# SDRAM 和重定位relocate

什么是看门狗：

看门狗（watch dog timer 看门狗定时器）：场景：家门口有一只狗，狗定时会饿（譬如两小时），够饿了后会胡乱咬人，人要安全进出必须提前喂狗（上次喂过的两小时内喂狗才行）。如果超时没喂狗就会被咬死，如果提前喂狗没有关系，但是本次喂狗时间就会从这里开始计算。

现实中因为一些外部因素，电子设备经常会跑飞或者死机，（譬如极端炎热，极端寒冷，工厂复杂场合）。在这种情况下我们希望设备自动复位而不需要人工干预（无人值守）。看门狗就是用来完成这个工作的。看门狗其实是我们SoC内部的一个定时器（类似于闹钟类似于门口的狗），定好时间后看门狗定时器会去计时，时间到之前（够饿了之前），必须去重新置位看门狗定时器（喂狗），如果没有喂狗系统会被强制复位。

系统在正常工作时，系统软件会自己去喂狗，所以看门狗定时器不会复位，但是系统一旦故障跑飞，看门狗就没人喂了，然后下一个周期就会自动复位，达到我们期望的效果。

为什么要关看门狗？

一般CPU设计，在CPU启动后看门狗默认是工作的（为什么默认不关闭而要工作？），好处就是没有空档和漏洞，坏处就是在启动代码段我们不方便去喂狗。

很多CPU内部都是没有BL0的，需要自己去关看门狗

1找到关键性SFR（特殊功能寄存器）

WTCON(0x2700000)

#define WTCON 0x2700000

.global \_start

\_start:

//第一步：关看门狗（向WTCON的bit5写入0即可）

ldr r0, =WTCON

ldr r1, =0x0

str r1, [r0]

C语言运行时需要和栈的意义

“C语言运行时（runtime）”需要一定的条件，这些条件又由汇编提供。C语言运行时主要需要栈。

C语言与栈的关系：C语言中的局部变量都是用栈来实现的。如果汇编部分没有给C部分预先设置合理合法的栈地址，那么C代码中定义的局部变量就会落空，整个程序就死掉了。

我们平时在编写单片机程序（譬如51单片机）或者编写应用程序时并没有去设置栈，但是C程序还是可以运行的。原因是：在单片机中由硬件初始化时提供了一个默认可用的栈，在应用程序中我们编写的C程序其实步兵师全部，编译器（gcc）在链接的时候会帮我们自动添加一个头，这个头就是一段引导我们的C程序能够执行的一段汇编代码实现，这个代码中就帮我们实现了栈和其他的运行时需要。

CPU模式和各种模式下的栈

在ARM中37个寄存器，每种模式下都有自己的独立的SP寄存器（r13），为什么这么设计？

如果各种模式都是用一个SP，那么就意味着整个程序（操作系统内核程序，用户自己编写的应用程序）都是用一个栈的。你的应用程序一旦出错（譬如栈溢出），就会连累操作系统的栈也损坏，整个操作系统的程序就会崩溃。这样的操作系统是非常脆弱的，不合理的。

解决方案就是各种模式下使用不同的栈，操作系统内核使用自己的栈，每个应用程序使用自己独立的栈，一个损坏不会影响其他。

我们要设置栈，不可能而且也没有必要去设置所有的栈，我们要找到自己的模式，然后设置自己模式下的栈到合理的位置即可

注意：系统在复位后默认是进入SVC（管理）模式的

我们如何访问SVC 模式下的SP呢？很简单，先把模式设置为SVC,再直接操作SP。但是因为我们复位后就已经是SVC模式了，所以直接设置SP模式即可。

查阅文档并设置栈指针至合法位置

栈必须是当前一段可用的内存（可用指的是：这个地方必须有初始化可以访问的内存，而且这个内存只会被我们用作栈，不会被其他程序征用）

当CPU刚复位（刚启动）外部的DRRAM尚未初始化，目前可用的内存只有内部的SRAM（因为他不需要初始化即可使用）。因此我们只能在SRAM中找一段内存来作为SVC的栈。

