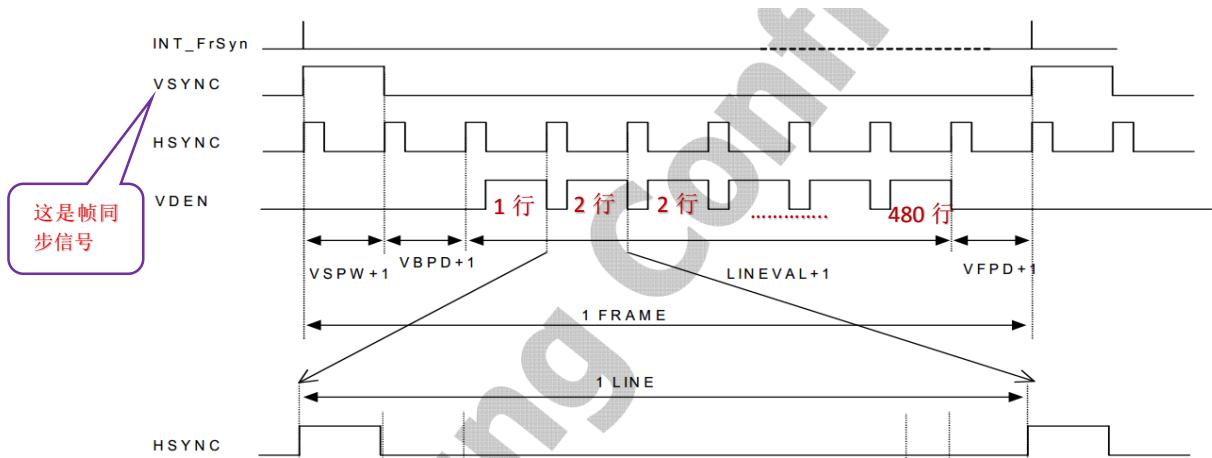
其实 HBPD 和 HFPD 就是时序的延时程序时间

HBPD：叫水平同步信号前肩

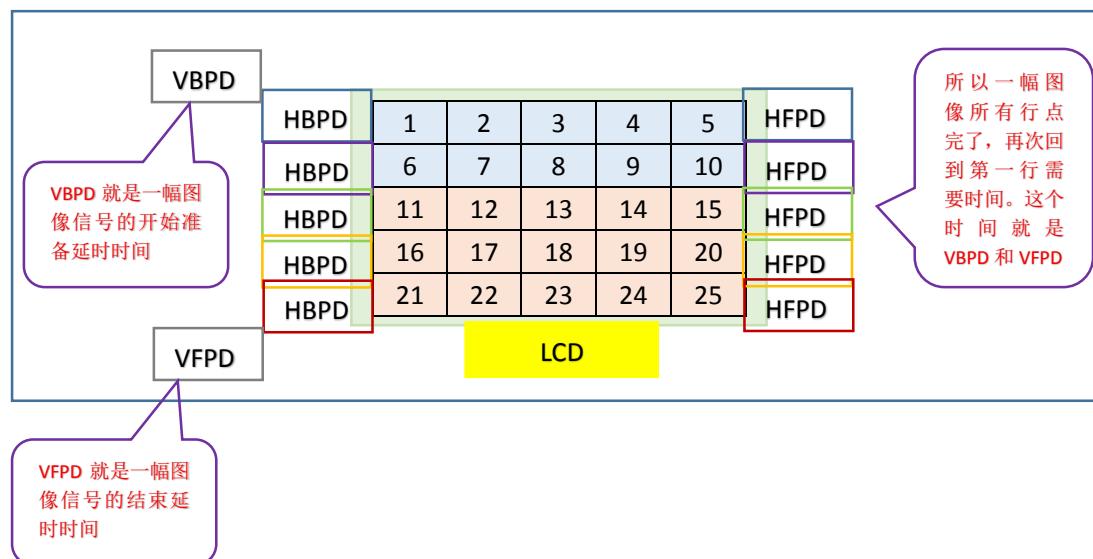
HFPD：叫水平同步信号后肩



HSYNC 信号就是告诉 LCD 驱动器，这个信号是我 cpu 的 LCD 控制器发送的，要求 LCD 驱动器准备接受一行像素的数据。数据包括(HBPD 开始延时->像素数据->HFPD 行结束延时).



帧 VSPW 和 VBPD, VFPD 其实和行信号的 HBPD, HFPD 是一个意思，只是 VSPW, VBPD 和 VFPD 针对的是一幅图像的信号，看下面分析



下面查看 LCD 型号的数据手册

3.3.3. Timing

LCD 一行多少个像素？这里是 800 个像素

这个时间单位 DCLK 和像素时钟有关，我这里是 33M 像素时钟，所以 1 个 DCLK 是 30ns

Item	Symbol	Values			Unit
		Min.	Typ.	Max.	
Horizontal Display Area	thd	-	800	-	DCLK
DCLK Frequency	像素时钟	fclk	26.4	33.3	46.8
One Horizontal Line	th	862	1056	1200	DCLK
HS pulse width	HSPW	thpw	1	-	40
HS Blanking	HBPD	thb	46	46	DCLK
HS Front Porch	HFPD	thfp	16	210	354

Item	Symbol	Values			Unit	
		Min.	Typ.	Max.		
Vertical Display Area	tvd	-	480	-	TH	
VS period time	tv	510	525	650	TH	
VS pulse width	VSPW	tvpw	1	-	20	TH
VS Blanking	VBPD	tvb	23	23	23	TH
VS Front Porch	VFPD	tvfp	7	22	147	TH

像素间距

No.	Item	Specification
1	LCD size	7.0 inch(Diagonal)
2	Driver element	a-Si TFT active matrix
3	Resolution	800 × 3(RGB) × 480
4	Display mode	Normally White, Transmissive
5	Dot pitch	0.0642(W) × 0.1790(H) mm
6	Active area	154.08(W) × 85.92(H) mm
7	Module size	164.9(W) × 100.0(H) × 5.7(D) mm

像素间距影响的是我们观看屏幕的最佳距离，间距越大，我们离屏幕越近那么显示效果越花，如果离屏幕越远显示效果越好，反之亦然

所以像素间距大时候远距离看

像素间距小适合近距离看

像素深度 bpp:

