## 一、段的概念，重定位的引入

S3C2440的cpu可以直接给sdram，nor flash，但是不能直接给nand flash发送命令

假如把程序烧写到nand flash上，CPU如何从里面执行代码

Nand启动：

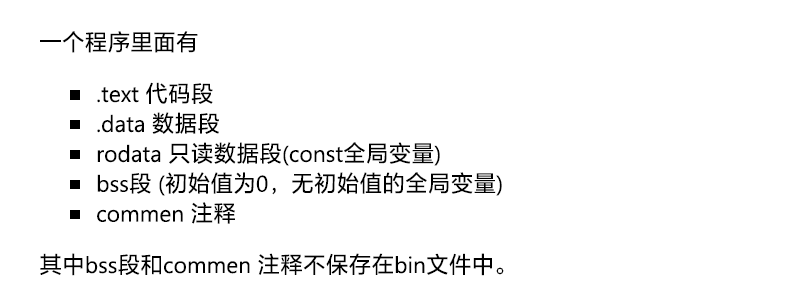
1. 上电后，硬件自动把nand flash前4k复制到sram
2. Cpu从0地址运行sram

如果程序大于4k

前4k的代码就得把整个程序放到sdram中，即代码重定位

Nor flash启动：

CPU认为0地址在nor flash上，片内sram的基地址变成0x40000000（nand启动时片内sram的基地址是0），由于nor flash可以像内存一样读，但不能直接写，因此需要，把全局变量和静态变量重定位，放到sdram。



## 二、链接脚本的引入与简单测试

对比下面的程序烧写到nor flash和nand flash启动有什么不同

main.c

#include "s3c2440\_soc.h"

#include "uart.h"

#include "init.h"

char g\_char = ‘A’;//定义一个全局变量

const char g\_char2 = ‘B’;//静态变量

int g\_A = 0;

int g\_B ;

int main(void)

{

uart0\_init();

while(1)

{

putchar(g\_char);

g\_char++;

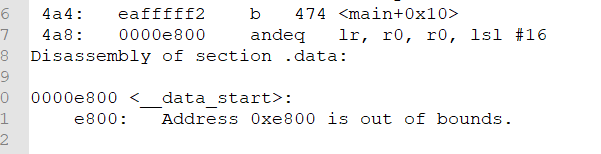
delay(100000);

}

return 0 ;

}

编译后发现sdram.dis文件很大，发现data段放在



Makefile 中改动



文件大小恢复正常

当nand启动时：

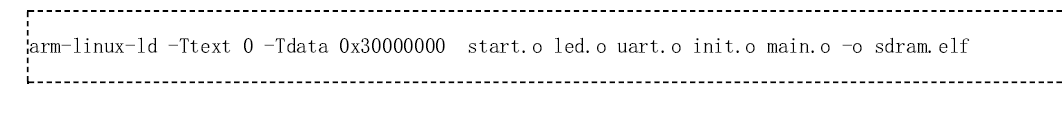


当nor启动时：程序不执行？？理论上是一直输出AAAA

把时钟设置里的MPLLCON放到最后就好了？？

这说明nor上g\_char++ 的操作是无效的，只能读取不能写入

为了解决这个问题，直接把该变量放在sdram里，



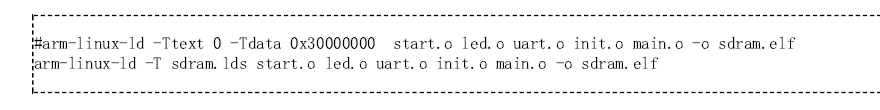
但这样编译出来的bin文件从0地址到0x30000000地址，有700M，text段和data段之间有间隔，称之为黑洞