栈有四种：满减栈，满增栈；空减栈，空增栈

在ARM中ATPCS（ARM关于程序应该怎么实现的一个规范）要求使用满减栈

结合iROM\_application\_note中的memory map，可知SVC栈应该设置为0xd0037D80(大地址)

汇编程序和C程序互相调用

bl cfunction

#define WTCON 0x2700000

#define SVC\_STACK 0xd0037D80

.global \_start

\_start:

//第一步：关看门狗（向WTCON的bit5写入0即可）

ldr r0, =WTCON

ldr r1, =0x0

str r1, [r0]

//第二步：设置SVC栈

ldr sp, =SVC\_STACK

//从这里就可以开始调用C语言程序了

bl led\_blink //led\_blink 是C语言实现的一个函数

b .

//汇编最后的这个死循环不能丢

C函数的编写和被汇编调用

在工程中新建并添加一个C语言源文件（led.c），注意添加时要修改makefile

在汇编启动代码中设置好栈后，使用bl xxx 的方式来调用C中的函数xxx

使用C语言来访问寄存器的语法

寄存器的地质类似与内存地址（IO与内存统一编址的）所以这里的问题是用C语言读写寄存器，就是用C语言来读写内存地址。用C语言来访问内存，就要用到指针

unsigned int \*p =(unsigned int \*)0xE0200240;

\*p =0x11111111;

编译报错（实际上是连接阶段报错）：undefined reference to `\_\_aeabi\_unwind\_cpp\_pr1'

解决：在编译时添加-nostdlib这个编译项既可以解决nostdlib就是不使用标准函数库，标准函数库就是编译器自带的函数库，用-nostdlib可以让编译器连接器优先选择我程序内自己写的函数库。

神奇的volatile

volatile的作用是让程序在编译时，编译器不要对程序做优化，优化有时候是ok的，但有时候造成程序不对，如果一个变量是易变的，不希望编译器帮我们做优化，就在这个变量定义时加volatile。

加不加有没有差别取决于编译器。

led.c

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

void delay(void)

//该函数要实现led闪烁效果

void led\_blink(void){

//led初始化，也就是把GPJ0CON中设置为输出模式

unsigned int \*p =(unsigned int \*)GPJ0CON;

unsigned int \*p1=(unsigned int \*)G0J0DAT;

\*p =0x11111111;

while(1){

//led亮

\*p1=((0<<3)|(0<<4)|(0<<5))

//延时

delay();

//led 灭

\*p1=((1<<3)|(1<<4)|(1<<5))

//延时

delay();

}

}

void delay(void){

volatile unsigned int i =900000; //volatile让编译器不要优化，这样才能真正的减

while(i--); //才能消耗时间，实现delay

}

等效版

led.c

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

#define rGPJ0CON=\*((volatile unsigned int \*p)GPJ0CON)

#define rGPJ0DAT=\*((volatile unsigned int \*p)GPJ0DAT)

void delay(void)

//该函数要实现led闪烁效果

void led\_blink(void){

//led初始化，也就是把GPJ0CON中设置为输出模式

//unsigned int \*p =(unsigned int \*)GPJ0CON;

//unsigned int \*p1=(unsigned int \*)G0J0DAT;

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))

//延时

delay();

}

}

void delay(void){

volatile unsigned int i =900000; //volatile让编译器不要优化，这样才能真正的减

while(i--); //才能消耗时间，实现delay

}

iROM中BL0对cache的操作

首先，icache的一切动作都是自动的，不需要人为干预，我们所需要做的就是打开/关闭icache

其次，在210的iROM中BL0已经打开了icache，所以之前看到的都是icache打开的现象。

汇编代码读写cp15以开关icache

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;