像素深度分为，1位，8位，16位，24位，32位

1位就是黑白

8位就可以从黑到白显示灰色，就是灰度

16位，24位，32位都是彩色，位数越高颜色越鲜艳

16位就是 RGB565: 5位红色，6位绿色，5位蓝色

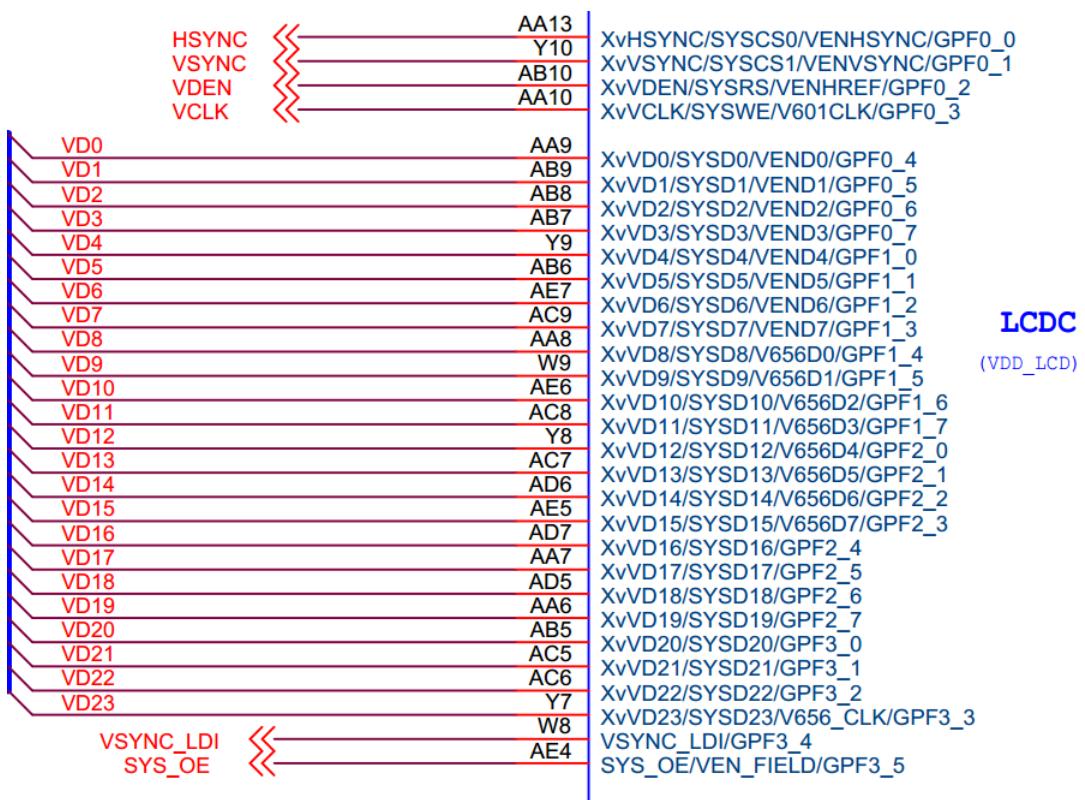
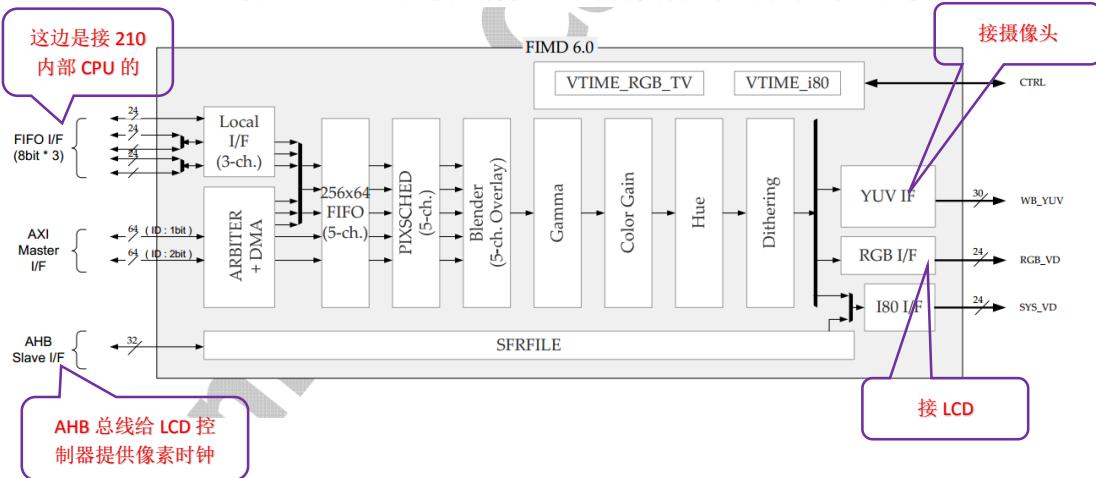
24位就是 RGB888: 8位红色，8位绿色，8位蓝色

32位就是前24位红绿蓝 RGB888，然后最后有个8位表示颜色的透明度，所以叫做 ARGB

S5PV210 的 LCD 控制器使用

S5PV210LCD 控制器是 FIMD 标准

FIMD 标准不仅可以接 LCD，还可以接摄像头。和图像有关联的设备都可以接



LCDC
(VDD_LCD)

根据原理图 LCD 的， HSYNC , VSYNC, VDEN , VCLK 和 RGB 等引脚都是接在 GPF 管脚组上的，所以我们要先设置 GPIOF 组的管脚复用功能为 LCD 模式

GPF0CON	Bit	Description
GPF0CON[7]	[31:28]	0000 = Input 0001 = Output
		0010 = LCD_VD[3]
		0011 = SYS_VD[3]
		0100 = VEN_DATA[3]
		0101 ~ 1110 = Reserved
		1111 = GPF0_INT[7]

根据数据手册我们要设置每个 GPF0 为 0010，才能成为 LCD 管脚模式

```

#define GPF0CON      (*(volatile unsigned long *)0xE0200120)
#define GPF1CON      (*(volatile unsigned long *)0xE0200140)
#define GPF2CON      (*(volatile unsigned long *)0xE0200160)
#define GPF3CON      (*(volatile unsigned long *)0xE0200180)

```

先映射 GPIO 管脚寄存器地址

```

void lcd_init(void)
{
    // 配置引脚用于LCD功能
    GPF0CON = 0x22222222;
    GPF1CON = 0x22222222;
    GPF2CON = 0x22222222;
    GPF3CON = 0x22222222;           全部设置为 LCD 管脚模式

    // 打开背光 GPD0_0 (PWMTOUT0)
    GPD0CON &= ~(0xf<<0);
    GPD0CON |= (1<<0);           // output mode
    GPDODAT &= ~(1<<0);         // output 0 to
                                  LCD 背光自己去设置 IO

```

```
#define DISPLAY_CONTROL (*(volatile unsigned long *)0xE010_7008)
```

设置显示控制寄存器

3.7.11.1 Miscellaneous SFRs (DISPLAY_CONTROL, R/W, Address = 0xE010_7008)

DISPLAY_CONTROL	Bit	Description
Reserved	[31:2]	Reserved
DISPLAY_PATH_SEL	[1:0]	Display path selection <ul style="list-style-type: none"> 00: RGB=--- I80=FIMD ITU=FIMD 01: RGB=--- I80=--- ITU=FIMD 10: RGB=FIMD I80=FIMD ITU=FIMD 11: RGB=FIMD I80=FIMD ITU=FIMD

```
// 10: RGB=FIMD I80=FIMD ITU=FIMD
```

```
DISPLAY_CONTROL = 2<<0;
```

设置 RGB 或者 I80 模式

```
#define VIDCON0      (*(volatile unsigned long *)0xF8000000)
```

设置控制寄存器

VIDOUT	[28:26]	Determines the output format of Video Controller. <ul style="list-style-type: none"> 000 = RGB interface 001 = Reserved 010 = Indirect I80 interface for LDIO 011 = Indirect I80 interface for LDI1 100 = WB interface and RGB interface 101 = Reserved 110 = WB Interface and i80 interface for LDIO 111 = WB Interface and i80 interface for LDI1
--------	---------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

选择 RGB 接口

RGSPSEL	[18]	Selects display mode (VIDOUT[1:0] == 2'b00). 0 = RGB parallel format 1 = RGB serial format
		Selects the display mode (VIDOUT[1:0] != 2'b00). 0 = RGB parallel format

CLKSEL_F	[2]	Selects the video clock source. 0 = HCLK 1 = SCLK_FIMD
		HCLK is the bus clock, whereas SCLK_FIMD is the special clock for display controller. For more information, refer to Chapter, "02.03 CLOCK CONTROLLER".

CPU 的 HCLK 是 166M,

```
// bit[26~28]: 使用RGB接口
// bit[18]: RGB 并行
// bit[2]: 选择时钟源为HCLK_DSYS=166MHz
VIDCON0 &= ~( (3<<26) | (1<<18) | (1<<2) );
```

ENVID	[1]	Enables/ disables video output and logic immediately. 0 = Disables the video output and display control signal. 1 = Enables the video output and display control signal.
ENVID_F	[0]	Enables/ disables video output and logic at current frame end 0 = Disables the video output and display control signal 1 = Enables the video output and display control signal. * If this bit is set to "on" and "off", then "H" is read and video controller is enabled until the end of current frame.

```
// bit[1]: 使能lcd控制器
// bit[0]: 当前帧结束后使能lcd控制器
VIDCON0 |= ( (1<<0) | (1<<1) );
```

CLKVAL_F	[13:6]	Determines the rates of VCLK and CLKVAL[7:0]. VCLK = HCLK / (CLKVAL+1), where CLKVAL >= 1 Notes. 1. The maximum frequency of VCLK is 100Mhz(pad:50pf). 2. CLKSEL_F register selects Video Clock Source.
----------	--------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

DCLK Frequency	像素时钟	fclk	26.4	33.3	46.8	MHz
----------------	------	------	------	------	------	-----

这是 LCD 模组的数据，最大像素时钟 46.8M，我们选择标准的 33.3 就是 33M

所以我们 CLKVAL 选择 4，根据公式 $VCLK = HCLK / (CLKVAL + 1) = 166M / (4 + 1) = 33.2M$

CLKDIR	[4]	Selects the clock source as direct or divide using CLKVAL_F register. 0 = Direct clock (frequency of VCLK = frequency of Clock source) 1 = Divided by CLKVAL_F
--------	-----	----------------------------------------------------------------------------------------------------------------------------------------------------------------------