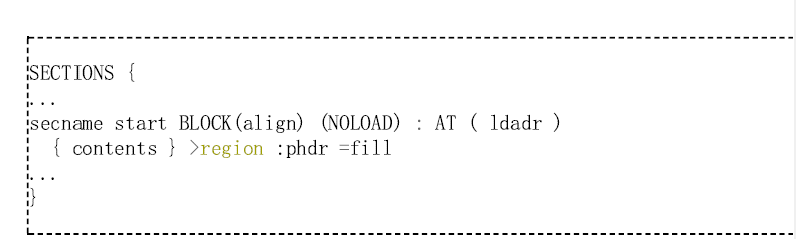
解决这个问题有两个办法：

1. 把g\_char放在nor上，但运行时在sdram
2. 把g\_char放在sdram上，运行时在sdram

先来尝试第一种办法：

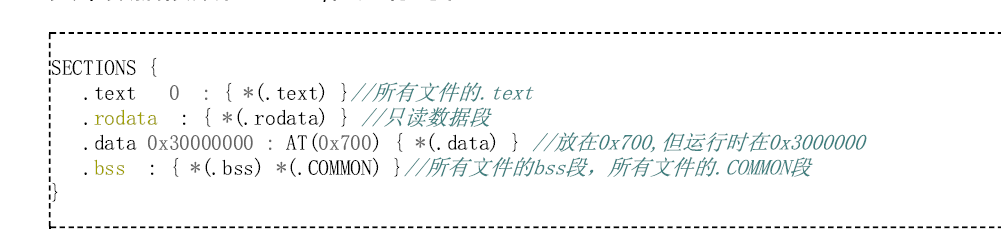
使用链接脚本



链接脚本的语法：

依次排列代码段，rodata段（只读数据段），data段，bss段，common段

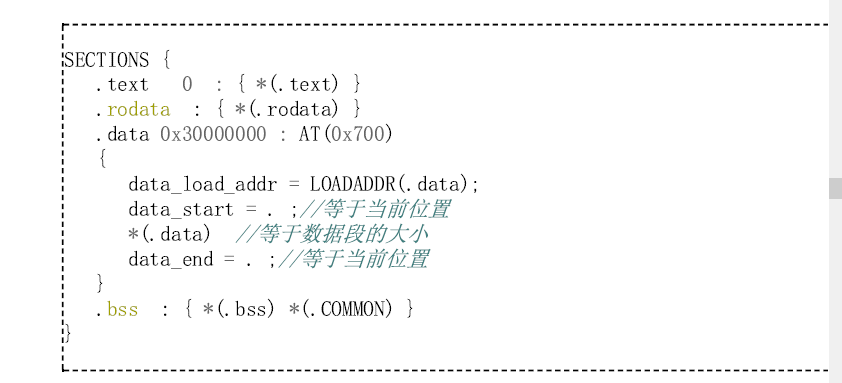
其中data段存放在0x700，运行时在0x30000000



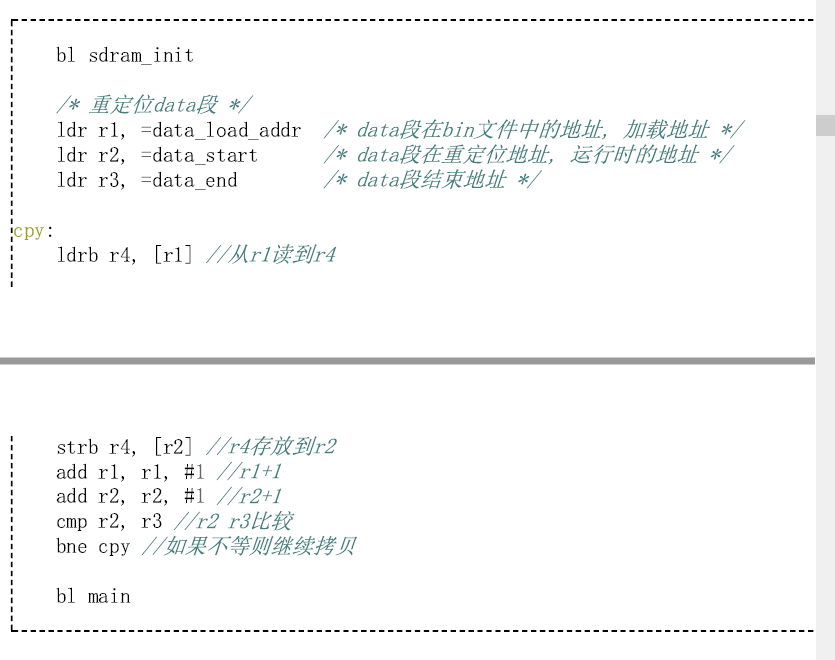
但只是这样还不够，这里并没有在0x30000000准备好数据，只是让他从0x30000000获取数据，需要先把g\_char重定位到0x30000000，在start.S加入：