三种情况下的实验现象

1. 直接使用BL0中对icache的操作
2. 关icache
3. 开icache

实验结果分析：

结论1：irom中确实是打开了icache的。

结论2：icache关闭确实比icache打开的时候led闪烁变慢，说明指令执行速度变慢。

#define WTCON 0x2700000

#define SVC\_STACK 0xd0037D80

.global \_start

\_start:

//第一步：关看门狗（向WTCON的bit5写入0即可）

ldr r0, =WTCON

ldr r1, =0x0

str r1, [r0]

//第二步：设置SVC栈

ldr sp, =SVC\_STACK

//第三步：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;

//从这里就可以开始调用C语言程序了

bl led\_blink //led\_blink 是C语言实现的一个函数

b .

//汇编最后的这个死循环不能丢

重定位引入和链接脚本

事实：大部分指令是位置有关编码

位置无关编码：（PIC position independent code）:会变源文件被编码成二进制可执行程序时编码方式和位置（内存地址）无关

位置有关编码：汇编源码编码成二进制可执行程序后和内存地址是有关的

我们在设计一个程序时，会给这个程序指定一个运行地址（链接地址），就是说我们在编译程序时其实是知道程序将来被运行时的地址（运行地址），而且必须给编译器链接器制定这个地址才行，最后得到的二进制程序理论上是和你指定的运行地址有关的，将来这个程序被执行时必须被放在当时编译连接时给定的那个地址下才行，否则不能运行（就叫位置有关代码）。但是有个特别的指令他可以跟指定的地址没有关系，也就是说这些代码实际运行时不管放在那里都能正常运行

链接地址和运行地址：可能相同也可能不同

对于位置有关代码来说：最终指定时的运行地址和编译连接时给定的链接地址必须相同，否则一定出错。之前的裸机程序中，Makefile中用 -Ttext 0x0

来指定链接地址是0x0.这意味着我们认为这个程序将来会放在0x0这个内存地址去运行。但是实际上我们运行时的地址是0xd0020010(用dnw下载时指定的下载地址)。这两地址看似不同，但实际相同，这是因为S5PV210内部做了映射，把SRAM映射到了0x0地址去。

链接地址：程序链接时指定的地址（Makefile中用-Ttext，或者链接脚本）

运行地址：程序实际运行时地址（指定方式：由实际运行时被加载到内存的哪个位置说了算）

再解S5PV210的启动过程：三星推荐和uboot的实现是不同的

三星推荐的启动方式中：bootloader必须小于96KB并大于16KB，假定bootloader为80KB，启动过程是这样子的：先开机上电后BL0运行，BL0会加载外部启动设备中的bootloader的前16KB（BL1）到SRAM中去运行，BL1运行时加载BL2（bootloader中80-16=64KB）到SRAM中（从SRAM的16KB处开始使用）去运行；BL2运行时会初始化DDR并且将OS搬运到DDR去执行OS，启动完成。

uboot实际使用的方式：uboot大小随意，假定为200KB，启动过程是这样子：先开机上电后BL0运行，BL0会加载外部启动设备中的bootloader的前16KB（BL1）到SRAM中去运行，BL1运行时会初始化DDR，然后整个uboot搬运到DDR中，然后用一句长跳转指令（从SRAM跳转到DDR）指令从SRAM中直接跳转到DDR中继续执行uboot直到uboot完全启动。uboot启动后在uboot命令中行中去启动OS。

重定位原因：

首先，有时候链接地址和运行地址必须不相同，而且还不能用位置无关代码，这时候只能重定位。

分散加载：把uboot分成2部分，（BL1和整个uboot），两部分分别指定不同的链接地址。启动时将两部分加载到不同的地址（BL1加载到SRAM，整个uboot加载到DDR），这时候不用重定位也能启动。