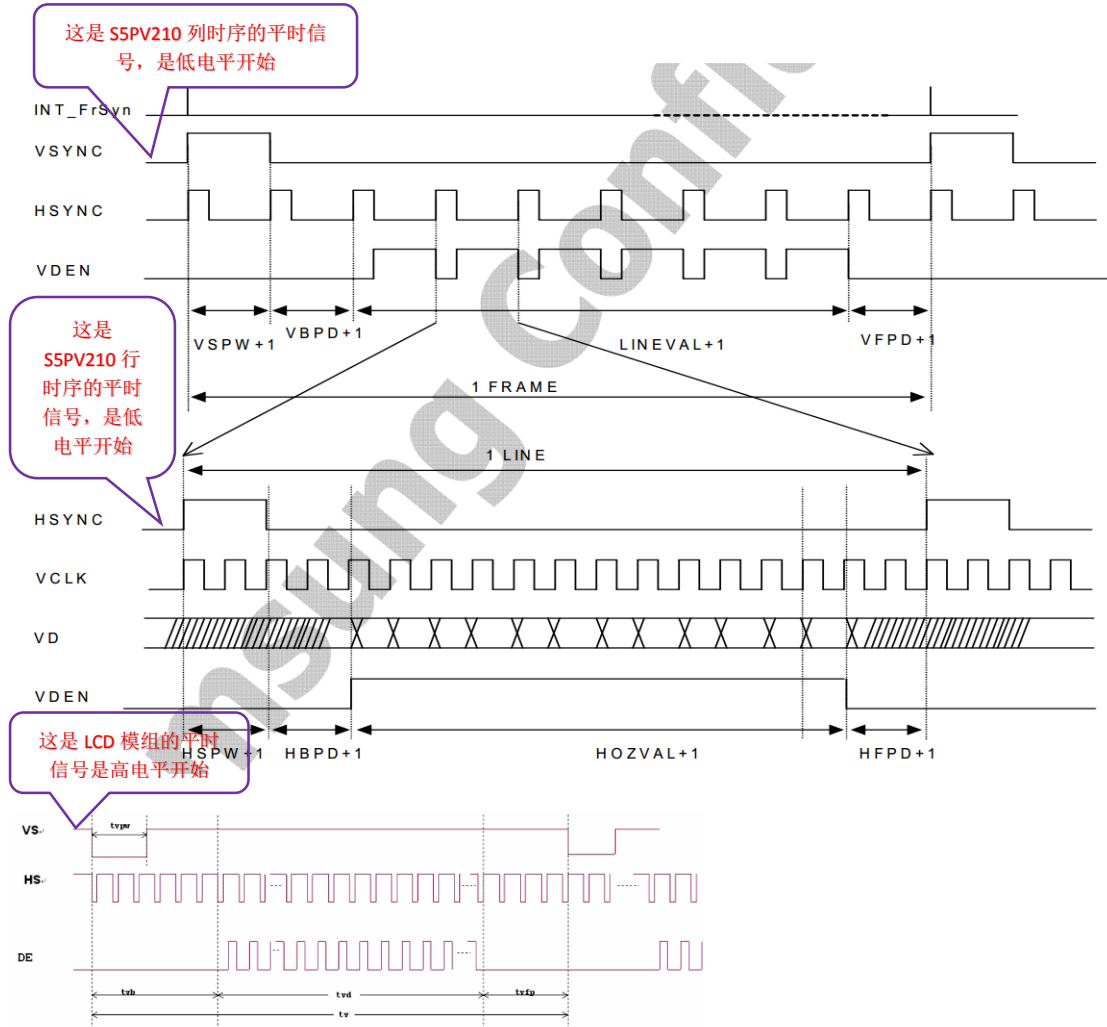
```

// bit[6]:选择需要分频
// bit[6~13]:分频系数为5, 即VCLK = 166M / (4+1) = 33M
VIDCON0 |= 4<<6 | 1<<4;

```

```
#define VIDCON1 (*(volatile unsigned long *) 0xF8000004)
```

IHSYNC	[6]	Specifies the HSYNC pulse polarity. 0 = Normal 1 = Inverted
IVSYNC	[5]	Specifies the VSYNC pulse polarity. 0 = Normal 1 = Inverted



所以为了 S5PV210 和 LCD 模组的行列时序信号一致，我们这里设置反向这样 S5PV210 和 LCD 模组的行列信号平时都处于高电平状态

```

// H43-HSD043I9W1.pdf(p13) 时序图: VSYNC和HSYNC都是低脉冲
// s5pv210芯片手册(p1207) 时序图: VSYNC和HSYNC都是高脉冲有效, 所以需要反转
VIDCON1 |= 1<<5 | 1<<6;

```

```
#define VIDTCON0      (*(volatile unsigned long *)0xF8000010)
#define VIDTCON1      (*(volatile unsigned long *)0xF8000014)
```

设置 VBPD, VFPD, VSPW 时序时间

1.5.2.5 Video Time Control 0 Register (VIDTCON0, R/W, Address = 0xF800_0010)

VIDTCON0	Bit	Description
VBPDE	[31:24]	Vertical back porch specifies the number of inactive lines at the start of a frame after vertical synchronization period. (Only for even field of YVU interface)
VBPD	[23:16]	Vertical back porch specifies the number of inactive lines at the start of a frame after vertical synchronization period.
VFPD	[15:8]	Vertical front porch specifies the number of inactive lines at the end of a frame before vertical synchronization period.
VSPW	[7:0]	Vertical sync pulse width determines the high-level width of VSYNC pulse by counting the number of inactive lines.

LCD 模组的时序

Item	Symbol	Values			Unit
		Min.	Typ.	Max.	
Vertical Display Area	tvd	-	480	-	TH
VS period time	tv	510	525	650	TH
VS pulse width	VSPW	1	-	20	TH
VS Blanking	VBPD	23	23	23	TH
VS Front Porch	VFPD	7	22	147	TH

// 设置时序

```
VIDTCON0 = VBPD<<16 | VFPD<<8 | VSPW<<0;
VIDTCON1 = HBPD<<16 | HFPD<<8 | HSPW<<0;
#define VIDTCON2      (*(volatile unsigned long *)0xF8000018)
```

设置显示行列像素个数

LCD 列是 480 的像素这里就写 479

1.5.2.7 Video Time Control 2 Register (VIDTCON2, R/W, Address = 0xF800_0018)

VIDTCON2	Bit	Description
LINEVAL	[21:11]	Determines the vertical size of display. In the Interlace mode, (LINEVAL + 1) should be even.
HOZVAL	[10:0]	Determines the horizontal size of display.

LCD 行是 800 的像素这里就写 799

// 设置长宽(物理屏幕)

```
VIDTCON2 = (LINEVAL << 11) | (HOZVAL << 0);
```

#define WINCON0	(*(volatile unsigned long *) 0xF8000020)
#define WINCON2	(*(volatile unsigned long *) 0xF8000028)
WINCON0	0xF800_0020
WINCON1	0xF800_0024
WINCON2	0xF800_0028
WINCON3	0xF800_002C
WINCON4	0xF800_0030

S5PV210 支持 5 个虚拟屏幕，我们这里只用了 1 个虚拟屏幕 wincon0

因为我们只用了 window1 窗口，所以使能 WINCON1 的功能

ENWIN_F	[0]	Enables/ disables video output and logic immediately. 0 = Disables the video output and video control signal. 1 = Enables the video output and video control signal.
---------	-----	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