这种方法不实用，修改如下：

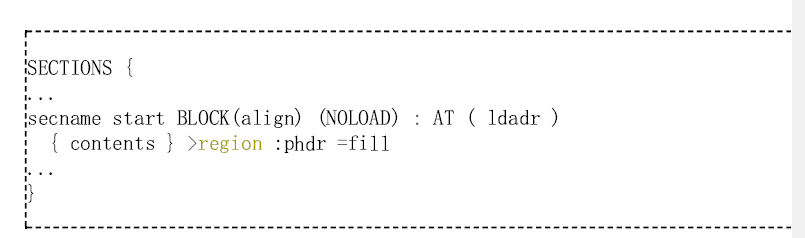


修改start.S

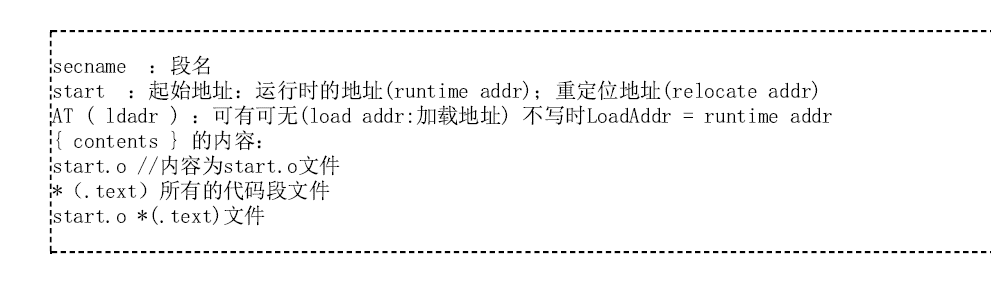


## 三、链接脚本的解析

链接脚本的语法



解释：



elf文件：

1. 链接得到elf文件，含有地址信息（load addr）
2. 使用加载器 对于裸板是jtag调试工具
3. 运行程序
4. 如果loadaddr ！= runtimeaddr 程序需要重定位

核心程序运行时应该位于runtimeaddr（reloate addr）或者链接地址

bin文件

1. elf生成bin文件
2. 硬件机制启动
3. 如果bin文件所在位置不等于runtimeaddr，程序本身实现重定位

bin 、elf文件都不保存bss段，初始值为0或者没有初始值的全局变量

程序运行时把bss段对应的空间清零

做个实验，把全局变量g\_A以16进制打印出来

void printHex(unsigned int val)

{

int i;

unsigned char arr[8];

for(i = 0; i < 8; i++)

{

arr[i] = val & 0xf;

val >> = 4;

}

puts(“0x”);

for(i = 7;i >= 0; i--)

{

if(arr[i] >= 0 && arr[i] <= 9)

` putchar(arr[i] + ‘0’);

else if(arr[i] >= 0xA && arr[i] <= 0xF)

putchar(arr[i] – 0xA + ‘A’);

}

}

int main (void)

{

uart0\_init();

puts(“\n\rg\_A =”);

printHex(g\_A);

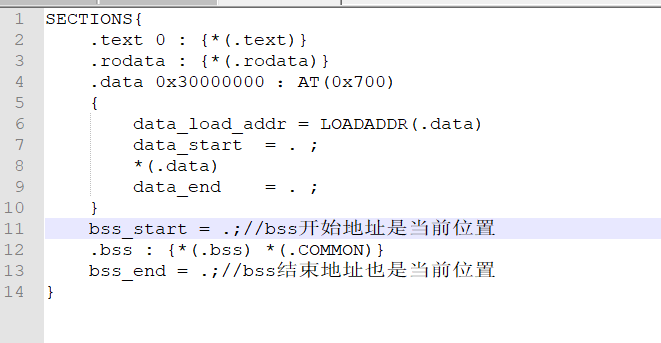
puts(“\n\r”);