评价：分散加载其实相当与手工重定位，重定位是用代码进行重定位，分散加载是手工进行重定位。

运行时地址由什么决定？

运行时地址是由运行是决定的（编译链接时是无法绝对确定运行时地址的）

链接地址是由什么决定的？

链接地址是由程序员在编译连接的过程中，通过Makefile中-Ttext xxx或者在链接脚本中指定的。程序员事先会预知自己的程序的执行要求，并且有一个期望的执行地址，并且会用这个地址来做链接地址。

举例：1、linux中的应用程序：gcc hello.c -o hello,这时使用的默认地址就是0x0，所以应用程序都是链接在0地址的。因为应用程序在操作系统的一个进程中，在这个进程中这个应用程序独享4G地址空间。所以应用程序都可以连接到0地址。因为每个进程都是从0地址开始的。（编译时可以不给定链接地址而都是用0）

2、210的裸机程序，运行地址是由我们下载时确定，下载时下载到0xd00200010,所以就从这里开始运行（这个下载地址也不是我们随意定的，是IROM中的BL0加载BL1时事先指定好的地址，这是由CPU的设计决定的）。所以理论上我们编译链接时应该将地址指定到0xd00200010,但是实际上我们在之前裸机程序中都是使用位置无关码PIC，所以链接地址可以是0.

从源码到可执行程序的步骤：预编译、编译、链接、strip

预编译：预编译器执行：譬如C中的宏定义就是预编译器处理，注释等

编译：编译器来执行。把源码.c .S变成机器码.o文件。

链接：链接器来执行。把.o文件中的各函数（段）按照一定规则（链接脚本来指定）累积在一起，形成可执行文件。

strip：strip是把可执行程序中的符号信息给拿掉，以节省空间。（Debug版本和Release版本）

objcopy：有可执行程序生成可烧录的镜像bin文件。

段名分两种：一种是编译器链接器内部指定好的，先天性名字；一种是程序员自己指定的、自定义段名

bss段:（.bss）又叫ZI（zero initial）段，就是零初始化段，对应于C语言中初始化为0的全局变量。

1. C语言中全局变量未显式初始化，值是0，本质是C语言把这类全局变量放在了bss段，从而保证为0.
2. c运行时环境如何保证显式初始化为非0的全局变量的值在main之前就被赋值了？就是因为他把这类变量放在了.data段中，而.data段会在main执行之前被处理（初始化）

链接脚本：

链接脚本其实是个规则文件，他是程序员用来指挥链接器工作的，链接器会参考链接脚本，并且使用其中规定的规则来处理.o文件中的那些段，将其链接成一个可执行程序

link.lds

SECTIONS{

. =0xd0024000;

.text :{

start.o

\* (.text)

}

.data :{

\* (.data)

}

bss\_start = .; //标志出当前位置地址，可以在其他处使用

.bss :{

\* (.bss)

}

bss\_end = .;

}

链接脚本关键内容有两部分：段名+地址（作为链接地址的内存地址）

链接脚本的理解：

SECTIONS{} 这个是整个链接脚本

. 点号在链接脚本中代表当前位置。

=等号代表赋值

代码重定位实战：

任务：在SRAM中将代码从0xd00200010重定位到0xd0024000

任务解释：本来代码是运行在0xd0020010的，但是因为一些原因我们又希望代码是在0xd0024000位置运行的，这时候就需要重定位了。

注解：本练习无实际意义，纯粹练习。但是某些情况重定位是必须的，譬如uboot中

思路：

1. 通过链接脚本将代码连接到0xd0024000
2. dnw下载时将bin文件下载到0xd0020010

以上两点保证了代码实际运行在0xd0020010,而却被链接在0xd0024000.从而为重定位奠定需要。

当我们把代码链接地址设置为0xd0024000时，实际隐含意思就是代码将来必须放在0xd0024000位置才能正确执行，如果实际位置不是这个地址就会出错（除非是PIC位置无关码）。所以重定位的作用就是：在PIC执行完之前（在代码中第一句位置有关码执行之前）必须将整个代码搬移到0xd0024000位置去执行，这就是重定位。