BPPMODE_F	[5:2]	Selects the Bits Per Pixel (BPP) mode for Window image. 0000 = 1 bpp 0001 = 2 bpp 0010 = 4 bpp 0011 = 8 bpp (palletized) 0100 = 8 bpp (non-palletized, A: 1-R:2-G:3-B:2) 0101 = 16 bpp (non-palletized, R:5-G:6-B:5) 0110 = 16 bpp (non-palletized, A:1-R:5-G:5-B:5) 0111 = 16 bpp (non-palletized, I :1-R:5-G:5-B:5) 1000 = Unpacked 18 bpp (non-palletized, R:6-G:6-B:6) 1001 = Unpacked 18 bpp (non-palletized, A:1-R:6-G:6-B:5) 1010 = Unpacked 19 bpp (non-palletized, A:1-R:6-G:6-B:6) 1011 = Unpacked 24 bpp (non-palletized R:8-G:8-B:8) 1100 = Unpacked 24 bpp (non-palletized A:1-R:8-G:8-B:7) *1101 = Unpacked 25 bpp (non-palletized A:1-R:8-G:8-B:8) *1110 = Unpacked 13 bpp (non-palletized A:1-R:4-G:4-B:4) 1111 = Unpacked 15 bpp (non-palletized R:5-G:5-B:5)
-----------	-------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

每个 WINCON0~5 都是这么设置

WSWP_F	[15]	Specifies the Word swap control bit. 0 = Swap Disable 1 = Swap Enable Note: It should be 0 when ENLOCAL is 1.
--------	------	------------------------------------------------------------------------------------------------------------------------

```
// 设置window0
// bit[0]:使能
// bit[2^5]:24bpp (RGB888)
WINCON0 |= 1<<0;
WINCON0 &= ~(0xf << 2);
WINCON0 |= (0xB<<2) | (1<<15);
```

清空 bpp 寄存器

再设置 bpp

```

#define VIDOSD0A      (*(volatile unsigned long *) 0xF8000040)
#define VIDOSD0B      (*(volatile unsigned long *) 0xF8000044)
#define VIDOSD0C      (*(volatile unsigned long *) 0xF8000048)

```

设置虚拟显示

一般都设置得和我们屏幕分辨率一样大，屏幕左上角 x, y 都为 0

VIDOSD0A	Bit	Description
OSD_LeftTopX_F	[21:11]	Specifies the horizontal screen coordinate for left top pixel of OSD image.
OSD_LeftTopY_F	[10:0]	Specifies the vertical screen coordinate for left top pixel of OSD image (For interlace TV output, this value must be set to half of the original screen y coordinate. The original screen y coordinate must be even.)

一般都设置得和我们屏幕分辨率一样大，屏幕右下角 x=799, y=479

VIDOSD0B	Bit	Description
OSD_RightBotX_F	[21:11]	Specifies the horizontal screen coordinate for right bottom pixel of OSD image.
OSD_RightBotY_F	[10:0]	Specifies the vertical screen coordinate for right bottom pixel of OSD image. (For interlace TV output, this value must be set to half of the original screen y coordinate. The original screen y coordinate must be odd value.)

VIDOSD0C	Bit	Description
Reserved	[25:24]	Reserved (should be 0)
OSDSIZE	[23:0]	Specifies the Window Size For example, Height * Width (Number of Word)

```

#define LeftTopX      0
#define LeftTopY      0
#define RightBotX    799
#define RightBotY    479

```

设置屏幕的横向尺寸和纵向尺寸，
其实就是和分辨率一样，横向 800，
纵向 480，这里不要减 1 哈，是多
少就是多少

```

// 设置window0的上下左右
// 设置的是显存空间的大小
VIDOSD0A = (LeftTopX<<11) | (LeftTopY << 0);
VIDOSD0B = (RightBotX<<11) | (RightBotY << 0);
VIDOSD0C = (LINEVAL + 1) * (HOZVAL + 1);

```

这里设置的其实就是虚拟屏幕就是显存的大小，其实你可以设置超出屏幕大小的范围

```

// 设置window0的上下左右
// 设置的是显存空间的大小
VIDOSD0A = (LeftTopX << 11) | (LeftTopY << 0);
VIDOSD0B = (1920<<11) | (1080 << 0);
VIDOSD0C = (LINEVAL + 1) * (HOZVAL + 1);

```

比如你设置成 1920*1080 的显存，
但是你屏幕是 800*480，那么你就
只能看到一半的图像

```
#define VIDW00ADD0B0      (*(volatile unsigned long *) 0xF80000A0)
#define VIDW00ADD1B0      (*(volatile unsigned long *) 0xF80000D0)
```

上面设置了显存大小，现在要给显存找一个内存的首地址，这样才好操作内存显示

VIDW00ADD0B0	0xF800_00A0	R/W	Specifies register,
VIDW00ADD0B1	0xF800_00A4	R/W	Specifies register,
VIDW00ADD0B2	0xF800_20A0	R/W	Specifies register,
VIDW01ADD0B0	0xF800_00A8	R/W	Specifies register.

// 设置fb的地址

VIDW00ADD0B0 = 0x23000000;

VIDW00ADD1B0 = (((HOZVAL + 1)*4 + 0) * (LINEVAL + 1)) & (0xffffffff);

这个内存的首地址可以自己随便设置，只要在内存的合理范围内就可以了

这个就是将屏幕的大小写入显存的结束地址

最后我们就在显存的开始地址到结束地址之间放图像数据

// 使能channel 0传输数据
SHADOWCON = 0x1;

Window1 窗口设置了，我们还要使能 window1 窗口才能用，这里就是使能

现在开始刷 LCD 屏幕

```
root@ubuntu:/home/xiang/SSPV210/nous_drtver/lcd# ls
lcd.c led.c link.lds main.c main.h Makefile mkv210_image.c s5pv210.h sdram_init.S start.S write2sd

1 led.bin: start.o led.o sdram_init.o lcd.o main.o
2 | arm-linux-ld -Tlink.lds -o led.elf $^
3 | arm-linux-objcopy -O binary led.elf led.bin
4 | arm-linux-objdump -D led.elf > led_elf.dis
5 | gcc mkv210_image.c -o mkx210
6 | ./mkx210 led.bin 210.bin

7 %.o : %.S
8 | arm-linux-gcc -o $@ $< -c -nostdlib

9 %.o : %.c
10 | arm-linux-gcc -o $@ $< -c -nostdlib

11 clean:
12 | rm *.o *.elf *.bin *.dis mkx210 -f
```

```
// µÚ4²%£°³öÈ%»`ddr
bl sram_asm_init
```

老规矩先在 Start.S 里面初始化 DDR

```
.global sram_asm_init
```

```
sram_asm_init:
    ldr r0, =_start
```

```
run_on_dram:
    ldr pc, =main
```

然后在 start.S 最后一段执行 C 文件的 main 主函数

```
#include "main.h"
int main(void)
{
    lcd_test();
    return 0;
}
```

main 主函数在 main.c 文件里面

```
void lcd_test(void)
{
    lcd_init();

    while (1)
    {
        lcd_draw_background(RED);
        delay();

        lcd_draw_background(GREEN);
        delay();

        lcd_draw_background(BLUE);
        delay();
    }
}
```

调用 LCD.c 文件里面的初始化函数

```
// 初始化LCD
static void lcd_init(void)
{
    // 配置引脚用于LCD功能
    GPF0CON = 0x22222222;
    GPF1CON = 0x22222222;
    GPF2CON = 0x22222222;
```

这个初始化函数我们前面已经实现了

```

// FB地址
#define FB_ADDR (0x23000000)
#define ROW (480)
#define COL (800)

u32 *pfb = (u32 *)FB_ADDR;

// 便能channel 0传输数据
SHADOWCON = 0x1;

}

// 在像素点(x, y)处填充为color颜色
static inline void lcd_draw_pixel(u32 x, u32 y, u32 color)
{
    *(pfb + COL * y + x) = color;
}