}

这里没有清楚bss段所以是一个随机值，不是0

下面来清楚bss段

lds链接脚本



修改start.S，清楚bss段

ldr r1, =bss\_start

ldr r2, = bss\_end

mov r3,#0

clean:

strb r3,[r1]

add r1,r1,#1

cmp r1,r2

bne clean

bl main

现在全局变量为0，减少几十上千个全局变量的存储空间

## 四、拷贝代码和链接脚本的改进

前面重定位时，需要ldrb命令从nor flash读取1字节数据再strb到sdram里面

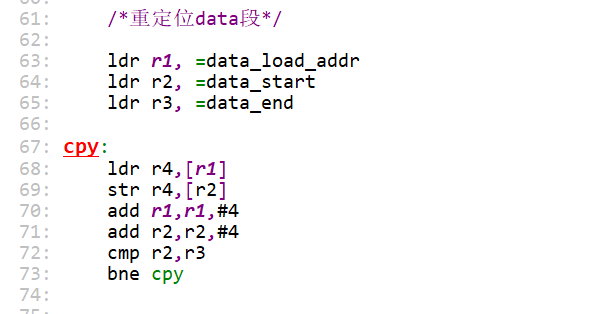
jz2440上nor flash是16位，sdram的32位

假设现在需要复制16byte数据，两个命令加起来要访问硬件32次

现在使用ldr从nor flash中读取，每次加载4字节数据，因此CPU只需要执行4次

由于nor是16位的，内存控制器每次收到cpu命令后，需要分两次执行，共需要8次

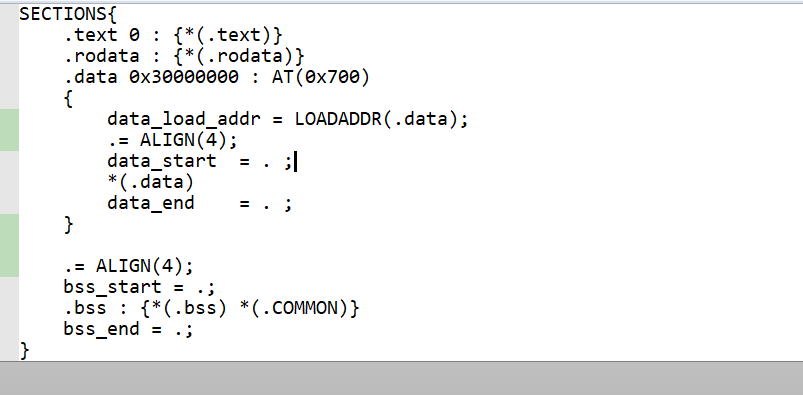
str写sdram，CPU只需执行4次，内存控制器每次收到命令后直接硬件访问32sdram，因此这里只需要4次。总控需要12次，极大提高了效率



这里又出现了新问题，没有四字节对齐，先要使对齐然后程序才能正常输出

修改链接脚本

data、bss段都四字节对齐



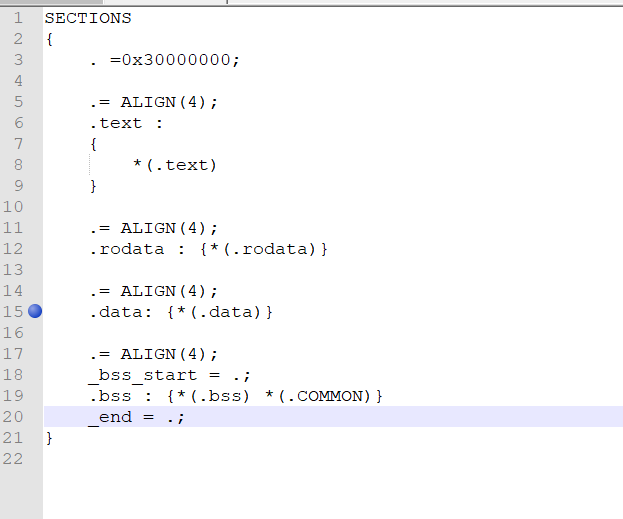
## 五、代码重定位和位置无关码

上面只是重定位了数据段

下面尝试重定位整个代码

1. 程序从flash复制到运行地址，链接脚本要指定运行地址为sdram
2. 编译连接生成的bin文件，需要sdram上运行，但上电后必须先在0四肢运行，这就要求定位之前的代码与位置无关，即位置无关码