3、代码执行时通过代码前段的少量位置无关码将整个代码搬移到0xd0024000

4、使用一个长跳转指令跳转到0xd0024000处的代码继续执行，重定位完成。

长跳转：首先这句代码就是一句跳转指令（ARM中跳转指令就是类似分支指令B、BL等作用指令），跳转指令通过给PC（r15）赋一个新值来完成代码段的跳转执行，长跳转指的是跳转到的地址和当前的地址差异比较大，跳转的范围比较广。

当我们执行完代码重定位后，实际上SRAM中有2份代码镜像（一份是下载到的0xd0020010的，另一份是重定位代码复制到0xd0024000处开头的），这两份内容是完全相同的，仅仅地址不同。重定位之后使用ldr pc, =led\_blink 这句长跳转直接从0xd0020010处代码跳转到0xd0024000开头的那一份代码的led\_blink函数处去执行（实际上此时在SRAM中有2个led\_blink函数镜像，两个都能执行，如果短跳转bl led\_blink则执行的就是0xd00020010开头的这一份，如果长跳转ldr pc , =led\_blink则执行的是0xd0024000开头处的这一份）。这就是长跳转和短跳转的区别。

当链接地址和运行地址相同时，短跳转和长跳转实际效果是一样的；但是不同时，而这就有差异了，这时短跳转就是运行地址的那一份，而长跳转是链接地址那一份。

总结：重定位实际是在运行地址处执行一段位置无关代码PIC，让这段PIC（也就是重定位代码）从运行地址出把整个程序镜像拷贝一份到链接地址处，完了之后使用一句长跳转指令从运行地址处直接跳转到链接地址处去执行同一个函数（led\_blink）。这样就实现了重定位之后的无缝连接。

adr与ldr伪指令的区别

ldr和adr都是伪指令，区别是ldr是长加载、adr是短加载。

重点：adr指令加载符号地址，加载的是运行时地址，ldr指令在加载符号地址时，加载的是链接地址。

重定位（代码拷贝）

重定位就是如下代码中的copy\_loop函数，代码的作用是使用循环结构来逐句复制代码到链接地址。复制的源地址是SRAM的0xd0020010,复制的目标地址是SRAM的0xd0024000,复制长度是bss\_start减去\_start，所以复制的长度就是整个重定位需要重定位的长度，也就是整个程序中代码段+数据段的长度。

bss段（bss段中就是0初始化的全局变量）不需要重定位

清bss段

清楚bss段是为了满足C语言的运行时要求（C语言要求显式初始化为0的全局变量，或者未显式初始化的全局变量的值为0，实际上C语言编译器就是通过清bss段来实现C语言的这个特性的）。一般情况下我们的程序是不需要负责清零bss段的（C语言编译器和链接器会帮我们的程序自动添加一段头程序，这段程序会在我们的main函数之前运行，这段代码就负责清除bss）。但是在我们代码重定位了之后，因为编译器帮我们附加的代码只是帮我们清除了运行地址那一份代码中的bss，而未清除重定位地址开头的那一份代码的bss，所以重定位之后需要自己去清除bss。

长跳转：

清理完bss段之后重定位就结束了，当前的状况是：

1. 当前运行地址还在0xd0020010开头的（重定位前的）那一份代码中运行着
2. 此时SRAM中已经有了2份代码，1份在d0020010开头，另一份在d0024000开头的位置。然后就要长跳转了。

#define WTCON 0x2700000

#define SVC\_STACK 0xd0037D80

.global \_start

\_start:

//第一步：关看门狗（向WTCON的bit5写入0即可）

ldr r0, =WTCON

ldr r1, =0x0

str r1, [r0]

//第二步：设置SVC栈

ldr sp, =SVC\_STACK

//第三步：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加载时就叫短加载。

adr r0, \_start

//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 //重定位的结束地址，重定位只需重定位代码段和数据段

cmp r0, r1 //比较\_start的运行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，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

//用汇编实现的while循环

copy\_loop:

ldr r3, [r0], #4 //源

str r3, [r1], #4 //目的 这两句代码就是完成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（0初始化段）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（没有要初始化的变量），直接下去

beq run\_on\_dram //清楚bss完之后的地址

mov r2, #0

clean\_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），然后r0=r0+4

cmp r0, r1

bne clear\_loop

run\_on\_dram:

//长跳转到led\_blink开始第二阶段

ldr pc, = led\_blink //ldr 实现长跳转

b .