```

LCD 映射的内存地址我们前面也已经实现了

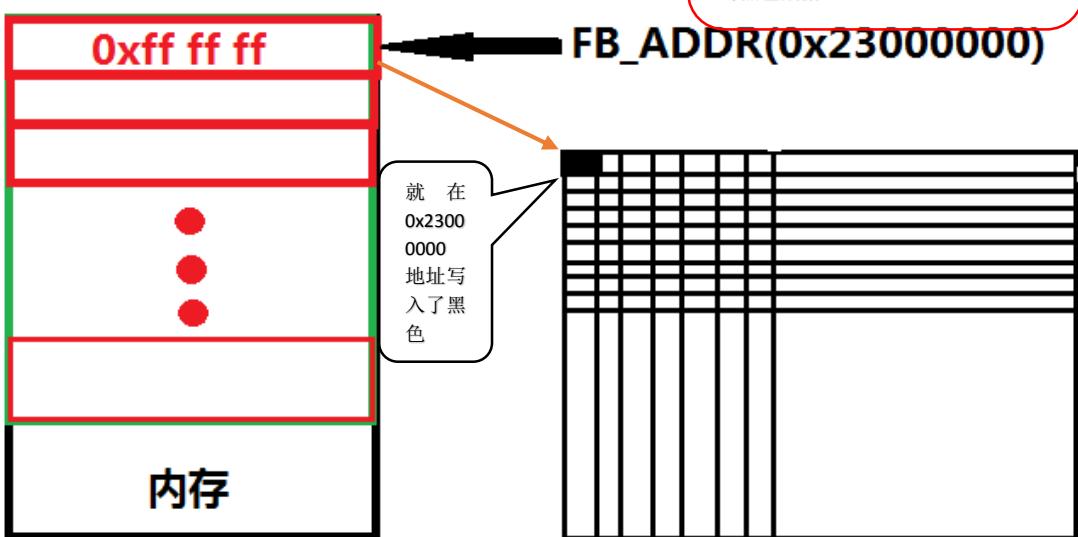
我用一个指针指向内存地址，方便操作

屏幕列像素个数

屏幕行像素个数

Window1 窗口我们前面也使能了

现在就是向屏幕固定座标点亮一个点

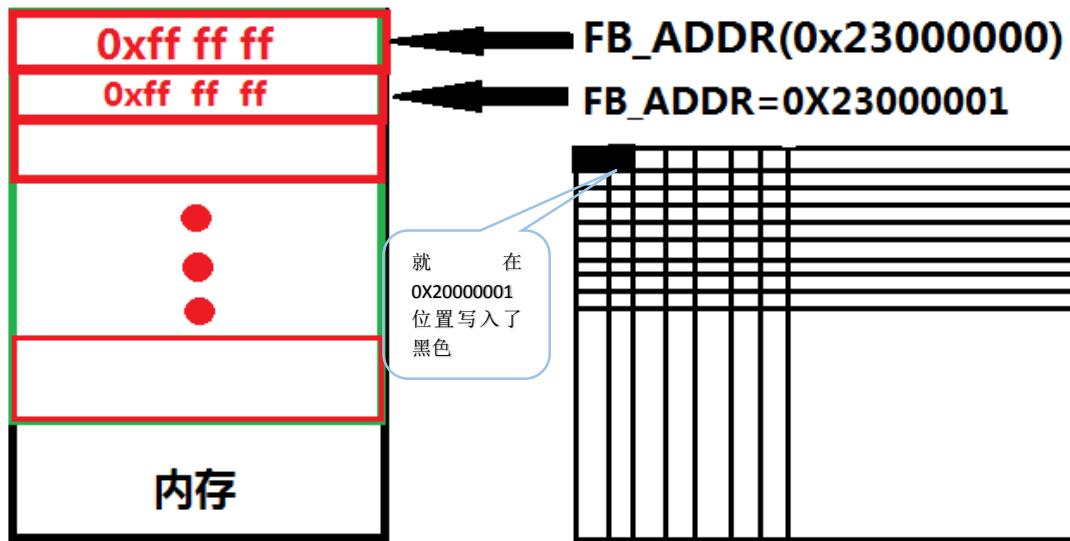


```

    // 便能channel 0传输数据
    SHADOWCON = 0x1;
}

// 在像素点(x, y)处填充为color颜色
static inline void lcd_draw_pixel(u32 x, u32 y, u32 color)
{
    *(pfb + COL * y + x) = color;
}

```



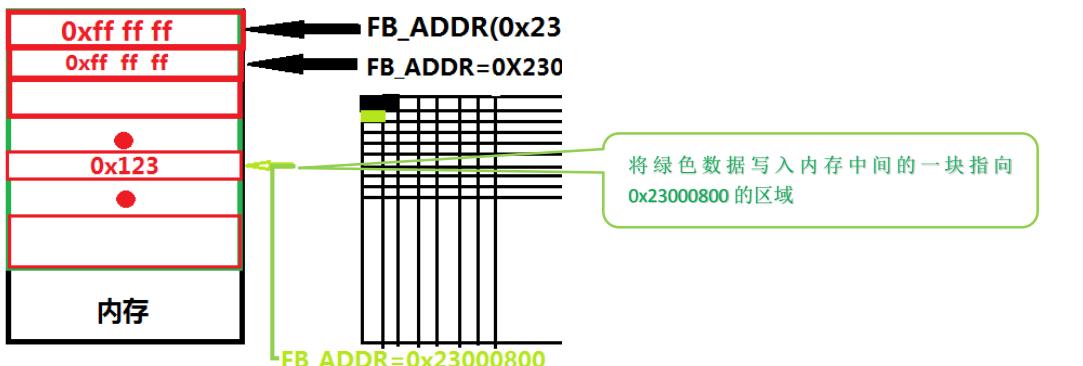
如果我要写第二行怎么办？

```

    // 便能channel 0传输数据
    SHADOWCON = 0x1;
}

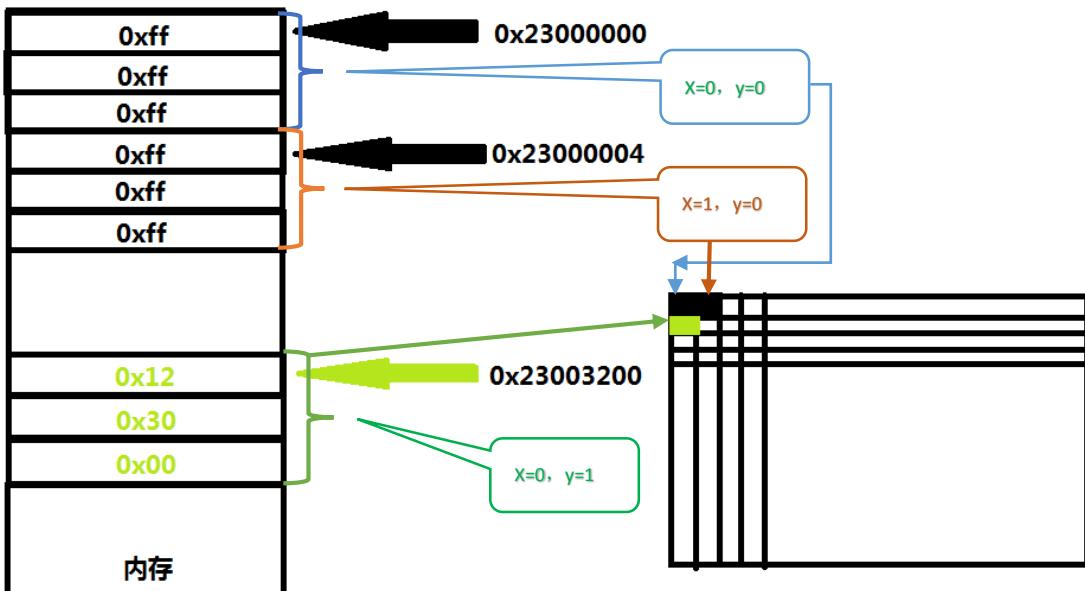
// 在像素点(x, y)处填充为color颜色
static inline void lcd_draw_pixel(u32 x, u32 y, u32 color)
{
    *(pfb + COL * y + x) = color;
}

```



以上就是向内存写颜色数据，内存映射给 LCD 显示，所以 LCD 的像素位置和内存是对应的

如果内存地址是 32 位，存放的数据是按照 8 位的话，那么排列方式应该是这样



这就是内存存放是 8 位空间的排列方法

LCD 刷屏函数实现

```
// 把整个屏幕全部填充成一个颜色color
static void lcd_draw_background(u32 color)
{
    u32 i, j;

    for (j=0; j<ROW; j++)
    {
        for (i=0; i<COL; i++)
        {
            lcd_draw_pixel(i, j, color);
        }
    }
}
```