参考uboot修改链接脚本

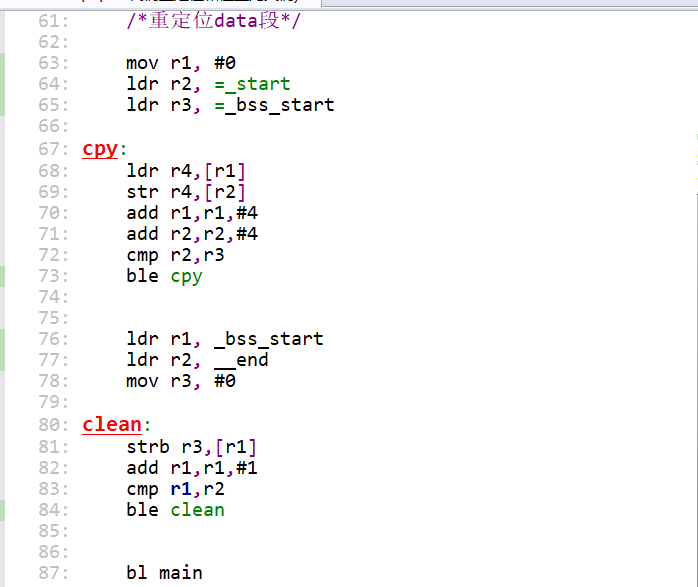
这个脚本称为一体式链接脚本，对比前面的分体式链接脚本的区别在于代码段和数据段是否分开存放。

一体式的链接脚本代码段是roadata段，data段，bss段，都是连续一起的

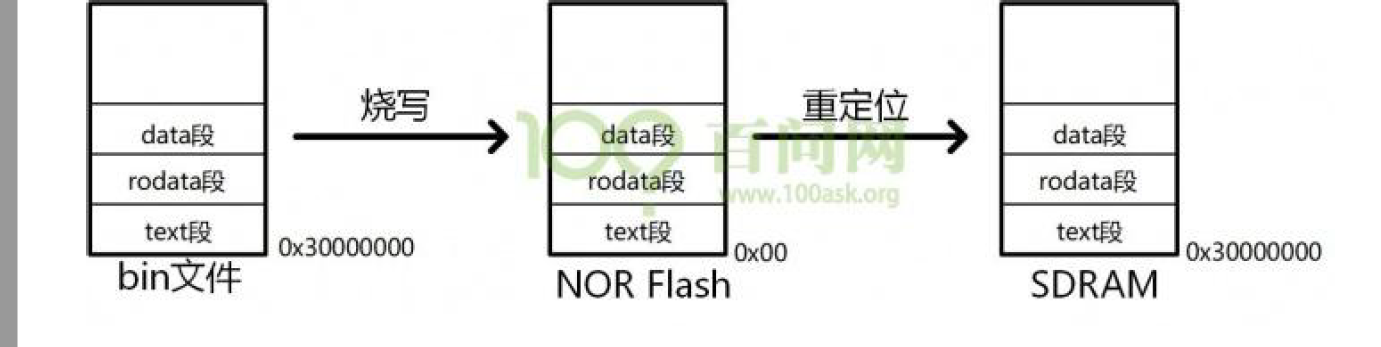
分体式链接脚本则是text段，rodata段，中间隔很远后才是data，bss段

以后更多采用一体式链接脚本，

1. 分体式适合单片机，单片机自带有flash，不需要再将代码复制到内存占用空间，而嵌入式系统内存很大，并且有些嵌入式系统没有nor flash等可以直接运行代码的flash，就需要从nand flash或者sd卡复制整个代码到内存
2. jtag等调试器一般只支持一体式链接脚本