SDRAM引入

1. SDRAM：Synchronized Dynamic Random Access Memory. 同步动态随机存储器
2. DDR：DDR就是DDR SDRAM，是SDRAM的升级版。（DDR:double rate ,双倍速的SDRAM）

DDR有好多代，：DDR1 DDR2 DDR3 DDR4 LPDDR

以上都是内存，都需要初始化。

SDRAM的特性（容量大、价格低、掉电易失、随机读写、总线式访问）

SDRAM/DDR都属于动态内存（相对于静态内存SRAM），都需要先运行一段初始化代码来初始化才能使用类似于SDRAM和SRAM开机上电后就可以直接运行

类似于SDRAM和SRAM的区别，还有NorFlash和NandFlash（硬盘）这两个

正是因为硬件本身的限制，所以才导致启动代码比较怪异、比较复杂。而我们研究裸机是为了研究uboot，在uboot中就充分利用了硬件的各种特性，处理了硬件复杂性

SDRAM数据手册

SDRAM在系统中属于SoC外接设备（外部外设。以前说过随着半导体技术发展，很多东西都逐渐集成到SoC内部去了。现在还长期在外部的一般有：Flash、SDRAM/DDR、网卡芯片如DM9000、音频Codec。现在有一些高度集成的芯片也试图把这几个集成进去，做成正真的单芯片解决方案。）

SDRAM通过地址总线和数据总线接口（总线接口）与SoC通信

汇编初始化SDRAM详解

初始化代码框架介绍

SDRAM初始化使用一个函数sdram\_asm\_init，函数在sdram\_init.S文件中实现，是一个汇编函数。

强调：汇编实现的函数在返回时需要明确使用返回指令（mov pc, lr）

初始化DDR2

1、首先，DDR初始化和SoC（准确说是和SoC中的DDR控制器）有关，也和开发板使用的DDR芯片有关。和开发板设计时DDR的连接方式有关

2、S5PV210的DDR步骤在SoC数据手册，

3、设置IO端口驱动强度：DRAM控制器对应的引脚设置为驱动强度2x，一般参考原厂给的代码

4、DRAM port的时钟设置：主要是开启DLL（dram pll）然后等待锁存

#define WTCON 0x2700000

#define SVC\_STACK 0xd0037D80

.global \_start

\_start:

//第一步：关看门狗（向WTCON的bit5写入0即可）

ldr r0, =WTCON

ldr r1, =0x0

str r1, [r0]

//第二步：设置SVC栈

ldr sp, =SVC\_STACK

//第三步：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：初始化SDRAM

bl sdram\_asm\_init

//第5步：重定位

// adr 指令用于加载\_start当前运行地址 adr加载时就叫短加载。

adr r0, \_start

//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 //重定位的结束地址，重定位只需重定位代码段和数据段

cmp r0, r1 //比较\_start的运行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，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

//用汇编实现的while循环

copy\_loop:

ldr r3, [r0], #4 //源

str r3, [r1], #4 //目的 这两句代码就是完成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（0初始化段）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（没有要初始化的变量），直接下去

beq run\_on\_dram //清楚bss完之后的地址

mov r2, #0

clean\_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），然后r0=r0+4

cmp r0, r1

bne clear\_loop

run\_on\_dram:

//长跳转到led\_blink开始第二阶段

ldr pc, = led\_blink //ldr 实现长跳转

b .