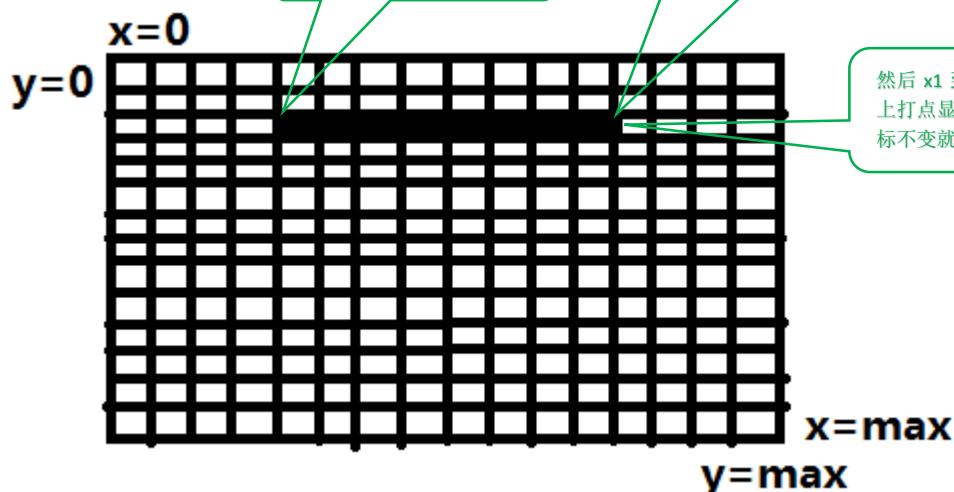
只要保证打点函数正常，刷屏也就是循环行列而已

LCD 画线原理

画横线

```
// 绘制横线，起始坐标为(x1, y)到(x2, y)，颜色是color
static void lcd_draw_hline(u32 x1, u32 x2, u32 y, u32 color)
{
    u32 x;

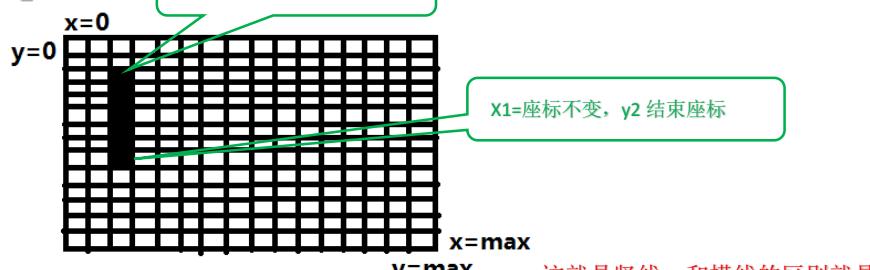
    for (x = x1; x < x2; x++)
    {
        lcd_draw_pixel(x, y, color);
    }
}
```



画竖线

```
// 绘制竖线，起始坐标为(x, y1)到(x, y2)，颜色是color
static void lcd_draw_vline(u32 x, u32 y1, u32 y2, u32 color)
{
    u32 y;

    for (y = y1; y < y2; y++)
    {
        lcd_draw_pixel(x, y, color);
    }
}
```



这就是竖线，和横线的区别就是 x 的变化改成 y

画斜线

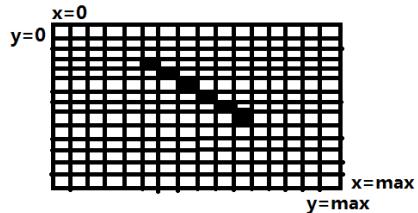
```
// glib 库中的画线函数，可以画斜线，线两端分别是(x1, y1)和(x2, y2)
void glib_line(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2, unsigned int color) { ... }

// glib 库中的画线函数，可以画斜线，线两端分别是(x1, y1)和(x2, y2)
void glib_line(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2, unsigned int color)
{
    int dx, dy, e;
    dx=x2-x1;
    dy=y2-y1;

    if(dx>=0)
    {
        if(dy >= 0) // dy>=0
        {
            if(dx>=dy) // 1/8 octant
            {
                e=dy-dx/2;
                while(x1<=x2)
                {
                    lcd_draw_pixel(x1, y1, color);
                    if(e>0) {y1+=1; e-=dx;}
                    x1+=1;
                    e+=dy;
                }
            }
            else // 2/8 octant
            {
                e=dx-dy/2;
                while(y1<=y2)
                {
                    lcd_draw_pixel(x1, y1, color);
                    if(e>0) {x1+=1; e-=dy;}
                    y1+=1;
                    e+=dx;
                }
            }
        }
        else // dy<0
        {
            dy=-dy; // dy=abs(dy)

            if(dx>=dy) // 8/8 octant
            {
                e=dy-dx/2;
                while(x1<=x2)
                {
                    lcd_draw_pixel(x1, y1, color);
                    if(e>0) {y1-=1; e-=dx;}
                    x1+=1;
                    e+=dy;
                }
            }
            else // 7/8 octant
            {
                e=dx-dy/2;
                while(y1>=y2)
                {
                    lcd_draw_pixel(x1, y1, color);
                    if(e>0) {x1+=1; e-=dy;}
                    y1-=1;
                    e+=dx;
                }
            }
        }
    }
}
```

移植 glib 函数主要是修改这个打点函数，换成你自己 LCD 能识别的打点函数

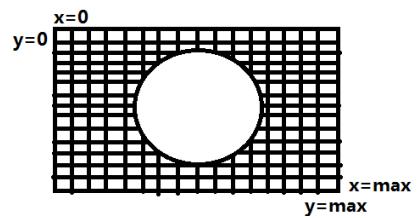


函数还没有展出完，反正去 glib 库找这个画斜线的函数就是了

画圆

```
//画圆函数，圆心坐标是(centerX, centerY)，半径是radius，圆的颜色是color
void draw_circular(unsigned int centerX, unsigned int centerY, unsigned int radius, unsigned int color)
{
    int x, y ;
    int tempX, tempY;;
    int SquareOfR = radius*radius;

    for(y=0; y<YSIZE; y++)
    {
        for(x=0; x<YSIZE; x++)
        {
            if(y<=centerY && x<=centerX)
            {
                tempY=centerY-y;
                tempX=centerX-x;
            }
            else if(y<=centerY&& x>=centerX)
            {
                tempY=centerY-y;
                tempX=x-centerX;
            }
            else if(y>=centerY&& x<=centerX)
            {
                tempY=y-centerY;
                tempX=centerX-x;
            }
            else
            {
                tempY = y-centerY;
                tempX = x-centerX;
            }
            if ((tempY*tempY+tempX*tempX)<=SquareOfR)
                lcd_draw_pixel(x, y, color);
        }
    }
},
```