烧写到nor flash上，运行



bin文件里，text段的为之而是0x30000000，烧写到nor上后变成0地址，text段又将整个程序复制到sdram

还有一个，反汇编文件里，b bl某个值，并不是真正起到跳转，而是为了方便查看

怎么写位置无关码：

1. 使用相对跳转命令b或者bl
2. 重定位之前，不可以使用绝对地址，不可访问全局变量，静态变量，不可访问有初始值的数组（初始值放在rodata里，使用绝对地址来访问）
3. 重定位后使用ldr pc，=xxx来跳转

前面的程序使用bl main，程序任在nor/sram执行，要想让main函数在sdram执行

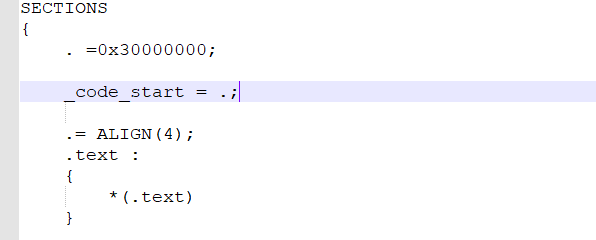
ldr pc，=main//绝对跳转

## 六、c实现重定位和bss段的清除

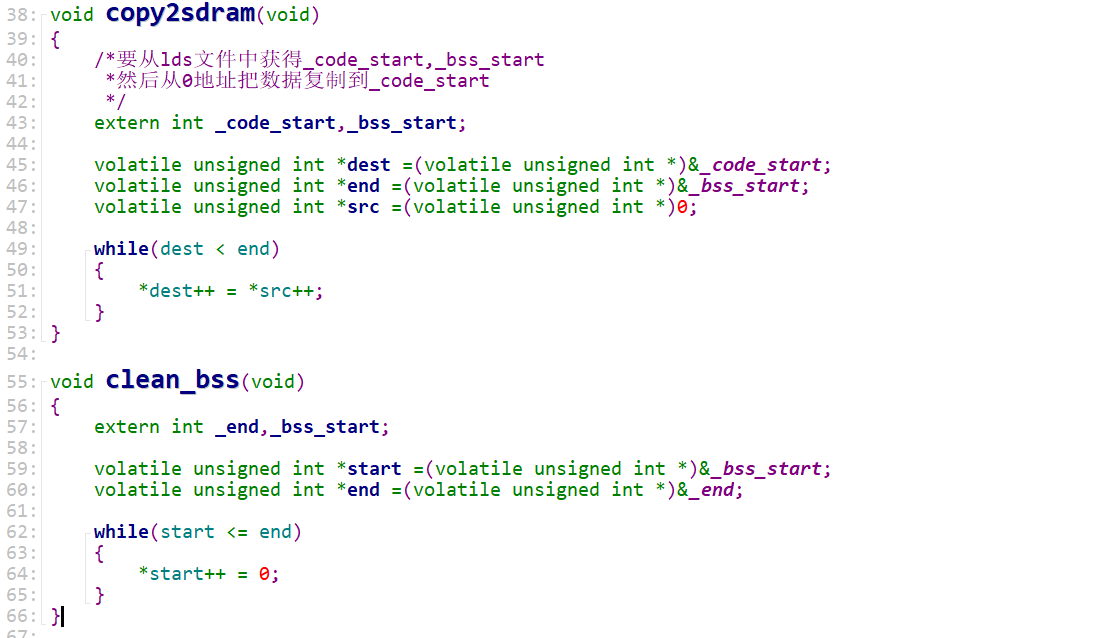
首先把start.S重定位和清除bss段的汇编改为调用c函数



修改链接脚本，让\_code\_start等于当前地址，也就是0x30000000



修改init.c用函数来获取参数



注意：1、c程序中不保存lds文件中的变量，lds再大也不影响

2、借助symbol table保存lds的变量，使用是加上&得到值，链接脚本的变量要在c程序中声明为外部变量，任何类型都可以

注意：



lds文件记得空格。不然会报错