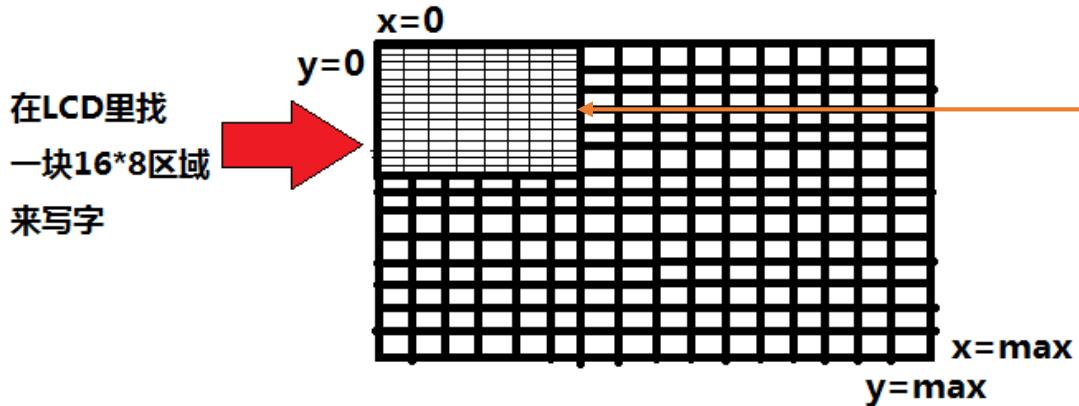
写字体，字体大小 8*16 个像素

```
// 写字
// 写字的左上角坐标(x, y), 字的颜色是color, 字的字模信息存储在data中
static void show_8_16(unsigned int x, unsigned int y, unsigned int color, unsigned char *data)
{
    // count记录当前正在绘制的像素的次序
    int i, j, count = 0;

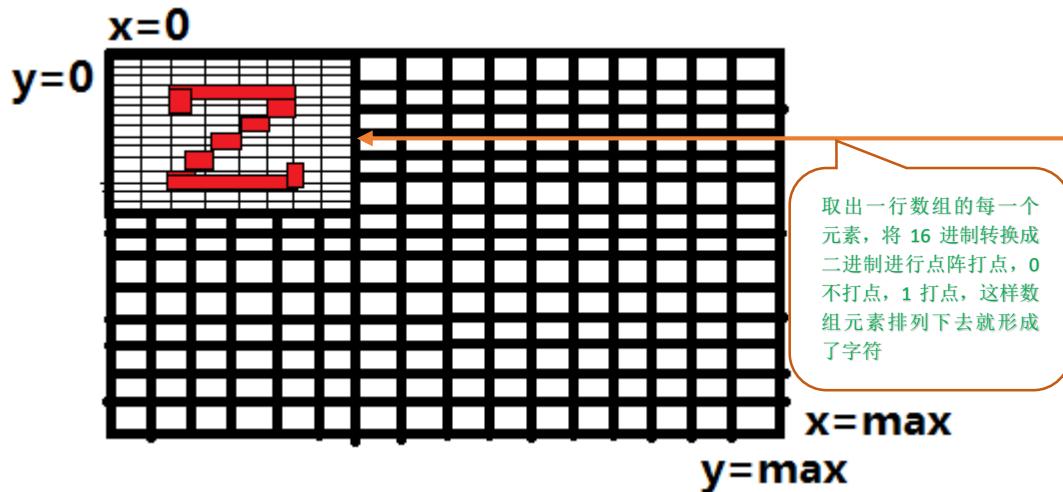
    for (j=y; j<(y+16); j++)
    {
        for (i=x; i<(x+8); i++)
        {
            if (i<XSIZE && j<YSIZE)
            {
                // 在坐标(i, j)这个像素处判断是0还是1, 如果是1写color; 如果是0直接跳过
                if (data[count/8] & (1<<(count%8)))
                    lcd_draw_pixel(i, j, color);
            }
            count++;
        }
    }
}

//以下是8*16的
//高是16
//横向扫描
const unsigned char ascii_8_16[95][16]=
{
{0x00, 0x00, 0x00}, /* " ", 0*/
{0x00, 0x00, 0x00, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x00, 0x18, 0x18, 0x00, 0x00}, /* "!", 1*/
{0x00, 0x48, 0x6C, 0x24, 0x12, 0x00, 0x00}, /* "", 2*/
{0x00, 0x00, 0x00, 0x24, 0x24, 0x7F, 0x12, 0x12, 0x12, 0x7F, 0x12, 0x12, 0x12, 0x00, 0x00}, /* "#", 3*/
{0x00, 0x00, 0x08, 0x1C, 0x2A, 0x2A, 0x0A, 0x0C, 0x18, 0x28, 0x2A, 0x2A, 0x1C, 0x08, 0x08}, /* {"} quot;, 4*/
{0x00, 0x00, 0x00}, /* "aww", 5*/
```

如果 x=0, y=0 就是寻找 LCD 的 0, 0 座标来显示 1 个字符, 该字符区域会占用 LCD16*8 个像素区域



```
//以下是8*16的
//高是16
//横向扫描
const unsigned char ascii_8_16[95][16]=
{
{0x00, 0x00, 0x00}, /* " ", 0*/
{0x00, 0x00, 0x00, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x00, 0x18, 0x18, 0x00, 0x00}, /* !"! ", 1*/
{0x00, 0x48, 0x6C, 0x24, 0x12, 0x00, 0x00}, /* "#", 2*/
{0x00, 0x00, 0x00, 0x24, 0x24, 0x24, 0x7F, 0x12, 0x12, 0x12, 0x7F, 0x12, 0x12, 0x12, 0x00, 0x00}, /* "#", 3*/
{0x00, 0x00, 0x08, 0x2A, 0x2A, 0x0A, 0x0C, 0x18, 0x28, 0x2A, 0x2A, 0x1C, 0x08, 0x08}, /* "[1] quot ; , 4*/
{0x00, 0x00, 0x00} /* "&gt;" , 5*/
}
```

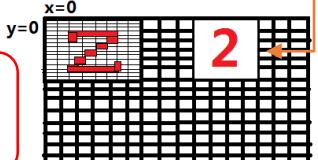


```
// 写字
// 写字的左上角坐标(x, y), 字的颜色是color, 字的字模信息存储在data中
static void show_8_16(unsigned int x, unsigned int y, unsigned int color, unsigned char *data)
{
    // count记录当前正在绘制的像素的次序
    int i, j, count = 0;
    for (j=y; j<(y+16); j++)
    {
        for (i=x; i<(x+8); i++)
        {
            if (i<XSIZE && j<YSIZE)
            {
                // 在坐标(i, j)这个像素处判断是0还是1，如果是1写color；如果是0直接跳过
                if (data[count/8] & (1<<(count%8)))
                    lcd_draw_pixel(i, j, color);
            }
            count++;
        }
    }
}
```

如果 $x=50, y=0$ 就是寻找 LCD 的 50, 0 座标来显示个字符

这个 $x+8$ 的 8 是该字符从 x 开始最多占用横坐标 8 个像素点

这个 XSIZE 和 YSIZE 是 LCD 屏幕最大的大小，我们这里 LCD 是 800×420 ，所以防止有人随便写 x, y 值超出屏幕范围



```

// 写字
// 写字的左上角坐标(x, y), 字的颜色是color, 字的字模信息存储在data中
static void show_8_16(unsigned int x, unsigned int y, unsigned int color, unsigned char *data)
{
    // count记录当前正在绘制的像素的次序
    int i, j, count = 0;

    for (j=y; j<(y+16); j++)
    {
        for (i=x; i<(x+8); i++)
        {
            if (i<XSIZE && j<YSIZE)
            {
                // 在坐标(i, j)这个像素处判断是0还是1, 如果是1写color; 如果是0直接跳过
                if (data[count/8] & (1<<(count%8)))
                    lcd_draw_pixel(i, j, color);
                count++;
            }
        }
    }
}

循环提取这一个 16 进制来进行打点
{0x00,
{0x00, 0x48, 0x6C, 0x24, 0x12, 0x00, 0x00}, /*"", 2*/
然后在获取下一个元素来循环打点

```

所以这个*data 到底获取数组那一行数据,得看下面字符串函数

```

// 写字符串
// 字符串起始坐标左上角为(x, y), 字符串文字颜色是color, 字符串内容为str
void draw_ascii_ok(unsigned int x, unsigned int y, unsigned int color, unsigned char *str)
{
    int i;
    unsigned char *ch;
    for (i=0; str[i]!='\0'; i++)
    {
        ch = (unsigned char *)ascii_8_16[(unsigned char)str[i]-0x20];
        show_8_16(x, y, color, ch);

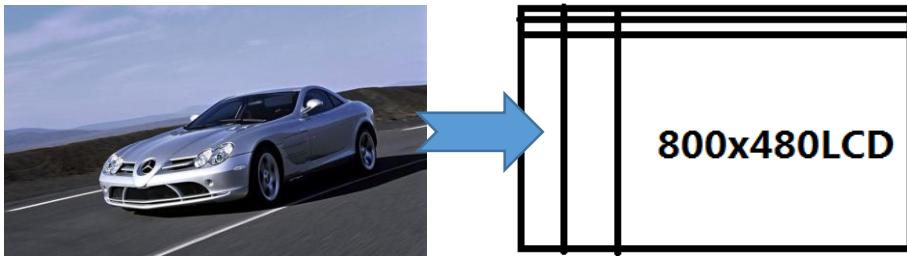
        x += 8;
        if (x >= XSIZEx
        {
            x -= XSIZE;           // 回车
            y += 16;              // 换行
        }
    }
}

X,y 字符在 LCD 开始座标, color 字符颜色, str 才是字符串
获取字符串的其中一个字符, 比如我这里是 'a' 字符, 这个 a 的 ascii 码是 97, 97-0x20=97-32=65 , 这里的 0x20 要转换成十进制来计算, 所以到数组 65 的元素
这里为什么要 unsigned char, 因为 str 取出来的是字符 a, 要转换成数字才能和 0x20 运算

```

{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3C, 0x42, 0x78, 0x44, 0x42, 0x42, 0xFC, 0x00, 0x00}, /*"a", 65*/
这样就在 LCD 指定位置显示一个字符, 然后循环多次, 就将字符串显示在 LCD 上了

画图片



先确定 jpeg 图片上 800X480 大小，然后将其放入 800X480LCD

尺寸	800 x 480
宽度	800 像素
高度	480 像素
水平分辨率	72 dpi
垂直分辨率	72 dpi
位深度	24

这是我们图片信息，800X480 像素，每个像素 bpp 是 24 位，也就是 R=8 位, G=8 位, B=8 位，所以这个图片大小就是 $800 \times 480 \times 3$ 字节数据=1152000 字节，也就是 1.152M

我们将这个图片转换成数组，就可以按照一个像素一个像素方式在图片每个像素的位置打颜色点，数组形式是 `unsigned char image[800*480*3]` 这就是数组的大小

用 **Image2LCD** 软件将 **JPEG** 图片转换成 **24** 位真彩色的，选择红绿蓝排列，还是绿蓝红排列，是根据 **LCD** 来确定。如果我选择红绿蓝，那么数组就是按照红绿蓝来排列。输出成 **C** 语言数组，

红 绿 蓝

```
const unsigned char gImage_800480[1152000] = { /* 0X00, 0X18, 0X20, 0X03, 0XE0, 0X01, 0X00, 0X1B, */  
    0X00, 0X24, 0X7C, 0X01, 0X26, 0X7E, 0X03, 0X28, 0X80, 0X03, 0X28, 0X80, 0X02, 0X27, 0X7F, 0X02,  
    0X27, 0X7F, 0X02, 0X27, 0X7F, 0X03, 0X28, 0X80, 0X01, 0X26, 0X7E, 0X02, 0X27, 0X7F, 0X02, 0X27,  
    0X7F, 0X02, 0X27, 0X7F, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C,  
    0X05, 0X27, 0X7E, 0X04, 0X26, 0X7D, 0X03, 0X25, 0X7C, 0X02, 0X24, 0X7B, 0X02, 0X24, 0X7B, 0X02,  
    0X24, 0X7B, 0X01, 0X23, 0X7A, 0X01, 0X23, 0X7A, 0X00, 0X25, 0X76, 0X02, 0X27, 0X78, 0X02, 0X26,  
    ...};
```

这就是转换出来的图片数组

这个图片数组就是 3 个字节一个像素，所以我们循环取出；来挨着 LCD 排列就是了

```
const unsigned char gImage_800480[1152000] = { /* 0X00, 0X18, 0X20, 0X03, 0XE0, 0X01, 0X00, 0X1B, */  
    0X00, 0X24, 0X7C, 0X01, 0X26, 0X7E, 0X03, 0X28, 0X80, 0X03, 0X28, 0X80, 0X02, 0X27, 0X7F, 0X02,  
    0X27, 0X7F, 0X02, 0X27, 0X7F, 0X03, 0X28, 0X80, 0X01, 0X26, 0X7E, 0X02, 0X27, 0X7F, 0X02, 0X27,  
    0X7F, 0X02, 0X27, 0X7F, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C, 0X01, 0X27, 0X7C,  
    0X05, 0X27, 0X7E, 0X04, 0X26, 0X7D, 0X03, 0X25, 0X7C, 0X02, 0X24, 0X7B, 0X02, 0X24, 0X7B, 0X02,  
    0X24, 0X7B, 0X01, 0X23, 0X7A, 0X01, 0X23, 0X7A, 0X00, 0X25, 0X76, 0X02, 0X27, 0X78, 0X02, 0X26,  
    ...};
```

// 画800×480的图，图像数据存储在pData所指向的数组中
void lcd_draw_picture(const unsigned char *pData)

```
{  
    u32 x, y, color, p = 0;  
  
    for (y=0; y<480; y++)  
    {  
        for (x=0; x<800; x++)  
        {  
            // 在这里将坐标点(x, y)的那个像素填充上相应的颜色值即可  
            color = (pData[p+0] << 0) | (pData[p+1] << 8) | (pData[p+2] << 16);  
            lcd_draw_pixel(x, y, color);  
            p += 3;  
        }  
    }  
}
```

这就是画图片

这个图像程序编译完成之后在板子上无法运行，因为程序大小超过 16KB 了，图像数组占用了 1M 多，所以这样编译程序不行。

所以我们还是模仿 SD 卡启动程序那样，把代码分成 BL1 和 BL2

在 BL1/start.S 里面初始化 DDR，然后跳到 copy_b12_2

```
//bic r0, r0, #(1<<12)          // bit12 置0 关icache
orr r0, r0, #(1<<12)           // bit12 置1 开icache
mcr p15,0,r0,c1,c0,0;

// 第4步：初始化ddr
bl sram_asm_init

// 第5步：重定位，从SD卡第45扇区开始，复制32个扇区内容到DDR的0x23E00000
bl copy_b12_2_ddr
```

// 汇编最后的这个死循环不能丢
b .

```
#define SD_START_BLOCK 45
#define SD_BLOCK_CNT (2048*2)
#define DDR_START_ADDR 0x23E00000

typedef unsigned int bool;

// 通道号: 0, 或者2
// 开始扇区号: 45
// 读取扇区个数: 32
// 读取后放入内存地址: 0x23E00000
// with_init: 0
typedef bool (*pCopySDMMC2Mem)(int, unsigned int, unsigned short, unsigned int*, bool);

typedef void (*pBL2Type)(void);

// 从SD卡第45扇区开始，复制32个扇区内容到DDR的0x23E00000，然后跳转到23E00000去执行
void copy_b12_2_ddr(void)
{
    // 第一步，读取SD卡扇区到DDR中
    pCopySDMMC2Mem p1 = (pCopySDMMC2Mem) (*(unsigned int *) 0xD0037F98);
    // pCopySDMMC2Mem p1 = (pCopySDMMC2Mem) 0xD0037F98;

    led2();
    delay();
    p1(2, SD_START_BLOCK, SD_BLOCK_CNT, (unsigned int *) DDR_START_ADDR, 0); // 读取SD卡到DDR中
    led1();
    delay();
    // 第二步，跳转到DDR中的BL2去执行
    pBL2Type p2 = (pBL2Type) DDR_START_ADDR;
    p2();

    led3();
    delay();
}

#define WTC0N      0xE2700000
#define SVC_STACK  0xd0037d80

.global _start          // 把_start链接属性改为外部，这样其他文件就可以看见_start了
_start:
//run_on_dram:
    // 长跳转到led_blink开始第二阶段
    ldr pc, =main           // ldr指令实现长跳转

// 汇编最后的这个死循环不能丢
b .
```

```
#include "main.h"

int main(void)
{
    lcd_test();
    return 0;
}
```

跳转到 0x23e00000
执行的程序是 BL2
的 start.S

将图像 BL2 程序下
载到 0x23e00000
位置，然后跳转到
这个位置去执行

该 start.S 跳转到
main 主函数去执行
图片显示程序

该程序就是在 LCD
显示图片

如果你的 Image2LCD 取模软件输出的数组是蓝绿红排列顺序

```
// 画800×480的图，图像数据存储在pData所指向的数组中
void lcd_draw_picture(const unsigned char *pData)
{
    u32 x, y, color, p = 0;

    for (y=0; y<480; y++)
    {
        for (x=0; x<800; x++)
        {
            // 在这里将坐标点(x, y)的那个像素填充上相应的颜色值即可
            color = (pData[p+0] << 0) | (pData[p+1] << 8) | (pData[p+2] << 16);
            lcd_draw_pixel(x, y, color);
            p += 3;
        }
    }
}
```

P+0 数组红色在前

P+1 数组绿色在中

P+2 数组蓝色在后

这是以前的红绿蓝排列顺序代码

现在是蓝绿红，所以我们要修改代码

```
// 画800×480的图，图像数据存储在pData所指向的数组中
```

```
void lcd_draw_picture(const unsigned char *pData)
{
    u32 x, y, color, p = 0;

    for (y=0; y<480; y++)
    {
        for (x=0; x<800; x++)
        {
            // 在这里将坐标点(x, y)的那个像素填充上相应的颜色值即可
            color = ((pData[p+2] << 0) | (pData[p+1] << 8) | (pData[p+0] << 16));
            lcd_draw_pixel(x, y, color);
            p += 3;
        }
    }
}
```

P+2 数组蓝色在前

P+1 数组绿色在中

P+0 数组红色在后

这样就搞定了